

Technische Informatik

Einleitung und Organisation

Thorsten Thormählen

14. Oktober 2025

Teil 1, Kapitel 1

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Wie viele Computer sind auf diesem Bild?



Thorsten Thormählen 3 / 14

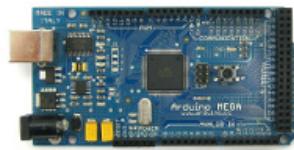
Wie viele Computer sind auf diesem Bild?

Antwort: Das kommt auf die Definition eines "Computers" an.

Eine Digitaluhr kann aus ein paar einfachen Logikbausteinen erstellt werden. Diese Bausteine "berechnen" zwar etwas, sind aber nicht programmierbar.



Programmierbare Kleinstrechner (Micro-Controller) werden in vielen Gegenständen des Alltags verwendet: Waschmaschine, DVD-Player, elektrische Zahnbürste, Staubsauger, etc. In der Regel verändert der Endanwender während der Lebenszeit dieser Geräte das Programm jedoch nicht.



Desktop-Rechner, Laptop, Tablet oder ein modernes Mobiltelefon sind Beispiele für Universalrechner, die je nach Aufgabe neu programmiert werden. Die meisten Anwender programmieren diese nicht selbst, sondern installieren, je nach aktuellem Bedarf, fertige Programme.



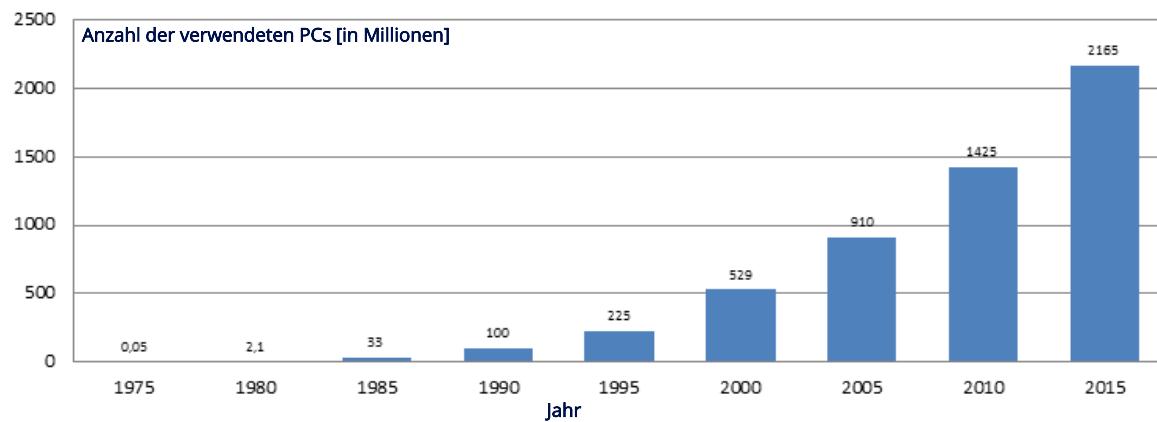
[Bildquelle: [Arduino Mega](#), Flickr User: David Mellis; [iPhone](#), Flickr User: Yutaka Tsutano; [Creative Commons License](#)]

Thorsten Thormählen 4 / 14

Wieviele Computer gibt es auf der Welt?

Thomas Watson, Vorstandsvorsitzender IBM, 1943:
"I think there is a world market for maybe five computers."

Zirka 2 Milliarden Personal Computer (PCs) in 2015



[Quelle: Worldwide PC market, Computer Industry Almanac Inc.]

Thorsten Thormählen 5 / 14

Das ubiquitäre Computerzeitalter

Computer werden immer kleiner und günstiger zu produzieren

Laut ITU haben $\frac{3}{4}$ alle Menschen auf der Welt, die älter als 10 Jahre sind, ein Mobiltelefon

Der Trend geht dahin, dass jeder Mensch mehrere Computer besitzt

Der Computer wird allgegenwärtig:
Das ubiquitäre Computerzeitalter hat begonnen

[Quelle: <https://www.itu.int/en/mediacentre/Pages/PR-2022-11-30-Facts-Figures-2022.aspx>]

Thorsten Thormählen 6 / 14

Was ist "Technische Informatik"?

Die Vorlesung *Technische Informatik* vermittelt Grundlagenwissen über Rechnerstrukturen und die Funktionsweise von mikroelektronischen Schaltungen.

Verständnis über die Hardware des Computers (als Komplement zur Vorlesung *Objektorientierte Programmierung*, die sich mit Software befasst)

Konzepte und Funktionsweise moderner Computersysteme

Wie kann, ausgehend von einzelnen Logikgattern, ein kompletter Mikroprozessor gebaut werden

Hardware-nahe Programmierung (Assembler)

Gewisse Schnittmenge mit der Elektrotechnik

Inhaltsverzeichnis der Vorlesung

1. Einleitung und Historisches
2. Darstellung von Zahlen und Zeichen
3. Boolesche Algebra
4. Normalformen
5. Umwandlung und Minimierung
6. Zeitverhalten und Hazards
7. Logikschaltungen
8. Speicher
9. CPU
10. Maschinensprache und Assembler
11. Prozessorarchitekturen

Vorlesungsfolien

Die Vorlesungsfolien werden jeweils kurz vor der Vorlesung auf folgender Webseite zur Verfügung gestellt:

<https://www.uni-marburg.de/de/fb12/arbeitsgruppen/grafikmultimedia/lehre/ti>

Es gibt zwei Versionen der Folien, intern und extern. Die internen Folien sind umfangreicher und benötigen folgendes Passwort:

Benutzername: "ti"

Passwort: "ws25/26!"

Die Vorlesungsfolien sind als HTML5-Webseiten abgelegt. In Google Chrome kann die Druckfunktion des Browsers verwendet werden, um eine PDF-Datei der Folien zu erzeugen.

Literatur

Allgemein:

H. P. Gumm, M. Sommer: *Einführung in die Informatik*,
10. Auflage; Oldenbourg Wissenschaftsverlag, 2013

N. Grund, M. Sommer: *Klausurvorbereitung zur Einführung in
die Informatik*, Oldenbourg Wissenschaftsverlag, 2013

Technische Informatik:

D. W. Hoffmann: *Grundlagen der Technischen Informatik*,
2. Auflage; Hanser 2009

B. Becker, P. Molitor: *Technische Informatik*, Oldenbourg
Wissenschaftsverlag München, 2008

C. Martin: *Rechnerarchitekturen*, Fachbuchverlag Leipzig,
2001

H. Bähring: *Mikrorechner-Systeme. Mikroprozessoren,
Speicher, Peripherie*, Springer 2005

P. Herrmann: *Rechnerarchitektur. Aufbau, Organisation und
Implementierung*, Vieweg 2000

D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und
Entwurf*, Spektrum 2005



Thorsten Thormählen 10 / 14

Übungen

Verpflichtende ILIAS Tests (Studienleistung / Klausurzulassung)

Es werden wöchentlich ILIAS Tests ausgegeben, die in der Übung am Dienstag vor- und nachbereitet werden

Die Ausgabe und Abgabe der Tests erfolgt immer montags um 12:00 Uhr im [ILIAS](#)
Ausgabe des ersten ILIAS Tests ist am Montag, den 20.10.2025

Der Übungsbetrieb startet in der zweiten Vorlesungswoche (ab Dienstag, den 21.10.2025)

Zum Erhalt der Klausurzulassung müssen 9 von 11 ILIAS Tests bestanden werden
(bestanden bedeutet $\geq 50\%$ der Punkte)

Strikte Umsetzung der Kriterien (**keine Ausnahmen / Sonderbehandlungen**)

Freiwillige Zusatzaufgaben

Zur Vorbereitung auf die Klausur werden in den Übungen weitere freiwillige Beispielaufgaben ausgegeben und vorgestellt, die Sie zusätzlich bearbeiten können. Diese werden nicht kontrolliert und zählen nicht für die Klausurzulassung.

Simulation einer Abschlussklausur (freiwillig)

In der letzten Woche vor der Weihnachtspause wird eine Abschlussklausur simuliert (basierend auf dem bis dahin behandelten Stoff)

Lernzentrum

Das [Lernzentrum](#) befindet sich im Raum 03A01 im Block A der Ebene 3 im Mehrzweckgebäude, Lahnberge

An Wochentagen während der Vorlesungszeit stehen i.d.R. zwischen 14:00 und 18:00 Uhr studentische Hilfskräfte für die Betreuung zur Verfügung

Klausur

Abschlussklausur, Termin: Mi., 25.02.2026, 09:00 - 11:00 Uhr s.t., (H | 05) +5/0030 (HS A), keine Hilfsmittel

Die Anmeldung zur Abschlussklausur ist eine **notwendige Voraussetzung** und erfolgt über [Marvin](#).

Der Anmeldezeitraum für die Prüfung ist typischerweise von Mitte Dezember bis Ende Januar (genaue [Termine](#) beachten, keine Ausnahmen / Sonderbehandlungen)

Wichtig: Für (1) Studienleistung und (2) Prüfung anmelden

Note für Vorlesung entspricht der Note der Abschlussklausur, d.h. Übungspunkte dienen nur zur Klausurzulassung

Bei bestandener Klausur wird das Modul mit Note und 9 ECTS-Punkten verbucht

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) .]

Thorsten Thormählen 14 / 14

Technische Informatik

Historisches

Thorsten Thormählen
14. Oktober 2024
Teil 1, Kapitel 2

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Inhalt

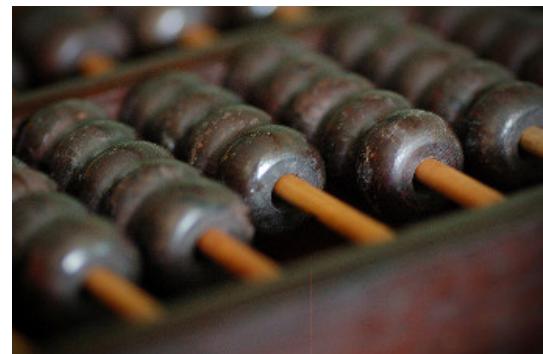
- Mechanische Rechenhilfen
- Mechanische Rechenmaschinen (ab 1600)
- Relais-basierte Computer (ab 1941)
- Elektronenröhren-basierte Computer (ab 1946)
- Transistor-basierte Computer (ab 1955)
- Computer mit integrierten Schaltungen (ab 1965)
- Mikroprozessoren (ab 1971)

Mechanische Rechenhilfe (Abakus), ca. 500 v.u.Z.

Ein Abakus ist ein mechanisches Gerät, welches das Zählen vereinfacht und bei einfachen Berechnungen als Gedächtnisstütze dient

Er besteht aus einem Rahmen und mehreren Stäben auf denen Steine oder Kugeln beweglich aufgereiht sind

Er wird seit Jahrtausenden, u.a. beim Handeln auf den Märkten eingesetzt

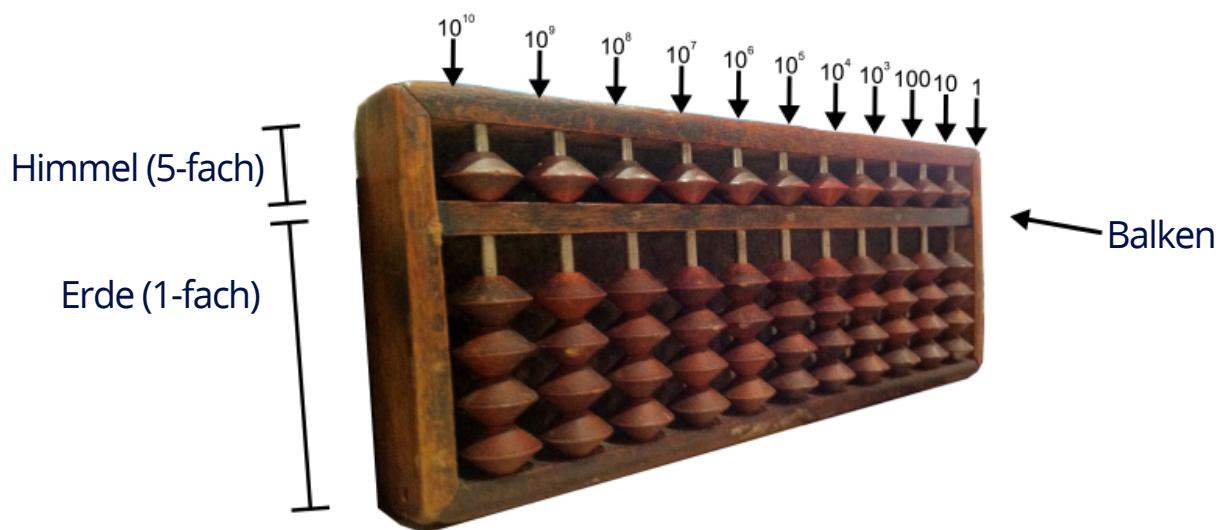


Abakus

[Bildquelle: [Abakus](#), Flickr user: zakulaan, [Creative Commons License](#)]

Thorsten Thormählen 4 / 58

Japanischer Abakus (Soroban), ca. 1600



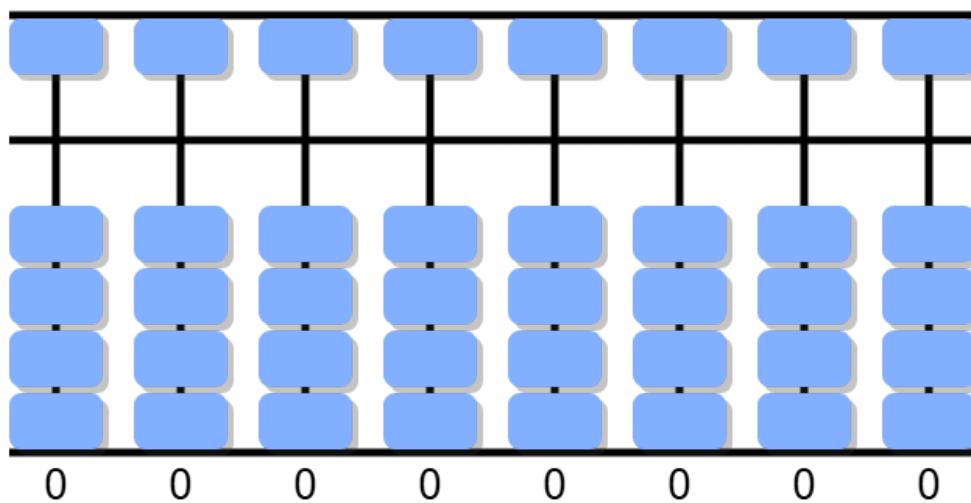
Jeder Stab repräsentiert eine Ziffer von 0 bis 9

Die Kugeln unterhalb des Trennbalken zählen 1-fach und die Kugel oberhalb 5-fach
Kugeln werden gezählt, wenn sie gegen den Trennbalken geschoben sind

[Bildquelle: [Abacus](#), Flickr user: Whity, [Creative Commons License](#), modifiziert]

Thorsten Thormählen 5 / 58

Japanischer Abakus (Soroban)



Dies ist ein interaktiver Soroban Simulator

Durch Klicken auf die Kugeln können diese verschoben werden

Japanischer Abakus (Soroban)

Beispiel: $1823 + 2333 = ???$

Schritt 1: Eingabe von 1823 (siehe Bild)

Schritt 2: Addition von 2333

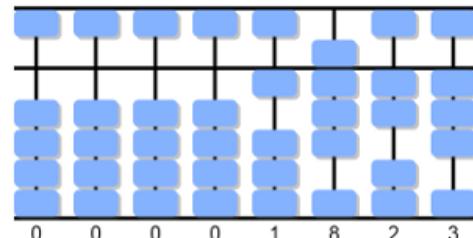
Vorgegangen wird von links nach rechts, nacheinander für jede Ziffer

Addition durch einfaches Hinzufügen von Kugeln

Sind keine passenden Kugeln vorhanden, kann das 5-er oder 10-er Komplement verwendet werden, beispielsweise:

5er-Komplement von 3 ist $(5 - 3) = 2$, d.h. anstatt 3 Kugeln hinzuzufügen, eine Kugel vom Himmel hinzufügen und zwei von der Erde wegnehmen

10er-Komplement von 3 ist $(10 - 3) = 7$, eine Kugel von der nächst höheren Ziffer hinzufügen und insgesamt sieben Zähler von Erde und Himmel wegnehmen



Napiersche Rechenstäbchen, 1617

Im Jahre 1617 beschrieb der schottische Mathematiker John Napier eine Rechenhilfe, mit der eine Multiplikation (bzw. Division) in eine Addition (bzw. Subtraktion) überführt werden kann

Dazu wird das Einmaleins auf Stäbchen notiert

Die Stäbchen werden auf einem Brett platziert, so dass in der obersten Zeile die zu multiplizierende Zahl steht

In den darunter liegenden Zeilen kann die Multiplikation der gelegten Zahl mit den Faktoren 2 bis 9 abgelesen werden

Das Ablesen erfolgt von rechts nach links durch Summation der Zahlen innerhalb der entstehenden Parallelogramme

Bei der Summation ist ein eventueller Übertrag zu berücksichtigen

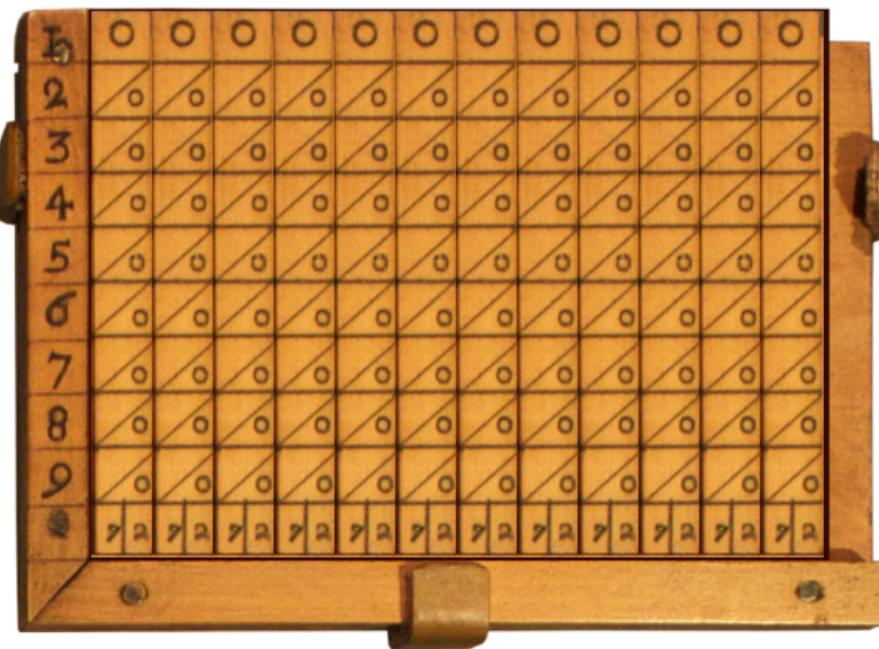


John Napier (1550 - 1617)

[Bildquelle: [John Napier](#), public domain]

Thorsten Thormählen 8 / 58

Napiersche Rechenstäbchen

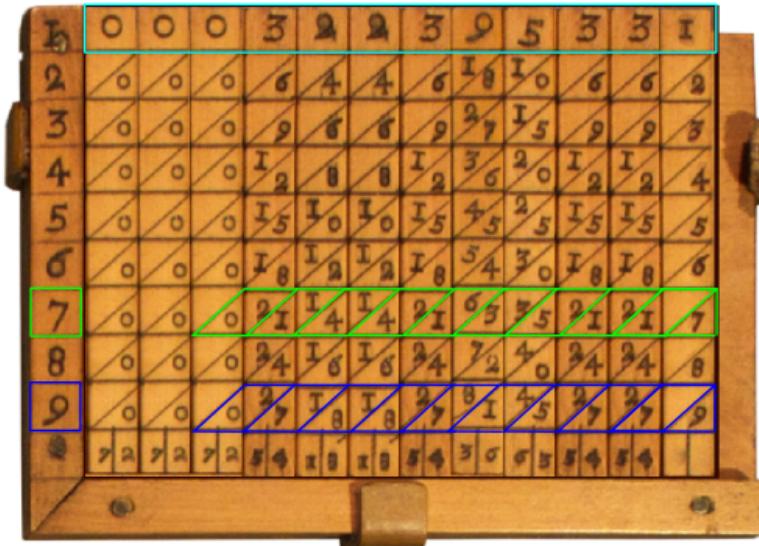


Dies ist ein interaktiver Simulator der Napierschen Rechenstäbchen

Durch Klicken auf die einzelnen Stäbchen kann deren Wertigkeit verändert werden

Napiersche Rechenstäbchen

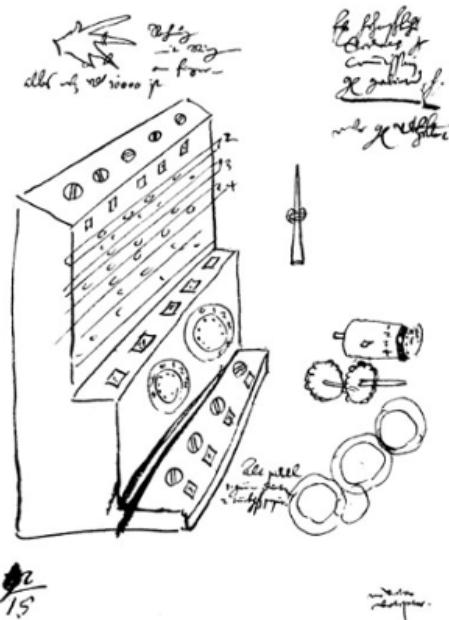
Beispiel:



$$\begin{array}{r} 97 * 322395331 \\ \hline 2256767317 \\ +2901557979 \\ \hline =31272347107 \end{array}$$

Thorsten Thormählen 10 / 58

Mechanische Rechenmaschinen (ab 1600)



Originalzeichnung der "Rechenuhr"

Wilhelm Schickard (1592-1635), Professor in Tübingen, baute im Jahr 1623 eine erste zahnradgetriebene Rechenmaschine

Er schreibt am 20. September 1623 an Johannes Kepler:

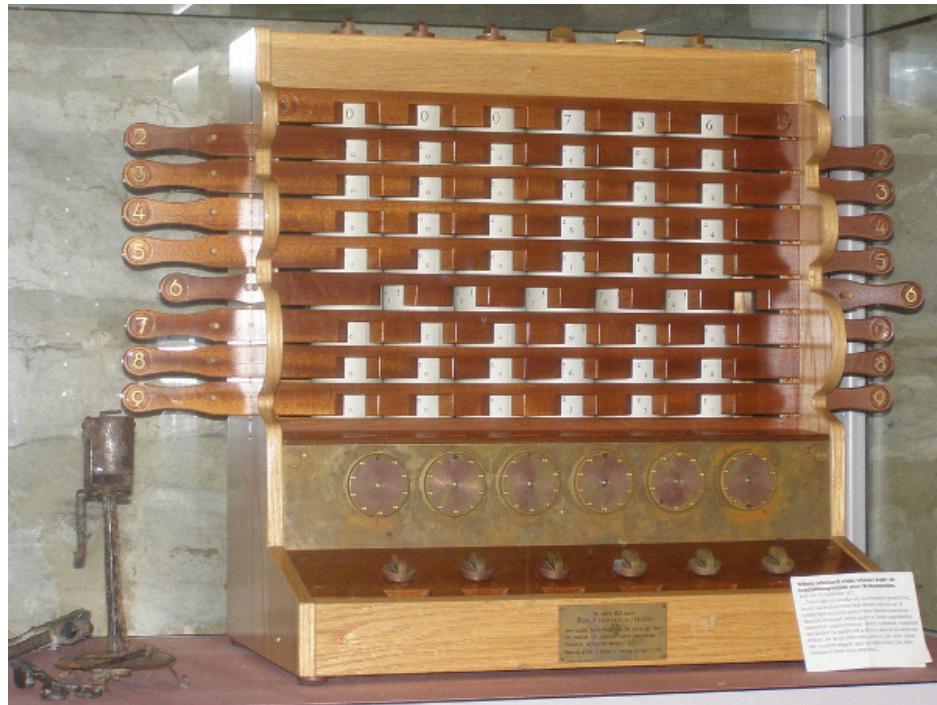
"Dasselbe, was Du auf rechnerischem Weg gemacht hast, habe ich kürzlich mechanisch versucht und eine aus 11 vollständigen und 6 verstümmelten Rädchen bestehende Maschine gebaut, welche gegebene Zahlen im Augenblick automatisch zusammenrechnet: addiert, subtrahiert, multipliziert und dividiert."

Du würdest hell auflachen, wenn Du da wärest und sehen könntest, wie sie, so oft es über einen Zehner oder Hunderter weggeht, die Stellen zur Linken ganz von selbst erhöht oder ihnen beim Subtrahieren etwas wegnimmt."

[Bildquelle: [Originalzeichnung von Wilhelm Schickard](#), public domain]

Thorsten Thormählen 11 / 58

Rechenuhr von Schickard, 1623



Rechenuhr von Schickard (Replika aus dem Jahre 1957)

[Bildquelle: [Wilhelm Schickard machine replica](#), Flickr user: Daniel Sancho, [Creative Commons License](#)]

Thorsten Thormählen 12 / 58

Rechenuhr von Schickard

Die Rechenuhr von Schickard beherrschte die automatische Addition und Subtraktion inkl. dem automatischen Zehnerübertrag

Der Zehnerübertrag wurde durch eine Zahnradkonstruktion erreicht

Zur Multiplikation wurden Napierische Rechenstäbchen (oben) verwendet, deren Werte manuell in das mechanische Additionswerk (unten) übertragen werden mussten

Difference Engine von Charles Babbage, ab 1822

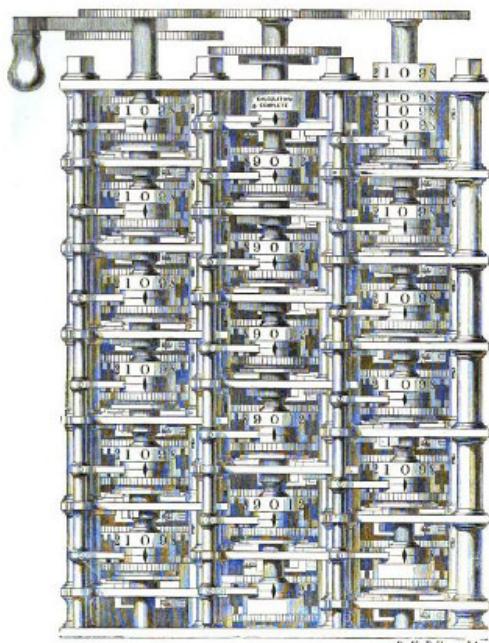
"What are you dreaming about? - I am thinking that all these tables of logarithms might be calculated by machinery."

Beginnend im Jahre 1822 arbeitete der englische Mathematikprofessor Charles Babbage an einer mechanischen Rechenmaschine zur Berechnung von Polynomen mit Newtons Differenzmethode

Die Maschine wurde leider nie von ihm fertig gestellt (trotz großer finanzieller Förderung durch die Britische Regierung bis ins Jahr 1842)

Die Motivation war, dass die damals verwendeten Tabellenwerke häufig fehlerhaft waren, da sie manuell in eintöniger Rechenarbeit erstellt wurden

Um Kopierfehler beim Übertragen der Ergebnisse zu vermeiden, hatte Babbage in einer zweiten Version "Difference Engine No. 2" sogar einen Drucker vorgesehen

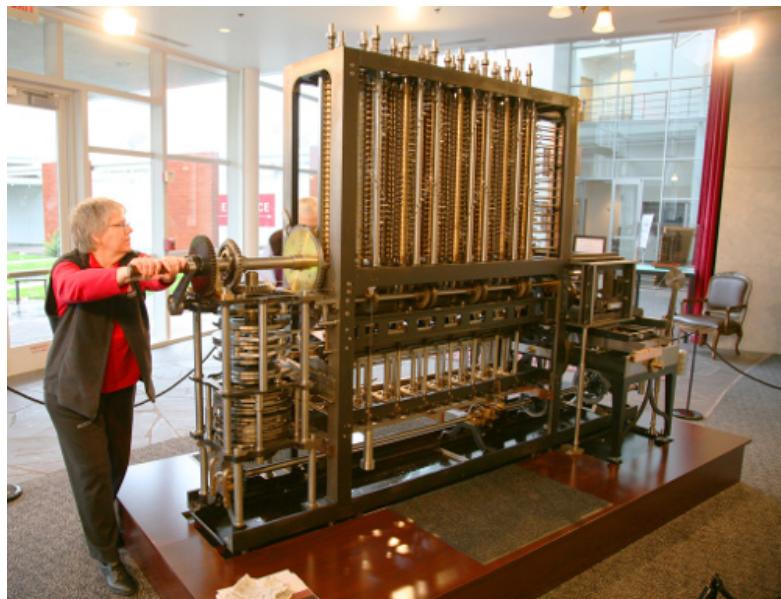


Teil der Difference Engine No. 1

[Quelle: [Harper's new monthly magazine / Volume 30, Issue 175, p.34](#), public domain]

Thorsten Thormählen 14 / 58

Difference Engine von Charles Babbage



Für die "Difference Engine No. 2" wurde im Jahr 1849 von Charles Babbage nur eine Konstruktionszeichnung angefertigt

Erst 1991 wurde sie am Science Museum in London gebaut (etwa fünf Tonnen schwer)

[Bildquelle: [Let the computing begin!](#), Flickr user: Jitze Couperus, [Creative Commons License](#)]

Thorsten Thormählen 15 / 58

Difference Engine von Charles Babbage

Nachbildung der "Difference Engine No. 2" im Computer History Museum, Kalifornien, USA

0:00 / 0:26

[Quelle: [Babbage Engine in Operation](#) by xRez Studio]

Thorsten Thormählen 16 / 58

Analytic Engine von Charles Babbage

Neben seinen Arbeiten an der Difference Engine hat Charles Babbage 1842 ebenfalls eine universell einsetzbare mechanische "Analytic Engine" beschrieben

Diese besaß bereits viele Komponenten eines heutigen Universalrechners (inkl. Trennung von Speicher und Rechenwerk, Schleifen, bedingte Sprünge, etc.)

Er war damit seine Zeit weit voraus

Aufgrund der unvorhersehbaren Kosten wollte die Britische Regierung den Bau jedoch nicht finanzieren



Charles Babbage

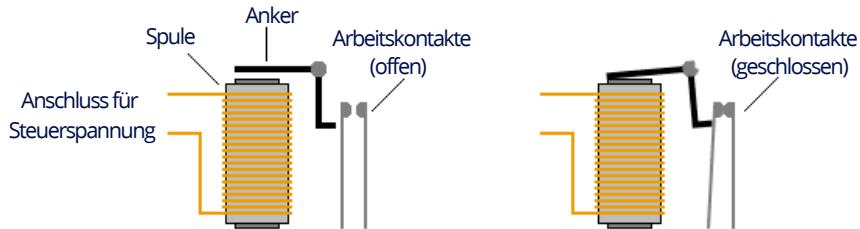
[Bildquelle: [The Late Mr. Babbage](#), The Illustrated London News, 4 November 1871, public domain]

Thorsten Thormählen 17 / 58

Relais-basierte Computer

Durch den Einsatz elektromechanischer Relais können Rechner viel leichter konstruiert werden, als mit reiner Mechanik

Der deutsche Computer-Pionier Konrad Zuse baute 1941 aus 2200 Relais einen voll funktionsfähigen Rechner



Ein Relais ist ein elektrisch gesteuerter Ein-/Aus-Schalter

Kein Strom durch Spule = Anker nicht angezogen = Arbeitskontakt offen

Strom durch Spule = Anker wird durch Magnetfeld angezogen = Arbeitskontakt geschlossen

Es gibt nur zwei Zustände für jeden Schalter: Ein/Aus (bzw. 1/0)

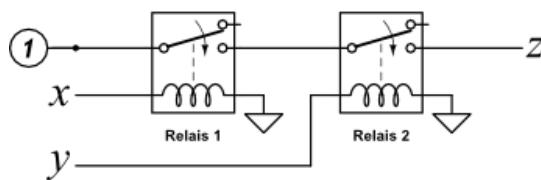
Bis heute verwenden Computer das Binärsystem

[Bildquelle: [Schematische Darstellung eines Relais](#), public domain]

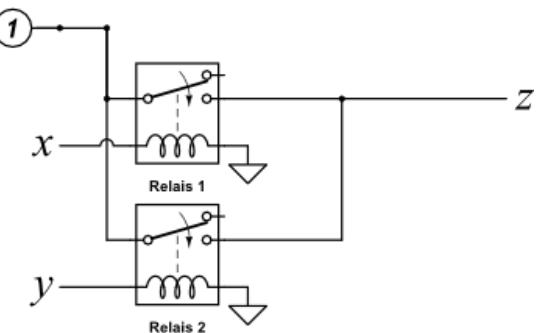
Thorsten Thormählen 18 / 58

Logische Verknüpfungen mittels Relais-Schaltungen

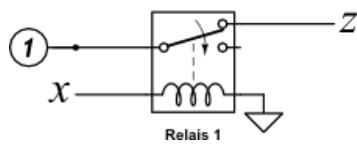
AND-Verknüpfung



OR-Verknüpfung



NOT-Verknüpfung



Die Z3 von Konrad Zuse, 1941

Basierend auf seinen Erfahrungen mit der mechanisch arbeitenden Z1 (1935 bis 1938), baut Konrad Zuse 1939 ein Versuchsmodell eines Relais-basierten Computers: die Z2

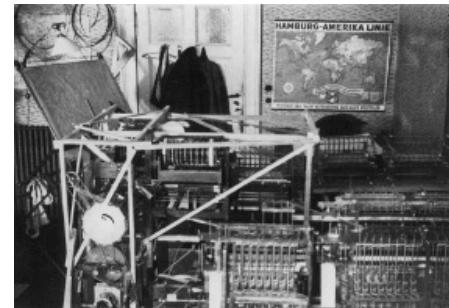
In 1941 baute er die Z3, der erste Relais-basierte funktionstüchtige Rechner mit einer Taktrate von 5,3 Hertz

Die Relaistechnik wurde zu diesem Zeitpunkt schon länger in der Telekommunikation eingesetzt

Die Z3 besaß einen Speicher (1600 Relais) sowie Steuer- und Rechenwerk (600 Relais)

Es gab 9 Befehle: Eingabe, Ausgabe, Speicher lesen, Speicher schreiben, Multiplizieren, Dividieren, Wurzelziehen, Addieren, Subtrahieren. Die Befehle konnten direkt über ein Bedienfeld mit Tastatur und numerischer Anzeige oder programmatisch mittels Lochstreifen übergeben werden

Vorführung der Z3 im Deutschen Museum



Die Z1 im Wohnzimmer der Eltern



Zuse mit Z3-Nachbau

[Bildquelle: [Deutsches Museum](#), Freizur Veröffentlichung nur mit diesem Vermerk]

Thorsten Thormählen 20 / 58

Die Z3 von Konrad Zuse, 1941

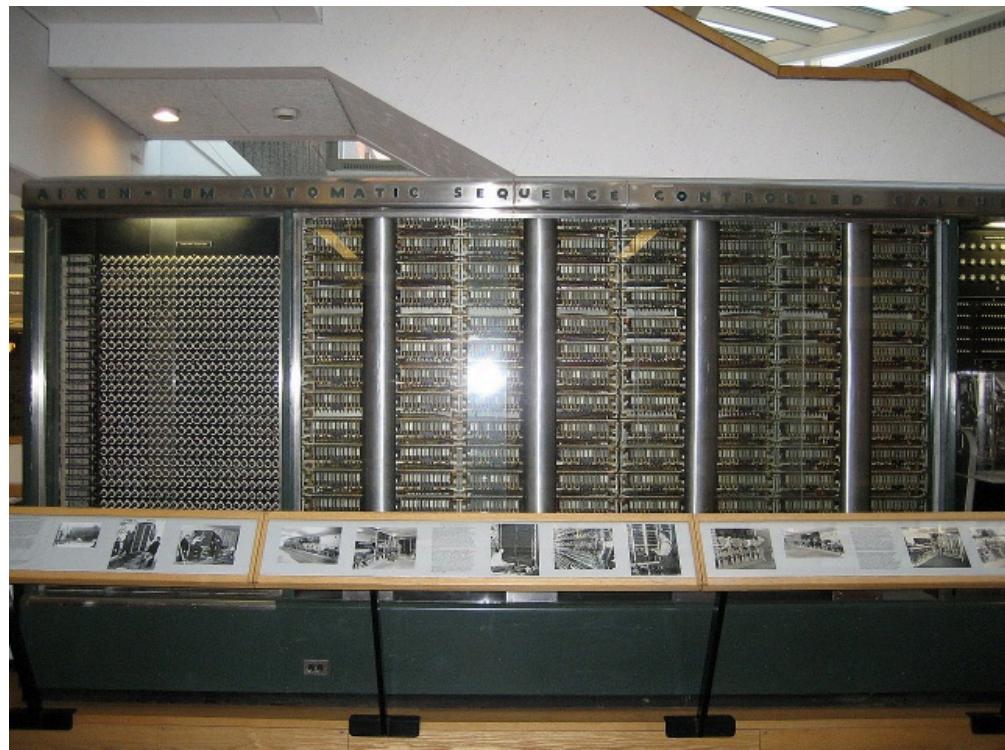
[Vorführung der Z3 im Deutschen Museum](#)

0:00 / 5:11

[Quelle:[Deutsches Museum](#)]

Thorsten Thormählen 21 / 58

Harvard Mark I, 1944



[Bildquelle: [HarvardMarkI](#) , public domain]

Thorsten Thormählen 22 / 58

Harvard Mark I, 1944

Die Harvard Mark I war ein Relais-basierter Rechner, der vom Harvard Professor Howard H. Aiken entworfen wurde

Aiken begann 1939 mit der Arbeit an dem "Automatic Sequence Controlled Calculator", der 1944 mit Unterstützung von IBM fertiggestellt wurde und den Namen "Harvard Mark I" erhielt

Die Befehle konnten programmatisch mittels Lochstreifen übergeben werden

Der Rechner hatte eine Länge von 15,5 Metern und war fast 5 Tonnen schwer

IBM hatte aufgrund ihrer Buchungs- und Tabelliermaschinen bereits viel Erfahrungen mit der Relaistechnik und Lochkartengeräten

Aiken entwarf seinen Rechner zeitgleich mit Zuse und kannte dessen Arbeiten aufgrund des fehlenden Informatikaustausches während des zweiten Weltkrieges nicht

Im Gegensatz zu Zuse verwendete Aiken Dezimalarithmetik

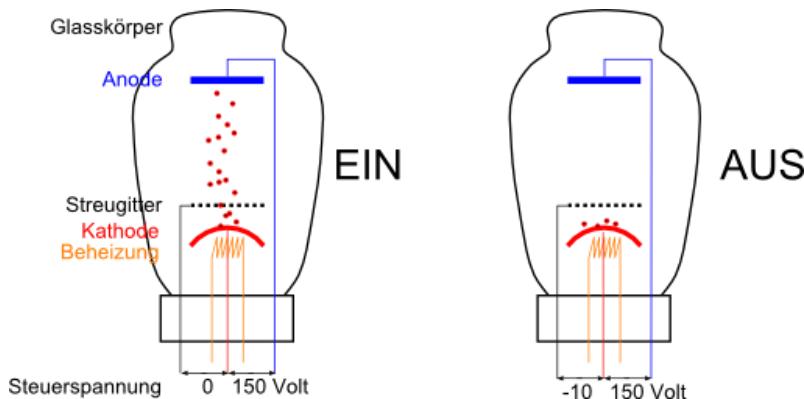


[Bildquelle:[Harvard Mark I Computer](#) by Rocky Acosta, [Creative Commons License](#)]

Thorsten Thormählen 23 / 58

Elektronenröhren-basierte Computer

Elektronenröhren können ebenfalls als Schalter verwendet werden (Triode). Sie erreichen ca. 1000 bis 2000-mal schnellere Schaltzeiten als die besten Relais.



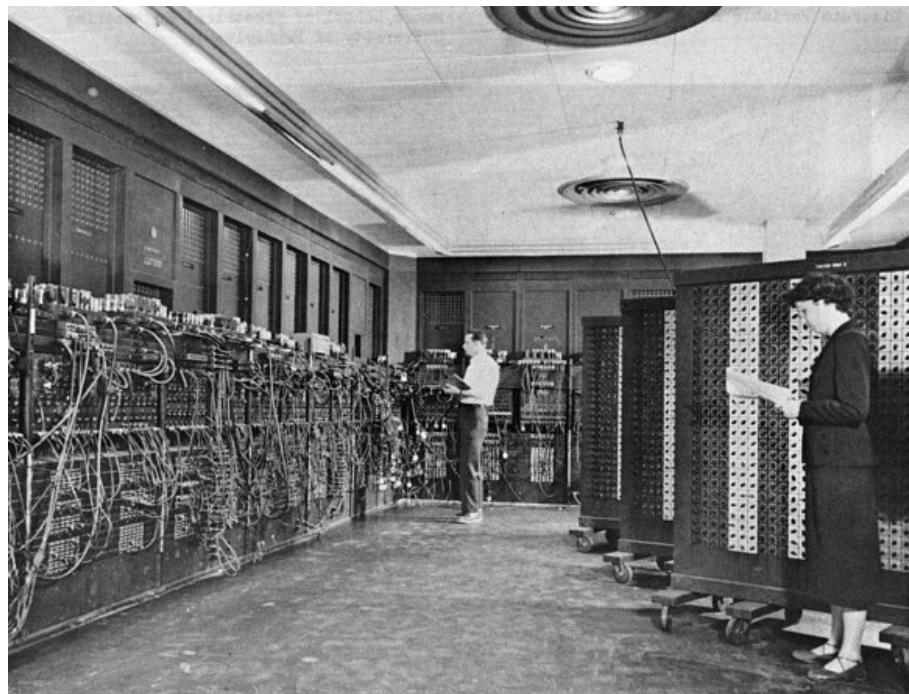
Im Vakuum treten aus der geheizten Kathode Elektronen aus und werden von der starken positiven Ladung der Anode angezogen

Keine Spannung an Streugitter = Strom fließt

Negative Spannung an Streugitter = kein Strom fließt

Damit gibt es (genau wie beim Relais) wieder zwei Zustände: Ein/Aus (bzw. 1/0)

ENIAC, der erste rein elektronische Rechner, 1946



"Electronic Numerical Integrator and Computer" (ENIAC)

[Bildquelle: [Eniac](#), U.S. Army Photo, public domain]

Thorsten Thormählen 25 / 58

ENIAC, 1946

ENIAC wurde ab 1942 von John W. Mauchly und Persper Eckert an der Moore School of Electrical Engineering, University of Pennsylvania, entwickelt, und 1946 öffentlich vorgestellt

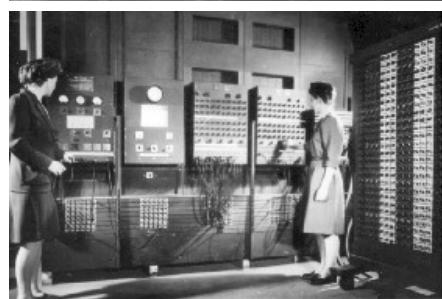
Sie benötigte einen ganzen Raum, wog ca. 30 Tonnen und bestand aus 17.468 Elektronenröhren, 7.200 Dioden, 1.500 Relais, 70.000 Widerständen und 10.000 Kondensatoren

Durch die Elektronenröhren war die ENIAC schneller (ca. 1000 Hz) als Relais-basierte Computer, hatte jedoch folgende Nachteile:

Sehr hoher Stromverbrauchs (174.000 Watt)

Die Röhren gingen schnell kaputt (Lebensdauer ca. 2 Jahre; d.h. im Durchschnitt war jede Stunde eine Röhre defekt). Durch Modulbauweise war es möglich, schnell ganze Module auszutauschen.

Die Programmierung erfolgte durch neue Verkabelung, daher war die ENIAC nicht sehr flexibel. Hauptaufgabe war die Berechnung ballistischer Tabellen für die U.S. Army



[Bildquelle: [Eniac](#), U.S. Army Photos, public domain]

Thorsten Thormählen 26 / 58

UNIVAC I, 1951

Eckert und Mauchly gründeten die "Eckert-Mauchly Computer Corporation", um ihre Erfindung zu kommerzialisieren

Nachfolge-Modell des ENIAC, war der UNIVAC I (UNIVersal Automatic Computer)

Der UNIVAC war Speicher-programmierbar und konnte Daten auf einem Magnetband speichern (12.800 Zeichen pro Sekunde)

Nach Übernahme durch "Remington Rand" wurden 46 Systeme verkauft



Rechenanlage UNIVAC I im Deutschen Museum, München

[Bildquelle: [Univac 1](#), by Jordi Marsol, [Creative Commons License](#)]

Thorsten Thormählen 27 / 58

IAS (von Neumann-Rechner), 1952

Am Institute for Advanced Study (IAS) in Princeton entwickelte John von Neumann einen auf Elektronikröhren basierten Rechner, der im Gegensatz zur ENIAC im Binärsystem arbeitete

Von Neumanns hatte bereits 1944 gemeinsam mit Eckert and Mauchly (zunächst theoretisch) ein Konzept für einen Universal-Rechner beschrieben.



John von Neumann

[[Bildquelle](#): John von Neumann, public domain]

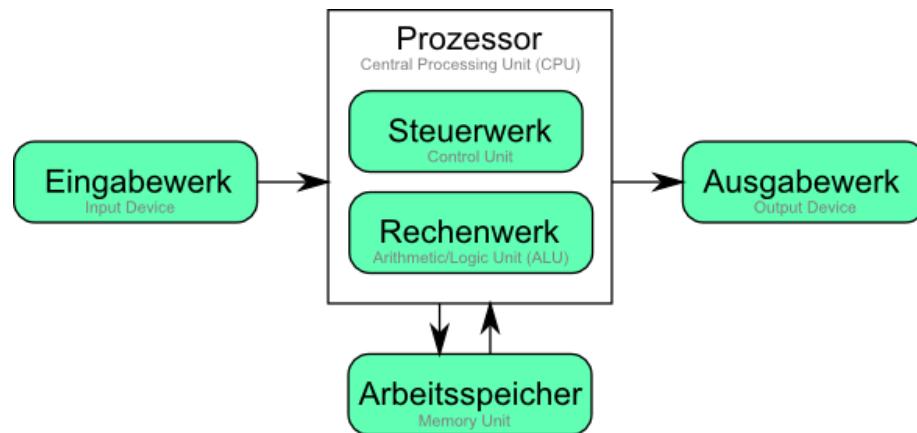
Thorsten Thormählen 28 / 58

Von Neumannsches Rechnerkonzept

Das von Neumannsche Rechnerkonzept besagt u.a.:

Die Struktur des Rechners ist unabhängig vom zu bearbeitenden Problem

Es gibt 5 Funktionseinheiten: Steuerwerk, Rechenwerk, Speicher, Eingabewerk und Ausgabewerk.



Von Neumannsches Rechnerkonzept

Befehle und Daten werden binär codiert und in einem gemeinsamen Speicher gehalten

Der Speicher ist in Zellen gleicher Größe geteilt, die mit fortlaufenden Nummern adressierbar sind

Befehle, die im Speicher hintereinander liegen, werden nacheinander abgearbeitet

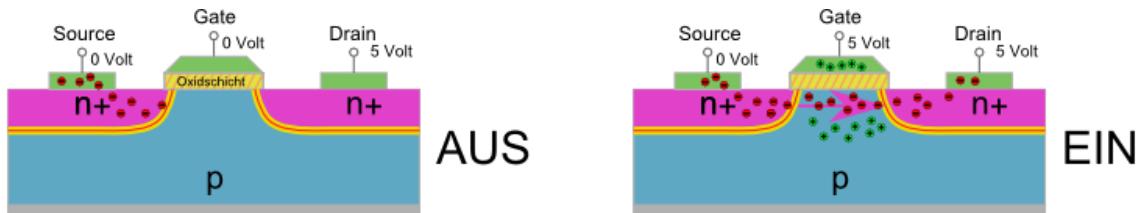
Es gibt Sprungbefehle, um den Ablauf zu ändern

Der Speicherinhalt (Daten und Befehle) kann durch die Maschine modifiziert werden

Das Konzept ist immer noch beliebt, da die Programmierung durch den streng sequentiellen Ablauf einfach ist (nichts passiert parallel)

Transistor-basierte Computer

Elektronenröhren waren unzuverlässig und wurden seit Anfang der Fünfzigerjahre durch ein neues elektrisches Bauteil ersetzt: **der Transistor**



Der hier gezeigte Feldeffekttransistor besteht aus p- und n-dotiertem Halbleitermaterial. Es gibt drei Kontakte aus Metall: Source, Drain und Gate.

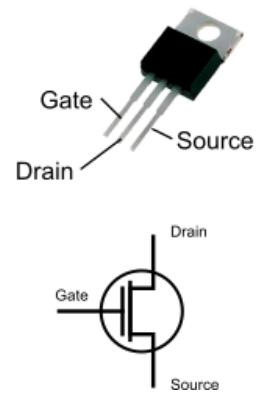
Der Strom zwischen Source und Drain wird über die Spannung zwischen Gate und Source gesteuert

0 Volt am Gate = die Elektronen können das p-dotierte Gebiet nicht überwinden = kein Strom fließt

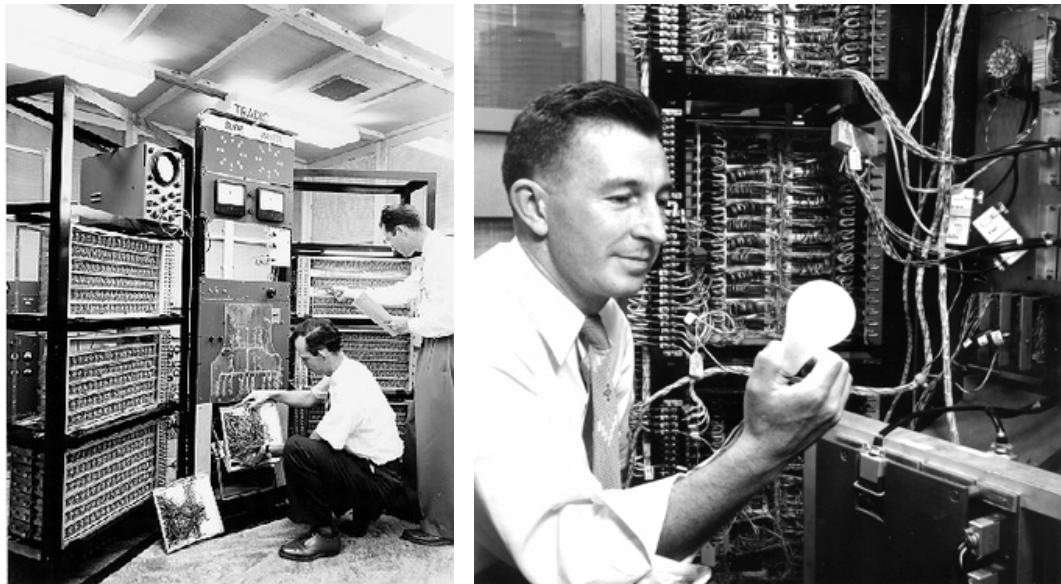
5 Volt am Gate = Elektronen reichern sich unterhalb des Gate an (n-Kanal) = Elektronen fließen durch Kanal = Strom fließt

Damit gibt es wieder zwei Zustände: Ein/Aus (bzw. 1/0)

Feldeffekttransistor



TRADIC, 1955



TRADIC (TRansistorized Airborne Digital Computer)

[Bildquelle:[Tradic](#) , [Tradic Detail](#)]

Thorsten Thormählen 32 / 58

TRADIC, 1955

TRADIC (TRansistorized Airborne Digital Computer) wurde von AT&T Bell Labs für die US Air Force entwickelt und 1955 in Betrieb genommen

Es wurden keine Elektronenröhren mehr verbaut, sondern ca. 700-800 einzelne Transistoren

Neben der Ausfallwahrscheinlichkeit und der Baugröße reduzierte sich auch die Leistungsaufnahme auf ca. 100 Watt

Die Rechengeschwindigkeit betrug bereits ca. eine Million logische Operationen pro Sekunde (1 MHz)

IBM 1401, 1959

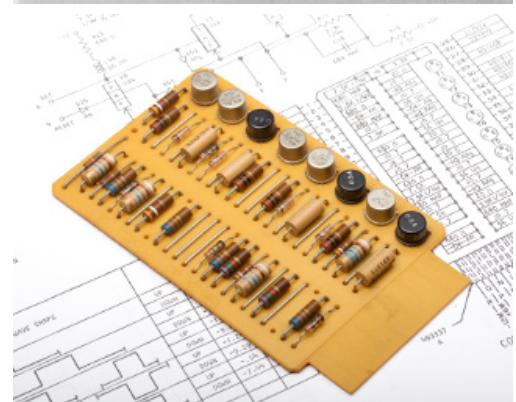
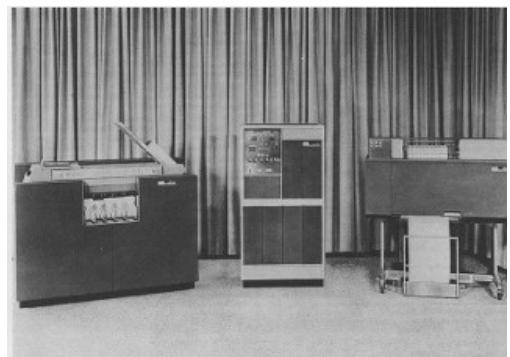
IBM 1401 war ein Großrechner, der für die Verarbeitung von Massendaten (Volkszählung, Buchhaltung, Kundendaten, etc.) in großen Firmen oder staatlichen Einrichtungen gedacht war

Insgesamt wurde mehr als 10.000 Stück gebaut

Die Logikschaltungen wurde aus einzelnen Platinen mit bedrahteten Bauteilen (Transistoren, Kondensatoren und Dioden) aufgebaut

Die Basisausstattung hatte einen Lochkartenleser (Bild links) und einen Drucker (rechts). Es konnten mehrere Magnetbandeinheiten angeschlossen werden (Übertragungsgeschwindigkeit 41.000 Zeichen pro Sekunde)

Die IBM 1401 konnte u.a. mit der höheren Programmiersprache FORTRAN programmiert werden



[Bildquelle: [Basic IBM 1401 system](#) , public domain, [IBM Standard Modular System card](#), by Marcin Wichařy, [Creative Commons License](#)]

Thorsten Thormählen 34 / 58

Computer mit integrierten Schaltungen, ab 1965

Anstatt einzelne Transistoren als diskretes Bauelement herzustellen, werden viele Transistoren auf einem Stück Halbleitermaterial integriert (engl. Integrated Circuit, IC)



Bereits 1958 gelang es Jack Kibly von Texas Instruments einen ersten IC zu erzeugen

Einige Jahre später waren die ICs marktreif. Das erste Computersystem, in dem sie kommerziell zum Einsatz kamen, waren die IBM /360 Rechner.

IBM /360, 1965

Einführung eines Familienkonzepts: Alle Rechner einer Familie sind kompatibel

Idee: Alle Maschinen haben den selben Maschinenbefehlssatz. Die Implementierung auf unterschiedlichen physikalischen Bausteinen erfolgt durch **Mikroprogrammierung**

Ein Mikroprogramm gibt an, wie bei der Ausführung eines bestimmten Maschinenbefehls (z.B. Addition) die einzelnen Logikbausteine angesteuert werden müssen

IBM /360 Modell 85 verwendete als erstes kommerzielles System einen "Cache" (schneller lokaler Speicher, der eine Kopie der Hauptspeicher-Daten vorhält)



[Bildquelle: [IBM System 360/20 computer, Deutsches Museum](#), by Ben Franske, [Creative Commons License](#)]

Thorsten Thormählen 36 / 58

IBM /360, 1965

0:00 / 0:43

Werbefilm zur Modellreihe /360 von IBM

[Quelle: [IBMMainframesvideos](#)]

Thorsten Thormählen 37 / 58

Intel 4004: der erste Mikroprozessor von Intel, 1971

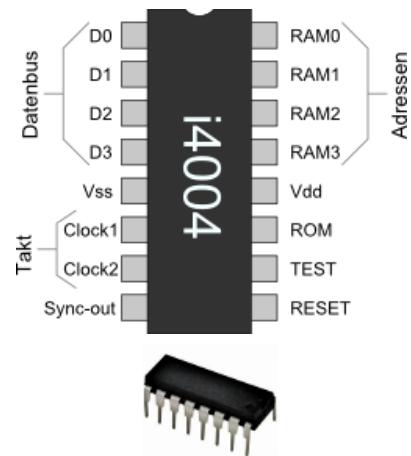
Intel (Integrated Electronics Corporation) wurde im Jahr 1968 von Gordon Moore und Robert Noyce gegründet

Im November 1971 kam Intel's erster Mikroprozessor, der 4004, auf den Markt

Der IC wurde mit einer Strukturbreite von 10 Mikrometern gefertigt. Er hatte 2250 Transistoren und arbeitete zunächst mit einer Taktrate von 108 KHz

Der 4004 hat einen Datenbus von 4 Bit, d.h. pro Bustakt können nur 4 Ein/Aus Zustände (= 4 Bit) gelesen werden

Ein Befehl war jedoch als eine Folge von 8 Einsen und Nullen kodiert, z.B. 01101000. Daher arbeitete der Datenbus doppelt so schnell, um einen Befehl pro Prozessortakt lesen zu können



Intel 8080, 1974

Im April 1974 stellt Intel den 8080 vor, der von vielen als der erste wirklich verwendbare Mikroprozessor bezeichnet wird

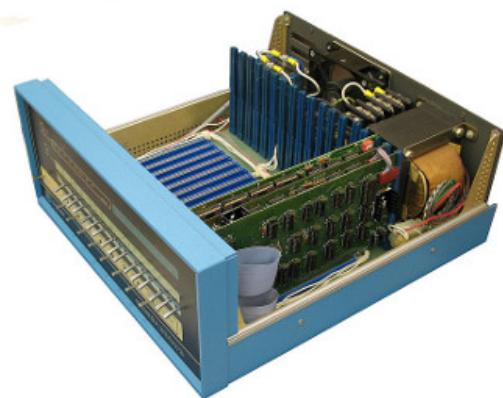
Der Intel 8080 und sein Vorgänger 8008 (1972) waren 8-Bit Computer, d.h dass 8 Bit (=1 Byte) innerhalb eines Taktes verarbeitet werden können

Der 8080 hat einen 8-Bit Datenbus und 16-Bit Adressbus. Damit war es möglich, (2^{16}) = 65536 Bytes im externen Speicher zu adressieren

Der IC wurde mit einer Strukturbreite von 6 Mikrometern gefertigt. Er hatte 4500 Transistoren und arbeitete mit einer Taktrate von 2 MHz

Basierend auf dem 8080 konnten sich Bastler ab 1975 den kostengünstigen Bausatz-Computer Altair 8800 bestellen

Damit hielt der Computer Einzug in den Privatbesitz von Technikbegeisterten: Der PC (Personal Computer) war geboren



Altair 8800 mit Intel 8080 CPU

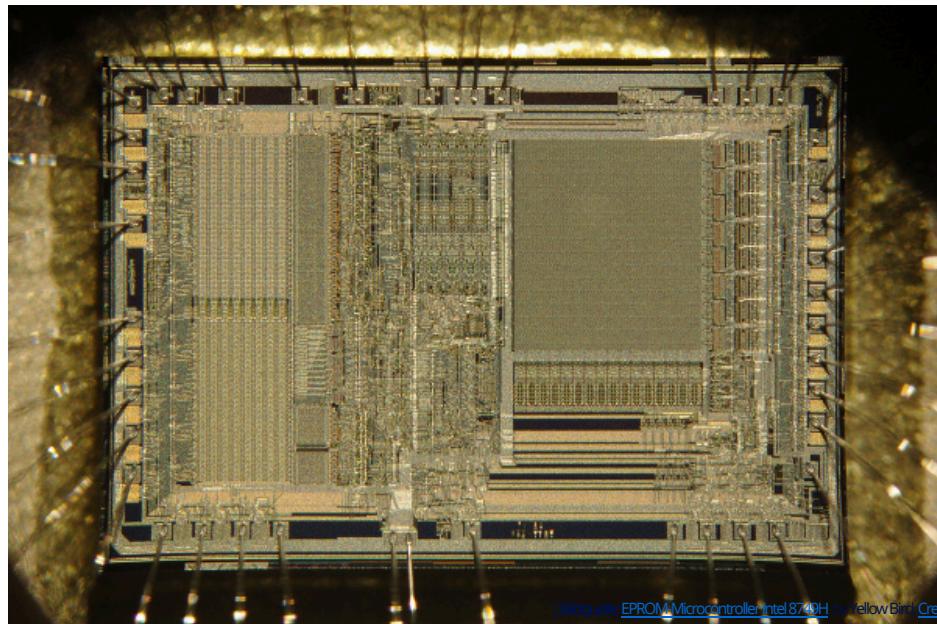
[Bildquelle: [MITS Altair 8800](#), public domain]

Thorsten Thormählen 39 / 58

Intel 8748 Mikrocontroller, 1976

Ab 1976 gelang es Intel, einen 8-Bit Micro-Controller, d.h einen vollständigen Rechner (Prozessor, Speicher und Ein-/Ausgabeeinheiten), auf einen IC zu integrieren

Damit startete Intel die Micro-Controller-Familie MCS-48 (Bild zeigt Intel 8749)



[Bildquelle: EPROM-Mikrocontroller Intel 8749H by yellowBird; Creative Commons License]

Thorsten Thormählen 40 / 58

Apple II, 1977

Der Apple II war vergleichsweise günstig und war als erster Personal Computer weit verbreitet. Der erste große kommerzieller Erfolg für die Gründer von Apple: Steve Wozniak und Steve Jobs

Interessant ist, dass die Baupläne des Apple II veröffentlicht wurden, d.h. andere Hersteller konnten ihn erweitern - aber auch nachbauen

Neben Text konnte der Apple II bereits Farbgrafiken darstellen: Entweder 15 Farben mit niedriger Auflösung (40×48 Pixel) oder 6 Farben mit hoher Auflösung (280×192 Pixel)

Selbst ausprobieren: [Apple II Emulator](#)



[Bildquelle: [Apple II](#), by Marcin Wichary, [Creative Commons License](#)]

Thorsten Thormählen 41 / 58

Intel 8086, 1978

Der 8086 ist ein 16-Bit Prozessor von Intel

Der Intel 8086 wurde mit einer Strukturbreite von 3 Mikrometern gefertigt, hatte 29000 Transistoren und arbeitete mit einer Taktrate von 5 MHz bis 10 MHz

Die nach dem 8086 benannte x86-Mikroprozessor-Architektur wird später zum Industrie-Standard, vor allem, weil IBM eine spätere Variante des Prozessors, den Intel 8088, ab 1981 in ihren PCs verbaute

Der IBM-PC wurde sehr erfolgreich. Auch gab es viele Nachahmer, die kompatible PCs mit den gleichen Komponenten bauten

Somit wurde die x86-Mikroprozessor-Architektur sehr verbreitet



IBM 5150 mit Intel 8086 CPU

[Bildquelle: [IBM PC](#) by Marcin Wicha, [Creative Commons License](#)]

Thorsten Thormählen 42 / 58

Intel-x86-CPU-Familie

Name	Datum	Taktrate	Anzahl Transistoren	adressierbarer Speicher	Anmerkungen
8086	6/1978	5-10 MHz	29000	1 MiB	16-Bit-CPU
80286	2/1982	6-20 MHz	134000	16 MiB	
80386	10/1985	16-33 MHz	275000	4 GiB	32-Bit-CPU
80486	4/1989	25-50 MHz	1,2 M	4 GiB	8K-Cache
Pentium	1993	60-233 MHz	3,1 M	4 GiB	zwei Pipelines
Pentium Pro	3/1995	150-200 MHz	5,5 M	4 GiB	zwei Cache-Ebenen
Pentium II	5/1997	233-400 MHz	7,5 M	4 GiB	MMX (SIMD)
Pentium III 2	1999	450-600 MHz	9,5 M	4 GiB	SSE (SIMD)
Pentium 4	2/2000	1,3-2,0 GHz	42 M	4 GiB	drei Cache-Ebenen
Pentium 4 Prescott	2/2004	3,8 GHz	125 M	4 GiB	64-Bit-CPU
Core 2 Duo	7/2006	2*(1,8-3,2) GHz	bis zu 410 M	4 GiB	2-Kern Prozessor
Core 2 Quad	1/2007	4*(2,5-3,2) GHz	bis zu 820 M	64 GiB	4-Kern Prozessor
Core i7 (1st Gen)	9/2009	6* (2,8-3,9) GHz	ca. 1 G	16 TiB	6-Kern Prozessor

[Quelle:[Wikipedia](#)]

Thorsten Thormählen 43 / 58

Moore's Law

"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. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least ten years" - Gordon E. Moore, 1965

1975 korrigierte Moore seine Aussage und sagte eine Verdoppelung der Anzahl der Transistoren auf einem Mikrochip alle zwei Jahre voraus (teilweise wird in der Literatur auch von einer Verdopplung alle 18 Monate gesprochen)

Auf die genaue Zeitspanne kommt es bei seiner Aussage jedoch gar nicht an, wichtig ist, dass die Anzahl der Transistoren exponentiell wächst

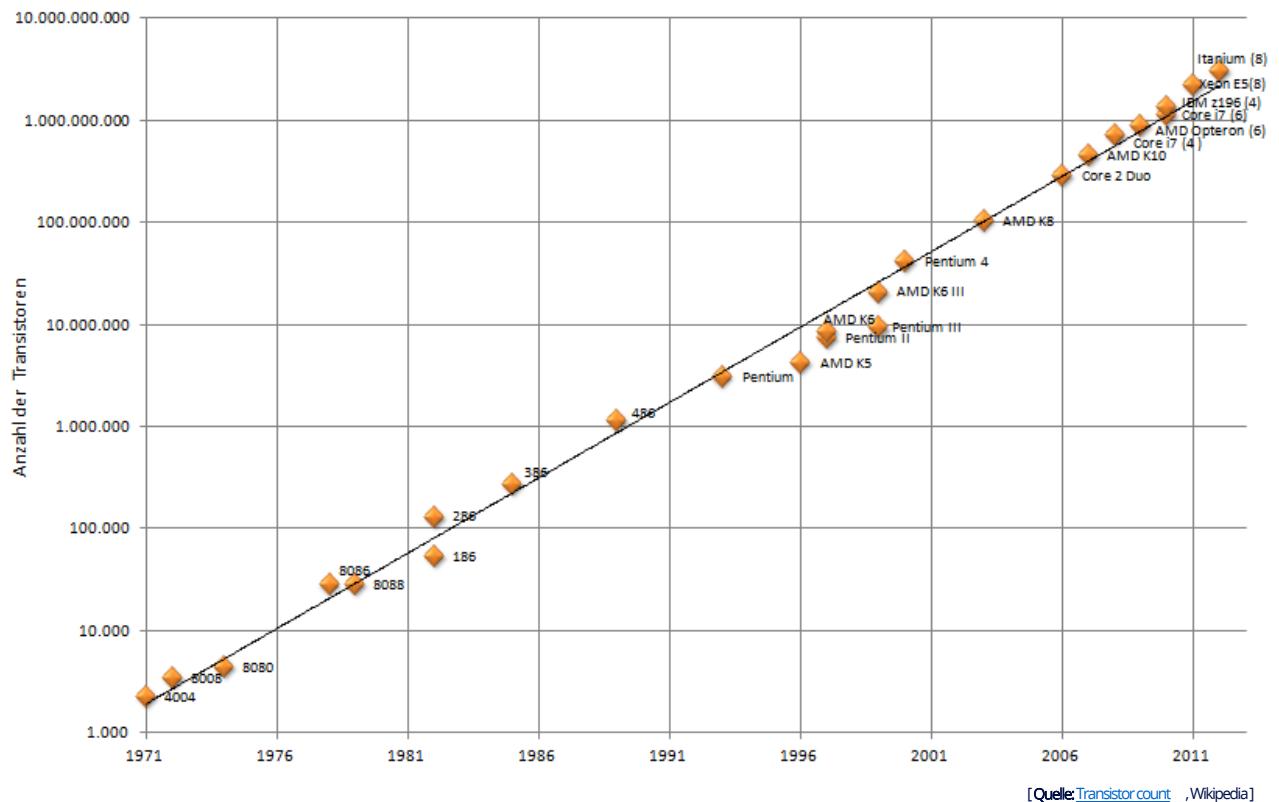
Die Mikrochip-Hersteller versuchen dieses exponentielle Wachstum aufrecht erhalten (selbsterfüllende Prophezeiung), obwohl schon häufig mit dem Ende von Moore's Law aufgrund von technologischen Limitationen gerechnet wurde

Seit 2006 kann beobachtet werden, dass die Taktraten nicht mehr stark steigen, sondern mehrere Recheneinheiten (Cores) auf einem Chip platziert werden

[Quelle: Gordon E. Moore, "Cramming More Components onto Integrated Circuits," Electronics, pp. 114-117, April 19, 1965.]

Thorsten Thormählen 44 / 58

Anzahl an Transistoren pro Mikroprozessor IC



Thorsten Thormählen 45 / 58

Commodore 64, 1982

Commodore 64 (C64) war ein 8-Bit Computer von "Commodore International", der ca. 17 Millionen Mal verkauft wurde

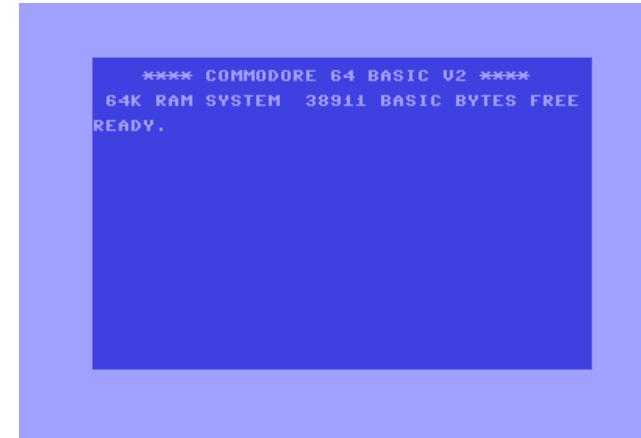
Er war sehr leicht zu bedienen und stand in den 80er-Jahren u.a. als "Spiel-Computer" in vielen Kinderzimmern

Neben Assembler-Programmierung konnte der C64 mit einem BASIC-Interpreter programmiert werden, der beim Start aus dem ROM (Read-Only-Memory) gelesen wurde

Selbst ausprobieren:

[C64 Online Emulator](#)

[C64 JavaScript Emulator](#)



[Bildquelle: [Commodore64](#), public domain]

Thorsten Thormählen 46 / 58

Apple Macintosh, 1984

1984 präsentierte Apple den Macintosh mit Maus und graphischer Benutzeroberfläche



[Bildquelle: [Macintosh](#), by Marcin Wichař, [Creative Commons License](#)]

Thorsten Thormählen 47 / 58

Die Erfolgsgeschichte von Microsoft und Apple

0:00 / 28:19

ZDF doku: Eine kurze Geschichte des PCs

Thorsten Thormählen 48 / 58

MIPS R2000, 1986

Der MIPS R2000 ist ein Vertreter der RISC-Architektur

Da die Maschinenbefehle der etablierten Architekturen immer umfangreicher und komplexer wurden kam in den 80er Jahren eine Gegenbewegung auf

Computer mit den bisherigen Befehlssätzen wurden bezeichnet als:
CISC (Complex Instruction Set Computer)

Eine neue Computerarchitektur wurde vorgeschlagen:
RISC (Reduced Instruction Set Computer)

Die einfachen RISC-Befehle sind schneller auszuführen und benötigen jeweils ungefähr die gleiche Zeit

Die einfachen Befehle verwenden dedizierte Hardware und ersetzen die bei CISC Prozessoren übliche Mikroprogrammierung

Die RISC-Prozessoren können daher schneller getaktet werden und die Fließbandverarbeitung von Befehlssequenzen (Pipelining) wird effizienter

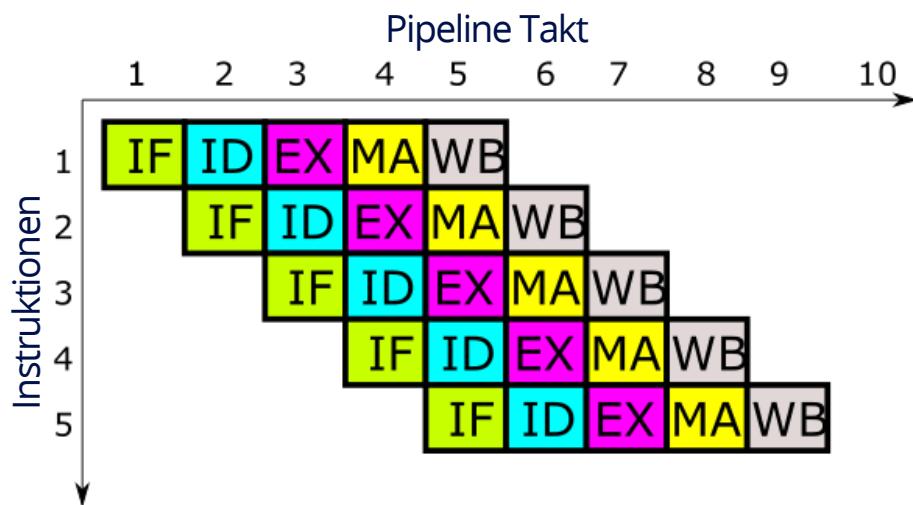
Motorola 68000, PowerPC (Apple), ARM (Smartphone) sind weitere Beispiele für RISC-Processoren

Fließbandverarbeitung (Pipelining)

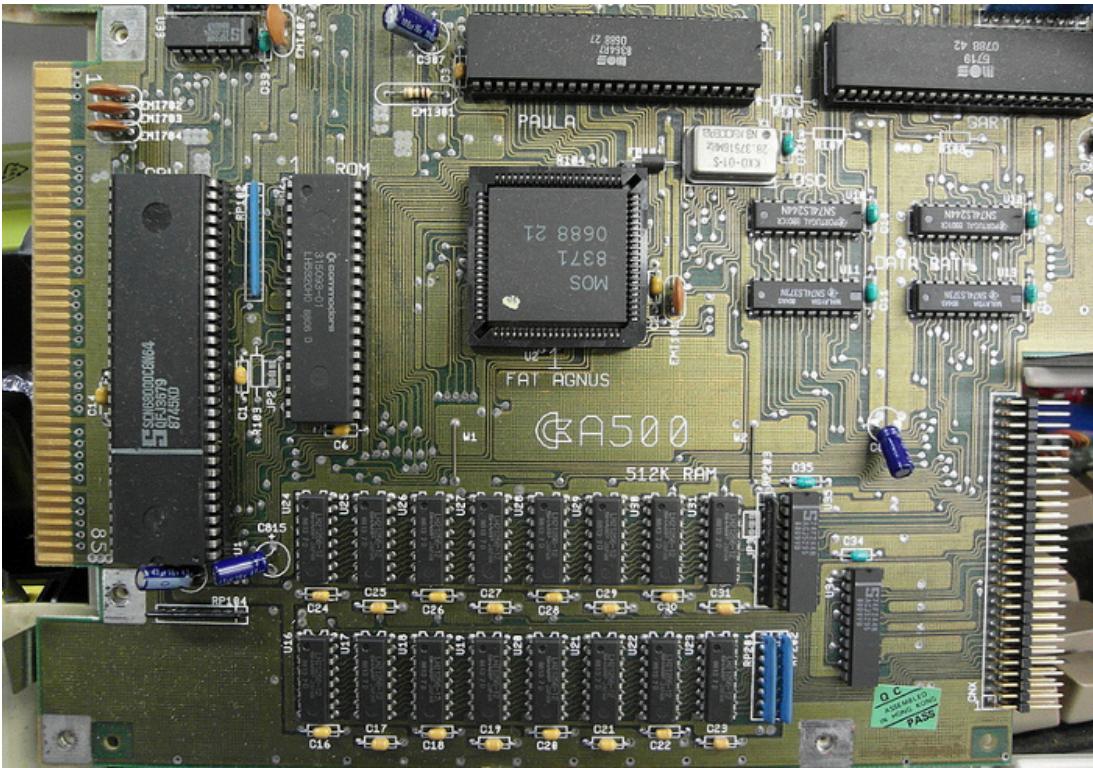
Mehrere Aufgaben werden mit mehreren Ressourcen simultan verarbeitet

In der MIPS Architektur z.B. dedizierte Hardware für: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Accessss (MA) und Write Back (WB)

Durch Pipelining kann so z.B. eine bis zu 5-fache Beschleunigung gegenüber einer hintereinander Ausführung erreicht werden



Amiga 500, 1987



[Bildquelle: [Amiga 500](#) by Dave Jones, [Creative Commons License](#)]

Thorsten Thormählen 51 / 58

iMac G3, 1998

Spätestens seit dem iMac wird bei Apple viel Wert auf das Design gelegt

Apple's Idee: Der iMac integriert Monitor und Rechner-Hardware in einem Gehäuse während dies sonst nicht üblich ist. Dies verhindert Kabelsalat auf dem Schreibtisch, hat aber den Nachteil, dass Komponenten nicht so leicht ausgetauscht werden können.

Das Konzept wird bis heute fortgeführt (oben iMac G3 von 1998, unten iMac von 2007)

Der iMac G3 hatte eine 233 MHz PowerPC CPU mit 512 KB Cache, 4 GB Festplatte, 32 MB RAM, 2 MB Video RAM und wurde mit Mac OS 8.1 ausgeliefert



[Bildquelle: [iMac G3](#), by Rudolf Schuba, [iMac 2007](#), by Burgermac, [Creative Commons License](#)]

Thorsten Thormählen 52 / 58

iPhone, 2007

Das Apple's iPhone revolutionierte den Mobiltelefonmarkt, indem es statt Tasten einen Multi-Touch Bildschirm verwendet

Dies erlaubt ein neuartiges und vereinfachtes Bedienkonzept

Als Mikrocontroller wurde ein Samsung 32-Bit RISC ARM-1176-Prozessor mit 667-MHz verwendet

Das Telefon hatte 128 MB DRAM, ein 3,5-Zoll Display mit einer Auflösung von 320x480 Bildpunkten und eine 2 Megapixel Kamera

D.h. das iPhone der ersten Generation war leistungsstärker als der iMac bei seiner Einführung 10 Jahre zuvor

Smartphones und Tablet-PC (wie z.B. iPad, 2010) erleben seitdem einen immer größeren Zulauf

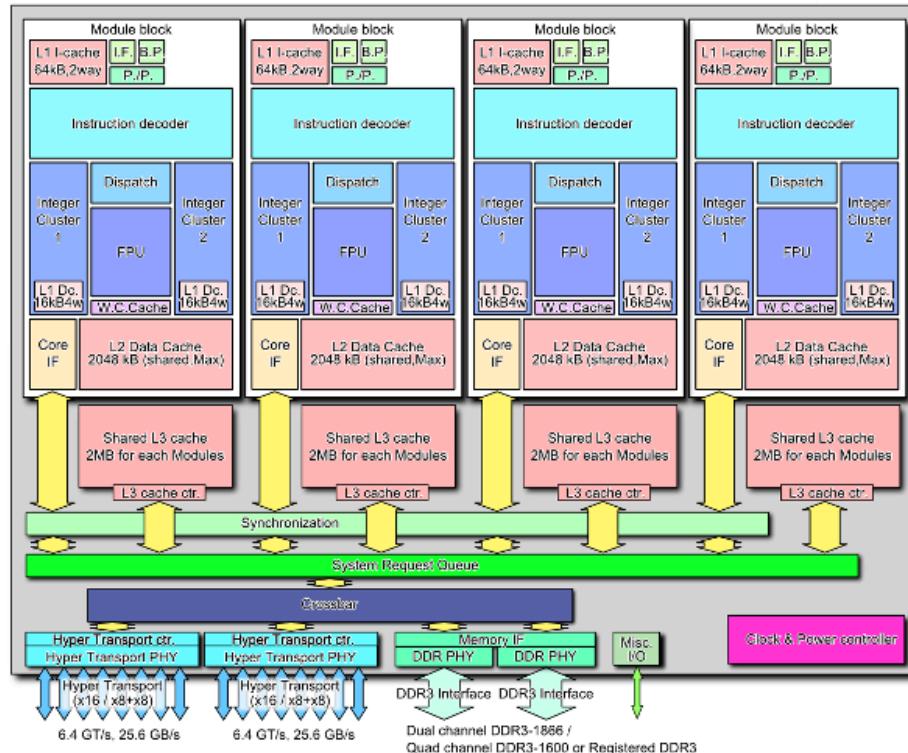


iPhone 4, 2010

[Bildquelle: [iPhone](#), Flickr User: Yutaka Tsutano; [Creative Commons License](#)]

Thorsten Thormählen 53 / 58

AMD Bulldozer (8-Kern-CPU), 2011



[Quelle: [AMD Bulldozer block diagram \(8-core CPU\)](#) , by Shigeru23, Creative Commons License]

Thorsten Thormählen 54 / 58

Intel's und AMD's aktuelle High-End-Desktop-Prozessoren

Intel's und AMD's aktuelle High-End-Desktop-Prozessoren haben beeindruckende technische Daten
(Stand: Okt. 2022)

Z.B. der AMD Ryzen 9 7950X hat:

- eine Strukturbreite von 5 Nanometern
- 13,1 Milliarden Transistoren
- 16 Kerne
- Taktfrequenz 4,5 GHz bis 5,7 GHz
- 80 MB Cache (Level 2 + 3)

der Intel Core i9-13900K hat:

- eine Strukturbreite von 10 Nanometern
- unbekannt (laut grober Schätzungen ca. 25 Milliarden)
- 24 Kerne
- Taktfrequenz 3,0 GHz bis 5,8 GHz
- 68 MB Cache (Level 2 + 3)

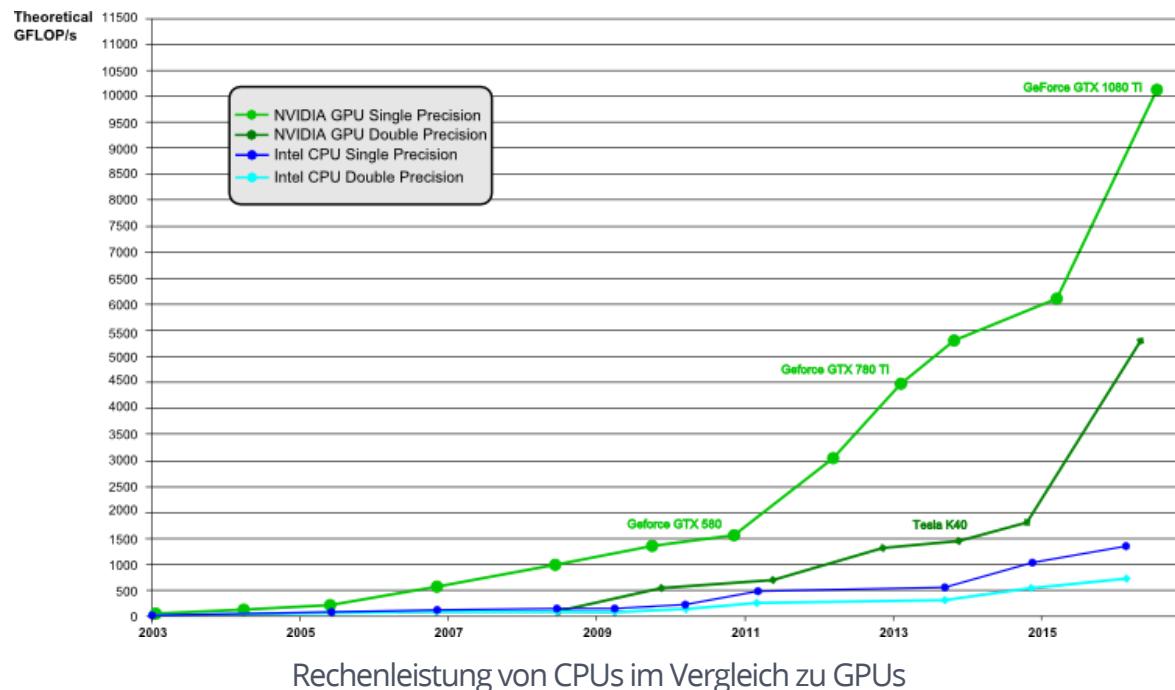


Intel Core i7 CPU
(1. Generation)

[Quelle: [Intel Core i9-13900K](#) , [AMD Ryzen 9 7950X](#) Bildquelle: [Intel Core i7 8200M Q1NG ES Processor Mobile CPU](#) , by Emilian Robert Vicol, [Creative Commons License](#)]

Thorsten Thormählen 55 / 58

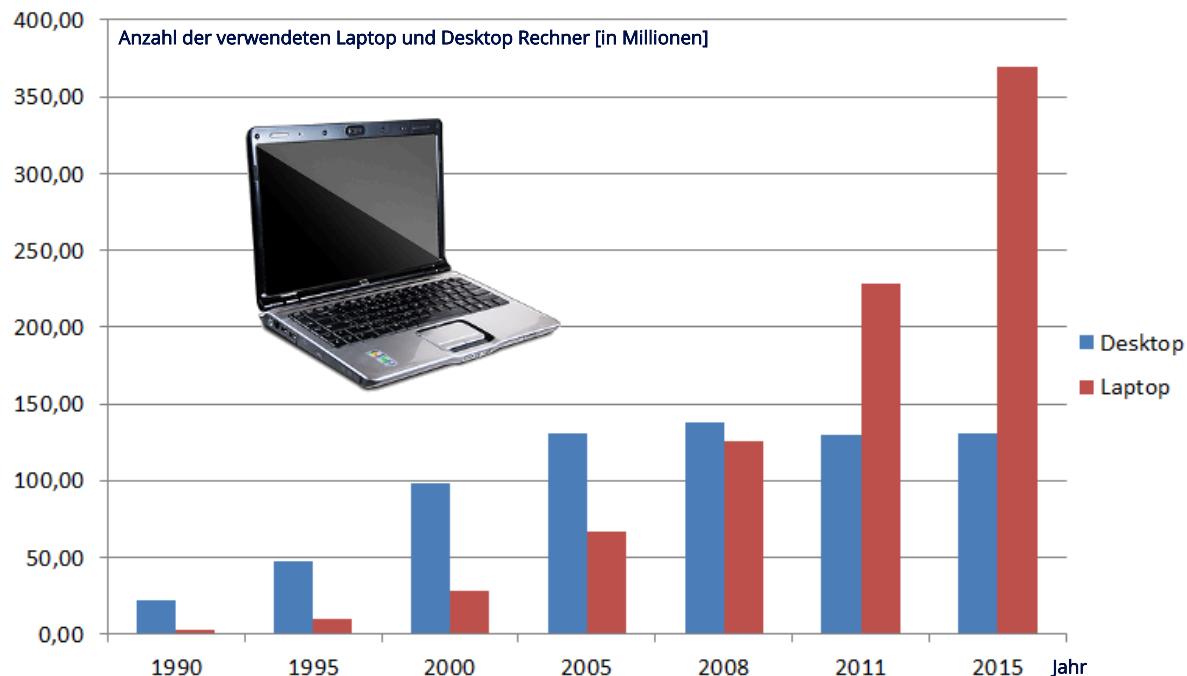
Rechenleistung von CPUs und Grafikkarten (GPUs)



[Quelle: Basiert auf CUDA C Programming Guide Version 6.5]

Thorsten Thormählen 56 / 58

Der Trend geht zum tragbaren Computer



[Quelle: Worldwide PC market, Computer Industry Almanac Inc., Bildquelle: Laptop, by Aaron Patterson, Creative Commons License]

Thorsten Thormählen 57 / 58

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Folien auf Englisch \(Slides in English\)](#)

[\[Impressum\]](#) , [\[Datenschutz\]](#) , []

Thorsten Thormählen 58 / 58

Technische Informatik

Zahlendarstellung

Thorsten Thormählen

22. Oktober 2024

Teil 2, Kapitel 1

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

"Es gibt 10 Gruppen von Menschen: Diejenigen, die Binärkodierungen verstehen und die anderen."

[Quelle:[Wikipedia](#)]

Thorsten Thormählen 4 / 31

Inhalt

Einfache Zahlendarstellungen

Stellenwertsysteme: b-adische Darstellung

Rationale Zahlen

Umwandlung zwischen Zahlensystemen

Zahlendarstellung durch Kerben und Striche

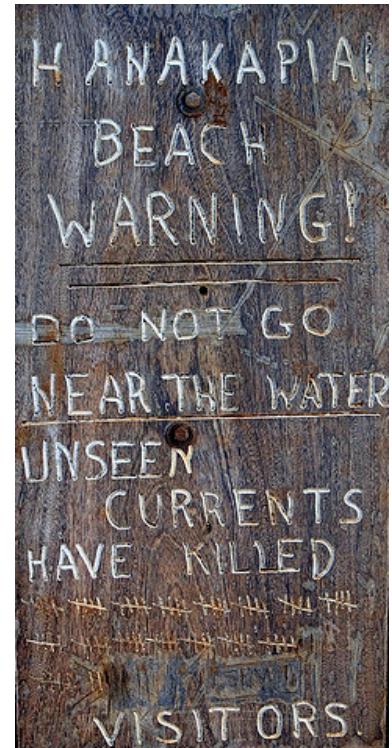
Eine Zahl x wird durch x -fache Wiederholung eines speziellen Zeichens (Strich, Kerbe, etc.) dargestellt

Stichsysteme wurden schon sehr früh von den Menschheit verwendet

Wird heute noch häufig verwendet: z.B. Kaffee-Liste, Inventur

Zwar praktisch, wenn die Zahlen klein sind, aber bei großen Zahlen nicht mehr anwendbar

Addition ergibt sich natürlich



[Bildquelle: [Unseen Currents Have Killed 82 Visitors](#), by eeethaan, [Creative Commons License](#)]

Thorsten Thormählen 6 / 31

Römische Zahlen

Zahlensymbol	Wertigkeit
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Römische Zahlen

Bei den Römischen Zahlen ist die *relative* Position der Zeichen wichtig

Beispiele:

I	= 1	X	= 10
II	= 2	XI	= $(10+1) = 11$
III	= 3	XII	= $(10+2) = 12$
IV	= $(5-1) = 4$	XXXIX	= $(30+10-1) = 39$
V	= 5	XL	= $(50-10) = 40$
VI	= $(5+1) = 6$	L	= 50
VII	= $(5+2) = 7$	LIX	= $(50+10-1) = 59$
VIII	= $(5+3) = 8$	LX	= 60
IX	= $(10-1) = 9$	XC	= $(100-10) = 90$

Dezimalsystem

Das von uns im Alltag verwendete Dezimalsystem kennt 10 Zahlensymbole:
 $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Dabei ist die *absolute Position* der Zeichen wichtig, um den Zahlenwert zu ermitteln

Beispiel: $123 \neq 321$

Beispiel: $5124 = 5 \cdot 1000 + 1 \cdot 100 + 2 \cdot 10 + 4$

Der Zahlenwert w berechnet sich also wie folgt aus den Ziffern z_{n-1}, \dots, z_1, z_0 einer Dezimalzahl:

$$\begin{aligned} w &= z_{n-1} \cdot 10 \underbrace{\dots 0}_{n-1} + \dots z_2 \cdot 100 + z_1 \cdot 10 + z_0 \cdot 1 = \\ &= \sum_{i=0}^{n-1} z_i \cdot 10^i \end{aligned}$$

Bei dem Dezimalsystem handelt es sich um ein so genanntes *Stellenwertsystem*

Das Dezimalsystem ist ein Stellenwertsystem mit der Basis 10

Stellenwertsysteme: b-adische Darstellung

Sei $b > 1$ eine beliebige natürliche Zahl ($b \in \mathbb{N}$), dann heisst die Menge $\{0, \dots, b - 1\}$ das Alphabet des b-adischen Zahlensystems (mit der Basis b)

Dezimalsystem:

$$b = 10 \rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Dualsystem (Binärsystem):

$$b = 2 \rightarrow \{0, 1\}$$

Oktalsystem:

$$b = 8 \rightarrow \{0, 1, 2, 3, 4, 5, 6, 7\}$$

Hexadezimalsystem:

$$b = 16 \rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$$

Stellenwertsysteme: b-adische Darstellung

Wie schon festgestellt, berechnet sich ein Zahlenwert w im Dezimalsystem gemäß:

$$w = \sum_{i=0}^{n-1} z_i \cdot 10^i$$

Dies lässt sich verallgemeinern. Für beliebige Basen $b > 1$, gilt:

$$w = \sum_{i=0}^{n-1} z_i \cdot b^i$$

wobei n die Anzahl der Stellen der Zahl w angibt

Ziffernschreibweise: $w = (z_{n-1} \dots z_2 z_1 z_0)_b$

Dualsystem (Binärsystem)

Der Basis $b = 2$ mit einem Alphabet von zwei Zahlensymbolen, 0 und 1, kommt in der Informatik eine besondere Bedeutung zu, da sich zwei Symbole leicht elektrisch kodieren lassen:

"kein Strom fließt" entspricht typischerweise der 0

"Strom fließt" der 1

Jede Stelle einer Binärzahl wird als "Bit" ("Binary Digit") bezeichnet

1 Bit ist demnach die kleinste Informationsmenge, die gespeichert werden kann

8 Bits werden als 1 Byte bezeichnet

Die Bitfolge $(01001101)_2$ entspricht der Dezimalzahl $(77)_{10}$

$$\begin{aligned}(01001101)_2 &= 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 0 \cdot 128 + 1 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 \\ &= 64 + 8 + 4 + 1 \\ &= 77\end{aligned}$$

Oktal- und Hexadezimalsystem

Oktalsystem:

Die Oktalzahl $(115)_8$ entspricht der Dezimalzahl $(77)_{10}$

$$\begin{aligned}(115)_8 &= 1 \cdot 8^2 + 1 \cdot 8^1 + 5 \cdot 8^0 \\&= 1 \cdot 64 + 1 \cdot 8 + 5 \cdot 1 \\&= 77\end{aligned}$$

Hexadezimalsystem:

Die Zahl $(4D)_{16}$ entspricht der Dezimalzahl $(77)_{10}$

$$\begin{aligned}(4D)_{16} &= 4 \cdot 16^1 + D \cdot 16^0 \\&= 64 + 13 \\&= 77\end{aligned}$$

Zahlensysteme mit verschiedenen Basen

Dies ist ein interaktiver Zähler. Durch Klicken auf den Zähler wird die Zahl erhöht:

+ - Reset

Es können verschiedene Basen ausgewählt werden:

Basis (oben): Basis (unten):

Beobachtung: Kleine Basen benötigen zwar ein kleineres Alphabet, aber dafür mehr Stellen

"Es gibt 10 Gruppen von Menschen: Diejenigen die Ternärkodierungen verstehen, jene die es für Binärkodierung halten und die anderen."

[Quelle:[Wikipedia](#)]

Thorsten Thormählen 15 / 31

Quiz

Frage: Die Zahl $(1220)_3$ im Ternärsystem soll in das Dezimalsystem umgewandelt werden. Was ist das Ergebnis?

Antwort 1: 51

Antwort 2: 17

Antwort 3: 24

Antwort 4: 49

Antwort 5: **keine Ahnung**

Am Online-Quiz teilnehmen durch Besuch der Webseite:

www.onlineclicker.org

Rationale Zahlen

Eine rationale Zahl $w \in \mathbb{Q}$ kann ebenfalls in der b -adischen Darstellung angegeben werden

Dazu wird der Nachkommaanteil durch negative Exponenten repräsentiert

Beispiel aus dem uns vertrauten Dezimalsystem:

$$913,64 = 9 \cdot 10^2 + 1 \cdot 10^1 + 3 \cdot 10^0 + 6 \cdot 10^{-1} + 4 \cdot 10^{-2}$$

Dies lässt sich wieder verallgemeinern:

$$\begin{aligned} w &= (z_{n-1} \dots z_2 z_1 z_0, z_{-1} \dots z_{-m})_b \\ &= \sum_{i=-m}^{n-1} z_i \cdot b^i \end{aligned}$$

wobei n die Anzahl der Vorkommastellen

und m die Anzahl der Nachkommastellen angibt

Beispiel im Binärsystem:

$$\begin{aligned} (11,101)_2 &= 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= 2 + 1 + 0.5 + 0,125 = (3,625)_{10} \end{aligned}$$

Dezimal-Präfixe

Für viele Zehnerpotenzen gibt es standardisierte Präfixe, die vor den eigentlichen physikalischen Einheiten stehen, z.B. $1000 \text{ m} = 1 \text{ km}$

10^1	10^2	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}
Deka	Hekto	Kilo	Mega	Giga	Tera	Peta	Exa	Zetta	Yotta
da	h	k	M	G	T	P	E	Z	Y

10^{-1}	10^{-2}	10^{-3}	10^{-6}	10^{-9}	10^{-12}	10^{-15}	10^{-18}	10^{-21}	10^{-24}
Dezi	Zenti	Milli	Mikro	Nano	Piko	Femto	Atto	Zepto	Yokto
d	c	m	μ	n	p	f	a	z	y

[Quelle: [Wikipedia](#)]

Thorsten Thormählen 18 / 31

Binär-Präfixe

Wenn in der Informatik von 1 kByte (Kilobyte) gesprochen wird, bezeichnet dies nicht immer 1000 Bytes, sondern häufig 1024 Bytes. Dies führt zu Verwirrungen. Daher sollten für die Potenzen von 1024 besser die folgenden Präfixe laut IEC verwendet werden:

Name	Präfix	Wert
kibi	Ki	$2^{10} = 1024^1 = 1.024$
mebi	Mi	$2^{20} = 1024^2 = 1.048.576$
gibi	Gi	$2^{30} = 1024^3 = 1.073.741.824$
tebi	Ti	$2^{40} = 1024^4 = 1.099.511.627.776$
pebi	Pi	$2^{50} = 1024^5 = 1.125.899.906.842.624$
exbi	Ei	$2^{60} = 1024^6 = 1.152.921.504.606.846.976$
zebi	Zi	$2^{70} = 1024^7 = 1.180.591.620.717.411.303.424$
yobi	Yi	$2^{80} = 1024^8 = 1.208.925.819.614.629.174.706.176$

[Quelle:[Wikipedia](#)]

Thorsten Thormählen 19 / 31

Umwandlung zwischen Zahlensystemen

Zur Entwicklung eines Algorithmus zum Umrechnen zwischen Zahlensystemen betrachten wir was passiert, wenn wir die hergeleitete Formel zur b -adischen Darstellung durch die Basis b dividieren:

$$\begin{aligned} w = \sum_{i=0}^{n-1} z_i \cdot b^i &\Leftrightarrow \frac{w}{b} = \frac{1}{b} \sum_{i=0}^{n-1} z_i \cdot b^i \\ &= \underbrace{\left(\sum_{i=0}^{n-2} z_{i+1} \cdot b^i \right)}_{s_1} + \frac{z_0}{b} \end{aligned}$$

Der erste Summand s_1 ist das ganzzahlige Ergebnis der Division und z_0 der Rest der Division

Damit wurde gezeigt, dass die letzte Ziffer z_0 aus dem Rest der Division mit der gewünschten Basis b errechnet werden kann

Das ganzzahlige Ergebnis der Division stellt den Zahlenwert der verbleibenden Ziffern (ohne z_0) dar. Durch rekursive Anwendung können alle Ziffern bestimmt werden.

Umwandlung zwischen Zahlensystemen

Beispiel: Umrechnung der Dezimalzahl 6485 ins Binärsystem:

```
6485 / 2 = 3242 Rest 1
3242 / 2 = 1621 Rest 0
1621 / 2 = 810 Rest 1
810 / 2 = 405 Rest 0
405 / 2 = 202 Rest 1
202 / 2 = 101 Rest 0
101 / 2 = 50 Rest 1
50 / 2 = 25 Rest 0
25 / 2 = 12 Rest 1
12 / 2 = 6 Rest 0
6 / 2 = 3 Rest 0
3 / 2 = 1 Rest 1
1 / 2 = 0 Rest 1
```

Das Verfahren endet, wenn das Ergebnis der Division gleich Null ist

Die Binärzahl kann anhand der Reste von unten nach oben ablesen werden:
 $(1100101010101)_2$

Hinweis: Soll ein solcher Algorithmus implementiert werden, kann der Divisionsrest mittels der Modulo-Operation ermittelt werden: $6485 \bmod 2 = 1$

Umwandlung zwischen Zahlensystemen

Beispiel: Umrechnung der Dezimalzahl 23521 ins Hexadezimalsystem:

$$\begin{aligned} 23521 \div 16 &= 1470 \text{ Rest } 1 \text{ (1)} \\ 1470 \div 16 &= 91 \text{ Rest } 14 \text{ (E)} \\ 91 \div 16 &= 5 \text{ Rest } 11 \text{ (B)} \\ 5 \div 16 &= 0 \text{ Rest } 5 \text{ (5)} \end{aligned}$$

Die Ergebnis kann wieder anhand der Reste ablesen werden: $(5BE1)_{16}$

Umwandlung zwischen Zahlensystemen

Beispiel: Umrechnung von $(360)_7$ ins Quinärsystem (Basis 5)

Zunächst muss $(360)_7$ ins Dezimalsystem gewandelt werden:

$$(360)_7 = 3 \cdot 49 + 6 \cdot 7 + 0 \cdot 1 = (189)_{10}$$

Dann erfolgt die Umrechnung von $(189)_{10}$ ins Quinärsystem:

189	/	5	=	37	Rest	4
37	/	5	=	7	Rest	2
7	/	5	=	1	Rest	2
1	/	5	=	0	Rest	1

Die Ergebnis kann wieder anhand der Reste ablesen werden: $(1224)_5$

Anstatt den Umweg über das Dezimalsystem zu gehen, hätten wir theoretisch das Verfahren auch direkt im 7er-System anwenden können. Dann müssten wir jedoch die Operationen (Division und Bildung des Rests) im 7-er System ausführen, was wir als Menschen nicht gewohnt sind.

Umwandlung zwischen Binär- und Hexadezimalsystem

Je 4 Bits entsprechen genau einer Hexadezimalziffer

0000 = 0	0100 = 4	1000 = 8	1100 = C
0001 = 1	0101 = 5	1001 = 9	1101 = D
0010 = 2	0110 = 6	1010 = A	1110 = E
0011 = 3	0111 = 7	1011 = B	1111 = F

Damit kann die Umwandlung eine Binärzahl direkt aus den 4-er Blöcken abgelesen werden:

$$(1101\ 0001\ 1110\ 1111)_2 = (D1EF)_{16}$$

Die Hexdezimaldarstellung eignet sich daher besonders, um eine Bitfolge übersichtlich und kompakt darzustellen

Ähnlich leicht ist die Umwandlung einer Binärzahl in andere Systeme mit einer Basis $b = 2^k$, wie z.B. 4 oder 8

"If only dead people understand hexadecimal,
how many people
understand hexadecimal?"

$$\begin{aligned}(DEAD)_{16} &= 13 \cdot 16^3 + 14 \cdot 16^2 \\ &\quad + 10 \cdot 16^1 + 13 \cdot 16^0 = (57005)_{10}\end{aligned}$$

Umwandlung bei rationalen Zahlen

Zur Umwandlung von rationalen Zahlen betrachten wir den Vorkommaanteil und Nachkommaanteil getrennt:

$$\begin{aligned} w &= (z_{n-1} \dots z_2 z_1 z_0, z_{-1} \dots z_{-m})_b \\ &= \sum_{i=-m}^{n-1} z_i \cdot b^i \\ &= \left(\sum_{i=0}^{n-1} z_i \cdot b^i \right) + \left(\sum_{i=-m}^{-1} z_i \cdot b^i \right) \end{aligned}$$

Die Umrechnung für den Vorkommaanteil ist bekannt, daher betrachten wir nun nur noch den Nachkommaanteil \tilde{w} :

$$\tilde{w} = (0, z_{-1} \dots z_{-m})_b = \left(\sum_{i=-m}^{-1} z_i \cdot b^i \right)$$

Umwandlung bei rationalen Zahlen

Im Gegensatz zum Vorgehen bei der Umwandlung des Vorkommaanteils (Rest der Division) muss beim Nachkommaanteil mit der Basis b multipliziert werden:

$$\begin{aligned}\tilde{w} = \sum_{i=-m}^{-1} z_i \cdot b^i &\Leftrightarrow \tilde{w} \cdot b = b \sum_{i=-m}^{-1} z_i \cdot b^i \\ &= \left(\sum_{i=-m}^{-2} z_i \cdot b^{i+1} \right) + z_{-1}\end{aligned}$$

Der Summand z_{-1} ist der einzige Anteil der ≥ 1 sein kann. Damit ist der Vorkommaanteil von $\tilde{w} \cdot b$ gleich z_{-1}

Der Nachkommaanteil von $\tilde{w} \cdot b$ stellt den Zahlenwert der verbleibenden Ziffern (ohne z_{-1}) dar. Durch rekursive Anwendung können alle Ziffern bestimmt werden.

Umwandlung bei rationalen Zahlen

Beispiel: Umrechnung der Dezimalzahl 0,6875 ins Binärsystem:

$$\begin{aligned}0,6875 * 2 &= 1,375 \\0,375 * 2 &= 0,75 \\0,75 * 2 &= 1,5 \\0,5 * 2 &= 1,0\end{aligned}$$

Das Ergebnis kann aus dem Vorkommaanteil von oben nach unten abgelesen werden: $(0,1011)_2$

Das Verfahren endet, wenn der Nachkommaanteil gleich Null ist

Umwandlung bei rationalen Zahlen

Beispiel: Umrechnung der Dezimalzahl 0,1 ins Binärsystem:

```
0,1 * 2 = 0,2  
0,2 * 2 = 0,4  
0,4 * 2 = 0,8  
0,8 * 2 = 1,6  
0,6 * 2 = 1,2  
0,2 * 2 = 0,4  
0,4 * 2 = ...
```

Der Algorithmus endet nie

Der Nachkommaanteil ist damit im Binärsystem periodisch, obwohl er es im Dezimalsystem nicht ist

Beim Runden auf eine endliche Anzahl von Stellen kommt es daher zu Rundungsfehlern

Umwandlung bei rationalen Zahlen

Rundungsfehler bei der Umwandlung ins Binärsystem müssen bei der Entwicklung von Software und Hardware stets berücksichtigt werden

Ansonsten verhält sich das System anders als erwartet

Beispiel:

```
double a = 0.1;
double b = a * 3;
if(a == 0.1) print("This is ");
if(b == 0.3) print("no"); else print("a");
print(" round-off error\n");
```


Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Folien auf Englisch \(Slides in English\)](#)

[\[Impressum\]](#) , [\[Datenschutz\]](#) , []

Thorsten Thormählen 31 / 31

Technische Informatik

Rechnerinterne Zahlenformate

Thorsten Thormählen
27. Oktober 2022
Teil 2, Kapitel 2

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

- Darstellung vorzeichenloser ganzer Zahlen
- Darstellung vorzeichenbehafteter ganzer Zahlen
 - Betrags-Vorzeichendarstellung
 - Einerkomplement
 - Zweierkomplement
- Darstellung vorzeichenbehafteter rationaler Zahlen
 - Festkommadarstellung
 - Gleitkommadarstellung

Darstellung vorzeichenloser ganzer Zahlen

Je nachdem wie viele Bits zur binären Speicherung einer vorzeichenlosen ganzen Zahlen (unsigned integer) zur Verfügung gestellt werden, können verschiedene große Zahlen dargestellt werden

Bytes	Bits	Wertebereich	Name des Datentyps (Microsoft Visual C++)
1	8	[0; 255]	unsigned char
2	16	[0; 65535]	unsigned short
4	32	[0; 4294967295]	unsigned int
8	64	[0; 18446744073709551615]	unsigned long long

[Quelle: Data Type Ranges, Visual Studio 2013]

Thorsten Thormählen 5 / 35

Big- und Little-Endian-Speicherung

Little-Endian

Das Byte mit der niedrigsten Wertigkeit wird *zuerst* gespeichert

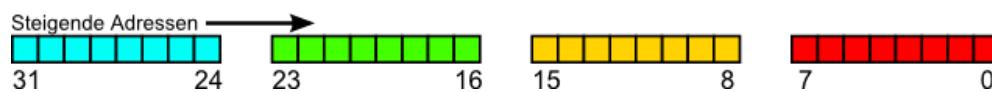
Alle x86-kompatiblen Rechner verwenden diese Ordnung



Big-Endian

Das Byte mit der niedrigsten Wertigkeit wird *zuletzt* gespeichert

MIPS, SPARC, PowerPC, Java Virtual Machine, etc.



Darstellung vorzeichenbehafteter ganzer Zahlen

Sollen positive und negative Zahlen gespeichert werden, so gibt es verschiedene Möglichkeiten:

Betrags-Vorzeichendarstellung

Einerkomplement

Zweierkomplement

Jede Darstellung hat Vor- und Nachteile bezüglich

- der Symmetrie des darstellbaren Wertebereichs
- der Eineindeutigkeit der Darstellung
- der Durchführung von arithmetischen Operationen

Betrags-Vorzeichendarstellung

Das höchstwertige Bit wird für das Vorzeichen verwendet:

0 = Zahl positiv; 1 = Zahl negativ

Die verbleibenden Bits werden für den Betrag verwendet.

Insgesamt kann so ein symmetrischer Wertebereich von $[-(2^{n-1} - 1), + (2^{n-1} - 1)]$ dargestellt werden

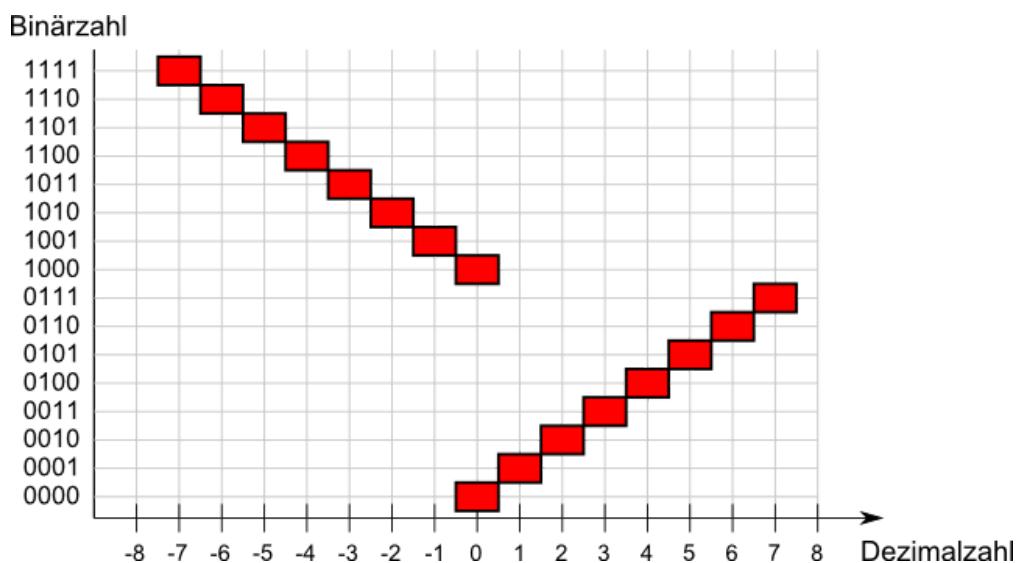
Beispiel für $n = 4$:

Dezimal	0	1	2	3	4	5	6	7
Binär	0000	0001	0010	0011	0100	0101	0110	0111

Dezimal	-0	-1	-2	-3	-4	-5	-6	-7
Binär	1000	1001	1010	1011	1100	1101	1110	1111

Betrags-Vorzeichendarstellung

Die Darstellung ist nicht eindeutig, da die Null doppelt repräsentiert ist. Dies führt zu Problemen beim Gleichheitstest: $-0 \neq 0$



Betrags-Vorzeichendarstellung

Die Addition/Subtraktion ist umständlich, da das Vorzeichen-Bit gesondert behandelt werden muss, ansonsten:

$$\begin{array}{r} 5 \\ + -6 \\ \hline = -1 \end{array} \qquad \begin{array}{r} 0101 \qquad (5) \\ +1110 \qquad (-6) \\ \hline = 10011 \qquad (-3) \end{array}$$

[Quelle: D.W. Hoffmann: Grundlagen der Technischen Informatik, 2. Auflage; Hanser 2009]

Thorsten Thormählen 10 / 35

Einerkomplement

Eine negative Zahl ($-w$) wird binär durch das bitweise Komplement der entsprechenden positiven Zahl w dargestellt

Beispiel: $(6)_{10} \hat{=} 0110 \rightarrow (-6)_{10} \hat{=} 1001$

Insgesamt kann so ein symmetrischer Wertebereich von $[-(2^{n-1} - 1), + (2^{n-1} - 1)]$ dargestellt werden

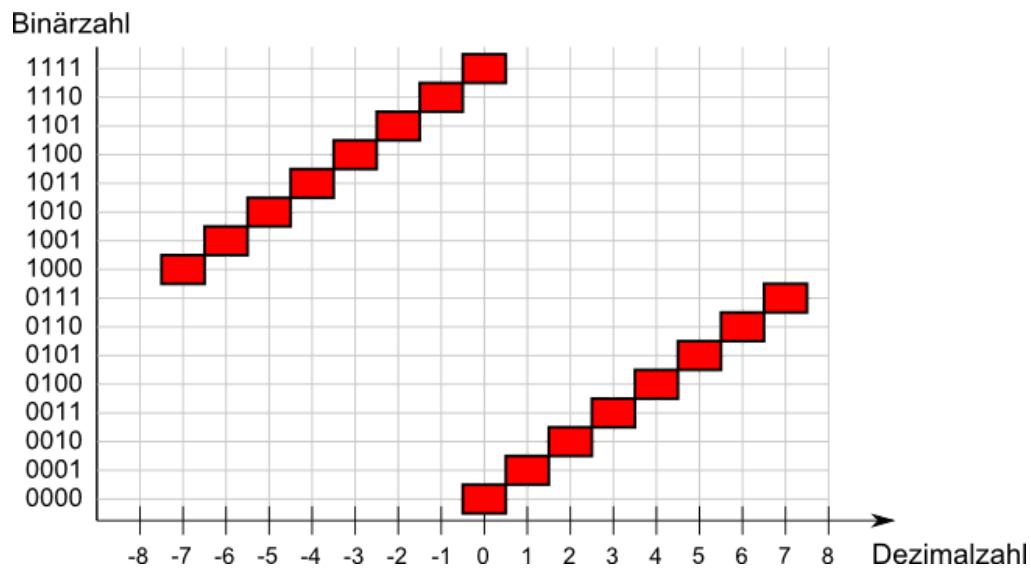
Beispiel für $n = 4$:

Dezimal	0	1	2	3	4	5	6	7
Binär	0000	0001	0010	0011	0100	0101	0110	0111

Dezimal	-0	-1	-2	-3	-4	-5	-6	-7
Binär	1111	1110	1101	1100	1011	1010	1001	1000

Einerkomplement

Die Darstellung ist nicht eineindeutig, da die Null doppelt repräsentiert ist



Einerkomplement

Die Addition/Subtraktion muss in zwei Schritten erfolgen

1. Normale Binäraddition
2. Zusätzliche Addition eines eventuellen Übertrags (der bei der 2. Addition entstehender Übertrag wird gestrichen)

Beispiel 1:

$$\begin{array}{r} 5 \\ + -6 \\ \hline = -1 \end{array} \quad \begin{array}{r} 0101 \quad (5) \\ +1001 \quad (-6) \\ \hline = 1110 \quad (-1) \end{array}$$

Beispiel 2:

$$\begin{array}{r} 5 \\ + -3 \\ \hline = 2 \end{array} \quad \begin{array}{r} 0101 \quad (5) \\ +1100 \quad (-3) \\ \hline = 1|0001 \quad (-14) \\ \quad \quad \quad +1 \quad (1) \\ \hline = 1|0010 \quad (2) \end{array}$$

[Quelle: D.W. Hoffmann: Grundlagen der Technischen Informatik, 2. Auflage; Hanser 2009]

Thorsten Thormählen 13 / 35

Zweierkomplement

Beim Zweierkomplement werden die negativen Zahlen durch Bestimmung des Einerkomplements und einer zusätzlichen Addition von 1 gebildet

Beispiel:

$$(-6)_{10} \hat{=} \text{not}(0110) + 1 = 1001 + 1 = 1010$$

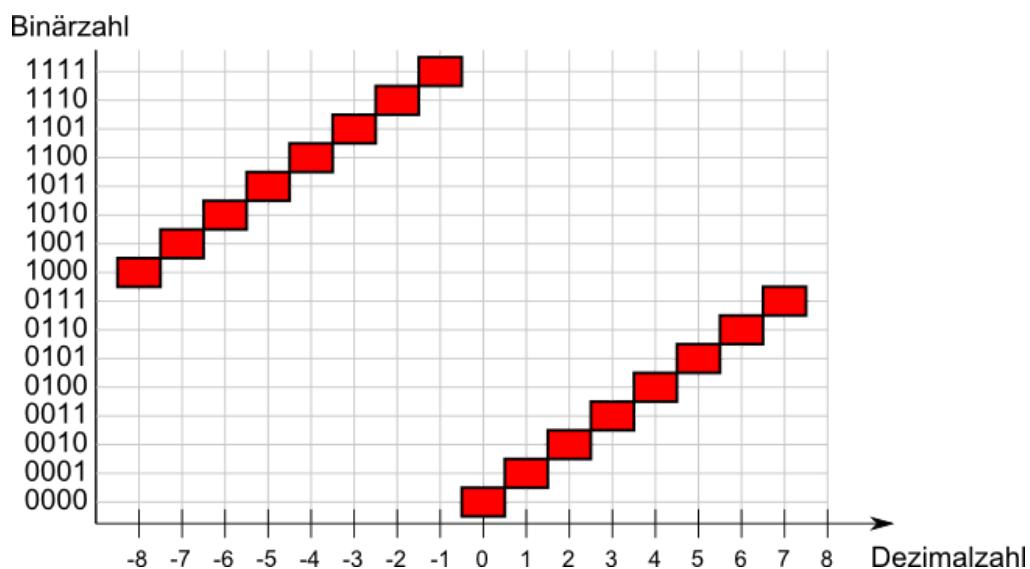
Wertebereich ist unsymmetrisch $[-2^{n-1}, +2^{n-1} - 1]$

Beispiel für $n = 4$:

Dezimal	0	1	2	3	4	5	6	7
Binär	0000	0001	0010	0011	0100	0101	0110	0111
Dezimal	-1	-2	-3	-4	-5	-6	-7	-8
Binär	1111	1110	1101	1100	1011	1010	1001	1000

Zweierkomplement

Die Darstellung ist eineindeutig



Zweierkomplement

Zur Addition/Subtraktion kann die normale vorzeichenlose Binäraddition verwendet werden

Subtraktion entspricht Negation und anschließender Addtion, z.B.
 $(5 - 6) = (5 + (-6))$

Beispiel 1:

$$\begin{array}{r} 5 \\ + -6 \\ \hline = -1 \end{array} \qquad \begin{array}{r} 0101 \quad (5) \\ +1010 \quad (-6) \\ \hline = 1111 \quad (-1) \end{array}$$

Beispiel 2:

$$\begin{array}{r} 5 \\ + -3 \\ \hline = 2 \end{array} \qquad \begin{array}{r} 0101 \quad (5) \\ +1101 \quad (-3) \\ \hline = 1|0010 \quad (2) \end{array}$$

Zweierkomplement

Beispiel 3:

$$\begin{array}{r} 4 \\ + -3 \\ \hline = 1 \end{array} \quad \begin{array}{r} 0100 \quad (4) \\ +1101 \quad (-3) \\ \hline =1|0001 \quad (1) \end{array}$$

Beispiel 4:

$$\begin{array}{r} -3 \\ + -4 \\ \hline = -7 \end{array} \quad \begin{array}{r} 1101 \quad (-3) \\ +1100 \quad (-4) \\ \hline =1|1001 \quad (-7) \end{array}$$

Es scheint als könnten wir (anders als beim Einerkomplement) den Übertrag beim Zweierkomplement einfach streichen.

Ist diese immer der Fall?

Zweierkomplement

Eigentlich ja, aber Vorsicht:

Beispiel 5:

$$\begin{array}{r} -4 \\ + -5 \\ \hline = -9 \end{array} \qquad \begin{array}{r} 1100 \quad (-4) \\ +1011 \quad (-5) \\ \hline =10111 \quad (7) \end{array}$$

Beispiel 6:

$$\begin{array}{r} 4 \\ + 5 \\ \hline = 9 \end{array} \qquad \begin{array}{r} 0100 \quad (4) \\ +0101 \quad (5) \\ \hline = 1001 \quad (-7) \end{array}$$

Hier kommt es jeweils zum falschen Ergebnis. Der Übertrag kann nur gestrichen werden, wenn es nicht zum Überlauf kommt, d.h. das Ergebnis den Bereich der darstellbaren Zahlen nicht verlässt

Zweierkomplement

Nochmal Beispiel 5, jetzt mit $n = 5$

$$\begin{array}{r} -4 \\ + -5 \\ \hline = -9 \end{array} \qquad \begin{array}{r} 11100 \quad (-4) \\ +11011 \quad (-5) \\ \hline =1|10111 \quad (-9) \end{array}$$

Nochmal Beispiel 6, jetzt mit $n = 5$:

$$\begin{array}{r} 4 \\ + 5 \\ \hline = 9 \end{array} \qquad \begin{array}{r} 00100 \quad (4) \\ +00101 \quad (5) \\ \hline = 01001 \quad (9) \end{array}$$

Bei ausreichender Anzahl von Stellen (kein Überlauf) kann der Übertrag gestrichen werden

Zweierkomplement

Beim Zweierkomplement berechnet sich der Zahlenwert w formal wie folgt als den Ziffern z_{n-1}, \dots, z_1, z_0 einer Binärzahl:

$$w = -z_{n-1} \cdot 2^{n-1} + z_{n-2} \cdot 2^{n-2} + \dots + z_0 \cdot 2^0$$

Beispiel:

$$1101 = -1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = -3$$

Wann kommt es beim Zweierkomplement zum Überlauf?

Betrachten wir die Addition von w_a und w_b mit:

$$w_a = -a_{n-1} \cdot 2^{n-1} + a_{n-2} \dots 2^{n-2} + \dots + a_0 \cdot 2^0$$
$$w_b = -b_{n-1} \cdot 2^{n-1} + b_{n-2} \dots 2^{n-2} + \dots + b_0 \cdot 2^0$$

Sei c_{n-1} ein eventueller Übertrag, der aus der binären Addition der Stellen a_{n-1} und b_{n-1} entsteht, und c_{n-2} der eventuelle Übertrag bei a_{n-2} und b_{n-2}

Vorzeichen w_a	Vorzeichen w_b	Richtiges Ergebnis, wenn	Überlauf (falsches Ergebnis), wenn
+	+	$c_{n-1} = 0$ und $c_{n-2} = 0$	$c_{n-1} = 0$ und $c_{n-2} = 1$
+	-	$c_{n-1} = c_{n-2}$	nie
-	+	$c_{n-1} = c_{n-2}$	nie
-	-	$c_{n-1} = 1$ und $c_{n-2} = 1$	$c_{n-1} = 1$ und $c_{n-2} = 0$

Bei zwei positiven Zahlen entsteht ein Überlauf, wenn der Übertrag in die letzte Stelle das Vorzeichenbit der Summe verfälscht

Bei zwei negativen Zahlen entsteht ein Überlauf, wenn die Umkehrung des Vorzeichenbits nicht durch einen Übertrag in die letzte Stelle kompensiert wird

Später werden wir lernen, dass ein Überlauf schaltungstechnisch durch ein XOR detektiert werden kann, d.h. Überlauf tritt auf, wenn $(c_{n-1} \text{ xor } c_{n-2}) = 1$

Inverse des Zweierkomplements

Die Inverse Operation des Zweierkomplements ist einfach eine erneute Anwendung des Zweierkomplements

Beispiel:

$$(6)_{10} \hat{=} 0110$$

$$(-6)_{10} \hat{=} \text{not}(0110) + 1 = 1001 + 1 = 1010$$

$$(6)_{10} \hat{=} \text{not}(1010) + 1 = (0101) + 1 = 0110$$

D.h. jede Anwendung des Zweierkomplements negiert die repräsentierte Dezimalzahl

Darstellung vorzeichenbehafteter ganzer Zahlen

Zusammenfassung

Verfahren	Wertbereich	Eineindeutig	Operationen
Betrags-Vorzeichendarstellung	symmetrisch	nein	Sonderbehandlung
Einerkomplement	symmetrisch	nein	2-Stufen Addition
Zweierkomplement	unsymmetrisch	ja	einfach

Das Zweierkomplement ist die in der Praxis vorherrschende Art zur Darstellung binärer vorzeichenbehafteter ganzer Zahlen (Unsymmetrie wird in Kauf genommen)

Bytes	Bits	Wertebereich	Name des Datentyps (Microsoft Visual C++)
1	8	[−128; 127]	char
2	16	[−32768; 32767]	short
4	32	[−2147483648; 2147483647]	int
8	64	[−9223372036854775808; 9223372036854775807]	long long

[Quelle: [Data Type Ranges, Visual Studio 2013](#)]

Thorsten Thormählen 23 / 35

Darstellung vorzeichenbehafteter rationaler Zahlen

Zur binären Repräsentation vorzeichenbehafteter rationaler Zahlen werden wir zwei Verfahren betrachten:

Festkommadarstellung

Gleitkommadarstellung

Festkommadarstellung

Die *Festkommadarstellung* ist ähnlich der Betrags-Vorzeichendarstellung bei ganzen Zahlen

Das höchstwertige Bit s wird für das Vorzeichen verwendet

Die verbleibenden $n - 1$ Bits werden als *Mantisse* bezeichnet und werden zur Speicherung der Vor- und Nachkommastellen verwendet

Beispiel für $n = 16$:



In der Mantisse werden die ersten $k \leq (n - 1)$ Stellen als Vorkommastellen festgelegt. Damit gibt es $j = (n - 1) - k$ Nachkommastellen

Der Zahlenwert w berechnet sich dann gemäß:

$$w = (-1)^s \left(\left(\sum_{i=0}^{k-1} m_{j+i} \cdot 2^i \right) + \left(\sum_{i=-j}^{-1} m_{j+i} \cdot 2^i \right) \right) = (-1)^s \left(\sum_{i=-j}^{k-1} m_{j+i} \cdot 2^i \right)$$

Festkommadarstellung

Beispiel: Umrechnung von 11011001 für $n = 8$, $k = 4$ und $j = 3$:



$$\begin{aligned} w &= (-1)^s \left(\sum_{i=-j}^{k-1} m_{j+i} \cdot 2^i \right) \\ &= (-1)^1 \left(1 \cdot 2^{-3} + 0 \cdot 2^{-2} + 0 \cdot 2^{-1} + 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 \right) \\ &= (-1)(0,125 + 1 + 2 + 8) = -11,125 \end{aligned}$$

Festkommadarstellung ist ein äquidistantes Zahlenformat, da der nummerische Abstand zwischen zwei darstellbaren Zahlen konstant ist

Ist der Wertebereich der verwendeten Zahlen sehr ähnlich und vorher bekannt, kann mit einer Festkommadarstellung somit eine bestimmte Genauigkeit der Darstellung garantiert werden

Die Festkommadarstellung kommt daher bei spezieller Hardware, wie z.B. digitalen Signalprozessoren (DSPs) zum Einsatz

Gleitkommadarstellung

In der Praxis müssen oft Zahlen mit verschiedenen Größenordnungen verarbeitet werden

Als Beispiel können physikalische Naturkonstanten betrachtet werden:

Lichtgeschwindigkeit: $2,99792458 \cdot 10^8$ m/s

Elektronenmasse: $9,10938291 \cdot 10^{-31}$ kg

Elementarladung: $1,60217656 \cdot 10^{-19}$ C

Bei der wissenschaftlichen Potenzschreibweise wird das Komma durch den Exponenten verschoben:

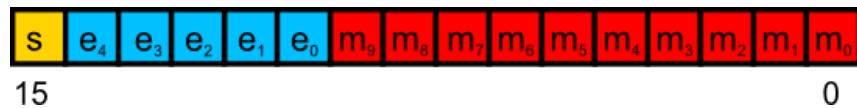
Verschiebung nach rechts für kleine Zahlen: $0,00041 = 4,1 \cdot 10^{-4}$

Verschiebung nach links für große Zahlen: $1200000,00041 \approx 1,2 \cdot 10^6$

Die binäre Gleitkommadarstellung (engl. "floating point") kopiert die wissenschaftlichen Potenzschreibweise und erweitert die Festkommadarstellung um einen binär-kodierten Exponenten

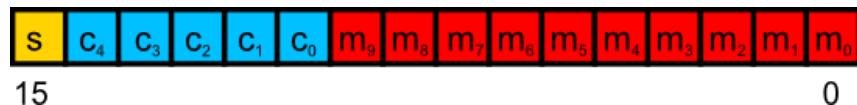
Gleitkommadarstellung

Beispiel: 1 Bit Vorzeichen, 5 Bit Exponent, 10 Bit Mantisse



Die 5 Bits des Exponenten können $2^5 = 32$ verschiedene Zahlen darstellen

Um negative Exponenten e darzustellen, werden statt dem Exponenten dessen Charakteristik $c \geq 0$ in den 5 Bits gespeichert:



Die Umrechnung zwischen Charakteristik und Exponenten erfolgt durch Subtraktion einer Konstanten, z.B. $e = c - 15$

Damit kann der Exponent Werte aus dem Intervall $[-15, 16]$ annehmen

Gleitkommadarstellung

Die Gleitkommadarstellung ist zunächst nicht eindeutig

$$\begin{aligned}0,00111011 &= 000001,11011 \cdot 2^{-3} \\&= 00000,111011 \cdot 2^{-2} \\&= 0000,0111011 \cdot 2^{-1} \\&= 000,00111011 \cdot 2^0 \\&= 00,000111011 \cdot 2^1 \\&= 0,0000111011 \cdot 2^2 \\&= \dots\end{aligned}$$

Bei der *Vorkomma-Normalisierung* wird der Exponent so gewählt, dass die erste Vorkommastelle ungleich 0 ist, hier also $000001,11011 \cdot 2^{-3}$

Bei der *Nachkomma-Normalisierung* wird der Exponent so gewählt, dass die erste Nachkommastelle ungleich 0 ist, hier also $00000,111011 \cdot 2^{-2}$

Bei einer normalisierten Darstellung ist bekannt, dass das erste relevante Bit immer eine 1 ist. Dieses implizite Bit kann daher in einer sogenannten *gepackten* Darstellung weggelassen werden.

Gleitkommadarstellung



Bsp 1: 1001011010000000 (Nachkomma-Normalisierung, ungepackt)

$$s = 1$$

$$e = c - (15)_{10} = (00101)_2 - (15)_{10} = (5)_{10} - (15)_{10} = -10$$

$$m = 0,1010000000 = 1 \cdot 2^{-1} + 1 \cdot 2^{-3} = 0,5 + 0,125 = 0,625$$

Ergebnis: $-0,625 \cdot 2^{-10}$

Bsp 2: 1001011010000000 (Vorkomma-Normalisierung, ungepackt)

$$s = 1$$

$$e = c - (15)_{10} = (00101)_2 - (15)_{10} = (5)_{10} - (15)_{10} = -10$$

$$m = 1,010000000 = 1 \cdot 2^0 + 1 \cdot 2^{-2} = 1 + 0,25 = 1,25$$

Ergebnis: $-1,25 \cdot 2^{-10}$

Gleitkommadarstellung



Bsp 3: 0101011010000000 (Nachkomma-Normalisierung, gepackt)

$$s = 0$$

$$e = c - (15)_{10} = (10101)_2 - (15)_{10} = (21)_{10} - (15)_{10} = (6)_{10}$$

$$m = 0,11010000000 =$$

$$= 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-4} = 0,5 + 0,25 + 0,0625 = 0,8125$$

$$\text{Ergebnis: } 0,8125 \cdot 2^6$$

Bsp 4: 0101011010000000 (Vorkomma-Normalisierung, gepackt)

$$s = 0$$

$$e = c - (15)_{10} = (10101)_2 - (15)_{10} = (21)_{10} - (15)_{10} = (6)_{10}$$

$$m = 1,1010000000 =$$

$$= 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-3} = 1 + 0,5 + 0,125 = 1,625$$

$$\text{Ergebnis: } 1,625 \cdot 2^6$$

Gleitkommadarstellung: IEEE 754 Formate

Single Precision (32-bit)



Double Precision (64-bit)



Charakteristik c	Mantisse m	32-/64-bit Precision
000...0000	beliebig	32: $(-1)^s 0, m \cdot 2^{-126}$
111...1111		64: $(-1)^s 0, m \cdot 2^{-1022}$
111...1111	= 0	$(-1)^s \cdot \infty$
alle anderen Bitfolgen (default)	$\neq 0$	Not a Number (NaN)
	beliebig	32: $(-1)^s 1, m \cdot 2^{c-127}$
		64: $(-1)^s 1, m \cdot 2^{c-1023}$

[Quelle: D.W. Hoffmann: Grundlagen der Technischen Informatik, 2. Auflage; Hanser 2009]

Thorsten Thormählen 32 / 35

Gleitkommadarstellung: IEEE 754 Formate

Die IEEE 754 Formate (aus dem Jahre 1985) werden heutzutage von allen gängigen Mikroprozessoren unterstützt

Normalerweise gepackte Darstellung mit Vorkomma-Normalisierung

Es gibt einige reservierte Bitmuster für die Charakteristik mit Sonderbedeutung:

Null

Charakteristik $c = 000 \dots 0000$ schaltet in eine ungepackte Darstellung um

Darstellung der Null durch $m = 0$

Unendlich

Charakteristik $c = 1111 \dots 111$ und $m = 0$

Vorzeichen s entscheidet zwischen $+\infty$ und $-\infty$

Dies entsteht z.B. bei Division durch 0

Not a Number (NaN)

Charakteristik $c = 1111 \dots 111$ und $m \neq 0$

Dies entsteht bei algebraischen Operationen mit undefiniertem oder nicht repräsentierbarem Ergebnis, wie z.B. $0/0$, $0 \cdot \infty$, usw.

Gleitkommadarstellung: IEEE 754 Formate

Für die Darstellung von Gleitpunktzahlen in einem (Quell-)Text wird typischerweise die Notation "52.21e-2" oder "52.21E-2" verwendet. Die Zahl hinter dem "e" gibt den Exponenten der Zehnerpotenz an.

Diese Notation ist auch in vielen Programmiersprachen üblich

Anstatt des deutschen Kommas wird im Englischen (und daher in fast allen Programmiersprachen) der Dezimalpunkt verwendet

Größenordnungen:

Bytes	Bits	Wertebereich	Name des Datentyps
4	32	$\pm 1.4e-45 \dots 3.403e38$	float
8	64	$\pm 4.94e-324 \dots 1.798e308$	double

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Folien auf Englisch \(Slides in English\)](#)

[\[Impressum\]](#) , [\[Datenschutz\]](#) , []

Thorsten Thormählen 35 / 35

Technische Informatik I

Zeichenkodierung

Thorsten Thormählen

31. Oktober 2023

Teil 2, Kapitel 3

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Inhalt

ASCII-Code

ISO 8859

Unicode

Zeichenkodierung

Um Text auf einem Computer darzustellen, muss jeder Buchstabe binär kodiert werden

Je nachdem wie viele Bits pro Zeichen verwendet werden, können unterschiedlich viele verschiedene Zeichen abgelegt werden

Beispiel:

7 Bits: $2^7 = 128$ verschiedene Zeichen

8 Bits: $2^8 = 256$ verschiedene Zeichen

16 Bits: $2^{16} = 65536$ verschiedene Zeichen

ASCII-Code

Der ASCII-Code (American Standard Code for Information Interchange) ist eine 7-Bit-Zeichenkodierung, die 1963 von der American Standards Association (ASA) beschlossen wurde

Ein Zeichen wird jedoch immer als 1 Byte (=8 Bits) abgelegt, d.h. das höchstwertige (8.) Bit ist immer Null

Insgesamt gibt es 128 Zeichen, davon 95 druckbare und 33 Steuerzeichen

ASCII-Code

In folgender Tabelle sind alle 128 ASCII-Zeichen angegeben

Wird der Mauszeiger über ein Zeichen gehalten, wird eine kurze Beschreibung angezeigt

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

Beispiel: Das Zeichen "A" hat den Hexadezimalwert $(41)_{16}$

$$(41)_{16} = (01000001)_2 = (65)_{10}$$

[Quelle: [Wikipedia](#)]

Thorsten Thormählen 6 / 16

ASCII-Code

Die Steuerzeichen stammen aus einer Zeit, in der der ASCII-Code zur Steuerung von Fernschreibern (elektrisch angesteuerte Schreibmaschinen) verwendet wurde

Heutzutage habe viele dieser Steuerzeichen ihre Bedeutung verloren

Wichtig ist eigentlich nur noch das Steuerzeichen für eine neue Zeile: "LF" (Line Feed, ASCII $(0A)_{16}$)

Beim Betriebssystem Windows muss dem "Line Feed" Zeichen allerdings noch ein "Carriage Return" vorangestellt werden: "CR LF" (=ASCII $(0D)_{16}$ $(0A)_{16}$)

ISO 8859

Bei der ASCII-Codierung werden nur 7 der 8 Bits eines Bytes genutzt

Der restliche Zahlenbereich (128 bis 255) kann also für weitere Zeichen verwendet werden

Die *International Organization for Standardization* definiert in ISO 8859 insgesamt 15 ASCII-Erweiterungen

ISO 8859-1 enthält z.B. die für uns in Deutschland wichtigen Buchstaben: "ä", "ö", "ü", "ß"

ISO 8859-1	Westeuropäisch (Latin-1)
ISO 8859-2	Mitteleuropäisch (Latin-2)
ISO 8859-3	Südeuropäisch (Latin-3)
ISO 8859-4	Nordeuropäisch (Latin-4)
ISO 8859-5	Kyrillisch
ISO 8859-6	Arabisch
ISO 8859-7	Griechisch
ISO 8859-8	Hebräisch
ISO 8859-9	Türkisch (Latin-5)
ISO 8859-10	Nordisch (Latin-6)
ISO 8859-11	Thai
ISO 8859-12	verworfen
ISO 8859-13	Baltisch (Latin-7)
ISO 8859-14	Keltisch (Latin-8)
ISO 8859-15	Westeuropäisch (Latin-9)
ISO 8859-16	Südosteuropäisch (Latin-10)

Unicode

Bei der Verwendung von ISO 8859 zum Austausch von Texten kommt es immer wieder zu fehlerhaften Darstellungen von Zeichen. Dies passiert leicht, wenn Sender und Empfänger nicht die gleiche ISO 8859-x Norm zur Dekodierung verwenden

Ausserdem sind in ISO 8859 längst nicht alle Schriftzeichen aus den unterschiedlichsten Kulturkreisen erfasst

Die Bestrebung des *Unicode* ist es, eine einzige universelle Kodierung zu definieren, die alle relevanten Zeichen enthält

Der Unicode wurde von der ISO als ISO-10646 standardisiert

Unicode

Der Unicode besteht aus 17 Ebenen (darstellbar mit 5 Bits).

Jede Ebene hat 16 Bits und kann damit theoretisch $2^{16}=65536$ Zeichen kodieren

Insgesamt kann ein Unicode also $5+16=21$ Bits benötigen

Die meisten aktuell verwendeten Zeichen sind in Ebene 0, der Basic Multilingual Plane (BMP), zu finden

Ein Unicode Zeichen wird üblicherweise als ein "U+" und einer Hexadzimalzahl mit mindestens 4 Stellen angegeben

Beispiele:

U+00E4 für das "ä"

U+1300C für die ägyptische Hieroglyphe 

[Bildquelle: [Unicodeblock Ägyptische Hieroglyphen](#)]

Thorsten Thormählen 10 / 16

Unicode

Die Kodierung aller möglichen Schriftzeichen ist ein andauernder Prozess, d.h. die Anzahl der Zeichen wächst ständig

Ein Problem bei der Darstellung ist, dass die meisten Schriftarten nur eine kleine Untermenge der im Unicode definierten Zeichen bereit halten

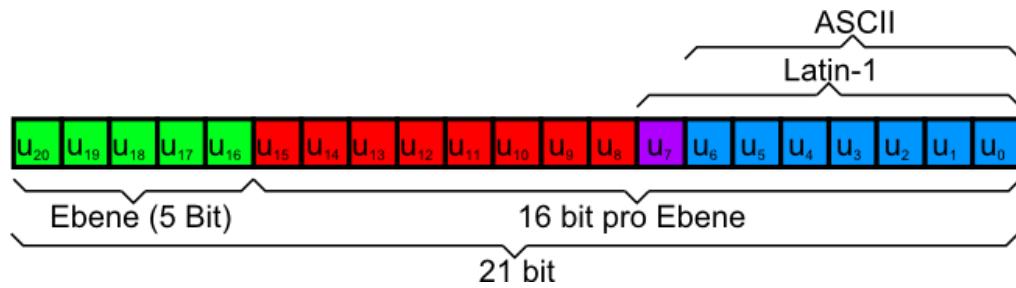
Ist ein Zeichen in einer Schrift nicht vorhanden, wird oftmals einfach ein Zeichen aus einer anderen Schriftart eingefügt

Die Webseite <http://www.decodeunicode.org/> hat es sich zur Aufgabe gemacht, alle aktuell im Unicode kodierten Zeichen darzustellen

Unicode

Beim Entwurf des Unicode wurde auf Kontinuität wert gelegt

Aus den 21 Bits des Unicodes entsprechen die ersten 7 Bits dem ASCII-Code und die ersten 8 Bits der ASCII-Erweiterungen ISO 8859-1 (Latin 1)



UTF

Unicode	
0x0000000-0x10FFFF	
UTF 32	
0x0000000-0x10FFFF	
UTF 16	
0x0000000-0x00FFFF	
0x0100000-0x10FFFF	
UTF 8	
0x0000000-0x00007F	
0x000080-0x0007FF	
0x000800-0x00FFFF	
0x0100000-0x10FFFF	

Zur Kodierung von Unicode-Zeichen wird meistens das UTF "Universal Transformation Format" verwendet

UTF-32 kodiert jedes Unicode-Zeichen mit 32 Bits, indem es die 21 Unicode Bits mit Nullen auffüllt

UTF-16 kodiert alle Bits der Basic Multilingual Plane (BMP) mit 16 Bits, nur für die anderen Ebenen werden 32 Bits benötigt

UTF-8

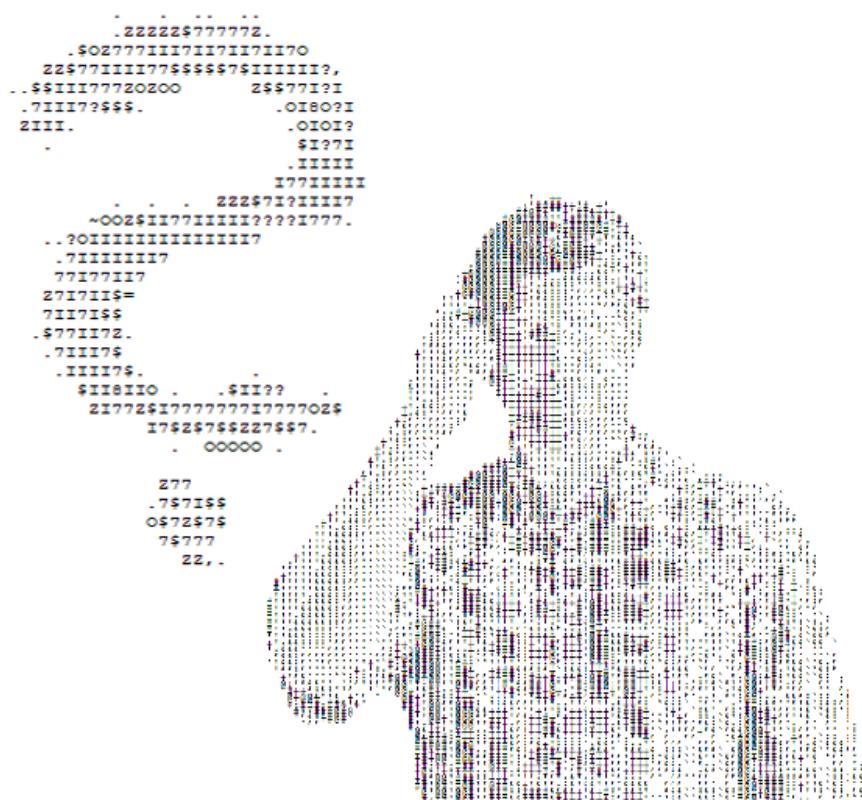
Unicode	
0x0000000-0x10FFFF	
UTF 32	
0x0000000-0x10FFFF	
UTF 16	
0x0000000-0x00FFFF	
0x0100000-0x10FFFF	
UTF 8	
0x0000000-0x00007F	
0x000080-0x0007FF	
0x000800-0x00FFFF	
0x0100000-0x10FFFF	

UTF-8 kodiert die ersten 7 Unicode Bits (entspricht ASCII) mit 8 Bits, die ersten 11 Unicode Bits mit 16 Bits, usw.

Ein UTF-8 kodierter Text, der nur ASCII Zeichen enthält, ist demnach vollständig mit ASCII kompatibel

UTF-8 ist heutzutage (besonders im Internet) weit verbreitet (Quasi-Standard der Zeichenkodierung)

Gibt es Fragen?



Thorsten Thormählen 15 / 16

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) .]

Thorsten Thormählen 16 / 16

Technische Informatik

Boolesche Algebra

Thorsten Thormählen

02. November 2023

Teil 3, Kapitel 1

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Schaltkreise und Wahrheitstafeln

Boolesche Algebra

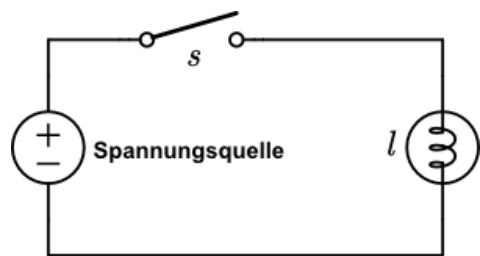
Schaltalgebra

Boolesche Funktionen

Boolesche Ausdrücke

Schaltkreise und Wahrheitstafeln

Betrachten wir einen einfachen Stromkreis bestehend aus einer Spannungsquelle (z.B. einer Batterie), einem Schalter s , und einer Lampe l



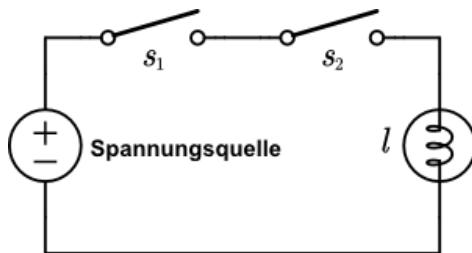
Ist der Schalter offen ($s=0$), brennt die Lampe nicht ($l=0$). Ist der Schalter geschlossen ($s=1$), brennt die Lampe ($l=1$).

Das Verhalten des Stromkreises kann in einer Wahrheitstafel dargestellt werden

s	l
0	0
1	1

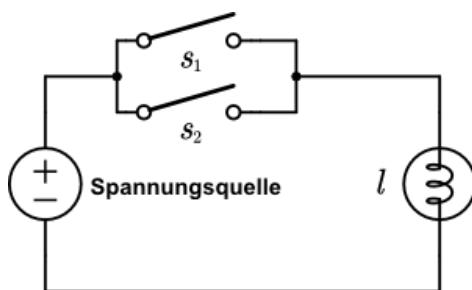
Schaltkreise und Wahrheitstafeln

Durch eine Reihenschaltung kann eine logische UND-Verknüpfung realisiert werden
(Licht brennt, wenn $s_1=1$ UND $s_2=1$)



s_1	s_2	l
0	0	0
0	1	0
1	0	0
1	1	1

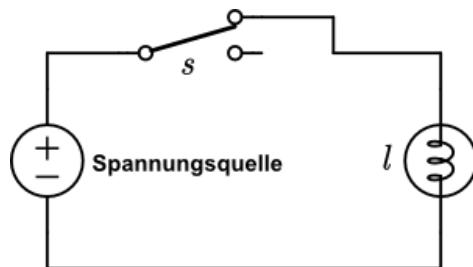
Durch eine Parallelschaltung kann eine logische ODER-Verknüpfung realisiert werden (Licht brennt, wenn $s_1=1$ ODER $s_2=1$)



s_1	s_2	l
0	0	0
0	1	1
1	0	1
1	1	1

Schaltkreise und Wahrheitstafeln

Durch umgekehrten Anschluss der Leitung kann eine logisches NICHT realisiert werden (Licht brennt, wenn $s=0$)



s	l
0	1
1	0

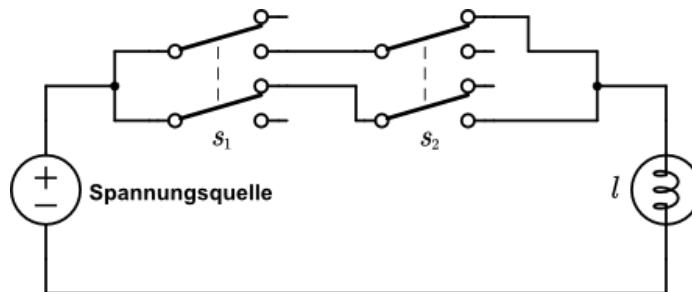
Damit haben wir schon die Grundelemente UND, ODER, NICHT (Englisch: AND, OR, NOT) kennengelernt, mit denen sich im Prinzip jeder mögliche Schaltkreis aufbauen lässt

Schaltkreise und Wahrheitstafeln

Wie könnte ein Schaltkreis aussehen, der folgende Wahrheitstabelle realisiert?

s_1	s_2	l
0	0	0
0	1	1
1	0	1
1	1	0

Lösung: $(s_1=0 \text{ AND } s_2=1) \text{ OR } (s_1=1 \text{ AND } s_2=0)$



Wir haben damit übrigens gerade das "EXKLUSIVE ODER" (Englisch: XOR) implementiert

Schaltkreise und Wahrheitstafeln

Wie viele mögliche Funktionen zweier Eingangsvariablen gibt es eigentlich?



Antwort:

Es gibt 16 mögliche Funktionen zweier Eingangsvariablen:

s_1	s_2	16 mögliche Funktionen l_0 bis l_{15}															
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Im Allgemeinen gibt es bei n Eingangsvariablen $2^{(2^n)}$ mögliche Funktionen

Boolesche Algebra

Definition mittels Huntington'scher Axiome:

Sei \mathcal{V} eine nicht leere Menge von Elementen mit den binären Operatoren $\circ : \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V}$ und $\bullet : \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V}$.

Das Tripel $(\circ, \bullet, \mathcal{V})$ heißt *Boolesche Algebra*, wenn für beliebige $a, b, c \in \mathcal{V}$ folgende Axiome gelten:

	Operator \bullet	Operator \circ
kommutativ	$a \bullet b = b \bullet a$	$a \circ b = b \circ a$
distributiv	$a \bullet (b \circ c) = (a \bullet b) \circ (a \bullet c)$	$a \circ (b \bullet c) = (a \circ b) \bullet (a \circ c)$
Identität	$a \bullet 1 = a$	$a \circ 0 = a$
Inversion	$a \bullet a^{-1} = 0$	$a \circ a^{-1} = 1$

Dabei müssen die Elemente 0 und 1 jeweils ein bestimmtes ausgezeichnetes Element aus \mathcal{V} sein

Boolesche Algebra

Ist der Körper der reellen Zahlen mit der Multiplikation und Addition $(\cdot, +, \mathbb{R})$ eine boolesche Algebra?

Antwort: Nein, einige Axiome gelten nicht

	Operator \cdot	Operator $+$
kommutativ	$a \cdot b = b \cdot a$	$a + b = b + a$
distributiv	$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$	$a + (b \cdot c) = (a + b) \cdot (a + c)$
Identität	$a \cdot 1 = a$	$a + 0 = a$
Inversion	$a \cdot a^{-1} = 0$	$a + a^{-1} = 1$

Schaltalgebra

Die Schaltalgebra ist eine boolesche Algebra über dem Tripel $(\wedge, \vee, \mathcal{B})$ mit
der Grundmenge $\mathcal{B} = \{0, 1\}$
der AND-Verknüpfung (Konjunktionsoperator): " \wedge "
der OR-Verknüpfung (Disjunktionsoperator): " \vee "
und dem inversen Element: der NOT-Operation (Negation): " \neg "

Es gilt somit laut Definition:

	Operator \wedge	Operator \vee
kommutativ	$a \wedge b = b \wedge a$	$a \vee b = b \vee a$
distributiv	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
Identität	$a \wedge 1 = a$	$a \vee 0 = a$
Inversion	$a \wedge \neg a = 0$	$a \vee \neg a = 1$

Schaltalgebra

Die Schaltalgebra besteht somit aus folgenden elementaren Grundoperationen

Konjunktion (AND): $y = a \wedge b$

a	b	$y = a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

Disjunktion (OR): $y = a \vee b$

a	b	$y = a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Negation (NOT): $y = \neg a$

a	$y = \neg a$
0	1
1	0

Boolesche Funktionen (Schaltfunktion)

Eine Funktion $y = f(x_1, x_2, \dots, x_n)$ mit $x_1, x_2, \dots, x_n \in \{0, 1\}$ heisst boolesche Funktion der Stelligkeit n

Boolesche Funktionen werden auch Schaltfunktionen genannt

Die Eingabevariablen x_1, x_2, \dots, x_n werden als *freie* Variablen bezeichnet

Die AusgabevARIABLE y wird *abhängige* Variable genannt

Beispiele:

Der Negationsoperator $y = \neg x_1$ ist eine boolesche Funktion der Stelligkeit 1

Konjunktionsoperator $y = x_1 \wedge x_2$ und Disjunktionsoperator $y = x_1 \vee x_2$ sind boolesche Funktionen der Stelligkeit 2

[Quelle: D.W. Hoffmann: Grundlagen der Technischen Informatik, 2. Auflage; Hanser 2009]

Thorsten Thormählen 14 / 31

Boolesche Funktionen (Schaltfunktion)

Eine boolesche Funktion kann eindeutig durch eine vollständige Wahrheitstafel beschrieben werden

Beispiele:

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

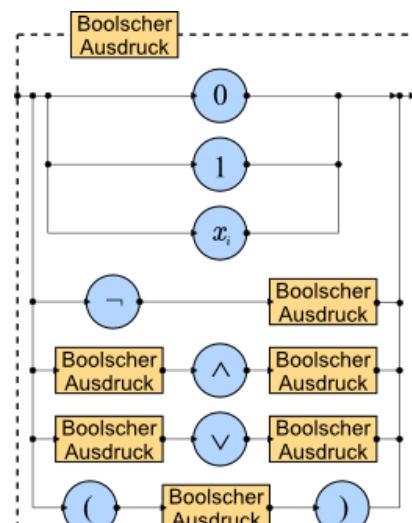
x_1	x_2	x_3	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Boolesche Ausdrücke

Die Darstellung durch Wahrheitstafeln wird schnell sehr groß und unübersichtlich

Daher werden boolesche Funktionen häufig in Formeldarstellung angegeben

Diese Formeldarstellungen werden *Boolesche Ausdrücke* genannt und sind durch das gezeigte Syntaxdiagramm definiert



Syntaxdiagramm

[Quelle: D.W. Hoffmann: Grundlagen der Technischen Informatik, 2. Auflage; Hanser 2009, Abbildung 4.5]

Thorsten Thormählen 16 / 31

Boolesche Ausdrücke

Die Darstellung einer booleschen Funktion mittels boolescher Ausdrücke ist nicht eindeutig

Beispiel: Die Funktion

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

kann durch viele verschiedene boolesche Ausdrücke beschrieben werden:

$$y = (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$$

$$y = (x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$$

$$y = \dots$$

Durch die Theoreme auf der nächsten Folie können verschiedene boolesche Ausdrücke in einander überführt werden

Liste der Axiome und Theoreme

Aus den Axiomen der booleschen Algebra lassen sich weitere Theoreme ableiten

#	Name	AND-Operator \wedge	OR-Operator \vee
1	Identität	$a \wedge 1 = a$	$a \vee 0 = a$
2	Elimination	$a \wedge 0 = 0$	$a \vee 1 = 1$
3	Idempotenz	$a \wedge a = a$	$a \vee a = a$
4	Involution	$\neg(\neg a) = a$	
5	Komplement	$a \wedge \neg a = 0$	$a \vee \neg a = 1$
6	Kommutativität	$a \wedge b = b \wedge a$	$a \vee b = b \vee a$
7	Assoziativität	$(a \wedge b) \wedge c = a \wedge (b \wedge c)$	$(a \vee b) \vee c = a \vee (b \vee c)$
8	Distributivität	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
9	Vereinigung	$(a \vee b) \wedge (a \vee \neg b) = a$	$(a \wedge b) \vee (a \wedge \neg b) = a$
10	Absorption	$a \wedge (a \vee b) = a$	$a \vee (a \wedge b) = a$
11	Absorption (2)	$(a \wedge \neg b) \vee b = a \vee b$	$(a \vee \neg b) \wedge b = a \wedge b$
12	Faktorisierung	$(a \vee b) \wedge (\neg a \vee c) = (a \wedge c) \vee (\neg a \wedge b)$	$(a \wedge b) \vee (\neg a \vee c) = (a \vee c) \wedge (\neg a \vee b)$
13	Konsens	$(a \vee b) \wedge (b \vee c) \wedge (\neg a \vee c) = (a \vee b) \wedge (\neg a \vee c)$	$(a \wedge b) \vee (b \wedge c) \vee (\neg a \wedge c) = (a \wedge b) \vee (\neg a \wedge c)$
14	de Morgans Gesetz	$\neg(a \wedge b \wedge \dots) = \neg a \vee \neg b \vee \dots$	$\neg(a \vee b \vee \dots) = \neg a \wedge \neg b \wedge \dots$

Dualität

Von den aufgelisteten Theoremen muss sich nur eine Spalte der Tabelle gemerkt werden. Die andere Spalte ergibt sich direkt durch die *Dualität*

Der duale boolesche Ausdruck f_d kann aus f ermittelt werden, indem \vee mit \wedge und 0 mit 1 vertauscht wird

$$f(x_1, x_2, \dots, x_n, 0, 1, \wedge, \vee) \Leftrightarrow f_d = f(x_1, x_2, \dots, x_n, 1, 0, \vee, \wedge)$$

De Morgansche Gesetze

Die Gesetze von August De Morgan (*1806; †1871) lauten

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

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

Die Gültigkeit der Gesetze kann leicht anhand von Wahrheitstafeln gezeigt werden



Augustus De Morgan

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

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

Negationstheorem

Theorem #14 aus unserer Liste ist eine Verallgemeinerung der De Morganschen Gesetze:

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

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

Noch allgemeiner formuliert es das Negationstheorem:

$$\neg(f(x_1, x_2, \dots, x_n, 0, 1, \wedge, \vee)) = f(\neg x_1, \neg x_2, \dots, \neg x_n, 1, 0, \vee, \wedge)$$

Beispiel:

$$\neg((a \wedge 1) \vee b) = (\neg a \vee 0) \wedge \neg b$$

Weitere Operatoren

Neben den elementaren logischen Operationen (AND, OR, NOT) gibt es noch weitere wichtige Operatoren

Implikation und Inverse Implikation

Äquivalenz und Antivalenz

Sheffer-Funktion (NAND) und Peirce-Funktion (NOR)

Diese Operationen könnten jedoch alle auch mit den elementaren Operationen realisiert werden

Sie verkürzen lediglich die Darstellung

Implikation und Inverse Implikation

Implikation: $y = a \rightarrow b = \neg a \vee b$

Gültigkeit der Aussage: "a ist eine hinreichende Bedingung für b".

Beispiel: "Dass die Sonne scheint, resultiert zwangsläufig (hinreichende Bedingung) in Sonnenbrand"

Diese Aussage ist nur falsch, wenn die Sonne scheint und trotzdem kein Sonnenbrand auftritt

a	b	$y = a \rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

Inverse Implikation: $y = a \leftarrow b = \neg b \vee a$

a	b	$y = a \leftarrow b$
0	0	1
0	1	0
1	0	1
1	1	1

Äquivalenz und Antivalenz

$$\text{Äquivalenz: } y = a \leftrightarrow b = (\neg a \wedge \neg b) \vee (a \wedge b)$$

Wahr, wenn beide Operanden den gleichen Wahrheitswert besitzen

a	b	$y = a \leftrightarrow b$
0	0	1
0	1	0
1	0	0
1	1	1

$$\text{Antivalenz: } y = a \leftrightarrow b = (\neg a \wedge b) \vee (a \wedge \neg b)$$

Wahr, wenn beide Operanden verschiedene Wahrheitswerte besitzen

Exklusives Oder (XOR)

a	b	$y = a \leftrightarrow b$
0	0	0
0	1	1
1	0	1
1	1	0

Sheffer-Funktion und Peirce-Funktion

Sheffer-Funktion: $y = a \bar{\wedge} b = \neg(a \wedge b)$

Konjunktion mit anschließender Negation (NAND)

a	b	$y = a \bar{\wedge} b$
0	0	1
0	1	1
1	0	1
1	1	0

Peirce-Funktion: $y = a \bar{\vee} b = \neg(a \vee b)$

Disjunktion mit anschließender Negation (NOR)

a	b	$y = a \bar{\vee} b$
0	0	1
0	1	0
1	0	0
1	1	0

Operatorsymbole der Schaltalgebra nach DIN 66000

Operator	Benennung	Beispiel	Sprechweise
\neg $-$	Negation, NOT	$y = \neg a = \bar{a}$	nicht a
\wedge	Konjunktion, AND	$y = a \wedge b$	a und b
\vee	Disjunktion, OR	$y = a \vee b$	a oder b
\rightarrow	Implikation	$y = a \rightarrow b$	a impliziert b
\leftrightarrow	Äquivalenz	$y = a \leftrightarrow b$	a äquivalent b
\leftrightarrow	Antivalenz, XOR	$y = a \leftrightarrow b$	a xor b
$\bar{\wedge}$	Sheffer-Funktion, NAND	$y = a \bar{\wedge} b$	a nand b
$\bar{\vee}$	Peirce-Funktion, NOR	$y = a \bar{\vee} b$	a nor b

Thorsten Thormählen 26 / 31

Operatorsymbole der Schaltalgebra nach DIN 66000

Gemäß DIN 66000 besitzen die Symbole folgende Vorrangregeln:

Stärkste Bindung: \neg

Mittlere Bindung: $\wedge, \vee, \bar{\wedge}, \bar{\vee}$

Niedrige Bindung: $\rightarrow, \leftrightarrow, \Leftrightarrow$

Symbole mit gleicher Bindungsstärke werden linksassoziativ ausgewertet

Beispiele:

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

$$y = a \wedge b \wedge c = (a \wedge b) \wedge c$$

$$y = a \vee b \vee c = (a \vee b) \vee c$$

$$y = a \vee b \wedge c = (a \vee b) \wedge c$$

$$y = a \leftrightarrow \neg b \vee \neg c \wedge d = a \leftrightarrow (((\neg b) \vee (\neg c)) \wedge d)$$

Operatorsymbole der Schaltalgebra, US-Schreibweise

Operator	Benennung	Beispiel	Sprechweise
-	negation, NOT	$y = \bar{a}$	not a
.	conjunction, AND	$y = a \cdot b = ab$	a and b
+	disjunction, OR	$y = a + b$	a or b
\Rightarrow	implication	$y = a \Rightarrow b$	a implies b
$\begin{array}{c} \equiv \\ \oplus \end{array}$	equivalence, (E)XNOR	$y = a \equiv b = \overline{a \oplus b}$	a exnor b
\oplus	exclusive disjunction, (E)XOR	$y = a \oplus b$	a exor b
$\overline{\cdot}$	NAND	$y = \overline{a \cdot b} = \overline{ab}$	a nand b
$\overline{+}$	NOR	$y = \overline{a + b}$	a nor b

Abgekürzte Schreibweise boolescher Ausdrücke

In dieser Vorlesung wird eigentlich durchgängig die DIN-Schreibweise verwendet

Ein entscheidender Unterschied zwischen DIN und US-Schreibweise ist die Bindung der UND-Verknüpfung

Während bei der DIN-Norm UND- sowie ODER-Verknüpfung gleichrangig sind, gilt in der US-Schreibweise analog zur Algebra "Punkt- vor Strichrechnung"

US-Schreibweise: $a + b \cdot c = a + (b \cdot c) = a + bc$

DIN-Schreibweise: $a \vee b \wedge c = (a \vee b) \wedge c$

Abweichend von der DIN-Norm wird in dieser Vorlesung (und den Übungen) analog zur US-Version teilweise die verkürzte Schreibweise $ab = (a \wedge b)$ angewendet, um die Ausdrücke lesbarer zu machen

Ausschließlich diese verkürzte Schreibweise der UND-Verknüpfung soll eine höhere Bindung haben, als die anderen Operatoren. Die stärkste Bindung hat weiterhin die Negation.

DIN-Schreibweise: $(a \wedge b \wedge \neg(c \wedge d)) \vee (\neg e \wedge f) = (a \wedge b \wedge \overline{c \wedge d}) \vee (\bar{e} \wedge f)$

Abgekürzte Schreibweise: $abcd \vee \bar{e}f$

Zusammenfassung: Alle zweistelligen Schaltfunktionen

a	0011	
b	0101	
$y = 0$	0000	Nullfunktion
$y = a \wedge b$	0001	Konjunktion
$y = a \wedge \neg b$	0010	
$y = a$	0011	
$y = \neg a \wedge b$	0100	
$y = b$	0101	
$y = a \leftrightarrow b$	0110	Antivalenz
$y = a \vee b$	0111	Disjunktion
$y = a \bar{\vee} b$	1000	Peirce-Funktion, NOR
$y = a \leftrightarrow b$	1001	Äquivalenz
$y = \neg b$	1010	
$y = a \leftarrow b$	1011	Inverse Implikation
$y = \neg a$	1100	
$y = a \rightarrow b$	1101	Implikation
$y = a \bar{\wedge} b$	1110	Sheffer-Funktion, NAND
$y = 1$	1111	Einsfunktion

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) .

Thorsten Thormählen 31 / 31

Technische Informatik

Rechnen mit booleschen Ausdrücken

Thorsten Thormählen
08. November 2022
Teil 3, Kapitel 2

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Beweise durch vollständige Wahrheitstafeln

Beweise durch Umformung boolescher Ausdrücke

Vereinfachung durch Umformung boolescher Ausdrücke

Literale und Monome

Beweise durch vollständige Wahrheitstafeln

Durch die Verwendung vollständiger Wahrheitstafeln können leicht Aussagen bewiesen werden

Beispiel:

Beweise das Vereinigungstheorem (9): $xy \vee x\bar{y} = x$

x	y	xy	$x\bar{y}$	$xy \vee x\bar{y}$
0	0	0	0	0
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1

Beweise durch Umformung

Die Verwendung von Wahrheitstafel ist für größere Funktionen aufwendig

Durch Verwendung der Liste der Axiome und Theoreme aus dem letzten Kapitel können ebenfalls Aussagen bewiesen werden

Beispiel 1:

Beweise das Vereinigungstheorem (9): $xy \vee x\bar{y} = x$

$$\begin{aligned} & xy \vee x\bar{y} \\ & \stackrel{(8)}{=} x \wedge (y \vee \bar{y}) \\ & \stackrel{(5)}{=} x \wedge 1 \\ & \stackrel{(1)}{=} x \end{aligned}$$

Beweise durch Umformung

Beispiel 2:

Beweise das Theorem der Absorption (10): $x \vee xy = x$

$$\begin{aligned} & x \vee xy \\ \stackrel{(1)}{=} & (x \wedge 1) \vee xy \\ \stackrel{(8)}{=} & x \wedge (1 \vee y) \\ \stackrel{(2)}{=} & x \wedge 1 \\ \stackrel{(1)}{=} & x \end{aligned}$$

Die Klammer über dem Gleichheitszeichen gibt jeweils das bei der Umformung verwendete Axiom an (siehe "Liste der Axiome und Theoreme" aus dem letzten Kapitel)

Beweise durch Umformung

Beispiel 3:

Beweise das Theorem der Involution (4): $\neg(\neg x) = x$

$$\begin{aligned}\neg(\neg x) &= \neg \bar{x} \\ \stackrel{(1)}{=} &\neg \bar{x} \wedge 1 \\ \stackrel{(5)}{=} &\neg \bar{x} \wedge (x \vee \bar{x}) \\ \stackrel{(8)}{=} &(\neg \bar{x} \wedge x) \vee (\neg \bar{x} \wedge \bar{x}) \\ \stackrel{(6)}{=} &(\neg \bar{x} \wedge x) \vee (\bar{x} \wedge \neg \bar{x}) \\ \stackrel{(5)}{=} &(\neg \bar{x} \wedge x) \vee 0 \\ \stackrel{(5)}{=} &(\neg \bar{x} \wedge x) \vee (x \wedge \bar{x}) \\ \stackrel{(6)}{=} &(\neg \bar{x} \wedge x) \vee (\bar{x} \wedge x) \\ \stackrel{(8)}{=} &(\neg \bar{x} \vee \bar{x}) \wedge x \\ \stackrel{(5)}{=} &1 \wedge x \\ \stackrel{(1)}{=} &x\end{aligned}$$

[Quelle D.W. Hoffmann: *Grundlagen der Technischen Informatik*, 2. Auflage; Hanser 2009, S. 122]

Vereinfachung durch Umformung

Beispiel 1:

$$\begin{aligned}& (x \leftrightarrow y) \vee (x \leftrightarrow y) \\&= (\neg x \wedge y) \vee (x \wedge \neg y) \vee (\neg x \wedge \neg y) \vee (x \wedge y) \\&= (\bar{x} \wedge y) \vee (x \wedge \bar{y}) \vee (\bar{x} \wedge \bar{y}) \vee (x \wedge y) \\&\stackrel{(6)}{=} ((\bar{x} \wedge y) \vee (\bar{x} \wedge \bar{y})) \vee ((x \wedge \bar{y}) \vee (x \wedge y)) \\&\stackrel{(8)}{=} (\bar{x} \wedge (y \vee \bar{y})) \vee (x \wedge (\bar{y} \vee y)) \\&\stackrel{(5)}{=} (\bar{x} \wedge 1) \vee (x \wedge 1) \\&\stackrel{(1)}{=} \bar{x} \vee x \\&\stackrel{(5)}{=} 1\end{aligned}$$

Der boolesche Ausdruck $(x \leftrightarrow y) \vee (x \leftrightarrow y)$ ist demnach immer gleich 1.

Eine solcher Ausdruck heißt *allgemeingültig*

[Quelle D.W. Hoffmann: *Grundlagen der Technischen Informatik*, 2. Auflage; Hanser 2009, S. 124]

Thorsten Thormählen 9 / 18

Vereinfachung durch Umformung

Beispiel 2:

$$\begin{aligned}(x \leftrightarrow y) &\wedge ((x \wedge \bar{y}) \vee y) \\&= (\bar{x} \wedge y) \vee (x \wedge \bar{y}) \wedge ((x \wedge \bar{y}) \vee y) \\&\stackrel{(6)}{=} (x \wedge \bar{y}) \vee (\bar{x} \wedge y) \wedge ((x \wedge \bar{y}) \vee y) \\&\stackrel{(7)}{=} ((x \wedge \bar{y}) \vee (\bar{x} \wedge y)) \wedge ((x \wedge \bar{y}) \vee y) \\&\stackrel{(8)}{=} (x \wedge \bar{y}) \vee ((\bar{x} \wedge y) \wedge y) \\&\stackrel{(7)}{=} (x \wedge \bar{y}) \vee (\bar{x} \wedge (y \wedge y)) \\&\stackrel{(3)}{=} (x \wedge \bar{y}) \vee (\bar{x} \wedge y) \\&\stackrel{(6)}{=} (\bar{x} \wedge y) \vee (x \wedge \bar{y}) \\&= (x \leftrightarrow y)\end{aligned}$$

[Quelle D.W. Hoffmann: *Grundlagen der Technischen Informatik*, 2. Auflage; Hanser 2009, S. 124]

Thorsten Thormählen 10 / 18

Tipps zum Rechnen mit booleschen Ausdrücken

Abkürzende Schreibweise erst verwenden, wenn Sie Sicherheit mit booleschen Ausdrücken erlangt haben

Beim Vereinfachen des Ausdrucks von den innersten geklammerten Ausdrücken nach außen vorgehen. Eventuell Klammerpaare farblich markieren.

Immer zunächst in folgender Reihenfolge prüfen, ob sich der Ausdruck vereinfachen lässt:

Absorption (10)

Absorption II (11)

Vereinigung (9)

Besonders Distributivität (8) erst nach Prüfen aller anderen Möglichkeiten verwenden, wenn sich dadurch der Ausdruck verlängert

Literale und Monome

Ein *Literal* ist eine Variable oder eine negierte Variable, z.B. $a, b, \neg c, \bar{x}$, etc.

Ein *Monom* ist eine UND-Verknüpfung von Literalen, z.B. $(a \wedge b \wedge \neg c) = ab\bar{c}, \bar{f}\bar{e}\bar{d}, \bar{x}\bar{y}, xyz$ etc.

Die Schaltfunktion eines Monoms kann nur dann 1 sein, wenn jedes vorkommende Literal 1 ist, d.h. jede vorkommende Variable mit 1 und jede negierte Variable mit 0 belegt ist

Das Monom $\bar{x}y\bar{z}$ ist also nur dann 1, falls $x = z = 0$ und $y = 1$ sind

Literale und Monome

Ein boolescher Ausdruck, der aus einer ODER-Verknüpfung von Monomen besteht, evaluiert genau dann zu 1, wenn mindestens eines der Monome zu 1 evaluiert

So evaluiert der Ausdruck $y = (\bar{x}yz \vee x\bar{y}z)$ zu 1, wenn $x = z = 0$ und $y = 1$
oder $x = z = 1$ und $y = 0$

Durch die ODER-Verknüpfung von Monomen kann somit eine beliebige Schaltfunktion abgebildet werden

Literale und Monome

Beispiel: 1-bit binärer Halbaddierer

Eingänge: Summanden a und b

Ausgänge: Summe s , Carry-out c_{out} (Übertrag)



a	b	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Aus der Wahrheitstafel können in jeder Spalte, in der das Ergebnis 1 sein soll, die Monome abgelesen werden

Damit ergibt sich:

$$s = \bar{a}\bar{b} \vee a\bar{b} = a \leftrightarrow b$$

$$c_{\text{out}} = ab$$

Literale und Monome

Beispiel: 1-bit binärer Volladdierer

Eingänge: Summanden a und b , Carry-in c_{in}

Ausgänge: Summe s , Carry-out c_{out}



Volladdierer

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Damit ergibt sich:

$$s = \bar{a}\bar{b}c_{\text{in}} \vee \bar{a}b\bar{c}_{\text{in}} \vee a\bar{b}\bar{c}_{\text{in}} \vee abc_{\text{in}}$$

$$c_{\text{out}} = \bar{a}bc_{\text{in}} \vee a\bar{b}c_{\text{in}} \vee ab\bar{c}_{\text{in}} \vee abc_{\text{in}}$$

Literale und Monome

Als Beispiel soll nun der boolesche Ausdruck für das c_{out} des Volladdierers vereinfacht werden:

$$\begin{aligned} c_{\text{out}} &= \bar{a}bc_{\text{in}} \vee a\bar{b}c_{\text{in}} \vee ab\bar{c}_{\text{in}} \vee abc_{\text{in}} \\ &\stackrel{(3)}{=} \bar{a}bc_{\text{in}} \vee a\bar{b}c_{\text{in}} \vee ab\bar{c}_{\text{in}} \vee abc_{\text{in}} \vee abc_{\text{in}} \\ &\stackrel{(6)}{=} (\bar{a}bc_{\text{in}} \vee abc_{\text{in}}) \vee (a\bar{b}c_{\text{in}} \vee abc_{\text{in}}) \vee (ab\bar{c}_{\text{in}} \vee abc_{\text{in}}) \\ &\stackrel{(8)}{=} ((\bar{a} \vee a)bc_{\text{in}}) \vee (a(\bar{b} \vee b)c_{\text{in}}) \vee (ab(\bar{c}_{\text{in}} \vee c_{\text{in}})) \\ &\stackrel{(5)}{=} ((1)bc_{\text{in}}) \vee (a(1)c_{\text{in}}) \vee (ab(1)) \\ &\stackrel{(1)}{=} bc_{\text{in}} \vee ac_{\text{in}} \vee ab \end{aligned}$$

Quiz

Frage: Ist der boolesche Ausdruck
 $(a \wedge b) \vee (a \wedge \neg b) \vee \neg a$
allgemeingültig?

Antwort 1: **Nein**

Antwort 2: **Ja**

Antwort 3: **keine Ahnung**

Am Online-Quiz teilnehmen durch Besuch der Webseite:
www.onlineclicker.org

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[\[Impressum\]](#) , [\[Datenschutz\]](#) , []

Thorsten Thormählen 18 / 18

Technische Informatik

Logikgatter

Thorsten Thormählen
08. November 2022
Teil 3, Kapitel 3

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

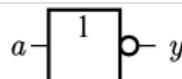
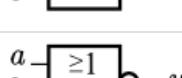
Inhalt

Gattersymbole

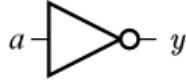
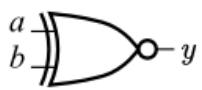
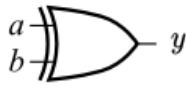
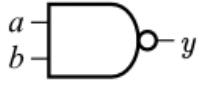
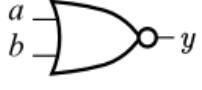
Realisierung einer booleschen Funktion durch Logikgatter

Praxisbeispiel: Aufbau einer Logikschaltung mit ICs

Gattersymbole nach DIN 40900

Benennung	Boolescher Ausdruck	Gattersymbol
Negation, NOT	$y = \neg a = \bar{a}$	
Konjunktion, AND	$y = a \wedge b$	
Disjunktion, OR	$y = a \vee b$	
Äquivalenz	$y = a \leftrightarrow b$	
Antivalenz, XOR	$y = a \leftrightarrow b$	
Sheffer-Funktion, NAND	$y = a \bar{\wedge} b$	
Peirce-Funktion, NOR	$y = a \bar{\vee} b$	

Gattersymbole, US ANSI 91

Benennung	Boolescher Ausdruck	Gattersymbol
negation, NOT	$y = \bar{a}$	
conjunction, AND	$y = a \cdot b = ab$	
disjunction, OR	$y = a + b$	
equivalence, (E)XNOR	$y = a \equiv b = \overline{a \oplus b}$	
exclusive disjunction, (E)XOR	$y = a \oplus b$	
NAND	$y = \overline{a \cdot b} = \overline{ab}$	
NOR	$y = \overline{a + b}$	

Gattersymbole

In dieser Vorlesung werden durchgängig die DIN-Symbole verwendet

Die neuere Empfehlung der International Electrotechnical Commission in IEC 60617-12 entspricht im Wesentlichen den rechteckigen DIN-Symbolen

Der US-Standard ist in der Praxis (besonders in der englischsprachigen Literatur) jedoch sehr häufig zu finden

Darstellung von Schaltnetzen mit Gattern

Um aus mehreren Gattern Schaltnetze aufzubauen, können Ein- und Ausgänge von Gattern mit durchgezogenen rechtwinklig verlaufenden Leitungen verbunden werden

Bei einer rechtwinkligen Verzweigung wird davon ausgegangen, dass eine Verbindung zwischen den Leitungen besteht

Dies kann zusätzlich mit einem ausgefüllten Punkt verdeutlicht werden

Um verbundene Kreuzungen zu kennzeichnen, ist der ausgefüllte Punkt unbedingt erforderlich, ansonsten werden die Leitungen als nicht verbunden interpretiert.

Dass zwei sich kreuzende Leitungen nicht verbunden sind, kann zusätzlich durch einen Halbkreis verdeutlicht werden

Verbunden



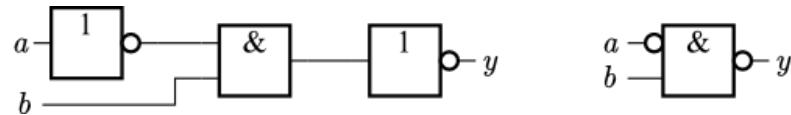
Nicht verbunden



Kompaktere Darstellung

Sowohl Ein- als auch Ausgänge lassen sich invertieren, indem sie mit dem (nicht ausgefüllten) Negationskreis versehen werden.

Beispiel: $y = \neg(\neg a \wedge b)$

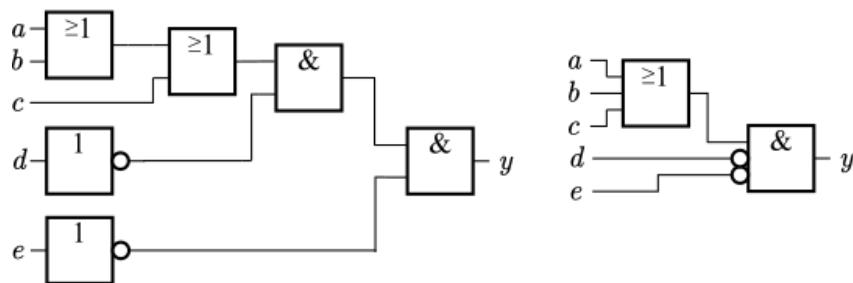


AND- und OR-Gatter können auch mehr als zwei Eingänge haben

Ein OR-Gatter mit n Eingängen realisiert den Ausdruck $y = x_1 \vee x_2 \vee \dots \vee x_n$

Ein AND-Gatter mit n Eingängen realisiert den Ausdruck $y = x_1 \wedge x_2 \wedge \dots \wedge x_n$

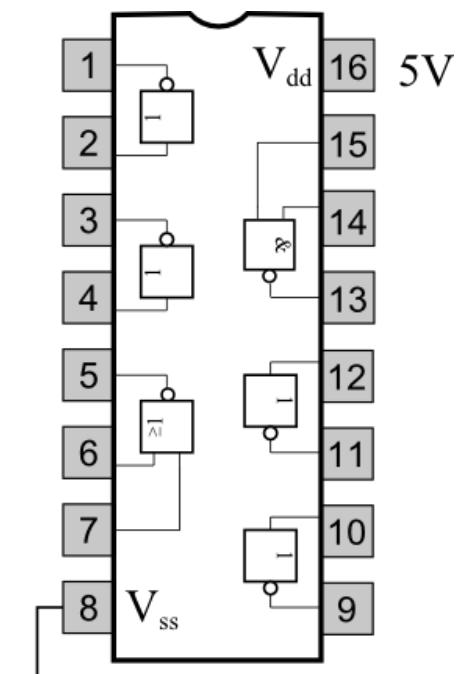
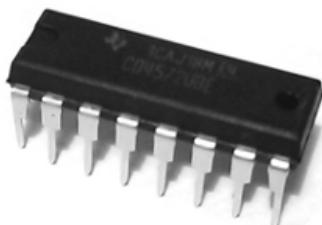
Beispiel: $y = (a \vee b \vee c) \wedge \neg d \wedge \neg e$



Praxisbeispiel: Aufbau einer Logikschaltung mit ICs

Im folgenden wollen wir eine reale Schaltung aufbauen, die einige Logikgatter enthält

Dazu verwenden wir den CMOS IC CD4572UB von Texas Instruments, der 4 Inverter, 1 NAND- und 1 NOR-Gatter bereitstellt



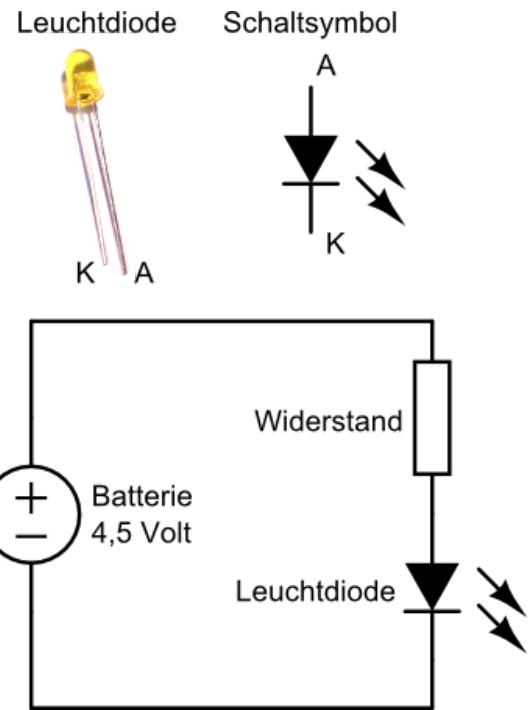
CD4572UB

Praxisbeispiel: Aufbau einer Logikschaltung mit ICs

Um die Zustände der Ein- und Ausgänge zu visualisieren, werden wir LEDs ("Light-Emitting Diodes") verwenden

Bei bedrahteten LEDs ist die Kathode (-) das kürzere Bein und das längere Bein die Anode (+)

Eine LED muss immer mit einem Vorwiderstand betrieben werden, der den Strom durch die LED einstellt



Das ohmsche Gesetz

Um die Größe des Widerstands zu berechnen, brauchen wir das ohmsche Gesetz

Das ohmsche Gesetz macht eine Aussage über Spannung und Stromstärke an einem Widerstand:

Spannung U : Kraft auf Ladungsträger, Einheit Volt [V]

Stromstärke I : durchfließende Ladungsträger pro Zeiteinheit, Einheit Ampere [A]

Das ohmsche Gesetz besagt, dass die Stromstärke I , die durch einen Widerstand R fließt, proportional zu der am Widerstand abfallenden Spannung U ist

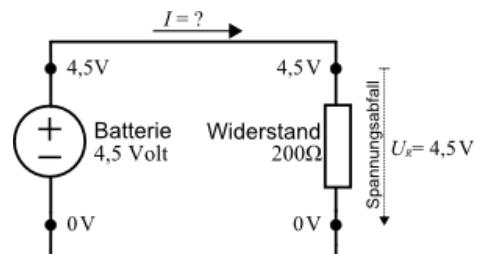
$$U = R \cdot I \Leftrightarrow I = \frac{U}{R} \Leftrightarrow R = \frac{U}{I}$$

Widerstand R : Proportionalitätsfaktor zwischen Spannung und Stromstärke, Einheit Ohm [Ω]

Wie groß ist die Stromstärke I in diesem Stromkreis?

Antwort:

$$I = \frac{U}{R} = \frac{4,5 \text{ V}}{200 \Omega} = 0,0225 \text{ A} = 22,5 \text{ mA}$$



Das ohmsche Gesetz

Spannungsteiler

Im oberen Stromkreis fließt überall der gleiche Strom I_0

Damit gilt laut ohmschen Gesetz:

$$I_0 = \frac{U_1}{R_1} \text{ und } I_0 = \frac{U_2}{R_2} \text{ und } I_0 = \frac{U_0}{R_1+R_2}$$

Durch Umformung ergibt sich für das Verhältnis der Spannungen U_1 und U_2 :

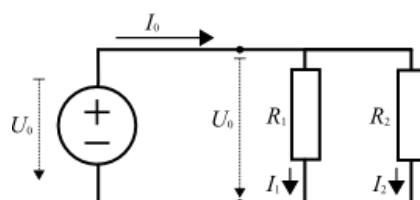
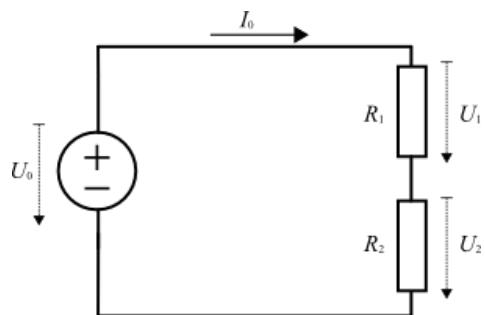
$$\frac{U_1}{R_1} = \frac{U_2}{R_2} \Leftrightarrow \frac{U_1}{U_2} = \frac{R_1}{R_2}$$

Stromteiler

Im unteren Stromkreis fällt an beiden Widerständen die gleich Spannung U_0 ab

Damit ergibt sich für das Verhältnis der Stromstärken I_1 und I_2 :

$$U_0 = R_1 \cdot I_1 = R_2 \cdot I_2 \Leftrightarrow \frac{I_1}{I_2} = \frac{R_2}{R_1}$$

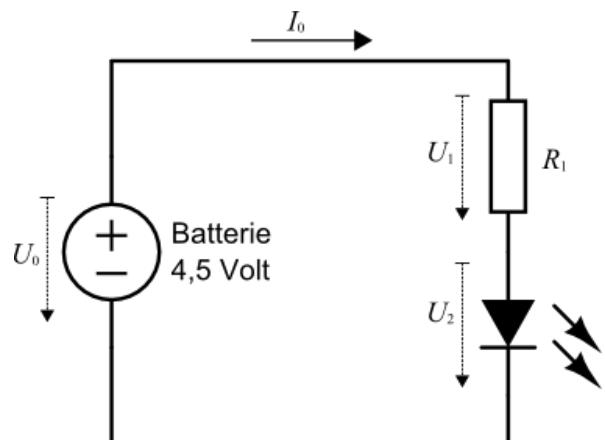


Berechnen des Vorwiderstands einer LED

Dem Datenblatt der von uns eingesetzten LED kann entnommen werden, dass sie mit einem Strom von $I_0 = 20 \text{ mA}$ betrieben werden soll. Laut Datenblatt fällt in diesem Fall $U_2 = 2,25 \text{ V}$ über der LED ab.

Bei einer Spannungsversorgung mit $U_0 = 4,5 \text{ V}$ ergibt sich somit für den Vorwiderstand R_1

$$\begin{aligned} I_0 &= \frac{U_1}{R_1} = \frac{U_0 - U_2}{R_1} \\ \Leftrightarrow R_1 &= \frac{U_0 - U_2}{I_0} \\ &= \frac{4,5 \text{ V} - 2,25 \text{ V}}{20 \text{ mA}} = 112,5 \Omega \end{aligned}$$



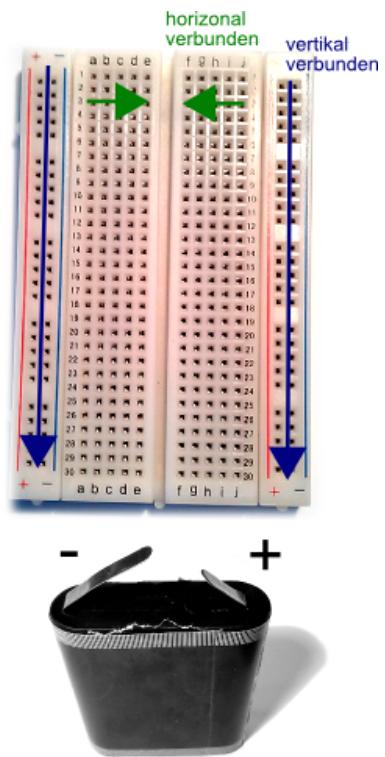
Praxisbeispiel: Aufbau einer Logikschaltung mit ICs

Die Schaltung soll mit einer Steckplatine realisiert werden

In den Spalten zur Spannungsversorgung ("+" bzw "-") sind die Steckplätze vertikal miteinander verbunden

Ansonsten sind die Steckplätze horizontal miteinander verbunden, jeweils ("a" bis "e") und ("f" bis "j")

Als Spannungsversorgung kommt eine 4,5 Volt Flachbatterie zum Einsatz

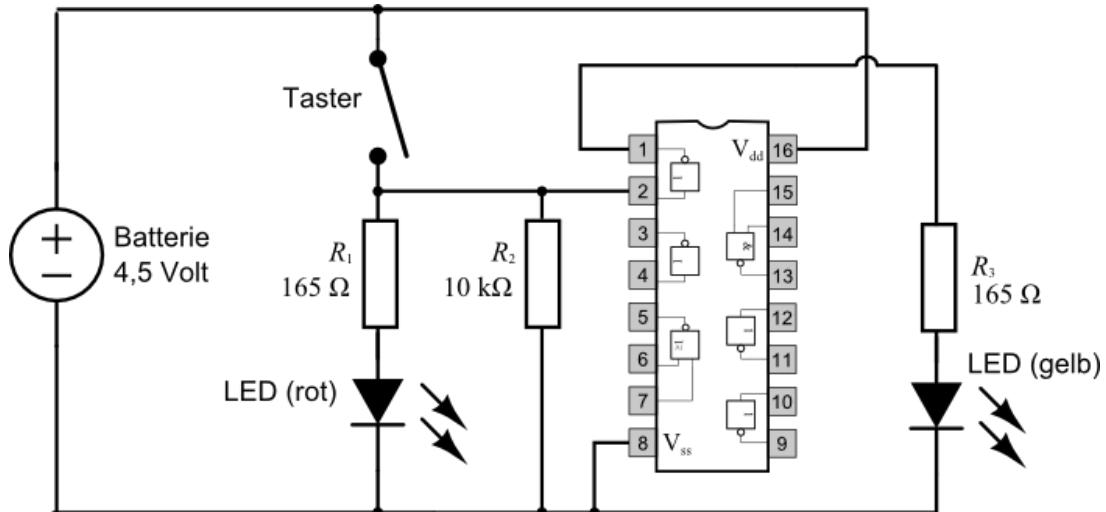


Praxisbeispiel: NOT-Gatter

Wenn der Taster offen ist, brennt die gelbe LED und die rote ist aus

Wenn der Taster geschlossen ist, brennt die rote LED und die gelbe ist aus

Beispielanwendung Alarmanlage: Taster ermittelt, ob Tür offen oder geschlossen ist; rote LED zeigt, Alarm ist eingeschaltet; gelbe LED alarmiert Wachdienst

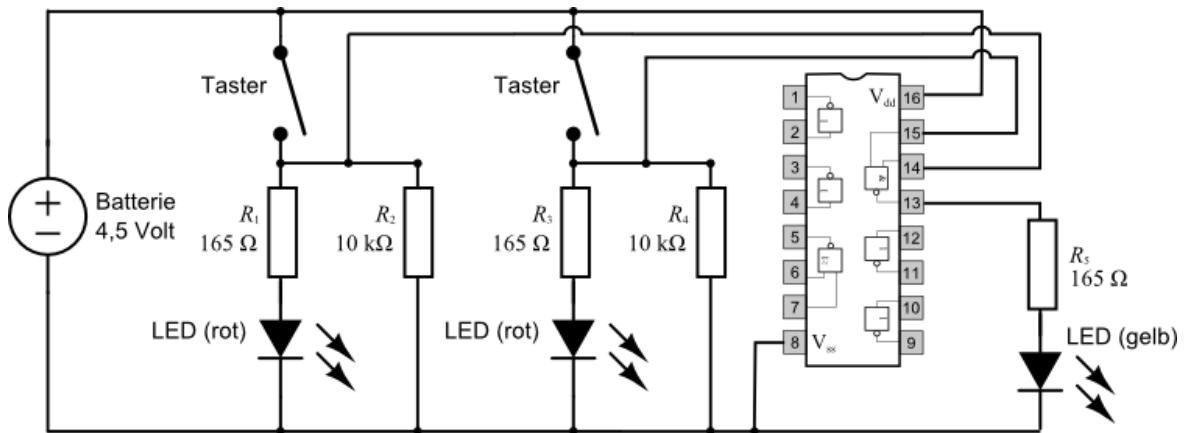


Praxisbeispiel: NAND-Gatter

Die roten LEDs zeigen den Status der beiden Taster an

Gelbe LED brennt nur dann nicht, wenn beide Taster geschlossen sind

Beispielanwendung: Alarmanlage für zwei Fenster



Praxisbeispiel: Aufbau einer Logikschaltung mit ICs

Dieses Video zeigt (im Schnelldurchlauf) den Aufbau der NOT-Gatter- und NAND-Gatter-Schaltung aus den vorangegangen Folien auf eine Steckplatine

0:00 / 0:45



Thorsten Thormählen 18 / 22

Praxisbeispiel: Aufbau einer Logikschaltung mit ICs

Bei den gezeigten Schaltungen wurden die Eingänge des Logik-ICs jeweils mit einem so genannten Pull-Down-Widerstand von $10\text{ k}\Omega$ versehen

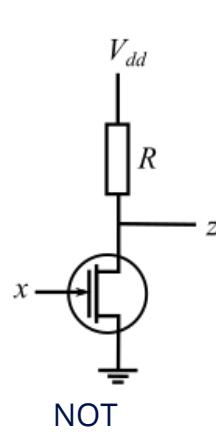
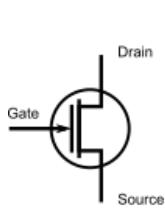
Wenn die Schalter geöffnet sind, würde ansonsten der Eingang auf undefiniertem Potenzial liegen und das Verhalten am Ausgang wäre zufällig

Bei geöffnetem Schalter zieht der Pull-Down-Widerstand den Eingang gegen Masse

Bei geschlossenem Schalter liegt die Versorgungsspannung am Eingang an und über den Pull-Down-Widerstand fließt ein kleiner Verluststrom $I_{\text{Verlust}} = \frac{4,5\text{V}}{10\text{ k}\Omega} = 0,45\text{ mA}$

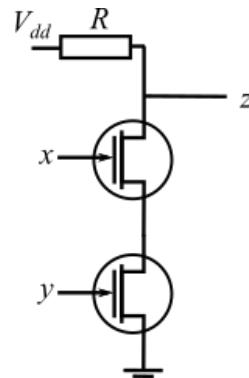
Realisierung mit n-Kanal-Feldeffekt-Transistoren

Inverter, NAND- und NOR-Gatter, wie sie im IC CD4572UB verwendet werden, können z.B. durch n-Kanal-Feldeffekt-Transistoren realisiert werden, die wir im Kapitel 1.2 "Historisches" kennengelernt haben



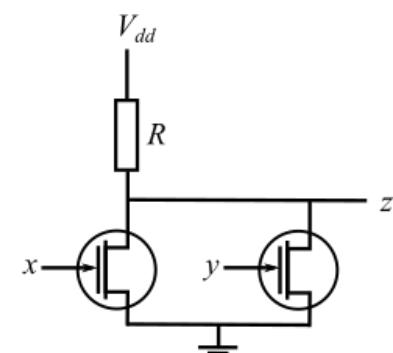
NOT

x	z
0	1
1	0



NAND

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



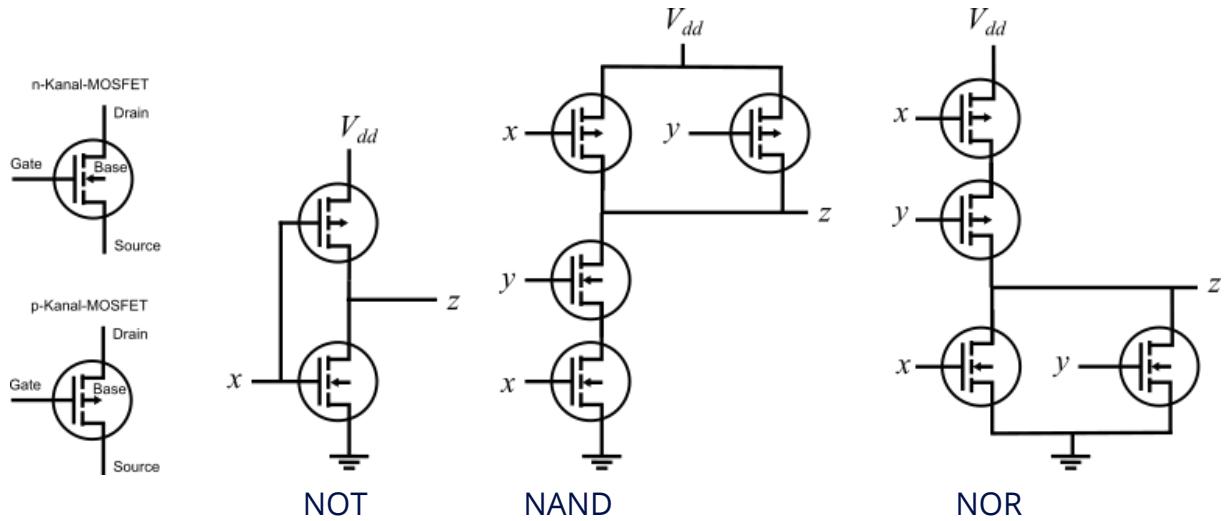
NOR

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

Realisierung mit CMOS-Technik

Durch den Widerstand entsteht jedoch eine relativ große Verlustleistung, daher wird bei der heutzutage vorherrschenden CMOS-Technik mit p- und n-Kanal-Feldeffekt-Transistoren gearbeitet (auch der CD4572UB ist ein CMOS IC)

Der n-Kanal-FET schaltet bei der logischen 1 durch, während der p-Kanal-FET bei der logischen 0 durchschaltet



Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) .

Thorsten Thormählen 22 / 22

Technische Informatik

Normalformen

Thorsten Thormählen

10. November 2022

Teil 4, Kapitel 1

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Typesetting math: 100%

Typesetting math: 100%

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Typesetting math: 100%

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Typesetting math: 100%

Inhalt

Minterm und Maxterm

Disjunktive Normalform

Konjunktive Normalform

Zusammenhang zwischen den Normalformen

Realisierung durch Logikgatter

Typesetting math: 100%

Normalformen

Die Wahrheitstabelle ist eine eindeutige Definition einer booleschen Funktion

Es gibt jedoch unendlich viele verschiedene Realisierungen mittels Logikgattern oder Beschreibungen in Form eines booleschen Ausdrucks

Normalformen (auch kanonische Formen):

Standarddarstellung für einen booleschen Ausdruck in einer eindeutigen algebraischen Form

Typesetting math: 100%

Minterm

Gegeben sei eine boolesche Funktion
 $y = f(x_n, \dots, x_1, x_0)$ und die Literale $\hat{x}_i \in \{\bar{x}_i, x_i\}$.

Minterm: $(\hat{x}_n \wedge \dots \wedge \hat{x}_2 \wedge \hat{x}_1 \wedge \hat{x}_0)$

Der Minterm evaluiert für genau eine bestimmte Konfiguration der Variablen x_i zu 1 und sonst zu 0

Genauer gesagt: Der Minterm evaluiert genau dann zu 1, wenn für alle negierten Variablen $x_i = 0$ und alle nicht negierten $x_i = 1$

Beispiel für einen Minterm: $y = \bar{x}_2 \wedge \bar{x}_1 \wedge x_0$

Ein Minterm ist ein Monom, in dem alle Variablen vorkommen müssen

Typesetting math: 100%

Maxterm

Gegeben sei eine boolesche Funktion
 $y = f(x_n, \dots, x_1, x_0)$ und die Literale $\hat{x}_i \in \{\bar{x}_i, x_i\}$.

Maxterm: $(\hat{x}_n \vee \dots \vee \hat{x}_2 \vee \hat{x}_1 \vee \hat{x}_0)$

Der Maxterm evaluiert für genau eine bestimmte Konfiguration der Variablen x_i zu 0 und sonst zu 1

Genauer gesagt: Der Maxterm evaluiert genau dann zu 0, wenn für alle negierten Variablen $x_i = 1$ und alle nicht negierten $x_i = 0$

Beispiel für einen Maxterm:

$$y = f(x_2, x_1, x_0) = \bar{x}_2 \vee \bar{x}_1 \vee x_0$$

Typesetting math: 100%

Disjunktive Normalform

Eine disjunktive Normalform (DNF) ist eine ODER-Verknüpfung von Mintermen

Alle Konfigurationen von Mintermen, in denen

$y = f(x_n, \dots, x_1, x_0) = 1$, müssen vorkommen

Beispiel:

x_2	x_1	x_0	y	DNF
0	0	0	0	
0	0	1	1	$(\neg x_2 \wedge \neg x_1 \wedge x_0)$
0	1	0	1	$\vee(\neg x_2 \wedge x_1 \wedge \neg x_0)$
0	1	1	0	
1	0	0	1	$\vee(x_2 \wedge \neg x_1 \wedge \neg x_0)$
1	0	1	0	
1	1	0	1	$\vee(x_2 \wedge x_1 \wedge \neg x_0)$
1	1	1	1	$\vee(x_2 \wedge x_1 \wedge x_0)$

Typesetting math: 100%

Disjunktive Normalform (DNF)

Anzahl der Variablen:

Durch Klicken auf die grauen Elemente kann die boolesche Funktion verändert werden

x_2	x_1	x_0	y	DNF
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	0	0

Typesetting math: 100%

Konjunktive Normalform

Eine konjunktive Normalform (KNF) ist eine UND-Verknüpfung von Maxtermen

Alle Konfigurationen von Maxtermen, in denen

$y = f(x_n, \dots, x_1, x_0) = 0$, müssen vorkommen

Beispiel:

x_2	x_1	x_0	y	KNF
0	0	0	0	$(x_2 \vee x_1 \vee x_0)$
0	0	1	1	
0	1	0	1	
0	1	1	0	$\wedge(x_2 \vee \neg x_1 \vee \neg x_0)$
1	0	0	1	
1	0	1	0	$\wedge(\neg x_2 \vee x_1 \vee \neg x_0)$
1	1	0	1	
1	1	1	1	

Typesetting math: 100%

Konjunktive Normalform (KNF)

Anzahl der Variablen: 3

Durch Klicken auf die grauen Elemente kann die boolesche Funktion verändert werden

x_2	x_1	x_0	y	KNF
0	0	0	0	$(x_2 \vee x_1 \vee x_0)$
0	0	1	0	$\wedge(x_2 \vee x_1 \vee \neg x_0)$
0	1	0	0	$\wedge(x_2 \vee \neg x_1 \vee x_0)$
0	1	1	0	$\wedge(x_2 \vee \neg x_1 \vee \neg x_0)$
1	0	0	0	$\wedge(\neg x_2 \vee x_1 \vee x_0)$
1	0	1	0	$\wedge(\neg x_2 \vee x_1 \vee \neg x_0)$
1	1	0	0	$\wedge(\neg x_2 \vee \neg x_1 \vee x_0)$
1	1	1	0	$\wedge(\neg x_2 \vee \neg x_1 \vee \neg x_0)$

Typesetting math: 100%

Normalformen der negierten Funktion

Teilweise kann es einfacher sein, die Normalform der negierten Funktion zu betrachten

Beispiel:

x_2	x_1	x_0	y	DNF y	$\neg y$	DNF $\neg y$
0	0	0	0		1	$(\neg x_2 \wedge \neg x_1 \wedge \neg x_0)$
0	0	1	1	$(\neg x_2 \wedge \neg x_1 \wedge x_0)$	0	
0	1	0	1	$\vee(\neg x_2 \wedge x_1 \wedge \neg x_0)$	0	
0	1	1	0		1	$\vee(\neg x_2 \wedge x_1 \wedge x_0)$
1	0	0	1	$\vee(x_2 \wedge \neg x_1 \wedge \neg x_0)$	0	
1	0	1	0		1	$\vee(x_2 \wedge \neg x_1 \wedge x_0)$
1	1	0	1	$\vee(x_2 \wedge x_1 \wedge \neg x_0)$	0	
1	1	1	1	$\vee(x_2 \wedge x_1 \wedge x_0)$	0	

Typesetting math: 100%

Umformen der negierten Funktion mit De Morgan

Disjunktive Normalform (Beispiel von Folie vorher)

$$\bar{y} = \bar{x}_2 \bar{x}_1 \bar{x}_0 \vee \bar{x}_2 x_1 x_0 \vee x_2 \bar{x}_1 x_0$$

Mit De Morgans Gesetz (Theorem 14)

$$\neg \bar{y} = \neg(\bar{x}_2 \bar{x}_1 \bar{x}_0 \vee \bar{x}_2 x_1 x_0 \vee x_2 \bar{x}_1 x_0)$$

$$y = (x_2 \vee x_1 \vee x_0) \wedge (\bar{x}_2 \vee \bar{x}_1 \vee \bar{x}_0) \wedge (\bar{x}_2 \vee x_1 \vee \bar{x}_0)$$

Konjunktive Normalform (neues Beispiel)

$$\bar{y} = (x_2 \vee \bar{x}_1 \vee x_0 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_1 \vee \bar{x}_0 \vee \bar{x}_3)$$

Mit De Morgans Gesetz (Theorem 14)

$$\neg \bar{y} = \neg((x_2 \vee \bar{x}_1 \vee x_0 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_1 \vee \bar{x}_0 \vee \bar{x}_3))$$

$$y = \bar{x}_2 x_1 \bar{x}_0 \bar{x}_3 \vee x_2 x_1 x_0 x_3$$

Typesetting math: 100%

Zusammenhang zwischen den Normalformen

Umrechnung DNF in KNF

Verwenden derjenigen Maxterme M_i , die nicht in der Erweiterung der Minterme m_i enthalten sind

Beispiel:

$$f(x_2, x_1, x_0) = \bigvee_{i \in \{1, 3, 5, 6, 7\}} m_i = \bigwedge_{i \in \{0, 2, 4\}} M_i$$

Umrechnung KNF in DNF

Umgekehrtes Vorgehen: Verwenden derjenigen Minterme m_i , die nicht in der Erweiterung der Maxterme M_i enthalten sind

Typesetting math: 100%

Zusammenhang zwischen den Normalformen

Anzahl der Variablen: 3

Durch Klicken auf die grauen Elemente kann die boolesche Funktion verändert werden

x_2	x_1	x_0	y	DNF	KNF
0	0	0	0		$(x_2 \vee x_1 \vee x_0)$
0	0	1	0		$\wedge(x_2 \vee x_1 \vee \neg x_0)$
0	1	0	0		$\wedge(x_2 \vee \neg x_1 \vee x_0)$
0	1	1	0		$\wedge(x_2 \vee \neg x_1 \vee \neg x_0)$
1	0	0	0		$\wedge(\neg x_2 \vee x_1 \vee x_0)$
1	0	1	0		$\wedge(\neg x_2 \vee x_1 \vee \neg x_0)$
1	1	0	0	0	$\wedge(\neg x_2 \vee \neg x_1 \vee x_0)$
1	1	1	0		$\wedge(\neg x_2 \vee \neg x_1 \vee \neg x_0)$

Thorsten Thormählen 15 / 20

Typesetting math: 100%

Typesetting math: 100%

Zusammenhang zwischen den Normalformen

Umwandlung der DNF von f in die DNF von $\neg f$

Verwenden für $\neg f$ diejenigen Minterme m_i , die nicht in f enthalten sind

Beispiel:

$$f(x_2, x_1, x_0) = \bigvee_{i \in \{1,3,5,6,7\}} m_i$$
$$\Leftrightarrow \neg f(x_2, x_1, x_0) = \bigvee_{i \in \{0,2,4\}} m_i$$

Umwandlung der KNF von f in die KNF von $\neg f$

Verwenden für $\neg f$ diejenigen Maxterme M_i , die nicht in f enthalten sind

Beispiel:

$$f(x_2, x_1, x_0) = \bigwedge_{i \in \{0,2,4\}} M_i$$
$$\Leftrightarrow \neg f(x_2, x_1, x_0) = \bigwedge_{i \in \{1,3,5,6,7\}} M_i$$

Typesetting math: 100%

Realisierung durch Logikgatter

Jede booleschen Funktion $f = (x_{n-1}, \dots, x_1, x_0)$ der Stelligkeit n lässt sich durch einen Standard-Schaltkreis realisieren, der der DNF bzw. KNF entspricht

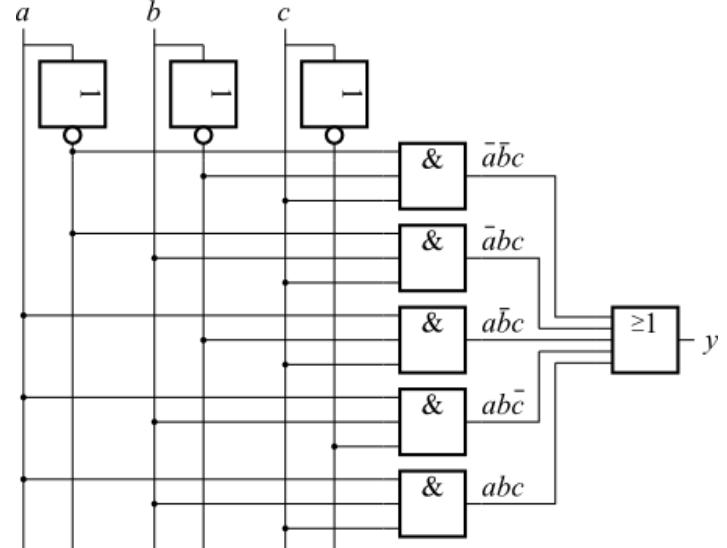
Im Fall der DNF werden die Minterme durch UND-Gatter mit n Eingängen realisiert

Die Erweiterung der Minterme zur DNF erfolgt durch ein ODER-Gatter

Beispiel:

$$y = f(a, b, c) = ab \vee c$$

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Thorsten Thormählen 17 / 20

Typesetting math: 100%

Typesetting math: 100%

Realisierung durch Logikgatter

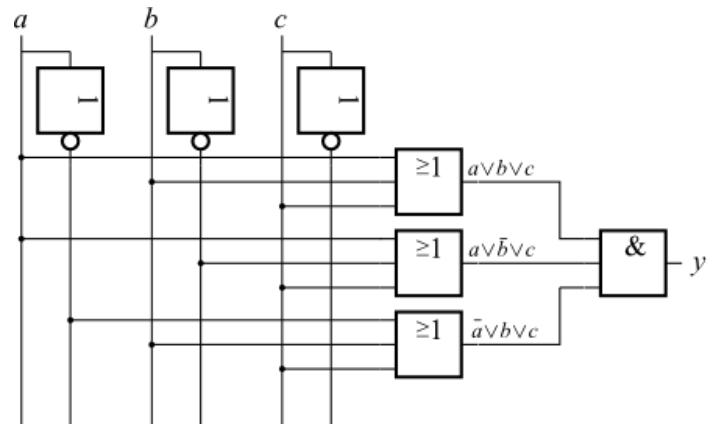
Im Fall der KNF werden die Maxterme durch ODER-Gatter mit n Eingängen realisiert

Die Erweiterung der Maxterme zur KNF erfolgt durch ein UND-Gatter

Beispiel:

$$y = f(a, b, c) = ab \vee c$$

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Thorsten Thormählen 18 / 20

Typesetting math: 100%

Typesetting math: 100%

Quiz

Frage: Was ist die DNF für folgende Boolesche Funktion?

<i>a</i>	<i>b</i>	<i>c</i>	<i>y</i>
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Antwort 1: $abc \vee a\bar{b}c \vee \bar{a}\bar{b}\bar{c}$

Antwort 2: $\bar{a}\bar{c} \vee abc$

Antwort 3: $\bar{a}\bar{b}\bar{c} \vee \bar{a}b\bar{c} \vee abc$

Am Online-Quiz teilnehmen durch Besuch der Webseite:
www.onlineclicker.org

Thorsten Thormählen 19 / 20

Typesetting math: 100%

Typesetting math: 100%

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) .

Thorsten Thormählen 20 / 20

Typeetting math: 100%

Typesetting math: 100%

Technische Informatik

Umwandlung in NAND und NOR

Thorsten Thormählen
15. November 2022
Teil 5, Kapitel 1

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

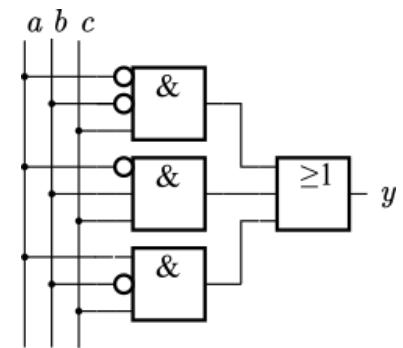
Umwandlung zweistufiger Logik in
NAND-Gatter-Schaltungen
NOR-Gatter-Schaltungen
Umwandlung mehrstufiger Logik

Realisierung zweistufiger Logik

Disjunktive Normalform (DNF)

AND-Gatter für die Minterme

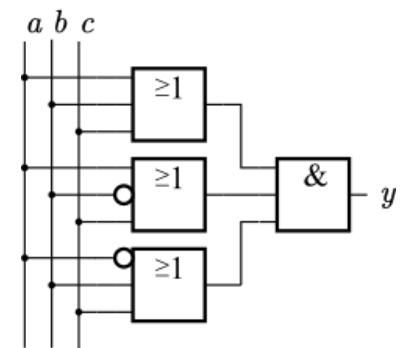
OR-Gatter für die Erweiterung



Konjunktive Normalform (KNF)

OR-Gatter für die Maxterme

AND-Gatter für die Erweiterung



Zweistufige Logik mit NAND-Gattern

1. Schritt

Ersetzen der AND-Gatter der Minterme mit NAND-Gattern

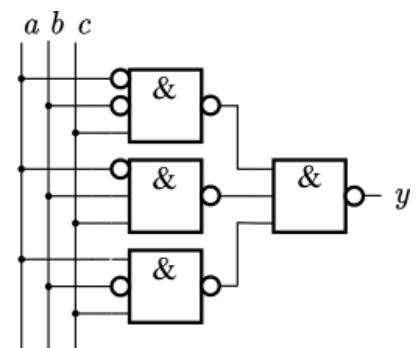
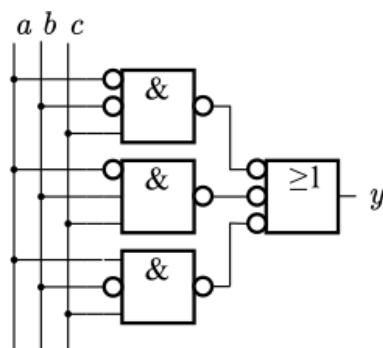
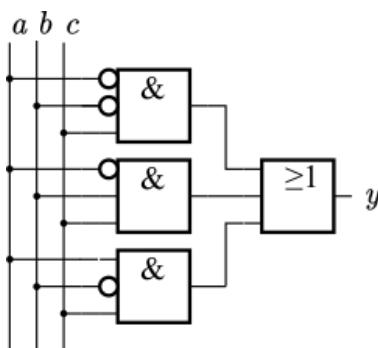
Kompensation durch Invertieren der Eingänge des OR-Gatters

2. Schritt

Laut De Morgan kann ein OR-Gatter mit invertierten Eingängen durch ein NAND-Gatter ersetzt werden

$$\neg x_0 \vee \neg x_1 \vee \neg x_2 = \neg(x_0 \wedge x_1 \wedge x_2)$$

Fertige Schaltung besteht nur noch aus NAND-Gattern



Zweistufige Logik mit NOR-Gattern

1. Schritt

Ersetzen der OR-Gatter der Maxterme mit NOR-Gattern

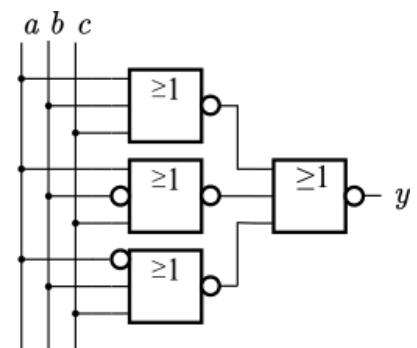
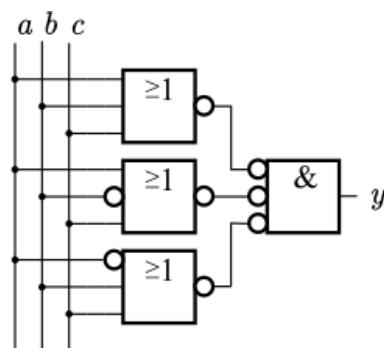
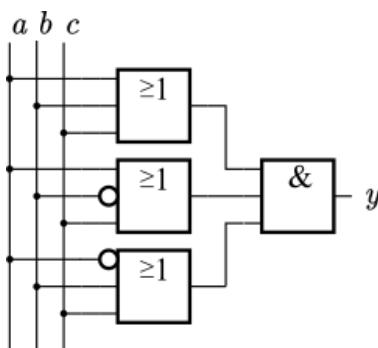
Kompensation durch Invertieren der Eingänge des AND-Gatters

2. Schritt

Laut de Morgan kann ein AND-Gatter mit invertierten Eingängen durch ein NOR-Gatter ersetzt werden

$$\neg x_0 \wedge \neg x_1 \wedge \neg x_2 = \neg(x_0 \vee x_1 \vee x_2)$$

Fertige Schaltung besteht nur noch aus NOR-Gattern



Äquivalente Gatter

Mit De Morgans Gesetzen lassen sich vier Gleichungen aufstellen:

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

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

Damit erhalten wir vier austauschbare Gattertypen

OR ist ein NAND mit invertierten Eingängen:

$$\begin{array}{c} a \\ b \end{array} \begin{array}{|c|} \hline \geq 1 \\ \hline \end{array} - y = \begin{array}{c} a \\ b \end{array} \begin{array}{|c|} \hline \& \\ \hline \end{array} - y$$

AND ist ein NOR mit invertierten Eingängen:

$$\begin{array}{c} a \\ b \end{array} \begin{array}{|c|} \hline \& \\ \hline \end{array} - y = \begin{array}{c} a \\ b \end{array} \begin{array}{|c|} \hline \geq 1 \\ \hline \end{array} - y$$

NAND ist ein OR mit invertierten Eingängen:

$$\begin{array}{c} a \\ b \end{array} \begin{array}{|c|} \hline \& \\ \hline \end{array} - y = \begin{array}{c} a \\ b \end{array} \begin{array}{|c|} \hline \geq 1 \\ \hline \end{array} - y$$

NOR ist ein AND mit invertierten Eingängen:

$$\begin{array}{c} a \\ b \end{array} \begin{array}{|c|} \hline \geq 1 \\ \hline \end{array} - y = \begin{array}{c} a \\ b \end{array} \begin{array}{|c|} \hline \& \\ \hline \end{array} - y$$

Mehrstufige Logik

Wieso überhaupt mehrstufige Logik? Zwei Stufen reichen doch eigentlich.

Motivierendes Beispiel:

$$y = adf \vee aef \vee bdf \vee bef \vee cdf \vee cef \vee g$$

Dies ist eine reduzierte (vereinfachte) DNF

Für die Realisierung werden 6 AND-Gatter mit 3 Eingängen und ein OR-Gatter mit 7 Eingängen benötigt (vielleicht nicht realisierbar)

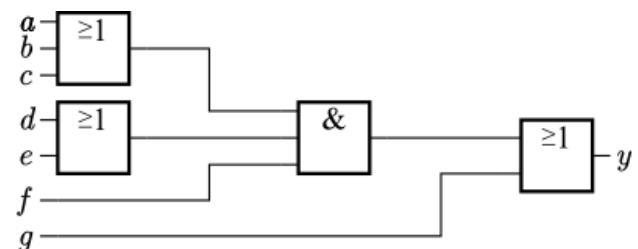
25 Verbindungen (19 Literale plus 6 interne Verbindungen)

Faktorisierte Form, die nicht als zweistufige DNF realisierbar ist

$$y = (a \vee b \vee c)(d \vee e)f \vee g$$

Für die Realisierung werden 1 OR-Gatter mit 3 Eingängen und 2 OR-Gatter mit 2 Eingängen und 1 AND-Gatter mit 3 Eingängen benötigt

10 Verbindungen (7 Literale plus 3 interne Verbindungen)



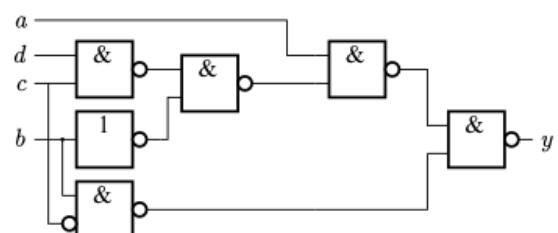
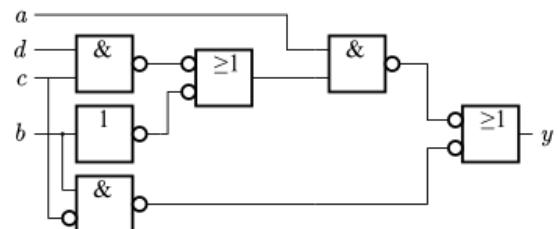
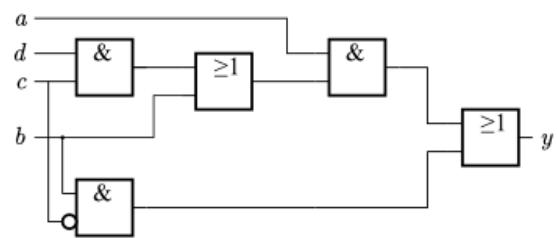
Umwandlung mehrstufiger Logik

Ursprüngliches AND / OR Schaltnetz

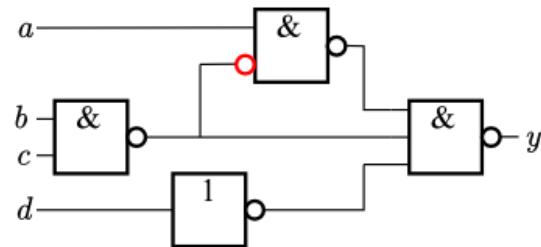
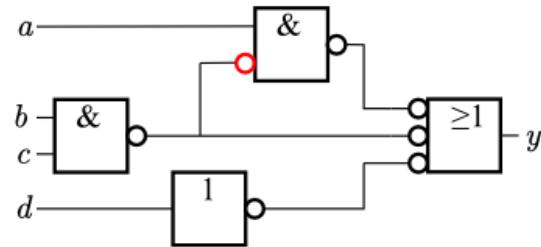
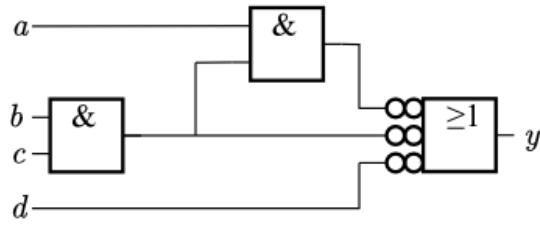
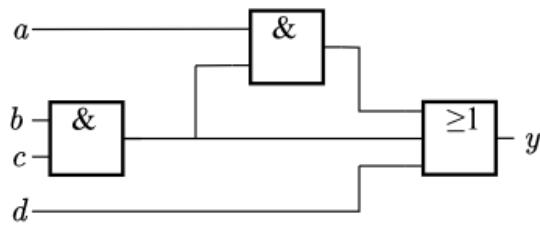
$$y = a(b \vee cd) \vee b\bar{c}$$

Einfügen geeigneter Inverter

Umwandeln in NAND-Gatter



Umwandlung mehrstufiger Logik



Thorsten Thormählen 11 / 12

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) .]

Thorsten Thormählen 12 / 12

Technische Informatik

Minimierung mit KV-Diagrammen

Thorsten Thormählen
17. November 2022
Teil 5, Kapitel 2

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Typesetting math: 59%

Typesetting math: 59%

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Typesetting math: 59%

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Typesetting math: 59%

Das Vereinigungstheorem

Das Vereinigungstheorem ist ein wichtiges Theorem zur systematischen Minimierung von Funktionen:

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

Grundsatz zur Vereinfachung bei zweistufiger Logik (ODER-Verknüpfung von Monomen):

Sind in der "1"-Menge zwei Monome, die sich nur in einer Variablen unterscheiden, dann können diese zusammengefasst werden. Zur Abdeckung dieser beiden Elemente der "1"-Menge kann ein einzelnes Monom verwendet werden, in welchem diese Variable nicht mehr vorkommt.

Beispiel 1:

b	a	y	DNF
0	0	1	$(\bar{a} \wedge \bar{b})$
0	1	0	
1	0	1	$\vee(\bar{a} \wedge b)$
1	1	0	

Die Variable b kommt nicht negiert und negiert vor, daher:

$$y = \bar{a}\bar{b} \vee \bar{a}b = \bar{a}$$

Typesetting math: 59%

Das Vereinigungstheorem

Beispiel 2:

x_2	x_1	x_0	y	DNF
0	0	0	0	
0	0	1	0	
0	1	0	1	($\neg x_2 \wedge x_1 \wedge \neg x_0$)
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	1	$\vee(x_2 \wedge x_1 \wedge \neg x_0)$
1	1	1	0	

Die Variable x_2 kommt nicht negiert und negiert vor, daher:

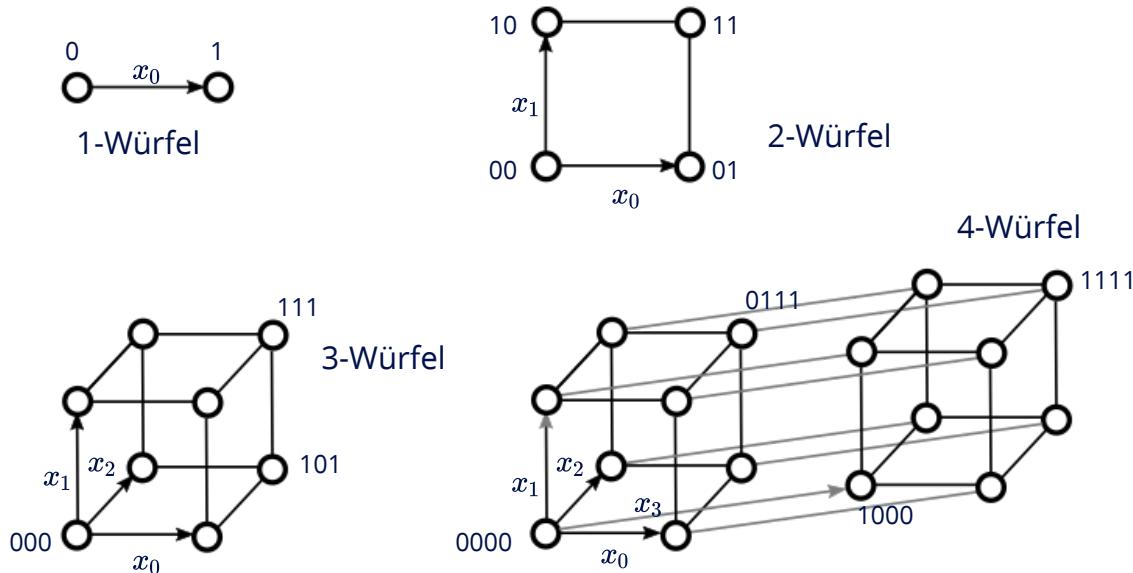
$$y = \bar{x}_2 x_1 \bar{x}_0 \vee x_2 x_1 \bar{x}_0 = x_1 \bar{x}_0$$

Typesetting math: 59%

Boolescher Würfel

Mit Hilfe des Booleschen Würfels kann einfach festgestellt werden, ob das Vereinigungstheorem angewendet werden kann

n -Eingangsvariablen $\equiv n$ -dimensionaler Würfel



Thorsten Thormählen 6 / 34

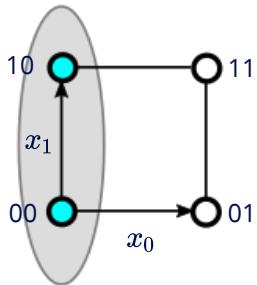
Typesetting math: 59%

Typesetting math: 59%

Boolescher Würfel

Das Vereinigungstheorem fasst zwei Unterräume eines Würfels zu einem größeren zusammen

Beispiel:



Die Variable x_1 kommt nicht negiert und negiert vor, daher:

$$y = \overline{x_1} \overline{x_0} \vee x_1 \overline{x_0} = \overline{x_0}$$

Zwei benachbarte Unterräume der Größe 0 (Knoten) werden zu einem Unterraum der Größe 1 (Kante) zusammengefasst.

x_1	x_0	y	DNF
0	0	1	$(\overline{x_1} \wedge \overline{x_0})$
0	1	0	
1	0	1	$\vee(x_1 \wedge \overline{x_0})$
1	1	0	

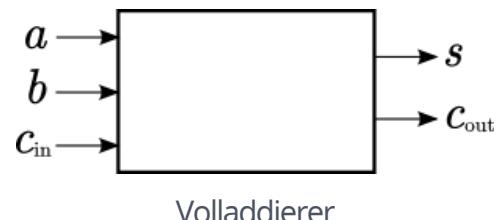
Typesetting math: 59%

Boolescher Würfel

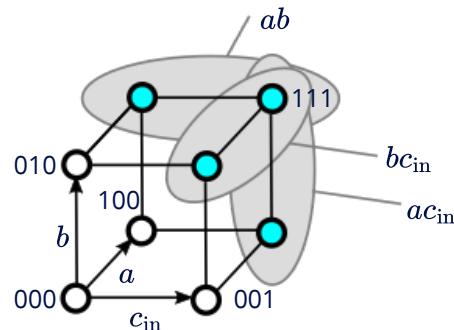
Beispiel: 1-bit binärer Volladdierer
(aus Kapitel 3.2)

Eingänge: Summanden a und b , Carry-in c_{in}

Ausgänge: Summe s , Carry-out c_{out}



Volladdierer



a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Die DNF für das Carry-out aus Tabelle: $c_{\text{out}} = \bar{a}bc_{\text{in}} \vee \bar{a}\bar{b}c_{\text{in}} \vee a\bar{b}c_{\text{in}} \vee abc_{\text{in}}$

Zweistufige Logik aus booleschem Würfel: $c_{\text{out}} = bc_{\text{in}} \vee ac_{\text{in}} \vee ab$

Typesetting math: 59%

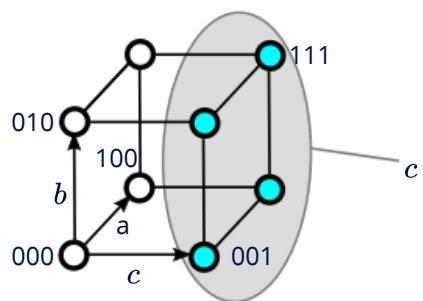
Boolescher Würfel

Das Verfahren kann rekursiv fortgeführt werden:

Zwei benachbarte Unterräume der Größe 0 (Knoten) werden zu einem Unterraum der Größe 1 (Kante) zusammengefasst. Dies ergibt für das Beispiel unten z. B. die beiden Kanten ac und $\bar{a}c$, d.h. $y = ac \vee \bar{a}c$

Zwei benachbarte Unterräume der Größe 1 (Kanten) werden zu einem Unterraum der Größe 2 (Seite) zusammengefasst.

Dies ergibt für das Beispiel: $y = c$



D. h. c ist wahr - bleibt unverändert. Die Variablen a und b variieren.

Typesetting math: 59%

Boolescher Würfel

In einem Würfel mit drei Variablen gibt es folgende Unterräume:

0-Würfel (Knoten), dies entspricht einem Term mit drei Literalen

1-Würfel (Kante), dies entspricht einem Term mit zwei Literalen

2-Würfel (Seite), dies entspricht einem Term mit einem Literal

3-Würfel (Würfel), dies entspricht der Konstanten 1

Im Allgemeinen gilt:

Ein m -Subwürfel in einem n -Würfel ($m < n$) repräsentiert einen Term mit $n - m$ Literalen

Typesetting math: 59%

Karnaugh-Veitch-Diagramme

Karnaugh-Veitch-Diagramme (KV-Diagramme) dienen zur übersichtlichen Darstellung und systematischen Vereinfachung boolescher Funktionen

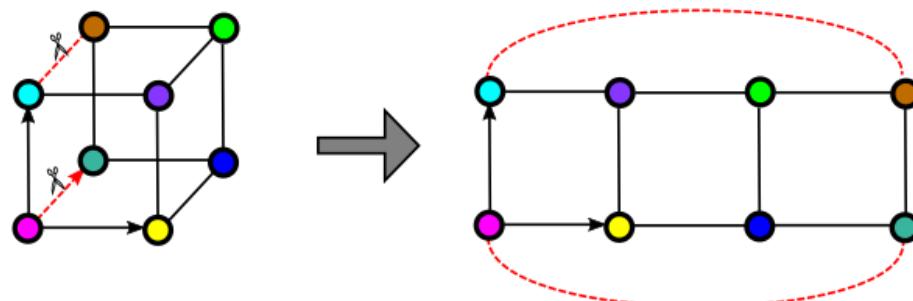
Ziel: Umwandlung einer disjunktiven Normalform in einen minimalen disjunktiven logischen Ausdruck (minimale ODER-Verknüpfung von Monomen)

Sie wurden 1952 von Edward W. Veitch entworfen und 1953 von Maurice Karnaugh zu ihrer heutigen Form weiterentwickelt

Sie entsprechen einer flachen Abbildung des booleschen Würfels

Verbindungen an den Schnittkanten (rote gestrichelte Linien) müssen sich gedacht werden

Schwierig zu zeichnen, wenn mehr als 4 Variablen vorkommen

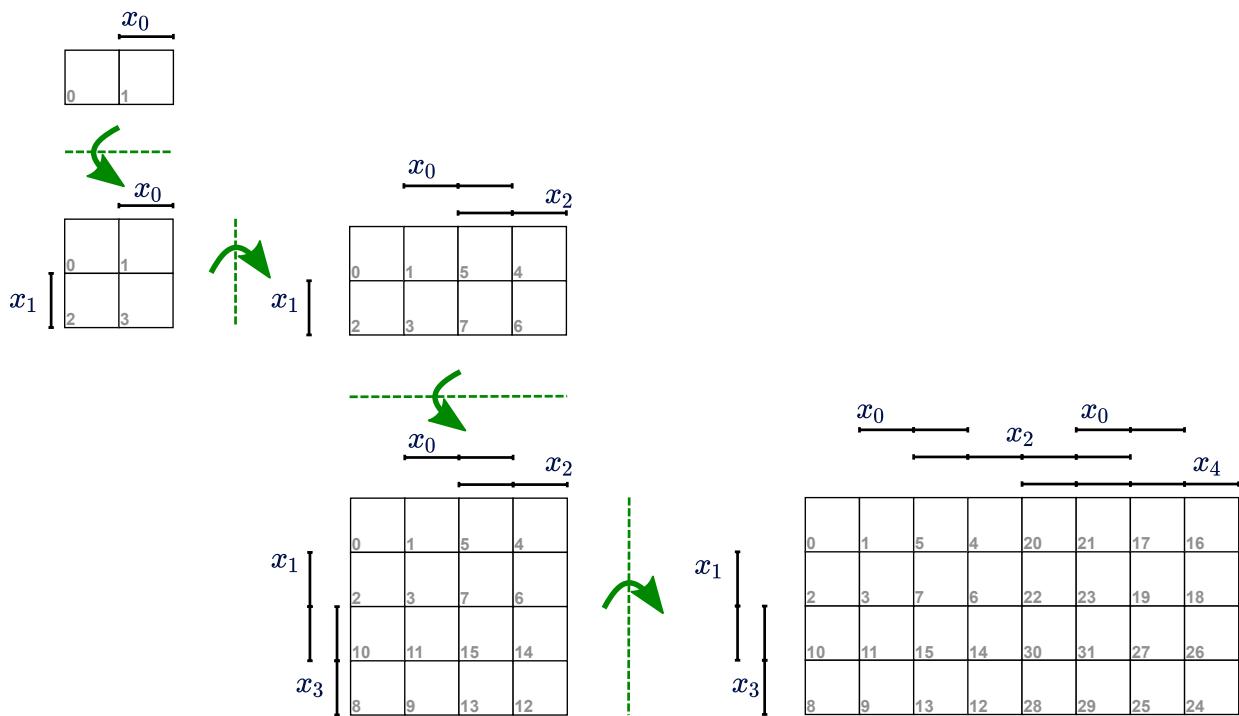


Thorsten Thormählen 11 / 34

Typesetting math: 59%

Typesetting math: 59%

Konstruktion von KV-Diagrammen



Thorsten Thormählen 12 / 34

Typesetting math: 59%

Typesetting math: 59%

Konstruktion von KV-Diagrammen

Das KV-Diagramm einer boolesche Funktion $y = f(x_0, x_1, \dots, x_{n-1})$ der Stelligkeit n hat 2^n Zellen

Bereiche, in denen eine Variable x_i den Wert 1 hat, sind durch einen Strich gekennzeichnet

Beim Hinzufügen einer neuen Variablen x_i wird das bisherige KV-Diagramm durch abwechselndes vertikales und horizontales Spiegeln erzeugt

Dabei verdoppeln sich jeweils die Anzahl der Zellen

Benachbarte Zellen unterscheiden sich in genau einer Variablen

Zellen an der linke und rechten (bzw. oberen und untern) Kante des Diagramm sind ebenfalls benachbart (warp-around)

Bei 5 Variablen ist der Bereich für x_0 räumlich gespalten

Allgemein gilt: Bei $n > 4$ Variablen sind die Bereiche für $n - 4$ Variablen räumlich gespalten

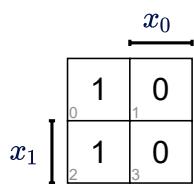
D. h. wirklich intuitiv ist die Nutzung nur bis zu 4 Variablen

Typesetting math: 59%

Übertragen von Wahrheitstafeln in das KV-Diagramm

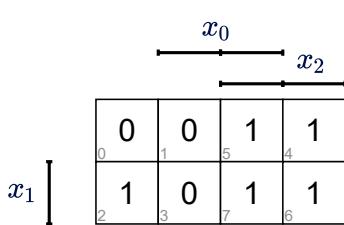
Für eine Funktion $y = f(x_1, x_0)$ wird folgendes Diagramm konstruiert:

x_1	x_0	y	KV-Zelle
0	0	1	0
0	1	0	1
1	0	1	2
1	1	0	3



Für eine Funktion $y = f(x_2, x_1, x_0)$ wird folgendes Diagramm konstruiert:

x_2	x_1	x_0	y	KV-Zelle
0	0	0	0	0
0	0	1	0	1
0	1	0	1	2
0	1	1	0	3
1	0	0	1	4
1	0	1	1	5
1	1	0	1	6
1	1	1	1	7

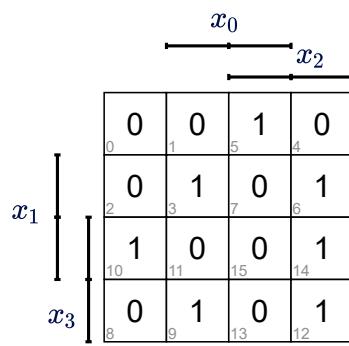


Typesetting math: 59%

Übertragen von Wahrheitstafeln in das KV-Diagramm

Für eine Funktion $y = f(x_3, x_2, x_1, x_0)$ wird folgendes Diagramm konstruiert:

x_3	x_2	x_1	x_0	y	KV-Zelle
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	2
0	0	1	1	1	3
0	1	0	0	0	4
0	1	0	1	1	5
0	1	1	0	1	6
0	1	1	1	0	7
1	0	0	0	0	8
1	0	0	1	1	9
1	0	1	0	1	10
1	0	1	1	0	11
1	1	0	0	1	12
1	1	0	1	0	13
1	1	1	0	1	14
1	1	1	1	0	15



Typesetting math: 59%

Karnaugh-Veitch-Diagramme

Zur Konstruktion der bisherigen Diagramme wurde von einer Wahrheitstafel ausgegangen

Nun soll aus einem Diagramm eine minimale Oder-Verknüpfung von Monomen abgelesen werden

Dazu werden die verwendeten Monome so gewählt, dass möglichst viele "Einsen" gemeinsam abgelesen werden

Ein Monom entspricht dabei einem rechteckigen Block im KV-Diagramm

Somit werden möglichst große Blöcke gesucht, welche die 1-Menge im KV-Diagramm überdecken

Die Blöcke im KV-Diagramm entsprechen Unterräumen im Booleschen Würfel

		x_0	x_2	
		0	1	1
		0	5	4
x_1	0	0	1	1
	1	0	0	0

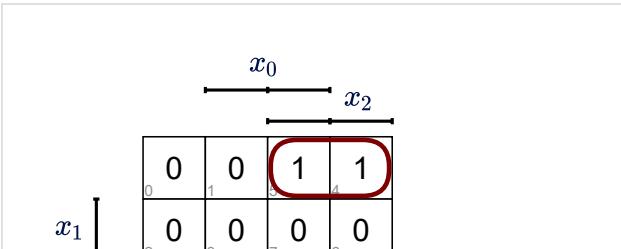
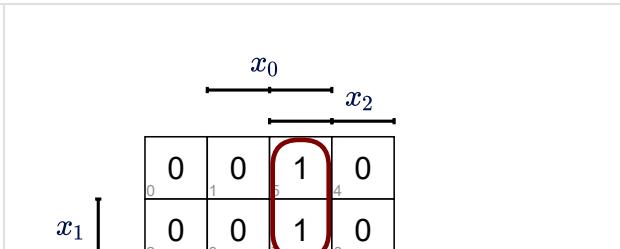
$$y = \bar{x}_2 \bar{x}_1 x_0 \vee x_2 \bar{x}_1 x_0 = \bar{x}_1 x_0$$

Thorsten Thormählen 16 / 34

Typesetting math: 59%

Typesetting math: 59%

Überdecken von 1-Mengen in KV-Diagrammen

 <p>x_0 x_2</p> <table border="1" style="border-collapse: collapse; text-align: center;"> <tr> <td>0</td><td>0</td><td>0</td><td>1</td> </tr> <tr> <td>0</td><td>1</td><td>5</td><td>4</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>1</td> </tr> <tr> <td>2</td><td>3</td><td>7</td><td>6</td> </tr> </table> <p>x_1</p> $y = x_2 \bar{x}_1 \bar{x}_0 \vee x_2 x_1 \bar{x}_0 = x_2 \bar{x}_0$	0	0	0	1	0	1	5	4	0	0	0	1	2	3	7	6	 <p>x_0 x_2</p> <table border="1" style="border-collapse: collapse; text-align: center;"> <tr> <td>0</td><td>0</td><td>1</td><td>1</td> </tr> <tr> <td>0</td><td>1</td><td>5</td><td>4</td> </tr> <tr> <td>0</td><td>1</td><td>1</td><td>1</td> </tr> <tr> <td>2</td><td>3</td><td>6</td><td>7</td> </tr> </table> <p>x_1</p> $y = x_2$	0	0	1	1	0	1	5	4	0	1	1	1	2	3	6	7
0	0	0	1																														
0	1	5	4																														
0	0	0	1																														
2	3	7	6																														
0	0	1	1																														
0	1	5	4																														
0	1	1	1																														
2	3	6	7																														

Thorsten Thormählen 17 / 34

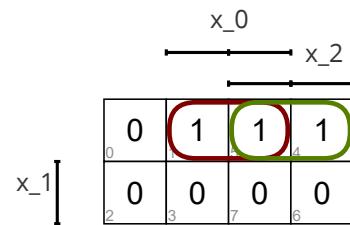
Typesetting math: 59%

Typesetting math: 59%

Überdecken von 1-Mengen in KV-Diagrammen

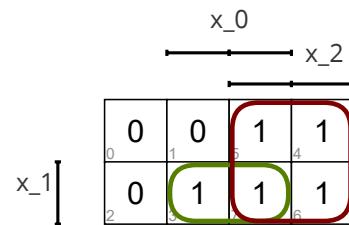
Die Blöcke dürfen sich überlappen

Blöcke haben immer Zweierpotenzen als Höhe und Breite. In diesem Beispiel ist es daher nicht möglich, ein 3×1 Block zu verwenden. Stattdessen ergeben sich zwei 2×1 Blöcke.



$$y = \overline{x_1} x_0 \lor x_2 \overline{x_1}$$

Es gibt häufig mehrere Möglichkeiten die 1-Menge zu überdecken. Es wird immer diejenige ausgewählt, die auf möglichst wenige Oder-Verknüpfungen führt (d.h. möglichst große Blöcke und davon wenige)



$$y = x_1 x_0 \lor x_2$$

Thorsten Thormählen 18 / 34

Typesetting math: 59%

Typesetting math: 59%

Überdecken von 1-Mengen in KV-Diagrammen

Es ist zu erkennen, dass bei größeren Blöcken das Vereinigungstheorem wiederholt angewendet wird, indem immer größere Blöcke aus benachbarten Blöcken gebildet werden

In dem Beispiel rechts werden aus zwei 2×1 Blöcken ein 2×2 Block

```
\begin{eqnarray} y &=& \overline{x}_3 x_2 \\ \overline{x}_1 x_0 \lor \overline{x}_3 x_2 x_1 x_0 \\ \lor \overline{x}_3 x_2 \overline{x}_1 \\ \overline{x}_0 \lor \overline{x}_3 x_2 x_1 \\ \overline{x}_0 \&=& \overline{x}_3 x_2 x_0 \lor \\ \overline{x}_3 x_2 \overline{x}_0 \&=& \\ \overline{x}_3 x_2 \end{eqnarray}
```

und in diesem Beispiel aus zwei 2×2 Blöcken ein 4×2 Block:

```
\begin{eqnarray} y &=& \overline{x}_3 x_2 \\ \overline{x}_1 x_0 \lor \overline{x}_3 x_2 x_1 x_0 \\ \lor \overline{x}_3 x_2 \overline{x}_1 \\ \overline{x}_0 \lor \overline{x}_3 x_2 x_1 \\ \overline{x}_0 \&\& \overline{x}_3 x_2 \overline{x}_1 \\ x_0 \lor x_3 x_2 x_1 x_0 \lor x_3 x_2 \\ \overline{x}_1 \overline{x}_0 \&=& x_3 x_2 x_1 \\ \overline{x}_0 \&=& \overline{x}_3 x_2 x_0 \lor \end{eqnarray}
```

Ein KV-Diagramm mit 16 Feldern (4 Zeilen und 4 Spalten). Die Spalten sind beschriftet als x_0 , x_1 , x_2 und x_3 . Die Zeilen sind mit den Werten 0 bis 15 beschriftet. Ein 2x2-Blöckchen im Bereich $x_1=1$ und $x_2=1$ ist rot eingekreist. Die vertikale Achse auf der linken Seite ist mit x_1 und x_3 beschriftet, die horizontale Achse oben mit x_0 und x_2 .

0	0	1	1
0	0	1	1
0	0	0	0
0	0	0	0

Ein KV-Diagramm mit 16 Feldern (4 Zeilen und 4 Spalten). Die Spalten sind beschriftet als x_0 , x_1 , x_2 und x_3 . Die Zeilen sind mit den Werten 0 bis 15 beschriftet. Ein 4x2-Blöckchen im Bereich $x_1=1$ und $x_2=1$ ist rot eingekreist. Die vertikale Achse auf der linken Seite ist mit x_1 und x_3 beschriftet, die horizontale Achse oben mit x_0 und x_2 .

0	0	1	1
0	0	1	1
0	0	1	1
0	0	1	1

```
\overline{x}_3 x_2 \overline{x}_0 \lor x_3 x_2 x_0  
\lor x_3 x_2 \overline{x}_0 \\ &= \overline{x}_3  
x_2 \lor x_3 x_2 \\ &= x_2 \end{eqnarray}
```

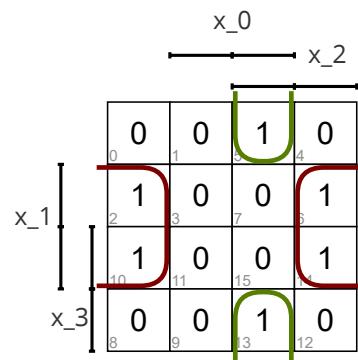
Typesetting math: 59%

Überdecken von 1-Mengen in KV-Diagrammen

Die Blöcke können auch über die Ränder hinweg gebildet werden.

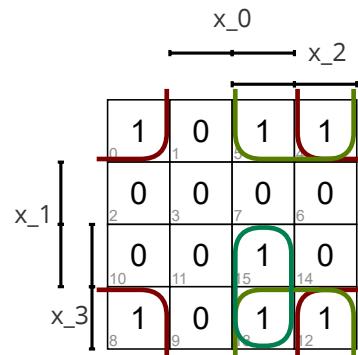
Beispiel 1:

$$y = \overline{x}_1 \overline{x}_0 \vee x_2 \overline{x}_1 x_0$$



Beispiel 2:

$$y = \overline{x}_1 \overline{x}_0 \vee \overline{x}_2 \overline{x}_1 \vee x_3 x_2 x_0$$



Thorsten Thormählen 20 / 34

Typesetting math: 59%

Typesetting math: 59%

Beispiel: 1-bit binärer Volladdierer

Eingänge: Summanden a und b , Carry-in c_{in}

Ausgänge: Summe s , Carry-out c_{out}

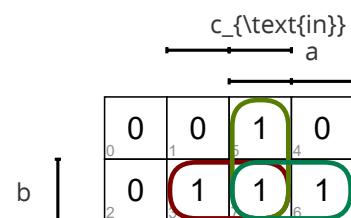
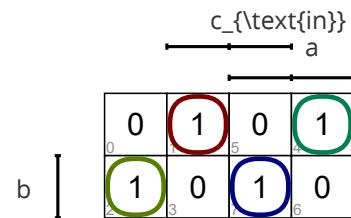


Volladdierer

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s = \overline{a} \overline{b} c_{\text{in}} \lor \overline{a} b \overline{c}_{\text{in}} \lor a \overline{b} c_{\text{in}} \lor abc_{\text{in}}$$

$$c_{\text{out}} = b c_{\text{in}} \lor a c_{\text{in}} \lor a b$$

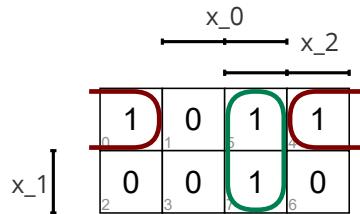


Thorsten Thormählen 21 / 34

Typesetting math: 59%

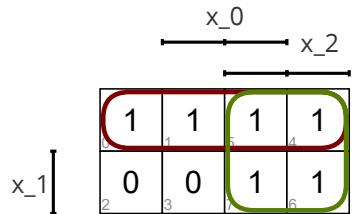
Weitere Beispiele mit 3 Variablen

$$\mathrm{f}(x_2, x_1, x_0) = \bigvee_{i \in \{0, 4, 5, 7\}} m_i$$



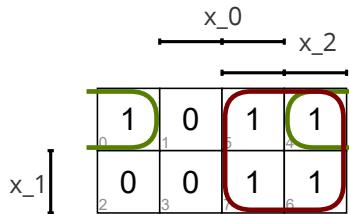
$$\mathrm{f}(x_2, x_1, x_0) = \overline{x}_1 \overline{x}_0 \vee x_2 x_0$$

$$\mathrm{f}(x_2, x_1, x_0) = \bigvee_{i \in \{0, 1, 4, 5, 6, 7\}} m_i$$



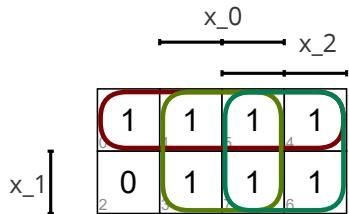
$$\mathrm{f}(x_2, x_1, x_0) = \overline{x}_1 \vee x_2$$

$$\mathrm{f}(x_2, x_1, x_0) = \bigvee_{i \in \{0, 4, 5, 6, 7\}} m_i$$



$$\mathrm{f}(x_2, x_1, x_0) = \overline{x}_1 \overline{x}_0 \vee x_2$$

$$\mathrm{f}(x_2, x_1, x_0) = \bigvee_{i \in \{0, 1, 3, 4, 5, 6, 7\}} m_i$$



$$\mathrm{f}(x_2, x_1, x_0) = \overline{x}_1 \vee \overline{x}_0 \vee x_2$$

Thorsten Thormählen 22 / 34

Typesetting math: 59%

Typesetting math: 59%

Weitere Beispiele mit 4 Variablen

Beispiel 1:

Aufgabe:

$$\mathrm{f}(x_3, x_2, x_1, x_0) = \bigvee \limits_{i \in \{0, 3, 4, 8, 9, 10, 11, 12, 13, 14, 15\}} m_i$$

Lösung:

$$\mathrm{f}(x_3, x_2, x_1, x_0) = x_3 \lor \overline{x}_1 \overline{x}_0 \lor \overline{x}_2 x_1 x_0$$

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

Beispiel 2:

Aufgabe:

$$\mathrm{f}(x_3, x_2, x_1, x_0) = \bigvee \limits_{i \in \{3, 7, 9, 10, 11, 13, 14, 15\}} m_i$$

Lösung:

$$\mathrm{f}(x_3, x_2, x_1, x_0) = x_1 x_0 \lor x_3 x_0 \lor x_3 x_1$$

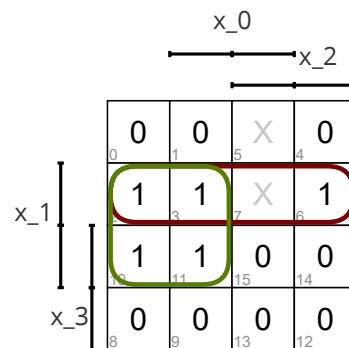
x_0	x_2		
x_1	0 0 0 0	0 1 1 0	0 0 0 0
x_3	1 1 1 1	1 1 1 1	1 1 1 1
0 2 4 6 8 10 12 14	1 3 5 7 9 11 13 15	0 2 4 6 8 10 12 14	1 3 5 7 9 11 13 15

Typesetting math: 59%

Don't cares in KV-Diagrammen

Don't cares können als Einsen oder Nullen behandelt werden, je nachdem, was mehr Vorteile bietet.

x_3	x_2	x_1	x_0	y	KV-Zelle
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	2
0	0	1	1	1	3
0	1	0	0	0	4
0	1	0	1	X	5
0	1	1	0	1	6
0	1	1	1	X	7
1	0	0	0	0	8
1	0	0	1	0	9
1	0	1	0	1	10
1	0	1	1	1	11
1	1	0	0	0	12
1	1	0	1	0	13
1	1	1	0	0	14
1	1	1	1	0	15



$$y = \overline{x}_3 x_1 \lor \overline{x}_2 x_1$$

Typesetting math: 59%

Terminologie KV-Diagramme

Implikant

Monom (bzw. zugehöriger Block), der eine Untermenge der 1-Menge (oder don't cares) abdeckt
z.B. 1 x 1 Block, 2 x 1 Block, 2 x 2 Block, 4 x 2 Block, usw.

Primimplikant

Primimplikanten können nicht (mehr) mit anderen benachbarten Implikanten zusammengefasst werden, um einen größeren Block zu bilden

Essentieller Primimplikant

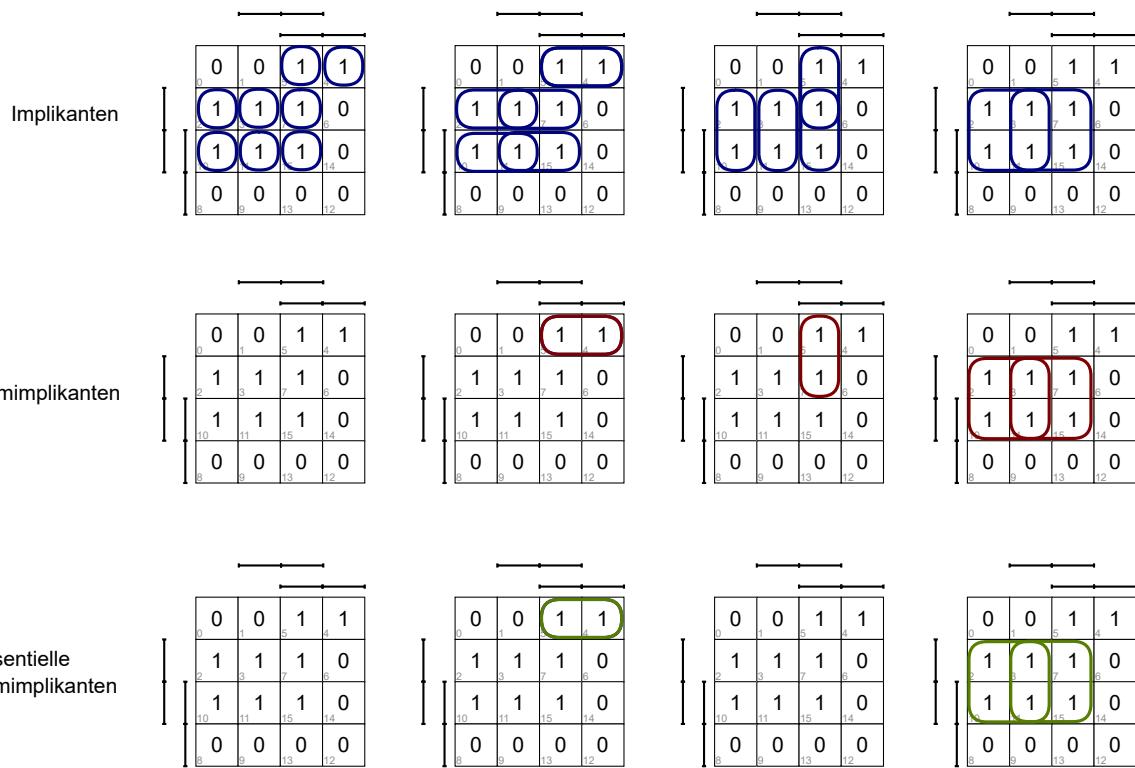
Ein Primimplikant ist essentiell, wenn er als einziger Primimplikant ein bestimmtes Element der 1-Menge abdeckt

Ein essentieller Primimplikant wird somit in jedem Fall für die Abdeckungen der 1-Menge benötigt

Don't cares werden genutzt, um Primimplikanten zu bilden, aber nicht, um einen Primimplikanten als essentiell anzusehen

Typesetting math: 59%

Terminologie KV-Diagramme



Thorsten Thormählen 26 / 34

Typesetting math: 59%

Typesetting math: 59%

Vorgehensweise zum Finden des minimalen booleschen Ausdrucks aus einem KV-Diagramm

Schritt 1: Finde alle Primimplikanten durch Zusammenfassen horizontaler und vertikaler benachbarter 1-en zu möglichst großen Blöcken

auch über die Ränder hinweg

Höhe und Breite der Blöcke müssen Potenzen von 2 sein, also 1, 2, 4, 8 usw.

Schritt 2: Überdecke die 1-Menge im KV-Diagramm mit einer minimalen Auswahl von Primimplikaten

Wird eine 1 nur von einem bestimmten Primimplikanten überdeckt, so ist dieser essentiell und ist Teil der Überdeckungsmenge

Alle 1-en, die von einem essentiellen Primimplikanten überdeckt werden, brauchen nicht mehr untersucht zu werden

Wenn noch 1-en existieren, die nicht durch essentielle Primimplikanten abgedeckt sind, wähle die kleinste Anzahl von Primimplikanten, die die verbleibenden 1-en abdecken. Dabei werden Primimplikanten bevorzugt, die zu großen Blöcken gehören

Typesetting math: 59%

Quiz

Frage: Was ist die minimale Oder-Verknüpfung von Monomen für folgende boolesche Funktion?

x_0		x_2	
		— — — —	— — — —
x_1		— — — —	— — — —
x_3		— — — —	— — — —
0	1	0	0
2	3	7	6
10	11	15	14
8	9	13	12

Antwort 1: $x_1 x_0$

Antwort 2: $\overline{x}_3 \overline{x}_2 x_0 \vee \overline{x}_3 x_2 x_1 \vee x_3 \overline{x}_2 x_1 \vee x_3 x_2 x_0$

Antwort 3: $x_1 x_0 \vee \overline{x}_3 x_2 x_1 \vee x_3 \overline{x}_2 x_1$

Am Online-Quiz teilnehmen durch Besuch der Webseite:

www.onlineclicker.org

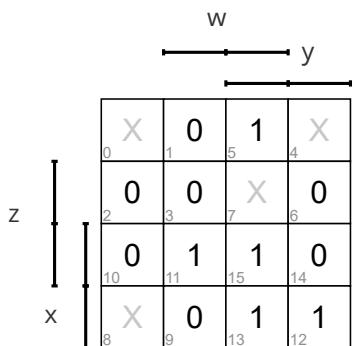
Thorsten Thormählen 28 / 34

Typesetting math: 59%

Typesetting math: 59%

Quiz

Frage: Was ist die minimale Oder-Verknüpfung von Monomen für folgende boolesche Funktion?



		w	
		—	y
	z	—	—
x	—	—	—
		0 1 5 4	X
	2	0 0 7 6	0
	10	0 1 1 0	
	8	X 0 1 1	1

Antwort 1: $zw \vee y \overline{z}$

Antwort 2: $yw \vee \overline{y} \overline{z} \overline{w}$

Antwort 3: $x z w \vee y \overline{z}$

Am Online-Quiz teilnehmen durch Besuch der Webseite:
www.onlineclicker.org

Thorsten Thormählen 29 / 34

Typesetting math: 59%

Typesetting math: 59%

Interaktiver KV-Diagramm-Rechner und Lernprogramm

Durch Klicken auf die Felder kann die boolesche Funktion verändert werden

Variablen:

Don't-Cares erlauben:

Ergebnis verstecken:

		x_0	
		x_2	
		x_1	x_3
0	0	0	0
2	0	0	0
10	0	0	0
8	0	0	0

$$y = 0$$

Thorsten Thormählen 30 / 34

Typesetting math: 59%

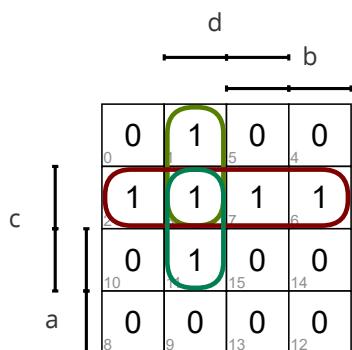
Typesetting math: 59%

Entwurfsbeispiel: 2-Bit-Komparator

$$l = (ab < cd) \quad (\text{less})$$

$$e = (ab == cd) \quad (\text{equal})$$

$$g = (ab > cd) \quad (\text{greater})$$



Abgelesene Lösung für l :

$$l = \overline{a} \overline{b} d \lor \overline{a} c \lor \overline{b} c$$

a	b	c	d	l	e	g	KV-Zelle
0	0	0	0	0	1	0	0
0	0	0	1	1	0	0	1
0	0	1	0	1	0	0	2
0	0	1	1	1	0	0	3
0	1	0	0	0	0	1	4
0	1	0	1	0	1	0	5
0	1	1	0	1	0	0	6
0	1	1	1	1	0	0	7
1	0	0	0	0	0	1	8
1	0	0	1	0	0	1	9
1	0	1	0	0	1	0	10
1	0	1	1	1	0	0	11
1	1	0	0	0	0	1	12
1	1	0	1	0	0	1	13
1	1	1	0	0	0	1	14
1	1	1	1	0	1	0	15

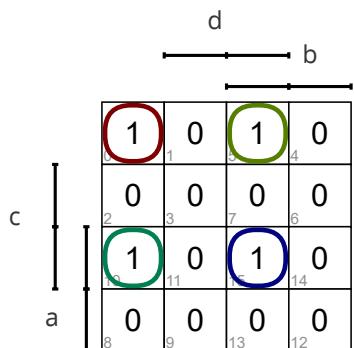
Typesetting math: 59%

Entwurfsbeispiel: 2-Bit-Komparator

$$l = (ab < cd) \quad (\text{less})$$

$$e = (ab == cd) \quad (\text{equal})$$

$$g = (ab > cd) \quad (\text{greater})$$



Abgelesene Lösung für e:

$$e =$$

$$\overline{a}\overline{b}\overline{c}\overline{d} \lor \overline{a}b\overline{c}d \lor a\overline{b}c\overline{d} \lor abcd$$

Dies ist die minimale Oder-Verknüpfung von Monomen, aber es geht noch besser:

$$e = (\overline{a}\overline{c} \land (\overline{b}\overline{d} \lor bd)) \lor (a\overline{c} \land \overline{b}\overline{d}) \quad 32/34$$

$$(\overline{b}\overline{d} \lor bd) = (\overline{a}\overline{c} \lor a\overline{c})$$

$$(\overline{b}\overline{d} \lor bd) = (a \rightarrow c)(b \rightarrow d)$$

a	b	c	d	l	e	g	KV-Zelle
0	0	0	0	0	1	0	0
0	0	0	1	1	0	0	1
0	0	1	0	1	0	0	2
0	0	1	1	1	0	0	3
0	1	0	0	0	0	1	4
0	1	0	1	0	1	0	5
0	1	1	0	1	0	0	6
0	1	1	1	1	0	0	7
1	0	0	0	0	0	1	8
1	0	0	1	0	0	1	9
1	0	1	0	0	1	0	10
1	0	1	1	1	0	0	11
1	1	0	0	0	0	1	12
1	1	0	1	0	0	1	13
1	1	1	0	0	0	1	14
1	1	1	1	0	1	0	15

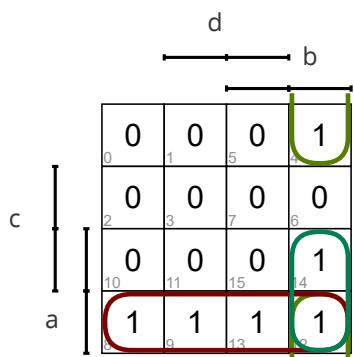
Typesetting math: 59%

Entwurfsbeispiel: 2-Bit-Komparator

$$l = (ab < cd) \quad (\text{less})$$

e = (ab == cd) (equal)

$$g = (ab > cd) \quad (\text{greater})$$



Abgelesene Lösung für g:

Ausgewählte Lösung für g:
 $g = b \overline{c} \overline{d} \vee a \overline{c} \vee ab\overline{d}$

a	b	c	d	l	e	g	KV-Zelle
0	0	0	0	0	1	0	0
0	0	0	1	1	0	0	1
0	0	1	0	1	0	0	2
0	0	1	1	1	0	0	3
0	1	0	0	0	0	1	4
0	1	0	1	0	1	0	5
0	1	1	0	1	0	0	6
0	1	1	1	1	0	0	7
1	0	0	0	0	0	1	8
1	0	0	1	0	0	1	9
1	0	1	0	0	1	0	10
1	0	1	1	1	0	0	11
1	1	0	0	0	0	1	12
1	1	0	1	0	0	1	13
1	1	1	0	0	0	1	14
1	1	1	1	0	1	0	15

Thorsten Thormählen 33 / 34

Typesetting math: 59%

Typesetting math: 59%

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) .]

Thorsten Thormählen 34 / 34

Typeetting math: 59%

Typesetting math: 59%

Technische Informatik

Minimierung mit Quine-McCluskey

Thorsten Thormählen
22. November 2022
Teil 5, Kapitel 3

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Minimierung eines booleschen Ausdrucks mit dem Quine-McCluskey-Verfahren

Finden der Primimplikanten (Phase 1)

Aufstellen der Primimplikantentafel (Phase 2)

Lösen des Überdeckungsproblems (Phase 3)

Lösen zyklischer Überdeckungsprobleme mit dem Verfahren von Petrick

Quine-McCluskey-Verfahren

Das Quine-McCluskey-Verfahren erlaubt, boolesche Funktionen zu minimieren

Während bei KV-Diagrammen Funktionen mit bis zu 4 Variablen leicht minimiert werden können, ist das Quine-McCluskey-Verfahren auch für Funktionen mit mehr Variablen geeignet

Großer Vorteil des Quine-McCluskey-Verfahrens ist, dass es sich relativ einfach auf einem Computer implementieren lässt

Quine-McCluskey-Verfahren

Eingabe: Eine Funktion $y = f(x_{n-1}, \dots, x_1, x_0)$ der Stelligkeit n

Angabe z.B. durch DNF

Beispiel:

$$\begin{aligned}y &= f(x_{n-1}, \dots, x_1, x_0) = \\&= m_0 \vee m_4 \vee m_6 \vee m_{11} \vee m_{12} \vee m_{13} \vee m_{14} \\&= \bar{x}_3 \bar{x}_2 \bar{x}_1 \bar{x}_0 \vee \bar{x}_3 x_2 \bar{x}_1 \bar{x}_0 \vee \\&\quad \bar{x}_3 x_2 x_1 \bar{x}_0 \vee x_3 \bar{x}_2 x_1 x_0 \vee \\&\quad x_3 x_2 \bar{x}_1 \bar{x}_0 \vee x_3 x_2 \bar{x}_1 x_0 \vee \\&\quad x_3 x_2 x_1 \bar{x}_0\end{aligned}$$

oder Wahrheitstafel (siehe rechts)

Ausgabe: Ein minimaler boolescher Ausdruck als ODER-Verknüpfung von Monomen (= UND-Verknüpfungen von Literalen)

Möglichst wenig ODER-Verknüpfungen

Möglichst wenig UND-Verknüpfungen

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

Quine-McCluskey-Verfahren

Die Minimierung nach dem Quine-McCluskey-Verfahren läuft in 3 Phasen ab:

Phase 1: Finden der Primimplikanten

Phase 2: Aufstellen der Primimplikantentafel

Phase 3: Lösen des Überdeckungsproblems

Phase 1: Finden der Primimplikanten

Um die Primimplikanten zu finden, wird (wie bei den KV-Diagrammen) das Vereinigungstheorem verwendet:

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

Bei KV-Diagrammen wird das Vereinigungstheorem rekursiv angewendet, indem immer größere Blöcke aus benachbarten Blöcken gebildet werden

Beim Quine-McCluskey-Verfahren wird analog vorgegangen, nur dass statt mit graphischen Blöcken mit Tabelleneinträgen gearbeitet wird

Phase 1: Finden der Primimplikanten

Die erste Tabelle enthält die Implikanten der Ordnung 0. Sie wird aus der 1-Menge der Wahrheitstafel extrahiert

Die zweite Tabelle enthält die Implikanten der Ordnung 1. Sie wird aus der vorgegangenen Tabelle konstruiert, indem diejenigen Zeilen zusammengefasst werden, die sich in genau einer Variablen unterscheiden

Implikanten, die zusammengefasst wurden, werden geeignet markiert (hier wird das Zeichen → verwendet)

Selbst wenn Implikanten schon zusammengefasst wurden, stehen sie noch zur Verfügung, um mit anderen Implikanten zusammengefasst zu werden

Implikanten, die nicht zusammengefasst wurden, sind die gesuchten Primimplikanten. Sie werden ebenfalls markiert (hier wird das Zeichen ✓ verwendet)

Dieses Verfahren wird rekursiv für Implikanten höherer Ordnung fortgesetzt bis keine Zusammenfassung mehr möglich ist

Phase 1: Finden der Primimplikanten

Wahrheitstafel:

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

Implikanten (Ordnung 0):

	x_3	x_2	x_1	x_0	
0:	0	0	0	0	→
4:	0	1	0	0	→
6:	0	1	1	0	→
11:	1	0	1	1	✓
12:	1	1	0	0	→
13:	1	1	0	1	→
14:	1	1	1	0	→

Phase 1: Finden der Primimplikanten

Implikanten (Ordnung 0): Implikanten (Ordnung 1): Implikanten (Ordnung 2):

	x_3	x_2	x_1	x_0	
0:	0	0	0	0	→
4:	0	1	0	0	→
6:	0	1	1	0	→
11:	1	0	1	1	✓
12:	1	1	0	0	→
13:	1	1	0	1	→
14:	1	1	1	0	→

	x_3	x_2	x_1	x_0	
0, 4:	0	-	0	0	✓
4, 6:	0	1	-	0	→
4, 12:	-	1	0	0	→
6, 14:	-	1	1	0	→
12, 13:	1	1	0	-	✓
12, 14:	1	1	-	0	→

	x_3	x_2	x_1	x_0	
4, 6, 12, 14:	-	1	-	0	✓

Phase 2: Aufstellen der Primimplikantentafel

Die Primimplikantentafel besteht aus allen Primimplikanten
(diese wurden in Phase 1 mit ✓ markiert)

Zeilen:

Pro Primimplikant jeweils eine Zeile

Spalten:

Jede Spalte steht für einen Minterm m_i der DNF

In jeder Spalte wird markiert, ob der Primimplikant den Minterm überdeckt
(hier wird das Zeichen ○ verwendet)

Ist der Primimplikant der einzige Primimplikant, der diesen Minterm überdeckt, so
wird er "essentieller Primimplikant" genannt und besonders markiert
(hier wird das Zeichen ● verwendet)

Essentielle Primimplikanten müssen in jedem Fall Teil des minimalen booleschen
Ausdrucks sein und werden sofort übertragen

Phase 2: Aufstellen der Primimplikantentafel

Implikanten (Ordnung 0): Implikanten (Ordnung 1): Implikanten (Ordnung 2):

	x_3	x_2	x_1	x_0	
0:	0	0	0	0	→
4:	0	1	0	0	→
6:	0	1	1	0	→
11:	1	0	1	1	✓
12:	1	1	0	0	→
13:	1	1	0	1	→
14:	1	1	1	0	→

	x_3	x_2	x_1	x_0	
0, 4:	0	-	0	0	✓
4, 6:	0	1	-	0	→
4, 12:	-	1	0	0	→
6, 14:	-	1	1	0	→
12, 13:	1	1	0	-	✓
12, 14:	1	1	-	0	→

	x_3	x_2	x_1	x_0	
4, 6, 12, 14:	-	1	-	0	✓

Primimplikantentafel:

	x_3	x_2	x_1	x_0	0	4	6	11	12	13	14
4, 6, 12, 14:	-	1	-	0		○	●		○		●
0, 4:	0	-	0	0	●	○					
12, 13:	1	1	0	-					○	●	
11:	1	0	1	1				●			

Extrahierte essentielle Primimplikanten: $(\bar{x}_3 \bar{x}_1 \bar{x}_0)$, $(x_2 \bar{x}_0)$, $(x_3 \bar{x}_2 x_1 x_0)$, $(x_3 x_2 \bar{x}_1)$

Phase 2: Aufstellen der Primimplikantentafel

Ist es zufällig so, dass die essentiellen Primimplikanten gemeinsam alle Minterme überdecken, kann der Algorithmus bereits hier beendet werden

Der minimale boolesche Ausdruck für das gezeigte Beispiel lautet daher:

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

Ist dies nicht der Fall, muss in Phase 3 eine geeignete Überdeckung gefunden werden

Phase 3: Lösen des Überdeckungsproblems

Die essentiellen Primimplikanten müssen in jedem Fall Teil des minimalen booleschen Ausdrucks sein

Daher können nach der Übertragung in den booleschen Ausdruck diese Zeilen in der Primimplikantentafel gestrichen werden

Ebenfalls können diejenigen Spalten gestrichen werden, die von den essentiellen Primimplikanten überdeckt werden

Es entsteht eine "reduzierte Primimplikantentafel"

Phase 3: Lösen des Überdeckungsproblems

Die reduzierte Primimplikantentafel kann folgendermaßen vereinfacht werden:

Zeilenregel:

Existiert für eine Zeile z_i eine andere Zeile z_j , die die gleichen und noch weitere Minterme überdeckt, so wird z_i gestrichen

Überdecken zwei Zeilen die gleichen Minterme, so wird die Zeile des Primimplikants behalten, der weniger Literale enthält

Spaltenregel

Wird ein Minterm immer überdeckt, wenn auch ein anderer überdeckt wird (alle Primterme behandeln diese beiden gleich), so kann eine der beiden zugehörigen Spalten gestrichen werden

Die Spaltenregel kann die reduzierte Primimplikantentafel übersichtlicher machen, ist aber letztendlich nicht wichtig für das Ergebnis des Verfahrens

Phase 3: Lösen des Überdeckungsproblems

Wahrheitstafel:

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

Implikanten (Ordnung 0):

	x_3	x_2	x_1	x_0	
0:	0	0	0	0	→
2:	0	0	1	0	→
3:	0	0	1	1	→
7:	0	1	1	1	→
8:	1	0	0	0	→
9:	1	0	0	1	→
12:	1	1	0	0	→
14:	1	1	1	0	→
15:	1	1	1	1	→

Implikanten (Ordnung 1):

	x_3	x_2	x_1	x_0	
0, 2:	0	0	-	0	✓
0, 8:	-	0	0	0	✓
2, 3:	0	0	1	-	✓
3, 7:	0	-	1	1	✓
7, 15:	-	1	1	1	✓
8, 9:	1	0	0	-	✓
8, 12:	1	-	0	0	✓
12, 14:	1	1	-	0	✓
14, 15:	1	1	1	-	✓

Phase 3: Lösen des Überdeckungsproblems

Primimplikantentafel:

	x_3	x_2	x_1	x_0	0	2	3	7	8	9	12	14	15	
0, 2:	0	0	-	0	○	○								($\bar{x}_3\bar{x}_2\bar{x}_0$)
0, 8:	-	0	0	0	○				○					($\bar{x}_2\bar{x}_1\bar{x}_0$)
2, 3:	0	0	1	-		○	○							($\bar{x}_3\bar{x}_2x_1$)
3, 7:	0	-	1	1			○	○						($\bar{x}_3x_1x_0$)
7, 15:	-	1	1	1				○				○		($x_2x_1x_0$)
8, 9:	1	0	0	-					○	●				($x_3\bar{x}_2\bar{x}_1$)
8, 12:	1	-	0	0					○		○			($x_3\bar{x}_1\bar{x}_0$)
12, 14:	1	1	-	0							○	○		($x_3x_2\bar{x}_0$)
14, 15:	1	1	1	-							○	○		($x_3x_2x_1$)

Extrahierte essentielle Primimplikanten: ($x_3\bar{x}_2\bar{x}_1$)

Phase 3: Lösen des Überdeckungsproblems

Reduzierte Primimplikantentafel (Iteration 0):

	x_3	x_2	x_1	x_0	0	2	3	7	12	14	15	
0, 2:	0	0	-	0	●	○						$(\bar{x}_3\bar{x}_2\bar{x}_0)$
2, 3:	0	0	1	-		○	○					$(\bar{x}_3\bar{x}_2x_1)$
3, 7:	0	-	1	1			○	○				$(\bar{x}_3x_1x_0)$
7, 15:	-	1	1	1				○		○		$(x_2x_1x_0)$
12, 14:	1	1	-	0					●	○		$(x_3x_2\bar{x}_0)$
14, 15:	1	1	1	-						○	○	$(x_3x_2x_1)$

Extrahierte essentielle Primimplikanten: $(\bar{x}_3\bar{x}_2\bar{x}_0)$, $(x_3x_2\bar{x}_0)$

Reduzierte Primimplikantentafel (Iteration 1):

	x_3	x_2	x_1	x_0	3	7	15	
3, 7:	0	-	1	1	●	○		$(\bar{x}_3x_1x_0)$
7, 15:	-	1	1	1		○	●	$(x_2x_1x_0)$

Extrahierte essentielle Primimplikanten: $(\bar{x}_3x_1x_0)$, $(x_2x_1x_0)$

Phase 3: Lösen des Überdeckungsproblems

Nach rekursiver Anwendung des Streichens der Zeilen der essentiellen Primimplikanten und Vereinfachung der reduzierten Primimplikantentafel terminiert das Verfahren häufig

Der minimale boolesche Ausdruck kann dann durch die essentiellen Primimplikanten aus allen Rekursionsschritten gebildet werden

Der minimale boolesche Ausdruck für das gezeigte Beispiel lautet daher:

$$y = (x_3\bar{x}_2\bar{x}_1) \vee (\bar{x}_3\bar{x}_2\bar{x}_0) \vee (x_3x_2\bar{x}_0) \vee (\bar{x}_3x_1x_0) \vee (x_2x_1x_0)$$

Es gibt aber auch zyklische Überdeckungsprobleme, bei denen das Verfahren nicht terminiert

Zyklische Überdeckungsprobleme

Ob ein zyklisches Überdeckungsprobleme vorliegt, kann daran erkannt werden, dass in der Primimplikantentafel (oder der reduzierten Primimplikantentafel) keine essentiellen Primimplikanten gefunden werden können

Damit ist es nicht eindeutig, welcher Primimplikant in die Lösung aufgenommen werden soll und welcher nicht

Eine einfache Lösung ist es, alle möglichen Kombinationen von Primimplikanten auszuprobieren, und die Kombination zu verwenden, die eine Überdeckung mit minimalen Kosten liefert

Die Anzahl der möglichen Kombinationen steigt jedoch schnell an. Es gibt 2^k Kombinationen, wenn k die Anzahl der beteiligten Primimplikanten ist

Also z.B. $2^{32} = 4294967296$ bei 32 Primimplikanten

Viele dieser Kombinationen liefern gar keine Überdeckung. Andere überdecken Minterme unnötigerweise mehrfach und erzeugen nicht minimale Lösungen.

Wie kann die Anzahl der betrachteten Kombinationen reduziert werden?

Verfahren von Petrick

Zunächst wird jeder Zeile k (bzw. dem korrespondierenden Primterm) eine boolesche Variable p_k zugeordnet

$(p_k = 1)$, wenn der Primterm in der Lösung verwendet wird, sonst $(p_k = 0)$

Mit den Variablen p_k wird ein boolescher Ausdruck aufgestellt, der nur wahr ist, wenn alle Spalten überdeckt werden

Der boolesche Ausdruck besteht dabei aus so vielen UND-Verknüpfungen wie es Spalten gibt:

(Spalte 1 überdeckt) UND (Spalte 2 überdeckt) UND

Die Aufgabe, eine Spalte zu überdecken, kann von einem beliebigen Primterm übernommen werden. Daher wird eine ODER-Verknüpfung aus alle Primtermen gebildet, die die jeweilige Spalte überdecken, z.B.:

$(p_1 \text{ ODER } p_3) \text{ UND } (p_3 \text{ ODER } p_5) \text{ UND } \dots$

Anschließend werden die Klammern aufgelöst und der Ausdruck vereinfacht

Vereinfacht wird dabei mit den beiden Theoremen:

Idempotenz: $p_k \wedge p_k = p_k$ und Absorption: $p_k \vee (p_k \wedge p_j) = p_k$

Nach Auflösung besteht der Ausdruck aus ODER-Verknüpfungen von Monomen der Variablen p_k . Jedes Monom beschreibt eine mögliche Lösung

Verfahren von Petrick

Wahrheitstafel:

	x_2	x_1	x_0	y
0:	0	0	0	1
1:	0	0	1	0
2:	0	1	0	1
3:	0	1	1	1
4:	1	0	0	1
5:	1	0	1	1
6:	1	1	0	0
7:	1	1	1	1

Implikanten (Ordnung 0):

	x_2	x_1	x_0	
0:	0	0	0	→
2:	0	1	0	→
3:	0	1	1	→
4:	1	0	0	→
5:	1	0	1	→
7:	1	1	1	→

Implikanten (Ordnung 1):

	x_2	x_1	x_0	
0, 2:	0	-	0	✓
0, 4:	-	0	0	✓
2, 3:	0	1	-	✓
3, 7:	-	1	1	✓
4, 5:	1	0	-	✓
5, 7:	1	-	1	✓

Verfahren von Petrick

Primimplikantentafel:

	x_2	x_1	x_0	0	2	3	4	5	7
0, 2:	0	-	0	○	○				
0, 4:	-	0	0	○			○		
2, 3:	0	1	-		○	○			
3, 7:	-	1	1			○		○	
4, 5:	1	0	-				○	○	
5, 7:	1	-	1					○	○

$$(\bar{x}_2 \bar{x}_0) \equiv p_0$$

$$(\bar{x}_1 \bar{x}_0) \equiv p_1$$

$$(\bar{x}_2 x_1) \equiv p_2$$

$$(x_1 x_0) \equiv p_3$$

$$(x_2 \bar{x}_1) \equiv p_4$$

$$(x_2 x_0) \equiv p_5$$

$$(p_0 \vee p_1)(p_0 \vee p_2)(p_2 \vee p_3)(p_1 \vee p_4)(p_4 \vee p_5)(p_3 \vee p_5)$$

$$\Leftrightarrow (p_0 \vee p_0 p_2 \vee p_0 p_1 \vee p_1 p_2)(p_1 p_2 \vee p_2 p_4 \vee p_1 p_3 \vee p_3 p_4)(p_3 p_4 \vee p_4 p_5 \vee p_3 p_5 \vee p_5)$$

$$\Leftrightarrow (p_0 \vee p_1 p_2)(p_1 p_2 \vee p_2 p_4 \vee p_1 p_3 \vee p_3 p_4)(p_3 p_4 \vee p_5)$$

$$\Leftrightarrow (p_0 p_1 p_2 \vee p_0 p_2 p_4 \vee p_0 p_1 p_3 \vee p_0 p_3 p_4 \vee p_1 p_2 \vee p_1 p_2 p_4 \vee p_1 p_2 p_3 \vee p_1 p_2 p_3 p_4)(p_3 p_4 \vee p_5)$$

$$\Leftrightarrow (p_0 p_2 p_4 \vee p_0 p_1 p_3 \vee p_0 p_3 p_4 \vee p_1 p_2)(p_3 p_4 \vee p_5)$$

$$\Leftrightarrow (p_0 p_2 p_3 p_4 \vee p_0 p_2 p_4 p_5 \vee p_0 p_1 p_3 p_4 \vee p_0 p_1 p_3 p_5 \vee p_0 p_3 p_4 \vee p_0 p_3 p_4 p_5 \vee p_1 p_2 p_3 p_4 \vee p_1 p_2 p_5)$$

$$\Leftrightarrow (p_0 p_2 p_4 p_5 \vee p_0 p_1 p_3 p_5 \vee p_0 p_3 p_4 \vee p_1 p_2 p_3 p_4 \vee p_1 p_2 p_5)$$

Thorsten Thormählen 24 / 27

Verfahren von Petrick

Jede Lösung kann bezüglich ihrer Kosten bewertet werden

Die geringste Anzahl an Primtermen haben die beiden Lösungen $p_0p_3p_4$ und $p_1p_2p_5$

Jedes Literal p_k steht dabei für einen Primterm, der in die Lösung eingeht

Gibt es gleich viele Literale, wie im gezeigten Beispiel, entscheidet die Länge der einzelnen Primterme (im gezeigten Beispiel auch gleich)

Im gezeigten Beispiel sind daher folgende zwei Lösungen gleichwertig:

$$\text{Lösung 1: } y = (\bar{x}_2\bar{x}_0) \vee (x_1x_0) \vee (x_2\bar{x}_1)$$

$$\text{Lösung 2: } y = (\bar{x}_1\bar{x}_0) \vee (\bar{x}_2x_1) \vee (x_2x_0)$$

Interaktiver Quine-McCluskey Rechner

Durch Klicken auf die grauen Elemente kann die boolesche Funktion verändert werden

Anzahl der Variablen: Don't-Cares erlauben: nein Generiere zufälliges Beispiel

Wahrheitstafel:

	x_2	x_1	x_0	y
0:	0	0	0	0
1:	0	0	1	0
2:	0	1	0	0
3:	0	1	1	0
4:	1	0	0	0
5:	1	0	1	0
6:	1	1	0	0
7:	1	1	1	0

Minimaler boolescher Ausdruck:

$$y = 0$$

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) ,]

Thorsten Thormählen 27 / 27

Technische Informatik

Zeitverhalten und Hazards

Thorsten Thormählen

24. November 2022

Teil 6, Kapitel 1

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

- Zeitverhalten von Schaltsystemen
- Gatterverzögerungen
- Ozillator-Schaltungen
- Hazards und Glitches
- Statische und dynamische Hazards
- Konstruktion Hazard-freier Schaltungen

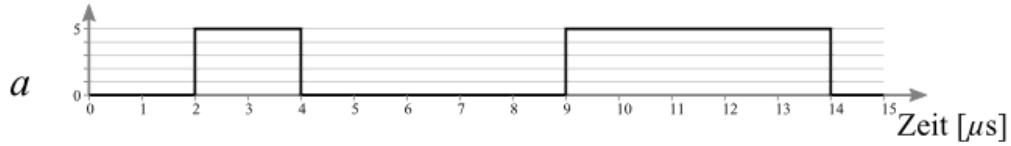
Analyse des Zeitverhaltens von Schaltnetzen

Das Zeitverhalten von Schaltsystemen kann durch Signalverläufe analysiert werden

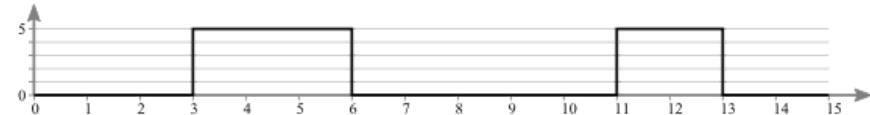
Darstellung der Signalwerte in einer Schaltung abhängig von der Zeit

Notwendig, um zeitliche Kausalitäten und Folgen von Ereignissen zu erkennen

Signal [V]



b



$a \wedge b$



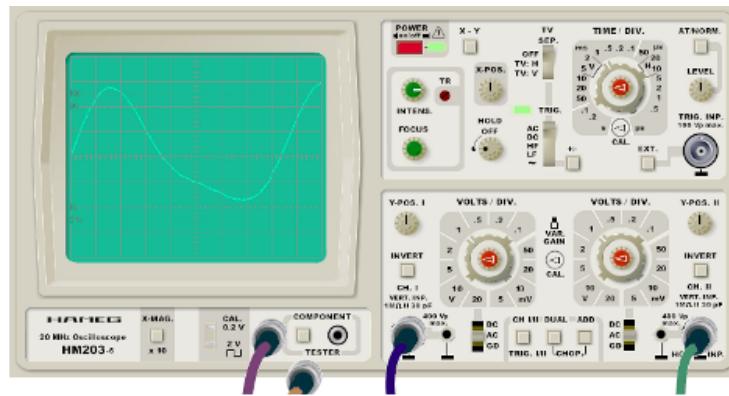
Messen des Zeitverhaltens

Bei Schaltungen aus diskreten Bauelementen kann das Zeitverhalten z.B. mit einem Oszilloskop gemessen werden

Auf der Internetseite www.virtuelles-oszilloskop.de kann ein analoges Oszilloskop ausprobiert werden

Digitale Oszilloskope können analoge Signalverläufe durch einen Analog-Digital-Wandler digital aufzeichnen und erlauben damit die genaue zeitliche Analyse eines Signals

Bei ICs (Integrated Circuits) können normalerweise nur die Signale gemessen werden, die nach außen auf die Pins geführt werden



[Bildquelle:www.virtuelles-oszilloskop.de]

Thorsten Thormählen 6 / 20

Simulation des Zeitverhaltens

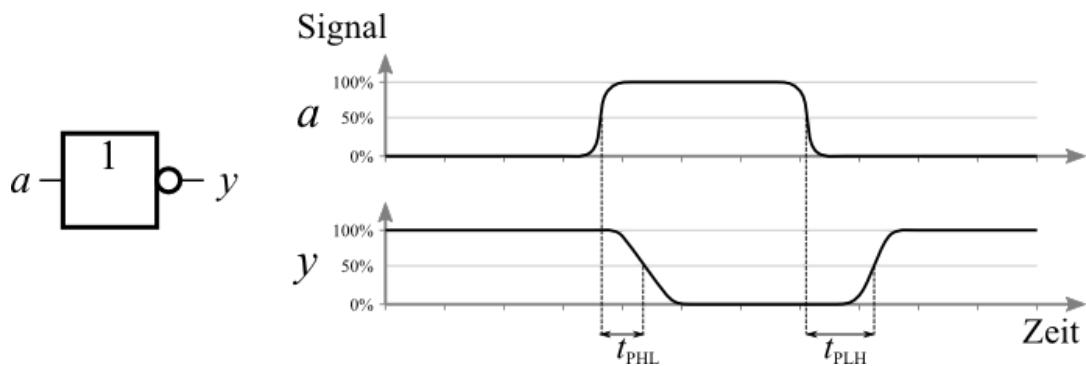
Ein Schaltnetz kann mit einer Simulations-Software nachgebildet werden

Eingabe: Schaltstruktur (d.h. die Gatter und deren Verbindungen) und Stimuli (d.h. die Eingangssignale für das Schaltnetz)

Ausgabe: Signalverläufe

Die Simulation basiert auf mathematischen Modellen der Bauteile

Gatterverzögerung



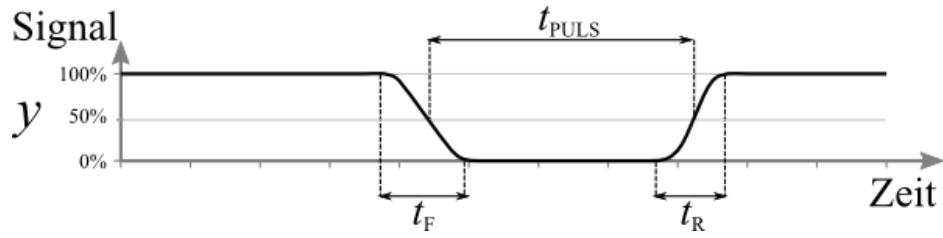
Eine Änderung am Gatter-Eingang bewirkt erst nach einer gewissen Verzögerung eine Änderung am Ausgang

Diese Zeit wird als Gatterverzögerung (engl. propagation delay time) bezeichnet

Die Gatterverzögerung bei einer fallenden Flanke t_{PHL} kann eine andere sein als bei einer steigenden Flanke t_{PLH}

Typische Gatterverzögerungen haben eine Größenordnung von ca. 100 Piko- bis 100 Nanosekunden (je nach Logikfamilie)

Anstiegs-, Abfallzeit, Pulsbreite



Bei Logiksignalen handelt es sich in der Praxis nicht um perfekte Rechtecksignale. Ein Gatter kann die Signalfanken verändern (typischerweise werden die Flanken mit jedem Gatterdurchgang "weicher")

Abfallzeit t_F (engl. fall time): Zeit, die ein Ausgang benötigt, um von HIGH nach LOW umzuschalten

Anstiegzeit t_R (engl. rise time): Zeit, die ein Ausgang benötigt, um von LOW nach HIGH umzuschalten

Pulsbreite t_{PULS} (engl. puls width): Zeit, die ein Ausgang auf LOW bzw. HIGH verbleibt

Amilosim (A Minimalistic Logic Simulator)

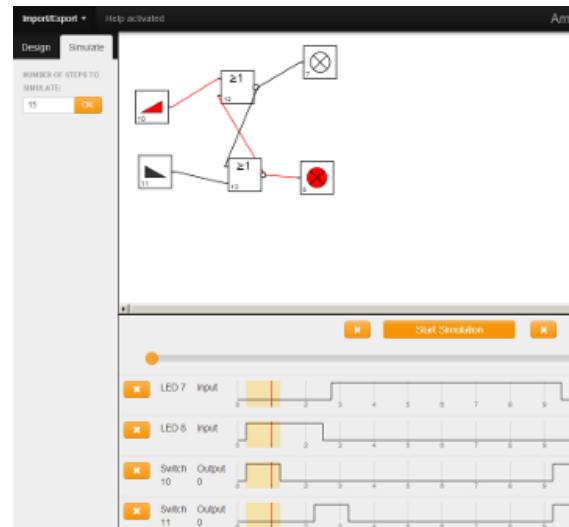
Zur Simulation und Veranschaulichung des Zeitverhaltens von Logikschaltungen wird in dieser Vorlesung [Amilosim](#) verwendet

Amilosim ist eine Webanwendung, die von Mareike Schilling im Rahmen Ihrer Bachelorarbeit (2013) entwickelt wurde

Die Zeitsimulation ist einheitenlos. Jedes Gatter hat immer exakt eine Zeiteinheit Gatterverzögerung.

Dies ist ein sehr grobes und minimalistisches Modell der Realität, ist jedoch ausreichend, um viele Effekte abzubilden

Zur komfortablen Navigation in den Signalverläufen, ermittelt die Software automatisch die maximale Gesamlaufzeit bis sich ein Signal vollständig in einem Schaltnetz (ohne Rückkopplungen) ausgebreitet hat und stellt dies als einen Simulationschritt im Zeitdiagramm dar

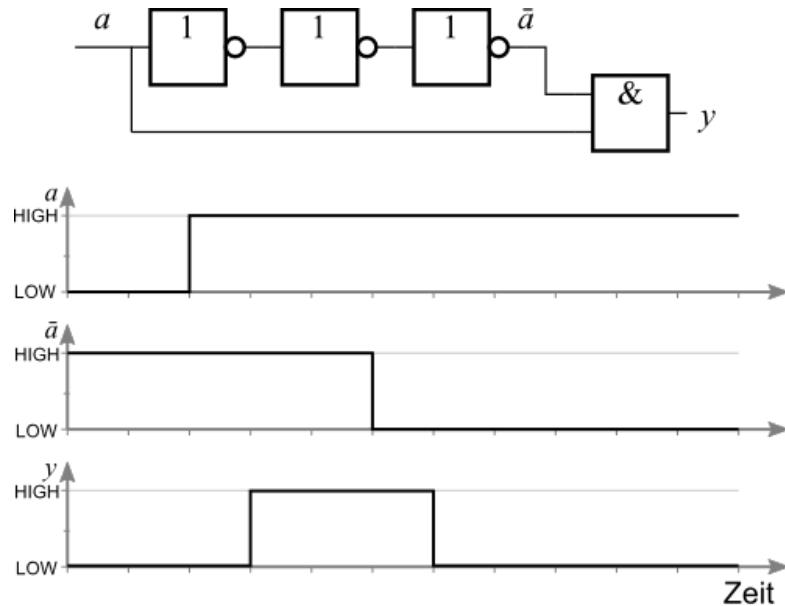


Amilosim

Kurzzeitige Änderungen am Ausgang

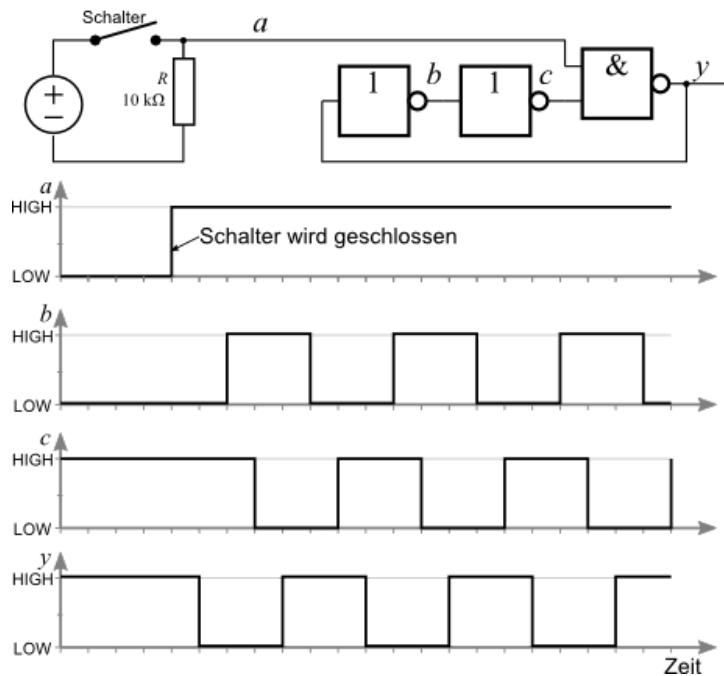
Wenn folgendes Schaltsystem betrachtet wird, müsste der Ausgang des UND-Gatters gemäß boolescher Algebra eigentlich immer Null sein, da $a \wedge \bar{a} = 0$

Durch die Laufzeitverzögerung an den Invertern entsteht jedoch ein kurzer Puls am Ausgang ([Öffnen in Amilosim](#))



Oszillatoren

Eine Oszillatorschaltung ist eine elektronische Schaltung zur Erzeugung einer periodischen Wechselspannung ([Öffnen in Amilosim](#))



Thorsten Thormählen 12 / 20

Hazards und Glitches

Laufzeitverzögerungen können zu gravierenden Problemen in Logikschaltungen führen

Ein *Glitch* ist eine nicht beabsichtigte Signaländerung am Ausgang eines Logikgatters

Ein Glitch kann auftreten, wenn verschiedene Pfade durch ein Schaltsystem unterschiedliche Laufzeiten haben

Ein *Hazard* ist eine Konfiguration, bei der ein Glitch auftreten kann (aber nicht muss)

Glitches (bzw. Hazards) sind kritisch, wenn die Störimpulse ungewollte Auswirkungen in der nachfolgenden Logik erzeugen (bzw. erzeugen können)

Klassifikation von Hazards

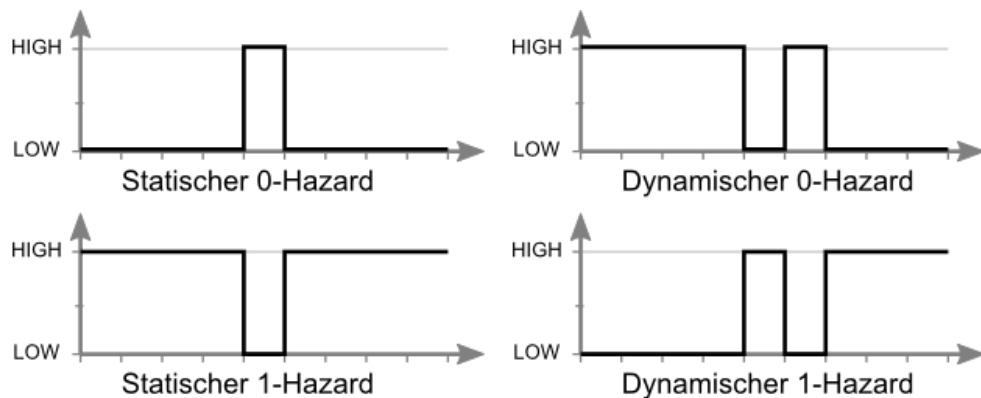
Statischer Hazard

Bei einem statischen Hazard kann ein Wechsel am Eingang einen einmaligen temporären Wechsel des Ausgangs zur Folge haben

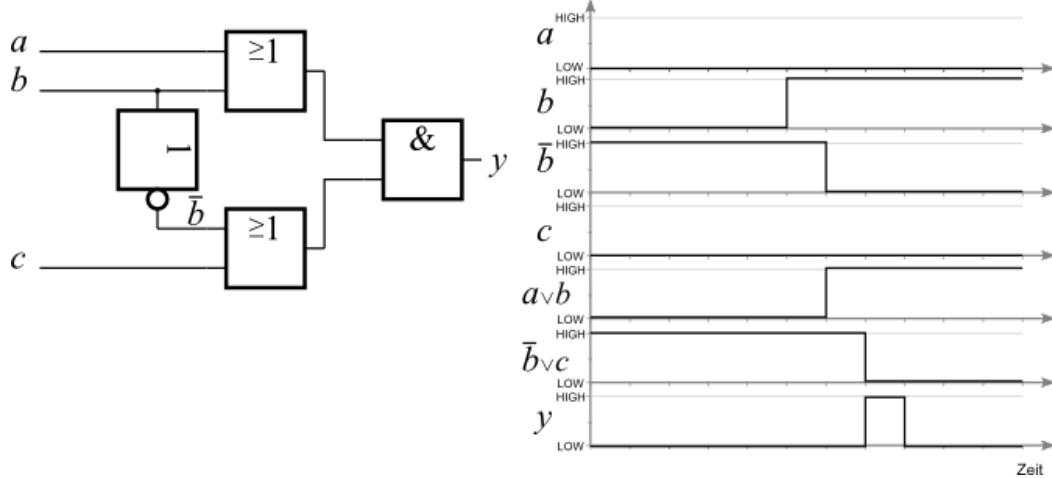
Dynamischer Hazard

Bei einem dynamischen Hazard kann ein Wechsel am Eingang einen mehrfachen Wechsel des Ausgangs zur Folge haben bis sich der Ausgang stabilisiert

Es wird ebenfalls zwischen 0-Hazards und 1-Hazards unterschieden. Bei 0-Hazards sollte das Signal eigentlich 0 sein, bei 1-Hazards eigentlich 1.



Beispiel für einen statischen Hazard



Laut boolescher Algebra müsste eigentlich gelten:

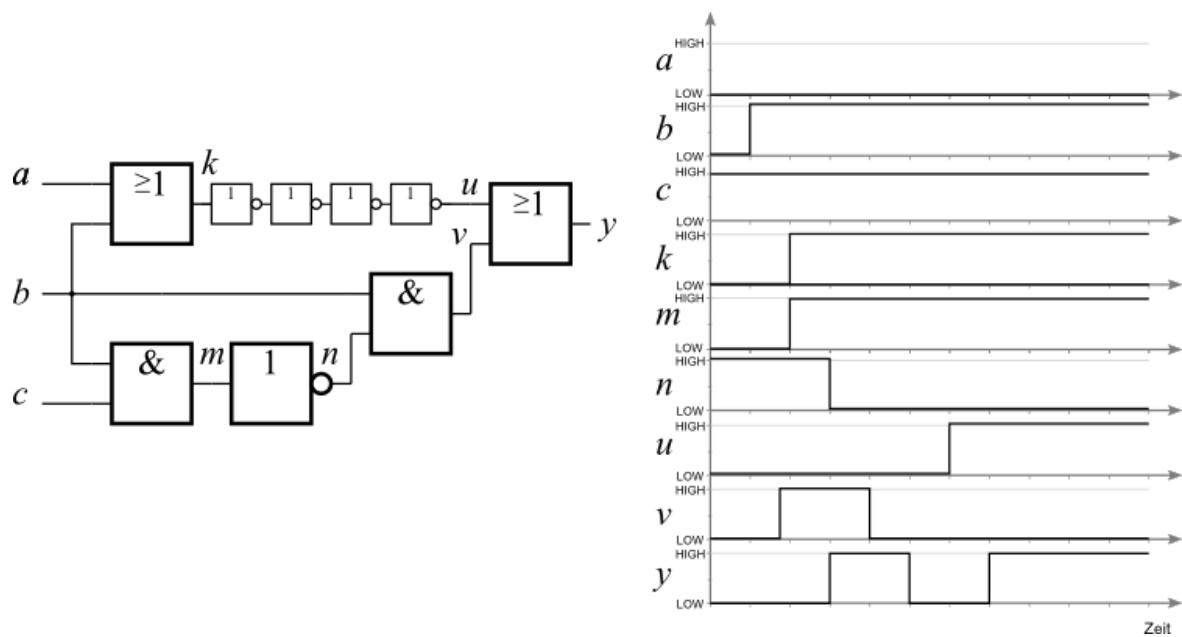
$$y = (a \vee b) \wedge (\neg b \vee c) = a \neg b \vee ac \vee b \neg b \vee bc = a \neg b \vee ac \vee bc$$

mit $a = 0$ und $c = 0$ folgt:

$$y = a \neg b \vee ac \vee bc = 0$$

Es handelt sich also um einen statischen 0-Hazard ([Öffnen in Amilosim](#))

Beispiel für einen dynamischen Hazard



Es handelt sich um einen dynamischen 1-Hazard ([Öffnen in Amilosim](#))

Dieser entsteht hier durch einen statischen 0-Hazard der Variablen v

Konstruktion Hazard-freier Schaltungen

Synchrones Design

Eine Möglichkeit, die Ausbreitung von Glitches zu vermeiden, ist zu warten bis alle Signale stabil sind

Dies kann durch ein Clock-Signal erreicht werden, dass anzeigt, wann ein Wert übernommen werden soll

Dies führt zu einem synchronen Schaltungsdesign

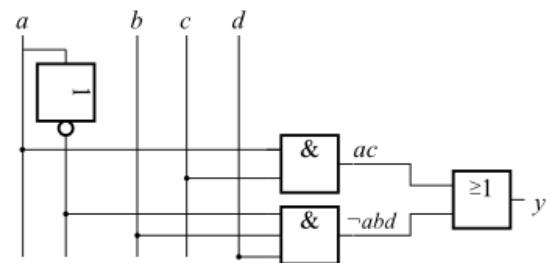
Zusätzliche Gatter

Durch zusätzliche Gatter kann eine Hazard-freie Schaltung realisiert werden

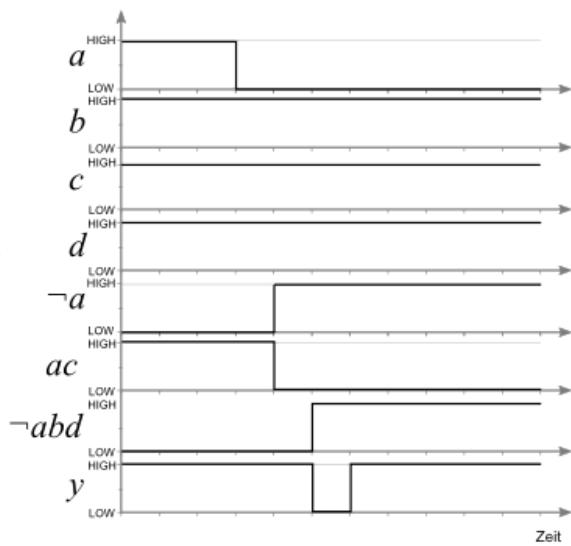
Ein KV-Diagramm kann helfen, Hazards zu entdecken

Konstruktion Hazard-freier Schaltungen

$$y = ac \vee \neg abd$$



	$\neg a \neg b$	$\neg ab$	ab	$a \neg b$
$\neg c \neg d$	0	0	0	0
$\neg cd$	0	1	0	0
cd	0	1	1	1
$c \neg d$	0	0	1	1

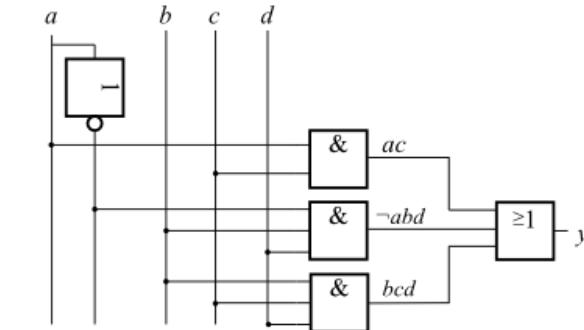


Beobachtung: Hazards entstehen, wenn zwei Primterm-Blöcke überlappungsfrei aneinander grenzen

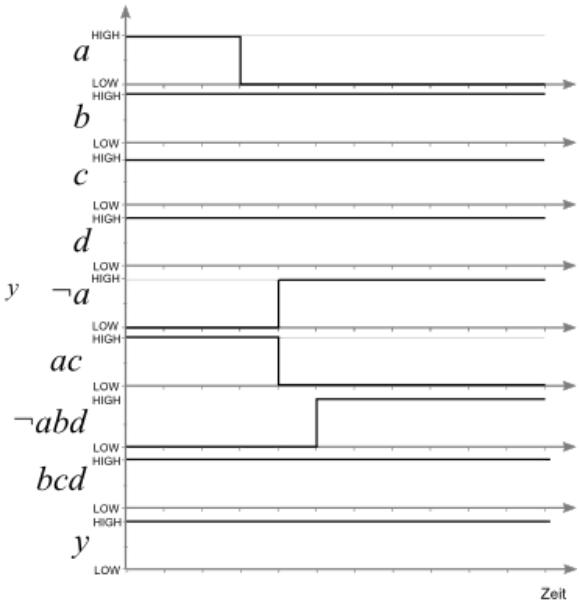
Konstruktion Hazard-freier Schaltungen

Lösung: Mit zusätzlichen Gattern wird ein weiterer Primterm-Block erzeugt, der die Überlappung der 1-Menge garantiert

$$y = ac \vee \neg abd \vee bcd$$



	$\neg a \neg b$	$\neg ab$	ab	$a \neg b$
$\neg c \neg d$	0	0	0	0
$\neg cd$	0	1	0	0
cd	0	1	1	1
$c \neg d$	0	0	1	1



Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) .]

Thorsten Thormählen 20 / 20

Technische Informatik

Reguläre Logikschaltungen

Thorsten Thormählen
30. November 2021
Teil 7, Kapitel 1

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Multiplexer / Demultiplexer

Programmierbare logische Schaltungen

PLAs

PALs

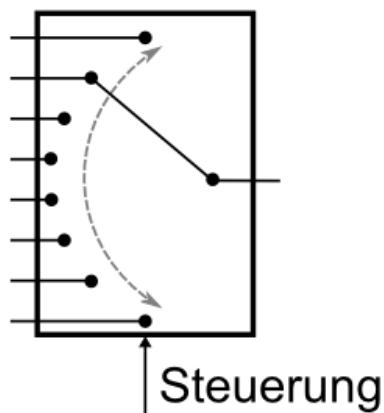
PROMs

Multiplexer / Demultiplexer

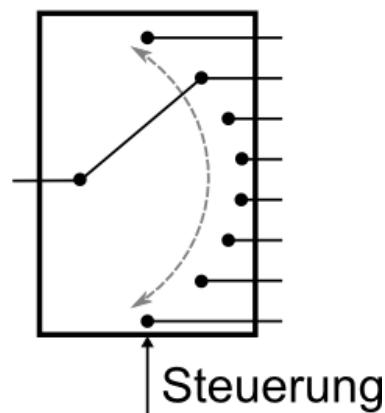
Ein Multiplexer (auch "Mux") schaltet von vielen Eingängen auf einen Ausgang

Ein Demultiplexer (auch "Demux") schaltet von einem Eingang auf viele Ausgänge

Multiplexer



Demultiplexer



Multiplexer

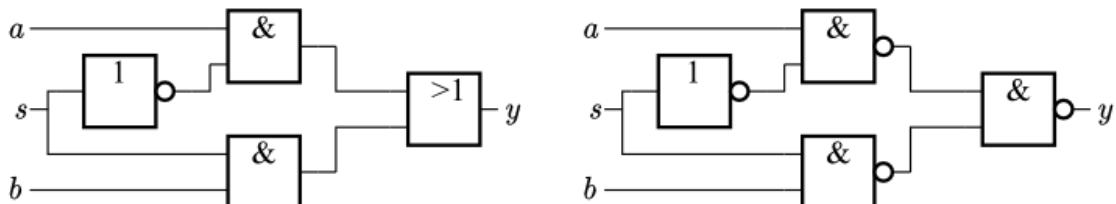
Für einen 2-zu-1 Multiplexer (auch "2:1 Mux") gilt:

Wenn das Steuersignal $s = 1$, dann ist $y = b$, ansonsten $y = a$

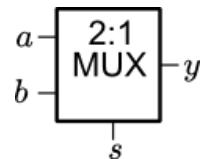
Diese entspricht einem if-then-else:

```
if(s) y = b; else y = a;
```

Ein 2-zu-1 Multiplexer kann leicht aus mehreren Logikgattern aufgebaut werden:



Da er häufig Bestandteil von digitalen Schaltungen ist, bekommt der Multiplexer ein eigenes Symbol



Multiplexer

2-zu-1 Multiplexer (2:1 Mux):

$$y = \bar{s}_0 i_0 \vee s_0 i_1$$

4-zu-1 Multiplexer (4:1 Mux):

$$y = \bar{s}_1 \bar{s}_0 i_0 \vee \bar{s}_1 s_0 i_1 \vee s_1 \bar{s}_0 i_2 \vee s_1 s_0 i_3$$

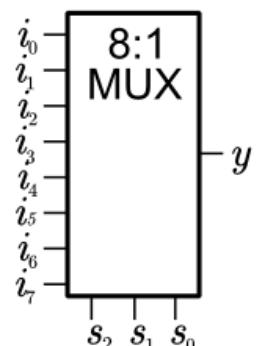
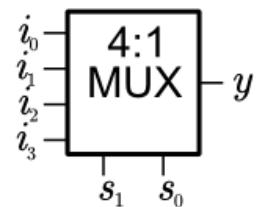
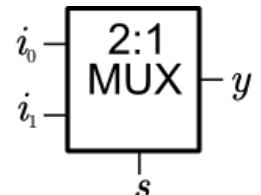
8-zu-1 Multiplexer (8:1 Mux):

$$\begin{aligned} y = & \bar{s}_2 \bar{s}_1 \bar{s}_0 i_0 \vee \bar{s}_2 \bar{s}_1 s_0 i_1 \vee \bar{s}_2 s_1 \bar{s}_0 i_2 \vee \bar{s}_2 s_1 s_0 i_3 \\ & s_2 \bar{s}_1 \bar{s}_0 i_4 \vee s_2 \bar{s}_1 s_0 i_5 \vee s_2 s_1 \bar{s}_0 i_6 \vee s_2 s_1 s_0 i_7 \end{aligned}$$

n -zu-1 Multiplexer ($n:1$ Mux):

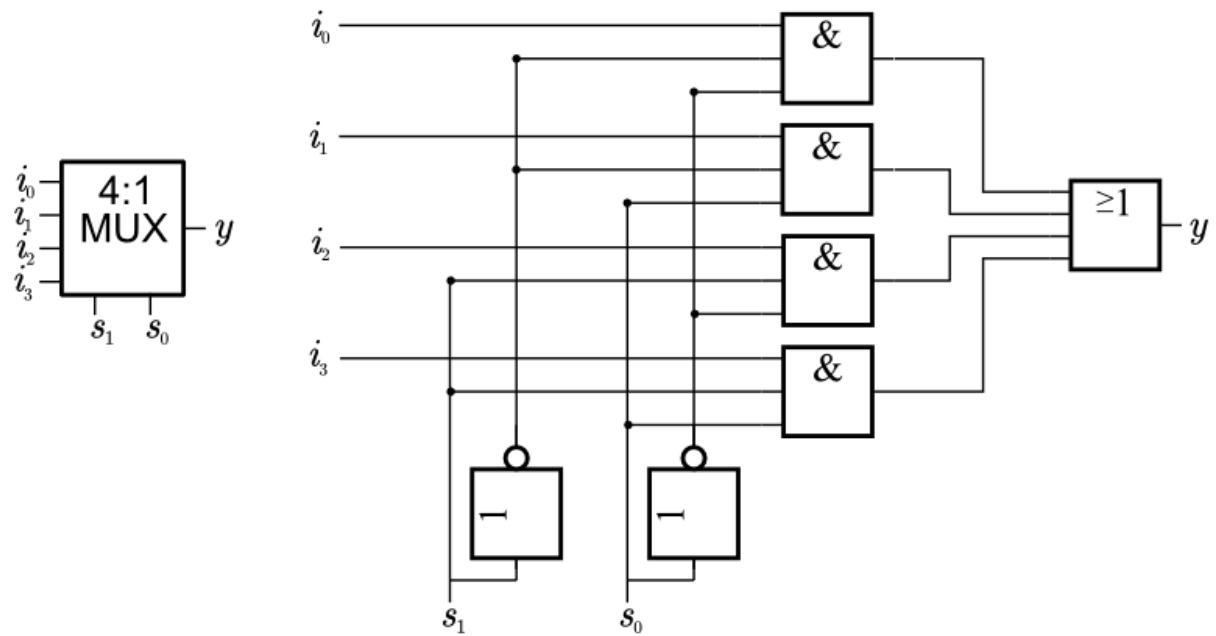
$$y = \bigvee_{j=0}^{n-1} m_j i_j$$

wobei m_j den Minterm der k Steuersignale s_{k-1}, \dots, s_1, s_0 bezeichnet und gilt $n = 2^k$



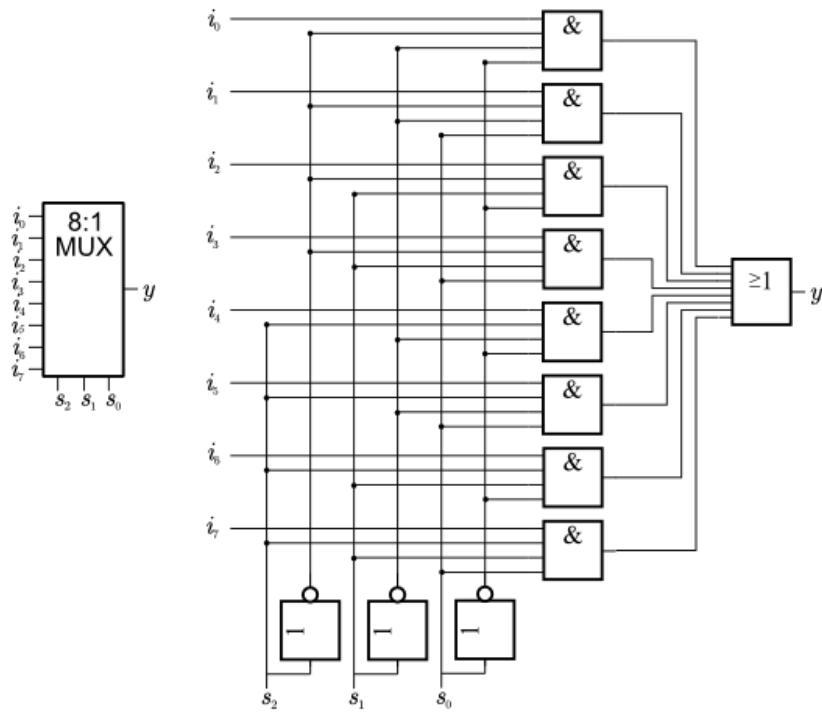
Realisierung von Multiplexern

Ein 4:1 Mux als zweistufige Logik:



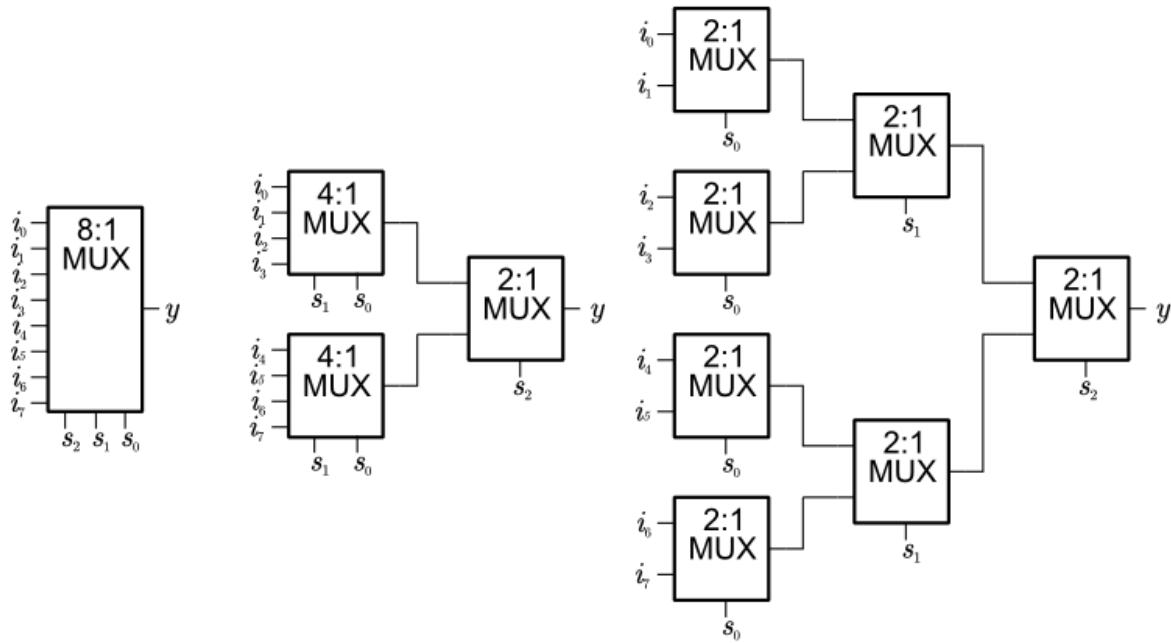
Realisierung von Multiplexern

Ein 8:1 Mux als zweistufige Logik:



Kaskadierung von Multiplexern

Große Multiplexer können durch Kaskadierung von kleinen Multiplexern aufgebaut werden



Was macht mehr Sinn? Kaskadierung oder zweistufige Logik?

Kaskadierung von Multiplexern

Was macht mehr Sinn? Kaskadierung oder zweistufige Logik?

Antwort: Es kommt darauf an, was optimiert werden soll.

Gatterverzögerung: Bei zweistufige Logik entstehen kleinere Gatterlaufzeiten

Chipfläche: Eine Kaskadierung benötigt insgesamt weniger Eingänge und kann daher auf kleinerer Chipfläche realisiert werden.

Beispielrechnung:

Eine zweistufige Logik benötigt UND-Gatter mit $(\log_2 n) + 1$ Eingängen.

Eine kaskadierte Lösung hat mehr UND-Gatter, diese haben jedoch weniger Eingänge.

Beispiel 8:1 Mux: $8 \cdot 4 = 32$ Eingänge im Vergleich zu $2 \cdot 2 \cdot 7 = 28$ Eingänge

Wenn der Flächenbedarf proportional zu den Eingängen angenommen wird, benötigt die kaskadierte Realisierung insgesamt weniger Fläche

Multiplexer als universelle Logik

n -zu-1 Multiplexer können jede Funktion von $k = \log_2 n$ Variablen implementieren

Die Variablen x_{k-1}, \dots, x_0 werden als Steuervariablen s_{k-1}, \dots, s_0 verwendet

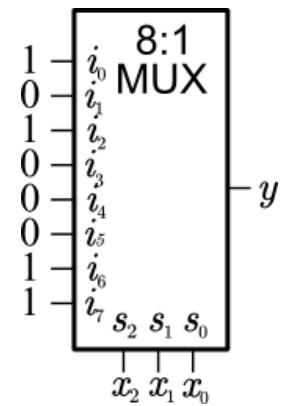
Dateneingänge i_{n-1}, \dots, i_0 werden auf 0 bzw. 1 gelegt

Beispiel:

$$y = m_0 \vee m_2 \vee m_6 \vee m_7$$
$$\bar{x}_2 \bar{x}_1 \bar{x}_0 \vee \bar{x}_2 x_1 \bar{x}_0 \vee x_2 x_1 \bar{x}_0 \vee x_2 x_1 x_0$$

Wahrheitstafel:

	x_2	x_1	x_0	y
0:	0	0	0	1
1:	0	0	1	0
2:	0	1	0	1
3:	0	1	1	0
4:	1	0	0	0
5:	1	0	1	0
6:	1	1	0	1
7:	1	1	1	1



Multiplexer als universelle Logik

Alternative: n -zu-1 Multiplexer können jede Funktion von $k = (\log_2 n) + 1$ Variablen implementieren

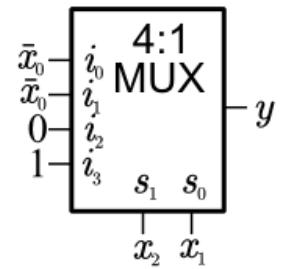
Die Variablen x_{k-1}, \dots, x_1 werden als Steuervariablen s_{k-2}, \dots, s_0 verwendet

Dateneingänge werden auf $x_0, \bar{x}_0, 0$ oder 1 gelegt

Beispiel:

Wahrheitstafel:

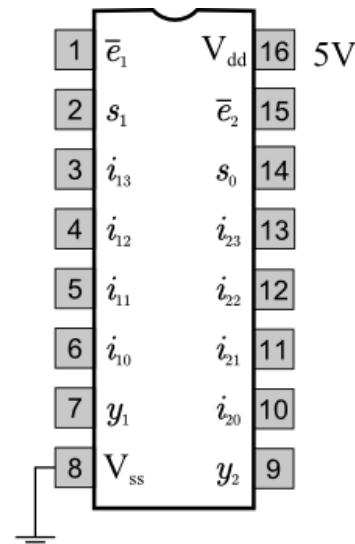
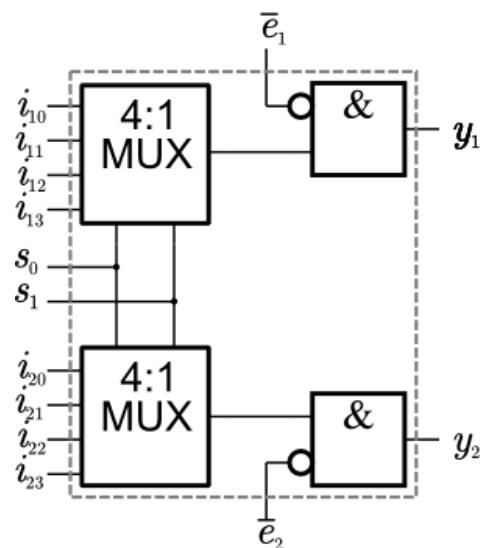
	x_2	x_1	x_0	y	
0:	0	0	0	1	\bar{x}_0
1:	0	0	1	0	
2:	0	1	0	1	\bar{x}_0
3:	0	1	1	0	
4:	1	0	0	0	0
5:	1	0	1	0	
6:	1	1	0	1	1
7:	1	1	1	1	



Praxisbeispiel: Multiplexer als universelle Logik

Im Folgenden soll mit Hilfe des CMOS IC CD74HCT253
eine reale Multiplexer-Schaltung aufgebaut werden

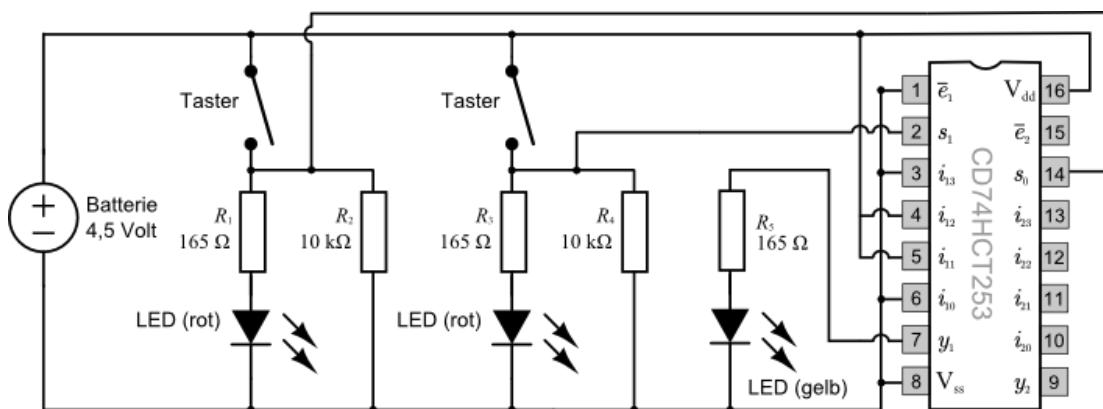
Der 74HCT253 enthält zwei 4-zu-1 Multiplexer



Praxisbeispiel: Multiplexer als universelle Logik

Einer der zwei 4-zu-1 Multiplexer wird derart beschaltet, dass ein XOR realisiert wird

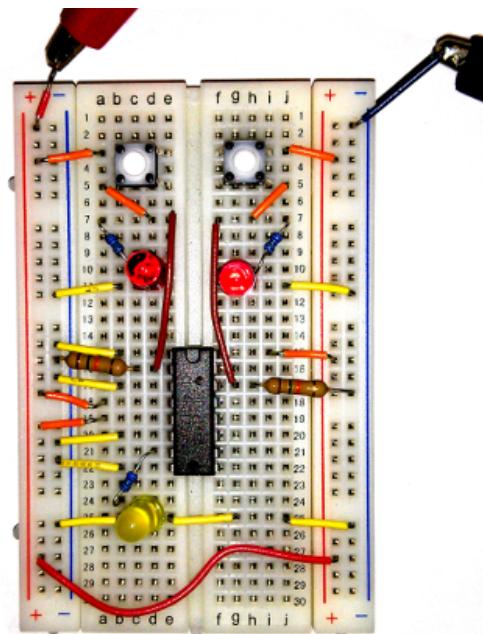
s_1	s_0	y_1
0	0	0
0	1	1
1	0	1
1	1	0



Thorsten Thormählen 15 / 28

Praxisbeispiel: Multiplexer als universelle Logik

Dieses Photo zeigt den Aufbau der Schaltung aus der vorangegangenen Folie auf eine Steckplatine



Thorsten Thormählen 16 / 28

Demultiplexer

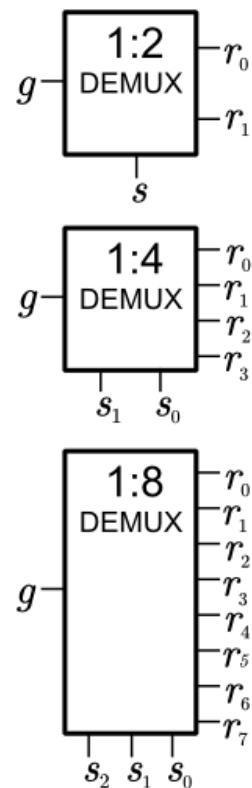
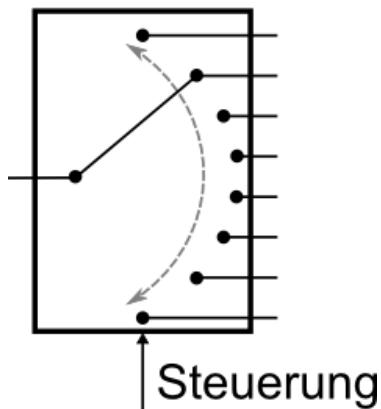
Das inverse Element zum Multiplexer ist der Demultiplexer

1 Eingang, k Steuereingänge, $n = 2^k$ Ausgänge r_{n-1}, \dots, r_0

Der Eingang g wird oft "Enable" genannt

Abhängig von den Steuereingängen wird der Eingang auf einen der Ausgänge geschaltet, die anderen Ausgänge sind 0.

Demultiplexer



Demultiplexer

1-zu-2 Demultiplexer (1:2 Demux):

$$r_0 = g \bar{s}$$

$$r_1 = g s$$

1-zu-4 Demultiplexer (1:4 Demux):

$$r_0 = g \bar{s}_1 \bar{s}_0$$

$$r_1 = g \bar{s}_1 s_0$$

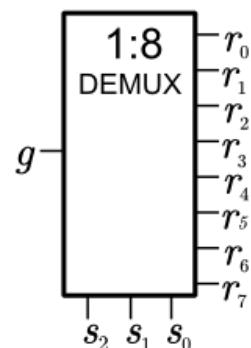
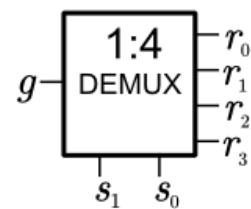
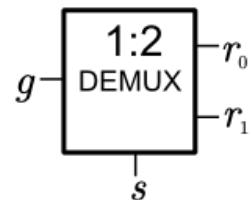
$$r_2 = g s_1 \bar{s}_0$$

$$r_3 = g s_1 s_0$$

1-zu- n Demultiplexer (1: n Demux):

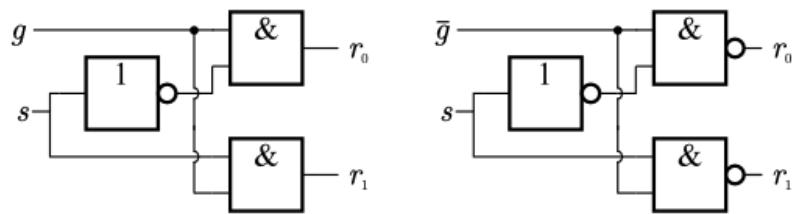
$$r_j = g m_j \quad \forall r_{n-1}, \dots, r_1, r_0$$

wobei m_j den Minterm der k Steuersignale s_{k-1}, \dots, s_1, s_0 bezeichnet und gilt $n = 2^k$

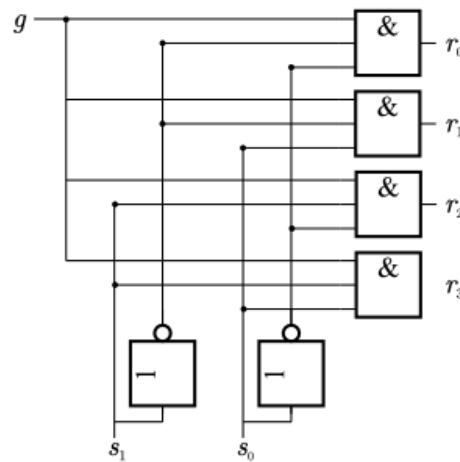


Realisierung von Demultiplexern

Ein 1:2 Demux mit Gattern:

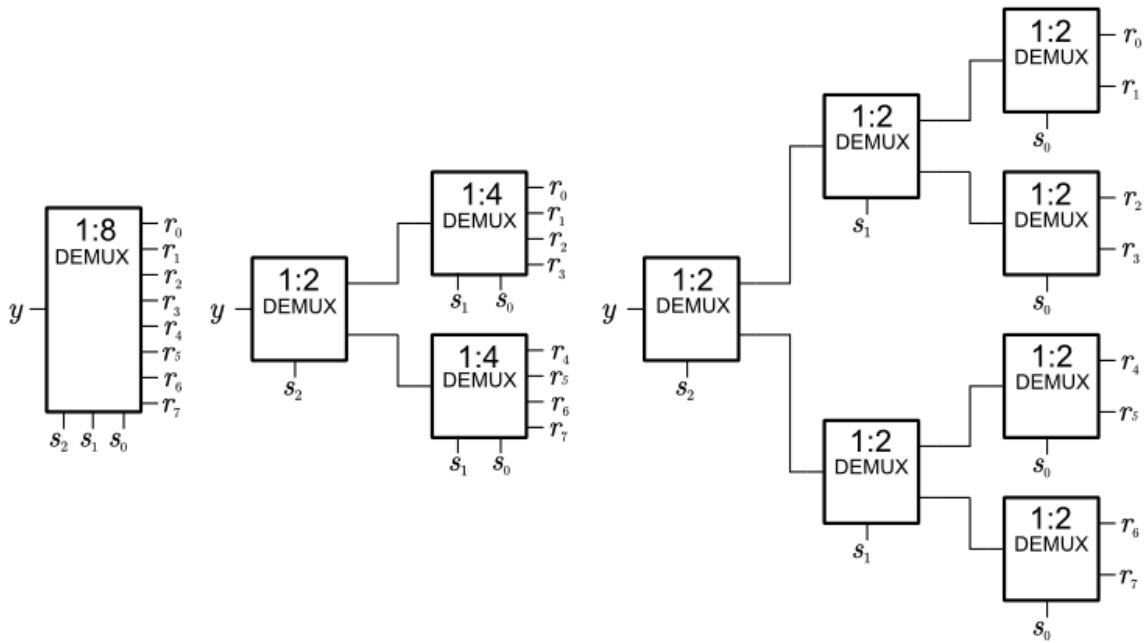


Ein 1:4 Demux mit Gattern:



Kaskadierung von Demultiplexern

Große Demultiplexer können durch Kaskadierung von kleinen Demultiplexern aufgebaut werden



Demultiplexer als universelle Logik

1-zu- n Demultiplexer können jede Funktion von $k = \log_2 n$ Variablen implementieren

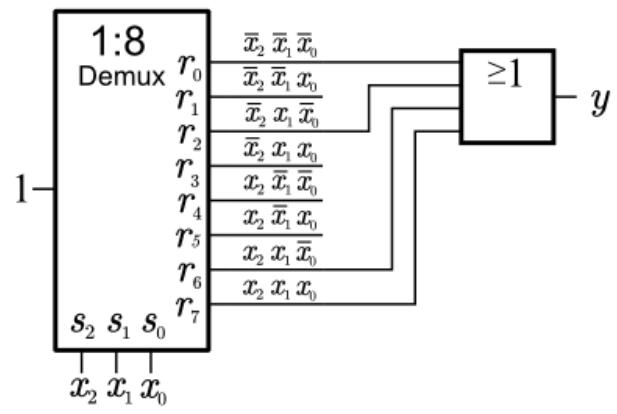
Die Variablen x_{k-1}, \dots, x_0 werden als Steuervariablen s_{k-1}, \dots, s_0 verwendet
Eingang wird auf 1 gelegt

An den Ausgängen liegen die Minterme der Steuervariablen an und können durch ein ODER-Gatter verknüpft werden, um die Funktion zu implementieren

Beispiel: $y = m_0 \vee m_2 \vee m_6 \vee m_7$

Wahrheitstafel:

	x_2	x_1	x_0	y
0:	0	0	0	1
1:	0	0	1	0
2:	0	1	0	1
3:	0	1	1	0
4:	1	0	0	0
5:	1	0	1	0
6:	1	1	0	1
7:	1	1	1	1

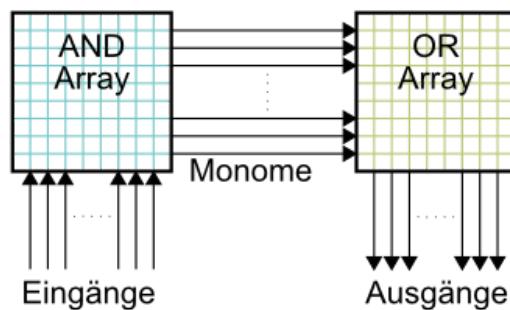


Programmierbare logische Schaltungen

Eine programmierbare logische Schaltung (engl.: Programmable Logic Device, PLD) ist ein vorgefertigter IC mit vielen AND- und OR-Gattern

Die Eingangsvariablen werden auf ein AND-Array geführt das Monome erzeugt

Die Monome werden durch ein OR-Array zu den Ausgangsfunktionen verknüpft



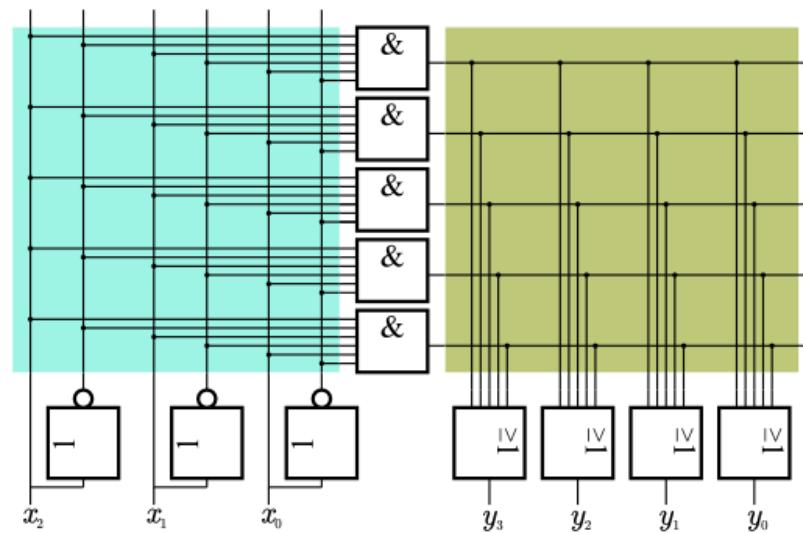
Vorteil: PLD können in großer Stückzahl gefertigt werden und erst später durch das Auf trennen von Verbindungen ihre Funktionalität bekommen ("programmiert werden")

PLA (Programmable Logic Array): AND-Array veränderbar, OR-Array veränderbar

PAL (Programmable Array Logic): AND-Array veränderbar, OR-Array fest

PROM (Programmable Read-Only Memory): AND-Array fest, OR-Array veränderbar

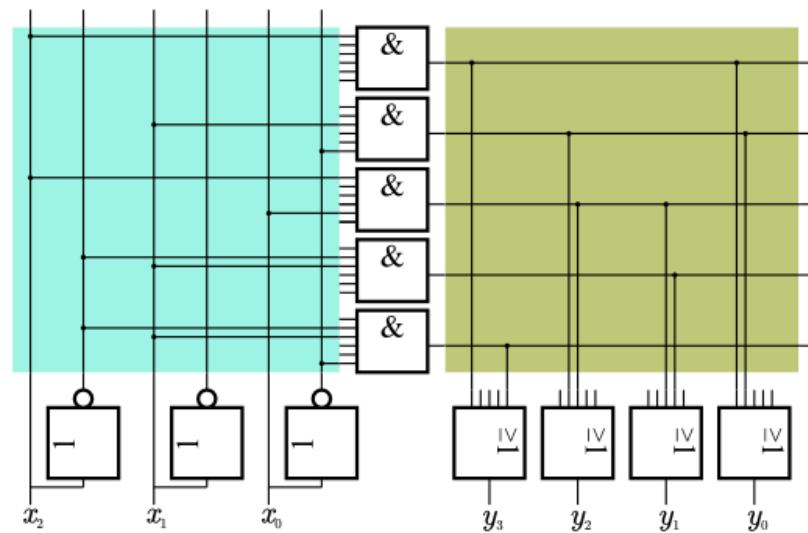
PLA (Programmable Logic Array)



Nicht benötigte Verbindungen werden aufgetrennt.

Realisierte Monome können für verschiedene Ausgänge y_i wieder verwendet werden

PLA (Programmable Logic Array)



Beispiel:

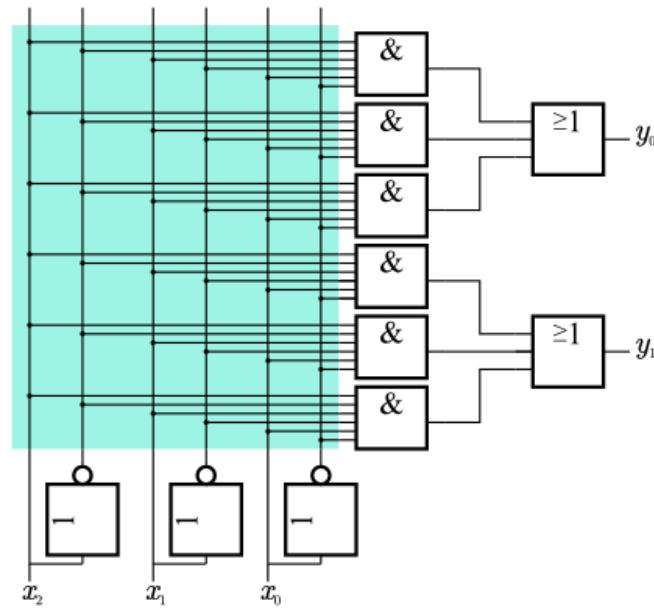
$$y_0 = x_1 \bar{x}_0 \vee x_2$$

$$y_1 = \bar{x}_2 x_1 \vee x_2 x_0$$

$$y_2 = x_1 \bar{x}_0 \vee x_2 x_0$$

$$y_3 = x_2 \vee \bar{x}_2 x_1 \bar{x}_0$$

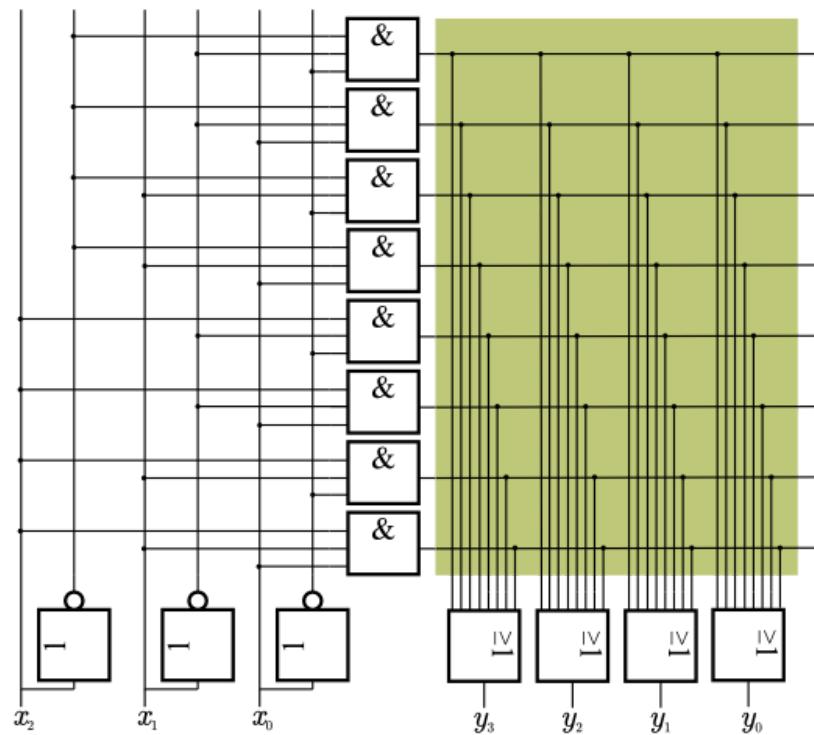
PAL (Programmable Array Logic)



Beim PAL ist nur das AND-Array veränderbar

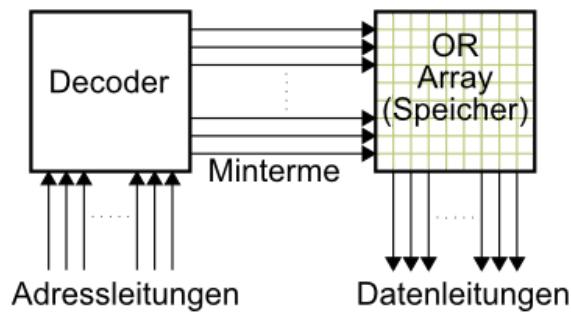
Die ODER-Verknüpfungen sind fest verdrahtet

PROM (Programmable Read-Only Memory)



Die UND-Verknüpfungen sind fest verdrahtet, alle Minterme realisiert

PROM (Programmable Read-Only Memory)



Beim Anlegen einer Adresse an die Eingänge liefert das PROM den Speicherinhalt für diese Adresse an den Ausgängen

Im Unterschied zum PLA hat das PROM ein fest verdrahtetes AND-Array, das alle möglichen Kombinationen der Eingänge (alle Minterme) realisiert

Bei k Adressleitungen entstehen somit 2^k Minterme (auch "Wortleitungen" genannt)

Bei m Ausgängen liegen im ROM-Speicher also 2^k Worte zu m Bits

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[\[Impressum\]](#) , [\[Datenschutz\]](#) , []

Thorsten Thormählen 28 / 28

Technische Informatik

Arithmetik Schaltungen

Thorsten Thormählen
02. Dezember 2021
Teil 7, Kapitel 2

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Arithmetik Schaltungen

Addition

Inkrement

Subtraktion

Multiplikation

Arithmetisch-logische Einheit (ALU)

Addition

Zur Entwicklung einer Schaltung für die Addition kann sich an der schriftlichen Addition orientiert werden:

$$\begin{array}{r} 13 \\ + 5 \\ \hline = 18 \end{array} \qquad \begin{array}{r} 1101 \quad (13) \\ + 0101 \quad (05) \\ \hline = 10010 \quad (18) \end{array}$$

Die Addition wird Bit-weise von rechts nach links durchgeführt (angefangen beim niederwertigsten Bit)

Es kann zu einem Übertrag (Carry) kommen, der beim nächsten Bit berücksichtigt werden muss

Halbaddierer

Beim niederwertigsten Bit müssen nur alle Kombinationen der zwei Summanden a und b betrachtet werden (4 Fälle)

Frage: Wie lautet jeweils das Ergebnis der Addition s und der Übertrag c_{out} (Carry-out)?

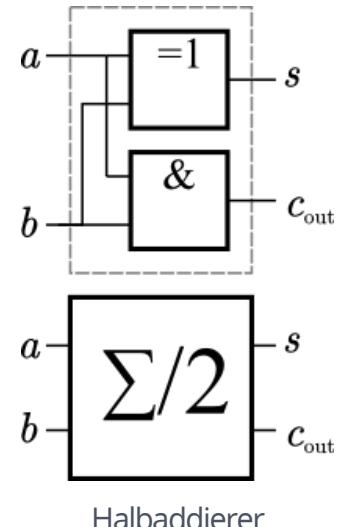
a	b	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Damit ergibt sich:

$$s = \bar{a}\bar{b} \vee a\bar{b} = a \leftrightarrow b$$

$$c_{\text{out}} = ab$$

Die Realisierung mit Logikgattern wird *Halbaddierer* genannt und erhält ein eigenes Symbol (siehe rechts)

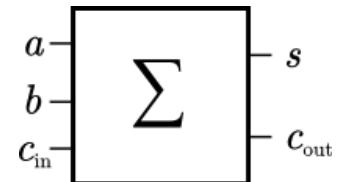


Halbaddierer

Volladdierer

Bei den nachfolgenden Bits muss nun ebenfalls der Übertrag c_{in} der vorangegangenen Bit-weisen Addition berücksichtigt werden (Carry-in)

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Volladdierer

Damit ergibt sich:

$$s = \bar{a}\bar{b}c_{\text{in}} \vee \bar{a}b\bar{c}_{\text{in}} \vee a\bar{b}\bar{c}_{\text{in}} \vee abc_{\text{in}}$$

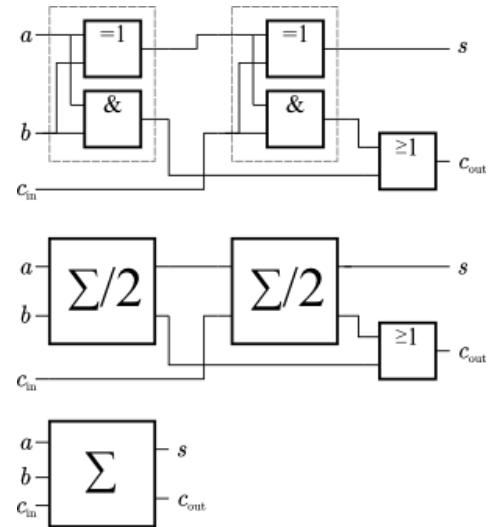
$$c_{\text{out}} = \bar{a}bc_{\text{in}} \vee a\bar{b}c_{\text{in}} \vee ab\bar{c}_{\text{in}} \vee abc_{\text{in}}$$

Volladdierer

Durch Umformung der booleschen Ausdrücke ergibt sich:

$$\begin{aligned}
 s &= \bar{a}\bar{b}c_{\text{in}} \vee \bar{a}b\bar{c}_{\text{in}} \vee a\bar{b}\bar{c}_{\text{in}} \vee abc_{\text{in}} \\
 &\stackrel{(6)}{=} (\bar{a}\bar{b}c_{\text{in}} \vee abc_{\text{in}}) \vee (\bar{a}b\bar{c}_{\text{in}} \vee a\bar{b}\bar{c}_{\text{in}}) \\
 &\stackrel{(8)}{=} ((\bar{a}\bar{b} \vee ab)c_{\text{in}}) \vee ((\bar{a}b \vee a\bar{b})\bar{c}_{\text{in}}) \\
 &= ((a \leftrightarrow b)c_{\text{in}}) \vee ((a \leftrightarrow b)\bar{c}_{\text{in}}) \\
 &= ((\overline{a \leftrightarrow b})c_{\text{in}}) \vee ((a \leftrightarrow b)\bar{c}_{\text{in}}) \\
 &= (a \leftrightarrow b) \leftrightarrow c_{\text{in}}
 \end{aligned}$$

$$\begin{aligned}
 c_{\text{out}} &= (\bar{a}bc_{\text{in}} \vee a\bar{b}c_{\text{in}}) \vee (ab\bar{c}_{\text{in}} \vee abc_{\text{in}}) \\
 &\stackrel{(8)}{=} ((\bar{a}b \vee a\bar{b})c_{\text{in}}) \vee (ab(\bar{c}_{\text{in}} \vee c_{\text{in}})) \\
 &\stackrel{(5)}{=} (a \leftrightarrow b)c_{\text{in}} \vee ab
 \end{aligned}$$



Volladdierer

Carry-Ripple-Addierer

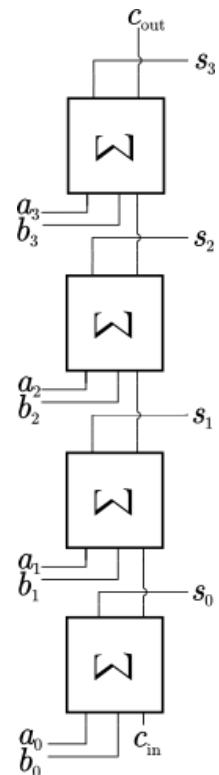
Ein Addierwerk für Binärzahlen der Stelligkeit n kann mit n Volladdierern realisiert werden

Jeder Ein-Bit-Addierer ist für eine Ziffer s_i der Summe verantwortlich

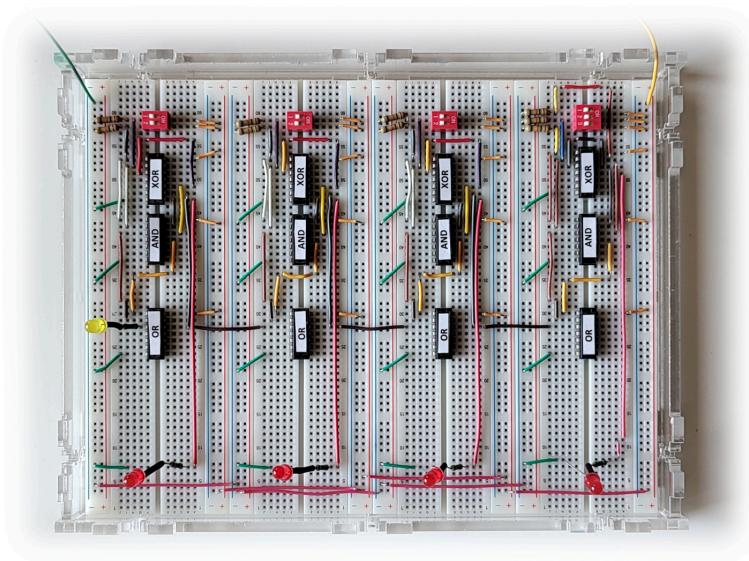
Der c_{out} -Ausgang wird jeweils mit dem c_{in} des nächsten Ein-Bit-Addierers verbunden

Die Laufzeit ist linear in der Anzahl der Stellen, da jeder Ein-Bit-Addierer zunächst auf den Übertrag seines Vorgängers warten muss (d.h. ein 8-Bit-Addierwerk braucht doppelt so lange, wie ein 4-Bit-Addierwerk)

Die Behandlung des Übertrags (Carry) limitiert die Laufzeit. Das Carry "rieselt" (engl. "ripple") langsam durch die Schaltung.

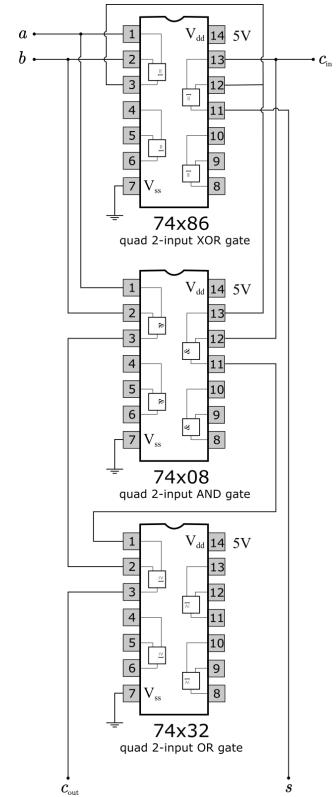


Aufbau eines Carry-Ripple-Addierers aus Standard-ICs



Im Wintersemester 2024/25 hat Célestine Schönau einen 4-Bit-Carry-Ripple-Addierer aus Standard-ICs der 74xx-Familie aufgebaut

Die Schaltung kann in der Vorlesung ausprobiert werden



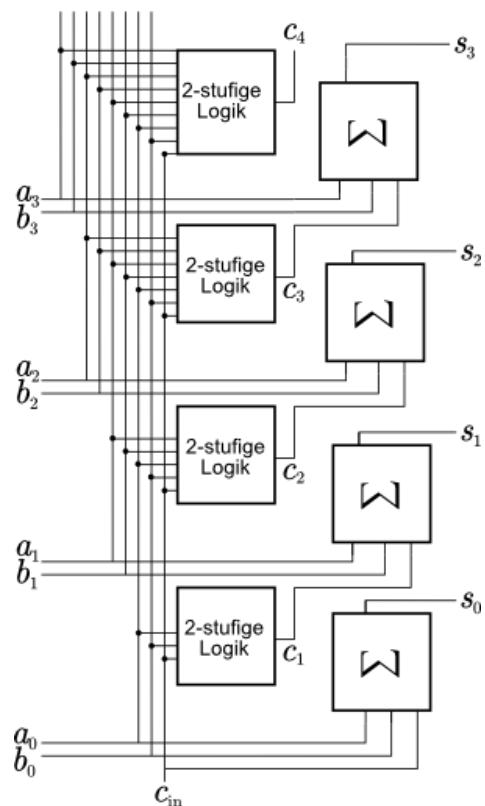
Carry-Look-Ahead-Addierer

Die Idee des Carry-Look-Ahead-Addierers ist, die Carry-Bits für alle Ein-Bit-Addierer durch eine separate Logikschaltung zu erzeugen

Wie in den vorangegangenen Kapiteln gezeigt wurde, kann jede boolesche Funktion als zweistufige Logik realisiert werden

Beim Carry-Look-Ahead-Addierer können alle Ein-Bit-Addierer parallel arbeiten und liefern zeitgleich ihr Ergebnis

Allerdings steigt mit jeder Ziffer der Aufwand für die Realisierung der Übertragsberechnung mittels zweistufiger Logik, da mit jeder Ziffer zwei Eingänge hinzukommen



Thorsten Thormählen 11 / 27

Carry-Look-Ahead-Addierer

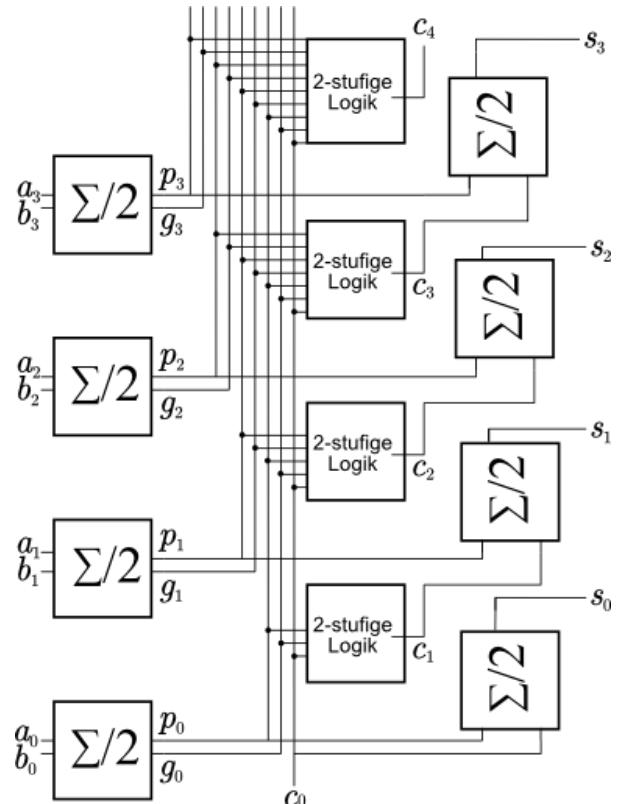
Um die Anzahl der Gatter innerhalb der 2-stufigen Logik zu reduzieren, wird der boolesche Ausdruck zur Berechnung des Übertrags beim Volladdierer betrachtet

Bei gegebenem Übertrag c_i des vorherigen Ein-Bit-Addierers berechnet sich c_{i+1}

$$c_{i+1} = \underbrace{(a_i \leftrightarrow b_i)}_{p_i} c_i \vee \underbrace{a_i b_i}_{g_i}$$

Dabei sind p_i und g_i gerade die Ausgänge eines entsprechenden Halbaddierers

Um diese Werte bei der Berechnung des Übertrags abzugreifen, werden die Volladdierer in zwei Halbaddierer aufgespaltet



Thorsten Thormählen 12 / 27

Carry-Look-Ahead-Addierer

Durch rekursive Anwendung des Ausdrucks zur Berechnung des Übertrags

$$c_{i+1} = \underbrace{(a_i \leftrightarrow b_i)}_{p_i} c_i \vee \underbrace{a_i b_i}_{g_i}$$

kann die benötigte 2-stufige Logik ausgerechnet werden:

$$\begin{aligned} c_1 &= p_0 c_0 \vee g_0 \\ c_2 &= p_1 c_1 \vee g_1 \\ &= p_1 (p_0 c_0 \vee g_0) \vee g_1 \\ &= p_1 p_0 c_0 \vee p_1 g_0 \vee g_1 \\ c_3 &= p_2 c_2 \vee g_2 \\ &= p_2 (p_1 p_0 c_0 \vee p_1 g_0 \vee g_1) \vee g_2 \\ &= p_2 p_1 p_0 c_0 \vee p_2 p_1 g_0 \vee p_2 g_1 \vee g_2 \\ c_4 &= p_3 c_3 \vee g_3 \\ &= p_3 (p_2 p_1 p_0 c_0 \vee p_2 p_1 g_0 \vee p_2 g_1 \vee g_2) \vee g_3 \\ &= p_3 p_2 p_1 p_0 c_0 \vee p_3 p_2 p_1 g_0 \vee p_3 p_2 g_1 \vee p_3 g_2 \vee g_3 \end{aligned}$$

Inkrement

Das Inkrementieren einer Zahl ($a=a+1$ bzw. kurz $a++$) ist eine sehr häufige arithmetische Operation

Z.B. muss nach jeder Befehlsausführung der Programmzähler erhöht werden, oder, beim sequentiellen Lesen aus dem Speicher, der Adresszähler

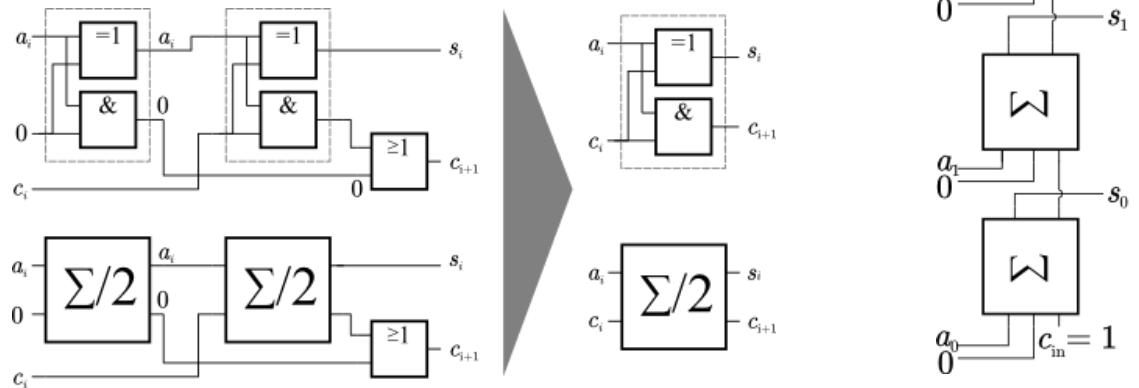
Ein weiteres Beispiel sind Schleifen, bei denen das Inkrementieren der Zählvariablen um 1 eine typische Operation ist:

```
for(int a = 0; a < 10; a++) {  
    doSomething(a);  
}
```


Inkrement

Ein n -stelliger Inkrementierer kann mit einem n -stelligen Addiererwerk realisiert werden, indem alle b_i gleich 0 und der Übertrag am Eingang des niedrigwertigsten Bits c_{in} gleich 1 gesetzt wird

Die Abbildung rechts zeigt dies für den Carry-Ripple-Addierer
Durch feste Verdrahtung einiger Variablen kann die Schaltung vereinfacht werden, z.B. können beim Carry-Ripple-Addierer die Volladdierer durch entsprechend beschaltete Halbaddierer ersetzt werden



Subtraktion

Die Subtraktion zweier Zahlen $y = a - b$ kann auf die Addition zurückgeführt werden $y = a + (-b)$

Dazu werden die Zahlen im Zweierkomplement dargestellt

Zur Wiederholung: Beim Zweierkomplement werden die negativen Zahlen durch das bitweise Komplement und einer zusätzlichen Addition von 1 gebildet

Beispiel:

$$(-6)_{10} \hat{=} 1001 + 1 = 1010$$

Subtraktion mit dem Zweierkomplement:

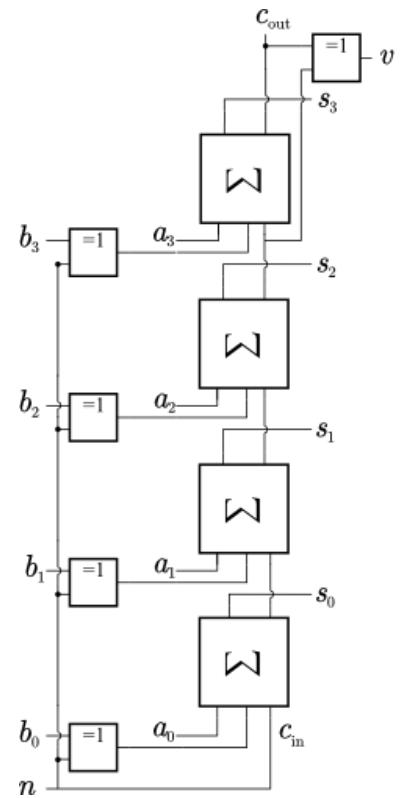
$$\begin{array}{r} 5 \\ + -6 \\ \hline = -1 \end{array} \qquad \begin{array}{r} 0101 \quad (5) \\ + 1010 \quad (-6) \\ \hline = 1111 \quad (-1) \end{array}$$

Subtraktion

Ein n -stelliger Subtrahierer kann mit einem n -stelligen Addiererwerk realisiert werden, indem eine zusätzliche Logik zu Berechnung des Zweierkomplements hinzugefügt wird

Die Abbildung rechts zeigt dies am Beispiel des Carry-Ripple-Addierers

Der Eingang n wählt aus, ob b negiert werden soll, und der Ausgang v zeigt an, ob ein Überlauf (Overflow) aufgetreten ist



Thorsten Thormählen 17 / 27

Multiplikation

Zur Entwicklung einer Schaltung für die Multiplikation wird zunächst die schriftliche Multiplikation (wie aus der Schule bekannt) betrachtet:

$$\begin{array}{r} 25 \cdot 13 \\ \hline 75 \\ +25 \\ \hline = 325 \end{array}$$

$$\begin{array}{r} 11001 \cdot 1101 \\ \hline 11001 \\ 00000 \\ 11001 \\ +11001 \\ \hline = 101000101 \end{array}$$

Es ist zu erkennen, dass die Multiplikation von Binärzahlen durch eine Verschiebung und Addition realisiert werden kann

Matrixmultiplizierer

Eine Schaltung, die das schriftliche Multiplizieren nachbildet, ist der Matrixmultiplizierer

Gegeben sei Multiplikator $a = a_{n-1} \dots a_1 a_0$ und Multiplikand $b = b_{n-1} \dots b_1 b_0$ mit Stelligkeit n

Die Anordnung der Bits beim schriftlichen Multiplizieren kann als $n \times (2n - 1)$ Matrix dargestellt werden

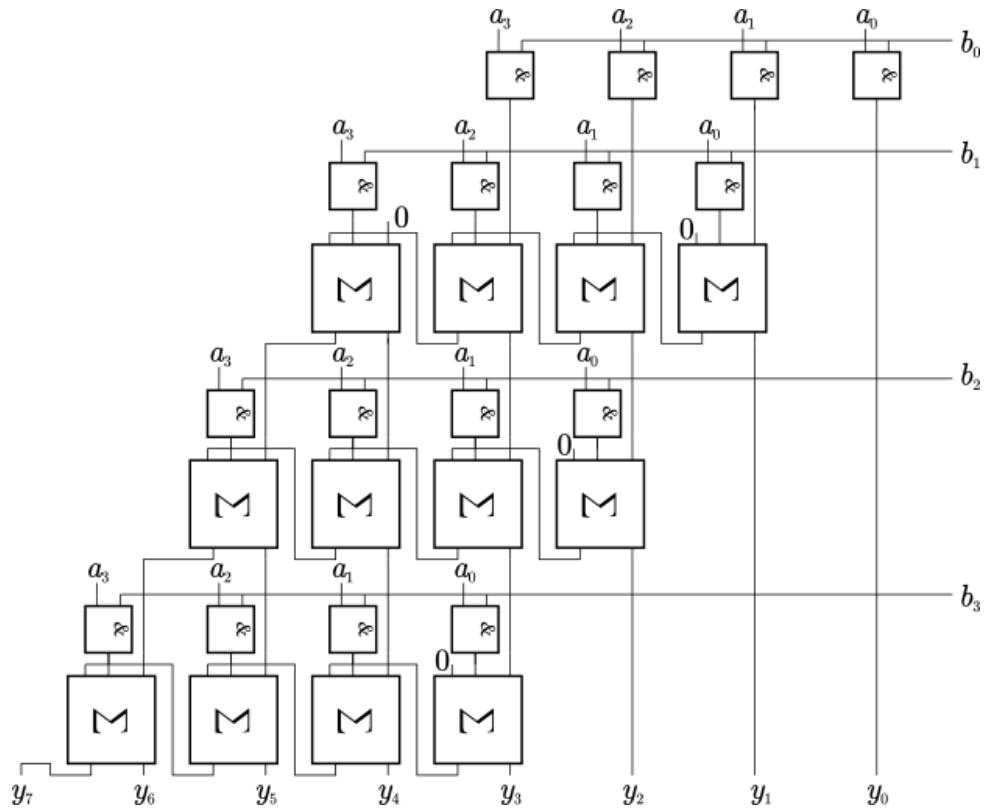
$$\begin{bmatrix} 0 & \dots & 0 & a_{n-1}b_0 & a_{n-2}b_0 & \dots & a_2b_0 & a_1b_0 & a_0b_0 \\ 0 & \dots & a_{n-1}b_1 & a_{n-2}b_1 & a_{n-3}b_1 & \dots & a_1b_1 & a_0b_1 & 0 \\ \vdots & \vdots \\ a_{n-1}b_{n-1} & \dots & a_1b_{n-1} & a_0b_{n-1} & 0 & \dots & 0 & 0 & 0 \end{bmatrix}$$

Diese Matrixstruktur findet sich auch in der entsprechenden Logikschaltung wieder

Die Elemente der Matrix werden durch UND-Verknüpfungen $a_j \wedge b_i$ berechnet

Jede Spaltensumme durch ein Addierwerk (benötigt maximal $n - 1$ Volladdierer)

4-Bit-Matrixmultiplizierer



Thorsten Thormählen 20 / 27

Multiplizierer

Die vorgestellte Realisierung eines Multiplizierers hat das gleiche Problem, wie der Carry-Ripple-Addierer: Das Warten auf den Übertrag limitiert die Laufzeit

Analog, wie beim Addierer, können Schaltungen entworfen werden, die mehr Logikgatter für die Berechnung des Übertrags bereitstellen und damit die Laufzeit reduzieren

Arithmetisch-logische Einheit (ALU)

Die arithmetisch-logische Einheit (engl. arithmetic logic unit, ALU) dient zur Realisierung der Elementaroperationen eines Rechners

Dazu gehören

Arithmetische Operationen: Addition, Subtraktion, Multiplikation, ...

Logische Operationen: Bitweise UND-Verknüpfung, ODER-Verknüpfung, Prüfen auf Gleichheit,

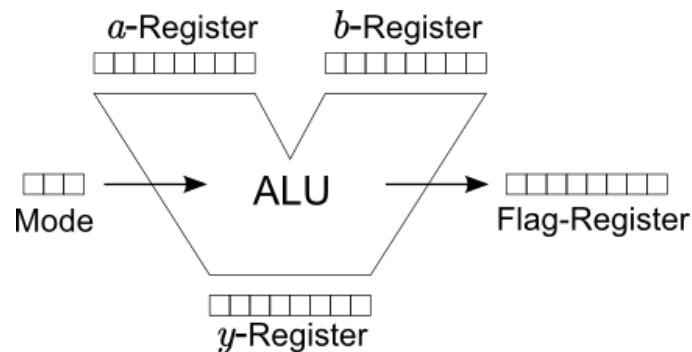
Arithmetisch-logische Einheit (ALU)

In einer ALU werden zwei Eingabewerte a und b zu einem Ergebniswert y verknüpft
Diese Werte stehen in Registern zur Verfügung (Registerbreite kann 8, 16, 32, 64 Bits sein)

Der "Mode" selektiert, welche Funktion die ALU ausführen soll

Im "Flag" wird der Status der ALU angezeigt, z.B. wenn ein Überlauf stattgefunden hat

Zur schematischen Darstellung hat sich folgendes Symbol etabliert:



Arithmetisch-logische Einheit (ALU)

Flag-Register (Statusregister)

Übertrag c (Carry flag): Übertrag ist aufgetreten

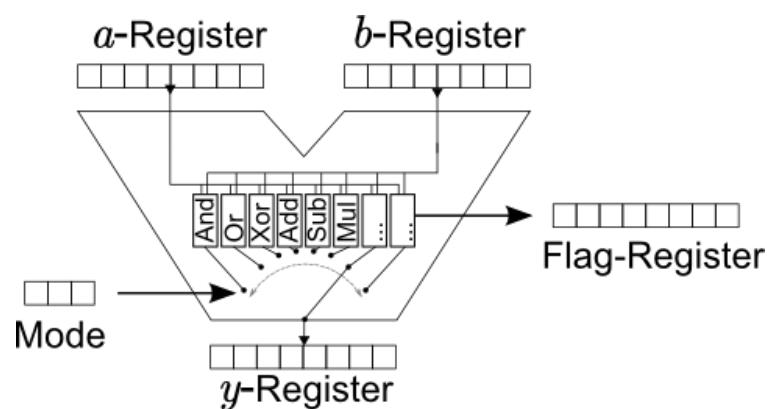
Überlauf v (Overflow flag): Ergebnis passt nicht ins y -Register

Null z (Zero flag): Ergebnis ist Null

Negativ n (Negation flag): Ergebnis ist negativ

...

Der Mode-Eingang wählt die auszuführende Operation aus



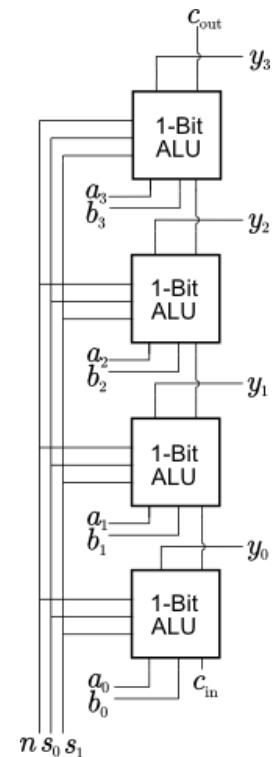
Beispiel für eine ALU

Zur Realisierung einer einfachen ALU soll diese aus mehreren 1-Bit-ALUs zusammengesetzt werden

Wie bereits von den Addierwerken bekannt, muss dabei die Carry-Information an den Nachfolger weitergegeben werden

Insgesamt soll die hier gezeigte beispielhafte ALU acht verschiedene Operationen ausführen:

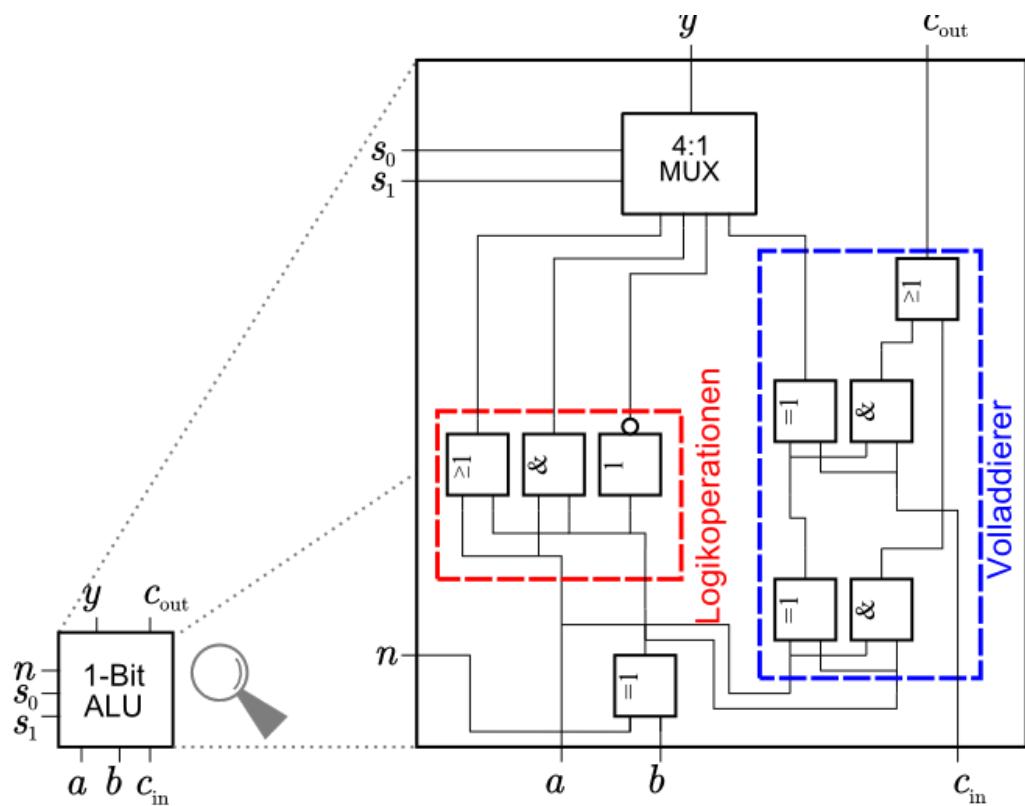
s_1	s_0	n	y
0	0	0	$a \vee b$
0	0	1	$a \vee \bar{b}$
0	1	0	$a \wedge b$
0	1	1	$a \wedge \bar{b}$
1	0	0	\bar{b}
1	0	1	b
1	1	0	$a + b$
1	1	1	$a + \bar{b}$



4-Bit-ALU

Thorsten Thormählen 25 / 27

1-Bit ALU



Thorsten Thormählen 26 / 27

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[\[Impressum\]](#) , [\[Datenschutz\]](#) , []

Thorsten Thormählen 27 / 27

Technische Informatik

Sequentielle Schaltungen

Thorsten Thormählen
07. Dezember 2021
Teil 7, Kapitel 3

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Sequentielle Schaltungen

Flipflops

Register

Schieberegister

Zähler

Sequentielle Schaltungen

Ohne Rückkopplung (Schaltnetz)

Werte an den Ausgängen sind nur abhängig von den Eingängen

Solche Schaltungen verhalten sich immer gleich (sind zustandslos) und sind durch ihre Schaltfunktion eindeutig beschrieben

Es ist jedoch nicht, möglich etwas zu speichern

Mit Rückkopplung (Schaltwerk)

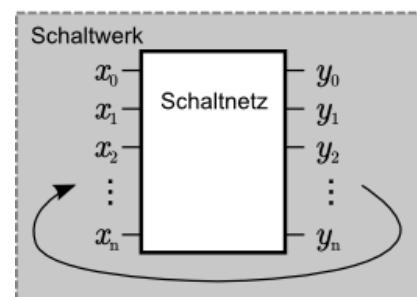
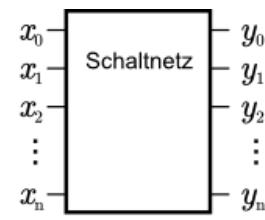
Werte an den Ausgängen sind abhängig von den Eingängen und den vorherigen Ausgangswerten

Das Zeitverhalten muss genau betrachtet werden

Die vorherigen Ausgangswerte können als Zustand der Gatter interpretiert werden.

Abhängig vom Zustand verhalten sich die Gatter anders (zustandsabhängige Schaltfunktion)

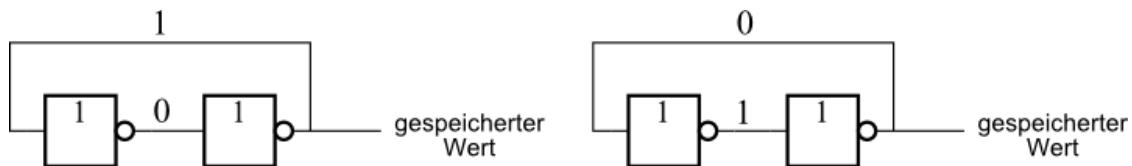
Es wird möglich, Zustände zu speichern



Speicher mit rückgekoppelten Gattern

Zwei Inverter bilden eine statische Speicherzelle

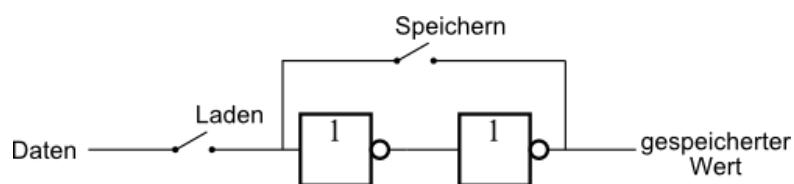
Der Wert bleibt erhalten bis die Versorgungsspannung abgeschaltet wird



Wie kann ein neuer Wert gespeichert werden?

Öffnen des Rückkopplungspfads

Laden des neuen Datenwertes

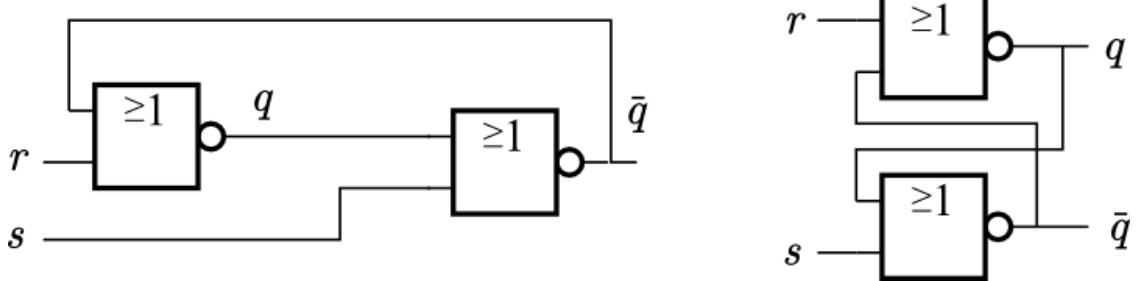


Speicher mit rückgekoppelten Gattern

Rückgekoppelte NOR-Gatter

Bei $r=0$ und $s=0$ bleibt der aktuelle Wert gespeichert (entspricht der Inverter-Schaltung auf der vorherigen Folie)

Der gespeicherte Wert q kann mit Reset $r=1$ auf $q=0$ und mit Set $s=1$ auf $q=1$ gesetzt werden

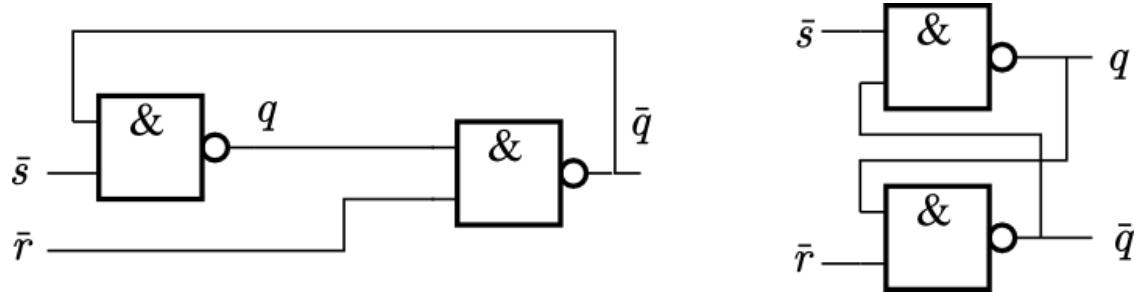


Speicher mit rückgekoppelten Gattern

Rückgekoppelte NAND-Gatter

Bei $\bar{r}=1$ und $\bar{s}=1$ bleibt der aktuelle Wert gespeichert

Der gespeicherte Wert q kann mit Reset $\bar{r}=0$ auf $q=0$ und mit Set $\bar{s}=0$ auf $q=1$ gesetzt werden

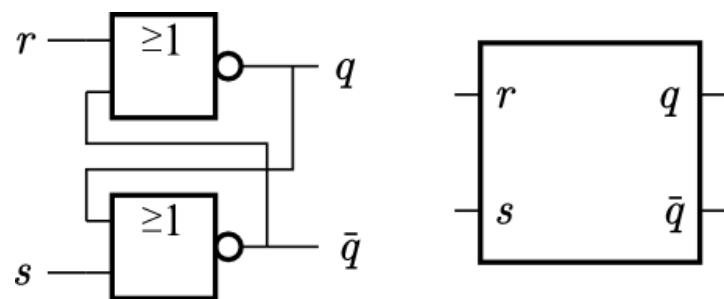


Asynchrones RS-Flipflop

Diese Schaltung wird asynchrones RS-Flipflop genannt

Das RS-Flipflop kann 1 Bit speichern

Für ein asynchrones RS-Flipflop wird auch folgendes Ersatzschaltbild verwendet

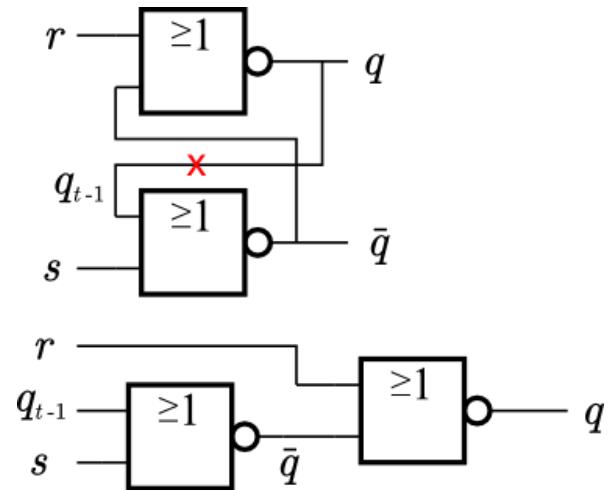


Asynchrones RS-Flipflop

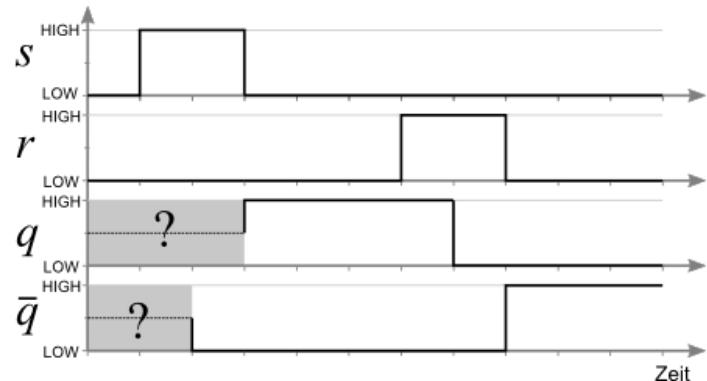
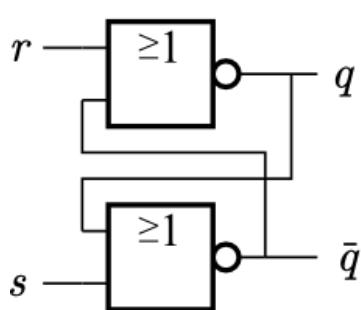
Wird der vorherige Ausgangswert von q mit q_{t-1} bezeichnet, kann folgende Wahrheitstafel aufgestellt werden

r	s	q_{t-1}	q	Funktion
0	0	0	0	halten
0	0	1	1	halten
0	1	0	1	setzen
0	1	1	1	setzen
1	0	0	0	rücksetzen
1	0	1	0	rücksetzen
1	1	0	x	illegal
1	1	1	x	illegal

D.h. der Zustand des vorherigen Ausgangswertes q_{t-1} hat bei der Schaltung keinen Einfluss auf die Funktion



Zeitverhalten eines asynchronen RS-Flipflops



Die Eingangsbelegung $r=1$ und $s=1$ sollte vermieden werden. Beim gleichzeitigen Wechsel auf 0 fängt das System sonst an zu schwingen (siehe [Simulation in Amilosim](#))

Zustandsdiagramm eines asynchronen RS-Flipflops

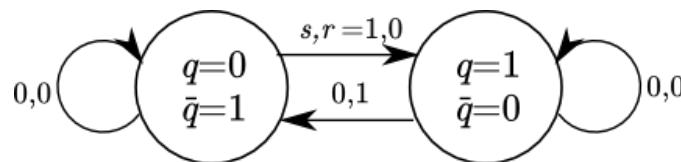
Wird ein RS-Flipflop ohne die illegale Eingangsbelegung $r=1$ und $s=1$ betrieben, gibt es zwei Zustände in dem sich das System befinden kann:

Zustand 1: $q=0$ und $\bar{q}=1$

Zustand 2: $q=1$ und $\bar{q}=0$

In dem Zustandsdiagramm (siehe unten) sind die Zustände mit Kreisen markiert

Abhängig von den Eingangsvariablen r und s können Zustandswechsel auftreten (dargestellt durch Pfeile im Zustandsdiagramm)



Das Verhalten des Flipflops an den Ausgängen ist nicht abhängig vom Zustand

Zustandsdiagramm eines asynchronen RS-Flipflops

Wird ein RS-Flipflop mit der illegalen Eingangsbelegung $r=1$ und $s=1$ betrieben, gibt es vier Zustände in dem sich das System befinden kann:

Zustand 1: $q=0$ und $\bar{q}=1$

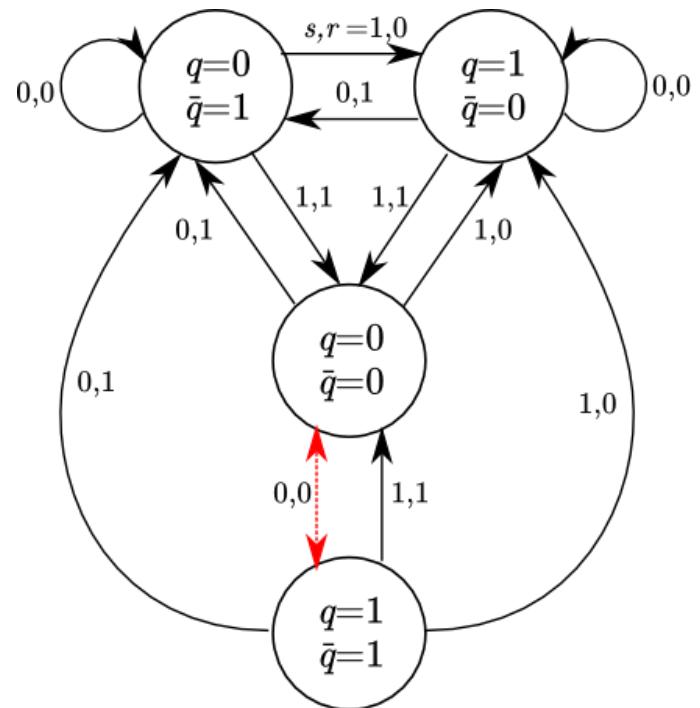
Zustand 2: $q=1$ und $\bar{q}=0$

Zustand 3: $q=0$ und $\bar{q}=0$

Zustand 4: $q=1$ und $\bar{q}=1$

Das Verhalten des Flipflops an den Ausgängen ist jetzt abhängig vom Zustand

Beim gleichzeitigen Wechsel auf $r=0, s=0$ im Zustand $q=0, \bar{q}=0$ fängt das System an zu schwingen

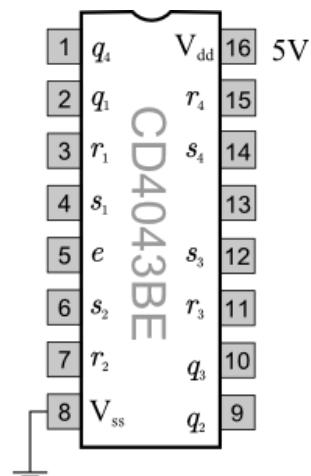
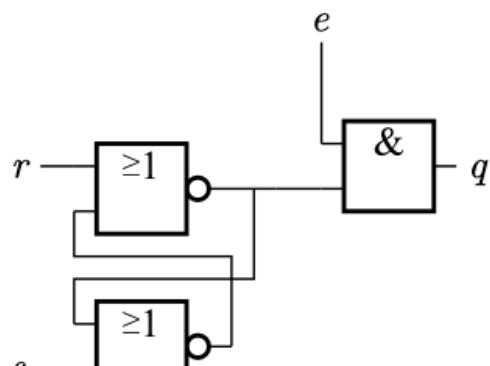


Praxisbeispiel: Flipflop-Schaltung

Nun soll mit Hilfe des CMOS-IC CD4043BE eine reale Flipflop-Schaltung aufgebaut werden

Der CD4043BE enthält 4 asynchrone RS-Flipflops und kann somit 4 Bits speichern

Der Enable-Pin e muss auf 1 gesetzt werden, damit die Ausgänge aktiv sind



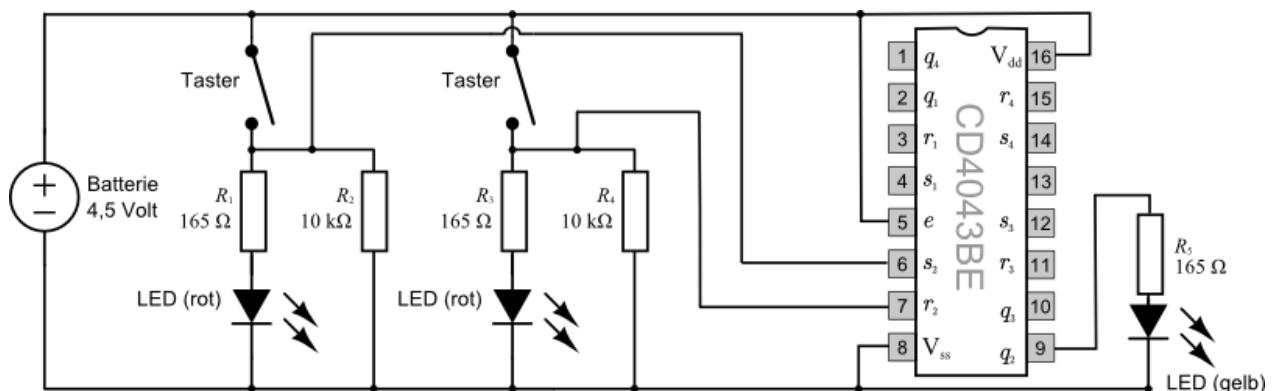
Praxisbeispiel: Flipflop-Schaltung

Die Pull-Down Widerstände sorgen dafür, dass bei geöffneten Tastern $r_2=0$ und $s_2=0$ ist, d.h. das Flipflop hält seinen Ausgangswert q_2

Werden die Taster gedrückt, gilt $s_2=1$ bzw. $r_2=1$.

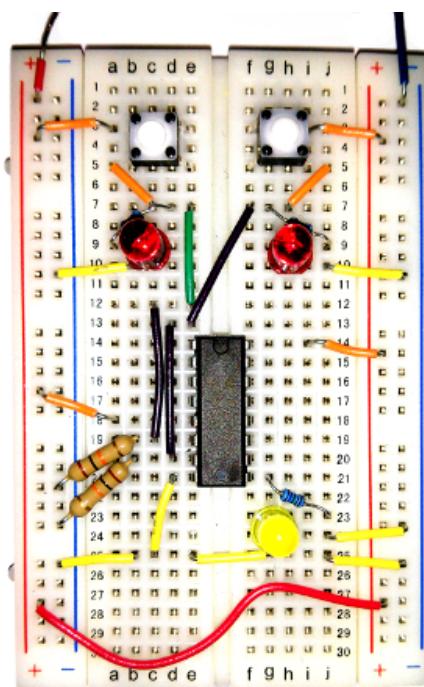
Bei $s_2=1$ und $r_2=0$ wird die LED am Ausgang q_2 eingeschaltet

Bei $s_2=0$ und $r_2=1$ wird die LED am Ausgang q_2 ausgeschaltet



Praxisbeispiel: Flipflop-Schaltung

Dieses Photo zeigt den Aufbau der Schaltung aus der vorangegangenen Folie auf eine Steckplatine

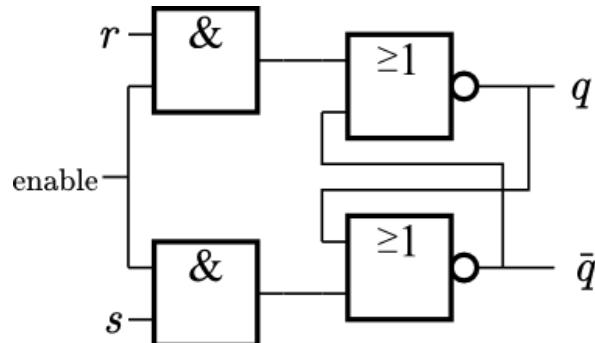


Thorsten Thormählen 16 / 42

Gesteuertes RS-Flipflops

Beim gesteuerten RS-Flipflop legt der "Enable"-Eingang fest, ob r und s eine Rolle spielen

Änderungen am Eingang werden nur noch für enable=1 übernommen



Diese Maßnahme schützt das Flipflop vor eventuellen Störungen am Eingang solange enable=0

Das Problem, dass die Schaltung in Schwingung geraten kann, löst die Maßnahme jedoch nicht

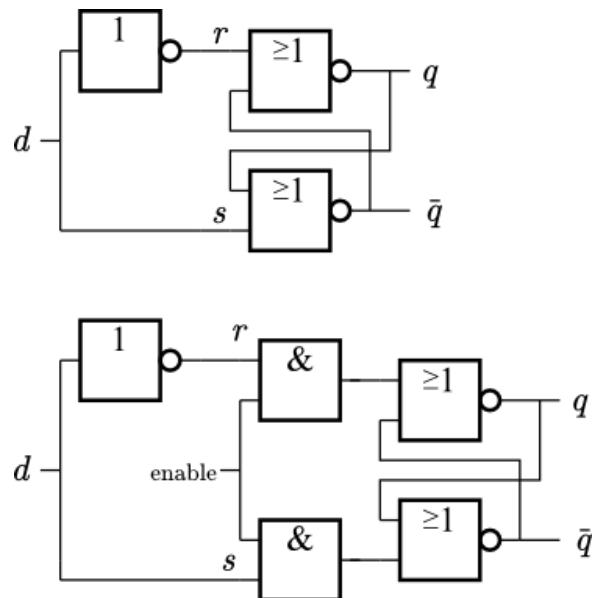
Gesteuertes D-Flipflop

Die einfachste Lösung, die illegale Eingangsbelegung $r=1$ und $s=1$ zu verhindern ist, nur ein Eingangssignal d zu verwenden (rechts oben)

Allerdings kann die Eingangsbelegung $s = 0$ und $r = 0$, die das Speichern des Wertes bewirkt, nicht mehr erzeugt werden

Wird die Schaltung mit der "Enable"-Schaltung kombiniert (rechts unten), ist Speichern wieder möglich (enable=0).

Diese Schaltung wird auch D-Flipflop genannt



Synchrone Flipflops

Durch einen Takt (engl. Clock) kann eine synchrone Schaltung entworfen werden

Der Takt gibt dem System eine gemeinsame Zeitbasis

Zweck bei Flipflops:

Solange warten, bis die Eingänge r und s stabil sind

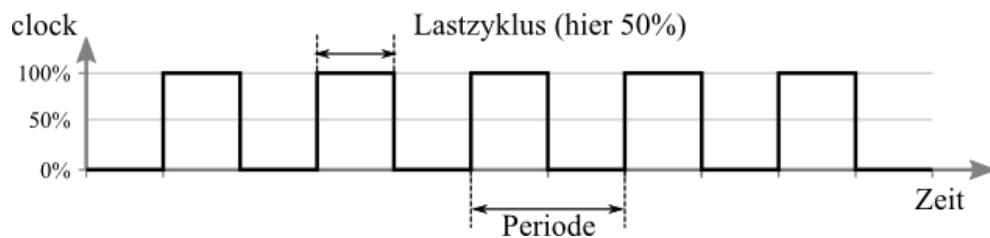
Dann die beabsichtigten Änderungen zulassen (enable=1)

Dies kann erreicht werden, indem das Taktsignal clock auf den enable-Eingang gelegt wird

Taktsignale sind periodische Signale

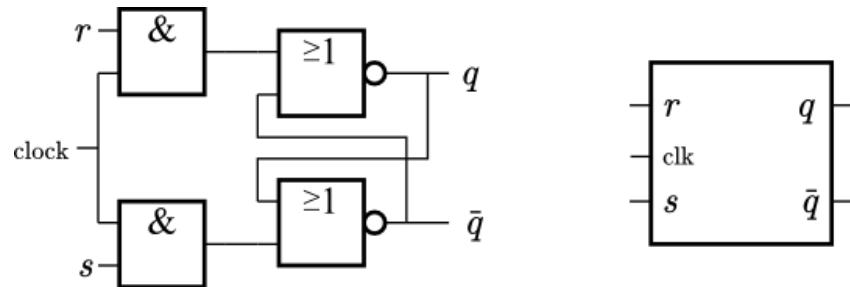
Periodendauer: Zeit zwischen zwei gleichen Flanken

Lastzyklus: hier 50%, da Zeit für 1 gleich lang, wie für 0



Taktzustandgesteuerte Flipflops

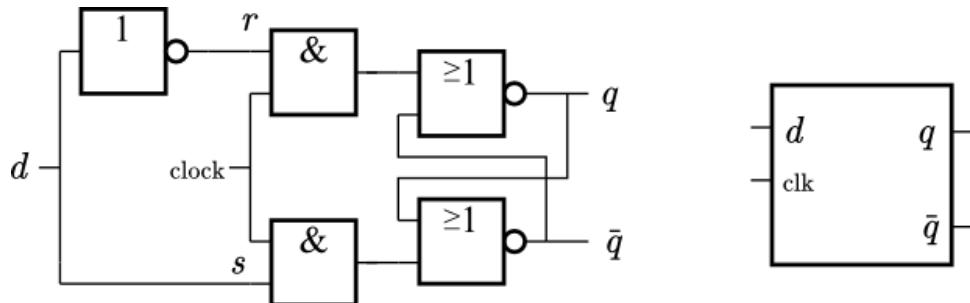
Synchrones RS-Flipflop ([Öffnen in Amilosim](#))



Änderungen am Eingang werden nur während des Lastzyklus übernommen (d.h. wenn $\text{clock}=1$)

Taktzustandgesteuerte Flipflops

Synchrones D-Flipflop ([Öffnen in Amilosim](#))



Änderungen am Eingang werden nur während des Lastzyklus übernommen (d.h. wenn $clock=1$)

Ist während des Lastzyklus $d=1$, wird $q=1$ gesetzt und für den Rest der Periode gehalten

Erst beim nächsten Lastzyklus kann sich q wieder ändern

Das D-Flipflop realisiert damit eine Verzögerung (engl. "Delay"), daher der Name "D"-Flipflop

Taktflankengesteuerte Flipflops

Bisher waren die Flipflops taktzustandsgesteuert

Häufig sinnvoller: taktflankengesteuerte Flipflops

Positiv taktflankengesteuert

Eingänge werden bei der steigenden Flanke abgetastet

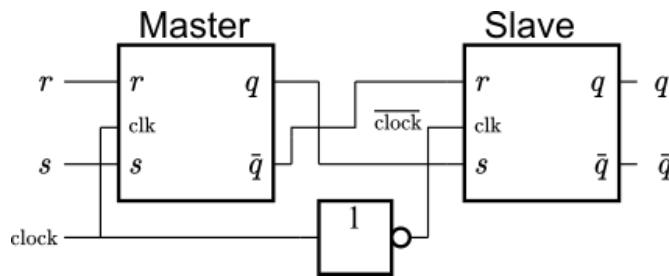
Ausgänge ändern sich nach der steigenden Flanke

Negativ taktflankengesteuert

Eingänge werden bei der fallenden Flanke abgetastet

Ausgänge ändern sich nach der fallenden Flanke

Taktflankengesteuertes Flipflop: Master-Slave Flipflop



Durch hintereinander Schalten von zwei taktzustandsgesteuerten RS-Flipflops entsteht ein taktflankengesteuertes Flipflop, auch "Master-Slave Flipflop" genannt:

r und s werden bei steigender Flanke des $clock$ -Signals vom Master eingelesen

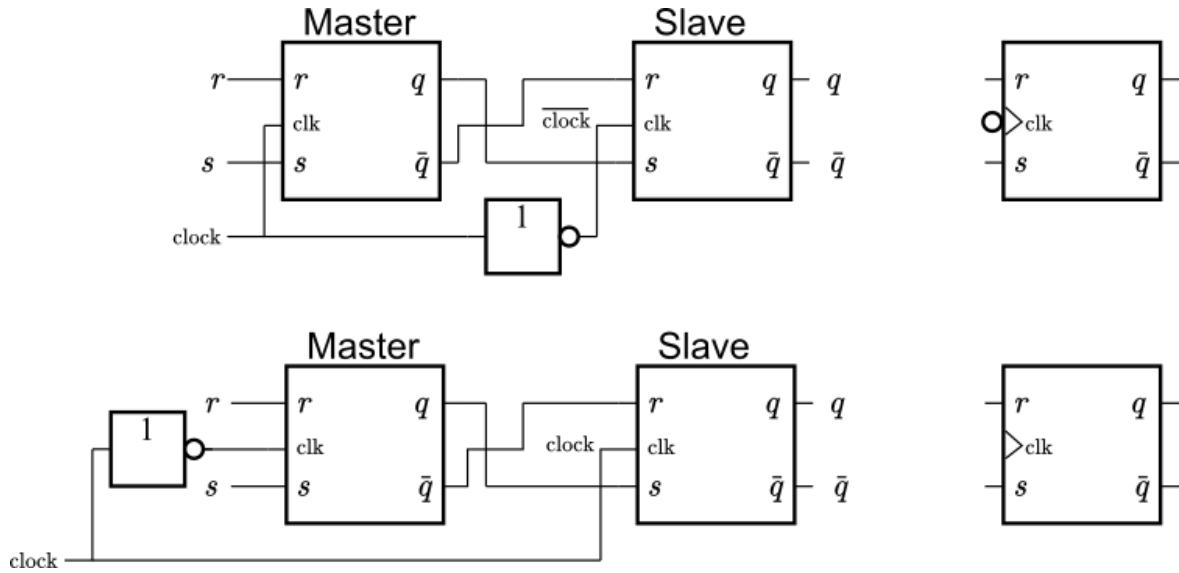
Während sich die Ausgänge des Masters ändern, passiert beim Slave nichts, da dort das Takt Signal invertiert wurde (Eingang $clock=0$) und das Slave Flipflop damit deaktiviert ist

Bei fallender Flanke werden die Ausgänge des Masters vom Slave eingelesen, die Ausgänge des Slaves ändern sich

Eine Änderung der Ausgänge ist beim Master-Slave Flipflop damit nur bei einer fallenden Flanke möglich (egal zu welchem Zeitpunkt sich die Eingangssignale ändern) ([Öffnen in Amilosim](#))

Taktflankengesteuertes RS-Flipflop

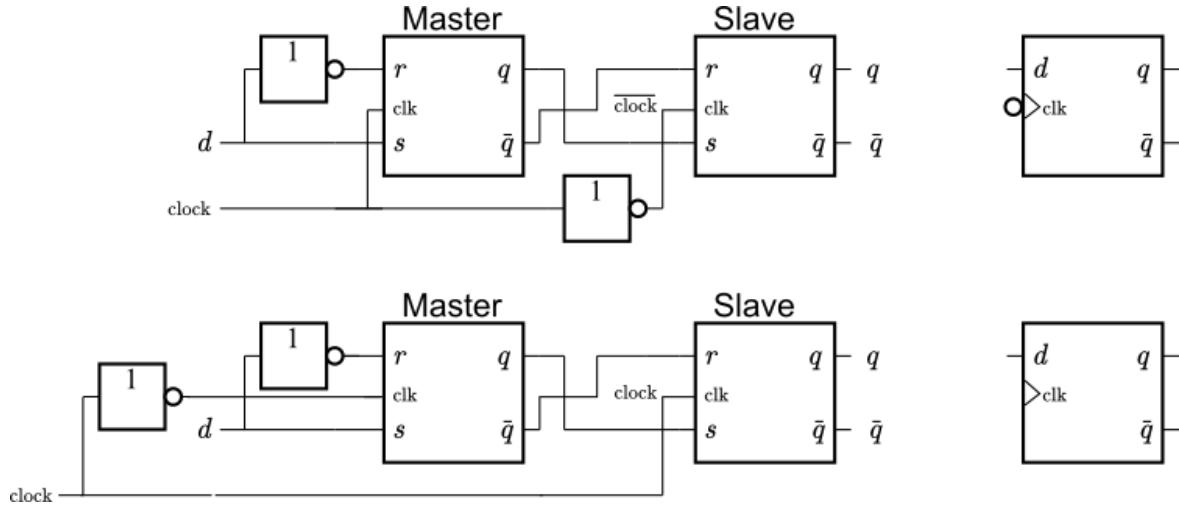
Es werden die folgenden zwei Ersatzschaltbilder für das taktflankengesteuerte RS-Flipflop verwendet, je nachdem, ob sich die Ausgänge bei fallender (Abb. oben) oder steigender (Abb. unten) Flanke ändern



Taktflankengesteuertes D-Flipflop

Ein taktflankengesteuertes RS-Flipflop kann leicht in ein taktflankengesteuertes D-Flipflop umgewandelt werden

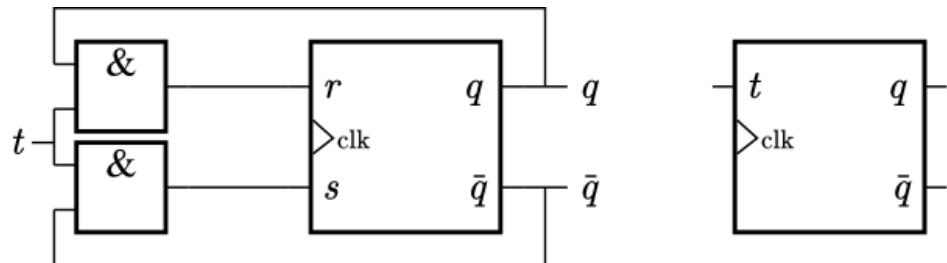
Auch für das taktflankengesteuerte D-Flipflop werden Ersatzschaltbilder eingeführt



T-Flipflop

Neben RS- und D-Flipflops gibt es auch T-Flipflops

Diese können aus einem RS-Flipflop durch das Vorschalten zweier UND-Gatter erzeugt werden



Wie das D-Flipflop hat das T-Flipflop nur einen Eingang t

Ist $t=1$ ändert sich der Ausgang q von 0 nach 1 bzw. von 1 nach 0. Der Ausgang wird also jeweils umgeschaltet (engl. "toggle"), daher der Name T-Flipflop.

Ist das T-Flipflop einmal in einem stabilen Zustand, bleibt es stabil, da die beiden Ausgänge q und \bar{q} nicht gleichzeitig 1 sein können

JK-Flipflop

Eine weitere Variante, die universell einsetzbar ist, ist das JK-Flipflop

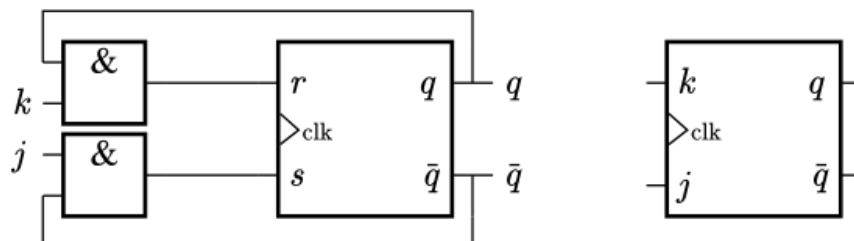
Dieses kann aus einem RS-Flipflop durch das Vorschalten zweier UND-Gatter erzeugt werden

Ist $j=1$ und $k=1$, entspricht das Verhalten dem eines T-Flipflop

Ansonsten dem Verhalten eines RS-Flipflops

D.h. die beim RS-Flipflop illegale Eingangsbelegung $j=1$ und $k=1$ bekommt eine eigene Funktionalität zugewiesen

k	j	q_{t-1}	q	Funktion
0	0	0	0	halten
0	0	1	1	halten
0	1	0	1	setzen
0	1	1	1	setzen
1	0	0	0	rücksetzen
1	0	1	0	rücksetzen
1	1	0	1	umschalten
1	1	1	0	umschalten



Register

Ein Register besteht aus mehreren Speicherelementen, die parallel gelesen bzw. geschrieben werden können

Eine wichtige Kenngröße ist die Anzahl n der Speicherelemente, z.B. 8-, 16-, 32-, 64-, 128-Bit Register

Im Folgenden werden drei verschiedene Registrytypen vorgestellt

Auffangregister

Schieberegister

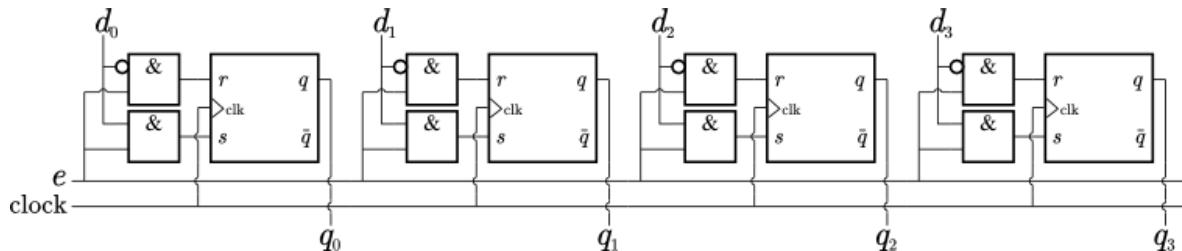
Universalregister

Auffangregister

Das Auffangregister dient zur Zwischenspeicherung von Datenworten

Ein n -Bit Auffangregister kann aus einer parallelen Anordnung von n Flipflops bestehen, die jeweils 1-Bit speichern

Hier wird die Realisierung eines 4-Bit Registers mit taktflankengesteuerten RS-Flipflops gezeigt, die mit einem gemeinsamen Takt *clock* und gemeinsamem Enable-Signal *e* angesteuert werden



Wenn $e=1$ ist, werden die Datenbits d_i parallel mit steigender Flanke in das Register übernommen

An den Ausgängen q_i kann das Datenwort ausgelesen werden

Wenn $e=0$ ist, werden die Ausgänge konstant gehalten, d.h. das Datenwort ist gespeichert

[Bildquelle: frei nach: D. W. Hoffmann: *Grundlagen der Technischen Informatik*, 2. Auflage; Hanser 2009, Abb. 9.2, S. 310]

Schieberegister

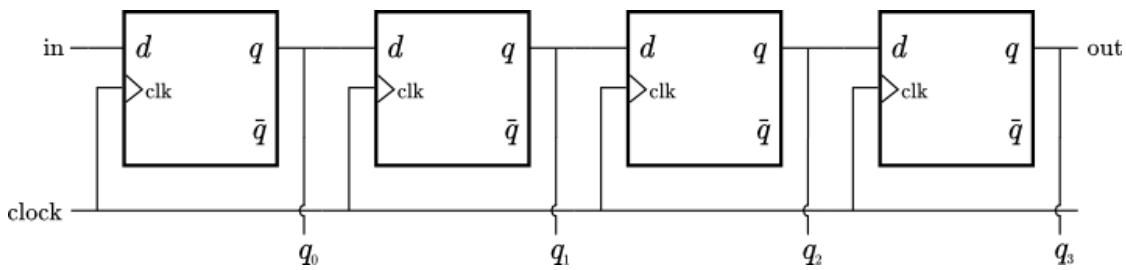
Ein n -bit Schieberegister hat nur einen Eingang in aber n parallele Ausgänge

Mit jedem Takt wird der Eingang in abgefragt und der Wert im ersten Flipflop gespeichert

Der Ausgang eines Flipflops ist jeweils mit dem Eingang des nächsten Flipflops verbunden

Mit jedem Takt wird die Information ein Register weitergeschoben (daher der Name "Schieberegister")

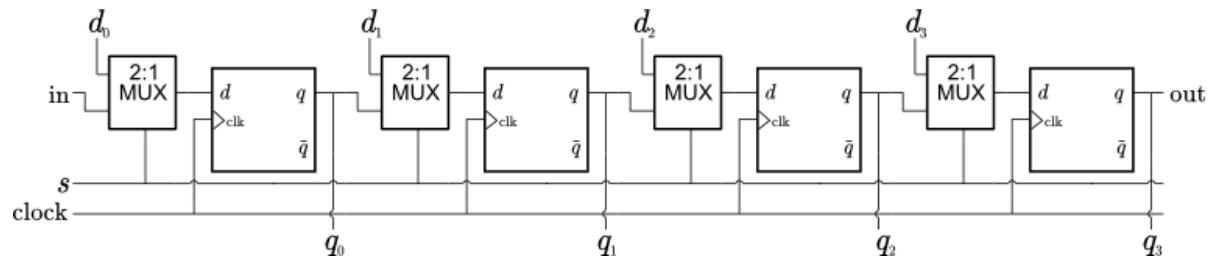
Hier ist die Realisierung eines 4-Bit Registers mit taktflankengesteuerten D-Flipflops gezeigt



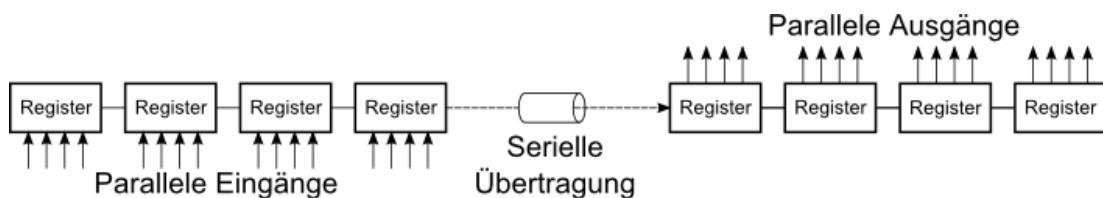
Eine wichtige Anwendung ist das Umsetzen eines seriellen Datenstroms in Datenwörter der Länge n

Schieberegister

Durch Vorschalten eines 2-zu-1 Multiplexers kann das Schieberegister so erweitert werden, dass entweder eine Schiebeoperation ausgeführt wird (Steuerleitung $s=1$) oder Daten parallel geladen werden (Steuerleitung $s=0$)



Anwendung: Serielle Übertragung



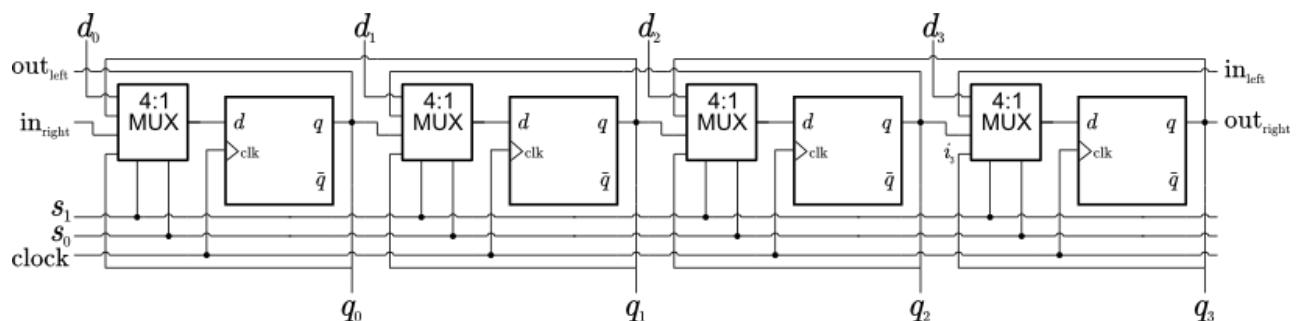
Universalregister

Das Universalregister vereint die Funktionalität des Auffang- und des Schieberegisters

Das hier gezeigte 4-Bit Universalregister besteht aus 4 taktflankengesteuerten D-Flipflops mit jeweils einem vorgeschalteten 4-zu-1 Multiplexer

Mit den gemeinsamen Steuerleitungen s_0 und s_1 der Multiplexer kann die gewünschte Funktion ausgewählt werden (siehe Tabelle)

s_1	s_0	Funktion
0	0	laden
0	1	links schieben
1	0	rechts schieben
1	1	halten



Zähler

Mit Registern lassen sich leicht Zähler aufbauen

Im Folgenden werden drei Beispiele gezeigt

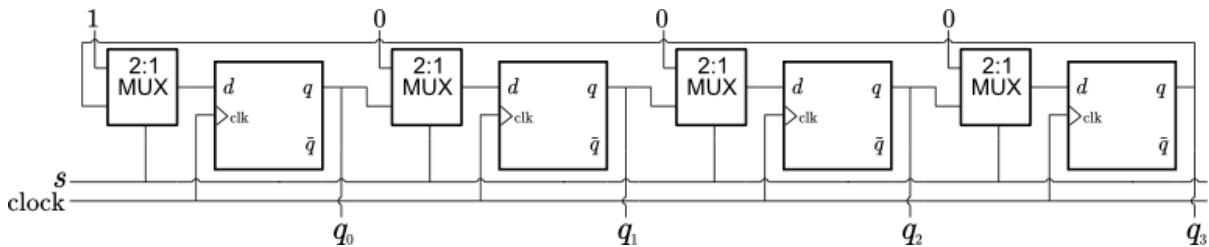
Ringzähler

Johnson-Zähler

Binärzahler

Ringzähler

Zur Realisierung eines Ringzählers kann ein rückgekoppeltes Schiebregister verwendet werden



Ein 4-Bit Ringzähler erzeugt z.B. als Ausgabe die Folge: 1000, 0100, 0010, 0001, 1000, 0100, ...

Es ist also jeweils nur ein Ausgang 1, die anderen 0

Das Schieberegister wird mit 1000 initialisiert (Steuerleitung $s=0$)

Anschießend wird die 1 nach rechts durch die Flipflops geschoben (Steuerleitung $s=1$)

Ein Ringzähler kann u.a. eingesetzt werden, um Aktionen nacheinander auszuführen. Dazu kann jeder Ausgang q_i mit einer Aktion verknüpft werden, die ausgeführt wird, sobald der Ausgang 1 ist

Ringzähler

Ein Ringzähler kann als Frequenzteiler verwendet werden

Wird ein 4-Bit Ringzähler mit 1010 initialisiert, wird die Taktrate halbiert

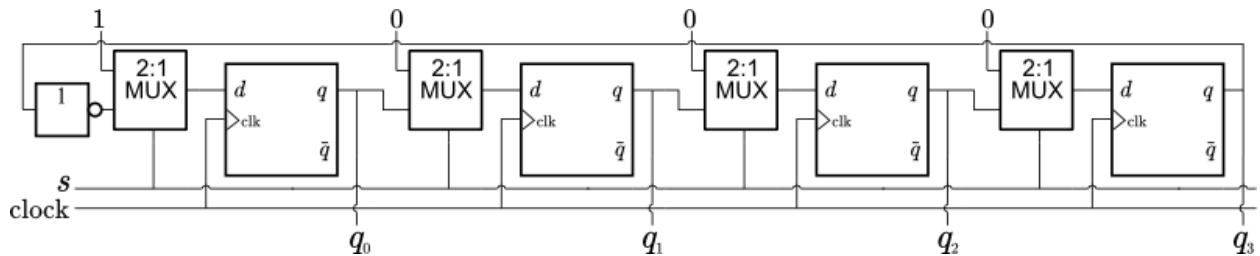
Initialisierung mit 1100, resultiert in einem 4-fach langsameren Takt

1. Steigende Flanke: 1100
2. Steigende Flanke: 0110
3. Steigende Flanke: 0011
4. Steigende Flanke: 1001 (Zyklus beginnt von vorn)

Der Eingangstakt hat 4 steigende Flanken, das Signal an einem Ausgang nur eine, daher 4-fach verlangsamter Takt

Johnson-Zähler

Beim Johnson-Zähler (auch Möbius-Zähler genannt) wird der rückgekoppelte Wert invertiert



Dadurch ergibt sich bei Initialisierung mit 1000 die Folge: 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000, 1000, ...

Binärzähler

Bei einem Binärzähler werden aufsteigende Binärzahlen erzeugt (hier niederwertiges Bit links)

Es soll also die Folge entstehen:

0000, 1000, 0100, 1100, 0010, 1010, 0110, 1110

0001, 1001, 0101, 1101, 0011, 1011, 0111, 1111

0000, ...

Ein Binärzähler kann ebenfalls mit Flipflops realisiert werden, allerdings werden einige weitere Logikbausteine benötigt

Das niederwertigste Bit q_0 wechselt mit jedem Takt, dies kann mit einem Inverter realisiert werden

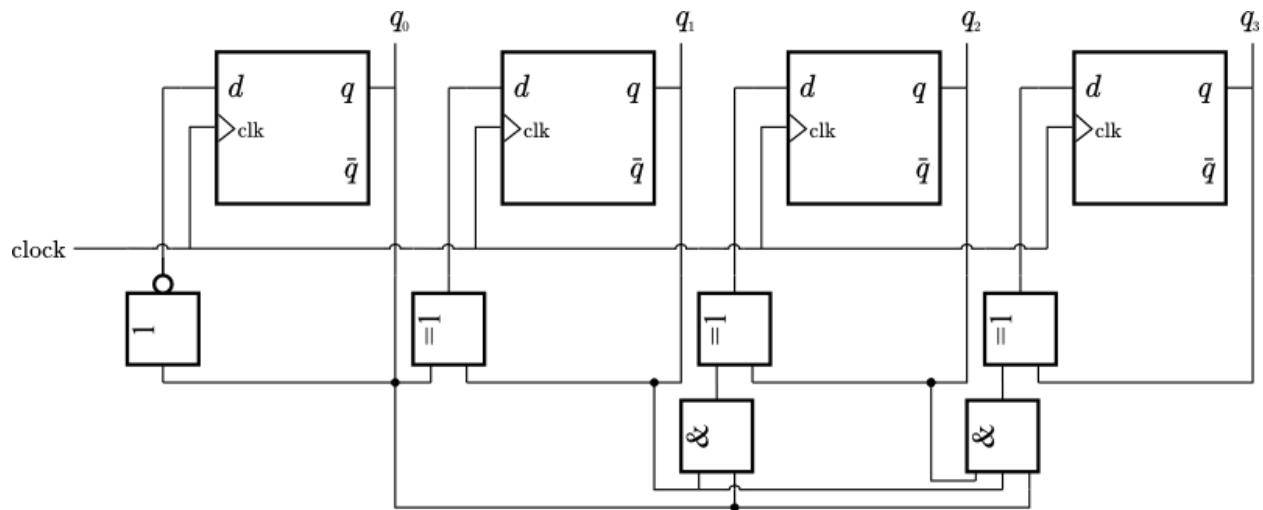
$$q_0^{t+1} = \neg q_0^t$$

Für q_1 gilt: $q_1^{t+1} = q_1^t \leftrightarrow q_0^t$

Für q_2 gilt: $q_2^{t+1} = q_2^t \leftrightarrow (q_1^t \wedge q_0^t)$

Für q_3 gilt: $q_3^{t+1} = q_3^t \leftrightarrow (q_2^t \wedge q_1^t \wedge q_0^t)$

Binärzahler



$$q_0^{t+1} = \neg q_0^t$$

$$q_1^{t+1} = q_1^t \leftrightarrow q_0^t$$

$$q_2^{t+1} = q_2^t \leftrightarrow (q_1^t \wedge q_0^t)$$

$$q_3^{t+1} = q_3^t \leftrightarrow (q_2^t \wedge q_1^t \wedge q_0^t)$$

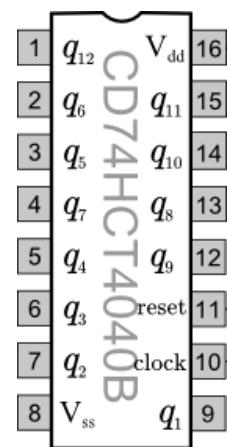
Praxisbeispiel: Binärzähler

Der CMOS IC CD74HCT4040B ist ein 12-Bit Binärzähler mit den Ausgängen

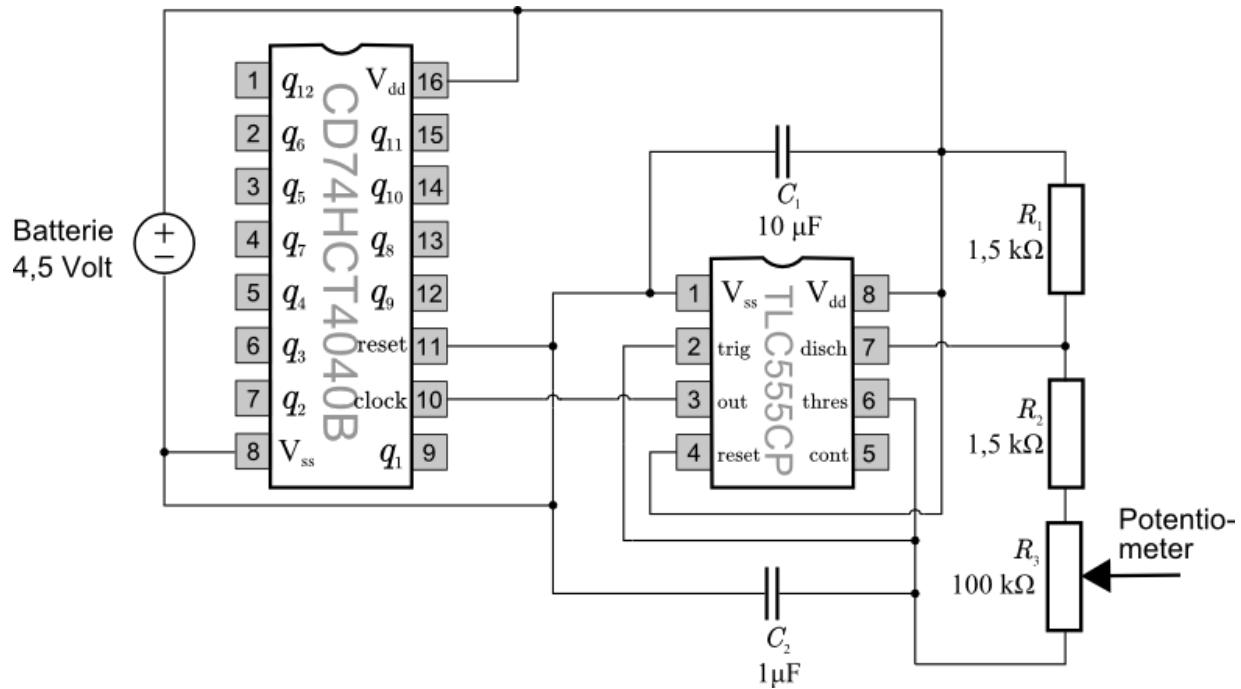
$$q_1, q_2, \dots, q_{12}$$

Ist $\text{reset} = 0$ und liegt am Pin *clock* ein Takt an, zählt der Baustein bei jeder fallenden Flanke um einen Schritt hoch

Der Takt kann z.B. mit dem Timer-Baustein TLC555CP erzeugt werden (nähere Informationen im [Datenblatt](#))



Praxisbeispiel: Binärzähler



Thorsten Thormählen 40 / 42

Praxisbeispiel: Binärzähler

Dieses Video zeigt den Aufbau der Schaltung aus der vorangegangenen Folie auf eine Steckplatine

0:00 / 0:10



Thorsten Thormählen 41 / 42

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) ,]

Thorsten Thormählen 42 / 42

Technische Informatik

Automaten

Thorsten Thormählen
10. Dezember 2023
Teil 7, Kapitel 4

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Automaten

Endliche Automaten

Akzeptoren und Transduktoren

Mealy- und Moore-Automaten

Beispiele

Schaltnetze und Schaltwerke

Ohne Rückkopplung (Schaltnetz)

Werte an den Ausgängen sind nur abhängig von den Eingängen

Solche Schaltungen verhalten sich immer gleich (sind zustandslos) und sind durch ihre Schaltfunktion eindeutig beschrieben

Es ist jedoch nicht möglich, etwas zu speichern

Mit Rückkopplung (Schaltwerk)

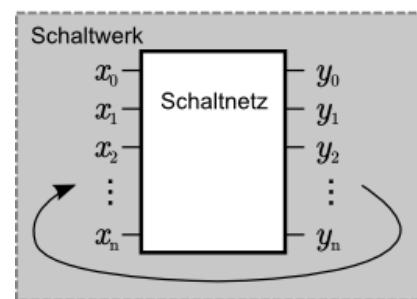
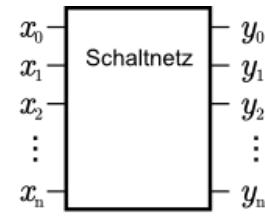
Werte an den Ausgängen sind abhängig von den Eingängen und den vorherigen Ausgangswerten

Das Zeitverhalten muss genau betrachtet werden

Die vorherigen Ausgangswerte können als Zustand der Gatter interpretiert werden.

Abhängig vom Zustand verhalten sich die Gatter anders (zustandsabhängige Schaltfunktion)

Es wird möglich, Zustände zu speichern



Schaltwerke

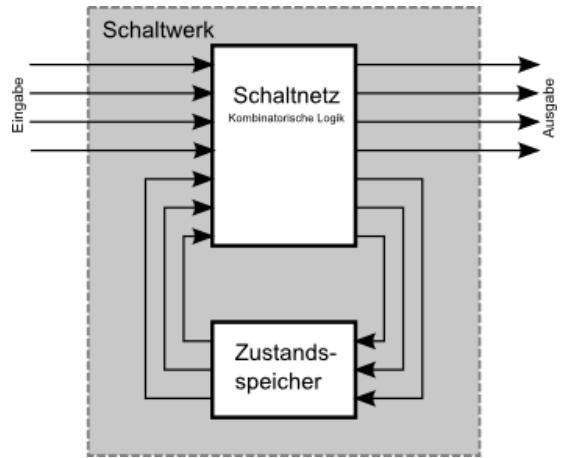
Im vorangegangenen Kapitel wurde gezeigt, dass durch Rückkopplung Speicherelemente (Flip-Flops, Register) realisiert werden können, die einen Zustand speichern

Demnach kann ein Schaltwerk auch als eine Kombination aus einem Schaltnetz und einem Zustandsspeicher dargestellt werden

Die Ausgabe eines Schaltwerks kann abhängig vom aktuellen Zustand sein

Abhängig von den Eingangswerten kann sich der Zustand eines Schaltwerks ändern

Zur Beschreibung der zustandsabhängigen Schaltfunktion können *endliche Automaten* verwendet werden



Endliche Automaten

Ein *endlicher Automat* (engl. Finite State Machine, FSM) ist definiert durch

eine endliche Menge \mathcal{A} von Eingabesymbolen $a_i \in \mathcal{A}$ (Alphabet)

eine endliche Menge \mathcal{S} von Zuständen $s_i \in \mathcal{S}$

einen Anfangszustand $s_0 \in \mathcal{S}$

eine Zustandsübergangsfunktion $\delta : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$

Des Weiteren kann er umfassen

eine endliche Menge \mathcal{B} von Ausgabesymbolen $b_i \in \mathcal{B}$

eine Ausgabefunktion $\lambda : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{B}$

Bei deterministischen Automaten erfolgen die Zustandsübergänge deterministisch (nicht zufällig)

Endliche Automaten können durch Zustandsübergangsgraphen dargestellt werden

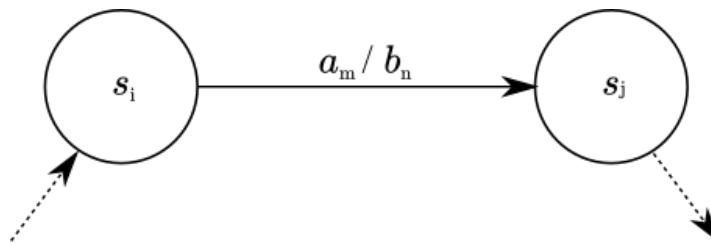
Zustandsübergangsgraphen

In einem Zustandsübergangsgraphen wird jeder Zustand $s_i \in \mathcal{S}$ als ein Kreis dargestellt

Die möglichen Zustandsübergänge werden mit Pfeilen gekennzeichnet

Jeder Pfeil wird mit dem zugehörigen Eingabesymbol $a_m \in \mathcal{A}$ beschriftet, für das dieser Zustandsübergang auftritt

Außerdem (abgetrennt durch einen "/") kann jeweils das Ausgabesymbol $b_n \in \mathcal{B}$ angegeben werden



Zustandsübergangsgraphen

Beispiel: Ampelschaltung

Es soll eine Schaltung für eine Fußgängerampelanlage erstellt werden. Es wird dabei die Ampel, die den Autoverkehr regelt, betrachtet (nicht die Fußgängerampel)

Die Ampel reagiert auf das Drücken eines Ampelknopfs durch einen Fußgänger:
 $a_0 = 0$ bedeutet der Ampelknopf wurde nicht gedrückt

$a_1 = 1$ bedeutet der Ampelknopf wurde gedrückt

Im Anfangszustand $s_0 \in \mathcal{S}$ ist die Ampel grün

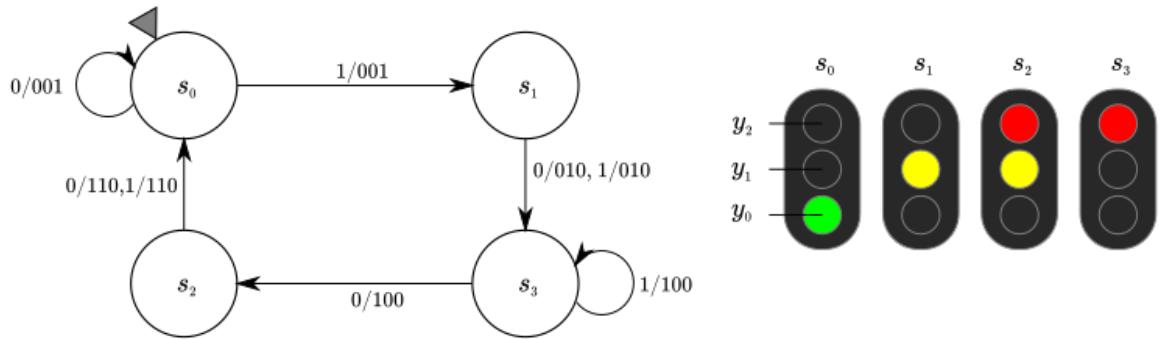
Wurde der Ampelknopf gedrückt, soll die Ampel zunächst auf gelb und dann auf rot schalten

Es wird weiterhin davon ausgegangen, dass das durch den Ampelknopf gesteuerte Eingabesymbol automatisch, nachdem der Fußgänger genug Zeit hatte die Straße zu überqueren, von a_1 nach a_0 wechselt

Die Ampel soll dann zunächst gelb-rot zeigen und schließlich wieder grün

Zustandsübergangsgraphen

Beispiel: Ampelschaltung



Menge der Eingabesymbole $\mathcal{A} = \{a_0, a_1\}$ binär kodiert mit $\{0, 1\}$

Menge der Zustände $\mathcal{S} = \{s_0, s_1, s_2, s_3\}$

Menge der Ausgabesymbole $\mathcal{B} = \{b_0, b_1, b_2, b_3\}$ binär kodiert mit $y_2y_1y_0$ zu $\{001, 010, 110, 100\}$

Es gibt $|\mathcal{S}| \cdot |\mathcal{A}| = 4 \cdot 2 = 8$ mögliche Zustandsübergänge

Transduktoren und Akzeptoren

Transduktoren

Enthält ein Automat eine Ausgabefunktion λ wird er als Transduktor bezeichnet

Transduktoren sind Automaten, die aus Sequenzen von (binären) Eingabesymbolen, Sequenzen von (binären) Ausgabesymbolen erzeugen

Diese "übersetzen" die Eingabe in eine Ausgabe

Akzeptoren

Akzeptoren sind Automaten, die bestimmt Sequenzen von Eingabewörtern akzeptieren oder verwerfen

Dazu wird eine Teilmenge $\mathcal{F} \subset \mathcal{S}$ der möglichen Zustände zu akzeptierenden Zuständen erklärt, so genannte "Finalzustände"

Endet ein Automat nach Abarbeiten einer Sequenz von Eingabesymbolen in einem Finalzustand, wird die Eingabe akzeptiert, sonst wird sie verworfen

Akzeptoren erzeugen somit keine Sequenz von Ausgabesymbolen

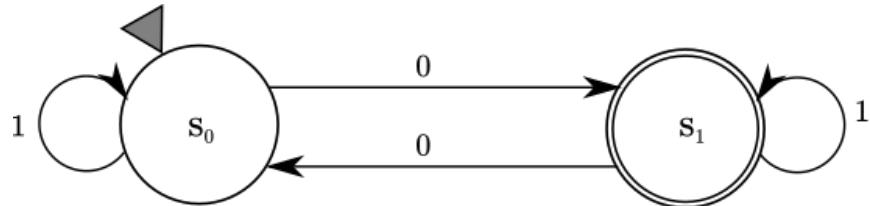
Akzeptoren

Finalzustände werden im Zustandsübergangsgraphen als doppelt gestrichene Kreise dargestellt

Der Anfangszustand wird mit einem auf den Zustand zeigendem Dreieck gekennzeichnet

Beispiel für einen Akzeptor:

Ausgehend vom Anfangszustand s_0 würde der dargestellte Automat alle Eingabesequenzen akzeptieren, die eine ungerade Anzahl an Nullen enthalten



Menge der Eingabewörter $\mathcal{A} = \{0, 1\}$

Menge der Zustände $\mathcal{S} = \{s_0, s_1\}$

Menge der Finalzustände $\mathcal{F} = \{s_1\}$

Mealy und Moore-Automaten

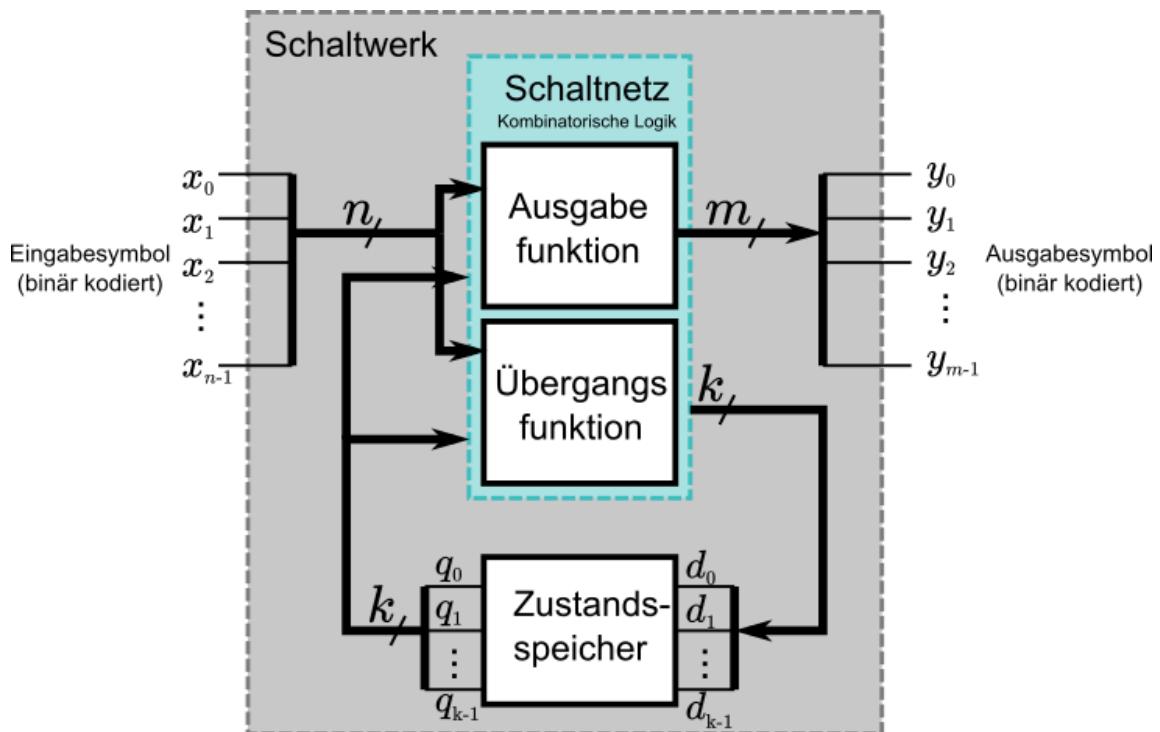
Mealy-Automaten

Bei einem Mealy-Automaten ist die Ausgabefunktion
 $\lambda : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{B}$ vom aktuellen Zustand und der
Eingabe abhängig

Moore-Automaten

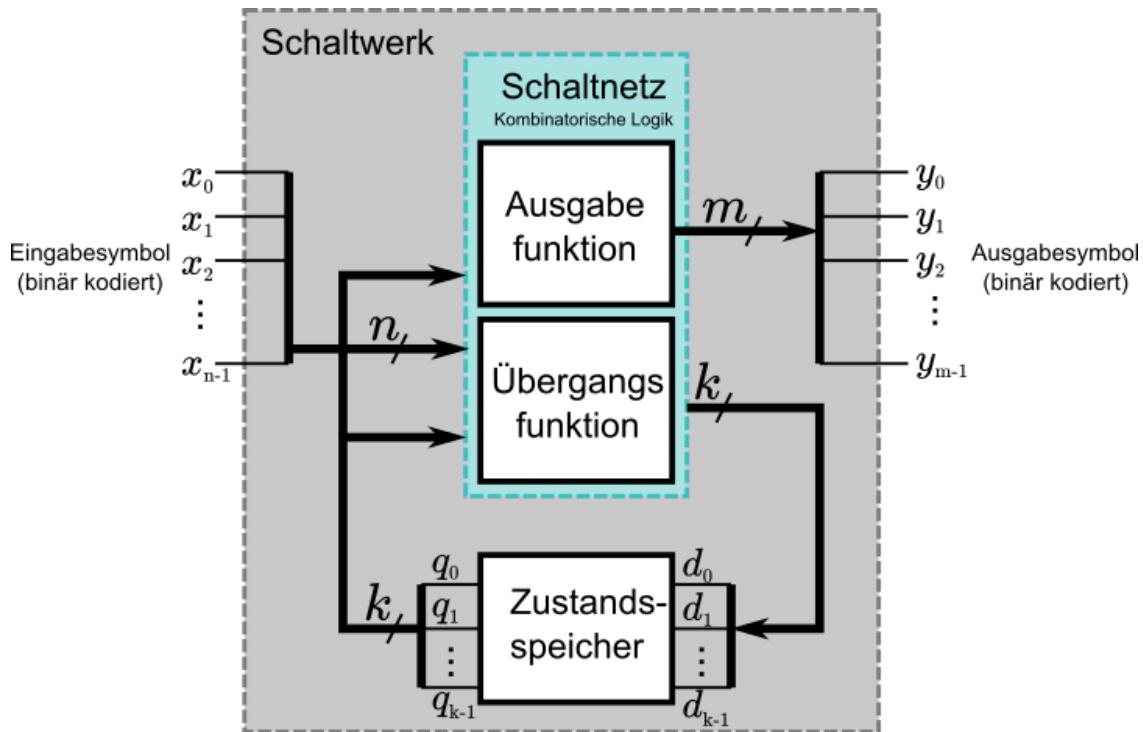
Bei einem Moore-Automaten ist die Ausgabefunktion
 $\lambda : \mathcal{S} \rightarrow \mathcal{B}$ nur vom aktuellen Zustand abhängig

Mealy-Automat



Thorsten Thormählen 14 / 24

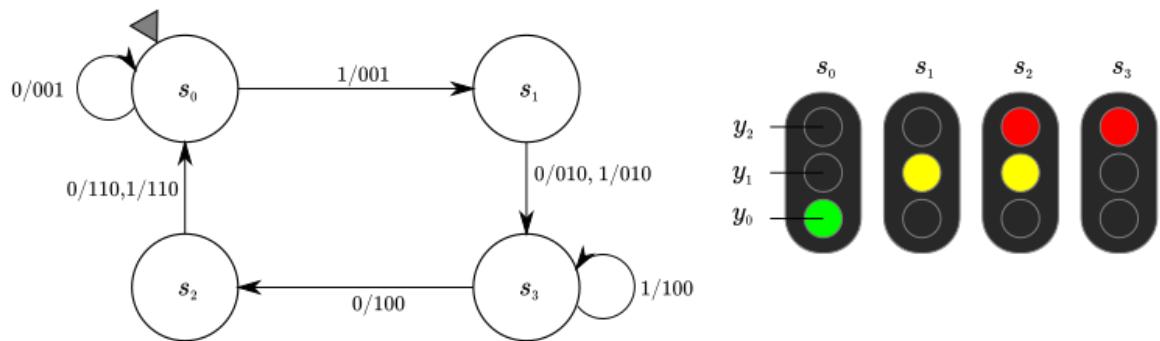
Moore-Automat



Thorsten Thormählen 15 / 24

Vom Zustandsübergangsgraphen zum Schaltwerk

Beispiel 1: Der Zustandsübergangsgraph der Ampelschaltung soll in ein Schaltwerk umgesetzt werden

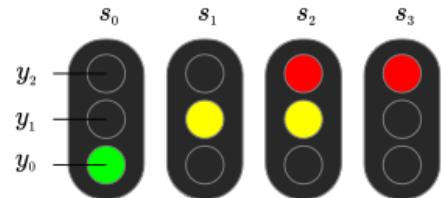
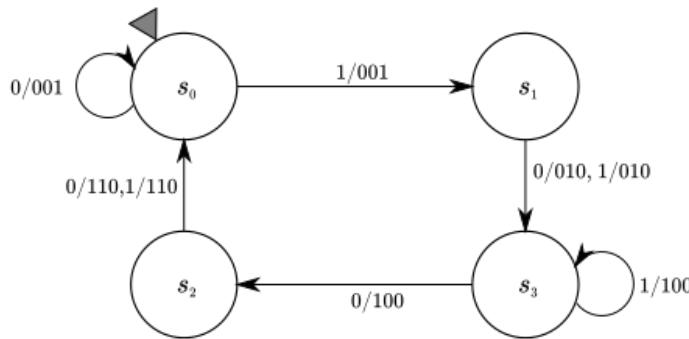


Zur Erstellung des Schaltwerks kann die Ausgabefunktion, die Übergangsfunktion und der Zustandsspeicher separat betrachtet werden

Für gewünschte Ausgabe- und Übergangsfunktion kann direkt aus dem Zustandsübergangsgraphen abgelesen und als Wahrheitstafel dargestellt werden

Zur Minimierung können anschließend z.B. KV-Diagramme oder das Quine-McCluskey-Verfahren verwendet werden

Vom Zustandsübergangsgraphen zum Schaltwerk



Handelt es sich um einen Moore- oder Mealy-Automaten?

Antwort: Moore-Automat, da
Ausgabefunktion nur abhängig vom
aktuellen Zustand

Ausgabefunktion:

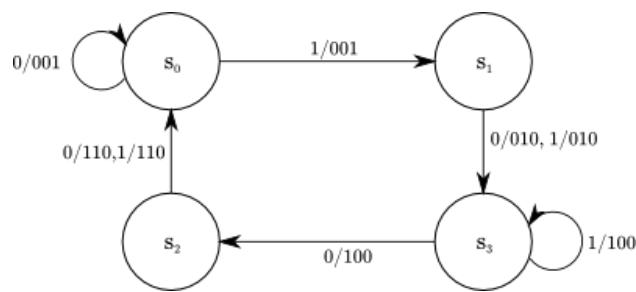
$$y_2 = q_1$$

$$y_1 = q_0 \leftrightarrow q_1$$

$$y_0 = \neg q_0 \wedge \neg q_1 = \neg(q_0 \vee q_1)$$

Zustand	q_1	q_0	y_2	y_1	y_0
s_0	0	0	0	0	1
s_1	0	1	0	1	0
s_2	1	0	1	1	0
s_3	1	1	1	0	0

Vom Zustandsübergangsgraphen zum Schaltwerk



Übergangsfunktion:

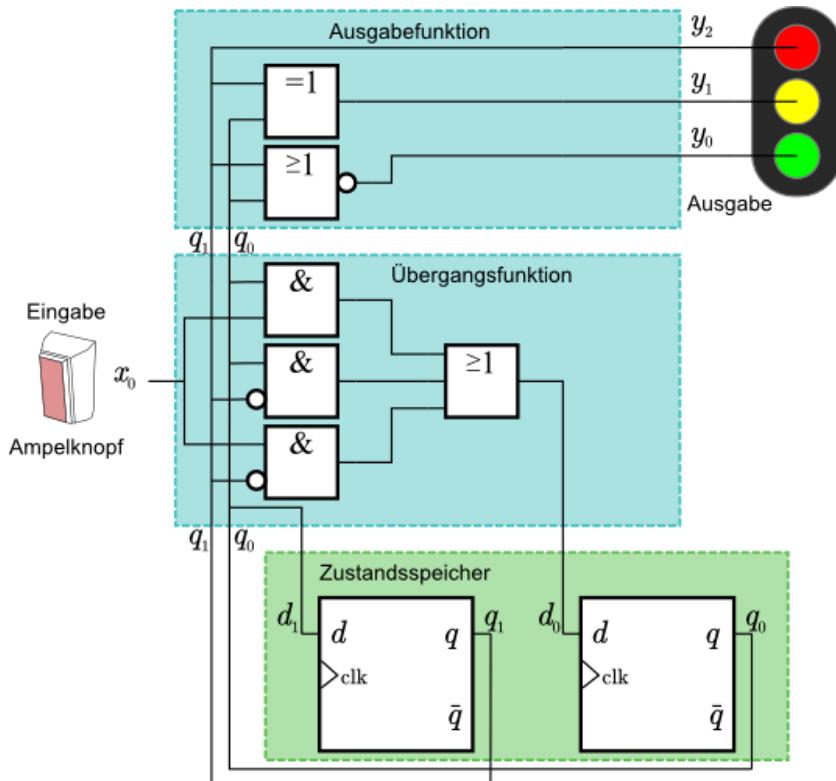
$$d_1 = q_0$$

Mittels KV-Diagramm oder Quine-McCluskey-Verfahren:

$$d_0 = (\bar{q}_1 x_0) \vee (\bar{q}_1 q_0) \vee (q_0 x_0)$$

Zustand	q_1	q_0	x_0	d_1	d_0
s_0	0	0	0	0	0
s_0	0	0	1	0	1
s_1	0	1	0	1	1
s_1	0	1	1	1	1
s_2	1	0	0	0	0
s_2	1	0	1	0	0
s_3	1	1	0	1	0
s_3	1	1	1	1	1

Vom Zustandsübergangsgraphen zum Schaltwerk



Thorsten Thormählen 19 / 24

Vom Zustandsübergangsgraphen zum Schaltwerk

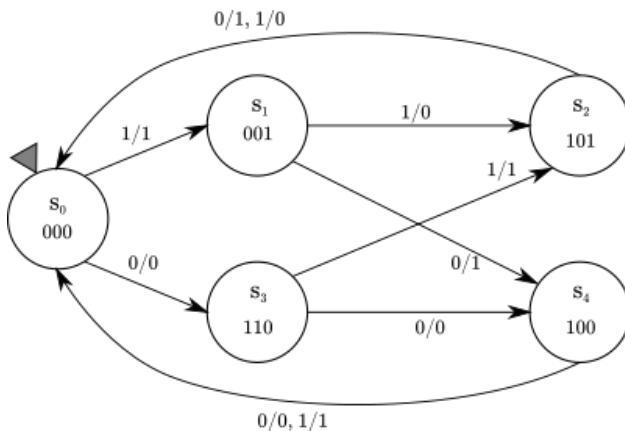
Beispiel 2: Dieser Zustandsübergangsgraph übersetzt einen 3-Bit Binär-Code in einen Gray-Code

Menge der Eingabesymbole $\mathcal{A} = \{0, 1\}$

Menge der Zustände $\mathcal{S} = \{s_0, s_1, s_2, s_3, s_4\}$

Menge der Ausgabesymbole $\mathcal{B} = \{0, 1\}$

Anfangszustand s_0



Eingabe	Ausgabe
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

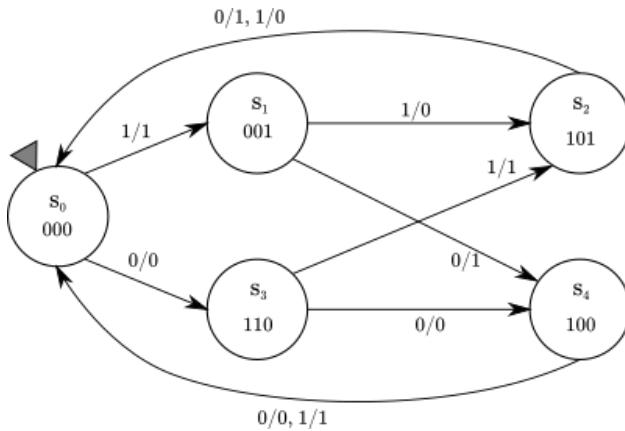
[Quelle: basierend auf: D.W. Hoffmann: Grundlagen der Technischen Informatik, 2. Auflage; Hanser 2009, Abb. 9.2, S. 298ff]

Thorsten Thormählen 20 / 24

Vom Zustandsübergangsgraphen zum Schaltwerk

Aufgabe: Entwurf des entsprechenden Schaltwerks

Handelt es sich um einen Moore- oder Mealy-Automaten?



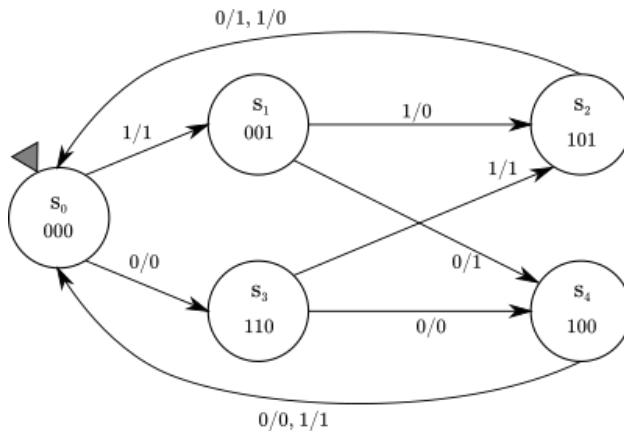
Eingabe	Ausgabe
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Antwort: Mealy-Automat, da Ausgabefunktion abhängig vom aktuellen Zustand und der Eingabe

[Quelle: basierend auf: D. W. Hoffmann: Grundlagen der Technischen Informatik, 2. Auflage; Hanser 2009, Abb. 9.2, S. 298ff]

Thorsten Thormählen 21 / 24

Vom Zustandsübergangsgraphen zum Schaltwerk



Mittels KV-Diagramm oder durch Ablesen und Vereinfachen ergibt sich:

$$d_2 = \bar{q}_2 \bar{q}_1 \bar{q}_0 \bar{x}_0 \vee \bar{q}_2 \bar{q}_1 q_0 \vee q_2 q_1 \bar{q}_0$$

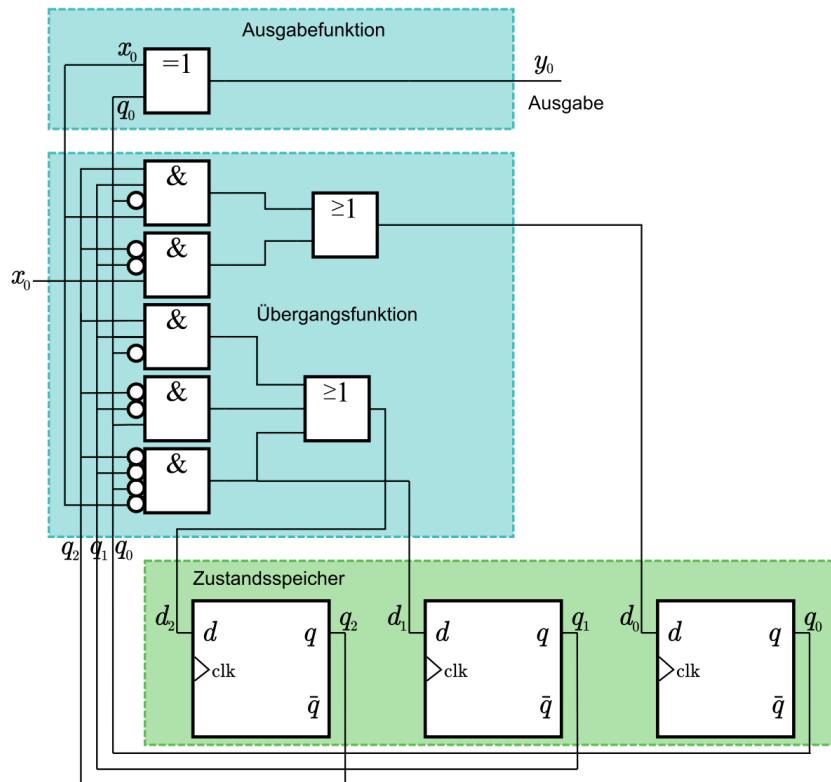
$$d_1 = \bar{q}_2 \bar{q}_1 \bar{q}_0 \bar{x}_0$$

$$d_0 = \bar{q}_2 \bar{q}_1 x_0 \vee q_2 q_1 \bar{q}_0 x_0$$

$$y_0 = x_0 \leftrightarrow q_0$$

q_2	q_1	q_0	x_0	d_2	d_1	d_0	y_0
0	0	0	0	0	1	1	0
0	0	0	1	0	0	1	1
0	0	1	0	1	0	0	1
0	0	1	1	1	0	1	0
1	0	1	0	0	0	0	1
1	0	1	1	0	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1

Vom Zustandsübergangsgraphen zum Schaltwerk



Thorsten Thormählen 23 / 24

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) ,]

Thorsten Thormählen 24 / 24

Technische Informatik

Speicher

Thorsten Thormählen
15. Dezember 2022
Teil 8, Kapitel 1

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Speicher

 Speicherhierarchie

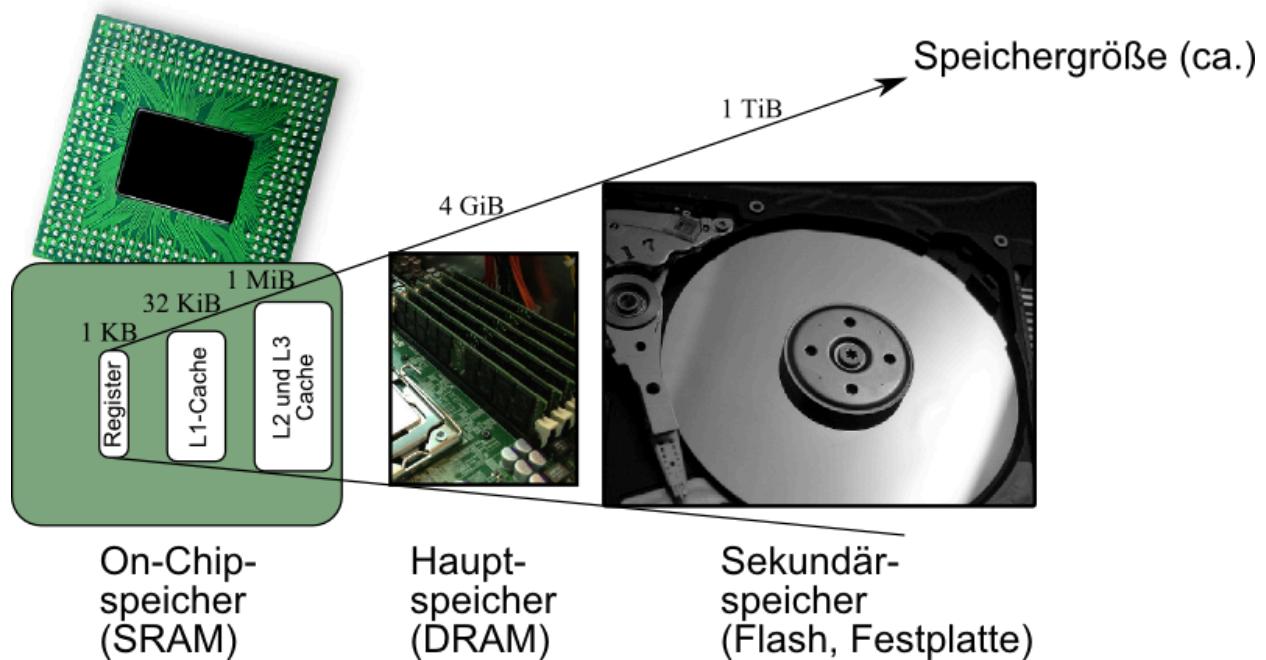
 SRAM

 DRAM

 Flash-Speicher

 Festplattenlaufwerk

Speicherhierarchie



[Bildquelle: Zusammenstellung aus: [CPU](#), Flickr user: VIA Gallery; [DRAM](#), Flickr user: Keith McDuffee; [HDD](#), Flickr user: ingenero.creativo; [Creative Commons License](#)]

Thorsten Thormählen 5 / 32

Speicherhierarchie

Schneller Speicher (d.h. Speicher mit geringer Zugriffszeit) hat hohen Aufwand pro Bit und damit einen hohen Preis pro Bit

In einem typischen Computersystem gilt: Je schneller der Speicher, desto weniger steht davon zur Verfügung

Um eine insgesamt möglichst schnelle Verarbeitung zu erreichen, sollten häufige Speicherzugriffe immer auf möglichst schnellem Speicher ausgeführt werden:

Speicherhierarchie:

Operand oder Ergebnis einer Rechenoperation → Register

Daten werden wahrscheinlich bald wieder verwendet → Cache

Daten werden gerade bearbeitet → Hauptspeicher

Daten sollen nicht-flüchtig gespeichert werden → Sekundärspeicher

Zugriffszeiten

Mittlere Zugriffszeiten flüchtiger Speicher

Register: 0,25 bis 0,5 ns

Cache (SRAM): 0,5 bis 5 ns

Hauptspeicher (DRAM): 10 bis 50 ns

Mittlere Zugriffszeiten nicht-flüchtiger Speicher

Flash-Speicher: 10 bis 250 µs

Festplatten: 3 bis 20 ms

Diese Zeiten sind nur ungefähre Größenordnungen und werden ständig geringer

Ebenfalls kommt es teilweise stark auf die Art des Zugriffs an. Konsekutiver Zugriff ("best access") auf Speicheradressen ist häufig schneller:

Cache (SRAM) 40 - 700 GiB/s

Hauptspeicher (DRAM): 10 GiB/s

Flash-Speicher: 2 GiB/s

Festplatten: 0.3 GiB/s

[Quelle: [Wikipedia: Solid-State-Drive](#)]

Thorsten Thormählen 7 / 32

Abstrakte Sicht auf Speicher

Zuordnung: Name → Wert

Die Namen entsprechen in der Regel Byteadressen

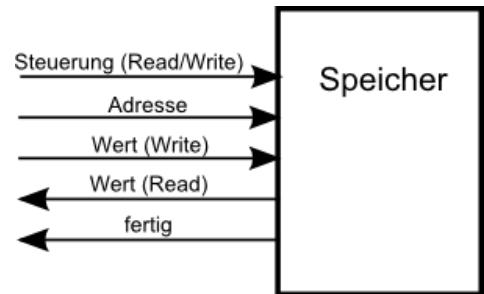
Werte besitzen häufig eine Größe von 2^n Bytes

(z.B. 4 oder 8 Byte für die Gleitkommadarstellung einer Zahl)

Folge von Lese- und Schreibzugriffen

Schreiben (engl. Write) bindet einen Wert an eine Adresse

Lesen (engl. Read) einer Adresse liefert den zuletzt an diese Adresse gebundenen Wert zurück



SRAM

SRAM (Static Random Access Memory) wird häufig zur Realisierung von schnellem Cache-Speicher verwendet

Es besteht im Prinzip aus vielen Flipflops

Im Gegensatz zu einem einzelnen Register, benötigt ein SRAM zusätzlich einer Dekodierlogik, die die Adresse in Steuersignale umsetzt, damit die richtigen Flipflops angesprochen werden

Auf der folgenden Folie wird eine mögliche Realisierung eines 4×4 -Bit SRAMs mit Hilfe von Logikgattern gezeigt

In der Realität kann eine Speicherzelle z.B. bereits aus einer Schaltung von 6 Transistoren gebildet werden

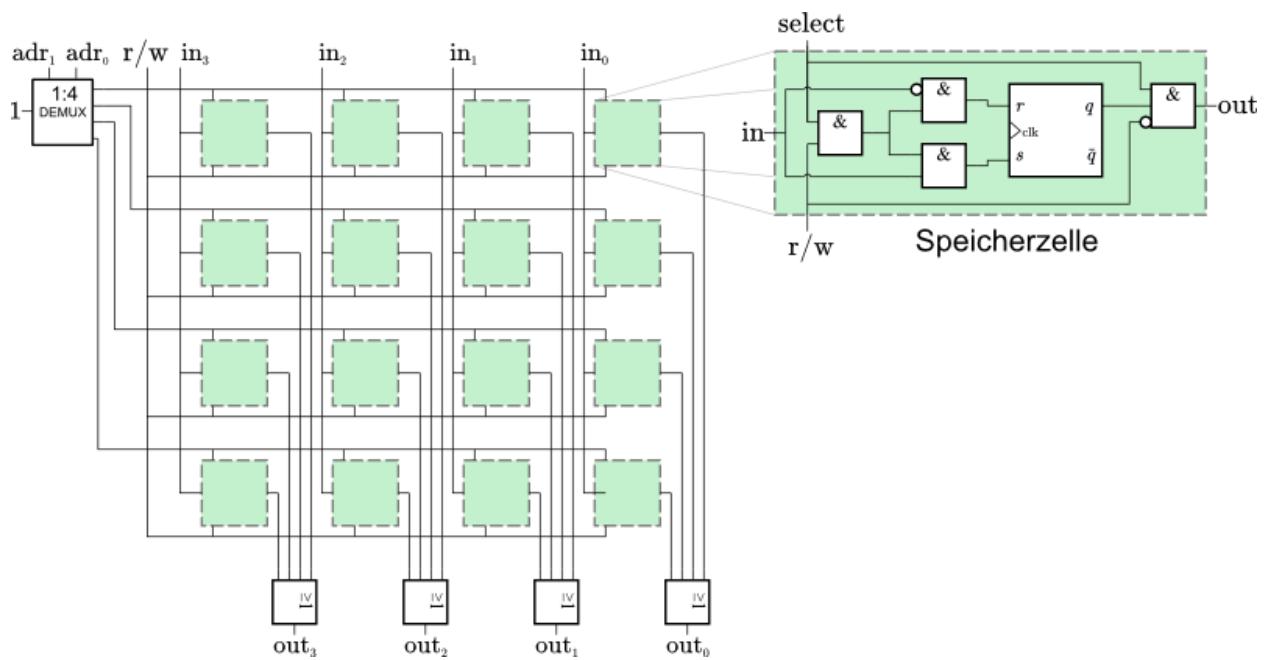
Bei Vergrößerung (rechts) durch ein Rasterelektronenmikroskop wird die regelmäßige Struktur des SRAMs sichtbar



[Bildquelle: [Die image of a STM32F103VGT6 ARM Cortex-M3 MCU](#), Autor: ZeptoBars, [Creative Commons License](#)]

Thorsten Thormählen 9 / 32

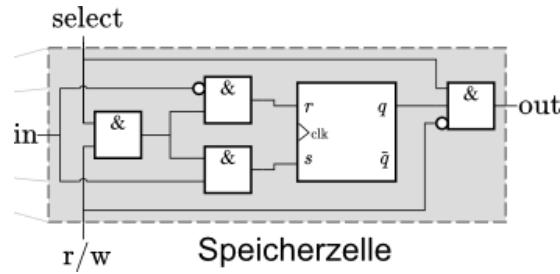
SRAM Speichermatrix



[Bildquelle: basierend auf: D. W. Hoffmann: *Grundlagen der Technischen Informatik*, 2. Auflage; Hanser 2009, Abb. 9.25, S. 329ff]

Thorsten Thormählen 10 / 32

SRAM Speicherzelle



select	r/w	in	out	q^{t+1}
0	x	x	0	q^t
1	0	0	q^t	q^t
1	0	1	q^t	q^t
1	1	0	0	0
1	1	1	0	1

Wenn $select=0$:

Die Speicherstelle wird nicht angesprochen

Der Ausgang out ist 0

Der gespeicherte Wert q ist unverändert

Wenn $select=1$:

Beim Lesen ("r/w"=0) wird der Speicherinhalt ausgegeben und bleibt unverändert

Beim Schreiben ("r/w"=1) ist der Ausgang 0 und der gespeicherte Wert q des Flipflops ändert sich gemäß dem Eingang in

DRAM

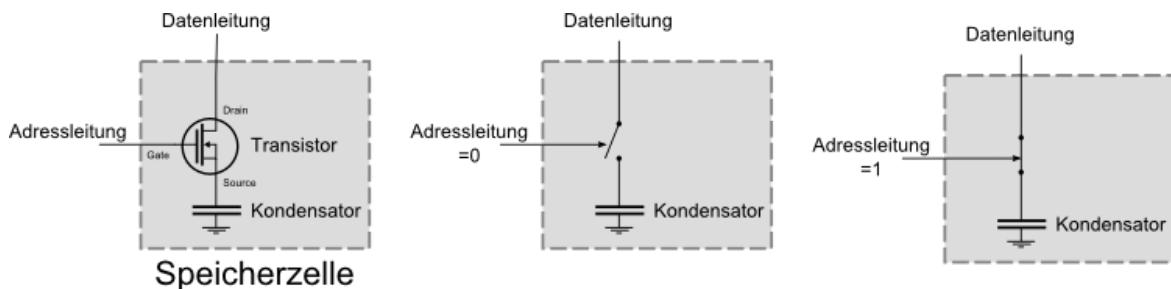
Beim DRAM (engl. Dynamic Random-Access Memory) wird eine Speicherzelle durch nur einen Kondensator und einen Transistor realisiert und benötigt daher sehr wenig Chipfläche

Damit ist es möglich, große Speicher kostengünstig herzustellen

Der Transistor wird an die Adressleitung angeschlossen und verhält sich wie ein Schalter:

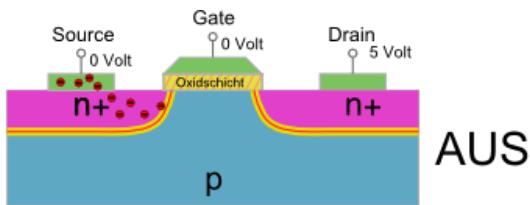
Adressleitung=1, Schalter ist geschlossen, Kondensator übernimmt die Spannung von der Datenleitung (lädt sich entweder mit Ladungsträgern auf oder nicht)

Adressleitung=0, Schalter ist geöffnet, Kondensator behält sein Spannungspotential (Ladungsträger bleiben erhalten)

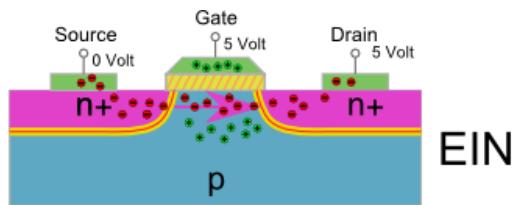


DRAM: Grundlagen Transistoren

Transistoren sind spannungsgesteuerte Schalter

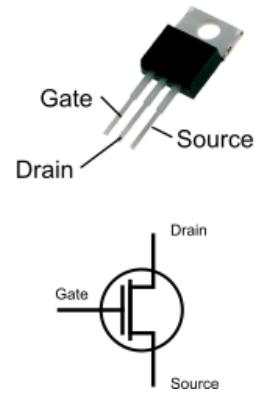


AUS



EIN

Feldeffekttransistor



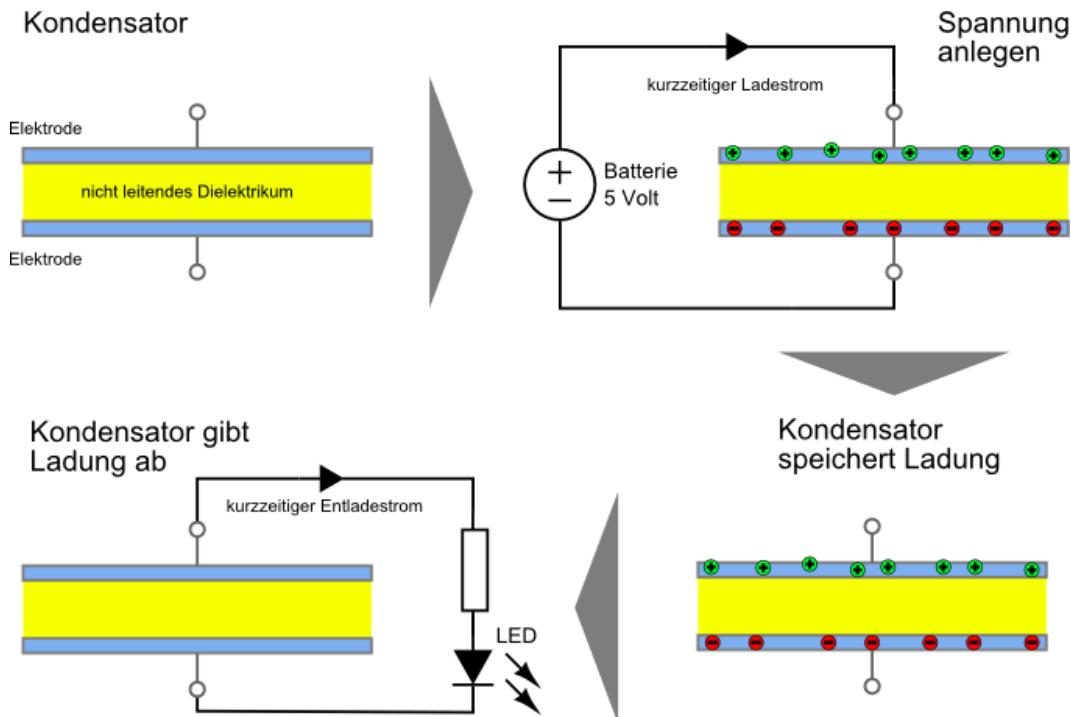
Ein Feldeffekttransistor hat drei Anschlüsse: Source, Drain und Gate.

Der Strom zwischen Source und Drain wird über die Spannung zwischen Gate und Source gesteuert

Niedrige Spannung am Gate = die Elektronen können das p-dotierte Gebiet nicht überwinden = kein Strom fließt

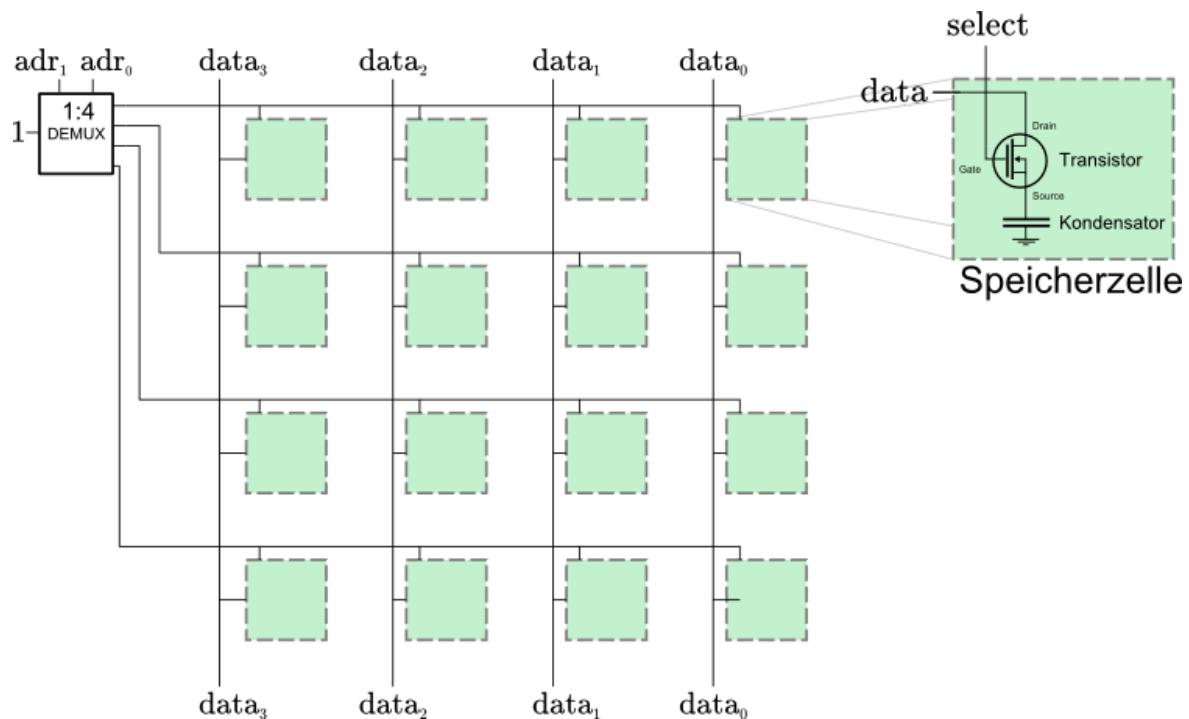
Ausreichend hohe Spannung am Gate = Elektronen reichern sich unterhalb des Gate an (n-Kanal) = Elektronen fließen durch Kanal = Strom fließt

DRAM: Grundlagen Kondensator



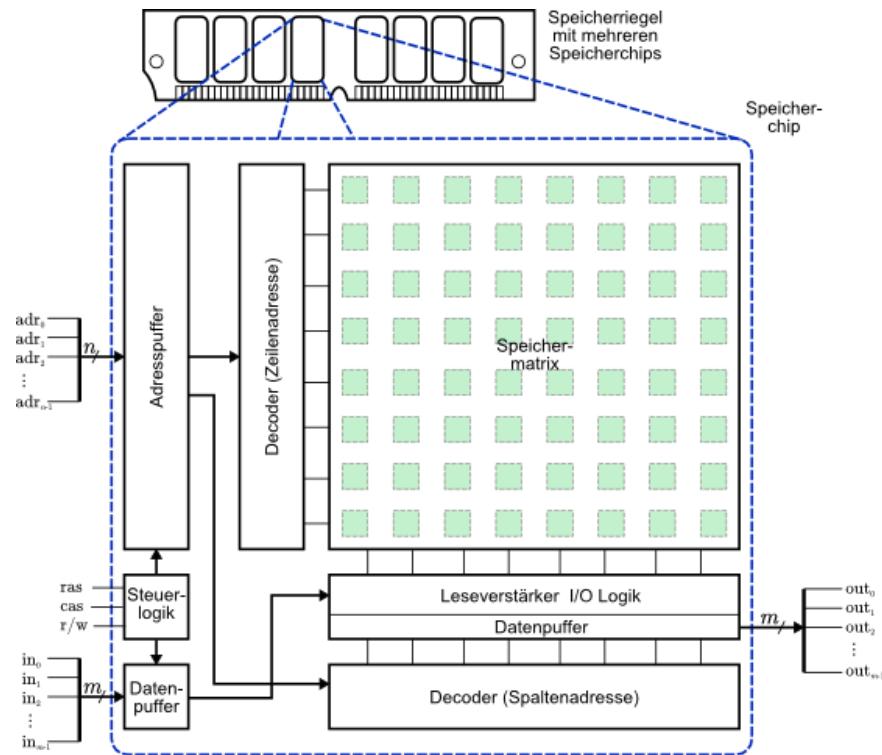
Thorsten Thormählen 14 / 32

DRAM Speichermatrix



Thorsten Thormählen 15 / 32

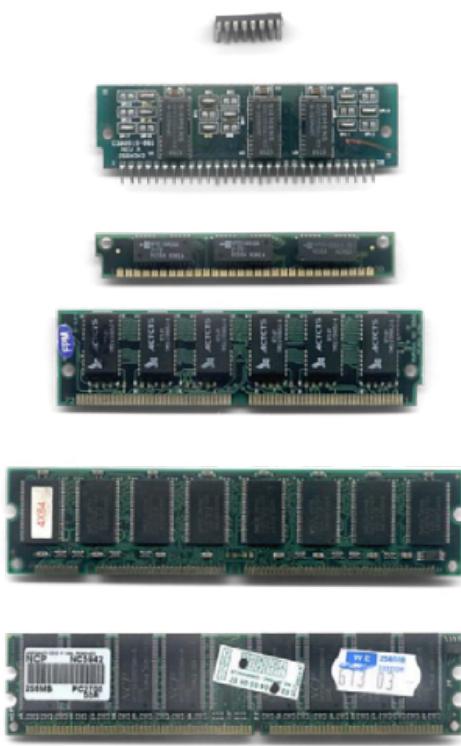
DRAM Speicherriegel mit mehreren Speicherchips



[Bildquelle: basierend auf: D. W. Hoffmann: *Grundlagen der Technischen Informatik*, 2. Auflage, Hanser 2009, Abb. 9.28, S. 332ff]

Thorsten Thormählen 16 / 32

Verschiedene Bauformen von DRAM Speicherriegeln



DIP

Dual In-Line

SIPP

Single In-Line Pin Package

SIMM 30 pin

Single Inline Memory Module

SIMM 72 pin

Single Inline Memory Module

DIMM (168-pin)

Dual Inline Memory Module

DDR DIMM (184-pin)

Dual Inline Memory Module

[Bildquelle:[Verschiedene RAM Typen](#) , Autor: Topory, [Creative Commons License](#)]

Thorsten Thormählen 17 / 32

DRAM Refresh

Der Kondensator in den Speicherzellen der DRAMs kann seine Ladung nicht sehr lange aufrechterhalten

Es gibt Leckströme über den Transistor, die zur Entladung führen

Ist der Kondensator zu stark entladen, kann nicht mehr festgestellt werden, ob eine logische 1 oder 0 gespeichert war

Daher werden alle Speicherzellen der Matrix periodisch "aufgefrischt" (engl. "refresh")

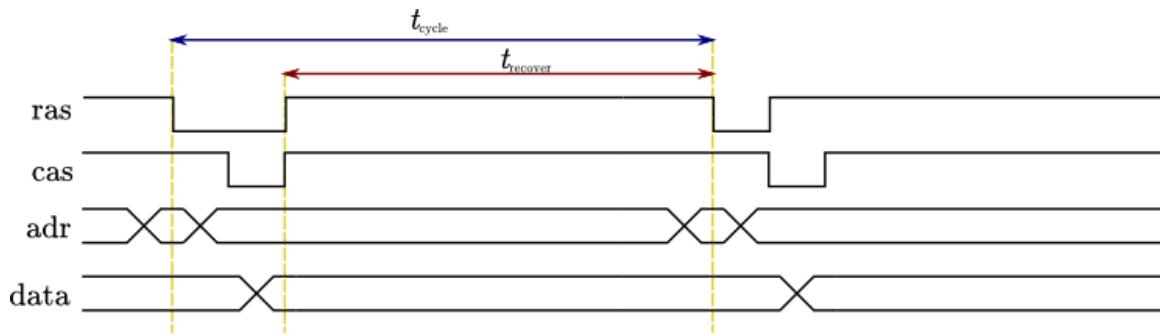
Eine typische Zeit bis zu einem Refresh sind 64 ms

Die Notwendigkeit, einen Refresh durchzuführen, gibt den DRAMs ihren Namen, da sie "dynamisch" betrieben werden müssen

Um ein Refresh durchzuführen, wird eine Zeile der Speichermatrix über den Leseverstärker in den Ausgangsdatenpuffer transferiert und anschließend wieder zurückgeschrieben

Frühere Chips benötigten dazu externe Ansteuerung, um die Adressen und Steuersignale zu generieren, heutige Chips haben dazu eigene Logik auf dem Chip, die den Refresh selbstständig durchführt

DRAM Zeitverhalten



Bei DRAM Bausteine wird häufig "Adressmultiplexing" betrieben

D.h. um Adressleitungen zu sparen, werden diese aufgeteilt und jeweils nur ein Teil der Adresse übertragen:

Ist $\text{ras}=0$, wird die Zeilenadresse gesendet ("Row Address Strobe")

Ist $\text{cas}=0$, wird die Spaltenadresse gesendet ("Column Address Strobe")

Die minimale Zeit t_{cycle} zwischen zwei Speicherzugriffen wird beim RAM maßgeblich geprägt durch die Erholzeit t_{recover} , die benötigt wird, um die Daten nach einem Zugriff wieder zurückzuschreiben, und die Bitleitungen für einen erneuten Zugriff vorzubereiten

DRAM Zeitverhalten

Um trotz langer Erholzeit $t_{recover}$ schnell Daten zu liefern, werden die Daten derart auf die einzelnen Speicherchips eines Speicherriegels verteilt, dass bei konsekutivem Zugriff jeweils andere Speicherchips Daten liefern müssen

Damit kann sich ein Chip in der Erholzeit befinden, während die anderen Chips des Speicherriegels Anfragen beantworten

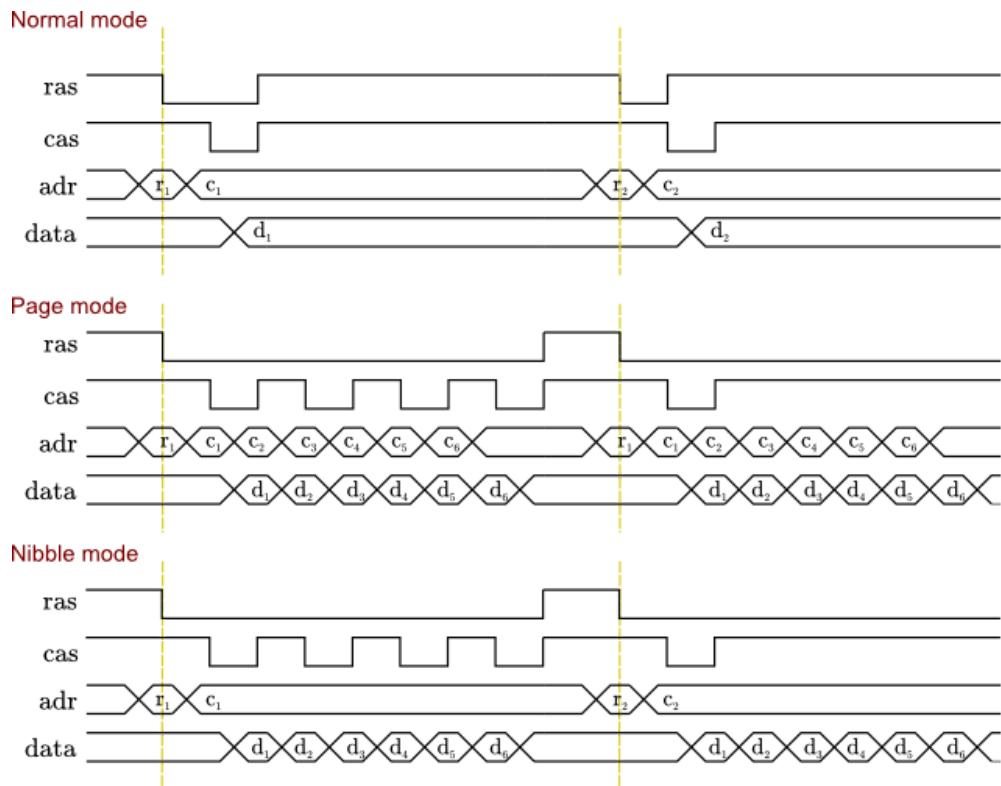
Des Weiteren gibt es den so genannten "Page" bzw. "Nipple" Modus, um direkt hintereinander liegende Speicherstellen auszulesen

Dies funktioniert allerdings nur, solange Daten mit der gleichen Zeilenadresse benötigt werden

Im "Page"-Mode können unterschiedliche Spaltenadressen übergeben werden

Im "Nibble"-Mode werden mit jeder fallenden Flanke vom cas-Signal konsekutive Daten der gleichen Zeile ausgegeben

DRAM Zeitverhalten



Thorsten Thormählen 21 / 32

DRAM Entwicklung

Jahr der Einführung	Größe	Preis pro MB	Zugriffszeit neue Zeile bzw. Spalte	Zugriffszeit gleiche Zeile
1980	64 KB	1500 USD	250 ns	150 ns
1983	256 KB	500 USD	185 ns	100 ns
1985	1 MB	200 USD	135 ns	40 ns
1989	4 MB	50 USD	110 ns	40 ns
1992	16 MB	15 USD	90 ns	30 ns
1996	64 MB	10 USD	60 ns	12 ns
1998	128 MB	4 USD	60 ns	10 ns
2000	256 MB	1 USD	55 ns	7 ns
2002	512 MB	0,25 USD	50 ns	5 ns
2004	1 GB	0,10 USD	45 ns	3 ns
2012	8 GB	0,01 USD	24 ns	0,25 ns

[Quelle: basierend auf: David A. Patterson, John L. Hennessy: *Computer Organization and Design*, 2005 by Elsevier Inc, S. 490, adaptiert gemäß [Memory Prices \(1957-2013\)](#) , John C. McCallum]

Thorsten Thormählen 22 / 32

Flash-Speicher

Flash-Speicher ist ein nicht-flüchtiger Datenspeicher, d.h. die Information bleibt auch ohne Stromversorgung erhalten

Im Gegensatz zu Festplatten oder optischen Laufwerken (CD, DVD, Blue-Ray) hat Flash-Speicher keine mechanisch-beweglichen Teile und kann damit in kleiner Bauform realisiert werden

Die Kosten pro Byte liegen allerdings höher als bei z.B. Festplatten

Anwendungen:

- Speicherkarten in mobilen Geräten (Kamera, Mobilfunkgerät, etc.)

- USB-Sticks

- Solid-State-Drive (kurz SSD), als schnelle Systemplatte in PCs und Laptops

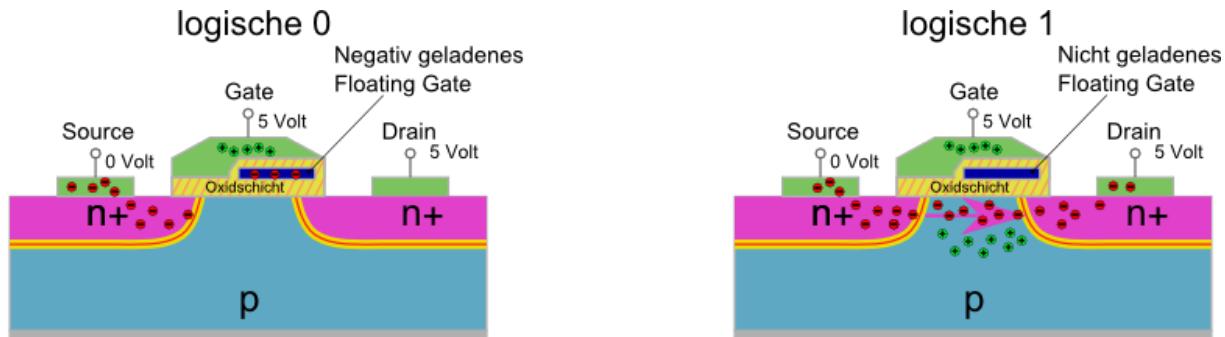
Flash-Speicher

Flash-Speicher basiert auf dem Floating-Gate-Transistor

Dies ist ein normaler Feldeffekttransistor mit zusätzlichem "Floating Gate"

Die elektrische Ladung auf dem Floating Gate kodiert, ob die Speicherzelle eine logische 0 oder 1 speichert

Das Floating Gate ist vom Dielektrikum umgeben, d.h. es behält die aufgebrachte Ladung permanent (> 1 Jahr)



Flash-Speicher

Lesen

Ist das Floating Gate negativ geladen, verhindert es, dass der Transistor normal betrieben werden kann = kein Strom fließt = Kodiert logische 0

Ist das Floating Gate nicht geladen, kann der Transistor normal arbeiten = Strom fließt = Kodiert logische 1

Schreiben

Logische 0: Es wird eine hohe positive Spannung (ca. 15V) angelegt. Elektronen "tunneln" aus dem Substrat auf das Floating Gate.

Logische 1: Es wird eine hohe negative Spannung (ca. -15V) angelegt. Elektronen "tunneln" vom Floating Gate in das Substrat.

Degeneration

Bei jedem Lese- oder Schreibvorgang wird die Oxidschicht leicht beeinträchtigt, so dass ein Transistor nach ca. 10.000 bis mehreren 100.000 Zyklen unbrauchbar wird.

Auf dem Chip gibt es Logik, die fehlerhafte Speicherzellen erkennt, sich deren Position in der Speichermatrix merkt, und die betroffene Speicherzelle durch andere ersetzt.

Festplattenlaufwerk

Eine Festplatte (engl. Hard Disk Drive, HDD) ist ein magnetisches Speichermedium für die nicht-flüchtige Speicherung großer Datenmengen (ca. 1 bis 4 TB in heutigen Standard-PCs)

Typischerweise besteht ein Laufwerk aus mehreren Platten, die auf einer gemeinsamen Spindel durch einen Elektromotor gedreht werden

Rotationsgeschwindigkeiten bewegen sich im Bereich von ca. 5.400 bis 15.000 Rotationen pro Minute

Auf einem gemeinsamen Actuatorarm befinden sich die Schreib-Lese-Magnetköpfe für die einzelnen Platten

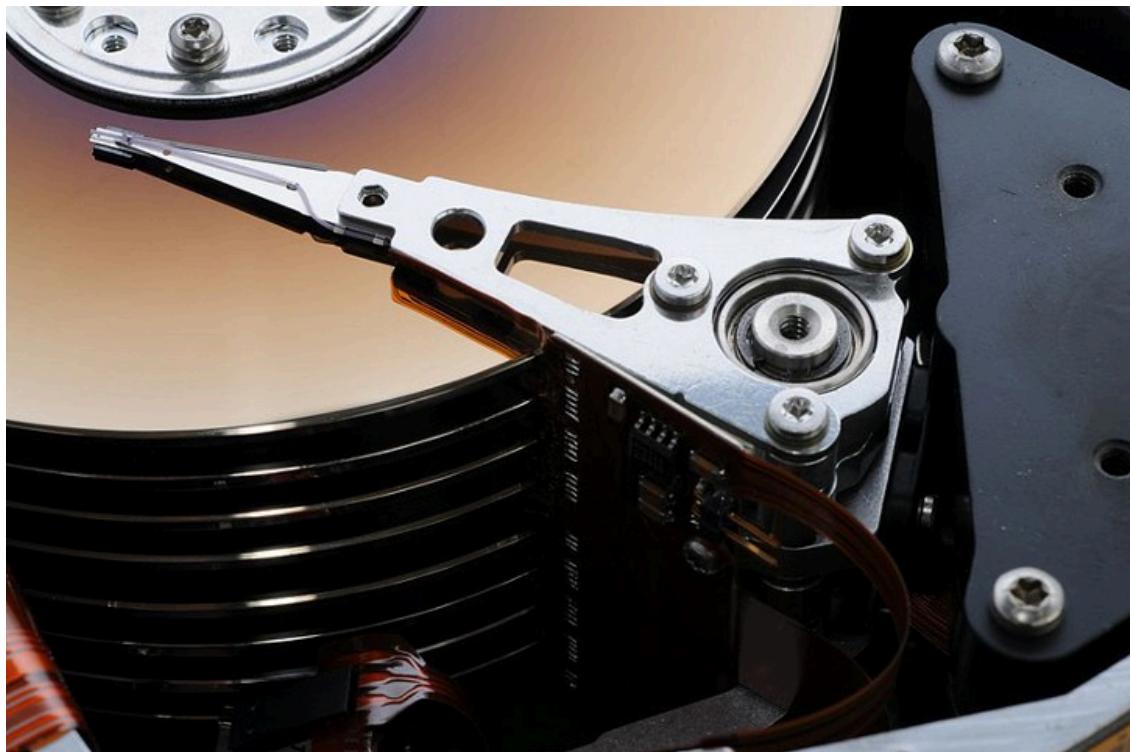
Die Magnetköpfe schweben mit einem sehr geringen Abstand von wenigen Nanometern über den Platten, die eine hart-magnetischen Beschichtung besitzen, und übertragen die magnetische Information während sich die Platte am Kopf vorbei dreht



[Bildquelle: An HDD, Autor: Ervins Strauhmanis, Creative Commons License]

Thorsten Thormählen 26 / 32

Festplattenlaufwerk



[Bildquelle: Festplatte Seagate ST4702N , Autor: Hubert Berberich, public domain]

Thorsten Thormählen 27 / 32

Festplattenlaufwerk

0:00 / 2:24

Video einer geöffneten Festplatte

[Quelle:[Hard Drive Operating Without Cover](#) , Autor: Nick Oakman]

Thorsten Thormählen 28 / 32

Festplattenlaufwerk

Daten werden blockweise adressiert. Ein Block hat z.B. 512, 2048 oder 4096 Bytes.

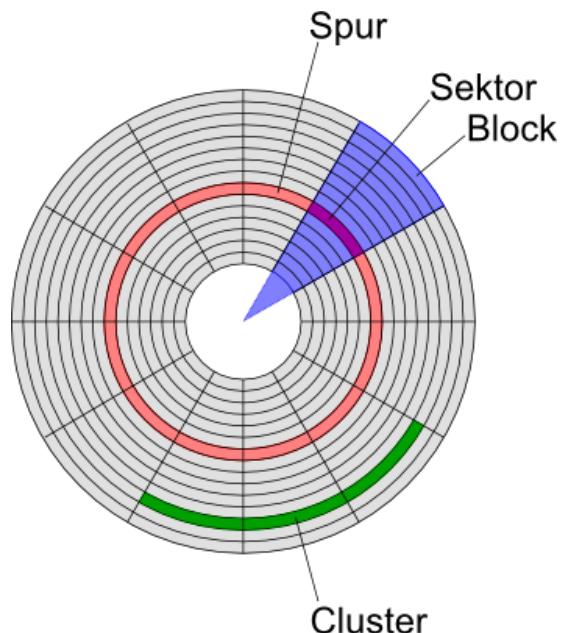
Eine Spur auf der Platte entspricht einer Kreisbahn, auf der der Magnetkopf läuft

Da typischerweise mehrere Magnetköpfe im Einsatz sind, werden die gemeinsam angesteuerten Spuren der verschiedenen Platten "Zylinder" genannt

Ein Block entspricht einem kleinen Winkelbereich auf einer Spur

Alle Blöcke mit dem gleichen Winkelbereich werden "Sektor" genannt

Cluster sind Gruppen von benachbarten Blöcken auf einer Spur, die gemeinsam angesprochen werden können



Festplattenlaufwerk

Platten können hohe kontinuierliche Übertragungsraten besitzen (heutzutage ca. 300 MB/s)

Allerdings ist die mittlere Zugriffszeit geringer (5-20 ms), da die Köpfe erst mechanisch an die richtige Stelle gebracht werden müssen und gewartet werden muss, bis der richtige Block vorbei rotiert

Die Blöcke enthalten nicht nur die eigentlichen Daten, sondern auch Zusatzinformationen für die Fehlererkennung

Aufgrund der langsamen Zugriffszeit von Festplatten, werden Daten typischerweise immer erst in den Hauptspeicher (DRAM) geladen und von dort aus vom Prozessor verarbeitet

Quiz

Frage: Eine 3,5 Zoll Festplatte dreht mit 7200 Umdrehungen pro Minute. Wie groß ist ca. die Geschwindigkeit auf der äußersten Spur?

Antwort 1: 120 km/h

Antwort 2: 1200 km/h

Antwort 3: 12000 km/h

Am Online-Quiz teilnehmen durch Besuch der Webseite:
www.onlineclicker.org

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) .

Thorsten Thormählen 32 / 32

Technische Informatik I

Mikrocode-basierter CPU-Simulator

Thorsten Thormählen
26. Januar 2021
Teil 9, Kapitel 1

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

CPU-Simulator

In diesem Kapitel wird anhand eines vereinfachten Modells die Funktionsweise einer CPU (Central Processing Unit) erläutert

Das Modell hat nur geringe Ähnlichkeit mit existierenden Prozessoren

Trotzdem ist es voll funktionsfähig

Dieses Modell-CPU entstand im Rahmen von Vorlesungen, die Peter Gumm und Manfred Sommer hier in Marburg gehalten haben

Von Martin Perner wurde ein entsprechendes Simulationsprogramm namens MikroSim erstellt, das lange in dieser Vorlesung eingesetzt wurde: [MikroSim](#)

Das Modell und Programm wurden 1995 auf der CeBit vorgestellt:

H. Peter Gumm, Martin Perner: *Der Mikrocodesimulator MicroSim*. CeBit 1995

Seit WS 2013/14 werden in der Vorlesung und der Übung eine alternative Implementierung in Form einer Web-Applikation eingesetzt: [CPU-Simulator](#)

CPU-Simulator

Die wichtigsten Einzelteile, aus denen eine CPU aufgebaut ist, wurden bereits besprochen:

ALU (Arithmetic Logic Unit)

Register

Speicher

Diese Komponenten sind durch Leitungen verbunden, die durch steuerbare Schalter geöffnet oder geschlossen werden können.

Das Öffnen und Schließen dieser Schalter muss in einer zeitlichen Abfolge koordiniert werden

Daher besitzt eine CPU zunächst einen Taktgeber, der die Zeit in einzelne Takte unterteilt

Diese Takte sind sehr kurz, bei einem 1-GHz-Prozessor dauert ein Takt 10^{-9} s, also eine Nanosekunde

In den folgenden Folien wird eine sehr einfache, idealisierte Definition eines CPU-Taktes beschrieben

Was genau in einem Takt passiert kann bei realen Prozessoren sehr unterschiedlich sein

Takt und Takt-Phasen

Jede Operation der CPU benötigt einen Takt

Für eine einfache Operation, wie etwa die Addition zweier Registerinhalte, werden dazu drei Phasen benötigt:

Phase 1: Hol-Phase (engl. "fetch")

Holt die Argumente aus den Registern und stelle sie der ALU bereit

Phase 2: Rechenphase (engl. "execute")

Führt die ALU-Operation durch

Phase 3: Bring-Phase (engl. "store")

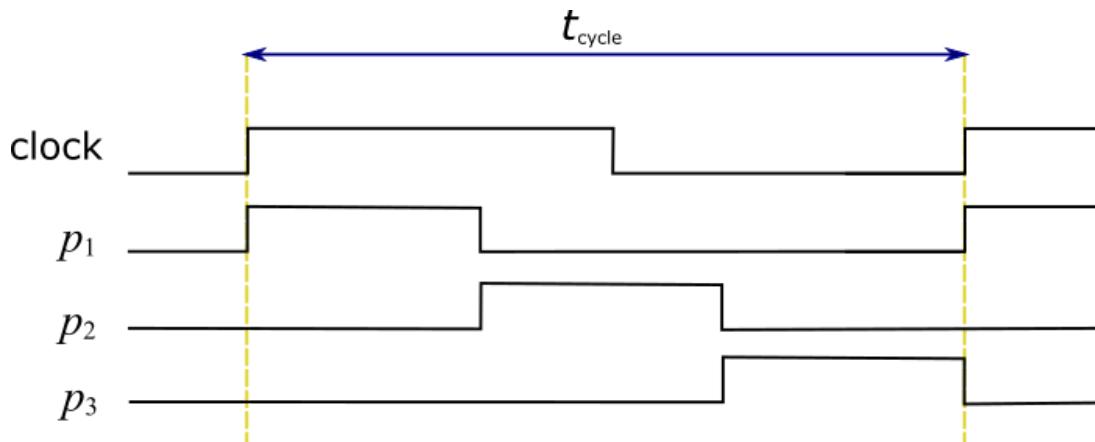
Speichert das Ergebnis in ein Register

Takt-Phasen

Bei unserer Modell-CPU müssen für jede dieser Phasen gewisse Schalter geöffnet, andere wieder geschlossen werden

Daher arbeitet die CPU intern mit drei Phasen-Steuersignalen p_1, p_2, p_3 , die abwechselnd auf 1, dann wieder auf 0 gesetzt werden

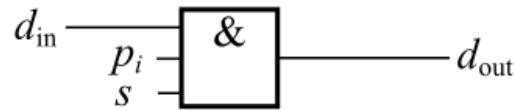
In Phase i ist $p_i = 1$, alle anderen $p_{j \neq i} = 0$



Steuerbare Schalter

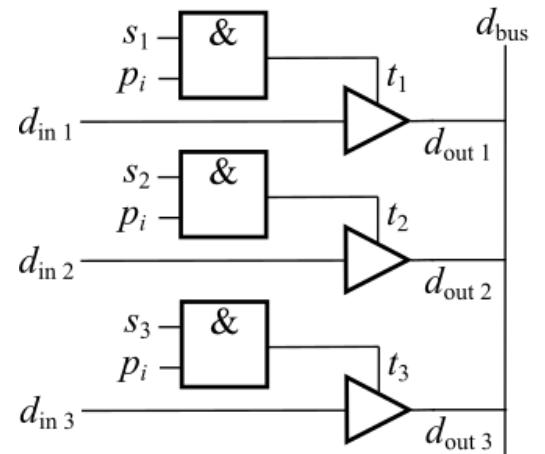
Damit die Datenleitungen zur richtigen Zeit offen bzw. geschlossen sind, werden sie durch Schalter gesichert, die nur für eine bestimmte Phase geöffnet werden können

Ein solcher Schalter könnte z.B. durch ein UND-Gatter mit 3 Eingängen realisiert werden



Dieser Schalter würde nur bei gesetztem Steuersignal $s = 1$ und nur in Phase i (d.h. $p_i = 1$) die Daten vom Eingang d_{in} an den Ausgang d_{out} weiterleiten

In der Praxis werden zum Ansteuern von Datenleitungen auch gerne so genannte Tristate Buffer eingesetzt, die eine Datenleitung bei $t = 0$ hochohmig schalten. D.h. es gibt 3 Zustände am Ausgang: 0, 1 und hochohmig.



Mikrobefehl

In unserer beispielhaften Modell-CPU finden sich zahlreiche der eben beschriebenen Schalter

Diese bleiben für einen Takt lang in einer bestimmten Einstellung und sind ggf. in bestimmten Phasen des Taktes offen

Die Stellung eines Schalters wird durch sein Steuersignal s_j definiert

In einem so genannten Mikrobefehlswort oder Mikrobefehl wird die Stellung vieler Schalter bzw. Steuersignale zusammengefasst

Eine Folge von Mikrobefehlen wird auch als Mikrocode bezeichnet

Ein Mikrobefehlswort bleibt einen Takt lang gültig und steuert in diesem Takt die Arbeitsweise der CPU

In unserer Modell-CPU besteht jedes Mikrobefehlswort aus genau 48 Bits:

s_1, s_2, \dots, s_{48}

Die Bedeutung dieser 48 Bits werden auf den folgenden Folien schrittweise erläutert

Microcode																																													
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MC	MCNext	CC	ALU-FC																																										

Register und Busse

Register enthalten Datenworte, d.h. aus mehreren Bits bestehende Daten

In unserer Modell-CPU werden Register als farbige Rechtecke dargestellt

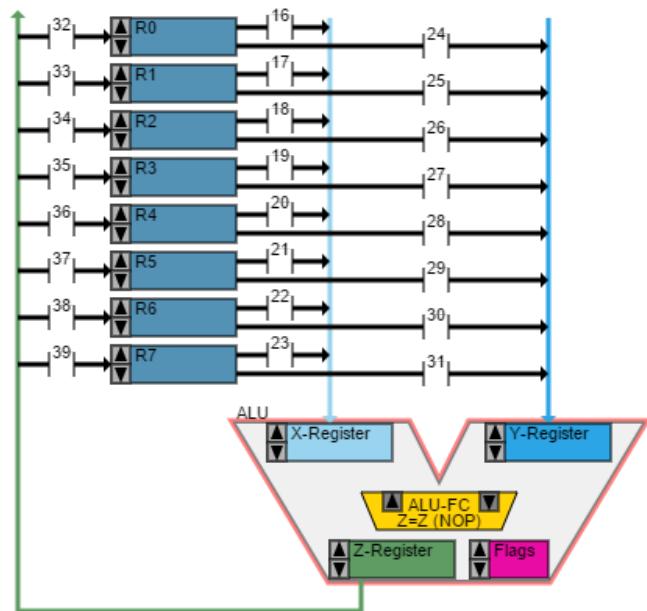
Wir beschreiben eine 16-Bit-Architektur, so dass alle Register 16 Bit breit sind

Der Registerinhalt wird jeweils über eine 4-stellige Hexadezimal-Zahl dargestellt

Datenleitungen verbinden Register miteinander

Bei einem Registertransfer werden die entsprechenden Bits von Quell- und Zielregister durch parallele Leitungen (Datenbus) verbunden

In unserer Modell-CPU wird der Datenbus zwischen zwei Registern als eine etwas dickere Linie mit einem Pfeil dargestellt, der die Richtung des Datentransfers angeigt



Bus-Schalter

Bus-Schalter schalten alle Datenleitungen für ein Register gemeinsam (16 parallele Schalter mit einem gemeinsamen Steuersignal)

Ein geöffneter Bus-Schalter wird durch zwei dünnere parallel Linien dargestellt

Bei einem geschlossenem Bus-Schalter sind die dünneren parallelen Linien mit einem Bus verbunden

Die Bus-Schalter bleiben für einen Takt lang in einer bestimmten Einstellung und sind ggf. in bestimmten Phasen des Taktes offen

Die Stellung eines Schalters wird durch sein Steuersignal s_j definiert, dabei ist der Index j jeweils als Zahl leicht oberhalb des Schalters zu finden

Die kleine Abbildung rechts zeigt den Bus-Schalter, der dem Steuerbit s_{32} des Mikrobefehlsworts gehorcht

Links geöffnet $s_{32} = 0$, rechts geschlossen $s_{32} = 1$



Selbst ausprobieren:

Öffnen des [CPU-Simulators](#)

Klicken auf den Schalter 32 ändert dessen Schalterstellung und Steuerbit s_{32}

Klicken auf ein Steuerbit ändert ebenfalls die Schalterstellung

CPU-Simulator: Komplexitätsstufe 1

Thorsten Thormählen 12 / 69

CPU-Simulator: Komplexitätsstufe 1

Nun wurden alle Bauelemente beschrieben, um einen Taschenrechner mit einigen Speicherzellen (Registern) zu bauen

Wir benötigen dazu zunächst eine ALU, eine Reihe von Registern (hier R_0, R_1, \dots, R_7), Busse und Bus-Schalter

Die ALU versehen wir mit zwei Operandenregistern, x und y, sowie einem Ergebnisregister z

Dann verbinden wir jedes Allzweckregister R_0, R_1, \dots, R_7 über zwei Busse (dem X-Bus und dem Y-Bus) mit den entsprechenden Operandenregistern der ALU

Das Ergebnisregister z der ALU wird über den Z-Bus mit den Allzweckregistern verbunden

CPU-Simulator: Komplexitätsstufe 1

Zwischen den Registern und den Bussen sitzen Schalter, die nur in bestimmten Phasen geöffnet werden können (`fetch`, `execute`, `store`).

Die Schalter zwischen den Allzweckregistern und dem X- und Y-Bus sind nur in der Hol-Phase `fetch` aktiv

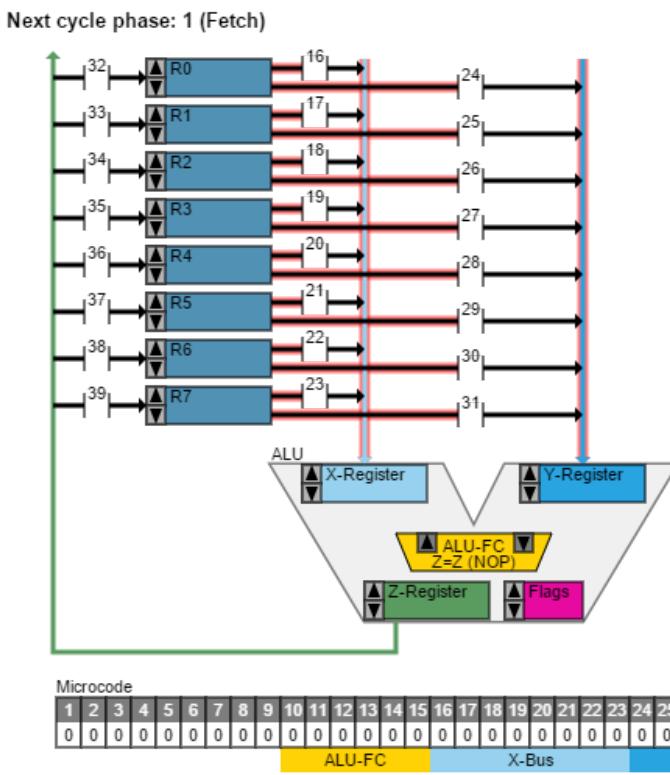
Nur dann können die Daten von den Registern zu den Operandenregistern der ALU fließen

In der zweiten Phase `execute` rechnet die ALU und schreibt das Ergebnis in `z`

Nur in dieser Bring-Phase `store` sind die Schalter zwischen Z-Bus und den Allzweckregistern aktiv, damit das Ergebnis in einem der Register abgelegt werden kann

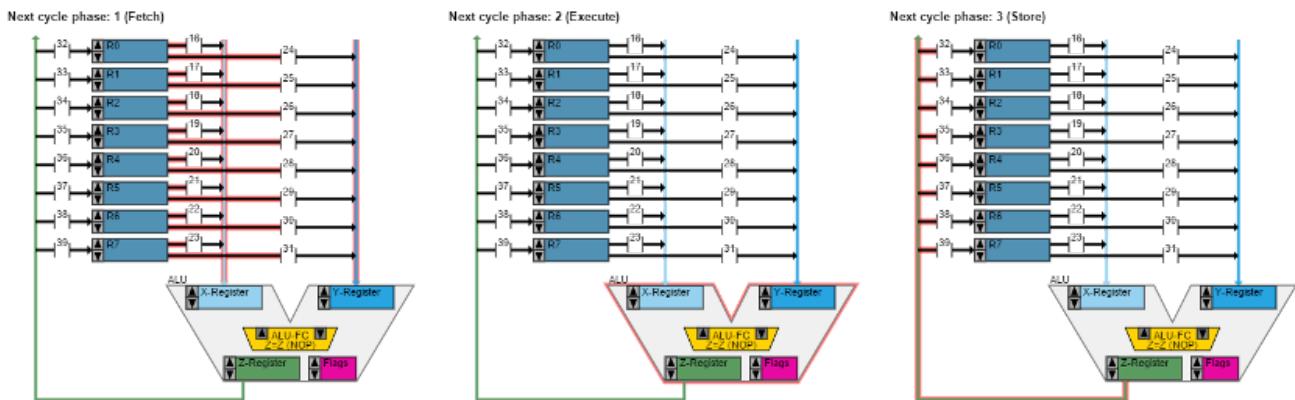
Der komplette Aufbau ist in der Abbildung auf der nächsten Folie dargestellt

CPU-Simulator: Komplexitätsstufe 1



Thorsten Thormählen 15 / 69

CPU-Simulator: Komplexitätsstufe 1



Um zu verdeutlichen, in welcher Phase (fetch, execute, store) welche Komponenten aktiv sind, werden diese bei Aktivität jeweils rot unterlegt

Selbst ausprobieren:

Öffnen des [CPU-Simulators](#)

Klicken auf den Druckknopf "Next" aktiviert die nächste Phase

Mikrocodegesteuerte Operationen

Schließlich muss noch an der ALU einstellt werden, welche Operation sie berechnen soll

Wir nehmen an, dass unsere ALU ein Repertoire von 64 Operationen umfasst, so dass wir die Operation mit 6 Bit im Mikrobefehlswort einstellen können

Diese 6 Bit steuern einen Multiplexer (dargestellt als symmetrisches Trapez) in der ALU, der die Operation auswählt

Neben den grundlegenden arithmetischen, logischen und vergleichenden Operationen sind auch Operationen mit Konstanten möglich

Diese dienen z.B. dazu, eine feste Konstante im Z-Register bereitzustellen

Die ALU-Funktionscodes (ALU-FC) sind in den folgenden Tabellen dargestellt

In der Operations-Spalte wird angegeben, wie sich der Wert im Z-Register aus den Werten in den X- und Y-Registern ergibt

Ggf. werden noch die Inhalte von X- und Y-Register einander zugewiesen bzw. vertauscht, was durch $y \rightarrow x$ bzw. durch $x \leftrightarrow y$ angedeutet wird

ALU Operationen

ALU-FC (Dezimal)	ALU-FC (Binär)	Operation
0	000000	$Z = Z$ (Keine Operation)
1	000001	$Z = -Z$
2	000010	$Z = X$
3	000011	$Z = -X$
4	000100	$Z = Y$
5	000101	$Z = -Y$
6	000110	$Z = Y, X \leftrightarrow Y$
7	000111	$Z = X, X \leftrightarrow Y$
8	001000	$Z = X, Y \rightarrow X$
9	001001	$Z = X + 1$
10	001010	$Z = X - 1$
11	001011	$Z = X + Y$
12	001100	$Z = X - Y$
13	001101	$Z = X * Y$
14	001110	$Z = X \text{ div } Y$
15	001111	$Z = X \text{ mod } Y$

ALU Operationen

ALU-FC (Dezimal)	ALU-FC (Binär)	Operation
16	010000	$Z = X \text{ sal } Y$ (<u>shift arithmetic left</u>)
17	010001	$Z = X \text{ sar } Y$ (<u>shift arithmetic right</u>)
18	010010	$Z = X \text{ cmpa } Y$ (compare arithmetic)
19	010011	$Z = X \text{ and } Y$
20	010100	$Z = X \text{ nand } Y$
21	010101	$Z = X \text{ or } Y$
22	010110	$Z = X \text{ nor } Y$
23	010111	$Z = X \text{ xor } Y$
24	011000	$Z = X \text{ nxor } Y$
25	011001	$Z = X \text{ sll } Y$ (shift logic left)
26	011010	$Z = X \text{ slr } Y$ (shift logic right)
27	011011	$Z = X \text{ cpl } Y$ (compare logic)
28	011100	$X = 0$
29	011101	$X = (\text{FFFF})_{16}$
30	011110	$Y = 0$
31	011111	$Y = (\text{FFFF})_{16}$

Thorsten Thormählen 19 / 69

ALU Operationen

ALU-FC	Operation
Falls $32 \leq \text{ALU-FC} < 48$	$Z = X = \text{ALU-FC} - 32$
Falls $48 \leq \text{ALU-FC} < 64$	$Z = Y = \text{ALU-FC} - 48$

Beispiele:

Für $\text{ALU-FC} = 35$ ergibt sich $Z = X = 3$

Für $\text{ALU-FC} = 63$ ergibt sich $Z = Y = 15$

Statusregister (Flags)

Führt die ALU eine Berechnung durch, wird nicht nur das Ergebnis im Z-Register abgelegt, sondern auch das Flag-Register gesetzt

Die ALU signalisiert damit, dass bestimmte Ereignisse bei der Berechnung aufgetreten sind

In unserer Modell-CPU sind dies 4 verschiedene, daher ist das Flag-Register 4 Bit breit:

Bit 0: Overflow einer arithmetischen Operation (overflow flag)

Bit 1: Ergebnis war negativ (sign neg flag)

Bit 2: Ergebnis war positiv (sign pos flag)

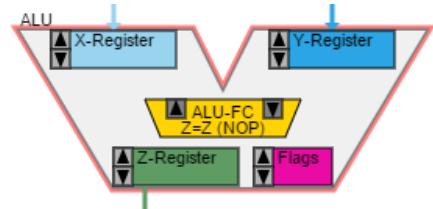
Bit 3: Ergebnis war null (zero flag)

Der Inhalt des Flag-Registers wird als Hexadezimal-Zahl dargestellt

Beispiele:

Flag = $(5)_{16} = (0101)_2$, d.h. ein positives Ergebnis und Overflow sind aufgetreten

Flag = $(8)_{16} = (1000)_2$, d.h. das Ergebnis ist null



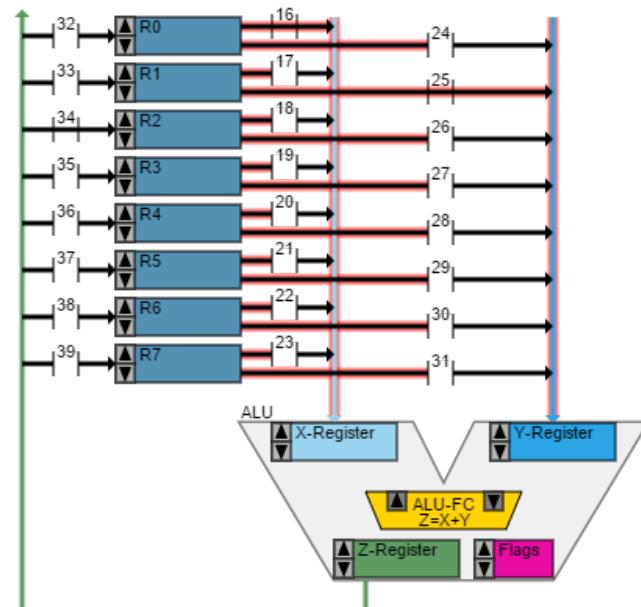
Beispiel: Addition zweier Zahlen

Als Beispiel für die Verwendung der Modell-CPU soll die Addition zweier Register und die Speicherung des Ergebnisses betrachtet werden:

$$R_3 = R_1 + R_2$$

Dazu muss ALU-FC Nr. 11 = $(001011)_2$ eingestellt sein

Die Steuersignale für den X-Bus sind 10000000, für den Y-Bus 01000000 und für den Z-Bus 00100000



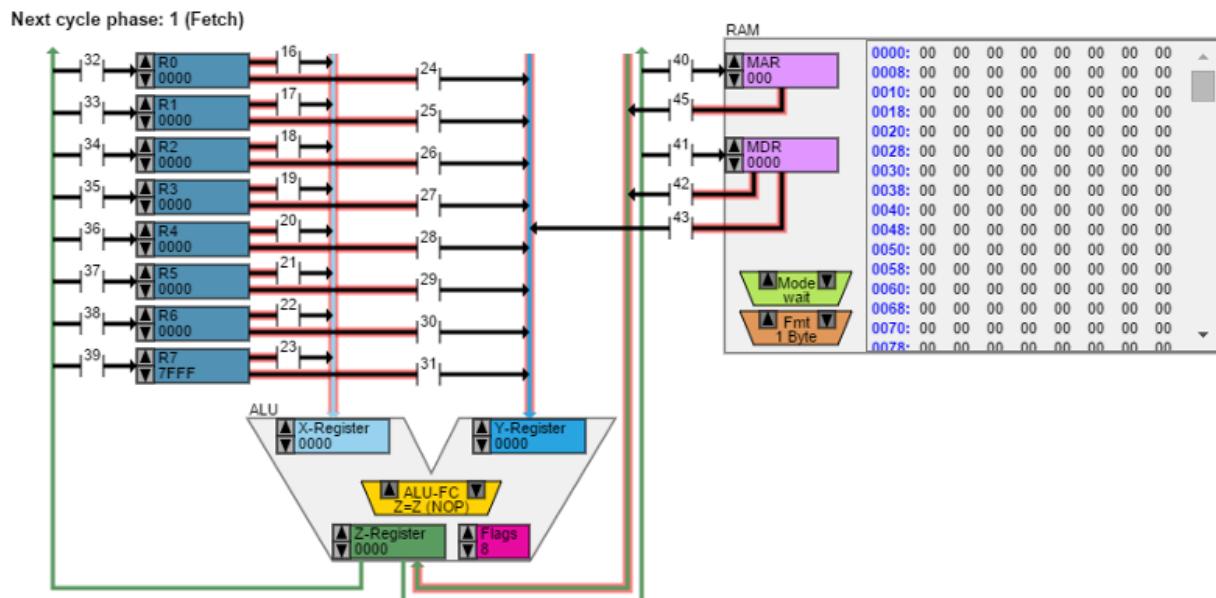
Microcode			
1	2	3	4
0	0	0	0
5	6	7	8
0	0	0	0
9	10	11	12
0	0	0	1
13	14	15	16
0	1	1	1
17	18	19	20
0	0	0	0
21	22	23	24
0	0	0	0
25	26	27	28
0	1	0	0
29	30	31	32
0	0	0	0
33	34	35	36
0	0	1	0
37	38	39	40
0	0	0	0
41	42	43	44
0	0	0	0
45	46	47	48
0	0	0	0

CPU-Simulator: Komplexitätsstufe 2

Thorsten Thormählen 23 / 69

CPU-Simulator: Komplexitätsstufe 2

Die CPU wird nun um einen Hauptspeicher (RAM) erweitert



Thorsten Thormählen 24 / 69

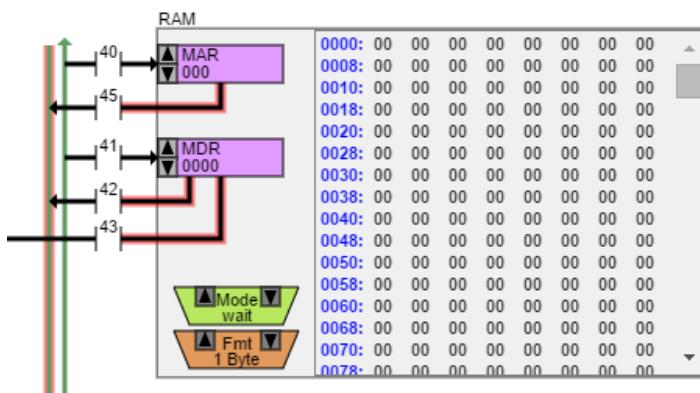
Hauptspeicher

Der Hauptspeicher (engl. Random Access Memory kurz RAM) hat als Schnittstelle zwei Register, eine Formatangabe und einen einstellbaren Modus

Bei den Registern handelt es sich um das Adressregister (**MAR** = Memory Address Register) und das Datenregister (**MDR** = Memory Data Register)

Wie bereits im Kapitel über Speicher besprochen, steht im Adressregister eine Speicheradresse und im Datenregister ein Wert, der an der angegebenen Adresse geschrieben werden soll oder von der angegebenen Adresse gelesen wurde

Insgesamt umfasst der RAM Speicher 1024 Bytes, $(000)_{16}$ bis $(3FF)_{16}$



Mode und Format

Um das RAM per Mikrocode anzusteuern, wird das Mikrobefehlswort erweitert

Microcode	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48		
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Die Bits 40 bis 45 (I/O RAM) steuern die Bus-Schalter der RAM-Register

Die Bits 46 und 47 (Mode) steuern den I/O Mode:

00 = wartend

01 = lesend

10 = schreibend

Und Bit 48 (Fmt), wählt das Datenformat (8 oder 16 Bit Lesen/Schreiben)

0 = 1 Byte Lesen/Schreiben

1 = 2 Byte Lesen/Schreiben

Beim Schreiben einer 8-Bit-Größe wird nur das niederwertige Byte aus dem MDR an die Adresse [MAR] geschrieben.

Beim Lesen werden die vorderen 8 Stellen des MDR durch Nullen aufgefüllt

Beim 16 Bit Lesen/Schreiben wird das Big-Endian-Format verwendet

Schreiben in den Hauptspeicher

Es werden in der Regel zwei Takte benötigt

Die Adresse im Hauptspeicher, an die geschrieben werden soll, wird meist mit Hilfe der ALU berechnet

Im ersten Takt muss die Adresse, an die geschrieben werden soll, in Phase 3 über den Z-Bus in das Adressregister `MAR` transportiert werden

Im zweiten Takt muss der Datenwert, der in den Speicher geschrieben werden soll, in Phase 3 über den Z-Bus in das Datenregister `MDR` transportiert werden

Nach Takt 2, Phase 3 hat der Hauptspeicher die Schreiboperation durchgeführt

Selbst ausprobieren:

Öffnen des [CPU-Simulators](#)

In der Combo-Box "Import example file" das Beispiel "Write to RAM" auswählen

Das Beispiel schreibt $(000F)_{16}$ an die Speicherstelle $(008)_{16}$

Lesen vom Hauptspeicher

Es werden in der Regel zwei Takte benötigt

Im ersten Takt muss die Adresse, von der gelesen werden soll, in Phase 3 über den Z-Bus in das Adressregister `MAR` transportiert werden

Im zweiten Takt, Phase 1 steht der gelesene Wert im Datenregister `MDR` zur Verfügung

Die Daten des `MDR` können wahlweise über den Y- bzw. über den Z-Bus in das Y- und/oder in das Z-Register übertragen werden

Vom Z-Register aus können sie in Takt 2, Phase 3 in eines der Register geschrieben werden

Vom Y-Register aus können sie in Takt 2, Phase 2 in der ALU weiterverarbeitet werden

Selbst ausprobieren:

Öffnen des [CPU-Simulators](#)

In der Combo-Box "Import example file" das Beispiel "Read from RAM" auswählen

Das Beispiel liest $(0102)_{16}$ von der Speicherstelle $(010)_{16}$

Idealisierter Hauptspeicher

Der in unserer Modell-CPU verwendete Hauptspeicher liest und schreibt ohne Verzögerung

Die ist eine idealisierte Annahme

In der Realität kommt es häufig vor, dass der Speicher für die CPU zu langsam ist

Die CPU muss ggf. ein oder mehrere Takte warten (waitstates), um dem Speicher Zeit zu geben, die Daten zu lesen bzw. zu schreiben

Die Anzahl der benötigten Warte-Takte kann sehr unterschiedlich sein, je nachdem ob die Daten aus einem schnellen oder langsamen Cache kommen oder womöglich sogar aus dem noch langsameren Hauptspeicher

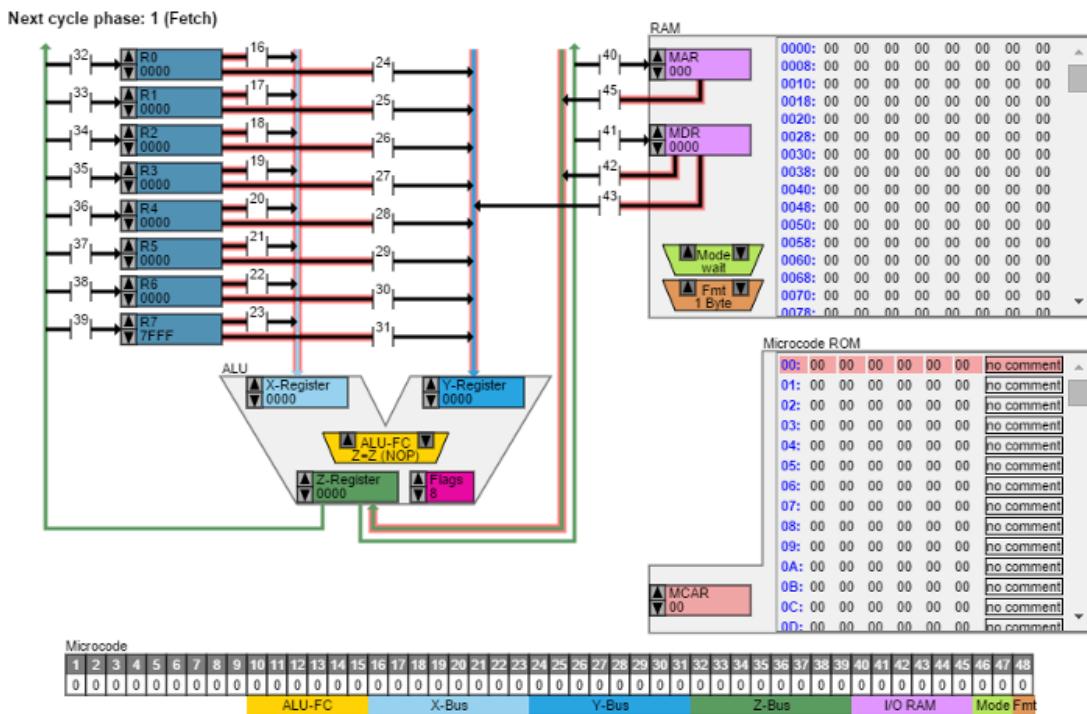
Während der Warte-Takte kann die CPU jedoch (falls möglich) andere unabhängige Berechnungen ausführen

CPU-Simulator: Komplexitätsstufe 3

Thorsten Thormählen 30 / 69

CPU-Simulator: Komplexitätsstufe 3

Die CPU wird nun um einen Mikrobefehlsspeicher erweitert



Thorsten Thormählen 31 / 69

Der Mikrobefehlsspeicher

Mikrobefehle sind Bitfolgen, die wie andere Daten auch in einem Speicher abgelegt werden können

Ein solcher Mikrobefehlsspeicher ist Teil der CPU

Der Speicher ist als ROM (Read-Only-Memory) ausgeführt, d.h. er kann nur gelesen, nicht aber von der CPU nicht verändert werden

Ansonsten ist das ROM wie jeder andere Speicher aufgebaut, insbesondere besitzt es ein Adressregister, in dem die Adresse eines Speicherwertes abgelegt wird, und ein Datenregister, in dem der dort befindliche Datenwert zurückgegeben wird

Weil die im ROM gespeicherten Daten als Mikrocode interpretiert werden, wird das Adressregister mit MCAR (MicroCode Address Register) bezeichnet

Das Datenregister ist im Simulator nicht als farbiges Rechteck dargestellt, da dessen Inhalt immer dem aktuellen Mikrocode entspricht, dessen 48 Bits sowieso jeweils unten dargestellt werden

Der Mikrobefehlsspeicher

Da in unserem CPU-Modell bis zu 256 Mikrobefehle im ROM speichern werden können, wird ein 8 Bit breites MCAR benötigt

Da jeder Mikrobefehl 48 Bits lang ist, wird das auf der CPU befindliche ROM eine Größe von $256 \times 48 \text{ Bit} = 1536 \text{ Bytes}$ besitzen

Wir können aber nicht jedes Byte adressieren, wie im RAM, sondern nur jeden Mikrobefehl, d.h. jeweils Worte mit 48 Bit

Ändert sich das Mikrocode-Adressregister wird das entsprechende Mikrocodewort gelesen und dessen Bits beeinflussen sofort als Steuersignale die Bus-Schalter und Multiplexer der restlichen CPU

Mit dem ROM kann die CPU nun komplexere Operationen über mehrere Takte ausführen, indem nach jedem Takt ein anderes Mikrocodewort aus dem ROM-Speicher angewendet wird

Zur Zeit muss das Mikrocode-Adressregister dazu noch von Hand erhöht werden

Der Mikrobefehlsspeicher

Beispiel: [5,6] = R1 + [5,6]

Takt 1	Phase 1	keine Operation	
	Phase 2	$Z = 5$	ALU-FC: 100101
	Phase 3	$MAR = Z$	I/O RAM: 100000 Mode: 01, Format: 1
Takt 2	Phase 1	RAM liefert Inhalt von [5,6] ins MDR	Mode: 01, Format: 1
	Phase 2	keine Operation	ALU-FC: 000000
	Phase 3	keine Operation	I/O RAM: 000000
Takt 3	Phase 1	$Y = MDR, X = R1$	X-Bus: 01000000
	Phase 2	$Z = X + Y$	ALU-FC: 001011
	Phase 3	$MDR = Z, RAM \text{ schreibt}$	I/O RAM: 010100 Mode: 10, Format: 1

Selbst ausprobieren:

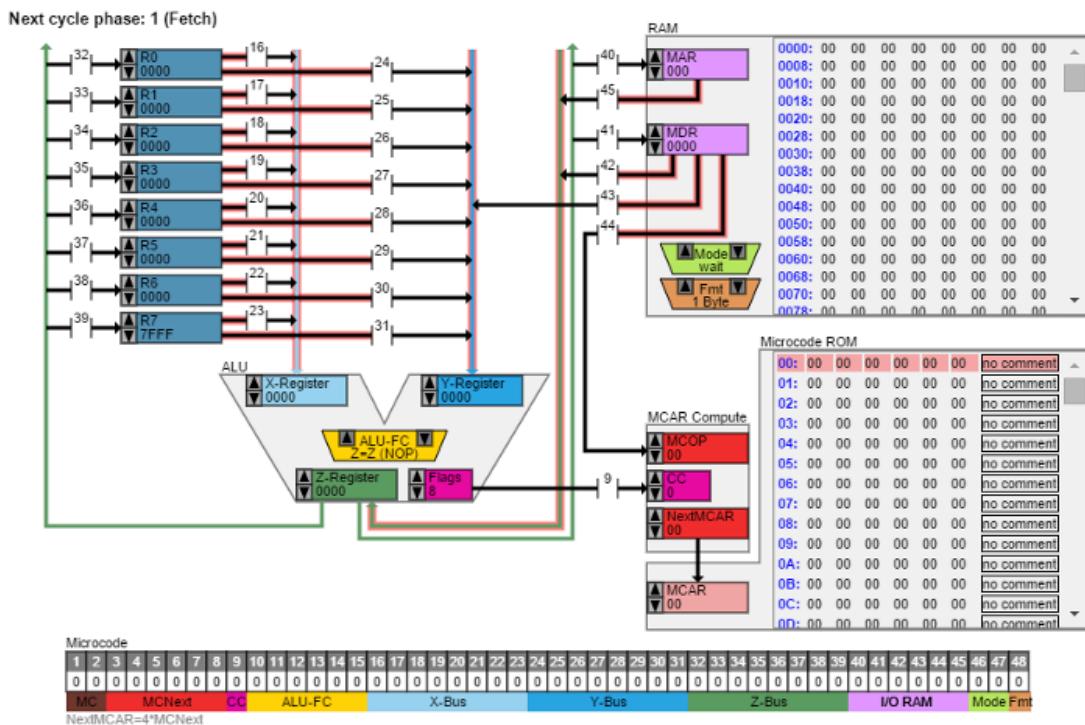
Im [CPU-Simulator](#) das Beispiel "Modify RAM: [5] = R1 + [5]" auswählen

CPU-Simulator: Komplexitätsstufe 4

Thorsten Thormählen 35 / 69

CPU-Simulator: Komplexitätsstufe 4

Die CPU wird nun um einen Adressrechner für den Mikrobefehlsspeicher erweitert



Thorsten Thormählen 36 / 69

Sprünge im Mikrobefehlsspeicher

Die im ROM befindlichen Befehle könnten von der CPU der Reihe nach abgearbeitet werden

Das wäre aber sehr eintönig und sinnlos, denn dann würde immer dasselbe Programm ablaufen, da der Inhalt des ROM ja unveränderbar ist

Daher ist die Modell-CPU so konstruiert, dass wir jeweils beliebige Mikrocode-Adressen in das MCAR schreiben können, um somit beliebige Sprünge im Mikrocode zu realisieren

Während die CPU den gegenwärtigen Mikrobefehl noch bearbeitet, berechnet die Adressberechnungseinheit "MCAR Compute" daraus die Adresse des nächsten Mikrobefehls im ROM und speichert das Ergebnis im NextMCAR Register

Der Wert von NextMCAR wird dann kurz vor Beginn des nächsten Taktes in das MCAR Register übertragen

Der nächste auszuführende Mikrobefehl wird dann aus dem ROM gelesen, liegt zu Beginn des nächsten Taktes vor und steuert die CPU

Für die Adressberechnungseinheit muss die Hardware der CPU um eine zusätzliche spezialisierte ALU erweitert werden. Sie wurde im CPU-Simulator aus Gründen der Übersichtlichkeit nicht explizit als ALU dargestellt.

Sprünge im Mikrobefehlsspeicher

Es werden folgende Fälle unterschieden:

Das Sprungziel steht von vornherein fest

Das Sprungziel ergibt sich als Wert einer Berechnung mit einer speziellen ALU

Das Sprungziel ergibt sich indirekt aus dem Inhalt des Speichers

Das Sprungziel ergibt sich aufgrund einer Bedingung, die aus dem Flag-Register der CPU-ALU ablesbar ist

Sprünge im Mikrobefehlsspeicher

Bits 1 bis 8 des Mikrobefehlwortes bestimmen, welcher Befehl als nächster auszuführen ist

Bit 1 und 2 werden mit MC bezeichnet

MC=00: absoluter Sprung: NextMCAR = 4 * MCNext

MC=01: relativer Sprung vorwärts: NextMCAR = MCAR + 1 + 4 * MCNext

MC=10: relativer Sprung rückwärts: NextMCAR = MCAR + 1 - 4 * MCNext

MC=11: OpCode-gesteuerte und bedingte Sprünge: (später)

Bits 3 bis 8 werden mit MCNext bezeichnet und kodieren das Sprungziel.

Mit 6 Bits erhält man $2^6=64$ mögliche Werte

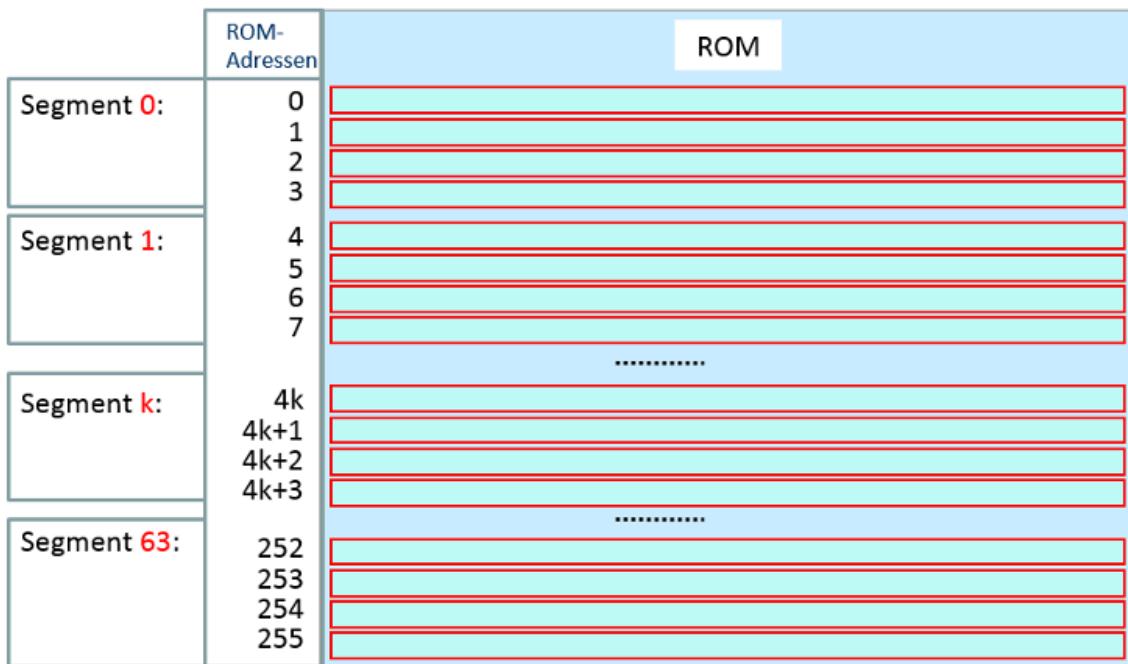
Um diese etwas besser über den Speicher zu verteilen, wird MCNext mit 4 multipliziert

Auf diese Weise ergibt sich eine logische Gruppierung von je 4 aufeinander folgenden Mikrocodeadressen zu einem Segment

Thorsten Thormählen 39 / 69

Absolute Sprünge

Je 4 ROM-Adressen werden zu einem Segment zusammengefasst
Absolute Sprünge ($MC=00$) sind nur an den Segmentanfang möglich



Relative Sprünge

Im Sprungmodus $MC=01$ wird ein Vorwärtssprung relativ zum gegenwärtigen MCAR ausgeführt. Die Adresse des neuen Befehls ergibt sich aus:

$$\text{NextMCAR} = \text{MCAR} + 1 + 4 * \text{MCNext}$$

Im Sprungmodus $MC=10$ wird ein Rückwärtssprung relativ zum gegenwärtigen MCAR ausgeführt. Die Adresse des neuen Befehls ergibt sich aus:

$$\text{NextMCAR} = \text{MCAR} + 1 - 4 * \text{MCNext}$$

Würde in den obigen Fällen die $+1$ fehlen, so könnte man immer nur Befehle am Anfang eines Segments erreichen

Wenn $\text{MCNext} = 0$ ist, führen die beiden relativen Sprungarten zum jeweils nächsten Befehl

Selbst ausprobieren:

Automatisierung der Abarbeitung des vorherigen Beispiels $[5,6] := R1 + [5,6]$

Im [CPU-Simulator](#) das Beispiel "Modify RAM with MCNext" auswählen

OpCode-gesteuerte und bedingte Sprünge

Auch mit den bisher behandelten Möglichkeiten, Sprünge zu programmieren, ist das Mikroprogramm noch nicht von außen beeinflussbar

Diese Möglichkeit wird dadurch geschaffen, dass sich Sprünge im Mikrobefehlsspeicher von dem Inhalt des RAM Speichers oder von Ergebnissen von Operationen beeinflussen lassen

Beides ist im Sprungmodus $MC=11$ möglich

OpCode-gesteuerte Sprünge

Computer werden durch Programme gesteuert, die aus Befehlen in Maschinensprache bestehen

Diese Maschinenbefehle befinden sich gemeinsam mit Daten im RAM

Jeder einzelne Maschinenbefehl besteht aus einem OpCode und ggf. weiteren Parametern

Ein Maschinenbefehl wird bei der Modell-CPU durch ein Mikroprogramm implementiert, das einige Mikrocodewörter umfassen kann und sich im Mikrobefehlsspeicher befindet

Der OpCode gibt, an wo sich dieser Befehl im Mikrobefehlsspeicher befindet

Daher werden OpCode-gesteuerte Sprünge im Mikrobefehlsspeicher benötigt

OpCode-gesteuerte Sprünge

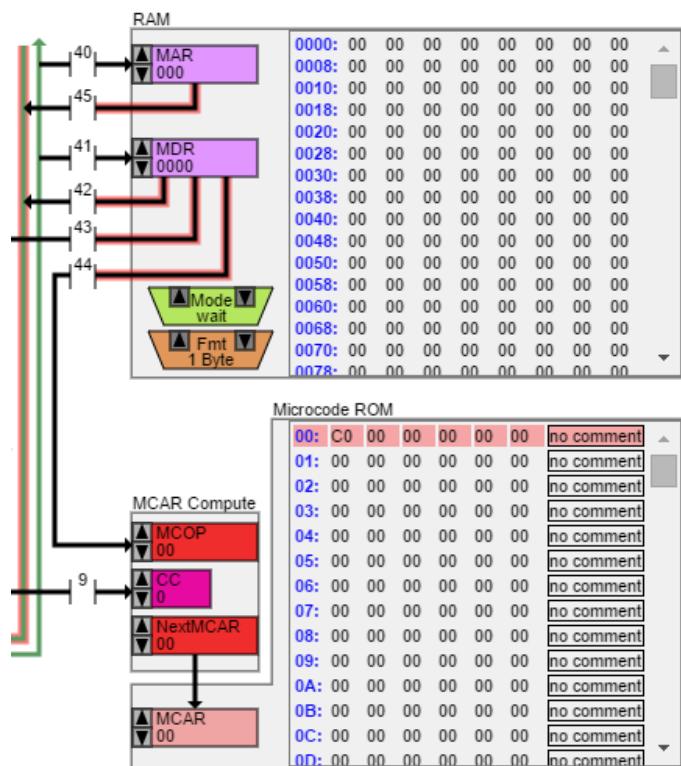
Für OpCode-gesteuerte Sprünge wird das MCOP Register verwendet, in das die Adresse des auszuführenden Sprungs von außen hineingeschrieben werden soll

Für diesen Zweck gibt es einen Datenpfad vom Datenregister des Speichers zum Register MCOP

Dieser Pfad kann über den I/O RAM Bus-Schalter $s_j = 44$ geöffnet oder geschlossen werden

Falls MC=11 ist und MCNext mit den Bits 00 beginnt, dann wird die Adresse für den nächsten Mikrobefehl so ermittelt:

$$\text{NextMCAR} = 4 * \text{MCOP}$$



Thorsten Thormählen 44 / 69

Bedingte Sprünge

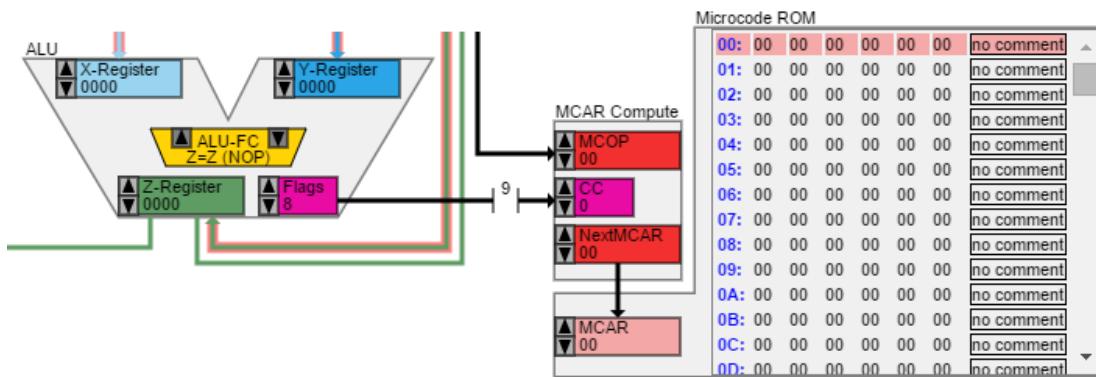
Letztendlich sollen auch Maschinenbefehle implementiert werden, die sich abhängig vom ALU Statusregister Flags anders verhalten

Dazu müssen bedingte Sprünge im Mikrocode ausgeführt werden, die sich auf den Inhalt des Statusregisters beziehen

Manchmal ist es erwünscht, die aktuellen Flags zu benutzen, manchmal die Flags aus einem vorangegangenen Befehl

Beides ist möglich. Dazu dient ein zusätzliches Condition-Code-Register cc

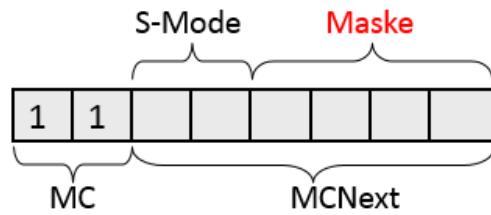
Nur wenn Bus-Schalter 9 gesetzt ist, wird der Inhalt von Flags nach cc übernommen



Bedingte Sprünge

Die Bedingung für bedingte Sprünge wird mit Hilfe einer Maske ausgewertet

Falls $MC=11$ ist, wird MCNext in S-Mode und Maske zerlegt



Der Fall $S\text{-Mode} = 00$ liefert einen OpCode-gesteuerten Sprung und wurde bereits besprochen

Andernfalls bei $S\text{-Mode} \neq 00$ werden die vier Bits des cc-Registers mit den vier Bits von Maske über ein logisches AND verknüpft

Ist das Ergebnis ungleich 0000, so wird der Sprung zur OpCode-Adresse ausgeführt:

$$\text{NextMCAR} = 4 * \text{MCOP}$$

Ist das Ergebnis gleich 0000, geht es mit dem nächsten Mikrobefehl weiter:

$$\text{NextMCAR} = \text{MCAR} + 1$$

Zusammenfassung: Sprünge im Mikrobefehlsspeicher

Sprungmode	Sprungziel
MC=00	NextMCAR = 4 * MCNext
MC=01	NextMCAR = MCAR + 1 + 4 * MCNext
MC=10	NextMCAR = MCAR + 1 - 4 * MCNext
MC=11	Zerlege MCNext in S-Mode (2 Bit) und Maske (4 Bit)

Für Sprungmode MC=11 gilt folgende Tabelle:

S-Mode	Sprungziel
S-Mode=00	NextMCAR = 4 * MCOP
S-Mode ≠ 00	NextMCAR = 4 * MCOP falls (Maske AND CC) ≠ 0000 NextMCAR = MCAR + 1 falls (Maske AND CC) = 0000

Thorsten Thormählen 47 / 69

Mikroprogramm "Kleiner Gauss"

Als Beispiel soll ein Mikroprogramm erstellt werden, dass die Summe aller Zahlen von 1 bis N berechnet, wobei N eine Zahl ist, die im RAM an der Stelle $(00)_{16}$ gespeichert ist. Das Ergebnis soll in Register R_2 geschrieben werden.

Laut [Gaußscher Summenformel](#) gilt:

$$1 + 2 + 3 + 4 + \dots + N = \sum_{n=1}^N n = N \cdot \frac{N+1}{2} = \frac{N^2+N}{2}$$

allerdings soll diese Formel nicht verwendet werden, sondern die Summe schrittweise berechnet werden

Wir erstellen das Mikroprogramm mit dem [CPU-Simulator](#)

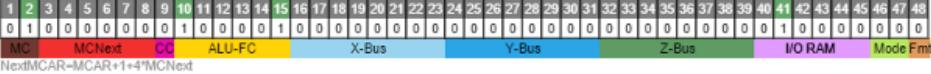
Dort können wir durch Anklicken die einzelnen Teile der Mikrobefehle zusammensetzen und diese mit Kommentaren versehen

Oder einfach in der Combo-Box "Import example file" das Beispiel "Small Gauss: 1+2+3+...+n" auswählen

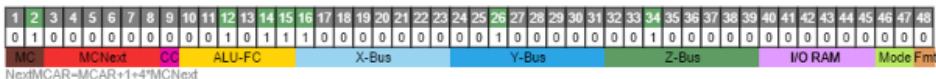
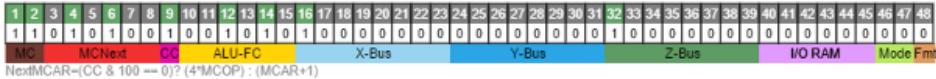
Mikroprogramm "Kleiner Gauss"

00:	Initialisiere R_0, \dots, R_7 und MAR mit 0	
	Phase 1	keine Operation
	Phase 2	$Z = 0$
	Phase 3	$R1\dots R7 = 0, MAR = 0$
 <small>NextMCAR=MCAR+1+4*MCNext</small>		
01:	Lese N aus [00H] und speichere N in R_0	
	Phase 1	Speicher liest, $Y = MDR$
	Phase 2	$Z = Y$
	Phase 3	$Z = R_0$
 <small>NextMCAR=MCAR+1+4*MCNext</small>		

Mikroprogramm "Kleiner Gauss"

02:	MDR = 1 (Sprungvorbereitung)	
	Phase 1	keine Operation
	Phase 2	Z = 1
	Phase 3	MDR = Z ₀
 <p>NextMCAR=MCAR+1+4*MCNext</p>		
03:	MCOP = MDR (noch Sprungvorbereitung)	
	Phase 1	MCOP = MDR
	Phase 2	keine Operation
	Phase 3	keine Operation
 <p>NextMCAR=MCAR+1+4*MCNext</p>		

Mikroprogramm "Kleiner Gauss"

04:	Addiere R ₀ zu R ₂	
	Phase 1	X = R ₀ , Y = R ₂
	Phase 2	Z = X + Y
	Phase 3	R ₂ = Z
	 <p>NextMCAR=MCAR+1+4*MCNext</p>	
05:	Dekrementiere R ₀ und springe zu 4 * MCOP = 04, falls Ergebnis > 0	
	Phase 1	X = R ₀
	Phase 2	Z = X - 1
	Phase 3	R ₀ = Z
	 <p>NextMCAR=(CC & 100 == 0)? (4*MCOP) : (MCAR+1)</p>	

Vom Mikroprogramm zu Maschinenbefehlen

Die Vorstellung, größere Programme in Mikrocode programmieren zu müssen, ist abschreckend

Außerdem ist die CPU per Mikrocode-Programmierung nicht univeral einsetzbar, da der Inhalt des ROMs unveränderbar ist

Programmierer sollten sich nicht damit plagen müssen, Schalter in Datenwegen zu betätigen, Daten mühsam via Adress- und Datenregister aus dem Speicher zu lesen, Code-Adressen in Code-Adress-Register zu schreiben oder ähnliche lästige Dinge festzulegen

Die Details der Benutzung der Busse und der zeitlichen Abfolge der Teilschritte in den einzelnen Phasen sollen dem Programmierer ebenfalls verborgen bleiben

Eine abstraktere Programmierschnittstelle für die CPU: **Maschinenbefehle**

Diese abstrakte Sicht der CPU zeigt immer noch Register und RAM, verschwunden sind aber Busse, ALU, Adressrechner, Phasen und Bus-Schalter

Maschinenbefehle erlauben, Operationen direkt auf Registerinhalten durchzuführen und Daten zwischen Registern und RAM zu verschieben

Außerdem gibt es Maschinenbefehle, die direkte Sprünge zu besonders gekennzeichneten Code-Stellen bewirken, anstatt dass mühsam aus Sprungmode und Masken Programmverzweigungen hergestellt werden müssen

Maschinensprache

Die Maschinensprache einer CPU ist die Menge von Maschinenbefehlen, die einem Programmierer zur Verfügung steht

Einige typischer Maschinenbefehle haben die Formen:

Op R1, R2

Op R1, W

Op R1, [Adresse]

Dabei ist

Op: eine Operation

R: ein Register

W: ein konstanter Wert

[Adresse]: eine Speicheradresse im RAM

Maschinensprache

Beispiele:

Maschinenbefehl	Bedeutung
Add R1, R2	$R1 = R1 + R2$
Mov R1, R2	$R1 = R2$
Sub R1, 2	$R1 = R1 - 2$
Mov R1, 2	$R1 = 2$
Add R1, [61h]	$R1 = R1 + \text{Inhalt von RAM Speicheradresse } (61)_{16}$
Mov R1, [61h]	$R1 = \text{Inhalt von RAM Speicheradresse } (61)_{16}$
Mov [42h], R2	Schreibe Inhalt von R2 an RAM Speicheradresse $(42)_{16}$

Sprünge in Maschinensprache

Ein absoluter Sprungbefehl lautet: `Jmp Adresse`
wobei `Adresse` eine RAM Speicheradresse ist, an der der nächste auszuführende Befehl beginnt

Bedingte Sprünge bestehen aus einer Vergleichsoperation mit anschließender Sprunganweisung, die auf dem Ergebnis des Vergleichs basiert
`Op R1, R2`
`CondJmp Adresse`

Jede Operation, die die Flags der ALU beeinflusst, kann als Vergleichsoperation dienen

Die Sprungbedingung ergibt sich aus den Flags

Meist werden die Flags mit einer Maske ausgewertet. Masken für typische Sprünge sind meist durch Buchstaben codiert, z.B.:
`JLE`: "Jump if less equal" ("Springe bei kleiner gleich")

Beispiel:

```
Sub R1, R2
JLE 127
```


Registerkonvention

Die Operanden von Maschinenbefehlen sind Register oder RAM Speicherplätze

Es dürfen aber nie beide Operanden RAM Speicherplätze sein

Die Register werden für bestimmte Zwecke reserviert, etwa als Programmzähler, als Zeiger auf den Stack oder den Datenbereich

Einige Register behält man als Rechenregister. Diese werden häufig Allzweckregister oder Akkumulatoren genannt. Viele Befehle der Maschinensprache sind nur mit Allzweckregistern durchführbar

Beispiel für eine Registerkonvention für unsere Modell-CPU:

Register	Vereinbarter Zweck
R0	Programmzähler (Program Counter) PC
R1	Akkumulator A
R2	Akkumulator B
R3	Loop Index I
R4	Hilfsregister Aux
R5	I/O Port
R6	Data Pointer
R7	Stack Pointer

Mikroprogrammierte Maschinenbefehle

Jeden Maschinensprachebefehl können wir durch ein kurzes Mikroprogramm implementieren

ADD A, B

ADD A, [B]

Mov A, 7

Mikroprogrammierte Maschinenbefehle

Jump if not zero

Falls das Ergebnis der vorigen Operation = 0, führe den nächsten Maschinenbefehl aus (MCOP sei dafür entsprechend gesetzt), sonst setze PC (R0) auf 7 (Sprung!)

JNZ 7

NextMCAR=MCAR+1+4*MCNext

NextMCAR=(CC & 1000 == 0)? (4*MCOP) : (MCAR+1)

MC MCNext
NextMCAB=4*MCOP

Thorsten Thormählen 58 / 69

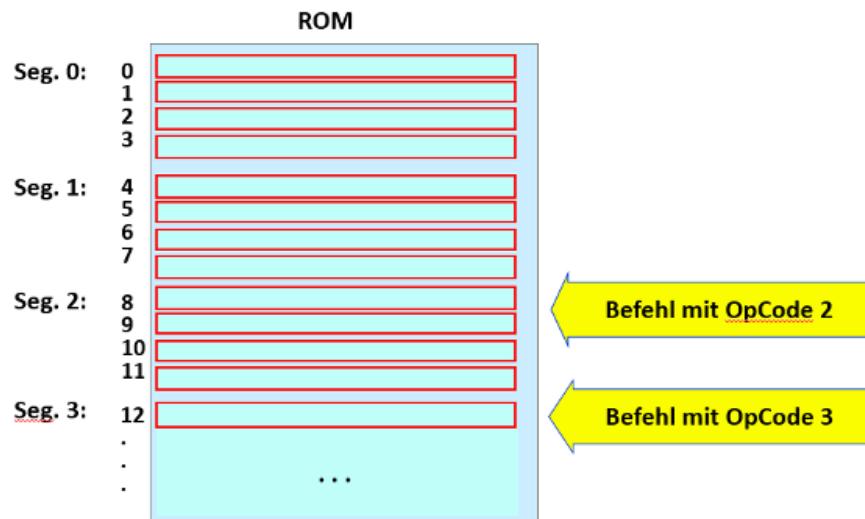
Der Maschinensprache-Interpreter

Jeder Maschinensprachebefehl erhält eine Nummer, den OpCode

Er wird als kurze Mikrocoderoutine im ROM abgelegt

Die Routine beginnt immer am Anfang eines Segments

Der OpCode ist identisch mit der Segmentnummer



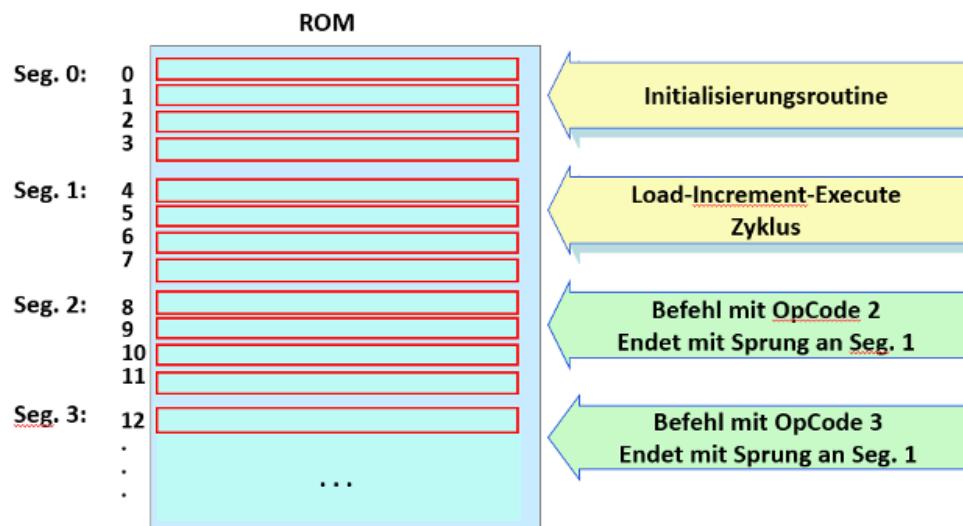
Der Maschinensprache-Interpreter

Ein Interpreter für Maschinensprache:

Segment 0 enthält Initialisierungsbefehle

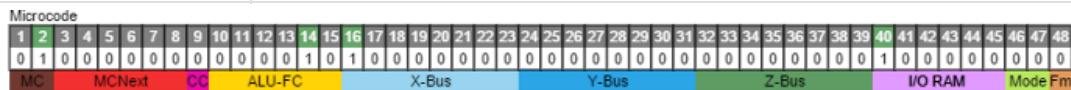
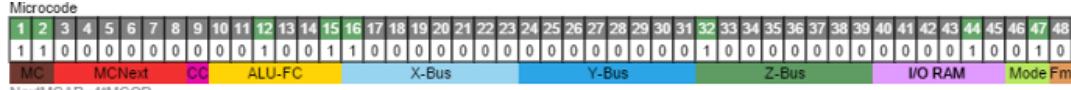
Segment 1 enthält einen Load-Increment-Execute-Zyklus

Jeder Maschinensprachebefehl endet mit einem Sprung an den Anfang von Segment 1



Load-Increment-Execute-Zyklus

Der Load-Increment-Execute-Zyklus besteht aus 2 Mikrobefehlen:

04:	Lade den Programmzähler	
	Phase 1	X = R ₀
	Phase 2	Z = X
	Phase 3	MAR = Z
Microcode  NextMCAR=MCAR+1+4*MCNext		
05:	Jump Execute	
	Phase 1	MCOP = MDR (Lies 1 Byte Opcode)
	Phase 2	Z = X + 1, MCAR = 4 * MCOP
	Phase 3	R ₀ = Z, MAR = Z
Microcode  NextMCAR=4*MCOP		

Load-Increment-Execute-Zyklus

Achtung: Die Befehle des Load-Increment-Execute-Zyklus dürfen keine Flags verändern, da die Flags evtl. vom nächsten Maschinenbefehl benötigt werden

Das cc-Bit $s_j = 9$ muss also auf 0 gesetzt werden, damit die Befehle $z = x$ und $z = x + 1$ das cc-Register nicht beeinflussen können

Maschinencode im RAM

Während das Mikroprogramm im ROM fest vorgegeben und nicht veränderbar ist, kann im RAM ein Programm in Form von Maschinenbefehlen zusammen mit seinen Argumenten liegen

Der Load-Increment-Execute-Zyklus führt dieses Programm aus



Jeder Maschinenbefehl, der Argumente benötigt, kann bei seiner Implementierung als Mikrocode davon ausgehen, dass MAR auf das erste benötigte Argument zeigt

Die Implementierung jedes Maschinenbefehls muss garantieren, dass hinterher der Programmzähler PC auf der Speicheradresse des Opcodes des nächsten Befehls im RAM steht

Selbst ausprobieren:

Im [CPU-Simulator](#) das Beispiel "Load-Increment-Execute cycle" auswählen

Entwicklung eines Maschinenspracheprogramms

1. Schritt: Java-Programm

```
// Kleiner Gauss: Berechne die Summe aller Zahlen von 1 bis N
static int gauss(int n) {
    int sum = 0;
    while (n > 0) {
        sum += n;
        n--;
    }
    return sum;
}
```

2. Schritt: Ersetze Programmstrukturen durch Sprünge (Linearisiere das Programm)

```
int sum = 0;
loop: if(n == 0) goto end;
      sum = sum + n ;
      n = n - 1;
      goto loop
end:
}
```


Entwicklung eines Maschinenspracheprogramms

2. Schritt: Ersetze Programmstrukturen durch Sprünge (Linearisiere das Programm)

```
int sum = 0;
loop: if(n == 0) goto end;
      sum = sum + n ;
      n = n - 1;
      goto loop
end:
}
```

3. Schritt: Wähle Register oder Speicherplätze für die Variablen, benutze Assemblerbefehle

```
// Registerbelegung: sum --> A, n --> B
MOV A, 0
loop: CMP B, 0
      JE end
      ADD A,B
      DEC B
      JMP loop
end: STOP
```


Entwicklung eines Maschinenspracheprogramms

4. Schritt: Implementiere die benötigten Befehle im Simulator

Für unser Beispiel sind es folgende 7 Befehle:

Befehl	Länge im RAM	OpCode
JMP <adr>	2 Byte	02
JE <adr>	2 Byte	03
DEC B	1 Byte	04
ADD A,B	1 Byte	05
CMP B, <num>	2 Byte	06
MOV A, <num>	2 Byte	07
STOP	1 Byte	08

Dabei kann der OpCode frei zwischen 2 und 63 gewählt werden

Entwicklung eines Maschinenspracheprogramms

5. Schritt:

- a) Übertrage das Programm in das RAM. Im ersten Durchlauf bleiben Sprungadressen noch offen

Position:	0	1	2	3	4	5	6	7	8	9	0A	0B	...
Inhalt:	07h	00h	06h	00h	03h	?	05h	04h	02h	?	08h	..	

MOV A CMP B JE ADD A,B DEC B JMP STOP

- b) Trage die richtigen Sprungadressen durch "Backpatching" nach:

Position:	0	1	2	3	4	5	6	7	8	9	0A	0B	...
Inhalt:	07h	00h	06h	00h	03h	0Ah	05h	04h	02h	02h	08h	..	

Selbst ausprobieren:

Im [CPU-Simulator](#) das Beispiel "Small Gauss (machine-code version)" auswählen

Abstraktionsebenen

Im Laufe der bisherigen Vorlesungen haben wir mehrere aufeinander aufbauende Abstraktionsebenen kennengelernt

CMOS Transistor Schaltungen

Logik Gatter

ALU, Register, Bus-Schalter

Mikrocode

Maschinenbefehle

Diese Abstraktionsebenen sind wichtig, um sich als Hardware-Designer bzw. beim hardware-nahen Programmieren, auf die aktuelle Aufgabe zu fokussieren

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[\[Impressum\]](#) , [\[Datenschutz\]](#) , []

Thorsten Thormählen 69 / 69

Technische Informatik I

x86 Maschinensprache + Assembler

Thorsten Thormählen
14. Januar 2025
Teil 10, Kapitel 1

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Einführung in die x86 Maschinen- und Assemblersprache

Register der x86 Familie

Assemblerbefehle der x86 Familie

Maschinensprache

Computer werden durch Programme gesteuert, die aus Befehlen in Maschinensprache bestehen (wie aus dem vorangegangenen Kapitel bekannt)

Diese Befehle liegen im Hauptspeicher und werden nacheinander abgearbeitet

Die Maschinensprache wird vom Hersteller einer CPU definiert und kann vom Programmierer nicht verändert werden

Der Hersteller legt fest, welcher Zahlencode im Speicher welchem Maschinensprachebefehl entspricht

Speicher:	Adresse:
	0x00B813A0 55 8b ec 81 ec c0 00 00 00
	0x00B813B1 03 b8 cc cc cc f3 ab 8b f4 68 3c 57 b8 00 ff 15 bc 82 b8 00 83
	0x00B813C2 c4 04 3b f4 e8 66 fd ff ff 33 c0 5f 5e 5b 81 c4 c0 00 00 00 3b ec
	0x00B813E2 08 54 fd ff ff 8b e5 5d c3 cc cc cc cc ee ee ee ee ee ee cc cc
	0x00B813F3 cc cc cc cc cc cc ff 25 bc 82 b8 00 cc
	0x00B8140E ee ee 75 01 c3 55 8b ec 83 ec 00 50 52 53 56 57 8b 45 04 6a 00 50
	0x00B81424 e8 80 fd ff ff 83 c4 08 6f 5e 5b 5a 58 8b e5 5d c3 cc cc cc cc cc
	0x00B8143A cc cc cc cc cc cc 8b ff 55 8b ec 51 53 56 57 33 ff 8b f2 39 3e 8b
Bedeutung: (Prozessortyp abhängig)	push ebp mov esp,ebp sub esp,0C0h push ebx push esi push edi

Damit ist ein Programm in Maschinensprache immer nur auf einer bestimmten CPU bzw. auf den untereinander kompatiblen CPUs einer Prozessorfamilie lauffähig

Maschinensprache / Assembler

Eine Maschinensprache ist die Menge von Befehlen, die einem Programmierer für eine konkrete CPU zur Verfügung steht

Diese Befehle werden normalerweise in der Praxis nicht direkt durch ihren Zahlencode eingegeben, sondern in einer Assemblersprache (auch "Assembler" genannt)

Eine Assemblersprache ist einer lesbaren Art, Programme in Maschinensprache zu formulieren

Die lesbare Variante eines Maschinenbefehls in Assemblersprache wird Mnemonik genannt

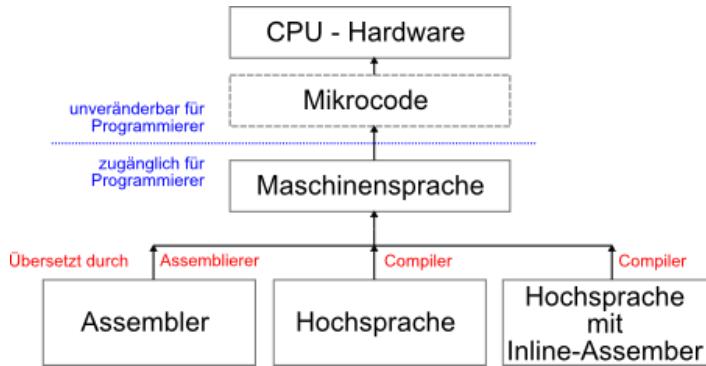
So wird z.B. für den Sprungbefehl die Mnemonik `jmp` verwendet (für engl. "jump");

Es gibt somit eine eindeutige Abbildung zwischen Maschinensprache und Assemblersprache

Ein Assemblierer ist ein Übersetzer von Assemblersprache in Maschinensprache

Ein Disassemblierer ist ein Übersetzer von Maschinensprache in Assemblersprache

Erzeugung von Maschinensprache mittels Compiler



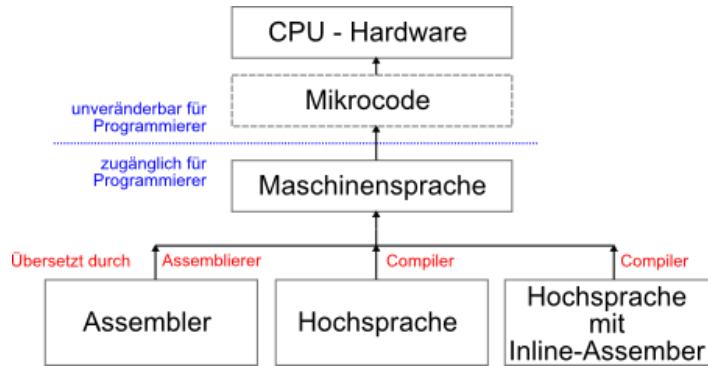
Softwareentwicklung findet heutzutage typischerweise in einer höheren Programmiersprache statt (z.B. C, C++, Fortran, Pascal, ...)

Um ein in einer höheren Programmiersprache geschriebenes Programm ausführen zu können, muss dies jedoch letztendlich durch einen Compiler in eine äquivalente Folge von Maschinenbefehlen übersetzt werden

Beim Kompilieren kann angegeben werden für welchen CPU Typ der Maschinencode erzeugt werden soll

Daher kann ein Programm in Hochsprache im Gegensatz zu einem Assemblerprogramm leicht auf andere Hardware angepasst werden

Inline-Assembler



Früher wurden viele Programme in Assembler erstellt

Heute ist mit der schnelleren Hardware die Bedeutung von Maschinensprache zurückgedrängt worden

Softwareentwicklung findet typischerweise in einer höheren Programmiersprache statt

Bei sehr zeitkritischen Funktionen innerhalb eines Programms wird jedoch weiterhin Maschinensprache in Form von Inline-Assembler verwendet

Der Compiler optimiert nur den Teil, der in der Hochsprache geschrieben ist, der Assemblerteil wird direkt in die Maschinensprache übertragen

x86 Maschinensprache

In dieser Vorlesung wird aufgrund der starken Verbreitung und Praxisrelevanz, im Folgenden die Maschinensprache der x86 Prozessorfamilie vorgestellt

Diese Maschinensprache ist auf allen 80x86 und x86-64 Prozessoren der Firmen Intel und AMD ausführbar, d.h. auf fast jedem Desktop-PC oder Laptop (Ausnahmen sind z.B. PowerPC-basierte Apple Computer)

Hörer der Vorlesung können zur Maschinensprache-Programmierung einen Rechner in den PC-Pools des Fachbereichs oder ihren privaten Rechner verwenden

Werkzeuge zur Erstellung von x86 Maschinensprache

Zur Erstellung vom x86 Assembler stehen verschiedene Werkzeuge zur Auswahl:

Microsoft Macro Assembler (MASM) ist ein verbreiteter Assembler für MS-DOS und Windows

MASM32 ist eine freie Variante von MASM

Turbo Assembler (TASM) ist ein Assembler von Borland für das Betriebssystem MS-DOS

NASM ("Netwide Assembler") ist ein beliebter Assembler, der für verschiedene Betriebssysteme (DOS, Windows, Unix) verwendet werden kann. Es gibt sogar Online-Dienste bei denen ohne Installation auf dem eigenen Rechner mit NASM experimentiert werden kann:

<https://www.jdoodle.com> , <https://onecompiler.com> ,

Diese Werkzeuge unterscheiden sich natürlich nicht in den Maschinenbefehlen (diese sind durch die x86 Prozessorfamilie vorgegeben) sondern lediglich in der Art der Darstellung und durch unterschiedliche Makros, die die Arbeit erleichtern sollen

Inline-Assembler in Microsoft Visual C/C++

In dieser Vorlesung werden die gezeigten Maschinensprache-Beispiele in Form von Inline-Assembler innerhalb eines C/C++-Programms vorgestellt

Als Entwicklungsumgebung wird Microsoft Visual Studio verwendet

Vorteile:

Inline-Assembler wird in der Praxis gerne verwendet, da niemand größere Projekte komplett in Assembler entwickeln möchte

Auf Symbole und Variablen des C/C++-Programms kann direkt innerhalb des Assembler-Codes zugegriffen werden

Die Syntax ist verhältnismäßig einfach

Microsoft Visual Studio hat einen integrierten Debugger

Auch Registerinhalte, Speicher etc. lassen sich komfortabel anzeigen

Visual Studio ist auf jedem Windows-Rechner im Fachbereich installiert

Microsoft Visual Studio (Community Version) steht kostenfrei zur Verfügung:

[Download](#)

Nachteil:

Plattformabhängigkeit: Für die Verwendung mit anderen Assemblern muss die Syntax entsprechend angepasst werden

Thorsten Thormählen 11 / 76

Inline-Assembler in Microsoft Visual C/C++

Beispiel:

```
#include <stdio.h> // includes "printf" command ;  
  
int add(int varA, int varB ) {  
    // inline assembler starts here  
    __asm {  
        mov eax, varA    ; // move first argument to EAX register  
        mov ecx, varB    ; // move second argument to ECX register  
        add eax, ecx     ; // EAX = EAX + ECX  
    }  
}  
  
int main() {  
    int result = add(1, 2);    // call C-function "add"  
    printf( "1 + 2 = %d\n", result); // output result to console  
    return 0;  
}
```

Quelldatei: [main.cpp](#)

Projekt für MS Visual Studio: [VS2012](#) , [VS2013](#) [VS2015](#) [VS2017](#) [VS2019](#) [VS2022](#)

Diese Projektdateien können für alle folgenden Beispiele wiederverwendet werden, indem jeweils die Quelldatei "main.cpp" ersetzt wird

Inline-Assembler in Microsoft Visual C/C++

Verwendung vom Microsoft Visual Studio:

Erstellen des Projekts: F7

Ausführen: CTRL+F5

Debuggen starten: F5

Debuggen Einzelschritt: F11

Debuggen Prozedurschritt: F10

Breakpoint setzen/entfernen: in die Leiste neben dem Quellcode klicken

Anzeigen von Registern:

Menü → Debuggen → Fenster → Register

Anzeigen von Arbeitsspeicher:

Menü → Debuggen → Fenster → Arbeitsspeicher

The screenshot shows the Microsoft Visual Studio interface during a debug session. The top window displays the source code for `main.cpp`, which contains C code with inline assembly instructions. The assembly code is annotated with comments explaining the purpose of each instruction. The bottom left window, titled "Registers", shows the current values of various CPU registers. The bottom right window, titled "Memory", shows the state of memory at address `0x00A713A0`. The assembly code in the source file is:

```
int add(int varA, int varB) {
    __asm {
        mov eax, varA ; // move first argument to EAX register
        mov ecx, varB ; // move second argument to ECX register
        add eax, ecx ; // EAX = EAX + ECX
    }
}

int main() {
    int result = add(1, 2); // call C-function "add"
    printf("1 + 2 = %d\n", result); // output result to console
    return 0;
}
```

The Registers window shows:

Register	Value
EAX	00000001
EBX	7E1B7000
ECX	0007CF9003
EDX	00000001
EST	0007C1114
EDT	0007C11144
ETP	0007C13A6
ESP	007CF90DC
EBP	007CF90DC
EBP	000000246

The Memory window shows the memory dump starting at address `0x00A713A0`.

Thorsten Thormählen 13 / 76

Alternative für Linux-Nutzer

Unter Linux ist die Verwendung von Microsoft Visual Studio nicht möglich. Als Alternative zum Kompilieren der bereitgestellten Beispiele kann der [Intel C and C++ Compiler](#) mit der Option `-use-msasm` verwendet werden

In einer Shell:

```
> gcc -use-msasm -o my_program main.cpp
```

Der Intel Compiler steht Studenten kostenfrei zur Verfügung: [Download](#)

Eine weitere Möglichkeit sind Online-Dienste, bei denen ohne Installation auf dem eigenen Rechner mit dem Microsoft Compiler experimentiert werden kann:

<https://gcc.godbolt.org>

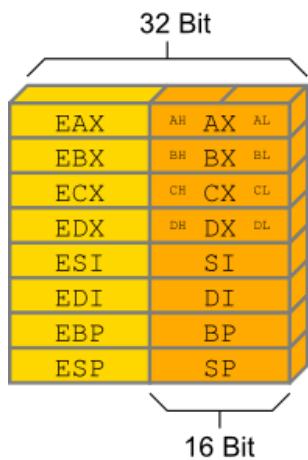
Als Compiler "x86 msvc (WINE)" auswählen

Diese Alternativen haben jedoch keinen integrierten Debugger und werden daher nur bedingt empfohlen

Microsoft Visual Studio ist auf jedem Windows-Rechner im Fachbereich installiert

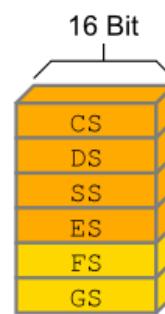
Register der x86-Familie

Allzweckregister



Sonderfunktion:
Akkumulator
Base
Counter
Data
Source Index
Destination Index
Base Pointer
Stack Pointer

Segmentregister



Funktion:
Code Segment
Data Segment
Stack Segment
Extra Segment

Befehlszähler



Flagregister



Register der x86-Familie

Die ersten x86-Prozessoren (8088, 8086 und 80286) waren 16-Bit Prozessoren
Sie hatten 14 Register (orange gekennzeichnet in Abb.)

8 Allzweckregister, die jedoch jeweils auch noch eine Sonderfunktion hatten

4 Segment Register

1 Befehlszähler

1 Flagregister

Das niederwertige Byte (Low Byte) bzw. höherwertige Byte (High Byte) der Register AX, BX, CX und DX sind als 8-Bit Register gesondert ansprechbar (z.B. AX besteht aus AL und AH)

Seit dem 80386 wird mit 32-Bit breiten Registern gearbeitet

Die bestehenden Register wurden erweitert (gelb gekennzeichnet) und können als EAX, EBX, usw. angesprochen werden

Die 16-Bit Teile der Register sind weiterhin ansprechbar. Neuerungen in der x86-Architektur waren bisher immer abwärts-kompatibel, d.h. neuere Prozessoren können ältere Programme immer noch ausführen

Spätere x86-Prozessoren haben weitere Register eingeführt (MMX-Register, SSD-Register, usw.) diese werden zunächst hier nicht weiter betrachtet

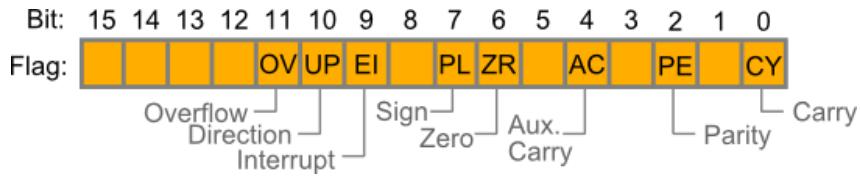
Thorsten Thormählen 16 / 76

Allzweckregister

Die Allzweckregister haben jeweils noch eine Sonderfunktion:

Register	Sonderfunktion	Erklärung
AX	Akkumulator	Ziel für Rechenoperationen
BX	Base	Zeiger für Zugriffe auf Speicher
CX	Counter	Zähler in Schleifen
DX	Data	Datenregister
SI	Source Index	Quellindex für Verarbeitung von Zeichenketten
DI	Destination Index	Zielindex für Verarbeitung von Zeichenketten
BP	Base Pointer	Anfangsadresse des Stapelsegments
SP	Stack Pointer	Stapelzeiger

Flagregister



Das Flagregister ändert sich nach arithmetischen Operationen

Es dient dazu, die Situationen nach der Durchführung einer ALU-Operation anzugeben

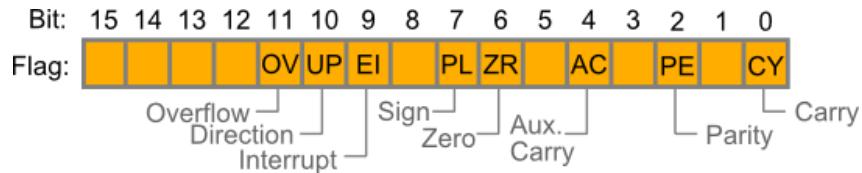
Es ist eigentlich ein Ausgaberegister, dennoch kann es auf dem Umweg über den später zu besprechenden Stack gezielt verändert werden

Von den 16 bzw. 32 Bits des Flag-Registers sind nur 8 für uns interessant:

Die Flags CY, AC, OV, PL, ZR, PE beziehen sich immer auf das Ergebnis einer gerade durchgeföhrten Operation

Die Flags UP, EI dienen als Schalter. Sie bleiben unverändert, bis man sie durch Spezialbefehle verändert

Flagregister



Flag	Bedeutung	Erklärung
CY	Carry	Bereichsüberschreitung für vorzeichenlose Zahlen
AC	Aux. Carry	Bereichsüberschreitung für vorzeichenlose 4-Bit Zahlen
OV	Overflow	Bereichsüberschreitung bei arithmetischer Operation auf Zahlen mit Vorzeichen
PL	Sign	Ergebnis war negativ
ZR	Zero	Ergebnis war null
PE	Parity	Ergebnis hat eine gerade Anzahl von Einsen (im niederwertigsten Byte)
UP	Direction	Legt die Richtung von String-Befehlen fest
EI	Interrupt	Bestimmt, ob Interrupts zugelassen werden

Thorsten Thormählen 19 / 76

Segmentregister

Die Segmentregister werden später bei der Adressierung des Speichers eine Rolle spielen:

Register	Funktion	Erklärung
CS	Code Segment	Anfangsadresse des Code-Segments
DS	Data Segment	Anfangsadresse des Daten-Segments
SS	Stack Segment	Anfangsadresse des Stack-Segments
ES	Extra Segment	keine besondere Funktion
FS		keine besondere Funktion
GS		keine besondere Funktion

Format von Assemblerbefehlen

In Assembler wird je ein Befehl pro Zeile geschrieben

Dabei hat ein Assemblerbefehl die Form:

```
[Label] [Operation] [Operanden] [;Kommentar]
```

Die eckigen Klammern sollen andeuten, dass diese Komponente optional ist, d.h. es ist auch eine leere Zeile möglich

Zwischen Groß- und Kleinschreibung wird nicht unterschieden (außer wenn im Inline-Assembler auf C/C++ Variablen zugegriffen wird)

Leerzeichen zwischen den Komponenten werden ignoriert

Bei einem Semikolon beginnt ein Kommentar, der sich bis zum Zeilenende erstreckt

Alternativ kann ein Kommentar in MS Visual Studio auch mit den in C/C++ üblichen doppelten Schrägstrichen "://" gekennzeichnet werden

Format von Assemblerbefehlen

Viele Assemblerbefehle haben die Form:

```
op ziel, quelle
```

Dies bedeutet, dass die Operanden Ziel und Quelle mit der Operation `op` verknüpft werden

Das Ergebnis wird in Ziel gespeichert

In Java bzw. C/C++ entspräche dies demnach: `Ziel = Ziel op Quelle;`

Ziel kann ein Register oder eine Speicheradresse sein

Quelle kann ein Register, eine Speicheradresse oder ein konstanter Zahlenwert sein

Die verknüpften Operanden müssen in ihrer Bitbreite zueinander passen

Beispiele:

```
mov eax, ecx ; eax = ecx
sub ax, cx ; ax = ax - cx
add cx, ax ; cx = cx + ax
```


Der Assemblerbefehl MOV

Einer der meist verwendeten Befehle ist der Move-Befehl:

```
mov ziel, quelle
```

Dabei werden die Daten aus der Quelle zum Ziel kopiert

Ziel kann ein Register oder eine Speicheradresse sein

Quelle kann ein Register, eine Speicheradresse oder eine Konstante sein

Einschränkungen:

- Es können nicht beide Operanden Speicheradressen sein

- Es können nicht beide Operanden Segmentregister sein

- Konstanten können nicht direkt in Segmentregister geschrieben werden

Der Assemblerbefehl MOV

Beispiele für den Move-Befehl:

```
int main() {
    int varA = 6;
    int varB = 7;
    __asm {
        mov eax, 10h          ; // move the hex value 10 to EAX
        mov eax, 10            ; // move the decimal value 10 to EAX
        mov eax, 010           ; // move the octal value 10 to EAX
        mov ecx, eax          ; // move EAX register to ECX register
        //mov fs, ds           ; // does not work
        //mov ax, ds            ; // this works instead: (but segment registers
        //mov fs, ax            ; // should not be modified)
        mov eax, 11111111h     ; // EAX = 11111111h;
        mov ax, 3333h          ; // EAX = 11113333h;
        //mov varA, varB       ; // does not work
        mov eax, varB          ; // this works instead:
        mov varA, eax
    }
    return 0;
}
```

Quelldatei: [main.cpp](#)

Der Assemblerbefehl ADD

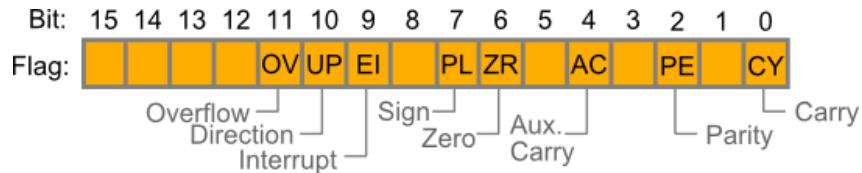
Der Befehl add führt eine Addition aus:

```
add ziel, quelle
```

Beispiele für den Add-Befehl (Quelldatei: [main.cpp](#)):

```
#include <stdio.h>
int main() {
    int varA = 6, varB = 7;
    __asm {
        //add varA, varB      ; // does not work
        mov eax, varB        ; // this works instead:
        add varA, eax        ; //
    }
    printf("varA = %d\n", varA);
    __asm {
        mov eax, 21
        mov edx, 24
        add eax, edx
        mov varA, eax
    }
    printf("varA = %d\n", varA);
    return 0;
}
```


Arithmetische Operationen und das Flagregister



Arithmetische Operationen, wie z.B. der Befehl add, verändern das Flagregister

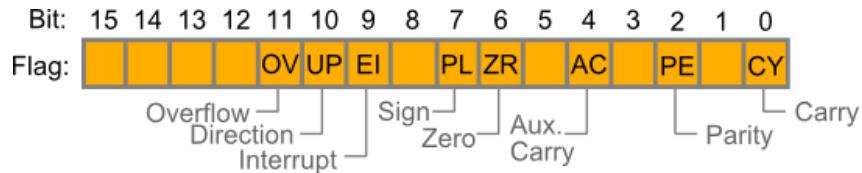
Der Prozessor weiß nicht, ob der Inhalt eines Registers als vorzeichenlose Zahl (unsigned) oder als vorzeichenbehaftete ganze Zahl in Zweierkomplement-Darstellung interpretiert werden muss

Die Operation wird einfach ausgeführt und die Flags gesetzt. Die Interpretation der Flags ist Sache des Programmierers

Nicht alle Befehle beeinflussen alle Flags, so dass auch später noch durch eine frühere Operation erzeugte Flags ablesbar sind

Der Befehl add beeinflusst: OV, PL, ZR, AC, PE und CY

Arithmetische Operationen und das Flagregister



Beispiele (Quelldatei: [main.cpp](#)):

```
int main() {
    __asm
    {
        mov eax, 00000000h
        mov ecx, 00000000h
        add eax, ecx      ; // EAX=0;           Flags: OV=0 PL=0 ZR=1 AC=0 PE=1 CY=0
        mov eax, 00000000h
        mov ecx, 00000001h
        add eax, ecx      ; // EAX=1;           Flags: OV=0 PL=0 ZR=0 AC=0 PE=0 CY=0
        mov eax, 00000001h
        mov ecx, 0000000Fh
        add eax, ecx      ; // EAX=10;          Flags: OV=0 PL=0 ZR=0 AC=1 PE=0 CY=0
        mov eax, 00000000h
        mov ecx, 7FFFFFFFh
        add eax, ecx      ; // EAX=7FFFFFFF; Flags: OV=0 PL=0 ZR=0 AC=0 PE=1 CY=0
        mov eax, 00000001h
```


Quiz

Frage: Was ist das Ergebnis folgender Assemblerbefehle?

```
mov eax, 00000002h  
mov ecx, 7FFFFFFEh  
add eax, ecx
```

Antwort 1: EAX=FFFFFFFFFF; Flags: OV=0 PL=1 ZR=0 AC=1 PE=0 CY=0

Antwort 2: EAX=80000000; Flags: OV=1 PL=1 ZR=0 AC=1 PE=1 CY=0

Antwort 3: EAX=80000000; Flags: OV=0 PL=0 ZR=1 AC=1 PE=0 CY=1

Am Online-Quiz teilnehmen durch Besuch der Webseite:
www.onlineclicker.org

Arithmetische Operationen und das Flagregister

```
mov al, -3 ; // AL=FDh;
mov cl, -1 ; // CL=FFh;
add al, cl ; // AL=FCh; Flags: OV=0 PL=1 ZR=0 AC=1 PE=1 CY=1
mov al, 253 ; // AL=FDh;
mov cl, 255 ; // CL=FFh;
add al, cl ; // AL=FCh; Flags: OV=0 PL=1 ZR=0 AC=1 PE=1 CY=1
```

In diesem Beispiel (Quelldatei: [main.cpp](#)) werden zweimal exakt die gleichen Operationen ausgeführt

Der Assembler übersetzt -3 und 253 in das gleiche Bitmuster: 11111101 = FDh

Der Assembler übersetzt -1 und 255 in das gleiche Bitmuster: 11111111 = FFh

Nach der Addition ist in beiden Fällen das Ergebnis FCh=252 und Overflow=0
Carry=1

Interpretation als Addition von vorzeichenlosen Zahlen:

Das gesetzte Carry zeigt, dass das Ergebnis 252 falsch ist: 253+255=508

Interpretation als Addition von vorzeichenbehafteten Zahlen im Zweierkomplement:

Das nicht gesetzte Overflow zeigt, dass das Ergebnis FCh=-4 richtig ist: -3 + -1 = -4

Erkenntnis: Für die richtige Interpretation sind die Flags wichtig

Der Assemblerbefehl ADC

Um die Addition $253+255=508$ doch richtig auszuführen, gibt es zwei Möglichkeiten

Ein größeres Register verwenden

```
mov eax, 253  
mov ecx, 255  
add eax, ecx ; // EAX=0000001FC; Flags: OV=0 PL=0 ZR=0 AC=1 PE=1 CY=0
```

Den Befehl `adc` verwenden, der genau wie der `add` Befehl arbeitet, jedoch zusätzlich das Carry-Bit hinzugaddiert

```
mov al, 253 ; // AL=FDh;  
mov cl, 255 ; // CL=FFh;  
mov ah, 0 ; // AH=0h;  
add al, cl ; // AL=FCh; AH=0h; Flags: OV=0 PL=1 ZR=0 AC=1 PE=1 CY=1  
adc ah, 0 ; // AL=FCh; AH=1h; Flags: OV=0 PL=0 ZR=0 AC=0 PE=0 CY=0  
// AX=01FCh
```


Der Assemblerbefehl SUB

Der Befehl `sub` führt eine Subtraktion aus:

```
sub ziel, quelle
```

Der Befehl beeinflusst die gleichen Register, wie der Add-Befehl: OV, PL, ZR, AC, PE und CY

Bei x86 wird das Carry CY gesetzt, wenn ($ziel < quelle$) (bei Interpretation als vorzeichenlose Zahl). Das Carry erfüllt dann die Funktion eines Borrow.

Bei vorzeichenbehafteten Zahlen ist wieder nur das Overflow-Flag OV relevant

Eine Subtraktion kann hardwaretechnisch durch eine Addition von `quelle` und dem Zweierkomplement von `ziel` implementiert werden, also $a - b = a + (-b)$ (siehe Kapitel [Arithmetik Schaltungen](#))

Dies funktioniert sofort perfekt bei Interpretation der Operanden als vorzeichenbehaftete Zahl

Bei der Interpretation als vorzeichenlose Zahl wird allerdings das Carry von der gezeigten Harwareschaltung gesetzt, wenn ($ziel \geq quelle$). Für den x86-sub Befehl, müsst das Carry der Harwareschaltung demnach invertiert werden

Der Assemblerbefehl SUB

Beispiel für den sub-Befehl (Quelldatei: [main.cpp](#)):

```
int main() {
    __asm {
        mov al, 2      ; // AL=02h;
        mov bl, -1     ; // BL=FFh;
        sub al, bl     ; // AL=03h; Flags: OV=0 PL=0 ZR=0 AC=1 PE=1 CY=1
        mov al, 2      ; // AL=02h;
        sub al, 255    ; // AL=03h; Flags: OV=0 PL=0 ZR=0 AC=1 PE=1 CY=1
    }
    return 0;
}
```

Kontrollrechnung mit dem Zweierkomplement (Carry cy wird invertiert):

$$\begin{array}{r} 00000010 & (2) \\ -11111111 & (-1) \text{ bzw. } (255) \\ \downarrow \\ 00000010 & (2) \\ +00000001 & \text{Zweierkomplement} \\ \hline 00000011 & (3) \end{array}$$

Thorsten Thormählen 32 / 76

Der Assemblerbefehl SBB

Der Befehl sbb steht für engl. "subtract with borrow", d.h. es wird eine Subtraktion ausgeführt und zusätzlich das Carry subtrahiert

Beispiel für den sbb-Befehl (Quelldatei: [main.cpp](#)):

```
int main() {
    __asm {
        //
        mov ax, 800      ; // AX=320h; AL=20h
        mov cl, 34       ; // CL=22h;
        sub al, cl       ; // AL=FEh=-2; Flags: OV=0 PL=1 ZR=0 AC=1 PE=0 CY=1
        sbb ah, 0         ; // AH=02h;     Flags: OV=0 PL=0 ZR=0 AC=0 PE=0 CY=0
        // AX=2FE=766
    }
    return 0;
}
```


Größenvergleich mittels SUB Befehl

Der Befehl `sub` kann verwendet werden, um zwei Operanden zu vergleichen

```
sub operandA, operandB
```

Ist es so einfach?

Zero-Flag=0, Sign-Flag=1 bedeutet $\text{operandA} < \text{operandB}$

Zero-Flag=1, Sign-Flag=0 bedeutet $\text{operandA} = \text{operandB}$

Zero-Flag=0, Sign-Flag=0 bedeutet $\text{operandA} > \text{operandB}$

Nein, es kommt darauf an, ob es sich um eine vorzeichenlose oder vorzeichenbehaftete Zahl handelt. Beispiel:

Ist der Hex-Wert 2h kleiner oder größer als FFh?

Wenn vorzeichenlos: 2h=2, FFh=255 → Ergebnis: kleiner

Wenn vorzeichenbehaftet: 2h=2, FFh=-1 → Ergebnis: größer

Daher werden zwei verschiedene Größenvergleiche eingeführt:

Wenn vorzeichenlos: "above" und "below"

Wenn vorzeichenbehaftet: "greater" und "less"

Größenvergleich mittels SUB Befehl

```
sub operandA, operandB
```

Vorzeichenlose Zahlen: "above" und "below"

Aussage	Bedingung
operandA below operandB	CY=1
operandA equal operandB	ZR=1
operandA above operandB	CY=0

Vorzeichenbehaftete Zahlen: "less" und "greater"

Aussage	Bedingung
operandA less operandB	PL ≠ OV
operandA equal operandB	ZR=1
operandA greater operandB	PL = OV

Größenvergleich mittels SUB Befehl

```
sub operandA, operandB
```

Vorzeichenlose Zahlen: "above" und "below"

Aussage	Bedingung
operandA below operandB	CY=1
operandA equal operandB	ZR=1
operandA above operandB	CY=0

Erklärung:

Beim `sub` Befehl wird das Carry als Borrow verwendet

Ein Borrow wird nur benötigt, wenn `operandA` kleiner als `operandB` ist

Größenvergleich mittels SUB Befehl

```
sub operandA, operandB
```

Vorzeichenbehaftete Zahlen: "less" und "greater"

Aussage	Bedingung
operandA less operandB	PL ≠ OV
operandA equal operandB	ZR=1
operandA greater operandB	PL = OV

Zur Erklärung betrachten wir den Fall, dass `operandA` größer als `operandB` ist:

$PL = OV = 0$, kein Overflow ist aufgetreten (d.h. keine unerwarteter Vorzeichenwechsel, Ergebnis stimmt) und Ergebnis ist positiv

$PL = OV = 1$, Overflow ist aufgetreten (d.h. unerwarteter Vorzeichenwechsel, Ergebnis falsch) und das Ergebnis ist negativ ("negativer Overflow").

Der negative Overflow bedeutet, dass das Ergebnis eigentlich positiv sein sollte und daher `operandA` größer als `operandB` ist.

Obwohl das Berechnungsergebnis falsch ist (angezeigt durch den Overflow), kann trotzdem ein Größenvergleich durchgeführt werden

Größenvergleich mittels SUB Befehl

Beispiel für den Größenvergleich von vorzeichenbehafteten Zahlen:

```
void main() {
    __asm {
        mov eax, -1
        mov ecx, -2
        sub eax, ecx          ; // OV = 0, PL = 0, ZR = 0 -> eax > ecx
        mov eax, -2
        mov ecx, -1
        sub eax, ecx          ; // OV = 0, PL = 1, ZR = 0 -> eax < ecx
        mov eax, 7FFFFFFFh    ; // largest positive signed integer
        mov ecx, -1
        sub eax, ecx          ; // OV = 1, PL = 1, ZR = 0 -> eax > ecx
        mov eax, 80000000h    ; // largest negative signed integer
        mov ecx, 1
        sub eax, ecx          ; // OV = 1, PL = 0, ZR = 0 -> eax < ecx
        mov eax, 1
        mov ecx, 1
        sub eax, ecx          ; // OV = 0, PL = 0, ZR = 1 -> eax == ecx
    }
}
```

Quelldatei: [main.cpp](#)

Der Assemblerbefehl CMP

Da für einen Vergleich nur die Flags nach der Subtraktion eine Rolle spielen, nicht aber das Ergebnis, gibt es folgenden Maschinenbefehl:

```
cmp operandA, operandB
```

Dieser Befehl setzt genau die Flags, die ein analoger Sub-Befehl setzen würde

Der wesentliche Unterschied ist, dass der operandA unverändert bleibt

Die Auswertung der gesetzten Flags erfolgt meist durch einen direkt nachfolgenden bedingten Sprung-Befehl.

Sprungbefehle und Sprungmarken

Assemblerbefehle werden in der Reihenfolge ausgeführt, in der sie im Text erscheinen, es sei denn, es handelt sich um einen Sprungbefehl

Ein solcher bewirkt die Fortsetzung des Programms an einer beliebigen anderen Stelle

Das Argument eines Sprungbefehls ist das Ziel des Sprungs, das über eine Adresse oder eine Sprungmarke (engl. "label") angegeben wird

Die Sprungmarke wird vom Assembler in eine Adresse umgesetzt

Eine Sprungmarke, die als Sprungziel dienen soll, muss mit einem Doppelpunkt abgeschlossen werden

Als Sprungmarke dürfen natürlich keine Wörter verwendet werden, die bereits in der Assemblersprache anderweitig genutzt werden

Die Sprungbefehle realisieren einen Sprung, in dem der Befehlszähler EIP auf die Zieladresse gesetzt wird

Der Assemblerbefehl JMP

Für den unbedingten Sprung gibt es den JMP-Befehl (engl. "jump")

```
jmp sprungziel
```

Sprungziel kann eine Sprungmarke oder eine Adresse sein

Beispiel:

```
int main() {
    __asm {
        mov eax, 3
        jmp Important
        mov eax, 33
    Important: mov ecx, 6
        add eax, ecx ; // eax = ?
    }
    return 0;
}
```

Quelldatei: [main.cpp](#)

Assemblerbefehle für bedingte Sprünge

Die bedingten Sprünge werten die Flags aus, die durch vorangegangene Befehle gesetzt wurden

Wenn die entsprechende Bedingung für den Sprung erfüllt ist, wird der Sprung ausgeführt, ansonsten der nächste Befehl

Die Bedingung ist in die Mnemonik kodiert. Es gibt mehrere Mnemoniks mit der gleichen Funktion (nicht alle sind hier aufgeführt)

Befehle für bedingte Sprünge, die das Zero-Flag auswerten sind:

Befehl	Bedeutung	Bedingung für Flags
JZ	Spring, wenn null ("jump if zero")	ZR=1
JE	Spring, wenn gleich ("jump if equal")	ZR=1
JNZ	Spring, wenn nicht null ("jump if not zero")	ZR=0
JNE	Spring, wenn nicht gleich ("jump if not equal")	ZR=0

Assemblerbefehle für bedingte Sprünge

Bedingte Sprünge nach dem Vergleich von vorzeichenlosen Zahlen:

Befehl	Bedeutung	Bedingung
JB	Spring, wenn niedriger ("jump if below")	CY=1
JNAE	Spring, wenn nicht höher oder gleich ("jump if not above or equal")	CY=1
JAE	Spring, wenn höher oder gleich ("jump if above or equal")	CY=0
JNB	Spring, wenn nicht niedriger ("jump if not below")	CY=0
JBE	Spring, wenn niedr. gleich ("jump if below equal")	CY=1 oder ZR=1
JNA	Spring, wenn nicht höher ("jump if not above")	CY=1 oder ZR=1
JA	Spring, wenn höher ("jump if above")	CY=0 und ZR=0
JNBE	Spring, wenn nicht niedriger oder gleich ("jump if not below or equal")	CY=0 und ZR=0

Assemblerbefehle für bedingte Sprünge

Bedingte Sprünge nach dem Vergleich von vorzeichenbehafteten Zahlen:

Befehl	Bedeutung	Bedingung für Flags
JG	Spring, wenn größer ("jump if greater")	ZR=0 und PL = OV
JNLE	Spring, wenn nicht kleiner oder gleich ("jump if not less or equal")	ZR=0 und PL = OV
JLE	Spring, wenn kleiner oder gleich ("jump if less or equal")	ZR=1 oder PL ≠ OV
JNG	Spring, wenn nicht größer ("jump if not greater")	ZR=1 oder PL ≠ OV
JL	Spring, wenn kleiner ("jump if less")	PL ≠ OV
JNGE	Spring, wenn nicht größer oder gleich ("jump if not greater or equal")	PL ≠ OV
JGE	Spring, wenn größer oder gleich ("jump if greater or equal")	PL = OV
JNL	Spring, wenn nicht kleiner ("jump if not less")	PL = OV

Thorsten Thormählen 44 / 76

Assemblerbefehle für bedingte Sprünge

Bedingte Sprünge basierend auf bestimmten Flags:

Befehl	Bedeutung	Bedingung
JC	Spring, wenn Carry gesetzt ("jump if carry")	CY=1
JNC	Spring, wenn Carry nicht gesetzt ("jump if not carry")	CY=0
JO	Spring, wenn Overflow gesetzt ("jump if overflow")	OV=1
JNO	Spring, wenn Overflow nicht gesetzt ("jump if not overflow")	OV=0
JS	Spring, wenn Sign gesetzt ("jump if sign")	PL=1
JNS	Spring, wenn Sign nicht gesetzt ("jump if not sign")	PL=0

Assemblerbefehle für bedingte Sprünge

Bedingte Sprünge basierend auf dem (E)CX-Register:

Befehl	Bedeutung	Bedingung
JCXZ	Spring, wenn CX-Register null ist ("jump if register CX is zero")	CX=0
JECXZ	Spring, wenn ECX-Register null ist ("jump if register ECX is zero")	ECX=0
LOOP	Dekrementiere (E)CX; springe, wenn (E)CX nicht null ("decrement (E)CX; jump if (E)CX not zero")	(E)CX≠0

Assemblerbefehle für bedingte Sprünge

Beispiel: JNA ("jump if not above")

```
int main() {
    __asm {
        mov eax, 2
        mov ecx, 3
        cmp eax, ecx
        jna ThisIsMyLabel
        mov eax, 33
    ThisIsMyLabel: mov ecx, 6
                    add eax, ecx; // eax = ?
}
return 0;
}
```

Quelldatei: [main.cpp](#)

Assemblerbefehle für bedingte Sprünge

Beispiel: Summe 1+2+3+ + n :

```
#include <stdio.h>
unsigned int smallGauss(unsigned int varA) { // version 1
    __asm
    {
        mov eax, 0
        mov ecx, varA
        cmp ecx, 0
        ExecuteLoop: jz EndLoop
                    add eax, ecx
                    sub ecx, 1
                    jmp ExecuteLoop;
        EndLoop:
    }
}

int main() {
    unsigned int n = 5;
    unsigned int result = smallGauss(n); // call C-function "smallGauss"
    printf( "smallGauss(%d) = %d\n", n, result); // output result to console
    return 0;
}
```

Quelldatei: [main.cpp](#)

Thorsten Thormählen 48 / 76

Die Assemblerbefehle INC und DEC

Die arithmetischen Operationen `inc` und `dec` dienen zum Inkrementieren bzw. Dekrementieren eines Speicher- oder Registerinhaltes um 1

Aufgrund ihrer Geschwindigkeit und Lesbarkeit sind sie einer Addition von 1 mit dem Add-Befehl bzw. Subtraktion von 1 mit dem Sub-Befehl vorzuziehen

Wird der Code aus der vorherigen Implementierung der Funktion `smallGauss` angepasst, ergibt sich:

```
unsigned int smallGauss(unsigned int varA) { // version 2
    __asm
    {
        mov eax, 0
        mov ecx, varA
        cmp ecx, 0
        ExecuteLoop: jz EndLoop
                    add eax, ecx
                    dec ecx
                    jmp ExecuteLoop;
        EndLoop:
    }
}
```

Quelldatei: [main.cpp](#)

Die Assemblerbefehle INC und DEC

Die Befehle `inc` und `dec` beeinflussen die Flags: OV,PL,ZR,AC und PE

Interessant ist, dass das Carry-Flag nicht gesetzt wird

Dies ist nicht nötig, da das Zero-Flag und das Carry-Flag bei Addition von 1 immer den gleichen Wert haben. D.h. der Befehl

```
add ziel, 1
```

setzt genau dann das Carry-Flag, wenn das Ergebnis null ist und dementsprechend das Zero-Flag gesetzt wird (z.B. bei einem 8-Bit-Register für `ziel=FFh`)

Bei Subtrahieren von 1 mit

```
sub ziel, 1
```

wird genau dann das Carry-Flag gesetzt, wenn vorher `ziel=0` war. Auch dies ist leicht zu prüfen, so dass auf das Carry-Flag verzichtet werden kann

Der Assemblerbefehl LOOP

Der Befehl `loop` wurde bereits bei den bedingten Sprüngen erwähnt

```
loop ziel
```

Ziel kann eine Sprungmarke oder eine Adresse sein

Beim Aufruf des Befehls wird das (E)CX-Register dekrementiert und der Sprung ausgeführt, wenn (E)CX ungleich null ist

Ist `ziel` eine 16-Bit Adresse, wird das CX- und bei 32-Bit das ECX-Register betrachtet

Die Funktion `smallGauss` kann somit nochmal verbessert werden: ([main.cpp](#))

```
unsigned int smallGauss(unsigned int varA) { // version 3
    __asm
    {
        mov eax, 0
        mov ecx, varA
        jecxz EndLoop
        ExecuteLoop: add eax, ecx
        loop ExecuteLoop
        EndLoop:
    }
}
```


Der Assemblerbefehl MUL

Der Befehl `mul` führt eine Multiplikation für vorzeichenlose Zahlen aus

Hierbei wird von dem üblichen Schema:

```
op ziel, quelle
```

abgewichen, da das Ergebnis im Allgemeinen nicht in ein Register passt

Stattdessen wird das Ziel, welches mehrere Register umfassen kann, fest vorgegeben und der Mul-Befehl benötigt nur die Quelle als Parameter:

```
mul quelle
```

Welche Register als Ziel verwendet werden hängt von der Bitbreite von `quelle` ab:

Bitbreite	Ziel	Operation
8	AX	$AX = AL * \text{quelle}$
16	DX:AX	$DX:AX = AX * \text{quelle}$
32	EDX:EAX	$EDX:EAX = EAX * \text{quelle}$

Überschreitet das Ergebnis die Bitbreite von `quelle` werden Carry- und Overflow-Flag gesetzt

Der Assemblerbefehl MUL

Beispiel:

```
int main() {
    __asm {
        mov cl, 16          ; // CL=10h (8-bit register)
        mov al, 8           ; // AL=08h
        mul cl              ; // AX=0080h=128           Flags: OV=0, CY=0
        mov cx, 4000h        ; // CX=4000h (16-bit register)
        mov ax, 5000h        ; // AX=5000h
        mul cx              ; // DX:AX= 1400:0000       Flags: OV=1, CY=1
        mov ecx, 7FFFFFFFh  ; // ECX=7FFFFFFFh (32-bit register)
        mov eax, 4            ; // EAX=00000004h
        mul ecx              ; // EDX:EAX=00000001:FFFFFFF0   Flags: OV=1, CY=1
    }
    return 0;
}
```

Quelldatei: [main.cpp](#)

Der Assemblerbefehl DIV

Der Assemblerbefehl `div` führt eine Division aus

```
div quelle
```

Das Ergebnis der Division ist ein ganzzahliger Anteil und ein Rest

Der Divisor wird mit `quelle` angegeben

Der Dividend muss in fest vorgegebenen Registern abgelegt werden. Welche dies sind, ist abhängig von der Bitbreite von `quelle`:

Bitbreite	Dividend	Ganzzahliges Ergebnis	Rest
8	AX	$AL = AX / \text{quelle}$	$AH = AX \bmod \text{quelle}$
16	DX:AX	$AX = DX:AX / \text{quelle}$	$DX = DX:AX \bmod \text{quelle}$
32	EDX:EAX	$EAX = EDX:EAX / \text{quelle}$	$EDX = EDX:EAX \bmod \text{quelle}$

Die Division ist demnach so implementiert, dass sie die exakte Umkehrfunktion einer Multiplikation realisiert

Flags werden nicht verändert

Der Assemblerbefehl DIV

Beispiel:

```
int main() {
    __asm {
        mov cx, 4000h      ; // CX=4000h (16-bit register)
        mov ax, 5000h      ; // AX=5000h
        mul cx             ; // DX:AX=1400:0000           Flags: OV=1, CY=1
        div cx             ; // AX=5000h DX=0000h
        mov dx, 0000h      ;
        mov ax, 0002h      ; // DX:AX=0000:0002h
        mov cx, 10h         ; // CX=10h
        div cx             ; // AX=0000h DX=0002h
        mov dx, 7FFFh      ;
        mov ax, 0000h      ; // DX:AX=7FFF:0000h
        mov cx, 2h          ; // CX=2h
        div cx             ; // result does not fit in AX, throws divide error exception
    }
    return 0;
}
```

Quelldatei: [main.cpp](#)

Der Assemblerbefehl DIV

Beispiel: Größter gemeinsamer Teiler (ggT) mit dem Euklidischen Algorithmus:

```
unsigned int ggT_C(unsigned int a, unsigned int b) {  
    unsigned int res = 0;  
    while(b != 0) { // while b not zero  
        res = (a % b); // res = (a mod b)  
        a = b;  
        b = res;  
    }  
    return a;  
}
```

Z.B. Größter gemeinsamer Teiler von 525 und 399:

$$525 = 1 \cdot 399 + 126$$

$$399 = 3 \cdot 126 + 21$$

$$126 = 6 \cdot 21 + 0$$

Ergebnis: 21

Der Assemblerbefehl DIV

Größter gemeinsamer Teiler (ggT) mit dem Euklidischen Algorithmus in Assembler

```
unsigned int ggT_Asm(unsigned int a, unsigned int b) {
    __asm {
        mov edx, 0      ;
        mov eax, a      ; // EDX:EAX = a
        mov ecx, b      ; // ECX = b
        WhileLoop: cmp ecx, 0      ;
                    jz EndLoop    ; // jump to "done" if ECX == 0
                    div ecx      ;
                    mov eax, ecx    ; // EAX = ECX
                    mov ecx, edx    ; // ECX = (a mod b)
                    mov edx, 0      ; // EDX = 0 for next "div" command
                    jmp WhileLoop   ;
        EndLoop:
    }
}
```

Quelldatei: [main.cpp](#)

Der Assemblerbefehl IMUL

Der Befehl `IMUL` führt eine Multiplikation von vorzeichenbehafteten Zahlen aus

Es gibt drei verschiedene Varianten:

Die Syntax entspricht dem `MUL`-Befehl

```
imul quelle
```

Das Ergebnis der Multiplikation von `quelle` und `ziel` wird in `ziel` gespeichert

```
imul ziel quelle
```

Das Ergebnis der Multiplikation von `operandA` und `konstante` wird in `ziel` gespeichert

```
imul ziel operandA konstante
```

Bei der 2. und 3. Variante werden die niederwertigen Bits des Ergebnisses abgeschnitten falls es nicht in `ziel` passen sollten. Carry- und Overflow-Flag werden in diesem Fall gesetzt

Quelldatei für das Beispiel auf der nächsten Folie: [main.cpp](#)

Der Assemblerbefehl IMUL

```
#include <stdio.h>
int main() {
    int result = 0;
    __asm {
        mov eax, -2
        mov ecx, 800
        imul ecx
        mov result, eax // EDX is ignored here
    }
    printf("result=%d \n", result);
    __asm {
        mov eax, -2
        mov ecx, 800
        imul eax, ecx
        mov result, eax
    }
    printf("result=%d \n", result);
    __asm {
        mov ecx, 800
        imul eax, ecx, -2
        mov result, eax
    }
    printf("result=%d \n", result);
    return 0;
}
```

Thorsten Thormählen 59 / 76

Der Assemblerbefehl IDIV

Der Befehl IDIV führt eine Division von vorzeichenbehafteten Zahlen aus

Der Befehl wird äquivalent zum Befehl DIV verwendet

```
#include <stdio.h>
int main() {
    int result = 0;
    int remainder = 0;
    __asm {
        mov edx, 0
        mov eax, 800
        mov ecx, -3
        idiv ecx
        mov result, eax
        mov remainder, edx
    }
    printf("result=%d \n", result);
    printf("remainder=%d \n", remainder);
    return 0;
}
```

Quelldatei: [main.cpp](#)

Der Assemblerbefehl NEG

Der Assemblerbefehl neg ändert das Vorzeichen einer vorzeichenbehafteten Zahl

Beispiel:

```
int main() {
    __asm {
        mov eax, -2      ; // EAX=FFFFFFFEh=-2
        neg eax         ; // EAX=00000002h= 2
        mov al, 254     ; // AL=FEh=-2
        neg al          ; // AL=02h= 2
    }
}
```

Quelldatei: [main.cpp](#)

Die Assemblerbefehle AND, OR, NOT

Die Assemblerbefehle `and`, `or`, `und` `not` führen elementare logische Grundoperationen aus, wie diese aus der booleschen Algebra bekannt sind

Konjunktion (AND): $y = a \wedge b$

a	b	$y = a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

Disjunktion (OR): $y = a \vee b$

a	b	$y = a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Negation (NOT): $y = \neg a$

a	$y = \neg a$
0	1
1	0

Die Assemblerbefehle AND, OR, NOT

Dabei wird jedes Bit einzeln betrachtet

Das Carry-Flag `cy` und Overflow-Flag `of` werden auf null gesetzt. Die Flags `PL`, `ZR` und `PE` werden gemäß des Ergebnisses gesetzt. Das Aux. Carry `AC` wird nicht beeinflusst.

Beispiel:

```
int main() {
    unsigned int a = 0x000078F1;
    unsigned int b = 0x00007F8F;
    __asm {
        mov eax, a      ; // EAX=000078F1h
        not eax        ; // EAX=FFFF870Eh
        and eax, b     ; // EAX=0000070Eh
        mov edx, b     ; // EDX=0000070Eh
        not edx        ; // EDX=FFFF8070h
        and edx, a     ; // EDX=00000070h
        or eax, edx    ; // EAX=0000077Eh
    }
    return 0;
}
```

Frage: Was wurde hier gerade umständlich implementiert?

Der Assemblerbefehl XOR

Antwort: XOR

```
int main() {
    unsigned int a = 0x000078F1;
    unsigned int b = 0x00007F8F;
    __asm {
        mov eax, a      ; // EAX=000078F1h
        not eax         ; // EAX=FFFF870Eh
        and eax, b      ; // EAX=0000070Eh
        mov edx, b      ; // EDX=0000070Eh
        not edx         ; // EDX=FFFF8070h
        and edx, a      ; // EDX=00000070h
        or eax, edx     ; // EAX=0000077Eh
    }

    // simple alternative
    __asm {
        mov eax, a      ; // EAX=000078F1h
        xor eax, b      ; // EAX=0000077Eh
    }
    return 0;
}
```

Quelldatei: [main.cpp](#)

Die Assemblerbefehle SHL und SHR

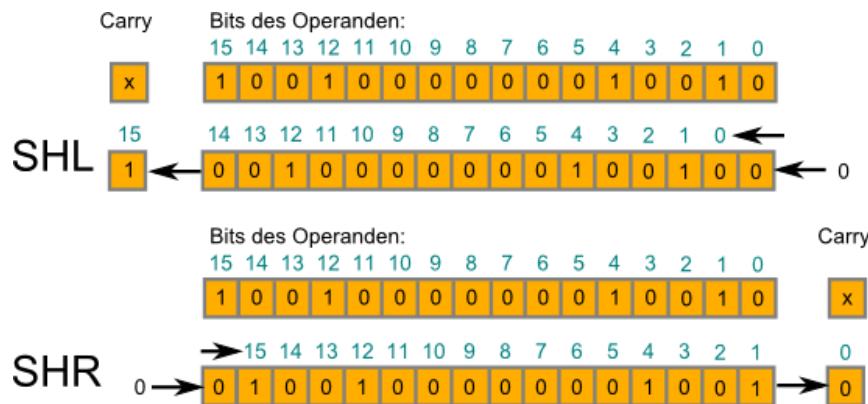
Der Befehl `shl` ("Shift logical left") und `shr` ("Shift logical right") schieben das Ziel um die angegebene Anzahl an Bits nach links bzw. rechts

```
shl ziel, anzahl  
shr ziel, anzahl
```

Eine Verschiebung nach links um n Bits entspricht einer Multiplikation mit 2^n

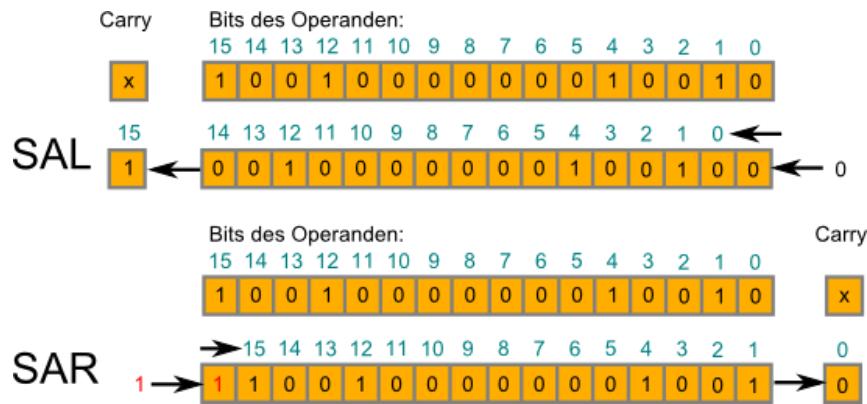
Eine Verschiebung nach rechts entspricht einer ganzzahligen Division mit 2^n

Das jeweils letzte herausgeschobene Bit landet im Carry:



Die Assemblerbefehle SAL und SAR

Der Befehle `sal` ("Shift arithmetic left") und `sar` ("Shift arithmetic right") entsprechen den Befehlen `shl` und `shr`. Der einzige Unterschied ist, dass bei `sar` statt einer 0 eine 1 hineingeschoben wird



Der Assemblerbefehl XCHG

Der Befehl `xchg` ("exchange") tauscht den Inhalt der beiden Operanden aus

Beispiel:

```
int main() {
    __asm {
        mov eax, 1      ; // EAX=1
        mov edx, 2      ; // EDX=2
        xchg eax, edx ; // EDX=1, EAX=2
    }
    return 0;
}
```

Quelldatei: [main.cpp](#)

Die Assemblerbefehle BT, BTS, BTR, BTC

Die "Bit Test"-Befehle, `bt`, `bts`, `btr` und `btc`, kopieren das durch `bitIndex` indizierte Bit ins Carry-Flag und modifizieren dieses Bit in `ziel` entsprechend der Semantik der Mnemonik

```
bt ziel, bitIndex
```

Befehl	Wirkung auf das Carry cy	Wirkung auf das indizierte Bit
<code>bt ("bit test")</code>	<code>cy=ziel[bitIndex]</code>	<code>ziel[bitIndex]</code> unbeeinflusst
<code>bts ("bit test and set")</code>	<code>cy=ziel[bitIndex]</code>	<code>ziel[bitIndex]=1</code>
<code>btr ("bit test and reset")</code>	<code>cy=ziel[bitIndex]</code>	<code>ziel[bitIndex]=0</code>
<code>btc ("bit test and complement")</code>	<code>cy=ziel[bitIndex]</code>	<code>ziel[bitIndex] = not(ziel[bitIndex])</code>

Die Assemblerbefehle CLC und STC

Die Assemblerbefehle `clc` ("clear carry") und `stc` ("set carry") löschen bzw. setzen das Carry-Flag

Beispiel:

```
int main() {
    __asm {
        stc          ; // CY=1
        clc          ; // CY=0
        mov ax, 00FFh ; // AX=00FFh
        bt  ax, 0      ; // CY=1
        bt  ax, 7      ; // CY=1
        bt  ax, 8      ; // CY=0
        bts ax, 8     ; // CY=0, AX=01FFh
    }
    return 0;
}
```

Quelldatei: [main.cpp](#)

Der Assemblerbefehl NOP

Der Assemblerbefehl `nop` ("no operation") führt keine Operation aus

Er dient z.B. dazu, die Ablaufgeschwindigkeit von Maschinespracheprogrammen zu beeinflussen, da ein NOP-Befehl jeweils eine definierte Verzögerung erzeugt

Weitere Assemblerbefehle der x86 Prozessorfamilie

Die hier beispielhaft gezeigten Befehle sind natürlich nur ein Auszug aus den insgesamt ca. 200 Mehrzweckbefehlen und weiteren 300 Spezialbefehlen

Für viele einfache Assemblerprogramme sind die hier gezeigten Befehle jedoch bereits ausreichend

Einige weitere Befehle werden in den folgenden Kapiteln vorgestellt

Die komplette Dokumentation der x86 Architektur steht auf den Webseiten von Intel zur Verfügung:

[Intel 64 and IA-32 Architectures Software Developer's Manual](#)

[Volume 1: Basic Architecture](#)

[Volume 2: Instruction Set Reference, A-Z](#)

[Volume 3: System Programming Guide](#)

Die x86-Befehlsreferenz

Die Dokumentation eines Assembler-Befehls besteht jeweils aus einer großen Tabelle, die zeigt für welche Operanden der Befehl vorhanden ist

Hier z.B. die Tabelle für den Befehl add (Seite 3-31 der [Befehlsreferenz](#)):

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 /b	ADD AL, imm8	I	Valid	Valid	Add imm8 to AL.
05 /w	ADD AX, imm16	I	Valid	Valid	Add imm16 to AX.
05 /d	ADD EAX, imm32	I	Valid	Valid	Add imm32 to EAX.
REX.W + 05 /d	ADD RAX, imm32	I	Valid	N.E.	Add imm32 sign-extended to 64-bits to RAX.
80 /0 /b	ADD r/m8, imm8	M	Valid	Valid	Add imm8 to r/m8.
REX + 80 /0 /b	ADD r/m8*, imm8	M	Valid	N.E.	Add sign-extended imm8 to r/m8.
81 /0 /w	ADD r/m16, imm16	M	Valid	Valid	Add imm16 to r/m16.
81 /0 /d	ADD r/m32, imm32	M	Valid	Valid	Add imm32 to r/m32.
REX.W + 81 /0 /d	ADD r/m64, imm32	M	Valid	N.E.	Add imm32 sign-extended to 64-bits to r/m64.
83 /0 /b	ADD r/m16, imm8	M	Valid	Valid	Add sign-extended imm8 to r/m16.
83 /0 /b	ADD r/m32, imm8	M	Valid	Valid	Add sign-extended imm8 to r/m32.
REX.W + 83 /0 /b	ADD r/m64, imm8	M	Valid	N.E.	Add sign-extended imm8 to r/m64.
00 /r	ADD r/m8, r8	MR	Valid	Valid	Add r8 to r/m8.
REX + 00 /r	ADD r/m8*, r8*	MR	Valid	N.E.	Add r8 to r/m8.
01 /r	ADD r/m16, r16	MR	Valid	Valid	Add r16 to r/m16.
01 /r	ADD r/m32, r32	MR	Valid	Valid	Add r32 to r/m32.
REX.W + 01 /r	ADD r/m64, r64	MR	Valid	N.E.	Add r64 to r/m64.
02 /r	ADD r8, r/m8	RM	Valid	Valid	Add r/m8 to r8.
REX + 02 /r	ADD r7*, r/m8*	RM	Valid	N.E.	Add r/m8 to r8.
03 /r	ADD r16, r/m16	RM	Valid	Valid	Add r/m16 to r16.
03 /r	ADD r32, r/m32	RM	Valid	Valid	Add r/m32 to r32.
REX.W + 03 /r	ADD r64, r/m64	RM	Valid	N.E.	Add r/m64 to r64.

Assembler

add eax, 0A0B0C0Dh

Maschinensprache

05 0D 0C 0B 0A

Opcode imm32
(Little Endian)

Die x86-Befehlsreferenz

Es fällt auf, dass je nach Art der angegebenen Operanden ein anderer Opcode verwendet wird

D.h. der verwendeten Maschinenbefehl (Opcode) bestimmt sich zum Einen aus der Mnemonik, ist aber auch abhängig von den Operanden

Abkürzungen:

`r8, r16, r32` Operand ist eine 8-, 16-, bzw. 32-Bit-Mehrzweckregister

`m8, m16, m32` Operand ist eine Speicheradresse (engl. "memory")

`m/r8, m/r16, m/r32` Operand ist ein Register oder eine Speicheradresse

`imm8, imm16, imm32` Operand ist eine konstante vorzeichenbehaftete Zahl (engl. "immediate value")

Eine komplette Liste der Abkürzungen ist in Kapitel 3.1.1.3 (Seite 3-5) der [Befehlsreferenz](#) angegeben

Neben der Tabelle und der Beschreibung eines Befehls ist für einen Programmierer besonders der Abschnitt über die veränderten Flags ("Flags Affected") des jeweiligen Befehls wichtig

Verkürzte Befehlsreferenz für die Klausur (1 von 2)

Assemblerbefehl	Beschreibung
ADC ziel, quelle	Addiere mit Übertrag aus vorangegangener Addition
ADD ziel, quelle	ziel = ziel + quelle
AND ziel, quelle	Logische Und-Verknüpfung von ziel und quelle
CMP x, y	Vergleiche x und y
DEC x	Dekrementiere x
DIV op	Teile durch op
IDIV op	Vorzeichenbehaftete, ganzzahlige Division
IMUL op	Vorzeichenbehaftete, ganzzahlige Multiplikation
INC x	Inkrementiere x
JA sprungziel	Springe, wenn größer
JAE sprungziel	Springe, wenn größer oder gleich
JB sprungziel	Springe, wenn kleiner
JBE sprungziel	Springe, wenn kleiner oder gleich
JECXZ sprungziel	Springe, wenn ECX gleich 0
JE sprungziel	Springe, wenn gleich
JMP sprungziel	Unbedingter Sprung

Thorsten Thormählen 74 / 76

Verkürzte Befehlsreferenz für die Klausur (2 von 2)

Assemblerbefehl	Beschreibung
J0 sprungziel	Springe, wenn Überlauf
JZ sprungziel	Springe, wenn gleich 0
LOOP sprungziel	Dekrementiere ECX, springe falls ECX ungleich 0
MOV ziel, quelle	Kopiere quelle nach ziel
MUL op	Multipliziere mit op
NEG ziel	Negiere Wert in ziel mit Zweierkomplement
NOT ziel	Logische Negation von ziel
OR ziel, quelle	Logische Oder-Verknüpfung von ziel und quelle
POP ziel	Hole obersten Wert vom Stack
PUSH quelle	Lege Wert aus quelle auf Stack
SAL ziel, schrittzahl	Arithmetische, bitweise Verschiebung nach links
SAR ziel, schrittzahl	Arithmetische, bitweise Verschiebung nach rechts
SHL ziel, schrittzahl	Logische, bitweise Verschiebung nach links
SHR ziel, schrittzahl	Logische, bitweise Verschiebung nach rechts
SUB ziel, quelle	ziel = ziel - quelle
XOR ziel, quelle	Logische XOR-Verknüpfung von ziel und quelle

Thorsten Thormählen 75 / 76

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[\[Impressum\]](#) , [\[Datenschutz\]](#) , []

Thorsten Thormählen 76 / 76

Technische Informatik I

Unterprogramme und Adressierung

Thorsten Thormählen
28. Januar 2020
Teil 10, Kapitel 2

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Unterprogramme in Maschinensprache

Stapelspeicher (Stack)

Speicheradressierung

Operationen auf Speicherblöcken

Unterprogramme in Maschinensprache

Um ein Unterprogrammen (auch "Subroutine", "Prozedur" oder "Funktion" genannt) in Assembler aufzurufen, wird der Befehl `call` verwendet

```
call ziel
```

Das Ziel bezeichnet dabei die Adresse (bzw. ein Label im Assembler-Code), an der sich das Unterprogramm im Speicher befindet

Der Befehl `ret` (engl. "return") steht am Ende des Unterprogramms und sorgt dafür, dass zum aufrufenden Programm zurückgekehrt und dieses weiter abgearbeitet wird

Um diese Sprünge auszuführen, modifizieren die Befehle `call` und `ret` den Befehlszähler EIP

Unterprogramme in Maschinensprache

Beispiel:

```
int main() {
    int counter = 0;
    __asm {
        call Subroutine_IncrementCounter ; // counter = 1
        call Subroutine_IncrementCounter ; // counter = 2
        call Subroutine_IncrementCounter ; // counter = 3
        jmp EndMainProgram

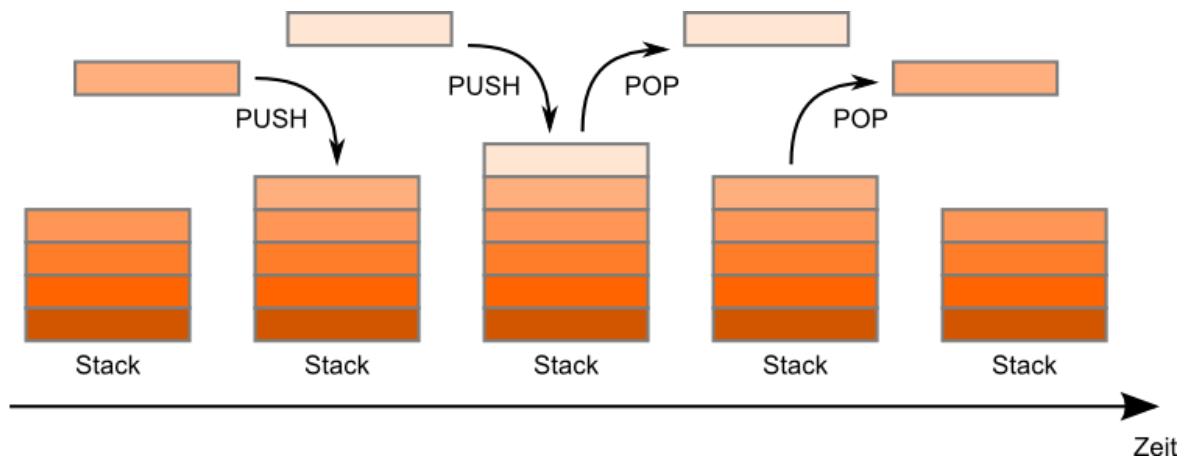
        Subroutine_IncrementCounter:
            mov eax, counter
            inc eax
            mov counter, eax
            ret

        EndMainProgram :
    }
}
```

Quelldatei: [main.cpp](#)

Um zu verstehen, was bei dem Aufruf eines Unterprogramms genau passiert, wird zunächst erläutert, was unter einem "Stack" verstanden wird

Stapelspeicher (Stack)



Auf einen **Stapelspeicher** (engl. "Stack") können Elemente abgelegt werden und später wieder entnommen werden

Elemente werden übereinander gestapelt und können nur in umgekehrter Reihenfolge wieder entnommen werden ("Last-In-First-Out")

Dabei gibt es nur zwei Operationen:

PUSH legt ein Element auf den Stapel

POP entnimmt das oberste Element vom Stapel

Die Assemblerbefehle PUSH und POP

Der Assembler-Befehl `push` legt den Inhalt von `quelle` auf den Stapel

```
push quelle
```

Der Assembler-Befehl `pop` entnimmt das oberste Element vom Stapel und kopiert es in `ziel`

```
pop ziel
```

Beispiel: Wiederherstellen von Registerinhalten

```
Subroutine_IncrementCounter:  
    push ebx      // push content of ebx to stack  
    mov ebx, counter  
    inc ebx  
    mov counter, ebx  
    pop ebx      // pop content from stack to ebx  
    ret
```

Quelldatei: [main.cpp](#)

Stapelspeicher und die Register BP und SP

Jedes Programm hat einen Stapelspeicher, der im RAM abgelegt ist

An welcher Speicheradresse sich der Stapelspeicher befindet, ist in den Registern **BP** und **SP** angegeben (bzw. in den entsprechenden 32-Bit Versionen: **EBP** und **ESP**):

BP: Anfangsadresse des Stapelspeichers

SP: Adresse des obersten Elements des Stapelspeichers

Der Stack wächst in Richtung kleinerer Speicheradressen

In MS Visual Studio kann der Stapelspeicher leicht angezeigt werden:

Menü → Debuggen → Fenster → Arbeitsspeicher
und dort als Adresse "ESP" eintragen

Memory 1			
Address:	Columns: 4		
0x0038F944	00 e0 fd 7e	.àý~	
0x0038F948	00 00 00 00	
0x0038F94C	9c f9 38 00	æù8.	
0x0038F950	59 15 1f 00	Y...	
0x0038F954	01 00 00 00	
0x0038F958	e0 ac 7b 00	à¬{.	
0x0038F95C	30 b9 7b 00	0.{.	

Der Stapspeicher bei Unterprogrammaufrufen

Bei der Ausführung von Unterprogrammen spielt der Stapspeicher (engl. "Stack") eine wichtige Rolle

Der Stack wird verwendet, um sich die Rücksprungadresse zu merken, an die der Ret-Befehl des Unterprogramms zurückkehren soll

Dazu legt ein Call-Befehl per PUSH die Adresse des ihm nachfolgenden Befehls (aktueller Befehlszähler + 1 Befehl) auf den Stack und führt dann durch Setzen des Befehlszählers EIP den Sprung in das Unterprogramm aus

Um aus dem Unterprogramm zurückzukehren, entnimmt der nächste Ret-Befehl per POP die Rücksprungadresse und lädt diese in den Befehlszähler

Der Stapelspeicher bei Unterprogrammaufrufen

Um dies zu verdeutlichen, wird nochmal das vorherige Beispiel betrachtet (zu Beginn jeder Zeile steht die Speicheradresse des Befehls):

```
00881033h    call Subroutine_IncrementCounter; // push 00881038h; set EIP=088103Ah
00881038h    jmp EndMainProgram           ; // set EIP=00881044h
Subroutine_IncrementCounter:
0088103Ah    push ebx      ; // push content of ebx to stack
0088103Bh    mov ebx, counter
0088103Eh    inc ebx
0088103Fh    mov counter, ebx
00881042h    pop ebx      ; // pop content of ebx from stack
00881043h    ret         ; // pop 00881038h; set EIP=00881038h
EndMainProgram:
00881044h
```

Quelldatei: [main.cpp](#)

Was würde passieren, wenn der `pop` Befehl an Adresse 00881042h auskommentiert würde

Antwort: der Befehlszähler `EIP` würde mit dem Inhalt von `EBX` geladen, d.h. das Programm würde an einer unsinnigen Stelle im Speicher weiter ausgeführt. Dies resultiert bei Windows typischerweise in einem Speicherzugriffsfehler (engl. "Access violation")

Der Stapspeicher bei Unterprogrammaufrufen

Durch Verwendung des Stacks sind auch verschachtelte oder (wie hier gezeigt) rekursive Unterprogrammaufrufe möglich (Quelldatei: [main.cpp](#)):

```
int main() {
    int counterForward = 0;
    int counterBackward = 0;
    __asm {
        call Subroutine_IncrementCounter    ;
        jmp EndMainProgram
    Subroutine_IncrementCounter:
        mov eax, counterForward
        inc eax
        mov counterForward, eax
        cmp eax, 5
        jz Subroutine_IncrementCounterPart2
        call Subroutine_IncrementCounter    ; // recursive call of subroutine
    Subroutine_IncrementCounterPart2:
        mov eax, counterBackward
        inc eax
        mov counterBackward, eax
        ret
    EndMainProgram :
    }
}
```


Sichern von Registerinhalten auf dem Stapelspeicher

Bei den Allzweck-Registern gibt es zwei Gruppen:

Caller-Saved-Register: EAX, ECX und EDX

Callee-Saved-Register: EBX, ESI und EDI (sowie EBP und ESP)

Bei Unterprogrammaufrufen ist es eine Konvention, dass die Caller-Saved-Register in einem Unterprogramm verändert werden können, während die Inhalte von Callee-Saved-Registern erhalten bleiben

Möchte ein Unterprogramm (engl. "Callee") trotzdem Callee-Saved-Register verwenden, muss es die Inhalten vor Gebrauch mit PUSH auf den Stack sichern und nach Gebrauch mit POP wiederherstellen

Möchte ein aufrufendes Programm (engl. "Caller") den Inhalt von Caller-Saved-Registern nach dem Aufruf eines Unterprogramms weiterverwenden, muss es die Inhalten vor dem Call-Befehl mit PUSH auf den Stack sichern und nach Gebrauch mit POP wiederherstellen

Parameterrückgabe mittels Registern

Subroutinen in Hochsprachen haben in der Regel einen Rückgabewert, z.B. in C/C++:

```
int add(int varA, int varB ) {  
    int retVal = varA + varB;  
    return retVal;  
}
```

Bei der Umsetzung in Maschinensprache gilt:

Ein 32-bit Rückgabewert wird im EAX-Register an das aufrufende Programm übergeben

Ein 64-bit Rückgabewert wird in den Registern EDX:EAX an das aufrufende Programm übergeben

Parameterübergabe mittels Stapelspeicher

Äquivalent zur Parameterrückgabe könnte auch bei der Parameterübergabe an Unterprogramme bestimmte Register verwendet werden

Dies hat jedoch den Nachteil, dass nur eine relativ begrenzte Anzahl an Parametern übergeben werden können

Stattdessen werden die Parameter von dem aufrufenden Programm auf den Stack gelegt und von dem Unterprogramm dort ausgelesen

Bei der konkreten Umsetzung in Maschinensprache gibt es viele Möglichkeiten dies zu tun. Im Folgenden wird die so genannte "C Declaration"-Konvention (kurz "cdecl") näher beschrieben

Es gibt aber auch andere Konventionen, z.B. "stdcall" oder "fastcall".

Welche Aufrufkonvention verwendet wird, kann bei Compilern typischerweise eingestellt werden. In MS Visual Studio unter:

Menü → Projekt → Eigenschaften → Konfigurationseigenschaften → C/C++ → Erweitert → Aufrufkonventionen

Parameterübergabe mittels Stapspeicher

Bei der Aufrufkonvention "cdecl" werden die Parameter von rechts nach links auf den Stack gelegt

Beispiel:

```
int add(int varA, int varB ) {  
    return varA + varB;  
}  
  
int main() {  
    int result = add(1, 2);  
    return 0;  
}
```

Quelldatei: [main.cpp](#)

Das heisst in diesem Beispiel würde in `main()` zunächst ein `push 2` und dann ein `push 1` ausgeführt

Wir wollen nun ausprobieren, ob sich der MS Visual Studio C/C++-Compiler an diese Konvention hält

Dazu ist auf der nächsten Folie der disassemblierte Maschinencode dieses Beispiels gezeigt

Parameterübergabe mittels Stapspeicher

```
--- main.cpp -----
 1: int add(int varA, int varB ) {
009C1020 55          push    ebp // push stack base pointer
009C1021 8B EC        mov     ebp,esp // setup new stack frame
 2:   return varA + varB;
009C1023 8B 45 08      mov     eax,[ebp+08h] // access parameters
009C1026 03 45 0C      add     eax,[ebp+0Ch]
 3: }
009C1029 5D          pop    ebp // pop stack base pointer
009C102A C3          ret
--- keine quelldatei ---
009C102B CC          int     3
009C102C CC          int     3
009C102D CC          int     3
009C102E CC          int     3
009C102F CC          int     3
--- main.cpp -----
 4:
 5: int main() {
009C1030 55          push    ebp
009C1031 8B EC        mov     ebp,esp
009C1033 51          push    ecx
 6:   int result = add(1 2).
```

Thorsten Thormählen 17 / 33

Stapelsegmente

Das aufrufende Programm legt demnach die Übergabeparameter von rechts nach links per `push` auf den Stapel

Es ist interessant zu sehen, dass das Unterprogramm die Parameter aber nicht per `pop` vom Stapel holt

Das wäre auch nicht so einfach, da bekanntlich der Call-Befehl die Rücksprungadresse auf den Stack legt und diese somit den direkten Zugriff auf die Parameter per `pop` blockiert

Stattdessen spielt scheinbar das `EBP`-Register eine Rolle:

Der Stapspeicher wird typischerweise in Segmente unterteilt. Jedes Unterprogramm hat sein eigenes Segment

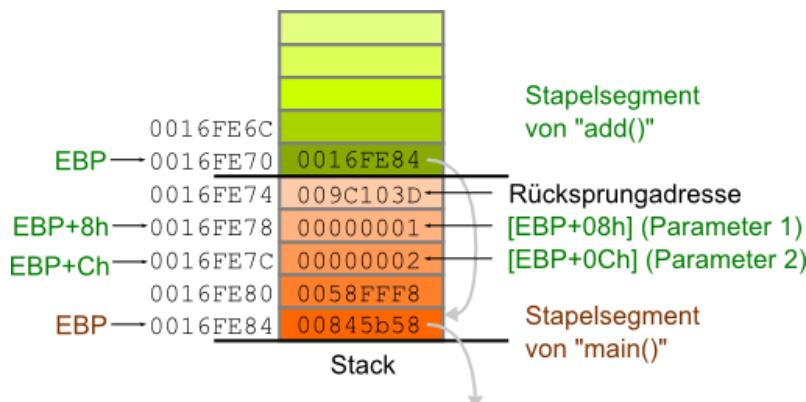
Das `EBP`-Register zeigt immer auf die Anfangsadresse des aktuellen Stapelsegments

Das Unterprogramm verwendet einen feste relative Adresse zum seinem `EBP` um auf die Übergabeparameter zuzugreifen

Stapelsegmente

Im Unterprogramm wird daher wie folgt vorgegangen:

```
1: int add(int varA, int varB ) {  
009C1020 push ebp // legt die alte Segmentanfangsadresse auf den Stack  
009C1021 mov ebp,esp // die aktuelle Stackadresse wird neue Segmentanfangsadresse  
2:   return varA + varB;  
009C1023 mov eax,[ebp+08h] // greift auf den ersten Parameter zu  
009C1026 add eax,[ebp+0Ch] // greift auf den zweiten Parameter zu  
3: }  
009C1029 pop ebp // stellt die alte Segmentanfangsadresse wieder her  
009C102A ret // entfernt die Rücksprungadresse vom Stack
```



Thorsten Thormählen 19 / 33

Parameterübergabe mittels Stapspeicher

Bei der Aufrufkonvention "cdecl" ist es die Aufgabe des aufrufenden Programms den Stack nach einer Parameterübergabe wieder aufzuräumen (bei "stdcall" Aufgabe des Unterprogramms)

Dies könnte in unserem Beispiel durch zwei `pop` Befehle passieren

Eine Alternative, die scheinbar der C/C++ Kompiler von Visual Studio im gezeigten Beispiel bevorzugt, ist die Addition von `ESP` mit 8 (8 Byte, weil 4 Byte pro 32-Bit Übergabeparameter):

```
add esp,8
```

Mit der Verständnis der Aufrufkonvention ist es damit möglich, das erzeugte Maschinenspracheprogramm des Kompilers von Visual Studio für das gezeigt Beispiel komplett nachzuvollziehen

Es wird dadurch auch offensichtlich, dass nur solche Programmteile sich gegenseitig aufrufen können, wenn sie die gleichen Aufrufkonvention verwenden

Direkte Speicheradressierung

Früher, z.B. unter DOS, war es kein Problem, direkt auf eine bestimmte Speicherstelle im Hauptspeicher lesend oder schreibend zuzugreifen

Eine in eckigen Klammern stehende Zahl wird als Adresse interpretiert

```
mov eax, [110h]
```

Dabei war 110h nur eine Offset-Adresse. Die komplette Adresse wurde bei Datenelementen aus dem Inhalt des ds (Daten-Segment) Segmentregister plus dieser Offset-Adresse gebildet, d.h. der oben angegebene Befehl war gleichbedeutend mit

```
mov eax, ds:[110h]
```


Direkte Speicheradressierung

Bei moderneren 32-bit Betriebssystemen, wie Windows oder Linux, sind solche direkten Speicherzugriffe nicht erlaubt, da sie andere Programme oder das Betriebssystem stören könnten

Jedes Benutzerprogramm erhält seinen eigenen Adressraum, in dem es Daten schreiben und lesen kann ("protected mode")

Wie dieser Adressraum auf den physikalisch vorhandenen Speicher abgebildet wird, ist Aufgabe des Betriebssystems

Innerhalb des eigenen Adressraums gibt es ein flaches Speichermodell. Ein Programmierer muss sich nicht um Segmentregister kümmern, wie ds (Daten-Segment), ss (Stapel-Segment) oder cs (Code-Segment)

Im Folgenden wird daher nicht mehr zwischen Offset-Adresse und Adresse unterschieden, gemeint ist immer die "virtuelle" Adresse im Adressraum des Programms

Direkte Speicheradressierung

Beim Inline-Assembler werden die Variablen typischerweise in der Hochsprache definiert und mit dem Variablenamen im Assemblercode auf die Speicheradresse zugegriffen

Dabei ist es egal, ob der Variablenname in eckigen Klammern gesetzt ist oder nicht. Gemeint ist immer das Lesen bzw. Schreiben des Werts an der Speicheradresse, die der Variablen zugeordnet ist

Wird vor den Variablenname der Bezeichner `offset` gesetzt ist die Speicheradresse selbst gemeint und nicht der gespeicherte Wert

Beispiel:

```
int myVar = 42;
int main() {
    __asm {
        mov eax, myVar          // move the content of myVar to eax
        mov eax, [myVar]         // move the content of myVar to eax
        mov eax, offset myVar   // move the address of myVar to eax
    }
}
```

Quelldatei: [main.cpp](#)

Speicheradressierung von Feldern

Beim der direkten Adressierung von Elementen in Feldern (engl. "Arrays") kann der Versatz (in Bytes) zur Startadresse des Felds angegeben werden

Beispiel:

```
#include <stdio.h>
int main() {
    int studentIds[] = {8012, 12341, 12412, 13343, 13423}; // create C-Array
    for(int i=0; i<5; i++) printf("studentIds[%d]: %d\n", i, studentIds[i]); // output

    __asm {
        mov eax, [studentIds+4] // move 12341 to eax
        mov ecx, [studentIds+12] // move 13343 to ecx
        mov [studentIds+4], ecx // move 13343 to studentIds[1]
        mov [studentIds+12], eax // move 12341 to studentIds[3]
    }

    for(int i=0; i<5; i++) printf("studentIds[%d]: %d\n", i, studentIds[i]); // output
    return 0;
}
```

Quelldatei: [main.cpp](#)

Speicheradressierung von Feldern

Dabei gibt es folgende äquivalente Notationen:

```
#include <stdio.h>
int main() {
    int studentIds[] = {8012, 12341, 12412, 13343, 13423}; // create C-Array
    for(int i=0; i<5; i++) printf("studentIds[%d]: %d\n", i, studentIds[i]); // output

    __asm {
        mov eax, studentIds+4      // move 12341 to eax
        mov ecx, studentIds[12]    // move 13343 to ecx
        mov studentIds[4], ecx     // move 13343 to studentIds[1]
        mov studentIds+12, eax     // move 12341 to studentIds[3]
    }

    for(int i=0; i<5; i++) printf("studentIds[%d]: %d\n", i, studentIds[i]); // output
    return 0;
}
```


Indirekte Speicheradressierung

Bei der indirekten Speicheradressierung wird die Adresse durch ein Register übergeben und steht somit erst zur Laufzeit fest

Dabei wird der Inhalt des Registers als Adresse interpretiert

Beispiel:

```
mov eax, [ebx]
```

Steht z.B. im EBX-Register der Wert 110h, dann würde der Wert an der Speicheradresse 110h in das EAX-Register kopiert

Der Assemblerbefehl LEA

Der Assemblerbefehl `lea` (engl. "load effective address") dient dazu die Adresse einer Variablen zu ermitteln

```
lea ziel quelle
```

Die Adresse der in `quelle` angegebene Variablen, wird in `ziel` geladen

Beispiel:

```
int main()
{
    int num = 8;
    __asm {
        mov eax, 4;
        lea ecx, num      ;// ecx = address of variable "num";
        add eax, [ecx]    ;// add 4 + 8
    }
}
```

Quelldatei: [main.cpp](#)

Der Assemblerbefehl LEA

Mit Hilfe des `lea`-Befehls und bei bekannten Aufrufkonventionen der C-Funktionen, kann die `printf` C-Funktion nun auch aus dem Inline-Assembler aufgerufen werden (Quelldatei: [main.cpp](#)):

```
#include <stdio.h>
char format[] = "%s %s %d\n";
char hello[] = "Hello";
char world[] = "world: num=";
int num = 8;
int main() {
    __asm {
        push num          ;// pushes parameter to stack
        lea   eax, world ;// load address of array "world" to eax
        push eax          ;// pushes parameter to stack
        lea   eax, hello ;// load address of array "world" to eax
        push eax          ;// pushes parameter to stack
        lea   eax, format ;// load address of array "format" to eax
        push eax          ;// pushes parameter to stack
        mov   eax, printf ;// get address of C-printf function
        call eax          ;// calls C-printf function
        add esp, 16         ;// clean up the stack
    }
}
```


Indirekte Speicheradressierung

Das Register `bx` bzw. `ebx` ("Base") ist dafür vorgesehen, um in Felder zu indizieren

Neben einfachen Indexangaben, wie `[studentIds+ebx]` sind auch einfache konstante Ausdrücke zugelassen wie z.B. `[studentIds+4*ebx-4]`

Als Faktoren sind nur die typischen Größen von Datentypen zulässig: 1, 2, 4 und 8

Beispiel: Alle Matrikelnummern um eins erhöhen:

```
int main() {
    int studentIds[] = {8012, 12341, 12412, 13343, 13423}; // create C-Array
    __asm {
        mov ebx, 5      ;// set loop counter
        LoopOverIds:
        mov eax, [studentIds+4*ebx-4]      ;// studentIds[4], studentIds[3], ...
        inc eax
        mov [studentIds+4*ebx-4], eax
        dec ebx;
        jnz LoopOverIds
    }
    return 0;
}
```

Quelldatei: [main.cpp](#)

Operationen auf Speicherblöcken

Zur Operation auf Zeichenketten oder Feldern bzw. ganz allgemein auf Speicherblöcken gibt es die `movs` ("Move Data from String to String") und `cmps` ("Compare String") Befehle

Diese gibt es für verschiedene Größen der Datenelemente (siehe Tabelle)

Zur Adressierung von Quelle und Ziel der Datenbewegung müssen die Register `esi` ("Extended Source Index") und `edi` ("Extended Destination Index") verwendet werden

So kopiert beispielsweise der Befehl `movsb` ein Byte von der Adresse in `esi` zur Adresse in `edi`

Danach werden die Register `esi` und `edi` automatisch erhöht oder erniedigt, je nachdem ob das Direction-Flag UP gesetzt ist oder nicht

Zur Manipulation des Direction-Flags gibt es die Befehle:

`cld` ("clear direction") für aufsteigende Abarbeitung

`std` ("set direction") für absteigende Abarbeitung

Befehl	Elementgröße
<code>movsb</code>	Byte (1 Byte)
<code>movsw</code>	Word (2 Bytes)
<code>movsd</code>	Double Word (4 Bytes)
<code>movsq</code>	Quad Word (8 Bytes)
<code>cmpsb</code>	Byte (1 Byte)
<code>cmpvsw</code>	Word (2 Bytes)
<code>cmpsd</code>	Double Word (4 Bytes)
<code>cmpsq</code>	Quad Word (8 Bytes)

Operationen auf Speicherblöcken

Beispiel (Quelldatei: [main.cpp](#)):

```
#include <stdio.h>
int main() {
    char myText[] = "John is the fastest kid in town";
    char myReplace[] = "Bill";

    printf("%s\n", myText);

    __asm {
        cld                ;// clear direction flag
        mov ecx, 4         ;// set loop counter
        lea eax, myReplace
        mov esi, eax       ;// set esi to address of myReplace
        lea eax, myText
        mov edi, eax;      ;// set edi to address of myText
        LoopOver:
        movsb;
        loop LoopOver
    }

    printf("%s\n", myText);
    return 0;
}
```


Operationen auf Speicherblöcken

Ein Wiederholungspräfix `rep` vor einem `movs` Befehl kann den `loop` Befehl ersetzen. Auch bei `rep` wird das `ecx` Register als Zähler verwendet (Quelldatei: [main.cpp](#)) :

```
#include <stdio.h>
int main() {

    char myText[] = "John is the fastest kid in town";
    char myReplace[] = "pet";

    printf("%s\n", myText);

    __asm {
        cld          ;// clear direction flag
        mov ecx, 3   ;// set loop counter
        lea eax, myReplace
        mov esi, eax ;// set esi to address of myReplace
        lea eax, myText+20
        mov edi, eax; ;// set edi to address of myText
        rep movsb;
    }

    printf("%s\n", myText);
    return 0;
}
```


Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) .

Thorsten Thormählen 33 / 33

Technische Informatik I

FPU, MMX, SSE, x86-64

Thorsten Thormählen

30. Januar 2018

Teil 10, Kapitel 3

Dies ist die Druck-Ansicht.

[Aktiviere Präsentationsansicht](#)

Steuerungstasten

- nächste Folie (auch Enter oder Spacebar).
- ← vorherige Folie
- d schaltet das Zeichnen auf Folien ein/aus
- p wechselt zwischen Druck- und Präsentationsansicht
- CTRL + vergrößert die Folien
- CTRL - verkleinert die Folien
- CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top,$ $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \mathcal{B} = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

- Fließkomma-Recheneinheit ("Floating Point Unit", FPU)
- Multi Media Extension (MMX)
- Streaming SIMD Extensions (SSE)
- x86-64 Register

Fließkomma-Recheneinheit

Bisher wurden nur Maschinensprachebefehle für das Rechnen mit ganzen Zahlen besprochen

Sollen Gleitkommazahlen verarbeitet werden, gibt es die Möglichkeit, die erforderlichen Operationen durch eigene Assemblerroutinen mit den bisherigen Befehlen zu implementieren

Die andere Möglichkeit ist, spezielle Gleitkomma-Befehl zu verwenden

Früher (bis zum Intel 80486) gab für die Berechnungen mit Gleitkommazahlen bei Intel die 80x87-Koprozessoren

Heutzutage ist die Fließkomma-Recheneinheit (engl. "Floating Point Unit", FPU) in die CPU integriert



Intel 80387 FPU

[Bildquelle: [Intel 80387 SX-20 SX105](#) by David Olsen, [Creative Commons License](#)]

Thorsten Thormählen 5 / 27

FPU Register

Die Fließkomma-Recheneinheit hat acht 80-bit Register: ST0, ST1, ..., ST6, ST7

Der Assemblerbefehl `f1d` zum Laden eines Registers bezieht sich immer auf ST0

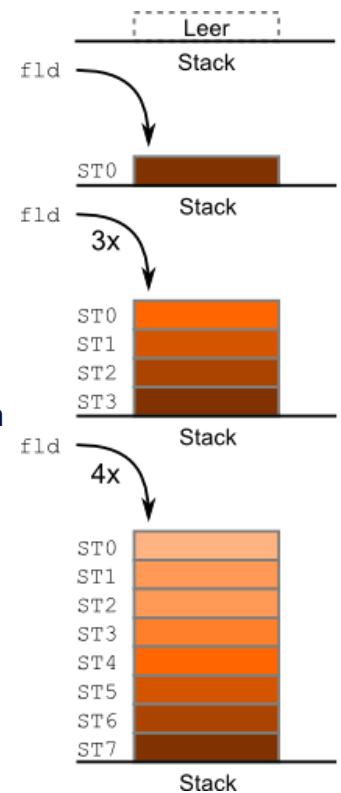
Die Register verhalten sich so ähnlich, wie ein auf maximal acht Elemente begrenzter Stapelspeicher

Dabei bezeichnet ST0 immer das oberste Element des Stapelspeichers

D.h. wird ein neues Element per Assemblerbefehl `f1d` geladen (d.h. auf den Stapel gelegt) werden die Daten an das nächste Register weitergereicht:

ST7=ST6, ST6=ST5, ..., ST1=ST0, ST0=neuer Wert

Anzeigen der FPU-Register in MS Visual Studio:
Menü → Debuggen → Fenster → Register → Kontext-Menü
(rechte Maustaste) → Floating Point



FPU Register

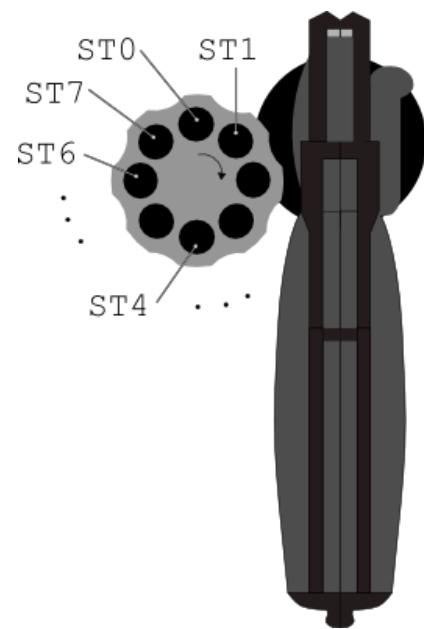
Als Analogie zur Funktionsweise der Register kann auch die Trommel eines Revolvers mit acht freien Kammern herangezogen werden

Der Inhalt von Register ST0 kann immer an der "12 Uhr" Position abgelesen werden, von ST1 bei "1:30 Uhr", von ST2 bei "3 Uhr", usw.

Das Laden von Werten aus dem Speicher per Assemblerbefehl f1d kann nur über die oberste Kammer an der "12 Uhr" Position (also über ST0) erfolgen

Bei einem Ladevorgang dreht sich die Trommel zunächst automatisch im Uhrzeigersinn und der Wert wird anschließend in das Register ST0 geschrieben

Analogie: Die Revolvertrommel dreht sich und die oberste Kammer wird mit einer Kugel geladen



Eine Revolvertrommel als Analogie

FPU Register

Mit der Revolvertrommel als Analogie kann leicht nachvollzogen werden, was passiert, wenn mehr als acht Werte geladen werden. Es gilt: Wird versucht, eine Kammer mit einer Kugel zu laden, die schon besetzt ist, wird die Kugel beschädigt

D.h. bei einem Ladevorgang dreht sich die Trommel zunächst automatisch im Uhrzeigersinn aber der `ST0` Wert wird ungültig. Beispiel (Quelldatei: [main.cpp](#)):

```
int main() {
    float a = 1.1f;
    float b = 0.3f;
    double c = 0.6f;
    __asm__ {
        fld a;// ST0=1.1
        fld b;// ST1=1.1, ST0=0.3
        fld c;// ST2= 1.1, ST1=0.3, ST0=0.6
        fld a;// ST3= 1.1, ST2=0.3, ST1=0.6, ST0=0.1
        fld b;// ST4= 1.1, ST3=0.3, ST2=0.6, ST1=0.1, ST0=0.1
        fld c;// ST5= 1.1, ST4=0.3, ST3=0.6, ST2=0.1, ST1=0.1, ST0=0.6
        fld a;// ST6= 1.1, ST5=0.3, ST4=0.6, ST3=0.1, ST2=0.1, ST1=0.6, ST0=1.1
        fld b;// ST7= 1.1, ST6=0.3, ST5=0.6, ST4=0.1, ST3=0.1, ST2=0.6, ST1=1.1, ST0=0.3
        fld c;// the stack (revolver cylinder) is full
        fld a;// values are still shifted but overwritten ones are invalid
    }
}
```


Der FPU Assemblerbefehl FLD

Zum Laden (engl. "load") des ST0 Registers mit einer Gleitkommazahl kann der Befehl f1d verwendet werden

```
f1d quelle
```

Dabei kann quelle eine Speicheradresse sein, an der die Gleitkommazahl im RAM abgelegt ist, oder ein anderes FPU-Register: ST1, ..., ST7

Beispiel (Quelldatei: [main.cpp](#)):

```
int main() {
    float a = 1.1f;
    float b = 0.3f;
    double c = 0.6f;
    __asm {
        f1d a      ;// ST0=1.1
        f1d b      ;// ST1=1.1, ST0=0.3
        f1d c      ;// ST2= 1.1, ST1=0.3, ST0=0.6
        f1d st(0) ;// ST3= 1.1, ST2=0.3, ST1=0.6, ST0=0.6
        f1d st    ;// ST4= 1.1, ST3=0.3, ST2=0.6, ST1=0.6, ST0=0.6
        f1d st(3) ;// ST5= 1.1, ST4=0.3, ST3=0.6, ST2=0.6, ST1=0.6, ST0=0.3
    }
    return 0;
}
```


Die FPU Assemblerbefehle FST und FSTP

Zum Speichern (engl. "store") des ST0 Register als Gleitkommazahl können die Befehle `fst` und `fstp` verwendet werden

```
fst ziel  
fstp ziel
```

Dabei kann `ziel` eine Speicheradresse sein oder ein anderes FPU-Register: ST1, ..., ST7

Während `fst` nur speichert, entfernt `fstp` das oberste Element vom Stack (engl. "store pop").

Analogie für `fstp`: Die Kugel wird an `ST0` entnommen und an `ziel` abgelegt und dann die Trommel gegen den Urzeigersinn gedreht

Die FPU Assemblerbefehle FST und FSTP

Beispiel:

```
int main() {
    float a = 1.1f;
    float b = 0.3f;
    double c = 0.6f;
    __asm {
        fld a          ;// ST0=1.1
        fld b          ;// ST1=1.1, ST0=0.3
        fst c          ;// c=0.3
        fst st(2)      ;// ST2=0.3, ST1=1.1, ST0=0.3
        fstp st(3)     ;// ST2=0.3, ST1=0.3, ST0=1.1
    }

    return 0;
}
```

Quelldatei: [main.cpp](#)

FPU Assemblerbefehle für ganze Zahlen

Folgende FPU-Befehle kommen beim Laden und Speichern von ganzen Zahlen zum Einsatz:

Befehl	Beschreibung
fild	Laden einer vorzeichenbehafteten ganzen Zahl
fist	Speichern einer vorzeichenbehafteten ganzen Zahl
fistp	Speichern und herausschieben ("pop") einer vorzeichenbehafteten ganzen Zahl

Beispiel (Quelldatei: [main.cpp](#)):

```
int main() {
    int a = 10;
    int b = 20;
    __asm {
        fild a      ;// ST0=10
        fild b      ;// ST1=10, ST0=20
        fistp b     ;// b=20;
        fistp b     ;// b=10;
    }
    return 0;
}
```


Einige Arithmetische FPU Assemblerbefehle

Befehl	Beschreibung
fadd ziel, quelle	Addiert ziel und quelle. Wenn kein Ziel angegeben, gilt ziel=ST0
faddp ziel, quelle	Wie fadd, anschließend herausschieben ("pop") von ST0
fsub ziel, quelle	Subtrahiert ziel und quelle. Wenn kein Ziel, gilt ziel=ST0
fsubp ziel, quelle	Wie fsub, anschließend herausschieben ("pop") von ST0
fdiv ziel, quelle	Dividiert ziel und quelle. Wenn kein Ziel, gilt ziel=ST0
fdivp ziel, quelle	Wie fdiv, anschließend herausschieben ("pop") von ST0
fmul ziel, quelle	Multipliziert ziel und quelle. Wenn kein Ziel, gilt ziel=ST0
fmulp ziel, quelle	Wie fmul, anschließend herausschieben ("pop") von ST0
fabs	Berechnet den Betrag von ST0
fchs	Ändert das Vorzeichen von ST0
fyl2x	Berechnet $ST1 = ST1 \cdot \log_2(ST0)$, und "pop" von ST0
fsqrt	Berechnet die Quadratwurzel von ST0

FPU Assemblerbefehle zum Laden einer Konstanten

Folgende FPU-Befehle können zum Laden einer Konstanten eingesetzt werden:

Befehl	Beschreibung
f1d1	Laden von 1.0
f1dl2e	Laden von $\log_2 e$
f1dl2t	Laden von $\log_2 10$
f1dlg2	Laden von $\log 2 (= \log_{10} 2)$
f1dln2	Laden von $\ln 2 (= \log_e 2)$
f1dpi	Laden von $\pi = 3.141592\dots$

FPU Assemblerbefehle

Beispiel: Berechnung des natürlichen Logarithmus $\ln(x)$

Gemäß den Rechenregeln für Logarithmen gilt:

$$\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

und

$$\log_a(b) = \frac{1}{\log_b(a)}$$

D.h. für den natürlichen Logarithmus ergibt sich:

$$\ln(x) = \log_e(x) = \frac{\log_2(x)}{\log_2 e} = \log_e(2) \cdot \log_2(x)$$

FPU Assemblerbefehle

Lösung:

```
#include <stdio.h>
#include <math.h>

float myLog(float x) {
    float result;
    __asm {
        fldln2    ;// load log_e(2) in ST0
        fld x     ;// load ST0=x; ST1=log_e(2)
        fyl2x    ;// ST1 = log_e(2)*log_2(x) = log_e(x); pop;
        fstp result; // store ST0 in result; pop;
    }
    return result;
}

int main() {
    float x = 25.0;
    float result = myLog(x);
    printf("ln(%f) = %f \n", x, result); // output result of myLog
    printf("ln(%f) = %f \n", x, log(x)); // check result with C log function
    return 0;
}
```

Quelldatei: [main.cpp](#)

Thorsten Thormählen 16 / 27

FPU Assemblerbefehle

Laut "cdecl"-Aufrufkonvention werden Gleitkommazahlen im ST0 Register zurückgegeben

Verbesserte Lösung:

```
#include <stdio.h>
#include <math.h>

float myLog(float x) {
    __asm {
        fldln2    ;// Load log_e(2) in ST0
        fld x     ;// load ST0=x; ST1=log_e(2)
        fyl2x    ;// ST1 = log_e(2)*log_2(x) = log_e(x); pop;
    }
}

int main() {
    float x = 25.0;
    printf("ln(%f) = %f \n", x, myLog(x)); // output result of myLog
    printf("ln(%f) = %f \n", x, log(x)); // check result with C log function
    return 0;
}
```

Quelldatei: [main.cpp](#)

Multi Media Extension (MMX)

Mit dem Pentium MMX führte Intel 1997 eine Befehlserweiterung ein, die es erlaubt, die acht FPU-Register für SIMD-Operationen ("Single Instruction Multiple Data") zu verwenden

Diese Register werden mit `MM0, MM1, ..., MM7` angesprochen

Dabei wird jeweils von einem 80-Bit FPU-Register nur 64-Bits für ein MMX-Register verwendet

Die MMX-Befehle arbeiten mit ganzzahligen Operanden

Es ist möglich, für acht 8-Bit-, vier 16-Bit-, oder zwei 32-Bit-Operanden parallel die gleiche Operation auszuführen

D.h. es kann mit einem maximalen theoretischen Geschwindigkeitsgewinn von Faktor 8 gerechnet werden

Der Name MMX motiviert sich daher, dass solche SIMD-Operationen vor allem bei der Verarbeitung von Audio-, Bild-, oder Videodaten eingesetzt werden, d.h bei Anwendungen bei denen häufig die gleiche Operation auf vielen Elementen ausgeführt werden müssen

Multi Media Extension (MMX)

Insgesamt umfasst die MMX-Erweiterung mehr als 40 Befehle

Beispiele sind: padd, psub, pmul, pxor, pand, por, usw.

Die Befehle werden teilweise mit einem Suffix versehen, das signalisiert ob es sich um

s (= "signed") vorzeichenbehaftete Operanden oder

us (= "unsigned") vorzeichenlose Operanden handelt

Außerdem kennzeichnet ein Suffix, wie viele Bytes die Operanden besitzen:

Suffix	Größe der Operanden
b	Byte (1 Byte)
w	Word (2 Bytes)
d	Double Word (4 Bytes)
q	Quad Word (8 Bytes)

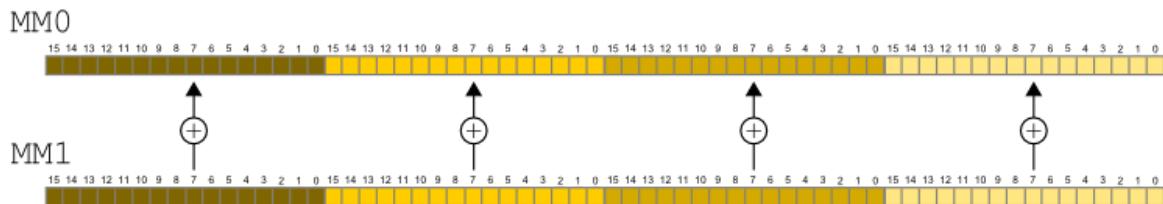
Frage: Der MMX-Befehl paddusw steht für welche Operation?

Multi Media Extension (MMX)

Der MMX-Befehl

```
paddusw mm0, mm1
```

addiert vier vorzeichenlose 16-Bit Zahlen



Die Ausführung erfolgt parallel, d.h. der Befehl berechnet zeitgleich vier Ergebnisse

Multi Media Extension (MMX)

Beispiel:

```
#include <stdio.h>

int main() {
    char text[] = "go fast!";
    char subMe[] = { 32, 32, 0, 32, 32, 32, 32, 0 };
    printf("%s \n", text);

    __asm {
        movq mm0, text      ;//move 64 bits into mm0 register
        psubsb mm0, subMe  ;//subtract content of "text" and "subMe" in parallel
        movq text, mm0
        emms            ;// emms = Empty MMX Technology State (reactivate FPU registers)
    }

    printf("%s \n", text);
    return 0;
}
```

Quelldatei: [main.cpp](#)

Streaming SIMD Extension (SSE)

SSE ist eine weitere x86 Befehlserweiterung, die von Intel 1999 eingeführt wurde und auch von AMD unterstützt wird

SSE verfolgt die gleiche Idee wie MMX, dass die gleichen Operationen auf mehreren Operanden parallel ausgeführt werden ("Single Instruction Multiple Data")

Der Unterschied ist, dass bei SSE 8 neue 128-Bit breite Register reserviert wurden:
XMM0, XMM1, ..., XMM7

Ein weiterer wesentlicher Unterschied ist, dass SSE das Rechnen mit Gleitkommazahlen erlaubt

SSE2, SSE3, SSE4 sind spätere Befehlserweiterungen, die zusätzliche Operationen hinzufügen

Anzeigen der SSE-Register in MS Visual Studio:
Menü → Debuggen → Fenster → Register → Kontext-Menü (rechte Maustaste) → SSE

Streaming SIMD Extension (SSE)

Hat ein SSE Befehl das Suffix `ps` ("Packed Single-precision"), z.B. `addps`, `subps`, `mulps`, `divps` usw., können vier 32-Bit-Gleitkommazahlen gleichzeitig von einem Befehl verarbeitet werden

Hat ein SSE Befehl das Suffix `pd` ("Packed Double-precision"), können zwei 64-Bit-Gleitkommazahlen gleichzeitig von einem Befehl verarbeitet werden

Des Weiteren gibt es die Suffixe `ss` ("Scalar Single-precision") und `sd` ("Scalar Double-precision"). Bei Befehlen mit diesen Suffixen ist der eine Operand ein Skalar und kein Vektor

Bei den `mov`-Befehlen kennzeichnet ein weiteres Suffix, ob die Speicheradresse, an der die Werte gelesen bzw. geschrieben werden sollen, auf ein Vielfaches von 16 Byte fällt:

Wenn ja, kann der schnellere Befehl `movaps` ("move aligned") verwendet werden

Wenn nein, der langsamere Befehl `movups` ("move unaligned")

Streaming SIMD Extension (SSE)

Beispiel:

```
#include <stdio.h>

int main() {
    float a[] = { 1.0f, 2.0f, 3.0f, 4.0f };
    float b[] = { 1.1f, 2.2f, 3.3f, 4.4f };
    float s = 2.0f;

    for (int i = 0; i < 4; i++) printf("a[%d] = %f\n", i, a[i]);

    __asm {
        movups xmm1, a    ;//load 4 floats = 128 bit
        movups xmm2, b    ;//load 4 floats = 128 bit
        addps xmm1, xmm2 ;//add all 4 floats in parallel
        addss xmm1, s     ;//add scalar s to 1st float, 2nd - 4th float remain unchanged
        movups a, xmm1    ;//store into a
    }

    for (int i = 0; i < 4; i++) printf("a[%d] = %f\n", i, a[i]);
    return 0;
}
```

Quelldatei: [main.cpp](#)

x86-64 Register

Beim Übergang auf die 64-Bit x86 Architektur (x86-64), wurden zunächst durch AMD und später auch durch Intel die Mehrzweckregister, das Flag Register und der Befehlszähler erneut erweitert und durch das Buchstabenpräfix R gekennzeichnet

Im 64-Bit Modus stehen dem Programmierer somit die Register RAX, RBX, ... RSP zur Verfügung. Die niederwertigen Teile davon können mit den älteren Bezeichnungen EAX, AX, AH, AL, etc. angesprochen werden

Neben diesen erweiterten Registern gibt es acht neue 64-Bit Mehrzweckregister mit den Bezeichnungen R8, .. , R15

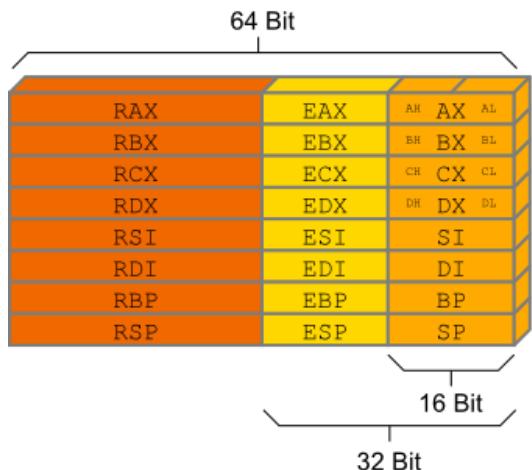
Alle neueren Intel und AMD Prozessoren können wahlweise als 64-Bit Rechner oder im älteren 32-Bit Modus betrieben werden

Im 64-Bit Modus werden ältere 32-Bit Programme in einem Kompatibilitätsmodus ausgeführt

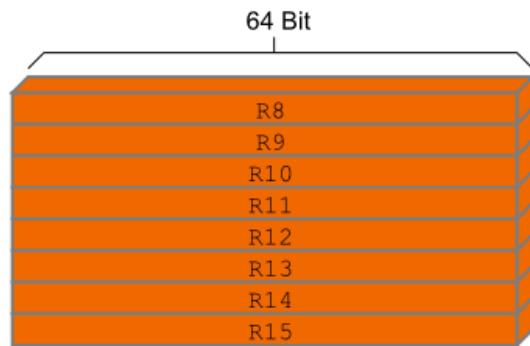
Davon machen auch die 64-Bit-Versionen der Betriebssysteme Windows 7 und 8 Gebrauch

x86-64 Register

Erweiterte Allzweckregister



Neue Allzweckregister



Befehlszähler



Flagregister



Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: [Kontakt](#)

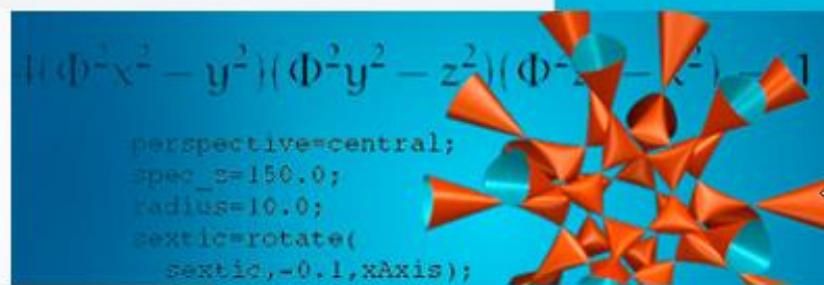
[Weitere Vorlesungsfolien](#)

[Impressum](#) , [Datenschutz](#) ,]

Thorsten Thormählen 27 / 27

Technische Informatik I

Teil 11, Kapitel 1:
Prozessor-Architekturen



Kapitel Inhalt

- ▶ RISC- und CISC Architekturen
- ▶ RISC-Prozessoren
- ▶ Pipelining

RISC-Prozessoren

- Zu Beginn der 80er Jahre wurde der Begriff **RISC** geprägt.
- Diese Abkürzung steht für ein CPU-Konzept mit einem **reduzierten** Befehlssatz.
 - ▶ **RISC** = Reduced Instruction Set Computer.
- Für alle bis dahin verwendeten CPU-Konzepte wurde der Begriff **CISC** geprägt
 - ▶ **CISC** = Complex Instruction Set Computer.
- Das Ziel der RISC-Philosophie war es, die CPU-Architektur an neuere Entwicklungen der Hardware-Technik anzupassen.
- Die Grundidee war, einen Maschinenbefehl nicht durch ein Mikroprogramm zu implementieren, sondern ihn direkt durch einen einzigen Mikrobefehl ausführen zu lassen.
- Anfang der 80er Jahre entstanden Prototypen zur RISC-Technologie
 - ▶ Stanford **MIPS**, Berkeley **RISC** und IBM **801**.
- Die ersten darauf aufbauenden kommerziellen Produkte waren nicht besonders erfolgreich.
 - ▶ Beispiel: IBM **R/6000** Serie.

RISC-Prozessoren

- Ende der 80er Jahre begann dann die Blütezeit der RISC-Technologie.
- In den folgenden Jahren gab es bzw. gibt es mehrere kommerziell und technisch mehr oder weniger erfolgreiche Produktreihen, die auf RISC-Prozessoren aufbauen:
 - ▶ IBM RS/6000, PowerPC,
 - ▶ DEC α,
 - ▶ SUN Sparc, Ultra Sparc,
 - ▶ SGI Iris Indigo, Crimson, Indy 2, O2, Challenger, Origin,
 - ▶ HP PA.
- Die ARM Architecture ist in Stückzahlen gemessen wohl die erfolgreichste RISC-Familie, sie findet sich in vielen Systemen, bei denen es um hohe Leistung, geringen Stromverbrauch und niedrige Kosten geht (Typisch 100–500 MHz im Jahr 2008).
- Die ARM Ltd., die diese Systeme konstruiert, baut allerdings selbst keine Prozessoren, sondern verkauft lediglich Lizenzen für das Design an ihre Kunden. Mittlerweile sollen **10 Milliarden** ARM-Cpus im Umlauf sein, die unter anderem in folgenden Systemen zum Einsatz kommen:
 - ▶ Apple: iPods (ARM7TDMI SoC),
 - ▶ Apple: iPhone (Samsung ARM1176JZF, Apple A4)

CISC Prozessoren

- Als besonders markante Vertreter von Computerfamilien, die auf CISC-CPUs aufbauen, gelten die Familien:
 - ▶ IBM /360, /370, /390, ES9000, System z
 - ▶ DEC VAX
- Aber auch die Mikroprozessoren der x86-Serie von Intel und der 680x0-Serie von Motorola müssen diesem CPU-Konzept zugeordnet werden, auch wenn die Hersteller Wert darauf legen, bei neueren Modellen weitgehend RISC-Konzepte zu berücksichtigen.
- Dies trifft z.B. insbesondere zu auf :
 - ▶ Intel Pentium, Core Duo, i7, ...

Motivation für CISC

- Die den CISC-Prozessoren zugrunde liegenden Ideen charakterisieren die Situation Anfang der 60er Jahre und gehen aus von:
 - ▶ einer relativ schnellen Arbeitsweise der CPU,
 - ▶ einem relativ langsamen Arbeitsspeicher,
 - ▶ einem sehr kleinen Arbeitsspeicher,
 - ▶ einem sehr teuren Arbeitsspeicher.
- Während die ersten beiden Punkte sich bis heute nicht wesentlich verändert haben, aber im Gegensatz zu früher durch Cache-Speicher im Wesentlichen kompensiert werden können, treffen die beiden letzten Punkte heute nicht mehr zu.
- Aus damaliger Sicht war es jedoch erstrebenswert, möglichst wenige, dafür **komplexe** Maschinenbefehle zu verwenden, mit dem Ziel, Programme zu verkürzen und die Zahl der Speicherzugriffe zum Laden von Instruktionen zu minimieren.

CISC und Mikroprogramme

- Ermöglicht werden komplexe Maschinenbefehle durch die **Mikroprogrammtechnik**.
- Nur durch die Einführung dieser zusätzlichen Abstraktionsschicht erhält man die Chance, eine große Zahl komplexer Befehle fehlerfrei in einer CPU zu realisieren.
- Die komplexen Maschinenbefehle werden als Einsprungpunkte in ein Mikroprogramm aufgefasst und von dem dort anzutreffenden Mikroprogramm gesteuert.
- Sie werden durch eine relativ einfache CPU-Logik ausgeführt.
- Das Mikroprogramm kann vor der Konstruktion der CPU entworfen werden und mit Hilfe von Simulationsprogrammen ausgetestet werden.
- Eine weitere Motivation für die Verwendung von Mikroprogrammen war die Implementierung von Prozessor-Familien:
 - ▶ Definiert wird eine Hard- bzw. Software-Schnittstelle in Form einer definierten Maschinenarchitektur.
 - ▶ Unterschiedlichen Mikroprogramme realisieren diese Architektur auf mehreren Modellen der Prozessorfamilie:
 - ▶ Einfache und damit billige Modelle
 - ▶ Mittlere Modelle
 - ▶ Aufwändige und damit leistungsfähige aber teure Modelle

CISC Eigenschaften

- Typische Eigenschaften von CPUs in CISC-Architektur:
 - ▶ Relativ wenige Register.
 - ▶ Die meisten Register sind keine Mehrzweckregister sondern haben Spezialfunktionen.
 - ▶ Die Maschinenbefehle haben meist sehr viele Varianten, die mit Tricks wie z.B. Modifier-Bits, Prefix-Bytes, etc. aufwändig codiert werden.
 - ▶ Insgesamt gibt es viele und teilweise sehr komplexe Maschinenbefehle.
 - ▶ Es gibt viele Typen von Operanden:
 - ▶ direkte Operanden (also explizite Werte)
 - ▶ Registeroperanden
 - ▶ Speicheroperanden (also Werte aus dem Speicher mit vielen Adressierungsarten)
- Die Speicheradressierung und die große Menge komplexer Instruktionen werden häufig damit begründet, dass auf Basis eines solchen Designs die Generierung von Maschinencode durch einen Compiler einfacher wird.
- Die obigen Punkte treffen offensichtlich auf die x86-Familie von Intel zu!

Von CISC zu RISC

- Einer der Ausgangspunkte des Übergangs zu RISC-Architekturen war die Untersuchung gängiger Compiler.
- Es wurden Statistiken bekannt, denen zufolge gängige **Compiler einen großen Teil der komplexen Instruktionen überhaupt nicht verwendeten.**
- Und auch dort, wo sie verwendet wurden, trugen die komplexen Instruktionen nur ca. 20 % zur Laufzeit des generierten Codes bei.
- Die meisten verwendeten Instruktionen sind dagegen so einfach, dass sie auch zur RISC Philosophie passen.
- Hinzu kam die Beobachtung, dass immer mehr Speicher innerhalb der CPU und im Arbeitsspeicher zur Verfügung stehen, so dass keine Notwendigkeit besteht, Instruktionen zusammenzustauen.
- Cache-Speicher verringern den zusätzlichen Zeitaufwand zum Laden von **mehreren** Instruktionen.
- Eine der Maßnahmen zur Reduzierung des Platzbedarfs von CISC-Instruktionen war die Definition zahlloser Befehlsformate: So kann die Befehlslänge bei der x86-Familie von 1 bis 32 Byte variieren, wobei fast jeder Zwischenschritt möglich ist.
- Bei dem weitgehend unbekannten Prozessor **iAPX 432** aus dem Jahr 1982 variierten Befehlsanfang und Befehlslänge nicht nur auf Byte-, sondern sogar auf Bitebene.

RISC-Eigenschaften

- RISC-Prozessoren sind gekennzeichnet durch:
 - ▶ wenige einfache Befehle, die möglichst in einem Maschinentakt ausgeführt werden,
 - ▶ wenige Befehlsformate, möglichst mit nur einer festen Befehslänge,
 - ▶ viele Mehrzweckregister,
 - ▶ Speicherzugriffe nur über Load- bzw. Store-Befehle.
- Letzteres bedeutet, dass Quelle und Ziel von Operationen nur Register, nie Hauptspeicher sein können.
- Werden Operanden aus dem Speicher benötigt, so müssen sie vorher durch einen gesonderten Load-Befehl in einem Register bereitgestellt werden.
- Ein typisches Beispiel hat der MIPS-Prozessor R3000
 - ▶ er hat nur 64 Maschinenbefehle,
 - ▶ der Operationscode wird mit 6 Bit codiert,
 - ▶ es gibt drei Befehlsformate,
 - ▶ alle Befehle haben die gleiche Länge,
 - ▶ es gibt 32 Mehrzweckregister.

RISC-Prozessoren: Registerzahl

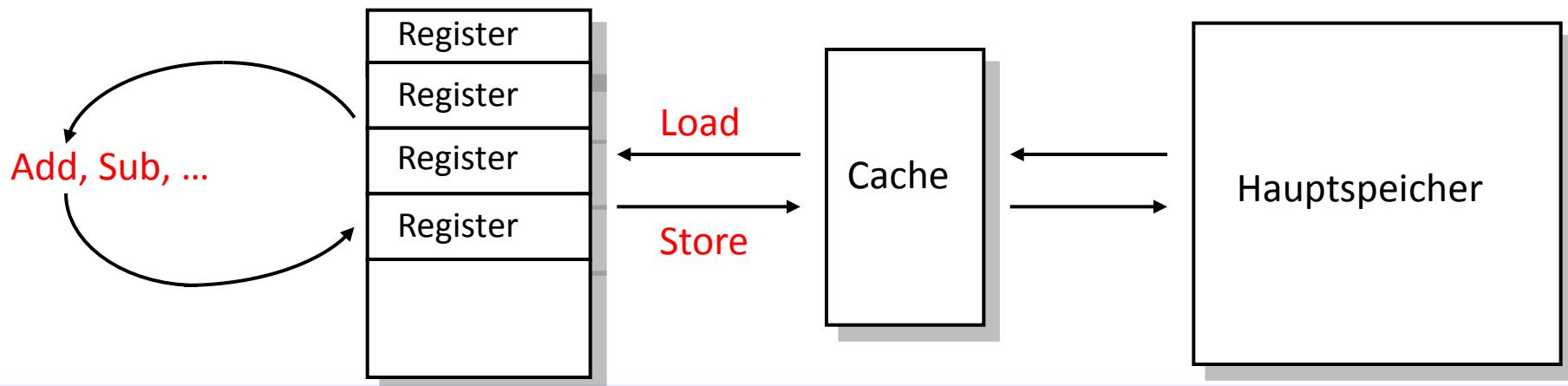
- Die Anzahl der Register war bei den ersten Prototypen der RISC-Architekturen sehr verschieden:
 - ▶ Der Stanford **MIPS** (Vorläufer der MIPS R Serien) hatte nur 16 Register.
 - ▶ Der Berkeley **RISC** (Vorläufer der SPARC-Prozessoren) hatte 138 Register.
 - ▶ Der IBM **801** (Vorläufer der IBM-POWER-Prozessoren) hatte 32 Register.
- Heute gilt die Zahl von 32 Registern als guter Kompromiss.
- Die Effekte, die man mit einer größeren Anzahl von Registern erzielen wollte, insbesondere die Verringerung von Speicherzugriffen, erreicht man heute besser mit einem On-Chip-Cache.
- Strittig ist noch die Frage, wie viele Befehle ein RISC-Prozessor haben soll.
 - ▶ Die von **64** Befehlen des MIPS R3000 erscheint aus heutiger Sicht sehr einengend.
 - ▶ Möglicherweise ist eine Codierung des Operationscodes durch **8** Bits sinnvoller.
 - ▶ Diese würde bis zu **256** Befehle zulassen und damit eigentlich der ursprünglichen RISC-Philosophie widersprechen.

RISC-Prozessoren: Motivation kleiner Befehlssätze

- Viele ursprüngliche RISC-Entwürfe gingen von einer Aufteilung der Funktionen auf mehrere Chips aus:
 - ▶ Diese sollten den Hauptprozessor und mehrere **Coprozessoren** realisieren.
 - ▶ Die Coprozessoren sollten über eigene Befehlssätze verfügen.
 - ▶ Daher glaubte man, für jeden einzelnen Prozessor mit wenigen Befehlen auskommen zu können.
- Mit der heutigen Integrationsdichte erscheint diese Vorgehensweise nicht mehr zeitgemäß.
- Es ist sinnvoller, Gleitpunktoperationen durch eigene Maschinenbefehle anzusprechen und nicht über eine Coprocessorschnittstelle abzuwickeln.
- Dies trifft auch auf alle anderen "ausgelagerten Befehlssätze" zu.

RISC-Prozessoren: Load- Store-Architektur

- Nach wie vor unumstritten ist die Reduktion des Speicherzugriffs auf **Load-** und **Store-** Befehle.
- Daten können nur manipuliert werden, wenn sie sich in Registern befinden.
 - ▶ Das vereinfacht das Design von CPUs und Cache's erheblich.
 - ▶ ... und macht sie schneller.
 - ▶ Das passt auch sehr gut zu optimierenden Compilern:
 - ▶ Häufig verwendete Variablen können für längere Abschnitte fest einem Register zugeordnet werden.



RISC-Prozessoren: Verzögerte Ausführung

- Häufig vorkommende Befehle wie **Load**, **Store**, **Add**, **Sub** etc. werden möglichst schnell ausgeführt, d.h. in einem Maschinentakt und ohne Hilfe eines Mikroprogramms.
- Einer besonderen Optimierung bedarf es auch bei **Sprungbefehlen**:
 - ▶ Diese kommen sehr häufig vor.
 - ▶ Wegen eines eventuell notwendigen Speicherzugriffs zum Laden der Zieladresse können sie meist nicht in einem Takt erledigt werden.
 - ▶ Häufig findet man daher das Konzept des **verzögerten Sprungs**:
 - ▶ Ein Sprungbefehl wird **in einem Takt abgearbeitet**, **springt** aber erst einen Takt später.
 - ▶ Der Programmierer/Compiler hat Gelegenheit dies zu nutzen, indem er die Befehle so umsortiert, dass unmittelbar nach dem Sprungbefehl noch ein Befehl abgearbeitet wird, der logisch gesehen vor dem Sprung ausgeführt werden soll und die Sprungbedingung nicht beeinflusst.
 - ▶ Falls ein solcher Befehl nicht gefunden werden kann, muss ein NoOp-Befehl (No Operation) eingefügt werden.
- Dieses Konzept der verzögerten Wirkung kann auch auf **Load-** und **Store-Befehle** angewendet werden, falls sich die Taktzeit auf diese Weise weiter reduzieren lässt.

RISC-Prozessoren: Mikroprogramme

- Die ursprüngliche RISC-Philosophie forderte, auf **Mikroprogramme** ganz zu verzichten.
- Solange nur ganz wenige Befehle mehrere Takte benötigen (z.B. die **Multiplikation** und vor allem die **Division**), ließ sich das auch durchhalten.
- Heute werden wegen der hohen Integrationsdichte wieder zunehmend komplexere Befehle in der CPU ausgeführt, z.B.
 - ▶ Gleitpunktbefehle,
 - ▶ Bitblockbefehle,
 - ▶ Multimediacbefehle,
- Heute fordert man daher lediglich
 - ▶ dass nur wenige Befehle mithilfe von **Mikroprogrammen** abgewickelt werden.

Transistorfunktionen und Leistungssteigerung

- Heute werden vor allem folgende Konzepte zur Leistungssteigerung angewendet:
 - ▶ Anwendung der Fließbandtechnik (Pipelining),
 - ▶ Parallelisierung (Superskalar-Technik).
- Diese Techniken steigern die Leistung der Prozessoren und nutzen die ungeheure Zahl von Transistorfunktionen, die in einem heutigen Chip potentiell zur Verfügung stehen.
- Im Jahr 2008 hatten die höchstintegrierten Chips (4 GBit DRAMs)
 - ▶ ca. 6.400.000.000 Transistorfunktionen.
- Die meisten bekannten CPU-Chips verwenden bisher erheblich weniger Transistorfunktionen:
 - ▶ Der Intel Core Duo E8600 hat ca. 450.000.000 Transistorfunktionen (Jahr 2008)
 - ▶ Der Intel Core i7 980X hat ca. 1.700.000.000 Transistorfunktionen (Jahr 2010)
 - ▶ Der Intel Itanium 9350 hat ca. 2.046.000.000 Transistorfunktionen (Jahr 2010)
- Relativ wenige der Transistorfunktionen werden für die CPU Logik benötigt.
- Ansätze zur Nutzung zusätzlicher Transistorfunktionen:
 - ▶ Mehrere CPU Kerne in einem Prozessorchip: 2, 4, 6 oder sogar 8 Kerne.
 - ▶ Integrierte L1-, L2 und L3 Caches.
 - ▶ Beim i7-980X: 6 Kerne; je Kern 64 kB L1 und 256 kB L2-Cache; ein L3 Cache von 12 MB für alle Kerne.

Pipelining

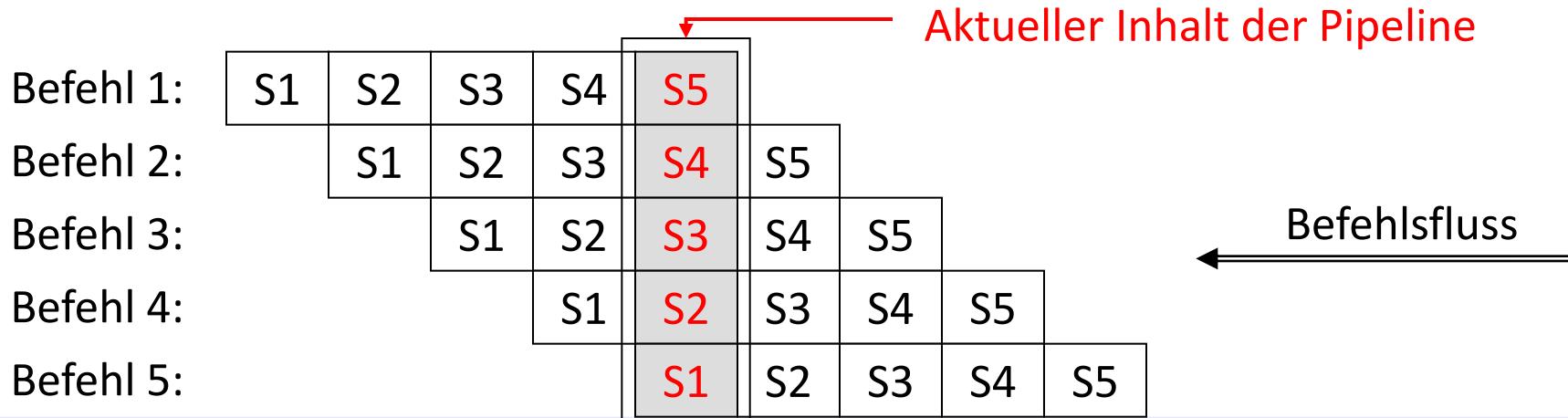
- Eine **Pipeline** ist eine Warteschlange, in der sich die als Nächstes abzuarbeitenden Befehle befinden.
- Jeder Befehl besteht aus einer Reihe von Phasen. Während noch die letzten Phasen der vorderen Befehle in der Pipeline abgearbeitet werden, kann bereits mit den ersten Phasen der nächsten Befehle begonnen werden.
- Mithilfe der Pipeline-Technik lassen sich die Taktzeiten einer CPU weiter reduzieren, wobei angestrebt wird, dass die durchschnittliche Ausführungszeit eines Befehls nahe bei einem Takt liegt.
- Der Befehl wird in mehrere Phasen aufgeteilt, die nacheinander, aber gleichzeitig mit anderen Phasen anderer Befehle, in einer Pipeline ausgeführt werden.
- Während eine Phase eines Befehls bearbeitet wird, erledigt die Pipeline schon andere Phasen weiterer Befehle.
 - ▶ Heute sind 5- bis 35-stufige Pipelines üblich.
- Bei einer 5-stufigen Pipeline könnte die Phasen-Aufteilung für einen Register/Register-Befehl etwa folgendermaßen aussehen:

Pipelining

- Bei einer 5-stufigen Pipeline könnte die Phasen-Aufteilung für einen Register/Register-Befehl etwa folgendermaßen aussehen:

-
- | | |
|------------|---------------------------------------|
| S1: | Befehlsbereitstellung |
| S2: | Dekodieren des Befehls |
| S3: | Lesen der beteiligten Register |
| S4: | ALU-Operation |
| S5: | Schreiben in das Ziel-Register |
-

- Dabei werden bis zu fünf Befehle gleichzeitig überlappend bearbeitet.
- Während die Ergebnisse des 1. Befehls noch in ein Register übertragen werden, wird bereits der 5. Befehl bereitgestellt, der 4. Befehl dekodiert usw.



Pipelining

- Die Verwendung einer Pipeline setzt voraus, dass zwischen den 5 beteiligten Befehlen keine störenden Zwischenbeziehungen existieren.
- Beispiel:
 - **Da der 2. Befehl seine Register bereits gelesen hat, dürfen diese nicht mit dem Register übereinstimmen, das der 1. Befehl noch schreiben will.**
- Es gibt Techniken, solche Zwischenbeziehungen auf Hardwareebene zu entdecken.
- In einem solchen Fall muss die Pipeline zwischen den beteiligten Befehlen so lange angehalten werden bis Konsistenz vorliegt.

Gibt es Fragen?



Anregungen oder Verbesserungsvorschläge können
auch gerne per E-mail an mich gesendet
werden: [Kontakt](#)