



Rechnerarchitektur

Beerdigung – die Fortsetzung

(Spaßi, nur alle Schlüsselfolien vom Bachmeier-Labermeier-
Quatschmeier-Yapmeier)

Computer

Elektronische Schaltungen eines Computers:

- „verstehen“: eine begrenzte Menge einfacher Befehle
- alle Programme müssen in diese Befehle konvertiert werden, erst dann können sie ausgeführt werden.
- Grundlegende Befehle sind „einfach“, z.B.:
 - Addiere zwei Zahlen.
 - Prüfe eine Zahl, um festzustellen, ob sie null ist.

Die Gesamtheit der elementaren Befehle eines Computers bildet die sogenannte **Maschinensprache** (Machine Language), in der der Mensch mit dem Computer kommunizieren kann.



Struktur und Funktion eines Computers

- Komplexes System
Milliarden, vielleicht in wenigen Jahren, bald mehr als eine Billion (1.000.000.000.000)
von elementaren elektronischen Komponenten (z.B. Transistoren)
- Problem:
Wie kann so ein System überhaupt beschrieben werden?
- Schlüssel:
 - Hierarchien erkennen
 - Aufteilen in kleinere Sub-Systeme (und Sub-Sub-Systeme...)
 - Prinzip: Teile und Beherrsche (Hoffentlich!)
- 2 Arbeitsfelder:
 - Struktur
Wie ist die Beziehung der Sub-Systeme untereinander?
 - Funktion
Wie funktioniert jede individuelle Komponente

[Stall]



Funktion eines Computers

- Daten-Verarbeitung (data processing)
der Haupt-Job
- Daten-Speicherung (data storage)
short-term, long-term
- Bewegung von Daten (data movement)
Kommunikation
- Steuerung (Control)
interne Steuereinheit im intern die Ressourcen zu orchestrieren

[Stall]

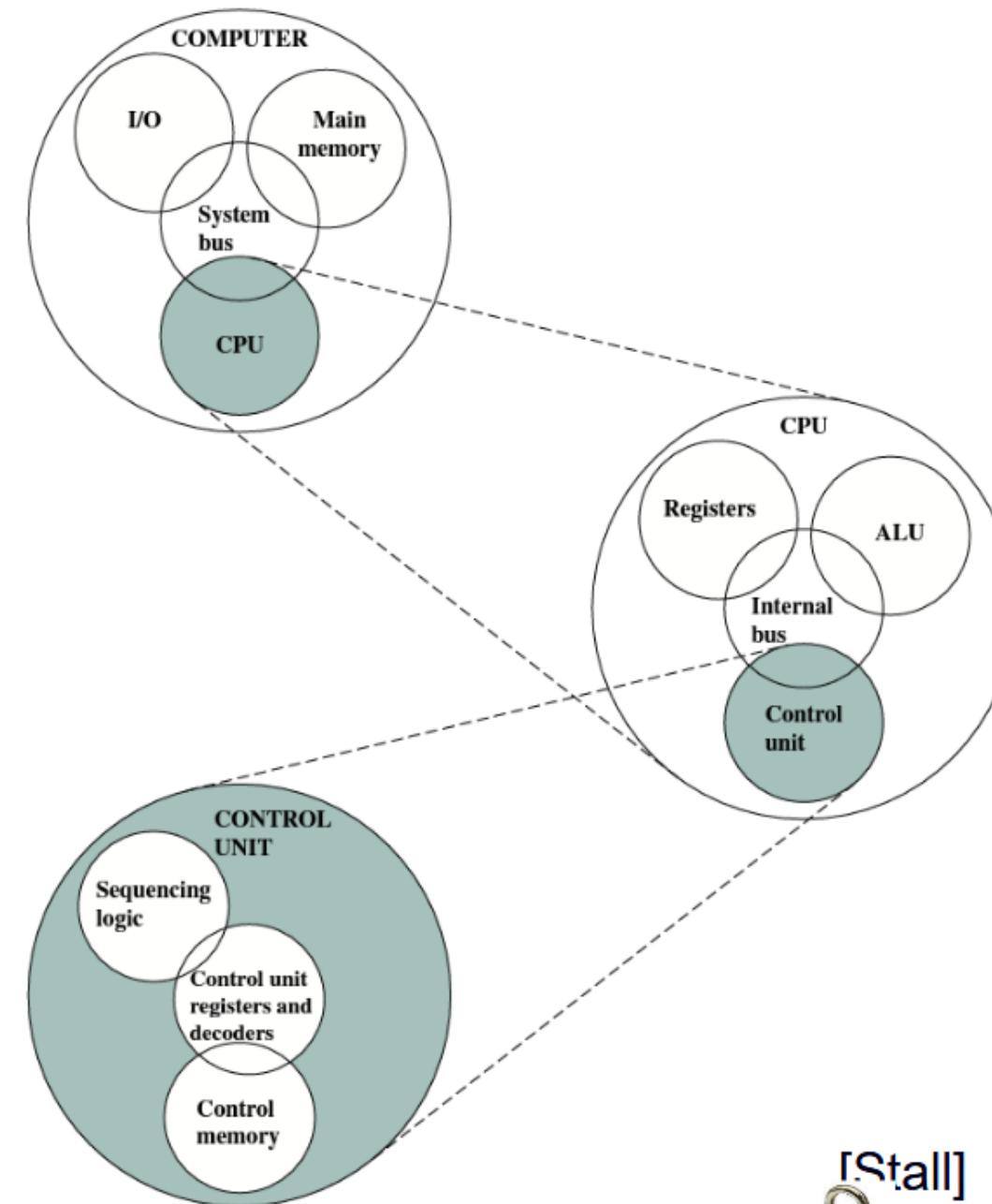


Struktur eines Computers: einfacher Ein-Prozessor-Computer

Hauptkomponenten:

1. Rechenwerk (Central processing unit - CPU):

- ALU mit Akku (ACC) und Registern
- Transfer von Programm und Daten
- Leit- / Steuerwerk: Steuerung der Befehlsausführung



2. Speicherwerk (Main memory):

- Programme und Daten (binär)
- Speicheradressregister SAR (MAR), Speicherdatenregister SDR (MDR / MBR)
- Speicherzellen mit je n Bit, n = Wortlänge = const.
- Speicherzellen über Adressen angesprochen; linearer Adressraum

3. I/O

- Transfer von Programm und Daten

4. System interconnection (System bus)

- Verbindet Teilwerke Zur Übertragung von Adressen, Daten, Steuersignalen

Struktur eines Computers: Multi-Core Computer

Mehrere Prozessoren,

Alle auf einem chip.

Jede Recheneinheit = Core

1. Core:

eine Processing Unit, auf einem Prozessor-Chip: entspricht einer CPU auf single-CPU-System. (ALU, control unit, Register)

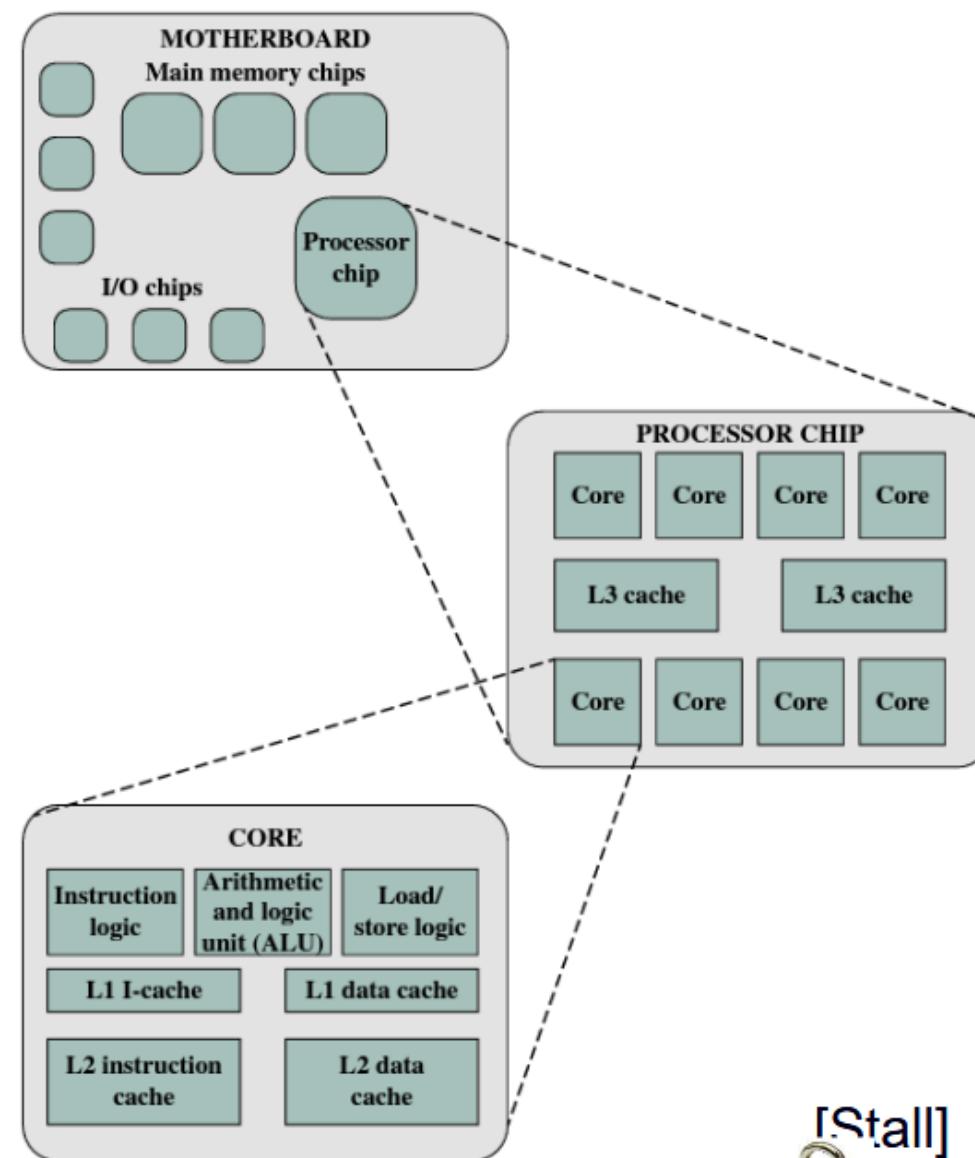
2. Multi-Core Processor

interpretiert Befehle und führt diese aus.

Das in mehreren Cores.

Cache-Memory:

Mehrere Schichten von (schnellem) Speicher, zwischen Processor und (nicht ganz so schnellem) Hauptspeicher.



Rechnerarchitektur und Rechnerorganisation

(Computer-Architecture und Computer-Organisation)

Definition

Rechnerarchitektur (Computer-Architecture)

(von DEC in Anlehnung an Amdahl, Blaauw und Brooks*)

- Computer architecture is defined as the attributes of a computer seen by a **machine-language programmer**.
- This definition includes the **instruction set**, instruction formats, operation codes, addressing modes, and all **registers and memory locations** that may be directly manipulated by a **machine-language programmer**.
- Includes Computer-Organisation (Implementation) as the actual hardware structure, logic design, and datapath organization.

Äußeres
Erscheinungsbild



Rechnerorganisation (Computer-Organisation)

- Teilgebiet der Rechnerarchitektur
- umfasst die Implementation

Implementierung



*Amdahl, G.M., G.A. Blaauw, and F.P. Brooks, Jr. in Architecture of the IBM System/360, IBM Journal of Research and Development, vol. 8, no. 2 (April 1964): 87-101



Mehrschichtige Maschinen – Struktur eines Computers mit Ebenen

Symbolische Namen,
z.B. „ADD“ (anstatt bit-string)
=> Assemblerprogramm: symbolische Darstellung
eines Maschinen-Programms.

wird übersetzt

Vollständige Satz von Befehlen,
die dem Anwendungsprogrammierer zur Verfügung stehen:

- ISA
- Systemaufrufe: aktiviert Betriebssystemdienst: z.B. Lesen aus Datei

ergänzt ISA

bit string („Word“):

001000	00001	00010	0000000101011110
OP Code	Addr 1	Addr 2	Immediate value

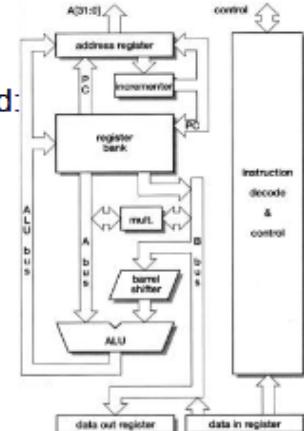
„machine code“
„Maschinen-Programm“



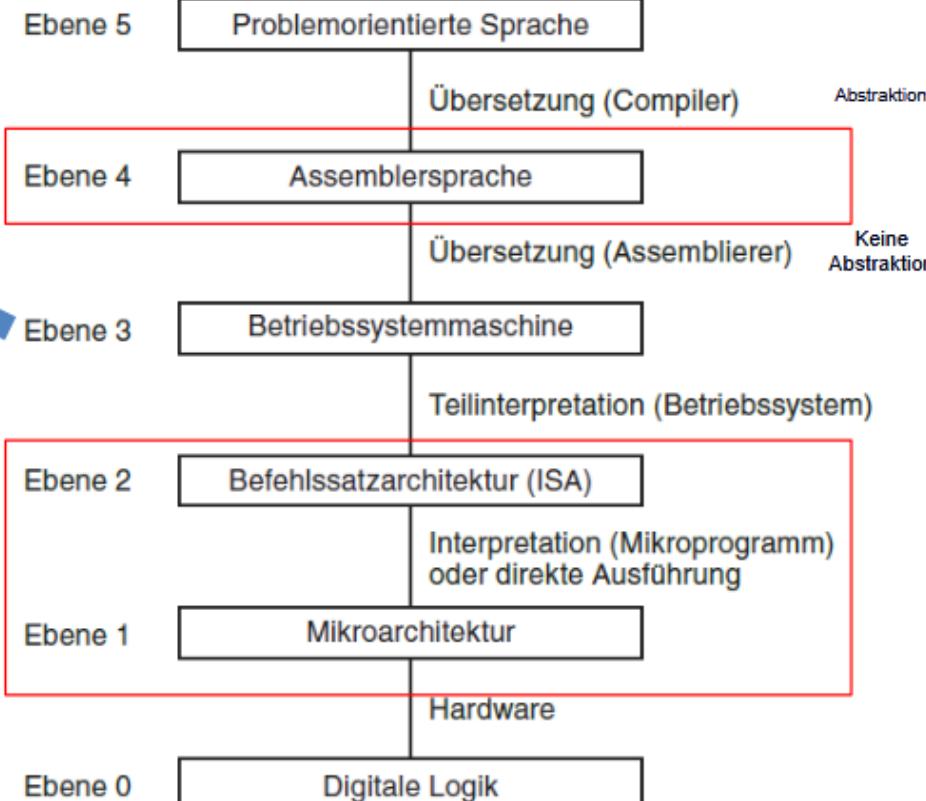
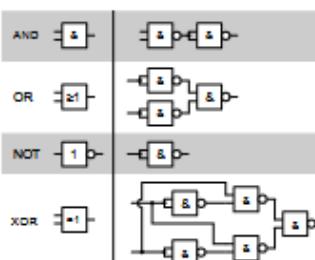
Steuerung Datenpfad:
z.B.:

- Daten aus Register A, B holen
- In der ALU addieren
- Ergebnis in Register C schreiben.

(Steuerung
in HW und/oder
SW: Mikroprogramm)



besteht aus



Schwerpunkte der Vorlesung



Mehrschichtige Maschinen – Evolution

Hardware und Software sind logisch äquivalent

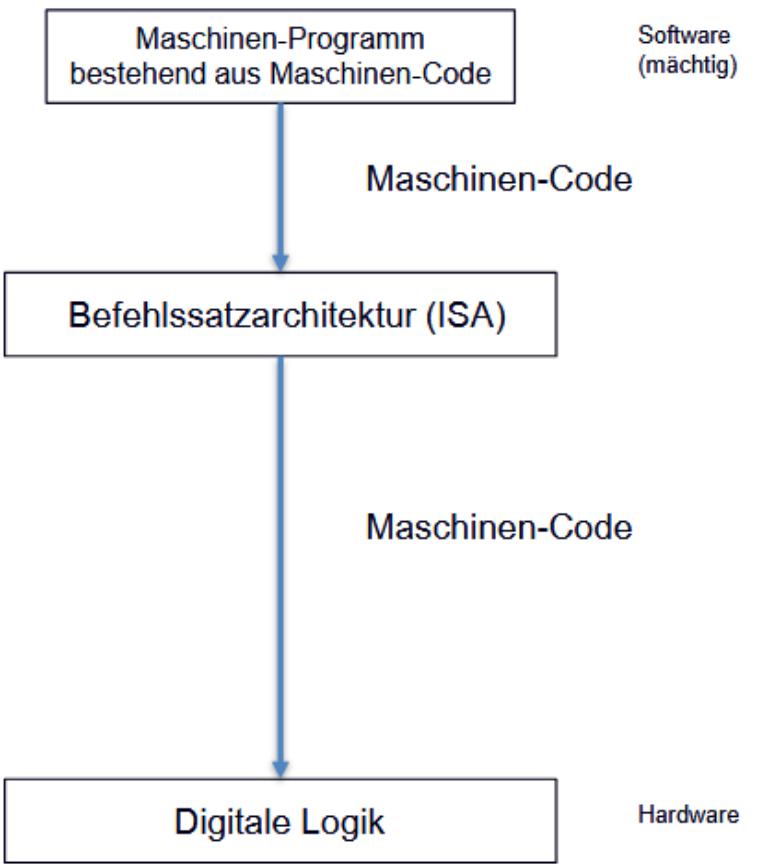
Die Realisierung von Rechnern besteht aus Hardware und Software. Welche Funktionen durch Hardware bzw. Software implementiert werden, wird abhängig von Faktoren wie Kosten, Geschwindigkeit und Anpassbarkeit bestimmt

- **Zu Beginn war die Grenze HW/ SW klar:**
Programme, die in der echten Maschinensprache eines Computers geschrieben sind, lassen sich ohne Interpreter oder Übersetzer direkt von den elektronischen Schaltungen des Computers ausführen.
- Im Laufe der Zeit kamen Ebenen in den Maschinen dazu
- **Heute: Die Grenzen sind verschwommen,
Hardware und Software sind mittlerweile logisch äquivalent:**
 - Eine durch Software ausgeführte Operation kann auch direkt in Hardware integriert werden
 - Jeder von Hardware ausgeführte Befehl lässt sich auch durch Software simulieren.
 - Welche Funktionen in Hardware, welche in Software implementieren?
Wichtig: Kosten, Geschwindigkeit, Zuverlässigkeit und Flexibilität/voraussichtliche Häufigkeit von Änderungen

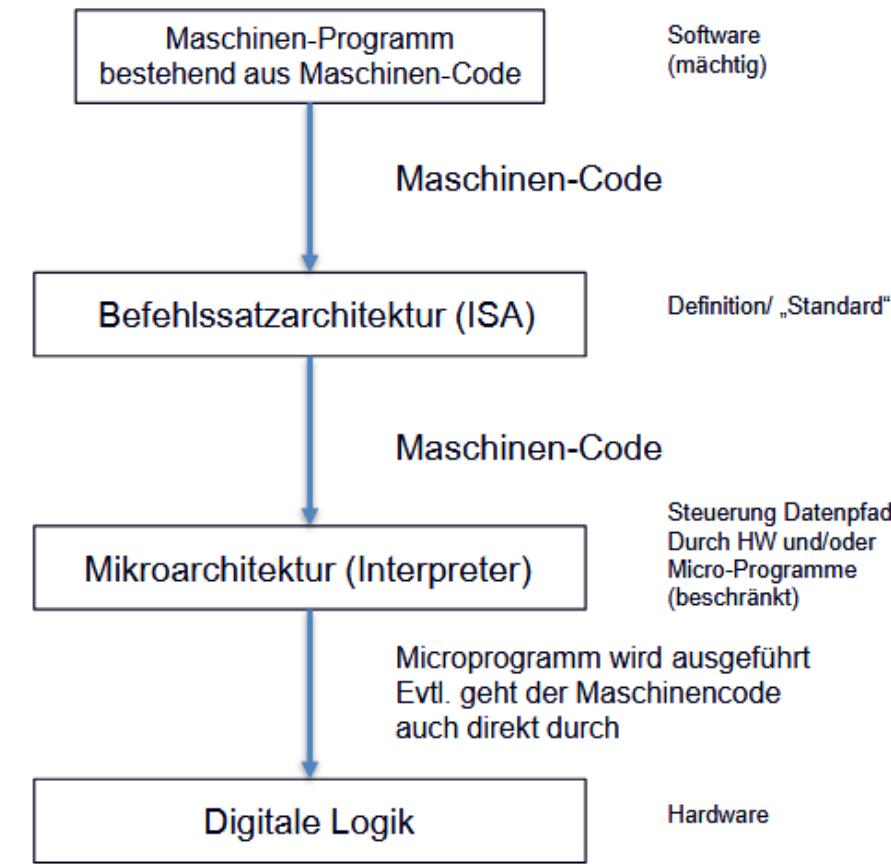
Mehrschichtige Maschinen – Evolution

Mikroprogrammierung/ Mikroarchitektur

So sah ein Computer aus:
vor 1951



So sah ein Computer aus:
Nach 1951 (bis ca. 1970):



Mehrschichtige Maschinen – Evolution

Betriebssystemmaschine

Aus heutiger Sicht:

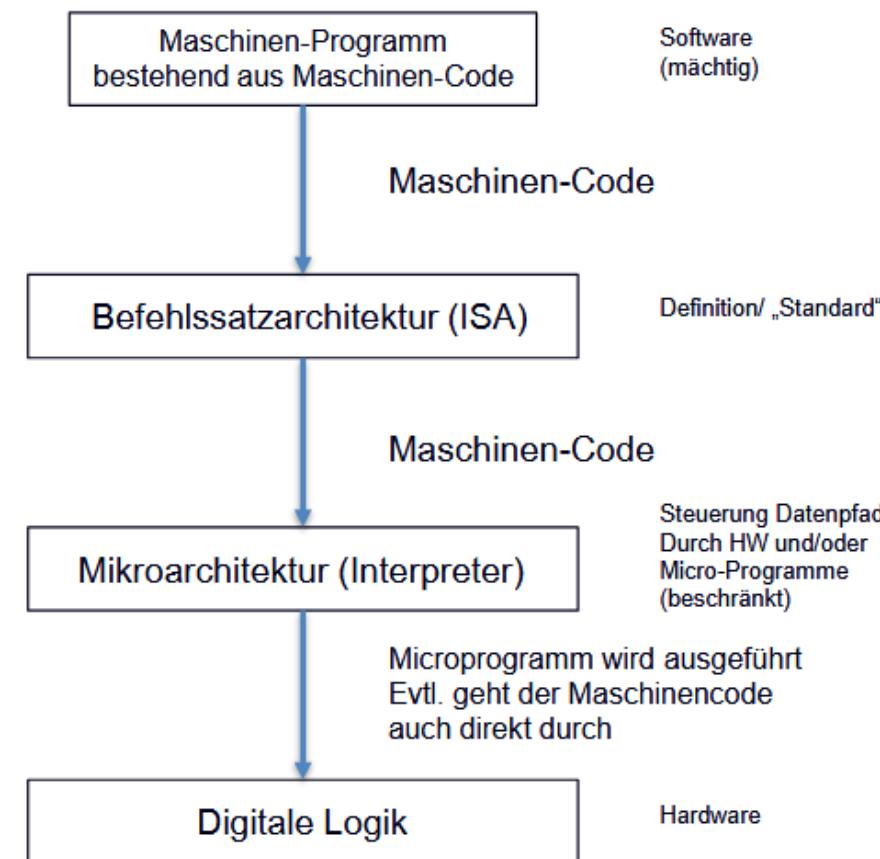
- Wie logt man sich ein?
- Mehrere User?
- Mehrere Programme?
- Zeitvergeudung?

Idee: „Betriebssystem“:

- Soll die Arbeit des Bedieners (Operators) automatisieren.
- Ergänzung der ISA durch Systemaufrufe: Ein/Ausgabe (Ein-Ausgabe ist die Hauptarbeit des Bedieners)
- Timesharing

So sah ein Computer aus:

Nach 1951 (bis ca. 1970):



Flynn'sche Taxonomie (Klassifikation) – 1966/72

1. SISD (Single Instruction, Single Data)

- Einkernprozessor-Rechner
- Aufgaben werden sequentiell abgearbeitet
- z.B. PCs, Workstations, von-Neumann- oder der Harvard-Architektur
- *Bemerkung: gibt nicht mehr so viele Systeme mit nur einem Core*

2. SIMD (Single Instruction, Multiple Data)

- schnelle Ausführung gleichartiger Rechenoperationen, gleichzeitig auf mehrere Eingangsdatenströme.
- Array-Prozessoren/ Vektorprozessoren
- Anwendung: Wenn die Verarbeitung von Daten hochgradig parallelisierbar ist.
z. B. bei Bearbeitung eines Bildes sind Operationen für jeden Bildpunkt identisch.
- *Standard (Bsp. Intel MMX)*

3. MISD (Multiple Instruction, Single Data)

- Mehrere, verschiedene Berechnungen, gleichzeitig auf den gleichen Daten ?!?
- Bsp: Schachcomputer
- *von untergeordneter Bedeutung*

4. MIMD (Multiple Instruction, Multiple Data)

- Gleichzeitig verschiedene Operationen auf verschiedenen gearteten Eingangsdatenströmen
- Verteilung der Aufgaben an die zur Verfügung stehenden Ressourcen notwendig
- Jeder Prozessor hat Zugriff auf die Daten anderer Prozessoren.
- Großrechner
- Bsp: „make“:
mehrere zusammengehörige Quellcodes files, werden gleichzeitig auf verschiedenen Cores übersetzte



Meilensteine der Computerarchitektur

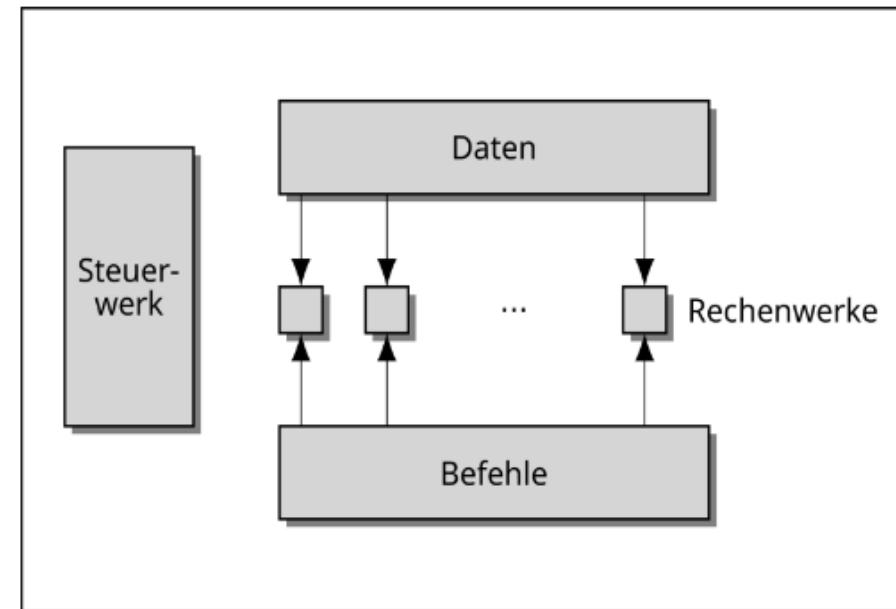
Einschub: Harvard-Architektur

https://en.wikipedia.org/wiki/Harvard_architecture

Befehlsspeicher logisch und physisch vom Datenspeicher getrennt!

Vorteile:

- Befehle und Daten können gleichzeitig gelesen bzw. geschrieben werden.
Bei von-Neumann sind aufeinander folgende Buszyklen notwendig.
- Typischerweise ist der Programmspeicher read-only (ROM) (verhindert ein Überschreiben des Codes), und der Datenspeicher read-write (RAM).



Nachteile:

- Hardwareaufwand höher, u.U. doppelt soviele Leitungen und CPU-Anschlüsse
- Race Conditions sind möglich bei Daten- und Befehlszugriffen und damit nicht-deterministischer Programmablauf.

Fun-Fact:

The term “Harvard architecture” did not exist in the era of the Harvard machines



Meilensteine der Computerarchitektur

Erste – Generation (1945-1955) Elektronisch (Röhren)

John von Neumann entwarf einen Rechner mit Binärarithmetik, bei dem das Programm im gleichen Speicher wie die Daten abgelegt wurde.

Dieser Entwurf hat die Entwicklung der Rechner bis in die heutige Zeit geprägt.

Von-Neumann-Architektur

(Erste Implementierung der Von-Neumann-Architektur

- Die Von-Neumann-Maschine besteht aus fünf grundlegenden Teilen:

- Speicherwerk
- Rechenwerk (Arithmetic Logic Unit, ALU)
- Steuerwerk
- Eingabewerk
- Ausgabewerk.

- Der Speicher bestand aus 4096 Wörtern.
Ein Wort umfasste 40 bit.

- Jedes Wort besteht aus
 - zwei 20-bit-Befehlen
 - oder einer 40-bit-Ganzzahl mit Vorzeichen

- Aufbau eines Befehls (20bit):

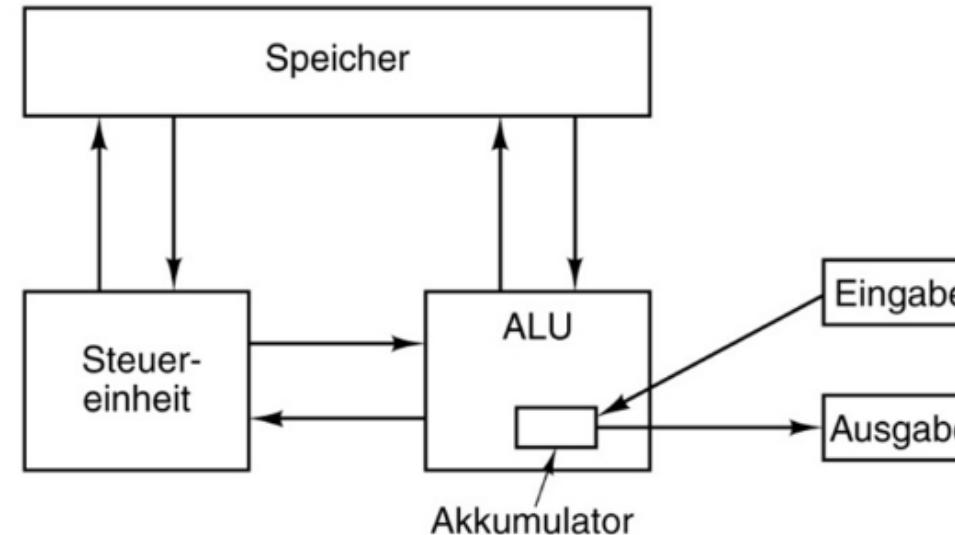
- Befehlstyp: 8 bit
- Adressierung eines der 4096 Speicherwörter: 12 bit

- Das Rechenwerk (ALU) und das Steuerwerk bildeten zusammen das „Gehirn“ des Computers,
(Heute: In modernen Computern sind sie auf einem einzelnen Chip vereint, der zentralen Verarbeitungseinheit oder CPU (Central Processing Unit))

- Im Rechenwerk befand sich ein spezielles internes 40-Bit-Register, der Akkumulator.

- Ein typischer Befehl addierte ein Speicherwort zum Akkumulator oder legte den Inhalt des Akkumulators im Speicher ab.

- Die Maschine besaß keine Gleitkommaarithmetik.



Meilensteine der Computerarchitektur

Erste – Generation (1945-1955) Elektronisch (Röhren)

Von-Neumann-Architektur

Universalrechner: Struktur des Rechners unabhängig von Problemstellung
(speicherprogrammierbar)

Bedeutende Neuerung:

- Programm und Daten erst in selben Speicher laden und dann ausführen
- Sprünge/Verzweigungen auf vorherige oder spätere Sequenzen möglich
- Befehle (Programm = Folge von Befehlen) und Daten (Operanden) liegen im selben Speicher und durch Rechner zur Laufzeit modifizierbar (Vorsicht!, siehe Nachteile)
- Rechner kann selbstständig logische Entscheidungen treffen
- Prozessor verarbeitet Befehle und Daten streng sequentiell,
Abweichung von der gespeicherten Reihenfolge durch Sprungbefehle
(Befehlszähler + ≥ 1)
- Speicher besteht aus Zellen gleicher Größe, Adressen = Nummern
- Daten und Programme sowie Adressen/Steuerungen sind Bitfolgen (binär)



Meilensteine der Computerarchitektur

Erste – Generation (1945-1955) Elektronisch (Röhren)

Von-Neumann-Architektur

Nachteile:

- Befehle und Daten im Speicher nicht (am Bitmuster) unterscheidbar, d.h. bei falscher Adressierung können Speicherinhalte verändert werden, die nicht verändert werden dürften
- Variablen und Konstanten nicht unterscheidbar, werden je nach Zugriffsart falsch interpretiert
- Sorgfaltspflicht des Programmierers gefragt oder Verwendung von Tags
- Nur ein Verbindungssystem (Bus) zwischen CPU und Speicher: von Neumann Flaschenhals (bottleneck)

Die von Neumann Architektur wird als Architektur des **minimalen Hardwareaufwands** und als Prinzip des **minimalen Speicheraufwands** bezeichnet.

Vielfalt der Computer – Komplexitätssteigerung

„Moore's Law“

[https://de.wikipedia.org/wiki/Moeresches_Gesetz](https://de.wikipedia.org/wiki/Mooresches_Gesetz)

- „Faustregel“
- Zahl der Transistoren integrierter Schaltkreise mit minimalen Komponentenkosten verdoppelt sich regelmäßig.
- je nach Quelle werden 12, 18 oder 24 Monate als Zeitraum genannt

Beispiel: Apple M3:
25 Milliarden Transistoren

<https://www.apple.com/de/newsroom/2023/10/apple-unveils-m3-and-m3-max-the-most-advanced-chips-for-a-personal-computer/>

(= 1 Billion in deutscher Zählweise,
= 1.000.000.000.000 Transistoren)

Ausblick:

Intel Research Fuels Moore's Law and Paves the Way to a Trillion Transistors by 2030

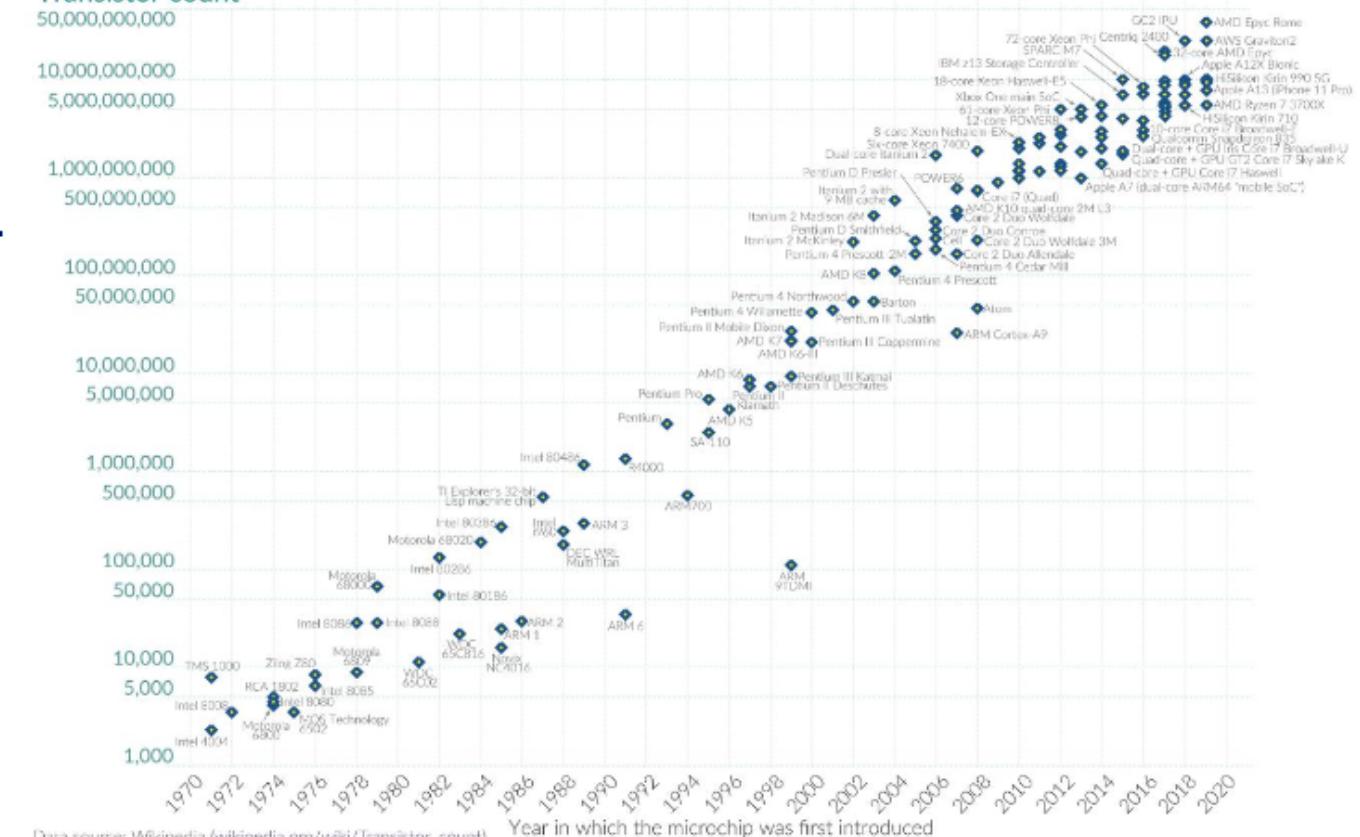
<https://www.intel.com/content/www/us/en/newsroom/news/moores-law-paves-way-trillion-transistors-2030.html#as.6jkgcq>

Moore's Law: The number of transistors on microchips doubles every two years



Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Transistor count



Beispiele von Computerfamilien

x86-Architektur

https://en.wikipedia.org/wiki/Marcian_Hoff

https://en.wikipedia.org/wiki/Intel_4004

Wie es begann:

- 1969 Intel bekommt Auftrag: von 12 kundenspezifischen Chips für einen geplanten elektronischen Rechner.
- Intel-Ingenieur Hoff: idea: „universal processor“ ist besser als 12 chips:
 - eine universelle 4-Bit-CPU, auf einen einzigen Chip gepackt
 - Kann das Gleiche
 - günstiger
 - Einfacher

Startschuss:

Erster Microprozessor auf einem Chip: Intel 4004 – geboren (2300 Transistoren)



Beispiele von Computerfamilien

x86-Architektur

<https://de.wikipedia.org/wiki/X86-Prozessor>

<https://www.datacenter-insider.de/was-ist-x86-a-e0e75a0822bf8f118d022ea28a79a7b6/>

- Prozessoren der x86 Familie ähneln sich im Befehlssatz:
Kompatibilität „in die Vergangenheit“. Auch
- Neue Hardwareleistungsmale können durch alte Programme nicht ausgenutzt werden
- Intel und AMD sind die führenden Anbieter.

Chip	Monat/ Jahr der Einführung	Takt- frequenz in MHz	Anzahl der Tran- sistoren	Adressier- barer Speicher	Bemer- kungen
4004	04/1971	0,108	2 300	640 Byte	Erster Mikroprozessor auf einem Chip
8008	04/1972	0,108	3 500	16 KB	Erster 8-Bit-Mikroprozessor
8080	04/1974	2	6 000	64 KB	Erste universelle CPU auf einem Chip
8086	06/1978	5–10	29 000	1 MB	Erste 16-Bit-CPU auf einem Chip
8088	06/1979	5–8	29 000	1 MB	Im IBM-PC eingesetzte „abgespeckte“ Version des 8086
80286	02/1982	8–12	134 000	16 MB	Speicherschutz vorhanden
80386	10/1985	16–33	275 000	4 GB	Erste 32-Bit-CPU
80486	04/1989	25–100	1,2 Mill.	4 GB	Integrierter 8-KB-Cache
Pentium	03/1993	60–233	3,1 Mill.	4 GB	Zwei Pipelines; spätere Modelle mit MMX
Pentium Pro	03/1995	150–200	5,5 Mill.	4 GB	Zwei integrierte Cache-Ebenen
Pentium II	05/1997	233–400	7,5 Mill.	4 GB	Pentium Pro plus MMX-Befehle
Pentium III	02/1999	650–1400	9,5 Mill.	4 GB	SSE-Befehle für 3-D-Grafiken
Pentium 4	11/2000	1300–3800	42 Mill.	4 GB	Hyperthreading; zusätzliche SSE-Befehle
Core Duo	1/2006	1600–3200	152 Mill.	2 GB	Zwei Kerne auf einem einzelnen Die
Core	7/2006	1200–3200	410 Mill.	64 GB	64-Bit-Architektur mit 4 Kernen (Quad-Core)
Core i7	1/2011	1100–3300	1160 Mill.	24 GB	Integrierter Grafikprozessor

16bit

32bit



- Computer „rechnen“, dh: sie führen Operationen auf Zahlen aus: + - * /
- Die Genauigkeit ist dabei endlich und feststehend.
- Computer arbeiten mit dem binären Zahlensystem und nicht mit dem Dezimalsystem.

⇒ von Computern verwendete Arithmetik unterscheidet sich von der Arithmetik, die uns Menschen geläufig ist.

Ziel dieses Kapitels:

Verständnis Realisierungsprinzipien der Zeichen-speicherung und -verarbeitung



Rechnerarithmetik: Was bedeutet:

„Die Genauigkeit ist dabei endlich und feststehend“ ?

Rechnen mit Papier und Bleistift:

- „unbegrenzter“ Speicherplatz
- Wir „speichern“ ab:
 - $a = 200$
 - $b = 70$
 - $c = 40$
- Erste Rechnung: $a + (b-c)$
 $= 200 + (70-40)$
 $= 200 + 30 = 230$

Der Rechner rechnet:

- Speicherplatz ist begrenzt:
z.B. 8 bit für eine ganze Zahl.
- Wir speichern ab:
 - $a = 200 = (128 + 64 + 8) = \textcolor{blue}{11001000}$
 - $b = 70 = (64 + 4 + 2) = \textcolor{green}{01000110}$
 - $c = 40 = (32+8) = \textcolor{pink}{00101000}$
 - $30 = (16+8+4+2) = \textcolor{brown}{00011110}$
 - $230 = (128+64+32+4+2) = \textcolor{blue}{11100110}$
- Erste Rechnung: $a + (b-c)$
 $= \textcolor{blue}{11001000} + (\textcolor{green}{01000110} - \textcolor{pink}{00101000})$
 $= \textcolor{blue}{11001000} + \textcolor{brown}{00011110}$
 $= \textcolor{blue}{11100110}$



Rechnerarithmetik: Was bedeutet:

„Die Genauigkeit ist dabei endlich und feststehend“ ?

Rechnen mit Papier und Bleistift:

- „unbegrenzter“ Speicherplatz
- Wir „speichern“ ab:
 - $a = 200$
 - $b = 70$
 - $c = 40$
- Erste Rechnung: $a + (b-c)$
 $= 200 + (70-40)$
 $= 200 + 30 = 230$
- Zweite Rechnung: $(a + b) - c$
 $= (200 + 70) - 40$
 $= 270 - 40 = 230$

Der Rechner rechnet:

- Speicherplatz ist begrenzt:
z.B. 8 bit für eine ganze Zahl.
- Wir speichern ab:
 - $a = 200 = (128 + 64 + 8) = \textcolor{blue}{11001000}$
 - $b = 70 = (64 + 4 + 2) = \textcolor{green}{01000110}$
 - $c = 40 = (32+8) = \textcolor{magenta}{00101000}$
 - $30 = (16+8+4+2) = \textcolor{brown}{00011110}$
 - $230 = (128+64+32+4+2) = \textcolor{black}{11100110}$
- Erste Rechnung: $a + (b-c)$
 $= \textcolor{blue}{11001000} + (\textcolor{green}{01000110} - \textcolor{magenta}{00101000})$
 $= \textcolor{blue}{11001000} + \textcolor{brown}{00011110}$
 $= \textcolor{black}{11100110}$
- Zweite Rechnung: $(a + b) - c$
 $= (\textcolor{blue}{11001000} + \textcolor{green}{01000110}) - \textcolor{magenta}{00101000}$

Entspricht 270

NICHT SPEICHERBAR

bei oben vorgegebenem Speicherplatz'



Codierung

Def: Abbildung (eindeutige Zuordnung nach einer bestimmten Vorschrift) einer Menge M1 in eine andere Menge M2.

Gründe:

- Zusammenfassung von unterschiedlichen Informationen
- Normierung
- Fehlererkennung und –korrektur
- Komprimierung
- Verschlüsselung
- Übertragungsvorteile
- Technische Randbedingungen

Elemente eines Codes:

- Bitstrings unterschiedlicher Länge:
Zahlen, Zeichen, Befehle, . . .
 - 4 Bit = Tetrade/Nibbel/Half-Byte
<https://de.wikipedia.org/wiki/Nibble>
 - 8 Bit = Byte
 - 16 Bit = Wort
[https://en.wikipedia.org/wiki/Word_\(computer_architecture\)](https://en.wikipedia.org/wiki/Word_(computer_architecture))
 - 32 Bit = Doppelwort
 - 64 Bit = Quadwort
 - Anzahl der Bits $n \rightarrow 2^n$ Info (i.d.R.)



Zahlensysteme

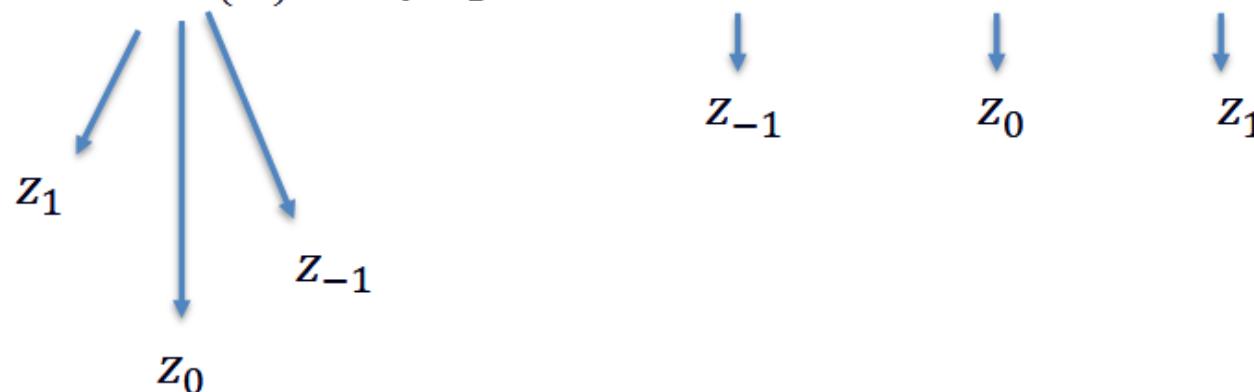
Stellenwertsystem/ Positionssystem/ Polyadisches System <https://de.wikipedia.org/wiki/Stellenwertsystem>

$$X_{(b)} = \sum_{i=-m}^n z_i \times b^i$$

- b = Basis
- $z = \{ \text{ganze Zahlen} \}$

Beispiel: Im Dezimalsystem (Zehnersystem) die Zahl 73,5:

$$73,5_{(10)} = \sum_{i=-1}^1 z_i \times 10^i = 5 \times 10^{-1} + 3 \times 10^0 + 7 \times 10^1 = (0,5 + 3 + 70) = 73,5$$



Zahlensysteme

System	Basis	Bit/ Ziffer	Zeichenvorrat
Dual	2	1	0,1
Oktal	8	3	0-7
Dezimal (BCD)	10	4	0-9
Hexadezimal	16	4	0-9, A-F

Beispiel: Dual: $1011,01_{(2)} = 11,25_{(10)}$: Dezimal

$$1011,01_{(2)} = \sum_{i=-2}^3 z_i \times b^i = 1 \times 2^{-2} + 0 \times 2^{-1} + 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = \frac{1}{4} + 0 + 1 + 2 + 0 + 8 = 11,25_{(10)}$$

The diagram illustrates the conversion of the binary number $1011,01_{(2)}$ to decimal. Blue arrows point from each digit to its corresponding power of 2 in the sum equation. The digits are labeled $z_3, z_1, z_{-1}, z_0, z_2, z_{-2}$.

https://en.wikipedia.org/wiki/Binary-coded_decimal



Umwandlung (1) : von beliebiger Basis in Dezimalsystem

Direkt mit Formel, „einfach“:

- Dual: $101110_{(2)} = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 32 + 8 + 4 + 2 = 46_{(10)}$: Dezimal
- Oktal: $56_{(8)} = 5 \times 8^1 + 6 \times 8^0 = 5 \times 8 + 6 \times 1 = 40 + 6 = 46_{(10)}$: Dezimal
- Hexadezimal: $2E_{(16)} = 2 \times 16^1 + E \times 16^0 = 2 \times 16 + 14 \times 1 = 32 + 14 = 46_{(10)}$: Dezimal



Umwandlung (2): Dual, Oktal und Hexadezimalzahlen ineinander überführen



System	Basis	Bit/ Ziffer	Zeichenvorrat
Dual	2	1	0,1
Oktal	<u>8</u>	3	0-7
Dezimal (BCD)	10	4	0-9
Hexadezimal	16	4	0-9, A-F

Dual-, Oktal- und Hexadezimalzahlen lassen sich durch Stellenzusammenfassungen einfach ineinander überführen:

$$46_{(10)} = 101110_{(2)} = \underbrace{101}_{5_{(8)}} \underbrace{110}_{6_{(8)}}_{(2)} = \underbrace{10}_{2_{(16)}} \underbrace{1110}_{E_{(16)}}_{(2)} = 56_{(8)} = 2E_{(16)}$$

$$\underbrace{5}_{5_{(8)}} \underbrace{6}_{6_{(8)}} \quad \underbrace{2}_{2_{(16)}} \underbrace{E}_{E_{(16)}}_{(2)}$$

$$56_{(8)}$$

$$2E_{(16)}$$



Umwandlung (3): von Dezimalsystem in Dual, Oktal oder Hexadezimalzahl

Methoden:

- „2er, 8er oder 16er Potenzen abziehen..“ : gut für hands on
- Klassischer / Euklidischer Algorithmus
- Abwandlung des Horner Schemas



1. Ganzzahlteil umwandeln (Division durch 2)

Beispiel: 13,375

Ganzzahlteil: 13

Immer wieder durch 2 teilen, den Rest merken:

Rechnung	Ergebnis	Rest
$13 \div 2$	6	1
$6 \div 2$	3	0
$3 \div 2$	1	1
$1 \div 2$	0	1

Reste von unten nach oben lesen → 1101

2. Nachkommateil umwandeln (Multiplikation mit 2)

Nachkommateil: 0,375

Immer mit 2 multiplizieren, den **Ganzzahlteil** notieren:

Rechnung	Ergebnis	Ganze Zahl
$0,375 \times 2$	0,75	0
$0,75 \times 2$	1,5	1
$0,5 \times 2$	1,0	1

3. Ergebnis zusammensetzen

$13,375_{10} = 1101,011_2$



Abbruch bei 0

Fertig!

Integer Repräsentation

1) Unsigned Integer (bzw. char)

- eine n-bit Sequenz von Binär-Ziffern $b_{n-1} b_{n-2} \dots b_0$
- als **Unsigned Integer** B_{10} interpretiert:

$$B_{10} = \sum_{i=0}^{n-1} b_i \times 2^i$$

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

Beispiel:
4-bit Sequenz



Integer Repräsentation

2) Vorzeichen-Betrags-Darstellung (Sign Magnitude Representation)

- eine n-bit Sequenz von Binär-Ziffern $b_{n-1} b_{n-2} \dots b_0$ in
- in als **sign magnitude Integer** B_{10} interpretiert:
$$B_{10} = \begin{cases} \sum_{i=0}^{n-2} b_i \times 2^i & \text{für } b_{n-1} = 0 \\ \sum_{i=0}^{n-2} b_i \times 2^i & \text{für } b_{n-1} = 1 \end{cases}$$

Anwendung:

- Am weitesten links stehende Bit (Most Significant Bit) MSB: Vorzeichenbit (0 für + und 1 für -)
- Die restlichen Bits: Betrag der Zahl
- Beispiel:
 - $|91_{10}| = 1011011_2$
 - $+91_{10} = 01011011_2$
 - $-91_{10} = 11011011_2$

Beobachtung:

- In Hochsprachen z.B. als signed (bzw. unsigned) int bzw. char bezeichnet
- $+0_{10} = -0_{10} = 00000000_2 = 10000000_2$
⇒ Erhöhter Aufwand bei Überprüfung auf Gleichheit

Mögliche Forderung:

- Nur eine Repräsentation der 0
- gleich viele positive und negative Zahlen
⇒ Nicht möglich (geht nur entweder..oder)

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

Beispiel:
4-bit Sequenz



Negative Binärzahlen

3) 1er Komplement (One's Complement)

<https://de.wikipedia.org/wiki/Einerkomplement>

- Ein Vorzeichenbit (0 für + und 1 für -)
- Um eine Zahl zu negieren:
 - ersetzt man jede 1 durch eine 0,
 - und jede 0 durch eine 1
 - Dies gilt auch für das Vorzeichenbit.
- Beispiel:
 - $|91_{10}| = \underline{1011011}_2$
 - $+91_{10} = 01011011_2$
 - $-91_{10} = 10100100_2$

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

Beispiel:
4-bit Sequenz



Negative Binärzahlen

4) 2er Komplement

- <https://de.wikipedia.org/wiki/Zweierkomplement>
- Ein Vorzeichenbit (0 für + und 1 für -)
- Um eine Zahl zu negieren:
 - ersetzt man jede 1 durch eine 0,
 - und jede 0 durch eine 1
 - Dies gilt auch für das Vorzeichenbit
 - Zusätzlich: **Addition einer 1 auf das LSB** (Least Significant Bit)
- Beispiel:
 - $|91_{10}| = 1011011_2$
 - $+91_{10} = 01011011_2$
 - $-91_{10} = 10100100_2 + \text{00000001}_2 = 10100101_2$
 - Probe: Dualzahl + Komplement = 01011011_2
 $+ 10100101_2 =$

$$100000000_2$$

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

Beispiel:
4-bit Sequenz



5) Offset Notation Excess (Excess 2^{n-1} Representation)

- https://en.wikipedia.org/wiki/Offset_binary
- Zahlenbereich von $z = [-2^{n-1}; 2^{n-1}-1]$
(Bsp n=4: 4 bit Sequenz: $z = [-2^3; -1] = [-8; +7]$)
- n : maximale Anzahl Bits (n-bit Sequenz)
- Darstellung:
 - geg: n
 - ges: y : zu jeder Zahl z wird addiert: 2^{n-1}
 - $y = [0; 2^n-1]$
- Beispiel:
 - $n = 8; z = [-2^{n-1}; 2^{n-1}-1] = [-128; +127]$
 - $2^{n-1} = 128$
 - $z = +91_{10}$
 - $y = +91_{10} + +128_{10} = 219_{10} = 11011011_2$
 - $z = -91_{10}$
 - $y = -91_{10} + 128_{10} = 37_{10} = 00100101_2$
 - Probe: $11011011_2 + 00100101_2 = 00000000_2$ und Overflow

Binär	5) Excess 2^{n-1}
0000	-8
0001	-7
0010	-6
0011	-5
0100	-4
0101	-3
0110	-2
0111	-1
1000	0
1001	+1
1010	+2
1011	+3
1100	+4
1101	+5
1110	+6
1111	+7

Beispiel:
4-bit Sequenz



Überblick

Binär	1) Unsigned	2) VZ-Betr	3) 1er Komp.	4) 2er Komp.	5) Excess 2^{n-1}
0000	0	+0	+0	+0	-8
0001	1	+1	+1	+1	-7
0010	2	+2	+2	+2	-6
0011	3	+3	+3	+3	-5
0100	4	+4	+4	+4	-4
0101	5	+5	+5	+5	-3
0110	6	+6	+6	+6	-2
0111	7	+7	+7	+7	-1
1000	8	-0	-7	-8	0
1001	9	-1	-6	-7	+1
1010	10	-2	-5	-6	+2
1011	11	-3	-4	-5	+3
1100	12	-4	-3	-4	+4
1101	13	-5	-2	-3	+5
1110	14	-6	-1	-2	+6
1111	15	-7	-0	-1	+7

Positive und negative Zahlen



Einschub: Codes

2 4 2 1

Dezimal	Binär	Hexadezimal	BCD	Stibitz	Aiken	Gray
0	0000	0	0000	0011	0000	0000
1	0001	1	0001	0100	0001	0001
2	0010	2	0010	0101	0010	0011
3	0011	3	0011	0110	0011	0010
4	0100	4	0100	0111	0100	0110
5	0101	5	0101	1000	1011	0111
6	0110	6	0110	1001	1100	0101
7	0111	7	0111	1010	1101	0100
8	1000	8	1000	1011	1110	1100
9	1001	9	1001	1100	1111	1101
10	1010	A	0001 0000	0100 0011	0001 0000	1111
11	1011	B	0001 0001	0100 0100	0001 0001	1110
12	1100	C	0001 0010	0100 0101	0001 0010	1010
13	1101	D	0001 0011	0100 0110	0001 0011	1011
14	1110	E	0001 0100	0100 0111	0001 0100	1001
15	1111	F	0001 0101	0100 1000	0001 1011	1000



Einschub: Binary-Codes: BCD

Binary Coded Dezimal (BCD) – (auch 8421 Code)

https://en.wikipedia.org/wiki/Binary-coded_decimal

Binär codierte Dezimalziffern (Dualzahl-Tetraden)

Bsp: $8127_{10} = \underline{1000} \underline{0001} \underline{0010} \underline{0111}_{BCD}$

8_{10} 1_{10} 2_{10} 7_{10}

Beobachtung:

- Tetraden = Dualzahlen : OK
- „Pseudo-Tetraden“ : 1010, 1011, 1100, 1101, 1110, 1111
 - Ungültig!
 - Kann für Fehlererkennung genutzt werden
 - Ursache für Speicher-Mehr-Bedarf
- 0000 : gültige Tetrade

<https://de.wikipedia.org/wiki/Tetrade>

Dezimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	0001 0000
11	0001 0001
12	0001 0010
13	0001 0011
14	0001 0100
15	0001 0101



Einschub: Binary-Codes: Stibitz-Code (Excess-3 – ohne <0 und >9)

<https://en.wikipedia.org/wiki/Excess-3>

- BCD + 0011 (jedes Code-Wort) => „offset“ = +3
- Codes für Ziffern 0 – 9 „rücken“ in die Mitte der zur Verfügung stehenden Codes 0000 bis 1111
- Ziffern 0 bis 4 spiegelbildlich komplementär zu 5 bis 9

Beobachtung:

- „Pseudo-Tetraden“:
0000, 0001, 0010, und
1101, 1110, 1111
- 0000, 1111 : ungültige Tetrade !
- Negationssymmetrisch
- Einfache 9er Komplementbildung: (Invertieren)

$$2_{10} = 0101_{\text{Stibitz}}$$

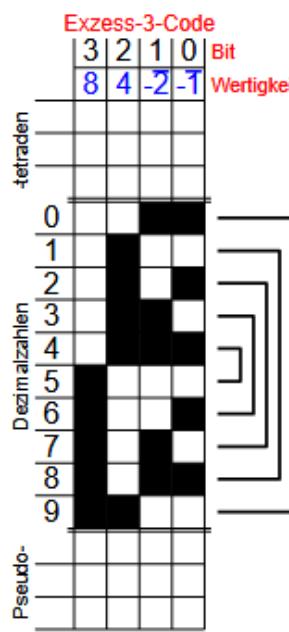
9er Komplement: $7_{10} = 1010_{\text{Stibitz}}$

$$+ 1_{10} = +0100_{\text{Stibitz}}$$

10er Komplement: $8_{10} = 1110_{\text{Stibitz}}$

$6_{10} - 2_{10} = 6_{10} + 8_{10} = 14_{10} = 4_{10}$

$1001_{\text{Stibitz}} - 0101_{\text{Stibitz}} = 1001_{\text{Stibitz}} + 1110_{\text{Stibitz}} = 10111_{\text{Stibitz}} = 0111_{\text{Stibitz}} = 4_{10}$



Dezimal	BCD	Stibitz
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100
10	0001 0000	0100 0011
11	0001 0001	0100 0100
12	0001 0010	0100 0101
13	0001 0011	0100 0110
14	0001 0100	0100 0111
15	0001 0101	0100 1000

<https://www.rechnerlexikon.de/artikel/Neunerkomplement>

Einschub: Binary-Codes: Gray Code

<https://de.wikipedia.org/wiki/Gray-Code>

- stetiger Code
- benachbarte Codewörter unterscheiden sich nur in einer einzigen binären Ziffer
- Hamming-Distanz benachbarter Codewörter : 1
- Übertragungsfehler bei sich kontinuierlich ändernden digitalen Signalen auf mehradrigen Leitungen werden so verringert, dass sich unterschiedliche Laufzeiten nicht auswirken können

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



Einschub: Codes und Redundanz R („Überschüssigkeit“)

Redundante Codes verwenden nicht alle möglichen Codierungen zur Informationsdarstellung.

Redundanz ist zentrales
Mittel der Fehlertoleranz

Beispiel: BCD:

- 4 mögliche Stellen im Codewort ergibt maximal 16 mögliche Codierungen
- genutzt werden nur 10 mögliche Codierungen: 0000 – 1001 (0-9)
- nicht genutzt: 1010, 1011, 1100, 1101, 1110, 1111

Definition:

- Redundanz $R := y - ld(x) = y - \log_2(x) = y - \log_2(x) = y - \frac{\ln(x)}{\ln(2)}$
- y : Zahl der benutzen Stellen (im Codewort)
- x : Menge der darstellbaren / codierbaren Zeichen

Beispiele:

- $R_{BCD} = 4 - ld(10) = 0,68 \text{ [bit]}$ => redundanter Code
- $R_{ASCII} = 7 - ld(128) = 0 \text{ [bit]}$ => vollständiger, nicht redundanter Code



Einschub: Parität

Paritätsbit einer Folge von Bits zeigt an:
ob Anzahl der mit „1“ belegten Bits der Folge
gerade oder ungerade ist:

- Codewort = Paritätsbit und Informationswort
- 2 Arten:
 - Gerade Parität: Anzahl der „1“ im gesamten Codewort gerade (even)
 - Ungerade Parität: Anzahl der „1“ im gesamten Codewort ungerade (odd)

Nachricht (Informationswort)	Codewort (even parity)	Codewort (odd parity)
000	0000	1000
001	1001	0001
010	1010	0010
011	0011	1011
100	1100	0100
101	0101	1101
110	0110	1110
111	1111	0111

Nutzen:

- Erkennung von Übertragungsfehlern



Einschub: Hamming Distanz h

<https://de.wikipedia.org/wiki/Hamming-Abstand>

- Maße für die Unterschiedlichkeit von Zeichenketten.
- Hamming-Abstand zweier binärer Codewörter: Anzahl unterschiedlicher Stellen.

Hamming-Distanz eines Codes:

- Minimum aller Hamming-Abstände zwischen allen Wörtern innerhalb des Codes

Beispiel:

- Ein Code besteht aus 3 Wörtern $x = 00110$, $y = 00101$, und $z = 01110$
- Hamming-Abstand zwischen x und y : 2 $001\textcolor{red}{1}0$
 $001\textcolor{red}{0}1$
- Hamming-Abstand zwischen x und z : 1 00110
 $0\textcolor{red}{1}110$
- Hamming-Abstand zwischen y und z : 3 00101
 01110



Hamming-Distanz
des Codes: **$h = 1$**

Einschub: Hamming Distanz h

Motivation / Problem:

Bei Datenübertragung eines Codewortes entsteht ein (einziger) Bitfehler.

Reflektion:

- $h=1$:
 - Ein Bitfehler ist **nicht** zu erkennen, da ein Codewort durch den Bitfehler in ein anderes gültiges Codewort überführt wird.
 - Fehlerkorrektur nicht möglich
- $h=2$:
 - Ein Bitfehler ist zu erkennen, da ein Codewort durch einen Bitfehler in ein **ungültiges** Codewort überführt wird.
 - Fehlerkorrektur nicht möglich, da ungültiges Codewort durch Bitfehler von 2 gültigen Codewörtern entstehen kann.
- $h=3$:
 - Ein Bitfehler ist zu erkennen, da ein Codewort durch einen Bitfehler in ein **ungültiges** Codewort überführt wird
 - Fehlerkorrektur ist möglich, da ungültiges Codewort durch Bitfehler von 2 gültigen Codewörtern entstehen kann.



Einschub: Hamming Distanz h

Aufbau von Codes die Bitfehler erkennen (und ggf. auch korrigieren) können:

- Informations-bits m
- Korrektur-bits k
- Wortlänge n = m+k

Mit k Prüf-bits bzw. Korrektur-bits können 2^k (Fehler)zustände gekennzeichnet werden



Einschub: Cyclic Redundancy Check (CRC)

Beispiel: Paritäts-bit: 2 Bitfehler im gleichen Codewort werden nicht erkannt:

Sender:

- Informationswort (Message): (5 mal die „1“): 1101101
- Even parity Codewort (6 mal die „1“): 11101101
- Übertragung des Codeworts zum Empfänger

Empfänger:

- Codewort wird empfangen: 2 Bitfehler:
11001111
1001111
- Informationswort:

Even Parity-Bit check: ergibt: OK
Im Informationswort wird
kein Fehler erkannt



⇒ Bessere Lösung notwendig als Parity-Bit!

Schönes Video zu Fehlererkennung – CRC:

https://www.youtube.com/watch?v=gTVD8z_ZYhM

Einschub: Cyclic Redundancy Check (CRC)

- Fehlererkennungsverfahren mit besserer Fehlererkennungsrate als Parity-Bits
- Es wird Redundanz hinzugefügt: Diese Redundanz besitzt keinen zusätzlichen Informationsgehalt
- Zyklischer Code (https://de.wikipedia.org/wiki/Zyklischer_Code)
dh: ist $(a_0, a_1, a_2, \dots, a_{n-2}, a_{n-1})$ ein Codewort von **C**, dann sind auch $(a_1, a_2, \dots, a_{n-2}, a_{n-1}, a_0)$, $(a_2, \dots, a_{n-2}, a_{n-1}, a_0, a_1), \dots, (a_{n-1}, a_0, a_1, a_2, \dots, a_{n-2})$ Codewörter von **C**
- Berechnung des CRC-Worts beruht auf Polynom-Division (modulo 2) in Modulo-2-Arithmetik
- Gegeben: ein Generator-Polynom G: Dies ist auf Empfängerseite und Senderseite bekannt

Modulo-2-Arithmetik:

- Keine Beachtung des Übertrags, entspricht XOR
- Bitweise:
 - $0 + 0 = 0$ $0 - 0 = 0$
 - $0 + 1 = 1$ $0 - 1 = 1$
 - $1 + 0 = 1$ $1 - 0 = 1$
 - $1 + 1 = 0$ $1 - 1 = 0$



Einschub: Cyclic Redundancy Check (CRC)

Beispiel:

Sender:

- Informationswort D = 1010101010 (10 bit)
9876543210

- Generator $G = 10011$ (5 bit)

- Darstellung als Polynome:

- $D(x) = x^9 + x^7 + x^5 + x^3 + x^1$
 - $G(x) = x^4 + x^1 + x^0$

- Berechnung Rest (Remainder) R durch Polynomdivision:

- $G = \underline{10011}$

- L = „Länge G“ = 5 [bit]

- Definition Platzhalter: L-1 (4-bit) an „0en“: 0000

- Polynomdivision: D Platzhalter : G

$$1010101010\underline{0000} : 10011 =$$

Modulo-2-Arithmetik!
=> Gesucht: Rest!



Einschub: Cyclic Redundancy Check (CRC)

10101010100000 : 10011

10011

0011001

10011

010100

10011

0011110

10011

011010

10011

010010

10011

0000100

Rest == CRC



Einschub: Cyclic Redundancy Check (CRC)

Beispiel:

Sender:

- Informationswort D = **1010101010** (10 bit)
9876543210
- Generator G = **10011** (5 bit)
- Darstellung als Polynome:
 - $D(x) = x^9 + x^7 + x^5 + x^3 + x^1$
 - $G(x) = x^4 + x^1 + x^0$
- Berechnung Rest (Remainder) R durch Polynomdivision:
 - G = 10011
 - L = „Länge G“ = 5 [bit]
 - Definition Platzhalter: L-1 (4-bit) an „0en“: **0000**
 - Polynomdivision: D Platzhalter : G
10101010100000 : 10011
Rest R = 0100 == CRC
 - **Codewort:** **1010101010 0100**
Dieses Codewort wird vom Sender an den Empfänger übertragen

Modulo-2-Arithmetik!
=> Gesucht: Rest!



Einschub: Cyclic Redundancy Check (CRC)

Beispiel:

Empfänger: empfängt ohne Bitfehler

- empfängt Codewort: **1010101010 0100**
- kennt Generator $G = 10011$ (5 bit)
- berechnet:
$$\begin{array}{r} 1010101010 0100 \\ \underline{-} 10011 \\ \hline 0011001 \\ \underline{-} 10011 \\ \hline 010100 \\ \underline{-} 10011 \\ \hline 0011110 \\ \underline{-} 10011 \\ \hline 011010 \\ \underline{-} 10011 \\ \hline 010011 \\ \underline{-} 10011 \\ \hline 000000 \end{array}$$

Modulo-2-Arithmetik!
=> Gesucht:
Gibt es einen Rest?

Rest = 0 => Kein Bitfehler



Einschub: Cyclic Redundancy Check (CRC)

Beispiel:

Empfänger: empfängt mit Bitfehler

- originales Codewort: 1010101010 0100

- empfangenes Codewort: 1110100010 0100

- berechnet: 1110100010 0100 : 10011

10011

00010000

10011

-00011100

10011

011111

10011

-011000

10011

-010110

10011

-00101

2 Bitfehler!

Modulo-2-Arithmetik!
=> Gesucht:
Gibt es einen Rest?

Rest != 0
=> Empfänger stellt fest das Bitfehler vorliegt.



Integer Arithmetik im 2er Komplement

$$\begin{array}{r} 1001 = -7 \\ +0101 = 5 \\ \hline 1110 = -2 \end{array}$$

(a) $(-7) + (+5)$

$$\begin{array}{r} 0011 = 3 \\ +0100 = 4 \\ \hline 0111 = 7 \end{array}$$

(c) $(+3) + (+4)$

$$\begin{array}{r} 0101 = 5 \\ +0100 = 4 \\ \hline 1001 = \text{Overflow} \end{array}$$

(e) $(+5) + (+4)$

$$\begin{array}{r} 1100 = -4 \\ +0100 = 4 \\ \hline 10000 = 0 \end{array}$$

(b) $(-4) + (+4)$

$$\begin{array}{r} 1100 = -4 \\ +1111 = -1 \\ \hline 11011 = -5 \end{array}$$

(d) $(-4) + (-1)$

$$\begin{array}{r} 1001 = -7 \\ +1010 = -6 \\ \hline 10011 = \text{Overflow} \end{array}$$

(f) $(-7) + (-6)$

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



Integer Arithmetik im 2er Komplement

Overflow-Regel

- 2 Zahlen werden addiert
- Beide sind positive, oder beide sind negativ.
- **Resultat hat anderes Vorzeichen**

⇒ then Overflow

$$\begin{array}{r} 1100 = -4 \\ +0100 = 4 \\ \hline 10000 = 0 \end{array}$$

(b) $(-4) + (+4)$

$$\begin{array}{r} 1100 = -4 \\ +1111 = -1 \\ \hline 11011 = -5 \end{array}$$

(d) $(-4) + (-1)$

$$\begin{array}{r} 0101 = 5 \\ +0100 = 4 \\ \hline \text{neg. } 1001 = \text{Overflow} \end{array}$$

(e) $(+5) + (+4)$

$$\begin{array}{r} 1001 = -7 \\ +1010 = -6 \\ \hline 10011 = \text{Overflow} \end{array}$$

pos. (f) $(-7) + (-6)$

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



Integer Arithmetik im 2er Komplement

$$\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array}$$

(a) M = 2 = 0010
 S = 7 = 0111
 -S = 1001

$$\begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline 10011 = 3 \end{array}$$

(b) M = 5 = 0101
 S = 2 = 0010
 -S = 1110

$$\begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline 11001 = -7 \end{array}$$

(c) M = -5 = 1011
 S = 2 = 0010
 -S = 1110

$$\begin{array}{r} 0101 = 5 \\ +0010 = 2 \\ \hline 0111 = 7 \end{array}$$

(d) M = 5 = 0101
 S = -2 = 1110
 -S = 0010

$$\begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline 1110 = \text{Overflow} \end{array}$$

(e) M = 7 = 0111
 S = -7 = 1001
 -S = 0111

$$\begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline 10110 = \text{Overflow} \end{array}$$

(f) M = -6 = 1010
 S = 4 = 0100
 -S = 1100

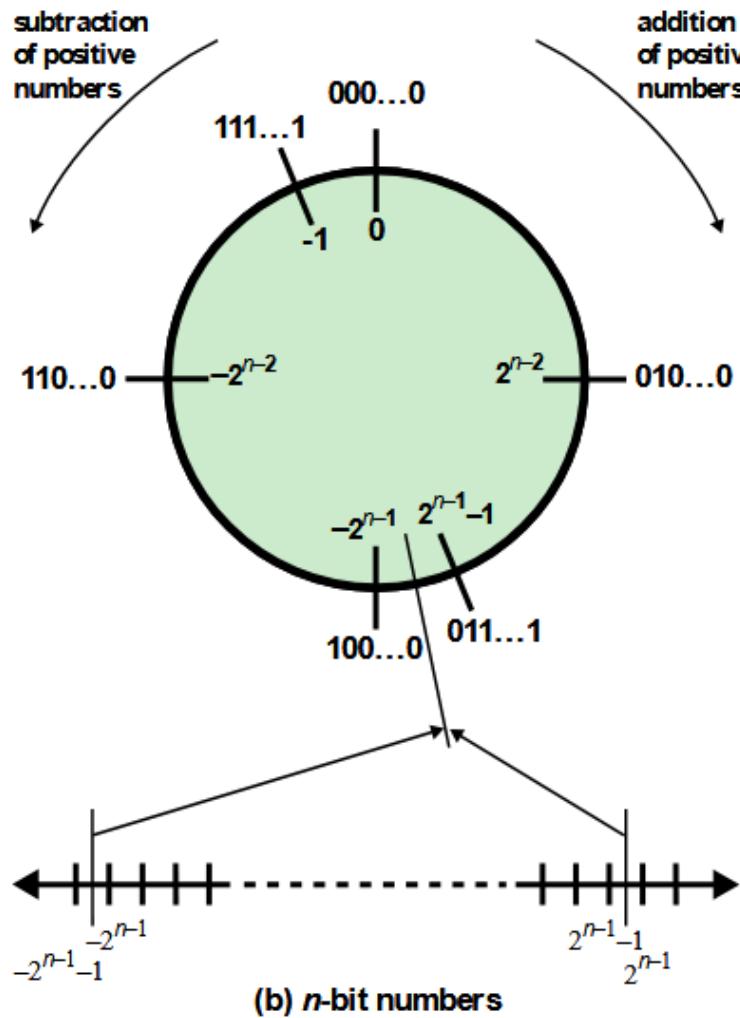
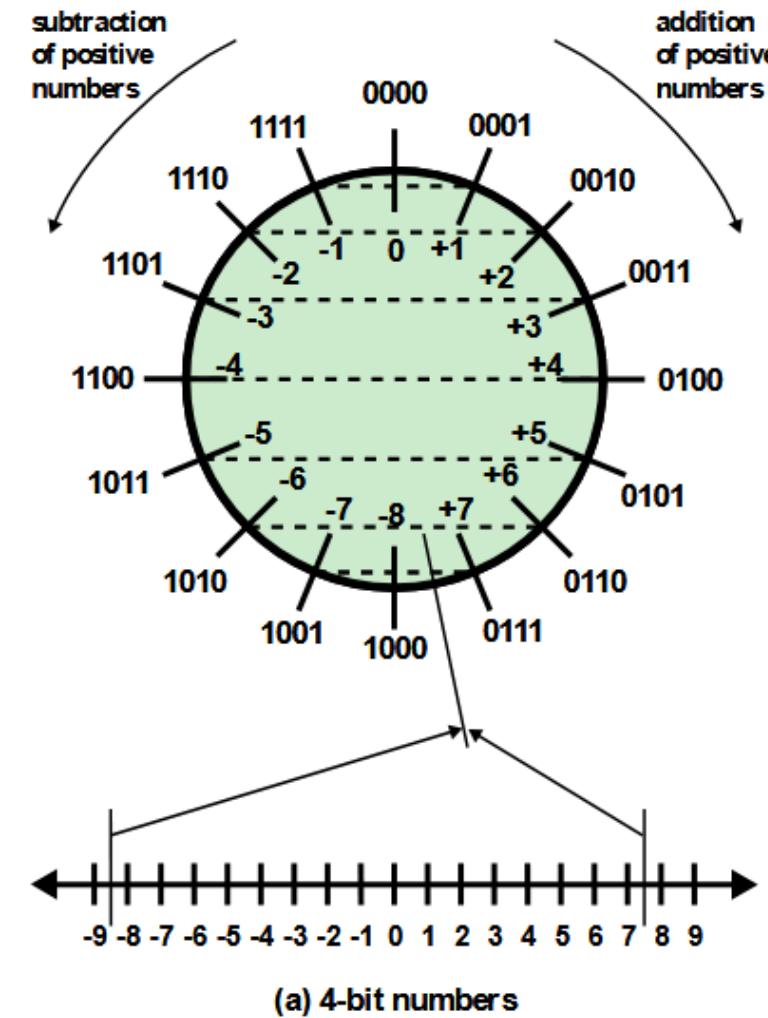
Subtraktion:

- Minuend – Subtrahend
- Minuend + (2er-Komplement vom Subtrahend)
- Overflow-Regel gilt (wie bei jeder Addition)

1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1



Integer Arithmetik im 2er Komplement



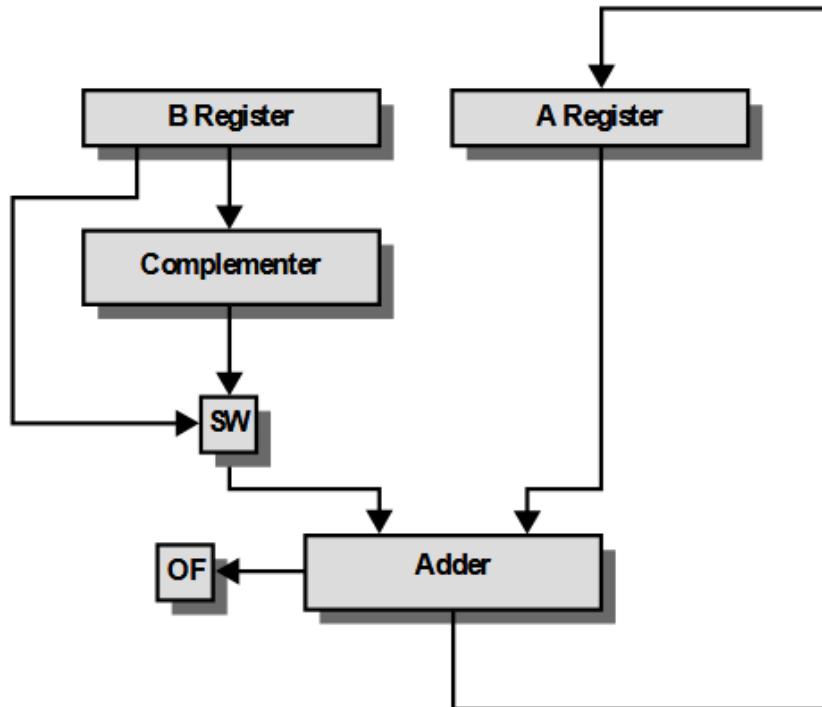
Binär	4) 2er Komp.
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1



Integer Arithmetik im 2er Komplement

Block-Diagramm für Hardware-basierte Addition (und Subtraktion)

- Bausteine und Datenpfad
- Zentrales Element: Binary-Adder:
 - Overflow Indicator: 1 bit
- Zusätzlich benötigt:
 - Steuer-Signal: Addition oder Subtraktion?



OF = overflow bit

SW = Switch (select addition or subtraction)



Gleitkommazahlen

Motivation: (Extrembeispiel)

- Masse Elektron: $m_e = 9 \times 10^{-28}$ g
- Masse Sonne: $m_S = 2 \times 10^{33}$ g

Bisher: Festkommazahlen:

$$Z_{(10)} = \sum_{i=-m}^n z_i \times 10^i = z_n \times 10^n + z_{n-1} \times 10^{n-1} + \dots + z_0 \times 10^0 + z_{-1} \times 10^{-1} + \dots + z_{-m} \times 10^{-m}$$

Darstellung (Basis 10):

$$m_e = 00000000000000000000000000000000,00000000000000000000000000000000_0 g$$

$$m_S = 20000000000000000000000000000000,00000000000000000000000000000000_0 g$$

34 Stellen (vor dem Komma)

28 Stellen (nach dem Komma)

62 Stellen

Probleme mit dieser Darstellung ?

Genauigkeit: Masse der Sonne ist nicht auf 62 Stellen bekannt...



Grundlagen Gleitkommaarithmetik

Trennung von Wertebereich und Genauigkeit (wissenschaftliche Notation)

$$n = m \times 10^e$$

- Mantisse m
gebrochener Anteil
- Exponent e
positive oder negative Ganzzahl

Beispiele:

- $3,14 = 3,14 \times 10^0 = 0,314 \times 10^1 = 0,0314 \times 10^2$
- $0,000001 = 0,1 \times 10^{-5} = 1 \times 10^{-6}$
- $1941 = 0,1941 \times 10^4 = 19,41 \times 10^2$

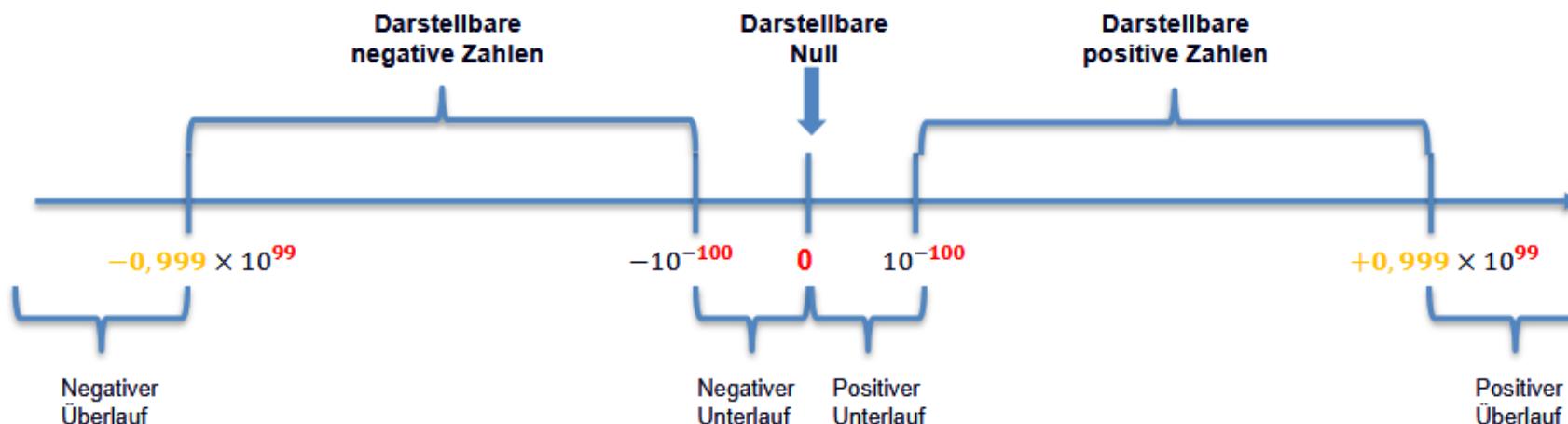
} Floating-Point Darstellung



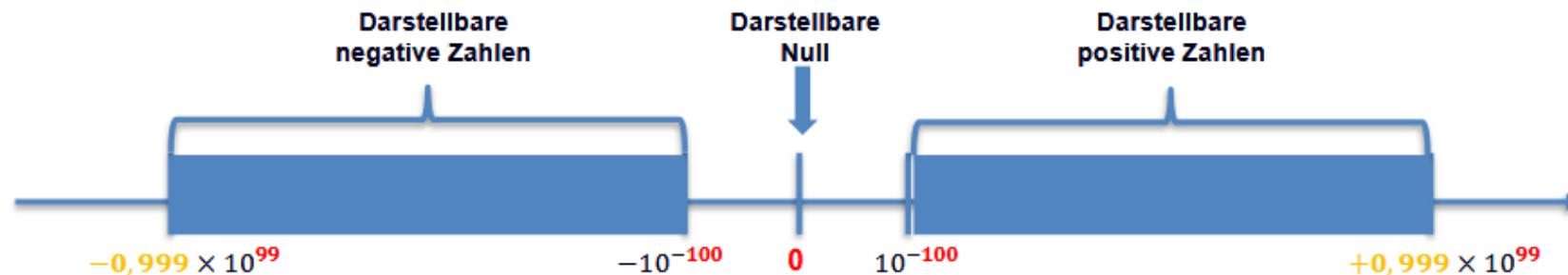
Grundlagen Gleitkommaarithmetik

Betrachtung am Beispiel

- $n = m \times 10^e$
- vorzeichenbehaftete dreistellige Mantisse
 - $0,1 \leq |m| < 1$
 - oder $m = 0$
- vorzeichenbehafteter zweistelliger ganzzahliger Exponent e
- Darstellung:
 - Kleinste mögliche Zahl: $m = -0,999; e = 99 \Rightarrow -0,999 \times 10^{99}$
 - größte mögliche Zahl: $m = +0,999; e = 99 \Rightarrow +0,999 \times 10^{99}$
 - kleinste mögliche positive Zahl $m = +0,100; e = -99 \Rightarrow +0,100 \times 10^{-99} = 10^{-100}$
 - größte mögliche negative Zahl $m = -0,100; e = -99 \Rightarrow -0,100 \times 10^{-99} = -10^{-100}$



Grundlagen Gleitkommaarithmetik



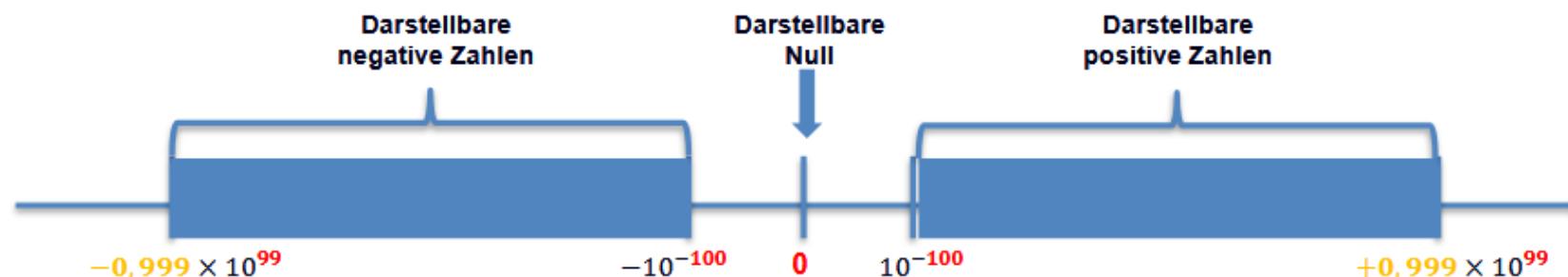
Vergleich der Dichte der darstellbaren negativen und positiven Zahlen mit den reellen Zahlen:

- Zwischen zwei beliebigen reellen Zahlen x und y gibt es eine Zahl $z = \frac{x+y}{2}$
=> reelle Zahlen kontinuierlich,
zwischen 10^{-100} und $+0,999 \times 10^{99}$ gibt es unendlich viele reelle Zahlen.
- Dies gilt nicht für unser Beispiel:
Zwischen
 $x = +0,999 \times 10^{99}$ und
 $y = +0,998 \times 10^{99}$
gibt es keine darstellbare Zahl!
- In unserem Beispiel lassen sich nur 358.201 verschiedene Zahlen darstellen
=> Kein Kontinuum
- Absoluter Abstand ist nicht durchgehend gleich:
 - Zwischen $0,999 \times 10^{99}$ und $+0,998 \times 10^{99}$ ist der Abstand $0,001 \times 10^{99}$
 - Zwischen $0,101 \times 10^{-99}$ und $0,100 \times 10^{-99}$ ist der Abstand $0,001 \times 10^{-99}$

Unterschied von
 10^{198}



Grundlagen Gleitkommaarithmetik



- Absoluter Abstand ist nicht durchgehend gleich:
 - Zwischen $0,999 \times 10^{99}$ und $+0,998 \times 10^{99}$ ist der Abstand $0,001 \times 10^{99}$
 - Zwischen $0,100 \times 10^{-99}$ und $0,101 \times 10^{-99}$ ist der Abstand $0,001 \times 10^{-99}$
 - Relativer Abstand ist auch nicht durchgehend gleich, aber viel geringer:
 - Zwischen $0,999 \times 10^{99}$ und $+0,998 \times 10^{99}$ ist der relative Abstand $\frac{0,001 \times 10^{99}}{0,999 \times 10^{99}} = \frac{0,001}{0,999}$
 - Zwischen $0,101 \times 10^{-99}$ und $0,100 \times 10^{-99}$ ist der relative Abstand $\frac{0,001 \times 10^{-99}}{0,101 \times 10^{-99}} = \frac{0,001}{0,101}$
- Unterscheid von 10^{198}*
- Unterscheid von 10^1*
- ⇒ Da der relative Abstand sich nicht gravierend ändert über den darstellbaren Bereich,
ist der durch Runden entstehende relative Fehler gering.

Schlussfolgerung aus Beispiel:

Eine Änderung der Ziffernanzahl der Mantisse oder im Exponenten bringt keine Änderung der Beobachtung.



Gleitkommazahlendarstellung nach IEEE 754

1985 festgelegte Darstellung von Gleitkommazahlen, die heute (fast) jeder Rechner beherrscht

- IEEE Standard Single Precision (32 bit)

- $432_{10} = 110110000_2 = +1, \underbrace{101100000000000000000000}_2 \times 2^8$
9 Ziffern 23 Nachkomma-Ziffern

- 1 Vorzeichenbit: 0 positiv / 1 negativ
Bedeutung: Ist die Zahl positiv oder negativ – gilt nicht für Exponent
Bsp: 0
- 8 Bit für den Exponent (bzw. Charakteristik)
Excess-127 Notation
8 in Excess-127 Notation = 1000 0111
 $8 := e - 127 \Rightarrow e = 135$
- 23 Bit für Mantisse
 - Mantisse wird normalisiert: 1, x..x
 - 1 wird nicht gespeichert
 - Gespeichert wird: x..x:
101100000000000000000000
- BSP:**
01000011110110000000000000000000

Binär	Unsigned „e“	Excess-127 „char“
0000 0000	0	-127
0000 0001	1	-126
0000 0010	2	-125
...
0111 1101	125	-2
0111 1110	126	-1
0111 1111	127	0
1000 0000	128	+1
...
1000 0111	135	+8
...
1111 1101	253	+126
1111 1110	254	+127
1111 1111	255	+128



Gleitkommazahlendarstellung nach IEEE 754

1985 festgelegte Darstellung von Gleitkommazahlen, die heute (fast) jeder Rechner beherrscht

- IEEE Standard Double Precision (64 bit)

$432_{10} = 110110000_2 = +1, \underbrace{10110000000000000000000000000000}_\text{9 Ziffern} \underbrace{00000000000000000000000000000000}_\text{52 Nachkomma-Ziffern} \times 2^8$

- 1 Vorzeichenbit: 0 positiv / 1 negativ

Bedeutung: Ist die Zahl positiv oder negativ – gilt nicht für Exponent

Bsp: 0

- 11 Bit für den Exponent (bzw. Charakteristik)

Excess-1023 Notation

8 in Excess-1023 Notation = 100 0000 0111

8 := e -1023 => e = 1031

- 52 Bit für Mantisse

- Mantisse wird normalisiert: 1,x..x

- 1 wird nicht gespeichert

- Gespeichert wird: x..x:

101100

- BSP:

010000000111101100

Binär	Unsigned	Excess-1023
000 0000 0000	0	-1023
000 0000 0001	1	-1022
000 0000 0010	2	-1021
...
011 1111 1101	1021	-2
011 1111 1110	1022	-1
011 1111 1111	1023	0
100 0000 0000	1024	+1
...
100 0000 0111	1031	+8
...
111 1111 1101	2045	+1022
111 1111 1110	2046	+1023
111 1111 1111	2047	+1024



Gleitkommazahlendarstellung nach IEEE 754

Einschränkungen/ Einteilung in 5 numerische Typen

	Vorzeichen	Charakteristik	Mantisse	
Normalisiert	\pm	$0 < \text{Exp} < \text{Max}$	Jedes Bitmuster	
Denormalisiert	\pm	0	Jedes Bitmuster ungleich 0	
Null	\pm	0	0	
Unendlichkeit	\pm	11..11	0	
NaN (not a number)	\pm	11..11	Jedes Bitmuster ungleich 0	Nicht verwenden als Operand!

} Erkennbar am Exp = C

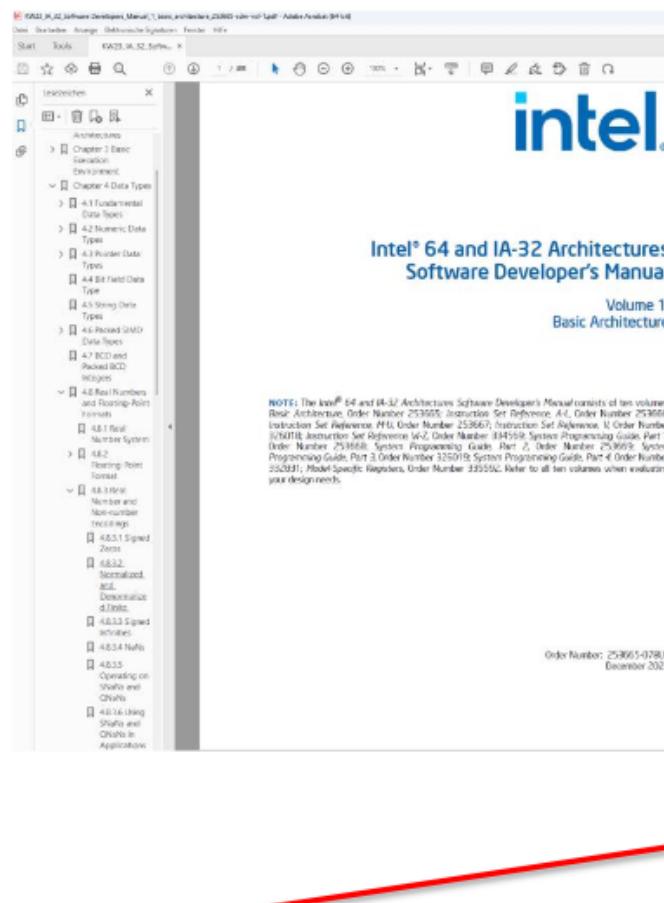
Eigenschaft	Einfache Genauigkeit	Doppelte Genauigkeit	
Bits im Vorzeichen	1	1	
Bits im Exponenten	8	11	
Bits in der Mantisse	23	52	
Bits insgesamt	32	64	
Exponentensystem	Exzess 127	Exzess 1023	
Exponentenbereich	-126 bis +127	-1022 bis +1023	Normalisiert: nicht 0, nicht 255, nicht 2047
Kleinste normalisierte Zahl	2^{-126}	2^{-1022}	$+1.00000000000000000000000_2 \times 2^{-126} = 2^{-126}$
Größte normalisierte Zahl	ca. 2^{128}	ca. 2^{1024}	$+1.11111111111111111111111_2 \times 2^{+127} \approx 2^{128}$
Dezimalbereich	ca. 10^{-38} bis 10^{38}	ca. 10^{-308} bis 10^{308}	
Kleinste nicht normalisierte Zahl	ca. 10^{-45}	ca. 10^{-324}	$+0.000000000000000000000001_2 \times 2^{-127} = 2^{-150}$

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>



Einschub:

Gleitkommazahlendarstellung nach IEEE 754



KW23_U_32 Software Developer's Manual 1 basic_architecture_259655-enm-en-1.pdf - Adobe Acrobat (PDF)

Start Tools KW23_U_32_Softw...

Lesezeichen X

Architectures > Chapter 3 Basic Execution Environment

< Chapter 4 Data Types > 4.1 Fundamental Data Types

> 4.2 Numeric Data Types

> 4.3 Pointer Data Types

> 4.4 Bit Field Data Type

> 4.5 String Data Types

> 4.6 Packed SIMD Data Types

> 4.7 BCD and Packed BCD Integers

> 4.8 Real Numbers and Floating-Point Formats

> 4.8.1 Real Number System

> 4.8.2 Floating-Point Format

> 4.8.3 Real Number and Non-number Encodings

> 4.8.3.1 Signed Zeros

> 4.8.3.2 Normalized and Denormalize formats

> 4.8.3.3 Signed Infinities

> 4.8.3.4 NaNs

> 4.8.3.5 Operating on Special and Quiet NaNs

> 4.8.3.6 Using Shifting and Clipping in Applications

NOTE: The Intel® 64 and IA-32 Architectures Software Developer's Manual consists of ten volumes: Basic Architecture, Order Number 259655; Instruction Set Reference, 4.0, Order Number 259666; Instruction Set Reference, 4.0A, Order Number 259667; System Programming Guide, Part 1, Order Number 259668; System Programming Guide, Part 2, Order Number 259669; System Programming Guide, Part 3, Order Number 325019; System Programming Guide, Part 4, Order Number 252931; Model-specific Registers, Order Number 335592. Refer to all ten volumes when evaluating your design needs.

Order Number: 25965-02805 December 2022

4.8.3.2 Normalized and Denormalized Finite Numbers

Non-zero finite numbers are divided into two classes: normalized and denormalized. The normalized finite numbers comprise all the non zero finite values that can be encoded in a normalized real number format between zero and ∞ . In the single precision floating-point format shown in Figure 4-12, this group of numbers includes all the numbers with biased exponents ranging from 1 to 254_{10} (unbiased), the exponent range is from -126_{10} to $+127_{10}$.

When floating-point numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros.

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called denormalized numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization may cause a loss of precision (the number of significant bits is reduced by the leading zeros).

When performing normalized floating-point computations, an IA-32 processor normally operates on normalized numbers and produces normalized numbers as results. Denormalized numbers represent an underflow condition. The exact conditions are specified in Section 4.9.1.5, "Numeric Underflow Exception (#U)."

A denormalized number is computed through a technique called gradual underflow. Table 4-6 gives an example of gradual underflow in the denormalization process. Here the single precision format is being used, so the minimum exponent (unbiased) is -126_{10} . The true result in this example requires an exponent of -129_{10} in order to have a

4-14 Vol.1

DATA

Mantisse beginnt mit 1,XXX

Mantisse beginnt mit 0,XXX

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result. The Intel 64 and IA-32 architectures deal with denormal values in the following ways:

- It avoids creating denormals by normalizing numbers whenever possible.
- It provides the floating-point underflow exception to permit programmers to detect cases when denormals are created.
- It provides the floating-point denormal-operand exception to permit procedures or programs to detect when denormals are being used as source operands for computations.

* Expressed as an unbiased decimal number.

Operation	Sign	Exponent*	Significant
True Result	0	-129	1.0101110000..00
Denormalize	0	-128	0.10101110000..00
Denormalize	0	-127	0.01010111000..00
Denormalize	0	-126	0.0010101100..00
Denormal Result	0	-126	0.0010101100..00

The Intel 64 and IA-32 architectures deal with denormal values in the following ways:

- It avoids creating denormals by normalizing numbers whenever possible.
- It provides the floating-point underflow exception to permit programmers to detect cases when denormals are created.
- It provides the floating-point denormal-operand exception to permit procedures or programs to detect when denormals are being used as source operands for computations.



Einschub:

Gleitkommazahlendarstellung nach IEEE 754

The screenshot shows the Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. The page displays the table of contents for Chapter 10, focusing on the MXCSR Control and Status Register. The section 10.2.3 is highlighted. The screenshot also includes the Adobe Acrobat toolbar at the top.

10.2.3 MXCSR Control and Status Register

The 32-bit MXCSR register (see Figure 10-3) contains control and status information for SSE, SSE2, and SSE3 SIMD floating-point operations. This register contains:

- flag and mask bits for SIMD floating-point exceptions
- rounding control field for SIMD floating-point operations

Vol. 1 10-3

PROGRAMMING WITH INTEL® STREAMING SIMD EXTENSIONS (INTEL® SSE)

• flush-to-zero flag that provides a means of controlling underflow conditions on SIMD floating-point operations

• denormals-are-zeros flag that controls how SIMD floating-point instructions handle denormal source operands

The contents of this register can be loaded from memory with the LDMXCSR and FXRSTOR instructions and stored in memory with STMXCSR and FXSAVE.

Bits 16 through 31 of the MXCSR register are reserved and are cleared on a power-up or reset of the processor; attempting to write a non-zero value to these bits, using either the FXRSTOR or LDMXCSR instructions, will result in a general-protection exception (#GP) being generated.

* The denormals-are-zeros flag was introduced in the Pentium 4 and Intel Xeon processor.

Figure 10-3. MXCSR Control/Status Register

enabled, the processor performs the following operations when it detects a floating-point underflow condition.

10-4 Vol. 1

PROGRAMMING WITH INTEL® STREAMING SIMD EXTENSIONS (INTEL® SSE)

- Returns a zero result with the sign of the true result.
- Sets the precision and underflow exception flags.

IEEE-Gleitkommaarithmetik

Rechnen mit Gleitkommazahlen

- Exponenten und Mantissen müssen getrennt berechnet werden
=> Gleitkomma-Arithmetik ist komplexer als Festkomma-Arithmetik

Verallgemeinertes Vorgehen: Addition / Subtraktion von 2 Gleitkommazahlen

- Unterschiedliche Exponenten?
=> Angleich des kleineren Exponenten an den Größeren
=> Verschiebung der Mantisse des kleineren Exponenten nach rechts
- Addition / Subtraktion der Mantissen
- Bestimmung Vorzeichen:
$$Vz := (|Exp_{op1}| > |Exp_{op2}|) ? Vz_{op1} : Vz_{op2}$$
- Normalisierung des Ergebnisses (wenn notwendig)

Verallgemeinertes Vorgehen: Multiplikation/ Division von 2 Gleitkommazahlen

- Multipliziere/ Dividiere die Mantissen
- Bestimme das Vorzeichen:

$$Vz := Vz_{op1} \oplus Vz_{op2}$$

- Addiere/ Subtrahiere die Exponenten,
und subtrahiere/ addiere 127_{10} (Bias)



IEEE-Gleitkommaarithmetik

Rechnen mit Gleitkommazahlen

Addition

Beispiel: Addition von 2 Gleitkommazahlen

- $56_{10} = 32 + 16 + 8 = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 = 111000_2 = 1,11000_2 \times 2^5$
 $56_{10} = +1,11000_2 \times 2^5;$

$$5 := e - 127 \Rightarrow e = 132 : \mathbf{1000\ 0100}$$

$\Rightarrow 0\ \mathbf{1000\ 0100\ 110...0}$ (IEEE einfache Genauigkeit)

- $0,75_{10} = 0,11_2 = 1,1_2 \times 2^{-1}$
 $-1 := e - 127 \Rightarrow e = 126 : \mathbf{0111\ 1110}$
 $\Rightarrow 0\ \mathbf{0111\ 1110\ 10...0}$ (IEEE einfache Genauigkeit)

- Unterschiedliche Exponenten?

$$\cdot 56_{10} = +1,11000_2 \times 2^5$$

$$\cdot 0,75_{10} = 1,1_2 \times 2^{-1} = 0,0000011_2 \times 2^5$$

- Addition der Mantissen

$$\begin{array}{r} 1,110000_2 \\ + 0,0000011_2 \\ \hline 1,1100011_2 \end{array}$$

- Bestimmung Vorzeichen:

$$Vz := (|Exp_{op1}| > |Exp_{op2}|)? Vz_{op1}: Vz_{op2}$$

$$Vz := (|5| > |-1|)? (+): (+):= +$$

Binär	Unsigned	Excess-127
0000 0000	0	-127
...
0111 1110	126	-1
...
1000 0100	132	+5



IEEE-Gleitkommaarithmetik

Rechnen mit Gleitkommazahlen

Beispiel: Subtraktion von 2 Gleitkommazahlen

- $56_{10} = +1,11000_2 \times 2^5;$
 $\Rightarrow 0\ 1000\ 0100\ 110...0$ (IEEE einfache Genauigkeit)

- $0,75_{10} = 0,11_2 = 1,1_2 \times 2^{-1}$
 $\Rightarrow 0\ 0111\ 1110\ 10...0$ (IEEE einfache Genauigkeit)

- Subtraktion der Mantissen

$$1,1100000_2$$

$$0,0000011_2$$

$$\begin{array}{r} & 11111 \\ - & 11111 \\ \hline & 1,1011101_2 \end{array}$$

- Bestimmung Vorzeichen:

$$Vz := (|Exp_{op1}| > |Exp_{op2}|) ? Vz_{op1} : Vz_{op2}$$

$$Vz := (|5| > |-1|) ? (+) : (+) := +$$

Subtraktion

Binär	Unsigned	Excess-127
0000 0000	0	-127
...
0111 1110	126	-1
...
1000 0100	132	+5



IEEE-Gleitkommaarithmetik

Rechnen mit Gleitkommazahlen

Subtraktion

Beispiel: Subtraktion von 2 Gleitkommazahlen

- $56_{10} = +1,11000_2 \times 2^5;$
 $5 := e - 127 \Rightarrow e = 132 : 1000\ 0100$
 $\Rightarrow 0\ 1000\ 0100\ 110...0$ (IEEE einfache Genauigkeit)
- $0,75_{10} = 0,11_2 = 1,1_2 \times 2^{-1}$
 $-1 := e - 127 \Rightarrow e = 126 : 0111\ 1110$
 $\Rightarrow 0\ 0111\ 1110\ 10...0$ (IEEE einfache Genauigkeit)
- Subtraktion der Mantissen

$$1,110000_2$$

$$0,0000011_2$$

$$\begin{array}{r} & 11111 \\ - & 11111 \\ \hline & 1,1011101_2 \end{array}$$

- Bestimmung Vorzeichen: +
- $0\ 1000\ 0100\ 10111010..0$ (IEEE einfache Genauigkeit)
- Ergebnis:

$$Z_{10} = 56_{10} - 0,75_{10} = +1,1011101_2 \times 2^5 = +110111,01_2 = 32+16+4+2+1+0,25 = 55,25_{10}$$

Binär	Unsigned	Excess-127
0000 0000	0	-127
...
0111 1110	126	-1
...
1000 0100	132	+5



IEEE-Gleitkommaarithmetik

Rechnen mit Gleitkommazahlen

Multiplikation

Beispiel: Multiplikation von 2 Gleitkommazahlen

- $56_{10} = +1,11000_2 \times 2^5;$
 $5 := e - 127 \Rightarrow e = 132 : 1000\ 0100$
 $\Rightarrow 0\ 1000\ 0100\ 110...0$ (IEEE einfache Genauigkeit)

- $0,75_{10} = 0,11_2 = 1,1_2 \times 2^{-1}$
 $-1 := e - 127 \Rightarrow e = 126 : 0111\ 1110$
 $\Rightarrow 0\ 0111\ 1110\ 10...0$ (IEEE einfache Genauigkeit)

- Multipliziere die Mantissen

$$\begin{aligned} 1,11_2 \times 1,1_2 &= (1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}) \times (1 \times 2^0 + 1 \times 2^{-1}) = \\ &\quad (1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}) \\ &\quad \quad (1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \end{aligned}$$

$$\begin{array}{r} + 1 \quad \quad 1 \quad \quad 1 \\ \hline (1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}) = 10,101_2 \end{array}$$

$$1,11_2 \times 1,1_2$$

$$1,11$$

$$0,111$$

$$\begin{array}{r} + 1\ 1\ 1 \\ \hline 10,101 \end{array}$$

Binär	Unsigned	Excess-127
0000 0000	0	-127
...
0111 1110	126	-1
...
1000 0011	131	+4
1000 0100	132	+5



IEEE-Gleitkommaarithmetik

Rechnen mit Gleitkommazahlen

Multiplikation

Beispiel: Multiplikation von 2 Gleitkommazahlen

- $56_{10} = +1,11000_2 \times 2^5;$
 $5 := e - 127 \Rightarrow e = 132 : 1000\ 0100$
 $\Rightarrow 0\ 1000\ 0100\ 110...0$ (IEEE einfache Genauigkeit)
- $0,75_{10} = 0,11_2 = 1,1_2 \times 2^{-1}$
 $-1 := e - 127 \Rightarrow e = 126 : 0111\ 1110$
 $\Rightarrow 0\ 0111\ 1110\ 10...0$ (IEEE einfache Genauigkeit)
- Multipliziere die Mantissen
 - $1,11_2 \times 1,1_2 = 10,101_2$
- Bestimme das Vorzeichen:

$$Vz := Vz_{op1} \oplus Vz_{op2} := (+) \oplus (+) := (+) = 0$$

- Addiere die Exponenten, und subtrahiere 127_{10} (Bias) (2-facher Bias, deshalb einmal abziehen)

$$\begin{array}{r} 1000\ 0100 \\ + 0111\ 1110 \\ \hline 1\ 1111\ 1 \\ \hline 1\ 0000\ 0010 \\ - 0111\ 1111 \\ \hline 1\ 1111\ 111 \\ \hline 1000\ 0011 \end{array}$$

Binär	Unsigned	Excess-127
0000 0000	0	-127
...
0111 1110	126	-1
...
1000 0011	131	+4
1000 0100	132	+5



IEEE-Gleitkommaarithmetik

Rechnen mit Gleitkommazahlen

Multiplikation

Beispiel: Multiplikation von 2 Gleitkommazahlen

- $56_{10} = +1,11000_2 \times 2^5;$
 $5 := e - 127 \Rightarrow e = 132 : 1000\ 0100$
 $\Rightarrow 0\ 1000\ 0100\ 110...0$ (IEEE einfache Genauigkeit)

- $0,75_{10} = 0,11_2 = 1,1_2 \times 2^{-1}$
 $-1 := e - 127 \Rightarrow e = 126 : 0111\ 1110$
 $\Rightarrow 0\ 0111\ 1110\ 10...0$ (IEEE einfache Genauigkeit)

- Multipliziere die Mantissen
 - $1,11_2 \times 1,1_2 = 10,101_2$
- Bestimme das Vorzeichen:

$$Vz := Vz_{Op1} \oplus Vz_{Op2} := (+) \oplus (+) := (+) = 0$$

- Addiere die Exponenten, und subtrahiere 127_{10} (Bias) (2-facher Bias, deshalb einmal abziehen)
 - Exponent: $1000\ 0011 = 2^4$
 $\Rightarrow +10,101_2 \times 2^4 = +1,0101_2 \times 2^5$

⇒ Normalisierung:

$$\begin{aligned} &\Rightarrow +1,0101_2 \\ &\Rightarrow 2^5 = 1000\ 0100 \end{aligned} \quad \left. \right\} 0\ 1000\ 0100\ 01010...0 = +1,0101_2 \times 2^5 = +101010_2 = 2^5 + 2^3 + 2^1 = 42_{10}$$

Binär	Unsigned	Excess-127
0000 0000	0	-127
...
0111 1110	126	-1
...
1000 0011	131	+4
1000 0100	132	+5

IEEE-Gleitkommaarithmetik

Rechnen mit Gleitkommazahlen

Division

Beispiel: Division von 2 Gleitkommazahlen

- $56_{10} = +1,11000_2 \times 2^5;$
 $5 := e - 127 \Rightarrow e = 132 : 1000\ 0100$
 $\Rightarrow 0\ 1000\ 0100\ 110...0$ (IEEE einfache Genauigkeit)
- $14_{10} = 8 + 4 + 2 = 1110_2 = 1,110_2 \times 2^3$
 $3 := e - 127 \Rightarrow e = 130 : 1000\ 0010$
 $\Rightarrow 0\ 1000\ 0010\ 110...0$ (IEEE einfache Genauigkeit)
- Dividiere die Mantissen
 - $1,11_2 \div 1,11_2 = 1,0_2$
- Bestimme das Vorzeichen:

$$Vz := Vz_{Op1} \oplus Vz_{Op2} := (+) \oplus (+) := (+) = 0$$

- Subtrahiere die Exponenten, und addiere 127_{10} (Bias)

$$\begin{array}{r} 1000\ 0100 \\ - 1000\ 0010 \\ \hline 1 \\ \hline 0000\ 0010 \\ + 0111\ 1111 \\ \hline 1111\ 11 \\ \hline 1000\ 0001 \end{array}$$

Binär	Unsigned	Excess-127
0000 0000	0	-127
...
0111 1110	126	-1
...
1000 0001	129	+2
1000 0010	130	+3
1000 0011	131	+4
1000 0100	132	+5



IEEE-Gleitkommaarithmetik

Rechnen mit Gleitkommazahlen

Division

Beispiel: Division von 2 Gleitkommazahlen

- $56_{10} = +1,11000_2 \times 2^5;$
 $5 := e - 127 \Rightarrow e = 132 : \mathbf{1000\ 0100}$
 $\Rightarrow 0\ \mathbf{1000\ 0100\ 110...0}$ (IEEE einfache Genauigkeit)
- $14_{10} = 8 + 4 + 2 = 1110_2 = 1,110_2 \times 2^3$
 $3 := e - 127 \Rightarrow e = 130 : \mathbf{1000\ 0010}$
 $\Rightarrow 0\ \mathbf{1000\ 0010\ 110...0}$ (IEEE einfache Genauigkeit)
- Dividiere die Mantissen
 - $1,11_2 \div 1,11_2 = 1,0_2$
- Bestimme das Vorzeichen:

$$Vz := Vz_{Op1} \oplus Vz_{Op2} := (+) \oplus (+) := (+) = 0$$

- Subtrahiere die Exponenten, und addiere 127_{10} (Bias)

Exponent: $\mathbf{1000\ 0001} = 2^{+2}$

$$\Rightarrow +1,0_2 \times 2^{+2}$$

$$0\ \mathbf{1000\ 0001\ 0...0} = +1,0_2 \times 2^2 = +100_2 = 2^2 = 4_{10}$$

$$56_{10} : 14_{10} = 4_{10}$$

Binär	Unsigned	Excess-127
0000 0000	0	-127
...
0111 1110	126	-1
...
1000 0001	129	+2
1000 0010	130	+3
1000 0011	131	+4
1000 0100	132	+5



IEEE-Gleitkommaarithmetik

Darstellbare Zahlenbereiche

Um verschiedene Gleitkommadarstellung bzgl. Ihrer Rechengenauigkeit miteinander zu vergleichen, definiert man **drei charakteristische Zahlen**:

Binär	Unsigned	Excess-127
0000 0000	0	-127
...
0110 1000	104	-23
...
0111 1110	126	-1
0111 1111	127	0
1000 0000	128	+1

- **maxreal:** größte darstellbare normalisierte positive Zahl

$$0\ 1111\ 1110\ \textcolor{red}{11111\ 11111\ 11111\ 11111\ 111} = +1 \times 2^{127} \times (2 - 2^{-23}) = 2^{128} - 2^{104} \approx 2^{128}$$

- **minreal:** kleinste darstellbare normalisierte positive Zahl

$$0\ 0000\ 0001\ \textcolor{red}{00000\ 00000\ 00000\ 00000\ 000} = +1,0 \times 2^{-126} = 2^{-126}$$

- **smallreal:** kleinste Zahl, die man zu 1 addieren kann, um einen Wert $\neq 1$ zu erhalten

- $1 = +(1,0 \times 2^0)$

$$0 := e - 127 \Rightarrow e = 127 : \textcolor{blue}{0111\ 1111}$$

$$0\ \textcolor{red}{0111\ 1111\ 00000\ 00000\ 00000\ 00000\ 000}$$

- $2^{-23} \approx 0,000000119$

$$= +0,\textcolor{red}{00000\ 00000\ 00000\ 00000\ 001}_2 = +1,\textcolor{red}{00000\ 00000\ 00000\ 00000\ 000}_2 \times 2^{-23}$$

$$0\ \textcolor{red}{0110\ 1000\ 00000\ 00000\ 00000\ 00000\ 000}$$

- Addition:

$$\begin{array}{r} 0\ 0111\ 1111\ 00000\ 00000\ 00000\ 00000\ 000 \\ + 0\ 0110\ 1000\ 00000\ 00000\ 00000\ 00000\ 000 \\ \hline \end{array}$$

- Exponent angeleichen
Mantisse
rechtsverschieben:
23 bit

$$\begin{array}{r} 0\ 0111\ 1111\ 00000\ 00000\ 00000\ 00000\ 000 \\ + 0\ 0111\ 1111\ 00000\ 00000\ 00000\ 00000\ 001 \\ \hline 0\ 0111\ 1111\ 00000\ 00000\ 00000\ 00000\ 001 \end{array}$$

- Ergebnis: $0\ 0111\ 1111\ 00000\ 00000\ 00000\ 00000\ 001 = +1,\textcolor{red}{00000\ 00000\ 00000\ 00000\ 001}_2 \times 2^0;$

$$\Rightarrow \text{smallreal} = 2^{-23} = 0\ 0110\ 1000\ 00000\ 00000\ 00000\ 000$$



IEEE-Gleitkommaarithmetik

Darstellbare Zahlenbereiche

Die Gesetzmäßigkeiten, die für reelle Zahlen gelten, werden für Maschinendarstellungen verletzt!

- Die gilt insbesondere auch, wenn diese Zahlen in einer höheren Programmiersprache oft „real“ heißen.
- Das Assoziativgesetz $(x+y) + z = x + (y + z)$ gilt selbst dann nicht unbedingt, wenn keine overflow oder underflow auftritt:
 - $x = 1; \quad y = \text{smallreal} / 2; \quad z = \text{smallreal} / 2$
 - $(x+y) + z = (1+\text{smallreal}/2) + \text{smallreal}/2$
= $1+\text{smallreal}/2$
= 1
 - $x + (y + z) = 1 + (\text{smallreal}/2 + \text{smallreal}/2)$
= 1 + smallreal
!= 1



IEEE-Gleitkommaarithmetik

Darstellbare Zahlenbereiche

Darstellungsgenauigkeit am Beispiel

Binär	Unsigned	Excess-127
0000 0000	0	-127
...
1000 0000	128	+1
1000 0011	131	+4
1001 1010	154	+27

- $217.063.424,0_{10} = 0\ 1001\ 1010\ \textcolor{red}{10011}\ 11000\ 00010\ 00000\ 000$

Nächste benachbarte darstellbare Zahl: Abstand = 16

Dazwischen sind
keine Zahlen
darstellbar!

- $217.063.440,0_{10} = 0\ 1001\ 1010\ \textcolor{red}{10011}\ 11000\ 00010\ 00000\ 001$

Frage:

Was wenn wir rechnen: $217.063.424,0_{10} + 1,0_{10} = ?$ → Das Ergebnis kann nur falsch sein

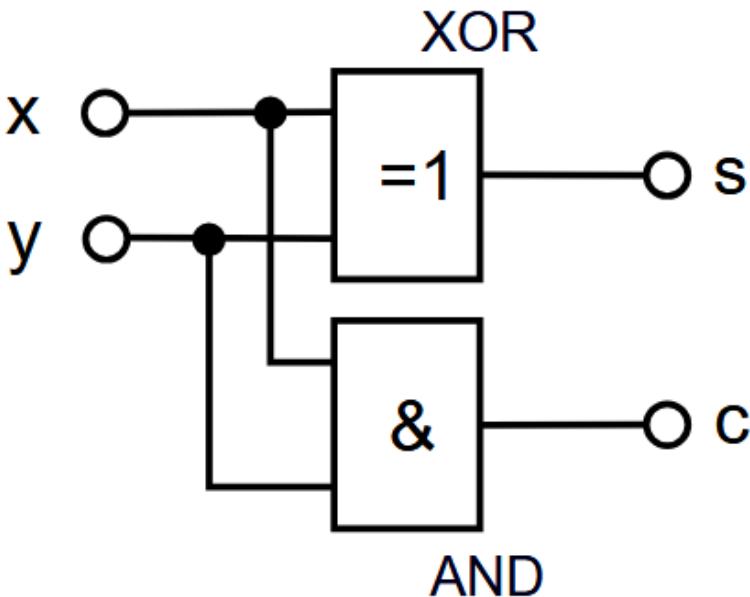


Grundtypen von Addierern: Halbaddierer (HA)

- zwei Eingänge: x, y
- zwei Ausgänge: s, c
- $s = x \oplus y$ (Summe)
- $c = x \wedge y$ (Carry/ Übertrag)

Funktionsstabelle

x	y	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Bemerkung: Kann nur für das LSB von mehrstelligen Zahlen benutzt werden.

<https://de.wikipedia.org/wiki/Halbaddierer>

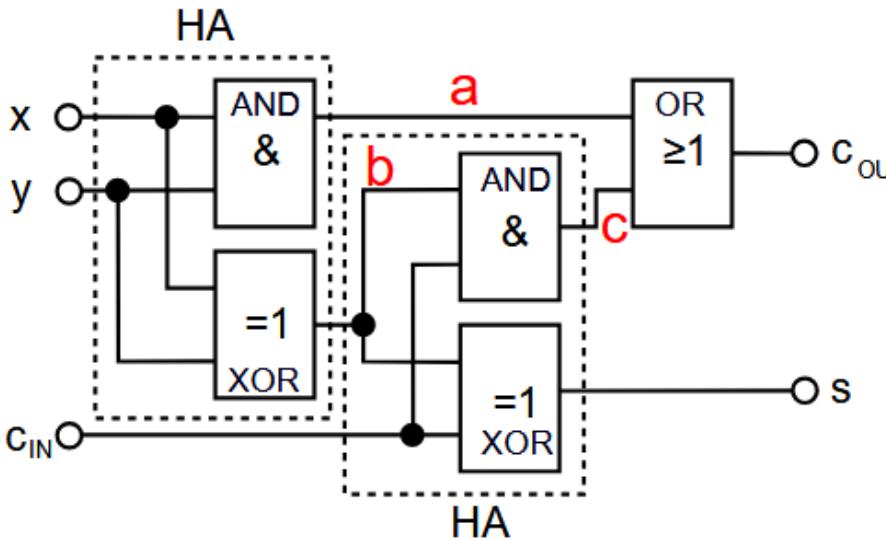


Grundtypen von Addierern: Volladdierer (HA)

- drei Eingänge: x, y, c_{IN}
- zwei Ausgänge: s, c_{OUT}
- $s = (x \oplus y) \oplus c_{IN}$ (Summe)
- $c_{OUT} = (x \wedge y) \vee ((x \oplus y) \wedge c_{IN})$ (Carry/ Übertrag)

Funktionstabelle

x	y	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



$$b = x \oplus y$$

$$c = b \wedge c_{IN}$$

$$c = (x \oplus y) \wedge c_{IN}$$

$$a = x \wedge y$$

$$c_{OUT} = a \vee c$$

$$c_{OUT} = (x \wedge y) \vee ((x \oplus y) \wedge c_{IN})$$

$$s = b \oplus c_{IN} = (x \oplus y) \oplus c_{IN}$$

<https://de.wikipedia.org/wiki/Volladdierer>



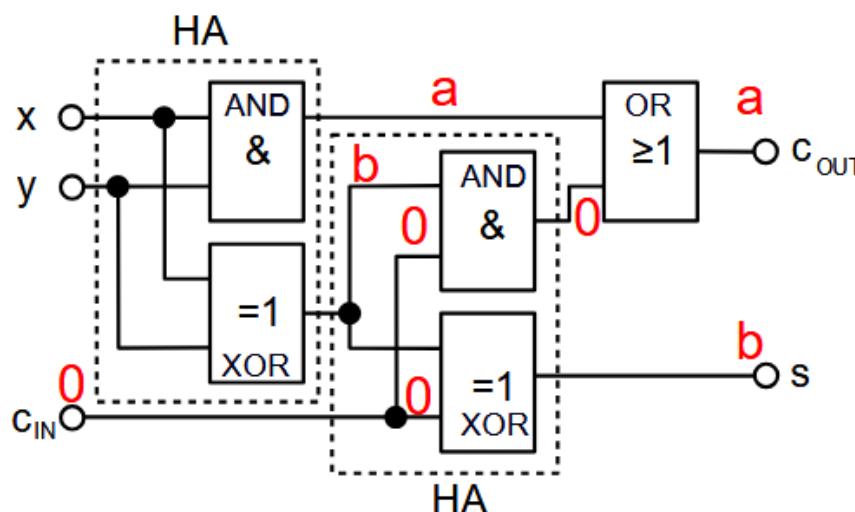
Vergleich Volladdierer (HA) mit Carry-IN =0 und Halbaddierer sind identisch

Volladdierer ($c_{IN} = 0$)

- drei Eingänge: x, y, c_{IN}
- zwei Ausgänge: s, c_{OUT}
- $s = (x \oplus y) \oplus 0 = x \oplus y$
- $c_{OUT} = (x \wedge y) \vee ((x \oplus y) \wedge 0) = x \wedge y$

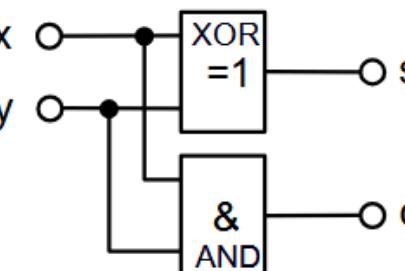
Halbaddierer:

- zwei Eingänge: x, y
- zwei Ausgänge: s, c
- $s = x \oplus y$ (Summe)
- $c = x \wedge y$ (Carry/ Übertrag)



Funktionstabelle ($c_{IN} = 0$)

x	y	c_{IN}	s	c_{OUT}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1



Funktionstabelle

x	y	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Grundtypen von Addierern: Mehrstellige Addition (Ripple Carry)

In ALUs werden i.d.R. zwei z.B. 16- oder 32 bit Zahlen verknüpft:

Carry-ripple Addierer (CRA):

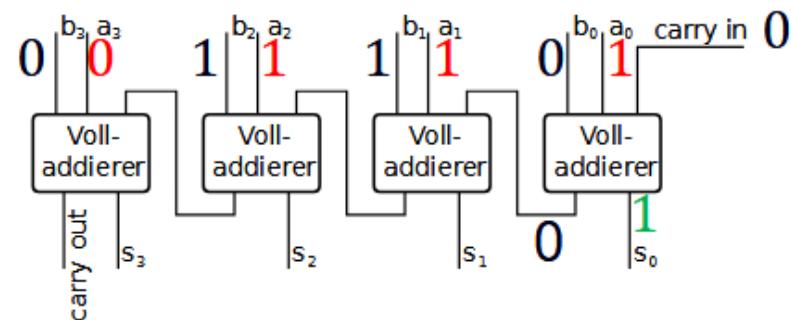
- Einfache Möglichkeit, zwei mehrstellige Dualzahlen zu addieren
- Je Stelle einen Volladdierer
- Beim LSB reicht ein Halbaddierer.

In der Praxis: Volladdierer mit $c_{IN} = 0$

Bsp:

Addition von 2 4-bit unsigned integer:

- $7+6=13$
- $a_{10} = 7_{10} = 0111_2 = a_3a_2a_1a_0_2$
- $b_{10} = 6_{10} = 0110_2 = b_3b_2b_1b_0_2$
- $s_{10} = 13_{10} = 1101_2 = b_3b_2b_1b_0_2$



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
$a_0 + b_0$		1 0 0 1 0		
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

<https://de.wikipedia.org/wiki/Carry-Ripple-Addierer>



Grundtypen von Addierern:

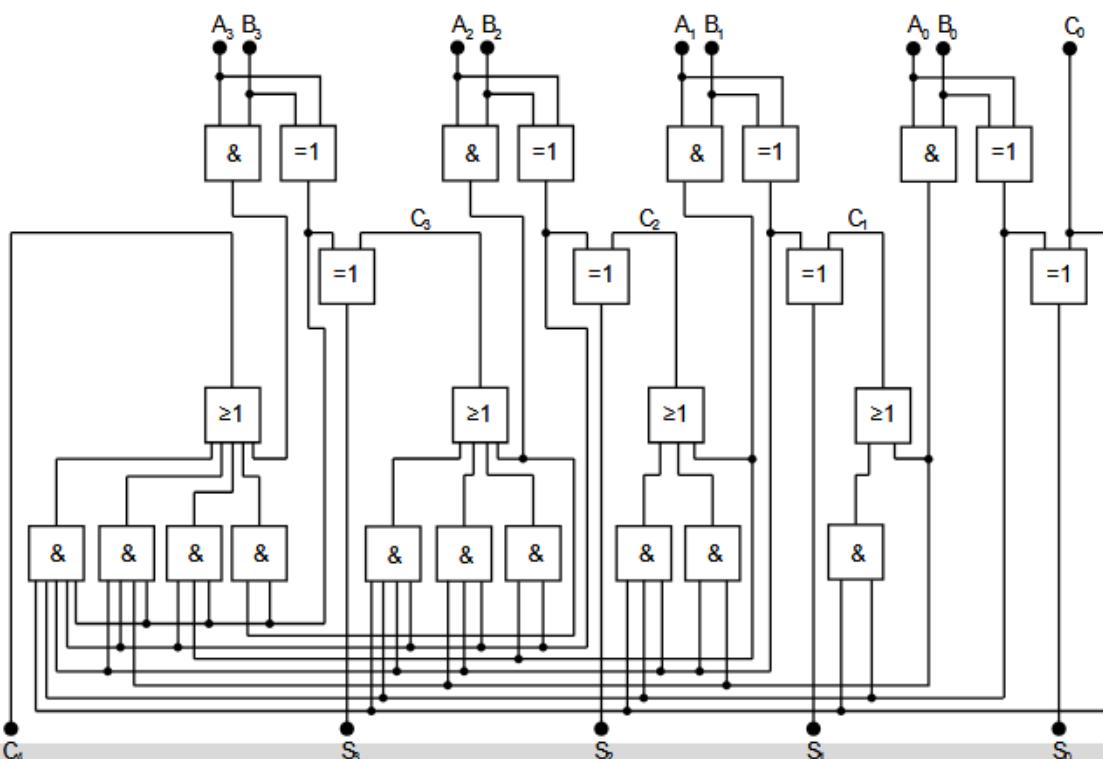
Mehrstellige Addition mit Carry Lookahead Addierer (CLA)

- Alle c_i und s_i können direkt nur aus den Eingangsvariablen a_i und b_i berechnet werden.
- Alle c_i und s_i werden parallel im ersten Takt berechnet, durch parallele Übertragungslogik.
- Schneller als RCA
- Größe des Hardware Aufwands steigt mit steigender Stellenzahl stark an.
Lsg: Kleine CLAs seriell kaskadiert.

Bsp:

Addition von 2 4-bit unsigned integer:

- $7+6=13$
- $a_{10} = 7_{10} = 0111_2 = a_3a_2a_1a_0_2$
- $b_{10} = 6_{10} = 0110_2 = b_3b_2b_1b_0_2$
- $s_{10} = 13_{10} = 1101_2 = s_3s_2s_1s_0_2$



https://de.wikipedia.org/wiki/Paralleladdierer_mit_%C3%9Cbertragsvorausberechnung



Grundtypen von Addierern:

Mehrstellige Addition mit Carry Lookahead Addierer (CLA)

Beispiel

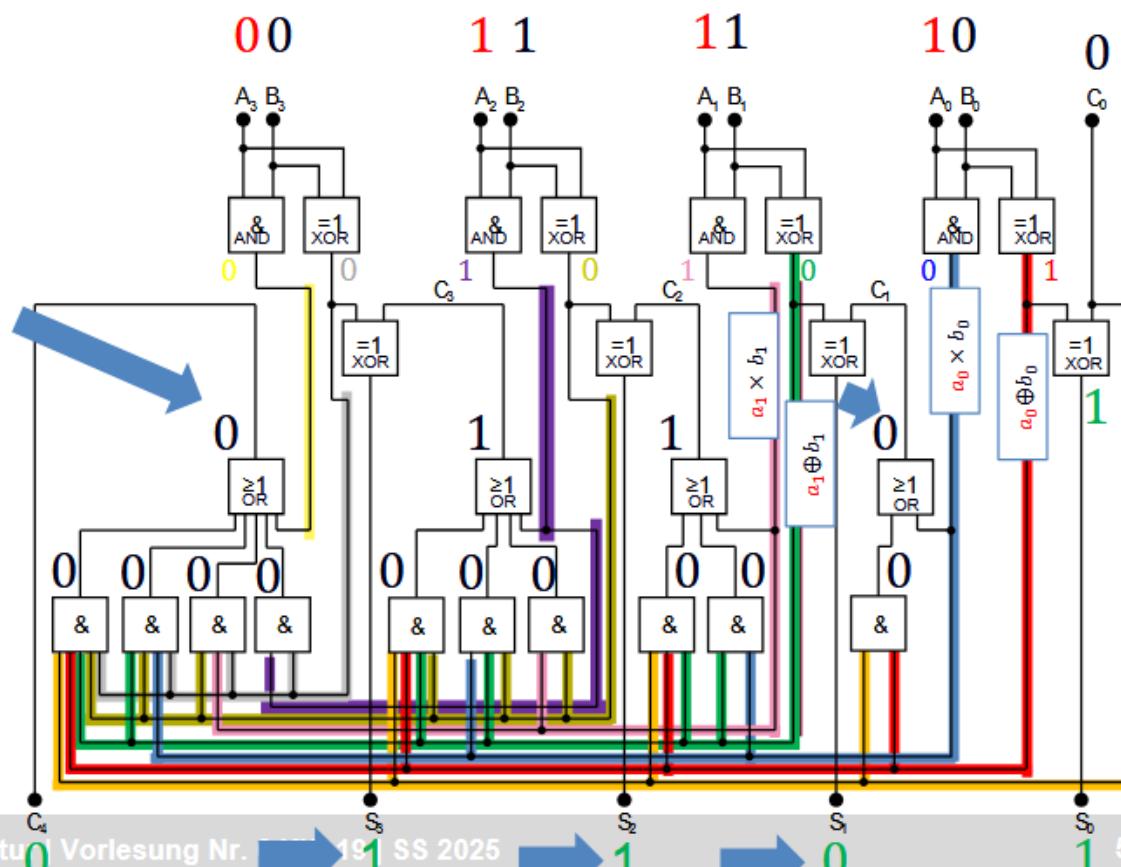
Schritt für Schritt by INSPECTION

(IV)

Bsp:

Addition von 2 4-bit unsigned integer:

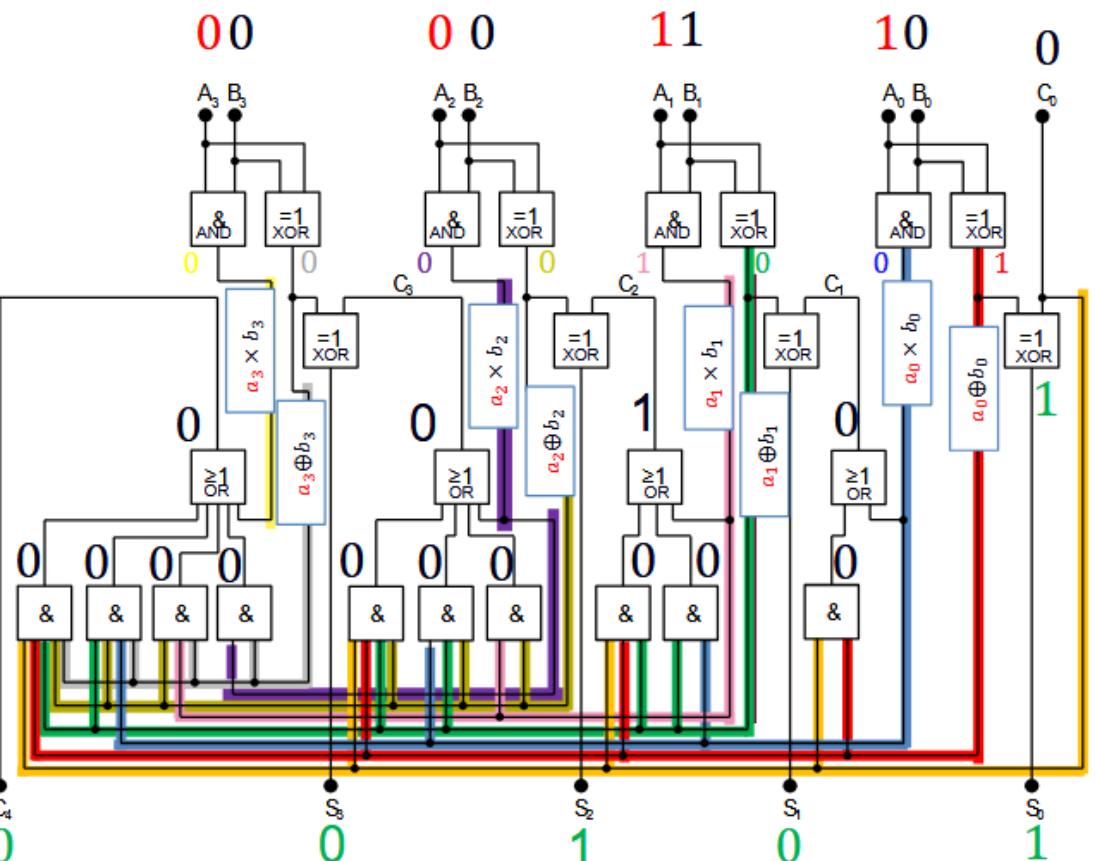
- $7+6=13$
- $a_{10} = 7_{10} = 0111_2 = a_3a_2a_1a_0_2$
- $b_{10} = 6_{10} = 0110_2 = b_3b_2b_1b_0_2$
- $s_{10} = 13_{10} = 1101_2 = s_3s_2s_1s_0_2$



Addition und Subtraktion in einer Schaltung

im 2er Komplement

Betrachtung nur des CLA



1) Beispiel: Addition

$$s_{10} = a_{10} + b_{10} = 3 + 2 = 5$$

$$1. \quad a_{10} = +3_{10} = 0011_2 = a_3 a_2 a_1 a_0_2$$

$$2. \quad b_{10} = +2_{10} = 0010_2 = b_3 b_2 b_1 b_0_2$$

$$3. \quad s_{10} = +5_{10} = 0101_2 = s_3 s_2 s_1 s_0_2$$

Steuersignal S:

- $S=0 \Rightarrow a + b$

Betrachtung CLA

- $a_{10} = 0011_2 = a_3 a_2 a_1 a_0_2$
- $b_{10} = 0010_2 = b_3 b_2 b_1 b_0_2$
- $c_0 = 0$
- $s_0 = 1$
- $c_1 = 0$
- $s_1 = 0$
- $c_2 = 1$
- $s_2 = 1$
- $c_3 = 0$
- $s_3 = 0$
- $c_4 = 0$

Betrachtung Überlauf

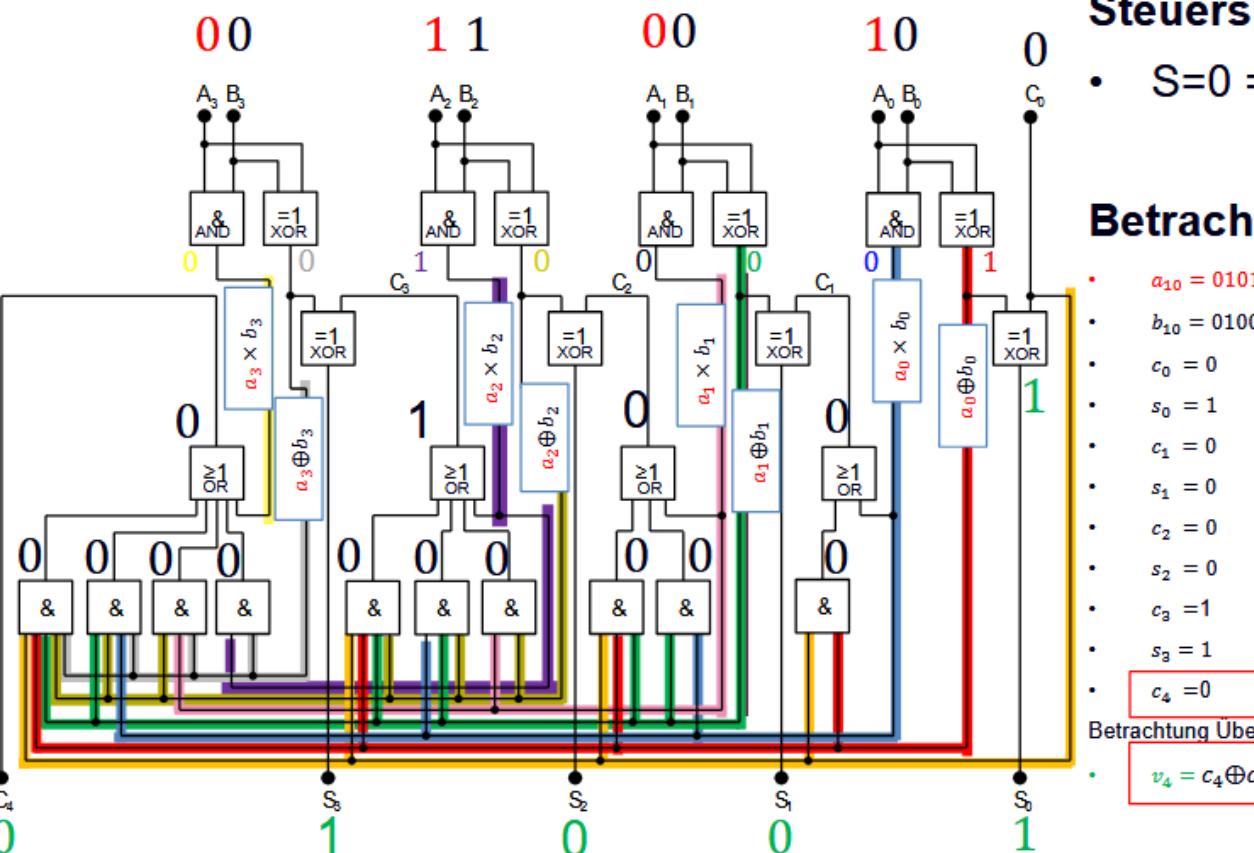
$$v_4 = c_4 \oplus c_3 = 0 \oplus 0 = 0 \Rightarrow \text{kein Überlauf}$$

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

Addition und Subtraktion in einer Schaltung

im 2er Komplement

Betrachtung nur des CLA



2) Beispiel: Addition

$$s_{10} = a_{10} + b_{10} = 5 + 4 = 9$$

$$1. \quad a_{10} = +5_{10} = 0101_2 = a_3 a_2 a_1 a_0_2$$

$$2. \quad b_{10} = +4_{10} = 0100_2 = b_3 b_2 b_1 b_0_2$$

$$3. \quad s_{10} = -7_{10} = 1001_2 = b_3 b_2 b_1 b_0_2 = ?$$



Steuersignal S:

- $S=0 \Rightarrow a + b$

Betrachtung CLA

- $a_{10} = 0101_2 = a_3 a_2 a_1 a_0_2$
- $b_{10} = 0100_2 = b_3 b_2 b_1 b_0_2$
- $c_0 = 0$
- $s_0 = 1$
- $c_1 = 0$
- $s_1 = 0$
- $c_2 = 0$
- $s_2 = 0$
- $c_3 = 1$
- $s_3 = 1$
- $c_4 = 0$

Betrachtung Überlauf

$$v_4 = c_4 \oplus c_3 = 0 \oplus 1 = 1 \Rightarrow \text{Überlauf}$$

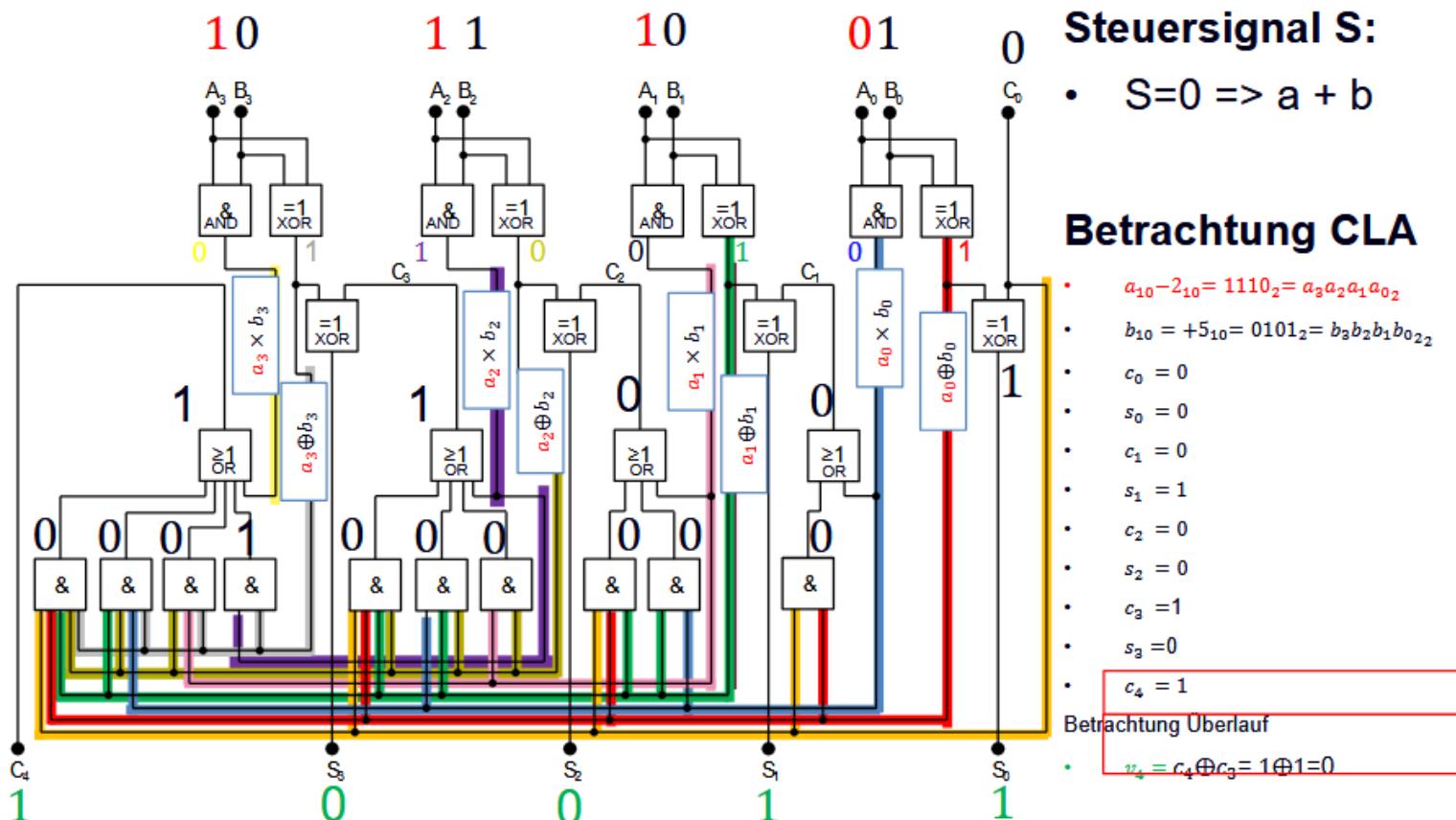
Binär	2er Komp.
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1



Addition und Subtraktion in einer Schaltung

im 2er Komplement

Betrachtung nur des CLA



3) Beispiel: Addition

$$s_{10} = a_{10} + b_{10} = -2 + 5 = +3$$

$$1. \quad a_{10} = -2_{10} = 1110_2 = a_3 a_2 a_1 a_0_2$$

$$2. \quad b_{10} = +5_{10} = 0101_2 = b_3 b_2 b_1 b_0_2$$

$$3. \quad s_{10} = +3_{10} = 0011_2 = s_3 s_2 s_1 s_0_2$$

Steuersignal S:

- S=0 => a + b

Betrachtung CLA

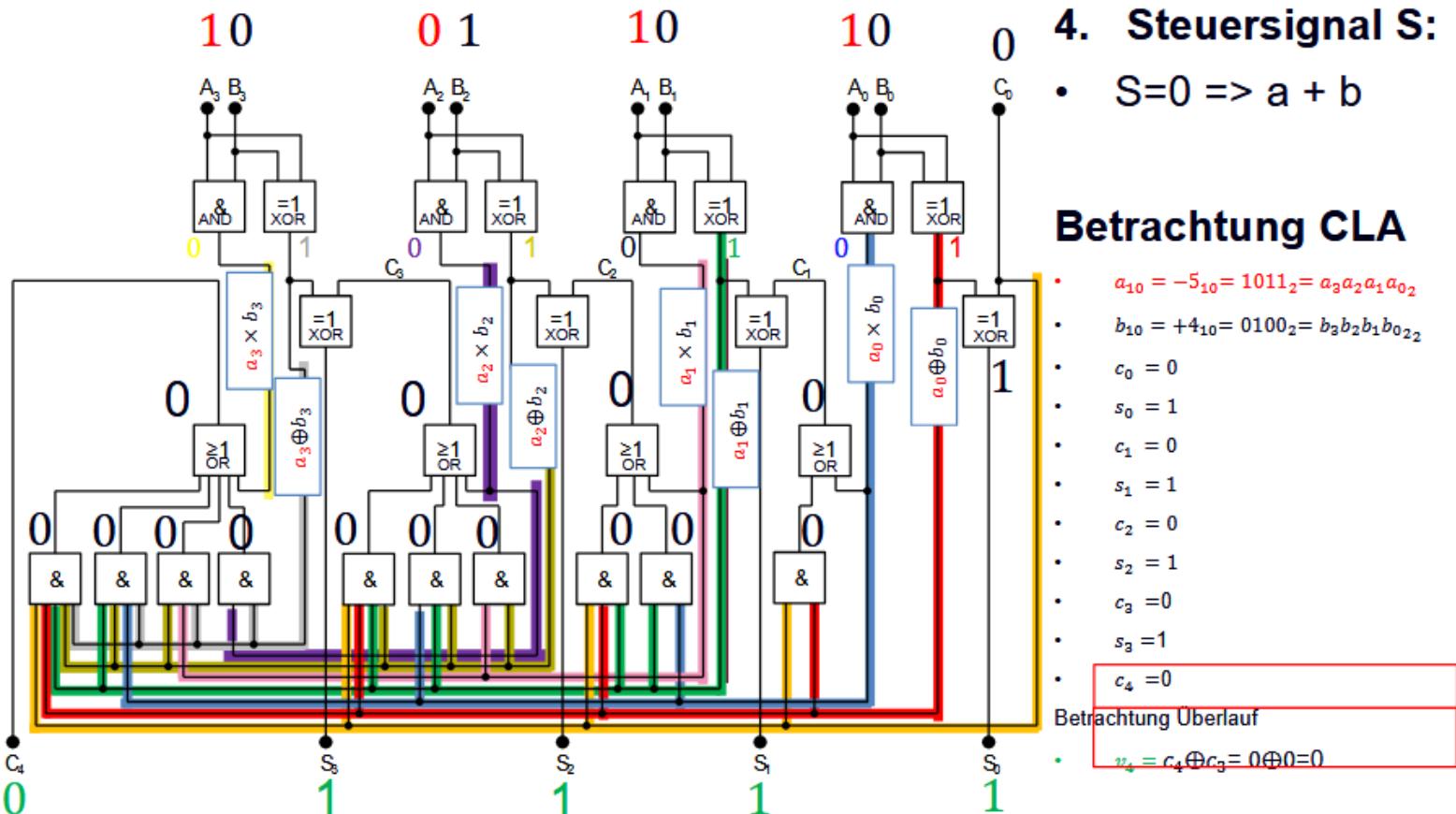
- $a_{10} - 2_{10} = 1110_2 = a_3 a_2 a_1 a_0_2$
 - $b_{10} = +5_{10} = 0101_2 = b_3 b_2 b_1 b_0_2$
 - $c_0 = 0$
 - $s_0 = 0$
 - $c_1 = 0$
 - $s_1 = 1$
 - $c_2 = 0$
 - $s_2 = 0$
 - $c_3 = 1$
 - $s_3 = 0$
 - $c_4 = 1$
- Betrachtung Überlauf
- $v_4 = c_4 \oplus c_3 = 1 \oplus 1 = 0$

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



Addition und Subtraktion in einer Schaltung im 2er Komplement

Betrachtung nur des CLA

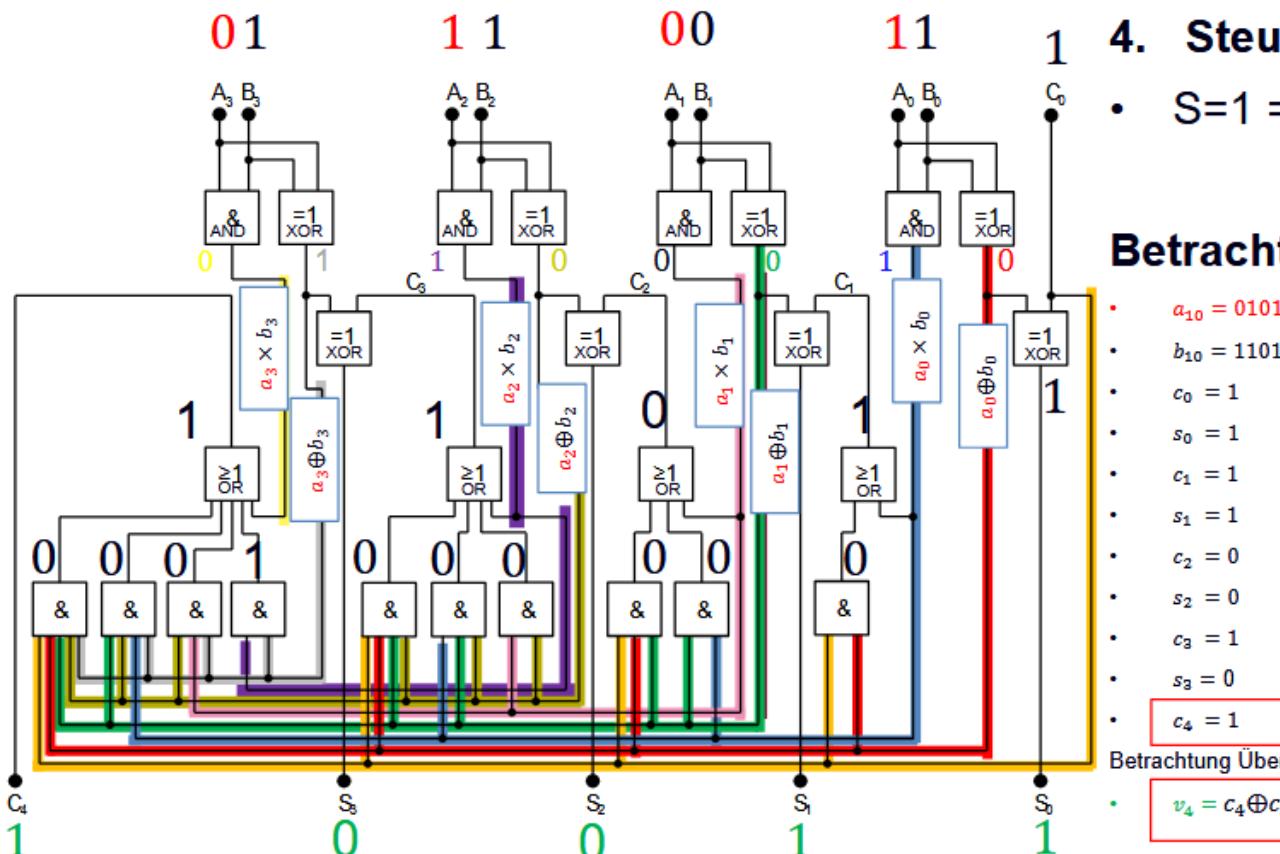


Binär	Zer. Kompl.
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Addition und Subtraktion in einer Schaltung

im 2er Komplement

Betrachtung nur des CLA



5) Beispiel: Subtraktion

$$s_{10} = a_{10} - b_{10} = 5 - 2 = 3$$

$$1. \quad a_{10} = +5_{10} = 0101_2 = a_3 a_2 a_1 a_0_2$$

$$2. \quad b_{10} = +2_{10} = 0010_2 = b_3 b_2 b_1 b_0_2$$

$$3. \quad s_{10} = 3_{10} = 0011_2 = s_3 s_2 s_1 s_0_2$$

4. Steuersignal S:

- $S=1 \Rightarrow a - b$

Betrachtung CLA

- $a_{10} = 0101_2 = a_3 a_2 a_1 a_0_2$
- $b_{10} = 1101_2$ (Invertierung da $S=1$)
- $c_0 = 1$
- $s_0 = 1$
- $c_1 = 1$
- $s_1 = 1$
- $c_2 = 0$
- $s_2 = 0$
- $c_3 = 1$
- $s_3 = 0$
- $c_4 = 1$

Betrachtung Überlauf

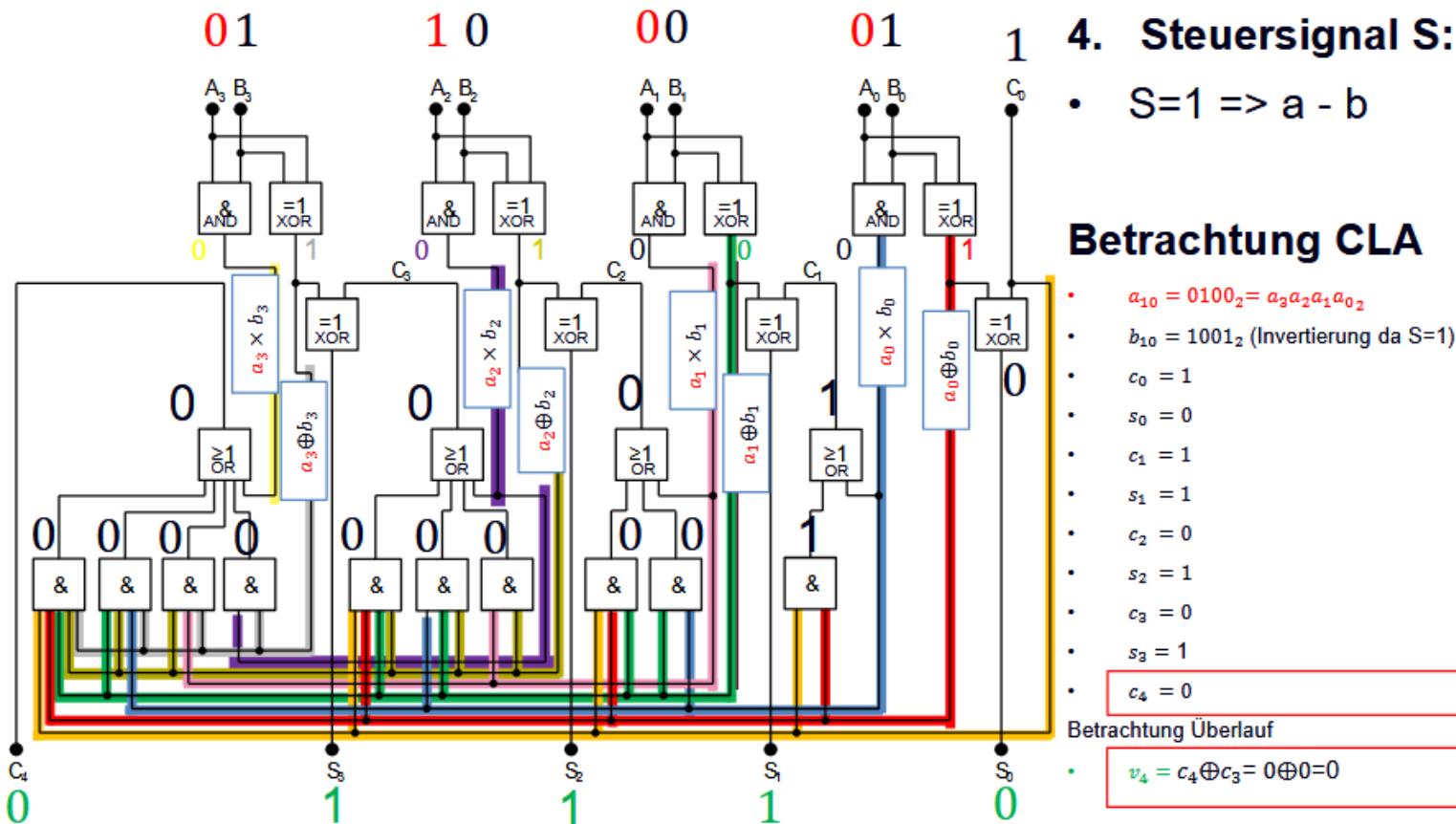
$$v_4 = c_4 \oplus c_3 = 1 \oplus 1 = 0$$

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

Addition und Subtraktion in einer Schaltung

im 2er Komplement

Betrachtung nur des CLA



6) Beispiel: Subtraktion

$$s_{10} = a_{10} - b_{10} = 4 - 6 = -2$$

1. $a_{10} = +4_{10} = 0100_2 = a_3 a_2 a_1 a_0_2$
2. $b_{10} = +6_{10} = 0110_2 = b_3 b_2 b_1 b_0_2$
3. $s_{10} = -2_{10} = 1110_2 = s_3 s_2 s_1 s_0_2$

4. Steuersignal S:

- $S=1 \Rightarrow a - b$

Betrachtung CLA

- $a_{10} = 0100_2 = a_3 a_2 a_1 a_0_2$
- $b_{10} = 1001_2$ (Invertierung da S=1)
- $c_0 = 1$
- $s_0 = 0$
- $c_1 = 1$
- $s_1 = 1$
- $c_2 = 0$
- $s_2 = 1$
- $c_3 = 0$
- $s_3 = 1$
- $c_4 = 0$

Betrachtung Überlauf

$$v_4 = c_4 \oplus c_3 = 0 \oplus 0 = 0$$

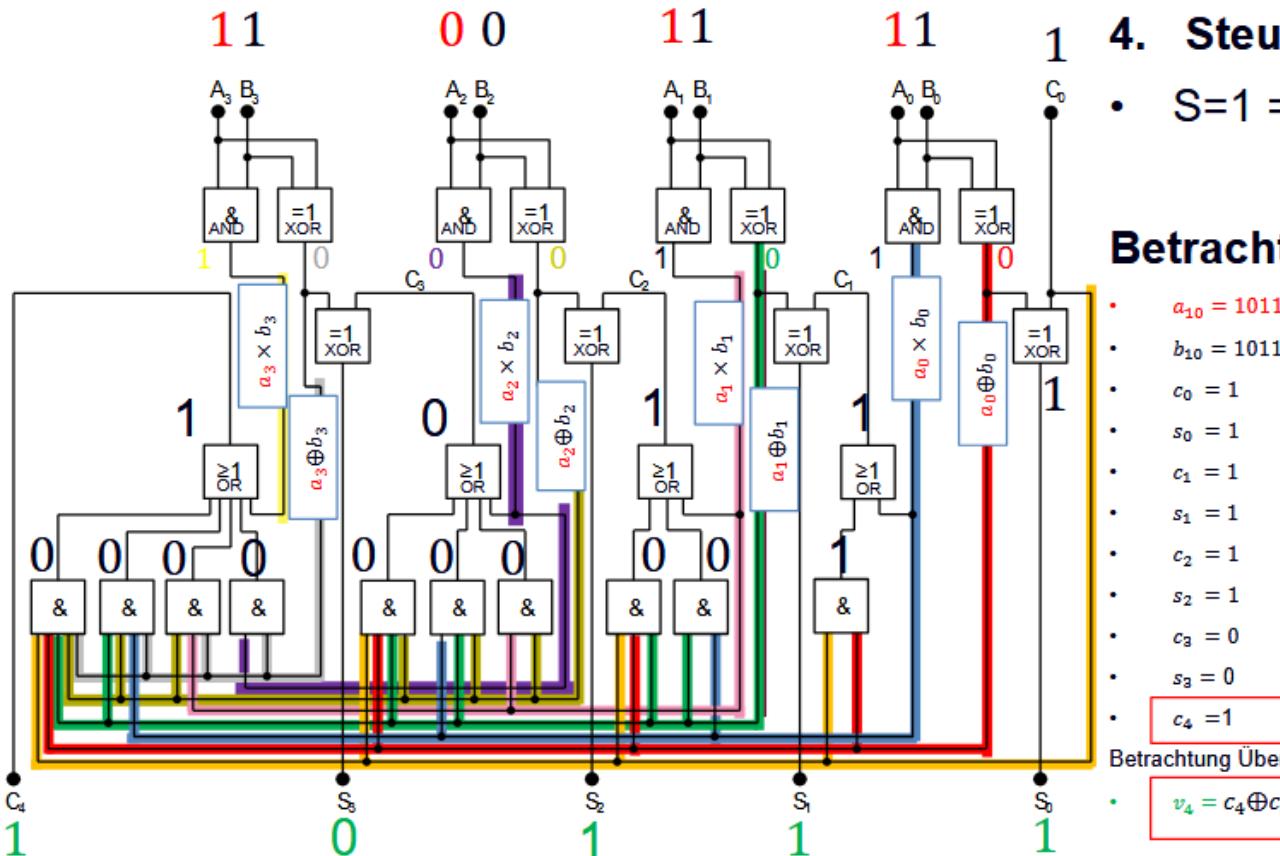
Binär	2er Komp.
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1



Addition und Subtraktion in einer Schaltung

im 2er Komplement

Betrachtung nur des CLA



7) Beispiel: Subtraktion

$$s_{10} = a_{10} - b_{10} = -5 - 4 = -9$$

1. $a_{10} = -5_{10} = 1011_2 = a_3 a_2 a_1 a_0_2$

2. $b_{10} = +4_{10} = 0100_2 = b_3 b_2 b_1 b_0_2$

3. $s_{10} = +7_{10} = 0111_2 = s_3 s_2 s_1 s_0_2$



4. Steuersignal S:

- $S=1 \Rightarrow a - b$

Betrachtung CLA

- $a_{10} = 1011_2 = a_3 a_2 a_1 a_0_2$
- $b_{10} = 1011_2$ (Invertierung da $S=1$)
- $c_0 = 1$
- $s_0 = 1$
- $c_1 = 1$
- $s_1 = 1$
- $c_2 = 1$
- $s_2 = 1$
- $c_3 = 0$
- $s_3 = 0$
- $c_4 = 1$

Betrachtung Überlauf

$v_4 = c_4 \oplus c_3 = 1 \oplus 0 = 1$

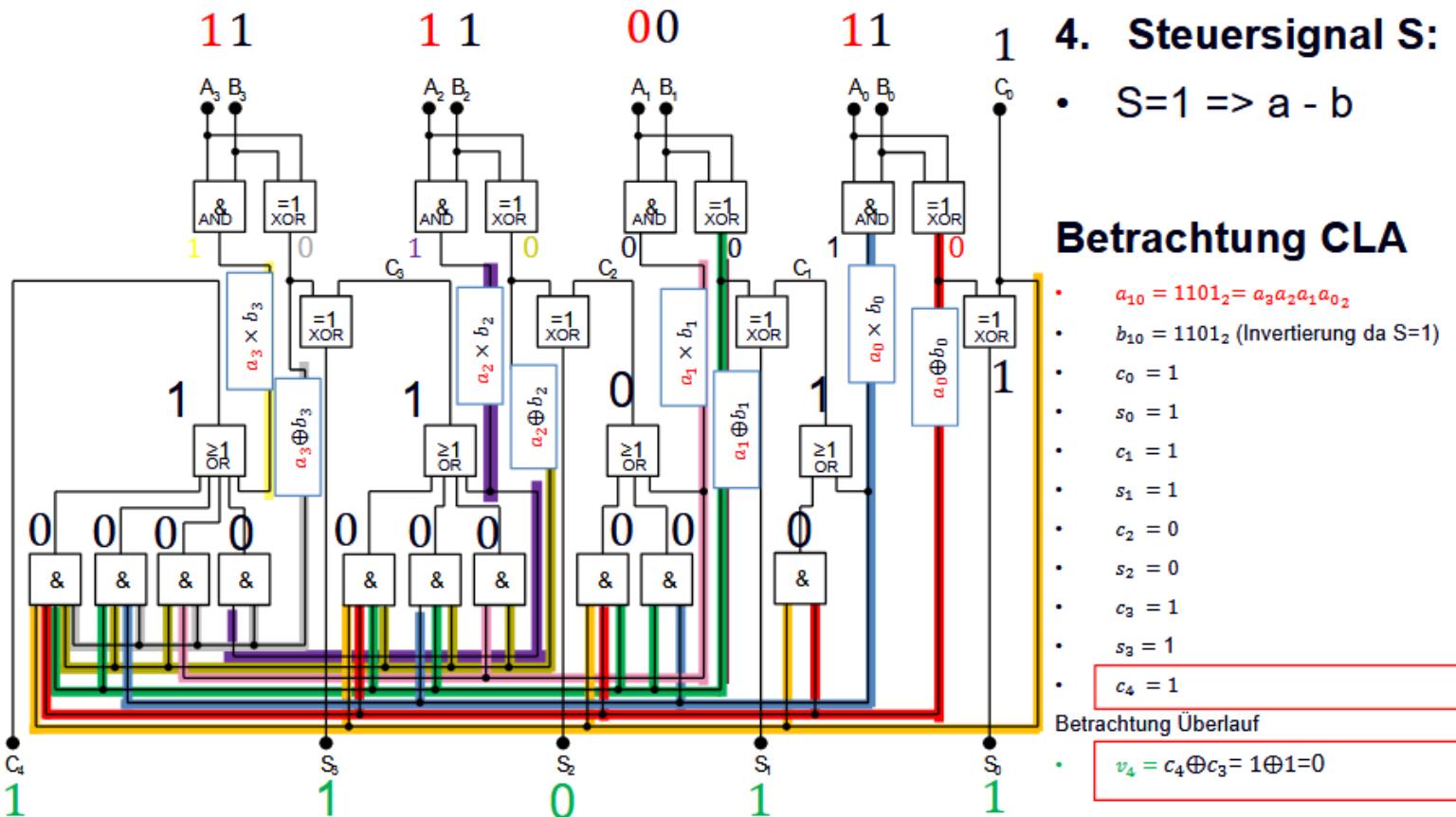
Binär	2er Komp.
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1



Addition und Subtraktion in einer Schaltung

im 2er Komplement

Betrachtung nur des CLA



8) Beispiel: Subtraktion

$$s_{10} = a_{10} - b_{10} = -3 - 2 = -5$$

1. $a_{10} = -3_{10} = 1101_2 = a_3 a_2 a_1 a_0_2$
2. $b_{10} = +2_{10} = 0010_2 = b_3 b_2 b_1 b_0_2$
3. $s_{10} = -5_{10} = 1011_2 = s_3 s_2 s_1 s_0_2$

4. Steuersignal S:

- $S=1 \Rightarrow a - b$

Betrachtung CLA

- $a_{10} = 1101_2 = a_3 a_2 a_1 a_0_2$
 - $b_{10} = 1101_2$ (Invertierung da $S=1$)
 - $c_0 = 1$
 - $s_0 = 1$
 - $c_1 = 1$
 - $s_1 = 1$
 - $c_2 = 0$
 - $s_2 = 0$
 - $c_3 = 1$
 - $s_3 = 1$
 - $c_4 = 1$
- Betrachtung Überlauf
- $v_4 = c_4 \oplus c_3 = 1 \oplus 1 = 0$

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



Addition und Subtraktion in einer Schaltung

im 2er Komplement

Schlussfolgerung (Regel):

Nr	Operation	Typ	?	VZ(a)	VZ(b)	VZ(s)	$c_{n-1} = c_3$	$c_n = c_4$	$v = c_n \oplus c_{n-1} = c_4 \oplus c_3$
1	$s_{10} = a_{10} + b_{10} = 3 + 2 = 5$	Addition	OK	+	+	+	0	0	0
3	$s_{10} = a_{10} + b_{10} = -2 + 5 = +3$	Addition	OK	-	+	+	1	1	0
4	$s_{10} = a_{10} + b_{10} = -5 + 4 = -1$	Addition	OK	-	+	-	0	0	0
5	$s_{10} = a_{10} - b_{10} = 5 - 2 = 3$	Subtraktion	OK	+	+	+	1	1	0
6	$s_{10} = a_{10} - b_{10} = 4 - 6 = -2$	Subtraktion	OK	+	+	-	0	0	0
8	$s_{10} = a_{10} - b_{10} = -3 - 2 = -5$	Subtraktion	OK	-	+	-	1	1	0
2	$s_{10} = a_{10} + b_{10} = 5 + 4 = -7$	Addition	Fehler	+	+	-	1	0	1
7	$s_{10} = a_{10} - b_{10} = -5 - 4 = +7$	Subtraktion	Fehler	-	+	+	0	1	1

Carry-bits
(Übertrag)
C
V

Wenn Carry-bits c_n und c_{n-1} gleich sind, dh:
Overflow $v = c_n \oplus c_{n-1} = 0$

=> Dann ist die Rechenoperation zu einem korrekten Ergebnis gekommen

Wenn Carry-bits c_n und c_{n-1} ungleich sind, dh:
Overflow $v = c_n \oplus c_{n-1} = 1$

=> Dann ist die Rechenoperation zu einem falschen Ergebnis gekommen



Multiplikation mit Festkomma-Dualzahlen

Prinzip der mehrfachen Addition:

Multiplikation von (Festkomma-)Dualzahlen ist analog zu Multiplikation im Dezimalsystem

Produkt = Multiplikand (MD) x Multiplikator (MR)

$$123 \times 456$$

$$49200 \quad (123 \times 400)$$

$$6150 \quad (123 \times 50)$$

$$+ \quad 738 \quad (123 \times 6)$$

$$11000 \quad (\text{Carry - bits})$$

$$56088$$

$$1101 \times 1001$$

$$1101000 \quad (1101 \times 1000)$$

$$000000 \quad (1101 \times 000)$$

$$00000 \quad (1101 \times 00)$$

$$+ \quad 1101 \quad (1101 \times 1)$$

$$0010000 \quad (\text{Carry - bits})$$

$$1110101$$

$$1101_2 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13_{10}$$

$$1001_2 = 2^3 + 2^0 = 8 + 1 = 9_{10}$$

$$13_{10} \times 9_{10} = 117_{10}$$

$$1110101_2 = 2^6 + 2^5 + 2^4 + 2^2 + 2^0 = 64 + 32 + 16 + 4 + 1 = 117_{10}$$



Multiplikation mit Festkomma-Dualzahlen

erster Algorithmus

Prinzip der mehrfachen Addition:

Multiplikation von (Festkomma-)Dualzahlen ist analog zu Multiplikation im Dezimalsystem

Produkt = Multiplikand (MD) x Multiplikator (MR)

Ebenso möglich:

$$\begin{array}{r} 123 \times 456 \\ 738 \quad (123 \times 6) \\ 6150 \quad (123 \times 50) \\ + \quad 49200 \quad (123 \times 400) \\ \hline 11000 \quad (\text{Carry - bits}) \\ \hline 56088 \end{array}$$

$$\begin{array}{r} 1101 \times 1001 \\ 1101 \quad (1101 \times 1) \\ \underline{0000} \quad (1101 \times 0) \\ 0000 \quad (1101 \times 0) \\ + \quad 1101 \quad (1101 \times 1) \\ \hline \underline{0010000} \quad (\text{Carry - bits}) \\ \hline 1110101 \end{array}$$

$$1101_2 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13_{10}$$

$$1001_2 = 2^3 + 2^0 = 8 + 1 = 9_{10}$$

$$13_{10} \times 9_{10} = 117_{10}$$

$$1110101_2 = 2^6 + 2^5 + 2^4 + 2^2 + 2^0 = 64 + 32 + 16 + 4 + = 117_{10}$$



Multiplikation mit Festkomma-Dualzahlen

erster Algorithmus

1) Mögliches Prinzip zur Umsetzung

1. Ergebnis := 0000 0000

2. MD := 1101

3. MR := 1001

4. M := LSB von MR

5. Wenn M = 1 dann:

Ergebnis := Ergebnis + MD

6. MD nach links verschieben

7. MR rechts verschieben

8. Wenn MR = 0 dann EXIT

9. Gehe zu 4.

Produkt = (MD) x (MR)

• Ergebnis = 0000 0000 (Teilergebnis)

• MD = 0000 1101

• MR = 0000 1001

• M = 1

• 0000 0000

+ 0000 1101

• Ergebnis = 0000 1101



• MD = 0001 1010

• MR = 0000 0100

$$1101 \times 1001$$

$$\begin{array}{r} 0000 \\ 1101 \end{array}$$

$$\begin{array}{r} 0000 \\ 0000 \end{array}$$

$$\begin{array}{r} 0000 \\ + 1101 \end{array}$$

$$\begin{array}{r} 0000 \\ + 1101 \\ \hline 1101000 \end{array}$$

Ergebnis: 0000 1101

=> keine Veränderung

=> keine Veränderung

+ 0110 1000

Ergebnis: 0111 0101



Multiplikation mit Festkomma-Dualzahlen

erster Algorithmus

1) Mögliches Prinzip zur Umsetzung

Produkt = $(MD) \times (MR)$

- Ergebnis = 0000 1101 (Teilergebnis)
- MD = 0001 1010
- MR = 0000 0100
- M = 0

4. M := LSB von MR

5. Wenn M = 1 dann:

Ergebnis := Ergebnis + MD

6. MD nach links verschieben

- MD = 0011 0100

7. MR rechts verschieben

- MR = 0000 0010

8. Wenn MR = 0 dann EXIT

9. Gehe zu 4.



Multiplikation mit Festkomma-Dualzahlen

erster Algorithmus

1) Mögliches Prinzip zur Umsetzung

4. M := LSB von MR

5. Wenn M = 1 dann:

Ergebnis := Ergebnis + MD

6. MD nach links verschieben

7. MR rechts verschieben

8. Wenn MR = 0 dann EXIT

9. Gehe zu 4.

Produkt = (MD) x (MR)

• Ergebnis = 0000 1101 (Teilergebnis)

• MD = 0110 1000

• MR = 0000 0001

• M = 1

0000 1101

+ 0110 1000

0001 0000 (Carry-bit)

• Ergebnis = 0111 0101

• MD = 1101 0000

• MR = 0000 0000 => EXIT

=> Ergebnis: 0111 0101



Multiplikation mit Festkomma-Dualzahlen

erster Algorithmus

Produkt = Multiplikand (MD) x Multiplikator (MR)

Beobachtung:

- **Prinzip der mehrfachen Addition**
- Shift- , ADD- und Vergleich-Funktionen reichen aus:
 - Addieren wenn LSB MR = 1 (Teilprodukt berechnen)
 - MD nach links verschieben
 - MR rechts verschieben
 - Wenn MR = 0 dann EXIT
- Höherer physikalischer Aufwand als bei Addition
 - Doppelt langes Ergebnisregister
 - 4 bit Zahl + 4 bit Zahl = 4 (bzw. 5) bit Zahl
 - 4 bit Zahl * 4 bit Zahl = 8 bit Zahl
 - Neben ADD auch Shift-Operationen zu implementieren

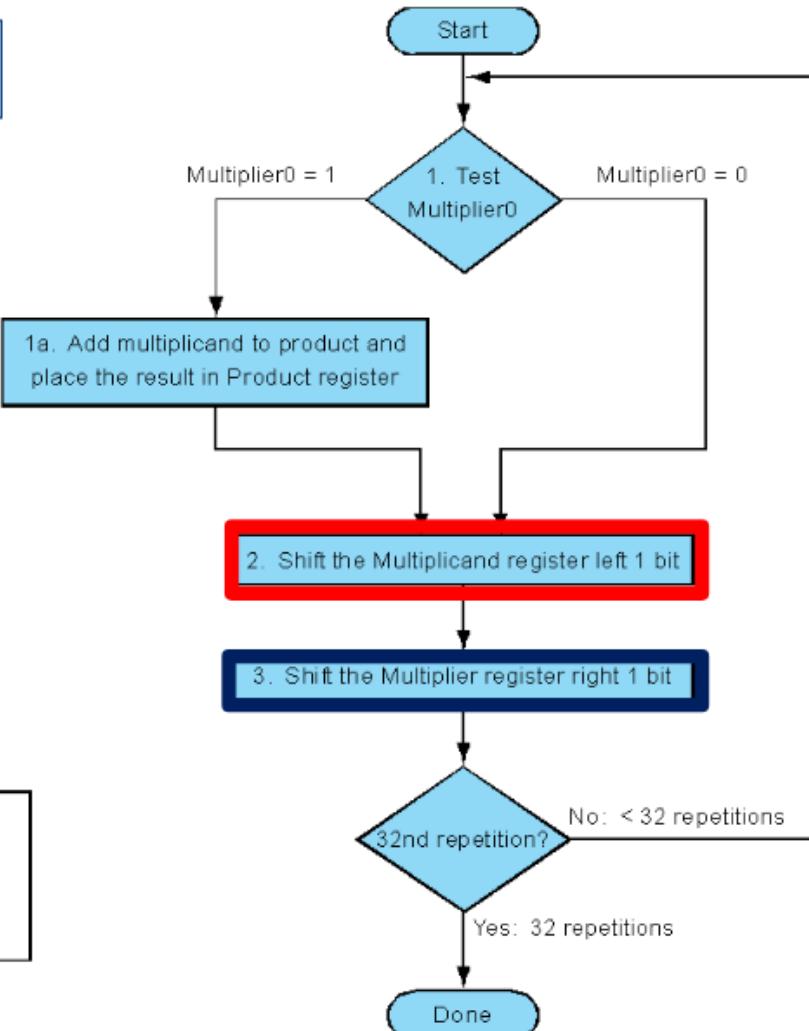
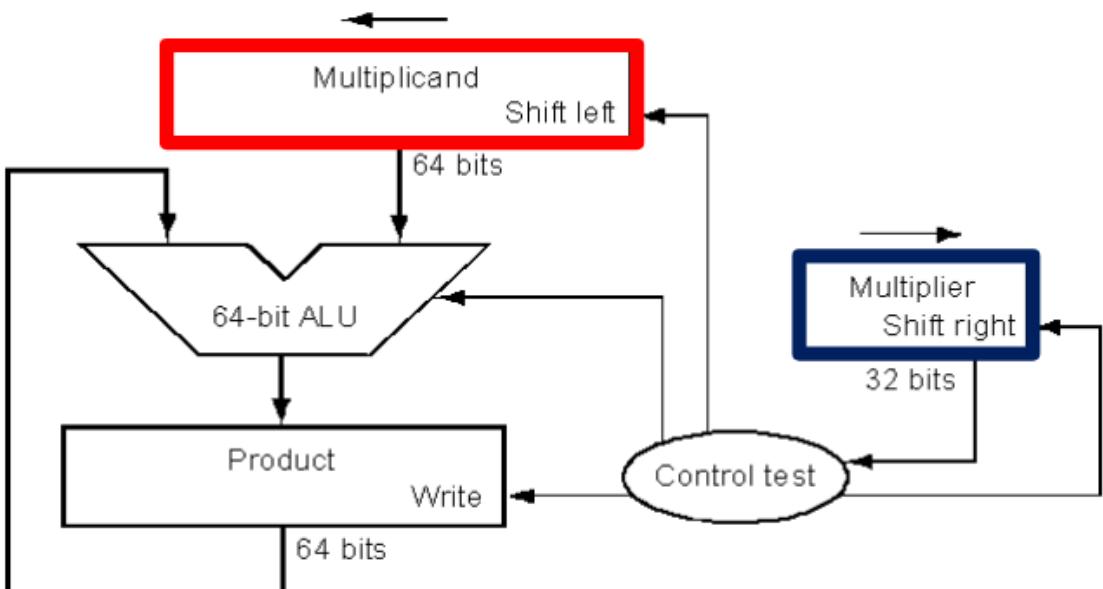


Multiplikation mit Festkomma-Dualzahlen

Zusammenfassung: erster Algorithmus

$$\text{Produkt} = \boxed{\text{Multiplikand (MD)}} \times \boxed{\text{Multiplikator (MR)}}$$

- Addieren wenn LSB MR = 1 (Teilprodukt berechnen)
- MD nach links verschieben
- MR rechts verschieben
- Hier 64 bit, dh:
 - $\text{MD} < 2^{32}$
 - $\text{MR} < 2^{32}$
 - dh: die bits 63-32 sind 0, nur die bits 31-0 sind belegbar.



Multiplikation mit Festkomma-Dualzahlen

Tabellen

Schnelle Multiplikationen mit Tabellen-Rechenwerken:

- Produkte aus zwei n-stelligen Faktoren werden in ROM gespeichert
- Statt Multiplikation nur Auslesen der Tabellen
- Realisierung mit Look Up Table

Bsp:

- $A = 2_{10} = 10_2$
- $B = 3_{10} = 11_2$
- $\Rightarrow C = A \times B = 2_{10} \times 3_{10}$
 $= 6_{10} = 0110_2$

A	B	P = A x B
00	00	0000
00	01	0000
00	10	0000
00	11	0000
01	00	0000
01	01	0001
01	10	0010
01	11	0011
10	00	0000
10	01	0010
10	10	0100
10	11	0110
11	00	0000
11	01	0011
11	10	0110
11	11	1001

Beobachtung:

- schnell durch lookup
- Speicherbedarf steigt exponentiell an
- A und B zusammen 4 bit $\Rightarrow 2^4 = 16$ Adressen notwendig
- A und B jeweils 32 bit $\Rightarrow 64$ bit $\Rightarrow 2^{64} = 18.446.744.073.709.551.616$ Adressen notwendig



Multiplikation mit Festkomma-Dualzahlen

Beobachtungen

- **Multiplikation zweier Zahlen in Vorzeichen Betrags-Darstellung, ist einfach:**
 - Beträge miteinander multipliziert (wie bei positiven Zahlen)
 - $VZ_{Ergebnis} = VZ_{Faktor\ 1} \oplus VZ_{Faktor\ 2}$
- **Multiplikation zweier Zahlen in 2er-Komplement Darstellung, geht nicht direkt..**

Lösungen:

1. Umwandeln:
 - 2er-Komplementzahlen in Betrag und Vorzeichen umwandeln.
 - Zahlen miteinander multiplizieren
 - Ergebnis in 2er-Komplementdarstellung umwandeln
2. Spezielle Multiplikations-Algorithmen
 - Booth Algorithmus



Booth's Algorithmus

Anforderung:

Multiplikation von Binärzahlen jeglichen Vorzeichens mit einem Verfahren

Problem:

- Multiplikation $(-3) \times (-7) == +21$
 - 4-bit 2erKomplement: $1101_2 \times 1001_2$
 - Unsigned Int 4-bit: $13 \times 9 == 117$
- } 2er Komplement
nicht direkt
multiplizieren!

Lösung: Booth!

Mittels Booth's Algorithm können Zahlen im 2er-Komplement multipliziert werden.



Einschränkung:

MD, MR und Produkt müssen alle mit der gleichen Anzahl an bits dargestellt werden.



Schnellere Multiplikation durch Subtraktion

Grundlage für Booth

Grundlage:

Jede binäre Zahl kann durch Summe und die Differenz anderer binärer Zahlen dargestellt werden

$$a \times b = 0111_2 \times bbbb_2 = \begin{array}{r} bbbb_2 \\ bbbb_2 \\ + 0000_2 \\ \hline \end{array} = \begin{array}{r} bbbb_2 \\ bbbb0_2 \\ + 0000000_2 \\ \hline \end{array}$$

$$\begin{aligned} a \times b &= 0111_2 \times bbbb_2 = (1000_2 - 0001_2) \times bbbb_2 = -bbbb_2 + bbbb000_2 \\ &= -bbbb_2 + bbbb000_2 = \begin{array}{r} bbbb000_2 \\ - bbbb_2 \\ \hline \end{array} \end{aligned}$$

=> Subtraktion => weniger Additionen => schnellere/einfachere Multiplikation



Multiplikation

Abschließende Beobachtungen

- **Multiplikation zweier Zahlen in Vorzeichen Betrags-Darstellung**

Beträge miteinander multipliziert (wie bei positiven Zahlen)

- $VZ_{Ergebnis} = VZ_{Faktor\ 1} \oplus VZ_{Faktor\ 2}$

Binär	Unsigned "e"	Excess-127 „char“
0000 0000	0	-127
0000 0001	1	-126
0111 1111	127	0
1111 1110	254	+127
1111 1111	255	+128

- **Multiplikation zweier Zahlen in 2er-Komplement Darstellung,
geht nicht direkt => Umwandeln in VZ-Betrag**

- Spezielle Multiplikationsalgorithmen (wie Booth)
=> Direkte Multiplikation von Zahlen im 2er Kompliment.

- Multiplikation von Gleitkommazahlen:

- Mantissen beider Zahlen multiplizieren und ihre Exponenten addieren:

- $(m_1 \times 2^{char1}) \times (m_2 \times 2^{char2}) = (m_1 \times m_2) \times 2^{char1+char2-offset}$

- Ergebnis muss nach Multiplikation unter Umständen noch normalisiert werden

- Bei unsigned-int-Addition der Charakteristiken (z.B. in Excess-127 Darstellung):

- $char1 = (e1 - offset); \quad char2 = (e2 - offset)$

- Bsp:

- $2^{char1} = 2^{-6} \Rightarrow char1 = e1 - 127 \Rightarrow -6 = e1 - 127 \Rightarrow e1 = 121 = 64 + 32 + 16 + 8 + 1 = 0111\ 1001$

- $2^{char2} = 2^4 \Rightarrow char2 = e2 - 127 \Rightarrow 4 = e2 - 127 \Rightarrow e2 = 131 = 128 + 2 + 1 = 1000\ 0011$

- $2^{-6} \times 2^4 = 2^{-6+4} = 2^{-2}$

←
-127

125 (Excess-127)
252 (unsigned)

+

1111 1100
0111 1111
- 1111 1110
0111 1101

-2 (Excess-127)
125 (unsigned)

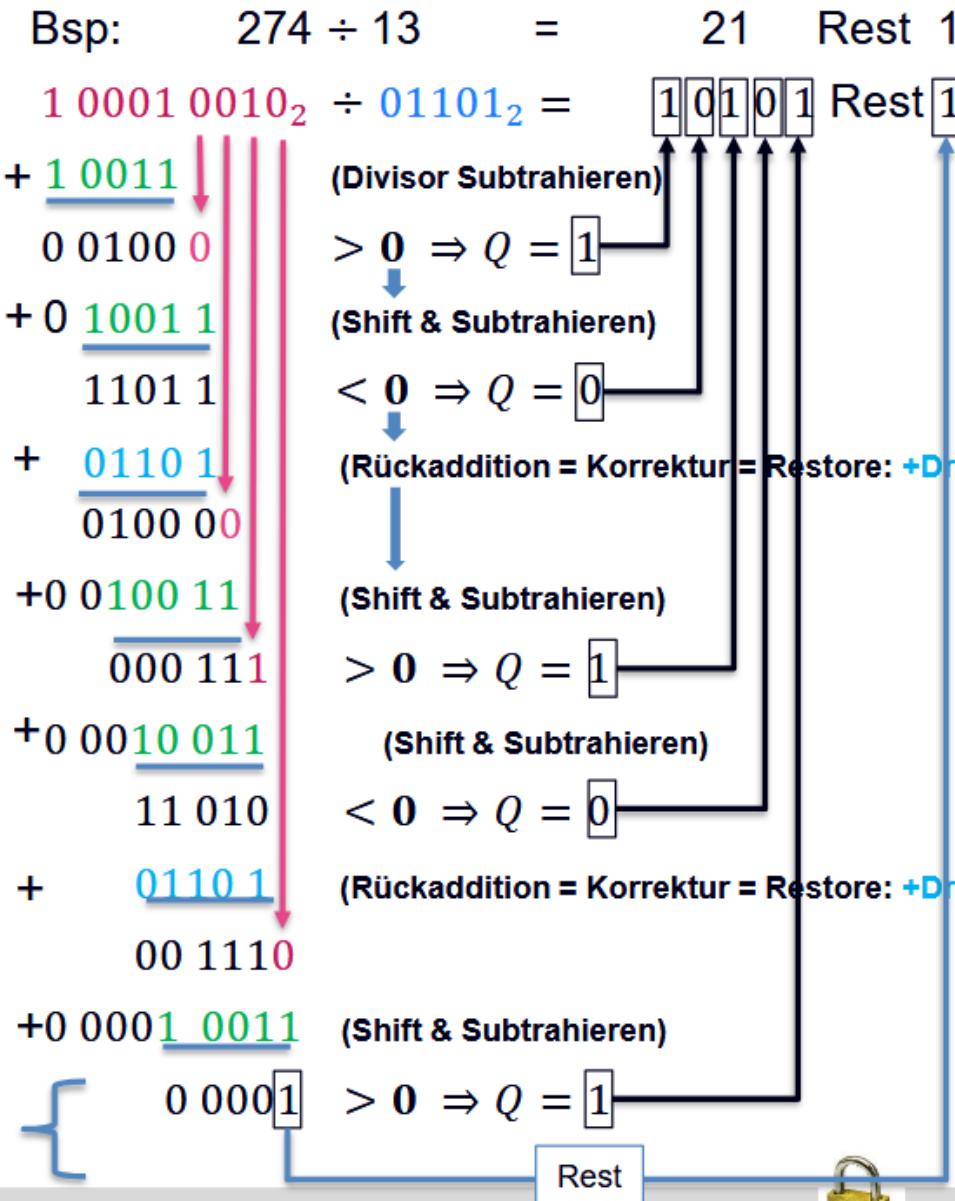
Restoring Division im Dualsystem am Beispiel

Quotient (Q) = Dividend (Dd) \div Divisor (Dr) + ggf. Rest; Rest < Divisor

• 274	= 1 0001 0010 ₂
• 13	= 0 1101 ₂
• 2erK(13)	= 1 0011 ₂
• 21	= 1 0101 ₂

entspricht Subtraktion Dr

- Im Dualsystem ist für die Bestimmung der Quotienten-Stelle nur eine Subtraktion erforderlich
- Im Dualsystem sind als Ergebnis einer Division nur diese Werte möglich:
 - Dd < (augenblicklicher Dr) = 0
 - Dd \geq (augenblicklicher Dr) = 1



Da Rest = 1 bzw. > 0
müsste gerundet
oder abgeschnitten werden

Restoring Division im Dualsystem am Beispiel

Quotient (Q) = Dividend (Dd) \div Divisor (Dr) + ggf. Rest; Rest < Divisor

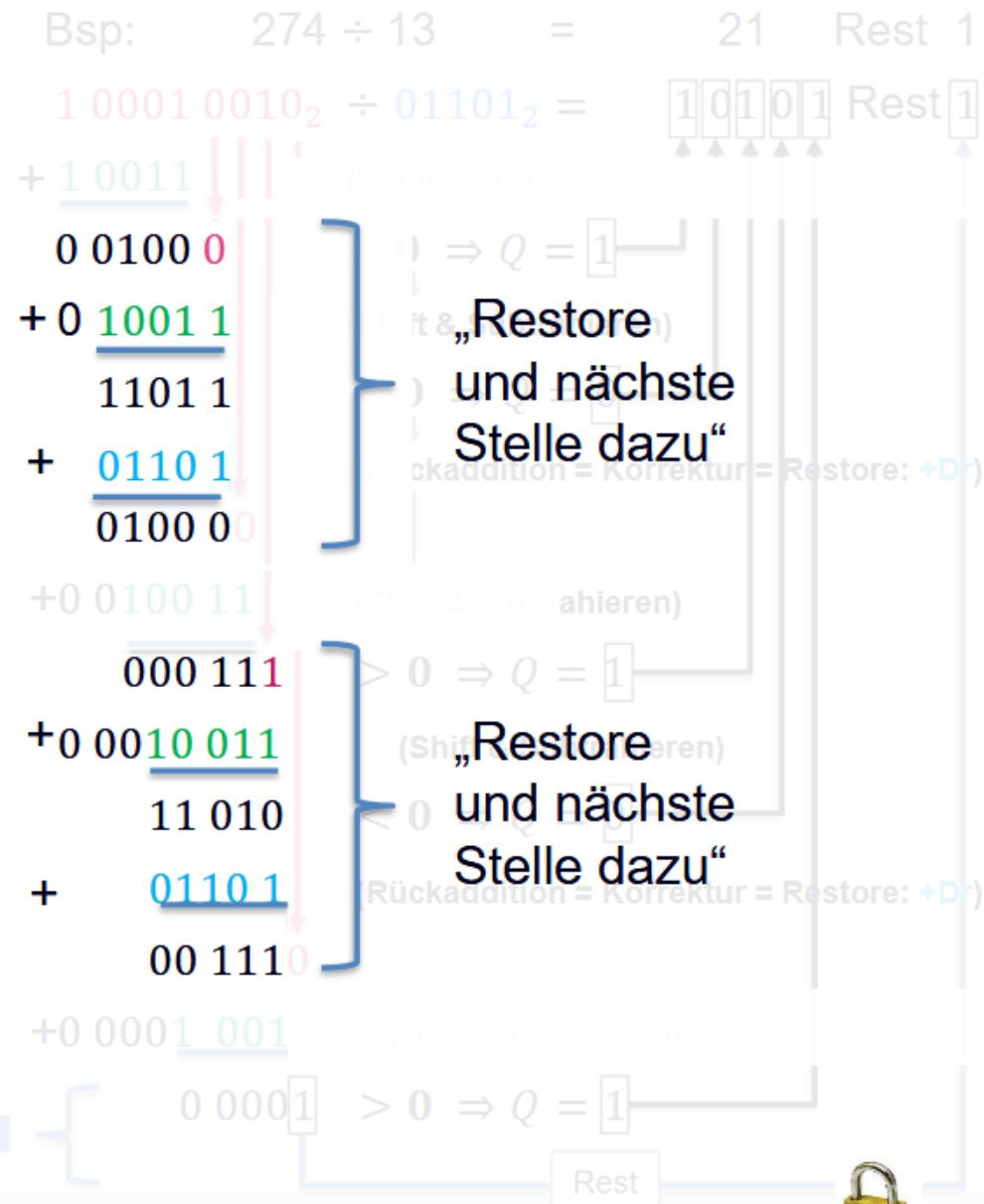


Restoring-Division:
Durch Rückaddition/
Korrektur/ Restore
wird ursprünglicher
Dividend bzw. Rest
wieder hergestellt

• Dd \geq (augenblicklicher Dr) = 0

• Dd \geq (augenblicklicher Dr) = 1

Der Rest = 1 bzw. > 0
müsste gerundet
oder abgeschnitten werden

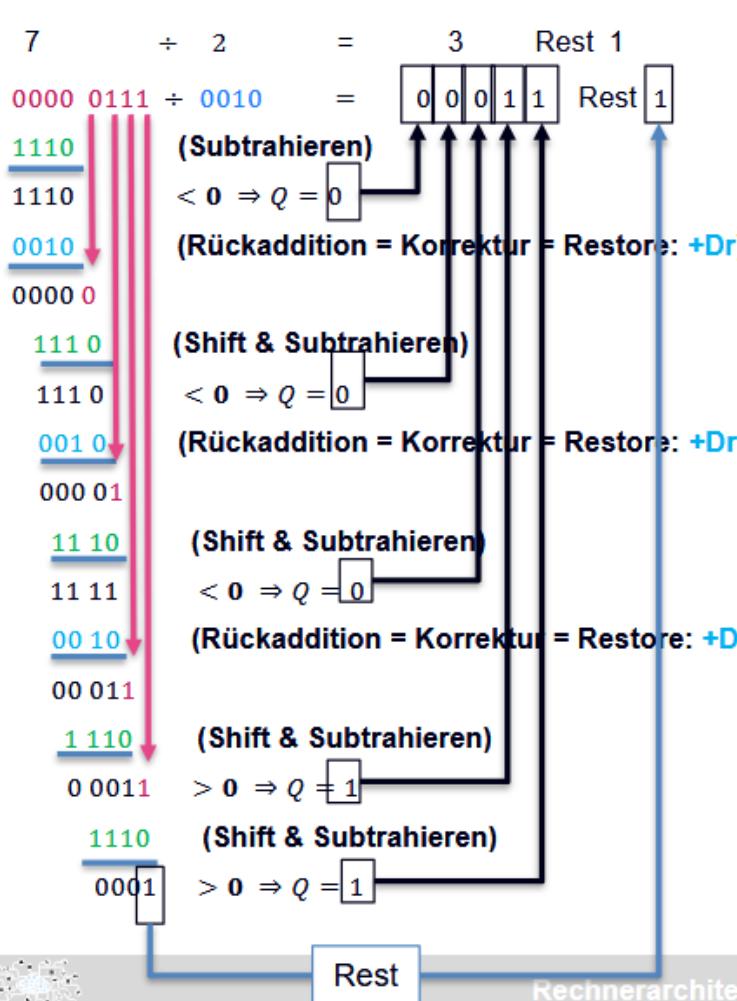


Restoring-Division: Algorithmus und Hardware

am Beispiel: $7 \div 2 = 3 \text{ Rest } 1$

Quotient (Q) = Dividend (Dd) \div Divisor (Dr) + ggf. Rest; Rest < Divisor

- $Dd = 7 = 0000\ 0111_2$
- $Dr = 2 = 0010_2$
- $2\text{erK}(2) = 1110_2$
- $Q = 3 = 0000\ 0011_2$



Implementierung in 8-bit Hardware:

- Remainder-Register (8-bit) („Rest“)
- Divisor-Register (8-bit)
- Quotient-Register (4-bit)

Initialisierung: (4-bit unsigned int)

- Remainder-Register: $Dd = 0111_2$
- Divisor-Register: linke Hälfte: $Dr = 0010_2$
- Quotient-Register = 0..0

Schleife #	Beschreibung	Divisor Register (8-bit)	Rest Register (8-bit)	Quotient Register (4-bit)
0	Initialisierung	0010 0000	0000 0111	0000



Restoring-Division: Algorithmus und Hardware

am Beispiel: $7 \div 2 = 3 \text{ Rest } 1$

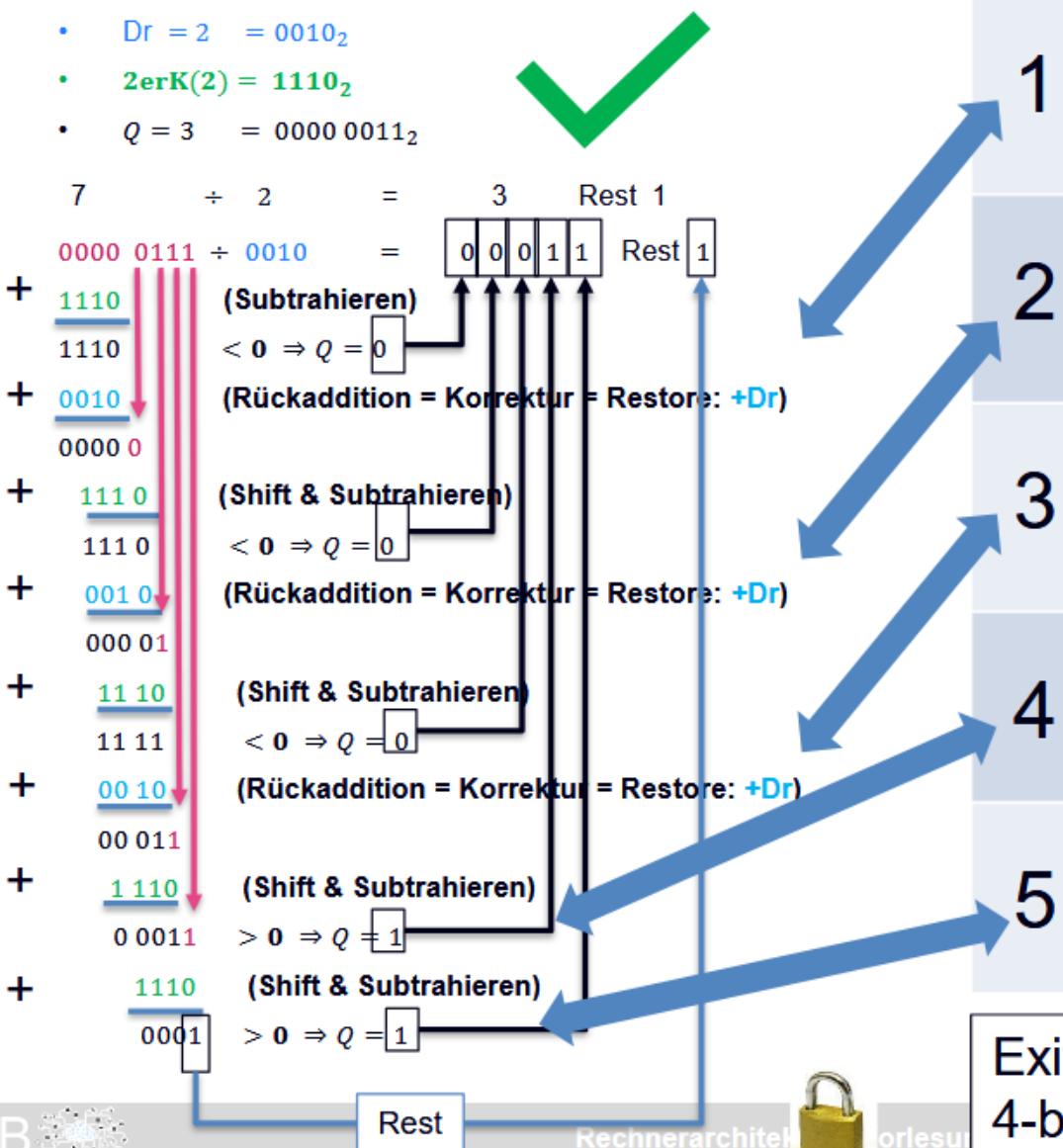
Quotient (Q) = Dividend (D_d) \div Divisor (D_r) + ggf. Rest; Rest $<$ Divisor

- $D_d = 7 = 0000\ 0111_2$

- $D_r = 2 = 0010_2$

- $2\text{erK}(2) = 1110_2$

- $Q = 3 = 0000\ 0011_2$



Schleife #	Beschreibung	Divisor Register (8-bit)	Rest Register (8-bit)	Quotient Register (4-bit)
0	Initialisierung	0010 0000	0000 0111	0000
1	Rest := Rest - Divisor * Rest := Rest + Divisor * shl Quotient	0010 0000	1110 0111	0000
2	shl Divisor	0001 0000	0000 0111	0000
3	Rest := Rest - Divisor * Rest := Rest + Divisor * shl Quotient	0001 0000	1111 0111	0000
4	shl Divisor	0000 1000	0000 0111	0000
5	Rest := Rest - Divisor * Rest := Rest + Divisor * shl Quotient	0000 1000	0000 0111	0000
	shl Divisor	0000 0100	0000 0111	0000
	Rest := Rest - Divisor	0000 0100	0000 0011	0000
	Wenn Rest ≥ 0 : * shl Quotient ($Q_0 = 1$)	0000 0100	0000 0011	0001
	shl Divisor	0000 0010	0000 0011	0001
	Rest := Rest - Divisor	0000 0010	0000 0001	0001
	Wenn Rest ≥ 0 : * shl Quotient ($Q_0 = 1$)	0000 0010	0000 0001	0011
	shl Divisor	0000 0001	0000 0001	0011

Exit: nach $n+1$ Schleifen
4-bit \Rightarrow 5 Schleifen

Ergebnis

Restoring-Division

Beobachtung

Im Divisor sind immer mindestens 50% der bits 0.
(Mind. Die Hälfte der ALU „verschwendet“)

Verbesserter Algorithmus möglich ?

Ja.

Schleife #	Beschreibung	Divisor Register (8-bit)	Rest Register (8-bit)	Quotient Register (4-bit)
0	Initialisierung	0010 0000	0000 0111	0000
1	Rest := Rest- Divisor	0010 0000	1110 0111	0000
	Wenn Rest <0: Restore * Rest := Rest + Divisor * shl Quotient ($Q_0 = 0$)	0010 0000	0000 0111	0000
	shr Divisor	0001 0000	0000 0111	0000
2	Rest := Rest- Divisor	0001 0000	1111 0111	0000
	Wenn Rest <0: Restore * Rest := Rest + Divisor * shl Quotient ($Q_0 = 0$)	0001 0000	0000 0111	0000
	shr Divisor	0000 1000	0000 0111	0000
3	Rest := Rest- Divisor	0000 1000	1111 1111	0000
	Wenn Rest <0: Restore * Rest := Rest + Divisor * shl Quotient ($Q_0 = 0$)	0000 1000	0000 0111	0000
	shr Divisor	0000 0100	0000 0111	0000
4	Rest := Rest- Divisor	0000 0100	0000 0011	0000
	Wenn Rest ≥ 0 : * shl Quotient ($Q_0 = 1$)	0000 0100	0000 0011	0001
	shr Divisor	0000 0010	0000 0011	0001
5	Rest := Rest- Divisor	0000 0010	0000 0001	0001
	Wenn Rest ≥ 0 : * shl Quotient ($Q_0 = 1$)	0000 0010	0000 0001	0011
	shr Divisor	0000 0001	0000 0001	0011



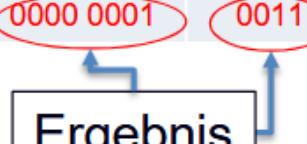
Vergleich Verbesserte Restoring-Division und Restoring-Division

am Beispiel: $7 \div 2 = 3 \text{ Rest } 1$

Schleife #	Beschreibung	Divisor Register (4-bit)	Rest Register (8-bit)	Quotient Register (4-bit)
0	Initialisierung	0010	0000 0111	unused
	1) shl Rest ($Q_0 = 0$)	0010	0000 1110	
	Rest := Rest- Divisor	0010	1110 1110	
1	Wenn Rest <0: Restore * Rest := Rest + Divisor * shl Rest ($R_0 = 0$)	0010	0001 1100	
	Rest := Rest- Divisor	0010	1110 1110	
2	Wenn Rest <0: Restore * Rest := Rest + Divisor * shl Rest ($R_0 = 0$)	0010	0011 1000	
	Rest := Rest- Divisor	0010	0001 1000	
3	Wenn Rest ≥ 0 : * shl Rest ($R_0 = 1$)	0010	0011 0001	
	Rest := Rest- Divisor	0010	0001 0001	
4	Wenn Rest ≥ 0 : * shl Rest ($R_0 = 1$)	0010	0010 0011	
5	shr (linke Hälfte von Rest)	0010	0001 0011	



Schleife #	Beschreibung	Divisor Register (8-bit)	Rest Register (8-bit)	Quotient Register (4-bit)
0	Initialisierung	0010 0000	0000 0111	0000
1	Rest := Rest- Divisor	0010 0000	1110 0111	0000
	Wenn Rest <0: Restore * Rest := Rest + Divisor * shl Quotient	0010 0000	0000 0111	0000
	shr Divisor	0001 0000	0000 0111	0000
2	Rest := Rest- Divisor	0001 0000	1111 0111	0000
	Wenn Rest <0: Restore * Rest := Rest + Divisor * shl Quotient	0001 0000	0000 0111	0000
	shr Divisor	0000 1000	0000 0111	0000
3	Rest := Rest- Divisor	0000 1000	1111 1111	0000
	Wenn Rest <0: Restore * Rest := Rest + Divisor * shl Quotient	0000 1000	0000 0111	0000
	shr Divisor	0000 0100	0000 0111	0000
4	Rest := Rest- Divisor	0000 0100	0000 0011	0000
	Wenn Rest ≥ 0 : * shl Quotient ($Q_0 = 1$)	0000 0100	0000 0011	0001
	shr Divisor	0000 0010	0000 0011	0001
5	Rest := Rest- Divisor	0000 0010	0000 0001	0001
	Wenn Rest ≥ 0 : * shl Quotient ($Q_0 = 1$)	0000 0010	0000 0001	0011
	shr Divisor	0000 0001	0000 0001	0011



Non-Restoring Division

am Beispiel: $274 \div 13 = 21 \text{ Rest } 1$

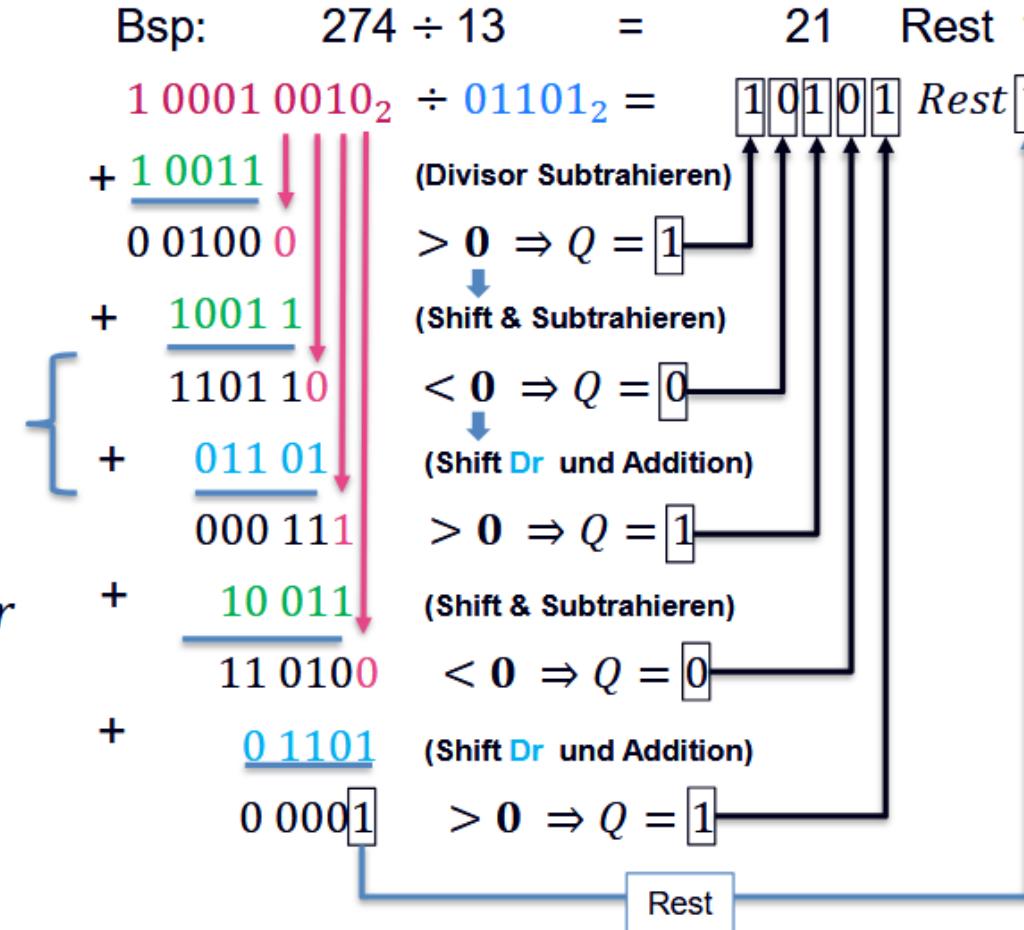
Betrachtungen:

- Einmal nach rechts verschieben
 $\hat{=} /2$
- Einmal nach links verschieben
 $\hat{=} *2$

Alternative Betrachtung:

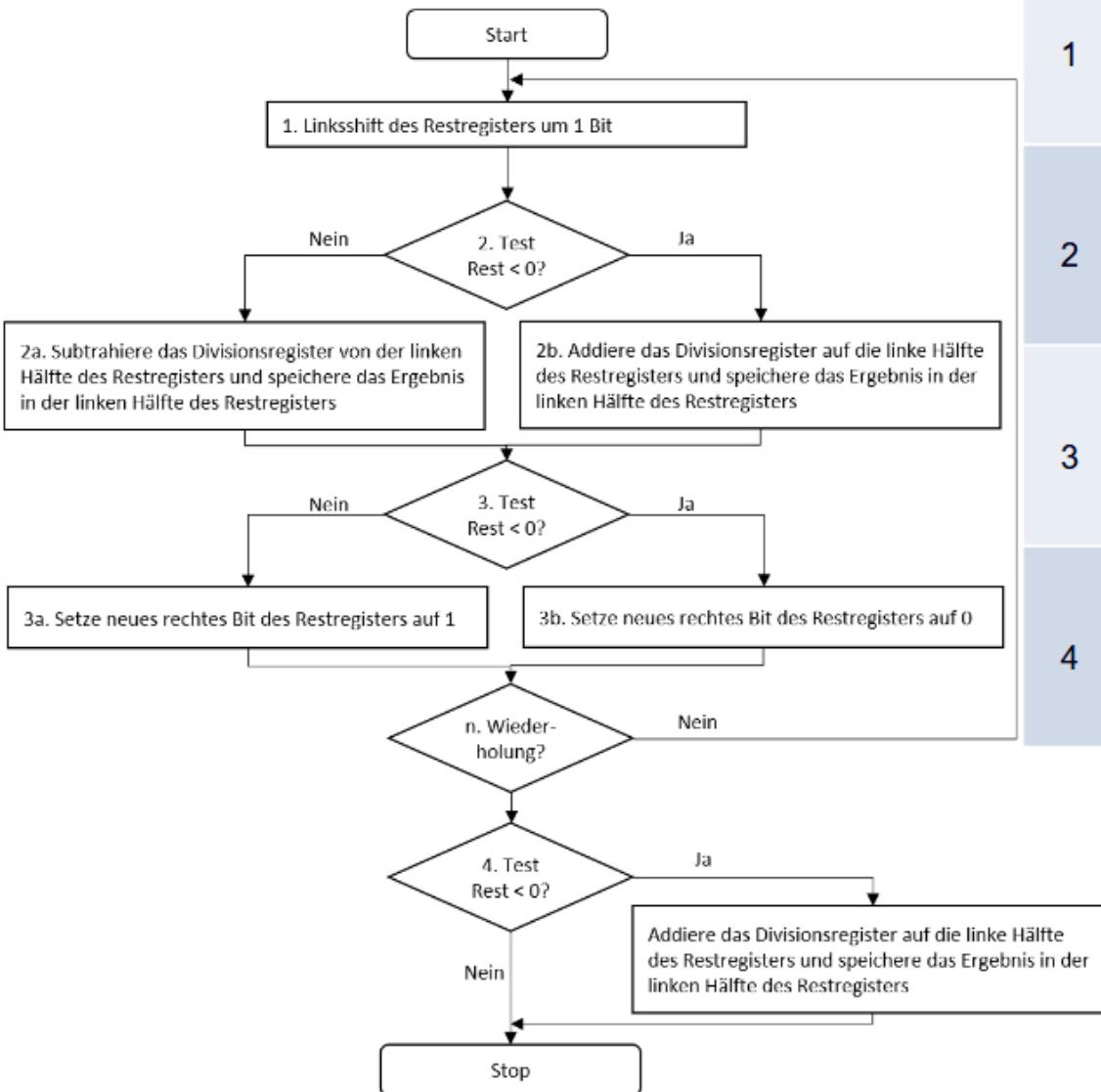
$$\text{Divisor} - \frac{1}{2} \text{Divisor} = +\frac{1}{2} \text{Divisor}$$

- Bei Unterlauf wird die Hälfte des
(1 nach rechts verschobenen)
Divisors addiert.



Non-Restoring Division

am Beispiel: $7 \div 2 = 3 \text{ Rest } 1$



Schritt #	Beschreibung	Divisor Register (4-bit)	Rest/Quotient Register (8-bit)
0	Initialisierung	0010	0000 0111
1	shl (Rest/Quotient) Wenn Rest >0: Rest := Rest - Divisor	0010	0000 1110
2	Wenn Rest <0: $Q_0 = 0$ shl (Rest/Quotient) Wenn Rest <0: Rest := Rest + Divisor	0010	1110 1110
3	Wenn Rest <0: $Q_0 = 0$ shl (Rest/Quotient) Wenn Rest <0: Rest := Rest + Divisor	0010	1111 1100
4	Wenn Rest <0: $Q_0 = 1$ shl (Rest/Quotient) Wenn Rest >0: Rest := Rest - Divisor	0010	0001 1001
	Wenn Rest >0: $Q_0 = 1$	0010	0011 0010

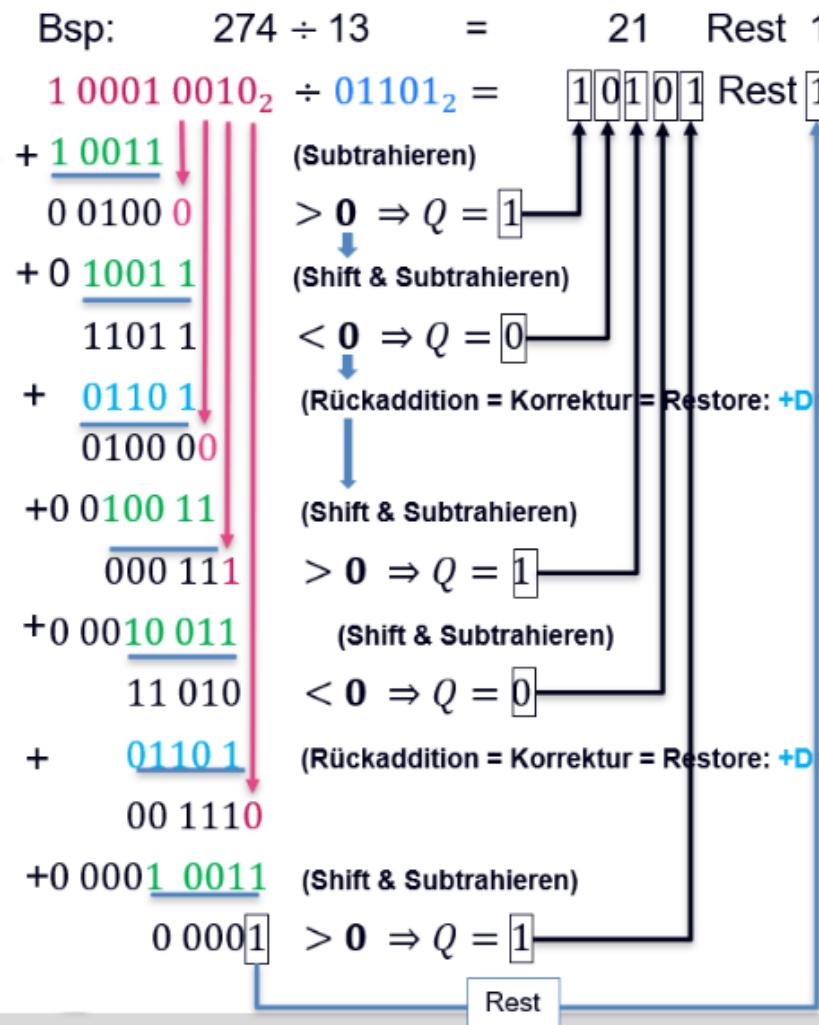
Rest Ergebnis



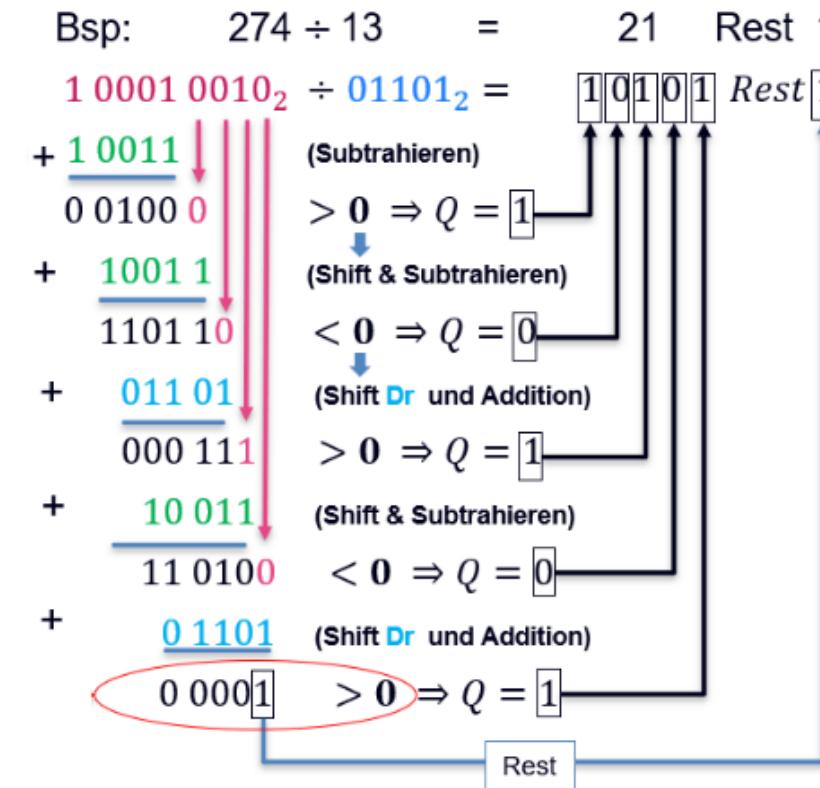
Vergleich Restoring-Division und Non-Restoring-Division

am Beispiel: $274 \div 13 = 21 \text{ Rest } 1$

Restoring-Division



Non-Restoring-Division



Vergleich Restoring-Division und Non-Restoring-Division

am Beispiel: $7 \div 2 = 3 \text{ Rest } 1$

Restoring-Division

Schleife #	Beschreibung	Divisor Register (8-bit)	Rest Register (8-bit)	Quotient Register (4-bit)
0	Initialisierung	0010 0000	0000 0111	0000
1	Rest := Rest - Divisor * Rest := Rest + Divisor * shr Quotient ($Q_0 = 0$)	0010 0000	1110 0111	0000
2	shr Divisor	0001 0000	0000 0111	0000
3	Rest := Rest - Divisor * Rest := Rest + Divisor * shr Quotient ($Q_0 = 0$)	0001 0000	1111 0111	0000
4	shr Divisor	0000 1000	0000 0111	0000
5	Rest := Rest - Divisor * Rest := Rest + Divisor * shr Quotient ($Q_0 = 1$)	0000 1000	1111 1111	0000
	shr Divisor	0000 0100	0000 0111	0000
	Rest := Rest - Divisor	0000 0100	0000 0011	0000
	Wenn Rest ≥ 0 : * shr Quotient ($Q_0 = 1$)	0000 0100	0000 0011	0001
	shr Divisor	0000 0010	0000 0011	0001
	Rest := Rest - Divisor	0000 0010	0000 0001	0001
	Wenn Rest ≥ 0 : * shr Quotient ($Q_0 = 1$)	0000 0010	0000 0001	0011
	shr Divisor	0000 0001	0000 0001	0011

Non-Restoring-Division

Schleife #	Beschreibung	Divisor Register (4-bit)	Rest/ Quotient Register (8-bit)
0	Initialisierung	0010	0000 0111
1	shl (Rest/Quotient) Wenn Rest > 0 : Rest := Rest - Divisor	0010	0000 1110
2	Wenn Rest < 0 : $Q_0 = 0$ shl (Rest/Quotient) Wenn Rest < 0 : Rest := Rest + Divisor	0010	1110 1110
3	Wenn Rest < 0 : $Q_0 = 0$ shl (Rest/Quotient) Wenn Rest < 0 : Rest := Rest + Divisor	0010	1111 1100
4	Wenn Rest < 0 : $Q_0 = 1$ shl (Rest/Quotient) Wenn Rest > 0 : Rest := Rest - Divisor	0010	1111 1100
	Wenn Rest > 0 : $Q_0 = 1$ shl (Rest/Quotient) Wenn Rest > 0 : Rest := Rest - Divisor	0010	0011 0010



Restoring-Division, Non-performing-Division und Non-Restoring-Division

- Der Non-Restoring-Algorithmus verzichtet auf Restoring (Wiederherstellungsschritte): daher Non-Restoring
- Der Non-Performing-Algorithmus bildet einen neuen Rest (Dividend) nur, wenn die **Differenz nicht negativ** ist.



Division – Abschließende Beobachtungen

- Bei Division durch 0 muss ein Ausnahmezustand erkannt werden und an die Steuereinheit (Prozessor) weitergemeldet werden.
- Die Division muss abgebrochen werden, wenn die vorhandene Bit-Anzahl des Ergebnisregisters ausgeschöpft ist (periodische Dualbrüche).
- Schaltungs-Bausteine für die Multiplikation können nach Modifikation auch für den Grundalgorithmus der Division eingesetzt werden:
 - Linksschieben des Dividenden (statt Rechtsschieben des Multiplikanten)
 - Subtraktion des Divisors (statt Addition des Multiplikators)



Aufbau und Arbeitsweise von Rechensystemen

Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Anforderung aus der Rechnerarithmetik:

- Addition (Wort-Addierer)
- Subtraktion (Wort-Addierer, 2er-Komplement)
- Multiplikation (Booth: Subtraktion, shift right)
- Division (Addition, Subtraktion, shift right, shift left)



Kern der ALU:

- Wort-Addierer
- Shifter

Des Weiteren: 1er/2er-Komplement, +1, -1, bitweise Disjunktion/Konjunktion/XOR



Aufbau und Arbeitsweise von Rechensystemen

Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Anforderung aus der Rechnerarithmetik:

- Addition (Wort-Addierer)
- Subtraktion (Wort-Addierer, 2er-Komplement)
- Multiplikation (Booth: Subtraktion, shift right)
- Division (Addition, Subtraktion, shift right, shift left)

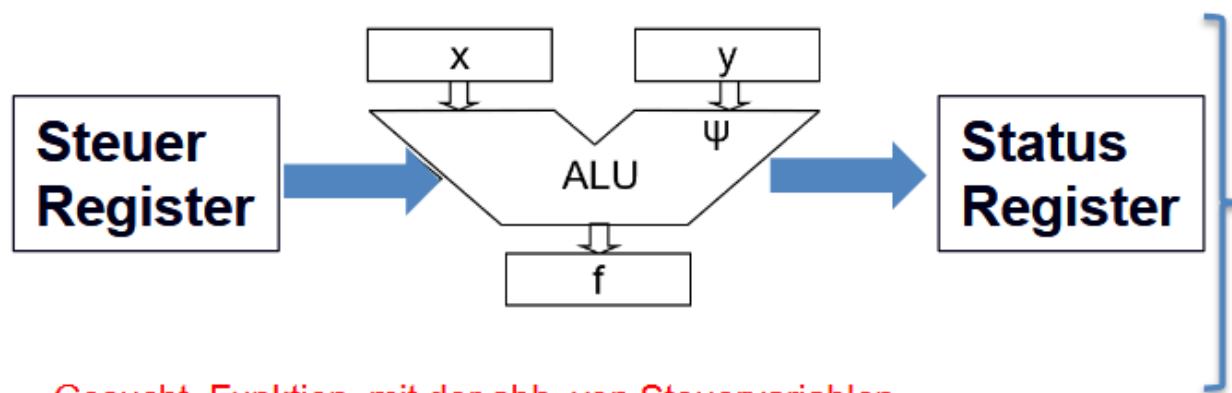


Kern der ALU:

- Wort-Addierer
- Shifter

Des Weiteren: 1er/2er-Komplement, +1, -1, bitweise Disjunktion/Konjunktion/XOR

Ziel:



Die Worte x und y sollen
arithmetisch UND
logisch
flexibel verknüpfbar sein.

Gesucht: Funktion, mit der abh. von Steuervariablen unterschiedliche Operationen einer ALU realisiert werden können



Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Baustein: parametrische Schaltung (Funktion in Hardware)

Parametrisierte Schaltung Ψ :

Herzstück der Steuerung

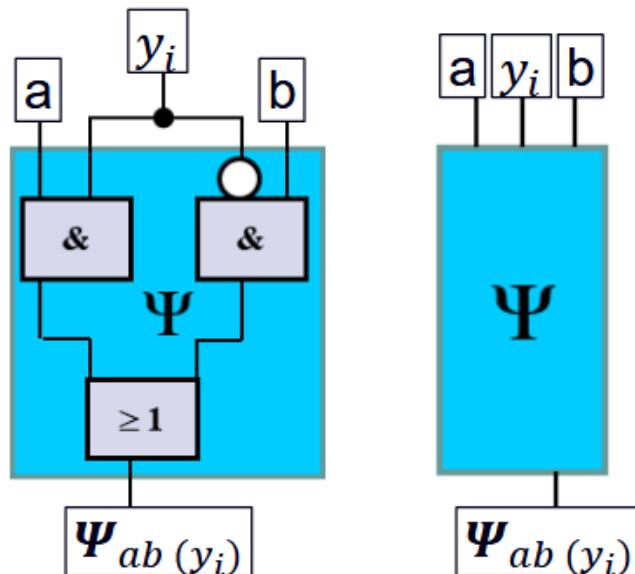
Für jedes bit y_i im Wort y:

werden alle 4 einstelligen Boole'schen

Funktionen $y_i, \overline{y_i}, 0, 1$ erzeugt!

Parametrisierte Schaltung $\Psi_{ab}(y_i)$:

- aus Eingangswert y_i
- und Steuerparametern a, b



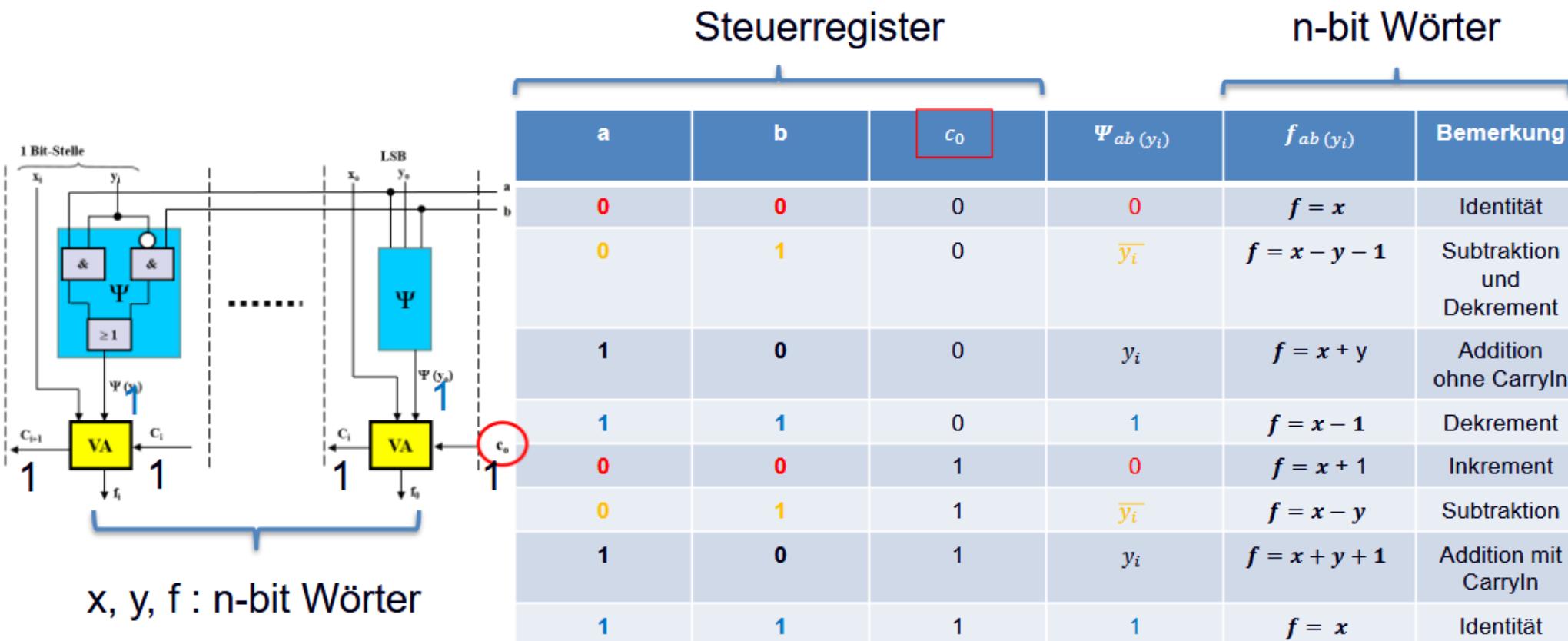
a	b	$\Psi_{ab}(y_i)$
0	0	0
0	1	$\overline{y_i}$
1	0	y_i
1	1	1

Griechischer Buchstabe Ψ wird gesprochen „Phi“



Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Konstruktion arithmetischer Funktionen

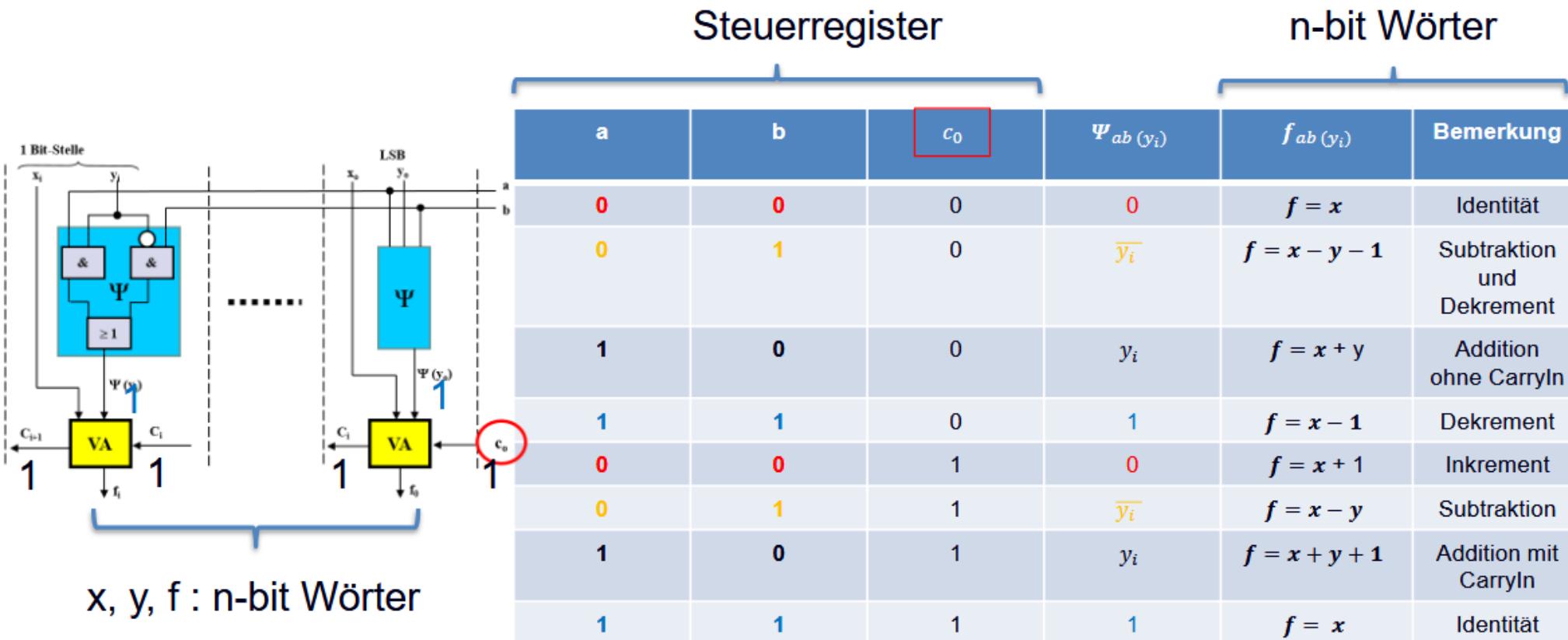


$$\Psi_i = (ay_i + b\bar{y}_i)$$



Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Konstruktion arithmetischer Funktionen



Beobachtung:

- Identität $f = x$: kommt zweimal vor
- $f = x + y$ und $f = x + y + 1 \Rightarrow$ sinnvoll, damit Additionen möglich
- $f = x - y$
- Inkrement und Dekrement
- $f = x - y - 1$

Notwendige
Funktionen

eher unnötig

Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Konstruktion logischer Funktionen

Anforderung bitweise Operationen:

- Negation: Einerkomplement eines Wortes x
- Antivalenz: bitweises XOR zweier Worte x und y
- Äquivalenz: bitweiser Vergleich: Sind zwei Worte x und y identisch?

Problem bei bitweisen Operationen: Carry-Bits

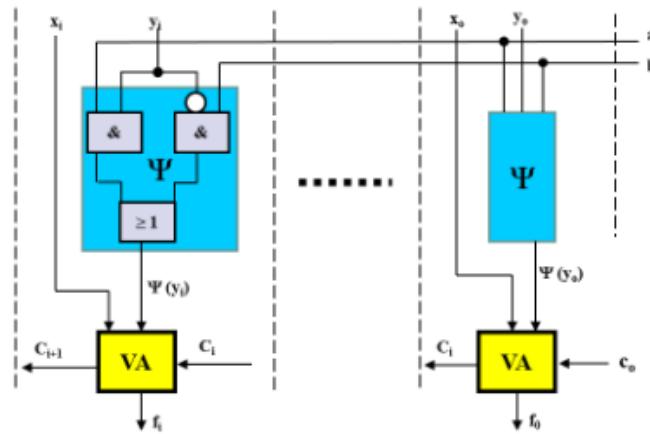
Lösung: Carry-Bits „ausschalten“



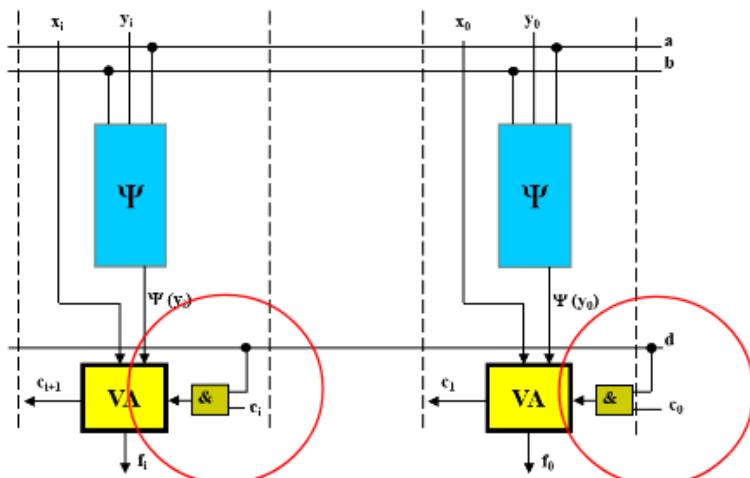
Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Konstruktion logischer Funktionen

Arithmetische Einheit:



Ergänzung:



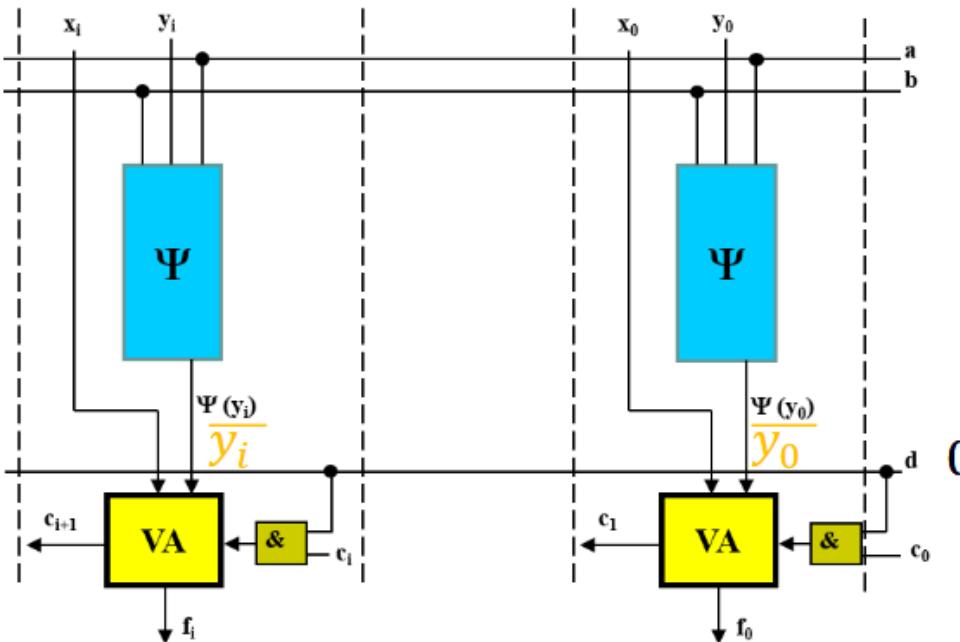
Ein weiteres Steuersignal d :

- $d = 1 \Rightarrow$ wie arithmetische Einheit
- $d = 0 \Rightarrow$ Carry-Bits alle disabled



Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Konstruktion logischer Funktionen



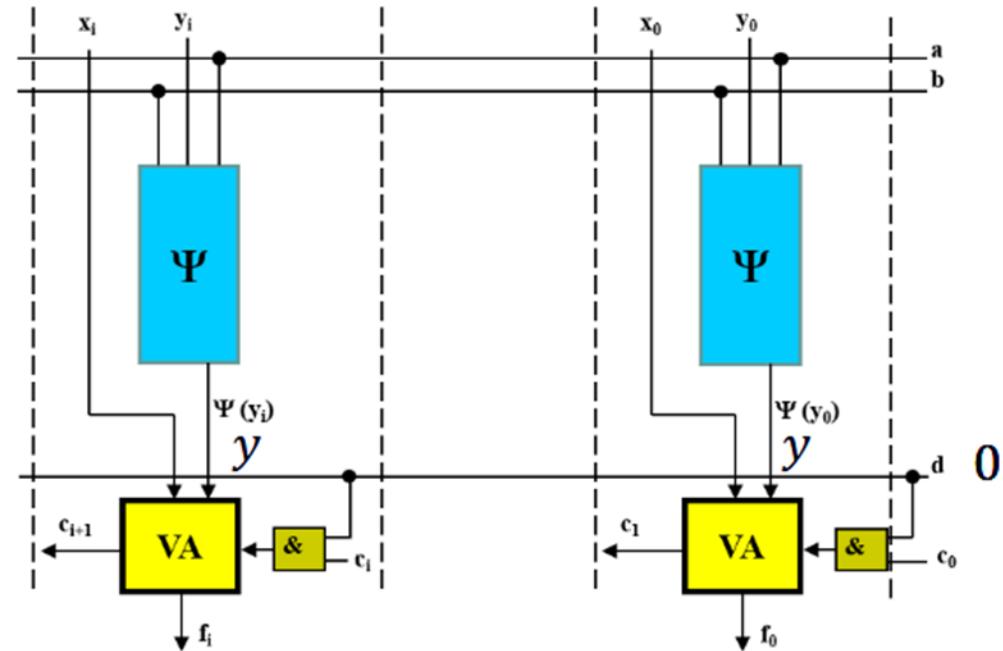
Steuerregister

n-bit Wörter

a	b	c_0	d	$\Psi_{abcd}(y)$	$f_{abcd(x,y)}$	Bemerkung
0	0	-	0	0..0	$f = x$	Identität
0	1	-	0	\bar{y}	$f = x \oplus \bar{y}$	Wenn x_i und y_i beide gleich dann $f_i=1$ Wenn x_i und y_i unterschiedlich, dann $f_i=0$ \Rightarrow Äquivalenz $x \Leftrightarrow y$ wenn $f = 2^n - 1$

Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Konstruktion logischer Funktionen



Steuerregister

n-bit Wörter

a	b	c_0	d	$\Psi_{abcd}(y)$	$f_{abcd(x,y)}$	Bemerkung
0	0	-	0	0..0	$f = x$	Identität
0	1	-	0	\bar{y}	$f = x \oplus \bar{y}$	Äquivalenz $x \Leftrightarrow y$ wenn $f = 2^n - 1$
1	0	-	0	y	$f = x \oplus y$	Wenn x_i und y_i beide gleich dann $f_i = 0$ Wenn x_i und y_i unterschie- dlich, dann $f_i = 1$ => Antivalenz wenn $f = 2^n - 1$

Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Konstruktion logischer Funktionen

a	b	c_0	d	$\Psi_{abcd}(y)$	$f_{abcd(x,y)}$	Bemerkung
0	0	-	0	0..0	$f = x$	Identität
0	1	-	0	\bar{y}	$f = x \oplus \bar{y}$	Äquivalenz $x \Leftrightarrow y$ wenn $f = 2^n - 1$
1	0	-	0	y	$f = x \oplus y$	Antivalenz wenn $f = 2^n - 1$
1	1	-	0	1..1	$f = x \oplus 1$ $= \bar{x}$	Negation (1er-Komplement)

0

Steuerregister

n-bit Wörter

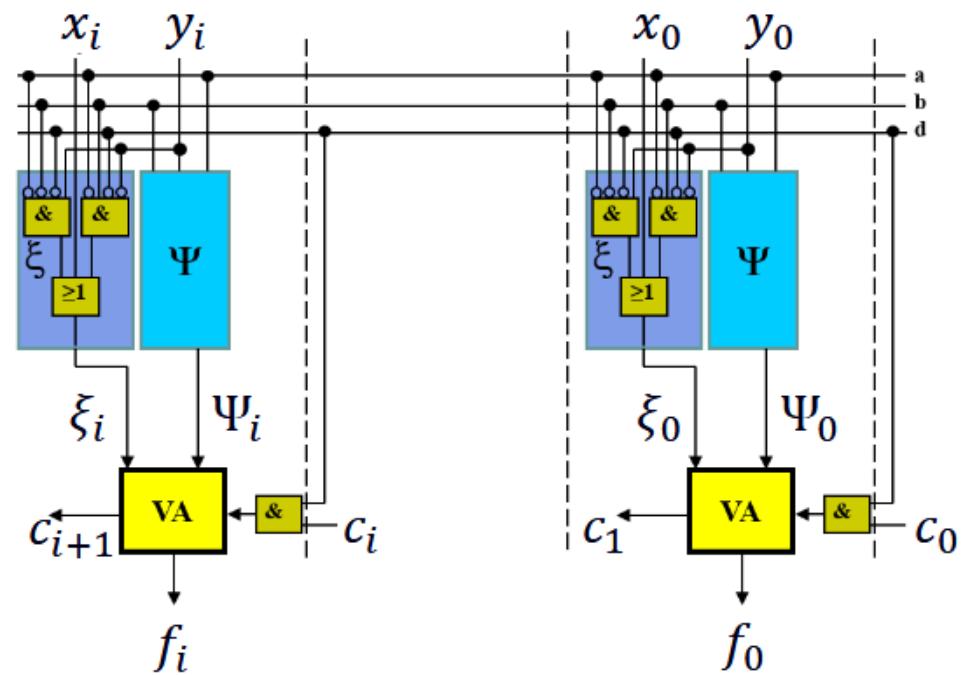
a	b	c_0	d	$\Psi_{abcd}(y)$	$f_{abcd(x,y)}$	Bemerkung
0	0	-	0	0..0	$f = x$	Identität
0	1	-	0	\bar{y}	$f = x \oplus \bar{y}$	Äquivalenz $x \Leftrightarrow y$ wenn $f = 2^n - 1$
1	0	-	0	y	$f = x \oplus y$	Antivalenz wenn $f = 2^n - 1$
1	1	-	0	1..1	$f = x \oplus 1$ $= \bar{x}$	Negation (1er-Komplement)

Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Konstruktion logischer Funktionen: Ergänzung Disjunktion & Konjunktion

- Zusätzliche Funktion:

$\bar{a}\bar{b}\bar{d}$ wird genutzt für die Disjunktion (Oder-Verknüpfung): $x + y$
 => neuer Funktionsbaustein notwendig: ξ („kleines Xi“)



Steuerregister bits

a	b	c_0	d	ψ_i	ξ_i	f_i	Bemerkung
0	0	-	0	0	$x_i + y_i$	$f_i = (y_i + x_i) \oplus 0$ $= x_i + y_i$	Disjunktion (bitweise Oder-Verknüpfung) $x_i + y_i$
0	1	-	0	\bar{y}_i	$x_i + \bar{y}_i$	$f_i = (x_i + \bar{y}_i) \oplus \bar{y}_i$ $= x_i \times y_i$	Konjunktion (bitweise Und-Verknüpfung) $x_i \times y_i$
1	0	-	0	y_i	x_i	$f_i = x_i \oplus y_i$	Antivalenz (bitweise XOR-Verknüpfung) $x_i \oplus y_i$
1	1	-	0	1	x_i	$f_i = x_i \oplus 1$ $= \bar{x}_i$	Negation (1er-Komplement) \bar{x}_i

$$\xi_i = (\bar{a}\bar{b}\bar{d}y_i + x_i + \bar{a}\bar{b}\bar{d}\bar{y}_i)$$

$$\Psi_i = (ay_i + b\bar{y}_i)$$

$$f_i = \xi_i \oplus \Psi_i = (\bar{a}\bar{b}\bar{d}y_i + x_i + \bar{a}\bar{b}\bar{d}\bar{y}_i) \oplus (ay_i + b\bar{y}_i)$$

Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Zusammenfassung und Überblick: eine erste „fertige“ ALU

Mit den beschriebenen schaltungstechnischen Ergänzungen ξ und Ψ

Sind folgende arithmetisch-logische Operationen möglich:

- logisch mit $d=0$
- arithmetisches mit $d=1$

Steuerregister

a	b	c_0	d	Ψ_i	ξ_i	f	Bemerkung
0	0	-	0	0	$x_i + y_i$	$f_i = x_i + y_i$	Disjunktion (bitweise Oder-Verknüpfung)
0	1	-	0	\bar{y}_i	$x_i + \bar{y}_i$	$f_i = x_i \times y_i$	Konjunktion (bitweise Und-Verknüpfung)
1	0	-	0	y_i	x_i	$f_i = x_i \oplus y_i$	Antivalenz (bitweise XOR-Verknüpfung)
1	1	-	0	1	x_i	$f_i = \bar{x}_i$	Negation (1er-Komplement)
0	0	0	1	0	x_i	$f = x$	Identität
0	0	1	1	0	x_i	$f = x + 1$	Inkrement
0	1	0	1	\bar{y}_i	x_i	$f = x - y - 1$	Subtraktion und Dekrement (Subtraktion im EK)
0	1	1	1	\bar{y}_i	x_i	$f = x - y$	Subtraktion
1	0	0	1	y_i	x_i	$f = x + y$	Addition ohne CarryIn
1	0	1	1	y_i	x_i	$f = x + y + 1$	Addition mit CarryIn
1	1	0	1	1	x_i	$f = x - 1$	Dekrement
1	1	1	1	1	x_i	$f = x$	Identität

Logische
Funktionen

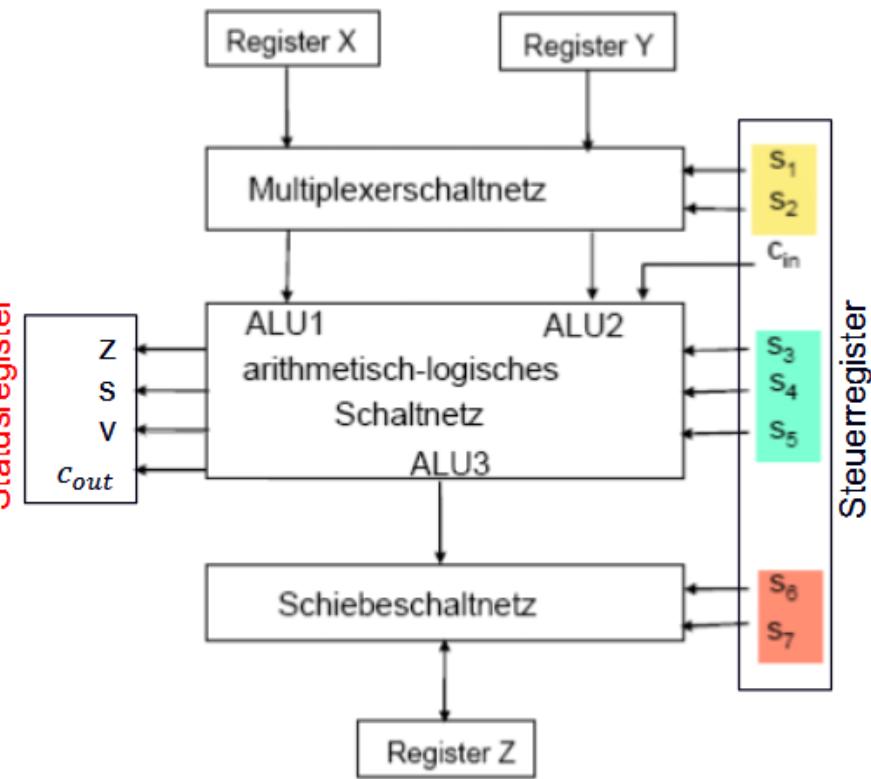
Arithmetische
Funktionen



Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Schematischer Aufbau eines Rechenwerks

Statusregister



- Register X, Y, Z
- **Statusregister**
mit den Statussignalen (z, s, v, c_{out})
- Steuerregister
mit den Steuersignalen (s₁, ..., s₇, c_{in})

Bsp zur Illustration:
Multiplexerschaltnetz

s ₁	s ₂	ALU1	ALU2
0	0	X	Y
0	1	X	0
1	0	Y	0
1	1	Y	X

ALU

s ₃	s ₄	s ₅	ALU3
0	0	0	ALU1 + ALU2 + c _{in}
usw: arithmetische und logische Verknüpfungen			

Schiebeschaltnetz

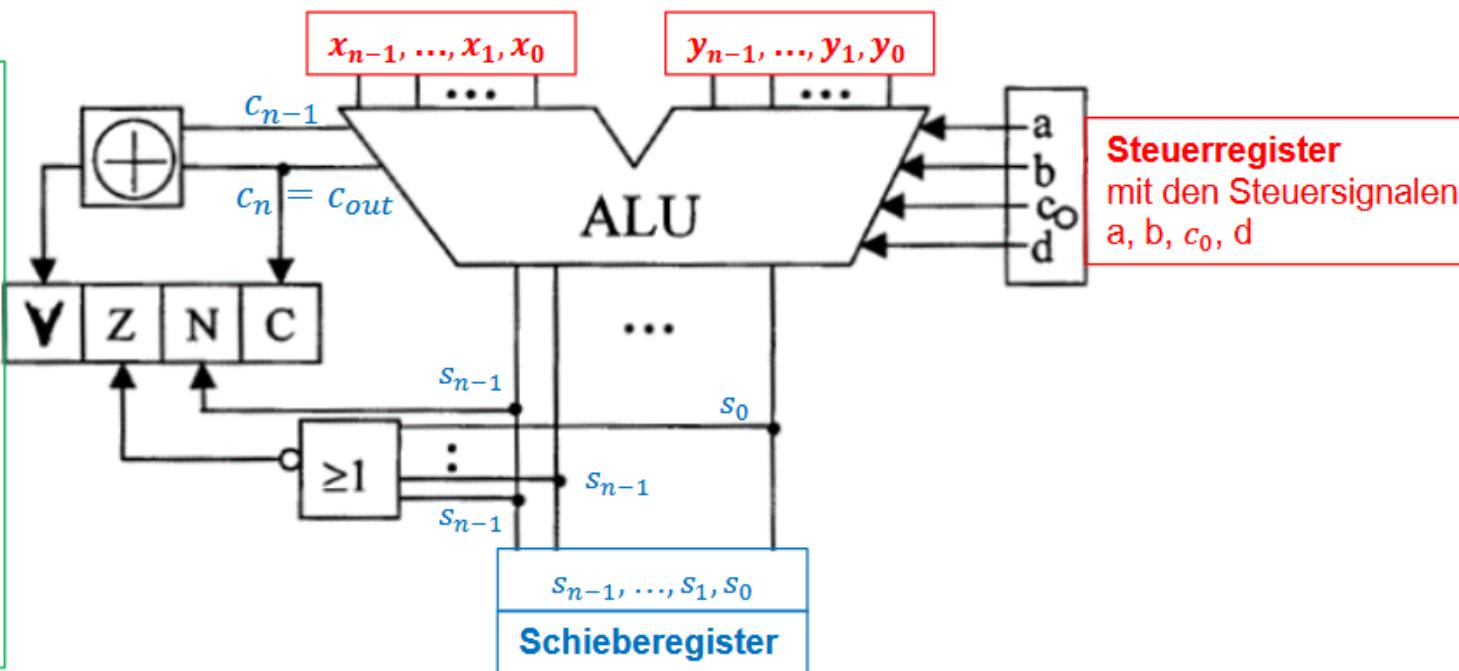
s ₆	s ₇	Z
0	0	ALU3
0	1	ALU3 ÷ 2 (shr)
1	0	ALU3 × 2 (shl)
1	1	„altes“ Z speichern (egal was ALU3 für einen neuen Wert hat)



Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Schematischer Aufbau eines Rechenwerks

- Statusregister**
mit den Statussignalen
- V (oVerflow-bit)
 $V = c_n \oplus c_{n-1}$
 - Z (Null-bit):
Wenn $Z = 0$ ist $S = 1$
 $Z = s_{n-1} \vee \dots \vee s_2 \vee s_1 \vee s_0$
 - N (Vorzeichen-bit)
 $N = 1 \Rightarrow$ negative Zahl
 - C (Carry-Out-bit $c_n = c_{out}$)



Für was kann das Statusregister noch genutzt werden?

Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

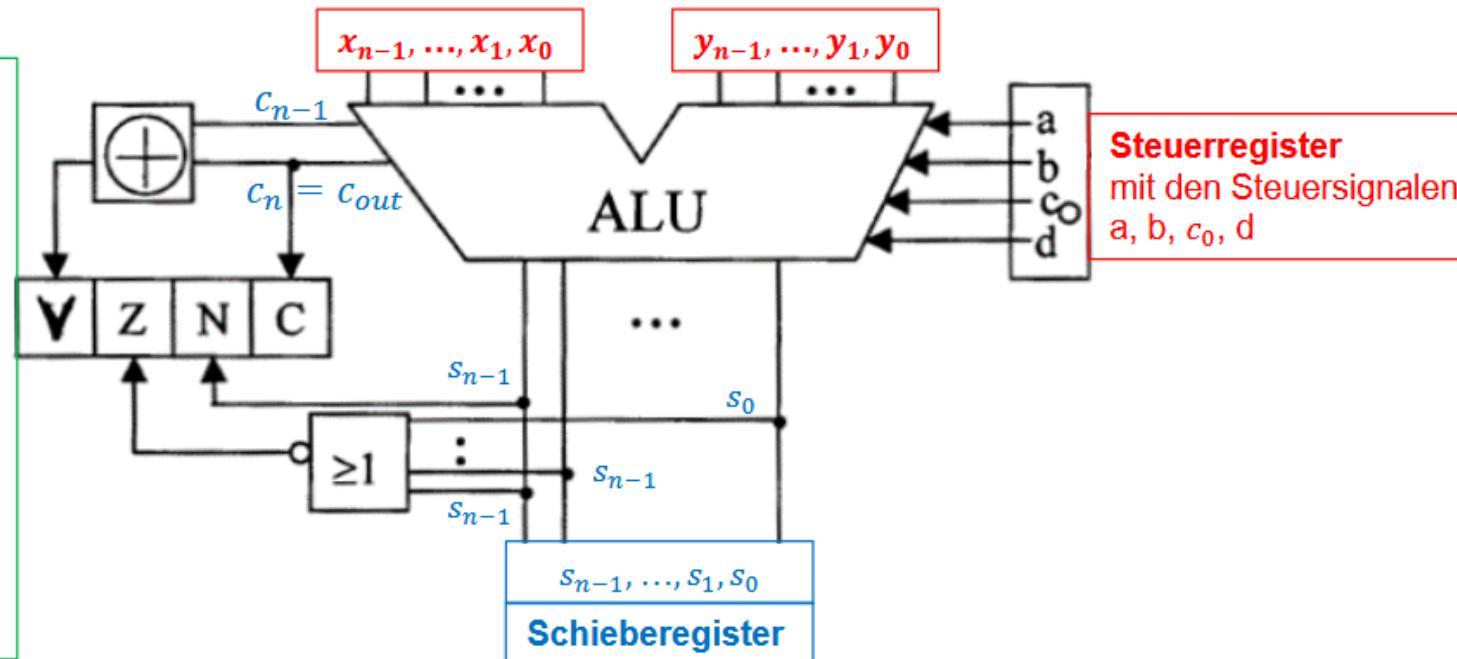
Schematischer Aufbau eines Rechenwerks

Statusregister
mit den Statussignalen
• V (oVerflow-bit)
 $V = c_n \oplus c_{n-1}$

• Z (Null-bit):
Wenn S = 0 ist Z = 1
 $Z = s_{n-1} \vee \dots \vee s_2 \vee s_1 \vee s_0$

• N (Vorzeichen-bit)
 $N = 1 \Rightarrow$ negative Zahl

• C (Carry-Out-bit $c_n = c_{out}$)



Statusregister erlaubt zusätzlich: Vergleich (Compare) von Eingabewerten X und Y:

- Subtraktion X-Y:
 $Z = 1 \Rightarrow X=Y; N = 0 \Rightarrow X > Y$
- Vorsicht: V = 0 sonst ungültige Rechnung

Aufbau und Steuerung einer Arithmetic Logic Unit (ALU)

Statusregister in modernen CPUs haben weitere Flags

- **Auxiliary-Carry-Flag AF: (Hilfsübertragsbit, Half Carry Flag)**
 - AF wird 1 gesetzt wenn es einen Übertrag an Bit 4 gegeben hat.
 - => Binary-Coded Decimal
- **Parity Flag PF: Paritätsbit**
 - even/ odd parity: 0110 0000 (**even parity-bit: 0**); 1111 1111(**even parity-bit: 1**)
- **Interrupt Enable Flag**
 - Wenn gesetzt: dann sind Interrupts enabled

https://en.wikipedia.org/wiki/Half-carry_flag

https://en.wikipedia.org/wiki/Parity_bit

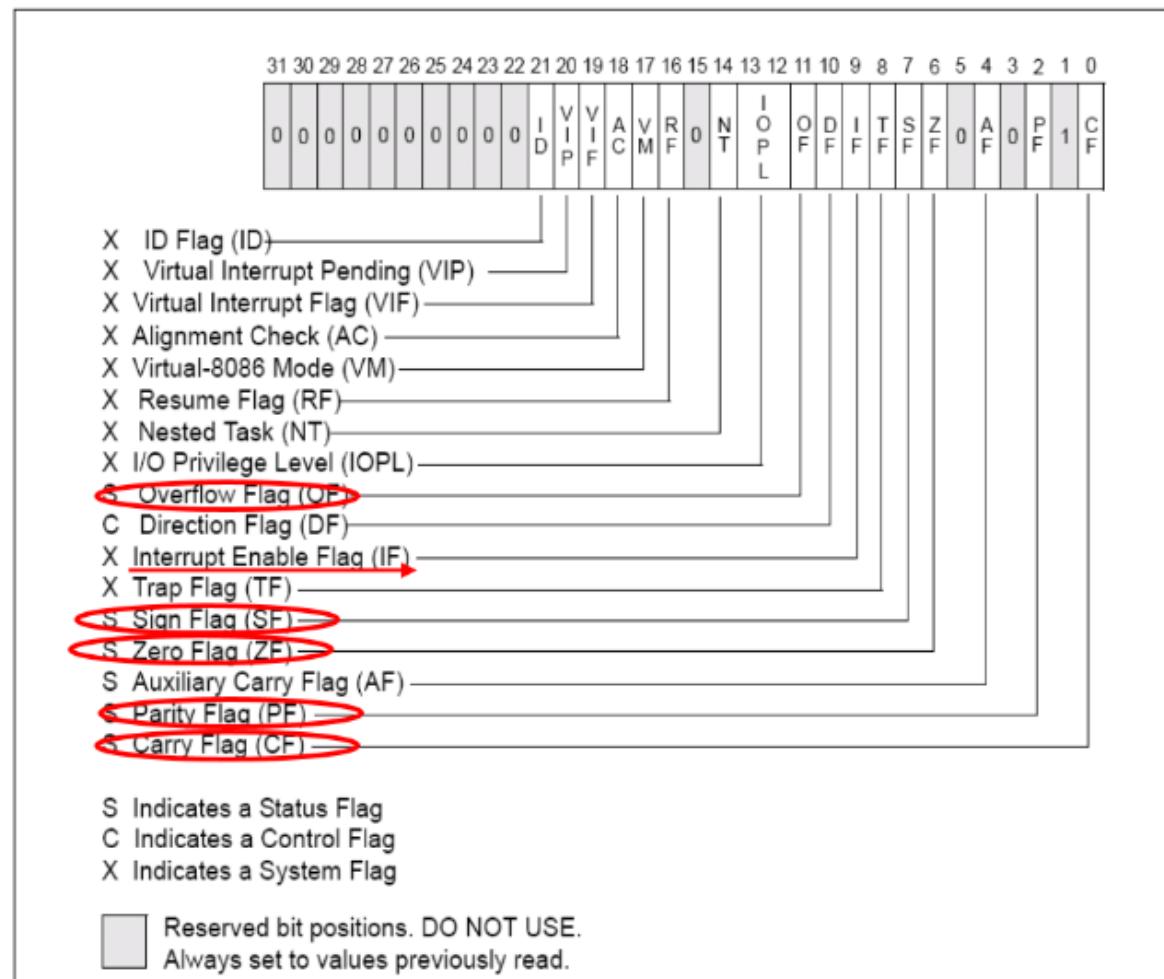


Statusregister IA-32 Architektur von Intel

Beispiel: IA-32 Architektur von Intel

Status-Bits als Resultat von arithmetischen Operationen:

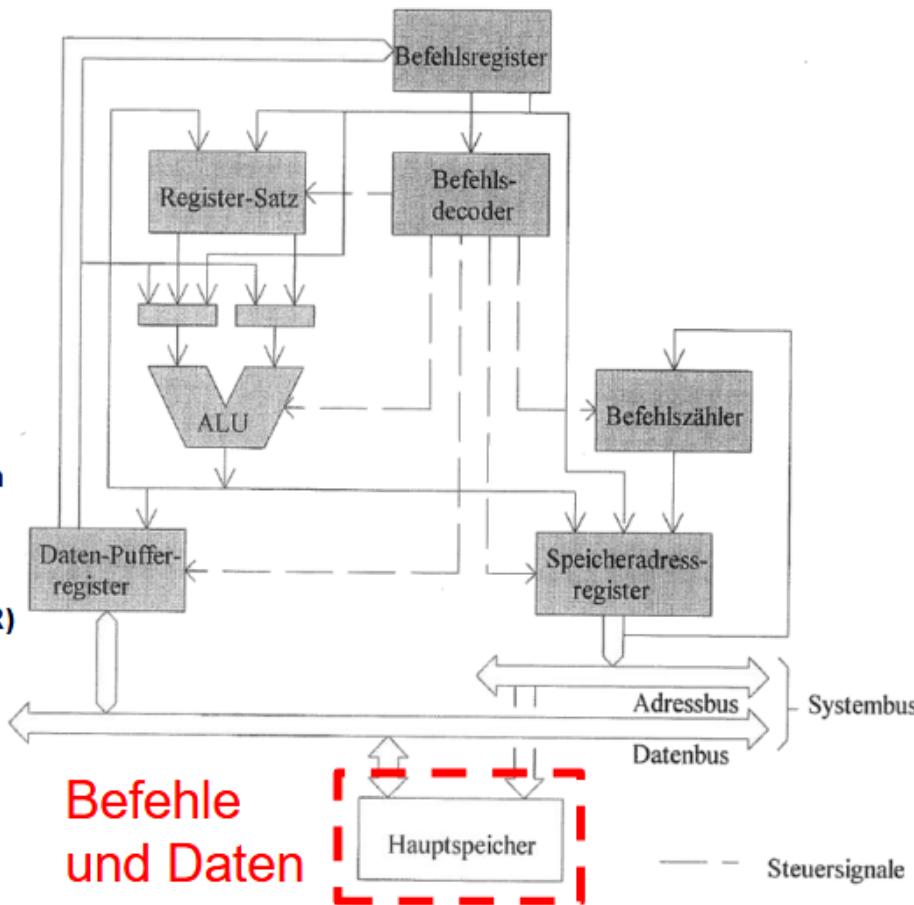
- 0 – Carry Flag (CF)
- 2 – Parity Flag (PF)
- 4 – Auxiliary Carry Flag (AF)
- 6 – Zero Flag (ZF)
- 7 – Sign Flag (SF)
- 11 – Overflow (OF)



Struktur einer zentralen Recheneinheit (CPU)

Einheiten und Steuerung eines Prozessors

- ALU, Shifter
- Register (Daten-, Adress-, Steuer-Register)
- Zur Koordination der Abläufe: Steuer- bzw. Leitwerk mit:
 - **Befehlsregister BR (oder Instruction Register IR)**
 - empfängt die Programmbefehle & speichert sie zwischen während sie gerade ausgeführt werden: vom Arbeitsspeicher geladene Maschinenbefehle werden dort abgelegt, während sie decodiert und ausgeführt werden.
 - **Befehlsdecoder**
 - interpretiert Maschinenbefehle, und erzeugt die zur Ausführung notwendigen Hardware-Steuersignale,
 - **Befehlszähler (oder Programm Counter PC bzw. Instruction Pointer IP)**
 - enthält Adresse des nächsten auszuführenden Befehls
 - **Speicheradressregister SAR (Memory Address Register MAR)** und Speicherdatenregister SDR
 - Um anzugeben, welche Adresse des Hauptspeichers in das SDR geschrieben werden soll, wird die Adresse in SAR geschrieben. SAR enthält die Adresse des Speicherwortes



Quelle: Rechnerarchitektur: Eine Einführung (etc.), Mildenberger, Malz



Steuerwerk erhält Aufgaben durch Befehle eines Programms

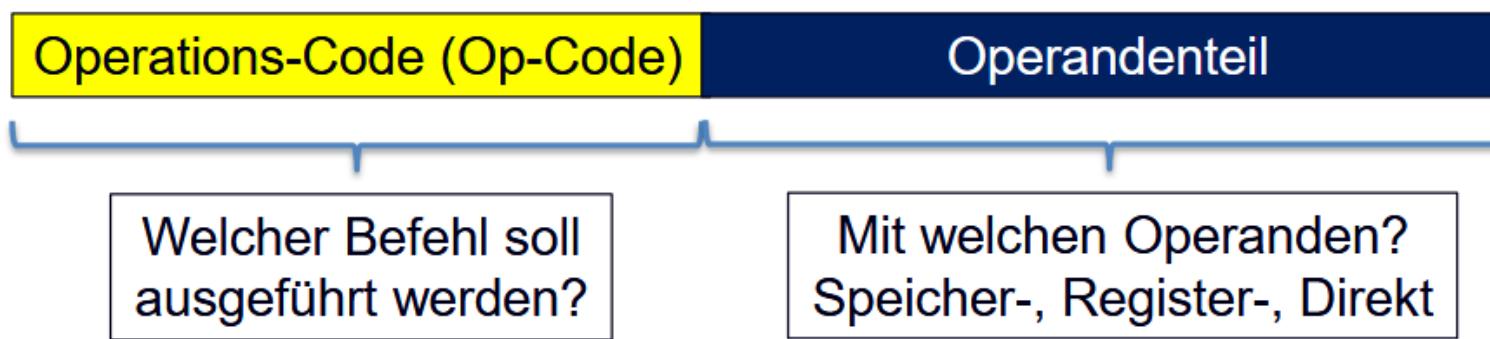
Befehlsworte

Maschinenbefehle (instructions) geben an:

- welche Operationen,
- mit welchen Operanden auszuführen sind,
- und wo das Ergebnis zu speichern ist.

Maschinenbefehle bestehen aus

- Befehlscode (Operations-Code)
- Operanden, Anzahl nach Rechnertyp verschieden



Bsp:



3-Adressmaschinen

Op-Code

Operand1, Operand2, Ergebnis

- Enthält 2 Operanden-Adressen und die Adresse für das Ergebnis
- Reihenfolge maschinenabhängig
- Ermöglicht kompakte Programme (weniger Quellcode)

Bsp:

add

reg1, reg2, reg3

entspricht: $\text{reg3} = \text{reg1} + \text{reg2}$



2-Adressmaschinen

Op-Code

Operand1, Operand2

- Verkürzung der Befehlslänge
- enthält 2 Operanden-Adressen,
die zweite Adresse ist meist auch die Zieladresse
=> Überdeckung: Adresse des 2. Operanden = Adresse des Ergebnisses

Bsp:

add

reg1, reg2

entspricht: $reg2 = reg1 + reg2$



1-Adressmaschinen



- Implizite Vereinbarung wo ein Operand zu finden ist
- Überdeckung

Bsp:

LDA	reg1
ADD	reg2
STORE	reg3

entspricht: $ACC = reg1$

$ACC = ACC + reg2 = reg1 + reg2$

$reg3 = ACC = reg1 + reg2$



0-Adressmaschinen

- Befehle enthalten keine Adressen
- **push** enthält Operand (aber keine Adresse)
- Keller- bzw. Stackmaschine (LiFo)
- Es gibt nur eine fest Adresse:
Stackpointer zeigt auf letztes Element auf dem Stack
- Postix-Notation (umgekehrte polnische Notation):
zunächst die Operanden eingegeben,
danach darauf anzuwendenden Operator



0-Adressmaschinen

- Befehle enthalten keine Adressen
- **push** enthält Operand (aber keine Adresse) 
- Keller- bzw. Stackmaschine (LiFo)
- Es gibt nur eine fest Adresse:
Stackpointer zeigt auf letztes Element auf dem Stack
- Postix-Notation (umgekehrte polnische Notation):
zunächst die Operanden eingegeben,
danach darauf anzuwendenden Operator

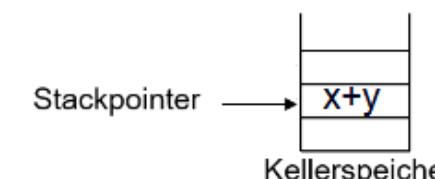
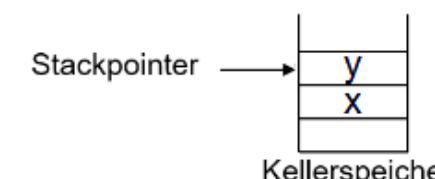
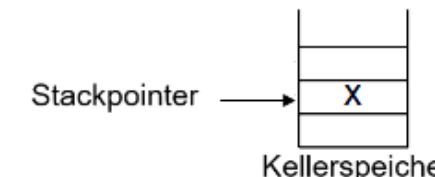
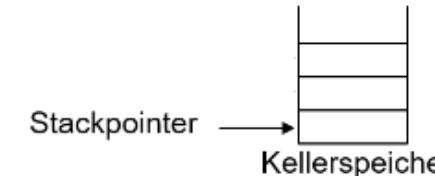
Beispiel:

1. Startpunkt:

2. push x

3. push y

4. add



- x und y vom Stapel geholt (pop),
dh. vom Stapel entfernt
- dann neues oberstes Element auf den Stapel gelegt:
- x+y

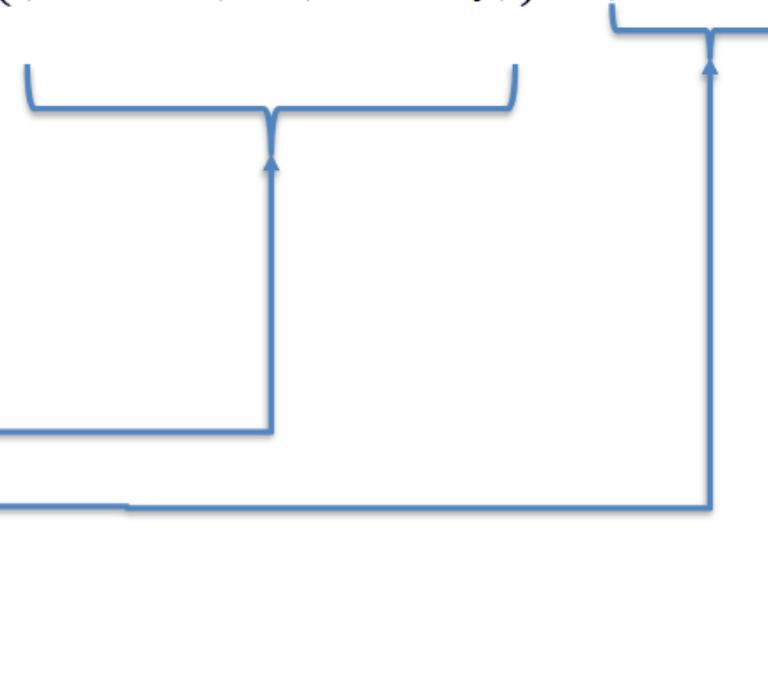


Beispielprogramm auf x-Adressmaschinen

Berechnung des Ausdrucks $z = ((a - b \times c) \times (d + e \div f)) \div (a + b \times c)$

3-Address

\times	b	c	h1
$-$	a	h1	h2
\div	e	f	h3
$+$	d	h3	h3
\times	h2	h3	h2
$+$	a	h1	h1
\div	h2	h1	z



Ergebnis



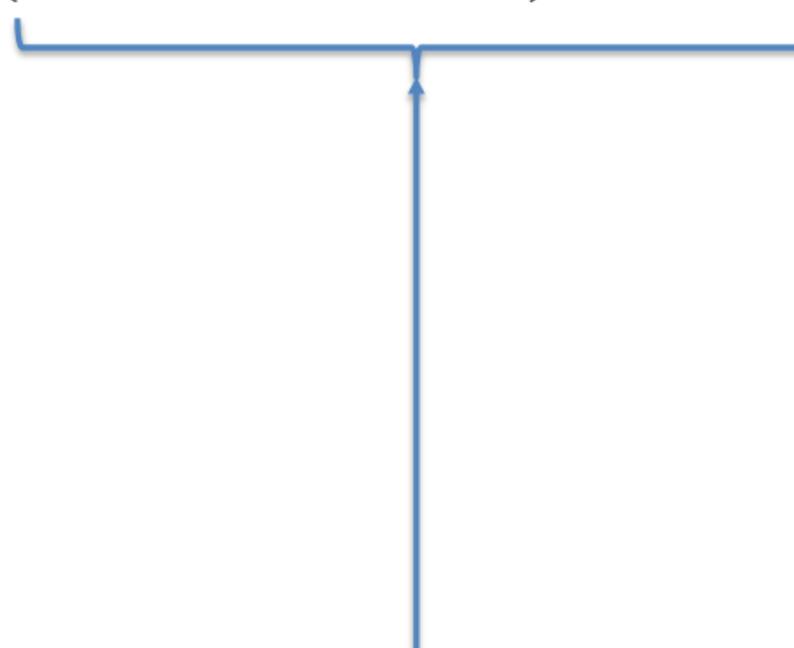
Beispielprogramm auf x-Adressmaschinen

Berechnung des Ausdrucks $z = ((a - b \times c) \times (d + e \div f)) \div (a + b \times c)$

2-Address

```
x  b  c    // c = b × c
→  c  h1   // h1 = c
-  a  c    // c = a - c
/  e  f    // f = e / f
+  d  f    // f = d + f
×  c  f    // f = c × f
+  a  h1   // h1 = a + h1
/  f  h1   // h1 = f / h1
→  h1 z    // z = h1
```

Ergebnis



Beispielprogramm auf x-Adressmaschinen

Berechnung des Ausdrucks $z = ((a - b \times c) \times (d + e \div f)) \div (a + b \times c)$

1 $\frac{1}{2}$ -Adress

```
→  c    r1  // r1 = c
×  b    r1  // r1 = b × r1
→  r1   r2  // r2 = r1
-  a    r2  // r2 = a - r2
→  f    r3  // r3 = f
/  e    r3  // r3 = e / r3
+  d    r3  // r3 = d + r3
×  r2   r3  // r3 = r2 × r3
+  a    r1  // r1 = a + r1
/  r3   r1  // r1 = r3 / r1
→  r1   z   // z = r1
```



Beispielprogramm auf x-Adressmaschinen

Berechnung des Ausdrucks $z = ((a - b \times c) \times (d + e \div f)) \div (a + b \times c)$

1-Address

```
load  b      // ACC = b
      ×  c      // ACC = ACC × c
store h1    // h1   = ACC
load  a      // ACC = a
      -  h1    // ACC = ACC - h1
store h2    // h2   = ACC
load  e      // ACC = e
      /  f      // ACC = ACC / f
      +  d      // ACC = ACC + d
      ×  h2    // ACC = ACC × h2
store h2    // h2   = ACC
load  a      // ACC = a
      +  h1    // ACC = ACC + h1
store h1    // h1   = ACC
load  h2    // ACC = h2
      /  h1    // ACC = ACC / h1
store z      // z    = ACC
```

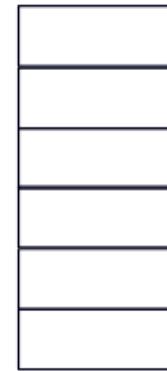


Beispielprogramm auf x-Adressmaschinen

Berechnung des Ausdrucks $z = ((a - b \times c) \times (d + e \div f)) \div (a + b \times c)$

0-Address

```
push  a
push  b
push  c
    × // c × b
    - // a-(c × b)
push  d
push  e
push  f
    /
    +
    ×
push  a
push  b
push  c
    × // c × b
    +
    /
pop   z  // z=((a - b × c) × (d + e ÷ f)) ÷ (a + b × c)
```



Stackpointer →

Beobachtung:
Ausdruck in Postfix-Notation:
 $abc^* - def / + ^ abc^* + /$
entspricht
0-Adress-Programm



Beispielprogramm auf x-Adressmaschinen

Berechnung des Ausdrucks $z = ((a - b \times c) \times (d + e \div f)) \div (a + b \times c)$

3-Adress	2-Adress	1 $\frac{1}{2}$ -Adress	1-Adress	0-Adress
$\times b \quad c \quad h1$	$\times \quad b \quad c$	$\rightarrow \quad c \quad r1$	<i>load b</i>	<i>push a</i>
$- a \quad h1 \quad h2$	$\rightarrow \quad c \quad h1$	$\times \quad b \quad r1$	$\times \quad c$	<i>push b</i>
$\div e \quad f \quad h3$	$- \quad a \quad c$	$\rightarrow \quad r1 \quad r2$	<i>store h1</i>	<i>push c</i>
$+ d \quad h3 \quad h3$	$/ \quad e \quad f$	$- \quad a \quad r2$	<i>load a</i>	\times
$\times \quad h2 \quad h3 \quad h2$	$+ \quad d \quad f$	$\rightarrow \quad f \quad r3$	$- \quad h1$	$-$
$+ a \quad h1 \quad h1$	$\times \quad c \quad f$	$/ \quad e \quad r3$	<i>store h2</i>	<i>push d</i>
$\div \quad h2 \quad h1 \quad z$	$+ \quad a \quad h1$	$+ \quad d \quad r3$	<i>load e</i>	<i>push e</i>
	$/ \quad f \quad h1$	$\times \quad r2 \quad r3$	$/ \quad f$	<i>push f</i>
	$\rightarrow \quad h1 \quad z$	$+ \quad a \quad r1$	$+ \quad d$	$/$
		$/ \quad r3 \quad r1$	$\times \quad h2$	$+$
		$\rightarrow \quad r1 \quad z$	<i>store h2</i>	\times
			<i>load a</i>	<i>push a</i>
			$+ \quad h1$	<i>push b</i>
			<i>store h1</i>	<i>push c</i>
			<i>load h2</i>	\times
			$/ \quad h1$	$+$
			<i>store z</i>	$/$
a, b, c, d, e, f, h1, h2, h3, z	a, b, c, d, e, f, h1, z	a, b, c, d, e, f, r1, r2, r3, z	a, b, c, d, e, f, h1, h2, z	pop z a, b, c, d, e, f, z



Übung : x-Adressmaschinen

gegebener Ausdruck $a = ((b + c) \times d - e)$

Aufgabe:

Schreibe für den gegebenen Ausdruck **a**

je ein Programm für eine 3-, 2-, 1-, und 0- Adressmaschine

3-Adress

add b c h1
mul h1 d h2
sub h2 e a

2-Adress

add b c // $c = b + c$
mul c d // $d = c \times d$
sub d e // $d = d - e$
Beobachtung:
 $mul\ a\ d\ //\ a = a \times d$
oder
 $mul\ a\ d\ //\ d = a \times d$
=> Festlegung notwendig.

1-Adress

load b // ACC = b
add c // ACC = ACC + c
mul d // ACC = ACC \times d
sub e // ACC = ACC - e
store a // a = ACC

0-Adress

push b
push c
add
push d
mul
push e
sub
pop a

Alternativ:

Alternativ:

Deshalb: Kennzeichnung durch Kommentar!

add b c c // $c = b + c$
mul c d d // $d = c \times d$
sub d e a
mul a d // d = a \times d
sub a e // $a = a - e$

oder auch (Unterstrich):

mul a d



Maschinenbefehlssatz- Architekturen

Unterscheidung nach Art der Operanden-Adressen

- **Stack-Architekturen bzw. Keller-Architekturen**

- z.B. Taschenrechner mit umgekehrt polnischer Notation
- von der Java Virtual Machine hergeleitete Java-Prozessoren

- **Akkumulator-Architekturen**

- - Die ersten 8-Bit-Mikroprozessoren hatten eine Akkumulator-Architektur
- - Bsp: MOS Technology 6502 (auch weitere Operationen)

- **Register-Register Architekturen**

- Zulässige Operanden nur Register
- RISC bzw. Load/Store Architekturen
- Bsp: Sun SPARC, MIPS R1x000, IBM POWERx, ARM

- **Register-Speicher Architekturen**

- Zulässige Operanden Speicheradressen und Register
- CISC
- Bsp.: Intel IA32, Motorola 680x0

- **Speicher-Speicher Architekturen**

- Wenn Operanden nur Speicheradressen: - Kaum verbreitet

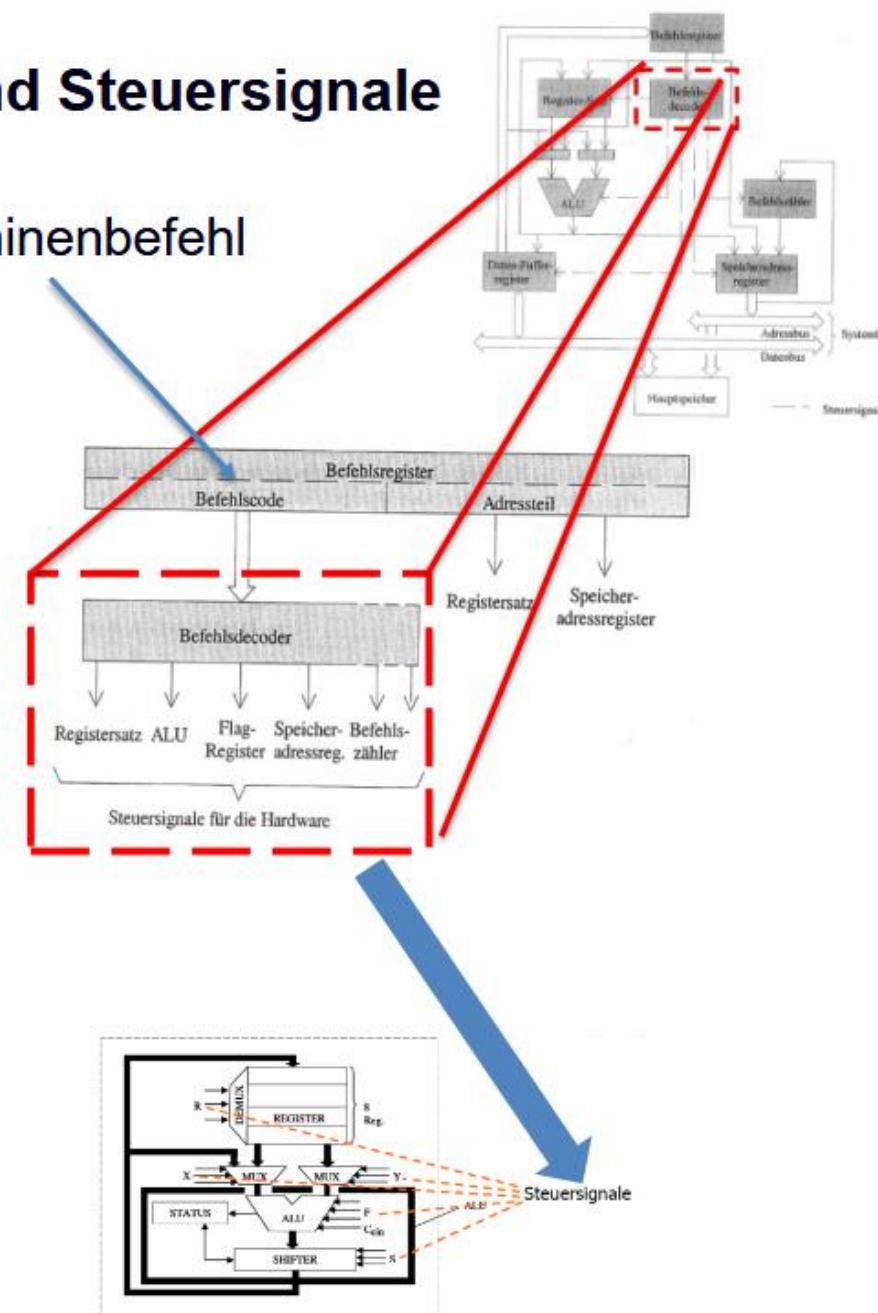


Zusammenhang Maschinenbefehle und Steuersignale

Aufgabe des Befehlsdecoder:

- Teil des Steuer-/Leitwerks
- empfängt Befehlscode aus Befehlsregister, „versteht“, und:
 - erzeugt zum richtigen Zeitpunkt die entsprechenden Steuersignale an den beteiligten Komponenten
- Hinweis: An dieser Stelle ist der Befehl bereits aus dem Speicher geholt und ins BR übertragen, der IP erhöht und der Befehl der Dekodierlogik übergeben worden.

Maschinenbefehl



Beispiel:

- Motorola 68060: ca. 190 Steuersignale



Realisierungsmöglichkeiten von Steuerwerken

- Abarbeitung von Maschinenbefehlen besteht i.d.R. aus Generierung einer Folge von Steuerworten (enthalten Steuersignale).
- Die Werte der Steuersignale sind abhängig von ausgeführter Instruktion und aktueller Phase.

Realisierungsmöglichkeiten der Ablaufsteuerung (im Steuerwerk = Control Unit):

1. Festverdrahtet:

Eine Hardware-Schaltung teilt den Befehlsablauf in mehrere Schritte auf und erzeugt im richtigen Moment die entsprechenden Steuersignale
(Synchrone Schaltwerk, Takt, Gatter,...)

Alles in Hardware, nicht änderbar.

2. Mikroprogramme bzw. μ -Programme (Firmware):

Maschinenbefehle werden als Folge von μ -Befehlen (Programme) beschrieben.
Ablage in Festwertspeicher wie EPROM, seltener RAM (ermöglicht Änderung)
=> Verschiebung des Entwurfs von Hard- auf Softwareebene



Realisierungsmöglichkeiten von Steuerwerken

	Festverdrahtete Steuerung	μ -programmierte Steuerung
Repräsentation	Endlicher Automat, Zustandsübergangsdiagramm	μ -Programm mit μ -Befehlen, repräsentiert Zustand
Ablaufsteuerung (sequencing control)	Explizite Funktion zur Bestimmung des Folgezustandes	μ -Programmzähler
Logische Darstellung	Logische (Boole'sche) Gleichungen	Wahrheitstabellen
Implementierungs- technik	Gatter, PLA	(EP)ROM, R/W-Speicher
Vorteile	Aufbau aus Gattern, schnell, FSM	<ul style="list-style-type: none">Flexibel: Implementierung neuer Maschinenbefehle durch Austausch von μ- Programmen („Firmware-update“)Emulation verschiedener Maschinensprachen auf einer CPU
Nachteile	Unflexibel	Langsame Ausführung: pro Makrobefehl viele μ -Befehle



Implementierung von Steuerwerken

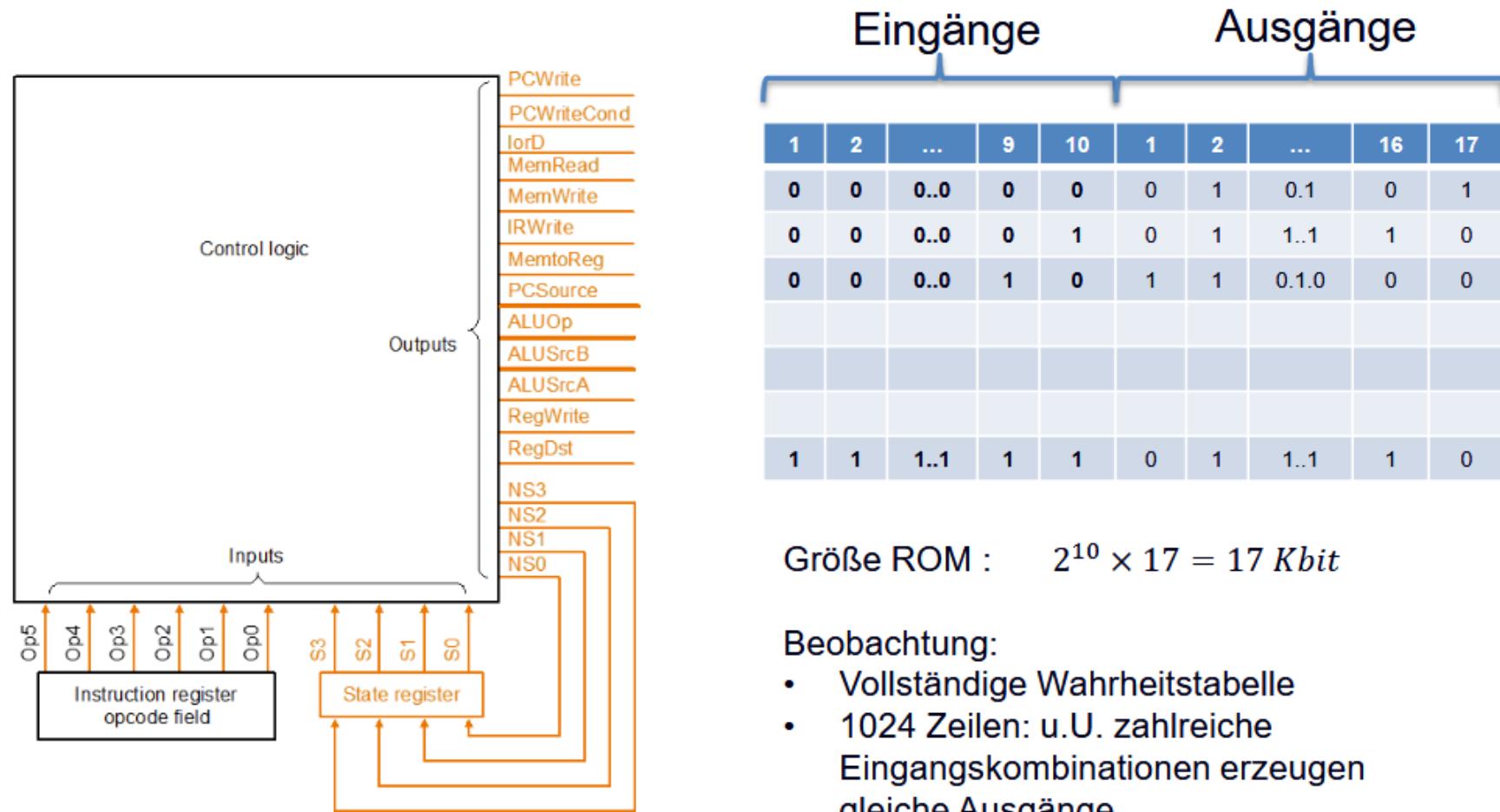
μ -programmierte Steuerwerke

- Festwertspeicher = Control Memory (CM) bzw. Control Store (ROM)
- pro **Makro-Befehl** (= Maschinenbefehl) liegt im Festwertspeicher des Steuerwerks ein μ -Programm vor.
- μ -Programm besteht aus μ -Befehlen/ **micro-instructions**/ **nano-instructions**
- μ -Programm-Sequenzer im Steuerwerk:
 - ist quasi ein „kleiner Prozessor innerhalb CPU“
 - führt μ -Programme aus
 - beherrscht wesentlich weniger Befehle als CPU (in der er sitzt), und ist dadurch einfacher zu realisieren
 - einzelne μ -Instruktionen werden von μ -Befehlszähler des Sequenzers adressiert
- Jedes Feld eines μ -Befehls steuert Funktion im Datenpfad oder bedingt Verzweigung im Steuerwerk: dadurch kann Sequenzer komplexe Folgen von Steuersignalen erzeugen
- Jeder Makrobefehl, den CPU ausführt, bedingt Ausführung eines kompletten μ -Programms durch Sequenzer
- ...



Implementierung von Steuerwerken

mit ROM (Beispiel)



Größe ROM : $2^{10} \times 17 = 17 \text{ Kbit}$

Beobachtung:

- Vollständige Wahrheitstabelle
- 1024 Zeilen: u.U. zahlreiche Eingangskombinationen erzeugen gleiche Ausgänge

Alternative:

- Aufteilung der Wahrheitstabelle



Implementierung von Steuerwerken mit Microcode-Sequenzer (μ -Sequencer)

Beobachtung:

- komplexe Instruktionssätzen (ISA Instruction Set Architecture)
=> viele unterschiedliche mögliche (Folge-)Zustände!
- Bestimmung von Folgezustand,
bei vielen möglichen Folgezuständen: Aufwand!

Mögliche Lösungen:

- mittels PLA oder ROM.

Bei vielen mögl. Folgezuständen: Aufwand!

- mittels μ -Sequencer

Warum?

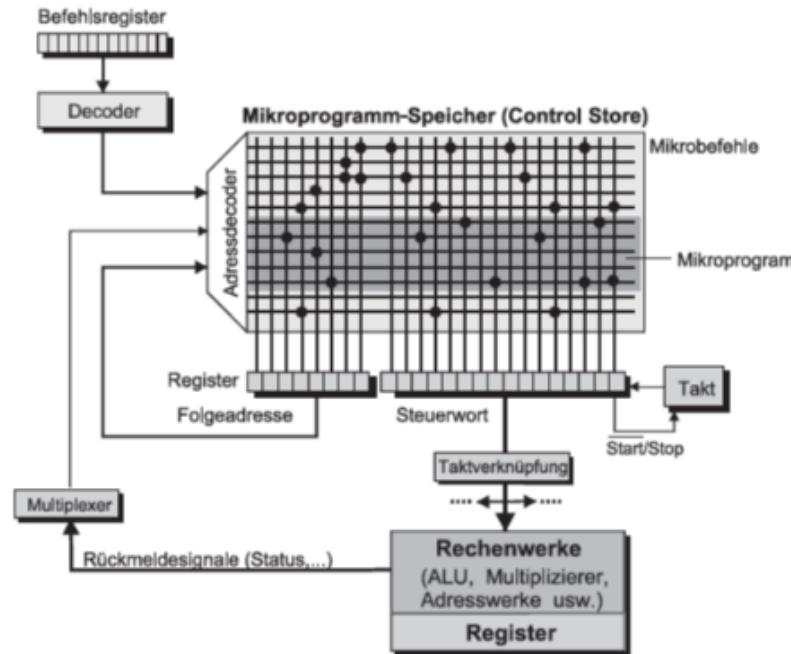
*Ist effizienter wenn es viele Folgezustände gibt,
und es auch viele Folgezustände gibt die nur einen weiteren möglichen
Folgezustand haben (was oft der Fall ist)*



Implementierung von Steuerwerken

Aufbau μ -programmierte Steuerwerke

Decoder bestimmt aus Opcode (im Befehlsregister) Startadresse des μ -Programms



Exemplarischer Aufbau eines μ -Befehls:

Folgeadresse	Steuerwort						
	Steuerwerk BR SR	Adresswerk PC AP	Rechenwerk StR	Register- satz	System- bus	externe Steuersignale	



Praxisbeispiel: Microcode

x86 Microcode update

Warum sind Microcode updates wichtig?

Einstieg:

<https://support.microsoft.com/de-de/topic/kb4093836-zusammenfassung-der-intel-microcode-updates-08c99af2-075a-4e16-1ef1-5f6e4d8637c4>

Beispiel Konkret: Spectre

<https://support.microsoft.com/de-de/topic/kb4100347-microcodeupdates-von-intel-33551904-eecd-cf4d-12a6-f5fb8d735fc6>

[https://de.wikipedia.org/wiki/Spectre_\(Sicherheitsl%C3%BCcke\)](https://de.wikipedia.org/wiki/Spectre_(Sicherheitsl%C3%BCcke))

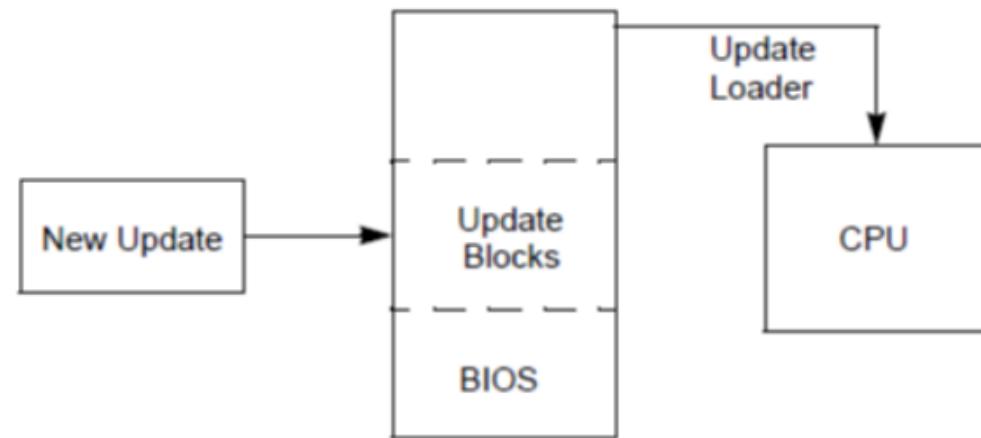
[https://de.wikipedia.org/wiki/Meltdown_\(Sicherheitsl%C3%BCcke\)](https://de.wikipedia.org/wiki/Meltdown_(Sicherheitsl%C3%BCcke))

<https://www.cloudflare.com/de-de/learning/security/threats/meltdown-spectre/>



Praxisbeispiel: Microcode

Wie funktionieren Microcode updates? (Firmware updates)



Befehlsausführung

Prinzip

Beobachtung:

Eine CPU führt jeden Befehl in einer Reihe kleiner Schritte aus.

Prinzipieller Ablauf: Zyklus:

Abrufen – Decodieren – Ausführen

1. nächsten Befehl aus dem Speicher in das Befehlsregister abrufen.
2. Programmzähler ändern (zeigt auf nächsten Befehl).
3. Typ des gerade eingelesenen Befehls bestimmen.
4. Falls der Befehl ein oder mehrere Speicherworte benötigt, dessen (deren) Position bestimmen.
5. Wort(e) bei Bedarf in ein CPU-Register laden.
6. Befehl ausführen.
7. Zu Schritt 1 gehen: nächster Befehl.



Complex Instruction Set Computing (CISC)

Eigenschaften

- Große Anzahl von Befehlen und Adressierungsarten
- verschiedene Instruktionsformate
- Eher kleinerer Registerkörper (nicht „so viele“ Register)
- z.T. orthogonaler Befehlssatz (Befehlsraum)
 - Befehlssatz heißt orthogonal, wenn Befehlscode, Adressierungsart und Datentyp (+ manchmal auch Anzahl der Operanden) unabhängig voneinander beliebig kombiniert werden können.
 - Orthogonalität vereinfacht Nutzbarkeit der zur Verfügung stehenden Befehle, führt aber zu großen Befehlssätzen, wenn viele Adressierungsarten und Datentypen (CICS Mainframes > 1000 Befehle)



Entwicklung CISC am Beispiel IA-32

Architekturmerkmale Intel Architecture 32 (Bit), IA-32

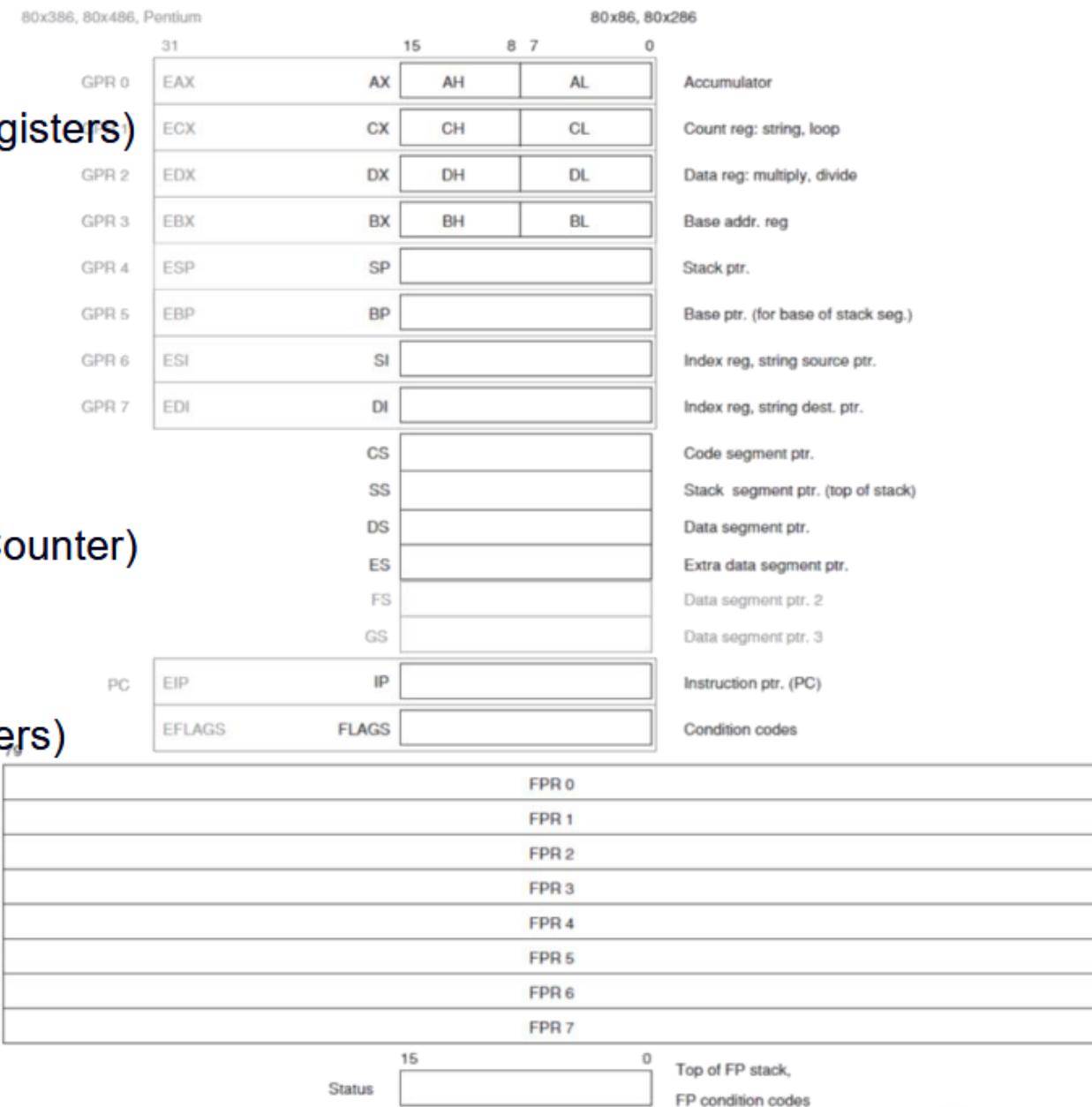
- Weiterentwicklung der 16-Bit-Architekturen von Intels 8086- und 80286-Prozessoren.
- Alle Register, einschließlich der Adressregister, wurden in dieser Architektur auf 32 Bits erweitert.
- Die Anzahl der Register blieb gleich.
- Die Mnemonic der erweiterten Register wurden mit einem vorangestellten E, für extended (deutsch: erweitert), gekennzeichnet:
 - z.B.: EAX (32-Bit) von zuvor AX (16-Bit)
- Abwärtskompatibilität:
 - 32-Bit-Register als Erweiterung der 16-Bit-Register der 80286-Architektur
 - Mit Bezeichnungen für 16-Bit-Register : Zugriff auf die unteren 16-Bit der 32-Bit-Register z.B.: AX liefert die unteren 16-Bit des 32-Bit-EAX-Register
- Der Adressbus ist 32 Bits breit und demzufolge ist die Adressierbarkeit auf 4 GiB (physischer Adressraum) begrenzt.
- Ab Pentium Pro: PAE (physical address extension) => Adresserweiterung auf 36 Bits
Mit 36 Bits lassen sich 64 GiB adressieren (=> PAE muss aber vom Betriebssystem unterstützt werden!)



Entwicklung CISC am Beispiel IA-32

Register der IA-32 Architektur

- Acht GPR (General Purpose Registers)
32-bit
- Sechs Segment-Registers
16-bit
 - Memory ist segmentiert.
Segment-Register zeigen auf Segmente
- Instruction Pointer (Programm Counter)
- EFLAGS
(Bsp: oVerflow-bit)
- Acht FPR (Floating Point Registers)
80-bit



Basis für
Assembler-Programmierung
in diesem Kurs

<https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>



IA-32-64 Register

“x64” : bezeichnet die 64-bit Extension der Intel (und AMD) 32-bit x86 instruction set architecture (ISA)

x86-Prozessoren sind rückwärts-kompatibel ausgelegt:

- x64-Prozessoren bieten 64-Bit-Betriebsmodus
- und ebenfalls den bisherigen 32-Bit-Modus
- sowie die ursprünglichen 16-Bit-Modus

=> Damit ergänzt x64 die Intel Architecture 32-Bit bzw. kurz IA-32 um einen 64-Bit-Betriebsmodus.

Änderungen (Bsp:) 16 GPR mit 64-bit

<https://de.wikipedia.org/wiki/X64>



Ideen für Reduced Instruction Set Computing (RISC)

Beobachtung: (bereits Mitte der 70er Jahre)

- Bei CISC belegen 20% der Instruktionen 80% der Ausführungszeit.
- Es gibt Beispiele, in denen eine Sequenz einfacher Instruktionen das Gleiche bewirkt wie eine komplexe Sequenz, jedoch eine geringere Ausführungszeit benötigt.
- Fortschritte in der Halbleitertechnologie absehbar, z.B. schnellere Arbeitsspeicher und Einführung von Cache-Speicher (SRAM).

Folgerung : Der prinzipielle Geschwindigkeitsvorteil des Mikroprogrammspeichers (ROM) gegenüber dem Arbeitsspeicher (RAM, mit jeweiligem Maschinenprogramm) reduziert sich.



Ideen für Reduced Instruction Set Computing (RISC)

Motivation für neue Prozessorstruktur: Leistungssteigerung!

Entwicklungen in Berkeley und Stanford:

- RISC: Reduced Instruction Set Computing (1980/81)
- Umgesetzt in der Firma MIPS
Microprocessor without Interlocked Pipeline Stages
in den MIPS R2000 / R3000 und nachfolgenden CPUs
(1984 Gründung von MIPS, 1994 Übernahme durch Silicon Graphics (SGI))
- Der Vollständigkeit halber: Bei IBM u.a. gab es entsprechende Überlegen früher.



RISC Merkmale

2 Kategorien konventioneller Maschinensprachen: CISC und RISC

Für RISC existiert keine Definition, sondern nur typische Merkmale:

- **Reduced:**
Reduzierung des Befehlssatzes auf einfache Befehle, aus denen sich komplexere Operationen zusammensetzen lassen.
- **Instruction:**
Load-Store-Architektur:
Die einzigen Befehle, die auf den Arbeitsspeicher zugreifen sind Load und Store, d.h. Register-Register Modell. Ergibt drastische Reduzierung des Steueraufwands gegenüber anderen Ausführungsmodellen (siehe z.B. Operandenteil von CISC).
- **Set:**
Fast alle Befehle können in einem (gleich langen) Maschinenzyklus abgearbeitet werden. Wichtige Grundlage für die Reorganisation der Befehlsabläufe gegenüber optimierenden Compilern.
- **Computing:**
Pipelining als eine Möglichkeit, die Anzahl der notwendigen Maschinenzyklen für die Ausführung eines Befehls zu reduzieren, ist die Überlappung der Ausführung von Befehlen.



RISC-Design-Prinzipien

- Hardware führt alle Befehle direkt aus (kein Interpreter)
- Befehle leicht zu decodieren
- Nur Lade- und Speichervorgänge auf dem Speicher
- Logische oder Arithmetische Verknüpfungen nur in Registern
- Viele Register notwendig („Zwischenspeicherung“)
- Befehle werden mit maximaler Rate initiiert:
 - Out of order execution (wenn gerade Ressource nicht zur Verfügung)



RISC contra CISC

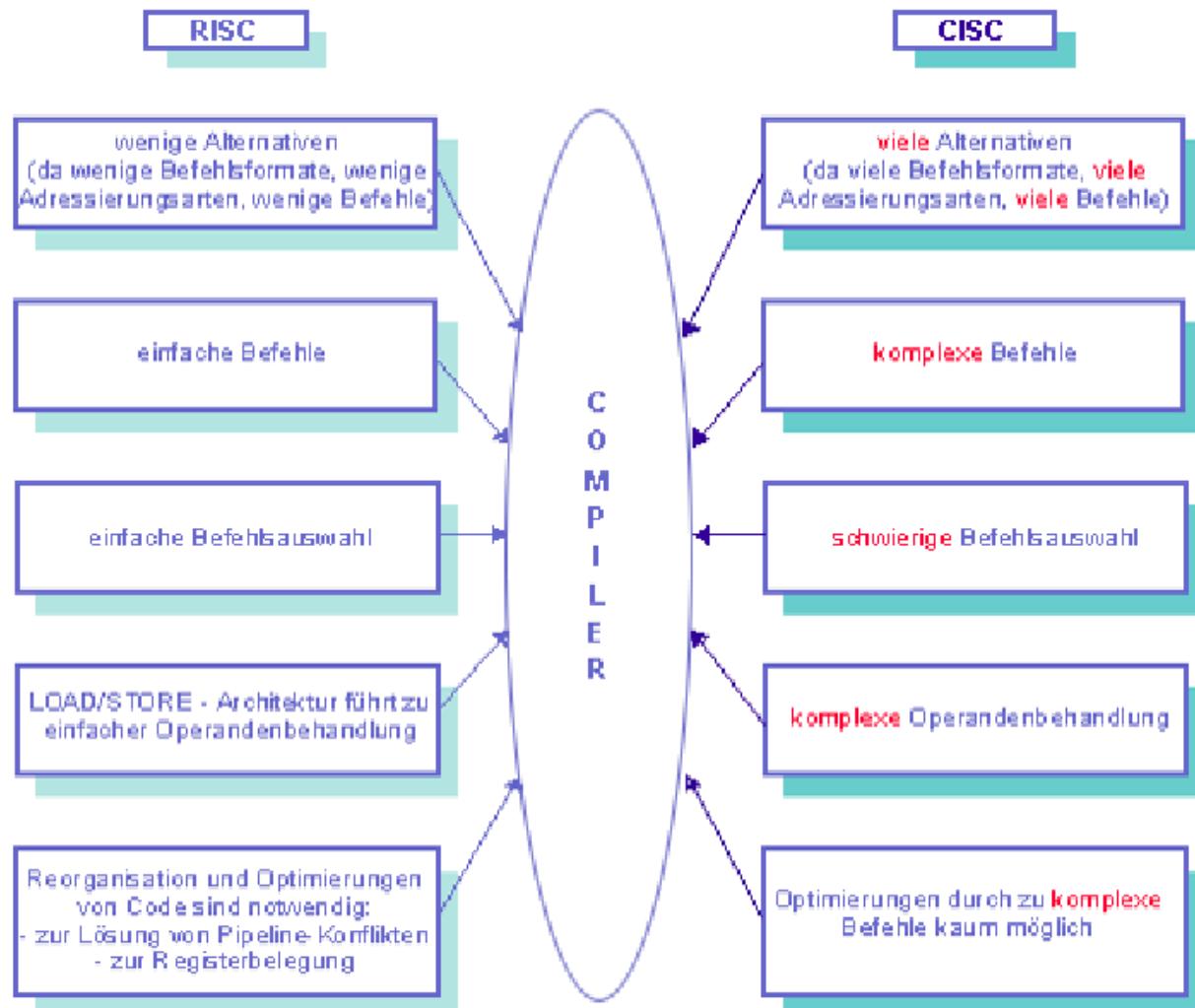
Eigenschaften RISC

- Befehlssatz:
 - geringe Zahl an Befehlen: nur unbedingt notwendige Befehle (weniger als 128)
 - einheitliches Befehlsformat: (typisch: 3- Adress-Instruktionen)
- Dadurch wird Steuerwerk vereinfacht!
- Typisch: viele universelle Register (mind. 32 GPRs)
- Der Zugriff auf Speicher nur über Lade-/Speicherbefehle
=> „Load / Store Architektur“.
- ALLE Befehle (außer Load/ Store) beziehen Operanden NUR aus Registern.
=> Register-Register-Architektur (heute Standard in allen modernen Architekturen)
- einfache Befehle: in einem Zyklus ausführbar:
zwei Register werden „irgendwie“ verknüpft,
Ergebnis wird wieder in ein Register geschrieben.
- Keine Mikroprogrammierung
- schnell
- Vergleich: Multiplikation von 2 Zahlen:
 - CISC: ein Befehl – dieser geht über viele Taktzyklen, am Ende steht das Ergebnis
 - RISC: viele schnelle einfache Befehle, nacheinander.
=> schneller.



Abschließender Vergleich

RISC / CISC



Heute: Hybrider Ansatz:

RISC-Ideen in CISC integriert.

- Pentium-Prozessoren (eigentlich CISC):
weisen intern RISC-ähnliche Architektur auf
- µ-Sequenzer sorgt für Abbildung von CISC Befehlssatz auf RISC-Befehle des Prozessorkerns
- Intel-RISC-Kern beruht auf typischen RISC-Merkmalen wie
 - Befehls-Pipelining
 - Überlappende Ausführung von Befehlen: Befehl holen, Befehl dekodieren, Operand holen, Operand verarbeiten, Ergebnis abspeichern
 - Superpipelining
 - Noch feinere Stufen als beim Befehls-Pipelining
 - Superskalarität
 - Pro Zyklus mehrere Befehle laden
 - Out-of-order execution
 - Von Neumann: in-order-execution von Befehlen
(in der Reihenfolge wie sie im Programm stehen)
 - Out-of-order execution von Befehlen:
Maschinenbefehle in einer anderen Reihenfolge auszuführen, ohne das Ergebnis zu verändern.



Heute: Hybrider Ansatz:

RISC-Ideen in CISC integriert.

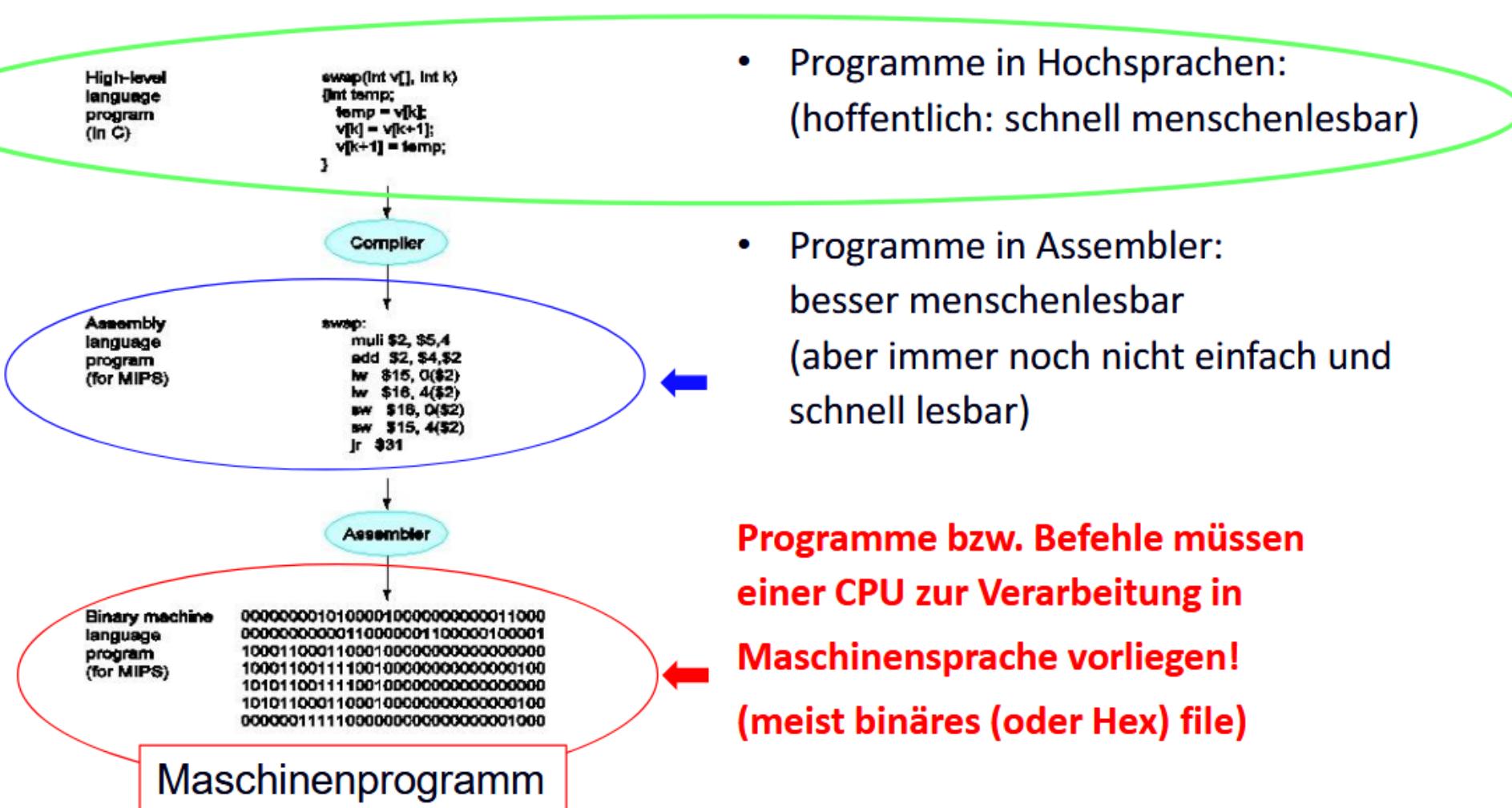
Bemerkung:

- Der hybride Ansatz ist langsamer als ein reiner RISC-Entwurf
 - Aber auch schneller als ein reiner CISC-Entwurf
- ⇒ Aufgrund des CISC-Entwurfs (z.B. Intel): KOMPATIBILITÄT!!!!!!
- ⇒ Deshalb Siegeszug des hybriden Ansatzes:
 - ⇒ Einigermaßen konkurrenzfähige Gesamtpfomance
 - ⇒ UND Abwärtskompatibilität (== Investitionen in Software können weiter genutzt werden)



Maschinensprache und Assembler

The big picture



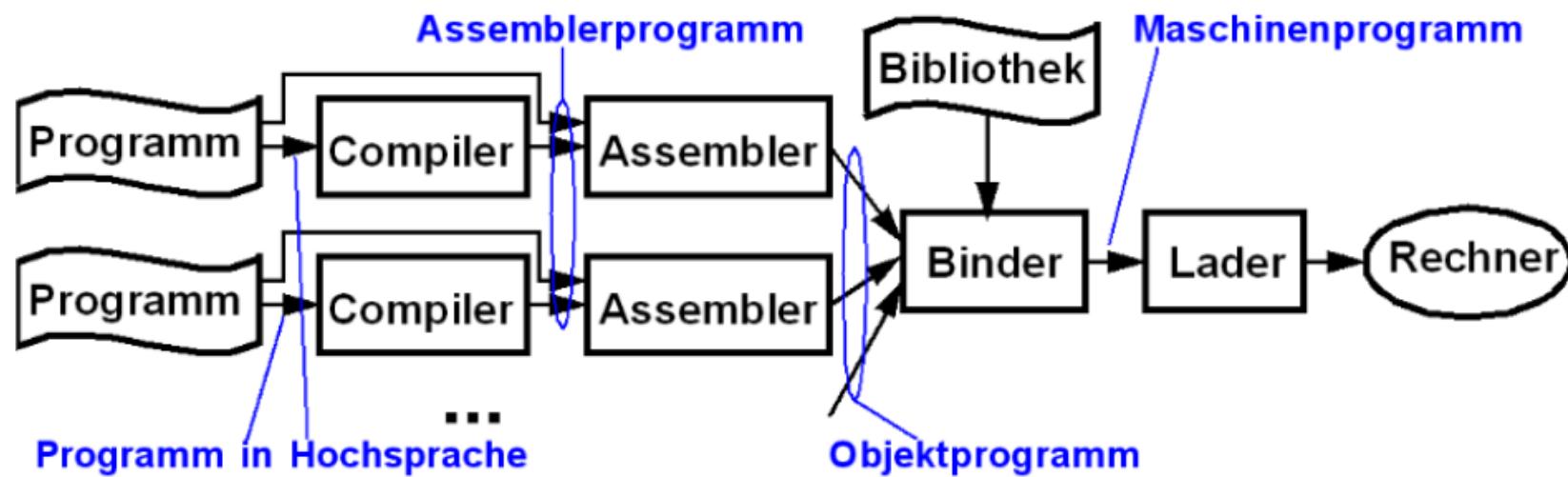
Bem: Vereinfachte Sichtweise
(Binder fehlt)



Hochsprache, ObjektCode, Maschinensprache

compiler, assembler, linker, loader

1. Programm in Hochsprache wird vom **Compiler** in ein Programm in Assemblersprache übersetzt
2. **Assembler** übersetzt in ein Objektprogramm
3. **Linker (Binder)** übersetzt Objektprogramme in ein Maschinenprogramm
4. **Loader** lädt Maschinenprogramm in Speicher



Assembler übersetzt Assemblerprogramm in ein Maschinenprogramm oder Objektprogramm.

Ein Assemblerprogramm enthält:

- symbolische Operationscodes
- symbolische Registernamen
- symbolische Marken (Kennzeichnung von Zeilen im Assemblerprogramm)
- Makros (Ersetzung oft vorkommender Programmteile)
- Kommentare

Darstellung eines Assemblerprogramms:

- Maschinenbefehle werden in definierter Schreibweise namens **Mnemonics** dargestellt
- Mnemotechnik: Mnemonics
 - Eingängige Abkürzungen als symbolische Darstellung von Maschinenbefehlen
 - vom Menschen leichter zu interpretieren, und zu merken als Maschinenbefehle in binärem oder Hex-Code.
- Programm aus Mnemonics wird als Assemblerprogramm bezeichnet
- Assembler setzt symbolische Maschinenbefehle in Objektcode um



Maschinensprache und Assembler

Mnemonics und Instruction Set Architecture (ISA)

Zur Befehlssatzarchitektur = Instruction Set Architecture (ISA) zählt:

- Maschinenbefehlssatz mit Art und Format der Befehle (z.B. CISC):
Definition einer Assemblersprache (Mnemonics)
- Art und Anzahl der adressierbaren Register des Rechnerkerns
incl. der Bitbreite der Daten-/ Adressregister und der Verarbeitungseinheiten
- Darstellungsmöglichkeiten für Daten (Maschinen-Datentypen)
z.B.: Byte, Integer-Zahlen, Fixkomma- und Floatingpointzahlen
- Adressierungsarten (Zugriff auf Speicher)
- Art der Dateneingabe und Datenausgabe



Assembler und Objektsprache

Aufbau eines Objektprogramms

- enthält Programmtext in Maschinensprache, sowie die zugehörigen binären Daten.
- enthält u.U. weitere Informationen, um mehrere Objektprogramme zu verbinden (linken)
z.B: Namen global definierter Unterprogramme

Objektprogramm Nr 1			
Object file header	Name	Procedure A	
	Text size	100 _{hex}	
	Data size	20 _{hex}	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(X)	
	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	—	
	B	—	
Objektprogramm Nr 2			
Object file header	Name	Procedure B	
	Text size	200 _{hex}	
	Data size	30 _{hex}	
Text segment	Address	Instruction	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(Y)	
	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	—	
	A	—	

Vgl. Thiele, ETH, Computer Engineering



Objektcode und Linker

Linker (Binder):

- fasst alle zu einem Maschinenprogramm gehörenden Teile (Objektcode) zusammen
- stellt sicher dass das Maschinenprogramm keine undefinierten Marken enthält
- löst Querbezüge zwischen Programmteilen auf
- findet in Bibliotheken alle vordefinierten Teilprogramme, die im Programm benutzt werden, und bindet diese in das Maschinenprogramm ein.
- Bestimmt Speicherbereiche, die durch einzelne Programmteile belegt werden und verschiebt die Anweisungen



Assembler-Befehle

- sind prozessorabhängig
- finden sich im Instruction Set
jedes Prozessorhandbuchs

Befehle und Pseudobefehle sind i.d.R. auch assemblerabhängig

- finden sich in Beschreibung des benutzen Assemblers (-Programms)
- Für IA32: nasm, masm, tasm
- Beispiel: Microsoft (masm) und AT&T (GCC inline Assembler) Syntax ist unterschiedlich

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

https://en.wikipedia.org/wiki/Microsoft_Macro_Assembler

<https://learn.microsoft.com/en-us/cpp/assembler/masm/microsoft-macro-assembler-reference>

https://en.wikipedia.org/wiki/Turbo_Assembler

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



Kategorien von Maschinen-/Assemblerbefehlen

(analog zu den meisten Instruction Set Summaries)

1. Arithmetische und logische Befehle

- Arithmetisch: ADD, ADC, SUB, SBC, DIV, MUL, DEC, INC
- Logisch: AND, OR, EOR, NOT, COM

2. Verzweigungen (Branch Instructions)

- Unbedingte Sprungbefehle: JMP, BRA, IRET, CALL
- Bedingte Sprungbefehle, abhängig vom Ergebnis der Vergleichsoperationen:
BREQ, JNZ

3. Speicherzugriffe (Data Transfer Instructions)

- Datentransfers mit Register, Arbeitsspeicher: LDA, mov, LoaD, STore, PUSH, POP, xchg
- Datentransfers mit Peripherie: IN, OUT, aber auch mov, ggf. READ, WRITE
- Stringmanipulationsbefehle: movs, lods, stos, cmps

4. Bit und Bit-Test Instructions (Flagcontrol)

- Schiebeoperationen: ASR, ASL, LSL, LSR, ROL, ROR
- Bit Set Operationen: BTS, SBI, SEI / CLI,

5. Control Instructions

- NOP, LEA, WDR



IA-32 Befehlsformat

- Mnemonics sind zwar i.d.R. gleich oder zumindest ähnlich (ADD, ADC,...)
- gesamtes Befehlsformat unterschiedlich: abhängig vom Maschinentyp

2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

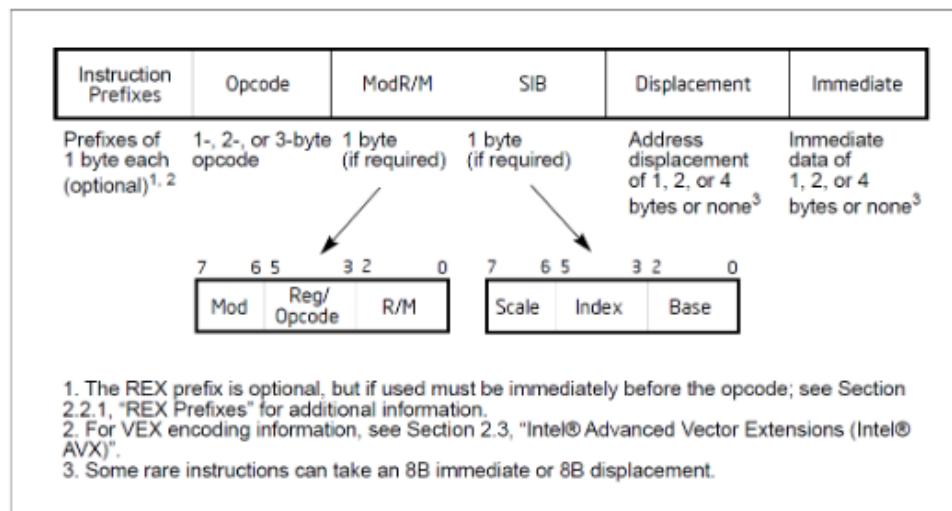


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Siehe:

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C, & 2D): Instruction Reference, A-Z



IA-32 Befehlsformat

Unser Beispielprogramm (Auszug)

Nr in Assembler Ausgabe	Beispielprogramm	ax	bx	A	B
	A dw 1			1	
1	B dw 2			1	2
2	mov ax, [B]	2		1	2
3	mov bx, ax	2	2	1	2
4	shl ax, 3	16	2	1	2
5	add ax, bx	18	2	1	2
6	dec ax	17	2	1	2
7	mov [B], ax	17	2	1	17
8	dec [B]	17	2	1	16
9	mov bx, [B]	17	16	1	16



IA-32 Befehlsformat

Unser Beispielprogramm (Aufgabe 1)

Nr in Assembler Ausgabe	Beispielprogramm	
	A	dw 1
1	B	dw 2
2	mov	ax, [B]
3	mov	bx, ax
4	shl	ax, 3
5	add	ax, bx
6	dec	ax
7	mov	[B], ax
8	dec	[B]
9	mov	bx, [B]

Restricted Mode is intended for safe code browsing. Trust this window to enable all features. [Manage](#) [Learn More](#)

```

beispielprogramm.asm
C:\Users\cb\Nextcloud\Rechnerarchitektur\06>0_Beispielprogramm-Vorlesung> beispielprogramm.asm
1 global main ;must be declared for using gcc
2 extern printf
3
4 section .data
5     A dw 1 ; 16 bit unsigned int: 1
6     B dw 2 ; 16 bit unsigned int: 2
7
8 section .text
9
10 > print_ALLES: ; Set up for calling _printf to print myVar's value...
11     ret
12
13     main: ; tell linker entry point
14         mov esi, 0 ; Initialisierung counter
15
16         mov eax, 0 ; Initialisierung: EAX (32 bit)= alle bit sind 0.
17         mov ebx, 0 ; Initialisierung: EBX (32 bit)= alle bit sind 0.
18
19         CALL print_ALLES ; (1) Prints all of interest
20
21         mov ax, [B] ; Lade Inhalt von B nach AX. In AX ist dann 2
22
23         CALL print_ALLES ; (2) Prints all of interest
24
25         mov bx, ax ; Lade Inhalt von AX nach BX. In BX ist dann auch 2.
26
27         CALL print_ALLES ; (3) Prints all of interest
28
29         shl ax, 3 ; Left-Shift von AX entspricht: 2*2*2*2 = 16. In AX steht dann: 16
30
31         CALL print_ALLES ; (4) Prints all of interest
32
33         add ax, bx ; Addiere BX (=2) auf AX (=16), entspricht: 16+2 = 18, In AX steht dann 18
34
35         CALL print_ALLES ; (5) Prints all of interest
36
37         dec ax ; Decrementiere AX um 1, entspricht 18-1 = 17. In AX steht dann 17.
38
39         CALL print_ALLES ; (6) Prints all of interest
40
41         mov [B], ax ; Lade AX an die Adresse auf die B zeigt in den Speicher.
42                     ; Die Speicherzelle mit der Adresse B enthält dann 17.
43
44         CALL print_ALLES ; (7) Prints all of interest
45
46         dec word[B] ; Decrementiere den Wert in Speicherzelle B um 1, entspricht 17-1 = 16.
47                     ; In Speicherzelle mit Adresse B steht dann der Wert 16
48
49         CALL print_ALLES ; (8) Prints all of interest
50
51         mov bx, [B] ; Lade Inhalt von B nach BX. In BX ist dann 16
52
53         CALL print_ALLES ; (9) Prints all of interest
54
55
56         ret ; Return from _main
57
58 section .data
59     formatString db '%u) EAX = %u | EBX = %u | A = %u | B = %u ', 10, 0 ; Format string
60     ;formatString db 'EAX = %d | EBX = %d', 10, 0 ; Format string for _printf
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88

```



IA-32 Basic Execution Environment

BASIC EXECUTION ENVIRONMENT

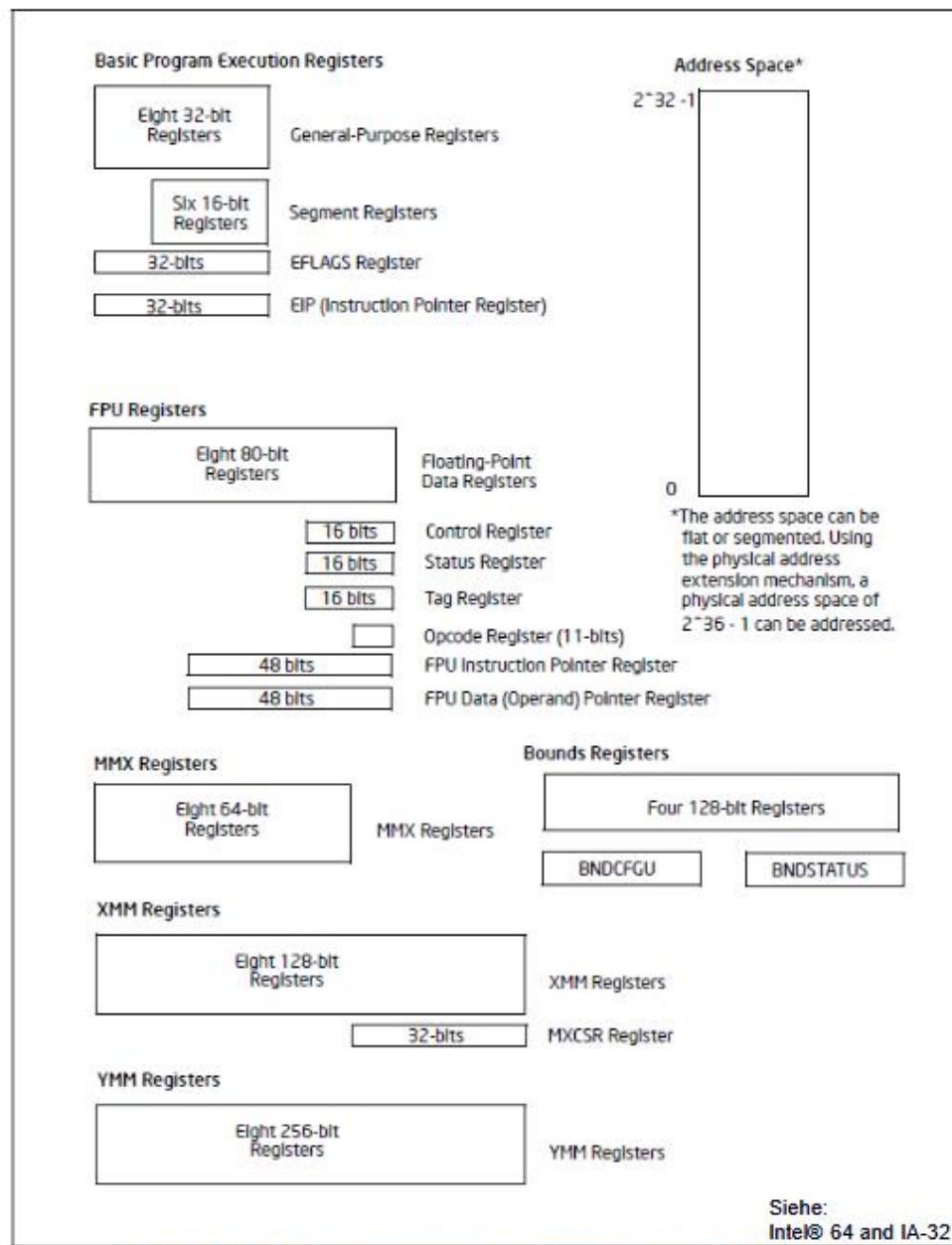


Figure 3-1. IA-32 Basic Execution Environment for Non-64-bit Modes



General Purpose Register (GPR)

3.4.1

General-Purpose Registers

The 32-bit general-purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

- Operands for logical and arithmetic operations
- Operands for address calculations
- Memory pointers

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for another purpose.

Many instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. When using a segmented memory model, some instructions assume that pointers in certain registers are relative to specific segments. For instance, some instructions assume that a pointer in the EBX register points to a memory location in the DS segment.

BASIC EXECUTION ENVIRONMENT

The special uses of general-purpose registers by instructions are described in Chapter 5, "Instruction Set Summary," in this volume. See also: Chapter 3, Chapter 4, Chapter 5, and Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*. The following is a summary of special uses:

- **EAX** — Accumulator for operands and results data
- **EBX** — Pointer to data in the DS segment
- **ECX** — Counter for string and loop operations
- **EDX** — I/O pointer
- **ESI** — Pointer to data in the segment pointed to by the DS register; source pointer for string operations
- **EDI** — Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
- **ESP** — Stack pointer (in the SS segment)
- **EBP** — Pointer to data on the stack (in the SS segment)

As shown in Figure 3-5, the lower 16 bits of the general-purpose registers map directly to the register set found in the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SI, DI, and SP. Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

General-Purpose Registers		
31	16	15
0	8	7
AH	AL	
BH	BL	
CH	CL	
DH	DL	
	BP	
	SI	
	DI	
	SP	

16-bit 32-bit

AX	EAX
BX	EBX
CX	ECX
DX	EDX
	EBP
	ESI
	EDI
	ESP

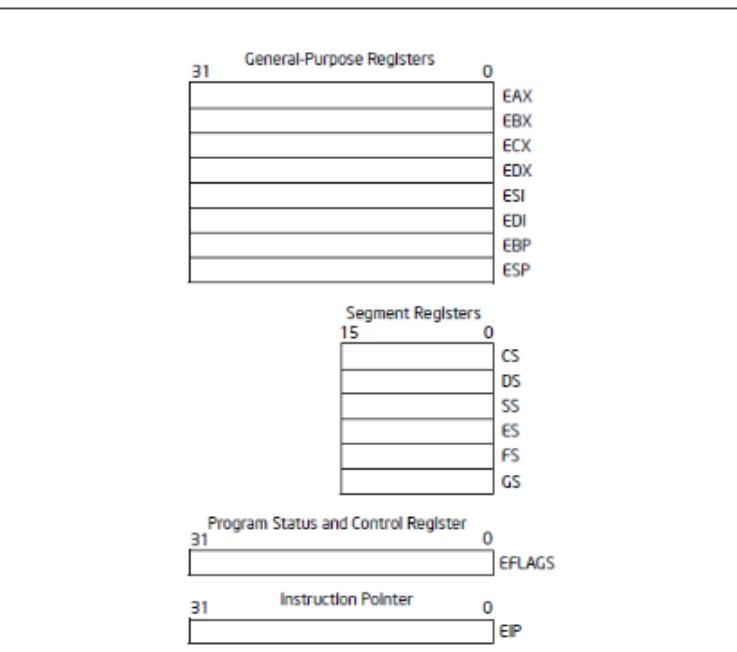


Figure 3-4. General System and Application Programming Registers

Siehe:
Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1 Basic Architecture

Vol. 1 3-11



IA-32 Speicher und Register

EFLAGS Register (status-, control-, system-flags)

BASIC EXECUTION ENVIRONMENT

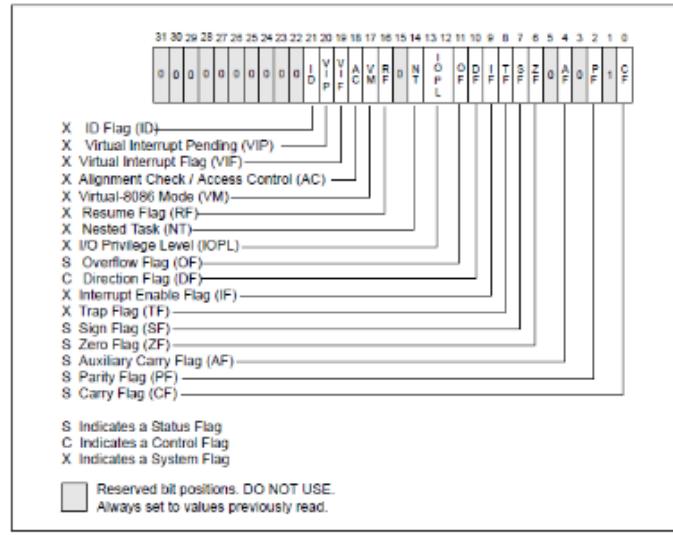


Figure 3-8. EFLAGS Register

As the IA-32 Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the IA-32 processors to the next. As a result, code that accesses or modifies these flags for one family of IA-32 processors works as expected when run on later families of processors.

3.4.3.1 Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

- | | |
|--------------------|---|
| CF (bit 0) | Carry flag — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic. |
| PF (bit 2) | Parity flag — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise. |
| AF (bit 4) | Auxiliary Carry flag — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic. |
| ZF (bit 6) | Zero flag — Set if the result is zero; cleared otherwise. |
| SF (bit 7) | Sign flag — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.) |
| OF (bit 11) | Overflow flag — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic. |

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.



3.5 INSTRUCTION POINTER

The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions.

The EIP register cannot be accessed directly by software; it is controlled implicitly by control-transfer instructions (such as JMP, Jcc, CALL, and RET), interrupts, and exceptions. The only way to read the EIP register is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack. The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET). See Section 6.2.4.2, "Return Instruction Pointer."

All IA-32 processors prefetch instructions. Because of instruction prefetching, an instruction address read from the bus during an instruction load does not match the value in the EIP register. Even though different processor generations use different prefetching mechanisms, the function of the EIP register to direct program flow remains fully compatible with all software written to run on IA-32 processors.

Siehe:

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1 Basic Architecture

3.7 OPERAND ADDRESSING

IA-32 machine-instructions act on zero or more operands. Some operands are specified explicitly and others are implicit. The data for a source operand can be located in:

- the instruction itself (an immediate operand)
- a register
- a memory location
- an I/O port

When an instruction returns data to a destination operand, it can be returned to:

- a register
- a memory location
- an I/O port

3.7.1 Immediate Operands

Some instructions use data encoded in the instruction itself as a source operand. These operands are called **immediate** operands (or simply **immediates**). For example, the following ADD instruction adds an immediate value of 14 to the contents of the EAX register:

`ADD EAX, 14`

All arithmetic instructions (except the DIV and IDIV instructions) allow the source operand to be an immediate value. The maximum value allowed for an immediate operand varies among instructions, but can never be greater than the maximum value of an unsigned doubleword integer (2^{32}).

3.7.2 Register Operands

Source and destination operands can be any of the following registers, depending on the instruction being executed:

- 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP)
- 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, or BP)
- 8-bit general-purpose registers (AH, BH, CH, DH, AL, BL, CL, or DL)
- segment registers (CS, DS, SS, ES, FS, and GS)
- EFLAGS register
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM7) and the MXCSR register
- control registers (CR0, CR2, CR3, and CR4) and system table pointer registers (GDTR, LDTR, IDTR, and task register)
- debug registers (DR0, DR1, DR2, DR3, DR6, and DR7)
- MSR registers

Some instructions (such as the DIV and MUL instructions) use quadword operands contained in a pair of 32-bit registers. Register pairs are represented with a colon separating them. For example, in the register pair EDX:EAX, EDX contains the high order bits and EAX contains the low order bits of a quadword operand.

Several instructions (such as the PUSHFD and POPFD instructions) are provided to load and store the contents of the EFLAGS register or to set or clear individual flags in this register. Other instructions (such as the Jcc instructions) use the state of the status flags in the EFLAGS register as condition codes for branching or other decision making operations.

The processor contains a selection of system registers that are used to control memory management, interrupt and exception handling, task management, processor management, and debugging activities. Some of these system registers are accessible by an application program, the operating system, or the executive through a set of system instructions. When accessing a system register with a system instruction, the register is generally an implied operand of the instruction.



Adressierungsarten

Addressierung (Addressing):

- CPU greift auf Speicheradresse zu
- schreibt oder liest Daten aus

Es gibt zahlreiche Möglichkeiten dies zu tun!

3.7.6 Assembler and Compiler Addressing Modes

At the machine-code level, the selected combination of displacement, base register, index register, and scale factor is encoded in an instruction. All assemblers permit a programmer to use any of the allowable combinations of these addressing components to address operands. High-level language compilers will select an appropriate combination of these components based on the language construct a programmer defines.

3.7.7 I/O Port Addressing

The processor supports an I/O address space that contains up to 65,536 8-bit I/O ports. Ports that are 16-bit and 32-bit may also be defined in the I/O address space. An I/O port can be addressed with either an immediate operand or a value in the DX register. See Chapter 19, "Input/Output," for more information about I/O port addressing.

Effektive Adresse

- durch Adressierungsart spezifizierte Speicheradresse im Hauptspeicher
- entsteht in CPU nach Ausführung der Adressrechnung (z.B. aus relativen Adressen).

virtuelle Speicherverwaltung:

- Effektive Adresse = logische Adresse wird weiteren Speicherverwaltungsoperationen in Speicherverwaltungseinheit = Memory Management Unit (MMU) unterworfen, um letztendlich physikalische Adresse zu erzeugen, mit der dann auf Hauptspeicher zugegriffen wird.



Implizite und explizite Adressierungsarten

Implizite Operanden-Adressierung:

- festgelegt in Architektur oder durch Opcode des Befehls
- Beispiel Kellerarchitektur:
Quell- und Zieloperand einer arithmetisch-logischen Operation sind implizit als die beiden bzw. das oberste Kellerregister festgelegt.
- Beispiel Akkumulator-Architektur:
Akkumulator-Register ist bereits implizit als Quell- und als Zielregister der arithmetisch-logischen Operationen fest vorgegeben.
- Bei Register-Speicher-Architektur:
ist meist ein Operanden-Register fest vorgegeben, während zweiter Operand und Ziel explizit adressiert werden müssen.

Explizite (fundamentale) Adressierungsarten:

- Unmittelbare Adressierung
- Registeradressierung
- Speicheradressierung

Bei IA32 können sich Operanden im Befehl, im Register, in Speicherstelle (ggf. in I/O Port) befinden.
(kein Operand auch möglich)

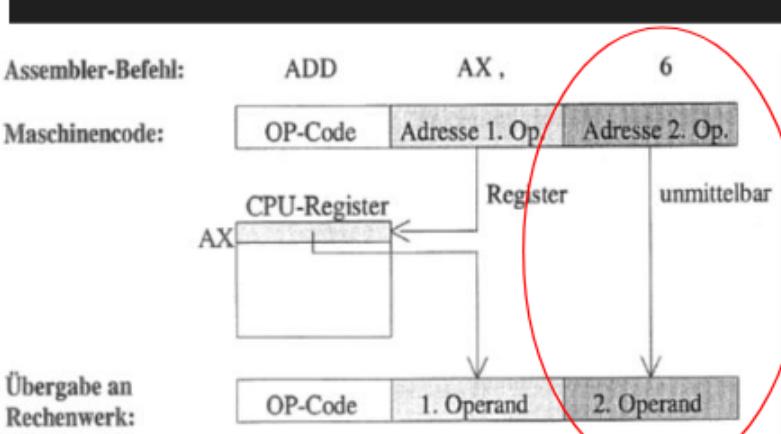


Unmittelbare Adressierung

immediate addressing, Direkt-Operand

- Operand ist im Befehl (im Code-Segment) enthalten.
- keine Berechnung der Operanden-Adresse
- kein „Operand Fetch“
- BSP:

```
add ax, 6 ; add constant 6 (Direktoperand) to register ax
```



INSTRUCTION SET REFERENCE, A-L

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 ib	ADD AL, imm8	I	Valid	Valid	Add imm8 to AL.
05 iw	ADD AX, imm16	I	Valid	Valid	Add imm16 to AX.
05 id	ADD FAX, imm32	I	Valid	Valid	Add imm32 to FAX.

imm16:
16-bit immediate
(word) value

- **imm8** — An immediate byte value. The imm8 symbol is a signed number between -128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.
- **imm16** — An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between -32,768 and +32,767 inclusive.
- **imm32** — An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and -2,147,483,648 inclusive.



Register Adressierung

- logische Adresse ist direkt im Befehl angegeben
- keine effektive Adresse zu berechnen
- Registeradresse i.d.R. 3 - 5 bit lang
- Kein Speicherzugriff erforderlich

```
add eax, esi      ; Register Operand Mode
mov eax, ebx      ; kopiere Inhalt von ebx nach eax
mov ds, ax        ; kopiere Inhalt von ds nach ax
dec eax          ; Dekrementiere eax
clc              ; implizit: clc (clear carry flag),
sar ax, 1        ; shift arithmetic right (SAR): 1 bit to the right.
| | | |           ; MSB: set according to sign of original value.
```



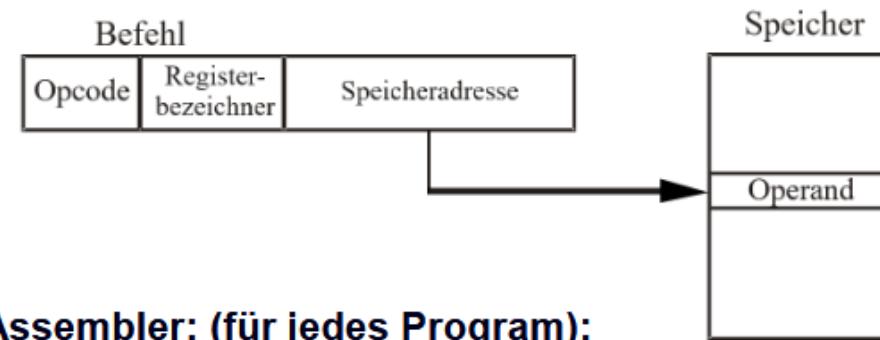
Speicheradressierung

Direkter Speicher Zugriff/ Direct Memory Addressierung

Auf Speicherzelle wird direkt

- über Adresse
- oder Offset/ Displacement
(bezogen auf Anfang eines Segments)

zugegriffen.



Assembler: (für jedes Program):

- berechnet **offset-Wert**
- speichert **symbol table:**
enthält für alle Variablen den offset-Wert

Zusammenhang:

Die genaue Lokation im Hauptspeicher =
segment start Adresse + offset Adresse.

- Startpunkt: segment start address
(typischerweise im DS-Register)
- Zusätzlich: offset - Wert.
(wird als effective address bezeichnet)
- **In direct addressing mode, the offset value is specified directly as part of the instruction**

Prinzip:

- Ein Operand bezieht sich auf Adresse im Hauptspeicher
für gewöhnlich: Als Variable dargestellt
- Ein Operand referenziert ein Register
- Notwendig: direkter Zugriff auf main memory (ist möglich);
für gewöhnlich eingeschränkt auf ein data segment
- Langsame Adressierungsart



Speicheradressierung

Direkte-Offset Adressierung

Beispiele: Direct-Offset-Adressierung:

- Variablenname steht im Befehl
- Assembler berechnet offset-Wert

direct_offset_adressing.asm

```
1 global main
2 extern printf
3
4 section .text
5
6 print_EAX:          ; Set up for calling _printf to print EAX's value
7     push EAX           ; Pushes the value at Address BYTE_VALUE onto the stack
8     push ECX           ; Pushes the string address onto the stack
9     call printf         ; Calls the _printf function from the C library
10
11     add ESP, 8          ; Cleans up the stack after the call to _printf: Corrects the stack pointer
12
13     ret
14
15 main:               ; set eax = 0
16     mov eax, 0x00000000
17     mov ecx, formatString0
18     mov AL, [BYTE_TABLE[0]] ; mov value at Address BYTE_VALUE[0] to AL
19     CALL print_EAX      ; Calls the procedure to print EAX's value
20
21     mov eax, 0x00000000
22     mov ecx, formatString1
23     mov AL, [BYTE_TABLE[1]] ; mov value at Address BYTE_VALUE[1] to AL
24     CALL print_EAX      ; Calls the procedure to print EAX's value
25
26     mov eax, 0x00000000
27     mov ecx, formatString2
28     mov AL, [BYTE_TABLE[2]] ; mov value at Address BYTE_VALUE[1] to AL
29     CALL print_EAX      ; Calls the procedure to print EAX's value
30
31     mov eax, 0x00000000
32     mov ecx, formatString3
33     mov AL, [BYTE_TABLE+3] ; mov value at Address BYTE_VALUE[1] to AL
34     CALL print_EAX      ; Calls the procedure to print EAX's value
35
36     ret                 ; Returns from _main
37
38 section .data
39
40 formatString dd 8          ;
41 formatString0 db 'BYTE_TABLE Element Nr. 0 equals to %d.', 10, 0 ; Format string for _printf
42 formatString1 db 'BYTE_TABLE Element Nr. 1 equals to %d.', 10, 0 ; Format string for _printf
43 formatString2 db 'BYTE_TABLE Element Nr. 2 equals to %d.', 10, 0 ; Format string for _printf
44 formatString3 db 'BYTE_TABLE Element Nr. 3 equals to %d.', 10, 0 ; Format string for _printf
45
46 BYTE_TABLE db 14, 15, 22, 45 ; A table of bytes is defined (Elements one byte each)
```

Variablen-Name

(Assembler hat Symbol-Table, und „weiss“
damit die offset-Adresse)

Daten aus dem
Hauptspeicher
werden in Register
AL
Und ECX geschrieben

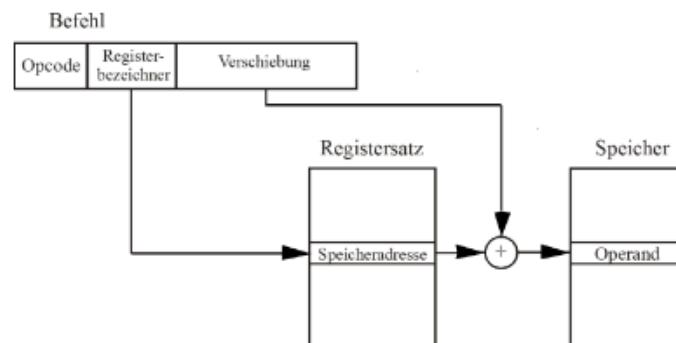


Direkte- und Registerindirekte-Adressierung mit Verschiebung (Displacement)

Die Summe eines Registerwertes und des konstanten Verschiebungswerts kombiniert die registerindirekte Adressierung mit der absoluten Adressierung

- Wenn Displacement = 0
=> Registerindirekte Adressierung
- Wenn Registerwert = 0
=> Direkte Adressierung

indirect_indexed_adressing.asm



Im Bsp:
8 = Verschiebung (Displacement)

```
3 gives main
4 extern printf
5
6 section .text
7
8 print_EAX: ; Set up for calling _printf to print EAX's value
9     push eax
10    push edx
11    call printf
12    add esp, 8 ; Clean up the stack after the call to printf
13
14 print_TABLE: ; First element
15    mov edx, 0 ; mov edx, 0
16    mov eax, DWORD_TABLE[edx] ; mov first Value of DWORD_TABLE to eax
17    mov ecx, formatString
18    push edx ; push edx
19    push ecx ; push ecx
20    call print_EAX ; calls the procedure to print EAX's va
21    pop edx
22    add edx, 1 ; next element
23    mov eax, DWORD_TABLE[edx*4] ; mov second Value of DWORD_TABLE to eax
24    mov ecx, formatString
25    push edx ; push edx
26    push ecx ; push ecx
27    call print_EAX ; calls the procedure to print EAX's va
28    pop edx
29
30    add edx, 1 ; next element
31    mov eax, DWORD_TABLE[edx*4] ; mov second Value of DWORD_TABLE to eax
32    mov ecx, formatString
33    push edx ; push edx
34    push ecx ; push ecx
35    call print_EAX ; calls the procedure to print EAX's va
36    pop edx
37
38    add edx, 1 ; next element
39    mov eax, DWORD_TABLE[edx*4] ; mov second Value of DWORD_TABLE to eax
40    mov ecx, formatString
41    push edx ; push edx
42    push ecx ; push ecx
43    call print_EAX ; calls the procedure to print EAX's va
44    pop edx
45
46    add edx, 1 ; next element
47    mov eax, DWORD_TABLE[edx*4] ; mov second Value of DWORD_TABLE to eax
48    mov ecx, formatString
49    push edx ; push edx
50    push ecx ; push ecx
51    call print_EAX ; calls the procedure to print EAX's va
52
53    mov eax, 100 ; effective Address of DWORD_TABLE is copied in eax
54    mov edx, DWORD_TABLE ; Das erste Double Word des DWORD_TABLE wird im Hauptspeicher mit 100 übe
55    mov [edx], eax ; Das erste Double Word des DWORD_TABLE wird im Hauptspeicher mit 100 übe
56
57    mov eax, 200 ; effective Address of DWORD_TABLE is copied in edx
58    mov edx, DWORD_TABLE ; Das dritte Double Word des DWORD_TABLE wird im Hauptspeicher mit 200 übe
59    mov [edx+8], eax ; Das dritte Double Word des DWORD_TABLE wird im Hauptspeicher mit 200 übe
60
61    CALL print_TABLE
62
63    mov eax, 300 ; effective Address of DWORD_TABLE
64    mov edx, DWORD_TABLE ; Das vierte Double Word des DWORD_TABLE
65    mov [edx], eax ; Das vierte Double Word des DWORD_TABLE
66
67    CALL print_TABLE
68
69    ret ; Returns from _main
70
71 section .data
72    Displacement DD 4 ; one element of the array: 4 bytes long
73    DWORD_TABLE DD 50, 51, 52, 53 ; Allocates 4 dwords (32 bit)
74
75 ; Formatingstring do DWORD_TABLE Element Nr. 0 equals to $d., 50, 0 ; Format string
76 ; Formatingstring do DWORD_TABLE Element Nr. 1 equals to $d., 51, 0 ; Format string for _printf
77 ; Formatingstring do DWORD_TABLE Element Nr. 2 equals to $d., 52, 0 ; Format string for _printf
78 ; Formatingstring do DWORD_TABLE Element Nr. 3 equals to $d., 53, 0 ; Format string for _printf
79
```



Indirect Indexed Addressing

Die effektive Adresse des Operanden ist die Addition des Wertes Registers mit einem vorgegebenen Indexregister.

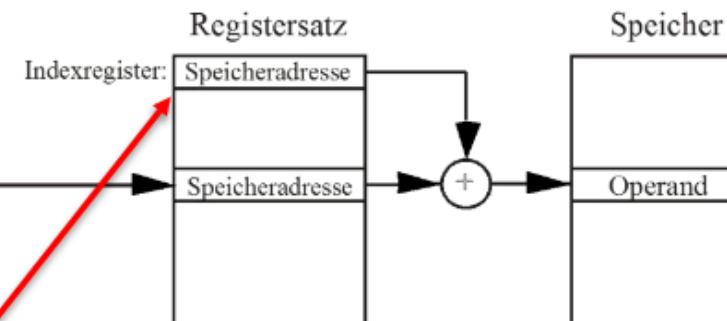
Mittels des Indexregister wird z.B. auf Elemente von Arrays (Tabellen) und Strings zugegriffen

indirect_indexed_adressing.asm

```
1 global _main
2 extern _printf
3
4 section .text
5
6 print_HEX:
7     push eax
8     push ecx
9     call _printf
10    add esp, 8
11    ret
12
13 print_TABLE:
14     mov edx, 0
15     mov eax, DWORD_TABLE[edx*4]
16     mov ecx, formatString3
17     push edx
18     call print_EAX
19     pop edx
20
21     add edx, 1
22     mov eax, DWORD_TABLE[edx*4]
23     mov ecx, formatString3
24     push edx
25     call print_EAX
26     pop edx
27
28     add edx, 1
29     mov eax, DWORD_TABLE[edx*4]
30     mov ecx, formatString3
31     push edx
32     call print_EAX
33     pop edx
34
35     add edx, 1
36     mov eax, DWORD_TABLE[edx*4]
37     mov ecx, formatString3
38     push edx
39     call print_EAX
40     pop edx
41
42     add edx, 1
43     mov eax, DWORD_TABLE[edx*4]
44     mov ecx, formatString3
45     push edx
46     call print_EAX
47     pop edx
48
49     add edx, 1
50     mov eax, DWORD_TABLE[edx*4]
51     mov ecx, formatString3
52     push edx
53     call print_EAX
54     pop edx
55
56     call _printf
57     mov eax, 300
58     mov edx, DWORD_TABLE
59     mov [edx], eax
60
61     mov eax, 300
62     mov edx, DWORD_TABLE
63     mov [edx+8], eax
64
65     CALL print_TABLE
66
67     ret
68
69 section .data
70     Displacement DD 4
71     DWORD_TABLE DD 50, 51, 52, 53
72
73 formatString3 db "DWORD_TABLE Element Nr. 0 equals to $d.", 50, 0 ; Format string
74 formatString3 db "DWORD_TABLE Element Nr. 1 equals to $d.", 50, 0 ; Format string for _printf
75 formatString3 db "DWORD_TABLE Element Nr. 2 equals to $d.", 50, 0 ; Format string for _printf
76 formatString3 db "DWORD_TABLE Element Nr. 3 equals to $d.", 50, 0 ; Format string for _printf
```

Befehl

Opcode	Register-bezeichner	...
--------	---------------------	-----



EDX dient als Indexregister

Based Indexed Addressing

- Base-Register : alle GPR-Register
- Index-Register : alle GPR-Register, außer dem Stack-Pointer ESP

```
mov EAX, [EBX+ESI]           ; EAX = memory[EBX + (ESI * 1) + 0]
mov eax, [ebx][esi]          ; Alternative Schreibweise

mov EAX, [EBX+ESI*4+2]       ; EAX = memory[EBX + (ESI * 4) + 2]
| | | | | | | | | | | | | | | ; Base + (Index * Scale factor) + signed displacement
```

PC (IP)-Relative Adressierung

Offset	Maschinencode	Sourcecode
00000000	B8 00000000	mov eax, 0
00000005	B9 000003E8	mov ecx, 1000
0000000A		L1:
0000000A	03 C1	add eax, ecx
0000000C	E2 FC	loop L1
0000000E

Programm Counter
(im Code-Segment)

Erstes Byte im Code-Segment
liegt an Adresse 0x00000000

Der Befehl **mov eax, 0**
belegt 5 Bytes.

Der nächste Befehl beginnt deshalb
an Adresse 0x00000005 :

mov ecx, 1000

Dieser Befehl ist zufällig auch 5 Byte lang

All hex!
B8 00 00 00 00

Der Befehl
mov eax, 0
ist 5 Bytes lang
im Code-
Segment!



PC (IP)-Relative Adressierung

Offset	Maschinencode	Sourcecode
00000000	B8 00000000	mov eax, 0
00000005	B9 000003E8	mov ecx, 1000
0000000A		L1:
0000000A	03 C1	add eax, ecx
0000000C	E2 FC	

INSTRUCTION SET REFERENCE A-L

3.1.1.1 Opcode Column in the Instruction Summary Table (Instructions without

The "Opcode" column in the table above shows the object code produced for each form of t possible, codes are given as hexadecimal bytes in the same order in which they appear in n entries other than hexadeciml bytes are as follows:

- NP — Indicates the use of GC/F2/F3 prefixes (beyond those already part of the instruction) allowed with the instruction. Such use will either cause an invalid-opcode exception (#L encoding for a different instruction).
- NFX — Indicates the use of F2/F3 prefixes (beyond those already part of the instruction) allowed with the instruction. Such use will either cause an invalid-opcode exception (#L encoding for a different instruction).
- REX.W — Indicates the use of a REX prefix that affects operand size or instruction sem. the REX prefix and other optional/mandatory instruction prefixes are discussed Chapter prefixes that promote legacy instructions to 64-bit behavior are not listed explicitly in th
- /digit — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses or memory' operand. The reg field contains the digit that provides an extension to the i
- /r — Indicates that the ModR/M byte of the instruction contains a register operand and cb, cw, cd, cp, co, ct — A 1-byte (cb), 2-byte (cw), 4-byte (cd), 6-byte (cp), 8-byte (co following the opcode. This value is used to specify a code offset and possibly a new value register.
- ib, iw, id, io — A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if th value. All words, doublewords, and quadwords are given with the low-order byte first.
- +rb, +rw, +rd, +ro — Indicated the lower 3 bits of the opcode byte is used to encode without a modR/M byte. The instruction has the corresponding hexadeciml value of the 3 bits as 000b. In non-64-bit mode, a register code, from 0 through 7, is added to the he opcode byte. In 64-bit mode, indicates the four bit field of REX.b and opcode[2:0] field operand of the instruction. "+ro" is applicable only in 64-bit mode. See Table 3-1 for thi
- i — A number used in floating-point instructions when one of the operands is ST(i) from The number i (which can range from 0 to 7) is added to the hexadecimal byte given at t to form a single opcode byte.

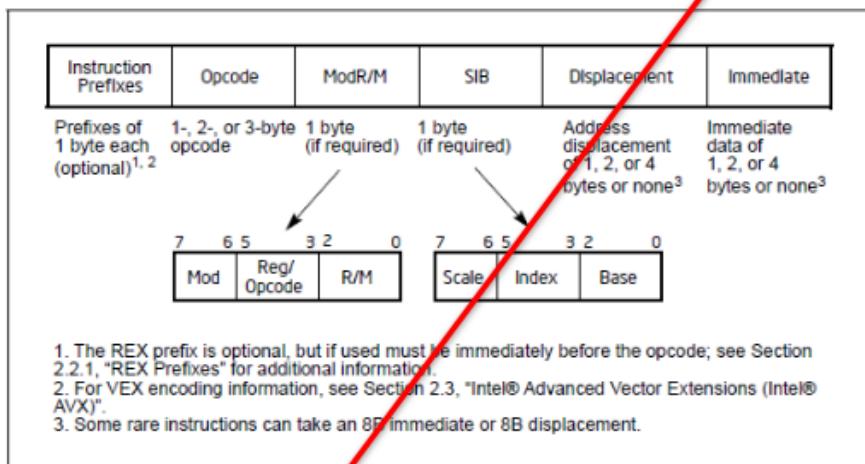


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

MOV-Move

$rd = 0 \Rightarrow B8+rd = B8$
Meaning: mov eax,...

Opcode	Instruction	Op/En	64-Bit Mode	Compa Leg Mo	
B8+ rd id	MOV r32, imm32	01	Valid	Valid	Move imm32 to r32.

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

Register	byte register		word register		dword register		Register
	REXB	Reg Field	Register	REXB	Reg Field	Register	
AL	None	0	AX	None	0	EAX	None
CL	None	1	CX	None	1	ECX	None
DL	None	2	DX	None	2	EDX	None
BL	None	3	BX	None	3	EBX	None
DL	Not encodedable (NE)	4	SP	None	4	ESP	None
CH	NE	5	BP	None	5	EBP	None
DH	NE	6	SI	None	6	ESI	None
BH	NE	7	DI	None	7	EDI	None
SPL	Yes	4	SP	None	4	ESP	None
BPL	Yes	5	BP	None	5	EBP	None



PC (IP)-Relative Adressierung

INSTRUCTION SET REFERENCE, A-L

3.1.1.1 Opcode Column in the Instruction Summary Table (Instructions without VEX Prefix)

The "Opcode" column in the table above shows the object code produced for each form of the instruction. When possible, codes are given as hexadecimal bytes in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- NP** — Indicates the use of 66/F2/F3 prefixes (beyond those already part of the instructions opcode) are not allowed with the instruction. Such use will either cause an invalid-opcode exception (#UD) or result in the encoding for a different instruction.
- NFx** — Indicates the use of F2/F3 prefixes (beyond those already part of the instructions opcode) are not allowed with the instruction. Such use will either cause an invalid-opcode exception (#UD) or result in the encoding for a different instruction.
- REX.W** — Indicates the use of a REX prefix that affects operand size or instruction semantics. The ordering of the REX prefix and other optional/mandatory instruction prefixes are discussed Chapter 2. Note that REX prefixes that promote legacy instructions to 64-bit behavior are not listed explicitly in the opcode column.
- /digit** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
- /r** — Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.
- cb, cw, cd, cp, co, ct** — A 1-byte (cb), 2-byte (cw), 4-byte (cd), 6-byte (cp), 8-byte (co) or 10-byte (ct) value following the opcode. This value is used to specify a code offset and possibly a new value for the code segment register.
- ib, iw, id, io** — A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words, doublewords, and quadwords are given with the low-order byte first.
- +rb, +rw, +rd, +ro** — Indicated the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte. The instruction lists the corresponding hexadecimal value of the opcode byte with low 3 bits as 000b. In non-64-bit mode, a register code, from 0 through 7, is added to the hexadecimal value of the opcode byte. In 64-bit mode, indicates the four bit field of REX.b and opcode[2:0] field encodes the register operand of the instruction. "+ro" is applicable only in 64-bit mode. See Table 3-1 for the codes.
- +i** — A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

Table 3-1. Register Codes Associated with +rb, +rw, +rd, +ro

Register	REXB	Reg Field	byte register			word register			dword register			quadword register (64-Bit Mode only)		
			Register	REXB	Reg Field	Register	REXB	Reg Field	Register	REXB	Reg Field	Register	REXB	Reg Field
AL	None	0	AX	None	0	EAX	None	0	RAX	None	0			
CL	None	1	CX	None	1	ECX	None	1	RCX	None	1			
DL	None	2	DX	None	2	EDX	None	2	RDX	None	2			
BL	None	3	BX	None	3	FBX	None	3	RBX	None	3			
AH	Not encodedable (NE)	4	SP	None	4	ESP	None	4	N/A	N/A	N/A			
CH	NE	5	BP	None	5	EBP	None	5	N/A	N/A	N/A			
DH	NE	6	SI	None	6	ESI	None	6	N/A	N/A	N/A			
BH	NE	7	DI	None	7	EDI	None	7	N/A	N/A	N/A			
SPL	Yes	4	SP	None	4	ESP	None	4	RSP	None	4			
BPL	Yes	5	BP	None	5	EBP	None	5	RBSP	None	5			

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
B8+ rd d	MOV r32, imm32	01	Valid	Valid	Move imm32 to r32.

Offset	Maschinencode	Sourcecode
00000000	B8 00000000	mov eax, 0
00000005	B9 00000031 3	mov ecx, 1000
0000000A		L1:
0000000A	03 C1	add eax, ecx
0000000C	E2 FC	loop L1
0000000E

32 bit

0x00000000 => 32 bit „0“

INSTRUCTION SET REFERENCE, M-U

Pipelining (Fließbandverarbeitung)

Vorstufe zum Pipelining: Prefetch-Puffer

Problem:

Abrufen der Befehle aus dem Hauptspeicher dauert lang.

=> Performance-Engpass

Lösung:

- Befehle schon vorab aus dem Hauptspeicher abrufen, vor sie gebraucht werden, und (Zwischen-)Speicherung in einem Register (genannt Prefetch-Puffer).
- Benötigt der Professor den nächsten Befehl, dann kann dieser sehr schnell aus dem Prefetch-Puffer geholt werden, es ist kein langsamer Hauptspeicher-Zugriff notwendig.

Analyse der Lösung:

- Holen eines Befehls aus dem Hauptspeicher, und weitere Befehlausführung sind parallelisierbar!
- Der aktuelle Befehl wird gerade ausgeführt, parallel kann der nächste Befehl aus dem Hauptspeicher in den Prefetch-Puffer geladen werden.

=> Dies führt zu einer Leistungssteigerung: Es werden mehr Instruktionen pro Zeiteinheit prozessiert

[Tanenbaum]



Pipelining (Fließbandverarbeitung)

Die Idee des Pipelining führt diese Strategie der Parallelität des Prefetch-Puffers noch (viel) weiter.

Grundidee Pipelining:

- Aufteilung der Befehlsausführung nicht nur in 2 Stufen (Holen aus dem Speicher UND weitere Verarbeitung), sondern Aufteilung in (u.U sehr) viele Stufen.
- Diese Stufen werden jeweils von einem extra „Hardware-Block“ nur für die bestimmte Stufe abgearbeitet.
- Alle Stufen können parallel laufen.



Stufe x Stufe y

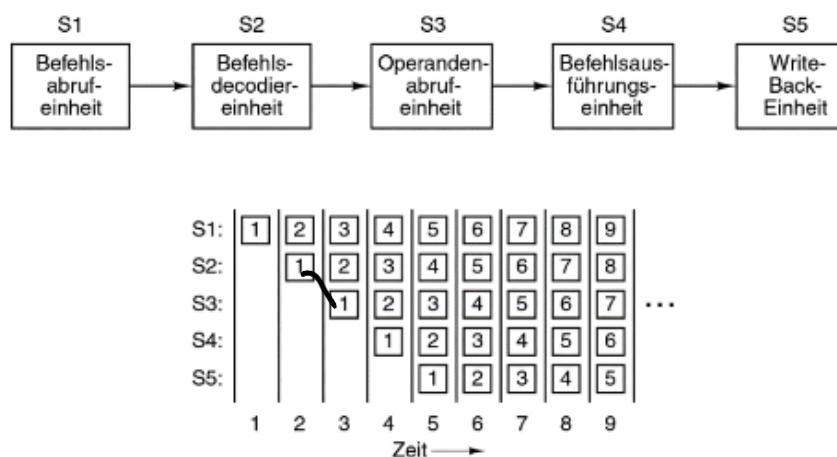
[Tanenbaum]



Pipelining (Fließbandverarbeitung)

Beispiel Pipeline mit fünf Einheiten
(Stufen/Stages):

- S1 Befehlsabrufeinheit (Instruction Fetch):
Holt Befehl aus Speicher in Puffer
- S2 Befehlsdecodiereinheit (Instruction Decode):
Decodiert den Befehl, und bestimmt dabei
Typ und notwendige Operanden, evtl.
Adressrechnung
- S3 Operandenabrufeinheit (Operand Fetch):
Auslesen des Operanden (Speicher/
Register)
- S4 Befehlausführungseinheit (Execute):
Eigentliche Ausführung eines Befehls,
Operanden durch Datenpfad schleusen
- S5 Write Back Einheit:
Zurückschreiben des Ergebnisses



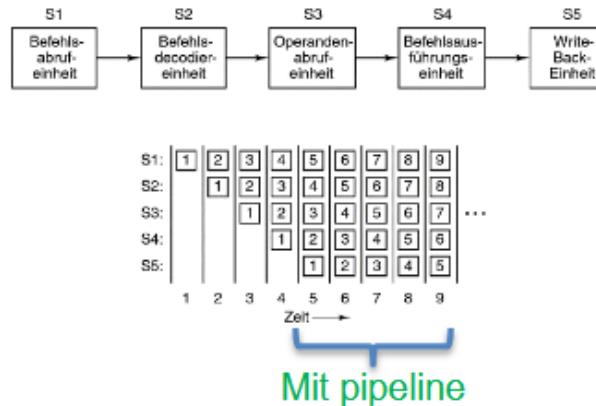
[Tanenbaum]

Pipelining (Fließbandverarbeitung)

Funktionsprinzip:

1. Taktzyklus
 - **S1 : Befehl 1**
2. Taktzyklus
 - S1 : Befehl 2
 - **S2 : Befehl 1**
3. Taktzyklus
 - S1 : Befehl 3
 - S2 : Befehl 2
 - **S3: Befehl 1**
4. Taktzyklus
 - S1 : Befehl 4
 - S2 : Befehl 3
 - S2: Befehl 2
 - **S4: Befehl 1**
5. Taktzyklus
 - S1 : Befehl 5
 - S2 : Befehl 4
 - S2: Befehl 3
 - S4: Befehl 2
 - **S5: Befehl 1**

Fließbandverarbeitung von Befehlen beruht auf der parallelen Bearbeitung von Teilaufgaben unterschiedlicher Befehle.



Beobachtung:

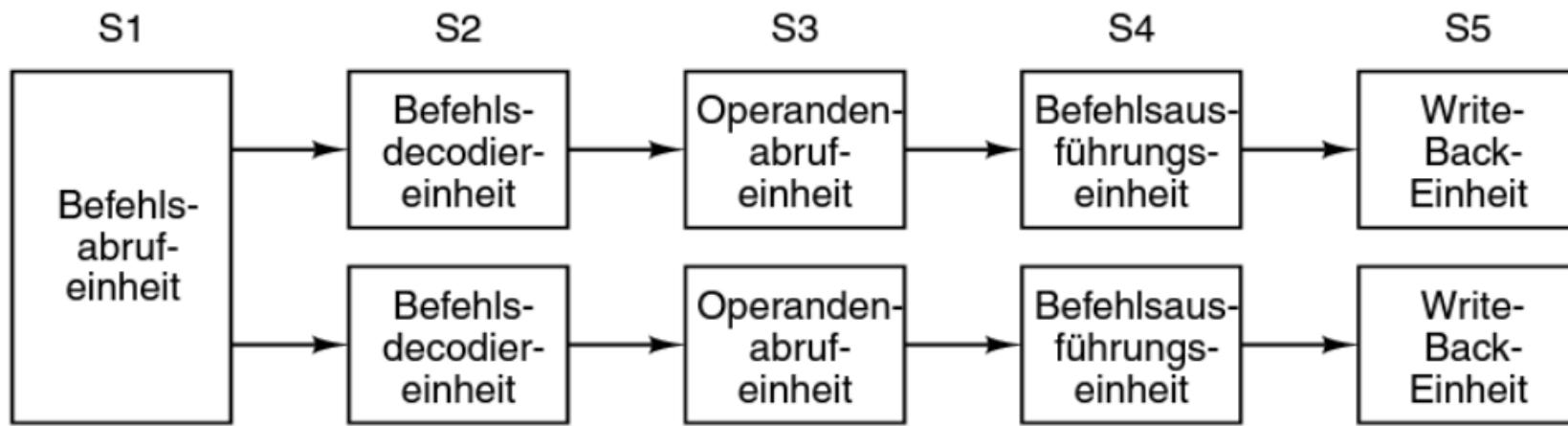
- Zykluszeit 1ns => $f = 1 * 10^9 \text{ Hz} = 1 \text{ GHz}$
- Dauer bis ein Befehl die gesamte 5stufige Pipeline durchläuft: 5ns
- **Ohne Pipeline:** Alle 5ns eine neue Instruction.
⇒ in 1 sec : $\frac{1}{5\text{ns}} = 0,2 \times 10^9 \text{ Hz} = 200 \text{ MHz}$
⇒ **200 MIPS** (Million Instructions per second)
- **Mit Pipeline:** Jede ns eine neue Instruction
⇒ in 1 sec : $\frac{1}{1\text{ns}} = 1 \times 10^9 \text{ Hz} = 1 \text{ GHz}$
⇒ **1.000 MIPS** (Million Instructions per second)

[Tanenbaum]



Superskalare Architekturen

Idee: Doppelte Pipeline



- Wie bisher: 5stufige Pipeline
- Befehls-Paare:
 - 2 Befehle
 - Kollidieren bei der parallelen Ausführung nicht (Nutzung der Register)
 - Sind nicht voneinander abhängig
- Eine gemeinsame Befehlsabrufeinheit:
Ruft ein Befehlspaar ab – Dann werden beide in eine Pipeline gesteckt
- Bsp: Pentium: 2 fünfstufige Pipelines (u-Pipeline, v-Pipeline)
i486 nur eine fünfstufige Pipeline => Pentium ca. doppelt so schnell (Integer Arithmetik)

[Tanenbaum]

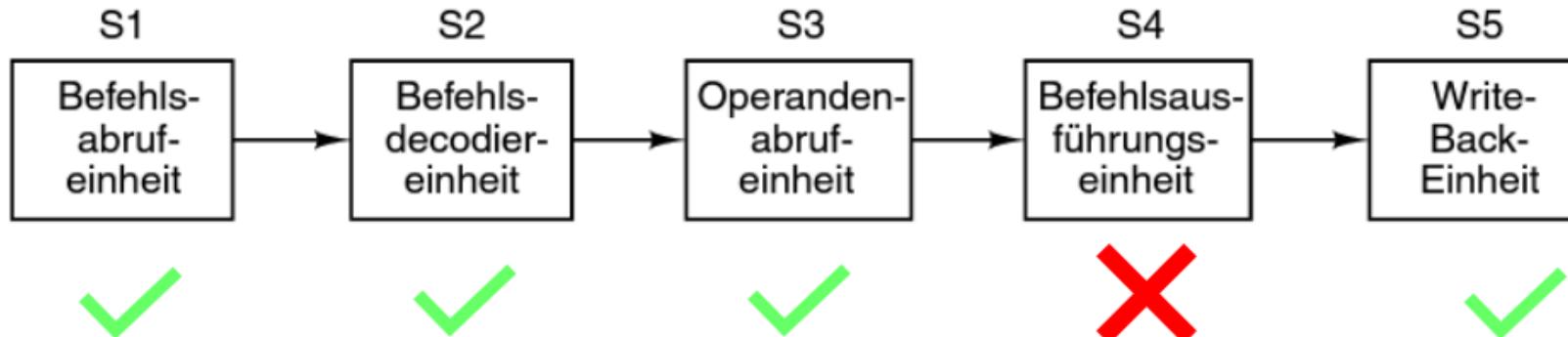
Superskalare Architekturen

Kritische Betrachtung

INSTRUCTION SET REFERENCE, A-L

IMUL—Signed Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F AF /r	IMUL r32, r/m32	RM	Valid	Valid	doubleword register := doubleword register * r/m32.



5stufige Pipeline:

Annahme-bisher:

Jede Stufe benötigt einen Taktzyklus!

Dauert u.U.
viele Taktzyklen

„Verstopfung ??“

[Tanenbaum]



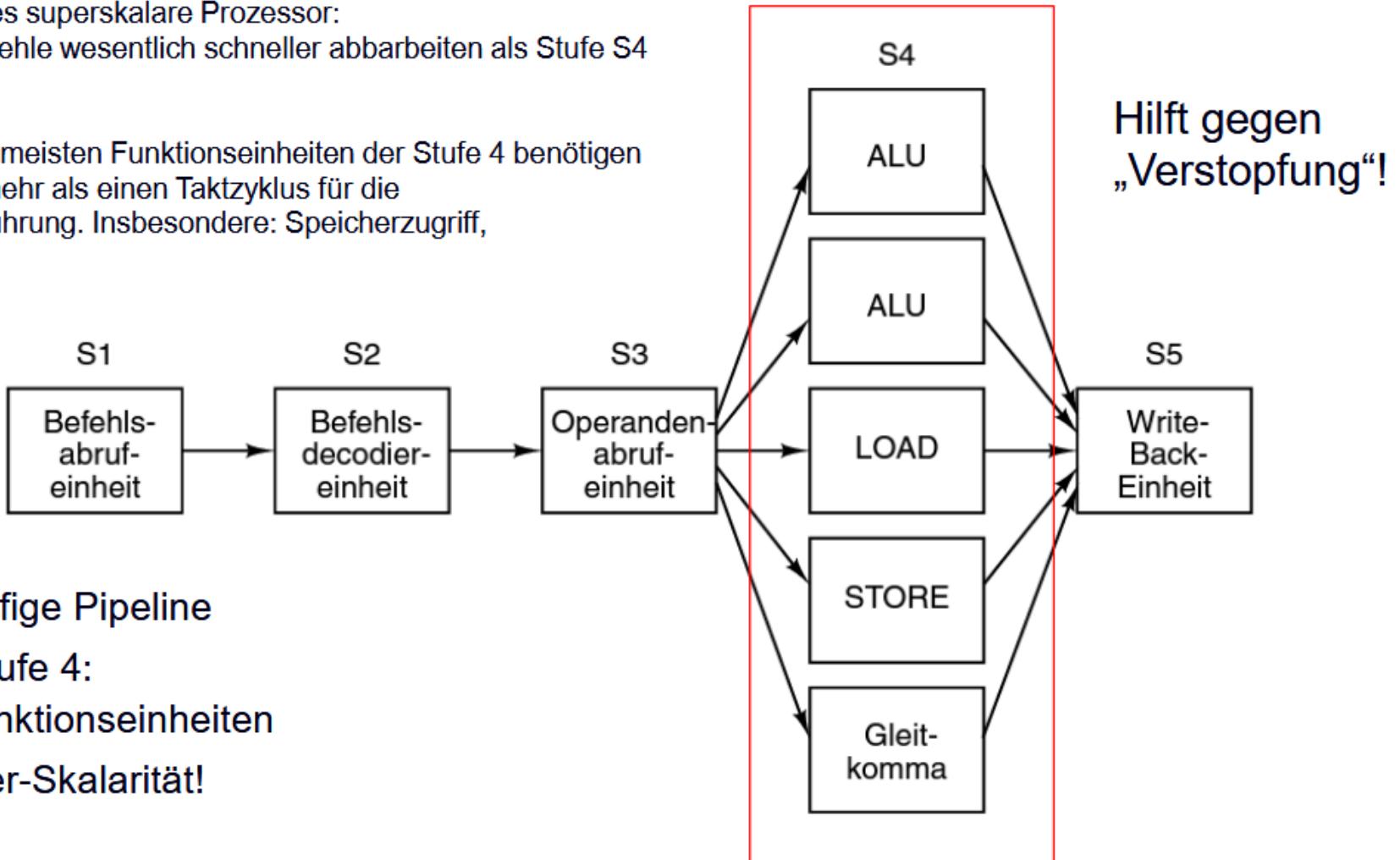
Superskalare Architekturen

Mögliche Lösung – mehrere Funktionseinheiten

Konzept eines superskalaren Prozessor:

Stufe S3 Befehle wesentlich schneller abarbeiten als Stufe S4 das kann.

Realität: Die meisten Funktionseinheiten der Stufe 4 benötigen wesentlich mehr als einen Taktzyklus für die Befehlsausführung. Insbesondere: Speicherzugriff, Gleitkomma.



[Tanenbaum]



Superskalare Architekturen

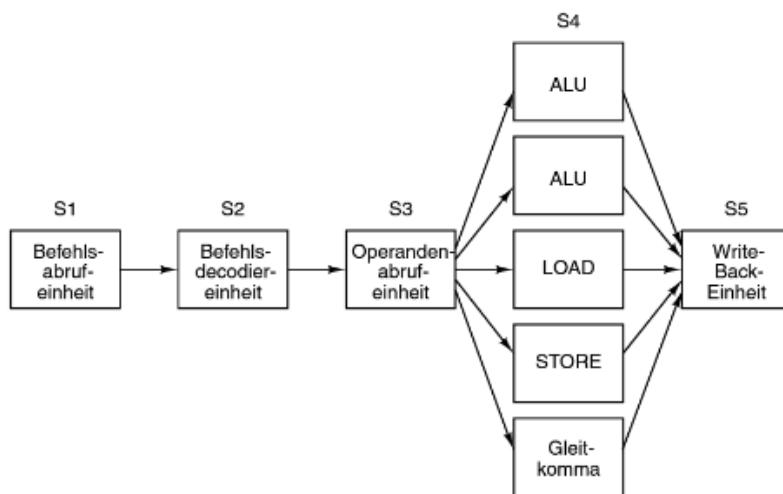
Definition

Die Definition von „superskalar“ hat sich im Laufe der Zeit gewandelt.

Heute beschreibt man damit Prozessoren, die mehrere – oftmals vier oder sechs – Befehle in einem einzigen Taktzyklus initiieren.

Dazu hat ein superskalarer Prozessor mehrere Funktionseinheiten, an die er alle diese Befehle übergibt.

Da superskalare Prozessoren im Allgemeinen eine Pipeline haben, ähneln sie dem beispielhaftem Aufbau:



[Tanenbaum]



Parallelität auf Befehlsebene

Parallelität:

bei gegebener Taktfrequenz laufen 2 oder mehr Jobs parallel ab.

Dadurch ergibt sich eine Zeitersparnis, pro Zeiteinheiten können mehr Befehle gestartet werden.

Eine Beschleunigung der Verarbeitung lässt sich durch die parallele Ausführung von Befehlen in:

- einem Prozessor
=> Parallelität auf Befehlsebene
- oder durch die Verwendung von mehreren Prozessoren
=> Parallelität auf Prozessorebene

erreichen

[Tanenbaum]



Parallelität auf Prozessorebene

Beobachtung:

Parallelität auf Befehlsebene:

- Fließbandverarbeitung
- Super-Skalarität

⇒ Kann Prozessoren um den Faktor 5-10 schneller machen.

Gesucht:

Performance-Zuwachs um Faktor 100 und mehr...

Lösung:

Parallelität auf Prozessorebene!

⇒ Mehrere Prozessoren

[Tanenbaum]



Pipeline Konflikte: 3 Arten

Pipelining führt zu Leistungssteigerung!

Aber: Es gibt Situationen, bei denen Befehle nicht taktsynchron die einzelnen Stufen einer Befehls-Pipeline durchlaufen können.

Solche Ereignisse werden als Pipeline-Konflikte (Hazards) bezeichnet.

1. Data Hazards

Datenkonflikte ergeben sich aus Datenabhängigkeiten zwischen Befehlen im Programm; z.B. Berechnung erfordert Ergebnis eines noch in der Pipeline befindlichen Vorgängerbefehles

2. Struktural Hazards (Resource Dependency)

Mehrere Pipeline-Stufen benötigen zeitgleich die selbe Ressource

3. Control Hazards (Branch Hazards)

Nachfolgebefehl bzw. auch IF hängt vom Ausgang einer Phase / Änderungen im Kontrollfluß ab (Sprungbefehl, wird gesprungen, wenn ja, wohin, ...)

Wesentlicher Erfolgsfaktor:
Behandlung dieser Pipeline-Konflikte

