Aufbau und Funktionsweise eines Computers

Ein Überblick

- Algorithmenbegriff beinhaltet ein „effektiv“ („mechanisch“) durchführbar
- Entwicklung von Algorithmen kann weitgehend ohne ein konkretes Maschinenmodell erfolgen
- Auch (moderne, höhere) Programmiersprachen sind so entworfen, dass Programme auf verschiedenen Computern ablaufen können
  - Es wird von speziellen Eigenschaften i.A. abstrahiert

- Trotzdem fließen Eigenschaften realer Computer auch in das Design von Programmiersprachen ein
  - Programme sollen nicht nur *effektiv* durchführbar sein, sondern auch *effizient*!
- Im folgenden nur kurze Übersicht über Aufbau und Funktionsweise eines Computers
  - Literatur z.B.: Oberschelp/Vossen oder Tanenbaum

- Computersysteme bestehen aus **Hardware** und **Software**
- Die Hardware ist fest gegeben, kann angefasst werden und ist (bis auf den Austausch von Komponenten) unveränderlich
- Die Software besteht aus den gespeicherten Programmen, die durch die Hardware ausgeführt werden
  - Software ist unsichtbar
  - Sehr leicht zu ändern / zu speichern / auszuführen, da sich dies nur in der Änderung von magnetischen (bei Festplatten) oder elektrischen (bei Speichern und Prozessoren) Zuständen der Hardware auswirkt, nicht aber in der Änderung fester Bestandteile

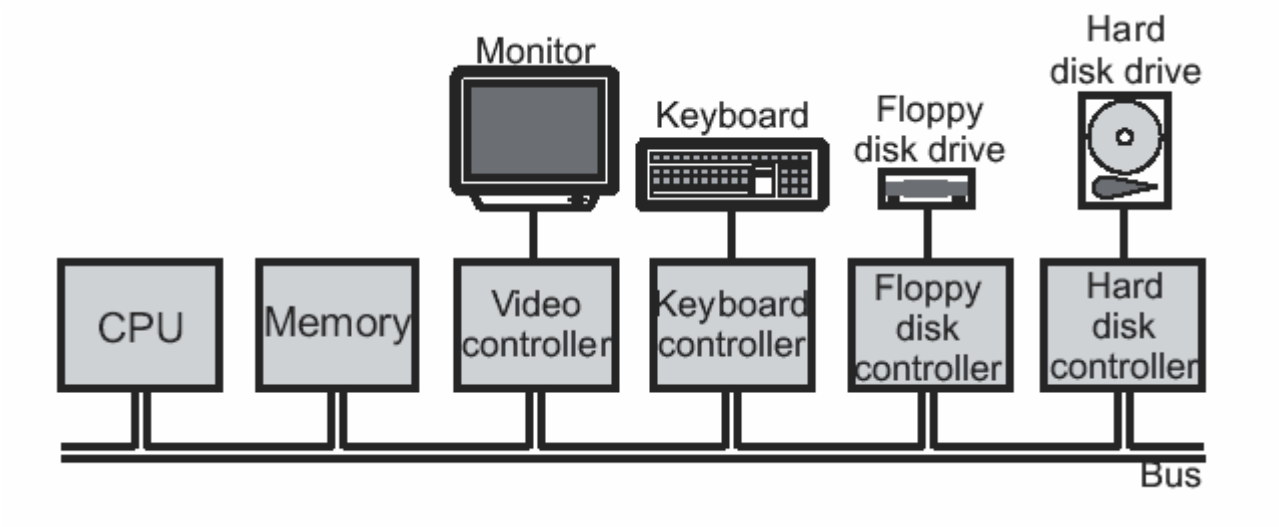
The Analytical Engine consists of two parts:

1st. The store in which all the variables to be operated upon, as well as all those quantities which have arisen from the result of other operations are placed.

2nd. The mill into which the quantities about to be operated upon are always brought.

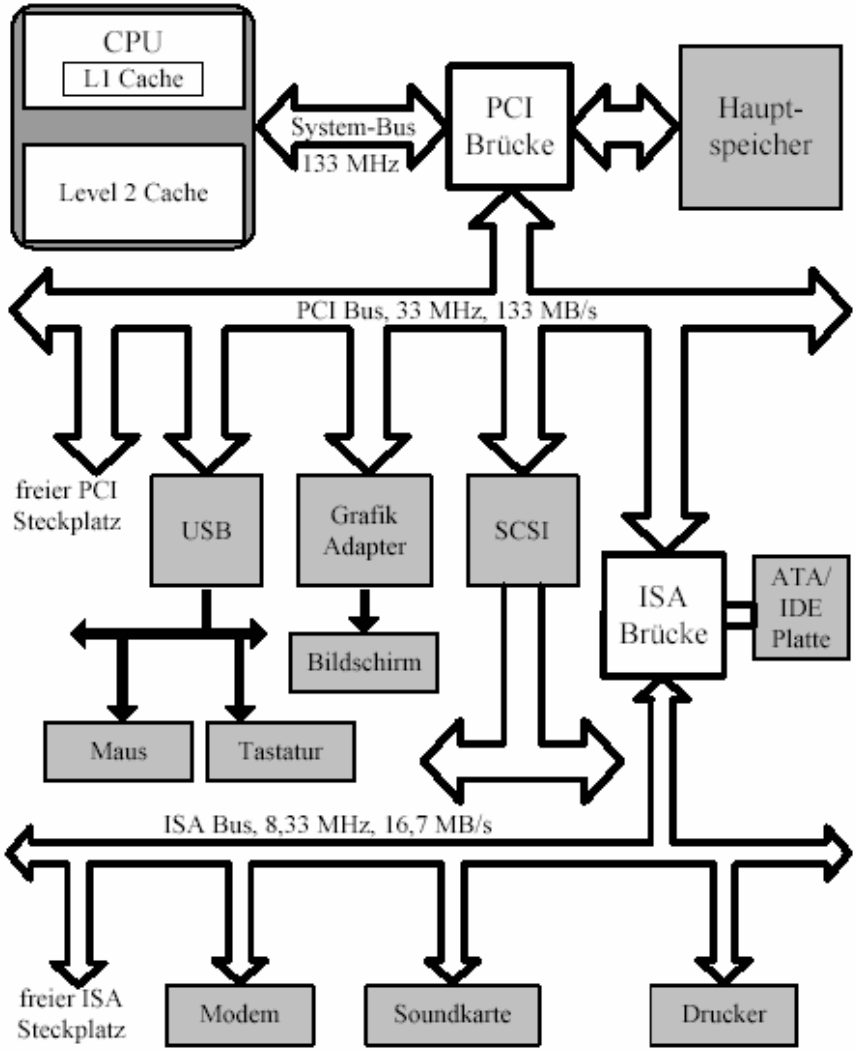
Charles Babbage (1864)

- Architektur eines einfachen Computersystems mit Bus

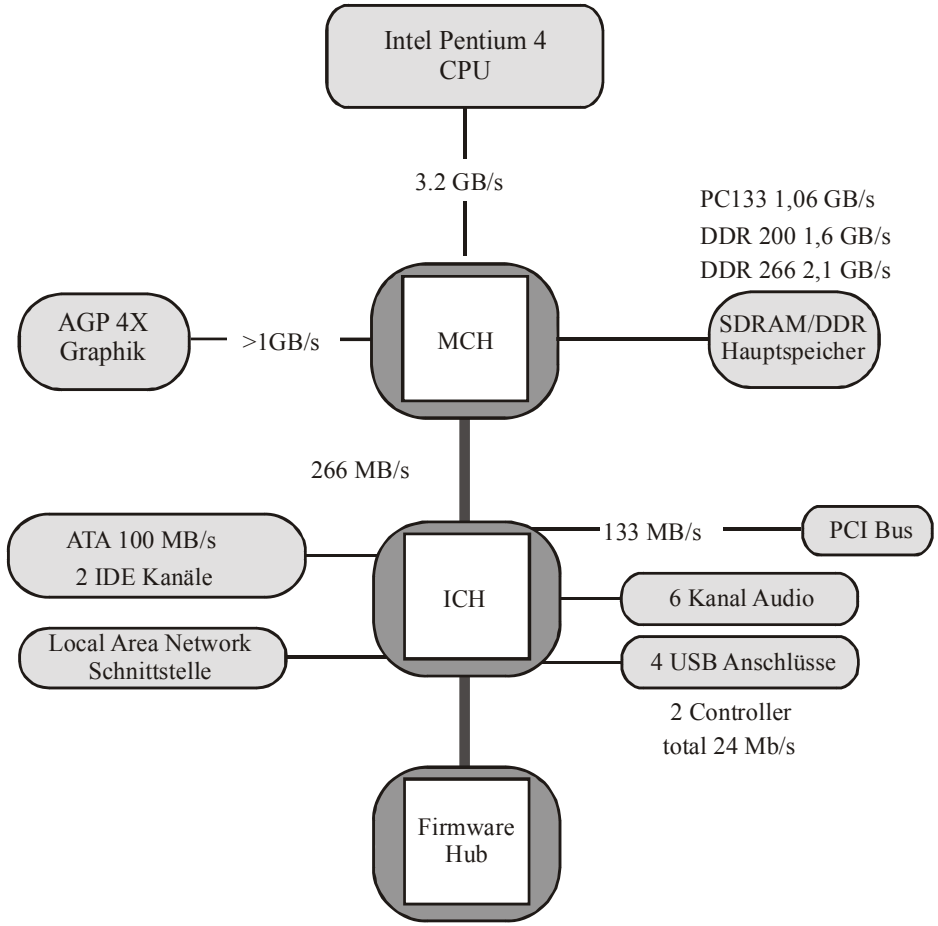


System-Architektur der Hardware

Architektur eines Intel P2 PC Systems mit mehreren Bussen an Brücken

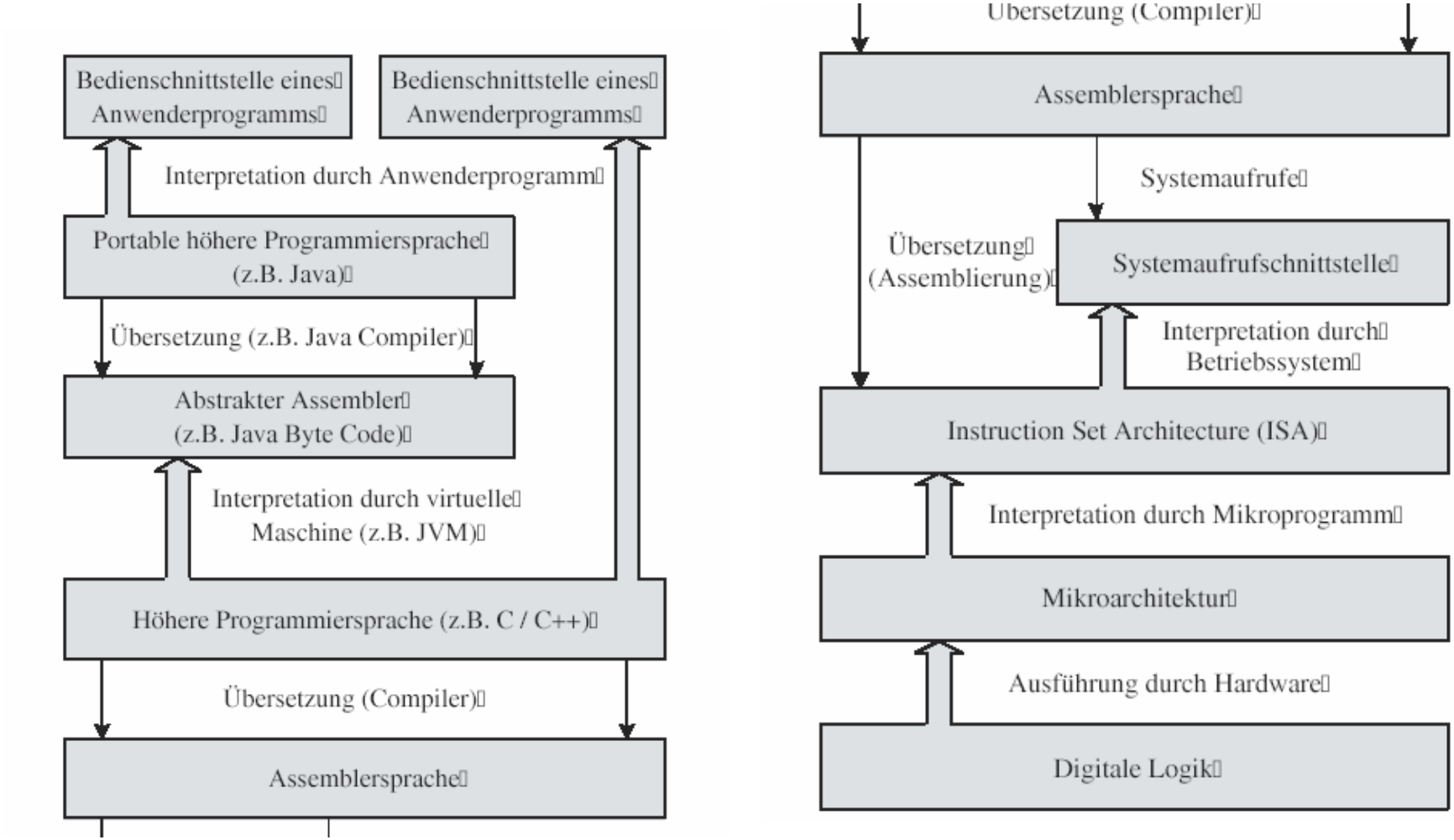


Architektur eines Intel P4 PC Systems mit mehreren Bussen an Hubs



- MCH: Memory controller hub
- ICH: I/O controller hub

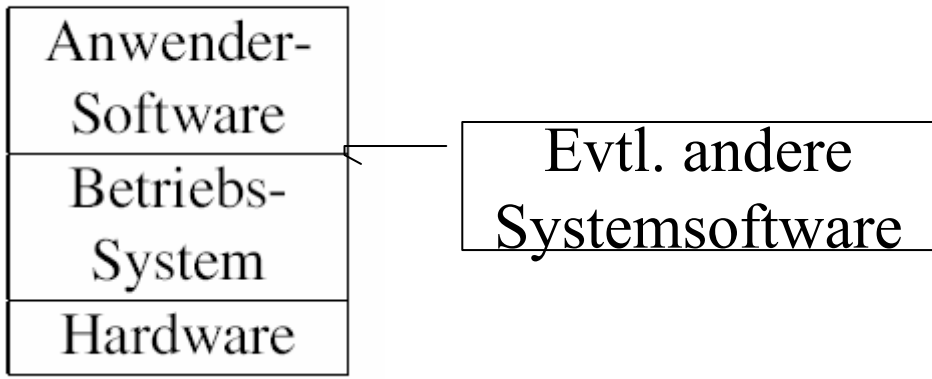
System-Architektur der Software: Schichtenaufbau eines Rechnersystems



- Typische Softwarekomponenten sind
  - Programme der **Anwendersoftware** (application software) zur Lösung von Problemen der externen Welt der Anwender,
  - sowie die Programme der **Systemsoftware** (system software) zur Lösung interner Aufgaben im Rechner.
- Anwendersoftware (z. B. Textverarbeitung, Tabellenkalkulation, Bildbearbeitung, Buchhaltung, Produktionsplanung, Lohn und Gehaltsabrechnung, Spiele) ist der Grund, weswegen der Anwender letztlich einen Rechner kauft
- Systemsoftware hilft beim Betrieb des Rechners und bei der Konstruktion der Anwendersoftware
  - Systemsoftware umfasst neben Datenbanksystemen, Übersetzern (compiler) etc. in jedem Fall das Betriebssystem.

Das **Betriebssystem** (*operating system*) isoliert die Anwendersoftware von der Hardware: das Betriebssystem läuft auf der Hardware und die Anwendersoftware auf dem Betriebssystem

- Das Betriebssystem verwaltet die Ressourcen der Hardware (wie z. B. Geräte, Speicher und Rechenzeit) und es stellt der Anwendersoftware eine abstrakte Schnittstelle (die Systemaufrufschnittstelle) zu deren Nutzung zur Verfügung
- Dadurch vereinfacht es die Nutzung der Ressourcen und schützt vor Fehlbedienungen
  - Betriebssysteme, die es mit diesem Schutz nicht so genau nehmen, führen zu häufigen **Systemabstürzen** (system crash)



- Es gibt heute eine große Vielzahl von Rechnersystemen
  - **Eingebettete Systeme** (embedded system) verbergen sich in allerlei Geräten, wie z. B. Haushaltsgeräten oder Handys
    - In Autos ist die Elektronik schon für ca. 25% des Wertes verantwortlich (wird bis ca. 40% steigen)

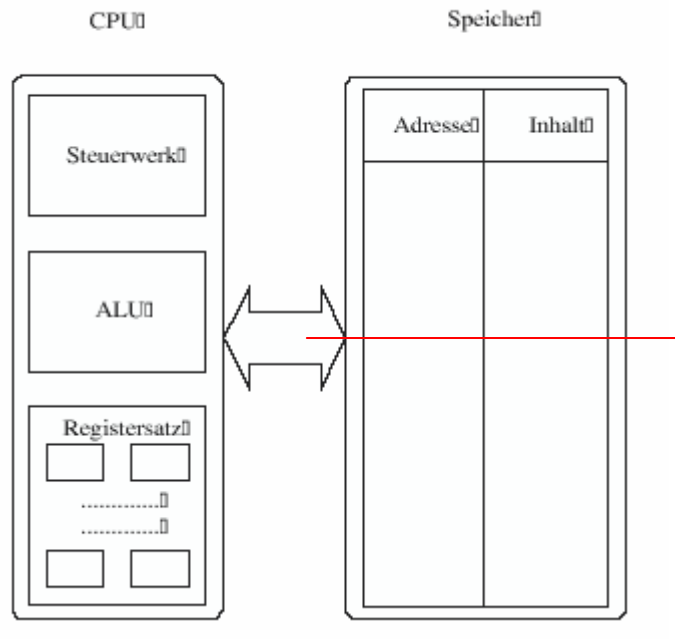
      Aus Sicht der Informatik sind sie rollende Rechnernetze
  - „Übliche Computer“ kann man grob einteilen in
    - **PCs** (personal computer),
    - **Arbeitsplatzrechner** (workstation),

      } Tendenz: zunehmend weniger unterscheidbar
    - **betriebliche Großrechner** (business mainframe)
    - **wissenschaftliche Großrechner** (supercomputer)

      } Tendenz: viele vernetzte „Standard PCs“ (z.B. Google)

Der Kern des Rechners: von Neumann Architektur

- Grundsätzlicher Aufbau verschiedener Rechnersysteme im Grundprinzip gleich



Wahlfreier (beliebiger) Zugriff (random access)

Von Neumann Architektur

- Kleinste Speichereinheit 1 Bit hat 2 Zustände
  - Strom aus/an, Kondensator geladen/ungeladen, …
  - Zustände werden i.A. mit 0 und 1 bezeichnet
- Paket aus *n* Bits hat 2^n Zustände
  - Mit 2 Speichereinheiten 2^2=4 Zustände
  - Mit 8 Bits 2^8=256 Zustände darstellbar
- 8 Bits = 1 Byte
  - Heutzutage sind Bytes die kleinsten *adressierbaren* Speichereinheiten
    - Kleinere Einheiten müssen aus einem Byte extrahiert werden

Bit 1	Bit 0	
0	0	Zustand 0
0	1	Zustand 1
1	0	Zustand 2
1	1	Zustand 3

- In der Informatik wird mit *kilo* meist $1024=2^{10}$ gemeint
  - 1 kByte = 1024 Byte
  - Mit Mega $1024 \cdot 1024 = 2^{20}$
    - 1 MByte = 1024 kByte
  - Mit Giga $1024 \cdot 1024 \cdot 1024 = 2^{30}$
    - 1 GByte = 1024 MByte
  - Entsprechend kBit, MBit
    - Manchmal auch kB für kByte und kb für kBit (entsprechend MB, Mb, GB, Gb); manchmal auch KB, Kb, „großes“ K = 1024.
  - Widerspricht eigentlich dem normierten Sprachgebrauch, in dem k immer 1000, und M immer 1000000 bezeichnen muss
    - Manche Festplattenhersteller benutzen diesen normierten Sprachgebrauch
      - Damit ist die Speicherkapazität scheinbar etwas höher ☺

Terabyte = 2^{40}

Petabyte = 2^{50}

- Weitere wichtige Einheiten
  - Wort (word) = 4 Byte = 32 Bit
  - Halbwort (short) = 2 Byte = 16 Bit
  - Doppelwort (long, double) = 8 Byte = 64 Bit
- Heutige Rechner können meist 32Bit oder 64Bit auf einmal verarbeiten
  - PCs mit Intel Pentium noch 32 Bit
    - Itanium 2 Prozessor schon mit 64 Bit
  - Bei RISC Workstations meist schon Übergang zu 64Bit vollzogen

- Mit Speicheradressen von 32Bit Länge können 2^{32}Byte=$4 \cdot 2^{30}$Byte=4GByte=4096 MByte adressiert werden
  - Mit 64Bit können 2^{64}Byte=2^{34}GByte≈10^{11}GByte adressiert werden
- Wenn ein Wort aus den Bytes mit den Adressen *n*, *n*+1, *n*+2, *n*+3 besteht, dann ist *n* die *Adresse* des Worts
  - In einem Speichermodul sind die Werte von *n*, die durch 4 teilbar sind, die natürlichen Grenzen für Worte
  - An solchen Stellen beginnende Worte sind an den **Wortgrenzen** (word boundary) **ausgerichtet** (aligned)

Binärcode

- In (heutigen) Computern kann also alles immer nur in der Form von Bitmustern gespeichert werden
- Eine Abbildung von
  - gewöhnlichem Klartext
  - in ein **Bitmuster** (bit pattern)

  nennt man einen **Binärcode** (binary code)
- Je nach dem *Typ* der Daten (Zahlen, Schriftzeichen, Befehle) benutzt man einen anderen Binärcode
- Bei **Kenntnis** des Typs kann man ein Bitmuster dekodieren und seinen Sinn erschließen
  - Verwechselt man den Typ, bekommt das Bitmuster eine ganz andere Bedeutung

- Da wir Menschen Dinge gerne mit Namen benennen statt mit numerischen Adressen, kennt jede Programmiersprache das Konzept einer **Variable** (*variable*) als abstraktes Analogon zu einer Speicherstelle
- Eine Variable hat einen symbolischen **Namen** (name),
  - hinter dem eine Adresse verborgen ist,
  - und der **Wert** (value) der Variable ist der Wert des dort gespeicherten Bitmusters
    - Um diesen erschließen zu können, hat die Variable einen **Typ** (type), der bei ihrer Vereinbarung angegeben werden muss

- In jeder Programmiersprache gibt es einige fest eingebaute elementare (Daten)Typen, wie etwa
  - **char** (Schriftzeichen, character),
  - **int** (endlich große ganze Zahlen, integer) oder
  - **float** (endlich große Gleitkommazahlen, floating point numbers)
- Jedem fundamentalen Typ entspricht ein Code, der jedem möglichen Wert des Typs ein Bitmuster einer festen Länge (z.B. 1 word für 1 int) zuordnet
  - char → ASCII oder UNICODE;
  - int → Dualzahlen im Zweierkomplement
  - float → IEEE 754

- Auch Programme können als Daten aufgefasst und wie solche gespeichert werden
- Programme im **Quelltext** (*source code*) sind einfach Texte in einer Programmiersprache wie Java
  - Sie bestehen also aus Schriftzeichen
    - Genauer: Der Typ des Binärcodes ist char
- Programme in **Objektcode** (*object code*) bestehen aus Befehlen, die in der spezifischen Sprache eines Prozessor-Typs geschrieben sind
    - Typ des Binärcodes also abhängig vom Prozessor-Typ
      - Beachte die unterschiedliche Bedeutung des Wortes *Typ* ☺

Prozessor und Programm-Ausführung

- Daten und Programm gemeinsam im Speicher
- Programm besteht aus Abfolge von Befehlen
- Jeder Befehl als Bitmuster codiert (*binary*)
- Fundamentaler Instruktionszyklus (→Steuerwerk):
  1. Hole nächsten Befehl in den Prozessor
  2. Decodiere den Befehl (Umsetzung in Steuersignale)
  3. Hole ggf. Operanden aus dem Speicher in Register
  4. Führe Operation aus (→ ALU)
  5. Wiederhole ab Schritt 1.

- CPU in VLSI Technik
  - VLSI=very large scale integration
- Transistoren als Grundstruktur
  - Größenordnung 0.1 Micron = 100 nm =
    = 1/10 000 000 m
- Moore´s Law: Anzahl Transistoren/Chip verdoppelt sich in jeweils 18 Monaten
  - Pentium II: 7 Mill. Transistoren
  - Itanium-2: 220 Mill. Transistoren
  - Phys. Grenze (1 Transistor besteht aus wenigen Atomen) um 2020

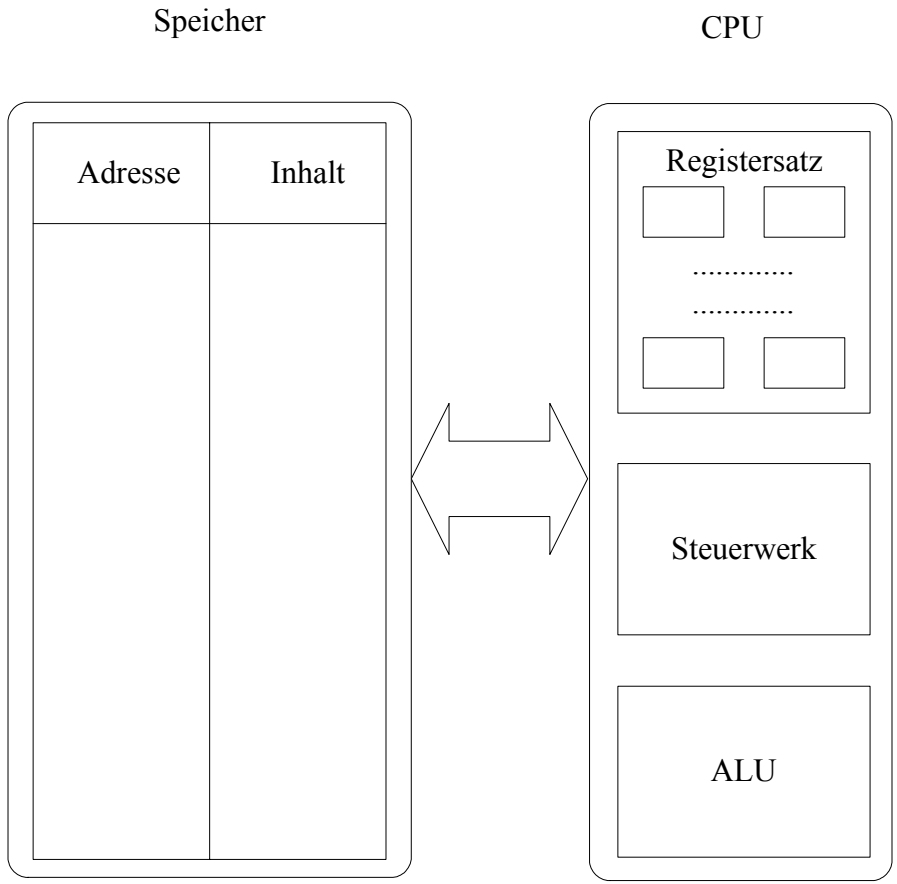
Ein paar Takte zur Geschwindigkeit ...

- Chips sind „getaktet“: Pro Takt wird eine einfache Operation ausgeführt
  - 2 GHz: 1 Takt = 0.5 ns (heute bis etwa 4 GHz)
  - Lichtstrecke bei 0.5 ns = 15 cm
  - Im Abstand von 1.5 m „beobachten“ wir am Chip also die vergangenen Zustände von vor 9-10 Takten!
- Komplexe Operationen brauchen mehrere Takte (z.B. Multiplikation, externe Operanden)
- Partielle Lösung: pipelining
  - Neues Problem: bei Sprüngen alles umsonst
  - Lösung hierzu: viele Register, schnelle Zwischenspeicher (Caches)

Binärdarstellung elementarer Datentypen

….kennen Sie bereits

- Generelle von-Neumann Architektur



Prozessor und Programmausführung 1

- Prozessor = Steuerwerk + arithmetisch-logische Einheit (ALU) + Register
- Steuerwerk: holt aus Speicher Befehle (=Bitmuster) und interpretiert sie
  - es setzt sie in elektrische Signale um, die die ALU und den Datentransport im Prozessor steuern
- Register: Plätze mit sehr schnellem Zugriff zur lokalen Zwischenspeicherung von Daten
- ALU: führt Operationen zur Bearbeitung von Daten aus (insbes. Verknüpfungen +, DIV, REM, Boolesche s.u.)

Von Neumann Architektur

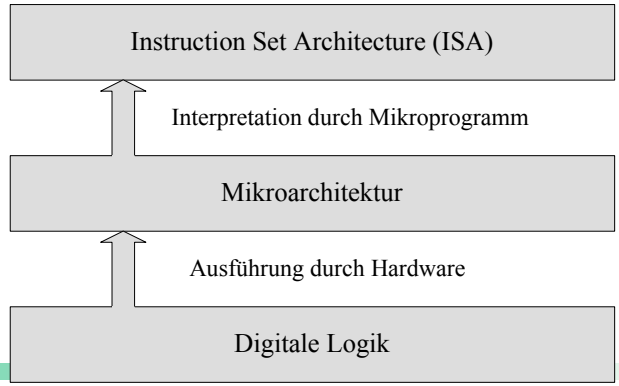
- Daten und Programm gemeinsam im Hauptspeicher
- Programm = Folge von Instruktionen (Befehlen), codiert als Bitmuster
- Befehl: Operationscode (OP-Code) plus Operanden
- Befehle sind in Maschinensprache
  - CISC = complex instruction set computer
  - RISC = reduced instruction set computer
- Fundamentaler Instruktionszyklus (unter Kontrolle des Steuerwerks) zur Programm-Ausführung
  - Befehlszähler (PC): spezielles Register zur Speicherung der Adresse des aktuellen Befehls
  - Instruktionsregister (IR): speichert auszuführenden Befehl

Fundamentaler Instruktionszyklus

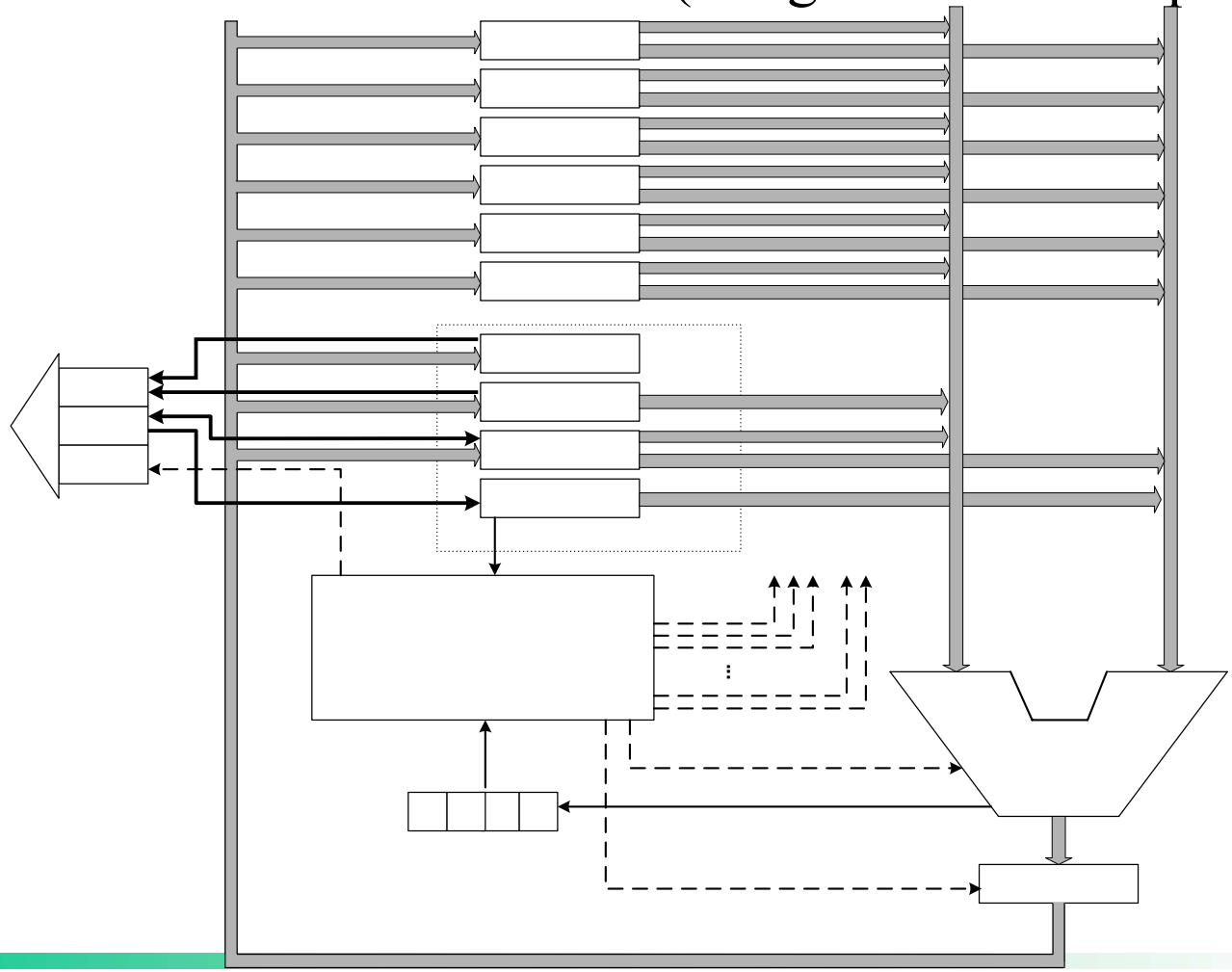
1. **Fetch**: Hole den Befehl, dessen Adresse im Befehlszähler steht, aus dem Speicher in das Befehlsregister
2. **Increment**: Inkrementiere den Befehlszähler (damit verweist er normalerweise auf die nächste auszuführende Instruktion)
3. **Decode**: Dekodiere die Instruktion: setze den OP-Code in elektrische Steuersignale um.
4. **Fetch operands**: Falls nötig, hole die Operanden aus den im Befehl bezeichneten Stellen des Speichers in ein Register
5. **Execute**: Führe die Instruktion aus, i.a. durch die ALU. (Bei einem Sprung wird neuer Wert in den Befehlszähler geschrieben.)
6. **Loop**: Wiederhole ab Schritt 1.

- Befehlbeispiele
  - **LOAD**: Lade Daten von Speicher in Register
  - **STORE**: Schreibe Daten von Register in Speicher
  - **OPERATION:** Verknüpfe zwei Register, Ergebnis in drittes Register (z.B. ADD), wird von ALU ausgeführt
  - **JUMP**: springe an eine (Befehls-)Adresse in Speicher (lade nächsten Befehl von dieser Adresse)
  - **CONDITIONAL JUMP**: springe nur, wenn bestimmtes Register gleich NULL

- **ISA**: Was der (Assembler-)Programmierer sieht (sichtbare Register und mögliche Operationen)
- **Mikro-Architektur**: ALU, Datenpfade, verborgene Register, Details und Zwischenschritte der Ausführung
- **Digitale Logik**: UND- , ODER-, NOT- Gatter: Logik der digitalen Schaltungen



- Mikro-Architektur IMiC (Integer Micro Computer)



IMiC Instruktionssatz (Teilmenge 1)

(Ergänzung zu [KW:02, Kap 2]. Autor: W. Küchlin, Stand 30.10.03)

```
Instruction format    //  Explanation                                   ; code layout (bits)
-------------------------------------------------------------------------------------------
LOAD   R  ADDR    //  Load Word at ADDR into register R    ; [8 | 8 | 16] = 32
STOR   R  ADDR    //  Store Register R at address ADDR     ; [8 | 8 | 16] = 32

BLOD   R  ADDR    //  Load byte at ADDR into low byte of R; [8 | 8 | 16] = 32
BSTR   R  ADDR    //  Store low byte of R at ADDR          ; [8 | 8 | 16] = 32

OP     Rc  Ra  Rb //  Rc = Ra OP Rb                        ; [8 | 8 | 8 | 8] = 32
OPI    Rc  Ra  V8 //  Rc = Ra OP V8, "immediate"           ; [8 | 8 | 8 | 8] = 32

LODI   R   V16    //  Load register R with V16, "immediate" ; [8 | 8 | 16]  = 32
-------------------------------------------------------------------------------------------
```

Abbreviations:
R is one of R0 | R1 | R2 | R3 | R4 | R5
ADDR is any 16-bit address.
OP is one of ADD | SUB | MUL | DIV | REM
V8 is any 8bit "immediate" value
V16 is any 16 bit "immediate" value

- Jede Instruktion in 32 Bits codiert
  - 1 Byte Operationscode (OP-Code, OPC)
  - 3 Bytes Operanden
    - 1 Byte Registercode
    - Ggf. 16 bit Adresse
- Binäre Form der Instruktionen: Maschinensprache (ISA)
- Assembler-Form abstrakter, leichter lesbar, symbolische Namen, dezimale Zahlen etc. (im Beispiel kaum Unterschied)
- Steuereinheit schaltet
  - Datenpfad
  - Funktion der ALU

  in Abhängigkeit von OP-Code und Operanden

- LOAD *R* *Addr*
  1. Transferiere *Addr* ins MAR (Memory Address Register)
  2. Signalisiere Lesewunsch auf MBus Steuer-leitungen und übertrage MAR auf die Adressleitungen des MBus
  3. Memory produziert Inhalt von *Addr* auf MBus. Übernehme MBus Daten ins MDR (Data Register)
  4. Transferiere (Inhalt von) MDR durch die ALU nach *R*

- STOR *R* *Addr*
  1. Transferiere *Addr* ins MAR (mem. addr. register)
  2. Transferiere *R* ins MDR (memory data register)
  3. Signalisiere Schreibwunsch auf MBus Steuer-leitungen, transferiere MAR und MDR auf MBus
  4. Memory übernimmt Bits vom Datenteil des MBus und speichert sie an die im Adressteil übermittelte Adresse.

- *OP Rc Ra Rb*
  1. Transferiere Inhalt von *Ra* in linken ALU Eingang und Inhalt von *Rb* in rechten ALU-Eingang
  2. Signalisiere der ALU, die Operation auszuführen, die dem Opcode *OP* entspricht
  3. Transferiere Ergebnis über C-Bus in *Rc*

- *OP*I *Rc Ra V8* (OP with “immediate” arg)
  1. Transferiere Inhalt von *Ra* in linken ALU Eingang und Inhalt des low byte des Instruktionsregisters IR in rechten ALU-Eingang
  2. Signalisiere der ALU, die Operation auszuführen, die dem Opcode *OP* entspricht
  3. Transferiere Ergebnis über C-Bus in *Rc*

- LODI *R V16* (“immediate” Load)
  1. Transferiere Inhalt der beiden lower bytes des Instruktionsregisters IR in rechten ALU-Eingang
  2. Signalisiere der ALU, die Bits am rechten Eingang zum Ausgang durchzureichen
  3. Transferiere Ergebnis über C-Bus in *R*

IMiC Assembler

Beispiel 1: Ausführung einer Addition z = x + y.

Gegeben: Wert von x im Speicher an Adresse 512
Wert von y im Speicher an Adresse 516
Gesucht: Wert von z, im Speicher an Adresse 508 abzulegen.

```
Instruction              //  Explanation
-------------------------------------------------------------------------------------------
LOAD   R0  512      //  Load (value of) x into R0
LOAD   R1  516      //  Load (value of) y into R1

ADD    R2  R0  R1  //  R2 = R0 + R1, i.e. (R2 = x + y)

STOR   R2  508      //  Store (value of) R2 into location of z
```

Bemerkung: alternativ geht auch

```
ADD    R0  R0  R1  //  R0 := R0 + R1, i.e. (R0 = x + y)
```

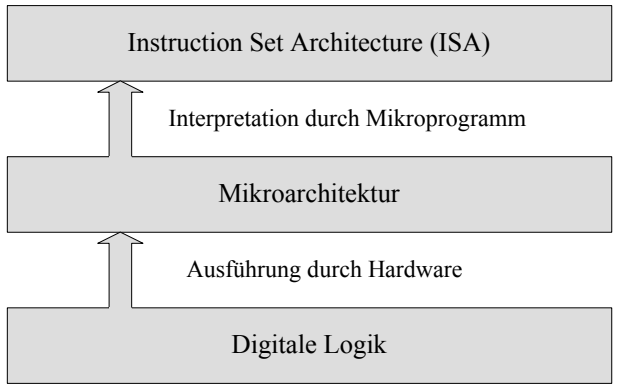
IMiC Assembler

Beispiel 2: Konversion eines ASCII Ziffernzeichens in entsprechende Dualzahl

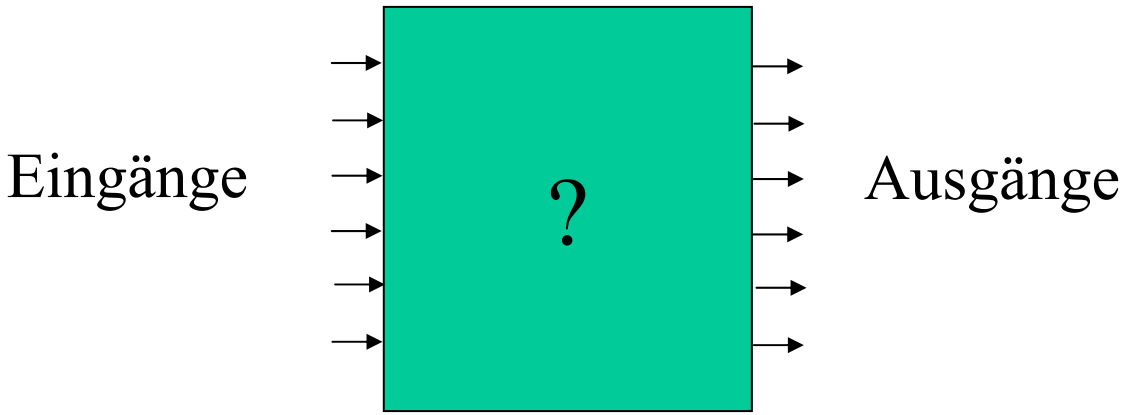
Gegeben: Wert von *a* im Speicher an Adresse 512, Wert ist ASCII Ziffer
Gesucht: Wert von *d*, im Speicher an Adresse 516 abzulegen.

```
Instructions            //  Comment
-------------------------------------------------------------------------------
BLOD   R0  512      //  Load (value of) a into R0
SUBI   R0  R0  48   //  R0 = a – 0x30
STOR   R0  516      //  Store (value of) R0 into location of d
```

- **Digitale Logik**: UND- , ODER-, NOT- Gatter: Logik der digitalen Schaltungen
- Mathematisch modelliert durch **Boole'sche Algebra**
- **Wie** realisiert man eine binäre Addition?
- **Wie** schaltet man Datenpfade durch?



- Wie sind Schaltungen im Computer aufgebaut?
- Es kommen nur die Signale 0 und 1 vor
- Signale 0 und 1 auf den Eingängen müssen wieder in Signale 0 und 1 auf den Ausgängen abgebildet werden
- Abbildungen heißen **Schaltfunktionen** (*switching functions*)



Hier: Schaltzustände bistabiler Schaltelemente

- Bistabile **Schaltelemente**:
  Schalter, Relais, Dioden, Transistoren, etc.

- Zwei stabile **Schaltzustände**:
  offen/geschlossen, nichtleitend/leitend, hohes/niedriges Potential, etc.

- Den Zuständen werden die sogenannten **Schaltwerte** 0 und 1 zugeordnet
  $\Rightarrow$ M = IB = {0,1}

- 0 und 1 nennt man auch **Schaltkonstanten**

- Darstellung mit Symbolen der Schaltertechnik:

Schaltelement	**Schaltzustand**	**Schaltwert**
	offen	0
	geschlossen	1

Definition:

a) Eine Variable, die nur endlich viele Werte annehmen kann, heißt auch **Schaltvariable**.

   Eine Variable heißt **binär**, wenn Sie genau zwei Werte annehmen kann.

b) Boolesche Funktionen f: $IB^n \rightarrow IB$ werden auch als **(binäre) Schaltfunktionen** bezeichnet.

c) Die technische Realisierung einer binären Schaltfunktion bezeichnet man als **Schaltung**.

Darstellung der Schaltkonstanten durch eine ständig offene bzw. ständig geschlossene Schaltung:

0

(ständig offen)

1

(ständig geschlossen)

(Interpretation des Schaltbildes:

Links liege stets eine 1 an

Rechts (am Ausgang) ergibt sich der Wert der Schaltung)

Verknüpfungen

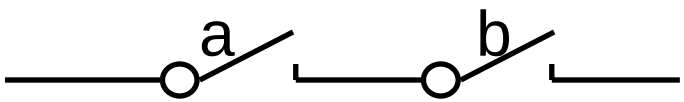
Wahl der Verknüpfungszeichen

$\wedge$, bislang $\cdot$ (Boolesches Produkt, Konjunktion)

$\vee$, bislang + (Boolesche Summe, Disjunktion)

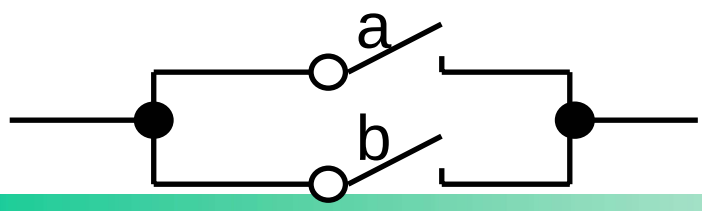
$\neg$, bislang ‘ (Boolesches Komplement)

Konjunktion realisiert durch **Reihenschaltung**



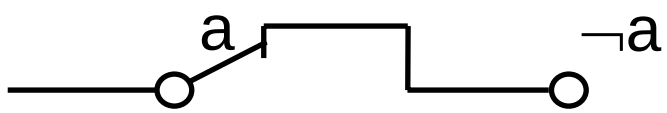
Schaltfunktion (a,b) $\rightarrow$ a $\wedge$ b

Disjunktion realisiert durch **Parallelschaltung**



Schaltfunktion (a,b) $\rightarrow$ a $\vee$ b

Komplement realisiert durch **Ruhekontaktschaltung**



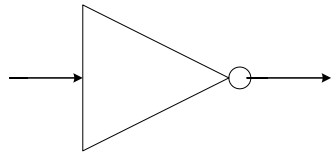
Schaltfunktion $a \rightarrow \neg a$

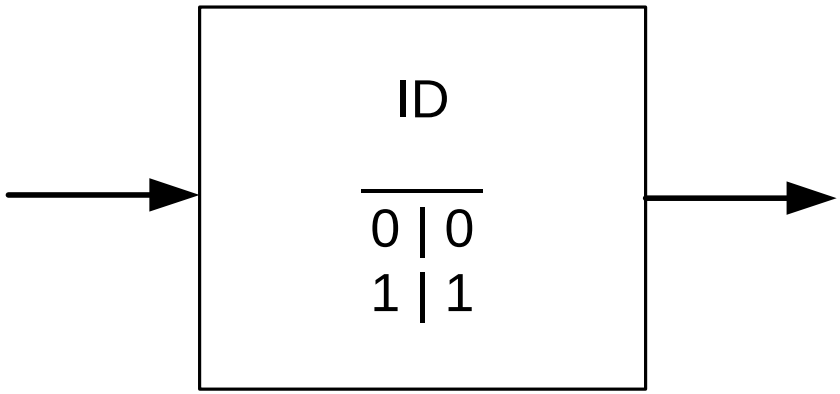
Wie man leicht überlegt, gilt

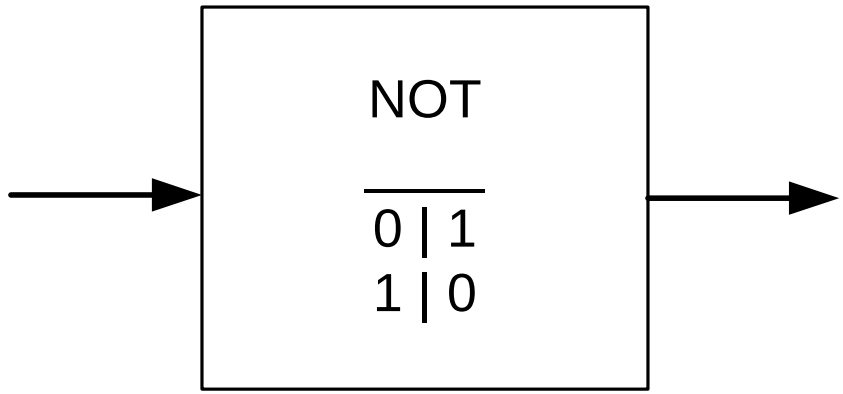
Satz:

Die algebraische Struktur (IB; $\wedge$, $\vee$, $\neg$), genannt **Schaltalgebra**, ist eine Boolesche Algebra.

- Spezialfall Boole´sche Funktionen: nur ein Ausgang
- Einfachster Fall: ein Eingang, ein Ausgang
- 4 mögliche Schaltfunktionen, NUL, ONE, ID und NOT
- NUL immer 0, ONE immer 1, ID uninteressant
- **NOT** heißt auch Negation, Schaltsymbol:







- Nächster Fall: 2 Eingänge, 1 Ausgang
- 4 mögliche Eingangskombinationen
- Je 2 Ausgangswerte möglich → $2^4 = 16$ mögl. Funktionen
- Einige weniger interessant: NUL, a, NOTa, b, NOTb, …
- Interessant: AND, OR, NAND, NOR, XOR, EQV, IMP

a, b	NUL	NOR		NOTa		NOTb	XOR	NAND	AND	EQV	b	IMP	a		OR	ONE
0 0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0 1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Verknüpfungsbasen

Zunächst:

Betrachtung aller 16 möglichen zweistelligen Schaltfunktionen in der folgenden Schalttabelle:

a b	0011 0101	kDN	Bezeichnung
f_0	0000	0	Nullfunktion
f_1	0001	$a \wedge b$	Konjunktion; AND
f_2	0010	$a \wedge \neg b$	
f_3	0011	$(a \wedge \neg b) \vee (a \wedge b)$	
f_4	0100	$\neg a \wedge b$	
f_5	0101	$(\neg a \wedge b) \vee (a \wedge b)$	
f_6	0110	$(\neg a \wedge b) \vee (a \wedge \neg b)$	Antivalenz; XOR
f_7	0111	$(\neg a \wedge b) \vee (a \wedge \neg b) \vee (a \wedge b)$	Disjunktion; OR

f_8	1000	$\neg a \wedge \neg b$	Peirce-Fkt.; NOR
f_9	1001	$(\neg a \wedge \neg b) \vee (a \wedge b)$	Äquivalenz
f_{10}	1010	$(\neg a \wedge \neg b) \vee (a \wedge \neg b)$	
f_{11}	1011	$(\neg a \wedge \neg b) \vee (a \wedge \neg b) \vee (a \wedge b)$	Implikation $a \leftarrow b$
f_{12}	1100	$(\neg a \wedge \neg b) \vee (\neg a \wedge b)$	
f_{13}	1101	$(\neg a \wedge \neg b) \vee (\neg a \wedge b) \vee (a \wedge b)$	Implikation $a \rightarrow b$
f_{14}	1110	$(\neg a \wedge \neg b) \vee (\neg a \wedge b) \vee (a \wedge \neg b)$	Sheffer-Fkt.; NAND
f_{15}	1111	$(\neg a \wedge \neg b) \vee (\neg a \wedge b) \vee (a \wedge \neg b) \vee (a \wedge b)$	Einsfunktion

Frage:

Können Schaltfunktionen auch mit weniger/anderen Operatoren als $\{\wedge, \vee, \neg\}$ dargestellt werden?

Definition:

Eine Menge V von Verknüpfungen heißt **Verknüpfungsbasis** für eine Funktionsmenge F, wenn sich jede Funktion $f \in F$ allein mit Verknüpfungen $v \in V$ darstellen lässt.

Im vorliegenden Fall:

$$F = \{ f_i \mid f_i : IB \times IB \rightarrow IB \ (i=0, \ldots, 15)\}$$

$$V = \{\wedge, \vee, \neg\}$$

Satz:

$V_1 = \{\wedge, \neg\}$ und $V_2 = \{\vee, \neg\}$ sind Verknüpfungsbasen für F.

Beweis: Übung

Definition:

(a) Die **NOR-Funktion** ist die Negation der Disjunktion (**N**OT **OR**)

$\text{NOR}: IB \times IB \rightarrow IB$

$(a,b) \rightarrow \text{NOR}(a, b) = \neg(a \vee b) =: a \downarrow b.$

(b) Die **NAND-Funktion** ist die Negation der Konjunktion (**N**OT **AND**)

$\text{NAND}: IB \times IB \rightarrow IB$

$(a,b) \rightarrow \text{NAND}(a, b) = \neg(a \wedge b) =: a \mid b.$

Satz:

$V_1 = \{ \downarrow \}$ und $V_2 = \{ | \}$ sind Verknüpfungsbasen für F.

Beweis:

(i) Rückführung von $\{\vee, \neg\}$ auf $\{ \downarrow \}$:

„ $\neg$“: $a = a \vee a \Rightarrow \neg a = \neg(a \vee a) = a \downarrow a = \mathrm{NOR}(a, a)$

„ $\vee$“: $a \vee b = (a \vee b) \wedge (a \vee b) = \neg(\neg(a \vee b) \wedge (a \vee b))$

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

$= (a \downarrow b) \downarrow (a \downarrow b)$

$= \mathrm{NOR}(\mathrm{NOR}(a, b), \mathrm{NOR}(a, b))$

(ii) Rückführung von $\{\vee, \neg\}$ auf $\{ \mid \}$:

„ $\neg$“: $a = a \wedge a \Rightarrow \neg a = \neg(a \wedge a) = a \mid a = \mathrm{NAND}(a, a)$

„ $\vee$“: $a \vee b = (a \wedge a) \vee (b \wedge b) = \neg(\neg(a \wedge a) \vee (b \wedge b))$

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

$= (a \mid b) \mid (a \mid b)$

$= \mathrm{NAND}(\mathrm{NAND}(a, a), \mathrm{NAND}(b, b))$

Bemerkungen:

Verknüpfungsbasen:

- $\{\wedge, \vee, \neg\}$ (AND, OR, NOT)
- $\{\wedge, \neg\}$ (AND, NOT)
- $\{\vee, \neg\}$ (OR, NOT)
- $\{\downarrow\}$ (NOR
- $\{|\}$ (NAND)

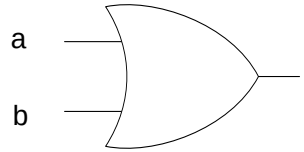
Eine Verknüpfung zur Darstellung aller zweistelligen Schaltfunktionen ausreichend.

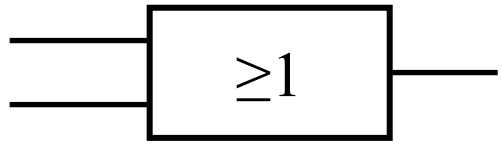
Dadurch: einfache technische Realisierung

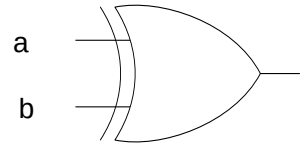
- Es muss nur ein Verknüpfungsglied gebaut werden (relativ häufig Realisierung nur mit NOR)
- Aber: i.d.R. erhöhte Anzahl der Verknüpfungsglieder, weshalb doch meist verschiedene verwendet werden

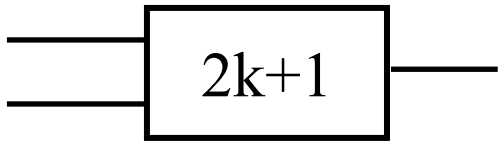
- Die wichtigsten (Schalt-)Gatter: AND, OR, NOT, (XOR, NAND, NOR)
- Schaltbilder nach IEEE Standard (anglo-amerikanisch); DIN 40700

a,b	OR	XOR
0 0	0	0
0 1	1	1
1 0	1	1
1 1	1	0

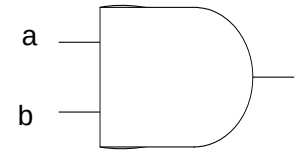


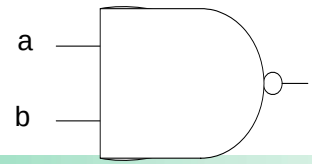


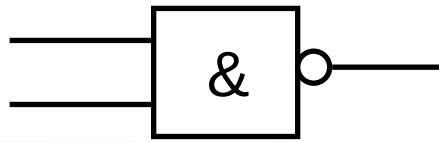




a,b	AND	NAND
0 0	0	1
0 1	0	1
1 0	0	1
1 1	1	0

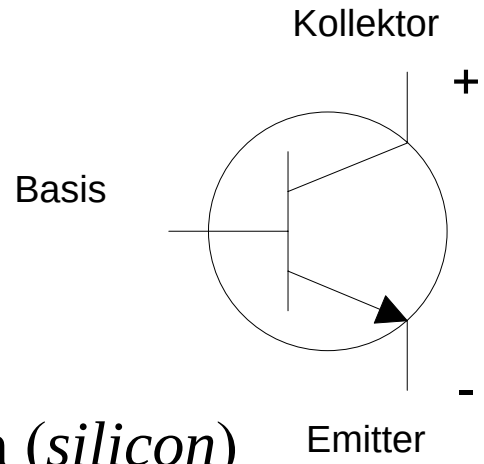






Exkurs: Bausteine Digitaler Logik (Ebene der Elektrotechnik)

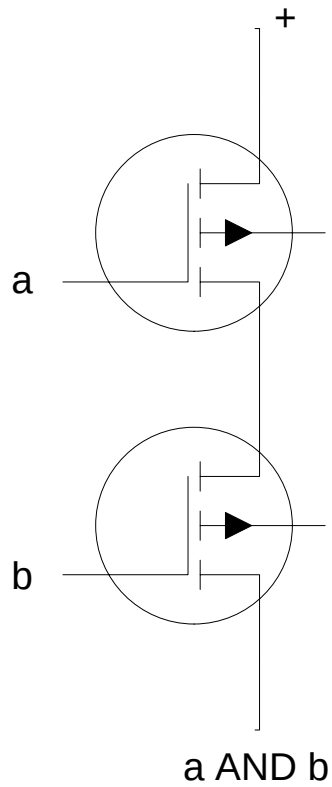
- Logik-Bausteine (**Gatter**, *gates*) heute durch **Transistoren** realisiert
- Transistor: elementarer elektronischer **Schalter**
  - schaltet Strom von **Kollektor** zu **Emitter** durch Spannung an **Basis**
  - **drain, source, (**transistor) **gate**
  - (Elektronenfluss umgekehrt zu Stromfluss)



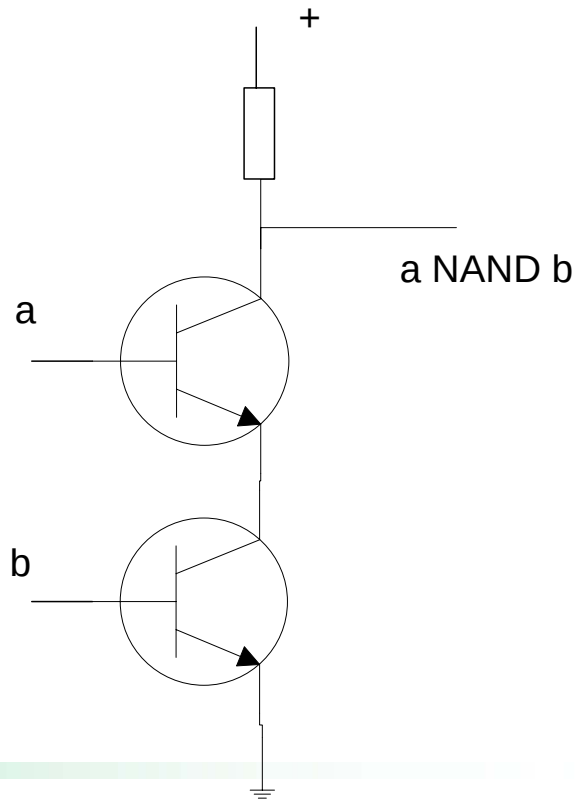
- Transistoren aus Halbleitermaterial Silizium (*silicon*)
- Halbleiter können isolieren oder leiten
- CMOS Feldeffekt-Transistoren induzieren Elektronen in Basis durch Kondensator-Effekt (verlustfrei)
- Bipolare Trans. bringen Elektronen durch (schwachen) Strom in Basis

Exkurs: Bausteine Digitaler Logik (Ebene der Elektrotechnik)

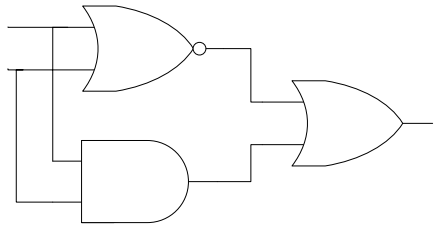
- Die wichtigsten Gatter: AND, OR, NOT, (XOR, NAND, NOR) auf *einfache* Weise realisierbar
- AND mit MOS-FET



- NAND (Spezialfall: NOT) bipolar



- Verschiedene Schaltungen können die selbe Schaltfunktion realisieren. Beispiele mit AND, OR, NOT:
  - NAND(a,b) = NOT(AND(a,b))
  - NOR(a,b) = NOT(OR(a,b))
  - EQV(a,b) = OR(AND(a,b), NOR(a,b))
  - …



- Beweis durch Vergleich der Funktionstabellen

a,b	NOR	XOR	NAND	AND	EQV	OR
0 0	1	0	1	0	1	0
0 1	0	1	1	0	0	1
1 0	0	1	1	0	0	1
1 1	0	0	0	1	1	1

a,b	AND	NOR	OR(AND,NOR)	EQV
0 0	0	1	1	1
0 1	0	0	0	0
1 0	0	0	0	0
1 1	1	0	1	1

- Jede Schaltfunktion ist Kollektion Boole'scher Funktionen
- Jede Boole'sche Funktion f kann durch Kombination von AND, OR, NOT realisiert werden (alternativ: NAND bzw. NOR)
- Beweis durch Einsicht:
  - f ist vollständig charakterisiert dadurch, wo sie 1 wird
  - jede 1-Stelle durch AND/NOT-Ausdruck beschreibbar
  - f durch OR-Ausdruck über die 1-Stellen charakterisiert

a,b	EQV	AND(NOT(a),NOT(b))	AND	OR(AND(NOT(a),NOT(b), AND(a,b))
0 0	**1**	**1**	**0**	**1**
0 1	**0**	**0**	**0**	**0**
1 0	**0**	**0**	**0**	**0**
1 1	**1**	**0**	**1**	**1**

- Arithmetik mittels Schaltfunktionen realisierbar
- Intuition: endlich viele Ziffern → endlich viele Fälle
- Beispiel: Eine Spalte der Addition $c = a+b$
  - Übertrag von rechts: c_{in} (*carry in*); Übertrag nach links c_{out}
- Kompletter Addierer ist Reihe davon: *ripple carry adder*

a, b, Cin	Cout	c	((a XOR b) AND Cin) OR (a AND B)	(a XOR b) XOR Cin
0 0 0	0	0	0	0
0 1 0	0	1	0	1
1 0 0	0	1	0	1
1 1 0	1	0	1	0
0 0 1	0	1	0	1
0 1 1	1	0	1	0
1 0 1	1	0	1	0
1 1 1	1	1	1	1

Beispiele: (für Schaltnetze)

(A) Halbaddierer (HA)

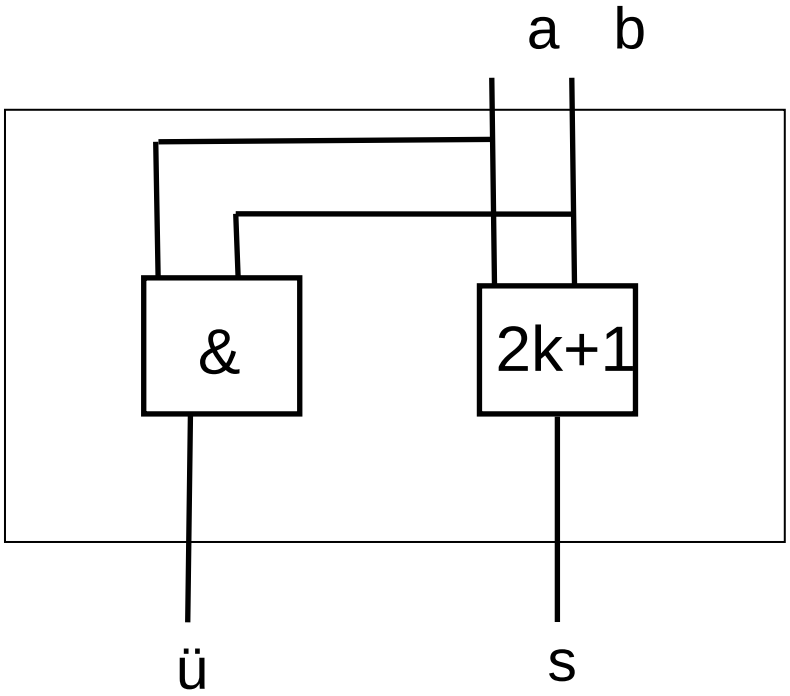
- Berechnet die Summe zweier Dualziffern a und b
- Darstellung der Summe in zwei Dualziffern s (Summe modulo 2) und ü (Übertrag)
- **Schalttabelle:**

a	b	s	ü
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

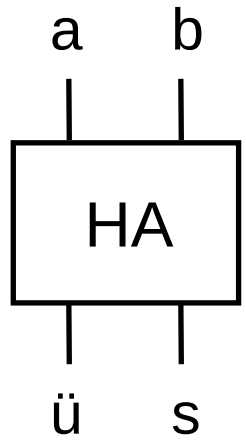
$ü(a,b) = a \wedge b$

$s(a,b) = a \oplus b$

- Aufbau eines HA



Symbol für HA



(B) **Volladdierer** (VA)

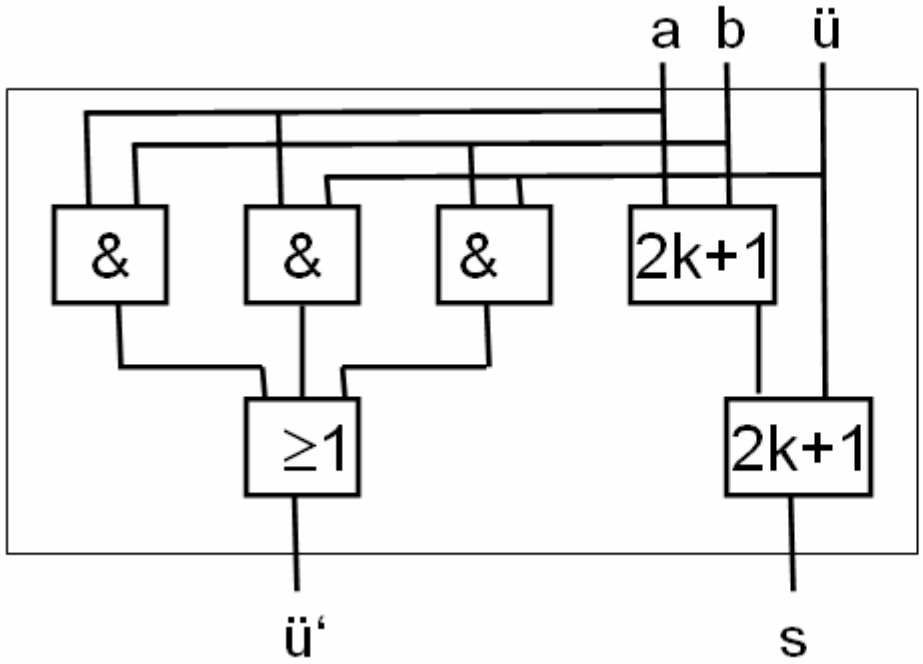
- Berechnet die Summe zweier Dualziffern a und b unter Berücksichtigung eines Übertrags ü (aus voriger Stelle)
- Darstellung der Summe in zwei Dualziffern s und ü‘
- **Schalttabelle:**

a	b	ü	s	ü
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

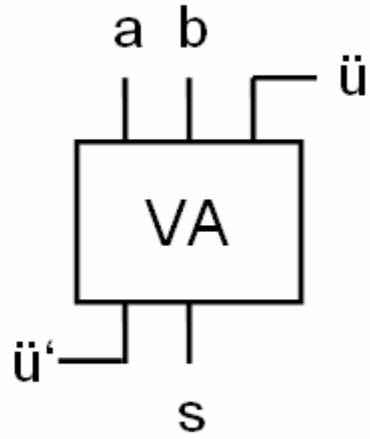
$s = (a \oplus b) \oplus ü$

$ü' = (a \wedge b) \vee (a \wedge ü) \vee (b \wedge ü)$

- Aufbau eines VA



Symbol für VA

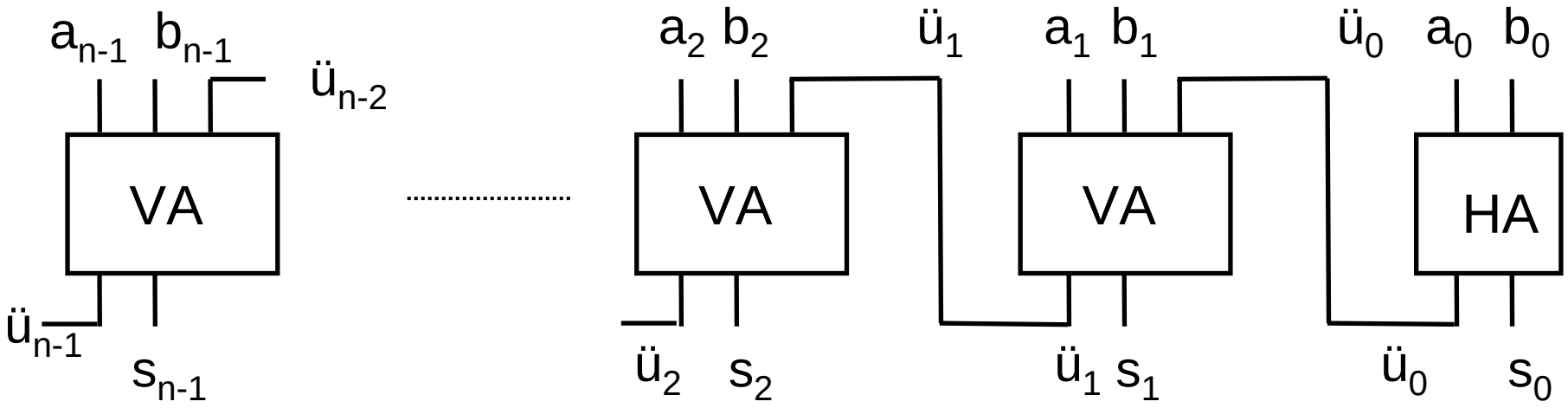


Verwendung von Halb-/Volladdierern z.B. in einem **Addierer mit durchlaufendem Übertrag:**

Addition zweier n-stelliger Dualzahlen

$a = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$ und $b = (b_{n-1}, b_{n-2}, \ldots, b_1, b_0)$

zur Summe $(\ddot{u}_{n-1}, a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$



Dieser Addierer heißt auch Ripple-Carry Adder, da der Übertrag von der niedrigst- zur höchstsignifikanten Dualstelle durchlaufen kann.

$\Rightarrow$ Lange Schaltzeit (linear in der Wortlänge)

Bemerkung:

Fügt man in jede „Übertragsleitung“ ein Verzögerungselement ein, so erhält man einen „getakteten Ripple-Carry Adder“, in den man die n Bitpaare der Summanden in n aufeinanderfolgenden Takten eingibt.

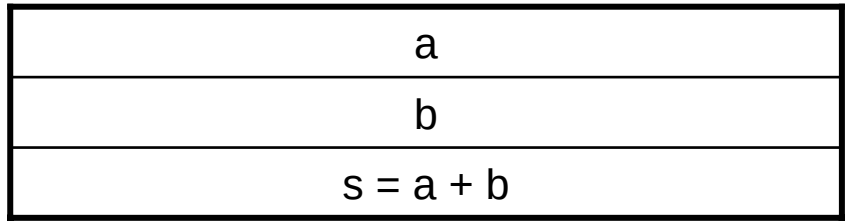
$\Rightarrow$ Jede Addition dauert zwar n Takte, aber man kann in jedem Takt eine neue Addition beginnen, d.h. es können bis zu n Additionen gleichzeitig jeweils um einen Takt versetzt ausgeführt werden.

$\Rightarrow$ Für k Additionen nur k+n-1 Takte!

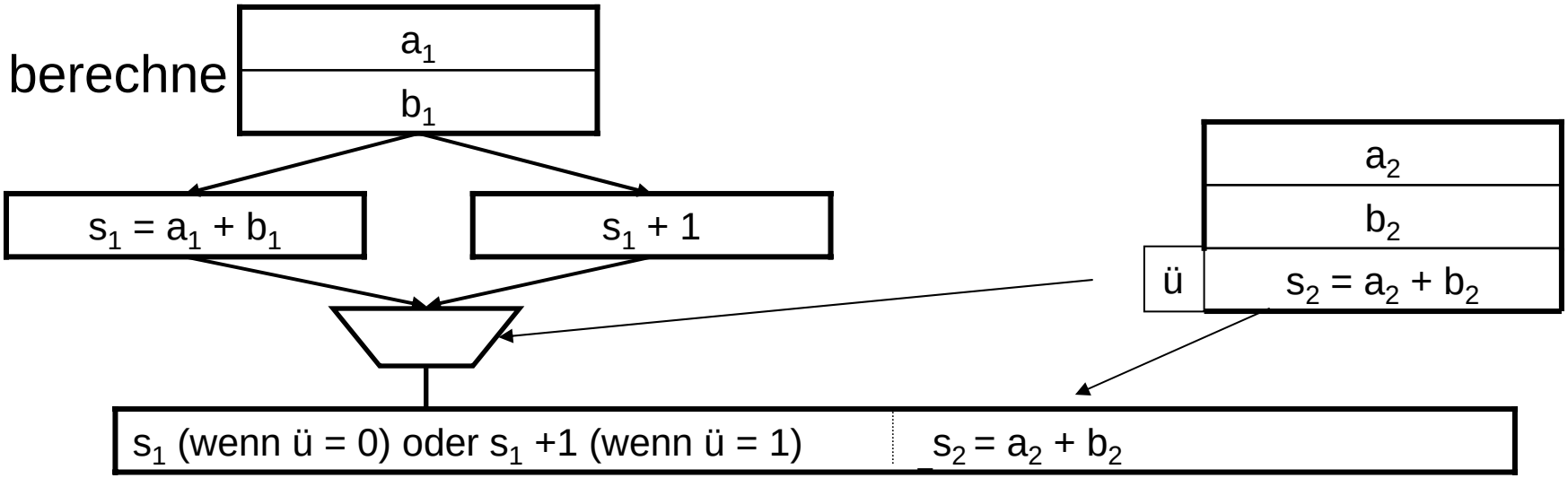
Alternativen:

Carry-Select Adder

Divide-and-Conquer Ansatz:



An Stelle von



Dabei sei a_1 die linke und a_2 die rechte Hälfte von a

(analog für b und s)

$\Rightarrow$ Rekursive Anwendung ergibt eine Rechenzeit in O(log n)!
(aber höheren Flächenaufwand)

Carry-Look-Ahead Adder

Berechnet Übertrag vor Addieren

Nutze aus, dass jedes Bitpaar entweder

Einen Übertrag erzeugt (1,1) – Funktion „generate“ $g_i = a_i \wedge b_i$

Einen Übertrag weiterleitet (1,0) oder (0,1) – „propagate“ $p_i = a_i \vee b_i$

$$\ddot{u}_0 = a_0 \wedge b_0 = g_0$$

Für i > 0: $\ddot{u}_i = a_i \wedge b_i \vee a_i \wedge \ddot{u}_{i-1} \vee b_i \wedge \ddot{u}_{i-1} =$

$$a_i \wedge b_i \vee (a_i \vee b_i) \wedge \ddot{u}_{i-1} =$$

$$g_i \vee p_i \wedge \ddot{u}_{i-1}$$

$\ddot{u}_1 = g_1 \vee p_1 \wedge g_0$

$\ddot{u}_2 = g_2 \vee p_2 \wedge \ddot{u}_1 = g_2 \vee p_2 \wedge (g_1 \vee p_1 \wedge g_0) = g_2 \vee p_2 \wedge g_1 \vee p_2 \wedge p_1 \wedge g_0$

$\ddot{u}_3 = g_3 \vee p_3 \wedge \ddot{u}_2 =$

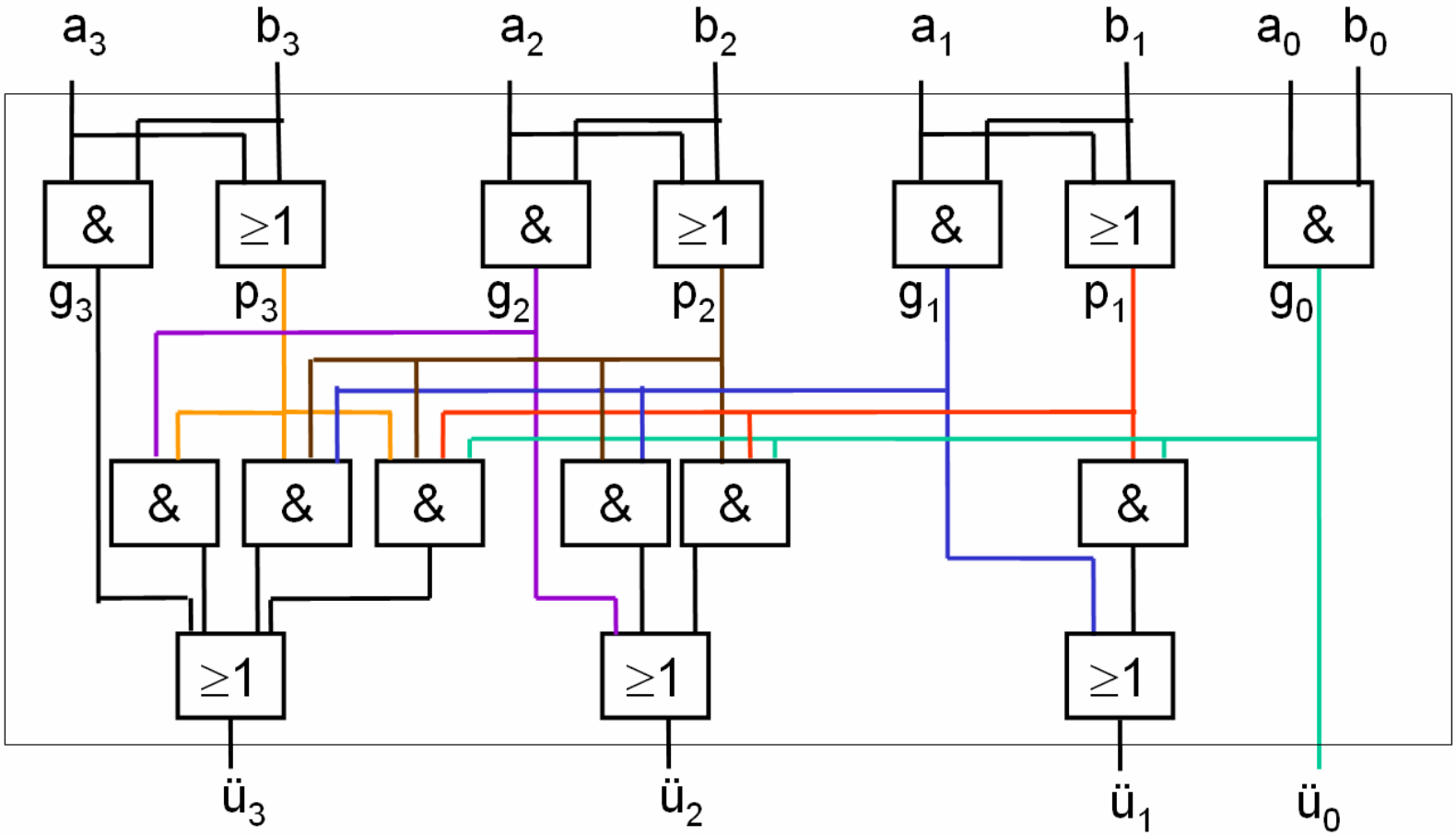
$g_3 \vee p_3 \wedge (g_2 \vee p_2 \wedge g_1 \vee p_2 \wedge p_1 \wedge g_0) = g_3 \vee p_3 \wedge g_2 \vee p_3 \wedge p_2 \wedge g_1 \vee p_3 \wedge p_2 \wedge p_1 \wedge g_0$

Carry-Look-Ahead Adder

$g_i = a_i \wedge b_i$ $p_i = a_i \vee b_i$

$\ddot{u}_0 = g_0$ $\ddot{u}_1 = g_1 \vee p_1 \wedge g_0$

$\ddot{u}_2 = g_2 \vee p_2 \wedge g_1 \vee p_2 \wedge p_1 \wedge g_0$ $\ddot{u}_3 = g_3 \vee p_3 \wedge g_2 \vee p_3 \wedge p_2 \wedge g_1 \vee p_3 \wedge p_2 \wedge p_1 \wedge g_0$



Definition: Sei B eine Menge und +, ^ seien zwei Verknüpfungen auf B und 0, 1 ∈ B zwei feste Elemente. Es gelte:

1. $\vee$ und ^ sind assoziativ und kommutativ.
2. Es gelten die Distributivgesetze
   $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ und $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$.
3. $x \wedge 0 = 0$, und $x \wedge 1 = x$ für alle x,
4. $x \vee 0 = x$, und $x \vee 1 = 1$ für alle x,
5. Zu jedem x gibt es genau ein x' mit $x \wedge x' = 0$ und $x \vee x' = 1$.
   (_′ ist ein einstelliger Operator in Postfix-Schreibweise)

Dann heißt [B; $\vee$,^,′ , 0, 1] (kürzer: **B**) eine **Boolesche Algebra**.

- Boole'sche Schaltfunktionen bilden eine Boole'sche Algebra

  Schaltalgebra [{0,1}; OR, AND, NOT, 0, 1]
- Beweis: rechne Axiome über Funktionstabellen nach
  - endliche Fallunterscheidung, da nur 0 und 1
  - Beispiel: x OR (y AND z) = (x OR y) AND (x OR z)

x,y,z	y AND z	x OR (y AND z)	x OR y	x OR z	(x OR y) AND (x OR z)
0 0 0	0	0	0	0	0
0 0 1	0	0	0	1	0
0 1 0	0	0	1	0	0
0 1 1	1	1	1	1	1
1 0 0	0	1	1	1	1
1 0 1	0	1	1	1	1
1 1 0	0	1	1	1	1
1 1 1	1	1	1	1	1

Weitere Gesetze (Sätze) der Boole´schen Algebra

1. **Doppelte Negation**: $(x')' = x$
2. **Idempotenz:** $x \vee x = x$, und $x \wedge x = x$
3. **Absorption:** $x \vee (x \wedge y) = x$ und $x \wedge (x \vee y) = x$
   $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ und
   $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$.
4. **Implikation:** $(x \Rightarrow y) = x' \vee y$ [in Schaltalgebra ist $\Rightarrow$ die Funktion IMP]
5. **De Morgan** Regeln: $(x \vee y)' = (x' \wedge y')$
   $(x \wedge y)' = (x' \vee y')$

- Mit den Gesetzen der Boole´schen Algebra lassen sich Boole´sche Audrücke in gleichwertige (äquivalente) umformen
- Gleichwertige Ausdrücke stellen dieselbe Schaltfunktion dar
- Dadurch funktional gleichwertiges Ersetzen möglich:
  - **Vereinfachung**: weniger Logik-Gatter
  - **Beschleunigung**: schnellere Schaltungen
  - **Kosten**: billigere/kleinere Bauteile
- Boole´sche Operatoren AND, OR, NOT in Java: &&, ||, !
  - günstigen Java Ausdruck für gewünschte Funktion finden