

# Technische Informatik

**Klausur:** 03.02.2017, 10:00–12:00 Uhr, Hörsaal B

## Computergestützte Tests:

- 09.11. 2016
- 23.11. 2016
- 11.01. 2017

## Ziele dieser Lehrveranstaltung:

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

## Inhaltsverzeichnis

Klausur-Irrelevantes Wissen .....	6
Grundlegende Begriffe und Informationen.....	6
Konvention zur Notation .....	6
Axiome .....	7
Sätze zu Axiomen.....	7
Weitere Gesetze .....	7
Boolesche Funktion.....	8
Definition .....	8
Spezialfall.....	8
Wie viele Boolesche Funktionen gibt es? .....	8
Bezeichnungen für zweistellige Boolesche Funktionen.....	9
Normalformen.....	10
Systematik der Darstellung.....	10
Bildung .....	10
Sätze zu Booleschen Funktionen.....	11
Schaltung .....	12
Realisierung einer (günstigen) Schaltung.....	12
„Universelle“ Gatter .....	13
Minimierung .....	13
Resolutionsregeln.....	13
Karnaugh-Veitch-Diagramme (KV).....	13
Praktische Realisierung .....	15

Programmable Read-Only Memory (PROM).....	16
Programmable Array-Logic (PAL).....	16
Programmable Logic Array (PLA).....	16
Kombinatorische Logik / Schaltnetze.....	16
Dekodierer.....	16
Kodierer.....	16
(De-)Multiplexer.....	17
Sequenzielle Logik.....	18
Schaltwerke.....	18
Rückkopplungen.....	18
Flipflop-Schaltung („Latch“).....	19
RS-Flipflop.....	19
Getaktetes RS-Flipflop.....	19
Flankengesteuertes RS-Flipflop.....	20
Arithmetik.....	20
2-Bit Multiplizierer.....	20
Addition und Subtraktion.....	20
Addition positiver n-stelliger Binärzahlen.....	20
Addition mit Zweierkomplement.....	20
Halbaddierer (Half Adder).....	21
Volladdierer (Half Adder).....	21
Serielles Addierwerk.....	21
Paralleles Addierwerk.....	22
Kombiniertes Addier-/Subtrahierwerk.....	22
ALU / Arithmetisch-logische Einheit.....	23
Multiplikation.....	23
Algorithmus zur Multiplikation positiver Binärzahlen.....	23
Schaltnetz.....	24
Serielles Schaltwerk.....	24
Feldmultiplizierer (array mutliplier).....	25
Optimierungsmöglichkeiten.....	26
Multiplikation negativer Zahlen.....	26
Algorithmus von Booth.....	26
Division (mit Restoring).....	27
Optimierungsmöglichkeiten.....	27
Rechnen mit Nachkommastellen.....	28
Festkomma.....	28
Gleitkomma.....	28

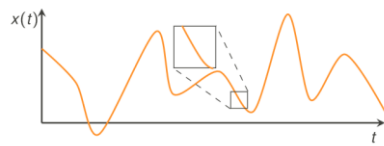
Befehlssatzarchitektur .....	30
ARM-Mikroarchitektur.....	30
Hintergrund.....	30
Registersatz .....	31
Endianness und Alignment .....	32
Kodierung der Instruktionwörter.....	32
Realisierung theoretischer Grundlagen .....	32
Divisionsalgorithmus mit „Restoring“ .....	32
Ausgabe von Zahlensystemen .....	33
Stapelorganisation und Funktionsaufrufe .....	36
Funktionsaufruf .....	36
Assembler .....	38
Vorbereitungen .....	38
Entwicklungsumgebung.....	38
Toolchain .....	38
Assembly Code-Snippets.....	38
Grundgerüst eines Assembly-Programms .....	38
Write-Syscall.....	39
Effiziente Multiplikation mit Konstanten .....	39
Typische Fehlerquellen .....	39
ARM-Befehlsreferenz .....	40
Flags .....	40
Negative (N) .....	40
Zero (Z) .....	40
Carry (C).....	40
Overflow (V).....	40
Bedingungen .....	41
Arithmetische Operationen .....	42
Beschreibung und Information zu den Mnemonics.....	42
Logische und Vergleichsoperationen.....	43
Beschreibung und Information zu den Mnemonics.....	43
Kopier-, Verschiebe- und Sprungoperationen.....	45
Beschreibung und Information zu den Mnemonics.....	45
Speicherzugriffsoperationen .....	48
Beschreibung und Information zu den Mnemonics.....	48
Weiterführende Links und Informationen.....	50
Prozessorarchitektur .....	50
Klassifikationen .....	50

Anbindung des Speichers.....	50
Anbindung der ALU (tatsächlich Speichers).....	50
Komplexität des Befehlssatzes .....	51
Klassifikation nach Art der Parallelverarbeitung.....	51
Intel x86-CISC-Architektur.....	52
Abwärtskompatibilität .....	52
Registersatz .....	53
FPU (Gleitkomma-Einheit) .....	55
Datenparallele Architekturen.....	56
Grafik-Pipeline mit Hardwareunterstützung.....	57
Optimierung für Datenparallelität.....	57
Ein- & Ausgabe .....	58
Touch-Eingabetechnologie .....	58
Optische Berührungsmessung .....	59
Resistive Berührungsmessung .....	59
Kapazitive Berührungsmessung.....	60
Ansteuerung von E/A-Bausteinen.....	62
Ansteuerung über Ports.....	62
Ansteuerung über Register .....	62
Typen von E/A-Registern.....	62
Speicherdirektzugriff.....	63
Unterbrechungsanforderung .....	63
Interrupt-Handler .....	64
Mehrprozessbetrieb auf einem Kern.....	64
Ausgabe (Praxisbeispiel).....	65
Initialisierung .....	65
Speicher.....	66
Speicherhierarchie .....	66
Definitionen und Erklärungen .....	67
Maßeinheiten .....	67
Begriffe/Akronyme .....	67
Dynamischer RAM (Hauptspeicher).....	68
Schreiben einer Zelle .....	68
Lesen einer Zelle .....	69
DRAM-Auffrischung .....	69
Aufbau von DRAM-Bausteinen.....	70
Hauptspeicher aus DRAM-Bausteinen .....	71
Flash-Speicher (Persistenter Speicher) .....	72

Zustandsübergangstabelle.....	73
Vergleich NOR/NAND .....	73
NOR-Flash .....	73
NAND-Flash .....	74
Leistung.....	77
Definition.....	77
Leistungsverhältnis.....	78
Messung der Ausführungszeit.....	78
Amdahlsches Gesetz .....	80
Optimierung des Speicherzugriffs durch Caches .....	80
Der Cache.....	80
Cache-Kohärenz .....	83
Leistungssteigerung durch Cache.....	84
Optimierte Programmierung.....	84
Optimierung der CPU-Nutzung durch Pipelines.....	85
Maschinenbefehlszyklus.....	85
Leistungssteigerung (theoretisch) .....	86
Gründe für Abweichungen in der Praxis (hazards).....	86
Optimierte Programmierung.....	87

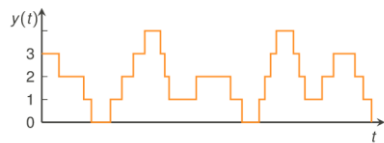
## Klausur-Irrelevantes Wissen

Analogtechnik: kontinuierliche Signale



Zeitlich  
variierendes  
analoges Signal

Digitaltechnik: diskrete (oft binäre) Signale



Wechsel zwischen  
endlich vielen  
festen Werten  
ohne Verzögerung

### Analogrechner

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

### Digitalrechner

- + weniger stör anfällig (z. B. Rauschen)
- + einfacher, modularer Entwurf
- + beliebig hohe Präzision erreichbar
- + exakte Reproduktion/Übertragung von Daten
- oft hoher Flächenbedarf
- hoher Energieverbrauch

## Grundlegende Begriffe und Informationen

Digitale Zustände können zwei Zustände annehmen  $\rightarrow \{1, 0\}$ .

Mehrstellige digitale Zustände werden in Vektorschreibweise angeschrieben:

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

Die Definition von Operatoren und deren Schreibweise kann variieren. Dabei wird die vollständige Bestimmung durch die Wahrheitstabelle genau festgelegt.

BEGRIFF	ERKLÄRUNG
<b>TRANSISTOR</b>	ein elektr. Halbleiter-Bauelement, zum steuern niedriger Spannung
<b>DIODE</b>	ein elektr. Bauelement, das Strom in nur eine Richtung fließen lässt
<b>WIDERSTAND</b>	Es wird eine bestimmte elektr. Stromstärke benötigt zum durchfließen
<b>TAU (<math>\tau</math>)</b>	Dauer zwischen den Zeitpunkten der Überschreitung des 50 %-Pegels an Ein- bzw. Ausgang
<b>STATISCHER NULL-HAZARD</b>	Ein Wechsel am Eingang, kann einen einmaligen, temporären Wechsel des Ausgangs zur Folge haben.

## Konvention zur Notation

Notation nach DIN 66000		Gesprochen	Klammerung [Richtung]	Bindung [Priorität]
Symbol	Verwendung			
$\neg$	$\neg x_1$	nicht $x_1$	—	hoch
$\wedge$	$x_1 \wedge x_2$	$x_1$ und $x_2$	$(x_1 \cdot x_2) \cdot x_3$	mittel
$\vee$	$x_1 \vee x_2$	$x_1$ oder $x_2$	$(x_1 + x_2) + x_3$	mittel
$\overline{\wedge}$	$x_1 \overline{\wedge} x_2$	$x_1$ nand $x_2$	$\overline{(x_1 \cdot x_2)} \cdot x_3$	mittel
$\overline{\vee}$	$x_1 \overline{\vee} x_2$	$x_1$ nor $x_2$	$\overline{(x_1 + x_2)} + x_3$	mittel
$\nleftrightarrow$	$x_1 \nleftrightarrow x_2$	$x_1$ xor $x_2$	$(x_1 \oplus x_2) \oplus x_3$	niedrig
$\leftrightarrow$	$x_1 \leftrightarrow x_2$	$x_1$ äquivalent $x_2$	$(x_1 \equiv x_2) \equiv x_3$	niedrig
$\rightarrow$	$x_1 \rightarrow x_2$	$x_1$ impliziert $x_2$	$(x_1 \Rightarrow x_2) \Rightarrow x_3$	niedrig

## Axiome

Dienen zur Umformung logischer Gleichungen

### Kommutativität

$$x_1 + x_2 = x_2 + x_1$$

$$x_1 \cdot x_2 = x_2 \cdot x_1$$

### Distributivität

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

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

### Sätze zu Axiomen

Abgeleitet aus den Axiomen

### Idempotenz

$$x + x = x$$

$$x \cdot x = x$$

### Assoziativität

$$x_1 + (x_2 + x_3) = (x_1 + x_2) + x_3$$

$$x_1 \cdot (x_2 \cdot x_3) = (x_1 \cdot x_2) \cdot x_3$$

### Absorption

$$x_1 + (x_1 \cdot x_2) = x_1$$

$$x_1 \cdot (x_1 + x_2) = x_1$$

### Neutrale Elemente

$$0 + x = x$$

$$1 \cdot x = x$$

### Komplementäres Element

$$x + \bar{x} = 1$$

$$x \cdot \bar{x} = 0$$

### Substitution

$$x + 1 = 1$$

$$x \cdot 0 = 0$$

### Doppelnegation

$$\overline{\bar{x}} = x$$

## Weitere Gesetze

### Abgeschlossenheit

Boolesche Operationen liefern nur boolesche Ergebnisse.

### Komplementäre Werte

$$\bar{0} = 1$$

$$\bar{1} = 0$$

### Dualität

Für jede aus Axiomen ableitbare Aussage existiert eine duale Aussage, die man erhält, wenn man die Operatoren, sowie die Werte austauscht:

A	B	$A \wedge B$	A	B	$A \vee B$
W	W	W	F	F	F
W	F	F	F	W	W
F	W	F	W	F	W
F	F	F	W	W	W

## De Morgansche Gesetze

$$\overline{x_1 * x_2} \rightarrow \bar{x}_1 + \bar{x}_2$$

$x_1$	$x_2$	$\overline{x_1 * x_2}$	$x_1$	$x_2$	$\bar{x}_1 + \bar{x}_2$
F	F	W	W	W	W
F	W	W	W	F	W
W	F	W	F	W	W
W	W	F	F	F	F

$$\overline{x_1 + x_2} \rightarrow \bar{x}_1 * \bar{x}_2$$

$x_1$	$x_2$	$\overline{x_1 + x_2}$	$x_1$	$x_2$	$\bar{x}_1 * \bar{x}_2$
F	F	W	W	W	W
F	W	F	W	F	F
W	F	F	F	W	F
W	W	F	F	F	F

Grundsätzlich gilt:

*Negation von Termen erfolgt durch Tausch der Operatoren + und \* sowie durch die Komplementierung aller Variablen*

$$\overline{(x_1 + x_2) * x_3} = (\bar{x}_1 * \bar{x}_2) + x_3$$

## Boolesche Funktion

### Definition

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

### Spezialfall

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

1. Eindeutig mit (sortierter) Wahrheitstabelle
2. Kompakter aber nicht eindeutig mit Booleschem Ausdruck (aus Booleschen Variablen und Operationen)

*Jede Schaltfunktion kann durch m Boolesche Funktionen zusammengesetzt werden.*

### Wie viele Boolesche Funktionen gibt es?

Kombination aller  $2^n$  n-Tupel aus  $\{0, 1\}$  der Argumente mit den Booleschen Werten  $\{0, 1\}$ :  $2^n \cdot 2 = 2^{n+1}$

Für  $n = 1$ :  
 $f_0(x) = 0$  Kontradiktion (0-stellig)  
 $f_1(x) = x$  Identität



$f_0(x) = \neg x$  Negation

 $f_0(x) = 1$  Tautologie (0-stellig)

Für  $n = 2$ : 16 zweistellige Boolesche Funktionen:

## Bezeichnungen für zweistellige Boolesche Funktionen

$x_2 =$	0	1	0	1	Term	Bezeichnung	Sprechweise
$x_1 =$	0	0	1	1			
$f_0$	0	0	0	0	0	Nullfunktion	
$f_1$	0	0	0	1	$x_1 x_2$	Konjunktion	$x_1$ AND $x_2$
$f_2$	0	0	1	0	$x_1 \overline{x_2}$	1. Differenz	$x_1$ AND NOT $x_2$
$f_3$	0	0	1	1	$x_1$	1. Identität	
$f_4$	0	1	0	0	$\overline{x_1} x_2$	2. Differenz	NOT $x_1$ AND $x_2$
$f_5$	0	1	0	1	$x_2$	2. Identität	
$f_6$	0	1	1	0	$\overline{x_1} x_2 + x_1 \overline{x_2}$	Antivalenz	$x_1$ XOR $x_2$
$f_7$	0	1	1	1	$x_1 + x_2$	Disjunktion	$x_1$ OR $x_2$
$f_8$	1	0	0	0	$\overline{x_1 + x_2}$	Negatdisjunktion	$x_1$ NOR $x_2$
$f_9$	1	0	0	1	$(\overline{x_1} + x_2)(x_1 + \overline{x_2})$	Äquivalenz	$x_1 \Leftrightarrow x_2$
$f_{10}$	1	0	1	0	$\overline{x_2}$	2. Negation	NOT $x_2$
$f_{11}$	1	0	1	1	$x_1 + \overline{x_2}$	2. Implikation	$x_2 \Rightarrow x_1$
$f_{12}$	1	1	0	0	$\overline{x_1}$	1. Negation	NOT $x_1$
$f_{13}$	1	1	0	1	$\overline{x_1} + x_2$	1. Implikation	$x_1 \Rightarrow x_2$
$f_{14}$	1	1	1	0	$\overline{x_1 x_2}$	Negatkonjunktion	$x_1$ NAND $x_2$
$f_{15}$	1	1	1	1	1	Einsfunktion	

## Normalformen

### Systematik der Darstellung

Produktterm	Konjunktion einfacher Variablen Bsp.: $x_1, x_1 * x_2, \bar{x}_1$
Summenterm	Disjunktion einfacher Variablen Bsp.: $\bar{x}_2 + \bar{x}_1, x_1 + x_2 + x_3$
Minterm	Ein Produktterm, in dem jede Variable einer Booleschen Funktion genau einmal vorkommt Bsp.: $\bar{x}_1 * x_2 = \text{Minterm v. } f(x_1, x_2)$
Maxterm	Ein Summenterm, in dem jede Variable einer Booleschen Funktion genau einmal vorkommt Bsp.: $\bar{x}_1 + x_2 = \text{Maxterm v. } f(x_1, x_2)$
Disjunktive Normalform	Disjunktion von Produkttermen Bsp.: $(\bar{x}_1 * x_2) + (x_1 * x_2)$
Kanonische Disjunktive Normalform	Darstellung einer Booleschen Funktion f als Disjunktion von Mintermen Bsp.: $(\bar{x}_1 * x_2 * \bar{x}_3) + (\bar{x}_1 * x_2)$
Konjunktive Normalform	Konjunktion von Summentermen Bsp.: $(\bar{x}_1 + x_2) * (x_1 + x_2)$
Kanonische Konjunktive Normalform	Darstellung einer Booleschen Funktion f als Konjunktion von Maxtermen Bsp.: $(\bar{x}_1 + x_2) * (x_1 + x_2 + \bar{x}_3)$

### Bildung

#### Bildung der Kanonischen Disjunktiven Normalform (KDNF)

Aus der Wahrheitstabelle:

Die Idee ist, wenn **einer der Summanden** (also eine Konjunktion, in der Disjunktion) den **Wert 1** annimmt, so ist die **Summe gleich 1**.

Wir ermitteln den **Minterm** für alle Zeilen mit  $f(x_1, \dots, x_n) = 1$  und verkettens/summieren die daraus resultierenden Produkte/Konjunktionen mittels Disjunktion/Summenterm.

Ist eine Variable = 0, wird sie im Minterm negiert.

$$(x_1 * \bar{x}_2 * x_3) \rightarrow 1 \ 0 \ 1 = 1$$

### Bildung der Kanonischen Konjunktiven Normalform (KKNF)

Aus der Wahrheitstabelle:

Die Idee ist, wenn **einer der Faktoren** (also eine Disjunktion, in der Konjunktion) den **Wert 0** annimmt, so ist das **Produkt gleich 0**.

Der **Maxterm** (also eine Disjunktion) für alle Zeilen mit  $f(x_1, \dots, x_n) = 0$  wird gebildet, und mittels einer Konjunktion/Produktterm verkettet.

Die Variable  $x_i$  wird negiert, wenn sie in der Wahrheitstabelle den Wert 1 hat.

$$(x_1 + \bar{x}_2 + x_3) \rightarrow 0 \ 1 \ 0 = 0$$

### Sätze zu Booleschen Funktionen

Alle Booleschen Funktionen können mithilfe der **Negation**, **Konjunktion** und **Disjunktion** dargestellt werden.

Weiteres können alle Booleschen Funktionen **entweder** mithilfe der **Negation** und **Konjunktion**, **oder** mithilfe der **Negation** und **Disjunktion** dargestellt werden.

Zusätzlich können alle Booleschen Funktionen **entweder** mithilfe der **NAND**-Verknüpfung **oder** mithilfe der **NOR**-Verknüpfung dargestellt werden.

Jede Boolesche Funktion lässt sich als

⊗ **genau eine KDNF**

⊗ **genau eine KKNF**

darstellen.

**Jede KDNF** kann in eine KKNF umgewandelt werden.

**Jede KKNF** kann in eine KDNF umgewandelt werden.

Aufgrund der **Dualität** gilt:

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

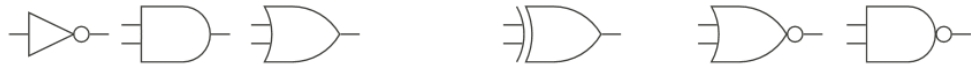
$$\text{KDNF}(f(x_1, x_2, \dots, x_n)) = \overline{\text{KKNF}(\overline{f(x_1, x_2, \dots, x_n)})}$$

Die Darstellung einer Booleschen Funktion als KDNF & KKNF ist **eindeutig**.

Zwei Darstellungen Boolescher Funktionen sind **äquivalent**, wenn sie auf die gleiche KDNF bzw. KKNF zurückgeführt werden können.

## Schaltung

Schaltsymbol



Bezeichnung

NOT

AND

OR

XOR

NOR

NAND

## Realisierung einer (günstigen) Schaltung

Eine boolesche Funktion  $f$  kann stets in drei systematischen Schritten realisiert werden:

Aufstellen der Wahrheitstabelle von  $f$

KDNF (oder KKNF) von  $f$  bilden

Schaltungstechnische Realisierung der gebildeten KDNF (o. KKNF) mit den obigen Gattern

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

### Bemerkung

Liegen **alle Eingangssignale  $x_i$  einfach, als auch negiert** vor, lässt sich **jede Boolesche Funktion** damit auf **maximal zwei Gatterebenen** realisieren. **Andernfalls** benötigt man **maximal drei Gatterebenen**.

### Realisierung einer KDNF

max.  $2^n$  AND-Gatter mit je  $n$  Eingängen  
(eines pro Minterm)

ein OR-Gatter zur Disjunktion aller  
Minterme  
(mit max.  $2^n$  Eingängen)

### Realisierung einer KKNF

max.  $2^n$  AND-Gatter mit je  $n$  Eingängen  
(eines pro Minterm)

ein OR-Gatter zur Disjunktion aller  
Minterme  
(mit max.  $2^n$  Eingängen)

### Realisierung einer kanonischen Normalform

Max.  $2^n (n-1) + (2^n - 1) = n * 2^n - 1$  Gatter (mit 2 Eingängen)

Max.  $\log_2 n + n$  Ebenen (aus Gattern mit 2 Ebenen)

### Anzahl an Gattern und Ebenen

Wird eine Realisierung mit Standardbauteilen (also zwei Eingängen pro Gatter) durchgeführt, gilt folgende Regelung:

Anzahl Gatter = [Anzahl Eingänge] - 1

Anzahl Ebenen =  $\log_2$  [Anzahl Eingänge]  $\Rightarrow 2^x =$  [Anzahl Eingänge]

$\Rightarrow$  Zwei hoch wieviel ist [Anzahl Eingänge]?

## „Universelle“ Gatter

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



De Morgan:

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

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



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

## Minimierung

Minimieren können wir die **Anzahl der Gatter** (also die **Anzahl der Booleschen Operationen**), die *Anzahl der Verbindungen* und der *Produkt- & Summenterme*. Die letzteren beiden sind für uns jedoch aus aktueller Sicht *nicht relevant*.

Die Anzahl der Gatter/Booleschen Operationen können wir durch simple **Umformung nach den Regeln der Booleschen Algebra** minimieren, oder aber mit dem **Karnaugh-Veitch-Diagramm** (bis 4-5 Variablen). In weiterer Folge kann auch mittels Algorithmen (z.B. Quine & McCluskey) gearbeitet werden.

## Resolutionsregeln

„Ich muss Terme loswerden“ → Resolutionsregel

*Wenn sich zwei Summanden/Faktoren nur in **genau einer komplementären Variable** unterscheiden, können wir beide Summanden durch ihren gemeinsamen Teil ersetzen.*

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

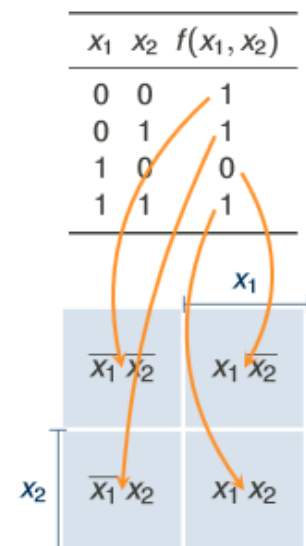
$$(x_1 + x_2 + \underline{\overline{x_3}} + \overline{x_4}) \cdot (x_1 + x_2 + \underline{x_3} + \overline{x_4}) \Leftrightarrow (x_1 + x_2 + \overline{x_4})$$

## Karnaugh-Veitch-Diagramme (KV)

Wahrheitstabelle wird so visualisiert, damit wir die Resolutionsregeln darauf anwenden können.

Wir stellen also die Funktionswerte der Wahrheitstabelle 2-dimensional dar. Dabei repräsentiert jedes Element der Matrix/des KV's einen Minterm aller Variablen.

Die Elemente werden so angeordnet, dass sich zwei benachbarte Elemente nur im Vorzeichen genau einer Variable unterscheiden.



## Beispiel für KV an einer KDNF

KDNF:  $f(x_1, x_2) = \overline{x_1} \cdot \overline{x_2} + \overline{x_1} \cdot x_2 + x_1 \cdot x_2$       logische Konsequenz:  $x_1 * \sim(x_2) = 0$

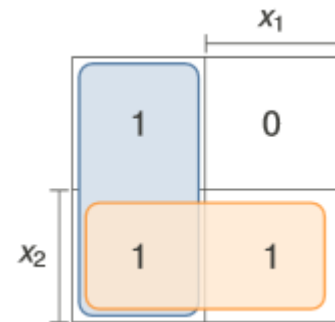
Dann erstellen wir das KV-Diagramm →

Wir setzen den Funktionswert in das Diagramm ein.

Nun markieren wir möglichst **wenige**, aber möglichst **große, zusammenhängende** (ggf. überlappender) **Bereiche aus  $2^k$  Einsen**, bis alle Einsen überdeckt sind.

Dann können wir die minimale DNF (keine kanonische) durch die Summierung von genau einem Produktterm pro markiertem Bereich:

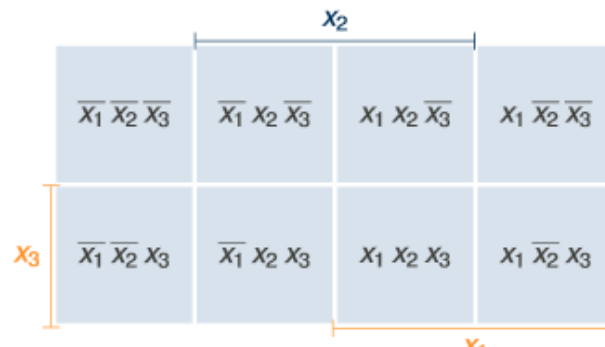
$$f(x_1, x_2) = \overline{x_1} + x_2$$



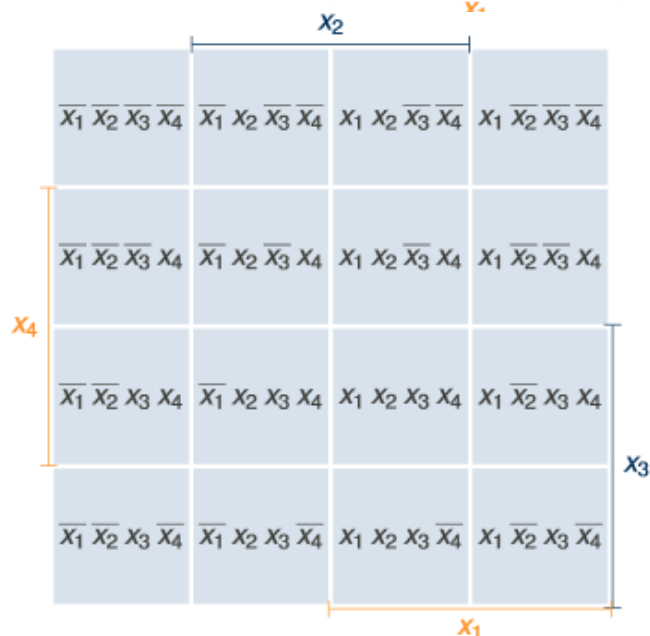
## KV-Diagramme für mehrstellige Boolesche Funktionen

Wir fügen an das KV-Diagramm von vorne an der linken Seite ein neues  $x_1$  (negiert) ein, und indizieren die anderen  $x_i$  mit  $i+1$ . Damit können wir beliebig dimensionierbare Faltechnik.

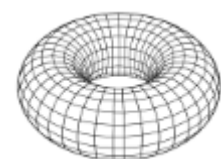
3 Variablen:



4 Variablen:



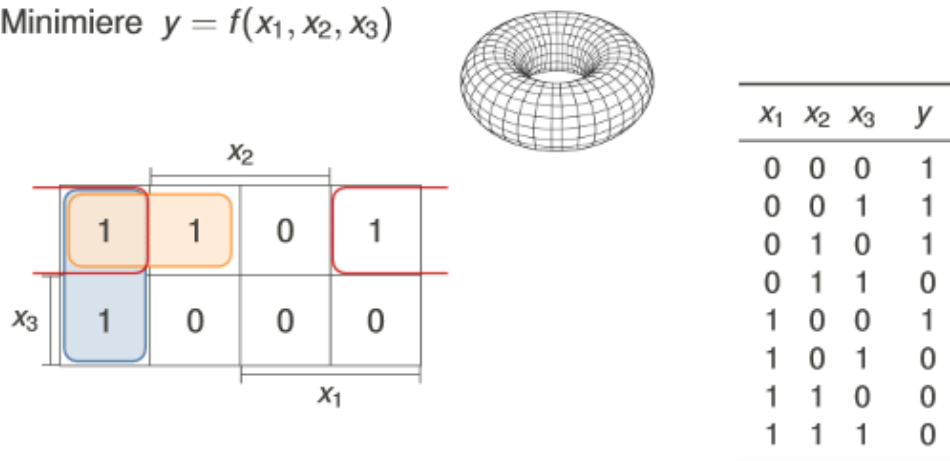
Hier unbedingt die Wahrheitstabelle stets vor Augen halten, sonst fügt man falsche Werte ein. Weiteres muss man sich unbedingt das KV-Diagramm als einen Donut vorstellen, damit wir auch zyklisch<sup>1</sup> markieren können:



<sup>1</sup>zyklisch bedeutet, wenn ich rechts/unten aufhöre, beginne ich links/oben wieder

Beispiel mit zyklischer Markierung einer mehrstelligen Booleschen Funktion

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



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

Minimierung einer KKNF

Die obigen Beispiele behandeln alle eine DNF. Für eine KNF müssen wir einen ähnlichen, dennoch anderen Weg gehen:

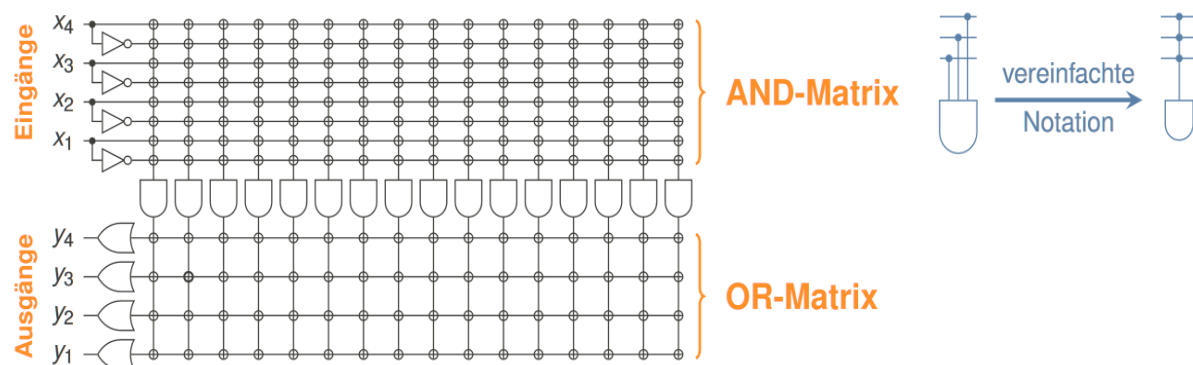
- Wir markieren gleich wie bei der DNF möglichst wenige, große Bereiche zusammenhängender (ggf. überlappender) Bereiche, allerdings aus **2<sup>k</sup> Nullen**.
- Bildung einer DNF durch die Summierung von genau einem Produktterm pro markierten Bereich
- Umwandlung der DNF in eine minimale KNF durch abschließende Negation.

Nachteile des KV-Diagrammes

Zum einen können zyklische Markierungen leicht übersehen werden, zum anderen sind sie ab fünfstelligen Booleschen Funktionen ungebräuchlich.

## Praktische Realisierung

In der Industrie werden „Logikbausteine“ entwickelt, welche einmalig programmiert werden können. **OTP-Logikanordnungen** (one time programmable) nennt man diese. Dabei wird vom Hersteller eine Platine vorgegeben, die alle Szenarien abdeckt, aber auf ein bestimmtes Szenario angepasst werden muss.



## Programmable Read-Only Memory (PROM)

**AND-Matrix** ist **festgelegt**, die **OR-Matrix** **programmierbar**. Die Wahrheitstabelle kann direkt realisiert werden

## Programmable Array-Logic (PAL)

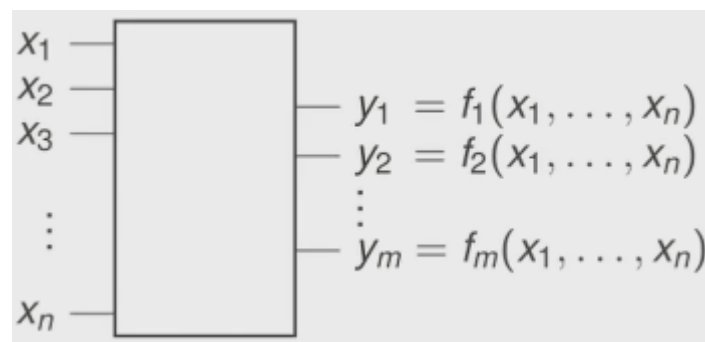
**AND-Matrix** **programmierbar**, **OR-Matrix** ist festgelegt. Kann DNFs bis zu einer Obergrenze von Produkttermen pro Summand realisieren. GAL (generic) ist eine wiederprogrammierbare Variante des PALs

## Programmable Logic Array (PLA)

**AND-Matrix und OR-Matrix** **programmierbar**. Damit lassen sich DNFs bis zu einer Obergrenze an Produkttermen realisieren.

## Kombinatorische Logik / Schaltnetze

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



*Ein Schaltnetz ist eine schaltungstechnische Realisierung einer Schaltfunktion und wird Kombinatorische Logik genannt.*

## Dekodierer

„k-zu-n-Dekodierer“

Man legt an den Eingängen des Dekodierers ein binäres Signal an, was auf den Ausgang referenziert. Das heißt, es wird **eines von  $n$  Ausgängen** durch **Binärdarstellungen an den Eingängen auf 1** gesetzt.

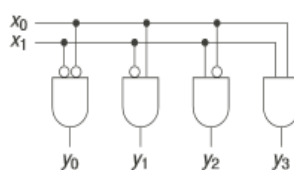
$x_1$	$x_0$	$y_3$	$y_2$	$y_1$	$y_0$
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

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

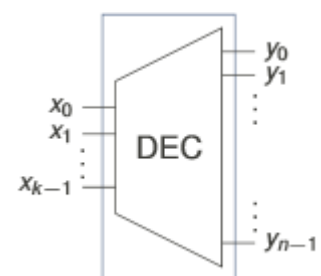
$$y_1 = x_0 \cdot \overline{x_1}$$

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

$$y_3 = x_0 \cdot x_1$$



→



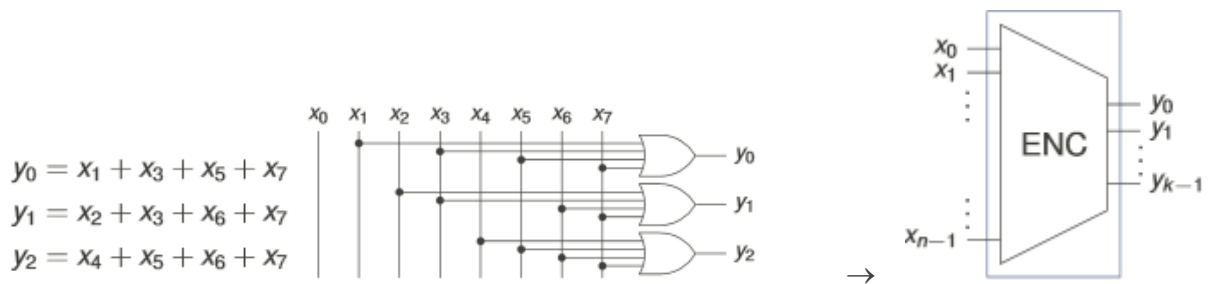
Anwendung findet ein Dekodierer vor allem in der Speicheradressierung.

## Kodierer

„n-zu-k-Kodierer“



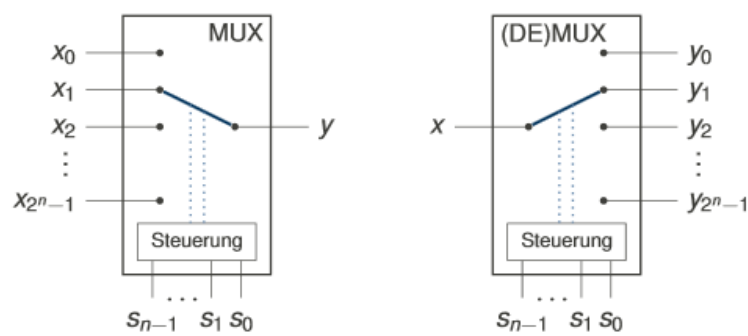
**Die Ausgabe** steht in **Binärdarstellung** für den Index **eines aktiven Einganges**  $x_i = 1$



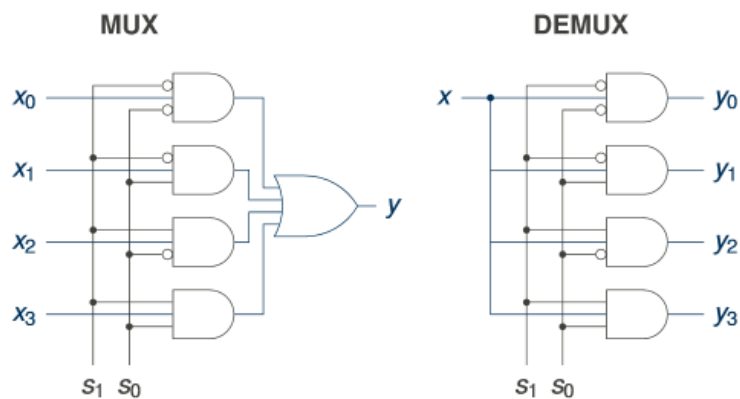
Der Kodierer wird z.B. für Tastendrücke verwendet. Da es allerdings zu **undefinierten Ausgaben bei mehreren aktiven Eingängen** kommt, ist das nicht mehr die moderne Methode.

### (De-)Multiplexer

Über  $n$ -Steuerleitungen kann ausgewählt werden, welcher Eingang auf den Ausgang geschaltet wird. Beim De-Multiplexer ist das logischerweise genau umgekehrt, d.h.:  $n$ -Steuerleitungen kontrollieren, an welchen Ausgang der Eingang geschaltet wird.



für  $n = 2$  Steuerleitungen



## Sequenzielle Logik

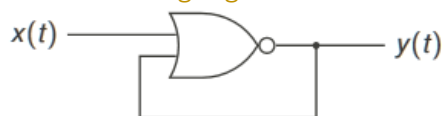
Worin liegt der Unterschied zwischen Kombinatorische und Sequenzielle Logik?

Kombinatorische Logik	Sequenzielle Logik
Keine Rückkopplungen	<i>Kontrollierte</i> Rückkopplungen
Grundelemente sind Gatter	Grundelemente sind Flipflops und Gatter
Schaltnetze sind idealisiert verzögerungsfrei	Schaltwerke berücksichtigen Zeitverhalten <sup>1</sup>
Beschreibung durch Wahrheitstabelle oder Boolesche Ausdrücke	Beschreibung durch Zustandstabelle, Zustandsdiagramm oder Ansteuer- und Ausgabegleichungen

## Schaltwerke

### Rückkopplungen

Ein Gatterausgang



NOR-Gatter

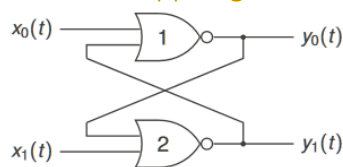
*Der Ausgang oszilliert unkontrolliert.*

Ist  $x(t)$  gleich 1, ist der Ausgang immer 0

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

Andernfalls oszilliert der Ausgang abhängig von der Gatterlaufzeit  $\tau$

Kreuzweise Rückkopplung von zwei Ausgängen



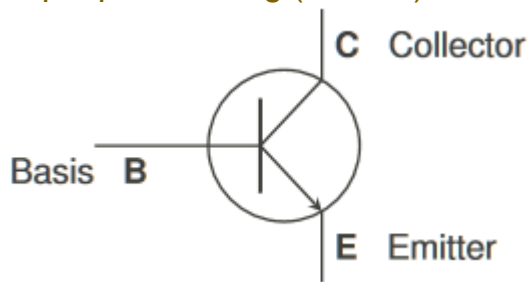
2 NOR-Gatter

**Bistabile Kippstufe:** Diese Schaltung kann ein Bit in ihrem Zustand  $Q$  speichern.

Diese bistabile Kippstufe nennt man auch „RS-Flipflop“

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

## Flipflop-Schaltung („Latch“)



Transistor:

*Strom fließt von C nach E nur wenn ein Steuerstrom von B nach E fließt.*

## RS-Flipflop

Ein RS-Flipflop kann man mittels einer mechanischen Analogie erklären:

Wird in einem RS-Flipflop ein Eingang gesetzt, wird der Zustand vom Flipflop „gespeichert“, bis der andere Eingang aktiviert wird.

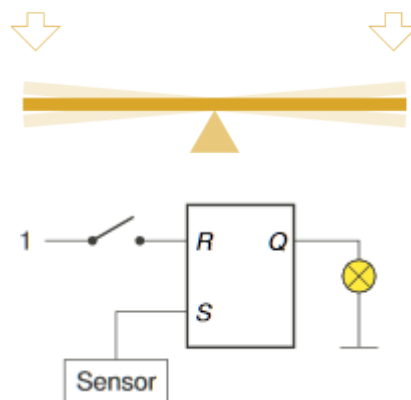
R	S	Q'
0	0	Q
0	1	1
1	0	0
1	1	nicht erlaubt

Folgezustand

S  
Set (setzen)

R  
Reset (zurücksetzen)

Symbol:



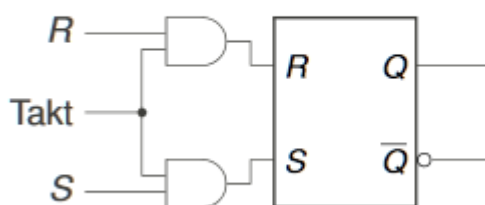
Dient zur Speicherung eines flüchtigen Wertes, erfordert allerdings eine Initialisierung.

Außerdem sind mehrere Zustandsänderungen pro Takt möglich.

## Getaktetes RS-Flipflop

Wir wollen vermeiden, dass in der Laufzeit unbeabsichtigte Zustandsübergänge auftreten können. Daher übernehmen wir die Signale an R und S nur dann, wenn ein Taktsignal aktiv vorliegt.

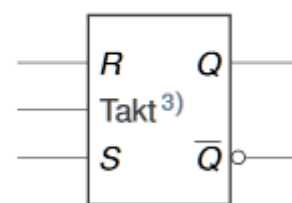
### Realisierung



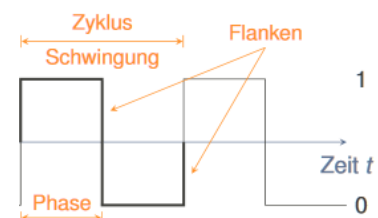
### Wahrheitstabelle

R	S	Takt	Q'
d <sup>1)</sup>	d	0	Q
0	0	1	Q
0	1	1	1
1	0	1	0
1	1	1	— <sup>2)</sup>

### Symbol



*Takt ist ein symmetrisches Rechtecksignal mit konstanter Taktfrequenz (also Schwingungen pro Zeiteinheit) gemessen in Hertz<sup>1</sup>*



<sup>1</sup> 1 Hertz = 1/s

Wie wird ein solcher Takt generiert?

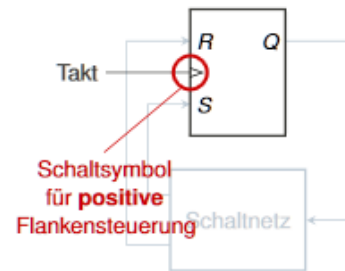
Industriell gefertigte Quarze schwingen besonders präzise, mit  $> 4$  MHz. Niedrigere Frequenzen erreicht man durch Teilungen.

### Flankengesteuertes RS-Flipflop

Da bei RS-Flipflops mehrere Zustandsänderungen in einer Taktphase möglich sind und kontrollierte Rückkopplungen über Schaltnetze damit erschwert werden, brauchen wir eine Alternative: **Die Flankensteuerung**.

Eingabewerte werden dann nur noch zu einem bestimmten Zeitpunkt:

- ⌘ **Positive** (also bei steigenden Flanken)
- ⌘ **Negative** (also bei fallenden Flanken)



## Arithmetik

### 2-Bit Multiplizierer

#### KOMBINATORISCHE LOGIK II - FOLIE 31

### Addition und Subtraktion

#### Addition positiver n-stelliger Binärzahlen

Erfolgt von rechts nach links, wobei an jeder Stelle  $i$  ein Carry/Übertrag entstehen kann, der auf das  $i+1$  Stelle addiert wird. Sollte die Summe nichtmehr mit  $n$ -Bits dargestellt werden können, also das  $n+1$ -te Bit gleich 1, wird dieses als Überlauf bezeichnet.

#### Addition mit Zweierkomplement

**Sind  $a$  und  $b$  positiv** (also  $a_{n-1} = b_{n-1} = 0$ ), gibt es einen **Arithmetischen Überlauf** bei  $y_{n-1} = 1$ .

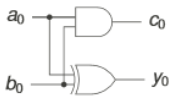
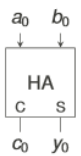
$$N = 5 \rightarrow 12 + 9 = 01100 + 01001 = 10101 = -11$$

**Sind  $a$  und  $b$  negativ** (also  $a_{n-1} = b_{n-1} = 1$ ), gibt es einen Arithmetischen Überlauf bei  $y_{n-1} = 0$ . Das korrekte Ergebnis wird also durch Abschneiden und Ignorieren des Übertragbits  $c_{n-1}$  erreicht.

Sind die **Vorzeichen von  $a$  und  $b$  unterschiedlich**, ist kein arithmetischer Überlauf möglich! Das korrekte Ergebnis wird durch Abschneiden und Ignorieren des Übertragbits  $c_{n-1}$  erreicht.

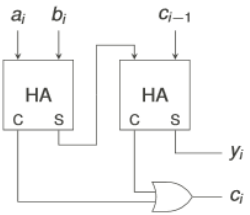
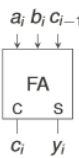
## Halbaddierer (Half Adder)

Dieser ermittelt aus  $a_0$  und  $b_0$  die Summe  $y_0$  und den Übertrag  $c_0$ . Durch eine Verzögerung  $\tau$  für  $c_0$  und  $2\tau$  für  $y_0$ , ist der Einsatz für niederwertigste Bits geeignet.

Wahrheitstabelle				Realisierung		Symbol
$a_0$	$b_0$	$y_0$	$c_0$	 $c_0 = a_0 \cdot b_0$ $y_0 = a_0 \oplus b_0$		
0	0	0	0			
0	1	1	0			
1	0	1	0			
1	1	0	1			

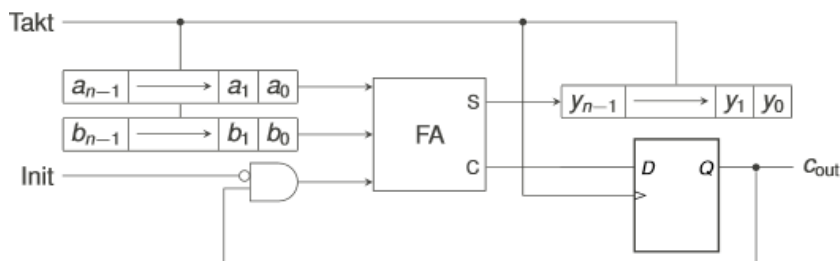
## Volladdierer (Half Adder)

Addiert  $a_i$ ,  $b_i$  und  $c_{i-1}$  von  $i=1, \dots, n-1$  und gibt die Summe  $y_i$ , sowie den Übertrag  $c_i$  aus. Dieser hat eine Verzögerung abhängig vom Pfad und Realisierung von  $2\tau$  bis  $4\tau$ .

Wahrheitstabelle					Realisierung		Symbol
$a_i$	$b_i$	$c_{i-1}$	$y_i$	$c_i$	 $c_i = a_i \cdot b_i + a_i \cdot c_{i-1} + b_i \cdot c_{i-1}$ $y_i = a_i \oplus b_i \oplus c_{i-1}$		
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			

## Serielles Addierwerk

Ist eine Konstruktion eines synchronen Schaltwerks aus einem Volladdierer, einem Flipflop und drei n-Bit-Schieberegister (für a, b, y)



Init dient zum Löschen des Übertrags, falls das Flipflop nicht initialisiert ist.

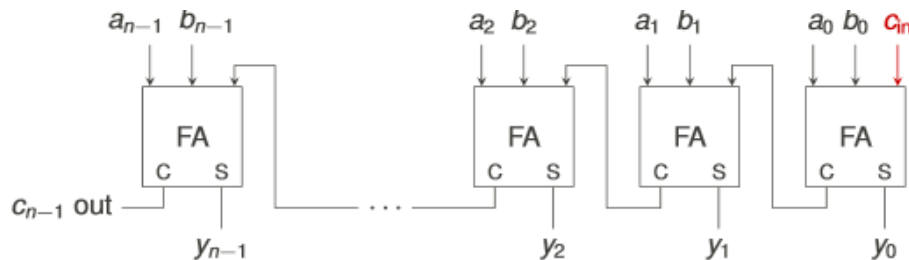
In Takt  $t$  wird Ergebnisbit  $y_t$  aus  $a_t$ ,  $b_t$  und  $c_{t-1}$  bestimmt.

Die Addition von zwei n-Bit-Zahlen benötigt n Taktzyklen.

## Paralleles Addierwerk

Wird auch „Ripple Carry“-Addierer (RCA) genannt, weil Übertrag an Position  $i=0$  alle Bitstellen bis  $n-1$  durchlaufen kann und ist eine Konstruktion aus  $n$  Volladdierern mit Überlauf Eingang. Er hat eine max. Verzögerung von  $2n \cdot \tau_{au}$ .

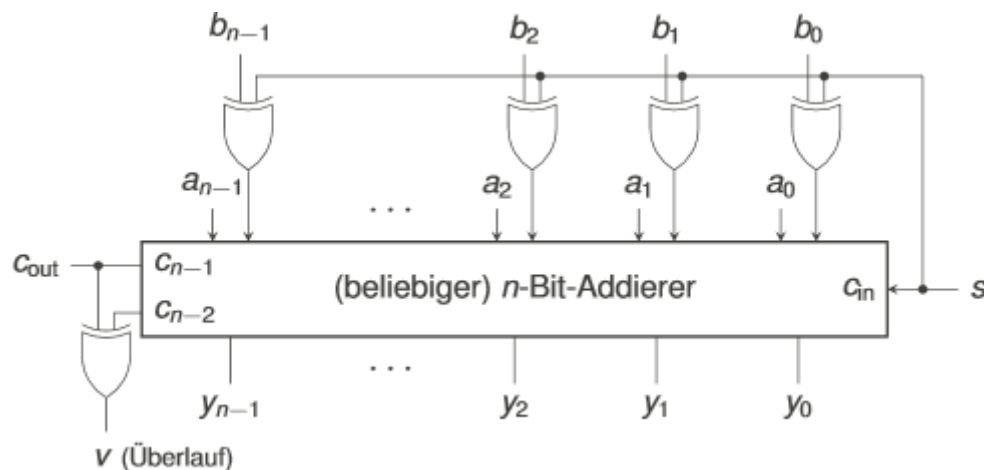
In der Praxis gibt es den besseren „Carry Look-Ahead“-Addierer (CLA), welcher mit konstanter Verzögerung (also unabhängig von  $n$ ) addiert.



## Kombiniertes Addier-/Subtrahierwerk

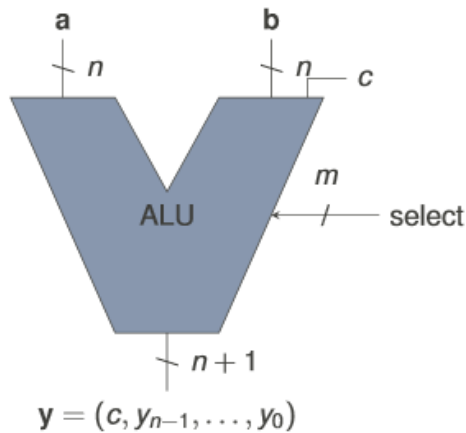
Ein Steuereingang  $s$  wählt zwischen Addition ( $a+b$ ) für  $s=0$  und Subtraktion ( $a-b$ ) für  $s=1$ .

Die Idee dahinter ist, dass XOR-Gatter die Bits  $b_i$  bei  $s=1$  invertieren.



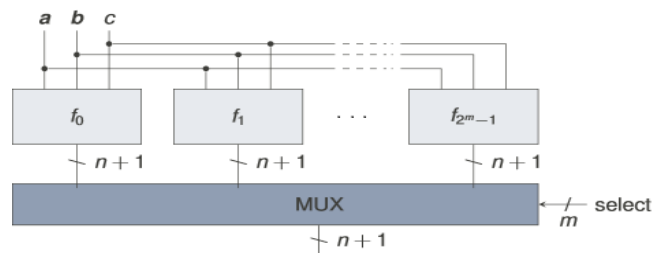
## ALU / Arithmetisch-logische Einheit

Ein elektronisches Rechenwerk, welches in der Praxis tatsächlich in Prozessoren zum Einsatz kommt.



„Multifunktionsmodul für Verknüpfung zwischen n-Bit-Registern.“

Sinnvollerweise kann sie zumindest die Addition, Negation und Konjunktion, in der Praxis oft mehr.



Die Select-Eingänge  $s_i$  sollten sinnvoll gewählt werden, z.B.:

$s_2$  entscheidet, zwischen arithmetischen und logischen Operationen

$s_1$  und  $s_0$  wählen konkrete Operationen.

Defacto ist die ALU dafür zuständig aus zwei Zahlen ( $a$ ,  $b$ ) mithilfe einer bestimmten Rechenoperation (select) eine neue Zahl  $y$  zu berechnen und diese weiterzugeben.

$s_2$	$s_1$	$s_0$	Funktion
0	0	0	0
0	0	1	$b - a$
0	1	0	$a - b$
0	1	1	$a + b + c$
1	0	0	$a \oplus b$
1	0	1	$a \vee b$
1	1	0	$a \wedge b$
1	1	1	1

## Multiplikation

Ein Multiplizierer kann durch ein **Zweistufiges Schaltnetz**, **Seriell**es Schaltwerk oder einem **Feldmultiplizierer** realisiert werden.

Zuvor müssen wir allerdings noch den Algorithmus zur Multiplikation positiver Binärzahlen verstehen.

### Algorithmus zur Multiplikation positiver Binärzahlen

Das Produkt  $y = a \times b$  aus zwei  $n$ -Bit-Faktoren hat  $2n$  Stellen. Wir führen die Multiplikation nun auf die bedingte Addition und Schiebeoperationen zurück:

Wir verschieben  $a$  um  $i$ -Bits, wenn  $b_i = 1$ , ansonsten setzen wir 0. Danach summieren wir alles:

$a \times b:$	01010	$\times$	01101	
	01010	$\times 1$	$b_0$	
	00000	$\times 0$	$b_1$	
	01010	$\times 1$	$b_2$	
	01010	$\times 1$	$b_3$	
	00000	$\times 0$	$b_4$	
$y =$	0010000010			

Weil ein schlauer Mensch einst aber einen einfacheren, modifizierten Algorithmus fand, kann man auch die Linksverschiebung von  $a$  durch eine Rechtsverschiebung von  $y$  durchführen:

Wenn  $b_i = 1$ , dann addieren wir  $a$  zu  $y$  ganz links.

Danach wird, egal was  $b_i$  ist, das Resultat nach rechts verschoben.

Das Ganze wiederholen wir für  $i = 0$  bis  $n-1$ .

01010	×	01101	
00000	00000		
+ 01010		add a	
01010	00000		
001010	0000	shift	
0001010	000	shift	
+ 01010		add a	
0110010	000		
00110010	00	shift	
+ 01010		add a	
10000010	00		
010000010	0	shift	
y = 0010000010		shift	

Ebenfalls möglich ist, dass die Angabe mittels **Steuerbefehle** durchgeführt werden soll:

```
clear y
load & shift
shift b
load & shift
shift b
load & shift
shift b
```

(Entsprechen nicht den Beispielen!)

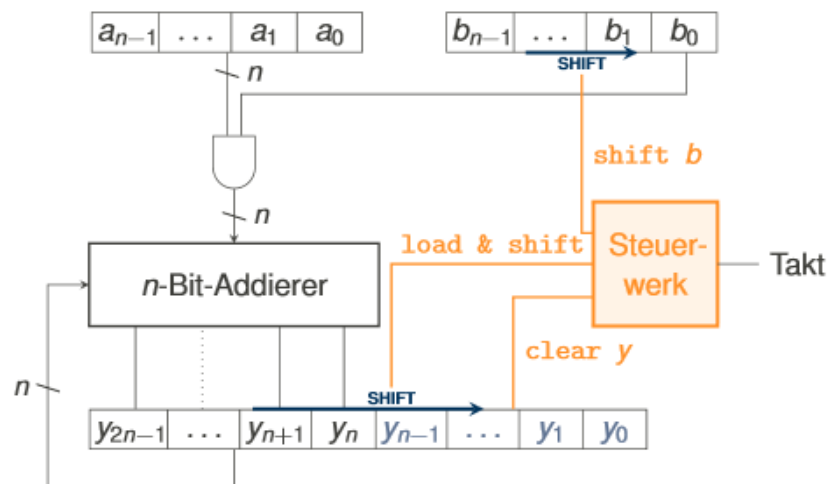
## Schaltnetz

Eine Realisierung eines  $n \times n$ -Bit-Multiplizierers als Schaltnetz mit je  $2n$  Ein-/Ausgängen erfolgt durch Implementierung z.B. in PROM (Seite 16) mit  $2^{(2n)}$  Zeilen aus  $2n$ -Bit-Worten.

Sehr geringe Zeitverzögerung, aber ein sehr hoher Schaltungs-/Speicheraufwand „skaliert nicht“, weil der Aufwand exponentiell explodiert.

## Serielles Schaltwerk

Mithilfe des Algorithmus zur Multiplikation positiver Binärzahlen (Seite 23) lässt sich folgendes Schaltwerk einfach realisieren:



Die Berechnung dauert  $n$ -Taktzyklen.

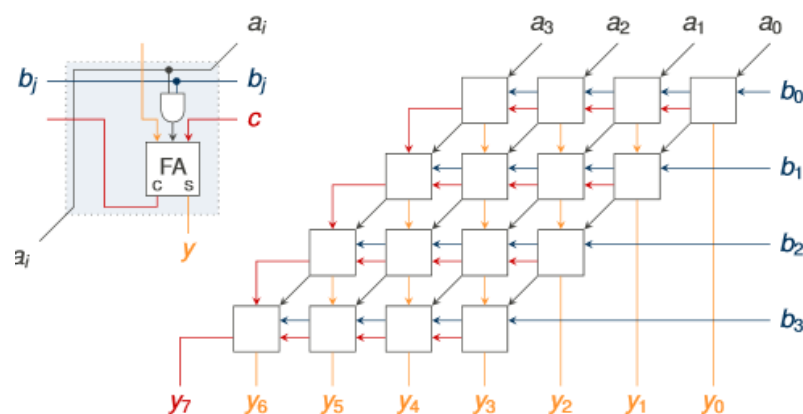


## Feldmultiplizierer (array multiplizer)

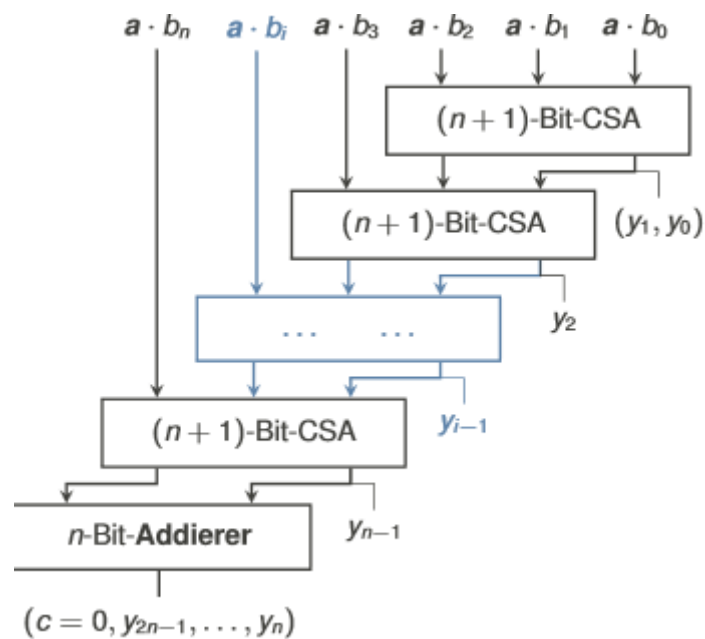
Ausgehend von folgender, direkten Schaltung einer schriftlichen Multiplikation

$(a_3, a_2, a_1, a_0)$				$\times (b_3, b_2, b_1, b_0)$				
				$a_3b_0$	$a_2b_0$	$a_1b_0$	$a_0b_0$	} $n$ partielle Produkte
			$a_3b_1$	$a_2b_1$	$a_1b_1$	$a_0b_1$	0	
		$a_3b_2$	$a_2b_2$	$a_1b_2$	$a_0b_2$	0	0	
	$a_3b_3$	$a_2b_3$	$a_1b_3$	$a_0b_3$	0	0	0	
$y_7$	$y_6$	$y_5$	$y_4$	$y_3$	$y_2$	$y_1$	$y_0$	

Lässt sich dieser strukturierte Aufbau aus  **$n^2$  Multiplizierzellen** umsetzen:



### Variante mit CSA-Kaskade:



## Optimierungsmöglichkeiten

Wenn du dir das da oben genau anschaust, merkst du, dass jede Eins in  $b$  eine Addition verursacht.

$$\begin{array}{rcl} a \times 111111 & \text{vs.} & a \times 100001 \\ a \times 111111 & = & a \times 1000000 - a \times 0000001 \end{array}$$

Die Multiplikation mit einer 1-Folge kann immer durch **eine** Addition und **eine** Subtraktion ersetzt werden. Dazu dient der **Algorithmus von Booth**. (Seite 26)

## Multiplikation negativer Zahlen

Wir hatten bislang immer mit positiven Zahlen multipliziert. Was aber mit negativen Zahlen?

Weil ich den Beweis eh nicht kapiert hab, fassen wir einfach den Konsens daraus zusammen:

---

*Naives Multiplizieren liefert falsche Ergebnisse*

---

Wir müssen also eine alternative Lösung finden:

- ⊗ Trennung von Vorzeichen und Betrag
- ⊗ Addition von Korrekturtermen
- ⊗ Algorithmus von Booth (mit vorzeichenrichtiger Ergänzung) <-- des da nehmen ;-)

## Algorithmus von Booth

Dieser analysiert zwei benachbarte Bits  $b_i$  und  $b_{i-1}$  und macht bei

<b>11 / 00</b>	Nichts
<b>10</b>	Addition von $a \times 2^i$
<b>01</b>	Subtraktion von $a \times 2^i$
<b>Initiale Ergänzung</b>	Wenn $b_0 = 1$ , dann muss $b_{-1} = 0$ gesetzt werden – um Anfang einer Folge nicht zu verpassen.

Beispiel für  $n = 8$ :

$$\begin{aligned} 42 \times 92 &= (00101010)_2 \times (01011100)_2 \\ -42 &= (11010110)_2 \end{aligned}$$

$$00101010 \times 01011100$$

$$\begin{array}{rcl} & 01011100 & \\ 11111111010110 & \leftarrow & 01011100 \\ & 01011100 & \\ & 01011100 & \\ 00000101010 & \leftarrow & 01011100 \\ 1111010110 & \leftarrow & 01011100 \\ 000101010 & \leftarrow & 01011100 \\ \hline 10000111100011000 & = & (3864)_{10} \end{array}$$

$$\begin{array}{rcl} (10)_{10} \times (-13)_{10} & & \\ 01010 \times 100110 = b_{-1} & & \\ 111110110 & \leftarrow & 100110 \\ 00001010 & \leftarrow & 100110 \\ 110110 & \leftarrow & 100110 \\ \hline 1110111110 & = & (-130)_{10} \end{array}$$

$$\begin{array}{rcl} (10)_{10} = (01010)_2 & & \\ (-10)_{10} = (10110)_2 & & \\ (-13)_{10} = (10011)_2 & & \\ 101101 \times 100110 = b_{-1} & & \\ 000001010 & \leftarrow & 100110 \\ 1110110 & \leftarrow & 100110 \\ 001010 & \leftarrow & 100110 \\ \hline 10010000010 & = & (130)_{10} \end{array}$$

## Division (mit Restoring)

Eine Division erfolgt durch bedingte Addition, Subtraktion und Schiebeoperationen.

Zuerst wird ein  $2n$ -Bit-Schieberegister  $y$  initialisiert („load“), wobei die Stellen  $y_0$  bis  $y_{n-1} = a$  sind, und alle Stellen  $y_{2n-1}$  bis  $y_{n+1}$  mit 0en aufgefüllt werden.

Danach wird für  $i = 0$  bis  $n-1$  jeweils  $y$  um einen Bit nach links verschoben, davon  $b$  subtrahiert und geprüft:

Ist  $y_{2n-1} = 0$ , müssen wir  $y_0 = 1$  setzen.

Andernfalls wird der Rest wiederhergestellt, d.h. Wir addieren  $b$ .

$y_{2n-1}$  bis  $y_n$  ist dann der Rest

$y_{n-1}$  bis  $y_0$  ist unser Quotient.

## Optimierungsmöglichkeiten

### Verbesserung der langsamen Verfahren:

- CSA statt Addierer/Subtrahierer und Überträge bei Korrektur verrechnen. (**SRT-Division**)
- Radix-4-Zahlendarstellung mit Quotienten-Tabelle in ROM kombiniert.

### Verbesserung der schnelle Verfahren:

Diese beginnen mit einer Schätzung und verdoppeln die Genauigkeit mit jedem Schritt.

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

Beispiel $29 : 6 = 4$ Rest 5			
$n = 5$ Bits	$000011101 : 00110 = 00100$	$0000011101$	load
	$- 00110$	$0000111010$	shift
	$11011$	$1101111010$	sub
Korrektur	$+ 00110$		
	$00011$	$0000111010$	add
	$- 00110$	$0001110100$	shift
	$11101$	$1111010100$	sub
Korrektur	$+ 00110$		
	$000111$	$0001110100$	add
	$- 00110$	$0011101000$	shift
	$000010$	$0000101001$	sub
	$- 00110$	$0001010010$	shift
	$11100$	$1110010010$	sub
Korrektur	$+ 00110$		
	$000101$	$0001010010$	add
	$- 00110$	$0010100100$	shift
	$11111$	$1111100100$	sub
Korrektur	$+ 00110$		
Rest	$00101$	$0010100100$	add

## Rechnen mit Nachkommastellen

### Exakte Darstellung rationaler Zahlen

Dies erreicht man, indem man Zähler und Nenner separat, als Ganzzahlen, speichert und verarbeitet.

Realisiert wird dies in Software für exaktes wissenschaftliches Rechnen. *Darauf gehen wir allerdings nicht mehr näher ein.*

### Festkomma

Jede (darstellbare) Kommazahl  $z$  wird durch lineare Skalierung auf eine ganze Zahl  $z'$  abgebildet.

Ein Rechner arbeitet unverändert auf ganzzahliger Abbildung. Er arbeitet **transparent** mit skalierten ganzen Binärzahlen  $z' = z \cdot 2^k$

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

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

Die Konstante  $k$  dient nur zur Interpretation der Zahlen.

### Gleitkomma

Darstellung von Kommazahlen erfolgt durch Argument (**Mantisse**)  $a$  und Charakteristik (**Exponent**)  $c$  zur Basis  $r$ :

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

Dies erfordert spezielle Rechenwerke!

### Allgemeine Darstellung nach IEEE 754

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

Binärformat:

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

Die **Mantisse** ergibt sich aus dem Vorzeichen  $s$  und dem normalisierten Betrag  $a = 1.f$  im Bereich 1.00...00 bis 1.11...11 ohne die führende Eins. Sie ist  $p$ -Bits lang.

Der **Exponent  $e$**  mit dem Bias  $b = 2^{p-1} - 1$

Damit erreicht man einen darstellbaren Zahlenbereich von  $\pm 2^{1-b}, \dots, (2 - 2^{-m}) \times 2^b$

Zwischen  $2^{e-b}$  und  $2^{e-b+1}$  gibt es  $2^m$  Gleitkommazahlen, deren Abstand von  $e$  abhängt.

---

*Bias ist bei Single immer 127, bei Double 1023*

---

## Standardisierte Formate nach IEEE754

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

Leider gibt es einige Spezialfälle, die du auch wissen musst:

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

## Ausnahmesituationen

Wenn nach Normalisierung für  $z : e \geq e_{\max} = 1$ , gibt es einen **Überlauf**.

- ⌘ Die Ausgabe wäre dann für  $z > 0$  gleich  $+\infty$ , andernfalls  $-\infty$ .
- ⌘ Bei Division durch Null  $\pm x : 0 = \pm\infty$  (falls  $x \neq 0$ )
- ⌘ Beachte:  $\infty + x = \infty$   $\infty * x = \pm\infty$

Wenn nach Normalisierung für  $z : e = 0$ , sprechen wir von einem **Unterlauf**.

- ⌘ Ausgabe von  $z = 0$
- ⌘ Oder Ausgabe einer denormalisierten Darstellung von  $z$

### Multiplikation mit Gleitkommazahlen

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

1. **Multipliziere Mantissen als Festkommazahlen:**  $y = a \times b$   
 $a = 1.f_a$  und  $b = 1.f_b$  haben  $m + 1$  Stellen  $\Rightarrow y$  hat  $2m + 2$  Stellen
2. **Addiere Exponenten**  $\gamma = \alpha + \beta - \text{bias}$
3. **Berechne Vorzeichen**  $u = s \oplus t$
4. **Normalisiere Produkt**  $(-1)^u \times y \times 2^{\gamma - \text{bias}}$ 
  - a. Falls  $y \geq 2$ , schiebe  $y$  um ein Bit nach rechts und erhöhe  $\gamma$  um 1.
  - b. Setze  $y = 1.f_y = 1.(y_{2m-1}, y_{2m-2}, \dots, y_m)_2$  mit Rundung.
5. **Behandle Ausnahmesituationen**
  - a. Überlauf, falls  $\gamma \geq e_{\max} = 2^p - 1 \Rightarrow$  Rückgabe  $\pm\infty$  (abh. von  $u$ )
  - b. Unterlauf, falls  $\gamma \leq e_{\min} = 0 \Rightarrow$  Denormalisierung
  - c. Zero, falls  $y = 0 \Rightarrow$  Rückgabe  $\pm 0$  (abh. von  $u$ )

### Addition mit Gleitkommazahlen

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

1. **Sortiere die Summanden, sodass**  $\alpha \leq \beta = \gamma$
2. **Bitposition der Mantisse anpassen:** Bestimme  $a'$ , sodass
 
$$(-1)^s \times a \times 2^{\alpha - \text{bias}} = (-1)^s \times a' \times 2^{\gamma - \text{bias}}$$
 durch Rechtsschieben von  $a$  um  $\beta - \alpha$  Bits.
3. **Addiere Mantissen**
  - a. Falls nötig, bilde Zweierkomplement von  $a'$  oder  $b$  (abh. von  $s$  und  $t$ )
  - b. Festkomma-Addition  $y = a' + b$
  - c. Falls  $y < 0$ , setze  $u = 1$  und bilde Zweierkomplement von  $y$
4. **Normalisiere Summe**  $(-1)^u \times y \times 2^{\gamma - \text{bias}}$ 
  - a. Falls  $y \geq 2$ , schiebe  $y$  nach rechts und erhöhe  $\gamma$  um 1.
  - b. Solange  $y < 1$ , schiebe  $y$  nach links und verringere  $\gamma$  um 1.
5. **Behandle Ausnahmesituationen:** Überlauf, Unterlauf,  $y = 0$

## Befehlssatzarchitektur

Dieses Kapitel ist sehr eng mit dem kommenden Kapitel „Assembler“ (Seite 38) verbunden. Hier wird die Theorie (wobei darunter auch die Praktische Umsetzung fällt) besprochen und in Assembler wird der Befehlssatz genau besprochen.

## ARM-Mikroarchitektur

### Hintergrund

Heute stellt ARM Ltd keine eigenen Chips mehr her, sondern verkauft Lizenzen an Halbleiterhersteller, die den Prozessor anpassen und integrieren dürfen.

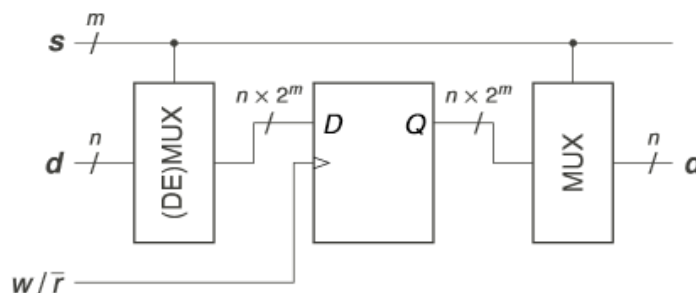
Lizenznehmer wie Intel oder Apple dürfen den Kern sogar weiterentwickeln, weshalb es eine Vielzahl an verschiedenen ARM-Varianten gibt. Wir besprechen innerhalb der Vorlesung den ARMv6 (32 Bit) von 2002, wie er in Mikrocontrollern (z.B. Raspberry Pi) verbreitet ist.

## Registersatz

CPUs sind Zustandsautomaten, deren Zustand mittels Logik in Registern gespeichert ist. Unter ARM stehen im User-Modus 16 Register mit je 32 Bit zur Verfügung:

r0 – r12	Freie Nutzung
r13	Stack-Pointer (SP)
r14	Rücksprungadresse (LR)
r15	Programmzähler (PC)

Üblicherweise lässt sich ein Arbeitsspeicher über den Systembus integrieren, womit der Zustandsraum erheblich erweitert werden kann.



Speicheranbindung des Prozessors über

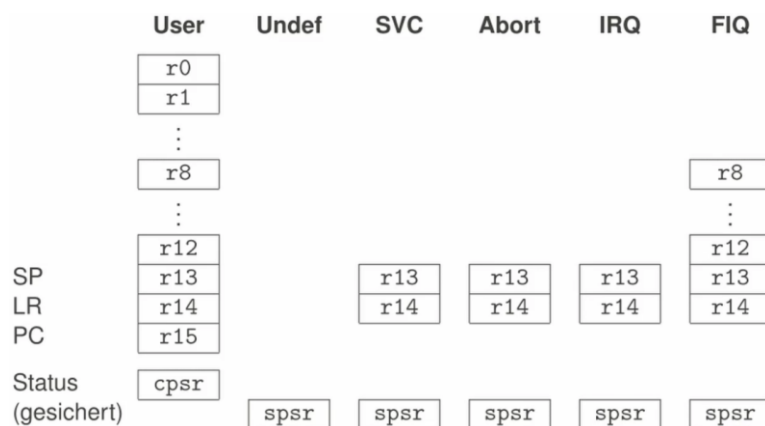
- Datenbus  $d$  der Breite  $n$  Bits, oft gleich der Registerbreite
- Adressbus  $s$  der Breite  $m$  Bits

## Beispiele

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

## Registersatz mit Bänken

In ARM gibt es verschiedene Modi, in denen der Programmierer seinen Code ausführen kann. Davon abhängig trennt ARM einige Registerinhalte nach Bänken:



Das bedeutet, User-Mode und IRQ-Mode haben eigene SP, LR und Statusregister. Damit können Register „ohne zu überschreiben“ genutzt werden.

## Endianness und Alignment

Endianness bezeichnet die Konvention zur **Reihenfolge der Ablage von Bytes eines Wortes** im Speicher:

- ☞ **Little-Endian: niederwertigstes Byte zuerst**, d.h. Wertigkeit nimmt mit zunehmender Adresse zu.
- ☞ **Big-Endian: höchstwertiges Byte an niedrigster Adresse**, d.h. Wertigkeit nimmt mit zunehmender Adresse ab

Obwohl wir nur Little-Endian verwenden, unterstützt ARM tatsächlich beides.

## Kodierung der Instruktionswörter

Jeder ARM-Befehl wird nach folgendem Schema in genau 32-Bit-Wörter kodiert.

Condition	0	0	I	OPCODE				S	Rn		Rs		OPERAND-2						Data Processing						
Condition	0	0	0	0	0	0	A	S	Rd		Rn		Rs		1	0	0	1	Rm		Multiply				
Condition	0	0	I	0	1	U	A	S	Rd HIGH		Rd LOW		Rs		1	0	0	1	Rm		Long Multiply				
Condition	0	0	0	1	0	B	0	0	Rn		Rd		0	0	0	0	1	0	0	1	Rm		Swap		
Condition	0	1	I	P	U	B	W	L	Rn		Rd		OFFSET						Load/Store - Byte/Word						
Condition	1	0	0	P	U	B	W	L	Rn		REGISTER LIST										Load/Store Multiple				
Condition	0	0	0	P	U	1	W	L	Rn		Rd		OFFSET 1		1	S	H	1	OFFSET 2		Halfword Trans. Imm. Off				
Condition	0	0	0	P	U	0	W	L	Rn		Rd		0	0	0	0	1	S	H	1	Rm		Halfword Trans. Reg. Off		
Condition	1	0	1	L	BRANCH OFFSET																	Branch			
Condition	0	0	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	1	S	H	1	Rn		Branch Exchange	
Condition	1	1	1	1	SWI NUMBER																	Software Interrupt			

## Realisierung theoretischer Grundlagen

### Divisionsalgorithmus mit „Restoring“

Grundlegend bedeutet Division mit Restoring einfach nur, dass wie bei der Grundschul-Division solange subtrahiert wird, bis eine negative Zahl rauskommt. Die negative Zahl muss dann wiederhergestellt („Restored“) werden, um zur nächsten Ziffer der Zahl überzugehen.

```

P := N
D := D << n      -- P and D need twice the word width of N and Q
for i = n-1..0 do -- for example 31..0 for 32 bits
  P := 2P - D     -- trial subtraction from shifted value
  if P >= 0 then
    q(i) := 1     -- result-bit 1
  else
    q(i) := 0     -- result-bit 0
    P := P + D    -- new partial remainder is (restored) shifted value
  end
end
end

```

N = Numerator,  
 D = Denominator,  
 n = #bits,  
 P = Partial remainder,  
 q(i) = bit #i of quotient



Beziehungsweise ein Auszug aus der Vorlesungsfolie, was das Ganze auch in Bezug auf Assembler widerspiegelt:

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

 $(y_{n-1}, \dots, y_0) \leftarrow a$            {Initialisiere ("load")  $2n$ -Bit-Register  $y$ .}
 $(y_{2n-1}, \dots, y_n) \leftarrow 0$ 

for  $i = 0$  to  $n - 1$  do
     $(y_{2n-1}, \dots, y_0) \leftarrow (y_{2n-2}, \dots, y_0, 0)$     {Schiebe  $y$  um ein Bit nach links.}
     $(y_{2n-1}, \dots, y_n) \leftarrow (y_{2n-1}, \dots, y_n) - b$ 
    if  $y_{2n-1} = 0$  then
         $y_0 \leftarrow 1$ 
    else
         $(y_{2n-1}, \dots, y_n) \leftarrow (y_{2n-1}, \dots, y_n) + b$     {Wiederherstellung des Rests}
    end if
end for

 $r \leftarrow (y_{2n-1}, \dots, y_n)$ 
 $q \leftarrow (y_{n-1}, \dots, y_0)$            {Ergebnis. Es gilt:  $a = b \times q + r$ }
```

### Realisierung in Assembler

Oben soweit die Theorie, hier die Realisierung in Assembler:

```
div:
    MOV r2, r2, LSL #16
    MOV r3, #16                ;Schleifenzähler
divloop:
    RSBS r1, r2, r1, LSL #1    ;schiebe und subtrahiere
    ORRPL r1, r1, #1
    ADDMI r1, r1, r2           ;Wiederherstellung des Rests
    SUBS r3, r3, #1
    BNE divloop               ;Quotient in r1(15), ..., r1(0)
                                ;Rest in r1(31), ..., r1(16)
    MOV pc, lr                ;Rücksprung
```

### Ausgabe von Zahlensystemen

Es werden folgende Tables verwendet:

```
lb: .ascii "\n"              // Zeilenumbruch
lutbin: .ascii "01"          //Binary Digits
lutokt: .ascii "01234567"    //Octal Digits
```

### Binär-Ausgabe

```
bin: // Binärzahl (32 Bit)
    STMFD sp!, {r0-r12, lr} // alle Register sichern
    MOV r3, #32 //32 Stellen, Schleifenzähler
    //Wir bereiten den SoftwareInterrupt Write vor, dann können wir in der Schleife
    direkt "printen"
    MOV r7, #4
    MOV r0, #0
    MOV r2, #1 //Weil ja immer nur eine Stelle ausgegeben wird

binloop:
    LDR r1, =lutbin
    ADD r1, r1, r4, LSR #31 //Addiere das 31-Bit von r4 zu r1
    SWI #0
    MOV r4, r4, LSL #1 //nächste Binärzahl
```

```
SUBS r3, r3, #1
BNE binloop //das Flag von Sub ist noch gesetzt ;)
LDR r1, =lb
SWI #0

LDMFD sp!, {r0-r12, lr} // alle Register wiederherstellen
MOV pc, lr           // Rücksprung
```

### Oktal-Ausgabe

```
okt:
    //Grundüberlegung: Oktal = 3 Bit, 32 Bit Register --> 30 Oktalziffern, 2 Bit bleiben
    über.
    //Weil nicht definiert wurde, wie mit diesen 2 Bits umgegangen werden soll (ob 10
    --> 010), habe ich sie in meiner Implementierung einfach ignoriert.
    STMFD sp!, {r0-r12, lr} // alle Register sichern
    MOV r3, #10 //10 Stellen (siehe oben), Schleifenzähler
    MOV r7, #4 //Sys-Call Write vorbereiten
    MOV r0, #0
    MOV r2, #1 //AUch Oktals können nur 1 Ziffer haben

    oktloop:
        LDR r1, =lutokt
        ADD r1, r1, r4, LSR #27 //wir brauchen die bits 27-29 von r4, weil wir ja 31 & 30
        ignorieren
        SWI #0
        MOV r4, r4, LSL #3 //weil oktal immer 3 bits
        SUBS r3, r3, #1
        BNE oktloop

        LDR r1, =lb
        SWI #0

        LDMFD sp!, {r0-r12, lr} // alle Register wiederherstellen
        MOV pc, lr           // Rücksprung
```

### Dezimal-Ausgabe

```
dec:
    // Ganzzahl in r1 (32 Bit, vorzeichenlos)
    STMFD sp!, {r0-r12, lr} // alle Register sichern
    LDR r5,=bufferdec+10 // Zeiger auf Ende des Puffers +1
    MOV r6, #0x30 // ASCII-Kode für 0 als Offset
    MOV r0, #0
    MOV r7, #0 // Stellenzähler

    decloop:
        ADD r7, r7, #1 // nächste Ziffer (mind. eine)
        MOV r2, #10 // Basis 10 (dezimal)
        BL div // r1 : r2 von Folie 5
        ADD r4, r6, r1, LSR #16 // Rest als Ziffer in ASCII . . .
        STRB r4, [r5,-r7] // . . . rückwärts in Puffer schreiben
        BICS r1, r1, #0x000f0000 // Rest löschen
        BNE decloop // mehr Stellen wenn Quotient > 0
        SUB r1, r5, r7 // Start der Zeichenkette im Puffer
        MOV r2, r7 // Länge der Zeichenkette
        MOV r7, #4 // Systemaufruf write wählen
        SWI #0

        LDR r1, =lb
        MOV r0, #1
        MOV r2, #1
        SWI #0
```

---

```
LDMFD sp!, {r0-r12, lr} // alle Register wiederherstellen
MOV pc, lr              // Rücksprung
```

---

## Hexadezimal-Ausgabe

hex:

```
STMFd sp!, {r0-r12,lr} // Register sichern
MOV r3, #8 // 8 Hexadezimalstellen
MOV r7, #4 // wähle Systemaufruf write
MOV r0, #0
MOV r2, #1 // Länge der Zeichenkette
```

hexloop:

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

```
LDR r1, =lb
SWI #0
```

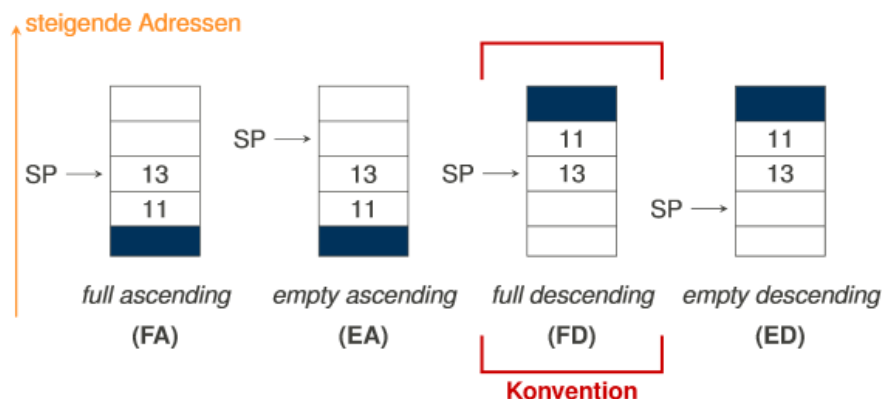
```
LDMFD sp!, {r0-r12, lr} // Register wiederherstellen
MOV pc, lr // Rücksprung
```

## Stapelorganisation und Funktionsaufrufe

Stapel (Stacks) wurden bereits in Praktische Informatik ausführlich besprochen. Uns interessiert an dieser Stelle nur die Realisierung und Nutzung. Dazu werden im Speicher zwei Zeiger definiert, die auf die Grenzen des Stapels zeigen:

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

Weiteres müssen wir 4 grundlegende Stapelvarianten unterscheiden:



Bei den Mnemonics STM/LDM (Seite 49) kann man die Suffixe FA, EA, FD und ED verwenden, um das gewünschte Verhalten zu erreichen.

## Funktionsaufruf

Der Ablauf ist schlicht:

Parameter werden an vereinbarter Stelle abgelegt, Die Ablaufsteuerung an das Unterprogramm abgegeben, Speicher für lokale Variablen zur Verfügung gestellt und das Unterprogramm vollständig abgelaufen.

Das Ergebnis wird an Stelle abgelegt, auf welche das aufrufende Programm zugreifen kann und die Ablaufsteuerung an das aufrufende Programm zurückgegeben. **Die Aufrufkonventionen definieren diese Schnittstelle abstrakt.**

Unter ARM sieht die Aufrufkonvention wie folgt aus:

#### Parameter

Die ersten vier Argumente werden in den Registern r0, ..., r3 übergeben.

Alle weiteren kommen auf den Full Descending Stapel.

#### Lokale Variablen

Liegen auf dem Stapel

#### Ergebnis

Rückgabe im Register r0

## Assembler

### Vorbereitungen

#### Entwicklungsumgebung

Die Compiler und Linker können mit

```
sudo apt-get install gcc-arm-linux-gnueabi  
sudo apt-get install qemu-user
```

installiert werden. Eine IDE hab ich noch nicht gefunden ☹

#### Toolchain

```
arm-linux-gnueabi-as file.asm -o file.o  
arm-linux-gnueabi-ld file.o -o file  
qemu-arm file
```

Damit lässt sich das der Quellcode von file.asm nach ./file kompilieren und linken.

### Assembly Code-Snippets

#### Grundgerüst eines Assembly-Programms

Der Aufbau einer Assembly-Datei ist für unsere Zwecke immer eine Abwandlung des folgenden Code-Gerüsts:

```
.arm  
.text  
    string: .ascii „FooBar“  
           stringLength = . - string //damit wir die Länge des Strings wissen  
.global _start  
.align  
  
_start:  
    //Code  
  
    /* Exit Syscall */  
    MOV r0, #0  
    MOV r7, #1  
    SWI #0
```

## Write-Syscall

```
MOV r0, #1
MOV r7, #4
MOV r1, string //der auszugebende String
MOV r2, lengthOfString //lengthOfString kann jede beliebige Zahl sein
```

## Effiziente Multiplikation mit Konstanten

Mithilfe des Barrel-Shifters lässt sich eine Multiplikation mit mit  $2^k \pm 1$  **in einem Taktzyklus** (statt 17 bei MUL) berechnen:

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

## Typische Fehlerquellen

Falsche Notation bei Registerzuweisung

```
LDR r0, #1 //FALSCH
```

statt:

```
LDR r0, =1
```

Oder

```
MOV r0, =1 //FALSCH
```

statt:

```
MOV r0, #1
```

Register in Unterprogramm überschreiben

Gerne wird auch ein Register in einem Unterprogramm überschrieben, zB.: wird r0, r1, ... als Schleifenzähler verwendet und in der Schleife ein Syscall() durchgeführt.

Vermeiden lässt sich das mit STMFD:

```
STMFD sp!, {r0,r2-r6,lr}
```

Damit speichern wir r0, r2 bis r6 und lr in den stackpointer. Mit LDMFD können wir das ganze Set wieder laden (und gegebenenfalls sogar woanders hin speichern):

```
LDMFD sp! {r0,r2-r6,pc} //Beachte: lr wurde direkt nach pc geschrieben, wir springen
                        also direkt an die Rücksprungadresse
```

Ich merk mir diese Befehle immer mit:

**Stack My Full Descending StackPointer!**

**Load My Full Descending StackPointer!**

### Flags werden nicht gesetzt

Nur CMP, CMN, TST, TEQ brauchen keinen Suffix S, um Flags zusetzen. Sonst immer!

### Befehle sind nicht aligned

.align wurde vergessen, oder ein .data-Segment am Ende des Programmes.

### Branches ohne Link nutzen (und dann auf lr zugreifen)

```
_start:
    B loop
    //Richtig wäre hier BL, um den link zu setzen

loop:
    MOV pc, lr
```

## ARM-Befehlsreferenz

Jeder Befehl (fast.. für uns alle) kann mit einer Bedingung verknüpft werden, indem man den Suffix für die Bedingung an den Befehl anhängt:

```
ADDMI //ADD wird nur ausgeführt wenn MI wahr, also das Negativ-Flag gesetzt
ADDHS //ADD wird nur ausgeführt, wenn vorherige CMP auf HigherSame setzte
```

Damit ein Befehl auch die Flags für Bedingungen setzt, muss man den **S** Suffix anhängen:

```
ADDS //setzt die Flags, die möglicherweise auftreten.
```

## Flags

### Negative (N)

Das Negativ-Flag wird auf **1** gesetzt, wenn das Resultat einer Operation **negativ** ist, **ansonsten** ist es **0**.

Negative-Flag: (resultat < 0) -> True: 1, False:0

### Zero (Z)

Ist das Resultat einer Operation gleich Null, so wird dieses Flag auf 1 gesetzt, wenn es irgendetwas anderes als Null ist, ist das Zero-Flag gleich 0

Zero-Flag: (resultat == 0) -> True: 1, False:0

### Carry (C)

Das Carry-Flag wird auf 1 gesetzt, wenn eine Operation ein Carry verursacht (egal ob links oder rechts). Das kann auf folgende Arten geschehen:

Bei Additionen (auch CMN!) wird das Flag gesetzt (=1), wenn die Addition einen unsigned Overflow produziert.

Bei Subtraktionen (auch CMP!) wird das Flag cleared (=0), wenn die Subtraktion „a borrow“ produziert, also einen unsigned Underflow produziert. Ansonsten =1

Bei Shift-Operationen bestimmt das letzte geshiftete Bit den Carry.

Bei allen anderen Operationen bleibt das Carry-Flag normalerweise unberührt.

### Overflow (V)

Das Overflow-Flag wird gesetzt (=1), wenn ein Overflow verursacht wurde. Hah.



Ein Overflow geschieht bei vorzeichenbehafteten Additionen, Subtraktionen und Compares, sobald der Wert  $2^{31}$  übersteigt oder  $-2^{31}$  untersteigt. Damit ist dieses Flag ein arithmetisches Flag, während das Carry Flag grundsätzlich einfach carried Bits darstellt.

## Bedingungen

Alle Instruktionen haben ein 4-Bit-Feld, welches Bedingungen angibt, unter denen eine Instruktion ausgeführt wird. Das ist eine Besonderheit von ARM, welche deutliche Ersparnisse gegenüber Verzweigungen bringen (Nicht ausgeführte Zyklen brauchen einen Takt, statt bei Verzweigung 3 Taktzyklen)

Suffix	Flags	Bedeutung
EQ	Z set	Gleich (Equal)
NE	Z clear	Ungleich (Not Equal)
HS, CS	C set	$\geq$ (Higher Same, Carry Set), vorzeichenlos
LO, CC	C clear	$<$ (Lower, Carry Cleared), vorzeichenlos
HI	C set & Z clear	$>$ (Higher), vorzeichenlos
LS	C clear    Z set	$\leq$ (Lower Same), vorzeichenlos
MI	N set	Negativ (minus)
PL	N clear	Positiv (plus)
VS	V set	Überlauf (overflow set)
VC	V clear	Kein Überlauf (overflow cleared)
GE	N and V the same	$\geq$ (Greater Equal), mit Vorzeichen
LE	Z set    (N and V different)	$\leq$ (Less Equal), mit Vorzeichen
AL	1	Ohne Bedingung, (Always)
NV	0	Reserviert, (Never)

## Arithmetische Operationen

Mnemonic	Kommentar
ADD	Addition
ADC	Addition mit Carry
SUB	Subtraktion
SBC	Subtraktion mit Carry
RSB	Reverse Subtract
RSC	Reverse Subtract mit Carry
MUL	Multiplikation
MLA	Multiply Accumulate

Bits 32-63 werden nach Multiplikation verworfen

### Beschreibung und Information zu den Mnemonics

#### Addition, (Reverse,) Subtraktion

Syntax für die folgenden Mnemonics der Addition und Subtraktion:

```
mnemonic dest, op1, op2  
mnemonic dest, op //-> dest = dest o op
```

#### ADD

Summiert op1 und op2 und speichert es in dest.

#### SUB

Zieht op2 von op1 ab und speichert es in dest, bzw. zieht op von dest ab.

#### RSB

Reverse Subtract subtrahiert op1 von op2 und speichert es in dest.

#### ADC

Summiert op1 und op2, addiert das Carry-Flag (wenn gesetzt) und speichert es in dest.

#### SBC

Zieht op2 von op1 ab bzw. zieht op von dest ab. Wenn das Carry-Flag cleared ist (also nicht gesetzt) dann wird nochmal 1 abgezogen:  $op1 - op2 + carry - 1$

#### RSC

Reverse Subtract subtrahiert op1 von op2, bzw. dest von op und speichert es in dest. Ist das Carry Flag nicht gesetzt (=clear) dann wird davor nochmal 1 abgezogen.

#### Zusätzliche Informationen

##### Conditional Flags

Der Suffix S aktualisiert die Flags **N**, **Z**, **C**, **V** entsprechend dem Resultat.

##### r15

Wird r15 als erster operand (op1) verwendet, wird die Adresse der Instruktion + 8 als Wert für die Kalkulation verwendet.

Wird r15 als dest verwendet, wird auf das Resultat als Adresse gesprungen.

### Multiplikation

#### MUL

Syntax:

```
MUL dest, op1, op2
MUL dest, op
```

Multipliziert op1 mit op2 und speichert es in dest. Dabei werden allerdings nur die 32-niederwertigsten Bits in dest gespeichert!

#### MLA

```
MLA dest, op1, op2, op3
```

Multipliziert op1 mit op2, addiert den Wert von op3 auf das Ergebnis und speichert es in dest. Dabei werden allerdings nur die 32-niederwertigsten Bits in dest gespeichert!

#### MLS

```
MLS dest, op1, op2, op3
```

Multipliziert op1 mit op2, subtrahiert das Ergebnis von op3 und speichert es in dest. Dabei werden allerdings nur die 32-niederwertigsten Bits in dest gespeichert!

### Zusätzliche Informationen

**ACHTUNG:** dest und op1 müssen bei den obigen Instruktionen unterschiedlich sein!

#### Conditional Flags

Der Suffix S aktualisiert das **N** und das **Z** Flag. (C und V nicht!)

## Logische und Vergleichsoperationen

Mnemonic	Kommentar
AND	Bitweiser AND-Verknüpfung ( $a \wedge b$ )
ORR	Bitweiser OR-Verknüpfung ( $a \vee b$ )
EOR	Bitweiser XOR-Verknüpfung ( $a \oplus b$ )
BIC	Bitweiser AND-NOT ( <b>bit clear</b> ) ( $a \wedge \bar{b}$ )
CMP	Vergleich ( $a - b$ )
CMN	Vergleich mit Negation ( $a + b$ )
TST	Test ( $a \wedge b$ )
TEQ	Test Equivalence ( $a \oplus b$ )

### Beschreibung und Information zu den Mnemonics

#### Bitweise Verknüpfungen

Syntax für die folgenden Mnemonics der Bitweise Verknüpfungen:

```
mnemonic dest, op1, op2
```

#### AND

Bitwise AND-Vergleich zwischen op1 und op2. Resultat wird in dest gespeichert

### ORR

Bitwise OR-Vergleich zwischen op1 und op2. Resultat wird in dest gespeichert

### EOR

Bitwise XOR-Vergleich zwischen op1 und op2. Resultat wird in dest gespeichert

### BIC

Das Bit Clear führt ein AND auf op1 und der komplementären Darstellung von op2 aus, das heißt op2 wird umgekehrt und damit wird dann der AND-Vergleich durchgeführt. Speichert in dest.

### Zusätzliche Informationen

#### Conditional Flags

Aktualisiert **N** und **Z** entsprechend dem Resultat.

Möglicherweise kann bei der Berechnung von OP2 das C-Flag gesetzt werden.

### r15

Wird r15 als erster operand (op1) verwendet, wird die Adresse der Instruktion + 8 als Wert für die Kalkulation verwendet.

Wird r15 als dest verwendet, wird auf das Resultat als Adresse gesprungen.

### Compare

Syntax für die folgenden Mnemonics der Compare-Anweisungen

<i>mnemonic</i> op1, op2
--------------------------

### CMP

Subtrahiert den Wert in op2 von op1. Damit ist CMP das gleiche wie SUBS, nur, dass das Resultat verworfen wird.

### CMN

Addiert den Wert in op2 zu op1. Damit ist CMN dasselbe wie ADDS, aber das Resultat wird verworfen.

### Zusätzliche Informationen

#### Conditional Flags

Der Suffix S aktualisiert die Flags **N**, **Z**, **C**, **V** entsprechend dem Resultat.

### Test

Syntax für die folgenden Mnemonics der Test-Anweisungen

<i>mnemonic</i> op1, op2
--------------------------

### TST

Führt eine bitweise AND-Operation von op1 und op2 durch. Damit ist diese Anweisung das Äquivalent zu ANDS, aber wie oben schon wird das Ergebnis verworfen.

### TEQ

Bitweise XOR Operation von op1 und op2 (gleich EORS, aber Ergebnis = futsch).

### Zusätzliche Informationen

#### Conditional Flags

Der Suffix S aktualisiert die Flags **N**, **Z**, **C**, **V** entsprechend dem Resultat.

## Kopier-, Verschiebe- und Sprungoperationen

Mnemonic	Kommentar
MOV	Registerinhalt kopieren
MVN	Bitweise invertierte Kopie
LSL	Logische Linksverschiebung
LSR	Logische Rechtsverschiebung
ASR	Arithmetische Rechtsverschiebung
ROR	Rechtsrotation
RRX	Erweiterte Rechtsrotation
B	Sprung an relative Zieladresse (ohne link)
BL	Wie B, zusätzlich Rücksprungadresse in r14 (lr)

### Beschreibung und Information zu den Mnemonics

#### Kopier-Operationen

Syntax für die folgenden Mnemonics der Move-Anweisungen

```
mnemonic dest, op
```

#### MOV

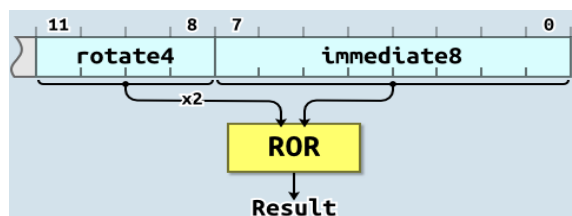
Zusätzlich kann ein Immediate-Wert geladen werden, Syntax:

```
MOV dest, #x
```

Kopiert den Wert in op nach dest, bzw. den Immediate-Wert nach dest.

#x steht stellvertretend für eine Zahl. Diese Zahl muss bestimmten Voraussetzungen entsprechen. Wir wissen, eine ARM-Instruktion lässt uns 12 Bits Platz für Werte. Die 4 höherwertigen Bits werden dabei bei MOV zum Shiften, die 8-niederwertigen für den zu shiftenden (und dann zu ladenden) Immediate-Wert verwendet.

Der 4-Bit Shift-Wert wird dabei jeweils mit 2 multipliziert, sprich jede **positive 2er Potenz** ist möglich, wir erreichen also 0-30 in Zweierschritten.



Gültige Immediate-Werte sind also 0x000000FF oder 0x000FF000.

Beachte, wird ein **Immediate-Wert** geladen, ist der **Suffix S nicht erlaubt!**

#### MVN

Nimmt den Wert in op, führt eine bitweise NOT-Operation darauf an und speichert das Resultat in dest. Es kehrt also das Ergebnis bitweise um.

MVN ist dasselbe wie MOV, nur dass alle Bits invertiert werden. Das heißt, wir erreichen genau das Gegenteil von MOV -> 0xFFFFFFFF lässt sich mit MOV laden, mit MVN müsste man für denselben Wert 0x0 (#0) laden.

#### Zusätzliche Informationen

##### Conditional Flags

Aktualisiert **N** und **Z** entsprechend dem Resultat. Möglicherweise kann bei der Berechnung von OP2 das C-Flag gesetzt werden.

##### Use of PC

MOV wird verwendet, um den Program-Counter (also die aktuell abzuarbeitende Adresse) manuell zu verändern:

```
MOV pc, r8 //lädt die Adresse in r8 nach r15 (pc)
```

#### Verschiebe-Operationen

Syntax für die folgenden Mnemonics der Move-Anweisungen

```
mnemonic dest, op1, op2  
mnemonic dest, op1, #constantShift //range depends on mnemonic
```

##### ASR

Liefert den vorzeichenbehafteten Inhalt von op1, dividiert durch eine Potenz von 2 ( $2^n$ ). Das Vorzeichenbit wird in das linke Bit kopiert.

Der ConstantShift-Range geht von 1-32.

##### LSL

Fügt rechts 0en ein, entsprechend der Anzahl von op2.

Also aus 0110101 wird mit LSL #3 gleich 0101000

Genau betrachtet, wird mit einer Potenz von 2 multipliziert.

##### LSR

Das umgekehrte LSL, das heißt die 0en werden links eingefügt.

Aus 0110101 wird mit LSR #3 gleich 0000110

Genau betrachtet, wird mit einer Potenz von 2 dividiert.

##### ROR

Rotiert die Bits nach rechts, wie in einem Kreisverkehr. Die Bits, die rechts nach außen geschoben wurden, werden links wieder angefügt.

Das heißt aus 0110101 wird mit ROR #3 gleich 1010110.

Um eine Links-Rotation um eins zu bewirken, muss man einfach um 31 Bits nach rechts verschieben.

##### RRX

Hat eigenen Syntax:

```
RRX dest, op
```

Liefert den Wert von dest um eins nach rechts geschiftet. Dabei wird am äußersten linken Bit das alte Carry Flag eingefügt. Ist der S Suffix dabei, wird das alte rechteste Bit in die Carry-Flag gesetzt.

### Zusätzliche Informationen

#### Conditional Flags

Aktualisiert **N** und **Z** entsprechend dem Resultat.

C wird entsprechen dem letzten „hinausgeschifteten“ Bit aktualisiert, sofern der constantShift nicht 0 ist.

#### Restrictions

PC und SP dürfen nicht verwendet werden.

Wird in der LSL-Instruktion 0 verwendet, ist das Ergebnis unter Umständen unvorhersehbar!

### Sprungoperationen

Syntax für die folgenden Mnemonics der Sprung-Anweisungen

<i>mnemonic</i> label
-----------------------

#### *B*

Sprung nach label ohne Link, das heißt die Rücksprungadresse wird nicht nach lr (r14) kopiert.

#### *BL*

Wie B, nur, dass zusätzlich die Adresse der nächsten Instruktion in lr (r14) gespeichert wird. (Damit gleicht der Sprung theoretisch einem Aufruf einer Funktion)

### Zusätzliche Informationen

#### Conditional Flags

Es werden keine Flags gesetzt.

## Speicherzugriffsoperationen

Mnemonic	Kommentar
LDR, STR, SWP	Lese, Schreibe, Tausche 32-Bit
LDRB, STRB, SWPB	Lese, Schreibe, Tausche Byte
LDRH, STRH	Lese, Schreibe 16-Bit (Halbwort)
LDRSB	Lese Byte mit Vorzeichenerweiterung
LDM	Lade 1-16 Register
STM	Speichere 1-16 Register

### Beschreibung und Information zu den Mnemonics

#### Lese-, Schreib-Operationen

Die Lese-, Schreib- Operationen sind „Erweiterbare“ Mnemonics, das heißt, man kann an ihren Befehl einen Typ ({type}) anhängen. Sie werden grundsätzlich zum Laden und Speichern von Registern aus/in eine Speicheradresse verwendet. Zudem lässt sich mittels Immediate-Werte ein Wert direkt laden.

```
mnemonic{type} rg, [ad {,#offset}] //Immediate Offset
mnemonic{type} rg, [ad], #offset //Post-Indexed Immediate Offset
mnemonic{type} rg, [ad, off] //Register Offset
mnemonic{type} rg, [ad], off //Post-Indexed Register Offset
```

**rg** ist das Register in welches **geladen** oder welches **gespeichert** wird.

**ad** ist das Register, in welchem die **Speicher-Adresse** des zu **ladenden Wertes** oder die Adresse, **wohin gespeichert werden soll**, steht.

**#offset** ist ein **Immediate-Wert**, während off ein **Register mit dem Wert** ist.

So kann auch z.b. mnemonic r0, [r1], r2, LSL #3 verwendet werden.

#### Type

##### B

B wird verwendet, um unsigned Byte-Werte zu laden/speichern/tauschen. Darf auch für SWP verwendet werden (siehe Tauschoperationen)

##### H

H steht für Halfword, also für 16-Bit.

##### SB

Dieser Typ ist LDR vorbehalten! Er lädt ein vorzeichenbehaftetes Byte (signed byte)

#### LDR

Um Immediate-Wert zu laden:

```
LDR rg, =const
```

Es ist stark empfehlenswert, immer diese Methode zum Laden von Werten zu verwenden, auch wenn MOV rg, #value effizienter ist. Der Grund liegt darin, dass MOV nicht alle 32-Bit Zahlen konstruieren kann, und sollte es dennoch möglich sein, wandelt der Compiler die LDR-Anweisung direkt in eine MOV-Anweisung um.



LDR rg, =const ist die effizienteste Single-Instruction um eine beliebige 32-Bit Zahl zu laden.

Interessant ist hier zu wissen, dass LDR zum Erzeugen der Zahl jene in einen Literal Pool wirft, ein Teil des Speichers explizit um konstante Werte zu halten. Dieser Pool muss jedoch in der Reichweite von LDR liegen!

### STR

Wird zum Speichern (store) von Registern in den Speicher verwendet. Damit lässt sich also ein Register in den Speicher kopieren.

### Tauschoperationen

#### SWP

Syntax:

SWP{B} dest, source, ad

B ist ein Typ (siehe Seite 48), dest ist das Zielregister, welches die Werte aus source erhält. Source bekommt dafür den Wert in dest. Source und Dest dürfen dasselbe Register sein, dann macht es aber Sinn, auch ad anzugeben. Ad ist eine Adresse im Speicher. Es darf nicht dasselbe Register sein.

Keines der Register darf PC sein. Außerdem empfiehlt die ARM-Instruction-Reference von der Verwendung von SWP ab, sie ist als **deprecated** gekennzeichnet. Stattdessen kann man LDREX und STREX verwenden. Siehe Weiterführende Links.

### Speicher-Operationen

Syntax:

mnemonic{addr\_mode} base{!}, reglist

addr\_mode kann eines der folgenden Suffixe sein:

- ⌘ **IA**: Increment address **nach** jedem Transfer, **default**
- ⌘ **IB**: Increment address **vor** jedem Transfer
- ⌘ **DA**: Decrement address nach jedem Transfer
- ⌘ **DB**: Decrement address vor jedem Transfer

Keinen Plan wofür das gut ist, aber ich schreibs mal hin

**base** ist das Register, welches die ursprüngliche Adresse für den Transfer hält, darf nicht PC sein.

Das Rufezeichen **!** als Suffix kann verwendet werden, um die endgültige Adresse wieder nach base zurückzuschreiben. Manchmal klappts ohne das Rufezeichen einfach nicht.

Die **Reglist** ist eine Liste von mind. einem Register, welches geladen oder gespeichert wird, muss unbedingt in geschwungene Klammern stehen: {r1, r2}.

PC und SP sollte (kann, ist aber deprecated) nicht mehr in der reglist angegeben werden.

### STM

Speichert die reglist in base

### LDM

Lädt aus der base die Werte und trägt sie in die register der reglist ein. Dabei muss die reglist nicht gleich der reglist von STM entsprechen!

## Weiterführende Links und Informationen

### ARM-Infocenter-Referenz

Die ARM und Thumb Instructions:

<http://infocenter.arm.com/help/topic/com.arm.doc.dui0489g/Cihedhif.html>

Die reine ARM Instructions Referenz:

<http://infocenter.arm.com/help/topic/com.arm.doc.dui0068b/CIHEDHIF.html>

## Prozessorarchitektur

Die ARM-Architektur ist eine der unzähligen Prozessorarchitekturen. Das folgende lässt sich also nicht auf die obige Befehlssatzarchitektur/Assembler anwenden, sondern erweitert unser Wissen um den Kontext.

## Klassifikationen

### Anbindung des Speichers

Ganz am Anfang nutzte man noch die **Von-Neumann-Architektur**, in der im **Speicher alle Programme und Daten** lagen.

1944 kam man auf der Harvard-Universität auf die Idee, mit der Harvard-Architektur zwei verschiedene Speicher, für Programme und für Daten, zu verwenden. Zwar ist man weniger flexibel und ist aufwändiger, hat aber mehr Vorteile.

### Anbindung der ALU (tatsächlich Speichers)

#### Register/Register-basiert

ARM-Architektur!

Werte aus zwei Registern werden miteinander verknüpft und durch ALU in ein weiteres Register gespeichert. Speicherzugriffe erfolgen jedoch separat (**Load-Store-Architektur**)

Beide Operanden sind Register, das Ergebnis landet auch in einem Register.

#### Register/Speicher-basiert

Ein Wert aus einem Register wird mit einem Wert aus einem Register *oder* Speicher verknüpft. Wird zum Beispiel bei Intel x86 verwendet

Der erste Operand ist ein Register, der zweite kann ein Register oder Speicher sein, das Ergebnis landet im Register oder Speicher (dann im selben Speicher wie geladen wurde).

#### Akkumulator-basiert

Vereinfachung der Register/Speicher-Architektur, wobei die Daten aus einem Spezialregister (dem Akkumulator) geladen werden und mit einem Speicher verknüpft werden. Das Ergebnis kommt dann wieder in den Akkumulator.

Hilfsregister dienen als Index-Register z.B. bei Schleifenzähler. Akkumulator ist historisch, wird aber vllt bei Internet of Things wieder relevant – verwendet wurde es bei Intel 4040.

### Stapel-basierte

Immer wichtiger, z.B. bei Java VM. Entspricht der Polnischen Notation.

Alle Register werden als Stapel organisiert, dabei werden immer die obersten beiden Werte auf dem Stapel miteinander verknüpft und das Ergebnis wieder auf dem Stapel abgelegt. Dabei ist der Wert unter der Stapelspitze der erste Operand.

### Komplexität des Befehlssatzes

Es kann auch nach der Komplexität des Befehlssatz klassifiziert werden, weil über die Jahre Prozessoren hoch komplex wurden.

### RISC (Reduced Instruction Set Computer)

Wenige, elementare Maschinenbefehle, die sich flexibel kombinieren lassen.

Ermöglichen schlanke Pipelines, um den Richtwert auf einen Taktzyklus pro Stufe zu bringen. Wir brauchen (meist) nur 3 Takte zum Abarbeiten: Fetch, Decode und Store.

Durch eine kompakte Instruktionskodierung gibt es zwar Einschränkungen bei Immediate-Werten, aber dafür können wir sie orthogonal verwenden. Das heißt jedes Register mit jeder Instruktion, etc. **Keine Sackgassen.**

Um die Immediate-Verluste zu kompensieren ist eine Load-Store-Architektur wichtig.

ARM ist stark RISC-nahe, aber natürlich nicht rein RISC

### CISC (Complex Instruction Set Computer)

viele mächtige Spezialbefehle, wie sie derzeit in Mikroprogrammen realisiert sind. Dadurch lassen sich performante Dinge erledigen, die Optimierung ist aber verdammt schwierig, weil Prozessoren oft Unterprozessoren haben, die nicht wissen was die tun.

Oft werden auch nur wenige Untermengen der verfügbaren Befehle verwendet. Intern wird heute meiste eine Kombination verwendet.

### Klassifikation nach Art der Parallelverarbeitung

„Flynn’sche Typologie“, bei der nach zwei Dimensionen gemessen wird:

Anzahl der Befehlsströme und Anzahl der Datenströme, die der Prozessor abarbeiten kann. Oft ist es nämlich sinnvoll, den gleichen Befehl auf eine Vielzahl an Daten anzuwenden. Dann können wir uns den Instruktionsdekoder sparen und mehr in ALUS investieren.

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

SIMD bei Grafikprozessoren!

## Intel x86-CISC-Architektur

1978 erschien die Intel 8086 (16-Bit), zu der dann auch noch 1980 die Intel 8087 FPU erschien, die ein Koprozessor für die Gleitkommazahlen darstellt.

1985 kam der Intel 80386, der eine 32-Bit-Architektur ermöglichte. Ein komplett neuer Adressierungsraum wurde ermöglicht.

1989 Intel 80486 gab es eine Integration der FPU, auf die sich der Entwickler verlassen konnte.

1993 wurde der Intel Pentium veröffentlicht, der die Integration von RISC-Prinzipien umsetzte (und eine komplette neue Pipeline intern verlangte), allerdings gab es dort einen Rechenfehler, der ein riesen PR-Chaos verursachte.

1997 führte der Pentium II 57 neue MMX-Instruktionen (MultiMedia) ein, die (De-)Komprimieren von JPGs und MP3 schnell ermöglichten und damit das Zeitalter von Multimedia in Gang setzten. Dafür nutzte man die SIMD-Technologie für Bilder.

Weil beim Pentium II die SIMD-Technologie über Ganzzahl-Arithmetik realisiert wurde und man rasch merkte, dass damit die neuen Möglichkeiten ausreichend bedient werden können, kam 1999 der Pentium III mit 70 neuen SSE-Instruktionen (Simmed-Streaming) mit Gleitkomma-SIMD für Vektorgrafiken und Codecs.

2001 kam Pentium 4 mit 144 neuen SSE2-Instruktionen.

Der nächste große Schritt war 2003 die 64-Bit-Architektur von AMD, die Intel 2004 übernommen hat, weil ihr eigenes Projekt scheiterte.

2006 wurde Hardwareunterstützte Virtualisierung ermöglicht.

2015 wurde Sicherheit immer bedeutender, wodurch Vertrauenswürdige Laufzeitumgebungen (TEE), also „Enklave“ entstanden, die eine komplett abgeriegelte Umgebung ermöglichen.

## Abwärtskompatibilität

Weil immer mehr aktualisiert und verändert wurde, musste man irgendwie auch Abwärtskompatibilität ermöglichen. Dafür hat Intel mehrere Betriebsmodi eingeführt:

### Real-Mode

Beim Reset wird der Real-Mode aufgerufen, der die Grundlegendesten Dinge tut, eben was der Intel 8086 machte. Will man mehr, kann man aus dem Real-Mode in den Protected Mode übergehen.

### Protected-Mode

Damit konnte man nicht mehr überall hinschreiben, wegen dem Hardware-Unterstützten Speicherschutz. Weil man dachte, der Protected-Mode ist viel besser und niemand will zurück in den Real-Mode, kam man nicht mehr zurück in den Real-Mode.

### Virtual 8086-Mode

Daher kam der Virtual 8086-Mode, der keine Virtualisierung sondern ein eigener Modus ist, der dem Real-Mode entspricht.

### System Management-Mode

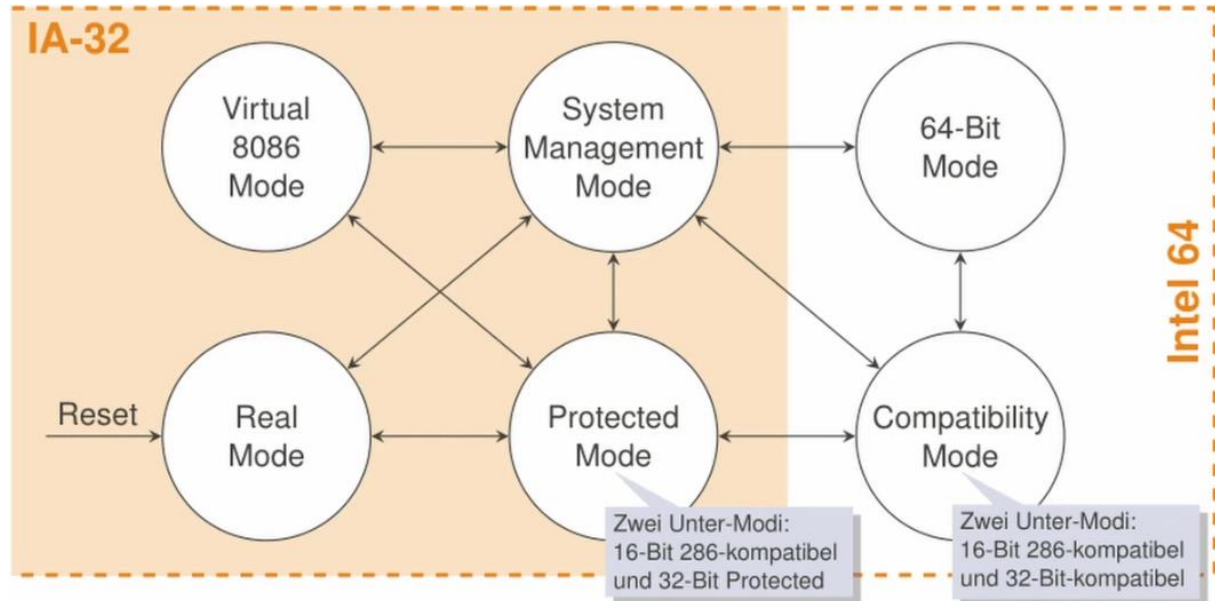
Weil das alles plötzlich kompliziert wurde, führte man den System Management Code ein, aus dem man in jeden anderen Modus gehen konnte und zu dem man aus jedem anderen Modus gehen konnte.

### 64-Bit-Mode

Neuerung, die dank dem System-Management-Mode ermöglicht wurde

### Compatibility-Mode

Joah, da ist viel möglich drinnen und viel kompatibel.



*Emulation ist bei gleicher Performance nur mit Systemen, die ca. 10 Jahre alt sind wirklich möglich.*

### Registersatz

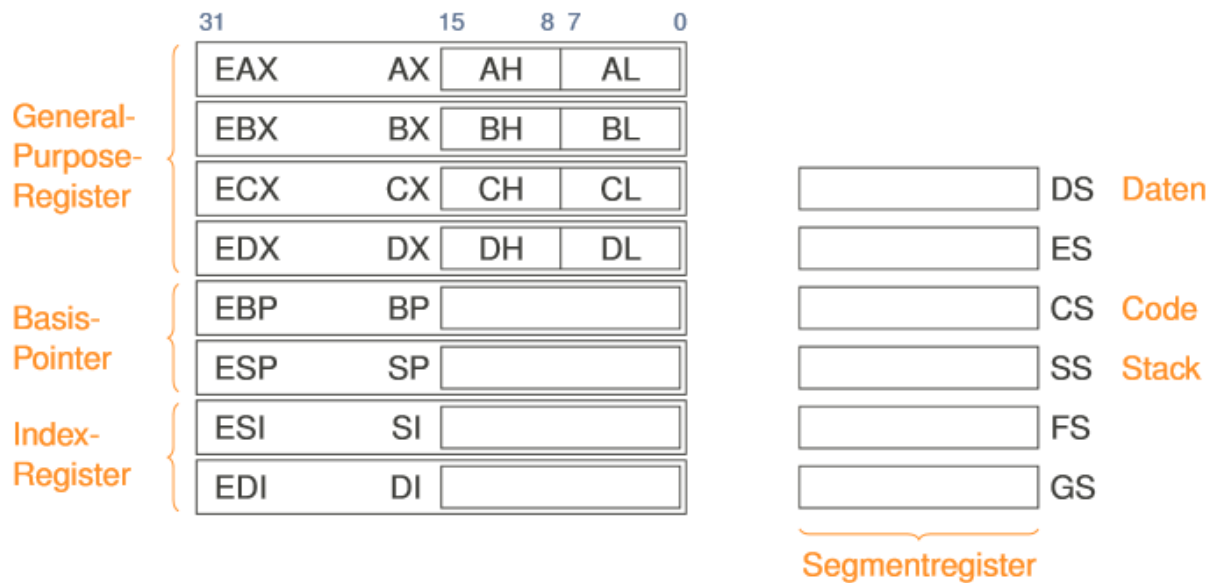
32-Bit-Modus, wie bei ARM, ohne Koprozessor, Kontrollregister, 128-Bit-Media-Register

Ursprünglich waren die Register 16 Bit lang! 32-Bit kam erst später (und hier weiter unten)

4 Register sind für General-Purpose, 2 Basis-Pointer (BP; SP) und 2 Index-Register (Akkumulator), und 4 Segmentregister (2x Daten, Code, Stack)

Die 4 General-Purpose-Register wurden nochmals in je 2 Unterregister aufgeteilt um noch mehr Flexibilität zu erreichen.

Mit 32-Bit erweiterte man einfach die Register um ein E (Extended) als Prefix erweiterte und damit 32-Bit ansteuern konnte.



### ALU-Anbindung

ALU-Befehle haben in der Regel zwei Operanden:

Der erste Operand ist immer Quelle und Ziel gleichzeitig. Durch die Register/Speicher-Architektur darf man aber einen (und nur einen) Operanden als Speicherwort verwenden:

```
ADD AX, mem16
```

### Spezialbefehle

Inkrementieren (und andere) konnte dank RISC-Spezialbefehle sehr performant genutzt werden, die aber oft mit speziellen Registern verknüpft sind.

### Adressierung

Absolut:

```
MOV EAX, wert ; Register mit Inhalt an Speicheradresse wert laden
```

Indirekt: (wie bei ARM)

```
MOV EAX, [EBX]
```

### Instruktionskodierung

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

Für Prüfung aber nicht relevant (glaub i)

### Stapelorganisation

Das Registerpaar SS:ESP (Stack Segment: Extended Stack Pointer) organisiert den full descending Stap über den Register.

Beim Disassemblieren trifft man oft folgende Spezialbefehle an:

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

## FPU (Gleitkomma-Einheit)

Ist in unserem Beispiel ein Koprozessor, aber dient hervorragend als Beispiel für Stapelorganisation und FPU (was bisher noch nicht besprochen wurde).

Die Stapel-basierte Befehlssatzarchitektur besitzt 8 Datenregister zu je 80 Bit (links, sowie einem 16-Bit Statusregister.



Weil es ein full descending Stapel ist, sprechen wir die Daten nicht über R1, R2, ... an, sondern verwenden den Stackpointer  $ST_0$ ,  $ST_1$ , ... . Entsprechend würde der Stackpointer  $ST_0$  beim Hinzufügen von Daten nach unten wandern.

Das Statusregister rechts enthält 3 Bits für den Top-Pointer (also der Pointer auf  $ST_0$ ), 1 Bit für StackFault (gibt an, ob der Stapel übergelaufen ist), sowie einige Flags.

Das 16. Bit (15) gibt an, ob die FPU belegt ist, weil die FPU parallel zur CPU arbeitet.

## Wichtigste FPU-Befehle

### Datentransfer

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

Wichtig ist, die Zahlenkonvertierung zwischen den Ganz- Gleitkomma und BCD-Zahlen übernimmt zur Gänze die FPU! Weiteres dürfen wir keine Immediate-Werte laden, bekommen dafür aber die wichtigsten mathematischen Konstanten über den ROM.



## Arithmetik-Befehle

Mnemonics	Kommentar
FADD FADDP F <b>I</b> ADD	Gleitkoma-Addition
FSUB FSUBP F <b>I</b> SUB	Gleitkoma-Subtraktion
FMUL FMULP F <b>I</b> MUL	Gleitkoma-Multiplikation
FDIV FDIVP F <b>I</b> DIV	Gleitkoma-Division
FSQRT	Gleitkoma-Quadratwurzel
FABS	Absolutbetrag
FRNDINT	auf Ganzzahl runden

Alle Gleitkoma-Operationen erfolgen nach IEEE 754 (Seite 28), die Varianten mit **I** erlauben Ganzzahlen als ersten Operanden und für jede Subtraktion und Division existieren „Reverse“-Varianten (FSUB**RP**, FIDIV**R**)

## Beispiel

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

## Assembler-Befehlsfolge

sprod:

```

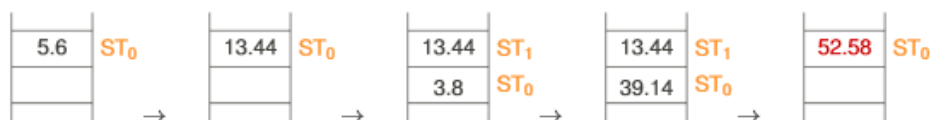
FLD    wert1
FMUL   wert2
FLD    wert3
FMUL   wert4
FADDP

```

## Belegungsbeispiel

Variable	Wert	Label (Speicheradresse)
$a_1$	5.6	wert1
$a_2$	3.8	wert3
$b_1$	2.4	wert2
$b_2$	10.3	wert4

## Ablauf

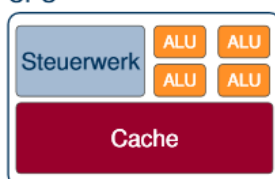


## Datenparallele Architekturen

Früher steigerte man die Leistung eines Computers durch die Erhöhung der Taktfrequenz. Weil diese bei ~4 GHz aber nicht mehr erhöhbar ist, musste man die Rechenleistung neu bemessen: GFLOPS. Diese GFLOPS geben die tatsächliche Rechenleistung eines Computers wieder und können durch mehrere Kerne multipliziert werden. So kann man die Leistung steigern, obwohl man die Taktfrequenz senkt.

Das Prinzip dahinter nennt man Datenparallele Architektur, wobei wenige Befehle einfach parallel auf viele Daten angewandt werden. Der Ursprung dieser Idee fand in der Spieleindustrie Anwendung, indem man die Chipfläche für mehr ALUs verwendete:

## CPU



## GPU

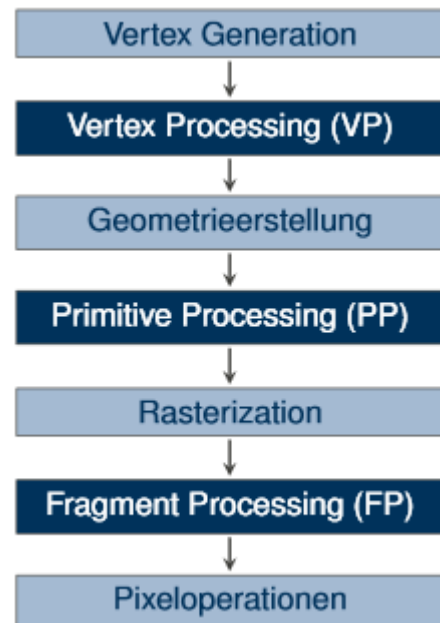




Man konnte damit sehr **effizient, gleichzeitig, gleichartige Daten** auf **gleiche Weise** bearbeiten. Dazu darf der **Kontrollfluss nicht von Zwischenergebnissen abhängen** (Keine Überprüfung „Add, wenn Division mit Rest“) und sollten **geringe Datenabhängigkeit** aufweisen.

### Grafik-Pipeline mit Hardwareunterstützung

Will man Vektorgrafik erzeugen, müssen die Punkte im Koordinatensystem berechnet, welche auf den Bildschirmbereich projiziert werden müssen (Spiegelung, Verzerrung), was natürlich programmierbar sein muss. Die Geometrieerstellung aufgrund dieser prozessierten Vektorpunkte erfolgt auf fester Logik und ist nicht programmierbar – erzeugt dann eben einfache Geometrische Objekte. Diese Objekte werden dann durch Primitive Processing gerastert werden (also tatsächlich am Bildschirm ausgegeben). Sie liegen dann eben im Speicher und werden durch die Rasterization mittels programmierbarer Fragment Processing zu Pixeln umgewandelt, auf die zum Schluss nochmals Pixeloperationen angewandt werden können.



Damit das alles flüssig laufen kann, brauchte die FPU ca 100 Kerne, welche in einer Matrix-Struktur organisiert sind. In jedem einzelnen Kern teilen sich 10 ALUs ein Steuerwerk, welche auf dem **SIMD**-Prinzip arbeiten.

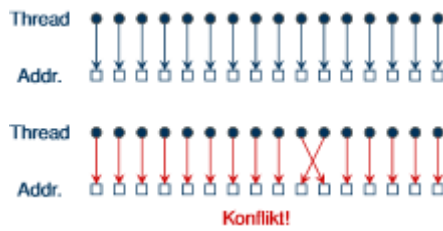
Weil das Speichern der Daten oft länger dauerte als die Operationen auf der ALU (Speicherlatenz =  $10^2 - 10^3$  Taktzyklen!), führte man Hardware-Threads in jedem Kern ein, um die überschüssige Zeit besser zu nutzen. Das heißt, die Register liegen gespiegelt vor und in Hardware umgeschaltet werden, welcher Registersatz verwendet wird. Während dann ein Registersatz auf die Werte wartet, kann mit den anderen Registern die ALU verwendet werden.

Weil man schnell merkte, dass die Grafikkarten so verdammt viel konnten, versuchte man die Grafikkarten noch ein wenig allgemeiner zu machen: **GPU, General Purpose GPU** wurde erschaffen, für beliebige Rechenaufgaben.

### Optimierung für Datenparallelität

Angenommen wir hätten eine SIMD-Breite von 32, d.h. es müssen mind. 32 Threads müssen das Gleiche tun, wobei wir Bedingungen verwenden dürfen. Das heißt, pro Thread dürfen wir Flags setzen (z.b. nur jeder 4te Thread tut was). Dann pausieren die anderen Threads. Logischerweise haben wir nur dann maximale Performance, wenn wir alle nutzen.

Weiteres Problem ist, dass wir keinen Cache haben, das heißt die Daten müssen im Speicher so liegen, dass wir sie parallel verwenden können. Ist dies nicht der Fall, haben wir eine 16-Fache Verzögerung!!

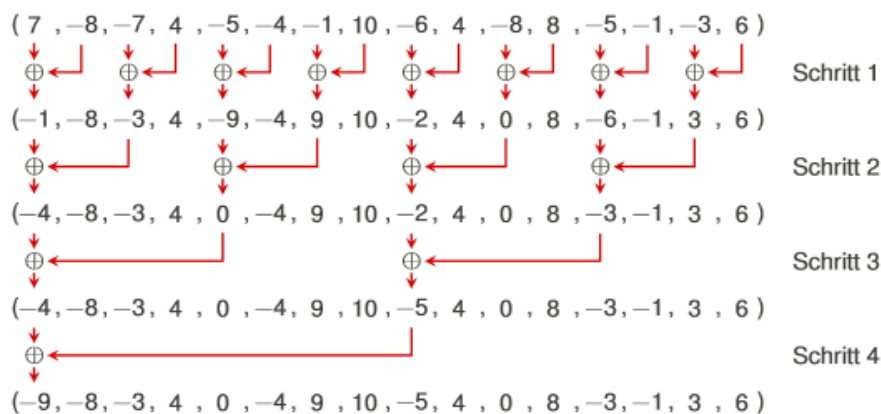


*Grundsatz: In Datenstruktur statt in Algorithmus denken!*

Also wie muss ich die Daten platzsparend und parallel ablegen, damit wir sie optimiert nutzen können.

Beispiel: Parallele Reduktion

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



## Ein- & Ausgabe

Weil Maus und Tastatur schon uralt sind, nähern wir uns modernen Technologien

## Touch-Eingabetechnologie

Verschiedene Technologien zur Berührungsmessung sind:

- ⌘ Lichtintensität: Infrarotgitter, Kamera-basiert
- ⌘ Spannung: Resistive Verfahren
- ⌘ Strom: Oberflächen-kapazitive Verfahren
- ⌘ Kapazitätsänderung: Menschliche Berührung
- ⌘ Zeitverzögerung: Zeitmessung bei Schwingung des Glases
- ⌘ Kraft

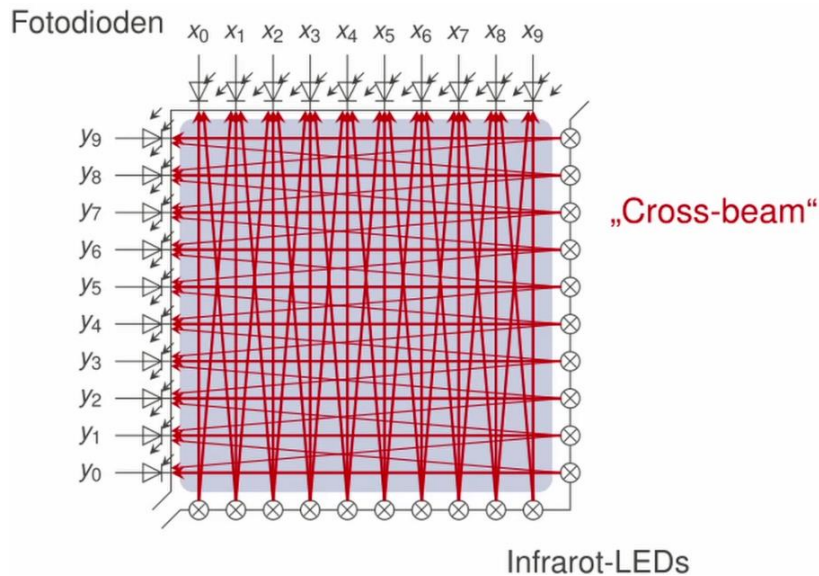
Keines davon ist das Beste! Wir müssen es nach bestimmten Kriterien an unsere Anwendung anpassen und vergleichen:

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

## Optische Berührungsmessung

Dabei werden einfach Infrarot-LEDs an ein Ende des Displays gelegt, welches Infrarot-Strahlen über das gesamte Display projiziert und am anderen Ende mittels Fotodioden gemessen. Wird das Display berührt, wird das „nicht vorhanden sein“ von LED-Strahlen gemessen und man kann die Koordinaten des Berührungspunktes messen.

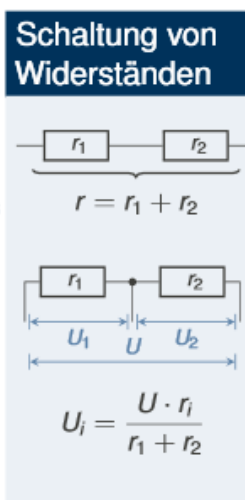
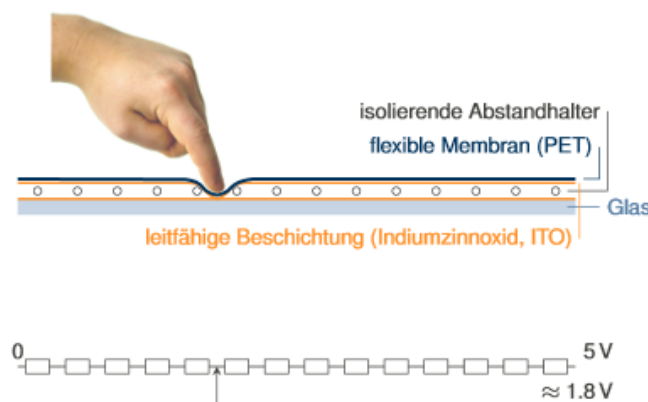
Mittels Cross-Beam konnte die Genauigkeit gesteigert werden.



## Resistive Berührungsmessung

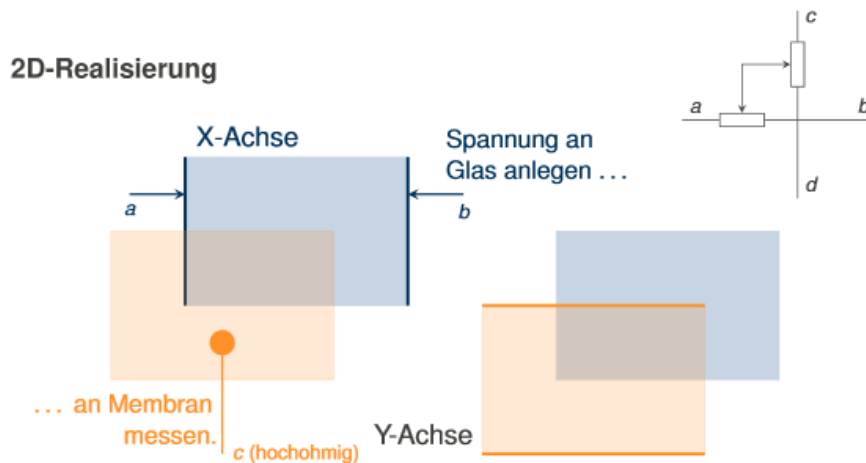
Wir haben zwei durchsichtige Schaltungen, getrennt von Abstandshalter, drüber eine Membran zum Schutz. Die leitfähige Beschichtung ist ein konstanter Widerstand, wo über die gesamte Fläche (hier Linie) ein Widerstand abgetragen wird. Wir legen also auf der einen Seite 0, auf der anderen Seite z.B. 5 Volt an. Dank der Ohm'schen Regeln halbiert sich der Widerstand zwischen zwei Widerständen, wenn diese gleich groß sind ( $r_1$  und  $r_2$ ). Damit können wir leicht berechnen, zwischen welchen zwei Widerständen „gedrückt“ wurde, wenn wir die Spannung dort messen (1,8 Volt)

### 1D-Modell



**Indium Zinnoxid** ist die Grundlage aller modernen Touch-Technologien, weil es das Einzige **leitende und durchsichtige Material** auf der Erde ist. Damit sind durchsichtige Schaltungen möglich.

Für die 2D-Realisierung verwenden wir selbige Idee, nur mit zwei Achsen:



### → Alternierende Messung von X- und Y-Koordinate

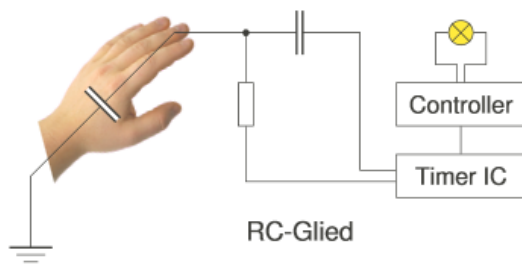
Wir legen zuerst die Spannung am Glas an und können an Membran hochohmig messen (zusätzlichen Spannungswiderstand ignorieren), dann wissen wir, wo wir auf der X-Achse liegen. Dann wird dasselbe an Membran getan, um die Position auf der Y-Achse zu messen.

Das heißt, wir können defacto mit jedem Gegenstand messen, weil ja die Membran auf das Glas gedrückt wird. Gerade dieser Umstand macht diese Technologie aber unnütz für Smartphones, weil wir dann eine Membran überm Display hätten.

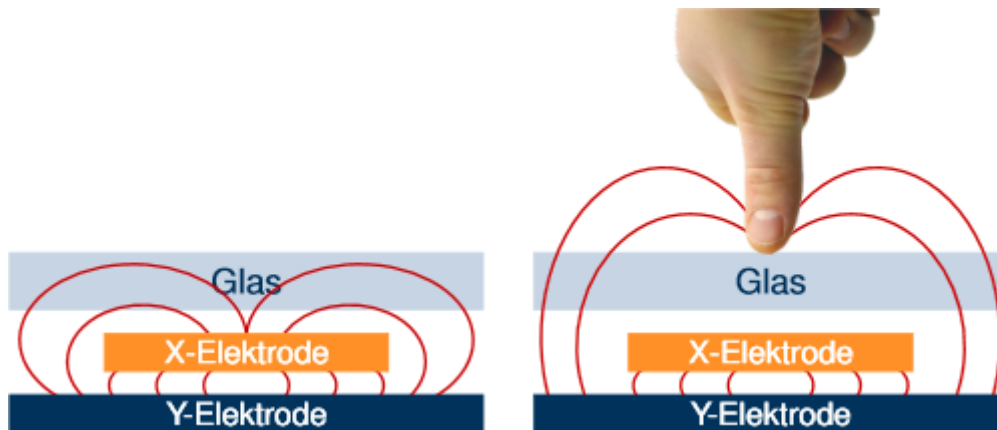
### Kapazitive Berührungsmessung

Das ist jene Technologie, wie sie in Smartphones verwendet wird.

Dabei misst ein Timer-Chip den Schwingkreis eines RC-Glieds ( $R$ =Widerstand,  $C$ =Kapazität). Der Timer-IC ersetzt die Spule, die man sonst benötigen würde. Der Timer-IC gibt die gemessen/gezählten Schwingungen an einen Controller, der dann abhängig von einer bestimmten Logik den Strom fließen lässt oder nicht.



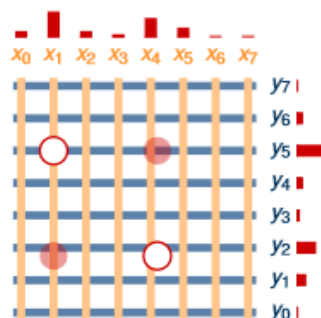
Der Trick dabei ist ganz einfach, dass Menschen wandelnde Kondensatoren sind. Das heißt Menschen sind immer geerdet, sie verändern also die Schwingung des Kreises, wenn sie damit in Berührung kommen. Glücklicherweise funktioniert diese Schwingung auch durch Glas hindurch.



Die zwei Elektroden sind Anschlüsse eines Kondensators, damit wir die Kapazität aufbauen können, ohne mehrere Kondensatoren-Bausteine einbauen zu müssen. Dadurch entstehen Schwingungen/ein elektrisches Feld zwischen den zwei Elektroden. Bei einer Berührung mit dem Finger stiehlt der Finger die Ladung von der X-Elektrode. Die Kapazität zwischen den Elektroden verändert sich.

Diese Berührung muss jetzt nur noch im zweidimensionalen Raum gemessen werden. Dabei gilt zu beachten, dass die X-Elektroden „schmäler“ sind, weil sie gekreuzt übereinanderliegen:

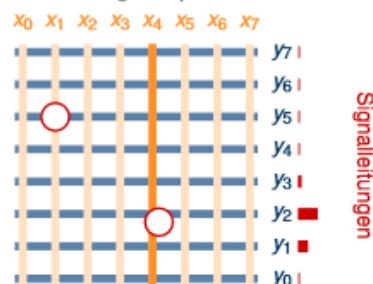
#### Variante 1: Perimeter Scan



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

#### Variante 2: Imaging

Steuerleitungen: Spaltenauswahl

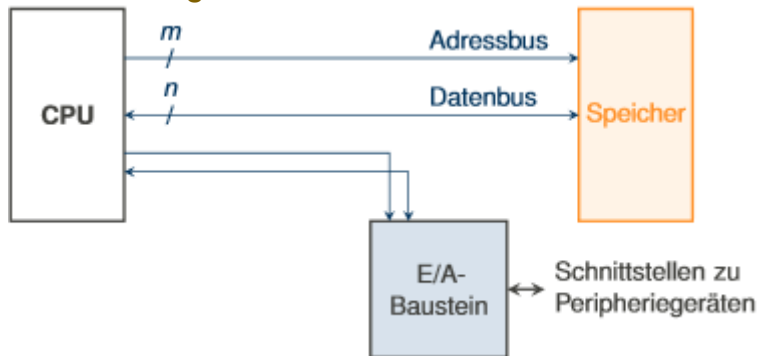


- ▶ Smartphones z. B.  $9 \times 16, 20 \text{ Hz}$
- ▶ Interpolierte Auflösung:  $1024 \times 1024$ , ca. 16 Punkte

Bei der zweiten Variante sind nur noch die y-Leitungen Signalleitungen wie links. Danach wird Spalte für Spalte über das gesamte Feld über die X-Spalten „gescannt“, damit lässt sich genau bestimmen, wo der Berührungspunkt liegt. Durch Interpolation kann die Auflösung sehr hoch aufgebracht werden.

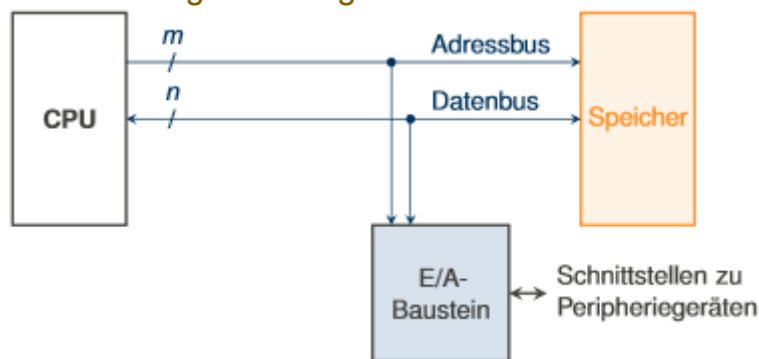
## Ansteuerung von E/A-Bausteinen

### Ansteuerung über Ports



CPU hat eigene Instruktionen (z.b. in, out), mit denen die Peripheriegeräte angesprochen werden können.

### Ansteuerung über Register



In modernen Architekturen wird das Peripheriegerät als Parasit an den Adressbus und Datenbus angehängt und hat bestimmte Speicherbereiche „gemappt“. Das heißt, greife ich mit den gewohnten Instruktionen auf diesen parasitären Speicherbereich zu, bekomme ich Zugriff auf den E/A-Baustein.

Diese Eigenschaften werden in den Spezifikationen und Datenblättern zu den entsprechenden Architekturen angegeben und müssen nicht auswendig gelernt werden!

## Typen von E/A-Registern

### Kontrollregister

Zur Initialisierung und Funktionswahl, d.h. zur Auswahl, ob Leitung als Eingang oder Ausgang verwendet werden soll, bzw. Aktivierung von Unterbreuchungsanforderungen.

### Datenregister

Sie puffern die ein- oder ausgehenden Daten. Weil E/A-Geräte verarbeitete Daten langsamer oder asynchron zur CPU arbeitet, braucht es eben den Puffer, der meist als FIFO-Puffer realisiert ist.

### Statusregister

Sind meist nur Leseregister zum Auslesen von Zustandsinformationen, wie Verfügbarkeit, Abschluss oder Timer.

Sie werden regelmäßig vom Hauptprogramm abgefragt (*polling*).

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

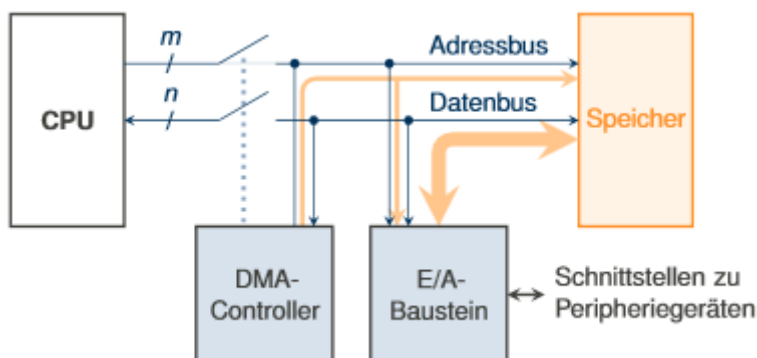
Daher muss der Programmierer die gelesenen Daten selber zwischenspeichern, wenn er sie selbst mehrfach verwenden will.

## Speicherdirektzugriff

Weil es unnötig viel Overheat produzieren würde, Daten aus dem E/A-Baustein in den CPU zu holen, damit die CPU diese dann in den Speicher ablegen kann, gibt es einen Direktzugriff, der es ermöglicht Daten direkt aus dem E/A-Baustein in den Speicher zu legen.

Dafür brauchen wir aber einen DMA-Controller (Direct Memory Access). Dieser übernimmt die Koordination zwischen Speicher, E/A-Baustein und CPU, weshalb er privilegiert arbeitet. Er darf den Adressbus und Datenbus zwischen CPU und Speicher abkoppeln, damit der Zugriff für den E/A-Baustein zur Verfügung steht. Die CPU kann dann andere Dinge tun (außer auf den Speicher zugreifen).

Wichtig hierbei ist, dass der DMA-Controller den Adressbus kontrolliert. Er bestimmt, auf welche Adresse geschrieben wird und übergibt diese zwischen Speicher und E/A-Baustein. Der Baustein verwendet dann den Datenbus zum Austausch der Daten.



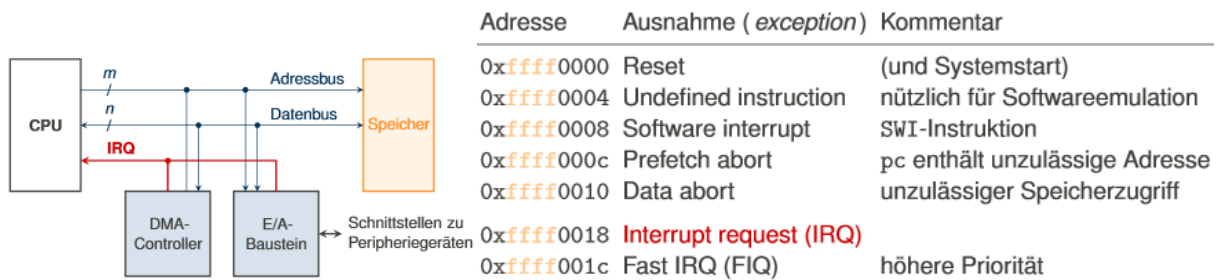
## Unterbrechungsanforderung

Das sogenannte „busy waiting“ ist ein Dilemma, in dem wir unsere wichtigste Komponente, die CPU, auf etwas bestimmtes warten lassen müssen. Z.b. bis die Zeit um ist. Das ist total unnötig und muss irgendwie anders gelöst werden. Eben mit Unterbrechungsanforderungen.

Dabei darf der E/A-Baustein ein Signal schicken, dass er fertig ist mit seiner Arbeit und den Prozessor anweisen den Baustein jetzt zu nutzen. Dazu liegt eine sogenannte IRQ-Leitung an CPU, DMA und E/A-Baustein an, die der E/A-Baustein jederzeit auf High setzen kann. Die CPU bleibt aber Chef im Haus und kann selber entscheiden, was es mit dieser Anforderung tut. Tatsächlich ist es bei ARM so realisiert, dass nach jeder Instruktion und vor jedem Fetch geprüft wird, ob eine der vielen IRQ-Leitungen auf high liegen. Die CPU legt die aktuelle Ausführungsadresse dann ins LinkRegister und führt die Sprunganweisung (B), welche in der EVT (Exception Vector Table) und zur Speicheradresse des Handlers führt, aus.



Genau das passiert auch bei unserem bekannten Software Interrupt in ARM. Dabei wird nichts anderes getan, als an die Speicheradresse des Linux-Syscall-Handlers zu springen.



Wobei der Fast IRQ einfach eine höhere Priorität hat.

## Interrupt-Handler

Wenn wir einen Interrupt-Handler programmieren, müssen wir einige Punkte besonders hervorheben:

- Der Handler muss unbedingt alle Register sichern und vor dem Beenden wiederherstellen
- Interrupts müssen vom Handler wieder ab geschaltet werden, weil es in ARM nur eine Leitung gibt und die sonst nicht unterschieden werden kann
- Weil der Interrupt vor dem Fetch aber nach der Instruktion ausgeführt wird, steht der programcounter bereits auf den nächsten Befehl. Würde ich daher nun einfach an den gesicherten lr (=der ja pc ist) zurückspringen, verpasse ich einen Befehl. Wir müssen daher entsprechend unserer Architektur um ein paar Bits zurückspringen (Bei ARM 32-Bit = 4 Bit zurückspringen)

irq:

```
STMFD    sp!, {r0-r12,lr}
BL       do_something

LDR      r0, =timerbase
STR      r0, [r0+0x0c]

LDMFD    sp!, {r0-r12,lr}
SUBS     pc, lr, #4
```

- Wie zu sehen ist, werden auch die Flags gesetzt: Damit setzen wir die IRQ-Flags zurück und die CPU kann die ursprünglichen Flags vor dem IRQ wiederherstellen
- Gibt es mehrere IRQ-Quellen, muss der Interrupt-Handler die Quelle herausfinden (durch interrupt pending Bits) und entsprechend Aufrufen

Hierbei noch einmal der Verweis auf Seite 31 „Registersatz mit Bänken“.

## Mehrprozessbetrieb auf einem Kern

Auch „Time sharing“ oder „Multitasking“ genannt. Dabei kann ein einzelner Kern mehrere Prozesse ausführen, indem jeder Prozess einen kurzen Anteil an Prozessorzeit bekommt und dann wird mittels Interrupt dem nächsten Prozess Rechenzeit zugeteilt.

Dies übernimmt meist das Betriebssystem, sodass das Betriebssystem einem Interrupt basierend auf einem Timer aufruft, Register sichert, den Stack-Frame anpasst und die Zugriffsrechte ändert. Dann wird das Register des nächsten



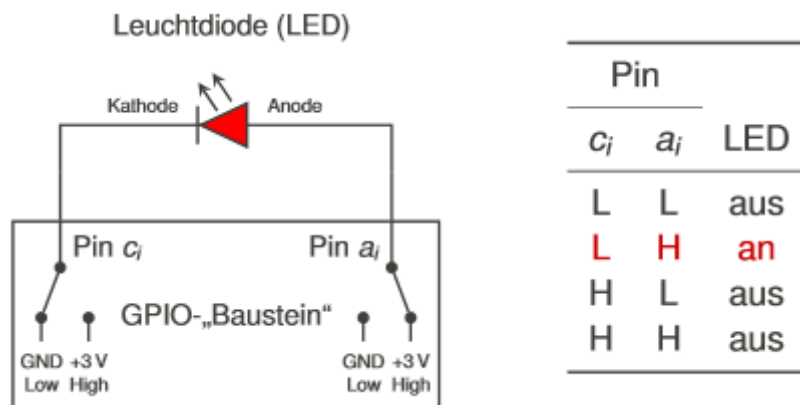
Prozesses wiederhergestellt und mit einem Rücksprung geht es weiter für den nächsten Prozess.

Mehr dazu aber in Betriebssysteme im 2. Semester.

## Ausgabe (Praxisbeispiel)

Zum besseren Verständnis wird ein Beispiel mithilfe des Raspberry Pi's mit ARM1176 und GPIO-Pins ein Praxisbeispiel durchgemacht.

Wir wollen über die GPIO-Pins LEDs zum Leuchten bringen. Dabei lässt nur ein Zustand von zwei Pins eine LED leuchten:



## Initialisierung

Wir müssen zuerst die LEDs initialisieren und auf **aus** stellen. Dazu müssen wir uns die Dokumentation ansehen, wie wir die Pins ansteuern können, diese Adresse dann in ein Register laden und die Pins hinter dieser Adresse mit einer Bit-Kodierung (welche ebenfalls in der Dokumentation auffindbar ist) auf „Ausgang“ stellen.

```
LDR  r9, =0x20200000 ; Basisadresse der GPIO-Komponente
MOV  r0, #4           ; Bit-Kodierung 100 für „Ausgang“ nach r0
STR  r0, [r9, #0]     ; setze Funktion in GPFSSEL-Register
...                    ; wiederholen für weitere Pins
```

Dies wiederholen wir für alle Pins, welche wir verwenden wollen (Kathoden und Anoden)

Im nächsten Schritt müssen wir (um die LED auszuschalten) die Kathoden auf High und die Anoden auf Low stellen. Das heißt, für jeden Kathoden-Pin müssen wir im GPSET-Register (welcher 32 Bit -> 32 Pins adressieren kann) das entsprechende Bit auf 1 stellen. Damit stellen wir den Pin auf High (die 0en haben **KEINE** Auswirkung auf die Pins! D.h. die werden nicht auf Low gestellt.)

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

Und das selbe machen wir für die Anoden-Pins, nur dass wir diese auf Low stellen

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

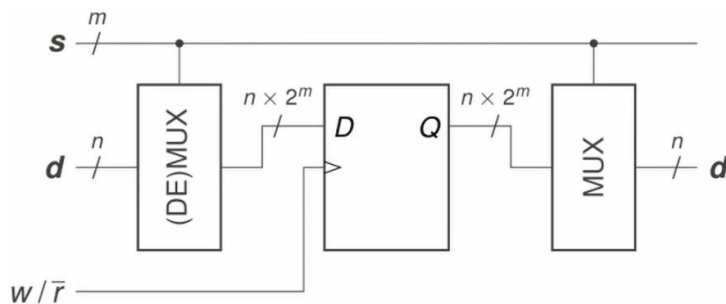
Mit diesem Clear-Register können wir den Ausgang auf Low stellen (überall wo das Bit auf 1 steht, wird der entsprechende Pin auf Low gestellt).

Der Rest ist mir zu anstrengend hier zusammenzufassen. Schauts euch einfach selber an entweder unter „/Technische Informatik/Vorlesungen (Video)/10-tech-vo.mp4“ oder die entsprechende Vorlesungsfolie „/Technische Informatik/Folien/Vorlesung/vor10\_ein\_ausgabe.pdf“

Greetz.

## Speicher

Bisheriges Speichermodell:



Dabei wird über einen Adressbus  $s$  angesprochen, der nur eine Steuerleitung für den Demultiplexer und Multiplexer darstellt. Der Speicher selber ist eine riesige Anzahl an D-Flipflops, denen wir über den Demultiplexer Daten geben, welche gespeichert werden und über den Multiplexer die Daten wieder anlegen. Dafür verwenden wir die Taktleitung um Lesen und Schreiben zu unterscheiden.

In der Theorie funktioniert das super, in der Praxis haben wir zwei Probleme: Kosten, aber auch die Persistenz ist scheiße. Ist der Strom weg, kann ein D-Flipflop den Inhalt nicht halten und vergisst ihn beim erneuten zuführen von Strom.

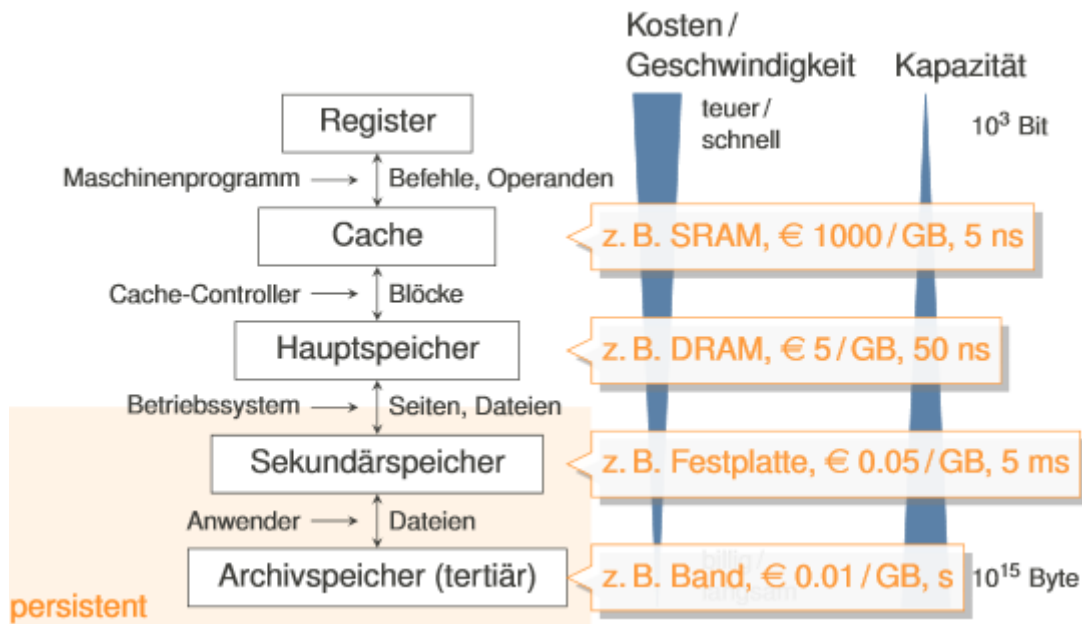
Daher suchen wir andere Möglichkeiten (welche historisch gewachsen sind):

- ⌘ **Modifikation von Struktur:** Lochkarten, Schallplatten
- ⌘ **Magnetismus:** Magnetband, Diskette, Festplatte
- ⌘ **Elektrische Ladung:** Kondensator, isoliertes Gatter (Flash)
- ⌘ **Rückkopplung:** Flipflop, Schwingkreis (meist nicht persistent)
- ⌘ **Optik:** Barcode, CD-ROM, DVD

Dabei gilt es **Persistenz**, **Geschwindigkeit** (Zugriff, Übertragung), **Kapazität**, **Dichte**, **Energiebedarf**, **Robustheit** und natürlich **Kosten** zu vergleichen.

## Speicherhierarchie

Wir müssen verschiedene Stufen der Speicherhierarchie unterscheiden, das heißt, wo befinden sich Daten gerade:



Desto weiter wir runter gehen in der Hierarchie, desto billiger/aber langsamer wird es, dafür können wir mehr speichern.

Register und Cache sind klar, der Hauptspeicher stellt dabei RAM dar und der Sekundärspeicher ist z.B. eine Festplatte. Dieser Sekundärspeicher ist nicht über Adressen erreichbar!

## Definitionen und Erklärungen

### Maßeinheiten

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

**Aufgepasst!** Kapazitäten von Speicherbausteinen (z. B. Kilobyte, Megabit) werden immer noch binär angegeben. Bei Speichermedien und in der Datenübertragung ist die dezimale Interpretation verbreitet.

### Begriffe/Akronyme

#### Flüchtiger Speicher

RAM (Random Access Memory)

Speicher mit wahlfreiem Zugriff auf beliebige Adressen

### SRAM (Static RAM)

Ist relativ gut vergleichbar mit D-Flipflop: 4-8 Transistoren pro Bit. Aber Transistoren sind scheiße teuer! Dafür unglaublich schnell.

### DRAM (Dynamic RAM)

Der Inhalt wird nach Auslesen und durch Entladen eines Kondensators im Zeitverlauf vergessen, braucht aber nur einen Transistor pro Bit, kann damit dichter und billiger als SRAM bauen.

### SDRAM (synchronous RAM)

Steuerleitungen wirken bei steigender Flanke des Speichertaktes.

Steuerleitungen sind an Flanke eines Speichertaktes gekoppelt. Wir haben also einen Prozessortakt und einen langsameren Speichertakt. Der SDRAM koppelt die Ansteuerung des Speichers direkt an den Speicher, weil der Controller das alles im langsameren Speichertakt macht, anstatt es beliebig zu tun.

### DDR-SDRAM (Double Data Rate SDRAM)

Wobei wir mittlerweile bei Triple und Quadrupel-Data Rate sind :P

Wir steigern die Leistung durch Auslesen mehrerer benachbarter Bits pro Zugriff.

### Persistenter Speicher

#### ROM (Read Only Memory)

Nur Lesezugriff, wird bei der Herstellung beschrieben und kann nicht geändert werden.

#### PROM (programmable ROM)

Ermöglicht einen einmaligen Schreibzugriff (z.B. durch Durchbrennen von Sicherungen bei der Herstellung, damit allgemeinere Chips für spezielle Dinge verwendet werden können.

#### EPROM (erasable PROM)

elektrisch programmierbares („brennen“) und durch UV-Licht lösches PROM, wobei bereits Sonnenstrahlen den PROM löschen können.

#### EEPROM (electrically erasable PROM)

Löschen und Wiederbeschreiben geschieht elektrisch (=> Das nennen wir heute Flash-Speicher)

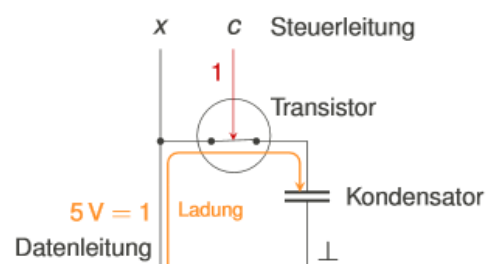
## Dynamischer RAM (Hauptspeicher)

Erinnerung: 1 Transistor

### Schreiben einer Zelle

Will man nun in eine DRAM-Zelle schreiben, setzen wir die **Datenleitung x auf das abzuspeichernde Potenzial**.

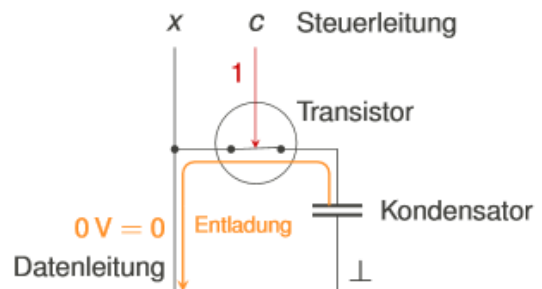
Die **Steuerleitung wird auf c = 1 gesetzt**, damit der Transistor leitet und wird **nach Abschluss auf c=0** gesetzt.



## Lesen einer Zelle

Wir schalten die **Datenleitung an den Eingang des Leseverstärkers**, der an Ende der Datenleitung hängt. (y = Ausgang)

Dann setzen wir die **Steuerleitung auf c = 1**, sodass der Transistor leitet. Der **Transistor entlädt sich** und wir erhalten am **Leseverstärker** einen (sehr kurzen) **Impuls**.



Nun ist zum einen der **Kondensator leer**, zum anderen aber wissen wir am Leseverstärker nach dem Impuls nicht mehr, was wir gelesen haben.

Der Leseverstärker hat also ein **stabiles Flipflop**, welches für alle DRAM-Zellen, welche an diesem Verstärker anliegen verwendet wird. Damit wird der **Impuls angehalten** (siehe **Signalverlauf**)

Das neue Problem ist: Ich weiß damit, dass in unserer Zelle eine 1 stand, ich kann sie auch weiterverwenden, aber sie steht nicht mehr in der Zelle drinnen ☹

Zusätzlich haben wir in unserer dichten Bauweise „Leckströme“. Das heißt, die Kondensatoren entladen sich selbst. Wir suchen also eine Logik, dass unsere Speicherzellen den Wert behalten. Wir nennen dies „**DRAM-Auffrischung**“.

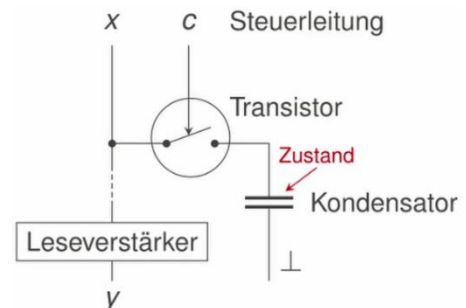
## DRAM-Auffrischung

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

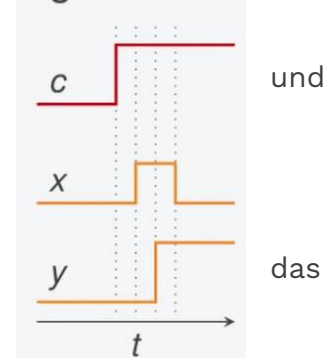
**Abhilfe:** Nach dem Auslesen und zyklisch, ca. alle 30-60 Millisekunden, wird der aktuelle Wert erneut geschrieben. Diese Aufgabe übernimmt i.d.R. der Memory-Controller die Ansteuerungslogik.

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

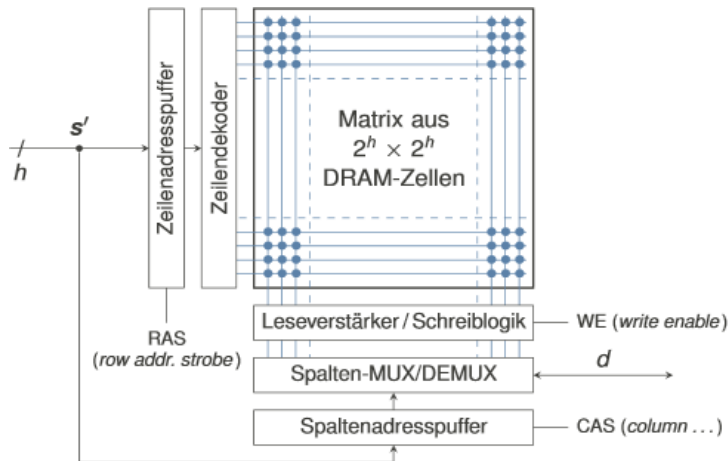
Die Leckströme sind jedoch Proportional zur Spannung, heißt, je weniger Spannung, desto weniger Leckströme, desto weniger Stromverbrauch.



## Signalverläufe



## Aufbau von DRAM-Bausteinen



Über den Zeilendekoder wählen wir eine Zeile, für die Spalten brauchen wir aber einen (De-)Multiplexer, weil wir die Daten ja auch einführen und auslesen müssen.

Die Leseverstärker sind Spaltenweise eingebaut, in denen auch die Schreiblogik eingebaut ist.

*Wie adressieren wir nun?*

Wir müssen den Adressbus  $h$  aufteilen in eine Wahl der Zeile und eine Wahl der Spalte aufteilen.

*Wie funktioniert der Zugriff?*

Wir wählen zunächst die Zeilen-Adresse über die  $h$ -Leitung und speichern diese in den Zeilenadresspuffer indem wir die RAS-Leitung auf High setzen und wählen im nächsten Schritt die Spaltenadresse aus, indem wir die CAS-Leitung auf High und die RAS auf Low setzen.

### Lesezugriff

WE (writeenable) deaktivieren -> wir wollen lesen

Dann legen wir die Zeilenadresse über  $h$  an, indem wir RAS auf High setzen.

In diesem Moment liegt automatisch jedes Bit (also der Bitvektor) dieser Zeile am Leseverstärker an.

Nun können wir über die  $h$ -Leitung durch aktivieren der CAS-Leitung auswählen, welches Bit des Leseverstärkers wir wollen (es wird über den Spaltenmultiplexer ausgegeben).

Weil nun alle Kondensatoren dieser Zeile leer sind, müssen wir WE aktivieren und den Bitvektor zurückschreiben (Auffrischung).

### Optimierung bei Mehrfachzugriff

Will ich nun zum Beispiel Daten nacheinander aus derselben Zeile aber unterschiedlichen Spalten auslesen, dann muss ich die Zeilenadresse nicht ändern. Wir legen also nur neue Spaltenadressen an (Fast Page Mode, FPM, ca. 2x so schnell). Noch schneller funktioniert der Burst-Mode (ca. 16x), dabei wird auch die Spaltenadresse nicht mehr angelegt, sondern über Arithmetik automatisch hochgezählt. Damit wird im nachfolgenden Takt automatisch das nächste Bit (o. Wort) gelesen.

### Schreibzugriff

Beim Lesezugriff haben wir ja auch schon einen Schreibzugriff (Auffrischung).

Wir legen die Zeilenadresse an (RAS-Leitung aktivieren), womit wieder der gesamte Bitvektor der aktivierten Zeile im Leseverstärker anliegt. Danach wählen wir noch aus, welche Spalten beschrieben werden soll und legen im Spaltendemultiplexer das Bit  $d$  in die Zeile und gewählte Spalte eingefügt und beim Auffrischen eingeschrieben.

Auch hier sind selbige Optimierungen möglich.

### Zuverlässigkeit

Weil das Ganze sehr riskant klingt, gab es zwei wichtige Erweiterungen zur Erhöhung der Zuverlässigkeit:

**Kodierung** zur Fehlererkennung und (im Idealfall) Fehlerkorrektur. Dabei nehmen zusätzliche DRAM-Zellen redundante Informationen auf, um Fehler erkennen zu können.

Genauere Informationen folgen in Rechnernetze und Internettechnik im 4. Semester

**Speicherschutz**, wobei die CPU den Adressbus vor Zugriff prüft. Wenn zum Beispiel eine Anwendung wichtige Daten des Betriebssystems überschreiben will. Das nennt man Speicherschutz, also zusätzliche Logik, ob am Adressbus den entsprechenden Berechtigungen entspricht (welche im privilegierten Modus [z.B. vom Betriebssystem] vergeben werden).

Das wiederum vertiefen wir in Betriebssysteme, 2. Semester

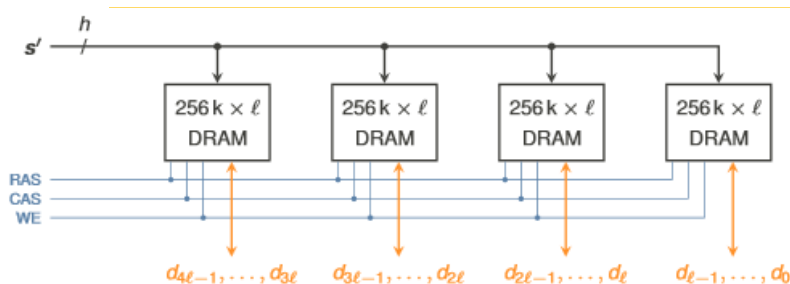
### Hauptspeicher aus DRAM-Bausteinen

Wir nehmen an, wir haben 4 Bausteine mit 9  $h$ -Steuerleitungen, also  $2^9 = 256k \times l$  Bit pro Baustein, wobei die Ansteuerung  $l$ -mal realisiert werden kann. Das heißt die obige Matrix wird  $l$ -mal nebeneinander angelegt, die eine Wortbreite (4- 8- 16- Bit) ergeben.

#### Variante 1: Erweiterung der Wortbreite

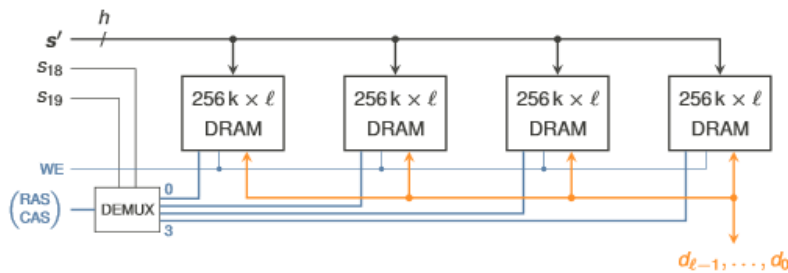
Wenn die obige Wortbreite nicht reicht, kann ich einfach mehrere Bausteine nebeneinanderstellen und die Steuerleitung, sowie alle RAS, CAS und WE-Leitungen parallelschalten. Die Datenleitungen hingegen werden einzeln und getrennt behandelt.

*Durch gemeinsame Adress- und Steuerleitungen, aber getrennte Datenleitungen erreichen wir größere Wörter*



### Variante 2: Erweiterung des Adressraums

Weil wir mit Variante 1 nur größere Wörter, aber keinen größeren Adressraum (d.h. wir haben immer noch nur 256k „Adressen“ und  $256k \times 4 \times l$  „Speicherplatz“) erreichen, müssen wir uns um einen anderen Weg suchen, um 1MB zu adressieren.



Hierbei haben wir immer noch eine gemeinsame Steuerleitung, aber zwei zusätzliche Steuerleitungen  $s_{18}$  und  $s_{19}$ , welche festlegen, mit welchem Baustein gesprochen werden soll.

Das heißt  $s_{18}$  und  $s_{19}$  entscheidet, an welchen Baustein die RAS und CAS Leitung angelegt wird, die Steuerleitung  $s$  wird trotzdem an alle weitergegeben, weil das ohne RAS und CAS nix außer Auffrischungen auslöst.

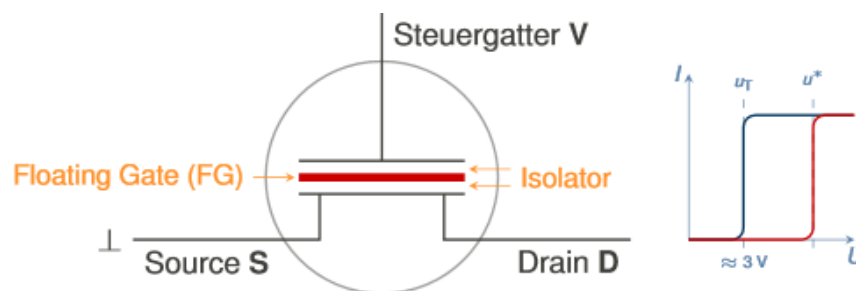
Der Datenbus wird gemeinsam verwendet.

Cooler Information zum Abschluss von DRAM:

Wenn wir DRAM kühlen/einfrieren, entstehen weniger Leckströme, dadurch können wir den DRAM vom Strom trennen und die Daten über mehrere Minuten erhalten.

### Flash-Speicher (Persistenter Speicher)

Grundidee: Informationen in isolierten Gattern speichern. Dazu lernen wir einen neuen Transistor kennen: den Feldeffekttransistor mit isoliertem Gatter (**FGMOS**).



Der Unterschied zu normalen Transistoren ist, dass bei der Herstellung ein Floating Gate und zwei Isolatoren zwischen S/D und V gelegt werden. Dieser Floating Gate kann bei hohen Spannungen ( $\sim 10V$ ) durch quantenmechanische Tunneleffekte Ladungsträger „einsperren“.

Ein normaler Transistor dient als „Schalter“, d.h. bei ca 3V ( $U_T$ ) leitet der Transistor von Source nach Drain. Davor nicht. Der **Feldeffekttransistor verhält sich gleich, wenn der Floating Gate nicht geladen** ist!

**Wenn** der **Floating Gate** jetzt **aber geladen** ist, dann **schirmt** er das **Steuergatter ab** und verschiebt den Schwellenpunkt  $U_T$  auf eine höhere (niedriger als 10V) Spannung  $U^*$ . Dazu kann man im Floating Gate (mindestens) ein Bit speichern, wobei mit mehr Bits eine unterschiedliche Verschiebung erreicht werden kann.



## Zustandsübergangstabelle

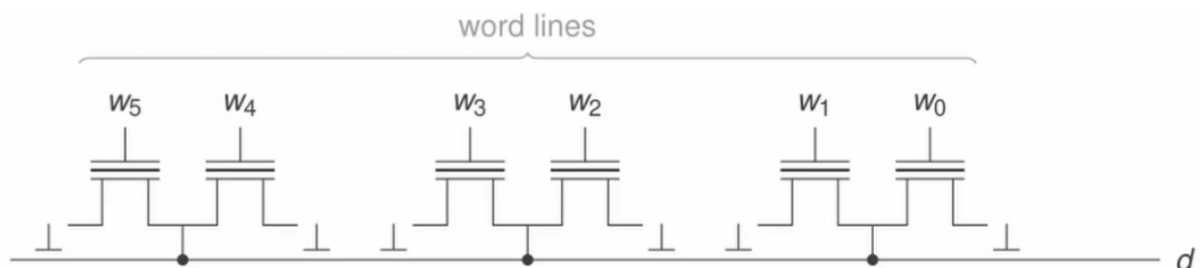
Steuergatter V	Drain D	FG-Ladung	Transistor	Zustand
<b>Lesezugriff</b>				
$u_T < 3.3\text{ V}$	$> \perp$	ungeladen	leitet	logisch 1
$u_T < 3.3\text{ V} < u^*$	$> \perp$	negativ geladen	sperrt	logisch 0
$> u^*$	$> \perp$	negativ geladen	leitet	logisch 0
<b>Schreib- und Löschzugriff</b>				
$> 10\text{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

## Vergleich NOR/NAND

NOR-Flash	NAND-Flash
<ul style="list-style-type: none"> <li>+ adressierbar wie RAM</li> <li>+ wahlfreies Lesen und Schreiben</li> <li>+ i. d. R. fehlerfrei</li> <li>– Schreiben nur langsam</li> <li>– hoher Energieverbrauch</li> <li>– geringe Speicherdichte</li> <li>– teuer (pro Bit)</li> <li>► geeignet für Programmcode</li> </ul>	<ul style="list-style-type: none"> <li>+ billig (pro Bit)</li> <li>+ hohe Speicherdichte</li> <li>+ schnell</li> <li>+ wahlfreies Lesen</li> <li>– Löschen nur blockweise</li> <li>– komplizierte Ansteuerung</li> <li>– Produktionsfehler nicht vernachlässigbar</li> <li>– Abnutzung</li> <li>► geeignet für Dateisysteme</li> </ul>

## NOR-Flash

Um die Flash-Bausteine nun adressieren zu können, kann man z.B. NOR-Flash verwenden. NOR, weil wenn mindestens an einem Wort eine Spannung anliegt.



Dabei gibt es eine Datenleitung **d**, wobei wir jeden einzelnen FGMOS über eine Wort-Leitung (w1, ... w5) ansprechen können.

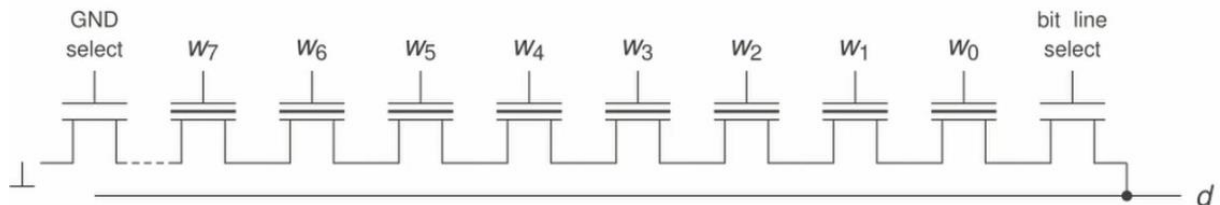
Jeder Feldtransistor hat bei NOR-Flash eine individuelle Erdung. Die Datenleitung geht auf 0, wenn an mindestens einer Wort-Leitung eine Spannung über dem Arbeitspunkt des FGMOS anliegt. Das heißt, ich lege 0V an alle Wort-Leitungen, die

mich nicht interessieren und **3,3V an alle die mich interessieren** und lese aus was an der Datenleitung anliegt. Damit komme ich an jedes Wort einzeln.

NOR-Flash lässt sich leicht ansteuern (zu DRAM ähnlich), baulich ist es aber sehr kompliziert und kostet viel Platz. Daher sehen wir heute öfter den:

## NAND-Flash

NAND, weil an allen Wörtern die Spannung über dem Arbeitspunkt anliegen muss.

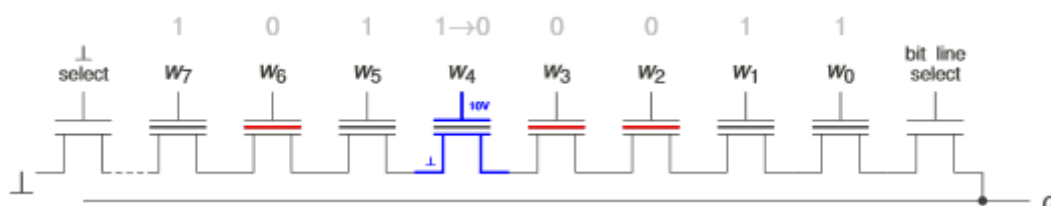


Die bit line geht auf 0, wenn an allen Wörtern eine Spannung über den jeweiligen Arbeitspunkten der FGMOS anliegt.

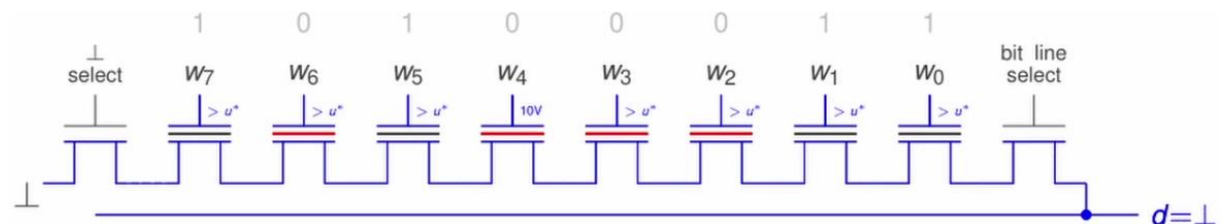
Das heißt, wir müssen beim Auslesen durchtasten, also bei allen was uns nicht interessieren eine Spannung über  $U^*$  anlegen und nur jene, **die uns interessieren eine Spannung zwischen  $U_T$  und  $U^*$** .

Dadurch kommen wir über einen sperrenden Transistor drüber, erreichen die uns interessierenden Wörter bis zum Datenbus.

## Schreiben



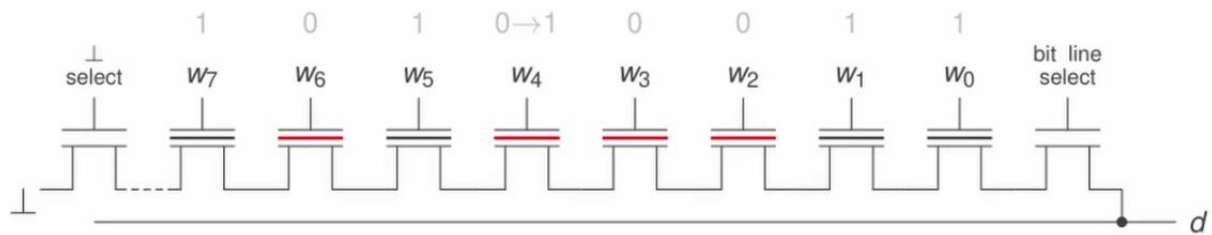
Das heißt wir wollen in w4 „0“ schreiben. Dazu müssen wir links und rechts von w4 geerdet sein, und die Steuerleitung muss auf über 10V liegen.



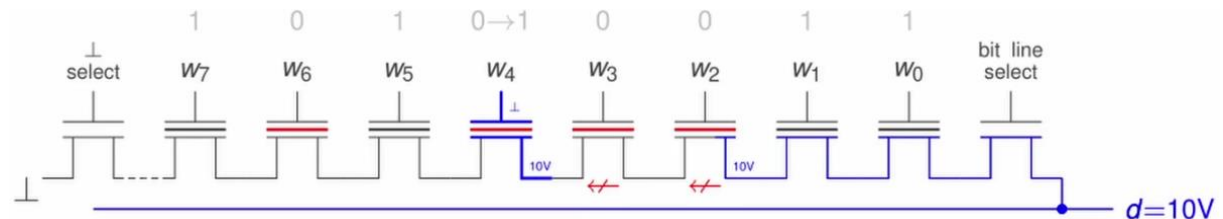
Dazu legen wir an die Datenleitung die Erdung an und an allen Wörtern außer w4 eine Spannung über  $u^*$ . Damit werden alle FGMOS außer w4 leitend und weil wir w4 10V anlegen, laden wir den FG darin und gehen auf 0.

## Löschen

Wollen wir w4 wieder löschen, also von 0 auf 1 stellen:



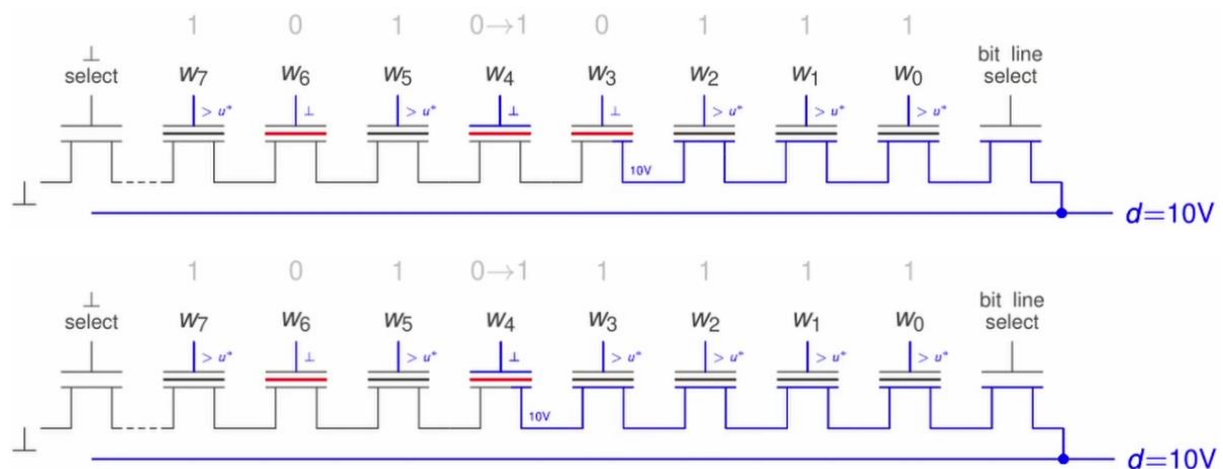
dann haben wir ein Problem:



Wir bräuchten oben die Erdung, aber von rechts 10V. Weil dazu alle FGMOS rechts des gewünschten Transistors leiten müssten, dies aber nie der Fall ist, haben wir ein Problem, wir bekommen keine 10V ran.

## Deshalb kann nur Blockweiße gelöscht werden!!

Dazu lassen wir schrittweise die 10V durch die Transistoren durch, indem wir jedes Wort Schritt für Schritt auf über  $u^*$  stelle.



*Aufgrund der Abnutzung sollte dieses Löschen so selten wie möglich getätigt werden.*

Weil ein Flash-Speicher schnell abgenutzt wird, können wir unsere Löschvorgänge (also jedes Mal wenn eine 1 geschrieben wird) nicht beliebig oft durchführen. Deshalb können wir auch nicht einzelne Transistoren löschen, sondern nur Blockweise. Damit nicht ein einzelner Block zu oft (öfter als andere Blöcke) gelöscht wird und damit schneller als die anderen abgenutzt werden, gibt es den Flash Translation Layer (FTL)

## Flash Translation Layer (FTL)

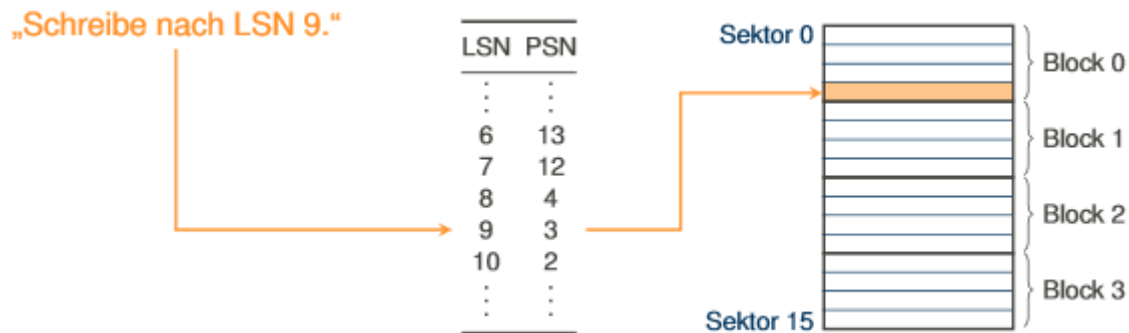
Dabei verbergen wir die Nachteile der NAND-Flash in einer Steuerschicht, indem wir die Löschoperationen gleichmäßig verteilen (**Wear Leveling**). Dafür wird die

logische (ansteuerbare) Adresse ( $L \cdot N$ ) in eine physische ( $P \cdot N$ ) Adresse (also die tatsächliche Adresse am Stick) übersetzt.

Außerdem können wir damit fehlerhafte Blöcke ausschließen aus unserem Flash-Speicher.

#### Variante 1: Sektor Mapping

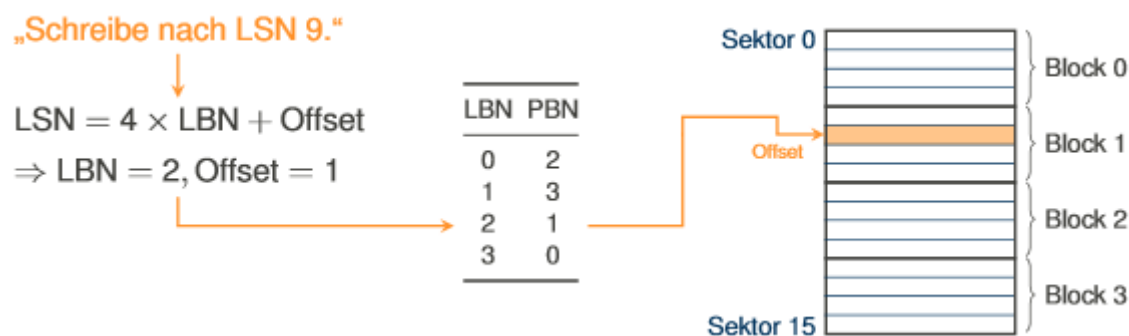
Blöcke sind vorgegeben (siehe oben, „jede Schnur“), die in Sektoren eingeteilt werden. Dabei wird einfach eine Tabelle jeder logischen Sektor Nummer zu einer Physikalischen Sektor Nummer gemappt:



Diese Tabelle wird laufend vom FTL aktualisiert. Es kann durchaus vorkommen, dass der FTL einen ganzen Block verschiebt, die Tabelle aktualisiert und den Block löscht, damit er neu zugewiesen werden kann.

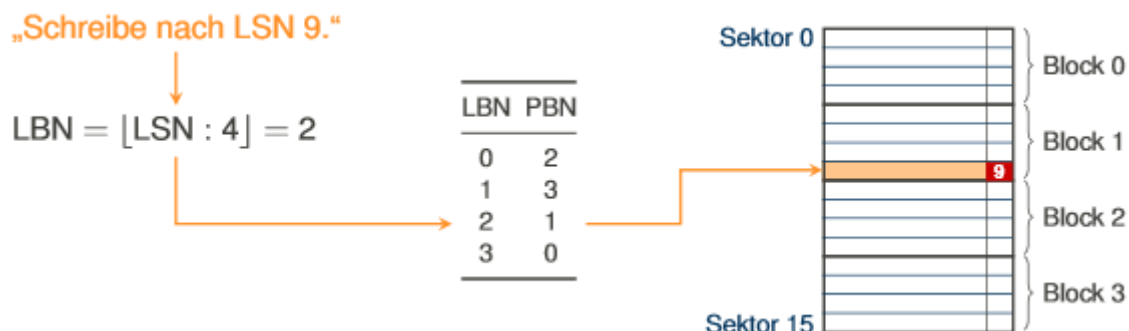
Das Problem dabei ist: Die Tabelle wird verdammt groß. Lösung:

#### Variante 2: Block-Mapping



Dabei darf logisch nur noch der Block angegeben werden, der Offset ergibt sich aus dem Rest bei Division durch die Anzahl an Blöcke. Der FTL bestimmt also nur noch, in welchen Block ein Schreibzugriff gehen wird, nicht in welchen Sektor.

#### Variante 3: Hybrides Mapping



Es werden weiterhin nur Blöcke verwaltet, wobei im Flash direkt besondere Felder freigehalten werden, um welchen logischen Sektor es sich handelt. Das bedeutet aber auch, dass ich nach dem bestimmen des Blocks alle Sektoren darin absuchen muss, bis ich den entsprechenden Logischen Sektor finde.

#### Probleme

Zum einen muss die Verwaltung selbst abgespeichert werden, also die Tabellen. Diese werden wir nicht genauer besprechen, weil es dazu unheimlich viele Konzepte gibt.

Für uns interessant ist aber eine Anforderung: **Die Konsistenz der Verwaltungsdaten bei Stromausfall.**

Wir müssen sicherstellen, dass die Verwaltungsinformation niemals verloren geht, sonst ist der Stick im Arsch. Stell dir vor der Nutzer zieht den Stick ab, während die Tabelle aktualisiert wird!

Daher wird die Verwaltungsinformation immer so kodiert, dass sie bei Problemen „wiederhergestellt“ wird (über Logik).

Auch wichtig für unsere Sicherheit: Das Wear-Leveling des FTL bewirkt, dass wir keinen Einfluss darauf haben, wann Daten wirklich gelöscht werden (sie werden ja nur neu gemappt, weil sie nur gelöscht werden dürfen, wenn wirklich keine andere Möglichkeit mehr besteht). Das bedeutet auch, wenn wir NAND-Flash mit 0en überschreiben wollen, dann löschen wir die Daten nicht wirklich, wir markieren sie nur als gelöscht!

## Leistung

Der Hauptspeicher wird jedes Jahr nur um ca. 7% schneller, die CPU aber um 50%. Daher entsteht eine Geschwindigkeitslücke, die wir überbrücken müssen!

## Definition

Allgemeine Definition aus der Physik:

---

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

---

### Wie definieren wir nun die Arbeit in der Informatik?

In der Regel durch die Rechenoperationen, die wiederum eingeteilt werden kann:

- ⌘ Instruktionen allgemein: **MIPS** (million instructions per second)  
Wir zählen, wieviele Befehle ein Prozessor ausführen kann.
- ⌘ Integer-Operationen: **MOPS** (million operations per second)  
Weil Befehle unterschiedlich viel erreichen können (Instruktion auf 4-Bit braucht weniger Arbeit als auf 64-Bit, Verschiebung/ALU-Operation?). Daher gibt's MOPS, die ALU-Operationen zählt
- ⌘ Gleitkomma-Operationen: **FLOPS** (floating point operations)  
I.d.R. sind die Flops kleiner als die Mops, weil man mehrere Instruktionen braucht.

*Umrechnung der Einheit ist nur bei definierter Mikroarchitektur möglich.*

## Wie definieren wir die Leistung im Verhältnis zur Zeit?

Messen wir die Leistung, wie sie in der Praxis beobachtbar ist? Oder doch das theoretische Maximum bei Idealfall?

- ☞ **Maximale Leistung** (peak performance)
  - Theoretisches Maximum, unter Idealbedingung
- ☞ **Durchschnittliche Leistung** (average performance)
  - Erwartungswert bei typischen Aufgaben
- ☞ **„Nachhaltige“ Leistung** (sustained performance)
  - Unterschranke für die über den gesamten Lebenszyklus versprochene Leistung eines Gesamtsystems oder von Komponenten mit Verschleißeffekt (Flash)

## Leistungsverhältnis

Wir wissen nun, wie Leistung definiert ist und wie wir sie messen, aber um verschiedene Leistungswerte mit alternativen Systemen vergleichen zu können, müssen wir das Leistungsverhältnis messen.

Bei konstanter Arbeit

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

### Beispiele:

Smartphone von vor zwei Jahren mit Aktuellem vergleichen, wobei wir auf beidem dieselbe Arbeit verrichten:

Referenzbedingung:	Smartphone von 2015
Testbedingung:	Smartphone von heute
$S = 2$ bedeutet, das neuere Smartphone ist <u>doppelt so leistungsfähig.</u>	

Auch auf konstanter Architektur kann man verschiedene Arbeiten vergleichen, indem wir zwei Sortieralgorithmen auf einer alten Intel 4040 mit denselben Daten vergleichen.

Referenzbedingung:	Bubble-Sort-Algorithmus auf Intel 4040
Testbedingung:	Quicksort-Algorithmus auf Intel 4040
$S = 4$ bedeutet, Quicksort ist (für die gewählten Daten, insb. Datenmenge) viermal schneller <b>Grund: weniger Instruktionen</b>	

## Messung der Ausführungszeit

Es werden Hardwarezähler zur messung der CPU-Zyklen verwendet, auf die wir in ARM über die Mnemonics MRC und MCR zugreifen können (kein Vorlesungsstoff).

Dann können wir die Ausführungszeit über Zykluszeit  $\Delta t$  oder Taktfrequenz  $1/\Delta t$  berechnen. Aber wir müssen unterscheiden zwischen Gesamtzeit (also ab Start bis Messpunkt, inklusive Interrupts, Betriebssystem) und der CPU-Zeit (also ohne wartezeit auf E/A-Geräte).

Einflussfaktoren auf die Leistung eines Programms

Implementierung

Darunter fallen Algorithmen, Programmiersprachen, Compiler und die entsprechenden Optimierungsstufen.

### Ausführungsumgebung

Wesentliches Merkmal ist die Befehlssatzarchitektur, Mikroarchitektur (wie ist ALU angebunden, wie viele Befehle braucht man um Daten an ALU anzuführen, ...) Betriebssystem (Interrupts) und sonstige Hardware.

### Methodische Schwierigkeiten der Leistungsmessung

#### Implementierung

Wir nehmen eine definierte Ausführungsumgebung (Hardware und Systemsoftware) an. Dann bekommen wir folgende Schwierigkeiten beim Vergleich:

- ⌘ Granulare Messung einzelner Programmteile
  - Messfehler, weil wir zusätzliche Befehle zur Leistungsmessung einführen müssen, welche den Programmablauf beeinflussen
- ⌘ Grobe Messung ganzer Programmabläufe
  - Umgebungseinflüsse (Unterbrechungsanforderung durch Mausbewegung)

### Ausführungsumgebung

- ⌘ Wahl vergleichbarer Aufgaben und Implementierungen
  - z.B. schaut Quicksort auf ARM anders aus als auf x86
- ⌘ Wahl der Testdaten bei Datenabhängigkeit
  - Also z.B. vorsortierter Array bei Bubblesort bringt bessere Ergebnisse

---

### Unschärfe bei Virtualisierungstechniken (keine Hardwarezähler)

---

### Benchmarks

Dienen zur Standardisierung von Leistungsmessung

Wobei hier oft **synthetische Benchmarks** durchgeführt werden, die Instruktionen pro Sekunde messen, wobei diese meist Operationen durchführen, die man im normalen Alltag nicht verwendet, aber gute Resultate liefern.

Bei **Kernel-Benchmarks** werden ausgewählte Algorithmen, die sich für Benchmarking etabliert haben, verwendet (vor allem wichtig für wissenschaftliche Arbeiten)

**Mikrobenchmarks** dienen zum Messen einzelner Komponenten (Grafik, Speicheranbindung) und sind darauf spezialisiert, sind also Benchmarks für alles was nicht die ALU betrifft (dort sind FLOPS und MOPS).

Benchmark-Suiten enthalten verschiedene, gut portierbare Programme aus unterschiedlichen Anwendungsgebieten. Zum Beispiel SPEC (2006).

### Schwierigkeit in der Realität

Das Problem in der Realität ist die Tatsache, dass wenn wir ein System über mathematische Mittel auf eine Kennzahl projizieren, die zum Leistungsverhältnis dient, können zukünftige Hardwarehersteller diese Berechnungen dazu „missbrauchen“ um mit ihren Systemen die möglichst beste Kennzahl zu erreichen. Damit wird die Kennzahl immer weniger aussagekräftig.



## Amdahlsches Gesetz

Angenommen, wir können einen ganz bestimmten Anteil eines Programmes um einen Faktor  $c$  beschleunigen. Dann können wir folgende Formel verwenden:

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

Braucht zum Beispiel ein Programm zu 20% seiner Ausführungszeit zur Division, und ein neuer Koprozessor beschleunigt Divisionen um den Faktor 10, dann hätten wir nach Einfügen der Formel eine rechnerische Beschleunigung von 1.2195.

### Das Amdahlsche Gesetz

Nehmen Sie an, dass 40% der Ausführungszeit eines gegebenen Programms Speicheroperationen sind. Durch einen neuartigen L1 Cache werden 75% der Speicheroperation um den Faktor 5 beschleunigt. Ein weiterer neuer L2 Cache beschleunigt zusätzlich drei Fünftel der übrigen 25% Speicheroperationen um den Faktor 2. Was ist die gesamte Beschleunigung, wenn beide Arten von Caches eingesetzt werden?

1. Massnahme:  $c_1 = 5$ ;  $\alpha_1 = .75 \cdot .4 \Rightarrow S_1 = 1/((1 - 0.3) + (0.3/5)) \approx 1.3158$
2. Massnahme:  $c_2 = 2$ ;  $\alpha_2 = .25 \cdot .4 \cdot .6 \Rightarrow S_2 = 1/((1 - 0.06) + (0.06/2)) \approx 1.0309$

Beide Massnahmen:  ~~$S_{tot} = S_1 \cdot S_2 \approx 1.3158 \cdot 1.0309 \approx 1.3565$~~

Generell:  $S_{tot} = 1/(\alpha_1/c_1 + \alpha_2/c_2 + (1 - \alpha_1 - \alpha_2))$

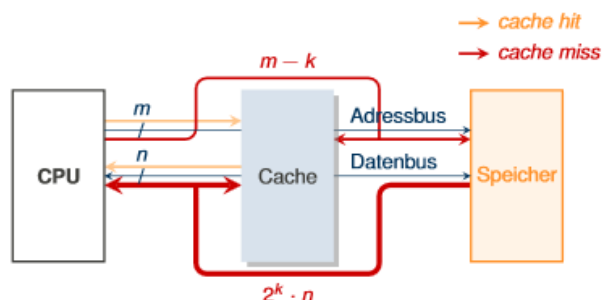
$S_{tot} = 1/((.3/5) + (.06/2) + (1 - .3 - .06)) \approx 1.3699$

## Optimierung des Speicherzugriffs durch Caches

### Der Cache

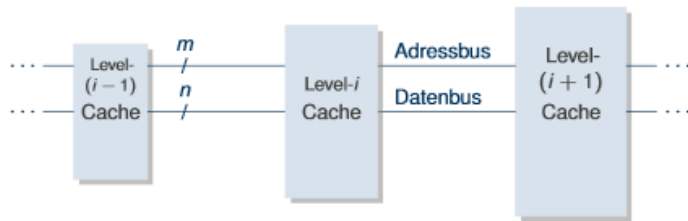
Der Cache ist ein kleiner, schneller, mit SRAM realisierter Pufferspeicher, dessen Aufgabe es ist „vorherzusehen“, welche Daten als nächstes abgefragt werden und die Daten aus dem Speicher vorlädt, um sie dem Prozessor schneller zur Verfügung zu stellen.

Wenn diese „Vorhersage“ stimmt, nennen wir dies einen **Cache hit**. Sonst ist es ein **Cache miss**, bei dem dann ein ganzer Block aus  $2^k \cdot n$  Wörtern aus dem Speicher geladen werden, wobei die CPU nur den Teil nimmt, der sie interessiert, der Cache aber den Rest zwischenspeichert, um einen weiteren möglichen Cache miss im nächsten Zyklus zu verhindern. (Liegt ja auch nahe, dass Daten die nebeneinander liegen eher zusammengehören, als Daten die weit auseinander sind).





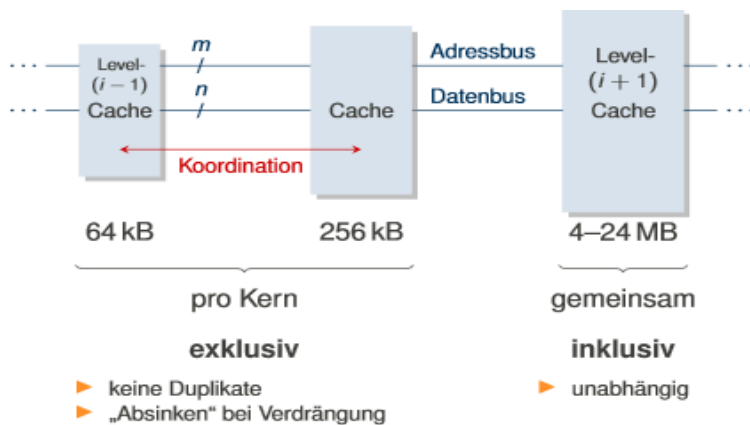
In der Realität findet dieses Prinzip aber mit kaskadenförmigen Caches Anwendung:



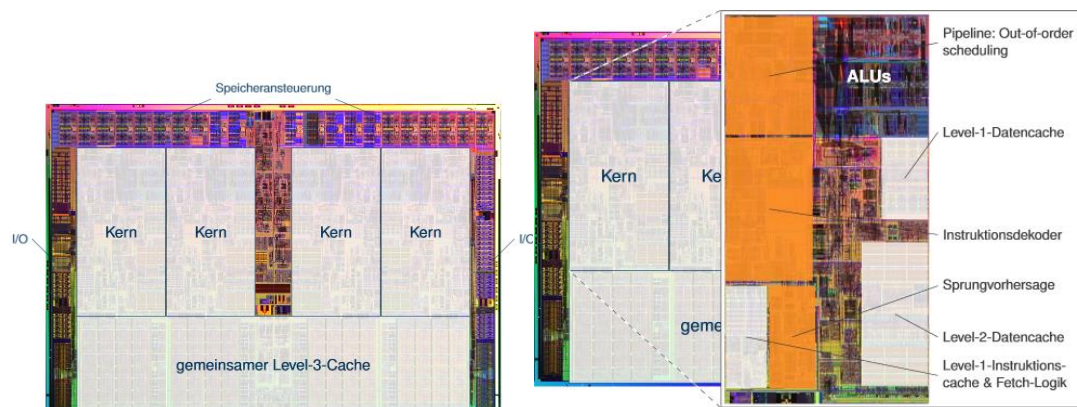
**Die Hierarchie selbst ist** für die CPU unbekannt. Das heißt, der Programmierer kümmert sich nicht, ob die Daten unter einer Adresse aus dem Cache oder aus dem Speicher geladen werden – dies erfolgt **transparent**.

### Die Cache-Hierarchie

Ein sehr beliebtes Beispiel ist die Intel Nehalem-Mikroarchitektur (2010):



Das bedeutet, die ersten zwei Level sind gemeinsam koordiniert innerhalb eines Kerns. Damit kann der Level 2 Cache das aufnehmen, was im Level 1 verdrängt werden muss.



### Ziel / Lokalitätsprinzip

#### Zeitliche Lokalität

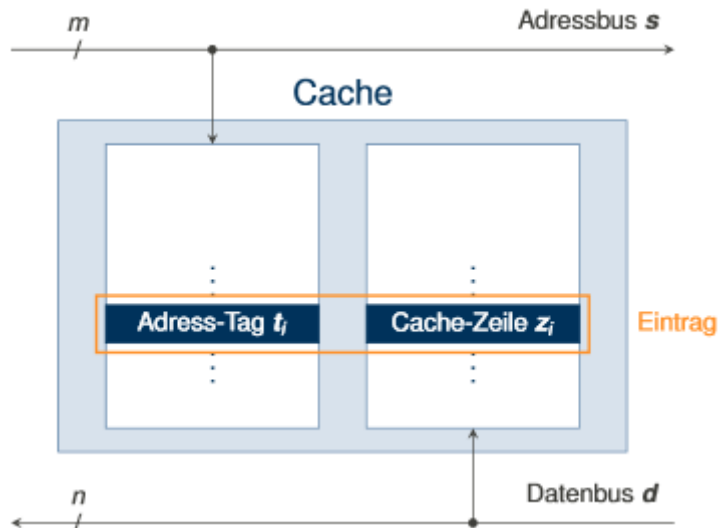
Es liegt nahe, dass Speicheradressen wiederholt verwendet werden (z.B. bei Schleifen), weshalb verwendete Adressen und Inhalte gepuffert werden

### Räumliche Lokalität

Ebenfalls gibt es eine hohe Wahrscheinlichkeit, dass benachbarte Elemente zusammengehören (z.B. Arrays), so wie es beim Cache miss passiert. Das heißt benachbarte Speicherworte werden gelesen und gepuffert.

Ebenfalls der DRAM-Burst Mode kann dank Cache besonders gut verwendet werden.

### Aufbau eines Caches



Dabei besteht ein einzelner Eintrag aus einem Adress-Tag und einer Cache-Zeile. Grundsätzlich sollte dieser Aufbau selbsterklärend sein, die Cache-Zeile ist der Inhalt über den Datenbus, der mittels dem Adress-Tag der Adresse im Speicher entspricht.

### Aber wie wird das Ganze verwaltet?

Grundsätzlich gibt es zwei Reinformen:

#### Vollassoziativ

Dabei kann jeder Block des Hauptspeichers in einer beliebigen Cache-Zeile abgespeichert werden. Das heißt, die Cache-Zeile muss so groß sein wie der Block, folglich können wir  $k$ -Bits zum Adressieren der Daten innerhalb des Blocks abziehen und der Tag muss  $m-k$  Bits lang sein:

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

Diese Form ist natürlich sehr Flexibel, weil alle Zeilen genutzt werden können. Dafür müssen aber alle Tags beim Lesen durchsucht werden (das ist sehr aufwändig!)

Auch muss geklärt werden, wie Daten aus dem Cache gelöscht werden

(welche Kriterien), das nennt man dann Verdrängungsstrategien.

#### Direkt abgebildet

Die  $j$  niederwertigsten Adressbits definieren eine Cache-Zeile für jeden Block des Hauptspeichers.

Das heißt, jeder Block des Hauptspeichers kann nur in einer Zeile stehen und zwar in der, wo die  $j$  niederwertigsten Bits übereinstimmen.

Damit lassen sich in einer Zeile  $2^j$  (höherwertige Bits) unterschiedliche Blöcke speichern, aber eben immer nur einer!

$$j > k \Rightarrow t \in \{0, 1\}^{(m-j)}$$

Damit sind die Tags kleiner.

Vorteil ist, das Lesen ist unheimlich schnell, weil nur ein einziger Vergleich

(und eine Dekodierung der Adresse) notwendig ist!

Mischformen ermöglichen z.B. dass jeder Block des Hauptspeichers in einer von zwei definierten Cache-Zeilen abgespeichert werden kann. Meist ist eine Mischform besser als die Reinformen.

## Cache-Kohärenz

Die Kohärenz beschäftigt sich mit dem Problem, wie geschriebene Daten aus der CPU in den Speicher auch in den Cache geschrieben werden, d.h.: wie stellen wir sicher, dass Cache und Speicher dieselben Daten haben?

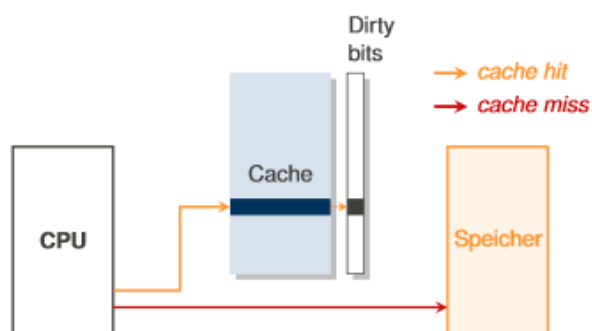
### Write through-Methode

Dabei werden im Cache vorhandene Daten beim Schreibzugriff auf den Speicher einfach direkt auch in der entsprechenden Cache-Zeile geschrieben. Das macht unser System zwar **langsamer** (zwei Schreibzugriffe statt einem), dafür **schließen** wir eine mögliche **Inkonsistenz komplett aus**.

### Write back-Methode

Dabei wird ein Wert aus dem Cache auch nur mehr in den Cache zurückgeschrieben. Der Speicher wird bei einem cache hit nicht aktualisiert, dafür brauchen wir im Cache zusätzliche Verwaltungsinformation, sogenannte Dirty Bits, die angeben, ob eine Cache-Zeile bei deren Verdrängung aus dem Cache in den Speicher geschrieben werden muss.

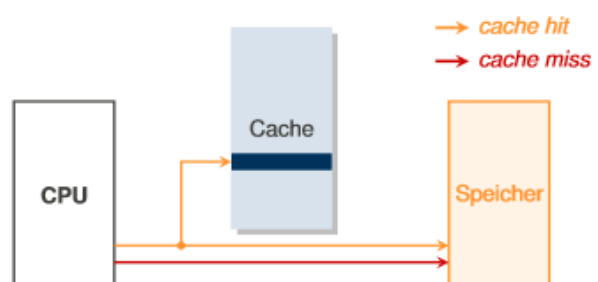
Dieses Zurückschreiben kann der Cache-Controller selber machen, wir haben also **kaum Performance-Einbußen**, müssen aber **mehr Verwaltungsaufwand** rechtfertigen und **haben eine Inkonsistenz** zwischen Speicher und Cache.



### Write allocation-Methode

Bei einem Cache-Hit entspricht diese Methode der write back-Methode.

Bei einem cache-miss wird jeder Schreibzugriff zuerst im Cache aktualisiert (dort also gepuffert) und bei deren Verdrängung in den Speicher geschrieben. Wichtig hierbei ist, dass durch einen Schreibzugriff eine Verdrängung im Cache herbeigeführt werden kann. Ist aber dafür sinnvoll



z.B.: für Laufvariablen, die gar nie in den Speicher müssen.

## Leistungssteigerung durch Cache

Wie können wir nun messen, wie gut der Cache die Leistung steigert?

### Trefferrate

$$h = \frac{\text{Anzahl } \text{cache hits}}{\text{Anzahl } \text{cache hits} + \text{Anzahl } \text{cache misses}}$$

Damit können wir dann (mit folgenden Parametern:) die mittlere Zugriffszeit berechnen:

c = Zugriffszeit auf den Cache (z.B. 5ns)

r = Zugriffszeit auf den Hauptspeicher (z.B. 50ns)

### mittlere Speicherzugriffszeit

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

Mit modernen Caches lassen sich Trefferraten von bis zu 90% realisieren, wodurch eine Leistungssteigerung um den Faktor 10 möglich sind (Bei unserem Beispiel).

Der Nachteil bei dieser Berechnung, das exakte Zeitverhalten kann schlecht vorhergesehen werden. Interrupts führen zB zur Verdrängung und daraus folgt eine längere Speicherzugriffszeit.

Wichtig beim Vergleich zweier Caches, dass in diese Trefferrate oben, alle Faktoren des Caches eingeflossen sind. Das heißt unterschiedliche Größen, Vollassoziativ oder direkt abgebildet, spielt für uns keine Rolle, nur die Trefferrate und die Zugriffszeit.

## Optimierte Programmierung

Mit dem Wissen, dass dort ein Cache ist und wie er möglicherweise arbeitet, können wir unseren Code so optimieren, dass der Cache zur Gänze ausgenutzt werden kann.

Das heißt wir wollen **cache misses vermeiden** und diese analysieren. Ein Cache miss kann folgende Ursachen haben:

- ⌘ Erstbelegung nach Programmstart
- ⌘ Verdrängung benötigter Zeilen mangels Kapazität
- ⌘ Verdrängung benötigter Zeilen durch Konflikte

Wir nennen diese Ursachen auch die 3 C's: **Compulsory, Capacity, Conflict**

Wir können den 3 C's entgegenwirken, indem wir:

- ☞ Mit **expliziten Prefetch-Anweisungen** Daten rechtzeitig holen  
Lässt sich manuell im Assemblerprogramm oder durch den Compiler erledigen
- ☞ **Vermeidung von Konflikten**  
Daten so strukturieren, dass sie auf eine Cache-Zeile passen. Die Dimension der Felder in Zweierpotenzen wählen (weil ja auch eine Cache-Zeile eine Zweierpotenz ist), wobei der restliche Platz mit Füllwörter belegt wird.
- ☞ **Lokalität** beim Zugriff auf Daten **erhöhen**

Vorher	Nachher
<pre> 1   char *name[1000]; 2   int  matrikelnr[1000]; </pre>	<pre> 3   struct student { 4       char *name; 5       int  matrikelnr; 6   } hoerer[1024]; </pre>

Rechts ist wesentlich Cache-freundlicher, weil wir die Daten mit einer räumlichen Lokalität versehen (Es ist wahrscheinlicher, dass nach abfragen der Matrikelnummer der Name ausgelesen werden soll)

Vorher	Nachher
<pre> 7   for (j=0; j&lt;100; j=j+1) 8       for (i=0; i&lt;5000; i=i+1) 9           x[i][j] = 2*x[i][j]; </pre>	<pre> 10   for (i=0; i&lt;5000; i=i+1) 11       for (j=0; j&lt;100; j=j+1) 12           x[i][j] = 2*x[i][j]; </pre>

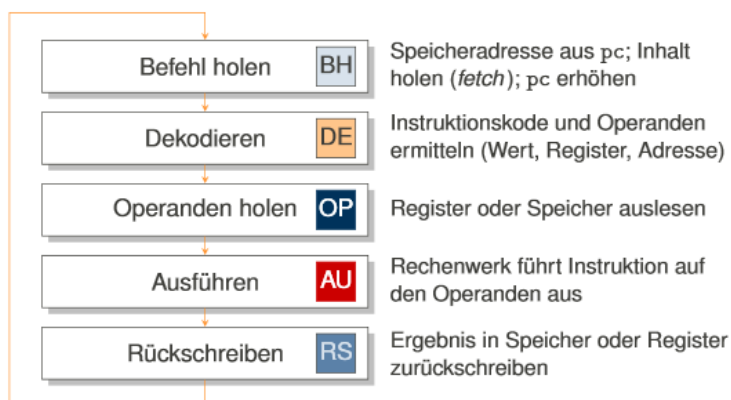
Damit nutzen wir die zeitliche Lokalität, wobei die innere Schleife den letzten Index steuert, damit die Daten nicht wild im Speicher gesammelt werden müssen.

## Optimierung der CPU-Nutzung durch Pipelines

Die Idee ist es, Sprunganweisungen vorauszusagen, um unser Programm zu beschleunigen.

### Maschinenbefehlszyklus

Dazu sehen wir uns zuerst mal einen Befehlszyklus einer Maschine an. Bisher haben wir mit dem primitiven Fetch, Dekode, Execute beschäftigt, aber es gibt auch andere Varianten:



Damit lässt sich die die Maschine sequentiell oder (besser) überlappend ausführen.

## Sequenzielle Abarbeitung der Teilschritte



## Überlappendes Abarbeiten



→ Pro Taktzyklus wird ein Maschinenbefehl abgeschlossen.

## Leistungssteigerung (theoretisch)

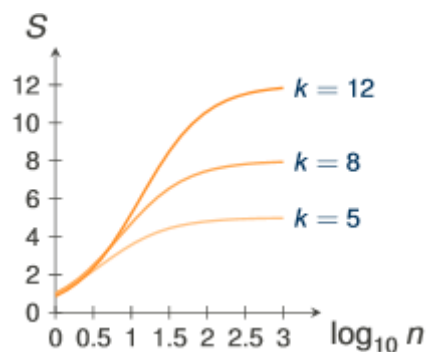
Was lässt sich nun mit diesem Überlappenden Idealfall theoretisch verbessern?

Wir nehmen an, wir haben eine ideale  $k$ -stufige Pipeline und ein Programm mit  $n$  Befehlen. Die Zykluszeit beträgt  $\Delta t$ , d.h. eine Taktfrequenz von  $1/\Delta t$

Berechnung der Beschleunigung:

$$S(k) = \frac{\text{Ausführungszeit ohne Pipeline}}{\text{Ausführungszeit mit Pipeline}} = \frac{n \cdot k \cdot \Delta t}{[n + (k - 1)] \cdot \Delta t}$$

Dabei ist die Beschleunigung Logarithmisch:



## Gründe für Abweichungen in der Praxis (hazards)

## Strukturkonflikte (structural hazard)

Die Hardware unterstützt die Befehlskombination nicht, zB. wenn bestimmte ALU-Operationen nicht mit Speicheroperationen gleichzeitig ausgeführt werden können.

**Lösung:** Umordnen (passiert durch Compiler), sonst NOPs (Leertakte)

## Datenkonflikte (data hazard)

Dabei ist einfach ein Operand nicht verfügbar/berechnet.

**Lösung:** dezidierte Forward-Logik, sonst wie bei Strukturkonflikte

## Steuerkonflikte (control hazard)

Geholter Befehl ist nicht der benötigte (wenn z.B. bedingte Sprünge/Verzweigung)

**Lösung:** Sprungvorhersage oder einfach Anhalten

### Operationen mit mehreren Zyklen

Division, Multitasking, Gleitkommaoperationen, ... Wenn also mehrere Zyklen benötigt werden.

**Lösung:** separate Gleitkomma-Pipeline, sonst Geduld

### Optimierte Programmierung

Wollen wir Beispielsweise simple Addition ausführen:

$$a = b + c$$

$$d = e - f$$

Langsamer Code	Schnellerer Code
LDR r2, b	LDR r2, b
LDR r3, c	LDR r3, c
ADD r1, r2, r3	LDR r5, e
STR r1, a	ADD r1, r2, r3
LDR r5, e	LDR r6, f
LDR r6, f	STR r1, a
SUB r4, r5, r6	SUB r4, r5, r6
STR r4, d	STR r4, d