

# Technische Informatik 2

## Rechnerarchitektur

### Eigenes Skript

Studiengang Angewandte Informatik

Duale Hochschule Baden-Württemberg Karlsruhe

von

Andre Meyering

Kurs: TINF16B2  
Dozent: Prof. Dr. Jürgen Röthig  
Semester: 3. Semester (09.10.2017 - 12.12.2017)  
*letzte Änderung:* 20. Dezember 2017

Dies ist das eigene Skript für „Rechnerarchitektur“ bei Herrn Prof. Dr. Jürgen Röthig für das 3. Semester im Jahr 2017. Es enthält fast alles, was im Unterricht an die Tafel geschrieben oder besprochen wurde. Die  $\LaTeX$ -Dateien sollten sich im gleichen Share befinden, in dem du diese PDF-Datei gefunden hast.

Bei Fragen, Fehlern oder Ergänzungen – oder sollten die  $\LaTeX$ -Dateien fehlen – wende dich bitte an [dhbw@andremeyering.de](mailto:dhbw@andremeyering.de). Ich hoffe, diese PDF hilft dir beim Lernen.

## Inhaltsverzeichnis

<b>1 Vorwort</b>	<b>1</b>
<b>2 Einstieg in die Vorlesung</b>	<b>2</b>
2.1 Rechnerarchitektur (Vorlesung)	2
2.2 Übersicht	2
<b>3 Rechner</b>	<b>3</b>
3.1 Geschichte	3
3.2 Fundamentalarchitektur	4
<b>4 Rechenwerk / Rechnen in Hardware</b>	<b>8</b>
4.1 Schaltnetzsyntax – Wiederholung / Grundlegendes	8
4.2 Halbaddierer	9
4.3 Volladdierer	9
4.4 Paralleladdierer (4-Bit-Ripple-Carry-Paralleladdierer RC-PA)	10
4.5 Paralleladdierer (4-Bit-Carry-Look-Ahead-Paralleladdierer CLA-PA)	11
4.6 Serielladdierer	14
4.7 Subtraktion	16
4.8 Multiplikation	17
4.9 Division	21
4.10 CPUs	21
<b>5 Speicher in Computern</b>	<b>23</b>
5.1 Speicherhierarchie	23
5.2 Sekundärspeicher	23
5.3 CPU-Cache	23
5.4 Hauptspeicher-Organisation	36
5.5 Matrixförmige Speicherorganisation	38
<b>6 Wiederholungsklausur 2016</b>	<b>41</b>
6.1 Aufgabe 1 - 8 Punkte (6/2)	41
6.2 Aufgabe 2 - 10 Punkte (2/4/4)	41
6.3 Aufgabe 3 - 12 Punkte (6/6)	41
6.4 Aufgabe 4 - 20 Punkte (4/6/6/4)	42
<b>7 Abkürzungsverzeichnis</b>	<b>43</b>

<b>Listingsverzeichnis</b> . . . . .	<b>46</b>
<b>Stichwortverzeichnis</b> . . . . .	<b>47</b>

## 1 Vorwort

Herr Röthig schreibt alles, was für seine Klausuren von Bedeutung ist, an die Tafel. Es ist daher nur zu empfehlen, alles mitzuschreiben, da er kein Skript besitzt und auch keinen Foliensatz. Der Unterricht im Vergleich zu anderen Dozenten unterscheidet sich darin, dass während der Klausur keine Hilfsmittel verwendet werden dürfen. Dafür besteht die Klausur zu 90% nur aus Abfrageaufgaben.

Dieses Skript enthält *alles*, was Herr Röthig 2017 an Wissen voraussetzt. Auf den letzten Seiten dieses Skripts findet sich zusätzlich noch eine Übungsklausur. Die Klausuren unterscheiden sich jedes Jahr nur um einige wenige Aufgaben. Ist man zwei, drei Übungsklausuren durchgegangen, so ist die Klausur einfach zu bestehen.

Zusammen mit meinem Kurs TINF16B2 haben wir dieses Skript ausgedruckt und korrigiert. Inhaltliche Fehler sollten daher (fast) keine mehr enthalten sein.

Das aktuellste Skript findest du unter:

[https://gitea.ameyering.de/DHBW\\_AI\\_16/Rechnerarchitektur\\_Roethig](https://gitea.ameyering.de/DHBW_AI_16/Rechnerarchitektur_Roethig)

Ich wünsche dir viel Erfolg bei Herrn Röthig im Fach Rechnerarchitektur (Technische Informatik 2). Solltest du diese Skript erweitern wollen, so kannst du dich an [dhbw@andremeyering.de](mailto:dhbw@andremeyering.de) wenden.

## 2 Einstieg in die Vorlesung

Dozent: Prof. Dr. Jürgen Röthig  
Modul: Technische Informatik II  
Fach: Rechnerarchitektur

### 2.1 Rechnerarchitektur (Vorlesung)

- 36h/33h: 4h pro Woche
- Klausur: 21.12.2017 | 60min (ohne Hilfsmittel, Verrechnung mit Betriebssysteme)
- kein Skript, kein Foliensatz

### 2.2 Übersicht

1. Einführung, Begriffsbildung, Historie, Fundamentalarchitektur
2. Rechenwerke
3. Speicherwerk: Hauptspeicherorganisation  
Speicher: Cache, nicht-flüchtige Speichertechnologie
4. ausgewählte Kapitel aus Steuer-/Ein- und Ausgabewerke sind in 2 + 3 enthalten.

## 3 Rechner

Hilfsmittel zum Durchführen von „Rechnungen“.

### Rechner

- schneller
- fehlerfreier
- besseres Speichervermögen

### Rechenmaschine

- Abakus (mechanisch, digital)
- Rechenschieber (mechanisch, analog)

### Arbeitsweise

Man unterscheidet zwischen „mechanisch vs elektrisch“ und „digital vs analog“. Moderne „Rechner“ (PC & Co.) arbeiten elektrisch und digital. Dem gegenüber stehen elektrische Analogrechner, die um die 1920er genutzt wurden.

## 3.1 Geschichte

### 3.1.1 Elektrischer Digitalrechner

#### ZUSE Z1, Z2 (ab ~1940)

Relais als zentrale Bauteile (elektromagnetischer Schalter mit Elektromagnet)

- ⊕ Automatismus möglich
- ⊖ langsame Geschwindigkeit
- ⊖ großer Platzverbrauch
- ⊖ Geräusche beim Schalten
- ⊖ hoher Energieverbrauch beim Schalten
- ⊖ großer Verschleiß

#### Electronic Numerical Integrator and Computer (**ENIAC**) (~1945)

Die **ENIAC** besitzt als zentrales Bauteil eine Elektronenröhre. Eine Elektronenröhre ist ein eigentlich analog arbeitender Verstärker, wird hier aber als digitaler Schalter genutzt. Die Funktionsweise wird in **Abbildung 3.1** dargestellt, wobei die Kathode negativ und die Anode positiv geladen sind.

- ⊕ (sehr) hohe Geschwindigkeit
- ⊖ großer Platzverbrauch
- ⊖ ständiges Summen bei 50Hz oft möglich und hörbar
- ⊖ hoher, ständiger Energieverbrauch
- ⊖ großer Verschleiß

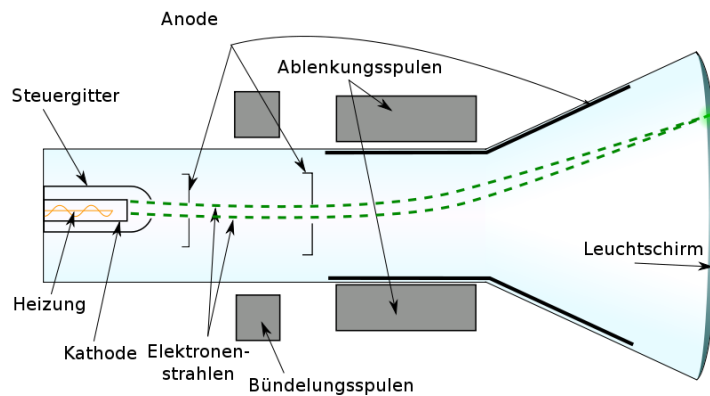


Abbildung 3.1: Funktionsweise Kathodenstrahlröhre [Quelle: [Wikipedia](#)]

**Moderne Rechner** Moderne transistorisierte Digitalrechner (z. B. [UNIVAC](#) ab Ende der 1950er).

- Transistor als zentrales Bauteil. Ein Transistor ist ein analog arbeitender Verstärker, wird hier aber als digital arbeitender Schalter genutzt.
- ⊕ sehr hohe Geschwindigkeit
- ⊕ sehr geringer Platzverbrauch
- ⊕ keine Geräuscentwicklung (außer Lüfter)
- ⊕ sehr niedriger Energieverbrauch
- ⊕ geringer Verschleiß

## 3.2 Fundamentalarchitektur

### 3.2.1 von-Neumann-Architektur

In [Abbildung 3.2](#) wird die von-Neumann-Architektur vereinfacht dargestellt. Diese besteht aus:

**Zentraleinheit (CPU)** Die CPU besteht aus:

**Rechenwerk** Rechnen mit Zahlen und logischen Werten

**Steuerwerk** Zuständig für das Steuern und Koordinieren aller anderen Komponenten  
⇒ Interpretation und Ausführung des (Maschinensprachen-)Programms

**Speicherwerk** (Hauptspeicher, Primärspeicher)

Speichern von Informationen (sowohl Programmcode als auch Nutzdaten *gleichermaßen*)

**Bus** verbindet alle Komponenten und ermöglicht den Informationsaustausch/Datenfluss zwischen ihnen.

**Eingabewerk** „Schnittstelle“ für Eingabegeräte (z. B. USB-Controller, S-ATA-Controller). Es ist jedoch nicht das Peripheriegerät selbst (also nicht die Tastatur) gemeint.

**Ausgabewerk** „Schnittstelle“ für Ausgabegeräte (z. B. Grafikkarte)

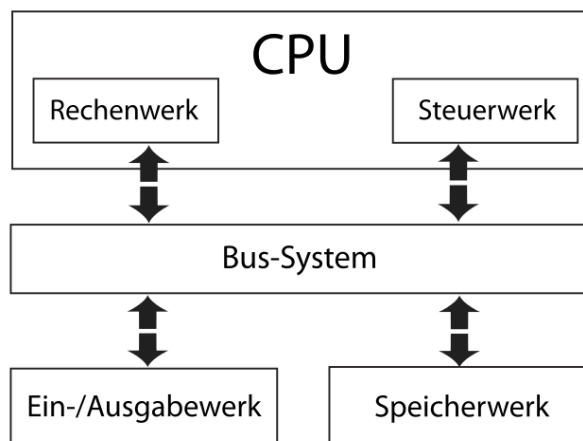


Abbildung 3.2: Vereinfachte Darstellung der von-Neumann-Architektur

### 3.2.2 Harvard-Architektur

Die Harvard-Architektur ist ähnlich der von-Neumann-Architektur, besitzt aber anstatt eines gemeinsamen, zwei getrennte Speicherwerke für Nutzdaten und Programmcode. Zusätzlich kann noch ein optionales zweites Eingabewerk existieren, welches nur für den Programmcode vorhanden ist. Das Speicher- und Eingabewerk für den Programmcode wird über einen zweiten Bus angebunden.

Dadurch ist eine klare physikalische Trennung von Programmcode und Nutzdaten möglich. Abbildung 3.3 auf der nächsten Seite zeigt die Harvard-Architektur und wie sich diese von der von-Neumann-Architektur unterscheidet.

### 3.2.3 Vergleich

Tabelle 3.1 auf Seite 7 vergleicht die von-Neumann-Architektur mit der Harvard-Architektur.



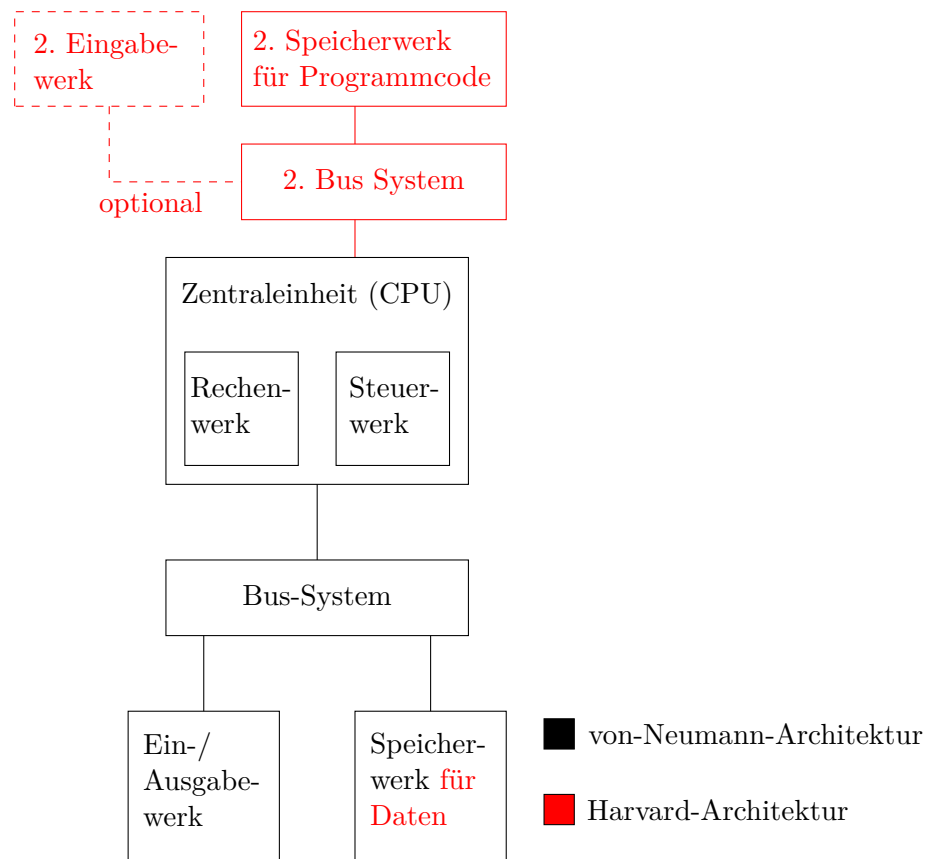


Abbildung 3.3: Vereinfachte Darstellung der Harvard-Architektur

von-Neumann-Architektur	Harvard-Architektur
<ul style="list-style-type: none"> <li>⊖ Virenanfälligkeit: Nutzdaten können als Programm ausgeführt werden</li> <li>⊕ universelle Programmierbarkeit (z. B. Compiler-Ausgabe wird als Programm ausgeführt)</li> <li>⊕ flexible Speicheraufteilung zwischen Programmcode und Daten</li> <li>⊖ möglicher Flaschenhals Bus &amp; Speicherwerk</li> <li>⊕ kostengünstig</li> </ul>	<ul style="list-style-type: none"> <li>⊕ nahezu immun gegen unabsichtlichen Virenbefall</li> <li>⊕ keine (unbeabsichtigten oder ungewollten) Änderungen an der Betriebssoftware möglich</li> <li>⊖ komplexer und teurer (durch 2 Bus und ggf. 2 Eingabewerke)</li> <li>⊖ schwer update-fähig</li> <li>⊕ bessere Performance möglich durch gleichzeitigen Zugriff auf beide Speicherwerke</li> <li>⊖ ohne zweites Eingabewerk gibt es keine Möglichkeit anderen Programmcode auszuführen.</li> <li>⊖ unflexible Aufteilung des Speichers: wenn das eine Speicherwerk voll ist, kann das andere Speicherwerk nicht genutzt werden.</li> </ul>
Einsatz	
<ul style="list-style-type: none"> <li>• übliche <b>PC</b>-Architektur – „Universal-PC“</li> </ul>	<ul style="list-style-type: none"> <li>• „embedded systems“ (z. B. in Waschmaschinen, KFZ-Elektronik, etc.)</li> <li>• Smartphones &amp; Co.</li> <li>• Bestandteile von <b>PCs</b>: BIOS, <b>CPU</b>-Cache in modernen <b>CPUs</b> (Trennung in Cache für Programmcode und Nutzdaten), NX-Flag (Non-Executable) im Hauptspeicher</li> </ul>

Tabelle 3.1: Vergleich der von-Neumann- und Harvard-Architektur

## 4 Rechenwerk / Rechnen in Hardware

### Grundlagen: Addition

	4 7 1 1	Die Addition von mehrstelligen Zahlen wird reduziert auf die Addition von zwei (oder drei) einstelligen Zahlen (bzw. Ziffern) zu einer einstelligen Zahl sowie einem einstelligen Übertrag, also einer zweistelligen Zahl als Ergebnis $\Rightarrow$ genauso funktioniert dies im Binärsystem.
	+ 0 8 1 5	
Übertrag	0 1 0 0	
	5 5 2 6	

#### Hinweis

Herr Röthig meint hier mit „einstellig“ und „mehrstellig“ die Anzahl der Zahlen und nicht die Stellenanzahl einer einzelnen Zahl. Im Beispiel oben werden zwei Zahlen addiert und daraus ergeben sich zwei weitere Zahlen als Ergebnis.

### 4.1 Schaltnetzsynthese – Wiederholung / Grundlegendes

**Schaltnetz** Kann nur die derzeitigen Eingangsdaten verarbeiten, da keine Rückkopplung vorliegt

**Schaltwerk** „Hat ein Gedächtnis“, da eine Rückkopplung vorliegt

**Vollkonjunktion/Minterm** UND-Verknüpfung aller vorkommenden Variablen entweder in negierter oder nicht-negierter Form

**Disjunktive Normalform (DNF)** Eine Disjunktion (ODER-Verknüpfung) von Konjunktionstermen (UND-Verknüpfungen).

**Disjunktive Minimalform (DMF)** Ist die minimale Darstellung einer Ausgabefunktion und damit eine Vereinfachung einer DNF

#### 4.1.1 Schaltungsanalyse

Eine Schaltungsanalyse ist die Bestimmung des „Aufwands“. Dabei kann der Aufwand sein:

- „Hardware-Aufwand“ (in Anzahl an Transistoren)
- Zeitaufwand (in Gatterlaufzeit)

#### 4.1.2 Warum soll der Zeitaufwand analysiert werden?

Es wird der Zeitaufwand betrachtet, da Gatter Schaltzeiten haben, welche typischerweise ~10 Pikosekunden betragen. Insgesamt werden bei einem Signaldurchgang auf dem IC sehr viele Gatter durchlaufen. Damit sind die Schaltzeiten um Größenordnungen größer als die reine Laufzeit der Signale auf dem Leiter angegeben (letztere wird vernachlässigt, Zeitverzögerung wird in „Anzahl Gatterlaufzeiten (GLZs)“ angegeben).

## 4.2 Halbaddierer

**KA: Unterschied was macht Halb-/ was macht volladdierer?**

Ein Halbaddierer vollzieht die Addition von zwei einstelligen Binärzahlen  $a$  und  $b$  zu einer zweistelligen Binärzahl  $c_{out}s$  (Übertrag und Summe). Schaltsymbol und Schaltnetz des Halbaddierers werden in [Abbildung 4.1](#) dargestellt.

Die folgende Tabelle zeigt den Gedankenweg, wie ein Halbaddierer funktioniert.

Nr.	$b$	$a$	$c_{out}$	$s$	Minterm	<b>DNF</b>
0	0	0	0	0		
1	0	1	0	1	$\bar{b}a$	$c_{out} = ba$
2	1	0	0	1	$b\bar{a}$	$s = \bar{b}a \vee b\bar{a}$
3	1	1	1	0	$ba$	$\Rightarrow$ beides gleichzeitig auch <b>DMF</b>

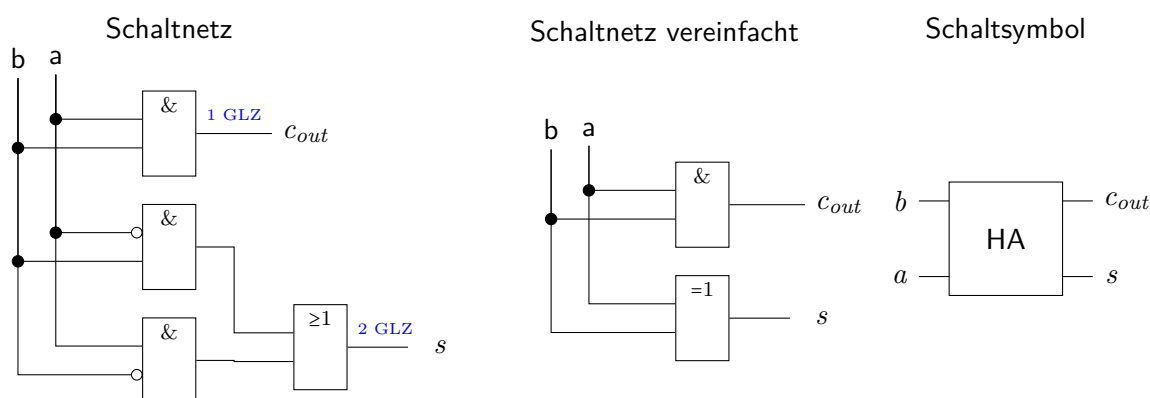


Abbildung 4.1: Halbaddierer – Schaltnetz und Schaltsymbol

## 4.3 Volladdierer

Vollzieht die Addition von drei einstelligen Binärzahlen  $a$ ,  $b$  und  $c_{in}$  zu einer zweistelligen Binärzahl  $c_{out}s$  (Übertrag und Summe). Schaltsymbol und Schaltnetz des Volladdierers werden in [Abbildung 4.2](#) dargestellt.

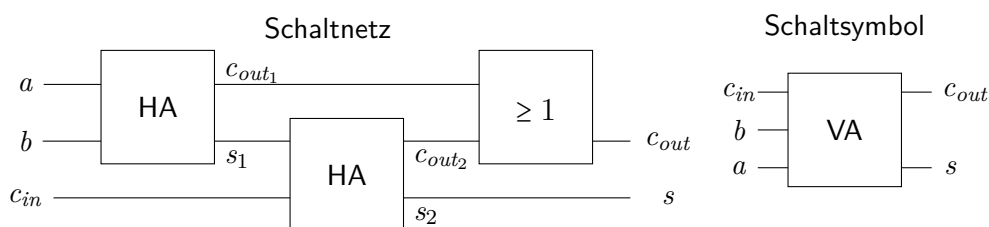


Abbildung 4.2: Volladdierer – Schaltnetz und Schaltsymbol

*Hinweis:* Für die Verknüpfung von  $c_{out1}$  und  $c_{out2}$  zu  $c_{out}$  wäre eigentlich ein XOR notwendig. Da aber der Fall  $c_{out1} = c_{out2} = 1$  (also beide Eingänge des XOR „1“) nie auftritt, reicht ein OR.

#### 4.4 Paralleladdierer (4-Bit-Ripple-Carry-Paralleladdierer RC-PA)

Der **RC-PA** ist ein mehrstelliger Addierer für Binärzahlen. In den folgenden Beispielen ist er ein Addierer vierstelliger Binärzahlen  $a_3a_2a_1a_0$  und  $b_3b_2b_1b_0$ . Das Ergebnis ist  $s_4s_3s_2s_1s_0$  und somit eine 5-stellige Zahl.  $s_4$  ist der Überlauf.

**Abbildung 4.3** zeigt das Schaltnetz und Schaltsymbol eines Paralleladdierers.

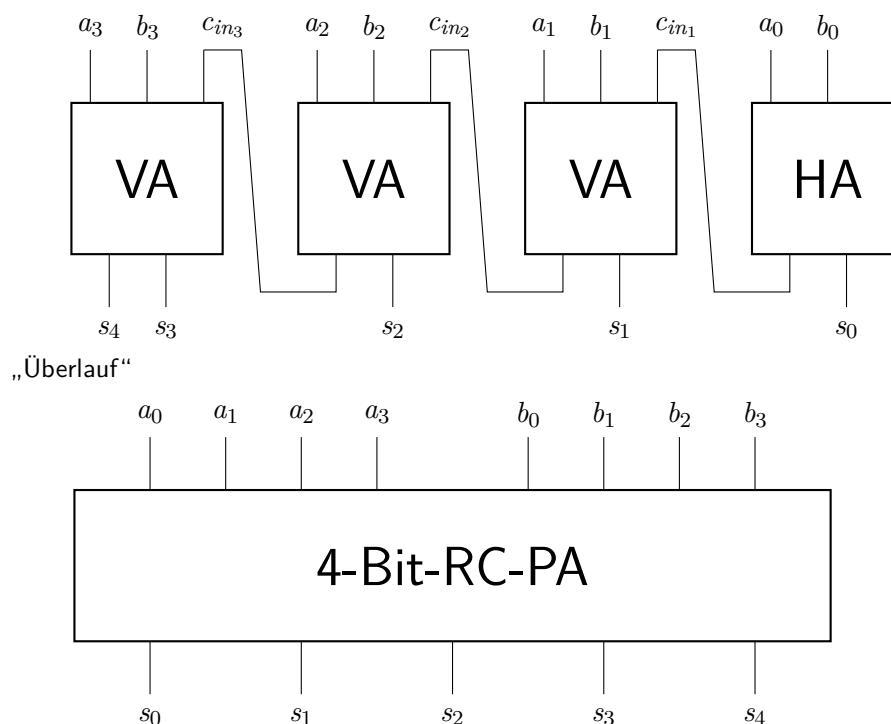


Abbildung 4.3: RC-Paralleladdierer – Schaltnetz und Schaltsymbol

##### Hinweis: Schaltnetz

Ein  $n$ -Bit **RC-PA** ist ein Schaltnetz, kein Schaltwerk! Eine zeichnerische Anordnung mit Verbindungen nur nach unten ist nämlich möglich.

##### 4.4.1 Hardwareaufwand des 4 Bit RC-PA

Aufwand für einen **HA**: 2 Tr. für  $c_{out}$  und 6 Tr. für  $s$   $\Rightarrow$  8 Transistoren

Aufwand für einen **VA**: 2 **HA** und 2 Transistoren für  $c_{out}$   $\Rightarrow 2 \cdot 8 + 2 = 18$  Transistoren

Für den 4 Bit **RC-PA** werden benötigt:

$$\begin{aligned}
 1 \text{ HA} + (n-1) \text{ VA} &\Rightarrow 8 \text{ Transistoren} + (n-1) \cdot 18 \text{ Transistoren} \\
 &\Rightarrow 8 + (18n - 18) \text{ Transistoren} = 18n - 10 \text{ Transistoren} \\
 &= O(n)
 \end{aligned}$$

Dies heißt, dass der **HW**-Aufwand linear mit der Breite der Summanden steigt. Dies ist gut, denn besseres (also weniger Aufwand) ist kaum zu erwarten.

#### 4.4.2 Zeitaufwand des 4 Bit RC-PA

Aufwand für **HA**: max. 2 Gatterlaufzeiten (GLZs) („Tiefe 2“, siehe **Abbildung 4.1** auf **Seite 9**)

Aufwand für **VA**: max. 4 Gatterlaufzeiten (GLZs)

Beim **RC-PA** liegen die einzelnen  $s_i$  nach unterschiedlicher Zeit an.  $s_i$  wird nach  $(i + 1) \cdot 2$  **GLZ** erreicht. Das längste  $s_i$  ist beim  $n$ -Bit-**RC-PA**  $i = n - 1$  und damit ergibt sich ein Zeitaufwand bei  $n$ -Bit-**RC-PA** von  $2n$  Gatterlaufzeit!

Dies ist ein schlechter Zeitaufwand bei einem Paralleladdierer, denn zu erwarten wäre  $O(1)$ !

*Auswirkung:* Beim Wechsel von 32- auf 64-Bit-**CPU** hätte sich die Taktfrequenz halbiert. Daraus lässt sich folgern, dass kein 64-Bit-**RC-PA** in der **CPU** verbaut ist.

#### Hinweis

Anmerkung zum **RC-PA**: Die  $2n$  **GLZ** Zeitaufwand werden nur im „schlimmsten Fall“ bei einer ununterbrochene Kette von  $c_{out}$ -Ausgängen, welche sich **alle** im Laufe der Berechnung von 0 auf 1 ändern, erreicht.

Diese Zeit muss aber trotzdem abgewartet werden!

Zeitaufwand: idealerweise sollte das Schaltnetz sofort nach einer Änderung der Eingangs-Belegung die Ausgangsbelegung entsprechend ändern, in der Realität geschieht das aber nur mit einer entsprechenden Verzögerung  
Gründe:  
endliche Signalgeschwindigkeit mit der eine gewisse Strecke zurückgeleitet werden muss  
Auf- bzw. Entladezeit der Basis, damit die Durchschaltspannung über- bzw. unterschritten wird

#### 4.5 Paralleladdierer (4-Bit-Carry-Look-Ahead-Paralleladdierer CLA-PA)

*Idee:* Der  $c_{in}$ -Eingang wird nicht von vorausgehenden **VA** (oder **HA**) übernommen, sondern durch ein „magisches CLA-Schaltnetz“ nachberechnet. *Genauer:* Für die Berechnung von  $c_{in_i}$  müssen alle vorherigen Eingänge  $a_j, b_j, j < i$  berücksichtigt werden. **Abbildung 4.4** zeigt dieses „magische CLA-Schaltnetz“.

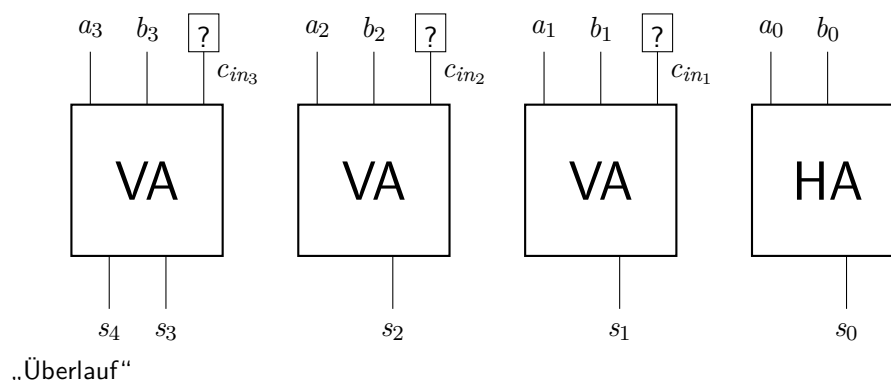


Abbildung 4.4: Carry-Look-Ahead-Paralleladdierer – „magisches“ Schaltnetz

Aber wie sieht das Schaltnetz (und die boolesche Formel) für die Berechnung von  $c_{in_i}$  aus?

$$\boxed{HA}: c_{out} = a \wedge b$$

$$s_{out} = a\bar{b} \vee \bar{a}b$$

$$\boxed{VA}: c_{out} = (a \wedge b) \vee (c_{in} \wedge (a\bar{b} \vee \bar{a}b))$$

$$c_{in_1} = c_{out_0} = a_0 \wedge b_0$$

$\boxed{HA}$

$$c_{in_2} = c_{out_1} = (a_1 \wedge b_1) \vee (c_{in_1} \wedge (a_1\bar{b}_1 \vee \bar{a}_1b_1))$$

$\boxed{VA}$

$$= (a_1 \wedge b_1) \vee (a_0 \wedge b_0 \wedge (a_1\bar{b}_1 \vee \bar{a}_1b_1))$$

Einsetzen

$$c_{in_3} = c_{out_2} = (a_2 \wedge b_2) \vee (c_{in_2} \wedge (a_2\bar{b}_2 \vee \bar{a}_2b_2))$$

$\boxed{VA}$

$$= (a_2 \wedge b_2) \vee ((a_1 \wedge b_1) \vee (a_0 \wedge b_0 \wedge (a_1\bar{b}_1 \vee \bar{a}_1b_1))) \wedge (a_2\bar{b}_2 \vee \bar{a}_2b_2) \dots$$

Einsetzen

...

#### 4.5.1 Zeitaufwand

Für die Berechnung der  $c_{in_i}$  kann jeweils eine  $\boxed{DNF}$ ,  $\boxed{DMF}$  oder jede andere DxF (oder KxF) verwendet werden. Diese haben jeweils nur genau (bzw. maximal) 2  $\boxed{GLZ}$  Zeitaufwand.

Insgesamt hat jedes  $s_i$  beim  $\boxed{CLA-PA}$  genau 6  $\boxed{GLZ}$  Zeitaufwand (außer  $s_0$ : 2  $\boxed{GLZ}$ ,  $s_1$ : 5  $\boxed{GLZ}$ ). Somit haben wir einen konstanten Zeitaufwand von  $O(1)$ .

Eigentlich nur  
4 GLZ

#### 4.5.2 Hardwareaufwand des 4 Bit CLA-PA

Für den Aufwand für die  $c_{in_i}$ -Berechnung gilt die Annahme, das Schaltnetz wäre eine Realisierung der  $\boxed{DNF}$ .

⇒ Für jedes  $c_{in_i}$  gibt es insgesamt  $2i$  Eingänge.

⇒ Insgesamt gibt es max.  $2^{2i}$  verschiedene Vollkonjunktionen für ein  $c_{in_i}$ , welche in der  $\boxed{DNF}$  auftreten können („Zeilen in der Wertetabelle“).

Jede dieser Vollkonjunktionen wird mit  $2i$  Transistoren realisiert.

⇒ Falls alle Vollkonjunktionen verwendet werden müssten, wäre der Hardwareaufwand  $2^{2i} \cdot 2i$  Transistoren  $= 2i \cdot 4^i = O(n \cdot 4^n)$ .

In der Realität werden natürlich nicht alle Vollkonjunktionen benötigt, sondern ein (vermutlich halbwegs konstanter) Anteil  $0 < k < i$ . Damit ist der Aufwand für  $c_{in_i} = O(i \cdot 4^i)$  und somit der Aufwand für  $n$ -Bit- $\boxed{CLA-PA}$ :  $O(n \cdot n \cdot 4^n) = O(n^2 \cdot 4^n)$

**Achtung**

Im folgenden wird fälschlicherweise von einem Aufwand  $O(n^2 \cdot 2^n)$  ausgegangen. Richtig wäre  $O(n^2 \cdot 4^n)$ . Herr Röthig hat die  $O$ -Notation falsch vereinfacht. Der 2015er Jahrgang hat dies „noch falscher“ gemacht und zu  $O(4^n)$  vereinfacht.

Der Hardwareaufwand steigt beim  $n$ -Bit-CLA-PA überexponentiell mit  $n$ . Beim Wechsel von 32-Bit auf 64-Bit-CLA-PA wäre der 16 Trillionen-fache Aufwand an Transistoren nötig gewesen.

Bei  $n = 4$ :  $4^2 \cdot 2^4 = 16 \cdot 16 = 256$  Transistoren

Bei  $n = 8$ :  $8^2 \cdot 2^8 = 64 \cdot 256 = \sim 16384$  Transistoren (64-fache von  $n = 4$ )  $\Rightarrow$  zu viel für die CPU

**4.5.3 Kombination mehrerer kleinen CLA-PAs**

Der 32-Bit Addierer wird in acht 4-Bit-CLA-PA gesplittet (siehe Abbildung 4.5).

$\Rightarrow$  hintereinander geschaltet nach RC-Prinzip

$\Rightarrow$  damit ist das  $n$  der nicht-CLA-PA noch klein  $\Rightarrow$  erträglicher Hardwareaufwand

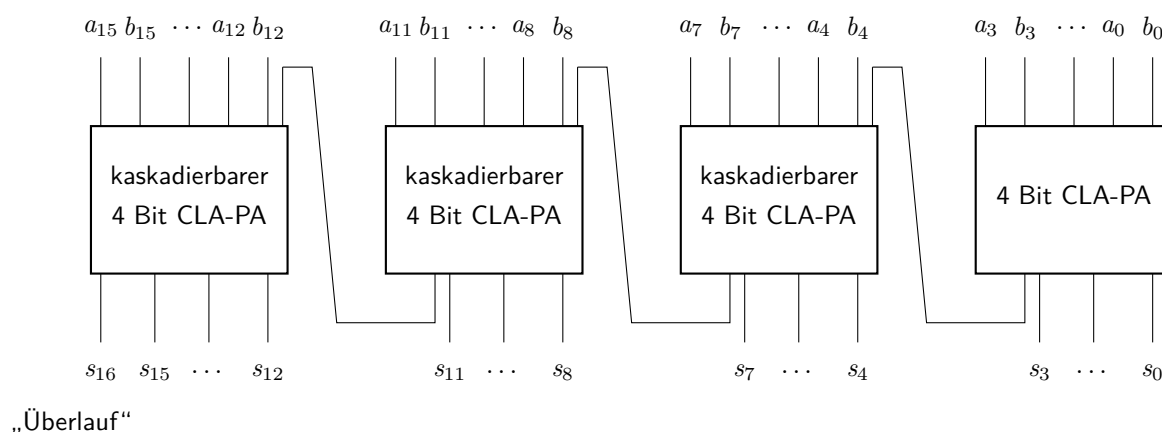


Abbildung 4.5: Kaskadierbarer 4-Bit-CLA-PA



## 4.6 Serielladdierer

*Idee:* Angelehnt an die Verfahrensweise des Menschen sollen die Stellen der beiden Summanden nacheinander (und nicht gleichzeitig) addiert werden. Dadurch wird nur **ein** Volladdierer (**VA**) und mehrere Schieberegister (**SR**) benötigt. Daher ist der Serielladdierer (**SA**) ein Schaltwerk, kein Schaltnetz! **Abbildung 4.6** zeigt das Schaltwerk eines Serielladdierers.

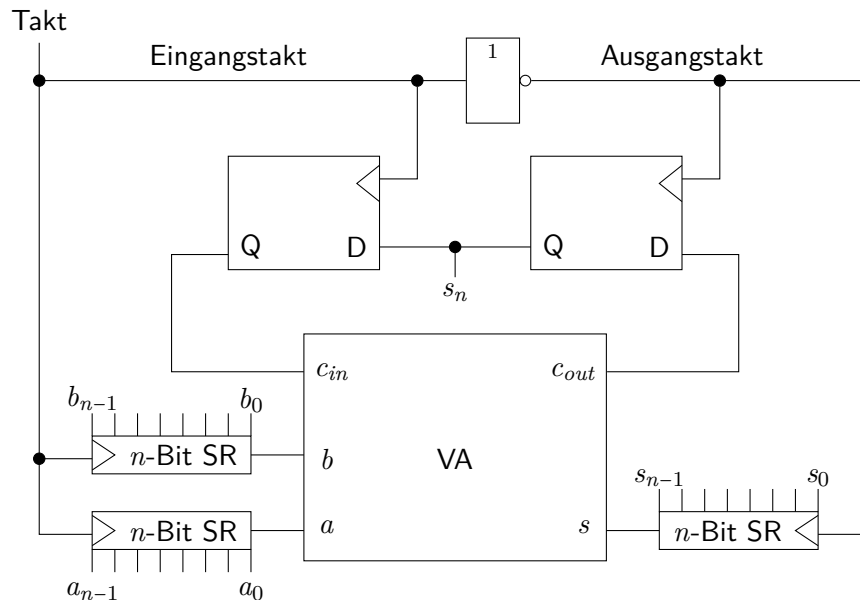


Abbildung 4.6: Serielladdierer

### 4.6.1 Zeitaufwand ( $n$ -Bit-SA)

Der Zeitaufwand für einen  $n$ -Bit-Serielladdierer beträgt  $n$  Taktzyklen, also  $O(n)$

*Ist dies wie beim **RC-PA**?*

Jein, denn 1 Taktzyklus dauert deutlich mehr als doppelt so lang wie die Berechnung des **VA** (Sicherheitsmargen!). Beispiel: 1 Taktzyklus  $> \sim 10$  **GLZ**  $\Rightarrow$  fünffache Berechnungszeit des **RC-PA**

### 4.6.2 Hardwareaufwand ( $n$ -Bit-SA)

1 **VA** = 18 Transistoren

2 **D-FF** =  $2 \cdot 6$  = 12 Transistoren. (siehe **Abbildung 4.7b**)

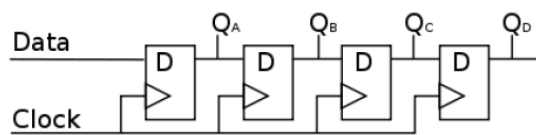
3  $n$ -Bit-**SR** =  $3 \cdot 6n$  =  $18n$  Transistoren (siehe **Abbildung 4.7a**)

Takterzeugung (im folgenden nicht näher betrachtet)

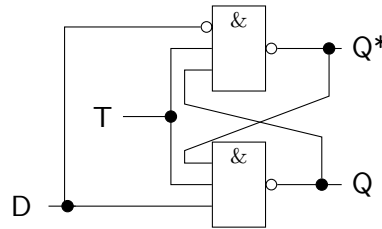
**gesamt**  $18n + 30$  Transistoren

Zum Vergleich: **RC-PA**  $18n - 10$ , d. h. der Serielladdierer braucht 40 Transistoren mehr (bei längerer Bearbeitungszeit)!

in einem halben Taktzyklus (zwischen aktiver Flanke von Eingangstakt und aktiver Flanke von Ausgangstakt) müssen die SR a und b sowie das zweite D-FF für cin ausgelesen (1 GZ), der VA rechnen (4 GZ) sowie das SR von s und das erste D-FF für cout ausgelesen werden (1 GZ)



(a) 4-Bit Schieberegister



(b) Hardwareaufwand für einen D-FF: 6 Tr.

Abbildung 4.7: Schieberegister aus D-FF mit Hardwareaufwand

### Achtung

Die Takterzeugung muss in der Klausur für den Serielladdierer auf jeden Fall genannt werden, auch wenn sie hier nicht weiter betrachtet wird!

### 4.6.3 Hardwareoptimierung des Serielladdierers

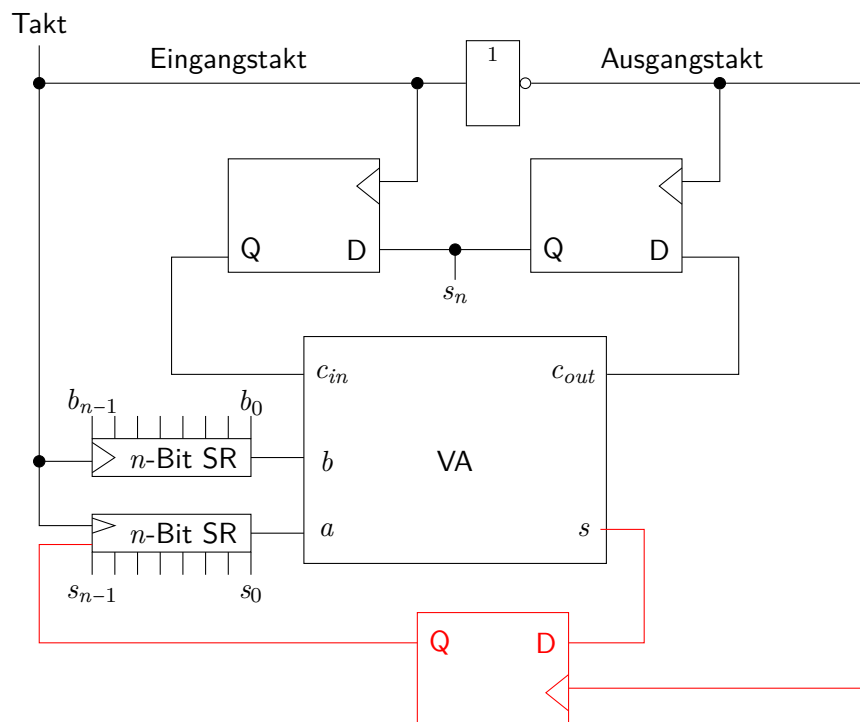


Abbildung 4.8: Serielladdierer mit Hardwareoptimierung

Wie in [Abbildung 4.8](#) zu sehen ist, wird ein Schieberegister weniger benötigt  
 $\Rightarrow 12n + 36$  Transistoren

n	RC-PA ( $18n - 10$ )	SA ( $12n + 36$ )
1	8	48
2	26	60
3	44	72
4	62	84

Tabelle 4.1: Vergleich des Hardwareaufwand eines RC-PA mit dem verbesserten SA

### Vergleich eines RC-PA mit dem verbesserten SA

Tabelle 4.1 vergleicht einen RC-PA mit dem verbesserten SA.

Break-Even („Gewinnschwelle“):  $18n - 10 = 12n + 36 \Rightarrow 46 = 6n \Rightarrow n = 7\frac{2}{3}$

$\Rightarrow$  Ab 8-Bit lohnt sich der SA

### Mögliche Anwendung eines Serielladdierers:

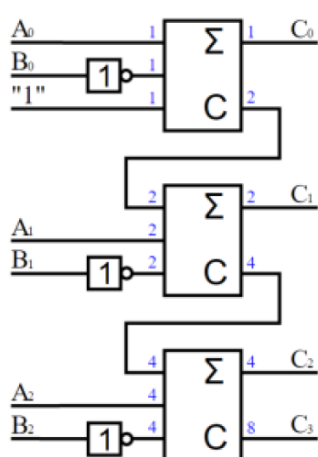
- nicht im Rechenwerk der CPU (aufgrund der Geschwindigkeit)!
- aber möglicherweise in „embedded systems“, falls z. B. Sensordaten sowieso seriell angeliefert werden

$\Rightarrow$  eventuell sind sogar weitere Hardware-Einsparungen möglich!

## 4.7 Subtraktion

Es ist kein spezieller Hardware-Subtrahierer notwendig. Eine Subtraktion wird über die Addition des 2er-Komplements realisiert:

1. Bits invertieren (Inverter)
2. +1 addieren (Addierer)



1. Erstellen eines 3-Bit Addierers
2. B wird negiert
3. +1 im ersten Schritt rechnen. Dafür den ersten Halbaddierer in einen Volladdierer wandeln und eine logische 1 anlegen!

Abbildung 4.9: Schaltnetz Subtrahierer

## 4.8 Multiplikation

4 7 1 1 × 8 1 5	Mögliche Probleme / Schwierigkeiten
3 5	1. kleines $1 \times 1$ ist ein wenig Lernaufwand
6 7 6 8 8	2. Überträge beim kleinen $1 \times 1$ sind möglich
4 7 1 1	3. Addition von mehr als zwei Ziffern gleichzeitig
2 3	4. mehrstellige Überträge bei der Summenbildung möglich
2 3 5 5 5	5. für jede Zwischensumme muss der Addierer eine Stelle mehr verarbeiten
3 8 3 9 4 6 5	

**Punkt 5** wird deutlich durch eine Multiplikation einer 6- und 4-stelligen Zahl im Binärsystem „von links nach rechts“, wie sie in **Tabelle 4.2** dargestellt wird.

101010 × 1010	(42 × 10)
101010 000000	7-stellige Addition
1010100 101010	Zwischensumme 8-stellige Addition
11010010 000000	Zwischensumme 9-stellige Addition
110100100	10-stelliges Ergebnis (Übertrag: 0)

Tabelle 4.2: Schriftliche Multiplikation „von links nach rechts“

## Abhilfe

b	a	ab	
0	0	0	1. Kleines $1 \times 1$ im Binärsystem ist ein einfaches UND
0	1	0	2. Es gibt keine Überträge beim kleinen $1 \times 1$ im Binärsystem
1	0	0	3. Bilden von Zwischensummen
1	1	1	4. Keine mehrstelligen Überträge bei Addition von zwei Summanden
			5. Rechten Faktor beginnend mit niederwertigster Stelle abarbeiten, siehe

1	0	1	0	1	0	$\times$	1	0	1	0	$(42 \times 10)$
				0	0	0	0	0	0	<b>0</b>	
				1	0	1	0	1	0	$p_0$	
				1	0	1	0	1	0	<b>0</b>	
				0	0	0	0	0	0	$p_1$	
				0	1	0	1	0	0	<b>1</b>	
				1	0	1	0	1	0	$p_2$	
				0	1	1	0	1	0	0	
				$p_9$	$p_8$	$p_7$	$p_6$	$p_5$	$p_4$	$p_3$	

Tabelle 4.3: Schriftliche Multiplikation „von rechts nach links“

### 4.8.1 Multiplikation mit Paralleladdierer

Ein  $n \times m$ -Bit-Parallelmultiplikierer (PM), bspw. ein  $5 \times 4$ -Bit-Parallelmultiplikierer ist in **Abbildung 4.10** dargestellt.

Schaltnetz

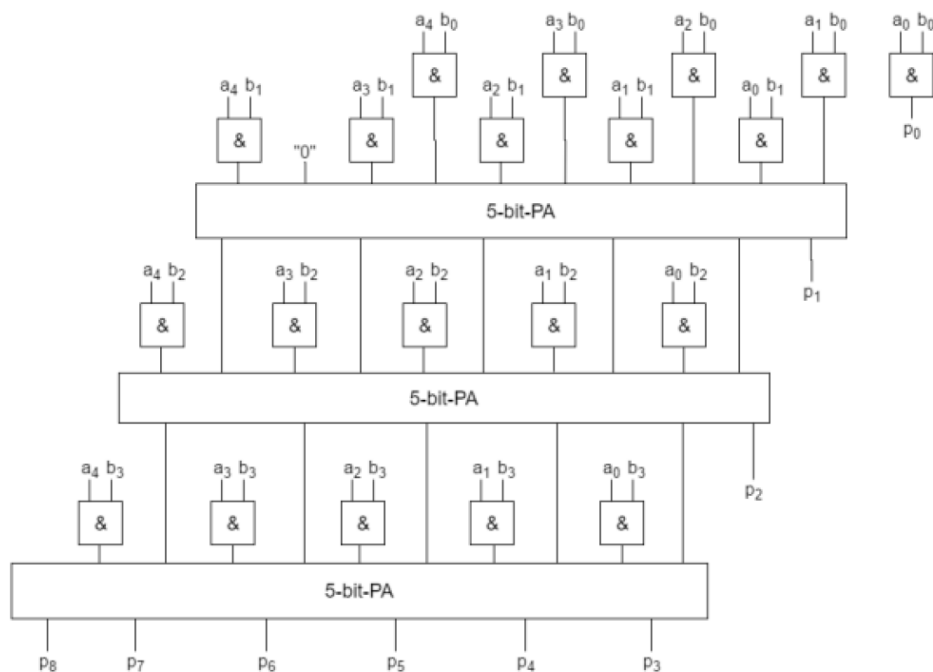


Abbildung 4.10: 5-Bit Parallelmultiplikierer

#### Hinweis zur Abbildung

Unbedingt auf die „0“ als Eingang achten! Ansonsten gibt es in der Klausur punktabzug! Die „0“ ist gesehen der erste Übertrag.

#### Hinweis

In der Klausur muss evtl. ein 4-Bit Parallelmultiplikierer gezeichnet werden, also nicht mit 5 oder 6 Bit.

#### Analyse: Hardwareaufwand für die Multiplikation mit Paralleladdierer

Benötigt werden:

- $m - 1$   $n$ -Bit-PA
- $n \cdot m$  UND mit jeweils 2 Eingängen  $\Rightarrow 2 \cdot n \cdot m$  Transistoren

Somit ergibt sich bei...

... Verwendung eines **RC-PA**:

- $\Rightarrow n\text{-Bit-RC-PA: } 18n - 10 \text{ Transistoren}$
- $\Rightarrow \text{davon } m - 1: (m - 1)(18n - 10) \text{ Transistoren} = 18nm - 18n - 10m + 10 \text{ Transistoren}$
- $\Rightarrow \text{Insgesamt: } 20nm - 18n - 10m + 10 \text{ Transistoren} \Rightarrow O(nm)$

... Verwendung eines **CLA-PA**:

- $n\text{-Bit-CLA-PA: } \approx O(n \cdot 2^n)$  (Achtung: eigentlich  $O(n^2 \cdot 4^n)$ )
- $\Rightarrow \text{davon } m - 1: (m - 1) \cdot O(n \cdot 2^n) = O(n \cdot m \cdot 2^n)$

Hier bei verschiedenen großen Faktoren also besser  $m > n$  bei Carry-Look-Ahead-Paralleladdierer (**CLA-PA**) für einen geringeren **HW**-Aufwand.

Demgegenüber sollte bei Verwendung von **RC-PA** besser  $n > m$  für geringeren **HW**-Aufwand sein.

### Analyse: Zeitaufwand für die Multiplikation mit Paralleladdierer

1 **GLZ** für einstellige Multiplikation (UND-Gatter) sowie  $(m - 1) \times \text{Berechnungszeit}(n\text{-Bit-PA})$

1. Annahme: **PA** sind  $n\text{-Bit-RC-PA}$

Berechnungszeit eines  $n\text{-Bit-RC-PA}$ :  $2n$  **GLZ**

Insgesamt:  $1 + (m - 1) \cdot 2n = 2nm - 2n + 1$  **GLZ**

Damit besser  $n > m$  bei Verwendung von **RC-PA**, um geringeren Zeitaufwand zu bekommen.

2. Annahme: **PA** sind  $n\text{-Bit-CLA-PA}$

Berechnungszeit eines  $n\text{-Bit-CLA-PA}$ :  $6$  **GLZ**

Insgesamt:  $1 + (m - 1) \cdot 6$  **GLZ**  $= 6m - 5$  **GLZ**  $= O(m)$

Damit besser  $n > m$  bei Verwendung von **CLA-PA**, um geringeren Zeitaufwand zu bekommen

⚡ zu großer **HW**-Aufwand (wächst exponentiell mit  $n$ )

### Hinweis für die Klausur

Den logarithmischen Aufwand für **CLA-PA** auf [Wikipedia](#) nachschauen. Dies wird zwar wahrscheinlich nicht in der Klausur abgefragt, aber er könnte abfragen, welche Alternativen es gibt. Unserer Variante hat exponentiellen Hardwareaufwand und konstanten Zeitaufwand. Die Variante auf [Wikipedia](#) nicht.

### 4.8.2 Seriellmultiplizierer

Motivation: Wir wollen eine noch engere Anlehnung an das schriftliche Multiplikationsverfahren, um den Aufwand für die Addierglieder gering zu halten.

Abbildung 4.11 zeigt einen 5-Bit-Seriellmultiplizierer.

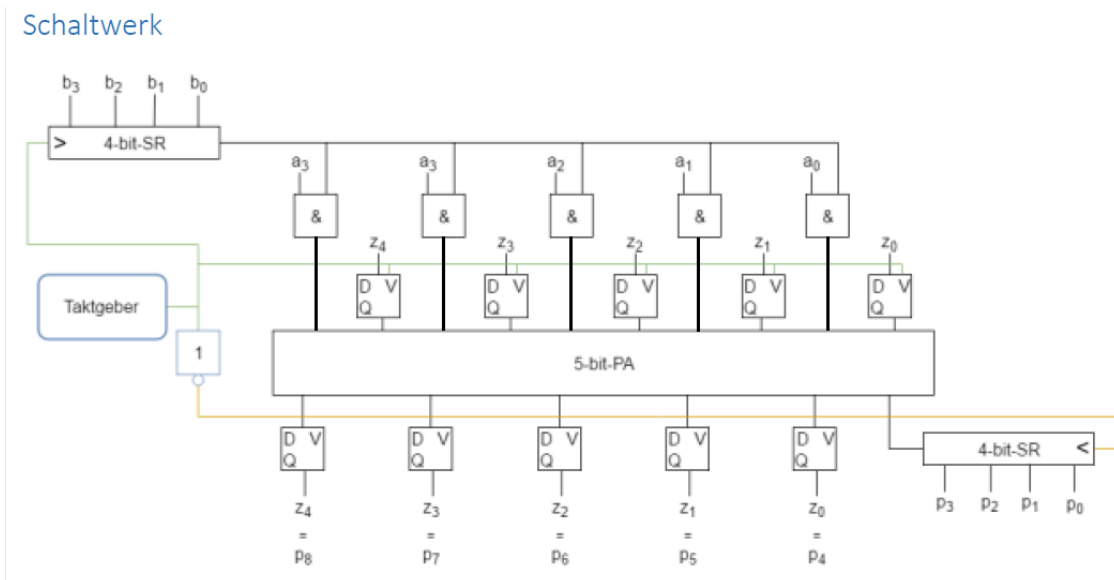


Abbildung 4.11: 5-Bit Seriellmultiplizierer

#### Hinweis

Vor dem ersten Takt müssen alle **D-FFs** auf 0 gesetzt werden („Reset-Eingang“ oder ähnliches)

#### Analyse: Hardwareaufwand für $n \times m$ -Seriellmultiplizierer

Benötigt werden:

- $n$ -Bit-**PA**  $\Rightarrow$   $n$ -Bit-**RC-PA**  $\Rightarrow 18n - 10$  Transistoren
- $n$  UND mit jeweils 2 Eingängen  $\Rightarrow 2n$  Transistoren
- $2n$  **D-FFs**  $\Rightarrow 12n$  Transistoren
- $2m$ -Bit-**SR**  $\Rightarrow 12m$  Transistoren
- Takterzeugung (wird im folgenden nicht berücksichtigt)

gesamt:  $18n - 10 + 2n + 12n + 12m = 32n + 12m - 10$  Transistoren

$$\Rightarrow O(n + m)$$

**Analyse: Zeitaufwand für  $n \times m$ -Seriiellmultiplizierer mit RC-PA**

Zeitaufwand:  $m$  Taktzyklen

Mindestdauer eines halben Taktzyklus: Berechnungszeit „innen“

$$\begin{aligned}
 \frac{1}{2} \text{ Taktzyklus} &\gtrsim \text{Auslesezeit (SR)} + \text{UND} + n\text{-Bit-PA} + \text{Einlesezeit (SR)} \\
 &= 1 + 1 + 2n + 1 = 2n + 3 \text{ GLZ} \\
 \Rightarrow 1 \text{ Taktzyklus} &\gtrsim 4n + 6 \text{ GLZ} \\
 \text{gesamt:} &\gtrsim m \cdot (4n + 6) = 4mn + 6m = O(nm)
 \end{aligned}$$

**4.9 Division**

4 7 1 1 : 1 3 = 0362, ...	Die Division wird runter gebrochen auf mehrere Divisionen mit einstelligem Ergebnissen.
0	
4 7	
3 9	Dies ist recht komplex bei einem großem Divisor. Aber im Binärsystem ist es ein einfacher Größenvergleich, da das einstellige Ergebnis nur 1 oder 0 sein kann.
8 1	(besser wäre Integration in anschließend notwendiger Subtraktion)
7 8	
3 1	
2 6	Trotzdem bleiben einige Schritte zu tun, was aufwändig ist. In heutigen CPUs ist deshalb (meist) kein Hardware-Dividierer eingebaut.
5	

*Grund:* Die Division wird deutlich seltener als Addition, Subtraktion und Multiplikation gebraucht.

*Stattdessen:* Division „in Software“/„als Programm“, z. B. als Bibliotheksroutine („Funktion“), evtl. bereitgestellt vom Betriebssystem oder als Mikroprogramm, welches Bestandteil der CPU ist und aufgerufen wird, wenn ein entsprechender Maschinensprachenbefehl ausgeführt werden soll.

**4.10 CPUs**

In Tabelle 4.4 werden RISC und CISC verglichen.



Reduced Instruction Set Computer ( <b>RISC</b> )	Complex Instruction Set Computer ( <b>CISC</b> )
für jeden Maschinensprachenbefehl gibt es eine „passende“ Hardwareeinheit	manche Maschinensprachenbefehle werden als Mikroprogramm ausgeführt
<ul style="list-style-type: none"> <li>⊕ Einheit für Mikroprogrammausführung kann bei der Produktion entfallen (CPU weniger komplex)</li> <li>⊖ <b>CPU</b> wird komplex, falls die Forderung nach komplexen Befehlen in Maschinensprache realisiert, besteht.</li> <li>⊕ jeder Befehl kann in wenigen (und jeweils einer festen Zahl an) Taktzyklen ausgeführt werden.</li> </ul>	<ul style="list-style-type: none"> <li>⊕ komfortables Programmieren in Maschinensprache</li> <li>⊖ bei manchen Befehlen (welche als Mikroprogramm ausgeführt werden) ist die Bearbeitungszeit sehr groß und variabel.</li> <li>⊖ Mikroprogramme sind Software, komplex und fehleranfällig, ggf. Austausch der <b>CPU</b> für „Bugfix“ notwendig (vgl. <b>Intel Pentium FDIV Bug</b>)</li> </ul>
Einsatz	
PowerPC- <b>CPU</b> in Apple-Rechnern bis vor 10 Jahren oft in „embedded systems“ (Bsp. ARM)	Intel- <b>CPU</b> s in heutigen Universalrechnern (und AMD-Äquivalent)

Tabelle 4.4: Vergleich von RISC und CISC

## 5 Speicher in Computern

### 5.1 Speicherhierarchie

**Tabelle 5.1** zeigt die Speicherhierarchie in modernen Universalrechnern

Speicher	von-Neumann-Rechner	Persistenz	Größe	Zugriffszeit
<b>CPU</b> Register	Zentraleinheit (Rechen- und Steuerwerk)	flüchtig	< 1kB Byte	≈ 200ps
Cache („CPU-Cache“)	nicht vorhanden	flüchtig	~4MB L1 ≈ 64KB L2 ≈ 512KB	≈ 10ns
Hauptspeicher/Primärspeicher	Speicherwerk	flüchtig	~8GB	≈ 100ns L1 schneller
Sekundärspeicher	Peripheriegeräte an Ein- und Ausgabe- werk	nicht flüchtig	HDD: 3T SSD: 512GB opt. <b>LW</b> bis 100GB <b>BLW</b> : wenige TB	HDD ≈ 10ms SSD ≈ 10μs opt. <b>LW</b> ≈ 1s <b>BLW</b> ≈ 1min

Tabelle 5.1: Speicherhierarchie und -Daten

### 5.2 Sekundärspeicher

Motivation für Sekundärspeicher: nicht-flüchtig, d. h. der Speicherinhalt bleibt auch bei Stromausfall erhalten.

#### Anwendungen:

- Programmcode (Betriebssystem, Anwendungsprogramme)
- Nutzdaten
- virtuelle Erweiterung des Speicherwerks (Swap-Datei/-Partition)
- Hibernation („Ruhezustand“)

### 5.3 CPU-Cache

Temporärer, flüchtiger, schneller Zwischenspeicher, um auf Informationen aus dem Hauptspeicher schneller zugreifen zu können.

Eigenschaften des Cache:

- flüchtig
- kleiner als das zu cachende Medium (Hauptspeicher)
- schneller als das zu cachende Medium (Hauptspeicher)
- transparent, d. h. es wird nicht auf den Cache, sondern auf das zu cachende Medium logisch zugegriffen (die **CPU** adressiert den Hauptspeicher und nicht den Cache)
- konsistent, d. h. alle Instanzen derselben Hauptspeicheradresse (**HSA**) haben denselben Wert
- kohärent, d. h. beim Zugriff auf eine **HSA** wird immer der aktuelle Wert geliefert

#### Anmerkung: Kohärenz und Konsistenz

Kohärenz ist Pflicht und Konsistenz ist Wunsch, da aus Konsistenz Kohärenz folgt. Zeitweise kann aber auf Konsistenz verzichtet werden.

### 5.3.1 Lokalität des Zugriffsmusters

Verantwortlich dafür, dass der Cache Geschwindigkeitsvorteile bringen kann.

**räumliche Lokalität** Wenn auf eine Adresse zugegriffen wird, wird auch auf naheliegende Adressen zugegriffen.

**zeitliche Lokalität** Die Zugriffe (auf nahe beieinanderliegende Adressen) erfolgen in relativ geringem zeitlichen Aufwand

#### Hinweis

Hinweis: Insbesondere die räumliche Lokalität ist beim Zugriff auf Programmcode und Nutzdaten sehr unterschiedlich (Programmcode: sequentiell, Nutzdaten zufällig innerhalb von Speicherblöcken).

⇒ Moderne **CPU**s weisen getrennte Caches für Programmcode und Nutzdaten auf!

### 5.3.2 Begriffe

**Hit** Zugriff auf Hauptspeicher-Daten, welcher aus dem Cache bedient werden kann

**Miss** Zugriff auf Hauptspeicher-Daten, welcher *nicht* aus dem Cache bedient werden kann und deshalb der Cache die Daten erst aus dem Hauptspeicher holen muss.

**Hit-Rate** Anteil der „erfolgreichen“ Zugriffe an allen Zugriffen.

$$\text{Hit-Rate} = \frac{\# \text{Hits}}{\# \text{Zugriffe}}$$

**Miss-Rate** Anteil der „nicht erfolgreichen“ Zugriffe an allen Zugriffen.

$$\text{Miss-Rate} = \frac{\# \text{Misses}}{\# \text{Zugriffe}}$$

**Zugriffe** Anzahl der Misses plus der Hits

<b>Anwendung:</b>	$0 \leq \text{Hit-Rate} \leq 1$	<b>Ziel:</b>	$\text{Hit-Rate} \rightarrow 1$ (nicht realistisch)
	$0 \leq \text{Miss-Rate} \leq 1$		$\text{Miss-Rate} \rightarrow 0$
	$\text{Hit-Rate} + \text{Miss-Rate} = 1$		$\text{Hit-Rate} \rightarrow \text{systembedingtes Maximum}$ (realistisch)

Das systembedingte Maximum hängt ab von:

- Lokalität des Zugriffsmusters
- Größe (und Größenverhältnis) von Cache und Hauptspeicher

Hinweis:  $\text{Hit-Rate}_{\text{erreicht}} \geq \frac{\text{Größe (Cache)}}{\text{Größe HS}}$  (Wahrscheinlichkeit, das ein beliebiger Zugriff eine bereits im Cache gespeicherte Adresse betrifft)

⇒ sobald das Zugriffsmuster Lokalität aufweist, ergibt sich eine bessere Hit-Rate

### 5.3.3 Betriebszustände des Cache

**kalter Cache** bei Betriebsbeginn ist der Cache leer

**sich erwärmender Cache** Während des Betriebs wird der Cache mit immer mehr Daten geladen und die Hit-Rate steigt.

**heißer Cache** Der Cache ist nach einer gewissen Betriebszeit (nahezu) voll. Die Hit-Rate erreicht das systembedingte Maximum.

### 5.3.4 Cachearchitekturen

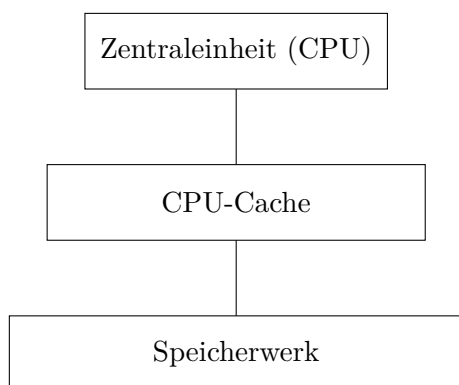


Abbildung 5.1: Look-Through-Cache

links: 1:1  
Verbindung  
rechts: Bus  
Verbindung

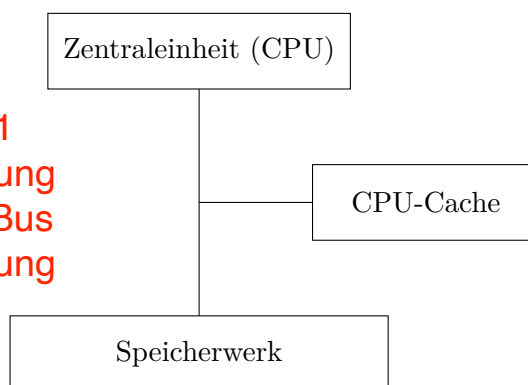


Abbildung 5.2: Look-Aside-Cache

#### Look-Through-Cache

Wie in [Abbildung 5.1](#) zu sehen, ist die CPU nur mit dem Cache und der Hauptspeicher ebenfalls nur mit dem Cache verbunden.

Die **CPU** greift über den Cache auf den Hauptspeicher zu:

#### Look-Aside-Cache

In [Abbildung 5.2](#) sind **CPU**, Cache und **HS** über einen Bus miteinander verbunden. Die Anfrage durch die CPU geht auf dem Bus an beide und ggf. antworten beide, d. h. die schnellere Antwort gewinnt.

$$\ominus t_{\text{Miss}} = t_{\text{Cache}} + t_{\text{HS}}$$

Zugriffszeit bei Miss größer als ohne Cache

⊕ optimale Konsistenz

$$\oplus t_{\text{Miss}} > t_{\text{HS}}$$

Zugriffszeit bei einem Miss genauso wie ohne Cache  $\Rightarrow$  immer „beste“ Zugriffszeit

⊖ Bus braucht Zugriffsprotokoll mit Overhead  $\Rightarrow$  langsamer als 1:1 Verbindung

⊖ Konsistenz durch zweite Antwort potentiell gefährdet

- gefährdete Kohärenz bei Inkonsistenz und Hit  
- Kollisionsgefahr, deshalb Medienzugriffsverfahren nötig  $\Rightarrow$  erhöhter Overhead, insgesamt langsamere Kommunikation (doppelte Bus Belastung bei jedem Hit)

### 5.3.5 Schreibstrategie

#### Write-Back

Schreibzugriff durch die **CPU** findet im Cache statt und der Cache aktualisiert die Daten bei nächster Gelegenheit im Hauptspeicher.

⊖ zeitwertige Inkonsistenz der Daten

⊕ Schreiben in Cache-Geschwindigkeit möglich

#### Write-Through

Schreibzugriff durch die **CPU** findet im Hauptspeicher statt. Parallel dazu müssen die Daten im Cache invalidiert (schlecht) oder ebenfalls geschrieben werden (gut).

⊕ optimale Konsistenz der Daten

⊖ Schreiben nur in **HS**-Geschwindigkeit möglich

Schreibstrategie	Write-Back	Write-Through
Cachearchitektur		
<b>Look-Through</b>	⊕ gute klassische Kombination, da die physische Gegebenheit vorhanden ist, um direktes Schreiben in Cache und Rückschreiben vom Cache in <b>HS</b> zu ermöglichen	⊖ Kombination nicht möglich, da kein direkter Zugriff der <b>CPU</b> auf <b>HS</b> physisch gegeben ist.
<b>Look-Aside</b>	⊖ schlechte Kombination, da bei jedem Schreibzugriff der Bus zweimal belastet wird	⊕ gute klassische Kombination, da Schreibzugriffe parallel im <b>HS</b> und Cache physisch gut machbar sind

### 5.3.6 Cache-Aufbau

**Hauptspeicherseite (**HSS**)** Gleich große Speicheranteile des Hauptspeichers. Eine **HSS** sollte  $2^m$  Hauptspeicheradressen (**HSA**) beinhalten, um eine einfache Umrechnung von **HSS**-Nummern und **HSA** zu ermöglichen.

**Cache Line (**CL**)** Kopie einer Hauptspeicherseite im Cache

**Tag** Die um  $m$  niederwertigsten Bit gekürzte **HSA**, welche der **HSS**-Nummer entspricht.

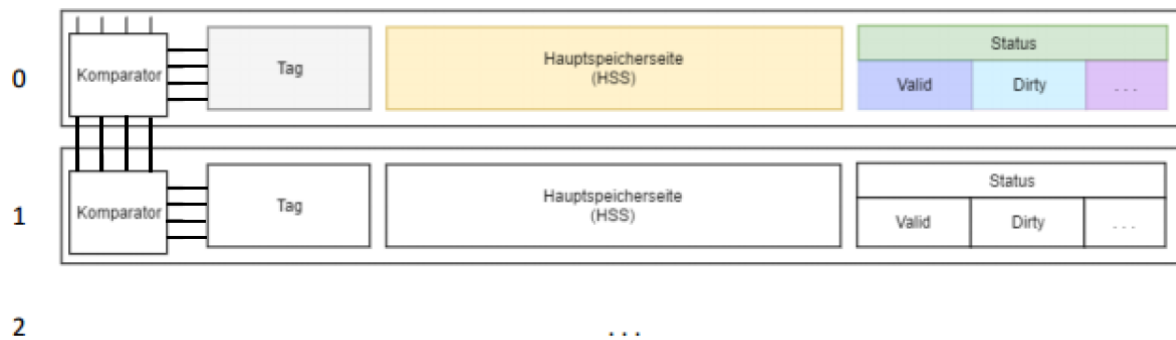


Abbildung 5.3: Aufbau des Caches

**Status** Zustandsinfo zur Cache-Line, z. B. Valid-Flag und weitere Zustandsinfos abhängig von Verdrängungsstrategie

**Valid-Flag** Die Daten werden im Cache geändert und müssen noch in den **HS** zurückgeschrieben werden (nur bei Write-Back-Schreibstrategie) **Gegenteil: Dirty-Flag**

### 5.3.7 Vollasoziativer Cache

Jede Hauptspeicherseite kann in jeder Cache Line eingelagert werden (nicht gleichzeitig!)

- ⇒ bei jedem Zugriff auf eine Hauptspeicheradresse (**HSA**) muss überprüft werden, ob die gekürzte **HSA** einem der Tags von validen Cache Lines entspricht!
- ⇒ Vergleichen der gekürzten **HSA** mit allen Tags (von validen **CL**)

Wie kann verglichen werden? Welche Möglichkeiten des Vergleichs gibt es?

- Sequentiell ⇒ Nachteil: erhöhte Zugriffszeit
- Parallel ⇒ gleichzeitiges Vergleichen der angelegten (gekürzten) **HSA** mit allen Tags über jeweils einen eigenen Komparator in jeder **CL**  
Nachteil: Hardware-Aufwand für Komparator in jeder **CL**

#### Einschub: Schaltungssynthese Komparator

Hier: Schaltnetz, welches zwei Zahlen auf Gleichheit, Größer und Kleiner vergleicht.



Abbildung 5.4: n-Bit-Komparator

Möchte man eine Wertetabelle für einen n-Bit-Komparator erstellen, sieht man, dass zu viele Zeilen in der Wertetabelle ( $2^{2n} \dots$ ) sind ⇒ besser mit kaskadierbaren 1-Bit-Komparatoren

### Komparator für Gleichheit

*Hinweis:* Im Cache reicht der Vergleich auf Gleichheit. **Abbildung 5.5** zeigt 1-Bit-Komparatoren für einen Gleichheits-Komparator.

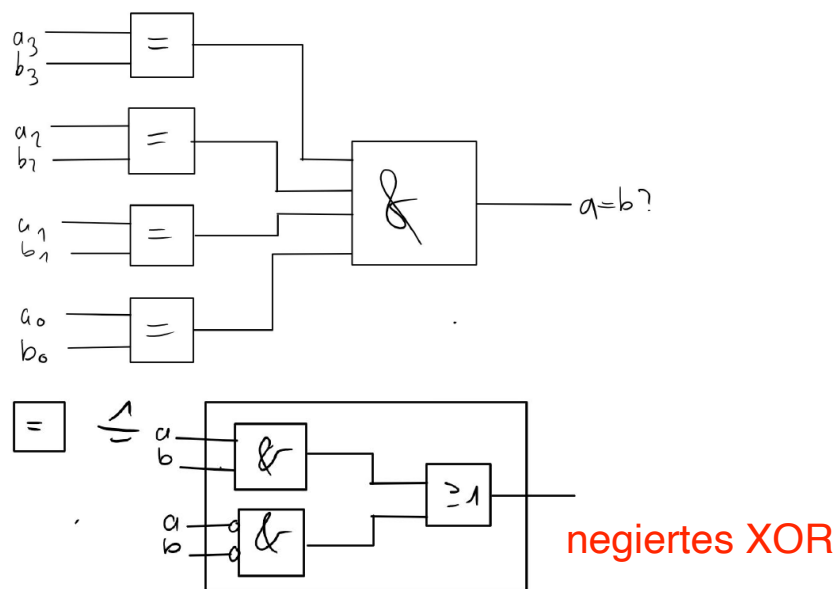


Abbildung 5.5: Aufbau des Gleichheits-Komparators im Cache

### Komparator

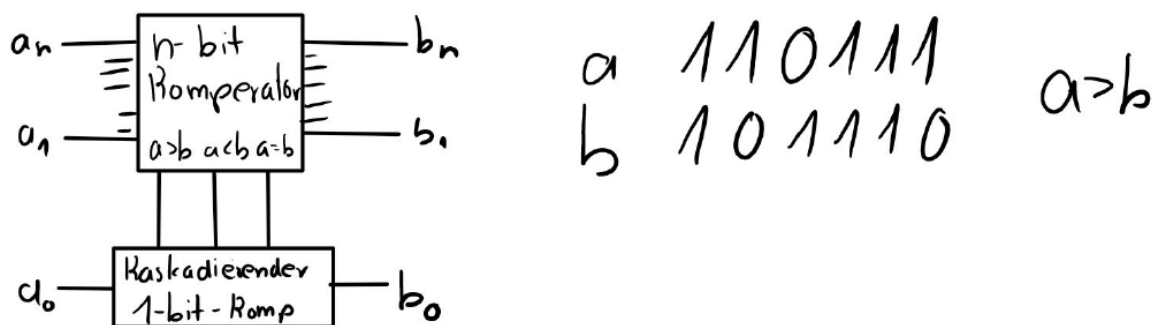


Abbildung 5.6: Aufbau des  $n$ -Bit-Komparator aus kaskadierbaren 1-Bit-Komparatoren

Aus **Abbildung 5.6** ergibt sich folgende Tabelle:

$b_n$	$a_n$	$(a > b)_{in}$	$(a = b)_{in}$	$(a < b)_{in}$	$(a > b)_{out}$	$(a = b)_{out}$	$(a < b)_{out}$
0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	1	0	0
0	1	x	x	x	1	0	0
1	0	x	x	x	0	0	1
1	1	0	0	1	0	0	1
1	1	0	1	0	0	1	0
1	1	1	0	0	1	0	0

Somit ergibt sich als Schaltnetz für einen 4-Bit-Komparator die [Abbildung 5.7](#)

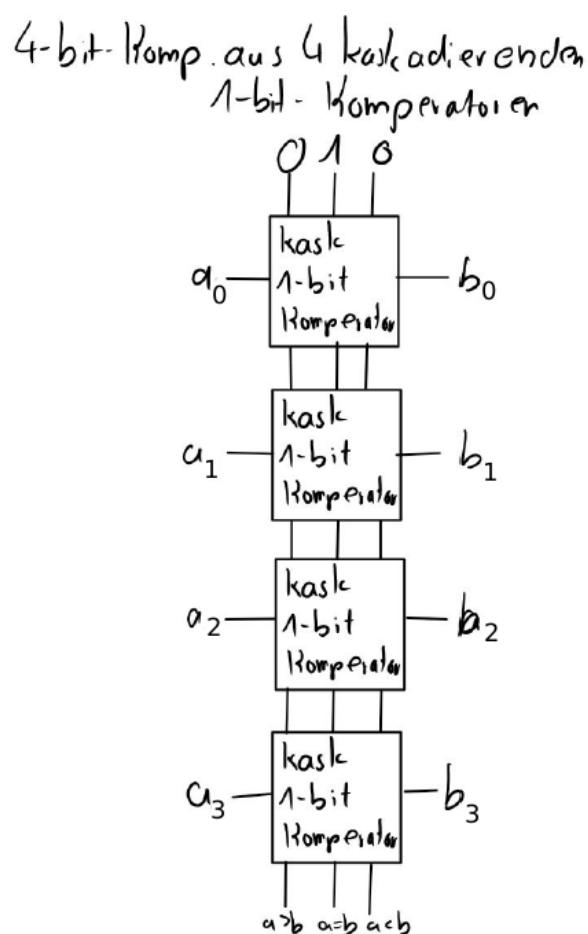


Abbildung 5.7: kaskadierbarer 4-Bit-Komparator

[HW](#): Aufwand für einen kaskadierbaren 1-Bit-Komparator

- HW: 30 Transistoren (8 ODER + 14 UND + 8 NOT)
- Zeit: 2 [GLZ](#)

Im Cache reicht der Vergleich auf Gleichheit aller Ziffern parallel:

- 2  $n$ -Bit-Zahlen:  $n$  Äquivalenzgatter und 1 UND mit  $n$  Eingängen. (vergleiche [Abbildung 5.5](#))



⇒  $7n$  Transistoren **HW**-Aufwand

⇒ 3 **GLZ** Zeitaufwand

### 5.3.8 Direct-Mapped Cache

Beim Direct-Mapped-Cache (**DMC**) gibt es für jede **HSS** nur/genau eine **CL**, in welche diese **HSS** eingelagert werden kann.

⇒ es ist nur ein Komparator nötig

Es ist eine Hash-Funktion notwendig, welche die gekürzte **HSA** auf die **CL**-Nummer abbildet. Die einfachste Hash-Funktion ist „Modulo“ (Rest einer Ganzzahldivision).

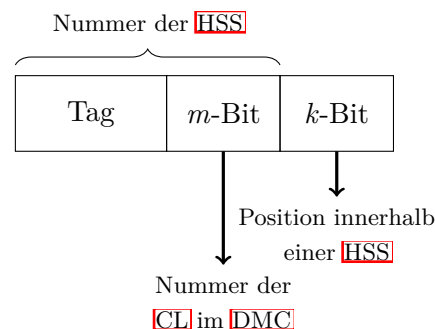


Abbildung 5.8: Direct-Mapped-Cache-Line

Modulo ist besonders einfach, falls der Divisor eine Zweierpotenz (z. B.  $2^m$ ) an Bits ist, denn dann stellen die niederwertigsten  $m$  Bit den gesuchten Rest dar! Nachteil ist, dass dadurch nur Zweierpotenzen an **CLs** möglich sind.

Viele Paare von **HSS** können nicht gleichzeitig im Cache gehalten werden (bei gleichem Ergebnis der Hash-Funktion). Eine mögliche Problemlösung: Ausweichfunktion (wird aus Zeitgründen beim Cache nicht verwendet).

**Abhilfe:**  $n$ -Wege-Assoziativ-Cache (**nWAC**)

### 5.3.9 $n$ -Wege-Assoziativ-Cache

Für jede **HSS** gibt es genau  $n$  Cache Lines, in welche die **HSS** eingelagert werden kann. Die Realisierung findet über  $n$  **DMC** statt, welche alle jeweils gleich aufgebaut sind.

### 5.3.10 Kollision

Falls beim Einlagern einer Hauptspeicherseite in den Cache bereits alle für diese **HSS** in Frage kommenden Cache Lines belegt sind, handelt es sich um eine Kollision.

Eine Kollision kann frühestens auftreten:

- beim **VAC** bei vollem (heißen) Cache
- beim **DMC** beim zweiten Zugriff
- beim  $n$ -Wege-**AC** beim  $(n + 1)$ -ten Zugriff

Wie groß ist die Kollisionswahrscheinlichkeit?

- **DMC**:  $p_{\text{Kollision beim 2. Zugriff}} = \frac{1}{\text{Anzahl CL}} = \frac{1}{2^m}$
- $n$ -Wege-**AC**:  $p_{\text{Kollision beim } n+1 \text{ Zugriff}} = \left(\frac{1}{2^m}\right)^n = \left(\frac{1}{2^{mn}}\right)$

Beispiel: Vergleich **DMC** mit 2-Wege-**AC**

- **DMC**: Anzahl **CL** = 16  $\Rightarrow m = 4 : p_{\text{Kollision 2. Zugriff}} = \frac{1}{2^4} = 0,0625 = 6,25\%$
- 2-Wege-**AC**: Anzahl **CL** je 8  $\Rightarrow m = 3 : p_{\text{Kollision 3. Zugriff}} = \frac{1}{2^{3 \cdot 2}} = \frac{1}{2^6} = 1,5625$

### 5.3.11 Verdrängung

Wenn eine **HSS** in den Cache eingelagert werden soll, muss eine andere aus dem Cache entfernt werden. Eine Kollision ist Voraussetzung für Verdrängung. Mit einer Verdrängungsstrategie wird darüber entschieden, welche der möglichen **HSS** verdrängt wird. In **Unterabschnitt 5.3.13** wird auf die Verdrängungsstrategie eingegangen.

#### Hinweis

Eine Verdrängungsstrategie ist nur notwendig beim **VAC** und beim  $n$ -Wege-**AC**. Beim **DMC** braucht man *keine* Verdrängungsstrategie!

### 5.3.12 Adressrechnen mit Cache

- 2-Wege-**AC**:  $n = 2$  Der Tag ist  $12 - 4 - 3 = 5$  Bit groß,
- 2 **DMC** mit jeweils 8 **CL**:  $m = 3$  siehe **Abbildung 5.9**
- jede **CL** beinhaltet 16 Worte:  $k = 4$
- **HS** beinhaltet 4096 **HSA**:  $2^{12}$  Speicherwerte

#### Hinweis

Die Größe eines Speicherwertes bzw. Wortes in Bit wird nicht definiert und ist auch unerheblich für die folgenden Überlegungen

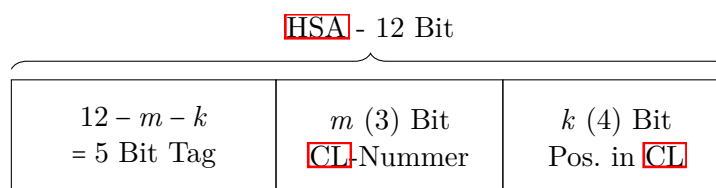


Abbildung 5.9: Cache-Line mit 5-Bit Tag

**Abbildung 5.10** zeigt Cache A (links) und Cache B (rechts), mit denen im folgenden gerechnet wird. Die angefragten Hauptspeicheradressen müssen auf 12-Bit erweitert werden, denn hier ist eine Hauptspeicheradresse 12-Bit groß, wie in **Abbildung 5.9** zu sehen ist.

angefragte **HSA**:  $42_{10} = 101010_2$

angefragte **HSA**:  $0815_{10} = 1100101111_2$

$\underbrace{0\ 0\ 0\ 0\ 0\ 0}_{\text{Tag}} \underbrace{1\ 0\ 1}_{\text{CL-Nr (2)}} \underbrace{0\ 1\ 0}_{\text{Pos. in CL (10)}}$

$\underbrace{0\ 0\ 1\ 1\ 0\ 0}_{\text{Tag}} \underbrace{0\ 1\ 0}_{\text{CL-Nr. (2)}} \underbrace{1\ 1\ 1\ 1}_{\text{Pos. in CL (15)}}$

Cache A **CL**-Nr. 2 ist nicht valide

Cache-A **CL**-Nr. 2 ist valide, aber falscher Tag.

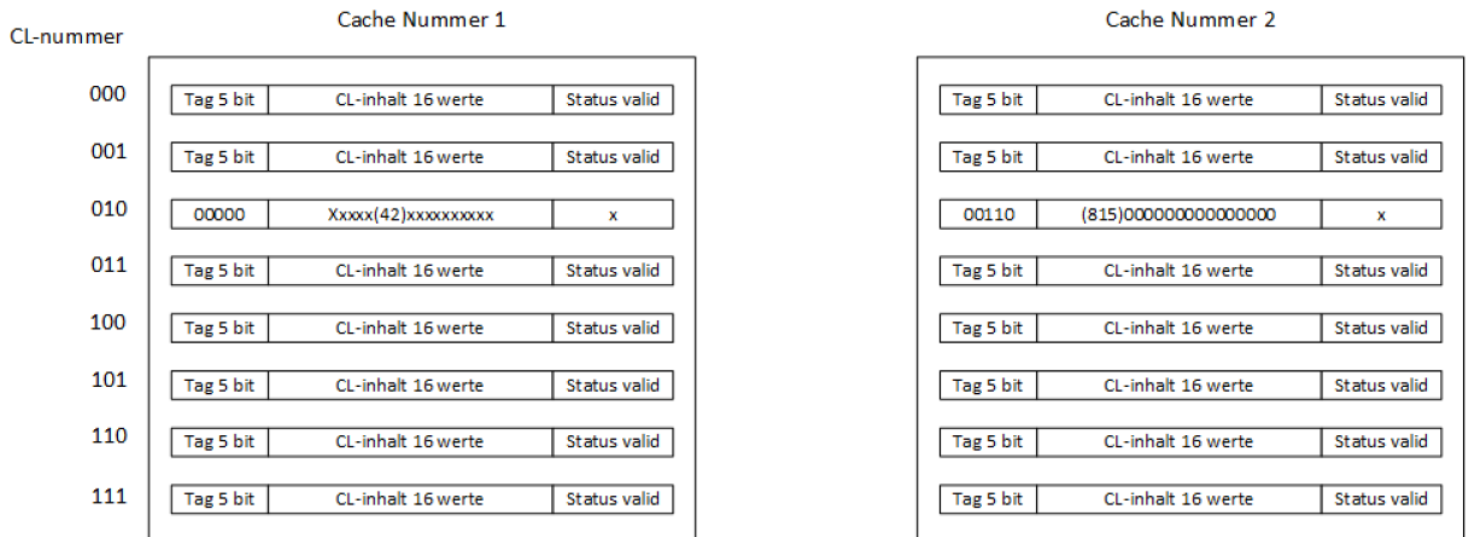


Abbildung 5.10: 2-Wege-AC - Beispiel für Adressrechnen

Cache B CL-Nr. 2 ist nicht valide

⇒ ein Miss

⇒ Einlagern der HSS (von HSA 32 bis 47)  
in CL-Nr. 2

Cache-B CL-Nr. 2 ist nicht valide.

⇒ insgesamt ein Miss

⇒ Einlagern der HSS (von HSA 800 bis 815)  
in CL-Nr. 2 in Cache B  
(Cache A nicht möglich)

angefragte HSA: 0271<sub>10</sub>

$$\underbrace{00010}_{\text{Tag}} \underbrace{000}_{\text{CL-Nr. 0}} \underbrace{1111}_{\text{Pos. in CL (15)}}$$

Cache A CL-Nr. 0 ist nicht valide

Cache B CL-Nr. 0 ist nicht valide

⇒ Miss

⇒ Einlagern der HSS von (HSA 256 bis 271)  
in CL-Nr. 0

angefragte HSA: 0675<sub>10</sub>

$$\underbrace{00101}_{\text{Tag}} \underbrace{010}_{\text{CL-Nr. 2}} \underbrace{0011}_{\text{Pos. in CL (3)}}$$

Tag in CL-Nr. 2 von Cache A und Cache B  
sind verschieden vom angefragten Tag.

⇒ Miss ⇒ sogar Kollision

⇒ Verdrängung notwendig

angefragte HSA: 37<sub>10</sub>

$$\underbrace{00000}_{\text{Tag}} \underbrace{010}_{\text{CL-Nr. 2}} \underbrace{0101}_{\text{Pos. in CL (5)}}$$

CL-Nr. 2 in Cache A ist valide, der Tag stimmt  
überein, also

⇒ Hit in Cache A CL-Nr. 2

### 5.3.13 Verdrängungsstrategie

#### Random

Die zu verdrängende Seite wird zufällig aus den möglichen **HSS** ausgewählt.

*Bewertung:*

Erwartete Hit-Rate =  $\frac{\text{Größe (Cache)}}{\text{Größe (HS)}}$  (bei zufällig verteilten Zugriffen, also ziemlich schlecht)

⇒ nur für Benchmark-Zwecke

notwendiger Aufwand: Zufallszahlengenerator

⇒ echter Zufall ist sehr teuer & aufwändig.

⇒ für Benchmarks reichen (meist) Pseudozufallszahlen aus

#### Optimale Strategie

Es wird die Hauptspeicherseite verdrängt, welche in der Zukunft gar nicht mehr oder am längsten nicht mehr gebraucht wird.

*Bewertung:*

Erwartete Hit-Rate = „systembedingtes Maximum“

*Aufwand:*

„Blick in die Zukunft“ bzw. „Kristallkugel“ ⇒ nicht möglich!

*Realisierung für Benchmarking:*

Zweimaliger Durchlauf für genau dieselben Parameter. Der erste Durchlauf für Logfile und zweiter Durchlauf mit optimaler Strategie anhand des Logfiles.

#### First-In-First-Out (FIFO)

Bei der First-In-First-Out (**FIFO**)-Strategie wird die **HSS**, welche sich am längsten im Cache befindet, verdrängt (klassische Warteschlangenbedienstrategie).

*Aufwand:*

Timestamp in jeder Cache Line. Bei Verdrängung: Suche nach dem Minimum der Timestamps (sehr aufwändig!).

*Besser:*

Verwaltung der **CLs** als einfach verwaltete Liste, d. h. Zeiger auf den Nachfolger in jeder **CL**. Globalen Zeiger auf den ersten und letzten Eintrag für Verdrängung und Einlagerung.

*Schlecht unterstützte, aber häufige Zugriffsmuster:*

Ständig genutzte Datenstücke werden genauso schnell verdrängt wie Daten, die nur ein einziges Mal gebraucht werden. Anders gesagt: Daten, die lange nicht benötigt wurden, werden auch nicht

schneller verdrängt, als Daten, die genauso lange im Cache sind, aber erst kürzlich gebraucht wurden.

### Least-Recently-Used (LRU)

Bei der Least-Recently-Used (**LRU**) Strategie wird die Hauptspeicherseite, auf welche am längsten nicht zugegriffen wurde, verdrängt.

*Aufwand:*

Timestamp mit Update bei jedem Zugriff  $\Rightarrow$  Die Suche bei Verdrängung ist zu aufwändig!

*Besser:*

Verwaltung als *doppelt* verkettete Liste, d. h. Zeiger auf Vorgänger *und* Nachfolger in jeder **CL**. Globalen Zeiger auf ersten und letzten Eintrag.

*Schlecht unterstützte Zugriffsmuster:*

Häufigkeit der Zugriffe wird nicht berücksichtigt, d. h. vielfach genutzte Hauptspeicherseiten werden genauso verdrängt wie **HSS** mit nur einem Zugriff)

### Least-Frequently-Used (LFU)

Bei der Least-Frequently-Used (**LFU**) Strategie wird die Hauptspeicherseite verdrängt, welche bisher am seltensten („am wenigsten häufig“) verwendet wurde.

*Aufwand (für Statusinfo):*

$$\text{Häufigkeit} = \frac{\text{Zugriffe}}{\text{Zeit}}$$

- Benutzungszähler
- Einlagerungszeit

$\Rightarrow$  Bei Verdrängung aufwändige Berechnung und Suche

*Problem (Zugriffsmuster):*

Neu eingelagerte Seiten werden schnell wieder verdrängt, wenn sich der Zugriffszähler am Anfang nicht schnell genug erhöht und andere etablierte Seiten eine höhere Häufigkeit aufgrund vieler „alter“ Zugriffe aufweisen.

*Lösungsansatz:*

Zugriffe müssen „altern“, d. h. alte Zugriffe werden weniger stark gewichtet als neue Zugriffe.

*Mögliche Implementierung:*

Mehrere Zugriffszähler in jeder **CL** für die letzten  $i$  Zeitscheiben. Dividieren ist damit überflüssig (alle Zeitscheiben sind gleich lang): Addition der mit  $2^j$  gewichteten Zähler als „Häufigkeit“ ( $j = i - 1$  für jüngste Zeitscheiben,  $j = 0$  für älteste Zeitscheibe).

*Weiterhin:*

Bei der Verdrängung gibt es ein Problem bei der Suche nach der niedrigsten „Häufigkeit“.

$\Rightarrow$  evtl. Lösung über eine bei Beginn jeden Zeitslots neu aufzubauende verkettete Liste.

Hier gibt es viel Platz für Optimierungen der **CPU**-Hersteller.

Klausur:

- 60 Minuten
- 50 Punkte
- Stichworte reichen aus (nicht rot), Bleistift erlaubt bei Schaltnetzen usw
- Aufgaben
- siehe Wiederholungsklausur weiter unten (2016), erste Aufgabe identisch
- ab Seite 41

### 5.3.14 Cache bei Mehrprozessor-/Mehrkernsystemen

In **Abbildung 5.11** besitzt jeder Kern einen eigenen Cache. Möglich wäre auch ein gemeinsamer Cache. Verglichen werden diese beiden Möglichkeiten in **Tabelle 5.2**.

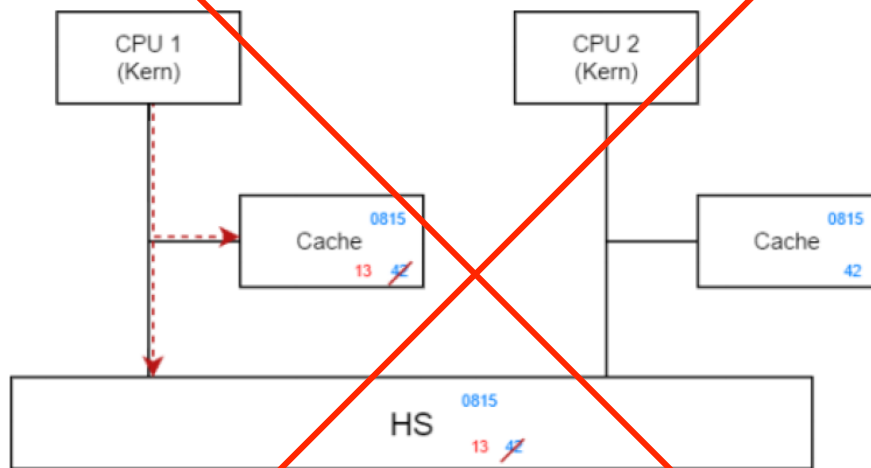


Abbildung 5.11: Mehrprozessor-/Mehrkernsystem

gemeinsamer Cache	getrennter Cache
⊕ dieselben Daten für beide Kerne benötigt/-bedeutet nur einmaliges Einlagern im Cache (Platzvorteil).	⊖ Dieselben Daten müssen ggf. mehrfach in den Caches gehalten werden.
⊖ komplexer Bus mit Zugriffsprotokoll (Zeitverlust zumindest bei Zugriffskollision)	⊕ einfache 1:1-Verbindungen (kein komplexes Zugriffsprotokoll)
	⊖ keine Konsistenz trotz Write-Through Strategie gewährleistet

Tabelle 5.2: Vergleich von getrennten/gemeinsamen Caches bei Mehrkernsystemen

Abhilfe für Inkonsistenzen bei mehreren Caches und Mehrprozessorsystemen:

- gemeinsamer Cache (ungünstige Performance aufgrund eines Bus)
- Snoop-Logik, d. h. jeder Cache schnüffelt bei den anderen Caches bzw. beim **HS**, welche Daten geschrieben wurden und invalidiert diese ggf. im eigenen Cache.

Tatsächlich haben moderne **CPUs** beim L1-Cache getrennte Caches für jeden Kern und beim L3-Cache einen gemeinsamen Cache. Ob eine Snoop-Logik verwendet wird, ist davon abhängig, ob es für das Level notwendig ist oder nicht.

## 5.4 Hauptspeicher-Organisation

Ein Hauptspeicher-Wort wird über eine Hauptspeicheradresse (binäre, ganze, nicht negative Zahl) angesprochen. Jedes Hauptspeicher-Wort hat eine feste Wortbreite (im besten Fall gleich der **CPU**-Wortbreite, also 64-Bit).

Im folgenden gehen wir von 1-Bit-Worten aus:

$$0 \leq \text{HSA} \leq (\text{HS-Größe in Worten}) - 1$$

Ein 1-Bit-Kondensatorspeicher wird in **Abbildung 5.12** gezeigt. Für jedes Hauptspeicher-Wort gibt es eine Select-Leitung. Für jedes Bit des Wortes gibt es einen Kondensator und einen Transistor. Diese werden dann parallel geschaltet.

Mehrere 1-Bit-Worte hängen an der selben Datenleitung, werden aber über unterschiedliche Select-Leitungen angesteuert.

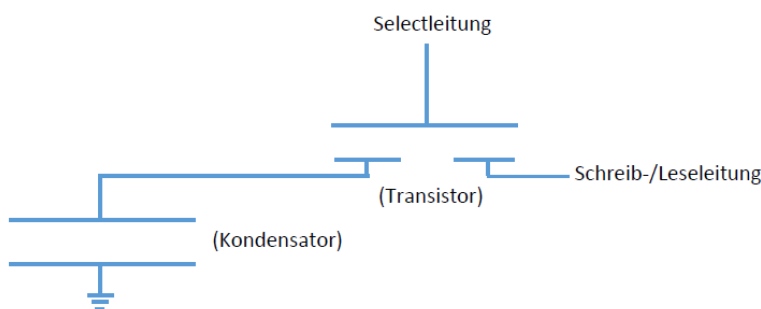


Abbildung 5.12: Hauptspeicher – 1-Bit-Kondensatorspeicher

Um aus der  $n$ -Bit-**HSA**  $2^n$  Select-Leitungen zu erzeugen, welche die Bits abfragen, ist ein  $n : 2^n$ -Decoder notwendig.

In **Abbildung 5.13** wird ein  $4 : 16$ -Decoder mit entsprechender Wertetabelle gezeigt.

Es müssen alle Minterme gebildet werden. Die **DNF** ist für jedes  $s_i$  gleichzeitig die **DMF** (da sie aus nur einem Minterm besteht).

Somit besteht der  $n : 2^n$  Decoder aus genau  $2^n$  Mintermen mit jeweils  $n$  Literalen.

**Zeitaufwand:** 1 **GLZ** ( $O(1)$ ) und damit perfekt

**Hardwareaufwand:**  $n \cdot 2^n$  Transistoren

**Hardwareaufwand für Speicher und Decoder:**

Am Beispiel vom Apple II/C64/...: 16 Bit **HSA**, 8 Bit Worte

$\Rightarrow 2^{16}$  8-Bit-Worte: 65536 Byte = 512 kBit Speicher

$\Rightarrow$  für Speicher: 1 Mio. Bauelemente

für  $16 : 2^{16}$  Decoder:  $16 \cdot 2^{16}$  Transistoren  $\approx$  1 Mio. Transistoren

$\Rightarrow$  der Decoder hat denselben Aufwand wie der Speicher!



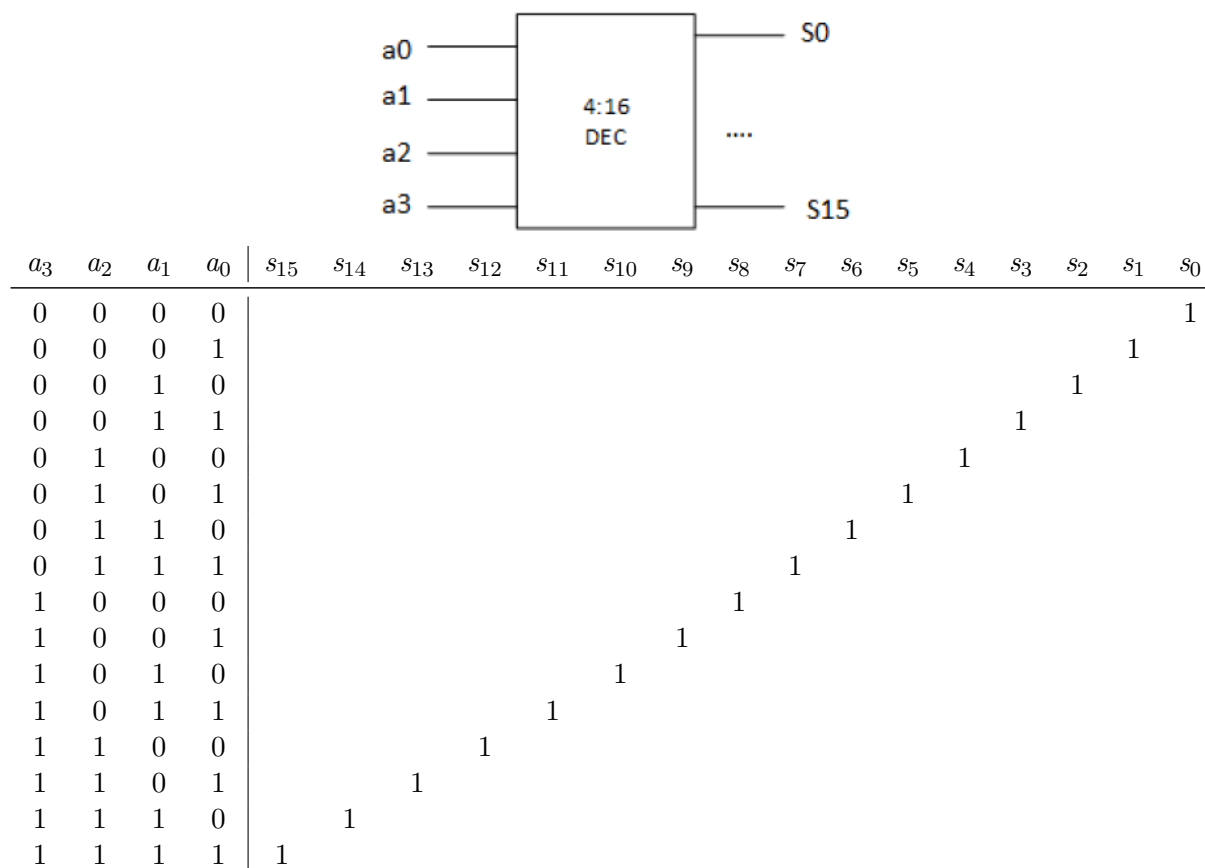


Abbildung 5.13: Hauptspeicher – 1-Bit-Kondensatorspeicher

**PC** mit 32 Bit **CPU**:

32 Bit = 4 Byte Hauptspeicher-Wortgröße

32 Bit-**HSA**:  $2^{32}$  **HS**-Worte  $\approx$  4 Mrd. **HS**-Worte

### Tipp

Intel-**CPU**s adressieren nach wie vor jedes einzelne Byte und nicht Hauptspeicher-Worte der Größe 4 oder 8 Byte.

Eine Ideale 32 Bit **CPU** hätte  $2^{32}$  Werte je 32 Bit:

$\Rightarrow 2^{32} \cdot 32 \text{ Bit} \approx 128 \text{ Mrd. Bit Speicher}$

$\Rightarrow 256 \text{ Mrd. Bauelemente Hardwareaufwand für Hauptspeicher}$

$\Rightarrow \text{Hardwareaufwand für einen } 32 : 2^{32} \text{ Decoder: } 32 \cdot 2^{32} \text{ Tr.} = 128 \text{ Mrd. Transistoren.}$

Ideale 64 Bit **CPU** hätte  $2^{64}$  Werte je 64 Bit:

$\Rightarrow 2^{64} \cdot 64 \text{ Bit} = 2^{70} \approx 1 \text{ Trilliarde Bit}$

$\Rightarrow 2 \text{ Trilliarden Bauelemente Hardwareaufwand für HS}$

$\Rightarrow 64 : 2^{64} \text{ Decoder: } 64 \cdot 2^{64} \text{ Transistoren} \approx 1 \text{ Trilliarde Transistoren}$

In jedem Fall ist der Hardwareaufwand für den Decoder in derselben Größenordnung wie für den eigentlichen Hauptspeicher und damit sehr (zu) groß!

*Abhilfe:* matrixförmige Speicherorganisation

## 5.5 Matrixförmige Speicherorganisation

Eine matrixförmige Speicherorganisation: zweidimensionale Anordnung der Speicherwerte in Zeilen und Spalten, siehe [Abbildung 5.14](#) auf [Seite 38](#).

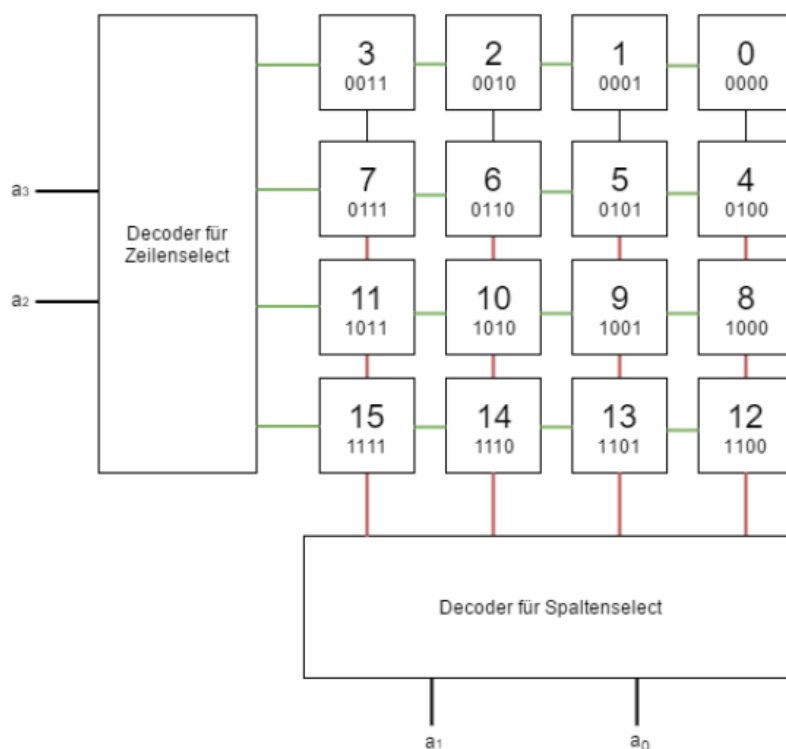


Abbildung 5.14: Matrixdecoder

$a_3 a_2$  gibt die Zeilennummer an und  $a_1 a_0$  die Spaltennummer. Somit reicht es statt eines 4:16-Decoders einen 2:4 Zeilen- und einen 2:4 Spalten-Decoder zu verwenden.

### Aufwand

Anstatt wie bei einem 4 : 16-Decoder mit 64 Transistoren beträgt der Aufwand bei zwei 2 : 4-Decodern mit  $2 \cdot 8$  Transistoren = 16 Transistoren.

64-Bit [HSA](#): als  $64 : 2^{64}$  Decoder:  $2^{70}$  Transistoren

als Matrix mit  $2^{32}$  Zeilen und  $2^{32}$  Spalten:

- $32 : 2^{32}$  Decoder mit jeweils  $32 \cdot 2^{32}$  Transistoren =  $2^{37}$  Transistoren
- 2 Stück:  $2^{38}$  Transistoren: 256 Mrd. Transistoren

In linearer Organisation betrüge der Decoder-Aufwand das 4-Mrd.-fache! Damit ergibt sich eine große Einsparung beim Decoder-Aufwand!

Aber: für die Realisierung des 2. Select-Eingangs wird ein weiterer Transistor je Bit benötigt  
 $\Rightarrow$  dieselbe Größenordnung Zusatzaufwand wie Einsparung?

Statt eines Spalten-Decoders werden die Datenleitungen aller Speicherbits einer Spalte zusammengeschaltet (als jeweils eine Spalten-Datenleitung). Von diesen wird aber über einen Spalten-Multiplexer ein Spalten-Daten-Signal ausgewählt. Dies wird in [Abbildung 5.15](#) dargestellt.

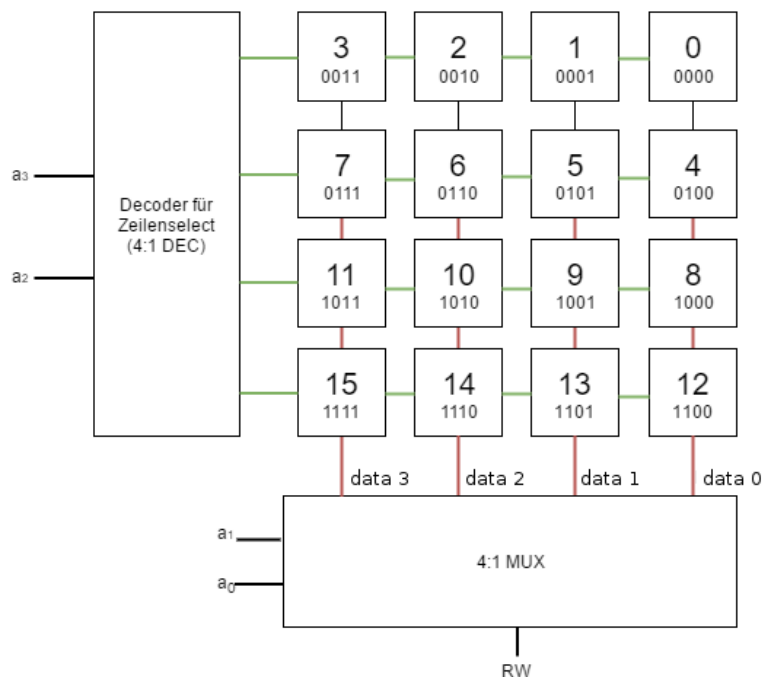


Abbildung 5.15: Matrixmultiplexer

$2^n : 1$ -MUX Baustein, welcher aus  $2^n$ -Dateneingängen anhand einer  $n$ -stelligen Adresse eine Datenleitung auswählt und ausgibt.

$2^4 : 1$ -MUX Hat 4 Adresseingänge und  $2^4 = 16$  Dateneingänge

Realisierung über  $2 : 4$  Decoder sowie 4 UND mit jeweils 2 Eingängen und ein ODER mit 4 Eingängen.

$2^n : 1$  MUX:

ein  $n : 2^n$  Decoder

$2^n$  UND mit jeweils 2 Eingängen

ein ODER mit  $2^n$  Eingängen

Gesamtaufwand:  $n \cdot 2^n + 2 \cdot 2^n + 2^n$  Transistoren =  $(n + 3) \cdot 2^n$  Transistoren

Insgesamt braucht man für einen kleinen Decoder und einem kleinen Multiplexer immer noch deutlich weniger Hardwareaufwand wie für einen großen Decoder.

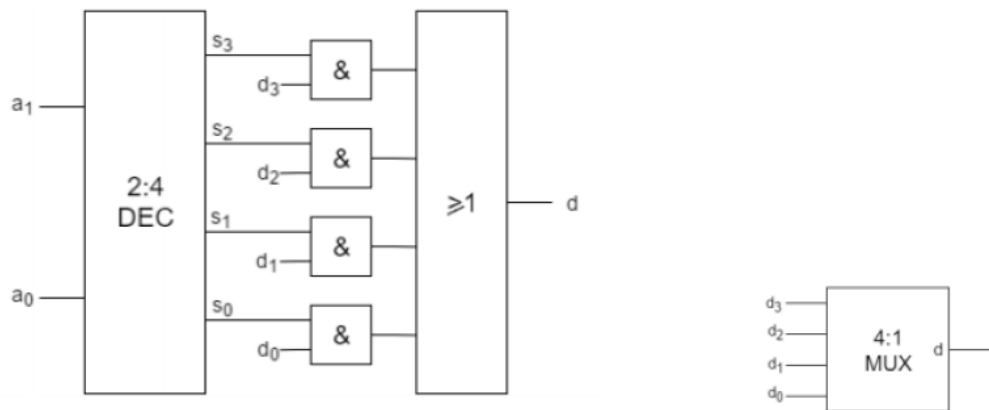


Abbildung 5.16: Schaltnetz eines 4 : 1-MUX

Vor den Multiplexer kann ein Cache geschaltet werden, sodass eine ganze **HSS** (=Matrixzeile) eingelesen werden kann.

### 5.5.1 Gründe für Matrixorganisation

1. weniger Aufwand für Adressdecodierung
2. Einlesen einer ganzen **HSS** (=Matrixzeile) in den Cache
3. zeilenweiser Refresh des **HSS** (wortweiser Refresh-Zyklus dauert viel zu lange)

## 6 Wiederholungsklausur 2016

### 6.1 Aufgabe 1 - 8 Punkte (6/2)

- a) Für den Aufbau von Rechnern lassen sich zwei grundsätzlich unterschiedliche Architekturen verwenden. Skizzieren Sie diese. Erläutern Sie kurz die einzelnen Komponenten und gehen Sie insbesondere auf den Unterschied der beiden Architekturen bezüglich Aufbau und auf ihre jeweiligen Vor- und Nachteile ein. **auch 2022 (nur anders formuliert)**
- b) Nennen Sie konkrete Anwendungsfälle für den Einsatz dieser beiden Architekturen in heutigen „Rechnern“. **auch 2022**

### 6.2 Aufgabe 2 - 10 Punkte (2/4/4)

- a) Zum Addieren von Zahlen lassen sich Halbaddierer und Volladdierer verwenden. Erläutern Sie den Unterschied zwischen einem Halb- und einem Volladdierer und gehen Sie dabei insbesondere auf Anzahl und Bedeutung der jeweiligen Ein- und Ausgänge ein. **same 2022**
- b) Aus mehreren Halb- und Volladdierern lassen sich Paralleladdierer in zwei verschiedenen Bauarten zusammenbauen. Skizzieren Sie den grundsätzlichen Aufbau eines 4-Bit-Paralleladdierers und erläutern Sie den Unterschied im Aufbau zwischen den beiden Varianten.  
**ODER für mehrstellige Multiplikationen einen Multiplizierer bauen (aus mehreren (egal welcher)) Varianten: Ein Addierer malen (2016), Multiplizierer (mult zwei 4 Bit Multiplizierer) => malen**

#### Hinweis

Es ist nur *ein* Paralleladdierer zu zeichnen. Es muss dann nur auf den Unterschied der beiden Varianten eingegangen werden. Der **RC-PA** ist in dieser Aufgabe zu empfehlen.

- c) Die Varianten aus Teilaufgabe b) unterscheiden sich insbesondere im Bezug auf ihren Aufwand. Was kann „Aufwand“ bedeuten? Erläutern Sie diese im Unterschied beim Aufwand. Geben Sie an, ab wann Sie welche Variante des Paralleladdierers einsetzen würden.

### 6.3 Aufgabe 3 - 12 Punkte (6/6)

- a) Der Hauptspeicher eines Computers ist logisch linear aber physikalisch als Matrix organisiert. Was bedeutet das? Nennen Sie Gründe für diese Matrixorganisation.
- b) Zur Realisierung dieser Hauptspeicherorganisation benötigt man verschiedene Bauelemente (Schaltnetze/Schaltwerke). Nennen Sie die drei wichtigsten Bauteile zur Realisierung dieser Matrixorganisation und beschreiben Sie ihr jeweiliges Verhalten, also ihre Funktionalität.

**6.4 Aufgabe 4 - 20 Punkte (4/6/6/4)**

- a) Was ist ein Cache? Welche Eigenschaften hat ein Cache und welche Eigenschaften sollte dieser außerdem haben? Erläutern Sie die Eigenschaften kurz.
- b) Was versteht man bei einem Cache als „Verdrängung“? Wann und warum tritt Verdrängung auf. Erläutern Sie die beiden Verdrängungsstrategien „Optimal“ und „**LFU**“. Gehen Sie insbesondere auf den Aufwand und die Realisierung ein.
- c) Bei Caches werden unter anderem Vollasoziativ, Direct-Mapped und  $n$ -Wege Cache unterschieden. Erläutern Sie den Unterschied dieser drei Architekturen. Welche davon stehen in Zusammenhang mit den in Teilaufgabe b) genannten Verdrängungsstrategien?
- d) Innerhalb eines Caches benötigt man einen oder mehrere Komparatoren. Beschreiben Sie um was es sich bei dem Bauteil „Komparator“ innerhalb des Caches handelt und skizzieren Sie seine Realisierung als Schaltnetz oder Schaltwerk.

**Hinweis**

Für den Komparator reicht das Schaltsymbol für Äquivalenz nicht aus! Es muss das Schaltnetz bestehend aus (NICHT-) UND und ODER gezeichnet werden.

## 7 Abkürzungsverzeichnis

<b>AC</b>	Assoziativ-Cache	
<b>BLW</b>	Bandlaufwerk	
<b>CISC</b>	Complex Instruction Set Computer .....	22
<b>CL</b>	Cache Line .....	26
<b>CLA-PA</b>	Carry-Look-Ahead-Paralleladdierer .....	19
<b>CPU</b>	Central Processing Unit	
<b>D-FF</b>	D-Flip-Flop	
<b>DMC</b>	Direct-Mapped-Cache .....	30
<b>DMF</b>	Disjunktive Minimalform .....	8
<b>DNF</b>	Disjunktive Normalform .....	8
<b>ENIAC</b>	Electronic Numerical Integrator and Computer .....	3
<b>IC</b>	Integrated Circuit	
<b>FIFO</b>	First-In-First-Out .....	33
<b>GLZ</b>	Gatterlaufzeit .....	8
<b>HA</b>	Halbaddierer	
<b>HS</b>	Hauptspeicher	
<b>HSA</b>	Hauptspeicheradresse .....	24
<b>HSS</b>	Hauptspeicherseite .....	26
<b>HW</b>	Hardware	
<b>LFU</b>	Least-Frequently-Used .....	34
<b>LRU</b>	Least-Recently-Used .....	34
<b>LW</b>	Laufwerk	
<b>PC</b>	Personal Computer	
<b>PA</b>	Paralleladdierer	
<b>PM</b>	Parallelmultiplizierer .....	18
<b>RC-PA</b>	Ripple-Carry-Paralleladdierer	
<b>RISC</b>	Reduced Instruction Set Computer .....	22
<b>SA</b>	Serielladdierer .....	14
<b>SM</b>	Seriellmultiplizierer	
<b>SR</b>	Schieberegister .....	14
<b>UNIVAC</b>	Universal Automatic Computer	
<b>VA</b>	Volladdierer .....	14
<b>VAC</b>	Vollassoziativer Cache	
<b>nWAC</b>	$n$ -Wege-Assoziativ-Cache .....	30

## Abbildungsverzeichnis

3.1 Funktionsweise Kathodenstrahlröhre [Quelle: Wikipedia]	4
3.2 Vereinfachte Darstellung der von-Neumann-Architektur	5
3.3 Vereinfachte Darstellung der Harvard-Architektur	6
4.1 Halbaddierer – Schaltnetz und Schaltsymbol	9
4.2 Volladdierer – Schaltnetz und Schaltsymbol	9
4.3 RC-Paralleladdierer – Schaltnetz und Schaltsymbol	10
4.4 Carry-Look-Ahead-Paralleladdierer – „magisches“ Schaltnetz	11
4.5 Kaskadierbarer 4-Bit-CLA-PA	13
4.6 Serielladdierer	14
4.7 Schieberegister aus D-FF mit Hardwareaufwand	15
4.8 Serielladdierer mit Hardwareoptimierung	15
4.9 Schaltnetz Subtrahierer	16
4.10 5-Bit Parallelmultiplizierer	18
4.11 5-Bit Seriellmultiplizierer	20
5.1 Look-Through-Cache	25
5.2 Look-Aside-Cache	25
5.3 Aufbau des Caches	27
5.4 $n$ -Bit-Komparator	27
5.5 Aufbau des Gleichheits-Komparators im Cache	28
5.6 Aufbau des $n$ -Bit-Komparator aus kaskadierbaren 1-Bit-Komparatoren	28
5.7 kaskadierbarer 4-Bit-Komparator	29
5.8 Direct-Mapped-Cache-Line	30
5.9 Cache-Line mit 5-Bit Tag	31
5.10 2-Wege-ACI - Beispiel für Adressrechnen	32
5.11 Mehrprozessor-/Mehrkernsystem	35
5.12 Hauptspeicher – 1-Bit-Kondensatorspeicher	36
5.13 Hauptspeicher – 1-Bit-Kondensatorspeicher	37
5.14 Matrixdecoder	38
5.15 Matrixmultiplexer	39
5.16 Schaltnetz eines 4 : 1-MUX	40



## Tabellenverzeichnis

3.1 Vergleich der von-Neumann- und Harvard-Architektur . . . . .	7
4.1 Vergleich des Hardwareaufwand eines RC-PA mit dem verbesserten SA . . . . .	16
4.2 Schriftliche Multiplikation „von links nach rechts“ . . . . .	17
4.3 Schriftliche Multiplikation „von rechts nach links“ . . . . .	17
4.4 Vergleich von RISC und CISC . . . . .	22
5.1 Speicherhierarchie und -Daten . . . . .	23
5.2 Vergleich von getrennten/gemeinsamen Caches bei Mehrkernsystemen . . . . .	35

## **Listingsverzeichnis**

## Stichwortverzeichnis

A		L	
Addition .....	8	Lokalität .....	24
C		M	
Cache		Mehrkernprozessor .....	35
$n$ -Wege-Assoziativ-Cache .....	30	Miss-Rate .....	24
Architektur .....	25	Multiplikation .....	17
Direct-Mapped .....	30	P	
erwärmender Cache .....	25	Paralleladdierer	
heißer Cache .....	25	CLA-PA .....	11
kalter Cache .....	25	RC-PA .....	10
Look-Aside .....	25	R	
Look-Through .....	25	RISC .....	21
Verdrängung .....	31	S	
Verdrängungsstrategie .....	33	Schaltnetz .....	8
Vollasoziativ .....	27	Schaltungsanalyse .....	8
CISC .....	21	Schaltwerk .....	8
D		Serielladdierer .....	14
Division .....	21	Seriellmultiplizierer .....	20
DMF .....	8	Speicher .....	23
DNF .....	8	Subtraktion .....	16
H		V	
Halbaddierer .....	9	Volladdierer .....	9
Harvard-Architektur .....	5	von-Neumann-Architektur .....	4
Hit-Rate .....	24	Z	
K		Zeitaufwand .....	8
Klausur .....	41		