Übersicht:

1 Zahlendarstellung

2 Numerik

3 Codierung

4 Informationstheorie

~~5 Boolsche Algebra~~ (Mathematik)

6 KV und BDD

7 Digitalschaltungen (kombinatorische Logik)

8 Automaten

9 Sequentielle Logik (Speicher-Elemente)

10 Speicher

11 Schaltwerke

12 Schaltwerke: Realisierung

13 Schaltwerke: Moore vs. Mealy

14 Micro 16

15 Befehlssatz

16 Pipelining

17 Speichermanagement

18 Multicore

19 Speichermodelle

01 Zahlendarstellung

Stellenwertsystem

Basis $b \geq 2$ ($b=1$ wäre unär)

Ziffern $[0; b-1]$

Zahlenumwandlung

Quellsystem ⇄ Zielsystem

Binär zu Dezimal

$(110,1101)_2 = (\dots)_{10}$

$1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = 6,8125$

oder im Horner-Schema:

Vorkommastelle:

$((1) \cdot 2 + 1) \cdot 2 + 0 = 6$

Nachkommastelle:

$(((1)/2 + 0)/2 + 1)/2 + 1)/2 = 0,8125$

Dezimal zu Binär

Vorkommastelle:

$a_0 = 6 \bmod 2 = 0$

$a_1 = \underbrace{(6-0)/2}_{=3} \bmod 2 = 1$

$a_2 = \underbrace{(3-1)/2}_{=1} \bmod 2 = 1$

$a_3 = (1-1)/2 \bmod 2 = 0 \longrightarrow$ Abbruch $(110)_2$

effizienter:

$a_0 = 6 \bmod 2 = 0$

$a_1 = \underbrace{\lfloor \frac{6}{2} \rfloor}_{=3} \bmod 2 = 1$

$a_2 = \lfloor \frac{3}{2} \rfloor \bmod 2 = 1$

$a_3 = \lfloor \frac{1}{2} \rfloor \bmod 2 = 0 \rightarrow$ Abbruch $(110)_2$

Nachkommastelle:

effiziente Herleitung:

$a_{-1} = \lfloor 0,8125 \cdot 2 \rfloor = \lfloor 1,625 \rfloor = 1$

$a_{-2} = \lfloor 0,625 \cdot 2 \rfloor = \lfloor 1,25 \rfloor = 1$

$a_{-3} = \lfloor 0,25 \cdot 2 \rfloor = \lfloor 0,5 \rfloor = 0$

$a_{-4} = \lfloor 0,5 \cdot 2 \rfloor = \lfloor 1 \rfloor = 1$

$a_{-5} = \lfloor 0 \cdot 2 \rfloor = \lfloor 0 \rfloor = 0$ → Abbruch $(0,1101)_2$

Binär - Hexadezimal - Beziehung

$2^4 = 16$ [ ][ ][ ][ ] 4er Blöcke zur Umwandlung

Darstellung negativer Zahlen

durch Kodierung

VZ-B: Vorzeichen und Betrag

Erstes Bit

0 positiv

1 negativ

Probleme:

±0

+0, 1, 2, 3, -0, -1, -2, -3

negative Zahlen stehen nach pos

Einer-Komplement

Bitwise Inversion

Probleme:

±0

+0, 1, 2, 3, -3, -2, -1, -0

Durch doppelte 0 entstehen leicht Überläufe, mit +1 korrigieren.

```
  011  (3)
  111  (-0)
 *010  (2)   --(+1)-->  011  (3)
```

Zweier-Komplement

Bitwise Inversion +1 (wenn negativ)

Dadurch eindeutige 0:

0, 1, 2, 3, -4, -3, -2, -1

Exzess

„Symmetrischer Exzess"

Kodierung: + kleinste darstellbare Zahl

$e = 2^{bits-1}$ oder $e = 2^{bits-1} - 1$

„Asymmetrischer Exzess"

Kodierung: + beliebige Zahl

Rechnen mit Exzess:

$e = 12$

$A = -3$

$B = 4$

$A_e = A + e = 9$

$B_e = B + e = 16$

$A_e + B_e = A + B + 2e$

$(A + B)_e = A_e + B_e - e$

02 : Numerik

Kodierung von Zahlen mit Nachkommastelle

Festpunkt

Gesamtlänge = 1 + q + n → Nur $\in \mathbb{Q}$ darstellbar

1 = VZ, q = vorkomma, n = nachkomma

Schrittweite / kleinste darstellbare Zahl

$\Delta x = 2^{-n}$

Gleitpunkt

Idee: je größer die Zahl, desto unwichtiger Δx.

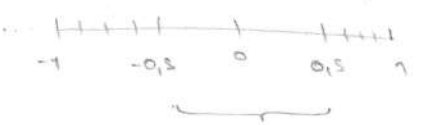
$= VZ \cdot basis^{exponent} \cdot mantisse$

Normalisierte Form

implizites Bit = 1

$\Delta x = x_{min} \cdot b^{e_{min}} = 1 \cdot 2^{e_{min}}$

zB: $b=2$ $e_{min}=-1$



Denormalisierte Form

implizites Bit = 0

„Subnormale Zahlen" lassen sich darstellen.

Problem:
- unerwünschte Lücke
- Null nicht darstellbar, muss kodiert werden

„Gleitpunkt-Darstellungs-Raum" $\mathbb{F}$

$\mathbb{F}(b, p, e_{min}, e_{max}, denorm)$

b ... base ≥ 2

p ... precision (mantisse mit implizitem Bit)

e_{min} / e_{max} ... kleinster / größter Exponent

denorm .... boolean

unit of last position (ulp)

$ulp = (0,000\ldots0001)_b = b^{-p+1}$ ← wegen Vorkommastelle

$\Delta x = ulp \cdot b^{e_{min}} = b^{e_{min}-p+1}$

IEEE 754

Eigene Parameter für IF.

Single : 32 Bit, 8 Exponent Bits, 24 Mantissenbits inkl impliz.

double : 64 Bit, 11 Exponent Bits, 53 Mantissenbits inkl impliz.

Kodierungen im Exponenten

VZ	Exponent	Mantisse	
–	0...000	–	Denormalisiert → e = emin obwohl de emin-1 steht
0	0...000	0	+0
1	0...000	0	-0
0	1...111	>0	NaN
0	1...111	0	+∞
1	1...111	0	-∞

Eigentlicher dekodierter Exponent-Wert: Exzessdarstellung.

Beispiel: IF (2, 11, -14, 15, true) → 16 Bit – 10 expl. Bit – 1 VZ Bit = 5 Bit

base ... 2
precision ... 11 ← inkl. impl.
emin : -14
emax : 15
denorm : true

$$Exzess = 2^{(expon.\ Bits)-1} - 1$$

$$Exzess = 2^{5-1} - 1 = 15 = (01111)_2$$

Dadurch:

$e_{min} - 1 = -15$ (00000)

$e_{max} + 1 = 16$ (11111)

Runden

außerhalb von arithmetischen Operationen

Abschneiden / truncate

$(-1,62\,|\,6)_{10} \longrightarrow \square x = (-1,62)_{10}$

$n = 2$

$(1,10\,|\,1)_2 \longrightarrow \square x = (1,10)_2$

$n = 2$

Gerichtetes Runden / directed rounding

$(-1,62\,|\,6)_{10}$ $\quad n = 2$

Aufrunden $\square x = \max(-1,62; -1,63) = -1,62$

Abrunden $\square x = \min(-1,62; -1,63) = -1,63$

$(1,10\,|\,1)_2$ $\quad n = 2$

Aufrunden $\square x = \max(1,10; 1,11) = 1,11$

Abrunden $\square x = \min(1,10; 1,11) = 1,10$

Optimale Rundung / round to nearest

$\hat{x} = \frac{x_1 + x_2}{2}$ bzw $\hat{x} = (y_1 + y_2) \cdot 0,1$

$(-1,62\,|\,6)_{10}$ $\quad n = 2$ $\longrightarrow \frac{(-1,62) + (-1,63)}{2} = -1,625 \neq \hat{x}$

$\square x = -1,63$

$(1,10\,|\,1)_2$ $\quad n = 2$ $\longrightarrow \frac{1,10 + 1,11}{10} = 1,101 = x$

round to even

$\square x = 1,10$ → es soll 0 stehen

round away from zero

$\square x = 1,11$

Runden mit GRS

→ Bei Addition & Subtraktion als sticky bit rechnen.

guard digit
round digit
sticky bit (bleibt 1 wenn beim Shiften eine 1 durch geht)

Optimale Rundung:

G	R	S	
0	X	X	—
1	X	X	+1
1	0	0	→ Round to even: wenn LSB von Mantisse = 0 — , = 1 +1

Round away from zero: +1

Beispiel: Addition in IEEE 754 (16 Bit)

$A = (5{,}58)_{10}$

$B = (62{,}27)_{10}$

Umrechnung ins Gleitpunkt-Format

1 Bit VZ

5 Bit Exponent ⟶ excess: $2^{5-1} - 1 = 15 = (0\,1111)_2$

10 Bit Mantisse explizit

Normalisierung

$A = (5{,}58)_{10} = (101{,}10010100)_2 \cdot 2^0 = (1{,}0110010100)_2 \cdot 2^2$

$B = (62{,}27)_{10} = (11111{,}0100)_2 \cdot 2^0 = (1{,}1111010000)_2 \cdot 2^5$

Exponenten nach Normalisierung Kodieren

A: $2 = (10)_2$ ⟶ $10 + 01111 = 10001$

B: $5 = (101)_2$ ⟶ $101 + 01111 = 10100$

Exponenten angleichen

$A = m \cdot 2^2$

$B = m \cdot 2^5$ (+3)

Da A<B muss A angeglichen werden:

$10001 + 11 = 10100$

Vorher: 0 | 10001 | 1 | 0110010100

Nachher: 0 | 10100 | 0 | 0010110010 100 → rsh(3)

GRS

implizit

Addieren

		G	R	S
A:	0 | 10100 | 0 1 0 0 1 0 1 1 0 0 1 0	1	0	0
B:	0 | 10100 | 1 1 1 1 1 0 0 1 0 0 0			
	0 | 10100 | 10 | 0 0 0 1 1 1 1 0 1 0	1	0	0

Normalisieren: rsh(1), Exp+1

GRS

0 | 10101 | 1 0 0 0 0 0 1 1 1 1 0 1 010

Runden

0 X X ⟶ unverändert

Fortsetzung: Subtraktion

- ✓ Umrechnung ins Gleitpunkt-Format
- ✓ Normalisierung, Kodierung des Exponenten
- ✓ Angleichung der Exponenten

Subtraktion

$A = (5,58)_{10}$

$B = (62,29)_{10}$

$A - B = -(B - A)$

vereinfacht Rechnen

Ab Sticky Bit

					GRS
	B:	0	10100	1 111100 1000	000
−	A:	0	10100	0 0010 1100 10	100
		①	10100	1 1100 0101 01	100

↑ Ergebnis bereits normalisiert

Runden

...101 100

LSB = 0 −

→ LSB = 1 +1

Lösung:

1 10100 1 1100 0101 10

Vorgang beim Rechnen mit IEEE Darstellung

1) Angleichung der Exponenten
2) Mantissen addieren/subtrahieren
3) Ergebnis normalisieren
4) Runden (Guard Digit, Round Digit, Sticky Bit)

Frage **2**

Falsch

Erreichte Punkte 0,00 von 1,00

Frage markieren

Addieren Sie die Zahlen 0100101010111000 und 1011101000111001.

Es gilt das Gleitpunktformat $\mathbb{F}(2, 11, -14, 15, true)$ IEEE 754-2008 mit half-precision (16 Bit — wie in der Vorlesung).

Verwenden Sie für das Ergebnis ebenfalls dieses Format.

Als Rundungsvorschrift verwenden Sie bitte *"round to nearest — round to even"*.

Antwort: 0011100010101001 ×

Die richtige Antwort ist: 0100101001010100

Angabe:
Addieren Sie die Zahlen 0100101010111000 und 1011101000111001
Als Rundungsvorschrift verwenden Sie bitte "round to nearest – round to even".

A: 0 10010 1010111000
e = 10010 - 01111 = 0011

B: 1 01110 1000111001
e = 01110 - 01111 = -0001

Unterschied zwischen Exponenten: 3 und -1: 4

Exponent von B um 4 erhöhen, Bits um 4 nach rechts shiften:

A: 0 10010 1010111000 | 000
B: 1 10010 0001100011 | 100 < implizites Bit zeigt sich durch shift
eigentlich müsste B kodiert werden weil es jetzt denormiert ist
aber das ist nur ein Zwischenschritt zum addieren

Mantissen addieren:
Da B negativ ist wird B von A abgezogen

A: 1.1010111000 | 000
B: 0.0001100011 | 100 < normalerweise ab „round digit" rechnen aber bei binär auch ab „stick bit"

L: 1.1001010100 | 100

Normalisieren:
Es ist bereits normalisiert

Runden:
1.1001010100 | 100 < weitere Rundungsregel
"round to nearest – round to even"
lsb = 0 → deshalb Mantisse ungändert
L: 1.1001010110

Meine Lösung:
0 10010 1001010100
Korrekte Lösung:
0 10010 1001010100

Frage 3

Falsch

Erreichte Punkte 0,00 von 1,00

Frage markieren

Subtrahieren Sie die Zahl 1101000100001101 von der Zahl 1111010000010010.

Es gilt das Gleitpunktformat $\mathbb{F}(2, 11, -14, 15, true)$ IEEE 754-2008 mit half-precision (16 Bit — wie in der Vorlesung).

Verwenden Sie für das Ergebnis ebenfalls dieses Format.

Als Rundungsvorschrift verwenden Sie bitte *"round to nearest — round to even"*.

Antwort: 111100000100011 ×

Die richtige Antwort ist: 1111010000001111

Angabe:
Subtrahieren Sie die Zahl 1101000100001101 von der Zahl 1111010000010010

A und B sind negativ
A endet mit 0
B endet mit 1

A - B --> wenn beide negativ sind: B - A = - (A - B)

A: 1 11101 0000010010
e = 11101 - 01111 = 1110

B: 1 10100 0100001101
e = 10100 - 01111 = 0101

Unterschied: 14 - 5 = 9

Exponent von B um 9 erhöhen, Bits um 9 nach rechts shiften:
A: 1 11101 0000010010 | 000
B: 1 11101 0000000010 | 101

Mantissen addieren:
A: 1.0000010010 | 000
B: 0.0000000010 | 101

L: 1.0000001111 | 011 (muss negativ sein)

Normalisieren:
ist bereits normalisiert

Runden:
Bei 011 --> unverändert

Meine Lösung:
1 11101 0000001111
Korrekte Lösung:
1 11101 0000001111

Multiplizieren Sie die Zahlen 1001100011111011 und 1100100000010010.

Es gilt das Gleitpunktformat $\mathbb{F}(2, 11, -14, 15, true)$ IEEE 754-2008 mit half-precision (16 Bit — wie in der Vorlesung).

Verwenden Sie für das Ergebnis ebenfalls dieses Format.

Als Rundungsvorschrift verwenden Sie bitte *"round to nearest — round to even"*.

Antwort: ×

Die richtige Antwort ist: 0010010100010001

Angabe:
Multiplizieren Sie die Zahlen 1001100011111011 und 1100100000010010

A: 1 00110 0011111011
e = 00110 - 01111 = -1001
e = 6 - 15 = -9

B: 1 10010 0000010010
e = 10010 - 01111 = 0011
e = 18 - 15 = 3

Neuer Exponent (kodiert): 00110 + 10010 - 01111 = 01001
6 + 18 - 15 = 9
Eigentlicher Wert: 9 - 15 = -9 + 3 = -6 (korrekt kodiert)

Multiplizieren:
1.0011111011 *
1.0000010010 =
1.0100010001 | 011

Runden:
011 --> unverändert

Meine Lösung:
0 01001 0100010001
Korrekte Lösung:
0 01001 0100010001

Angabe:

Gegeben ist die Zahl $(10.011000)_2$.

Runden Sie die Zahl mittels "round to nearest - round away from zero" auf 3 Nachkommastellen.

Antwort: 10.100 ×

Die richtige Antwort ist: 10.011

```
x = 10.011000

x1 = 10.011
x2 = 10.100

x´ = (x1+x2) * 0.1 = 100.111 * 0.1 = 10.0111 --> ungleich mit x
                                                  Lösung: 10.011
```

~~Anlauf 2:~~
Wichtig: Punkt statt Komma benutzen damit es richtig compiled wird

Gegeben ist das folgende Bitmuster: 001011111
Interpretieren Sie dieses als eine Festpunktzahl mit 4 Nachkommastellen und geben sie den entsprechenden dezimalen Wert an.

0 0101.1111
5.9375

Gegeben ist die Zahl (10.101101)2.
Runden Sie die Zahl mittels "Abrunden (round min)" auf 3 Nachkommastellen.

x = 10.101 (unverändert)

Gegeben ist die Zahl (11.100011)2.
Runden Sie die Zahl mittels "round to nearest - round away from zero" auf 3 Nachkommastellen.

x1 = 11.100
x2 = 11.101
x´ = (11.100 + 11.101) * 0.1 = 11.1001
Lösung : 11.100

Angabe:
Multiplizieren Sie die Zahlen 1101010011111000 und 1000110010000010.
Als Rundungsvorschrift verwenden Sie bitte "round to nearest – round to even".

A: 1 10101 0011111000
B: 1 00011 0010000010

Neuer Exponent (kodiert):
10101 + 00011 - 01111 = 01001 $A_e + B_e = A+B-e$

Mantissen multiplizieren:
1.0011111000 *
1.0010000010 =
1.0110011001 | 011

Runden:
unverändert

Meine Lösung:
✓ 0 010010110011001

Angabe:
Subtrahieren Sie die Zahl 1011010110001111 von der Zahl 0011001010110100.
Als Rundungsvorschrift verwenden Sie bitte "round to nearest – round to even".

A (pos) - B (neg) = A + B (pos)

A: 0 01100 1010110100
B: 1 01101 0110001111

Exponent von A angleichen (+1), Bit shift nach rechts
Implizite 1 wird sichtbar

A: 0 01100 1101011010 | 000 -->

Addieren:
0.1101011010
1.0110001111

10.0011101001

Normalisieren:
Exponent +1 und shift nach rechts

01100 + 1 = 01101
10.0011101001 → 1.0001110100 | 100

Runden:
Weitere Rundungsregel -> lsb = 0
unverändert lassen

Meine Lösung:
0 01101 0001110100

Richtige Lösung:
0 01110 0001110100

Angabe:
Addieren Sie die Zahlen 1 01000 1000110111 und 0 11100 0101101001

A (neg)
B (pos)

A: 1 01000 1000110111
B: 0 11100 0101101001

Exponenten ausgleichen:
A: 01000
B: 11100

Exp von A ist um 20 kleiner als B
A: 1 01000 1000110111
A: 1 11100 0000000000 | 000

Addieren:
A = 0
Deshalb bleibt b gleich und ist die Lösung und muss nicht gerundet werden

Meine Lösung:
B: 0111000101101001

03: Codierungen

Beispiele für Zeichencodierungen:

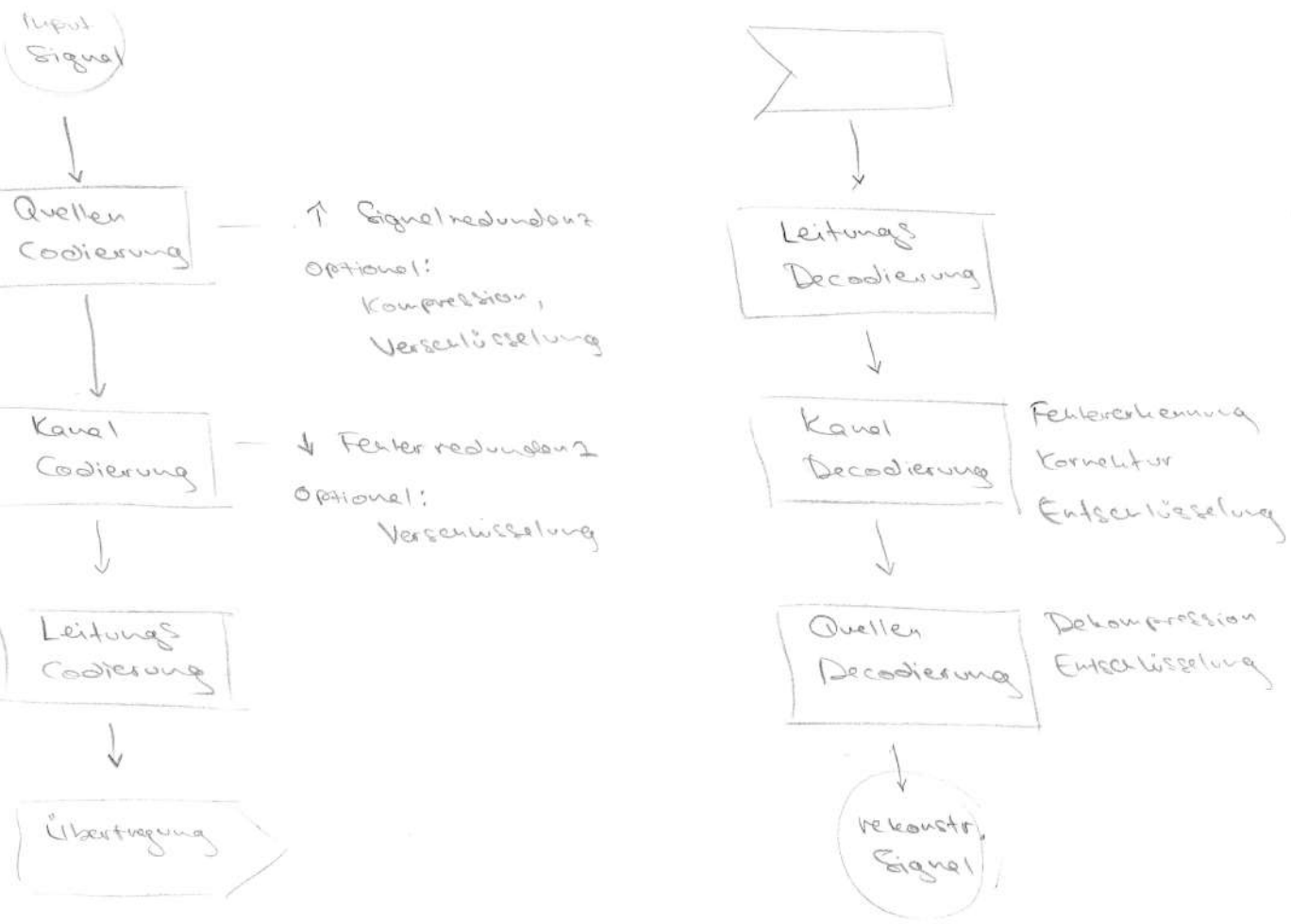
- ASCII
- Unicode
- UTF
- ISO-Latin-1

Im Alltag:

- QR-Codes
- Bar-Codes - (ISO 15420)

Codierungstheorie

Die mehrstufige Codierung: Trade-off
Sicherheit / Redundanz =. Aufwand



In der Quellencodierung

Signal → binärer Code, komprimiert, verschlüsselt

In der Kanalcodierung

binäres Code → Umcodierung, senkung redundanter Bits, Verschl.

In der Leitungscodierung

Optimierung von binärem Code für Übertragung.

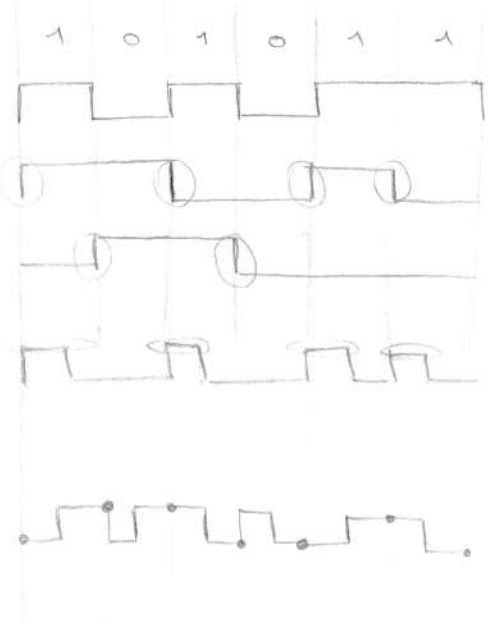
ZB Licht für Glasfaser, Strom, etc

Pegelfolgen = Bitfolgen

Leitungs-Code NRZ: Non return to zero

benötigt seperaten Taktsignal

kann unterscheidet zwischen high und low



NRZ-L low = 0 high = 1

NRZ-M Pegelwechsel = 1

NRZ-S Pegelwechsel = 0

RZ: unipolarer Return-to-Zero-Code

zweite Takthälfte immer "low"

D-Manchester:

Es muss in jedem Bit-Takt eine Flanke geben

- wenn man nach der Flanke auf derselben Seite ist wie davor: 0
- wenn man Seite wechselt: 1

Arten von Binärcodes:

Codes mit variabler Länge

zB UTF-Code, Huffman Code (Code-Baum) / Morse-Code

Codes mit fixer Länge (Blockcodes)

Nicht linear

zB EAN-13, US-ASCII früher

Linear

Durch Summe von Codewörtern bleibt es gültig (Modulo)

zB Hamming-Code

Zyklische Codes

Durch shiften / rotieren bleibt Code gültig

zB Polynom-Code (CRC-Codes)

Blockcodes können system. Codes sein

Systematische Codes (Prüffehlern Codes)

Prüfstellen / redundante Bits

zB. Paritätsbits (auch mehrdimensional)

Paritätsbits für Anteile (Hamming-Code)

Prüfsummen (EAN, Polynom-Codes)

EAN-13

fixe Länge, nicht linear

13 = 12 Stellen + Prüfziffer

$$(z_1) + 3z_2 + z_3 + 3z_4 + z_5 + 3z_6 + z_7 + 3z_8 + z_9 + 3z_{10} + z_{11} + 3z_{12} + \text{Prüfziffer} \equiv$$

z_1: Implizit: Aus A, B Muster (AABA..)

A, B

C

$$\equiv 0 \mod 10$$

Hamming-Distanz

Maß für die Störanfälligkeit von Code (Bestimmung mit XOR möglich)

zwischen 2 Wörtern:

$d(00,01) = 1 = D$

zwischen mehreren Wörtern:

minimale Hamming Distanz

Bei Abstand D gilt:

D-1 Fehler erkennbar

$k < \frac{D}{2}$ Fehler korrigierbar

Paritätsbits

[m] [P]

even-parity = # Einsen gerade bleiben

odd-parity = # Einsen ungerade bleiben

Hamming-Code

Linear, systematisch (fehlerkorrigierend), fixe Länge

Abstand D immer = 3

1 Bit korrigierbar

2 Bit erkennbar

Code indizes ab 1 zählen, an jeder Zer Potenz steht ein Prüfbit

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	...	c_{16}	Code
2^0	2^1		2^2				2^3			2^4	
↓	↓		↓				↓			↓	Prüfbit
P_1	P_2		P_3				P_4			P_5	

Da ZB. 27 = 16 + 8 + 4 + 1 wird c_{27} die Prüfbits c_{16}, c_8, c_4 und c_1 beeinflussen.

" " " "

P_5 P_4 P_3 P_1

Beispiel

c_1	c_2	c_3	c_4	c_5	c_6	c_7
P_1	P_2	d_1	P_3	d_2	d_3	d_4

$P_1 = c_1 = (c_3 + c_5 + c_7) \bmod 2 = c_3 \otimes c_5 \otimes c_7$

$P_2 = c_2 = (c_3 + c_6 + c_7) \bmod 2 = c_3 \otimes c_6 \otimes c_7$

$P_3 = c_4 = (c_5 + c_6 + c_7) \bmod 2 = c_5 \otimes c_6 \otimes c_7$

↑ XOR

Fortsetzung des Beispiels:

Sender: Datenwort 0110

Prüfbits

$P_1 = 0 \oplus 1 \oplus 0 = 1$

$P_2 = 0 \oplus 1 \oplus 0 = 1$

$P_3 = 1 \oplus 1 \oplus 0 = 0$

→ Zu übertragen: 1 1 | 0 | 0 | 1 1 0

Empfänger: Störung (1 1) 0 0 0 (1) 0

$P_1 = 0 \neq 1$

$P_2 = 1 = 1$

$P_3 = 1 \neq 0$

→ $1+4 = 5$ → c_5 wurde gestört

„Indikatoren"

↓

Beim Empfang Prüfbits selbst neu berechnen und vergleichen

Problem:

Fehlerbündel

Polynom - Codes

Sender und Empfänger kennen Generatorpolynom $G(x)$

$G(x) = x^3 + x^2 + 1 \quad (1\ 1\ 0\ 1)$

Sender

Nachrichtenwort: 0 1 1 0 (4 Bit)

Message - Polynom: $M(x) = 0x^3 + 1x^2 + 1x^1 + 0x^0 = x^2 + x$

1. $x^r \cdot M(x)$ wobei r = Grad von $G(x)$

$x^3 \cdot M(x) = x^3 (x^2 + x) = x^5 + x^4 \quad 0110000$

2. $\frac{x^r \cdot M(x)}{G(x)}$ und $R(x)$ bestimmen

$(x^5 + x^4) : (x^3 + x^2 + 1) = x^2$

$x^5 + x^4 + x^2$

x^2 Rest $\rightarrow R(x) = x^2 \quad 100$

3. $T(x) = x^r \cdot M(x) - R(x)$

Transmission = Übertragung

$T(x) = (x^5 + x^4) - x^2 = x^5 + x^4 + x^2$

↑ Rest abgezogen, dadurch Restfrei!

0110 100 → Absenden

$M(x)$ Sicherungsbits

(Hier kann auch addieren)

Empfänger

0110100 wird fehlerbehaft übertragen

1. $T(x) + E(x)$ Error-Polynom empfangen

2. $\frac{T(x) + E(x)}{G(x)}$ decodieren

Da $T(x)$ Restfrei war, zeigt sich ein Rest

CRC - cyclic redundancy check codes

Zyklische Polynom-Codes mit Standard $G(x)$

ZB: CRC-12 Generator – Polynom 12. Grades

Dadurch 12 Absicherungsbits für 6 Bit Nachrichten

Insgesamt: $6 + 12 = 18$ Bit zu übertragen

12 Absicherungsbits

- 11 Fehler erkennen
- < 6 Fehler korrigieren

Fehler $< \frac{\text{Distanz}}{2}$

N = message m + redundancy r

Es sind 2^r mögliche redundante Codierungen möglich

- 1 Muster für Fehlerfreiheit
- $2^r - 1$ weitere für weitere Informationen über Fehlerquelle

04: Informationstheorie

Nach dem Übertragungsmodell:

Das Medium überträgt den Signal.

Analog: Wert - kontinuierlich

Diskret: Wert - diskret

Sprache

Alphabet: Menge aller Zeichen → Wort

Syntax: Grammatik

Semantik: Bedeutung

Meta-Sprache: Zur Definition einer neuen Sprache

Code und Codierung

Codierung: bijektive Abbildung von 2 Wörtern zwischen 2 Sprachen

Wort ⟼ Codewort

↑ Verarbeitung, Störsicherheit

z.B. Fernschreiber mit binary Digits" = Bits

8 Bit = 1 Byte

2^{10} Bit = 1 Kibit (Kilo)

⋮

Informationstheorie nach Claude Shannon

Informationsgehalt / Überraschungswert

P = Auftrittswahrscheinlichkeit / Ereignis-Wsl

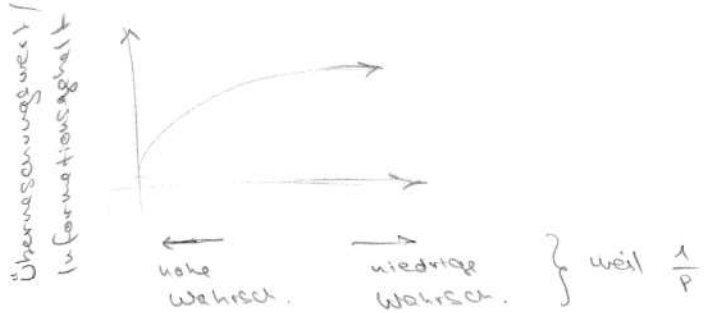
Es muss gelten:

↑ Wahrscheinlichkeit = ↓ Überraschung / Informationsgehalt $f\left(\frac{1}{p}\right)$

Weil Zeichen voneinander unabhängig Σ Einzelzeichen = gesamt. Wort

Also:

$h(x) + h(y) = h(x \cdot y)$ → Erfüllt vom Logarithmus



Informationsgehalt h

$$h = ld\left(\frac{1}{p}\right) = ld\left(p^{-1}\right) = -ld(p) \quad [\text{Bit}]$$

p ↑ Wahrscheinlichkeit.

Mittlerer inf. Gehalt H

$$H = \sum_i p_i h_i = \sum_i p_i \, ld\left(\frac{1}{p_i}\right) = -\sum_i p_i \, ld(p_i) \quad [\text{Bit}]$$

Code-Wortlänge L

zB $x \longmapsto 1 \quad l_0 = 1$

$y \longmapsto 01 \quad l_1 = 2$

$z \longmapsto 00 \quad l_2 = 2$

Mittlere Wortlänge L

$$L = \sum_i p_i l_i \quad [\text{Bit}]$$

Redundanz R

$$R = L - H \quad [\text{Bit}]$$

relative Redundanz r

$$r = \frac{R}{L} \quad [\text{Einheitslos}]$$

Huffman-Code

↓ Redundanz durch Datenreduktion (Auch Morse Code)

Wenn man ganze Zeichenfolgen kodiert → beliebig kleine Redundanz

a	0,40	a	0,4	a
b	0,20	de	0,22	bc ...
c	0,18	b	0,20	de
d	0,11	c	0,18	
e	0,11			

Adaptiver Huffman-Code:

relative Häufigkeit in laufender Interaktion.

Nach n Zeichen Baum neu aufbauen

06 : KV-Diagramme, Binary decision diagrams (BDD)

KNF : Konjunktive Normalform
DNF : Disjunktive Normalform

für bool'sche Funktionen

A	B	C	Ergebnis	Klausel
0	0	0	0	$A \vee B \vee C$
0	0	1	0	$A \vee B \vee \neg C$
0	1	0	1	$\neg A \wedge B \wedge \neg C$
0	1	1	1	$\neg A \wedge B \wedge C$
1	0	0	0	$\neg A \vee B \vee C$
1	0	1	1	$A \wedge \neg B \wedge C$
1	1	0	0	$\neg A \vee \neg B \vee C$
1	1	1	1	$A \wedge B \wedge C$

Wenn Ergebnis 0 dann invertiert und als Disjunktion

Wenn Ergebnis 1 dann als Konjunktion

Alle Disjunkte mit "und" verbinden → Konjunktive Normalform : KNF

$(A \vee B \vee C) \wedge (A \vee B \vee \neg C) \wedge (\neg A \vee B \vee C) \wedge (\neg A \vee \neg B \vee C)$

Alle Konjunkte mit "oder" verbinden → Disjunktive Normalform: DNF

$(\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge B \wedge C) \vee (A \wedge \neg B \wedge C) \vee (A \wedge B \wedge C)$

„Vollkonjunktion"

Diese Repräsentation ist nicht optimal.

Wiederholung:

$(a \wedge \neg b \wedge c)$ Konjunktive Form

$(a \vee \neg b \wedge \neg c)$ Disjunktive Form

Negationen nur in atomarer Form!

$\neg(a \wedge b) = \neg a \vee \neg b$ → nicht atomar

$(\neg a \vee \neg b) = \neg a \vee \neg b$ → atomar

Vereinfachung mit KV-Diagrammen

„Karnaugh map"

- Nicht immer minimal
- Nur bis 4 Variablen sinnvoll

	$\neg x_3 \neg x_4$	$\neg x_3 x_4$	$x_3 x_4$	$x_3 \neg x_4$
$\neg x_1 \neg x_2$	1	1	0	0
$x_1 \neg x_2$	0	1	1	0
$x_1 x_2$	0	0	1	X
$\neg x_1 x_2$	1	0	1	1

Beinhaltet alle möglichen Vollkonjunktionen

1 = kommt vor

0 = kommt nicht vor

X = don't care

eine mögliche minimale Blockeinteilung:

Wir suchen minimale Anzahl an Blöcken um alle 1en oder 0en abzudecken

- Pro Block nur 2, 4, 8, 16, 32 ...
- Nachbarn unterscheiden sich nur um eine Variable: $(\neg x \vee x) = 1$

Davor:

$\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4$

$\neg x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4$

→ „minimale disjunktive Normalform"

$(\neg x_1 \wedge \neg x_3 \wedge x_4) \vee (\ldots) \vee (\ldots) \vee (\ldots)$

↑ weil 4 Blöcke gefunden

ITE - If then else

$a \wedge b \equiv ITE(a, b, 0)$

$\neg a \equiv ITE(a, 0, 1)$

$a \vee b \equiv ITE(a, 1, b)$

Konversion:
Wahrheitstabelle ⇄ ITE

Shannon-Zerlegung

$f: \mathbb{B}^n \mapsto \mathbb{B}$ bool'sche Funkt.

„Kofaktor von f" in der $x_i = 1$

$f_{i,1}: \mathbb{B}^n \mapsto \mathbb{B}$

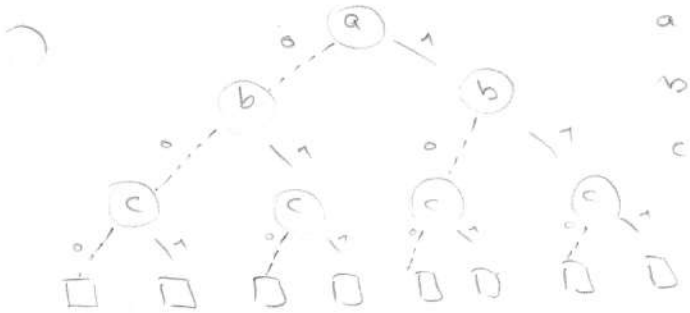
oder

$f_{i,0}: \mathbb{B}^n \mapsto \mathbb{B}$

$f = (x_i \wedge f_{i,1}) \vee (\neg x_i \wedge f_{i,0})$

Binary Decision Diagrams BDDs

Beispiel: Von if, else Baum



a - Ebene

b - Ebene

c - Ebene

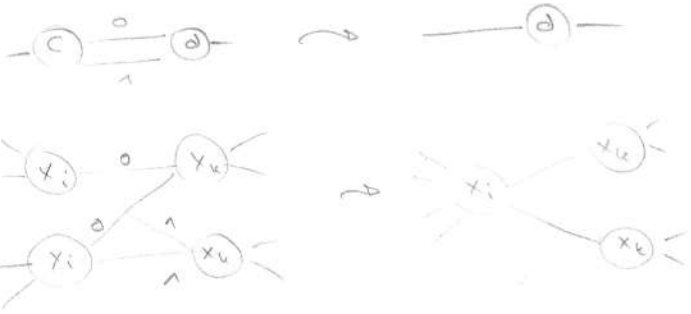
→ Alle Variablen zeigen auf die selbe 1/0

Vereinfachen mit

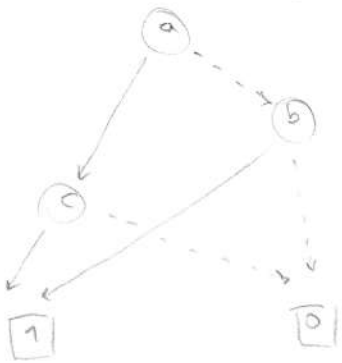
delete - rule

merge - rule

(nicht minimal)



DNF ablesen, KNF ablesen



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

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

Reduktion mit Pseads

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

1010 1100

↑ Abhängig davon ob man a,b,c oder eine andere Permutation hat

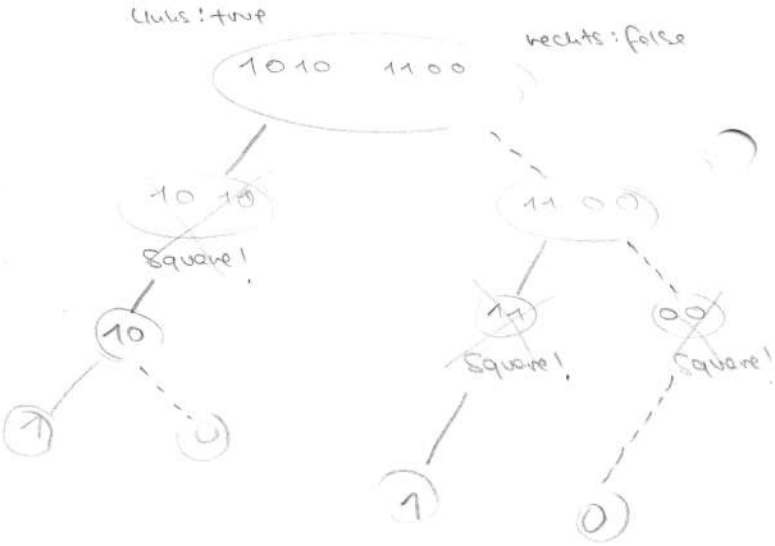
Pseud: erste und zweite Hälfte stimmen nicht

Square: 7 Pseud

Bei „don't cares":

Greedy: Gleich am Anfang

Lazy: Aufschieben

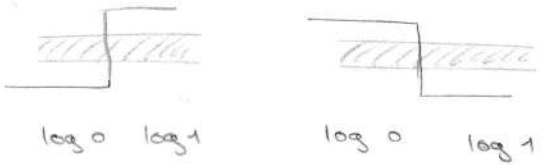


07 : Kombinatorische Logik, Digitalschaltungen, kombinat. Digitaltechnik

Bool'sche Algebra mit digitalen Systemen

Diskretisierung

kont. analoge Signale ⟶ boolsche Werte



Boolsche Funktionen durch Gatter-Bausteinen

Boolsche Operatoren sind „funktional vollständig" wenn mit ihnen alle anderen Operat. hergeleitet werden können;

$\{\wedge, \neg\}$, {NAND}, {NOR}

weniger Transistoren, billiger herzustellen ⟶ Substitution mit Nand und Nor

Standard-Baugruppen

Encoder: verdichtet Code

n-zu-m-Encoder

n Eingänge ≥ m Ausgänge

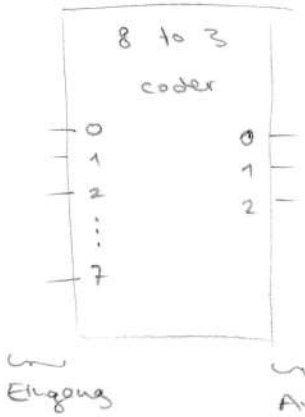
Decoder

n zu m Decoder

n Eingänge ≤ m Ausgänge

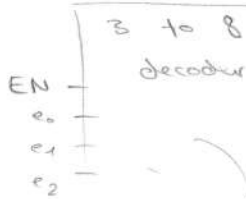
Beispiel:

„1 aus 2^n-Code" ⟶ n Bit Binärzahl



e_7	...	e_3	e_2	e_1	e_0	a_2	a_1	a_0
0	...	0	0	0	1	0	0	0
0	...	0	0	1	0	0	0	1
0	...	0	1	0	·	0	1	0
0	...	1	0	0	·	0	1	1
·		·	·	·	·	·	·	·
·		·	·	·	·	·	·	·

Bei Decoder umgekehrt.



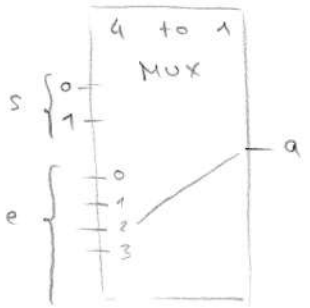
Lässt sich auch kaskadieren:

4 to 16

=

5 × [2 to 4 Decoder]

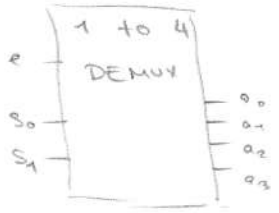
Multiplexer



S_1	S_0	a
0	0	e_0
0	1	e_1
1	0	e_2
1	1	e_3

Demultiplexer

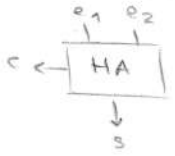
Äquivalent zu Decoder mit Enable-Eingang



Eingang = Enable Input

Addierer

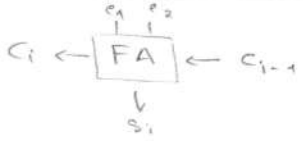
HA Half Adder



e_1	e_2	$e_1 + e_2$
0	0	0 0
0	1	0 1
1	0	0 1
1	1	1 0

Carry: $(e_1 \wedge e_2)$ — Sum: $(e_1 \oplus e_2)$

FA Full Adder



e_{1i}	e_{2i}	c_{i-1}	c_i	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

↑ Kanonische DNF

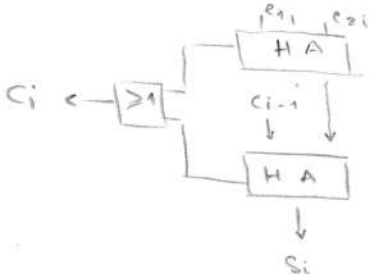
$$s_i = (\neg e_{1i} \wedge \neg e_{2i} \wedge c_{i-1}) \vee (e_{1i} \wedge \neg e_{2i} \wedge c_{i-1}) \vee \dots$$

Es gilt:

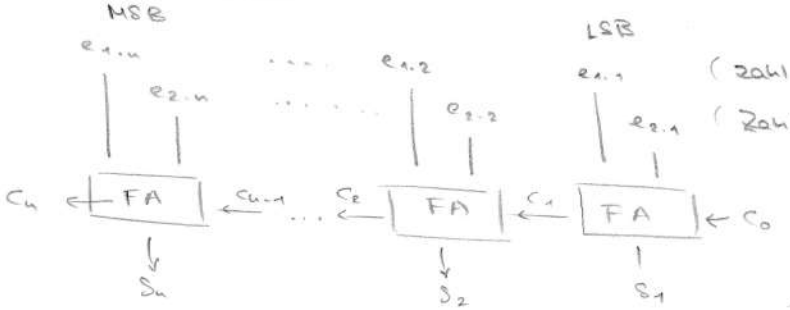
$$s_i = e_{1i} \oplus e_{2i} \oplus c_{i-1}$$

$$c_i = (e_{1i} \wedge e_{2i}) \vee (c_{i-1} \wedge (e_{1i} \oplus e_{2i}))$$

Lässt sich mit Kaskadierung von HA implementieren:



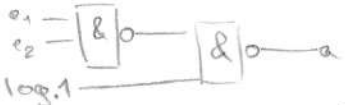
Paralleladdierer



Auch Subtraktion möglich: eine Zahl invertieren, ersten FA: $C_0 = 1$

Beispiele: Substitution mit NAND / NOR:

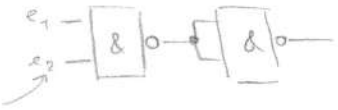
AND Substitution



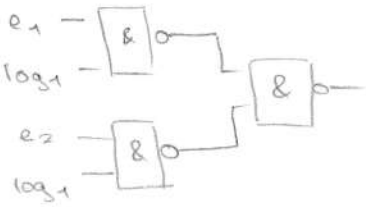
$(e_1 \wedge e_2) = a$

$\neg(\neg(e_1 \wedge e_2) \wedge 1) = \neg((\neg e_1 \vee \neg e_2) \wedge 1)$

$= e_1 \wedge e_2 \vee 0$



OR Substitution



$(e_1 \vee e_2) = a$

$\neg(\neg e_1 \wedge \neg e_2) = (e_1 \vee e_2)$

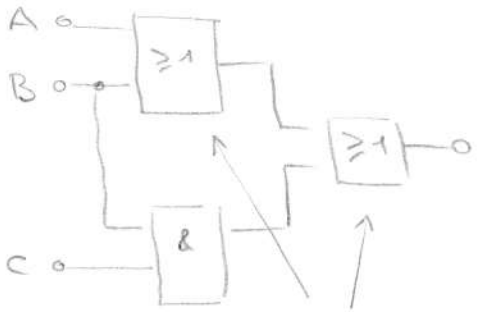
NOT Substitution

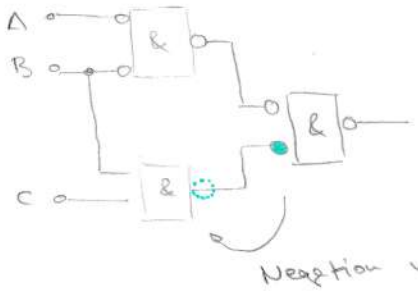
$\neg e = a$

$\neg(e \wedge 1) = \neg e \vee 0$

Beispielsfunktion: Ersetzen der Gatter mit NAND

$f(A, B, C) = (A \vee B) \vee (B \wedge C)$

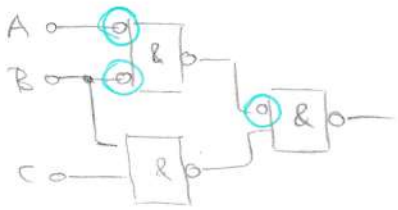


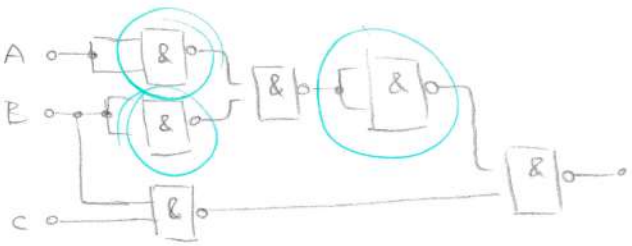


Negation verschieben

OR ersetzen:

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

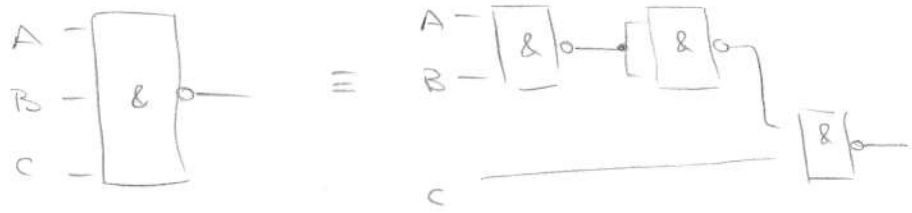




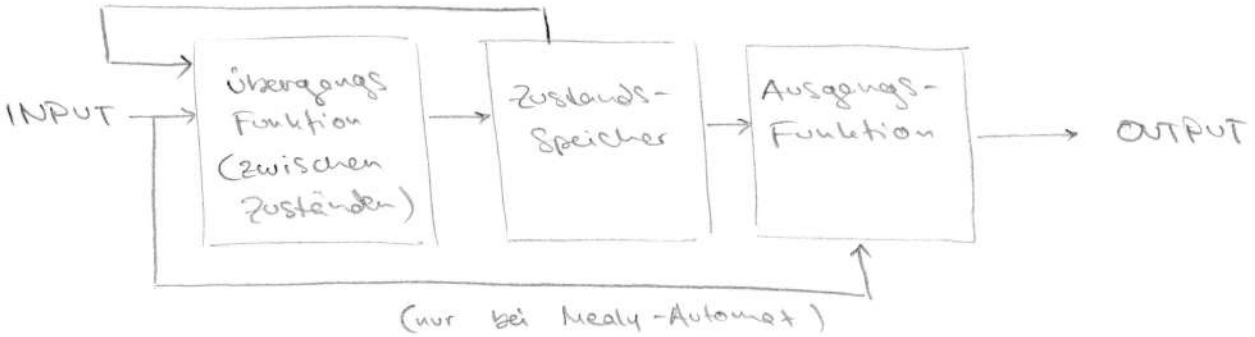
Jetzt wurden alle OR sowie AND mit NAND ersetzt

Aber wir haben noch 3 Negationsgatter

3 NAND Gatter Eingänge



08 Automaten



Formale Definition

Systeme mit Zustand (Schaltwerk) werden mit Automaten beschrieben.

Endliche Automaten:

- endliche Zustandsmenge Q
- Startzustand $s \in Q$
- Endzustandsmenge $F \subseteq Q$
- Übergangsfunktion δ
- Endliches Eingabealphabet Σ

Moore-Automat

- Ausgabealphabet Ω, und Ausgabefunktion $\lambda : Q \mapsto \Omega$

Mealy-Automat

- Ausgabealphabet Ω, und Ausgabefunktion $\lambda : Q \times \Sigma \mapsto \Omega$

(Q: Zustand, Σ: Eingabesymbol)

○ Deterministische Automaten:

- Nachfolgezustand genau definiert

$\delta : Q \times \Sigma \mapsto Q$

Nicht-Deterministische Automaten

- $\exists$ ein Zustand-Eingabepaar ohne Nachfolge-Zustand

$\delta \subseteq Q \times \{\Sigma \cup \{\epsilon\}\} \times Q$ → $\exists$ Zustände ohne Eingaben

(ϵ: Leerwort)

○ Vollständige Automaten

- $\forall (z, x) \in Q \Rightarrow \exists z' : \delta(z, x) = z'$

Wir gehen davon aus:

Automat vollständig & deterministisch

Moore

- Zustand bestimmt Output
- Input beeinflusst Zustand
- Output ändert sich nur bei Zustandsänderung

Mealy

- Input & Zustand bestimmen Output
- Für jeden Zustand sind Input-Abhängig mehrere Outputs möglich
- Meistens weniger Zustände

Zusammengefasst:

Deterministisch: Eindeutiger Folgezustand für $\forall$ möglicher Eingaben in $\forall$ möglichen Zuständen:

Übergangsfunktion: $\delta : Q \times \Sigma \mapsto Q$

Nicht determin.: Übergangsfunktion: $\delta \subseteq Q \times \{\Sigma \cup \{\epsilon\}\} \times Q$

(ϵ: Leerwort und Alpleb., Ausgabe)

Vollständig: Min. 1 Folgezustand — " —

$$\text{zu } \forall (z, x) \in Q \times \Sigma \quad \exists\, z' : \delta(z, x) = z'$$

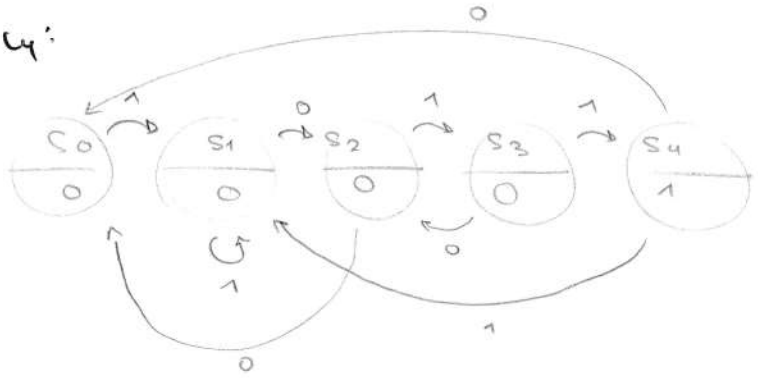
($\exists$: genau 1)

Endlich: Beschränkte # an Zuständen

Aufgabenstellung:

Output soll 1 sein, wenn Input Bitfolge 1011 ist,
(Überlappungen (1011011) auch bewerten!)

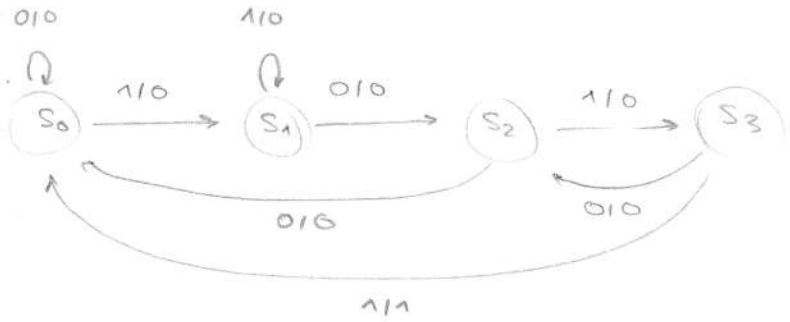
Mealy:



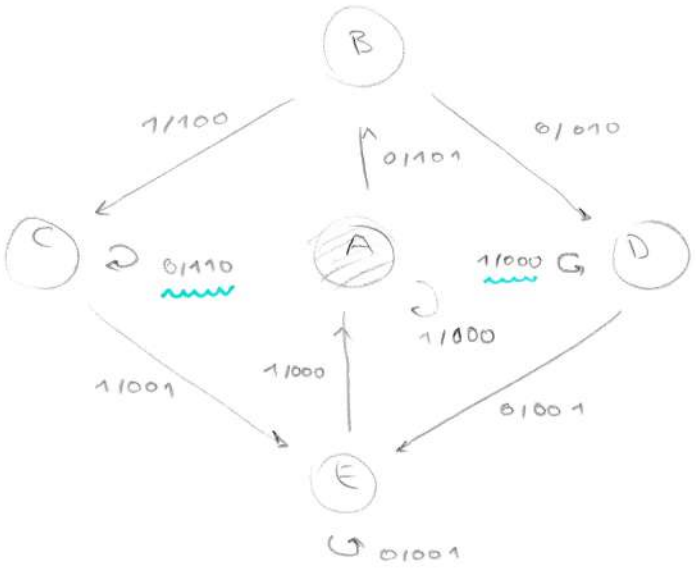
Zustand	gelesene Folge
S_0	-
S_1	1
S_2	10
S_3	101
S_4	1011

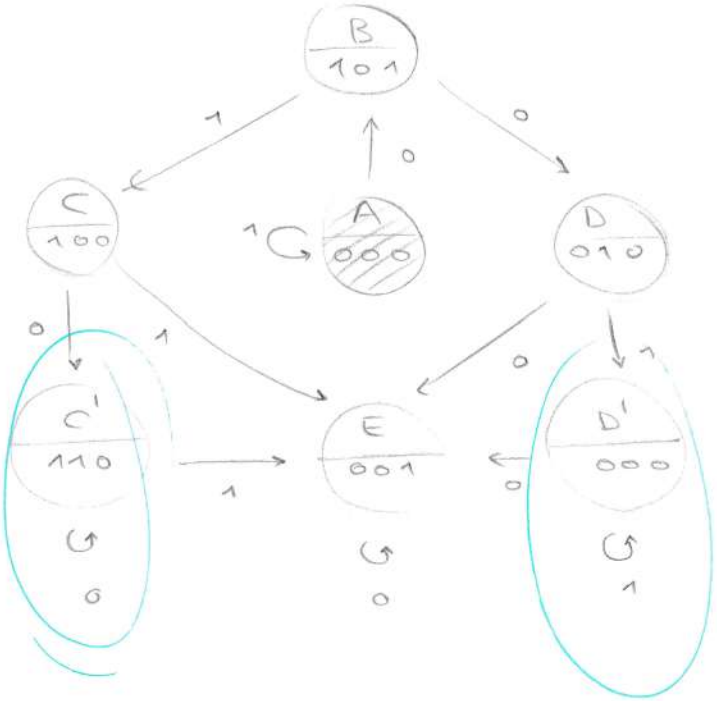
Bei jedem Übergang wird ein Bit vom Input „konsumiert".

Moore:



Konversion / Transformation



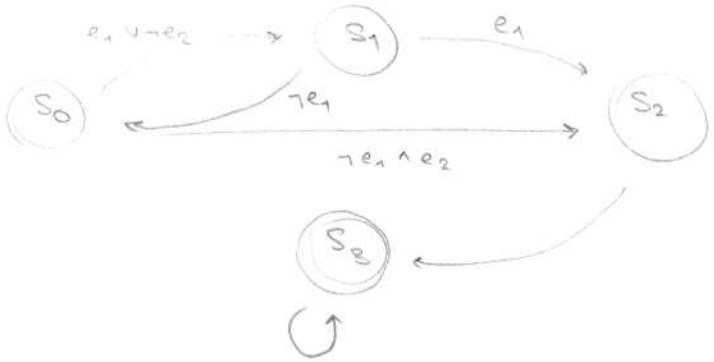


Zustandsgraph - Beispiel

$Q = \{S_0, S_1, S_2, S_3\}$

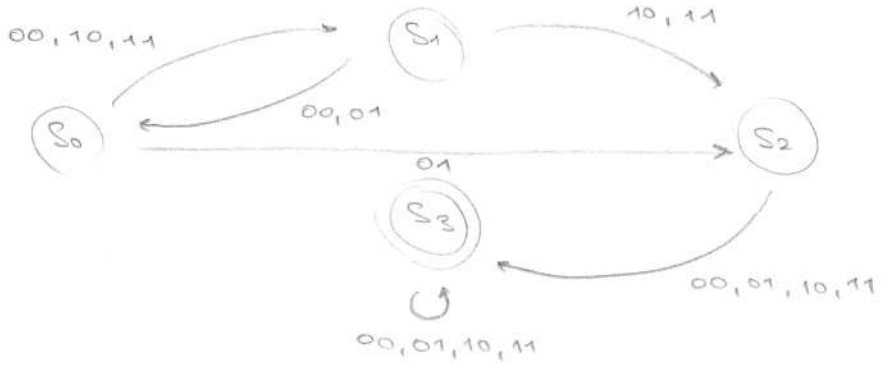
$s = S_0$

$F = \{S_3\}$ (wenn kein F mit 2 Ringen eingezeichnet, dann $\forall$ Knoten = Endzust.)



Wir können e_1, e_2 mit Boitmuster angeben:

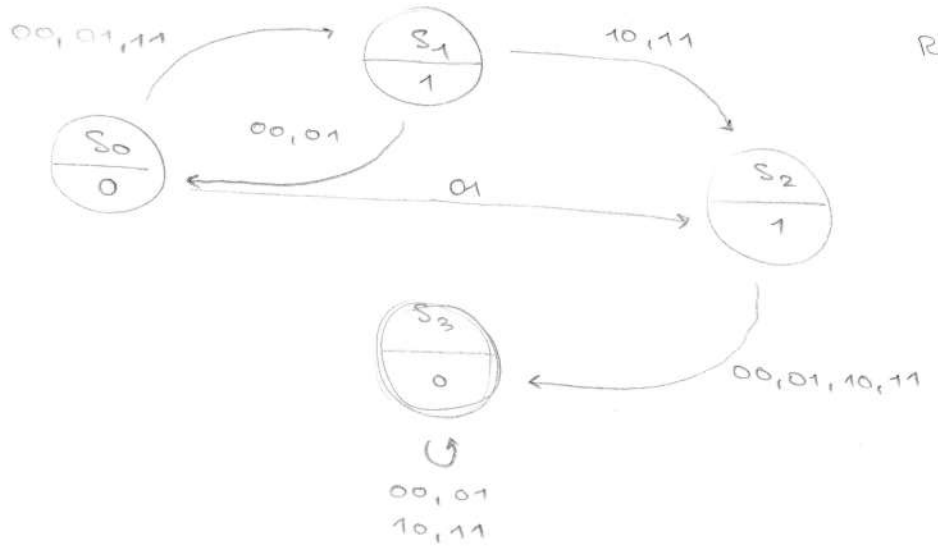
Reihenfolge: $e_1 e_2$



Moore Schaltwerk

Zustand bestimmt Ausgabe:

$$\lambda = \{(S_0,0), (S_1,1), (S_2,1), (S_3,0)\}$$



Reihenfolge der Eingabe: e_1, e_2

Ausgabe: a

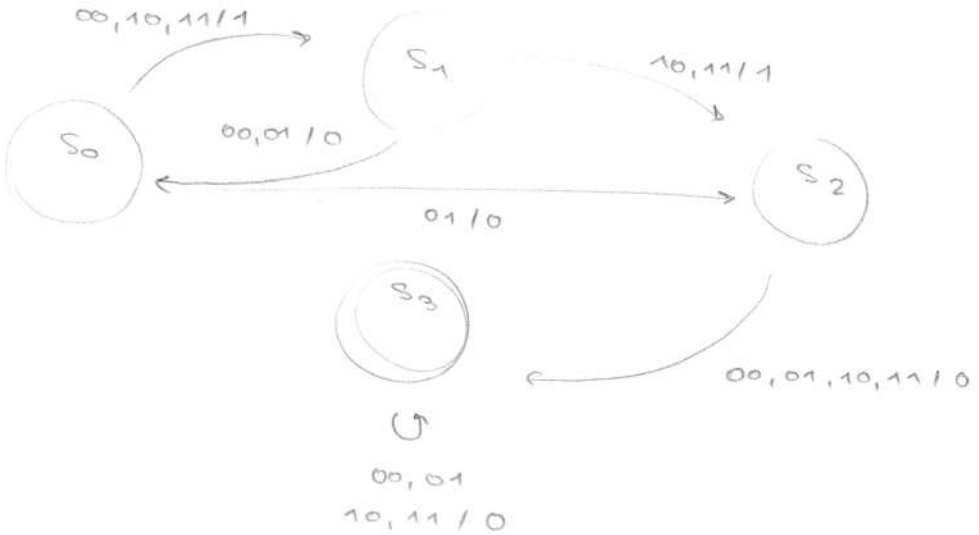
↑

Immer Ausgaben!

Mealy Schaltwerk

Zustand UND Eingabe bestimmen Ausgabe:

$$\lambda = \{((S_0,00),1), \ldots, ((S_0,01),0), \ldots\}$$



09 : Sequenzielle Logik

Kombinatorische Logik

Output vom aktuellen Input-Zustand abhängig

Keine Speicherung von Informationen / Zuständen.

Sequenzielle Logik

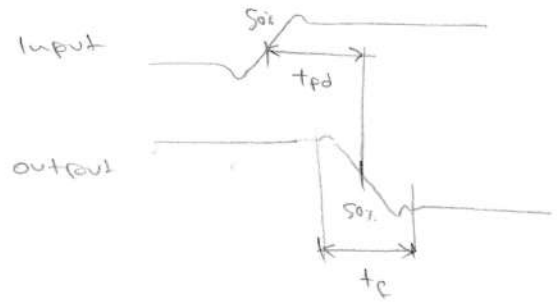
Output vom aktuellen Input und bisherigem Ereignisverlauf (Historie) abhängig

Mit Speicherelementen

Zeitverhalten

Idealisiert:

Real:



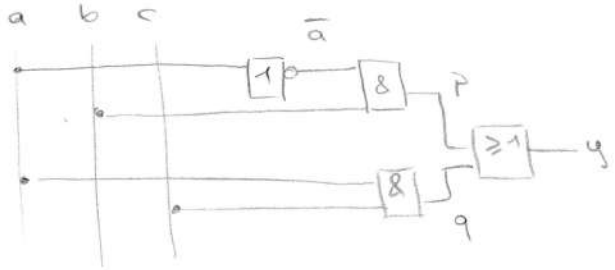
t_{pd}: Durchlaufzeit

t_f: Flankensteilheit

Electronics Glitch / logical hazard

Wenn man keinen Takt hat:

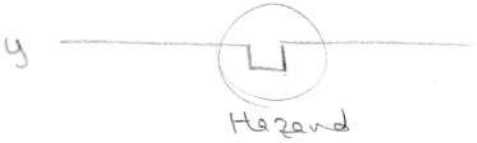
Output kann kurzfristig falsch sein



$f(a,b,c) = (\neg a \wedge b) \vee (a \wedge c)$

Wenn b und c wahr sind, muss y unabhängig von a wahr bleiben.

Wenn $a=1 \rightsquigarrow a=0$ wird q vor p true sein



Speicherelemente: Latches und Flipflops

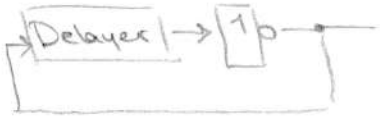
Speichern 1 Bit (so lange Spannung anliegt)

Taktzustand → Latches
Taktflanken → Flipflops

Takt-Signal (Clock-Signal) [Hz]

Synchronisiert (vor allem große) Schaltungen

- synchrone Taktung (periodisch)
- asynchrone Taktung

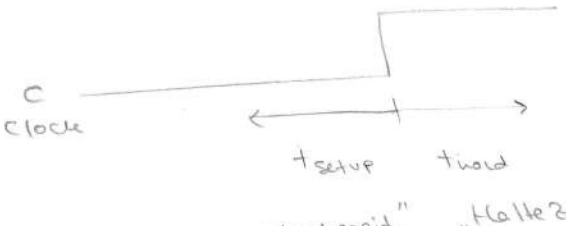


Stabiler:
Quarzoszillator

Flanke = Zustandswechsel

positive Flanke | negative Flanke | durch pos flanke getriggert | durch neg flanke getriggert

Flipflop / Zeitspezifikation beim Takten:



„Vorbereit." „Haltezeit"

ZB bei D-Flipflop:
Eingang muss in Vorber. und Haltezeit stabil bleiben.

Sonst metastabile Zustände

Glitches und Hazards

Wenn am Ausgang kurz falscher Wert steht, erzwugt durch das nicht-Einhalten von t_{setup} / t_{hold} und generell durch t_{pd}

Metastabiler Zustand

unkontrolliertes Togglen.

ZB wenn man bei RS-Latch gleichzeitig von $R=S=1$ auf $R=S=0$ wechselt

$R=S=1$ = unerlaubte Eingabe

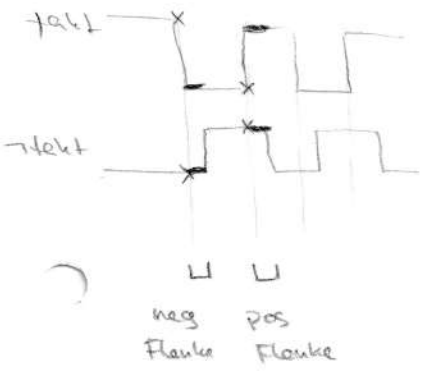
Bistabile Kippstufe

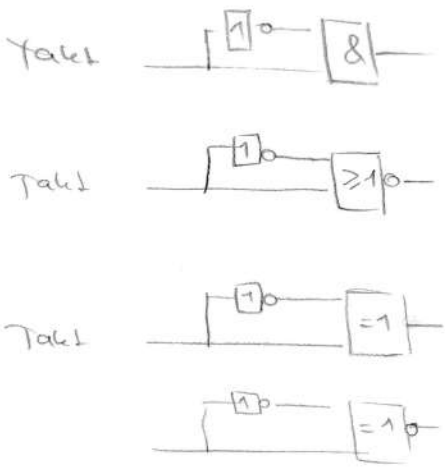
Schaltungen die 1 Bit speichern können besitzen 2 stabile Zustände

ZB Set, Reset

Monostabile Kippstufe

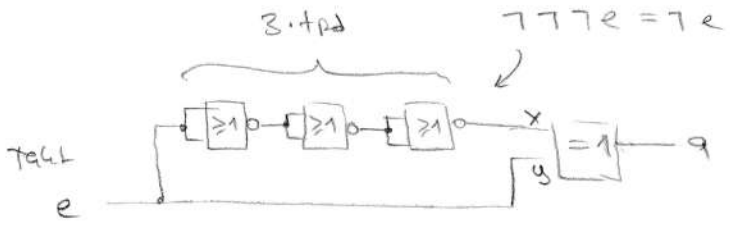
Nur ein Zustand ist stabil —> So entsteht ein Takt: es fällt immer nach der Flanke zurück zum stabilen Zustand





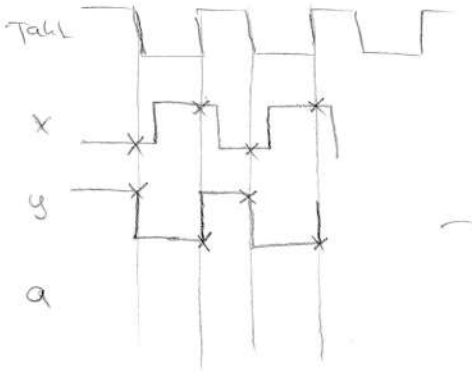
wäre besser als die Version ohne Negation von XOR

Lösung des Beispiels:



	NOR	XOR
0 0	1	0
0 1	0	1
1 0	0	1
1 1	0	0

Äquivalent zu NOT

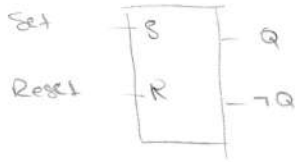


→ für negative Flanke $x = 0$ $y = 1$ ⟶ 1

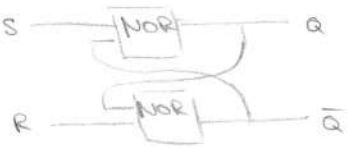
für positive Flanke $x = 1$ $y = 0$ ⟶ 1

Speicherelemente

RS - Latch



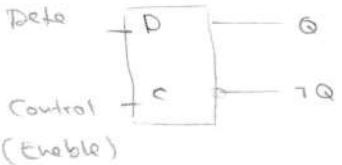
S	R	
0	0	Hold
0	1	Reset = 0
1	0	Set = 1
1	1	Nicht erlaubt → Instabilität



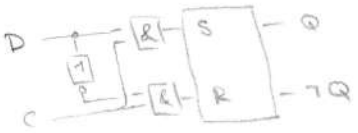
Wenn R=S=1 und dann R=S=0
dann ständiger Wechsel

$Q = \neg Q$ ⟳ $Q = \neg Q$
$= 0$ $= 1$

D - Latch

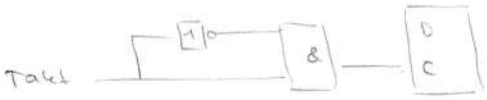


Baut auf RS-Latch auf:



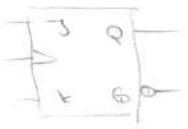
D = 1 → Set
D = 0 → Reset
C = 0 → Hold

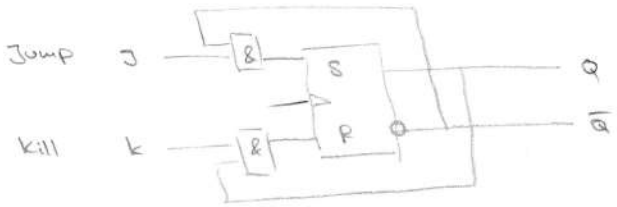
D - Flip-Flop



Bewusst erzeugter Hazard: Erkennung von Taktflanke

JK Flip-Flop

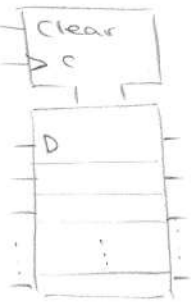


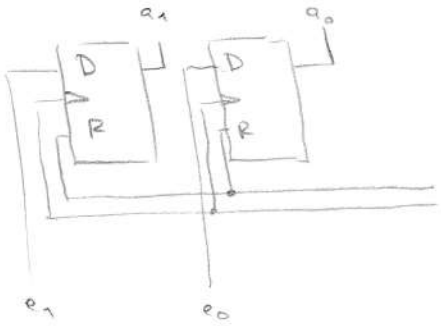


J	K	
0	0	Hold
0	1	0
1	0	1
1	1	Toggle

Anwendungen

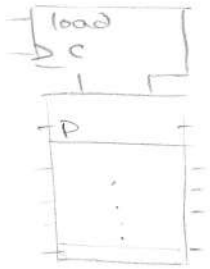
Register

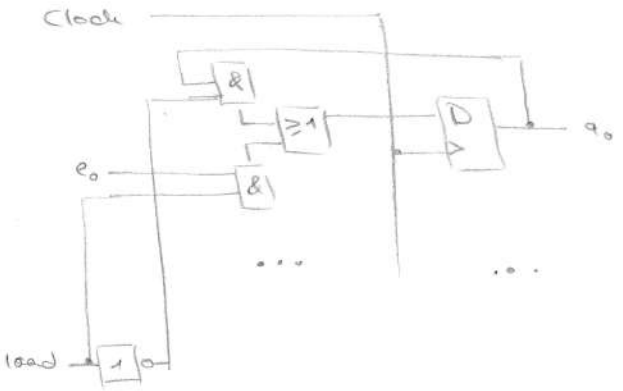




bei RS Latch in D-Flipflop ist R noch erreichbar

Parallel - Register



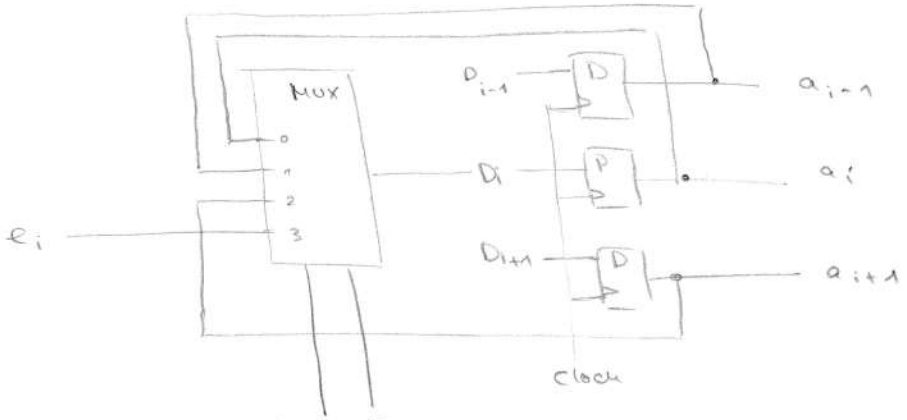


wenn load: Zustand übernehmen, ansonsten Cold

Schiebe - Register

Anwendung: paralleles Addieren

Universal - Register

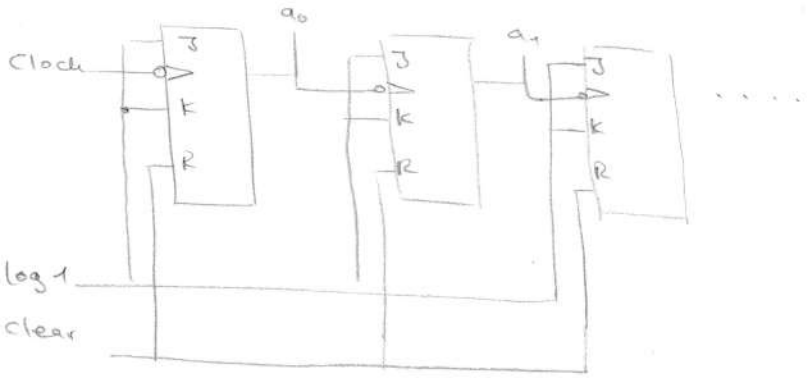


	S_0	S_1	
0	0	0	—
1	0	1	shift left
2	1	0	sift right
3	1	1	parallel load

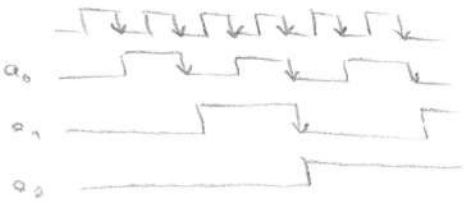
Alle Mux werden gleichzeitig mit selber Angabe geschaltet

Zähler

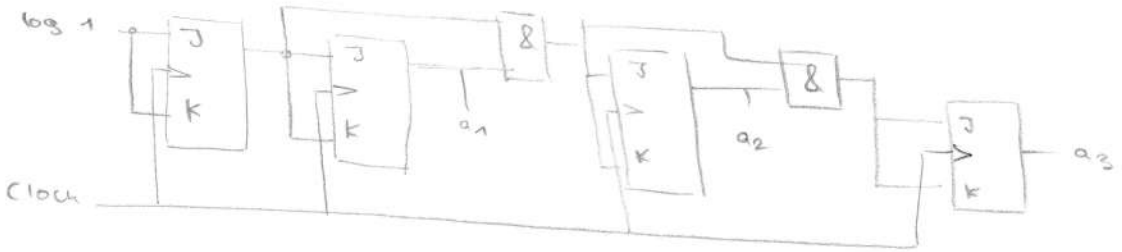
(asynchron getaktet)



Dadurch, dass J=K=1 wird getoggled bei jeder negativen Flanke:



(synchron getaktet)



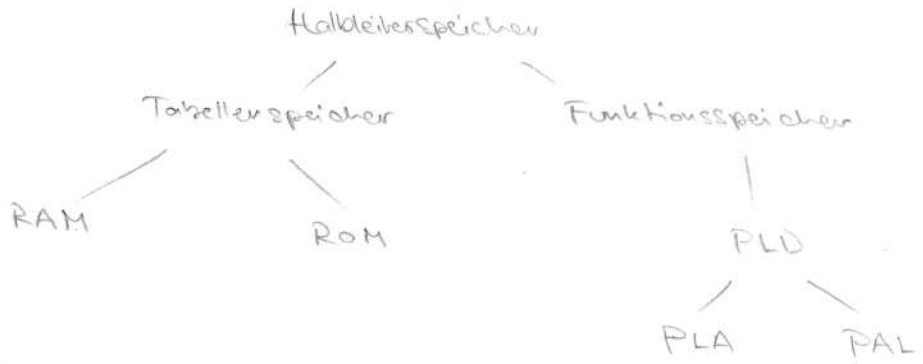
↑ toggled mit jedem Takt

↑ toggled wenn $a_0 = 1$

↑ toggled wenn $a_0 \& a_1 = 1$

↑ toggled wenn $a_0 \& a_1 \& a_2 = 1$

10: Speicher



Tabellenspeicher speichern Werte in Zeilen die ansprechbar sind (Gedächtnis)
Funktionsspeicher speichern bool'sche Funktionen (kein Gedächtnis)

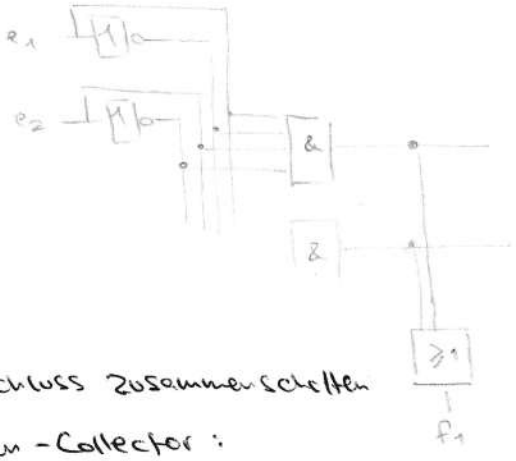
Funkt. Speicher lassen sich als Tabell. Speicher implementieren, vice versa

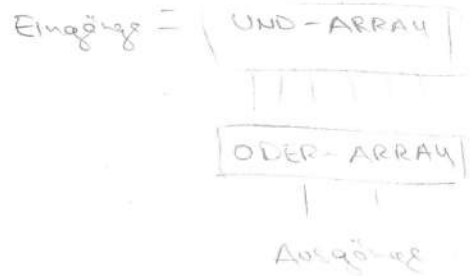
Funktionsspeicher: PLD

PLD programmable logic device
PLA programmable logic array — UND-ODER Matrizen
PAL programmable array logic — ODER ist vordefiniert

Vom Anwender 1x programmierbar:
Dioden / Fuse (—▷|—) verbindet zwei Leitungen und kann durchgebrannt werden mit versch. Stromstärken

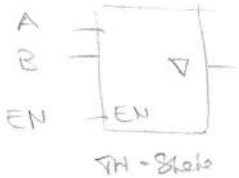
Funktion als disjunktive Normalform DNF:



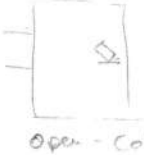


Ausgänge ohne Kurzschluss zusammenschalten

Tri-State und Open-Collector:



Leitet nicht wenn enable aus ist



ASIC

Application Specific integrated circuit

- Schnell, effizient, nur 1 Funktion
- teuer

→ z.B. für Handy: GSM-Protokolle fix

Es gibt auch **FPGA** (field programm. gate arr.)
wie PLA aber effizienter.

Tabellenspeicher: ROM

Read-only-memory

- permanente Speicherung, nicht flüchtig
- 3 mehrfach programmierbare

1K → # Datenwörter, # Zeilen (1024)
8 → Bitbreite

Beispiel: 16 x 4 Rom

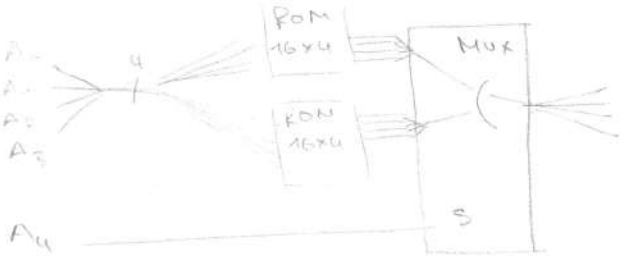
Mit einem 4 zu 16, wird eine Zeile gelesen

↑ # Eingang ↑ # Ausgang

Zum Programmieren wird ein Anti-Fuse benutzt (Beim verschmelzen wird kontakt erzeugt.)

Erweiterung von ROM: Multiplexer

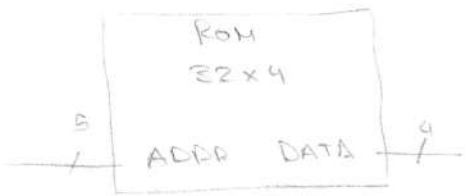
Statt einem 32 x 4 Rom kann man zwei 16 x 4 Roms verwenden



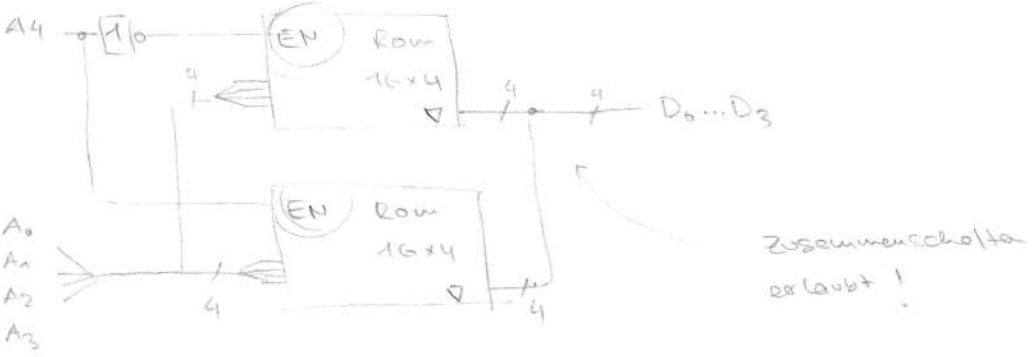
↑ Multiplexer

Rom 1 für Zeile 0 bis 15
Rom 2 für Zeile 16 bis 31

Dadurch:

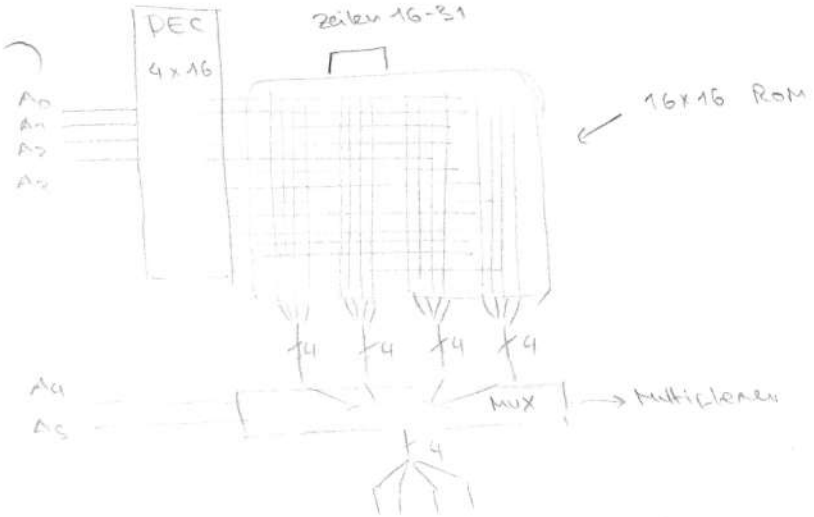


Erweiterung von ROM: Tri-State



Erweiterung von ROM: Multiplexer

Aus 16x16 ROM → 64 x 4 ROM



Anwendungen von ROM heute:

- Flash EEPROM
  USB, SSD, sehr schnell
- EEPROM
  Speichern von Betriebssystemen, Embedded Systems → daten die selten geändert werden.

ROM vs RAM:

- Nicht flüchtig
- langsamer
- hohe kosten pro GB
- begrenzte # an Löschzyklen

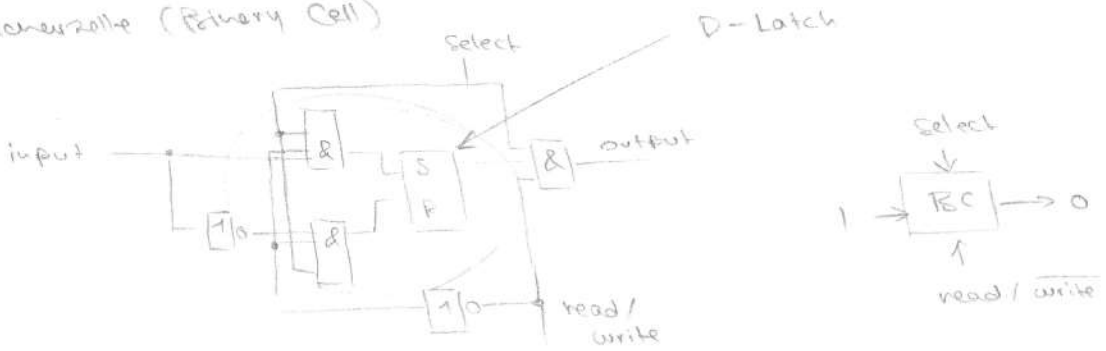
Tabellenspeicher : RAM

random-access-memory (historisch bedingt)

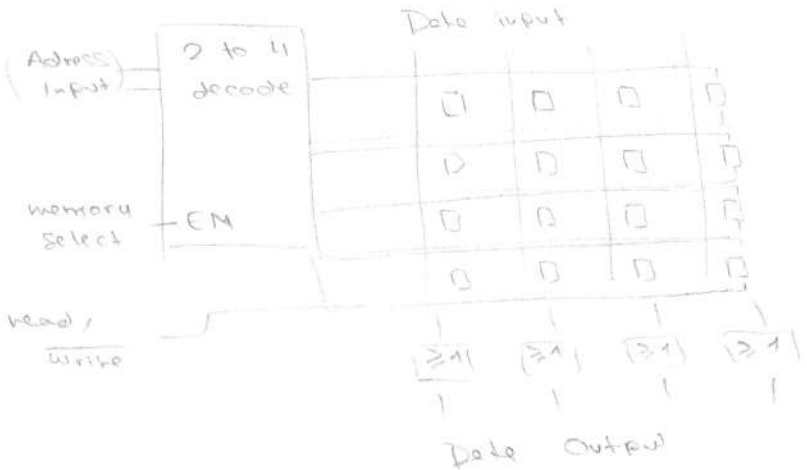
DRAM : flüchtig, Speicherung im Kondensator

SRAM : nicht flüchtig, aus CMOS Transistoren

Speicherzelle (Binary Cell)



Matrix aus binary cells:



Kaskadierung wie bei ROM mit decoder möglich.

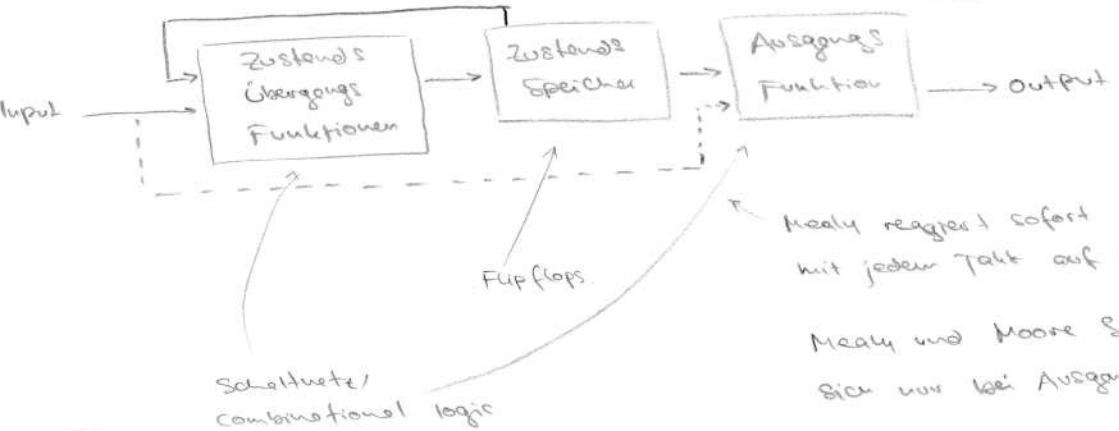
11 : Schaltwerke

Schaltnetze
Combinational logic

logische Funktionen ohne
Speicher / internen Zuständen

Sequentielle Schaltwerke
Sequential logic

Ausgang abhängig von internem Zustand
und Eingang.



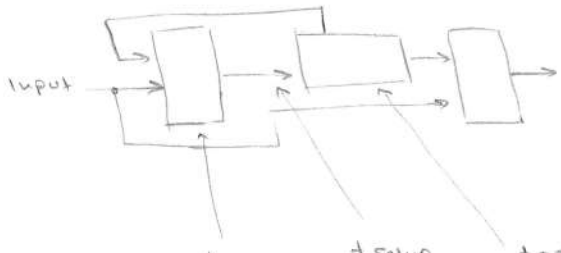
Mealy reagiert sofort
mit jedem Takt auf Eingänge

Mealy und Moore Schaltwerke unterscheiden
sich nur bei Ausgangsfunktion.

Taktung, Vermeidung von Meta-Zuständen

Annahmen:
- immer synchron: Änderung nur bei Taktsignal
- Eingangssignal auch synchron mit Takt, weil mealy sofort auf input reagiert

Zustand muss aktualisiert werden bevor nächste Eingabe folgt:



t_G — Durchlaufzeit Übergangsfunkt.

t_{setup} — Vorbereit. Flipflops

t_{FF} — Verzögerung Flipflops

Minimaler Taktabstand:

$$T_{min} = t_{FF} + t_G + t_{setup}$$

Maximaler Takt f_{max}:

$$f_{max} = \frac{1}{T_{min}}$$

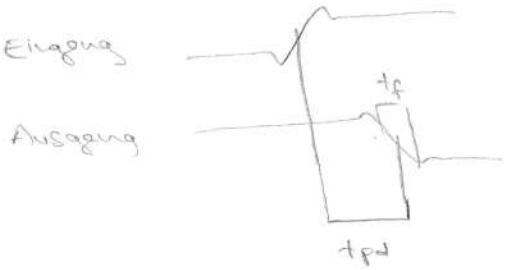
Zeitverhalten

Abstraktion:

NOT Gatter

Realität:

Es braucht Zeit bis Ladungsträger fließen



propagation delay t_{pd}

Durchlaufzeit

Von der Taktflanke bis zur Ausgangsänderung, quasi Reaktionszeit

rise / fall time t_f

Anstiegs / Abfallzeit

Steilheit der Flanke

Weiters:

Bei Flipflops (zB D-Flipflop) muss Eingang während Taktflanke stabil bleiben, weil sonst instabile Zustände entstehen können (unkontrolliertes Toggeln)

Setup time t_{setup}

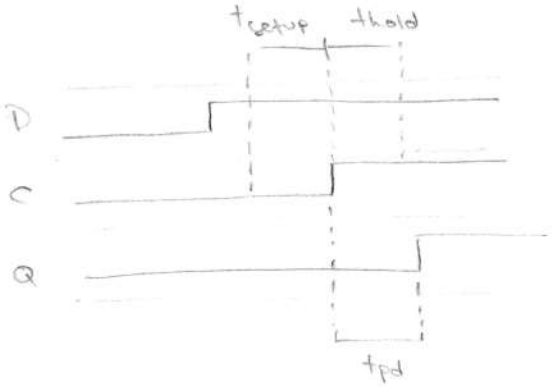
Vorbereitungszeit

Zeit vor Taktflanke

Hold time t_h

Haltezeit

Zeit nach der Taktflanke



Maximale Taktfrequenz

$$\text{Frequenz [Hz]} = \frac{1}{T_0 \text{ [s]}}$$

→ Zeit benötigt für eine Periode

SI-Präfixe:

T	Tera	10^{12}
G	Giga	10^{9}
M	Mega	10^{6}
k	kilo	10^{3}
h	hekto	10^{2}
da	deka	10^{1}
		10^{0}
d	dezi	10^{-1}
c	zenti	10^{-2}
m	milli	10^{-3}
µ	mikro	10^{-6}
n	nano	10^{-9}
p	piko	10^{-12}

selten benutzt

→ Beispiele:

$1\,ns = 10^{-9} \cdot s$

$1\,MHz = 10^{6} \cdot Hz$: 1 Mio Zyklen pro Sekunde

Beispiele zur Berechnung der maximalen Taktfrequenz von Flipflops

Benötigte Angaben:

Gatter Durchlaufzeit	15 ns
Flipflop Durchlaufzeit von Takt bis Ausgang	45 ns
t_{setup} für Flipflops	5 ns
	65 ns

$$\rightarrow \frac{1}{65 \cdot 10^{-9}} =$$

$= 15\,384\,615,384\ldots =$

$= 15{,}4\ MHz$

Weiteres Beispiel:

Gatter t_{pd}	40 ns
Flipflop t_{pd}	40 ns
Flipflop t_{setup}	10 ns
	90 ns

$$\rightarrow \frac{1}{90 \cdot 10^{-9}} =$$

$= 11{,}1\ MHz$

$\downarrow$

$11{,}1\ MHz < 25\ MHz$

$15{,}4 < 25\ MHz$

Max flipf. — Maximale Taktfrequ.

↑

keine obere Schranke!

Systematische Schaltwerkentwicklung

1. Aufgabenstellung verstehen
   - prüfe vollständigkeit

2. Entwurf des Zustandsgraphen (Es gäbe auch andere Wege Zustands übergänge zu modellieren)
   - prüfe vollständigkeit für alle mögl. Inputs
   - prüfe Eindeutigkeit

Deterministic finite State machine

State machine: Zustandsabhängiger konzeptueller Automat

finite: Endliche # an Zuständen

deterministic: Eindeutige Nachfolge Zustände

3. Minimierung der Zustands #
   - automatisiert

4. Zustandscodierung
   - 1 aus n" oder „dichte Codes."

5. Übergangs- und Ausgangsfunktion bestimmen
   - aus Zustandsgraph ableiten
   - mit KV-Diagramm vereinfachen

6. Dokumentation der Gesamtschaltung

7. Berechnung von f_{max} Taktfrequenz

Systematische Schaltwerkentwicklung

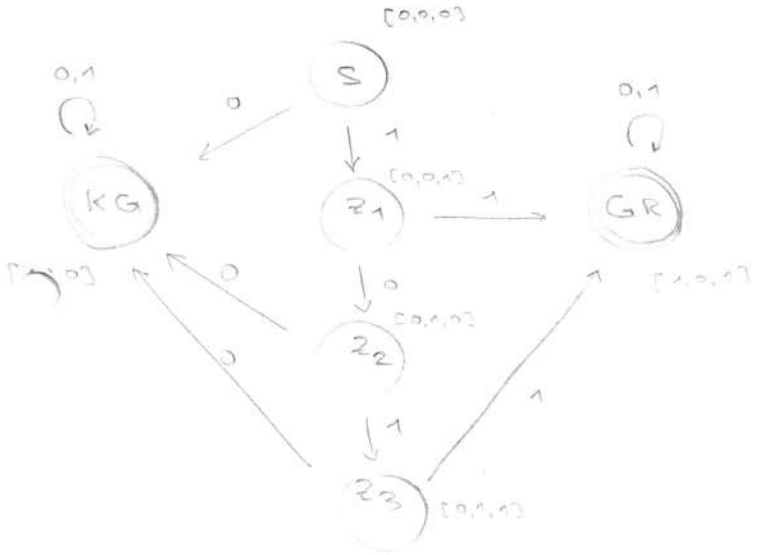
Beispiel

Aufgabe:

Schaltung mit Eingang E, durch die 4 Bits sequentiell geschoben werden (MSB zuerst)

Soll erkennen ob die Zahl (basis 2) $\leq (10)_{10} = (1010)_2$ ist.

Zustandsgraph:



Zustandscodierung:

Zustand	Bedeutung	D_2	D_1	D_0
S	BS (MSB)	0	0	0
Z_1	B2	0	0	1
Z_2	B1	0	1	0
Z_3	B0 (LSB)	0	1	1
KG	≤ 10	1	0	0
GR	>10	1	0	1

Bezieht sich auf Stelle in der Zahl

Eindeutige Codierung für jeden Zustand

→ direkte Codierung

Zustandsübergangstabelle:

alter Zustand				neuer Zustand		
D_2	D_1	D_0	E	D_2'	D_1'	D_0'
0	0	0	0	1	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	1	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	1
⋮						

→ Mit KV Diagramm vereinfachen

$D_2' = D_2 \vee (D_0 \wedge E) \vee (D_1 \wedge \neg E) \vee (\neg D_0 \wedge \neg E)$

$D_1' = (\neg D_0 \wedge D_1 \wedge E) \vee (D_0 \wedge \neg D_1 \wedge D_2 \wedge \neg E)$

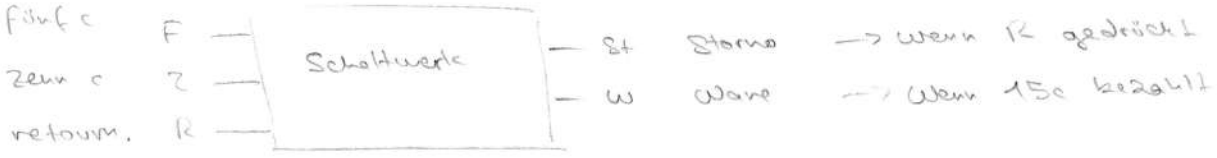
$D_0' = (\neg D_2 \wedge E) \vee (D_0 \wedge D_2)$

Abschließend:

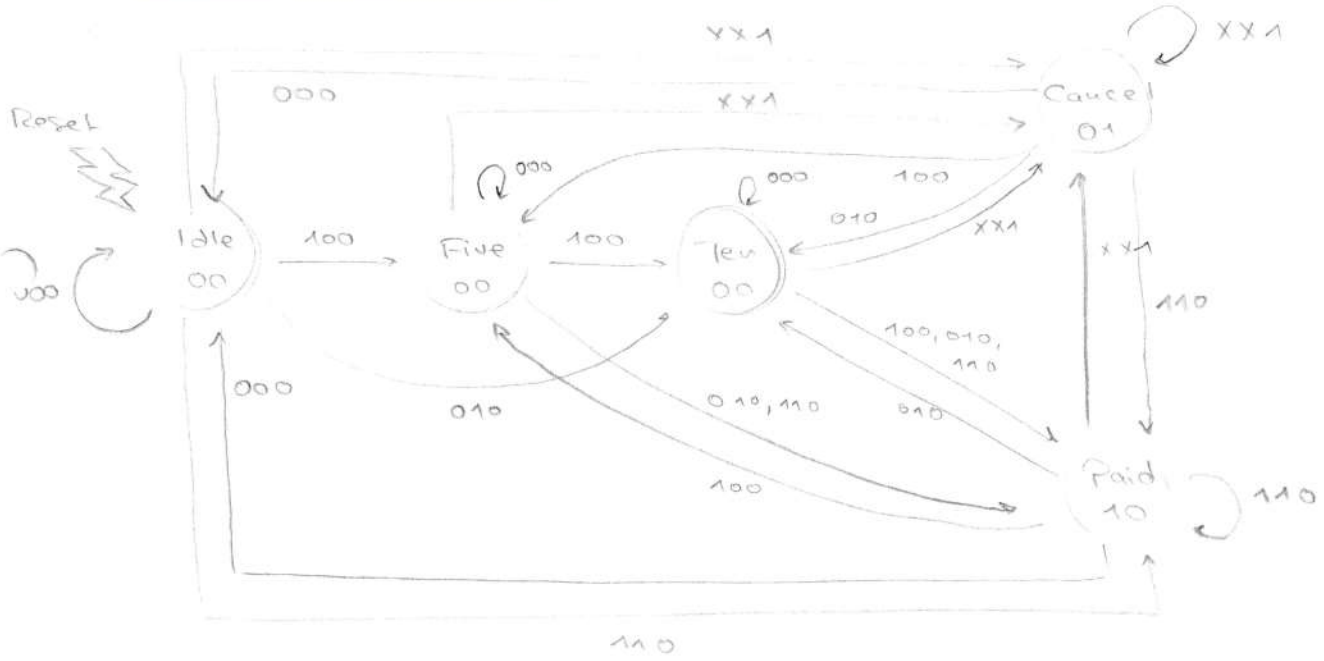
Entwurf des Schaltwerks

12 : Realisierung von Schaltwerken

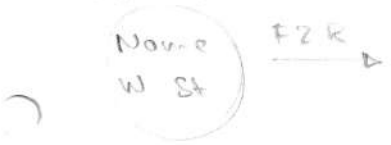
Beispiel: Münzautomat



Moore Zustandsgraph



Notation:



Wir wandeln den Zustandsgraphen, von der Zustandscodierung in eine Tabelle für die Übergangsfunktion um:

Input	F	0	... 0
	Z	0	... 1
	R	0	... 0
Aktueller Zustand	Idle	Idle	... Ten ...
	Five	Five	... Paid ...
	Ten	Ten	... Paid ...
	Paid	Paid	... Ten ...
	Cancel	Cancel	... Ten ...

→ neuer Zustand

Zustandscodierung

1 aus n - Codierung

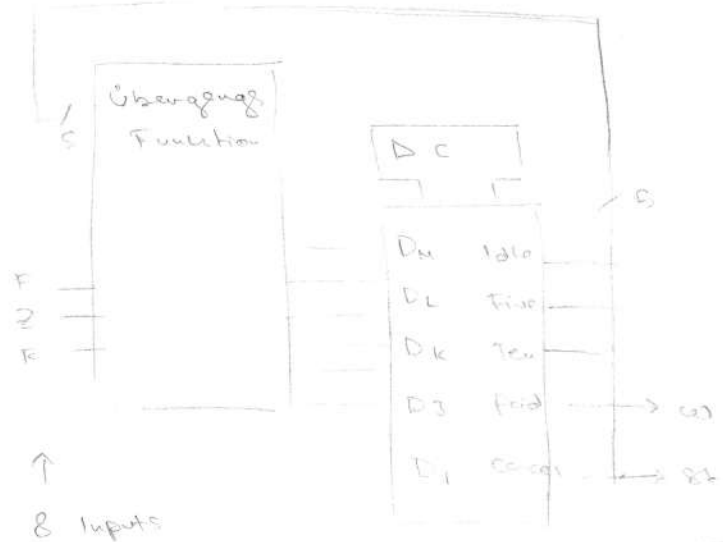
I	J	K	L	M	Zustand
0	0	0	0	1	Idle
0	0	0	1	0	Five
0	0	1	0	0	Ten
0	1	0	0	0	Paid
1	0	0	0	0	Cancel

"Dichte Codierung"

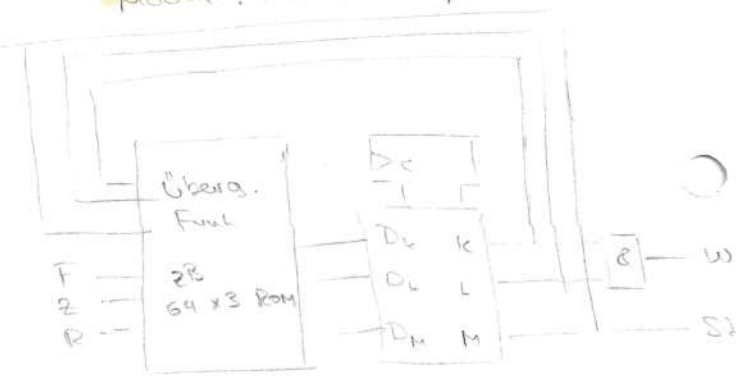
K	L	M	Zustand
0	0	0	Idle
1	0	0	Five
0	1	0	Ten
1	1	0	Paid
0	0	1	Cancel

↑ aus effizienzgründen diese Anordnung

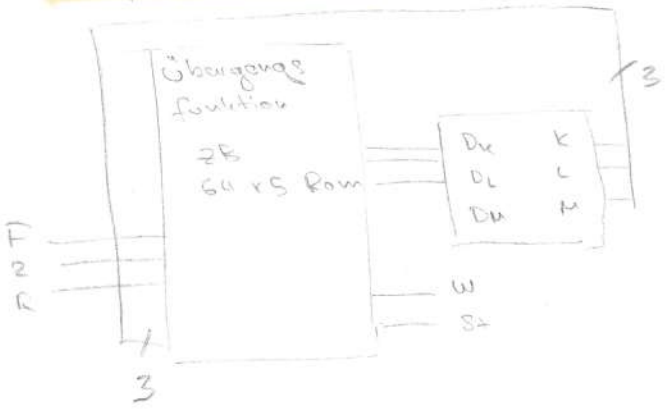
Moore: 1 aus n cod.



Moore: Dichte Cod.

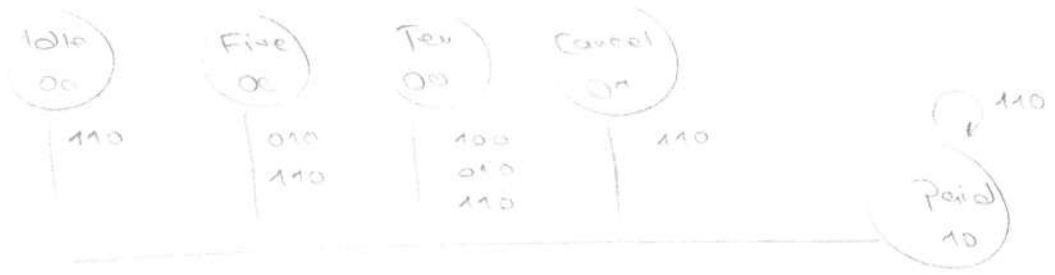


Mealy: Dichte Cod.



Realisierung der Übertragungsfunktion

Funktionsspeicher (Einzelne Gatter oder PLA, PAL)

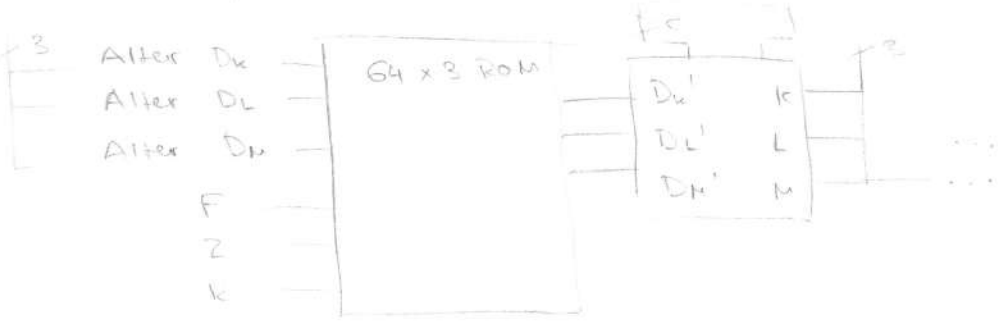


$D_{paid} =$
$(Idle \wedge F \wedge Z \wedge \neg R) \vee$
$(Five \wedge Z \wedge \neg R) \vee \dots$

Tabellenspeicher (ROM)

Angenommen: „Moore und Dense Encoding"

$2^6 = 64$, deshalb 64 x 3 ROM



Zeitlicher Ablauf

Siehe Folien für synchrone Signale (Eingang & Zustand)

Auch möglich: asynchr. Eingang

Weiteres Beispiel:

Binärzähler (zählen mit jedem Takt) → keine Eingaben: Autonomer Automat

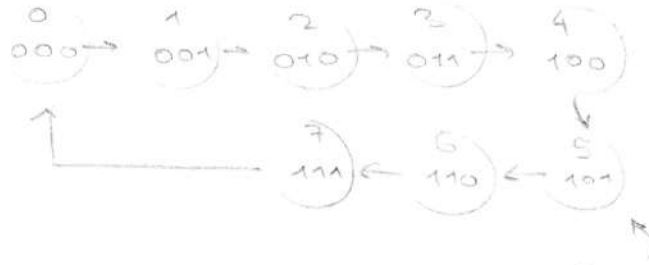
Zustandsspeicher: JK (Jump-Kill) - Flipflops

Übergangsfunktion: mit Gattern

Zustandsdiagramm:

(Urlaub around bei 7)

C	B	A
MSB		LSB



Dichte Codierung

Zustand	
0	...000
1	...001
2	...010
3	...011
.	
7	...111

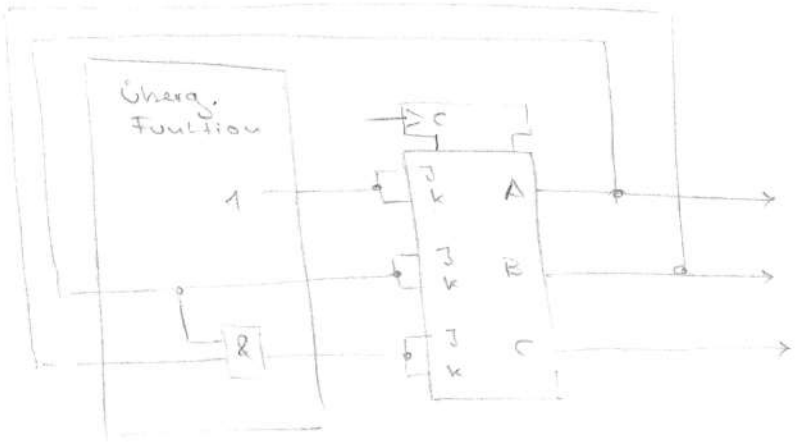
Übergangs und Ausgangsfunktion

- Ausgangsfunktion Zustand: = JK-Flipflop-Ausg.
- JK-Flipflop zum kontinuierlichen benutzt

JK	
00	hold
01	0
10	1
11	toggle

bei J=K=0 hold

J=K=1 toggle



Berechnung d. Maximalen Taktfrequenz

- pro Gatter 15 ns
- Durchlauf Takt bis Ausgang 45 s
- Vorbereitungszeit 5 u
- Haltezeit 2 us
- max Taktfrequenz JK-Flipflop 25 MHz

Berechnung:

Durchl. Flipflop: 45 us

Durchl. Übergang.: 15 us

(nur 1 Gatter)

Vorbereitung flipflop: 5 us

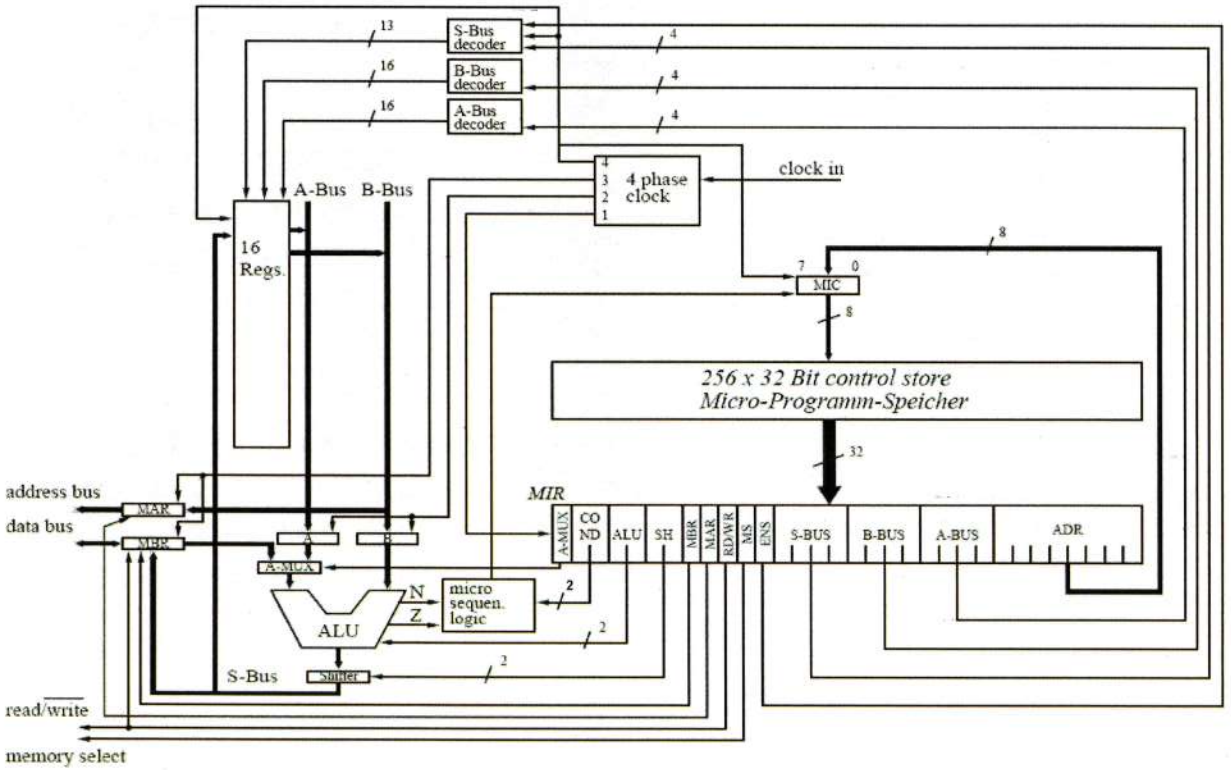
65 us

$\frac{1}{65\,us} = 15{,}4\ MHz$

15,4 MHz < 25 MHz

→ nicht beschränkend

Übersicht


S-Bus decoder
B-Bus decoder
A-Bus decoder
13
16
16
4
4
4
4 phase clock
clock in
A-Bus
B-Bus
16 Regs.
MIC
8
8
256 x 32 Bit control store
Micro-Programm-Speicher
32
MIR
address bus
data bus
MAR
MBR
A
B
A-MUX
ALU
N
Z
micro sequen. logic
2
2
2
S-Bus
Shifter
read/write
memory select
A-MUX
CO ND
ALU
SH
MBR
MAR
RD/WR
MS
ENS
S-BUS
B-BUS
A-BUS
ADR

14: Micro 16 (Mikroprocessor)

Prozessor Schaltwerk zur Verarbeitung von Daten
besteht aus Rechenwerk und Leitwerk

Rechenwerk Führt Rechenoperationen aus

Leitwerke Steuerung der Reihenfolge mit der Befehle ausgeführt werden

ALU Arithmetic logic unit

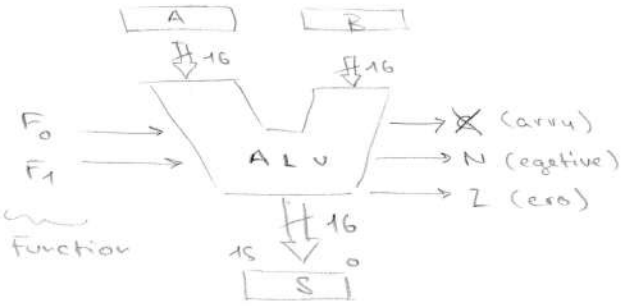
Funktionen:

16 Bit ALU benutzt 2 Register als Input

- Durchschalten ohne Änderung
- Bitwise UND
- Bitwise Komplementbildung
- Parallele Addition zweier Datenwörter → Carry bit am Ende ausgelassen (nicht returned)

Es gibt viele Wege zu addieren!

- Carry ripple adder
- carry skip adder
- Carry select adder
- carry look ahead adder

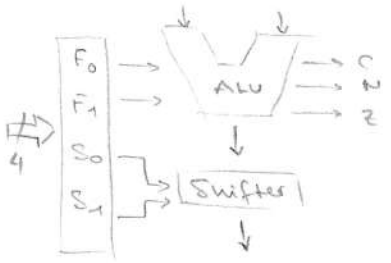


Micro Instruktionen:

F_0	F_1		symbolisch
0	0	unverändert A	$R \leftarrow A$
0	1	A + B	$R \leftarrow A+B$
1	0	$A \wedge B$	$R \leftarrow A \wedge B$
1	1	$\neg A$	$R \leftarrow \neg A$

Shifter

Folgt nach ALU



S_0	S_1		Bezieht sich auf Output
0	0	keine Änderung	SH ← R
0	1	shift left	SH ← lsh(R)
1	0	shift right	SH ← rsh(R)
1	1	–	–

Register File und Steuerleitungen

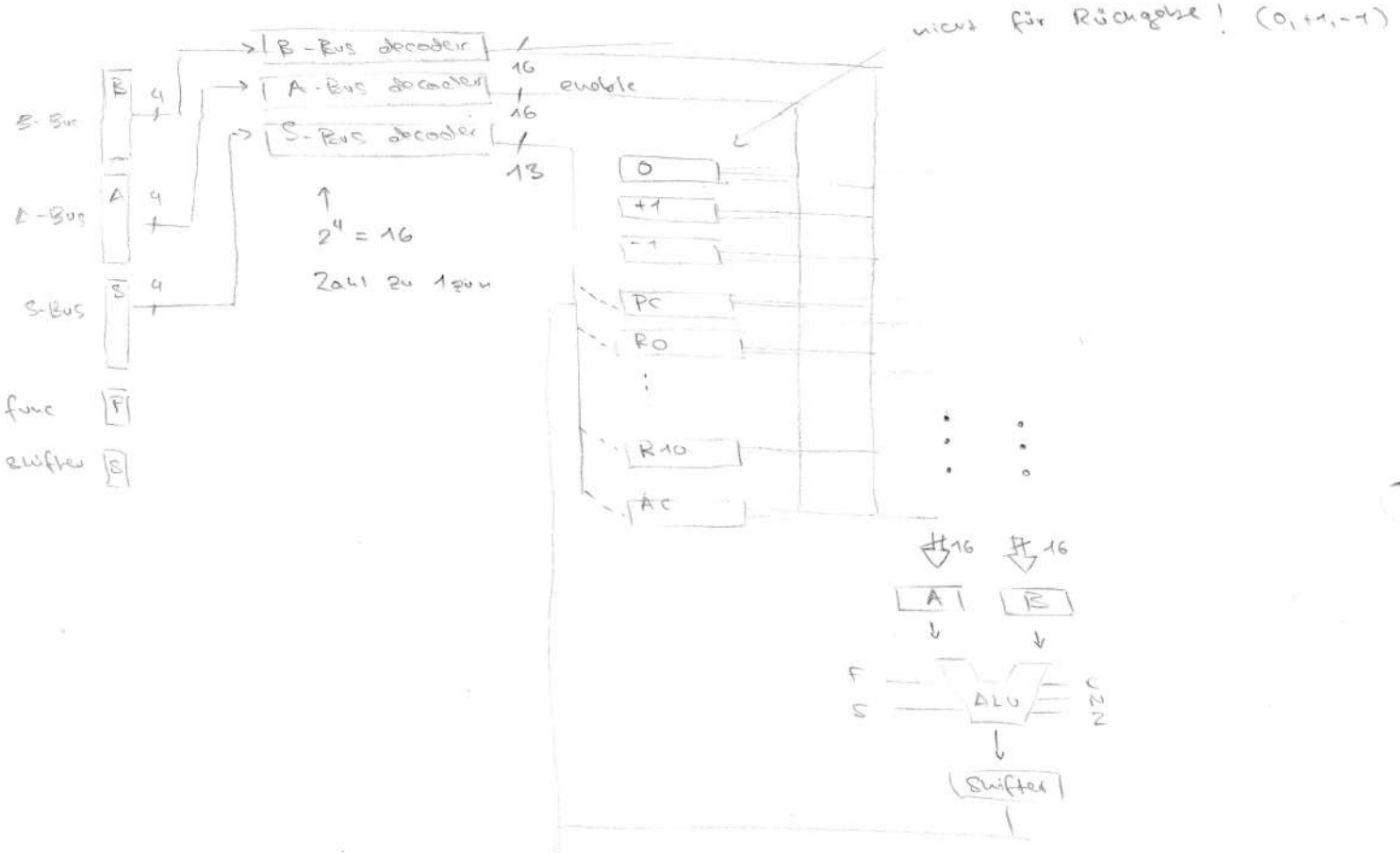
Multiplexer schaltet Inhalt von beliebigem Register zu A / B

Bus parallele Leitungen von 4 Registern zu A / B

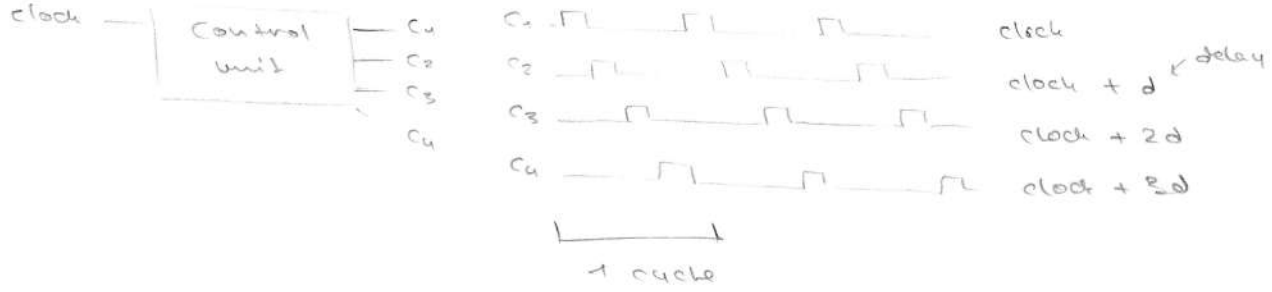
A - Bus : nach A

B - Bus : nach B

S - Bus : von Shifter zurück



Control Unit



Ablauf:

C_1 : Microinstruction laden
Daten auf A/B-Bus legen
Auswahl ALU-Function, Shift

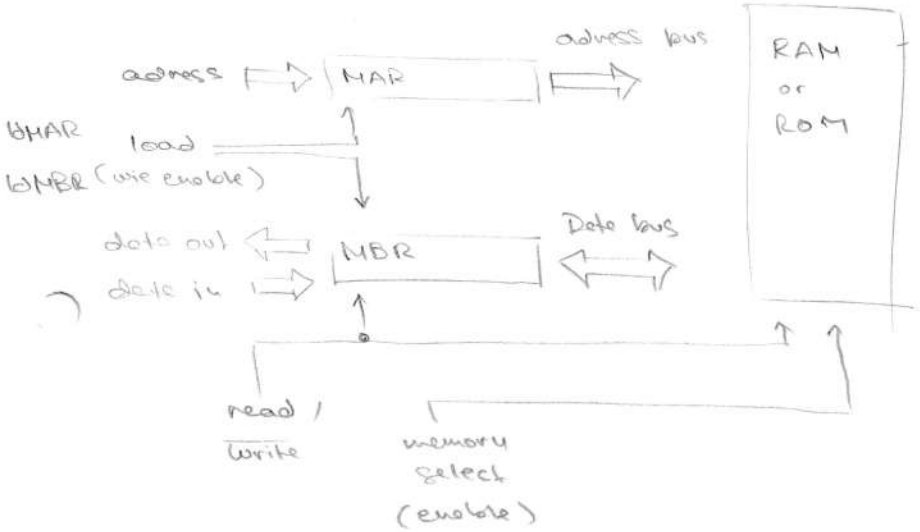
C_2 : Register A/B mit Daten laden

C_3 : ALU und Shifter Zeit geben

C_4 : Ergebnis von S-Bus in das Zielregister laden

Speicheranbindung

MAR Memory Adress register
MBR Memory Buffer register



Speicherzugriff

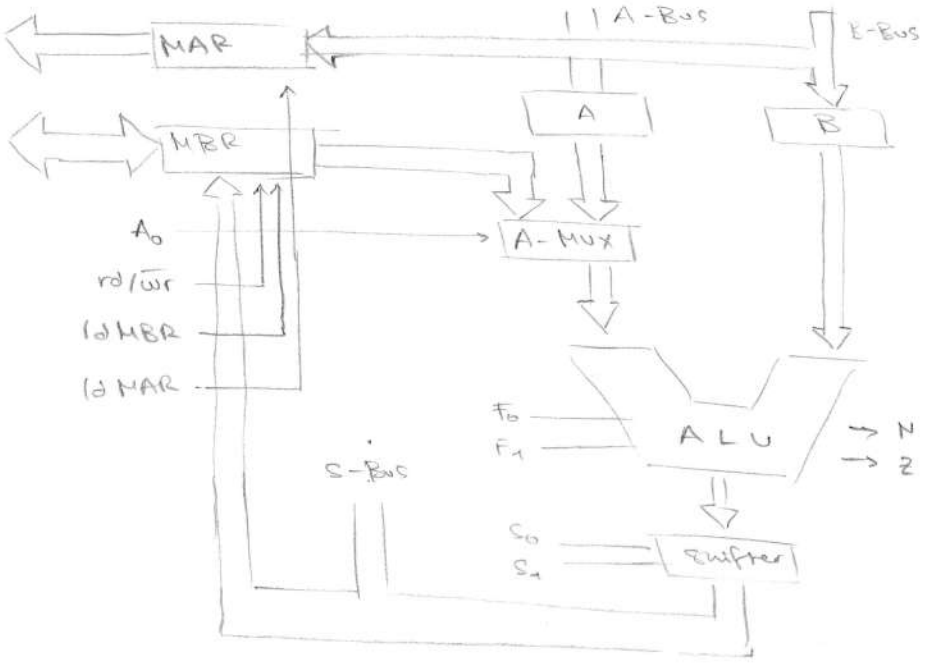
- ~~Handshake verfolgen (zu kompliziert)~~
- Speicherzugriffszeit abwarten (genau 2 Taktzyklen)

1. Adresse in MAR laden
2a) schreiben: MBR befüllen
2b) lesen: MBR inhalt in Register speichern

2a) schreiben: load für MAR und MBR auf 1 stellen
read / $\overline{\text{write}}$ auf 0

2b) lesen: load nur für MAR auf 1
read / $\overline{\text{write}}$ auf 1

ALU Schnittstelle zu RAM / ROM

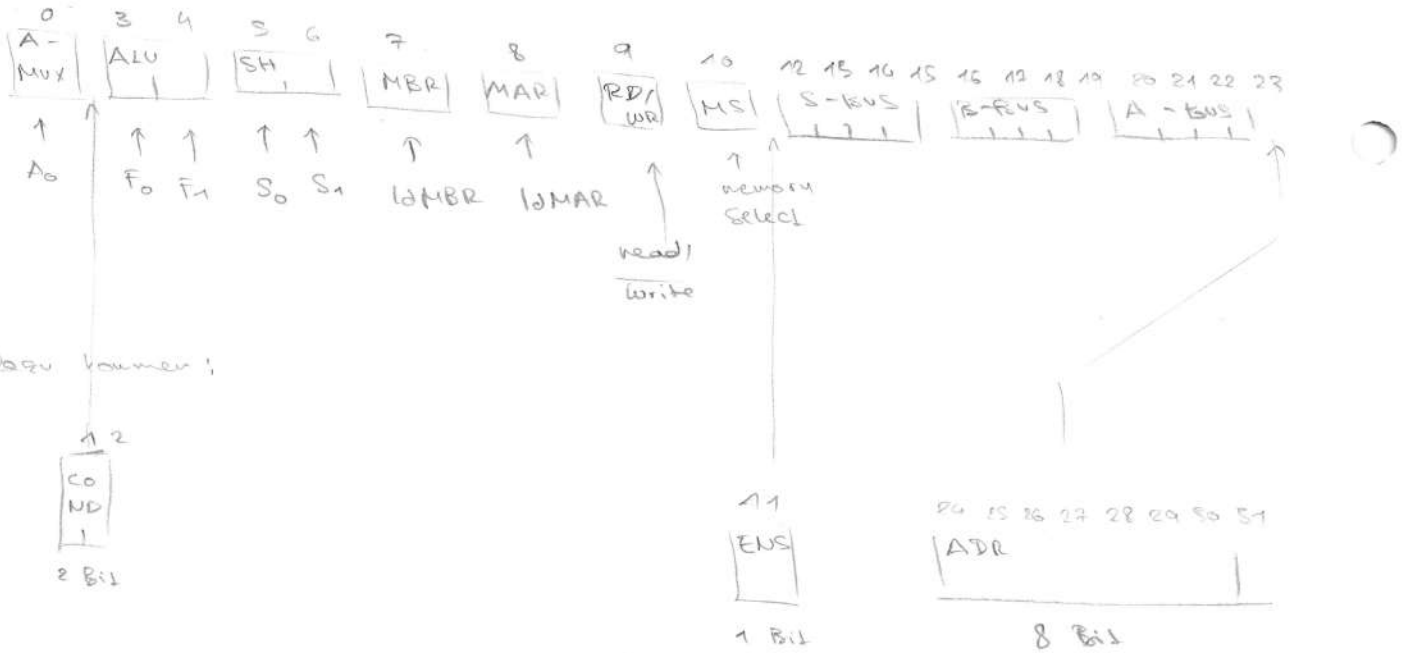


Man sieht:

- MAR wird immer mit E-Bus Eingang gefüllt.
- A-Mux entscheidet ob MBR output oder A-Bus nach ALU geschrieben wird
- S-Bus schickt immer nach MBR

Micro Instruction Register

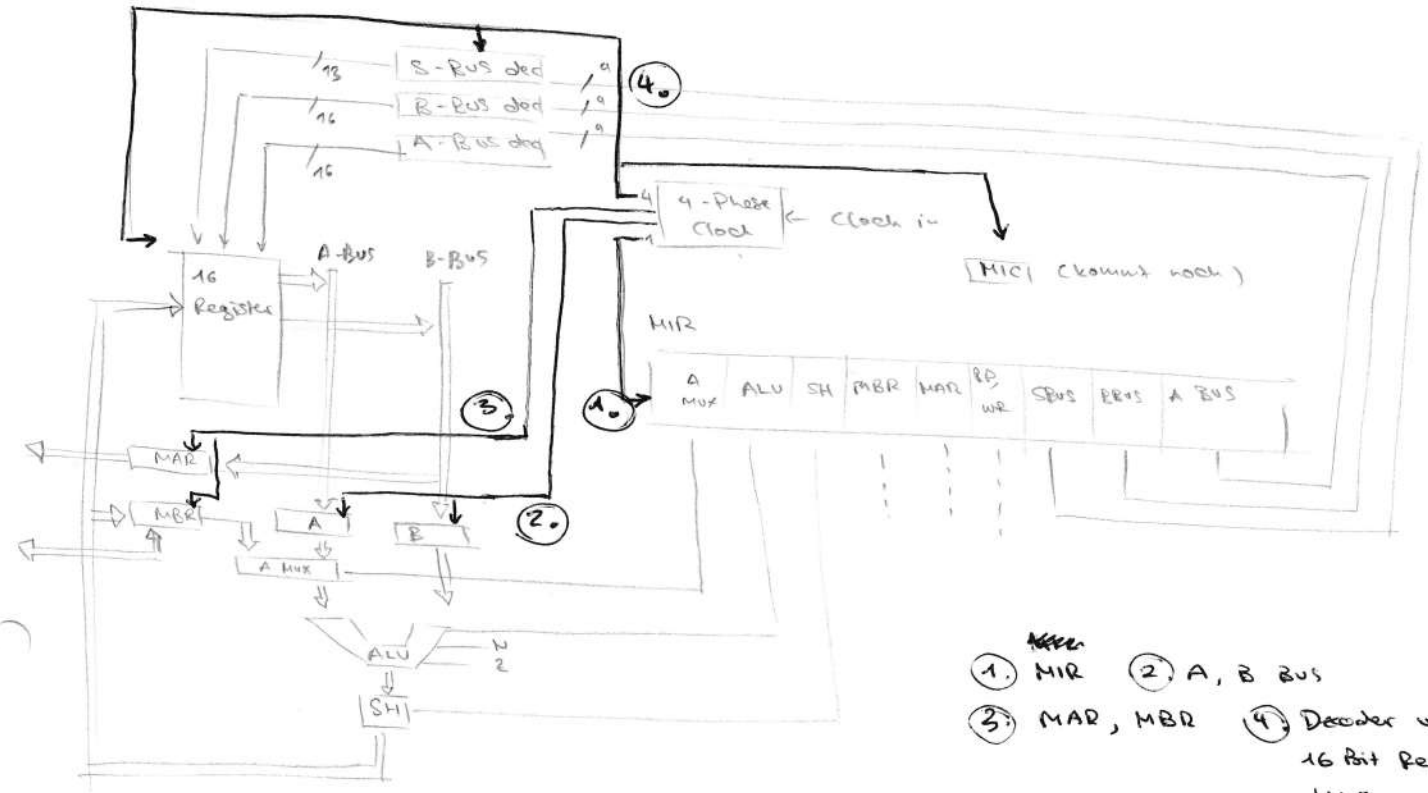
(Wir lassen Carry Bit von ALU aus, damit wir genau 32 Bit haben)



Dazu kommen:

$|[0;31]| = 32$

Vereinfachte Micro 16 Architektur (bisher - unvollständig)



1. MIR
2. A, B Bus
3. MAR, MBR
4. Decoder und 16 Bit Register, MIC

Dazu kommen:

256 x 32 Bit control Store
Mikroprogrammspeicher] Micro-Code ROM

ADR

8 Bit → $2^8 = 256$ Adressen des Speichers

ENS

Enable S-Bus-Decoder

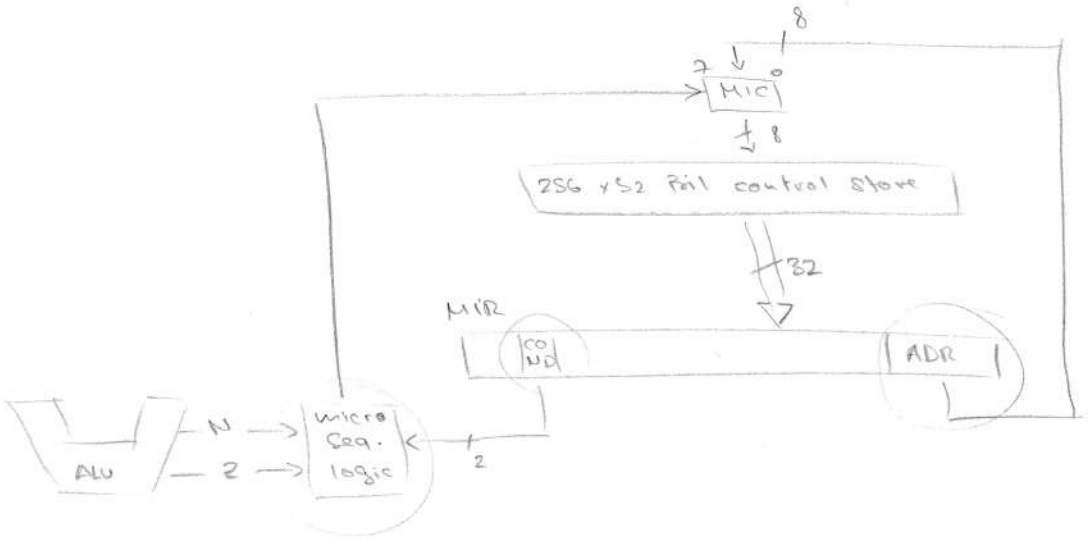
COND

2 Bit → Conditions für micro sequencing logic

00	kein Sprung	–
01	Sprung zu ADR in ROM wenn N=1	if N goto ADDR
10	Sprung zu ADR in ROM wenn Z=1	if Z goto ADDR
11	Sprung zu ADR in ROM	goto ADDR

MIC - Micro instruction controller

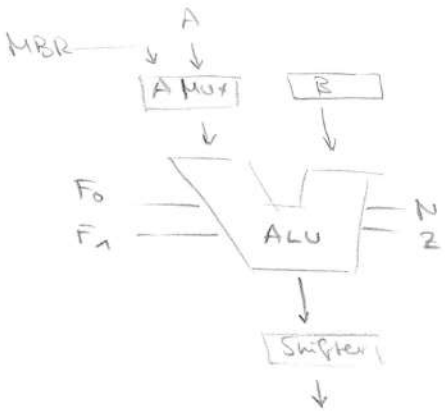
Gibt Input an Micro Code ROM



MIC wird immer mit einer ADR geladen,
micro seq. logic bestimmt aber auf Grund von N, Z
ob zu dieser Adresse gegangen werden soll

Micro Instructions für die ALU

F_0 F_1		
0 0	A durchschalten	$R \leftarrow A$
0 1	A + B	$R \leftarrow A+B$
1 0	A ∨ B Bitwise	$R \leftarrow A \wedge B$
1 1	¬A	$R \leftarrow \neg A$



Shifter

S_0 S_1		
0 0	keine Änderung	$SH \leftarrow R$
0 1	shift left	$SH \leftarrow lsh(R)$
1 0	shift right	$SH \leftarrow rsh(R)$
1 1	ungültig	—

Speicheranbindung

MAR : Memory address register
MBR : Memory buffer register

Lesen:

$MAR \leftarrow R_i$; rd
rd
$R_k \leftarrow MBR$

Zum lesen:

read / $\overline{write}$ = 1
memory select = 1
ADR ⇒ MAR und load MAR = 1
Bits von MBR holen

Schreiben:

$MAR \leftarrow R_i$;
$MBR \leftarrow R_j$; wr
wr

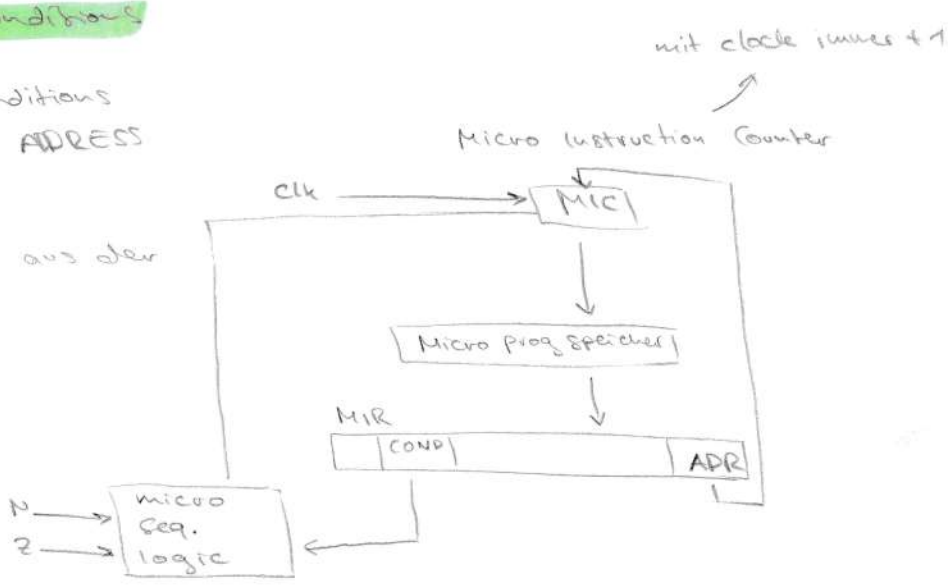
Zum schreiben

read / $\overline{write}$ = 0
memory select = 1
ADR => MAR und load MAR = 1
Daten => MBR und load MBR = 1

Micro Sequencing logic - conditions

Entscheidet basierend auf conditions und N,Z ob MIC wirklich zu ADRESS gehen soll oder nicht

(führt ansonsten die command aus der nächsten Zeile aus)



COND

0 0	—
0 1	[if N goto ADDR]
1 0	[if Z goto ADDR]
1 1	goto ADDR

Als Reihfolge : 32 Bits

(memory select)

A-MUX	COND	ALU	SH	MBR	MAR	RD/$\overline{WR}$	MS	ENS	S-BUS ADR	B-BUS ADR	A-BUS ADR	ADR
1	2	2	2	1	1	1	1	↑ 1	4	4	4	8

ENS: Enable S-Bus Decoder

Lese-Zugriff : ENS = 0

Schreib-Zugriff : ENS = 1

15 Prozessorarchitekturen, Befehlssatz

Prozessoren haben eigene Architekturen
von diesen leiten sich Befehlssätze ab

Mit Befehlssätzen schreibt man Programme (Mikroprogramme → in Mikro 16: ROM)

Ziel:
mächtigere erweiterbare Befehlssätze (zB Multiplikation) → Mehr Maschinenbefehle
Interpreter im Mikrocode dafür benötigt!
die mehr können als Prozessor

Gen 5: Very high level Language VHLL
Gen 4: Fourth gen Language (anwendungsbezogen)
Gen 3: Höhere Programmiersprachen
Gen 2: Assemblerbefehle
Gen 1: Binäre Instruktionen

Gen 3 bis Gen 1:) compiler
Gen 2 bis Gen 1:) interpreter

Abhängig von Prozessorarchitektur

Maschinenbefehle

Arithmetische Befehle
Addition, Subtraktion, Multiplikation, Division, Inkrement, Dekrement
Arithmetic Shift

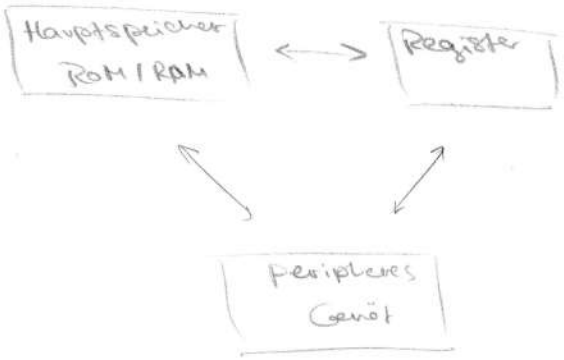
Logische Befehle
AND, OR, XOR, Invert

Schiebebefehle
Shift, rotieren

Datentransferbefehle
MOV, I/O Befehle → Daten aus Peripherals durch Ports, Stackmanipulation

(Flow Control) Kontrollbefehle
Funktionsaufrufe, Sprünge

Befehle für Datentransfer



- MOV
- Datenwort kopieren, zB String und unadressierten

- PIO (Programmierbare I/O)
  Peripherie wie I/O behandelt
- DMA (direct memory access)
  Eigener Controller verwaltet I/O Zugriffe
  CPU setzt nur DMA-Kommando ab
  Interrupt nach Beendigung

Adressierung

- Register Mode R1 ← R2
- Immediate Mode R1 ← 1024
- Direct accessing Mode : goto 128
  RB ← memory $[(500)_{16}]$
- Register-indirect Mode : R6 ← memory [R5]
  ← Pointer
- Programm-Counter-Relative-Adressing-Mode
  oder auch Pointer und Displacement
  $[(500)_{16} + R5]$
  ↑
  ADD (R5) +
- Indirect Adressing Mode
  R6 ← memory [memory $[(500)_{16}]$]

Befehle für Datentransfer: Zwischen Register und Stack

Für Stacks wird RAM benutzt und mit Pointer auf Stack head gezeigt
(Alte Elemente nicht gelöscht)

push-16 (R1)
1. MAR ← SP (Stackpointer)
2. MBR ← R1; wr
3. wr
SP ← SP + (-1)

push-16 (R3)
← speichert Ereignis in R3

Control flow Befehle

Man unterscheidet zwischen bedingten & unbedingten Sprungen

↑

branch if zero flag (boolean)
branch if equal
:

- Funktion / Prozedur / Unterprogramm
  Fasst code zu einer Einheit mit input und output zusammen

  Zu merken:
  - Wo ist Funktion
  - wohin weiter nach Funktion
  - Argumente
  - Rückgabewert (auf Stack oder Register)

  Vorgehen:

  Call subroutine
  1. Registerinhalte und PC (Programm Counter Register) speichern (ZB auf Stack)
  2. PC ← Startadresse der Funktion
  3. Abarbeiten der Funktion

  PC = Return Adress

  Return from subroutine
  1. Registerinhalte & PC wiederherstellen
  2. Mit PC wert fortfahren
  3. Ablauf fortsetzen bis Programmende

Ein Stack kann dazu dienen lokale Variablen und Aufrufhierarchie zu speichern.
Gefahr: Stackoverflow / Stackunderflow (<0)

Ermöglicht rekursive Aufrufe,

Interrupts

asynchrone Unterbrechung des Programmablaufs
ausgelöst durch spezial. Hardware: Interrupt Controller

Auslöser: INTERN (Software) → Exception
- Division by Zero
- Illegaler Befehl
- Stack overflow

EXTERN (Hardware) → Error
- Keyboard Taste gedrückt
  (damit Prozessor nicht ständig warten muss auf keyboard)
- Speicherzugriff von DMA beendet

Eigene Routinen für verschiedene Interruptions
Interrupt Service Routine - ISR
für Fehler-Behandlung

Fehler (Error / Exception) selbst erkennen oder durch codierten Interrupt via Datenbus error-Type erhalten

Wichtig: Interrupt-Adresse ist immer an der selben Stelle und wird mit Interrupt-Vektor bestimmt.

- Prozesse müssen möglichst kurz sein
- Interrupts haben Prioritäten (falls sie von einem anderen Interrupt unterbrochen werden)
- Sie können auch aufgeschoben / verzögert werden in spezifischen wichtigen code Teilen
  ↳ Dann: flag setzen → Interrupt pending als Status im Statusregister.

2 Wichtige Architekturen:

RISC Reduced instruction set cpu (processor) → Bessere Taktraten
- Schnelle Zugriffe auf Speicher um Routinen auszuführen, wenige Befehlssätze
- Weniger Adressierungsarten
- Viele Register
- Speicherhierarchien

→ ARM (Advanced RISC Machines)
MIPS

CISC Complex instruction set cpu
- Gegenteil von RISC → Langsamer und nicht alle Befehle oft benützt
- Unterschiedl. Datentypen und Adressierungsarten
- Befehle haben unterschiedliche Ausführungszeiten
- Steuerung durch Mikroprogramm

→ IBM
Intel

16 Pipelining

MIPS Microprocessor without interlocked pipeline stages

Wiederholung:

- Maschinenbefehle zur Erweiterung der Prozessorfunktionalität
  (mehr als Mikroinstruktionen)

  Beispiel: Stack, dient um Rücksprungadressen bei Funktionsaufrufen zu speichern

- Interrupts
  Synchron oder Asynchron

Speicher-Adressierung

Direkt (immediate)

R1 ← memory [0xCAFE]

Indirekt

R2 ← 0xCAFE
R1 ← memory [R2]

Prä und Post Inkrement / Dekrement

R1 ← memory [(R2) +]

↑ egal ob prä oder post: Danach immer gleich

Taktung bisher in Micro16

1. ⎍⎍⎍ (MIR)
2. ⎍⎍⎍ (REG)
3. ⎍⎍⎍ (Adr)
4. ⎍⎍⎍ (WB)

⎍⎍⎍ „4 Phasen Rechenwerk"

← 2ns →

Je nach Befehl kann eine der 4 Phasen redundant sein:

- 4 Phasen / Takte pro Befehl
- 2ns Taktung
- 8ns pro Befehl
- 125 Mio Instr. pro Sek (MIPS)

Unnötige Phasen lassen sich nicht überspringen

Segmentierung von MIPS Datenpfad

IF	Instruction Fetch
ID	Instruction Decode and source register read
EX	Execute
MEM	Memory access
WB	Write back

Dadurch: 5 Stufige Pipeline, 2ns Taktung

inklusive dem Einlaufen brauchen 6 Befehle (ohne Hazards) 10 Takten.

2 ns - Taktung

Absolute Ausführungszeit

6 Befehle in 10 Takten je 2ns → 20ns

Realer Durchsatz

6 Befehle, 20 ns → $\frac{6}{20 \cdot 10^{-9}}$ = 300 MIPS

oder 3,33 ns / Befehl

Theoretischer Durchsatz

1 Befehl, 2 ns → $\frac{1}{2 \cdot 10^{-9}}$ = 500 MIPS

(kein Einlaufen)

Performance Verbesserung:

- Ausführungszeit für einzelne Befehle im Datenpfad bleibt gleich: System selbst wird nicht in Verarbeitung schneller
- Durchsatz verbessert sich im best case k-fach

Bottlenecks / Begrenzungen:

- Balance der Pipeline Stufen
  sie sollten alle möglichst gleich lange brauchen
- Einlaufen der Pipeline
- Hazards
  - Datenabhängigkeiten
  - Kontroll Strukturabhängigkeiten
  - Strukturelle Abhängigkeiten (nicht in MIPS – keine Architekturfehler)

16 Pipelining : Parallelverarbeitung zur Beschleunigung

Bisher aus Micro-16 Control Unit:

1. (MIR) Befehl vom MIR fetchen
2. (REG) Register A,B für ALU enablen
3. (Adr) MAR und MBR enablen
4. (WR) S-Bus Decoder, Scratchpad / Register Files und MIC (micro instr. counter) für nächste Mikroinstruktion aktivieren

Angenommen wir haben das Micro16-Programm:

Phasen die beansprucht werden:	(1)	(2)	(3)	(4)	
RG ← RG + RS	✓	✓	x	✓	(braucht MAR und MBR nicht)
MAR ← R4; rd	✓	✓	✓	x	(schreibt nichts in Speicher)
rd	✓	x	x	x	(wartet nur)
R2 ← ISH (R1)	✓	✓	x	✓	(braucht MAR und MBR nicht)

Bei Micro-16:

4 Phasen Rechenwerk mit 4 Takter pro Befehl (4 µs) → 8 µs pro Befehl
bei 2µs Taktung (4 · 2µs)

(→ 125 Mio instruct. p. sec (MIPS)
Flops = floating point operation

Ermöglichung der Parallelisierung durch neue Architektur:

MIPS: Microprocessor without interlocked pipeline stages

- Eigene Instructionen: MIPS Assembly language
- 5 Stages:

1. IF Instruction fetch
2. ID Instruction decode
3. EX Execute / Adress Calculation
4. MEM Memory Access
5. WB Write Back

Befehle aus Vorlesung:

lw $10, 20($1) — R10 ← Memory [R1 + 20]

sub $11, $2, $3 — R11 ← R2 - R3

lw $1, 100 ($0) — R1 ← Memory [R0 + 100] → siehe MIPS Assembler Instructions

Aber nicht alle Befehle beanspruchen alle 5 Pipeline Stages gleich stark!

Befehl	IF	ID	EXE	MEM	WB	Σ
lw	2ns	1ns	2ns	2ns	1ns	8ns
sw	2ns	1ns	2ns	2ns		7ns
add, sub, and, or, slt	2ns	1ns	2ns		1ns	6ns
beq	2ns	1ns	2ns			5ns

(branch if equal)

obwohl < 2ns, beanspruchen sie jedoch zur vereinfachung 2 ganze ns.

Durchsatz:

5 Stufig, 2ns Taktung
Ein Befehl 10ns
insgesamt 10 Takte

Real 3,33 ns pro Befehl

Theoretisch 2ns pro Befehl

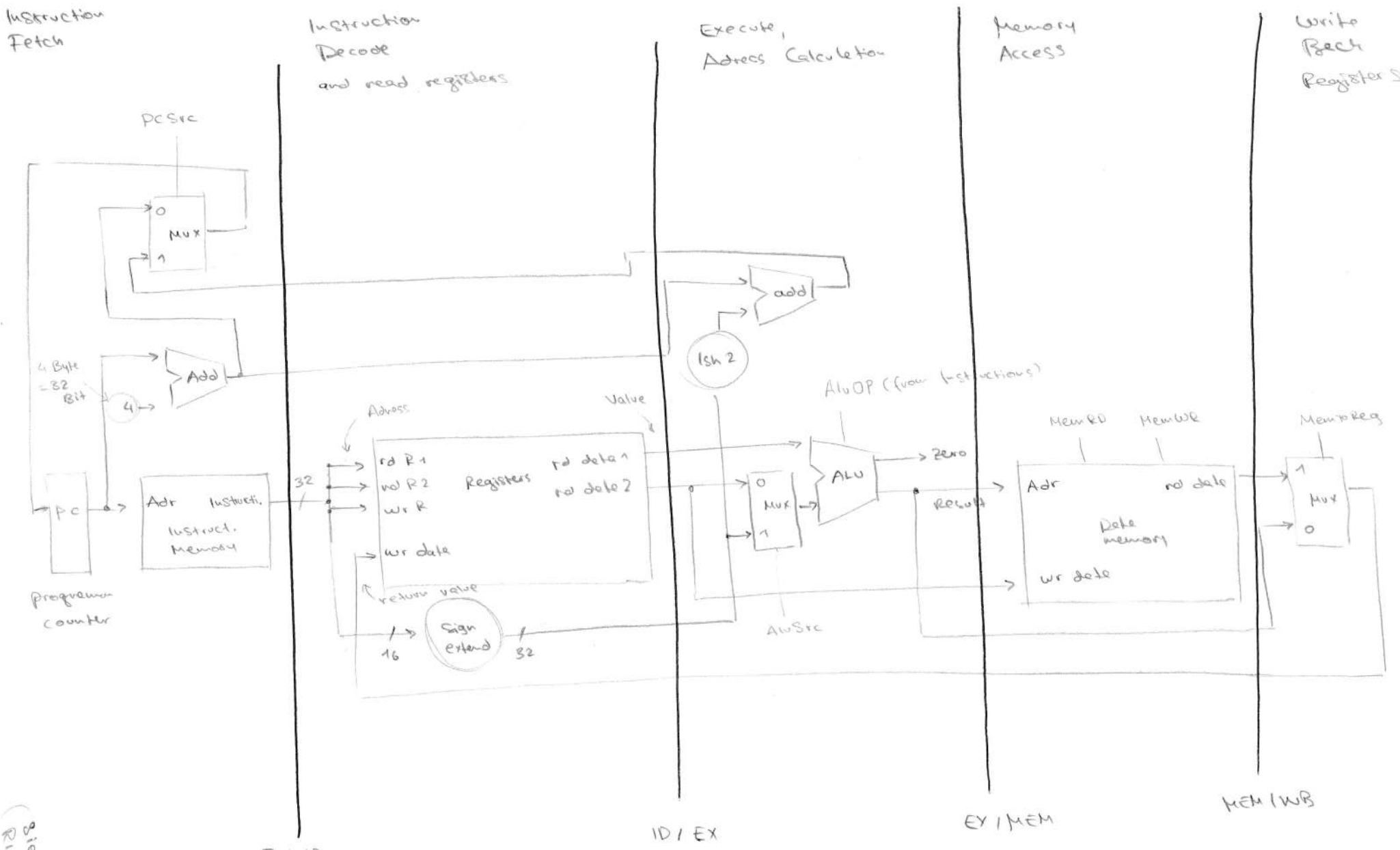
Wichtig: mehr Stufen ≠ mehr performance!
(sinkt ab k=15)
„Super pipelining"

Bottlenecks:
- Auffüllen der Pipelines „Einlaufen"
- Balance der Stufen
- Hazards (Abhängigkeiten)

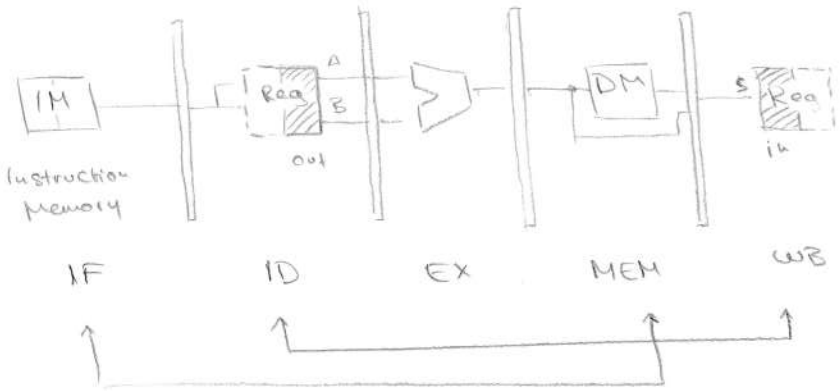
Sind mehr pipeline Stufen besser?

Intel Pentium: doppelte # Stufen (jede Stufe in 2 aufteilen) → doppelter theoretischer Durchsatz durch Verdopplung der Taktfrequenz / Rate

Problem:
deutlich mehr hazards


Instruction Fetch
Instruction Decode and read registers
Execute, Adress Calculation
Memory Access
Write Back Register S
PCSrc
0
MUX
1
add
lsh 2
4 Byte = 32 Bit
Add
4
Adress
Value
ALUOP (from Instructions)
MemRD
MemWR
MemtoReg
rd R1
rd R2
wr R
Registers
rd data 1
rd data 2
wr data
return value
PC
Adr
Instructi.
Instruct. Memory
32
0
MUX
1
ALU
Zero
Result
Adr
rd data
Data memory
wr data
1
MUX
0
16
Sign extend
32
ALUSrc
Programm counter
IF / ID
ID / EX
EX / MEM
MEM / WB
RISC-V diese

Erhöhte Abstraktion:



Dependencies:

- Gleicher Speicher für Write-Back und Instruction Decode (Annahme: in der ersten Taktthälfte schreiben, dann Lesen)
- Gleicher Speicher für Instruktionen und Daten

Hazards

Strukturelle Hazards

Pipeline Stufe nicht für alle Befehle verfügbar

→ Lösung: Architektur Neuentwerfung, Stall

Control Hazards

Nachfolgebefehl hängt von Verzweigung ab

→ Lösung: prediction, delayed branching (code umändern), Stall

Data Hazards

Berechnung abhängig von Vorgänger-Befehl, falsche Ausführungsreihenfolge

→ Lösung: forwarding, Code-Optimierung, Stall

RAW – Read after Write
gelesen bevor richtig beschrieben

$R2 \leftarrow R1 + R3$ (2.)
$R4 \leftarrow R2 + R1$ (1.)

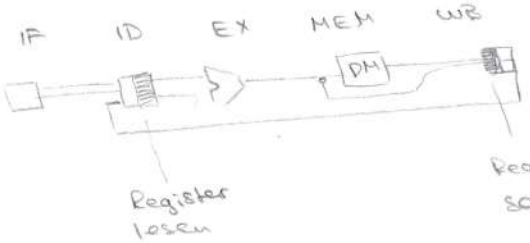
WAR – Write after Read
(überschrieben bevor richtig gelesen

$R2 \leftarrow R1 + R3$ (2.)
$R3 \leftarrow 1$ (1.)

WAW – Write after Write
vom vorherigen Befehl falsch überschrieben

$R2 \leftarrow R1 + R3$ (2.)
$R2 \leftarrow 1$ (1.)

Die Data Hazards entstehen wegen:



Gemeinsamer Registersatz:
1. Takthälfte Schreiben
2. Takthälfte lesen

Beispiel: Stall bei RAW – Read after Write

write $R2 \leftarrow R1 + R3$
read $R4 \leftarrow R2 + R1$
} problem: Read before write

Takt	IF	ID	EX	MEM	WB
1	$R2 \leftarrow R1+R3$				
2	$R4 \leftarrow R2+R1$	$R2 \leftarrow R1+R3$			
3	○	$R4 \leftarrow R2+R1$	$R2 \leftarrow R1+R3$		
4	○	$R4 \leftarrow R2+R1$		$R2 \leftarrow R1+R3$	
5	○	$R4 \leftarrow R2+R1$			$R2 \leftarrow R1+R3$ ausgeführt ✓
6	□	○	$R4 \leftarrow R2+R1$		
7	△	□	○	$R4 \leftarrow R2+R1$	
8	⬡	△	□	○	$R4 \leftarrow R2+R1$

Hazards

Strukturelle Hazards (Hardware Problem) → für MIPS irrelevant
Mehrere Stufen brauchen dieselben Ressourcen → Stalling, Architektur ändern

Control Hazards
Nachfolgebefehl hängt vom Ausgang des Sprungs ab → prediction, delayed branching
Stalling

Data Hazards (am wichtigsten)
Berechnung benötigt Ergebnis des Vorgängers → forwarding, code-optimierung, Stalling

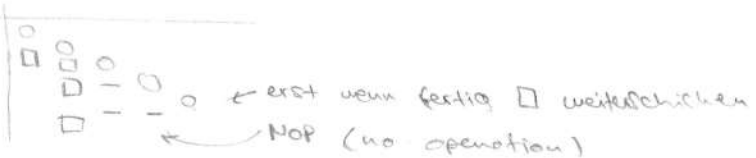
RAW: read after write
1. R2 ← R1 + R3 schreiben
2. R4 ← R2 + R1 lesen → Falsch
lesen bevor Register befüllt wurde mit richtigem Wert

WAR: write after read
1. R2 ← R1 + R3 lesen ← irrtümlich mit R3 ← 1 ausgeführt
2. R3 ← 1 schreiben
Schreiben bevor gelesen wurde

für MIPS irrelevant

WAW: write after write
1. R2 ← R1+3
2. R2 ← 1
vom Vorgänger überschrieben

Stall / Pipeline Stall
Man lässt bubbles entstehen → Entweder durch Architektur oder Compiler



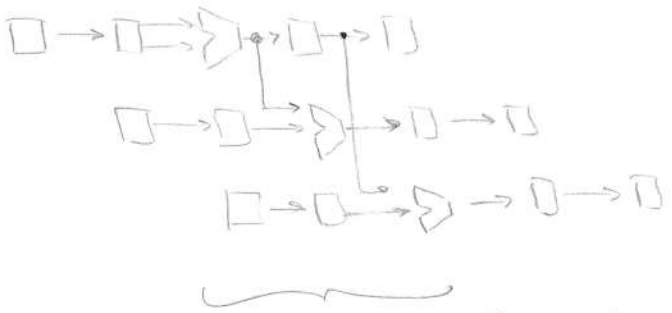
Delayed branching
Branch Befehle zurückschieben und in Zwischenzeit andere Befehle exekutieren

Code Optimierung
Dort wo keine Abhängigkeiten, Delay erzeugen und andere Befehle währenddessen machen

Data Forwarding

```
add  $t0, $t1, $t2
sub  $t3, $t0, $t1
or   $t4, $t0, $t2
```

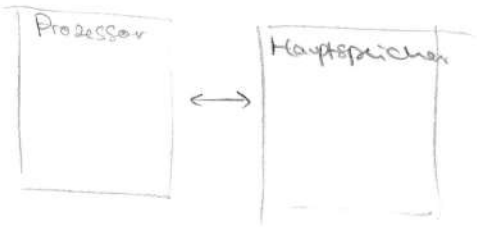
R0 ← R1 + R2
R3 ← R0 + R1
R4 ← R0 + R2



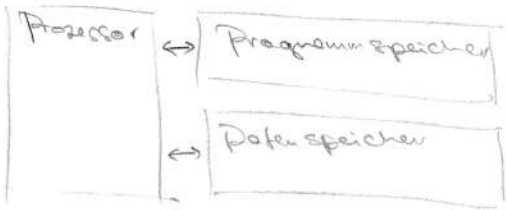
leitet Daten sofort weiter um Hazards zu vermeiden

17 Speicherverwaltung, Caches und Hierarchien

Von-Neumann-Architektur

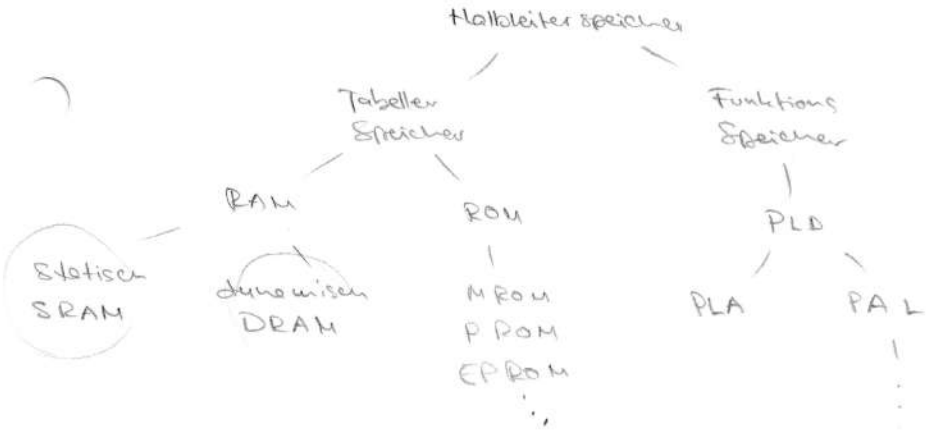


Harvard Architektur



(Bei Micro 16:
Microinstruktionen - Programmspeicher
Datenspeicher als ROM / RAM mit MAR)

Speicherbausteine (Wiederholung)



SRAM
Speichert in Latches
\+ Sehr schnell (kurze Zugriffszeiten
\- sehr teuer

↓
Ziel: Preis - Performance - Balance

DRAM
Speichert in Kondensatoren
hat refresh-cycles

Beobachtung:

Temporal locality - Oft dieselben Speicherzugriffe hintereinander

Spatial locality - Oft benachbarte Speicherzugriffe hintereinander

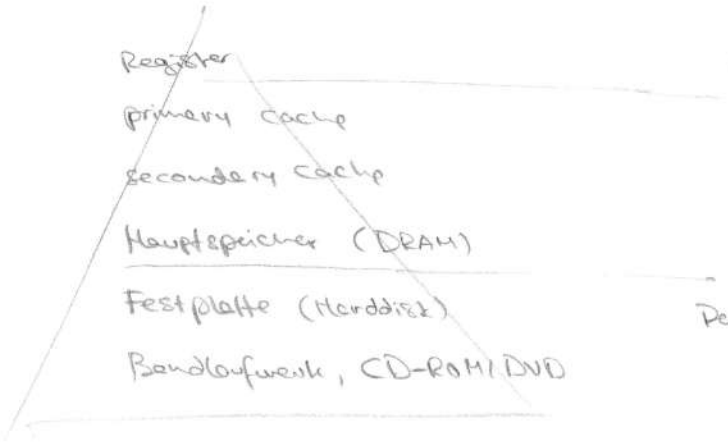
Lösung

CPU
↕
Speicher Hierarchie

Dadurch erzeugt man gepufferte Speicherinhalte

↑ Zugriffszeit
↓ Volumen

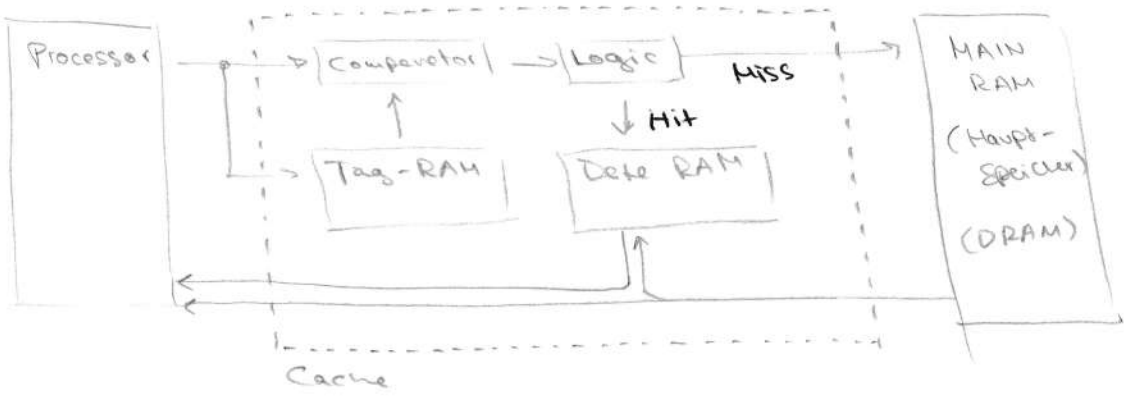
Typische Hierarchie



CACHE (SRAM)
↕
HAUPTSPEICHER (DRAM)

Cache

Schneller Zugriff der CPU auf Daten / Befehle



Cache-Performance

$t_{eff} = h \cdot t_{cache} + (1-h) \cdot t_{main}$

t_{eff} ... effektive Speicherzugriffszeit
h ... Trefferquote (Hit-Rate)
t_{cache} ... Zugriffszeit auf Cache
t_{main} ... Zugriffszeit auf Main

$t_{AMA} = t_{hit} + m \cdot t_{penalty}$

t_{AMA} ... Ø Zugriffszeit
t_{hit} ... Hit time
m ... Miss Rate (1-h)
$t_{penalty}$ ... Miss penalty

↓
Befüllen bei Miss
↓
1. Stall der Pipeline
2. Adresse an Speicher management
3. Daten von Main auf Cache

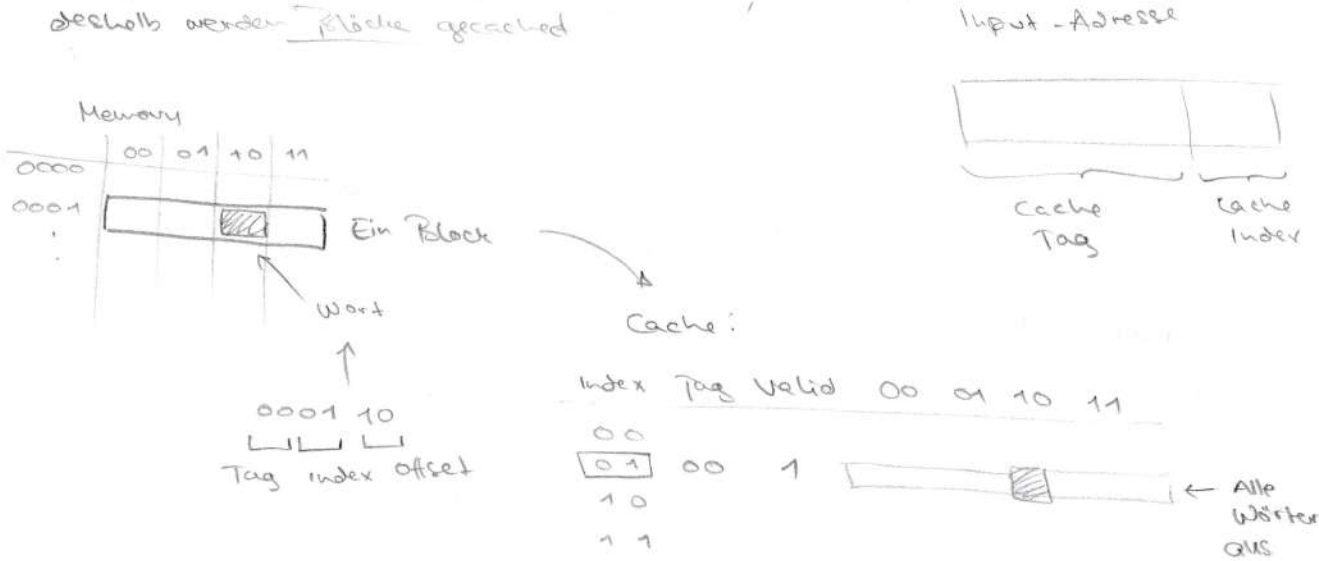
Wichtige Parameter:
- Hit-Time
- Hit-Rate → Miss-Rate = 1 - Hit Rate
- Miss-Penalty

Cache Performance wichtig, weil sonst bottle-neck für CPU

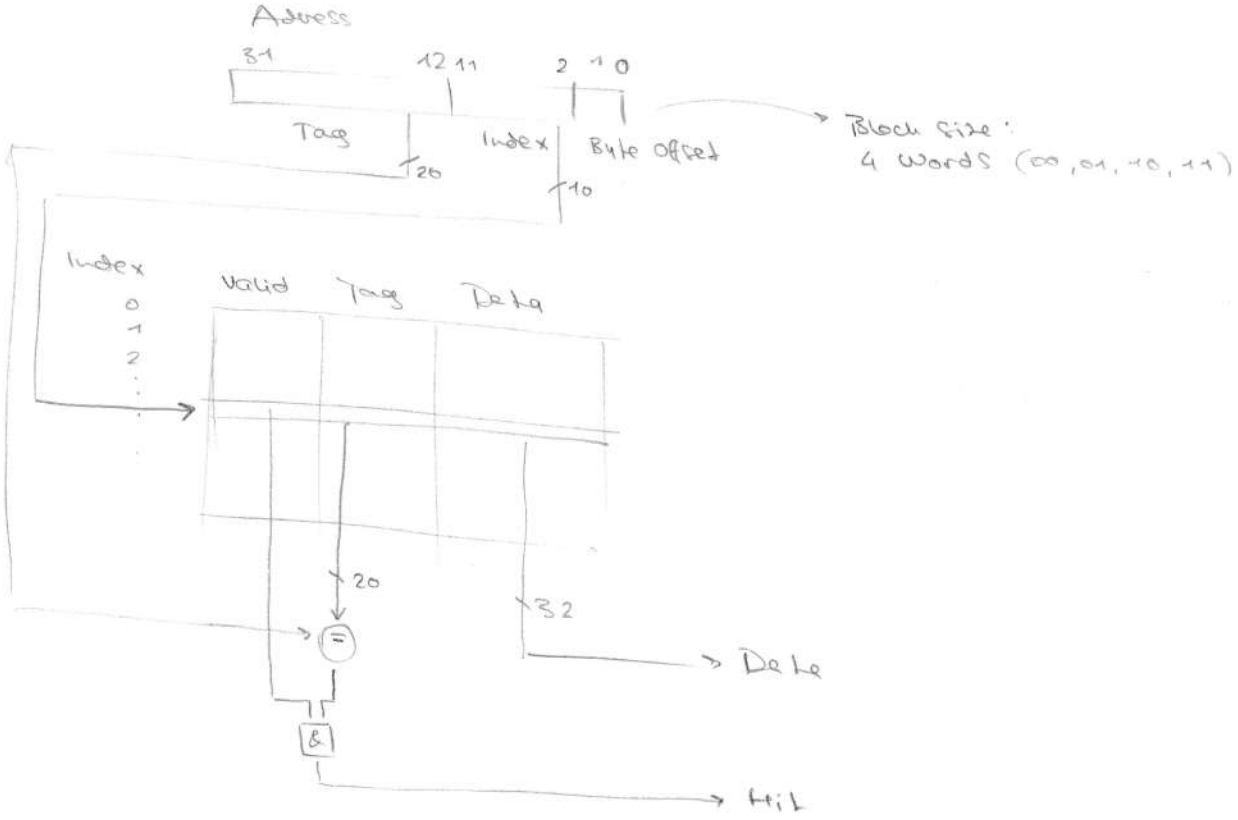
Cache Verwaltungsstrategien

Direct Mapped

- Ein Teil der Adresse bestimmt Cache Position (wie Modulo Rechnung)
  - ↳ niederwertige Bits: „Cache-Index"
- Restlichen Bits zur eindeutigen Identifikation als „Cache-Tag"
- „Valid-Bit" zur Erkennung bisher unbenutzter Positionen
- oft ineffizient einzelnes Wort / Bytes zu cachen, deshalb werden Blöcke gecached



Detailierter:



Fully associative Cache

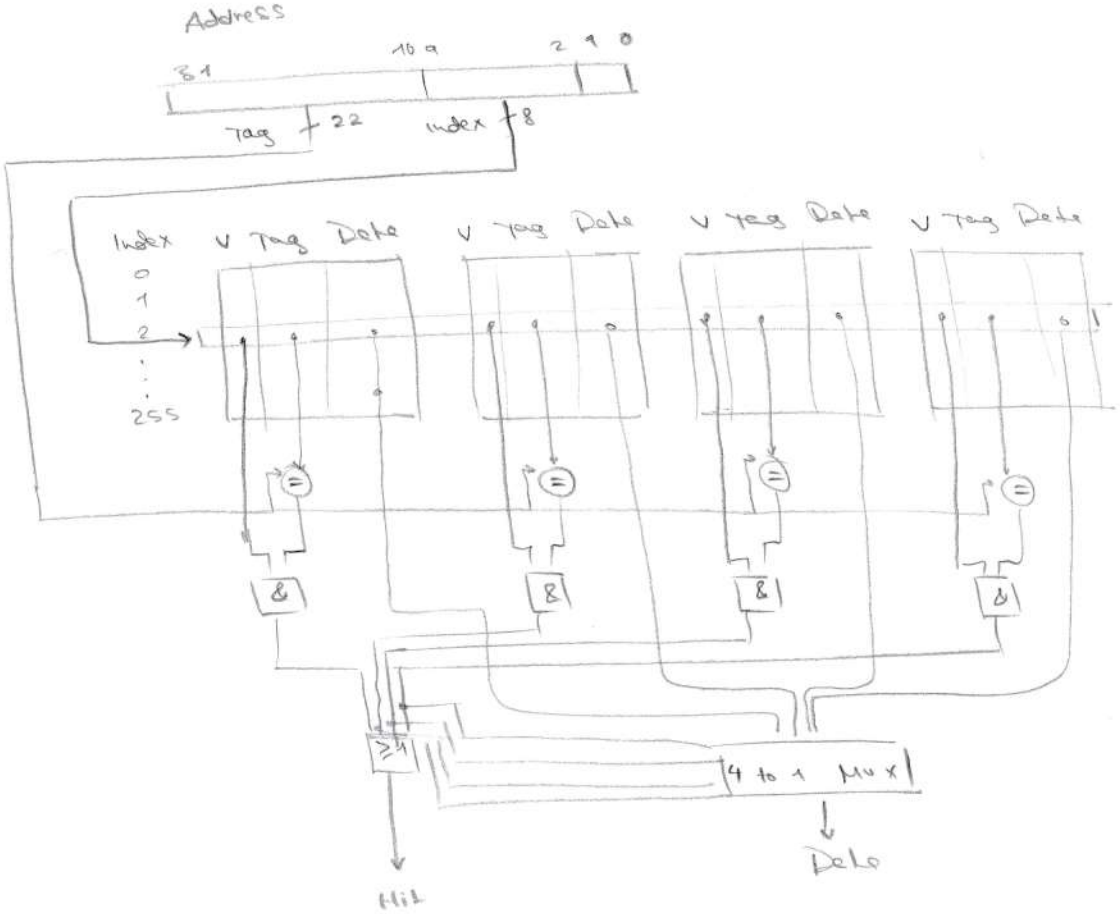
Jeder Block darf in jeder beliebigen Position im Cache abgelegt werden

(komplexester Aufbau)

↳ kein Index → volle Adresse als Tag

N-way Set associative Cache

Jeder Block bekommt einen Set zugewiesen der fully associative ist



Blöcke = # Sets · # ways

Index ways

Cache Verwaltungsstrategien im Vergleich

Direct-mapped cache

+ Eindeutige Speicher-position im Cache für jeden Block
+ einfache Cache Verwaltung ohne Multiplexer
~ mäßige Hit-Rate

Fully associative Cache

- Block kann überall sein im Cache (Suche aber mit XOR Block! nicht einzeln)
- komplizierte Verwaltung, großer Multiplexer
+ optimale Hit-Rate

N-way set associative Cache

+ Suche beschränkt → Pro Block gibt es N mögliche Positionen (im Set)
+ Einfache Ersetzungsregeln (LRU)
+ handhabbarer Multiplexer
~ vernünftige Hit Rate

Alle caches beinhalten:

Verwaltungsdaten:
- Tag
- Validbit
- Dirty bit
- Bits für Ersetzungsregeln

Nutzdaten

Beispiel: Assoziativität bei 8 Einträgen:

1-way set associative
(= direct mapped)

	Way 0	
Set	Verw.	Data
0		
1		
2		
3		
4		
:		
7		

8-way set associative
(= fully associative)

	Way 0			Way 7
Set	Verw. Date	Verw. Date	...	Verw. Date
0				

4-way set associative

	Way 0		Way 1		Way 2		Way 3	
Set	ver	D	V	D	V	D	V	D
0								
1								

2-way set associative

	Way 0		Way 1	
Set	V	D	V	D
0				
1				
2				
3				

Umgang mit Hit & Miss bei Write-Zugriffen

Bei Lesezugriffen:

Hit → fertig

Miss → nachladen und noch Regeln ein Set, Index überschreiben

Bei Schreibzugriffen

Hit → Cache und DRAM müssen konsistent sein

Miss → Nachladen optional

Write-Hit

(WT) Write through

aktualisiere cache und DRAM gleich dazu

\+ Datenkonsistenz immer gewährleistet

\- Häufige Zugriff auf DRAM → Performance Verlust

(CB) Copy Back / write back

aktualisiere cache und markiere Cache-Block mit „dirty" tag,
aktualisiere Hauptspeicher wenn Block auf Cache entfernt wird

\- Datenkonsistenz nicht gewährleistet

\- Read miss wird langsamer wenn Blöcke entfernt werden

\+ Seltenere Zugriffe auf DRAM → Performance Gewinn

Buffered write through

(Vorteile von WT und CB)

neue Werte werden in Cache und zweiten schnellen Zwischenspeicher eingetragen,

falls Puffer voll (nicht wenn cache Block entfernt wird), muss Prozessor warten

Write-Miss

Write-Around

Schreibe direkt in Speicher bei miss

(meist mit WT verwendet)

Fetch-on-write

Fülle ein Feld cache und lade restlichen Blockwörter aus DRAM nach

(Simuliert dann quasi write hit, danach)

Verwaltungsdaten:

- Valid Bit
- Dirty Bit
- Ersetzungsstrategie Bits

Cache Replacement Strategien

Auswahl zum ersetzen:

- Direct Mapped Cache : Eindeutig
- Set Associative Cache : Innerhalb von Set
- Fully Associative Cache : beliebige Auswahl

→ braucht replacement Strategie

LRU – Least recently used

Am Längsten (Zeit) nicht verwendet (mit Queue)

– aufwändig zu realisieren

LFU – Least frequently used

Am wenigsten verwendet (mit Counter)

Häufig verwendet → Erst nach vielen Misses ersetzbar

"reference bit"

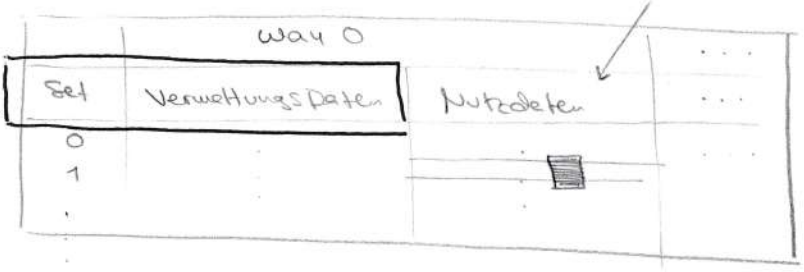
Random

FIFO (first in – first out)

Einfach

Übersicht: Cache - Aufbau:

Ganze Blöcke mit jedem Wort übertragen



Angenommen bei 8 Einträgen:

1 way, 8 sets ——> direct mapped
8 ways, 1 set ——> fully associative
Sonst ——> n-way set associative

} # Blöcke = # sets · # ways
Pro block x Wörter

Index/Set
Spricht Zeile an

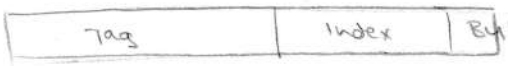
Tag
XXXX[Index]
↑
Macht eindeutig

Valid-Bit
Sagt ob benutzt

Dirty-Bit
Siehe copy back vs write through

Ersetzungsregel-Bits
—

Adress:

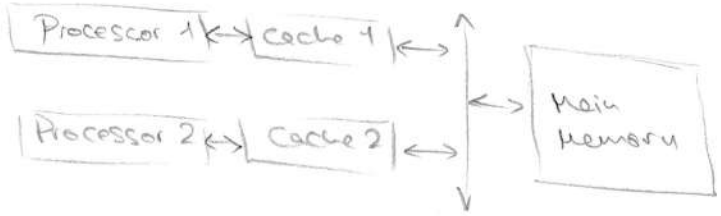


Bestimmt aus Adresse Cache Position

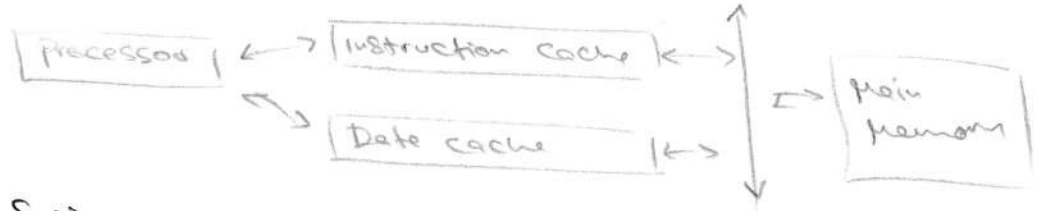
↑
hier: 4 Wörter pro Block: (00, 01, 10, 11)

Exkurs: Multiprozessor-Systeme

Mehrprozess-Systeme mit Caches



Harvard-Architektur



Exkurs: Speicherverwaltung

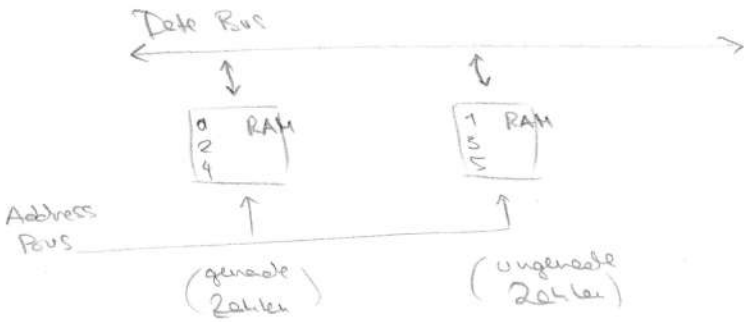
Bus ... früher: Backplanes und Sockets

- Data-Transfer-Bus: zwischen CPU und Memory
  - Address
  - Data
  - Memory
- Arbitration-Bus: für mehrere Nutzer
- Interrupt-Bus

} System Bus

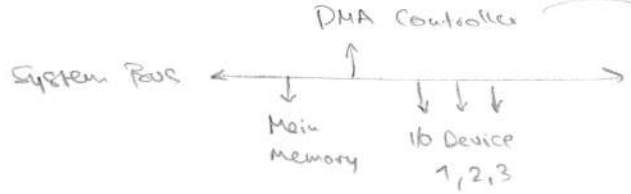
Interleaved Memory

Speicher aufteilen in Segmente um Zugriffe zu beschleunigen



(2-Fach Interleaved Memory)

Direct Memory Access



ist zuständig für Kommunikation zwischen Prozessor & peripheren Geräten

ähnlich wie Bridge verschiedene System-Busse miteinander verbindet

18 Multicore

Clock-Speed und Performance kann nicht mehr steigen.

Deshalb mehrere Cores (CPUs) auf einem Chip → Parallelisierung — schwache Speichermodelle (im nächst. Kapitel)

parallelis. Software | Multicore Systeme

Amdahls Gesetz

über Parallelisierung

t_p parallelisierbare Laufzeit

t_s sequentielle (nicht parallelisierbare) Laufzeit

Laufzeit vor Parallelisierung

$$t^v_{tot} = t_s + t_p$$

Laufzeit nach Parallelisierung

$$t^n_{tot} = t_s + \frac{t_p}{c}$$ ← c threads

$$\text{Speedup } S = \frac{t^v_{tot}}{t^n_{tot}}$$

$$\text{Ausnützung } a = \frac{t^v_{tot} / c}{t^n_{tot}}$$ ← maximale Auslast. / ← echte Auslast.

Erkenntnis:

Große Speedup verbesserung bei gleichem Problem SCHWERER als bei schwierigeres Problemstellung!

↑ höhere Laufzeit

Vektor-Rechner

meist bei Bildverarbeitung benutzt

→ ALU rechnet mit Blöcken (Vektoren) (zB 4 Gleitkommazahlen à 32 Bit)

+ vereinfachte Synchronisation
+ reduzierte Kontrollhardware
+ effizient bei parallelen Strukturen (zB Rastern)
+ erweiterter CPU Instruktionssatz

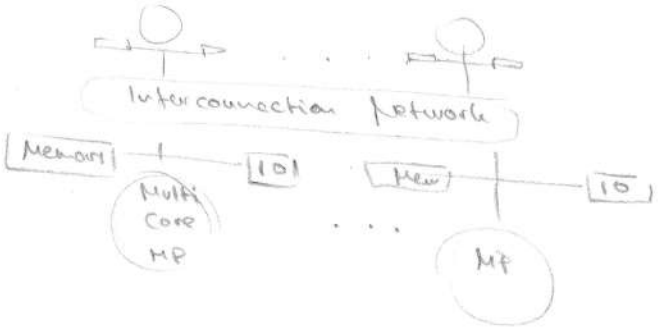
GPU Graphic processing unit (nicht LVA relevant)

(Graphik-Karten)

ganz andere Architektur als CPU → für parallele Berechnungen geeignet

eigene Programmiertechniken

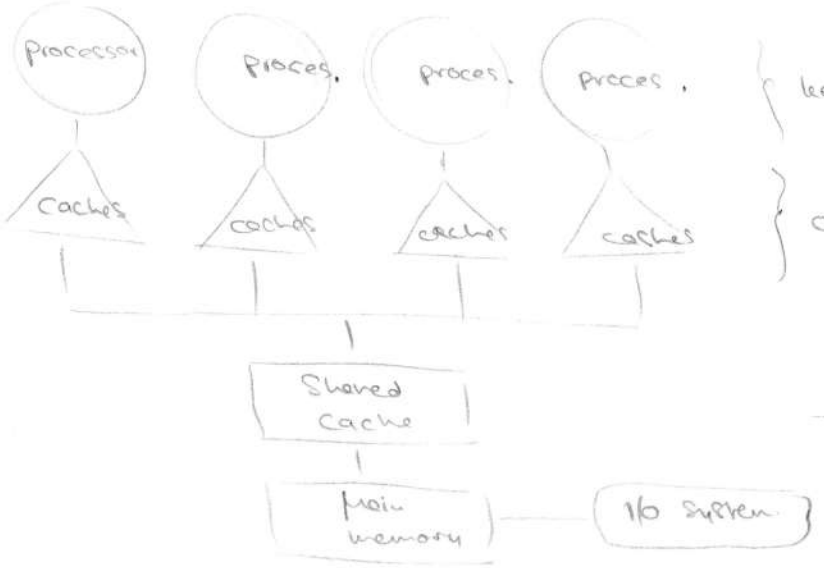
DSM - Distributed Shared Memory



- Ein Speicher per CPU
- Problem: NUMA
  Non uniform memory access / Latenc
  (Speicherzugriff auf einen Speiche
  außerhalb der eigenen CPU)

SMP - Symmetric Multiprocessors

(Für TGI das einzig relevante)



keine # an Processoren

cache Hierarchy

→ Uniform memory access / (g
→ gleicher physikalischer Adressraum für A Processoren

→ Synchronisation mit Shared Variables

Sequentially consistent world:

Interleavings

Alle möglichen Statementreihenfolgen in Threads

Thread 1 (a, c)
Thread 2 (b, d)

→ lokale Ordnung muss erhalten bleiben:

$a < c$, $b < d$

(a vor c), (b vor d)

Aber Threads selbst können an beliebigen Zeitpunkten ausführen:

(a, c, b, d) (b, a, c, d)
(a, b, c, d) (b, a, d, c)
(a, b, d, c) (b, d, a, c)

↓

Verschiedene Scheduling-Strategien (Hardware abhängig)

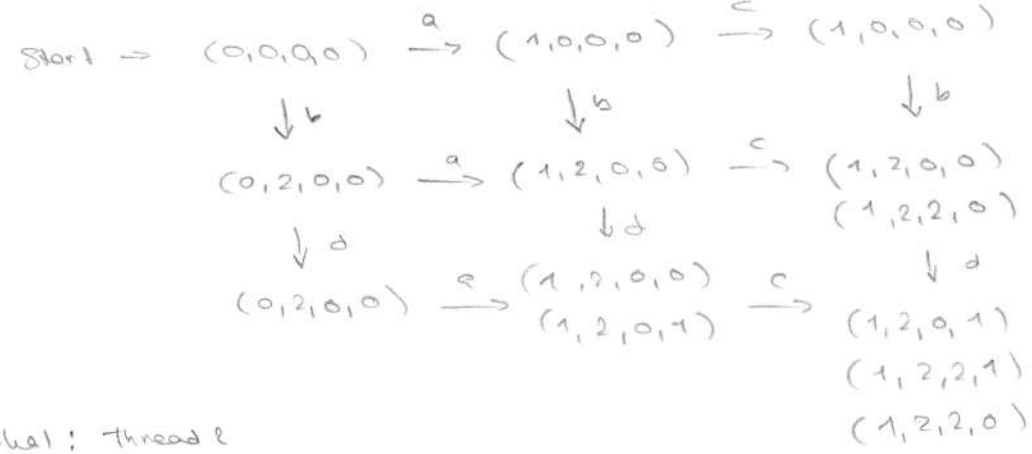
alle mögl.

Interleavingsgraph

$(A, B, C, D) = (0, 0, 0, 0)$

Thread 1:
a: A := 1
c: c := B

Thread 2:
b: B := 2
d: D := A



↓ Vertikal: Thread 2
→ Horizontal: Thread 1

Woran erkennt man dass wir in einer sequentially consistent world sind?

Sequentielle Ausführung innerhalb eines Threads.

Beispiel:
Output kann nie (1, 2, 0, ·) sein, da a immer vor d ausgeführt werden muss.
D

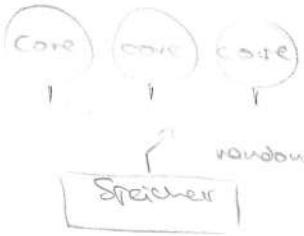
→ SEQUENTIALLY CONSISTENT MEMORY MODEL (SC)

Race Condition (im SC Modell)

Konstellation, in der Output von der zeitlich verschränkten Ausführung Anweisungen abhängt. (bei nebenläufigen Threads)

↳ Setzt voraus, dass Zuweisungen atomar sind

Vereinfachte Erklärung für SC



- Ein einziger globaler Speicher,
  Jeder Core erzeugt Speicheroperationen in Programm-Ordnung
- Speicher Ordnung sequentiell weil Scheduler zufällig einen Core access erlaubt zum Speicher

→ Alle Operationen atomar, weil jeweils global nur eine Operation ausgeführt werden kann

Zusammenfassung:

Das SC Modell setzt voraus:

- atomare Wertzuweisungen
- Ausführung von Thread internem Code nach Programm-Ordnung
  (in modernen Rechnern nicht)

Man kann mit Hardware Hilfsmittel SC gewährleisten.

+ einfach zu programmieren / debuggen
+ einfach Korrektheit zu beweisen
- langsam und nutzt Multi-Core Potential nicht aus.

Atomics

Beispiel: 16-Bit Wort Zuweisung erfolgt nicht atomar: je 8 Bit

1. Thread 1 überträgt 1. Hälfte auf Variable store_here:

   (-1) 1111 1111 1111 1111 → store_here = 1111 1111 0000 0000

2. Thread 2 liest store_here → Falscher Wert! (-128)
3. Thread 1 überträgt den Rest.

Wie kann man Ausführungen atomar machen?

Nur durch Hardware: atomare Operationen als Instruktionen

programmierbar durch atomare Typen und Operationen auf atomaren Variablen in diverser Programmiersprachen

Synchronisation & Kommunikation zwischen Threads (Einfachste Art)

Beispiel: Data von Thread 1 auf Thread 2 ohne Unterbrechung → mit flag

F=1 → Wert gefahrenlos fertig übertragen!

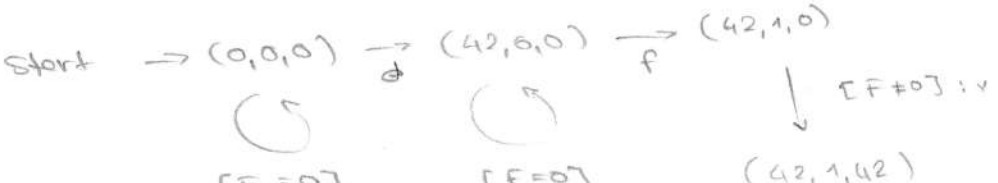
(F ist atomar, weil es nur ein Bit hat)

Anwendung von Atomaren Operationen:

„Producer - Consumer - Pattern"

(D, F, X) = (0, 0, 0)

Thread 1:		Thread 2:	
d:	D = 42	if:	if F=0 then goto if (wait)
f:	F = 1	x:	X = D



Hier keine Operation für F≠0 existent.

Großer Nachteil:

Thread 2 muss so lange warten bis Thread 1 fertig ist.

Wir nutzen also die Vorteile von Multithreading bei diesem Lösungsansatz nicht!

(mehr dazu folgt ...)

Alles bisher war in der Sequentially consistent world mit dem SC - Memory - Mo

Dabei wurde vorausgesetzt dass:

1. - Atomare Wertzuweisungen
2. - Programmordnung = Ausführungsreihenfolge in einem Thread

Aus dem Interleavings - Graphen konnte man sehen, dass:

Mit einem Core / Thread, kann man Ausführungsreihenfolge nicht umordnen

Mit mehreren Cores / Threads, kann Ergebnis unterschiedlich sein

Optimierung im Hintergrund:

Programmordnung PO ≠ Exekutionsordnung EO
(Ausführungsreihenfolge)

Programm wird von Compiler und CPU umgeordnet.
Es soll aber nicht zu einem anderen Ergebnis führen (bei einem Thread / Core)

→ also immer noch sequentially consistent obwohl 2. Bedingung verletzt wurde

Bei Multicore - Rechnern aber:
nicht mehr sequentially consistent!

Beispiel für SC - Verletzung bei Multithreading wobei EO ≠ PO:

Einleitendes Beispiel:

Thread 1:
a: A = 1
c: C = B

Thread 2:
b: B = 2
d: D = A

Ohne Einhaltung der PO innerhalb der Threads:

Thread 1:
c A = 1
a C = B

Thread 2:
d D = A
b B = 2

→ SC ist verletzt, dadurch ist (1,2,0,0) als Endzustand möglich

obwohl es kontraintuitiv ist und zeitliche relativ. verletzt, neben kausalität

→ RELAXED MEMORY MODEL

Im relaxed memory model funktioniert "Producer - Consumer - Pattern" nicht.

Initialisierung

$D = F = x = 0$

Thread 1	Thread 2:
d: $D = 42$	if: if $F = 0$ then goto if
f: $F = 1$	x: $x = D$

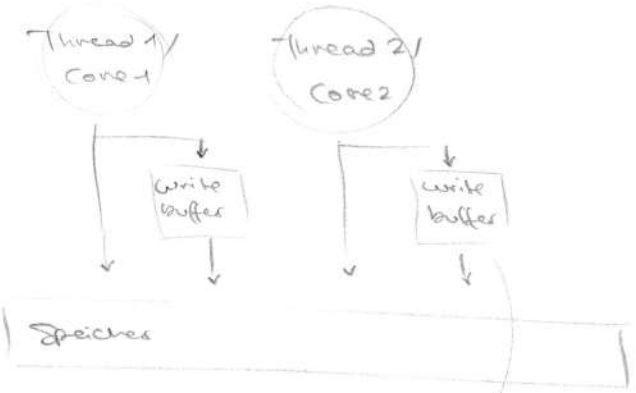
Reihenfolge:

f, if, x, d ⟶ (42, 1, 0)

nicht erfolgreich synchronisiert!

Beispiel für SC-Verletzung bei Multithreading wobei EO ≠ PO:

Architektur ohne Caches → Producer - Consumer Pattern



wenn Buffer voll, dann alles mit Speicher abgleichen und Buffer ausleeren

Initial:
Flag1 = 0
Flag2 = 0

Thread 1:
Flag1 = 1
if Flag2 = 0 then
... critical (no interruption)

Thread 2:
Flag2 = 1
if Flag1 = 0 then
... critical

Problem:
Lese-Operationen (wie die in IF Abfrage) überholen write Buffer!

Möglicher Ablauf:
Flag1 = 1 (im write-buffer)
if (Flag1 = 0) then ... → wird ausgeführt obwohl es nicht sein darf!
ERROR

Conclusion:
SC ist verletzt, weil durch Cache die Wertzuweisungen nicht mehr atomar sind.

Definition von Atomizität mit mehreren Caches und Cores:

SCHREIBEN:
Operation darf nicht unterbrochen werden
Alle Cores müssen gleichzeitig Zugriff auf den neuen geschriebenen Cache haben, wenn es ein Core hat.

LESEN:
So lange aufschieben bis alle Cores gleiche Cache-Kopien haben
ab letzter Schreiboperation

Weiteres Beispiel:
(Demonstration der Verletzung der Zeitlichen Relativität)

Beispiel:

T_1	T_2	T_3	T_4
a: $A = 1$	b: $B = 1$	c: $C = A$	e: $E = A$
		d: $D = B$	f: $F = B$

Initialisierung:
$A = B = C = D = E = F = 0$

Da alle Anweisungen in beliebiger Reihenfolge ausgeführt werden dürfen:

$(A, B, C, D, E, F) = (0,0,0,0,0,0)$
↓ d: $D = B$
$(0,0,0,\underline{0},0,0)$
↓ b: $B = 1$
$(0,\underline{1},0,0,0,0)$
↓ f: $F = B$
$(0,1,0,0,0,\underline{1})$
↓ e: $E = A$
$(0,1,0,0,\underline{0},1)$
↓ a: $A = 1$
$(\underline{1},1,0,0,0,1)$
↓ c: $C = A$
$(1,1,\underline{1},0,0,1)$ ——— (A B C D E F / 1 1 1 0 0 1)

Aus der Sicht von Thread 3:
$C = 1, D = 0$ ⟹ a vor b ausgeführt (also T_2 nach T_3)

Aus der Sicht von Thread 4:
$E = 0, F = 1$ ⟹ b vor a ausgeführt (also T_1 nach T_4)

Das würde bedeuten:

$T_2 < T_3$
$T_1 < T_4$ ⟶ nicht möglich mit diesem Output

Allgemein: Speichermodelle

- Hardware muss wissen: welche Art von Instruction Scheduling
- Programmiersprache sagt Hardware wo fences gelegt werden müssen, beschreibt Thread Interaktion mit Speicher
- Fences können nicht automatisiert gesetzt werden. (und benötigen zusätzliche Hardware)

→ Speichermodell - Operationen

Store : atomic_store (atomic_var, memory_order)
load : atomic_load (atomic_var, memory_order)
exchange : atomic_exchange (atomic_var1, atomic_var2, memory_order)

Auswahl unter Sprachen:

JAVA: SC
C++ : SC, RA, Relaxed ... } (schwaches Speichermodell)
C : SC, RA, Relaxed ... }

Auswahl bei memory_order:
SC, relaxed, release und/oder aqui-r

Auswahl unter Architekturen:

Intel x86/64: SC
ARM: schwaches Speichermodell
RISC-V: schwaches Speichermodell

Performance:

SC < RA < Relaxed

↑
Sinnvoll wenn keine negativen Konsequenzen durch Anwendung.

→ RELEASE / ACQUIRE MEMORY MODEL

Benötigt Memory fences / Memory Barriers (die nur load und store Operationen betreffen)

- Verbieten Einseitig das Umordnen über 2ten
- Ausgesprochen mit Release / Acquire Commands
  ↳ Nur Operation auf atomare Variablen (≠ gewöhnlich) möglich

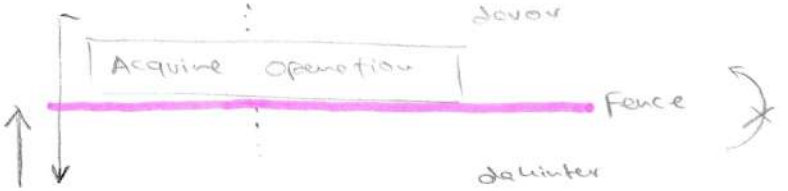
Relaxed

keine Einschränkung

Release (store, speichern)

Operationen dahinter dürfen Fence überschreiten

Acquire (load, lesen)



Operationen davor dürfen Fence überschreiten

Alles unter der Annahme, dass Befehl selbst NIE verschoben wird

Wichtig: Datenabhängigkeiten per default immer berücksichtigt:

⇒B
```
atomic (counter++)      ← schreiben auf counter
if (atomic(counter) > 0)  ← lesen auf counter
...
```

Darf relaxed werden

Release - Aquire Beispiele:

Producer - Consumer - Pattern

D = F = X = 0

Thread 1

Fence
d: D = 42
f: atomic_store (F, 1)

Thread 2

if: if atomic_load (F) = 0 then goto if
Fence
x: X = D

Reduzen sieht man:

Synchronisation und Kommunikation mit diesem Pattern funktioniert mittels RA - Pattern

Wann ist es sinnvoll aus RA relaxed zu machen?

Thread 1:

z: Z = 15
y1: Y = 11
a: atomic_store (A, 1)

Thread 2:

if: if atomic_load (A) ≠ 1 goto if
y2: Y = 2

→ Lösung:

hier gar nicht:

1. z darf nicht hinter a kommen
2. y2 darf nicht vor if kommen
3. y1 darf nicht hinter a kommen, sonst könnte nach y2 nochmals y1 y überschreiben

Thread 1:

y: Y = 42
a1: atomic_store (A1, 0)
a2: atomic_store (A2, 1)

Thread 2:

if: if atomic_load (A1) ≠ 0 OR atomic_load (A2) ≠ 1 goto if
x: X = Y

→ Lösung:

if Schleife wird verlassen, wenn (A1 = 0 ∧ A2 = 1)

a1 darf relaxed werden, weil es egal ist ob zuerst a1 oder a2 gespeichert wird, aber da UND a_2 dürfen nicht gleichzeitig relaxed werden

a1 vor a2 darf relaxed werden

relaxed werden darf

Wenn Compiler if - booleans von links nach rechts bewertet, beeinflusst das (neben der Tatsache dass wenn a1 oder a2 relaxed ist) welche if Abfrage

Das relaxen der ersten (linken) if-Abfrage, wäre ausreichend dafür, dass nichts nach vorne wandern kann.

Somit hätten wir am Ende 2 memory-Fences weniger!

Des relaxen der ersten (linken) if-Abfrage, wäre ausreichend dafür, dass nichts noch vorne wandern kann.

Somit hätten wir am Ende 2 Memory-Fences weniger!

Fortsetzung des Beispiels:

Angenommen

Thread 1

y : y = 42

a1 : atomic_store (A1, 0)

a2 : atomic_store (A2, 1)

Thread 2:

if atomic_load (A1) ≠ 0 OR
atomic_load (A2) ≠ 0 goto if

x = y

Case 1: relaxed a1

Mögliche Anordnungen:

Thread 1

y: y = 42

a1 : A1 = 0

a2 : atomic_store (A2, 1)

möglich

$\begin{pmatrix} y \\ a1 \\ a2 \end{pmatrix} \begin{pmatrix} a1 \\ y \\ a2 \end{pmatrix}$ ✓ gültig

Case 2: relaxed a2

Mögliche Anordnungen

Thread 1

y: y = 42

a1 : atomic_store (A1, 0)

a2 : A2 = 1

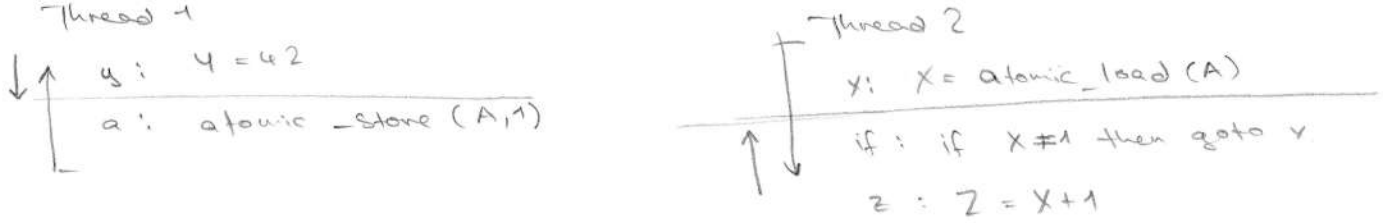
möglich

$\begin{pmatrix} y \\ a2 \\ a1 \end{pmatrix} \begin{pmatrix} a2 \\ y \\ a1 \end{pmatrix}$ ✓ gültig

Der einzig ungültige Fall den man vermeiden muss:

$\begin{pmatrix} a1 \\ a2 \\ y \end{pmatrix} \begin{pmatrix} a2 \\ a1 \\ y \end{pmatrix}$

Weiteres Beispiel:



→ Lösung:

- a kann relaxed werden, weil y irrelevent ist
- x kann relaxed werden, weil if und z über x Datenabhängig sind und by default nur danach stehen können

  (nur z darf mit if vertauscht werden)

- theoretisch muss A nicht atomic sein.

Blockierendes Verhalten

bisher in „Kommunikation & Synchronisation" : 1 Thread wartet in Endlos-Schleife auf Flag

Alternative : **Semaphore**

Eine Semaphore ist eine atomare Variable die wie ein Zähler funktioniert und am Anfang auf eine Zahl ≥ 0 gesetzt wird (max # des Threads die parallel laufen dürfen)

(Siehe Wikipedia: a Bücherei Beispiel mit Zimmern)

Semaphore = Zähler = 1 initialisiert

(wait()) **Lock()**

Zähler --

Wenn Zähler ≥ 0, darf aufrufender Thread weiterexekutieren,

wenn aber < 0 muss Thread in Queue warten (exekution stoppen)

(post()) **Unlock()**

Zähler ++

Wenn Zähler ≤ 0 zu 1 wird, Thread aus Queue holen, sonst nichts

Race Condition

Ausführungsreihenfolge beeinflusst Ergebnis wenn nicht alles atomar abläuft, memory fences deshalb notwendig!

Lösung:

Read-Modify-Write Operationen (in Hardware → Prozessorarchitektur abhängige Instruktion)

Versichert Atomizität, für Semaphoren geeignet

Vor und Nachteile von Semaphoren

- Korrektheit schwer prüfbar → für jedes Lock braucht es ein Unlock

↓

Deshalb leichter handhabbare Alternativen in höheren Sprachen:

- Monitore
- Synchronized Objects, Java
- Protected Object, Ada

+ leicht verständlich (siehe Beispiel)
+ performant
+ Sequentially Consistent wenn Zähler = 1

- Wenn ein Thread während kritischer Operation zwischen lock() und unlock abstürzt, können andere keinen Fortschritt machen
- Thread dispatching!
  (Stoppen und Starten von Threads) sehr langsam

Beispiel für Semaphoren:

globale Variable Z soll nur gleichzeitig von einem Thread benutzt werden.

Global:

```
Z = ...
S = 1
```

Thread:

```
lock(S)
Z_local = Z
Z_local = Z_local + ...;
Z = Z_local
Unlock(S)
```

Nicht Blockierendes Warten

Statt Semaphoren direkt Read-Modify-Write-Operation von Hardware benutzen

„Optimistisches Vorgehen":

zB
1. Probieren Änderung auf globale Variable Z durchzuführen
2. Falls nicht unterbrochen Ende
   Sonst bei 1. fortsetzen

Funktion: RMV (V, alter_Wert, neuer_Wert)

returned true: Atomare Variable V hatte immer noch alten_Wert und Überschreibung erfolgreich

returned false: Sonst

Beispiel:

Z ist atomar

Thread:

1. Z_local = Z
2. if not RMV (Z, Z_local, Z_local + ...) then goto 1

(alter Wert, neuer Wert; false, true)

Auswirkung:
Endlosschleife, so lange bis Z = Z_local.
Dann bekommt Z = Z_local + ...
und Programm endet in diesem Thread

Vor und Nachteile

+ Effizient bei wenigen Threads, da sie nicht gestoppt / gewartet werden müssen. (Über 100x schneller →)
+ Wenn ein Thread abstürzt können andere Fortschritte machen
+ Sequentially consistent und Release and Acquire Speichermodell möglich
- Schwer verständlich → vor allem mit RA-Speichermodell