

Zweierkomplement

gge. Dez Zahl in ZKPL:

(1) positive Zahl als Dualzahl schreiben, ggf. auf gewünschte Anzahl Bit auffüllen

(2) falls geg. Zahl negativ:

nach Umrechnung in Dualzahl Einerkomplement bilden: invertieren
 $0 \rightarrow 1 / 1 \rightarrow 0$

(3) +1 addieren zur erhaltenen Zahl aus (2)

gge. ZKPL-Zahl in Dez.Zahl:

(1) führende 0: einfach als Dualzahl umrechnen \rightarrow positive Zahl

(2) führende 1: -1 abziehen, komplementieren, übersetzen, ! Vorzeichen mitnehmen

Zahlensysteme

EUKLIDISCHER ALGORITHMUS.

Idee: Zahl im 10er System in Potenzen im Zielsystem zerlegen

Bsp.: 27042013₍₁₀₎ in Hexadez.system

Vorbereitung: höchste Potenz finden:

$$16^7 > 27042013 > 16^6$$

$$27042013 = (1)_{16} \cdot 16^6 + 10264797 \leftarrow \text{Rest}$$

$$10264797 = (9)_{16} \cdot 16^5 + 827613$$

$$827613 = (C)_{16} \cdot 16^4 + 44181$$

$$44181 = (A)_{16} \cdot 16^3 + 221$$

$$221 = (D)_{16} \cdot 16^2 + 21$$

$$21 = (1)_{16} \cdot 16^1 + 13$$

$$13 = (D)_{16} \cdot 16^0$$

$$\sim 27042013_{(10)} = 19CA0DD_{(16)}$$

Dez	Hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

↑ niedr. weit.
↑ hochst. weit.

Trick: $16^6 = (2^4)^6 = 2^{24}$

bei Nachkommastellen analog weiterrechnen

HORNER SCHEMA

Vorkomponente der Zahl: immer wieder durch neue Basis teilen, Rest ist Stelle

ZB. $27042013 : 16 = 1690125$ Rest 13 (13_{16}) niedr. weit.

$$1690125 : 16 = 105632 R 13 (13_{16})$$

$$105632 : 16 = 6602 R 0 (0_{16})$$

:

$$1 : 16 = 0 R 1 (1)_{16}$$

Abbruchbed.

hochstwert.

↗!

Nachkommateil: mit Zielbasis multiplizieren

Bsp.: $0,22_{(10)}$ in 5er System:

$$\begin{aligned} 0,22 \cdot 5 &= 1,1 \rightarrow 1 \\ 0,1 \cdot 5 &= 0,5 \rightarrow 0 \\ 0,5 \cdot 5 &= 2,5 \rightarrow 2 \\ 0,5 \cdot 5 &= 2,5 \rightarrow 2 \end{aligned}$$

$\Rightarrow 0,22_{(10)} = 0,10\bar{2}_{(5)}$

! periodisch

↓ Verarbeitung

IEEE-754

s1	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	b16	b17	b18	b19
v2	Charakteristik	Mantisse																	
b32	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16	b15	b14	b13

single precision

double precision

$$\text{Dec} = (-1)^{\text{V2}} \cdot (1, \text{Mantisse}) \cdot 2^{\text{Exp}}$$

$$\text{Exp} = \text{char} - 127/1023$$

erstes Bit der Mantisse implizit 1

NAN: char = 11...11, Mantisse ≠ 0...0

±∞: V2=0/1 char = 11...11, Mantisse = 0...0

Q: V2 = 0 char = 0...0, Mantisse = 0...0

insg. 2^4 Zahlen darstellbar, 10-15 aber nicht benötigt

→ 1010 - 1111 Pseudofakturen, kodieren hier nichts

Fehler finden: Prüfbits erneut bilden, aber Prüfbit miteinrechnen

also:

$$u_1 = u_1 \oplus m_1 \oplus m_2 \oplus m_3 \oplus \dots$$

$$u_2 = u_2 \oplus m_1 \oplus m_2 \oplus \dots$$

Wenn fehlerfrei sind alle $u_i = 0$

ansonsten: ... $u_1 u_2 u_3 u_4 u_5 u_6 u_7 u_8$ als Dualzahl interpretieren → an dieser Position Fehler

Schaltalgebra

Ass. ges.: $(a \wedge b) \wedge c = a \wedge (b \wedge c)$ gleiches für \vee

Idemp.: $a \wedge a = a$ $a \vee a = a$

Abs g.: $a \wedge (a \vee b) = a$

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

DeMorg.: $\overline{a \wedge b} = \overline{a} \vee \overline{b}$

$\overline{a \vee b} = \overline{a} \wedge \overline{b}$

$\rightarrow a \Leftrightarrow b = (a \wedge b) \vee (\overline{a} \wedge \overline{b})$; $a \oplus b = (a \wedge \overline{b}) \vee (\overline{a} \wedge b)$

$\rightarrow \pi$ NAND, τ NOR nicht assoziativ; $\bar{a} = a \bar{\wedge} a$, $\text{NAND}_3(a, b, a) \neq \bar{a} \bar{\wedge} b \bar{\wedge} a$

KONVERSIONEN:

\rightarrow disjunktive Form \rightarrow NAND: doppelte Negation \leftarrow De Morgan

\rightarrow konjunktive Form \rightarrow NOR:

disjunktive Form \rightarrow NAND } konjunktive Form \rightarrow NOR } doppelt negieren + DeMorgan

disjunktive Form \rightarrow NOR } Funktion negieren
konjunktive Form \rightarrow NAND } \rightarrow Konj. / Disj. doppelt negieren
 \rightarrow De Morgan \rightarrow Ergebnis negieren

$f = \overline{f} \wedge \overline{\overline{f}}$

Shannonscher Entwicklungssatz

- Entwicklung nach x_i : Variable in Funktion auf 1 setzen, entsprechenden Term mit x_i verbinden (\wedge), verbinden mit: Variable auf 0 setzen, Term mit $\overline{x_i}$ verbinden

$$y = f(x_1, \dots, x_n) = [x_i \wedge f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n)]$$

$$\vee [\overline{x_i} \wedge f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n)]$$

Würfelkalkül

Funktion im Würfelkalkül darstellen: z.B.: Minterme darstellen

$C = \{ \text{Menge aller Minterme als Würfel} \}$

Minterm als Würfel: 0 falls \bar{x} , 1 falls x , - falls x nicht vorkommt

z.B. $DKF = \bar{a}\bar{b}c \vee \bar{b}c \vee ac \rightarrow C = \{(1,0,1); (-,0,1); (1,-,1)\}$

Normalformen

DNF: Minterme, alle Belegungen die ausgewertet 1 sind
(der in Fkt. Tabelle)

- der aus Fkt. Tabelle ablesen, entspr. Eingangsvariablen mit \wedge verknüpfen, alle Minterme mit \vee verknüpfen

KNF: Maxterme, Funktion wird 0

- Der aus Fkt. Tabelle ablesen, Eingangsvariablen negiert mit \vee verknüpfen, Maxterme mit \wedge verknüpfen

Bsp:

b	a	f(b,a)	Minterme	Maxterme
0	0	1	$\bar{b}\bar{a}$	$b\bar{a} \vee b\bar{a}$
0	1	0		$\bar{b}a \vee \bar{b}a$
1	0	0		$\bar{b}a \wedge \bar{b}a$
1	1	1	$b\bar{a}$	

0/21

$$\text{DNF: } f(b,\bar{a}) = \bar{b}\bar{a} \vee b\bar{a} = \text{MIN}(0,3)$$

$$\text{KNF: } f(b,a) = (\bar{b} \vee a) (\bar{b} \vee \bar{a}) = \text{MAX}(1,2)$$

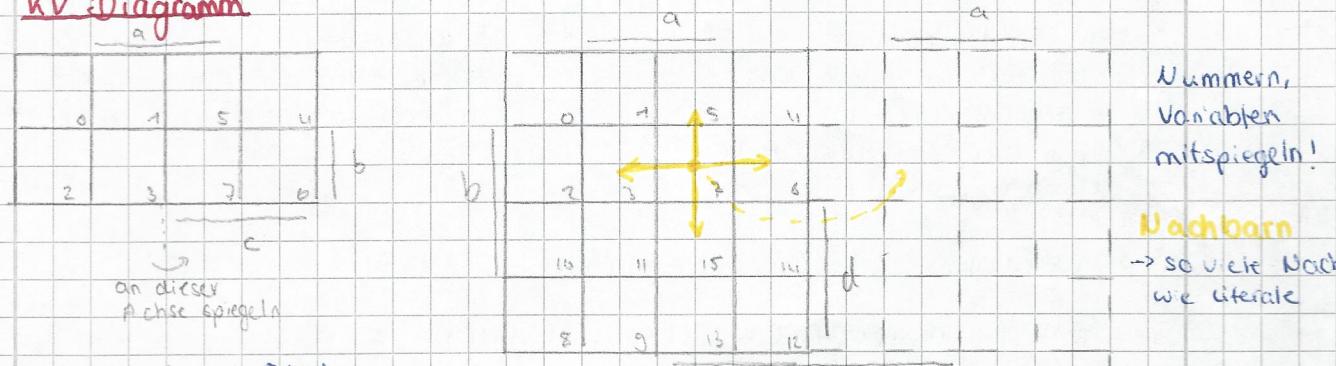
Vereinfachen:

- Zusammenfassen mit Gesetzen der Schaltalgebra $\rightarrow a \vee a = a / a \wedge a = a$ verwenden!

Minimalformen

Normalformen so weit vereinfachen wie möglich

KV-Diagramm



Primimplikat: 0-Blöcke (= Maxt.)

Primimplikant: 1-Blöcke (= Mint.)

→ immer Größe einer Zel Pot. \rightarrow müssen maximal sein!

DNF: Primimplikante suchen (muss max. sein \rightarrow don't cares $\sim 0,1$), mit oder verknüpfen

KMF: Primimplikate suchen (0-Blöcke: liegt 0 Block bei $a \wedge b \rightarrow \bar{a} \vee b$), mit und verknüpfen

Kernprimimpl.: notwendige Primimpl., da sie als einzige eine 1-Stelle/0-Stelle abdecken
entbehrliche Pr.: Pr., die durch Kernpr. schon abgedeckt sind
Wahlpr.: Stellen werden von mehreren abgedeckt, kann aussuchen welchen ich nehme

Quine-McClusky-Verfahren

Ziel: DMF/KMF aus geg. MINT/MAXT der Funktion

MINT/MAXT nach Anzahl 1en in binärer Darstellung sortieren
 ↳ 1. Quine'sche Tabelle o. Ordnung

Nr.	1. Ordnung	Nr. (MINT)	o. Ordnung
	0 1 1 0 +	2	0 0 0 1 0
	0 1 1 - 0	4	0 0 1 0 0
	0 1 1 - 1	5	0 0 1 0 1
	0 1 1 - 1	10	0 1 0 1 0
		:	:

z.B.: Block 1 mit Block 2 vergleichen

2 mit 3

→ unterscheidet sich nur eine Stelle → "don't care"

→ diese zusammenfassen, entsprechend zusam. gefasste Zeilen streichen

→ so oft wiederholen bis nichts mehr zusammenfassbar

→ diese übrigen Ausdrücke sind Primimplikanten(n) → benennen mit A, B, C, ...
 (Test: alle Pr. im. müssen alle MINT/MAXT enthalten (- durch "0"/"1" ersetzen))

Zwischenstand: Alle Primimplikanten(n) bestimmt

→ Überdeckungstabelle aufstellen

	A	A ∨ B	A ∨ C	
	12	13	15	19 30 35 ... wie MINT/MAXT
A	x	x	x	x
B		x	x	
C		x	x	
:				

→ Kreuzchen setzen: welche MINT/MAXT deckt A/B/C/... ab?

→ Vereinfachungsregeln:

(a) Spaltendominanz:

überdeckt eine Spalte eine andere mit ihren Kreuzen, so kann die Spalte mit mehr Kreuzen gestrichen werden

(b) Zeitendominanz:

streiche Zeilen, die durch andere überdeckt werden

→ Gleichungen aufstellen, vereinfachen

$$\text{zB: } f = A \times B \times C \cdot (A) \wedge (A \vee B) \wedge (A \vee C) \wedge (B) \wedge (C) = A \wedge B \wedge C$$

→ Rückübersetzen z.B. $A = (\overset{\text{e} \text{dc}}{0} \overset{\text{ba}}{1} \cdots) = \overset{\text{e} \text{dc}}{0}$

→ je nach gewünschter Form verknüpfen

Consensus-Verfahren ! liefert nur Pimimplikanten, keine DNF!

Vorteil: einfacher im Rechner zu implementieren

(1) Funktion als Würfelmenge darstellen \rightarrow "Don't care" als 1

Würfelmenge: $C = \{(0,1,0,-), (-1,1,0)\}$
 $\begin{array}{c} \diagup \\ d \\ \diagdown \end{array} \begin{array}{c} \diagup \\ b \\ \diagdown \end{array} \quad \begin{array}{c} \diagup \\ c \\ \diagdown \end{array} \bar{a} \quad \rightarrow \text{kann nur DF darstellen!}$

(2) Würfel in Tabelle übertragen:

z.B.	Nr.	gebildet aus	würfli	gestrichen wegen
	1		-, 0, 0, -	
	2		-, 0, 1, 1	c 4 (Teilmenge von 1)
	3		0, 1, 0, 0	c 5
	4	2, 1	-, 0, -, 1	
	5	3, 1	0, -, 0, 0	
		5, 4	0, 0, 0, -	c 1

- \rightarrow bei Nr. 2 anfangend: vgl. Würfel mit allen ungestrichenen drüber (von unten nach oben) und bilde wenn mögl. consensus-Würfel
- \rightarrow unten anfügen, fortlaufend nummerieren
- ! immer schaute ab Würfel nicht schon irgendwo enthalten
- \rightarrow aufhören wenn kein neuer Würfel mehr gebildet werden kann

Consensus-Würfel: zB $\begin{array}{c} -, 0, 0, 1 \\ 0, -, 1, 1 \end{array} \quad \begin{array}{l} \text{? suche ein } 0\text{-1-Pair} \\ \text{ } \end{array}$

\Downarrow

$\begin{array}{c} 0, 0, -, 1 \\ -, 0, -, 1 \\ -, -, - \end{array} \quad \begin{array}{l} 0, 0 / 1, 1 \rightarrow \text{schreibe } 0/1 \text{ an diese Stelle} \\ -, 0/1, - \rightarrow -" \quad 0/1 \quad -" \\ -, -, - \rightarrow -" \quad - \text{ an diese Stelle} \end{array}$

Schaltwerke: Ausgabe auch abh. von vergangenen Belegungen der Eingangsvariablen
↳ ZUSTÄNDE

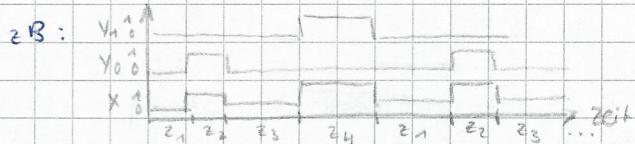
Moore-Automat: Ausgabe hängt nur vom Zustand ab
(synchrone, asynchrone Schaltwerke)

Meady-Automat: Ausgabe hängt von Zustand, Eingabe ab

1. Ordnung: Ausgabe abh. von altem Zustand (synchrone Schaltwerke)
2. Ordnung: Ausgabe abh. von neuem Zustand (asynchr. Schaltwerke)

DARSTELLUNGSMÖGLICHKEITEN

Zeitdiagramm: Veranschaulichung des Problems, beispielhafte Folge von Eingabebereignungen



Ablauftabelle: Folgezustand und Ausgabe wird für jede mögliche Kombination aus Zustand und Eingabe angegeben

		Eingabe	neuer Zustand		Ausgabe
			z_1	z_2	y
zustand	e	$z_1 \rightarrow z_2$	z_1	z_2	$y = 0$
		$z_1 \rightarrow z_1$	z_1	z_1	$y = 1$
z_2	0	$z_2 \rightarrow z_1$	z_2	z_1	$y = 1$
	1	$z_2 \rightarrow z_2$	z_2	z_2	$y = 0$
:					
alle Kombinationen angeben!					

Automatentabelle:

senkrecht: Zustände

waagrecht: Eingänge

Matrix: Folgezustände

z.B.: Moore-Automat

z_k	z_{k+1}	$y_0 y_1$
$x=0$	$x=1$	
z_1	z_1	00
z_2	z_3	10
z_3	z_3	00
z_4	z_1	01 → Ausgabe 01

mit altem Zustand z_4 und Eingabe 0 ist z_1 der neue Zustand

Meady-Automat

z_k	$z_{k+1} / y_0 y_1$
$x=0$	$x=1$
z_1	$z_1/00$
z_2	$z_3/00$
z_3	$z_3/00$
z_4	$z_1/00$

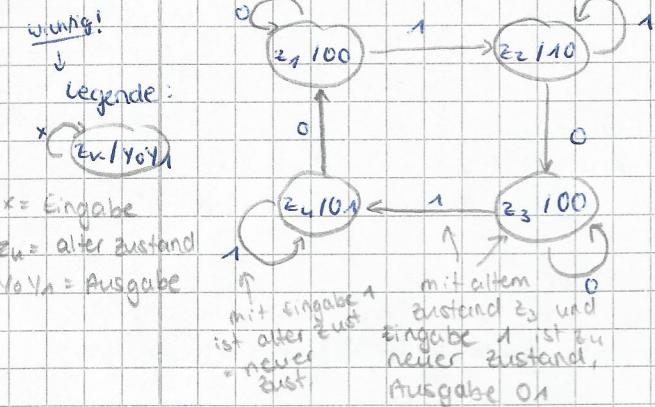
mit altem Zustand z_4 und Eingabe 1 ist z_1 der neue Zustand und Ausgabe 01

* synchrones Schaltwerk: alle Zustandsspeicher (Speicherung vergangener Informationen, zB durch Rückkopplung) werden von einem Takt gesteuert / mehreren Takten
→ synchrones Taktsignal

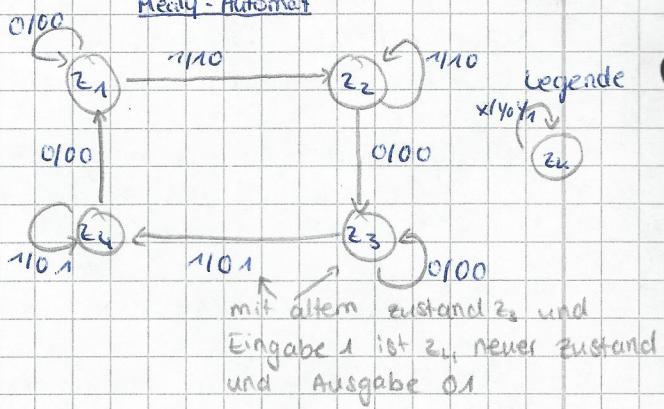
asynchrones Schaltwerk: nicht synchron, kein allgemeiner Takt

Automatengraph: $AG = (Z, \kappa)$ Z = Zustände, κ = Menge der Übergänge κ zw. Zuständen

z.B. Moore-Automat



Mealy-Automat

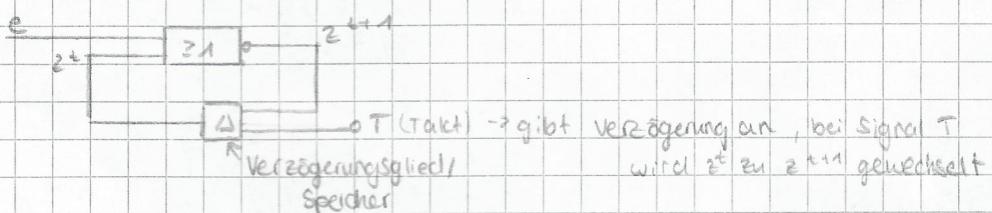


* Regelsteuerung: schaltet nur durch wenn über bestimmter Spannung

Flankensteuerung: während pos. ($0 \rightarrow 1$) oder neg. ($1 \rightarrow 0$) Taktflanken werden Werte übernommen; Vorteil: Eingänge müssen nur kurz gültig sein (steift über ganze Takthälfte)

Takt: abhängig von Totzeit, Verzögerungszeit
 längste Verzögerung sollte etw. kleiner als Takt sein

Rückkopplung:



Entwurf asynchroner Schaltwerke

- (1) Aufstellen des Automatengraphen
- (2) Aufstellen der Automatentabelle
- (3) Wahl der Zustandscodierung (z.B. $a=00, b=01, c=10$)
- (4) Erzeugen der Ausgabe- und Übergangsschaltung
- (5) Analyse asynchroner Schaltwerke (Übergänge)

Flussmatrix Automatentabelle für asynchrone Schaltwerke

→ stabile Zustände eingekreist; stabiler Zustand: aktueller Zustand = Folgezustand

zB..	z	z^+	
	$e_1 \ e_2 \ e_3 \ e_4$		Übergänge
0	⑦ 3 2 1		direkter Übergang: stabiler Zustand geht direkt in stabilen Folgezustand über
1	① ② ④ 0		
2	② 3 3 3		
3	0 ③ ③ 0		

indirekter Übergang: stabiler Folgezustand stellt sich über mehrere instabile Zwischenzustände ein

Oszillation: es stellt sich kein stabiler Folgezustand ein

Wettlauf

Zustandscodierung: Zuständen 1, 2, ... Binärzahlen zuordnen
→ bei Übergängen ändern sich ggf. mehrere Zustandsvariablen

zB:	$\rightarrow z$	u_2	u_1	u_2, u_1 codieren z
	zustand	0	0 0	zustandsvariablen
		1	0 1	
		2	1 0	{ muss nicht unbedingt bin. Darstellung der Zahl sein,
		3	1 1	} könnte auch $3 = (0,1)$ oder $1 = (1,1)$ sein

→ zB Übergang $(u_2, u_1) = (0,0) \rightarrow (1,1)$, also $0 \rightarrow 3$

→ Problem: Zustandsvariablen ändern sich nicht genau gleichzeitig
→ kurzfristig Zwischenwerte

zB. $(0,0) \rightarrow (0,1) \rightarrow (1,1)$
oder $(0,0) \rightarrow (1,0) \rightarrow (1,1)$

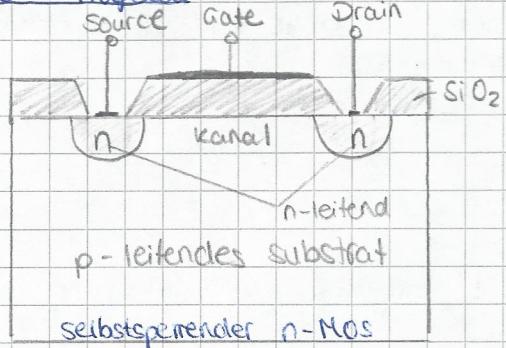
→ eine Zustandsvariable ist schneller → WETTLAUF

KRITISCHER WETTLAUF:
→ Übergang enthält Zwischenwerte, die stabil sind
→ richtiger Endzustand wird nie erreicht → Schaltweise funktioniert nicht richtig
→ Übergang enthält Zwischenzustand, der anderen Folgezustand liefert als Endzustand sein sollte

→ Beheben durch geeignete Wahl der Zustandscodierung
→ bei jedem Übergang ändert sich max. eine Zustandsvariable
→ erfordert mehr Zustandspeicher nötig als mit optimaler Codierung

MOSFET

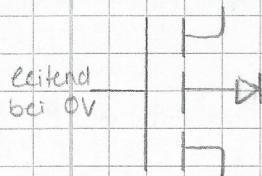
n-Mos Aufbau:



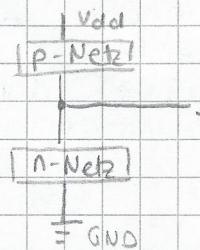
- leitet bei pos. Spannung

- selbstsperrend wenn keine Spannung anliegt

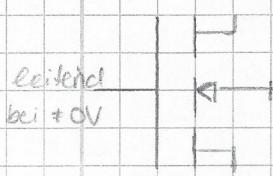
p-Mos: leitet wenn OV anliegen



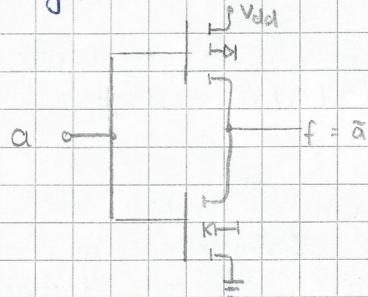
CMOS: pMOS + nMOS



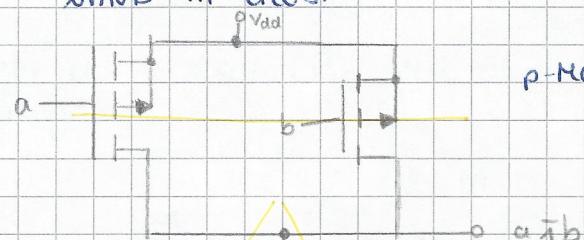
n-Mos: leitet wenn \neq OV anliegen



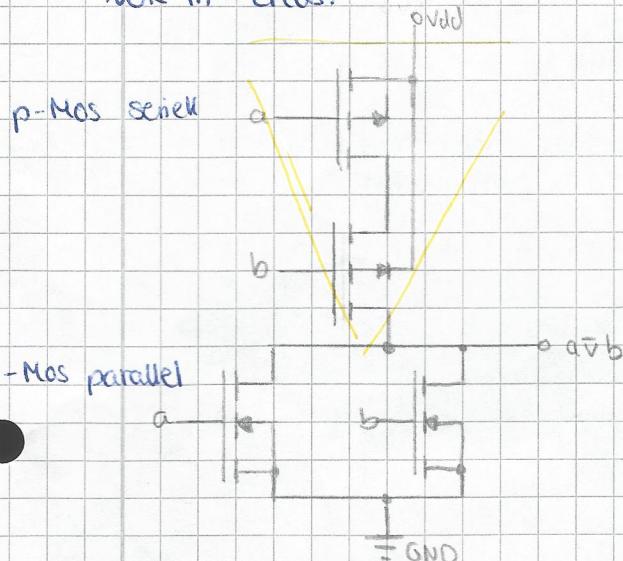
Negation in CMOS:



NAND in CMOS:

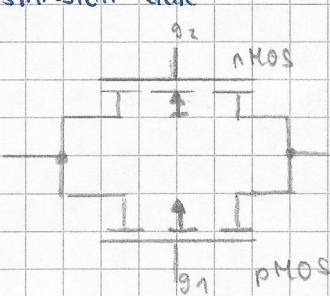


NOR in CMOS:



n-Mos schnell

Transmission-Gate:

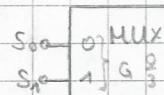


Zustand	leitend
g_2 niedrige Spannung	Sperrend
g_1 hohe Spannung	leitend

Zustand	leitend
g_2 hohe Spannung	Sperrend
g_1 niedrige Spannung	leitend

Zustand 1

Multiplexer



s_1, s_0 "bildet" Dualzahl je nach anliegender Spannung
 ~> entsprechende Dez. Zahl wird durchgeschaltet
 z.B. $s_1, s_0 = 11_{(2)} = 3_{(10)}$ $a = e_3$

$e_0 = 0$	0
$e_1 = 0$	1
$e_2 = 0$	2
$e_3 = 0$	3

4:1 Mux $\rightarrow 2^2$ Eingänge, 2 Steuereingänge (s_1, s_0)

Allg.: n Steuereingänge $\rightarrow 2^n$ Eingänge $\rightarrow 1$ Ausgang

Implementierungstabelle

Aus Eingängen Steuereingänge wählen z.B. cb, einen Eingang auslassen z.B. a

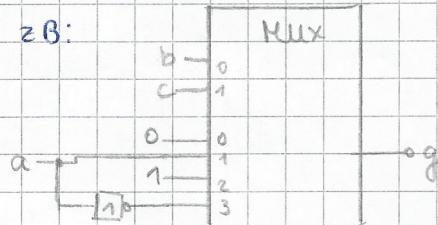
- Alle mögl. Kombinationen als erste Zeile
- nächste zwei Zeilen: übrigen Eingang einmal mit 0, einmal mit 1 eintragen
- Tabelle durchgehen und auf Kreuzungspunkten entspr. Wert eintragen

z.B.:

cb	00	01	10	11
$a=0$	0	0	1	1
$a=1$	0	1	1	0
g	0	a	1	\bar{a}

Daraus Schaltnetz: cb als Dualzahl lesen, entsprechender Eingang mit 0/1/a/ \bar{a} verknüpfen

z.B.:



Strukturausdruck: Struktur der Funktion aufschreiben, nicht vereinfachen! ggf. mit Pfadvariablen

Schaltfunktion: wann wird f für Funktion 1, nicht vereinfachen

PLA - Baustein

PLA = programmable logic array

2 Matrizen: UND: Aufbau von Implikanten aus Eingangsvariablen
 ODER: Aufbau der Funktion aus Implikanten

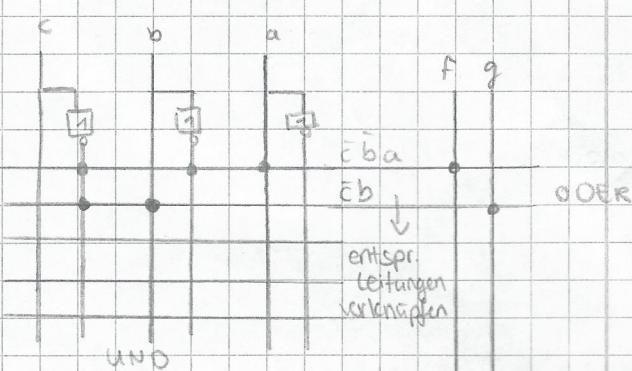
Eingänge

$\downarrow \downarrow \downarrow$

UND

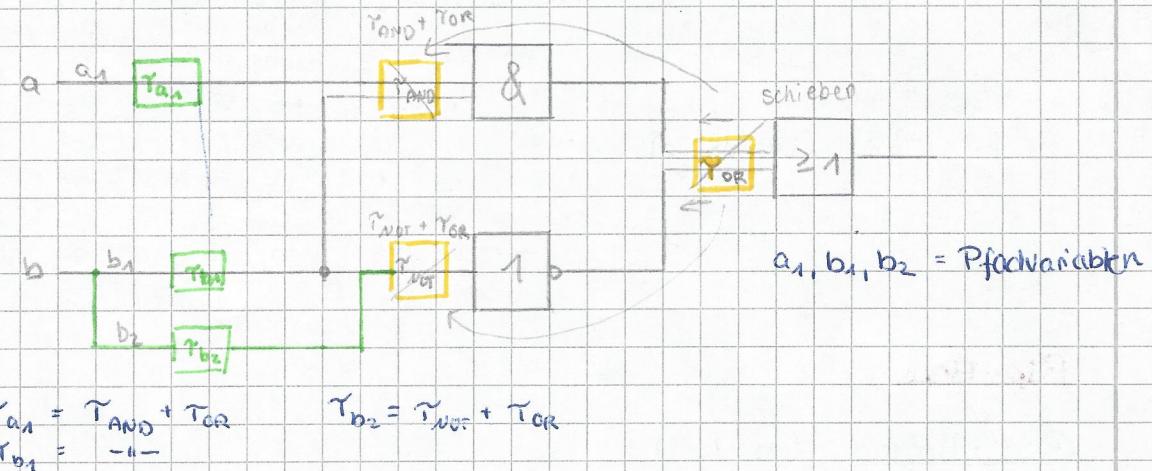
ODER

$\downarrow \downarrow \downarrow$ Ausgänge



Totzeitmodell

Totzeiten für jedes Schalteil bis vorne durchschreiben, ggf. aufteilen in Pfadvariablen → Trennung in reinen Verzögerungs- und reinen Verknüpfungsteil



Übergänge

Problem: Übergänge nicht exakt, Variablen springen nicht unbedingt gleichzeitig
 zB: $0 \rightarrow 1$ sieht so aus: $(0,0,0) \rightarrow (0,1,0) \rightarrow (1,1,0) \rightarrow (1,1,1)$
 ~ zwischenständige können problematisch werden

Unterscheidung in:

statischer Übergang:

- > stat. 0-Übergang: Funktionswert vor und nach Übergang 0
- > stat. 1-Übergang: Funktionswert vor und nach Übergang 1

dynamischer Übergang:

- > dyn. 0-1-Übergang: Fkt. Wert wechselt von 0 auf 1
- > dyn. 1-0-Übergang: Fkt. Wert wechselt von 1 auf 0

Hasardfehler: Mehrmalige Änderung der Ausgangsvariable während ~~Übergang~~ Übergang

Hasard: logisch strukturelle logisch-strukturelle Voraussetzung für Hasardfehler, ohne Berücksichtigung der konkreten Verzögerungszeiten

$$\begin{array}{ll} \text{Hasardfehler} & \Rightarrow \text{Hasard} \\ \text{Hasard} & \nRightarrow \text{Hasardfehler} \end{array}$$

Funktionshasard:

- > Ursache in zu realisierender Funktion
- > tritt immer auf und tritt in allen Schaltnetzen auf
- > nicht behobbar
- > kann verhindert werden (Anpassung d. Totzeiten)

Strukturhasard:

- > Ursache in Struktur des Schaltnetzes
- > kann behoben werden (Änderung der Struktur)

Funkt.h.: normales KV-D.

Strukt.h.: KV-D mit Pfadvariablen

Erkennen von Hasard: KV-Diagramm: betrachte alle Wege, die Übergang nehmen kann (in Variablen wechseln Wert \rightarrow m! Wege)
 → Hasardbehaftet wenn mind. 1 Weg nicht monoton ($0 \rightarrow 1 \rightarrow 0 / 1 \rightarrow 0 \rightarrow 1$)

Satz von Eichelberger:

Ein Schaltnetz, das Disjunktion aller Primimplikanten / Konjunktion aller Primimplikante einer Funktion realisiert, ist frei von allen Strukturhasards, allen dyn. Strukt.h. bei denen nur eine Variable wechselt

Demultiplexer

DY	n Steuerleitungen $\rightarrow 2^n$ Ausgänge
s ₀ 0 0 0 0 0 0	e=0 \rightarrow alle Ausgänge auf 0
s ₁ 0 1 1 0 0 1	e=1 \rightarrow n Eingänge s. bilden Dualzahl \rightarrow an diesen Ausgang wird e durchgeschalten
	z=0 a ₂
	z=0 a ₃

\rightarrow für diesen Demux also:

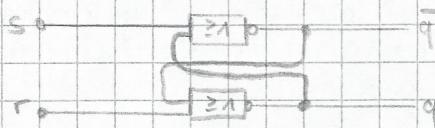
s ₁	s ₀	a ₀	a ₁	a ₂	a ₃
0	0	e	0	0	0
0	1	0	e	0	0
1	0	0	0	e	0
1	1	0	0	0	e

Flip-Flops

RS-Flipflop (reset/set)

\rightarrow Asynchroner RS-Flipflop

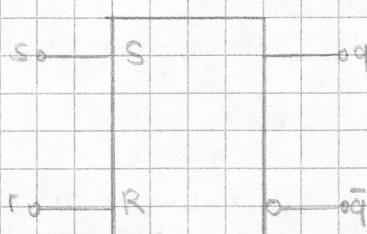
Schaltwerk:



Ansteuertabelle:

q ^t	q ^{t+1}	r ^t	s ^t
0	0	-	0
0	1	0	1 <small>"bin in Zustand 0"</small>
1	0	1	0 <small>"will in Zustand 1"</small>
1	1	0	- <small>"was muss ich bei r,s anlegen"</small>

Schaltsymbol:



Funktionstabelle:

(Ansteuertabelle
andersrum)

r	s	q ^{t+1}	Funktion
0	0	q ^t	Speichern
0	1	1	Set
1	0	0	Reset
1	1	-	Verboten!

\rightarrow Pegelgesteuerter RS-Flipflop Verwendung in synchronen Schaltwerken

\rightarrow Takt T:

\sim Eingänge mit Takt verknüpfen

Schaltwerk:

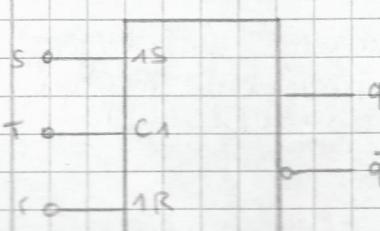


Ansteuertabelle:

(gleich wie
asynchroner)

q ^t	q ^{t+1}	r ^t	s ^t
0	0	-	0 <small>don't care</small>
0	1	0	1
1	0	1	0
1	1	0	-

Schaltsymbol:



1 hinter Buchstabe:

Eingang verursacht
Taktabhängigkeit

1 nach Buchstabe:

Eingabe abhängig
von Takt

Funktionstabelle:

siehe asynchroner RS

D-Flipflop (\leftarrow D-Setzen)

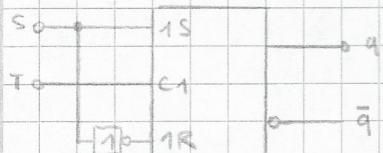
Idee: verhindere verbotene (1,1)-Belegung

CP eine Eingangsvariable d

CP d als neuer Zustand \rightarrow d wird ein Takt gespeichert

CP Eingangssignal um eine Taktperiode verzögert am Ausgang ("delay")

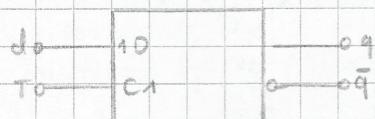
Schaltwerk: D-Latch



Ansteuertabelle:

q^t	q^{t+1}	d
0	0	0
0	1	1
1	0	0
1	1	1

Schalsymbol: D-Latch



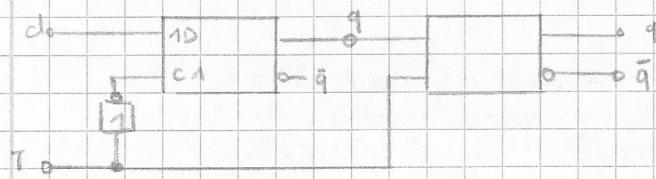
Funktionstabelle:

d	q^t	q^{t+1}
0	0	0
0	1	0
1	0	1
1	1	1

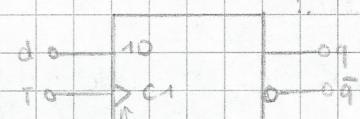
\rightarrow bin in Zustand 1,
und Eingabe 1 \rightarrow neuer
Zustand = 1

\Rightarrow flankengesteuertes D-Flipflop: zwei D-latches mit komplementären
einfachstes flankengesteuertes Speicher-
element Takteggeln

Schaltwerk:



Schalsymbol:

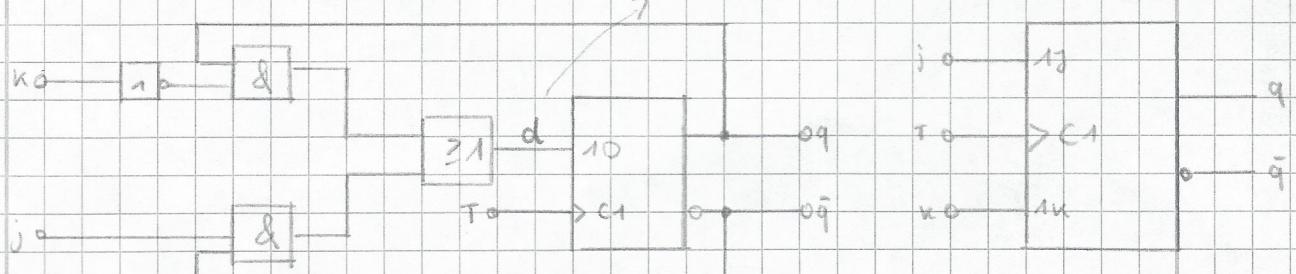


bedeutet: Taktflankensteuerung
(Negationszeichen davor falls neg.
Taktflankensteuerung)

JK-Flipflop (Jump / Kill)

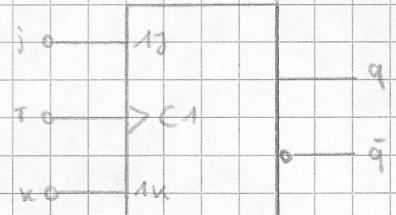
Idee: verbotene (1,1)-Belegung verwenden \rightarrow Wechsel

Schaltwerk:



$$d = q^t \bar{k} \vee \bar{q}^t j$$

Schalsymbol:



Ansteuertabelle:

q^t	q^{t+1}	j k
0	0	0 -
0	1	1 -
1	0	- 1
1	1	- 0

Funktionstabelle:

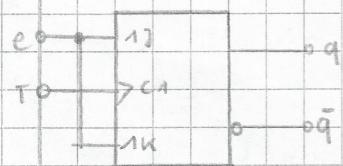
j	k	q^{t+1}
0	0	q^t
0	1	0
1	0	1
1	1	\bar{q}^t

Funktion:
speichern
rücksetzen \rightarrow kill: auf 0 springen
setzen \rightarrow jump: auf 1 springen
wechseln
(0 \rightarrow 1 / 1 \rightarrow 0)

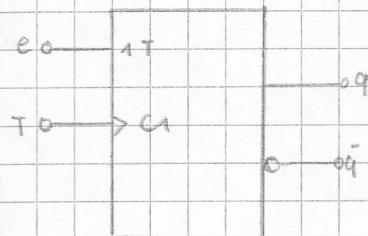
T-Flipflop (toggle)

nur ein Eingang, liegt 1 an: Zustandswechsel; liegt 0 an: alter Zustand bleibt erhalten

Schaltwerk:



Schaltsymbol:



Ansteuertabelle:

q^t	q^{t+1}	e
0	0	0
0	1	1
1	0	1
1	1	0

$e=0 \rightarrow$ Zustand speichern
 $e=1 \rightarrow$ Zustandswechsel

Funktionstabelle:

e	q^{t+1}	Funktion
0	q^t	Speichern
1	\bar{q}^t	Wechseln

Addierer:

Carry-Ripple: bei Addition einer Stelle muss auf jedem Übertrag aus vorhergehenden Stellen gewarnt werden
→ Add. Zeit proportional zur Anzahl Stellen

Carry-Lookahead: Überträge werden direkt aus Eingangsvariablen berechnet

Mikroprogrammierung

Alternative zu festverdrahtetem Steuerwerk

→ wird durch

→ Maschinenbefehl wird durch Mikroprogramm implementiert

Mikroprogramm besteht aus Mikrobefehlen

Mikrobefehle bestehen aus Bitfolge unmittelbarer Steuersignale für versch. Rechnerkomponenten

MIMA = mikroprogrammierte Minimalmaschine (von Neumann - Prinzip)

Lesephase → Dekodierphase → Ausführungsphase

3 Taktzyklen für Lese- / Schreibzugriffe

Befehlsformat, ALU-Operationen:

OpCode	Adr. / Konstante	0	OpCode				
			23	20	16	0	0
0	LD C	c → Akku	F				
1	LDV a	a → Akku					
2	STV a	Akku → a					
3	ADD a	Akku + a → Akku					
4	AND a	Akku AND a → Akku					
5	OR a	Akku OR a → Akku					
6	XOR a	Akku XOR a → Akku					
7	EQL a	Falls Akku = a : -1 → Akku ; sonst: 0 → Akku					
8	JMP a	a → IAR					
9	JMN a	Falls Akku < 0: a → IAR					
F0	HALT	Stoppt MIMA					
F1	NOT	EKP von Akku → Akku					
F2	RAR	rotiere Akku eins nach rechts → Akku					

Lesephase: immer gleich:

- 1. Takt: IAR → SAR ; IAR → X ; R=1
- 2. Takt: Eins → Y ; R=1
- 3. Takt: ALU auf Addieren ; R=1
- 4. Takt: Z → IAR
- 5. Takt: SDR → IAR
- 6. Takt: Dekodierphase
- 7. Takt: eigentliches Programm
- :

Mikrobefehlsformat:

Pro Takt eine Bitfolge
→ 1xx an entspr. Stellen
verwendeter "Variablen"
0xx sonst
→ als Hex Zahl lesen

! Folgeradr. bei Jump beachten

MIPS - Assembler

MIPS = Microprocessor without Interlocked Pipeline Stages

MIPS = Load/Store-Architektur ; Register-Register-Maschine mit 32 celly. verwendbaren Registern

Register:

\$t0 - \$t7

Allzweckregister, falls nochmal benötigt Speichern notwendig

\$s0 - \$s7

-" -

hi / lo

Ergebnis von mult: höherwertige Bits in "hi"
niederwertige in "lo"
div: Ergebnis in "hi"
Rest in "lo"

\$v0, \$v1

Rückgabewerte von Unterprogrammen

\$zero

Konstante 0

\$ra

Rücksprungadr.

Datenformate:

Byte = 8 Bit

Byteorder je nach System

Wort = 32 Bit

Zahlen entw. unsigned oder ZKPL
(int)

Byteweise Adressierung: von einem Byte zum nächsten: Adr. +1

Halbwort

: Adr. +2

Wort

: Adr. +4

Registeradressierung: 0x123(\$t1) bedeutet: 0x123 + Wert in \$t1 als offset

Befehlsatz:

ZB. if \$s1 < \$s2 then
 \$t0=1 } slt \$t0, \$s1, \$s2 slt = set less than
 else }
 \$t0=0 }

> beq / bne: vgl. Register, verzweigen falls gleich/un gleich

> beqz / bnez: verzweigen wenn Register gleich/un gleich Null

ZB. if (i != j)
 i = i - j; } beq \$s1, \$s2, label1
 else Sub \$s3, \$s1, \$s2
 i = j ; } j label2
 label1: move \$s3, \$s2
 label2: ...

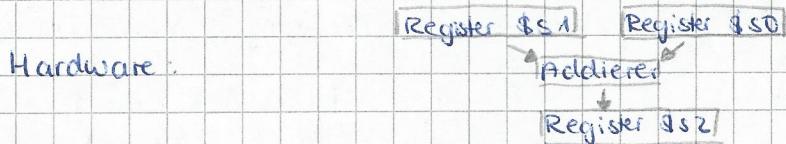
Pseudobefehl: Erweiterung der Assemblerbefehle, werden vom Assembler in Folge von Maschinenbefehlen übersetzt

Assemblerdirektive: Anweisung an Assembler zur z.B. Steuerung des Assemblierabgangs / Platzierung von Daten/...

Programmiersprache C

C- Programm : `int summe = a+b;`

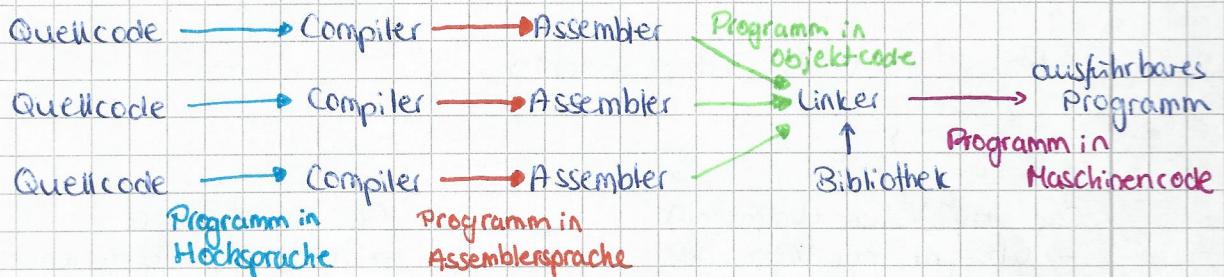
Maschinenbefehl / Assembler: add \$s2,\$s1,\$s0



Maschinensprache: Repräsentation von Anweisungen, die für Mikroprozessor unmittelbar verständlich sind (Folge von 0en und 1en)

Assembler-Sprache: Symbolische Repräsentation der Maschinensprache, die für Menschen anschaulich / verständlich ist
 ⇒ Symbolischer Befehl ≡ Maschinenbefehl

Quellcode → Programm:



Assembler: übersetzt Assemblersprache in Maschinensprache

Objektcode: enthält noch ungelöste Referenzen auf externe Unterprogramme / + evtl. Infos zum Debuggen Speicherbereiche

Linker: löst ungelöste Referenzen, verbindet alles zu ausführbarem Programm

C = Zwischenstellung zw. Assemblier und Hochsprache

→ gute Anpassung an Rechnerarchitektur + hohe Portabilität

Übersicht:

- > 4 Grundlegende Datentypen: char, int, float, double ! kein boolean / String
 - > keine Stringoperationen / Operationen auf zusammengesetzten Datentypen
 - > Zeiger (Pointer)
 - > signed / unsigned = Vorzeichen / Vorzeichen ignorieren

Operatoren:

*	Multiplikation	~	Not	>	größer
/	Division	<<	Links shift	>=	größer gleich
%	Modulo	>>	rechts shift	<	kleiner
+	Addition	&	AND	<=	kleiner gleich
-	Subtraktion	^	XOR	==	gleich
			OR	!=	ungleich

Linksshift um n ; mal 2^n

postfix / prefix Inkrement / Dekrement

postfix: z.B. $z = x++$
 $\rightarrow z = x$
 $x = x + 1$

prefix: $z = ++x$
 $\rightarrow x = x + 1$
 $z = x$

Kontrollstrukturen

> if: if (Bedingung) { ... } else { ... }

kurz: $x \# \text{Bedingung?} \quad x \text{true} : x \text{false}$

$\sim d \doteq \begin{cases} \text{if (Bed.) \{ } & x \text{true} \\ \text{else } & \{ x \text{false} \} \end{cases}$

> switch: switch (Ausdruck) {
 case a: ...
 break;
 case b: ...
 break;
 default: ...
 break;
 }

> while: while (Bedingung) { ... }

> do-while: do { ... }

{ while (Bedingung); }

> for: for (init; Bed.; update) { ... }

Call by Value (Normalfall)
 → übergibt nur Kopie des Parameters an Funktion
 → Funktion kann nur Wert der Kopie ändern
 → keine globale Parameteränderung

Call by Reference

- nur mit Zeigern realisierbar
- Funktion erhält Speicheradr. des Parameters
- Funktion ändert Speicherinhalt
 \rightsquigarrow ändert Parameter

Pointer: Variabie, deren Wert eine Speicheradr. ist

Deklaration: Datentyp * Variablename;

z.B. int *a (a ist vom Typ: Zeiger auf int -Value)

char **a ** = Pointer auf Pointer

\sim falls Datentyp noch unklar: void *

Zugriff: z.B. int *p

$\rightarrow \&a = \text{Adr. von } a \rightarrow p = \&a \rightarrow p \text{ Pointer auf } a$

$\rightarrow *p = \text{Wert auf den } p \text{ zeigt also } *p \neq a$

Instruction Set Architecture (ISA) Hardware - Software - Schnittstelle Beschreibung der Attribute und des funktionalen Verhaltens eines Prozessors

AUSFÜHRUNGSMODELL

Register - Register - Modell

- Operanden und Ergebnis in Allzweckregistern
- Dreiaadressformat: je eine Adr. für Operanden und Ziel z.B. add R1, R2, R3 $\rightarrow R1 = R2 + R3$
- Load/store - Architektur: bestimmte Befehle für Speicher \rightarrow Register / Register \rightarrow Speicher
z.B. load R2, A : $R2 \leftarrow \text{mem}[A]$
store C, R1 : $\text{mem}[C] \leftarrow R1$

Register - Speicher - Modell

- ein Operand im Speicher, einer im Register, Ergebnis in Register oder Speicher
z.B.
- Zweiaadressformat: Quellregister / Quellspeicherstelle ist Ziel z.B. add A, R1 : $\text{mem}[A] \leftarrow \text{mem}[R1]$
 $\text{add } R1, A : R1 \leftarrow R1 + \text{mem}[A]$

Akku - Register - Architektur

- Einadressformat: Akku als Quelle des einen Operanden und als Ziel
z.B. add A : $\text{akku} \leftarrow \text{akku} + \text{mem}[A]$
 $\text{addx } A : \text{akku} \leftarrow \text{akku} + \text{mem}[A+x]$

Keller - Architektur (Keller = Stack)

- Nulladressformat: Operanden auf obersten Kellererelementen
Ergebnis wieder auf Keller
z.B. add : $\text{tos} \leftarrow \text{tos} + \text{next}$ ($\text{tos} = \text{top of stack}$)

Speicher - Speicher - Architektur

- Dreiaadressformat
- Operanden + Ergebnis im Speicher z.B. add A, B, C : $\text{mem}(A) \leftarrow \text{mem}(B) + \text{mem}(C)$

DATENFORMATE

- Byte: 8 Bit
- Halbwort: 16 Bit
- Wort: 32 Bit
- Doppelwort: 64 Bit
- "Wort" je Hersteller unterschiedlich

DATENTYPEN

- Vorzeichenlose Dualzahl (bit unsigned int)
Standardformate $n=8, 16, 32, 64$
Wertebereich $0 \dots 2^n - 1$
- Zweierkomplement (signed int)
Standardformate $n=8, 16, 32, 64$
Wertebereich $-2^{n-1} \dots +2^{n-1} - 1$
MSB = Vorzeichenbit
- Bitfeld: variable Bitanzahl bis 32 Bit,
beweglicher "Ausschnitt" aus Speicher
 \rightarrow aus Speicher durch Byte-Adr. + Offset + Länge

SPEICHERADRESSIERUNG

Datenzugriff

- > Byte-adressierbarer Speicher: auf Byte / Wort / Halbwort direkt zugreifbar, Adressen beziehen sich auf Bytegrenzen (Byte-Adr.)
- > Wort-organisierter Speicher: Zugriffsbreite = Datenbusbreite

Data-alignment Ausrichten der Daten im Speicher

Datum aus s Bytes ausgerichtet im Speicher wenn Adr. A ganzzahliges Vielfaches von s ist, also $A \bmod s = 0$
 $\Rightarrow s = 2^i$: letzten i Bit der Byte-Adr. = 0

nicht ausgerichtet: Speicherung an beliebigen Byteadr.
 \Rightarrow lückenlose Speichernutzung bei Mischung der Datenformate

Datenanordnung im Speicher (Daten größer als ein Byte)

- > Little Endian Ordering: niedrigwertiges Byte an niedrigwertiger Adr. hochwertiges Byte an höchswertiger Adr.
- > Big Endian ordering: niedrigwertiges Byte an höchswertiger Adr. höchswertiges Byte an niedrigwertiger Adr.

Adresseierungstypen

- > Programmadr.: im Programm verwendete Adr.
- > Prozessadr.: wird aus Programmadr. erzeugt, immer noch keine "reale" Adr.
- > Maschinenadr.: = physikalische Adr., reale Position der Daten im Speicher übersetzung durch memory management unit (MMU)

BEFEHLSATZ: legt Grundoperation eines Prozessors fest

- > Befehlsformat definiert Codierung der Befehle
- > Befehlcodierung beginnt mit OPCODE

\Rightarrow Adressformate:
Dreiaddr. format: OPCODE Dest Src1 Src2
Zweiaddr. format: OPCODE Dest, Src1 Src2
Einaddr. format: OPCODE Src
Nulladdr. format: OPCODE

\Rightarrow Befehlsformate MIPS-Prozessor:

- > Typ R: Register-Register-Befehl 5bit Zielregister 6bit Codierung der Fkt

OPCODE	rs	rt	rd	shamt	funkt
31-26	25-21	20-16	15-11	10-6	5-0
5bit Quell-/Zielregister					5bit Shiftamount

- > Typ I: Immediate-Register-Befehl 5bit Quell-/Zielreg. unmittelbarer Wert / Adr. Verschiebung

OPCODE	rs	rt	Immediate / Adr.
31-26	25-21	20-16	15-0

Typ J: Jump

, 26 Bit Sprungadr.

Opcode 31-26	Target Adr. 25-0
-----------------	---------------------

CISC / RISC

> CISC = Complex Instruction Set Computers

- umfangreicher Befehlssatz, versch. Formate
- viele versch. Datentypen und Datenformate

→ hohe Entwicklungs kosten / lange Entwicklungsdauer

→ Befehlausführung verschieden lang (durch versch. Adressierungsarten)

→ komplexe Befehle schneller, führen zu kürzeren Programmen

> RISC = Reduced Instruction Set Computers

- einfache, oft genutzte Befehle so schnell wie möglich machen
- Operanden am Besten in großen Registern → schneller Zugriff
- Verzicht auf Mikroprogrammierung (so weit möglich)
- möglichst alle Befehle in einem Takt ausführbar
- möglichst einheitliches Befehlsformat
- typisch: Load / Store - Architektur

CISC	RISC
komplexe Befehle, Ausführung in mehreren Taktzyklen	Einfache Befehle, Ausführung in einem Taktzyklus
Jeder Befehl kann auf Speicher zugreifen	Nur Load-/Speicherbefehle können auf Speicher zugreifen
wenig Pipelining	intensives Pipelining
Befehle mit Mikroprogrammierung	Befehle durch festverdrahtete Hardware
Befehlsformat variabler Länge	alle Befehle feste Länge
Komplexität im Mikroprogramm	Komplexität im Compiler
einfacher Registersatz	mehrere Registersätze

RISC - PROZESSOR

Hinweis: Harvard-Architektur:

- getrennte Programm- und Datenspeicher → zwei Adress- und Datenbusse
- paralleles Holen von Operanden / Befehlen

Systembuschnittsstelle: Registerblocks für Daten und Adr. → gleichzeitiges Lesen eines Datums und Zwischen speichern eines Ergebnisses

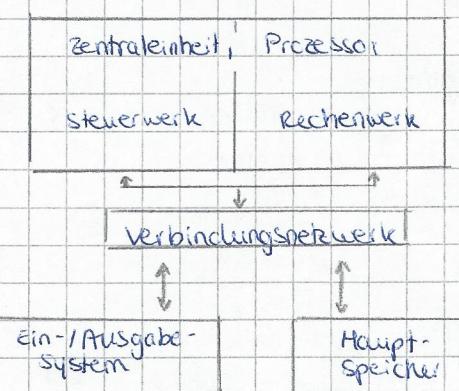
Steuerwerk: festverdrahtet, Befehlsregister als FIFO, Register für jede Pipeline-Stufe

Registersatz: große Zahl von Registern, erlaubt gleichzeitige Auswahl von 3-4 Registern

Rechenwerk: Operanden über Operandenbusse aus Registersatz in ALU, Ergebnis über Ergebnisbus in Registersatz

Rechnermodelle

Von-Neumann-Rechner



> Zentraleinheit (CPU): verarbeitet Daten gemäß Programm

- holt Befehle aus Speicher, dekodiert sie
- steuert Ausführung

Rechenwerk (ALU):

- führt arithm. /logische Operationen aus

> Hauptspeicher: jede Speicherzelle hat eindeutige Adr.
ein Speicher für Daten und Programme

> Verbindungsnetzwerk: Adress-/Daten-/Steuerleitungen, z.B. Busse

Phasen der Befehlausführung

> Holphase: lädt nächsten Befehl ins Befehlsregister

> Dekodierphase: Befehlsdecoder ermittelt Startadr. des Mikroprogramms, welches den Befehl ausführt

> Ausführungsphase: Mikroprogramm steuert Befehlausführung

ALU-Operationen

- arithmetische Operationen mit/ohne Übertrag
- logische Bitweise Verknüpfungen
- Schiebe-/Rotations-Operationen
- Transport-Operationen

Schiebeoperationen: logisches Linksschieben / Rechtsschieben \rightleftharpoons Mult. pl. / Division mit 2
~ VZ-Bit bleibt nicht erhalten

arithmetisches Rechtsschieben: VZ-Bit bleibt erhalten

Rotationsoperationen: Register als geschlossene Bitkette:

Adressregister

> Basisregister: enthält Anfangsadr. eines Speicherbereichs

> Indexregister: enthält Offset zu Basisadr. zur Auswahl eines bestimmten Datums im Speicherbereich

Pipeline - Verarbeitung

- Befehle werden in Teilaufgaben zerlegt mit spezieller Ausführungseinheit
 - ~> Maschinenoperationen können takt synchron bearbeitet werden
- idealerweise: Befehl in k-stufiger Pipeline wird in k Takt von k Stufen ausgeführt
 - ~> pro Takt ein neuer Befehl in Pipeline: k Befehle werden parallel bearbeitet

Maßzahlen:

> Latenz: Zeit, die ein Befehl benötigt um alle Pipeline - Stufen zu durchlaufen

> Durchsatz: Anzahl Befehle, die eine Pipeline pro Takt verlassen können

> Leistungssteigerung: n Befehle, k-stufige Pipeline

- ~> ohne Pipeline: $n \cdot k$ Taktzyklen
- ~> mit Pipeline: $n + (k-1)$ Taktzyklen (ideale Bed.)

$$\rightarrow \text{Leistungssteigerung } S = \frac{n \cdot k}{n + (k-1)} , \lim_{n \rightarrow \infty} S = k$$

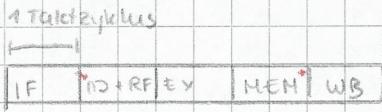
Durchsatz: Befehle pro Sekunde = Befehle pro Takt · Takte pro Sekunde

$$D = \frac{n}{n + (k-1)} \cdot \frac{1}{T} \quad T = \text{Länge eines Taktzyklus}$$

$$\lim_{n \rightarrow \infty} D = \frac{1}{T} = D_{\max}$$

DLX - PIPELINE

Pipeline Stufen



- > IF = Instruction Fetch:
 - durch Befehlszähler adr. Befehl wird aus Arbeitsspeicher / Befehlsspeicher geladen
 - Befehlszähler wird weitergeschaltet
- > ID / RF = Instruction Decode & Register Fetch:
 - aus Opcode des Maschinenbefehls werden Steuersignale erzeugt
 - Operanden werden aus Registerfile bereit gestellt (z.Takthilfe)
- > EX = Execute - Phase:
 - Operation wird auf Operanden ausgeführt
 - Load-/Speicherbefehle / Sprünge: ALU berechnet effektive Adr.
- > MEM = memory access:
 - Speicherzugriff bei Load-/Speicherbefehlen
- > WB = write back:
 - Ergebnis wird ins Registerfile geschrieben (1. Takthilfe)
 - Befehle ohne Ergebnis durchlaufen diese Phase passiv

Pipeline-Konflikte

- > **DATENKONFLIKTE**: treten auf, wenn benötigter Operand in Pipeline (noch) nicht verfügbar ist
 - ausgelöst durch Datenabhängigkeiten

Datenabhängigkeiten:

- > echte DA (true dependence) S^t :

zwischen zwei aufeinander folgenden Befehlen besteht S^t von Inst₁ zu Inst₂, wenn Inst₁ sein Ergebnis in ein Quellregister von Inst₂ schreibt

zB: S₁: add r₁, r₂, r₂ # r₁ = r₂ + r₂
 S₂: add r₄, r₁, r₃ # r₄ = r₁ + r₃

⇒ Lese-nach-Schreibe-Konflikt (RAW)

- > Gegnabh. (Antidependence) S^a :

zwischen zwei aufeinander folgenden Befehlen besteht S^a von Inst₁ zu Inst₂, wenn Inst₁ Daten aus Register liest, das anschließend von Inst₂ über schrieben wird

zB: S₂: add r₄, r₁, r₃ # r₄ = r₁ + r₃
 S₃: mul r₃, r₅, r₃ # r₃ = r₅ · r₃

⇒ Schreibe-nach-Lese-Konflikt (WAR)

(tritt zB auf, wenn in Pipeline Schreibstufe vor Lesestufe ist)

- > Ausgabeabhängigkeit (output dependence) S^o :

zwischen zwei aufeinander folgenden Befehlen besteht S^o von Inst₁ zu Inst₂, wenn beide in das gleiche Register schreiben

zB: S₃: mul r₃, r₅, r₃ # r₃ = r₅ · r₃
 S₄: mul r₃, r₆, r₃ # r₃ = r₆ · r₃

⇒ Schreibe-nach-Schreibe-Konflikt (WAW)

(tritt in Pipelines auf, die in mehreren Stufen schreiben)

→ kein WAR + WAW in DLx-Pipeline, da alle Befehle durch S laufen gehen (kein Überholen) und lesen immer in R-Stufe 2, schreiben immer in Stufe S ist

Lösungen:

- > Software-Lösung:
 - Einfügen von Leeroperationen (nop)
 - Instruction Scheduling → umordnen der Befehle

Hardware-Lösung:

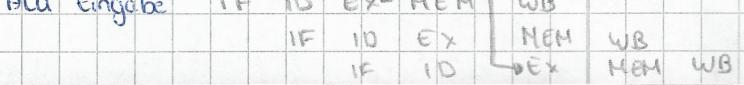
- Interlocking: Pipeline für zwei Takte anhalten, verlaufen lassen

- Forwarding: Operand nicht aus Register, sondern aus ALU oder vorherigen Operation holen

RESULT FORWARDING: Rückführung der ALU-Ausgabe auf ALU-Eingabe



LOAD FORWARDING: Rückführung des Ergebnisses nach MEM-Phase auf ALU-Eingabe



- > RESSOURCENKONFLIKTE: treten auf, wenn zwei Pipeline -Stufen dieselbe Ressource benötigen, auf diese aber nur einmal zugegriffen werden kann
 - ~> treten bei DLX - Pipeline nicht auf

Lösungen:

- Interlocking : Arbitrierungslogik hält Befehl an, der Konflikt verursacht
- Überlaktung: Ressource, die Konflikt hervorruft schneller Takte als übige Pipeline - Stufen
- Ressourcenreplizierung: Vervielfachung der Ressourcen

- > STEUERFLÜSSKONFLIKTE: treten auf, wenn in Befehlsbereitstellungsphase Zieladr. des nächsten Befehls noch nicht berechnet ist bzw. wenn noch nicht klar ist, ob benötigter Sprung genommen wird
 - ~> ausgelöst durch Steuerflussabhängigkeiten

treten in DLX - Pipeline auf, da Programmsteuerbefehl erst in ID - Phase erkannt wird
 ~> nächster (evtl falscher) Befehl ist schon in der Pipeline
 ~> Sprungziel wird erst in EX - Phase berechnet, PC erst am Ende der MEM - Phase ersetzt (PC = Program Counter)
 => insg. 3 Befehle in Pipeline die ggf. gelöscht werden müssen
 ~> delay slots nach genommenen Sprung
 ABER: Standardmäßig werden 3 Befehle ausgeführt in DLX

Lösungen:

- > Software - Lösungen:
 - delay slots nach Sprungbefehl
 - drei Leerbefehle nach jedem Programmsteuerbefehl (so bei DLX - Pipeline)
 - Compiler verschiebt Befehle vor Sprung in delay slots
- > Hardware - Lösungen:
 - Pipeline - Leerlauf: Hardware erkennt Verzweigungsbefehl in ID - Phase, lässt keine neuen Befehle in Pipeline bis Sprungentscheidung getroffen ist
 - Spekulation: Vorhersage ob takeit / not takeit

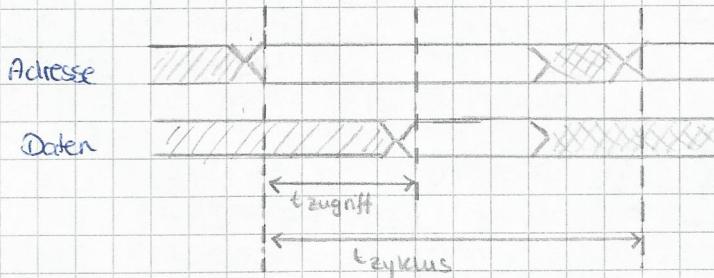
Halbleiterspeicher

Speicherorganisation: $n = \text{Anzahl Zeilen}$
 $m = \text{Anzahl Spalten (Speicherelemente pro Zeile)}$
 $\rightarrow \text{Angabe in Form } n \times m \text{ Bit}$

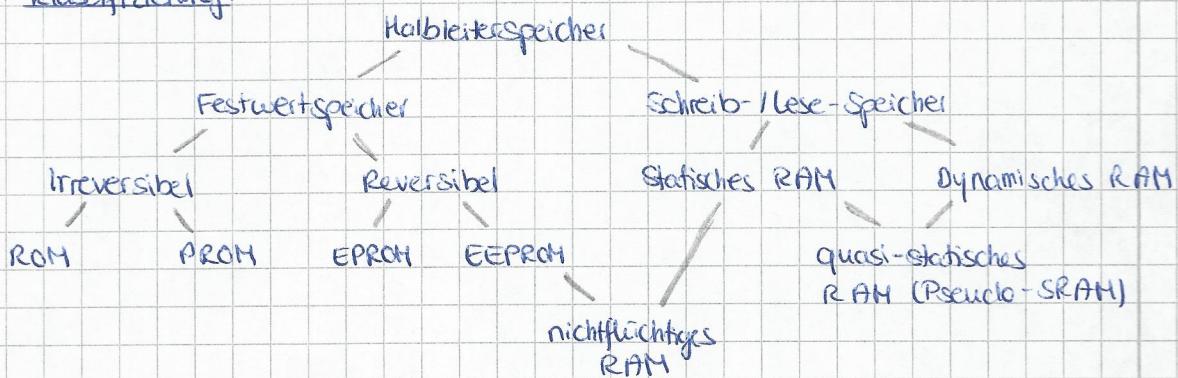
Kapazität: Informationsmenge (in Bit), die im Speicher untergebracht werden kann: $n \times m \text{ Bit}$

Arbeitsgeschwindigkeit:

- Zugriffszeit: max. Zeitdauer zwischen Anlegen einer Adr. an Speicher und Ausgabe der gewünschten Daten
- Zykluszeit: min. Zeitdauer, die zwischen zwei Adr.-Anfragen an den Speicher vergehen muss



Klassifizierung:



Speicherzugriff durch Angabe der Zeile (RAS) und Spalte (CAS)

	SRAM	DRAM
Speicherungsart	in Flipflops, benötigt 6 Transistoren	in Kondensator, benötigt 1 Transistor
Pro	nur zum Umschaltzeitpunkt Stromfluss hohe Leistungsaufwand	klein hohe Integrationsdichte
Contra	teuer	Lesen bewirkt Entladung \rightarrow muss neu eingeschrieben werden periodische Aufladung nötig (sonst Datenverlust)

Cache-Typen

> VOLL-ASSOZIATIVER CACHE

- jede Adr. kann in jeder Zeile stehen (beliebige Platzwahl)
- alle Zeilen gleichberechtigt
- Komparator für jede einzelne Zeile

Adresse :	Tag	Wort-Anwähl	Byte-Anwähl
-----------	-----	-------------	-------------

- (1) Vergleiche Tag mit Tags bei im Adr. Speicher
- (2) bei hit: prüfe valid bit → ist Zeile gültig?
- (3) hole Daten, wähle gesuchtes Wort

> DIRECT-MAP CACHE

- jede Zeile aus Hauptspeicher wird einer Zeile aus Cache zugeordnet
- 1 Vergleich

Adresse :	Tag	Zeilenindex	Wort-Anw	Byte-Anw
-----------	-----	-------------	----------	----------

- (1) Zeile mit diesem Zeilenindex suchen
- (2) Komparator: stimmen Tags überein, ist Zeile gültig
- (3) hole Daten und wähle gesuchtes Wort
- (2) (3) können gleichzeitig ablaufen

> n-WAY SET-ASSOZIATIV CACHE

- mehrere Cachezeilen werden zu Sätzen zusammengefasst, n = Anzahl Zeilen pro Satz
- n Komparatoren, da innerhalb eines Satzes vollauf.

Adresse :	Tag	Satzindex	W. Anw	Byte-Anw
-----------	-----	-----------	--------	----------

- (1) Satz mit diesem Satzindex suchen
- (2) Tags vergleichen, ist Zeile gültig
- (3) hole Daten, wähle gesuchtes Wort

Ersetzungsstrategien welcher Cache-Eintrag soll bei Cache miss überschrieben werden?

> FIFO/LIFO : ~~erst~~ ersetze ältesten/jüngsten Eintrag

> LRU = least recently used : ersetze Eintrag, der am längsten nicht mehr benutzt wurde

analog: MRU = most recently used

> Pseudo-LRU : Second Chance Bit: bei Hit wird Bit gesetzt;

Verdrängungsfall: hat aktuelle Pos. gesetztes Bit: lösche Bit, zeiger zur nächsten Pos., ... so lange bis Stelle ohne Bit

> LFU = least frequently used : ersetze Eintrag, der am wenigsten häufig benutzt wurde

analog: MFU

> Random: ersetze zufälligen Eintrag

Kohärenz

Kohärenzproblem:

- System konsistent, wenn alle Kopien eines Datums im Hauptspeicher und allen Caches identisch

→ Bus-Schnüffeln (Bus-Snooping)

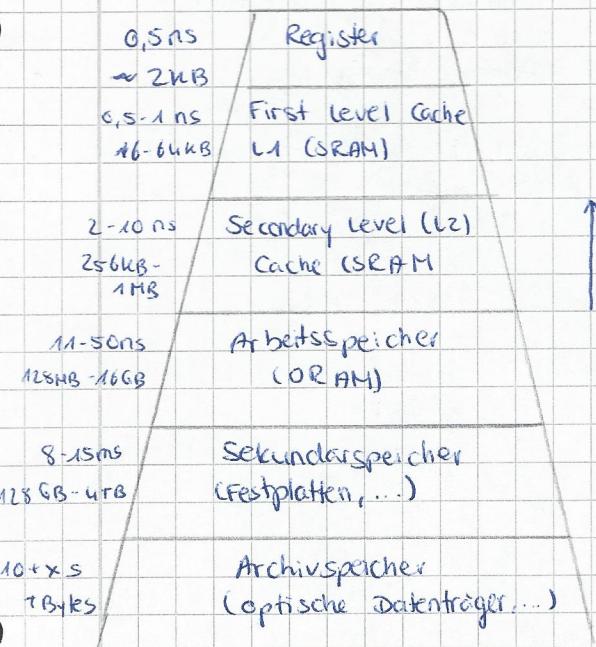
- Schnüffellogik eines Caches hört am Bus Adr. mit, die anderen Caches auf den Bus legen, vergleicht sie mit eigenen

- falls gleiche Adr. im eigenen Cache modifiziert wurde: bricht Bustransaktion des anderen Caches ab, sendet 'retry'-Signal

→ sp. schreibe modifizierten Inhalt in Hauptspeicher

Cache-Speicher

Kurze Zugriffszeiten, wird durch Speicherhierarchie wie ein großer, schneller Speicher



↑ zunehmende Worte/Byte
↓ abnehmende Kapazität
↓ abnehmende Zugriffsspitze

Größe des Caches bestimmt Zugriffsspitze

- > verschiedene große Caches
- > teilweise bis 14 Caches (L3, L4 meistens off-chip)
- > häufig getrennte L1 Caches für Befehle und Daten

Lokalitätseigenschaften:

Zeitliche Lokalität: Informationen, die in naher Zukunft angesprochen werden, wurden mit großer Wahrscheinlichkeit schon früher einmal angesprochen

Ortliche Lokalität: zukünftiger Zugriff wird mit großer Wahrscheinlichkeit in der Nähe des bisherigen Zugriffs liegen

-> Cachespeicherverwaltung sorgt dafür, dass sich am häufigsten benötigte Daten im Cache befinden

Lesezugriffe:

Cache hit:

Datum im Cache, kann ohne Wartezyklen aus Cache entnommen werden

read miss:

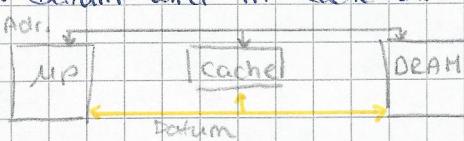
Datum nicht im Cache, wird mit Wartezyklen aus Arbeitsspeicher gelesen und in Cache eingefügt

Schreibzugriffe: Datum wird entweder in Cache oder Arbeitsspeicher oder beides geschrieben

write miss: Datum nicht im Cache, soll neu in Cache geschrieben werden
no alter Wert muss zuerst aus Hauptspeicher geladen werden

-> Caches speichern im avg. größere Blöcke als nur Wörter (Cache-Block/-Line)

Durchschreibverfahren: Datum wird in Cache und Arbeitsspeicher geschrieben (write through)



Gepuffertes Durchschreibverfahren: kleiner Schreib-Puffer nimmt zu schreibende Daten (buffered write through) temporär auf
 -> werden automatisch in Hauptspeicher übertragen während Prozessor parallel weiterarbeit kann

Rückschreib-Verfahren: Datum wird nur in Cache geschrieben und durch dirty bit gekennzeichnet
 -> wird in Arbeitsspeicher geschrieben wenn so gekennzeichnetes Datum aus Cache verdrängt wird

	write through	buff. wr. th.	write back
Vorteile	Konsistenz zw. Cache und Arbeitsspeicher	schneller als ohne Puffer	schnelle Zykluszeit des Cache genutzt geringere Last auf Systembus
Nachteile	Schreibzugriffe benötigen immer längsame Zykluszeit d. Arbeitsspeichers, belasten Systembus		Konsistenzprobleme zw. Cache und Arbeitsspeicher

Konsistenzprobleme: u.U. veraltete Daten im Hauptspeicher, die von CPU schon längst geändert wurden
 ed. alte Daten im Cache, aber neuer in Hauptspeicher

Hit-Rate: Anzahl Treffer / Anzahl Zugriffe

mittlere Zugriffszeit: $t_{access} = (\text{Hit-Rate}) \cdot t_{hit} + (1 - \text{Hit-Rate}) \cdot t_{miss}$

t_{hit} = Zugriffszeit auf Cache

t_{miss} = Zugriffszeit auf alles nach Cache

Cache-Aufbau

- zwei Speichereinheiten:

> Datenspeicher: enthält die im Cache abgelegten Daten

> Adressspeicher: enthält Adr., wo diese Daten im Arbeitssp. stehen

- jeder Dateneintrag besteht aus Datenblock (Cache-line, bis 64 Byte)

~ Umgebung eines gewünschten Datums wird mit eingelagert

~ im Adr. speicher Basisadr. jedes Blocks

- jede Cache-Zeile enthält (Adress, Daten)-Paar und Statusbits (valid, dirty)

- Cache-Tag: enthält Basisadr. des aktuellen Blocks in Hauptspeicher

- Ermittlung ob Datum im Cache: Komparator (Adr. vergl.)

Blockgröße: n Bit Byte-Offset \Rightarrow Blockgröße = 2^n Byte

Anzahl Einträge: Kapazität / Blockgröße = Zeilen im Cache

Cache-Organisation: n Bit Index $\rightarrow 2^n$ Cache-Sätze adr. bar

$$\rightarrow \text{Assoziativität} = \frac{\text{Anzahl Einträge}}{\text{Cachesatz}}$$

Byte-Offset: 2^n Byte $\rightarrow n$ Bit Byte-Offset

= Blockgröße

Speicherbedarf: Speicherbed. pro Zeile = Tag + Statusbits + Daten pro Zeile ! Einheiten

$$\approx \text{ges. : } \downarrow \cdot \underbrace{\text{Satzanzahl}}_{\text{Anzahl Zeilen im Cache}} \cdot n \quad \text{wobei } n = \text{Zeilen pro Seite}$$

Wohin wird Block abgebildet?

$$\text{Blocknr.} = \text{MS-Adr.} \text{ div } \text{Blockgröße}$$

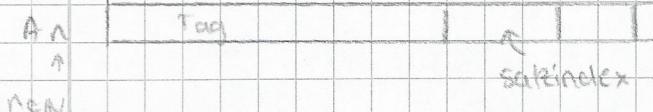
$$\text{Zeilennr.} = \text{Blocknr.} \text{ mod } \text{Zeilenanzahl}$$

$$\text{Satznr.} = \text{Blocknr.} \text{ mod } \text{Satzanzahl}$$

Taglänge: Adressgröße - $\lceil \log(\text{Satzanzahl}) \rceil - \lceil \log(\text{Blockgröße}) \rceil$

Bits für Satzindex

Bits für Byte-Offset



Virtuelle Speicherverwaltung

- Adressunterteilung:

Virt. Seitenr. | Byte-Offset

2ⁿ Bit Seitengröße → n Bit Byte-Offset

physikalische Adr. ermitteln:

- (1) virtuelle Adr. als Dualzahl schreiben
- (2) ersten Bits (je nach Anzahl) als virtuelle Seitennummer
- (3) in Tabelle nachschauen welche physik. Seitenr. zugeordnet ist
- (4) erzebe Bits der virt. Seitenr. durch Dualrepräsentation der physikalischen Seitenr. (\cong physik. Seitenr. mal Seitengröße) addiere Byte-Offset

TLB = Translation Lookaside Buffer

Speichert letzten paar Übersetzungen um Zugriffe / Adr. Berechnungen zu beschleunigen

→ Beschleunigung erst ab zweitem Zugriff und solange Einträge aus TLB noch nicht verdrängt

Hauptaufg. Betriebssystem:

- Speicherverwaltung
- Auftragsverwaltung
- Betriebsmittelverwaltung

Komponenten von -Neumann - Rechner:

- Steuer-, Rechenwerk
- Speicher
- Bus
- Ein-/Ausgabe-Einheiten