

Technische Informatik

Wissen

Inhaltsverzeichnis

[Inhaltsverzeichnis](#)

[Für die Prüfung](#)

[Aufgaben](#)

[Klausur WS2011/2012](#)

[Wissen](#)

[Beweise](#)

[Einerkomplement](#)

[Zweierkomplement](#)

[Boolsche Algebra](#)

[Gesetze](#)

[Timing](#)

[Timing Physikalisch](#)

[Timing von Bausteinen](#)

[Timing Reti](#)

[Zahlensysteme](#)

[Festkommazahl](#)

[Betrag- und Vorzeichen](#)

[Einerkomplement](#)

[Zweierkomplement](#)

[Gleitkomma-Zahlen](#)

[Zusammensetzung der Gleitkommazahlen nach IEEE 754](#)

[Verteilung der Zahlen](#)

[Normalisierte Gleitkommazahlen](#)

[Definition einer Normalisierten Gleitkommazahl](#)

[Dezimalwert einer Normalisierten Gleitkommazahl berechnen](#)

[Normalisieren einer Gleitkommazahl:](#)

[Fehlertoleranz](#)

[Parity-Code](#)

[Fehlererkennender Code](#)

[Fehlerkorrigierender Code](#)

[Hamming-Abstand](#)

[Hamming-Code](#)

[Huffman-Code](#)

[Binary Decision Diagram \(BDD\)](#)

[Ordered Binary Decision Diagram \(OBDD\)](#)
[Reduced Ordered Binary Decision Diagram \(ROBDD\)](#)
[ON-Menge, Minterm, Maxterm, Literal, Monom und Polynom](#)
[ON-, OFF-Menge](#)
[Maxterm](#)
[Literal](#)
[Monom](#)
[Minterm](#)
[Polynom](#)
[Implikant und Primimplikant](#)
[Minimierung](#)
[Programmable Logic Array \(PLA\)](#)
[Kosten von Monomen](#)
[Kosten von Polynomen](#)
[Die primären Kosten](#)
[Die sekundären Kosten](#)
[PLA-Beispiel:](#)
[Problem der 2-stufigen Logikminimierung](#)
[Mehrdimensionale Würfel](#)
[Konstruktion einer Hypercubes](#)
[Berechnung von Minimalpolynomen \(Quine/McCluskey\)](#)
[Quine/McCluskey](#)
[Quine McCluskey Visualisierung auf dem Cube](#)
[Matrix-Überdeckungsproblem](#)
[Primimplikantentafel](#)
[Petrick's Methode](#)
[„Greedy-Heuristik“ zur Lösung von Überdeckungsproblemen](#)
[Normalformendarstellungen](#)
[Wahrheitstafeldarstellung](#)
[Disjunktive und konjunktive Normalform](#)
[Reduced Ordered Binary Decision Diagram \(ROBDD\)](#)
[Schaltungssynthese](#)
[Logikgatter](#)
[Zweistufige Schaltungssynthese](#)
[BDD-basierte Schaltungssynthese](#)
[Wie man mit dem NAND-Gatter alle anderen Logikgatter baut](#)
[Formale Definition eines Schaltkreises](#)
[Complementary Metal Oxide Semiconductor \(CMOS\)](#)
[Bewerten von Schaltungen](#)
[Stufigkeit einer Schaltung](#)
[Schaltungsgröße](#)
[Flächenmetrik](#)
[Laufzeitmetrik](#)
[Fortgeschrittene BDDs mit IfThenElse \(ITE\) und mehr](#)

Schaltungen

Addierer
Halbaddierer
Volladdierer
Carry-Ripple Addierer
Bus
nBit-Inkrementer
nBit-Multiplexer
Conditional Sum Addierer
Subtrahierer
SRAM
RS-Flipflop
D-Latch
D-Flipflop
n-Bit Register
Schieberegister
Zähler
n-Bit Zähler
Treiber

CISC vs. RISC

Caching

Lesezugriff
Cache als assoziativer Speicher
Verdrängen alter Daten aus dem Cache
Direct Mapped Cache
Schreibzugriff

Automaten

Datenpfade

Datenpfade Timing Diagramm
Schritte mit Instruktionsformat kombinieren
Haltetermine
Bus contention

Lösungen

Klausur WS2011/2012

Aufgabe 1
Aufgabe 2
Aufgabe 3
Aufgabe 4
Aufgabe 5
Aufgabe 6
Aufgabe 7
Aufgabe 8
Aufgabe 9
Aufgabe 10

[Aufgabe 11](#)

Für die Prüfung

Datum: 18.09.2012
Zeit: 14:00 Uhr
Ort: 101-Gebäude
Hilfsmittel: keine zugelassen
wichtig: Studentenausweis nicht vergessen

Aufgaben

Klausur WS2011/2012

Aufgabe	Thema	Erledigt?	Besprochen?	Aufbereitet/Bearbeitet von
1	Hamming			
2	Zahlensysteme			
3	PLA			
4	McCluskey			
5	Automaten			
6	FlipFlop			
7	Timing			
8	Pfade			
9	Direct -Mapped -Cache			
10	Boolesche Algebra			
11	BDD	x		
?	Huffman	x		

Detail-Liste:

1. Hamming-Codes, Parity-Codes

2. Einer- und Zweierkomplement (mit Beweis)
3. PLAs, Schaltkreiskonstruktion, formale Definition Schaltkreis
4. Quine-McCluskey
5. Mealy-Automat, Schaltwerkkonstruktion aus STD
6. RS-FlipFlop, Schreibpulsweite (spikefreies Schalten), Verzögerungszeiten
7. Timing ReTI
8. Pfade in der ReTI
9. Direct-Mapped-Cache
10. Boolesche Algebra Beweis
11. Addierer, Tiefe und Kosten eines Schaltkreises, BDDs
12. Huffman-Code

Wissen

Beweise

Einerkomplement

Beweis:

$$\begin{aligned}[d']_1 + [d]_1 &= \\ &= \left(\sum_{i=-k}^{n-1} d_i \cdot 2^i \right) - d_n \cdot (2^n - 2^{-k}) + \left(\sum_{i=-k}^{n-1} d'_i \cdot 2^i \right) - d'_n \cdot (2^n - 2^{-k}) \\ &= \left(\sum_{i=-k}^{n-1} (d_i + d'_i) \cdot 2^i \right) - (d_n + d'_n) \cdot (2^n - 2^{-k}) \\ &= \left(\sum_{i=-k}^{n-1} (d_i + (1 - d_i)) \cdot 2^i \right) - (d_n + (1 - d_n)) \cdot (2^n - 2^{-k}) \\ &= \sum_{i=-k}^{n-1} 2^i - (2^n - 2^{-k}) \\ &= (2^n - 2^{-k}) - (2^n - 2^{-k}) \\ &= 0\end{aligned}$$

Quelle: http://nirvana.informatik.uni-halle.de/~molitor/pearson/7092/vorlesung/kapitel_03/kapitel3.pdf

Zweierkomplement

$$\begin{aligned}
 [d']_2 + [d]_2 &= \\
 &= \left(\sum_{i=-k}^{n-1} d_i \cdot 2^i \right) - d_n \cdot 2^n + \left(\sum_{i=-k}^{n-1} d'_i \cdot 2^i \right) - d'_n \cdot 2^n \\
 &= \left(\sum_{i=-k}^{n-1} (d_i + d'_i) \cdot 2^i \right) - (d_n + d'_n) \cdot 2^n \\
 &= \sum_{i=-k}^{n-1} 2^i - 2^n \\
 &= (2^n - 2^{-k}) - 2^n \\
 &= -2^{-k}
 \end{aligned}$$

$$\implies [d']_2 + 2^{-k} = -[d]_2$$

Quelle: http://nirvana.informatik.uni-halle.de/~molitor/pearson/7092/vorlesung/kapitel_03/kapitel3.pdf

Boolsche Algebra

Gesetze

Kommutativgesetze	(1) $a \wedge b = b \wedge a$	(1') $a \vee b = b \vee a$
Assoziativgesetze	(2) $(a \wedge b) \wedge c = a \wedge (b \wedge c)$	(2') $(a \vee b) \vee c = a \vee (b \vee c)$
Idempotenzgesetze	(3) $a \wedge a = a$	(3') $a \vee a = a$
Distributivgesetze	(4) $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	(4') $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
Neutralitätsgesetze	(5) $a \wedge 1 = a$	(5') $a \vee 0 = a$
Extremalgesetze	(6) $a \wedge 0 = 0$	(6') $a \vee 1 = 1$
Doppelnegationsgesetz (Involution)	(7) $\neg(\neg a) = a$	
De Morgansche Gesetze	(8) $\neg(a \wedge b) = \neg a \vee \neg b$	(8') $\neg(a \vee b) = \neg a \wedge \neg b$
Komplementärgesetze	(9) $a \wedge \neg a = 0$	(9') $a \vee \neg a = 1$
Dualitätsgesetze	(10) $\neg 0 = 1$	(10') $\neg 1 = 0$
Absorptionsgesetze	(11) $a \vee (a \wedge b) = a$	(11') $a \wedge (a \vee b) = a$

Timing

Timing Physikalisch

Muss noch deutlich verbessert werden, so dass man es versteht.

Das Wissen stammt aus dem Script vom WS2011/12.

Bitte nicht so ohne weiteres Löschen. (Das Aufschreiben hat viel Arbeit gemacht.)

Beim Timing geht es darum wie schnell geschaltet werden kann ohne das es zu Fehlern kommt. Also wie hoch die Taktfrequenz maximal sein darf, damit es zu keinen ungewollten physikalischen Effekten kommt.

Nehmen wir als Beispiel einen einfachen Schaltvorgang:

Der Wechselzeitpunkt sei:	t_0
Unsere Schaltung habe drei Eingänge:	A, B, E
Und es wird geschalten von der Eingangsbelegung: 110	
Zur der Eingangsbelegung:	101

(Was die Schaltung macht ist uns hier egal.)

t_{PLH} = Pegel Low-High (Verögerungszeit bei 0->1) (am Ausgang)

t_{PHL} = Pegel High-Low (Verögerungszeit bei 1->0) (am Ausgang)

Spike = Unerwünschter Schaltzugang aufgrund vorhergehender zögernder Schaltelemente

Zur Umgehung gibt es zwei Möglichkeiten:

1. Nicht von 10 auf 01 direkt schalten. Also Zwischenschritt einfügen: 10->00->01
2. Einen sicheren Abstand einfügen (hier: 0.12 ns + $2^* \varphi = 0.38$ ns)

Versorgungsspannung VCC = 1.1V

Zwei Input-Pegel:

- V_{IH} (V in High)

- V_{IL} (V in Low)

Spannung U ist Element von [0, VCC]

Aus der Spannung U bilden wir den logischen Wert I(U):

$$I(U) = 0, \text{ falls } U \leq V_{IL}$$

$$I(U) = 1, \text{ falls } U \geq V_{IH}$$

$$I(U) = \text{undef}$$

Bsp: NanGate:

$$V_{IL} = 30\% * VCC = 0.33V$$
$$V_{IH} = 70\% * VCC = 0.77V$$

Will man den Ausgang eines Gatters mit dem Eingang eines anderen Gatters verbinden, so muss gelten:

$$V_{OL} \text{ (von Ausgang)} \leq V_{IL} \text{ (von Eingang)}$$
$$V_{OH} \text{ (von Ausgang)} \geq V_{IH} \text{ (von Eingang)}$$

Liegt eine feste Spannung M an mit $V_{IL} < M < V_{IH}$
so lassen sich die Zeiten t_1, t_2 bestimmen durch: $x(t_1) \rightarrow y(t_2) = M$

$$\text{propagation delay (Verzögerungszeit)} \quad t_p = t_2 - t_1$$

Bsp: NanGate: $M = 0.55V$
Bausteine: NAND, NOT, AND, OR, EXOR
 t_p zwischen 0.00 und 0.21 ns (1 ns = 10^9 sec)

Werte müssen gemessen werden:

$$\begin{array}{ll} \text{Anstiegszeit} & = \text{rise time} \\ \text{Abfallzeit} & = \text{fall time} \end{array}$$

Falls am Gattereingang: rise time, fall time $\leq \varphi$
Dann gilt am Gatterausgang: rise time, fall time $\leq \varphi$

Beispiel: NanGate: φ ca. 0.13 ns

Start-Zeitpunkt $m(0)$
 $m(0) + t_{PLH}$ = Endzeitpunkt von Gatter 1
 $m(0) + t_{PLH} + t_{PLH}$ = Endzeitpunkt von Gatter 1

M ist auch Mittelwert

φ ist größer-gleich der Abfall- / Anstiegszeit

Durchläuft $X(t)$ nach Zeit $m(0)$ die Spannung M,
dann durchläuft $Y_{n(t)}$ nach der Zeit $m(0) + n * t_{PLH}$ die Spannung M

Falls $X(t)$ mit Anstiegszeit $\leq \varphi$
dann existiert auch $Y_{1(t)}, \dots, Y_{n(t)}$.

Also ist Y_n auf jeden Fall zur Zeit $m(0) + n * t_{PLH} + \varphi$ logisch 1

Allgemein:

Die Zeiten, an denen die entsprechenden Signale wohldefinierte logische Werte 0,1 annehmen, unterscheiden sich von denen für M um höchstens φ .

Eine rise/fall time $\leq \varphi$ an den Primären Eingängen einer Schaltung kann man garantieren wenn man den Schaltvorgang zur Zeit t_0 beginnt und spätestens(?) zur Zeit $t_0 + \varphi$ abschließt.

Beginnt man im Beispiel den Schaltvorgang bei t_0
und beendet ihn bei $t_0 + \varphi$
dann gilt $m(0) \leq t_0 + \varphi$
und Y_n ist spätestens nach $t_0 + n * t_{PHL} + 2 * \varphi$ logisch 1

Einfluss auf Verzögerungszeiten:

- Betriebstemperatur
- Fertigung des Chips
- kapazitive Last am Gatterausgang (Fanout)

=> Hersteller geben keine festen Zeiten t_{PLH}/t_{PHL} an
sondern 3 Werte:

- τ^{min} = untere Schranke
- τ^{max} = obere Schranke
- τ^{typ} = typischer Wert

für die tatsächliche Verzögerungszeit t_p gilt:

$$\tau^{min} \leq t_p \leq \tau^{max}$$

Wir nehmen an das t_p im Intervall (τ^{min}, τ^{max}) liegt
wenn Temp zwischen 0°-70° C + Fanoutbeschränkung von 10.

Für τ^{typ} gilt ebenfalls $\tau^{min} \leq \tau^{typ} \leq \tau^{max}$.

Aber mit τ^{typ} rechnen wir nicht,
sondern mit dem Intervall (τ^{min}, τ^{max}) .
(Sonst: Fehler unbekannter Größe)

Zur Zeit (a, b) heißt: frühestens zur Zeit a, spätestens zur Zeit b
Zur Zeit a = zur Zeit a, a)

$$\min(a, b) = a$$

$$\max(a, b) = b$$

$$(a, b) + (c, d) = (a+c, b+d)$$

t_{PLH}, t_{PHL} haben jeweils ein min und ein max-Wert

Beispiel-Rechnung (Wechsel von 0 auf 1):

$$\begin{aligned} t_1 &= t_0 + (\tau_{LH}^{\min} \text{ (von AND)}, \tau_{LH}^{\max} \text{ (von AND)}) \\ &= t_0 + (0.02, 0.12) \\ t_2 &= t_1 + (\tau_{LH}^{\min} \text{ (von AND)}, \tau_{LH}^{\max} \text{ (von AND)}) \\ &= t_0 + 2*(0.02, 0.12) \\ &= t_0 + (0.04, 0.24) \end{aligned}$$

Beispiel-Rechnung 2 (Unbekannt welche (und wieviele) Eingänge wie wechseln):

Für Gatter v gilt:

$$\begin{aligned} \tau^{\min} &= \min(\tau_{LH}^{\min}, \tau_{HL}^{\min}) \\ \tau^{\max} &= \max(\tau_{LH}^{\max}, \tau_{HL}^{\max}) \end{aligned}$$

Schaltzeit von Gatter v:

$$v = (\min(a_1, a_2), \max(b_1, b_2)) + (\tau^{\min}, \tau^{\max})$$

(Wobei a_1, b_1 für Gatter 1 gilt und a_2, b_2 für Gatter 2)

$$A, B, E: \quad t_0 + (0.0, 0.0)$$

$$C: \quad t_0 + (0.02, 0.12)$$

$$D: \quad (t_0 + 0.0, t_0 + 0.12) + (0.02, 0.12) = (t_0 + 0.02, t_0 + 0.24)$$

Der Übergang von (0, 1) auf (1, 0) bzw. umgekehrt ist der einzige bei dem an AND/NAND-Gattern ein Spike auftreten kann.
(Wenn ein AND/NAND als Input des zweiten verwendet wird)

$$\delta = 13$$

$$\text{Sicherer Abstand: } 0.12 + 2 * \delta = 0.38$$

Lösung für Spikefreies Umschalten (bei 1,0 auf 0,1):

Zuerst wird A auf 0 gesenkt bei $t_0 = 0$

Zum Zeitpunkt $t_1 = t_0 + 0.12 + \delta$ hat C spätestens auf 0 geschalten (wenn es nicht vorher schon so war)

B ist zum Zeitpunkt t_1 weiterhin 0 und etwas später
zum Zeitpunkt $t_2 = t_1 + \delta$ hebt man es auf 1.
=> hieraus folgt der oben angegebene sichere Abstand

$\delta = 0.13$ ist ein fester Gatter-unabhängiger Wert!

Timing von Bausteinen

Das Timing von Bausteinen fehlt praktisch noch komplett:
Sollte möglichst praxisnah aufgeschrieben werden.

lck_{pre}
lck
/DCLdoe_{pre}
/DCLdoe
ck
/ck
s₀
s₁
E
I[31:24]
ACC[31:0]
/reset

$t_{\bar{p}}^+$ = Verzögerungszeit von Ck bis Q
 $t_{\bar{p}}$ = Verzögerungszeit von Ck bis Q (von /Ck angesteuert)

t_{SDC}^+ = Setupzeit von D bis Ck
 t_{SDC} = Setupzeit von D bis /Ck

t_{HCD}^+ = Holdzeit von D bis Ck
 t_{HCD} = Holdzeit von D bis /Ck

Timing Reti

Zykluszeit :

$$\tau \geq 71.0 \text{ ns}$$

Taktfrequenz :

$$v = \frac{1}{\tau} \cdot 10^9 \text{ Hz} = 14.1 \text{ MHz}$$

8 Takte pro Befehl →
1.76 Millionen Befehle pro Sekunde,
d.h. Befehlsrate von 1.76 MIPS
(= Million Instructions Per Second)

Zahlensysteme

Triplet $S = (b, Z, \delta)$ mit $b \in N$ und $b \geq 2$ als Basis, Z eine b -elementige Menge von Symbolen, $\delta: Z \rightarrow \{0, 1, \dots, b-1\}$ eine Abbildung, die jeder Ziffer umkehrbar eindeutig eine natürliche Zahl zwischen 0 und $b-1$ zuordnet.

Festkommazahl

Besteht aus $n+1$ Vorkommastellen und $k \geq 0$ Nachkommastellen.

Betrag- und Vorzeichen

$$[d_n, d_{n-1}, \dots, d_0, \dots, d_{-k}]_{BV} := (-1)^{d_n} \sum_{i=-k, \dots, n-1} d_i 2^i$$

Man erhält für a die inverse Zahl, indem man das erste Bit komplementiert. Benachbarte Zahlen haben den gleichen Abstand 2^{-k} . Beispiel: $n=2, k=0$ 001 = 1; 101 = -1

Einerkomplement

$$[d_n, d_{n-1}, \dots, d_0, \dots, d_{-k}]_1 := \sum_{i=-k, \dots, n-1} d_i 2^i - d_n (2^n - 2^{-k})$$

Man erhält für a die inverse Zahl, indem man alle Bits komplementiert. Benachbarte Zahlen haben den gleichen Abstand 2^{-k} . Beispiel: $n=2, k=0$ 001 = 1; 110 = -1

Zweierkomplement

$$[d_n, d_{n-1}, \dots, d_0, \dots, d_{-k}] := \sum_{i=-k, \dots, n-1} d_i 2^i - d_n 2^n$$

Man erhält für a die inverse Zahl, indem man alle Bits komplementiert und an der niedrigwertigsten Stelle eins addiert. Benachbarte Zahlen haben den gleichen Abstand 2^{-k} . Beispiel: $n=2, k=0$ 001 = 1; 111 = -1

Gleitkomma-Zahlen

$a = (-1)^S * M * 2^E$ Vorzeichen S, Exponent E, Mantisse M

Siehe auch folgende Grafik:

Definition (Gleitkommazahl)

Es seien $n, i \in \mathbb{N}$. Eine **Gleitkommazahl** (engl.: *Floating Point Number*) d ist eine n -stellige Bitfolge, die — als Dezimalzahl $[d]$ interpretiert — das folgende Format besitzt:

$$[d] := (-1)^S \cdot M \cdot 2^E$$

Dabei bildet

S das **Vorzeichen**,

M die **Mantisse** und

E den **Exponenten** (inkl. dessen Vorzeichen)

der dargestellten Zahl $[d]$.

Vorzeichen, Exponent und Mantisse bestehen jeweils aus Teil-(Bit-)folgen von d , die zusammengesetzt gerade d ergeben. Das Vorzeichen wird durch ein einzelnes Bit repräsentiert, der Exponent durch eine Folge von i Bits. Die verbleibenden $n - i - 1$ Bits dienen zur Repräsentation der Mantisse, sodass sich folgende Aufteilung ergibt:

d_{n-1}	$d_{n-2} \dots d_{n-(i+1)}$	$d_{n-(i+2)} \dots d_1 d_0$
S	i Bits Exponent E	$n - i - 1$ Bits (vorzeichenlose) Mantisse M

Zusammensetzung der Gleitkommazahlen nach IEEE 754

einfache Genauigkeit:

31	30	...	23	22	...	1	0
S	8 Bits Exponent E			23 Bits vorzeichenlose Mantisse M			

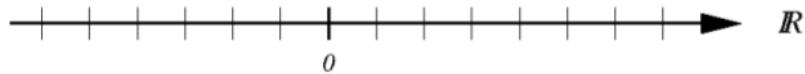
doppelte Genauigkeit:

63	62	...	52	51	...	1	0
S	11 Bits Exponent E			52 Bits vorzeichenlose Mantisse M			

Verteilung der Zahlen

In **Festkommadarstellungen** ist die *Dichte* der darstellbaren Zahlen in jedem Teilintervall von $[-2^n, 2^n - 2^{-k}]$ gleich hoch!

Festkommazahlen:



Gleitkommazahlen:



Bei **Gleitkommadarstellungen** ist die *Anzahl* der darstellbaren Zahlen in jedem Teilintervall $[2^i, 2^{i+1}]$ (bzw. $[-2^{i+1}, -2^i]$) unabhängig von i . Die *Dichte* erhöht sich demnach exponentiell je näher das Teilintervall an der 0 liegt.

Normalisierte Gleitkommazahlen

Die Gleitkommadarstellung, wie sie bis jetzt bei uns eingeführt worden ist, ist **nicht kanonisch**, d. h. es gibt für eine Zahl mehrere Darstellungen, auch wenn man sich entweder auf einfache Genauigkeit oder doppelte Genauigkeit beschränkt, z. B.

$$\begin{aligned} & [0\ 00000001\ 0000000000000000000000000010]_s \\ &= (-1)^0 \cdot 2 \cdot 2^1 \\ &= 4 \\ &= (-1)^0 \cdot 1 \cdot 2^2 \\ &= [0\ 00000010\ 0000000000000000000000000001]_s \end{aligned}$$

Eindeutigkeit erhält man, indem man die Menge der erlaubten Darstellungen einschränkt → **normalisierte Gleitkommazahlen**

Definition einer Normalisierten Gleitkommazahl

Definition (normalisierte Gleitkommazahl)

Es sei $n := 32$ und $d := s e_7 \dots e_0 m_{22} \dots m_0$ eine Bitfolge der Länge n . Zudem sei $e_7 \dots e_0 \notin \{00000000, 11111111\}$. Wird d als **normalisierte Gleitkommazahl nach IEEE 754** interpretiert, so repräsentiert d die Dezimalzahl $[d]_s$ mit

$$[d]_s := (-1)^s \cdot \left(1 + \sum_{i=-1}^{-23} m_{23+i} \cdot 2^i \right) \cdot \left(2^{\left(\sum_{i=0}^7 e_i 2^i\right) - BIAS} \right).$$

Es gilt hierbei $BIAS := 127$

Dezimalwert einer Normalisierten Gleitkommazahl berechnen

Es wird die Definition einer normalisierten Gleitkommazahl verwendet:

Example

Gegeben sei die normalisierte Gleitkommazahl a_1 :

$$a_1 := 1 \textcolor{red}{1000\ 0001\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000}$$

Der dezimale Wert $[a_1]_s$ von a_1 lässt sich gemäß obiger Definition wie folgt ermitteln.

$$\begin{aligned}[a_1]_s &:= (-1)^1 \cdot (1 + (2^{-1} + 2^{-2})) \cdot 2^{2^7 + 2^0 - 127} \\ &= (-1) \cdot (1 + 0,5 + 0,25) \cdot 2^{128+1-127} \\ &= -1,75 \cdot 2^2 \\ &= -7\end{aligned}$$

Normalisieren einer Gleitkommazahl:

- fehlt

Fehlertoleranz

Parity-Code

Eine Bitfolge besteht den Paritätstest wenn die Anzahl der auf 1 gesetzten Bitstellen gerade ist.

Fehlererkennender Code

Ein Code fester Länge heißt k-fehlererkennend, wenn der Empfänger in jedem Fall Entscheiden kann ob bis zu k Bits gekippt sind. $dist(c) \geq k+1$, $dist(c)$ kleinster Abstand zweier Codewörter in c.

Fehlerkorrigierender Code

Ein Code fester Länge heißt k -fehlerkorrigierend, wenn $\text{dist}(c) \geq 2k+1$

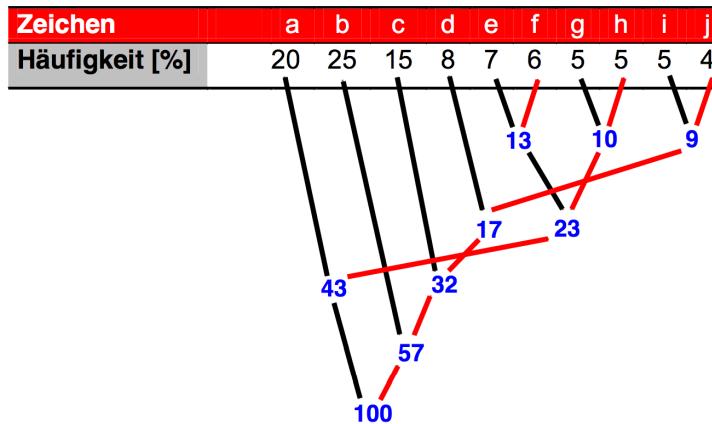
Hamming-Abstand

Hamming-Abstand $\text{dist}(v, w)$ zweier n -Bitfolgen v und w ist die Anzahl der Stellen, an denen v und w sich unterscheiden. Beispiel: $\text{dist}(1100, 0011) = 2$

Hamming-Code

Paritäts-Bits sind alle 2^j Bits. Alle anderen Bits sind Datenbits. Die Paritätsbits prüfen die Parität der j nachfolgenden Datenbits. Hamming-Codes haben die Länge $q \geq \text{abrunden}(\log_2 m)$, das bedeutet das $\text{abrunden}(\log_2 q) + 1$ Paritätsbits hinzugefügt werden. Das Paritätbit an der 2^i Stelle überprüft alle Bitstellen, die in ihrer Binärdarstellung an der $j+1$ Stelle eine 1 haben. Beispiel: 2^2 prüft alle Bitstellen, die an der 3. Stelle eine 1 haben, also 5 (101), 6 (110), 7 (111), 12 (1100), ...

Huffman-Code



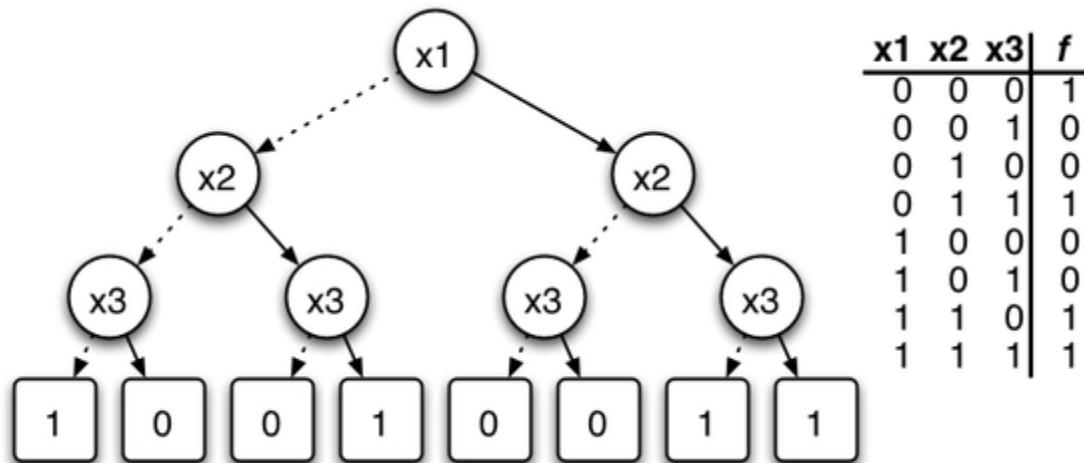
Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden. Makiere linke Kanten mit 0, rechte mit 1. (Präfix-Code, da kein i, j aus $\{1, 2, \dots, m\}$ existiert so dass $c(a_i)$ Präfix von $c(a_j)$)

Binary Decision Diagram (BDD)

Ordered Binary Decision Diagram (OBDD)

BDDs sind Graphen, welche eine Boolesche Funktion darzustellen.

Betrachten wir dazu folgende Grafik:



Auf der rechten Seite sehen wir eine Wahrheitstabelle (auch eine Möglichkeit eine Boolesche Funktion darzustellen) und links davon das dazugehörige BDD.

Anzumerken bleibt das die gestrichelte Kanten das Gewicht "0" haben und die durchgezogenen Kanten das Gewicht "1". (Das ist etwas doof dargestellt.)

Was wir hier genau sehen ist ein Unreduziertes Binäres Entscheidungsdiagramm.

Wenn wir genau hinschauen erkennen wir, dass egal welchen Weg wir durch den Graphen von der "Wurzel" aus nehmen: Wir Durchlaufen immer alle Variablen und zwar in der exakt selben Reihenfolge ($x_1 \rightarrow x_2 \rightarrow x_3$). Und diese Eigenschaft macht dieses BDD zu einem sogenannten "Ordered Binary Decision Diagram" (OBDD).

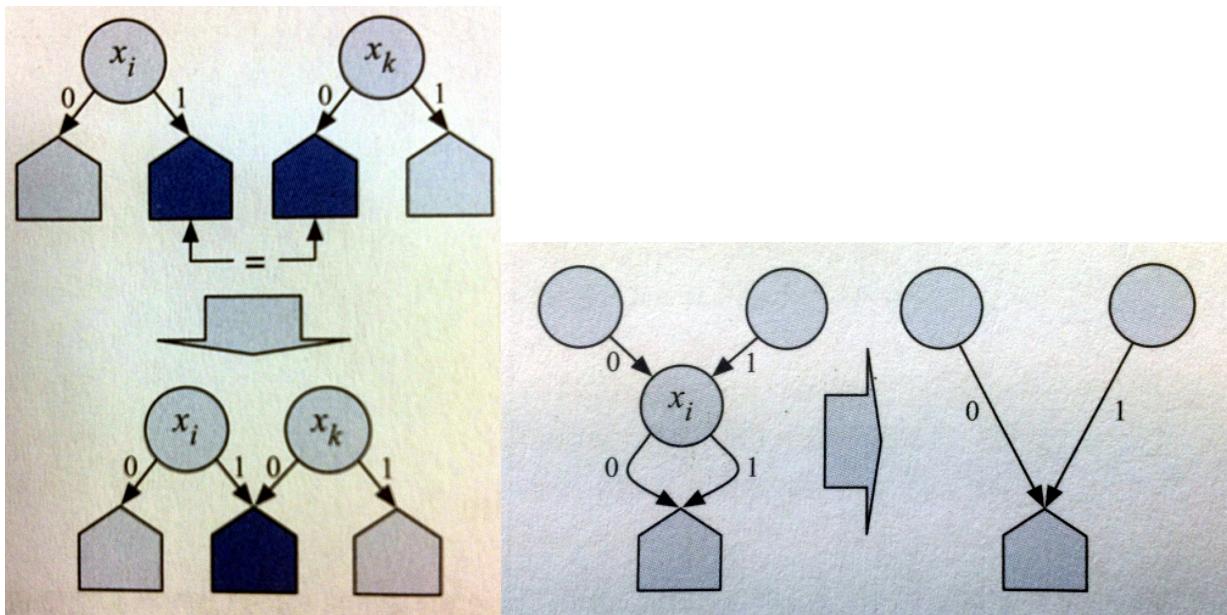
Somit ist auch jedes Unreduziertes Binäres Entscheidungsdiagramm automatisch auch ein Ordered Binary Decision Diagram.

Reduced Ordered Binary Decision Diagram (ROBDD)

Wie bereits angemerkt kann man OBDDs vereinfachen.

Dazu gibt es zwei Techniken:

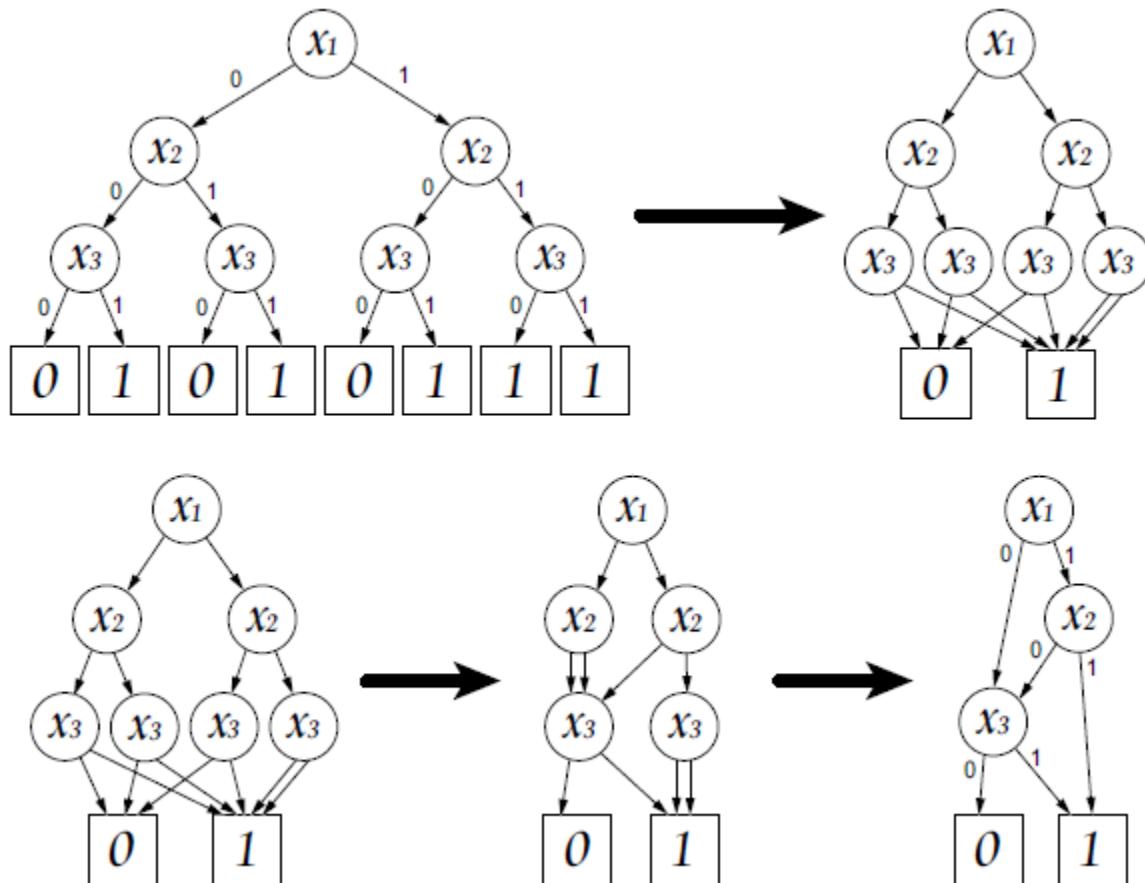
1. Verschmelzen gleicher Teilgraphen
2. Löschen von Knoten mit zwei gleichen Nachfolgern



Hier sehen wir auf der linken Seite die Verschmelzung...

...und auf der rechten Seite die Löschung eines Knotens.

Hier ein Beispiel:



Die beiden Regeln werden so oft angewandt wie Möglich. Lässt sich keine der Regeln mehr anwenden, so sprechen wir von einem Reduced Ordered Binary Decision Diagram (ROBDD).

Randal Bryant konnte zeigen, dass Reduced Ordered Binary Decision Diagram (ROBDD) jede boolesche Funktion auf eindeutige Weise darstellen.

Damit sind Reduced Ordered Binary Decision Diagram (ROBDD) eine weitere Normalenformdarstellung boolescher Funktionen.

ON-Menge, Minterm, Maxterm, Literal, Monom und Polynom

ON-, OFF-Menge

Für $f \in B_n$:

- $ON(f) := \{x \in B^n; f(x) = 1\}$ bezeichnet die Erfüllbarkeitsmenge von f .
- $OFF(f) := \{x \in B^n; f(x) = 0\}$ bezeichnet die Nichterfüllbarkeitsmenge von f .

Maxterm

Definition:

$x_1 \vee \dots \vee x_n$ mit $x_i \in \{\bar{x}_i, x_i\}$

Beispiel:

$x_1 \vee x_2 \vee \bar{x}_3$ ist ein Maxterm

Literal

x_i und \bar{x}_i sind Literale

Monom

Ein Monom ist eine Konjunktion (Verknüpfung durch \wedge) von Literalen, in der kein Literal doppelt auftritt und die außerdem für kein $i \in N$ sowohl x_i^0 als auch x_i^1 enthält.

Minterm

Ein Monom m heißt vollständig, wenn jede Variable entweder als positives oder als negatives Literal in m vorkommt. Ein solches Monom wird auch als Minterm bezeichnet.

Definition:

$x_1 \wedge \dots \wedge x_n$ mit $x_i \in \{\bar{x}_i, x_i\}$

Beispiel:

$x_1 \wedge x_2 \wedge \bar{x}_3$ ist ein Minterm

Polynom

Ein Polynom ist ein Boolescher Ausdruck, der aus einer Disjunktion (Verknüpfung durch \vee) paarweise verschiedener Monome besteht. Sind alle Monome vollständig, so handelt sich um ein vollständiges Polynom.

Implikant und Primimplikant

- Sei $f \in B_n$. Ein Monom heißt Implikant von f , falls $\psi(m) \leq f$.
- Sei $f \in B_n$. Ein Implikant m von f heißt Primimplikant (PI) von f , falls kein echtes Teilmonom des Implikanten m ebenfalls Implikant von f ist.

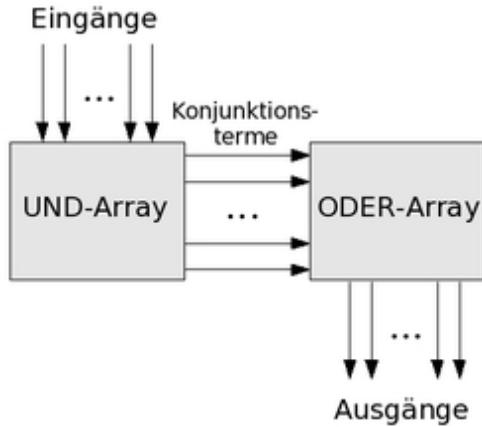
Beispiel: Sei $f = x_1x_2 + x_2$ eine Boolesche Funktion über zwei Variablen. Dann sind x_1x_2 und x_2 Implikanten. Aber nur x_2 ist ein Primimplikant, da x_1x_2 das echte Teilmonom x_2 besitzt, das auch ein Implikant von f ist.

Alle Monome eines Polynoms p von f sind Implikanten von f .

Minimierung

Direkte Umsetzung in eine Schaltung ist meist zu kostspielig, da deren Größe mit der Zahl der Eingangsvariablen oft exponentiell wächst. Deshalb benötigen wir Minimierung.

Programmable Logic Array (PLA)



Wenn man eine Funktion oder ein Polynom (z.B. $f(x_1, x_2, x_3) = x_1 \cdot x_2 + x_1 \cdot x_3 + /x_2 \cdot x_3$) mithilfe eines PLA realisieren möchte, dann kann man in jeder "Zeile" des PLAs ein Monom (z.B. $x_1 \cdot x_2$) berechnen lassen, wie oben zu sehen geschieht das im "UND-Array". Mithilfe des ODER-Arrays werden dann die einzelnen Zeilen, also die Monome zur ganzen Funktion zusammengesetzt.

Die Anzahl der Zeilen und der Spalten des PLAs, d.h. die Fläche wird folgendermaßen berechnet:

Spalten: $(m+2^n)$, wobei
 m = Anzahl an verschiedenen Polynomen auf dem PLA
 n = Anzahl an benutzten Literaten insgesamt

Zeilen: Anzahl an Monomen, die für die Realisierung der Polynome gebraucht werden

Kosten von Monomen

Sei $q = q_1 \cdot \dots \cdot q_r$ ein Monom, dann sind die Kosten $|q|$ von q gleich der Anzahl der zur Realisierung von q benötigten Schalter im PLA, also $|q| = r$.

Kosten von Polynomen

Seien p_1, \dots, p_m Polynome, dann bezeichne $M(p_1, \dots, p_m)$ die Menge der in diesen Polynomen verwendeten Monome.

Die primären Kosten

Die primären Kosten $cost_1(p_1, \dots, p_m)$ einer Menge $\{p_1, \dots, p_m\}$ von Polynomen sind gleich der Anzahl der benötigten Zeilen im PLA, um p_1, \dots, p_m zu realisieren, d.h.

$$cost_1(p_1, \dots, p_m) = |M(p_1, \dots, p_m)|$$

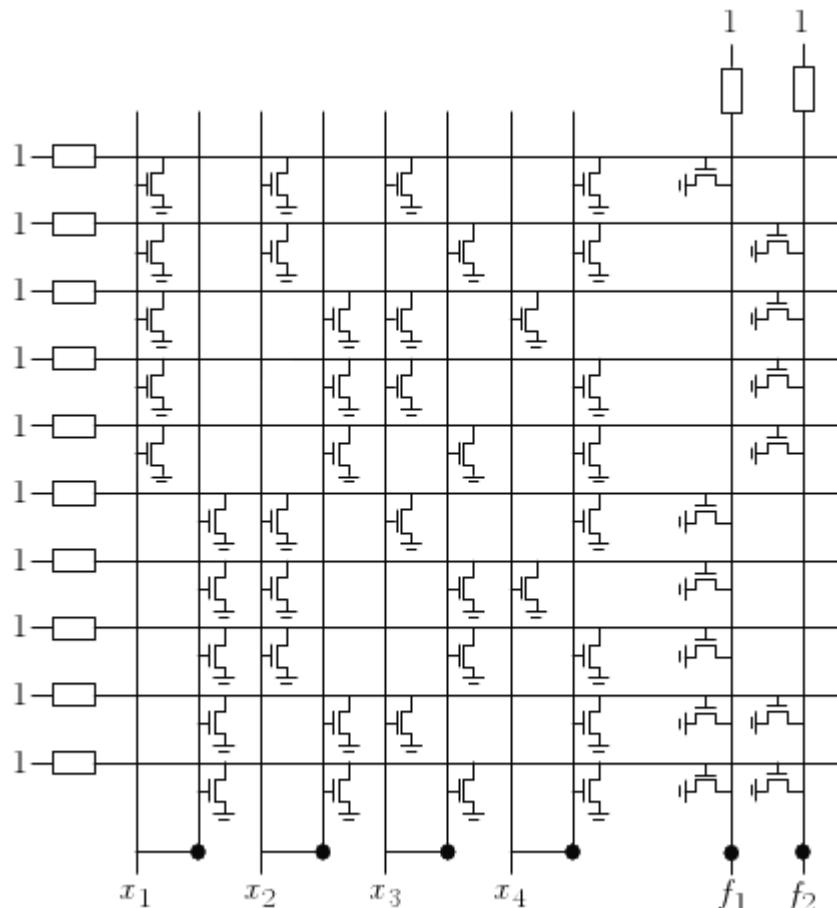
Die sekundären Kosten

Die sekundären Kosten $\text{cost}_2(p_1, \dots, p_m)$ einer Menge $\{p_1, \dots, p_m\}$ von Polynomen sind gleich der Anzahl der benötigten Transistoren im PLA, d.h.

$$\text{cost}_2(p_1, \dots, p_m) = \sum_{q \in M(p_1, \dots, p_m)} |q| + \sum_{i=1, \dots, m} |M(p_i)|$$

In der ersten Summe werden die Anzahl der Schalter im AND-Feld und in der zweiten die Kosten der Schalter im OR-Feld betrachtet.

PLA-Beispiel:



Hier haben wir $n=4$, da wir vier Literale x haben. Es werden zwei Funktionen bzw. Polynome realisiert, also $m=2$. Es werden 10 verschiedene Monome benutzt, deshalb haben wir 10 Zeilen. Das obige PLA besitzt also die Maße $(m \cdot 2n) \times 10 = 10 \times 10$.

$$p_1 = \bar{x}_1 \bar{x}_2 \bar{x}_3 x_4 \vee x_1 \bar{x}_2 \bar{x}_3 x_4 \vee x_1 \bar{x}_2 x_3 \bar{x}_4 \vee x_1 \bar{x}_2 x_3 x_4 \vee x_1 x_2 \bar{x}_3 x_4 \vee x_1 x_2 x_3 x_4$$

$$p_2 = \bar{x}_1 \bar{x}_2 x_3 x_4 \vee \bar{x}_1 x_2 \bar{x}_3 \bar{x}_4 \vee \bar{x}_1 x_2 \bar{x}_3 x_4 \vee \bar{x}_1 x_2 x_3 x_4 \vee x_1 x_2 \bar{x}_3 x_4 \vee x_1 x_2 x_3 x_4$$

$$\text{cost}_1(p_1, p_2) = 10$$

$$\text{cost}_2(p_1, p_2) = 40 + 12 = 52$$

$$\text{cost}(p_1, p_2) = (10, 52)$$

Problem der 2-stufigen Logikminimierung

Bei PLAs kann es zu der Situation kommen, dass viele Zeilen und Spalten "dünn besetzt" sind, d.h. kaum Schalter verwendet werden, oder sogar ganz frei bleiben. Die resultierenden Freiräume können genutzt werden, indem Leitungen verlegt oder gestrichen werden. Dieser Vorgang wird als Faltung bezeichnet.

Wie wählt man unter der Menge von verschiedenen Polynomen die die Boolesche Funktion beschreiben die beste aus? Also welche Möglichkeiten gibt es ein PLA zu "falten".

Gesucht ist eine Menge von m Polynomen $\{g_1, \dots, g_m\}$ mit den Eigenschaften $\psi(g_i) = f_i$ für alle i, wobei gelten muss, dass die primären und sekundären Kosten für alle Polynome g_1, \dots, g_m minimal sind.

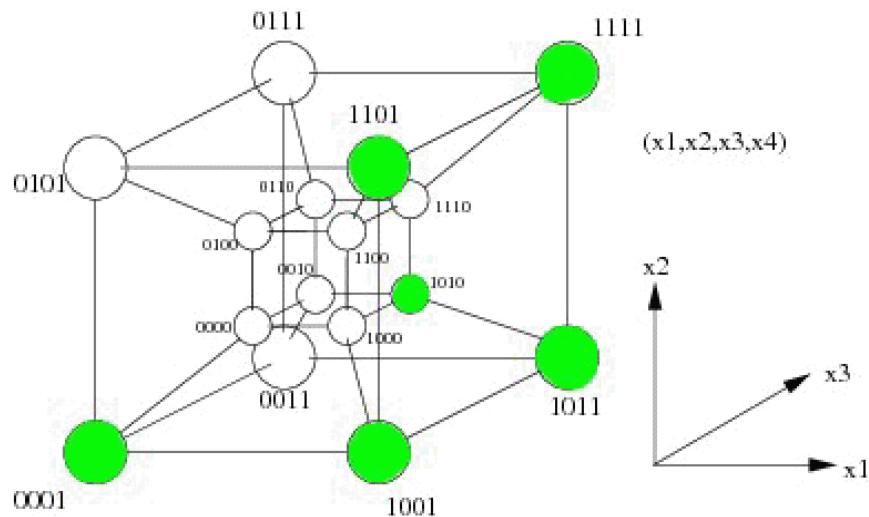
Mehrdimensionale Würfel

Eine Boolesche Funktion kann z.B. durch einen mehrdimensionalen Würfel veranschaulicht werden (macht aber nur Sinn bei Funktionen mit höchstens vier Variablen).

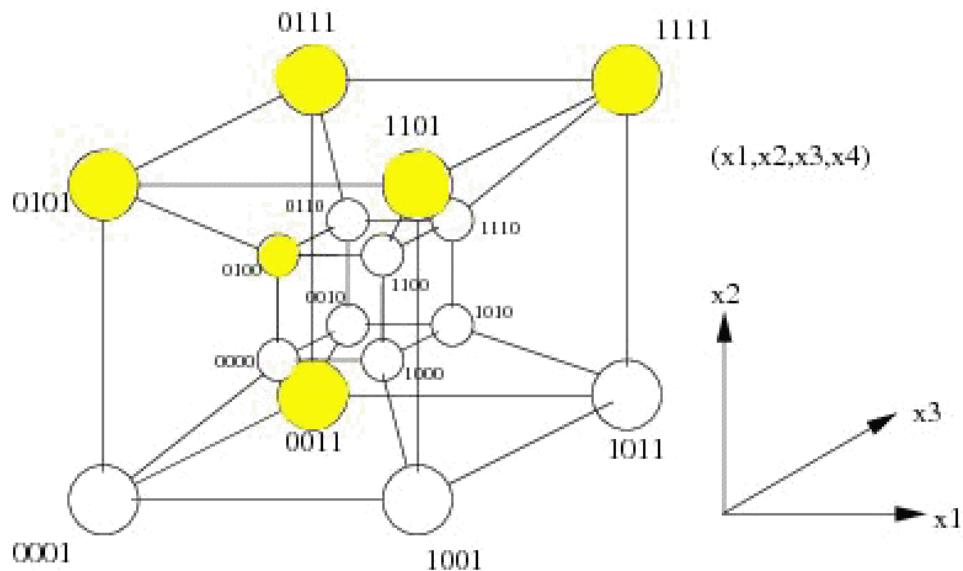
Wichtig ist, dass benachbarte Ecken, die durch eine Kante verbunden sind, sich genau in einer Stelle unterscheiden.

Betrachten wir nun das PLA-Beispiel von oben, dann können wir die ON-Menge von p_1 und p_2 in folgende Würfel einzeichnen. Es handelt sich hier um zwei verknüpfte 3-dimensionale Würfel, da die Eingabemenge = $2^4 = 16$ ist und wir somit 16 Ecken haben. Die Ecken entsprechen den Elementen in B^n .

Der Würfel für p_1 :

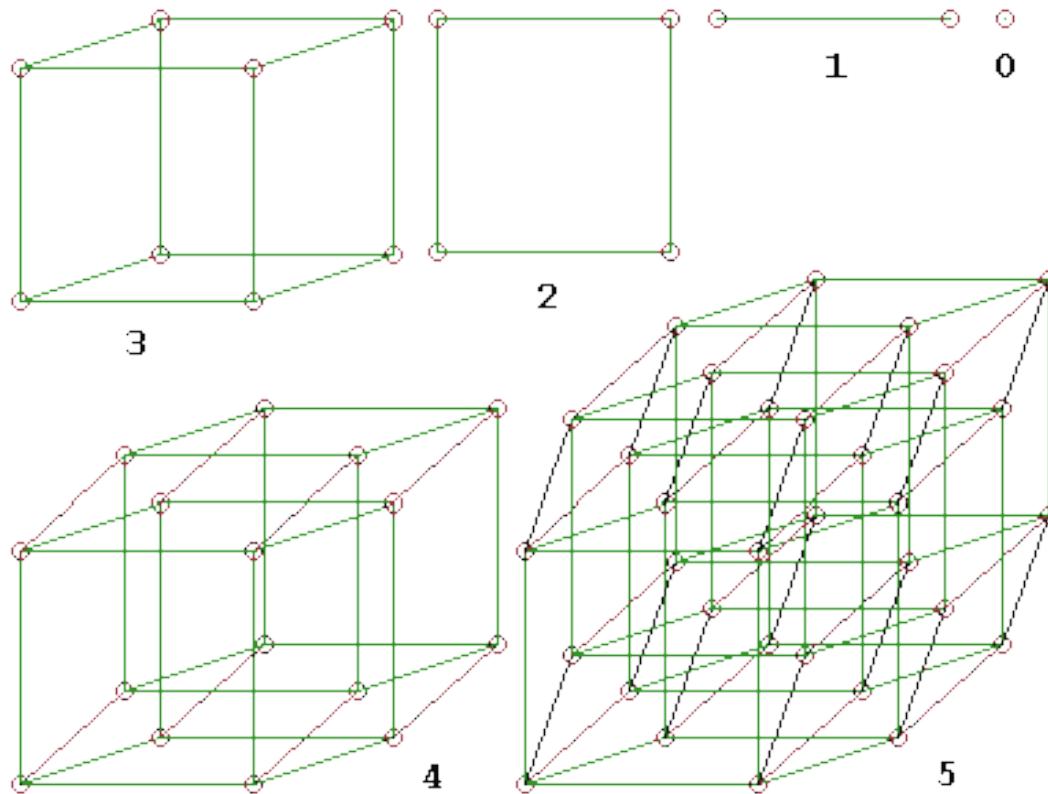


Der Würfel für p_2 :



Konstruktion einer Hypercubes

Ein n -dimensionaler Würfel kann wie folgt konstruiert werden: Es werden zwei Instanzen des $(n-1)$ -dimensionalen Würfels benötigt. Alle Ecken mit der gleichen Bezeichnung werden verbunden. Anschließend wird die Eckenbezeichnung um eine Stelle an vorderster Position erweitert - in einer Instanz mit 0, in der anderen mit 1.



Berechnung von Minimalpolynomen (Quine/McCluskey)

Primimplikantensatz von Quine: Sei $f \neq 0$. Dann besteht ein Minimalpolynom p für eine Boolesche Funktion f ausschließlich aus Primimplikanten von f .

Der Algorithmus für die Bestimmung der Primimplikanten baut auf folgenden beiden Lemmata auf:

- Ist m ein Implikant von f , so auch $m \cdot x$ und $m \cdot \bar{x}$ für jede Variable x , die in m weder als positives noch als negatives Literal vorkommt.
- Sind $m \cdot x$ und $m \cdot \bar{x}$ Implikanten von f , so auch m .

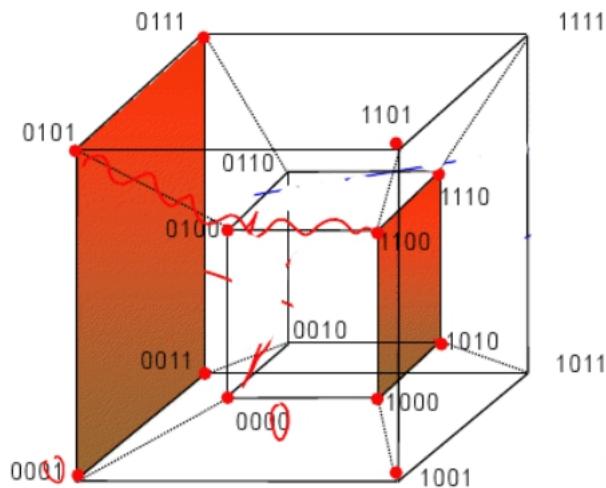
Berechnung von Minimalpolynomen:

1. Berechnung aller Primimplikanten, da nach obigem Satz ein Minimalpolynom nur aus solchen besteht. => Anwendung des Verfahrens von Quine/McCluskey.
 - Eingabe: Minterme
 - Ausgabe: Primimplikanten
2. Auswahl der Primimplikanten: Es müssen für das Minimalpolynom nicht alle gefundenen Primimplikanten verwendet werden. Die Auswahl wird als Überdeckungsproblem formuliert. Es gibt hier auch Verfahren dies zu lösen.

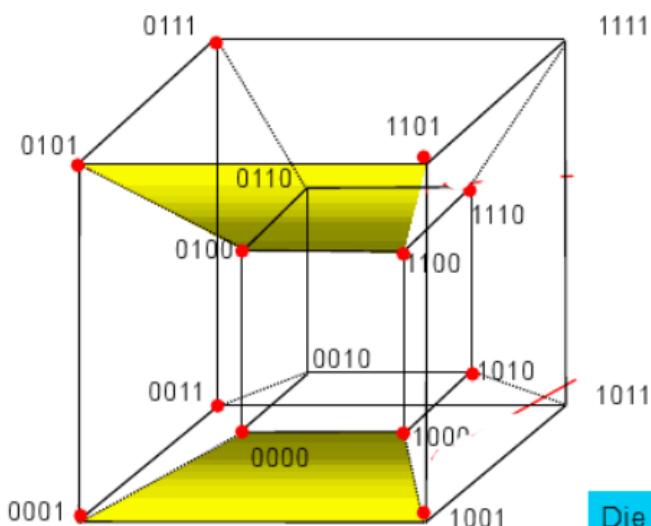
Quine/McCluskey

1. Schritt: Initialisierung
 - a. $L_0 = \{m; m \text{ Minterm von } f\}$
2. In Iteration i werden die Mengen berechnet:
 - a. $L_i = Q(L_{i-1})$
 - b. $P_i = \{m; m \in L_{i-1} \text{ und } m \text{ hat keinen Partner in } L_{i-1}\}$
3. Das Verfahren bricht ab, wenn $L_i = \emptyset$ oder $i = n$ gilt.

Quine McCluskey Visualisierung auf dem Cube



Die markierten Implikanten-Flächen sind nicht Rand eines 3-dim. Implikanten. Sie sind also prim! $\rightarrow \text{Prim}(f) = \{x_1' x_4, x_1 x_4'\}$



Die markierten Implikanten-Flächen sind Rand eines 3-dimensionalen Implikanten. Sie sind also nicht prim! $\rightarrow \text{Prim}(f) = \{x_1' x_4, x_1 x_4'\}$

Matrix-Überdeckungsproblem

Matrix-Überdeckungsproblem

- Für eine Expedition wird ein Fahrer, ein Messtechniker und ein Kameramann benötigt. Es stehen fünf Kandidaten mit unterschiedlichen Fähigkeiten und Gehaltsvorstellungen zur Auswahl. Finde das kostengünstigste Team.

Kandidat	Fahrer?	Messtechniker?	Kameramann?	Gehalt
Alice	Ja	Nein	Ja	4000
Dilbert	Ja	Ja	Nein	2000
Dogbert	Ja	Ja	Ja	5000
Ted	Nein	Nein	Ja	1000
Wally	Nein	Ja	Ja	1500

- {Alice, Dilbert}, {Dogbert}, {Dilbert, Wally} sind legale, jedoch nicht kostenminimale Lösungen.
- {Dilbert, Ted} ist die kostenminimale Lösung.

BB - TI

Kap. 3.3

25

=> Mehrere Kombinationsmöglichkeiten zur Lösung eines Problems.

=> Aber welche ist die Beste.

Primimplikantentafel

Zuerst bauen wir die Primimplikantentafel einer funktion f als boolesche Matrix PIT(f) auf:

- Die Zeilen entsprechen eindeutig den Primimplikanten von f.
- Die Spalten entsprechen eindeutig den Mintermen von f. Bzw. etwas genauer Formuliert.
0 = erster Minterm, 1 = zweiter Minterm 2 = ...
und zwar aus der On-Menge (nicht Off-Menge)
- Die 1en zwischen Zeilen und Spalten werden wie folgt gesetzt: Immer wenn der Primimplikant auf den Minterm zutrifft wird eine 1 geschrieben.

Beispiel:

	0	0	0	0									
	0	0	0	1									
	0	1	0	0									
	1	0	0	0									
	0	0	1	1									
	0	1	0	1									
	1	0	0	1									
	1	0	1	0									
	1	1	0	0									
	1	1	0	1									
	1	1	1	0									
	1	1	1	0									
	1	1	1	0									
	1	1	1	0									

$$\text{Prim}(f) = \{x_1'x_4, x_1x_4', x_3'\}$$

Primimplikantentafel PIT(f):

	0	1	3	4	5	7	8	9	10	12	13	14	
$x_1'x_4$	1	1		1	1								<u>1</u> 1 0 0
x_1x_4'					1		1	1	1			1	0 1 1 1
x_3'	1	1		1	1	1	1	1	1	1	1	1	1 1 0 1 1 1 1 0

Gesucht:

Eine kostenminimale Teilmenge M von $\text{Prim}(f)$, so dass jede Spalte von PIT(f) überdeckt ist.

Primimplikantentafel PIT(f):

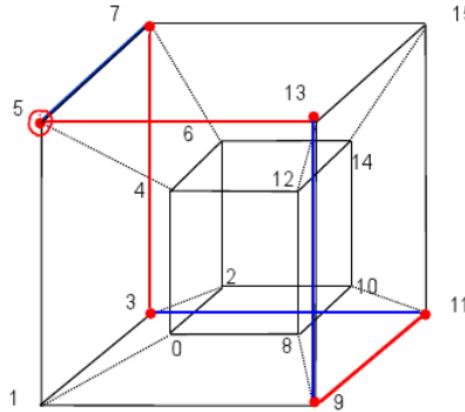
	0	1	3	4	5	7	8	9	10	12	13	14	
$x_1'x_4$	1	1		1	1								
x_1x_4'					1		1	1	1			1	
x_3'	1	1		1	1	1	1	1	1	1	1	1	

→ Alle Primimplikanten sind **wesentlich!**

In unserem Beispiel sind alle Primimplikanten wesentlich, da es bei jedem Primimplikanten eine Spalte gibt in der nur er zutrifft (mit einem Punkt in der Grafik dargestellt).

Hier ein Beispiel wo keiner der Primimplikanten wesentlich ist (also das genaue Gegenteil):

$$\text{Prim}(f) = \{\{7,5\}, \{5,13\}, \{13,9\}, \{9,11\}, \{11,3\}, \{3,7\}\}$$



Primimplikantentafel PIT(f):

	3	5	7	9	11	13
{7,5}	1	1				
{5,13}		1				1
{13,9}			1		1	
{9,11}				1	1	
{11,3}	1				1	
{3,7}	1	1				

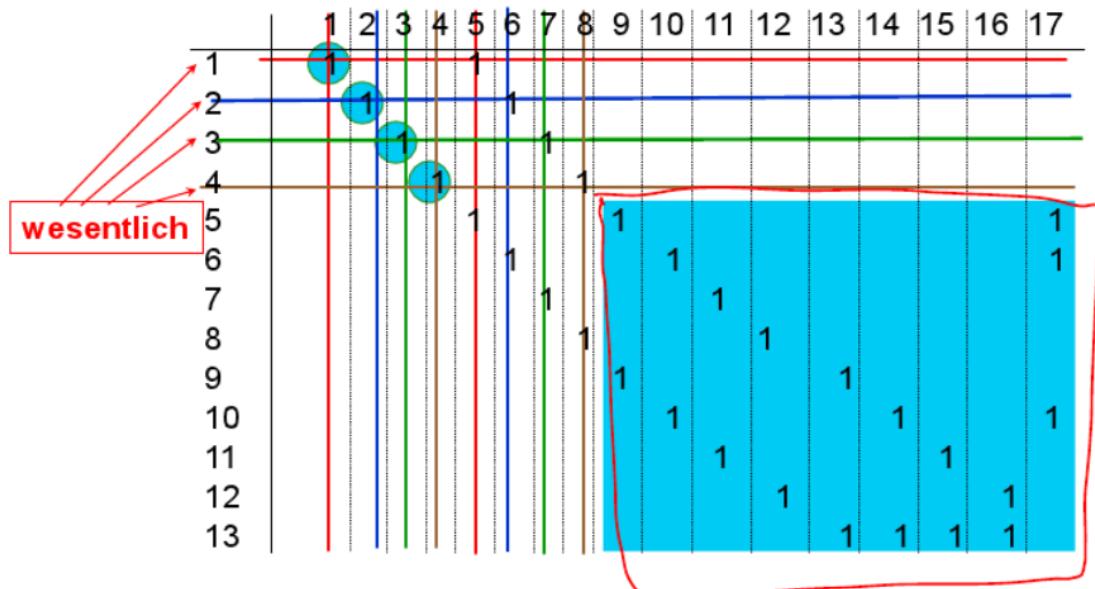
Kein Primimplikant ist wesentlich!

Wir halten fest:

Ein Primimplikant m von f heißt *wesentlich*, wenn es mindestens einen Minterm von f gibt, der nur von diesem Primimplikanten überdeckt wird.

1. Reduktionsregel:

Entferne aus der Primimplikantentafel PIT(f) alle wesentlichen Primimplikanten und alle Minterme, die von diesen überdeckt werden. Neues Beispiel:



2. Reduktionsregel:

Entferne aus der Primimplikantentafel PIT(f) alle Minterme, die einen anderen Minterm in PIT(f) dominieren.

	9	10	11	12	13	14	15	16	17
5	1								1
6		1							1
7			1						
8				1					
9	1				1				
10		1				1			1
11			1				1		
12				1				1	
13					1				1

Spalte 17 dominiert Spalte 10 => Spalte 17 fliegt heraus

3. Reduktionsregel:

Reduktionsregel: Entferne aus der Primimplikantentafel PIT(f) alle Primimplikanten, die durch einen anderen, nicht teureren Primimplikanten dominiert werden.

	9	10	11	12	13	14	15	16
5	1							
6		1						
7			1					
8				1				
9				1				
10		1			1			
11			1			1		
12				1			1	
13					1			1

werden dominiert

So sieht jetzt unsere Primimplikantentafel aus:

	9	10	11	12	13	14	15	16
9	1				1			
10		1			1			
11			1				1	
12				1				1
13					1	1	1	1

Wie wir sehen sind die Primimplikanten 9, 10, 11 und 12 wieder wesentlich. Ergo können wir die 1. Reduktionsregel erneut anwenden und daraufhin ist unsere Matrix leer.

=> Das gefundene Minimalpolynom ist $1+2+3+4+9+10+11+12$.

Wir halten fest:

Eine Primimplikantentafel PIT(f) heißt reduziert, wenn keine der drei Reduktionsregeln anwendbar ist.

In unseren Beispiel ist die resuzierte Matrix leer.

Ist eine reduzierte Matrix nicht leer, so spricht man von einem zyklischen Überdeckungsproblem.

Diese kann man z.B. mit Petrick's Methode oder mit der „Greedy-Heuristik“ lösen.

Petrick's Methode

Wird verwendet zum Lösen von zyklischen Überdeckungsproblemen.

Verfahren

- Übersetze die PIT in eine Produktsumme, d.h. in ein (OR,AND)-Polynom, das alle Möglichkeiten der Überdeckung enthält.
- Multipliziere die Produktsumme aus, so daß ein (AND-OR)-Polynom entsteht.
- Die gesuchte minimale Überdeckung ist gegeben durch das **Monom**, das einer PI-Auswahl mit minimalen Kosten entspricht.

	1	2	3	4
1		1	1	
2			1	1
3		1	1	
4			1	1
5		1		1
6			1	1

notwendig,
um Spalte 4
zu überdecken

wird übersetzt in

$$(1+3+5)(1+4+6)(2+3+6)(2+4+5)$$

$$= (1+14+16+13+34+36+15+45+56)*$$

$$(2+24+25+23+34+35+26+46+56)$$

$$= 12+124+125+123+134+\dots+34+\dots+56$$

bei gleichen Kosten für alle PIs sind
12, 34 und 56 minimal

„Greedy-Heuristik“ zur Lösung von Überdeckungsproblemen

„Greedy-Heuristik“ zur Lösung von Überdeckungsproblemen

- Wende alle möglichen Reduktionsregeln an.
 - Ist die Matrix A leer, ist man fertig.
 - Sonst wähle die Zeile *i*, die die meisten Spalten überdeckt. Lösche diese Zeile und alle von ihr überdeckten Spalten und gehe zu 1.
- Dieser Algorithmus liefert nicht immer die optimale Lösung!
 - Hinweis:** Bei der Ausgangs-Matrix aus unserem Beispiel überdeckt Zeile 13 die meisten Spalten. Diese ist nicht Teil der gefundenen Lösung!

Normalformendarstellungen

Stellt eine Boolesche Funktion eindeutig dar. Eindeutig bedeutet, dass jede von ihnen auf exakt eine einzige Art repräsentiert wird.

Boolesche Ausdrücke beispielsweise sind keine Normalformendarstellung, da sich jede Boolesche Funktion durch unendlich viele Ausdrücke darstellen lässt

Beispiel:

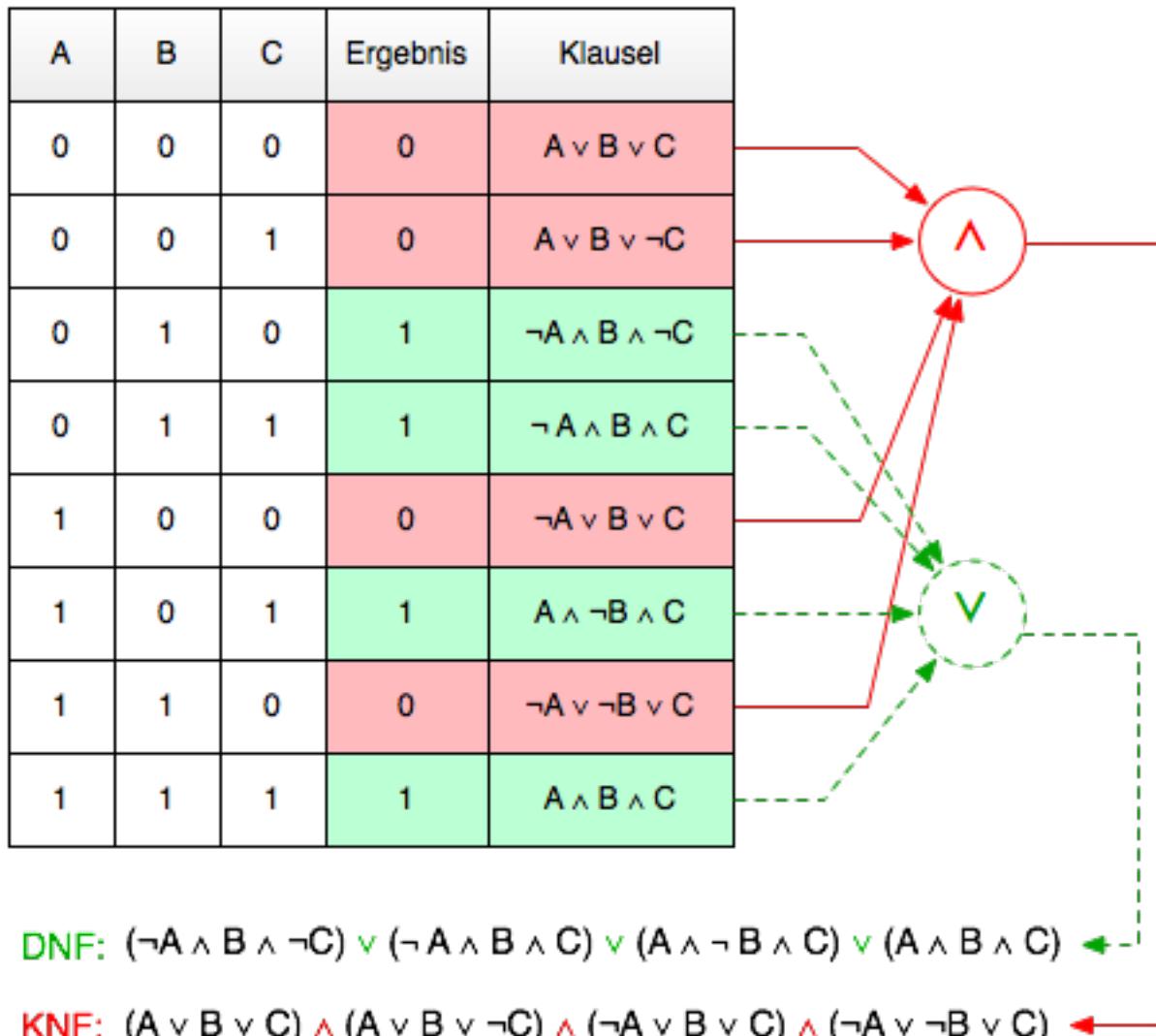
Wahrheitstafeldarstellung

x1	x2	y
0	0	1
0	1	1
1	0	1
1	1	0

Auch eine Wahrheitstafel ist eine Normalform da zwei boolesche Funktionen genau dann gleich sind, wenn ihre Wahrheitstafeln identisch sind.

Disjunktive und konjunktive Normalform

Siehe folgende Grafik:



(statt A, B und C bei uns x_1, x_2, \dots
statt Ergebnis bei uns y)

DNF = Disjunktive Normalform
KNF = Konjunktive Normalform

Wir halten fest:

Bei der disjunktiven Normalform: $y == 1 \Rightarrow x_i == 1$, innen \wedge , außen \vee
konjunktiven Normalform $y == 0 \Rightarrow x_i == 0$, innen \vee , außen \wedge

Reduced Ordered Binary Decision Diagram (ROBDD)

(und zwar nur ROBDD, da diese sowohl geordnet als auch reduziert sind)
(siehe weitere Erklärung unter BDD)

Schaltungssynthese

Umschreibt die systematische Umwandlung einer booleschen Funktion in eine Hardwareschaltung.

Logikgatter

amerikanisch	deutsch (alt)	deutsch (neu)	Bezeichnung:
	Driver	1	Treiber
	NOT	1	Inverter Negation
	AND	&	UND -Gatter Konjunktion
	NAND	&	Nicht UND -Gatter
	OR	≥ 1	ODER -Gatter Disjunktion
	NOR	≥ 1	Nicht ODER -Gatter
	XOR	+	Exclusives ODER Antivalenz
	NXOR	+	Äquivalenz

Deutlich bessere Auflistung und Beschreibung: <http://de.wikipedia.org/wiki/Logikgatter>

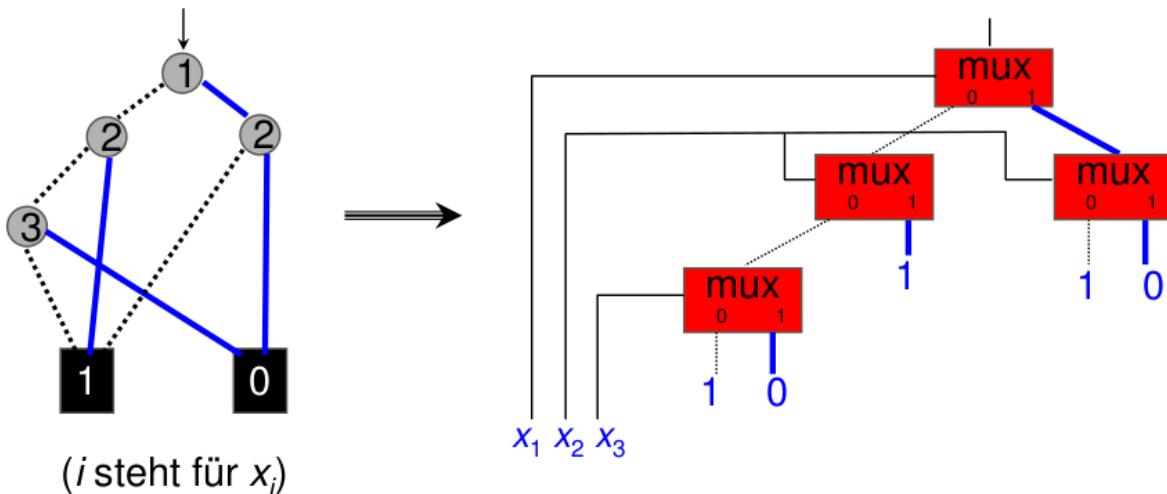
AND			OR			XOR			NOT	
x1	x0	y	x1	x0	y	x1	x0	y	x1	y
0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	1	1	0	1	1	1	0
1	0	0	1	0	1	1	0	1	1	0
1	1	1	1	1	1	1	1	0		

NAND			NOR			XNOR				
x1	x0	y	x1	x0	y	x1	x0	y	x1	y
0	0	1	0	0	1	0	1	0	0	1
0	1	1	0	1	0	1	0	0	1	0
1	0	1	1	0	0	1	0	0	0	0
1	1	0	1	1	0	1	1	1		

Zweistufige Schaltungssynthese

Auf Basis der konjunktiven oder disjunktiven Darstellungsform lassen sich Schaltungen ganz einfach mit UND, ODER und Invertern bauen.
(Wirklich ganz banal. Man setzt einfach die Formel um.)

BDD-basierte Schaltungssynthese



... beschreibt die Boolesche Funktion $x_1'x_2'x_3' + x_1'x_2 + x_1x_2'$

=> Jeder Knoten wird zu einem Multiplexer

Wie man mit dem NAND-Gatter alle anderen Logikgatter baut

http://en.wikipedia.org/wiki/NAND_logic

Formale Definition eines Schaltkreises

$SK_1 := (X_3, G, typ, IN, Y)$, wobei

$$\begin{aligned} X_3 &= \{x_1, x_2, x_3\} \\ Y &= \{y_1\} \\ G &= (V, E) \\ V &= X_3 \cup \{v_1, \dots, v_7\} \cup Y \\ E &= \{(v_4, v_2), (v_6, v_2), (v_1, v_4), (x_2, v_5), (x_1, v_3), (x_3, v_5), \\ &\quad (v_3, v_7), (v_7, v_1), (v_2, y_1), (x_3, v_6), (x_1, v_7), (v_5, v_4)\} \\ typ &= \{(v_i \mapsto \text{NOT}) \mid i \in \{1, 3, 6\}\} \cup \{(v_i \mapsto \text{AND}) \mid i \in \{2, 7\}\} \cup \{(v_i \mapsto \text{OR}) \mid i \in \{4, 5\}\} \\ IN &= \{(v_1 \mapsto (v_7)), (v_2 \mapsto (v_4 v_6)), (v_3 \mapsto (x_1)), (v_4 \mapsto (v_1 v_5)), \\ &\quad (v_5 \mapsto (x_2 x_3)), (v_6 \mapsto (x_3)), (v_7 \mapsto (x_1 v_3))\} \end{aligned}$$

G besteht aus (V,E)

V vereinigt X_3 , die v und Y

E ist die Verknüpfung aller Elemente von V

typ schreibt sich etwas speziell

IN beschreibt für jedes v (und nur für die v) welches deren Vorfahren sind. (Beispiel: Die Vorfahren von v5 sind hier x2 und x3).

Complementary Metal Oxide Semiconductor (CMOS)

Kapitel 3:

Transistoren (in der Metalloxidhalbleiter/MOS-Technologie)

Realisierung von Gattern in CMOS-Technologie

CMOS-Inverter mit 1 am Gate

CMOS-Inverter mit 0 am Gate

CMOS-NAND-Gatter

Weitere CMOS-Gatter

Bewerten von Schaltungen

Schnelle Schaltungen benötigen gewöhnlich viel Fläche, während kompakte Schaltungen selten zu den schnellsten Implementierungen gehören.

Stufigkeit einer Schaltung

= max. Pfadlänge (=Anzahl durchlaufener Logikgatter) (Inverter werden gewöhnlich nicht mitgezählt) von den Eingängen zu den Ausgängen.

$C_S = \text{Schaltungstiefe}$

Schaltungsgröße

= Anzahl an Logikgattern insgesamt

$C_A = \text{Gatterzahl}$

Flächenmetrik

$C'_A = \sum_{\text{Gatter}} \text{Eing}[?][?] \text{nge von Gatter}$

Also wenn wir beispielsweise eine Schaltung mit 3 Gattern haben wovon zwei 2 Eingänge haben und eines 3, dann ist $C'_A = 2 + 2 + 3 = 7$

Laufzeitmetrik

$C'_S = (100 * C_S) + C'_A$

Fortgeschrittene BDDs mit IfThenElse (ITE) und mehr

Operationen zwischen BDDs

Berechnung von ITE auf BDDs –

Prinzip

ITE (If then else) auf reduzierten BDDs

ITE-Berechnung: Beispiel

Zur Komplexität von ITE

ITE-Algorithmus

Größe von BDDs

Schaltkreis direkt aus BDD

Schaltungen

Addierer

Addieren nach der Schulmethode: Carry-Ripple-

Addierer.

Effizienteres Addieren: Conditional-Sum-Addierer.

Addition von Zweierkomplement-Zahlen.

Subtrahierer.

Kosten von Schaltkreisen

Um unterschiedliche Schaltkreise, die eine Funktion (z.B. Addierer) implementieren, miteinander zu vergleichen, benötigt man einen Kostenmaß.

Definition: Die Kosten $C(\text{SK})$ eines Schaltkreises SK sind durch die Anzahl seiner Gatter gegeben.

Deutet auf die Fläche und den Energieverbrauch des

resultierenden Hardware-Blocks hin.

Definition: Die Tiefe depth(SK)

Beispiel: Kosten und Tiefe

$x_1 x_2 x_3 x_4 x_5$

$x_6 x_7 x_8$

$C(SK) = 8$

$Depth(SK) = 3$

Hierarchische Schaltkreise

Addierer für nichtnegative Zahlen

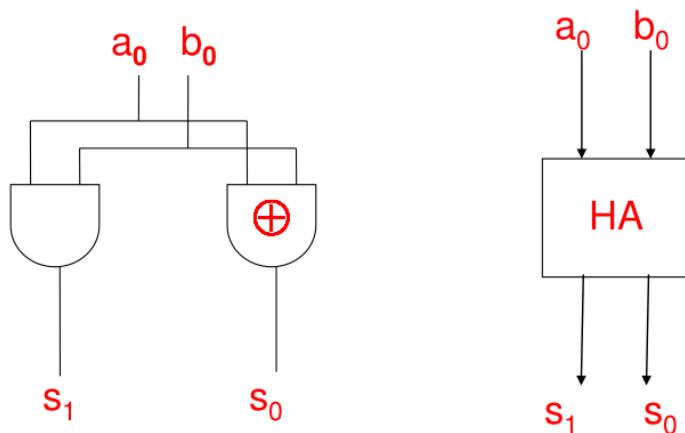
Formale Definition n-Bit-Addierer

Addieren nach der Schulmethode

Halbaddierer

$$S = \neg(\neg i_a \wedge \neg i_b) \vee (i_a \wedge i_b)$$

$$C_H = i_a \wedge i_b$$

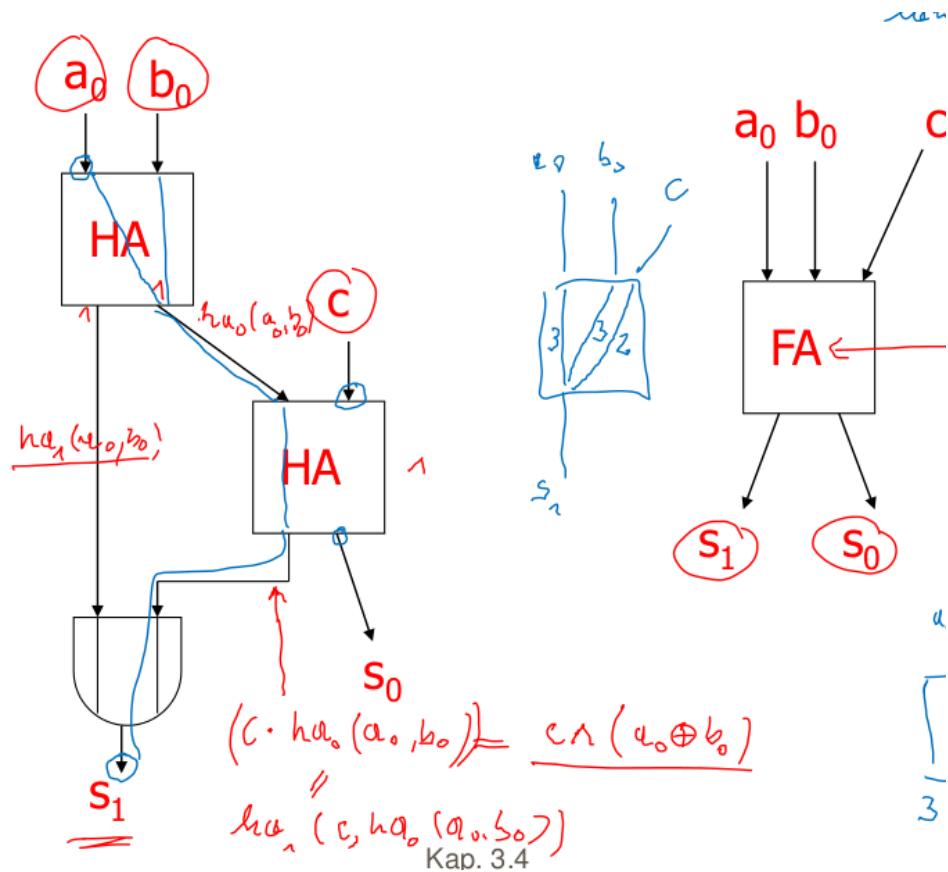


...
...

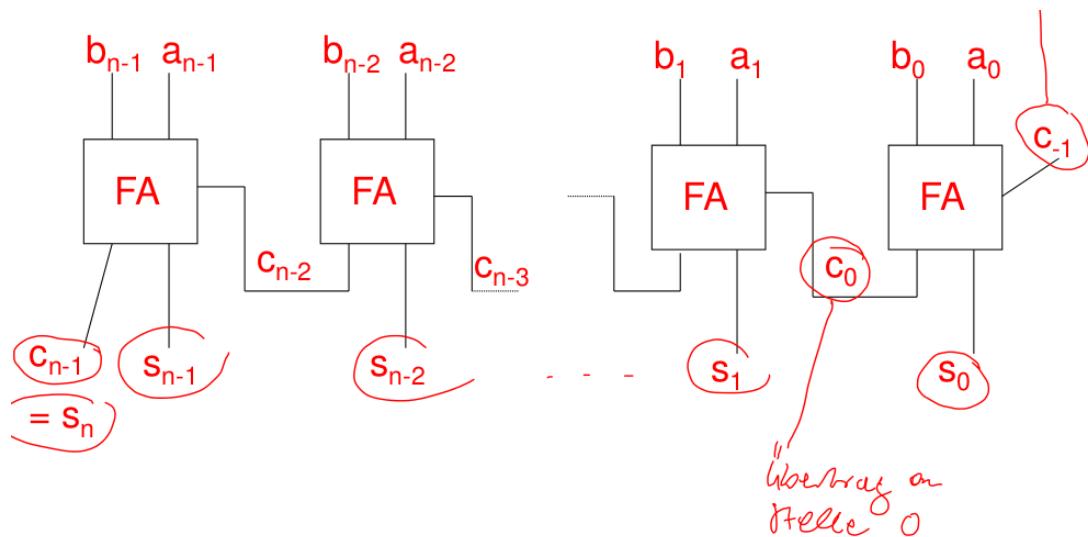
Volladdierer

$$S = \neg(\neg i_a \wedge \neg i_b) \vee (i_a \wedge i_b)$$

$$C_F = C_H \vee (i_c \wedge S)$$

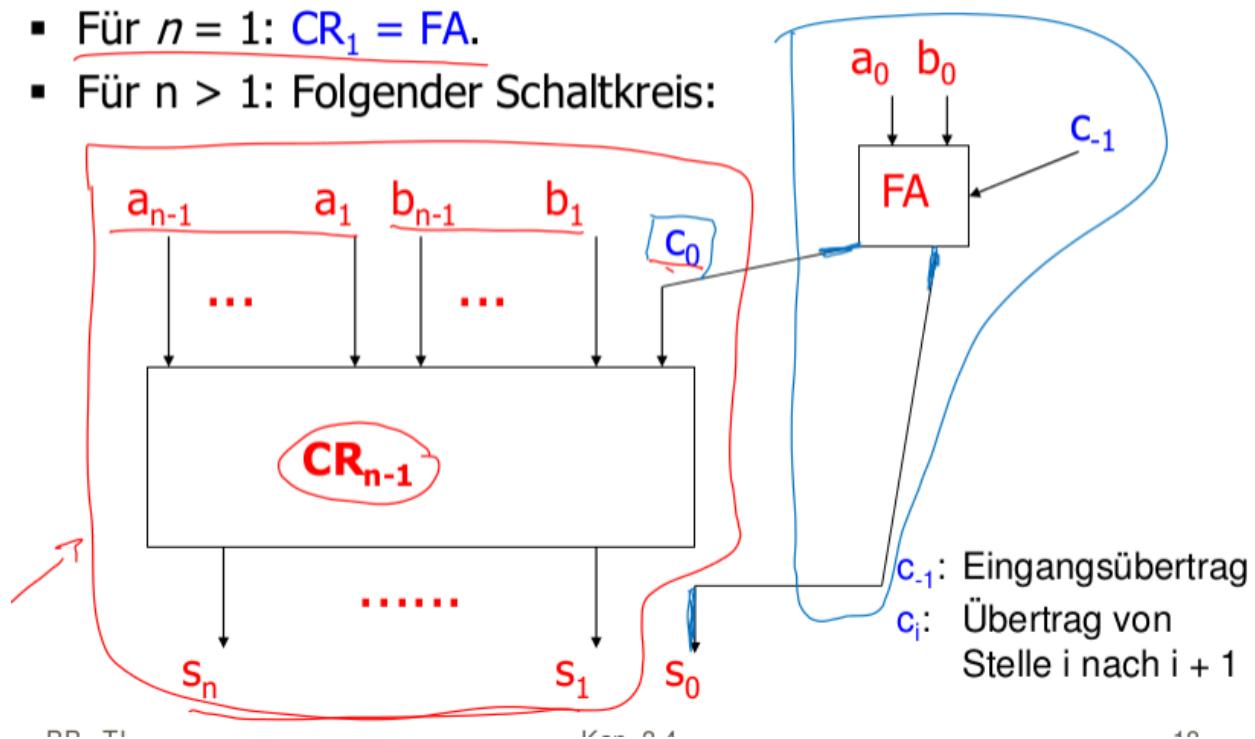


Carry-Ripple Addierer

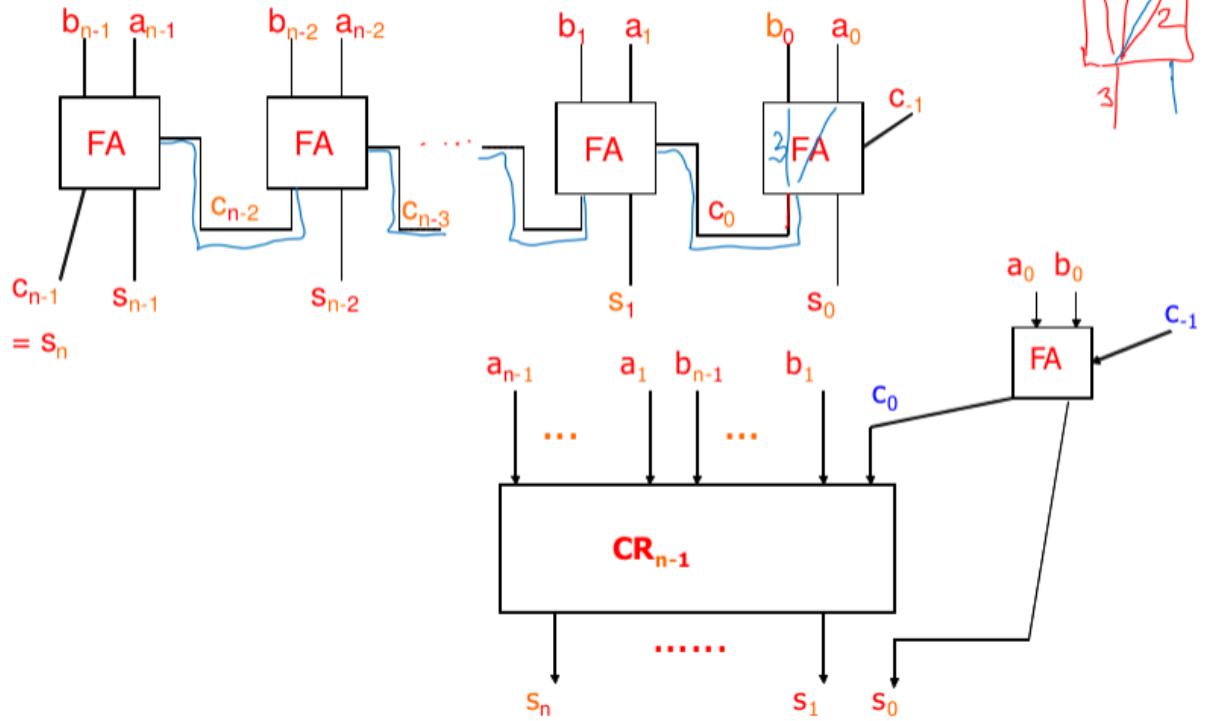


Induktive Definition des Carry-Ripple-Addierers CR

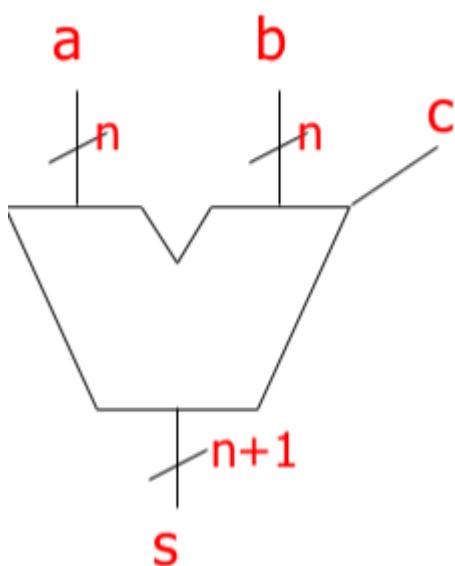
- Für $n = 1$: $\text{CR}_1 = \text{FA}$.
- Für $n > 1$: Folgender Schaltkreis:



Zwei (identische) Darstellungen von CR



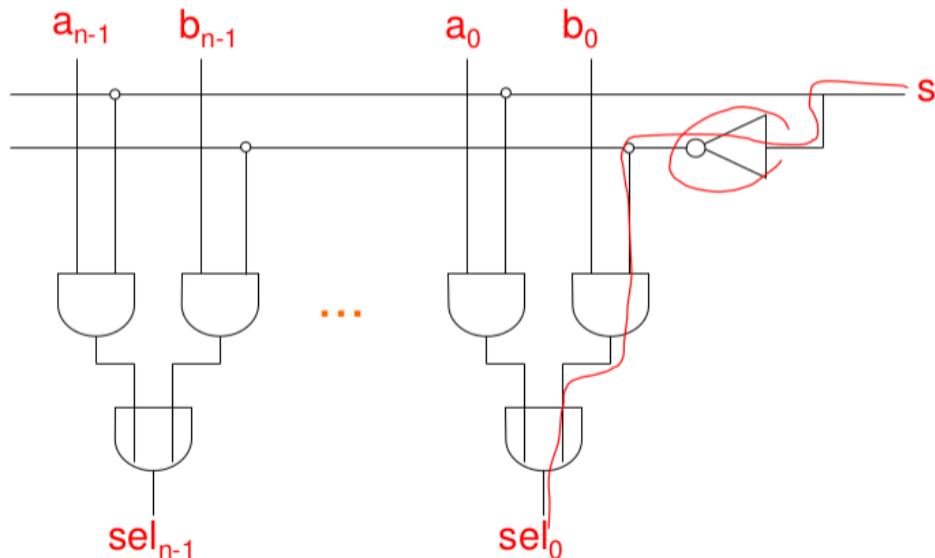
- $C(CR_n) = n \cdot C(FA) = 5n$
- $\text{depth}(CR_n) = 3 + 2(n - 1)$



Bus

nBit-Inkrementer

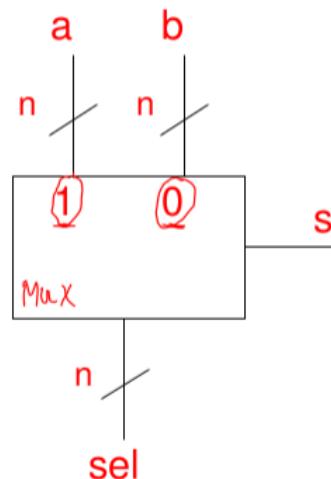
nBit-Multiplexer



Kosten und Tiefe:

$$C(MUX_n) = 3n + 1.$$

$$\text{depth}(MUX_n) = 3.$$

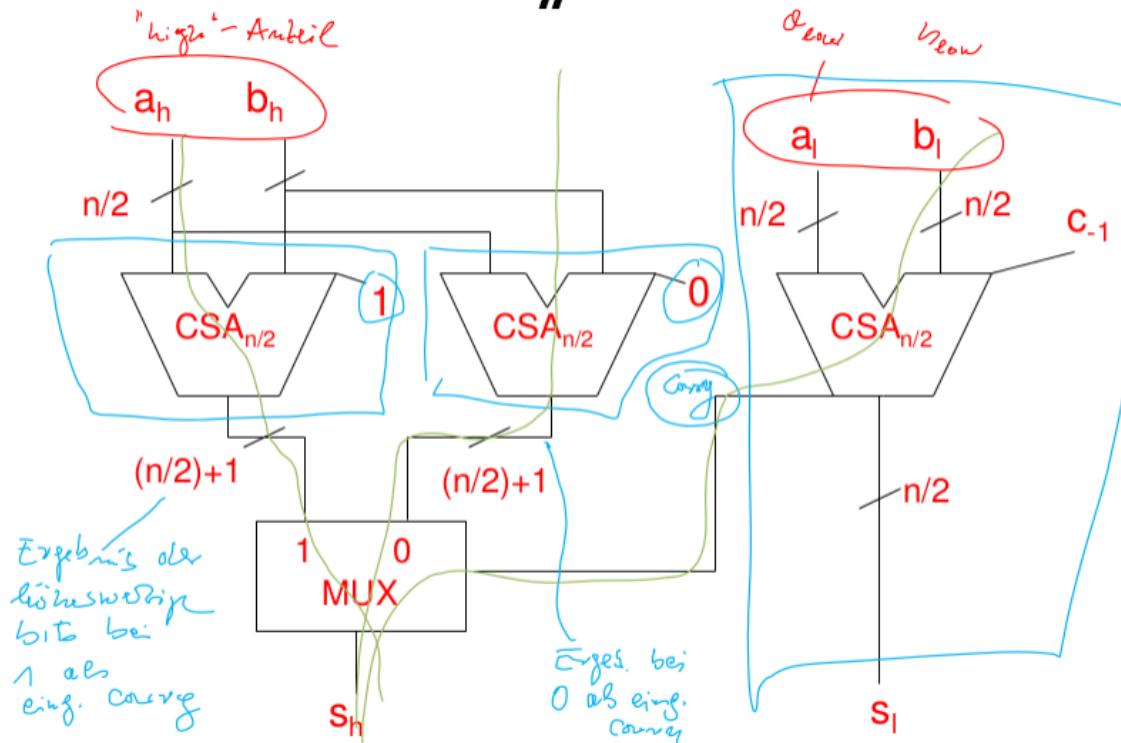


Bus vs Multiplexer

Conditional Sum Addierer

Addition von Zweierkomplement-Zahlen.

Aufbau von CSA_n



Komplexität von CSA_n : Tiefe

- Satz:** CSA_n hat Tiefe $\leq 3 \log n + 3$.

- Beweis:**

- $n = 1$: $\text{depth}(\text{CSA}_1) = \text{depth}(\text{FA}) = 3$.
- $n > 1$: $\text{depth}(\text{CSA}_n) \leq \text{depth}(\text{CSA}_{n/2}) + \text{depth}(\text{MUX}_{(n/2)+1})$
 $\leq \text{depth}(\text{CSA}_{n/2}) + 3$
 $\leq \text{depth}(\text{CSA}_{n/4}) + 3 + 3$
 $\leq \text{depth}(\text{CSA}_{n/8}) + 3 + 3 + 3$
 \dots
 $\leq \text{depth}(\text{CSA}_{n/(2^k)}) + k \cdot 3$
 $= \text{depth}(\text{CSA}_1) + k \cdot 3$
 $\leq 3 \cdot (k + 1) = 3 \log(n) + 3$.

$$n = 2^k$$

Satz (ohne Beweis): $C(\text{CSA}_n) = 10n^{\log 3} - 3n - 2$.

Subtrahierer

SRAM

RS-Flipflop

D-Latch

D-Flipflop

n-Bit Register

Schieberegister

Zähler

Aufbau eines Zählers

n-Bit Zähler

Treiber

+Tristate-Treiber

CISC vs. RISC

CISC (Complex Instruction Set Computer)	RISC (Reduced Instruction Set Computer)
Sehr großer CPU-Befehlssatz (üblicherweise mehr als 100 Befehle) - Befehle können oftmals sehr komplexe Operationen ausführen	Kleiner CPU-Befehlssatz, umfasst nur elementare Befehle und Operationen, dafür allerdings sehr schnell
Große Anzahl unterschiedlicher Adressierungsarten durch die die Befehle auf den Hauptspeicher zugreifen können	Nur wenige Adressierungsarten und wenige Befehle durch die auf den Speicher zugegriffen werden kann.
Da CISC auch über einfache Befehle verfügt, kommt es zu unterschiedlichen Ausführungszeiten bei der Abarbeitung => dies erschwert Pipelining	Nur einfache Befehle, die in einem Verarbeitungsschritt ausführbar sind. Alle Befehle sollen das gleiche Befehlsformat haben. Diese Vereinheitlichung ist wesentliche Voraussetzung für Pipelining
Direkte Unterstützung mehrerer Datentypen durch die Hardware	Befehle werden direkt durch die Hardware interpretiert.

CPU verfügt über kleine Anzahl an jeweils spezialisierten Registern für vorbestimmte Aufgaben	Großer interner Registersatz für eine schnelle bearbeitung vieler kurzer Befehle
In der Regel werden die einzelnen Befehle durch ein Mikroprogramm interpretiert, das auf einem ROM (Read-only memory) liegt und vom Chipset geliefert wird	Die Befehle werden direkt durch die Hardware interpretiert. Das Steuerwerk ist fast verdrahtet realisiert => Beschleunigung der Befehlsverarbeitung

Caching

Ein Rechner besitzt einen Cache für Instruktionen (Instruktionscache) und Daten (Datencache). Er ist software-transparent, dh. der Benutzer braucht nichts von der Existenz wissen. Ein Cache versucht die Zugriffszeiten (auf bspw. DRAM) gering zu halten, indem Daten, die als nächstes benötigt werden dort gespeichert werden (→ Lokalitätsprinzip).

Lesezugriff

CPU überprüft, ob Kopie der Hauptspeicherzelle a im Cache abgelegt ist.

- cache hit (im Cache vorhanden):
 - Es wird aus dem Cache gelesen. Überprüfung und Lesen erfolgt ohne Wartezyklen.
- cache miss (nicht im Cache):
 - CPU greift auf Arbeitsspeicher zu, Datum wird in Cache und CP gleichzeitig geladen.

Die mittlere Zugriffszeit beim Lesen beträgt: $c + (1-h)*m$ (Zugriffszeit auf Cache **c**, Zugriffszeit auf Hauptspeicher **m**, Trefferrate **h**)

Cache als assoziativer Speicher

Die angelegte Adresse wird parallel mit allen im Adressspeicher des Caches vorhandenen Adressen verglichen. Jedes neue Datum kann an jeder beliebigen freien Stelle im Cache abgelegt werden.

Verdrängen alter Daten aus dem Cache

Verhalten bei **cache miss** und **vollem Cache**: verdränge Block aus dem Cache, lade an seiner Stelle den gerade benötigten Block.

Verdrängungsstrategien:

- **Least Recently Used (LRU)** verdränge den Block, der am längsten nicht benutzt wurde.
- **Least Frequently Used (LFU)** verdränge den Block, auf den am wenigsten zugegriffen wird.

- **First in, First out (FIFO)** verdränge den Block, der am längsten im Cache ist.

Direct Mapped Cache

Feste Abbildung der Hauptspeicher-Adresse auf die Cache-Adressen, deshalb ist kein assoziativer Speicher und keine Verdrängungsstrategien nötig. Die von der CPU angelegte Adresse wird in 2 Teile gespalten: Die i niederwertigen Bits adressieren einen Eintrag im Adressspeicher, dieser Eintrag wird mit den $n-i$ höherwertigen Bits der Adresse verglichen. Speicherzellen des Hauptspeichers, deren i niederwertige Stellen gleich sind, werden auf die gleiche Position im Cache abgebildet.

Schreibzugriff

- cache hit (Hauptspeicherzelle a im Cache)
 - write-through: CPU schreibt Datum gleichzeitig in Hauptspeicher und Cache
 - write-back: CPU schreibt Datum nur in Cache, Speicherzelle wird mit "dirty bit" markiert. Die Hauptspeicherzelle wird erst aktualisiert, wenn die Kopie aus dem Cache verdrängt wird.
Dadurch werden Schreibzugriffe auf Cache ohne Wartezyklus möglich, Datenkonsistenz ist aber schwieriger.
- cache miss
 - write-non-allocate: CPU schreibt Datum in Hauptspeicherzelle mit Adresse a, Cache bleibt unverändert
 - write-allocate: CPU schreibt Datum in den Cache

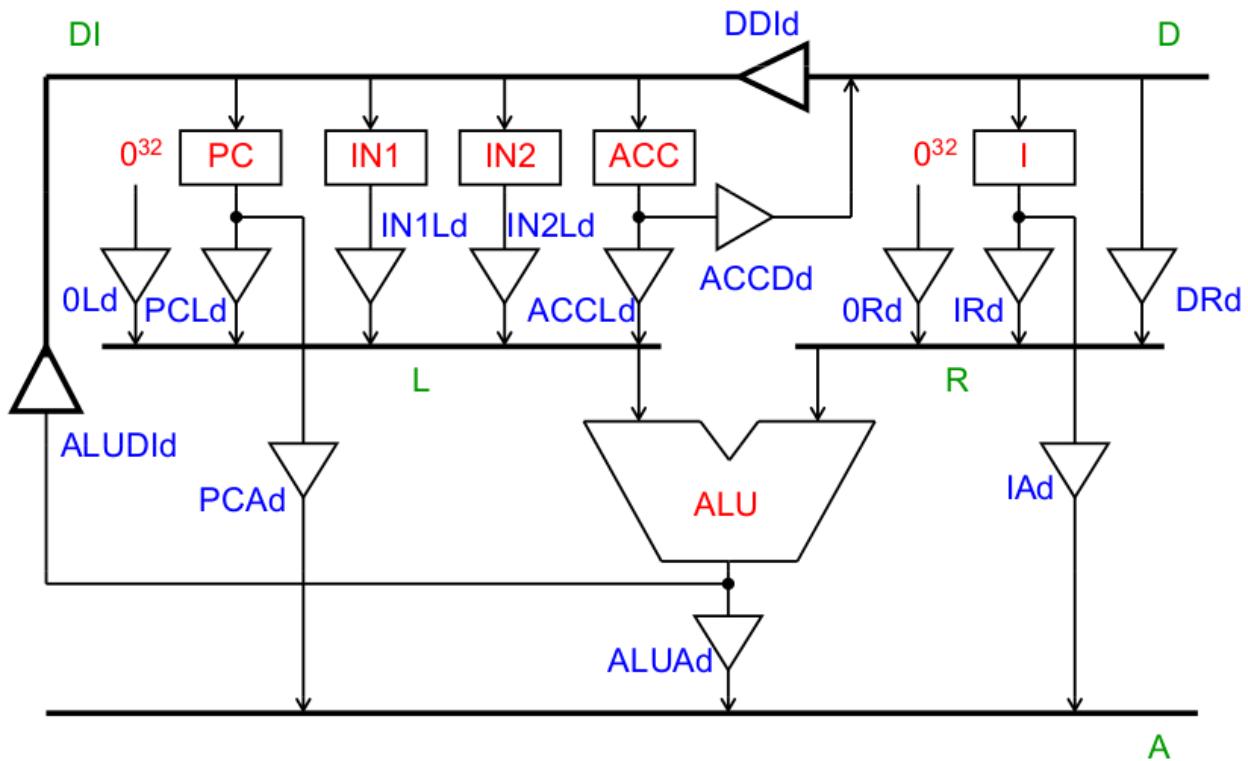
Automaten

Mealy
Moore

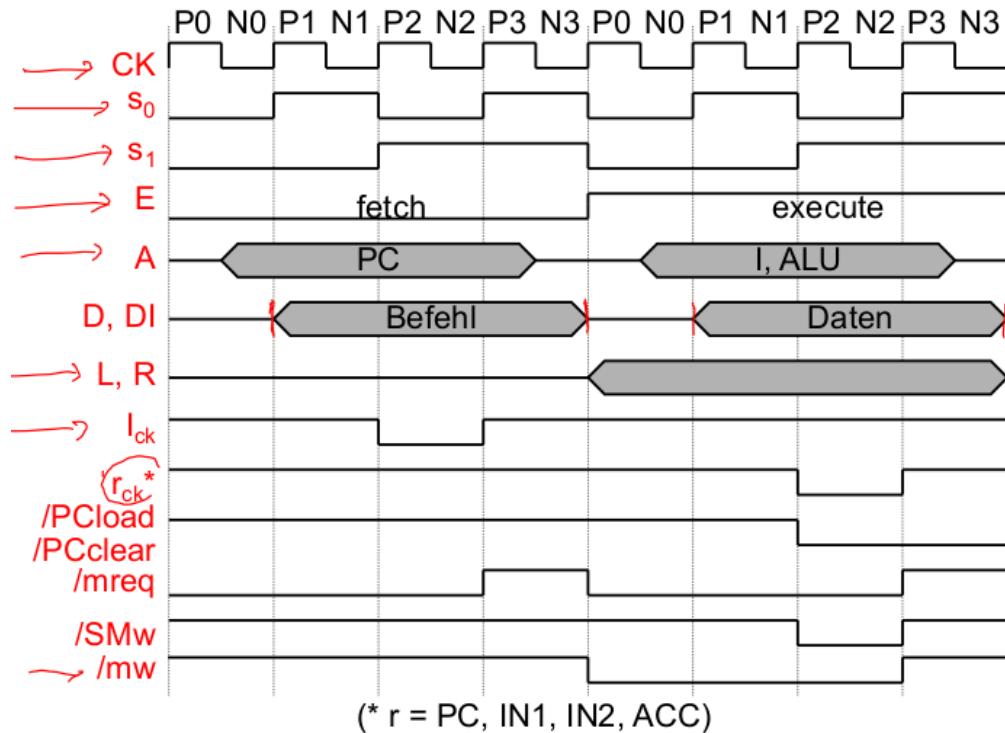
Datenpfade

Am besten Aufzeichnungen anschauen.
Siehe auch Aufgabe 7 und 8.

Wir benötigen die Screenshots von den einzelnen Datenbefehle (der Befehle Load, Store, usw.). Die müssen wir uns nämlich einprägen und damit das Timing berechnen üben.



Datenpfade Timing Diagramm



Schritte mit Instruktionsformat kombinieren

Beispiel (ff)

PCLdœ := /E * s1 * s0 * I31 * I30 } ; Start bei P0 von Jump

+ /E * s1 * s0
 * /I31 * /I30 ; Compute ; D = PC
 * /I25 * /24 ; D = PC
 * /E * s1 * s0
 * I31 * /I30 * I29 * I28 ; MOVE 1011
 * /I27 * /I26 ; S = PC

+ ...

11
kodierung für Jump

00
keine für Compute
PC = 00



Halteterme

Angenommen wir haben PCLoe definiert. Aber das Signal soll über auch die nächsten Takte so sein. Dann kann man das wie folgt notieren:

- ★ + PCLdœ * E * /s1 * /s0 ; bei P1 halten
- + PCLdœ * E * /s1 * s0 ; bei P2 halten
- + PCLdœ * E * s1 * /s0 ; bei P3 halten

Bus contention

Bus contention, in computer design, is an undesirable state of the bus in which more than one device on the bus attempts to place values on the bus at the same time.

Lösungen

Klausur WS2011/2012

Aufgabe 1

a)

	0101101	0011110	1010001	1100010
0101101	-			
0011110	4	-		
1010001	5	5	-	
1100010	5	5	4	-

b) $\text{dist}(c) \geq k + 1$ $\text{dist}(c) = 4$ $4 \geq k + 1$ $k = 3$

c) $\text{dist}(c) \geq 2k + 1$ $\text{dist}(c) = 4$ $4 \geq 2k + 1$ $k = 1$

d) MINIMALPOLYNOM $A = \{M, I, N, A, L, P, O, Y\}$ $2^k = 8 \Rightarrow k = 3$

Die Zeichenkette ist 14 Zeichen lang.

$\Rightarrow 3 \text{ Bits} * 14 \text{ Zeichen} = 42 \text{ Bits}$

~~$\Rightarrow \text{Es werden } 112 \text{ Bits benötigt um "MINIMALPOLYNOM" zu kodieren.}$~~

e)

M	I	N	A	L	P	O	Y
3	2	2	1	2	1	2	1

Y=1111

O=011

P=1110

L=110

A=010

N=101

I=100

M=00

MINIMALPOLYNOM

$2+3+3+3+2+3+3+4+3+3+4+3+2 = 41 \text{ Bits}$

Es werden 41 Bits benötigt wenn man es mit dem Huffman "von Hand auf Papier" löst und anschließend als Binärkode zusammenbaut.

f) Aufgabenstellung unvollständig.

Aufgabe 2

- a) $[d_n, d_{n-1}, \dots, d_0, \dots, d_{-k}] := \sum_{i=-k, \dots, n-1} d_i 2^i - d_n 2^n$
- b) $[\bar{a}]_1 = -[a]_1 \Leftrightarrow [\bar{a}]_1 + [a]_1 = 0 = [0 \dots 0] \Leftrightarrow [a]_1 - [0 \dots 0] = [\bar{a}]_1 \Leftrightarrow [a]_1 + [1 \dots 1] = [\bar{a}]_1$

Obiges würde meiner Meinung nicht reichen!

Beweis von Dr. Paul Molitor:

Beweis:

$$\begin{aligned}
 [d']_1 + [d]_1 &= \\
 &= \left(\sum_{i=-k}^{n-1} d_i \cdot 2^i \right) - d_n \cdot (2^n - 2^{-k}) + \left(\sum_{i=-k}^{n-1} d'_i \cdot 2^i \right) - d'_n \cdot (2^n - 2^{-k}) \\
 &= \left(\sum_{i=-k}^{n-1} (d_i + d'_i) \cdot 2^i \right) - (d_n + d'_n) \cdot (2^n - 2^{-k}) \\
 &= \left(\sum_{i=-k}^{n-1} (d_i + (1 - d_i)) \cdot 2^i \right) - (d_n + (1 - d_n)) \cdot (2^n - 2^{-k}) \\
 &= \sum_{i=-k}^{n-1} 2^i - (2^n - 2^{-k}) \\
 &= (2^n - 2^{-k}) - (2^n - 2^{-k}) \\
 &= 0
 \end{aligned}$$

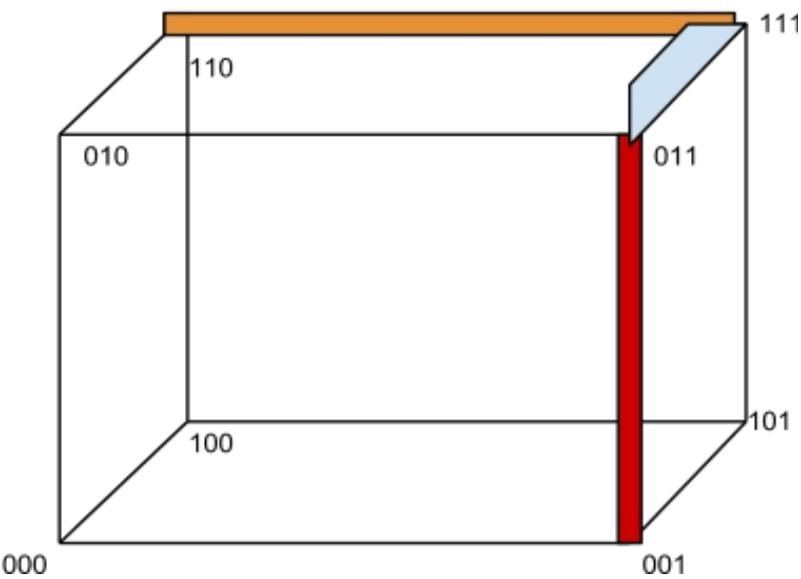
Der Beweis ist gut.

Aufgabe 3

a) $p_g = \neg x_1 x_3 \vee x_2 x_3 \vee x_1 x_2$

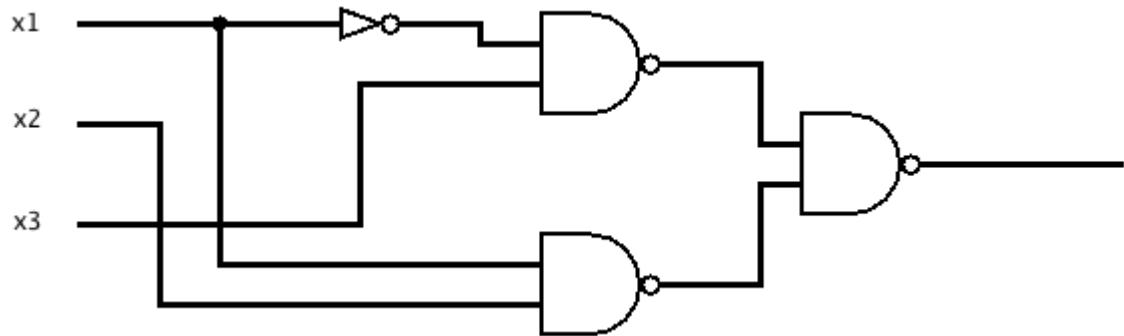
b)

$ON(p_g) = \{001, 011, 110, 111\}$



Der Hypercube zeigt, dass das Monom x_2x_3 weggelassen werden kann. Daher:
 $p_{min} = \neg x_1x_3 \vee x_1x_2$

c)



d)

$$\begin{aligned}
 X_3 &= \{x_1, x_2, x_3\} \\
 Y &= \{p_4\} \\
 G &= \{V, E\} \\
 V &= X_3 \cup \{v_1, v_2, v_3, v_4\} \cup Y \\
 E &= \{(x_1, v_1), (v_1, v_2), (v_2, v_3), (v_3, p_4), (x_1, v_4), (v_4, v_3)\} \\
 \text{typ} &= \{(v_i \mapsto \text{NOT}) \cup (v_i \mapsto \text{NAND}) / i \in \{2, 3, 4\}\} \\
 \text{in} &= \{(v_1 \mapsto x_1), (v_2 \mapsto (v_1, x_3)), (v_3 \mapsto (v_2, v_4)), (v_4 \mapsto (x_1, x_2))\}
 \end{aligned}$$

Aufgabe 4

$$ON(f) = \{0000, 0001, 0010, 0011, 0101, 0110, 0111, 1000, 1010, 1110, 1111\}$$

1. Initialisierung von L_0 : Nur Minterme, Sortieren
 - a. $L_0 = \{0000, 0001, 1000, 0010, 1010, 0011, 0101, 0110, 0111, 1110, 1111\}$
2. Iteratives Vorgehen

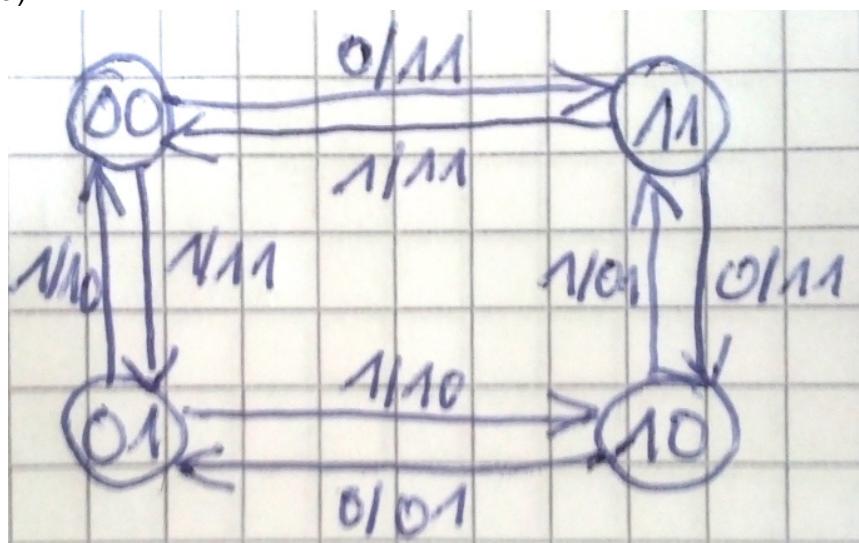
L_0	L_1	L_2
0000	-000	00--
-----	-010	-0-0
0001	-111	0--1
1000	-110	0-1-
0010	0-01	--10
-----	0-11	-11-
1010	1-10	
0011	10-0	
0101	01-1	
0110	00-0	
-----	00-1	
0111	000-	
1110	011-	
-----	111-	
1111	001-	
P_0	P_1	P_2

		00-- -0-0 0--1 0-1- --10 -11-
--	--	--

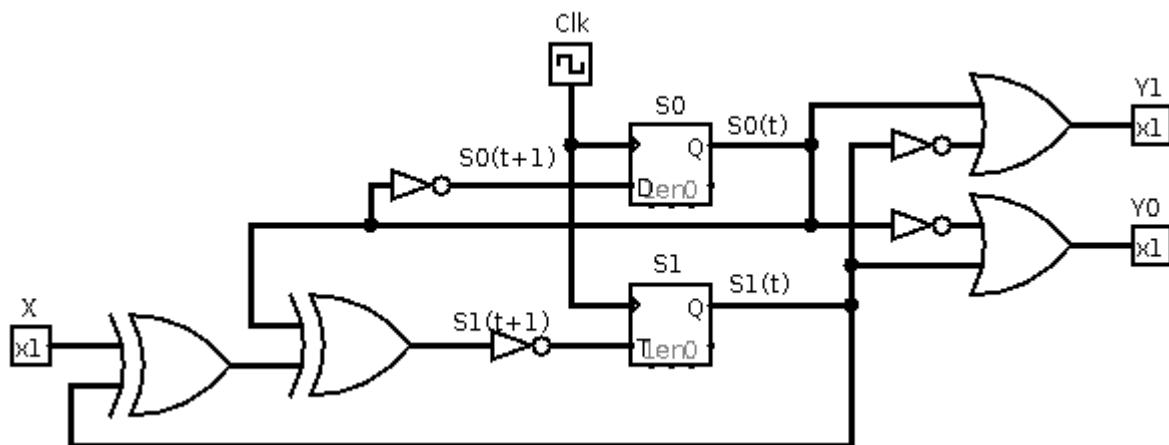
$$Prim_f = P_1 \cup P_2 = \{-000; -0-0; 0--1; 0-1-; --10; -11-\}$$

Aufgabe 5

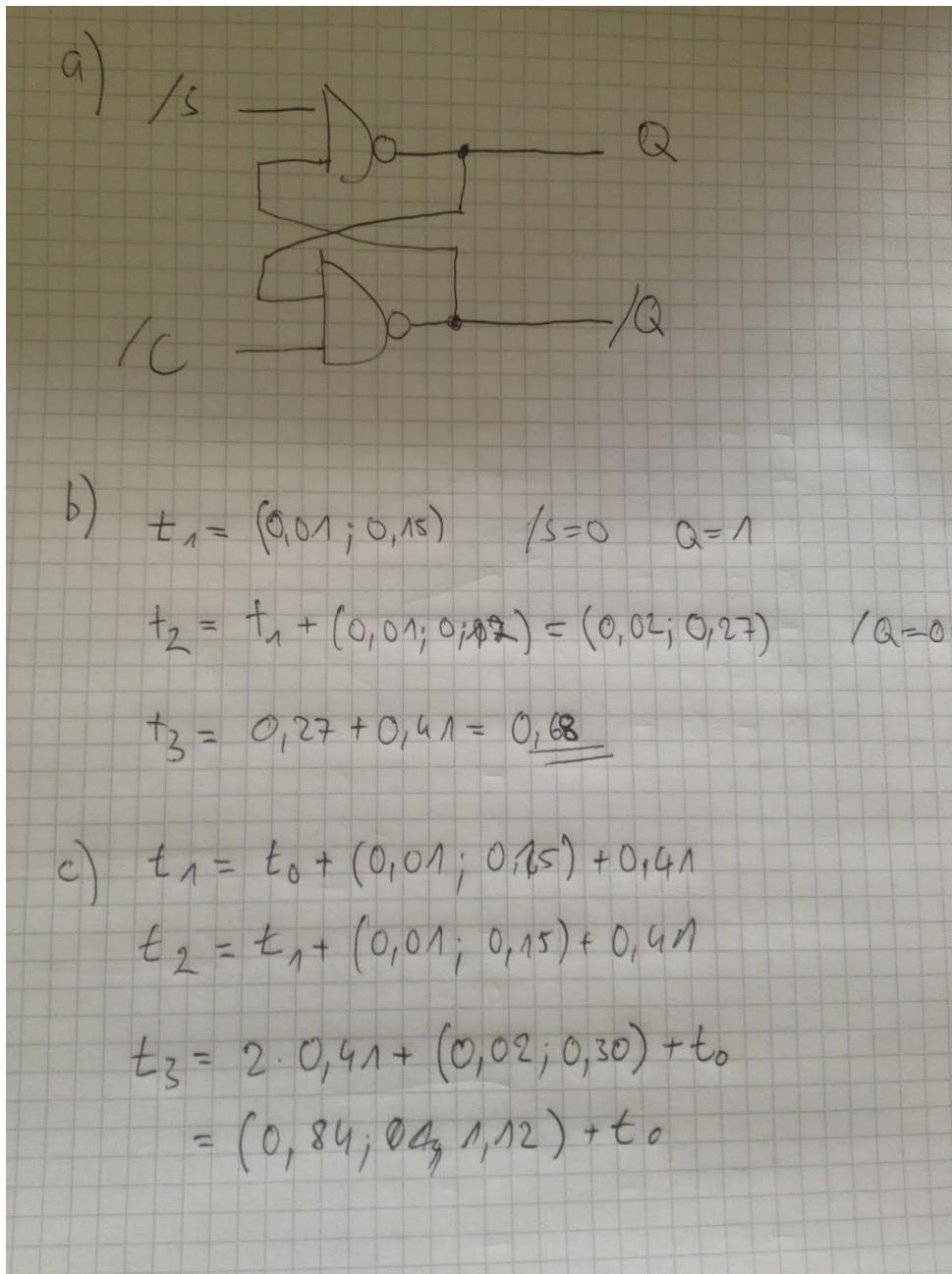
a)



b)

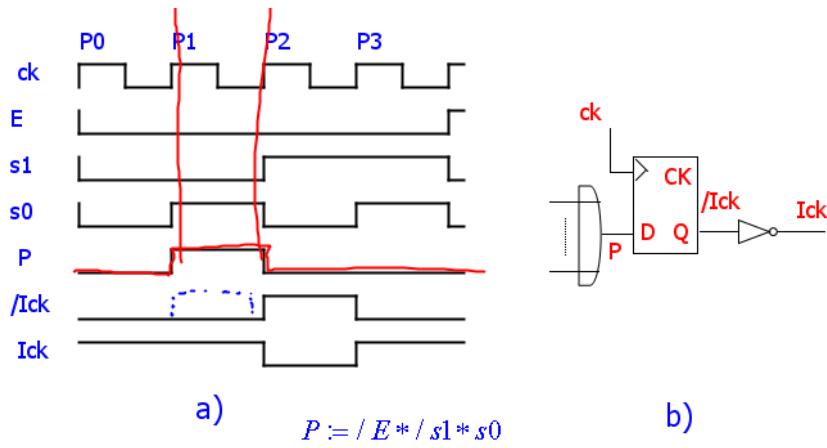


Aufgabe 6



Aufgabe 7

Um die Aufgabe zu Lösen lohnt es sich folgende Grafik aus dem Script zu betrachten:



$$P := /E * s1 * s0$$

Es gibt hier 3 wichtige Informationen die man Wissen muss:

1. Das Signal in der Schaltung nach dem OR-Gatter erzeugt also bei P. Und da dieses OR-Gatter vor dem FlipFlop ist, muss das Signal fdem gewollten Signal erzeugt werden.
2. Muss das Signal invertiert vorliegen.
3. Dürfen wir das Signal nicht Ick nennen, wir nennen es /Ick (auch wenn P eigentlich richtig wäre und auch hier auf dieser Folie so geschrieben wird. Aber diese Info kommt erst auf der nächsten Folie.)

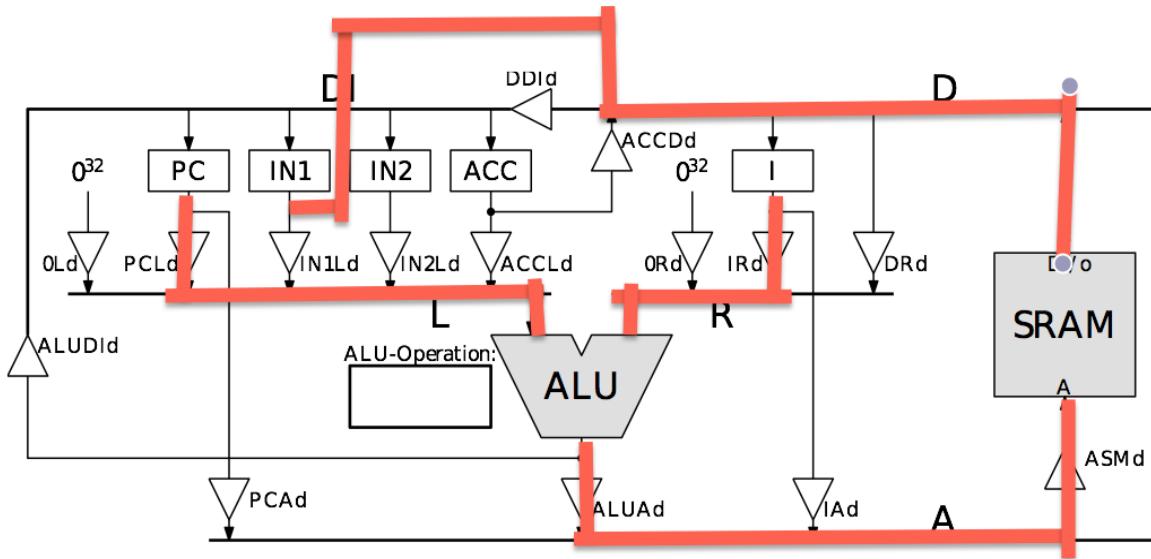
$$a) /I_{CK} = \neg E * \neg S_1 * \neg S_0$$

$$b) /IN2_{CK} = E * \neg S_1 * \neg S_0$$

Aufgabe 8

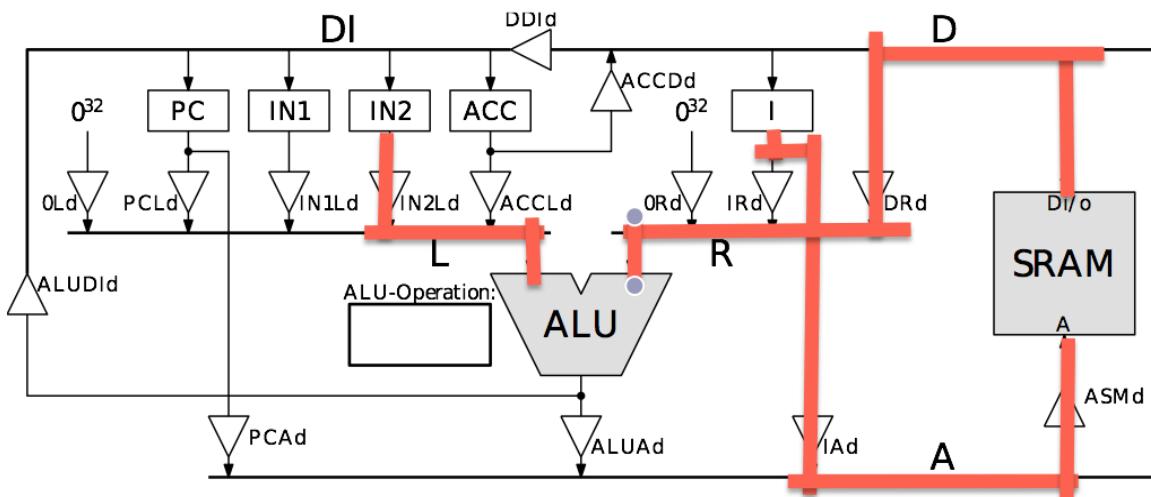
a)

STOREREL S, i; [M(<PC>) + [i]] := [S]



b)

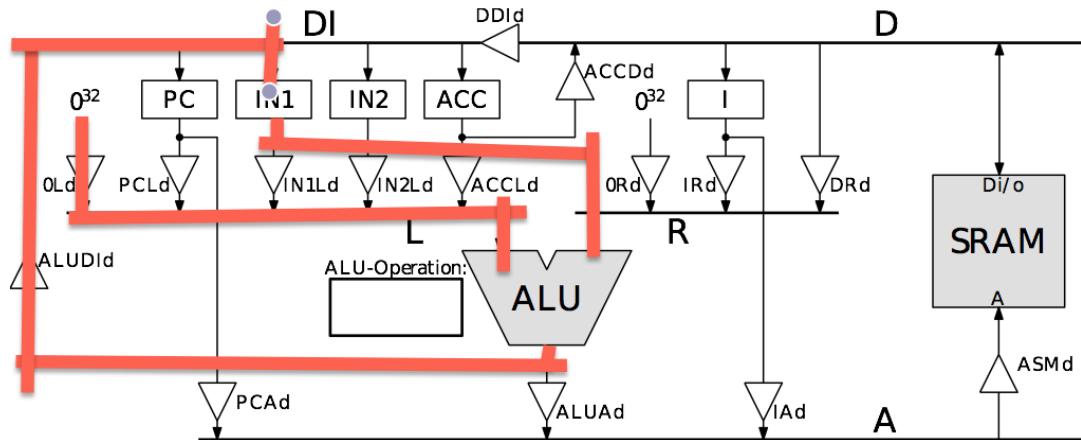
XORMEM D, i; [M(<i>)] := [M(<i>)] ⊕ [S]



ALU-Operation: XOR
Leider nicht möglich, da der Datenbus bereits belegt ist.

c)

$NEG\ S; [S] := -[S]$

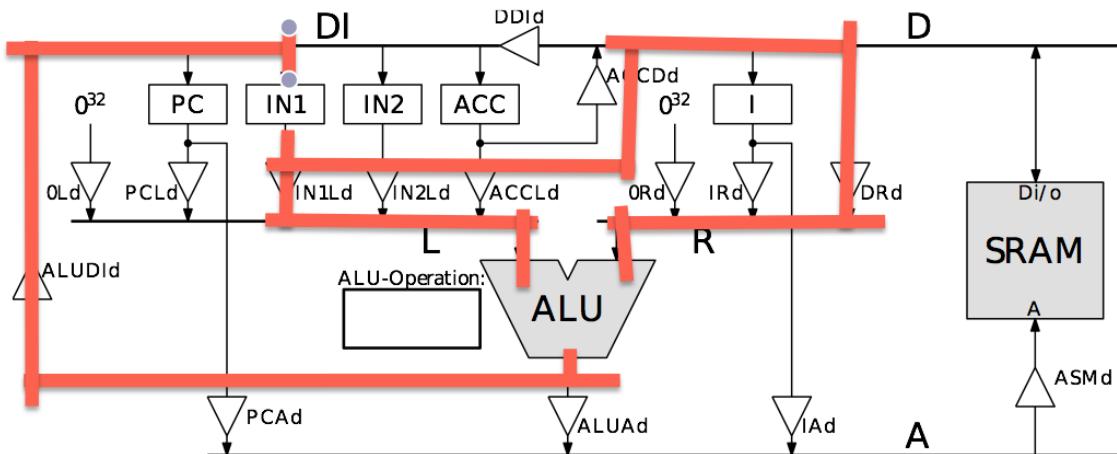


ALU-Operation: Subtraktion

Hinzugefügte Elemente: IN1Dd

d)

$LEFTSHIFT\ S; (s_{31}s_{30}\dots s_0) := (s_{30}\dots s_00)$



ALU-Operation: +

Hinzugefügte Treiber: IN1Dd

Aufgabe 9

a) $1024 = 2^{10}$ $4 = 2^2$ Deshalb ist der Adress-Tag 10-2 = 8 Bits groß.

- b) Der Adress-Tag sollte in den höherwertigen Bits der Speicheradresse gelegt werden, mit Adressen, die im Adressraum des Hauptspeichers nahe beieinander liegen, nicht auf dieselbe Position im Cache abgebildet werden.
c) Bei einem assoziativem Cache ist der parallele Vergleich mit allen im Cache gespeicherten Adressen notwendig. Das ist bei einem großen Cache deutlich langsamer, da beim DMC nur die Dekodierung und ein Vergleich von zwei Adress-TAGs notwendig ist.

Aufgabe 10

a)

$x \wedge 1 = 1$ Behauptung

$$x \wedge (1 \vee \bar{1}) = (x \wedge 1) \vee (x \wedge \bar{1}) = 1 \vee (x \wedge \bar{1}) = 1 \vee (x \wedge 0) = 1 \vee 0 = 1$$

Angewendete Regeln: Auslöschung (rückwärts), Distributivgesetz, Behauptung eingesetzt,

$\bar{1} = 0$ (Dualität), $x \wedge 0 = 0$ (Extremalgesetz), $x \vee 0 = x$ (Nullelement)

b)

In einer Booleschen Algebra gilt $\forall x \in B \ \exists e \in B : x \vee e = e$.

Angenommen es existiert neben $e_1 = 1$ noch $e_2 = \bar{1}$ ($e_1, e_2 \in B$), dann müsste auch $e_1 \vee e_2 = e_2$ gelten. Das heißt:

$$1 \vee \bar{1} = \bar{1} \Leftrightarrow 1 \vee 0 = 0 \Leftrightarrow 1 = 0 \quad (\text{Widerspruch!})$$

Die Aussage ist also falsch, 1 ist also eindeutig.

Aufgabe 11

a)

depth (ADD₃) = 6 (Pfad des Carry-Bits)

$$C(\text{ADD}_3) = 5 * 3 = 15$$

b)

$$S = \overline{(i_a \wedge \overline{i_b}) \vee (i_a \wedge i_b)}$$

$$C_{\text{Half}} = (i_a \wedge i_b)$$

$$C_{\text{Full}} = C_{\text{Half}} \vee (i_c \wedge S)$$

$a = 010$ $b = 110$

$$S_0 = \overline{(0 \wedge 1) \vee (1 \wedge 0)} = 0$$

$$C_1 = (0 \wedge 1) \vee (0 \wedge 0) = 0$$

$$S_1 = \overline{(0 \wedge 0) \vee (0 \wedge 1)} = 0$$

$$C_2 = (1 \wedge 1) \vee (0 \wedge 0) = 1$$

$$S_2 = \overline{(1 \wedge 0) \vee (0 \wedge 1)} = 1$$

$$S_3 = (0 \wedge 1) \vee (1 \wedge 1) = 1$$

c)

$$c = (a \wedge b) \vee (c \wedge (\bar{a}_0 \wedge b) \vee (\bar{a}_1 \wedge \bar{b}))$$

~~a₀~~

$$f(a_0, b_0) = (a_0 b_0) \cancel{\vee}$$

$$f(a_0, b_0, a_1, b_1) = (a_1 \wedge b_1) \vee ((a_0 b_0) \wedge \overline{(a_1 \wedge \bar{b}_1)} \wedge \overline{\vee (a_1 \wedge b_1)})$$

$$f_{a_0=0} = (a_1 \wedge b_1) \quad f_{a_0=1} = a_1 b_1 \vee (b_0 \wedge \dots)$$

$$f_0 = a_1 b_1 \quad f_1 = a_1 b_1 \vee (b_0 \wedge ((\bar{a}_1 \wedge \bar{b}_1) \vee (a_1 \wedge b_1)))$$

$$f^1_{b_0=0} = (f_{a_0=0})_{b_0=0} = a_1 b_1 \quad f^2_{b_0=0} = (f_{a_0=1})_{b_0=0} = a_1 b_1$$

$$f^1_{b_0=1} = \dots_{b_0=1} = a_1 b_1 \quad f^2_{b_0=1} = (f_{a_0=1})_{b_0=1} = \frac{a_1 b_1}{\vee (b_0 \wedge ((\bar{a}_1 \wedge \bar{b}_1) \vee (a_1 \wedge b_1)))}$$

$$f^3_{a_1=0} = 0 \quad f^4_{a_1=0} = \emptyset$$

$$f^3_{a_1=1} = 0 \quad f^4_{a_1=1} = b_1 \vee \cancel{b_1} = 1$$

