Vorlesung

Grundlagen der Informatik

Skript zur Vorlesung an der Fachhochschule Birkenfeld
WS 96/97

Dr. Arno Formella

© Arno Formella, Universität des Saarlandes, FB14 Informatik, LS Prof. W.J. Paul

Tel.	(0681) 302 5538
FAX	(0681) 302 4290
e-mail	`formella@cs.uni-sb.de`
web	`http://www-wjp.cs.uni-sb.de/~formella`

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Woher kommt Informatik? 1
- 1.2 Was ist Informatik? 1
- 1.3 Historisches 1
- 1.4 Erste Rechenanlagen 1
- 1.5 Erste Algorithmen 2
- 1.6 Problemlösungsstrategie 2
- 1.7 Ziel der Vorlesung 2

2 Grundlagen **3**
- 2.1 Begriffe aus der Mengenlehre 3
- 2.2 Induktionsbeweis 4
- 2.3 Funktionen 4
- 2.4 Wichtige Rechenregeln 6
  - 2.4.1 Rechnen mit Potenzen und Logarithmen 6
  - 2.4.2 Rechnen mit Summen 7
  - 2.4.3 Geometrische Reihe 8
  - 2.4.4 Rekursionsgleichungen 9
- 2.5 Zahlendarstellungen 11
  - 2.5.1 Unärdarstellung 11
  - 2.5.2 Zahlendarstellung zu einer Basis 12
  - 2.5.3 Darstellung negativer Zahlen 13

3 Schaltkreistheorie **15**
- 3.1 Hardware-Ebenen und „Digitalität“ 15
- 3.2 Schaltfunktionen 16
- 3.3 BOOLE’sche Ausdrücke 17
- 3.4 Äquivalenz von BOOLE’schen Ausdrücken 20
- 3.5 BOOLE’sche Algebra 22
- 3.6 Weitere einfache Schaltfunktionen 22
- 3.7 BOOLE’sche Ausdrücke und Schaltfunktionen 23
- 3.8 Kosten BOOLE’scher Ausdrücke 25
- 3.9 Beispiele von Schaltfunktionen 25

4 Graphen und Schaltkreise **28**
- 4.1 Endlicher gerichteter Graph 28
- 4.2 Schaltkreise als Graphen 30
- 4.3 Kosten und Tiefe eines Schaltkreises 31

5 Spezielle Schaltfunktionen **32**
- 5.1 Multiplexer 32
- 5.2 Decoder 33
- 5.3 Rekursive Schaltkreiskonstruktion 35
- 5.4 Demultiplexer 39
- 5.5 Vergleicher 40

6 Addierer und ALUs **45**
- 6.1 1-Bit Addierer 45
  - 6.1.1 Halbaddierer 45
  - 6.1.2 Volladdierer 45
- 6.2 n-Bit Addierer 46
  - 6.2.1 ‘ripple carry’ Addierer 47
  - 6.2.2 ‘conditional sum’ Addierer 49
  - 6.2.3 ‘two sum’ Addierer 51
  - 6.2.4 ‘block carry’ Addierer 53
  - 6.2.5 ‘carry look ahead’ Addierer 56

6.2.6 Zusammenfassung der Addierer 56
6.3 Subtrahierer 57
6.4 Bereichsüberschreitung 58
6.5 Arithmetisch Logische Einheit 60

7 Zusammenfassung der Symbole und Notationen 64

8 Technologien 65
8.1 Daten und Begriffe 65
8.2 Parameter zur Zusammenschaltung 67

9 ‘timing’-Analyse 70
9.1 Rechnen mit Intervallen 72
9.2 Detailliertes ‘timing’-Diagramm 72
9.3 ‘hazards’ 74
9.4 Speichernde Schaltungen 75
9.4.1 R/S-Flipflop 75
9.4.2 D-Latch 76
9.4.3 D-Flipflop 77

1 Einleitung

Motivation: Geschichte, Hintergrund, Ziel der Vorlesung

1.1 Woher kommt Informatik?

Man könnte heute denken:

Informatik = *Inform*ation + Autom*atik*

kam wohl aus dem Französischen:

Informatik = von franz. „informatique“

im englischen Sprachraum:

Informatik = ‘computer science’

1.2 Was ist Informatik?

Wissenschaft (und Technik) der (automatischen) Informationsverarbeitung

man kann unterscheiden:

Informatik	Methode	verwandte Bereiche
theoretische	beweisen	Mathematik
praktische	programmieren	das wohl eigenständigste
technische	bauen	Elektrotechnik
angewandte	benutzen	alle Disziplinen

Erforschung grundsätzlicher Verfahrensweisen

Theorie gliedert sich in:

Architekturtheorie: „Worauf kann gerechnet werden?“;

Informationstheorie: „Womit/Worüber kann gerechnet werden?“;

Algorithmentheorie: „Wie kann etwas berechnet werden?“

Komplexitätstheorie: „Was ist berechenbar? Welcher Aufwand ist dazu notwendig? Wie lange dauert die Berechnung?“

Die Einteilung ist nur prinzipiell, alle Bereiche sind sehr eng miteinander verknüpft.

1.3 Historisches

Erfindung der Zahlen z. B. Araber, wir nutzen deren Schreibweise; Versuche Rechnen zu mechanisieren sind schon sehr alt; Babylonier hatten Rechentechniken, die auf Stellenwertsystem beruhten; logisches Schlussfolgern und Axiomatisierung der Griechen; Abacus als Rechenhilfe der Chinesen; Automaten zu Anfang der Neuzeit (Glockenspiele, Musikmaschinen); schematische Rechenverfahren nach Adam Riese und Leibniz.

1.4 Erste Rechenanlagen

Schickardt, Pascal, Leibniz: 17. Jahrhundert, Beschreibung vollmechanischer Rechenanlagen, unabhängig voneinander angegeben,

Babagge: 19. Jahrhundert, Tischrechner mit Programmsteuerung und Speicher, nicht vollendet (es gibt einen zweiten, modernen Versuch),

Zuse: 1941 Z3, elektromechanischer programmierbarer Rechner mit Gleitkommaarithmetik z. B. $1.023 \cdot 109.34$ (unabhängig von Babagge),

Eckart, Mauckley: 1943–1945 ENIAC, elektronische Rechenanlage mit 18000 Röhren,

Aiken: 1944, MARK-I, elektromechanische Rechenanlage,

Stiebitz: 1945, Relaisrechner,

vonNeumann: 1945 > jetzt ging es rasend ...

Aus dem „Zeitalter“ der elektromechanischen Rechner blieb das Wort
‘de bug ging’, von „Käfer-Entfernen“, erhalten.

Wichtig für die Informatik: das Konzept eines Algorithmus (von IBN MUSA AL-KHOWARIZMI, 825)

Algorithmus = Verfahrensweise, Vorgehensweise, Abfolge präziser Schritte, Rezept, mechanisierbarer Prozess, Rechenschema,

1.5 Erste Algorithmen

Multiplikationsalgorithmus der Ägypter, benutzte Operationen sind sehr einfach; Addition, Multiplikation mit 2, ganzzahlige Division durch 2, konnte alles durch Abzählen mit Steinen (Unärsystem) leicht erledigt werden:

x	$\cdot$	y		$= ?$	$7 \cdot 22$		$= ?$
x_0	$= x$	y_0	$= y$		7	22	—
x_1	$= 2 \cdot x_0$	y_1	$= y_0/2$		14	11	
x_2	$= 2 \cdot x_1$	y_2	$= y_1/2$		28	5	
$\cdots$					56	2	—
x_k	$= 2 \cdot x_{k-1}$	y_k	$= 1$		112	1	
$\sum x_i$	alle Zeilen mit y_i ungerade				154		

1.6 Problemlösungsstrategie

- Problembeschreibung (beschreibe)
- Problemanalyse (suche Ähnliches, zerteile)
- Lösungsmodell (abstrahiere)
- Lösungsverfahren, Algorithmus (schematisiere)
- Korrektheitsnachweis (überprüfe)
- Kostenanalyse (bewerte)
- Implementierung (realisiere)
- Test (überprüfe)
- Dokumentation (mache zugänglich)

Die einzelnen Schritte sind nicht unabhängig voneinander und es kann sinnvoll sein, mehrere Wiederholungen der verschiedenen Teile zu durchlaufen.

1.7 Ziel der Vorlesung

von hier bis zum Mikroprozessor

2 Grundlagen

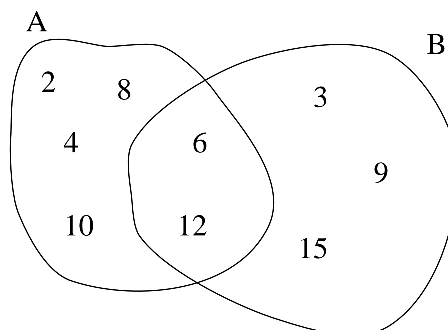
Motivation: Wiederholung von Bekanntem, Kennenlernen von Formalismus, Grundlagen einer gemeinsamen „Sprache“: Induktion, Funktionen, Zahlendarstellungen, Rekursionsgleichungen.

2.1 Begriffe aus der Mengenlehre

Notationen

$A, B, \ldots$	Mengen
$\lvert A \rvert$	Mächtigkeit (Anzahl der Elemente) der Menge
$\mathbb{N}$	Menge der natürlichen Zahlen
$\mathbb{N}_0$	einschließlich der Null
$\mathbb{Z}$	Menge der ganzen Zahlen $= \{\ldots, -2, -1, 0, 1, 2, 3, \ldots\}$
$[0 : k]$	Intervall der Zahlen von 0 bis k
$a \in A$	a ist Element der Menge A
$a \notin A$	a ist kein Element der Menge A
$A = \{a_0, a_1, \ldots, a_n\}$	Menge A mit $n + 1$ Elementen
$A \cup B$	Vereinigung von A und B $= \{a \mid a \in A \text{ oder } a \in B\}$
$A \cap B$	Durchschnitt von A und B $= \{a \mid a \in A \text{ und } a \in B\}$
$A - B$	Differenz von A und B $= \{a \mid a \in A \text{ und } a \notin B\}$
$A \subset B$	A ist Teilmenge von B $a \in A \Longrightarrow a \in B$ z. B. für $x, y \in \mathbb{Z}$ gilt $[x : y] \subset \mathbb{Z}$
$\wp(A)$	Potenzmenge von A $= \{B \mid B \subset A\}$
$\emptyset$	leere Menge z. B. für $x > y$ gilt $[x : y] = \emptyset$
$\exists, \nexists$	Existenzquantor (es gibt, es gibt kein) z. B. $\nexists a : a \in \emptyset$
$\forall$	Allquantor (für alle) z. B. $\forall A : A \in \wp(A)$ oder $\forall A : \emptyset \subset A$
$A \times B$	kartesisches Produkt $= \{(a, b) \mid a \in A \text{ und } b \in B\}$

Der Mengenbegriff und die Notationen bergen ihre Schwierigkeiten (weshalb schon Generationen von Mathematikern ausgestorben sind).



Einige Mengen

$$A = \{a \mid a \text{ ist Vielfaches von 2 und } a \leq 12\}$$

$$B = \{b | b \text{ ist Vielfaches von 3 und } b \leq 15\}$$

$$A \cap B = \{6, 12\}$$

$$\wp(A \cap B) = \{\emptyset, \{6\}, \{12\}, \{6, 12\}\}$$

einige Eigenschaften von Mengen:

$$|M \cup N| \leq |M| + |N|$$

$$|M \times N| = |M| \cdot |N|$$

Für eine Menge M und Zahlen $n \in \mathbb{N}$ definieren wir induktiv das *n-fache kartesische Produkt* von M:

$$\begin{aligned} M^1 &= M \\ M^{n+1} &= M \times M^n \end{aligned}$$

Beispiel:

$$\begin{aligned} \{0,1\}^3 &= \{0,1\} \times \{0,1\}^2 \\ &= \{0,1\} \times \{(0,0),(0,1),(1,0),(1,1)\} \\ &= \{(0,(0,0)),(0,(0,1)),(0,(1,0)),(0,(1,1)), \\ &\quad (1,(0,0)),(1,(0,1)),(1,(1,0)),(1,(1,1))\} \end{aligned}$$

Die inneren Klammern lässt man weg: statt $(1,(0,1))$ schreibt man (1,0,1). Diese Vereinfachung ist problematisch (Vorsicht vor dem Aussterben!).

2.2 Induktionsbeweis

Es gilt: $|M^n| = |M|^n$ also z. B. $|\{0,1\}^n| = 2^n$

Beweis (durch vollständige Induktion über n):

$n = 1$: $|M^1| = |M| = |M|^1$

$n - 1 \to n$:

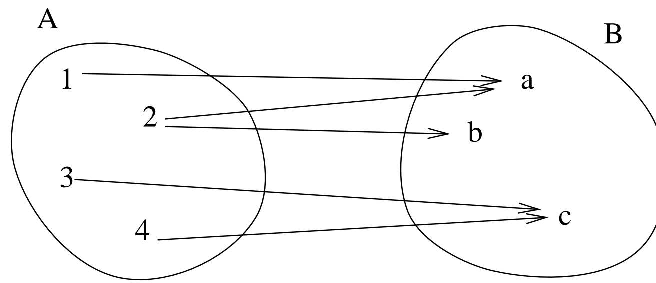
Induktionsvoraussetzung: $|M^{n-1}| = |M|^{n-1}$

Induktionsschritt: $|M^n| = |M \times M^{n-1}| = |M| \cdot |M^{n-1}| = |M| \cdot |M|^{n-1} = |M|^n$

2.3 Funktionen

Ist $R \subset A \times B$, dann heißt $f = (A, B, R)$ *Relation* von A in B.

Gilt für eine Relation $f = (A, B, R)$, dass für alle $a \in A$ ein $b \in B$ existiert mit $(a, b) \in R$ (d. h. $\forall a \in A \exists b \in B : (a, b) \in R$), dann heißt f *total*, ansonsten *partiell*.



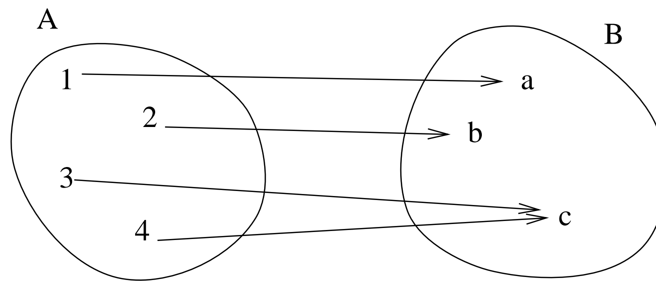
Totale Relation

$$R = \{(1, a), (2, a), (2, b), (3, c), (4, c)\}$$

Ist $f = (R, A, B)$ eine totale Relation und gilt mit $(a, b) \in R$ und $(a, b') \in R$, dass $b = b'$, dann heißt f *Abbildung* oder Funktion von A nach B.

Schreibweise: $f : A \longrightarrow B; f(a) = b$.

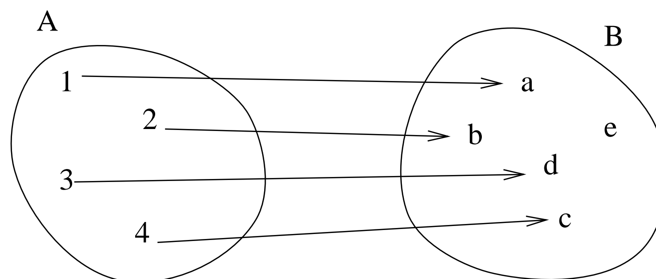
A heißt auch *Quelle* oder *Definitionsbereich* von f, B heißt *Ziel* oder *Wertebereich* von f.



Funktion

d. h. In allen Elementen in A (der linken Menge) beginnt genau ein Pfeil.

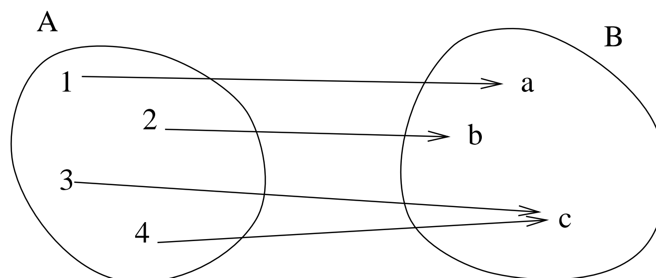
$f : A \longrightarrow B$ heißt *injektiv* (eindeutig), wenn für zwei beliebige $a, b \in A$ gilt: $f(a) = f(b) \Longrightarrow a = b$.



Injektive Funktion

d. h. in den Elementen in B (der rechten Menge) endet höchsten ein Pfeil.

$f : A \longrightarrow B$ heißt *surjektiv*, wenn für alle $b \in B$ gilt: $\exists a \in A : f(a) = b$.

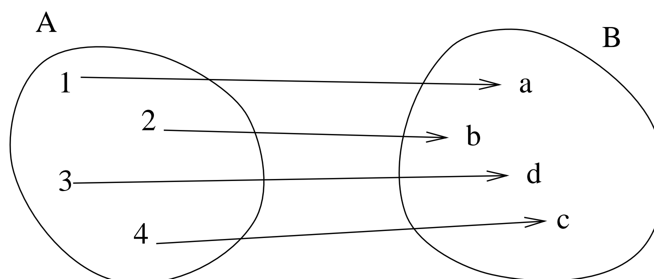


Surjektive Funktion

d. h. in den Elementen in B (der rechten Menge) endet mindestens ein Pfeil.

f heißt *bijektiv*, wenn f sowohl injektiv als auch surjektiv ist.

Ist f bijektiv, dann ist f umkehrbar, d. h. definieren wir für $f = (A, B, R)$ die Relation $R^{-1} = \{(b, a)|(a, b) \in R\}$, dann ist $f^{-1} = (B, A, R^{-1})$ ebenfalls eine Abbildung. f^{-1} heißt die zu f *inverse Abbildung*.

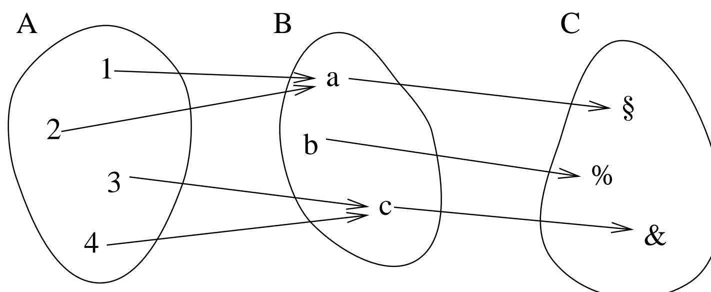


Bijektive also umkehrbare Funktion

d. h. in jedem Element in A beginnt genau ein Pfeil und in jedem Element in B endet genau ein Pfeil.

Man auch partielle Funktionen definieren.

Sind $f : A \longrightarrow B$ und $g : B \longrightarrow C$ Abbildungen, so kann man diese hintereinander ausführen: $g \circ f : A \longrightarrow C; (g \circ f)(a) = g(f(a)) = c$. ($g \circ f$ liest sich: g nach f.)



Hintereinanderausführen von Funktionen

Betrachten wir einige spezielle Abbildungen. Sei M eine Menge.

Eine Abbildung $f : M \times M \longrightarrow M$ heißt *kommutativ*, wenn für alle $a, b \in M$ gilt: $f(a, b) = f(b, a)$.

Beispiel: die Addition der natürlichen Zahlen.
Gegenbeispiel: die Subtraktion der ganzen Zahlen.

Eine Abbildung $f : M \times M \longrightarrow M$ heißt *assoziativ*, wenn für alle $a, b, c \in M$ gilt: $f(a, f(b, c)) = f(f(a, b), c)$.

Beispiel: die Addition der natürlichen Zahlen.
Gegenbeispiel: die Subtraktion der ganzen Zahlen.

2.4 Wichtige Rechenregeln

2.4.1 Rechnen mit Potenzen und Logarithmen

Seien im Folgenden a, b und c positive reelle Zahlen, sowie x und y reelle Zahlen. Wir wiederholen die folgenden Rechenregeln für Potenzen und Logarithmen, die wir in späteren Abschnitten benötigen:

$$\begin{aligned} b^0 &= 1 \\ b^x \cdot b^y &= b^{x+y} \\ 1/b^x &= b^{-x} \\ b^x/b^y &= b^{x-y} \end{aligned}$$

$$
\begin{aligned}
(b^x)^y &= b^{x \cdot y} \\
b^x = y &\Longrightarrow x = \log_b y \\
& \quad \text{falls } b \neq 1 \text{ und } b \neq 0 \\
\log_b(a \cdot c) &= \log_b a + \log_b c \\
\log_b(a/c) &= \log_b a - \log_b c \\
\log_b(c^x) &= x \cdot \log_b c \\
\log_b 1 &= 0 \\
\log_b b &= 1 \\
c &= b^{\log_b c} \\
a^{\log_b c} &= c^{\log_b a}
\end{aligned}
$$

Letztere Gleichung gilt wegen:

$$a^{\log_b c} = \left(b^{\log_b a}\right)^{\log_b c} = b^{\log_b a \cdot \log_b c} = b^{\log_b c \cdot \log_b a} = \left(b^{\log_b c}\right)^{\log_b a} = c^{\log_b a}$$

Wir schreiben im Folgenden stets einfach $\log$, wenn wir den Logarithmus zur Basis 2 meinen, in allen anderen Fällen geben wir die Basis explizit an. Damit ergeben sich folgende Identitäten:

$$
\begin{aligned}
2^{\log n} &= n \\
2^{\log n - 1} &= \frac{n}{2}
\end{aligned}
$$

2.4.2 Rechnen mit Summen

Wir benutzen das Zeichen $\sum$ zum Abkürzen:

$$x^0 + x^1 + x^2 + x^3 + \ldots + x^k = \sum_{i=0}^{k} x^i$$

Damit ist insbesondere

$$x^0 = \sum_{i=0}^{0} x^i$$

Ferner kann man Summen natürlich aufspalten:

$$x^0 + x^1 + x^2 + x^3 + \ldots + x^k = \sum_{i=0}^{j} x^i + \sum_{i=j+1}^{k} x^i$$

Aber auch wie folgt:

$$\sum_{i=0}^{k} (x^i + y^i) = \sum_{i=0}^{k} x^i + \sum_{i=0}^{k} y^i$$

Und wir definieren eine „Schreibweise für die Null“:

$$0 = \sum_{i=0}^{-1} x^i$$

d. h. wenn Endwert des Indexes kleiner als der Startwert ist, bezeichnet die Summe den Wert 0.

2.4.3 Geometrische Reihe

Die bereits oben notierte Summe

$$\sum_{j=0}^{k-1} x^j = x^0 + x^1 + x^2 + \ldots + x^{k-1}$$

heißt *geometrische Reihe*.
Für irgendein $x \in \mathbb{N}$ ungleich 0 und 1 gilt:

$$\sum_{j=0}^{k-1} x^j = \frac{x^k - 1}{x - 1}$$

Wir wollen das „einfach“ beweisen:
Sei hierzu

$$s = \sum_{j=0}^{k-1} x^j = x^0 + x^1 + x^2 + \ldots + x^{k-1}$$

damit haben wir

$$\begin{aligned} x \cdot s &= x \cdot \sum_{j=0}^{k-1} x^j = x \cdot (x^0 + x^1 + x^2 + \ldots + x^{k-1}) \\ &= \sum_{j=1}^{k} x^j = x^1 + x^2 + x^3 + \ldots + x^k \end{aligned}$$

aber auch

$$\begin{aligned} (x-1) \cdot s &= x \cdot s - s \\ &= \sum_{j=1}^{k} x^j - \sum_{j=0}^{k-1} x^j \\ &= (x^1 + x^2 + x^3 + \ldots + x^k) - (x^0 + x^1 + x^2 + \ldots + x^{k-1}) \\ &= x^1 + x^2 + x^3 + \ldots + \underbrace{x^k - x^0}_{\text{bleibt nur übrig}} - x^1 - x^2 - \ldots - x^{k-1} \\ &= x^k - x^0 \\ &= x^k - 1 \end{aligned}$$

Also haben wir für $x \neq 1$

$$s = \frac{x^k - 1}{x - 1}$$

was zu beweisen war.
Bemerkungen: Man kann die Gleichung der geometrischen Reihe auch mit Induktion beweisen. (Die Formel gilt nicht nur für natürliche Zahlen sondern für alle reellen Zahlen.)
Insbesondere ergibt sich für $x = 2$:

$$\sum_{i=0}^{k-1} 2^i = \frac{2^k - 1}{2 - 1} = 2^k - 1$$

2.4.4 Rekursionsgleichungen

Wir lernen hier zwei einfache Rekursionsgleichungen kennen, die uns später sehr nützlich sein werden. Die Idee dahinter ist recht einfach zu verstehen (hier sehr informal): Gegeben ein Problem der Großenordnung n (was immer das bedeuten mag), kennt man die Lösung des Problems kleinerer Ordnung sowie eine Konstruktion, wie man mithilfe dieser kleineren Lösung, die Lösung für das große Problem erhält, so kann man den Aufwand mithilfe der Rekursionsgleichung berechnen. Das kleinere Problem kann einfach von der Größenordnung $n-1$ sein, oder aber lediglich von der Größenordnung n/b sein.

Betrachten wir die beiden Fälle:

Sei $f : \mathbb{N} \longrightarrow \mathbb{N}$ eine Funktion mit:

$$
\begin{aligned}
f(1) &= c \qquad \text{für irgendein } c \in \mathbb{N} \\
f(n) &= a \cdot f(n-1) + g(n)
\end{aligned}
$$

Dann gilt:

$$f(n) = a^{n-1} \cdot c + \sum_{i=0}^{n-2} a^i \cdot g(n-i)$$

Wir beweisen dies durch Induktion über n.

Induktionsanfang:

$n = 1$

Wir berechnen $f(1)$ gemäß der Formel in der Behauptung:

$$
\begin{aligned}
f(1) &= a^{1-1} \cdot c + \sum_{i=0}^{1-2} a^i \cdot g(n-i) \\
&= a^0 \cdot c + \underbrace{\sum_{i=0}^{-1} a^i \cdot g(n-i)}_{=0} \\
&= c
\end{aligned}
$$

und somit gilt der Induktionsanfang.

Wir führen den Induktionsschritt von n nach $n+1$.

Induktionsvoraussetzung:

$$f(n) = a^{n-1} \cdot c + \sum_{i=0}^{n-2} a^i \cdot g(n-i)$$

Es ist zu zeigen:

$$f(n+1) = a^n \cdot c + \sum_{i=0}^{n-1} a^i \cdot g(n-i)$$

und dies ergibt sich wie folgt:

$$
\begin{aligned}
f(n+1) &= a \cdot f(n) + g(n+1) \\
&= a \cdot (a^{n-1} \cdot c + \sum_{i=0}^{n-2} a^i \cdot g(n-i)) + g(n+1) \\
&= a^n \cdot c + a \cdot \sum_{i=0}^{n-2} a^i \cdot g(n-i)) + 1 \cdot g(n+1) \\
&= a^n \cdot c + \sum_{i=0}^{n-2} a^{i+1} \cdot g(n-i)) + a^0 \cdot g(n+1) \\
&= a^n \cdot c + \sum_{i=1}^{n-1} a^i \cdot g(n-(i-1)) + a^0 \cdot g(n+1) \\
&= a^n \cdot c + \sum_{i=0}^{n-1} a^i \cdot g(n+1-i)
\end{aligned}
$$

was zu beweisen war.

Und jetzt der zweite Fall:

Sei $f : \mathbb{N} \longrightarrow \mathbb{N}$ eine Funktion mit:

$$
\begin{aligned}
f(1) &= c \qquad \text{für irgendein } c \in \mathbb{N} \\
f(n) &= a \cdot f\left(\frac{n}{b}\right) + g(n)
\end{aligned}
$$

für alle Potenzen $n = b^k$ (d. h. uns interessiert die Funktion f nur an den Stellen $b_0 = 1, b, b^2, b^3, \ldots$).

Dann gilt:

$$
f(n) = a^{\log_b n} \cdot c + \sum_{i=0}^{\log_b n - 1} a^i \cdot g\left(\frac{n}{b^i}\right)
$$

Wir beweisen dies durch Induktion über die Potenzen b^k.

Induktionsanfang:

$k = 0$, d. h. $n = b^k = b^0 = 1$:

Wir berechnen $f(1)$ gemäß der Formel in der Behauptung:

$$
\begin{aligned}
f(1) &= a^{\log_b 1} \cdot c + \sum_{i=0}^{\log_b 1 - 1} a^i \cdot g\left(\frac{1}{b^i}\right) \\
&= a^0 \cdot c + \underbrace{\sum_{i=0}^{0-1} a^i \cdot g\left(\frac{1}{b^i}\right)}_{=0} \\
&= c
\end{aligned}
$$

und somit gilt der Induktionsanfang.

Wir führen den Induktionsschritt von k nach $k+1$, also gemäß den Potenzen von b.

Induktionsvoraussetzung:

$$
\begin{aligned}
f(b^k) &= a^{\log_b b^k} \cdot c + \sum_{i=0}^{\log_b b^k - 1} a^i \cdot g\left(\frac{b^k}{b^i}\right) \\
&= a^k \cdot c + \sum_{i=0}^{k-1} a^i \cdot g(b^{k-i})
\end{aligned}
$$

Es ist zu zeigen:

$$
f(b^{k+1}) = a^{k+1} \cdot c + \sum_{i=0}^{k} a^i \cdot g(b^{k+1-i})
$$

und dies ergibt sich wie folgt:

$$
\begin{aligned}
f(b^{k+1}) &= a \cdot f\left(\frac{b^{k+1}}{b}\right) + g(b^{k+1}) \\
&= a \cdot f(b^k) + g(b^{k+1}) \\
&= a \cdot \left(a^k \cdot c + \sum_{i=0}^{k-1} a^i \cdot g(b^{k-i})\right) + g(b^{k+1}) \\
&= a \cdot a^k \cdot c + a \cdot \sum_{i=0}^{k-1} a^i \cdot g(b^{k-i})) + 1 \cdot g(b^{k+1}) \\
&= a^{k+1} \cdot c + \sum_{i=0}^{k-1} a^{i+1} \cdot g(b^{k-i})) + a^0 \cdot g(b^{k+1}) \\
&= a^{k+1} \cdot c + \sum_{i=1}^{k} a^i \cdot g(b^{k-(i-1)})) + a^0 \cdot g(b^{k+1}) \\
&= a^{k+1} \cdot c + \sum_{i=0}^{k} a^i \cdot g(b^{k+1-i})
\end{aligned}
$$

2.5 Zahlendarstellungen

2.5.1 Unärdarstellung

Ein *Zeichenvorrat* oder ein *Alphabet* A ist eine endliche Menge von Zeichen. Wir bezeichnen mit A^+ die Menge aller endlicher Zeichenreihen, die nur aus Zeichen aus A bestehen. Die *Länge* $l(w)$ einer Zeichenreihe $w \in A^+$ ist die Anzahl der Zeichen, aus denen w besteht.

Für $n \in \mathbb{N}$ schreiben wir

$$
A^n = \{w \in A^+ | l(w) = n\}
$$

und es gilt offensichtlich

$$
A^+ = A^1 \cup A^2 \cup A^3 \cup ... = \bigcup_{i \in \mathbb{N}} A^i
$$

damit haben wir auch eine *unäre Zahlendarstellung* (wir wählen als Alphabet die Menge $\{|\}$ oder eben einen Stein der Ägypter).

Es gibt in dieser Zahlendarstellung noch keine Null. Wir definieren uns deshalb ein Symbol für eine Zeichenreihe der Länge Null: ε. (Dies ist kein Witz, sondern ein eindeutig bestimmtes Zeichen, was wir schreiben, wenn wir *kein* Wort schreiben wollen.)

Wir definieren weiter

$$A^0 = \{\varepsilon\}$$

$$A^* = A^+ \cup A^0$$

Somit haben wir die unäre Zahlendarstellung um eine Null erweitert.

Die unäre Zahlendarstellung ist recht aufwendig, da „gebräuchliche" Zahlen recht schnell sehr lang werden. Stellenwertsysteme eignen sich da wesentlich besser.

2.5.2 Zahlendarstellung zu einer Basis

Dezimalzahlen schreiben wir üblicherweise dadurch, dass wir sogenannte Ziffern hintereinander schreiben und dann entsprechend interpretieren.

Also: Sei A ein Alphabet (sogenannte Ziffern) mit $|A| > 1$. Wir ordnen jeder Ziffer mithilfe einer bijektiven Funktion einen Wert als natürliche Zahl aus einem Intervall zu:

$$\varphi : A \longrightarrow [0 : |A| - 1]$$

Der Wert von $|A|$ wird *Basis* der Zahlendarstellung genannt.

Beispiele für übliche Zuordnungen

Name	Basis	Alphabet (Ziffern)
Binärzahlen	2	0,1
Oktalzahlen	8	0,1,2,3,4,5,6,7
Dezimalzahlen	10	0,1,2,3,4,5,6,7,8,9
Hexadezimalzahlen	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Die Abbildung φ ordnet den Ziffern die Werte im Intervall $[0 : |A| - 1]$ entsprechend der Reihenfolge in der Tabelle zu, insbesondere ist z. B. $\varphi(A) = 10$ und $\varphi(E) = 14$.

Sei $n \in \mathbb{N}$. Jede Zeichenreihe (Ziffernwort) $a = a_{n-1} \dots a_0 \in A^n$ ist eine n-stellige Zahlendarstellung zur Basis $|A|$. Wir definieren die durch a dargestellte Zahl als

$$\langle a \rangle_{|A|} = \sum_{i=0}^{n-1} \varphi(a_i) \cdot |A|^i$$

Damit das ganze funktioniert, muss eine Zahl *eindeutig* in Ziffern zerlegbar sein. Ferner ist die vorgestellte Zahlendarstellung *eindeutig*, d. h. zu gegebener Basis (und gewählten Ziffern) hat jede natürliche Zahl genau eine Zahlendarstellung zu dieser Basis. Dies müsste man beweisen.

Beachte: wir müssen dazu schon wissen was Addieren und Multiplizieren ist, und was die abkürzende Schreibweise mit Exponenten bedeutet. Das wissen wir bereits (oder machen es wie die Ägypter mit Steinen).

Beispiel: $\langle 347 \rangle_{10} = 3 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0$

Natürlich schreiben wir in Zukunft statt $\langle a \rangle_{10}$ einfach a und unterscheiden nicht zwischen der Darstellung und der dargestellten Zahl. Das mit der 10 hat wohl nur den besonderen Grund, dass wir 10 Finger haben. (Man sieht: wir haben nicht nur die Schreibweise der Ziffern, sondern auch die Stelligkeit von rechts nach links aufsteigend von den Arabern übernommen!)

Beispiel: $\langle 101011 \rangle_2 = \langle 43 \rangle_{10}$

Wir schreiben weiterhin in Zukunft statt $\langle a \rangle_2$ einfach a; aus dem Kontext – und dem Nur-Vorhandensein von 0 und 1 – wird klar sein, dass es sich um eine Binärdarstellung und nicht um ein Dezimaldarstellung handelt.

Die Länge einer Zahl ist in dieser Darstellung erheblich kürzer als in der Unärdarstellung. Für eine Zahl $n \in \mathbb{N}$ gilt:
n in Unärdarstellung als w: $l(w) = n$,
n in Binärdarstellung als $\langle a \rangle_b$: $l(a) = \lceil \log_b(n+1) \rceil$,
wobei wir mit $\lceil r \rceil$ die nächstgrößere ganze Zahl zu r meinen bzw. gerade r selbst, wenn r bereits eine ganza Zahl ist.
$l(a)$ nennt man dann auch Stellenzahl von a.

Um eine Zahl bis zur Größe $n - 1$ darstellen zu können, sind $\lceil \log n \rceil$ viele Stellen erforderlich.

Bemerkung: in vielen Programmiersprachen ist die Zahlendarstellung zu anderen Basen als 10 ebenfalls vorhanden! (in **C** schreibt man beispielsweise Hexadezimalzahlen als `0x39A7B` und Oktalzahlen sind durch führende Null und *Nicht-Auftreten* der Ziffern 8 und 9 gekennzeichnet; dies kann leicht zu Verwirrung führen!)

2.5.3 Darstellung negativer Zahlen

Neben positiven Zahlen wollen wir auch negative Zahlen darstellen. Wir beschränken uns hier auf die Binärzahlen!

Die erste Möglichkeit wäre analog zur Schulmethode, wir führen ein Vorzeichenbit ein und stellen wir folgt dar:
Sei $n \in \mathbb{N}$, $A = \{0, 1\}$. Jede Zeichenreihe (Ziffernwort) $c = c_n \dots c_0 \in A^{n+1}$ ist eine $n+1$-stellige Binärdarstellung mit *Vorzeichen*. Wir definieren die durch c dargestellte Zahl z als

$$z = \begin{cases} \langle c_{n-1} \dots c_0 \rangle & \text{falls} \quad c_n = 0 \\ -\langle c_{n-1} \dots c_0 \rangle & \text{falls} \quad c_n = 1 \end{cases}$$

Das heißt, ist das führende Bit gleich 1, so handelt es sich um eine *negative* Zahl und die restlichen Stellen geben deren Betrag an; ist das führende Bit gleich 0, so handelt es sich um eine *positve* Zahl und die restlichen Stellen geben deren Betrag an.
Die Darstellung heißt *Zahlendarstellung mit Betrag und Vorzeichen*. Man beachte, es gibt *zwei* Darstellungen für die Null.

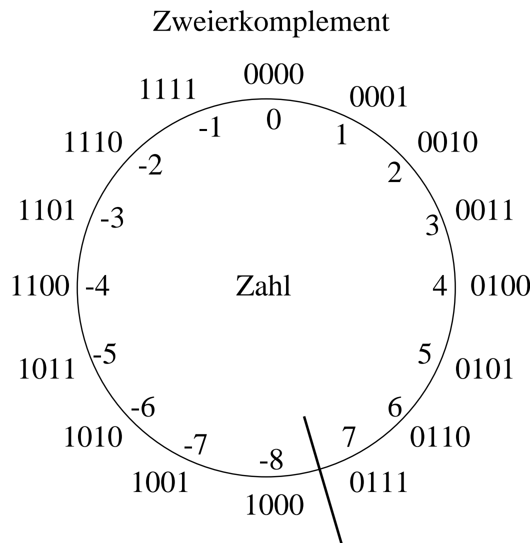
Wir werden später bei der Beschreibung der arithmetisch-logischen Einheiten sehen, dass die folgende alternative Zahlendarstellung sehr nützlich ist.
Sei $n \in \mathbb{N}$, $A = \{0, 1\}$. Jede Zeichenreihe (Ziffernwort) $c = c_n \dots c_0 \in A^{n+1}$ ist eine $n+1$-stellige *Zweier-Komplement-Darstellung*. Wir definieren die durch c dargestellte Zahl $[c_n \dots c_0]$ als

$$[c_n \dots c_0] = -c_n \cdot 2^n + \langle c_{n-1} \dots c_0 \rangle$$

Beispiel: 7–Bit Zahl im Zweierkomplement

$$\begin{aligned} [1010111] &= \underbrace{-1 \cdot 2^6}_{=-64} + \underbrace{\langle 010111 \rangle}_{=23} \\ &= -41 \end{aligned}$$

Damit können wir Zahlen im Bereich $\{-2^n, \dots, 2^n - 1\}$ darstellen. Folgende Abbildung veranschaulicht die Definition.



Veranschaulichung des Zweierkomplements

Die Null ist nun eindeutig darstellbar. Wir schreiben Zahlen im Zweierkomplement mit eckigen Klammern. Eine Zahl $[c]$ ist *genau dann* größer gleich Null, wenn das führende Bit c_n gleich Null ist. Man nennt deshalb dieses Bit ebenfalls Vorzeichenbit; beachte jedoch den Unterschied zu oben, es folgt *nicht* der Betrag!

Schreiben wir im Folgenden für ein $c_i \in \{0,1\}$: $\overline{c_i} = 1 - c_i$ und entsprechend $[\overline{c}] = [\overline{c_n} \ldots \overline{c_0}]$. Damit hat die Zweierkomplement-Darstellung nun folgende sehr nützliche Eigenschaft:

$$-[c] = [\overline{c}] + 1$$

Wollen wir dies beweisen. Wir zeigen $[\overline{c}] = -[c] - 1$.

$$\begin{aligned}
[\overline{c}] &= [\overline{c_n} \ldots \overline{c_0}] \\
&= -\overline{c_n} \cdot 2^n + \langle \overline{c_{n-1}} \ldots \overline{c_0} \rangle \\
&= -\overline{c_n} \cdot 2^n + \sum_{i=0}^{n-1} \overline{c_i} \cdot 2^i \\
&= -(1 - c_n) \cdot 2^n + \sum_{i=0}^{n-1} (1 - c_i) \cdot 2^i \\
&= -2^n + c_n \cdot 2^n + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} c_i \cdot 2^i \\
&= -2^n + c_n \cdot 2^n + 2^n - 1 - \langle c_{n-1} \ldots c_0 \rangle \\
&= -[c_n \ldots c_0] - 1
\end{aligned}$$

Wir erhalten zu einer Zahl z die negative Zahl $-z$, indem wir die Zweierkomplementdarstellung von z bitweise invertieren und dann 1 hinzu addieren. Dies wird auch in obiger Abbildung klar: man geht horizontal von links nach rechts oder umgekehrt und „sieht“ die Rechenregel!

3 Schaltkreistheorie

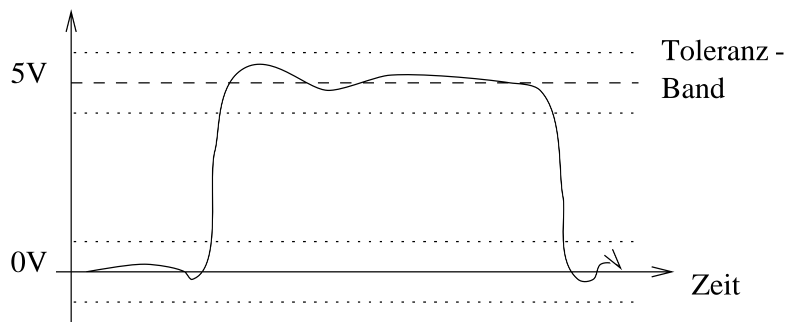
Motivation: aus einfachen Dingen, komplizierte Dinge herstellen.

3.1 Hardware-Ebenen und „Digitalität“

Rechner		Das Ziel
Speicher Prozessor		programmierbare Einheit
Schaltwerk		endliche Automaten
	Register	Zustandsvektoren
Schaltnetz (Schaltkreis)		Logik, Schalt„algebra“
	Flip-Flop	ein Bit
Gatter		Elementarfunktionen
Transistor		Kennlinien, Verstärker
Elektrotechnik		Ladung, Spannung, Strom
Physik		Gesetze

Von unten nach oben nimmt die Abstraktion zu.

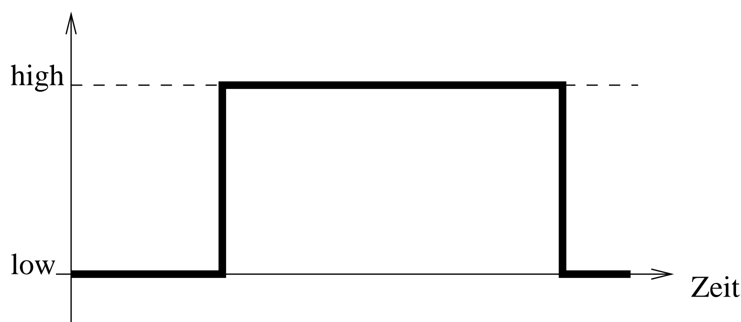
Digitale Größen: es gibt nur Null und Eins und die *Zeit*. Null und Eins muss interpretiert und realisiert werden. Heute meist durch elektische Spannung, wie im Spannungs-Zeit-Diagramm verdeutlicht:



Elektrisches Signal

Man unterscheidet (und realisiert) zwei Zustände (in einer Technologie z. B. 0V und 5V, aber auch: magnetisiert und nicht-magnetisiert, gelocht und nicht-gelocht, vertikal polarisiert und horizontal polarisiert).

Damit das halbwegs funktioniert, darf man das nicht so eng sehen, sondern man nutzt Toleranzbänder. Somit idealisiert sich das elektrische Signal zu:



Idealisiertes elektrisches Signal

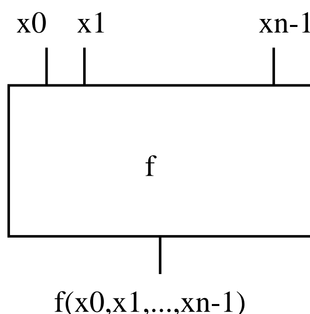
Dieses muss interpretiert werden. Hier ist man frei und nutzt „positive“ und „negative Logik“, d. h. man interpretiert, was der eine oder andere Zustand bedeuten soll.

Entweder *high* oder *low* wird als logische Null oder Eins interpretiert (*high*$= 1$: positive Logik, ‘active high logic’); *high*$= 0$: negative Logik, ‘active low logic’).

Im oberen Bild interessieren wir uns für recht komplexe Gebilde, die aber im Innern nur aus „Elementen“ aufgebaut sind, die zwei Zustände „verkraften“. Wir beginnen weit unten.

3.2 Schaltfunktionen

Sei $n \in \mathbb{N}$. Wir interessieren uns (vorerst) für Schaltnetze mit nur einem Ausgang und n Eingängen $X_0, \ldots, X_{n-1}$. Jeder Eingang und auch der Ausgang soll nur einen der beiden Zustände annehmen können. Der Ausgang soll *allein durch die Eingangsbelegung* festgelegt sein (Auch dieses präzisieren wir später).



Ein Schaltnetz als Kasten mit n Eingängen und einem Ausgang

Sei $n \in \mathbb{N}$. Eine n-stellige *Schaltfunktion* ist eine Abbildung $f : \{0,1\}^n \longrightarrow \{0,1\}$.

Spezielle Schaltfunktionen:

Konjunktion $\wedge : \{0,1\}^2 \longrightarrow \{0,1\}$

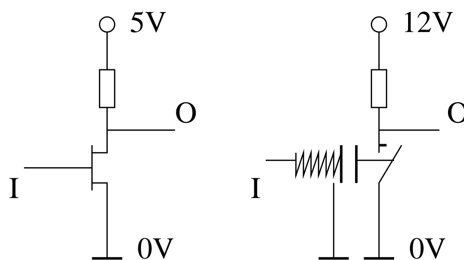
Disjunktion $\vee : \{0,1\}^2 \longrightarrow \{0,1\}$

Negation $\sim: \{0,1\} \longrightarrow \{0,1\}$

Wertetabellen der Funktionen:

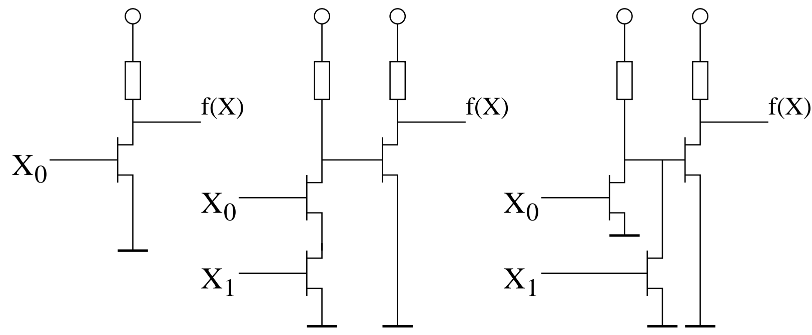
x_0	x_1	$x_0 \wedge x_1$	$x_0 \vee x_1$	$\sim x_0$
0	0	0	0	1
0	1	0	1	
1	0	0	1	0
1	1	1	1	

Realisierung der speziellen Schaltfunktionen:



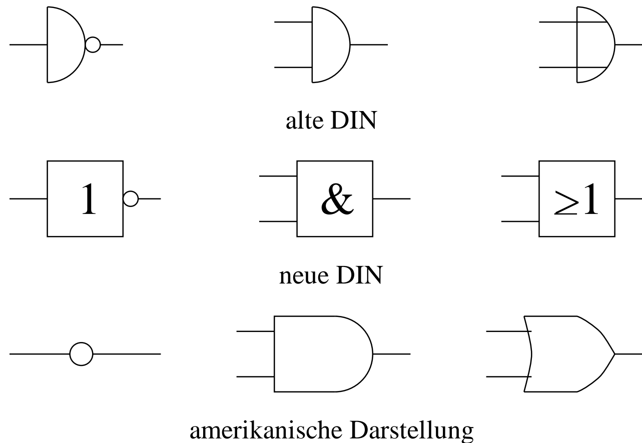
Transistor als Inverter, (auch Relais)

Damit folgende Darstellungen:



NOT-Gate AND-Gate OR-Gate

Prinzip-Schaltung der speziellen Schaltfunktionen



Das Zeichnen der Gatter verbirgt die eigentliche elektronische Schaltung. Die Bausteine sind auf Chips realisiert (hierzu später mehr). Wir abtrahieren vorerst von einer Signallaufzeit in den Gattern. Dies ändert sich aber später!

3.3 BOOLE'sche Ausdrücke

Man kann obige Schaltfunktionen auch wie folgt schreiben, wobei X_0 und X_1 Variablennamen sind, die Werte aus $B = \{0, 1\}$ annehmen können:

BOOLE'sche Funktion:

$$NOT(X_0) \qquad AND(X_0, X_1) \qquad OR(X_0, X_1)$$

BOOLE'scher Ausdruck mit Variablen (zwei Varianten)

$$\sim X_0 \qquad X_0 \wedge X_1 \qquad X_0 \vee X_1$$

$$\overline{X_0} \qquad X_0 \cdot X_1 \qquad X_0 + X_1$$

Auch der Punkt wird analog zur „gewohnten Buchstabenrechnung“ häufig weggelassen ($X_0 \cdot X_1$ gleich $X_0 X_1$).

Nun wollen wir aber komplexere Funktionen durch Schaltnetze realisieren und mit ihnen „rumrechnen“ (eventuell beweisen, denn wir wollen, dass unsere Schaltungen korrekt sind). Deshalb:

Fassen wir dies formaler.

Wir müssen uns zunächst auf eine Schreibweise einigen (genauso wie die Ägypter auf die Striche oder Steine ihrer Zahlen).

Sei $A = \{0, 1, \wedge, \vee, \sim, X_0, \ldots, X_{n-1}, (,)\}$ ein Alphabet (für ein festes $n \in \mathbb{N}$). Wir nennen $V = \{X_0, \ldots, X_{n-1}\} \subset A$ Menge der Variablen. Wir schreiben auch $X = (X_0, \ldots, X_{n-1})$ für einen Vektor von Variablen.

Die Menge der $\mathcal{B}$ der *vollständig geklammerten* BOOLE*'schen Ausdrücke* ist eine Menge von Zeichenreihen in A^+ (also $\mathcal{B} \subset A^+$) mit folgenden induktiv definierten Eigenschaften:

Es ist $\mathcal{B}_1 = \{0, 1, X_0, \ldots, X_{n-1}\}$. Sei $i \in \mathbb{N}$ und seien a, b zwei beliebige Zeichenreihen aus $\mathcal{B}_i$, dann liegen folgende Zeichenreihen in $\mathcal{B}_{i+1}$:

1. a
2. $(\sim a)$
3. $(a \wedge b)$
4. $(a \vee b)$

Eine Zeichenreihe $z \in A^+$ liegt genau dann in $\mathcal{B}$, wenn z in einer der Mengen $\mathcal{B}_i$ liegt.

Schreiben wir $\cdot$ für $\wedge$ und $+$ für $\vee$, dann sind vollständig geklammerte Ausdrücke nichts anderes, als das, was wir gewohnt sind zu schreiben, wenn wir alle Klammern in Ausdrücken mit $\cdot$ und $+$ „richtig“ setzen.

Beispiel: $((0 \vee (\sim 1)) \wedge (X_2 \vee (\sim (\sim X_{52})))) \in \mathcal{B}_5$

Diese Definition findet sich (so oder so ähnlich) in vielen Büchern, hat allerdings den Nachteil, dass nur die speziellen Funktionen auftauchen, nicht jedoch komplizierte Funktionen „aufgerufen“ werden können
(wie z. B. $(X_1 \wedge f(X_2, 0, (X_1 \vee X_3))))$.

Deshalb: Definition von erweiterten Ausdrücken.

Wir vereinbaren Namen für Funktionen.

Sei $F = \{f_0, f_1, \ldots\}$ eine Menge von Funktionsnamen (F sei abzählbar, d. h. wir können einen Algorithmus angeben, der F aufzählt, z. B. lexikographische Anordnung von Worten aus dem üblichen Alphabet).

Wir ordnen jeder Funktion f mithilfe der Funktion $s : F \longrightarrow \mathbb{N}_0$ eine Stelligkeit zu. Sie gibt einfach an, wie viele Argumente f benötigt.

z. B.: $s(NICHT) = 1, s(UND) = 2, s(ODER) = 2$.

Damit erweitern wir das Alphabet:

$$A = V \cup F \cup \{0, 1, \wedge, \vee, \sim, (,), ,\}$$

Beachte: auch das Komma ist jetzt im Alphabet enthalten! Die Funktionsnamen zählen wie ein einzelnes Zeichen!

Die Menge $\mathcal{C}$ der *erweiterten* BOOLE*'schen Ausdrücke* ist eine Menge von Zeichenreihen in A^+ (also $\mathcal{C} \subset A^+$) mit folgenden induktiv definierten Eigenschaften:

Es ist $\mathcal{C}_1 = \{0, 1, X_0, \ldots, X_{n-1}\}$. Sei $i \in \mathbb{N}$ und seien $e_0, e_1, \ldots$ beliebige Zeichenreihen aus $\mathcal{C}_i$, dann liegen folgende Zeichenreihen in $\mathcal{C}_{i+1}$:

1. e_0
2. $(\sim e_0)$
3. $(e_0 \wedge e_1)$
4. $(e_0 \vee e_1)$
5. $f(e_0, \ldots, e_{s(f)-1})$ für alle $f \in F$

Eine Zeichenreihe $z \in A^+$ liegt genau dann in $\mathcal{C}$, wenn z in einer der Mengen $\mathcal{C}_i$ liegt.

Damit liegt $(X_1 \wedge f(X_2, 0, (X_1 \vee X_3)))$ in $\mathcal{C}_3$, wobei $s(f) = 3$ sein muss, denn:

$\in \mathcal{C}_1 : X_1, X_2, 0, X_3$
$\in \mathcal{C}_2 : (X_1 \vee X_3)$

$\in \mathcal{C}_3 : f(X_2, 0, (X_1 \vee X_3))$
$\in \mathcal{C}_4 : (X_1 \wedge f(X_2, 0, (X_1 \vee X_3)))$

Damit können wir jetzt BOOLE'sche Ausdrücke schreiben. Insbesondere ist mit den Definitionen klar, dass die Ausdrücke $NOT(X_0)$, $AND(X_0, X_1)$, $OR(X_0, X_1)$, aber auch der Ausdruck $AND(OR(X_1, X_{22}), NOT(X_5))$, richtig geschriebene Ausdrücke sind, allerdings haben wir bis jetzt nur die Syntax und es ist noch keineswegs klar, was mit solchem „Hingeschriebenen" gemeint ist.

Was machen wir mit diesen Ausdrücken?

Jeder BOOLE'scher Ausdruck $e \in \mathcal{B}$ kann als Vorschrift zur Berechnung einer Funktion $||e|| : \{0,1\}^n \longrightarrow \{0,1\}$ aufgefasst werden:

Für $a = (a_0, \ldots, a_{n-1}) \in \{0,1\}^n$ berechnet man den Wert der Funktion $||e||$ an der Stelle a, indem man für alle i die Konstante a_i für die Variable X_i einsetzt und dann auf die übliche Art auswertet.

Beispiel: $((0 \vee (\sim 1)) \wedge (X_2 \vee (\sim (\sim X_{52})))) \in \mathcal{B}_5$ für $a = (0, 1, \ldots, 1) \in \{0,1\}^{53}$

Auswerten!:

$$||((0 \vee (\sim 1)) \wedge (X_2 \vee (\sim (\sim X_{52}))))||$$

$$\underbrace{(\underbrace{(0 \vee \underbrace{(\sim 1)}_{0})}_{0} \wedge \underbrace{(0 \vee \underbrace{(\sim \underbrace{(\sim 1)}_{0})}_{1})}_{1})}_{0}$$

Also: $||e||(a) = 0$

Auch dieses mit dem bisherigen Werkzeug etwas genauer.

Eine *Einsetzung* ist eine Abbildung $\Phi : V \longrightarrow \{0,1\}$.

Damit ist $\Phi(X_i) \in \{0,1\}$ gerade die Konstante, die für die Variable X_i eingesetzt werden soll.

Durch eine Einsetzung Φ ist bereits für jeden BOOLE'schen Ausdruck e der Wert von e an der Stelle $(\Phi(X_0), \ldots, \Phi(X_{n-1}))$ festgelegt. Wir nennen diesen Wert $\Phi(e)$ und definieren ihn formal, indem wir induktiv die Funktion Φ von $V \subset \mathcal{C}$ auf die ganze Menge $\mathcal{C}$ *fortsetzen*.

Wir definieren:

$\Phi(0) = 0$, $\Phi(1) = 1$, damit haben wir bereits eine Abbildung $\Phi : \mathcal{C}_1 \longrightarrow \{0,1\}$.

Seien $e_0, e_1, \ldots \in \mathcal{C}$ erweiterte BOOLE'sche Ausdrücke. Wir definieren

1. $\Phi(\sim e_1) = \sim \Phi(e_1)$
2. $\Phi(e_0 \wedge e_1) = \Phi(e_0) \wedge \Phi(e_1)$
3. $\Phi(e_0 \vee e_1) = \Phi(e_0) \vee \Phi(e_1)$
4. $\Phi(f(e_0, \ldots, e_{s(f)-1})) = f(\Phi(e_0), \ldots, \Phi(e_{s(f)-1}))$ für alle $f \in F$

Bemerkung: Wir haben wieder äußere Klammern bei den Ausdrücken weggelassen. Links bedeuten die Operationssymbole $\vee, \wedge, \sim$ ein Zeichen aus dem Alphabet; rechts stehen sie für eine Aufforderung zur Funktionsauswertung. Die Regeln dazu stehen in obiger Tabelle. Für andere Funktionen muss man erst die Auswertungsvorschrift festlegen (z. B. durch Angabe einer Tabelle).

Dies ist nun etwas, was man einfach handhaben (und sogar programmieren kann). Schreiben wir deshalb wieder $\cdot$ für $\wedge$ und $+$ für $\vee$, da diese auf einer Tastatur zu finden sind. (Man kann statt $\cdot$ auch $*$ nehmen.)

Beispiel: $((X_0 \cdot X_1) + X_2)$ für $\Phi(X_0) = 1, \Phi(X_1) = 0, \Phi(X_2) = 1$.

Auswerten!:

$((X_0 \cdot X_1) + X_2)$
$((1 \cdot 0) + 1)$

$(0+1)$
1

Beispiel: Wertetabelle für eine 3-stellige Funktion f

a_0	a_1	a_2	$f(a_0, a_1, a_2)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Einsetzung für die Variablen wie oben: $\Phi(X_0) = 1, \Phi(X_1) = 0, \Phi(X_2) = 1$.

Erweiterter BOOLE'scher Ausdruck: $(X_0 \cdot f(X_1, 0, (X_0 + X_2)))$

Auswerten! (mit Überprüfung, dass korrekt „hingeschrieben“).

$(X_0 \cdot f(X_1, 0, (X_0 + X_2)))$
$(1 \cdot f(0, 0, (1+1)))$
$(1 \cdot f(0, 0, 1))$
$(1 \cdot 0)$
0

Man beachte, dass wir stets eindeutig auswerten können. Dies liegt im Wesentlichen daran, dass wir die Ausdrücke *eindeutig* zerlegen können.

Was haben wir bisher erreicht?

Wir können sehr genau (vollständig geklammerte) BOOLE'sche Ausdrücke hinschreiben. Wir können erweiterte BOOLE'sche Ausdrücke hinschreiben. Wir können eine Variablenbelegung einsetzen. Wir können den Ausdruck auswerten (sofern wir eine Auswertungsvorschrift für die Funktionen kennen).

Was wollen wir damit?

Herausfinden, ob verschiedene Ausdrücke gleich sind. Herausfinden, welche Funktion von einem Ausdrück berechnet wird. Ausdrücke so umformen, dass wir sie realisieren können. Herausfinden, was uns das Realisieren kostet.

3.4 Äquivalenz von BOOLE'schen Ausdrücken

Seien e_0, e_1 zwei erweiterte BOOLE'sche Ausdrücke. Es gilt $e_0 \equiv e_1$, d. h. die *Ausdrücke sind äquivalent*, genau dann wenn $\Phi(e_0) = \Phi(e_1)$ für alle Einsetzungen Φ gilt.

Warum schreiben wir hier $\equiv$ und nicht $=$? Nun, manchmal werden wir $=$ statt $\equiv$ schreiben, sobald wir den Unterschied verstanden haben. Das Gleichheitszeichen sagt, dass beide Seiten gleich sind und dies gilt *nicht* für die Zeichenketten, sondern nur für die Einsetzung. (Erinnern sie sich an die Ägypter: was würde ein solcher zu $154 =$ „dein Arbeitslohn“ sagen?)

Sei $e \in \mathcal{C}$ ein erweiterter BOOLE'scher Ausdruck. Dann gibt es genau eine n-stellige Schaltfunktion f, so dass $f(X) \equiv e$.

Die Funktion f mit $f(X) \equiv e$ heißt die durch e berechnete Funktion.

Damit können wir ein paar Identitäten, d. h. Gleichungen für Schaltfunktionen, hinschreiben:

(B1)	$(X_0 \wedge X_1) \equiv (X_1 \wedge X_0)$	Kommutativität
	$(X_0 \vee X_1) \equiv (X_1 \vee X_0)$	
(B2)	$((X_0 \wedge X_1) \wedge X_2) \equiv (X_0 \wedge (X_1 \wedge X_2))$	Assoziativität
	$((X_0 \vee X_1) \vee X_2) \equiv (X_0 \vee (X_1 \vee X_2))$	
(B3)	$(X_0 \wedge (X_1 \vee X_2)) \equiv ((X_0 \wedge X_1) \vee (X_0 \wedge X_2))$	Distributivität
	$(X_0 \vee (X_1 \wedge X_2)) \equiv ((X_0 \vee X_1) \wedge (X_0 \vee X_2))$	
(B4)	$(X_0 \wedge (X_0 \vee X_1)) \equiv X_0$	Absorption
	$(X_0 \vee (X_0 \wedge X_1)) \equiv X_0$	
(B5)	$(X_0 \wedge (X_1 \vee (\sim X_1))) \equiv X_0$	
	$(X_0 \vee (X_1 \wedge (\sim X_1))) \equiv X_0$	
(B6)	$(X_0 \wedge (\sim X_0)) \equiv 0$	Komplement
	$(X_0 \vee (\sim X_0)) \equiv 1$	
(B7)	$(X_0 \wedge 1) \equiv X_0$	neutrale
	$(X_0 \wedge 0) \equiv 0$	Elemente
	$(X_0 \vee 1) \equiv 1$	
	$(X_0 \vee 0) \equiv X_0$	
(B8)	$(\sim (X_0 \wedge X_1)) \equiv ((\sim X_0) \vee (\sim X_1))$	DEMORGAN'sche
	$(\sim (X_0 \vee X_1)) \equiv ((\sim X_0) \wedge (\sim X_1))$	Formeln
(B9)	$(\sim (\sim X_0)) \equiv X_0$	
(B10)	$(X_0 \wedge X_0) \equiv X_0$	Identität
	$(X_0 \vee X_0) \equiv X_0$	

Hierin verbirgt sich die Definition einer BOOLE'schen Algebra, auf die wir jedoch an dieser Stelle nicht näher eingehen müssen. Sie sei im folgenden Abschnitt lediglich in allgemeiner Form definiert.

Man erkennt in obigen Identitäten, dass man durch Vertauschen von $\vee$ und $\wedge$ sowie von 0 und 1 aus einer Zeile die entsprechend folgende erhält. Dieses nennt man das Dualitätsprinzip der BOOLE'schen Algebra.

Man kann die Identitäten ganz stur durch Einsetzen der acht verschiedenen Belegungen der vorkommenden Variablen beweisen.

Beispiel (B8):

X_0	X_1	$(X_0 \wedge X_1)$	$(\sim (X_0 \wedge X_1))$	$(\sim X_0)$	$(\sim X_1)$	$((\sim X_0) \vee (\sim X_1))$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Nun wollen wir ein paar Klammern weglassen (da wir faul sind, jedoch genau wissen, was wir tun). Wir vereinbaren die üblichen Prioritäten: $\sim$ vor $\wedge$ vor $\vee$ und schreiben auch (wie schon oft erwähnt), $\overline{X}$, $\cdot$ und $+$, wobei wir auch oft den Punkt einfach weglassen.

Damit erhalten wir *unvollständig geklammerte* BOOLE*'sche Ausdrücke* (sozusagen als Abkürzungen für vollständig geklammerte BOOLE'sche Ausdrücke).

also statt $((X_0 \wedge (X_1 \wedge (\sim X_2)))$ schreiben wir $X_0 X_1 \overline{X_2}$

Es gilt die folgende Verallgemeinerung der DEMORGAN'schen Formeln

$$\overline{X_0 X_1 X_2 \ldots X_{n-1}} \equiv \overline{X_0} + \overline{X_1} + \overline{X_2} + \ldots + \overline{X_{n-1}}$$
$$\overline{X_0 + X_1 + X_2 + \ldots + X_{n-1}} \equiv \overline{X_0}\,\overline{X_1}\,\overline{X_2} \ldots \overline{X_{n-1}}$$

die mithilfe von Induktion aus obigen Identitäten abgeleitet werden können.

Das Dualitätsprinzip und insbesondere die DEMORGAN'schen Formeln erlauben sogar folgende einfache Umformungen:

$$\overline{X_1 + X_2\overline{X_3} + X_4 X_5 + \overline{X_0}} \equiv \overline{X_1}(\overline{X_2} + X_3)(\overline{X_4} + \overline{X_5})X_0$$

3.5 BOOLE'sche Algebra

Sei $\mathcal{A}$ eine Menge. Eine algebraische Struktur $(\mathcal{A}, \vee, \wedge, \sim)$ heißt BOOLE'sche Algebra, wenn folgendes gilt:

(A_0)	$\vee, \wedge$ sind Abbildungen von $\mathcal{A} \times \mathcal{A}$ nach $\mathcal{A}$ und $\sim$ ist Abbildung von $\mathcal{A}$ nach $\mathcal{A}$	
(A_1)	$X_0 \vee X_1 = X_1 \vee X_0$ $X_0 \wedge X_1 = X_1 \wedge X_0$	Kommutativität
(A_2)	$(X_0 \vee X_1) \vee X_2 = X_0 \vee (X_1 \vee X_2)$ $(X_0 \wedge X_1) \wedge X_2 = X_0 \wedge (X_1 \wedge X_2)$	Assoziativität
(A_3)	$(X_0 \vee X_1) \wedge X_0 = X_0$ $(X_0 \wedge X_1) \vee X_0 = X_0$	Absorption
(A_4)	$X_0 \vee (X_1 \wedge X_2) = (X_0 \vee X_1) \wedge (X_0 \vee X_2)$ $X_0 \wedge (X_1 \vee X_2) = (X_0 \wedge X_1) \vee (X_0 \wedge X_2)$	Distributivität
(A_5)	$X_1 \vee (X_0 \wedge \sim X_0) = X_1$ $X_1 \wedge (X_0 \vee \sim X_0) = X_1$	

Man kann nachprüfen, dass für eine Menge A die Struktur $(\wp(A), \cup, \cap, \mathcal{C})$, wobei $\mathcal{C}$ für die Komplementbildung steht, eine BOOLE'sche Algebra darstellt.

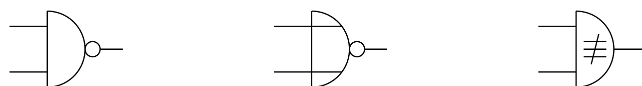
3.6 Weitere einfache Schaltfunktionen

Es gibt 16 Schaltfunktionen mit zwei Eingängen (Erinnerung: $\{0,1\}^2 \longrightarrow \{0,1\}$):

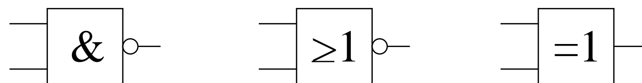
X_0	X_1	ZERO	AND		X_0		X_1	XOR	OR
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

X_0	X_1	NOR	EQUI	$\overline{X_1}$		$\overline{X_0}$		NAND	ONE
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

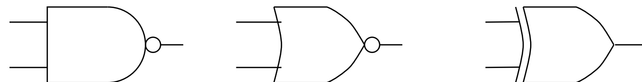
NAND steht für NOT AND, NOR steht für NOT OR, und XOR steht für 'eXclusive OR' also „entweder-das-eine-oder-das-andere" oder auch „mindestens-eins-aber-nicht-beide" oder auch „genau eins" und diese letzte Sichtweise ist die beste!.



alte DIN: NAND, NOR, XOR



neue DIN: NAND, NOR, XOR



amerikanisch: NAND, NOR, XOR

weitere mögliche Symbole NOT und XOR

Allgemein gilt: Es gibt $2^{2^n} = 2^{(2^n)}$ viele n-stellige Schaltfunktionen. Dies sieht man wie folgt ein: man hat $|\{0,1\}^n| = 2^n$ viele Funktionswerte und für jeden Funktionswert zwei Möglichkeiten.

Interpretation der logischen Werte bestimmt den „Gattertyp": was ein UND-Gatter in der einen Interpretation ist ein ODER-Gatter in der anderen. Wir verwenden hier – bloß aus Gewohnheit – die positive Logik.

3.7 BOOLE'sche Ausdrücke und Schaltfunktionen

Wir kennen nun alle 2-stelligen Schaltfunktionen und „eine Menge" von BOOLE'schen Ausdrücken. Nun wollen wir die beiden „Dinge" zusammenbringen. Dazu brauchen wir aber wieder etwas Notation:

Wir schreiben für Variablen $X_i \in V$ und $\epsilon \in \{0,1\}$:

$$X_i^\epsilon = \begin{cases} \overline{X_i} & \text{falls } \epsilon = 0 \\ X_i & \text{falls } \epsilon = 1 \end{cases}$$

Wir nennen BOOLE'sche Ausdrücke der Form X_i^ϵ *Literale*.

Für $a = (a_0, \ldots, a_{n-1}) \in \{0,1\}^n$ heißen die BOOLE'schen Ausdrücke

$$m(a) = X_0^{a_0} \wedge \ldots \wedge X_{n-1}^{a_{n-1}} = \bigwedge_{i=0}^{n-1} X_i^{a_i}$$

die zu a gehörenden *Minterme* und die BOOLE'schen Ausdrücke

$$c(a) = X_0^{\overline{a_0}} \vee \ldots \vee X_{n-1}^{\overline{a_{n-1}}} = \bigvee_{i=0}^{n-1} X_i^{\overline{a_i}}$$

die zu a gehörenden *Maxterme*.

Ein Minterm ist also eine UND-Verknüpfung, bei der alle n Variablen genau einmal entweder negiert oder nicht-negiert vorkommen. Ein Maxterm ist also eine ODER-Verknüpfung, bei der alle n Variablen genau einmal entweder negiert oder nicht-negiert vorkommen. Beachte jedoch: bei Maxtermen „negiert" man bezüglich a!

Beispiel: $m(0,1,0) = \overline{X_0} \wedge X_1 \wedge \overline{X_2}$ und $c(0,1,0) = X_0 \vee \overline{X_1} \vee X_2$

Für n-stellige Schaltfunktionen f heißt die Menge

$$\mathcal{T}(f) = \{a \in \{0,1\}^n | f(a) = 1\}$$

Trägermenge von f.

Die Trägermenge fasst also die Elemente von $\{0,1\}^n$ zusammen, die mit der Funktion f nach 1 abgebildet werden.

So, nun haben wir alles beisammen, um den wichtigen Darstellungssatz zu formulieren:

Sei $f : \{0,1\}^n \longrightarrow \{0,1\}$ eine n-stellige Schaltfunktion. Dann gilt (mit obigen Notationen):

$$f(X) \equiv \bigvee_{a \in \mathcal{T}(f)} m(a)$$

$$f(X) \equiv \bigwedge_{a \notin \mathcal{T}(f)} c(a)$$

Die erste Darstellung heißt die *vollständige disjunktive Normalform* (VDNF), die zweite Darstellung heißt die *vollständige konjunktive Normalform* (VKNF). Manche Bücher schreiben statt *vollständig* auch *kanonisch*. Dies ist ein Satz, den man beweisen muss (wir werden hier darauf verzichten und ihn glauben).

In anderen Worten: die VDNF ist die ODER-Verknüpfung aller Minterme, für die der Funktionswert den Wert 1 annimmt; die VKNF ist die UND-Verknüpfung aller Maxterme, für die der Funktionswert den Wert 0 annimmt.

Man sieht: die Normalformen sind BOOLE'sche Ausdrücke (hier unvollständig geklammert!).

Mit ein bisschen Rechnerei (Anwenden obiger Identitäten) kann man zeigen, dass die VDNF in die VKNF umwandelbar ist.

Beispiel:

X_0	X_1	X_2	$f(X)$	Minterme für VDNF	Maxterme für VKNF
0	0	0	0		$X_0 + X_1 + X_2$
0	0	1	0		$X_0 + X_1 + \overline{X_2}$
0	1	0	1	$\overline{X_0} X_1 \overline{X_2}$	
0	1	1	0		$X_0 + \overline{X_1} + \overline{X_2}$
1	0	0	0		$\overline{X_0} + X_1 + X_2$
1	0	1	0		$\overline{X_0} + X_1 + \overline{X_2}$
1	1	0	1	$X_0 X_1 \overline{X_2}$	
1	1	1	0		$\overline{X_0} + \overline{X_1} + \overline{X_2}$

Trägermenge: $\mathcal{T}(f) = \{(0,1,0),(1,1,0)\}$

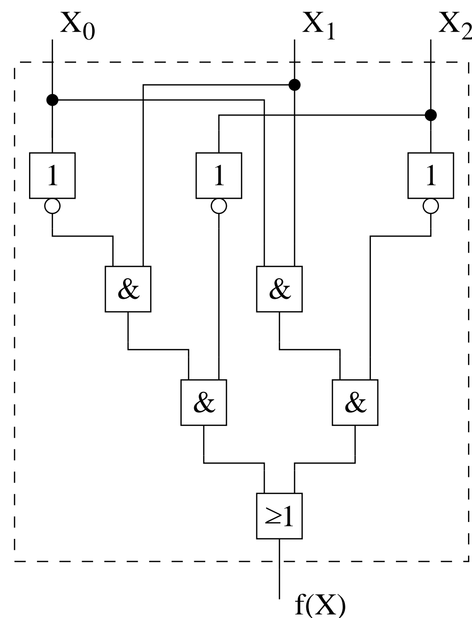
vollständige disjunktive Normalform für f:

$$f(X) \equiv \overline{X_0} X_1 \overline{X_2} + X_0 X_1 \overline{X_2}$$

vollständige konjunktive Normalform für f:

$$\begin{aligned} f(X) \quad \equiv \quad & (X_0 + X_1 + X_2) \cdot (X_0 + X_1 + \overline{X_2}) \cdot (X_0 + \overline{X_1} + \overline{X_2}) \cdot \\ & (\overline{X_0} + X_1 + X_2) \cdot (\overline{X_0} + X_1 + \overline{X_2}) \cdot (\overline{X_0} + \overline{X_1} + \overline{X_2}) \end{aligned}$$

Nun können wir (an dieser Stelle noch etwas informal und ungenau) jeder Schaltfunktion mithilfe eines äquivalenten BOOLE'schen Ausdrucks einen berechnenden Schaltkreis konstruieren:



Realisierung eines BOOLE'schen Ausdrucks

3.8 Kosten BOOLE'scher Ausdrücke

Wir wissen nun: jede Schaltfunktion kann mithilfe eines BOOLE'schen Ausdrucks berechnet werden.

Was kostet uns dieses?

Wir definieren die Kosten $L(e)$ eines BOOLE'schen Ausdrucks $e \in \mathcal{B}$ als die Anzahl der Vorkommen der Zeichen $\wedge$, $\vee$ und $\sim$ (motiviert durch Gatteranzahl, man beachte: auch die Drähte/Verbindungen spielen eine wichtige Rolle für reale Kosten).

Damit sind die Kosten eines unvollständig geklammerten Ausdrucks gleich denen des entsprechend vollständig geklammerten.

Beispiel: $L(((X_0 \wedge (X_1 \wedge (\sim X_2)))) = L(X_0 X_1 \overline{X_2}) = 3$

Wie teuer kann eine Berechnung einer n-stelligen Schaltfunktion werden?

Betrachten wir hierzu die Darstellung einer vollständigen disjunktiven Normalform für eine n-stellige Schaltfunktion f.

- ein Literal X_i^ε kostet höchstens 1, d. h. $L(X_i^\varepsilon) \leq 1$
- ein Minterm m (bestehend aus n Literalen) kostet somit höchstens $2n - 1$, eben $n - 1$ $\wedge$-Zeichen und die Kosten der Literale, d. h. $L(m) \leq 2n - 1$.
- eine vollständige disjunktive Normalform besteht aus genau $|\mathcal{T}(f)|$ vielen Mintermen, somit gilt für die Anzahl v der $\vee$-Zeichen

$$v = \begin{cases} 0 & \text{falls} \quad |\mathcal{T}(f)| \leq 1 \\ |\mathcal{T}(f)| - 1 & \text{falls} \quad |\mathcal{T}(f)| > 1 \end{cases}$$

Da nun $|\mathcal{T}(f)| \leq |\{0,1\}^n| = 2^n$ folgt

$$\begin{aligned} L(e) &\leq |\mathcal{T}(f)| \cdot L(m) + v \\ &\leq 2^n \cdot (2n-1) + 2^n - 1 \\ &\leq n \cdot 2^{n+1} \end{aligned}$$

Beispiel: Kosten eines BOOLE'schen Ausdrucks e für eine 9-stellige Schaltfunktion sind: $L(e) \leq 9 \cdot 2^{10} = 9216$.

Es gibt (geringfügig) bessere obere Schranken (z. B. $L(e) \leq 2.5 \cdot 2^n - 4$), aber auch die folgende untere Schranke, d. h. es gibt n-stellige Schaltfunktionen, zu deren Berechnung jeder BOOLE'sche Ausdruck e mindestens folgende Kosten hat:

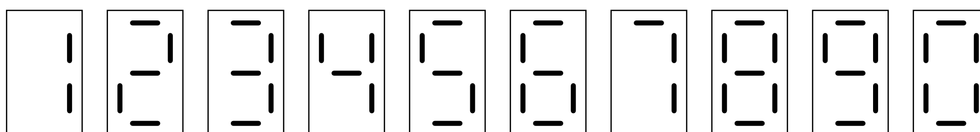
$$L(e) \geq 0.25 \cdot (2^n / \log_2(n+7) - 2)$$

Beispiel: Die Minimalkosten eines bestimmten BOOLE'schen Ausdrucks e für eine 9-stelligen Schaltfunktion sind: $L(e) \geq 31$. Für eine 20-stellige Schaltfunktion ergibt sich $L(e) \geq 550\,000$ und für eine 100-stellige $L(e) \geq 47 \cdot 10^{27}$.

Für diese untere Schranke ist nur ein Existenzbeweis bekannt, man kennt hingegen keine beweisbar schwierige Funktion (die schwersten beweisbaren sind linear in n). (Man kann sich das so vorstellen: ich weiß, es gibt Millionäre, aber keiner meiner Bekannten ist einer.)

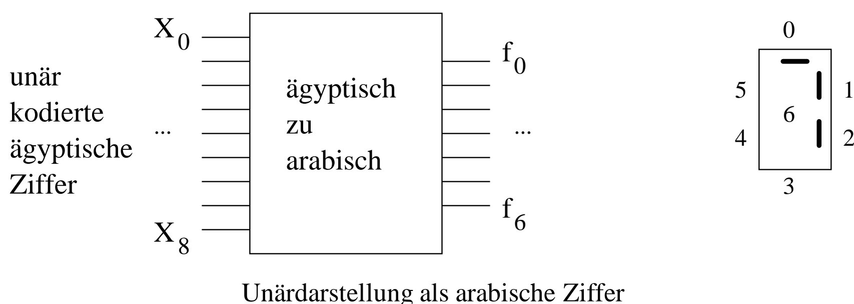
3.9 Beispiele von Schaltfunktionen

Ansteuerung einer Sieben-Segment-Anzeige



Sieben-Segment-Anzeige

Betrachten für zunächst einfach eine Umwandlung der „ägyptischen“ Unärdarstellung ($A = \{|\}, A^{+} = \{|, ||, |||, ||||, \ldots\}$), die keine Null erlaubt, in die arabischen Ziffern, eben ohne Null.



Unärdarstellung als arabische Ziffer

Man erhält mit den Vereinbarungen aus der Abbildung folgende Wertetabelle:

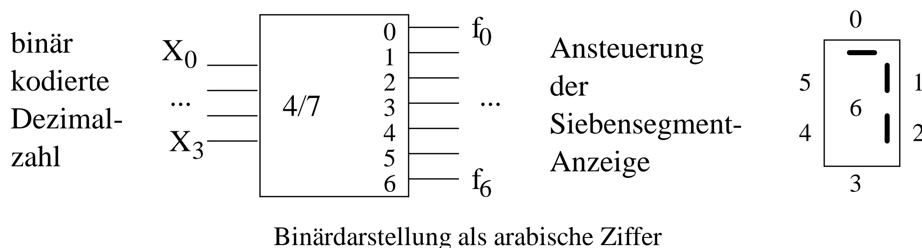
X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	f_0	f_1	f_2	f_3	f_4	f_5	f_6
1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
1	1	0	0	0	0	0	0	0	1	1	0	1	1	0	1
1	1	1	0	0	0	0	0	0	1	1	1	1	0	0	1
1	1	1	1	0	0	0	0	0	0	1	1	0	0	1	1
1	1	1	1	1	0	0	0	0	1	0	1	1	0	1	1
1	1	1	1	1	1	0	0	0	1	0	1	1	1	1	1
1	1	1	1	1	1	1	0	0	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
alle anderen Mögl.									alles 0						

Damit ergibt sich der BOOLE'sche Ausdruck als VDNF für die Funktion f_0 zu:

$$
\begin{aligned}
f_0(X) \quad \equiv \quad & X_0X_1\overline{X_2}\,\overline{X_3}\,\overline{X_4}\,\overline{X_5}\,\overline{X_6}\,\overline{X_7}\,\overline{X_8} + X_0X_1X_2\overline{X_3}\,\overline{X_4}\,\overline{X_5}\,\overline{X_6}\,\overline{X_7}\,\overline{X_8} + \\
& X_0X_1X_2X_3X_4\overline{X_5}\,\overline{X_6}\,\overline{X_7}\,\overline{X_8} + X_0X_1X_2X_3X_4X_5\overline{X_6}\,\overline{X_7}\,\overline{X_8} + \\
& X_0X_1X_2X_3X_4X_5X_6\overline{X_7}\,\overline{X_8} + X_0X_1X_2X_3X_4X_5X_6X_7\overline{X_8} + \\
& X_0X_1X_2X_3X_4X_5X_6X_7X_8
\end{aligned}
$$

Die weiteren Funktionen erhält man auf analoge Art und Weise. Eine VKNF wäre an dieser Stelle wegen der vielen „Nullen“ sehr teuer.

Die heute gebräuchlichere Umsetzung geht von einer Binärdarstellung (sogenannter *binär kodierter Dezimalzahlen* oder einfach BCD-Zahlen) aus. Man benötigt 4 Bit, kann damit aber sogar 16 Ziffern kodieren. Wir nehmen an, dass nur 10 Ziffern umgesetzt werden (eine Erweiterung auf die Hexadezimalzahlen ist einfach), die nicht erlaubten Ziffern dürfen eine beliebige Anzeige erzeugen.



Binärdarstellung als arabische Ziffer

Die Wertetabelle (nun einschließlich einer Null!) sieht wie folgt aus:

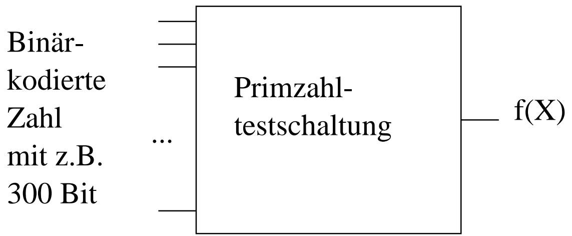
X_0	X_1	X_2	X_3	f_0	f_1	f_2	f_3	f_4	f_5	f_6
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0							
1	0	1	1							
1	1	0	0				hier kann man			
1	1	0	1				frei wählen			
1	1	1	0							
1	1	1	1							

Da oft weniger 0en als 1en vorkommen (d. h. $|\mathcal{T}(f)| > 2^{n-1}$) empfehlen sich für viele der Funktionen die VKNFs, z. B. (man wählt die freien Einträge entsprechend als 1en):

$$f_2(X) \quad \equiv \quad X_0 + X_1 + \overline{X_2} + X_3$$

Man erkennt, dass die Methode manchmal Ausdrücke liefert, die relativ klein sind ($L(f_2) = 4$), oft jedoch auch solche, die teuer sind (weiter oben $L(f_0) = 85$). Die frei wählbaren Einträge nennt man auch ‘don’t care’ Terme oder redundante Terme. Wir werden später nochmal auf die Minimierung der Kosten einer Realisierung zu sprechen kommen.

Man kann sich auch kompliziertere Funktionen vorstellen: z. B. eine Primzahltestschaltung. Diese Schaltung scheint schwierig zu sein, aber niemand konnte es bis jetzt beweisen!



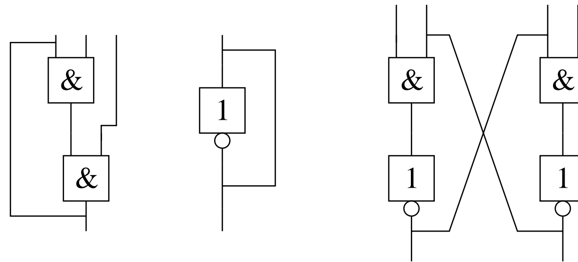
„Möchte-Gern-Schaltung“ zur Primzahlerkennung

4 Graphen und Schaltkreise

Motivation: Kennenlernen eines wichtigen Werkzeugs.

Wir möchten Schaltkreis zeichnen, und herausfinden, ob das was wir gezeichnet haben korrekt ist, was eine solche Realisierung kostet und – später – welche Rechenzeit der Schaltkreis benötigt.

Das Problem: entsprechen den folgenden Zeichnungen Schaltkreise?



Problematische (?) „Schaltkreise“

Um solche und ähnliche Probleme zu handhaben, führen wir das sehr nützliche Werkzeug eines Graphen ein, welches auch für eine Analyse von Datenstrukturen und geometrischen Problemen sehr gut einsetzbar ist.

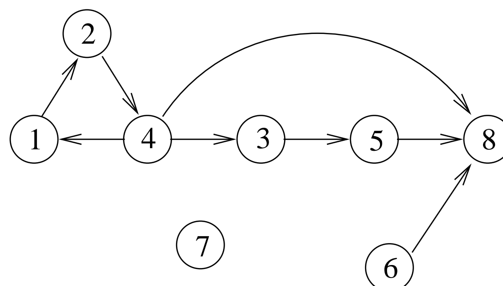
4.1 Endlicher gerichteter Graph

Ein *endlicher gerichteter Graph* wird durch ein Paar $G = (V, E)$ spezifiziert. Dabei ist V eine endliche Menge, die sogenannte Knotenmenge, und $E \subset V \times V$, die sogenannte Kantenmenge, also eine Relation auf der Menge V.

Die Elemente $v \in V$ heißen Knoten, die Elemente $e = (u, v) \in E$ heißen Kanten. Abkürzend sagen wir einfach *Graph*. Wir schreiben V wegen 'vertex' und E wegen 'edge'. Wir hatten V schon einmal für Variablen eines BOOLE'schen Ausdrucks verwendet, das macht uns aber nichts aus!

Graphen kann man zeichnen, die Knoten und Kanten kann man beschriften (was wieder Abbildungen aus der Menge der Knoten bzw. Kanten in Mengen von Beschriftungen wären).

Beispiel:



Ein endlicher gerichteter Graph

$$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

$$E = \{(1, 2), (2, 4), (4, 1), (4, 3), (4, 8), (3, 5), (5, 8), (6, 8)\}$$

Sei im Folgenden $G = (V, E)$ ein Graph.

Was ist an einem Graphen interessant?

Eine Folge von Kanten $e_i = (v_i, w_i) \in E$ für $i \in \{1, \ldots, k\}$ heißt *Pfad*, falls $v_{i+1} = w_i$ für alle $i = 1, \ldots, k - 1$. k ist die *Länge des Pfades*, d. h. gleich der Anzahl der Kanten, die den Pfad bilden.

Wir schreiben entsprechend für eine Pfad p der Länge $l - 1$:

$$p = ((v_1, v_2), (v_2, v_3), \ldots, (v_{l-1}, v_l))$$

oder kürzer

$$p = (v_1, v_2, \ldots, v_l)$$

Beispiel: $(4, 3, 5, 8)$ und $(1, 2, 4, 1, 2)$ sind Pfade.

Gilt für einen Pfad $v_0 = v_l$, so ist es ein *Zykel*. Gerichtete Graphen, die keine Zykel enthalten heißen *zykelfrei*.

Beispiel: $(1, 2, 4, 1)$ ist ein Zykel im obigen Beispielgraph.

Ingrad	$in(v) = \|\{u\|(u, v) \in E\}\|$ Anzahl der Eingangskanten eines Knotens v $in(G) = \max\{in(v)\|v \in V\}$ maximaler Ingrad eines Knotens im Graph
Outgrad	$out(v) = \|\{u\|(v, u) \in E\}\|$ Anzahl der Ausgangskanten eines Knotens v $out(G) = \max\{out(v)\|v \in V\}$ maximaler Outgrad eines Knotens im Graph
Quelle	$in(v) = 0$ Quellen sind Knoten ohne Eingangskanten
Senke	$out(v) = 0$ Senken sind Knoten ohne Ausgangskanten

Beispiel (für obigen Beispielgraphen):

Ingrade einiger Knoten: $in(1) = 1, in(4) = 1, in(8) = 3$.

Outgrade einiger Knoten: $out(1) = 1, out(4) = 3, out(8) = 0$.

Quellen sind die Knoten 6 und 7.

Senken sind die Knoten 7 und 8.

Weiter gilt: $in(G) = 3, out(G) = 3$.

Bemerkung: Graphen müssen nicht zusammenhängend sein, so gibt es oben zum Beispiel keine „Verbindung" von Knoten 7 zum Rest des Graphen.

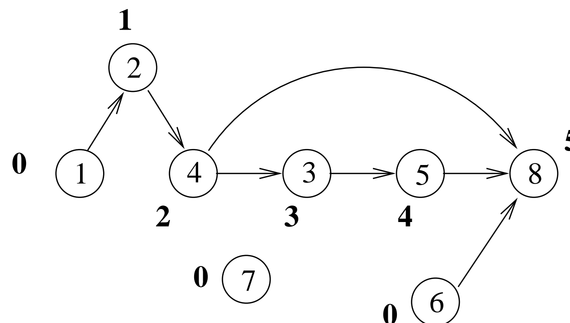
Die *Tiefe* $T(v)$ eines Knotens $v \in V$ wird definiert als die Länge eines längsten Pfades von einer Quelle zu v, falls ein solcher existiert (sonst ist $T(v)$ undefiniert). Die Tiefe einer Quelle wird definiert als 0.

Beispiel: nur den Knoten 6, 7 und 8 kann in obigem Graphen eine Tiefe zugeordnet werden ($T(6) = 0, T(7) = 0, T(8) = 1$).

In einem endlichen zykelfreien Graphen gerichteten Graphen hat jeder Knoten eine Tiefe.

Beweis?!

Entfernen wir die Kante $(4, 1)$ aus obigen Beispielgraphen, so erhalten wir den folgenden zykelfreien Graphen mit entsprechender Tiefe für jeden Knoten.



Tiefe der Knoten in einem zykelfreien Graph

Wir definieren die Tiefe eines zykelfreien Graphen als die maximale Tiefe aller seiner Knoten:

$$T(G) = \max\{T(v) | v \in V\}$$

Damit ist die Tiefe in obigem zykelfreien Graph 5.

Man kann dies alles programmieren und vieles beweisen, aber wir machen hier das meiste durch Hinsehen.

4.2 Schaltkreise als Graphen

Nun können wir Schaltkreise als spezielle Graphen $G = (V, E)$ beschreiben.

Die Elemente der Menge $I = \{0, 1\} \cup \{X_0, \ldots, X_{n-1}\}$ als Teilmenge von V heißen die Eingänge des Schaltkreises. Sie sind die Quellen des Graphen, d. h. $T(v) = 0$ für $v \in I$.

Jeder weitere Knoten in $v \in V - I$ hat einen Ingrad $in(v)$ von 1 oder 2.

Die Kanten E des Graphen geben die Verdrahtung wider.

Jedem Knoten in $v \in V - I$ ordnen wir ein Gatter aus der Menge

$$\{NOT, AND, NAND, OR, NOR, XOR\}$$

zu, dabei muss der Ingrad der Stelligkeit der entsprechenden Funktion entsprechen, also $in(v) = 1$ falls wir v ein NOT zugeordnet haben, sonst ist $in(v) = 2$.

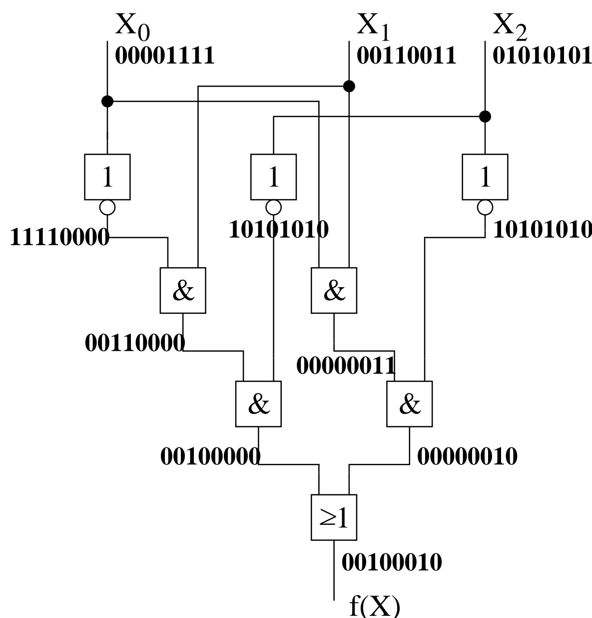
Wir kennzeichnen eine Teilmenge der Knoten als Ausgänge.

Wir können also einen Schaltkreis S dadurch zeichnen, dass wir den Graphen zeichnen, den Quellen die Variablen zuordnen, den Knoten Gatter die Gatter zuordnen und einige der Knoten als Ausgänge markieren.

Da die Richtung der Kanten aus den Schaltsymbolen hervorgeht, die an den Knoten erlaubt sind, kann man sich die Pfeile an den Kanten und die Beschriftungen der Knoten sparen. Ferner zeichnen wir keine Quellen.

Was berechnet ein Schaltkreis?

Wir können analog zu der Vorgehensweise bei den BOOLE'schen Ausdrücken induktiv das Rechnen mit einem Schaltkreis definieren, d. h. wir bestimmten welche Werte an allen Kanten berechnet werden. Insbesondere interessieren uns die Werte an den Ausgängen. Dabei verläuft die Induktion über die Tiefe der Knoten. Man bestimmt, was an den Eingängen anliegt und folgt den Kanten sukkzessive in Richtung der Senken. Wir wollen dies an dieser Stelle nicht weiter formalisieren, sondern begnügen uns mit einem Beispiel:



Was ein Schaltkreis berechnet.

An einen gegebenen Schaltkreis kann man weitere Knoten und Kanten anhängen. Dabei bleibt ein Schaltkreis vorhanden, sofern man keine der oben angegebenen Regeln verletzt. Es werden durch Anbauen keine Werte bereits vorhandener Knoten verändert!

Dies gilt in der Theorie, nicht jedoch in der Praxis, da man nicht beliebig viele Eingänge von Gattern mit dem Ausgang eines Gatters verbinden darf (sogenannte 'fanout'-Beschränkung in einer Technologie). Hierzu später mehr.

Zu jedem BOOLE'schen Ausdruck $e \in \mathcal{B}$ gibt es einen Schaltkreis S mit genau einem Ausgang $f(X)$, so dass

$$e \equiv f(X)$$

Dies beweist man durch Induktion mithilfe der Definition der BOOLE'schen Ausdrücke.

Damit wissen wir auch, zu jeder Schaltfunktion f gibt es einen Schaltkreis, der f berechnet.

Wir bilden also zuerst einen BOOLE'schen Ausdruck (z. B. in einer der Normalformen) und dann bauen wir den entsprechenden Schaltkreis.

4.3 Kosten und Tiefe eines Schaltkreises

Um die Kosten eines Schaltkreises zu bestimmen, ordnen wir zuerst den Gattern gewisse Kosten zu:

Die Funktion $g : \{NOT, AND, OR, NAND, NOR, XOR\} \longrightarrow \mathbb{N}$ ordnen jedem Gattertyp eine natürliche Zahl zu.

Der Einfachheithalber wählen wir als Kosten jedes Gatters 1. Bis auf einen konstanten Faktor schätzen wir damit die Kosten gut ab.

Die Kosten sollen in irgend einer Art und Weise „reale Kosten“ widerspiegeln, z. B. Anzahl der Gatter, Preis der Gatter, Platzverbrauch der Gatter, Verlustleistung der Gatter usw. Durch entsprechende Anpassung der Funktion g können die „realen Kosten“ abgeschätzt werden.

Da wir für g die konstante Funktion 1 gewählt haben, erhalten wir:

Die *Kosten eines Schaltkreises* S ist die Anzahl der inneren Knoten des zugrundeliegenden Graphen (= Anzahl der Gatter), also

$$C(S) = |V - I|$$

Später werden wir sehen, dass Signale eine Zeitlang brauchen um „durch ein Gatter“ zu schalten. Da uns auch die Rechenzeit einer Schaltung interessiert, schätzen wir an dieser Stelle auch diese Laufzeit grob mit der Pfadlänge des zugrundeliegenden Graphen ab. Später wird dieser wichtige Punkt noch ausführlicher behandelt.

Die *Tiefe eines Schaltkreises* S ist die Tiefe des zugrundeliegenden Graphen (= maximale Pfadlänge im Graphen), also

$$T(S) = T(G)$$

Beachte: da nicht nur die Ausgänge Senken des Graphen sein müssen, kann die Tiefe größer sein, als die maximale Pfadlänge von einem Eingang zu einem Ausgang.

5 Spezielle Schaltfunktionen

Motivation: Konstruktion eines Baukastens von „nützlichen“ Schaltfunktionen, Kennenlernen eines rekursiven Konstruktionsverfahrens.

5.1 Multiplexer

Ein Multiplexer ist ein Schaltkreis, der je nach dem Wert eines Auswahlsignal, eines der beiden anderen Eingangssignale auf den Ausgang schaltet.

Schaltfunktion $MUX : \{0,1\}^3 \longrightarrow \{0,1\}$ mit

$$MUX(X_0, X_1, X_2) = \begin{cases} X_0 & \text{falls} \quad X_2 = 0 \\ X_1 & \text{falls} \quad X_2 = 1 \end{cases}$$

Wertetabelle:

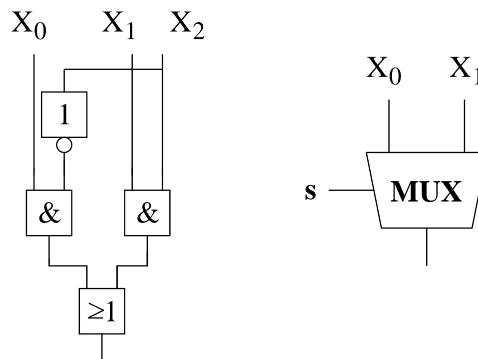
X_0	X_1	X_2	MUX
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

VDNF:

$$MUX(X_0, X_1, X_2) = \overline{X_0}X_1X_2 + X_0\overline{X_1}\ \overline{X_2} + X_0X_1\overline{X_2} + X_0X_1X_2$$

BOOLE’scher Ausdruck geringerer Kosten (durch Anwenden der Identitäten):

$$\begin{aligned} MUX(X_0, X_1, X_2) &= X_0\overline{X_2}(\overline{X_1} + X_1) + X_1X_2(\overline{X_0} + X_0) \\ &= X_0\overline{X_2} + X_1X_2 \end{aligned}$$



Multiplexer mit Schaltsymbol

Kosten: $C(MUX) = 4$, Tiefe: $T(MUX) + 3$

Der Eingang X_2 wird auch häufig mit S bezeichnet (‘select’).

Wir werden im Folgenden diesen Konventionen entsprechen und nicht mehr streng die Variablennamen $X_0, \ldots, X_{n-1}$ verwenden, sondern allgemein übliche Namen zulassen.

Statt nur ein Signal zu multiplexen, kann man auch n Signale multiplexen.

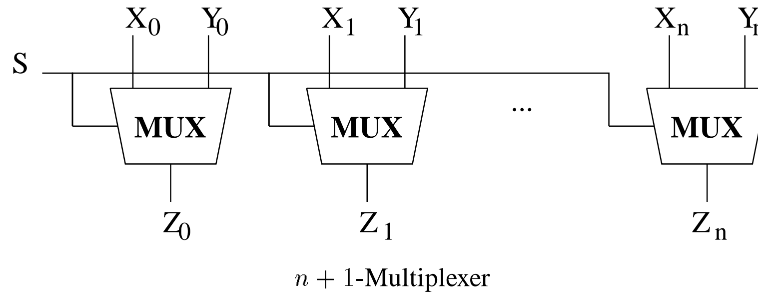
Schaltfunktion $MUX_n : \{0,1\}^{2n+1} \longrightarrow \{0,1\}^n$ mit

$$MUX_n(X_0, \ldots, X_{n-1}, Y_0, \ldots, Y_{n-1}, S) = (Z_0, \ldots, Z_{n-1})$$

wobei

$$(Z_0, \ldots, Z_{n-1}) = \begin{cases} (X_0, \ldots, X_{n-1}) & \text{falls} \quad S = 0 \\ (Y_0, \ldots, Y_{n-1}) & \text{falls} \quad S = 1 \end{cases}$$

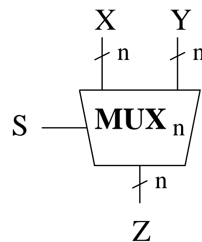
Damit ist der MUX von oben ein MUX_1. Wir können einen breiten Multiplexer wie folgt konstruieren.



$n + 1$-Multiplexer

Man erkennt sofort, Kosten: $C(MUX_n) = 4n$, Tiefe: $T(MUX_n) + 3$

Der Eingang S wird nun allerdings in jedem „inneren“ Multiplexer invertiert. Dies kann man natürlich einsparen und nur einen Inverter verwenden. Damit ergeben sich als Kosten: $C(MUX_n) = 3n+1$. Die Tiefe ändert sich dadurch allerdings nicht.



Schaltsymbol eines n-Multiplexers

Wie im Schaltsymbol für einen n-Multiplexer angegeben, kennzeichnen wir die Anzahl der Verbindungen, die eine einzelne Linie repräsentiert entsprechend und notieren lediglich den Vektor der Schaltvariablen.

5.2 Decoder

Ein Decoder ist ein Schaltkreis, der je nach Wert (Interpretation als Binärzahl) am Eingang, genau einen Ausgang auf 1 setzt.

Schaltfunktion $DEC : \{0,1\} \longrightarrow \{0,1\}^2$ mit

$$DEC(X_0) = (Y_0, Y_1)$$

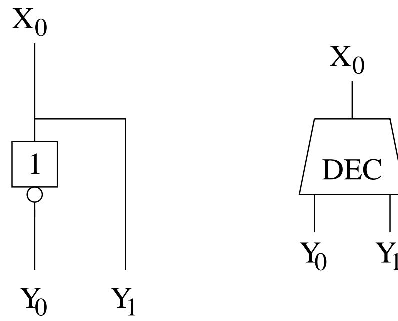
wobei für $i = 0, 1$ gilt:

$$Y_i = \begin{cases} 1 & \text{falls} \quad \langle X_0 \rangle = i \\ 0 & \text{sonst} \end{cases}$$

Wertetabelle:

X_0	Y_0	Y_1
0	1	0
1	0	1

Man sieht sofort: $Y_0 = \overline{X_0}, Y_1 = X_0$



Decoder mit Schaltsymbol

Schaltfunktion $DEC_n : \{0,1\}^{\log n} \longrightarrow \{0,1\}^n$ mit

$$DEC_n(X_0, \ldots, X_{\log n-1}) = (Y_0, \ldots, Y_{n-1})$$

wobei für alle $i = 0, \ldots, n-1$ gilt:

$$Y_i = \begin{cases} 1 & \text{falls} \quad \langle X_{\log n-1} \ldots X_0 \rangle = i \\ 0 & \text{sonst} \end{cases}$$

Damit ist der DEC von oben ein DEC_2.
Wertetabelle für $n = 8$:

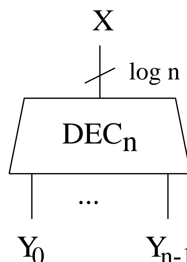
X_2	X_1	X_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

VDNFs für DEC_8:

$$\begin{aligned} Y_0 &= \overline{X_0}\,\overline{X_1}\,\overline{X_2} \\ Y_1 &= \overline{X_0}\,\overline{X_1} X_2 \\ \ldots & \\ Y_7 &= X_0 X_1 X_2 \end{aligned}$$

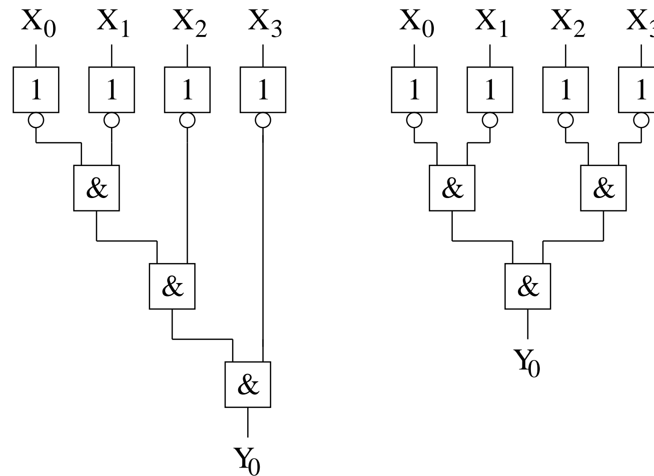
Man kann nun den Decoder mithilfe der Normalformen realisieren, wobei sich für $n = 8$ die Kosten zu $2 \cdot 8 + 8 \cdot 3/2 = 28$ ergeben. Allgemein gilt für die Kosten: $C(DEC_n) = n \cdot (\log n - 1) + n/2 \cdot \log n$, da jeder der n Minterme $\log n - 1$ viele UND-Gatter und insgesamt genauso viele NICHT-Gatter wie Nullen auf der linken Seite der Tabelle vorkommen.

Die Anzahl der Inverter kann reduziert werden, da jede Schaltvariable nur einmal negiert werden muss; es sind also lediglich $\log n$ viele NICHT-Gatter erforderlich. Also: $C(DEC) = n \cdot (\log n - 1) + \log n = n \log n - n + \log n$.



Schaltsymbol des n-Decoder

Für die Tiefe der Konstruktion betrachten wir die folgende Realisierung des Minterms für Y_0 eines 16-Decoders:

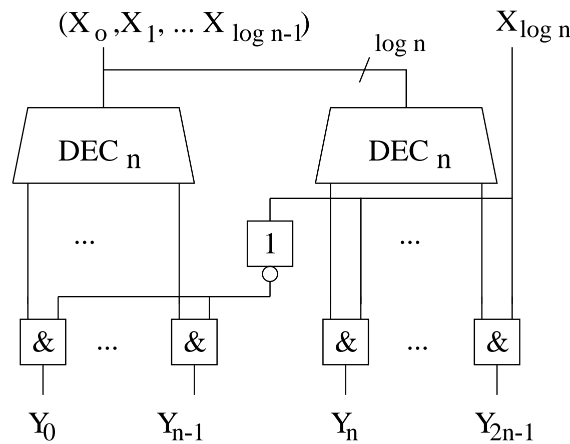


Tiefe eines DEC_{16}

Wir erkennen, dass es viele Möglichkeiten gibt, die UND-Gatter miteinander zu verbinden. Oben ergibt sich links $T(DEC_{16}) = 4$ und rechts $T(DEC_{16}) = 3$. Genaueres erfahren wir später. Hier wollen wir festhalten, dass durch „geschickte" Anordnung der UND-Gatter eine Tiefe von $T(DEC_n) = \log\log n + 1$ erreicht werden kann, wenn mit den VDNFen gearbeitet wird.

5.3 Rekursive Schaltkreiskonstruktion

Um einen Decoder zu realisieren, kann man aber auch zu folgender Methode greifen. Angenommen wir hätten schon einen Decoder mit n Ausgängen. Wir konstruieren aus zwei solchen DEC_n einen DEC_{2n} durch folgende Schaltung:



Rekursive Konstruktion eines $2n$-Decoders

(Bemerkung: diese Schaltung ist offensichtlich nicht die schlauste; man kann auch einfach nur einen DEC_n verwenden, aber trotzdem ist sie korrekt und wir können Kosten und Tiefe berechnen. Später sehen wir die einfachere Schaltung.)

Schreiben wir dies einfacher wie folgt:

$$\begin{aligned}
DEC(n) &= 2 \cdot DEC(n/2) + n + 1 \\
&= 2 \cdot (2 \cdot DEC(n/4) + n/2 + 1) + n + 1 \\
&= 2^2 \cdot DEC(n/4) + n + 2 + n + 1 \\
&= 2^2 \cdot (2 \cdot DEC(n/8) + n/4 + 1) + 2n + 2 + 1 \\
&= 2^3 \cdot DEC(n/8) + 3n + 4 + 2 + 1 \\
&= 2^3 \cdot DEC(n/2^3) + 3n + 2^2 + 2^1 + 2^0
\end{aligned}$$

Wir „sehen“ also die Formel:

$$DEC(n) = 2^i \cdot DEC(n/2^i) + i \cdot n + \sum_{j=0}^{i-1} 2^j$$

Was wir noch nicht kennen, ist i, aber wir wissen $DEC(2) = 1$ und dann hört die Rekursion auf, d. h. wenn gilt:

$$n/2^i = 2$$

damit ist aber durch einfache Rechnung: $i = \log n - 1$ und wir erhalten

$$\begin{aligned} DEC(n) &= 2^{\log n - 1} \cdot DEC(n/2^{\log n - 1}) + (\log n - 1) \cdot n + \sum_{j=0}^{\log n - 1 - 1} 2^j \\ &= n/2 \cdot DEC(2) + n \log n - n + \sum_{j=0}^{\log n - 2} 2^j \\ &= n/2 + n \log n - n + 2^{\log n - 1} - 1 \\ &= n \log n - 1 \end{aligned}$$

Überprüfen wir, ob die Formel für die Kosten korrekt ist. Wir verwenden eine Induktion über die Anzahl der Ausgänge, also Zweierpotenzen $n = 2^k$.

Induktionsanfang für $n = 2$:

$$C(DEC) = C(DEC_2) = 2 \cdot \log 2 - 1 = 2 \cdot 1 - 1 = 1$$

Nehmen wir an, die Induktionsvoraussetzung gilt, also es ist:

$$C(DEC_n) = n \log n - 1$$

Wir führen den Induktionsschritt von n nach $2n$:

Zu zeigen ist: $C(DEC_{2n}) = 2n \log(2n) - 1$

Dies sieht man wie folgt ein:

$$\begin{aligned} C(DEC_{2n}) &= 2 \cdot C(DEC_n) + 2n + 1 \\ &= 2 \cdot (n \log n - 1) + 2n + 1 \\ &= 2n \log n - 2 + 2n + 1 \\ &= 2n \cdot (\log n + 1) - 1 \\ &= 2n \cdot (\log n + \log 2) - 1 \\ &= 2n \log(2n) - 1 \end{aligned}$$

was zu beweisen war.

Diese Rechnung kann man sich sparen, wenn man die allgemeine Rekursionsgleichung anwendet. Diese lautet für eine Funktion $f : \mathbb{N} \longrightarrow \mathbb{N}$ mit

$$\begin{aligned} f(1) &= c \qquad \text{für irgendein } c \in \mathbb{N} \\ f(n) &= a \cdot f\left(\frac{n}{b}\right) + g(n) \end{aligned}$$

für alle Potenzen $n = b^k$ wie folgt

$$f(n) = a^{\log_b n} \cdot c + \sum_{i=0}^{\log_b n - 1} a^i \cdot g\left(\frac{n}{b^i}\right)$$

Die Funktion f ergibt sich für den Decoder als:

$$\begin{aligned} f(1) &= C(DEC_2) = 1 \\ f(n) &= C(DEC_{2n}) \\ &= 2 \cdot C(DEC_n) + 2n + 1 \end{aligned}$$

und wir erkennen die Konstanten sowie die Funktion g als

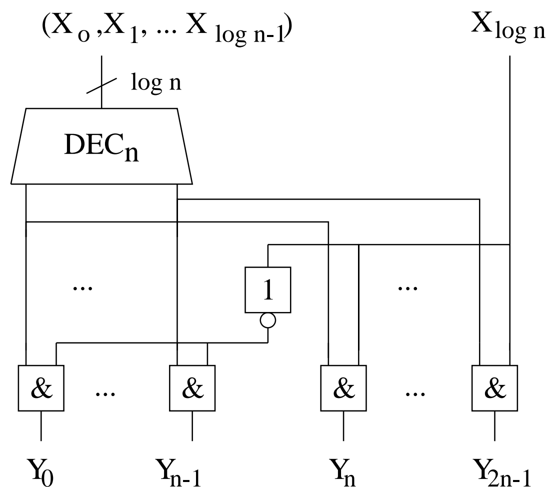
$$a = 2 \qquad b = 2 \qquad c = 1 \qquad g(n) = 2n + 1$$

womit sich die Kosten eines $2n$-Decoders wie folgt berechnen

$$\begin{aligned} f(n) &= a^{\log_b n} \cdot c + \sum_{i=0}^{\log_b n - 1} a^i \cdot g\left(\frac{n}{b^i}\right) \\ &= 2^{\log_2 n} \cdot 1 + \sum_{i=0}^{\log_2 n - 1} 2^i \cdot \left(\frac{2n}{2^i} + 1\right) \\ &= n + \sum_{i=0}^{\log n - 1} 2n + \sum_{i=0}^{\log n - 1} 2^i \\ &= n + 2n \log n + 2^{\log n} - 1 \\ &= 2n \log n + 2n - 1 \\ &= 2n \log(2n) - 1 \end{aligned}$$

Der n-Decoder mit dieser rekursiven Konstruktion ist nur geringfügig teurer als der durch die Normalformen realisierte, obwohl es offensichtlich keine gute Realisierung ist.

Einfachere rekursive Konstruktion eines DEC_{2n}:



Verbesserte rekursive Konstruktion eines $2n$-Decoders

Damit ergibt sich die folgende Rekursionsgleichung:

$$\begin{aligned} f(1) &= C(DEC_2) = 1 \\ f(n) &= C(DEC_{2n}) \\ &= C(DEC_n) + 2n + 1 \end{aligned}$$

und wir erkennen die Konstanten sowie die Funktion g als

$$a = 1 \qquad b = 2 \qquad c = 1 \qquad g(n) = 2n + 1$$

womit sich die Kosten eines $2n$-Decoders einfach wie folgt berechnen

$$\begin{aligned} f(n) &= C(DEC_{2n}) \\ &= a^{\log_b n} \cdot c + \sum_{i=0}^{\log_b n - 1} a^i \cdot g\left(\frac{n}{b^i}\right) \\ &= 1^{\log_2 n} \cdot 1 + \sum_{i=0}^{\log_2 n - 1} 1^i \cdot \left(\frac{2n}{2^i} + 1\right) \\ &= 1 + \sum_{i=0}^{\log n - 1} \frac{2n}{2^i} + \sum_{i=0}^{\log n - 1} 1 \\ &= 1 + 2n \cdot \sum_{i=0}^{\log n - 1} \left(\frac{1}{2}\right)^i + \log n \\ &= 1 + 2n \cdot \frac{(1/2)^{\log n} - 1}{1/2 - 1} + \log n \\ &= 1 + 2n \cdot (1/2^{\log n} - 1) \cdot -2 + \log n \\ &= 1 - 4n \cdot (1/n - 1) + \log n \\ &= 4n + \log n - 3 \end{aligned}$$

und es ergeben sich die Kosten eines n-Decoders als

$$C(DEC_n) = 2n + \log n - 4$$

(wegen $C(DEC_{2n}) = 4n + \log n - 3 = 2 \cdot 2n + \log(2n) - 4$) was erheblich billiger als obige Konstruktionen ist.
Berechnen wir die Tiefe des Decoders ebenfalls mit der Rekursionsformel. Nach Konstruktion gilt offensichtlich

$$T(DEC_n) = \max(T(DEC_{n/2}), 1) + 1$$

nun hat ein Decoder mindestens Tiefe 1, also können wir das Maximum weglassen. Also

$$T(DEC_n) = T(DEC_{n/2}) + 1$$

Damit ergibt sich die folgende Rekursionsgleichung:

$$\begin{aligned} f(1) &= T(DEC_2) = 1 \\ f(n) &= T(DEC_{2n}) \\ &= T(DEC_n) + 1 \end{aligned}$$

und wir erkennen die Konstanten sowie die Funktion g als

$$a = 1 \qquad b = 2 \qquad c = 1 \qquad g(n) = 1$$

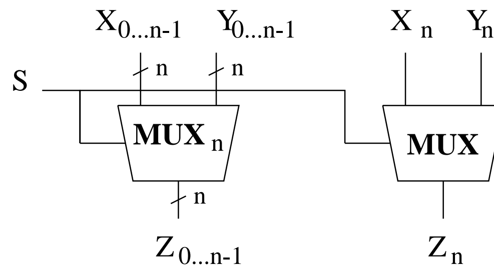
womit sich die Tiefe eines $2n$-Decoders wie folgt berechnet

$$\begin{aligned} f(n) &= T(DEC_{2n}) \\ &= a^{\log_b n} \cdot c + \sum_{i=0}^{\log_b n - 1} a^i \cdot g\left(\frac{n}{b^i}\right) \\ &= 1^{\log_2 n} \cdot 1 + \sum_{i=0}^{\log_2 n - 1} 1^i \cdot 1 \\ &= 1 + \sum_{i=0}^{\log n - 1} 1 \\ &= \log n + 1 \end{aligned}$$

Die Tiefe eines n-Decoders beträgt also $\log(n/2) + 1$ oder

$$T(DEC_n) = \log n$$

Wenden wir dieses Verfahren auch zur Konstruktion eines n-Multiplexers an. Wir nehmen an, wir hätten schon einen $n-1$-Multiplexer konstruiert. Der MUX_n ergibt sich dann wir folgt:



Rekursive Konstruktion des n-Multiplexers

5.4 Demultiplexer

Schaltfunktion $DEMUX : \{0,1\}^2 \longrightarrow \{0,1\}^2$ mit

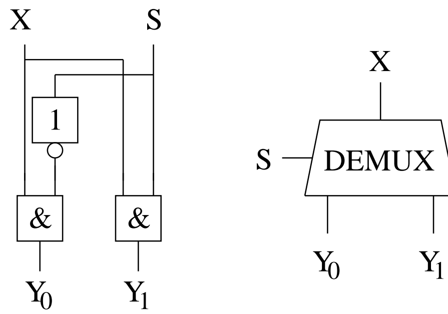
$$DEMUX(X,S) = (Y_0, Y_1) = \begin{cases} (X,0) & \text{falls} \quad S = 0 \\ (0,X) & \text{falls} \quad S = 1 \end{cases}$$

Wertetabelle

X	S	Y_0	Y_1
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

VDNFs:

$$Y_0 = X\overline{S} \qquad Y_1 = XS$$



Demultiplexer mit Schaltsymbol

Kosten: $C(DEMUX) = 3$, Tiefe: $T(DEMUX) = 2$

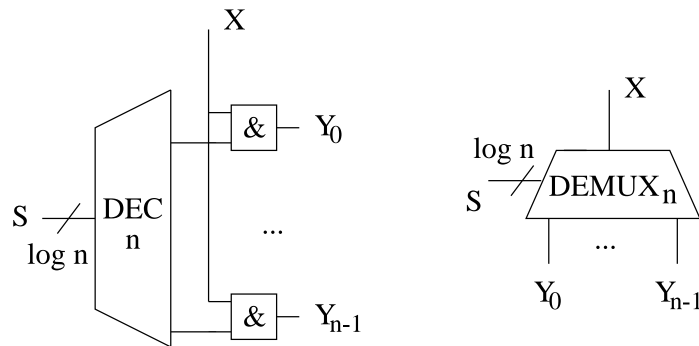
Statt einen Eingang nur auf zwei mögliche Ausgänge zu legen, können wir dies auch für n mögliche Ausgänge tun.

Schaltfunktion $DEMUX_n : \{0,1\}^{1+\log n} \longrightarrow \{0,1\}^n$ mit

$$DEMUX(X, S_0, \ldots, S_{\log n-1}) = (Y_0, \ldots, Y_{n-1})$$

wobei für alle $i = 0, \ldots, n-1$ gilt:

$$Y_i = \begin{cases} X & \text{falls} \quad \langle S_{\log n-1} \ldots S_0 \rangle = i \\ 0 & \text{sonst} \end{cases}$$



n-Demultiplexer mit Schaltsymbol

Kosten: $C(DEMUX_n) = C(DEC_n) + n = 2n + \log n - 4 + n = 3n + \log n - 4$
Tiefe: $T(DEMUX_n) = T(DEC_n) + 1 = \log n + 1$

5.5 Vergleicher

Ein Vergleicher ist ein Schaltkreis, der zwei Eingangssignale miteinander vergleicht und genau einen Ausgang auf 1 setzt, je nachdem, ob der Vergleich kleiner, gleich oder größer ergibt.

Schaltfunktion $CMP : \{0,1\}^2 \longrightarrow \{0,1\}^3$ mit

$$CMP(X_0, Y_0) = (Z_0, Z_1, Z_2)$$

wobei gilt:

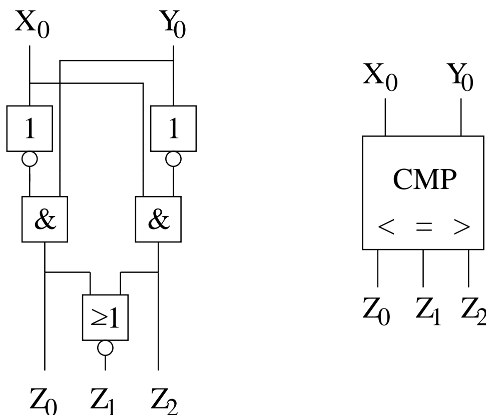
$$\begin{aligned} Z_0 &= \begin{cases} 1 & \text{falls} \quad \langle X_0 \rangle < \langle Y_0 \rangle \\ 0 & \text{sonst} \end{cases} \\ Z_1 &= \begin{cases} 1 & \text{falls} \quad \langle X_0 \rangle = \langle Y_0 \rangle \\ 0 & \text{sonst} \end{cases} \\ Z_2 &= \begin{cases} 1 & \text{falls} \quad \langle X_0 \rangle > \langle Y_0 \rangle \\ 0 & \text{sonst} \end{cases} \end{aligned}$$

Wertetabelle:

X_0	Y_0	Z_0	Z_1	Z_2
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Wir sehen sofort:

$$Z_0 = \overline{X_0}Y_0 \qquad Z_2 = X_0\overline{Y_0} \qquad Z_1 = \overline{Z_0 + Z_2}$$



Vergleicher mit Schaltsymbol

Kosten: $C(CMP) = 5$, Tiefe: $T(CMP) = 3$

Will man zwei n-stellige Binärdarstellungen zweier Zahlen miteinander vergleichen, muss man folgende Schaltfunktion realisieren.

Schaltfunktion $CMP_n : \{0,1\}^{2n} \longrightarrow \{0,1\}^3$ mit

$$CMP(X_0, \ldots, X_{n-1}, Y_0, \ldots, Y_{n-1}) = (Z_0, Z_1, Z_2)$$

wobei gilt:

$$Z_0 = \begin{cases} 1 & \text{falls} \quad \langle X_{n-1} \ldots X_0 \rangle < \langle Y_{n-1} \ldots Y_0 \rangle \\ 0 & \text{sonst} \end{cases}$$

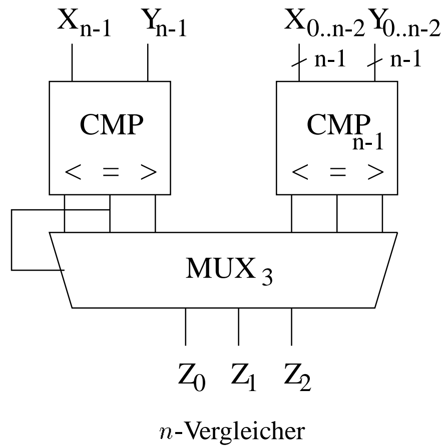
$$Z_1 = \begin{cases} 1 & \text{falls} \quad \langle X_{n-1} \ldots X_0 \rangle = \langle Y_{n-1} \ldots Y_0 \rangle \\ 0 & \text{sonst} \end{cases}$$

$$Z_2 = \begin{cases} 1 & \text{falls} \quad \langle X_{n-1} \ldots X_0 \rangle > \langle Y_{n-1} \ldots Y_0 \rangle \\ 0 & \text{sonst} \end{cases}$$

Nun gilt aber offensichtlich folgendes:

Schreiben wir hierzu abkürzend für $\langle X_{n-1} \ldots X_0 \rangle$ einfach X und entsprechend für $\langle Y_{n-1} \ldots Y_0 \rangle$ einfach Y. X ist kleiner als Y, wenn bereits die oberste Stelle kleiner ist, also $\langle X \rangle < \langle Y \rangle$ falls $\langle X_{n-1} \rangle < \langle Y_{n-1} \rangle$. X ist genau dann größer als Y, wenn bereits die oberste Stelle größer ist, also $\langle X \rangle > \langle Y \rangle$ falls $\langle X_{n-1} \rangle > \langle Y_{n-1} \rangle$. Sind die beiden obersten Stellen gleich, also $\langle X_{n-1} \rangle = \langle Y_{n-1} \rangle$, so entscheiden allein die unteren Stellen.

Wir können also einen n-Vergleicher wie folgt aufbauen:



n-Vergleicher

Für die Kosten ergibt sich die folgende Rekursionsgleichung:

$$\begin{aligned} C(CMP) &= 5 \\ C(CMP_n) &= C(CMP_{n-1}) + C(CMP) + C(MUX_3) \\ &= C(CMP_{n-1}) + 15 \end{aligned}$$

Wir wenden die entsprechende Formel an:

$$\begin{aligned} f(1) &= c \\ f(n) &= a \cdot f(n-1) + g(n) \\ &= a^{n-1} \cdot c + \sum_{i=0}^{n-2} a^i \cdot g(n-i) \end{aligned}$$

Wir sehen:

$$a = 1 \qquad c = 5 \qquad g(n) = 15$$

also

$$\begin{aligned} C(CMP_n) &= 1^{n-1} \cdot 5 + \sum_{i=0}^{n-2} 1^i \cdot 15 \\ &= 5 + (n-1) \cdot 15 \\ &= 15n - 10 \end{aligned}$$

Für die Tiefe ergibt sich die folgende Rekursionsgleichung:

$$\begin{aligned} T(CMP) &= 3 \\ T(CMP_n) &= \max(T(CMP_{n-1}), T(CMP)) + T(MUX_3) \\ &= T(CMP_{n-1}) + 3 \end{aligned}$$

Auch hier können wir das Maximum eliminieren, da stets gilt:

$$T(CMP_{n-1}) \geq T(CMP)$$

. Wir sehen auch hier die Konstanten für die Rekursionsgleichung:

$$a = 1 \qquad c = 3 \qquad g(n) = 3$$

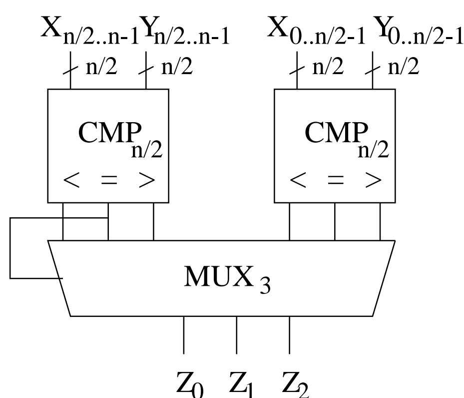
also

$$\begin{aligned} T(CMP_n) &= 1^{n-1} \cdot 3 + \sum_{i=0}^{n-2} 1^i \cdot 3 \\ &= 3 + (n-1) \cdot 3 \\ &= 3n \end{aligned}$$

Die obige Konstruktion benutzt nur das oberste Bit für die Rekursion. Es gilt aber doch auch die allgemeinere Form:

$$\langle X \rangle < \langle Y \rangle \text{ falls } \langle X_{n-1} \ldots X_{n-j} \rangle < \langle Y_{n-1} \ldots Y_{n-j} \rangle$$

für ein beliebiges $j = 1, \ldots, n$. Die anderen Fälle für gleich und größer gelten analog. Lassen wir für n nur Zweierpotenzen zu, so können wir den n-Vergleicher durch folgende Konstruktion realisieren:



Bessere Realisierung des n-Vergleichers

Das heißt, wir vergleichen zuerst die oberen und die unteren Hälften der Eingabewerte und entscheiden anschließend über das Resultat. Für die Kosten ergibt sich nun die folgende Rekursionsgleichung:

$$\begin{aligned} C(CMP) &= 5 \\ C(CMP_n) &= 2 \cdot C(CMP_{n/2}) + C(MUX_3) \\ &= 2 \cdot C(CMP_{n/2}) + 10 \end{aligned}$$

Wir wenden die entsprechende Rekursionsgleichung an:

$$\begin{aligned} f(1) &= c \\ f(n) &= a \cdot f(n/b) + g(n) \\ f(n) &= a^{\log_b n} \cdot c + \sum_{i=0}^{\log_b n - 1} a^i \cdot g\left(\frac{n}{b^i}\right) \end{aligned}$$

Wir sehen:

$$a = 2 \qquad c = 5 \qquad g(n) = 10$$

also

$$
\begin{aligned}
C(CMP_n) &= 2^{\log n} \cdot 5 + \sum_{i=0}^{\log n - 1} 2^i \cdot 10 \\
&= 5 \cdot n + 10 \cdot (n-1) \\
&= 15n - 10
\end{aligned}
$$

Die Kosten haben sich also nicht verändert, was wir auch genau so erwartet haben. Für die Tiefe ergibt sich die folgende Rekursionsgleichung:

$$
\begin{aligned}
T(CMP) &= 3 \\
T(CMP_n) &= T(CMP_{n/2}) + T(MUX_3) \\
&= T(CMP_{n/2}) + 3
\end{aligned}
$$

Wir sehen auch hier die Konstanten für die Rekursionsgleichung:

$$
a = 1 \qquad c = 3 \qquad g(n) = 3
$$

also

$$
\begin{aligned}
T(CMP_n) &= 1^{\log n} \cdot 3 + \sum_{i=0}^{\log n - 1} 1^i \cdot 3 \\
&= 3 + 3 \cdot \log n
\end{aligned}
$$

Die Tiefe hat sich also von $3n$ auf $3(\log n + 1)$ reduziert.

6 Addierer und ALUs

Motivation: Konstruktion verschiedener Addierer sowie einer arithmetisch-logischen Einheit (ALU).

6.1 1-Bit Addierer

6.1.1 Halbaddierer

Ein Halbaddierer realisiert eine 1-stellige Addition zweier Binärzahlen.
Schaltfunktion $HA : \{0,1\}^2 \longrightarrow \{0,1\}^2$ mit

$$HA(a_0, b_0) = (c_1, c_0)$$

wobei gilt:

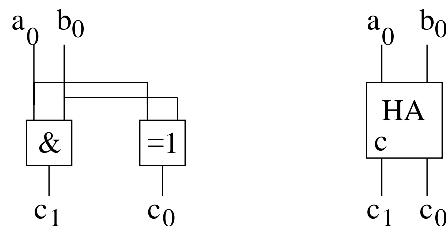
$$\langle a_0 \rangle_2 + \langle b_0 \rangle_2 = \langle c_1 c_0 \rangle_2$$

Der Ausgang $c - 0$ wird auch als Summenbit bezeichnet, c_1 als Übertragsbit oder auch ‘carry bit’.
Wertetabelle:

a_0	b_0	c_1	c_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Die Schaltfunktionen können direkt mit jeweils einem Gatter realisiert werden.

$$\begin{aligned} c_0 &= a_0 \oplus b_0 \\ c_1 &= a_0 b_0 \end{aligned}$$



Halbaddierer mit Schaltsymbol

Kosten: $C(HA) = 2$, Tiefe: $T(HA) = 1$

6.1.2 Volladdierer

Ein Volladdierer realisiert eine 1-stellige Addition zweier Binärzahlen mit Eingangsübertrag.
Schaltfunktion $FA : \{0,1\}^3 \longrightarrow \{0,1\}^2$ mit

$$FA(a_0, b_0, c_0) = (s_1, s_0)$$

wobei gilt:

$$\langle a_0 \rangle_2 + \langle b_0 \rangle_2 + \langle c_0 \rangle_2 = \langle s_1 s_0 \rangle_2$$

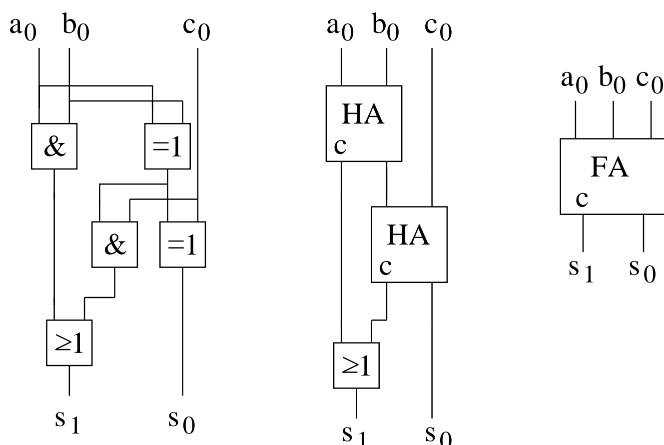
Analog zum Halbaddierer bezeichnet man s_0 als Summenbit und s_1 als Übertrags– bzw. ‘carry’-Bit.

Wertetabelle:

a_0	b_0	c_0	s_1	s_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Einfache Ausdrücke:

$$
\begin{aligned}
s_0 &= a_0 \oplus b_0 \oplus c_0 \\
s_1 &= a_0 b_0 + (a_0 \oplus b_0) c_0
\end{aligned}
$$



Volladdierer, mit Halbaddierern, als Schaltsymbol

Kosten: $C(FA) = 5$, Tiefe: $T(FA) = 3$.

6.2 n-Bit Addierer

Wir haben gesehen, dass es verschiedene Möglichkeiten gibt, einen Schaltkreis aufzubauen. Man kann versuchen die Kosten oder die Tiefe des Schaltkreises zu minimieren. Es ergibt sich häufig ein ‘trade-off’ zwischen Kosten und Tiefe und man muss sich je nach Gegebenheit für eine Variante entscheiden.

Wir wollen im Folgenden einige Addierer kennen lernen, die sich bezüglich Kosten und Tiefe zum Teil erheblich unterscheiden. Wir werden zwar sehen, dass es möglich ist, asymptotisch schnelle und billige Addierer zu realisieren, bei kleinen Bit-Breiten hat man allerdings Wahlmöglichkeiten.

Wie haben wir in der Schule Addieren gelernt?

	1	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1. Summand
	1	1	1	0	1	1	1	0	1	0	1	0	1	1	0	1	2. Summand
1	1	1	1	1	1	1		1		1	1	1	1	1	1		Überträge
1	1	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	Summe

Zuerst die formale Definition eines Addierers.

Ein n-Bit Addierer realisiert eine n-stellige Addition zweier Binärzahlen mit Eingangsübertrag.

Schaltfunktion $ADD_n : \{0,1\}^{2n+1} \longrightarrow \{0,1\}^{n+1}$ mit

$$ADD_n(a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, c_0) = (s_0, \ldots, s_{n-1}, s_n)$$

wobei gilt:

$$\langle a_{n-1} \dots a_0 \rangle_2 + \langle b_{n-1} \dots b_0 \rangle_2 + \langle c_0 \rangle_2 = \langle s_n s_{n-1} \dots s_0 \rangle_2$$

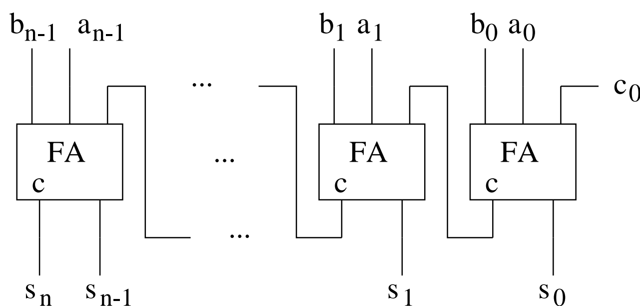
Analog zu den 1-Bit Addierern bezeichnet man s_{n-1} bis s_0 als Summenbits und s_n als Übertrags– bzw. 'carry'-Bit.

Insbesondere gibt es nur einen einzigen Übertrag, der sich gerade auf die nachfolgende Stelle auswirkt (d. h. „eins im Sinn" hat seinen Sinn).

Wir wollen im Folgenden auf formale Beweise der Korrektheit der Addierer verzichten. Der Leser sei jedoch gewarnt, man muss diese Beweise führen.

6.2.1 'ripple carry' Addierer

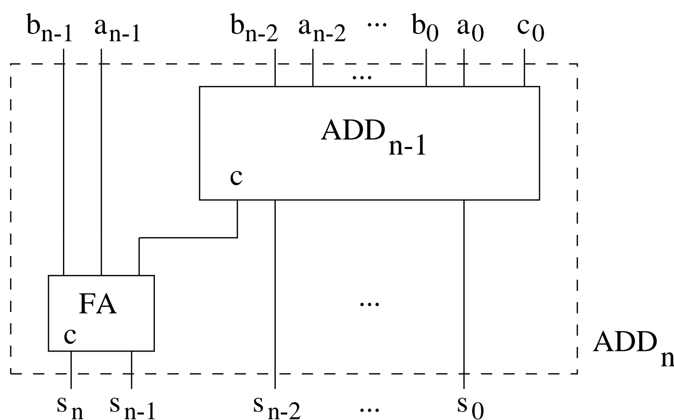
Der erste Addierer arbeitet nach der Schulmethode.



'ripple carry' Addierer

Von rechts nach links werden die entsprechenden Bits und der Übertrag der vorhergehenden Stellen addiert. (Das Bild verdeutlicht auch, weshalb der Addierer 'ripple carry' Addierer heißt: 'ripples' sind kleine regelmäßige Wellen, auf deutsch müsste man „Kräuseladdierer" sagen.) Der 'ripple carry' Addierer wird auch häufig 'carry chain' Addierer genannt. Den Eingangsübertrag der gesamten Schaltung c_0 belegt man mit 0. Ist das oberste Bit der Summe s_n gleich 1, so ist die Summe nicht mehr als n-Bit Zahl darstellbar. Dies wird später in den arithmetisch-logischen Einheiten (Mikroprozessoren) wichtig werden, da wir dort nur n-Bit breite Zahlen weiter verarbeiten können.

Wir können den 'ripple carry' Addierer auch mit folgender Konstruktion rekursiv realisieren:



rekursive Darstellung des 'ripple carry' Addierers

Kosten des 'ripple carry' Addierers RCA_n:

$$\begin{aligned} C(RCA_1) &= C(FA) = 5 \\ C(RCA_n) &= C(RCA_{n-1}) + C(FA) = C(RCA_{n-1}) + 5 \end{aligned}$$

Anwenden der Rekursionsformel:

$$\begin{aligned} f(1) &= c \\ f(n) &= a \cdot f(n-1) + g(n) \end{aligned}$$

Wir sehen die Konstanten:

$$a = 1 \qquad c = 5 \qquad g(n) = 5$$

Damit

$$\begin{aligned} f(n) &= a^{n-1} \cdot c + \sum_{i=0}^{n-2} a^i \cdot g(n-i) \\ &= 1^{n-1} \cdot 5 + \sum_{i=0}^{n-2} 1^i \cdot 5 \\ &= 5 + (n-1) \cdot 5 \\ &= 5n \end{aligned}$$

Die Kosten eines 'ripple carry' Addierers betragen also $C(RCA_n) = 5n$, gerade pro Stelle die Kosten eines Volladdierers, was wir auch so aus der Konstruktion erwartet haben.

Tiefe des 'ripple carry' Addierers RCA_n:

$$\begin{aligned} T(RCA_1) &= T(FA) = 3 \\ T(RCA_n) &= T(RCA_{n-1}) + T(FA) = T(RCA_{n-1}) + 3 \end{aligned}$$

Anwenden der Rekursionsformel:

$$\begin{aligned} f(1) &= c \\ f(n) &= a \cdot f(n-1) + g(n) \end{aligned}$$

Wir sehen die Konstanten:

$$a = 1 \qquad c = 3 \qquad g(n) = 3$$

Damit

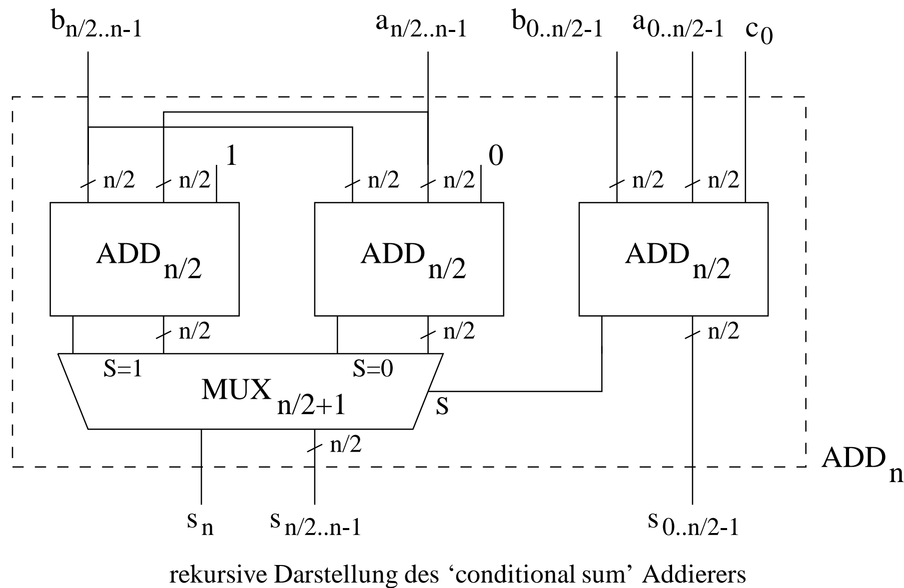
$$\begin{aligned} f(n) &= a^{n-1} \cdot c + \sum_{i=0}^{n-2} a^i \cdot g(n-i) \\ &= 1^{n-1} \cdot 3 + \sum_{i=0}^{n-2} 1^i \cdot 3 \\ &= 3 + (n-1) \cdot 3 \\ &= 3n \end{aligned}$$

Die Tiefe eines 'ripple carry' Addierers beträgt also $T(RCA_n) = 3n$, also gerade Stellenanzahl mal die Tiefe eines Volladdierers, was wir auch so aus der Konstruktion erwartet haben.

Lassen wir wieder nur Zweierpotenzen für die Länge der Eingabezahlen zu, so können wir einen wesentlich schneller addierenden Schaltkreis entwerfen.

6.2.2 'conditional sum' Addierer

Sei n im Folgenden eine Zweierpotenz.



rekursive Darstellung des 'conditional sum' Addierers

Kosten des 'conditional sum' Addierers CSA_n:

$$\begin{aligned}
C(CSA_1) &= C(FA) = 5 \\
C(CSA_n) &= 3 \cdot C(CSA_{n/2}) + C(MUX_{n/2+1}) \\
&= 3 \cdot C(CSA_{n/2}) + 3(n/2+1) + 1 \\
&= 3 \cdot C(CSA_{n/2}) + \frac{3}{2}n + 4
\end{aligned}$$

Anwenden der Rekursionsformel:

$$\begin{aligned}
f(1) &= c \\
f(n) &= a \cdot f(n/b) + g(n)
\end{aligned}$$

Wir sehen die Konstanten:

$$a = 3 \qquad b = 2 \qquad c = 5 \qquad g(n) = \frac{3}{2}\,n + 4$$

Damit erhalten wir (unter Verwendung der geometrischen Reihe und einiger Potenzrechenregeln):

$$\begin{aligned}
f(n) &= a^{\log_b n} \cdot c + \sum_{i=0}^{\log_b n - 1} a^i \cdot g\left(\frac{n}{b^i}\right) \\
&= 3^{\log n} \cdot 5 + \sum_{i=0}^{\log n - 1} 3^i \cdot \left(\frac{3}{2} \cdot \frac{n}{2^i} + 4\right) \\
&= 3^{\log n} \cdot 5 + \sum_{i=0}^{\log n - 1} 3^i \cdot \frac{3}{2} \cdot \frac{n}{2^i} + \sum_{i=0}^{\log n - 1} 3^i \cdot 4
\end{aligned}$$

$$
\begin{aligned}
&= 3^{\log n} \cdot 5 + \frac{3}{2}\, n \cdot \sum_{i=0}^{\log n - 1} \left(\frac{3}{2}\right)^i + 4 \cdot \sum_{i=0}^{\log n - 1} 3^i \\
&= 3^{\log n} \cdot 5 + \frac{3}{2}\, n \cdot \frac{\left(\frac{3}{2}\right)^{\log n} - 1}{\frac{3}{2} - 1} + 4 \cdot \frac{3^{\log n} - 1}{3 - 1} \\
&= 3^{\log n} \cdot 5 + 3n \cdot \left(\frac{3^{\log n}}{2^{\log n}} - 1\right) + 2 \cdot 3^{\log n} - 2 \\
&= 5 \cdot 3^{\log n} + 3 \cdot 3^{\log n} - 3n + 2 \cdot 3^{\log n} - 2 \\
&= 10 \cdot 3^{\log n} - 3n - 2 \\
&= 10 n^{\log 3} - 3n - 2
\end{aligned}
$$

Die Kosten eines ‘conditional sum’ Addierers betragen also ungefähr:

$$C(CSA_n) = 10n^{1.585} - 3n - 2$$

Er ist also um einiges teurer als der ‘ripple carry’ Addierer.

Tiefe des ‘conditional sum’ Addierers CSA_n:

$$
\begin{aligned}
T(CSA_1) &= T(FA) = 3 \\
T(CSA_n) &= T(CSA_{n/2}) + T(MUX_{n/2+1}) = T(CSA_{n/2}) + 3
\end{aligned}
$$

Anwenden der Rekursionsformel:

$$
\begin{aligned}
f(1) &= c \\
f(n) &= a \cdot f(n/b) + g(n)
\end{aligned}
$$

Wir sehen die Konstanten:

$$a = 1 \qquad b = 2 \qquad c = 3 \qquad g(n) = 3$$

Damit

$$
\begin{aligned}
f(n) &= a^{\log_b n} \cdot c + \sum_{i=0}^{\log_b n - 1} a^i \cdot g\left(\frac{n}{b^i}\right) \\
&= 1^{\log n} \cdot 3 + \sum_{i=0}^{\log n - 1} 1^i \cdot 3 \\
&= 3 + 3 \log n
\end{aligned}
$$

Die Tiefe eines ‘conditional sum’ Addierers beträgt also $T(CSA_n) = 3 \cdot (\log n + 1)$; er ist also wesentlich schneller als der ‘ripple carry’ Addierer.

Wir können den ‘conditional sum’ Addierer jedoch noch erheblich verbessern, wie der folgende Abschnitt erläutert.

6.2.3 'two sum' Addierer

Wir haben beim 'conditional sum' Addierer gesehen, dass es sich ausgezahlt hat, beide Möglichkeiten — also *ein* Übertrag von rechts bzw. *kein* Übertrag von rechts — vorzuberechnen.

Bauen wir mit dieser Methode gleich einen ganzen Addierer, der generell stets beide Summen berechnet. Dies scheint ein Mehraufwand zu bedeuten, wir werden aber sehen, dass der Addierer nicht zu teuer, aber sehr schnell ist.

Beginnen wir wieder mit der 1-Bit Version:

Schaltfunktion $TSA' : \{0,1\}^2 \longrightarrow \{0,1\}^4$ mit

$$TSA(a_0, b_0) = (s_0, s_1, t_0, t_1)$$

wobei gilt

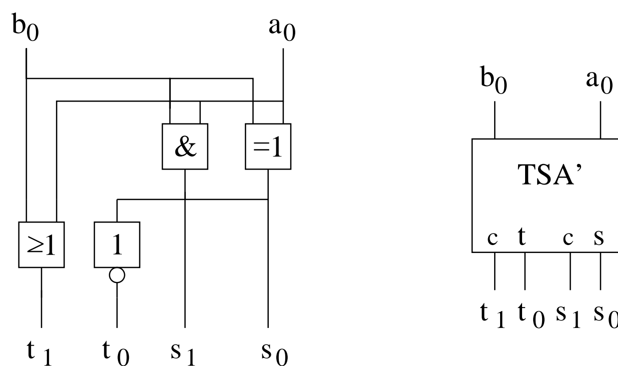
$$\begin{aligned} \langle a_0 \rangle + \langle b_0 \rangle + 0 &= \langle s_1 s_0 \rangle \\ \langle a_0 \rangle + \langle b_0 \rangle + 1 &= \langle t_1 t_0 \rangle \end{aligned}$$

Wertetabelle

a_0	b_0	s_1	s_0	t_1	t_0
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	0	1	1

Und wir sehen sofort

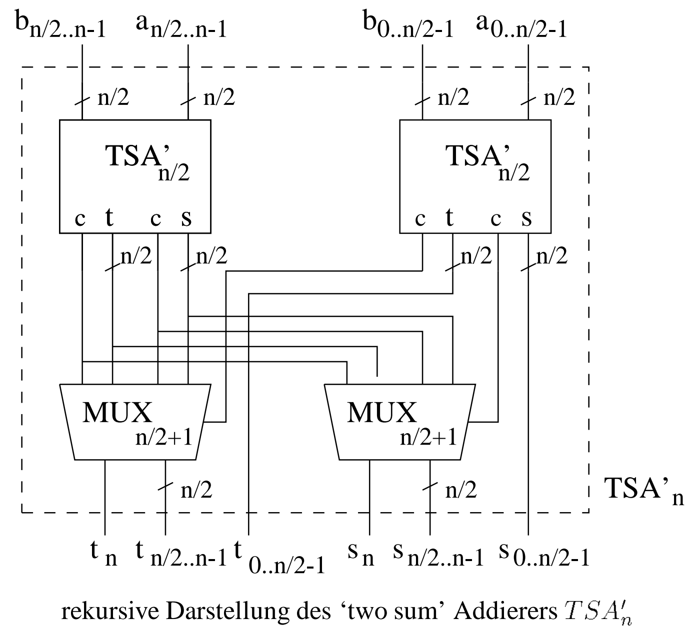
$$\begin{aligned} s_1 &= a_0 b_0 \\ s_0 &= a_0 \oplus b_0 (= a_0 \overline{b_0} + \overline{a_0} b_0) \\ t_1 &= a_0 + b_0 \\ t_0 &= \overline{s_0} \end{aligned}$$



Schaltkreis des 1-stelligen 'two sum' Addierers TSA'

Kosten: $C(TSA') = 4$, Tiefe $T(TSA') = 2$.

Nun konstruieren wir rekursiv einen n-stelligen TSA'_n. Nehmen wir an, wir hätten zwei $TSA'_{n/2}$, die die beiden Summen für die obere und die untere Hälfte der Eingabezahlen berechnen. Die untere Hälfte der Ausgabe wird nur durch den unteren $TSA'_{n/2}$ bestimmt. Die obere Hälfte der Ausgabe wählen wir entsprechend des entstehenden Übertrags an der unteren Hälfte aus.



rekursive Darstellung des 'two sum' Addierers TSA'_n

Kosten des 'two sum' Addierers TSA'_n:

$$\begin{aligned}
C(TSA'_1) &= C(TSA') = 4 \\
C(TSA'_n) &= 2 \cdot C(TSA'_{n/2}) + 2 \cdot C(MUX_{n/2+1}) \\
&= 2 \cdot C(TSA'_{n/2}) + 2 \cdot (3 \cdot (n/2 + 1) + 1) \\
&= 2 \cdot C(TSA'_{n/2}) + 3n + 8
\end{aligned}$$

Anwenden der Rekursionsformel:

$$\begin{aligned}
f(1) &= c \\
f(n) &= a \cdot f(n/b) + g(n)
\end{aligned}$$

Wir sehen die Konstanten:

$$a = 2 \qquad b = 2 \qquad c = 4 \qquad g(n) = 3n + 8$$

Damit

$$\begin{aligned}
f(n) &= a^{\log_b n} \cdot c + \sum_{i=0}^{\log_b n - 1} a^i \cdot g\left(\frac{n}{b^i}\right) \\
&= 2^{\log n} \cdot 4 + \sum_{i=0}^{\log n - 1} 2^i \cdot \left(3 \cdot \frac{n}{2^i} + 8\right) \\
&= 4n + 3n \log n + 8n - 8 \\
&= 3n \log n + 12n - 8
\end{aligned}$$

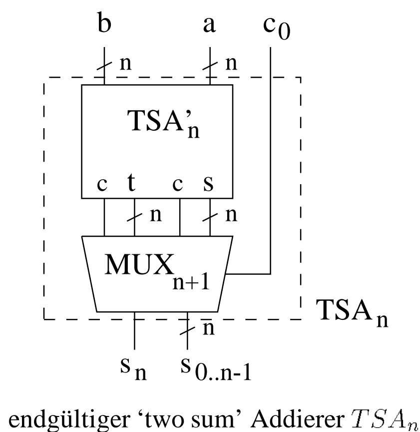
Tiefe des 'two sum' Addierers TSA'_n:

$$\begin{aligned}
T(TSA'_1) &= T(TSA') = 2 \\
T(TSA'_n) &= T(TSA'_{n/2}) + T(MUX_{n/2+1}) = T(TSA'_{n/2}) + 3
\end{aligned}$$

Bis auf die Konstante c ist dies die gleiche Formel wie oben beim CSA_n und wir finden schnell:

$$T(TSA'_n) = 3\log n + 2$$

Da wir am Ende noch mithilfe des Eingangsübertrags die eigentlich gewollte Summe selektieren müssen, ergibt sich der gewünschte Addierer TSA_n aus dem TSA'_n durch Anbauen eines weiteren Multiplexers.



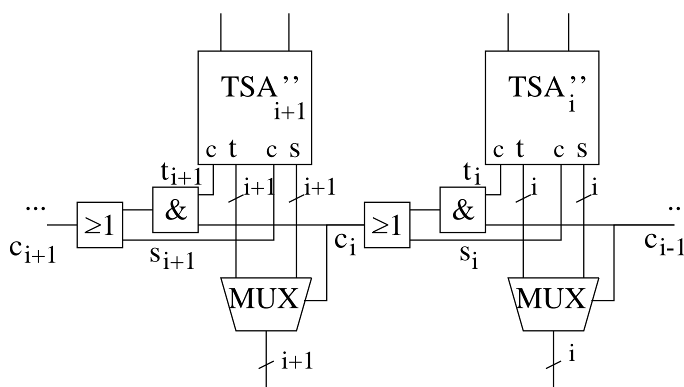
endgültiger 'two sum' Addierer TSA_n

Und wir erhalten für die Kosten und Tiefe letztendlich:

$$\begin{aligned}
C(TSA_n) = C(TSA'_n) + C(MUX_{n+1}) &= 12n + 3n\log n - 8 + 3\cdot(n+1) + 1 \\
&= 3n\log n + 15n - 4 \\
T(TSA_n) = T(TSA'_n) + T(MUX_{n+1}) &= 3\log n + 5
\end{aligned}$$

6.2.4 'block carry' Addierer

Wir wollen einen weiteren recht billigen Addierer kennen lernen, der nicht ganz so schnell wie obige 'conditional sum' bzw. 'two sum' Addierer ist, allerdings wesentlich billiger. Er arbeitet ebenfalls nach dem „Zwei-Summen-Prinzip" und nutzt den „Trick", die zu addierenden Zahlen so in Blöcke einzuteilen, dass die Berechnung des Übertrags der vorhergehenden Blöcke gerade ungefähr so lange dauert wie die Berechnung der Summe des aktuellen Blocks. Man betrachte den folgenden Ausschnitt aus einem Schaltkreis.



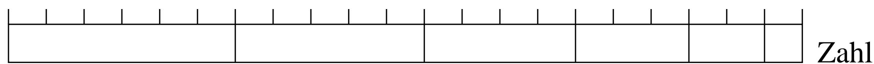
Aufbau zweier Blöcke des 'block carry' Addierers BCA_n

Das Selektsignal c_i für den Block $i+1$ berechnet sich als

$$c_i = s_i + t_i c_{i-1}$$

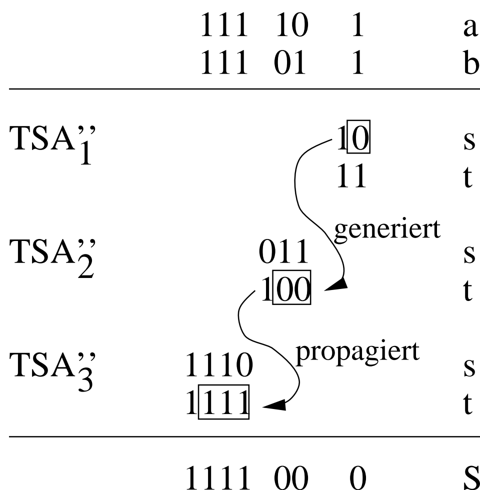
Warum? Nun, wir müssen für Block $i+1$ die Summe mit Übertrag t auswählen, falls bereits der Block i auf jeden Fall einen Übertrag generiert, d. h. $s_i = 1$, *oder* wir müssen für Block $i+1$ die Summe mit Übertrag t auswählen, falls der Block i einen Übertrag propagiert, d. h. $t_i = 1$ und es wird ein Übertrag vom Block $i-1$ geliefert, d. h. $c_{i-1} = 1$.

Wählen wir die Blockgröße so, dass deren Länge stets um 1 wächst, so erhalten wir folgende Einteilung der Zahlen. Die Tabelle gibt die Blockgröße, die gleich der Blocknummer ist, und die maximale Länge n der Zahl an.



Blockeinteilung für den 'block carry' Addierer

Blöcke	1	2	3	4	5	6	7	8	9	10	11
n	1	3	6	10	15	21	28	36	45	55	66



Beispiel der Addition zweier 6-stelliger Zahlen mit BCA

Für eine Stellenzahl n berechnet sich die Anzahl k der Blöcke zu:

$$k = \lceil \sqrt{2n + 0.25} - 0.5 \rceil \leq \lceil \sqrt{2n} \rceil$$

Dies sieht man wie folgt ein. Die Anzahl der Stellen ist gerade die Summe der Blockgrößen:

$$n = \sum_{i=1}^{k} i = \frac{k \cdot (k+1)}{2}$$

Um nun die Anzahl der Blöcke zu berechnen, lösen wir die quadratische Gleichung $n = k \cdot (k+1)/2$ und erhalten die obige (positive und aufgerundete) Lösung für k. Die rechte Abschätzung überprüft man leicht!

Wir wollen den 'block carry' Addierer geschickt aufbauen, so dass die Tiefe in einem Block i, die den Selekteingang — Signal c_i im obigen Schaltkreis — der Stufe $i+1$ bestimmt, gerade ungefähr gleich der Tiefe des dortigen TSA''_{i+1} ist. Es soll also gelten

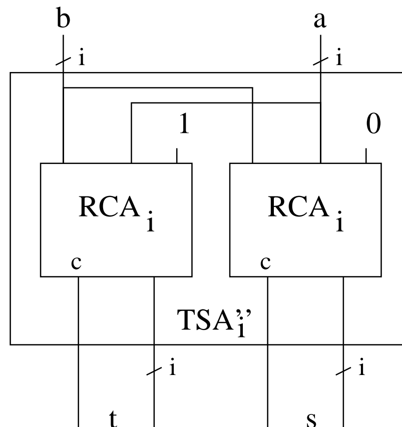
$$T(c_i) \leq T(TSA''_{i+1})$$

Nun gilt für die Tiefe dieses Selekteingangs c_i

$$\begin{aligned} T(c_i) &= 2 + \max(T(TSA''_i), T(c_{i-1})) \\ &= 2 + T(TSA''_i) \end{aligned}$$

d. h. wenn wir als spezielle 'two sum' Addierer TSA''_i gerade einfache 'ripple carry' Addierer — einen für jede zu berechnende Summe — nehmen, haben wir die Bedingung erfüllt und es gilt:

$$\begin{aligned} T(c_i) &= T(TSA''_i) + 2 \\ &= T(RCA_i) + 2 \\ &= 3i + 2 \\ &< 3(i+1) \\ &= T(RCA_{i+1}) \\ &= T(TSA''_{i+1}) \end{aligned}$$



TSA''_i Addierer aufgebaut aus zwei RCA_i Addierern

Damit ist aber die Tiefe eines BCA_n durch die Tiefe seines größten Blocks $T(TSA''_k)$ und des nachfolgenden Multiplexers bestimmt und wir erhalten:

$$T(BCA_n) = T(TSA''_k) + T(MUX_k) = T(RCA_k) + T(MUX_k) = 3k + 3$$

Berechnen wir die Kosten.

Wir benötigen für jeden Block zwei 'ripple carry' Addierer, einen Multiplexer, der die korrekte Summe auswählt und — außer für den letzten Block — die beiden Gatter für die Übertragsweitergabe. Es ergibt sich also

$$\begin{aligned} C(BCA_n) &= \sum_{i=1}^{k} (2 \cdot C(RCA_i) + C(MUX_i)) + 2 \cdot (k-1) \\ &= \sum_{i=1}^{k} (2 \cdot 5 \cdot i + 3 \cdot i + 1) + 2 \cdot (k-1) \\ &= \sum_{i=1}^{k} (13i + 1) + 2 \cdot (k-1) \\ &= 13k \cdot (k+1)/2 + k + 2 \cdot (k-1) \\ &= 0.5 \cdot (13k^2 + 19k) - 2 \end{aligned}$$

Mit obiger Abschätzung für k ergibt sich also abschließend

$$\begin{aligned} C(BCA_n) &\leq 13n + 9.5 \cdot \lceil \sqrt{2n} \rceil - 2 \\ T(BCA_n) &\leq 3 \cdot (\lceil \sqrt{2n} \rceil + 1) \end{aligned}$$

6.2.5 'carry look ahead' Addierer

Der 'carry look ahead' Addierer CLA ist theoretisch und auch in vielen Fällen praktisch der beste Addierer, insbesondere, wenn große Zahlen addiert werden sollen und Verdrahtung nicht dominierend wirkt.

Die Idee dieses Addierers besteht darin, parallel für alle Stellen des Addierers gleichzeitig zu berechnen, ob eine Stelle einen Übertrag generiert oder propagiert. Dies haben wir bereits beim 'block carry' Addierer kennen gelernt. Man beobachtet nun, dass die Berechnung der beiden Signale 'carry generate' und 'carry propagate' für jede Stelle als assoziative Verknüpfung aufgefasst werden kann. Mithilfe eines 'parallel prefix' Schaltkreises können dann alle diese Signale mit Kosten linear in Anzahl der Stellen und Tiefe logarithmisch in Anzahl der Stellen berechnet werden.

Wir möchten an dieser Stelle nicht genau auf diesen *wichtigen* Addierer und das dahinterstehende Konzept eingehen. Dies wird jedoch an anderer Stelle nachgeholt!

Kosten und Tiefe des CLA sind in nachfolgendem Abschnitt aufgeführt.

6.2.6 Zusammenfassung der Addierer

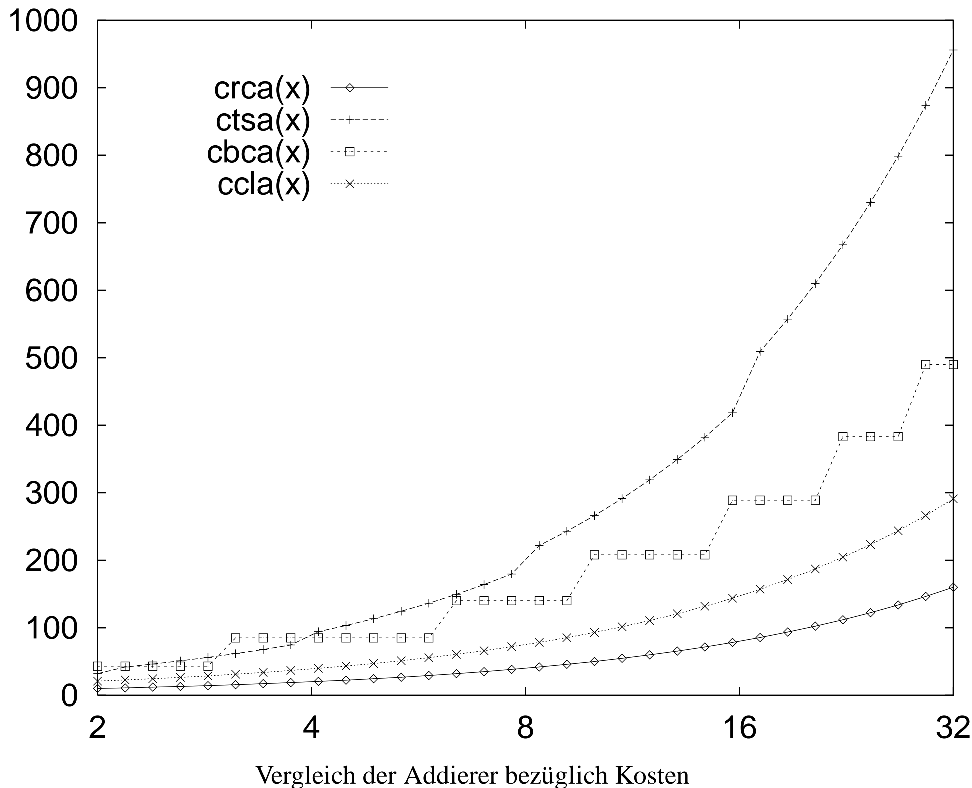
Folgende Tabelle fasst die Kosten und die Tiefe der verschiedenen Addierer einer Breite n zusammen:

Name	Abk.	Kosten	Tiefe
'ripple carry'	RCA	$5n$	$3n$
'conditional sum'	CSA	$\approx 10n^{1.585} - 3n - 2$	$3 \log n + 3$
'two sum'	TSA	$3n \log n + 15n - 4$	$3 \log n + 5$
'block carry'	BCA	$\approx 13n + 9.5\sqrt{2n} - 2$	$\approx 3\sqrt{2n} + 3$
'carry look ahead'	CLA	$9n + 3$	$4 \log n + 2$

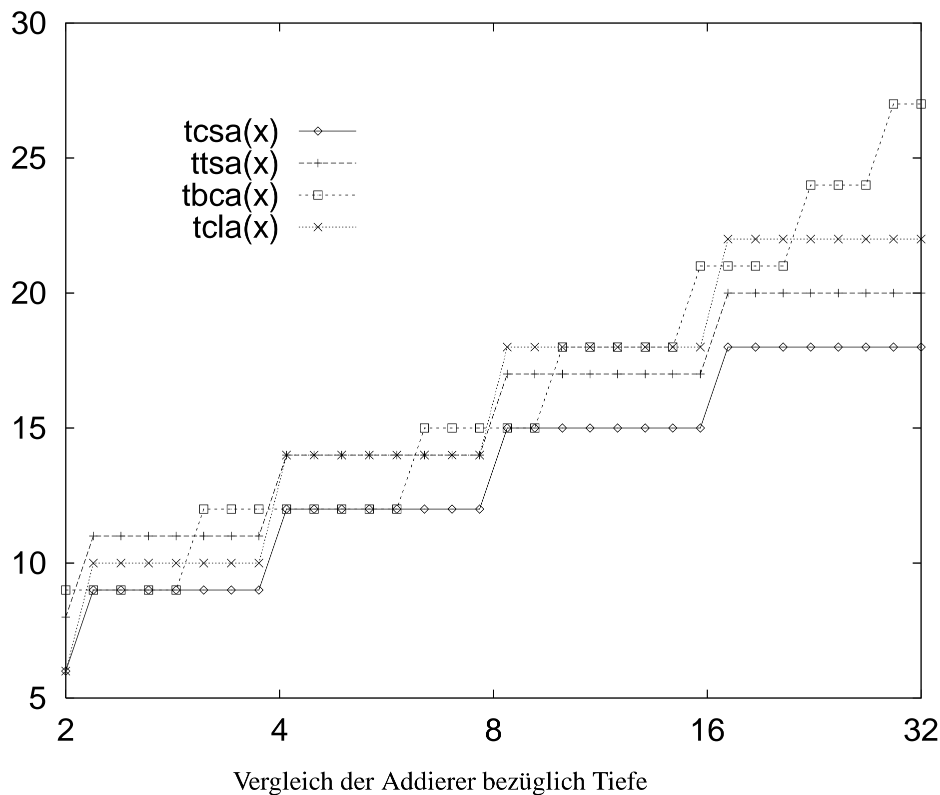
Damit ergibt sich für 32-Bit Addierer:

Addierer	Kosten	Tiefe
RCA	160	96
CSA	2342	18
TSA	956	20
BCA	490	27
CLA	291	22

Dies kann man auch als Funktionsgraphen darstellen:



Vergleich der Addierer bezüglich Kosten



Vergleich der Addierer bezüglich Tiefe

Man beachte, dass diese Diagramme und auch die obigen Formeln so nur für das angegebene Kostenmaß (nämlich pro verwendetem Gatter 1) gelten.

- der schnellste Addierer ist der TSA
- der billigste Addierer ist der RCA
- der CSA ist eine Lösung, wenn man drei kleine Addierer hat und schnell einen schnellen doppelt so großen wie einer der kleinen aufbauen möchte
- gut verdrahten lässt sich der BCA, der dabei nicht langsam ist
- der CLA hat den Vorteil sowohl sehr schnell als auch billig zu sein

Die verschiedenen Konzepte der Addierer können je nach Anforderungen der zu realisierenden Schaltung auch kombiniert angewendet werden. So kann man sich z. B. vorstellen aus 4-Bit Addierern, die schon als sogenannte Macros auf einem Chip vom Hersteller vorliegen, einen 32-Bit CSA zusammenzubauen, dabei jedoch genau nur 15 der Macros verwendet und den Rest nach dem „Zwei-Summen-Prinzip“ über Multiplexer verdrahtet.

6.3 Subtrahierer

Als erstes sei bemerkt: wir können einen Subtrahierer mit der gleichen Methodik wie einen Addierer aufbauen: Realisierung eines Halbsubtrahierers, Realisierung eines Vollsubtrahierers, der den ‘borrow’ (den, den man im Sinn hat) berücksichtigt; und dann z. B. Realisierung eines ‘ripple borrow’ Subtrahieriers, der genau nach der Schulmethode arbeitet.

Versuchen Sie dies zu tun! Beachten Sie, dass wir noch keine besondere „Kennzeichnung“ der negativen Zahlen festgelegt haben. Bis jetzt argumentieren wir einfach mit dem Betrag!

Nun jedoch zur Darstellung negativer Zahlen mit Betrag und Vorzeichen (ist bereits vorne angeführt!).

Wir können also Zahlen in Darstellung mit Betrag und Vorzeichen addieren bzw. subtrahieren, was aber zwei Einheiten und eine spezielle Auswahlschaltung erforderlich machen würde.

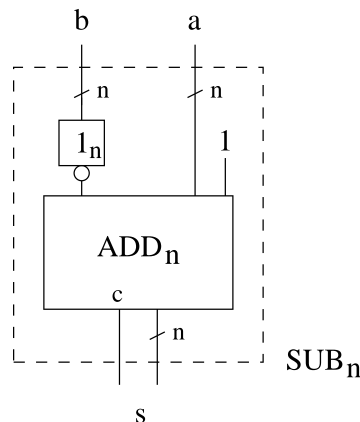
Es gibt aber auch eine geschicktere Methode unter Verwendung der Darstellung im Zweierkomplement.

Seien a und b zwei Zahlen im Bereich $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$, d. h. sie sind im Zweierkomplement als $[a]$ und $[b]$ mit n Bits darstellbar.

Wir wissen, dass gilt: $a - b = a + (-b)$ und von oben: $-[b] = [\overline{b}] + 1$. Wir können also die Subtraktion auf eine Addition zurückführen:

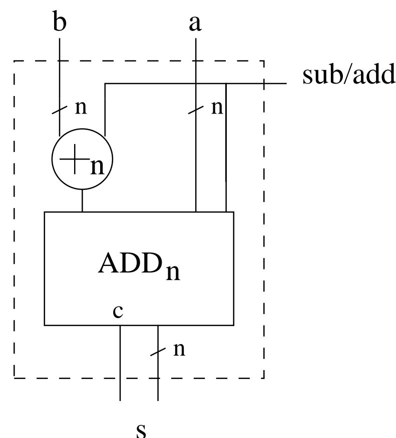
$$\begin{aligned}
[a] - [b] &= [a] + (-[b]) \\
&= [a] + ([\overline{b}] + 1) \\
&= [a] + [\overline{b}] + 1
\end{aligned}$$

Mit anderen Worten, um zu subtrahieren, invertieren wir b bitweise, addieren a wie gewohnt und setzen dabei den Eingangsübertrag auf 1. Nun sehen wir auch ein, weshalb wir diesen Eingangsübertrag bei obigen Addierern stets berücksichtigt haben.



Schaltkreis eines Subtrahierers auf der Basis eines Addierers

Ersetzen wir die NOT-Gatter durch XOR-Gatter, so erhalten wir einen Schaltkreis, der im Zweierkomplement sowohl addieren als auch subtrahieren kann. Folgende Abbildung zeigt diesen Schaltkreis, dabei bedeutet das Symbol des n-XOR, dass die Eingabezahl b bitweise mit dem Eingangsübertrag verknüpft wird. Ist dieser 1, so wird subtrahiert (die XOR bewirken dann eine bitweise Invertierung!), ist dieser 0, so wird addiert.



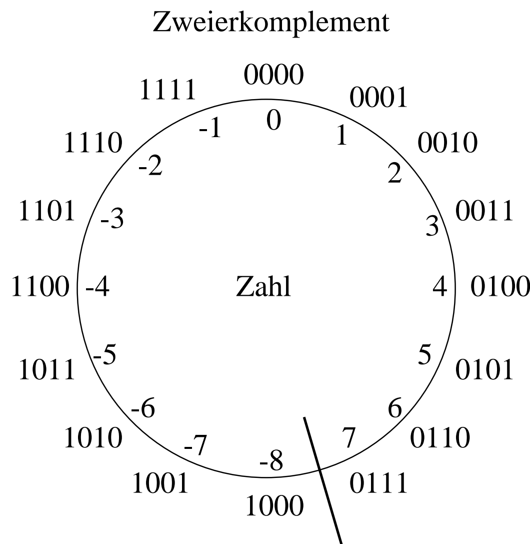
Schaltkreis, der sowohl addiert als auch subtrahiert

6.4 Bereichsüberschreitung

Die obigen Addierer addierten zwei Zahlen in Binärdarstellung. Eine Bereichsüberschreitung (bei diesen nur positiven Zahlen) haben wir an dem Übertrag s_n festgestellt. Wir konnten die Eingabezahlen jedoch auch als Zahlen im Zweierkomplement interpretieren. Damit stellt sich das Problem:

Wir müssen *natürlich* aufpassen, dass wir eine Bereichsüberschreitung erkennen, da sowohl bei der Addition als auch bei der Subtraktion zweier Zahlen aus dem Bereich $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$ eine Bereichsüberschreitung auftreten kann (weshalb ist Ariane-V abgestürzt?).

Betrachten wir nochmals die Veranschaulichung des Zweierkomplements.



Veranschaulichung des Zweierkomplements

Wir erkennen, dass eine Bereichsüberschreiung nur auftreten kann, wenn man entweder zwei positive oder zwei negative Zahlen addiert. Haben beide Summanden unterschiedliches Vorzeichen, so kann keine Bereichsüberschreitung auftreten, denn man „nähert“ sich auf jeden Fall der Null.

Eine Bereichsüberschreitung tritt also nur dann auf, wenn man die markierte Stelle im Kreis überschreitet und dies kann nur durch Addition zweier positiver Zahlen — also im Uhrzeigersinn — oder durch Addition zweier negativer Zahlen — also entgegen dem Uhrzeigersinn — passieren.

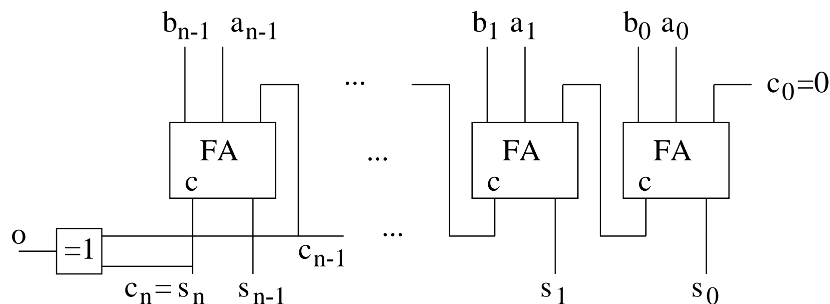
Wie kann man dies einfach feststellen?

Betrachten wir hierzu die Überträge der beiden obersten Stellen. Eine „Überschreitung“ der Grenze im Uhrzeigersinn ist gleichbedeutend damit, dass ein Übertrag für die oberste Stelle erzeugt wurde. Man beachte, dass für positive Zahlen das führende Bit 0 ist, also kein Übertrag von der obersten Stelle erzeugt wird. Andererseits ist eine „Überschreitung“ der Grenze entgegen dem Uhrzeigersinn gleichbedeutend damit, dass gerade für die oberste Stelle *kein* Übertrag erzeugt wurde, denn nur dann erhält man für diese Stelle eine 0, da bei negativen Zahlen das führende Bit stets 1 ist und deren Addition eine 0 als Summenbit und einen Übertrag erzeugt.

Zusammengefasst erkennen wir: es liegt eine Bereichsüberschreitung vor, wenn die beiden obersten Überträge bei der Addition voneinander verschieden sind.

Überprüfen wir noch, dass dies bei der Addition einer positiven mit einer negativen Zahl nicht der Fall sein kann. Nun, dies ist einfach zu sehen: da die obersten Stellen auf jeden Fall unterschiedlich sind, wird kein Übertrag von dieser Stelle generiert, sondern höchsten propagiert. Und dies heißt aber, die Überträge der beiden obersten Stellen sind gleich (es wird propagiert also $c_n = c_{n-1} = 1$ oder es wird nicht propagiert also $c_n = c_{n-1} = 0$).

Eine Bereichsüberschreitung kann also mithilfe eines einzigen XOR-Gatters festgestellt werden.



Erkennung einer Bereichsüberschreitung beim RCA

Das Ausgangssignal o zeigt mit dem Wert 1 an, dass eine Bereichsüberschreitung bei der Addition zweier Zahlen a und b in Zweierkomplementdarstellung vorliegt. c_n ist der Übertrag *von der* obersten Stelle, c_{n-1} ist der Übertrag *für die* oberste Stelle.

6.5 Arithmetisch Logische Einheit

Wir konstruieren eine (einfache) arithmetisch logische Einheit (ALU) für Eingabezahlen im Zweierkomplement. Eine ALU ist der Schaltkreis innerhalb eines Mikroprozessors, der die eigentlichen Rechenoperationen ausführt.

Arithmetisch steht für Addieren und Subtrahieren, *logisch* steht für bitweises AND, OR, und XOR.

Wie baut man eine ALU?

Gehen wir Schritt für Schritt vor!

Welche Funktionen soll die ALU genau berechnen?

Für zwei Eingabezahlen a und b (n-Bit breite Signale) soll *unsere* AUL folgendes können:

0	Berechnen der Konstanten 0
$b-a$	
$a-b$	arithmetische Operationen
$a+b$	
$a \oplus b$	
$a \vee b$	bitweise logische Operationen
$a \wedge b$	
-1	Berechnen der Konstanten -1

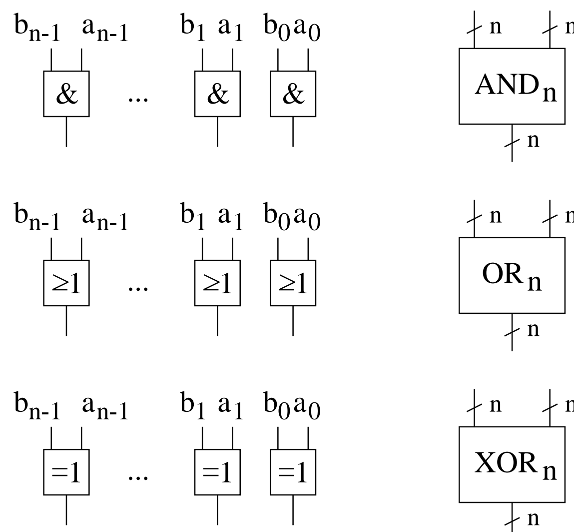
(Warum gerade diese? Es gibt Chips, die genau dieses tun! und man kann schon eine Menge damit tun ...)

Wir konstruieren sie mithilfe unseres „Baukastens“ von Schaltfunktionen (Addierer, die man auch zum Subtrahieren benutzen kann, Multiplexer, AND, XOR, NOT, NAND usw.).

Wir realisieren die „0“ und die „-1“:

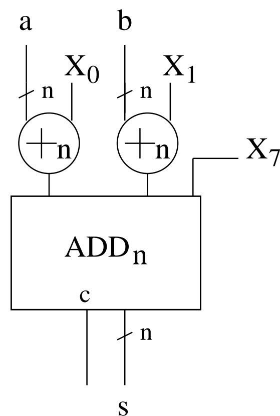
Schaltkreise für die Konstanten

Genauso einfach realisieren wir die bitweisen logischen Operationen, d. h. n-Bit AND, n-Bit OR und n-Bit XOR:



Schaltkreise für die logischen Operationen

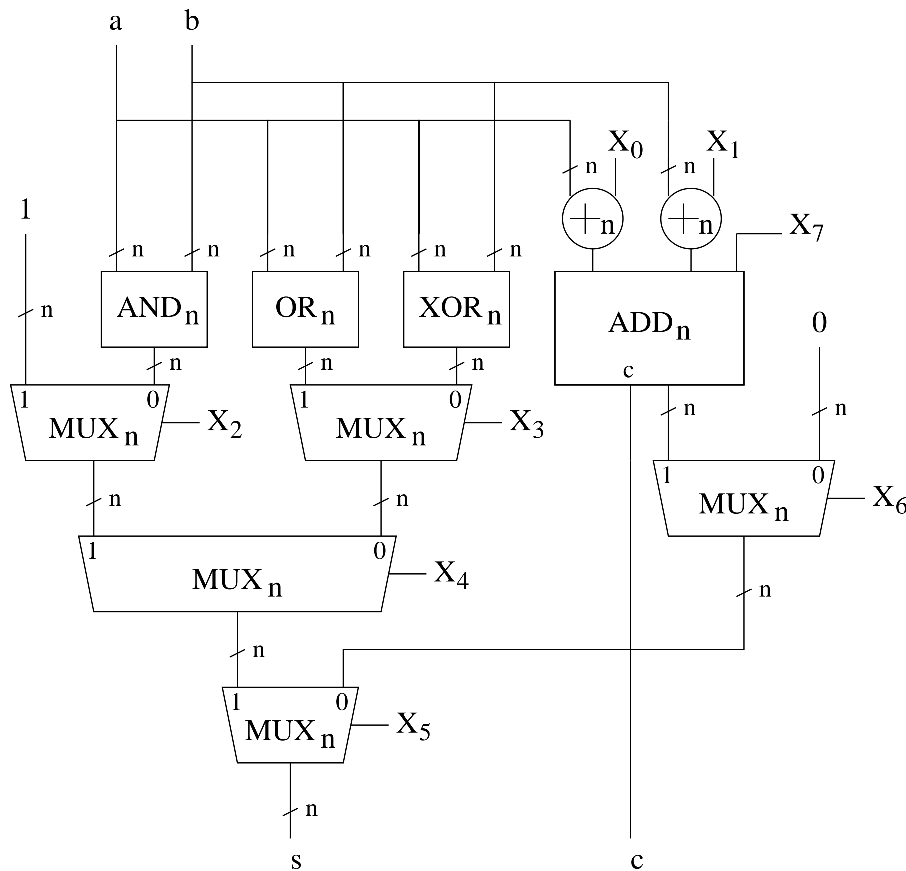
Für die arithmetischen Operationen verwenden wir einen Addierer/Subtrahierer gemäß obiger Ausführungen. Damit wir sowohl $a - b$ als auch $b - a$ berechnen können, verwenden wir an jedem Eingang des Addierers eine entsprechende XOR-Schaltung.



Addierer/Subtrahierer für die ALU

In der obigen Schaltung sind die Kontrollsignale X_0 und X_1 zur bitweisen Invertierung der Operanden und der Eingangsübertrag X_7 getrennt. Wir überlegen uns — nachdem wir die Datenpfade gezeichnet haben — wie diese Kontrollsignale für die drei arithmetischen Operationen eingestellt werden müssen.

Mithilfe von Multiplexern setzen wir diese Schaltkreise nun zu den Datenpfaden der ALU zusammen:



Datenpfade der ALU

Um die Datenpfade korrekt zu steuern, müssen wir die Kontrollsignale $X_0, \ldots, X_7$ je nach anliegender Kodierung (f_0, f_1, f_2) für die zu berechnende Funktion realisieren. Es ergibt sich die folgende Wertetabelle, wobei die mit - gekennzeichneten Werte frei wählbar sind.

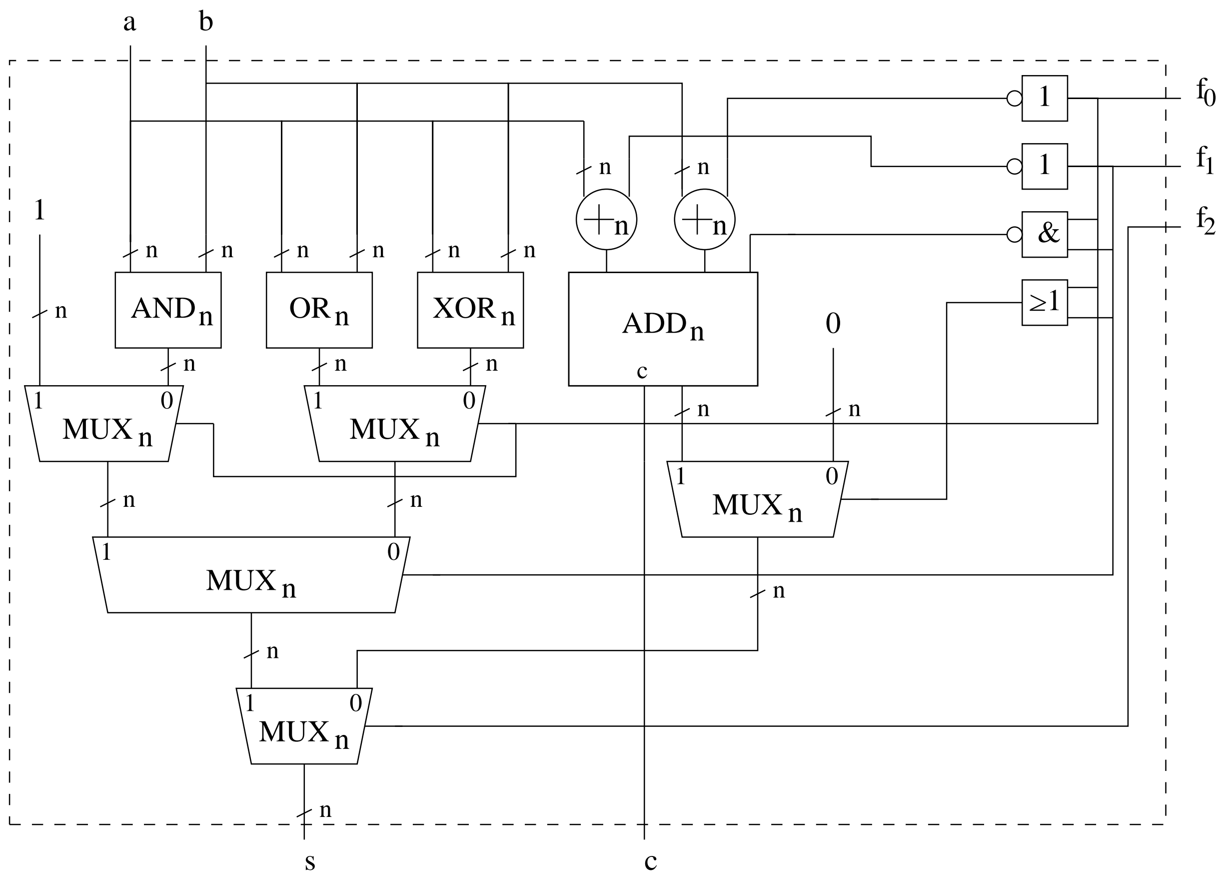
Funktion	f_2	f_1	f_0	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
0	0	0	0	-	-	-	-	-	0	0	-
$b - a$	0	0	1	1	0	-	-	-	0	1	1
$a - b$	0	1	0	0	1	-	-	-	0	1	1
$a + b$	0	1	1	0	0	-	-	-	0	1	0
$a \oplus b$	1	0	0	-	-	-	0	0	1	-	-
$a \vee b$	1	0	1	-	-	-	1	0	1	-	-
$a \wedge b$	1	1	0	-	-	0	-	1	1	-	-
-1	1	1	1	-	-	1	-	1	1	-	-

Die Kontrollsignale können wir nun mit der bekannten Technik (z. B. mit VDNF) realisieren. Allerdings können wir auch die frei wählbaren Werte gemäß folgender Tabelle festlegen und erhalten dann die anschließend angeführten *einfachen* Gleichungen.

Funktion	f_2	f_1	f_0	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
0	0	0	0	1	1	0	0	0	0	0	1
$b - a$	0	0	1	1	0	1	1	0	0	1	1
$a - b$	0	1	0	0	1	0	0	1	0	1	1
$a + b$	0	1	1	0	0	1	1	1	0	1	0
$a \oplus b$	1	0	0	1	1	0	0	0	1	0	1
$a \vee b$	1	0	1	1	0	1	1	0	1	1	1
$a \wedge b$	1	1	0	0	1	0	0	1	1	1	1
-1	1	1	1	0	0	1	1	1	1	1	0

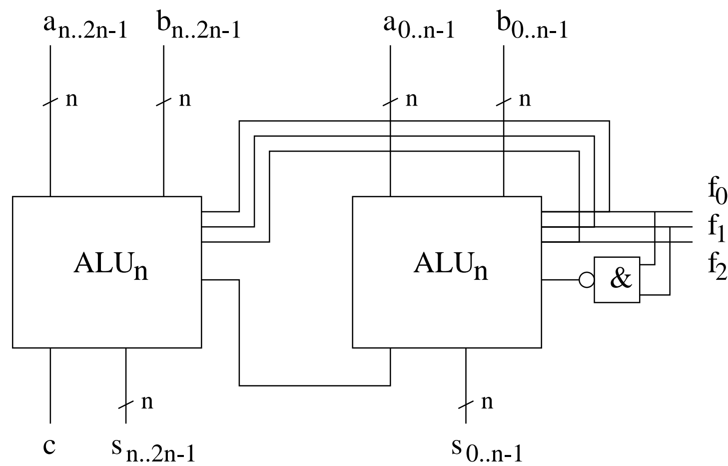
$$
\begin{aligned}
X_0 &= \overline{f_1} \\
X_1 &= \overline{f_0} \\
X_2 &= f_0 \\
X_3 &= f_0 \\
X_4 &= f_1 \\
X_5 &= f_2 \\
X_6 &= f_0 + f_1 \\
X_7 &= \overline{f_0 f_1}
\end{aligned}
$$

Damit ergibt sich der Schaltkreis der ALU einschließlich der Kontrollsignale als:



Datenpfade und Kontrollsignale der ALU

Legen wir die Berechnung von X_7, also des Eingangsübertrags für den Addierer, außerhalb der ALU, so können wir mehrere ALUs kaskadieren. Mithilfe zweier n-Bit ALU ergibt sich die folgende $2n$-Bit ALU:



Kaskadierung von ALUs

Dies kann man so fortsetzen. Das Verfahren spezielle ALUs (unsere ALU ist eine *spezielle* ALU) nach diesem einfachen Schema zu kaskadieren, nennt man ‘bit slice’ Verfahren, die zugrundeliegende kleine ALU ‘bit slice’ ALU. Es stehen als Chips 4-Bit ‘bit slice’ ALUs zur Verfügung, diese sind auch häufig in Makrobibliotheken für programmierbare Bausteine vorhanden (hierzu später mehr).

Bemerkung: berechnet man sich wieder geeignet ‘generate’ und ‘propagate’ Signale, dann kann man auch mithilfe von ‘bit slice’ ALUs und entsprechenden Kopplungsschaltkreisen (sogenannter ‘carry lookahead’ Generatoren) schnelle ALUs nach dem ‘carry lookahead’ Prinzip aufbauen. Auch hierzu gibt es entsprechende Chips zu kaufen.

7 Zusammenfassung der Symbole und Notationen

Motivation: neue Symbole und Notationen wiederholen.

A	Alphabet
A^+	Menge von Worten über A
	Unärdarstellung ohne Null
A^*	A^+ mit leerem Wort ϵ, Unärdarstellung mit Null
$\mathcal{B}$	Menge der BOOLE'sche Ausdrücke
$c(a)$	Maxterm zu a
$C(S)$	Kosten eines Schaltkreises
$\mathcal{C}$	Menge der erweiterten BOOLE'sche Ausdrücke
e	BOOLE'scher Ausdruck
E	Kantenmenge eines Graphen
ϵ	leeres Wort
$f(X)$	Schaltfunktion über Vektor von Schaltvariablen
$g()$	Kostenfunktion der Gatter
G	$G = (V, E)$ endlicher gerichteter Graph
$in(v)$	Ingrad eines Knotens in einem Graphen
$l(a)$	Stellenzahl einer Zahl $\langle a \rangle$
$l(w)$	Länge eines Wortes
$L(e)$	Kosten eines BOOLE'sche Ausdrücke
$m(a)$	Minterm zu a
$out(v)$	Outgrad eines Knotens in einem Graphen
$\varphi(x)$	bijektive Zuordnungsfunktion
$\Phi(x)$	Einsetzungsfunktion
$s(f)$	Stelligkeit von f
$T(v)$	Tiefe eines Knotens in einem Graphen
$T(S)$	Tiefe eines Schaltkreises
$\mathcal{T}(f)$	Trägermenge von f
V	Variablenmenge für einen BOOLE'schen Ausdruck
	Knotenmenge eines Graphen
X	Vektor der Schaltvariablen
$X_0, \ldots, X_{n-1}$	Schaltvariablen
X_i^ε	Literal
$\sim$	Nicht-Operator
$\wedge$	Und-Operator
$\vee$	Oder-Operator
$\oplus$	Exor-Operator
$\langle a \rangle_b$	Zahl zur Basis b

8 Technologien

Motivation: sehr kurze Vorstellung von Chip-Technologie.

8.1 Daten und Begriffe

Integrierte Schaltungen (IC, ‘integrated circuits’) sind auf einem Chip (Basismaterial meist Silizium oder Galium-Arsenid) aufgebaute Schaltkreise (Schaltwerke), die nicht getrennt werden können ohne sie zu zerstören.

Die wesesentlichen sogenannten *Funktionselemente* (FE) auf einem Chip sind *Transistoren* (Schalter), *Widerstände* (Lasten, Leitungen) und *Dioden* (Sperren, Potentialregler).

Die *Integrationsdichte* nahm ständig zu: 1960 ca. 10 FE, 1995 ca. 10^8 FE pro Chip; man spricht von einer Vervierfachung alle drei Jahre.

Bemerkung: die Integrationsdichte ist im eigentlichen Sinn keine Dichte, sondern die Anzahl der FE pro Chip.

Klassifizierung nach Integrationsdichte:

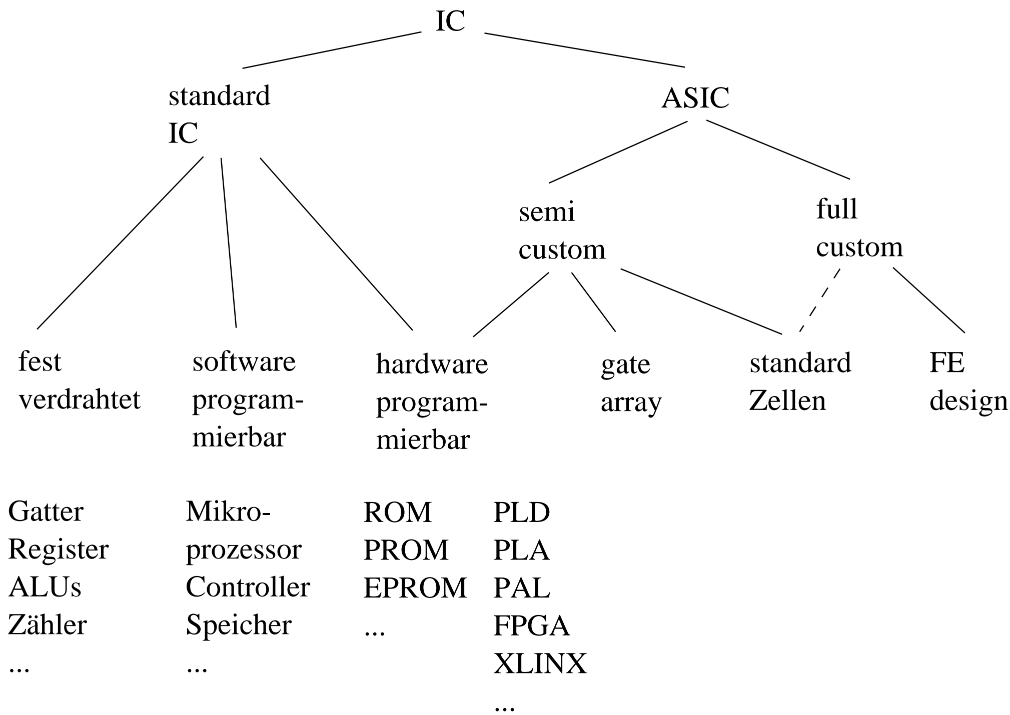
$< 10^2$	SSI	‘small scale integration’
$10^2 - 10^3$	MSI	‘medium scale integration’
$10^3 - 10^5$	LSI	‘large scale integration’
$10^5 - 10^6$	VLSI	‘very large scale integration’
$10^6 - 10^9$	ULSI (V^2LSI)	‘ultra large scale integration’
$> 10^9$	GLSI	‘wafer scale integration’

Einschub: standardisierte Vorsilben für Maßeinheiten

			a	atto	10^{-18}
P	Peta	10^{15}	f	fempto	10^{-15}
T	Tera	10^{12}	p	pico	10^{-12}
G	Giga	10^9	n	nano	10^{-9}
M	Mega	10^6	μ	micro	10^{-6}
K	Kilo	10^3	m	milli	10^{-3}
			d	dezi	10^{-1}
			c	centi	10^{-2}

Bemerkungen: Manchmal benutzt man auch k statt K. In der Informatik wird häufig Kilo mit $2^{10} = 1024$ und Mega mit $2^{20} = 1048576$ gleichgesetzt.

Heute werden in der Massenproduktion fast ausschließlich Dünnschichttechnologien verwendet, d. h. die einzelnen Strukturen des Chips werden in dünnen Schichten (Halbleitermaterial verschiedener Dotierung, Oxidschichten und Metallschichten) übereinander angeordnet.



Klassifizierung nach Spezialisierung

Im Allgemeinen nimmt die Integrationsdichte im obigen Diagramm von links nach rechts zu. Dies gilt fast ebenso für die Preise (beachte, einige der StandardICs sind 'full custom' Chips).

Klassifizierung nach Technologie:

TTL 'transistor transistor logic', ECL 'emitter coupled logic', I^2L 'integrated injection logic', MOS 'metal oxide silicon' (PMOS, NMOS, CMOS, BiMOS)

Die wesentlichen Unterschiede bestehen darin, in welchen Kennlinienbereichen die Transistoren betrieben werden, wie die Transistoren technisch aufgebaut sind und welche Methode der Steuerung verwendet wird. Wir wollen nicht auf die genauen technischen und physikalischen Unterschiede dieser Technologien eingehen, sondern geben uns mit den für uns wichtigen Parametern zufrieden (z. B. Spannungsversorgung, Zeitverhalten, Zusammenschaltbarkeit).

Fast alle Hersteller benutzen zur Kennzeichnung der Standardbausteine die folgende Nomenklatur (zumindest für SSI, MSI und LSI Bausteine). Die Zeichenkette der Bezeichnung eines Chips kann in die Komponenten *BTNPt* zerlegt werden.

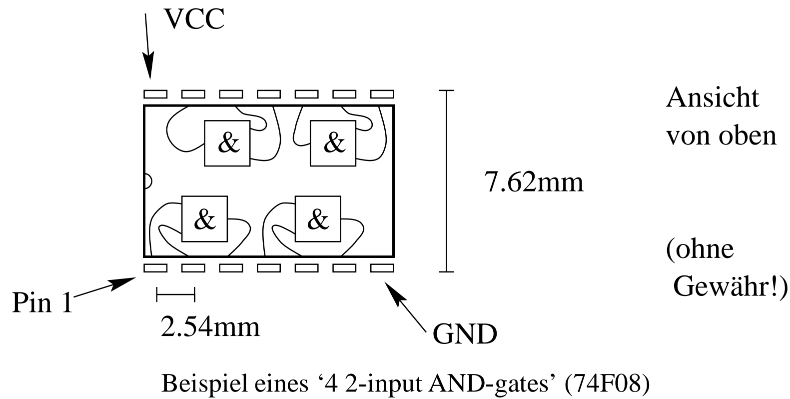
B Betriebsbedingungen, insbesondere Temperaturbereich, z. B. 74 steht für kommerziellen Temperaturbereich von 0 bis 70 Celcius, oder 54 für militärischen Temperaturbereich von -55 bis 125 Celcius.

T Herstellungsprozess und damit implizit Hersteller, hier eine kleine Auswahl: LS 'low power schottky', ALS 'advanced low power schottky', ACT 'advanced CMOS technology', PCT 'performance CMOS technology', F 'FAST fairchild advanced schottky technology', FACT 'fairchild advanced CMOS technology', CY 'cypress', XR 'Exar', HN 'hitachi' usw.

N Nummer nach standardisierter Vereinbarung (damit man die Chips in den Katalogen findet): z. B. 08 für '4 2-input AND-gate', 04 für '6 NOT-gate' oder 00 für '4 2-input NAND-gate'.

P Gehäuseinformation ('package type'), häufig nicht auf dem Chip geschrieben, z. B. DIP 'dual inline plastic'.

t Speziell für Register und Speicherbausteine Schaltzeit, z. B. PAL22V10-7 für 'programmable array logic in 7ns technology'. (Diese Angabe steht auch häufig vor der Gehäusespezifikation.)

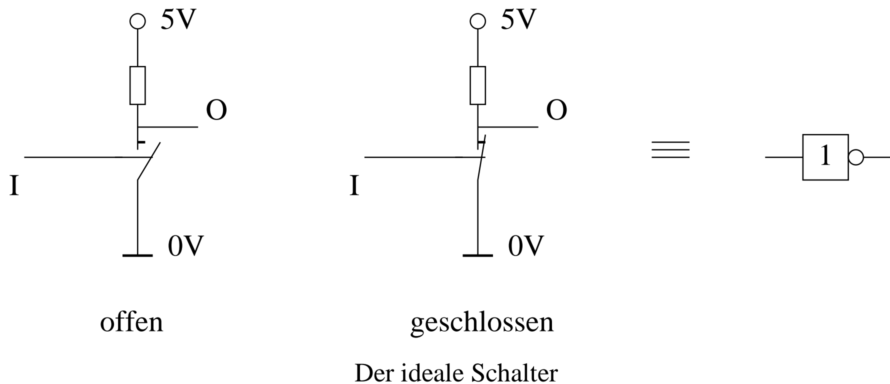


Beispiel eines ‘4 2-input AND-gates’ (74F08)

Hier muss man die Datenbücher studieren, die auch von vielen Herstellern im Internet zu finden sind.

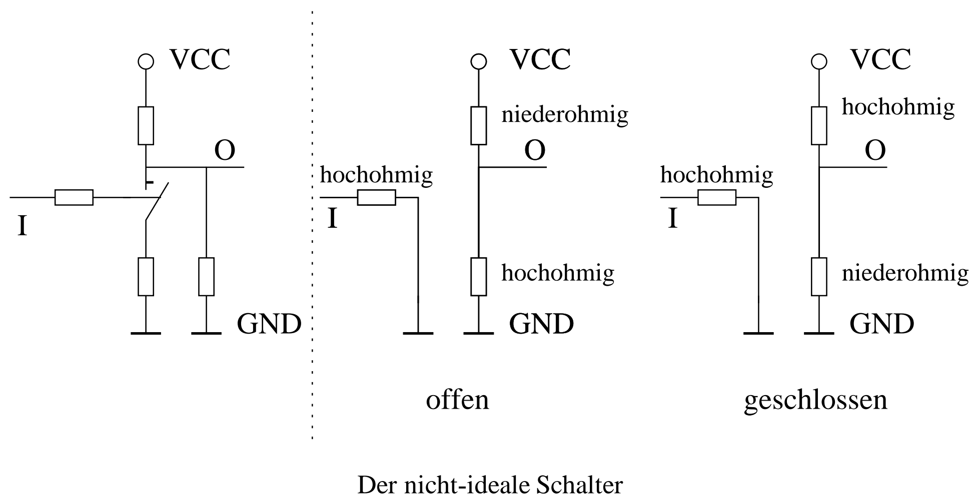
8.2 Parameter zur Zusammenschaltung

Wiederholung des idealen Schalters:



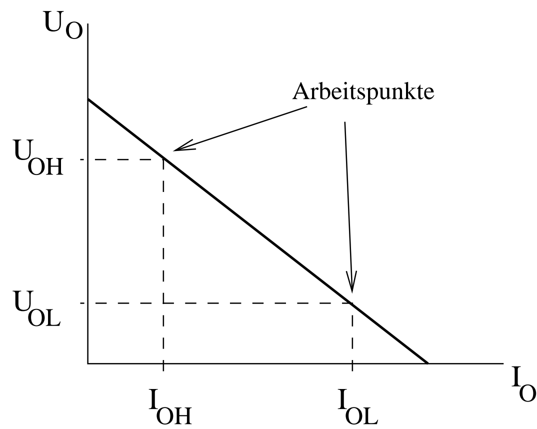
Der ideale Schalter

d. h. die logischen Werte 0 („offen“) und 1 („geschlossen“) am Eingang, sind 0 (0V) und 1 (5V) am Ausgang zugeordnet. Mithilfe von Relais/Spulen kann man auch „offen“ und „geschlossen“ durch Spannungen bzw. Ströme erreichen *oder* mithilfe von Transistoren durch Ausnutzung des strom- oder spannungsgesteuerten Transistoreffektes. Leider kann man mit dem Transistoreffekt keinen idealen Schalter realisieren. Wir modellieren deshalb einen *nicht-idealen* Schalter:



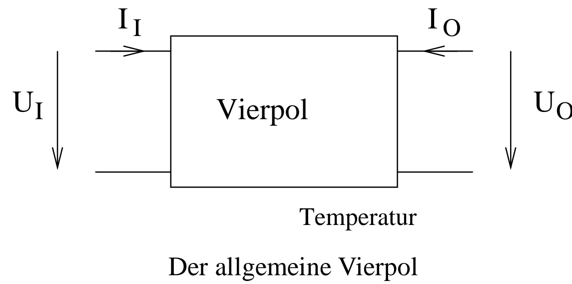
Der nicht-ideale Schalter

Wir erhalten (sehr vereinfacht und nur zur Anschauung gedacht) eine Widerstandskennlinie des Schalters:



(Prinzip-) Widerstandskennlinie des nicht-idealen Schalters

Diese Interpretation ist immer noch sehr modellhaft. Die eingezeichneten Widerstände sind spannungs–, frequenz– und temperaturabhängig, d. h. man müsste eigentlich mit allgemeinen Vierpolen rechnen.

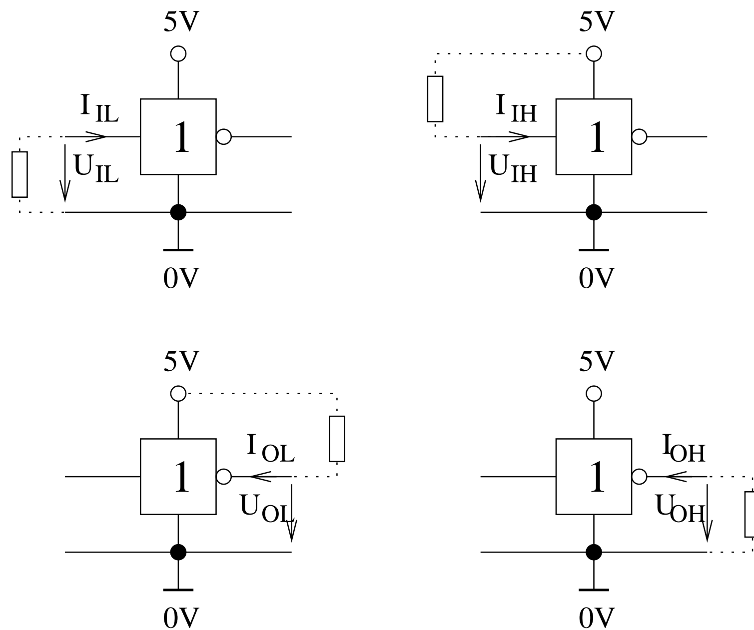


Der allgemeine Vierpol

Der allgemeine Vierpol realisiert eine Funktion

$$Vierpol(U_I, I_I, T) = (U_O, I_O)$$

die aus der Eingangsspannung und dem Eingangsstrom sowie der Temperatur eine Ausgangsspannung und einen Ausgangsstrom berechnet. Die folgende Vereinfachung soll uns jedoch aus der Sichtweise eines Informatikers genügen.



Klassifizierung der Eingangs– und Ausgangsgrößen

Typische Werte der Eingangs– und Ausgangsstrom– und –spannungswerte für TTL Gatter, die immer noch am meisten eingesetzt werden, werden beispielsweise wie folgt angegeben. Man beachte die definierten Richtungen und damit die Vorzeichen der Werte. Die Spannung VCC ist die Versorgungsspannung und wird über dafür vorgesehene Pins an alle Chips geführt.

VCC	$=$	5V	GND	$=$	0V
U_{IL}	$\leq$	0.8V	U_{OL}	$\leq$	0.4V
I_{IL}	$\geq$	-1.6mA	I_{OL}	$\leq$	16mA
U_{IH}	$\geq$	2.4V	U_{OH}	$\geq$	2.4V
I_{IH}	$\leq$	40μA	I_{OH}	$\geq$	-0.4μA

Bemerkung: jede TTL-Technologie hat geringfügig andere Werte, siehe Datenbuch! Neuerdings gibt es auch Bausteine, die mit $VCC = 3V$ arbeiten.

Der Tabelle kann man entnehmen, dass der maximale ‘fan out’, d. h. die Anzahl der Eingänge, mit denen man *einen* Ausgang verbinden kann, 10 beträgt (Vergleich der Eingangs– und Ausgangsströme und Anwendung der KIRCHHOFF’schen Gesetze), d. h. es können höchsten 10 Eingänge von Gattern an einen Ausgang angeschlossen werden. Die angegebenen Werte sind typische Werte, einige spezielle Treiberbausteine erlauben größere Lasten, d. h. sie liefern größere Ströme I_{OL} und I_{OH}.

Ist der Spannungspegel an einem Eingang kleiner als U_{IL}, so sagt man auch: das Signal ist ‘low’; ist er größer als U_{IH}, so sagt man: das Signal ist ‘high’. ‘low’ und ‘high’ werden dann als logische Werte interpretiert. Analog gilt dies für Ausgangssignale. Man kann Bausteine miteinander verbinden, sofern für die entsprechenden Ein– und Ausgänge gilt:

$$U_{OL} \leq U_{IL} \quad \text{und} \quad U_{OH} \geq U_{IH}$$

Wie bereits gesagt, kann man nicht einfach mit Widerstandsnetzen argumentieren, sondern muss mit allgemeinen Vierpolen rechnen. Tatsächlich beschränkt man sich jedoch in den meisten Fallen auf RC-Netze. Für die dynamischen Eigenschaften (Zeitverhalten) eines Gatters (und einer Leitung) sind besonders die Kapazitäten wichtig. Wir wollen hier nicht genauer auf diesen Teil der Elektrotechnik eingehen, sondern beschränken uns auf eine einfache Sichtweise, die aus dem Studium der Datenbücher hervorgeht.

Bemerkung: Statt mit Strömen und Spannungen zu rechnen, benutzt man den Begriff der Einheitslast, der im Wesentlichen die zu treibende Kapazität eines Eingangs oder eines Ausgangs spezifiziert.

Zusammenfassung der charakteristischen Größen einiger TTL Varianten (Angaben sind typische Werte für ein Gatter):

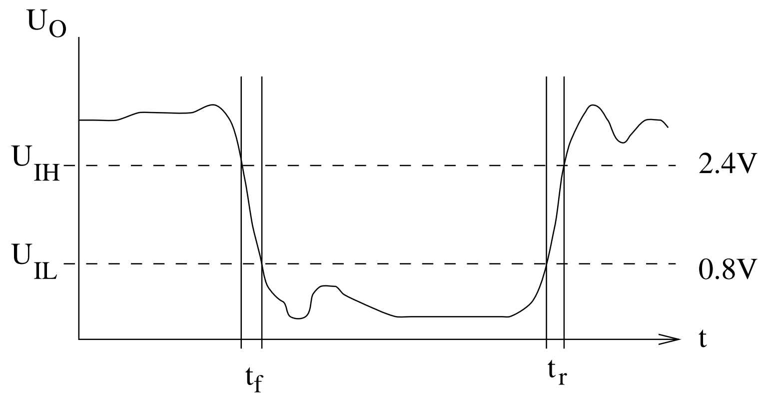
	‘standard’ TTL	‘low’ ‘power’ TTL	‘high’ ‘speed’ TTL	‘schottky’ TTL	‘low power’ ‘schottky’ TTL
Kennbuchstabe		L	H	S	LS
Leistungsaufnahme	10mW	1mW	23mW	20mW	2mW
Schaltzeit	10ns	33ns	5ns	3ns	9.5ns
max. Schaltfrequenz	50MHz	3MHz	80MHz	130MHz	50MHz

Sehr viele aber *nicht alle* Bausteine TTL Familien lassen sich miteinander verwenden. **Studium der Datenbücher!** Andere Technologien haben oft andere Werte! Wir verwenden im Folgenden stets als Beispiele Daten aus dem Katalog der FAST Bausteine von Fairchild.

Betrachten wir nun das Zeitverhalten der TTL Gatter genauer.

9 ‘timing’-Analyse

Motivation: Analysieren des Zeitverhaltens und damit der Korrektheit einer Schaltung

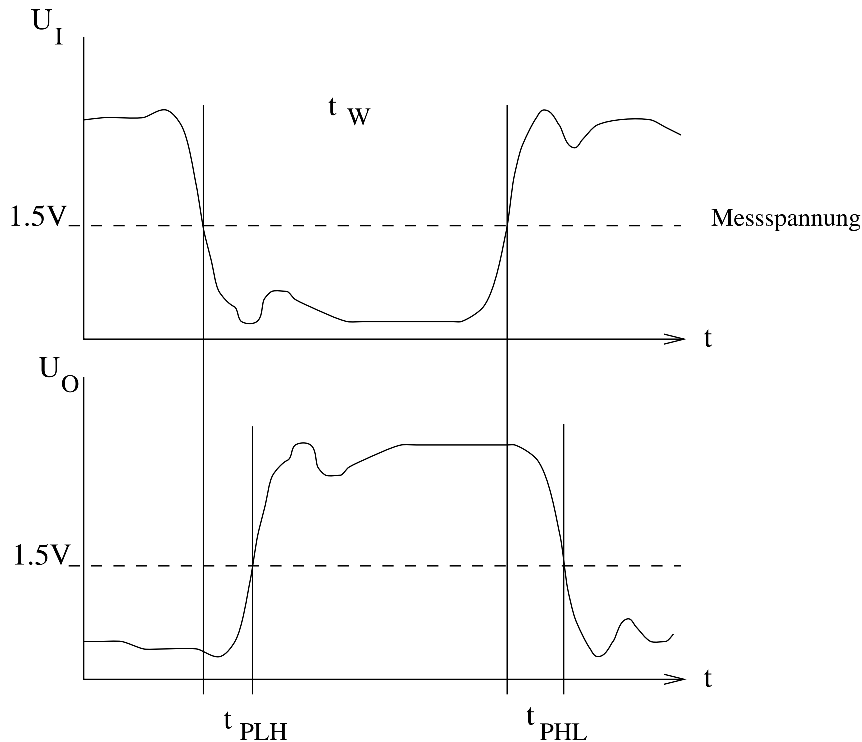


Dynamisches Verhalten eines Signals

t_r	Anstiegszeit	‘rise time’
t_f	Abfallzeit	‘fall time’

Die Anstiegs– und Abfallzeiten sind die Zeiten, die ein Signal benötigt, um von U_{IL} nach U_{IH} anzusteigen bzw. umgekehrt abzufallen (Beachte: manchmal wird auch ein Anstieg bzw. Abfall auf 90% bzw. 10% gefordert). Die maximal zulässigen Zeiten sind ebenfalls im Datenbuch angegeben (für FAST liegt sie im Bereich von 2–3ns). Werden diese Zeiten an den Eingängen eines Gattern eingehalten, so garantiert der Hersteller, dass sie auch an den Ausgängen gegeben sind, d. h. man muss bei der Zeitanalyse darauf keine Rücksicht nehmen. Hat man es mit einem Signal mit sehr flacher steigender oder fallender Flanke zu tun, so kann man spezielle sogenannte Schmidt-Trigger-Bausteine verwenden, die die flachen Eingangsflanken in steile Ausgangsflanken transformiert.

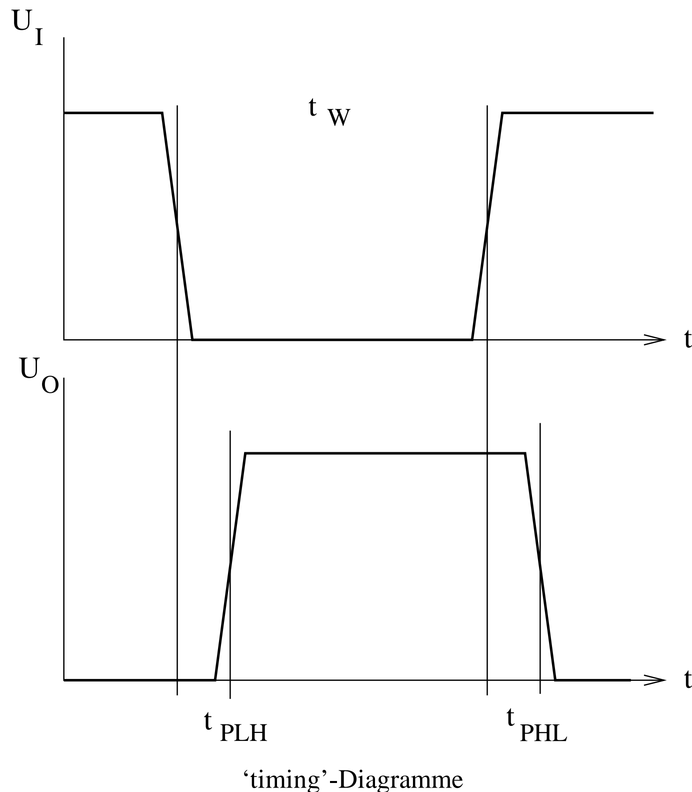
Betrachten wir nun sowohl den Eingang als auch den Ausgang eines Inverters (NICHT-Gatters). Das Ausgangssignal folgt dem Eingangssignal mit einer gewissen Verzögerung. Eigentlich müsste man die Verzögerung bezüglich der Spannungen U_{IL} und U_{OH} bzw. U_{IH} und U_{OL} angeben, aus praktischen Gründen misst man die Verzögerungszeiten jedoch sowohl bei der steigenden als auch bei der fallenden Flanke bei 1.5V. Da t_r und t_f klein sind und nicht propagieren, macht man dabei nur einen geringen Fehler bezüglich der tatsächlichen logischen Werte.



Verzögerungszeiten

t_{PHL}	Verzögerungszeit fallendes Signal	'propagation delay high low'
t_{PLH}	Verzögerungszeit steigendes Signal	'propagation delay low high'
t_W	Pulsweite	'pulse width'

Das heißt, wir beachten nicht die Flanken und betrachten die Verzögerungszeiten nur bezüglich der 1.5V Marke. Der Index bezieht sich stets auf das *Ausgangssignal*. Wir zeichnen vereinfachend sogenannte 'timing'-Diagramme wie folgt:



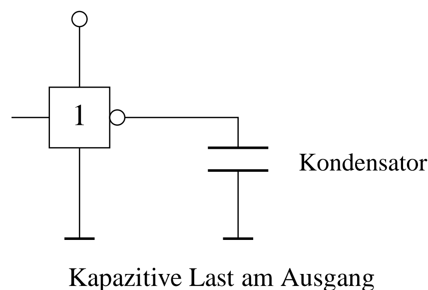
'timing'-Diagramme

Man beachte jedoch, dass es in einigen Fällen notwendig sein kann, doch genauer zu rechnen, insbesondere dann, wenn man Bausteine verschiedener Technologien miteinander verwendet oder externe Signal anschließt.

Die Verzögerungszeiten t_{PHL} und t_{PLH} hängen von mehreren Faktoren ab. Im Wesentlichen sind dies:

- der tatsächlichen Versorgungsspannung,
- der kapazitiven Last am Ausgang,
- der Betriebstemperatur,
- und den Bedingungen bei der Fertigung des Chips.

Die kapazitive Last ergibt sich dadurch, dass der Ausgang über Leitungen mit anderen Eingängen verbunden ist. Sowohl die Leitungen als auch die Eingänge wirken wie Kapazitäten, die „aufgeladen“ bzw. „entladen“ werden müssen. Auch dieses ist nur ein vereinfachtes Modell!



Kapazitive Last am Ausgang

In Datenbüchern werden Verzögerungszeiten nur in Intervallen spezifiziert, die die Variation des Herstellungsprozesses berücksichtigen, sowie wenn bestimmte Annahmen über die Betriebstemperatur, die zu treibende Last und die Versorgungsspannung gemacht werden! Im Einzelfall muss/sollte eine genaue Analyse durchgeführt werden.

So gilt beispielsweise näherungsweise für die Verzögerungszeit t_P bezüglich einer kapazitiven Last C, die von der Standardlast C_0 abweicht:

$$t_P(C) = t_P(C_0) + \alpha \cdot (C - C_0)$$

wobei für FAST gilt: $\alpha \approx 0.03ns/pF$. Die kapazitive Last berechnet sich (unter Vernachlässigung der Leitungen) als die Summe der Kapazitäten der zu treibenden Eingänge. Für FAST gilt meist für die Kapazität eines Eingangs: 4–5pF.

Beispiele für Verzögerungszeiten (FAST Bausteine) bei $VCC = 5V$, 25^oC und Last am Ausgang $= 50pF$:

	NAND 74F00		NOT 74F04		AND 74F08		OR 74F32		XOR 74F86	
t_{PLH}	2.4	6.0	2.4	6.0	3.0	6.6	3.0	6.6	3.5	8.0
t_{PHL}	1.5	5.3	1.5	5.3	2.5	6.3	3.0	6.3	3.0	7.5

Man beachte: muss man eine Verzögerungszeit *nach unten* abschätzen, z. B. bei der Berechnung von Haltezeiten (siehe nächsten Abschnitt), so ist man eventuell auf eine genaue Rechnung mit Kapazitäten angewiesen! So ergibt sich z. B.: die Standardlast C_0 wird bei FAST mit 50pF angegeben; verbindet man nur einen Eingang mit einer kurzen Leitung an einen Ausgang, so ergibt sich mit obiger Gleichung:

$$t_P(C) = 2.4 + 0.03 \cdot (5 - 50) = 1.05$$

Die minimale Verzögerungszeit eines NAND-Gatters $t_P LH$ verkürzt sich also auf 1.05ns.

9.1 Rechnen mit Intervallen

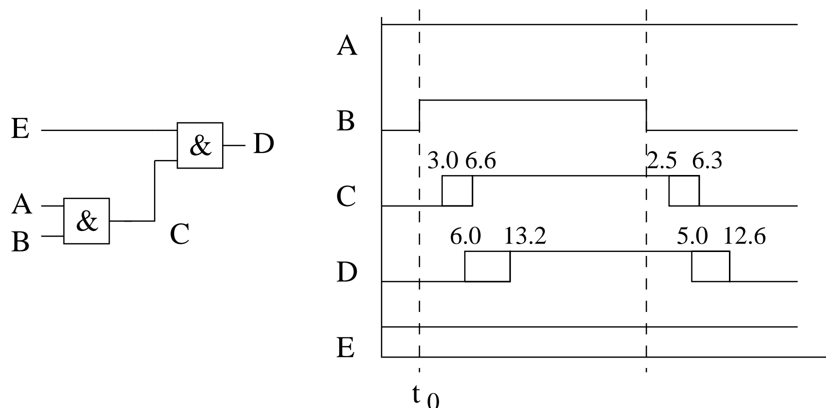
Um nun *exakte Zeitanalysen* durchführen zu können, rechnen wir mit Intervallen (a, b), mit $a, b \in \mathbb{R}$ und $a \leq b$.

Wir sagen, ein Signal schaltet „zur Zeit“ $t = (a, b)$, wenn es frühestens zum Zeitpunkt a und spätestens zum Zeitpunkt b schaltet. Es gelten die folgenden Regeln:

$$\begin{aligned} \min(a, b) &= a \\ \max(a, b) &= b \\ (a, b) + (c, d) &= (a + c, b + d) \end{aligned}$$

9.2 Detailliertes ‘timing’-Diagramm

Mit obiger Tabelle können wir die folgende Schaltung analysieren und erhalten das entsprechende detaillierte ‘timing’-Diagramm. Nehmen wir hierzu an, dass die Signale A und E konstant ‘high’ bleiben und dass das Signal B seinen Zustand zum Zeitpunkt t_0 von ‘low’ nach ‘high’ ändert.



Beispiel eines Schaltkreises mit detailliertem ‘timing’-Diagramm

Das Signal C ändert sich von 'low' nach 'high' zur Zeit

$$t_1 = t_0 + (3.0, 6.6)$$

und damit Signal D ebenfalls von 'low' nach 'high' zur Zeit

$$t_2 = t_1 + (3.0, 6.6) = t_0 + (3.0, 6.6) + (3.0, 6.6) = t_0 + (6, 13.2)$$

Entsprechendes gilt für die eingezeichnete Änderung von 'high' nach 'low'. Im 'timing'-Diagramm schreiben wir nicht die absoluten Zeiten, sondern das Intervall zum verursachenden Zeitpunkt (hier t_0). Dieser Zusammenhang ist oft im Diagramm durch entsprechende Pfeile gekennzeichnet.

Neben den Intervallgrenzen sind in vielen Datenbüchern auch noch typische Werte angegeben. Es ist aber nicht beschrieben, inwieweit diese Werte typisch sind und *welche Gefahren* damit verbunden sind. **Wir ignorieren diese Werte deshalb.**

Wir untersuchen die Schaltung nochmals, allerdings unter weniger spezifischen Voraussetzungen.

Zum Zeitpunkt t_0 ändere sich irgendeins der Signale A, B oder E, vielleicht aber auch mehrere gleichzeitig. Weiterhin sei uns egal, ob sie sich von 0 nach 1 oder von 1 nach 0 ändern.

Damit ergibt sich für das Signal C folgendes Intervall zum Umschalten:

$$t_1 = t_0 + (\min(t_{PLH}, t_{PHL}), \max(t_{PLH}, t_{PHL})) = t_0 + (2.5, 6.6)$$

und damit gilt für die Zeit, wann sich D ändert:

$$t_2 = t_1 + (2.5, 6.6) = t_0 + (5.0, 13.2)$$

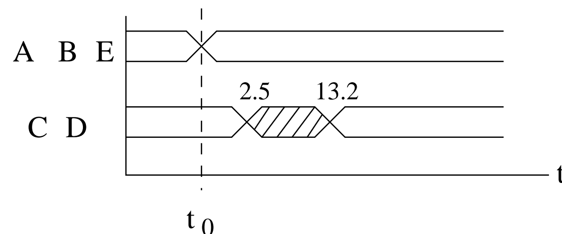
Es könnte sich aber auch E geändert haben, deshalb besteht für D auch die Möglichkeit einer Änderung zur Zeit:

$$t_2 = t_0 + (2.5, 6.6)$$

also gilt für eine mögliche Änderung von D (als Ausgangssignal der Schaltung)

$$t_2 = (2.5, 13.2)$$

und wir zeichnen dies als:



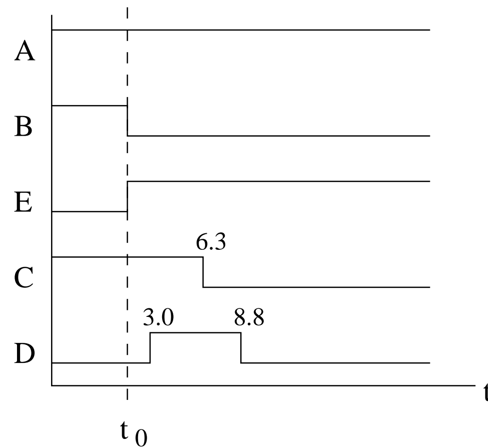
Vereinfachung der 'timing'-Analyse

Diese Vereinfachung verursacht manchmal kleine Probleme ('hazards').

Im Intervall $(2.5, 13.2)$ können sehr interessante Dinge passieren.

Beispiel: zur Zeit t_0 ändern sich A, B, E gleichzeitig von 1,1,0 nach 1,0,1. Man erwartet (naiverweise) für D, dass es konstant 0 bleibt (unter Anwendung der formalen Regeln. Tatsächlich kann aber folgendes auftreten:

Wir erhalten folgendes 'timing'-Diagramm:



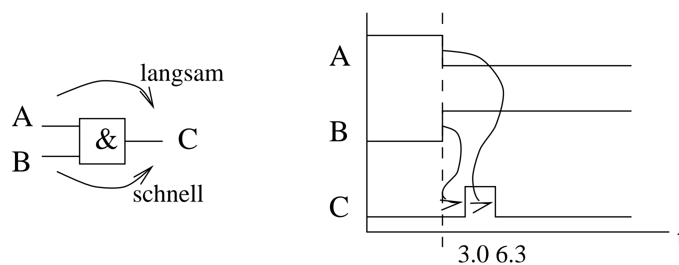
Entstehung eines ‘spikes’ auf einem Signal

Dies geschieht, wenn das Gatter für C langsam und das Gatter für D schnell ist.

Diese „unerwarteten“ Änderungen von D nennt man auch ‘spike’ oder ‘hazard’. Sie entstehen erst durch das Zeitverhalten der Gatter.

9.3 ‘hazards’

Betrachten wir den noch einfacheren Fall eines einzigen Gatters:



‘hazard’ am UND-Gatter

Diese Signalform kann vorkommen, muss aber nicht!

Somit merken wir uns: bei gleichzeitigem Ändern beider (einiger) Eingangssignale kann man *keine* Aussage über den genauen Verlauf des Ausgangssignals (der Ausgangssignale) während des Umschaltintervalls treffen.

Wir haben gesehen, dass bereits bei einem einfachen UND-Gatter ‘spikes’ auftreten können, wenn beide Eingänge gleichzeitig ihren Zustand ändern.

Dies ist in bestimmten Situationen sehr unerwünscht. Wir berechnen deshalb die notwendigen Bedingungen für die Zeit, die zwischen der Änderung eines Signals und der Änderung des anderen Signals vergehen muss, damit *kein* ‘spike’ auftritt.

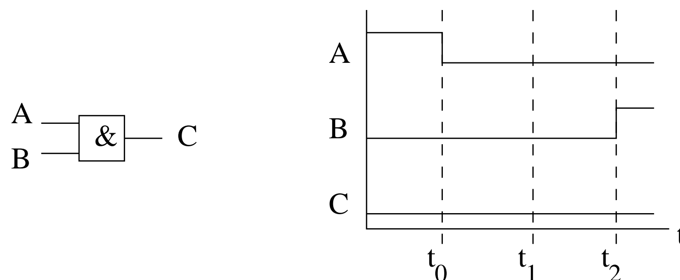
Gemäß Datenbuch wissen wir nichts über das Ausgangssignal eines Gatters, wenn

- beide Eingangssignale zwischen U_{IL} und U_{IH} liegen,
- ein Signal bereits eine Transition durchgeführt hat, diese aber noch nicht zum Ausgang propagiert ist, und das andere Signal seinen Zustand ändert.

Wir gehen wie folgt vor:

- Wir schalten das Signal A zum Zeitpunkt t_0 bezüglich 1.5V von ‘high’ nach ‘low’.
- C bleibt damit ‘low’.

- Intern schaltet das Gatter C nach 'low' bezüglich A jedoch erst nach $t_{PHL} + t_f = (2.5, 6.3) + 2.5$ (wir addieren die 'fall time' t_f um sicher zu gehen, da bei 1.5V U_{IL} noch nicht erreicht ist).
- C ist also spätestens zum Zeitpunkt $t_1 = 8.8$ 'low' *wegen* A 'low'.
- Nun können wir B zum Zeitpunkt $t_2 = t_1 + t_r$ beruhigt anheben, ohne dass sich dies auf C auswirkt (wir addieren auch hier die 'rise time' t_r aus Sicherheitsgründen).
- Die minimale Zeit zwischen einer Änderung von A und B ohne einen 'spike' an C zu erzeugen, beträgt also: $t_2 = 8.8 + 2.5 = 11.3$.



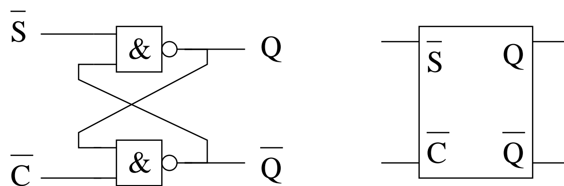
'spike' freies Umschalten beider Eingänge eines UND-Gatters

Bemerkung: die Zeiten t_0, t_1 und t_2 sind bezüglich 1.5V gemessen. Analog erhält man ein minimales Intervall von 11.0ns zum 'spike' freien Umschalten eines NAND-Gatters. Hier bleibt dann der Ausgang sicher 'high'.

Die exakte 'timing'-Analyse ist das einzige Mittel, für eine Schaltung ein bestimmtes physikalisches Verhalten bezüglich eines gewünschten logischen Verhaltens zu garantieren, sofern die verwendeten Daten vom Hersteller garantiert werden. In vielen Fällen erspart eine eher „konservative" Sichtweise spätere Probleme.

9.4 Speichernde Schaltungen

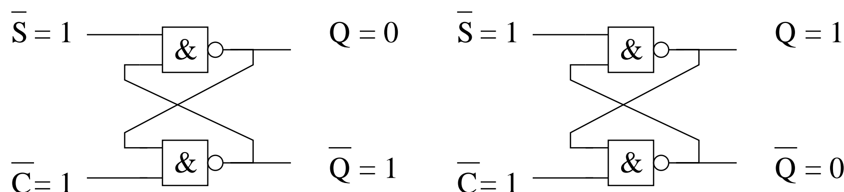
9.4.1 R/S-Flipflop



R/S-Flipflop

Wir wollen nun speichernde Schaltungen untersuchen. Diese lassen sich *nicht* als Schaltkreise beschreiben, da sie Zykel enthalten. Wir haben die obige Schaltung bereits gesehen und festgestellt, dass sie keinen Schaltkreis darstellt (nicht jedem Gatter ist eindeutig eine Tiefe zuzuordnen).

Die Schaltung kann jedoch für die Eingangsbelegung $\overline{S} = 1$ und $\overline{C} = 1$ zwei stabile Zustände annehmen:



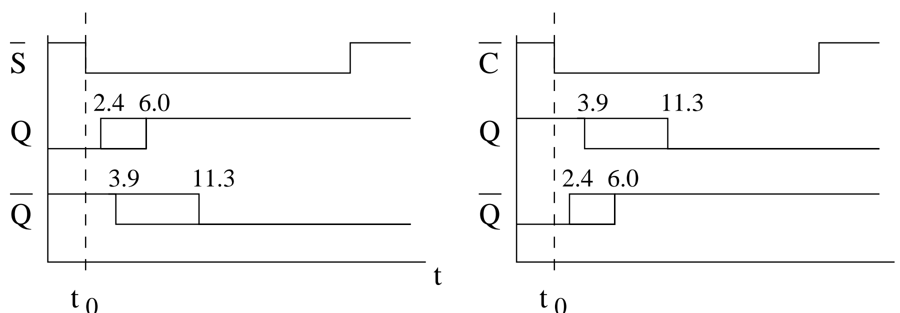
Stabile Zustände eines R/S-Flipflops

Den linken Zustand nennt man naheliegenderweise „Zustand $Q = 1$" und den rechten „Zustand $Q = 0$".

Hält man gewisse Bedingungen für die Eingangssignale $\overline{C}$ und $\overline{S}$ ein, so kann man das Flipflop kontrolliert von einem in den anderen stabilen Zustand umkippen lassen.

Betrachten wir den Fall, wie wir das Flipflop vom Zustand $Q = 0$ in den Zustand $Q = 1$ umschalten.

Sei das Flipflop im Zustand $Q = 0$ und beide Eingänge 'high'. Wir schalten $\overline{S}$ zum Zeitpunkt t_0 nach 'low'. Damit schaltet nach einer Zeit Q nach 'high' und weiter danach $\overline{Q}$ nach 'low'. Dabei müssen wir darauf achten, dass $\overline{S}$ in einem ausreichend langen Intervall 'low' bleibt, danach hat eine Änderung von $\overline{S}$ keinen weiteren Einfluss auf den Zustand des Flipflops.



'timing'-Analyse des R/S-Flipflops

Berechnen wir die minimale Breite des Intervalls, indem $\overline{S}$ 'low' gehalten werden muss.

$\overline{Q}$ schalten zur Zeit $(3.9, 11.3)$ nach 'low'. Damit befindet sich das obere NAND-Gatter in der vorher detailliert analysierten Situation, dass sich ein Signal ändert und wir garantieren wollen, das sich diese Änderung erst auf den Ausgang auswirkt, bevor wir den anderen Eingang ändern wollen.

Hierfür hatten wir 11ns als minimale „Stillhaltezeit" berechnet. Somit dürfen wir $\overline{S}$ frühestens nach $\max(3.9, 11.3)+11 = 22.3$ns nach Initiieren des Setzens wieder ändern.

Aus Symmetriegründen gilt die gleiche Überlegung für den anderen Fall, nämlich Schalten in Zustand $Q = 0$.

Wir schreiben die Signalnamen mit einem Querstrich, da es sich um sogenannte 'active low' Signale handelt; sie bewirken eine Aktion aus, wenn sie 'low' sind. Das Schalten von $Q = 1$ heißt auch *Setzen* ('set') des Flipflops, das Schalten von $Q = 0$ heißt auch *Löschen* ('clear') des Flipflops (das R kommt von 'reset' statt 'clear').

Da Signalform von $\overline{S}$ bzw. $\overline{C}$ nennt man auch *Puls* (hier genau 'active low pulse').

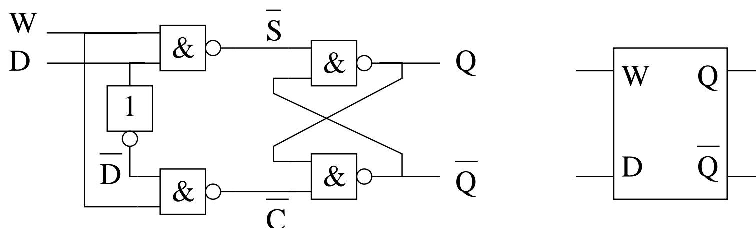
Zusammengefasst erhalten wir die folgenden Zeiten:

Symbol	Bezeichnung	min	max
$t_{W\overline{S}}$	Pulsweite für $\overline{S}$	22.3	
$t_{W\overline{C}}$	Pulsweite für $\overline{C}$	22.3	
$t_{P\overline{S}Q}$	Verzögerungszeit von $\overline{S}$ bis Q	2.4	6.0
$t_{P\overline{S}\overline{Q}}$	Verzögerungszeit von $\overline{S}$ bis $\overline{Q}$	3.9	11.3
$t_{P\overline{C}Q}$	Verzögerungszeit von $\overline{C}$ bis Q	3.9	11.3
$t_{P\overline{C}\overline{Q}}$	Verzögerungszeit von $\overline{C}$ bis $\overline{Q}$	2.4	6.0

9.4.2 D-Latch

Das R/S-Flipflop erlaubt das Speichern eines Bits, allerdings nur wenn vorher bekannt ist, welcher Wert gespeichert werden soll (man muss nämlich wissen, ob gesetzt oder gelöscht werden soll).

Möchte man ein Bit speichern – sei es 0 oder 1 –, so muss man die beiden Signale $\overline{S}$ bzw. $\overline{C}$ aus dem Datenbit berechnen. Dies bewerkstelligt man mit der folgenden Schaltung:



D-Latch

Das sogenannte Schreibsignal W (‘write’) ist ‘active high’. Ist es ‘low’, so sind beide Signale $\overline{S}$ und $\overline{C}$ ‘high’ und das R/S-Flipflop ändert seinen Zustand nicht.

Wird W aktiv, so bewirkt dies, dass bei $D = 1$ das Signal $\overline{S}$ und bei $D = 0$ das Signal $\overline{C}$ aktiviert (sprich ‘low’) wird. Somit wird das Datenbit in das R/S-Flipflop übernommen.

Hierbei müssen die folgenden Bedingungen an W und D eingehalten werden:

$\overline{D}$ darf sich 11ns vor der steigenden Flanke von W nicht ändern, rechnet man die Schaltzeit des NOT-Gatters hinzu, ergibt sich eine sogenannte *setup*-Zeit von 17ns, die angibt, wie weit vor Aktivierung des Schreibpulses nach einer Änderung der Daten verzögert sein muss.

Die Pulsbreite des Schreibsignals ergibt sich aus der Minimalbreite eines Pulses (siehe vorherigen Abschnitt) für das R/S-Flipflop plus die Schaltzeiten der vorgeschalteten NAND-Gatter. Sie beträgt also $22.3 - 2.4 + 5.4 = 25.2$.

Auch nachdem das Schreibsignal W wieder auf 0 gesetzt wird, muss das Datenbit eine zeitlang konstant gehalten werden, damit kein ‘spike’ entsteht. Die bekannte Bedingung verlangt mindestens 11ns.

Die Auswirkung des Schreibsignales auf den Ausgang Q stellt sich, wie leicht zu sehen ist, nach $(3.9, 16, 6)$ ein.

Desweiteren rechnet man leicht nach, dass während $W = 1$ ist, das Latch „transparent“ arbeitet, d. h. jede weitere Änderung des Datenbits propagiert im Intervall $(3.9, 22.6)$ nach Q.

Rechnen Sie dies nach!

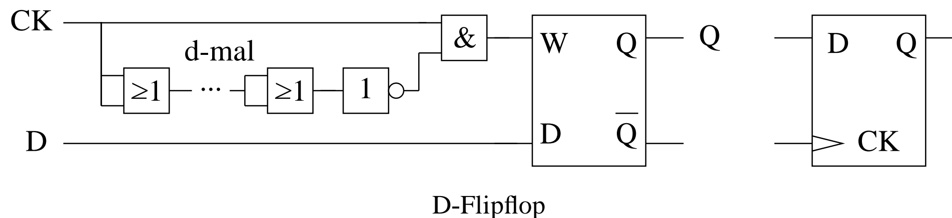
9.4.3 D-Flipflop

Das D-Latch wurde mithilfe eines Pulses gesteuert.

Man kann die Steuerung, d. h. die Übernahme des Datenbits in das R/S-Flipflop, mit einer einfachen Flanke, einem sogenannte ‘clock’-Signal realisieren.

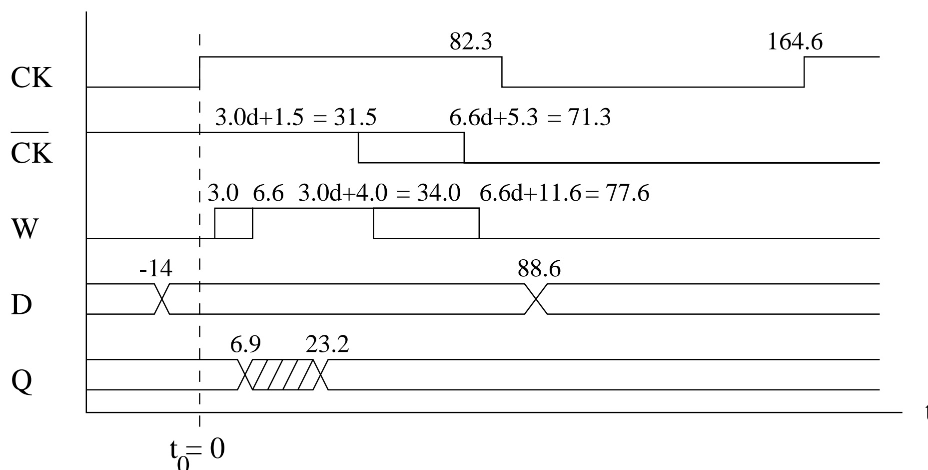
Das Datenbit muss nun einige Bedingungen vor und nach der Flanke einhalten.

Die Schaltung eines D-Flipflops kann man beispielsweise wie folgt aufbauen:



D-Flipflop

Damit ergibt sich das folgende ‘timing’-Diagramm:



‘timing’-Diagramm eines D-Flipflops

Um die minimale Pulsweite des Schreibsignals zu garantieren, müssen 10 OR-Gatter eingesetzt werden, damit die Pulsbreite des Schreibsignales des Latches mindestens 25.2ns beträgt.

Die 'hold time', also die Zeit, in der das Datenbit nach steigender 'clock'-Flanke konstant gehalten werden muss, ist wegen der großen „Unsicherheit" der OR-Gatter, den Durchgangszeiten durch das NOT-Gatter und das AND-Gatter, sowie die notwendige „Stillhalte"-Zeit von 11ns, recht groß und beträgt $10 \cdot 6.6 + 11.6 + 11 = 88.6$

Die Zeit, wann der Ausgang Q das Datenbit nach der Flanke übernommen hat, ist größer als beim Latch, da zusätzlich das AND-Gatter durchlaufen werden muss, und beträgt somit: (6.9,23.2)

Die 'setup'-Zeit verringert sich gegenüber dem Latch jedoch um die minimale Verzögerung des AND-Gatters von 17 auf 14ns.

Obwohl die Steuerung des D-Flipflops mit der steigenden Flanke erfolgt, darf das 'clock'-Signal nicht an beliebiger Stelle wieder abgesenkt werden. Man rechnet leicht die Bedingung nach, dass dies frühestens nach 82.3ns geschehen darf. Rechnen Sie dies nach!

Setzen wir eine symmetrische 'clock' voraus, d. h. die Pulsbreite sowohl für 'high' als auch für 'low' gleich, so kann das D-Flipflop mit einer Frequenz von 6.1 MHz $(1/164 \cdot 10^{-9}$s) getaktet werden.

Beachte: 'hold times' sind Minimalzeiten und müssen eventuell unter Berücksichtung der Kapazitäten berechnet werden.

Kommerzielle Bausteine haben wesentlich geringer Zeiten als hier angegeben. (Datenbücher!)

Signale auf denen man 'hazards' erwartet, dürfen nicht als 'clock'-Signale verwendet werden, da es praktisch unmöglich ist, zu garantieren, dass die notwendigen Bedingungen eingehalten werden können.

Eine interessante Forschungsrichtung beschäftigt sich mit dem Minimieren der Schaltungsvorgänge innerhalb der ICs, insbesondere innerhalb kombinatorischer Schaltkreise, da ICs praktisch nur während des Schaltens Energie benötigen und durch eine Verringerung der Schaltvorgänge die Verlustleistung verkleinert werden kann.