Kapitel 1

Einleitung

Wir haben im vorigen Semester die Programmiersprache PROSA eingeführt. Das ist eine Pascal-ähnliche Sprache. Viele der Hörer haben schon einmal mit einer solchen Sprache gearbeitet. Wir haben eine mathematische Methode kennengelernt, die es erlaubt, zu beschreiben, was korrekte PROSA-Programme sind und was ein PROSA-Programm bei einer bestimmten Eingabe ausrechnet. Gelehrt gesprochen haben wir die Syntax und Semantik von PROSA mit mathematischen Mitteln beschrieben.

Hat man dem Rechner ein korrektes PROSA-Programm und passende Daten eingegeben, so rechnet er das durch die Semantik vorgeschriebene Ergebnis aus. Aber wie macht er das? Der Rechner selbst ist ja ein äußerst primitives Gerät, das nur wenige elementare Operationen ausführen kann, wie z.B. Lesen von ganzen Zahlen aus Eingabe und Speicher, deren Addition, Subtraktion und so weiter. Der Rechner kann darum das Programm auch nicht direkt verstehen und verarbeiten. Er muß es erst in eine Primitivsprache, die sogenannte *Assemblersprache* übersetzen, in der es nur wenige sehr einfache Befehle gibt. Als Befehlsvorrat in der Assemblersprache könnte man die Befehle nehmen, die der Rechner direkt versteht. Um nicht jedesmal einen neuen Übersetzer schreiben zu müssen, ist es aber günstiger, die Hochsprache in eine standardisierte Assemblersprache zu übersetzen und für diese auf jedem Rechner einen kleinen Interpreter zu schreiben, was sehr einfach ist.

Im ersten Teil dieser Vorlesung führen wir zuerst den Prototyp eines Rechners ein, die P-Maschine, und beschreiben ihren Befehlsumfang. Die Programme, die man mit Hilfe der Befehle der P-Maschine schreiben kann, bilden die standardisierten Assemblerprogramme. Wir zeigen, daß man jedes PROSA-Programm in ein P-Programm übersetzen kann, das bei gleicher Eingabe die gleiche Ausgabe produziert und bis auf einen konstanten Faktor die gleiche Laufzeit hat.

Eine der P-Maschine ähnliche Maschine wird später im Hardwaredesignpraktikum auch wirklich gebaut und im zweiten Teil dieser Vorlesung wird erklärt, wie das geht. Den Rechner selbst muß man nämlich aus noch primitiveren elektronischen Grundbausteinen zusammensetzen, z.B. AND- und OR-Gatter, Flipflops, etc. Daß man mit solchen Elementen Computer bauen kann, zeigt man z.B. mit Hilfe der boolschen Algebra.

Kapitel 2

Übersetzung

Es geht in diesem Abschnitt darum, eine Funktion code zu definieren, die jedem PROSA-Programm p ein Programm code(p) für die primitive P-Maschine zuordnet. Das Programm code(p) arbeitet genauso und bis auf einen konstanten Faktor genauso schnell wie das Programm p selbst. Darüberhinaus wird die Funktion code so definiert, daß sie leicht mechanisch berechnet werden kann.

2.1 Einführung der P-Maschine

Wir beginnen mit der Beschreibung der P-Maschine und ihrer Programme.

Die P-Maschine hat einen linearen Datenspeicher $STORE$ mit $maxstr + 1$ Speicherzellen und einen linearen Programmspeicher $CODE$ mit $codemax + 1$ Speicherzellen. Die Speicherzellen in $STORE$ und $CODE$ sind durchnumeriert. Die Numerierung beginnt mit 0. Für $0 \leq i \leq maxstr$ ist $STORE[i]$ die Speicherzelle in $STORE$ mit der Nummer i und $CODE[i]$ die Speicherzelle in $CODE$ mit Nummer i. Die Zellen von STORE werden als Keller benutzt, in dem z.B. Zwischenergebnisse abgelegt werden können. SP (stack pointer) ist die Nummer der obersten belegten Kellerzelle. PC ist die Nummer der Speicherzelle in $CODE$, in der der nächste auszuführende Befehl steht. Anfangs ist $PC = 0$, da das Programm bei $CODE[0]$ beginnt. Die Funktionsweise der P-Maschine muß man sich so vorstellen: Jeder Befehl des P-Programms belegt eine Zelle im Programmspeicher. Außer bei Sprüngen werden die Befehle einfach sequentiell abgearbeitet.

Jetzt werden die elementaren Operationen erklärt, die der P-Maschine zur Verfügung stehen. Die zweistelligen Operationen werden jeweils auf die Inhalte der beiden obersten Speicherzellen des Kellers angewandt. Das Ergebnis wird in die zweitoberste Speicherzelle des Kellers geschrieben. Danach wird der Stack Pointer um eins erniedrigt. Einstellige Operationen werden auf den Inhalt der obersten Kellerzelle angewandt. Das Ergebnis wird in die oberste Kellerzelle geschrieben. Folgende Tabelle zeigt die Operationen und ihre Wirkung. Viele Operationen sind durch eine Sorte parametrisiert. Dabei steht N für eine numerische Sorte und s steht für eine beliebige Sorte. Außerdem seien $+_N, -_N$ usw. die entsprechenden Operationen für zwei Operanden vom Typ N.

Befehl	Wirkung
add N	$STORE[SP-1] := STORE[SP-1] +_N STORE[SP]; SP := SP-1$
sub N	$STORE[SP-1] := STORE[SP-1] -_N STORE[SP]; SP := SP-1$
mul N	$STORE[SP-1] := STORE[SP-1] *_N STORE[SP]; SP := SP-1$
div N	$STORE[SP-1] := STORE[SP-1] /_N STORE[SP]; SP := SP-1$
neg N	$STORE[SP] := -STORE[SP]$
and	$STORE[SP-1] := STORE[SP-1] \wedge STORE[SP]; SP := SP-1$
or	$STORE[SP-1] := STORE[SP-1] \vee STORE[SP]; SP := SP-1$
not	$STORE[SP] := \neg STORE[SP]$
equ s	$STORE[SP-1] := STORE[SP-1] = STORE[SP]; SP := SP-1$
neq s	$STORE[SP-1] := STORE[SP-1] \neq STORE[SP]; SP := SP-1$
geq N	$STORE[SP-1] := STORE[SP-1] \geq_N STORE[SP]; SP := SP-1$
leq N	$STORE[SP-1] := STORE[SP-1] \leq_N STORE[SP]; SP := SP-1$
grt N	$STORE[SP-1] := STORE[SP-1] >_N STORE[SP]; SP := SP-1$
les N	$STORE[SP-1] := STORE[SP-1] <_N STORE[SP]; SP := SP-1$

Es muß ferner möglich sein, den Datenspeicher zu beeinflussen. Dazu gibt es verschiedene *Ladebefehle*, deren Wirkung wir im einzelnen erklären:

Befehl	Wirkung
ldo s q	$SP := SP + 1; STORE[SP] := STORE[q]$
ldc s q	$SP := SP + 1; STORE[SP] := q$
ind s	$STORE[SP] := STORE[STORE[SP]]$
sro s q	STORE[q] := STORE[SP]; SP = SP - 1
sto s	STORE[STORE[SP - 1]] := STORE[SP]; SP:= SP -2

In dieser Beschreibung steht s wieder für eine Basissorte und q ist eine Speicheradresse, also eine ganze Zahl zwischen 0 und *maxstr*. Da die Ladebefehle ungewohnt sind, sollen sie an einem Beispiel illustriert werden.

2.1. Beispiel Angenommen, es ist $SP = 3$ und der Inhalt von $STORE$ hat folgende Gestalt:

Nummer	1	2	3	4	5	6	7	8	9	10
Inhalt	3	2	1	*	*	7	4	6	1	1

Die Speicherzellen enthalten nur Integers und darum verzichten wir im folgenden auf die explizite Sortenangabe.

Der Befehl **ldo** 9 erhöht den Stack Pointer um 1 und lädt den Inhalt der Speicherstelle mit der Nummer 9 in die Speicherzelle mit der Nummer SP. Damit ist $SP = 4$ und der neue Zustand des Datenspeichers sieht so aus:

Nummer	1	2	3	4	5	6	7	8	9	10
Inhalt	3	2	1	1	*	7	4	6	1	1

Der Befehl **ldc** 9 erhöht den Stack Pointer um 1, lädt die Konstante 9 in die Speicherzelle mit der Nummer SP. Damit ist $SP = 5$ und der neue Zustand des Datenspeichers ist so:

Nummer	1	2	3	4	5	6	7	8	9	10
Inhalt	3	2	1	1	9	7	4	6	1	1

Der Befehl **ind** lädt in die Speicherzelle mit der Nummer SP den Inhalt der Speicherstelle mit der Nummer $STORE[SP]$, also mit der Nummer 9. Damit ist immer noch $SP = 5$ und der neue Zustand des Datenspeichers sieht jetzt so aus:

Nummer	1	2	3	4	5	6	7	8	9	10
Inhalt	3	2	1	1	1	7	4	6	1	1

Der Befehl **sro** 1 lädt in die Speicherzelle mit der Nummer 1 den Inhalt der Speicherstelle mit der Nummer SP, also mit der Nummer 9. Anschließend wird der Stack Pointer um 1 erniedrigt. Der neue Zustand des Datenspeichers sieht jetzt so aus:

Nummer	1	2	3	4	5	6	7	8	9	10
Inhalt	1	2	1	1	1	7	4	6	1	1

Außerdem ist jetzt $SP = 4$.

Der Befehl **sto** schließlich lädt den Inhalt der Speicherzelle mit der Nummer SP in die Speicherzelle mit der Nummer $STORE[SP-1]$. Anschließend wird der Stack Pointer um 2 erniedrigt. Der neue Zustand des Datenspeichers ist gleich dem alten aber $SP = 2$.

2.2 Übersetzung von Zuweisungen

Der Übersetzer erzeugt Code, der in den Programmspeicher der P-Maschine geladen und dann bei entsprechender Eingabe abgearbeitet wird.

Angenommen, die Deklarationen des PROSA-Programms sind bereits abgearbeitet, und dabei sind nur Konstanten und gewöhnliche Variable, aber keine Felder, Records oder Prozeduren definiert worden. Sei $\Gamma = (S, \Omega, \varphi, \pi)$ die entsprechende Signatur. Dann ist ja

$$S = S_B \cup S_V$$

mit

$$S_B = \{\text{int, real, bool, char, string}\} \qquad \text{und} \qquad S_V = \{(\text{var}, s) : s \in S_B\}.$$

Die Konstantenmenge ist

$$C = C_B \cup C_V.$$

Hierin ist

$$C_B = \mathbb{Z} \cup \mathbb{R} \cup \{true, false\} \cup \Sigma,$$

wobei Σ der Standard-Zeichensatz ist, und

$$C_V = \{x : \text{Konstante der Sorte } (\text{var}, s),\ s \in S_B\}.$$

Konstanten der Sorte (var, s) nennen wir Variablen der Sorte s. Die Stelligkeiten der Konstanten sind erklärt wie üblich. Die Operationen in Γ sind dann

$$\Omega = C \cup \{+, -, *, /, \text{div}, \bmod, \wedge, \vee, \neg, =, \leq, \geq, <, >, \neq\}.$$

Deren Stelligkeiten,Prioritäten und Bedeutung sind ebenfalls definiert wie üblich. Die U-Terme über Γ waren induktiv definiert:

2.2. Definition *1. Konstanten der Sorte s sind U-Terme der Sorte s und Priorität $p(\Gamma)$.*

2. Ist t ein U-Term der Sorte s und Priorität $p(\Gamma)$ und ist ω eine einstellige Operation der Stelligkeit (s, s'), so ist ωt ein U-Term der Sorte s' und der Priorität $p(\Gamma)$.

3. Sind t_i U-Terme der Sorten s_i und der Prioritäten p_i, $i = 1, 2$, ist ω eine Infixoperation der Stelligkeit $(s_1 s_2, s)$ der Priorität p, gilt $p_1 \geq p$ und $p_2 > p$, so ist $t_1 \omega t_2$ ein U-Term der Sorte s und der Priorität p.

4. Ist t ein U-Term der Sorte s, so ist (t) ein U-Term der Sorte s und Priorität $p(\Gamma)$.

Man beachte, daß in der Signatur Γ keine mehrstelligen Präfixoperationen vorkommen. Im folgenden sagen wir statt U-Term über Γ auch kurz Term.

Wir beschreiben jetzt die Übersetzung der Zuweisung

$$x := t,$$

wobei x eine Konstante der Sorte (var, s) und t ein Term der Sorte s über Γ ist. Bei einer solchen Zuweisung soll folgendes passieren:

Bestimme die Adresse q von x
Berechne den Wert von t
Speichere den Wert von t in die Speicherzelle q

Die Variablenbezeichnung x auf der linken Seite der Zuweisung muß also anders übersetzt werden als die Variablenbezeichnungen auf der rechten Seite, weil man auf der linken Seite an einer Adresse, auf der rechten Seite aber an Werten interessiert ist. Daher übersetzen wir die Zuweisung folgendermaßen:

2.3. Definition *Falls t ein Term der Sorte s ist, so setzen wir*

$$\mathrm{code}(x := t) = \mathrm{code}_L\, x; code_R t; \mathbf{sto}\ s \tag{2.1}$$

Die Funktion code_L wird später im Abschnitt über Adressierung noch genauer erklärt. Wir müssen also jetzt noch die Funktion code_R definieren.

2.4. Definition *Ist $c \in C_B$ eine Konstante der Sorte s, so ist $\mathrm{code}_R\, c = \mathbf{ldc}\ s\ c$.*

Für Variablen y wird $\mathrm{code}_R(y)$ später erklärt, wenn wir über die Speicherorganisiation reden.

2.5. Definition *Auf zusammengesetzten Termen ist die Funktion code_R so erklärt, wie in der folgenden Tabelle beschrieben. Dabei sind t_1, t_2, t U-Terme über Γ der Sorte s. Die Sorten int und real werden als numerische Sorten bezeichnet.*

u	*Bedingung*	$\text{code}_R(u)$
$-t$	s *numerisch*	$\text{code}_R\, t$; **neg** s
$t_1 + t_2$	s *numerisch*	$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **add** s
$t_1 - t_2$	s *numerisch*	$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **sub** s
$t_1 * t_2$	s *numerisch*	$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **mul** s
t_1/t_2	$s =$ *real*	$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **div** *real*
t_1 *div* t_2	$s =$ *int*	$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **div** *int*
$\neg t$	$s =$ *bool*	$\text{code}_R\, t$; **not**
$t_1 \wedge t_2$	$s =$ *bool*	$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **and**
$t_1 \vee t_2$	$s =$ *bool*	$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **or**
$t_1 < t_2$	s *numerisch*	$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **les** s
$t_1 > t_2$	s *numerisch*	$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **grt** s
$t_1 \leq t_2$	s *numerisch*	$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **leq** s
$t_1 \geq t_2$	s *numerisch*	$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **geq** s
$t_1 = t_2$		$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **equ** s
$t_1 \neq t_2$		$\text{code}_R\, t_1$; $\text{code}_R\, t_2$; **neq** s
(t)		$\text{code}_R\, t$

Die Funktion code ist natürlich nur dann wohldefiniert, wenn es eine eindeutige Vorschrift gibt, die bestimmt, ob ein "$-$" in einem Term unär oder binär ist. Daß eine solche Vorschrift, die sogar berechenbar ist, wirklich existiert, zeigen wirn in den Übungen.

Wir illustrieren die Übersetzung an einem Beispiel:

2.6. Beispiel Seien x und y Variablen der Sorte int. Sei $t = x < -(y+4) \wedge x \neq 0$. Der Wert dieses Terms berechnet sich wie folgt:

$$\begin{aligned}
\text{code}_R(t) &= \text{code}_R(x < -(y+4);\ \text{code}_R(x \neq y);\ \textbf{and} \\
&= \text{code}_R(x);\ \text{code}_R(-(y+4));\ \textbf{les}\ \text{int};\ \text{code}_R(x);\ \text{code}_R(y);\ \textbf{neq}\ \text{int};\ \textbf{and} \\
&= \text{code}_R(x); \text{code}_R((y+4)); \textbf{neg}\ int; \textbf{les}\ int; \text{code}_R(x); \text{code}_R(y); \\
&\quad \textbf{neq}\ int; \textbf{and} \\
&= \text{code}_R(x); \text{code}_R(y+4); \textbf{neg}\ int; \textbf{les}\ int; \text{code}_R(x); \text{code}_R(y); \\
&\quad \textbf{neq}\ int; \textbf{and} \\
&= \text{code}_R(x); \text{code}_R(y); \text{code}_R(4); \textbf{add}\ int; \textbf{neg}\ int; \textbf{les}\ int; \text{code}_R(x); \\
&\quad \text{code}_R(y); \textbf{neq}\ int; \textbf{and} \\
&= \text{code}_R(x); \text{code}_R(y); \textbf{ldc}\ int4; \textbf{add}\ int; \textbf{neg}\ int; \textbf{les}\ int; \text{code}_R(x); \\
&\quad \text{code}_R(y); \textbf{neq}\ int; \textbf{and}
\end{aligned}$$

2.3 Anweisungsfolgen

In PROSA ist eine *Anweisungsfolge* eine endliche Folge von der Form $st_1; st_2; \ldots; st_k$, wobei die st_i Anweisungen sind für $1 \leq i \leq k$.

2.7. Definition *Sei $k \in \mathbb{N}$, $st = st_1; st_2; \ldots; st_k$ eine PROSA-Anweisungsfolge. Dann definiert man* $\mathrm{code}\, st = \mathrm{code}\, st_1;\ \mathrm{code}\, st_2;\ \ldots;\ \mathrm{code}\, st_k$.

2.4 Bedingte und iterative Anweisungen

Um IF- oder WHILE-Anweisungen übersetzen zu können, müssen wir im Programmspeicher hin und her springen können. Hierzu benötigen wir neue Befehle, nämlich bedingte und unbedingte Sprünge. Diese Befehle werden in der nächsten Tabelle vorgestellt. In dieser Tabelle steht q für eine Adresse im Programmspeicher $CODE$:

Befehl	Bedeutung
ujp q	$PC := q$
fjp q	falls $STORE[SP] = false$ setze $PC := q$ fi; $SP := SP - 1$

Hierbei steht **ujp** für unconditional jump und **fjp** für false jump.

Damit man mit solchen Befehlen gezielt zu anderen Befehlen springen kann, führt man an der entsprechenden Stelle eine Bezeichnung für die Adressen des anzuspringenden Befehls ein. Zum Beispiel schreibt man statt

ldo int q

die erweiterte Form

n: **ldo** int q.

Der Name n steht dann für die Nummer der Speicherzelle, in der sich dieser Befehl **ldo** int q befindet. Die Wirkung von

ujp n

ist dann, daß dieser Befehl **ldo** *int q* angesprungen wird. Damit man auch das Ende einer Folge von P-Befehlen anspringen kann, ist es möglich, Befehlsfolgen mit $n :$ abzuschließen. Man hat dann im Programmspeicher $CODE$ eine leere Speicherzelle, die man anspringen kann.

Damit kann man jetzt die code-Funktion für IF-Anweisungen definieren.

2.8. Definition *Sei b ein Term der Sorte bool und seien st, st_1, st_2 PROSA-Anweisungsfolgen. Dann definiert man*

$$\mathrm{code}(\mathit{IF}\ b\ \mathit{THEN}\ st\ \mathit{FI}) \quad = \quad \mathrm{code}_R\, b;\ \mathbf{fjp}\ l;\ \mathrm{code}\, st;\ l :$$

und

$$\mathrm{code}(\mathit{IF}\ b\ \mathit{THEN}\ st_1\ \mathit{ELSE}\ st_2\ \mathit{FI}) \quad = \quad \mathrm{code}_R\, b;\ \mathbf{fjp}\ l_1;\ \mathrm{code}\, st_1;\ \mathbf{ujp}\ l_2;\ l_1 : \mathrm{code}\, st_2;\ l_2 :$$

Entsprechend kann man die code-Funktion für WHILE-Schleifen definieren:

2.9. Definition *Sei b ein Term der Sorte bool und sei st eine PROSA-Anweisungsfolge. Dann definiert man*

$$\mathrm{code}(\mathit{WHILE}\ b\ \mathit{DO}\ st\ \mathit{OD}) \quad = \quad l_1 : \mathrm{code}_R\, b;\ \mathbf{fjp}\ l_2;\ \mathrm{code}\, st;\ \mathbf{ujp}\ l_1;\ l_2 :$$

2.5 Adressierung von Variablen

Die Belegung von $STORE$ wird durch eine Funktion

$$\rho : \{\text{Namen}\} \to \mathbb{N} \tag{2.2}$$

geregelt. Diese ordnet den Namen von deklarierten Variablen Nummern von Speicherzellen in $STORE$ zu. Der Compiler legt eine Tabelle für diese Funktion an, die sogenannte *Symboltabelle*. Wir gehen also davon aus, daß für jede Variable genau eine Speicherzelle zur Verfügung steht. Die Wirklichkeit ist aber komplizierter. Da wird aus Speichereffizienzgründen mehreren Variablen der Sorte *bool* ein Speicherwort zugeordnet, während man für Variable der Sorte *int* bei einer Wortlänge von 8 Bit im allgemeinen vier Speicherworte reserviert.

Bei der Übersetzung der Deklarationen werden den deklarierten Variablen der Reihe nach Speicherplatznummern zugeordet. Die erste Nummer, die vergeben wird, ist aus technischen Gründen 5.

2.10. Beispiel Angenommen, der Compiler findet in einem PROSA-Programm die folgenden Deklarationen vor:

VAR x, y, z : int;
VAR t : bool;

Dann hat die ρ-Funktion folgende Werte:

Name	ρ(Name)
x	5
y	6
z	7
t	8

2.11. Definition *Sei x der Name einer deklarierten Variablen der Sorte s. Dann setzen wir*

$$\text{code}_L\, x = \textbf{ldc}\ int\ \rho(x)$$

und

$$\text{code}_R\, x = \textbf{ldo}\ s\ \rho(x).$$

Damit ist jetzt eine vollständige Übersetzung von Zuweisungen und Schleifen möglich.

2.12. Beispiel Angenommen der Compiler muß den folgenden PROSA-Programmteil übersetzen:

VAR x, y, z : int;
VAR t : bool;

$x := 2;$
$y := 2 * x;$
$t := y > x;$

IF t THEN $x := 2 * x$ FI;

Die Wirkung der Deklaration wurde schon in Beispiel 2.10 beschrieben. Das entsprechende P-Programm sieht so aus:

ldc int $\rho(x)$
ldc int 2
sto int
ldc int $\rho(y)$
ldc int 2
ldo int $\rho(x)$
mul int
sto int
ldc int $\rho(t)$
ldo int $\rho(y)$
ldo int $\rho(x)$
grt int
sto bool
ldo bool $\rho(t)$
fjp end
ldc int $\rho(x)$
ldc int 2
ldo int $\rho(x)$
mul int
sto int
end:

Man sieht am vorangegangenen Beispiel noch einmal sehr deutlich, daß die Funktion code von der gewählten Adressierungsfunktion ρ abhängt.

2.6 Adressierung von Feldern

Die Funktion ρ ordnet einem Feldnamen die Anfangsadresse dieses Feldes zu. Dies ist die nächste verfügbare Adresse in $STORE$. Die Zuordnung der Feldelemente zu den Speicherplätzen wird im nächsten Beispiel erläutert.

2.13. Beispiel Angenommen, alle Speicherplätze bis zur Nummer 16 sind bereits vergeben und jetzt findet der Compiler die Deklaration

$$\text{var } a : \text{array}[1..2, 2..4] \text{ of int;}$$

Dann ist $\rho(a) = 17$ und den Feldelementen ordnet der Compiler folgende Speicherplatznummern zu:

Feldelement	$a[1,2]$	$a[1,3]$	$a[1,4]$	$a[2,2]$	$a[2,3]$	$a[2,4]$
Nummer	17	18	19	20	21	22

Die Zuordnung erfolgt also entsprechend einer lexikographischen Ordnung der Indextupel: Einem Indextupel I_1 wird ein höherer Speicherplatz zugeordnet als einem anderen Indextupel I_2, wenn der die erste von Null verschiedene Komponente in dem Differenztupel $I_1 - I_2$ positiv ist. Die Zuordnungsfunktion wird im folgenden Lemma beschrieben.

2.14. Lemma *Sei $k \in \mathbb{N}$, seien $u_i, o_i \in \mathbb{Z}$, $d_i = o_i - u_i + 1 > 0$, $D_i = \prod_{j=i+1}^{k} d_j$ für $1 \leq i < k$ und $D_k = 1$ sowie $D = \prod_{i=1}^{k} d_i$.*

Seien ferner $j_i \in \mathbb{Z}$, $u_i \leq j_i \leq o_i$, $1 \leq i \leq k$. Sei die Funktion

$$\nu : \quad \prod_{i=1}^{k} [u_i, o_i] \rightarrow [0, D-1]$$

definiert durch $\nu(j_1, \ldots, j_k) = \sum_{i=1}^{k} (j_i - u_i) \cdot D_i$. Dann gilt:

Die Funktion ν ist eine Bijektion und für $I_j \in \prod_{i=1}^{k} [u_i, o_i]$, $j = 1, 2$ gilt $\nu(I_1) > \nu(I_2)$ genau dann, wenn der am weitesten links stehende von Null verschiedene Eintrag in $I_1 - I_2$ positiv ist.

Um den Speicherplatzbedarf für den Deklarationsteil feststellen zu können, führen wir zusätzlich noch die Funktion

$$\mathrm{gr} : \{\text{Namen}\} \rightarrow \mathbb{N}$$

zu. Diese Funktion ordnet einem Namen die Anzahl der Speicherplätze zu, die für diesen Namen belegt sind. Beispielsweise gilt in Beispiel 2.13 $\mathrm{gr}(a) = 6$, während im Beispiel 2.10 $\mathrm{gr}(x) = \mathrm{gr}(y) = \mathrm{gr}(z) = \mathrm{gr}(t) = 1$ gilt.

Jetzt können wir die Adressierung von Feldern definieren.

2.15. Definition *Sei $k \in \mathbb{N}$, seien $u_i, o_i \in \mathbb{Z}$, $d_i = o_i - u_i + 1 > 0$, $D_i = \prod_{j=i+1}^{k} d_j$ für $1 \leq i < k$ und $D_k = 1$ und sei a deklariert als*

$$\textit{var } a\textit{: array}[u_1..o_1, \ldots, u_k..o_k] \textit{ of } s;$$

wobei s eine numerische Basissorte ist. Dann ist $\mathrm{gr}(a) = \prod_{i=1}^{k} d_i$. Seien ferner $j_i \in \mathbb{Z}$, $u_i \leq j_i \leq o_i$, $1 \leq i \leq k$. Dann wird dem Feldelement mit der Nummer $j_1, \ldots, j_k$ die Speicherzelle mit der Nummer $\rho(a) + \sum_{i=1}^{k} (j_i - u_i) D_i$ zugeordnet.

Die in Definition 2.15 beschriebene Adressierung von Feldelementen hängt einerseits von *statischen* Größen ab, also von solchen, die bereits zur *Compilezeit* berechnet werden können, weil sie sich zur *Laufzeit* nicht mehr ändern. Eine solche Größe ist die Anfangsadresse des Feldes. Andererseits hängt die Adresse eines Feldelementes aber auch von *dynamischen*, also nur zur Laufzeit bekannten Grösen ab, nämlich vom aktuellen Indextupel. Es lohnt sich daher, die Größen D_i, $1 \leq i \leq k$ und $H = \sum_{i=1}^{k} u_i \cdot D_i$ zur Compilezeit vorzuberechnen, denn es gilt

$$\rho(a) + \sum_{i=1}^{k} (j_i - u_i) \cdot D_i \quad = \quad \rho(a) - H + \sum_{i=1}^{k} j_i \cdot D_i.$$

Die Adressberechnung könnte man jetzt mit den bisher eingeführten Funktionen durchführen. Dabei würde aber der Vorteil, daß einige der benötigten Größen statisch sind, teilweise verschenkt. Man benutzt daher noch folgende Befehle:

2.16. Definition *Sei q eine Konstante der Sorte int. Dann sind folgende Befehle definiert:*

Befehl	*Wirkung*
ixa q	$STORE[SP-1] := STORE[SP-1] + STORE[SP] * q; SP := SP - 1$
inc q	$STORE[SP] := STORE[SP] + q$
dec q	$STORE[SP] := STORE[SP] - q$

Damit sind wir jetzt in der Lage, Feldelemente zu übersetzen.

2.17. Definition *Seien $i_1, \ldots, i_k$ Terme der Sorte int. Dann setzen wir unter den Voraussetzungen von Definition 2.15*

$$\begin{aligned} \mathrm{code}_L\, a[i_1, \ldots, i_k] = \quad & \\ & \mathbf{ldc}\ int\ \rho(a) \\ & \mathrm{code}_R\, i_1;\ \mathbf{ixa}\ D_1 \\ & \mathrm{code}_R\, i_2;\ \mathbf{ixa}\ D_2 \\ & \vdots \\ & \mathrm{code}_R\, i_k;\ \mathbf{ixa}\ D_k \\ & \mathbf{dec}\ H \end{aligned}$$

und

$$\mathrm{code}_R\, a[i_1, \ldots, i_k] = \mathrm{code}_L\, a[i_1, \ldots, i_k];\ \mathbf{ind}\ s.$$

2.7 Ein- und Ausgabe

Um einfache PROSA-Programme vollständig übersetzen zu können, benötigen wir noch Ein- und Ausgabefunktionen.

2.18. Definition *Sei s eine Basissorte. Die Befehle* **inp** *und* **out** *sind folgendermaßen definiert:*

Befehl	*Wirkung*
inp s	$SP := SP + 1$; *Konstante der Sorte s wird gelesen und auf den Stack geschrieben*
out s	$STORE[SP]$ *wird ausgegeben,* $SP := SP - 1$

Damit können wir die Übersetzung von $READ$ und $PRINT$-Anweisungen definieren.

2.19. Definition *Falls x eine Variable der Sorte s ist, setzen wir*

$$\mathrm{code}(READ\ \ x) = \mathrm{code}_L\, x;\ \mathbf{inp}\ s;\ \mathbf{sto}\ s \qquad \textit{und} \qquad \mathrm{code}(PRINT\ \ x) = \mathrm{code}_R\, x;\ \mathbf{out}\ s.$$

Nun machen wir die Übersetzung von einfachen PROSA-Programmen an einem abschließenden Beispiel klar:

2.20. Beispiel Der Compiler soll folgendes Programm übersetzen:

PROGRAM Quadratzahlen;
VAR x : ARRAY [1..100] OF *int*;
VAR b, y : *int*;
BEGIN
READ b;
$y := 1$;
WHILE $(y \le b \wedge y \le 100)$ DO $x[y] := y * y; y := y + 1$ OD
END.

Der Compiler arbeitet erst die Deklarationen ab. Dabei legt er eine Symboltabelle an. Dort werden den deklarierten Namen Sorten und Speicherplatznummern zugeordnet, bei Feldern werden zusätzlich Dimension und die anderen statisch bekannten Größen zugeordnet:

Namen	Sorte	(Anfangs-)Adresse	statische Größen
x	$(1, int)$	5	$\mathrm{gr}(x) = 100, H = 1, D_1 = 1, D = 1$
b	(var, int)	105	
y	(var, int)	106	

Der Anweisungsteil des PROSA-Programms wird dann in folgendes P-Programm übersetzt:

ldc *int* 105; **inp** *int*; **sto** *int*; **ldc** *int* 106; **ldc** *int* 1; **sto** *int*; A1: **ldo** *int* 106; **ldo** *int* 105; **leq** *int*; **ldo** *int* 106; **ldc** *int* 100; **leq** *int*; **and** ; **fjp** A2; **ldc** *int* 5; **dec** 1; **ldo** *int* 106; **ixa** 1; **ldo** *int* 106; **ldo** *int* 106; **mul** *int*; **sto** *int*; **ldc** *int* 106; **ldo** *int* 106; **ldc** *int* 1; **add** *int*; **sto** *int*; **ujp** A1; A2:

Übungen

1. Gegeben sei eine P-Maschine, die einen Rechenspeicher STORE der Größe 100 besitzt (d.h. $maxstr = 99$). Dabei sei der Inhalt von STORE gegeben durch

$$STORE[i] \quad = \quad 3 \cdot i + 2 \qquad 0 \le i < 100.$$

   Sei der aktuelle Wert des Stackpointers $SP = 2$. Geben Sie alle Veränderungen bei Abarbeitung der folgenden Operationen (in der gegebenen Reihenfolge) an:

   ldo *int* 2; **ind** *int*; **ldc** *real* 1.0; **sro** *real* 1; **ldo** *int* 77; **sto** *int*.

2. Schreiben Sie ein Programm für eine P-Maschine, das zu gegebenem $n \in \mathbb{N}$ die Zahl $n!$ berechnet. Dabei sei am Anfang $SP = 0$ und $STORE[SP] = n$.

3. Geben Sie einen Algorithmus an, der entscheidet, ob ein Minus "−" in einem Term über Γ unär oder binär ist.

4. Konstruieren Sie einen Kellerautomaten mit Ausgabe, der code($x := t$) berechnet, wobei x eine Variable und t ein Term über Γ sind.

   Hinweis: Verwenden Sie den in Informatik I konstruierten Kellerautomaten, der U-Terme in P-Terme übersetzt.

5. Bestimmen Sie $\text{code}_R(t_1 \bmod t_2)$, wobei t_1 und t_2 Terme der Sorte int sind.

6. Geben Sie die code-Funktion für REPEAT-Schleifen an.

7. Betrachten Sie das folgende P-Maschinen-Programm:

	ldc *int* 0	
	ldc *int* 0	
loop :	**ldo** *int* 1	
	ldo *int* 0	
	les *int*	$(*)$
	fjp *ende*	
	ldo *int* 1	
	ldc *int* 1	
	add *int*	
	sro *int* 1	
	ldo *int* 1	
	ldo *int* 2	
	add *int*	
	sro *int* 2	
	ujp *loop*	
ende :		

   Beim Start des Programms sei $SP = 0$ und $STORE[SP] = n \in \mathbb{N}_0$. Bestimmen Sie den Wert der Speicherzelle $STORE[2]$ nach Ablauf des Programms in Abhängigkeit von n. Versuchen Sie, Ihre Antwort zu beweisen.

   Hinweise: Offensichtlich stellt der Programmteil zwischen der Marke *loop* und dem Befehl **ujp** *loop* eine Schleife dar. Zum Beweis der Korrektheit bestimme man eine Aussage der Gestalt "wenn wir zum i-ten mal den Befehl **les** *int* in Zeile $(*)$ ausführen, besitzt $STORE[1]$ bzw. $STORE[2]$ den Wert $g(i)$ bzw. $h(i)$" (für geeignete Funktionen $g(i), h(i)$). Diese Aussage beweise man dann durch Induktion über die Anzahl der Schleifendurchläufe und folgere daraus die Korrektheit der vorgeschlagenen Ausgabefunktion.

8. Berechnen Sie unter der Voraussetzung $\rho(a) = 5, \rho(b) = 6, \rho(c) = 7$

   (a) $code(a := ((b + b) = c))$

   (b) $code_R(a + (a * (a + b)))$

(c) $code_R(((a + a) * a) + b)$

Sind bei diesen Termen "bessere" Befehlsfolgen möglich? Modifizieren Sie ggf. das Übersetzungsschema.

9. Wie hängt die Anzahl der für die Auswertung eines Terms maximal benötigten Kellerzellen von der Struktur des Ausdrucks ab?

   Hinweis: Betrachten Sie die beiden "Extremfälle" $a + (a + (a + \ldots)\ldots)$ und $(\ldots(a + a) + a) + \ldots)$.

10. Beweisen Sie Lemma 2.14, in dem die Grundlage zur Addressierung von Feldelementen beschrieben wurde.

11. Gegeben sei der folgende PROSA-Programmteil:

    var $y, x :$ int;
    var $a :$ **array**$[1..3, 2..3]$ **of** int;

    $x := 3;$
    $a[3, 2] := x + 3;$
    $a[1, 2] := 7;$
    $y := a[1, 2] * 2;$

    Geben Sie die Adressierung aller deklarierten Variablen sowie der einzelnen Feld-Elemente (bei zeilenweiser Ablage) an und übersetzen Sie den Programmteil.

12. In Analogie zu der in der Vorlesung gezeigten zeilenweisen Ablage von Feldern soll in dieser Aufgabe die spaltenweise Ablage von Feldern hergeleitet werden. Bestimmen Sie die Adressen aller Feldelemente bei der Deklaration

    var $a :$ **array**$[1..3, 2..4]$ **of** int

    und $\rho(a) = 10$, wenn a spaltenweise abgelegt wird. Versuchen Sie dann, Formeln zur Bestimmung der Adresse eines beliebigen Elementes zu finden.

Kapitel 3

Schaltfunktionen und boolsche Algebra

3.1 Einleitung

Zahlen, Buchstaben etc. werden auf Rechnern binär kodiert. Einem Bitstring $(b_1, b_2, \ldots, b_k) \in \{0,1\}^k$ entspricht dabei die Zahl $\sum_{i=1}^{k} b_i 2^{k-i}$. Bei der Kodierung von Integers verwendet man auf zur Zeit benutzten Rechnern im allgemeinen $k = 32$. Möchte man auf einem Rechner zwei 32-Bit-Zahlen addieren, so muß man die entsprechende Funktion

$$\mathbf{add} : \{0,1\}^{32} \times \{0,1\}^{32} \to \{0,1\}^{32}$$

bestimmen.

3.1. Definition *Eine Funktion*

$$\{0,1\}^n \to \{0,1\}^m$$

mit $m, n \in \mathbb{N}$ *heißt* Schaltfunktion *oder* boolsche Funktion.

In diesem Abschnitt geht es um die Darstellungen von Schaltfunktionen durch elektrische Schaltungen.

3.2 Schaltkreise

Elektronische Schaltungen baut man aus Gattern und Speicherelementen auf. Hier interessieren uns zunächst die Gatter. Es gibt NOT-, OR-, AND- und NAND-Gatter. Diese realisieren jeweils die folgenden boolschen Funktionen

Gatter	Argument	Funktionswert
NOT	$x \in \{0,1\}$	$\neg x$
OR	$(x,y) \in \{0,1\}^2$	$x \vee y$
AND	$(x,y) \in \{0,1\}^2$	$x \wedge y$
NAND	$(x,y) \in \{0,1\}^2$	$\neg(x \wedge y)$

3.2. Definition *Für $k \in \mathbb{N}$ ist die* Variable x_k *der String $I^k\$$ und wir setzen $X = \{x_k : k \in \mathbb{N}\}$.*

3.3. Definition *Ein* Schaltkreis *ist ein azyklischer gerichteter Graph mit Knotenbenennung, der folgende Eigenschaften erfüllt:*

1. *Die Knoten mit Eingangsgrad Null heißen* Eingabeknoten *oder* Konstantenknoten. *Ihr Ausgangsgrad ist wenigstens eins. Die Eingabeknoten sind mit verschiedenen Variablen aus X bezeichnet. Es gibt wenigstens einen Eingabeknoten. Die Konstantenknoten sind mit 0 oder 1 bezeichnet.*

2. *Die Knoten mit Ausgangsgrad Null haben Eingangsgrad eins und heißen* Ausgabeknoten. *Hat der Schaltkreis k Ausgabeknoten, so sind diese mit $y_1, \ldots, y_k$ bezeichnet. Es gibt wenigstens einen Ausgabeknoten.*

3. *Alle anderen Knoten haben Eingangsgrad eins oder zwei und heißen* Gatter.

4. *Gatter mit Eingangsgrad eins heißen NOT-Gatter.*

5. *Gatter mit Eingangsgrad zwei heißen AND oder OR-Gatter.*

3.4. Definition *Ein Schaltkreis, bei dem alle Gatter Ausgangsgrad eins haben, heißt* Baumschaltkreis.

Die Funktionsweise von Schaltkreisen muß man sich so vorstellen: An die Eingangsknoten werden in einem bestimmten Rhythmus Signale angelegt. Die Kanten des Graphen sind Leitungen. Dabei bedeuten Signale über 2 Volt z.B. die logische Eins, während Signale unter 0.8 Volt die logische Null bedeuten. Gemäß der oben beschriebenen Funktionsweise der Gatter entsteht in Abhängigkeit von den Signalen an den Eingangskanten der inneren Knoten ein Signal an der Ausgangskante und damit liegen schließlich an den Ausgabeknoten Signale an, die wieder als logische Werte interpretiert werden können. Ein Schaltkreis berechnet also eine boolsche Funktion. Dies wird nun genauer untersucht.

3.3 Boolsche Ausdrücke

3.5. Definition *Die unvollständig geklammerten Terme über der Signatur mit den Konstanten $X \cup \{0, 1\}$ und den Operationen $\{\vee, \wedge, \neg\}$ mit den üblichen Prioritäten heißen* boolsche Ausdrücke. *Ein boolscher Ausdruck, in dem die Variable x_k vorkommt, aber keine Variable mit höherer Nummer, heißt* boolscher Ausdruck mit k Variablen.

Dann können wir folgendermaßen eine Verbindung zwischen Schaltkreisen und boolschen Ausdrücken herstellen:

3.6. Satz *Sei S ein Schaltkreis mit Knotenmenge V und Kantenmenge E. Dann wird durch folgende Festsetzungen in eindeutiger Weise jedem Knoten v von S ein boolscher Ausdruck $\phi(v)$ zugeordnet:*

1. *Ist v ein Konstantenknoten mit der Bezeichnung $b \in \{0,1\}$, so ist $\phi(v) = b$.*
2. *Ist v ein Eingabeknoten mit der Bezeichnung $x \in X$, so ist $\phi(v) = x$.*
3. *Ist $v \in V$ ein NOT-Gatter und $u \in V$ mit $(u,v) \in E$, dann ist $\phi(v) = \neg\phi(u)$.*
4. *Ist $v \in V$ ein OR-Gatter und sind $u_1, u_2 \in V, u_1 \neq u_2$ mit $(u_i, v) \in E, i = 1,2$, so ist $\phi(v) = (\phi(u_1) \vee \phi(u_2))$.*
5. *Ist $v \in V$ ein AND-Gatter und sind $u_1, u_2 \in V, u_1 \neq u_2$ mit $(u_i, v) \in E, i = 1,2$, so ist $\phi(v) = (\phi(u_1) \wedge \phi(u_2))$.*
6. *Ist v ein Ausgangsknoten und $u \in V$ mit $(u,v) \in E$, dann ist $\phi(v) = \phi(u)$.*

Beweis: Der Beweis erfolgt durch Induktion über die Tiefe der Knoten (Übung). ■

Schaltkreisen mit einem Ausgabeknoten kann man jetzt auf folgende Weise boolsche Ausdrücke zuordnen:

3.7. Definition *Sei S ein Schaltkreis mit einem Ausgabeknoten o. Dann setze $\phi(S) = \phi(o)$.*

Umgekehrt kann man auch jedem boolschen Ausdruck einen Schaltkreis zuordnen.

3.8. Definition

1. *Sei S ein Schaltkreis mit einem Ausgabeknoten. Dann ist $\neg S$ der Schaltkreis, den man erhält, indem man in die Kante zum Ausgabeknoten ein NOT-Gatter einfügt.*
2. *Seien S_1, S_2 Schaltkreise. Dann ist $S_1 \wedge S_2$ der Schaltkreis, den man erhält, indem man die Eingabeknoten und die Konstantenknoten mit gleicher Bezeichnung identifiziert, dann die Ausgabeknoten identifiziert, den so neu entstandenen Knoten mit AND bezeichnet und von diesem Knoten eine Kante in einen neuen Ausgabeknoten führt. Der Schaltkreis $S_1 \vee S_2$ ist entsprechend definiert.*

3.9. Satz *Den boolschen Ausdrücken wird durch folgende Vorschrift ein bis auf Isomorphie eindeutiger Baumschaltkreis $C(A)$ zugeordnet.*

1. *Ist $A = b \in \{0,1\}$, so sei $C(A)$ der Schaltkreis mit einem Konstantenknoten mit der Benennung b, von dem aus eine Kante in den einzigen Ausgabeknoten geht. Andere Knoten und Kanten gibt es nicht.*
2. *Ist $A = x_i \in X$, so sei $C(A)$ der Schaltkreis mit einem Eingabeknoten mit der Benennung x_i, von dem aus eine Kante in den einzigen Ausgabeknoten geht. Andere Knoten und Kanten gibt es nicht.*
3. *Ist $A = (A')$, so setze $C(A) = C(A')$.*
4. *Ist $A = \neg A'$, so setze $C(A) = \neg C(A')$.*
5. *Ist $A = A_1 \wedge A_2$, so setze $C(A) = C(A_1) \wedge C(A_2)$.*
6. *Ist $A = A_1 \vee A_2$, so setze $C(A) = C(A_1) \vee C(A_2)$.*

Beweis: Der Satz folgt aus dem Zerlegungssatz für U-Terme. ■

3.4 Berechnete Funktionen

3.10. Satz *Durch folgende Vorschrift wird jedem boolschen Ausdruck A in eindeutiger Weise eine boolsche Funktion $\beta(A) : \{0,1\}^k \to \{0,1\}$ zugeordnet ($k \geq 0$):*

1. *Ist $A = b \in \{0,1\}$, so ist $k = 0$ und $\beta(A)$ definiert durch $\epsilon \mapsto b$.*
2. *Ist $A = x_k \in X$, so sei $\beta(A)$ definiert durch $(b_1, \ldots, b_k) \mapsto b_k$.*
3. *Ist $A = (A')$, so setze $\beta(A) = \beta(A')$.*
4. *Ist $A = \neg A'$, so setze $\beta(A) = \neg\beta(A')$.*
5. *Ist $A = A_1 \wedge A_2$, wobei A_i boolscher Ausdruck mit k_i Variablen ist, i=1,2. Sei $k = \max\{k_1, k_2\}$. Dann ist $\beta(A)$ definiert durch*
$$(b_1, \ldots, b_k) \mapsto \beta(A_1)(b_1, \ldots, b_{k_1}) \wedge \beta(A_2)(b_1, \ldots, b_{k_2}).$$
6. *Ist $A = A_1 \vee A_2$, wobei A_i boolscher Ausdruck mit k_i Variablen ist, i=1,2. Sei $k = \max\{k_1, k_2\}$. Dann ist $\beta(A)$ definiert durch*
$$(b_1, \ldots, b_k) \mapsto \beta(A_1)(b_1, \ldots, b_{k_1}) \vee \beta(A_2)(b_1, \ldots, b_{k_2}).$$

Beweis: Der Satz folgt aus dem Zerlegungssatz für U-Terme. ■

Wir geben nun zuerst ein Beispiel an:

3.11. Beispiel Wir betrachten den boolschen Ausdruck

$$A = \underbrace{(x_1 \vee x_2)}_{=A_1} \wedge \underbrace{(x_3 \vee x_4)}_{=A_2}.$$

Wir konstruieren nun $\beta(A)$ induktiv von unten:

$$\begin{array}{rrcll}
\beta(x_1): & \{0,1\} & \longrightarrow & \{0,1\}, & b_1 \longmapsto b_1 \\
\beta(x_2): & \{0,1\}^2 & \longrightarrow & \{0,1\}, & (b_1, b_2) \longmapsto b_2 \\
\beta(x_3): & \{0,1\}^3 & \longrightarrow & \{0,1\}, & (b_1, b_2, b_3) \longmapsto b_3 \\
\beta(x_4): & \{0,1\}^4 & \longrightarrow & \{0,1\}, & (b_1, b_2, b_3, b_4) \longmapsto b_4
\end{array}$$

Bei allen diesen Termen haben wir Regel 2 angewendet. Nun wenden wir Regel 6 an und erhalten

$$\begin{array}{rrcll}
\beta(x_1 \vee x_2): & \{0,1\}^2 & \longrightarrow & \{0,1\}, & (b_1, b_2) \longmapsto \beta(x_1)(b_1) \vee \beta(x_2)(b_1, b_2) = b_1 \vee b_2 \\
\beta(x_3 \vee x_4): & \{0,1\}^4 & \longrightarrow & \{0,1\}, & \\
 & (b_1, b_2, b_3, b_4) & \longmapsto & \multicolumn{2}{l}{\beta(x_3)(b_1, b_2, b_3) \vee \beta(x_4)(b_1, b_2, b_3, b_4) = b_3 \vee b_4.}
\end{array}$$

Damit sind wir schon fast fertig. Nach Regel 3 gilt

$$\beta(\,(x_1 \vee x_2)\,) = \beta(x_1 \vee x_2)$$

und analog für $\beta((x_3 \vee x_4))$. Damit erhalten wir dann als Ergebnis

$$\begin{array}{rcl} \beta((x_1 \vee x_2) \wedge (x_3 \vee x_4)) : \quad \{0,1\}^4 & \longrightarrow & \{0,1\}, \\ (b_1, \ldots, b_4) & \longmapsto & \beta(A_1)(b_1, b_2) \wedge \beta(A_2)(b_1, b_2, b_3, b_4) \\ & & = \quad (b_1 \vee b_2) \ \wedge \ (b_3 \vee b_4). \end{array}$$

3.12. Definition *Sei S ein Schaltkreis mit einem Ausgabeknoten. Dann nennt man $\beta(S) = \beta(\phi(S))$ die von S* berechnete Funktion.

Der folgende Satz faßt die Resultate der letzten Abschnitte zusammen.

3.13. Satz

1. *Ist S ein Schaltkreis mit eimem Ausgabeknoten, so ist $\phi(S)$ ein vollständig geklammerter boolscher Ausdruck.*
2. *Ist A ein boolscher Ausdruck, so ist $C(A)$ ein Baumschaltkreis.*
3. *Sind S und S' Schaltkreise, so gilt $\phi(S) = \phi(S')$ genau dann, wenn S und S' isomorph sind.*
4. *Ist A ein vollständig geklammerter boolscher Ausdruck, dann ist $\phi(C(A)) = A$.*

3.14. Korollar *Sei $\mathcal{S}$ die Menge der Isomorphieklassen von Baumschaltkreisen. Sei $\mathcal{A}$ die Menge der vollständig geklammerten boolschen Ausdrücke. Für $K \in \mathcal{S}$ und irgendein $S \in K$ sei $\Phi(K) = \phi(S)$. Für $A \in \mathcal{A}$ sei $\mathcal{C}(A) = \{S : S$ isomorph zu $C(A)\}$. Dann ist $\Phi : \mathcal{S} \to \mathcal{A}$ wohldefiniert, bijektiv und die Umkehrabbildung ist $\mathcal{C}$.*

3.5 Realisierung boolscher Funktionen durch Schaltkreise

3.15. Definition *Sei β eine boolsche Funktion, S ein Schaltkreis und A ein boolscher Ausdruck. Falls $\beta(S) = \beta$ gilt, sage ich, daß der Schaltkreis S die Funktion β* realisiert *und falls $\beta(A) = \beta$ gilt, sage ich, daß der Ausdruck A die Funktion β* realisiert.

3.16. Definition *Sei $k \in \mathbb{N}$, $b = (b_1, \ldots, b_k) \in \{0,1\}^k$. Für $1 \leq i \leq k$ setze $y_i = x_i$, falls $b_i = 1$ und $y_i = \neg x_i$, falls $b_i = 0$. Dann heißt $m(b) = y_1 \wedge y_2 \wedge \ldots \wedge y_k$* vollständiges Monom *für b.*

3.17. Lemma *Sei $k \in \mathbb{N}$,$b = (b_1, \ldots, b_k) \in \{0,1\}^k$. Dann ist $\beta(m(b))$ eine k-stellige boolsche Funktion und es gilt $\beta(m(b))(c) = 1$ genau dann, wenn $c = b$ für alle $c \in \{0,1\}^k$.*

3.18. Definition *Sei β ein k-stellige boolsche Funktion. Dann heißt der boolsche Ausdruck*

$$A(\beta) = \bigvee_{b \in \{0,1\}^k, \beta(b)=1} m(b).$$

disjunktive Normalform *von β.*

3.19. Satz *Sei β ein k-stellige boolsche Funktion. Dann gilt $\beta(A(\beta)) = \beta$.*

3.6 Kosten

Satz 3.18 zeigt zwar, daß man jede boolsche Funktion durch einen boolschen Ausdruck realisieren kann. Er ist aber zur Konstruktion solcher boolscher Ausdrücke ungeeignet, da boolsche Funktionen fast nie durch ihre Funktionstabelle gegeben sind. Stattdessen sind boolsche Funktionen oft bereits durch boolsche Ausdrücke gegeben. Diese werden dann entweder direkt als Schaltkreise realisiert oder zuerst in "billigere" Ausdrücke transformiert, die dieselbe Funktion berechnen. Was "billig" heißt, wird nun definiert.

3.20. Definition *1. Die* Kosten eines Schaltkreises *S sind definiert als die Anzahl der Gatter in dem Schaltkreis. Sie werden mit $L(S)$ bezeichnet.*

2. Sind S und S' Schaltkreise, dann heißt S billiger *als S', wenn $L(S) < L(S')$ und* teurer, *falls $L(S) > L(S')$ ist.*

3. Die Kosten eines boolschen Ausdrucks A sind definiert als die Anzahl der nicht konstanten Operationszeichen in A. Sie werden mit $L(A)$ bezeichnet. Sind A und A' boolsche Ausdrücke, dann heißt A billiger *als A', wenn $L(A) < L(A')$ und* teurer, *falls $L(A) > L(A')$ ist.*

4. Für eine boolsche Funktion β bezeichne ich mit $L(\beta)$ die Kosten des billigsten Schaltkreises, der β realisiert.

3.21. Lemma *Ist S ein Schaltkreis mit einem Ausgabeknoten und A ein boolscher Ausdruck, so gilt $L(\phi(S)) = L(S)$ und $L(C(A)) = L(A)$.*

Beweis: Der Beweis folgt mittels Induktion aus der Definition von $C(A)$ und $\phi(S)$. ■

Das Ziel der nun folgenden Untersuchung besteht darin, für eine gegebene boolsche Funktion einen möglichst billigen Schaltkreis zu konstruieren, der diese Funktion realisiert. Aus Lemma 3.21 folgt, daß der billigste Schaltkreis, der eine boolsche Funktion β realisiert, höchstens so teuer ist wie der billigste boolsche Ausdruck, der β realisiert. Die bisher bekannten Methoden machen es leichter, für β billige boolsche Ausdrücke zu finden als billige boolsche Schaltkreise.

Oben wurde gezeigt, daß man jede boolsche Funktion β durch den boolschen Ausdruck $A(\beta)$ realisieren kann. Dessen Kosten schätze ich zuerst ab:

3.22. Satz *Sei β eine k-stellige boolsche Funktion. Dann gilt $L(\beta) \leq L(A(\beta)) \leq 2^{k+1} \cdot k - 1$.*

Beweis: Der boolsche Ausdruck $A(\beta)$ ist eine Konjunktion von höchstens 2^k Disjunktionen. In jeder dieser Disjunktionen kommen $k - 1$ Konjunktions- und höchstens k Negationszeichen vor. Außerdem ist die Anzahl der $\vee$-Zeichen damit maximal $2^k - 1$. Damit erhalten wir als obere Schranke für die Kosten $2^k \cdot (k + k - 1) + 2^k - 1 = 2^{k+1} \cdot k - 1$. ■

3.7 Rechenregeln für boolsche Funktionen

Um boolsche Ausdrücke, die eine bestimmte Funktion realisieren, zu optimieren, kann man eine Reihe von Rechenregeln anwenden.

3.23. Definition *Seien $k, l \in \mathbb{N}$, $m = \max\{k, l\}$, sei α eine k-stellige und β eine l-stellige boolsche Funktion.*

1. *Die boolsche Funktion $\neg\alpha$ ist definiert durch $\{0,1\}^k \to \{0,1\}, \quad b \mapsto \neg\alpha(b)$.*
2. *Die boolsche Funktion $\alpha \wedge \beta$ ist definiert durch*
$$\{0,1\}^m \to \{0,1\}, \qquad (b_1,\dots,b_m) \in \{0,1\}^m \mapsto \alpha(b_1,\dots,b_k) \wedge \beta(b_1,\dots,b_l).$$
3. *Die boolsche Funktion $\alpha \vee \beta$ ist definiert durch*
$$\{0,1\}^m \to \{0,1\}, \qquad (b_1,\dots,b_m) \in \{0,1\}^m \mapsto \alpha(b_1,\dots,b_k) \vee \beta(b_1,\dots,b_l).$$

Der nächste Satz besagt, daß die boolschen Funktionen mit diesen Rechenregeln eine *boolsche Algebra* bilden.

3.24. Satz *Seien α, β und γ boolsche Funktionen. Dann gilt*

1. *Kommutativität:* $\alpha \wedge \beta = \beta \wedge \alpha, \quad \alpha \vee \beta = \beta \vee \alpha.$
2. *Assoziativität:* $\alpha \wedge (\beta \wedge \gamma) = (\alpha \wedge \beta) \wedge \gamma, \quad \alpha \vee (\beta \vee \gamma) = (\alpha \vee \beta) \vee \gamma.$
3. *Distributivität:* $\alpha \wedge (\beta \vee \gamma) = \alpha \wedge \beta \vee \alpha \wedge \gamma, \quad \alpha \vee (\beta \wedge \gamma) = (\alpha \vee \beta) \wedge (\alpha \vee \gamma).$
4. *Absorbtion:* $\alpha \wedge (\alpha \vee \beta) = \alpha, \quad \alpha \vee (\alpha \wedge \beta) = \alpha.$
5. *Neutralität:* $\alpha \wedge (\beta \vee \neg\beta) = \alpha, \quad \alpha \vee (\beta \wedge \neg\beta) = \alpha.$

Die folgenden Rechenregeln gelten allgemein in boolschen Algebren. Hierbei verwende ich die Bezeichnungen 0 und 1 für die boolschen Funktionen, deren Wert entweder konstant Null oder konstant Eins ist. Die Bezeichnung ist zwar nicht eindeutig, aber die Stelligkeit ergibt sich jeweils aus dem Zusammenhang.

3.25. Satz *Seien α und β boolsche Funktionen. Dann gilt*

1. $\alpha \vee \neg\alpha = 1, \quad \alpha \vee 1 = 1, \quad \alpha \vee 0 = \alpha, \quad \alpha \wedge \neg\alpha = 0, \quad \alpha \wedge 1 = \alpha, \quad \alpha \wedge 0 = 0$
2. *Doppelte Negation:* $\neg\neg\alpha = \alpha$
3. *DeMorgan-Regeln:* $\neg\alpha \wedge \neg\beta = \neg(\alpha \vee \beta), \quad \neg\alpha \vee \neg\beta = \neg(\alpha \wedge \beta)$

3.26. Beispiel Es gilt

$$\begin{aligned}
(b_1 \wedge b_2) \vee \neg(\neg b_1 \vee b_2) &= (b_1 \wedge b_2) \vee (\neg\neg b_1 \wedge \neg b_2) \\
&= (b_1 \wedge b_2) \vee (b_1 \wedge \neg b_2) \\
&= b_1 \wedge (b_2 \vee \neg b_2) \\
&= b_1 \wedge 1 \\
&= b_1.
\end{aligned}$$

Damit hat man eine wesentlich einfachere Darstellung der gegebenen boolschen Funktion gefunden.

3.8 Boolsche Polynome

Man hätte gern ein systematisches Verfahren zur Berechnung möglichst kleiner boolscher Ausdrücke, die eine gegebene Funktion darstellen. Ein solches Verfahren ist aber nicht bekannt. Darum macht man sich das Leben etwas leichter und versucht, boolsche Funktionen durch möglichst kleine *Polynome* darzustellen.

3.27. Definition 1. *Ein (boolsches)* Literal *ist ein boolscher Ausdruck von der Form $A = x$ oder $A = \neg x$ mit $x \in X$. Die Nummer eines Literals ist die Nummer der entsprechenden Variablen.*

2. *Sei $k \in \mathbb{N}$. Ein (boolsches)* Monom *ist ein boolscher Ausdruck von der Form $A = l_1 \wedge l_2 \wedge \ldots \wedge l_k$, wobei die l_i Literale mit aufsteigenden Nummern sind.*

3. *Sei $k \in \mathbb{N}$. Ein (boolsches)* Polynom *ist ein boolscher Ausdruck von der Form $A = m_1 \vee m_2 \vee \ldots \vee m_k$, wobei die m_i Monome sind.*

Ich habe bereits oben bewiesen, daß man jede boolsche Funktion durch ein Polynom, nämlich durch die vollständige disjunktive Normalform darstellen kann. Diese Darstellung soll jetzt optimiert werden. Aber zuvor beweise ich, daß sich manche boolsche Funktionen nur durch sehr große boolsche Funktionen darstellen lassen.

3.28. Lemma *Sei A ein boolscher Ausdruck ohne Klammern mit $L(A) \geq 1$. Dann ist die Länge von A höchstens $3 \cdot L(A)$.*

Beweis: Induktion über $L(A)$. ■

3.29. Satz *Sei $k \in \mathbb{N}$. Dann gibt es eine boolsche Funktion, die sich nicht durch ein Polynom darstellen läßt, das billiger als $2^k/(3\log(k+3))$ ist.*

Beweis: Die Anzahl der k-stelligen boolschen Funktionen ist 2^{2^k}. Sei $K \in \mathbb{N}$ und sei p ein boolsches Polynom mit $L(p) \leq K$, das eine k-stellige boolsche Funktion β realisiert. Dann ist p ein klammerfreier boolscher Ausdruck, der aus den Zeichen $\wedge, \vee, \neg, x_1, \ldots, x_k$ aufgebaut ist. Nach Lemma 3.28 gibt es davon höchstens $(k+3)^{3K}$ viele. Damit alle boolschen Funktionen realisiert werden können, muß $(k+3)^{3K} \geq 2^{2^k}$ gelten. Dies stimmt nur für $K \geq 2^k/(3\log(k+3))$. ■

3.9 Optimierung von Polynomen

Angenommen, eine boolsche Funktion β ist dargestellt durch das Polynom p. In diesem Abschnitt beschreibe ich ein Verfahren, aus p ein möglichst billiges Polynom q zu konstruieren, das β immer noch darstellt.

3.30. Definition *Sei β eine boolsche Funktion. Ein Polynom p, welches β realisiert und dabei minimale Kosten hat, heißt* Minimalpolynom *von β.*

"Effiziente" Verfahren, Minimalpolynome zu bestimmen, sind nicht bekannt. Die folgende Untersuchung wird aber ein Verfahren ergeben, möglichst minimale Polynome zu erzeugen. Ich setze in diesem Abschnitt voraus, daß $k \in \mathbb{N}$ ist und daß alle betrachteten boolschen Funktionen k-stellig sind. Außerdem werden alle betrachteten Polynome nur die Variablen $x_1, \ldots, x_k$ enthalten. Für ein solches Polynom p sei $\alpha(p)$ die boolsche Funktion, die man erhält, indem man die Funktion $\beta(p)$ in natürlicher Weise zu einer Funktion mit der Definitionsmenge $\{0,1\}^k$ fortsetzt.

3.31. Beispiel Sei $k = 3$ und $p = x_1 \wedge x_2$. Dann wird die zugehörige boolsche Funktion

$$\beta(p) : \{0,1\}^2 \to \{0,1\}, \quad (b_1, b_2) \mapsto b_1 \wedge b_2$$

und wir erhalten die entsprechende Funktion $\alpha(p)$ als

$$\alpha(p) : \{0,1\}^3 \to \{0,1\}, \quad (b_1, b_2, b_3) \mapsto b_1 \wedge b_2.$$

Gilt $\alpha(p) = \beta$ für eine boolsche Funktion β und ein Polynom p, so sage ich, daß p die Funktion β *darstellt.* Ich benötige noch einige Begriffe.

3.32. Definition

1. *Seien $a, b \in \{0,1\}^k$, $a = (a_1, \ldots, a_k)$, $b = (b_1, \ldots, b_k)$. Dann schreibe ich $a \leq b$, wenn $a_i \leq b_i$ gilt für $1 \leq i \leq k$.*
2. *Seien α und β k-stellige boolsche Funktionen. Dann schreibe ich $\alpha \leq \beta$, wenn $\alpha(a) \leq \beta(a)$ gilt für alle $a \in \{0,1\}^k$.*
3. *Eine k-stellige boolsche Funktion β heißt* monoton, *falls $\beta(a) \leq \beta(b)$ gilt für alle $a, b \in \{0,1\}^k$ mit $a \leq b$.*

3.33. Definition *Sei β eine k-stellige boolsche Funktion. Ein Monom m heißt* Implikant *von β, wenn $\alpha(m)(b) \leq \beta(b)$ für alle $b \in \{0,1\}^k$.*

Offensichtlich gilt:

3.34. Lemma *Sei β eine boolsche Funktion und sei p ein Polynom, das β darstellt. Dann ist jedes Monom in p ein Implikant von β.*

Beweis: Nimmt ein Monom an einer Stelle den Wert 1 an, so nimmt auch die Funktion den Wert 1 an. ■

3.35. Definition 1. *Sei z ein Literal. Falls $z = x$ ist mit $x \in X$, so setze ich $\overline{z} = \neg x$. Falls $z = \neg x$ ist mit $x \in X$, dann setze ich $\overline{z} = x$.*

2. *Sei $l \in \mathbb{N}_{\geq 2}$, seien $z_1, \ldots, z_l$ Literale und sei $m = z_1 \wedge \ldots \wedge z_l$ ein Monom. Dann schreibe ich kurz $m = z_1 z_2 \ldots z_l$ und für $i \in \{1, \ldots, l\}$ sage ich, daß das Monom $z_1 \ldots z_{i-1} z_{i+1} \ldots z_l$ aus m durch Streichen des Literals z_i hervorgeht.*

3. *Ein* Teilmonom *eine Monoms m ist ein Monom, das durch Streichen von Literalen aus m hervorgeht.*

Der wesentliche Trick bei der Verkleinerung von Polynomen wird an folgendem Beispiel erläutert:

3.36. Beispiel Angenommen, die boolsche Funktion β wird dargestellt durch das Polynom $p = x_1 x_2 \overline{x_3} \vee x_1 x_2 x_3$. Dann hängt die Funktion offensichtlich nicht von x_3 ab. Ihr Wert wird vollständig bestimmt durch den Wert von $x_1 x_2$. Man kann die Funktion deshalb auch durch $x_1 x_2$ darstellen. Das Monom $x_1 x_2 \overline{x_3}$ heißt *Partner* von $x_1 x_2 x_3$ und umgekehrt. Die Ersetzung der beiden Partner durch $x_1 x_2$ heißt ein *Resolutionsschritt.* Der Resolutionsschritt macht das darstellende Polynom offensichtlich billiger. Wir werden dieses Konzept nun allgemein beschreiben.

3.37. Definition *Sei $l \in \mathbb{N}, l \geq 2$, seien $z_1, \ldots, z_l$ Literale und sei $i \in \{1, \ldots, l\}$. Dann heißen die Monome*

$$m = z_1 z_2 \ldots z_{i-1} z_i z_{i+1} \ldots z_l \qquad \text{und} \qquad m' = z_1 z_2 \ldots z_{i-1} \overline{z_i} z_{i+1} \ldots z_l$$

Partner *und das Monom $R(m, m') = z_1 \ldots z_{i-1} z_{i+1} \ldots z_l$ heißt* Resolvente *des Paars m, m'.*

3.38. Lemma *Seien die Monome m und m' Partner. Dann gilt*

$$\alpha(m \vee m') \quad = \quad \alpha(R(m, m')).$$

Beweis: Angenommen das Literal z kommt in m vor und $\overline{z}$ kommt in m' vor. Dann gilt offensichtlich $\alpha(m \vee m') = \alpha(R(m, m')) \wedge (\alpha(z) \vee \neg\alpha(z)) = \alpha(R(m, m'))$. ■

Der folgende Algorithmus reduziert die Kosten der Darstellung einer boolschen Funktion durch ein Polynom p:

3.39. Algorithmus [Kostenreduktion von Polynomen]

WHILE Es gibt ein Paar von Partnern in p		(1)
	Ersetze dieses Paar durch seine Resolvente und streiche es aus p.	(2)

Dann gilt der folgende Satz über die Länge des Ausgabepolynoms:

3.40. Satz *Sei p ein Polynom mit Kosten $L(p) \geq 3$. In jeder Iteration des Algorithmus 3.39 verringern sich die Kosten von p wenigstens um 4. Das Polynom, das am Ende entsteht, stellt dieselbe Funktion dar wie das eingegebene Polynom.*

Beweis: In jedem Resolutionsschritt wird wenigstens ein Monom gestrichen. Wird genau ein Monom gestrichen, so fällt auch ein $\vee$-Zeichen weg. Außerdem enthielt das gestrichene Monom wenigstens ein $\wedge$ und ein $\neg$-Zeichen. Ein weiteres Monom wird durch Streichen eines Literals verkleinert. Dadurch fällt ein $\wedge$-Zeichen weg. Das bedeutet eine Kostenreduktion um 4. Zwei Monome können nicht gleichzeitig gestrichen werden, denn dann wären diese beiden Monome sogar Literale und damit ist nach Definition 3.37 kein Resolutionsschritt möglich.

Die Invarianz der dargestellten Funktionen folgt unmittelbar aus Lemma 3.38. ∎

Man sollte beachten, daß dieser Algorithmus im allgemeinen nicht das Minimalpolynom zu einem gegebenen Polynom berechnet. Dieses Problem werden wir in den nächsten Abschnitten behandeln.

3.10 Geometrie boolscher Funktionen

In diesem ganzen Abschnitt gehen wir davon aus, daß alle benutzten boolschen Funktionen k-stellig sind und in allen Polynomen nur die Variablen $x_1, \ldots, x_k$ auftreten. Die 2^k Argumente der boolschen Funktionen kann man sich vorstellen als die Ecken des k-dimensionalen Einheitswürfels im $\mathbb{Z}^k$. Man kann eine solche boolsche Funktion α darstellen, indem man diejenigen Ecken, an denen der Funktionswert 1 wird, mit einem dicken Punkt versieht, die anderen Ecken aber nicht. Wie sehen die Funktionen aus, die durch Monome dargestellt werden? Dazu betrachte ich ein Beispiel. Sei $k = 3$ und sei $m = x_1$. Die von m dargestellte Funktion wird genau dann 1, wenn die erste Koordinate des Argumentes 1 ist. Die entsprechenden Punkte sind die Eckpunkte eines Seitenquadrates des 3-dimensionalen Würfels. So etwas nennt man einen zweidimensionalen Teilwürfel.

3.41. Definition *Seien $a, b \in \{0,1\}^k$. Dann heißt $H(a,b) = \sum_{i=1}^{k} |a_i - b_i|$ der* Hamming-Abstand *oder die* Hamming-Distanz *von a und b.*

3.42. Lemma *Seien $a, b, c \in \{0,1\}^k$. Dann gilt*

1. *$H(a,b) \geq 0$ und $H(a,b) = 0$ genau dann, wenn $a = b$.*
2. *$H(a,c) \leq H(a,b) + H(b,c)$.*

Beweis: Die beiden ersten Aussagen sind trivial. Die letzte folgt unmittelbar aus der Dreiecksungleichung für den gewöhnlichen Absolutbetrag. ∎

3.43. Definition *1. Der* k-dimensionale Würfel W_k *ist der ungerichtete Graph mit Knotenmenge* $\{0,1\}^k$, *in dem je zwei Knoten der Hamming-Distanz* 1 *durch eine Kante verbunden sind.*

2. Sei $l \in \mathbb{N}$ *und sei* G *ein ungerichteter Graph. Ein Teilgraph* W' *von* G, *der isomorph zu* W_l *ist, heißt* l-dimensionaler Teilwürfel *von* G.

3.44. Beispiel Betrachten wir den 3-dimensionalen Einheitswürfel W_3. Er besitzt genau 8 0-dimensionale Teilwürfel (alle Knoten), 12 1-dimensionale Teilwürfel (alle Kanten), 6 2-dimensionale Teilwürfel (Seitenflächen) und genau einen 3-dimensionalen Teilwürfel.

Jeder boolschen Funktion wird ein Teilgraph von W_k zugeordnet.

3.45. Definition *Sei* α *eine* k*-stellige boolsche Funktion. Dann ist* $G(\alpha)$ *der Teilgraph von* W_k *mit Knotenmenge* $\alpha^{-1}(1)$, *in dem je zwei Knoten genau dann durch eine Kante verbunden sind, wenn das auch in* W_k *der Fall ist. Für ein Polynom* p *setze ich außerdem* $G(p) = G(\alpha(p))$.

Auch die auf den boolschen Funktionen eingeführte Ordnungsrelation läßt sich dann geometrisch deuten.

3.46. Satz *Seien* α *und* β *boolsche Funktionen. Dann gilt* $\alpha \leq \beta$ *genau dann, wenn* $G(\alpha)$ *ein Teilgraph von* $G(\beta)$ *ist.*

Jetzt kann ich die Funktionen charakterisieren, die durch Monome dargestellt werden.

3.47. Satz *Sei* m *ein Monom mit* l *Literalen. Dann ist* $G(m)$ *ein* $k-l$*-dimensionaler Teilwürfel von* W_k.

Beweis: Ich nehme o.B.d.A an, daß in m genau die ersten l Variablen vorkommen und ich konstruiere einen Isomorphismus von $G(m)$ nach W_{k-l}. Es gibt genau eine Weise, in der man die ersten l Koordinaten von $b \in \{0,1\}^k$ wählen muß, damit $\alpha(m)(b) = 1$ wird. Die anderen Koordinaten sind beliebig. Also ordnet man jedem solchen b einfach das Tupel aus den letzten $k-l$ Variablen zu. Daß dies ein Isomorphismus ist, läßt sich sehr leicht verifizieren. ■

3.11 Primimplikanten

Auch in diesem Abschnitt gehen wir davon aus, daß $k \in \mathbb{N}$ fest gewählt ist und alle benutzten boolschen Funktionen k-stellig sind. Außerdem sollen in allen Polynomen nur die Variablen $x_1, \ldots, x_k$ auftreten. Ich untersuche nun, inwieweit der Algorithmus 3.39 wirklich das optimale Polynom liefert.

3.48. Definition *Sei* β *eine boolsche Funktion. Ein Monom* m, *für das* $\alpha(m) \leq \beta$ *gilt, heißt* Implikant *von* β. *Gibt es kein echtes Teilmonom von* m, *welches Implikant von* β *ist, so heißt* m Primimplikant *von* β.

Implikanten und Primimplikanten lassen sich geometrisch charakterisieren.

3.49. Satz *Es sei α eine boolsche Funktion und es sei m ein Monom. Dann gilt:*

1. *Genau dann ist m ein Implikant von α, wenn $G(m)$ ein Teilwürfel von $G(\alpha)$ ist.*
2. *Genau dann ist m ein Primimplikant von α, wenn $G(m)$ ein bezüglich Inklusion maximaler Teilwürfel von $G(\alpha)$ ist.*

3.50. Lemma *Seien α, β und γ boolsche Funktionen. Dann folgt aus $\alpha = \beta \vee \gamma$, daß $G(\beta)$ ein Teilgraph von $G(\alpha)$ ist, und es folgt aus $\alpha = \beta \wedge \gamma$, daß $G(\alpha)$ ein Teilgraph von $G(\beta)$ ist.*

Aus diesem Satz kann man einige Aussagen folgern:

3.51. Korollar *Es sei β eine boolsche Funktion und es sei p ein Polynom, das β darstellt. Dann ist jedes Monom in p ein Implikant von β.*

3.52. Korollar *Ist m ein Monom und ist m' ein Teilmonom von m, dann ist $G(m)$ ein Teilwürfel von $G(m')$.*

3.53. Lemma *Seien α, β und γ boolsche Funktionen und sei $G(\alpha)$ ein Teilgraph von $G(\beta)$. Dann ist $G(\alpha \vee \gamma)$ ein Teilgraph von $G(\beta \vee \gamma)$ und $G(\alpha \wedge \gamma)$ ist ein Teilgraph von $G(\beta \wedge \gamma)$.*

Beweis: Es genügt zu zeigen, daß der Wert der rechten Funktion Eins ist, wenn der Wert der linken Funktion Eins ist. Sei also $\beta \in \{0,1\}^k$.

Ich beweise zuerst die erste Aussage. Wenn $\alpha(b) \vee \gamma(b) = 1$ ist, so folgt, daß $\alpha(b) = 1$ oder $\gamma(b) = 1$ ist. Nach Voraussetzung ist im ersten Fall auch $\beta(b) = 1$ und daher $\beta(b) \vee \gamma(b) = 1$.

Jetzt zur zweiten Ungleichung. Wenn $\alpha(b) \wedge \gamma(b) = 1$ ist, so folgt, daß $\alpha(b) = 1$ und $\gamma(b) = 1$ ist. Das bedeutet, daß nach Voraussetzung $\beta(b) = 1$, also auch $\beta(b) \wedge \gamma(b) = 1$, ist. ■

3.54. Satz *Sei β eine boolsche Funktion und sei p ein Minimalpolynom von β. Dann ist jedes Monom von p ein Primimplikant von β.*

Beweis: Es sei m ein Monom in p. Dieses Monom ist ein Implikant von β. Angenommen, es ist kein Primimplikant von β. Sei m' ein echtes Teilmonom von m, welches auch noch ein Implikant von β ist; sei $p = p' \vee m$ und sei $q = p' \vee m'$. Ich zeige, daß dann $\alpha(q) = \beta$ ist. Dies ist ein Widerspruch zur Minimalität von p. Nach Lemma 3.50 und Lemma 3.53 gelten folgende Ungleichungen:

$$\beta \quad = \quad \alpha(p) \quad = \quad \alpha(p') \vee \alpha(m) \quad \leq \quad \alpha(p') \vee \alpha(m') \quad = \quad \alpha(q).$$

Weil aber m' ein Implikant von β ist, folgt andererseits

$$\alpha(q) \quad = \quad \alpha(p') \vee \alpha(m') \quad \leq \quad \beta \vee \beta \quad = \quad \beta.$$

■

Die Aufgabe, das Minimalpolynom einer boolschen Funktion β zu finden, kann man geometrisch deuten als den Versuch, in einer Teilmenge von $\{0,1\}^k$ eine minimale Überdeckung mit maximalen Teilwürfeln zu finden.

Man könnte glauben, daß ein Minimalpolynom aus allen Primimplikanten einer Funktion zusammengesetzt ist. Das ist aber im allgemeinen nicht so. Minimalpolynome sind noch nicht einmal eindeutig. Ein Beispiel dazu soll in der Übung konstruiert werden.

Ich beweise jetzt noch zwei Sätze, die es ermöglichen, ein Minimalpolynom für eine boolsche Funktion zu berechnen.

3.55. Definition *Der* Grad *eines Monoms m ist die Anzahl der in m vorkommenden Literale.*

3.56. Satz *Alle Implikanten vom Grad k einer k-stelligen boolschen Funktion β kommen als Monome in der vollständigen disjunktiven Normalform von β vor.*

Beweis: Offensichtlich sind alle Monome, die in $A(\beta)$ vorkommen, Implikanten von β.

Umgekehrt ist für ein Monom m vom Grad k der Würfel $W(m)$ 0-dimensional, d.h. $\alpha(m)$ ist an genau einer Stelle 1. Ist m ein Implikant von β, so muß m nach Konstruktion in $A(\beta)$ vorkommen. ■

3.57. Lemma *Jeder Implikant einer boolschen Funktion β hat ein Teilmonom, das ein Primimplikant von β ist.*

Geben wir nun zuerst einen Algorithmus an, der alle Primimplikanten einer boolschen Funktion berechnet und zeigen wir dann anschließend die Korrektheit des Algorithmus.

3.58. Algorithmus [Bestimmung aller Primimplikanten]

Bilde die Menge M aller vollständigen Monome für alle Elemente aus $\beta^{-1}(1)$.		(1)
WHILE Es gibt ein Paar von Partnern		(2)
	Bestimme in M alle Paare von Partnern	(3)
	Nimm für jedes Paar die Resolvente zu M dazu	(4)
	Streiche aus M alle in (2) benutzten Monome	(5)

3.59. Satz *Es sei β eine k-stellige boolsche Funktion und es sei M die Menge, welche man als Ergebnis von Algorithmus 3.58 erhält. Dann sind alle Monome in M Primimplikanten von β. Andere Primimplikanten von β gibt es nicht.*

Beweis: Ich beweise für den Algorithmus folgende Invariante:

Für $0 \leq l \leq k$ gilt: Die Menge M_l, die man nach der l-ten Iteration von Algorithmus 3.58 erhält, enthält alle Implikanten von β vom Grad $k-l$, alle Primimplikanten von höherem Grad und sonst keine Monome.

Für $l = 0$ folgt diese Behauptung aus Satz 3.56.

Angenommen, die Behauptung stimmt für l. Es sei m ein Implikant vom Grad $k-l-1$ von β und es sei x eine Variable, die nicht in m vorkommt. Dann ist sowohl $m \wedge x$ als auch $m \wedge \overline{x}$ ein Implikant von β. Nach Voraussetzung kommen diese beiden Implikanten in M_l vor. Also gehört m zu M_{l+1}. Sei nun m ein Primimplikant vom Grad $k-l$. Dann kann kein Partner von m ein Monom in M_l sein und darum gehört m auch zu M_{l+1}. Sei schließlich m irgendein Monom in M_l. Dann ist m sicher ein Implikant von β. Ist m kein Primimplikant von β, so ist ein Teilmonom von m ein Primimplikant von β, und hieraus konstruiert man leicht einen Partner von m, der nach Voraussetzung auch in M_l vorkommen muß. Also kann m die nächste Iteration nicht überleben und alle Monome in M_{l+1} vom Grad $> k-l-1$ sind Primimplikanten von β. ∎

Um das Minimalpolynom einer boolschen Funktion β zu bestimmen, kann man so vorgehen. Man berechnet zuerst die vollständige disjunktive Normalform von β. Dann bestimmt man durch Resolution alle Primimplikanten von β. Durch Ausprobieren berechnet man daraus das Minimalpolynom. Man kann zeigen, daß aus der Existenz eines Polynomzeitalgorithmus zur Berechnung von Minimalpolynomen die Existenz eines Polynomzeitalgorithmus für viele andere schwere Probleme, z.B. für die Faktorisierung großer Zahlen, folgt. Daher glauben viele Leute nicht, daß es einen solchen Algorithmus gibt.

3.60. Beispiel Wir wollen alle Primimplikanten der folgenden boolschen Funktion bestimmen:

b_1, b_2, b_3, b_4	$\beta(b_1, b_2, b_3, b_4)$	b_1, b_2, b_3, b_4	$\beta(b_1, b_2, b_3, b_4)$
(0,0,0,0)	0	(1,0,0,0)	0
(0,0,0,1)	1	(1,0,0,1)	0
(0,0,1,0)	1	(1,0,1,0)	0
(0,0,1,1)	1	(1,0,1,1)	0
(0,1,0,0)	0	(1,1,0,0)	0
(0,1,0,1)	1	(1,1,0,1)	0
(0,1,1,0)	0	(1,1,1,0)	0
(0,1,1,1)	1	(1,1,1,1)	0

Damit erhalten wir die Menge aller vollständigen Monome als

$$M_0 \;=\; \Big\{ \underbrace{\overline{x}_1\overline{x}_2\overline{x}_3x_4}_{=p_1}, \underbrace{\overline{x}_1\overline{x}_2x_3x_4}_{=p_2}, \underbrace{\overline{x}_1x_2\overline{x}_3x_4}_{=p_3}, \underbrace{\overline{x}_1x_2x_3x_4}_{=p_4}, \underbrace{\overline{x}_1\overline{x}_2x_3\overline{x}_4}_{=p_5} \Big\}.$$

Dann finden wir folgende Partner:

$$(p_1, p_2), \quad (p_1, p_3), \quad (p_2, p_4), \quad (p_2, p_5), \qquad (p_3, p_4).$$

In der ersten Schleifeniteration wird für alle diese Paare die Resolvente dazugenommen und dann p_1, p_2, p_3, p_4, p_5 gestrichen. Damit erhalten wir die Menge

$$M_1 \;=\; \{\overline{x}_1\overline{x}_2x_4, \overline{x}_1\overline{x}_3x_4, \overline{x}_1x_3x_4, \overline{x}_1\overline{x}_2x_3, \overline{x}_1x_2x_4\}\,.$$

Alle Paare dieser Menge erhalten wir dann als

$$(\overline{x}_1\overline{x}_2x_4, \overline{x}_1x_2x_4) \qquad \text{und} \qquad (\overline{x}_1\overline{x}_3x_4, \overline{x}_1x_3x_4).$$

Man beachte, daß diese beiden Paare dieselbe Resolvente besitzen und diese Resolvente nur einmal aufgenommen wird. Nach einer weiteren Iteration erhalten wir damit

$$M_2 \;=\; \{\overline{x}_1x_4, \overline{x}_1\overline{x}_2x_3\}\,.$$

Nun gibt es offensichtlich keine Partner mehr und der Algorithmus terminiert. Damit enthält M_2 alle Primimplikanten von β. Wie man leicht erkennt, werden für ein Minimalpolynom alle diese Primimplikanten benötigt. Damit erhalten wir das Minimalpolynom

$$p \quad = \quad \overline{x}_1x_4 \vee \overline{x}_1\overline{x}_2x_3.$$

Man sollte allerdings beachten, daß es durchaus noch einen billigeren boolschen Ausdruck mit derselben Bedeutung gibt. Durch "Ausklammern" erhalten wir nämlich

$$\overline{x}_1\,(x_4 \vee \overline{x}_2x_3).$$

Die Kosten dieses Ausdrucks sind 5, während die Kosten von p'' 6 sind.

Wir haben schon bemerkt, daß es im allgemeinen schwierig ist, ein Minimalpolynom zu berechnen. Es gibt aber auch Fälle, in denen man ein Minimalpolynom direkt angeben kann.

3.61. Definition *Sei $l \in \mathbb{N}$. Die boolsche Funktion, die jedem $b \in \{0,1\}^k$, in dem wenigstens l Einsen vorkommen, den Wert 1, allen anderen b aber den Wert 0 zuordnet, heißt k-*stellige Schwellenfunktion *mit Schwelle l und wird mit s_l^k bezeichnet.*

Dann gilt der folgende Satz.

3.62. Satz *Sei $l \in \mathbb{N}$. Dann ist*

$$p \quad = \quad \bigvee_{1\le i_1<i_2<\ldots<i_l\le k} x_{i_1}x_{i_2}\ldots x_{i_l}$$

das eindeutig bestimmte Minimalpolynom von s_l^k.

Beweis: Offensichtlich ist $\alpha(p) = s_l^k$.

Seien $1 \le i_1 < i_2 < \ldots < i_l \le k$ und sei $m = x_{i_1}x_{i_2}\ldots x_{i_l}$. Dann ist m ein Primimplikant von s_l^k, weil für jedes echte Teilmonom m' von m ein Argument $b \in \{0,1\}^k$ existiert mit weniger als l Einsen und $\alpha(m')(b) = 1$. Wir zeigen, daß es keine weiteren Primimplikanten gibt. Angenommen, m wäre ein weiterer. Dann muß dieses Monom wenigstens l Literale enthalten, die nicht negierte Variablen sind. Das Teilmonom von m, das genau l solcher Variablen enthält, ist dann immer noch ein Implikant von s_l^k. Damit kommt jeder Primimplikant von s_l^k in p vor und wir müssen nur noch beweisen, daß wir keine weglassen dürfen. Sei $1 \le i_1 < i_2 < \ldots < i_l \le k$ und sei $b \in \{0,1\}^k$ so gewählt, daß genau die Koordinaten mit den Nummern $i_j, 1 \le j \le l$ den Wert Eins haben und alle anderen Null sind. Dann ist $s_l^k(b) = 1$ und $m = x_{i_1}x_{i_2}\ldots x_{i_l}$ ist das einzige Monom in p mit $\alpha(m)(b) = 1$. Also darf man m nicht weglassen. ■

Übungen

1. Beweisen Sie Satz 3.13.

2. Geben Sie jeweils einen Schaltkreis an, der folgende boolesche Funktionen berechnet (dabei sind x_i Eingabe- und y_i Ausgabewerte):

   (a) $y_1 \leftarrow \neg((x_1 \wedge x_2) \vee (\neg x_2 \vee x_3))$
   (b) $(y_1, y_2) \leftarrow ((x_1 \wedge \neg x_1) \vee x_2 \,,\, (x_1 \wedge x_2) \wedge \neg x_1)$
   (c) $y_1 \leftarrow (x_1 \wedge x_2) \vee (x_3 \wedge x_2) \vee (x_1 \wedge x_3)$

   Versuchen Sie, möglichst "billige" Schaltkreise zu finden.

3. Gegeben seien folgende boolsche Ausdrücke:

   (a) $x_1 \wedge (\neg x_2 \vee (x_1 \wedge x_2))$
   (b) $x_1 \vee \neg x_3 \wedge x_2 \vee x_4 \vee (\neg x_3) \wedge x_6 \vee (\neg x_1 \wedge x_2)$

   Geben Sie zu beiden boolschen Ausdrücken jeweils einen Baumschaltkreis und eine boolsche Funktion wie in der Vorlesung gezeigt an (mit allen Zwischenschritten). Bestimmen Sie dann die Kosten Ihrer Schaltkreise.

4. Berechnen Sie zu den folgenden boolschen Funktionen jeweils die disjunktive Normalform (dabei liefere die Operation mod2 die Werte 0 bzw. 1):

   (a) $\beta : \{0,1\}^4 \to \{0,1\}, \quad (b_1, \ldots, b_4) \mapsto (b_1 + b_2 + b_3 + b_4) \bmod 2.$
   (b) $\beta : \{0,1\}^4 \to \{0,1\}, \quad (b_1, \ldots, b_4) \mapsto (b_1 - b_2 * b_1 + b_3 - b_4 * b_2) \bmod 2.$
   (c) $\beta : \{0,1\}^4 \to \{0,1\}, \quad (b_1, \ldots, b_4) \mapsto (b_1^{b_2+1} + b_3) \bmod 2.$

5. Beweisen Sie: Für eine k-stellige boolsche Funktion β gilt $\beta(A(\beta)) = \beta$ (dabei ist $A(\beta)$ die disjunktive Normalform für β).

6. Beweisen Sie die beiden deMorgan-Regeln mit Hilfe der Axiome für eine boolsche Algebra.

7. Finden Sie zu folgenden boolschen Funktionen billige Polynome, die die Funktionen darstellen:

   (a) $\beta : \{0,1\}^4 \to \{0,1\}, \quad \beta(b_1, \ldots, b_4) = 1 \Leftrightarrow \sum_{i=1}^{4} b_i \geq 2.$
   (b) $\beta : \{0,1\}^4 \to \{0,1\}, \quad \beta(x) = 0 \Leftrightarrow x \in \{0000, 0010, 0011, 1000, 1100, 1101\}.$

8. Seien $(a_{n-1}, \ldots, a_0)$ und $(b_{n-1}, \ldots, b_0)$ die Binärdarstellungen zu $a, b \in \mathbb{N}$. Entwickeln Sie einen boolschen Schaltkreis zur simultanen Berechnung dreier Funktionen $f_i : \{0,1\}^{2n} \to \{0,1\}$ für $i = 1, 2, 3$ mit folgenden Eigenschaften:

   $$f_1 = 1 \Leftrightarrow a > b, \qquad f_2 = 1 \Leftrightarrow a = b, \qquad f_3 = 1 \Leftrightarrow a < b.$$

   Bestimmen Sie zuerst einen Schaltkreis für $n = 2$, der diese Funktionen simultan berechnet. Geben Sie dann für beliebiges n eine rekursive Beschreibung eines solchen Schaltkreises an, indem Sie das *Divide and Conquer* Prinzip benutzen. Versuchen Sie zu beweisen, daß Ihr Schaltkreis lineare Kosten und logarithmische Tiefe besitzt.

9. Sei β eine k-stellige boolsche Funktion. Dann definieren wir $x_i^1 := x_i$ und $x_i^0 := \neg x_i$. Die *konjunktive Normalform* zu β ist dann definiert als

$$\bigwedge_{b \in \{0,1\}^k, \beta(b)=0} \left(x_1^{\overline{b_1}} \vee x_2^{\overline{b_2}} \vee \ldots \vee x_k^{\overline{b_k}} \right).$$

Bestimmen Sie zu den boolschen Funktionen aus Aufgabe 2 die konjunktive Normalform.

10. Bestimmen Sie für folgende boolsche Funktionen alle Primimplikanten:

    (a) $\beta_a : \{0,1\}^4 \to \{0,1\}, \quad \beta_a(x) = 0 \Leftrightarrow x \in \{0100, 1010, 1100, 1101, 1111\}$

    (b) $\beta_b : \{0,1\}^4 \to \{0,1\}, \quad (b_1, \ldots, b_4) \mapsto (b_1 - b_2 * b_1 + b_3 - b_4 * b_2) \bmod 2$

    (c) $\beta_c : \{0,1\}^4 \to \{0,1\}, \quad \beta_c(x) = 0 \Leftrightarrow x \in \{1110, 0001, 0101\}$

    Veranschaulichen Sie die Funktion β_c am 4-dimensionalen Einheitswürfel und bestimmen Sie graphisch alle Minimalpolynome von β_c.

11. Bestimmen Sie für $l \leq k$ die Anzahl aller l-dimensionalen Teilwürfel des k-dimensionalen Einheitswürfels.

12. Konstruieren Sie eine 3-dimensionale boolsche Funktion, die genau zwei Minimalpolynome besitzt.

Kapitel 4

Addierer und Multiplizierer

4.1 Addierer

In diesem Abschnitt behandeln wir die Realisierung der Addition durch Schaltungen.

Dafür müssen wir zuerst Schaltkreisen mit beliebig vielen Ausgabeknoten Funktionen zuordnen. Es sei daran erinnert, daß jedem Knoten v eines Schaltkreises S in eindeutiger Weise ein boolscher Ausdruck $\phi(v)$ zugeordnet ist.

4.1. Definition *Es sei S ein Schaltkreis mit l Ausgabeknoten. Für $1 \leq i \leq l$ sei o_i der Ausgabeknoten mit der Bezeichnung y_i. Dann ist die Funktion $\beta(S)$ definiert als*

$$\beta(S) \quad = \quad \Big(\beta(\phi(o_1)), \ldots, \beta(\phi(o_l))\Big).$$

Wir sagen, daß S die Funktion $\beta(S)$ realisiert.

Jetzt definieren wir, welche Zahlen durch einen Bit-String dargestellt werden. Dabei seien alle auftretenden Zahlen nichtnegativ.

4.2. Definition *Sei $k \in \mathbb{N}$ und sei $a = (a_{k-1}, \ldots, a_0) \in \{0,1\}^k$. Dann setzen wir*

$$[a] \quad = \quad [a_{k-1}, \ldots, a_0] \quad = \quad \sum_{i=0}^{k-1} a_i \cdot 2^i.$$

Wir werden gelegentlich folgende Rechenregeln benötigen:

4.3. Lemma *Sei $k \in \mathbb{N}$ und sei $a = (a_{k-1}, \ldots, a_0) \in \{0,1\}^k$. Dann gilt für alle $1 \leq l \leq k$*

$$[a] \quad = \quad [a_{k-1}, \ldots, a_l, \underbrace{0, \ldots, 0}_{l}] + [a_{l-1}, \ldots, a_0] \quad = \quad 2^l \cdot [a_{k-1}, \ldots, a_l] + [a_{l-1}, \ldots, a_0].$$

Beweis:

$$
\begin{aligned}
[a] &= \sum_{i=0}^{k-1} a_i \cdot 2^i = \sum_{i=0}^{l-1} a_i \cdot 2^i + \sum_{i=l}^{k-1} a_i \cdot 2^i \\
&= [a_{l-1}, \ldots, a_0] + 2^l \cdot \sum_{i=0}^{k-1-l} a_{l+i} \cdot 2^i = 2^l \cdot [a_{k-1}, \ldots, a_l] + [a_{l-1}, \ldots, a_0].
\end{aligned}
$$

■

4.1.1 Halbaddierer und Volladdierer

Als nächstes konstruieren wir zwei Primitivaddierer:

4.4. Definition *Ein* Halbaddierer *ist ein Schaltkreis, der die Funktion*

$$\{0,1\}^2 \longrightarrow \{0,1\}^2, \qquad (a,b) \longmapsto (c,s)$$

mit $[c,s] = [a] + [b]$ *realisiert. Das Bit* c *heißt* Übertragsbit *oder auf Englisch* Carry Bit.

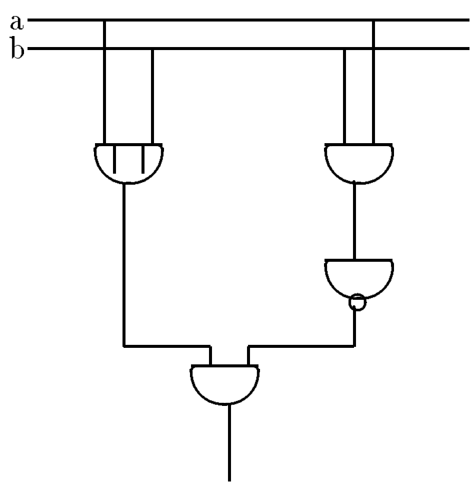
Wir konstruieren nun einen Halbaddierer. Es seien $a, b \in \{0,1\}$ und $[a] + [b] = [c,s]$. Offensichtlich gilt dabei $c = a \wedge b$. Zur Bestimmung einer boolschen Funktion für s führen wir die folgende Definition ein.

4.5. Definition *Die Funktion*

$$\otimes : \{0,1\}^2 \longrightarrow \{0,1\}, \qquad (a,b) \longmapsto a \otimes b \;=\; (a \wedge \neg b) \vee (\neg a \wedge b) \;=\; (a \vee b) \wedge \neg(a \wedge b)$$

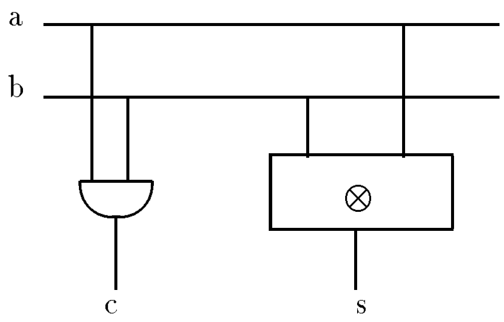
heißt exklusives oder *und auf Englisch* exclusive or.

Die Funktion $\otimes$ läßt sich durch folgenden Schaltkreis der Tiefe 4 und der Kosten 4 realisieren (beachte: bei unserer Tiefendefinition wird der Ausgabeknoten mitgezählt):



Einen solchen Baustein bezeichnen wir im folgenden immer als XOR-Gatter und stellen es als $\boxed{\otimes}$ dar. Damit wissen wir auch, wie wir das Bit s bestimmen können. Wir erhalten nämlich $s = a \otimes b$.

Mit Hilfe eines UND-Gatters und eines XOR-Bausteins können wir also leicht einen Halbaddierer der Tiefe 4 und der Kosten 5 bauen, den wir im folgenden mit $\boxed{\text{HA}}$ bezeichnen.



Man sollte beachten, daß man, wenn man den XOR-Baustein nicht als Black Box verwendet, die Kosten des Halbaddierers sogar auf 4 reduzieren kann. Innerhalb des XOR-Bausteins berechnet man nämlich schon $a \wedge b$ und dieses Resultat kann man direkt in den Ausgang c führen. Wir werden aber im folgenden weiterhin den modularen Aufbau von Schaltkreisen unterstützen.

4.6. Definition *Ein* Volladdierer *ist ein Schaltkreis, der die Funktion*

$$\{0,1\}^3 \longrightarrow \{0,1\}^2, \qquad (a,b,u) \longmapsto (c,s)$$

mit $[c,s] = [a] + [b] + [u]$ *realisiert.*

Aus zwei Halbaddierern kann man durch Ausnutzung der Aussage des folgenden Lemmas einen Volladdierer bauen.

4.7. Lemma *Seien* $a, b, u \in \{0,1\}$. *Sei* $[a] + [b] = [c_1, s_1]$ *und* $[s_1] + [u] = [c_2, s_2]$. *Dann gilt*

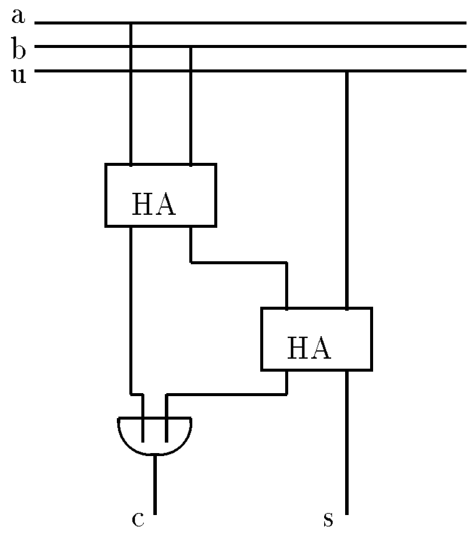
$$[a] + [b] + [u] \quad = \quad [c_1 \vee c_2, s_2].$$

Beweis: Es gilt nach Voraussetzung

$$[a] + [b] + [u] \;=\; [c_1, s_1] + [u] \;=\; [c_1, 0] + [s_1] + [u] \;=\; [c_1, 0] + [c_2, s_2].$$

Da c_1 und c_2 nicht gleichzeitig Eins sein können, folgt $[c_1, 0] + [c_2, s_2] = [c_1 \vee c_2, s_2]$ wie behauptet. ■

Der damit entwickelte Baustein wird im folgenden mit $\boxed{\text{FA}}$ für Full Adder bezeichnet. Er besitzt die Tiefe 8 und Kosten 11.



4.1.2 Carry Chain Adder

Wir konstruieren nun einen Addierer, der die Schulmethode zur Addition verwendet. Die Idee ist ganz einfach:

Angenommen, man möchte $a = [10111]$ und $b = [00111]$ addieren. Man addiert die letzten beiden Bits (auf Englisch heißen die Least Significant Bits). Dann ergibt sich daraus das letzte Bit der Summe und ein Übertrag, der 0 oder 1 sein kann. Anschließend muß man nur noch zwei 4-Bit-Zahlen und einen Übertrag addieren.Dazu addiert man die beiden letzten Bits der neuen Zahlen und den Übertrag. Daraus ergibt sich das vorletzte Bit der Summe und ein Übertrag. Dieses Verfahren kann man fortsetzen, bis die Summe gefunden ist.

Das Problem, zwei k-Bit Zahlen und einen Übertrag zu addieren, kann man also auf das Problem, drei Bits zu addieren und zwei $k-1$-Bit Zahlen und einen Übertrag zu addieren, zurückführen. Drei Bits kann man mit Hilfe eines Volladdierers addieren. Damit kann man mit Hilfe von k Volladdierern einen k-Bit Addierer konstruieren. Wir beschreiben diese Methode nun etwas formaler.

4.8. Definition *Sei $k \in \mathbb{N}$. Die Funktion $\{0,1\}^{2k+1} \to \{0,1\}^{k+1}$, die einem Paar (a,b,u) mit $a, b \in \{0,1\}^k$ und $u \in \{0,1\}$ ein Paar (c,s) mit $c \in \{0,1\}$, $s \in \{0,1\}^k$ und $[c,s] = [a]+[b]+[u]$ zuordnet, benenne ich mit σ_k.*

Aus einem Volladdierer kann man induktiv einen Schaltkreis konstruieren, der die Funktion σ_k für beliebiges k realisiert. Hierzu benötigt man folgendes Resultat:

4.9. Lemma *Es sei $k \in \mathbb{N}$, $a = (a_{k-1}, \ldots, a_0), b = (b_{k-1}, \ldots, b_0) \in \{0,1\}^k$ und $u \in \{0,1\}$. Weiterhin sei $a' = (a_{k-1}, \ldots, a_1), b' = (b_{k-1}, \ldots, b_1) \in \{0,1\}^{k-1}$ und $[a_0]+[b_0]+[u] = [c_0, s_0]$ mit $c_0, s_0 \in \{0,1\}$. Dann gilt*

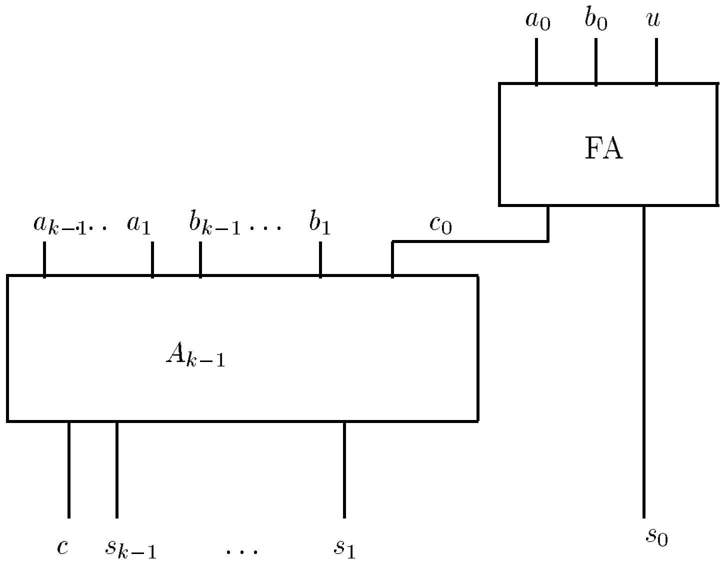
$$[a]+[b]+[u] \quad = \quad 2 \cdot \Big([a'] + [b'] + [c_0]\Big) + [s_0].$$

Beweis: Es ist $[a] = 2 \cdot [a'] + [a_0]$ und $[b] = 2 \cdot [b'] + [b_0]$. Hieraus folgt

$$\begin{aligned} [a] + [b] + [u] &= 2 \cdot \Big([a'] + [b']\Big) + [a_0] + [b_0] + [u] \\ &= 2 \cdot \Big([a'] + [b']\Big) + [c_0, s_0] \\ &= 2 \cdot \Big([a'] + [b']\Big) + 2 \cdot [c_0] + [s_0] \\ &= 2 \cdot \Big([a'] + [b'] + [c_0]\Big) + [s_0]. \end{aligned}$$

■

Die gerade bewiesene Formel rechtfertigt, daß man einen Schaltkreis, der σ_k realisiert, mit Hilfe eines Schaltkreises konstruieren kann, der σ_{k-1} realisiert. Sei A_{k-1} etwa ein Addierer, der σ_{k-1} berechnet. Offensichtlich wird A_1 gegeben durch einen Volladdierer. Wir konstruieren dann A_k aus A_{k-1}:



Ein so konstruierter Addierer heißt k-Bit *Carry Chain Adder*, weil der Übertrag die Kette aller Volladdierer durchlaufen muß. Diesen Addierer kann man als Graph auffassen, dessen Knoten Gatter oder Volladdierer sind. Dann ergibt sich folgender Satz:

4.10. Satz *Benutzt man den oben konstruierten Volladdierer, so hat der k-Bit Carry Chain Adder Tiefe $7k+1$ und Kosten $11k$.*

Beweis: Der 1-Bit Carry Chain Adder ist ein Volladdierer. Also ist $L_1 = 11$ und $T_1 = 8$. Außerdem gilt $L_k = L_{k-1} + 11$ und $T_k = T_{k-1} + 7$, woraus die Behauptung durch Induktion folgt. ■

Damit ist das Problem, einen k-Bit Addierer zu konstruieren, zwar im Prinzip gelöst, dieser Addierer ist aber sehr teuer und sehr langsam.

4.1.3 Conditional Carry Adder

In diesem Abschnitt wird ein Addierer wesentlich geringerer Tiefe konstruiert. Wir nehmen hierzu an, daß die binäre Länge k der verwendeten Zahlen eine Potenz von 2 ist, also $k = 2^n$ mit $n \in \mathbb{N}$.

Wir erklären wiederum zuerst die Idee, die im wesentlichen darin besteht, die Addition voller Wörter auf die Addition der oberen Hälften und der unteren Hälften zurückzuführen.

4.11. Definition *Für $c = (c_{k-1}, \ldots, c_0) \in \{0,1\}^k$ setzt man $c_h = (c_{k-1}, \ldots, c_{k/2})$ und $c_l = (c_{k/2-1}, \ldots, c_0)$. c_h heißt* Hochwort *von c und c_l heißt* Tiefwort *von c.*

4.12. Lemma *Sei $a, b \in \{0,1\}^k$, $u \in \{0,1\}$ und sei*

$$[a] + [b] + [u] \quad = \quad [c, s]$$

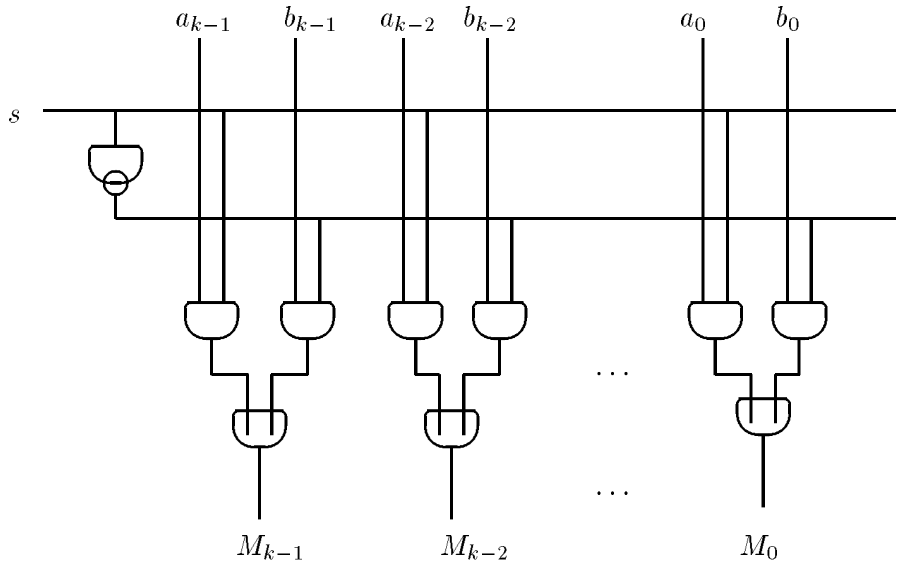
mit $c \in \{0,1\}$ und $s \in \{0,1\}^k$. Dann gilt

$$[a_l] + [b_l] \quad = \quad [c_l, s_l]$$

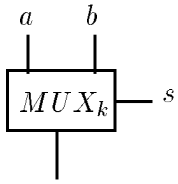
mit $c_l \in \{0,1\}$. Weiter gilt

$$[c, s_h] \quad = \quad \begin{cases} [a_h] + [b_h] & \text{falls } c_l = 0 \\ [a_h] + [b_h] + 1 & \text{falls } c_l = 1 \end{cases}$$

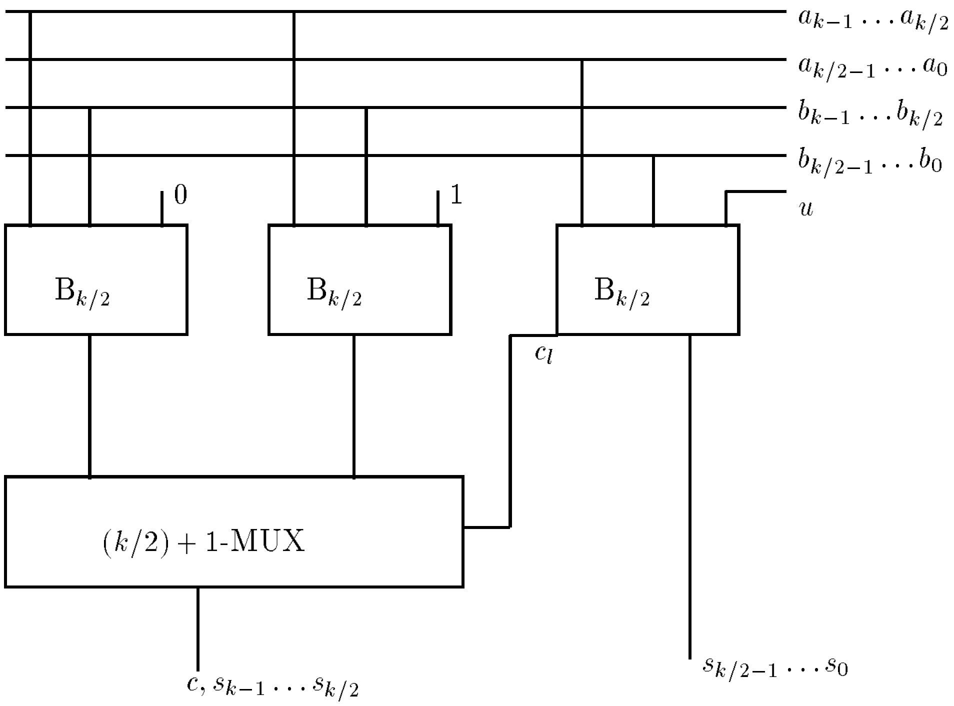
Die letzten $k/2$ Bits der Summe kann man also schon durch Addition der beiden Zahlen $[a_l]$ und $[b_l]$ gewinnen. Für $[c, s_h]$ gibt es aber zwei Möglichkeiten, je nachdem, ob c_l gleich 0 oder gleich 1 ist. Die Idee besteht darin, die Additionen der Hoch- und Tiefwörter parallel vorzunehmen. Da man aber vorher nicht weiß, welches Carry Bit c_l bei der Addition der Tiefwörter auftritt, berechnet man beide möglichen Ergebnisse und wählt dann in Abhängigkeit von c_l das richtige Hochwort aus. Um diese Auswahl zu ermöglichen, benötigt man noch einen speziellen Schaltkreis, der in Abhängigkeit eines Steuerbits s aus zwei k-Bit Strings einen auswählt. Dieser Schaltkreis sieht so aus:



Der Baustein heißt k-Bit *Multiplexer*, kurz k-*MUX*. Seine Kosten sind $3k+1$, seine Tiefe ist 4. Wenn wir diesen Baustein im folgenden als Black Box (wie in der folgenden Zeichnung) verwenden, so gehen wir immer davon aus, daß bei Steuersignal $s=1$ die k Bits von b und bei $s=0$ entsprechend die k Bits von a ausgewählt werden:



Mittels der Formeln aus Lemma 4.12 kann man dann rekursiv einen Schaltkreis B_k konstruieren, der die Funktion σ_k realisiert. Für B_1 nimmt man einen gewöhnlichen Volladdierer. Dieser hat Kosten 11 und Tiefe 8. Angenommen, man hat schon $B_{k/2}$ konstruiert. Aus drei Kopien dieses Schaltkreises und aus einem $\frac{k}{2}+1$-MUX kann man dann auf folgende Weise B_k konstruieren:



4.13. Satz *Bei Verwendung des im letzten Abschnitt konstruierten Volladdierers gilt*

$$T(B_k) \quad = \quad 11 + 3 \cdot \log_2 k \qquad \textit{und} \qquad L(B_k) \quad = \quad 16 \cdot k^{\log_2 3} - 3 \cdot k - 2.$$

Beweis: Wie oben schon bemerkt, gilt $L(B_1) = 11$ und $T(B_1) = 8$. Für die Tiefe gilt außerdem

$$T(B_k) \quad = \quad T(B_{k/2}) + 3,$$

woraus durch Induktion die erste Behauptung folgt. Für die Kosten gilt (beachte: $k = 2^n$)

$$\begin{aligned}
L(B_k) &= 3 \cdot L(B_{k/2}) + 3k/2 + 4 \\
&= 3 \cdot \Big(3 \cdot L(B_{k/4}) + 3k/4 + 4\Big) + 3k/2 + 4 \\
&\vdots \\
&= 3^n \cdot L(B_1) + \underbrace{\sum_{i=0}^{n-1} 4 \cdot 3^i}_{=2\cdot(3^n-1)} + \underbrace{\sum_{i=1}^{n} \left(\frac{3}{2}\right)^i \cdot 2^n}_{=2^n\cdot 3\cdot((3/2)^n-1)} \\
&= 11 \cdot 3^n + 2 \cdot 3^n - 2 + 3 \cdot 3^n - 3 \cdot 2^n.
\end{aligned}$$

Nutzen wir nun aus, daß $2^n = k$ und $3^n = k^{\log_2 3}$ gilt, so folgt daraus die Behauptung des Satzes. ■

4.2 Darstellung negativer Zahlen

In diesem Abschnitt sei $k \in \mathbb{N}$ fest gewählt. Eine Möglichkeit, negative Zahlen darzustellen, besteht darin, dem darstellenden Bitstring ein Sign-Bit voranzustellen. Der Bitstring $(a_k, \ldots, a_0) \in \{0,1\}^{k+1}$ stellt dann die Zahl

$$(-1)^{a_k} \cdot \sum_{i=0}^{k-1} a_i \cdot 2^i$$

dar. Dies hat aber den Nachteil, daß man die Addition zweier Signed Integers nicht so leicht auf die Addition zweier Unsigned Integers zurückführen kann. Das ist anders bei der nun eingeführten *Komplementdarstellung.*

4.14. Definition *Sei* $(a_k, \ldots, a_0) \in \{0,1\}^{k+1}$. *Dann setzen wir*

$$\langle a_k, \ldots, a_0 \rangle \quad = \quad -[a_k] \cdot 2^k + [a_{k-1}, \ldots, a_0].$$

4.15. Beispiel Es ist $\langle 0101 \rangle = 5$ und $\langle 1101 \rangle = 5 - 8 = -3$. Man beachte weiterhin, daß $\langle 1 \ldots 1 \rangle = -1$ gilt.

Diese Darstellung ist eindeutig, insbesondere wird die Null eindeutig als $(0 \ldots 0)$ dargestellt. Weiterhin gilt offensichtlich $-2^k \leq \langle a_k, \ldots, a_0 \rangle \leq 2^k - 1$, d.h. das Intervall aller mit $k+1$ Bits darstellbarer Zahlen ist nicht symmetrisch. Es gilt dann folgende Rechenregel:

4.16. Lemma *Sei* $a = (a_k, \ldots, a_0) \in \{0,1\}^{k+1}$. *Dann gilt*

$$\langle a \rangle \quad = \quad \langle a_k, a_{k-2}, a_{k-3}, \ldots, a_0 \rangle + 2^{k-1} \cdot \Big([a_{k-1}] - [a_k]\Big).$$

Beweis: Zum Beweis benutzt man die Definition und rechnet dann beide Seiten aus. ■

Für eine in Komplementdarstellung gegebene Zahl a kann man leicht die Komplementdarstellung von $-a$ berechnen, wie das folgende Lemma 4.19 zeigt.

4.17. Definition *Sei $a = (a_k, \ldots, a_0) \in \{0,1\}^{k+1}$. Dann setzen wir $\overline{a} = (\neg a_k, \ldots, \neg a_0)$.*

4.18. Lemma *Seien $a, b \in \{0,1\}$. Dann ist $[a] - [b] = [\overline{b}] - [\overline{a}]$.*

4.19. Lemma *Sei $a \in \{0,1\}^{k+1}$. Dann gilt $\langle \overline{a} \rangle = -\langle a \rangle - 1$.*

Beweis: Der Beweis erfolgt mittels Induktion über k. Für $k = 0$ kann man die Aussage leicht verifizieren. Angenommen, $k \in \mathbb{N}$ und die Aussage stimmt für $k - 1$. Sei $a = (a_k, \ldots, a_0)$. Dann ist nach Lemma 4.16 und Lemma 4.18

$$\begin{aligned}
\langle \overline{a} \rangle &= \langle \overline{a}_k, \overline{a}_{k-2}, \overline{a}_{k-3}, \ldots, \overline{a}_0 \rangle + 2^{k-1} \cdot \Big([\overline{a}_{k-1}] - [\overline{a}_k] \Big) \\
&= -\langle a_k, a_{k-2}, a_{k-3}, \ldots, a_0 \rangle - 1 + 2^{k-1} \cdot \Big([a_k] - [a_{k-1}] \Big) \\
&= -\langle a_k, a_{k-2}, a_{k-3}, \ldots, a_0 \rangle - 2^{k-1} \cdot \Big([a_{k-1}] - [a_k] \Big) - 1 \\
&= -\langle a_k, a_{k-1}, \ldots, a_0 \rangle - 1.
\end{aligned}$$

■

Die Addition zweier Zahlen in Komplementdarstellung kann man auf die Addition zweier Binärzahlen zurückführen. Dabei tritt noch eine Komplikation auf. In realen Rechnern haben die Signed Integers eine feste Länge. Es kann sein, daß man zur Darstellung der Summe zweier Zahlen in Komplementdarstellung mehr Bits benötigt. Das nennt man *Bereichsüberschreitung*. Dies muß man feststellen und dem Benutzer mitteilen können. Die Reduktion der Addition von Zahlen in Komplementdarstellung auf die Addition von Unsigned Integers und die Kontrolle der Bereichsüberschreitung wird durch folgendes Ergebnis möglich:

4.20. Lemma *Es sei $a = (a_k, \ldots, a_0), b = (b_k, \ldots, b_0) \in \{0,1\}^{k+1}$, $u \in \{0,1\}$ und es sei $[a_k, a] + [b_k, b] + [u] = [s]$ mit $s = (s_{k+2}, \ldots, s_0) \in \{0,1\}^{k+3}$. Dann gilt*

1. *$-2^k \leq \langle a \rangle + \langle b \rangle + [u] < 2^k$ genau dann, wenn $s_k = s_{k+1}$.*
2. *Falls $s_k = s_{k+1}$ ist, so folgt $\langle a \rangle + \langle b \rangle + [u] = \langle s_k, \ldots, s_0 \rangle$.*

Beweis: Es ist

$$[a_{k-1}, \ldots, a_0] + [b_{k-1}, \ldots, b_0] + [u] = [c, s_{k-1}, \ldots, s_0]$$

mit $c \in \{0,1\}$. Daher gilt

$$\begin{aligned}
\langle a \rangle + \langle b \rangle + [u] &= -2^k \cdot \Big([a_k] + [b_k] \Big) + \Big([a_{k-1}, \ldots, a_0] + [b_{k-1}, \ldots, b_0] \Big) + [u] \\
&= 2^k \cdot \Big([c] - [a_k] - [b_k] \Big) + [s_{k-1}, \ldots, s_0].
\end{aligned}$$

Eine Bereichsüberschreitung liegt genau dann vor, wenn entweder $[c] - [a_k] - [b_k]$ größer als Null oder kleiner als -1 ist. Dies ist genau dann der Fall, wenn $c = 0$ und $a_k = b_k = 1$ oder $c = 1$ und $a_k = b_k = 0$ gilt. Da aber

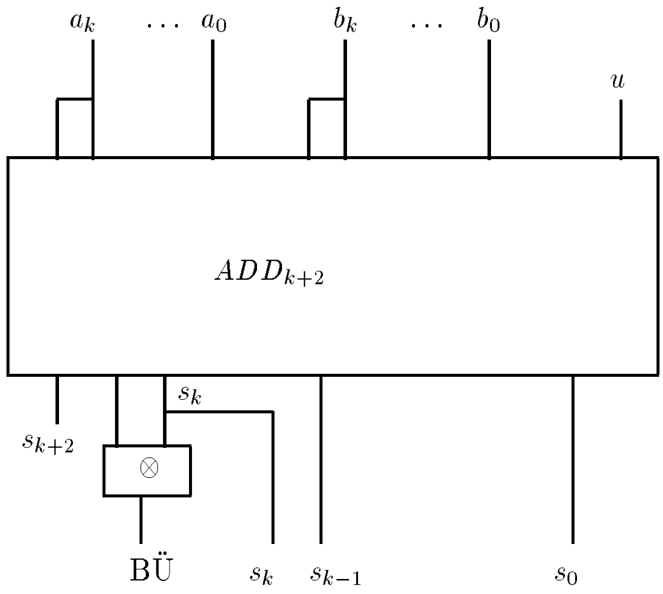
$$[s_{k+2}, s_{k+1}, s_k] = [a_k, a_k] + [b_k, b_k] + [c]$$

ist, kann man leicht verifizieren, daß genau in den Fällen der Bereichsüberschreitung die Werte s_{k+1} und s_k verschieden sind. Außerdem rechnet man leicht nach, daß in den Fällen, wo keine Bereichsüberschreitung vorliegt, die Summe wie behauptet berechnet wird. ■

Um einen Addierer für Zahlen in Komplementdarstellung mit Bereichskontrolle konstruieren zu können, brauchen wir noch einen Schaltkreis, der die Gleichheit von zwei Bits testet.

4.21. Lemma *Es seien $a, b \in \{0,1\}$. Dann gilt $a = b$ genau dann, wenn $(a \wedge b) \vee \neg(a \vee b) = \neg(a \otimes b) = 1$ ist.*

Ein Addierer für 2 $(k+1)$-Bit Komplement Zahlen und einen Übertrag u sieht also so aus. Dabei sei ADD_{k+2} ein Addierer für $(k+2)$-Bit Zahlen ohne Vorzeichen sowie ein Carry . Bei einer Addition zweier Zahlen a und b in Komplementdarstellung muß man den Wert von u in obigem Lemma natürlich als 0 wählen.



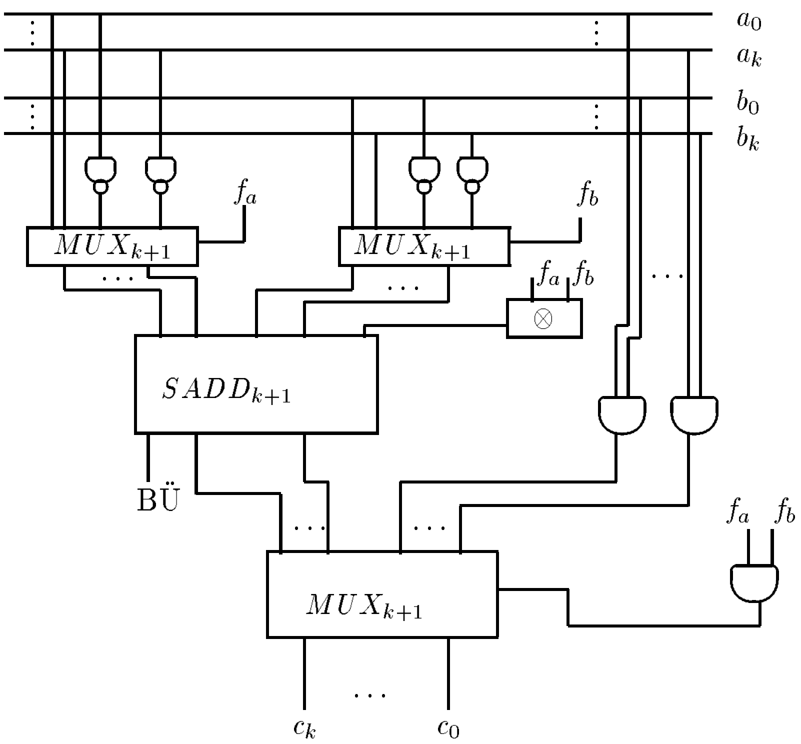
Hieraus kann man eine Arithmetic Logic Unit (ALU) machen, die bei Eingabe von zwei Bitstrings $a, b \in \{0,1\}^{k+1}$ und zwei Steuerbits f_a, f_b folgende Ausgabe $c \in \{0,1\}^{k+1}$ produziert:

f_a	f_b	c
0	0	$\langle a\rangle + \langle b\rangle$
1	0	$-\langle a\rangle + \langle b\rangle$
0	1	$\langle a\rangle - \langle b\rangle$
1	1	$a \wedge b$

Dazu beachte man, daß nach Lemma 4.19 gilt

$$\langle a \rangle - \langle b \rangle \quad = \quad \langle a \rangle + \langle \overline{b} \rangle + 1.$$

Dann können wir den gerade angegebenen Addierer $SADD_{k+1}$ für Signed Integers folgendermaßen benutzen:



4.3 Multiplizierer

Es seien $k, l \in \mathbb{N}$ und $a \in \{0,1\}^k, b \in \{0,1\}^l$. Offensichtlich ist die Vorzeichenbehandlung bei der Multiplikation kein Problem. Daher nehmen wir im folgenden an, daß die Bitstrings a und b ganze nichtnegative Zahlen darstellen. Dann gilt $0 \leq [a] \cdot [b] < 2^k \cdot 2^l = 2^{k+l}$, aber es gibt auch Bitstrings a, b mit $[a] \cdot [b] \geq 2^{k+l-1}$. Dies rechtfertigt die folgende Definition.

4.22. Definition *Es sei $k, l \in \mathbb{N}$. Ein (k,l)-*Multiplizierer *ist ein Schaltkreis, der die Funktion $\{0,1\}^k \times \{0,1\}^l \to \{0,1\}^{k+l}$, $(a,b) \mapsto c$ mit $[c] = [a] \cdot [b]$ realisiert.*

4.3.1 Ein einfacher Multiplizierer

In diesem Abschnitt benutzen wir die Schulmethode zur Multiplikation, um induktiv einen Multiplizierer zu bauen. Mit nur einem AND-Gatter kann man einen $(1,1)$-Multiplizierer bauen. Allgemeiner kann man mit k AND-Gattern einen $(k,1)$-Multiplizierer bauen:

4.23. Lemma *Sei $a = (a_{k-1}, \ldots, a_0) \in \{0,1\}^k$ und $b \in \{0,1\}$. Dann gilt*

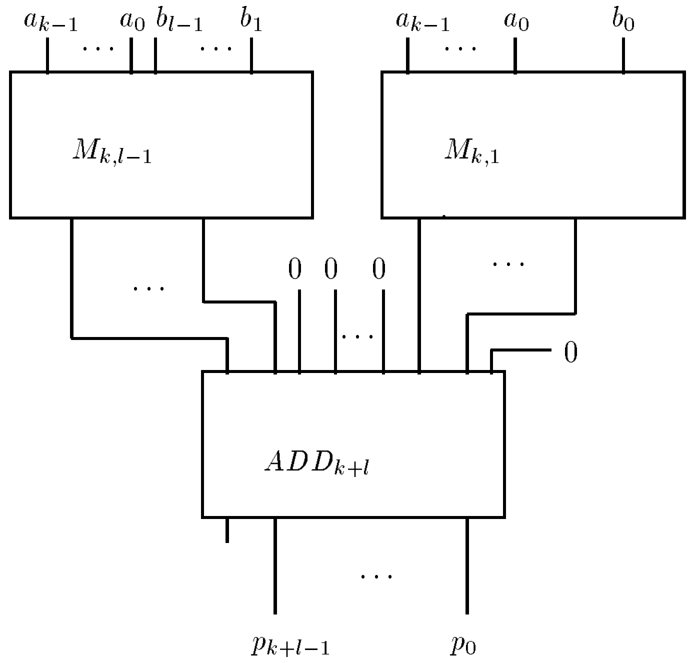
$$[a] \cdot [b] \quad = \quad [0, a_{k-1} \wedge b, \ldots, a_0 \wedge b].$$

Angenommen, man hat einen $(k, l-1)$-Multiplizierer. Dann kann man daraus durch folgende Formel einen (k, l)-Multiplizierer bauen.

4.24. Lemma *Sei $a \in \{0,1\}^k$ und $b = (b_{l-1}, \ldots, b_0) \in \{0,1\}^l$. Dann gilt*

$$[a] \cdot [b] \quad = \quad 2 \cdot [a] \cdot [b_{l-1}, \ldots, b_1] + [a] \cdot [b_0].$$

Damit erhalten wir den folgenden Multiplizierer $M_{k,l}$:



4.25. Satz *Bei Verwendung des Carry Chain Adders aus Abschnitt 4.1.2 gilt*

$$L(M_{k,l}) \quad = \quad O(k \cdot l + l^2) \qquad \textit{und} \qquad T(M_{k,l}) \quad = \quad O(k \cdot l + l^2).$$

Beweis: Es gilt $L(M_{k,1}) = k$ und $T(M_{k,1}) = 2$. Außerdem gelten die Rekursionsformeln

$$L(M_{k,l}) \; = \; L(M_{k,l-1}) + L(A_{k+l}) + L(M_{k,1}) \quad \text{und} \quad T(M_{k,l}) \; = \; T(M_{k,l-1}) + T(A_{k+l}) + 1.$$

Wir führen den Beweis jetzt nur für die Formel für $T(M_{k,l})$, die Kosten können analog berechnet werden. Nach Satz 4.10 hat A_{k+l} die Tiefe $7(k+l) - 1$. Damit erhalten wir

$$\begin{aligned}
T(M_{k,l}) \quad &= \quad T(M_{k,l-1}) + 7 \cdot (k+l) \\
&\vdots \\
&= \quad T(M_{k,1}) + 7 \cdot \sum_{i=2}^{l} (k+i) - (l-2) \\
&= \quad 2 + 7 \cdot k \cdot (l-1) + \frac{l \cdot (l+1)}{2} - (l-1) \\
&= \quad O(k \cdot l + l^2).
\end{aligned}$$

■

4.3.2 Ein Multiplizierer geringerer Tiefe

In diesem Abschnitt wollen wir dieselbe Idee etwas modifizieren, um damit einen Multiplizierer geringerer Tiefe zu erhalten.

Seien dazu die Bitstrings $a = (a_{k-1}, \ldots, a_0) \in \{0,1\}^k$ und $b = (b_{l-1}, \ldots, b_0) \in \{0,1\}^l$ gegeben. Wir möchten $c = (c_{k+l-1}, \ldots, c_0) \in \{0,1\}^{k+l}$ mit $[a] \cdot [b] = [c]$ berechnen. Dazu benutzen wir die Formel

$$[a] \cdot [b] \quad = \quad \sum_{j=0}^{l-1} [a][b_j] \cdot 2^j \quad = \quad \sum_{j=0}^{l-1} [a_{k-1} \wedge b_j, \ldots, a_0 \wedge b_j] \cdot 2^j .$$

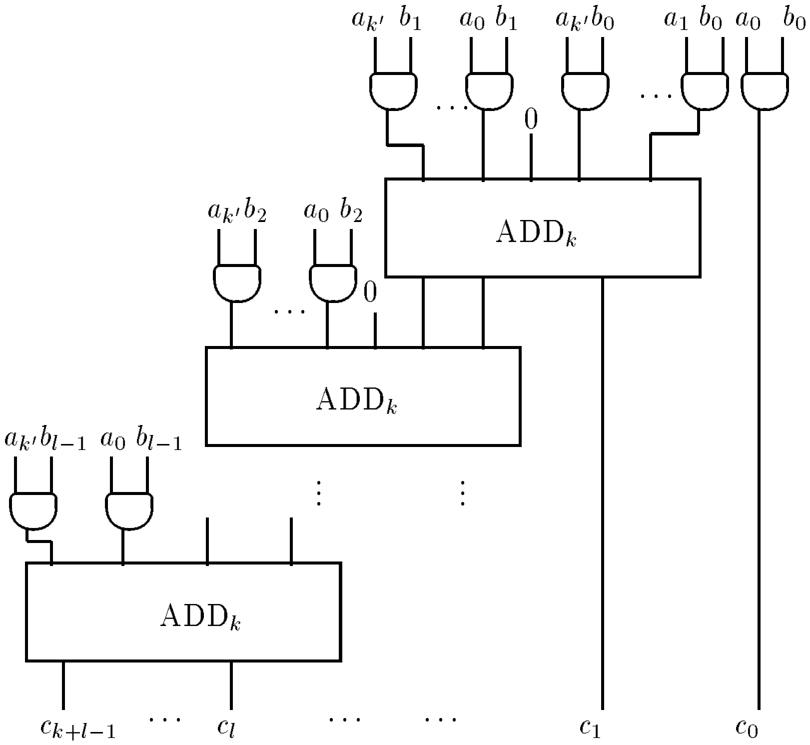
Angenommen, man hat einen Bitstring $(c_{i+k-1}, \ldots, c_0)$ berechnet mit

$$[c_{i+k-1}, \ldots, c_0] \quad = \quad \sum_{j=0}^{i-1} [a] \cdot [b_j] \cdot 2^j .$$

Dann gilt

$$\begin{aligned} \sum_{j=0}^{i} [a][b_j] 2^j \quad &= \quad [c_{i+k-1}, \ldots, c_0] + [a] \cdot [b_i] \cdot 2^i \\ &= \quad 2^i \cdot \Big([c_{i+k-1}, c_{i+k-2}, \ldots, c_i] + [a_{k-1} \wedge b_i, \ldots, a_0 \wedge b_i] \Big) + [c_{i-1}, \ldots, c_0]. \end{aligned}$$

Um also den neuen Bitstring $(c_{i+k}, \ldots, c_0)$ zu erzeugen, muß man zwei k-Bit Zahlen addieren. Für $i = l-1$ sind wir fertig und wir haben das Ergebnis $(c_{k+l-1}, \ldots, c_0)$ berechnet. Für einen Iterationsschritt verwendet man einen Carry Chain Adder und kommt so zu dem folgenden Multiplizierer (dabei sei $k' = k-1$):



Jetzt sollen Tiefe und Kosten dieses Multiplizierers abgeschätzt werden. Hierzu wird folgendes Ergebnis benötigt.

4.26. Satz *Die Tiefe des oben angegebenen Multiplizierers ist* $\leq 7 \cdot (k+l)+1$ *und seine Kosten sind* $12 \cdot k \cdot l$.

Beweis: Da man l k-Bit Carry Chain Addierer und kl AND-Gatter braucht, sind die Kosten $12 \cdot kl$.

Seien $A_0, A_1, \ldots, A_{l-1}$ die verwendeten Carry Chain Addierer. Wir zeigen durch Induktion, daß die Tiefe des j-ten Ausgangsknotens von A_m höchstens $7(j+m+1)+1$ ist. Sei also zuerst $m = 0$. Der längste Pfad zum j-ten Ausgangsknoten von A_0 führt vom 0-ten Eingangsknoten durch $j+1$ Volladdierer. Die Länge dieses Pfades ist höchstens $7(j+1)+1$. Angenommen, die Behauptung gelte für $m-1$. Der Pfad vom i-ten Ausgangsknoten von A_{m-1} zum j-ten Ausgangsknoten von A_m führt durch $j-i+1$ Volladdierer. Damit ist die Tiefe des j-ten Ausgangsknotens von A_m höchstens $7(i+m)+7(j-i+1)+1 = 7(j+m+1)+1$. ■

4.3.3 Ein (k,k)-Multiplizierer der Tiefe $O((\log k)^2)$

Wir wollen jetzt einen Multiplizierer noch geringerer Tiefe konstruieren. Dafür gehen wir wieder davon aus, daß k eine Potenz von 2 ist, also $k = 2^n$ mit $n \in \mathbb{N}$.

Dem neuen Multiplizierer liegen die folgende Formeln zugrunde. Seien $a, b \in \{0,1\}^k$. Dann gilt

$$[a]\cdot[b] \quad = \quad 2^k \cdot [a_h]\cdot[b_h] + 2^{k/2} \cdot \Big([a_h]\cdot[b_l] + [a_l]\cdot[b_h]\Big) + [a_l]\cdot[b_l]$$

sowie

$$[a_h]\cdot[b_l] + [a_l]\cdot[b_h] \quad = \quad \Big([a_h]-[a_l]\Big)\cdot\Big([b_l]-[b_h]\Big) + [a_h]\cdot[b_h] + [a_l]\cdot[b_l].$$

Hierin ist

$$\Big|[a_h]-[a_l]\Big| \quad < \quad 2^{k/2} \qquad \text{sowie} \qquad \Big|[b_l]-[b_h]\Big| \quad < \quad 2^{k/2}.$$

Man bestimmt also zuerst

$$p_h \quad = \quad [a_h]\cdot[b_h] \qquad \text{und} \qquad p_l \quad = \quad [a_l]\cdot[b_l].$$

Dies sind zwei Multiplikationen von $k/2$-Bit Zahlen. Dann berechnet man

$$d_a \quad = \quad [a_h]-[a_l], \qquad d_b \quad = \quad [b_l]-[b_h].$$

Das sind zwei Subtraktionen von $k/2$-Bit Zahlen. Weiter bestimmt man $s = [d_a]\cdot[d_b]+[p_h]+[p_l]$. Dies bedeutet eine Multiplikation zweier $k/2$-Bit Zahlen sowie zwei Additionen von $k+1$-Bit Zahlen. Schließlich wird das Ergebnis bestimmt als

$$[a]\cdot[b] \quad = \quad 2^k \cdot p_h + 2^{k/2}\cdot s + p_l$$

und dafür braucht man wieder zwei Additionen von k-Bit Zahlen. Damit hat man die Aufgabe, eine k-Bit Multiplikation auszuführen, darauf zurückgeführt, drei $k/2$-Bit Multiplikationen und einige Additionen auszuführen.

Die Kosten und die Tiefe dieses Multiplizierers werden in dem folgenden Satz beschrieben:

4.27. Satz *Es gilt* $T(M_k) = O((\log k)^2)$ *und* $L(M_k) = O(k^{\log_2 3})$.

Beweis: Bestimmen wir zuerst die Tiefe. In der Rekursion führen wir drei Multiplikationen von halb so großen Zahlen parallel aus. Danach werden einige Additionen bzw. Subtraktionen von $k/2$-Bit Zahlen durchgeführt, was mit einem CCA-Adder in logarithmischer Tiefe geht. Also erhalten wir mit einer geeigneten Konstanten c

$$\begin{aligned} T(M_k) &= T(M_{k/2}) + c \cdot \log(k/2) \\ &= T(M_{k/4}) + c \cdot (\log(k/4) + \log(k/2)) \\ &\vdots \\ &= T(M_1) + c \cdot \sum_{i=1}^{\log k - 1} \log\left(\frac{k}{2^i}\right) \\ &= O((\log k)^2). \end{aligned}$$

Betrachten wir noch kurz die Kosten. Sei dazu $k = 2^n$. In der Übung wurde gezeigt, daß es einen CCA-Adder mit logarithmischer Tiefe und linearen Kosten gibt. Benutzen wir diesen Addierer und beachten wir, daß auch alle anderen benutzten Bausteine wie MUXe und Subtrahierer lineare Kosten besitzen, so erkennen wir, daß es eine Konstante d gibt mit

$$\begin{aligned} L(M_k) &= 3 \cdot L(M_{k/2}) + d \cdot 2^{n-1} \\ &= 3 \cdot \left(3 \cdot L(M_{k/4}) + d \cdot 2^{n-2}\right) + d \cdot 2^{n-1} \\ &\vdots \\ &= 3^n \cdot L(M_1) + d \cdot \sum_{i=1}^{n} 2^{n-i} \cdot 3^i \\ &= d' \cdot k^{log_2 3} + d \cdot 2^n \cdot \sum_{i=1}^{n} \left(\frac{3}{2}\right)^i \\ &= O(k^{\log_2 3}). \end{aligned}$$

■

4.3.4 Ein (k, k)-Multiplizierer der Tiefe $O(\log k)$

Sei k eine Zweierpotenz und seien die Bitstrings $a = (a_{k-1}, \ldots, a_0), b = (b_{k-1}, \ldots, b_0) \in \{0,1\}^k$ gegeben. Wir möchten $p = (p_{k+l-1}, \ldots, p_0) \in \{0,1\}^{2k}$ mit $[a] \cdot [b] = [p]$ berechnen. Wieder benutzen wir die Formel

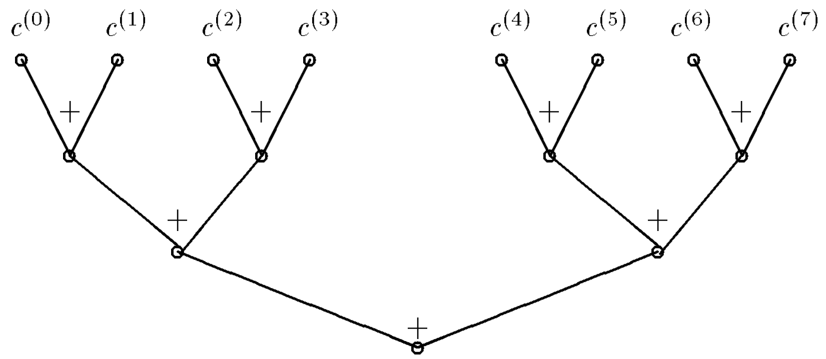
$$[a] \cdot [b] \quad = \quad \sum_{j=0}^{k-1} [a] \cdot [b_j] \cdot 2^j \quad = \quad \sum_{j=0}^{k-1} [a_{k-1} \wedge b_j, \ldots, a_0 \wedge b_j] \cdot 2^j.$$

Diesmal berechnen wir aber für $0 \leq j \leq k-1$ einen Bitstring $c^{(j)} \in \{0,1\}^{2k}$ mit $[c^{(j)}] = [a] \cdot [b_j] \cdot 2^j$. Es ist klar, daß die Berechnung aller dieser $c^{(j)}$ mit einem Schaltkreis der Tiefe

2 und Kosten k^2 durchgeführt werden kann. Es besteht nun die Aufgabe, die Zahlen $[c^{(j)}]$ zu summieren. Man kann diese Summe durch folgende Methode berechnen: Man bestimmt

$$[c^{(0)}]+[c^{(1)}], \quad [c^{(2)}]+[c^{(3)}], \quad [c^{(4)}]+[c^{(5)}] \quad \dots \quad [c^{(k-2)}]+[c^{(k-1)}].$$

Die resultierenden Bitstrings bezeichnen wir wieder mit $c^{(0)},\dots,c^{(k/2)}$. Durch Iteration erhalten wir schließlich die Summe. Das Verfahren läßt sich graphisch mittels eines vollständigen binären Baums veranschaulichen (hier $k=8$):



Dieser Baum hat Tiefe $\log k$. Es gibt aber ein Problem. Für die auftretenden Additionen benötigt man $2k$-Bit Addierer und der beste Addierer, den wir konstruiert haben, hat Tiefe $O(\log k)$. Damit wäre gegenüber dem Addierer des vorigen Abschnitts nichts gewonnen. Also benutzt man einen Trick. Man entwickelt einen Addierer $A_m(4,2)$, der in konstanter Tiefe aus vier Zahlen der Länge m zwei Zahlen der Länge $m+1$ macht und zwar so, daß die Summe der vier Zahlen gleich der Summe der zwei Zahlen ist.

Obiges Verfahren kann man dann modifizieren. Man bestimmt

$$[c^{(0)}]+[c^{(1)}]+[c^{(2)}]+[c^{(3)}], \quad [c^{(4)}]+[c^{(5)}]+[c^{(6)}]+[c^{(7)}] \quad \dots \quad [c^{(k-4)}]+[c^{(k-3)}]+[c^{(k-2)}]+[c^{(k-1)}]$$

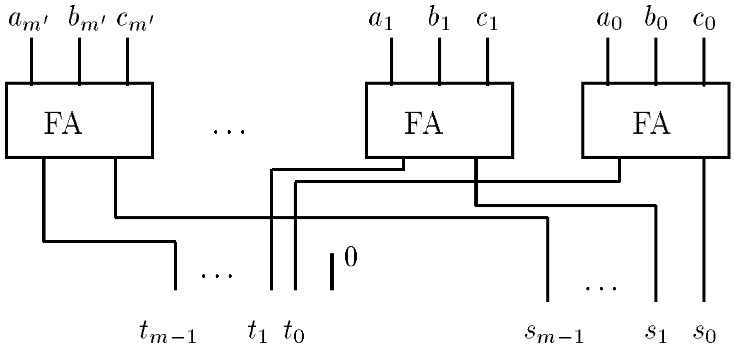
mit Hilfe von $A_{2k}(4,2)$ Addierern. Dabei kann man die "neu entstehenden" Bits vergessen, denn wir wissen, daß unser Ergebnis mit $2k$ Bits darstellbar ist und daß damit auch alle Zwischenergebnisse nur $2k$ Bits benötigen. Die resultierenden Bitstrings bezeichnet man wieder mit $c^{(0)},\dots,c^{(k/2)-1}$. Durch Iteration erhält man schließlich die beiden Bitstrings $c^{(0)},c^{(1)}$. Mittels eines $2k$-Bit Addierers der Tiefe $O(\log k)$ führt man die letzte Addition aus.

Nun zur Konstruktion von $A_m(4,2)$. Zuerst konstruieren wir einen Addierer $A_m(3,2)$, der in konstanter Tiefe aus drei Zahlen der Länge m zwei Zahlen der Länge $m+1$ macht und zwar so, daß die Summe der drei Zahlen gleich der Summe der zwei Zahlen ist. Man benutzt dazu folgende Formel.

4.28. Lemma *Sei $m \in \mathbb{N}$ und $a=(a_{m-1},\dots,a_0), b=(b_{m-1},\dots,b_0), c=(c_{m-1},\dots,c_0) \in \{0,1\}^m$. Seien $t_i, s_i \in \{0,1\}$ mit $[t_i,s_i]=[a_i]+[b_i]+[c_i]$ für $0 \le i \le m-1$. Sei schließlich noch $t=(t_{m-1},\dots,t_0)$ und $s=(s_{m-1},\dots,s_0)$. Dann ist*

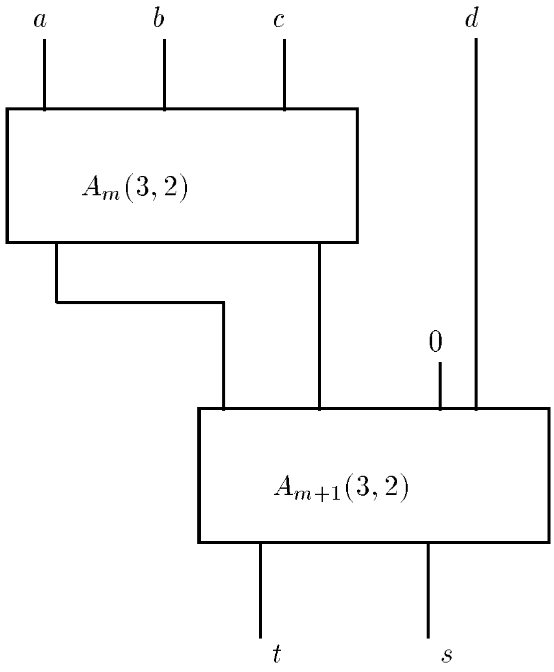
$$[a]+[b]+[c] \quad = \quad 2\cdot[t]+[s].$$

Hieraus gewinnt man unmittelbar folgenden Schaltkreis $A_m(3,2)$. Dabei ist $m'=m-1$:



4.29. Satz *Der Addierer $A_m(3,2)$ hat Tiefe 8 und Kosten $11m$.*

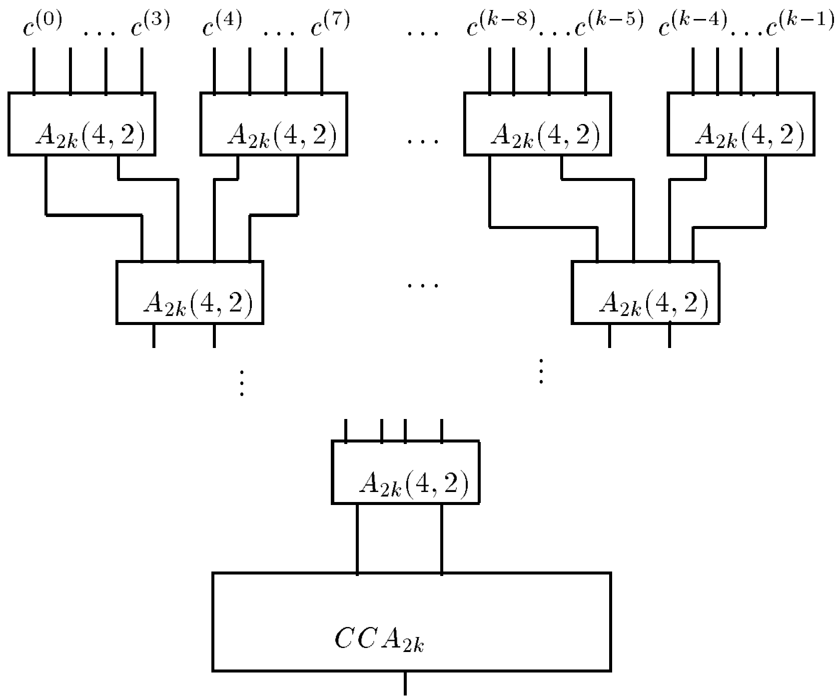
Mit Hilfe eines Addierers $A_m(3,2)$ und eines Addierers $A_{m+1}(3,2)$ kann man dann folgendermaßen den Addierer $A_m(4,2)$ bauen.



Dabei sollte man beachten, daß r und s wirklich $m+1$-Bitstrings sind. Wie wir bei Konstruktion des $A_m(3,2)$ gesehen haben, besitzt höchstens einer der dabei entstehenden Bitstrings (und damit auch nur maximal einer der Eingabestrings von $A_{m+1}(3,2)$) das höchste Bit Eins. Damit kann man es leicht regeln, daß ein evtl. auftretendes Carry nicht zu einer Vergrößerung führt.

4.30. Satz *Der Addierer $A_m(4,2)$ hat Tiefe 15 und Kosten $22m+11$.*

Ein schematisches Schaltbild für den oben bereits beschriebenen Multiplizierer M_k sieht dann so aus:



4.31. Satz *Der Multiplizierer M_k hat Tiefe $O(\log k)$ und Kosten $O(k^2)$.*

Beweis: Wie bereits erwähnt, benötigt die Berechnung der Summanden $c^{(i)}$ für $0 \leq i \leq k-1$ Tiefe 2 und verursacht Kosten k^2. Im i-ten Additionsschritt benötigt man 2^{n-i-1} Addierer vom Typ $A_{2k}(4,2)$. Insgesamt braucht man also

$$\sum_{i=1}^{n-1} 2^{n-i-1} \quad = \quad 2^{n-1} - 1 \quad = \quad k/2 - 1$$

solche Addierer. Am Schluß braucht man noch einen $2k$-Bit Conditional Carry Addierer. Rechnet man alles zusammen, ergibt sich die Behauptung. ∎

Übungen

1. In der Praxis verwendet man statt AND-Gattern häufig NAND-Gatter. Ein solches Gatter berechnet die boolsche Funktion

$$\{0,1\}^2 \to \{0,1\}, \qquad (b_1, b_2) \to \neg(b_1 \wedge b_2).$$

Geben Sie einen möglichst billigen Schaltkreis an, der die 2-stellige boolsche Funktion XOR ausschließlich mit NAND-Gattern berechnet.

2. In dieser Aufgabe soll der Conditional Carry Adder der Vorlesung noch weiter verbessert werden. Betrachten wir dazu die folgende boolsche Funktion:

$$\begin{aligned} f^k_{CCA} : \{0,1\}^{2k} &\longrightarrow \{0,1\}^{2k+2}, \\ (a_{k-1},\ldots,a_0,b_{k-1},\ldots,b_0) &\longmapsto (c_k,c_{k-1},\ldots,c_0,d_k,d_{k-1},\ldots,d_0), \end{aligned}$$

so daß $[c_k,c_{k-1},\ldots,c_0] = [a_{k-1},\ldots,a_0] + [b_{k-1},\ldots,b_0] + 1$ und $[d_k,d_{k-1},\ldots,d_0] = [a_{k-1},\ldots,a_0] + [b_{k-1},\ldots,b_0]$ gilt. Sei im folgenden k eine Zweierpotenz und A_k ein Schaltkreis, der f^k_{CCA} berechnet.

(a) Bestimmen Sie einen Schaltkreis A_1, der f^1_{CCA} berechnet.

(b) Mit Hilfe der Bits c_k und d_k kann offensichtlich festgestellt werden, ob ein Übertrag aufgetreten ist oder nicht. Benutzen Sie diese Beobachtung, um durch ein Divide-and-Conquer Verfahren die Bestimmung von f^k_{CCA} auf die Anwendung von $f^{k/2}_{CCA}$ zurückzuführen. Teilen Sie dazu $(a_{k-1},\ldots,a_0)$ und $(b_{k-1},\ldots,b_0)$ in Hoch-/Tiefwörter auf und wenden Sie $f^{k/2}_{CCA}$ auf diese an. Untersuchen Sie dann, wie die Ergebnisbits $c'_{k/2}, d'_{k/2}$ benutzt werden können, um aus den Teilergebnissen das Gesamtergebnis zu bestimmen.

(c) Versuchen Sie, die Ergebnisse aus (b) zu einer rekursiven Beschreibung des Schaltkreises A_k zu verwenden. Dieser Schaltkreis sollte zwei $(\frac{k}{2}+1)$-MUXe und zwei Bausteine $A_{k/2}$ enthalten.

(d) Bestimmen Sie die Kosten und die Tiefe des Schaltkreises A_k.

3. Wir wollen einen weiteren Addierer mit logarithmischer Tiefe bestimmen, den sogenannten *Carry Lookahead Adder*. Seien dazu $a = (a_{k-1},\ldots,a_0), b = (b_{k-1},\ldots,b_0) \in \{0,1\}^k$ die zwei Bitwörter, die wir addieren wollen. Sei weiterhin $c_i, 0 \le i < k$ der Übertrag an der Stelle i, d.h. der Übertrag, der sich nach Abarbeitung der Bits $a_{i-1},\ldots,a_0,b_{i-1},\ldots,b_0$ ergeben hat.

(a) Bestimmen Sie eine Formel für c_i (in Abhängigkeit von a_i, b_i und c_{i-1}).

(b) Wir wollen nun einen Schaltkreis entwerfen, der in logarithmischer Tiefe für alle $0 \le i < k$ den Wert von c_i bestimmt. Dazu definieren wir Blöcke folgendermaßen: der Block $B_{ij}, i \ge j$ besteht aus den folgenden Teilwörtern von a und b: $(a_i \ldots a_j, b_i \ldots b_j)$. Zu diesen Blöcken definieren wir nun folgende boolsche Werte:

$G_{ij} = 1 \;\Leftrightarrow\;$ der Block B_{ij} (für sich betrachtet) erzeugt ein Carry (generate)

$P_{ij} = 1 \;\Leftrightarrow\;$ der Block B_{ij} propagiert ein rechts einlaufendes Carry durch den ganzen Block hindurch (propagate).

Beispiel: Betrachten wir $a = 1101$ und $b = 0011$, so ist $B_{10} = (01,11)$ und $B_{31} = (110,001)$. Es gilt $G_{10} = 1$, $P_{10} = 1$, $G_{31} = 0$ und $P_{31} = 1$.
Angenommen, wir kennen die Werte P_{ij}, G_{ij} und $P_{k,i+1}, G_{k,i+1}$. Man beachte, daß die entsprechenden Blöcke "benachbart" sind. Bestimmen Sie eine Formel, wie man aus diesen Informationen P_{kj} und G_{kj} berechnen kann.

(c) Überlegen Sie sich, wie man mit der in (b) bestimmten Methode die Werte P_{j0} und G_{j0} für alle $0 \leq j < k$ bestimmen kann. Dies soll in logarithmischer Tiefe geschehen. Beschreiben Sie, wie man damit einen Addierer konstruieren kann.

4. Konstruieren Sie für natürliche Zahlen $a, b \in \{-2^n, \ldots, 2^n - 1\}$ in Komplementdarstellung eine ALU, die in Abhängigkeit zweier Steuerbits f_1, f_0 folgende Ausgabe berechnet:

$[f_1, f_0]$	Ausgabe
0	$a + b$
1	$a - b$
2	$a \cdot b$
3	$-a \cdot b$

Außerdem soll es einen Ausgang U geben, der genau dann Eins ist, wenn eine Bereichsüberschreitung oder -unterschreitung aufgetreten ist. Benutzen Sie dazu Addierer, Multiplizierer, Multiplexer geeigneter Größe und Standardgatter.

5. Überlegen Sie sich, wie man Zahlendarstellungen zur Basis 3 und 4 binär kodiert. Geben Sie dann die Schaltfunktionen für einen Volladdierer für so kodierte Zahlen und die entsprechenden Schaltkreise an.

6. In der Vorlesung wurde ein (k, k)-Multiplizierer mit Tiefe $O((\log k)^2)$ angegeben. Dabei wurde die Eingabe in Hoch-/Tiefwörter aufgetrennt und dann durch Multiplikationen und Additionen nur noch halb so großer Zahlen das Ergebnis berechnet. Dieser Trick kann auch zur Multiplikation langer Zahlen auf einem Rechner angewandt werden (die sogenannte Karatzuba-Multiplikation). Angenommen, unser Rechner arbeitet mit 16-Bit-Zahlen (d.h. alle Integers sind aus der Menge $\{0, \ldots, 2^{16} - 1\}$). Wir wollen nun die beiden Zahlen
$$65538 \cdot 2^{32} + 196612 \qquad \text{und} \qquad 262147 \cdot 2^{32} + 131073$$
mit diesem Trick multiplizieren. Bestimmen Sie dazu die 2^{16}-adische Darstellung dieser beiden Zahlen und wenden Sie dann den Karatzuba-Trick an, um die 2^{16}-adische Darstellung des Ergebnisses zu bestimmen.

7. Versuchen Sie, einen Schaltkreis zu entwerfen, der eine Division mit Rest für zwei k-Bit-Zahlen durchführt und ganzzahligen Anteil und Rest ausgibt. Wenden Sie dazu die Schulmethode zur Division an. Bestimmen Sie dann die Kosten und Tiefe Ihres Schaltkreises.

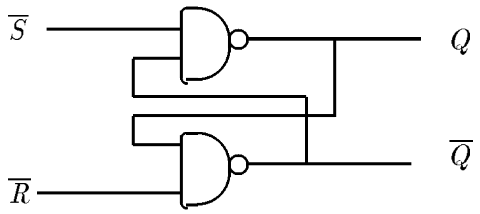
Kapitel 5

Speicherelemente

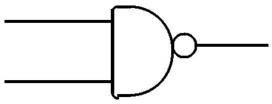
5.1 Flipflops

5.1.1 RS-Flipflops

Wir behandeln nun das Problem, Informationen zu speichern. Hierzu benötigt man statt Schaltkreisen *Schaltwerke*. Dies sind Schaltkreise, in denen man zusätzlich noch Rückkopplung erlaubt. Auf die allgemeine Definition von Schaltwerken verzichten wir, weil nur sehr spezielle benötigt werden. Betrachten wir das folgende Beispiel:



Dieses Schaltwerk nennt man *RS-Flipflop*. Hierin werden zwei NAND-Gatter verwendet, deren Benutzung wir in Zukunft zulassen wollen. Es hat folgende Darstellung



und realisiert die Funktion $f(b_1, b_2) = \neg(b_1 \wedge b_2)$. Die Eingänge sind mit $\overline{S}$ für *Set* und $\overline{R}$ für *Reset* bezeichnet. Der Querstrich bedeutet, daß der Normalzustand dieses Schaltwerkes der ist, daß an den Eingängen Einssignale anliegen. Ist also ein solches Flipflop in ein Gerät eingebaut, so wird beim Einschalten an $\overline{S}$ und $\overline{R}$ die Spannung angelegt, die der logischen Eins entspricht. Die Eingänge sind *active high*. Wäre der Normalzustand ein Nullsignal an beiden Eingängen, so hießen diese R und S und die Eingänge hießen *active low*. Das ist eine Konvention, die es erleichtert, aus der bildlichen Beschreibung zu entnehmen, wie der Baustein eingesetzt wird.

Versuchen wir uns also zu überlegen, welche Ausgangssignale sich ergeben. Angenommen, der Ausgang Q würde gerade Null asugeben. Dann ergibt sich am Ausgang des unteren NAND-Gatters eine Eins, also ist $\overline{Q} = 1$ und dies ist konsistent damit, daß $Q = 0$ ist. Man könnte also denken, so sei notwendig die Ausgabebelegung. Nimmt man aber an, daß $Q = 1$ ist, so ergibt sich $\overline{Q} = 0$ und dies ist ebenfalls konsistent mit $Q = 1$. Dagegen sind die Zustände $Q = \overline{Q} = 1$ und $Q = \overline{Q} = 0$ inkonsistent. Alles was man also sagen kann, ist, daß das Flipflop beim Einschalten in einen der beiden *stabilen Zustände* $Q = 1, \overline{Q} = 0$ oder $Q = 0, \overline{Q} = 1$ übergeht. Die beiden Zustände $Q = 1, \overline{Q} = 1$ und $Q = 0, \overline{Q} = 0$ sind *instabil.* Welcher der stabilen Zustände angenommen wird, kann man nicht vorhersagen.

Interessant ist nun der Umstand, daß man das RS-Flipflop zwingen kann, seinen Zustand zu ändern, also vom augenblicklichen in einen stabilen Zustand überzugehen. Angenommen, das Flipflop ist im Zustand $Q = 1, \overline{Q} = 0$. Senkt man nun das Signal an $\overline{R}$ auf das Null-Signal ab, so ergibt sich $\overline{Q} = 1$. Am oberen NAND-Gatter liegt dann zweimal Eins an, damit also $Q = 0$. Angenommen, das Flipflop ist im Zustand $Q = 0, \overline{Q} = 1$. Senkt man das Signal an $\overline{S}$ auf das Null-Signal ab, so ergibt sich $Q = 1$. Also liegen am unteren NAND-Gatter zwei Eins-Signale an; damit ergibt sich also $\overline{Q} = 0$. Absenken des Signals am Eingang $\overline{S}$ ist ein *Set-Befehl.* Er setzt Q auf 1. Absenken des Signals am Eingang $\overline{R}$ ist ein *Reset-Befehl.* Dadurch wird Q auf Null zurückgesetzt. Damit sind wir jetzt im Prinzip in der Lage, ein Bit zu speichern. Es stellt sich aber noch die Frage, wie lange ein Set- oder Reset-Befehl sein muß. Dies wird im nächsten Abschnitt analysiert.

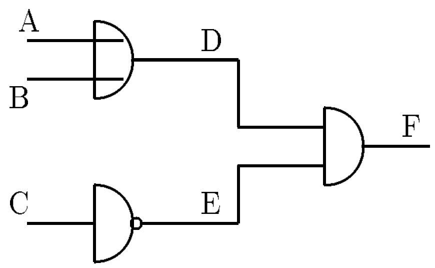
5.1.2 Timing Analyse

In elektronischen Schaltungen werden die logische Eins und die logische Null durch unterschiedliche Spannungen dargestellt. Spannungen oberhalb eines Wertes α entsprechen der logischen Eins. So etwas nennen wir ein *High-Signal.* Wird die Spannung zu hoch, gehen die Bauteile kaputt. Spannungen unterhalb eines Wertes $\beta < \alpha$ entsprechen der logischen Null. Solche Spannungen bezeichnen wir als *Low-Signal.* Liegt am Eingang eines NOT-Gatters ein High-Signal an, so liegt an seinem Ausgang ein Low-Signal an. Ändert man das Eingangssignal in ein Low-Signal, so ändert sich am Ausgang erst mal gar nichts. Die Signaländerung braucht eine gewisse Zeit, um zum Ausgang des Gatters "durchzukommen". Diese Zeit heißt *propagation time high low.* Entsprechend ist die *propagation time low high* definiert. Auf deutsch heißen diese Zeiten *Verzögerungszeiten.*

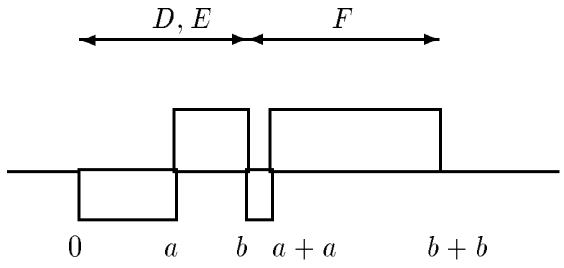
In den technischen Beschreibungen der Gatter werden diese mit ihren Verzögerungszeiten in Nano-Sekunden ($= 10^{-9}$ Sekunden) angegeben. Genauer gesagt: Es wird angegeben, zu welcher Klasse ein Gatter gehört und die Zugehörigkeit zu dieser Klasse bedeutet, daß die Verzögerungszeiten in gewissen Bereichen liegen. Außerdem ist für die Gatter einer Klasse bekannt, welche zulässige Betriebstemperatur sie haben und in wieviel Gattereingänge man den Gatterausgang führen darf. Letzteren Wert nennt man *Fanout.*

Um abschätzen zu können, zu welchem Zeitpunkt ein Signal am Ausgang eines Schaltkreises gültig ist, müssen wir eine sogenannte Timing-Analyse durchführen. Wir veranschaulichen das an folgendem Beispiel:

5.1. Beispiel Betrachten wir den folgenden Schaltkreis:



Wir nehmen im folgenden an, daß das Durchlaufen der Leitungen keine Zeit benötigt. Angenommen, zum Zeitpunkt 0 ändert man an wenigstens einem der Eingänge A, B oder C das Signal. Dann ändert sich bei D oder E frühestens nach a und spätestens nach b Nanosekunden das Signal. Dabei seien a und b untere bzw. obere Grenzen für die Verzögerungszeit aller einfacher Gatter. Das wird so veranschaulicht:



Dabei gibt ein Rechteck unterhalb der Linie an, daß zu diesem Zeitpunkt das Signal an dem entsprechenden Ausgang noch nicht korrekt vorliegt; ein Rechteck oberhalb der Linie zeigt an, daß es in diesem Zeitraum nicht gewährleistet werden kann, daß das Signal korrekt vorliegen kann. Dies kann wegen der Zeitspanne für die Verzögerungszeit nicht eindeutig bestimmt werden.

Die Verwendung der genauen Verzögerungszeiten für spezielle Bausteingruppen ist einerseits zu speziell, weil solche Parameter schnell überholt sein können und andererseits zu kompliziert. Wir wollen daher in Zukunft annehmen, daß die minimale Verzögerungszeit für ein Gatter a Nanosekunden und die maximale Verzögerungszeit b Nanosekunden ist.

5.1.3 Timing Analyse für das RS-Flipflop

Wie in Beispiel 5.1 kann man auch das RS-Flipflop analysieren.

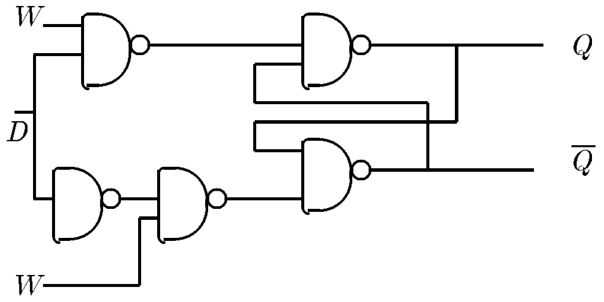
5.2. Definition *Der Zustand eines RS-Flipflops, an das ein Set- oder Reset-Signal angelegt ist, heißt* stabil, *wenn sich die Werte für* Q *und* $\overline{Q}$ *nach Rücknahme des Signals nicht ändert.*

5.3. Satz *Gibt man auf ein RS-Flipflop ein Set- oder Reset-Signal von wenigstens* $3b - a$ *Nanosekunden Dauer, so wird sein Zustand stabil.*

Beweis: Betrachten wir den Fall des Set-Signals. Dies braucht höchstens b Nanosekunden, um sich auf Q auszuwirken. Unabhängig vom Wert von $\overline{Q}$ ist jetzt $Q = 1$. Nach weiteren b Nanosekunden ist $\overline{Q} = 0$. Würde man aber jetzt das Set-Signal zurücknehmen, so könnte es passieren, daß sich wegen der Verzögerung des oberen NAND-Gatters noch nicht der neue Wert $\overline{Q} = 0$, sondern der alte Wert $\overline{Q} = 1$ auf Q auswirkt. Zusammen mit $\overline{S} = 1$ würde das bedeuten, daß sich wieder $Q = 0$ und $\overline{Q} = 1$ ergäbe. Damit wäre also nichts erreicht. Man muß also dafür sorgen, daß die Auswirkung der Rücknahme des Set-Signals frühestens nach $3b$ Nanosekunden eintritt. Also darf man das Set-Signal frühestens nach $3b - a$ Nanosekunden zurücknehmen. Die Analyse für das Reset-Signal geht genauso. ■

5.1.4 Das WD-Flipflop

Angenommen, eine Rechnung hat als Ergebnis Eins ergeben und dieses Ergebnis möchte man speichern. Verwendet man hierzu ein RS-Flipflop, so muß man jetzt ein Set-Signal geben. Hat die Rechnung dagegen Null ergeben, so muß man ein Reset-Signal geben. Man benötigt also noch ein Zwischenstück, das in Abhängigkeit von dem angelegten Datum ein Set- oder Reset-Signal erzeugt. Dieses Zwischenstück ist in folgendem WD-Flipflop direkt vorgeschaltet.



Es funktioniert so: An den Eingang D legt man ein Datum an, also ein Eins-Signal oder ein Null-Signal. Dann gibt man auf den Eingang W einen Schreibpuls, d.h. ein Eins-Signal. Man sieht leicht, daß sich an den Eingängen des nachgeschalteten RS-Flipflops für $D = 1$ ein Set- und für $D = 0$ ein Reset-Signal ergibt.

5.4. Definition *Der Zustand eines WD-Flipflops heißt* stabil, *wenn sich bei Schreibpuls Null und Änderung des Datums die Werte von Q und $\overline{Q}$ nicht ändern.*

Die Mindestzeiten, die das Datum gültig und der Schreibpuls angelegt sein muß, damit sich das WD-Flipflop stabilisiert, werden in folgendem Satz angegeben.

5.5. Satz *Wird auf ein WD-Flipflop zuerst ein Datum und nach b Nanosekunden für $4b - 2a$ Nanosekunden ein Schreibpuls gegeben und bleibt das Datum insgesamt wenigstens $6b - 3a$ Nanosekunden unverändert, so ist der Zustand des WD-Flipflops stabil.*

Beweis: Wir wissen schon aus Satz 5.3, daß Set- oder Reset-Signale der Länge wenigstens $3b - a$ das RS-Flipflop stabilisieren. Wir müssen also dafür sorgen, daß diese Signale wenigstens so lang sind. Nach spätestens b Nanosekunden wirkt sich das Datum D auf die Eingänge der beiden NAND-Gatter aus. Also fängt man nach b Nanosekunden an, den Schreibpuls zu geben. Datum und Schreibpuls wirken sich nach spätestens b weiteren Nanosekunden auf die Ausgänge der vorderen NAND-Gatter aus. Dann steht also das Set oder Reset-Signal. Hält man den Schreibpuls weitere x Nanosekunden stabil, so bleibt das Set- oder Reset-Signal für $x + a$ Nanosekunden gültig. Dies bedeutet, daß $x + a \geq 3b - a$ und daher $x \geq 3b - 2a$ sein muß. Man kann also nach $4b - 2a$ Nanosekunden den Schreibpuls zurücknehmen. Man muß aber noch $b - a$ Nanosekunden warten, bevor man das Datum verändert. Der Schreibpuls $W = 1$ ist nämlich noch für höchstens b Nanosekunden am Ausgang der vorderen NAND-Gatter wirksam. ■

5.2 Dekodierer

Wir haben in den vorangegangenen Abschnitten gelernt, wie man einzelne Bits in D-Flipflops speichert. Angenommen, man hat eine große Zahl von solchen Speicherzellen. Ein Datum soll in eine solche Zelle abgelegt werden. Wir nehmen weiter an, daß die Speicherzellen durchnumeriert sind. Eine solche Nummer nennt man *Adresse*. Sei weiter angenommen, daß der Rechner die Adresse der Speicherzelle kennt, in der das Datum abgelegt werden soll. Dann kann man so vorgehen, um das Datum zu speichern: man legt an die Dateneingänge aller Speicherzellen das aktuelle Datum an. Man wartet, bis sich das Datum stabilisiert hat. Man gibt auf die richtige Speicherzelle einen Schreibpuls und man hält das Datum so lange stabil, bis sich das Flipflop stabilisiert hat. Es bleibt zu klären, wie der Rechner die richtige Speicherzelle findet. Die Adresse liegt ja zunächst als Binärzahl vor. Man benötigt jetzt einen *Dekodierer*, der k Eingänge und 2^k Ausgänge hat. Die Ausgänge sind mit den Schreibpulseingängen der Speicherzellen verbunden. Liegt am Eingang eine k-Bit Adresse a an, so entsteht genau am Ausgang mit der Nummer $[a]$ ein Signal.

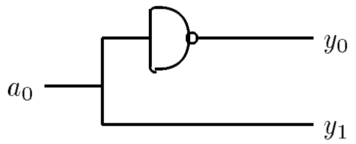
Wir fassen diese Beschreibung in der nächsten Definition zusammen.

5.6. Definition *Sei $k \in \mathbb{N}$. Ein k-Bit Dekodierer ist ein Schaltkreis, der die Funktion*

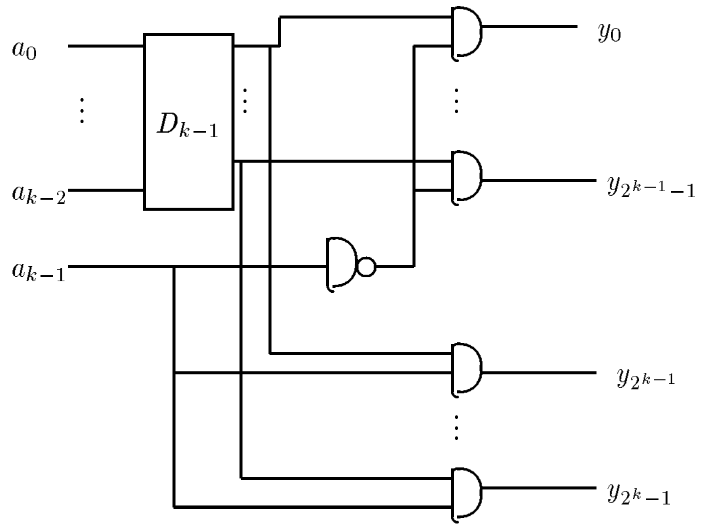
$$\{0,1\}^k \longrightarrow \{0,1\}^{2^k}, \quad a \longmapsto b = (b_0, \ldots, b_{2^k-1}),$$

wobei für $0 \leq i \leq 2^k - 1$ das Bit b_i genau dann den Wert Eins hat, wenn $i = [a]$ ist, realisiert.

Wir konstruieren induktiv einen k-Bit Dekodierer. Für $k = 1$ kann man folgenden Schaltkreis D_1 verwenden:



Dieser Dekodierer hat die Kosten 1 und die Tiefe 2. Angenommen, $k \in \mathbb{N}_{\geq 2}$ und man hat einen Dekodierer D_{k-1} gefunden. Dann konstruieren wir daraus auf folgende Art und Weise einen Dekodierer D_k:



Dabei ist die Idee die folgende: Wegen

$$[a_{k-1}, \ldots, a_0] \quad = \quad a_{k-1} \cdot 2^{k-1} + [a_{k-2}, \ldots, a_0]$$

kann man zuerst den Bitstring $(a_{k-2}, \ldots, a_0)$ mit Hilfe des Dekodierers D_{k-1} dekodieren. Dann entscheidet das oberste Bit a_{k-1}, ob dieses Ergebnis geshiftet werden muß oder nicht. Falls $a_{k-1} = 0$ ist, kann man das Ergebnis direkt verwenden; für $a_{k-1} = 1$ muß das Resultat noch um 2^{k-1} geshiftet werden.

Wir wollen uns nun noch mit der Tiefe und den Kosten sowie der Verzögerungszeit eines solchen Dekodierers beschäftigen.

5.7. Satz *Der Dekodierer D_k hat Kosten $2^{k+1} + k - 4$ und Tiefe $k + 1$.*

Beweis: Der Satz wird durch Induktion über k bewiesen. Seien hierzu T_k die Tiefe von D_k und L_k die Kosten von D_k. Offensichtlich ist $T_1 = 2$ und $L_k = 1$. Sei nach Induktionsannahme dann $T_{k-1} = k$ und $L_{k-1} = 2^k + k - 5$. Offensichtlich ist

$$T_k \quad = \quad T_{k-1} + 1 \quad = \quad k + 1$$

und

$$L_k \quad = \quad L_{k-1} + 2^k + 1 \quad = \quad 2^{k+1} + k - 4.$$

■

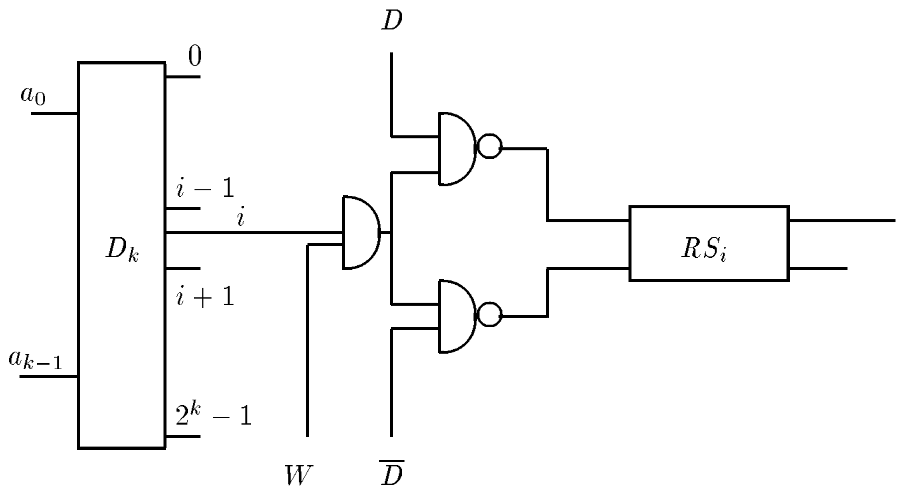
5.8. Korollar *Spätestens kb Nanosekunden nach Anlegen der Adresse ergibt sich bei dem Dekodierer D_k am richtigen Ausgang ein Signal.*

5.3 SRAM

5.3.1 Aufbau einer SRAM

Die bis jetzt besprochenen Bauteile kann man verwenden, um ein SRAM$_k$ (statisches Random Access Memory) zu bauen. Ein solcher Baustein kann 2^k Bits speichern. Er verwendet hierzu 2^k Flipflops. Der Baustein besitzt zwei Funktionsweisen, Schreiben und Lesen. Beim Schreiben wird eine Adresse i und ein Datum D angelegt und dann genügend lange ein Schreibpuls gegeben. Dadurch wird genau in die i-te Speicherzelle das Datum D geschrieben. Beim Lesen wird nur die Adresse i eingegeben. Am Ausgang erscheint dann das Datum, das in der i-ten Zelle gespeichert wurde. Solange das SRAM mit Strom versorgt wird, bleiben die Daten gespeichert. Im Gegensatz dazu gibt es auch billigere DRAMs (Dynamic Random Access Memory). In denen werden die Daten in Kondensatoren gespeichert. Die Speicherung muß in regelmäßigen Abständen aufgefrischt werden.

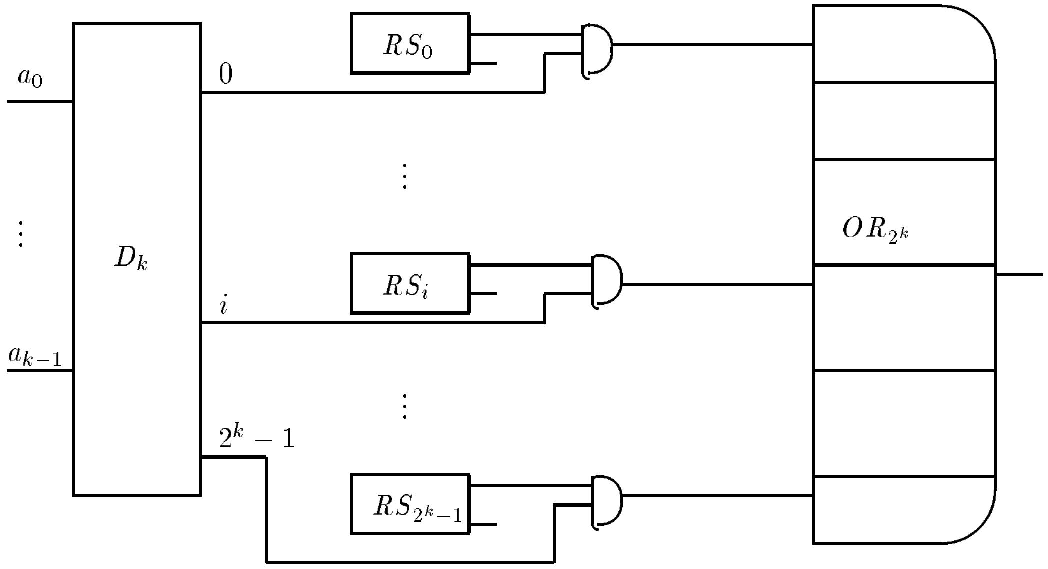
Jetzt soll das SRAM etwas genauer besprochen werden. Sei $a = (a_{k-1}, \ldots, a_0) \in \{0,1\}^k$ und sei $i = [a]$ die Adresse, in die man das Datum $D \in \{0,1\}$ schreiben will. Diese Adresse wird zuerst in einem Dekodierer D_k dekodiert. Es entsteht dann genau am i-ten der 2^k Ausgänge von D_k ein Signal. Das SRAM braucht zwei zusätzliche Eingänge, einen für das Datum D und einen für den Schreibimpuls W. Aus Adressignal y_i, Schreibimpuls W und Datum D macht man in folgender Weise das Set- und Reset-Signal für das RS-Flipflop. Dabei sind in der folgenden Zeichnung nur das i-te Flipflop und die entsprechenden Gatter für dieses Flipflop eingezeichnet. Dieselbe Steuerschaltung existiert auch für alle anderen Flipflops.



Der Schreibzyklus einer SRAM sieht dann folgendermaßen aus:

1. Adresse und Daten stabil machen
2. Schreibimpuls geben
3. Adresse und Daten noch stabil halten
4. Ergebnis: Datum ist in der durch die Adresse gegebenen Zelle gespeichert.

Nun besprechen wir noch den Lesezyklus. Dabei muß wieder die Adresse des gewünschten Datums angelegt werden und dann die Information aus dem entsprechenden Flipflop gelesen werden. Dazu kann man die folgende Schaltung benutzen. In der Zeichnung werden nur die Gatter angegeben, die für das Lesen benötigt werden; die Gatter zum Schreiben des Flipflops werden aus Übersichtsgründen weggelassen.

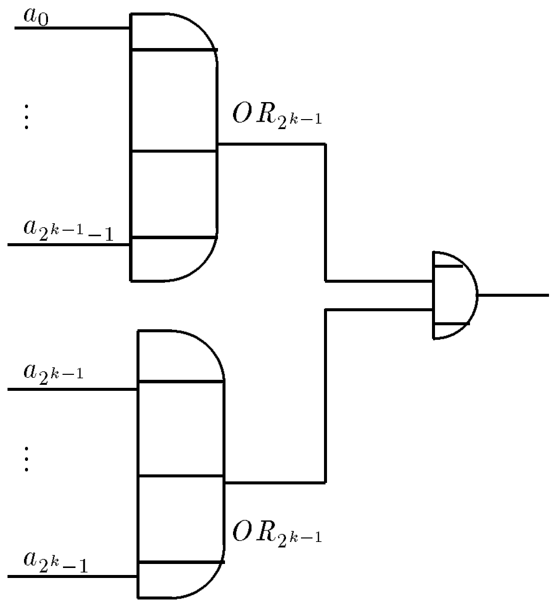


Dabei wurde das Gatter OR_{2^k} verwendet, was ein OR-Gatter mit 2^k Eingängen darstellt. Wir werden gleich darauf eingehen, wie wir solch ein Gatter konstruieren können.

Der Lesezyklus einer SRAM sieht dann so aus:

1. Adresse stabil machen
2. Ergebnis: Datum erscheint am i-ten Ausgang

Wir wollen nun noch zeigen, wie man Gatter OR_{2^k} für $k \in \mathbb{N}$ baut und welche Kosten und Tiefe diese Gatter besitzen. Die Konstruktion geschieht wiederum induktiv. Offensichtlich ist das Gatter OR_{2^1} ein ganz normales OR-Gatter. Nehmen wir an, wir wissen schon, wie man ein $\mathrm{OR}_{2^{k-1}}$ Gatter konstruiert. Dann erhalten wir daraus durch folgende einfache Konstruktion ein OR_{2^k} Gatter:



Damit gilt:

5.9. Satz *Das OR_{2^k} Gatter besitzt Tiefe $k+1$ und Kosten $2^k - 1$.*

Beweis: Induktion. ■

5.3.2 Timing-Analyse einer SRAM

Damit ergeben sich die folgenden Timing-Analysen für einen Lese- und einen Schreibzugriff:

5.10. Satz *Sind an einer SRAM mit 2^k Speicherplätzen Datum und Adresse für wenigstens kb Nanosekunden gültig, gibt man dann für genau $5b - 3a$ Nanosekunden einen Schreibpuls und hält danach Datum und Adresse noch für wenigstens $3b - a$ Nanosekunden stabil, so ist das durch die Adresse spezifizierte Flipflop stabil und enthält das eingegebene Datum.*

Beweis: Angenommen, man legt eine Adresse i und ein Datum an. Nach kb Nanosekunden ist die Adresse am Ausgang des Dekodierers stabil. Zu diesem Zeitpunkt sind auch Datum und inverses Datum am i-ten Flipflop gültig. Jetzt wird der Schreibimpuls gegeben. Spätestens nach weiteren b Nanosekunden erzeugt dieser zusammen mit der Adresse am i-ten Flipflop einen Schreibpuls. Dieser soll wenigstens $4b - 2a$ Nanosekunden gültig sein. Also gibt man einen Schreibimpuls von $5b - 3a$ Nanosekunden. Nach spätestens $6b - 3a$ Nanosekunden ist der Schreibpuls am i-ten Flipflop wieder Null. Das Datum muß jetzt noch $2b$ Nanosekunden gültig sein. Hierzu genügt es, das Datum noch $2b - a$ Nanosekunden stabil zu halten. ■

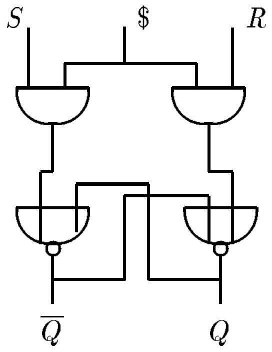
5.11. Satz *Ist an einer SRAM mit 2^k Speicherplätzen eine Adresse für wenigstens $(2k+1)b$ Nanosekunden gültig, so erscheint am Ausgang das Datum, das in dem durch die Adresse spezifizierten Flipflop steht.*

Beweis: Angenommen, man legt eine Adresse i an. Nach spätestens kb Nanosekunden ist die Adresse am Ausgang des Dekodierers stabil. Nach spätestens $(k+1)b$ Nanosekunden erscheint das Datum des i-ten Flipflops am Ausgang des i-ten AND-Gatters. Nach weiteren $(k-1)b$ Nanosekunden erscheint dieses Datum am Ausgang der SRAM. ■

Übungen

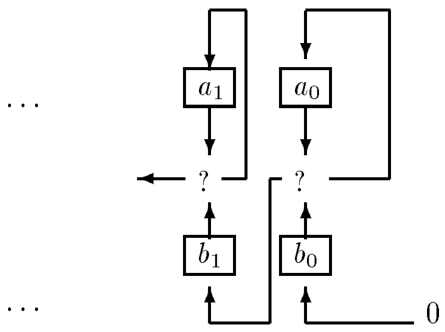
1. Bauen Sie aus geeigneten Flipflops ein 32-Bit Register zusammen (ein Register ist ein Speicherelement für mehrere Bits).

2. Betrachten Sie das folgende Schaltwerk:



   Dabei sei \$ der Eingang für den Takt. Untersuchen Sie in Abhängigkeit von der Belegung des Takteingangs \$, wie sich die Ausgänge Q und $\overline{Q}$ in Abhängigkeit von S und R verhalten. Wie kann man dieses Bauelement als Speicher benutzen?

3. Angenommen, sie haben zweimal n Flipflops, in denen sich die Binärdarstellungen von Zahlen $a, b \in \{0, \dots, 2^n - 1\}$ befinden. Bauen Sie daraus ein Schaltwerk, so daß nach einer gewissen Anzahl von Runden in den ersten n Flipflops die Summe der beiden Zahlen a und b steht. Dies sollte etwa folgendermaßen aussehen:



   (a) Bestimmen Sie, welcher Baustein für das Zeichen ? in obiger Zeichnung eingesetzt werden muß.

(b) Zeichnen Sie ein solches Addierwerk für zwei 8-Bit Zahlen.

(c) Geben Sie die Inhalte aller Flipflops Ihres Addierwerks bei Addition der Bitstrings $a = (10011101)_2$ und $b = (01110011)_2$ an.

(d) Bestimmen Sie die maximale Anzahl von "Runden", die nötig sind, bis das Ergebnis fertig berechnet ist.

(e) Bestimmen Sie die Kosten eines solchen n-Bit Addierwerks (dabei sollen Flipflops die Kosten 1 besitzen).

Bemerkung: Ein solches Addierwerk nennt man auch **von-Neumann Addierer**.

4. In der Vorlesung wurde der Dekodierer D_k vorgestellt. Geben Sie eine andere Konstruktion eines solchen Dekodierers an. Verwenden Sie dabei zur Konstruktion von D_k einen Dekodierer D_{k-1} und einen MUX_{2^k}. Bestimmen Sie die Tiefe und die Kosten Ihres Dekodierers.

5. Versuchen Sie, einen von-Neumann Subtrahierer zu entwerfen. Dieser soll ähnlich wie der von-Neumann Addierer aufgebaut sein. Dabei soll zu Beginn in den beiden benutzten Registern die Binärdarstellung (nicht das Zweierkomplement !!) zweier ganzer Zahlen a und b stehen. Nach einer bestimmten Anzahl von Runden soll sich – falls $a \geq b$ ist – in einem der Register die Binärdarstellung von $a - b$ befinden.

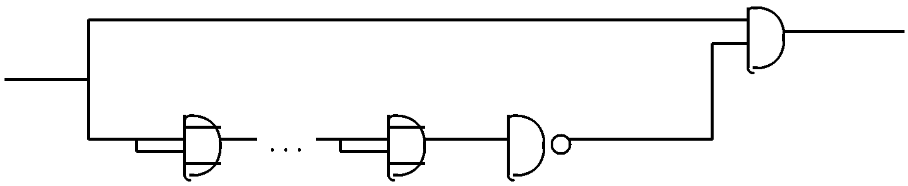
   Überlegen Sie sich, wie ein solches Schaltwerk konstruiert werden kann und zeichnen Sie einen von-Neumann Subtrahierer für 4 Bits.

Kapitel 6

Register, Zähler, Treiber

6.1 D-Flipflops

Manchmal möchte man sich nicht darum kümmern, wie lange man den Schreibpuls geben muß, bevor das Flipflop stabil bleibt. Mittels einer *Flanke*, d.h. einem Umspringen des Clock-Signals von Null nach Eins, soll das Flipflop selbst einen Schreibpuls der richtigen Länge erzeugen. Dies erreicht man, indem man das Write-Signal durch einen *Verzögerungsschaltkreis* schickt.

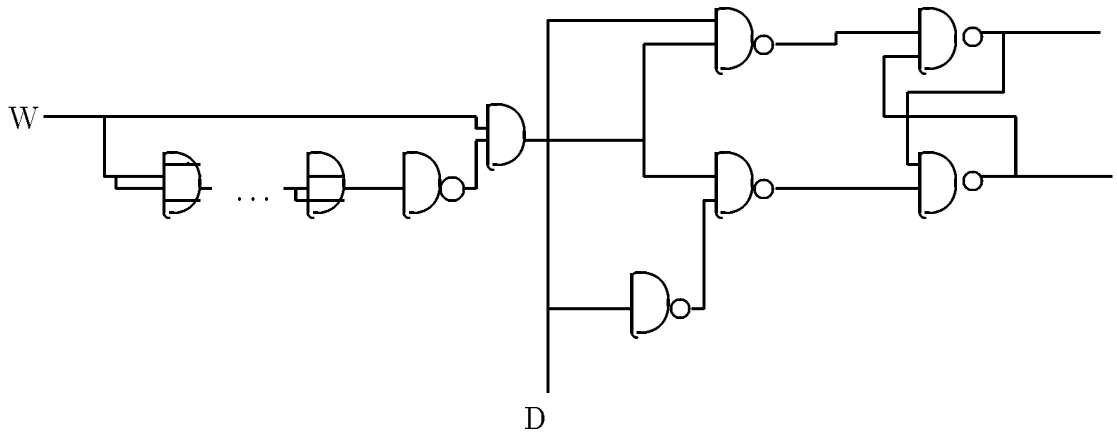


Bei Verwendung von $k-1$ OR-Gattern bezeichnen wir diesen Schaltkreis einen *CKD-Schaltkreis* der Stufe k. CKD steht dabei für Clock Delay.

6.1. Satz *Gibt man auf den Eingang eines CKD-Schaltkreises der Stufe k ein Eins-Signal der Länge mindestens $(k+1)b-a$ Nanosekunden, so entsteht am Ausgang nach spätestens b Nanosekunden ein Eins-Signal, das dann noch wenigstens $(k+1)a-b$ Nanosekunden stabil ist. Nach spätestens $(k+1)b-a$ Nanosekunden ist das Ausgangssignal wieder Null.*

Beweis: Die Flanke erscheint sofort an dem Eingang des letzten AND-Gatters. Am unteren Eingang dieses Gatters ist dort zunächst eine Eins. Also wird nach spätestens b Nanosekunden die Ausgabe auf Eins gehen. Nach frühestens ka und spätestens kb Nanosekunden ist am unteren Eingang des letzten AND-Gatters eine Null, denn dann hat das Eingabesignal die ganze untere Kette von OR-Gattern und das NOT-Gatter durchlaufen. Nach frühestens $(k+1)a$ und spätestens $(k+1)b$ Nanosekunden ist damit das Ausgabesignal wieder Null. Damit ist am Ausgang das Eins-Signal mindestens $(k+1)a-b$ und längstens $(k+1)b-a$ Nanosekunden lang gültig. ■

Um einen hinreichend langen Schreibpuls zu erhalten, genügt es also, vor den Write-Eingang eines WD-Flipflops einen CKD-Schalkreis der Stufe $\lceil (4b - 2a)/a \rceil$ zu schalten. Ein solches Flipflop nennt man *D-Flipflop*:



6.2. Definition *Der Zustand eines D-Flipflops heißt* stabil, *wenn sich bei Schreibpuls Null und Änderung des Datums die Werte von Q und $\overline{Q}$ nicht ändern.*

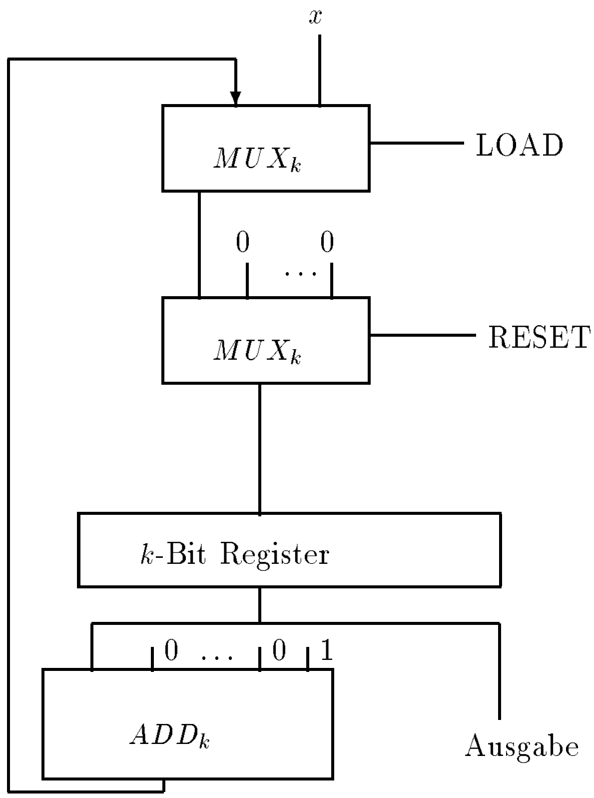
6.3. Satz *Wird auf ein D-Flipflop zuerst ein Datum und nach $b-a$ Nanosekunden ein Schreibpuls von wenigstens $2b + (k-2)a$ Nanosekunden mit $k = \lceil (4b - 2a)/a \rceil$ gegeben und bleibt das Datum insgesamt wenigstens $(k+3)b - a$ Nanosekunden unverändert, so ist der Zustand des Flipflops stabil.*

Beweis: Nach b Nanosekunden hat sich das Datum und sein Inverses stabilisiert. Frühestens nach b und spätestens nach $2b - a$ Nanosekunden beginnt der Schreibpuls. Dieser bleibt dann wenigstens $ka \geq 4b - 2a$ Nanosekunden stabil, sofern der Schreibimpuls $2b + (k-2)a$ Nanosekunden stabil bleibt. Nach Satz 5.5 reicht das, um das RS-Flipflop zu stabilisieren. Der Schreibpuls geht spätestens nach $(k+1)b$ Nanosekunden zurück auf Null. Dann muß das Datum noch $2b$ Nanosekunden gültig sein. Das Datum muß also insgesamt $(k+3)b - a$ Nanosekunden stabil sein. ■

Ein *k-Bit Register* ist eine Folge von k Flipflops, die simultan von einer Clock gesteuert werden. Register werden zum Beispiel gebraucht, um Zwischenergebnisse abzulegen. Sie werden auch verwendet, um Zähler zu konstruieren.

6.2 Zähler

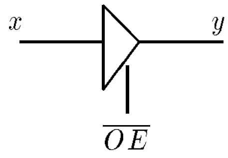
Ein k-Bit Zähler sieht folgendermaßen aus:



Er funktioniert folgendermaßen: Am Eingang liegt ein Bitstring $x \in \{0,1\}^k$ an. Wird ein LOAD-, aber kein RESET-Signal gegeben, so wird dieser String im nächsten Clock-Zyklus in das Register geladen. Wird dagegen ein RESET-Signal gegeben, so wird der String, der nur aus Nullen besteht, in das Register geladen. Wird weder ein LOAD- noch ein RESET-Signal gegegeben, so wird in jedem Clock-Zyklus der Registerinhalt um eins hochgezählt.

6.3 Treiber

Treiber haben die Aufgabe, Signale zu verstärken. Treiber haben drei mögliche Ausgangszustände: Null, Eins und neutral.



Schaltet man einen Treiber hinter den Ausgang einer Schaltung, so ist sein Ausgang zunächst neutral. Erst wenn das Steuersignal $\overline{OE}$ (= Output Enable) abgesenkt wird, entsteht je nach Ausgang der Schaltung am Treiberausgang eine Null oder eine Eins. Die Zeitspanne, nach der dies frühestens/spätestens der Fall ist, heißt *minimal enable time/maximal enable time*. Setzt man das Steuersignal wieder auf Eins, so wird der Ausgang des Treibers wieder neutral. Die Zeitspanne, nach der dies frühestens/spätestens der Fall ist, heißt *minimal disable time/maximal disable time*.

Kapitel 7

RESA

In diesem Kapitel wird besprochen, wie man aus den dargestellten Bauelementen einen einfachen Rechner, die RESA (Rechenanlage Saarbrücken) baut.

7.1 Funktionsweise der RESA

Die RESA hat drei Elemente, den *Speicher* (S), die *Ein/Ausgabe Einheit* (I/O) und die CPU (Central Processing Unit). Der Speicher ist noch unterteilt in einen *Rechenspeicher* (RS) und einen *Programmspeicher* (PS). CPU, Speicher und I/O sind über drei Leitungen miteinander verbunden:

- einen 24-Bit *Adressbus*, über den Adressinformationen ausgetauscht werden
- einen 32-Bit *Datenbus*, über den Daten versendet werden
- einen 8-Bit *Kontrollbus*, der Steuerinformationen wie z.B. Lese und Schreibbefehle befördert.

Am liebsten würden wir nun eine CPU konstruieren, die die Funktionen der P-Maschine hat. Das ginge im Prinzip zwar, wäre aber ziemlich kompliziert, denn die P-Maschine besitzt Bauelemente (Keller) und Funktionen (Division), die wir mit den uns nun eingeführten Bauteilen nicht unmittelbar realisieren können. Außerdem wurde die RESA unabhängig von der P-Maschine entwickelt. Um den Übersetzer zu benutzen, der im ersten Kapitel beschrieben wurde, muß man daher die P-Maschine auf der RESA simulieren.

Die nun zu beschreibende CPU arbeitet folgendermaßen. Sie lädt den Befehl, dessen Adresse vom 24-Bit *Befehlszähler* (BZ) angegeben wird, aus dem Programmspeicher in das 32-Bit *Befehlsregister* (BR). Diesen Befehl führt sie aus. Das Resultat steht im *Akkumulator* (ACC), einem 24-Bit Register. Zur Ausführung des Befehls braucht sie zwei 24-Bit *Indexregister* (IR1, IR2) und eine 24-Bit ALU. Die Indexregister dienen dabei der *indirekten Adressierung.*

7.2 Maschinensprache der RESA

Im Befehlsregister können verschiedene Sorten von Befehlen stehen. Da sind zunächst die *Sprungbefehle* für den Befehlszähler. Ein solcher Befehl heißt entweder JUMPFORW(ω, x) oder JUMPBACKW(ω,x), wobei $\omega \in \{=, <, \leq, >, \geq, \neq\}$ und $x \in \{0, 1, \ldots, 2^{24} - 1\}$ ist. Zum Beispiel bedeutet der Befehl JUMPFORW(=,x), daß die Adresse im Befehlszähler um x erhöht wird, falls der Inhalt des Akkumulators 0 ist. Andernfalls wird die Adresse im Befehlszähler um 1 erhöht. Wir schreiben hierfür auch kurz $BZ := BZ + x$, falls $ACC = 0$. In folgender Tabelle listen wir alle Sprungbefehle auf:

Befehl	Wirkung
JUMPFORW(ω,x)	$BZ := BZ + x$, falls ACC ω 0; $BZ := BZ + 1$ andernfalls
JUMPBACKW(ω,x)	$BZ := BZ - x$, falls ACC ω 0; $BZ := BZ + 1$ andernfalls
JUMPFORW(x)	$BZ := BZ + x$
JUMPBACKW(x)	$BZ := BZ - x$

Desweiteren gibt es Speicher- und Ladebefehle. Für $I \in \{IR1, IR2\}$ und $A, B \in \{IR1, IR2, ACC, BZ\}$ sind dies folgende:

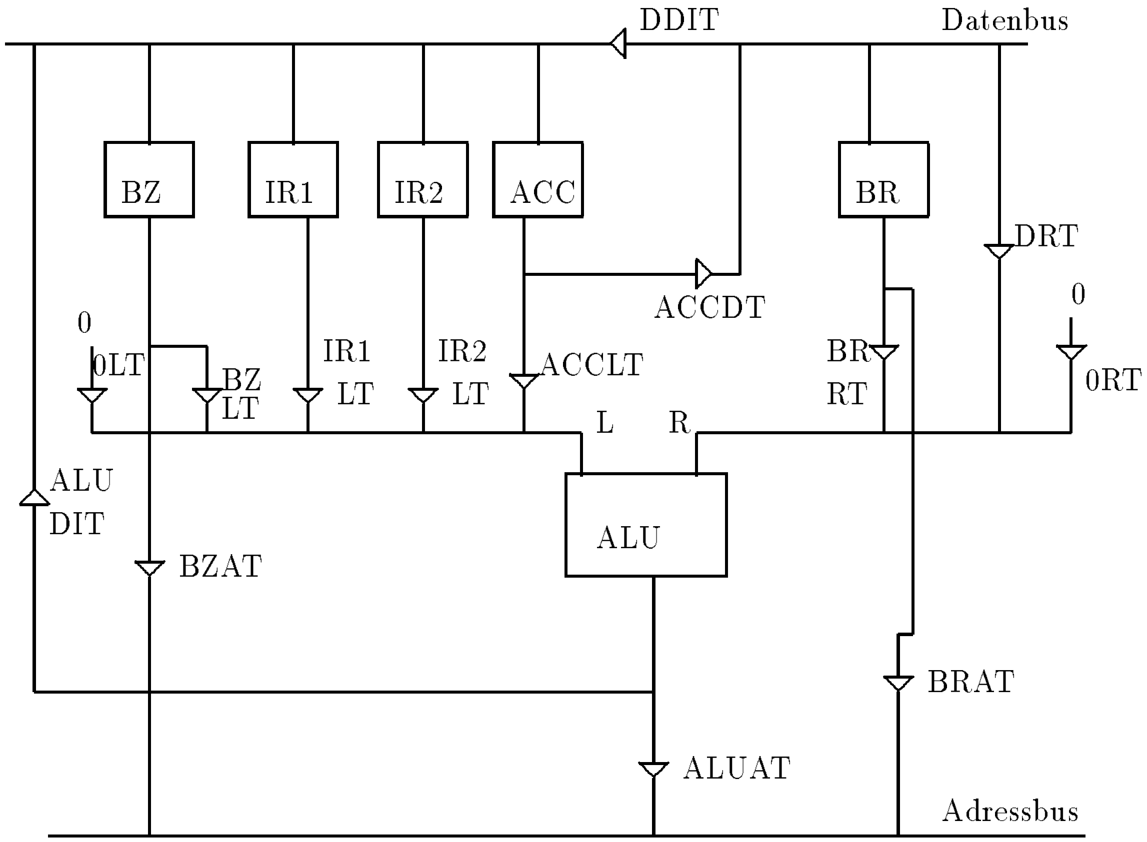
Befehl	Wirkung
LOAD x	$ACC := RS(x)$; $BZ := BZ + 1$
STORE x	$RS(x) := ACC$; $BZ := BZ + 1$
STORE I	$I := ACC$; $BZ := BZ + 1$
LOAD I	$ACC := I$; $BZ := BZ + 1$
STORE I x	$RS(I + x) := ACC$; $BZ := BZ + 1$
LOAD I x	$ACC := RS(I + x)$; $BZ := BZ + 1$
LOADNUM x	$ACC := x$; $BZ := BZ + 1$
STOREINT A B	$B := A$; $BZ := BZ + 1$

Folgende Rechenbefehle gehören auch zum Befehlsvorrat der RESA (I gewählt wie oben):

Befehl	Wirkung
ADD x	$ACC := ACC + RS(x)$; $BZ := BZ + 1$
SUB x	$ACC := ACC - RS(x)$; $BZ := BZ + 1$
ADD I x	$I := I + RS(x)$; $BZ := BZ + 1$
SUB I x	$I := I - RS(x)$; $BZ := BZ + 1$
ADDC x	$ACC := ACC + x$; $BZ := BZ + 1$
SUBC x	$ACC := ACC - x$; $BZ := BZ + 1$
ADDC I x	$I := I + x$; $BZ := BZ + 1$
SUBC I x	$I := I - x$; $BZ := BZ + 1$

7.3 Datenpfade

Um diese Befehle ausführen zu können, werden die Bauelemente der CPU in folgender Weise verbunden.



Wir zeigen, daß die Datenpfade alle Aktionen und Befehle unterstützen. Die nötigen Kontrollsignale werden zunächst ignoriert.

7.3.1 Fetch

Zuerst wird der Befehl, dessen Adresse in BZ steht, aus dem BS geladen. Diesen Vorgang nennt man *fetch*. Aus dem BZ über den BZAT (BZ-Adresstreiber) wird die Adresse auf den Adressbus gegeben. Der Befehl kommt auf den Datenbus. Er wird in das BR geladen. Die letzten 24 Bit im BR enthalten immer ein x, das entweder als Adresse oder als Rechenkonstante interpretiert wird.

7.3.2 LOAD x

Die Adresse des zu ladenden Datums wird über den BRAT auf den Adressbus gegeben. Das Datum gelangt auf den Datenbus. Das Datum gelangt über den DDIT in den ACC.

7.3.3 LOAD I

Der Inhalt von I ist linkes Argument der ALU. Das rechte Argument ist eine 0. In der ALU wird die Summe gebildet. Das Ergebnis gelangt über den ALUDIT und den Datenbus in den ACC, wo es gespeichert wird.

7.3.4 LOAD I x

Der Wert, der im Indexregister I gespeichert ist, kommt als linkes Argument in die ALU. Der Wert x kommt aus dem BR als rechtes Argument in die ALU. Dort wird die Summe gebildet. Das Ergebnis wird über den ALUAT auf den Adressbus gegeben. Im Speicher wird der Inhalt der entsprechenden Speicherzelle auf den Datenbus gegeben. Dieser wird im ACC geladen.

7.3.5 LOADNUM x

Die Zahl x aus dem BR kommt über den BRT als rechtes Argument in die ALU. Als linkes Argument wird eine 0 verwendet. Das Ergebnis wird über den ALUDIT in ACC geladen.

7.3.6 STORE x

Die Adresse x gelangt aus dem BR über den BRAT auf den Adressbus. Der Wert des ACC gelangt über den ACCDT auf den Datenbus. Der Wert wird in die richtige Speicherzelle des RS gespeichert.

7.3.7 STORE I

Der Wert von ACC gelangt über den ACCDT und den DDIT auf den Datenbus. Er wird in das Register I geladen.

7.3.8 STORE I x

In der ALU wird die Summe aus I und x gebildet. Der Wert des ACC gelangt über den ACCDT auf den Datenbus. Der Wert wird in die richtige Speicherzelle des RS gespeichert.

7.3.9 STOREINT A B

Der Wert des entsprechenden Registers ist linkes Argument der ALU. Das rechte Argument ist 0. Das Ergebnis der Addition gelangt über den ALUDIT auf den Eingang der Register und wird am richtigen Register gespeichert.

7.4 Kodierung der Befehle

Wie bereits erwähnt werden die Befehle als 32-Bit Wörter $(b_{31}, \ldots, b_0)$ kodiert. Die letzten 24 Bit dienen der Kodierung einer Zahl, die entweder die Bedeutung einer Adresse oder die Bedeutung einer Rechenkonstante hat. Die Kodierung wird jetzt im einzelnen besprochen.

7.4.1 COMPUTE

COMPUTE hat folgende Kodierung.

BR	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	$b_{23}b_{22}\ldots b_0$
COMPUTE	0	0	R	*	S_1	S_0	L_1	L_0	x

Das Bit R gibt an, welches der rechte Operand der ALU ist. Besteht der rechte Operand aus den niederen 24 Bit im BR, so ist $R = 0$; kommt der rechte Operand vom Datenbus, so ist $R = 1$. Die Bits S_1, S_0 steuern die ALU. Die Bits L_1, L_0 bestimmen den linken Operanden der ALU und das Ziel des Ergebnisses und zwar mittels folgender Kodierung.

L_1L_0	00	01	10	11
Register	BZ	IR1	IR2	ACC

7.4.2 LOAD

Die Kodierung von LOAD-Befehlen sieht so aus.

BR	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	$b_{23}b_{22}\ldots b_0$
LOAD	0	1	M_1	M_0	*	*	D_1	D_0	x

Die Art (Mode) des LOAD-Befehls wird in folgender Weise von den Bits M_1, M_0 bestimmt.

M_1M_0	00	01	10	11
Mode	LOAD x	LOAD IR1 x	LOAD IR2 x	LOADNUM

Die Befehle LOAD I entsprechen STOREINT-Befehlen, die weiter unten besprochen werden.

7.4.3 STORE

Die STORE-Befehle sind folgendermaßen kodiert:

BR	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	$b_{23}b_{22}\ldots b_0$
STORE	1	0	M_1	M_0	*	*	*	*	x

Die Art (Mode) des STORE-Befehls wird in folgender Weise von den Bits M_1, M_0 bestimmt.

M_1M_0	00	01	10	11
Mode	STORE x	STORE IR1 x	STORE IR2 x	STOREINT

Bei STOREINT werden die Bits 27, 26 etc. allerdings anders besetzt als bei dem gewöhnlichen STORE. Dies wird nun erläutert.

7.4.4 STOREINT

Die STOREINT-Befehle werden folgendermaßen kodiert.

BR	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	$b_{23}b_{22}\dots b_0$
STOREINT	1	0	1	1	S_1	S_0	D_1	D_0	$*\dots*$

Die Source-Bits S_1, S_0 und die Destination-Bits D_1, D_0 haben die gleiche Bedeutung wie die Bits L_1, L_0 in Abschnitt 7.4.1. Es wird aus der Source in die Destination gespeichert.

7.4.5 JUMP

Die JUMP-Befehle sind so kodiert.

	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	$b_{23}b_{22}\dots b_0$
JUMP	1	1	$<$	$=$	$>$	DIR	*	*	x

Für $DIR = 0$ wird vorwärts gesprungen. Für $DIR = 1$ wird rückwärts gesprungen. Die Bits mit den Nummern 29, 28 und 27 legen die Bedingung für den Sprung fest. Diese Bedingung ist so kodiert:

$b_{29}b_{28}b_{27}$	000	001	010	011	100	101	110	111
Beding.	nie	$ACC > 0$	$ACC = 0$	$ACC \geq 0$	$ACC < 0$	$ACC \neq 0$	$ACC \leq 0$	immer

7.4.6 Übersicht

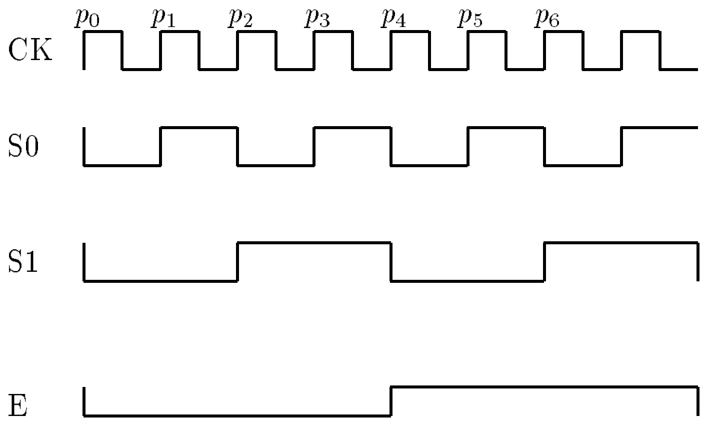
Hier sind noch einmal alle Kodierungen im Überblick.

BR	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	$b_{23}b_{22}\dots b_0$
COMPUTE	0	0	R	*	S_1	S_0	L_1	L_0	x
LOAD	0	1	M_1	M_0	*	*	D_1	D_0	x
STORE	1	0	M_1	M_0	*	*	*	*	x
STOREINT	1	0	1	1	S_1	S_0	D_1	D_0	$*\dots*$
JUMP	1	1	$<$	$=$	$>$	DIR	*	*	x

7.5 Timing Architektur

Die Aktionen der RESA werden von einer Reihe von Kontrollsignalen gesteuert. Deren Erzeugung wird jetzt besprochen. Da ist zunächst die Uhr CK. Diese erzeugt eine Impulsfolge, die idealisiert wie eine Rechteckkurve aussieht.

Ein Zähler macht hieraus drei weitere Uhren $S0$, $S1$ und E. Die Impulsweite von $S0$ ist doppelt, die von $S1$ viermal und die von E achtmal so weit wie die Impulsweite von CK. Die Konstruktion des Zählers bleibt dem Leser als Übung überlassen.

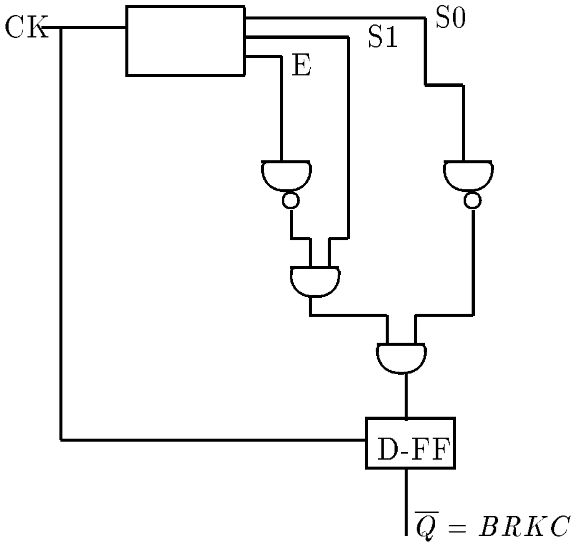


Die Uhr E steuert die beiden Phasen der RESA. Während $E = 0$ ist, wird ein neuer Befehl geladen (fetch). Während $E = 1$ ist, wird dieser Befehl ausgeführt (execute).

Durch Kombination der verschiedenen Uhren in geeigneten Schaltungen werden die notwendigen Steuerimpulse erzeugt. Dies soll an einem Beispiel erläutert werden. Das Befehlsregister soll einen Schreibimpuls $BRCK$ erhalten, der nur genau von p_2 bis p_3 low ist und sonst high. Das bedeutet, daß $BRCK$ genau dann Null sein soll, wenn $E = 0$ und $S1 = 1$ und $S0 = 0$ ist. Wir können also folgende boolsche Formel aufstellen:

$$\overline{BRCK} \quad = \quad \overline{E} \wedge \overline{S0} \wedge S1.$$

Man führt die Ausgänge der Uhr in einen Schaltkreis, der gemäß obiger Formel aufgebaut ist. An dessen Ausgang entsteht durch die unterschiedlichen Verzögerungen eine unregelmäßige Signalfolge. Um diese mit CK zu synchronisieren, führt man den Ausgang in den Dateneingang eines D-Flipflops, der von CK getaktet ist.



Auf diese Weise kann man das Timing der ganzen RESA steuern.

Kapitel 8

Übersetzung von Prozeduren

8.1 Kellerrahmen

In diesem Abschnitt besprechen wir, wie Deklaration und Aufruf einer Prozedur übersetzt werden. Angenommen, der Übersetzer findet folgendes Programm vor:

```
program nonsense;
    var x, z : int;
    procedure dummy(var x: int)
        var y : int;
        procedure subst(var y: int)
        begin
            y := z
        end;
    begin
        subst(y);
        x := x + y;
        while x < 0 do dummy(x) od;
    end;
begin
    read z, x;
    dummy(x);
end.
```

Zur Laufzeit werden die Adressen, die den Namen x, y, z entsprechen, mit Hilfe der Einträge im Bindungskeller ermittelt. Die Zuweisungen können dann wie üblich interpretiert werden. Diese Adressen sind aber zur Compilezeit noch nicht bekannt. Trotzdem kann der Compiler die Adressberechnung weitgehend vorbereiten. Dies geschieht, indem er einen *Kellerrahmen* vorbereitet, der dann zur Laufzeit ausgefüllt wird. Der Kellerrahmen ist ein Organisationsschema, das den Speicherbedarf angibt, der bei einem Prozeduraufruf zur Laufzeit anfällt. Außerdem enthält der Kellerrahmen Informationen, die bei der Adreßberechnung helfen und bereits zur Compilezeit bekannt sind.

Bei jedem Prozeduraufruf wird der Kellerrahmen entsprechend ausgefüllt und oben auf den

Keller geschrieben. Dies nennt man eine *Inkarnation* der Prozedur. Beim Verlassen der Prozedur wird diese Inkarnation aus dem Keller gelöscht.

Der Kellerrahmen beginnt mit den *organisatorischen Zellen.* Dies sind

- eine Zelle SVV für die Anfangsadresse des *statischen Vorgängers.* Das ist die letzte Inkarnation des Blocks, in dem die aufgerufene Prozedur deklariert wurde. Diese braucht man zur Adreßberechnung.
- eine Zelle DVV für die Anfangsadresse des *dynamischen Vorgängers.* Das ist der vorhergehende Rahmen im Bindungskeller. Diese braucht man, um die Inkarnation beim Verlassen der Prozedur zu eliminieren und zum dynamischen Vorgänger zurückzukommen
- eine Zelle RSA. Diese enthält die schon zur Compilezeit bekannte *Rücksprungadresse*, d.h. die Adresse des Befehls im Codespeicher, der nach dem Prozeduraufruf ausgeführt werden muß.

Der Datenbereich enthält ferner Platz für die Werte und Adressen der aktuellen Parameter und Platz für die lokalen Variablen der Prozedur. Die Anfangsadresse des aktuellen Kellerrahmens steht im Register MP (Mark Pointer).

Dies wird an obigem Beispiel illustriert. Bei der Übersetzung der Anweisung

dummy(x)

wird ein Kellerrahmen erzeugt. Dieser enthält

- Platz für SVV, DVV und den Wert von RSA. Dabei enthält RSA die Adresse der Speicherzelle im Programmspeicher, in der sich die Übersetzung des PROSA-Befehls "end." befindet.
- Platz für die Adresse von x und y.

Die anderen Informationen, die der Kellerrahmen noch enthält, werden in den folgenden Abschnitten erläutert.

8.2 Adressberechnung

Beim Aufruf einer Prozedur müssen den lokalen Namen Adressen oder Werte zugeordnet werden. Diese sind aber zur Compilezeit nicht bekannt, weil man nicht weiß, in welcher Kellertiefe die entsprechende Inkarnation der Prozedur sitzt und was die Adressen und Werte der aktuellen Parameter bei dem Aufruf sind. Was aber zur Compilezeit bekannt ist, ist der Speicherbedarf für die formalen Parameter und lokale Variablen. In einer Inkarnation von "dummy" braucht man Platz für die lokale Variable y. Sind bei Anlegen des Prozedurrahmens einer neuen Inkarnation alle Speicherplätze bis zur Nummer $n-1$ besetzt, hat also der erste zu vergebende Platz die Nummer $n = MP$, so kann man der Variablen y den Speicherplatz $MP+5$ zuordnen. Dazu beachte man, daß die Zelle mit der Nummer MP leer bleibt (um Rückgabewerte bei

Funktionen zu ermöglichen) und die drei folgenden Zellen durch SVV, DVV und RSA schon belegt sind. In der nächsten Zelle mit der Nummer $MP + 4$ wird Platz für den formalen Parameter x gelassen. Gäbe es noch drei weitere lokale Variable, so würden denen entsprechend die Speicherplätze $MP + 6$, $MP + 7$, $MP + 8$ zugeordnet. Die *Relativadresse* von y zu MP ist also 5. Die von den anderen Variablen wäre entsprechend 6, 7, 8. Die Relativadressen können zur Compilezeit ermittelt werden. Man enthält die aktuelle Adresse einer lokalen Variablen bei einer Inkarnation, indem man die Relativadresse zur Adresse MP addiert.

Bei globalen Namen ist die Sache etwas komplizierter. Hier muß man beim Aufruf der Prozedur zunächst die Kette der statischen Vorgänger zurückverfolgen, bis man die erste Definition dieses Namens findet. Man kann dieses Verfahren aber abkürzen, weil man die Anzahl der statischen Vorgänger, die man durchlaufen muß, um das definierende Vorkommen der Namen zu finden, bereits zur Compilezeit feststellen kann. Betrachten wir zur Erläuterung wieder das Beispiel. In der Prozedur "subst" kommt der globale Name z vor. Der statische Vorgänger einer Inkarnation von "subst" muß eine Inkarnation von "dummy" sein. Der statische Vorgänger einer Inkarnation von "dummy" ist die einzige Inkarnation des Hauptprogramms. Unabhängig davon, wo die Inkarnation von "subst" im Bindungskeller steht, muß man zwei Schritte in der Kette der statischen Vorgänger von "subst" zurückgehen, um die Bedeutung des globalen Namens z zu finden. Man sagt auch, die Definition von z befindet sich auf *Schachtelungstiefe* 0. Das angewandte Vorkommen von z befindet sich auf Schachtelungstiefe 2. Im allgemeinen ist die Schachtelungstiefe des Hauptprogramms 0 und die Schachtelungstiefe einer Prozedur ist um eins größer als die Schachtelungstiefe der Programmeinheit, in der sie definiert wurde. Die Schachtelungstiefe von von "dummy" ist also eins, die von "subst" ist zwei. Die Anzahl der Schritte, die man die Kette der statischen Vorgänger zurückverfolgen muß, um vom angewandten Vorkommen eines Namens zum zuständigen definierenden Vorkommen zu kommen, ist die Differenz der Schachtelungstiefen von angewandtem und definierendem Vorkommen.

Zur Berechnung der Adressen benötigt man die Prozedur *base*, die verwendet wird, um Basisadressen von Inkarnationen zu bestimmen, die man zur Bestimmung der Bedeutung eines angewandten Vorkommens eines Namens braucht:

$$base(p, a) \quad = \quad \text{if } p = 0 \text{ then } a \text{ else } base(p-1, STORE[a+1])$$

Die Adresse einer lokalen Variablen mit der Relativadresse q ist dann $base(0, MP) + q$. Bei der Adressierung globaler Variablen wird davon ausgegangen, daß die Relativadresse von SVV den Wert 1 hat. In der Speicherzelle $MP+1$ steht also der Verweis auf den statischen Vorgänger. In der Speicherzelle $STORE[MP + 1] + 1$ steht der Verweis auf dessen statischen Vorgänger. Beginnt man also mit $a = MP$, so ist zu Beginn der Prozedur *base* der Wert von a die Basisadresse des Kellerrahmens der aufgerufenen Prozedur. Nach einer Iteration ist a die Basisadresse des statischen Vorgängers. Nach einer weiteren Iteration ist a die Adresse von dessen statischem Vorgänger und so weiter. Im allgemeinen ist also $base(p, MP) + q$ die Adresse, die man erhält, indem man die Relativadresse q zur Basisadresse des p-ten statischen Vorgängers addiert. Das ist also die Adresse, die ein Name hat, dessen Definition im p-ten statischen Vorgänger steht und der dort die Relativadresse q hat.

Um Anweisungen in Prozeduren zu übersetzen, werden folgende Befehle der P-Maschine benötigt:

Befehl	Wirkung
lod s p q	$SP := SP + 1; STORE[SP] := STORE[base(p, MP) + q]$
lda s p q	$SP := SP + 1; STORE[SP] := base(p, MP) + q$
str s p q	$STORE[base(p, MP) + q] := STORE[SP]; SP := SP - 1$

Der Befehl **lod** schreibt oben auf den Stack den Inhalt des Speicherplatzes, der im p-ten statischen Vorgänger die Relativadresse q hat; der Befehl **lda** schreibt die Adresse dieses Speicherplatzes oben auf den Stack. Der Befehl **str** schließlich schreibt in den Speicherplatz, der im p-ten statischen Vorgänger die Relativadresse q hat, den Wert, der oben auf dem Stack steht.

8.3 Berechnung der Adreßumgebungen

In Hauptprogrammen wurde bei der Übersetzung einer Variablendeklaration eine *Adreßumgebung* ρ erzeugt. Das ist eine Funktion, die einem Variablennamen einen Speicherplatz zuordnet. In Prozeduren muß man zu den bisher bekannten Deklarationen auch Parameterspezifikationen übersetzen. Außerdem kann es sein, daß ein bereits deklarierter Name neu deklariert wird. Dies erfordert eine Erweiterung des Konzepts zur Übersetzung von Deklarationen. Für formale Variablen wird eine Zelle reserviert, in der ein Verweis auf ihren Inhalt oder ihr Inhalt selbst steht. Außerdem bekommt jede Prozedur bei ihrer Deklaration eine eigene Adreßumgebung, in der den Namen Relativadressen und Schachtelungstiefen bzw. im Fall von Prozedurnamen die Anfangsadresse des Prozedurcodes und die Schachtelungstiefe zugeordnet wird. Die Relativadressen der lokalen Namen, die in einer umgebenden Programmeinheit schon deklariert waren, werden überschrieben.

Es werden drei Funktionen eingeführt, die bei der Übersetzung von Deklarationen verwendet werden. Die Funktion *elab_specs* macht aus einer Liste von Parameterspezifikationen, einer Adreßumgebung, einer nächsten zu vergebenden Relativadresse und einer Schachtelungstiefe eine neue Adreßumgebung und eine neue freie nächste Relativadresse. Die Funktion ist rekursiv definiert als $elab\_specs\ \epsilon\ \rho\ n_a\ st = \rho\ n_a\ st$ und

$$elab\_specs\ (\mathrm{var}\ x : s; specs)\ \rho\ n_a\ st \quad = \quad elab\_specs\ specs\ \rho[x/(n_a, st)]\ (n_a + 1)\ st.$$

Sind die formalen Parameter verarbeitet, kommen die Variablendeklarationen dran. Die Funktion *elab_vdecls* ordnet einer Folge von Variablen- und Konstantendeklarationen, einer Adreßumgebung, einer nächsten zu vergebenden Relativadresse und einer Schachtelungstiefe eine neue Adreßumgebung und die Relativadresse der nächsten freien Zelle hinter dem Platz für die Deklarationen zu. Feld- und Recorddeklarationen werden analog behandelt. Die Definition für die leere Liste läuft analog zu der Funktion *elab_specs*.

$$elab\_vdecls\ (\mathrm{var}\ x : s; vdecls)\ \rho\ n\_a\ st \quad = \quad elab\_vdecls\ vdecls\ \rho[x/(n\_a, st)]\ (n\_a + gr(s))\ st$$

bzw.

$$elab\_vdecls\ (\mathrm{const}\ x = t; vdecls)\ \rho\ n\_a\ st \quad = \quad elab\_vdecls\ vdecls\ \rho[x/T]\ n\_a\ st.$$

Dabei sei T die Bedeutung des Termes t in der gegebenen Signatur. Die Funktion *elab_pdecls* schließlich ordnet einer Folge von Prozedurdeklarationen, einer Adreßumgebung und einer

Schachtelungstiefe eine neue Adreßumgebung und den markierten Code der übersetzten Prozeduren zu:

$$\begin{aligned}&elab\_pdecls(\text{proc } p_1;\ldots;\text{proc } p_k)\,\rho\ st \quad = \\ &\quad\Big(\rho',\ l_1 : \text{code proc } p_1\ \rho'\ st+1;\ldots,l_k : \text{code proc } p_k\ \rho'\ st+1\Big),\end{aligned}$$

wobei $\rho' = \rho[p_1/(l_1, st), \ldots, p_k/(l_k, st)]$ und proc p_i die komplette Deklaration einer Prozedur mit Namen p_i ist. Da die hierin verwendete Funktion code ihrerseits die *elab*-Funktionen benutzt, muß sie als Argumente die aktuelle Adreßumgebung und Schachtelungstiefe erhalten, d.h. die code-Funktion ist nunmehr eine Funktion mit drei Argumenten.

8.4 Übersetzung von Prozeduraufruf und Deklaration

Angenommen, die augenblicklich aktive Prozedur p ruft eine Prozedur q auf. Dann muß

1. SVV, DVV gesetzt werden,
2. MP und RSA gesetzt werden,
3. Werte und Adressen der aktuellen Parameter gesetzt werden,
4. auf die erste Instruktion der Übersetzung von q gesprungen werden.

Hierzu werden die folgenden Befehle benötigt.

Befehl	Wirkung
mst u	$STORE[SP+2] := base(u, MP)$
	$STORE[SP+3] := MP;$
	$SP := SP+4$
cup u v	$MP := SP-(u+3)$
	$STORE[MP+3] := PC$
	$PC := v$
ssp v	$SP := MP+v-1$
retp	$SP := MP-1$
	$PC := STORE[MP+3]$
	$MP := STORE[MP+2]$

Der Befehl **mst** geht die Kette der statischen Vorgänger u Schritte zurück, um die Basisadresse des statischen Vorgängers der aufgerufenen Prozedur zu ermitteln. Diese Basisadresse speichert er in $STORE[SP+2]$, also der Speicherzelle für den SV-Verweis. Im nächsten Speicherplatz steht DVV, also der alte Wert von MP. Der Befehl **cup** berechnet bei einem Bedarf von u Speicherplätzen für die lokalen Namen den neuen Wert von MP, rettet RSA und setzt den PC auf die Adresse v. Die Übersetzung einer Prozedur funktioniert dann folgendermaßen. Es wird

$$(\rho', n\_a') \quad = \quad elab\_specs\ specs\ \rho\ 4\ st$$

berechnet. Damit ist die zu den Parameterspezifikationen gehörende Adreßumgebung bekannt und n_a' ist die nächste freie Relativadresse. Dann wird entsprechend

$$(\rho'', n\_a'') \quad = \quad elab\_vdecls\ vdecls\ \rho'\ n\_a'\ st$$

und

$$(\rho''', proc\_code) \ = \ elab\_pdecls\ pdecls\ \rho''\ st$$

berechnet. Dann wird die code-Funktion angewendet, deren Argument ein Tripel bestehend aus dem Prozedurcode, der zu Beginn der Übersetzung gültigen Adreßumgebung und der Schachtelungstiefe der aufrufenden Programmeinheit ist. Damit ist

$$\begin{aligned} \mathrm{code}(\mathrm{procedure}\ p(specs); vdecls; pdecls; body)\ \rho\ st \quad &= \\ &\mathbf{ssp}\ n\_a'';\ \mathbf{ujp}\ l;\ proc\_code;\ l : \mathrm{code}\ body\ \rho'''\ st;\ \mathbf{retp}\ ; \end{aligned}$$

Dabei passiert folgendes. Der Speicherbereich, der für die organisatorischen Zellen und für die lokalen Name gebraucht wird, wird übersprungen. Im Code-Speicher wird zum ersten Befehl des Prozedurrumpfes gesprungen. Das ist nötig, weil durch die rekursive Auswertung des Befehls *elab_pdecls* vor dem Code für den Prozedurrumpf noch anderer Code stehen kann, der zu Prozeduren gehört, die auf niedrigerer Schachtelungstiefe deklariert wurden. Nach Ausführung der Prozedur wird die alte Speichersituation restauriert und zum folgenden Befehl gesprungen. Die Übersetzung von Hauptprogrammen erfolgt analog und wird dem Leser als Übungsaufgabe überlassen.

Nun noch zum Aufruf von Prozeduren. Angenommen, die Prozedur p wird aus der Prozedur q aufgerufen. Hierbei sei $\rho(p) = (l, st')$. Der Prozedurcode beginnt also bei Adresse l im Codespeicher und die Schachtelungstiefe des definierenden Vorkommens von p ist st'. Außerdem sei st die Schachtelungstiefe der aufrufenden Prozedur q. Dann wird der Aufruf so übersetzt:

$$\mathrm{code}\ p(e_1, \ldots, e_k) \quad = \quad \mathbf{mst}\ st - st';\ \mathrm{code}_A\ e_1\ \rho\ st;\ \ldots;\ \mathrm{code}_A\ e_k\ \rho\ st; \mathbf{cup}\ s\ l$$

Dabei sei s der Speicherplatz, den man zur Abspeicherung aller aktuellen Parameter benötigt (also der Platz, der bei Übersetzung der Prozedurdeklaration als Platz für formale Parameter reserviert wurde). Zuerst wird also Code erzeugt, der die Besetzung der organisatorischen Zellen regelt. Dann geschieht die Parameterübergabe (durch code_A) und schließlich wird zum übersetzten Code der Prozedur gesprungen.

8.5 Parameterübergabe

Bei Übergabe eines Variablenparameters wird die Adresse des aktuellen Parameters in die für den formalen Parameter reservierte Speicherstelle geladen. Die Adresse des aktuellen Parameters ergibt sich ja immer durch die Differenz d der Schachtelungstiefen von angewandtem und definierendem Vorkommen und der Relativadresse q. Variablenparameter werden also so übersetzt:

$$\mathrm{code}_A\ \ x\ \rho\ st \quad = \quad \mathrm{code}_L\ \ x\ \rho\ st \quad = \quad \mathbf{lda}\ \ a\ d\ q.$$

Hierin ist jetzt auch noch die Funktion code_L neu definiert worden. Auch bei Übersetzung des Rumpfes einer Prozedur muß immer diese neue Funktion code_L benutzt werden. Außerdem muß in Prozeduren die in dem zweiten Kapitel benutzte Übersetzungsfunktion code_R auch so verändert werden, daß die Adressierung immer über eine Basisadresse und eine Relativadresse erfolgt.

Aktuelle Konstantenparameter sind Ausdrücke e vom richtigen Typ. Diese müssen bei der Parameterübergabe ausgewertet und dann in den reservierten Speicherplatz geschrieben werden. Damit werden Übergaben von Konstantenparametern so übersetzt:

$$\mathrm{code}_A \; e \; \rho \; st \quad = \quad \mathrm{code}_R \; e \; \rho \; st.$$

Die Auswertung von code_R erfolgt dabei rekursiv unter Verwendung der Relativadressen und der Schachtelungstiefen.