Computersysteme

2. Grundlagen Digitaler Schaltungen
    - 2.16 Standard-Schaltnetze
    - 2.17 Schaltnetzrealisierung durch Speicher

3. Computer Arithmetik
    - 3.1 Addition
    - 3.2 Subtraktion
    - 3.3 Multiplikation
    - 3.4 Division

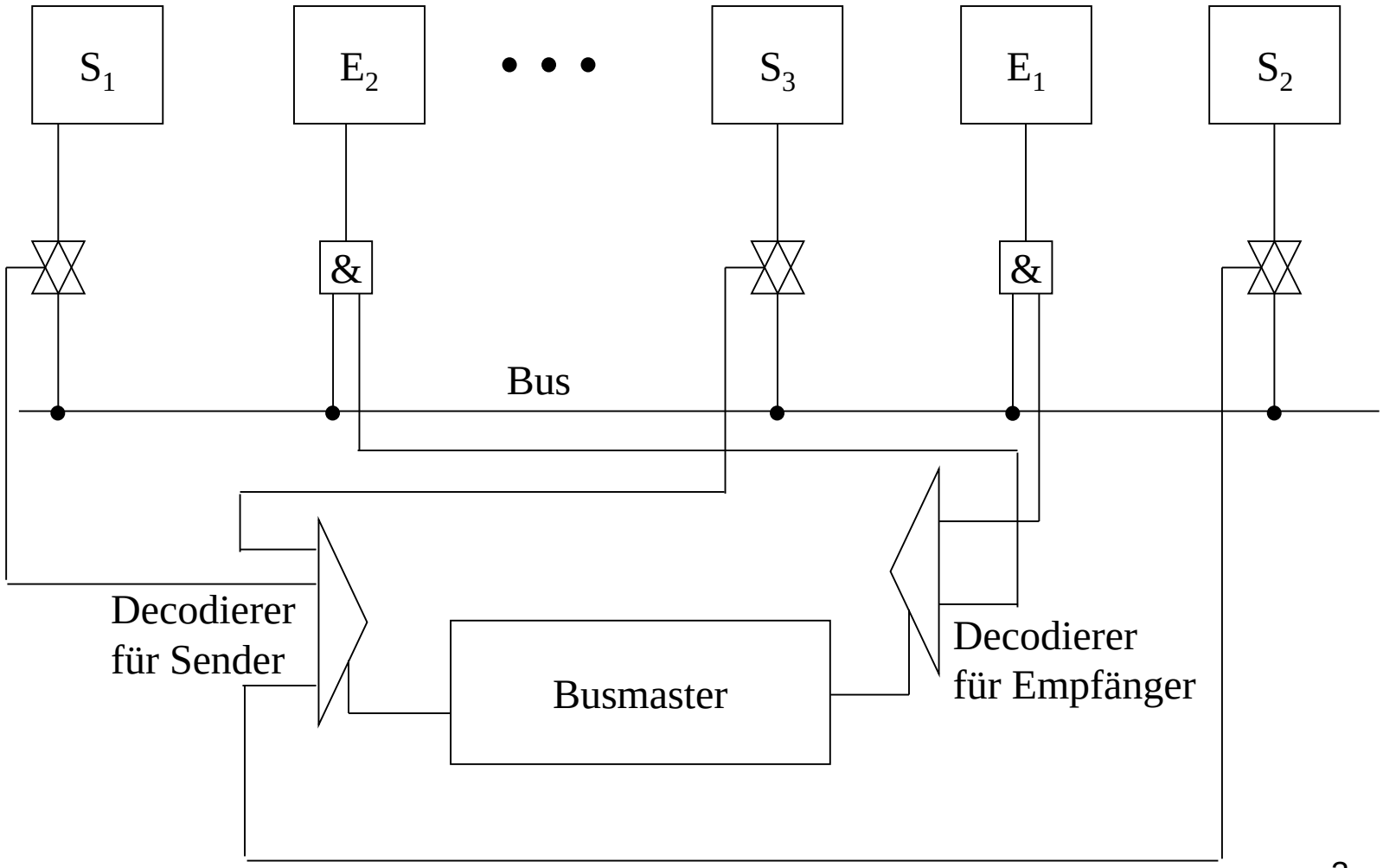
Der Datenbus

Ein Zeitmultiplexschema ist sehr statisch, denn jedes Kommunikationspaar bekommt einen festen Anteil an der Übertragungsbandbreite der Leitung. In vielen Anwendungen in der Informatik ist es allerdings wesentlich wichtiger, dass die Leitungsbandbreite dynamisch jeweils an die Kommunikationspaare zugeteilt werden kann, die aktuell den Bedarf zur Kommunikation hat. Dies führt zum Konzept des Datenbus.

Ein Datenbus ist eine Leitung oder ein Leitungsbündel, über das verschiedene angeschlossene Geräte Daten austauschen können. Diese Geräte können schreibend oder lesend auf den Bus zugreifen. Schreibende Zugreifer bezeichnet man auch als Sender, lesende als Empfänger. Natürlich kann dasselbe Gerät manchmal als Sender und manchmal als Empfänger zugreifen.

In seiner einfachsten Form ist der Datenbus eine gemeinsame Leitung (oder ein Leitungsbündel), auf das jeder Teilnehmer über ein Transmissionsgatter schreiben kann und von dem er über einen Datenwegschalter lesen kann. Die Steuerleitungen werden wieder von einer zentralen Einheit (Busmaster, Bus-Controller) bedient, die den Bus an die Kommunikationspaare nach Bedarf zuteilt.

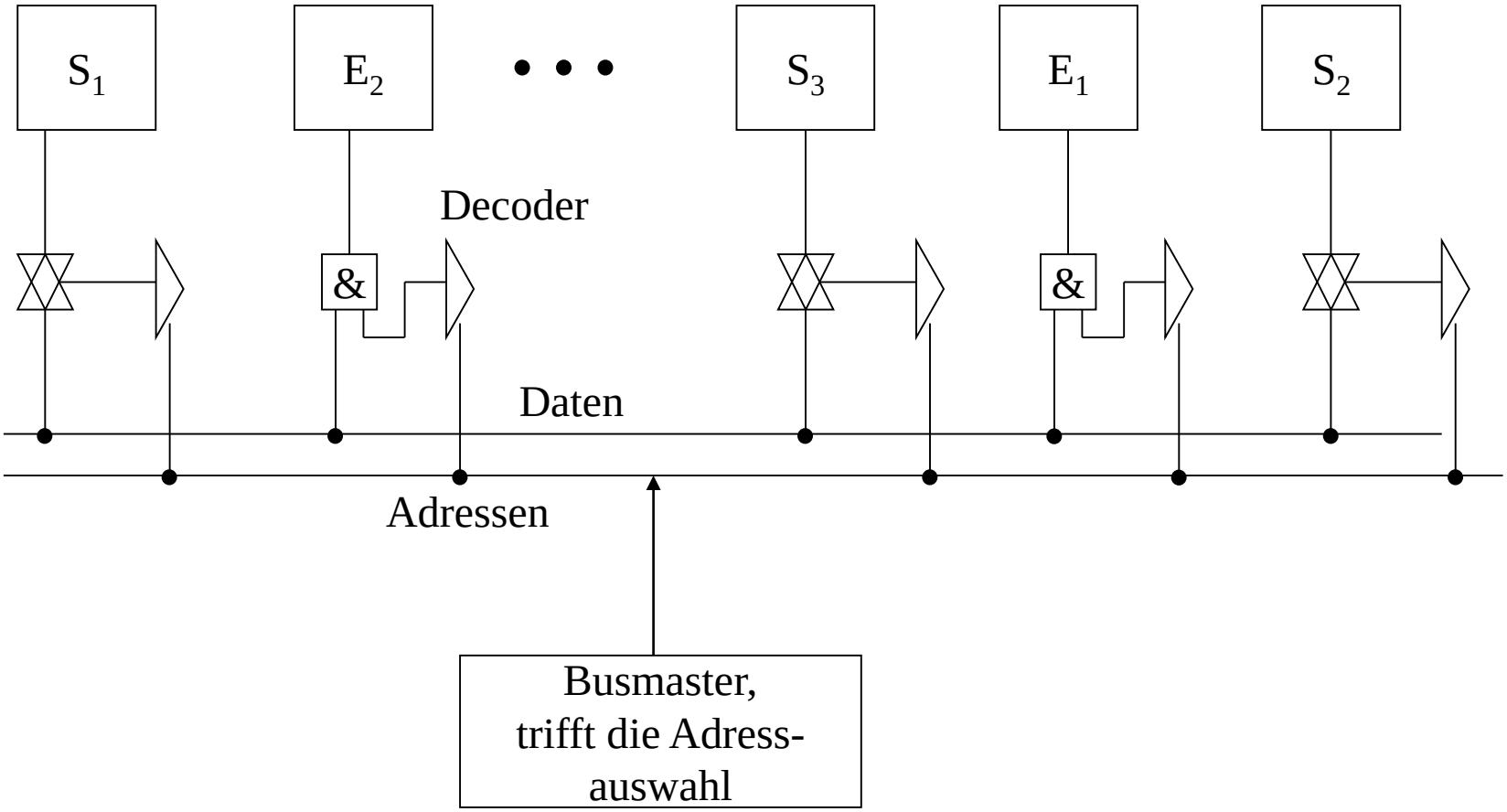
Datenbus


S1
E2
S3
E1
S2
&
&
Bus
Decodierer
für Sender
Busmaster
Decodierer
für Empfänger

Der Nachteil ist offensichtlich: Der Busmaster (Bus Controller, Arbiter) muss eine direkte Leitung von seinem jeweiligen Decodierer zu jedem potentiellen Sender und Empfänger haben, die er zu geeigneter Zeit auf 1 legt, wenn ein Sender etwas für einen der mehrere Empfänger auf den Bus legen möchte.

Dieser Nachteil lässt sich vermeiden, indem man den Bus um ein Adressleitungsbündel ergänzt. Der Busmaster muss jetzt nur noch die Adressen der Kommunikationspartner auf den Adressbus legen, und jeder Sender und Empfänger hat einen Decoder, der den Adressbus liest und im Falle, dass seine Adresse auf dem Bus liegt, eine 1 für sein Transmissionsgatter bzw. seinen Datenwegschalter generiert.

Datenbus mit Adressbus



Nun stellt sich allerdings heraus, dass häufig dieselben beiden Kommunikationspartner nacheinander sehr viele Daten austauschen wollen, bevor das Kommunkationspaar wechselt. In diesem Falle bleibt die Adressauswahl lange gleich, während die Daten dauernd wechseln. Es wäre nun unwirtschaftlich, für die (über lange Zeit konstanten) Adressen ein eigenes Leitungsbündel zu realisieren.

Stattdessen verwendet man das vorhandenen Leitungsbündel für die Daten im Zeitmultiplex auch für die Adressen. Genauer: Am Anfang einer Kommunikation werden die Adressen des Senders und des/der Empfänger auf dem Bus gelegt. Diese beschalten ihre Transmissionsgatter und ihre Datenwegschalter. Danach werden kontinuierlich Daten übertragen, bis der Kommunikationsvorgang beendet ist.

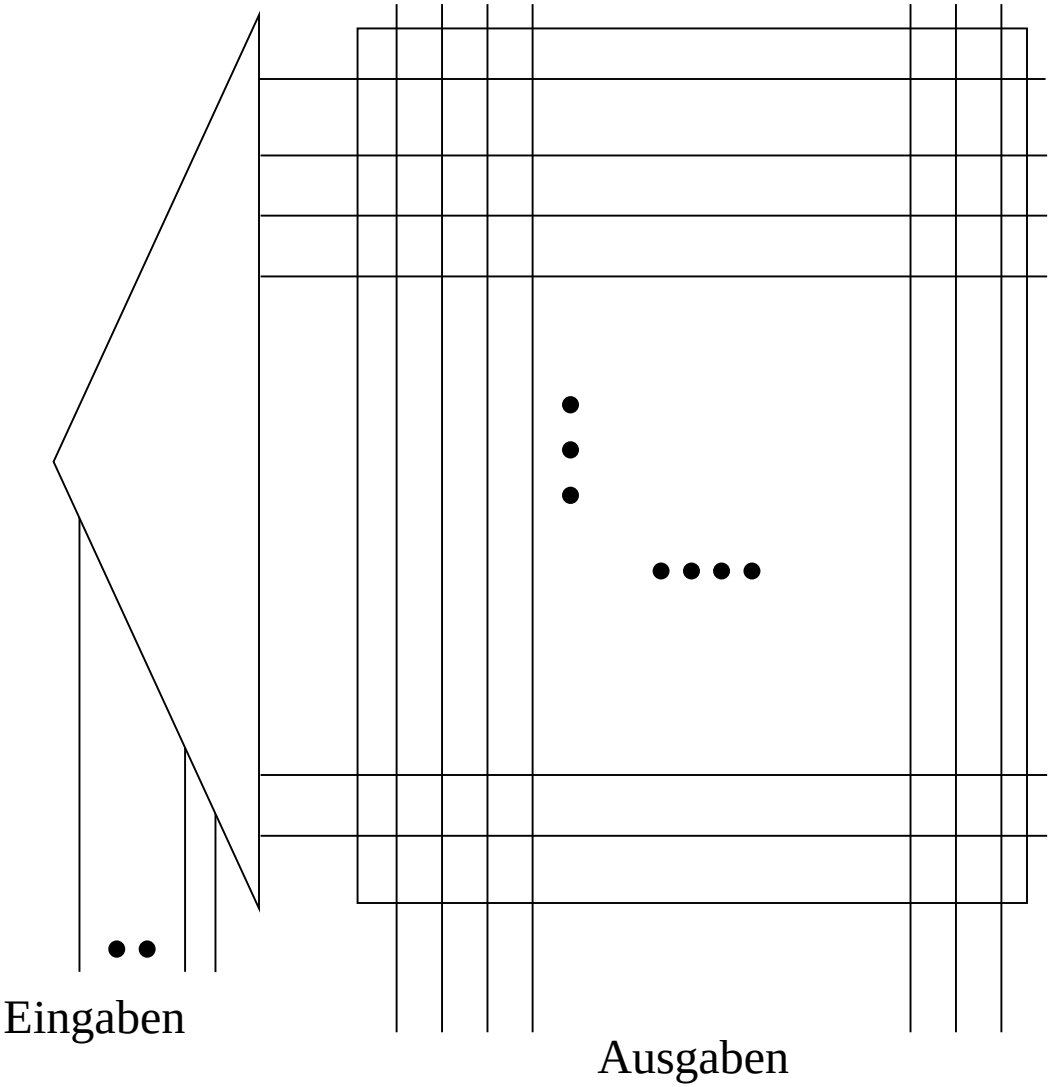
Für einen solchen Bus ist ein Protokoll erforderlich, das bestimmte Absprachen über die Interpretation der Signale auf dem Bus beinhaltet. Zum Beispiel muss jeder Teilnehmer am Bus informiert werden, ob der Bus zur Zeit Adressen oder Daten überträgt. Auch das Ende einer Übertragung muss signalisisert werden.

Schaltnetzrealisierung als Speicher

In der modernen Schaltungstechnik ist es häufig zu teuer, für jedes individuelle Schaltnetz eine eigene Realisierung zu bauen oder gar einen eigenen Chip. Daher gibt es viele gebräuchliche Techniken, wie man bestehende universelle Chips verwenden kann, so dass sie die Funktionalität eines gegebenen Schaltnetzes erfüllen.

Die einfachste davon ist die Realisierung von Schaltnezten als Speicher: In einen Speicher wird die Wertetabelle der Boole‘schen Funktion geschrieben. Die Eingänge werden an die Adressleitungen des Speichers gelegt. Die Ausgänge des Speichers bilden die Ausgänge des Schaltnetzes.

Speicher als Schaltnetzrealisierung

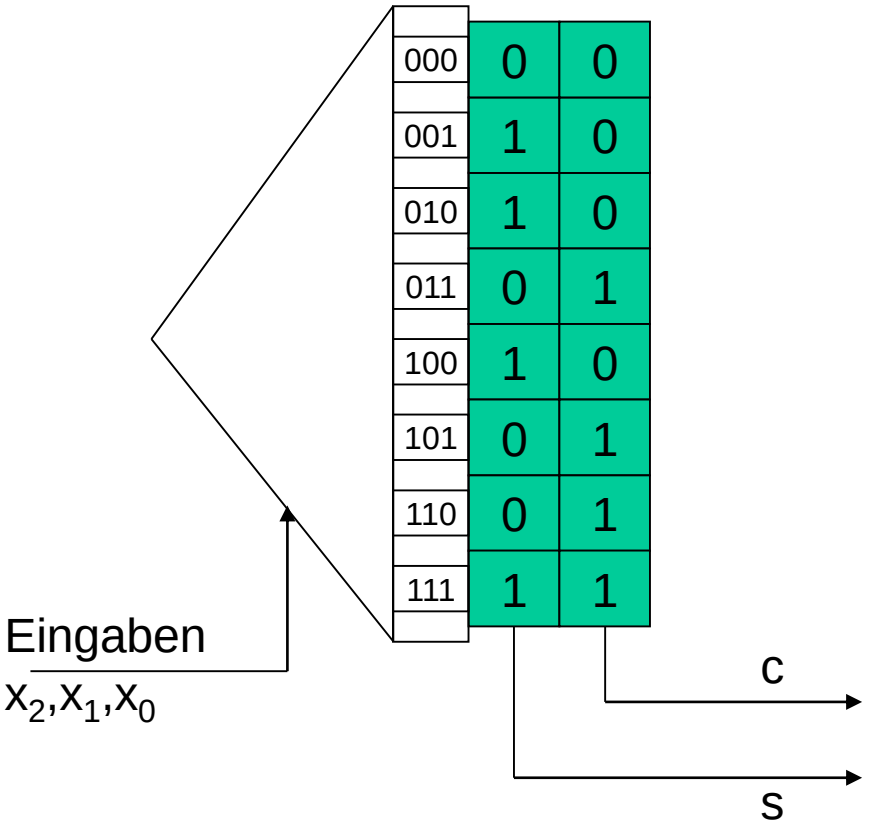


Beispiel: Ein Volladdierer, realisiert mit einem 8x2-Bit ROM

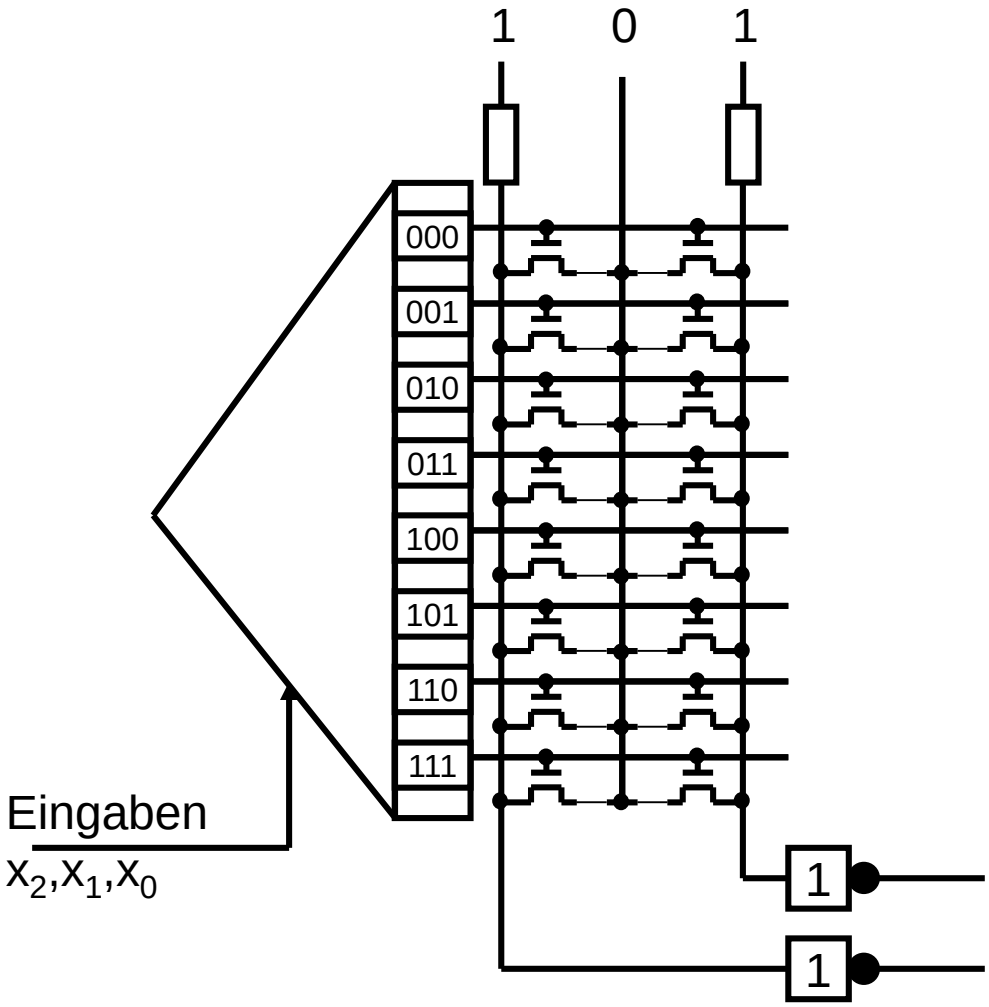
Wertetabelle

x_2	x_1	x_0	s	c
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

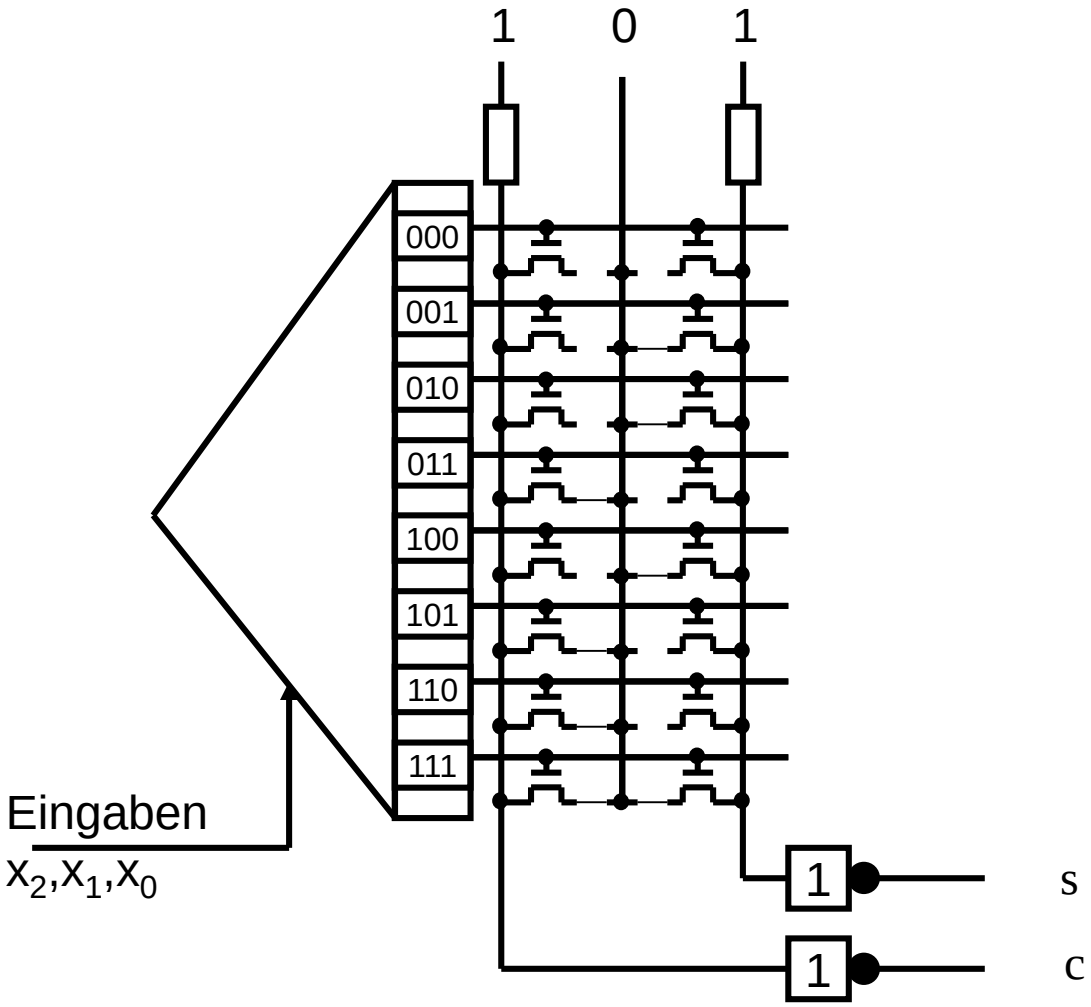
ROM



Beispiel: Ein 8x2-Bit ROM (Antifuse-Technik, d.h. nicht reprogrammierbar) vor dem Brennen zu einem Volladdierer



Beispiel: Ein Volladdierer, realisiert mit einem 8x2-Bit ROM (Antifuse-Technik, d.h. nicht reprogrammierbar)



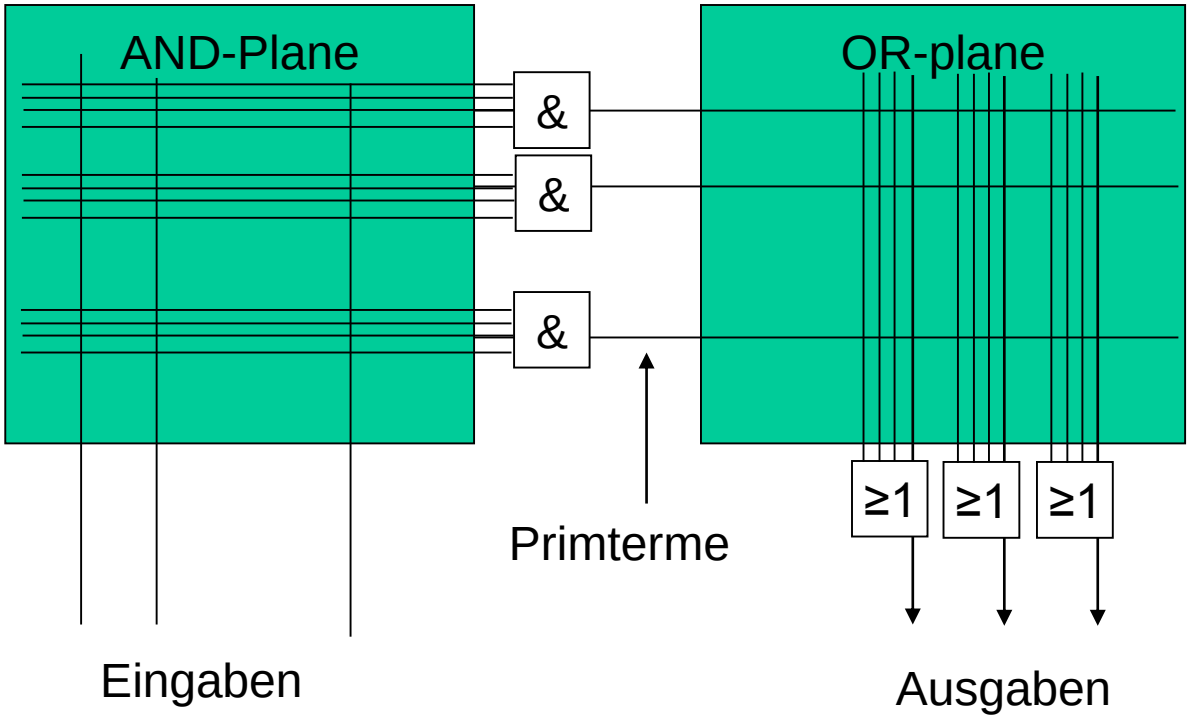
Schaltnetzrealisierung als PLA (Programmable Logic Array)

Eine andere mögliche Realisierung eines Schaltnetzes ist das PLA. Es basiert auf der Idee, ein freies Formular für eine Boole‘sche Funktion mit n Eingängen und m Ausgängen bereitzustellen, das in Abhängigkeit von der individuellen Funktion „ausgefüllt“ (gebrannt) werden kann.

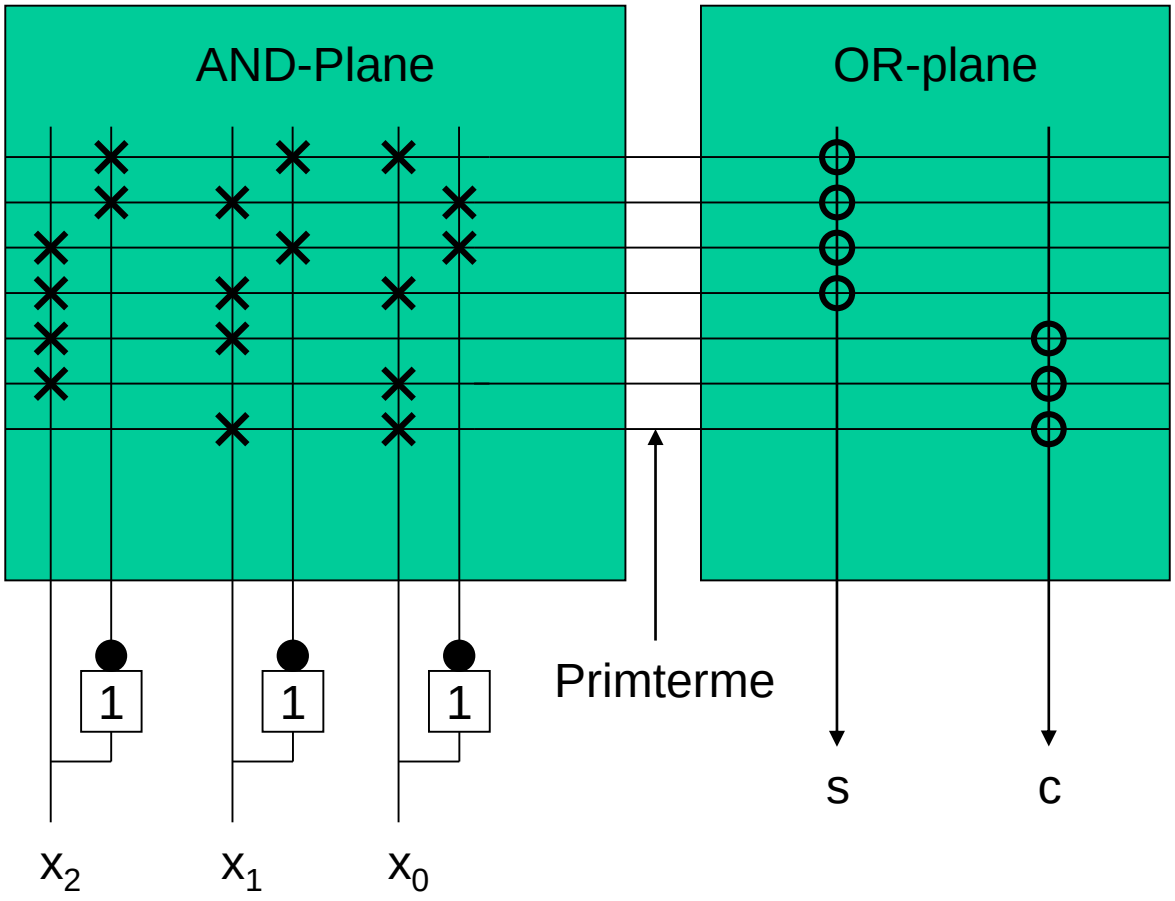
Die AND-Plane generiert die Produktterme (sinnvollerweise in DMF, also Primterme), die OR-Plane fasst diese in jeweils einer Disjunktion zusammen. Natürlich werden in der tatsächlichen Realisiserung nicht AND und OR, sondern NAND-Gatter verwendet. Diese Technik kennen wir bereits.

Eine Schreibweise ist bei solchen Darstellungen sinnvoll: Die AND-Verknüpfung mehrerer Leitungen, die eine Ausgabeleitung schneiden, wird durch Kreuze, die OR-Verknüpfung durch Kreise (Eselsbrücke: Buchstabe O für ODER) über dem Schnittpunkt dargestellt.

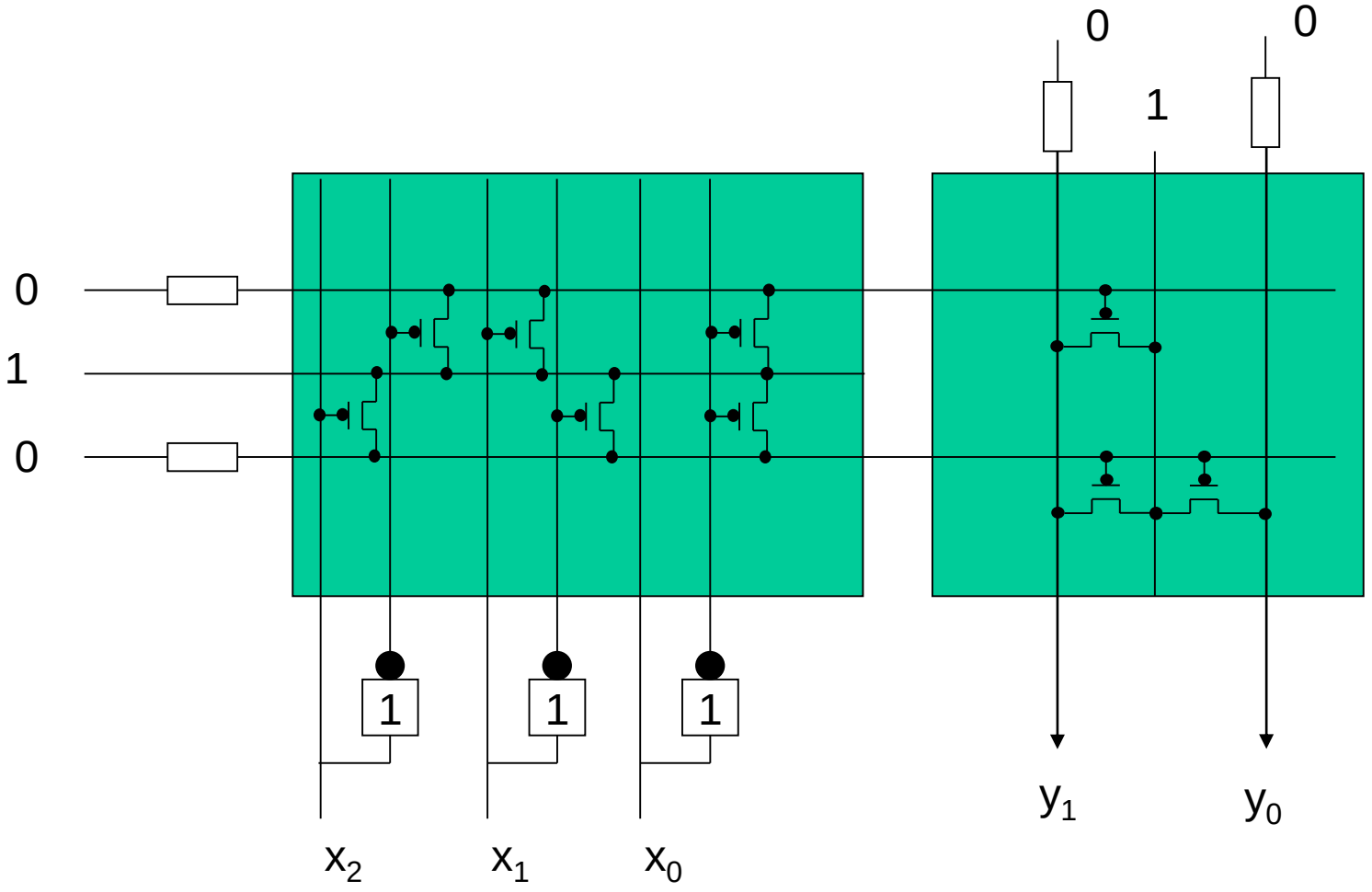
Realisierung eines Schaltnetzes als PLA mit AND- und OR-Plane



Beispiel: Volladdierer als PLA mit AND- und OR-Plane



Beispiel: Funktionen $y_1 = \overline{x}_0 x_1 \overline{x}_2 + \overline{x}_0 \overline{x}_1 x_2$ und $y_0 = \overline{x}_0 \overline{x}_1 x_2$
realisiert als PLA mit AND- und OR-Plane

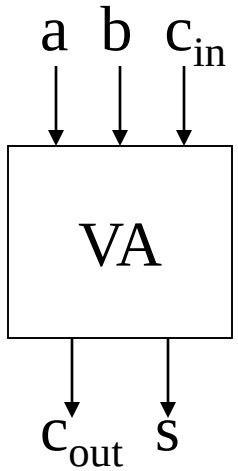


3. Computer Arithmetik

In diesem Abschnitt wollen wir einige grundlegende Techniken kennen lernen, mit denen in Computern arithmetische Operationen ausgeführt werden. Das dabei erworben Wissen werden wir später in den Abschnitten über Schaltwerke, ALU-Aufbau und Rechnerarchitektur vertiefen.

Addition

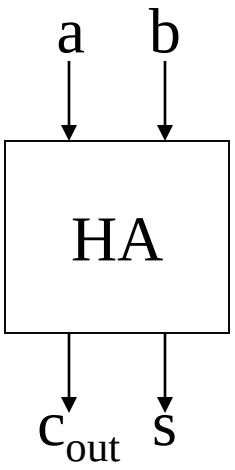
Wir kennen bereits einen Volladdierer. Es ist ein Schaltnetz mit drei Eingängen a, b, c_{in} und zwei Ausgängen s und c_{out}. Der Volladdierer ist in der Lage, drei Bits zu addieren und das Ergebnis als 2-Bit-Zahl auszugeben. Das Ergebnis liegt ja zwischen 0 und 3 und ist daher in zwei Bits zu codieren. Wir sehen hier das Schaltbild eines Volladdierers und seine Wertetabelle:



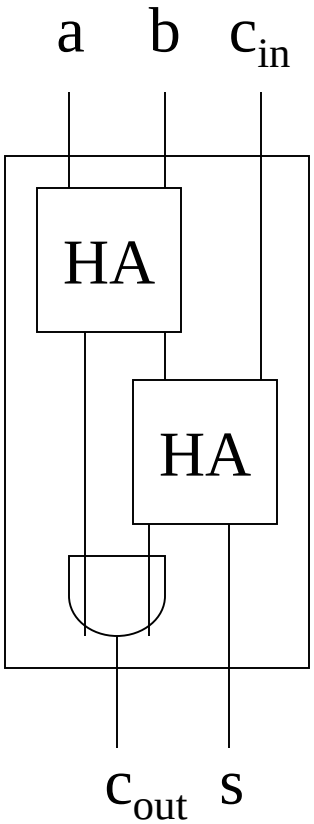
a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Häufig realisiert man einen Volladdierer nicht in DMF sondern in einer mehrstufigen Form, wobei man sogenannte Halbaddierer benutzt. Halbaddierer sind Schaltnetze, die zwei Bits addieren können (und demzufolge ein Ergebnis im Bereich 0 bis 2 produzieren). Durch Zusammenschalten von zwei Halbaddierern und einem Oder-Gatter erhält man die Funktionalität eines Volladdierers. Wie sehen im folgenden das Schaltsymbol eines Halbaddierers, seine Wertetabelle und den Aufbau eines Volladdierers aus Halbaddierern.

a	b	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



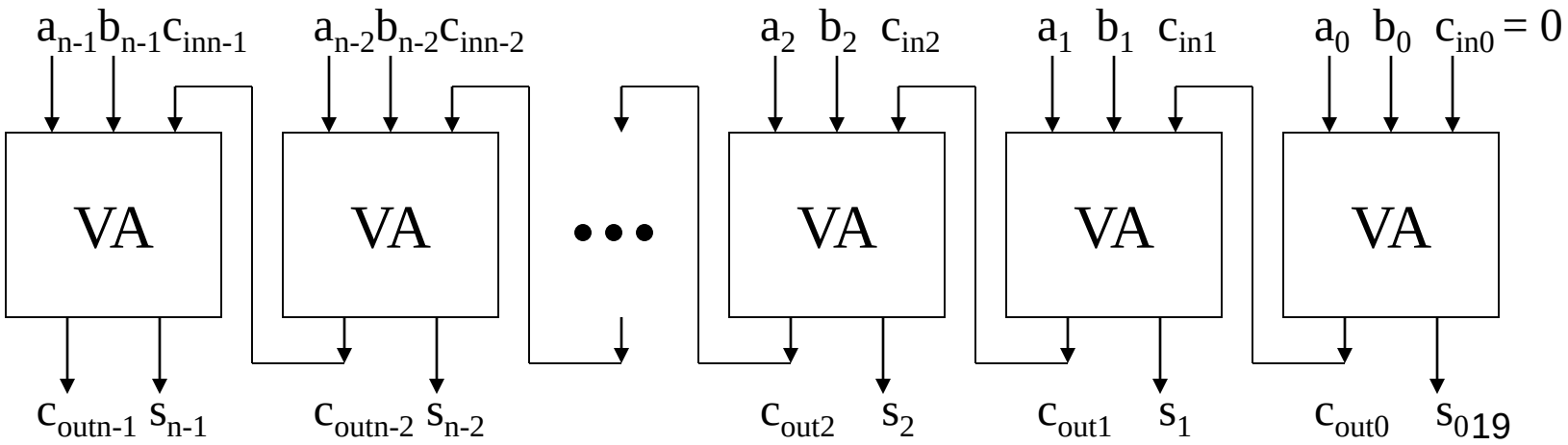
Volladdierer aus zwei Halbaddierern und einem Oder-Gatter



Nun wollen wir aber in der Regel längere Operanden addieren, zum Beispiel die Binärzahlen $a= a_{n-1}a_{n-2}...a_1a_0$ und $b= b_{n-1}b_{n-2}...b_1b_0$. Natürlich könnte man ein dafür erforderliches Addierwerk in DNF oder DMF aufbauen. Dies bringt aber eine Reihe von Problemen mit sich:

- für jedes n ergibt sich eine völlig andere Realisierung
- das Fan-in und das Fan-out an den Gattern wächst polynomiell mit n

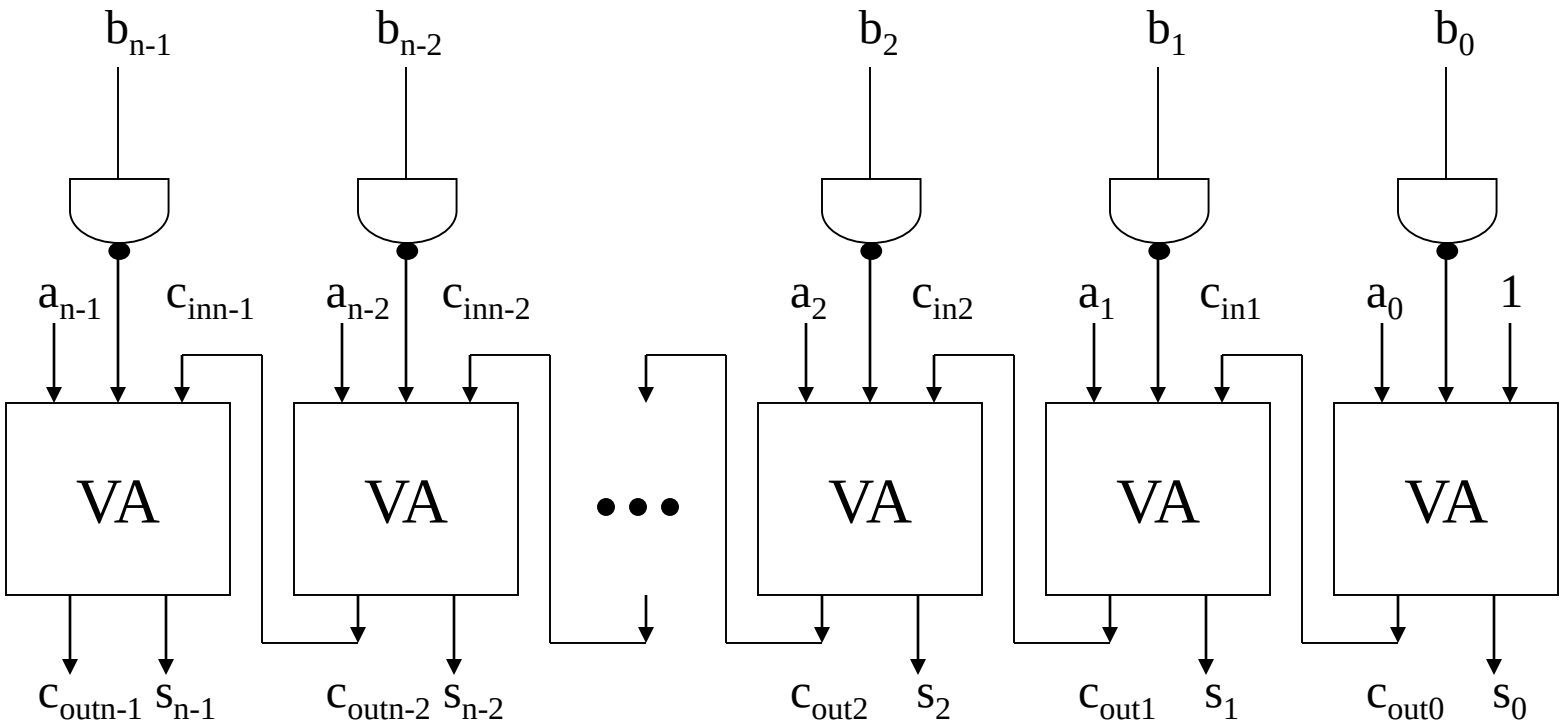
Insbesondere wegen dieser zweiten Eigenschaft ist der zweistufige Aufbau z.B. in DMF nicht sinnvoll. Stattdessen verwendet man im einfachsten Fall eine Kette von Volladdierern, die im Grunde genau das machen, was wir von der Addition in der „Schulmethode“ kennen. Man beginnt mit den LSBs (least significant bits), addiert diese, erzeugt einen Übertrag, mit dessen Kenntnis man das nächste Bit bearbeiten kann, usw. Ein entsprechendes Schaltnetz sieht dann so aus:



Das Ergebnis der Addition von zwei n-Bit-Zahlen ist eine n+1-Bit Zahl. Diese ist repräsentiert durch die Ausgänge $c_{n-1}s_{n-1}s_{n-2....}s_2s_1s_0$.

Einen solchen Addierer nennt man einen **ripple-carry-adder**. Sein Vorteil ist der einfache und modulare Aufbau. Sein wesentlichster Nachteil wird bereits durch diesen Namen ausgedrückt: Wenn die Operanden gerade eine ungünstige Bit-Kombination aufweisen, muß die Carry- (übertrags-) -Information durch alle Volladdierer hindurch von der Stelle mit der geringsten Wertigkeit bis zur Stelle mit der höchsten Wertigkeit hindurchklappern (rippeln). Damit ergibt sich die Schaltzeit eines ripple-carry-adders als proportional zur Zahl n der Stellen. Dies ist insbesondere dann ein Problem, wenn in unserem Rechner ein Zahlenformat mit vielen Bits (z.B. 64 Bits) verarbeitet werden soll. Sicher wollen wir den Maschinentakt nicht so langsam machen, daß in einem Takt 64 Volladdierer nacheinander schalten können. Wir werden bald sehen, wie man dieses Problem behandeln kann.

Zunächst wollen wir uns aber damit beschäftigen, wie man mit einem Addierer auch subtrahieren kann. Wir wissen bereits: das Zweierkomplement einer Zahl läßt sich berechnen als Einerkomplement plus 1. Ferner ist das Einerkomplement die bitweise Negation der Zahl. Wenn wir nun die Addition der 1 über den Carry-Eingang c_{in0} erledigen, können wir mit dem Schaltnetz auf der folgenden Folie a-b berechnen, indem wir zu a das Zweierkomplement von b hinzu addieren:

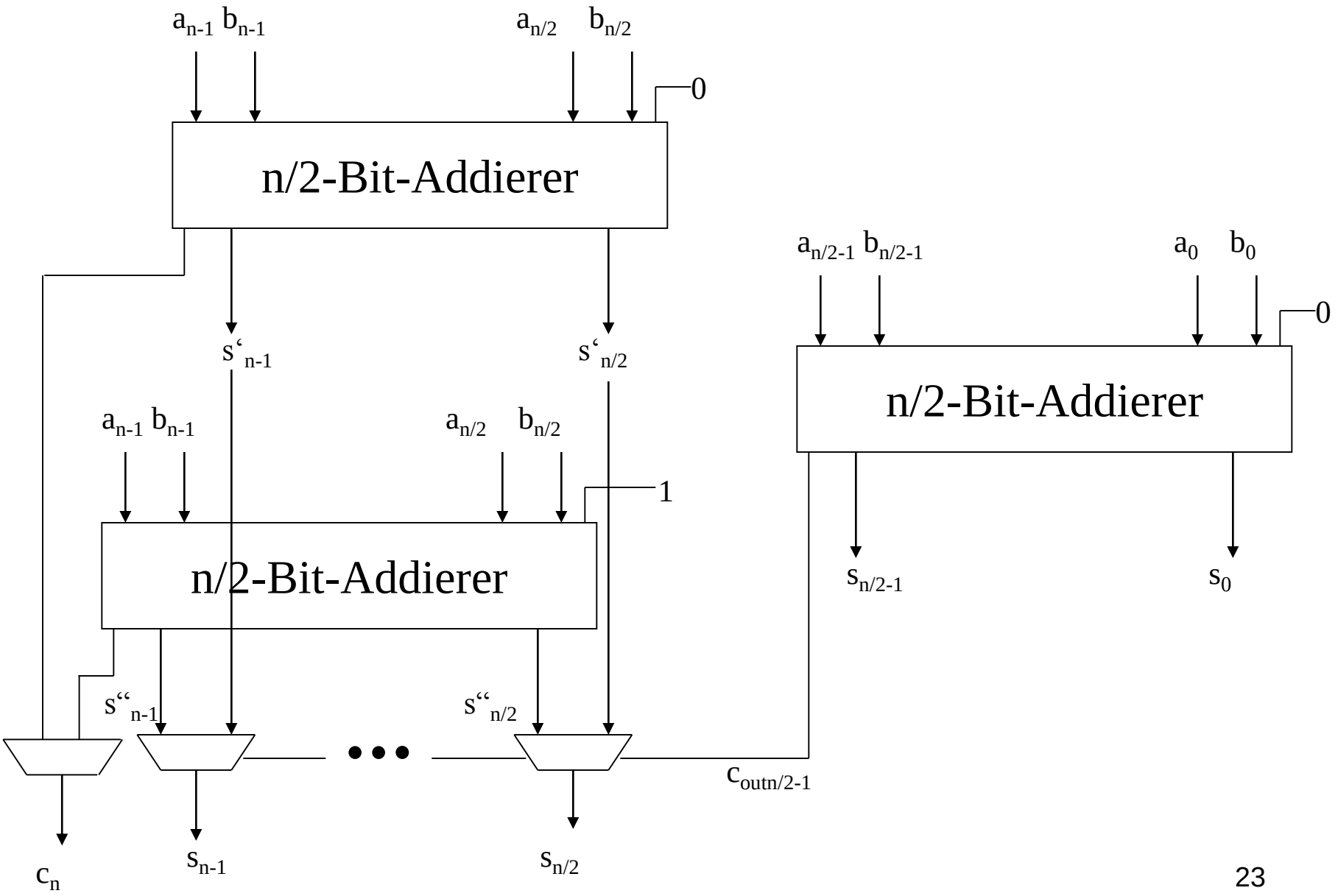

b_{n-1}
b_{n-2}
b_2
b_1
b_0
a_{n-1}
c_{inn-1}
a_{n-2}
c_{inn-2}
a_2
c_{in2}
a_1
c_{in1}
a_0
1
VA
VA
VA
VA
VA
c_{outn-1}
s_{n-1}
c_{outn-2}
s_{n-2}
c_{out2}
s_2
c_{out1}
s_1
c_{out0}
s_0

Schnellere Addition und Subtraktion

Wodurch ergibt sich die Schaltzeit für eine Addition oder Subtraktion? Durch die Anzahl der Volladdierer, durch die ein Carry nacheinander hindurchklappern muß. Wenn wir z.B. +1 und -1 addieren, liegt an den Eingängen des Addierers a = 00000001 und b = 11111111. Bei der Addition entsteht an der letzten Stelle ein Übertrag, dieser bewirkt an der vorletzten Stelle einen Übertrag usw. bis hin zur ersten Stelle, wo schließlich auch ein Übertrag entsteht. Der Zeitaufwand ist also die Anzahl der Stellen, durch die ein Übertrag hindurchwandern muß. Wenn jeder Volladdierer die Zeit t_{VA} benötigt, ist die Gesamtzeit also $n * t_{VA}$. Wie kann man diese Zeit nun vermindern. Eine hübsche Lösung, die auch in der Praxis der Rechnerarchitektur häufig Verwendung findet, bietet der **carry-select-adder**.

Die Idee ist folgende: Der Addierer wird in zwei gleich lange Hälften unterteilt. Und für beide Hälften wird gleichzeitig mit der Addition begonnen. Bei der linken (höher signifikanten) Hälfte wissen wir aber nicht, ob am Carry-Eingang des rechtesten Volladdierers eine 1 oder eine 0 ankommt. Deshalb führen wir die Addition der linken Hälfte gleichzeitig zweimal aus, einmal mit einer 0 am Carry-Eingang und einmal mit einer 1. Wenn die rechte Hälfte mit ihrer Addition fertig ist, kennen wir das eingehende Carry der linken Hälfte. Somit wissen wir, welches der Ergebnisse das richtige ist, das wir sodann auswählen (select). Das andere (falsche) Ergebnis wird einfach verworfen.

Die Auswahl geschieht über eine Menge von Multiplexern, die vom eingehenden Carry gesteuert werden. Das Prinzip ist auf der folgenden Folie dargestellt.


a_{n-1} b_{n-1}
$a_{n/2}$ $b_{n/2}$
0
n/2-Bit-Addierer
s'_{n-1}
$s'_{n/2}$
a_{n-1} b_{n-1}
$a_{n/2}$ $b_{n/2}$
1
n/2-Bit-Addierer
s''_{n-1}
$s''_{n/2}$
$c_{outn/2-1}$
c_n
s_{n-1}
$s_{n/2}$
$a_{n/2-1}$ $b_{n/2-1}$
a_0 b_0
0
n/2-Bit-Addierer
$s_{n/2-1}$
s_0

Wir sehen sofort: der Aufwand an Gattern ist etwas mehr als eineinhalb mal soviel wie beim ripple-carry-adder. Wie ist nun der Schaltzeitaufwand für einen solchen Addierer. Da alle n-2-Bit-Addierer gleichzeitig arbeiten, benötigen wir nur noch die halbe Zeit, nämlich $n/2 * t_{VA}$ für die Addition. Dazu kommt noch eine kleine konstante Zeit für die Multiplexer also ist die Gesamtzeit gleich $n/2 * t_{VA} + t_{MUX}$

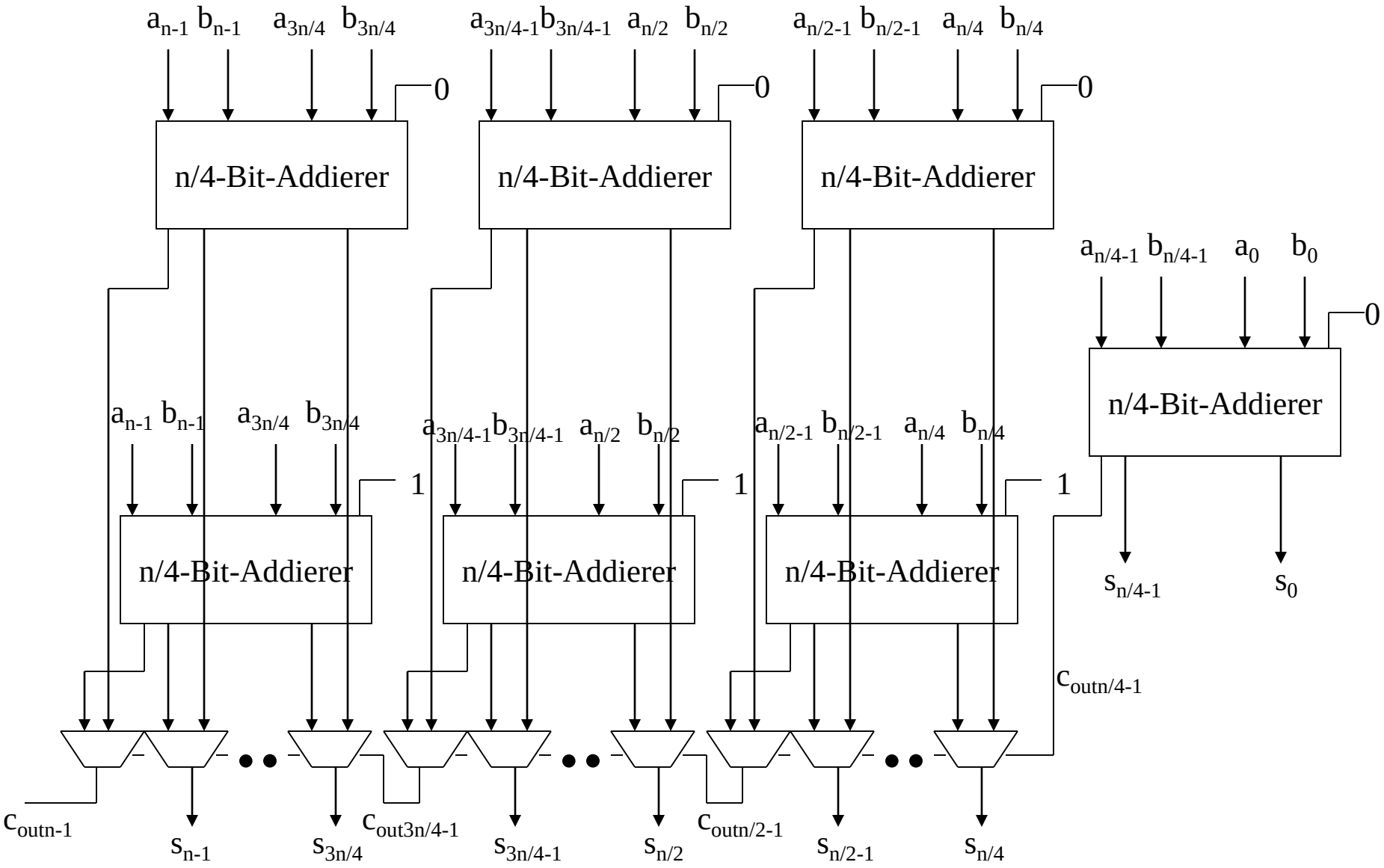
Wir haben also etwa einen Faktor 2 in der Zeit gewonnen. Nun läßt sich dieses Prinzip natürlich wiederholt anwenden: Anstelle von Addierern der Länge n/2 können wir auch solche der Länge n/4 oder n/8 usw. verwenden. Alle solchen Addierer (außer dem am wenigsten signifikanten) werden doppelt ausgelegt, wovon einer mit einem Carryeingang 0 und der andere mit einem Carryeingang 1 arbeitet. Welches der Ergebnisse schließlich verwendet wird, entscheidet das Carry der nächst niedrigeren Stufe.

Die folgende Folie zeigt das Ergebnis dieser Technik für eine Unterteilung in vier Abschnitte. Die Laufzeit reduziert sich auf $n/4 * t_{VA} + 3t_{MUX}$. Allgemein gilt für eine Unterteilung in m abschnitte:

$$t_{Gesamt} = n/m * t_{VA} + (m-1) * t_{MUX}$$

Wenn wir das Minimum für die Gesamtzeit ausrechnen wollen, setzen wir zur Vereinfachung $t_{VA}=t_{MUX}$. Dann müssen wir t_{Gesamt} nach der freien Variablen m ableiten und die Ableitung = 0 setzen.

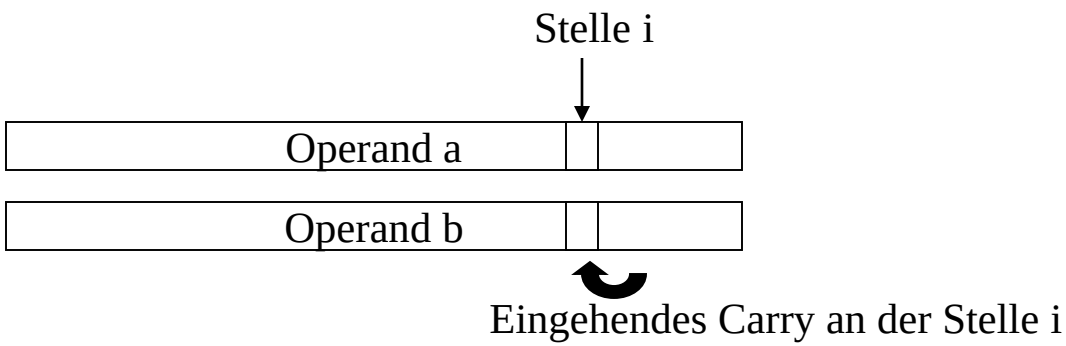
Diese Gesamtzeit nimmt ein Minimum an für $m = n^{1/2}$, also ist die Addition mit einem Carry-select-Addierer in $O(n^{1/2})$, während der Ripple-carry-Addierer in O(n) arbeitete.


a_{n-1} b_{n-1} $a_{3n/4}$ $b_{3n/4}$
$a_{3n/4-1}$ $b_{3n/4-1}$ $a_{n/2}$ $b_{n/2}$
$a_{n/2-1}$ $b_{n/2-1}$ $a_{n/4}$ $b_{n/4}$
$a_{n/4-1}$ $b_{n/4-1}$ a_0 b_0
0
n/4-Bit-Addierer
1
c_{outn-1}
$c_{out3n/4-1}$
$c_{outn/2-1}$
$c_{outn/4-1}$
s_{n-1}
$s_{3n/4}$
$s_{3n/4-1}$
$s_{n/2}$
$s_{n/2-1}$
$s_{n/4}$
$s_{n/4-1}$
s_0

Carry-Lookahead-Addition

Ein Addierertyp, der in heutigen Mikroprozessoren sehr häufig eingesetzt wird, weil er bei kleinem konstanten Faktor asymptotisch optimal in der Zeit ist, ist der Carry-Lookahead-Addierer.

Die Idee besteht darin, für jede Stelle i so früh wie irgend möglich zu bestimmen, ob ein Carry an dieser Stelle eingeht oder nicht.



Dies ist natürlich einfach für die ganz rechte (LSB) Stelle und es wird schwieriger, je weiter wir nach links gehen. Der Trick besteht nun darin, die Carry-Situation auch an den Positionen der höher signifikanten Bits aus der Kenntnis aller beteiligten Operandenbits vorherzusehen (lookahead).

Definition:

Sei M = {0, 1, ..., n-1} , $a = a_{n-1}\, a_{n-2} \ldots a_1\, a_0$, , $b = b_{n-1}\, b_{n-2} \ldots b_1\, b_0$

Wir definieren $p : MxM \longrightarrow \{0, 1\}$ (partiell definiert)
$g : MxM \longrightarrow \{0, 1\}$ (partiell definiert)

für $i \in M$ $p(i,i) := a_i$ XOR b_i
$g(i,i) := a_i \bullet b_i$

für $i,j \in M$, $i<j$ $p(i,j) := p(j,j) \bullet p(i,j-1)$
$g(i,j) := g(j,j) + g(i,j-1) \bullet p(j,j)$

für $i,j \in M$, $i>j$ p(i,j) := undefiniert
g(i,j) := undefiniert

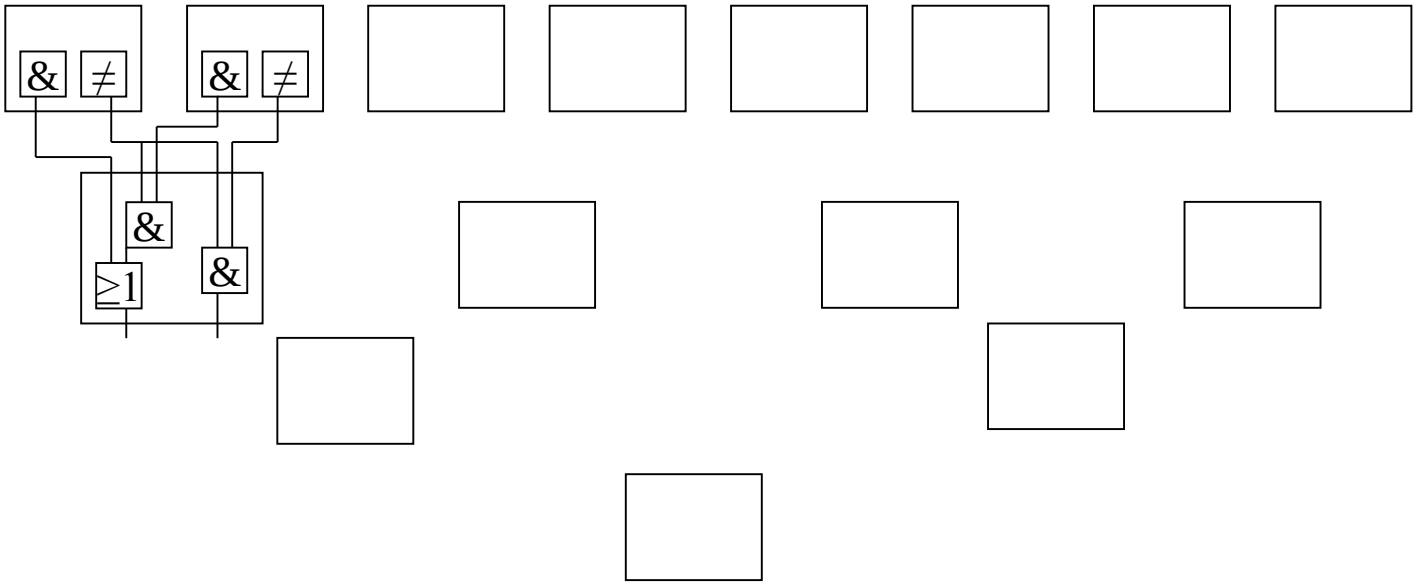
Satz:

(i) für $i < j$ gilt: $p(i,j) = p(j,j) \bullet p(j-1,j-1) \bullet p(j-2,j-2) \bullet \ldots \bullet p(i+1,i+1) \bullet p(i,i)$

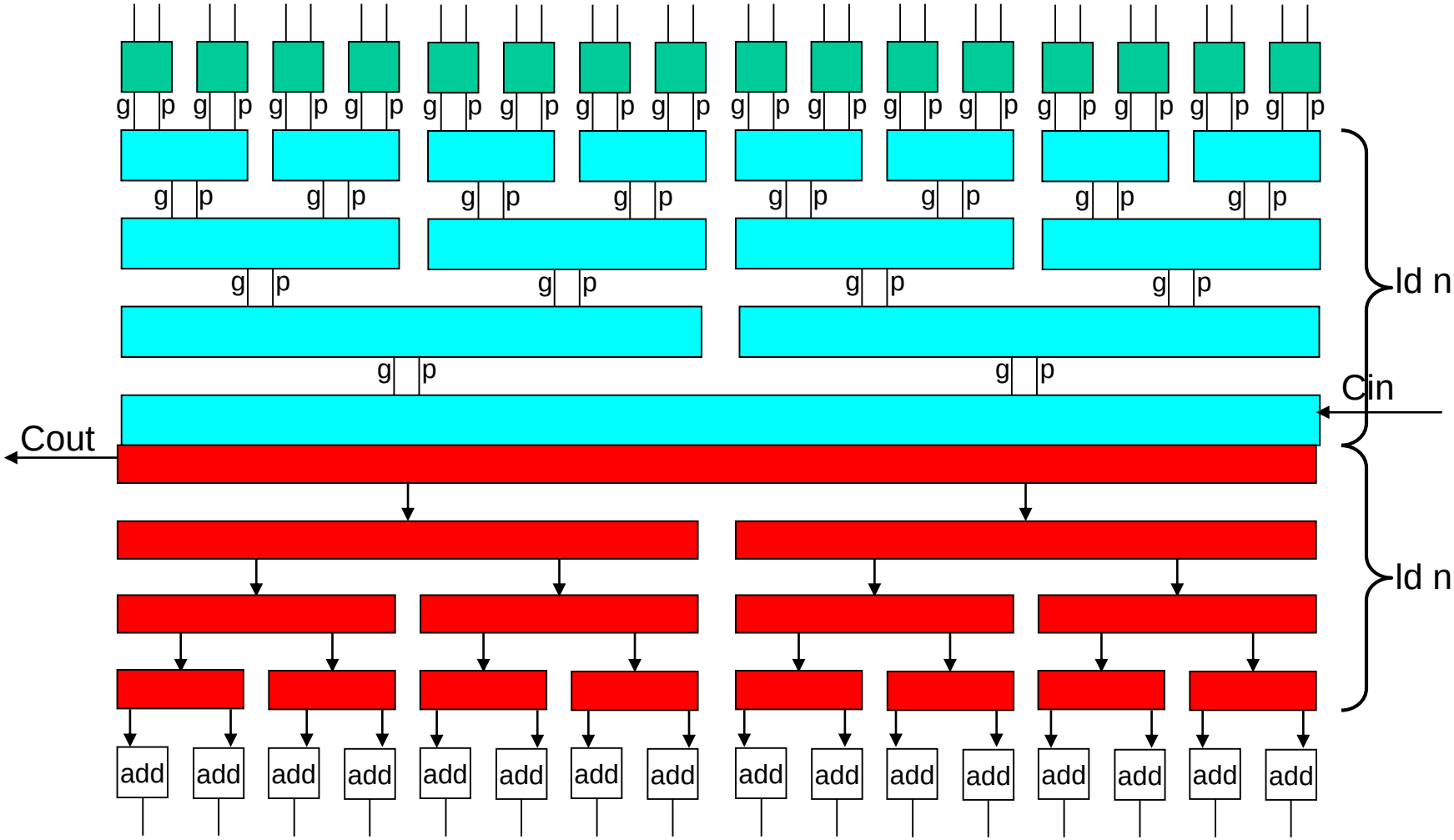
(ii) für $i \leq k < j$ gilt: $p(i,j) = p(k+1,j) \bullet p(i,k)$

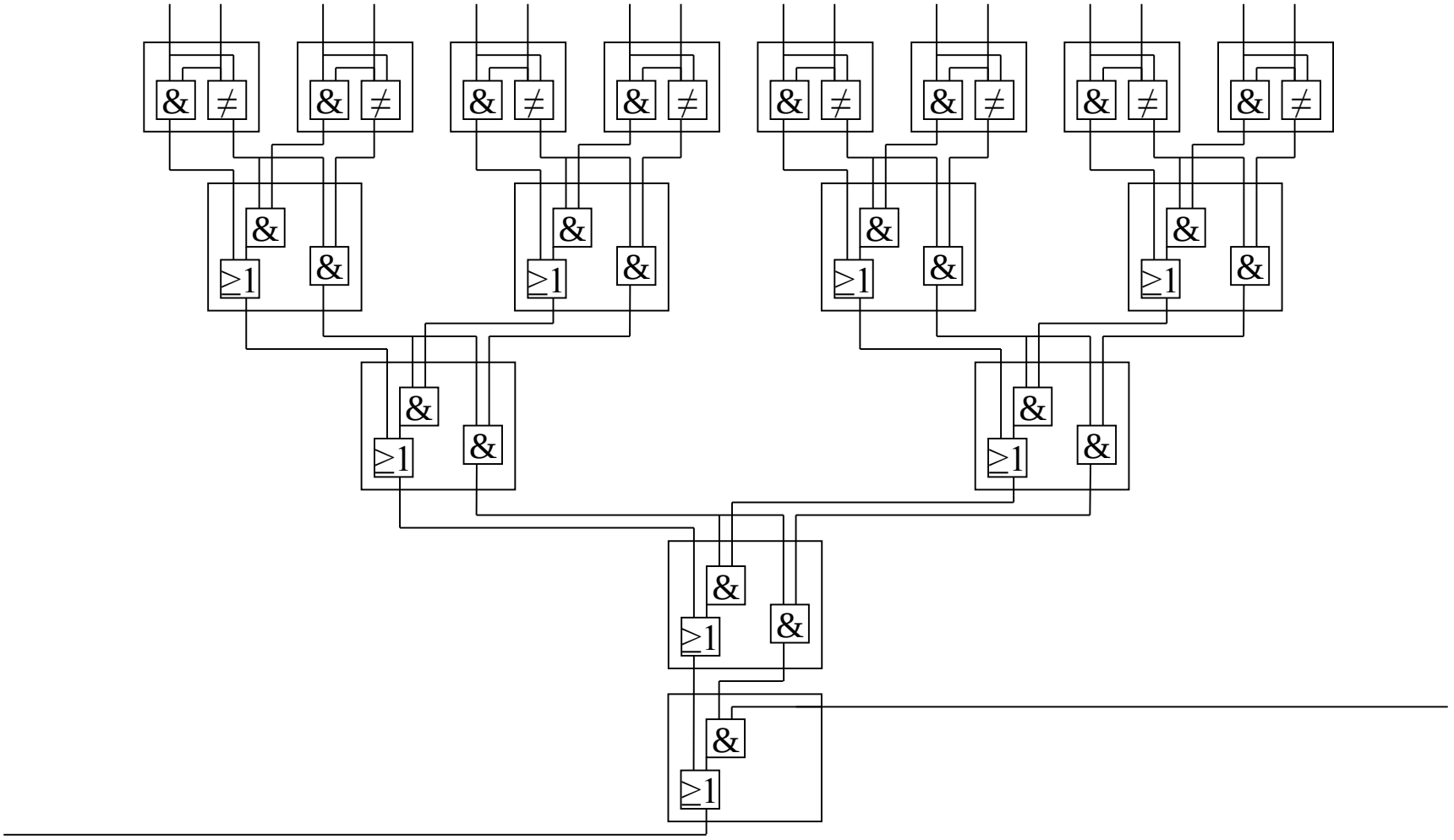
(iii) für $i \leq k < j$ gilt: $g(i,j) = g(k+1,j) + g(i,k) \bullet p(k+1,j)$

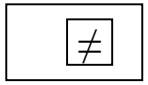
```
{Stufe 1: Berechne g(i,i), p(i,i) }
for i:=0 to n-1 pardo
   p(i,i):=a(i) XOR b(i);
   g(i,i):=a(i)·b(i)
endpardo;

{Stufe 2: Berechne g(i,j), p(i,j) }
for m:=0 to (log n)-1 do
   for k:=0 step 2^(m+1) to n-2^(m+1) pardo
      p(k,k+2^(m+1)-1):= p(k+2^m,k+2^(m+1)-1)·p(k,k+2^m-1);
      g(k,k+2^(m+1)-1):= g(k+2^m,k+2^(m+1)-1)+g(k,k+2^m-1)·p(k+2^m,k+2^(m+1)-1)
   endpardo;

{Stufe 3: Berechne c_i }
c(0):=0;
for m:=(log n)-1 downto 0 do
   for k:=0 step 2^(m+1) to n-2^(m+1) pardo
      c(k+2^m) := g(k,k+2^m-1)+ c(k)·p(k,k+2^m-1)
   endpardo;

{Stufe 4: Berechne s_i }
for i:=0 to n-1 pardo
   s(i) := a(i) XOR b(i) XOR c(i)
endpardo;
```


&
≠
&
≠
&
≥1
&

Der Carry-Lookahead-Addierer




&
≠
≥1


≠

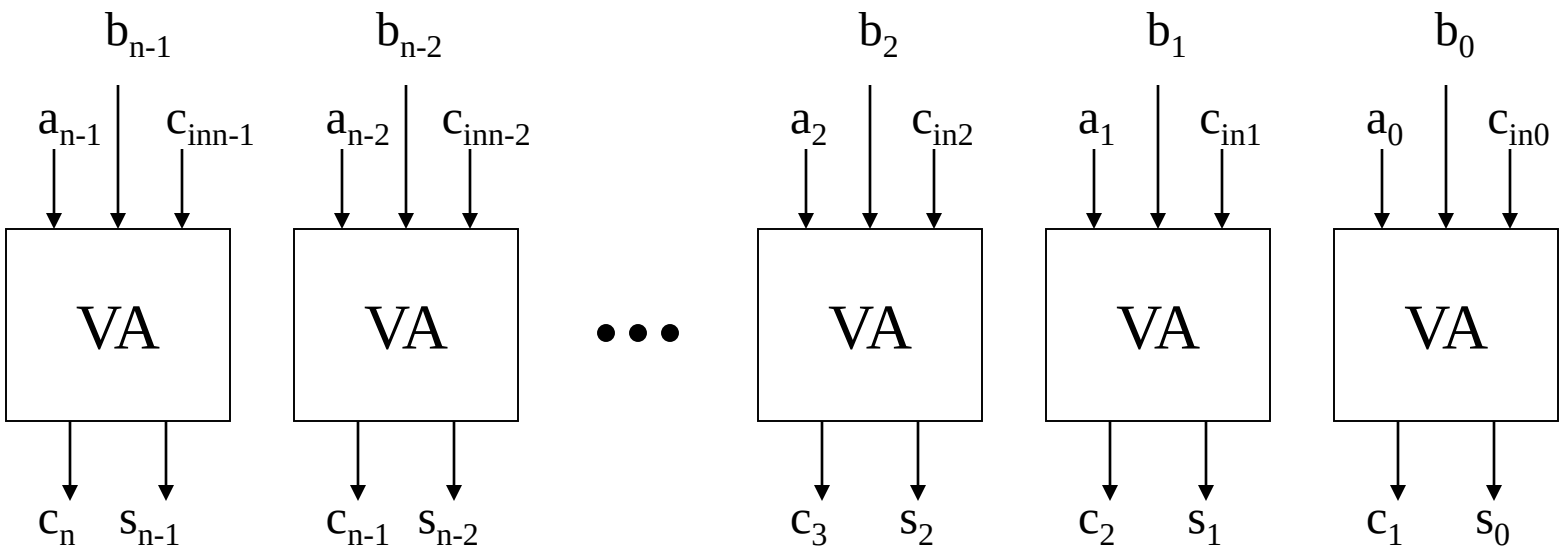
Carry Save Adder

Ein anderer Ansatz ist die Verwendung einer redundanten Zahlendarstellung für die Zwischenergebnisse. Diese Technik stellt sicher, dass ein eventuell auftretendes Carry nur einen Einfluß in seinem unmittelbaren Bereich hat, aber nicht zu einer „Kettenreaktion“ führen kann, wie beim ripple-carry-adder. Der hier vorgestellte Addierer heißt **carry-save-adder**.

Die Idee besteht darin, nicht zwei Operanden zu einem Ergebnis zu addieren, sondern ***drei*** Operanden zu ***zwei*** Ergebnissen. Dies erscheint zwar für unser Verständnis zunächst unnatürlich, es hat aber den Vorteil einer sehr einfachen und schnellen Realisierung. Die Anwendung eines solchen Carry-save-Addierers ist immer dann sinnvoll, wenn man mehrere Additionen nacheinander ausführen möchte. Und dies wiederum ist in Multiplizierern erforderlich. Wir werden daher sehen, wie ein extrem schneller Multiplizierer aus Carry-save-Addierern aufgebaut werden kann.

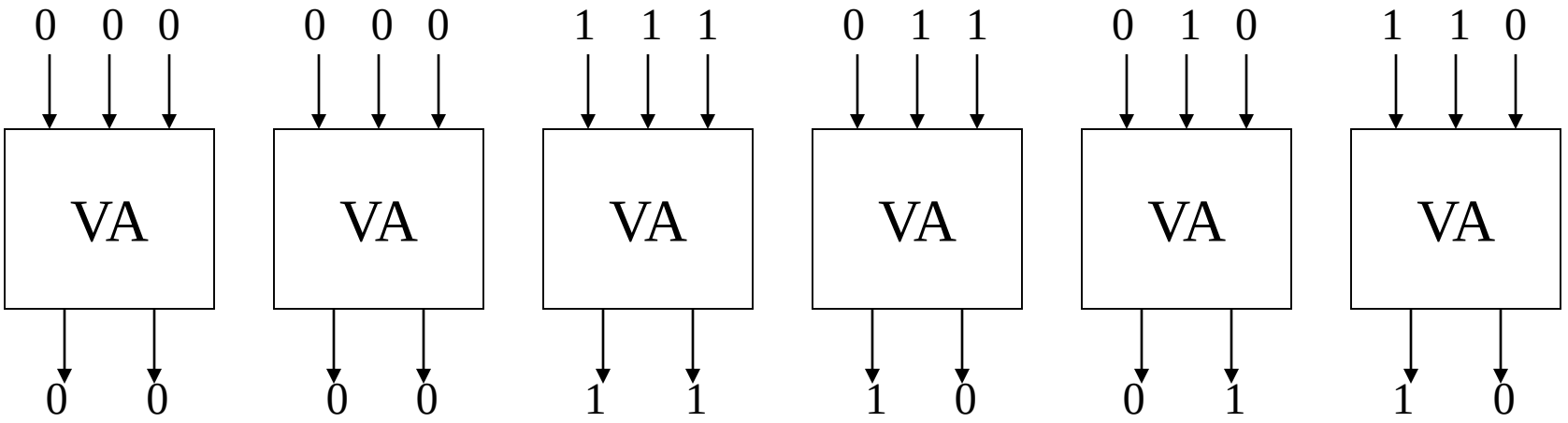
Der Aufbau eines Carry-save-Addierers ist lediglich die Parallelschaltung von n Volladdierern. Diese sind nun nicht verkettet, sondern jeder leifert zwei Ausgabebits, ein s-Bit und ein c-Bit. Aus diesen wird ein s-Wort und ein c-Wort gebildet, die zusammen die beiden Ergebnisworte darstellen. Das c-Wort wird durch eine 0 an der am wenigsten signifikanten Stelle ergänzt und das s-Wort um eine 0 an der höchst signifikanten Stelle.

Somit ist die Summe aus s- und c-Wort gleich der Summe der drei Operandenworte a, b, c_{in}.


b_{n-1}
b_{n-2}
b_2
b_1
b_0
a_{n-1}
c_{inn-1}
a_{n-2}
c_{inn-2}
a_2
c_{in2}
a_1
c_{in1}
a_0
c_{in0}
VA
VA
VA
VA
VA
c_n
s_{n-1}
c_{n-1}
s_{n-2}
c_3
s_2
c_2
s_1
c_1
s_0

Beispiel:

Wir addieren die Worte a = 001001, b = 001111, c_{in} = 001100



Die Ergebnisworte sind also c = 011010 und s = 001010. Wenn wir die drei Operanden im Dezimalsystem addieren, kommt 36 heraus. Dasselbe Ergebnis bekommen wir, wenn wie c und s addieren. Man beachte, daß bei dieser Art der Addition kein Carry „durchklappern“ kann. Die Zeit für eine ist also t_{VA} und somit in O(1).

Wo können wir diese Art von Addition sinnvoll einsetzen?

Angenommen, wir müssen eine Kolonne von 64 Zahlen addieren. Dann können wir diese mit 62 Carry-Save-Additionen zusammenzählen, so dass schließlich zwei Ergebnisworte berechnet werden. Diese müssen dann mit einem „richtigen“ Addierer, z.B. einem Carry-Select-Addierer zu einem Endergebnis zusammengezählt werden. Die 62 Additionen benötigen 62 * t_{VA}. Die letzte Addition benötigt 8 * t_{VA} + 7 t_{MUX}. Der Zeitaufwand istgesamt ist also 70t_{VA} + 7 t_{MUX}. Hätten wir die gesamte Addition mit einem Carry-Select-Addierer gemacht, würden wir zusammen 63 * (8 t_{VA} + 7 t_{MUX}) = 504 t_{VA} + 441 t_{MUX} .

Man sieht, um wieviel sparsamer die Carry-Save-Addition in diesem Falle ist. Allgemein gilt: Wenn wir m Additionen der Länge n Bit machen wollen, benötigen wir mit einem Ripple-Carry-Addierer Zeit O(n*m), mit einem Carry-Select-Addierer Zeit O($n^{1/2}$*m) und mit einem Carry-Save-Addierer (mit nachgeschaltetem Carry-Select-Addierer für den letzten Schritt) Zeit O($n^{1/2}$ +m).

Die optimale Zeit bei der Addition zweier Zahlen erhält man mit einem sogenannten Carry-Lookahead-Addierer. Dieser benötigt nur die Zeit O(log n) für eine Addition. Aus Zeitgründen wird dieser Addierertyp aber an dieser Stelle noch nicht behandelt.

Multiplikation

Bei der Multiplikation nach der Schulmethode wird für jede Stelle des einen Operanden das Produkt dieser Stelle mit dem anderen Operanden berechnet. Danach werden alle diese Produkte addiert. An dieser Stelle können wir den soeben erlernten Carry-Save-Addierer einsetzen, denn jetzt haben wir den Fall einer großen Anzahl von Operanden, die addiert werden müssen.

Beispiel:

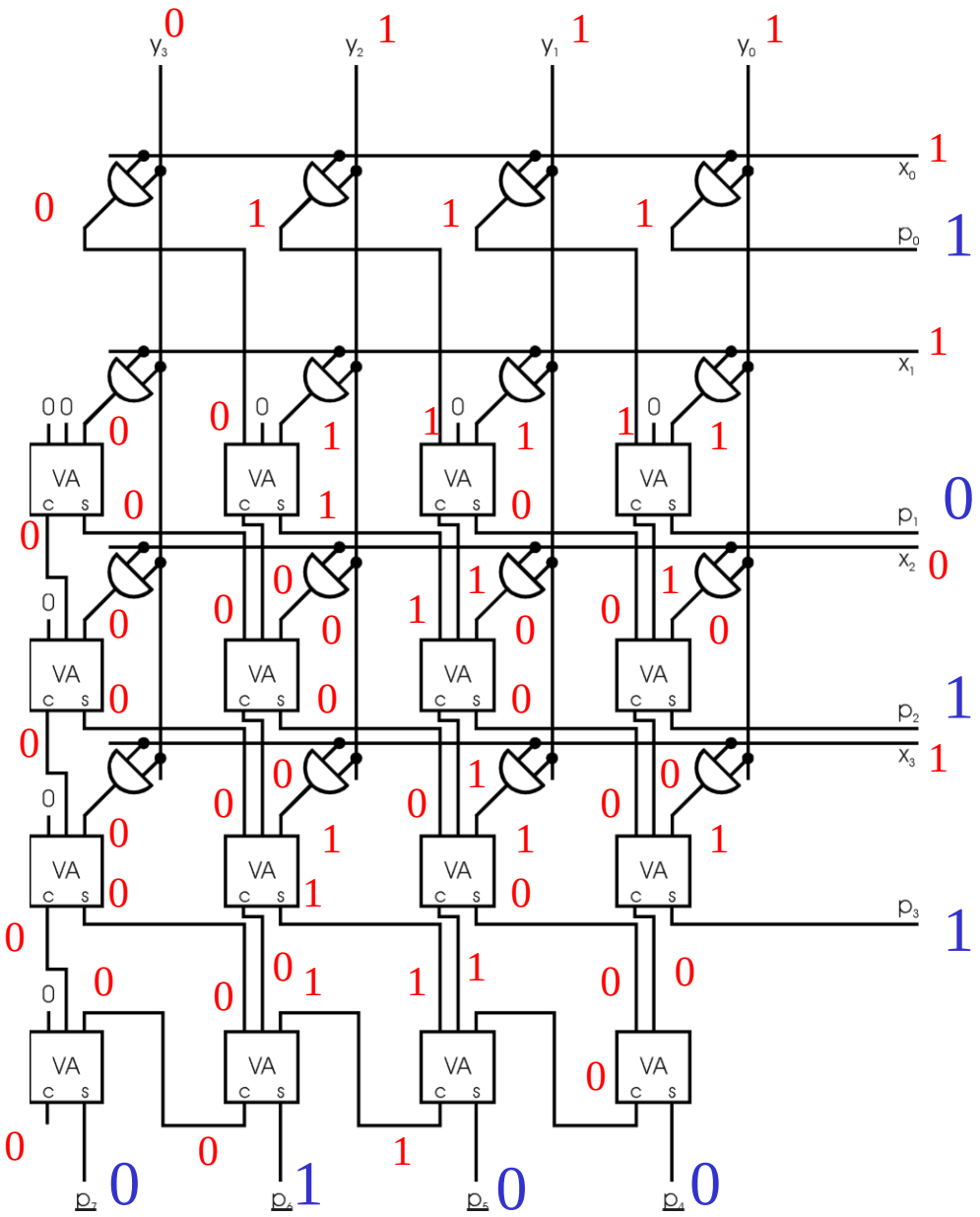
```
10110011 * 10010111
    10110011
     00000000
      00000000
       10110011
        00000000
         10110011
          10110011
           10110011
01101001100101 01
```

Ein Carry-Save-Multiplizierer für zwei Operanden der Länge n besteht aus n^2 AND-Gattern, die gleichzeitig alle erforderlichen 1-Bit-Multiplikationen ausführen (Die binäre Multiplikation von 1-Bit Zahlen ist gerade das logische „UND“). Danach sind nur noch alle Teilprodukte (wie bei der Schulmethode) zu addieren. Dies geschieht nun in der bekannten 3-auf-2 Operanden Manier, die wir soeben beim Carry-Save-Addierer kennengelernt haben. Am Ende ist für die höchstsignifikanten n Bits noch eine (klassische) 2-auf-1-Operanden-Addition erforderlich. Diese wird mit einem konventionellen Addierer ausgeführt.

Wie lang ist die Verarbeitungszeit für eine solche Multiplikation? Wir müssen in diesem Schaltnetz den „kritischen Pfad“ suchen, also den Pfad, bei dem ein Signal durch die maximale Anzahl von Schaltelementen hindurchwandern muß, bevor das Endergebnis berechnet ist. Dieser Pfad besteht zunächt einmal aus den n-2 Stufen, bei denen jeweils ein Operand neu hinzuaddiert wird plus die n-1 Volladdierer, durch die ein Carry bei der letzten Addition hindurchklappern muß (wir setzen hier einen ripple-carry-adder voraus).

Ein Beispiel für einen solchen 4-Bit Multiplizierer sehen wir auf der nächsten Folie.

4-Bit-carry-save-Multiplizierer



Division

Einige Prozessoren haben eigene Divisionseinheiten, die in der Regel ähnlich des Carry-Save-Adders eine interne Darstellung der Zwischenergebnisse durch zwei Worte benutzt.

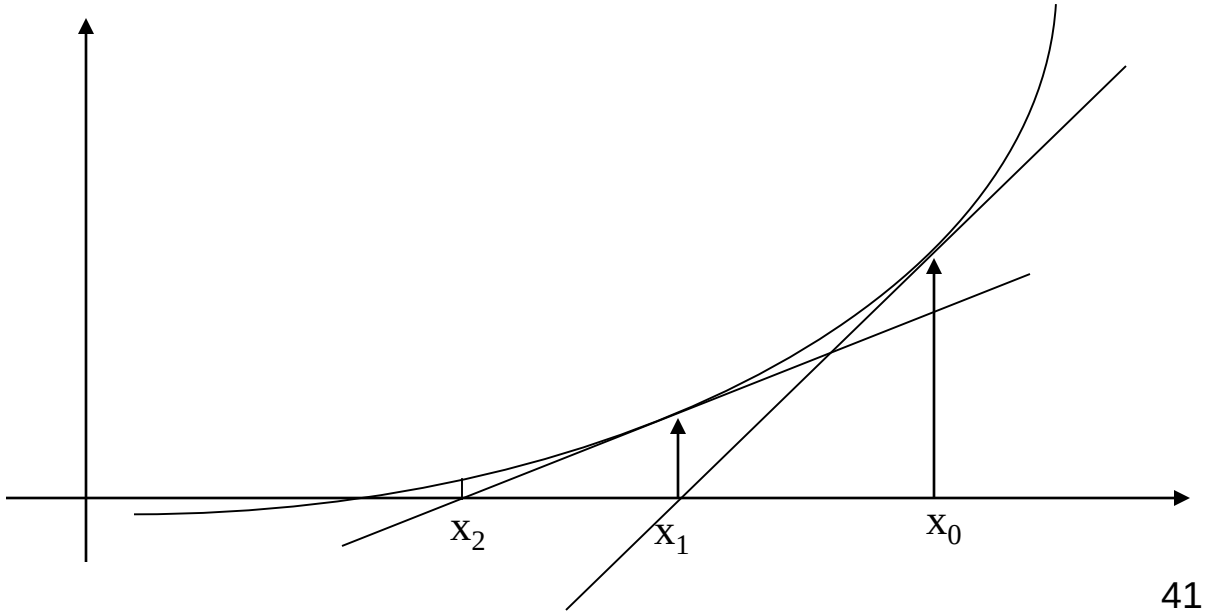
Andererseits ist die Division eine seltene Operation. Daher (make the common case fast) verzichten viele Prozessoren auf eigene Hardware für die Division, sondern implementieren sie in Software. Ein gängiges Verfahren dafür ist die Newton-Raphson-Methode, die wir hier kennenlernen wollen.

Vorweg sei erwähnt, daß frühere Rechner (< 1980) die Division in der Regel (ebenfalls in Software) entsprechend der Schulmethode ausführten, d.h. die Ergebnisbits werden eins nach dem anderen berechnet durch Vergleich des Divisors mit den verbleibenden höchstsignifikanten Stellen des Dividenden. Wenn der Divisor größer ist, ergibt sich ein Bit 0 sonst ein Bit 1. Im letzteren Falle wird sodann der Divisor von den höchstsignifikanten Stellen des Dividenden subtrahiert und eine weitere Stelle des Dividenden wird für die nächste Ergebnisstelle herangezogen.

Dieses Verfahren ist natürlich in O(n), wenn der Dividend n Stellen hat. Genauer: Man braucht n Schritte, und in jedem Schritt muß ein Vergleich und eine Subtraktion ausgeführt werden. Das erwies sich zu Zeiten steigender Rechenleistung als zu langsam. Daher suchte man nach Verfahren, die in weniger Schritten zu genauen Ergebnissen führten.

Die Idee des Newton-Verfahrens ist die Approximation der Nullstelle einer Funktion durch Konstruktion einer Folge von Werten, die sehr schnell gegen die Nullstelle konvergiert. Man beginnt damit, daß man die Tangente an die Funktion in einem geschätzen Anfangswert anlegt und deren Schnittpunkt mit der Abszisse als nächsten Folgenwert berechnet. Nun legt man an dessen Funktionswert die Tangente an usw. Wenn die Funktion bestimmte Bedingungen erfüllt und wenn der Anfangswert geeignet gewählt ist, konvergiert diese Folge gegen die Nullstelle.

Beispiel:



Für zwei Werte x_i und x_{i+1} in dieser Folge gilt:

$$f'(x_i) = \frac{f(x_i)}{x_i - x_{i+1}}$$

Aufgelöst nach x_{i+1} bedeutet das

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Betrachten wir nun f(x) = 1/x - B . Diese Funktion hat in 1/B eine Nullstelle. Wenn wir in die obige Formel einsetzen, ergibt sich

$$x_{i+1} = x_i - \frac{\frac{1}{x_i} - B}{-\frac{1}{x_i^2}} = 2x_i - Bx_i^2$$

Damit haben wir eine sehr einfache Iterationsformel, die mit zwei Multiplikationen und einer Subtraktion für einen Iterationsschritt auskommt.

Wieviele Schritte benötigen wir aber, oder anders gefragt, wie schnell konvergiert die Folge?

Betrachten wir den Fehler ε, also die Differenz des Folgenwertes x_i von der gesuchten Nullstelle 1/B. Es gilt:

$$x_{i+1} = 2x_i - Bx_i^2 = 2\left(\frac{1}{B} - \varepsilon\right) - B \cdot \left(\frac{1}{B} - \varepsilon\right)^2 = \frac{1}{B} - B \cdot \varepsilon^2$$

Somit ist der Fehler nach der nächsten Iteration nur noch $B\varepsilon^2$. Wenn B nun zwischen 0 und 1 liegt, bedeutet dies, dass wir eine quadratische Konvergenz haben, genauer: wenn das Ergebnis nach der i-ten Iteration bereits auf m Bits genau ist, ist es nach der i+1-ten Iteration auf 2m Bits genau.

Wir müssen also dafür sorgen, dass B zwischen 0 und 1 liegt und das x_0 auf 1 Bit genau ist. Dann wird x_1 auf zwei Bits genau sein, x_2 auf vier Bits usw., x_i auf 2^i Bits genau.

Angenommen, wir müssen a:b berechnen. Dann können wir dies mit einer Multiplikation als a * 1/b berechnen, wobei wir in der Lage sein müssen, den Kehrwert der Zahl b also 1/b zu ermitteln. Wenn b zwischen 0 und 1 ist und die erste Stelle nach dem Komma eine 1 ist (also normalisiert), dann geht das mit obiger Iteration. Wenn nicht, müssen wir $B = 2^k * b$ nehmen mit einem geeigneten k, so dass B normalisiert ist. Sodann berechnen wir 1/B und multiplizieren dies schließlich mit 2^{-k} (Bitverschiebung).

Fazit: Durch die Iteration wird der Aufwand von 2n Operationen auf 3log n Operationen reduziert, nämlich log n Iterationen und in jeder drei Operationen. Bei einer 64-Bit Division ist das eine Reduktion von 128 auf 18 Operationen.