PARALLELMULTIPLIZIERER:
schnelle, platzeffiziente Algorithmen -
VLSI-gerechte Realisierungen

Bernd Becker

Techn. Bericht [A] 12/82

Fachbereich 1o
Angewandte Mathematik und Informatik
Universität des Saarlandes

66 Saarbrücken
WEST GERMANY

PARALLELMULTIPLIZIERER:

schnelle, platzeffiziente Algorithmen - VLSI-gerechte Realisierungen

Bernd Becker)*
Universitaet des Saarlandes
Fachbereich 10
66 Saarbruecken

Abstract:

Die Entwicklung eines 32-bit Multiplizierchips (fuer Integer-Zahlen dargestellt im 2-Komplement) ist Ausgangs- und Zielpunkt der hier angestellten Ueberlegungen.
Zuerst gehen wir kurz auf den theoretischen Hintergrund ein und geben dann 4 Algorithmen an, in denen die wichtigsten Methoden zum Parallelmultiplizieren exemplarisch vorgestellt werden. Im einzelnen sind dies:

1) Matrix-Multiplizierer
2) Iteratives Array
3) Modifizierter Booth-Algorithmus mit Wallace-Tree
4) Redundante Zahlendarstellung und binaerer Baum

Wir versuchen bei allen Algorithmen, theoretische Guete und praktische Qualitaet gegenueberzustellen und daraus am Ende ein Fazit fuer die konkrete Aufgabe (32-bit Multiplizierer) zu ziehen.

)* Dieser Bericht entstand waehrend eines Aufenthalts im September und Oktober 1982 bei der Siemens AG, ZT ZTI SYS 2, in Muenchen.

I) ZIEL

VLSI-Entwurf eines 32-bit Multiplizierwerkes,
die Operanden sind Integerzahlen,
die im 2-Komplement dargestellt sind.

II) THEORETISCHER HINTERGRUND

Die theoretische Komplexitaet eines n-bit Multiplizierwerks wird allgemein durch das Produkt aus der belegten Flaeche A und dem Quadrat der verbrauchten Zeit T bestimmt. (Das Mass AT**2 beruecksichtigt also Zeit staerker als Flaeche, dies entspricht der Tendenz eher Flaeche zu spendieren, um Zeit zu sparen.)

Als untere Schranke fuer Multiplikation gilt nach [BrKu]
AT**2 >= c.n**2 mit c positive Konstante. Diese Schranke ist scharf, wie in [PrVu] gezeigt wird. Allerdings ist der Algorithmus fuer kleine n (n<=2000) nicht geeignet.

Wir werden im folgenden bei den von uns untersuchten Verfahren jeweils die Komplexitaeten A, T, AT**2 untersuchen, aber auch gleichzeitig die Konstanten betrachten, d.h. Ueberlegungen ueber die "reale" Groesse und Laufzeit anstellen.

Eine uebersichtliche Zusammenfassung der bisher bekannten Algorithmen und ihrer bisherigen Anwendungsmoeglichkeiten findet man in [Sp]. Der klassische sequentielle Multiplizieralgorithmus ist auch bekannt als der serielle Shift-Algorithmus. Wir betrachten als erstes Verfahren eine parallele Version dieses Algorithmus', die wir mir mit Matrix-Multiplizierer bezeichnen. Diese Architektur laesst sich relativ leicht ins Bit-Slice Konzept einpassen und damit VLSI-gerecht entwerfen. Wir arbeiten diese Version bis zum Entwurf der Einzelzellen detailliert aus.

Als zweites wird ein Multiplikationsverfahren mit Hilfe eines iterativen Arrays angedeutet. (Das Konzept der iterativen Arrays wird hier auch deshalb erwaehnt, weil es sich gut ins Bit-Slice Konzept einpassen laesst.)

Zu den schnellsten bisher benutzten Algorithmen gehoert die dritte betrachtete Version, ein als Wallace-Tree organisierter modifizierter Booth-Algorithmus. Wir gehen nicht ins Detail, sondern verweisen auf [ReKuMa] und deuten die Schwierigkeiten bei einer VLSI-Realisierung an.

Das letzte hier vorgestellte Verfahren ist bisher unbekannt und wurde von Herrn Prof. K.Mehlhorn, Saarbruecken, und dem Autor konzipiert. Es arbeitet mit Baumstrukturen und benutzt redundante Zahlendarstellung. Auf eine (moegliche?) VLSI-Realisierung wird kurz eingegangen.

Die oben angesprochenen Verfahren bieten einen repraesentativen Querschnitt der wichtigsten bekannten Methoden beim Parallelmultiplizieren und koennen so als angemessene Beispiele bei Ueberlegungen zum Verhaeltnis "schnelle, platzeffiziente Algorithmen - VLSI-gerechte Realisierungen" gelten.

III) MATRIX-MULTIPLIZIERER

Ab jetzt gehen wir von folgenden Bezeichnungen aus:
a = a[n-1] a[n-2] ... a[0] ("Multiplikand"),
b = b[n-1] b[n-2] ... b[0] ("Multiplikator")
mit a[i],b[i] ∈ {0,1}
sind Integer-Zahlen im 2-Komplement.
Boole'sches "Or" bezeichnen wir mit "|",
"And" mit "&",
"Exor" mit "⊕",
"Not" mit "'".

Die Matrix-Multiplikation entsteht aus Anwendung der Schulmethode: zu addieren sind die n Zahlen

```
                          a[n-1]&b[0] ... a[1]&b[0] a[0]&b[0]
              a[n-1]&b[1]       ...       a[o]&b[1]     0

                                                        .
                                                        .
                                                        .
a[n-1]&b[n-1]   ...    a[0]&b[n-1] 0      ...          0
```

Wir geben zunaechst eine informale Beschreibung des Verfahrens (es gliedert sich in 3 Phasen):

1) Man denke sich die n Summanden als quadratische Matrix aufgeschrieben. Die n Summanden werden sukzessive mit Hilfe von je einer Kette von n Fulladdern addiert. Die entstehenden Summen und Uebertraege werden jeweils folgendermassen zur naechsten Zeile der Matrix weitergeleitet: das Summenbit nach rechts unten, das Uebertragsbit senkrecht nach unten. Dadurch wird die unterschiedliche Stelligkeit der Summanden zueinander beruecksichtigt. Der dritte noch freie Eingang eines jeden Fulladders in der naechsten Zeile wird dazu benutzt, den naechsten Summanden aufzusummieren.
   In jeder Zeile liefert der am weitesten rechts stehende Full-Fulladder ein Bit des Ergebnisses. Man erhaelt also auf der rechten Seite der Matrix die n lower bits des Ergebnisses der Multiplikation. Aus der letzten Zeile werden je n Summen- und Uebertragsbits geliefert.
   Diese Matrix bildet den Kern des Algorithmus , wir nennen sie MULTIPLIZIERMATRIX.

2) Danach schliessen sich zwei Fulladderzeilen an, die die Vorzeichenkorrektur erledigen. Man rechnet leicht nach, dass bei Darstellung im 2-Komplement folgendes gilt:

   sei c := c[n-1] ... c[1] c[0],

   $$w(c) := \sum_{i=0}^{n-2} c[i].2^{**}i - c[n-1].2^{**}(n-1)$$

   heisst zu c gehoerige 2-K-Zahl,

   $$w0(c) := \sum_{i=0}^{n-1} c[i].2^{**}i$$

   heisst zu c gehoerige Dualzahl,

   $$w1(c) := \sum_{i=0}^{n-2} c[i].2^{**}i$$

   heisst zu c gehoeriger Betrag,

dann ist

w(a)*w(b) = w0(a)*w0(b) - 2**n(b[n-1]*w1(a)+a[n-1]*w1(b)).

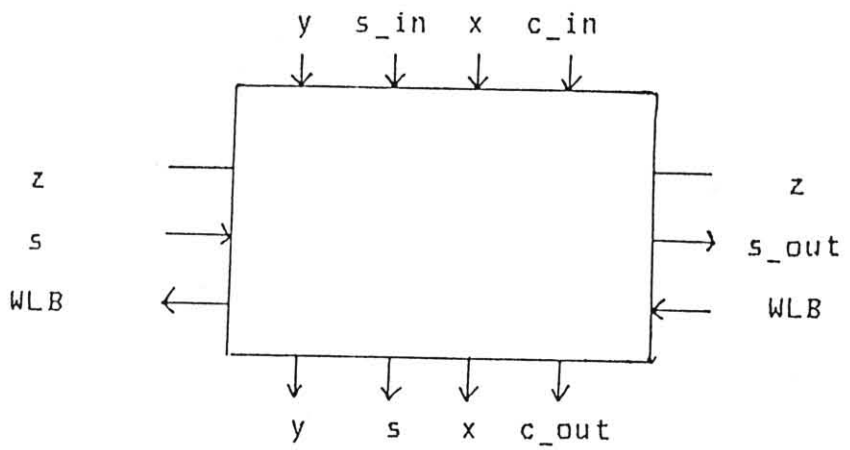
Wir koennen das vorzeichenkorrigierte Ergebnis also so berechnen:

Die Multipliziermatrix berechnet (in durch Summe und Uebertrag dargestellter Form) w0(a)*w0(b).
Im Fall a[n-1]=1,
b[n-1]=0 (d.h. a negativ, b positiv)
addieren wir b'+1 (b' sei das bitweise Komplement von b),
im Fall a[n-1]=0,
b[n-1]=1 (d.h. a positiv, b negativ)
addieren wir a'+1,
im Fall a[n-1]=b[n-1]=1 (d.h. a,b negativ)
addieren wir a'+b'+1+1.

Wir realisieren dies durch zwei weitere Zeilen von Zellen, die wieder im wesentlichen aus Fulladdern bestehen. Die Uebertraege werden nach links unten, die Summen senkrecht nach unten weitergeleitet. Die Uebertraege der am weitesten links stehenden Falladder bleiben unberuecksichtigt. (Das Ergebnis der Multiplikation zweier n-bit Zahlen ist immer als 2*n-bit Zahl darstellbar!)
Als Ergebnis erhalten wir je n Summen- und Uebertragsbits.
Wir nennen diese Stufe VORZEICHENKORREKTUR.

3) In einer letzten Stufe, der ADDIERSTUFE, werden diese n Uebertrags- und Summenbits durch einen n-bit Addierer zu den n higher bits des Ergebnisses umgewandelt.

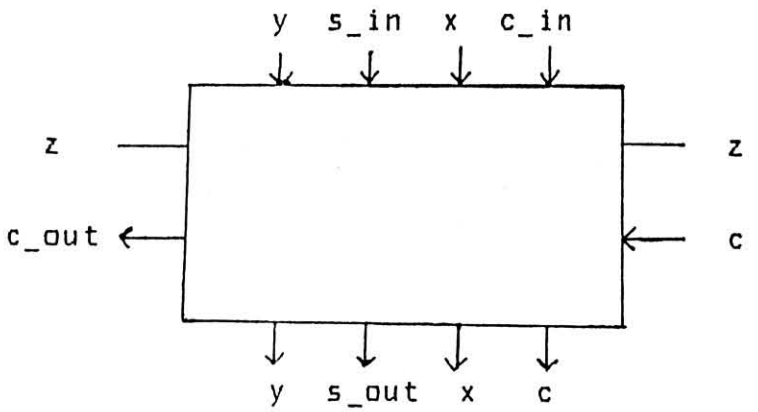
Wir geben jetzt die genaue Beschreibung des Matrix-Multiplizierers, indem wir die Zelltypen exakt funktional beschreiben und ihre Einfuegung ins Bit-Slice Konzept angeben.
Die Multipliziermatrix besteht aus zwei Zelltypen Mf, Mfd.
Mfd steht in der Diagonalen von links unten nach rechts oben, sonst steht ueberall Mf.
Die Zellen haben folgendes Ausehen:



Fuer Mf gilt: s_out = x&z ⊕ s_in ⊕ c_in,
c_out = x&z&(s_in⊕c_in) ∣ s_in&c_in.

Fuer Mfd gilt: y = z,
s_out = x&z ⊕ s_in ⊕ c_in,
c_out = x&z&(s_in⊕c_in) ∣ s_in&c_in.

Die Vorzeichenkorrektur setzt sich aus sechs Zelltypen zusammen.
Die erste Zeile besteht aus Zellen des Typs Ma, einer rechten Randzelle Mac und einer linken Randzelle Mav.
Die zweite Zeile besteht aus Zellen des Typs Mb, einer rechten Randzelle Mbc und einer linken Randzelle Mbv.
Die Zellen haben folgendes Aussehen:



Fuer Ma gilt: $s\_out = z\&x' \oplus s\_in \oplus c\_in$,
$c\_out = z\&x'\&(s\_in \oplus c\_in) \mid s\_in\&c\_in$.

Fuer Mav gilt: $y = z$,
$s\_out = z\&x' \oplus s\_in \oplus c\_in$,
$c\_out = z\&x'\&(s\_in \oplus c\_in) \mid s\_in\&c\_in$.

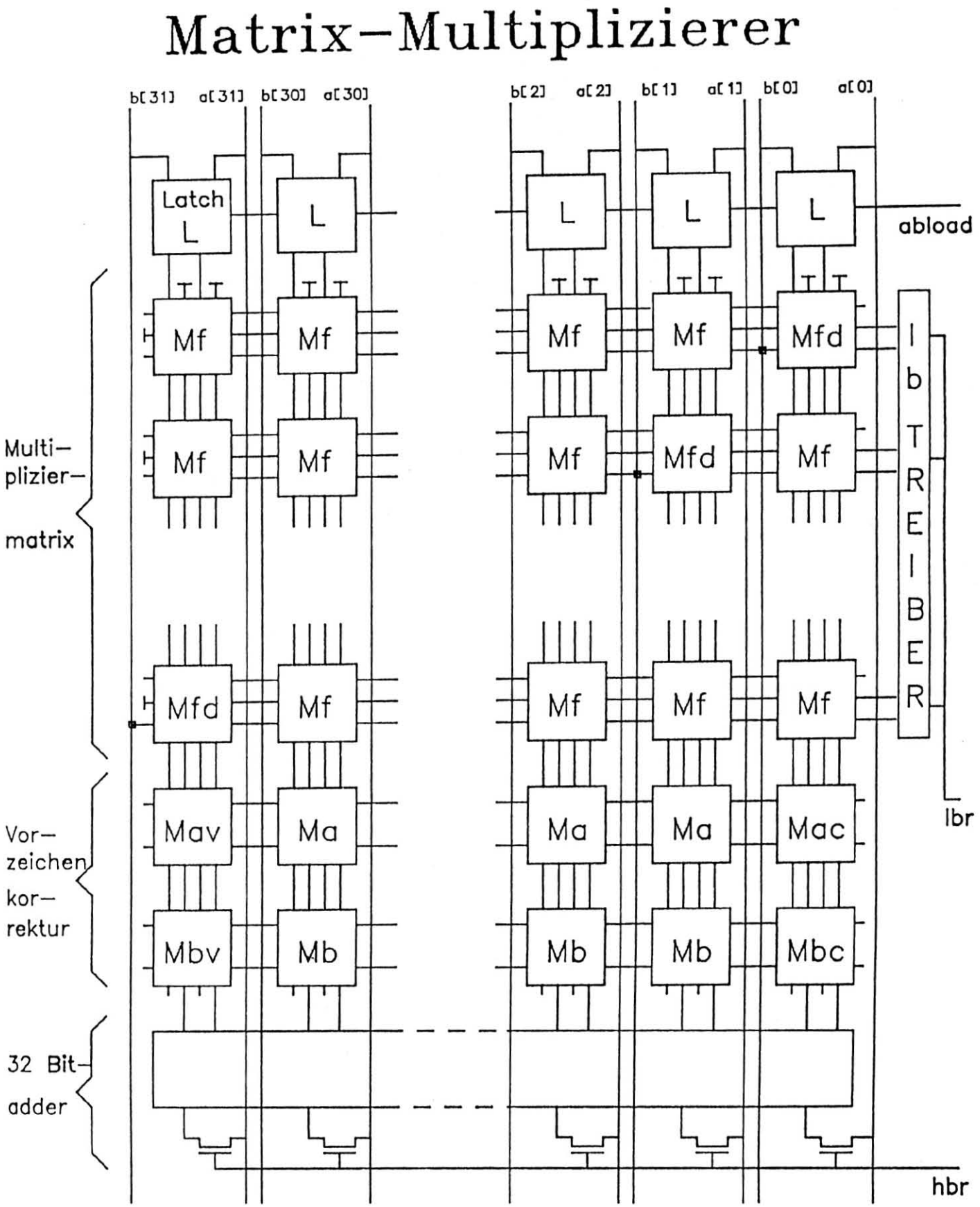
Fuer Mac gilt: $z = c$,
$s\_out = z\&x' \oplus s\_in \oplus c\_in$,
$c\_out = z\&x'\&(s\_in \oplus c\_in) \mid s\_in\&c\_in$.

Fuer Mb gilt: $s\_out = z\&y' \oplus s\_in \oplus c\_in$,
$c\_out = z\&y'\&(s\_in \oplus c\_in) \mid s\_in\&c\_in$.

Fuer Mbv gilt: $x = z$,
$s\_out = z\&y' \oplus s\_in \oplus c\_in$,
$c\_out = z\&y'\&(s\_in \oplus c\_in) \mid s\_in\&c\_in$.

Fuer Mbc gilt: $z = c$,
$s\_out = z\&y' \oplus s\_in \oplus c\_in$,
$c\_out = z\&y'\&(s\_in \oplus c\_in) \mid s\_in\&c\_in$.

Der gesamte Multiplizierer hat folgendes Aussehen:



Zur Erlaeuterung:

Die Operanden a und b werden auf dem a-, b-Bus zur Verfuegung gestellt und in ein Latch gelesen. Von dort werden sie mit Hilfe von abload in die Matrix eingegeben.
Die lower bits werden durch lbr auf den b-Bus geschrieben.
Nach der Vorzeichenkorrektur liefert der Addierer die higher bits, die durch hbr auf den a-Bus geschrieben werden.

Die Korrektheit des Algorithmus ist einsichtig.
Zudem wurde das Verfahren durch folgende Simulationen getestet:

CAP-Simulation	von Herrn R.Kolla,
Verdipus-Simulation	von Herrn S.Naeher,
	beide Uni. Saarbruecken.

Genauere Informationen dazu und die notwendigen Listings sind im Anhang zu finden.

Abschliessend einige Bemerkungen zur Komplexitaet dieses Verfahrens. Man sieht sofort, dass folgendes gilt:

A = O(n**2),
T = O(n),
AT**2 = O(n**4),

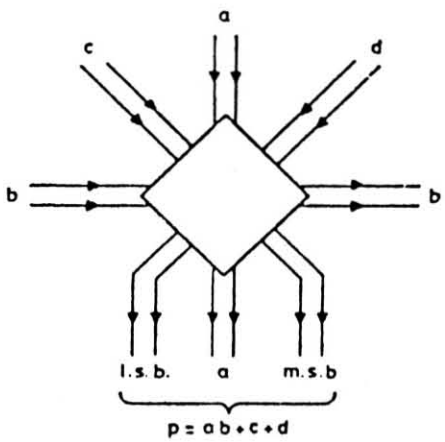
(d.h. das Verfahren ist vom theoretischen Standpunkt weit vom Optimum entfernt).

Im Fall n=32 liefert obige Realisierung
fuer die Flaeche: im wesentlichen 34*32 modifizierte Fulladder (=FA) und einen 32-bit Addierer,
fuer die Verzoegerungszeit: im wesentlichen 34 FA-delays und ein 32-bit Adder-delay.
(Genauere Zeit- und Platzwerte sind erst nach einem noch ausstehenden Design der Zelltypen moeglich.)

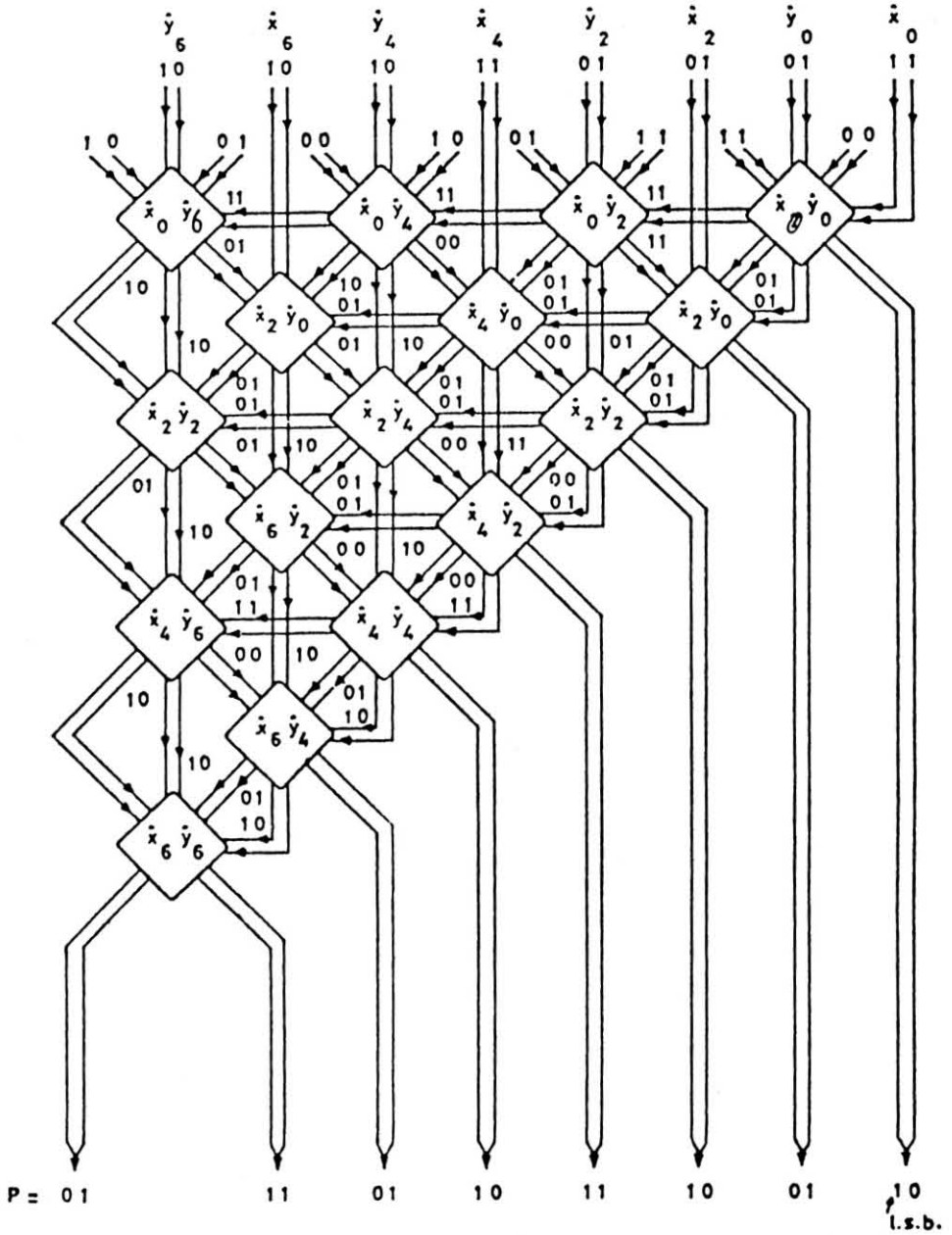
IV) ITERATIVES ARRAY

Wir kommen zur zweiten Methode, bei der eine Multiplikation mit Hilfe eines iterativen Arrays durchgefuehrt wird.
Vorliegende Bemerkungen stuetzen sich, ohne genauer ins Detail zu gehen, auf eine Arbeit von Dormido ([Do]).

Grundlage des Arrays ist die folgende Zelle:
(a,b,c,d,lsb,msb sind hier 2-bit Zahlen.)



Die Multiplikation wird dann von folgendem array (n=8) erledigt:



Zur Komplexitaet laesst sich folgendes sagen:

A = O(n**2),
T = O(n),
AT**2 = O(n**4).

Bei einer 32-bit Realisierung sind 16*16 Einzelzellen notwendig. Entscheidend fuer die Effektivitaet ist eine guenstige Realisierung der Grundzelle. Die Grundzelle ist so komplex, dass eine Realisierung mit weniger als 3 FA-delays nicht zu erwarten ist. Damit ist eine Verzoegerungszeit von mehr als 3*31 FA-delays nicht zu verhindern. Dabei ist dann eine Vorzeichenbehandlung noch nicht eingeschlossen, d.h. bei vergleichbarem Platzbedarf (eine Grundzelle benoetigt mindestens den Platz von 4 FA) ist ein deutlich schlechteres Zeitverhalten als beim Matrixmultiplizierer festzustellen.

V) MODIFIZIERTER BOOTH-ALGORITHMUS MIT WALLACE-TREE

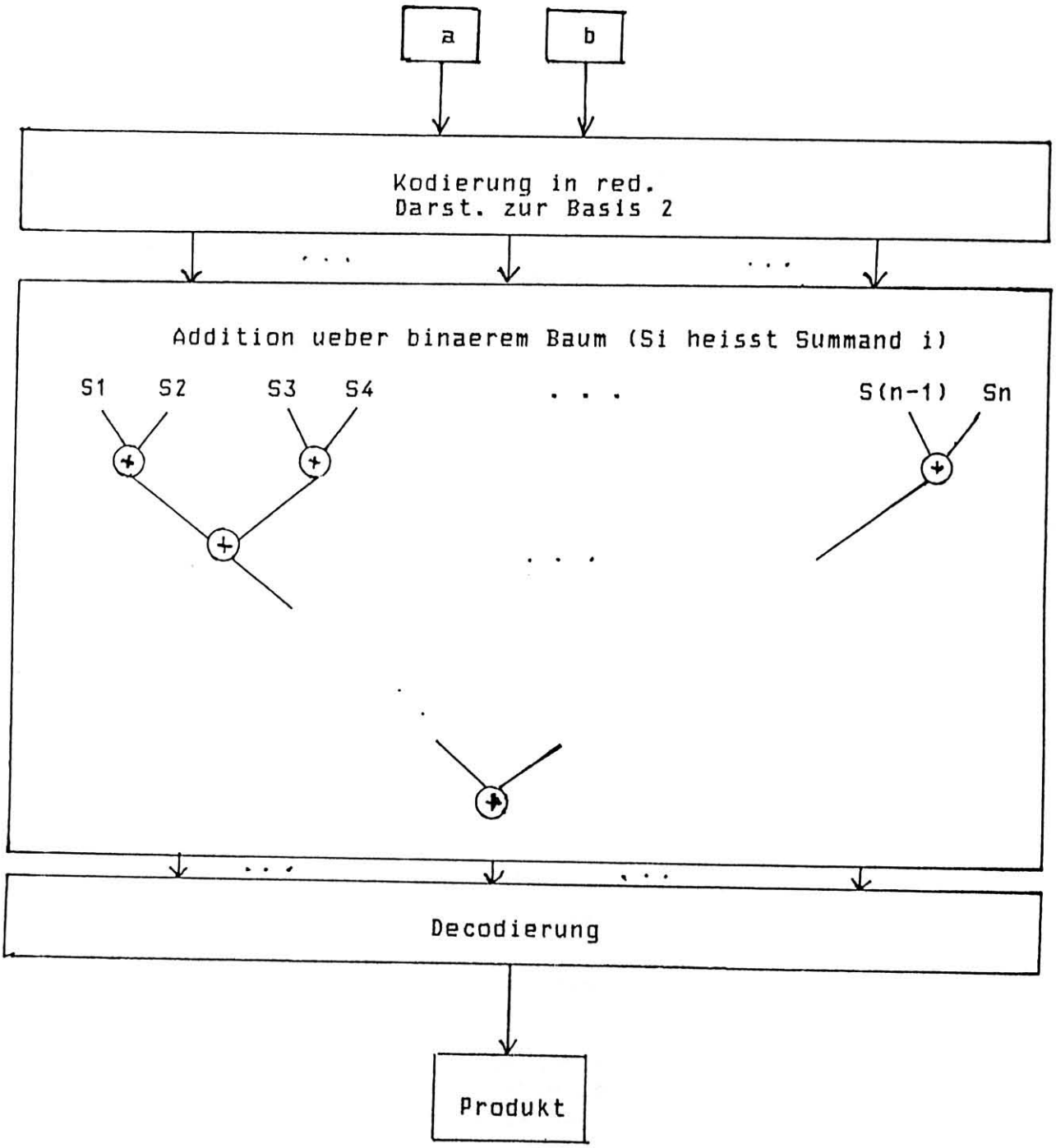
Im modifizierten Booth-Algorithmus wird die Addition von n Summanden bei einer n-bit Multiplikation durch die Addition von n/2 Summanden ersetzt, indem in Abhaengigkeit von je zwei (!) Bits des Multiplikators b die Werte 2*a,-2*a,0,-a oder a (unter Beruecksichtigung der Stelligkeit) als Summanden aufgenommen werden. Auf eine genaue Beschreibung des Verfahrens und den Beweis der Korrektheit wird hier verzichtet. Man findet beides in [GrReNi].
Die Addition der Summanden erfolgt dann mit einem Wallace-Tree, d.h. die Addierstufen werden baumartig angeordnet (zur exakten Definition siehe [Sp]).
In [ReKuMa] wird die oben beschriebene Methode auf einen 24-bit Multiplizierer angewendet. Die dabei auftretenden Probleme treffen natuerlich auch auf den Fall n=32 zu:
Der Platzbedarf steigt im Vergleich zur Version 1 um mehr als das doppelte, das liegt zum Teil auch an Routing-Problemen, die zwischen den einzelnen Adderstufen auftauchen (siehe [ReKuMa]) und bei den heutigen Technologien nicht mit vernuenftigem Aufwand zu loesen sind. Zudem wird der Platzbedarf durch die Berechnung von 2*a,-2*a,-a erhoeht. Vom Standpunkt der theoretischen Komplexitaet gilt folgendes:

A = O(n**2*log(n)),
T = O(log(n)),
AT**2 = O(n**2*log(n)**3).

Das Verfahren liefert also guenstige theoretische Werte, bei einer konkreten Realisierung (n=32) tauchen aber grosse Probleme auf, die vor allem im vergroesserten Platzbedarf und der starken Irregularitaet liegen.

VI) REDUNDANTE ZAHLENDARSTELLUNG UND BINAERER BAUM

Zunaechst geben wir einen kurzen Ueberblick ueber das Verfahren: Die zwei Operanden a und b werden im 2-Komplement angeliefert. Danach werden beide Operanden in redundanter Zahlendarstellung zur Basis 2 interpretiert ("Kodierung"). Im folgenden werden wie bei der Schulmethode die n Summanden kreiert. Diese werden nun in einem binaeren Baum aufsummiert. Da es sich um Zahlen in redundanter Darstellung handelt, kann eine Addition in konstanter Zeit (unabhaengig von der Laenge der Summanden!) ausgefuehrt werden. Am Ende liegt die Summe in redundanter Form vor, wir schliessen noch eine Dekodierung an, nach der dann das Produkt von a und b im 2-Komplement vorliegt. Wir fassen das Prinzip in folgender Skizze zusammen:



Als naechstes wird die Zahlendarstellung genauer erlaeutert.

1) Redundante Zahlendarstellung zur Basis 2

Eine Zahl a = a[n-1] a[n-2] ... a[1] a[0] heisst REDUNDANTE ZAHL zur Basis 2, falls a[i] $\in$ {0,1,-1}.
Im folgenden zeigen wir, dass sich zwei redundante Zahlen addieren lassen, ohne dass der Uebertrag laueft.

Sei dazu

```
      a = a[n-1] a[n-2] ...  a[0]
      b = b[n-1] b[n-2] ...  b[0]
    ---------------------------------
      d = d[n-1] d[n-2] ...  d[0]   mit d[i] = a[i]+b[i]
                                          d[i] ∈ {0,1,2,-1,-2}
```

ist dann die formale Summe von a und b. Wir formen d jetzt um in eine redundante Zahl, die der Summe entspricht. Dazu wird d[i] in Abhaengigkeit von d[i-1] (d[-1]:=0) durch die Summe s[i] und den Uebertrag c[i] (d.h. d[i] = s[i] + c[i]*2) gemaess folgender Tabelle ersetzt:

d[i-1]>0 :	d[i]	s[i]	c[i]
	0	0	0
	1	-1	1
	2	0	1
	-1	1	0
	-2	0	-1
d[i-1]<=o:	0	0	0
	1	1	0
	2	0	1
	-1	1	-1
	-2	0	-1

Anhand der Tabelle prueft man nach, dass folgendes gilt:

BEHAUPTUNG:

Mit c[-1] := 0 gilt: s[i] + c[i-1] $\in$ {o,1,-1}.

Damit laesst sich d[n-1] d[n-2] ... d[0] ersetzen durch

```
              s[n-1] s[n-2] ...  s[0]
    +   c[n-1] c[n-2]  ...   c[0] c[-1]   und das wiederum ergibt
  ---------------------------------------
      e[n] e[n-1] ...     e[1] e[0]   wobei e[i] := s[i]+c[i-1]
                                            (s[n] := 0)
```

Gemaess der Behauptung ist e := e[n] ... e[0] eine redundante Zahl; ausserdem stellt sie nach obiger Konstruktion die Summe von a und b dar. Bei der Berechnung von e[i] genuegt die Kenntnis der Bits a[i], b[i],a[i-1],b[i-1]. Also ergibt sich die zweite

BEHAUPTUNG:

Die Summe zweier redundanter Zahlen laesst sich unhaengig von deren Laenge in konstanter Zeit berechnen, und das Ergebnis liegt als redundante Zahl vor.

BEISPIEL:

```
      a  =          1  0  0 -1 -1  1  0 -1        (  107)
      b  =         -1 -1 -1  1 -1  0  1  0        ( -214)
                  ---------------------------
      d  =          0 -1 -1  0 -2  1  1 -1

      s  =          0  1  1  0  0 -1  1  1
      c  =       0 -1 -1  0 -1  1  0 -1
                  ---------------------------
      e  =       0 -1  0  1 -1  1 -1  0  1        ( -107)
```

2) Entwurf einer Addierzelle

Es wird eine Zelle entworfen, die das folgende leistet:

Sie hat als Eingaenge drei Zahlen a[i], b[i], c[i-1] ∈{0,1,-1}, dargestellt im 1-Komplement durch Bits a[i,1],a[i,0],b[i,1],b[i,0], c[i-1,1],c[i-1,0]; ausserdem gibt es noch ein Eingangsbit V[i-1], V[i-1]=1 heisst: fuer die Summe der vorhergehenden Stelle gilt d[i-1]<=0,
V[i-1]=0 heisst: d[i-1]>0;
als Ausgaenge haben wir ein Bit V[i] (Bedeutung analog zu V[i-1]), je zwei Bits c[i,1],c[i,0],e[i,1],e[i,0], die dem Wert von c[i], bzw. e[i] entsprechen.

Durch Aufstellen und geschicktes Auswerten der Funktionstabellen erhaelt man folgende Boole'schen Ausdruecke, die die gewuenschte Funktionsweise sichern:

```
    V[i] = a[i,0]' & b[i,0]',

  c[i,1] = V[i-1]&a[i,1]&b[i,0]' | V[i-1]&a[i,0]'&b[i,1] |
           a[i,1]&b[i,1],
  c[i,0] = V[i-1]'&a[i,1]'&b[i,0] | V[i-1]'&a[i,0]&b[i,1]' |
           a[i,0]&b[i,0],

  e[i,1] = (a[i,0]⊕a[i,1]⊕b[i,0]⊕b[i,1])&V[i-1]'&c[i-1,0]' |
           (a[i,0]⊕a[i,1]⊕b(i,0]⊕b[i,1])'&c[i-1,1],
  e[i,0] = (a[i,0]⊕a[i,1]⊕b[i,0]⊕b[i,1])'&c[i-1,0] |
           (a[i,0]⊕a[i,1]⊕b[i,0]⊕b[i,1])&V[i-1]&c[i-1,0]'.
```

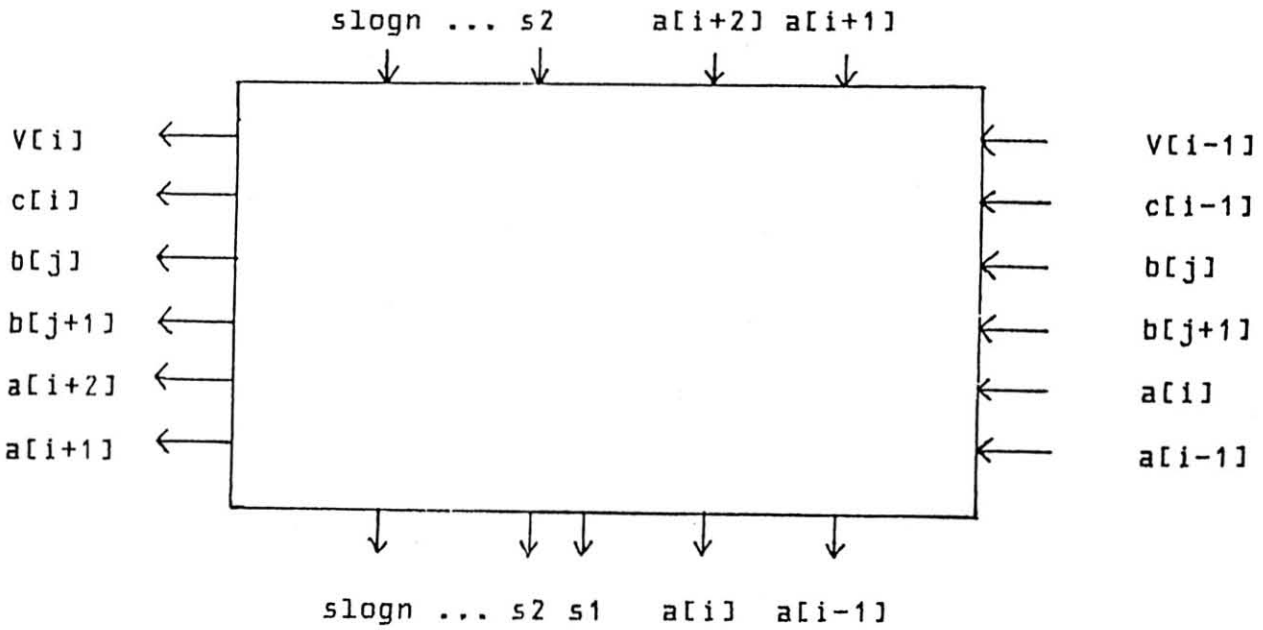
Als Kurzschreibweise fuer die vier letzten Gleichungen benutzen wir:

```
      c[i] := cred(a[i],b[i],V[i-1])
      e[i] := ered(a[i],b[i],c[i-1],V[i-1]).
```

Eine kurze Ueberlegung zeigt, dass eine Realisierung der Addierzelle schon mit weniger als 10 Gatterlaufzeiten moeglich ist.
Diese Addierzelle wird als Kern in den Zelltypen des Multiplizierers mit redundanter Zahlendarstellung vorkommen.
Wir deuten im folgenden skizzenhaft eine Realisierung an.

3) Skizze einer Realisierung des Multiplizierers

Wir gehen im folgenden davon aus, dass der Multiplikand a von oben, der Multiplikator b von rechts an das Multiplizierwerk herangefuehrt werden. Unten wird dann das Ergebnis erscheinen.
Gemaess der zu Beginn des Abschnittes angegebenen Skizze ist als erstes die Kodierung zu realisieren. Da eine 2-Komplement Zahl auch als redundante Zahl interpretiert werden kann, bereitet die Kodierung keine Schwierigkeiten, deshalb gehen wir hier nicht naeher darauf ein.
Danach werden die n Summanden mit Hilfe eines binaeren Baumes aufsummiert. Bei n Summanden ist dazu ein Baum mit log(n) Stufen notwendig.
Wir gehen zunaechst auf die erste Stufe ein, die die Blaetter des Baumes (die n Summanden) paarweise aufsummiert. Wir definieren dazu die Zelle ADD(1):

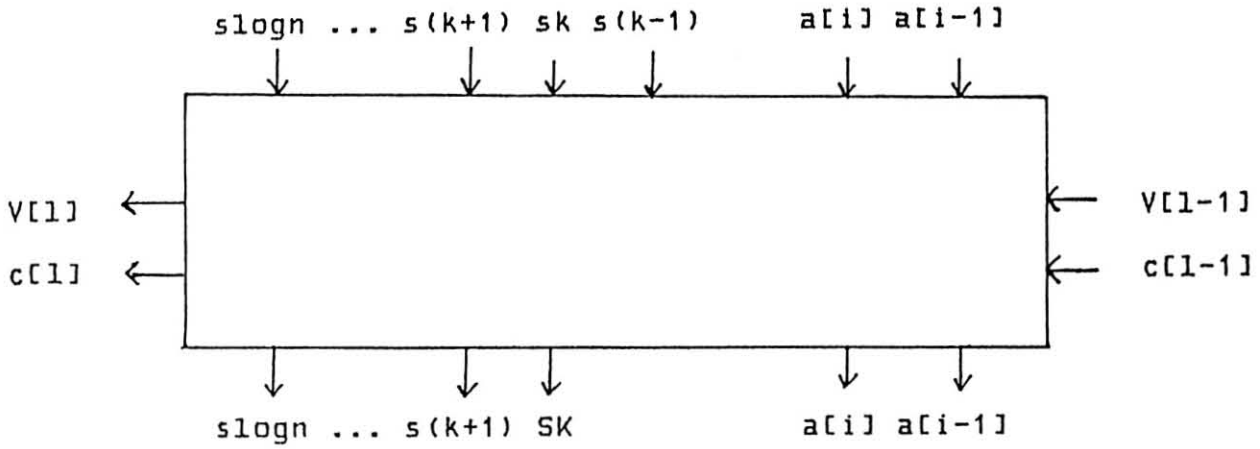


Die Funktionsweise der Zelle wird durch folgende Gleichungen gegeben:

$$c[i] = cred(a[i+2]*b[j], a[i+1]*b[j+1], V[i-1])$$

$$s1 = ered(a[i+2]*b[j], a[i+1]*b[j+1], c[i-1], V[i-1]).$$

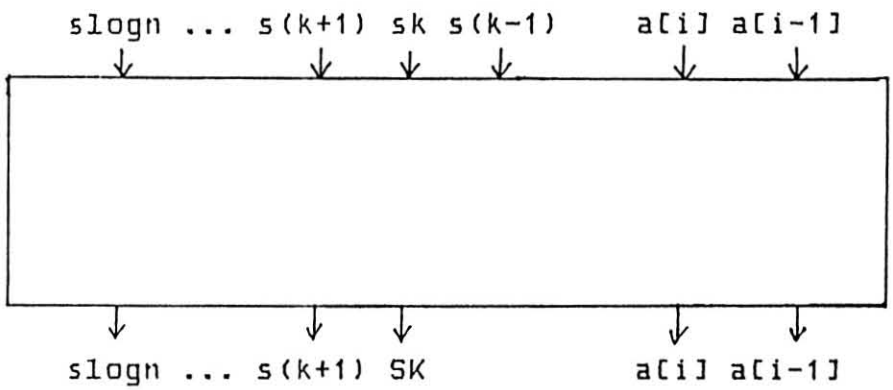
Als naechstes definieren wir die Addierzellen fuer die restlichen Stufen des binaeren Baumes. Sie werden mit ADD(k), k=2,...,log(n), bezeichnet und haben folgendes Aussehen:



dabei gilt: c[1] = cred(sk,s(k-1),V[1-1]),
SK = ered(sk,s(k-1),c[1-1],V[1-1]).

Um diese Zellen korrekt miteinander verknuepfen zu koennen, ist noch der folgende Zelltyp notwendig:

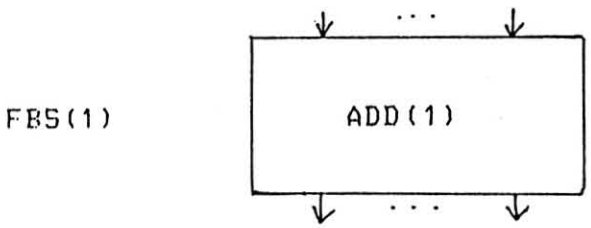
VERSCHIEBE(k) fuer k=2,...,log(n) wird gegeben durch:



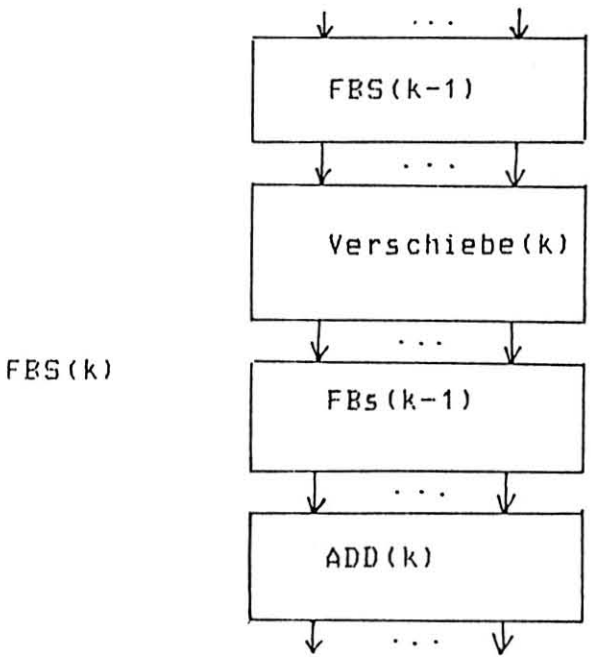
Es gilt: SK = s(k-1).

Damit sind alle Zellen, die zum Aufbau des Multiplizierers notwendig sind, definiert. Wir geben nun an, wie diese Zellen zusammenzusetzen sind:
Es wird rekursiv eine Zelle function-bit-slice FBS(k), k=1,...,log(n), durch Zusammensetzen der oben eingefuehrten Zellen definiert und zwar:

FBS(1) ist gegeben durch ADD(1):



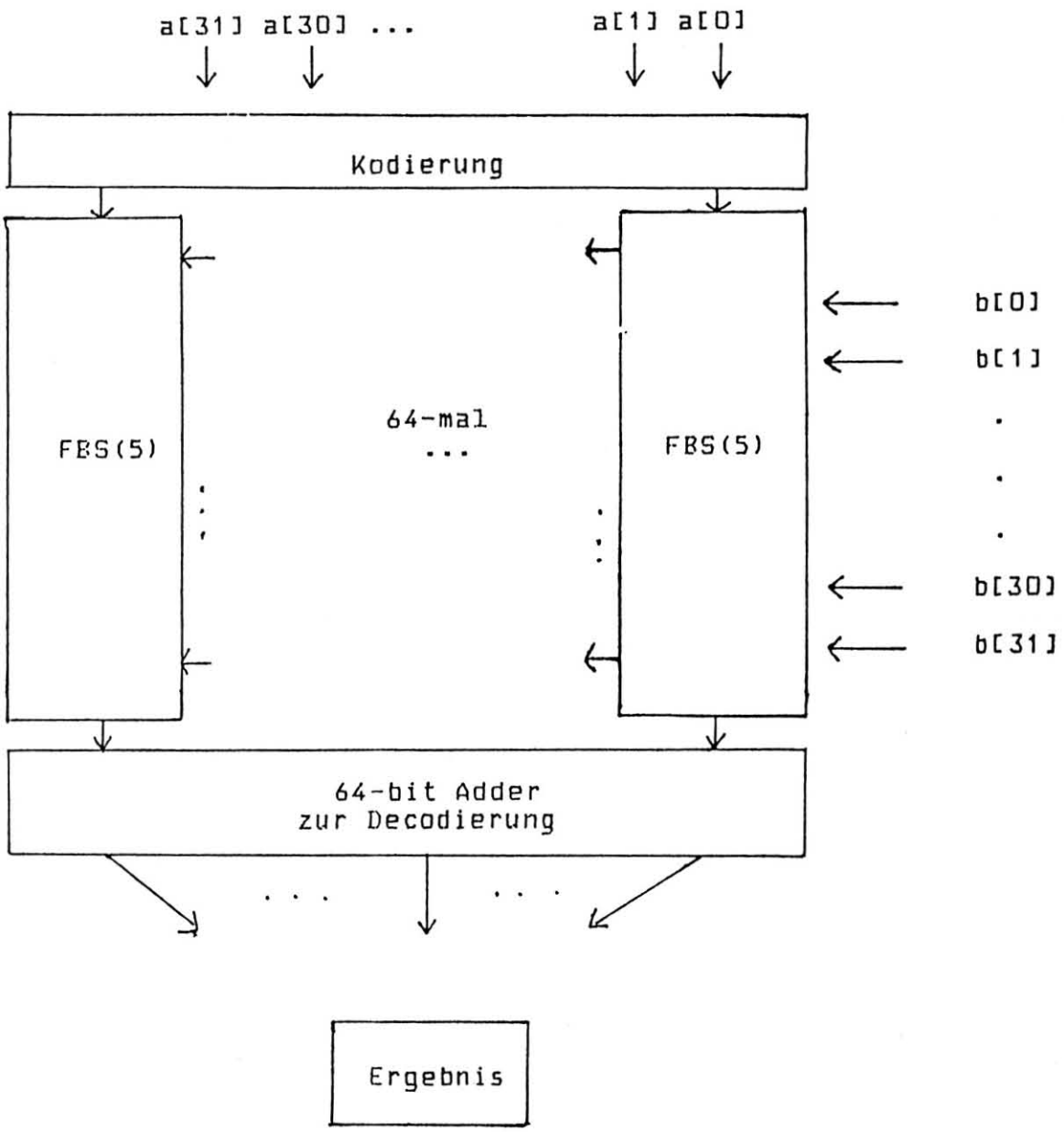
FBS(k) ist gegeben durch Zusammensetzung von ADD(k), VERSCHIEBE(k), FBS(k-1) auf folgende Art:



Durch Aufdoppeln von FBS(log(n)) laesst sich jetzt die Summation mit Hilfe des binaeren Baumes elegant darstellen.

Als letzter Schritt muss noch die Dekodierung der entstehenden redundanten Zahl vorgenommen werden, dies geschieht nach folgendem Prinzip:
Sei a = a[n] a[n-1] ... a[0] eine redundante Zahl, aus a konstruieren wir zwei 2-Komplement Zahlen a1 und a2:
a1 entsteht, indem jede -1 in a durch 0 ersetzt wird,
a2 entsteht, indem jede -1 in a durch 1 und jede 1 durch 0 ersetzt wird; danach ist noch a2 von a1 zu subtrahieren.
Man sieht leicht, dass die entstehende 2-Komplement Zahl der redundanten Zahl a (und damit dem Ergebnis der Multiplikation) entspricht.
Die Dekodierung einer redundanten Zahl laesst sich also als Addition zweier 2-Komplement Zahlen auffassen.

Fuer den Fall n=32 erhalten wir folgenden Aufbau fuer das gesamte Multiplizierwerk:



Damit ist der Entwurf des Multiplizierers vollkommen skizziert.

Eine Cap-Simulation, die die algorithmische Korrektheit des gesamten Verfahrens (inklusive der Korrektheit der Boole'schen Ausdruecke fuer die Adderzelle) testet und verdeutlicht, wurde fuer den Fall n=8 von Herrn R.Kolla durchgefuehrt (genaueres siehe Anhang).

Wir untersuchen zum Abschluss die Komplexitaeten. Es gilt:

```
A = O(n**2*log(n),
T = O(log(n)),
AT**2 = O(n**2*log(n)**3),
```

d.h. vom theoretischen Standpunkt aus gesehen handelt es sich wie in IV) auch hier um einen "fast optimalen" Algorithmus, im Bezug auf das Zeitverhalten ist kein besserer bekannt.

Kommen wir nun zum Verhalten und Problemen bei der praktischen Realisierung fuer n=32:
Die Zeitverzoegerung laesst sich nach obigem durch

```
   5*10 Gatterdelays
+  64-bit Adderdelay  abschaetzen.
```

Bei einer guenstigen Adder-Realisierung ist dies wesentlich schneller als die Verzoegerungszeit des Matrix-Multiplizierers. Allerdings ist bei VI) eine Vorzeichenkorrektur noch nicht beruecksichtigt, sie laesst sich aber analog zu III) realisieren und ist hier deshalb nicht betrachtet worden.

Die Probleme, die man dafuer in Kauf nimmt, liegen hauptsaechlich beim Platzbedarf. Es wird allein eine Breite von mehr als 64*16 Leitungen benoetigt. Ausserdem ist bei heutigen Designhilfsmitteln der rekursive Aufbau nur mit Muehe nachzuvollziehen.

Versuchen wir abschliessend ein

VII) FAZIT

Version 1 (Matrix-Multiplizierer) ist wegen ihres regelmaessigen nichtrekursiven Aufbaus und der Einfachheit der Zelltypen ohne Schwierigkeiten ins Bit-Slice Konzept einzupassen. Nach einem Design der Einzelzellen muesste eine konkrete Realisierung innerhalb kurzer Zeit moeglich sein.

Alle Versuche einen qualitativ besseren Algorithmus zu entwerfen, der ebenso gut in dieses Konzept passt, liefern einen stark vergroesserten Platzbedarf, ohne entsprechende Zeitverbesserungen mit sich zu bringen (siehe iterative Arrays oder modifizierter Booth-Algorithmus).

Aus diesem Grunde wurden in diesem Bericht in V) und VI) Algorithmen untersucht, die ein wesentlich besseres (sogar "optimales") Zeitverhalten erwarten lassen, die aber nicht oder ohne weiteres nicht in bisher uebliche Designkonzepte passen. Es wurde aber versucht, Regelmaessigkeiten in der algorithmischen Beschreibung bei der Realisierung auszunutzen.

Der entscheidende Nachteil von V) (modifizierter Booth-Algorithmus mit Wallace-Tree) ist die grosse Irregularitaet des Verfahrens, die eine angemessen "kurze" Beschreibung nicht zulaesst.

In VI) (redundante Zahlendarstellung und binaerer Baum) ist ein Verfahren gegeben, das sich mit dem Bit-Slice Konzept allein nicht angemessen beschreiben laesst, das aber durch eine elegante rekursive Beschreibung exakt wiedergegeben werden kann.
Elegante Beschreibung mittels Rekursion bietet sich bei allen Algorithmen, die z.B. binaere Baumstrukturen benutzen, an. Da fast alle "schnellen" Algorithmen in irgendeiner Form Baumstrukturen verwenden, ist eine Beschreibung mit rekursiven Hilfsmitteln sinnvoll und wichtig. Auf die Entwicklung von Designmethoden oder -hilfsmitteln, die dies beruecksichtigen, sollte also nicht verzichtet werden.

LITERATURVERZEICHNIS:

[BrKu] R.P.Brent, H.T.Kung:
The chip complexity of binary arithmetic
Proc. 12th Annual ACM Symposium on Theory of Computing,
ACM, May 1980, pp 190-200

[Do] B.S.Dormido:
High-speed Iterative Array for Binary Multiplication,
Electronics Letters, 9th Nov. 1978, Vol.14, No.23,
pp.742-743

[GrReNi] R.Gregorian, K.R.Reddy, W.E.Nicholson:
Round-off scheme for a high-speed modified
Booth's algorithm multiplier,
Microelectronics Journal, Vol.10, No.4, 1979, pp.27-30

[PrVu] F.P.Preparata, J.Vuillemin:
Area-time optimal VLSI network based
on the cube-connected cycles,
INRIA Report No.13, Rocquencourt, 1980

[ReKuMa] P.Reussens, W.H.Ku, Y.Mao:
Fixed-Point High-Speed Parallel Multipliers in VLSI,
in: VLSI Systems and Computations, Springer Verlag 1981,
pp. 301-310

[Sp] O.Spaniol:
Arithmetik in Rechenanlagen,
Teubner, 1976