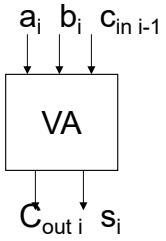
5. Computer Arithmetik

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. Alle Grundrechenarten werden dabei letztendlich auf die Addition zurück geführt.

Addition mit Volladddierer (1 Bit)

Wir kennen bereits einen Volladdierer für eine Stelle. 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 für die Stelle i:

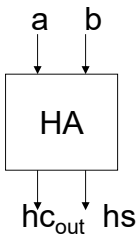


Eingabe			Ausgabe	
a_i	b_i	$C_{in\ i-1}$	$C_{out\ i}$	s_i
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

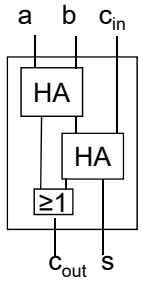
1

Halbadddierer (1 Bit)

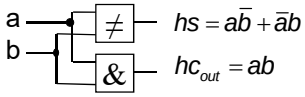
Häufig realisiert man einen Volladdierer nicht in DMF sondern in einer mehrstufigen Form, wobei man sogenannte Halbaddierer benutzt. Halbaddierer sind Schaltnetze, die zwei Bits addieren können (und demzufolge ein Ergebnis im Bereich 0 bis 2 produzieren). Durch Zusammenschalten von zwei Halbaddierern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



Eingabe		Ausgabe	
a	b	hc_{out}	hs
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0



hs = a⊕b
hc_{out}= a&b



2

Addition von zwei n-Bit Zahlen

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

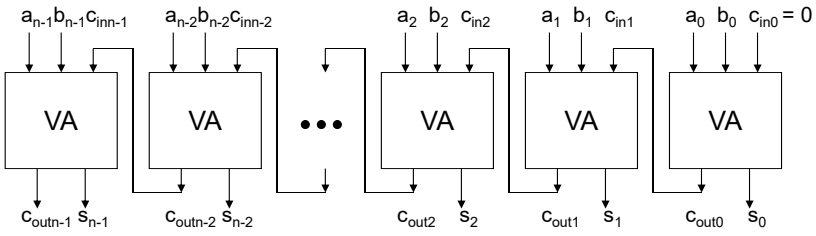
Insbesondere wegen dieser zweiten Eigenschaft ist der zweistufige Aufbau z.B. in DMF nicht sinnvoll. Stattdessen verwendet man im einfachsten Fall eine Kette von Volladdierern, die im Grunde genau das machen, was wir von der Addition in der „Schulmethode“ kennen. Man beginnt mit den LSBs (least significant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

3

Addition von zwei n-Bit Zahlen mit dem Ripple-Carry-Addierer

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

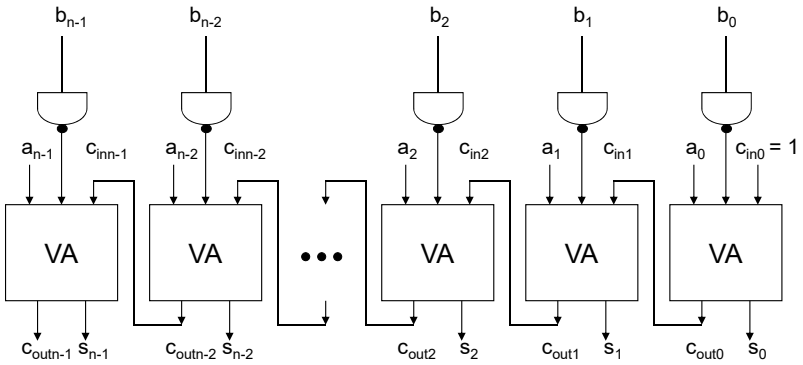
Einen solchen Addierer nennt man einen **ripple-carry-adder**. Sein Vorteil ist der einfache und modulare Aufbau. Sein wesentlichster Nachteil wird bereits durch diesen Namen ausgedrückt: Wenn die Operanden gerade eine ungünstige Bit-Kombination aufweisen, muß die Carry- (übertrags-) -Information durch alle Volladdierer hindurch von der Stelle mit der geringsten Wertigkeit bis zur Stelle mit der höchsten Wertigkeit hindurchklappern (ripple). Bei n-Bit Zahlen sind das n sequentielle Volladdierer-Berechnungen. Das carry c_{in0} wird auf 0 gesetzt.



4

Subtraktion von zwei n-Bit Zahlen mit dem Ripple-Carry-Addierer

Subtraktion ist sehr einfach durch die Addition mit dem 2-Komplement berechenbar. Das 2-Komplement einer Zahl läßt sich berechnen als 1-Komplement 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 a - b berechnen, indem wir zu a das Zweierkomplement von b hinzu addieren (n Inverter für b und c_{in0} = 1 setzen):



5

Rechenzeit für 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}.

In diesem Falle ist die Berechnungszeit linear von das Stellenzahl abhängig: Bei der Berechnung einer 64 Bit-Addition z.B. ist die Gesamtzeit 64 * t_{VA}.

In der O-Notation der Informatik spricht man hier von O(n). Nicht der exakte Zeitaufwand in Sekunden wird hier abgeschätzt, sondern die Abhängigkeit von der Anzahl der Stellen n.

O(n), also lineare Abhängigkeit bei der Addition, ist nicht gut für einen Addierer, denn wenn wir später z.B. die Multiplikation von zwei n-stelligen Zahlen berechnen wollen, nutzen wir die n-fache Addition, das führt dann auf eine Zeitaufwand von n*n* t_{VA} oder $O(n^2)$.

6

Beschleunigung der Addition und Subtraktion

Eine gute Lösung, die auch in der Praxis der Rechnerarchitektur häufig Verwendung findet, bietet der **carry-select-adder**.

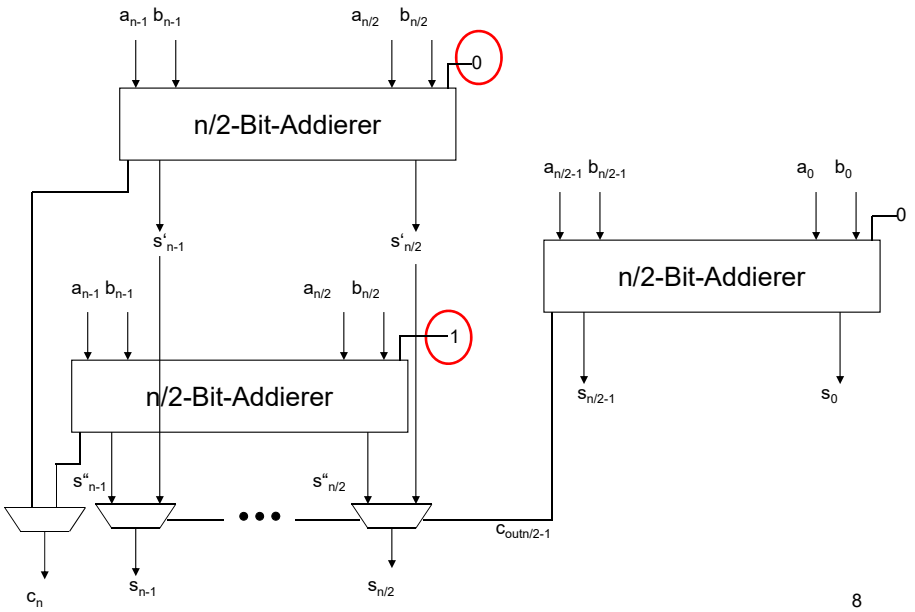
Die Idee ist folgende: Der Addierer wird in zwei gleich lange Hälften von n/2 Bit unterteilt. Für beide Hälften wird gleichzeitig mit der Addition begonnen. Bei der linken (höher signifikanten) Hälfte wissen wir aber nicht, ob am Carry-Eingang des rechtesten Volladdierers eine 1 oder eine 0 ankommt. 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. Der Preis, den wir dafür zahlen, ist die Chipfläche der zusätzlichen Hardware für die parallele Berechnung der oberen n/2 Bits.

Das Prinzip ist auf der folgenden Folie dargestellt.

7

Carry-Select-Addierer mit Unterteilung 2*n/2



8

Zeit- und Hardwareaufwand beim Carry-Select-Addierer

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 drei n/2-Bit-Aaddierer 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}

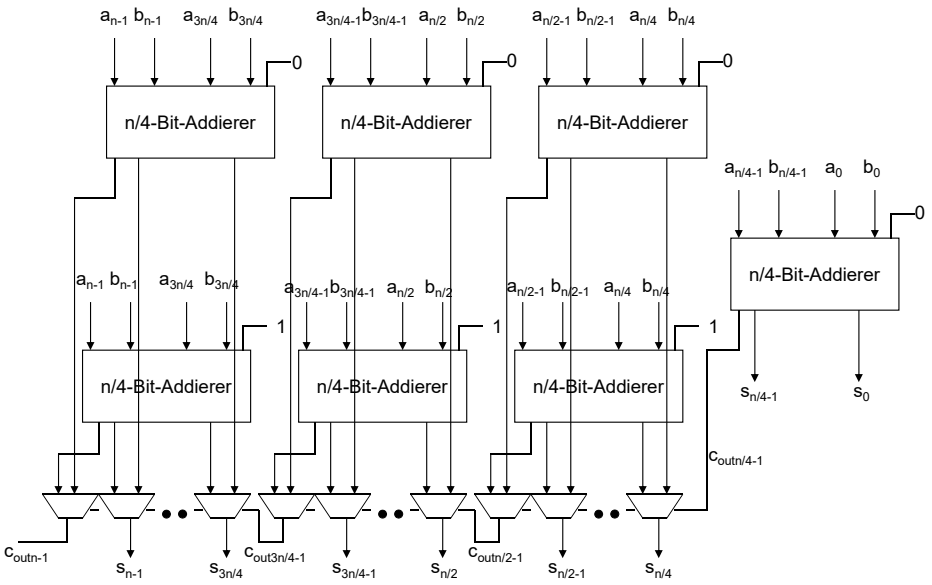
Wir haben also etwa einen Faktor 2 in der Zeit gewonnen, bei einer Erhöhung der Gatterzahl auf 3/2. Nun läßt sich dieses Prinzip natürlich wiederholt anwenden: Anstelle von Addierern der Länge n/2 können wir auch solche der Länge n/4 oder n/8 usw. verwenden. Alle solchen Addierer (außer dem am wenigsten 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:

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

9

Carry-Select-Addierer mit Unterteilung 4*n/4



10

Zeit- und Hardwareaufwand beim Carry-Select-Addierer

Die Gesamtzeit bei m Unterteilungen ist $t_{Gesamt} = n/m * t_{VA} + (m-1) * t_{MUX}$

Da die Durchlaufzeit eines Volladdierers ähnlich ist wie ein Schalten der Multiplexer, setzen wir beide gleich. Wenn wir das Minimum für die Gesamtzeit ausrechnen wollen, setzen wir zur Vereinfachung $t_{VA}=t_{MUX}$.

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

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.

Die Einsparung ist erheblich, z.B. ist bei 64 Bit Addition das Verhältnis $O(n)/O(n^{1/2}) = 8$. Der Carry-Select-Addierer ist dann ca. 8x schneller als Ripple-Carry. Allerdings ist die Anzahl der benötigten Gatter (und damit auch die Stromaufnahme) bei Carry-Select ungefähr doppelt so groß wie beim Ripple-Carry.

Um den Gesamteffekt der Optimierung abschätzen zu können, wird oft sowohl die insgesamt benötigte Gatterfläche A der Schaltung als Vielfaches x der Gatterfläche eines Volladdierers $x = A / A_{VA}$ abgeschätzt und zudem noch die Laufzeit T als Vielfaches y der Zeit eines Volladdierers $y = T / T_{VA}$ berechnet. Das Produkt x*y berücksichtigt dann sowohl den Flächenzuwachs wie auch die Zeitreduktion.

11

Carry Save Addierer und redundante Zahlendarstellung

Ein anderer Ansatz zur Beschleunigung 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**.

Auf John von Neumann geht die Idee zurück, das Ergebnis eines n-Bit-Addierers für jede Stelle getrennt zu behandeln. Da bei einem 1-Bit-Volladdierer $(a,b,c_{in}) \rightarrow (c_{out},s)$ die Zahlen {0,1,2,3} berechnet werden, kann dieses Resultat nicht im Binärsystem gespeichert werden. Wohl aber kann die entstehende Zahl durch zwei Zahlen gespeichert werden, wobei mehrere (redundante) Darstellungen des Zifferntupels {c,s} möglich ist. So kann 2 = 0+2=1+1=2+0 dargestellt werden. Beispiel: berechne Binäraddition stellenweise, Ergebnis für jede Stelle ist aus dem Bereich {0,1,2}, es wird kein carry berücksichtigt. Das Ergebnis muss nachträglich wieder in die Binärdarstellung transformiert werden (mit Carry-Übertragsrechnung).

```
  10111010101011011111000000001101
+ 11011110101011011011111011101111
= 21122120202022022122111011102212
```

Addition von 2 Zahlen. Ziffern 0-2

```
  10111010101011011111000000001101
+ 11011110101011011011111011101111
s: 01100100000000000100111011100010
c:10011010101011011011000000001101
```

Resultat binär mit 2 Ergebnissen s und c

12

Carry Save Addierer

Der Carry-Save-Addierer greift diese Darstellung auf. 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.

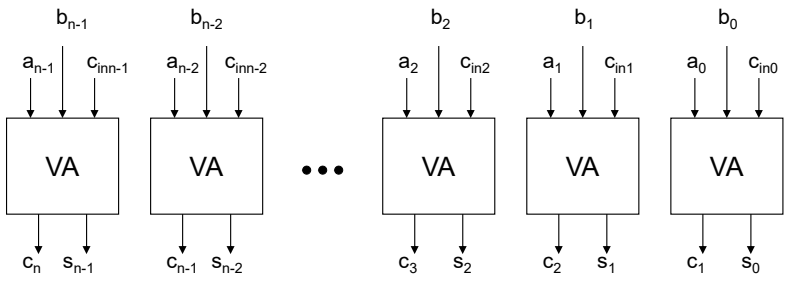
Der Aufbau eines Carry-save-Addierers ist lediglich die Parallelschaltung von n Volladdierern. Diese sind nun nicht verkettet, sondern jeder liefert zwei Ausgabebits, ein s-Bit und ein c-Bit. Aus diesen wird ein s-Wort und ein c-Wort gebildet, die zusammen die beiden Ergebnisworte darstellen. Das c-Wort wird durch eine 0 an der am wenigsten signifikanten Stelle ergänzt (Shift um 1 Stelle entspricht Multiplikation mit 2) 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}.

Es ist aber zu bedenken, dass diese Darstellung noch kein vollständiges Additionsergebnis darstellt, sondern ein Zwischenergebnis, das aber sehr einfach und in konstanter Zeit O(1) berechenbar ist, da alle Volladdierer parallel arbeiten. Am Ende muss noch ein Carry-Select oder Ripple-Carry Addierer die Summe s+c erechnen.

13

Carry Save Addierer mit n parallel arbeitenden Volladdierern

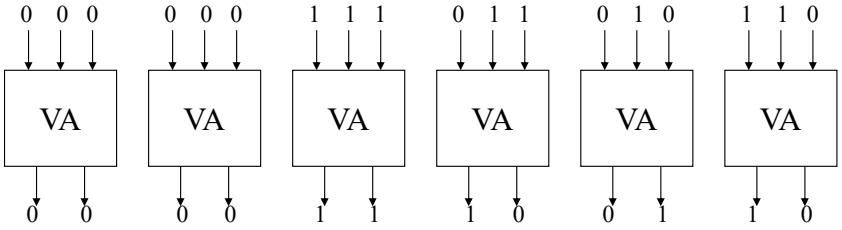
Der Carry-Save-Addierer berechnet $a_i + b_i + c_{in,\,i} = s_i + c_{out,\,i+1}$ für jede Stelle unabhängig.



Es muss beachtet werden, dass das Ergebnis c_{out} um 1 Stelle nach links verschoben ist, so dass das Ergebnis für c mit 2 multipliziert werden muss (links shift um 1)!

14

Beispiel:
Wir addieren die Worte:
$a = 9_{10} = (001001)_2$, $b = 15_{10} = (001111)_2$, $c_{in} = 12_{10} = (001100)_2$
=> 9 + 15 + 12= 36



Das abgelesene Ergebnis ist: s = 001010, c_{out} = 001101

Das Rechenergebnis ist r = s + 2* c_{out} (mit Verschiebung von c um 1 Stelle)

s	(10)	0001010
+ 2*c_{out}	+(2*13)	+ 0011010
= r	= (36)	= 0100100

15

Einsatz des Carry-Save-Addierers für Kolonnen-Addition

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 nicht behandelt.

16

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
-------------------
0110100110010101
```

17

Carry-Save Multiplikation

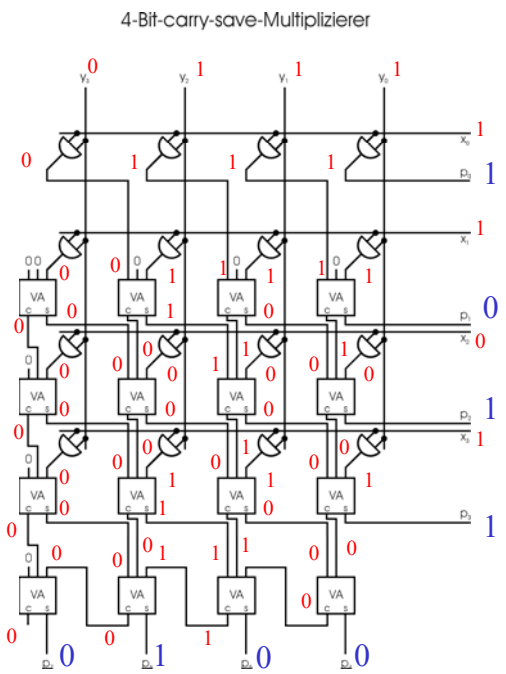
Ein Carry-Save-Multiplizierer für zwei Operanden der Länge n besteht aus n^2 AND-Gattern, die gleichzeitig alle erforderlichen 1-Bit-Multiplikationen ausführen (Die binäre Multiplikation von 1-Bit Zahlen ist gerade das logische „UND“). Danach sind nur noch alle Teilprodukte (wie bei der Schulmethode) 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 zunächst 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.

18

4-Bit-carry-save-Multiplizierer



19

Division

Frühe Rechner (< 1980) führten die Division in Software entsprechend der Schulmethode aus, 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.

20

Schnelle 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. Dabei wird die eigentliche Division wieder auf eine Multiplikation mit dem Kehrwert zurück geführt, und durch die Carry-Save-Methode letztendlich wieder auf die Addition.

Gür die Berechnung von D=A/B wird nicht dividiert, sondern mit dem Kehrwert des Divisors 1/B multipliziert: D= A*(1/B). Dafür muss zunächst der Kehrwert von B gefunden werden. Ein gängiges Verfahren dafür ist die Newton-Raphson-Methode, die wir hier kennenlernen wollen.

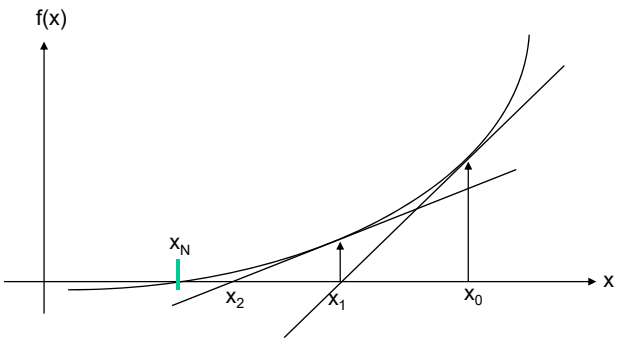
Die Idee des Newton-Verfahrens ist die Approximation der Nullstelle einer Funktion durch Konstruktion einer Folge von Werten, die sehr schnell gegen die Nullstelle konvergiert. Für unsere Anwendung benötigen wir daher ein Funktion f(x), die an der Stelle x=1/B eine Nullstelle hat. Sie ist einfach konstruiert:

f(x)= 1/x – B => für x=1/B gilt : 1/(1/B) –B = B – B =0

21

Newton-Methode zur Bestimmung von Nullstellen

Es wird die Tangente der Funktion f(x) bei einem Startwert x_0 berechnet und deren Schnittpunkt mit der Abszisse x_1 wird als nächster Folgenwert berechnet. Nun legt man an dessen Funktionswert x_1 die Tangente an usw. Wenn die Funktion bestimmte Bedingungen erfüllt und wenn der Anfangswert geeignet gewählt ist, konvergiert die Folge der Werte x_i gegen die Nullstelle x_N.



22

Folge zur Bestimmung von 1/B

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 und ihre Ableitung f'(x)= -1/x². Diese Funktion hat in 1/B eine Nullstelle. Wenn wir f(x) und f'(x) 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.

23

Konvergenz der Folge und Approximationsfehler

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 Iteration drei Operationen. Bei einer 64-Bit Division ist das eine Reduktion von 128 auf 18 Operationen.

24