Untersuchung und Bewertung von Hardwarestrukturen für die VHDL Implementation schneller digitaler Rechenwerke

Studienarbeit

Universität Hamburg - Fachbereich Informatik
Arbeitsbereich Technische Grundlagen der Informatik

Sebastian A. Wallner
11. Dezember 1997

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren ist ein Trend, ausgehend von „klassischen von Neumann Architekturen“ hin zu Spezialhardware für höhere Rechenleistung und damit neue Einsatzgebiete, zu beobachten. Hierzu kam eine Vielzahl neuer Anwendungsbereiche wie etwa digitale Kommunikation, digitale Signalverarbeitung oder Sprach- und Bildverarbeitung. In der heutigen Zeit werden diese, früher analog bewerkstelligten Verfahren immer weiter von Digitalen abgelöst, da diese gravierende Vorteile im Bezug auf Elimination der Einflüsse von Fertigungstoleranzen, Alterung und Zufallsprozessen wie etwa Rauschen bieten. Grundlage dieser digitalen Verarbeitung sind schnelle digitale Rechenbausteine, die es schaffen Rechenoperationen in ns Bereich auszuführen. Ziel dieser Studienarbeit sollen Untersuchungen und Bewertungen von geeigneten und schnellen Addierern und Multiplizierern sein, die in schnellen Rechenanlagen, digitalen Signal- und Standardprozessoren Verwendung finden. Nur so können die enormen Mengen an Daten in Echtzeit, was häufig der Fall sein muß, man denke an digitale Filterung, verarbeitet werden. Es werden mehrere geeignete Hardwarestrukturen, die in der Hardwarebeschreibungssprache VHDL[1] programmiert wurden, für digitale Rechenanlagen vorgestellt. Darüber hinaus war die Realisierung einer geeigneten Simulationsumgebung in VHDL notwendig, um die erzielten Ergebnisse beurteilen zu können. Das Ergebnis dieser Arbeit sind Hardwarestrukturen für schnelle Additionen bzw. Multiplikationen die später als Hardwareimplementation in ver-schiedenen Gebieten der digitalen Signalverarbeitung für die benötigten Grundrechen-operationen Verwendung finden können.

[1] **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage (VHDL)

1.1 Motivation

Die digitale Signalverarbeitung ist ein rapide wachsendes Gebiet, das die Darstellung, Auswertung, Transformation und Manipulation von Signalen umfaßt. Wesentliche Grundlagen zur digitalen Verarbeitung von Signalen setzen heutzutage schnelle Rechenkomponenten wie Addierer und Multiplizierer voraus. Besonders zeitliche Bearbeitungsbedingungen führen zu hohen Anforderungen der digitalen Signalverarbeitung bezüglich Rechenleistung und Zugriffsrate. So wäre das ISDN[1] Netz oder das heutige Mobilfunknetz, das digitale Signal-verarbeitung voraussetzt ohne eine enorme Rechenleistung nicht in dieser Form möglich gewesen, wie es heute existiert. Diesen hohen Anforderungen ist nur Spezialhardware gewachsen, da selbst die heutigen programmierbaren Standardprozessoren häufig nicht die benötigte Signalverarbeitungsleistung bereitstellen. Als Lösung dieses Problems bieten sich hier anwendungsspezifische Schaltungen (ASIC[2]) an, bei denen Basisoperationen der digitalen Signalverarbeitung direkt und mit einer hohen Parallelität realisiert sind, mit fester Operationsverarbeitung und ohne Kontrollfluß. Eine Verarbeitung im „von Neumann" Stil mit einer leicht änderbaren Befehlsabarbeitung ohne Parallelität würde nie die benötigte Rechenleistung erbringen und scheidet somit für die meisen Signalverarbeitungsaufgaben aus. Die Spezialhardware hat jedoch den Nachteil mangelnder Flexibilität, so daß eine nachträgliche Modifikation von dort realisierten Algorithmen erschwert wird. Auch ein Einsatz der gleichen Hardware für verschiedene Anwendungen ist nicht möglich. Trotzdem bietet eine Spezialhardware derzeit den einzige Weg mit Echtzeitanforderungen, wie die Auswertung von Bildsequenzen oder Radarsignalen eine Signalverarbeitungsleistung, die nicht von heutigen digitalen Signalprozessoren (DSPs) bereitgestellt wird, zu realisieren. Man denke an eine digitale Filterung von Signalen im unteren Mhz Bereich, wo heute immer noch analoge Technik eingesetzt wird. Heutige moderne DSPs enthalten zwar spezielle und mehrere parallel arbeitende Einheiten, können jedoch höchstens als Mittelweg zur Lösung obiger Probleme angesehen werden. Auch ein Einsatz dieser Spezialarchitekturen zur Erhöhung des Gesamtdurchsatzes in heutigen Standardmikroprozessoren wäre möglich. Vereinzelnt findet man diese Architekturen schon in neueren Prozessorgenerationen, wie etwa in den Alpha, PowerPC und einigen RISC Maschinen [Pirsch 95]. Aufgabe dieser Studienarbeit ist es zu prüfen, wie Addition und Multiplikation, die als Basisopationen der digitalen Signalverarbeitung bezeichnet werden können, in VHDL synthesegerecht realisiert werden können. Diese Komponenten können dann als Grundbauelemente in weitaus komplexeren Schaltungen, wie etwa in digitalen Filterbausteinen, weiterverwendet werden.

[1] **A**plication **S**pecific **I**ntegrated **C**ircuit (ASIC)
[2] **I**ntegreated **S**ervice **D**igital **N**etwork (ISDN)

2 Chips die sich schon auf dem Markt befinden

Die heutigen Möglichkeiten zur schnellen arithmetischen Berechnungen sind entweder digitale Signalprozessoren oder eine Spezialhardware, realisiert als Chipentwurf. Zu diesem Zweck haben viele Halbleiterfirmen DSPs (Digitale Signalprozessoren) oder spezielle arithmetische Bausteine entwickelt. Im Nachfolgenden sollen die beiden Realisierungsmöglichkeiten kurz vorgestellt und erläutert werden.

2.1 Spezielle arithmetische Chips

Viele Hardwaremultiplizierer sind mit Hilfe des erweiterten Booth Algorithmus realisiert (vergl. Booth Multiplizierer Abschnitt 5.2.4). Dabei werden die Anzahl der zu addierenden Partialprodukte durch Umkodierung des Multiplikators für einen Geschwindigkeitsvorteil reduziert. Durch diese Maßnahe reduziert sich auch die Anzahl der Teilkomponenten, die an einer Multiplikation beteiligt sind. Die Wortbreiten derartiger Multiplizierer liegen zwischen 8 und 32 Bit. Da bei der Multiplikation ein Ergebnis mit doppelter Wortbreite entsteht, verwenden die meisen Typen ein Multiplexverfahren oder es findet ein Datenaustausch über bidirektionale Leitungen statt, um bei diesen Multiplizierchips Pins zu sparen und somit die Kosten zu senken. Trotz dieser Maßnahmen besitzen derartige Chips immer noch über 100 Anschlußpins.
Die meisten Chips besitzen zusätzlich noch einige Pins, mit denen sie zwischen Betrags und Zweierkomplementrechnung und Darstellung umgeschaltet werden können [Motorola89]. Ferner besitzen einige Typen eine MAC[1] Einheit, mit der es möglich ist, zu einem Produkt einen Summanten zu addieren [Harris91]. Diese Aufgabe ist in der digitalen Signalverarbeitung sehr oft zu finden und stellt daher einer der wichtigsten erweiterten Operationen dar, besonders bei der Realisierung des Faltungssatzes [Hamming83], der als fundamental in der digitalen Signalverarbeitung anzusehen ist.

Als Fertigungsprozeß findet heute fast ausschließlich ein CMOS[2] Prozeß Verwendung. Trotzdem sind diese Bausteine aufgrund großer Fortschritte in der Prozeßtechnologie (besonders durch kleinere Strukturbreiten), nicht langsamer als ihre Vorgänger in TTL[3] oder ECL[4] Technik.

---

[1] **M**ultiply **a**nd **A**ccumulate (MAC)
[2] **C**omplementär **M**etal **O**xyd **S**ilizium (CMOS)
[3] **T**ransistor **T**ransistor **L**ogic (TTL)
[4] **E**mitter **C**oupled **L**ogic (ECL)

Größter Vorteil ist jedoch, daß diese Bausteine selbst bei hohen Taktraten mit nur einem Bruchteil des Stromes ihrer Vorgänger auskommen und dadurch auf Kühlkörper und Lüfter verzichtet werden kann. Das spart enorme Kosten und hilft das Gesamtsystem unanfälliger gegen Störungen zu machen. Trotzdem werden Systeme für sehr hohe Leistungen meist noch vollständig in ECL Technik aufgebaut, da bei einem Gemisch verschiedener Techniken, die dafür benötigten Pegelumsetzer für die Signale eine zu hohe Verzögerung hätten.

2.2 Digitale Signalprozessoren

Die digitalen Signalprozessoren, die eine hohe Leistung im Bezug auf arithmetische Signalverarbeitung aufweisen, bedienen sich einer speziellen Architektur. Die meisten Signalprozessoren sind durchweg in CMOS realisiert, nur einige wenige, meistens ältere, benutzen andere Prozesse. Die Basisarchitektur von Standartmikroprozessoren haben als wesentlicher Merkmal den gemeinsamen Speicher für Programme und Daten („von Neumann“ Architektur). Daten und Programme können somit ineinander verschachtelt sein und es kann auf einen einzigen Hauptspeicher nur sequentiell gelesen und geschrieben werden. Neben dem gemeinsamen Daten und Programmspeicher enthält der Standartmikroprozessor nur ein Rechen- und Steuerwerk. Wesentliches Element des Rechenwerkes ist die ALU[1]. Diese ALU kann alle arithmetischen Basisoperationen und bitweise logische Verknüpfungen nur sequentiell durchführen.

Signalprozessoren dagegen sind nicht im „von Neumann“ Stil aufgebaut, sondern benutzen die „Harvard Architektur“. Kennzeichnung dieser Architektur sind die getrennten Daten und Programmspeicher d.h. es kann gleichzeitig auf diese Speicher gelesen und geschrieben werden. Zu diesem Zweck existieren im Chip mehrere Daten- und Adreßbusse. Der wesentliche Teil der Geschwindigkeit eines digitalen Signalprozessors beruht auf die Anwendung dieser Architektur. In heutigen digitalen Signalprozessoren verrichten außerdem mehrere ALUs parallel ihre Arbeit, was den Durchsatz noch weiter steigert [DSP56002]. Die einzelnen ALUs enthalten alle Array Multiplizierer und Hardware Barrel Shifter, um Bits schnell verschieben zu können. Ferner sind ein MAC und andere Befehle, die häufig in der digitalen Signalverarbeitung Verwendung finden, als Hardwarerealisierung in diesen Bausteinen vorhanden. Größter Vorteil dieser Baustein ist deren universeller Einsatz, da ihre Funktionen programmiert werden kann.

[1] **A**rithmetische **L**ogische **E**inheit (ALU)

3 Theoretische Grundlagen

3.1 Zahlendarstellung

Die Grundlage der meisten Zahlendarstellung ist ein Stellenwertverfahren. Die Zahlen werden durch eine Anreihung von Ziffern dargestellt, wobei sich der Wert der Zahl aus der Summe des Produktes aus Zifferwert und Stellenwert ergibt. Dieses bedeutet, daß der Wert der Zahl über die Position der Ziffern bestimmt wird. Für natürliche Zahlen mit n Ziffern wird eine Darstellung

$$A = a_{n-1}, a_{n-2}, \ldots\ a_1, a_0 \qquad (3.1.0)$$

mit steigendem Stellenwert nach links gewählt. Der Wert der Zahl ist dann gegeben durch

$$V(a) = \sum_{i=0}^{n-1} a_i r^i \qquad (3.1.1)$$

wobei r die Basis des Zahlensystems und i die Position als Potenzfunktion dargestellt ist. Digitalschaltungen können nur binäre d.h. zweiwertige Informationen verarbeiten. Aus diesem Grund müssen die gewohnten Dezimalzahlen der Basis 10, die man normalerweise verwendet, in ein binäres System der Basis 2 übersetzt werden.

3.1.1 Positive ganze Zahlen im Dualcode

Die einfachste binäre Zahlendarstellung ist der Dualcode, der im nachhinein in etwas abgewandelter Form auch weiter Verwendung findet. Die Stellen sind nach steigenden Zweierpotenzen angeordnet, wobei für die Darstellung einer N-stelligen Zahl im Dualcode gilt:

$$Z_N = z_{N-1}\,2^{N-1} + z_{N-2}\,2^{N-2} + \ldots + z_1\,2^1 + z_0\,2^0 \quad = \sum_{i=0}^{N-1} z_i\,2^i \qquad (3.1.2)$$

Wie beim Dezimalsystem schreibt man einfach die Ziffernfolge $\{z_{N-1} \dots z_0\}$ auf und denkt sich die Multiplikation mit der betreffenden Zweierpotenz und die Addition dazu.

Beispiel(Wortbreite 8Bit):

199dez =	1	1	0	0	0	1	1	1		Dualzahl
	2^7			...				2^0		Stellenwert

Mit einer 8 stelligen positiven Dualzahl kann also der Bereich von 0 bis 255 darstellt werden.

3.1.2 Ganze Zahlen mit beliebigem Vorzeichen

Eine negative Zahl läßt sich einfach dadurch charakterisieren, daß man vor die höchste Stelle ein Vorzeichen s setzt. Durch diese Maßnahme wird die Zahl aufgeteilt in Betrag und Vorzeichen. Eine Null im Vorzeichenbit bedeutet eine „positive“ eine Eins eine „negative“ Zahl. Voraussetzung ist hierbei jedoch die Vereinbarung einer festen Wortbreite.

Beispiel: (Wortbreite 8Bit):

+32dez =	0	0	1	0	0	0	0	0
- 32dez =	1	0	1	0	0	0	0	0
	(-1^s)	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Nachteilig an der Darstellung nach Betrag und Vorzeichen ist u.a. jedoch, daß positive und negative Zahlen nicht einfach addiert werden können[1]. Diese Art von Addierer müßte beim Auftreten eines Minuszeichens auf Subtraktion umgeschaltet werden, was einen höheren Hardwareaufwand bedeuten würde. Abhilfe schafft hier eine andere Zahlendarstellung, bei der ein einfacher Addierer ausreicht, um auch negative Zahlen einfach addieren zu können.

3.1.3 Vorzeichen 2-er Komplement Darstellung

Bei der 2er Komplementdarstellung gibt man wie bei den Dualzahlen mit beliebigem Vorzeichen dem höchsten Bit ein negatives Gewicht. Der Rest der Zahl wird als normale Dualzahl dargestellt. Auch hier muß eine feste Wortbreite vereinbart werden, damit das höchste Bit eindeutig definiert ist. Ein eventueller Übertrag ist dann gesondert zu erkennen und zu behandeln.

[1] Ferner existieren in dieser Darstellung zwei Repräsentationen für die 0 (10000000 = -0 00000000 = 0), was die Arithmetik weiter erschwert.

Wie bei der obigen Zahlendarstellung ist bei einer positiven Zahl das höchste Bit 0, bei einer negativen Zahl 1. Nur diese Stelle trägt ein negatives Gewicht.
Formal bedeutet dies:

$$Z_N = -1z_{N-1}2^{N-1} + z_{N-2}\,2^{N-2} + \ldots + z_1\,2^1 + z_0\,2^0 = \sum_{i=0}^{N-1} s_i z_i 2^i \tag{3.1.3}$$

Beispiel: (Wortbreite 8 Bit):

+32dez =	0	0	1	0	0	0	0	0
- 32dez =	1	1	1	0	0	0	0	0
	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Angenehm an der Darstellung negativer Zahlen im 2er Komplement ist die Einfachheit der resultierenden Arithmetik. Die Subtraktion von a und b wird auf die Addition zurückgeführt, indem man a und das 2er Komplement von b addiert. So ist es möglich während der arithmetischen Operation das Vorzeichen der Zahlen zu ignorieren, bis man das Vorzeichen des Ergebnisses wissen will. Ein eventueller Überlauf muß jedoch vorher erkannt und gesondert bearbeitet werden (Signextension), was jedoch kaum ein Problem darstellt. Addition, Subtraktion, Multiplikation sind dafür relativ einfach zu implementieren, auch die Division in 2er Komplement wird nicht sonderlich schwieriger.
Haben die arithmetischen Operationen nicht zu einem Überlauf und damit zu einer Zerstörung der höchstwertigen Binärstelle geführt, dann gibt diese Stelle das richtige Vorzeichen des Ergebnisses an. Aufgrund dieser Vorteile wird überwiegend nur im 2er Komplementdarstellung gerechnet. Die Addition einer 1 kann leicht durch den Übertragseingang bei der Rechnung realisiert werden. Das Ergebnis ist immer in 2er Komplementdarstellung. Die Umwandlung von negativen Binärzahlen in 2er Komplementzahlen ergibt sich durch Bildung des 1er Komplements (einfache Invertierung aller Stellen) und Addition von 1.

Beispiel: (Wortbreite 8Bit):
Wandlung in 2'er Komplement:

+32dez =		0	0	1	0	0	0	0	0	
Einerkomplement:		1	1	0	1	1	1	1	1	
	+								1	
Zweierkomplement:		1	1	1	0	0	0	0	0	= -32dez

Die Rückwandlung erfolgt genau so:

Zweierkomplement:	1	1	1	0	0	0	0	0		
Einerkomplement:	0	0	0	1	1	1	1	1		
+								1		
	0	0	1	0	0	0	0	0	=	+32dez

Es ist leicht zu sehen, daß das 2'er Komplement keine symetrische Zahlendarstellung, wie beispielsweise die 1'er Komplementdarstellung ist, bei der ein Bereich von -127 bis +127 umfaßt wird. Mit einer 8 stelligen 2er Komplementzahl kann beispielsweise der Bereich von -128 bis +127 abgedeckt werden.

+127	0	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
-128	1	0	0	0	0	0	0	0

3.1.4 Vorzeichenerweiterung (Signextension 2-er Komplement)

Relativ oft muß der benutzte Zahlenbereich im 2er Komplement erweitert werden, da entweder ein Übertrag aufgetreten ist, oder einfach die benötigten Zahlen größer werden. Wollte man bsw. eine positive Zahl auf eine größere Wortbreite erweitern, so ergänzt man einfach führende Nullen. In Zweierkomplementdarstellung gilt eine andere Regel: Man muß das Vorzeichenbit in die höheren Stellen kopieren (Signextension).
Hierzu ein Beispiel:

Beispiel:	8Bit	16Bit
+ 32dez	0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
- 32dez	1 1 1 0 0 0 0 0	1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0

4 Schnelle Addierstrukturen

4.1 Addierer

Basisoperation vieler Signalverarbeitungsaufgaben ist die Addition. Schaltungen zur Realisierung der Addition sind daher von besonderer Bedeutung. Für die Erziehung hoher Durchsatzraten werden im Nachfolgenden Grundprinzipien und Schaltungen mit geringer Verzögerung für Addierrealisierungen vorgestellt. Zum Teil werden diese dann auf Standardzellen abgebildet, um einen besseren Vergleich zu ermöglichen.

4.2 Addiergrundschaltungen für positive Dualzahlen

Eine direkte Hardwarerealisierung ist die direkte Umsetzung des Addierverfahrens, wie es per Hand durchgeführt wird. Hierbei werden ausgehend von er niedrigsten Bitebene die Operanden nacheinander bitweise unter Berücksichtigung eines eventuellen Übertrages (Carry) der vorangegangenen Bitebene addiert. Eine Schaltung, die 2 Operandenbits und ein Übertragbit in einen Summenbit und ein Übertragsbit für die nächst höhere Bitebene überführt, wird als Volladdierer (VA) bezeichnet.
Ein Habladdierer (HA) hingegen berücksichtigt keinen eingehenden Übertrag erfüllt jedoch sonnst die gleiche Funktion. Eine Addierschaltung unter Verwendung von Volladdierern zur Addition zweier 1 Bit Zahlen a und b zeigt Abbildung 4.2.

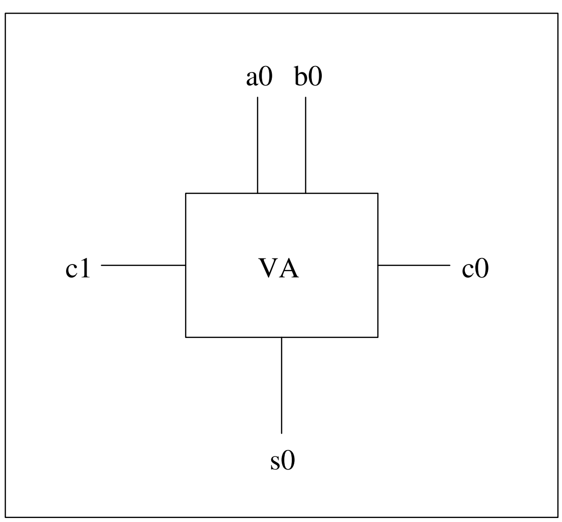


Abbildung 4.2 : 1 Bit Volladdierer

Die Funktion der Volladdierers kann wie folgt beschrieben werden. Das Summenbit ist immer dann gleich 1, wenn die Anzahl der Einsen der drei Operanden a_i b_i c_i ungerade ist.
Das Übertragsbit ist immer dann gleich 1, wenn mindestens zwei der drei Operanden Eins sind. Die Logikfunktion des Volladdierers kann aus einer entsprechenden Logiktabelle in disjunktiver Normalform, bestehend aus „und , oder , nicht“ Verknüpfungen umgesetzt und dann weiter mit Hilfe der xor (⊕) Funktion minimiert werden.
Diese Schritte sind in Gleichung 4.2.1 illustriert.

$$
\begin{aligned}
S_i &= a_ib_i'c_i' \vee a_i'b_ic_i' \vee a_i'b_i'c_i \vee a_ib_ic_i &&= a_i \oplus b_i \oplus c_i \\
C_{i+1} &= a_ib_ic_i' \vee a_i'b_ic_i \vee a_ib_i'c_i \vee a_ib_ic_i &&= a_ib_i \vee a_ic_i \vee b_ic_i
\end{aligned}
\qquad (4.2)
$$

Eine aus Gleichung 4.2 resultierende Gatterrealisierung könnte wie in Abbildung 4.2 aussehen.

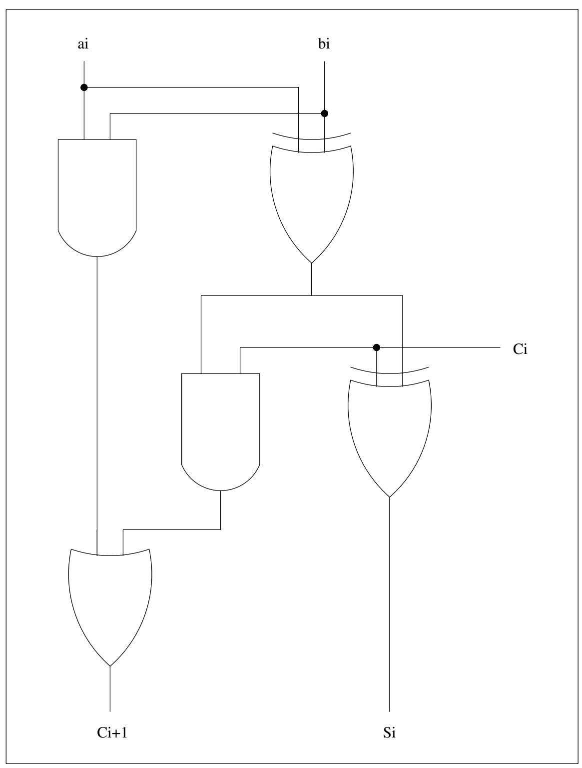


Abbildung 4.2: 1 Bit Volladdiererlogik

4.2.1 Serielle Addierer

Der Ablauf der Addition ähnelt einer handschriftlich vorgenommenen Addition von 2 Dualzahlen der Länge n. Der prinzipielle Aufbau einer seriellen Addierschaltung wird in Abbildung 4.2.1 gezeigt. Die erste zu addierende Zahl (1 Summand), ist im Schieberegister A gespeichert entsprechend befindet sich der 2 Summand im Schieberegister B. Die seriellen Ausgänge der Schieberegisters sind auf den Volladdierer geführt. Der Ausgang b des Volladdierers liefert Ergebnissignale, die in das Ergebnisschieberegister, bestehend aus D-Flip Flops (Speicherelemente), aufgenommen werden. Das Signal, das am Übertragsausgang des Volladdierers liegt wird mittels eines Flip Flop verzögert und bei der nächsten Addition mitbenutzt. Mit dem 1.Takt werden die beiden Bits der Wertigkeit 2^0 dem Volladdierer zugeführt. Am Ausgang erscheint das Ergebnis mit dem dem dazugehörigen Übertragsbit. Das Addierergebnis wird ins Ergebnisschieberegister übernommen, während das Übertragsbit im Flip Flop gespeichert wird.
Mit dem 2. Takt werden die beiden Bits mit der Wertigkeit 2^1 und das Übertragsbit auf die Eingänge des Volladdieres gegeben. Diese Prozedur wird n mal wiederholt. Das Ergebnis steht dann vollständig im Ergebnisschieberegister. Das Ergebnisschieberegister kann eingespart werden indem die Ergebnisbits in das Schieberegister A „eingetaktet“ werden, da die Bitpositionen, die ganz links stehen, nicht mehr benötigt werden. Diese Realisierung, die sehr platzsparend ist, benötigt jedoch für ein n-Bit langes Ergebnis auch n Takte und ist somit nur für langsame Additionen geeignet.

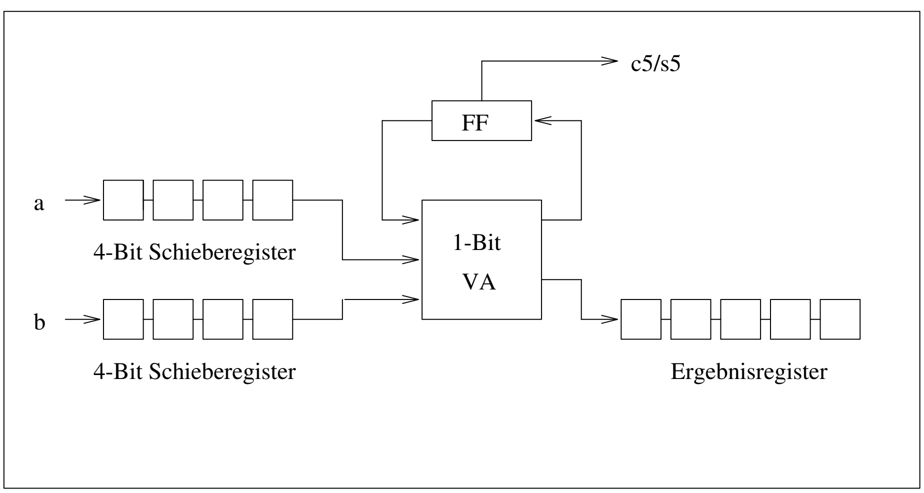


Abbildung 4.2.1: 4 Bit serieller Addierer

4.2.2 Addierer mit sequentieller Berechnung von Überträgen (Ripple Carry)

Anders wie beim seriellen Addierer braucht man hier um zwei mehrstellige Dualzahlen addieren zu können, für jede Stelle einen Volladdierer. Bei der niedrigsten Stelle kommt man mit einem Halbaddierer aus, da kein Übertrag von einer niedrigeren Stelle existiert. Eine Schaltung die sich zur Addition zweier 8 Bit Zahlen a und b eignet ist in Abbildung 4.2.2 dargestellt.
Die Rechenzeit des Gesamtaddierers ist wesentlich größer als die der Einzelstufen, denn der Übertrag c_8 kann erst dann einen gültigen Wert annehmen wenn sich c_7 auf einen gültigen Wert eingestellt hat. Dasselbe gilt für die vorhergehenden Überträge.

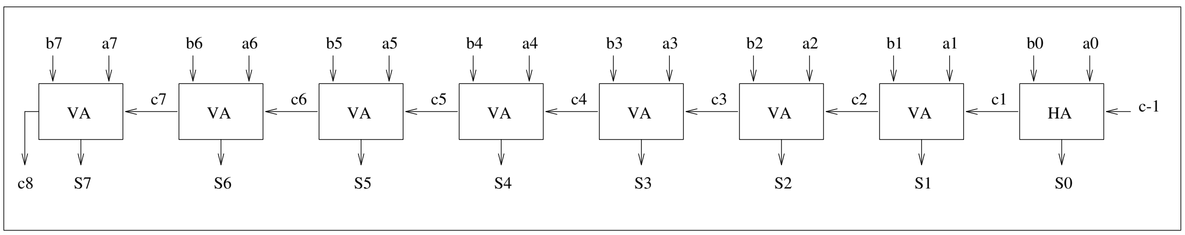


Abbildung 4.2.2 : 8 Bit Ripple Carry Addierer

4.2.3 Addierer mit paralleler Übertragslogik (Carry look-ahead)

Um die Rechenzeit bei der Addition zu verkürzen, kann man eine parallele Übertragslogik (PCL) verwenden. Bei dieser Methode werden alle Überträge direkt aus den Eingangsvariablen berechnet. Die hierfür benötigten Hilfssignale für die Übertragsfunktion können aus der Wahrheitstafel eines Volladdierers abgeleitet werden.
Für den Übertrag aus der Stufe i gilt die allgemeine Beziehung

$$C_{i+1} = a_i b_i \vee (a_i \oplus b_i)\ c_i \qquad (4.2.3)$$

Wobei die abgeleiteten Hilfssignale für die Übertragungsfunktion sind:

$$\begin{aligned} g_i &= a_i \wedge b_i && \text{Carry generate} \\ p_i &= a_i \oplus b_i && \text{Carry propagate} \qquad (4.2.4) \end{aligned}$$

[1] **P**arallel **C**arry **L**ogic (PCL)

Die Größe g_i gibt an ob in der Stufe ein Übertrag aufgrund der Eingangskombination a_i b_i erzeugt wurde. Man bezeichnet sie als Generate Variable (g_i).Die Größe p_i gibt an, ob aufgrund der Eingangskombination ein Übertrag, der von der nächstniedrigsten Stelle kommt, weitergegeben oder absorbiert wird. Sie wird deshalb als Propagate Variable bezeichnet. Aus Gleichung 4.2.3 erhält man durch fortlaufendes Einsetzen von z.B. C_1 in C_2 u.s.w folgende Übertragsgleichungen für die parallel berechneten Carrys.

$$\begin{aligned}
C_1 &= g_0 \vee p_0 c_0 \\
C_2 &= g_1 \vee p_1 c_1 = g_1 \vee p_1 g_0 \vee p_1 p_0 c_0 \\
C_3 &= g_2 \vee p_2 c_2 = g_2 \vee p_2 g_1 \vee p_2 g_1 g_0 \vee p_2 p_1 p_0 c_0 \\
C_4 &= g_3 \vee p_3 c_3 = g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0 \vee p_3 p_2 p_1 p_0 c_0 \\
&\;\vdots
\end{aligned} \tag{4.2.5}$$

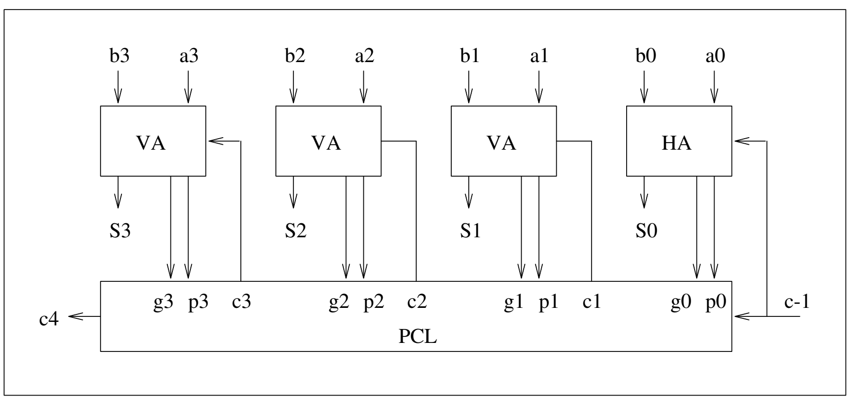


Abbildung 4.2.3 : Carry Look Ahead Addierer

Abbildung 4.5.4 zeigt das Blockschaltbild eines 4 Bit Addierers mit paralleler Übertragslogik Die Gleichungen aus Abbildung 4.2.3 sind im Übertragsberechnungsblock PCL realisiert. Addierer mit mehr als 4 Stellen kann man durch Aneinanderreihen mehrerer 4 Bit Blöcke realisieren, wobei der Übertrag c_4 dann als c_0 an den nächst höheren Block anzuschließen ist. Abbildung 4.2.4 zeigt eine mögliche Realisierung.

Eine Beschränkung auf 4 Bit Blöcke ist vorteilhaft, da für eine technische Realisierung mit mehr als 4 Bit Blöcke eine mehrstufige Realisierung nötig wäre, da aus technischen Gründen Gatter meist nicht mehr als 4 Eingänge haben. Dieses bedeutet jedoch längere Durchlaufzeiten der Signale, da mehr Gatter durchlaufen werden müssen. Der Zeitvorteil, den man durch das parallele Berechnen der Überträge gewinnt, würde damit mindestens wieder kompensiert werden. Als nachteilig stellt sich ferner heraus, daß der Übertrag innerhalb der Blöcke zwar parallel berechnet wird, von Block zu Block jedoch sequentiell verarbeitet wird.

Um eine noch kürzere Rechenzeit zu erhalten, müßte man wie folgt auch die Überträge von Block zu Block parallel verarbeiten. Dazu noch einmal die Beziehung aus Gleichung 4.2.5

$$C_4 = g_3 \vee p_3g_2 \vee p_3p_2g_1 \vee p_3p_2p_1 \vee p_3p_2p_1p_0c_0 \qquad (4.2.6)$$

Um diesen Nachteil auszugleichen werden wie folgt neue Block-Generate und Block-Propagate Variablen eingeführt, die wie folgt berechnet werden.

$$\begin{aligned} G &= g_3 \vee p_3g_2 \vee p_3p_2g_1 \vee p_3p_2p_1g_0 \\ P &= p_3p_2p_1p_0c_0 \end{aligned} \qquad (4.2.7)$$

Vereinfacht erhält man dann

$$C_4 = G \vee Pc_0 \qquad (4.2.8)$$

Man braucht in den einzelnen 4 Bit Additionsblöcken nur die zusätzlichen Hilfsvariablen G und P zu bilden und kann dann mit dem selben Algorithmus wie er in Gleichung 4.2.5 für die Überträge von Stelle zu Stelle angegeben wurde, die Überträge von Block zu Block berechnen. Nach diesem Prinzip lassen sich aus 4 Bit (n Bit) PCL Blöcken baumartig organisiere Carry Look Achead Addierer beliebiger Bitbreite bilden. So ergibt sich bsw. Für 16 Bit die in Abbildung dargestellte Struktur.

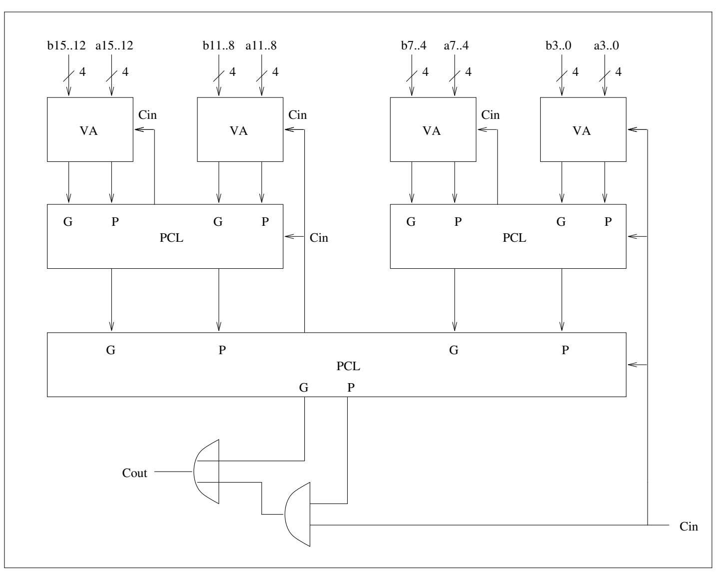


Abbildung 4.2.4 : 4 Bit Block Kaskadierung auf 16 Bit

4.3 Hierarchische Addierstrukturen

Insbesondere für große Wortbreiten n ist die vorgestellte Carry Look Ahead Technik nicht sehr effizient, auch gibt es noch optimalere Strukturen, die nicht durch aufwendige Übertragsberechnung die Additionsdauer verkürzen, sondern andere Beschleunigungsmöglichkeiten ausnutzen.

4.3.1 Carry Select Addierer

Grundkonzept dieses Addierers ist es alle möglichen Summen- und Carrybits in einem Schritt zu berechnen und später das Carrybits zur Selection der korrekten Summenbits zu verwenden. Abbildung 4.7.1 zeigt eine einzelne 4 Bit Carry Select Zelle die jedoch je nach Operandengröße mehrfach instanziert werden kann. Der genaue Ablauf kann wie folgt erläutert werden.
Es werden parallel alternative Ergebnisse für den Fall daß das Carrybit 0 und 1 ist berechnet. Diese Berechnung kann in zwei einfachen Carry Look Ahead Addierer wie in Abbildung 4.7.2 geschehen (siehe auch Carry Look Ahead Addierer Abschnitt 4.1.5). Anschließend wird in einem einstufigen oder auch mehrstufigem hierarchischen Verfahren das korrekte Ergebnis (Summenbits) mit Hilfe der vorberechneten Carry 0 bzw. 1 Bits mit einem Multiplexer selektiert [Bedrij62] [Pirsch96].

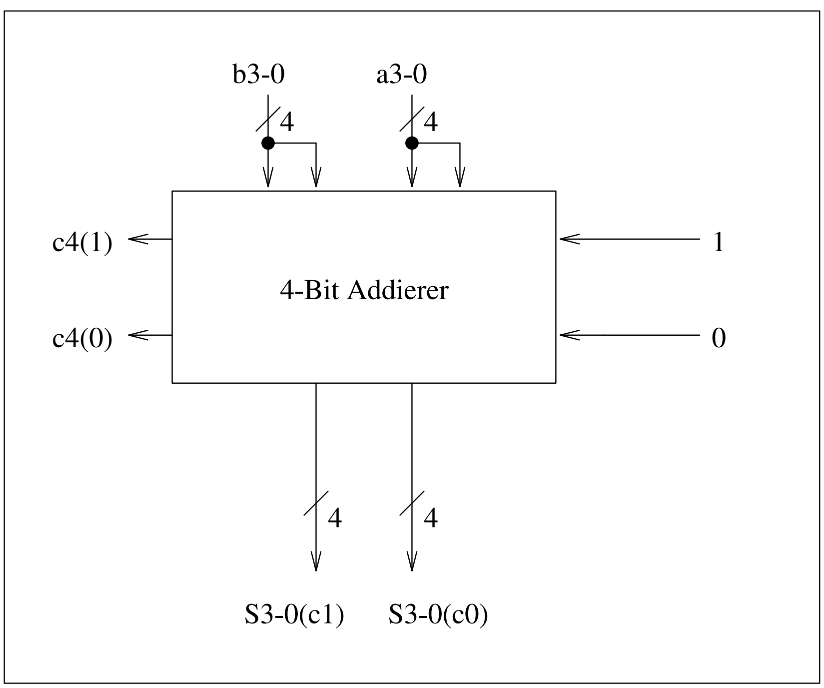
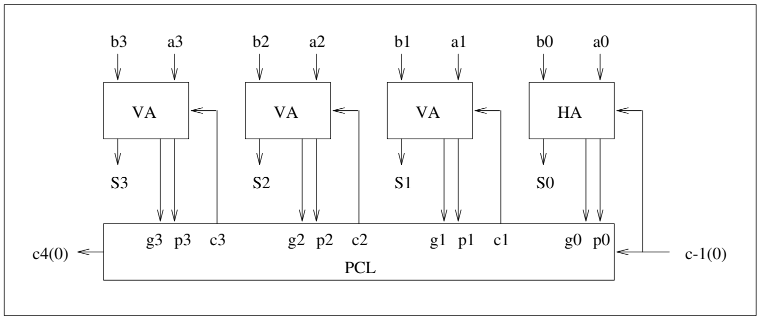


Abbildung 4.3.1 : 4 Bit Carry Select Addierzelle



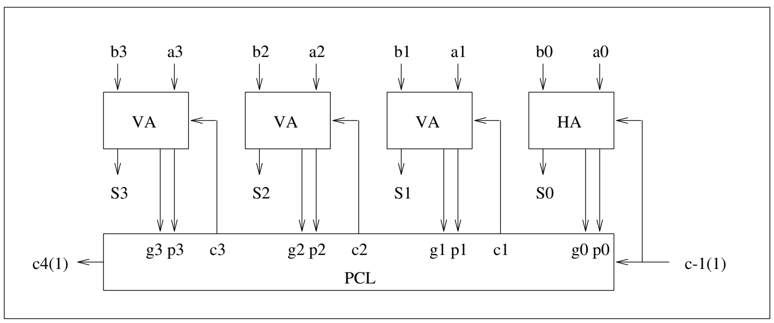


Abbildung 4.3.2 : innerer Aufbau einer möglichen Carry Select Addierzelle mit Carry Look Ahead

Eine mögliche Gesamtrealisierung eines 8 Bit Carry Select Addierers zeigt Abbildung 4.3.3.
Eine einzelne 4 Bit Addierzelle verarbeitet Bitgruppen der Länge m = 4. Für die Alternativen Carry Input 0 bzw. 1 werden Summenbits und Ausgangscarry berechnet. Multiplexer zur Auswahl der korrekten Ergebnisse werden, für höhere 4 Bit Addierer (Gruppen), von einer Carry Select Logik (CS) gesteuert. Die CS-Logikfunktion lautet für die (k+1)te Gruppe:

$$\text{CS (k+1)}: \quad c_{(k+1)m} = /c\text{-}1 \wedge c_{(k+1)m}(0) \vee c\text{-}1 \wedge c_{(k+1)m}(1) \qquad (4.3.1)$$

wobei $c_{(k+1)m}(0)$ die ermittelte Carryfunktion für $c_{km} = 0$ und $c_{(k+1)m}(1)$ für $c_{km} = 1$ ist.

/ in der Gleichung 4.7.1 bedeutet eine Negation des Einganges c-1

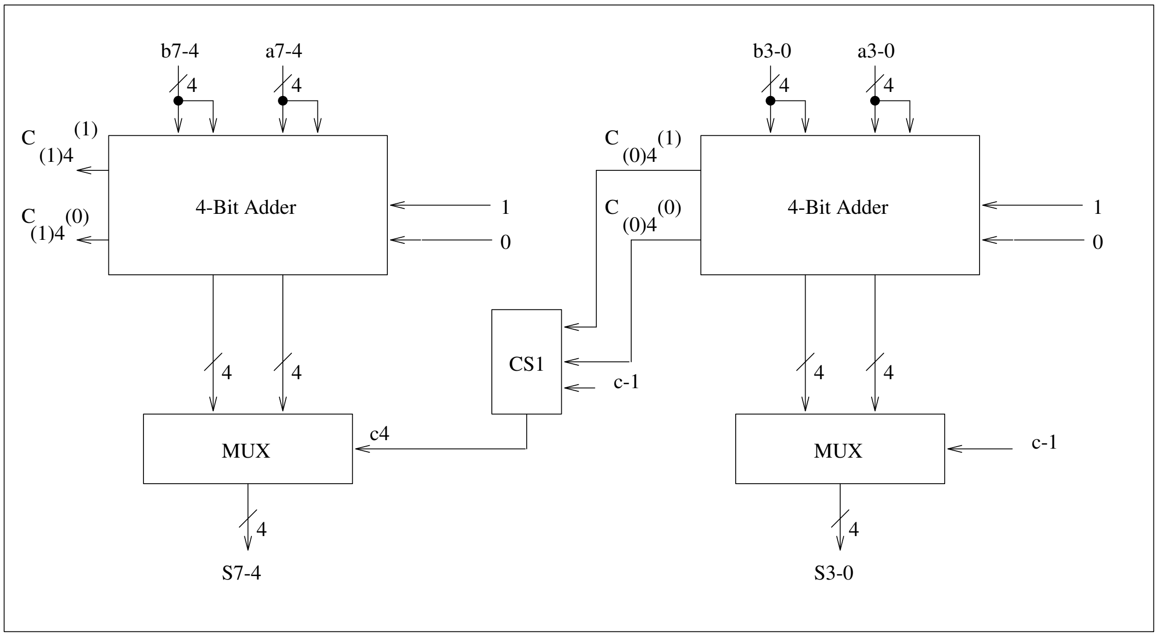


Abbildung 4.3.3 : 8 Bit Carry Select Addierer

Es werden ähnlich der Block Carry Funktion (vergl. Abschnitt 4.2.3 Carry Look Ahead Addierer) entsprechende Gruppen für neue Carry Funktionen definiert

$$c_{(k+1)m} = GG_k \vee GP_k\, c_{km} \tag{4.3.2}$$

mit GG_k als Gruppen Generate und GP_k als Gruppen Propagate der k-ten Gruppe durch m-fache Operation auf die zugehörigen Generate und Propagate Signale ermittelt. Ein Einsetzen von CS-Logikfunktionen der darunterliegenden Bitgruppen führt auf eine Logikfunktion, die nur von den Carry Ausgängen der Addierer abhängt. Der zeitkritische Pfad geht über den Addierer der untersten Bitgruppen, die Carry Select Logic (CS) für den Multiplexer der höchsten Bitgruppe und den zugehörigen Multiplexer. Der Carry Save Addierer ist eine Verallgemeinerung eines Conditional Sum Addierers, der im nächsten Abschnitt besprochen wird.

4.3.2 Pyramiden Addierer

Eine andere Art von leicht erweiterbaren Addierern ist ein hierarchisch aufgebauter Pyramiden Addierer. Dieser besteht aus mehreren hintereinandergeschalteten Verknüpfungsstufen (Hierarchistufen), die jeweils bestimmten Aufgaben zugeordnet sind. Abbildung 4.3.2 zeigt einen 8 Bit Pyramiden Addierer mit 4 Hierarchiestufen.

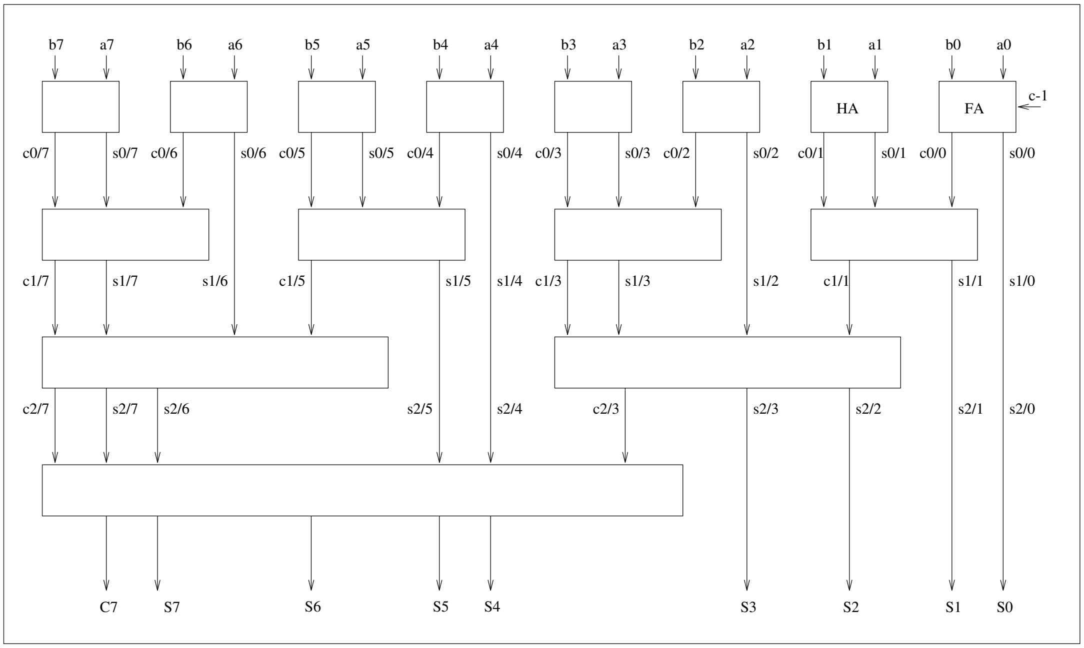


Abbildung 4.3.2 : 8 Bit Pyramiden Addierer

Vereinfacht kann man sagen, daß er aus zwei Hauptteilen, die hintereinander geschaltet sind, besteht. Der erste Teil aufgebaut aus Halbaddierern erzeigt Partialsummen und Partialcarry Bits für alle Stufen, der andere mit einer pyramidartigen Struktur, die Partialcarrys mit dem Partialsummen aufnimmt und verknüpft. Die einzelnen Stufen sind hierbei wie folgt aufgebaut:

Stufe0:
Erzeugung von Partialsummen und Partialcarrys der eingehenden Summanden. Für die Partialsummen gilt $S_{0/i} = a_i \oplus b_i$, für die Partialcarrys $C_{0/i} = a_i b_i$ für alles Stufen i = 1,2,...,n-1. Dieses entspricht der Logik eines normalen Halbaddierers. Ist ein Eingangscarry in der Stufe 0 bekannt, so kann an dieser Stelle ein Volladdierer eingesetzt werden. Die Partialsummen und Partialcarrys berechnen sich dann wie folgt (verlg. auch Abschnitt 4.1):

$$S_{0/0} = S_0 = (a_0 \oplus b_0) \oplus c_{-1} \quad \text{und}$$
$$C_{0/0} = C_0 = a_0 b_0 \vee (a_0 \oplus b_0) \wedge c_{-1} \qquad (4.3.2)$$

Stufe1:
Aus 3 Eingangswerten der Stufe 0 werden eine neue Summe und ein neues Carry berechnet.

$$S_{1/i+1} = S_{0/i+1} \oplus C_{0/i}\,, \qquad i = 0, 2, 4, \ldots, n-2$$
$$C_{1/i+1} = C_{0/i+1} \vee S_{0/i+1} C_{0/i} \qquad (4.3.3)$$

Stufe2:
4 Eingangswerte der Stufe 1 werden zu 2 Summenbits und 1 Carrybit zusammengefaßt.

$S_{2/i+1} = S_{1/i+1} \oplus C_{1/i}$ $\qquad i = 1, 5, 9, \ldots, n-3$
$S_{2/i+2} = S_{1/i+2} \oplus S_{1/i+1}C_{1/i}$
$C_{2/i+2} = C_{1/i+2} \vee S_{1/i+2}S_{1/i+2}C_{1/i}$ $\qquad (4.3.3)$

Stufe3:
Stufe 3 verknüpft dann alle 8 Eingangswerte a b.

$S_{3/i+1} = S_{2/i+1} \oplus C_{2/i}$ $\qquad i = 3, 11, 19, \ldots, n-5$
$S_{3/i+2} = S_{2/i+2} \oplus S_{2/i+1}C_{2/i}$
$S_{3/i+3} = S_{2/i+3} \oplus S_{2/i+2}S_{2/i+1}C_{2/i}$
$S_{3/i+4} = S_{2/i+4} \oplus S_{2/i+3}S_{2/i+2}S_{2/i+1}C_{2/i}$
$C_{3/i+4} = C_{2/i+4} \vee S_{2/i+4}S_{2/i+3}S_{2/i+2}S_{2/i+1}C_{2/i}$ $\qquad (4.3.4)$

Stufe k:
Weitere Stufen werden nach dem selben Schema aufgebaut, wobei sich die Gruppengröße der Eingangswerte in jedem Schritt verdoppelt. Die Gruppengröße errechnet sich daher nach 2^k, während sich die Ausdrücke wie folgt ermitteln lassen.
Es gilt:

$i = 2^{k-1}-1 , 3*2^{k-1}-1 , 5*2^{k-1}-1 \ldots , n-1-2^{k-1}$

$S_{k/i+1} = S_{k-1/i+1} \oplus C_{k-1/i}$
$S_{k/i+2} = S_{k-1/i+2} \oplus S_{k-1/i+1}C_{k-1/i}$
.
.
.
$S_{k/i+2^{k-1}} = S_{k-1/i+2^{k-1}} \oplus S_{k-1/i+2^{k-1}-1} S_{k-1/i+2^{k-1}-2} \ldots S_{k-1/i+1}C_{k-1/i}$
$C_{k/i+2^{k-1}} = C_{k-1/i+2^{k-1}} \vee S_{k-1/i+2^{k-1}-1} S_{k-1/i+2^{k-1}-1} \ldots S_{k-1/i+1}C_{k-1/i}$ $\qquad (4.3.5)$
$S_{k/i} = S_{k-1/i}$ $\qquad$ für alle anderen i

Die Vorteile dieses Addierers sind seine Regelmäßigkeit und damit leichte Erweiterbarkeit und das Wegfallen jeglichen Weitergebens von Carry Signalen. Nachteilig sind besonders in der Praxis die Länge der Ausdrücke höherer Stufen und die damit verbundene technische Realisierung.

4.3.3 Conditional Sum Addierer

Auch in Stufen aufgebaut ist der schon in Abschnitt 4.2.1 erwähnte Conditional Sum Addierer eine Alternative zum Pyramiden Addierer, der eine ähnliche Struktur und Verzögerungszeiten besitzt. Abbildung 4.3.3 zeigt einen 8 Bit Conditional Sum Addierer [Slansky60] [Omondi 94]. Das Grunddesign dieses Addierers ist alle möglichen Summen- und Carrybits in einem Schritt zu generieren, um dann mit den Carrybits mit Hilfe eines Multiplexerstruktur in höheren Leveln die richtigen Summenbits zu selektieren. Dieses Selektieren benötigt dabei jedoch mehrere Schritte, bis das Gesamtergebnis an den letzten Stufen des Addierers anliegt. Bei diesem Addierer ergeben sich ähnliche Verzögerungszeiten wie bei dem Pyramiden Adder. Der Unterschied zum Carry Select Addierer besteht nur darin, daß der Conditional Sum Addierer nur 2 Gruppen zu einer neuen Stufe zusammenfaßt, während der Carry Select Addierer wie in Abschnitt 4.3.1 gezeigt zumindest in der untersten Stufe direkt 4 Bit Gruppen bildet. Zu beachten sind die anderen Bezeichnungen der Leitungen gegenüber dem Pyramiden Addierer (vergl. Abbildung 4.3.2).

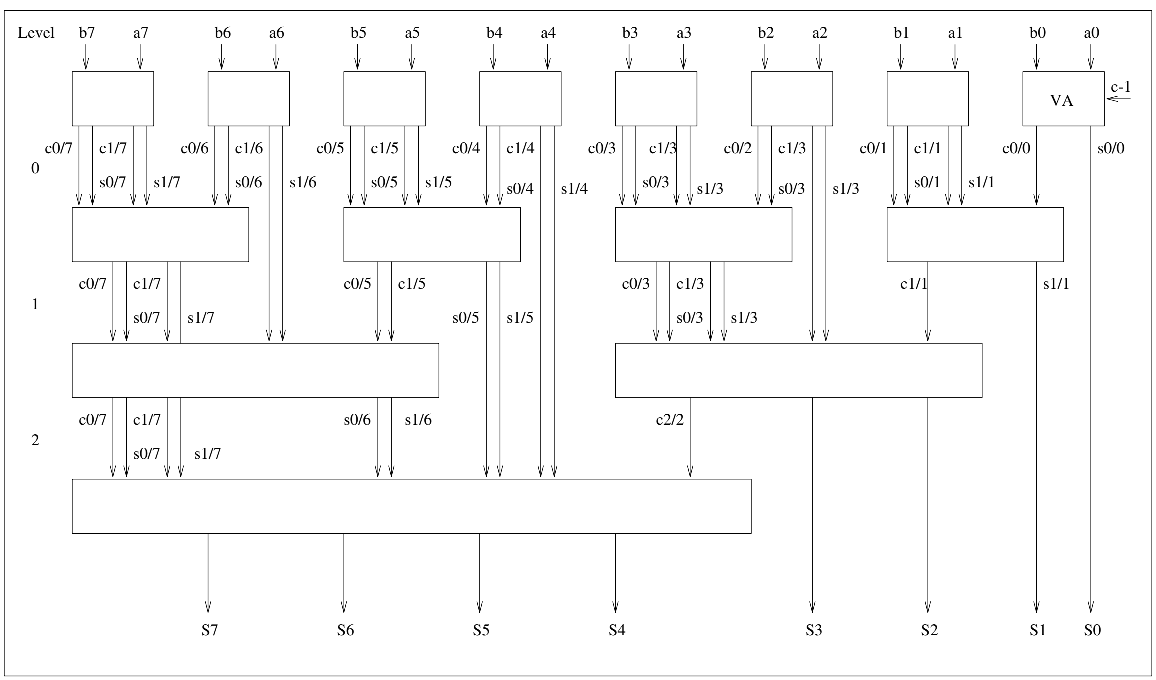


Abbildung 4.3.3 : 8 Bit Conditional Sum Addierer

4.3.4 Carry Save Addierer

Kennzeichen der bisher behandelten bitparalleler Addierer, die unter dem Begriff Carry Propagate Addierer (CPA) zusammengefaßt werden, war, daß die Carrysignale zu Bestimmung der Summe ausgewertet werden. Dies bedeutet, daß je nach Addierstruktur die Carrysignale entweder sequentiell die Logikblöcke von der niederwertigsten zur höchstwertigen Stufe durchlaufen oder auch entlang einer Baumstruktur weitgehend parallel zur Summenwertermittlung verwendet werden. Eine andere Art von Addierern, die sich insbesondere für die Addition einer größeren Zahl von Operanden eignet, ist die Verwendung von Carry Save Addierern (CSA). Hierbei werden die Übertragssignale nicht für die aktuelle Addition eingesetzt, sondern bei nachfolgenden Additionen verwendet. Abbildung 4.9.1 zeigt einen n-Bit Carry Save Addierer, der aus n Volladdieren besteht. Es werden drei Operanden X, Y und Z mit n Summenbits S_i und n Übertragbits C_{i+1} zusammengefaßt. Die Summenbits bilden einen Summenvektor S und die Übertragsbits einen Carry Vektor C, wobei der Carry Vector in der Wertigkeit um eine Stelle verschoben ist. Erst die erneute Addition von Summenvektor S und Carryvektor in einem Carry Propagate Addierer (Vektor Merge Addierer) liefert das vollständige Ergebnis. Diese Addierart eignet sich besonders für die Addition zahlreicher Operanden und wird deshalb besonders bei Multiplikationsverfahren, wie später noch gezeigt wird, angewendet.

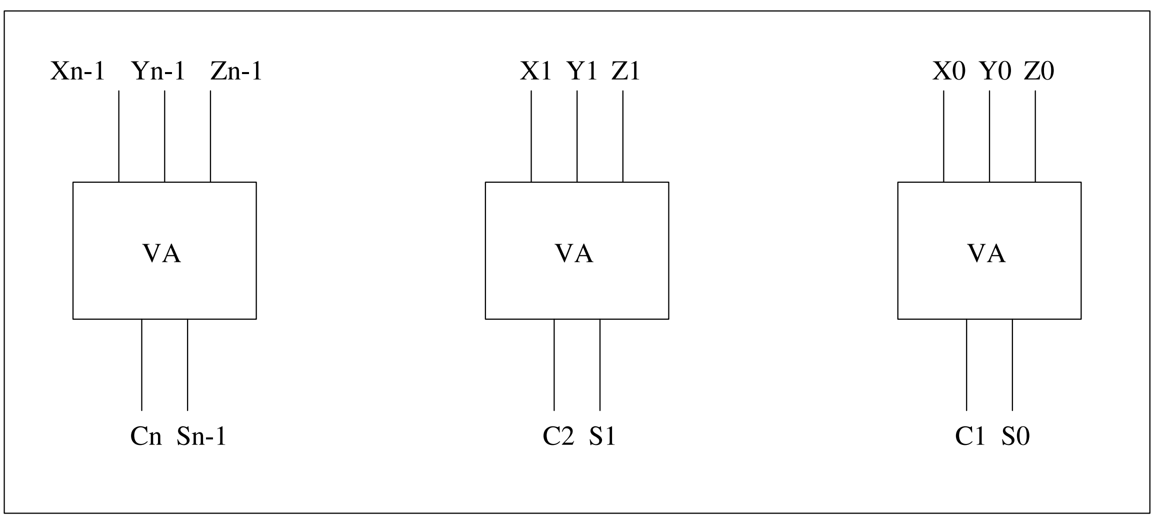


Abbildung 4.3.4 : n Bit Carry Save Addierer

5 Schnelle Multiplizierstrukturen

5.1 Multiplizierer

Neben der Addition ist die Multiplikation eine häufig benutzte Basisoperation in der digitalen Signalverarbeitung. Zur Erzielung hoher Durchsatzraten werden schnelle Multiplizierer benötigt, die bedingt durch ihre Anordnung von Basiszellen und Komplexität, jedoch einen erhöhten Hardwareaufwand im Bezug auf Chip- und Verdrahtungsfläche zur Folge haben. Da eine hohe Signalverarbeitungsleistung im Vordergrund steht wird dieses jedoch gerne in Kauf genommen. Nachfolgend werden verschiedene Möglichkeiten zur Realisierung von Multiplizierern mir kurzen Multiplikationszeiten für positive Zahlen vorgestellt. Im zweiten Abschnitt dann Multiplizierer für die Multiplikation vorzeichenbehafteter Zahlen in Zweierkomplement.

5.1.1 Multiplizierer für positive Zahlen

Durch die Multiplikation zweier nichtvorzeichenbehafteter Zahlen, bestehend aus a als Multiplikand und b als Multiplikator, entsteht das Produkt P.

$$P = a * b \quad \text{oder} \quad V(P) = V(a) * V(b) \tag{5.1.1}$$

Sei a eine ganzzahlige positive m-bit und b eine ganzzahlige positive n-bit Zahl. Das resultierende Produnkt P benötigt dann eine Zahlendarstellung mit m+n Bits. Mit der Beziehung (3.1.1) kann durch Einsetzen des Wertes von b das Produkt als Funktion der einzelnen Bits b_j des Multiplikators dargestellt werden. Die Teilprodukte werden hiernach gebildet, indem die einzelnen Bits a_i b_j „verundet“ werden, da der boolesche Ausdruck und einen 1 Bit Multiplizierer darstellt.

$$V(P) = V(a) \sum_{j=0}^{n-1} b_j 2^j \tag{5.1.2}$$

Nach Gleichung 5.1.1 sind ferner die Teilprodukte von a mit den Bits b_j von b stellenwert-verschoben zu addieren. Das Teilprodukt $V(a)b_j$ ist für $b_j = 0$ gleich 0 sonst gleich V(a). Diese, auch bei der Berechnung per Hand angewandte Methode, kann auch als Hardware-implementierung benutzt werden (sequentielle Multiplikation mit Hilfe von ALUs[1]). Durch Auflösung der Zahl a in ihre Bits a_i kann das Produkt P als Summe von Teilprodukten auf Bit-ebene dargestellt werden.

[1] **A**rithmetsche **L**ogische **E**inheit (ALU)

$$V(P) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (a_i b_j) 2^{i+j} \tag{5.1.3}$$

Als Beispiel ist die Bildung der Produktbits p_k mit Hilfe der Teilprodukte und deren Summation für m = n = 4 Bits dargestellt.

				a_3	a_2	a_1	a_0	
				a_3b_0	a_2b_0	a_1b_0	a_0b_0	$\leftarrow b_0$
			a_3b_1	a_2b_1	a_1b_1	a_0b_1		$\leftarrow b_1$
		a_3b_2	a_2b_2	a_1b_2	a_0b_2			$\leftarrow b_2$
	a_3b_3	a_2b_3	a_1b_3	a_0b_0				$\leftarrow b_3$
	=	=	=	=	=	=	=	
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

Abbildung 5.1.1 : Partialproduktmatrix zur Darstellung der Operationen bei der binären Multiplikation positiver Zahlen

5.1.2 Ripple Carry Array Multiplizierer

Aus der Beziehung 5.1.3 und dem Abbildung 5.1.1 kann direkt eine Multiplikationsrealisierung als n-fache Addition abgeleitet werden. Durch eine reguläre Arrayanordnung von „und“ Gattern zur Teilproduktbildung und Addierzellen kann eine Produktbildung realisiert werden. Abbildung 5.2.1 zeigt einen 4 Bit Array Multiplizierer auf der Basis von Ripple Carry Addierern. Hierbei beinhaltet jeder Multiplizierzelle sowohl die Teilproduktbildung, als auch die Summation. Die hierfür genötigten Funktionen sind gegeben durch

$$\begin{aligned} s &= (ab) \oplus s' \oplus c' \\ c &= abs' \vee abc' \vee s'c' \end{aligned} \tag{5.1.2}$$

Dabei sind a, b Bits der Operanden A und B, s’ das Summenbit und c’ das Übertragsbit einer Vorgängerzelle. Als Ergebnis werden das Summenbit s und das Übertragsbit c an nachfolgende Zellen weitergegeben.

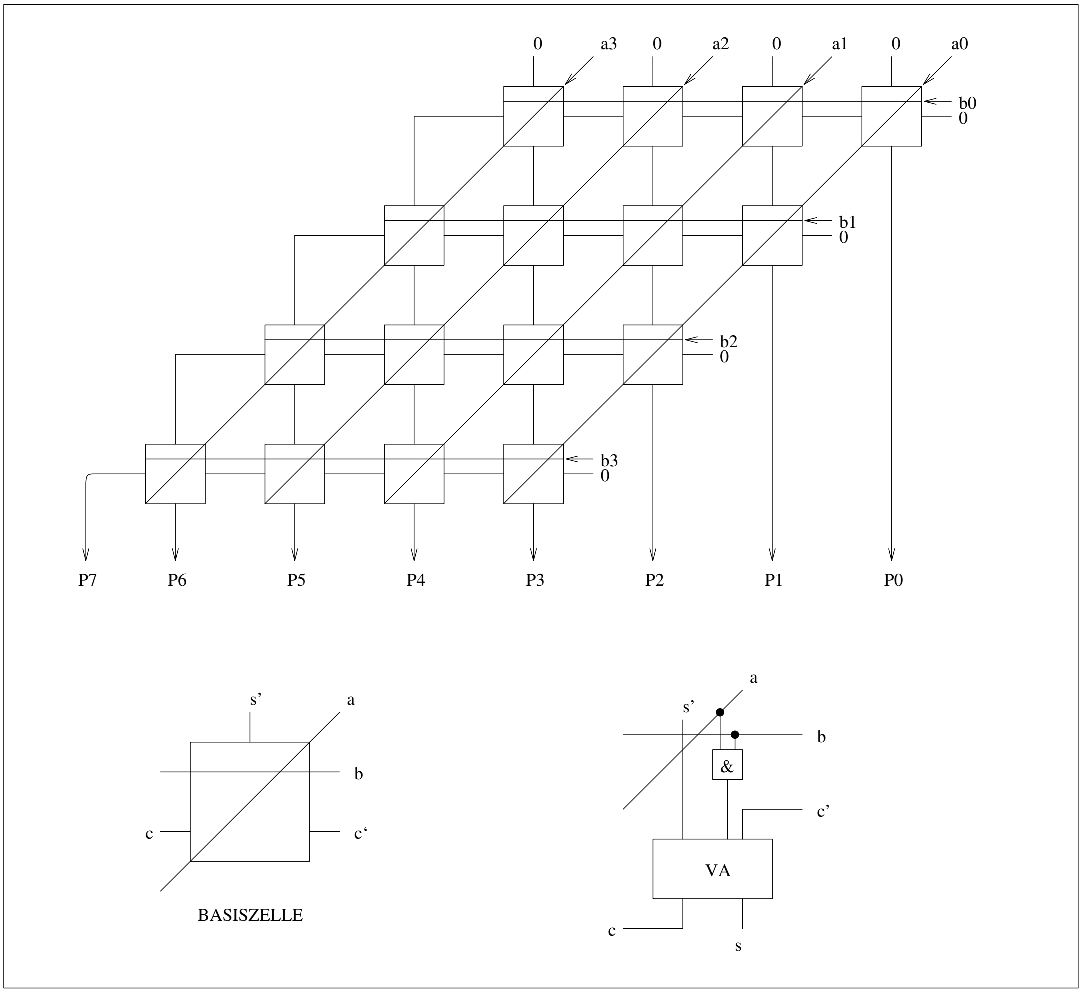


Abbildung 5.1.2 : 4 Bit Ripple Carry Array Multiplizierer mit der Basiszelle und ihrem Aufbau

In Abbildung 5.1.2 werden die Übertragsbits horizontal und die Summenbits vertikal transportiert. An bestimmten Stellen des Arrays kann die Multiplizierzelle abgeändert werden, sofern s' = 0 oder c' = 0 sind. Dann vereinfacht sich der Volladdierer VA zu einem Halbaddierer HA[1]. Sind sowohl s' und c' = 0, dann entfällt die Addierzelle vollständig. Der Gesamtaufwand des Array Multiplizierers besteht aus:

[1] Halbaddierer (HA) sind Addierzellen mit 2 statt 3 Eingangsbits. Es existiert kein Carryin Eingang einer vorherigen Zelle (siehe auch Abschnitt 4.1.2).

UND	:	nm
HA	:	n
VA	:	nm - n - m

Nachteilig bei dieser Multiplizierrealisation ist, daß die Carrys waagerecht in jeder Stufe weitergegeben werden müssen, was einem Carrybitrippeln entspricht und somit Zeit kostet, da die Addierzelle auf das ankommende Carrybit erst „warten“ muß. Vorteilhaft erscheint jedoch, daß diese Multiplizierstruktur leicht auf größere Operandenvektoren durch Mehrfachinstanzierung der Basiselemente erweiterbar ist, da sie sehr regulär aufgebaut ist [Pirsch96] [Koren93].

5.1.3 Carry Save Multiplizierer

Für die Addition mehrerer Operanden bietet sich auch die Carry Save Technik an (siehe auch Kapitel 4.2.3). Abbildung 5.1.3 zeigt einen 4*4 Bit Carry Save Multiplizierer. Die Übertragssignale sind dabei jeweils den Addierern der nachfolgenden Bitebene zugeführt. Der abschließende m-bit Addierer (ADD) übernimmt die Funktion des Vector Merge Addierers, der den Summen- und Carryvektor zusammenfaßt (vergl. auch Abschnitt 4 Carry Save Addierer). Als schneller Addierer kann hier ein m-Bit Carry Propagate Addierer (CPA) eingesetzt werden. Vorteilhaft bei diesem Multiplizierer, im Gegensatz zu dem Ripple Carry Array Multiplizierer ist, daß keine Carrybits waagerecht von Zelle zu Zelle „gerippelt“ werden müssen. Somit ergibt sich ein Geschwindigkeitsvorteil, der um die Größe des ersten Operanden besser ist. Leicht zu Übersehen ist jedoch, daß der abschließende CPA Addierer zusätzlich aus mehreren Logikstufen besteht und damit das Endergebnis weiter verzögert wird. Aus diesem Grund ergibt sich kein großer Geschwindigkeitsgewinn. Wird der abschließende Addierer als Ripple Carry Addierer realisiert, so wird dieser Multiplizierer als Braun Array Multiplizierer bezeichnet. Siehe dazu Abbildung 5.1.4 a und b. Der Gesamtaufwand ist identisch mit dem Ripple Carry Array Multiplizierer, da Anzahl und Typ der Logikelemente identisch sind. Die einzelnen Logikelemente sind nur anders angeordnet und verdrahtet. Der zeitkritische Pfad verläuft hier vom ersten Addierers rechts senkrecht nach unten und dann im abschließenden Addierer entlang des Übertragungspfades. Die Verzögerungszeit ist im Vergleich zum Ripple Carry Array Multiplizierer geringer. Ferner bietet der Braun Array Multiplizierer besondere Vorteile beim Einsatz von Pipelining [Pirsch96].

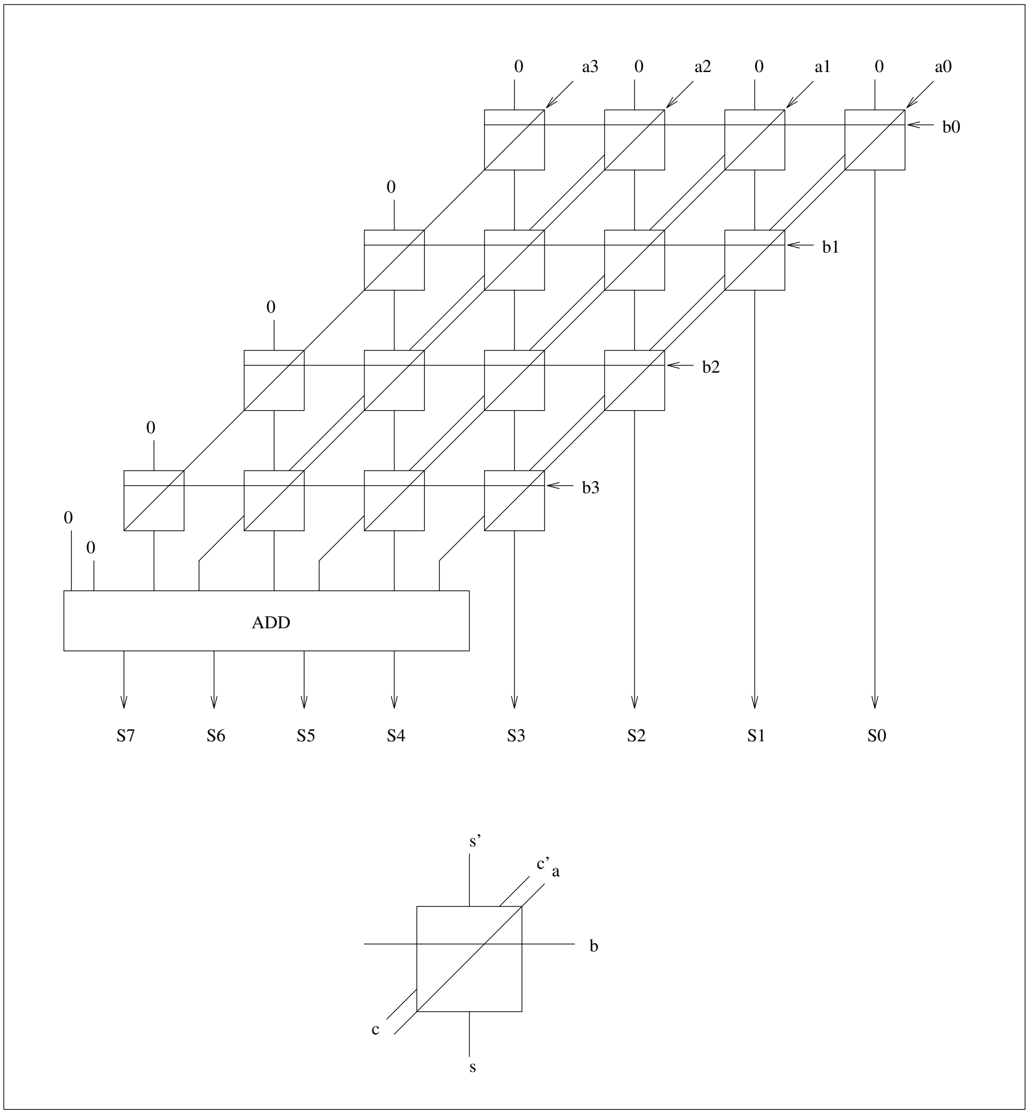


Abbildung 5.1.3 : 4 Bit Carry Save Addier Array mit Grundzelle

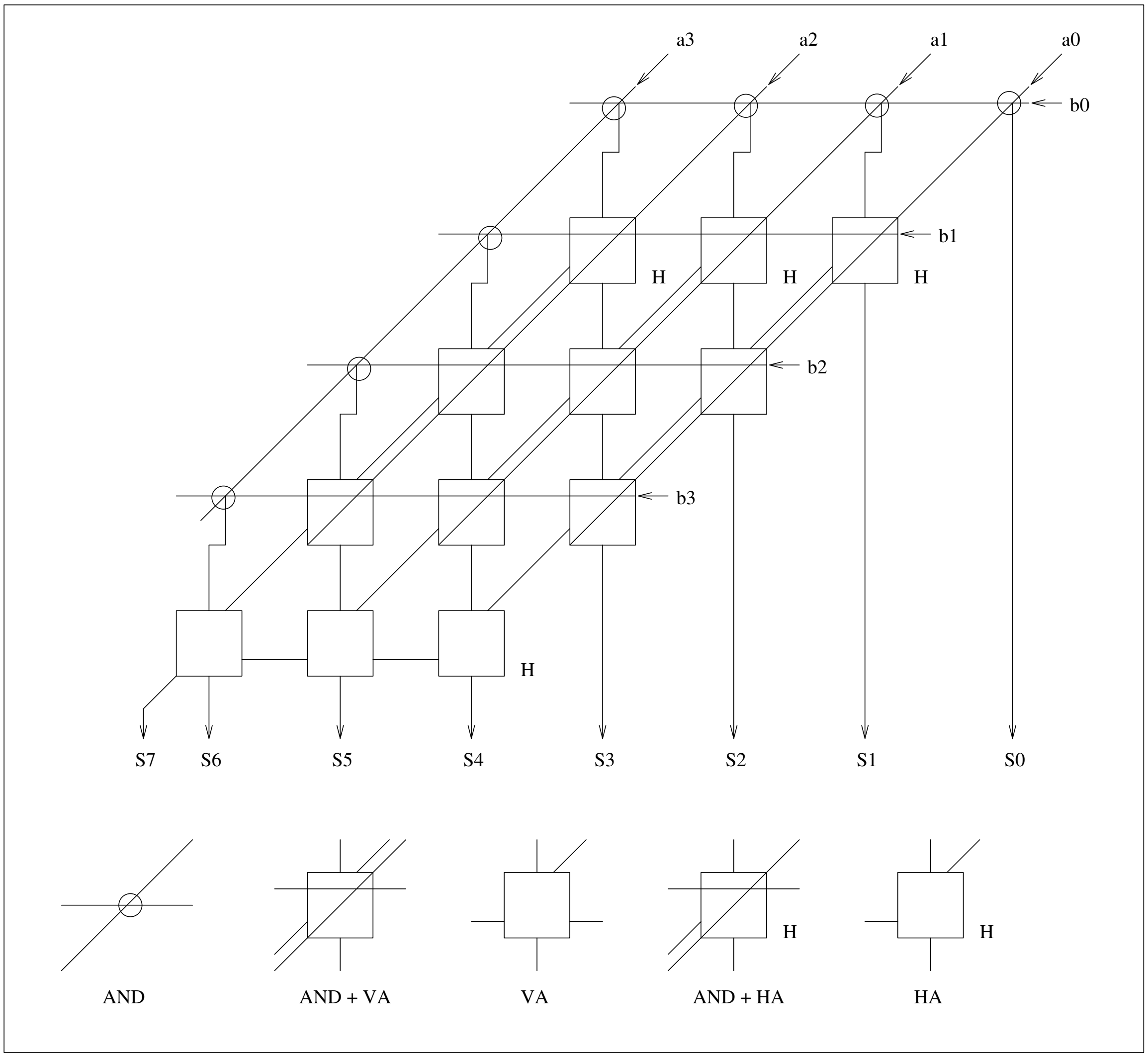


Abbildung 5.1.4 a,b : 4 Bit Braun Array Multiplizierer und die Grundzellen

5.1.4 Wallace Baum Multiplizierer

Eine weitere Möglichkeit zu den bisherigen Multiplikationsrealisierung stellt der Wallace Baum Multiplizierer dar. Abbildung 5.1.4 zeigt den prinzipiellen Aufbau eines solchen Multiplizierers. Bei diesem Multiplizierer werden in der ersten Stufe alle Teilprodukte parallel durch „und“ Verknüpfungen von Multiplikant und Multiplikator gebildet. Die Teilprodukte werden in Carry Save Addierbäumen (CSA) addiert. Die abschließend erhaltenen Carry und Summenvektoren werden dann in einen Carry Propagate Addierer (CPA) zusammengefaßt [Wallace64].

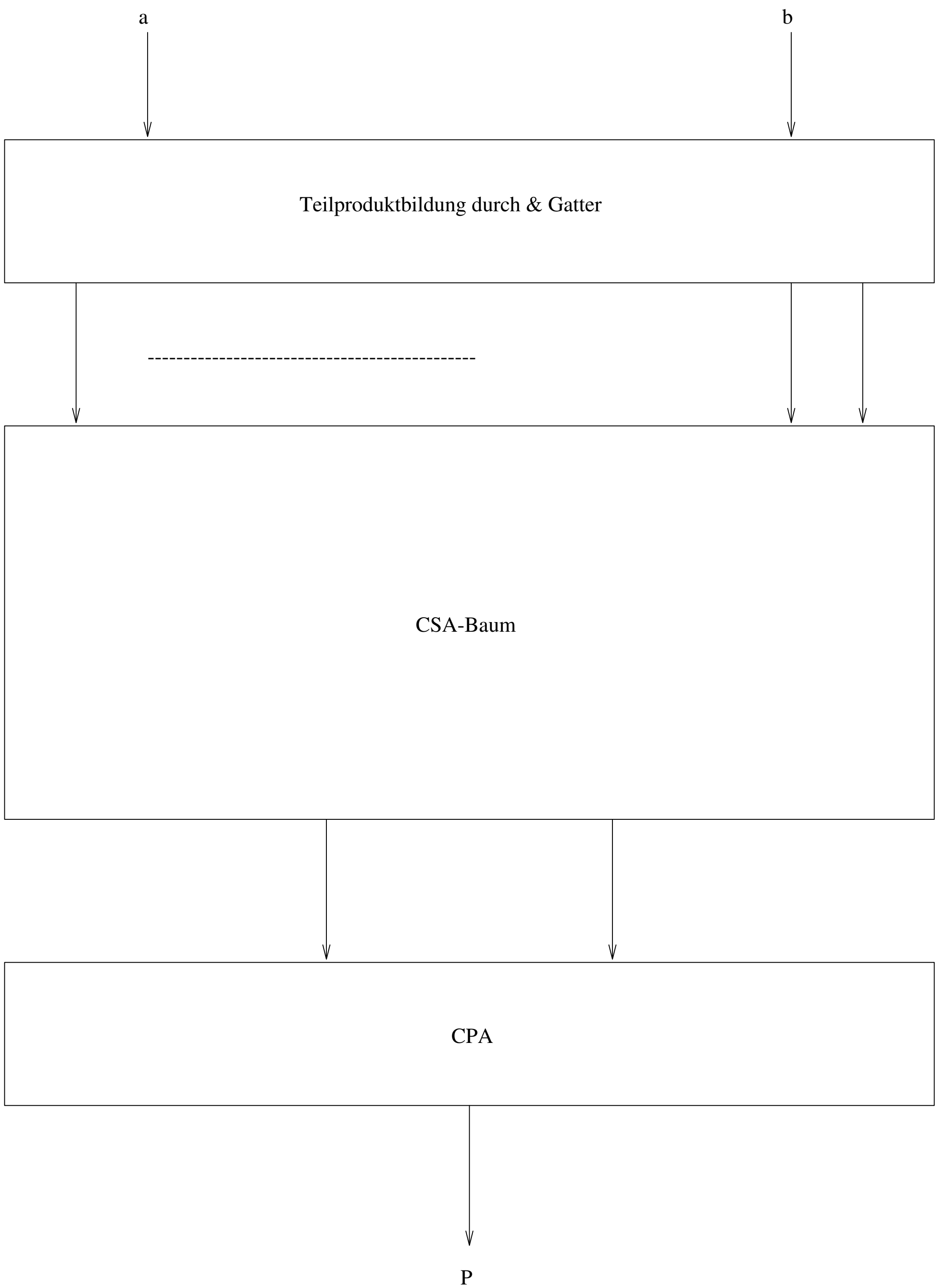


Abbildung 5.1.4 : Wallace Baum Struktur

Diese Grundstruktur kann dem Carry Save Array Multiplizierer zugeordnet werden. Zur schnellen Addition der Teilprodukte wird jedoch nicht eine Anordnung wie bei diesem verwendet (vergl. Abbildung 5.1.3), sondern eine Baumstruktur. Die CSA Addierer sind hierbei einfache Volladdierer, die jeweils 3 parallele Operandenbits zu einem Summen- und einem Übertragsbit zusammenfassen. Der Volladdierer kann als ein Zähler aufgefaßt werden, der 3 gleichwertige Operandenbits addiert (Zählung der Einsen) und in eine Dualzahl von 2 Bit (Summe, Carry) zusammenfaßt. Dies bedeutet eine Reduktion der parallel vorliegenden Operanden um maximal $^2/_3$ in der Hierarchiestufe des Addierbaumes.

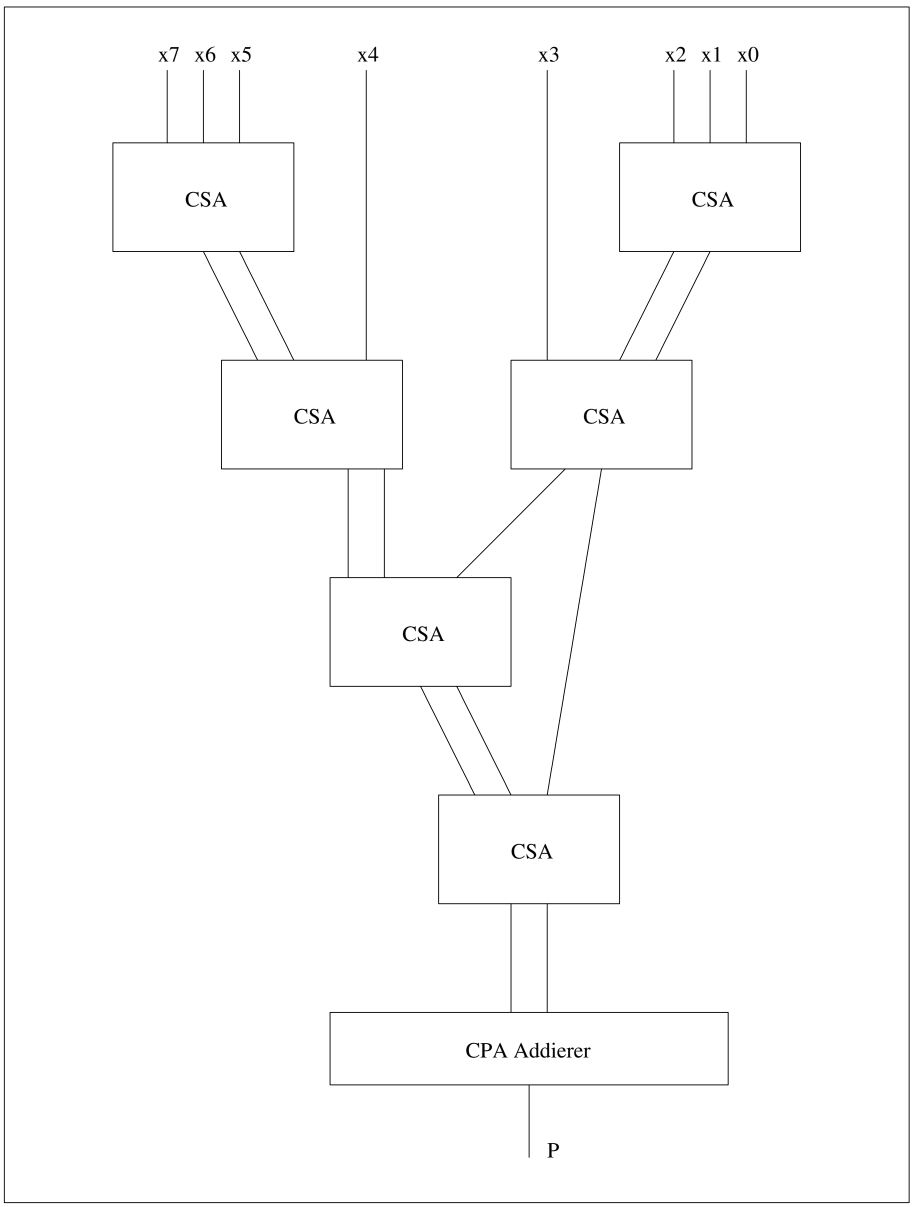


Abbildung 5.1.5 : m * 8 Bit Wallace Baum Multiplizierer

Eine mögliche Realisierung des Baumes für 8 Teilprodukte, für eine m * 8 Bit Multiplikation, aufgebaut aus CSA Addierern (vergl. auch dazu Abschnitt 4.2.3), zeigt Abbildung 5.1.5.
Vorteil dieses Addierers ist, sollte man eine derartige Baumstruktur einsetzen, die Möglichkeit viele Summationen gleichzeitig durchzuführen, um dann die einzelnen Zwischensummen am Schluß mit einem schnellen Addierer aufaddieren zu können. Dieses erhöht den Durchsatz enorm, da ein Parallelverarbeitung vorliegt. Nachteilig wirkt sich jedoch auch hier in jedem Fall der CPA Addierer am Schluß des Baumes aus, der wiederum mehrere Gatterdurchlaufzeiten hat [Spaniol76] [Omondi94].

5.2 Multiplizierer vorzeichenbehafteter Zahlen (2er Komplementzahlen)

5.2.1 Multiplizierer mit Pre- und Postkomplement

Eine Möglichkeit zwei Zahlen in Zweierkomplement zu multiplizieren besteht darin, die Beträge der beiden Zahlen zu multiplizieren und das Ergebnis entsprechend der üblichen Vorzeichenregeln in eine Zweierkomplement zu verwandeln. Es sei Multiplikand A und Multiplikator B 2 n-bit Zweierkomplementzahlen. Die Umwandlung in die Betragsdarstellung wird durch das Vorzeichen a_{n-1} bzw. b_{n-1} gesteuert. Im Falle $a_{n-1} = 1$ bzw. $b_{n-1} = 1$ ist das bitweise Komplement (1er Komplement vergl. Abschnitt 3.1.3) durchzuführen und eine 1 zu addieren. Das Produkt der Beträge $|a| * |b|$ erhält das Vorzeichen,

$$a_{n-1} \oplus b_{n-1} \tag{5.2.1}$$

das eine Komplementierung des Ergebnisvectors steuert. Eine Anordnung zur Multiplikation von Zweierkomplementzahlen mit diesem Schema ist in Abbildung 5.2.1 dargestellt. Hierbei führen die beiden Blöcke COMPL eine durch ein Mode Signal a_{n-1} b_{n-1} gesteuerte Komplementierung durch.

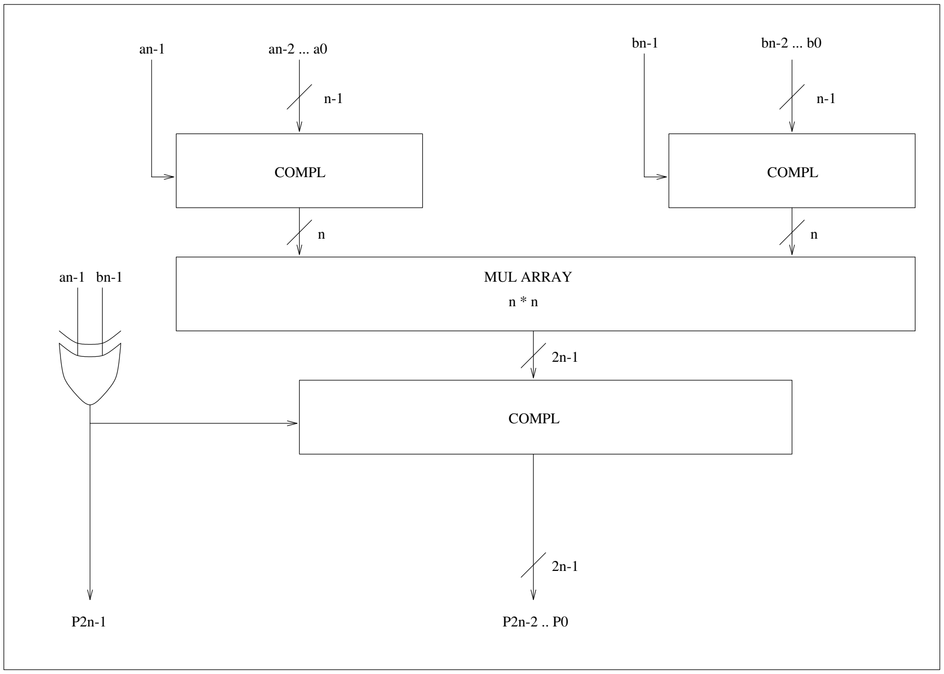


Abbildung 5.2.1 : Pre-Postkomplementationsmultiplikation

5.2.2. Pezaris Array Multiplizierer

Eine andere schnellere und bessere Möglichkeit bieten spezielle Multipliziererarrayanordnungen die direkt mit Zweierkomplementzahlen rechnen können. Zu diesem Zweck kann eine Sonderform eines Carry Save Array Multiplizierers (vergl. Abschnitt 5.1.3) eingesetzt werden, die 2er Komplementzahlen berücksichtigt. Dabei wird ausgenutzt, daß die Vorzeichenstelle einer Zweierkomplementzahl negatives Gewicht hat, was zur Konstruktion dieser Multiplizier Array Anordnungen verwendet werden kann [Pezaris71].
Ein solcher 2er Komplementmultiplizierer ist in Abbildung 5.2.2 illustriert.

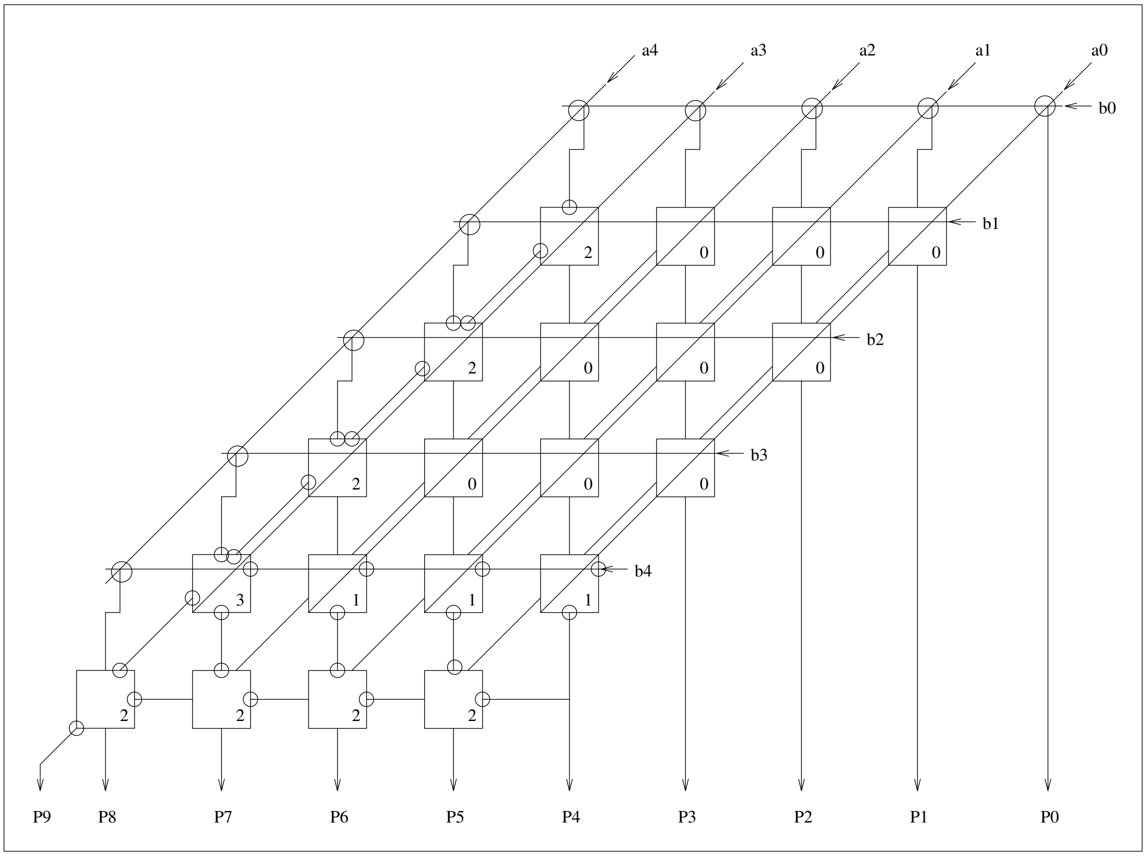


Abbildung 5.2.2 : 5 * 5 Bit Pezaris Multiplizierer

Als Beispiel seien A und B Zweierkomplementzahlen, die zur Kennzeichnung der negativen Bewertung durch eine Klammer eingeschlossen sind.

$$A = (a_{n-1})\ a_{n-2} \ldots a_1\ a_0$$
$$B = (b_{n-1})\ b_{n-2} \ldots b_1\ b_0 \qquad (5.2.2)$$

					(a_4)	a_3	a_2	a_1	a_0	
					$(a_4 b_0)$	$a_3 b_0$	$a_2 b_0$	$a_1 b_0$	$a_0 b_0$	$\leftarrow b_0$
				$(a_4 b_1)$	$a_3 b_1$	$a_2 b_1$	$a_1 b_1$	$a_0 b_0$		$\leftarrow b_1$
			$(a_4 b_2)$	$a_3 b_2$	$a_2 b_2$	$a_1 b_2$	$a_0 b_2$			$\leftarrow b_2$
		$(a_4 b_3)$	$a_3 b_3$	$a_2 b_3$	$a_1 b_3$	$a_0 b_3$				$\leftarrow b_3$
	$a_4 b_4$	$(a_3 b_4)$	$(a_2 b_4)$	$(a_1 b_4)$	$(a_0 b_4)$					$\leftarrow (b_4)$
P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	

Abbildung 5.2.3 : Partialproduktmatrix für eine Zweierkomplementmultiplikation[1]

Als Beispiel zeigt Abbildung 5.2.4 eine Partialproduktmatrix für 5 Bit Zweierkomplementzahlen. Alle Partialprodukte in Klammern (Produkte mit einem Vorzeichen) sind negativ bewertet, alle ohne Klammern sind positiv. Das Produkt P dieses Pezaris Addierers wird hierbei durch eine gemischte Addition und Subtraktion gewonnen, die durch 4 Grundelemente (Typen) realisiert sind.

Typ 0:
Für einen normalen Volladdierer werden alle 3 Eingangsbits x, y, z positiv bewertet, sowie die ausgehenden Bits c, s (Carrybit, Summenbit).Bei diesem dem Typ 0 zugeordneten Addierer wird kein Eingangsbit negativ bewertet.

Typ 1:
Besteht aus einem Volladdierer an deren Eingang ein Bit z. B. z als Carry einer vorherigen Stufe negativ bewertet wird.

Typ 2:
Wie Typ 1 nur erhalten hier die Eingangsbits x, y ein negatives Gewicht.

Typ 3:
Bei diesem Typ werden alle Eingangswerte negativ betrachtet.

[1] Die Klammern bedeuten eine negative Bewertung

Die 4 Typen der verallgemeinerten Volladdierern sind in Abbildung 5.2.4 zusammengefaßt.

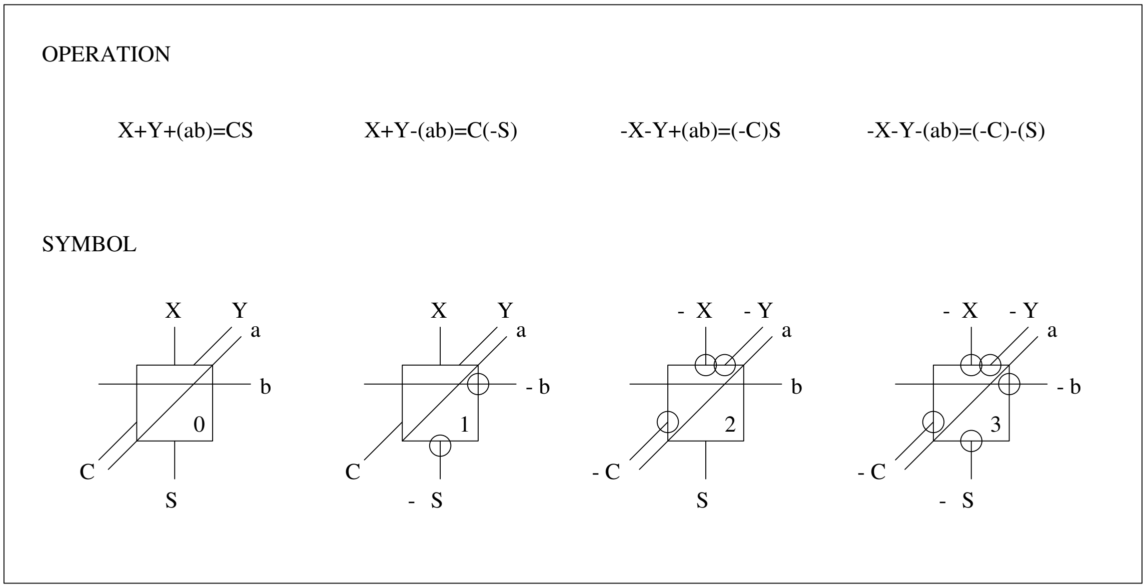


Abbildung 5.2.4 : Die Grundtypen eines Pezaris Multiplizierers

Hierbei bezeichnen die Kreise an den Blöcken die negative Bewertung, d.h. daß diese Bits subtrahiert werden müssen, anstatt eine Addition mit diesen stattfindet. Wichtig ist, daß diese keine Invertierungen darstellen. Alle 4 Volladdiertypen repräsentieren sehr ähnliche Logikfunktionen. Zur Illustration seien x, y, z die Eingangssignale und c, s die Ausgangssignale der verallgemeinerten Volladdierern. Es gelten dann folgende Logikfunktionen:

$$s = x \oplus y \oplus z \qquad \text{Typ } 0 - 3 \tag{5.2.4}$$

$$c = xy \vee z\,(x \vee y) \qquad \text{Typ } 0,\ 3$$

$$c = xy \vee z'\,(x \vee y) \qquad \text{Typ } 1,\ 2 \tag{5.2.5}$$

Trotz negativer Bewertung einiger Signale weichen nur die Logikfunktion der Übertragsfunktion beim Typ 1 und 2 geringfügig vom Typ 0 und 3 ab, indem z negiert wurde (z’) (Gleichung 5.2.5). Der Realisierungsaufwand aller Volladdier Typen ist dadurch praktisch gleich. Baut man diese Volladdiertypen zu einem Array Multiplizierer zusammen, so erhält man eine Struktur wie in Abbildung 5.1.4.

Nicht schwer zu erkennen ist die Ähnlichkeit zu einem Braun Multiplizierer (siehe Abbildung 5.1.4 a). Der einzige Unterschied zum Braun Addierer besteht lediglich in einer negativen Bewertung der Teilprodukte mit einem Vorzeichenbit. Auf folgende Besonderheiten muß weiterhin geachtet werden (vergl. hierzu Abbildung 5.2.2): Damit die Teilprodunkte ohne Vorzeichenbit mit normalen Volladdierern summiert werden können, muß für die Volladdierer von Partialprodukten mit Vorzeichenbit a_{n-1} ein Typ 2 eingesetzt werden. Aufgrund des Ergebnisses des Typ 1 Volladdierers (a_0 b_0) ist die Ebene p_{n-1} negativ zu bewerten. Eine positive Bewertung wird ähnlich wie bei der Vorzeichenerweiterung durch Addition einer negativ bewerteten Eins bei der nächst höheren Bitebene erreicht [Koren93] [Pirsch96].

5.2.3 Baugh Wooley Array Multiplizierer

Eine weiterer 2er Komplementmultiplizierer, der als Alternative zu dem Pezaris Multiplizierer zu sehen ist, ist die von Baugh Wooley vorgeschlagen worden [Pirsch 95]. Für dessen Aufbau ist eine besondere Partialproduktmatrix für positive und negative Summanden aufzustellen. Die negativ bewerteten Summanden können als zwei Zahlen interpretiert werden, die subtrahiert werden müssen. Wie aus den vorherigen Kapitel über Zweierkomplementzahlen hervorgeht, kann auch das Zweierkomplement dieser beiden Zahlen addiert werden.
Der wesentliche Unterschied vom Pezaris zum Baugh Wooley Multiplizierer kann wie folgt dargestellt werden. Bei dem Pezaris Addierer werden die Partialprodukte genau wie bei positiven Zahlen gebildet. Dabei werden jedoch 2 verschiedene Volladdierrealisierungen eingesetzt, während bei dem Baugh Wooley Addierer nur eine Volladdierrealisierung nötig ist. Wichtig ist jedoch hierbei, daß abweichende Partialprodukte durch den Einsatz von Komplementen zu bilden sind. Aufwand und Verzögerung des Baugh Wooley Array Multiplizierers entsprechen weitgehend dem Braun Array Multiplizierer [Pirsch 95].

5.2.4 Booth's Algorithmus Multiplizierer

Eine anderer Ansatz eines schnellen Multiplizierers könnte darin bestehen, die Anzahl der zu addierenden Partialprodukte für einen Geschwindigkeitsgewinn zu reduzieren. Damit würde sich auch die Anzahl der Teilkomponenten, die an der Multiplikation beteiligt sind, verkleinern. Eine Verwirklichung dieser Idee könnte nicht nur den Durchsatz eines Multiplizierers erhöhen, sondern auch die dessen Größe minimieren.
Der Booth Algorithmus beruht darauf, für Gruppen von 0 en (0 Bits) im Multiplikator, keine neuen Partialprodukte zu generieren. Gruppen von 1 en werden genauso zusammengefaßt. Die vorher addierten Partialprodukte müssen bei einer 0 nur um 1 Position nach rechts verschoben werden, während bei einem gesetzten Bit (1 Bit) ein neues Partialprodukt generiert werden muß. Um dieses zu bewerkstelligen wird eine Umkodierung des Multiplikators durchgeführt.

Tabelle 5.2.5 illustriert die Umrechnungstabelle:

b_i	b_{i-1}	b_i'	Operation	Beschreibung
0	0	0	schieben	Kette von 0 en
0	1	1	addieren und schieben	Ende einer 1 er Folge
1	0	-1	subtrahieren und schieben	Beginn einer 1 er Folge
1	1	0	schieben	Kette von 1 en

Tabelle 5.2.5 : Umrechnungstabelle für den Booth Algorithmus

Mit dieser Umrechnungstabelle wird jedes Bit b_i und das vorhergehende Bit b_{i-1}[1] des Multiplikators $b_{n-1}b_{n-2} \ldots b_1b_0$ in eine umkodierte Sequenz $b'_{n-1}b'_{n-2} \ldots b'_1b'_0$ gewandelt.

Beispiel:

orginal Multiplikatorbitfolge:	0 0 1 1 1 1 1 1 1 0 0
umkodierte Bitfolge:	0 1 0 0 0 0 0 0 0 -1 0

Die Umkodierung brachte anstatt 7 nur 2 Addier-Subtrahier Operationen. Nicht in jedem Fall ist ein Gewinn zu erzielen. So können in bestimmten Fällen nach der Umkodierung auch mehr Operationen nötig werden, um eine Multiplikation durchzuführen. Der Algorithmus wird uneffizient. Das nachfol gende Beispiel verdeutlicht diesen Nachteil, bei dem anfangs aus 2 Rechenoperationen nach der Umkodierung 4 werden.

Beispiel:

orginal Multiplikatorbitfolge:	0 1 0 1
umkodierte Bitfolge:	1 -1 1 -1

Ein Ausweg aus dem eben angesprochenen Nachteil bietet der Basis-4 Booth Algorithmus. Bei diesem werden gleich 3 Bits des Multiplikators umkodiert. Daraus ergibt sich eine höhere Effizienz besonders im Bezug auf isolierte 0 oder 1 Bits (siehe auch Beispiel oben) [Koren93].
Der Booth's Algorithmus läßt sich auch für den Aufbau eines Array Multiplizierers für Zweierkomplementzahlen verwenden, indem des Signbit b_{n-1} und b_{n-2} weiter umkodiert werden [Koren93]. So wird, sollte beim umkodierten Multiplikator eine -1 stehen, das 2er Komplement des Multiplikanten gebildet und weiterverwendet.
Ein Hardwarerealisierung besteht aus n Reihen von Basiszellen und ist in Abbildung 5.2.6 für 4 Bitoperanden dargestellt. N ist hierbei die Anzahl der Bits der in Tabelle 5.2.5 vorgestellten Kodierung des Multiplikators. Aufgabe jeder Zeile ist es entweder eine Addition oder Subtraktion mit dem zuvor gespeicherten Partialprodukt durchzuführen. Die Einzelzellen, die CASS[2] Zellen genannt werden, sind in Abbildung 5.2.7 dargestellt.

[1] Das vorherige b_{i-1} Bit von b agiert nur als Referenz Bit
[2] CASS bedeutet „circuit add / subtract / shift)

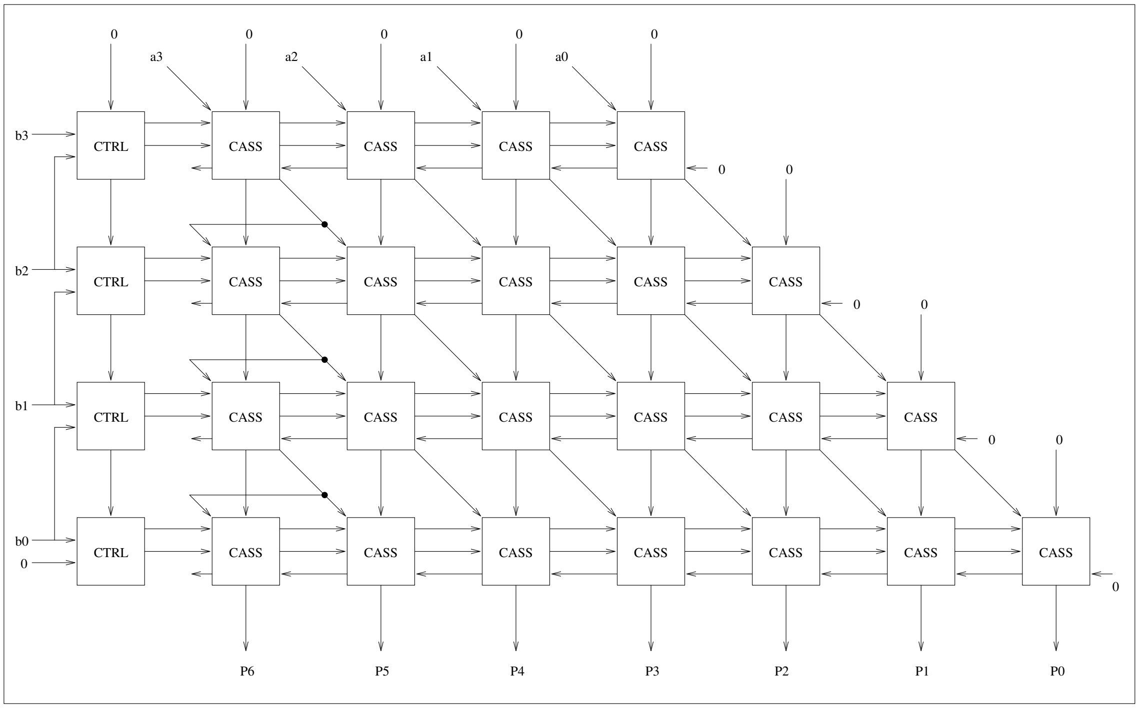


Abbildung 5.2.6 : Booth's Array 2'er Komplementmultiplizierer

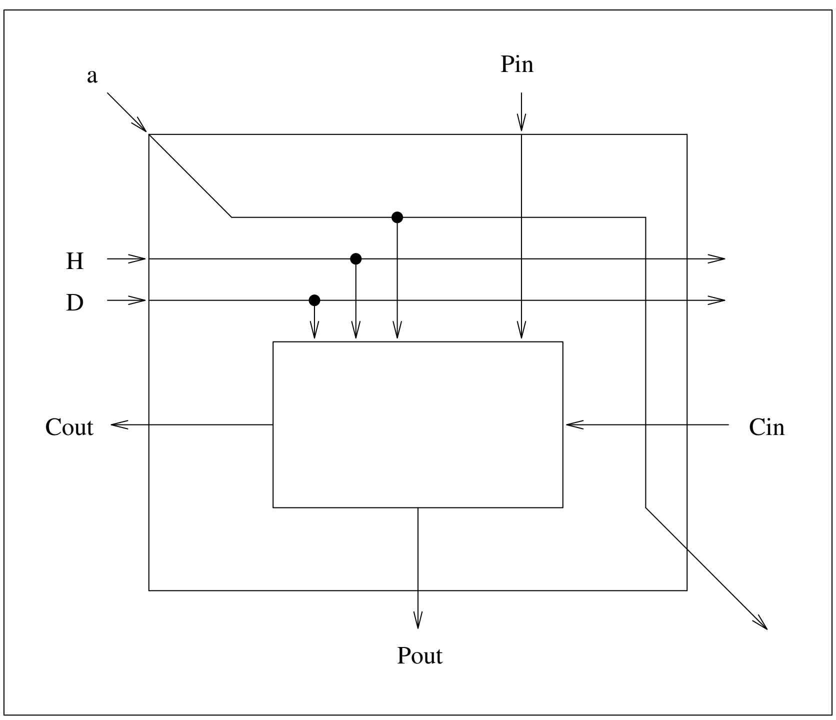


Abbildung 5.2.7 : steuerbare add/sub/shift (CASS) Zelle

Abhängig des Wertes von x_i und dem zugehörigen Referenzbit kann jede Zelle eine Addition, Subtraktion oder einfach nur einen Transfer durchführen. Zur Steuerung der Basiszelle des Multiplizierers (Addition, Subtraktion, Rechtsschiebung) finden 2 Signale H, D Verwendung. Je nach dem Typ der Operation, die eine CASS Zelle durchführen soll, werden diese gesetzt oder gelöscht. Soll keine Operation durchgeführt werden so muß H auf 0 gesetzt werden. Dies bedeutet, daß der Wert den p_{out} hat einfach von p_{in} „durchgeschoben“ wird, ohne verändert zu werden. Ist H = 1, so soll eine arithmetische Operation durchgeführt werden, die von Signal D abhängt. Ist D = 0, so wird das Multiplikatorbit zu p_{in} mit c_{in} als Carrybit einer vorangegangenen Berechnung addiert. Die Zelle selbst berechnet dann p_{out} und ein neues Carry c_{out} für eine andere Zelle. Sollte D = 1 ein, dann wird der Multiplikant a von p_{in} mit c_{in} als eingehendes geborgtes Bit subtrahiert.

Die Logik der CASS Zelle kann wie folgt generiert werden:

$$\begin{aligned} P_{out} &= P_{in} \oplus (a \wedge H) \oplus (c_{in} \wedge H) \\ C_{out} &= (P_{in} \oplus D) \wedge (a \vee c_{in}) \vee a \wedge c_{in} \end{aligned} \tag{5.2.4}$$

Eine andere Möglichkeit existiert für die Zellgenerierung, soll hier aber weiter nicht besprochen werden, kann jedoch in [Koren93] nachgelesen werden. Die Signale H und D für jede Zeile werden abhängig des Booth's Algorithmus von einer CTRL Schaltung mit Multiplizier b_i und dem Referenzbit b_{i-1} generiert [Koren 93]. Wie aus Abbildung 5.2.6 zu ersehen ist erhöht sich die Bitanzahl von Stufe zu Stufe, weshalb man den Multiplikand um 1 Bit erweitern muß, bevor man eine Addition oder Subtraktion durchführen kann (Signbiterweiterung 2er Komplement vergl. Abschnitt 3.1.3). Vorteil dieses Multiplizierers ist die Möglichkeit Zweierkomplementzahlen zu multiplizieren ohne zusätzliche Korrekturschritte einzufügen. Ferner muß die Operation in Zeile i nicht verzögert werden, bis alle i-1 Zeilen ihre Berechnungen vollendet haben, was ein Vorteil im Bezug auf unterschiedliche Wortbreiten b_m darstellt. Das Ergebnis P_0 liegt somit an, sobald die erste CASS Zelle mit ihren Verzögerungszeiten durchlaufen ist. P_1 nach 2 CASS Zellenverzögerungen u.s.w. Das höchste Bit P_{2n-2} wird nach 2_{n-1} CASS Zellenverzögerungen generiert.

6 Vorgehensweise für die VHDL Implementation

6.1 Entwurfsvorgehen

Es ist möglich eine integrierte Schaltung, die von einer Problemstellung ausgeht, direkt in einem Schritt zu entwickeln oder der Entwurf geschieht in mehreren Stufen, die systematisch von einer sehr abstrakten Beschreibung, beispielsweise in VHDL, zur einer konkreten Struktur führen. Hierbei wird die sogenannte „Top Down“ Methode angewendet.
Die Hardwarebeschreibungssprache VHDL wurde 1983 vom amerikanischen „Department of Defence“ initiiert und ist seit Ende 1987 als IEEE[1] Standart genormt. Inzwischen ist VHDL quasi der Standard der Hardwarebeschreibungssprachen besonders in Europa. Sie wird für die Beschreibung und Simulation digitaler Systeme (ASICs) und deren Umgebung benutzt.

VHDL nutzt eine Syntax, die mit einer prozedualen Programmiersprache wie Ada und Pascal vergleichbar ist, jedoch hochparallel arbeitet. VHDL unterstützende Simulations- und Synthesewerkzeuge ermöglichen es, einen Großteil der Schritte von der Spezifikation bis zum Layout eines Chips nahezu nahtlos aneinanderzufügen. Es sind keine Konvertierungen nötig, da alle Abstraktionsebenen in dem Sprachumfang abgedeckt werden. Da VHDL eine Programmiersprache ist, kann eine Struktur detailliert oder aber völlig frei von konkreten Strukturbeschreibungen beschrieben werden.
Der eigentliche Design Prozeß bei der Verwendung von VHDL entspricht einem „Top Down“ Vorgehen. Beim Entwurf der Algorithmen ist der Ausgangspunkt die Verhaltensbeschreibung der Schaltung, deren Funktionalität durch Simulation geprüft werden soll. Die „Top Down“ Strukturierung erhält man, indem man im weiteren Vorgehen die Schaltung in Funktionsblöcke gliedert, bis man letztlich nur noch eine Strukturbeschreibung erhält. Dieser Prozeß des algorithmischen Entwurfes von Funktionseinheiten, hierarchischer Verfeinerung und Umsetzung in Strukturbeschreibungen wird rekursiv ausgeführt, bis man schließlich bei Elementen einer Zellbibliothek angekommen ist, und die Schaltung praktisch realisiert werden kann. Dadurch, daß die Schnittstelle der Schaltung und deren eigentliche Realisierung voneinander getrennt sind, können Alternativen im Entwurf unterstützt werden. (exploring the design space). Durch den Einsatz von Synthesewerkzeugen wird dann die Entwurfsaufgabe auf den unteren Abstraktionsebenen dabei zunehmend automatisiert.

[1] Institute of Electrical and Electronics Engineers (IEEE)

6.2 Simulation

Nach dem Entwurf und der Beschreibung einer digitalen Schaltung in der Hardwarebeschreibungssprache VHDL ist zu überprüfen, ob die beschriebene Schaltung der vorgegebenen Spezifikation entspricht, noch bevor ein Synthesewerkzeug eingesetzt wurde. Zu diesem Zweck wird immer zuerst eine Simulation der entworfenen Hardwarestrukturen direkt vom VHDL Code aus durchgeführt.
Zur eigentlichen Simulation wird ein Stimulus File in VHDL benutzt, daß der Programmierer erstellt und eine Umgebung in der alle Signale, die in das Design hinein- und hinausgehen, beschrieben sind. In diesem sind auch Stimuli Prozesse und die hierfür benötigten Deklarationen enthalten. Das dem zu testende Design hierarchisch übergeordnete Entity[1], das als „Testbench" bezeichnet wird, ist für jeden Port, des zu simulierenden Designs, ein mit ihm verbundenes Signal deklariert. Im Anweisungsteil des Testbench Architektur können in VHDL Prozessen diesen Ports Werte zugewiesen werden und die resultierende Antwort an den Output Port Signalen beobachtet werden. Das Stimulus File zu einem Design Projekt enthält demnach die VHDL Beschreibung dieser „Testbench"
Hierbei kann schon geprüft werden, inwieweit die Hardwarestruktur mit Hilfe eines Grundtaktes funktionsfähig ist. Nach der Synthese wird dieses taktbasierte Zeitverhalten um die entsprechenden Verzögerungszeiten der Bibliothekselemente ergänzt, was jedoch mehr Zeitaufwand bedeutet und schwieriger zu „debuggen" ist.

Es können zwei Aspekte des Designs unterschieden werden. Zum einen die Funktionalität der Schaltung, beschrieben u.a. durch die logische Funktion der dazugehörenden Schaltnetze, zum anderen das detailierte zeitliche Verhalten der Schaltungen unter Berücksichtigung der auftretenden Verzögerungszeiten. Beide Aspekte können und müssen durch eine Simulation mit geeigneten Stimulies[2] dahingehend untersucht werden, ob sie der geforderten Spezifikation entsprechen. Sollte die Simulation nicht das gewünschten Ergebnisse erbringen, so muß der VHDL Code neu überarbeitet werden, um dann in einer weiteren Simulation neu getestet zu werden. Hierbei ist es sehr hilfreich, wenn man Gatterverzögerungszeiten zunächst aus Geschwindigkeitsgründen nicht berücksichtigt. Diese Simulationsart wird als „funktionale" Simulation bezeichnet, im Gegesatz zur „timing Simulation", die mit allen Verzögerungszeiten durchgeführt wird.

Für die Durchführung der Simulation benötigt man eine Beschreibung der zu überprüfenden Schaltung, in diesem Fall der einzelnen Addierer- und Multiplizierstrukturen, die sämtliche Verzögerungszeiten exakt oder zumindest in ausreichender Näherung enthalten.

---

[1] Ein Entity definiert für eine Komponente des Entwurfs die „externe Sichtweise". Namen, Ein-Ausgänge des Designs und zusätzliche Deklarationen werden hier festgelegt.
[2] Stimulies sind Simulationsdateien, quasi ein Interface, um ein Design zu simulieren und dieses auf die Funktion hin zu überprüfen.

Da für diesen Zweck die physikalische Realisierung der Schaltung bekannt sein muß, werden diese Verzögerungszeiten vom Compiler[1] aus dem Design extrahiert, nachdem sie auf die Standardzellen eines Herstellers gemappt worden sind. Hierzu erzeugt der Entwurfswerkzeug optional eine Designbeschreibung in VHDL, in der diese Verzögerungszeiten enthalten sind. Diese Beschreibung ist eine Netzliste, die in ihren beschriebenen Strukturen die physikalische Gliederung des Designs in Standardzellen des Herstellers wiederspiegelt. Sowohl die hierarchische Strukturierung der ursprünglichen VHDL Beschreibung des Designs, wie die auch eventuell verwendeten Namen für Signale, Instanzen, Automaten, Automatenzustände und die I/O[2] Signale stimmen zwar in der Netzliste mit denen in der VHDL Beschreibung überein, sind jedoch zusätzlich erweitert. Dieses führt zu einer Unübersichtlichkeit, die das „Debuggen“ erschwert. Höhere Datentypen werden durch entsprechenden Bitvektoren ersetzt.

Für das Aufspüren von Fehlern in VHDL Beschreibungen ist es aber unerläßlich, alle Signale und verwendeten Variablen beobachten, oder gar der ursprünglichen Beschreibung zuordnen zu können.
Erkennung und Lokalisierung eines Fehlers wird immer erleichtert, wenn die simulierte Schaltungsbeschreibung genau so strukturiert ist wie in der ursprünglichen Beschreibung, d.h.

- Design Hierarchie,
- Busse,
- Namen aller Design-Objekte, also Automaten, Zustandsnamen, Instanz und Signalnamen

sollten in dem Simulationsmodell enthalten sein. Der Simulator unterstützt die funktionale Simulation eines solchen Schaltungsmodells [Maeder93].

---

[1] Ein Synopsys Design Compiler ist ein Werkzeug, des die Firma Synopsys für die VHDL Übersetzung zu Verfügung stellt.
[2] I/O Signale sind Input/Output Signale, also Signale die aus einem Design kommen und hineingehen.

6.3 Synthese

Simulation ist nur ein Schritt im Entwurfsablauf einer integrierten Schaltung. Einen anderen wichtigsten Schritte im Ablauf einer Chipentwicklung stellt die Synthese, wie schon in Abschnitt 6.2 kurz erwähnt dar, besonders die Verwendung von VHDL zu synthesegerechten Schaltungsmodellierung. VHDL ist der ideale Ausgangspunkt für Synthesewerkzeuge, da Ein- und Ausgabe der Programme in VHDL erfolgen kann, wobei, abhängig von den Möglichkeiten, eine mehr oder minder abstrakte VHDL Beschreibung verarbeitet wird. Die Hardwarebeschreibungssprache VHDL deckt mit seinen Ausdrucksmöglichkeiten alle im digitalen IC Entwurf verwendeten Abstraktionsebenen ab, während andere Netzlistenbeschreibungen mit dem Datentypen „0“ und „1“ nur einen Aufbau aus Logikgattern nachbilden. So kann das Verhalten kompletter Systeme mit abstrakten Datentypen modelliert und simuliert werden (siehe auch Abschnitt 6.1).
Zusammenfassend kann man sagen, daß die Transformation von Verhaltensbeschreibungen in Strukturbeschreibungen aus Elementen niedrigerer Abstraktionsebenen als Synthese bezeichnet wird, die einer Zellbibliothek eines Herstellers entsprechen. So werden heutzutage bei kommerziellen Werkzeugen die Logiksynthese von Schaltnetzen, die Synthese von Datenpfaden für Register und Funktionseinheiten und die Synthese endlicher Automaten relativ problemlos beherrscht. Nicht jede VHDL Beschreibung ist möglicher Ausgangspunkt für Synthesetools, d.h. auf Logikzellen eines Herstellers abbildbar. Wie in Programmiersprachen lassen sich auch in VHDL Verhaltensweisen auf viele verschiedene Arten ausdrücken, wobei abhängig auch vom Werkzeug einige besonders gut für die Synthese geeignet sind andere wiederum nicht synthetisierbar sind d.h. zu uneffizienten Hardwarelösungen führen. Ist ein VHDL Code mit Hilfe eines Syntheswerkzeuges synthetisierbar, so spricht man von *synthesis ready code*.

Neben der Art der Sprachbeschreibung wirken sich vor allem noch die Art der Zielbibliothek d.h. welche Elemente stellt der Hersteller zur Verfügung und die Art und Weise wie der Benutzer den Syntheseprozeß steuert, auf des Syntheseergebnis aus. Als noch wichtiger erscheinen jedoch die Möglichkeiten die das Synthesewerkzeug zur Verfügung stellt. Die meisten Synthesewerkzeuge bieten vielfältige Möglichkeiten um Randbedingungen für die Synthese einzustellen.
So können

- Zeitabhängigkeiten zwischen Signalen
- Schranken für Flächen
- Belastung vorangeschalteter Stufen
- Treiberleistung einzelner Gatter

angegeben werden.

Durch die Angabe dieser Randbedingungen wird einerseits der Suchraum des Synthesesystems eingeschränkt, andererseits können Anforderungen der späteren Schaltungsumgebung im Bezug auf schnellere, gezieltere Ergebnisse berücksichtigt werden.
Einer der entscheidensten Kriterien, wie und ob VHDL Code synthetisierbar ist, die Art der Zuweisungen an Prozesse oder an Signale.

Es ist möglich eine Synthese in den Bereich der Registertransferebene, teilweise sogar bis hin zur Hauptblockebene problemlos durchzuführen, wobei die einzelnen Komponenten durch Verhaltensbeschreibungen spezifiziert sind [Maeder93].

Nach der Synthese erhält man die benötigte maximale Chipfläche, also die Summe der Fläche für die verwendeten Standardzellen, plus einen Schätzwert für den Flächenbedarf der Verdrahtung. Bei der, sich durch den technologischen Fortschritt ergebenen Verkleinerung, der Strukturbreiten, nimmt diese Verdrahtungsfläche einen immer größer werdenden Anteil an der Gesamtfläche des IC's ein. Bei vielen Entwürfen belegt die Verdrahtung schon die größte Fläche.

7 Ergebnisse

7.1 Diskussion der Ergebnisse

Der eigentliche Vorteil, den die hier vorgestellten und entworfenen Addier- und Multiplizierschaltungen bieten, ist im Aufbau und der Kenntnis der bereitgestellten Eigenschaften zu sehen. So sind bei den bereitgestellten + und * Operatoren der Beschreibungssprache VHDL die dahinterstehenden Strukturen abhängig von den verschiedenen Implementationen der Operatoren, die ein Hersteller anbietet. Verschiedene Hersteller bieten unterschiedlich Implementationen.
Um die einzelnen Addier- und Multiplizierstrukturen untereinander vergleichen zu können wurden einige ausgesucht und in der Hardwarebeschreibungssprache VHDL programmiert[1]. Die Selektion wurde auf Geschwindigkeit und leichte Erweiterbarkeit im Bezug auf größere Wortbreiten durchgeführt. Dabei kristallisierten sich einige Strukturen heraus, die im Nachfolgenden näher besprochen werden sollen. Um eine Simulation der Hardwarestrukturen durchzuführen, wurde jeweils für die Addier- und Multiplizierstrukturen ein Testumgebungsprogramm in VHDL geschrieben, das eine schnelle Verifizierung im Bezug auf ihre Funktionsweise bietet (vergl. hierzu Addier und Multiplizier Stimulusfile Abschnitt 6.2). Diese Simulationsumgebung ermöglicht es von außen die beschriebene Hardwarestruktur mit Eingangsparametern zu belegen und ihre Ausgangsparameter unter anderem als Wellenform darzustellen. Auf diese Weise war es möglich die Funktionsweise der programmierten Strukturen zu überprüfen.
Bei den Addierern wurde als erstes der von VHDL bereitgestellte + Operator (vergl. hier + Operator Programm auf der Zusatzdiskette) ein 8 Bit Ripple Carry Addierer (vergl. Abschnitt 4.2.2), ein 8 Bit Carry Look Ahead Addierer (Abschnitt 4.2.3) und ein 8 Bit Pyramiden Addierer (Abschnitt 4.3.2) programmiert und synthetisiert. Wie aus der Spezifikation des + Operators in VHDL entnommen werden kann, ist die Wortbreite leicht auf 32 Bit und 2er Komplement (vergl. Abschnitt 3.1.3) erweiterbar.
Wollte man größere Wortbreiten verarbeiten, so könnte man, wie in dieser Studienarbeit vorgeschlagen, eigene Addierstrukturen in VHDL beschreiben. Für die meisten Anwendungen reicht jedoch die 32 Bit Breite aus. Es stehen eine Ripple Carry Addierer, Carry Look Ahead und Fast Carry Look Ahead zur Verfügung. Je nach selektierter Addierart variieren die Addierer in Größe und Geschwindigkeit.

Der 8 Bit Ripple Carry Addierer ist aufgrund seines regulären Aufbaus (Kaskadierung eines Volladdierelementes) sehr leicht durch Aneinanderreihung von 1 Bit Volladdierern auf jede nur gewünschte Wortbreite erweiterbar. Als nachteilig wirkt sich jedoch, besonders auf die Geschwindigkeit, das „Carryrippeln“ aus (vergl. auch Abschnitt 4.2.2), also das Weitergeben eines unteren Übertragsbits an eine höhere Stelle.

[1] alle hier genannten Programme sind auf einer Programmdiskette enthalten, die dieser Studienarbeit beiliegt.

Als besser erwies sich der 8 Bit Carry Look Ahead Addierer, der einen größeren Geschwindigkeitsvorteil gegenüber dem Ripple Carry Addierer bietet, da die nächsthöheren Stufen nicht warten müssen, bis eine untere Stufe das Übertragsbit (Carrybit) zur Verfügung gestellt hat. Die einzelnen Übertragsbits werden hier parallel berechnet und liegen sofort an jeder Stufe an, wenn sie gebraucht werden. Keine Stufe muß auf das Übertragsbit aus einer niedrigeren Stufe warten (vergl. hierzu Abschnitt 4.2.3).
Die Erweiterung auf größere Bitbreiten ist ähnlich einfach wie beim Ripple Carry Addierer. Für sehr große Bitbreiten gibt es jedoch einige Einschränkungen (vergl. Abschnitt 4.2.3).

Eine andere Variante eines programmierten Addierers, der im Bezug auf Geschwindigkeit die bessere Resultate gegenüber den anderen Addierern liefert, ist der Pyramiden Addierer (vergl. Abschnitt 4.3.2). Vorteile des Addierer sind seine leichte Erweiterbarkeit auf größere Bitbreiten und seine hierarchisch aufgebaute Struktur ohne jegliches Übertragsrippeln.

Bei den Multiplizierstrukturen wurde ein 4 * 4 Bit Ripple Carry Array Multiplizierer (vergl. Abschnitt 5.1.2), ein Unterart des Carry Save Multiplizierers, der Braun Array Multiplizierer (vergl. Abschnitt 5.1.3) gleicher Bitbreite und ein 5 * 5 Bit Pezaris 2-er Komplement Multiplizierer programmiert und synthetisiert.

Alle 3 Multiplizierer erwiesen sich als relativ problemlos, wie auch die Addierer, in VHDL umzusetzen nicht zuletzt wegen ihrer Regularität.
Der 4 * 4 Bit Ripple Carry Multiplizierer ist der langsamste, da in jeder Zeile des Arrays (vergl. Abschnitt 5.2.1) das Carrybit weitergegeben werden muß. Ein zeitkritischer Pfad, also der Pfad, wo ein Signal am längsten braucht, um durchzulaufen, verläuft hier im Zickzack durch das Array. Vorteilhaft erscheint nur die leichte Erweiterbarkeit auf größere Bitbreiten. Ferner läßt sich durch die hohe Regularität diese Multiplizierstruktur, wie auch alle Anderen leicht als Full Custom Design[1] realisieren.

Der Braun Array Multiplizierer (vergl. Abschnitt 5.1.3) kommt ohne das „Carryrippeln“ aus und zeigt deshalb bessere Eigenschaften im Bezug auf Geschwindigkeit. Durch seine Regularität und Einfachheit der Basiszellen kann er als guter Multiplizierlösung für positive Zahlen eingesetzt werden. Auch die Erweiterung auf größere Bitbreiten stellt kein Problem dar. Einen Geschwindigkeitsverlust erhält diese Multiplizierform nur in der letzten Stufe durch den Einsatz eines Carry Propagate Addierers (CPA). Der zeitkritische Pfad verläuft hier senkrecht von oben nach unten und dann den CPA Addierer entlang zum letzten Ausgabebit.

[1] Full Custom Design bedeutet nicht die Standardzellen eines Herstellers zu benutzen, sondern sich quasi jeden benötigten Transistor und damit seine Treiberleistung selber zu erstellen und zu verbinden.

Als letzte Multiplizierart wurde ein 5 * 5 Bit Pezaris 2-er Komplementmultiplizierer (vergl. Abschnitt 5.3.2) umgesetzt. Der Pezaris Array Multiplizierer hat einen Aufwand und eine Verzögerung, die weitgehend einem Braun Array Multiplizierers entsprechen müßte. Auch die Erweiterbarkeit auf größere Bitbreiten ist ähnlich einfach. Nachteilig im Bezug auf Geschwindigkeit hier der abschließende Carry Propagate Addierer (CPA) am Ende des Arrays und das damit verbundene „Carryrippeln“. Der zeikritische Pfad verläuft auch hier wie beim Braun Array Multiplizierer. Ein Multiplizierer ohne abschließenden CPA Addierer und damit vermutlich auch der schnellste hier vorgestellte Multiplizierer, stellt nur der Booth Multiplizierer (vergl. Abschnitt 5.3.4) dar.

7.2 Abschätzung der Hardwareaufwandes

Nach der Programmierung einiger ausgesuchter schneller Addierer und Multiplizierer in VHDL, war zu untersuchen wie diese auf die Standardzellen eines Herstellers abgebildet werden. Der Zusammenschluß und das Zusammenwirken dieser Standardzellen spiegeln im Endeffekt die Funktion des Chips wieder. Der Zusammenschluß, Größe, Plazierung und Verdrahtung dieser Standardzellen entspricht der Funktion des Chips. Die Komplexität dieser Standardzellen führt dann zu der Größe der Schaltung Eine Vergrößerung dieser bedeutet u.a. eine Erhöhung der Gesamtkosten eines Chips. Somit sind die Hersteller immer bemüht, möglichst kleine und kompakte Standardzellen in ihre Zellbibliothek aufzunehmen, die trotzdem eine gewisse Treiberleistung zu Verfügung stellen. Dies ist wichtig, um nachfolgende Zellen treiben zu können d.h. mit genügend Strom zu versorgen, damit die Funktion dieser erhalten bleibt. Abgebildet wurden die programmierten Strukturen auf die Standardzellen des Herstellers ES2 in einem 2 Metallebenen 0.7u CMOS Prozeß[2].

Die Synthese wurde jeweils für jede Struktur im Bezug auf eine Flächen- und eine Zeitoptimierung durchgeführt. Zu diesem Zweck mußte dem Synthesotool angewiesen werden, für jede zu untersuchende Struktur verschiedene Alternativen durchzuprobieren und das Beste im Bezug auf Platz und Geschwindigkeit festzuhalten. Die Anweisungen an das Synthesetool dies zu tun, wurden mit einer eigens dafür geschriebenen Batchdatei[1] durchgeführt. Ergebnis was jeweils das Optimalste im Bezug auf Flächenbedarf und Geschwindigkeit der einzelnen Strukturen.

Es wurden bei den Additionsgrundschaltungen der 8 Bit Ripple Carry Addierer, der 8 Bit Carry Look Ahead Addierer und der 8 Bit Pyramieden Addierer synthetisiert. Bei den Multipliziergrundschaltungen der 4 * 4 Bit Ripple Carry Multiplizierer, der 4 * 4 Bit Braun Multiplizierer und der 5 * 5 Bit Pezaris 2-er Komplementmultiplizierer synthetisiert.

[1] Batchdateien sind Ansammlungen von Befehlen mit Parameterangaben, die nacheinander ausgeführt werden sollen.
[2] Nähere Informationen findet man in „ES2 ECPD07 library databook“ im Anhang

Die einzelnen Ergebnisse der jeweiligen Strukturen für die Platz- und Geschwindigkeitsoptimalste Lösungen wurden in der Tabelle 7.2 a und b zusammengefaßt. Leicht zu erkennen ist hier, daß die geschätzte Verdrahtungsfläche im Bezug auf die Standardzellenfläche immer größer ausfällt. Auch die Vermutungen die im vorherigen Abschnitt über die Geschwindigkeit gemacht wurden, bestätigten sich nur zum Teil (vergl. Tabelle 7.2 und Abschnitt 7.1).
Die Flächenangaben sind in μm^2, die Zeitangaben in ns Bereich. Die maximale Taktraten, mit der die einzelnen Strukturen „getaktet" werden können, stehen in Klammern.

	Ripple Carry Addierer	Carry Look Addierer	Pyramiden Addierer
Flächenoptimal			
Zellfläche:	24711.400391	20520.000000	28739.400391
Verdrahtungsfläche:	34816.000000	33024.000000	44544.000000
Gesamtfläche:	24711.400391	53544.000000	73283.398438
Geschwindigkeit:	12.31 (81.2 Mhz)	9.69 (103 Mhz)	9.11 (110 Mhz)
Zeitoptimal			
Zellfläche:	45600.000000	43236.398438	40192.601562
Verdrahtungsfläche:	75872.000000	70512.000000	63824.000000
Gesamtfläche:	121472.000000	113748.398438	104016.601562
Geschwindigkeit:	5.2 (192 Mhz)	5.28 (189.4 Mhz)	5.62 (178 Mhz)

Tabelle 7.2 a : Zeit- und Flächenbedarf für Addiergrundstrukturen

	Ripple Array Mult.	Braun Mult.	Pezaris 2-er Komplement
Flächenoptimal			
Zellfläche:	38456.000000	10389.200195	55871.398438
Verdrahtungsfläche:	56736.000000	20368.000000	74224.000000
Gesamtfläche:	95192.000000	30757.199219	130095.398438
Geschwindigkeit:	18.81 (53.2 Mhz)	4.17 (240 Mhz)	14.85 (67.3 Mhz)
Zeitoptimal			
Zellfläche:	63444.796875	9864.800781	70334.203125
Verdrahtungsfläche:	104592.000000	20336.000000	99088.000000
Gesamtfläche:	168036.796875	30200.800781	169422.203125
Geschwindigkeit:	10.30 (97 Mhz)	2.97 (336.7 Mhz)	8.39 (119 Mhz)

Tabelle 7.2 b : Zeit- und Flächenbedarf für Multipliziergrundstrukturen

8 Ausblick

8.1 Verbesserung und Ausbaumöglichkeiten

Für eine hohe Durchsatzrate eines Systems sind geringe Verzögerungszeiten notwendig, die einerseits durch technologische andererseits durch schaltungstechnische Maßnahmen erreicht werden können. Aufgrund großer Fortschritte in der Prozeßtechnologie (besonders durch Verkleinerung von Strukturbreiten) können Kapazitäten auf einem Chip reduziert werden und damit eine Verringerung der Verzögerungszeit erzielt werden. Jede Erhöhung der Kapazitäten auf einem Chip erhöht die Laufzeit von Signalen [Weste/Eshraghain93].

Neben technologischen Maßnahmen zur Durchsatzsteigerung sind auch schaltungstechnische Maßnahmen zur Erhöhung der Durchsatzrate möglich. In Kapitel 4 und 5 wurden verschiedene Schaltungsstrukturen für Addierer und Multiplizierer vorgestellt. Die verschieden Alternativen zeigen unterschiedliches Verhalten im Bezug auf Verzögerungszeit und Fläche auf einem Chip (vergl. Kapitel 7). Die in den Kapiteln 4 und 5 vorgestellten Strukturen arbeiten in der jeweils beschriebenen Form noch nicht sehr effizient, da zu einer gegebenen Zeit höchstens eine Stufe des Werkes, bzw. zur Multiplikation zweier Zahlen, beschäftigt ist. Sollten beispielsweise viele Multiplikationen zur Ausführung anstehen, so könnte eine nächste Multiplikation bereits begonnen haben, wenn die erste Multiplikation die erste Stufe verlassen hat (Pipelining Prinzip). Dazu kann man sich eine Aufgabe vorstellen, die in mehrere gleichartige Teilaufgaben gespalten wurde und jede Teilaufgabe dann von einer Ausführungseinheit bearbeitet und an die nächste weitergegeben wird. Die Lösung der Teilaufgaben, die einzelne unabhängige Prozesse erledigen, ergeben dann die Gesamtlösung des Problems. Die Abarbeitung der Teilaufgaben kann mir einem Fließband in einer Fabrik verglichen werden, wo mehrere Arbeiter immer nur eine Teilaufgabe erledigen und diese dann an den nächsten Arbeiter weitergeben. Abbildung 8.1 verdeutlicht dies:

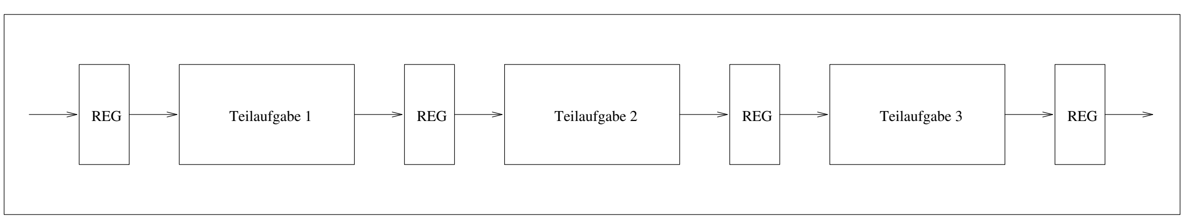


Abbildung 8.1 : Aufteilung eines Problems in einzelne Teilprobleme für eine Pipelinestruktur.

Diese Maßnahme führt neben einer erhöhten Auslastung der einzelnen Stufen, zu einer erheblichen Geschwindigkeitssteigerung. Dieses Pipelining Konzept ist nur dann sinnvoll, wenn beispielsweise viele Multiplikationen gleichzeitig zur Ausführung anstehen. Prinzipiell kann man sagen, daß der Durchsatz eines Systems immer von seiner langsamsten Teilkomponente bestimmt wird. Ist es möglich diese langsame Teilkomponente im Bezug auf Geschwindigkeit zu optimieren, so verbessert sich der Durchsatz des Gesamtsystems. Auch die speichernden Register die bei jedem getakteten digitalen System Verwendung finden, liefern einen Beitrag zur Gesamt-verzögerungszeit. Gesucht werden immer Architekturen zur Erhöhung der Durchsatzrate, bei dem die Dominanz des langsamsten Moduls in einer Kette durchbrochen werden kann. So könnten bei einer Signalverarbeitung mit mehrfacher Addition und Multiplikation die Multiplizierer laufzeitmäßig optimiert werden.

Wesentlich effizienter ist es dagegen, eine arithmetische Operation in mehrere Teiloperationen zu zerlegen, auf die man Pipelining Techniken anwenden kann. So könnte bei einer Multiplikation der Multiplikator der Länge n in n/h Gruppen der Größe h zerlegt werden. Dadurch zerfällt die Gesamtmultiplikation in n/h Teilmultiplikationen mit einem kürzeren Multiplikator. Die h Summanden einer Multiplikation können dann beispielsweise mit Hilfe einer Wallace Baum Struktur (vergl. Abschnitt 5.2.3) reduziert werden. Das Teilergebnis kann dann weiter um h Bits nach rechts geschoben und mit dem Summanden der nächsten Teilmultiplikation verarbeitet werden. Abschließend werden alle Teilmultiplikationen mit Hilfe eines schnellen Summierers addiert. Abbildung 8.2 verdeutlicht dieses:

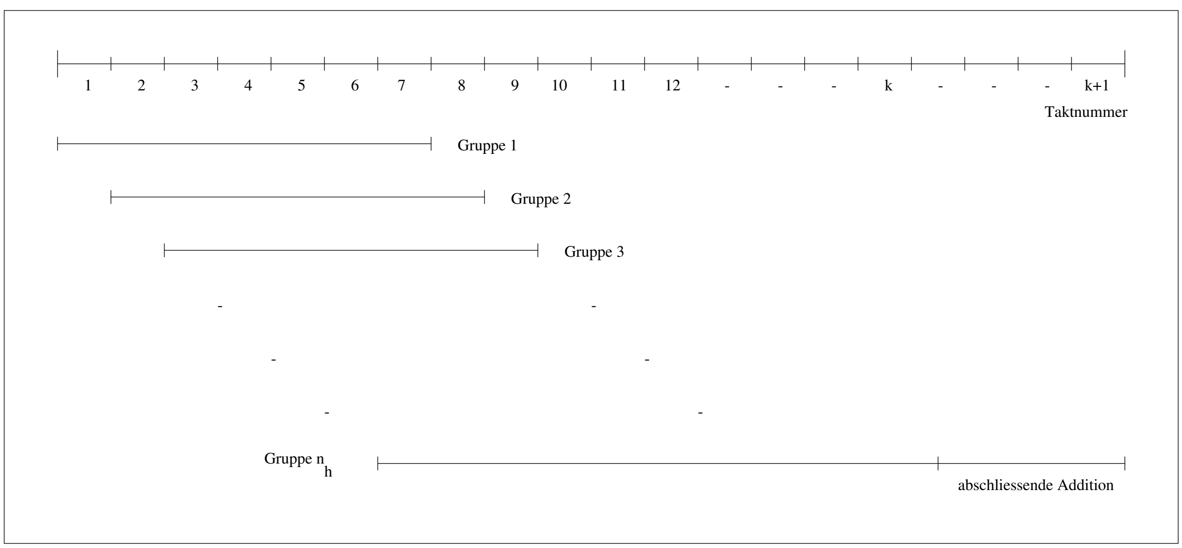


Abbildung 8.2 : Aufbau und Struktur einer Multiplikation mit Pipelinestufen

Um eventuelle Hazards[1] zu vermeiden und für den gesamten Aufbau der Pipelinestruktur, müssen verschiedene Stufen durch Latches[2] oder Flip Flops voneinander getrennt werden. Hierbei müssen Setup und Holdzeiten eingehalten werden was sich negativ auf die Höhe der Taktrate des Gesamtsystems auswirkt.

[1] Hazards sind kurze Signalflanken, die u.a. durch Laufzeitdifferenzen anderer Signale entstehen können.
[2] Latches sind Speicherelemente und werden eingesetzt, um ein oder mehrere Bits zu speichern.

9 Literaturverzeichnis

[Bedrij62]
Bedrij O. J.: Carry Select Adders, IRE Trans. Electron. Comput., Vol EC-11, S. 340-346, 1962

[Brand93]
Brand Hans Jürgen: Spezifikation und Synthese von arithmetischen Datenpfadkomponenten: VDI Verlag 1993

[DSP56002]
DSP56002 User's Manual: User Manual for the DSP56002 series: Motorola 1993

[ATMEL ES2]
ES2 ECPD07 library Databook: Version Nr. E02A07, September 1997

[Hamming83]
Hamming R.W.: Digital Filters: Prentice-Hall Signal Processing Series, second edition, 1983

[Harris91]
Harris Semiconductor DSP Produkts: Digital Signal Processing Databook 1991

[Slansky60]
Slansky J.: Conditional Sum Addition Logic, IRE Trans. Electron. Comput., Vol. EC-11, S. 340-346,1960

[Koren93]
Koren Israel: Computer Arithmetic Algorithms: Prentice-Hall Inc., 1993

[Maeder93]
Maeder, Andreas: VHDL Kurzbeschreibung zum Projekt „VLSI-Entwurf“, 1993

[Motorla89]
Motorola Semiconduktor: Cascadable-Adaptive Finite-Impulse-Response(CAFIR) Digital Filter Chip: Motorola 1989

[Omondi94]
Omondi Amos R.: Computer Arithmetic Systems: Prentice-Hall Inc., 1994

[Pezaris71]
Pezaris S. D.: A 40 ns 17-bit-by-17-bit Array Multiplier, IEEE Trans. on Computers, Vol. C-20, No.4,S. 442-447, 1971

[Pirsch96]
Pirsch P.: Architekturen der digitalen Signalverarbeitung: Teubner Stuttgart, 1996

[Weste/Eshraghian93]
Weste N. H. E. Eshraghian K.: Prinziples of CMOS VLSI Design, Second Edition 1993

[Spaniol76]
Spaniol Otto: Arithmetik in Rechenanlagen Logik und Entwurf: Teubner Stuttgart 1976

[Tietze93]
Tietze Ulrich, Schenk Christoph: Halbleiter Schaltungstechnik: Springer, 10. Auflage 1993

[Wallace64]
Wallace C. S.: A Suggestion for a fast Multiplier, IEEE Trans. on Electr. Comput., Vol EC-13, S. 14-17, 1964

10 Anhang