4 VLSI-Algorithmen als vierter Weg

Vor der Verbreitung der Mikroelektronik konnte man zwei elementare Arten der Implementierung von Algorithmen unterscheiden: Hardware oder Software. Zur Software-Implementierung wird ein Programm für einen Rechner entwickelt. Infolge verschiedener Phänomene um das von-Neumann-Paradigma, die oft unter dem Begriff von-Neumann-Engpaß *(von Neumann bottleneck)* zusammengefaßt werden [3], ist eine Software-Realisierung relativ ineffizient. Dies kann bei sehr komplexen Anwendungen zu unannehmbar langen Antwortzeiten führen. Ein Vorteil der Software-Realisierung ist die Vermeidung der Kosten einer Hardware-Entwicklung. Der atemberaubende Fortschritt der Technologie bei Mikroprozessoren [1] [6] (vgl. auch Bild 1.10) ist auf die Dauer kein Argument, denn der Technologie-Fortschritt kommt letztlich auch der direkten Hardware-Realisierung zugute, und: die Ansprüche der Anwender steigen mit dem Fortschritt der Möglichkeiten.

Klassische Digital-Hardware. Die traditionelle Alternative zur Software war klassische Digital-Hardware, die nach Anzahl der aus einem Katalog ausgewählten Bausteine optimiert wurde (Bild 4.1). Entsprechend wurden Methoden der Logik-Optimierung entwickelt mit dem Ziel der Minimierung der Anzahl der benötigten Logik-Gatter. Bei den niedrigen Integrationsdichten der 60-er- und 70er-Jahre (vgl. Bild 1.12) war die Minimierung der Gatter-Anzahl ein direkter Weg zur Optimierung der Anzahl benötigter integrierter Bausteine.

Analog-Technik. Ein dritter Weg ist die Realisierung eines Algorithmus durch Analog-Technik: durch Programmierung eines Analog-Rechners oder den Entwurf einer Analog-Schaltung. Analog-Rechner [5] und deren Techniken sind heute jedoch praktisch meist bedeutungslos. Andererseits liegt analoge Schaltungstechnik jenseits des Rahmens dieses Buches.

VLSI: planare Hardware. Die Mikroelektronik ermöglicht eine vierte, grundlegend andersartige Form der Implementierung von Algorithmen, nämlich in Form sogenannter ***VLSI-Algorithmen***, die ich wegen ihrer strengen Flächen-Implementierung auf der ebenen Oberfläche eines Silizium-Chips mit dem Begriff ***planare Algorithmen*** zusammenfassen möchte. In ihrer konsequentesten Form liegt eine planare Realisierung bei sogenannten Voll-Kundenschaltungen *(full custom circuits)* vor, im Gegensatz zu den sogenannten ASICs (die wiederum einen Kompromiß in Richtung klassischer Hardware bedeuten, vgl. Kapitel 3 und 14). Wichtigstes Optimierungs-Kriterium bei planaren Algorithmen ist der Flächenbedarf.

Strukturierter VLSI-Entwurf. Da bei planaren Algorithmen die Verdrahtung meist sehr viel mehr Fläche verbraucht als Transistoren, wurden unter dem Begriff strukturierter VLSI-Entwurf ([9], s. Kapitel 19) besondere Methoden zur möglichst weitgehenden Vermeidung globaler

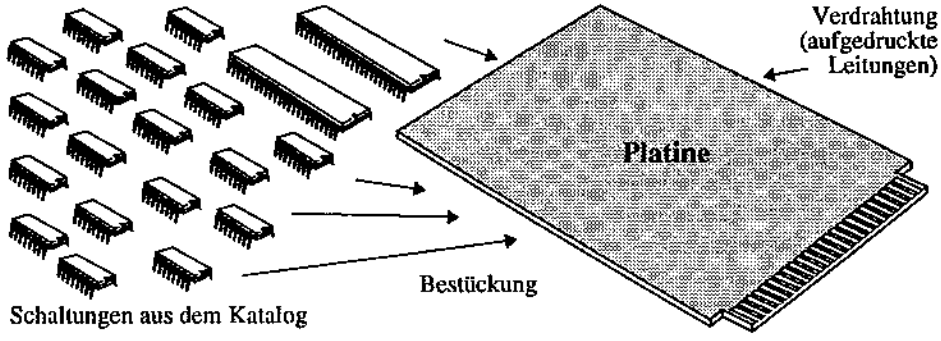

Bild 4.1: Baustein-orientierter Hardware-Entwurf über Stückliste und Verdrahtungsliste.

Verdrahtung (Langstrecken-Verdrahtung) entwickelt (s. auch Abschnitt 11.4 und Kapitel 18 und 20). Im folgenden wird die planare Realisierung am Beispiel des Prioritäts-Operators anschaulich eingeführt und den anderen Realisierungs-Möglichkeiten gegenübergestellt.

4.1 Ein Beispiel: der Prioritäts-Operator

Die Entwicklung der neuen Methodologie für den VLSI-Entwurf, hat praktisch zu den drei klassischen Methoden der Implementierung von Algorithmen eine grundlegend andere, vierte Methode hinzugefügt. Dies sei anhand eines einfachen Entwurfs-Problemes veranschaulicht. Ein Prioritäts-Operator sei in Software, in klassischer Hardware und dann noch auf VLSI-gerechte Weise realisiert. Anmerkung: unter VLSI werden üblicherweise nur integrierte Digitalschaltungen verstanden, da reine Analogschaltungen innerhalb der Grenzen eines Chip die Zahl von 30000 Transistoren meist bei weitem nicht erreichen.

Zweck des Prioritäts-Operators. Zuvor sei jedoch erst die Anwendung des Prioritäts-Operators veranschaulicht. Der Prioritäts-Operator wird z.B. als Arbiter (Entscheider) für die Vergabe-Strategie bei der Steuerung eines Bus eingesetzt (Bild 4.2a), da ein Bus (Datenbus) zu einer bestimmten Zeit stets nur von einem einzigen Sender benutzt werden kann. Jedes Gerät Nr. i hat einen Anforderungs-Bit-Ausgang r_i (r = request), den es zum Melden einer Anforderung des Busses auf "1" setzt (Bild 4.2b). Die Bus-Steuerung meldet das Gewähren durch eine "1" am Zuteilungsbit-Eingang g_i des Gerätes (g = grant). Bild 4.2c faßt die Werte dieses Signalpaares zusammen. Wenn mehrere Sender den Bus anfordern (der Anforderungsvektor R enthält mehrere "1-en"), entscheidet ein Prioritäts-Operator über die Vergabe des Busses (Zuteilungsvektor G enthält nur eine einzige "1"). Als Beispiel diene ein Bus mit 4 Geräten (Bild 4.2a), die alle senden können, also potentielle Benutzer dieses Busses sein können.

Ein 4-Bit-Beispiel. Die 4-Bit Prioritäts-Funktion unseres Beispiels (Bild 4.2d) habe als Eingang einen 4-Bit-Anforderungsvektor $R = (r_1, r_2, r_3, r_4)$ (Wertevorrat je Bit: {0, 1} mit 0 = "keine Anforderung" und "1" = "Anforderung") und als Ausgang einen 4-Bit-Zuteilungsvektor $G = (g_1, g_2, g_3, g_4)$ (Wertevorrat je Bit mit "0" = "keine Zuteilung" und "1" = "Zuteilung") (Bild 4.2c). Beispielsweise $r_3 = 1$ heißt, Gerät Nr. 3 fordert den Bus an, und $g_1 = 1$ heißt, das

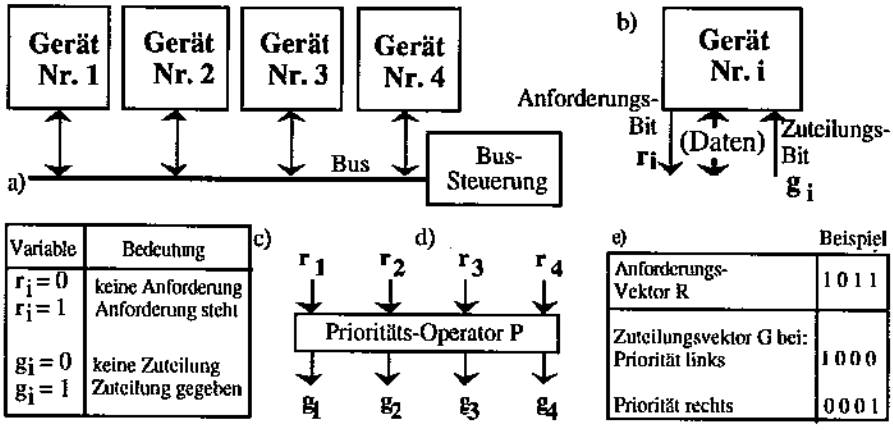

Bild 4.2: Aufgabe des Prioritäts-Operators: a) Bus mit angeschlossenen Geräten, b) Anforderungsbit und Zuteilungsbit bei einem Gerät, c) Legende zu (d), Blockdiagram, e) Ausschnitt aus dem Beispiel einer Funktionstabelle.

Gerät Nummer 1 bekommt den Bus zugeteilt. Die Prioritätsfunktion sorgt dafür, daß im Zuteilungsvektor höchstens eine einzige "1" vorkommt, auch dann wenn der Anforderungsvektor mehrere Einsen enthält. (Beispiel in Bild 4.2e: R = 1011 bewirkt G = 1000, also: "Priorität von links" oder G = 0001 falls "Priorität von rechts").

Klassische Lösungsansätze. Nun wollen wir diese Prioritätsfunktion zunächst nach zwei klassischen Methoden realisieren, einmal per Software, einmal per klassischer Hardware. Diesen Lösungen wollen wir dazu eine neue dritte gegenüberstellen: die *Chipware*-Lösung, eine planare Lösung.

Die Software-Lösung. Bild 4.3a zeigt das Flußdiagramm der Software-Lösung zum 4-Bit-Beispiel eines Prioritäts-Operators. Zu Beginn wird der Zuteilungsvektor (etwa in einem Register) zu Null gesetzt. Dann wird Bit für Bit des Anforderungsvektors der Reihe nach abgefragt, solange bis eine Eins gefunden worden ist (z. B. $r_i = 1$). Im Zuteilungsvektor wird sodann dasjenige Bit g_i zu Eins gesetzt, welches den aktuellen Index hat, während die übrigen Bits den Wert Null beibehalten. Es liegt eine sequentielle Realisierung des Prioritäts-Operators vor.

Die klassische Hardware-Lösung. Die klassische Hardware-Lösung ist Baustein-orientiert: man nehme den Chip-Katalog eines Herstellers und suche sich da die passenden Chips heraus, z.B. TTL-Bausteine. Für unseren Prioritäts-Operator wären dies beispielsweise 2 AND-Gatter mit 3 Eingängen, 3 AND-Gatter mit 5 Eingängen, sowie 4 Inverter (Bild 4.3b). Wir haben also 3 Typen von Gattern und realisieren nun mit Hilfe der Methoden des Logikentwurf die Prioritätsfunktion. Das wichtigste Optimierungs-Kriterium ist die Zahl der Bausteine. In der klassischen Hardware-Theorie stehen Optimierungsverfahren entsprechender Art im Vordergrund. Das wichtigste Zwischenergebnis ist also ein optimierter Logik-Plan. Das eigentliche Endergebnis des Entwurfsprozeß ist die Stückliste und die Verdrahtungsliste. Mit diesen Daten kann dann die Fertigung vorbereitet werden.

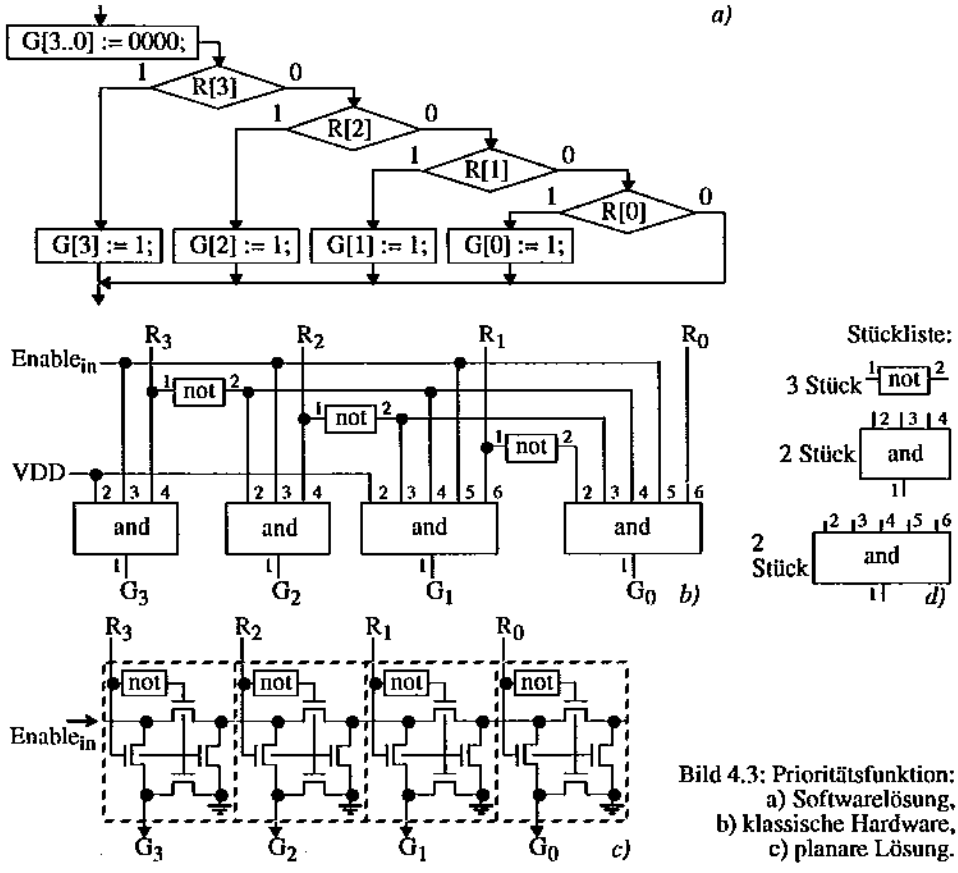

Bild 4.3: Prioritätsfunktion:
a) Softwarelösung,
b) klassische Hardware,
c) planare Lösung.

Die VLSI-Lösung. Die klassische Hardware-Theorie ist eine Disziplin, die inzwischen sehr umfangreich geworden ist. Sie ist jahrzehntelang gewachsen. Das am stärksten im Vordergrund stehende Optimierungs-Kriterium in der klassischen Hardware-Theorie betrifft die Anzahl der aktiven Elemente, oder die Anzahl der Bausteine, wie z.B. der logischen Gatter. Für den VLSI-Entwurf hingegen sind andere Kriterien wichtiger (besonders der Flächenbedarf, da mit steigender Chipfläche die Fertigungsausbeute zurückgeht). Anstelle einer Hardware-Lösung wird eine planare "Chipware-Lösung" benötigt. Hier haben nicht die aktiven Elemente (die Transistoren) den größten Flächenverbrauch. Vielmehr hat die Verdrahtung auf dem Chip einen sehr viel höheren Flächenbedarf, insbesondere, wenn es sich um "Langstrecken-Verdrahtung" handelt. Es wird eine neue Entwurfs-Theorie benötigt.

***Chipware*-Lösung der Prioritäts-Funktion** Die Grundlage der VLSI-gerechten Lösung besteht nur aus einer 1-Bit-Zelle (linke Seite in Bild 4.3c). Prioritäts-Operatoren beliebiger Wortlänge können durch wiederholtes "Aneinanderstecken" von Kopien dieser 1-Bit-Zelle

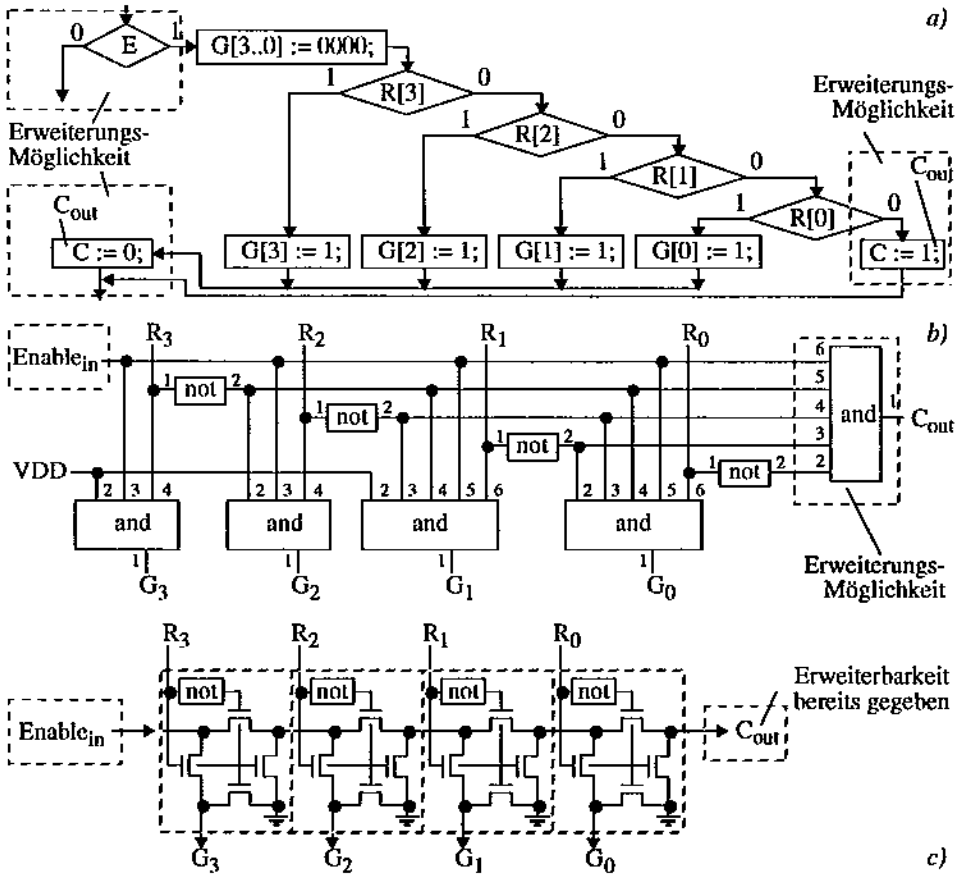

Bild 4.4: Erweiterbare Versionen zum Prioritäts-Operator in Bild 4.3; a) Erweiterbare Softwarelösung, b) Erweiterbare klassische Hardwarelösung, c) Erweiterbare VLSI-Lösung.

realisiert werden (Ersparnis an Verdrahtungsfläche über Verbindung durch Anstoßen: *wiring by abutment*). Die rechte Seite in Bild 4.3c zeigt ein Beispiel mit 4 Kopien (auch *Slices* genannt) dieser Zelle: es entsteht ein Prioritäts-Operator mit einer Pfadbreite von 4 Bit.

Wiring by Abutment. Das Duplizieren und Aneinanderstecken dieser Zellenkopien können wir auch per Computer mit CAD-Werkzeugen leicht durchführen. Der Ausgang C einer Zelle, paßt gleichzeitig zu ihrem eigenen Eingang E. Wenn die Zelle kopiert wird, ist der Ausgang C der Zelle der Eingang E ihrer eigenen Kopie. Das Problem der Verdrahtung hierbei wurde durch *wiring by abutment* [7] gelöst ("verbinden durch aneinanderstoßen").

Erweiterbare Schaltungs-Familienplanung. Außerdem ist die Schaltung auf flexiblere Weise erweiterbar (extensible), im Vergleich zur Lösung nach Bild 4.3. Wir können von der

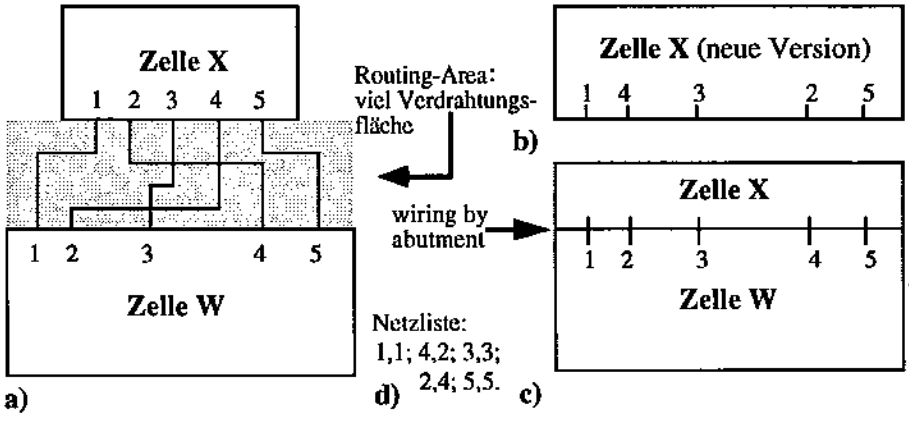

Bild 4.5: Veranschaulichung der Vorteile des "wiring by abutment". a) unstrukturierter Entwurf, schlechte Lösung, b) topologisch geänderte Version der Zelle X c) bessere Lösung: wiring by abutment durch pitch matching, d) Netzliste (Netz-Format: #(X),#(W);)

Grundzelle "parity slice" beliebig viele Kopien nehmen. Somit liegt also ein nahezu idealer erweiterbarer Entwurf vor. Eine solche Erweiterbarkeit ist wichtig für eine schnelle Produktstrategie. Angesichts einer heute durch starken Wettbewerb bedingt oft sehr kurzen Produktlebensdauer ist eine rasche Pflege der Produktfamilie enorm wichtig. Flexibilität durch Parametrisierung und Erweiterbarkeit soll erlauben, daß ohne jeweils kompletten Neu-Design eine Produktpalette rasch immer wieder fortentwickelt werden kann.

Strukturierung durch Bit-Slices. Sogenannter *strukturierter Entwurf* mit Zellen, die man in beliebig vielen Wiederholungen aneinandersetzen kann, führt dann zu in sogenannte Bit-Scheiben (bit slices) gegliederten Architekturen, die vor allen Dingen bei Datenpfaden (data paths) in Computer-Schaltungen (Prozessoren) häufig vorkommen. Bei der ALU (arithmetic/logic unit: arithmetisch-logische Einheit) und anderen Teilen von Mikrorechnern kann man deshalb meist sehr gut mit der Methode des strukturierten Entwurfs arbeiten. Die Technik dieses Entwurfs-Stils ist im folgenden Abschnitt veranschaulicht.

Erweiterbare Schaltungen. Über den Eingang C_{in} kann der oben beschriebene Prioritäts-Operator aktiviert werden. Der Ausgang C_{out} erzeugt eine "1" für den Fall, daß keine Zuteilung erfolgt. Hier kann der C_{in}-Eingang eines weiteren Exemplares des Prioritätsbausteines angeschlossen werden. Auf diese Weise kann der Operator für 4 Bit auf eine Pfadbreite von 8 Bit, 12 Bit etc. erweitert werden. (vgl. Bild 4.4a bis Bild 4.4c). Somit liegt eine erweiterbare Schaltung vor.

Produkt-Familienplanung. Das Anstreben von Erweiterbarkeit ist eine wichtige Entwurfs-Strategie zur Erreichung einer möglichst langen Produkt-Lebensdauer, bzw. Lebensdauer einer Produkt-Familie. Eine solche Erweiterbarkeit der Schaltung bringt die erforderliche Flexibilität zur Anpassung des Produktes an eine Weiterentwicklung des Marktes.

4.2 Strukturierter VLSI-Entwurf

Die planare Version des obigen Prioritäts-Operators ist bereits ein einfaches Beispiel eines strukturierten VLSI-Entwurfes. Ein wichtiges Ziel des strukturierten Entwurfes ist aus o. g. Gründen die Erreichung von ***wiring by abutment*** zur Minimierung bzw. lokalen Eliminierung oder Vermeidung von Verdrahtungs-Kanälen. Bild 4.5 verdeutlicht dies wie folgt. Bild 4.5a zeigt einen ersten Lösungs-Ansatz als Ausschnitt einer Chip-Planning-Bemühung. Zwei einander gegenüberliegende Zellen W und X müssen nun miteinander verdrahtet werden. Bild 4.5a zeigt deutlich den "Verdrahtungs-Kanal" (routing channel) zwischen den beiden Zellen. Wie man sieht, wird hier ein beträchtlicher Prozentsatz der Fläche für reine Verdrahtung verbraucht. Wir wollen nun die Verdrahtungsfläche durch einen Neuentwurf der Zelle X minimieren: es entsteht deren neue Version wie Bild 4.5b zeigt. Diese Version wurde derart gestaltet, daß die Anschluß-Punkte zu denen der Partnerzelle W (Pins Nr. 1 bis 5) genau passen, sowohl in der Reihenfolge, als auch in der Geometrie: es wurde also *port matching* zwischen den beiden Zellen erreicht.

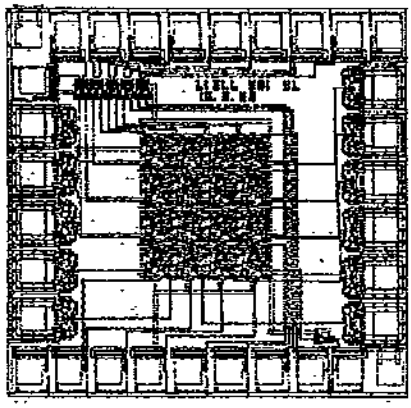

Bild 4.6: Strukturiertes Layout (Voll-Kundenschaltungs-Version) zum Verdrahtungsproblem nach Bild 4.7.

Voraussetzung für Wiring by Abutment. Dies ist die Voraussetzung für *wiring by abutment* (gezeigt in Bild 4.5c), wobei keine separate Fläche für Verdrahtung benötigt wird (vgl. auch obigen Abschnitt dieses Textes). Ein solcher Entwurfs-Stil spielt bei modernen Entwurfsmethoden eine ganz wichtige Rolle. Man spricht dann auch von ***strukturierten Entwurf***, wenn man in ganz hohem Maße wiring by abutment angewendet wird. Ein wichtiger Gesichtspunkt beim strukturierten Entwurf ist also das wiring by abutment.

Neue Optimierungskriterien. Warum sind die Methoden der klassischen Hardware-Theorie jetzt plötzlich bei der Chipware nicht mehr ausreichend? Die Verdrahtung und weniger die Anzahl der Transistoren ist ein Problem, das Kummer macht - wie wir gesehen haben. Man muß also nicht in erster Linie die Zahl der Transistoren optimieren, sondern es muß vielmehr die Fläche, die für die Verdrahtung verbraucht wird, minimiert werden. Die klassische Hardware-Theorie ist hierzu nicht hilfreich, da diese nach Stückzahl anstatt nach Fläche optimiert. Dieses Beispiel verdeutlicht, weshalb für VLSI-Algorithmen neue Entwurfsverfahren, ja sogar ein ganzer neuer Wissenschafts-Zweig (in USA genannt: Design Sciences) entwickelt werden müssen. Vielversprechende Ansätze dazu sind bereits vorhanden, die z.T. auch schon in recht leistungsfähigen CAD-Programme implementiert wurden.

Ein Multiplizierer als Beispiel. Bild 4.6 zeigt das Layout eines einfachen 4-mal-4-Bit-Multiplizierers im Stil der Voll-Kundenschaltungen, der in Kaiserslautern aus einer studentischen Übung hervorgegangen ist. Der Kern der Schaltung ist ein 4 mal 4 Zellen großes 2-dimensionales Feld von 1-mal-1-Bit-Multiplizierern, die durch *wiring by abutment* miteinander verbunden sind. (Diese Schaltung wird in Abschnitt 20.4.2 detailliert behandelt; vgl. auch

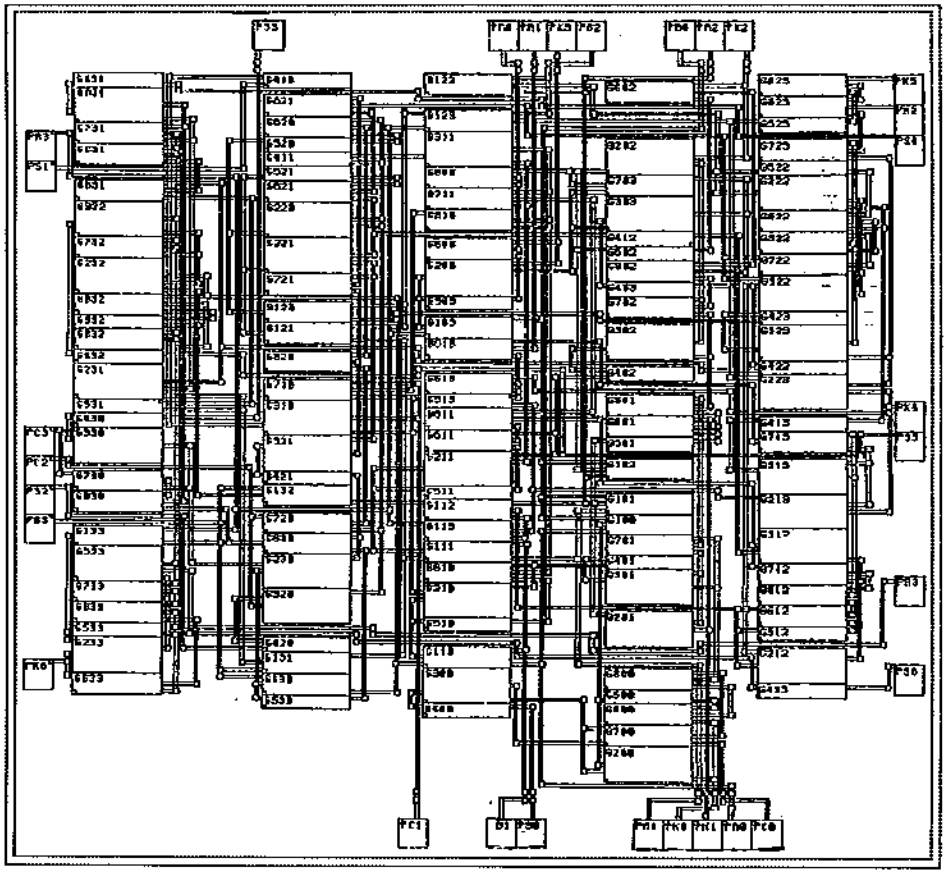

Bild 4.7: Mit Routing- und Placement-Programm (Silvar-Lisco) erzeugte Version des Multiplizierers nach Bild 4.6: automatisch generierte Verdrahtung bei automatisch plazierten Zellen.

Bild 20.34.) Bei den Zellen am Rande des Chip nach Bild 4.6 handelt es sich nur um Eingangs- und Ausgangs-Treiber mit Bonding-Pads (vgl. beispielsweise Bild 13.21).

Ein Gegen-Beispiel. Zur Demonstration der enormen Flächen-Ersparnis durch strukturierten Entwurf, soll dieser mit dem Gegen-Beispiel nach Bild 4.7 und Bild 4.8 verglichen werden. Bild 4.7 zeigt eine mit einem Routing- und Placement-Programm (Baustein-orientiert: s. a. Seite 294 in Kapitel 14) erzeugte Layout-Version des Multiplizierers nach Bild 4.6 (Bild 4.8 b zeigt letzteren im gleichen Maßstab zum Vergleich des Flächenbedarfs). Die Zellen wurden automatisch plaziert und dann automatisch verdrahtet. Bild 4.8 a zeigt eine durch von Hand optimierte Zellen-Plazierung verbesserte Version des Layout nach Bild 4.7. Hierbei wurden

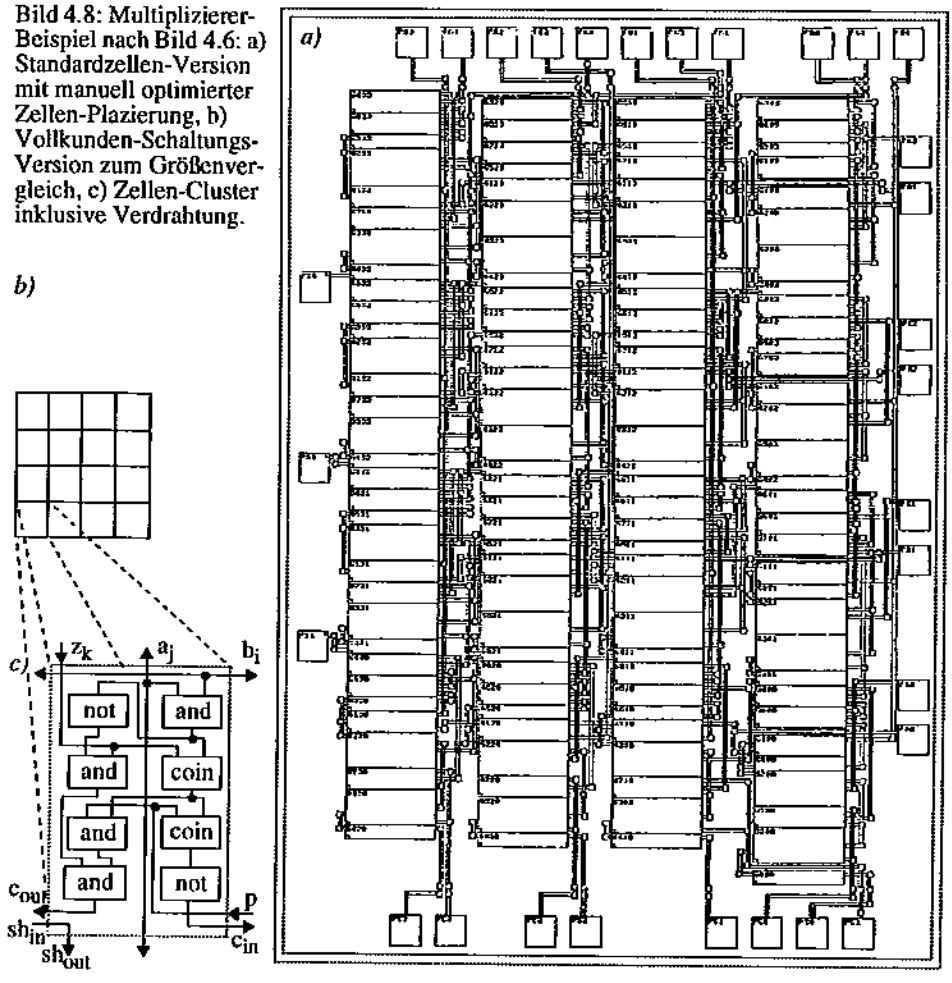

Bild 4.8: Multiplizierer-Beispiel nach Bild 4.6: a) Standardzellen-Version mit manuell optimierter Zellen-Plazierung, b) Vollkunden-Schaltungs-Version zum Größenvergleich, c) Zellen-Cluster inklusive Verdrahtung.

Erfahrungen aus dem strukturierten Entwurf (Bild 4.6) verwertet: die jeweils zu einem 1-mal-1-Bit-Multiplizierer gehörigen 8 Gatter-Zellen (vgl. Bild 4.8 c) wurden als Cluster nebeneinander plaziert.

Vergleich der drei Multiplizierer-Versionen. Die Tabelle in Bild 4.10 vergleicht den Flächenbedarf der drei obigen Lösungen relativ zur Lösung in Bild 4.7. Die von Hand optimierte Standardzellen-Lösung (Bild 4.8) spart mehr als 25% der Fläche ein. Bild 4.6 zeigt deutlich die hohe Flächeneffizienz des Strukturierten Entwurfes bei Voll-Kundenschaltungen. Der Kern wurde auf nur 2,7% reduziert. Die Gesamtschaltung wurde jedoch nur auf 18% reduziert, weil die nur einreihig erlaubte Anordnung der Anschluß-Pads auf Grund ihrer Anzahl die im

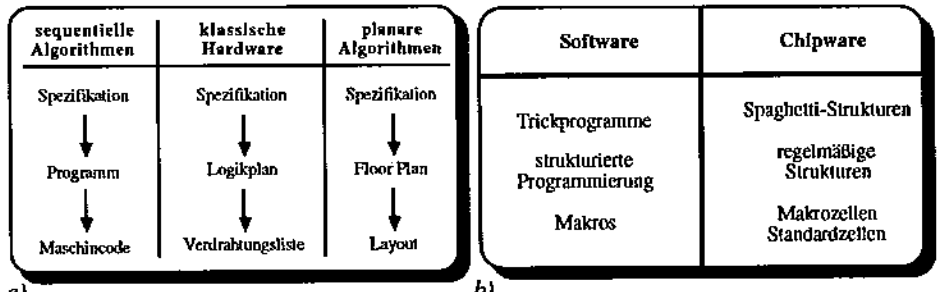

Bild 4.9: Hardware gegen Software: a) Gegenüberstellung der drei Grundmethoden der Implementierung digitaler Algorithmen-, b) Gegenüberstellung von Software und Chipware.

Bild 4.6 sichtbare Ausdehnung der Chip-Peripherie erzwingt. Demgegenüber hat die Standardzellen-Lösung durch automatischen Entwurf den Vorteil niedriger Entwurfskosten und kurzer Entwicklungszeit. Letzteres ist bei der heutigen oft schnellen Marktentwicklung und dadurch raschen Veraltung von Produkten ein entscheidender Vorteil.

Vergleich der drei Methodologien. Wir fassen unsere drei Entwurfsverfahren noch einmal zusammen. Normalerweise beginnt ein Entwurf mit einer Spezifikation. Im weiteren Verlauf bekommt man wichtige Zwischenergebnisse, die immer weiter optimiert werden bis schließlich das Endprodukt vorliegt (vgl. Bild 4.9 a). Bei der Software ist dies der Maschinen-Kode, den der Compiler automatisch erzeugt und beim klassischen Hardware-Entwurf die Verdrahtungsliste, die sich durch direkte Ableitung aus dem Logikplan ergibt. Der Logikplan ist in diesem Fall das Entwurfs-Medium. Bei der Realisation von planaren Algorithmen beginnen wir mit einer Spezifikation und das wichtigste Zwischenergebnis an dem man einen guten Entwurf erkennt, ist der sogenannte *floor plan* (auf deutsch Grundrißplan).

Flächen-Nutzungsplan. Das könnte man auch direkt als Flächennutzungsplan übersetzen. In einem solchen *floor plan* lassen sich die eigentlich aktiven Zellen von den passiven leicht unterscheiden und somit feststellen, ob der Entwurf großen oder geringen Verbrauch an Verdrahtungsfläche hat. Aus diesem Kriterium ergeben sich Ansatzpunkte für die Optimierung des Entwurfs. Nach dem Entwurf des floor plans wird der Entwurfsprozeß mit der Verfeinerung der Zellen solange fortgesetzt, bis zum Schluß das Layout vorliegt. Das Layout besteht aus geometrischen Mustern, die einem Hersteller, z.B. auf einem Datenträger, übergeben werden. Der Hersteller überträgt diese dann stark verkleinert in einem komplizierten Verfahren auf die Siliziumscheibe (s. Abschn. 9.3 ab S. 198).

Version aus	Kern	Gesamt-Chip
Bild 4.7	100%	100%
Bild 4.8	69%	73%
Bild 4.6	2,7%	18%

Bild 4.10: Vergleich des Flächenbedarf: Multiplizierer-Versionen aus Bild 4.6 bis Bild 4.8.

Drei Arten von Algorithmen (zusammengefaßt). Die drei Grundmethoden der digitalen Algorithmen-Implementierung, d.h. Entwurfsprobleme gelöst werden können sind: Software, klassische Hardware, sowie planare Hardware bzw. Chipware.

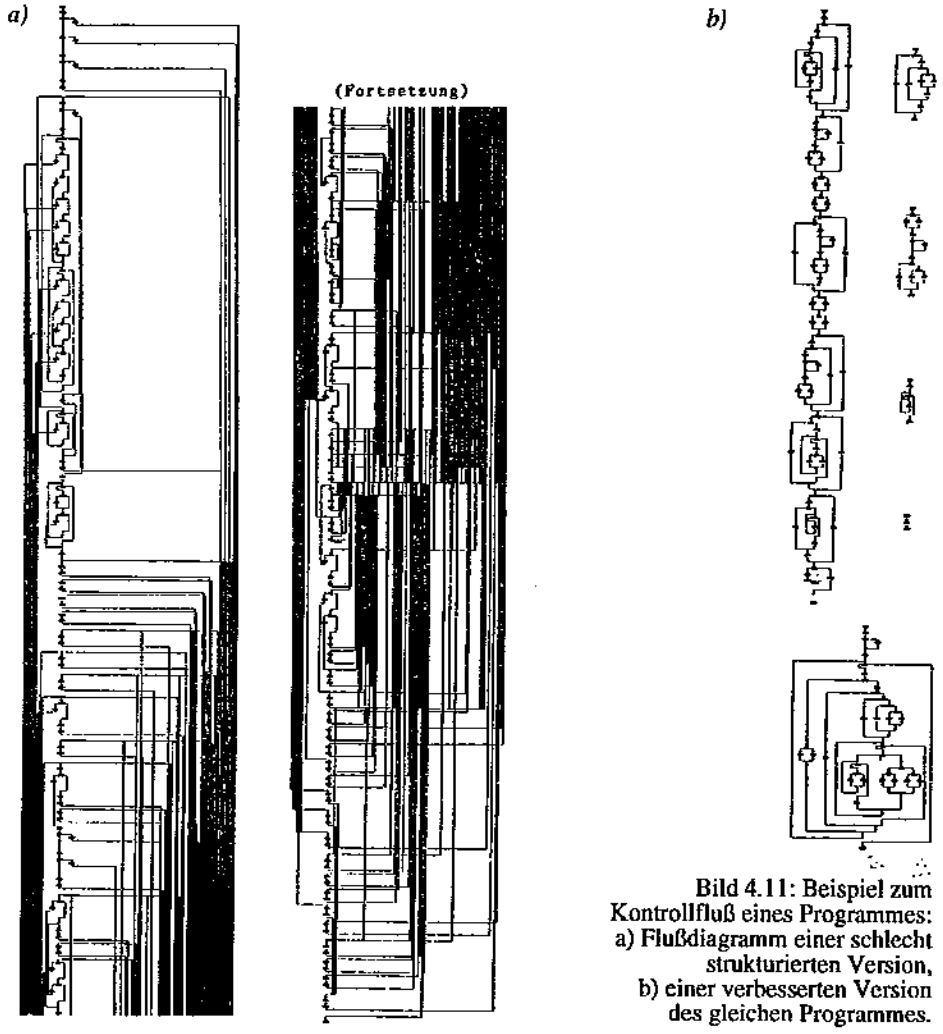

Bild 4.11: Beispiel zum Kontrollfluß eines Programmes: a) Flußdiagramm einer schlecht strukturierten Version, b) einer verbesserten Version des gleichen Programmes.

Sequentiell-zu-planar-Wandlung. Was bedeutet der Begriff planarer Algorithmus? Zur planaren Realisierung muß ein Algorithmusmeist geringfügig geändert werden. Die entsprechend geänderte Variante des Algorithmus nennen wir dann einen planaren Algorithmus, im Gegensatz zu einem sequentiellen Algorithmus (Software-Lösung). In Kapitel 19 wird diese Form der Umwandlung am Beispiel des Bubble-Sort-Algorithmus ausführlich gezeigt.

Parallelen zwischen Chipware und Software. Ich möchte nun veranschaulichen, daß es zwischen Chipware und Software Parallelen gibt: *Structured VLSI-Design* als Pendant zum

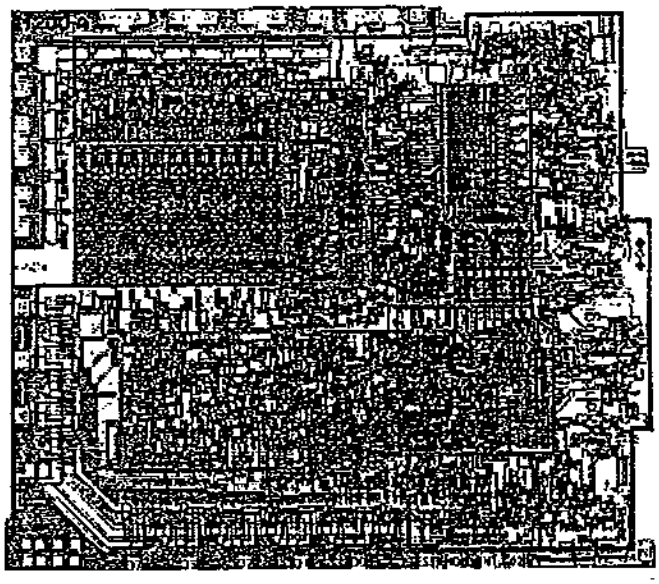

Bild 4.12: Intel-Mikroprozessor 8080.

Structured Programming.. In der Software gibt es viele Trick-Programme (Bild 4.9 b). Dies sind Programme, die man oft wegwerfen muß, wenn der Entwickler nicht mehr greifbar ist, weil außer ihm niemand mehr mit dem Programm vertraut ist. Es gibt auch Chip-Designer mit Entwurfsstilen, die mit den obigen vergleichbar sind, und zu undurchsichtigen Layout-Strukturen führen. Diese werden im Fachjargon gern als *Spaghetti-Strukturen* bezeichnet. Jedoch gibt es auch auf der positiven Seite Parallelen zwischen Software und Chipware. Unter *Structured Programming* entstehende Kontroll-Strukturen kann man vergleichen mit den regelmäßigen Strukturen, die bei *Structured VLSI Design* und ähnlichen Entwurfs-Stilen entstehen. Dies wird im Folgenden veranschaulicht (Bild 4.11).

Strukturierte Programme. Ein gutes oder schlechtes Programm, kann man an der graphischen Darstellung der Kontroll-Struktur erkennen. Bild 4.11 a zeigt die Kontroll-Struktur eines schlecht strukturierten Programms. Diese Struktur läßt sich nicht modularisieren. Man sieht dies daran, daß normalerweise ein Programm nur einen einzigen "entry point" und nur einen einzigen "exit point" haben sollte. In der Struktur aus Bild 4.11 a gibt es keine Stelle, wo man unter vernünftigem Aufwand einen Schnitt hindurchlegen könnte. Das ist also die Kontroll-Struktur eines schlechten Programms. Bild 4.11 b zeigt die graphische Darstellung des Kon-

integrierte Schaltung	Regelmäßigkeits-Faktor	Entwurfszeit	Zeit zur Fehlerbeseitigung
Z80	≈1	18 Monate	8 Monate
68000	20	9 Monate	3 Monate
Gleitkomma-Einheit [8]	75		

Bild 4.13: Beispiele von Regelmäßigkeits-Faktoren bei integrierten Voll-Kunden-Schaltungen.

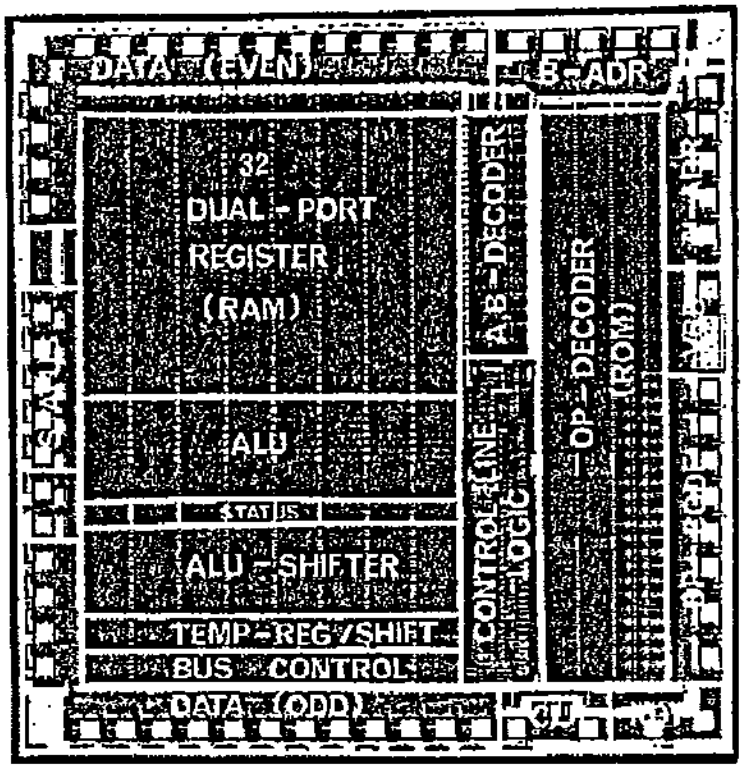

Bild 4.14: Mike 83 Mikroprozessor (nach einem Werkfoto der Siemens AG).

troll-Flusses einer neu strukturierten Version des gleichen Programms. Hier können an vielen Stellen auf sehr einfache Weise Schnitte durch den Graphen gelegt werden, sodaß eine gute Modularisierung leicht realisiert werden kann.

Strukturiertes Layout in der Industrie. Bild 4.12 zeigt das Layout des Intel-Mikroprozessors 8080, der um ca. 1973 auf den Markt kam. Bild 4.14 zeigt die Mikrophotographie des Mike-83-Mikroprozessor-Chip. Dieser ist das Ergebnis eines Pilotprojekts, das in der Einführung einer verbesserten CAD-Umgebung in den Zentrallaboratorien der Siemens-AG bestand. Sehr interessant ist ein Vergleich dieser beiden Layouts. Das Layout des Mike-83 ist hoch regelmäßig, sodaß ein hochgradig strukturiertes Layout vorliegt. Beim ca. 10 Jahre älteren Design des 8080 überwiegen die schon angesprochenen *Spaghetti-Strukturen*. Nur der Register-File sieht relativ regelmäßig aus da man einen Register-File - selbst mit Absicht - kaum in Spaghetti-Strukturen realisieren kann.

Der Regelmäßigkeitsfaktor als Qualitätsmaß Bild 4.12 und Bild 4.14 zeigen, wohin die Tendenz geht. Dies läßt sich auch durch Zahlen wie folgt ausdrücken: Man definiert ein Regelmäßigkeitsmaß. Der *Regelmäßigkeitsfaktor* RF gibt die Gesamtzahl der Transistoren, dividiert durch die Anzahl der individuell plazierten Transistoren an. Im Fall, daß überhaupt keine Re-

gelmäßigkeit vorliegt, ist der Regelmäßigkeitsfaktor 1. Bild 4.13 stellt Beispiele von Regelmäßigkeitsfaktoren einander gegenüber und zeigt den Einfluß dieses Faktors auf den Entwurf.

Der Begriff "individuell plaziert" sei im Folgenden erklärt. Nehmen wir die Prioritätsfunktion zusammengesetzt aus 4 gleichartigen Zellen (vgl. Bild 4.4c). Innerhalb einer der 4 Zellen-Kopien ist kein Transistor individuell plaziert worden. Allgemein, wenn man die eine Zelle viermal, achtmal, sechzehnmal wiederholt, dann gehört ein Transistor nicht zu den individuell plazierten, sondern er befindet sich innerhalb einer regelmäßigen Struktur. Wenn man aber noch einen extra Transistor außerhalb dieses iterativen Zellen-Feldes anschließt, der auf dem ganzen Chip nur einmal vorkommt (in Bild 4.4 c nicht zu sehen), dann wäre dieser ein "individuell plazierter Transistor".

So ermittelt man dann den Regelmäßigkeitsfaktor (RF, s. a. Bild 4.13). Der RF wurde von Craig Mudge erstmals definiert [8]. Der Mikroprozessor 8080 von Intel hat einen RF von nur etwa 1. Der Mikroprozessor 68000 von Motorola hingegen ist sehr viel regelmäßiger aufgebaut: er hat einen RF von ca. 20. Es gibt ein Entwurfs-Beispiel speziell vom "Erfinder" des RF. Er hat für DEC (Digital Equipment Corporation) einen Single-Chip Gleitkomma-Prozessor entwickelt mit einem RF von 75. Man sieht deutlich: der RF ist sozusagen ein Gütekriterium für die Durchführung eines strukturierten Entwurfs.

4.3 Literatur

[1] D. Alpert, D. Avnon: Architecture of the Pentium Microprocessor; IEEE Micro 13,3 (June 1992)

[2] R. Hartenstein: Die "Neue Mikroelektronik" in der Informatik: Voraussetzungen und auswirkungen; GI-Jahrestagung, 1981 Kaiserslautern, Springer-Verlag, 1981

[3] R. Hartenstein: Xputers; IT Press Verlag, Bruchsal (in Vorbereitung)

[4] P. Jespers, C. Sequin, F. van de Wiele: Design Methodologies for VLSI Circuits; Proc. NATO-ASI "Very Large Scale Integration", Louvain-la-Neuve 1981, Noordhoff & - Stijthoff, Rockville, Maryland, 1981

[5] G. A. Korn, T. M. Korn: Electronic Analog and Hybrid Computers; McGraw-Hill, 1972

[6] E. McLellan: The Alpha AXP Architecture and 21064 Processor; IEEE Micro 13,6 (June 1993)

[7] C. Mead, L. Conway: Introduction to VLSI Systems; Addison-Wesley, 1980

[8] C. Mudge et al.: A single-Chip Floating-Point Processor - A Case Study in Structured Design; in [4]

[9] Ph. C. Treleaven: VLSI Processor Architectures; Computer, June 1982