Begleitmaterial zur Vorlesung „Rechnergestützter Entwurf von Mikroelektronik“ - Synthesealgorithmen - Sommersemester 2007

Peter Marwedel
Informatik 12
(Technische Informatik & Eingebettete Systeme)
Universität Dortmund

10. Juni 2007

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Synthese mikroelektronischer Systeme

Die moderne Informationsverarbeitung basiert fast ausschließlich auf der elektronischen Verarbeitung von Informationen in sehr kleinen Systemen auf der Basis mikroelektronischer Systeme. Andere Prinzipien der Informationsverarbeitung findet man nur in biologischen Systemen sowie in einigen Konzeptstudien beispielsweise zu Quantenrechnern oder biomolekularen Rechnern. Moderne Systeme sind extrem komplex. Es ist es nicht möglich, die Fertigungsunterlagen für derartige Schaltungen von Hand zu erstellen. Vielmehr ist man darauf angewiesen, diese Unterlagen in Form eines rechnerunterstützten Entwurfs (engl. *computer-aided design* (CAD)) zu erstellen. Entsprechend mussten Entwurfswerkzeuge in der Vergangenheit immer weiter entwickelt werden. So gab es historisch beispielsweise bei der Entwicklung von integrierten Schaltungen einen Übergang von einfachen Werkzeugen zur Eingabe der geometrischen Fertigungsdaten bis hin zur heutigen Werkzeugen, welche viele Entwurfsaufgaben automatisieren. Die Techniken von Werkzeugen zur Erzeugung von mikroelektronischen Systemen sind Gegenstand dieses Textes.

Die Gliederung des Textes entspricht einem vereinfachten, typischen Entwurfsablauf mikroelektronischer Systeme. Gemäß Abb. 1.1 behandeln wir diesen Ablauf im Kontext eingebetteter Systeme.

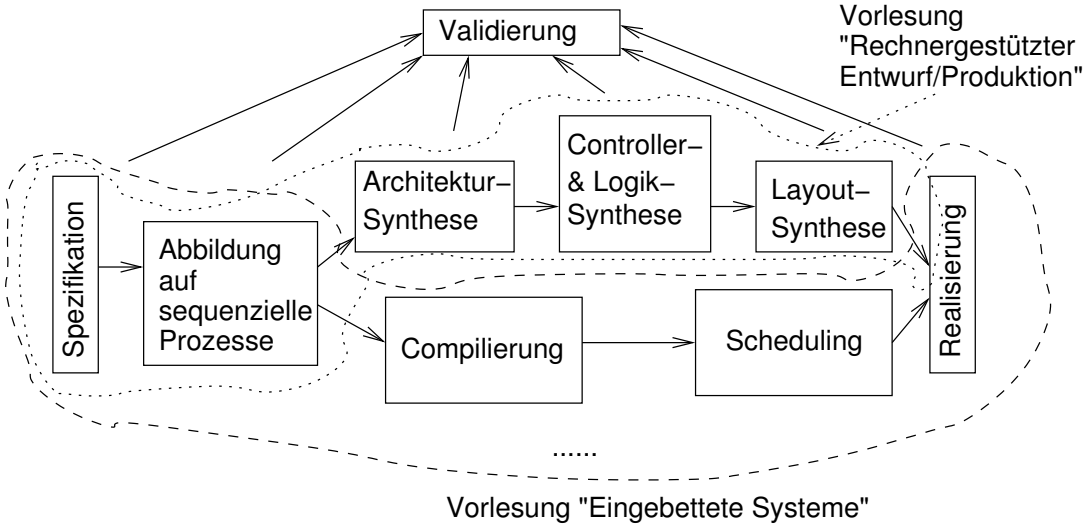


Abbildung 1.1: Vereinfachter Entwurfsablauf

Im Allgemeinen wird für ein solches System sowohl Hardware wie auch Software zu entwerfen sein. Die Entwicklung von Software ist Gegenstand der Veranstaltung „Eingebettete Systeme“. Dieser Text beschränkt sich auf die Entwicklung von Hardware. In Kapitel 2 betrachten wir dazu die so genannte Architektursynthese, die eine Realisierung in Form von **Register-Transfer-Strukturen** (siehe Vorlesung „Rechnerstrukturen“) als Ziel hat. In diesem Zusammenhang gehen wir u.a. das am Lehrstuhl Informatik 12 entstandene Architektursynthesesystem OSCAR [LM+93] ein.

Als nächstes betrachten wir exemplarisch Controller-Syntheseverfahren. Dabei werden wir das ASYL-System [dPD89] vorstellen, welches einen Eindruck von der Wirkung bestimmter Zustandskodierungen vermittelt. Daran schließen sich Logiksynthese-Verfahren an, wobei wir v.a. auf die Techniken im ESPRESSO-Werkzeug[1] eingehen werden.

Für die Realisierung in Form von integrierten Schaltkreisen bildet die Layouterzeugung den letzten Schritt in unserer Entwurfskette. Bei der Darstellung orientieren wir uns am vierten Kapitel des Buchs „Synthese und Simulation von VLSI-Systemen" [Mar93] sowie an dem Buch von Sherwani [She98].

Zu einem Studium gehört unverzichtbar auch das selbständige Lesen von Literatur. Exemplarisch sollte dies trotz des vorliegenden Skripts durchgeführt werden. Ein Skript, welches zur Vernachlässigung des Lesens von Originalliteratur führt, würde den Lernerfolg nicht mehr gewährleisten können. Im Skript sind in einigen Fällen englischsprachige Publikationen integriert. Deren Integration lag aus inhaltlichen Gründen nahe. Sie bieten aber auch eine Gewöhnung an die englische Sprache und das Lesen von Originalliteratur.

Eine der besonderen Herausforderungen beim Entwurf von mikroelektronischen Systemen ist der große Unterschied der Modelle der Spezifikation und der Implementierung. Eine systematische Darstellung dieser und der Modelle auf Zwischenebenen zeigt die Tabelle 1.1.

	Domäne		
Ebene	Verhalten	Struktur	Geometrie, Layout
Applikation	UML *use case*		
Processor/Memory/ Switch (PMS)-Ebene	Kommunizierende Prozessoren	Prozessoren (P), Speicher ("memories", M), Schalter (S)	geometrisch/physikalische Information über ein System (Gehäuse)
Prozesse, *threads*	Variablen, Zuweisungen, Schleifen	Speicherzellen	geometrisch/physikalische Information für einen Prozessor, Aufteilung auf Karten
Register-Transfer (RT)-Ebene	Register-Transfers	Register, RAMs, arithmetische Einheiten (ALUs), Busse	Layout von RT-Bausteinen
Logik-Ebene	Boolesche Gleichungen	Gatter, Flip-Flops	Gatter-Layout, Standardzelle
Schaltkreis-Ebene	Netzwerk-Gleichungen	Kondensatoren, Transistoren, Stromquellen, usw.	Schaltkreis-Layout
...	...	...	...

Tabelle 1.1: Mögliche Modellierungsebenen

Dabei unterscheiden wir zunächst zwischen verschiedenen Bereichen oder **Domänen**:

1. in der **Verhaltensdomäne** wird der Zusammenhang zwischen Eingaben und Ausgaben an ein System beschrieben. Beschreibungen dieser Domäne sind entscheidend für die funktionale Simulation eines Systems.
2. in der **Strukturdomäne** wird beschrieben, aus welchen Komponenten ein System besteht.
3. in der **Geometriedomäne** werden geometrische (und topologische) Informationen über ein System beschrieben.

Zusätzlich haben wir eine Beschreibung auf verschiedenen Ebenen (Zeilen in Tabelle 1.1). Die oberste Ebene ist die der Spezifikation mit den heute üblichen Spezifikationstechniken. Die übrigen Zeilen beschreiben eine immer detailliertere Sichtweise des Systems.

Bemerkenswert ist, dass die Entwurfsaufgabe einem Weg von rechts oben nach links unten in der Tabelle 1.1 entspricht.

[1] Darstellung nach Pusch [Pus90].

Diagramm Ebenen und Domänen kann man sich in recht einprägsamer Form anhand des sog. **Y-Diagrammes** (des *Y-charts*) merken, das von Gajski [GK83] vorgeschlagen wurde (siehe Abb. 1.2).

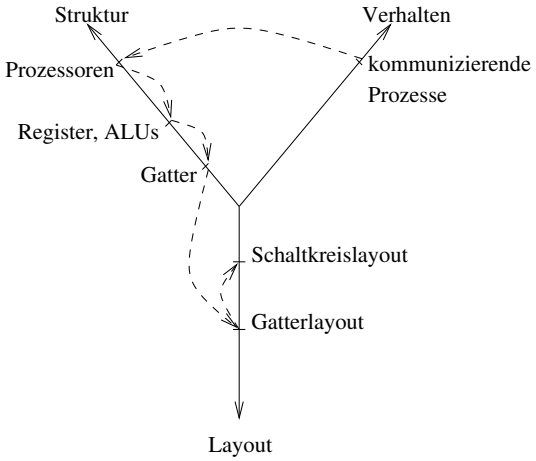


Abbildung 1.2: Y-Diagramm nach Gajski

Auch im Y-Diagramm kann man den Entwurfsprozess durch einen Weg beschreiben (gestrichelte Linien).

Die wesentliche Aufgabe der Konstruktion mikroelektronischer Systeme besteht darin, Komponenten der möglichen Zieltechnologien so auszuwählen und zu kombinieren, dass die Spezifikation realisiert wird (siehe auch Abb. 1.3).

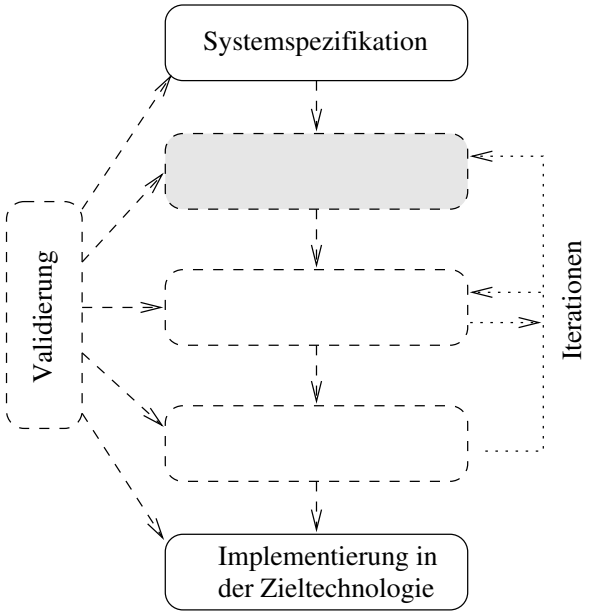


Abbildung 1.3: Erste Schritte zur Implementierung

Hierfür gibt es mehrere Möglichkeiten:

- die automatisierte bzw. die automatische Erzeugung der Realisierung,
- den Entwurf der Realisierung von Hand.

Den ersten Fall bezeichnet man als **Synthese**.

Def.: Synthese ist das Zusammensetzen von Komponenten oder Teilen einer niedrigen (Modell-) Ebene zu einem Ganzen mit dem Ziel, ein auf einer höheren Ebene beschriebenes Verhalten zu erzielen ([Mar93], in Anlehnung an [Hor]).

Ideal wäre es, wenn für alle Entwurfsaufgaben Syntheseverfahren, die effiziente Realisierung erzeugen können, zur Verfügung stehen würden.

Trotz allen Bemühens wird man diesen Idealfall wohl nicht erreichen. Nach H. de Man liegt ist dies prinzipiell begründet:

1. Die CAD-Hersteller sind so mit der Pflege vorhandener Software beschäftigt, dass sie kaum in der Lage sind, die Innovationen zu leisten, welche aufgrund des Fortschreitens der Technologie eigentlich erforderlich wären.
2. Die Anwender wollen aufgrund der hohen, für Wartung aufzuwendenden Kosten keine Werkzeuge selbst entwickeln und pflegen.
3. Die Forschungsinstitute müssen zunächst von den Anwendern über Probleme informiert werden, in Forschungsarbeiten Lösungen erarbeiten und diese dann zusammen mit CAD-Firmen in Produkte umsetzen. Da dies häufig mit Hilfe von Dissertationen geschieht, für welche ca. 5 Jahre benötigt werden, vergehen vom Auftauchen des Problems bis zur breiten Markteinführung knapp 10 Jahre.

H. de Man's Schlussfolgerung ist, etwas überspitzt:

tomorrow's tools solve yesterday's problems.

Er folgert weiter, dass eine qualitativ hochwertige Ausbildung benötigt wird, damit Entwerfer in der Lage sind, die Lücken in verfügbaren Werkzeugen selbst zu überbrücken. Eine andere Möglichkeit wäre eine Forschung, die sich nicht nur an den Tagesproblemen orientiert und die langfristig vorab zukünftige Probleme erkennt. Dies ist natürlich nicht einfach.

Trotz der Unzulänglichkeiten wäre ein termingerechter Entwurf praktischer Elektroniksysteme ohne Synthese nicht mehr möglich. Wir werden uns im der Vorlesung mit den Grundlagen solcher Syntheseverfahren beschäftigen.

Kapitel 2

Mikroarchitektur-Synthese

Bei der **Mikroarchitektur-Synthese** wird die Mikroarchitektur eines Rechenssystems, d.h. dessen innerer Aufbau synthetisiert. Dieser Aufbau wird auf der Ebene der RT-Strukturen beschrieben. Es gibt eine Reihe weitgehend synonymer Begriffe für dieselbe Form der Synthese, nämlich *behavioral synthesis, high-level synthesis* oder auch *architectural synthesis*[1] Wird nur das Rechenwerk synthetisiert, so spricht man auch von *data path synthesis*. Auch die eingedeutschte Bezeichnung **Datenpfad-Synthese** ist üblich, wenngleich nach Meinung des Autors damit mehr Datenleitungen als Recheneinheiten assoziiert werden.

2.1 Synthese aus funktionalen Spezifikationen

Zunächst soll einer der Fälle betrachtet werden, in denen eine Spezifikation in funktionaler Form erfolgt, nämlich der systematische Entwurf von **systolischen Feldern** (engl. *systolic arrays*). Diese sind dadurch gekennzeichnet, dass **Daten synchron durch ein- oder zweidimensionale, regulär verbundene Felder von relativ einfachen Prozessoren „gepumpt“ und dabei auf diesen Daten parallel Berechnungen ausgeführt werden**. Der Name bezieht sich auf die „Systolen“, durch die der Körper Blut pumpt. Ein Bespiel für ein solches Feld ist *odd-even transposition sort*, ein Feld zum parallelen Sortieren. Abb. 2.1 zeigt ein solches Feld zum Sortieren von Mengen von maximal n=8 Elementen.

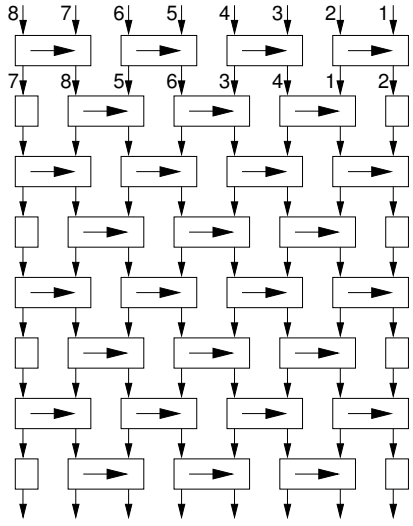


Abbildung 2.1: *Odd-even transposition sort,* n=8

Die einzelnen Prozessoren vergleichen dabei mit jedem Takt die Daten an ihren beiden Eingängen und leiten sie sortiert an die beiden Ausgänge weiter. Die Prozessoren sind nicht in der Lage, irgendwelche anderen Befehle auszuführen. Die Quadrate mit lediglich einem Eingang stellen Register dar. Man stelle sich nunmehr

[1] Siehe auch: [Mar93, Tei97].

einmal vor, die Daten mögen in umgekehrt sortierter Folge an den Eingängen anliegen (obere Zahlenreihe). Mit dem ersten Takt werden die Daten um jeweils eine Position vertauscht (untere Zahlenreihe). Mit dem zweiten Takt werden einige der Daten ein zweites Mal vertauscht. Nach insgesamt 8 Takten erscheint die sortierte Folge am Ausgang des Feldes. Dabei kann der Sortierer fließbandartig betrieben werden: während die Elemente der ersten Folge zum zweiten Mal verglichen werden erfolgt bereits der erste Vergleich für eine zweite Folge usw. Insgesamt kann nach dem Anlauf des Fließbandes mit jedem Takt eine Folge von 8 Elementen sortiert den Ausgängen entnommen werden. Solange das Feld ausreichend groß ist, ist also ein Sortieren in einer Zeitkomplexität von $O(1)$ möglich. Allerdings ist die Hardwarekomplexiät $O(n^2)$.

Es ist nicht immer ganz einfach, sich von der korrekten Funktion eines von Hand entworfenen systolischen Feldes zu überzeugen. Daher interessiert man sich für Techniken, mit denen systolische Felder automatisch aus Spezifikationen abgeleitet werden können. Gelänge es, diese Techniken als formal korrekt nachzuweisen, so wären die erzeugten Felder automatisch korrekt. Auch wenn es nicht gelingt, diese Techniken als korrekt nachzuweisen, erhält man mit automatischen Erzeugungstechniken eine relativ hohe Wahrscheinlichkeit eines korrekten Entwurfs.

Eine Möglichkeit der systematischen Konstruktion dieser Felder besteht in der Vorgabe von Rekursionsformeln, aus denen die Funktionswerte berechnet werden können. Auf diese Weise wird das System völlig losgelöst von Prozessorstrukturen und der Art und Weise der Ausführung beschrieben.

Beispiele

1. Fakultät

$$n! \quad = \quad \begin{cases} 1, \text{ falls } n = 1 \\ n * (n-1)!, \text{ falls } n \neq 1 \end{cases}$$

2. Hermit-Polynome

$$He_{n+1}(x) = x * He_n(x) - n * He_{n-1}(x)$$

Eine der ersten Methoden zur Synthese aus funktionalen Spezifikationen stammt von Quinton [Qui84]. Seine Methode erlaubt es, auf systematische Weise systolische Arrays aus Rekursionsformeln zu generieren. Vorausgesetzt wird ein System von Rekursionsgleichungen der Form

$$\begin{aligned} U_1(z) &= f(U_1(z-m_1), ..., U_p(z-m_p)) \\ U_2(z) &= U_2(z-m_2) \\ &.... \\ U_p(z) &= U_p(z-m_p) \text{ mit } z \in D \subseteq \mathbb{Z}^n; \end{aligned} \tag{2.1}$$

Die Vektoren m_j, mit $j \in \{1...p\}$, heißen Abhängigkeitsvektoren. Die Knoten aus dem Gebiet $D \subseteq \mathbb{Z}^n$ zusammen mit den Kanten $M = \{m_1, ..., m_p\}$ bilden den Abhängigkeitsgraphen $G = (D, M)$. $x \in D$ heißt abhängig von $y \in D$ falls $\exists i : x = y - m_i$.

Der Entwurf des systolischen Feldes geht nach Quinton in folgenden Schritten vor sich:

1. Aufstellen eines Systems von Rekursionsformeln

   Beispiel:

   Die in Physik und Elektrotechnik häufig benutzte Faltung[2] eines Vektors x mit einem Vektor w von Gewichten lässt sich schreiben als:

$$\forall i \geq 0 : y(i) = \sum_{k=0}^{K} w(k) * x(i-k)$$

   Die Berechnung von $y(i)$ ist in dieser Gleichung noch nicht als Rekursionsformel beschrieben. Eine Rekursionsformel kann man erzeugen, indem man alle Zwischenergebnisse $Y(i,k)$ der Summation bis k betrachtet:

[2] Wir beschränken uns hier auf die diskrete Form.

$$Y(i,k) = \sum_{k'=0}^{k} w(k') * x(i-k') \text{ mit} \tag{2.2}$$

$$Y(i,K) = y(i) \tag{2.3}$$

$Y(i,k)$ läßt sich in Form einer Rekursionsgleichung bestimmen über

$$\forall i \geq 0, \forall k \in \{0..K\} : Y(i,k) = Y(i,k-1) + w(k) * x(i-k) \text{ mit } Y(i,-1) = 0 \tag{2.4}$$

Damit ist Y über einem zweidimensionalen Gebiet definiert: $D \subseteq \mathbb{Z}^2$

Um auf die Form der Gleichung 2.1 zu kommen, ersetzen wir jetzt auch w und x durch Funktionen W und X über einem zweidimensionalen Gebiet. Für W propagieren wir die Werte lediglich zum jeweils nächsten ersten Index:

$$\forall i \geq 0, \forall k \in \{0..K\} : W(i,k) = W(i-1,k) \text{ mit der Anfangsbedingung:} \tag{2.5}$$

$$\forall k \in \{0..K\} : W(-1,k) = w(k) \tag{2.6}$$

Hier hätte man grundsätzlich auch eine andere Rekursionsformel nehmen können. Propagieren der Werte entlang der Achsen oder Diagonalen über kleine Indexdifferenzen führt zu regulärer Kommunikation über kurze Entfernungen.

Mit x können wir ähnlich verfahren. Wir wählen ein Propagieren der Werte entlang der Diagonalen:

$$\forall i \geq 0, \forall k \in \{0..K\} : X(i,k) = X(i-1,k-1) \text{ mit den Anfangsbedingungen} \tag{2.7}$$

$$\forall i \geq 0 : X(i-1,-1) = x(i) \text{ und} \tag{2.8}$$

$$\forall k \in \{0..K\} : X(-1,k-1) = 0 \tag{2.9}$$

Damit kann man jetzt Gleichung 2.4 umschreiben zu

$$Y(i,k) = Y(i,k-1) + W(i-1,k) * X(i-1,k-1) \tag{2.10}$$

mit den o.a. Anfangsbedingungen. Durch Vergleich mit der Gleichung 2.1 ergeben sich die Abhängigkeitsvektoren $m_1 = (0,1)$, $m_2 = (1,0)$ und $m_3 = (1,1)$ sowie der Abhängigkeitsgraph nach Abb. 2.2.

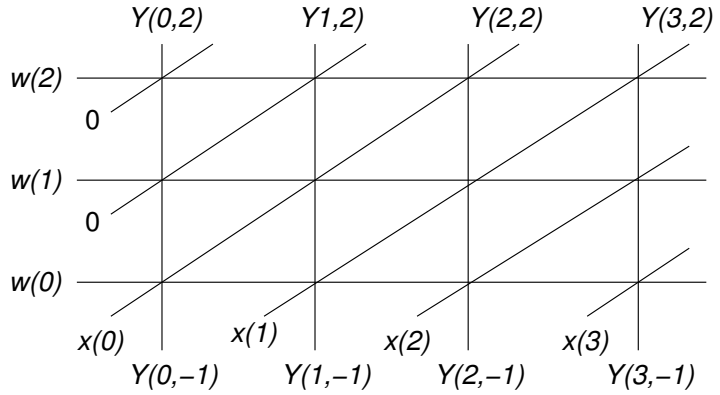


Abbildung 2.2: Abhängigkeitsgraph für die Faltung, $K = 2$

Dieser Graph drückt aus, in welcher Weise die Argumente von Berechnungen von den Ergebnissen anderer Berechnungen abhängen. Z.B. gehen in die Berechnung von $Y(2,2)$ die Ergebnisse der Berechnungen von $Y(1,2)$, $Y(1,1)$ und $Y(2,1)$ ein.

2. Suche nach einer *Timing* Funktion (Scheduling)

   Im nächsten Schritt werden allen Operationen Ausführungszeiten zugeordnet. Sei τ die Funktion, die allen Operationen eine Zeit zuordnet. Die folgenden Bedingungen sind an τ zu stellen:

   $\tau : \mathbb{Z}^n \rightarrow \mathbb{Z}$ mit $\forall s \in D : \tau(s) \geq 0$ und $\forall s, t \in D : s$ abhängig von $t \Rightarrow \tau(s) > \tau(t)$

   Falls eine Berechnung s von einer Berechnung t abhängt, so muss s nach t ausgeführt werden. Quinton gibt ein konstruktives Verfahren zur Erzeugung möglicher *Timing*-Funktionen an.

   Beispiel:

   Die Werte einer möglichen *Timing*-Funktion für die Faltung sind in der Abb. 2.3 in den Knoten eingetragen.

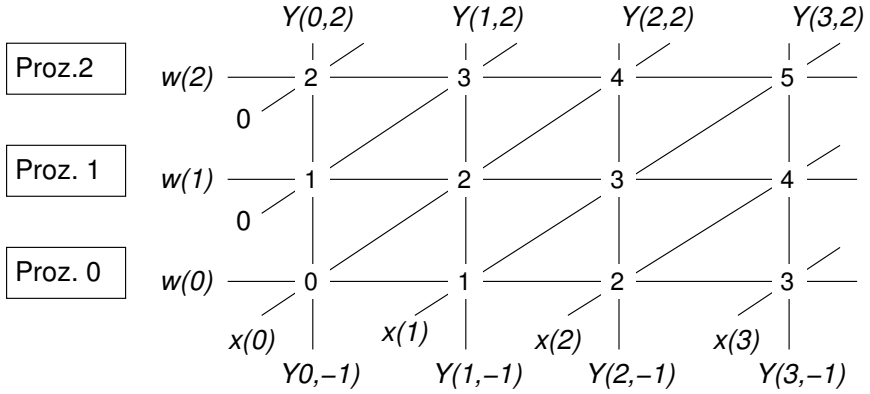


Abbildung 2.3: *Timing*-Funktion für die Faltung

   Z. B. wird die Berechnung von $Y(2,2)$ eine Zeiteinheit (einen Takt) nach den Berechnungen von $Y(1,2)$ und $Y(2,1)$ sowie zwei Zeiteinheiten (Takte) nach der Berechnung von $Y(1,1)$ ausgeführt.

3. Suche nach einer Prozessor-Zuordnung (*Assignment*)

   Im dritten Schritt wird jeder Operation ein Prozessor zugeordnet. Diese Zuordnung geschieht über die Funktion a, welche jedem Punkt des obigen Graphen einen Prozessor eines möglicherweise m-dimensionalen Feldes von Prozessoren zuordnet. Da jeder Prozessor zu einer bestimmten Zeit nur eine Berechnung ausführen kann, muss gelten:

   (2.11) $$a : \mathbb{Z}^n \rightarrow \mathbb{Z}^m \text{ mit } \forall_{s \neq t} s, t \in D : a(s) = a(t) \Rightarrow \tau(s) \neq \tau(t)$$

   d.h., je zwei Berechnungen, die demselben Prozessor zugeordnet werden, müssen zu unterschiedlichen Zeiten ausgeführt werden.

   Im Allgemeinen wird man Projektionen auf bestimmte Achsen wählen. Dabei muss so projiziert werden, dass Achsen, entlang derer Berechnungen unbeschränkt sind (wie im Beispiel die i-Achse), auf die Zeit abgebildet werden. Die andere Achse des Beispiels (die k-Achse) kann man auf ein eindimensionales Feld von Prozessoren abbilden. Man wählt also:

   $$\forall i, k : a(i, k) = (k, 0)$$

   Abb. 2.4 zeigt die entsprechende Projektion.

4. Bestimmung der Verbindungen und notwendiger Register

   Der Abb. 2.4 entnimmt man, dass der Strom der Werte entlang der Diagonalen (d.h. der x-Werte) jeweils eine Zeiteinheit zwischengespeichert werden muss, bevor er weiterverarbeitet werden kann. Diese Aufgabe übernimmt ein Register. Die y-Werte werden unverzögert dem nächsten Prozessor zugeführt. Daraus ergibt sich das systolische Prozessorfeld der Abb. 2.5. Bei dieser Realisierung verbleiben die Werte des Gewichtsvektors w fest in den Prozessoren.

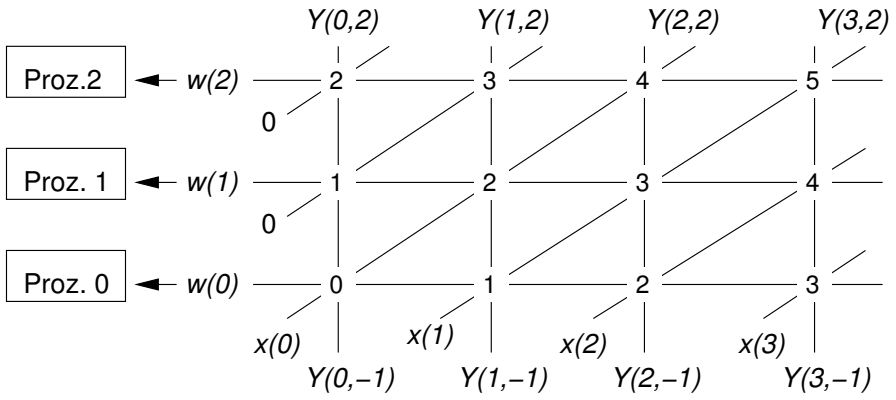


Abbildung 2.4: Projektion auf Prozessoren

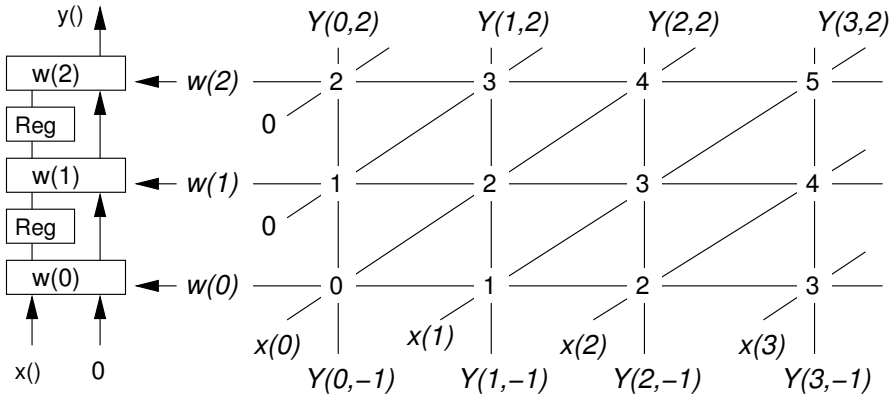


Abbildung 2.5: Eindimensionales Prozessorfeld für die Faltung

Systolische und befehlssystolische Felder wurden in den 80er Jahren intensiv untersucht. Über die hier angesprochene Art der Synthese gibt es zahlreiche Publikationen (z. B. [LM85], [Mol83], [Ull84]). Nach einer Zeit der relativen Ruhe sind sie durch die Verfügbarkeit von FPGAs wieder sehr interessant geworden.

Befehlsystolische Felder (engl. *instruction systolic arrays*) sind eine Erweiterung von systolischen Feldern, bei der Befehle durch das Prozessorfeld hindurch geschoben werden können [KLS+88]. Damit ist eine größere Flexibilität möglich.

2.2 Synthese aus imperativen Spezifikationen

2.2.1 Ziel der Synthese

Die Synthese auf der Basis **imperativer** Spezifikationen wurde zuerst an der Universität Kiel (Zimmermann et al.[Zim79, Mar79]) und der Carnegie Mellon Universität in Pittsburgh (Barbacci, Thomas, Parker[Par84]) entwickelt.

Im Folgenden beschränken wir uns zunächst auf das Rechenwerk (Steuerwerke werden später behandelt). Eine gründliche Analyse der Anforderungen ist v.a. bei berechnungsintensiven Spezifikationen erforderlich.

Berechnungen (ohne Sprünge) können mit Datenfluss-Graphen dargestellt werden.

Beispiel: Berechnung von Determinanten.

Die Determinante von

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

kann durch die Formel

$$d = a * (e * i - f * h) + b * (f * g - d * i) + c * (d * h - e * g)$$

beschrieben werden. In VHDL wird die Formel meist in Form von einfachen Variablen- oder Signalzuweisungen notiert:

```
h1 := e * i;
h2 := f * h;
h3 := f * g;
h4 := d * i;
....
```

Def.: Ein **Basisblock** ist eine (maximale) Codesequenz, die eine Verzweigung höchstens an ihrem Ende und mehrere Vorgänger (im Sinne eines Verschmelzens von Kontrollflüssen) höchstens an ihrem Anfang besitzt.

Bei Beschränkung auf Basisblöcke können die Anweisungen zu **Datenfluss-Graphen** (DFGs) zusammengefasst werden.

Der Datenfluss-Graph für die Determinantenberechnung sieht wie folgt aus:

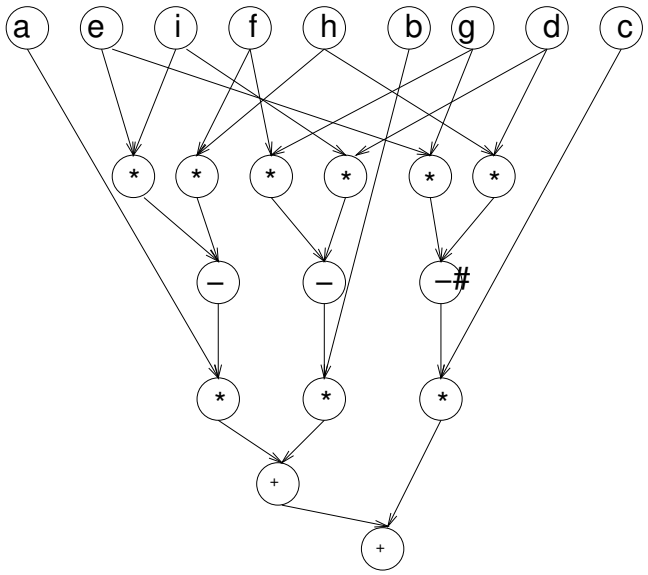


Abbildung 2.6: Datenfluss-Graph (-# = - mit vertauschten Eingängen)

Kommen in der Spezifikation auch Verzweigungen vor, so verwendet man einen separaten **Kontrollfluss-Graphen** zur Darstellung der Programm-Kontrolle. In Kombination mit Datenfluss-Graphen und Verweisen zwischen diesen kommt man zu **Kontroll/Datenfluss-Graphen** (CDFGs).

Beispiel:

Zur Spezifikation `IF bed. THEN <statements-1> ELSE <statements-1>` gehört der CDFG der Abb. 2.7.

Die Aufgabe der Mikroarchitektur-Synthese besteht darin, zu einem vorgegebenen CDFG eine Hardware-Implementierung zu erzeugen, die vorgegebene Randbedingungen (wie z.B. minimale Rechenleistung, maximaler Stromverbrauch, Ausnutzung von Bibliothekskomponenten usw.) einhält.

Man erwartet von der Mikroarchitektur-Synthese üblicherweise, dass drei Aufgaben gelöst werden:

- Das *Scheduling* (deutsch „**Ablaufplanung**“). Hierdurch wird für jede Operation festgelegt, **wann** sie ausgeführt wird.
- Die **Bereitsstellung** (engl. *allocation*) von Ressourcen (Addierern, usw.).
- Die **Zuordnung** (engl. *(resource) binding*). Hierdurch wird für jede Operation festgelegt, welche Hardware-Ressource (**wer**) die Ausführung übernimmt.

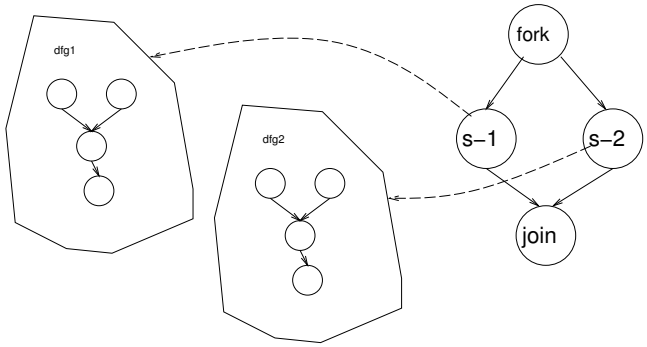


Abbildung 2.7: Kontroll/Datenflussgraph

Die Suche nach „optimalen" Architekturen erlaubt leider keine unabhängigen Verfahren zur Lösung der drei Aufgaben. Da aber bereits das Scheduling NP-hart ist, wird die Architektur-Synthese häufig dennoch in einzelne Phasen zerlegt.

Im Folgenden beschränken wir uns wieder auf DFGs. CDFGs werden häufig behandelt, indem sequenziell die einzelnen DFGs bearbeitet werden.

2.2.2 Scheduling

Die Bestimmung des Ausführungszeitpunktes von Operationen oder Tasks ist ein Standardproblem, das in verschiedenen Disziplinen (in der Logistik, der Fabrikationsplanung, in den Wirtschaftswissenschaften usw.) immer wieder vorkommt. Wir können grob zwischen dem Scheduling bei unabhängigen und bei abhängigen Aufgaben unterscheiden (siehe Abb. 2.8).

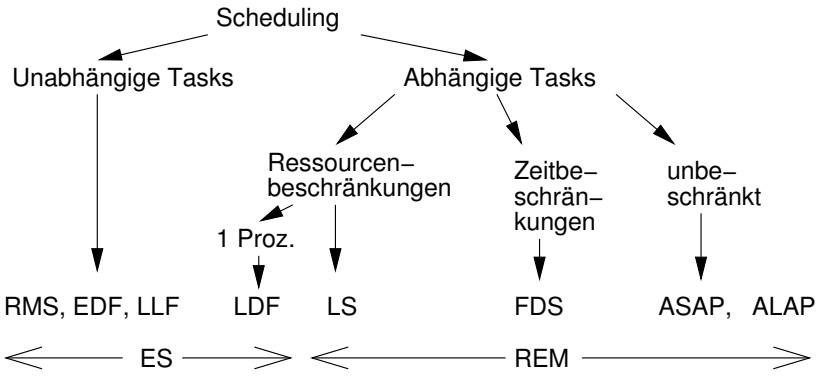


Abbildung 2.8: Eine Klassifikation von Scheduling-Verfahren

Zum Scheduling von unabhängigen Aufgaben sei hier auf die Vorlesung „Eingebettete Systeme" verwiesen [Mar07]. Scheduling-Verfahren für abhängige Operationen werden hier vorgestellt.

2.2.2.1 ALAP- und ASAP-Scheduling

Die einfachsten Scheduling-Verfahren für abhängige Tasks sind *as soon as possible* (ASAP) und *as late as possible* (ALAP) Scheduling. Beim ASAP-Verfahren werden alle Operationen so früh wie aufgrund der Abhängigkeiten möglich ausgeführt. Beim ALAP-Verfahren werden alle Operationen so spät wie aufgrund der Abhängigkeiten möglich eingeplant. Für das Beispiel der Determinantenberechnung führen diese Verfahren den Ergebnissen gemäß Abb. 2.9 und 2.10.

Das Beispiel beruht auf der Annahme gleicher Ausführungsgeschwindigkeiten aller Operationen. Üblicherweise wird eine synchrone Hardware vorausgesetzt. Die einzelnen „Zeiten" kennzeichnen Intervalle zwischen zwei Taktimpulsen. Man nennt diese Intervalle auch **Kontrollschritte**, (engl. *control steps*) da zu jedem Intervall ein Zustand des Steuerwerks gehört, das für eine geeignete Ansteuerung aller Bausteine des Rechenwerks sorgen muss (dies entspricht den Zuständen des Mikroprogramm-Steuerwerks, wie es für die

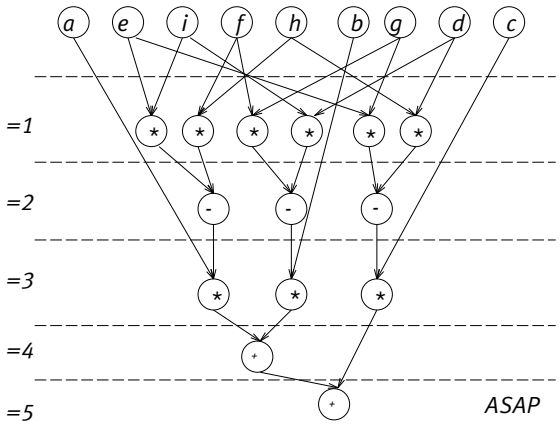


Abbildung 2.9: As-soon-as-possible Scheduling

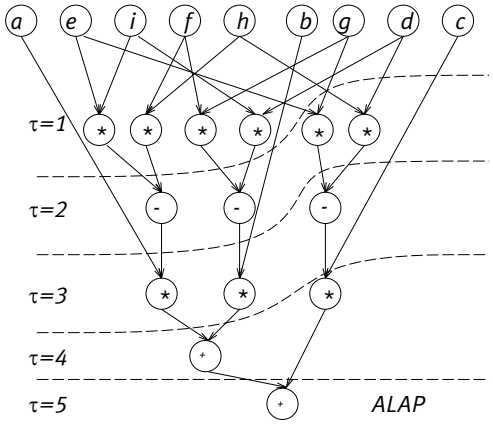


Abbildung 2.10: As-late-as-possible Scheduling

mikroprogrammierte Version der MIPS-Maschine in der Vorlesung „Rechnerstrukturen“ besprochen wurde).

2.2.2.2 Bezeichnungen

ASAP- und ALAP-Verfahren berücksichtigen keine Ressourcenbeschränkungen. Vor der Vorstellung von Verfahren, die Ressourcenbeschränkungen berücksichtigen, müssen wir zunächst einmal einige Bezeichnungen einführen.

Wir nehmen an, dass das Verhalten des zu entwerfenden Systems durch einen Datenflussgraphen gegeben ist. Die Knoten dieses Graphen bezeichnen Operationen wie Additionen und Multiplikationen. Genauer gesagt: sie bezeichnen Instanzen von Operationen (Operationstypen). Die Knoten dieses Graphen seien eindeutig durchnummeriert mit *integern* aus dem Bereich $J = \{1..j_{max}\}$. Als Variable für derartige Integer werden wir j benutzen.

Für die benutzten Operationstypen werden wir die Indexmenge G verwenden.

Sei *optype* eine Funktion, welche den Knoten des DFG die jeweiligen Operationstypen zuordnet (siehe Abb. 2.11).

Weiterhin nehmen wir an, dass die Struktur durch eine Netzliste mit den Baustein-Instanzen $k \in K = \{1..k_{max}\}$ beschrieben wird. Jede Baustein-Instanz wird aus einem zugehörigen Bibliothekstyp abgeleitet. Die Variablen $m \in M$ mögen die Typen von Bibliothekselementen bezeichnen. Die Funktion *type* bildet Baustein-Instanzen auf Baustein-Typen ab:

$$type : K \rightarrow M$$

Die Funktionalität eines Bausteins m wird durch die Funktion $\beta : M \rightarrow \wp(G)$ beschrieben:

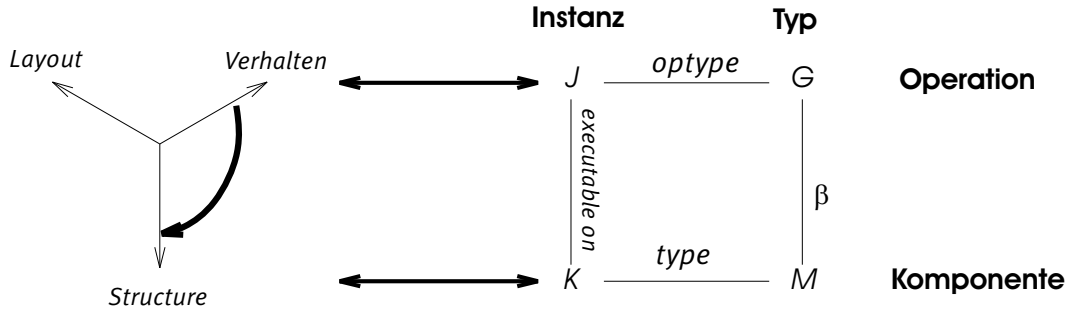


Abbildung 2.11: Notation in Synthese-Modellen

$\forall m \in M, g \in G : g \in \beta(m) \iff$ Bausteintyp m kann g ausführen.

Aus dieser Funktion leiten wir die folgende Relation ab:

Def.: $j \in J$ *executable on* $k \in K \iff optype(j) \in \beta(\wp(type(k)))$.

Die meisten Werkzeuge generieren nicht nur Struktur. Sie generieren auch eine Bindung zwischen Operationen und Kontrollschritten, in denen sie gestartet werden. Dies trifft auf alle uns bekannten Mikroarchitektur-Synthesewerkzeuge zu. Wir werden Indizes $i \in I$ benutzen, um Kontrollschritte zu bezeichnen. Modelle lassen in der Regel zu, sog. *multi-cycle operations* zu verwenden, welche mehrere Kontrollschritte benötigen. Der Einfachheit halber nehmen wir an, dass $\ell(j)$ für jede Operation die benötigte Anzahl an Kontrollschritten bezeichnet (d.h., vereinfachend nehmen wir an, dass alle Bausteine j gleich schnell ausführen können). Ausgereifte Synthesesysteme erlauben allerdings in der Regel eine Ausführungszeit, die von der Operation und vom Bausteintyp abhängt (d.h. $\ell = \ell(j, m)$).

2.2.2.3 *List Scheduling*

Eine Ressourcen-beschränkte Berechnung von Startzeitpunkten kann mit der listenorientierten Ablaufplanung (engl. *list scheduling*) durchgeführt werden. Wir setzen dabei voraus, dass für jeden Bausteintyp m ein Anzahl vorhandener Instanzen B_m festgelegt ist.

List scheduling beginnt zunächst einem topologischen Sortieren der Konten des Datenflussgraphens. Anschließend wird für jeden Knoten eine **Dringlichkeitsfunktion** (Priorität) p heuristisch bestimmt (s.u.).

Im Scheduling-Algorithmus sind einige Rechnungen wiederholt auszuführen:

- die Berechnung der in einem Kontrollschritt i auf einem Bausteinstyp m ausführungsbereiten Operationen, deren Vorgänger alle bekannt sind:

$$A_{i,m} = \{v_j \in V : type(v_j) \in \beta(m) \wedge \forall j' : (v_{j'}, v_j) \in E : i > \tau(v_{j'}) + \ell(j) - 1\}$$

- die Menge der Operationen des Typs m, die im Kontrollschritt i noch ausgeführt werden:

$$H_{i,m} = \{v_j \in V : type(v_j) \in \beta(m) \wedge \tau(v_j) + \ell(j) - 1 \geq i\} \tag{2.12}$$

- Bestimmung einer Menge S_i von zu startenden Operationen mit

$$|S_i| + |H_{i,m}| \leq B_m \tag{2.13}$$

Der eigentliche Algorithmus betrachtet dann in einem linearen Durchlauf durch die Menge notwendiger Kontrollschritte die jeweils einplanbaren Operationen [Tei97]:

```
List(G(V,E), β, B, p, ℓ){
  i:=0;
  repeat {
    for (m=1) to Anzahl Modultypen {
```

```
        Bestimme Kandidatenmenge A_{i,m};
        Bestimme Menge nicht beendeter Operationen G_{i,m} ;
        Wähle eine Menge S_i maximaler Priorität mit
          |S_i| + |H_{i,m}| ≤ B_m
        foreach (v_j ∈ S_i) : τ(v_j) := i; (*setze schedule fest*)
      }i := i + 1;
    }
    until (alle Knoten von V geplant);
    return (τ);
  }
```

Es gibt unterschiedliche Heuristiken zur Berechnung der Dringlichkeitsfunktionen (zitiert nach [MR92]):

- die Mobilitäts- oder Beweglichkeits-Heuristik: die Mobilität (engl. *mobility*) ist definiert als Differenz zwischen den aus ASAP- und ALAP-Knoten errechneten Zeitwerten. Knoten mit einer kleinen Beweglichkeit erhalten danach eine hohe Priorität.

  Die Mobilitätsheuristik will das Auswahlproblem des Knotens dadurch lösen, dass die einzuplanenden bereiten Operationen nach ihrer Mobilität sortiert werden. Da aufgrund von Ressourcen-Restriktionen nicht alle bereiten Operationen eingeplant werden können, versucht die Mobilitätsheuristik, wenigstens die Operationen auf dem kritischen Pfad einzuplanen, deren Mobilität aufgrund der Definition 0 ergibt. Bei knappen Ressourcen führt dieser Algorithmus allerdings nicht zu besonders guten Lösungen, weil dann die Mobilität als Auswahlkriterium keine gute Entscheidungshilfe ist.

- Lebenszeit-Heuristik:

  Die Lebenszeit-Heuristik hat das Ziel, eine gute Lösung für das Auswahlproblem zu finden, indem die Anzahl der benötigten Speicherzellen minimiert wird. Mit einer Lebenszeit-Analyse der Eingangswerte einer Operation kann festgestellt werden, ob durch das Einplanen der Operation die Speicherzellen frei werden, in denen die Eingangswerte dieser Operation gespeichert sind. Das Auswahlproblem wird also bevorzugt dadurch gelöst, dass die Operationen dann eingeplant werden, wenn die Lebenszeiten der Eingangswerte dieser Operation in dem vorliegenden Kontrollschritt enden.

- Vorausschau-Heuristik (*look ahead*):

  Während die bisher beschriebenen Heuristiken für die Lösung des Auswahlproblems im wesentlichen „greedy"-Strategien verfolgen und insbesondere keine Möglichkeit bieten, nachteilige Folgen einer einmal getroffenen Auswahl für spätere Kontrollschritte zu berücksichtigen, sollen die vorausschauenden Verfahren die Konsequenzen einer Auswahl für die Folgeschritte vorher bestimmen und berücksichtigen. Diese Verfahren sind zwar aufwendiger, führen aber in der Praxis zu wesentlich besseren Ergebnissen. Beispiele für solche Vorausschau-Heuristiken sind die Konflikt-Reduktions-Heuristik oder die Verzögerungs-Reduktions-Heuristik. Konflikte sind definiert als die Differenz zwischen der Anzahl der bereiten Operationen und der Anzahl der zur Verfügung stehenden Ressourcen. Die KonfliktReduktions-Heuristik versucht nun abzuschätzen, wieviele Konflikte sich in späteren Kontrollschritten unter der Annahme einer bestimmten Auswahl ergeben. Die Folgekonflikte können durch ein ASAP-Scheduling ohne Ressourcen-Beschränkung sehr effizient abgeschätzt werden.

Listenorientierte Ablaufplanung ist im Prinzip auf die Vorgabe von Ressourcen und die selbständige Wahl der Schedulelänge ausgelegt. Sie kann aber auch ohne Ressourcen-Beschränkungen durchgeführt werden.

2.2.2.4 *Force-directed scheduling*

Das nachfolgend beschriebene kräftebasierte Verfahren geht im Gegensatz zu listenorientierten Verfahren davon aus, dass die Schedulelänge fest ist und eine kostengünstige Ressourcenmenge gesucht wird. *Die zugrunde liegende Modellvorstellung, die auch bei Plazierungsverfahren verwendet wird, greift auf eine physikalische Analogie zurück: Zwischen Kontrollschritten werden, bezogen auf eine bestimmte Verarbeitungseinheit, „Kräfte" eingeführt, deren Stärke von der unterschiedlichen Auslastung der Operationseinheiten in den verschiedenen Kontrollschritten abhängt. Die auf diesem Kräfteverfahren basierende Ablaufplanung versucht nun, durch Verschieben von Operationen zwischen Kontrollschritten diese Kräfte zu minimieren. Dadurch wird eine möglichst gleichmäßige Auslastung aller Verarbeitungseinheiten erreicht und eine Minimierung der Verarbeitungseinheiten erzielt. Es wird davon ausgegangen, dass Operationen für jeden Zeitpunkt zwischen dem frühest- und dem spätestmöglichen mit gleicher Wahrscheinlichkeit eingeplant werden*

können. Aus dieser Annahme ergeben sich Wahrscheinlichkeiten für die Einplanung zu einem bestimmten Zeitpunkt. Das Kräfteverfahren geht nun im einzelnen wie folgt vor: Zunächst werden die Operationen eingeplant, deren frühestmöglicher Zeitpunkt mit dem spätestmöglichen zusammenfällt. Diese Operationen liegen also offensichtlich auf dem kritischen Pfad, während für alle anderen Operationen eine Verschiebung innerhalb dieser Zeitspanne möglich ist, ohne den gesamten Ablauf zu verzögern. In einem iterativen Verfahren zur Festlegung der Einplanung dieser freien Operationen wird versucht, die vorhandenen Komponenten möglichst gut auszunutzen, was einer Reduktion der Anzahl der benötigten Komponenten entspricht [MR92].

Die Einplanung der freien Operationen basiert auf der Berechnung einer „Wahrscheinlichkeitsverteilung“ der Operationen über die Kontrollschritte.

Deren Berechnung sei am Beispiel der Berechnung der Lösung einer Differentialgleichung (einem Benchmark der Synthese-*Community*) gezeigt. Lösungen der Differentialgleichung $y'' + 3zy' + 3y = 0$ werden danach durch das folgende kleine Programm bestimmt:

```
while (z < a) do
  begin
    zl:= z+dz
    ul:=u-(3 *z * u * dz)-(3 * y * dz);
    yl:=y+(u*dz);
    z:=zl;
    u:=ul;
    y:=yL;
  end;
```

Abb. 2.12 zeigt die dazugeörigen ASAP- und ALAP-Schedules.

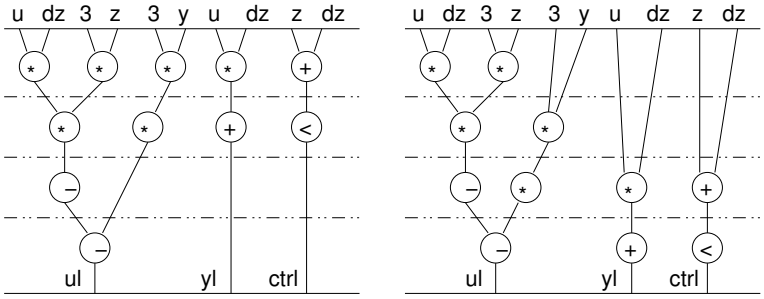


Abbildung 2.12: Schedules für das *diffeq*-Beispiel

Wir betrachten jetzt jeweils eine Menge von Operationstypen $H \subseteq \wp(G)$, für die dieselben Hardwarebausteine zur Realisierung in Frage kommen, also typischerweise $H = \{+, -\}$ oder $H = \{*\}$. Aus den Schedules bestimmen wir für die Operationen mit einem Typ aus H die **Zeitrahmen**, in denen eine Operation j ausgeführt werden kann. Wir bezeichnen die Menge der für j möglichen Kontrollschritte mit $R(j)$, ihre Anzahl mit $|R(j)|$.

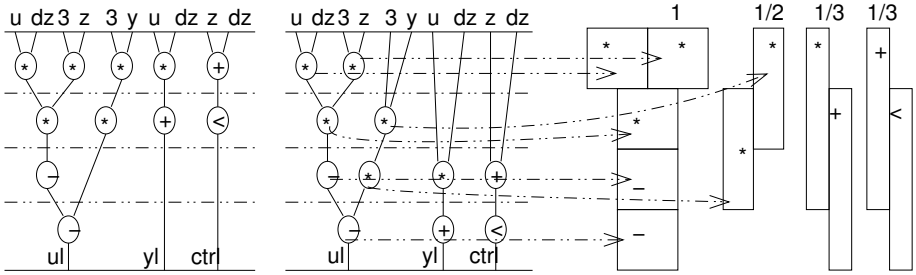


Abbildung 2.13: Zeitrahmen für das *diffeq*-Beispiel

Aus den Zeitrahmen bestimmen wir die „Wahrscheinlichkeit“ $P(j, i)$ für die Zuordnung von j zu i mittels der Gleichung [MR92]:

$$P(j,i) = \begin{cases} \frac{1}{|R(j)|} \text{ falls } i \in R(j) \\ 0 \text{ sonst} \end{cases}$$

Aus den „Wahrscheinlichkeiten“ für einzelne Operationen bestimmen wir „Verteilungen“ (engl. *distributions*) $D(i)$ mittels Betrachtung aller Operationen aus H:

$$D(i) = \sum_{j,type(j)\in H} P(j,i) \tag{2.14}$$

Abb. 2.14 zeigt die resultierenden „Wahrscheinlichkeiten“ und „Verteilungen“ für Multiplikationen im *diffeq*-Beispiel.

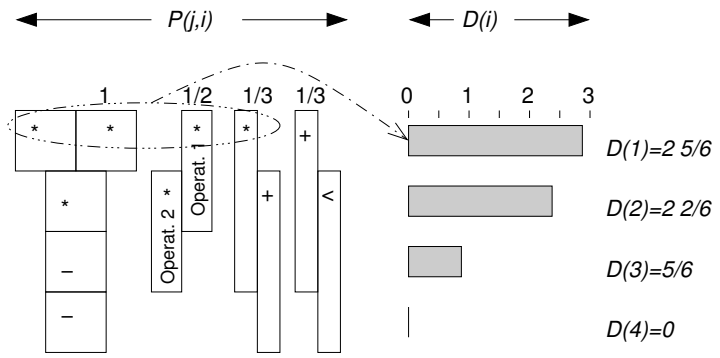


Abbildung 2.14: „Wahrscheinlichkeiten“ und „Verteilungen“ für Multiplikationen im *diffeq*-Beispiel

In Kontrollschritten, in denen die Verteilungsfunktion für einen gegebenen Operationstyp einen hohen Wert hat, sind demnach parallele Komponenten erforderlich. Um diese Parallelität zu reduzieren, müssen Operationen zu Kontrollschritten mit niedrigerem Wert der Verteilungsfunktion verschoben werden [MR92].

Zum diesem Zweck rechnen wir Kräfte aus. Wenn eine Operation j in einem festen Kontrollschritt i eingeplant wurde, gibt die Kraft $SF(j,i)$ die Änderung dieser Festlegung auf die Verteilung wieder:

$$SF(j,i) = \sum_{i' \in R(j)} D(i') \Delta P_i(j,i') \tag{2.15}$$

Dabei bezeichnet $\Delta P_i(j,i')$ die Änderung der Wahrscheinlichkeit im Kontrollschritt i', wenn die Operation j im Kontrollschritt i eingeplant wird. Die neue Wahrscheinlichkeit für die Einplanung von j in i ist 1. Vorher betrug diese $P(j,i)$, also ist die Differenz $1-P(j,i)$. Für Kontrollschritte $i' \neq i$ betrug die Wahrscheinlichkeit $P(j,i')$. Nach der Einplanung von j in i ist die Wahrscheinlichkeit 0, also ist die Differenz $-P(j,i')$. Daraus folgt:

$$\Delta P_i(j,i') = \begin{cases} 1 - P(j,i) \text{ falls } i = i' \\ -P(j,i') \text{ sonst} \end{cases}$$

Abb. 2.15 zeigt die Berechnung für unser Beispiel. Der positive Wert zeigt, dass es nicht ratsam ist, die Multiplikation im Schritt $i = 1$ einzuplanen.

Die Einplanung kann sich allerdings auch auf die vorhergehenden und nachfolgenden Kontrollschritte auswirken, was sich mit sog. Vorgänger bzw. Nachfolgerkräften ausdrücken läßt. Die Vorgänger- bzw. Nachfolgerkräfte VF bzw. NF sind definiert als:

$$VF(j,i) = \sum_{j' \in \text{ Vorgänger von } j} \sum_{i' \in I} D(i') \Delta P_{j,i}(j',i') \tag{2.16}$$

$$NF(j,i) = \sum_{j' \in \text{ Nachfolger von } j} \sum_{i' \in I} D(i') \Delta P_{j,i}(j',i') \tag{2.17}$$

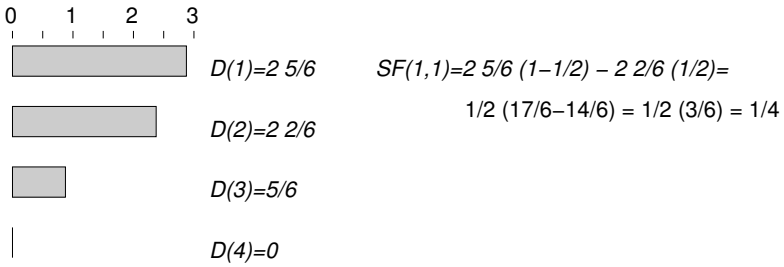


Abbildung 2.15: Berechnung von (direkten) Kräften für das *diffeq*-Beispiel

Dabei ist jetzt $\Delta P_{j,i}(j', i')$ eine Verallgemeinerung von $\Delta P_i(j', i')$. $\Delta P_{j,i}(j', i')$ ist die Änderung der Wahrscheinlichkeit der Einplanung von j' im Kontrollschritt i', wenn die Operation j im Kontrollschritt i eingeplant wird. Die Gesamtkraft $F(j, i)$ ergibt sich als Summe über die direkten, die Vorgänger- und die Nachfolgerkräfte:

$$F(j,i) = SF(j,i) + VF(j,i) + NF(j,i) \tag{2.18}$$

Betrachten wir beispielsweise die Einplanung der Multiplikation 1 (siehe Abb. 2.14) im Kontrollschritt 2. Die direkte Kraft bei einer solchen Entscheidung ergibt sich zu

$$\begin{aligned} SF(1,2) &= D(1) * \Delta P_2(1,1) + D(2) * \Delta P_2(1,2) \\ &= 2\frac{5}{6} * (-0,5) + 2\frac{2}{6} * 0.5 \\ &= -\frac{17}{12} + \frac{14}{12} \\ &= -\frac{3}{12} = -\frac{1}{4} \end{aligned} \tag{2.19}$$

Die zweite Multiplikation (siehe Abb. 2.14) wird aufgrund einer solchen Entscheidung dem dritten Kontrollschritt zugeordnet. Daher ergibt sich folgende Kraft für den Nachfolger:

$$\begin{aligned} NF(1,2) &= D(2) * \Delta P_{1,2}(2,2) + D(3) * \Delta P_{1,2}(2,3) \\ &= 2\frac{2}{6} * (-0,5) + \frac{5}{6} * 0.5 \\ &= -\frac{14}{12} + \frac{5}{12} \\ &= -\frac{9}{12} = -\frac{3}{4} \end{aligned} \tag{2.20}$$

Für die Gesamtkraft ergibt sich

$$F(1,2) = SF(1,2) + NF(1,2) = -\frac{1}{4} + (-\frac{3}{4}) = -1 \tag{2.21}$$

Unter Berücksichtigung von indirekten Kräften ergibt sich der niedrige Wert von -1. Dieser zeigt an, dass eine Einplanung der Multiplikation im Kontrollschritt 2 zu empfehlen ist.

Mit diesen Kräften lässt sich der vollständige Ablauf des kräftebasierten Verfahrens angeben [MR92]:

```
procedure kräfteverfahren;
  begin
    ASAP-Scheduling;
```

```
    ALAP-Scheduling;
    while nicht alle Operationen eingeplant do
      begin
        wähle Operation mit niedrigster Gesamtkraft aus;
        plane Operation in dem Kontrollschritt mit niedrigster Kraft ein;
        berechne Ausführungsintervalle neu;
        berechne D(i) neu;
      end
end
```

Das Verfahren muss für jede zu betrachtende Menge H wiederholt werden, also typischerweise für Multiplikationen und Addtionen getrennt durchgeführt werden. Überlappen sich die Funktionalitäten aber (können also etwa Multiplizierer auch addieren), so gerät das ursprüngliche Verfahren an seine Grenzen. Würde man Additionen und Multiplikationen zu einem H zusammenfassen, so könnte man die unterschiedlichen Kosten von Addierern und Multiplizierer nicht mehr berücksichtigen.

Force-directed scheduling ist ein sehr populäres Verfahren, von dem verschiedene Varianten in Benutzung sind, auch außerhalb des Mikroelektronikentwurfs.

Komplexere Scheduling-Verfahren müssen u.a. auch unterschiedliche Ausführungsgeschwindigkeiten der verschiedenen Operationen berücksichtigen und zu möglichst günstigen Allokationen führen.

2.2.3 Bereitstellung (*allocation*)

Dieser Schritt legt fest, welche wesentlichen Hardware-Ressourcen zur Lösung des Syntheseproblems zur Verfügung gestellt werden sollen. Als Ressourcen kommen u.a. arithmetisch/logische Einheiten, Speicherzellen und Verbindungen in Frage. Im Folgenden ist die Erklärung[3] allgemein an „Ressourcen“ ausgerichtet. Im konkreten Fall ist dafür eine der erwähnten Ressourcen-Klassen einzusetzen.

Während der Bereitstellung wird noch keine Zuordnung zwischen Ressourcen und ausgeführten Operationen vorgenommen. Die Methode der Kopplung der verschiedenen Synthesephasen hängt dabei vom Synthesesystem ab. Zum Teil sind die Phasen weitgehend unabhängig voneinander, zum Teil ist aber eine gewisse Vorausschau auf die späteren Synthesephasen vorhanden und zum Teil werden Synthesephasen zusammengefasst.

Es gibt mehrere Methoden, Ressourcen zur Verfügung zu stellen. Dabei unterscheiden sich die einzelnen sowohl hinsichtlich der benutzten Teilalgorithmen als auch in der Reihenfolge von deren Anwendung.

2.2.3.1 Vorgabe durch den Benutzer

Bei dieser Form erklärt der Benutzer bereits, welche Ressourcen einer bestimmten Klasse in der späteren Struktur vorhanden sein sein sollen. Dieses ist die einfachste Form, Ressourcen auszuwählen. Die Möglichkeit der Vorgabe von Ressourcen bereits in der Spezifikation sollte vorhanden sein. Hierzu werden Spezifikationssprachen benötigt, die eine Beschreibung einer Ressourcen-Bibliothek ermöglichen. Über diese Möglichkeit hinaus kann auch eine selbstständige Auswahl der Ressourcen durch das Synthesesystem erfolgen.

2.2.3.2 Direkte Compilation

Eine noch recht einfache Möglichkeit der Bereitstellung besteht darin, für jedes Vorkommen eines Operators eine eigene Ressource zu reservieren und keinerlei Mehrfachausnutzung vorzusehen. Wie in gewöhnlichen Compilern wird für jede Operation „Code“ erzeugt. Bei direkter Compilation werden Bereitstellung und Zuordnung generell in derselben Synthesphase durchgeführt.

Dieser Ansatz ist nur für kleine Verhaltsbeschreibungen geeignet, denn für eine Verhaltensbeschreibung mit (beispielsweise) 1000 Additionen würden ja 1000 Addierer erzeugt. Dennoch wird dieser Ansatz in einfachen Systemen benutzt. Vielfach wird einer Variablen der Verhaltensbeschreibung genau ein Register exklusiv zugeordnet. Dies ist dann ein Fall von direkter Compilation.

[3] Die Darstellung basiert auf dem Buch [Mar93] des Autors.

2.2.3.3 Mehrfachausnutzung von Ressourcen mit gleichen Fähigkeiten

Bei dieser Form wird davon ausgegangen,, dass jede Operation von Ressourcen aus genau einer Klasse ausgeführt werden kann. Durch Mehrfachausnutzen von Ressourcen sollen deren Anzahl und damit auch die Kosten minimiert werden. Ein Beispiel hierfür ist die Bereitstellung von Zellen für Variablen der Verhaltensbeschreibung. Derartige Variablen werden häufig nur in einem Teil der Kontrollschritt benötigt. Das Intervall von der ersten definierenden bis zur letzten lesenden Referenz heißt **Lebensdauer** der Variablen. Die Lebensdauer innerhalb von verzweifungsfreien Sequenzen von Kontrollschritten kann in Diagrammen wie der Abb. 2.16 dargestellt werden. Variablen, deren Lebensdauern sich nicht überlappen, können derselben Zelle zugeordnet werden. Die insgesamt benötigte Anzahl von Zellen ergibt sich danach aus der Maximalzahl gleichzeitig „lebendiger“ Variablen.

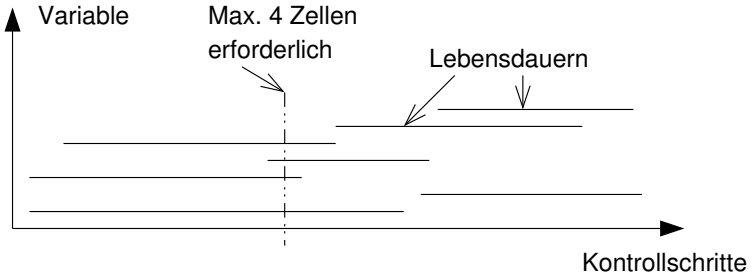


Abbildung 2.16: Maximale Anzahl gleichzeitig benötigter Ressourcen

Im Beispiel zu Beginn des Kapitels ergeben sich für das ASAP-Schedule 6 Multiplizierer, 3 Subtrahierer und 1 Addierer. Beim ALAP-Schedule werden nur 4 Multiplizierer benötigt. Die Abhängigkeit zwischen Scheduling und Allokation kommt damit klar zum Ausdruck.

2.2.3.4 Konstruktion von ALUs

Bei diesem Verfahren werden Operationen aufgrund der Relation „gleichzeitiges bzw. nicht gleichzeitiges Vorkommen im Programm“ zu Ressourcen gruppiert.

Beispiel: Gegegeben sei Kontrollschritt CS1 mit dem zu berechnenden Ausdruck $(a =_1 b) \wedge_2 (c \geq_3 d)$ und Kontrollschritt CS2 mit dem Ausdruck $(c =_4) \vee_5 (g \leq_6 h)$. Dabei wurden die Operationen eindeutig durchnummeriert. Die Abb. 2.17 zeigt dann links die Relation „gleichzeitiges Vorkommen“. Dieser Graph wird auch als **Konfliktgraph** bezeichnet. Rechts enthält die Abbildung die Relation „kein gleichzeitiges Vorkommen“. Dieser Graph wird auch als **Kompatibilitätsgraph** bezeichnet.

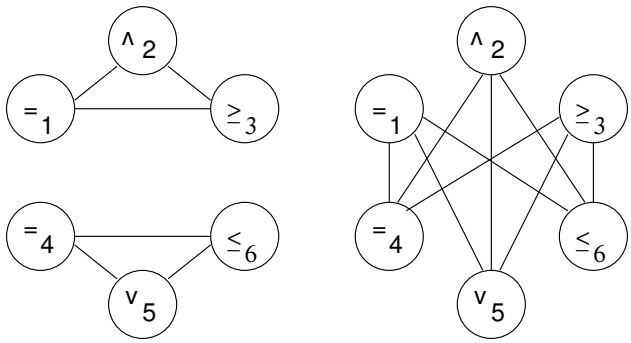


Abbildung 2.17: Relationen „gleichzeitiges“ bzw. „nicht gleichzeitiges Vorkommen“

Die minimale Anzahl von ALUs kann durch **Färben** des Konfliktgraphen bestimmt werden. Bei **Färbeproblem für Graphen** ist generell die Aufgabe zu lösen, zu einem gegebenen Graphen eine minimale Zahl von Farben so zu bestimmen, dass je zwei miteinander über Kanten verbundene Knoten eine unterschiedliche Farbe haben. In der Anwendung auf die Konstruktion von ALUs entspricht jede notwendige Farbe einer ALU. Allerdings ist das Färbeproblem im Allgemeinen NP-hart. Abb. 2.18 zeigt links eine mögliche Lösung des Färbeproblems.

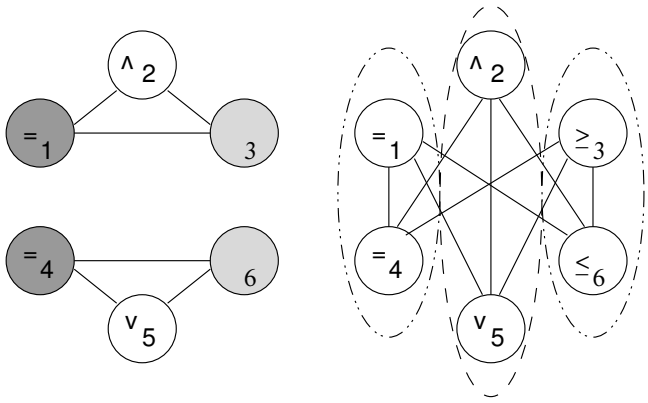


Abbildung 2.18: Mögliche Lösungen des Färbe- bzw. des Cliquenproblems

Alternativ kann man die minimale Anzahl von ALUs auch über den Kompatibilitätsgraphen bestimmen. Dieser Graph ist mit einer minimalen Zahl von Mengen von Knoten zu überdecken, sodass in jeder Menge alle Knoten miteinander vollständig über Kanten verbunden sind, also alle Knoten miteinander kompatibel sind. Knotenmengen, die vollständig miteinander über Kanten verbunden sind, heißen **Cliquen**. Das Problem, einen Graphen mit einer minimalen Zahl von Cliquen zu überdecken, heißt **Cliquenproblem**. Das Cliquenproblem ist im Allgemeinen ebenfalls NP-hart. Abb. 2.18 zeigt rechts eine mögliche Lösung des Cliquenproblems (Cliquen sind durch Ellipsen umschlossen).

Die Komplexität reduziert sich deutlich, wenn die Konfliktgraphen bzw. Kompatibilitätsgraphen aus Basisblöcken (d.h. einer linear geordneten Menge von Kontrollschritten) entstehen. Dann muss nämlich nur die Anzahl der maximal gleichzeitig vorkommenden Operationen gesucht werden. Diese Anzahl entspricht der Zahl der benötigten Farben bzw. Cliquen. Abb. 2.19 zeigt das Vorgehen für den Fall unseres Beispiels unter der Annahme, dass alle Operationen in einem Kontrollschritt abgeschlossen werden.

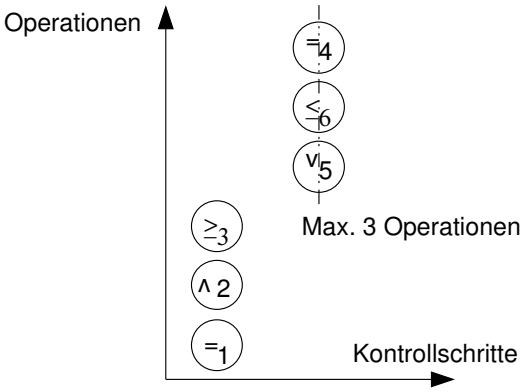


Abbildung 2.19: Maximale Anzahl gleichzeitig benötigter ALUs

Wenn Operationen mehrere Kontrollschritte andauern können, so entsteht wieder eine Situation analog zu Abb. 2.16.

Standardverfahren zur Erzeugung einer minimalen Anzahl von ALUs minimieren nicht unbedingt deren Kosten. Bessere Verfahren sollten die Operationen derart zu Gruppen zusammenfassen, dass die Kosten minimiert werden. Dies ist grundsätzlich mit dem nachfolgend beschriebenen Verfahren möglich, dass allerdings die Konstruktion einer Bibliothek mit sinnvollen ALU-Bausteinen voraussetzt.

2.2.3.5 Mehrfachausnutzung von Ressourcen mit unterschiedlichen Fähigkeiten

Sofern die Ressourcen unterschiedliche Fähigkeiten haben, reicht das Zählen der maximal benötigten Ressourcen nicht mehr aus. Vielmehr werden die Ressourcen nunmehr unterschiedlich komplex und unterschiedlich teuer sein. Statt der Anzahl der Ressourcen ist nunmehr die Summe ihrer Kosten zu minimieren. Diese Kosten können reale Kosten sein (wie z.B. die benötigte Chipfläche) oder auch fiktive Kosten, die

lediglich ausdrücken, dass gewisse Bausteine aufwendiger sind als andere. Dabei müssen notwendige Bedingungen für die jeweils benötigte Mindestzahl von Ressourcen eingehalten werden. Wie bestimmt man nun diese Anzahl?

$f_{i,g}$ sei die Häufigkeit von Operation g in Kontrollschritt i. Seien β_m die vom Bausteintyp m bereit gestellten Operationstypen. Sei B_m die **gesuchte** Anzahl der Bausteine vom Typ m.

Wir wollen uns die Verhältnisse anhand eines Beispiels klar machen. Wir benutzen hierfür die Bereitstellung von ALUs. Da jetzt die Kosten minimiert werden sollen, müssen für alle möglichen ALUs die Kosten bekannt sein (das bedeutet nicht, dass sie bis in alle möglichen Einzelheiten hinein entworfen sein müssen).

Für unsere Zwecke möge eine Bibliothek mit drei Bausteintypen gegeben sein, mit $\beta(1) = \{+\}$, $\beta(2) = \{-\}$, $\beta(3) = \{+, -\}$ und $f_{1,+} = 1$, $f_{1,-} = 1$ sowie $f_{2,+} = 2$. Offensichtlich müssen für die Bausteinanzahlen folgende Ungleichungen gelten:

Für den ersten Kontrollschritt:

$$\begin{aligned} B_1 + \quad + B_3 &\geq 1 \text{ wegen der Zahl von Additionen} \\ B_2 + B_3 &\geq 1 \text{ wegen der Zahl der Subtraktionen} \\ B_1 + B_2 + B_3 &\geq 2 \text{ wegen der Gesamtzahl von Operationen} \end{aligned}$$

Für den zweiten Konrtrollschritt gilt:

$$B_1 + \quad + B_3 \geq 2 \text{ wegen der Zahl der Addtionen}$$

Dabei haben wir in jedem Kontrollsschritt alle Kombinationen von Operationen betrachtet. Wir können die Gleichungen über Kontrollschritte hinweg zusammenfassen, indem wir nur noch für jede vorkommende Kombination die minimal erforderliche Anzahl von Bausteinen bestimmen. Im Beispiel führt das zu

$$\begin{aligned} B_1 + \quad + B_3 &\geq 2 \text{ (Zusammenfassung der Kontrollschritte 1 und 2)} \\ B_2 + B_3 &\geq 1 \\ B_1 + B_2 + B_3 &\geq 2 \end{aligned}$$

Allgemein gilt: **Für jede Kombination der jeweils vorkommenden Operationen muss die Gesamtzahl der Ressourcen, die mindestens eine der Operationen ausführen können, mindestens gleich der Gesamtzahl der Operationen sein.** Diese Aussage wird nun nachfolgend präzisiert.

Seien

$$\begin{aligned} M &: \text{die Bausteinbibliothek} \\ c_m &: \text{die Kosten des Typs } m \\ B_m &: \text{die Anzahl der Exemplare des Typs } m \\ \beta_m &: \text{die Operationen des Typs } m \\ f_{i,g} &: \text{die Häufigkeit des Operationstyps } g \text{ in Schritt } i \\ F_i &: \text{die Menge der in Schritt } i \text{ benutzten Operationen} \\ F_i^* &: \text{die Potenzmenge von } F_i \text{ ohne die leere Menge} \end{aligned}$$

Dann muss gelten:

$$\forall i, \forall h \in F_i^* : \sum_{\substack{m \in M \\ h \cap \beta_m \neq \emptyset}} B_m \geq \sum_{g \in h} f_{i,g} \tag{2.22}$$

$$\text{mit } \sum_{m \in M} B_m * c_m \rightarrow \text{ Minimum} \tag{2.23}$$

Das heißt: Für alle Kontrollschritte i und alle Elemente h der Potenzmenge der in i benutzten Operationstypen ist die Summe der Exemplare jener Ressourcetypen, die mindestens eine benötigte Operation ausführen können, mindestens gleich der Gesamtzahl der ausgeführten Operationen.

Die Aussage ist zunächst einmal für einen einzelnen Kontrollschritt hinreichend. Mit einfachen Rechnungen kommt man zu Ungleichungen, die für alle Kontrollschritte ausreichen [Mar90]:

$$\forall h \in F^* : \sum_{m \in M} a_{h,m} * B_m \geq S_h \tag{2.24}$$

$$\text{mit } F^* = \cup_i F_i^* \tag{2.25}$$

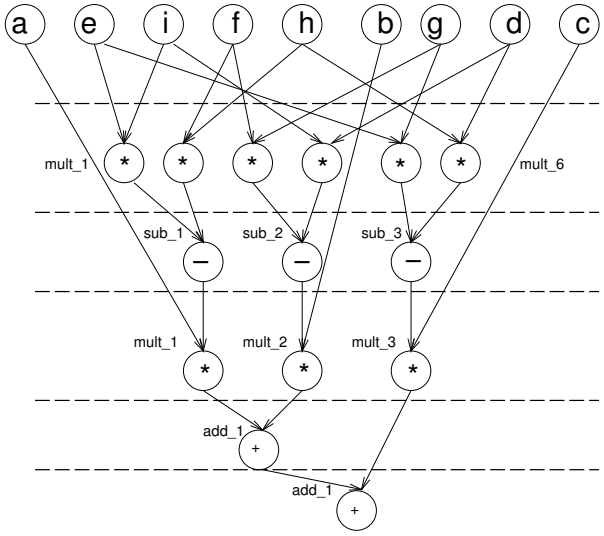
$$a_{h,m} = \begin{cases} 1, \text{ falls } h \cap \beta_m \neq \emptyset \\ 0, \text{ sonst} \end{cases} \tag{2.26}$$

$$\forall h \in F^* : S_h = max_i \left(\sum_{g \in h} f_{i,g} \right) \tag{2.27}$$

Die Formel 2.24 zeigt, dass die Randbedingungen in Form von in B_m linearen Ungleichungen mit binären Koeffizienten darstellbar sind. Damit ist die optimale Bereitstellung auf die Ganzzahlige Programmierung (engl. *integer programming*, IP) zurückgeführt.

2.2.4 Zuordnung

Die einfachste Zuordnung besteht in dem simplen Durchzählen innerhalb der Kontrollschritte.



Die Zuordnung bestimmt implizit die Verbindungsstruktur innerhalb des Rechenwerks. So muss im Beispiel `mult_1` verbunden sein mit den "Registern" `e` und `i` sowie mit `subh_1` und `a` (siehe Abb. 2.20).

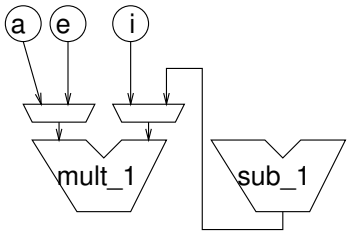


Abbildung 2.20: Verbindungen als Folge einer Zuordnung von Bausteinen zu Operationen

2.2.4.1 Der *left edge* Algorithmus

Eine Zuordnung bei Ressourcen mit denselben Fähigkeiten (z.B. von Variablen bzw. Speicherzellen) gelingt mit einer einfachen Erweiterung des Verfahrens zur Allokation: Wir durchlaufen die Kontrollschritte und ordnen jeweils zu Beginn einer Lebensdauer eine Ressource zu und geben sie am Ende der Lebendauer wieder frei (siehe Abb. 2.21). Dies ist die Grundidee des sog. *left edge* Algorithmus.

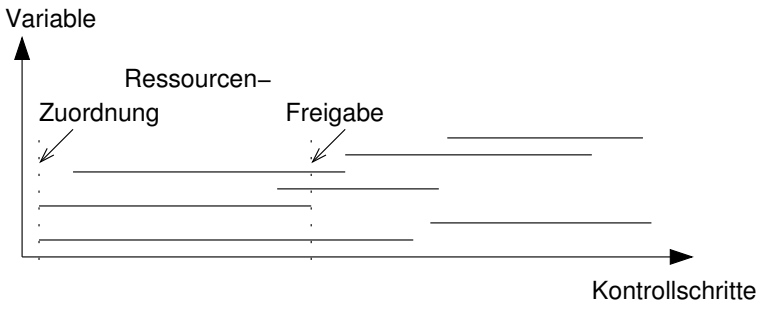


Abbildung 2.21: Graphische Darstellung des *left edge* Algorithmus

Zur genauen Darstellung des *left edge* Algorithmus anhand der Zuordnung von Variablen zu Speicherzellen benötigen wir die nachfolgenden Größen:

$first[i]$	:	die Menge der im Kontrollschritt i zuerst benutzten Variablen
$last[i]$	:	die Menge der im Kontrollschritt i zuletzt benutzten Variablen
$imax$	:	die Anzahl der Kontrollschritte
$kmax$	:	die Anzahl verfügbarer Speicherzellen
$assign[j]$	:	die einer Variablen j zugeordnete Speicherzelle

Der *left edge* Algorithmus hat die folgende Form:

```
Free := {1..kmax};
for i=1 to imax do (*Schleife über Kontrollschritte*)
  begin
    for each j ∈ first[i] do
      begin
        k ist kleinstes Element aus Free
        assign[j] := k; (* Gesuchte Zuordnung*)
        Free := Free − {k};
      end
    for each j ∈ last[i] do
      Free := Free ∪ {assign[j]}
  end;
```

Dabei haben wir angenommen, dass eine Variable in $first[i]$ zu Beginn des Kontrollschritts i bereits zugeordnet sein muss und eine Variable in $last[i]$ erst nach Abschluss von i frei wird (wir also die Lebensdauerbereiche als abgeschlossene Intervalle betrachten). **Der Algorithmus liefert einer Zuordnung mit einer minimalen Anzahl von Speicherzellen, da nie mehr Zellen belegt werden, als wirklich gleichzeitig benötigt werden.** Der Algorithmus wird als **linear** bezeichnet (was eine Implementierung der inneren Schelife in konstanter Zeit voraussetzt). Der Algorithmus löst das unter „Zuordnung“ beschriebene Färbeproblem für Konfliktgraphen. Damit entsteht die Frage, wie sich ein Graph in linearer Zeit färben lassen kann, wenn das allgemeine Färbeproblem NP-hart ist. Die Antwort liegt in der speziellen Struktur des Graphen: beim Färben von Konfliktgraphen die aus linear geordneten Kontrollschritten entstehen, handelt es sich nämlich um sog. **Intervallgraphen**.

Def.: Ein Graph heißt **Intervallgraph**, falls es eine umkehrbar eindeutige Abbildung seiner Knoten auf eine linear geordnete Menge von Intervallen existiert, so dass zwei Knoten genau dann mit einer Kante verbunden sind, wenn sich die Intervalle schneiden.

Unsere Konfliktgraphen sind per Konstruktion Intervallgraphen, wenn wir uns auf Konflikte innerhalb von Basisblöcken beschränken.

Intervallgraphen sind ein Spezialfall von sog. **perfekten Graphen** [Gol80]. Sie lassen sich in linearer Zeit färben, z.B. mit dem *left edge* Algorithmus.

Der *left edge* Algorithmus ist ein Standardverfahren für die sog. lokale **Registerzuordnung** in Compilern, d.h. die Registerzuordnung bei lokalen Optimierungen innerhalb von Basisblöcken. Mit der Verwendung in der Synthese wurde v.a. Kurdahi bekannt [KP87].

2.2.4.2 Das Quadratische Zuordnungsproblem

Der *left edge* Algorithmus nimmt keine Rücksicht auf evtl. unterschiedliche Funktionalitäten von Ressourcen oder auf Verbindungsstrukturen. Das Problem der Berücksichtigung von Verbindungskosten kann am Besten anhand eines Beispiels eingeführt werden. Man betrachte wie in Abb. 2.22 einen Ausschnitt aus einem Datenflussbaum, der auf eine Hardwarestruktur abgebildet werden soll.

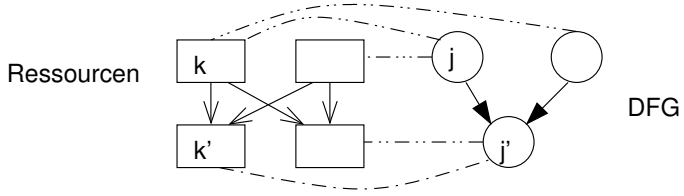


Abbildung 2.22: Wirkung von Ressourcezuordnungen auf die Verbindungsstruktur

Sei $a_{j,j'} = 1$ wenn die Operation j ein Argument für die Operation j' liefert und 0 sonst. Sei K die Menge der Ressourcen und J die Menge der Operationen und seien $c_{k,k'}$ die Kosten einer Verbindung von $k \in K$ zu $k' \in K$. Sei $x_{j,k} = 1$ wenn der Operation j die Ressource k zugeordnet wird und 0 sonst. Sei $\beta_{j,k} = 1$ falls Ressource k die Operation j ausführen kann und 0 sonst. Bei einer Zuordnung von Operationen zu Hardwarekomponenten entstehen dann Verbindungskosten in Höhe von

$$C = \sum_{\substack{j,j' \in J \\ k,k' \in K}} a_{j,j'} c_{k,k'} x_{j,k} x_{j',k'} \tag{2.28}$$

Dabei sind Randbedingungen einzuhalten:

- Innerhalb eines Kontrollschrittes darf eine Ressource nur einer Operation zugeordnet werden (Annahme: $\ell = 1$):

$$\forall k \in K : \sum_{j \in J} x_{j,k} \leq 1 \tag{2.29}$$

- Jeder Operation muss ein geeigneter Baustein zugeordnet werden:

$$\forall j \in J : \sum_{k \in K} \beta_{j,k} x_{j,k} = 1 \tag{2.30}$$

Diese Gleichungen bilden einen speziellen Fall des sog. **Quadratischen Zuordnungsproblems** (engl. *quadratic assignment problem*, QAP), benannt nach dem quadratischen Auftreten der Variablen x in der Kostenfunktion. Im allgemeinen Fall dürfen die Randbedingungen beliebige lineare Funktionen in den Variablen $x_{j,k}$ enthalten. Das Quadratische Zuordnungsproblem ist ebenfalls NP-hart. Übliche Programmpakete zur Lösung von Ganzzahligen Programmierungsproblemen stellen in der Regel auch Verfahren zur Lösung von Quadratischen Zuordnungsproblemen bereit.

Das QAP tritt in vielen Anwendungen auf, z.B. in der Logistik.

Komplexere Zuordnungsverfahren müssen versuchen, die entstehenden Verbindungen zu optimieren und dabei auch komplexere Komponenten (ALUs usw.) berücksichtigen. Teilweise gelingt dies in dem nachfolgend beschriebenen, integrierten Verfahren.

2.2.5 Integration von Scheduling, Allokation und Assignment in OSCAR

Wir betrachten dazu das Synthesewerkzeug OSCAR (*optimum scheduling, allocation, and resource binding*).

Die Syntheseaufgabe kann als das Problem modelliert werden, jeden Knoten j an einen Baustein k und einen Kontrollschritt i zu binden, in dem sie gestartet wird.

$$x_{i,j,k} = \begin{cases} 1, & \text{Wenn Operation } j \text{ auf der Baustein-Instanz } k \text{ im Kontrollschritt } i \text{ gestartet wird} \\ 0, & \text{sonst} \end{cases}$$

Als Hilfvariablen werden die folgenden b_k benötigt:

$$b_k = \begin{cases} 1, & \text{falls die Bausteinintanz } k \text{ benötigt wird} \\ 0, & \text{sonst} \end{cases} \tag{2.31}$$

Für diese Variablen gelten eine Reihe von Beschränkungen (*constraints*):

1. *Operation assigment constraints*: Diese stellen sicher, dass jede Operation an exakt einen Kontrollschritt und an einen Hardware-Baustein gebunden werden:

$$\forall j \in J : \sum_{i \in R(j)} \sum_{\substack{k \in K \\ optype(j) \in \beta(type(k))}} x_{i,j,k} = 1$$

2. *Resource assignment constraints*: Diese stellen sicher, dass jede Komponente k höchstens alle $\ell(k)$ Kontrollschritte eine neue Operation startet.

$$\forall k \in K : \forall i \in I : \sum_{j \in J} \sum_{i'=i}^{i+l(k)-1} x_{i',j,k} \leq b_k$$

3. *Precedence constraints*: Diese stellen sicher, dass Berechnungen, welche die Ergebnisse anderer Berechnungen benötigen, erst nach diesen gestartet werden. Um diese Constraints aufzustellen, gehen wir von einer Operation j_2 aus, welche von einer Operation j_1 datenabhängig ist. Seien $R(j_1)$ und $R(j_2)$ die zugehörigen möglichen Kontrollschritt-Bereiche.

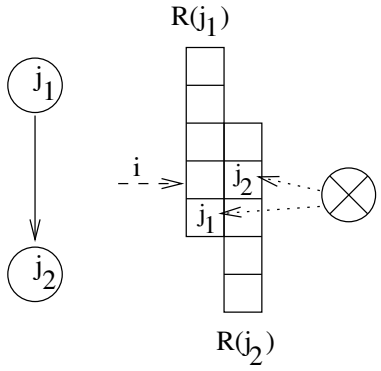


Abbildung 2.23: Verbotene Bindung bei datenabhängigen Operationen

Sofern der Schnitt zwischen beiden leer ist, ist keine besondere Bedingung aufzustellen, denn die Operationen können dann schon wegen der *operation assignment constraints* nicht in der falschen Reihenfolge ausgeführt werden. Sofern der Schnitt nicht leer ist, muss für jeden Kontrollschritt i in diesem Schnitt sichergestellt werden, dass nicht j_2 vor oder in i und j_1 nach oder in i ausgeführt werden. Dies leistet die folgende Menge von Ungleichungen:

$$\forall i \in R(j_1) \cap R(j_2) : \sum_{k} \sum_{\substack{i' \leq i \\ i' \in R(j_2)}} x_{i',j_2,k} + \sum_{k} \sum_{\substack{i' \geq i \\ i' \in R(j_1)}} x_{i',j_1,k} \leq 1$$

Diese Ungleichungen müssen für alle Paare (j_1, j_2) datenabhängiger Operationen gelten. Die angegebene Form der Ungleichungen basiert auf der Annahme, dass alle Operationen nur einen Kontrollschritt benötigen und dass zwei abhängige Operationen nie innerhalb desselben Kontrollschritts ausgeführt werden können. Das allgemeine Modell von OSCAR kommt ohne diese Annahmen aus.

Das Ziel des Entwurfs kann als Minimierung von Kosten definiert werden. Seien c_m die Kosten des Bausteintyps m und c_{k_1,k_2} die Kosten für eine Verbindung vom Baustein k_1 zum Baustein k_2. Ferner sei definiert:

$$w_{k_1,k_2} = \begin{cases} 1, & \text{falls der Baustein } k_1 \text{ mit Baustein } k_2 \text{ verbunden ist} \\ 0, & \text{sonst} \end{cases} \tag{2.32}$$

Dann ergeben sich die Gesamtkosten als:

$$C = \sum_{m \in M} (c_m * \sum_{\substack{k \in K \\ type(k)=m}} b_k) + \sum_{k_1,k_2} c_{k_1,k_2} * w_{k_1,k_2}$$

Ein einfaches Beispiel:

Gegeben seien die beiden VHDL-Zuweisungen

```
s := (u + v) * (w + x);
t:= y * z
```

Für die erste Zuweisung seien drei, für die zweite vier Kontrollschritte zulässig (siehe Abb. 2.24). Wir benutzen dabei im Folgenden die Indizes a, b, c und d zur Indizierung der Operationen, wie in der Abbildung angedeutet.

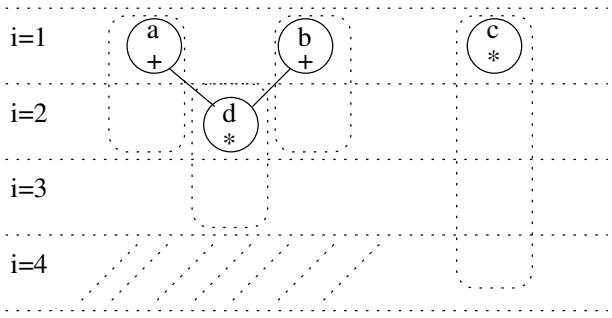


Abbildung 2.24: Einfacher Datenfluss-Graph

Für diese Operationen ergeben sich die folgenden ASAP/ALAP-Kontrollschritt-Bereiche:

$$R(a) = [1, 2]; R(b) = [1, 2]; R(c) = [1, 4]; R(d) = [2, 3]$$

Wir nehmen an, dass unsere Bausteinbibliothek drei Bausteine enhält (siehe Tabelle 2.1).

Typ	Operationen
1	+
2	*
3	+, *

Tabelle 2.1: Einfache Bausteinbibliothek

Alle Bausteine sollen ihre Ergebnisse in einem Kontrollschritt berechnen ($\forall k : \ell(k) = 1$). Weiter sei $\forall m : c_m = 1$.

Wir nehmen an, dass wir maximal zwei Exemplare pro Bausteintyp benötigen und ordnen diesen gemäß Abb. 2.25 Indizes zu.

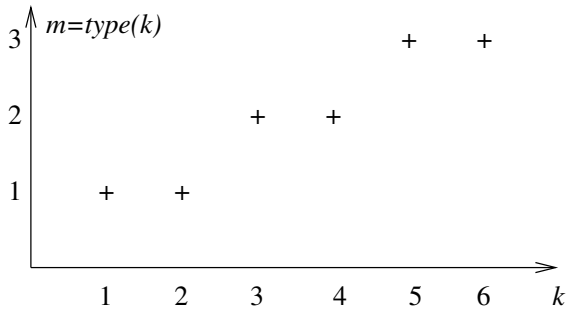


Abbildung 2.25: Indexreservierung

Da diese Programme zur Ganzzahligen Programmierung die Zielfunktionen häufig maximieren, arbeiten wir mit einer mit -1 multiplizierten Zielfunktion.

Damit ergibt sich die folgende Zielfunktion:

$C = -b_1 - b_2 - b_3 - b_4 - b_5 - b_6 - w_{1,3} - w_{1,4} - w_{1,5} - w_{1,6} - w_{2,3} - w_{2,4} - w_{2,5} - w_{2,6} - w_{5,3} - w_{5,4} - w_{5,6} - w_{6,4} - w_{6,5}$

Die $w_{i,j}$ spielen nur eine Rolle, falls innerhalb desselben Kontrollschritts Daten von einer Arithmetikeinheit zur nächsten weitergeleitet werden (sog. *chaining*). Wir nehmen an, dass dies nicht der Fall ist und dass diese Einheiten daher nicht direkt miteinander verbunden werden müssen. Wir setzen daher $\forall j, i : w_{i,j} = 0$.

Die *resource assignment constraints* nehmen in diesem Fall die folgenden Formen an:

Für $k = 1, 2$ (Addierer):

$i = 1$	x_{1a1}	$+x_{1b1}$	$\leq b_1$		x_{1a2}	$+x_{1b2}$	$\leq b_2$	
$i = 2$	x_{2a1}	$+x_{2b1}$	$\leq b_1$		x_{2a2}	$+x_{2b2}$	$\leq b_2$	

Für $k = 3, 4$ (Multiplizierer):

$i = 1$	x_{1c3}		$\leq b_3$		x_{1c4}		$\leq b_4$	
$i = 2$	x_{2c3}	$+x_{2d3}$	$\leq b_3$		x_{2c4}	$+x_{2d4}$	$\leq b_4$	
$i = 3$	x_{3c3}	$+x_{3d3}$	$\leq b_3$		x_{3c4}	$+x_{3d4}$	$\leq b_4$	
$i = 4$	$+x_{4c3}$		$\leq b_3$		x_{4c4}		$\leq b_4$	

Für $k = 5, 6$ (Kombinationsbausteine):

$i = 1$	x_{1a5}	$+x_{1b5}$	$+x_{1c5}$		$\leq b_5$	$+x_{1a6}$	$+x_{1b6}$	$+x_{1c6}$		$\leq b_6$
$i = 2$	x_{2a5}	$+x_{2b5}$	$+x_{2c5}$	$+x_{2d5}$	$\leq b_5$	$+x_{2a6}$	$+x_{2b6}$	$+x_{2c6}$	$+x_{2d5}$	$\leq b_6$
$i = 3$	x_{3c5}	$+x_{3d5}$			$\leq b_5$	$+x_{3c6}$	$+x_{3d6}$			$\leq b_6$
$i = 4$	x_{4c5}				$\leq b_5$	$+x_{4c6}$				$\leq b_6$

Die *operation assignment constraints* lauten für das Beispiel wie folgt:

1. $x_{1a1} + x_{1a2} + x_{1a5} + x_{1a6} + x_{2a1} + x_{2a2} + x_{2a5} + x_{2a6} = 1$ (die Operation a muss in Schritt 1 oder in Schritt 2 ausgeführt werden und an einen zur Addition fähigen Baustein gebunden werden);
2. wie 1., jedoch für die Operation b;
3. $x_{2d3} + x_{2d4} + x_{2d5} + x_{2d6} + x_{3d3} + x_{3d4} + x_{3d5} + x_{3d6} = 1$ (die Operation d muss in Schritt 2 oder in Schritt 3 ausgeführt werden und an einen zur Multiplikation fähigen Baustein gebunden werden);
4. $x_{1c3} + x_{1c4} + x_{1c5} + x_{1c6} + x_{2c3} + x_{2c4} + x_{2c5} + x_{2c6} + x_{3c3} + x_{3c4} + x_{3c5} + x_{3c6} + x_{4c3} + x_{4d4} + x_{4c5} + x_{4c6} = 1$ (die Operation c kann in den Schritten 1 bis 4 ausgeführt werden und muss an einen zur Multiplikation fähigen Baustein gebunden werden).

precedence constraints sind für die Paare (b,d) und (a,d) erforderlich. Der Schnitt der Kontrollschritte besteht aus dem Schritt 2. Also sind folgende *constraints* erforderlich:

1. $x_{2d3} + x_{2d4} + x_{2d5} + x_{2d6} + x_{2a1} + x_{2a2} + x_{2a5} + x_{2a6} \leq 1$ (Schritt 2 darf nicht a und d enthalten);
2. $x_{2d3} + x_{2d4} + x_{2d5} + x_{2d6} + x_{2b1} + x_{2b2} + x_{2b5} + x_{2b6} \leq 1$ (Schritt 2 darf nicht b und d enthalten).

Gleichungen für Verbindungen werden in dem einfachen Beispiel nicht betrachtet.

Werte der Entscheidungsvariablen, welche die Kostenfunktion minimieren, können mittels üblicher *integer programming*-Pakete bestimmt werden. Im Falle von OSCAR wird *lp_solve* der Universität Eindhoven eingesetzt. Aufgrund der gewählten Kosten wird sicherlich ein Kombinationsbaustein ausgewählt werden, die Operation c wird im Schritt 4 und die Operation d wird in Schritt 3 ausgeführt werden. Die Operationen a und b werden den Schritten 1 und 2 zugeordnet.

Die Verwendung von *integer programming*-Modellen ist für die beschriebene Anwendung nicht unumstritten. Da *integer programming* NP-vollständig ist, wird vielfach eingewandt, dass diese Methode nur auf kleine Beispiele anwendbar ist. Als Alternative werden dann häufig heuristische Verfahren eingesetzt, die stark von der imperativen Programmierung geprägt sind (z.B. werden die Operationen sukzessiv gebunden). Im Zusammenhang mit OSCAR konnte aber gezeigt werden, dass *integer programming* durch geschickte Reduktion auf die wesentlichen Entscheidungen durchaus für praktisch relevante Problemgrößen erträgliche Laufzeiten liefern kann. *integer programming* besitzt gegenüber den Heuristiken die folgenden Vorteile:

1. Es kann Optimimalität bezüglich des gewählten Kostenmodells garantiert werden,
2. es liegt ein präzises mathematisches Modell der zu lösenden Aufgabe vor,
3. es kann formal nachgewiesen werden, welche Eigenschaften eine synthetisierte Implementierung besitzt,
4. zusätzliche Bedingungungen können verhältnismäßig leicht integriert werden.

OSCAR besitzt insgesamt die folgenden wesentlichen Merkmale:

1. OSCAR ist eng an vorhandene kommerzielle Software gekoppelt und bildet damit eine Ergänzung zu den auf dem Markt erhältlichen Werkzeugen,
2. OSCAR nutzt komplexe Baustein-Bibliotheken (einschließlich *multiply-accumulate*-Bausteinen) aus,
3. OSCAR unterstützt Bausteine mit unterschiedlichen Geschwindigkeiten (z.B. schnelle und langsame Multiplizierer),
4. OSCAR entscheidet selbst, ob *chaining* (Ausführen abhängiger Berechnungen in einem Kontrollschritt) sinnvoll ist,
5. OSCAR erlaubt die Angabe von Zeitbedingungen,
6. OSCAR nutzt algebraische Regeln (wie z.B. das Distributiv-Gesetz) aus.

Bei der Anwendung algebraischer Regeln entsteht das Problem, eine Kombination von Regelanwendungen zu finden, die zu kostengünstiger Hardware führen. Das Problem, die beste Folge von *rewrite*-Regeln zu bestimmen, ist ebenfalls NP-vollständig. OSCAR setzt als Optimierungsverfahren an dieser Stellen einen genetischen Algorithmus ein. Ein Ergebnis dieser Optimierung zeigt die Tabelle 2.2.

Kontrollschritte	ohne Optimierung	Ersetzung von * durch Schiebe-operationen	Ausnutzung der Assozia-tivität	Ausnutzung der Assoziativität und von Schiebe-operationen
9	-	-	-	4a
10	-	-	-	4a
11	-	4a	-	3a
12	-	3a	3m, 3a	3a
13	-	3a	2m, 3a	3a
14	2m, 3a	3a	1m, 3a	2a
15	1m, 3a	2a	1m, 2a	2a

Tabelle 2.2: Anzahl von Addierern (a) und Multiplizierern (m) für das Elliptical Wave Filter-Beispiel

Für das Benchmark-Beispiel *elliptical wave filter* benötigt die "optimale" Lösung 14 Kontrollschritte, wenn keine algebraischen Regeln ausgenutzt werden. Die Ausnutzung der algebraischen Regeln erlaubt die Reduktion des Hardware-Aufwandes bei gleicher Kontrollschritt-Zahl und die Reduktion der Kontrollschritt-Zahl.

2.2.6 Ersetzung von höheren Sprachelementen

Bevor die eigentliche Synthese ablaufen kann, werden Synthesesysteme in der Regel einige höhere Sprachelemente (wie z.B. Prozeduraufrufe) durch andere Sprachelemente (wie z.B. Zuweisungen) ersetzt. Diese Programmtransformationen sind in einigen Fällen fest in die Synthesesoftware eingebaut, in anderen Fällen durch Regelsysteme oder Vorcompiler flexibel gehalten. Ebenso ist es vom Synthesesystem abhängig, welche Sprachelemente ersetzt werden und welche von den späteren Phasen im Synthesesystem verarbeitet werden. Für die Ersetzung von Sprachelementen bieten sich in der Regel mehrere Alternativen. Eine repräsentative Menge von Möglichkeiten sei nachfolgend angegeben:

- Prozedur- und Funktionsaufrufe

  Einige Synthesesysteme (wie z.B. XST) betrachten Funktionen als einfache Makros, die per *inlining* einzukopieren sind. Rekursive Funktionen wird man in diesem Fall ausschließen. Dieser Ansatz funktioniert, ohne dass über die Zielhardware irgendwelche Annahmen gemacht werden müssen.

  Man kann allerdings auch annehmen, dass eine prozessorartige Hardware erzeugt werden soll. Dann können Prozeduraufrufe wie in üblichen Compilern durch eine Menge von Zuweisungen ersetzt werden. In diesem Fall gibt es keine Einschränkungen hinsichtlich der Funktionen und von deren Parametern.

- Bedingungsabfragen

  Auch Bedingungsabfragen können unterschiedlich realisiert werden. Wird die Spezifikation in ein Maschinenprogramm übersetzt, so können die üblichen bedingten Sprünge zur Realisierung von **if**-*statements* eingesetzt werden. Alternativ ist auch die bedingte Ausführung (engl. *predicated execution*) möglich. Dabei wird die Ausführung von Befehlen unterdrückt, wenn eine Bedingung nicht erfüllt ist. Wird eine Spezifikation direkt in Hardware übersetzt, so können Hardwarekomponenten über Boolesche Bedingungen gesteuert werden.

- Schleifen

  Wird die Spezifikation in ein Maschinenprogramm übersetzt, so können die üblichen Maschinenbefehle zur Realisierung von **for**- oder **while**-*statements* eingesetzt werden. Hierbei gibt es dann keine Einschränkungen hinsichtlich der Grenzen oder Schrittweiten der Schleifen. Wenn keine Übersetzung in Maschinenprogramme erfolgt, so werden **for**-Schleifen häufig zur Compilezeit expandiert (wie z.B. in XST). In diesem Fall müssen die Schleifengrenzen und -schrittweiten dann häufig konstant sein.

  Fortgeschrittene Synthesesysteme enthalten spezielle Schedulingverfahren für Schleifen, um so den laufzeitmäßig wichtigsten Teil von Programmen besonders schnell ausführen zu können.

- Variable

  Einige einfache Synthesesysteme identifizieren Variablen mit Speichern, d.h. erzeugen für skalare Variable Register und für Arrays adressierbare Speicher. Anspruchsvollere Systeme bieten eine optimierte Zuordnung von Variablen zu Speichern und erlauben dabei ggf. ein Falten von Variablen, d.h. ein Zuordnen von mehreren Variablen zu derselben Speicherzelle, wenn sich die Lebendauerbereiche der Variablen nicht überlappen.

- Speicherallokation

  Die meisten Synthessysteme erlauben keine explizite Allokation von Speicher mit Funktionen wie *malloc*. Wenn das Synthesesystem allerdings eine „normale“ Umgebung zur Programmlaufzeit erzeugt, kann selbstverständlich auch eine dynamische Speicherallokation realisiert werden.

Um den Aufwand für die Realisierung aller Hochsprachelemente im Synthesesystem zu vermeiden, versucht man neuerdings die Synthese auf der Basis des von einem Compiler erzeugten Codes zu starten. Alle vom Compiler unterstützten Sprachelemente sind damit automatisch realisert. Allerdings ist der Aufwand für eine Erzeugung z.B. von Datenflussgraphen aus dem Assemblercode beträchtlich.

Kapitel 3

Controller-Synthese

3.1 Aufteilung in Rechenwerk und Controller

Damit die Bausteine eines Rechenwerks in jedem Kontrollschritt die beabsichtigte Funktion ausführen, müssen sie entsprechend kontrolliert werden. Zu diesem Zweck wird ein **Controller** oder **Steuerwerk**) (vgl. Abb. 3.1) benötigt.

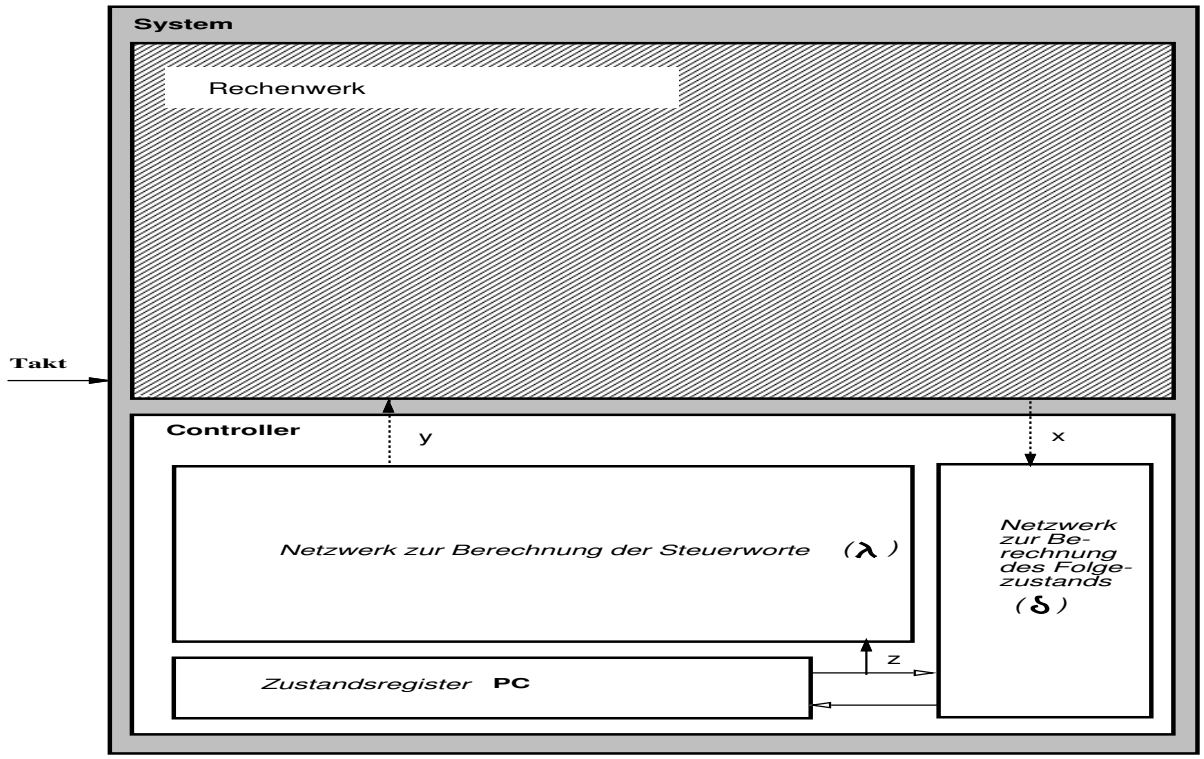


Abbildung 3.1: Aufteilung des Entwurfs in Rechenwerk und Controller

Wesentliche Teile des Controllers sind ein Zustandsregister, ein Lese-Speicher, welcher jedem Zustand ein Steuerwort zuordnet sowie eine Logik, welche anhand des Zustandes und der Werte der Ausgangssignale des Rechenwerks den jeweils nächsten Zustand bestimmt. Die Belegung des Steuerwortes, d.h. der Ausgangssignale des Steuerwerks, sind vom aktuellen Zustand abhängig. Dieser Controller ist also ein Moore-Automat. Üblicherweise ist man bemüht, nicht sowohl das Rechenwerk wie auch das Steuerwerk als Mealy-Automaten auszulegen, da dies zu asynchronen Rückkopplungen führen könnte.

3.2 Zustandsreduktion

Eine der Vereinfachungstechniken für endliche Automaten ist die Reduktion der Zahl der Zustände. Hierbei fasst man Zustände mit gleichen Ausgaben und Folgezuständen (sog. **äquivalente Zustände**) zusammen. Beispiel: in Abb. 3.2 sind die Zustände `a` und `b` äquivalent.

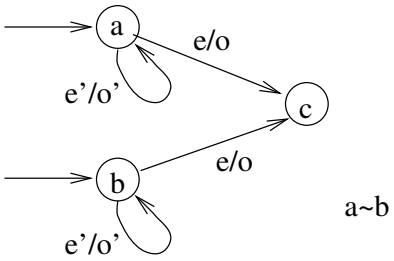


Abbildung 3.2: Äquivalente Zustände

Durch Zusammenfassen äquivalenter Zustände kommt man zu *reduzierten Automaten*. Die Theorie der Automatenreduktion ist lange bekannt (siehe z.B. [Koh87]). Diese Technik hat meist nur eine untergeordnete Bedeutung, da es in praktisch vorkommenden Spezifikationen meist wenig äquivalente Zustände gibt.

Von der Forderung nach äquivalenten Folgezuständen kann man absehen, wenn man Controller mit einem **Keller** (engl. "stack") versieht, auf den man den Inhalt des Zustandsregisters retten und von dem man den Zustand zurückschreiben kann. Man kommt so zu (parameterlosen) Unterprogrammen und braucht mehrfach vorkommende Zustandssequenzen nur einmal in den Schaltnetzen für δ und λ zu berücksichtigen. Beispiel: in Abb. 3.3 können die Zustände `a` und `b` bzw. `a'` und `b'` zu einem Unterprogramm zusammengefaßt werden, obwohl die Folgezustände `d` bzw. `d'` ein unterschiedliches Ein/Ausgabeverhalten haben.

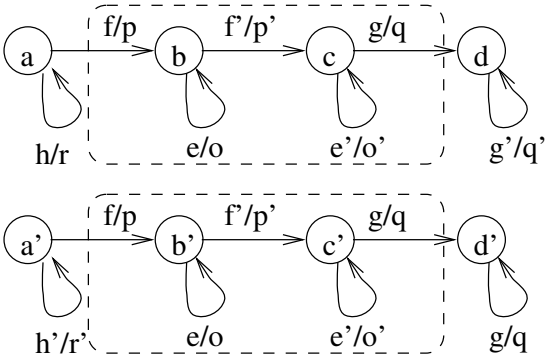


Abbildung 3.3: Unterprogramme in Automaten

Mit Unterprogrammkellern ausgerüstete Controller sind in der Mikroprogrammierung vielfach benutzt worden, z.B. in den Controller-Bausteinen der Fa. AMD [AMD83]. In der automatischen Controllersynthese werden sie erst seit kurzem berücksichtigt.

3.3 Zustandskodierung

Der Aufwand für die Implementierung der Funktionen λ und δ hängt in starkem Maße von der Kodierung dieser Zustände ab.

Die Grundideen der Controllersynthese im **ASYL-System** [dPD89] sind verhältnismäßig einfach zu erklären. ASYL ist regelbasiert. Zunächst werden Regeln für die Kodierung von Zuständen aufgestellt. Im wesentlichen gibt es drei Klassen von Regeln:

1. *"Join"-Regeln:*

   Falls unter einer bestimmten Eingabe Übergänge von mehreren Ausgangszuständen zu demselben Folgezustand führen, so sollten die Ausgangszustände möglichst viele Bits gemeinsam haben (betrachte

Abb. 3.4).

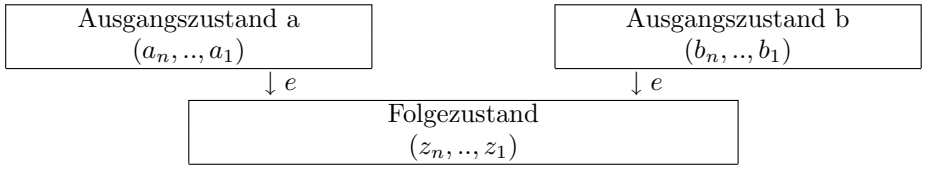


Abbildung 3.4: Kodierung bei "Joins"

$(a_n, .., a_1)$, $(b_n, .., b_1)$ und $(z_n, .., z_1)$ stehen dabei für die Kodierung der drei Zustände. e sei ein Wert der Eingabesignale, für den von beiden Ausgangszuständen in denselben Folgezustand verzweigt wird. Sei i ein Bit des Folgezustands mit $z_i = 1$. Um dieses Bit bei Eingabe von e zu setzen, wird ein Ausdruck $e(a_n...a_2a_1 \vee b_n..b_2b_1)$ benötigt. Sofern die Kodierung der Zustände a und b nur in einem Bit, sagen wir im Bit 1, verschieden ist, vereinfacht sich der Term zu $e(a_n...a_2)$ und wir haben einen Produktterm eingespart. Dies ist bei der in Abb. 3.5 angegebenen Kodierung der Fall.

Abbildung 3.5: Kodierung bei "Joins"

Die Forderung, dass sich Zustandskodierungen in nur einem Bit unterscheiden (man sagt, der **Hamming-Abstand** sei 1), lässt sich durch Graphen ausdrücken. Dieser enthält für jeden Zustand einen Knoten. Je zwei Knoten sind genau dann mit einer Kante verbunden, wenn die Kodierung einen Hamming-Abstand von 1 haben soll (siehe Abb. 3.9).

Im Fall vom m Ausgangszuständen lassen sich maximal $m - 1$ Produktterme einsparen.

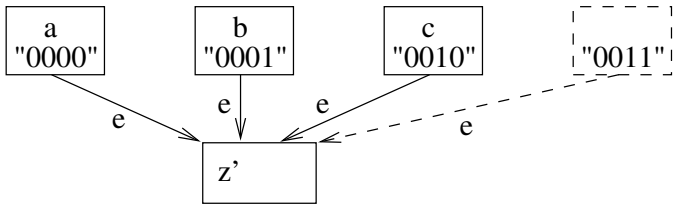


Abbildung 3.6: m=3 Ausgangszustände

Sofern m keine 2-er Potenz ist, müssen $2^k - m$ (mit $k = \lceil log_2(m) \rceil$) nicht ausgenutzte Codes von einer weiteren Verwendung ausgeschlossen werden, da von diesen Codes bei Eingabe von e ebenfalls in den Folgezustand z' verzweigt werden würde (siehe Abb. 3.6).

2. *"Fork"-Regeln*

   Analog zu 1. ist die Berechnung des Folgezustands bei Verzweigungen am einfachsten, wenn sich die Folgezustände in möglichst wenigen Bits unterscheiden (siehe Abb. 3.7).

3. **Regeln für die Ausgabe**

   Sofern zwei Zustände dieselbe Ausgabe erzeugen, ist die Berechnung der Ausgabe am einfachsten, wenn sich die Kodierung der Zustände in möglichst wenigen Bits unterscheidet.

   Beispiel: In Abb. 3.8 erzeugen die Zustände `a` und `b` dieselbe Ausgabe. Die Anzahl der Terme zur Berechnung von `Stop` wird reduziert, wenn sich die Kodierungen von `a` und `b` in möglichst wenigen Bits unterscheiden.

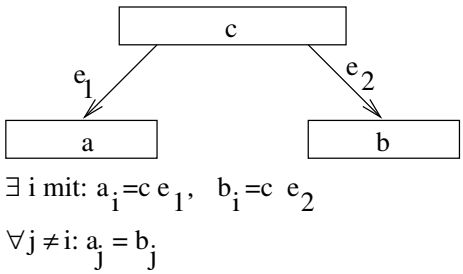


Abbildung 3.7: Fork-Regel

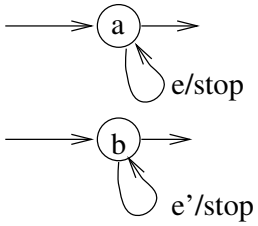


Abbildung 3.8: Zustände, welche die Ausgabe Stop erzeugen

Nach Aufstellen des o.a. Graphen versucht ASYL, durch Betrachtung dieses Graphen möglichst viele der Forderungen hinsichtlich der Kodierungen zu erfüllen. Dazu wird zunächst dem Knoten mit den meisten Kanten ein Code zugeordnet (`0000` in der Abb. 3.9). Anschließend wird allen mit diesem Knoten verbundenen Knoten ein Code zugeordnet. Dieser Prozess wird fortgeführt, bis alle Knoten erreicht sind (*breadth-first search*).

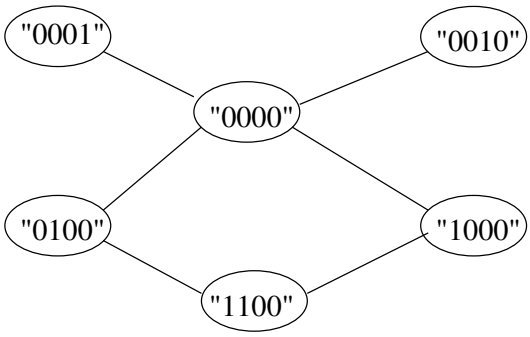


Abbildung 3.9: Graph der gewünschten einschrittigen Kodierungen

Nähere Informationen zum Umfang des Backtracking und zum Verhalten bei nicht erfüllbaren Abstandsforderungen entnehme man der Originalliteratur.

Optimale Verfahren zur Zustandskodierung sind seit vielen Jahren bekannt (siehe z.B. [HS66]). Für praktische Entwurfsaufgaben sind sie wegen ihrer Komplexität nicht geeignet. Einen Überblick über neuere Verfahren geben Reusch und Merzenich in [RM86]. In der Praxis werden heuristische Verfahren eingesetzt, welche lediglich (mehr oder weniger) gute, aber nicht mehr unbedingt optimale Lösungen liefern.

Eines der ersten in der Praxis wirklich eingesetzten Systeme ist das **LOGE–System** (siehe u.a. [GBH80]). Nachfolger hiervon (wie das MEGA-System [BLMR87]) werden einschließlich umfangreicher Entwurfsunterstützung kommerziell eingesetzt.

Aus der Theorie der formalen Sprachen ist bekannt, dass zu jeder sog. **regulären Sprache** ein endlicher Automat existiert, welcher entscheidet, ob es sich bei einem Textstring um ein Wort aus dieser Sprache handelt (siehe z.B. [Koh87]). Reguläre Sprachen kann man auch durch **reguläre Ausdrücke** charakterisieren. Ein Syntheseverfahren für endliche Automaten auf der Basis **regulärer Ausdrücke** stellt Ullman in [Ull84] vor.

Als ein gutes Werkzeug zur Zustandskodierung gilt **NOVA** [VSV89]. NOVA basiert auf der symbolischen

Minimierung[1] und liefert daher als Ergebnis auch die Binärkodierung zunächst nur symbolisch beschriebener Ein- und Ausgabesignale. NOVA besteht aus einer Reihe von Algorithmen, unter denen der Anwender auswählen kann. Unter diesen befindet sich auch ein exakter Algorithmus, der jedoch nicht immer in vertretbarer Zeit terminiert. Es werden dabei verschiedene mögliche Zustandscodes (nicht nowendig minimaler Länge) untersucht und die Realisierung der Booleschen Funktionen mittels ESPRESSO miteinander verglichen. NOVA berücksichtigt bei der Bewertung der Komplexität einer Realisierung nicht nur die Berechnung des Folgezustands, sondern auch die Berechnung der Ausgabefunktion. Eine ausreichende Darstellung von NOVA würde den Rahmen dieses Textes sprengen und es sei daher ebenfalls auf die Originalliteratur verwiesen. Für eine Reihe von Benchmarks zeigte Crastes [dP91], dass NOVA im Mittel 6,5% weniger Produktterme erzeugt als ASYL.

Für die Synthese von Automaten, welche durch lange, unverzweigte Folgen von Zuständen charakterisiert werden (wie z.B. Controller, welche indirekt durch die oben angegebenen imperativen Programme definiert werden) existieren spezielle Verfahren. Diese nutzen aus, dass wegen der Art der Abfolge von Kontrollschritten ein **Zähler als Zustandsregister** eine deutliche Reduktion der Logik zur Bestimmung des Folgezustandes ermöglicht, da zur Berechnung des Folgezustandes keine Produktterme mehr benötigt werden sondern lediglich der Zähler erhöht werden muss (das Zählen muss allerdings mittels eines zusätzlichen Kontroll–Signales abgeschaltet werden können). Eine Ausnutzung dieser Tatsache (und damit eine systematische Verbindung der Techniken der Mikroprogrammierung mit denjenigen der Automatentheorie) wurde von Amann und Baitinger [AB89] vorgeschlagen. Auf ihrem Verfahren bauen mehrere Controllersynthese-Verfahren auf.

Ein weiteres Synthesesystem, welches Techniken der Mikroprogrammierung nutzt, ist das in Dresden entwikkelte **MIPRE-System** [FR90].

Um zu möglichst schnellen Controllern zu kommen, werden die Eingabesignale z.Tl. erst bei Übergang vom Folgezustand in den übernächsten Zustand berücksichtigt. Zu diesem Zweck werden die Eingabesignale in einem Register zwischengepuffert. Man kommt so zu einer **Fließbandverarbeitung** (engl. *pipelining*): Während noch die Eingaben, die sich als Reaktion auf den gegenwärtigen Zustand ergeben, berechnet werden, werden bereits die Belegungen der δ- und λ-Netzwerke für den Folgezustand bestimmt, da dieser nicht mehr von den Eingaben abhängt. Diese Technik der **verzögerten Sprünge** (engl. *delayed jumps*) wird u.a. bei RISC-Prozessoren und im CATHEDRAL-Silicon Compiler [MRS87] benutzt.

Ein **Standard-Anwendungsbeispiel** der Controller-Synthese ist die Ampelsteuerung bei Mead und Conway [MC80].

3.4 Realisierung der Ausgabefunktion

3.4.1 Einfache Techniken

Für eine günstige Realisierung der Ausgabefunktion gibt es viele Techniken. Einige sind in der folgenden Liste aufgeführt:

1. Zunächst sind natürlich die Techniken der Logikoptimierung auf die Implementierung von λ anwendbar. Zum Teil berücksichtigen die Methoden der Zustandskodierung bereits das Ziel, diese Funktion möglichst kostengünstig zu realisieren.
2. Ferner lassen sich symbolische Minimierungstechniken einsetzen, um eine Kodierung der Ausgabewerte zu erreichen.

3.4.1.1 Techniken aus der Mikroprogammierung

Andere Techniken sind aus der Mikroprogrammierung bekannt. Diese Techniken sollen hier entsprechend der Literatur zur Mikroprogrammierung [AR76, Das79, Bod84] dargestellt werden:

1. *Direct control*, “keine Kodierung”

   Jedes Befehlsbit veranlaßt die Ausführung einer Operation.

[1] Eine gute Einführung in die symbolische Minimierung im Zusammenhang mit Controllern bietet De Micheli in [Mic87].

Dieser Form der Ansteuerung liegt die Idee zugrunde, dass Einheiten wie Addierer, Shifter, logische Einheiten u.s.w. voneinander getrennt aufgebaut sind und deren Ergebnisse über „Tore“ (*gates*) oder Treiber ausgewählt werden. In dieser Form wurde die Mikroprogrammierung ursprünglich von Wilkes vorgeschlagen. Sie ist die Ausgangsbasis für eine Reihe von Algorithmen, mit denen eine minimale Breite des Steuerwortes berechnet werden kann (Stichwort: **Kompatibilitätsklassen**, siehe [Bod84, DN90]).

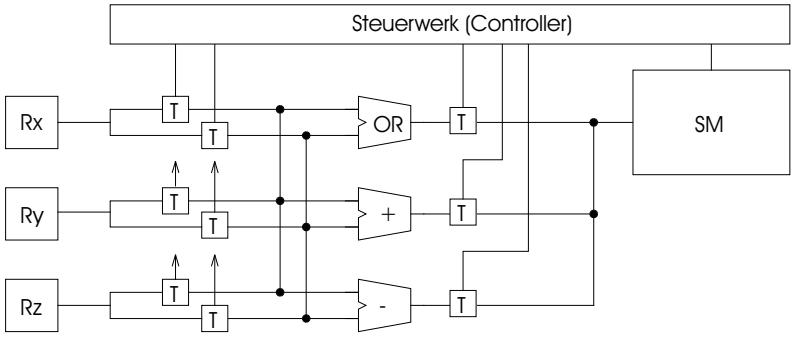


Abbildung 3.10: *Direct control* (T:Treiber, z.B.Tristate-Treiber)

In der Praxis kommt sie v.a. bei der Codierung von Schreiboperationen vor. Wir können sie beispielsweise zur Codierung der Schreiboperationen der mikroprogrammierten MIPS-Maschine gemäß Vorlesung „Rechnerstrukturen“ einsetzen (siehe Abb. 3.11). In diesem Fall ist jeweils in einem Ausgabefeld codiert, ob ein Register oder ein Speicher beschrieben wird.

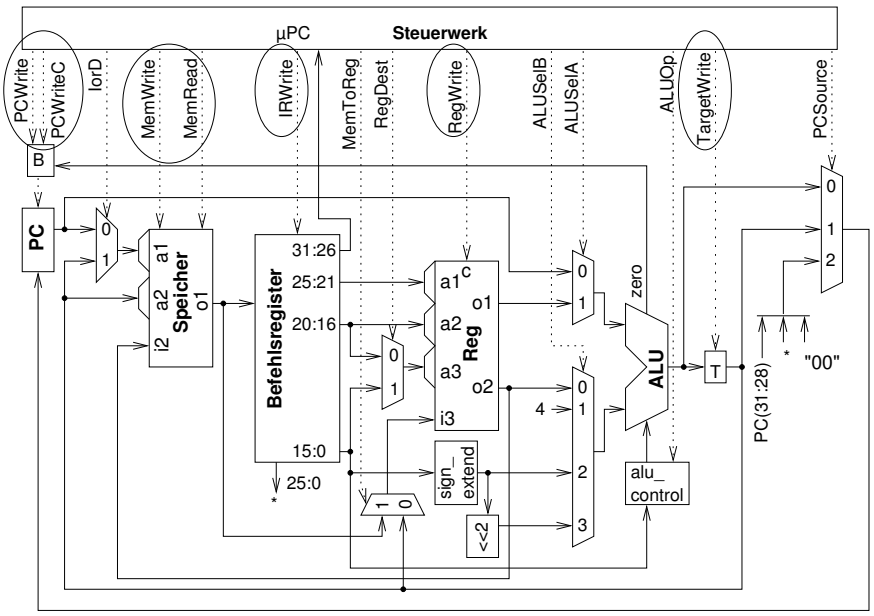


Abbildung 3.11: Per *direct control* codierte Ausgänge des Steuerwerks der MIPS-Maschine

2. *Single level encoding, direct encoding, minimal encoding*

   Operationen, die sich gegenseitig ausschließen, werden in einem Befehlsfeld gemeinsam kodiert. Die häufigste Anwendung findet sich bei Multifunktionseinheiten wie z.B. ALUs oder Multiplexern, die zu einer bestimmten Zeit höchstens eine Operation ausführen können. Es ergibt sich kein Verlust an möglicher Parallelarbeit.

   Ein Vorteil des *direct encoding* liegt in der Orthogonalität der einzelnen Operationen: die Auswahl von Quellen, durchgeführter Verknüpfung und Ziel sind voneinander unabhängig. Dies erleichert die Erstellung von Entwurfswerkzeugen.

   Diese Methode werden wir bei der MIPS-Maschine in allen Fällen an, in denen eine Quelle ausgewählt wird sowie bei der Codierung der Operation der arithmetisch/logischen Einheit (siehe Abb. 3.13).

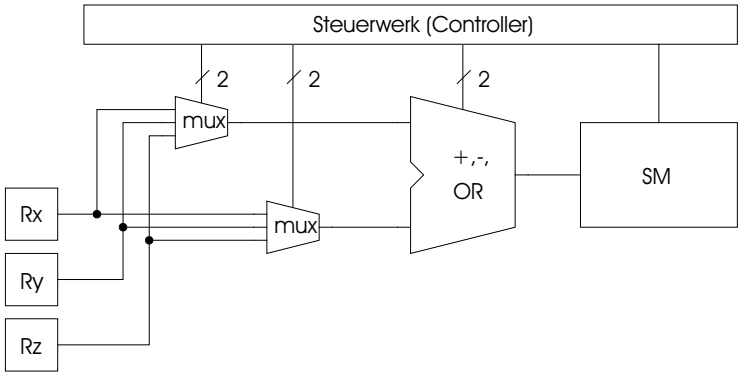


Abbildung 3.12: *Direct encoding*

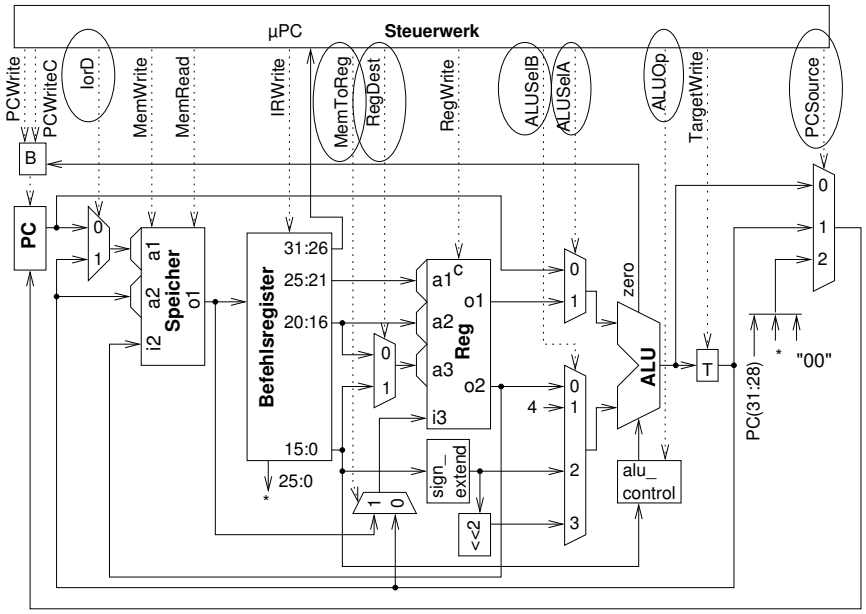


Abbildung 3.13: Per *minimal encoding* codierte Ausgänge des Steuerwerks der MIPS-Maschine

3. *Direct encoding* mit *sharing*

   Werden an zwei Steuereingängen entweder nie gleichzeitig Steuercodes benötigt oder werden stets nur gleiche Codes benötigt, so kommt man mit einem einzigen Steuerfeld im Befehl aus. Beispielsweise mögen die Steuereingänge von ALUs und Multiplexern miteinander verbunden sein. Die Orthogonalität geht dadurch verloren.

   Setzt man *sharing* im Rahmen eines Synthesesystems ein, so bleibt die Parallelität für das vorgegebene Verhalten erhalten; später zugefügte Erweiterungen können aber eventuell nicht die aufgrund der Hardware-Bausteine mögliche Parallelität ausnutzen.

   Auch diese Methode können wir bei der MIPS-Maschine einsetzen. Der Tabelle 3.1 können wir entnehmen, dass wir die Ausgabefelder IorD und Mem2Reg zusammenfassen können.

   Das entsprechend geänderte Rechenwerk zeigt die Abb. 3.15.

   Als Erweiterung des einfachen *Sharings* können wir beim Zusammenfassen noch eine einfache Umkodierung vornehmen. So zeigt das Beispiel der MIPS-Maschine, dass wir das Steuerfeld ALUSelA einfach durch Negation aus dem weniger signifikanten Bit des Feldes ALUSelB gewinnen können.

4. *Bit steering*

   Leiten wir ein Steuerfeld aus den Belegungen mindestes zweier anderer Felder ab, so hängt die Bedeutung eines Befehlsfeldes von einem anderen Befehlsfeld ab. Man spricht in diesem Fall von *bit steering* (siehe Abb. 3.17).

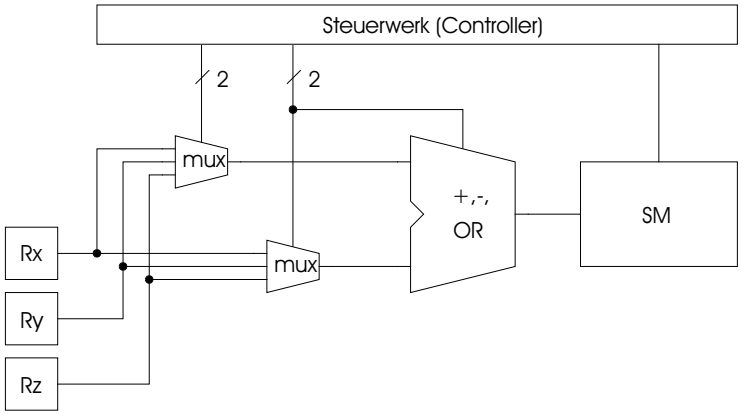


Abbildung 3.14: *Direct encoding* mit *sharing*

Zustand	Folgezustand	Folgezustands	PCWrite	PCWriteC	IorD	MemWrite	MemRead	IRWrite	Mem2Reg	RegDest	RegWrite	ALUSelB	ALUSelA	ALUOp	TargetWrite	PCSource
fetch	decode	Art der Bestimmung des	'1'	'0'	'0'	'0'	'1'	'1'	'X'	'X'	'0'	"01"	'0'	+	'0'	"00"
decode	f(Opcode)		'0'	'0'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"XX"	'X'	X	'0'	"XX"
mar	load, store		'0'	'0'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"10"	'1'	+	'1'	"XX"
load	fetch		'0'	'0'	'1'	'0'	'1'	'0'	'1'	'0'	'1'	"XX"	'X'	X	'0'	"XX"
store	fetch		'0'	'0'	'X'	'1'	'0'	'0'	'X'	'X'	'0'	"XX"	'X'	X	'0'	"XX"
rr1	rr2		'0'	'0'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"00"	'1'	IR	'1'	"XX"
rr2	fetch		'0'	'0'	'X'	'0'	'0'	'0'	'0'	'1'	'1'	"XX"	'X'	X	'0'	"XX"
branch	branch2		'0'	'0'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"11"	'0'	+	'1'	"XX"
branch2	fetch		'0'	'1'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"00"	'1'	=	'0'	"01"
jump	fetch		'1'	'0'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"XX"	'X'	X	'0'	"10"

Tabelle 3.1: Automatentabelle des Steuerwerks

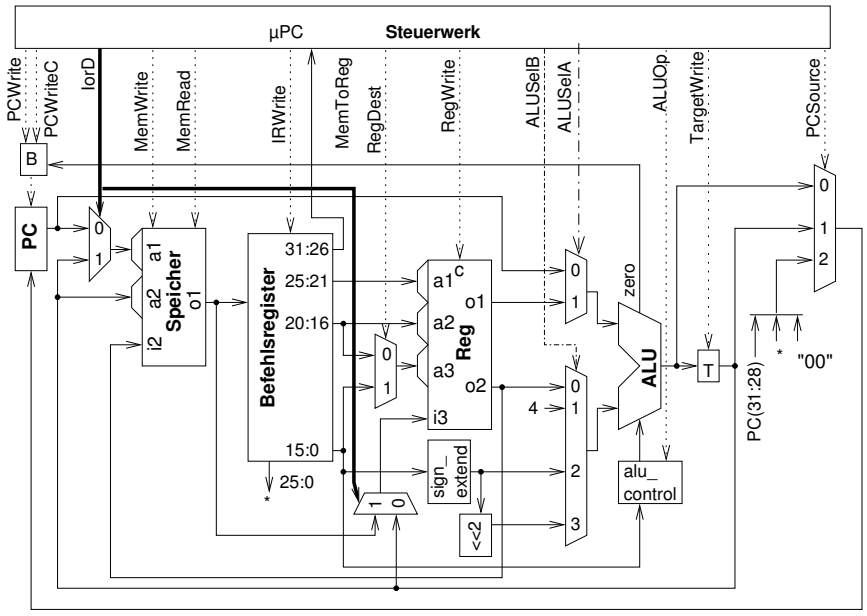


Abbildung 3.15: Per *sharing* codierte Ausgänge des Steuerwerks der MIPS-Maschine

5. *Residual control*

   Hängt die Funktion der Hardware nicht nur vom aktuellen Befehlswort, sondern auch von einem inneren Zustand ab, so spricht man von *residual control*. In diesem Fall können Steuercodes in speziellen Registern, den *residual control* – Registern abgespeichert werden (vgl. Abb. 3.18).

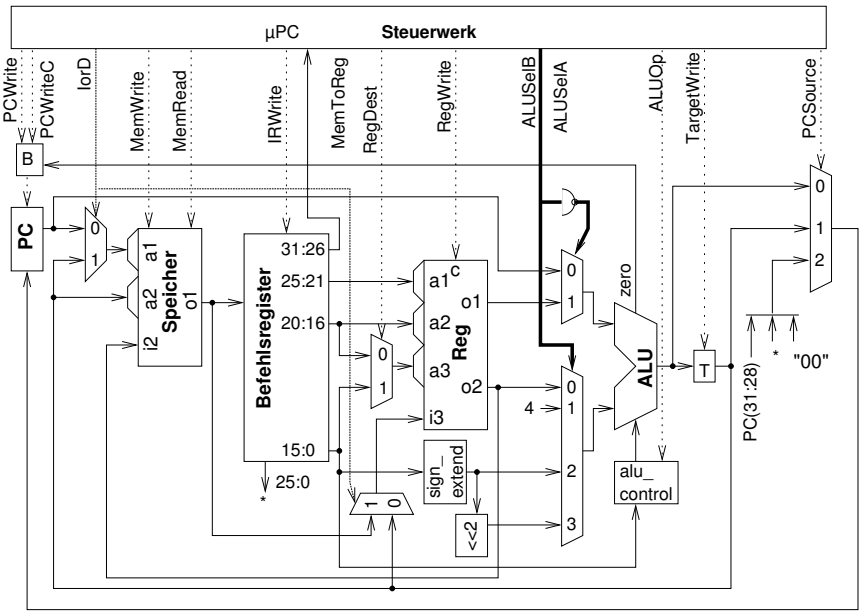


Abbildung 3.16: Umkodierung von Ausgaben des Steuerwerks

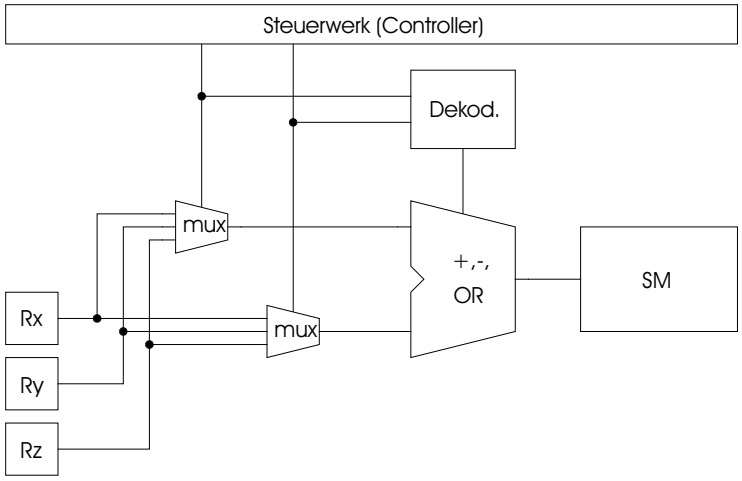


Abbildung 3.17: *Bit steering*

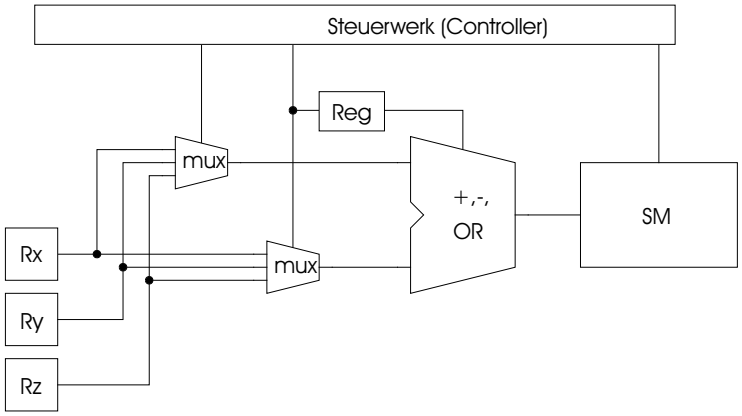


Abbildung 3.18: *Residual control*

Die Register werden auch als *Mode-Register* bezeichnet, z.B. bei DSPs. Diese Technik wird vor allem bei seltenen Wechseln der Belegung der Kontrolleingänge genutzt. Häufig handelt es sich dabei um die Emulation (Nachbildung) von Befehlssätzen, wie z.B. um die Emulation des (16-Bit-) PDP-11 Befehlssatzes auf (32-Bit-) VAX–Rechnern. Hierdurch können z.B. im PDP-11 Modus Adreßbits mit Null belegt werden.

6. *Format shifting*

   Hängt die Bedeutung eines Feldes von der Belegung eines *residual control* – Registers ab, so bezeichnet man dies als *format shifting* .

7. *Two-level control store*

   Wird ein großer Teil der Hardware über einen solchen Dekoder (oder - äquivalent - über ein ROM) gesteuert, so spricht man vom *two-level control store* . Alle vorkommenden Kontrollworte werden genau einmal im ROM abgelegt und dort **adressiert** (indirekte Adressierung der Kontrollworte)[2] Abb. 3.19 zeigt die Hardware-Organisation unter Einschluß der Logik zur Bestimmung des Folgezustandes.

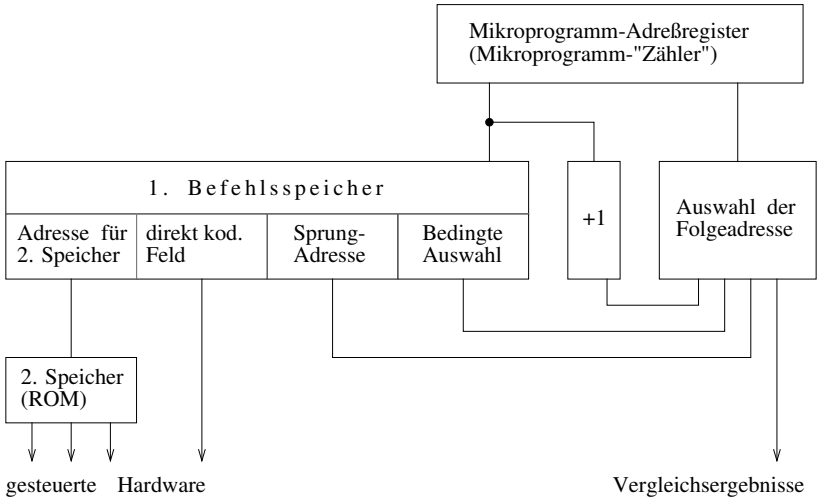


Abbildung 3.19: *Two level control store*

Direkt im ersten Befehlsspeicher werden häufig noch ein Teil der Steuerinformation direkt kodiert (z.B. Operanden). Beim Vergleich auf Gleichheit von Steuerworten spielt dieser Teil keine Rolle. Die Anzahl der verschiedenen Steuerworte, die im zweiten Speicher hinterlegt werden müssen, kann so reduziert werden. Bislang ist kein CAD-Werkzeug bekannt, welches die insgesamt benötigte Speichergröße minimiert und dabei den direkt zu kodierenden Teil der Steuerinformation automatisch auswählt.

Weiterhin enthält der erste Speicher noch ein Feld, welches die Kodierung einer Sprungbedingung erlaubt. Sprünge können abhängig von im Rechenwerk berechneten Vergleichsergebnissen ausgeführt werden. Es können stets nur zwei Folgezustände benutzt werden: der Zustand, der durch Inkrementieren des Mikroprgramm-Zählers und der Zustand, der als Sprungsadresse hinterlegt ist.

Abb. 3.20 zeigt die Benutzung dieser Hardware zur Realisierung eines konkreten Flußgraphen.

Links ist ein Flußdiagramm eines Mooreautomaten zu sehen. Die Rechtecke enthalten jeweils die Ausgaben, am linken Rand sind die Zustände kodiert. Verweigungen gehören jeweils zu dem darüber notierten Zustand. Es wurde angenommen, daß das direkt kodierte Feld nur aus dem “rechtesten” Bit besteht. Rechts enthält die Abbildung die konkreten Inhalte des Speichers ersten und zweiter Stufe. Wichtig ist, daß die Anzahl der Einträge im zweiten Speicher relativ klein ist, da -vom direkt kodierten Feld abgesehen- nur drei verschiedene Ausgabeworte vorkommen.

Benutzt wird diese die *two level control store*-Technik z.B. im Mikroprozessor Motorola MC 68.000. Sie wird dort allerdings als Nanoprogrammierung bezeichnet.

[2]Eine ähnliche Idee stellt die indirekte Adressierung von Farbwerten in einer *color lookup table* dar (siehe Vorlesungen zu Graphischen Systemen).

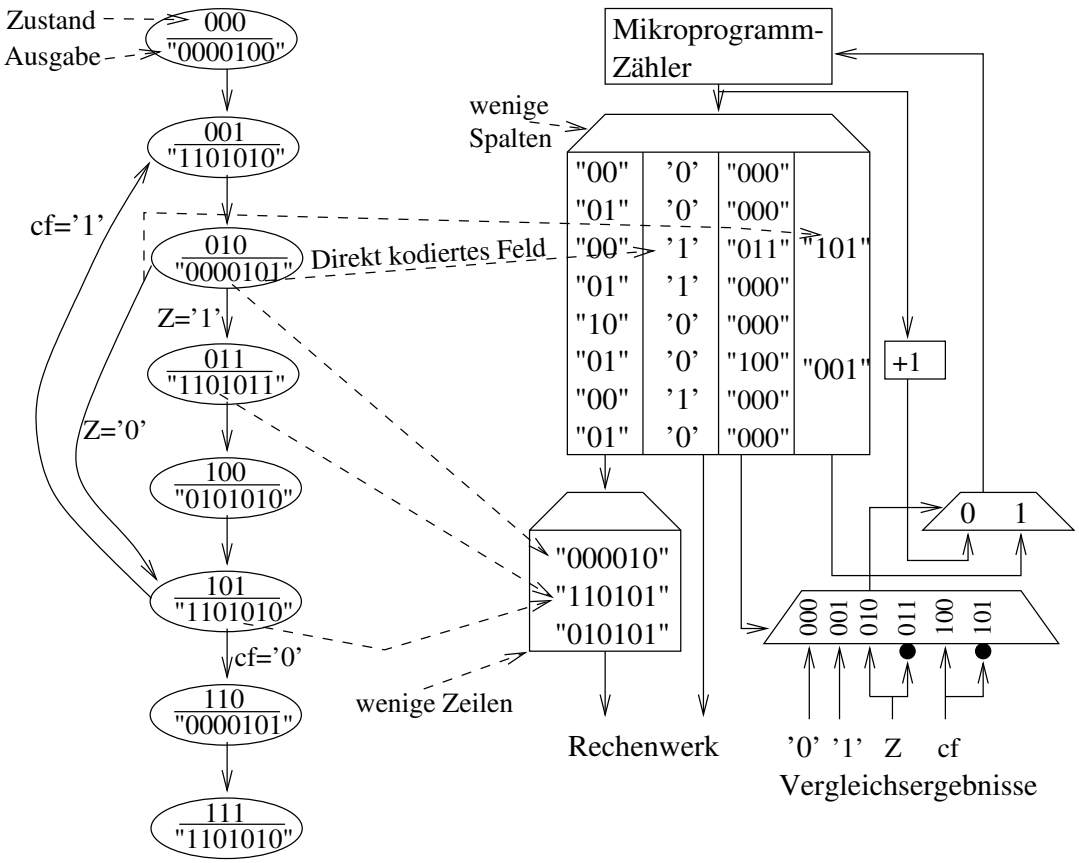


Abbildung 3.20: Realisierung eines konkreten Flußgraphen

3.4.2 Befehlsfeldüberlagerung in TODOS

TODOS [Mar86] benutzt die "direct encoding"–Methode: Jedem Steuereingang wird direkt ein Befehlsfeld zugeordnet. Da in einigen Befehlen die Beschaltung der Steuereingänge redundant ist, können sich die Befehlsfelder überlagern. Durch eine solche Überlagerung reduziert sich die Befehlswortbreite. Klassische Verfahren zur Wortbreitenreduktion (vgl. [Das79]) sind im Fall des MSS nicht anwendbar, da sie auf dem "direct control" beruhen, welches Multifunktionsbausteine nicht berücksichtigt. Für TODOS wurde daher ein spezielles Verfahren zur Überlagerung von Befehlsfeldern entwickelt und implementiert.

Sei l_i die Länge des Befehlsfeldes i in Bits. Sei $c_{i,j} =$ **true**, falls die Felder i und j in mindestens einem Befehl nicht gleichzeitig redundant sind und $c_{i,j} =$ **false** sonst. Gesucht ist eine Anordnung der Felder im Befehlsformat derart, daß insgesamt ein möglichst schmales Befehlswort resultiert.

Dieses Problem ist äquivalent zu einem Scheduling–Problem von Jobs der Ausführungszeit l_i mit durch $c_{i,j}$ beschriebenen Ressource–Konflikten. Zur Lösung wird daher ein Scheduling-Verfahren benutzt:

1. Den jeweils längsten Feldern wird zuerst eine Position im Befehlsformat zugeordnet.
2. Unter den gleich langen Feldern haben die Felder mit den meisten Konflikten Vorrang.
3. Kann ein Feld aufgrund der $c_{i,j}$ mehreren Positionen innerhalb des Befehlsformates zugeordnet werden, so erfolgt die Auswahl nach der "Best fit"–Methode.

Ein Beispiel hierfür zeigt Abb. 3.21.

Die Felder f_i sollen jeweils einen Baustein des Operationswerkes mit der benötigten Steuerinformation versorgen. Die längsten Felder sind f_1 und f_2. Da f_1 mehr Konflikte besitzt, wird diesem Feld zuerst eine absolute Bitposition im Steuerwort zugeordnet. Danach wird f_2 betrachtet. Da es gleichzeitig mit f_1

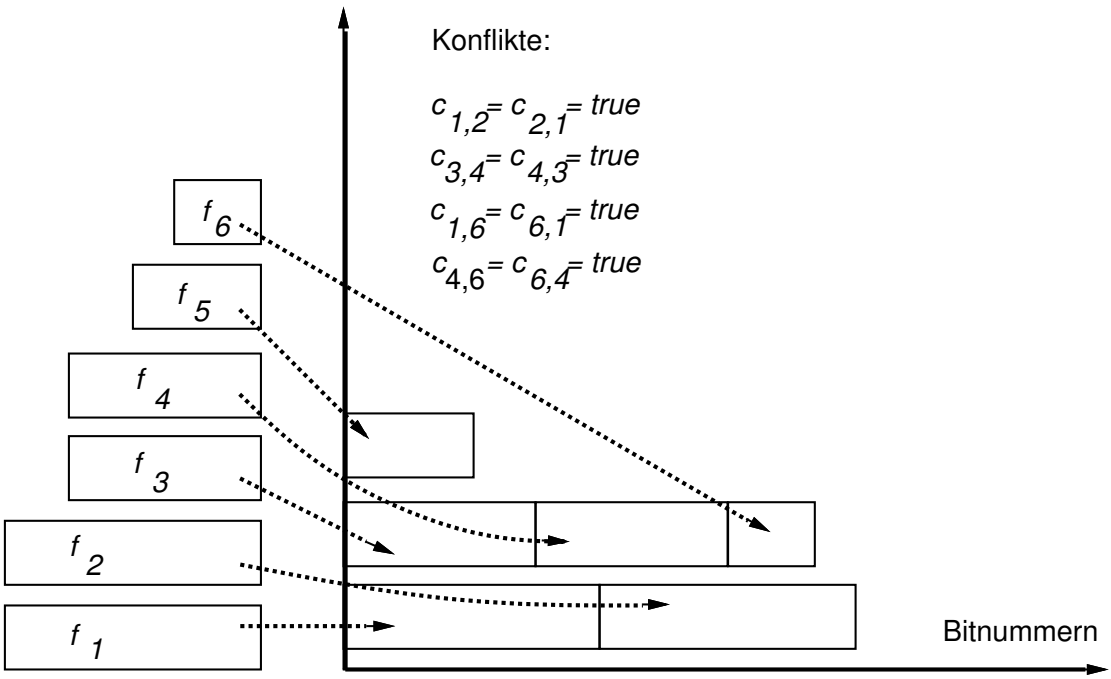


Abbildung 3.21: Wortbreiten-Reduktion bei TODOS

benötigt wird, darf es sich nicht mit diesem Feld überlappen. f_3 dagegen darf sich wieder mit f_1 überlappen. Auf diese Weise wird die Zuordnung fortgesetzt, bis auch das kürzeste Feld behandelt wurde. In praktischen Anwendungen werden Wortbreiten-Reduktionen zwischen 0% und 50% erzielt.

Kapitel 4

Logik-Synthese

Dieser Abschnitt behandelt Syntheseverfahren, die im wesentlichen von einer Spezifikation einer Funktion in Form von Booleschen Gleichungen ausgehen. Diese Formen der Synthese werden als Logiksynthese bezeichnet. In diesem Fall bezeichnet der Zusatz zum Wort "Synthese" die Ebene der Spezifikation. Leider werden die Zusätze zum Wort "Synthese" uneinheitlich gewählt. Zum Teil bezeichnen sie die Ebene der Spezifikation, zum Teil die der Implementierung.

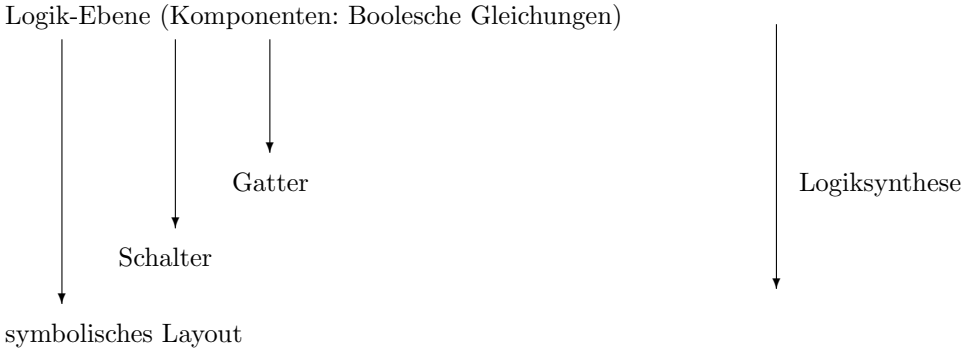


Abbildung 4.1: Zum Begriff "Logiksynthese"

Die Implementierung kann auf unterschiedlichen Ebenen beschrieben werden. Traditionell wurden vor allem Realisierungen durch Gatter erzeugt. Als Verfahren hierzu existieren u.a. das klassiche Quine–McCluskey– und das KV–Verfahren. Für die VLSI–Technik sind diese Verfahren unzureichend, da beliebig verschaltete Gatternetze im allgemeinen keine effiziente Flächenausnutzung erlauben. Daher werden häuifig andere, regulärere und dadurch flächeneffizientere Realisierungen erzeugt. In diesen Bereich fallen z.B. PLAs. Hierbei wird nicht nur der Aufbau der Arrays aus Transistoren, sondern auch deren gegenseitige Lage festgelegt. Derartige Synthesesysteme überdecken also den Bereich von Booleschen Gleichungen bis hin zum symbolischen Layout (vgl. Abb. 4.1).

4.1 Klassische Minimierungstechniken für 2-stufige Logik

4.1.1 Definitionen

PLAs können zur flächeneffizienten, 2-stufigen Realisierung Boolescher Funktionen eingesetzt werden.

Die spezifischen Eigenschaften eines bestimmten PLAs werden vielfach in seiner **Individualität** (engl. "personality") ausgedrückt. Sie soll durch zwei Matrizen '*and*' und '*or*' dargestellt werden. '*and*' und '*or*' beschreiben die im folgenden definierten Abbildungen.

Def.: Die Funktion $and : [1..k] \times [1..n] \rightarrow [0..2]$ ist definiert durch:

$and(i,j) = 0$, falls in Produktterm i die Variable x_j invertiert vorkommt [1],

$and(i,j) = 1$, falls in Produktterm i die Variable x_j nicht-invertiert vorkommt,

$and(i,j) = 2$, falls in Produktterm i die Variable x_j nicht vorkommt.

Def.: Die Funktion $or : [1..k] \times [1..m] \rightarrow [0..1]$ ist definiert durch:

$or(i,j) = 0$, falls der Produktterm i in die Berechnung von y_j nicht eingeht,

$or(i,j) = 1$, falls der Produktterm i in die Berechnung von y_j eingeht.

Beispiel:

Gegeben sei das PLA mit der Individualität

and	*or*
200	11
121	01
122	10

Die erste Zeile der *and*-Matrix realisiert den Term $\overline{x_2}\ \overline{x_3}$, die zweite den Term x_1x_3 und die dritte den Term x_1. Die *or*-Matrix legt dann fest, daß der erste und der dritte Term in die Berechnung von y_1 und der erste und der zweite Term in die Berechnung von y_2 eingehen. Es werden also die folgenden Funktionen realisiert:

$y_1 := \overline{x_2}\ \overline{x_3} + x_1$
$y_2 := \overline{x_2}\ \overline{x_3} + x_1\ x_3$

Im folgenden soll nun versucht werden, die Fläche von PLAs zu reduzieren.

Zur Minimierung der Fläche von PLAs gibt es verschiedene Optimierungstechniken. Als erstes sollen hier Techniken der Booleschen Minimierung angesprochen werden. Später werden speziell auf PLAs zugeschnittene Techniken folgen.

Für PLAs sind die Techniken der klassischen Booleschen Minimierung wie z.B. die Methoden nach Quine-McCluskey und Karnaugh-Veitch (siehe z.B. [Koh87]), die auf 2-stufige Realisierungen beschränkt sind, im Prinzip anwendbar. Allerdings sind sie für die Implementierung durch ein CAD-Programm und für praktisch relevante Problemgrößen u.a. aufgrund ihrer Laufzeiten nicht geeignet. Eine der Ursachen liegt darin, daß das Quine-McCluskey-Verfahrens zunächst die Generierung **aller** Primterme voraussetzt. Deren Zahl wächst aber, wie man zeigen kann, exponentiell mit der Anzahl der Eingangsvariablen.

4.1.2 Kodierung der Eingangsvariablen

Häufig kommen als Eingabe an ein PLA nicht alle möglichen Werte der Eingabevariablen vor. Man kann die Eingabevariablen evtl. mit einem kürzeren Bitmuster kodieren, um so Spalten für die Eingabevariablen einzusparen.

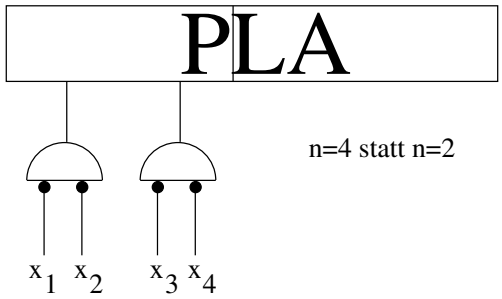


Abbildung 4.2: Reduktion der Spaltenzahl, falls Funktion von x_1x_2 und x_3x_4 abhängt

Diese Technik wird in den Verfahren benutzt, die mit symbolischen Variablen (s.u.) arbeiten.

[1] Bei der MOS-Technik ist in diesem Fall wegen der oben beschriebenen Benutzung komplementierter Eingangssignale die **nicht-invertierte** Spaltenleitung an einen Transistor angeschlossen.

4.1.3 Streichen von Variablen in Termen

Eine der schnellen Vereinfachungstechniken besteht in der Suche nach Variablen, welche in einzelnen Termen redundant sind[2]. Damit wird die Fläche des PLA noch nicht unmittelbar reduziert. Bei einer anschließenden Suche nach redundanten Zeilen können sich dadurch aber zusätzliche Reduktionsmöglichkeiten ergeben.

Es soll also untersucht werden, ob die Variable x_j in dem Term

$$x_j * f(x_1, (x_j), .., x_n)$$

überflüssig ist, der Term also durch

$$f(x_1, (x_j), .., x_n)$$

ersetzt werden kann. Darin soll (x_j) bedeuten, daß f nicht von der Variablen x_j abhängt.

Diese Ersetzung ist zulässig, falls $\overline{x_j} * f(x_1, (x_j), .., x_n)$ durch die übrigen Terme bereits überdeckt wird (d.h. durch das Weglassen von x_j entstehen keine weiteren Einsen z.B. in der KV–Tafel). Diese Ersetzung heißt **Term-Expansion** (engl. "raising").

Um zu überprüfen, ob "raising" zulässig ist, geht man zunächst einmal davon aus, daß die Individualität eines unoptimierten PLAs vorgegeben ist. Diese wird mittels entsprechender Algorithmen (siehe z.B. [Ull84]) auf eine mögliche Term-Expansion überprüft.

4.1.4 Elimination redundanter Terme

Die von PLAs belegte Fläche ist proportional zur Anzahl der Zeilen, also zur Anzahl der Produktterme. Eine der Optimimierungsmöglichkeiten für PLAs besteht folglich in der **Elimination** redundanter PLA-Zeilen. Eine Zeile eines PLA kann insgesamt entfernt werden, wenn alle logischen Einsen, die diese Zeile an den Ausgängen erzeugt, auch schon durch die übrigen Zeilen erzeugt werden[3]. Formal wird dies durch das folgende Lemma ausgedrückt:

Lemma: Zeile row ist redundant $\iff$
$\forall c \in [1..m]$ mit $or(row, c) = 1$ gilt:
$R := \{r | r \in [1..k],\ or(r, c) = 1, r \neq row\}$ überdeckt die Zeile row.

Auf der Basis dieses Lemmas lassen sich wie bei der allgemeinen Booleschen Minimierung Produktterme streichen. Beispiel:

Gegeben sei das PLA:

and	*or*
2101	1
0002	1
0211	1
1112	1
1021	1

Durch abwechselndes "raising" und Elimination redundanter PLA-Zeilen läßt sich dieses zu einem PLA mit der folgenden Individualität vereinfachen:

and	*or*
2221	1
0002	1
1112	1

Der Flächenbedarf kann also wegen der geringeren Anzahl von Produkttermen wesentlich reduziert werden.

Da völlig ohne Überdeckungstabellen gearbeitet wird, kann allerdings keine optimale Lösung garantiert werden. Für Details bezüglich dieses einfachen Verfahrens sei auf die Literatur [Ull84] verwiesen.

[2] Dies entspricht in KV-Diagrammen dem Übergang auf jeweils größere Blöcke von Einsen.

[3] Dies entspricht in KV-Diagrammen dem Weglassen eines Blockes von Einsen, welche schon durch andere Blöcke überdeckt werden.

4.1.5 ESPRESSO

Die bislang erwähnten Methoden der Minimierung sind für praktische Anwendungen zu primitiv. Bessere Ergebnisse werden z.B. mit dem Programm ESPRESSO von der Universität Berkeley erzielt.

ESPRESSO geht davon aus, daß die zu realisierende Funktion als Summe von Produkttermen gegeben ist. Der grobe Ablauf des Verfahrens ist dann der folgende[4]:

1. Expansion der Terme zu Primtermen
2. Extraktion essentieller Primterme, Elimination total redundanter Primterme
3. Behandlung der partiell-redundanten Terme
   (a) Reduktion der Restterme
   (b) erneute Expansion der Primterme, Entfernung von redundanten Primtermen

   bis keine redundanten Primterme mehr entfernt werden können

ESPRESSO macht gegenüber dem klassischen Ansatz zwei wesentliche Unterschiede. Erstens verzichtet man auf die Erzeugung **aller** Primterme, sondern man verallgemeinert nur die **gegebenen** Terme zu Primtermen. Zweitens versucht man nur, aus den Primtermen eine kostengünstige, aber nicht notwendig die optimale Überdeckung auszuwählen.

Zu den einzelnen Schritten des Verfahren sollen nun noch die folgenden Bemerkungen gemacht werden:

1. **Expansion der Terme zu Primtermen:**

   Man ersetzt nur noch jeden Term der Ausgangsüberdeckung durch einen aus diesem abgeleiteten Primterm. Infolgedessen bleibt die Anzahl der Terme bei diesem Schritt konstant und der Speicherbedarf des Algorithmus erhöht sich nicht. Dabei überdecken die Primterme nicht nur jene Werte der Eingangsvariablen, in denen die Funktion eine "1" liefern soll, sondern ggf. auch Argumentwerte, für die der Funktionswert redundant ist. Das Ziel der Expansion ist eine Primüberdeckung, deren Terme sich hochgradig überschneiden. Abb. 4.3 zeigt diesen Vorgang für ein Beispiel anhand eines KV-Diagramms (das Diagramm wird nur zur Verdeutlichung, nicht zur Minimierung benutzt).

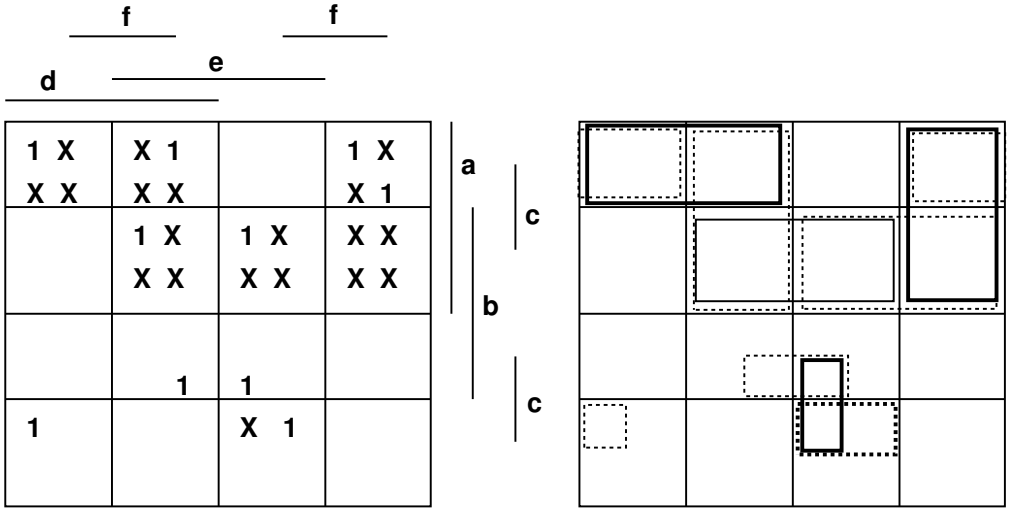


Abbildung 4.3: ESPRESSO: a) Ausgangsfunktion; b) Expansion

2. **Extraktion essentieller Primterme, Elimination total redundanter Primterme:**

   Die Terme teilt man nun in drei Klassen ein:

   1. Ein Primterm, für den Argumentwerte existieren, an denen einzig dieser Term für eine Überdekkung sorgt, heißt **essentieller Primterm**.

[4] Darstellung nach Pusch [Pus90].

2. Ein Primterm, welcher allein schon von den essentiellen Primtermen überdeckt wird, heißt **total redundant**.
3. Alle übrigen Terme heißen **partiell redundant**.

Total redundante Primterme können ohne Verlust an Allgemeinheit eliminiert werden (vgl. Abb. 4.4).

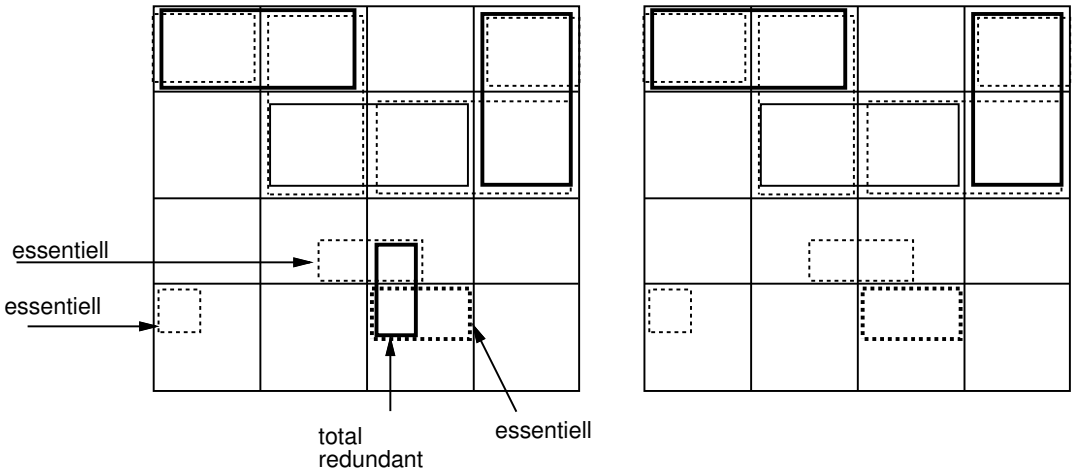


Abbildung 4.4: ESPRESSO: a) Essentielle und total redundante Terme; b) Elimination

3. **Behandlung der partiell redundanten Terme:**

   Partiell redundante Terme werden jeweils durch einen anderen Term überdeckt. Ein einzelner partiell redundanter Term kann also entfernt werden, ohne daß die Überdeckungseigenschaft verlorengeht. Für mehrere Terme gilt dieses nicht.

   (a) **Bestimmung einer Überdeckung:**
   Zur Bestimmung einer Auswahl von zu eliminierenden Termen, welche die Überdeckungseigenschaft erhält, benutzt ESPRESSO einen Algorithmus für das sog. “minimum set covering problem” [KGN87] (vgl. Abb. 4.5 a)).

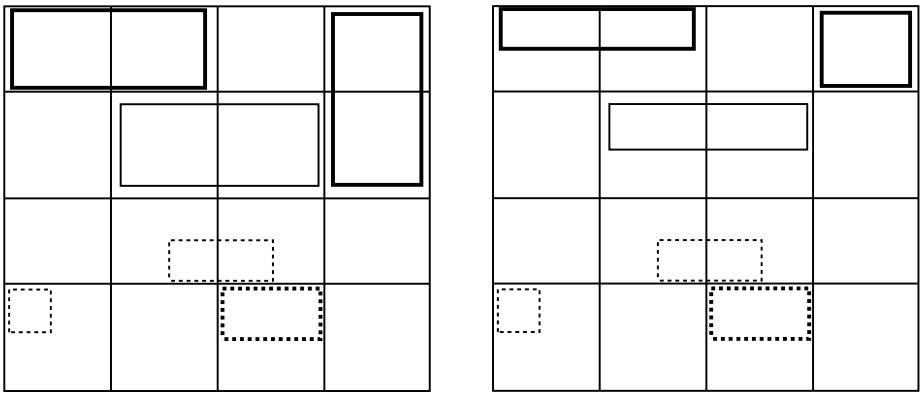


Abbildung 4.5: ESPRESSO: a) Überdeckung; b) Reduktion

(b) **Reduktion der Restterme:**
Zunächst versucht man, sich von dem bisherigen lokalen Minimum zu entfernen, indem man in Umkehrung des "raising" die partiell-redundanten Terme auf eine Überdeckung der Argumentwerte begrenzt, die nicht durch andere Terme überdeckt werden (vgl. Abb. 4.5 b)). Durch diese sog. **maximale Reduktion** kann es passieren, daß ein Term völlig überflüssig wird, weil er nur noch don't cares überdeckt und so die Anzahl der Terme (welche als Kostenfunktion gilt) reduziert wird. Die Wirkung dieses Schritte hängt ganz entscheidend von der Reihenfolge ab, in der ESPRESSO die Reduktion durchgeführt.

(c) **erneute Expansion der Primterme, Entfernung von redundanten Primtermen:**
Nach der Reduktion liefert eine erneute Expansion in der Regel wieder andere Primterme, für die die bislang angeführte Prozedur wiederholt werden kann.

Mit diesem Verfahren kann keine global optimale Auswahl garantiert werden. Zur Verbesserung führt man die o.a. Schritte in einer Schleife aus:

GOTO (a) bis keine redundanten Primterme mehr entfernt werden können.

Die gefundene Lösung ist auch hinsichtlich der Minimierung der Anzahl der Produktterme nicht notwendig minimal, u.a. da nicht alle Reihenfolgen der Termreduktion untersucht werden. ESPRESSO ist dennoch aufgrund des durchsuchten Lösungsraumes ein laufzeitintensives Programm.

Eine Erweiterung von ESPRESSO ist ESPRESSO-MV. "MV" steht dabei für die Fähigkeit, statt Boolescher Argument-Variablen auch mehrwertige und **symbolische Variable** zu behandeln. Unter symbolischen Variablen versteht man hier Variable, die symbolische Namen als Werte annehmen können, ohne daß für diese Namen eine Binärkodierung festgelegt wurde. Symbolische Variable bieten den Vorteil, daß ihre Kodierung durch das Minimierungswerkzeug festgelegt werden kann und damit häufig eine effizientere Realisierung möglich ist. Wir werden später sehen, daß symbolische Variable insbesondere bei der ALU-Erzeugung und bei der Zustandskodierung von Vorteil sind.

Eine vollständige Beschreibung von ESPRESSO oder ESPRESSO-MV würde den Rahmen dieses Buches sprengen und es muß daher auf die Literatur verwiesen werden [BHM84, RSV87].

4.1.6 Faltung der AND-Plane

Eine andere Form der Flächenreduktion von PLAs ist die sog. **Faltung** (engl. "folding"). Bei der Faltung der AND-Plane (engl. "column folding") nutzt man aus, daß viele Eingangsvariablen nur in einen Teil der Terme eingehen. Man unterbricht dann die vertikalen Leitungen der AND–Plane und führt von oben und von unten Eingangsvariable zu.

Eine Zuführung der Eingangsvariablen von oben könnte für manche Variable einen erheblichen Mehraufwand an Verdrahtung bedeuten. Es gibt daher Faltungsverfahren, die Vorgaben hinsichtlich der Zuführung von oben oder unten berücksichtigen (sog. "constrained (column) folding"). Benutzt man PLAs zur Realisierung endlicher Automaten, so lassen sich zumindest die Zustandsbits problemlos von oben zuführen, da diese von der OR–Plane auch problemlos nach oben abgeführt werden können.

Um umfangreiche Verdrahtungen zu vermeiden, soll im weiteren die Einschränkung gemacht werden, daß die invertierten und die nicht–invertierten Eingangsvariablen stets von derselben Seite nebeneinander zugeführt werden sollen. Für die Verwendung der Spalten der AND–Plane bleiben dann noch zwei Möglichkeiten: Die Spalten werden entweder für zwei Signale gleicher oder gegensätzlicher Polarität genutzt (siehe Abb. 6.10). Die beiden Möglichkeiten werden im folgenden als "gerade" bzw. "vertauscht" bezeichnet.

Zur Charakterisierung der Trennstellen p, q genügt eine einfache Indizierung, da jede Eingangsvariable nur in einem Paar vorkommt. Wir verwenden hier den Index der oberen Variablen.

Zur Durchführung der Faltung wird ein Algorithmus benötigt, der die Eingangsvariablen zu geordneten Paaren (x_i, x_j) gruppiert, wobei x_i jeweils oben und x_j unten zuzuführen ist.

Wesentlicher Bestandteil des im folgenden angegebenen Algorithmus ist ein Graph, mit dem die Zulässigkeit einer partiellen Lösung überprüft werden kann. Dieser Graph enthält je einen Knoten für jeden Produktterm, d.h. für jede Zeile des PLAs. Zusätzlich enthält der Graph je einen Knoten, der die Trennstelle zwischen den beiden vertikalen Leitungssegmenten symbolisiert. Der Graph enthält eine (gerichtete) Kante vom Knoten n_i zum Knoten n_j genau dann, wenn das durch n_i repräsentierte Objekt oberhalb von dem durch n_j repräsentierten Objekt anzuordnen ist.

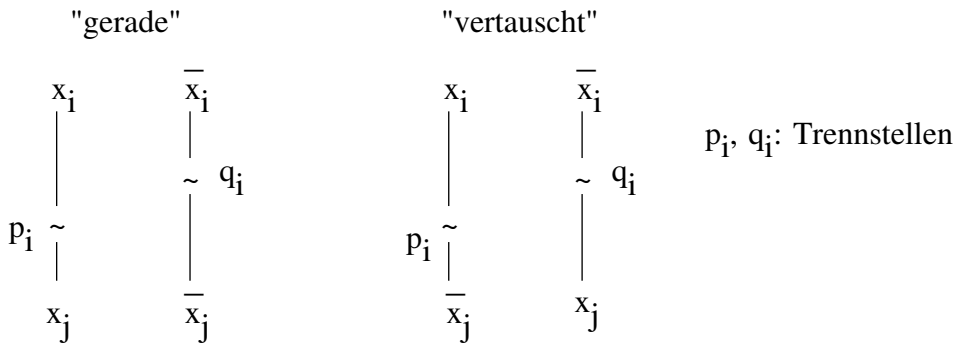


Abbildung 4.6: Möglichkeiten der Zuführung von zwei Variablen

Sofern der Graph zyklenfrei ist, beschreibt er eine partielle Ordnung. Die Anordnung der Objekte kann dann in einer dazu kompatiblen totalen Ordnung erfolgen. Hierzu benötigt man einen Algorithmus zum sog. **topologischen Sortieren** [AHU83]. Enthält der Graph Zyklen, so existiert keine zulässige Anordnung.

Der Test auf Zyklenfreiheit bildet daher den Kern des folgenden Greedy–artigen Algorithmus. Der Graph G wird zunächst mit je einem Knoten für jeden Produktterm initialisiert. Alle bislang noch nicht gepaarten Eingangsvariablen werden auf eine mögliche Paarung hin untersucht. Dabei kann jede Variable sowohl von oben als auch von unten und sowohl vertauscht wie auch "gerade" zugeführt werden.

Im geraden Fall wird eine Paarung von x_i und x_j unmittelbar zurückgewiesen, falls ein Produktterm (eine Zeile der AND–Plane) existiert, der sowohl x_i wie auch x_j oder sowohl $\overline{x_i}$ wie auch $\overline{x_j}$ zu seiner Berechnung benötigt. Trivialerweise kann unter diesen Bedingungen keine zulässige Trennstelle zwischen den vertikalen Leitungen gefunden werden. Im vertauschten Fall sind die Rollen von x_j und $\overline{x_j}$ auszutauschen.

Soll die Zulässigkeit einer Paarung von x_i und x_j überprüft werden, so wird der Graph G temporär um zwei Knoten p_i, q_i erweitert, die die Trennstellen der beiden vertikalen Leitungen symbolisieren. Es werden sodann gerichtete Kanten erzeugt, welche die Restriktionen bezüglich der Anordnung widerspiegeln. Für i oben und den "geraden" Fall werden z.B. folgende Kanten erzeugt:

- Für alle Produktterme r, zu deren Berechnung x_i benötigt wird, eine Kante von r nach p_i. Diese Kante drückt aus, daß der Produktterm r in einer Zeile oberhalb der Trennstelle p_i berechnet werden muß.
- Für alle Produktterme r, zu deren Berechnung x_j benötigt wird, eine Kante von p_i nach r. Diese Kante drückt aus, daß der Produktterm r in einer Zeile unterhalb der Trennstelle p_j berechnet werden muß.
- Für alle Produktterme r, zu deren Berechnung $\overline{x_i}$ benötigt wird, eine Kante von r nach q_i. Diese Kante drückt aus, daß der Produktterm r in einer Zeile oberhalb der Trennstelle q_i berechnet werden muß.
- Für alle Produktterme r, zu deren Berechnung $\overline{x_j}$ benötigt wird, eine Kante von q_i nach r. Diese Kante drückt aus, daß der Produktterm r in einer Zeile unterhalb der Trennstelle q_i berechnet werden muß.

Sofern der generierte temporäre Graph zyklenfrei ist, wird die erzeugte Paarung protokolliert und der temporäre Graph wird als aktueller Graph übernommen.

```
Graph G := (V :=Produktterme; E := {});
Used := {};
FOR all i,j ∈ [1..n] − Used DO
 FOR i oben, i unten DO
  FOR gerade, vertauscht DO
   IF ex. keine konfliktbehaftete Zeile THEN
    BEGIN
     V' := V ∪ {p_i, q_i}; E' := E;
     Für i oben und gerade:
      ∀r mit and(r,i) = 1 : E' := E' ∪ (r, p_i);
```

```
  ∀r mit and(r, j) = 1 : E' := E' ∪ (p_i, r);
  ∀r mit and(r, i) = 0 : E' := E' ∪ (r, q_i);
  ∀r mit and(r, j) = 0 : E' := E' ∪ (q_i, r);
 Entsprechend für i unten oder vertauscht.
 IF G' = (V', E') ist zyklenfrei THEN
  BEGIN G := G'; E := E'; Used := Used ∪ {i, j} END;
END;
```

Abb. 4.7 enthält ein Beispiel (nach Ullman [Ull84]) für eine Spezifikation einer AND-Plane.

and:	A	B	C	D	E
a	0	0	1	2	2
b	1	0	2	0	2
c	2	1	2	2	0
d	2	2	0	0	1
e	0	2	2	2	2
f	2	2	2	1	1

A–E : Eingangsvariable
a–f : Produktterme

Abbildung 4.7: Beispiel einer ungefalteten AND–Plane

Wegen Term a wird eine gerade Paarung von A und B unmittelbar zurückgewiesen. Wegen Term b können sie nicht vertauscht gepaart werden. Die Kombination von A und C, mit A oben, ergibt einen Graphen G nach Abb.4.8 (links).

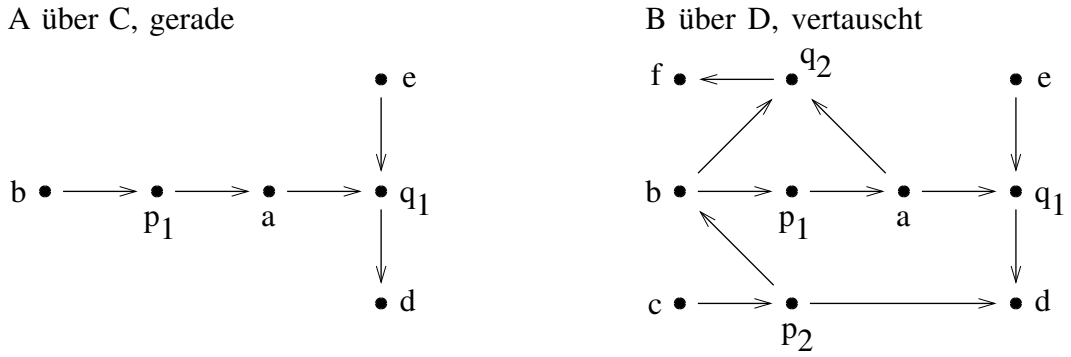


Abbildung 4.8: Graph G nach Paarung von A und C

Die nächste mögliche Paarung ist die von B und D. Wegen Term b ist keine gerade Paarung möglich, wohl aber eine ungerade. Diese zweite Paarung führt zum rechten Teil der Abb.4.8.

Weitere Paarungen sind mangels weiterer Eingangsvariablen nicht möglich. Abb.4.9 zeigt das resultierende PLA.

Der angegebene Algorithmus arbeitet vollständig ohne Backtracking und es werden nicht alle möglichen Paarungen von Variablen geprüft. Unter den ungeprüften kann sich insbesondere die optimale Lösung befinden. Es sind daher andere Algorithmen entwickelt worden, die das Finden einer optimalen Lösung zum Ziel haben. Beispiele solcher Verfahren sind die "Branch-and-Bound"-Verfahren[5] von Lewandowski und C.L.Liu [LC84] sowie von Grass [Gra82].

4.1.7 Faltung der OR–Plane

Neben der AND–Plane kann auch die OR–Plane gefaltet werden (engl. "row folding"). Dabei wird dann ausgenutzt, daß viele der Ausgangsbits nur von einem Teil der Produktterme abhängen (vgl. Abb.4.10).

Für diese Faltung können prinzipiell die gleichen Algorithmen wie bei der Faltung der AND-Plane verwendet werden, jedoch entfällt die Behandlung von negierten Variablen.

[5] Eine Einführung in die Technik der "Branch-and-Bound"-Verfahren enthält z.B. das Buch von Horowitz und Sahni [HS81].

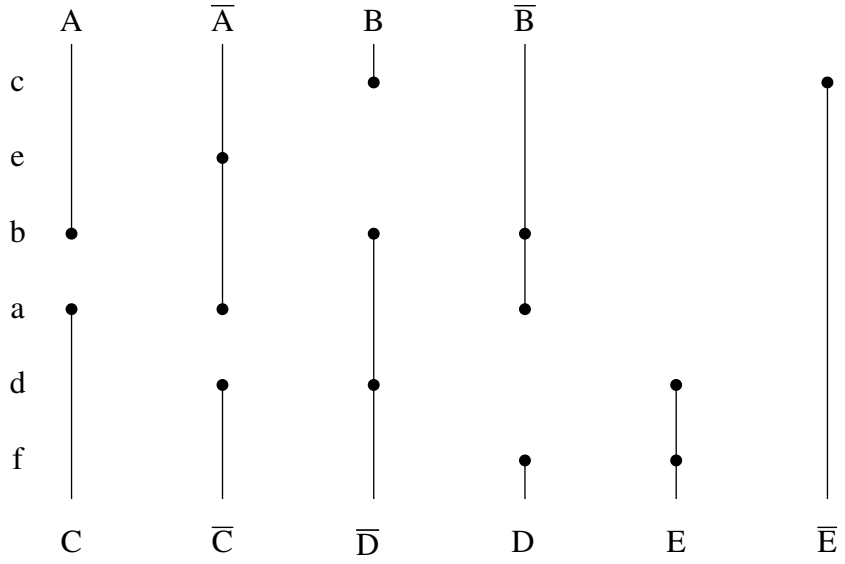


Abbildung 4.9: PLA nach Faltung der AND–Plane

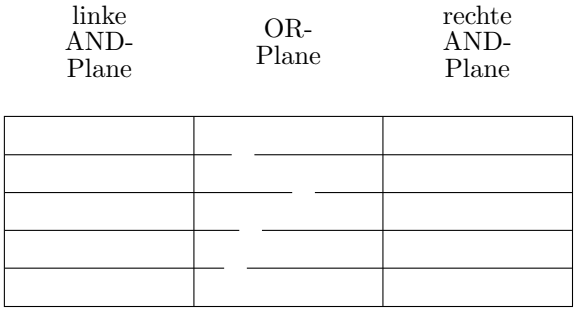


Abbildung 4.10: Faltung der OR–Plane

4.1.8 Partitionierung von PLAs

Neben der Faltung kann auch die Partitionierung eines PLAs in zwei oder mehr kleinere PLAs einen Flächengewinn bringen. Die Fläche eines PLAs mit k Zeilen, n Eingangs– und m Ausgangsvariablen ist proportional zu

$$S = k * (n + m);$$

Nach Partitionierung in zwei Teile ist die Fläche proportional zu:

$$S' = \sum_i k_i * (n_i + m_i)$$

Wenn gilt: $\forall i \in [1..2] : m_i = m/2, n_i = n/2$ und $k_i = k/2$, so wird eine Flächenreduktion um die Hälfte ($S' = S/2$) erreicht.

Neben der direkten Flächenreduktion wird vielfach eine Flächenreduktion dadurch erreicht, daß viele kleine PLAs leichter in ein Gesamtlayout zu integrieren sind als ein großes.

4.2 Klassische Optimierungstechniken für mehrstufige Schaltungen

Sofern mehr als zwei Gatterstufen zur Realisierung Boolescher Funktionen verwendet werden dürfen, wird die Berücksichtigung gemeinsamer Teilausdrücke zu einem wesentlichen Ziel. Gemeinsame Teilausdrücke werden insbesondere dann wichtig, wenn nicht nur eine einzelne Boolesche Funktion zu minimieren ist, sondern eine Menge solcher Funktionen, ein sog. **Funktionenbündel**.

Beispiel:

Zu berechnen seien die Funktionen y_1 und y_2, die wie folgt definiert sind:

$$y_1 = \overline{a}bcd \vee a\overline{b}cd \vee ab\overline{c}d \vee abc\overline{d}$$
$$y_2 = \overline{a}bcd \vee a\overline{b}cd \vee efg$$

Eine einfachere, mehrstufige Realisierung ist:

$$h_1 = cd(\overline{a}b \vee a\overline{b})$$
$$y_1 = h_1 \vee ab(\overline{c}d \vee c\overline{d})$$
$$y_2 = h_1 \vee efg$$

mit der Hilfsfunktion h_1.

Die Minimierung von Funktionenbündeln ist wie die Minimierung in n-stufiger Logik seit langem Gegenstand intensiver Forschung. Im Zusammenhang mit dem Einsatz in VLSI–Systemen treten dabei neue Kostenfunktionen in den Vordergrund. Primäre, gegeneinander abzuwägende Ziele sind jetzt die Minimierung der Chipfläche und der Verzögerungszeit. Eine Schwierigkeit ist in diesem Zusammenhang die Berücksichtigung der kapazitiven Belastung der Ausgänge in Abhängigkeit von den Leitungslängen. Dies ist nur möglich, wenn das Layout mit hinreichender Genauigkeit abgeschätzt werden kann (siehe z.B. [PB91]).

Bekannte Systeme zur n-stufigen Logiksynthese sind LSS (Logic Synthesis System) von IBM [DBG84, BT88], SOCRATES [dGC85] und MIS [BR87]. MIS-MV [LMB90] ist die Erweiterung von MIS auf mehrwertige, symbolische Logik.

Kapitel 5

Layout-Synthese

In diesem Kapitel[1] sollen die Fragen des automatisierten Layout-Entwurfs, d.h. der Platzierung der elektrischen Bauelemente innerhalb eines VLSI-Chips sowie deren Verdrahtung untereinander, behandelt werden. Aufgrund einer vorzugsweise graphentheoretischen Formulierung der Probleme ist die Mehrzahl der Verfahren auch auf andere Bereiche, wie etwa die Unternehmensplanung, anwendbar.

Als Ergänzung dieses Kapitels sind vor allem die Bücher von Ohtsuki [Oht86], Preas et al. [PL88], Brück [Brü94], Lengauer [Len90] und Sherwani [She98] sowie der Übersichtsartikel von Shahookar et al. [SM91] geeignet.

Die Erzeugung des gesamten Layouts wird aus Komplexitätsgründen in fast allen Fällen in die Phasen **Platzierung**[2] und **Verdrahtung** zerlegt, wobei die letztere meist noch in die Phasen der **globalen Verdrahtung** und der **detaillierten Verdrahtung** weiter unterteilt wird. Wir werden in diesem Kapitel auf diese insgesamt drei Phasen genauer eingehen.

5.1 Platzierung

5.1.1 Einführung

Ziel der Platzierungsphase im elektrischen Entwurf ist die Anordnung von Bausteinen auf der verfügbaren Fläche. Das Problem tritt sowohl in Form der Platzierung von integrierten Schaltkreisen auf Platinen, als auch in Form der Platzierung von Bausteinen innerhalb der Schaltkreise auf. Die Bausteine nennt man im Zusammenhang mit der Layout-Synthese meist **Zellen**. Zellen besitzen eine Anzahl von **Anschlüssen** (engl. *pin*, bei VHDL ist eine Gruppe von Anschlüssen ein **Port**, siehe Kap. 2). Über diese Anschlüsse können Zellen mit anderen Zellen leitend verbunden werden. Dies geschieht über **Netze**.

Unter einem **Netz** versteht man einen elektrisch leitend verbundenen Bereich, also alle miteinander verbundenen Anschlüsse, sowie die Verbindungen dazwischen. Sind n Anschlüsse miteinander verbunden, spricht man von **n–Punkt–Netzen**.

Die Eingabe der Platzierungsphase besteht aus einer Liste von Zellen (einschließlich der Zell-Abmessungen). Bei einigen Entwurfsstilen sind praktisch alle Bausteine gleich groß (Standard 16–Pin Gehäuse, Gate Arrays), bei anderen Entwurfsstilen schwankt die Größe aber beträchtlich.

Weiterhin ist eine Liste von Verbindungen zwischen den Anschlüssen gegeben. Diese wird allgemein als **Netzliste** bezeichnet.

Die Netze bilden zusammen mit den Zellen als Knoten den **Netz-Graphen** [3].

Def. 1: Ein Graph ist ein Paar (V, E) mit:

[1] Dieses Kapitel basiert auf dem entsprechenden Kapitel in dem Buch *P. Marwedel: Synthese und Simulation von VLSI-Systemen, Hanser, 1993* (vergriffen). Copyright des Originals: Hanser-Verlag, 1993.

[2] Nach alter Rechtschreibung: Plazierung.

[3] Streng genommen ist dies kein gewöhnlicher Graph, da zwei Zellen über mehrere Netze miteinander verbunden sein können. Um diese Tatsache zu modellieren, kann man bipartite Graphen (mit Netzen als zweitem Knotentyp) oder Hypergraphen (mit Knoten, die selbst wieder Graphen sind) verwenden. Dies braucht im Folgenden aber nicht weiter berücksichtigt zu werden.

V ist die endliche, nichtleere Menge der **Knoten** (engl. *vertices*) und
$E \subseteq V \times V$ ist die Menge der **Kanten** (engl. *edges*).

Mit der Platzierung sollte eigentlich die Verdrahtung einhergehen, aber wegen der Komplexität werden beide meist sequentiell ausgeführt. Dies kann dazu führen, dass für eine vorgegebene Platzierung keine Verdrahtung existiert.

Im Prinzip sind bei der Platzierung viele Randbedingungen zu beachten: es darf kein Übersprechen zwischen Leitungen auftreten, die maximale Verlustleistung pro Fläche muß klein genug bleiben, die nachfolgende Verdrahtung muß möglich sein usw. Statt der expliziten Berücksichtigung aller Randbedingungen definiert man meist eine Zielfunktion, die in gewissem Umfang auch Randbedingungen indirekt berücksichtigt. Z.B. nimmt man an, dass Algorithmen, die die gesamte Verdrahtungslänge minimieren, auch die Verdrahtunglänge einzelner Netze klein halten. Da dies aber natürlich nicht garantiert werden kann, sollte nach dem Entwurf des Layouts eine **Extraktion der elektrischen Parameter** (z.B. der resultierenden Kapazitäten) erfolgen. Durch eine Simulation oder durch Benutzung eines *timing–verifiers* kann man die Funktion der Schaltung dann noch einmal überprüfen. Neuerdings existieren auch kommerzielle Werkzeuge, mit denen die thermische Erhitzung nach Erzeugung des Layouts überprüft werden kann.

Praktisch anwendbar sind drei Zielfunktionen:

a) Gesamte Verdrahtungslänge → Min,

b) Maximum der von Schnittlinien durchschnittenen Netze → Min; man betrachtet die Anzahl der Netze, die von einer senkrecht dazu geführten Linie geschnitten werden (siehe Abb. 5.1).

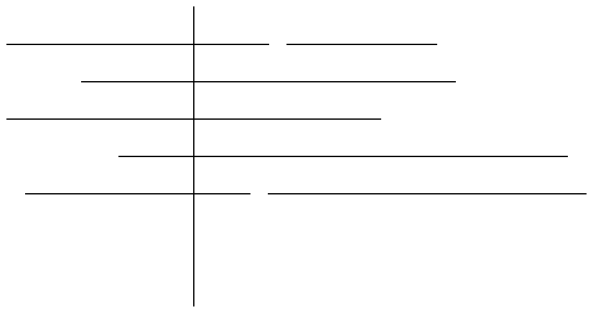


Abbildung 5.1: Schnitt von Netzen

Als Ziel der Optimierung versucht man nun, das Maximum der von Schnittlinien durchschnittenen Netze zu minimieren.

c) Maximale Dichte von Leitungen pro Fläche → Min.

5.1.2 Platzierung nach dem Kräftemodell

Frühe Veröffentlichungen über Platzierungsverfahren basierten häufig auf dem Kräftemodell idealer Federn (siehe z.B. [HK72]). Man betrachte dazu Abb. 5.2.

Zwischen allen Zellen, die über Netze miteinander verbunden sind, stelle man sich Federn vor. Die Kraft zwischen zwei Zellen i und j ist dann:

$$\begin{aligned} F_{i,j} &= -D * x_{i,j}, \text{ mit} \\ D &= \text{Anzahl der Netze zwischen } i \text{ und } j, \text{ sowie} \\ x_{i,j} &= \text{vorzeichenbehafteter Abstand zwischen } i \text{ und } j \\ &\quad \text{(im 1-dimens. Fall Differenz der Koordinaten)} \end{aligned}$$

Eine Zelle i wird sich dann so bewegen, dass die resultierende Kraft $F_i = \sum_j F_{i,j}$ zu Null wird. Dadurch werden die Verbindungen einer Zelle zu den übrigen Zellen berücksichtigt. Man stelle sich vor, dass die

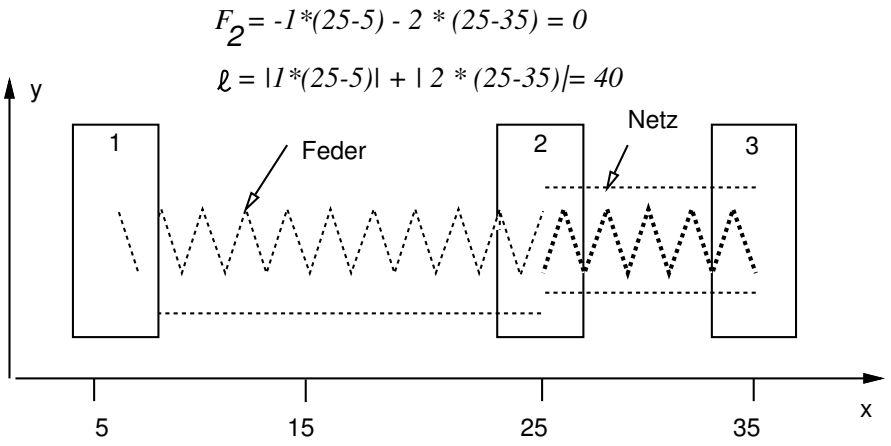


Abbildung 5.2: Platzierung nach dem Kräftemodell

Platzierung der Zellen 1 und 3 fest sei und eine Platzierung der Zelle 2 mit minimalen Kräften gesucht sei. Die Platzierung der Zelle 2 an der angegebenen Stelle läßt die resultierende Kraft auf diese Zelle zu Null werden. Dabei sei angenommen, dass die Netze bis zur Mitte der Zellen zu führen sind und dass auch die Federn in der Mitte der Zellen angebracht sind.

Probleme bereitet dieses Modell aus den folgenden Gründen:

- Algorithmen nach diesem Modell tendieren dazu, alle Zellen demselben Platz zuzuorden, denn eine solche Zuordnung würde in der Tat alle Kräfte zu Null werden lassen. Beispielsweise durch Vorgabe der Position einiger Zellen (im obigen Beispiel der Positionen von 1 und 3) versucht man, dies zu verhindern. Es bleibt dennoch eine gewisse Tendenz, alle Zellen möglichst in der Mitte der verfügbaren Fläche zu platzieren.
- Eine Platzierung mit minimalen resultierenden Kräften ist nicht unbedingt auch eine Platzierung mit minimaler Verdrahtungslänge ℓ. Dies zeigt die Platzierung in Abb. 5.3. Die Verdrahtungslänge ist hier gegenüber der Abb. 5.2 reduziert.

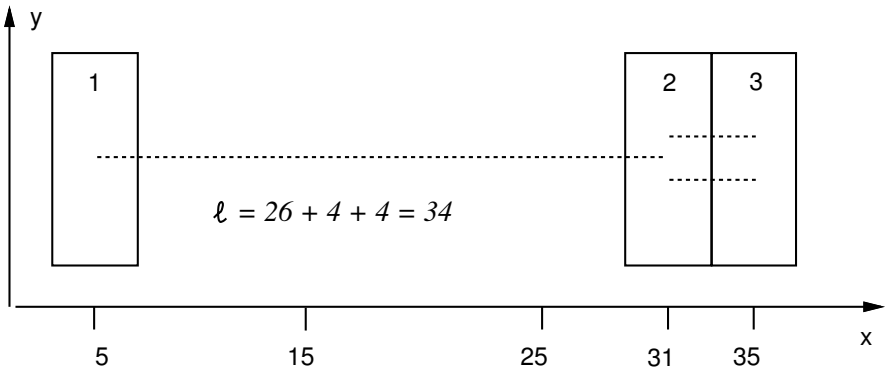


Abbildung 5.3: Platzierung mit minimaler Verdrahtungslänge

5.1.3 Modellierung als Quadratisches Zuordnungsproblem

Eine explizite Minimierung der Verdrahtungslänge wird bei der Modellierung der Platzierung als Quadratisches Zuordnungsproblem angestrebt. Dieses Modell geht zunächst von 2–Punkt–Netzen aus. Bei 2–Punkt–Netzen gilt für die Gesamtlänge T der Verdrahtung als Funktion der Platzierung p:

$$T(p) = \sum_{i,j \in M; i<j} w_{i,j} * d_{p(i),p(j)}$$

Darin bedeuten:

M	:	die Menge der durchnummerierten Zellen;
$w_{i,j}$	:	Zahl der den Zellen i und j gemeinsamen Netze;
p	:	Platzierungsfunktion, die allen Zellen einen Platz zuordnet mit der Bedingung: $\forall i,j \in M, i \neq j : p(i) \neq p(j)$;
$d_{k,l}$	:	Abstand der Plätze k und l voneinander; die verfügbaren Plätze sind zum Zweck der Indizierung ebenfalls mit natürlichen Zahlen durchzunummerieren.

Abb. 5.4 zeigt ein Beispiel für eine Platzierung von drei Zellen *1, 2* und *3* auf Plätzen im Bereich von `1` bis `15`.

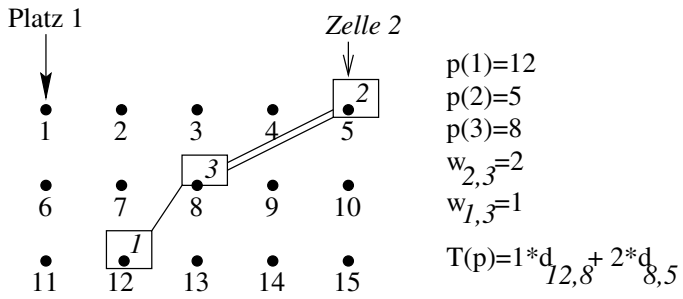


Abbildung 5.4: Beispiel für eine Platzierung

In die Länge T der Verdrahtung gehen dabei die Abstände $d_{12,8}$ und $d_{5,8}$ der den Zellen zugeordneten Plätze ein.

Das Ziel der Platzierung besteht darin, die Funktion p so zu bestimmen, dass die Verdrahtungslänge $T(p)$ minimal wird. Die Zuordnung mit der angegebenen Zielfunktion ist auch als **Quadratisches Zuordnungsproblem** (engl. *quadratic assignment problem*, QAP)[4] bekannt. Der Name rührt daher, dass wir mit:

$$x_{i,k} := \begin{cases} 1, & \text{falls} \quad p(i) = k \quad \text{ist} \\ 0 & \text{sonst} \end{cases}$$

$T(p)$ umschreiben können zu:

$$T(p) = \sum_{i,j,k,l;i<j} w_{i,j} * d_{k,l} * x_{i,k} * x_{j,l}$$

Die **Entscheidungsvariablen** $x_{.,.}$ treten in dieser Formel quadratisch auf.

Das Quadratische Zuordnungsproblem tritt auch in vielen anderen Anwendungen auf. Beispiel: Gegeben sei eine Menge M von Maschinen, die jeweils genau eine Aufgabe bearbeiten. Zwischen den Maschinen sei ein Fluß (Zahl zu transportierender Güter) $w_{i,j}$ erforderlich. Wenn dann die Maschine i dem Platz $p(i)$ und die Maschine j dem Platz $p(j)$ mit dem Abstand $d_{p(i),p(j)}$ zugeordnet wird, mögen Transportkosten $w_{i,j} * d_{p(i),p(j)}$ zwischen den beiden Maschinen anfallen. Die Minimierung der gesamten Transportkosten durch geeignete Wahl der Funktion p führt wieder auf das QAP–Problem.

Zur optimalen Lösung des QAP–Problems werden in der Regel Branch–and–Bound–Verfahren[5] verwandt.

Für das QAP sind im Gegensatz zum allgemeinen Platzierungsproblem aber gute heuristische Verfahren bekannt. Daher wird das allgemeine Platzierungsproblem häufig durch das QAP–Problem ersetzt [HK72].

Bei der Erweiterung des QAP auf n-Punkt-Netze mit $n \geq 3$ tritt das folgende Problem auf: bei der Berechnung der Verdrahtungslänge sind jetzt nur noch Terme für Zellen zu berücksichtigen, die **direkt** miteinander verbunden sind. Man betrachte dazu das Beispiel eines 3-Punkt-Netzes in Abb. 5.5.

In diesem Beispiel entfällt der Term für die Verbindung zwischen den Zellen 1 und 3. Welche Terme zu berücksichtigen sind, hängt aber vom Verdrahtungsbaum und damit von der Platzierung ab. Damit sind die

[4] Veröffentlichungen hierzu gibt es v.a. von Burkard (siehe z.B. [Bur84]).
[5] Zur Darstellung allgemeiner Branch–and–Bound–Verfahren siehe z.B. [AHU74].

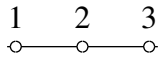


Abbildung 5.5: 3-Punkt-Netz

$w_{i,j}$ nicht mehr allein aufgrund der Kenntnis der Netze zu bestimmen, vielmehr hängen diese Koeffizienten von der Funktion p ab.

Dazu definiert man zunächst wieder $w_{i,j}$ als die Zahl der den Zellen i und j gemeinsamen Netze. Damit rechnet man so, als wären alle Zellen eines Netzes **direkt** miteinander zu verbinden, d.h. man rechnet statt mit Verdrahtungs**bäumen** mit **vollständigen Graphen** (Graphen, in denen alle Knoten direkt miteinander verbunden sind). Damit zumindest die Summe der $w_{i,j}$ für 2- und 3-Punkt-Netze den richtigen Wert ergibt, berücksichtigt man ein n–Punkt–Netz statt mit 1 mit $2/n$. Auf diese Weise wird für 3-Punkt-Netze kompensiert, dass ein Verdrahtungsbaum 2 Kanten hätte, der vollständige Graph aber 3.

5.1.4 Platzierung mittels Partitionierung

5.1.4.1 Begriffe

Eine vielbenutzte Methode der Platzierung besteht in der fortgesetzten Aufteilung der Menge der Zellen in Teilmengen und der Zuordnung der Teilmengen zu Teilflächen der verfügbaren Fläche. Eine solche Zerlegung von Mengen heißt **Partitionierung**. Formal läßt sich die Partitionierung wie folgt definieren:

Gegeben sei ein Graph G mit einer **Knoten**menge V und einer **Kanten**menge E, mit $E \subseteq V \times V$. Knoten und Kanten seien gewichtet, d.h. es existieren Abbildungen k und w mit

$$k : V \rightarrow I\!R^+$$

$$w : E \rightarrow I\!R^+$$

Def. 2: Die Zerlegung der Knotenmenge V in **Blöcke** P_i heißt Partitionierung des Graphen $G \Leftrightarrow$

1. $\forall i : P_i \in \wp(V) \qquad \wedge$
2. $\forall i, j \quad \text{mit} \quad i \neq j : P_i \cap P_j = \emptyset \qquad \wedge$
3. $\cup_i P_i = V.$

Der Spezialfall der Zerlegung in zwei Blöcke $A = P_1, B = P_2$ heißt Bipartitionierung oder **Bisektion**.

Def. 3: Die **externen Kosten** E_i eines Knotens i in einem Block P_j sind definiert durch $E_i = \sum_{l \notin P_j} w_{i,l}$

Def. 3: Die **internen Kosten** I_i eines Knotens i in einem Block P_j sind definiert durch $I_i = \sum_{l \in P_j} w_{i,l}$

Ziel der Partitionierung ist in der Regel die Minimierung der gesamten externen Kosten:

$$\sum_j E_j \rightarrow \quad \text{Min.}$$

Dabei sind meist gewisse Beschränkungen einzuhalten. Dazu gehören Beschränkungen der externen Kosten durch eine Konstante E_{max} und/oder der Größe der Blöcke durch eine Konstante K_{max}:

$$\forall i : E_i \leq E_{max};$$

$$\forall j : \sum_{i \in P_j} k_i \leq K_{max};$$

Für die Bipartitionierung wird meist die Einhaltung eines Balancekriteriums gefordert. Dieses soll bewirken, dass beide Blöcke in etwa gleich groß sind. Ohne dieses Kriterium würde man alle Knoten einem Block zuordnen, da dies sicherlich die Anzahl der geschnittenen Netze, d.h. die externen Kosten minimieren würde.

Das allgemeine Partitionierungsproblem ist NP–hart [1]. Zur Partitionierung werden daher üblicherweise Heuristiken angewandt.

Man unterscheidet dabei zwischen den **konstruktiven Verfahren** und den **iterativen Verfahren**. Letztere setzen voraus, dass eine Anfangspartition vorhanden ist. In einem iterativen Prozeß versuchen sie, diese zu verbessern. Erstere liefern eine solche Anfangspartition.

5.1.4.2 Konstruktive Partitionierung

Der folgende Algorithmus kennzeichnet die Grundidee der meisten konstruktiven Verfahren:

```
H := V; F := ∅;
WHILE H ≠ ∅ DO
  BEGIN
    Sei Q der Knoten aus H, für den (Zahl der Kanten zwischen Q und F)
    - (Zahl der Kanten zwischen Q und H) maximal ist.
    Ordne Q dem Block P_i zu, der die maximale Zahl von Verbindungen
    zu Q besitzt. Zusätzliche Beschränkungen sind hierbei zu beachten.
    H := H − {Q}; F := F + {Q}
  END
```

Algorithmus 4.1: Grundidee der konstruktiven Partitionierung

Beispiel:

Gegeben sei der Graph nach Abb. 5.6 (links). Eine Beschränkung möge bewirken, dass etwa die Hälfte aller Zellen (Knoten) einem Block zugeordnet werden.

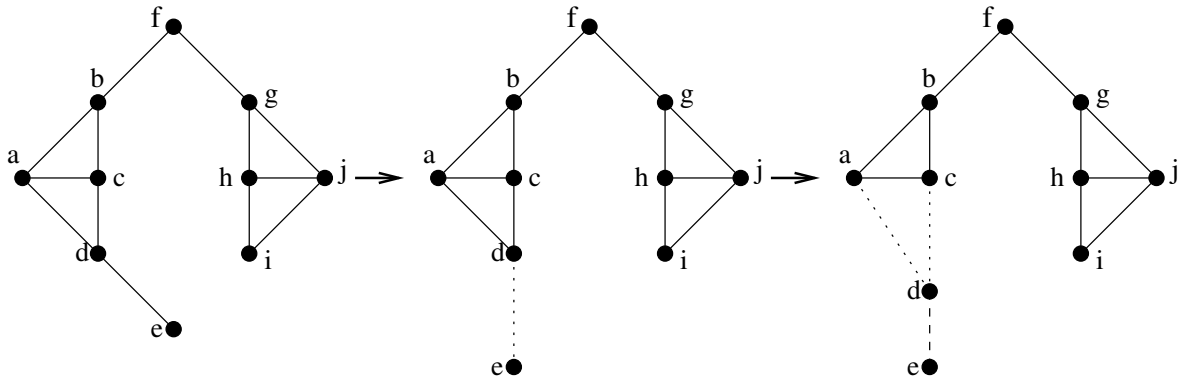


Abbildung 5.6: Interatives Partitionieren eines Graphen

Zunächst wird der Knoten e abgespalten, da für ihn die Anzahl der Kanten zwischen Q und F gleich Null, die Anzahl der Kanten zwischen Q und H gleich 1 und die sich ergebende Differenz (-1) maximal ist. Die Abbildungen von Abb. 5.6 (mitte) bis Abb. 5.7 (rechts) zeigen, wie der Graph iterativ partitioniert wird.

Die Ergebnisse konstruktiver Verfahren sind häufig verbesserungsfähig, da einmal getroffene Zuordnungen bei den meisten Verfahren nicht wieder verworfen werden. Die Güte der Lösung scheint bei den verschiedenen Varianten nicht wesentlich unterschiedlich zu sein [Oht86].

5.1.4.3 Verfahren von Kernighan und Lin

Der wohl bekannteste iterative Bisektions–Algorithmus ist der Algorithmus von Kernighan und Lin [KL70]. Trotz seines Alters wird er in der Praxis noch vielfach angewandt, neben der Platzierungsphase z.B. auch in der Logiksynthese.

[1] NP-hart bedeutet: "mindestens so schwer wie das Erfüllbarkeitsproblem (SAT)"; der häufig benutzte Begriff "NP–vollständig" läßt sich streng genommen nur auf Entscheidungsprobleme anwenden, siehe dazu z.B. [HS76].

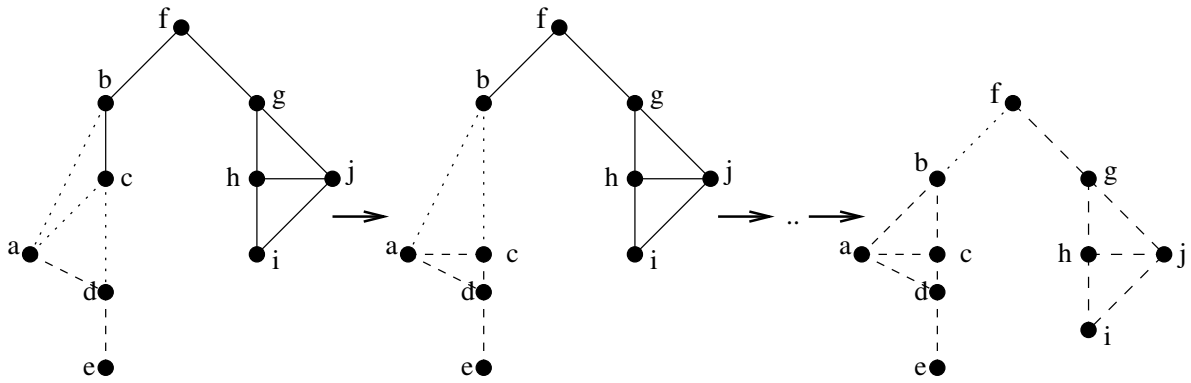


Abbildung 5.7: Interatives Partitionieren eines Graphen

Der Algorithmus setzt voraus, dass eine Startpartition mit $|A| = |B| = n$ gegeben ist. Eine gerade Zahl von Knoten kann durch Einfügen von Dummy–Knoten immer erreicht werden. Das Verfahren basiert auf einer paarweisen Vertauschung von Knoten aus A und B. Einfachere Verfahren beenden das Vertauschen, sofern keine Knoten mehr existieren, deren Vertauschung einen Gewinn (d.h. eine Reduktion der Zahl der geschnittenen Netze) erbringt. Der Kernighan–Lin–Algorithmus dagegen vertauscht probehalber auch dann noch weiter. Stellt sich danach noch einmal wieder ein hoher positiver Gesamtgewinn ein, so wird auch die Vertauschung mit negativen Gewinn akzeptiert. Dadurch führt der Algorithmus auch dann Vertauschungen durch, wenn nur die Vertauschung von mehreren Knoten insgesamt einen Nutzen bringt.

In einer äußeren Schleife wird der soeben beschriebene Prozeß wiederholt, solange der Gesamtgewinn positiv ist. Das unmittelbare Beenden der äußeren Schleife bei negativem Gewinn wird als sog. **Greedy–Verhalten** bezeichnet. Die innere Schleife dagegen besitzt kein Greedy–Verhalten, da der Algorithmus hier bei einem lokalen Minimum nicht sofort abbricht.

```
REPEAT Max := 0; G := 0;
  A';=A; B':=B;
  FOR i:=1 TO n DO
    BEGIN
      Wähle a_i ∈ A', b_i ∈ B' so, dass das Vertauschen von a und b
      den Gewinn g maximal werden läßt.
      A' := A' − {a_i}; B' := B' − {b_i};
      g_i:=Gewinn durch das Vertauschen von a_i und b_i.
      G := G + g_i;
      IF G > Max THEN BEGIN k := i; Max := G END;
    END;
  IF Max > 0 THEN
    BEGIN
      A := A − {a_1..a_k} ∪ {b_1..b_k};
      B := B − {b_1..b_k} ∪ {a_1..a_k};
    END;
UNTIL Max ≤ 0;
```

Algorithmus 4.2: Verfahren von Kernighan und Lin

Abb. 5.8 zeigt ein Beispiel eines partitionierten Graphen, bei dem eine Reduktion der Anzahl der geschnittenen Kanten von 2 auf 1 nur über eine momentane Erhöhung der Anzahl der geschnittenen Kanten möglich ist.

5.1.4.4 Linearer Algorithmus von Fiduccia und Mattheyses

Die Komplexität des Kernighan–Lin–Verfahrens in der eben angegebenen Form bleibt zunächst unklar. Für die Auswahl von a_i und b_i müßten ggf. alle Knotenpaare aus A und B betrachtet werden. Schon ohne die Gewinnberechnung würde sich ein quadratischer Effekt ergeben. Dieser Effekt wäre unerwünscht, da man Graphen mit einigen Hundert Knoten noch in vertretbarer Zeit partitionieren möchte. Fiduccia und Mattheyses [FM82] haben Datenstrukturen entwickelt, die ein insgesamt lineares Laufzeitverhalten

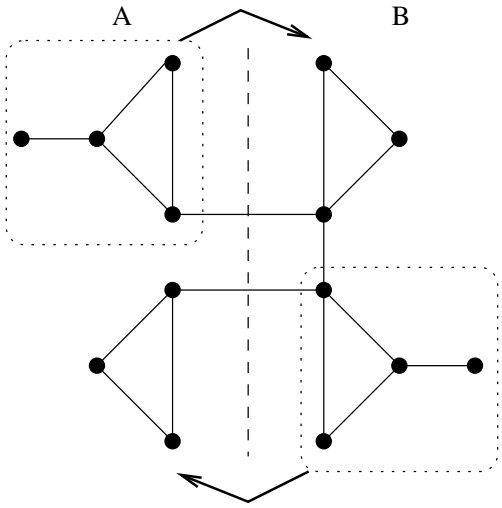


Abbildung 5.8: Graph, welcher ein nicht-lokales Verhalten der Partitionierung erfordert

bewirken. Statt einer paarweisen Vertauschung betrachten Fiduccia et al. nur noch die Bewegung eines Knotens in einen anderen Block. Das Gewicht aller Kanten wird mit 1 angenommen, so dass nur noch die **Anzahl** der geschnittenen Netze berücksichtigt wird. Eine der wesentlichen Ideen von Fiduccia et al. ist die sorgfältige Buchführung über die durch Vertauschung möglichen Gewinne. Dadurch ist es möglich, auf der Basis typischer Anwendungen eine Gesamtkomplexität von $O(P = \sum p_i)$ (mit p_i = Zahl der Pins der Zelle i) zu erreichen.

5.1.4.5 Algorithmus von Breuer

Nachdem wir jetzt wissen, wie wir effizient eine Zerlegung der Zellen in zwei Mengen durchführen können, wollen wir jetzt darstellen, wie man durch mehrfache Zerlegung zu einer Platzierung der Zellen kommt. Algorithmen, welche aufgrund solcher Zerlegungen platzieren, heißen **Min–Cut–Algorithmen**.

Sei C eine Menge von Schnittlinien durch die Platzierungsfläche und sei $v(c)$ für $c \in C$ eine Funktion, die einer Schnittlinie die Zahl der geschnittenen Netze zuordnet. Ein ideales Min–Cut–Verfahren würde das Maximum der Zahl der geschnittenen Netze minimieren:

$$\max_{c \in C}(v(c)) \rightarrow \min$$

Da ein solches Verfahren zu komplex wäre, wird zunächst nur eine grobe Platzierung vorgenommen, die für den ersten Partitionierungsschritt die Anzahl der geschnittenen Netze minimiert. Eine genauere Platzierung wird dann so bestimmt, dass die Anzahl der geschnittenen Netze im zweiten Partitionierungsschritt unter Beibehaltung der Entscheidungen des ersten Partitionierungsschritts minimal wird. Weitere Partitionierungsschritte folgen bis alle Zellen genau platziert sind oder bis ein anderes Abbruchkriterium erfüllt ist (vgl. Abb. 5.9).

Def. 4: Im Folgenden sei ein **Block** eine Menge von Zellen.

Die meisten Min–Cut–Verfahren gehen zurück auf den Algorithmus von Breuer[6] [Bre77]:

1. Der Block $g_j = g_1$ enthalte zunächst alle Zellen.
2. Wähle eine Reihenfolge der Bearbeitung der Schnittlinien C.
3. Wähle die nächste Schnittlinie $c_i \in C$.
4. Platziere die Zellen aus g_j auf beiden Seiten der Schnittlinie c_i so, dass $v(c_i)$ unter Einhaltung eines Balancekriteriums minimal wird. Bilde aus den Zellen jeder Seite je einen neuen Block.

[6] Gesprochen: Bru–er.

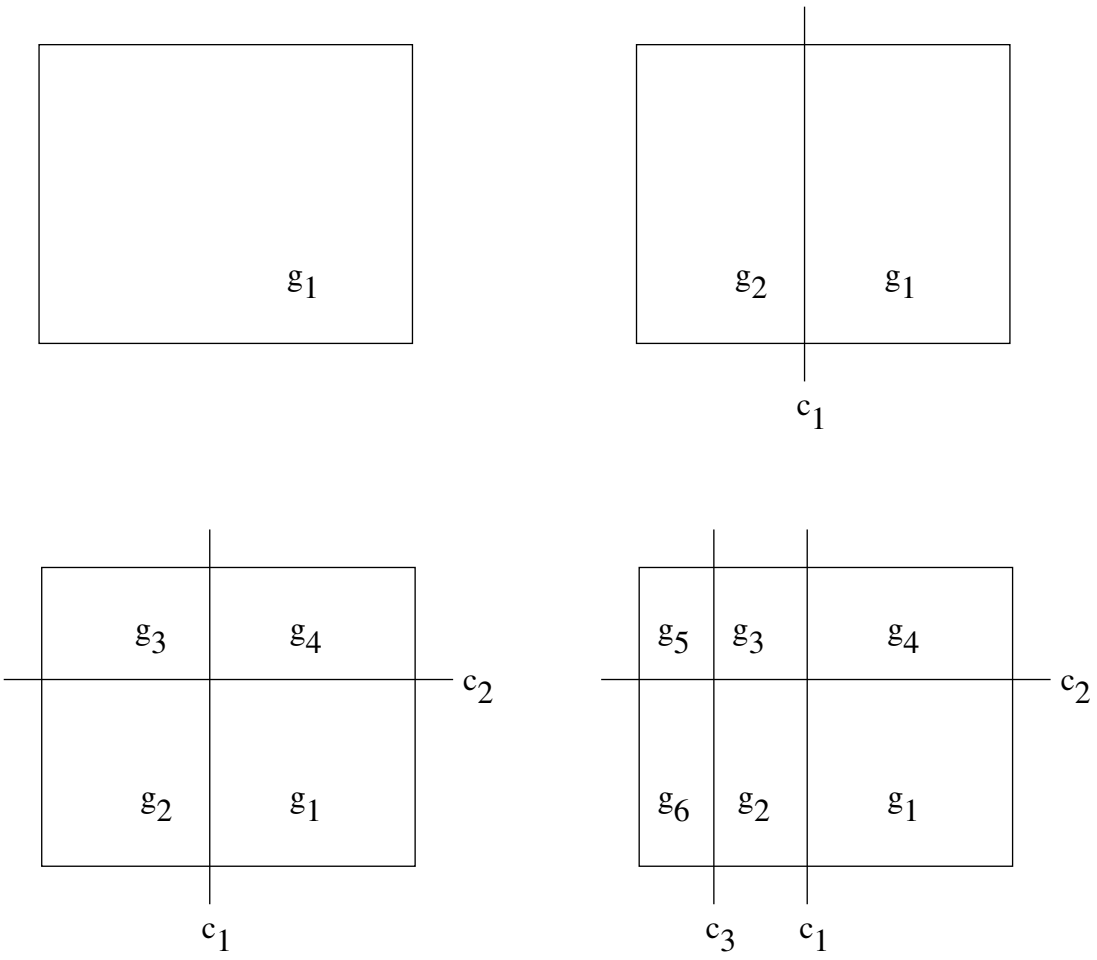


Abbildung 5.9: Min-Cut-Verfahren nach Breuer

5. Für jeden neuen Block g_j wiederhole das Verfahren ab 3. solange das gewählte Abbruchkriterium nicht erfüllt ist.

Mit diesem Verfahren werden die Zellen rekursiv immer genauer platziert. Als Teilprobleme entstehen dabei:

a) Die Partitionierung eines Graphen in zwei Teilgraphen mit minimalem Schnitt. Derartige Bisektionsverfahren wurden in Abschnitt 3.1 behandelt.

b) Die Selektion der Schnittlinien. Breuer schlägt zwei Verfahren vor, nämlich

   b1) abwechselnd horizontale und vertikale Schnitte;

   b2) zunächst "dünne" Scheiben in einer Richtung, dann nach Art der binären Suche in der dazu senkrechten Richtung.

Da das Verfahren von Breuer die Fläche immer in der gesamten Länge schneidet, ist es ungeeignet, wenn Zellen unterschiedlicher Größe zu platzieren sind. Für Zellen gleicher Größe, wie z.B. die Platzierung gleichgroßer ICs auf einer Platine oder für Gate–Arrays ist das Verfahren dagegen durchaus anwendbar. Um auch Zellen unterschiedlicher Größe bearbeiten zu können, müssen die Datenstrukturen um Abmessungen der Zellen und deren relative Lage zueinander erweitert werden.

5.1.4.6 Min-Cut–Verfahren für beliebige Zellen

Das grundlegende Verfahren zur Min–Cut–Platzierung beliebiger Zellen stammt von Lauther [Lau79]. In diesem Verfahren wird eine Fläche durch je eine Kante zweier einander zugeordneter Graphen dargestellt.

Die Kanten e_x^i eines der Graphen enthalten die Abmessungen in x–Richtung, die Kanten e_y^i des anderen die der y–Richtung. Die Zuordnung der Kanten beider Graphen zueinander durch ausgedrückt, dass sich die beiden Kanten schneiden (man sagt, sie **inzidieren**).

Die gesamte zur Verfügung stehende Fläche wird entsprechend durch zwei Graphen mit inzidierenden Kanten dargestellt. In Abb. 5.10 ist einer der Graphen mit gestrichelten, der andere mit einer durchgezogenen Kante dargestellt.

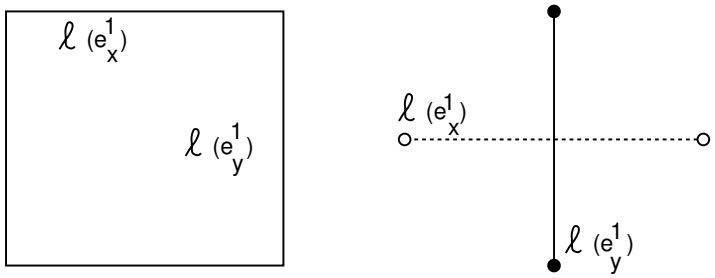


Abbildung 5.10: Darstellung einer einzelnen Fläche durch 2 Graphen

Zur Darstellung von n Flächen werden $2 * n$ Kanten benötigt. Die relative Lage der Flächen zueinander wird durch eine partielle Ordnung der Kanten ausgedrückt. In Abbildung 5.11 liegen die Flächen 1 und 2 untereinander, folglich sind auch die Kanten e_y^1 und e_y^2 entsprechend geordnet.

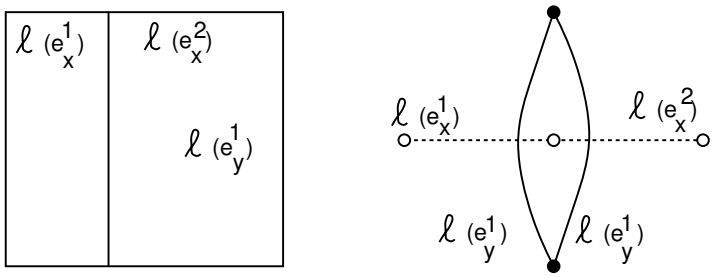


Abbildung 5.11: Darstellung der relativen Lage zweier Flächen

Die entstehenden Graphen enthalten stets je zwei ausgezeichnete Knoten, die die vier Begrenzungslinien darstellen. Aufgrund ihres Aussehens heißen diese Graphen **Polargraphen**.

Der Algorithmus von Lauther verläuft nun wie folgt:

1. Die insgesamt benötigte Fläche wird zunächst durch die Summe der Flächen der Zellen angenähert. Es wird eine quadratische Fläche angenommen:

   $$\ell(e_x^1) = \ell(e_y^1) = \sqrt{(\text{Summe der Zellflächen})}$$

   Daraus ergibt sich ein initialer Graph nach Abb. 5.10.

2. Nunmehr wird der Netzgraph (zur Definition siehe Seite 57) mittels Bisektion geschnitten. Hierbei ist wieder ein Balancekriterium zu beachten. Danach wird die Platzierungsfläche so geschnitten, dass die Teilflächen so groß sind wie die Summe der Flächen der darin enthaltenen Zellen. Bei vertikalem Schnitt ist also

   $$\ell(e_x^1) = (\text{Summe der Zellflächen aus Teilgraph 1})/\ell(e_y^1)$$

   $$\ell(e_x^2) = (\text{Summe der Zellflächen aus Teilgraph 2})/\ell(e_y^2)$$

   Es ergibt sich daraus eine Aufteilung wie in Abbildung 5.11.

3. Für jeden der resultierenden Teilgraphen wird der Schritt 2 wiederholt, und zwar abwechselnd mit horizontalem und vertikalem Schnitt.

Als Ergebnis könnte beipielsweise bei zweimaligem Schneiden die Abb. 5.11 und bei mehrfachem Schneiden die Abb. 5.12 entstehen[7]. Man beachte die unterschiedliche Größe der Flächen.

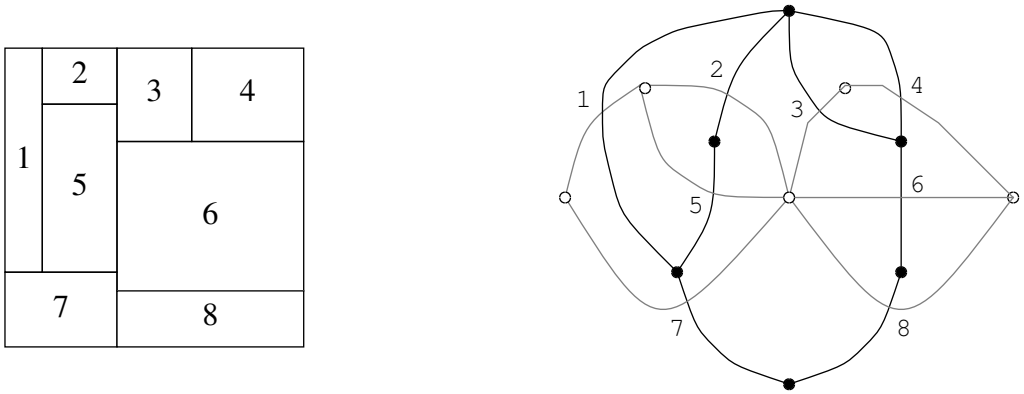


Abbildung 5.12: Platzierung mit unterschiedlich großen Zellen

Das Min–Cut–Verfahren liefert nur Lösungen, die durch fortgesetzte Flächenaufteilungen entstehen. Solche Flächenaufteilungen kann man auch durch sog. *slicing–trees* darstellen. Die Knoten enthalten dabei jeweils die Information, ob vertikal oder horizontal zu schneiden ist. Ferner kann man verabreden, dass der linke Teilbaum jeweils die linke bzw. die obere Fläche beschreibt. Abb. 5.13 a) gibt den Slicing–tree der Flächenaufteilung in Abb. 5.12 wider.

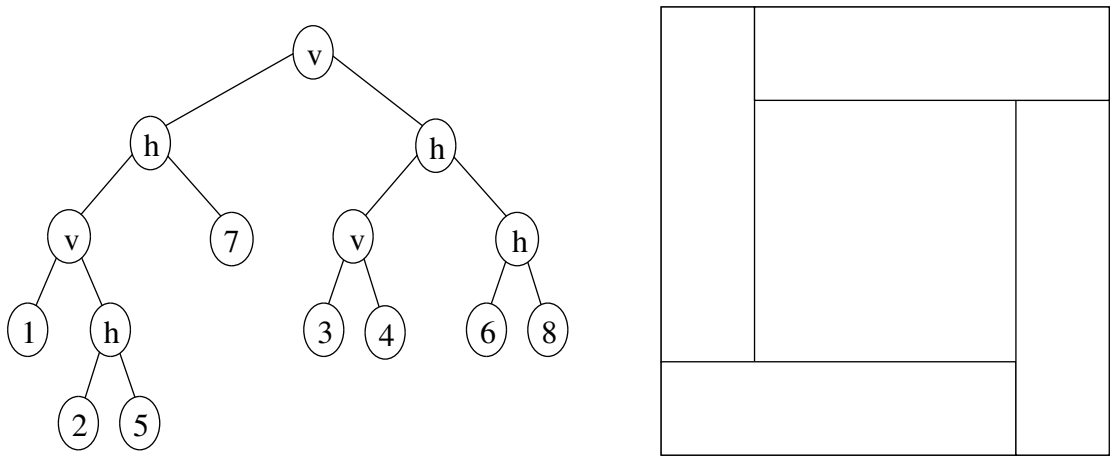


Abbildung 5.13: Slicing-Tree (a) und Pin-Wheel (b)

Slicing–trees sind keine allgemeine Darstellung von Flächenaufteilungen, da sie sog. *pin–wheels* nach Abb. 5.13 b) nicht beschreiben können.

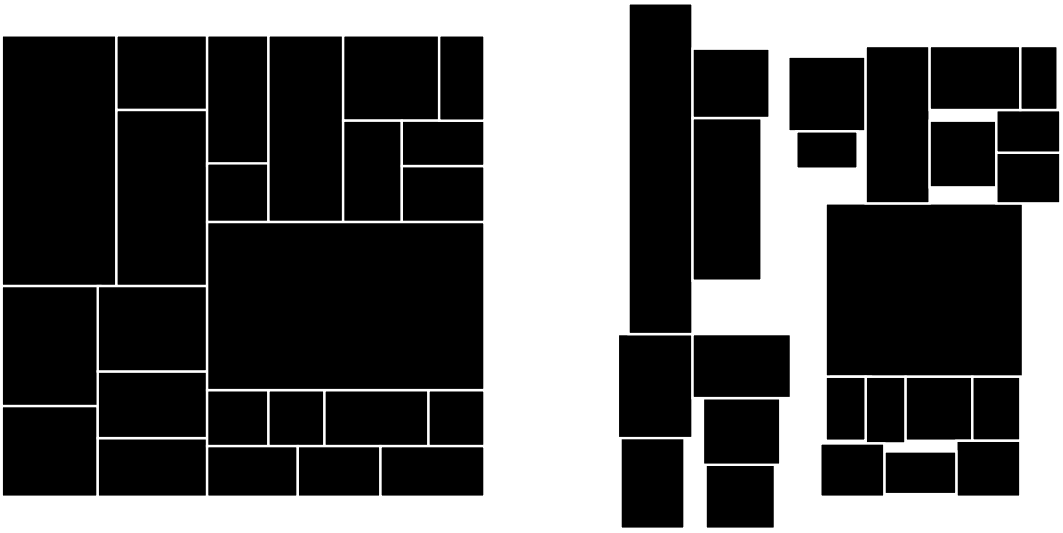
Die Teilflächen der Abb. 5.12 besitzen aufgrund der Konstruktion genau die Fläche der zu platzierenden Zellen. Das Breiten/Längenverhältnis (engl. *aspect ratio*) wird jedoch noch nicht richtig wiedergegeben. Das Gleiche gilt für das Beispiel in Abb. 5.14 a).

Als erste Maßnahme zur korrekten Berücksichtigung der Zellabmessungen erfolgt nunmehr eine **Orientierung**: die längste Seite der Zellen wird parallel zur längsten Seite der ihr zugeteilten Teilfläche gelegt. Anschließend werden die Abmessungen der Teilflächen auf diejenigen der Zellen gesetzt. Dadurch entstehen jetzt Leerflächen (vgl. Abb. 5.14 b)).

Die absolute Lage kann jetzt einfach mittels der Polargraphen berechnet werden: die Koordinaten der linken unteren Ecke einer Zelle i ergeben sich als Länge des längsten Weges vom linken bzw. unteren Polknoten bis zur Kante e_x^i bzw. e_y^i. Man vergleiche dazu Abb. 5.14 b).

Verbesserungen

[7] Um Abb. 5.12 zu erhalten, wurde zweimal nacheinander horizontal geschnitten.

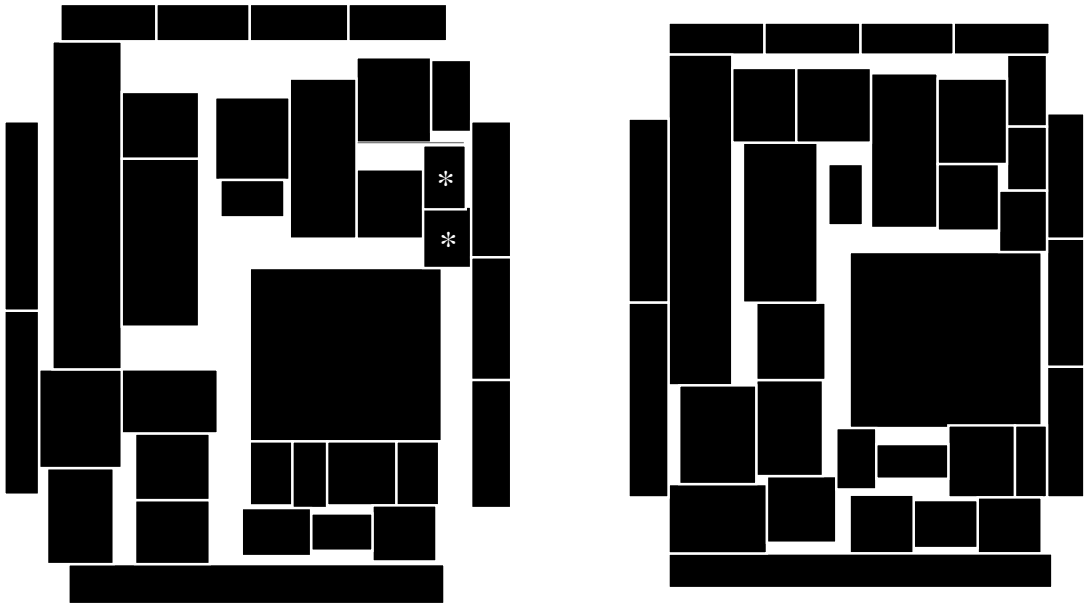


a) Initiales Placement b) Berücksichtigung der Zellabmessungen

Abbildung 5.14: Verfahren von Lauther (©1979 IEEE)

Es gibt eine Reihe von Methoden, die Leerflächen zu reduzieren:

a) Eine Vertauschung von $\ell(e_x^i)$ und $\ell(e_y^i)$ (**Rotation**) kann zu einer Flächenreduktion führen. Aufgrund der vorangegangenen Orientierung ist dies aber selten der Fall. Abb. 5.15 a) zeigt ein Beispiel für einen solchen Fall. Die mit einem Stern gekennzeichneten Zellen wurden gedreht. Damit wurde die Ausdehnung in x-Richtung reduziert. Man beachte, dass die Abbildung zusätzlich am Rand Zellen für die äußeren Anschlüsse enthält.



a) Nach Rotation (E/A-Pins eingefügt) b) Nach "Squeezing"

Abbildung 5.15: Ergebnisse des Verfahrens von Lauther (©1979 IEEE)

b) Abb. 5.16 a) zeigt einen kritischen Pfad durch einen der Polargraphen. In Abb. 5.16 b) ist zu erkennen, dass aufgrund der Lage der Schnittlinie eine größere Leerfläche entsteht. Die Schnittlinie ist für das

weitere Verfahren unerheblich, und die Zellen können verschoben werden. Im Polargraphen kann die resultierende neue Länge einfach durch Einführung einer Kante der Länge 0 bestimmt werden (siehe Abb. 5.17). Aus der Abb. 5.15 a) entsteht durch diese Optimierung die Abb. 5.15 b).

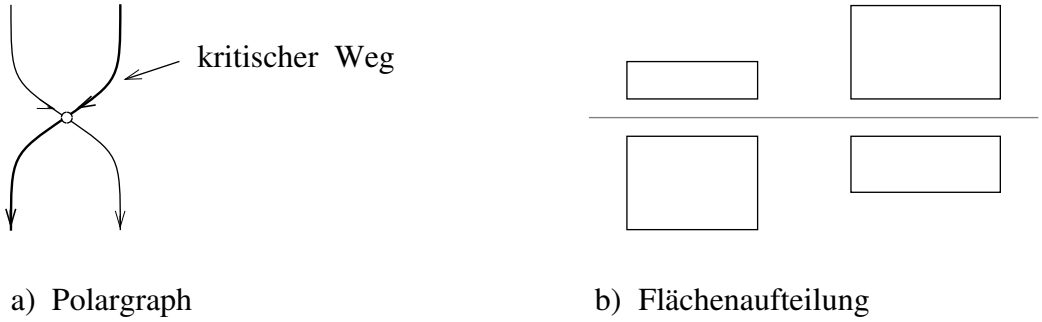


Abbildung 5.16: Squeezing : Ausgangssituation

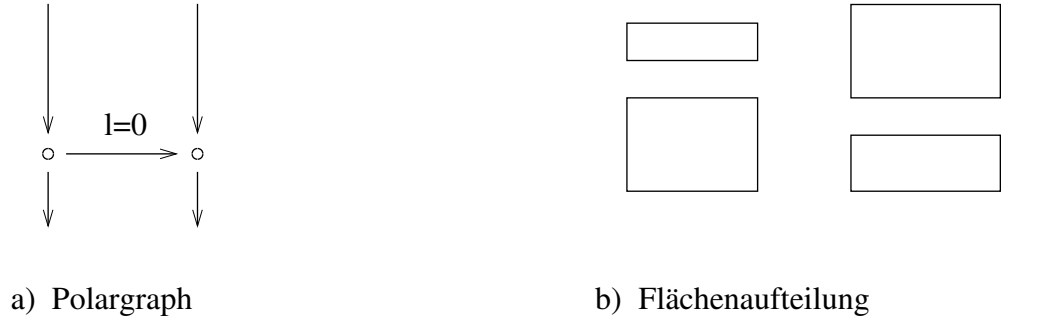


Abbildung 5.17: Squeezing : Ergebnis

5.1.4.7 Berücksichtigung von Netzanschlüssen außerhalb der gegenwärtigen Fläche

Ein Grundproblem der bislang vorgestellten fortgesetzten Bipartitionierung besteht darin, dass Netzanschlüsse außerhalb der zu teilenden Fläche nicht berücksichtigt werden. Man betrachte dazu Abb. 5.18.

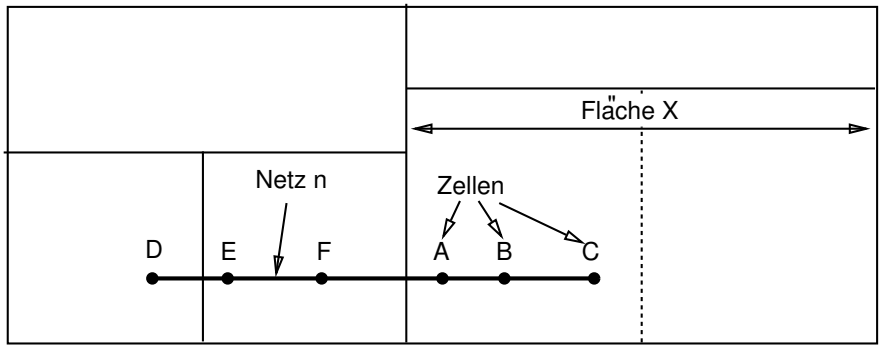


Abbildung 5.18: Zum Problem der Anschlüsse außerhalb der aktuellen Fläche `X`

Zu partitionieren sei die Fläche `X`. Ein Netz `n` habe, wie gezeigt, Anschlüsse an Zellen `A, B, C` innerhalb und an Zellen `D, E, F` außerhalb von `X`. Beachtet man lediglich die in `X` enthaltenen Zellen und die Schnitte von Netzen an der gestrichelten Linie, bietet es keinen Kostenvorteil, die Zellen `A, B` und `C` in der linken Teilfläche zu platzieren. Um den Effekt der äußeren Anschlüsse zu berücksichtigen, ersetzen Dunlop und

Kernighan [DK85] die Anschlüsse des Netzes `n` außerhalb von `X` durch eine "Dummy-Zelle" `Q` am linken Rand von `X` (siehe Abb. 5.19).

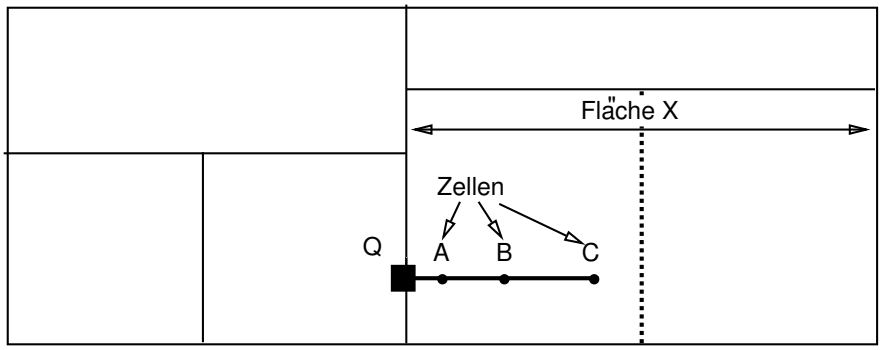


Abbildung 5.19: Einführung einer "Dummy"-Zelle `Q`

Diese Technik kann bei jeder Partitionierung benutzt werden, insbesondere auch für die externen Anschlüsse eines Chips.

5.1.4.8 Quadripartitionierung

Bei der Bipartitionierung bleiben die Entscheidungen relativ lokal. Die echte spätere Entfernung der Zellen wird nicht berücksichtigt. Als Verbesserung haben Suaris und Kedem [SK87] die Aufteilung der Zellen in vier Teilmengen und ihre Zuordnung zu vier Quadranten vorgeschlagen.

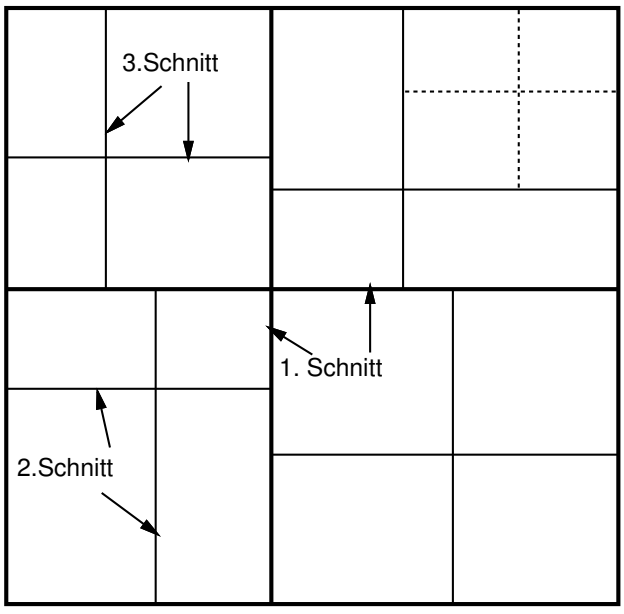


Abbildung 5.20: Aufteilung einer Fläche durch Quadripartitionierung

Danach kann eine Zelle A in einem bestimmten Quadranten mit einer Zelle B aus einem anderen Quadranten vertauscht werden. Gewählt wird jeweils das Paar (A, B), dessen Vertauschung den größten Gewinn erbringt. Zu diesem Zweck müssen für das Vertauschen von Zellen zwischen allen vier Quadranten separate Gewinn–Arrays geführt werden. Zur Auswahl von 2 aus 4 Quadranten gibt es 6 Möglichkeiten. Statt zweier Arrays bei der Bipartitionierung benötigt man daher $2 * 6 = 12$ Arrays.

Liegen die Anschlüsse eines Netzes in diagonal gegenüberliegenden Quadranten, so benötigt die Verdrahtung

sowohl horizontale wie auch vertikale Leitungen. Man kann die Kosten für solche Netze höher ansetzen, als die Kosten für Netze mit Anschlüssen in horizontal oder vertikal benachbarten Quadranten. Es ist ferner möglich, die Gewichte horizontaler Schnitte und vertikaler Schnitte unterschiedlich zu wählen, um so eine Verdrahtungsrichtung zu bevorzugen.

Die Quadripartitionierung erlaubt zwar eine Unterscheidung zwischen diagonal und nebeneinander liegenden Zellen, kann aber den endgültigen Abstand zwischen Zellen nach fortgesetztem Partitionieren nicht berücksichtigen. Daher hat man versucht, andere Optimierungsmethoden einzusetzen.

5.1.5 Simulated Annealing

simulated annealing ist eine allgemeine Methode zur Lösung von kombinatorischen Optimierungsproblemen. Vorbild für diese Methode ist das langsame Abkühlen von kristallinen Flüssigkeiten. Bei hinreichend langsamer Abkühlung entsteht dabei ein homogener, energetisch günstiger Kristallverband. Bei Anwendung dieser Methode wird eine sog. **Konfiguration**, d.h. ein Zustand des zu optimierenden Systems, Änderungen unterworfen. Im Falle der Platzierung bestehen diese z.B. aus dem Verschieben bzw. Vertauschen von Zellen. Das besondere am *simulated annealing*-Verfahren ist, dass auch Änderungen, die zu einer (bezügl. einer Kostenfunktion) schlechteren Konfiguration führen, mit einer gewissen Wahrscheinlichkeit akzeptiert werden. Wesentlicher Parameter dieser Methode ist der sog. **Temperaturparameter** T. Diese Wahrscheinlichkeit, mit der energetisch ungünstigere Konfigurationen akzeptiert werden, nimmt mit zurückgehender Temperatur ab.

Die folgende Prozedur beschreibt den groben Ablauf eines *simulated annealing*–Verfahrens:

```
PROCEDURE SimulatedAnnealing;
VAR i, T : Integer;
 BEGIN
  i := 0; T := MaxT;
  Konfiguration := <beliebige Ausgangskonfiguration>;
  WHILE NOT Abbruchkriterium(i, T) DO
   BEGIN
    WHILE NOT Schleifenende DO
     BEGIN
      NeueKonfiguration := Variation (Konfiguration);
      delta:=Beurteilung(NeueKonfiguration,Konfiguration);
      IF delta < 0
       THEN Konfiguration := NeueKonfiguration
       ELSE
        IF KleinGenug( delta, T, Zufall(0,1))
         THEN Konfiguration := NeueKonfiguration;
     END;
    T := NeuesT(i, T); i := i + 1;
   END;
 END;
```

Algorithmus 4.3: *simulated annealing*--Verfahren

Zunächst wird eine zufällige Ausgangskonfiguration (hier: eine Platzierung) erzeugt und die Temperatur `T` wird auf einen hohen Wert `MaxT` gesetzt. Dieser Wert wird in einer äußeren Schleife über die Funktion `NeuesT` langsam reduziert.

Für jeden Wert der Temperatur werden in einer inneren Schleife dann über die Funktion `Variation` zufällige Änderungen der jeweiligen Konfiguration erzeugt. Änderungen, die zu einer Reduktion der Energie bzw. **Kosten** führen (`delta < 0`), werden sofort akzeptiert. Änderungen, die zu einer Erhöhung der Energie bzw. **Kosten** führen (`delta >= 0`), werden mit einer Wahrscheinlichkeit akzeptiert, die von der Temperatur abhängt. Dadurch soll vermieden werden, dass das Verfahren in einem lokalen Minimum der Kosten anhält. Nach einer bestimmten, in der Regel von `T` abhängigen Anzahl von Iterationen wird die innere Schleife über die Funktion `Schleifenende` beendet.

Über die Funktion `Abruchkriterium` wird die äußere Schleife beendet, falls die Anzahl von Iterationen `i` oder die Temperatur eine gewisse Schwelle über- bzw. unterschritten haben.

Ein solches Verfahrung kann durch eine entsprechende Wahl der Funktionen weitgehend parametrisiert werden. Es ist nachweisbar [vLA87], dass es eine Konfiguration mit minimalen Kosten liefert, falls gewisse Bedingungen eingehalten werden. Die Resultate, die mit diesem Verfahren erzielt werden, sind bei geeigneten Parametern in der Regel sehr gut. Allerdings sind die Rechenzeiten sehr lang.

Eine der bekanntesten Anwendungen von *simulated annealing* für die Platzierung ist das Programm *TimberWolf 3.2* (siehe z.B. [Sec88]) zur Platzierung und Verdrahtung. Das Programm verwendet zwei Methoden zur Erzeugung neuer Konfigurationen:

1. Eine einzelne Zelle wird zufällig ausgewählt und ebenfalls zufällig neu platziert.
2. Zwei Zellen werden zufällig ausgewählt und vertauscht.

Die Entfernung, über die Zellen bewegt werden können, ist bei den hohen Temperaturen am größten. Die innere Schleife wird in dieser Version nach einer festen, durch den Benutzer bestimmten Anzahl von Iterationen abgebrochen.

TimberWolf 3.2 war aufgrund seiner guten Ergebnisse bereits ein recht erfolgreiches Programm. Mit *TimberWolf 4.2* [SL87] konnten die Ergebnissse durch eine Feinabstimmung der Parameter (z.B. ist die Zahl der Iterationen der inneren Schleife nicht mehr fest) und eine geschickte Programmierung noch einmal deutlich verbessert werden.

5.1.6 Genetische Algorithmen

Genetische Algorithmen sind ebenso wie der *Simulated Annealing*–Ansatz eine allgemeine Technik zur Lösung von kombinatorischen Optimierungsproblemen. Man beginnt dabei zunächst mit einer Menge zulässiger Lösungen. Jede Lösung nennt man in diesem Zusammenhang ein **Individuum**. Die Menge der aktuellen Lösungen heißt **Population**. Jedes Individuum wird dabei als ein String von Symbolen dargestellt. Die einzelnen Symbole heißen **Gene** und der String heißt **Chromosom**. Alle Individuen einer Population werden anhand einer **Fitneß–Funktion** hinsichtlich ihrer Leistungsfähigkeit zur Erzeugung einer guten Lösung beurteilt. Paare von leistungsfähigen Individuen werden sodann als **Eltern** der nächsten **Generation** von Individuen ausgewählt. Die Chromosomen der **Kinder** werden mittels dreier genetischer Operatoren aus den Chromosomen der Eltern abgeleitet. Diese sind:

1. Die **Kreuzung**: Aus den Chromosomen der Eltern wird jeweils ein Teil übernommen
2. Die **Mutation**: Im Chromosom des Kindes wird zufallsgesteuert ein Gen geändert
3. Die **Selektion**: Mittels einer Funktion wird aus den Kindern und der Elterngeneration die neue Generation ausgewählt

In der Anwendung auf Platzierungsprobleme repräsentiert jedes einzelne Symbol in der Regel die Platzierung einer Zelle (Zellname und Koordinaten).

Der Vorteil dieser Technik besteht darin, dass in einer Population stets eine Menge guter Lösungen der Optimierungsaufgabe verfügbar ist und dass aus guten Lösungen mittels komplexer Operatoren noch bessere Lösungen generiert werden können.

Wenn man stets nur die gegenwärtig beste Lösung speichert, so ist man evtl. nicht in der Lage, ein lokales Optimum zu verlassen. Es muß daher mit einer gewissen Wahrscheinlichkeit stets auch eine weniger gute Lösung akzeptiert werden. Beim Simulated–Annealing–Verfahren ist dies dann die einzige gespeicherte Lösung. Bei genetischen Algorithmen bleiben die besseren Lösungen weiterhin erhalten. Genetische Algorithmen bilden daher eine vielversprechende (und noch relativ junge) Optimierungstechnik. Andererseits benötigen sie deutlich mehr Speicherplatz als Simulated–Annealing–Verfahren. Details der Anwendung können dem Artikel von Shahookar et al. [SM91] entnommen werden.

5.1.7 Chip-Planning

Die bisherigen Verfahren gehen davon aus, dass die Längen/Breitenverhältnisse der Zellen fest vorgebenen sind. Bei komplexen Zellen gibt es jedoch in der Regel eine Vielzahl von Alternativen. Es ist günstig, wenn

man die Wahl einer Alternative, d.h. eines konkreten Längen/Breitenverhältnisses anhand der Umgebung im Layout treffen kann, damit sich die Zelle möglichst gut in das Gesamtlayout einfügt. Algorithmen, die dies leisten, nennt man **Chip-Planning-Algorithmen** (engl. *chip planning algorithms* oder *floor planning algorithms*). Es konnte gezeigt werden [Zim86, Sch88], dass damit erheblich kompaktere Layouts erreicht werden können.

5.2 Globale Verdrahtung

5.2.1 Allgemeines zur Verdrahtung

Zunächst wollen wir uns mit Beschreibungsmodellen der Verdrahtung beschäftigen. Hierzu werden einige graphentheoretische Begriffe benötigt.

Def. 5: Falls gilt: $\forall(v_1, v_2) \in E \Rightarrow (v_2, v_1) \in E$, so heißt der Graph **ungerichtet** und sonst **gerichtet**.

Def. 6: Ein **kantengewichteter Graph** ist ein Tripel (V, E, w), wobei (V, E) ein Graph ist und w eine Abbildung $w : E \rightarrow I\!R^+$.

Def. 7: Ein **Pfad** von $x \in V$ nach $y \in V$ ist eine Folge $(v_1, v_2, ..., v_n)$ mit $v_1 = x$, $v_n = y$ und $\forall_{1 \leq i \leq n-1} : (v_i, v_{i+1}) \in E$.

Im Folgenden beschränken wir uns auf ungerichtete Graphen.

Def. 8: Ein Graph heißt **zusammenhängend**, wenn zwischen je zwei Knoten ein Pfad existiert.

Def. 9: Ein Graph heißt **zyklenfrei**, wenn zwischen je zwei Knoten nicht mehr als ein Pfad existiert.

Def. 10: Ein (freier) **Baum** ist ein zusammenhängender, zyklenfreier Graph.

Def. 11: Ein **Baum mit Wurzel** (engl. *rooted tree*) ist ein freier Baum mit einem ausgezeichneten Knoten, der Wurzel.

Man beachte, daß man bei einem ungerichteten, freien Baum jeden Knoten zur Wurzel eines Baums mit Wurzel erklären kann.

Verdrahtungsnetze sind sicherlich zusammenhängend (sonst würde eine Leitung fehlen) und zyklenfrei (sonst wäre eine Leitung überflüssig). Verdrahtungsnetze bilden somit Bäume.

Def. 12: Ein **Spannbaum** (engl. *spanning tree)* eines Graphen $G = (V, E)$ ist ein (freier) Baum $B' = (V', E')$ mit $V = V'$ und $E' \subseteq E$.

Def. 13: Ein **minimaler Spannbaum** eines kantengewichteten Graphen G ist ein Spannbaum des Graphen G, der unter allen möglichen Spannbäumen die minimale Summe der Kantengewichte besitzt.

Bekannte Algorithmen zur Bestimmung von minimalen Spannbäumen stammen von Prim und von Kruskål (siehe z.B. [Sed88]).

Def. 14: Ein **Steiner-Baum** eines Graphen $G = (V, E)$ zur Knotenmenge $S \subseteq V$ ist ein Baum $B = (V', E')$ mit $E \subseteq E'$ und $S \subseteq V' \subseteq V$.

Def. 15: Ein **minimaler Steiner-Baum** eines kantengewichteten Graphen G zu einer Knotenmenge S ist ein Steiner-Baum, der unter allen möglichen Steiner-Bäumen die minimale Summe der Kantengewichte besitzt.

Abhängig von den Eigenschaften der Kantengewichte kann man zwischen verschiedenen Fällen des Problems der Bestimmung minimaler Steiner-Bäume unterscheiden. Für die Verdrahtung in einer Ebene ist zunächst einmal der spezielle Fall der Bestimmung des Steiner-Baumes für ein Manhattan-Abstandsmaß interessant.

Def. 16: Der Manhattan–Abstand zweier Punkte A und B in der Ebene mit den Koordinaten (x_A, y_A) bzw. (x_B, y_B) ist definiert als $d(A,B) = |x_A - x_B| + |y_A - y_B|$.

In Manhattan sind alle Straßen, mit Ausnahme des Broadway, rechtwinklig angeordnet. Daher beschreibt das Manhattan-Abstandsmaß die in Manhattan zwischen zwei Punkten A und B zurückzulegende Strecke. Das Manhattan-Abstandsmaß Layout-Erzeugung wichtig, da vielfach eine Beschränkung auf die rechtwinklige Verdrahtung erfolgt. Für das Manhattan-Maß gilt die Dreiecksungleichung.

Einen minimalen Steiner-Baum für ein Rechteckraster zeigt die Abbildung 5.21. Alle Kantengewichte sind als 1 angenommen.

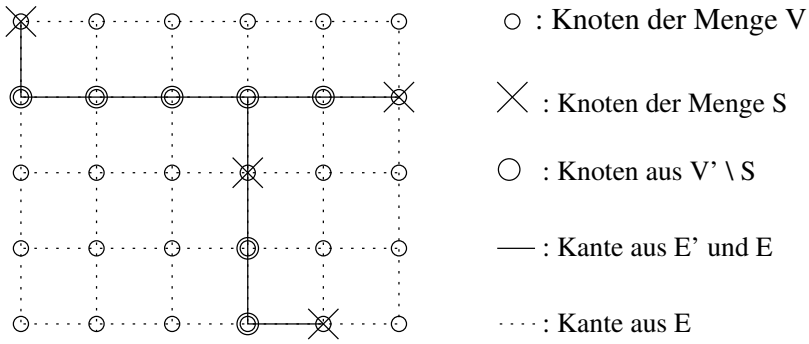


Abbildung 5.21: Minimaler Steiner-Baum in einem Rechteckraster

Das allgemeine Minimierungsproblem heißt **Steiner tree on graph problem** (STOGP) (siehe z.B. Abb. 5.22).

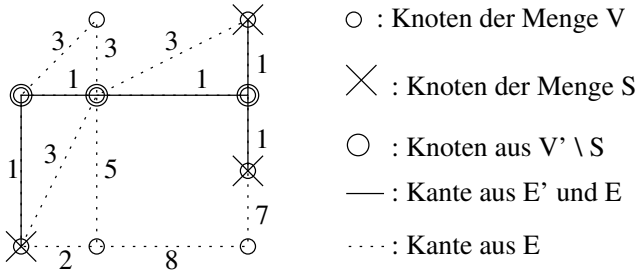


Abbildung 5.22: Minimaler Steiner-Baum in einem Graphen

Im allgemeinen Fall sind beliebige Kantengewichte zulässig.

Steiner- und Spannbäume stellen Alternativen für die Verdrahtung dar, wie anhand der folgenden Liste von Verdrahtungsalternativen zu sehen ist.

1. Steiner-Bäume:

   Der minimale **Steiner–Baum** ist ein Baum mit minimaler Verdrahtungslänge, der sich an beliebigen Knoten verzweigt (siehe Abb. 5.23 a, nach [Oht86]).

2. Spannbäume:

   Der minimale **Spannbaum** ist ein Baum mit minimaler Verdrahtungslänge, der sich nur an den Netzknoten verzweigt (siehe Abb. 5.23 b).

3. Minimale Ketten:

   Die minimale **Kette** ist ein Baum mit minimaler Verdrahtungslänge, der sich nirgends verzweigt (Abb. 5.24 a).

4. Minimale direkte Ein/Ausgangsverbindung:

   Die minimale **direkte Ein/Ausgangsverbindung** (siehe Abb. 5.24 b) ist ein Baum mit minimaler Verdrahtungslänge, in dem alle Eingänge direkt mit dem Ausgang verbunden sind. In den Abbildungen 5.23 und 5.24 ist die jeweilige Verdrahtungslänge mit ℓ bezeichnet.

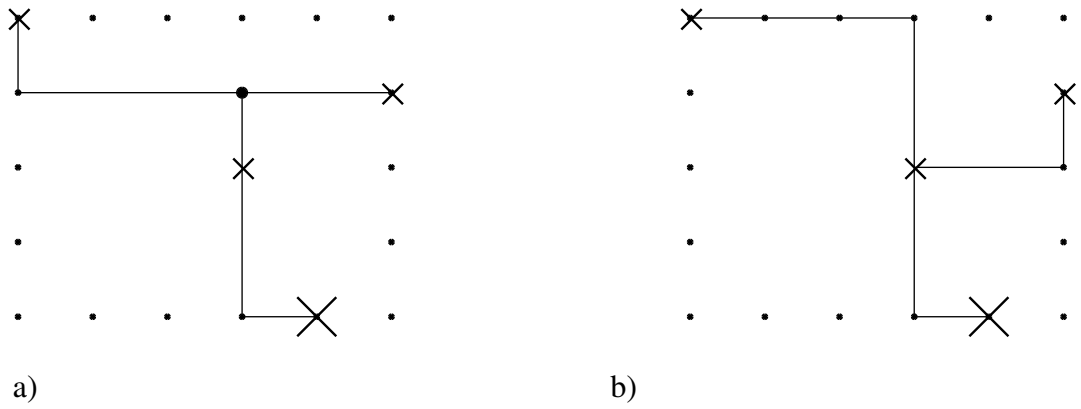


Abbildung 5.23: a) Minimaler Steinerbaum ($\ell = 10$) b) Minimaler Spannbaum ($\ell = 11$) (.=Rasterpunkte, x=Eingänge, X=Ausgang, o=Verdrahtungsknoten)

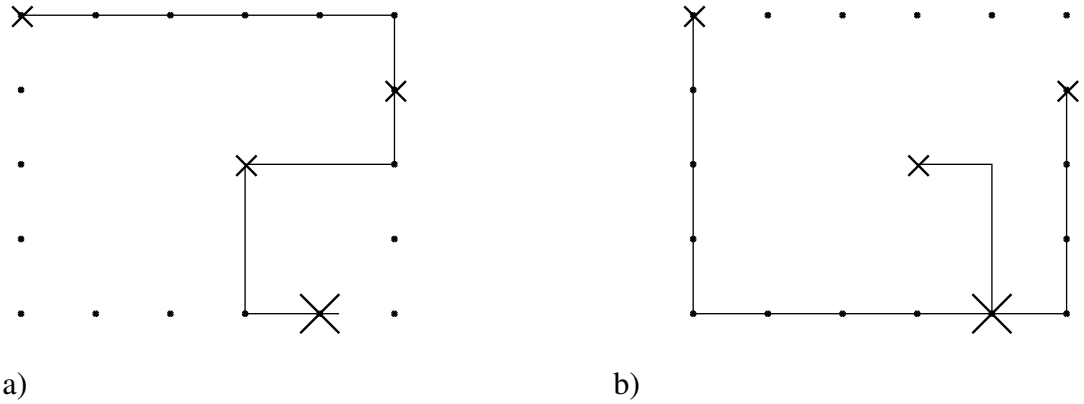


Abbildung 5.24: a) Kette ($\ell = 12$) b) direkte E/A-Verbindung ($\ell = 15$)

Zur Abschätzung der Verdrahtungslänge (ohne tatsächlich zu verdrahten) dient die **Methode des halben Umfangs** (engl. *half perimeter method*): Die Länge der Verdrahtung wird durch die Kantensumme des kleinsten Rechtecks, in dem alle zu verdrahtenden Knoten liegen, abgeschätzt (siehe Abb. 5.25). Bei 2- und 3-Punkt-Netzen ist die Abschätzung exakt gleich der Länge des minimalen Steiner–Baumes (Übungsaufgabe!).

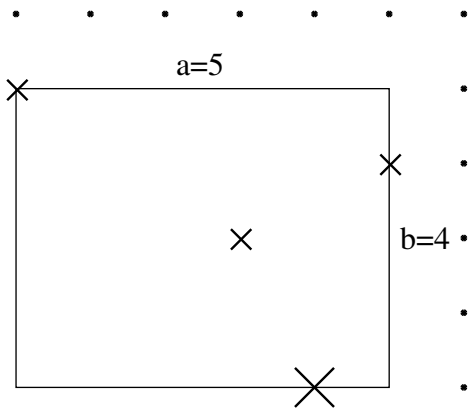


Abbildung 5.25: Umschreibendes Rechteck, (ℓ=9)

5.2.2 Problemstellung der globalen Verdrahtung

Verdrahtungs-Algorithmen, die eine detaillierte Verdrahtung vornehmen, sind nicht geeignet, global über ein ganzes Chip mit z.B. einer Million Transistoren angewandt zu werden. Daher wird zwischen der Platzierung und der detaillierten Verdrahtung, wie sie im nächsten Abschnitt vorgestellt werden wird, meist ein weiterer Schritt eingefügt. Dieser Schritt heißt **globale Verdrahtung** oder **globales Routing** (engl. *global routing*). In diesem Schritt werden die Netze einer Menge von sog. **Verdrahtungsregionen** zugeordnet. Abb. 5.26 zeigt eine mögliche Einteilung einer Fläche in solche Regionen.

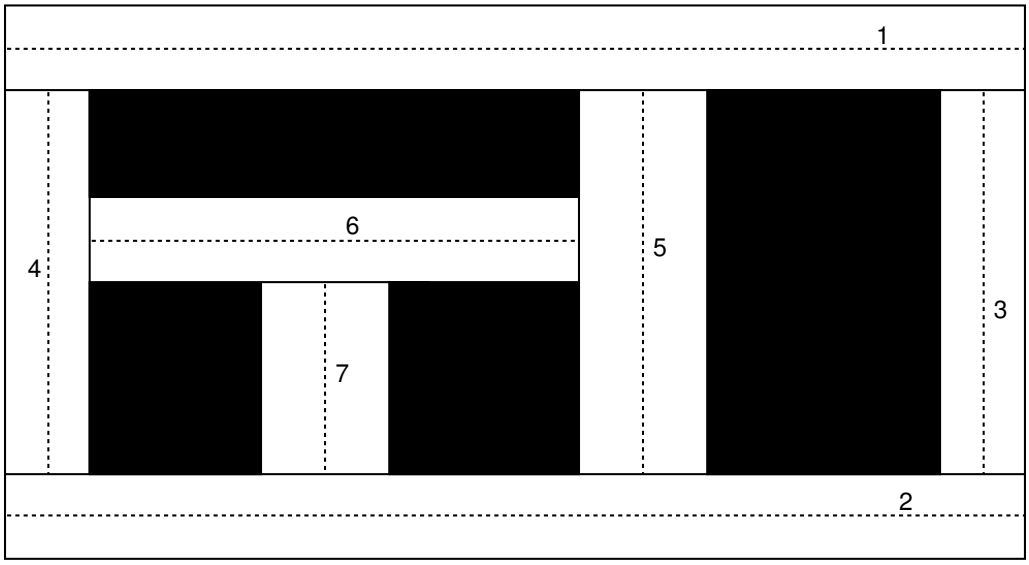


Abbildung 5.26: Verdrahtungskanäle

Diese Einteilung könnte durch fortgesetztes Schneiden der Fläche entstanden sein. Die Schnittlinien sind in der Abbildung gestrichelt gekennzeichnet. Eine mögliche Reihenfolge des Schneidens ist durch die Zahlen angedeutet. Jede Schnittlinie führt zu einem Rechteck, in dem eine Verdrahtung möglich ist. Diese Bereiche heißen **Kanäle**[8]. In der Abbildung wurde angenommen, dass auch die Ränder geschnitten wurden, um zu entsprechenden Verdrahtungsregionen am Rand zu kommen. Die Kanäle der Abb. 5.26 erstrecken sich über die volle Länge der entsprechenden Flächenschnitte.

Die Einteilung der Flächen in derartige Kanäle ist für viele Zwecke zu grob. Für die Zwecke der globalen Verdrahtung werden wir im Folgenden die feinere Flächenaufteilung nach Abb. 5.27 voraussetzen.

Diese ergibt sich aus der vorhergehenden dadurch, dass wir Kanäle in kleinere Rechtecke unterteilen, welche jeweils an jedem Rand homogen, d.h. entweder von einer Zelle oder von einem anderen Kanal begrenzt sind. Wir nennen diese kleineren Kanäle **Minikanäle**. In der globalen Verdrahtung soll festgelegt werden, durch welche der Minikanäle z.B. die Verdrahtung des Netzes `n` zu führen ist, für das die Abb. 5.27 die Anschlüsse enthält.

Zur Modellierung des globalen Verdrahtungsproblems werden vor allem Graphen benötigt, welche die Nachbarschaft von Verdrahtungsregionen darstellen. Zu diesem Zweck arbeitet man mit sog. **globalen Verdrahtungsgraphen**. Für diese gibt es recht unterschiedliche Definitionen. Sie hängen beispielsweise davon ab, ob Gate–Arrays, Standard–Zellen oder Macro–Zellen eingesetzt werden.

Def. 17: Ein **Nachbarschaftsgraph** (engl. *regions adjacency graph*) ist ein ungerichteter Graph, der für jede Verdrahtungsregion genau einen Knoten enthält. Zwei Knoten sind genau dann mit einer Kante verbunden, wenn die entsprechenden Verdrahtungsregionen benachbart sind[9].

Abb. 5.28 zeigt einen Nachbarschaftsgraphen für die Minikanäle der Abb. 5.27.

[8] Eine genaue Definition des Problems der Verdrahtung innerhab von Kanälen erfolgt im Abschnitt **Kanalverdrahtung**.
[9] In einer anderen möglichen Definition werden die Kanäle als Kanten und die Zellen als Knoten modelliert.

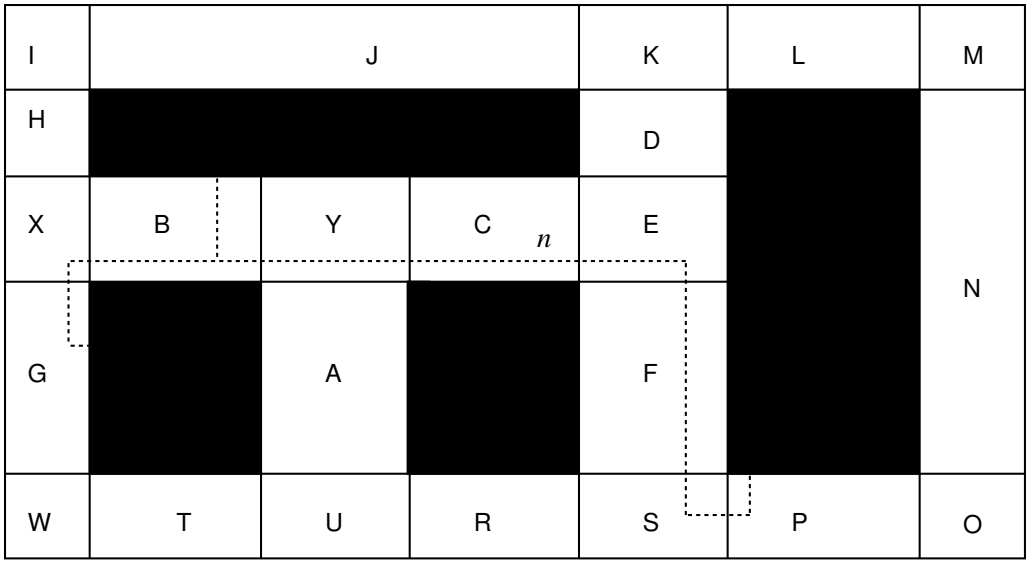


Abbildung 5.27: Minikanäle , Verdrahtung eines Netzes `n`

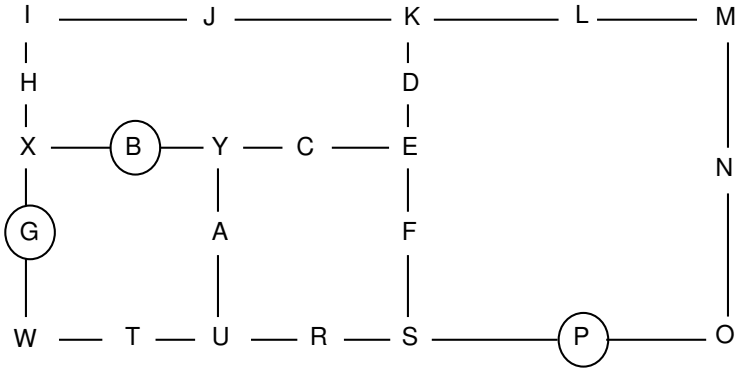


Abbildung 5.28: Nachbarschaftsgraph (Kreise kennzeichnen die Anschlüsse des Netzes `n`)

5.2.3 Das *Steiner tree on graph*-Problem

Das Ziel der globalen Verdrahtung können wir jetzt wie folgt präzisieren: Wir wollen bestimmen, durch welche Minikanäle Netze verlaufen sollen. Die genaue Verdrahtung innerhalb der Minikanäle bzw. der Kanäle erfolgt dann in der detaillierten Verdrahtung.

Betrachten wir dazu jetzt das Netz `n` der Abb. 5.27. Dieses Netz hat Anschlüsse in den Minikanälen `B, G` und `P`. Durch diese Minikanäle ist das Netz auf jeden Fall zu führen. Die entsprechenden Knoten sind in der Abb. 5.28 eingekreist. Gesucht sind jetzt die Minikanäle, durch die das Netz außerdem noch verlaufen soll, um die Anschlüsse miteinander zu verbinden. Diese müssen im Nachbarschaftsgraphen einen Baum bilden. Da Verzweigungen an jedem Knoten möglich sind, wird ein **Steiner-Baum** (siehe Definition 12) in einem Graphen G gesucht.

Die Menge der Minikanäle, in denen sich Anschlüsse befinden, bildet dabei die Menge S. V' ist die Menge der in der endgültigen Verdrahtung benutzten Minikanäle.

Bei Verwendung des Steiner-Baum-Modells der globalen Verdrahtung müssen die Netze einzeln nacheinander den Minikanälen zugeordnet werden. Um zu verhindern, dass einzelne Kanäle dabei zu stark gefüllt werden, kann man zu gewichteten Graphen übergehen und für stark gefüllte Kanäle ein großes Gewicht vergeben. Üblicherweise gewichtet man die Kanten des Graphen mit einem Gewicht w. Ziel ist dann die Minimierung des Gesamtgewichtes, also die Bestimmung eines sog. **minimalen Steiner-Baumes**.

Das Problem der Bestimmung eines minimalen Steiner-Baumes wird als *Steiner tree on graph*–Problem (**STOGP**) bezeichnet. Das allgemeine STOGP ist NP–hart ([GJ79] enthält den Beweis des zugehörigen

Entscheidungsproblems als Übungsaufgabe). Drei Spezialfälle sind effizienter lösbar:

1. Im Spezialfall $S = V$ fällt das STOGP mit dem Problem der Bestimmung des minimalen Spannbaumes zusammen.
2. Im Spezialfall einer zweielementigen Menge S entsteht das Problem der Bestimmung des kürzesten Weges zwischen eben diesen beiden Knoten im Graphen. Dieser Spezialfall kann z.B. mittels des Algorithmus von Dijkstra [Dij59, AHU74] effizient gelöst werden.
3. Im Fall einer dreielementigen Menge kann eine optimale Lösung noch effizient mittels eines mehrfachen Aufrufs des Dijkstra-Algorithmus gefunden werden (siehe unten).

Wir betrachten zunächst den zweiten Spezialfall. Dazu müssen wir zunächst die Abbildung w verallgemeinern.

5.2.4 Dijkstra's Algorithmus

Die Abbildung w ist nur für die existierenden Kanten des globalen Verdrahtungsgraphen definiert. Diese Abbildung werde jetzt zur Abbildung ℓ erweitert, die jedem Knotenpaar ein Gewicht zuordnet:

Def. 18:

$$\forall v, v' \in V : \ell(v, v') := \begin{cases} 0 & \text{, falls } v = v' \\ w(e) & \text{, falls } e = (v, v') \in E \\ \infty & \text{, sonst} \end{cases}$$

Der Algorithmus von Dijkstra berechnet die Länge der kürzesten Wege zwischen einem festen Knoten v_0 und allen übrigen Knoten v eines Graphen.

```
S' := {v0};
D[v0] := 0;
FOR each v ∈ V − {v0} DO D[v] := ℓ(v0, v);
WHILE S' ≠ V DO     (*''Expansion''*)
  BEGIN
    Wähle Knoten x ∈ V − S' mit D[x] ist minimal;
    S' := S' ∪ {x};
    FOR each v ∈ V − S' DO
      D[v] := min (D[v], D[x] + ℓ(x, v))
  END;     (*D[v] enthält Abstand zwischen v0 und v*)
```

Algorithmus 4.4: Dijkstra's Algorithmus zur Berechnung kürzester Wege

Die Betrachtung der Knoten in der **Reihenfolge des wachsenden Abstandes von** v_0 und die jeweilige Erweiterung des Graphen um diese Knoten bilden die Grundoperation in Dijkstra's Algorithmus. Diese Grundoperation heißt **Expansion**.

Der Algorithmus läßt sich so erweitern, dass die Rückverfolgung des benutzten Weges möglich ist. Dazu muß Buch geführt werden, aufgrund welchen aktuellen Weges ein in D eingetragener Abstand reduziert wird.

Beispiel für Dijkstra's Algorithmus:

Die Tabelle 5.30 zeigt Momentaufnahmen der Variablenbelegungen zu Beginn der `WHILE`-Schleife in Algorithmus 4.4. Es ist erkennbar, wie die in D kodierte Länge reduziert wird, wenn weitere Knoten direkt erreichbar werden.

Für den schnellen Zugriff auf den Knoten x mit minimalem Abstand $D[x]$ können sog. *Fibonacci-Heaps* nach Fredman und Tarjan [FT87] benutzt werden. Mit diesen ergibt sich für den Dijkstra-Algorithmus eine Komplexität von $O(|E| + |V|log|V|)$ [LN86].

Der Dijkstra-Algorithmus liefert in der angegebenen Form im Array D nur die Länge der Pfade im Graphen. Er kann so erweitert werden, dass auch der jeweils benutzte Pfad mit abgespeichert wird (Übungsaufgabe!).

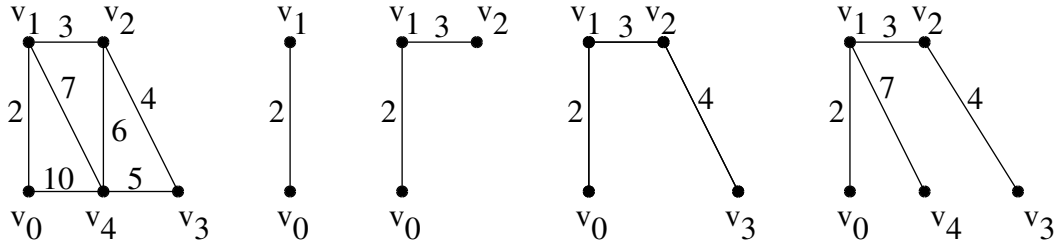


Abbildung 5.29: a) Kantengewichteter Graph $G = (V, E)$ und b) Expansionen

ITERATION	S'	x	$D[x]$	$D[v_1]$	$D[v_2]$	$D[v_3]$	$D[v_4]$
–	v_0	–	–	2	∞	∞	10
1	$\{v_0, v_1\}$	v_1	2	2	5	∞	9
2	$\{v_0 - v_2\}$	v_2	5	2	5	9	9
3	$\{v_0 - v_3\}$	v_3	9	2	5	9	9
4	$\{v_0 - v_4\}$	v_4	9	2	5	9	9

Abbildung 5.30: Bestimmung kürzester Wege nach Dijkstra

5.2.5 Optimaler Algorithmus für das 3-Punkt-STOGP

Im Falle einer dreielementigen Menge S besitzt ein in $G = (V, E)$ eingebetteter Baum genau eine Verzweigung, d.h. genau einen Knoten mit mehr als einer Kante.

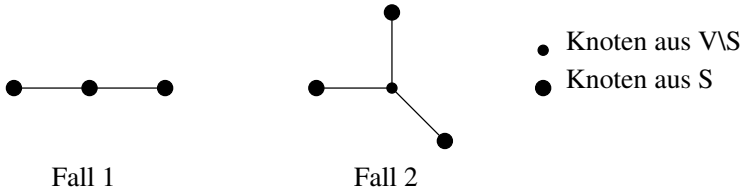


Abbildung 5.31: Verzweigungsknoten bei 3-Punkt-Steiner-Bäumen

Dieser Knoten läßt sich durch dreimaliges Aufrufen des Dijkstra–Algorithmus berechnen[10]: Man starte den Dijkstra–Algorithmus von jedem Knoten $s \in S$[11]. Für jeden Knoten wird dabei die Entfernung vom Startknoten bis zu diesem Knoten notiert. Für das zuletzt benutzte Beispiel zeigt die Abb. 5.32 die Entfernungen von den Knoten `a, b` und `c` als Tripel in runden Klammern.

Der Knoten mit der kleinsten Summe ist nach Definition des Steiner–Baumes der Verzweigungspunkt. Die Suche nach diesem Knoten erfordert maximal $|V|$ Schritte. Mit dem Dijkstra-Algorithmus können anschließend auch die Pfade vom Verzweigungspunkt zu den Knoten aus S bestimmt werden, wenn der Algorithmus hierzu um eine Buchführung über die benutzten Pfade erweitert wird.

5.2.6 *single component growth*–Algorithmus

Heuristiche Algorithmen werden vielfach bereits ab $|S| = 3$ benutzt, obwohl dies erst $|S| = 4$ notwendig ist. Diese Algorithmen bestimmen zunächst den kürzesten Weg zwischen 2 Punkten des Graphen. Dieser Weg einschließlich aller enthaltener Knoten bildet einen temporären Graphen G_1.

Anschließend bestimmt man den kürzesten Weg zwischen G_1 und einem dritten Knoten. Zu diesem Zweck ist Dijkstra's Algorithmus so zu erweitern, dass auch der kürzeste Weg zwischen einem Graphen und einzelnen Knoten gesucht werden kann.

Dieser Prozess wird mit dem vierten, fünften usw. Knoten fortgesetzt, bis alle Knoten aus S betrachtet wurden. Der folgende Text zeigt den Rumpf einer entsprechenden Prozedur:

[10] Nach Floren [Flo91a]. Die Möglichkeit hierzu wird in [HKK+90] ohne Angabe von Details erwähnt.
[11] Falls die übrigen Knoten aus S erreicht sind, kann der Dijkstra-Algorithmus abgebrochen werden.

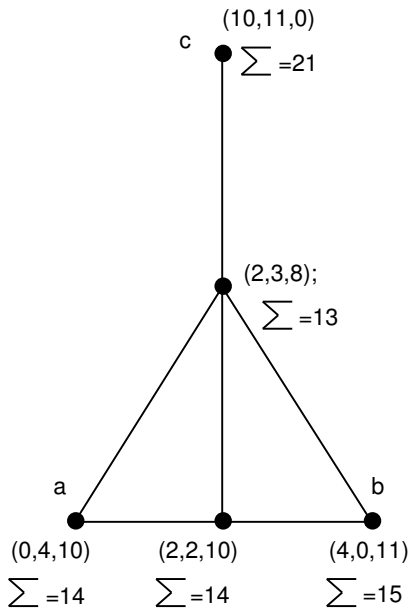


Abbildung 5.32: Lösung des obigen Beispiels mit dem optimalen Algorithmus

```
IF (|S| < 2) THEN RETURN;
G1 := ({s}, ∅, w) mit s ∈ S (irgendein s ∈ S);
S' := {s};
WHILE S' ≠ S DO
  BEGIN
    expandiere G1 bis ein Knoten t ∈ S − S' erreicht ist;
    G1 := G1∪ (irgendein) kürzester Weg von S' nach t;
    S' := S' ∪ {t};
  END;
```

Algorithmus 4.5: Single Component Growth-Algorithmus

Abb. 5.33 zeigt ein Beispiel für diesen Algorithmus. Für die heuristische Lösung ergibt sich eine Verdrahtungslänge von $\ell = 14$. Die optimale Lösung besitzt dagegen eine Verdrahtungslänge von $\ell = 13$.

Selbstverständlich wird mit diesem Algorithmus nicht die beste Lösung gefunden. Nach der Erweiterung des Graphen G_1 können nämlich Wege, die bislang die kürzesten waren, ggf. unnötig lang werden.

5.2.7 Approximative Lösung des STOGP mittels Distanzgraphen

Eine andere Basis sehr guter Algorithmen für das STOGP ist das Verfahren von Kou, Markowsky und Berman [KMB81]. Die Idee besteht im Wesentlichen aus der Modifikation eines geeigneten Spannbaumes, für den die Verdrahtungslänge nachweisbar maximal knapp das Doppelte der minimalen Verdrahtungslänge betragen kann. Der Algorithmus besteht aus den folgenden Schritten:

1. Die Grundidee des Verfahrens ist die Konstruktion eines vollständigen Graphens mit den Elementen aus S als Knoten. Die Kanten besitzen das Gewicht des Abstandes der Knoten voneinander im Graphen G. Dieser Graph heißt **Distanzgraph** für S.

   Beispiel:

   Abb. 5.34 a) zeigt einen Ausgangsgraphen G und Abb. 5.34 b) zeigt den zugehörigen Distanzgraphen G_1.

   Die Berechnung des Distanzgraphen bildet den ersten Schritt des Algorithmus von Kou et al. : Berechne $G_1 = (V_1, E_1)$ mit $V_1 = S$ und $\forall x, y \in S : (x, y) \in E_1$ sowie $w(x, y) = \text{Abstand}(x, y)$.

2. Ein minimaler Spannbaum dieses Distanzgraphen bildet die Ausgangsbasis für die Konstruktion des

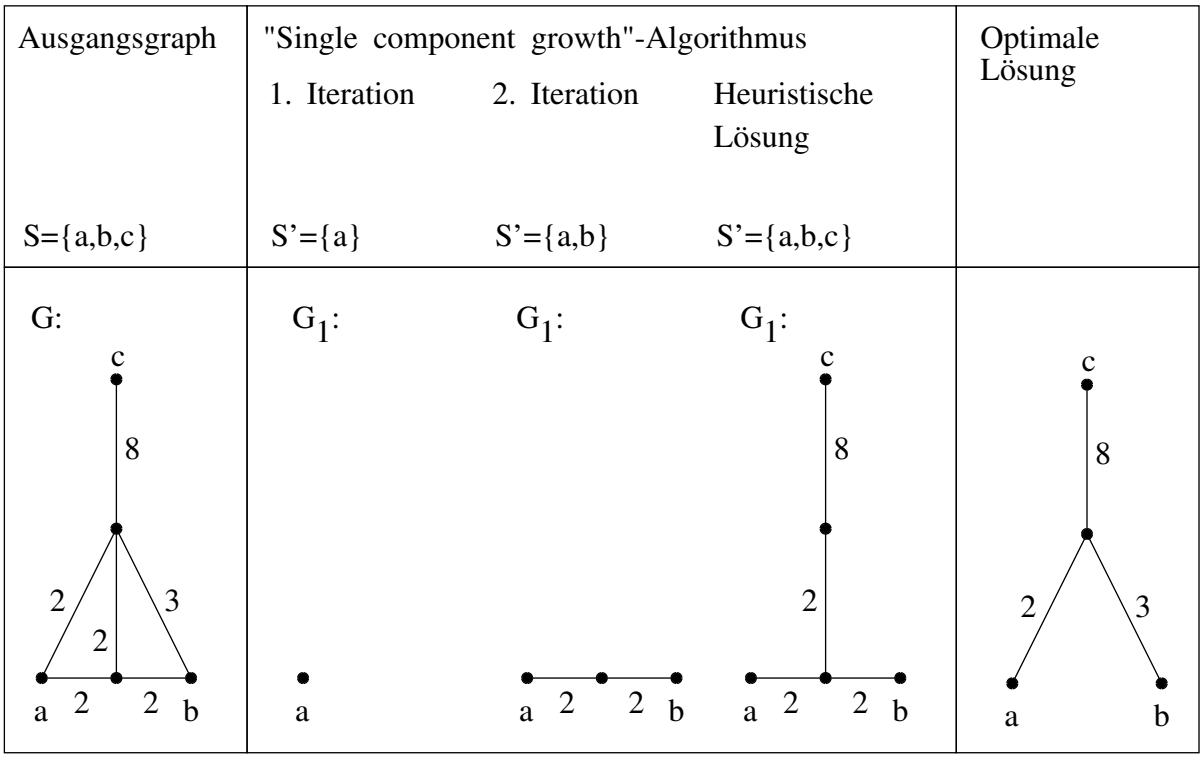


Abbildung 5.33: Beispiel für den *single component growth*-Algorithmus

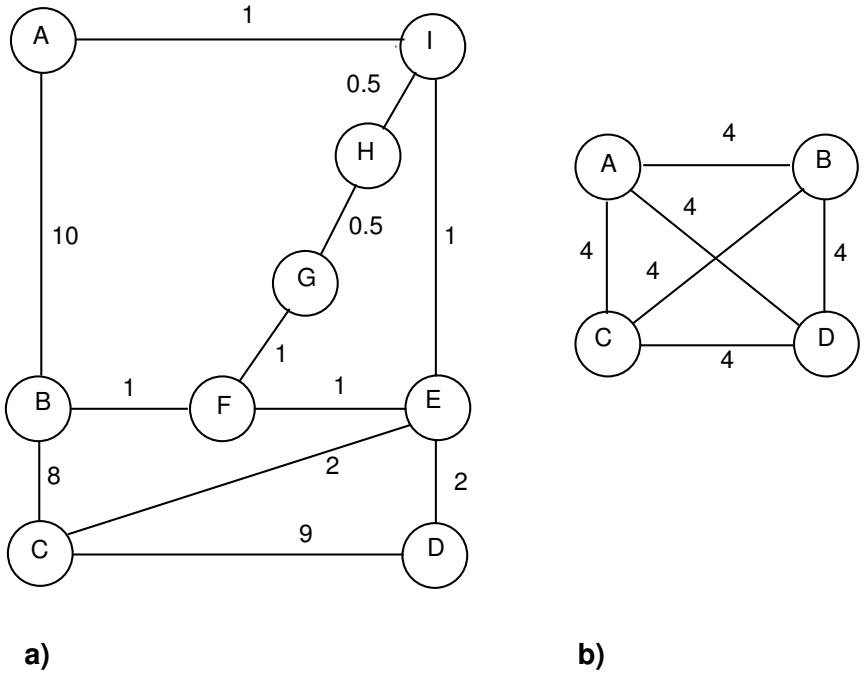


Abbildung 5.34: a) Ausgangsgraph G mit $S = \{A, B, C, D\}$; b) Distanzgraph G_1

Steiner-Baums. Abb. 5.35 a) zeigt einen der minimalen Spannbäume des soeben gezeigten Graphen G_1.

Die Berechnung eines minimalen Spannbaums G_2 von G_1 bildet den zweiten Schritt des Algorithmus von Kou et al.

3. Ersetze in G_2 jede Kante durch einen Pfad derselben Länge in G. Der erhaltene Graph heiße G_3.

   Abb. 5.35 b) zeigt die Pfade, die in den Graphen G_3 übernommen werden, für unser Beispiel. Die zugehörigen Kanten in G_2 sind gestrichelt eingezeichnet.

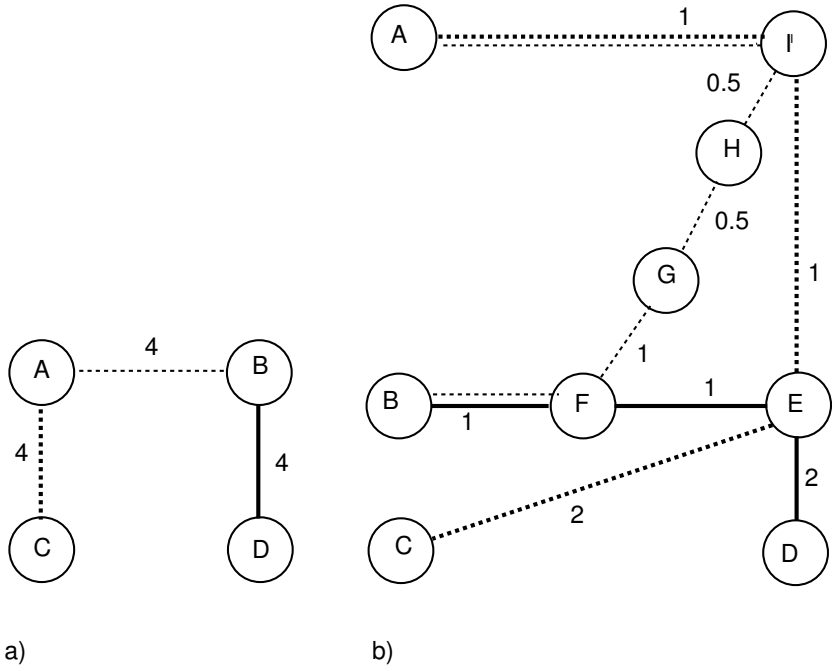


Abbildung 5.35: a) Spannbaum G_2; b) Zum Spannbaum gehörende Pfade (Löschen doppelter Kanten ergibt G_3)

4. Berechne einen minimalen Spannbaum G_4 von G_3. Abb. 5.36 a) zeigt den Graphen G_4 für unser Beispiel.

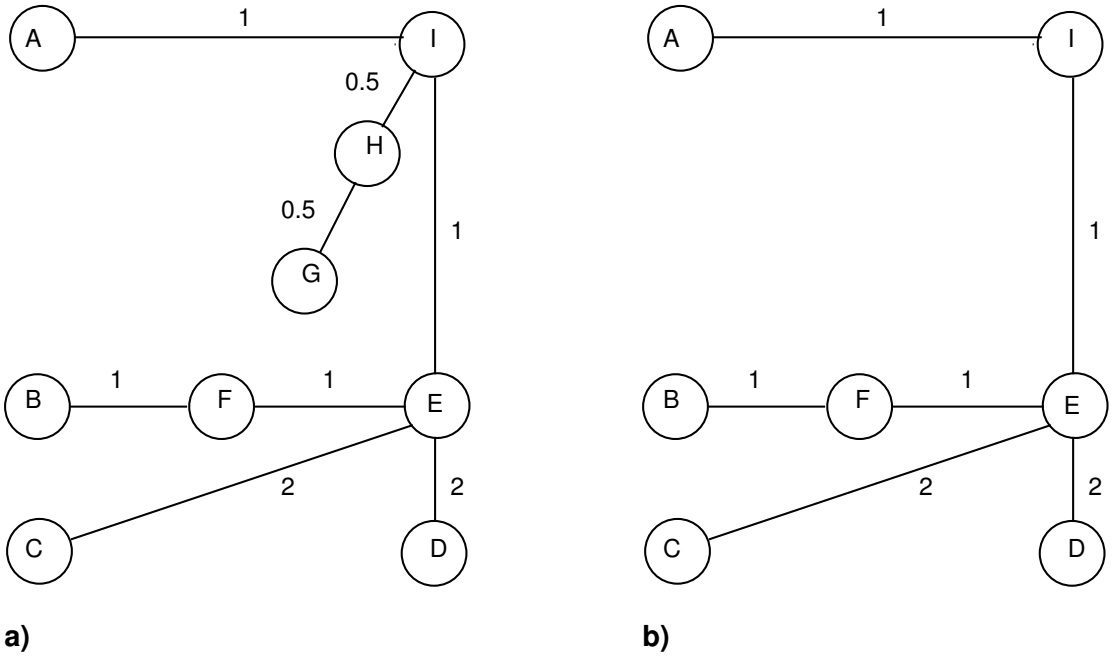


Abbildung 5.36: a) Graph G_4; b) Graph G_5

5. Entferne in G_4 die Blätter, die nicht zu S gehören. Der erhaltene Graph heiße G_5. Abb. 5.36 b) zeigt das Ergebnis für unser Beispiel.

Der Algorithmus liefert per Konstruktion in Schritt 4 sicher einen Baum. Aufgrund der Schritte 1 bis 3 sind in diesem Baum alle Knoten aus S enthalten. Aufgrund von Schritt 5 sind alle Blätter Knoten aus S. Der Algorithmus liefert folglich einen Steiner–Baum. Sei ℓ_{opt} die Kantensumme eines minimalen Steiner–Baumes. Dieser habe e Blätter. Man kann zeigen [Len90], dass für die Kantensumme ℓ_{Kou} des Graphen G_5 gilt:

$$\ell_{Kou} \leq 2 * \ell_{opt}(1 - 1/e)$$

Die Beweisidee ist folgende: Man nehme dazu an, dass der in Abb. 5.37 gezeigte Baum der minimale Steiner-Baum eines Steiner-Baum-Problems sei.

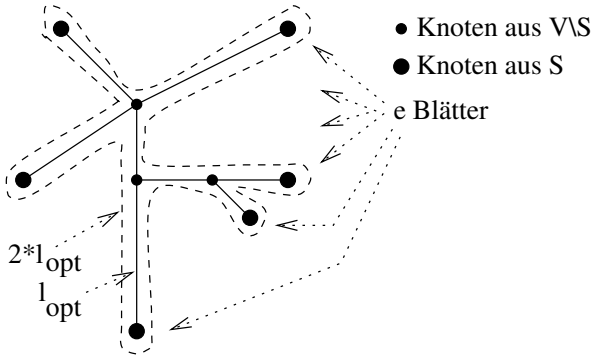


Abbildung 5.37: Zur oberen Schranke für die Länge des Steiner-Baumes nach der o.a. Methode

Der oben beschriebene Algorithmus sucht Wege zwischen Paaren von Knoten aus S. Die Wege innerhalb des Steiner-Baums entlang der gestrichelten Linie beschreiben Wege zwischen Paaren von Knoten aus S, wenngleich nicht notwendigerweise die kürzesten. Da der oben beschriebene Algorithmus die kürzesten Wege sucht, wird deren Länge sicher nicht größer sein als diejenigen entlang der gestrichelten Linie. Die Länge der gestrichelten Linie ist gleich dem Zweifachen der Länge des Steiner-Baumes.

Die gestrichelte Linie enthält einen Zyklus. Wir können aus ihr die längste Verbindung zwischen einem Paar von Knoten entfernen und verbinden dennoch alle Punkte aus S durch einen (entarteten) Baum. Die Länge der längsten Verbindung ist mindestens ℓ_{opt}/e, also verbleibt für die unterbrochene gestrichelte Linie eine Länge von maximal $2 * \ell_{opt}(1 - 1/e)$. Der im o.a. Algorithmus aufgebaute Spannbaum des Distanzgraphen hat diese Länge als obere Schranke. Die weiteren Schritte verkürzen den Spannbaum nur.

Die Laufzeit wird durch Schritt 1 bestimmt. Bei Verwendung des schnellen Algorithmus von Fredman und Tarjan [FT87] zur Bestimmung des kürzesten Weges ergeben sich $|S|$ Aufrufe der Komplexität $O(|E| + |V|log|V|)$, d.h. eine Gesamtkomplexität von $O(|S|(|E| + |V|log|V|))$.

Mehlhorn zeigte 1988 [Meh88], dass nur ein Teil des Distanzgraphen aus Schritt 1 wirklich benötigt wird, und dass daher die Komplexität auf $O(|E| + |V|log|V|)$ reduziert werden kann.

Floren zeigte 1990 [Flo91b], dass beim Vorgehen nach Mehlhorn in Schritt 1 die Schritte 4 und 5 überflüssig sind, da in diesem Fall G_3 stets ein Baum ist.

5.2.8 Probleme sequentieller Router

Ein wesentlicher Nachteil der bislang vorgestellten und vieler der nachfolgenden Verdrahtungsverfahren liegt in ihrem sequentiellen Vorgehen (man spricht auch von der *net at a time*-Verdrahtung). Da nicht alle Netze auf einmal verdrahtet werden, wird das Minimum der Gesamtverdrahtungslänge nicht notwendig erreicht.

In der detaillierten Verdrahtung werden im Extremfall sogar existierende Lösungen nicht gefunden. Dieser Fall tritt ein, wenn es zur Lösung des Gesamtproblems notwendig ist, dass Netze mit nicht-minimaler Länge realisiert werden, weil sie sonst den Platz für andere Netze blockieren. Falls es zwei Netze gibt, bei denen

die Wahl des kürzesten Weges für ein Netz jeweils das andere Netz blockiert, hilft auch keine Vorsortierung der Netze.

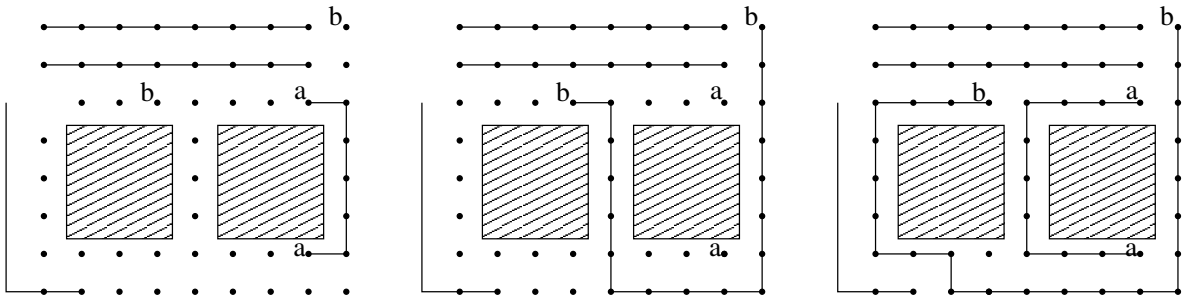


Abbildung 5.38: Problem, welches bei *net at a time*-Verdrahtung unlösbar ist

In diesen Fällen müssen die Netze möglichst in ihrer Gesamtheit betrachtet werden, z.B. mit Mitteln der Kombinatorik.

Ein mögliches Verfahren nach Lengauer beschreibt Sherwani [She98]. Danach betrachtet man für jedes Netz eine Menge alternativer Steiner-Bäume. Die Auswahl je eines Baums erfolgt über ein *integer programming*-Modell, welches die wechselseitigen Abhängigkeiten der Auswahl widerspiegelt. Die Randbedingungen stellen u.a. sicher, dass für jedes Netz ein Steiner-Baum ausgewählt wird. Das Modell ist zu komplex ist, um geschlossen zu lösen. Ist wird daher in hierarchische Teilmodelle handhabbarer Komplexität zerlegt.

5.2.9 Sortierung der Verdrahtungsregionen

Während der im Abschnitt 5.3 beschriebenen detaillierten Verdrahtung werden wir vor allem eine Verdrahtung innerhalb der Kanäle betrachten, wie sie in Abb. 5.26 betrachtet wurde. Um dazu die Kanalverdrahtungs-Algorithmen aus dem nächsten Abschnitt anwenden zu können, muß die Verdrahtung der Kanäle in einer bestimmten Reihenfolge erfolgen. Man betrachte dazu die Kanäle I und II in Abb. 5.39. Kanal I muß vor Kanal II verdrahtet werden, damit während der Kanalverdrahtung des Kanals II bereits die Lage der Netze am Rand des Kanals bekannt ist und damit der Abstand zwischen den Zellen A und B während der Verdrahtung des Kanals I noch verändert werden kann.

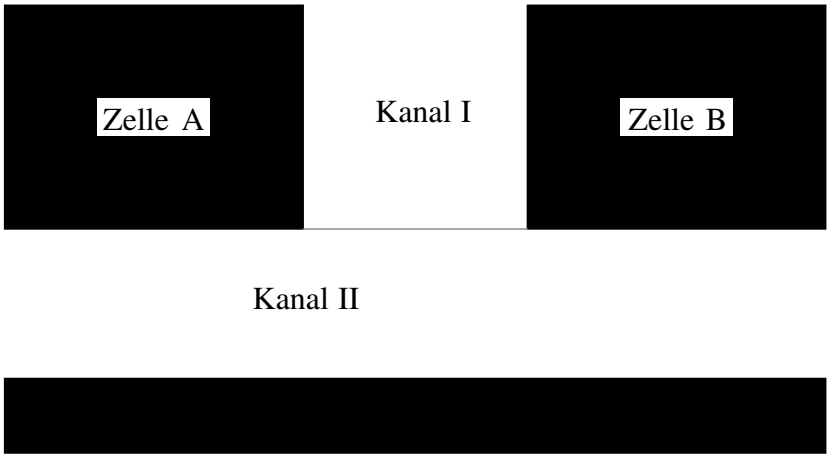


Abbildung 5.39: Zur Reihenfolge der detaillierten Verdrahtung

Als Konsequenz muß die Verdrahtung einer Slicing–Struktur in einer gegenüber der Schnittabfolge umgekehrten Reihenfolge durchgeführt werden (siehe Abb. 5.40).

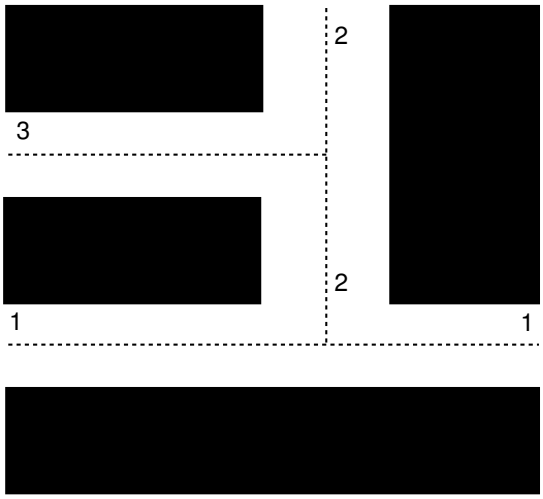


Abbildung 5.40: Reihenfolge der Schnitte bei Slicing-Trees

5.3 Detaillierte Verdrahtung

5.3.1 Verdrahtung bei einer Verdrahtungsebene

Als Sonderfall der Verdrahtung kann man die Verdrahtung in einer einzelnen Verdrahtungsebene betrachten. Dies ist selbst dann wichtig, wenn mehrere Verdrahtungsebenen existieren. So werden die Netze für die Betriebsspannung und für das Taktsignal typischerweise in einer Ebene ausgeführt. Ein Netzgraph kann in einer Ebene realisiert werden, wenn der Netzgraph kreuzungsfrei gezeichnet werden kann (wenn er **planarisierbar** ist).

Wenn alle Netze Zweipunkt-Netze sind, das bezeichnet man die Verdrahtung in einer Ebene als *river routing*. Leitungen in diesem Fall wie Höhenlinien auf Landkarten.

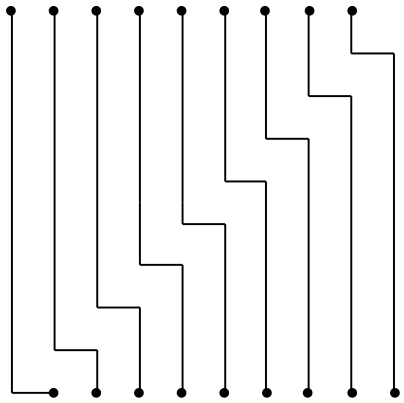


Abbildung 5.41: River-Routing (besonders ungünstiger Fall)

Als Nächstes betrachten wir einen Speziallfall der Verdrahtung in zwei Ebenen: die Kanalverdrahtung.

5.3.2 Kanalverdrahtung in zwei Ebenen

5.3.2.1 Problemstellung

Aus den vorangegangenen Abschnitten wird klar, dass die Verdrahtung innerhalb von Rechtecken (Kanälen) einen besonders wichtigen Fall darstellt.

Für die Verdrahtung innerhalb von Kanälen gilt das folgende Modell (vgl. Abb. 5.42):

- Ein Kanal ist ein Rechteck, das frei von a priori blockierten Flächen ist.
- Die Anschlüsse der Netze mögen an zwei gegenüberliegenden Seiten des Rechtecks liegen. Ohne Beschränkung der Allgemeinheit stellen wir diese beiden Seiten im folgenden in Abbildungen als obere und untere Seite dar. Die Anschlüsse werden in Abbildungen mit der Nummer des jeweiligen Netzes beschriftet. Nicht beschaltete Anschlüsse werden dabei mit 0 beschriftet.
- Die Länge der beiden anderen Seiten ist (in der Regel) variabel und sollte so klein wie möglich gewählt werden[12].

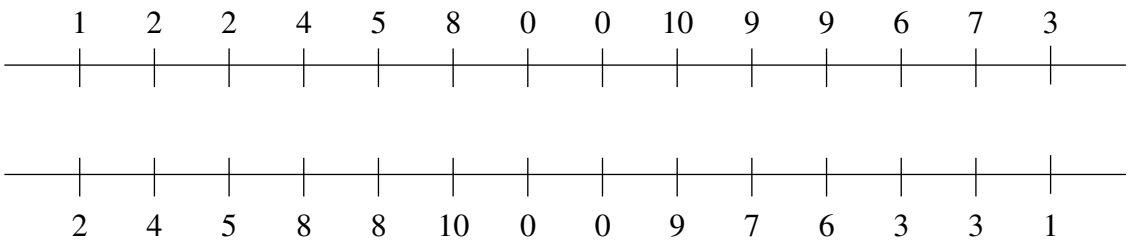


Abbildung 5.42: Beispiel zur Kanalverdrahtung

Dieser Spezialfall des allgemeinen Verdrahtungsproblems wird als **Kanalverdrahtung** (engl. *channel routing*) bezeichnet.

Soweit nicht anders angegeben, wird im Folgenden eine 2–Lagen–Verdrahtung angenommen. Davon wird jede Lage ausschließlich für die Verdrahtung in je einer Richtung (vertikal bzw. horizontal) benutzt.

Das jeweilige Verdrahtungsproblem wird üblicherweise in zwei Arrays Top und Bottom kodiert. Diese enthalten als i–te Komponente die Nummer des Netzes des i–ten Anschlußpunktes der oberen bzw. der unteren Seite.

Beispiel: Für das Problem der Abb. 5.42 [Bur86] gilt:

Top [1..14] = 1,2,2,4,5,8,0,0,10,9,9,6,7,3
Bottom[1..14] = 2,4,5,8,8,10,0,0,9,7,6,3,3,1

Zur Beschreibung zulässiger Verdrahtungen werden insgesamt zwei Graphen definiert. Zunächst werde der *horizontal constraints graph* (HCG) betrachtet:

Def. 19: Eine Spalte heißt **von einem Netz belegt**, wenn die Spalte im Intervall zwischen dem äußersten linken und dem äußersten rechten Anschluß des Netzes enthalten ist.

Beispiel:

Für das Problem der Abb. 5.42 werden die Intervalle in Abb. 5.43 betrachtet.

Def. 20: Der *horizontal constraints graph* (HCG) ist ein ungerichteter Graph, der für jedes Netz genau einen Knoten enthält. Eine Kante zwischen zwei Knoten i und j existiert genau dann, wenn eine Spalte existiert, die von beiden Netzen belegt wird.

Per Definition ist der HCG damit ein Intervallgraph, der wie folgt definiert wird:

Def. 21: ([Gol80]): Ein Graph $G = (V, E)$ heißt Intervallgraph $\Leftrightarrow$
Es existiert eine eineindeutige Abbildung der Knoten $v \in V$ auf Intervalle einer linear geordneten Menge derart, dass Knoten genau dann mit einer Kante $e \in E$ verbunden sind, wenn sich die Intervalle schneiden.

Die Abb. 5.44 enthält den HCG für das Beispiel aus Abb. 5.42.

[12] Viele Algorithmen sind in der Lage, Anschlüsse an einer der beiden Seiten (in den folgenden Abbildungen an der "linken" Seite) ohne Zusatzaufwand ebenfalls zu verarbeiten.

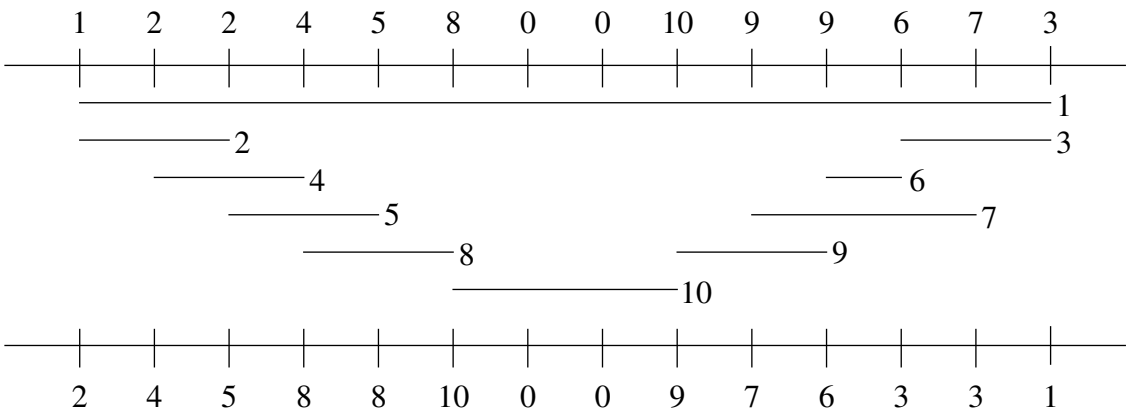


Abbildung 5.43: Intervalle der Netze für obiges Beispiel

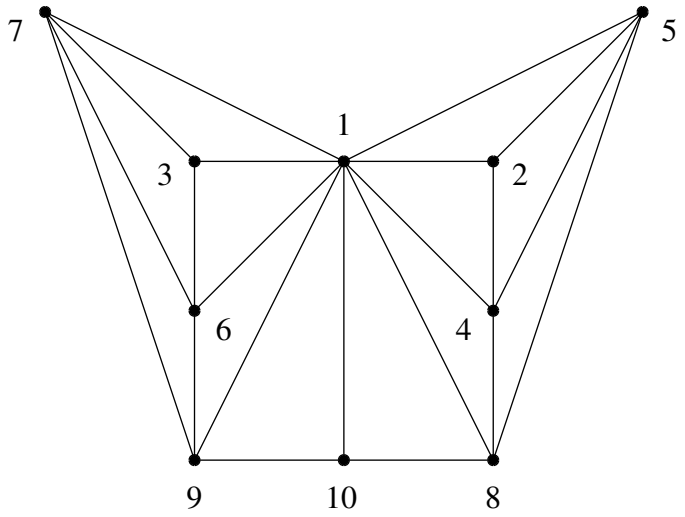


Abbildung 5.44: HCG für obiges Beispiel

5.3.2.2 *Restrictive routing* und der *left edge*-Algorithmus

Falls pro Netz maximal ein horizontales Segment im Kanal erlaubt wird, spricht man von *restrictive routing*. Ursprünglich wurde eine Einschränkung auf diesen Fall für sinnvoll gehalten.

Den horizontalen Segmenten der Netze müssen bestimmte **Spuren** (engl. *tracks*) innerhalb des Kanals zugeordnet werden. Sofern zwei oder mehr Segmente ausschließlich verschiedene Spalten belegen, können sie derselben Spur zugeordnet werden.

5.3.2.2.1 Vernachlässigung der vertikalen Segmente

Zunächst werde das Problem der Spurzuordnung unabhängig von den benötigten vertikalen Segmenten zu den Anschlußpunkten betrachtet.

Die Zuordnung von Spuren muß so erfolgen, dass je zwei Netze, die im HCG eine gemeinsame Kante besitzen, verschiedenen Spuren zugeordnet werden.

Def.: Gegeben sei ein Graph $G = (V, E)$. Das Problem, eine Abbildung der Knoten V auf eine minimale Anzahl von Farben zu finden derart, dass je zwei über eine Kante verbundene Knoten eine verschiedene Farbe haben, heißt **Färbungsproblem**.

Satz: Das allgemeine Färbeproblem ist NP-hart.

Ersetzen wir die Nummern der Spuren im HCG durch “Nummern von Farben”, so sehen wir, dass wir

es hier mit einem Färbungsproblem für Intervallgraphen zu tun haben. Es ist bekannt [Gol80], dass sich Intervallgraphen in linearer Zeit optimal färben lassen.

Zur Kanalverdrahtung wurde der sog. *left edge*–Algorithmus, entwickelt.

Seien:

`first[i]`	Die Menge Netze mit linkem Rand bei Spalte `i`
`last[i]`	Die Menge Netze mit rechtem Rand bei Spalte `i`
`imax`	Die Anzahl der Spalten
`kmax`	Eine obere Schranke für die Anzahl der Spuren
`assign[j]`	Dem Netz `j` zugeordnete Spur.

Wir präsentieren den *left edge*–Algorithmus hier in einer modifizierten Form, bei der die Spalten von links nach rechts durchlaufen werden. Beginnt in einer Spalte ein Netz, so ordnet er eine freie Spur zu. Endet ein Netz, so gibt er die zugeordnete Spur frei:

```
Free := {1..kmax}                    % alle Spuren sind frei
FOR i:= 1 to imax DO                 % Schleife über Spalten
 BEGIN
  FOR EACH j IN first[i] DO
   BEGIN
    k := das kleinste Element aus Free;
    assign[j] := k;                  % Zuordnung Netz -> Spur
    Free := Free - {k};              % Spur ist belegt
   END;
  FOR EACH j IN last[i] DO
   Free := Free + assign[j];         % gebe Spur frei
 END;
```

Der *left edge*–Algorithmus benutzt nie ein Element `k` unnötig. Der größte benutzte Wert `k` entspricht der Anzahl der maximal in einer Spalte vorkommenden Netze. Eine bessere Lösung gibt es nicht, der Algorithmus liefert also stets eine Lösung mit der minimal möglichen Anzahl von Spuren. Unter der Annahme, dass die Anzahl der in einer Spalte beginnenden Netze konstant ist, bezeichnet man den Algorithmus als linear (in der Anzahl der Spalten).

Als Ergebnis für das Beispiel der Abb. 5.42 ergibt sich eine Zuordnung mit 4 benötigten Spuren (siehe Abb. 5.45).

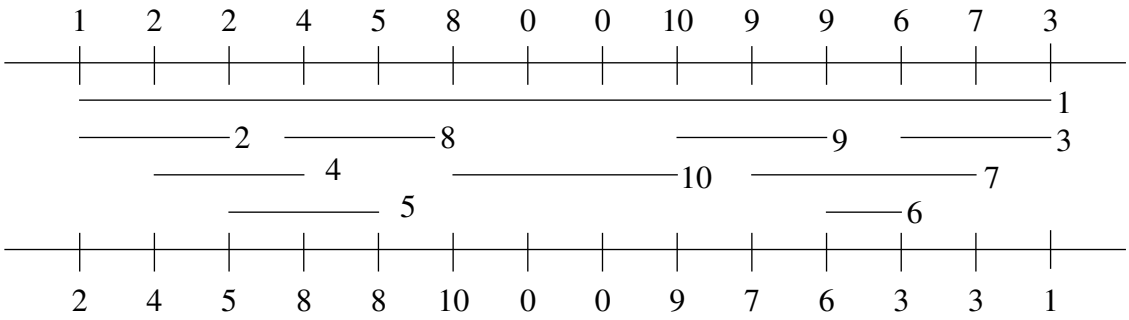


Abbildung 5.45: Ergebnis der Spurzuordnung mit dem *left edge*-Algorithmus

Die Zahl der benötigten Spuren wird als *channel density* bezeichnet. Diese Zahl ist eine untere Schranke für die Zahl von Spuren, die für das allgemeine Problem (d.h. unter Berücksichtigung der vertikalen Segmente) notwendig sind.

5.3.2.2.2 Berücksichtigung der vertikalen Segmente

Sofern drei Verdrahtungsebenen zur Verfügung stehen, kann das Ergebnis des *left edge*–Algorithmus unmittelbar realisiert werden: für die horizontalen Segmente, die vertikalen Segmente zur oberen Seite und die vertikalen Segmente zur unteren Seite steht dann je eine Verdrahtungsebene zur Verfügung.

Bei zwei Verdrahtungsebenen dürfen sich die vertikalen Segmente nicht überlappen. Diese Beschränkung wird durch den *vertical constraints graph* (VCG) modelliert:

Def. 22: Der *vertical constraints graph* (VCG) ist ein gerichteter Graph, der für jedes Netz genau einen Knoten enthält. Eine Kante von Knoten i zum Knoten j existiert genau dann, wenn eine Spalte k existiert mit $\text{Top}[k] = i$ und $\text{Bottom}[k] = j$.

Der VCG modelliert also die Relation "horizontales Segment des Netzes i muß oberhalb des horizontalen Segments des Netzes j liegen". Falls der VCG azyklisch ist, beschreibt er eine partielle Ordnung. Die Segmente können dann vertikal in einer totalen Ordnung angeordnet werden, die mit der partiellen Ordnung des VCG (sowie dem HCG) verträglich ist. Den Vorgang der Bestimmung einer solchen Ordnung nennt man **topologisches Sortieren** (eng. *topological sort*, siehe z.B. [AHU74]). Im azyklischen Fall ist die Länge des längsten Weges gleichzeitig eine untere Schranke für die *channel density*.

Enthält der VCG Zyklen, ist eine vertikale Anordnung ohne Überschneidung der vertikalen Segmente nicht möglich.

Beispiel: Der VCG zur Abb. 5.42 ist in Abb. 5.46 dargestellt.

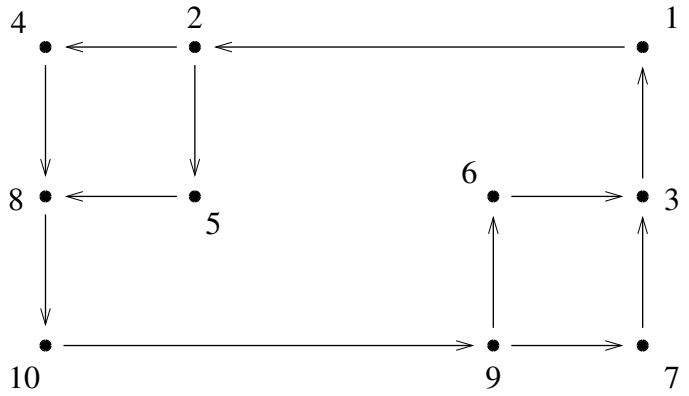


Abbildung 5.46: VCG des obiges Beispiels

Optimales *restrictive routing* unter Berücksichtigung der vertikalen Segmente ist NP–hart (bewiesen in [LaP80], zitiert nach Burstein [Bur86]).

Als heuristische Lösung kann der *left edge*–Algorithmus um eine Berücksichtigung der vertikalen Segmente erweitert werden. Beim Durchlauf von links nach rechts werden dann jeweils nur noch solche Spuren zugeordnet, die mit dem *vertical constraints*–Graphen verträglich sind. Abb. 5.47 zeigt das Ergebnis des Beispiels nach Abb. 5.42.

Wie aufgrund des Zyklus im Graphen der Abb. 5.46 zu erwarten war, ist eine Verdrahtung mittels des *left edge*-Algorithmus nicht möglich. Neben einer Erweiterbarkeit über den rechten Rand hinaus wäre für das Netz 3 ein zweites horizontales Segment (siehe gestrichelte Linie in Abb. 5.47) erforderlich. Zur Verdrahtung wären dann 11 Spuren notwendig.

Eine Verdrahtung ist auch im zyklenbehafteten Fall möglich, sofern pro Netz und Spalte mindestens 2 horizontale Segmente zugelassen sind und sofern ggf. zusätzliche Spalten außerhalb des Kanals zur Verbindung der horizontalen Segmente untereinander zur Verfügung stehen.

5.3.2.3 *Dogleg*-Kanalverdrahtung nach Deutsch

Def.: Vertikale Segmente zur Überbrückung zweier horizontaler Segmente heißen *doglegs*. Das entsprechende Verdrahtungsproblem heißt *dogleg channel routing* und ist ebenfalls NP–hart.

Zur Reduktion der Komplexität werden beim ersten Dogleg–Router von Deutsch [Deu76] die Doglegs begrenzt:

- Doglegs werden nur in Spalten mit Netzanschlüssen erlaubt.

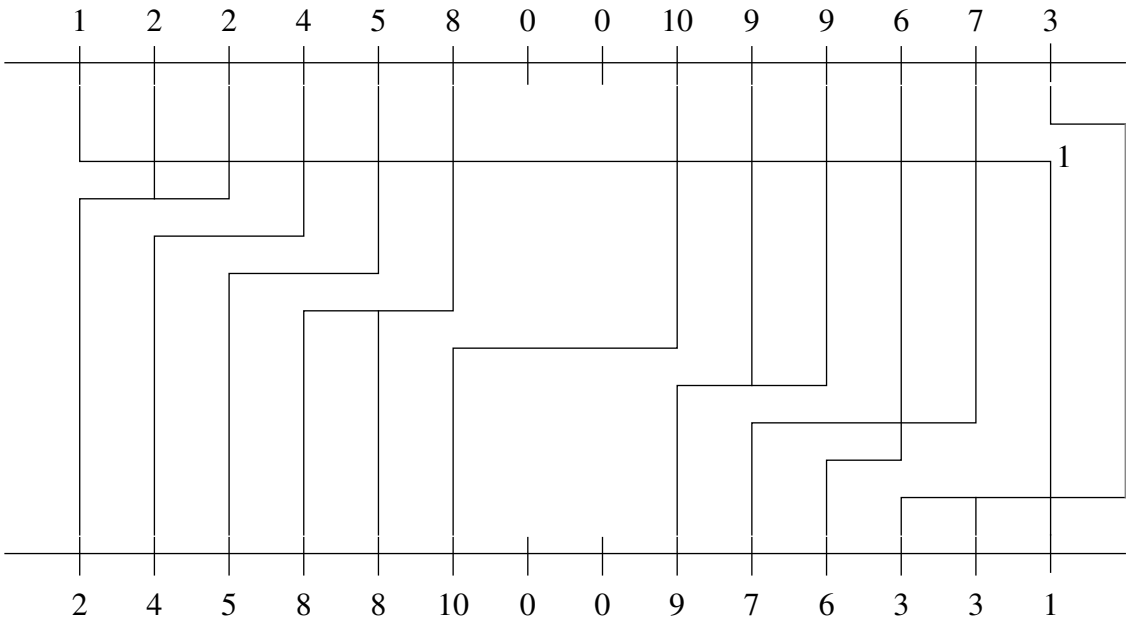


Abbildung 5.47: Spurzuordnung mit dem *constrained left edge*-Algorithmus

- Der Abstand aufeinanderfolgender Doglegs ist stets größer als eine Systemkonstante.
- Pro Spalte und Netz existiert höchstens ein horizontales Segment.

Die erste Einschränkung erlaubt es, n–Punkt–Netze in $n-1$ 2–Punkt–Netze zu zerlegen und getrennt zu verdrahten. Zwar können mit dem Dogleg–Router von Deutsch einige Zyklen aufgebrochen werden, allgemein ist dies aber wegen der dritten Einschränkung nicht möglich. Abb. 5.48 (nach [Bur86]) zeigt je ein Beispiel für einen zu lösenden und einen nicht zu lösenden Zyklus.

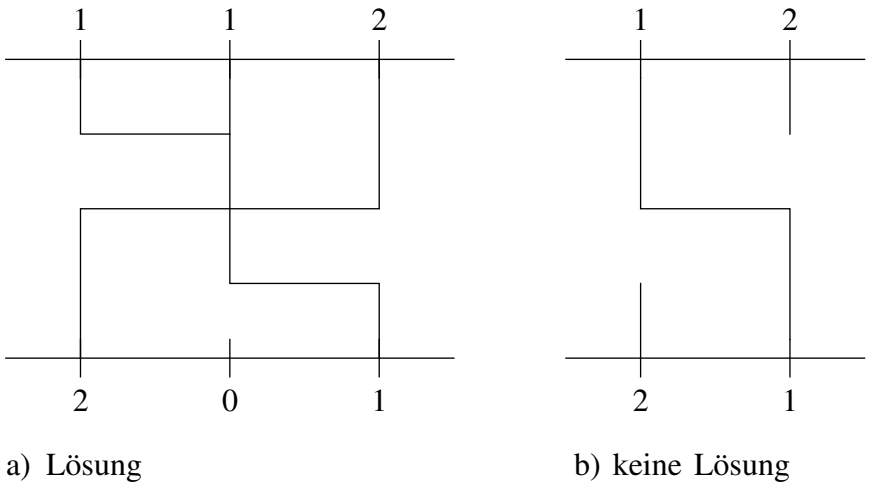


Abbildung 5.48: Verdrahtung bei zyklischen vertikalen Randbedingungen

Um Zyklen allgemein aufbrechen zu können, wird nunmehr die Beschränkung auf ein horizontales Segment pro Netz und Spalte aufgehoben (siehe z.B. Abb. 5.49).

5.3.2.4 *Greedy channel router* von Rivest und Fiduccia

Der *greedy channel router* von Rivest und Fiduccia [RF82] erlaubt eine beliebige Zahl horizontaler Segmente pro Spalte und Netz. Damit ist es möglich, stets eine Lösung zu generieren. Die Verdrahtung schreitet vom

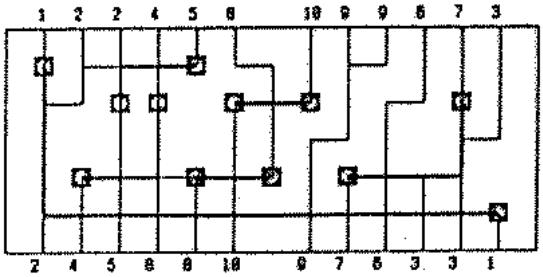


Fig. 15. Burstein's channel.

Abbildung 5.49: Kanalbeispiel nach Burstein (©IEEE)

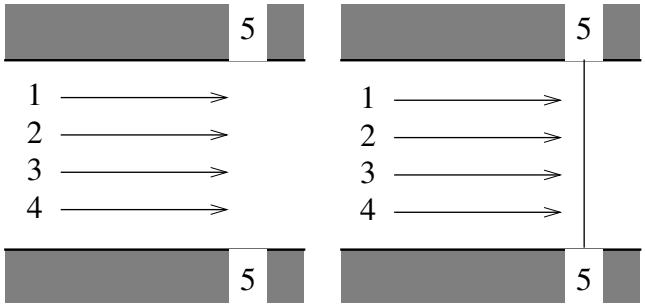
linken zum rechten Kanalrand fort. Falls notwendig, werden mehrere horizontale Segmente pro Netz erzeugt und später vertikal verbunden. Dazu können auch vertikale Segmente jenseits des rechten Kanalrandes benutzt werden. Im Folgenden wird der Algorithmus zusammen mit Beispielen erklärt. Höchste Priorität hat zunächst einmal die Verbindung mit den Anschlüssen (siehe "erzeuge mögliche Top- und Bottom-Verbindungen" in Abb. 5.50).

Ist ein Netz sowohl am oberen als auch am unteren Kanalrand anzuschließen, so erfolgt dies als erstes. Im zweiten Fall der Abb. 5.50 sollen **beide** Anschlüsse mit horizontalen Segmenten im Kanal verbunden werden. Sofern dies ohne Überschneidung der vertikalen Verdrahtung möglich ist, werden beide Netze verbunden. Im Falle von Überschneidungen wird die kürzeste Verbindung realisiert. Im dritten Fall der Abb. 5.50 ist **genau ein** Anschluß in den Kanal zu führen. Dieser Anschluß wird an die nächste Spur geführt, die entweder frei oder von dem gerade betrachteten Netz belegt ist. Im Beispiel ist die freie Spur die nächste Spur und es wird nicht mit der untersten Spur verbunden, obwohl diese ein Segment des Netzes 1 enthält. Ist weder am oberen noch am unteren Anschluß eine Verbindung erforderlich, so wird der für vertikale Segmente freie Platz zur Vereinigung von Netzen genutzt.

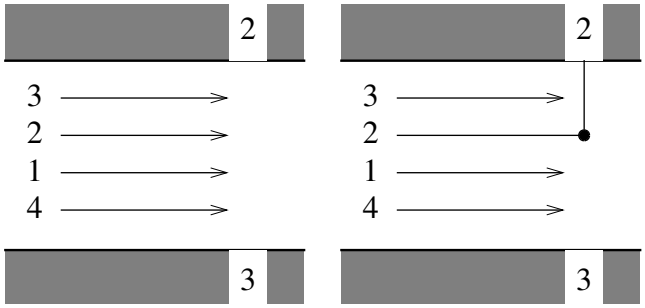
Als nächstes wird geprüft, ob in der für vertikale Segmente genutzten Ebene noch Platz ist, um geteilte Netze möglichst nahe aneinander rücken zu lassen (siehe Abb 5.51).

Ist weiterhin noch Platz frei, so werden Netze unter gewissen Bedingungen vorsorglich in Richtung auf ihren nächsten Anschluß hin verschoben. Nachdem jetzt aller Platz in der vertikalen Verdrahtungsebene ausgenutzt ist, muß noch dafür gesorgt werden, dass die Anschlüsse der gegenwärtigen Spalte verdrahtet werden, falls dies ohne neue Spuren nicht möglich war. Zu diesem Zweck werden bei Bedarf neue Spuren erzeugt. Im letzten Schritt werden die horizontalen Segmente für den Übergang auf die nächste Spalte erzeugt.

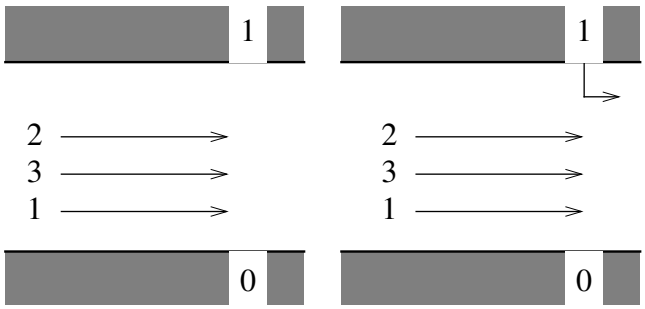
Als Benchmark für das Channel Routing dient das "schwierige Beispiel von Deutsch". Abb. 5.52 und Abb. 5.53 zeigen eine Lösung dieses Beispiels mit dem Greedy Channel Router. Die geteilten Netze und die vorsorglichen Spurwechsel nach oben und unten sind mehrfach vorhanden.



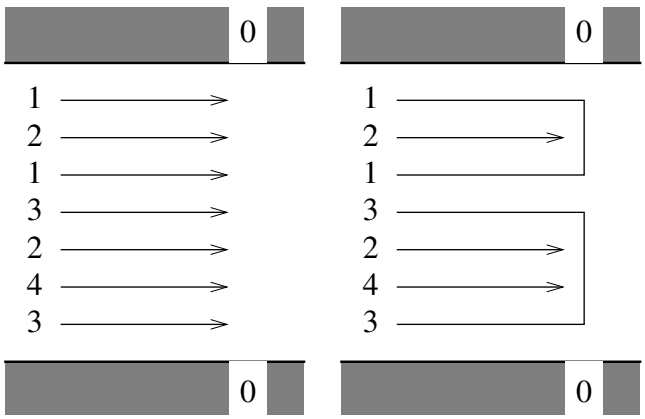
i:=1; (* Spaltennummer *)
REPEAT (* von links nach rechts *)
Erzeuge mögliche Top- und Bottom-Verbindungen:
1: Falls (Top[i]=Bottom[i]≠0)
dann verbinde Top und Bottom.



2: Falls (Top[i]≠Bottom[i]) und
(Bottom[i]≠0) und
es existiert ein Konflikt,
dann erzeuge die kürzeste Verbindung.



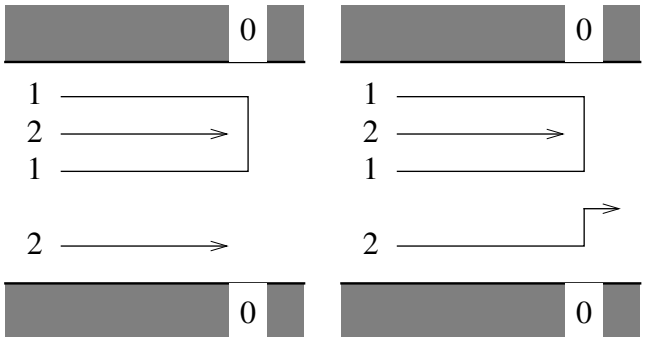
3: Falls (Top[i]≠0) oder
(Bottom[i]≠0)
dann bringe das Netz an diejenige nächste Spur, die entweder frei oder bereits von Netz belegt ist. (Hier wird nicht mit der belegten Spur verbunden!)



Vereinigung von geteilten Netzen

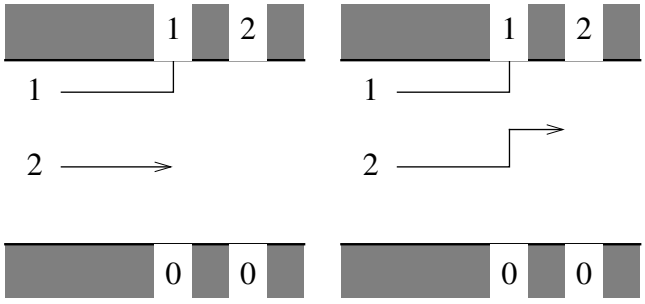
Vollständige Suche nach Dogles, die den größten Gewinn erbringen.
Gewinn: Pro vereinigtem Netz eine Spur sowie für jedes Netz, das beendet wird eine weitere Spur.

Abbildung 5.50: Greedy Channel Router: Anschluß-Verbindungen und Netzvereinigung



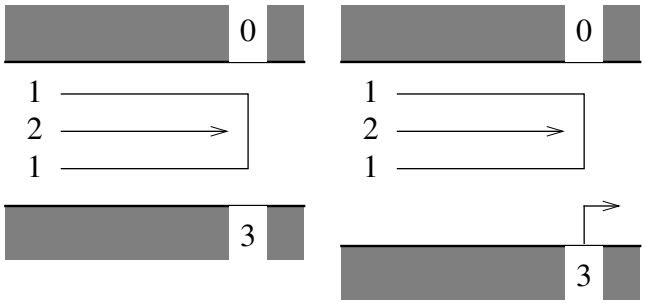
Annäherung von geteilten Netzen

Äußere Spuren von geteilten Netzen rücken soweit wie möglich in die Mitte.
(Vorbereitung der Vereinigung)

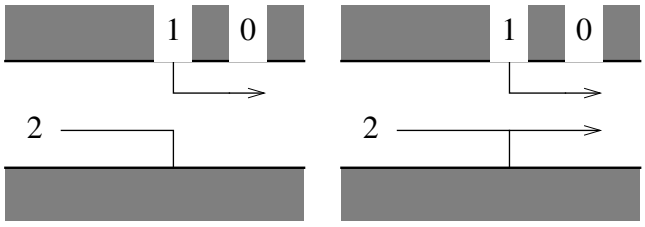


Annäherung an den nächsten Anschluß

Dogleg nach oben, wenn der nächste Anschluß des Netzes sich oben befindet un in den nächsten k Spalten (k: Systemkonstante) kein Anschluß unten existiert. Entsprechend für unten.



Kanalverbreiterung
Verbreitere den Kanal, falls Top- oder Bottom-Anschluß nicht möglich war.



Erzeuge horizontale Segmente für den Übergang zur nächsten Spalte;
i:=i+1;
UNTIL (i>rechter Kanalrand) und
es existieren keine geteilten Netze.

Abbildung 5.51: Greedy Channel Router: Annäherung und Kanalverbreiterung

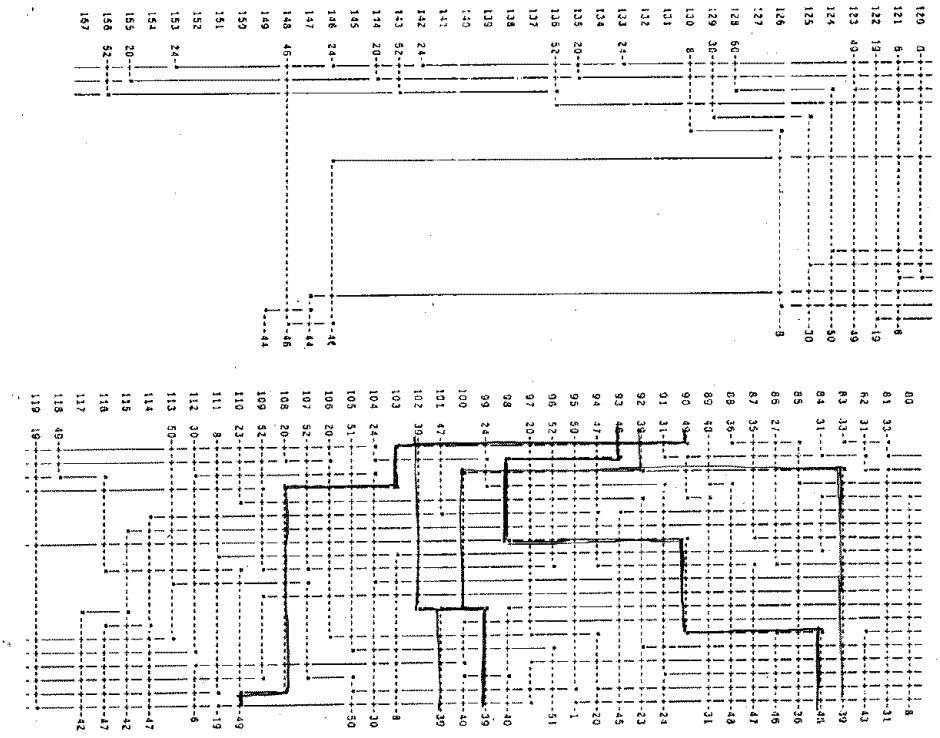


Abbildung 5.52: Lösung des schwierigen Beispiels von Deutsch mit dem Greedy Channel Router (20 Spuren; optimale Lös.:19 Spuren; ©1982 IEEE)

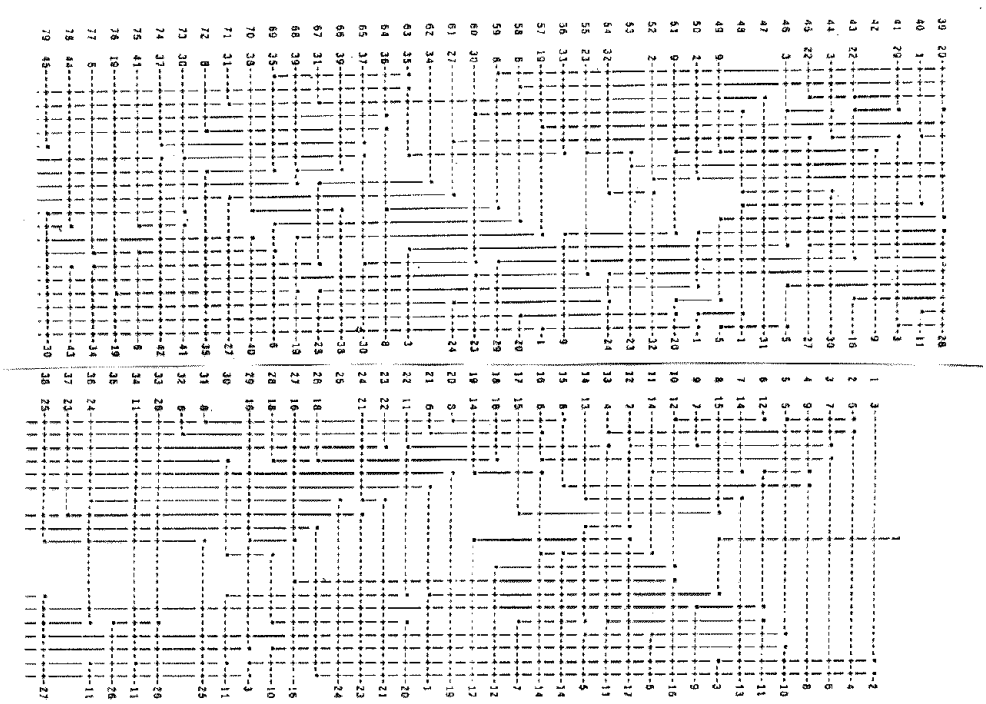


Abbildung 5.53: Beispiel von Deutsch (Fortsetzung; ©1982 IEEE)

5.3.2.5 Switchbox-Routing (in zwei Ebenen)

Als Variante des Kanalverdrahtungsproblems hat das *switchbox routing* eine besondere Bedeutung erlangt. Beim *switchbox routing* [DPT90]) befinden sich an allen 4 Seiten Anschlüsse. Eine Kanalverbreiterung ist dann nicht mehr möglich (siehe Abb. 5.54 und Abb. 5.55).

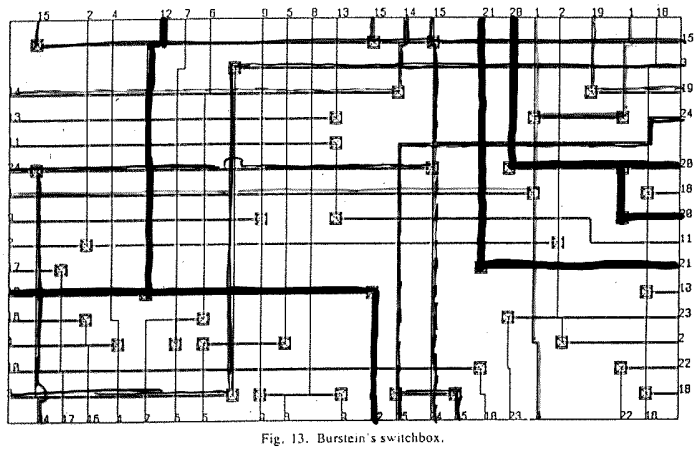


Abbildung 5.54: *Switchbox* nach Burstein (©IEEE)

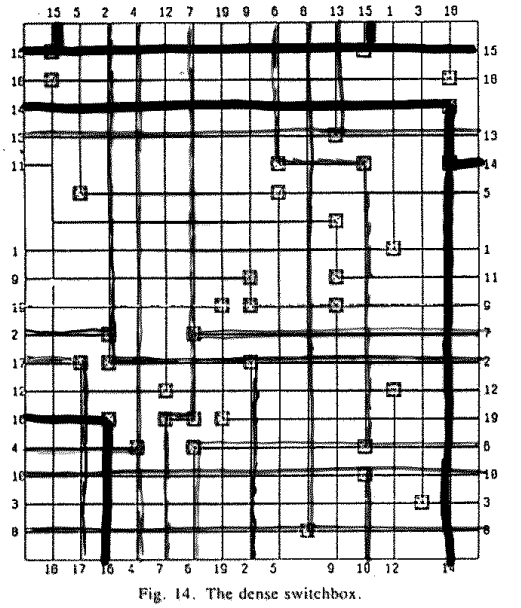


Abbildung 5.55: Sog. 'dichte' *Switchbox* (©IEEE)

5.3.2.6 Verdrahtung in drei Ebenen

Mit der Zunahme der Metallebenen in integrierten Schaltkreisen enstand auch der Bedarf nach Algorithmen zur Verdrahtung in mehr als zwei Ebenen.

Hier kann man zwischen zwei Klassen von Verfahren unterscheiden:

1. Verfahren ohne reservierte Ebenen
2. Verfahren mit reservierten Verdrahtungsebene.

   Beispielsweise verwendet man bei der VHV-Kanalverdrahtung zwei vertikale und eine horizontale Ebene und bei der HVH-Verdarhtung ist es genau umgekehrt.

Die Abbildung 5.56 zeigt die unterschiedlichen Kanalbreiten für insgesamt drei verschiedenen Verfahren.

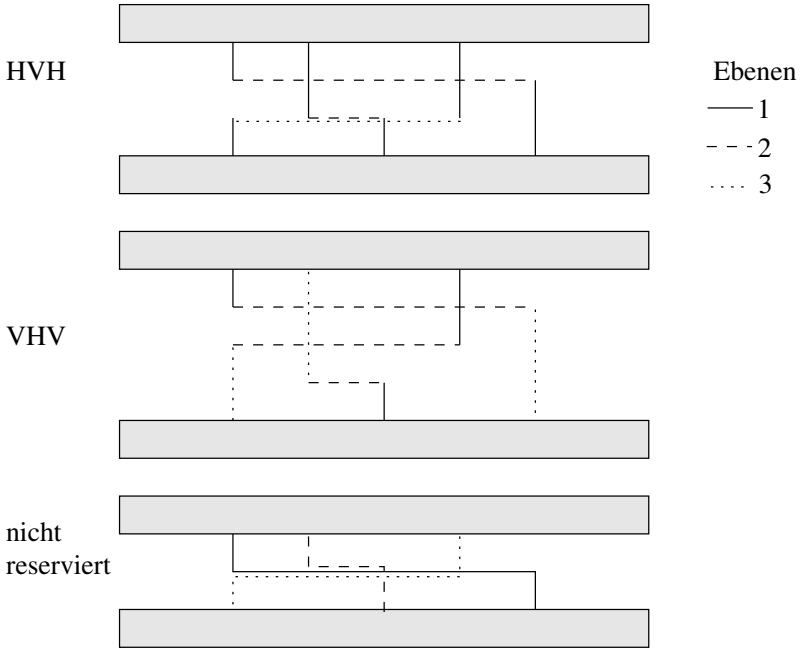


Abbildung 5.56: Kanalverdrahtung in drei Ebenen

5.3.3 Der Lee-Algorithmus

5.3.3.1 Ursprüngliche Form

Im Folgenden sollen noch einige der klassischen älteren Algorithmen vorgestellt werden, die zur Verdrahtung in beliebigen Verdrahtungsregionen geeignet sind.

Der Lee–Algorithmus [Lee61] ist der erste Algorithmus, der speziell zur Verdrahtung elektronischer Schaltungen entwickelt wurde. Ziel des Algorithmus ist das Finden eines kürzesten Weges in einem **Labyrinth** (engl. *maze*) zwischen zwei Punkten A und B.

In der ursprünglichen Form ist der Algorithmus auf die Verdrahtung innerhalb einer Ebene begrenzt. Er setzt eine Rasterung dieser Ebene voraus. Durch die Rasterung wird garantiert, dass die minimal erlaubten Abstände zwischen Leitungen eingehalten werden.

Der Lee–Algorithmus läßt sich wie folgt formulieren (vgl. Abb. 5.57 bis 5.59):

1. Wähle einen der beiden Punkte A bzw. B als Startpunkt aus. Der andere Punkt werde als Zielpunkt bezeichnet. Markiere den Startpunkt mit 0. Setze $i := 0$;
2. REPEAT
   Unmarkierte Nachbarn von Feldern, die mit dem aktuellen Wert von i markiert sind, werden mit $i + 1$ markiert. Anschließend wird i um 1 erhöht. UNTIL (Zielpunkt erreicht) oder (alle Felder sind markiert).
   Dieser Vorgang heißt **Wellenausbreitung** (engl. *wave propagation*).
3. Kehre über streng absteigende Markierungen vom Ziel zum Start zurück. Falls verschiedene Wege möglich sind, versuche die aktuelle Richtung beizubehalten, um die Zahl der Knicke klein zu halten. Dieser Vorgang heißt *backtrace*.
4. Im letzten Schritt wird der gefundene Weg für weitere Leitungen blockiert und die Marken werden gelöscht. Dieser Vorgang heißt *clearance*.

Die Marken bezeichnen offensichtlich gerade den jeweiligen Abstand vom Startpunkt, falls nur rechtwinkelige Leitungsführung erlaubt ist. Dieses Abstandsmaß heißt **Manhattan–Abstand** (siehe Seite 59). Ein Weg entlang streng absteigender Marken ist also immer auch ein Weg minimaler Länge.

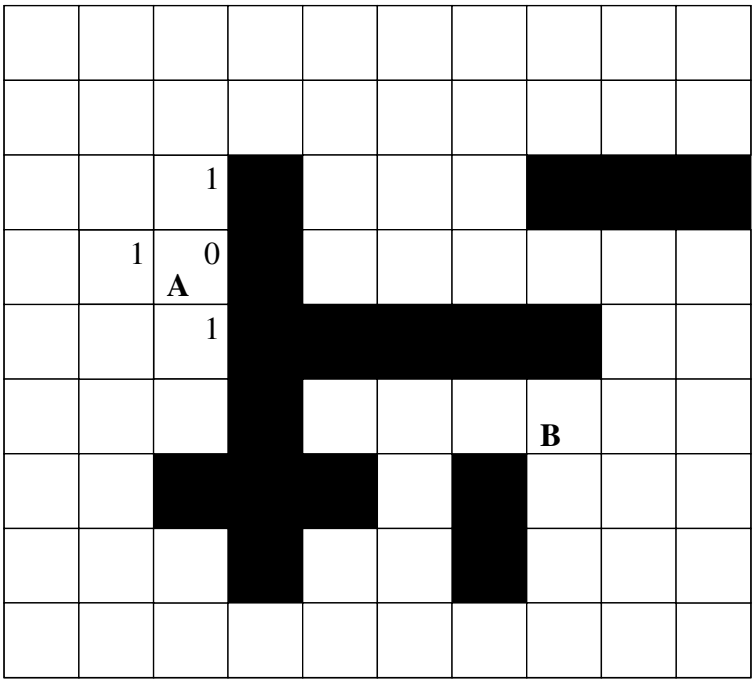


Abbildung 5.57: Markierung mit i=1

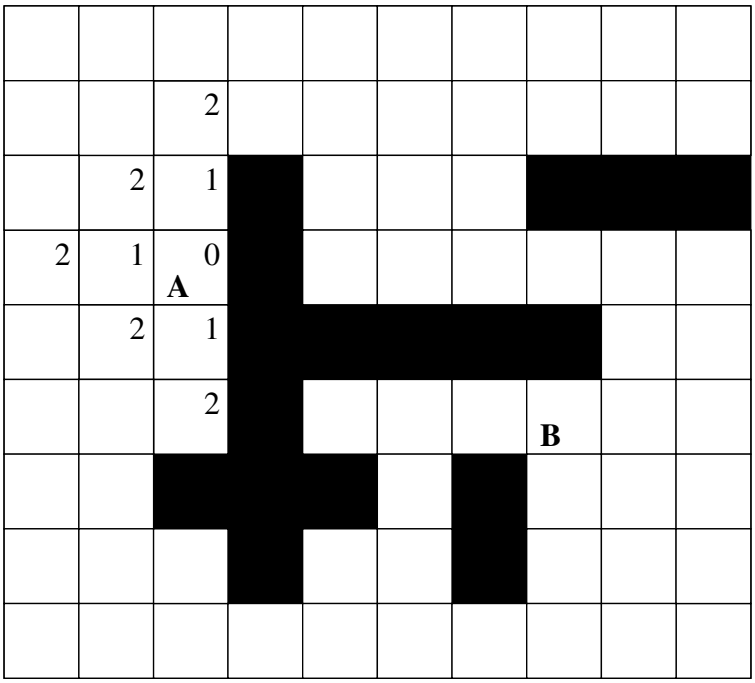


Abbildung 5.58: Markierung mit i=2

Bei einer Kantenlänge von n Feldern benötigt der Lee–Algorithmus $O(n^2)$ Speicherzellen für die Matrix sowie zusätzlich Zellen, um über die aktuelle Wellenfront Buch zu führen. Spezielle Markierungstechniken erlauben eine Reduktion dieses Speicherbedarfs. Die Laufzeit beträgt im schlimmsten Fall $O(n^2)$ für die Wellenausbreitung sowie $O(\ell)$, mit ℓ =Abstand(A, B), für die Backtrace–Phase. Für eine globale Verdrahtung größerer Platinen oder Schaltkreise ist er damit nicht geeignet. Man denke etwa an integrierte Schaltkreise mit 12 mm Kantenlänge und $1,2\mu$m Leitungsabstand.

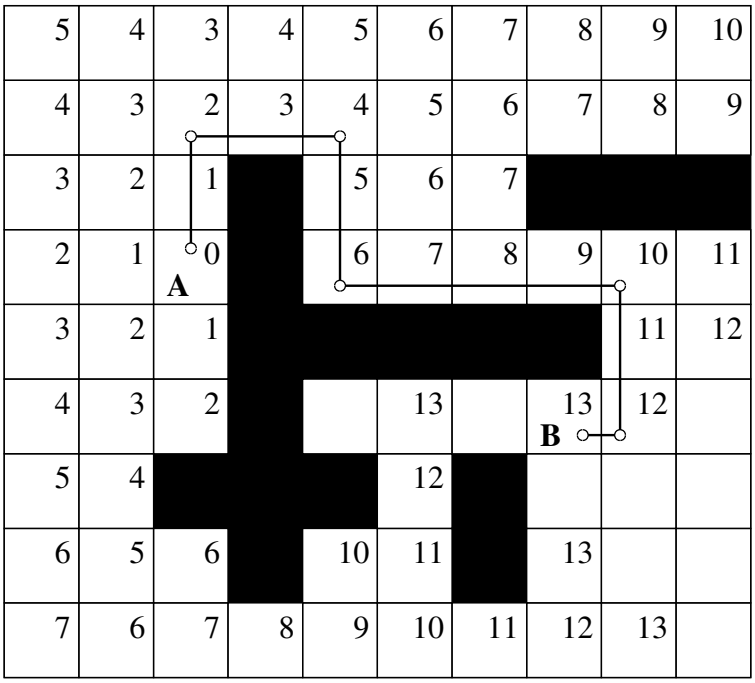


Abbildung 5.59: *backtrace*

5.3.3.2 Erweiterungen

Einfache Methoden der Beschleunigung

- Als Startpunkt wird der Punkt benutzt, der dem Rand näher liegt. Damit brauchen weniger Felder markiert zu werden.
- Man beginnt an beiden Punkten gleichzeitig. Auch damit werden in der Regel weniger Felder markiert.
- Die Markierung wird auf ein Rechteck begrenzt, das A und B einschließt und nur wenig (10-20 %) größer als das minimale umschließende Rechteck ist. Wird hierbei eine Lösung nicht gefunden, so wird in einem zweiten Versuch diese Begrenzung aufgehoben.

Mehrpunktnetze

Für 2–Punkt–Netze liefert der Lee–Algorithmus stets einen Weg mit kürzestem Abstand, sofern dieser existiert. Bei n–Punkt–Netzen wäre dazu ein Steiner–Baum zu konstruieren.

Mehrlagenverdrahtung

Im Prinzip läßt sich der Lee–Algorithmus ins 3–dimensionale übertragen, indem man auf Abstände innerhalb von Quadern übergeht. Allerdings werden die Komplexitätsprobleme dadurch noch größer. Eine Vereinfachung ergibt sich, wenn die Entfernung zwischen den Lagen vernachlässigt wird und wenn man weiterhin annimmt, dass die zu verbindenden Punkte in allen Lagen erreichbar sind. Letzteres ist bei der Verdrahtung von bestückten Platinen erfüllt, bei integrierten Schaltkreisen sind die Kosten für einen Lagenwechsel dagegen recht hoch.

Für jede zu verdrahtende Lage verwendet man ein Tableau für die Blockierungsinformation sowie ein zusätzliches Tableau für die Abstände (der Abstand ist ja in allen Lagen der gleiche).

Vermeidung der Blockierung für folgende Wege

Bislang nimmt das Verfahren keinerlei Rücksicht darauf, wie schwer die Verdrahtung der folgenden Netze sein wird. Generell ist es günstig, die Wege möglichst eng parallel zu existierenden Wegen anzuordnen. Durch Einführung von geeignet gewichteten Rasterpunkten kann man dafür sorgen, dass gewisse Wege bevorzugt werden. Während der Wellenausbreitung wird dann die Pfadlänge nicht mehr um 1, sondern um das Gewicht des jeweiligen Rasterpunktes erhöht.

Trotz seiner Komplexität ist der Lee–Algorithmus sehr beliebt, vermutlich vor allem, weil er sich leicht an unterschiedliche Randbedingungen anpassen läßt. So sind z.B. auch Modifikationen möglich, bei denen eine Ebene vorzugsweise für die horizontale, die andere vorzugsweise für die vertikale Verdrahtung benutzt wird.

5.3.4 Liniensuch-Algorithmen

Auch dann, wenn über relativ große Distanzen ohne Knicke zu verdrahten wäre, sucht der Lee–Algorithmus eine große Zahl von Rasterpunkten ab. Diesen Nachteil vermeiden die **Liniensuch-Algorithmen** (engl. *line search algorithm*). Statt mit Rasterpunkten arbeiten diese Algorithmen zumindest zum Teil mit Linien.

5.3.4.1 Soukup's schneller Labyrinth-Algorithmus

Der schnelle Labyrinth–Algorithmus von Soukup [Sou78] ist eine Kombination von Lee–Algorithmus und dem eigentlichen *line search*–Verfahren, welches später vorgestellt wird.

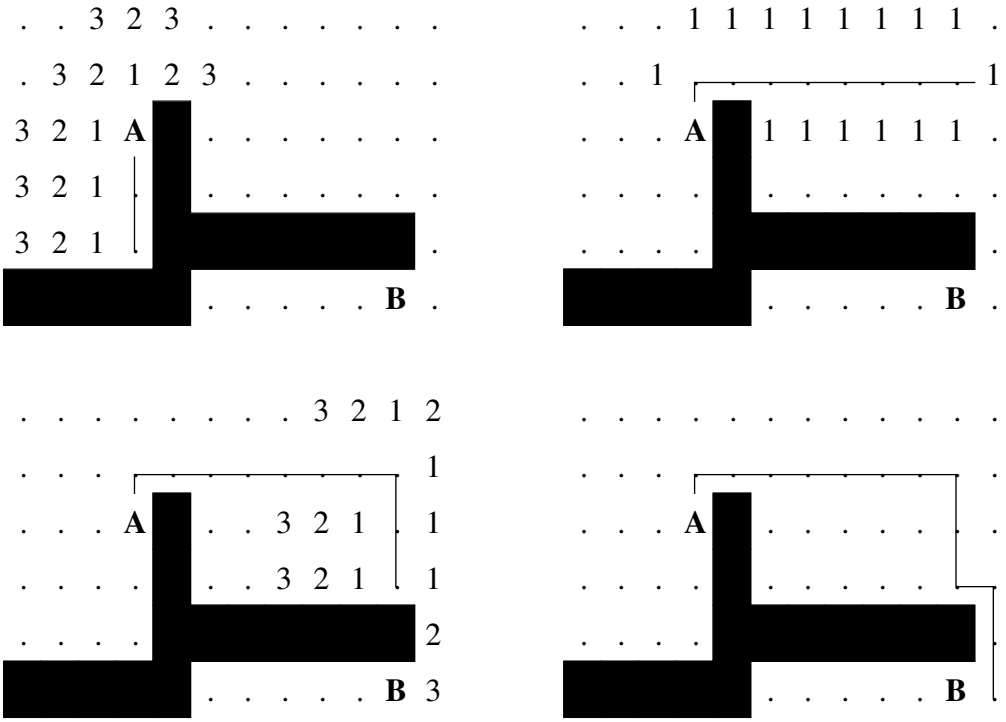


Abbildung 5.60: Beispiel für Soukup's Algorithmus (Zielpunkt ist **B**)

Der Algorithmus generiert zunächst eine Linie vom Ausgangspunkt in Richtung des Zielpunktes. Sofern der Zielpunkt nicht erreicht wird, wird **um diese Linie herum mittels des Lee-Algorithmus ein Rasterpunkt gesucht, der den Abstand zum Zielpunkt gegenüber dem bisherigen Abstand von der Linie zum Zielpunkt verkürzt**. Die Menge der Linien wird sodann um Linien durch diesen Rasterpunkt erweitert. Mit der jeweils zuletzt gefundenen Linie als Menge neuer Startpunkte wird dieser Prozeß wiederholt, bis der Endpunkt erreicht ist (siehe Beispiel in Abb. 5.60).

Falls eine Lösung existiert, wird von diesem Algorithmus auch immer eine Lösung gefunden, da notfalls immer mittels des Lee–Algorithmus gesucht wird. Die gefundene Lösung ist aber nicht notwendig optimal. Laut Soukup soll der Algorithmus für typische Beispiele 10–50–fach schneller als der Lee–Algorithmus sein. Bei sehr engen Platzverhältnissen reduziert sich dieser Geschwindigkeitsvorteil.

5.3.4.2 *Line search*-Verfahren von Mikami und Tabuchi

Die eigentlichen *line search*–Verfahren kommen ohne Lee–Algorithmus aus. Die ersten Verfahren wurden unabhängig voneinander von Mikami und Tabuchi [MT68] und von Hightower [Hig69] entwickelt.

Im Verfahren von Mikami und Tabuchi werden zunächst die senkrechten und waagerechten Linien durch Ausgangs- und Zielpunkte gebildet. Diese 4 Linien heißen **Versuchslinien** der Ebene 0. Sofern sich diese Linien schneiden, ist bereits ein Weg gefunden. Andernfalls werden alle Senkrechten auf den Versuchlinien der Ebene 0 erzeugt. Diese heißen **Versuchslinien** der Ebene 1. Schneiden sich die vom Startpunkt ausgehenden Versuchlinien mit vom Zielpunkt ausgehenden Versuchslinien, so ist ein Weg gefunden. Ansonsten wird der Prozeß mit jeweils wachsender Ebene fortgesetzt.

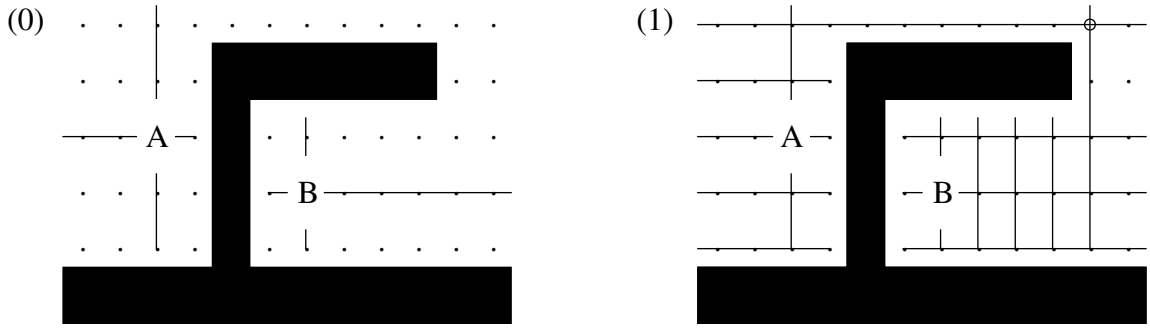


Abbildung 5.61: Zum Liniensuchverfahren von Mikami und Tabuchi

Ein Schnitt von Versuchlinien der Ebene i mit Versuchslinien der Ebene j enthält $i + j + 1$ Knicke. Durch eine geeignete Reihenfolge der Linienbildung kann man dafür sorgen, dass stets mit wachsendem $(i+j)$ auf Schnitt getestet wird. Auf diese Weise wird immer ein Weg mit minimaler Zahl von Knicken gefunden.

Für eine bestimmte Ebene werden die Linien mit dem kleinsten Abstand zu den Ausgangspunkten zuerst betrachtet, um so möglichst auch die Verdrahtungslänge klein zu halten.

Falls eine Lösung existiert, wird sie von diesem Algorithmus auch immer gefunden, sofern wirklich alle Versuchslinien gespeichert werden.

5.3.4.3 *Line search*-Verfahren von Hightower

Beim Liniensuch-Algorithmus von Hightower wird statt aller Senkrechten lediglich eine einzelne Linie gespeichert. Um mit nur einer Linie auszukommen, enthält das Verfahren eine ganze Reihe von Details zur Auswahl dieser Linie. Dennoch kann nicht verhindert werden, dass das Verfahren existierende Lösungen unter Umständen nicht findet. Für einfache Verdrahtungsprobleme ist Hightower's Algorithmus dafür schneller als die bisher vorgestellten Verfahren.

5.3.4.4 Linienerweiterungs-Algorithmus

Der **Linienerweiterungs–Algorithmus** (engl. *line expansion algorithm*) [HSB80] findet stets eine Lösung, falls eine solche existiert. Er ist ebenfalls recht schnell. Man betrachtet dazu zwei Anschlüsse, die miteinander zu verbinden sind. Wir nehmen an, dass diese Anschlüsse am Rand zweier Zellen liegen. Auf dem Rand errichtete Senkrechte, die die Anschlußpunkte enthalten, heißen **Aktivlinien** (siehe Abb. 5.62).

Für die Punkte auf den Aktivlinien wird analysiert, ob eine Expansion seitlich der Linien möglich ist. Alle Flächen, die über Senkrechte auf den Aktivlinien zu erreichen sind, heißen Expansionsflächen. Senkrechte, die die Expansionsflächen begrenzen, werden als weitere Aktivlinien betrachtet. Im Algorithmus werden von beiden Punkten aus neue Aktivlinien erzeugt, bis sich die Expansionsflächen schneiden. Durch Rückwärtsverfolgung kann dann ein Weg zwischen den beiden Punkten bestimmt werden.

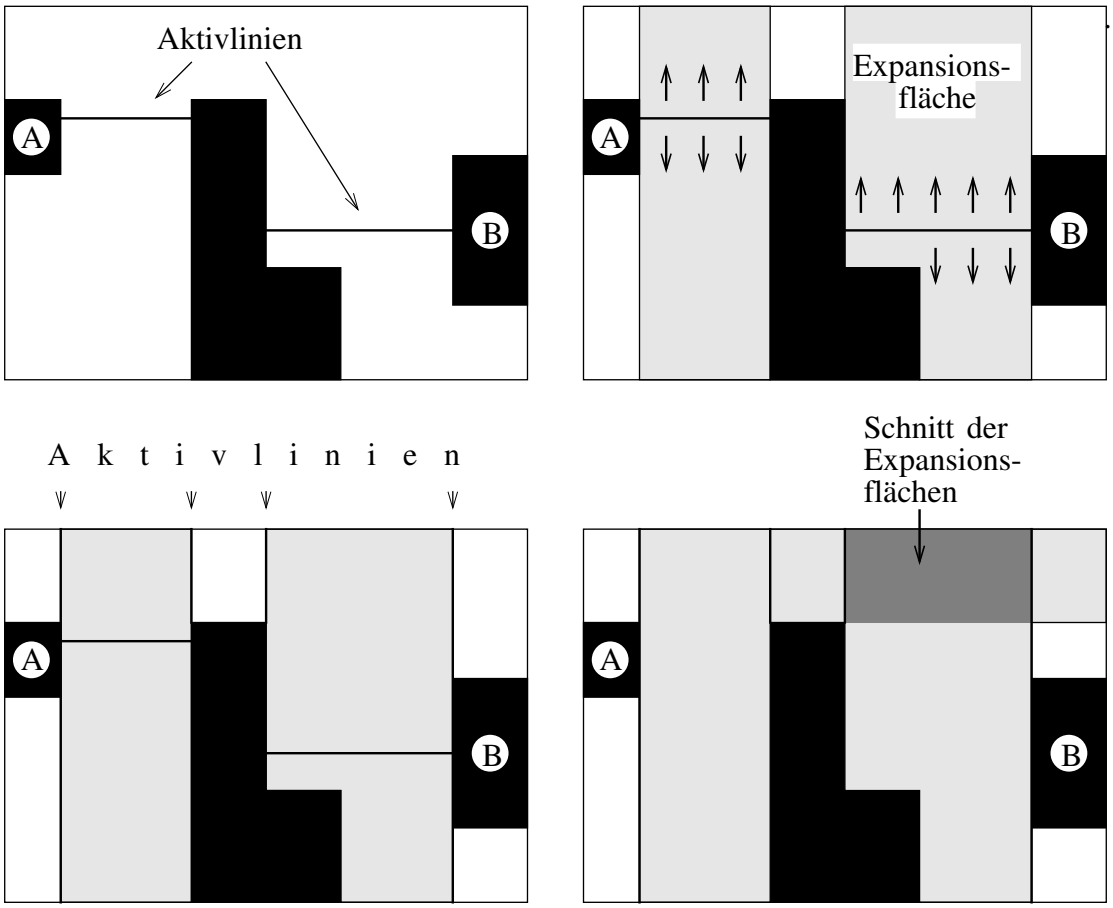


Abbildung 5.62: Zum Linienerweiterungs-Algorithmus

```
REPEAT
  Errichte die Senkrechten auf den vorhandenen Aktivlinien;
  Falls sich ein Schnitt mit den Senkrechten des anderen Punktes
  ergibt, dann STOP.
  Sonst speichere die Begrenzungslinien als mögliche neue Aktivlinien.
UNTIL False;
```

Algorithmus 4.6: Linienerweiterungs-Algorithmus

5.3.5 Spezielle Probleme bei modernen Technologien

Insbesondere bei modernen Technologien mit ihren komplexen Schaltungen und ihren im Verhältnis zum Querschnitt langen Leitungen taucht eine Reihe von Spezialproblemen der Layouterzeugung auf, welche mit den oben beschrieben allgemeinen Methoden nicht gelöst werden können. Beispielsweise sind dies die folgenden Sonderfälle:

1. **Leitungslaufzeiten**

   Bei langen Leitungen kann die Leitungslaufzeit die Laufzeit in den Gattern übersteigen. Hierdurch kann die Taktrate weit unter das gewünschte Minimum gesenkt werden. Bislang werden die Laufzeiten bestimmt, imdem aus dem Layout die Kapazitäten bestimmt und die Netzlisten damit annotiert werden (sog. *back annotation*) . Damit können die Simulatoren dann überprüfen, ob bei einer bestimmten Taktfrequenz noch eine einwandfreie Funktion möglich ist. Besonders problematische Netze können als **kritische Netze** gekennzeichnet und in einer Iteration der Layouterzeugung bevorzugt berücksichtigt werden. Derartige Iterationen waren bei älteren Technologien einige wenige Male durchzuführen. Bei

neueren Technologien wie der 0.18μ-Technologie ist es zunehmend problematisch, überhaupt Konvergenz zu erreichen (die Kennzeichnung einiger Netze führt zu zu langen Leitungen anderer Netze).

2. **Signalverflachung**

   Durch Kapazitäten können die Spannungspegel unzulässig reduziert werden. Mögliche Probleme müssen analysiert und evtl. durch stärkere Treiber korrigiert werden.

3. Betrachtung der *electromigration*

   Bei sehr dünnen Leitungen kann es unter Stromfluss zur Metallwanderung kommen. Eine Folge hiervon ist eine stark verkürzte Lebensdauer der Schaltung. Potentielle Problemfälle müssen analysiert und durch verstärkte Leitungen beseitigt werden.

4. *power and ground routing*

   Die Metallwanderung muss v.a. auch in Bezug auf die Spannungsversorgung beachtet werden. Zusätzlich müssen der erwartete und der zulässige Spannungsabfall miteiander in Beziehung gebracht werden. Für die Spannungsversorgung der einzelnen Teilschaltungen müssen Leitungsbahnen eines größeren Querschnittes erzeugt werden.

5. **Taktverdrahtung**

   Um eine hohe Taktrate zu erreichen, muss der zeitliche Unterschied des Eintreffens der Taktflanken an den verschiednen Teilen der Schaltung möglichst klein sein. Hierfür ist in der Regel eine spezielle Taktverdrahtung erforderlich. Neben starken Leitungstreibern ist auch eine möglichst gleich lange Leitungsführung erforderlich. Eine Technik hierfür ist der H-Baum.

6. **Übersprechen**

   Bei langen Leitungen kann es zum Übersprechen zwischen sehr lang parallel geführten Leitungen kommen. Mögliches Übersprechen muss erkannt und durch Änderung der Geometrie beseitigt werden (während der Layouterzeugung wird dieser Einfluss bsilang nur begrenzt berücksichtigt.

7. **Elektromagnetische Verträglichkeit**

   Bei langen Leitungen kann es auch zu einer starken Empfindlichkeit gegenüber externer elektromagnetischer Strahlung kommen. Auch hiergegen sind geeignete Maßnahmen zu treffen.

8. **Erwärmung der Schaltungen**

   Es ist zu vermeiden, dass einzelne Teile einer Schaltung zu heiß werden. Aus diesem Grund muss die Temperaturverteilung vorhergesagt werden. Hierfür sind spezielle Simulationen erforderlich.

Diese speziellen Probleme werden in dem Buch von Sherwani [She98] angesprochen.

Literaturverzeichnis

[AB89] R. Amann and U. Baitinger. Optimal state chains and state codes in finite state machines. *IEEE Trans. on CAD, Vol.8*, pages 153–170, 1989.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison–Wesley, 1974.

[AHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[AMD83] Advanced Micro Devices AMD, editor. *Bipolar Microprocessor Logic and Interface Data Book.* Sunnyvale, 1983.

[AR76] A.K. Agrawala and T.G. Rauscher. *Foundations of Microprogramming.* Acadamic Press, New York, 1976.

[BHM84] R.K. Brayton, G.D. Hachtel, and C.T. McMullen. Logic minimization algorithms for VLSI synthesis. *Kluwer Academic Publishers*, 1984.

[BLMR87] U. Baitinger, H.M. Lipp, D.A. Mlynski, and K. Reiss. The MEGA system for semi-custom design. *in: G. De Micheli et al.(Hrsg.): Design Systems for VLSI Circuits, Martinus Nijhoff Publishers*, pages 527–540, 1987.

[Bod84] A. Bode. Mikroarchitekturen und Mikroprogrammierung: Formale Beschreibung und Optimierung. *Informatik-Fachberichte, Bd.82, Springer, Berlin*, 1984.

[BR87] R.K. Brayton and R. Rudell. MIS: A multiple-level logic optimization system. *IEEE Trans. on CAD, Vol.6*, pages 1062–1081, 1987.

[Brü94] R. Brück. *Physikalischer Entwurf ....* Hanser-Verlag, 1994.

[Bre77] M.A. Breuer. A class of min–cut placement algorithms. *14th Design Automation Conf.*, pages 284–290, 1977.

[BT88] L. Berman and L. Trevillyan. Improved logic optimization using global flow analysis (extended abstract). *IEEE Int. Conf. on Computer-Aided Design(ICCAD)*, pages 102–105, 1988.

[Bur84] R. E. Burkard. Quadratic assignment problems. *European Journal of Operation Research, Vol.15*, pages 283–289, 1984.

[Bur86] M. Burstein. Channel routing. *in: T. Ohtsuki: Layout Design and Verification, North–Holland*, pages 133–167, 1986.

[Das79] S. Dasgupta. The organization of microprogram stores. *Computing Surveys, Vol. 11*, pages 39–65, 1979.

[DBG84] J. Darringer, D. Brand, and J. Gerbi. Logic synthesis through local transformations. *IBM Journ. of Research & Development*, 1984.

[Deu76] D.N. Deutsch. A dogleg channel router. *13th Design Automation Conf.*, pages 425–433, 1976.

[dGC85] A.J. de Geus and W. Cohen. A rule-based system for optimizing combinational logic. *IEEE Design & Test*, pages 22–32, 1985.

[Dij59] E.N. Dijkstra. A note on two problems in connection with graphs. *Numer. Math., Vol. 1*, pages 269–271, 1959.

[DK85] A.E. Dunlop and B.W. Kernighan. A procedure for placement of standard cell VLSI circuits. *IEEE Trans. on CAD, Vol. 4*, pages 92–98, 1985.

[DN90] S.R. Das and A.R. Nayak. A survey on bit dimension optimization strategies of microprograms. *Proc. 23rd Ann. Workshop on Microprogramming and Microarchitecture*, pages 281–291, 1990.

[dP91] M. Crastes de Paulet. ECIP-ESPRIT 2072. *EURO-Benchmark Letter #2, INPG/CSI, Grenoble*, 1991.

[dPD89] M. Crastes de Paulet and C. Duff. ASYL: A logic and architecture design automation system. *EUROASIC'89*, pages 183–209, 1989.

[DPT90] P. F. Dubious, A. Puissochet, and A. M. Tagant. A general and flexible switchbox router: Carioca. *IEEE Trans. on CAD, Vol. 9*, pages 1307–1317, 1990.

[Flo91a] R. Floren. *Ein inkrementeller Plazierer und Verdrahter zur Flächenabschätzung.* Diplomarbeit, Universität Dortmund, Informatik XII, 1991.

[Flo91b] R. Floren. A note on "a faster approximation algorithm for the Steiner problem in graphs". *Information Processing Letters*, 1991.

[FM82] C.M. Fiduccia and R.M. Mattheyses. A linear–time heuristic for improving network partitions. *19th Design Automation Conf.*, pages 175–181, 1982.

[FR90] G. Franke and S. Rülke. Ein zugang zur synthese digitaler systeme auf hohem abstraktionsniveau. *Informationstechnik* **it**, pages 440–445, 1990.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimizations. *Journal of the ACM*, pages 596–615, 1987.

[GBH80] W. Grass, G. Biehl, and S. Hall. Loge-mat, a program for the synthesis of microprogrammed controllers. *CAD80, Brighton*, pages 543–558, 1980.

[GJ79] M. R. Garey and D. S. Johnson. Computers and intractability. *Bell Labaratories, Murray Hill, New Jersey*, 1979.

[GK83] D.D. Gajski and R.H. Kuhn. New VLSI tools. *IEEE Computer*, pages 11–14, 1983.

[Gol80] M.C. Golumbic. Algorithmic graph theory and perfect graphs. *Academic Press*, 1980.

[Gra82] W. Grass. A depth–first branch and bound algorithm for optimal pla folding. *19th Design Automation Conference*, pages 133–140, 1982.

[Hig69] D.W. Hightower. A solution to the line–routing problem on the continous plane. *6th Design Automation Workshop*, pages 1–24, 1969.

[HK72] M. Hanan and J.M. Kurtzberg. Placement techniques. *in: M.A. Breuer (Hrsg.): Design Automation of Digital Systems, Prentice Hall*, pages 213–282, 1972.

[HKK+90] A. Hetzel, B. Korte, R. Krieger, H.J. Prömel, U. D. Radicke, and A. Steger. Globale und lokale verdrahtungsalgorithmen für sea-of-cell-design. *Informatik Forschung und Entwicklung*, pages 2–19, 1990.

[Hor] A.S. Hornby. *The Advanced learner's dictionary of contemporary English.* Oxford University Press.

[HS66] J. Hartmanis and R.E. Stearns. Algebraic structure of sequential machines. *Prentice-Hall*, 1966.

[HS76] E. Horowitz and S. Sahni. Fundamentals of data structures. *Pitman*, 1976.

[HS81] E. Horowitz and S. Sahni. Algorithmen. *Springer-Verlag*, 1981.

[HSB80] W. Heyns, S. Sansen, and H. Beke. A line–expansion algorithm for the general routing problem with guaranteed solution. *17th Design Automation Conf.*, pages 243–249, 1980.

[KGN87] D.W. Kochetkov, B. Goetze, and W. Nehrlich. An algorithm for the minimum set covering problem. *Preprint P-Math31/87,Akademie der Wissenschaften der DDR, Berlin*, 1987.

[KL70] B.W. Kernighan and S. Lin. A efficient heuristic procedure for partitioning graphs. *Bell Sys. Tech. Journal, Vol.49*, pages 291–307, 1970.

[KLS+88] M. KUNDE, H.-W. LANG, M. SCHIMMLER, H. SCHMECK, and H. SCHROEDER. The instruction systolic array and its relation to other models of parallel computers. *Parallel Computing*, 7:25–39, 1988.

[KMB81] L. Kou, G. Markowsky, and L. Berman. A fast algorithms for Steiner trees. *Acta Informatica*, pages 141–145, 1981.

[Koh87] Z. Kohavi. Switching and finite automata theory. *Tata McGraw-Hill Publishing Company, New Delhi, 9th reprint*, 1987.

[KP87] F.J. Kurdahi and A.C. Parker. REAL: A program for register allocation. *Proceedings of the 24th Design Automation Conference*, pages 210–215, 1987.

[LaP80] A.S. LaPaugh. Algorithms for integrated circuit layout: An analytic approach. *Ph.D. thesis, MIT*, 1980.

[Lau79] U. Lauther. A min-cut placement algorithm for general cell assemblies based on a graph representation. *16th Design Automation Conf.*, pages 1–10, 1979.

[LC84] J.L. Lewandowski and C.L.Liu. A branch and bound algorithm for optimal PLA folding. *21st Design Automation Conference*, pages 426–431, 1984.

[Lee61] C.Y. Lee. An algorithm for path connections and its application. *IRE Trans. on Electronic Computers, Vol.EC–10*, pages 346–365, 1961.

[Len90] T. Lengauer. Combinatorial algorithms for integrated circuit layout. *Wiley-Teubner*, 1990.

[LM85] M. S. Lam and J. Mostow. A transformational model of VLSI systolic design. *IEEE Computer, Vol. 18*, pages 42–52, 1985.

[LM+93] B. Landwehr, P. Marwedel, et al. OSCAR: Optimal constrained simultaneous scheduling, allocation and resource binding in high-level-synthesis. Technical Report 484, (in preparation), Computer Science Dpt., University of Dortmund, 1993.

[LMB90] L. Lavagno, S. Malik, and R. Brayton. MIS-MV: Optimization of multi-level logic with multiple-valued inputs. *Int. Conf. on Computer-Aided Design(ICCAD)*, pages 560–563, 1990.

[LN86] T. Lengauer and S. Näher. An analysis of ternary simulation as a tool for race tetection in digital MOS circuits. *INTEGRATION, Vol.4*, pages 309–330, 1986.

[Mar79] P. Marwedel. The MIMOLA design system: Detailed description of the software system. *Proceedings of the 16th Design Automation Conference*, pages 59–63, 1979.

[Mar86] P. Marwedel. A new synthesis algorithm for the MIMOLA software system. *23rd Design Automation Conf.*, pages 271–277, 1986.

[Mar90] P. Marwedel. Matching system and component behaviour in MIMOLA synthesis tools. *Proc. 1st EDAC*, pages 146–156, 1990.

[Mar93] P. Marwedel. *Synthese und Simulation von VLSI-Systemen.* Hanser, 1993.

[Mar07] Peter Marwedel. *Eingebettete Systeme.* Springer Verlag, 2007.

[MC80] C. Mead and L. Conway. Introduction to VLSI systems. *Addison–Wesley*, 1980.

[Meh88] K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, pages 125–128, 1988.

[Mic87] G. De Micheli. Synthesis of control systems. *in: G. De Micheli et al.(Hrsg.): Design Systems for VLSI Circuits, Martinus Nijhoff Publishers*, pages 571–646, 1987.

[Mol83] D.I. Moldovan. On the design of algorithms for VLSI systems. *Proc. IEEE*, pages 113–120, 1983.

[MR92] P. Marwedel and W. Rosenstiel. Synthese von Register-Transfer-Strukturen aus Verhaltensbeschreibungen. *Informatik-Spektrum, Band 15*, pages 5–22, 1992.

[MRS87] H. De Man, J. Rabaey, and P. Six. CATHEDRAL II: A synthesis and module generation system for multiprocessor systems on a chip. *in: G.DeMicheli, A.Sangiovanni-Vincentelli, P.Antognetti: Design Systems for VLSI Circuits–Logic Synthesis and Silicon Compilation–,Martinus Nijhoff Publishers*, 1987.

[MT68] K. Mikami and K. Tabuchi. A computer program for optimal routing of printed circuit connectors. *IFIPS Proc., Vol.H47*, pages 1475–1478, 1968.

[Oht86] T. Ohtsuki. Layout design and verification. *Advances in CAD for VLSI, Vol. 4, North–Holland*, 1986.

[Par84] A.C. Parker. Automated synthesis of digital systems. *IEEE Design and Test of Computers*, pages 763–776, 1984.

[PB91] M. Pedram and N. Bhat. Layout driven technology mapping. *26st Design Automation Conference*, pages 99–105, 1991.

[PL88] B. Preas and M. Lorenzetti. Physical design automation of VLSI systems. *Benjamin Cummings*, 1988.

[Pus90] D. Pusch. Optimierung von PLAs für Mikroprogrammsteuerwerke. *Diplomarbeit, Inst. f. Informatik &P.M., Univ. Kiel*, 1990.

[Qui84] P. Quinton. Automatic synthesis of systolic arrays from recurrent equations. *11th Ann. Int. Symp. on Computer Architecture*, page 209, 1984.

[RF82] R.L. Rivest and C.M. Fiduccia. A "greedy" channel router. *19th Design Automation Conf.*, pages 418–424, 1982.

[RM86] B. Reusch and W. Merzenich. Minimal coverings of incompletely specified sequential machines. *Acta Informatica*, pages 663–678, 1986.

[RSV87] R.L. Rudell and A. Sangiovanni-Vincentelli. Multiple–valued minimization for pla optimization. *IEEE Trans. on CAD, Vol.6*, pages 727–750, 1987.

[Sch88] B. Schürmann. Hierarchisches top-down chip planning. *Informatik-Spektrum*, pages 57–70, 1988.

[Sec88] C. Sechen. *VLSI Placement and Global Routing Using Simulated Annealing.* Kluwer, Deventer, 1988.

[Sed88] R. Sedgewick. *Algorithms.* Addison-Wesley, 1988.

[She98] N.A. Sherwani. *Algorithms for VLSI Physical Design Automation.* Kluwer Academic Publishers, 3. auflage edition, 1998.

[SK87] P. Suaris and G. Kedem. Quadrisection: A new approach to standard cell layout. *IEEE Int. Conf. on Computer-Aided Design(ICCAD)*, pages 474–477, 1987.

[SL87] C. Sechen and K.-W. Lee. An improved simulated annealing algorithm for row-based placement. *IEEE Int. Conf. on Computer-Aided Design(ICCAD)*, pages 478–481, 1987.

[SM91] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *acm computing surveys, Vol. 23*, pages 143–221, 1991.

[Sou78] J. Soukup. A fast maze router. *15th Design Automation Conf.*, pages 100–101, 1978.

[Tei97] J. Teich. *Digitale Hardware/Software-Systeme.* Springer, 1997.

[Ull84] J.D. Ullman. Computational aspects of VLSI. *Computer Science Press*, 1984.

[vLA87] P.J.M. van Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications*. D. Riedel, Dortrecht, 1987.

[VSV89] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment of finite state machines for optimal two-level logic implementations. *26st Design Automation Conference*, pages 327–332, 1989.

[Zim79] G. Zimmermann. The MIMOLA design system: A computer aided digital processor design method. *Proceedings of the 16th Design Automation Conference*, pages 53–58, 1979.

[Zim86] G. Zimmermann. Top-down design of digital systems. *in: E. Hörbst (ed.): Logic Design and Simulation, Advances in CAD for VLSI, Vol. 2, North Holland*, pages 9–30, 1986.

Index