Begleitmaterial zur Vorlesung
"Rechnergestützter Entwurf / Produktion (Mikroelektronik)"
Wintersemester 1998/99

Peter Marwedel
Informatik 12
(Technische Informatik & Didaktik der Informatik)
Universität Dortmund

5. März 1999

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Gegenstand der Vorlesung

Gegenstand der Vorlesung sind v.a. Entwurfstechniken für den Entwurf mikroelektronischer Systeme. Die Fertigung tritt demgegenüber in den Hintergrund, da die Fertigung mikroelektronischer Systeme nicht zu den Aufgabenfeldern von Informatikern gehört.

Innerhalb der mikroelektronischen Systeme erfolgt in der Vorlesung eine Konzentration auf die sog.

eingebetteten Systeme.

Def.: Eingebettete Systeme sind informationsverarbeitende Systeme[1], die physikalische Größen aufnehmen, verarbeiten und beeinflussen.

In den üblichen Vorlesungen über Programmierung beschäftigt man sich in der Regel mit Programmen, die zu gewissen Eingaben nach etwas Rechenzeit Ausgaben liefern sollen. Letztlich berechnen diese Programme eine *Funktion*. Man denke etwa daran, dass die Diskussion der Berechenbarkeit v.a. von der Aufgabe ausgeht, durch ein Programm eine Funktion berechnen zu lassen. Die Zeit spielt nur insofern eine Rolle, als man die Laufzeit nicht allzu stark wachsen lassen möchte.

Im Unterschied dazu handelt es sich bei eingebetteten Systemen um sog. **reaktive Systeme**.

Def. [BLR95]: *A reactive system is one that is in continual interaction with its environment and executes at a pace determined by that environment.*

Klassen von EIS sind:

- **Zahlungssysteme auf der Basis EIS**

  Beispiel:

  Abb. 1.1 zeigt den sog. Smartpen.

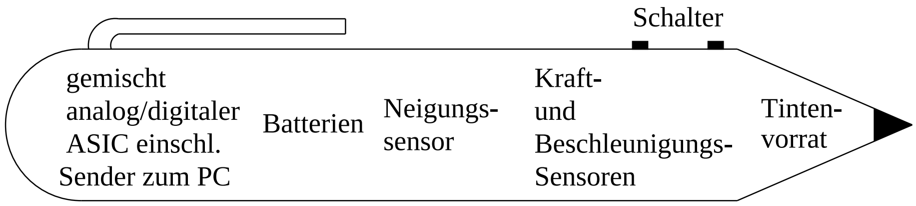


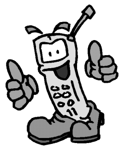
Abbildung 1.1: SMARTpen

  Dies ist ein Schreibgerät, welches der Identifizierung des Schreibers dient. Zur Identifizierung wird hier nicht nur die Unterschrift benutzt, sondern auch die Bewegungen des Schreibgeräts während des Unterschreibens. Hierzu gibt es verschiedene Sensoren im Schreibgerät, welches es einem Rechner erlauben, aus den per Funk übertragenen Daten ein Bild der Bewegung des Geräts im Raum zu rekonstuieren.

[1] Im Folgenden werden diese Systeme mit EIS (=eingebettete informationsverarbeitende Systeme) abgekürzt.

- **EIS in der Telekommunikation**

  Abb. 1.2 zeigt als Beispiel ein Blockdiagramm eines Ein-Chip Videotelefons [LNV+97].



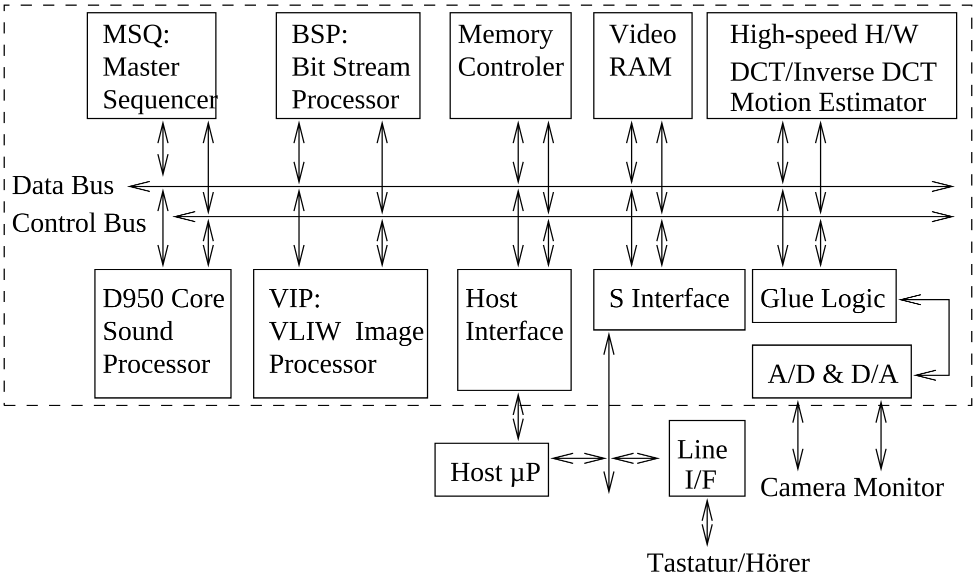
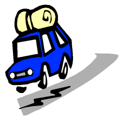


Abbildung 1.2: Ein-Chip Videotelefon (SGS Thomson)

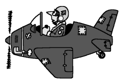
- **EIS in der Fahrzeugelektronik** (ABS-Systeme, Zug-Sicherungssysteme)

  - **Automobilelektronik**

    Moderne Automobile sind überhaupt nur mit umfangreicher elektronischer Ausrüstung auf dem Markt konkurrenzfähig. Einfache Systeme aus Analogschaltkreisen werden dabei verstärkt durch digitale Informationsverabeitung abgelöst. Hochwertige Automobile besitzen aus diesem Grund schon ca. 100 Mikroprozessoren.

  - **Flugzeugelektronik**

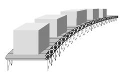
    Zur Information der Piloten werden elektronische Systeme schon seit Jahrzehnten genutzt. Der Anteil der Elektronik wird dabei ständig ausgeweitet, wie z.B. durch die Einführung eines eigenen Radarsystems zur Vermeidung der Kollision mit anderen Flugzeugen. Eine große Diskussion wurde durch die Einführung des *fly-by-wire* Systems ausgelöst, welches durch hydraulische Steuerung in manchen Flugzeugmodellen abgelöst hat. Unfälle haben deutlich vor Augen geführt, welche hohen Anforderungen an den Entwurf zu stellen sind.

  - **Schienenfahrzeugelektronik**

    Moderne Schienenfahrzeuge nutzen die Elektronik für vielfältige Aufgaben, beispielsweise zur Kommunikation mit der Zugwegs-Sicherheitstechnik, zur Information der Reisenden, zum effizienten Umgang mit der Energie, zur Information des Zugführers über den Zustand des Zuges usw.

- **EIS in der Fertigungs- und Automatisierungstechnik** (Fabriksteuerungen, Industrieroboter, Speicherprogrammierbare Steuerungen)

  Es sei hier das Beispiel einer Fabriksteuerung (nach Kopetz [Kop97]) angeführt:

  Gegeben sei ein Behälter mit einer Flüssigkeit, deren Durchflußmenge durch ein Ventil geregelt werden soll (siehe Abb. 1.3).

  Diese Durchflußmenge soll in allen Umständen erhalten bleiben, unabhängig von dem Füllstand des Behälters, der Temperatur usw. Zu diesem Zweck enthält das System einen Sensor, der den Fluß mißt. Das Ventil ist ein spezielles Beispiel eines **Aktors**, einer Komponente, über die das physikalische System gesteuert werden kann. Wie in diesem Beispiel werden viele Aktoren mit Sensoren kombiniert, um deren Funktion zu überwachen.

  Bei der Konstruktion eines Regelungsverfahrens für dieses Beispiel muss beachtet werden, dass es eine gewisse Zeit dauert, bis das Ventil geöffnet oder geschlossen ist. Man kann z.B. annehmen, dass

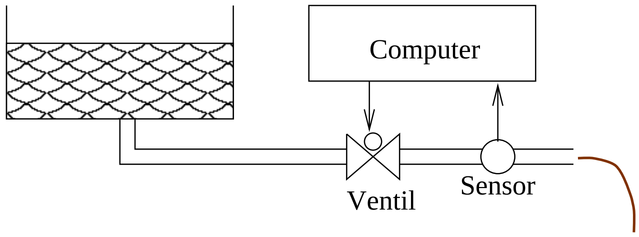


Abbildung 1.3: Regelung eines Ventils

es 10 Sekunden dauert, das Ventil voll zu öffnen oder zu schließen. Weiter ist zu berücksichtigen, dass der Sensor auch nur eine begrenzte Genauigkeit hat, z.B. eine solche von 1%. Dann könnte man alle 100 ms einen Sensormeßwert dem regelnden Computer zuführen, da sich in dieser Zeit auch das Ventil um 1% bewegen kann. Weiter ist die Zeitverzögerung zwischen Änderungen am Ventil und den Auswirkungen am Sensor zu berücksichtigen. Alle diese Effekte müssen bereits beim Entwurf eines einfachen Regelsystems beachtet werden. Viel komplexer sind Regelvorgänge mit vielen Sensoren und Aktoren.

Die Abbildungen 1.4 und 1.5 zeigen zwei Beispiele von Prototypen von Industrierobotern. Die Abbildung 1.4 zeigt einen sog. Rohr-Krabbler, ein Gerät, welches sich weitgehend selbständig (auch durch gekrümmte) Rohre bewegen kann.

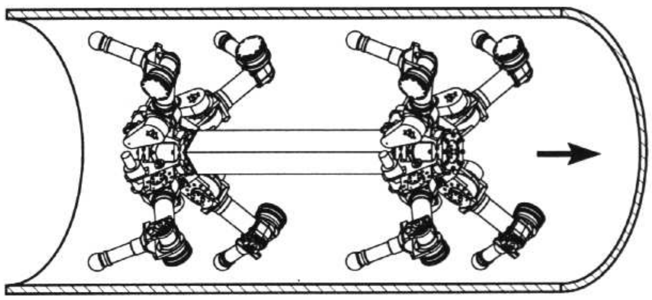


Abbildung 1.4: Rohr-Krabbler

Abbildung 1.5 zeigt das mechanische Modell einer Stabheuschrecke, welche ein Vorbild für den Rohr-Krabbler war.

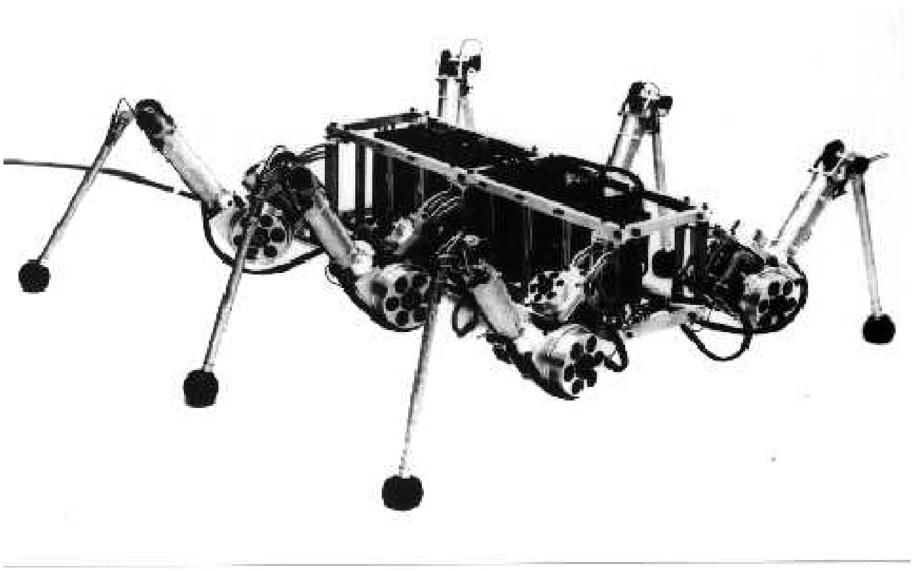


Abbildung 1.5: Stabheuschrecke (Modell)

- **EIS in der Gebäudeautomatisation**

  Beispielhaft sei hier ein System der Telefongesellschaft Helsinki (HSY) genannt [Pag98]:

  Bei diesem System melden Sensoren den Eintritt in einen intelligenten Raum. Licht und Belüftung werden in diesem Moment eingeschaltet. Gekoppelt ist dies mit Helligkeitsreglern und Sensoren. Um Kosten zu sparen, werden unbenutzte Räume nicht unnötig gewärmt oder gekühlt. Ein Kohlendioxid-Sensor beeinflusst die Luftzufuhr benutzter Räume. Das durch Lüfter verursachte Geräusch wird auf das jeweils benötigte Minimum reduziert. Ein zentraler Monitor gibt es jederzeit Auskunft über den Belegungszustand der einzelnen Räume. Der Energieverbrauch kann für jeden Raum erfasst werden. Das zur Kommunikation mit Sensoren und Regler benötigte Netz ist mit dem lokalen Rechnernetz integriert.

- **EIS in der Medizintechnik** (medizinische Analyse- und Überwachungssysteme)

Diese Klassen und Beispiele geben einen Eindruck von der Vielfalt eingebetteter Systeme.

Charakteristisch ist, dass diese Systeme in der Regel ohne das bei PCs und Workstations übliche Benutzerinterface in Form von Bildschirmen und Tastaturen auskommen. Sie werden daher häufig gar nicht als informationsverarbeitende Systeme wahrgenommen.

Die Bedeutung eingebetteter Systeme wird vielfach unterschätzt. Dies belegt auch das folgende Zitat von Mary Ryan [Rya95]:

> Embedded chips aren't hyped in TV and magazine ads ... but embedded chips form the backbone of the electronics driven world in which we live. ... they are part of almost everything that runs on electricity.

Der **Markt** für eingebettete Systeme ist sehr stark im Wachsen begriffen und wird voraussichtlich den Markt für Terminal-basierte Anwendungen übertreffen. Bereits heute werden nach Angaben der Zeitschrift *Electronic Design* 79% aller hochwertigen Mikroprozessoren in eingebetteten Systemen eingesetzt. Das bedeutet: pro Pentium-Prozessor, dessen man sich bewusst ist, existieren ca. vier hochwertige Prozessoren in eingebetteten Systemen, von deren Existenz man in der Regel nicht weiß. Gerade für Europa bilden eingebettete Systeme den Hauptmarkt im Bereich informationsverarbeitender Systeme (Rechner sind ja v.a. ein Haupt-Exportgut der USA). Daher zählt der Entwurf eingebetteter Systeme zu den häufigen Aufgaben europäischer Spezialisten für Informationsverarbeitung und soll daher hier behandelt werden.

Weitere charakteristische Eigenschaften eingebetteter Systeme sind die folgenden:

1. Es existieren vielfach **Realzeit-Bedingungen**, die unbedingt einzuhalten sind.
2. Es ist insgesamt eine hohe **Zuverlässigkeit** und **Sicherheit** erforderlich.
3. Die Anwendungen bei sind zur Entwicklungszeit i.W. **vollständig bekannt** (wenngleich Anpassungen an neuere Normen, Fehlerkorrekturen usw. auch in diesem Bereich später noch durchgeführt werden müssen).
4. Es sind **effiziente Lösungen** erforderlich. Mobile Geräte müssen beispielsweise mit wenig Energie auskommen. Siliziumfläche sollte bei Massenproktukten effizient genutzt werden.

Mit dem Schwerpunkt bei den eingebetteten Systemen reiht sich diese Vorlesung ein in eine Reihe von Vorlesungen über solche Systeme, die an vielen Universitäten inzwischen Standard sind. Die Bedeutung eingebetter Systeme wird beispielsweise dadurch unterstrichen, dass die Universität Kaiserslautern einen Studienschwerpunkt "eingebettete Systeme" innerhalb des Informatikstudiengangs eingerichtet hat.

Dabei können die in der Vorlesung vermittelten Techniken nicht nur im Bereich des Hardware-Entwurfs eingesetzt werden. Die Spezifikationstechniken des ersten Hauptkapitels etwa können auch zur Erzeugung reiner Software-Lösungen eingesetzt werden. Im Zusammenhang mit dem sog. Hardware/Software-Codesign sind die Grenzen zwischen Hardware und Software ohnehin fließend geworden. Aus ein und derselben Beschreibung können Teile und Hardware und andere Teile in Software realisiert werden.

Die Vorlesung gliedert sich in sieben Kapitel, einschließlich der Einleitung in Kapitel 1. Im Kapitel 2 werden wir uns mit Spezifikationsmethoden für eingebettete Systeme beschäftigen. Im Kapitel 3 werden wir Zielarchitekturen besprechen, die zur Realisierung von eingebetteten Systemen dienen können. Im Kapitel

4 werden wir über Validierung von Systembeschreibungen sprechen. Die Erzeugung einer Realisierung aus einer Spezifikation ist Gegenstand des fünften und des sechsten Kapitels. Im abschließenden Kapitel werden noch Methoden der Erzeugung von Schaltkreis-Layouts vorgestellt.

1.2 Literatur

Leider ist es nicht möglich, sich bei der Stoffauswahl für diese Vorlesung auf eine kleine Zahl von umfassenden Büchern zu beziehen. Hierfür ist das Gebiet einfach noch zu jung.

Statt dessen stützt sich die Vorlesung auf eine ganze Reihe von Texten über spezielle Themen. Für die sechs Hauptkapitel sind dies die folgenden:

- **Spezifikation und Spezifikationssprachen**

  In diesem Kapitel werden wir hierarchische Automatenmodelle vor allem anhand der von Harel beschriebenen StateCharts und der Variante SpecCharts [GVNG94] vorstellen. Daneben erläutern wir die Hardwarebeschreibungs-Sprache VHDL [IEE92], sowie die Sprachen SDL, Estelle und UML. Die Transformation der verschiedenen Sprachen ineinander kann in einigen Fällen bei Bergé [BLR95] nachgelesen werden. Eine Beschreibung der in VHDL benutzten Wertemengen für mögliche Signalwerte findet man in dem Buch "Synthese und Simulation von VLSI-Systemen" [Mar93] sowie in Dokumentionen zu auf VHDL basierenden Standards [Spe93].

- **Zieltechnologien**

  Zieltechnologien sind in Büchern von Post [Pos89] sowie Rosenstiel und Camposano [RC89] beschrieben. Verschiedene Prozessorkerne sind im web miteinander verglichen [EED]. FPGAs sind in speziellen Datenbüchern über diese Zieltechnologie beschrieben.

- **Validierung**

  Hier benutzen wir das Kapitel 1 des Buches "Synthese und Simulation von VLSI-Systemen" [Mar93], Standardtexte aus dem Bereich des Testens sowie neuere Literatur zur formalen Verifikation.

- **System-Synthesetechniken**

  In diesem Kapitel benutzen wir als Grundlage einige Arbeiten am Lehrstuhl 12 zum Hardware/Software-Codesign [NM96], zur Codeerzeugung für eingebettete Prozessoren [MG95, Leu96, Leu97b], und zur Mikroarchitektur-Synthese [LM$^+$93].

- **Logik- und Controller-Syntheseverfahren**

  Logik- und Controller-Syntheseverfahren diskutieren wir anhand des Buches "Synthese und Simulation von VLSI-Systemen" [Mar93] auf der Basis von Originalarbeiten (v.a. über BDDs).

- **Layouterzeugung**

  In diesem Kapitel orientieren wir uns ausschließlich an dem vierten Kapitel des Buchs "Synthese und Simulation von VLSI-Systemen" [Mar93].

Der vorliegende Text soll das Gebiet der Vorlesung definieren, relevante Literaturquellen aufzeigen und vermeiden, dass eine zu große Anzahl von Publikationen parallel zur Vorlesung gelesen werden muss. Er ist als Begleitmaterial für die Vorlesung gedacht. Für ein Selbststudium ohne Besuch der Vorlesung ist er nicht konzipiert, da hierfür wesentlich ausführlichere Kommentare erforderlich gewesen wären.

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 drei 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.

1.3 Einbindung der Vorlesung in das Lehrangebot des Fachbereichs

Thematisch eng benachbart ist die Vorlesung "Prozeßrechnertechnik (Eingebettete Realzeit-Systeme)", die seit dem Sommersemester 1998 angeboten wird. Diese Vorlesung konzentriert sich wie die zum vorliegenden Skript gehörende Vorlesung auf das Thema **eingebettete Systeme**. Letztere konzentriert sich aber auf CAD-Aspekte, während erstere sich in einem weiteren Umfeld mit Funktionsweise und Algorithmen eingebetteter Systeme beschäftigt. Dabei gibt es allerdings Themenbereiche, welche für beide Vorlesungen relevant sind. Hierzu zählen insbesondere die Spezifikationssprachen und die Zielarchitekturen. Bei der gegenwärtigen Aufteilung sind beide Themen der zum vorliegenden Skript gehörenden Vorlesung zugeschlagen worden. Ein Grund dafür ist weniger inhaltlich bestimmt: Die Vorlesung "Rechnergestützter Entwurf ..' wird im längeren Wintersemester, die andere im kürzeren Sommersemester durchgeführt. Beide Vorlesungen sind so angelegt, dass sie unabhängig voneinander gehört werden können. In Kombination miteinander sind beide Vorlesungen hervorragend geeignet, um ein Vertiefungsgebiet zu bilden. Nur in wenigen Fällen (wie z.B. dem jeweils ersten Kapitel) gibt es eine sehr enge Überlappung zwischen den beiden Vorlesungen.

Zu dieser Vorlesung werden Übungen angeboten. Die Übungen lehnen sich eng an die Vorlesung an. Sie beginnen mit einer Benutzung des StateMate-Systems (einer kommerziellen Implementierung von StateCharts) zur Erfassung und Verarbeitung von Spezifikationen. Dieses System wird benutzt, um verschiedene Realisierungen zur erzeugen. Im Verlauf des Semesters findet dann schrittweise eine Abkehr von der Benutzung dieses Systems statt.

Die Studienordnung des Studiengangs "Angewandte Informatik" sieht eine Wahlpflicht-Vorlesung "Rechnergestützter Entwurf / Produktion" vor. Die vorliegende Vorlesung ist eine Spezialisierung einer solchen Vorlesung auf die Mikroelektronik. In Bezug auf mündliche Prüfungen und beim Nachweis von Studienleistungen wird die vorliegende Vorlesung stets wie die Vorlesung "Rechnergestützter Entwurf / Produktion" ohne irgendwelche Namenszusätze behandelt.

Für den Studiengang "Informatik" ist die vorliegende Vorlesung als Spezialvorlesung wählbar. Damit können sich Studierende über den entsprechenden Stoff im Rahmen der Prüfung "Informatik III" (Vertiefungsgebiet) prüfen lassen. Leider ist dann bei demselben Prüfer eine Prüfung über "Rechnerorganisation" im Rahmen der Prüfung "Informatik II" z.Z. (Anfang 1999) nicht möglich. "Rechnergestützter Entwurf / Produktion (Mikroelektronik)" läßt sich mit verschiedenen anderen Themen zu einem Vertiefungsgebiet kombinieren. Im Bedarfsfall sollte eine Rücksprache beim Veranstalter dieser Vorlesung (oder beim Prüfer, falls dies eine andere Person ist) erfolgen.

Die Vorlesung enthält eine Reihe von Anknüpfungspunkten an weitere Vorlesungen im Fachbereich, wie beispielsweise die Vorlesungen "Rechnerarchitektur", "Rechnernetze und verteilte Systeme"und "Softwaretechnologie". Kenntnisse aus dem Studium des Anwendungsfaches Elektrotechnik sind gelegentlich hilfreich, um den Hintergrund des behandelten Stoffes zu verstehen.

Kapitel 2

Spezifikationssprachen

2.1 Anforderungen an Spezifikationssprachen

2. Vorles.

Zunächst müssen wir uns mit Spezifikationstechniken für eingebettete Systeme beschäftigen (siehe auch Abb. 2.1).

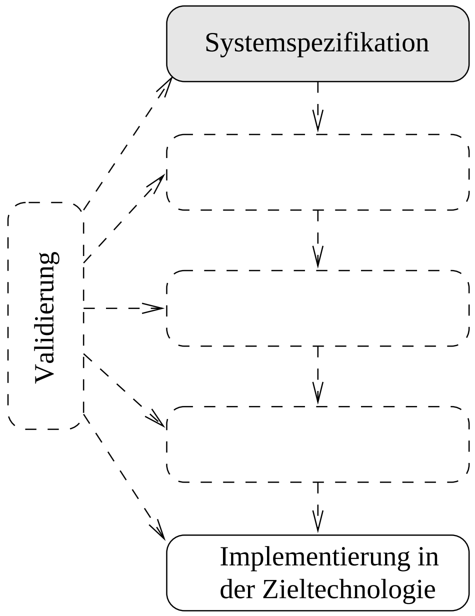


Abbildung 2.1: Kontext der Spezifikation

In vielen Fällen werden eingebettete Systeme auch heute noch mittels natürlicher Sprachen spezifiziert. So kann es vorkommen, dass komplexe Systeme in vielen Ordnern in englischer oder deutscher Sprache beschrieben werden. Diese Methode ist aber völlig ungeeignet, denn es ist kaum möglich, eine solche Beschreibung auf Widerspruchsfreiheit und Vollständigkeit zu überprüfen oder aus ihr systematisch eine Implementierung abzuleiten. Daher sollte man immer eine formale Sprache einsetzen, deren Semantik klar definiert ist und deren Verarbeitung durch Rechner möglich ist.

Bergé [BLR95] nennt folgende Anforderungen an Spezifikationssprachen für reaktive Systeme:

1. **Zustandsorientiertes Verhalten**:

   Reaktive Systeme sind Systeme, die auf Eingaben zu einer Zeit aufgrund ihres internen Zustandes nach einer meist genau bekannten Zeit eine Ausgabe liefern, in einen anderen internen Zustand übergehen und danach weitere Eingaben verarbeiten sollen. Das Verhalten besteht also aus einer Abfolge von Eingaben und zugehörigen Reaktionen.

Als Modell reaktiver Systeme sind daher Automaten geeignet. Allerdings sind die klassischen Beschreibungstechniken für Automaten für praktische Anwendungen unzureichend.

2. **Zeitliches Verhalten:**

   Für die Übergänge können präzise Zeitangaben vorgegeben sein. Man spricht auch von **Realzeitsystemen mit harten Zeitbedingungen**.

   Def.: Nach Kopetz [Kop97] heißen Zeitbedingungen **hart**, wenn ihre Nichteinhaltung eine Katastrophe auslösen könnte.

3. **Nebenläufiges Verhalten** mit Möglichkeiten der Angabe von Synchronisation und Kommunikation:

   Es muss möglich sein, räumlich verteilte Systeme und Systeme, die mehrere Aufgaben gleichzeitig bearbeiten, zu beschreiben. Dies ist nur mittels Nebenläufigkeit effizient und übersichtlich (d.h. ohne Bildung des Kreuzproduktes aller Aktivitäten) möglich. Klassische deterministische Automaten, die sich jeweils nur in einem Zustand befinden können, sind daher nicht geeignet.

4. **Ausnahmeorientiertes Verhalten:**

   Praktische Systeme müssen vielfach auf Sondersituationen weitgehend unabhängig von ihrem aktuellen Zustand reagieren. Typisch sind hierfür Fehlersituationen, Abbrüche durch Benutzer usw. Sollen die Spezifikationen lesbar bleiben, so müssen solche Sonderfälle effizient beschreibbar sein. Sonderfälle für jeden betroffenen Zustand spezifizieren zu müssen, ist nicht akzeptabel.

5. **Umgebungsabhängiges Verhalten:**

   Es muss möglich sein, globale Einflüsse der Umgebung zu beschreiben, ohne alle Angaben über die Umgebung als Parameter an alle Teile der Modelle zu übergeben. Beispielsweise muss in jedem Teilmodell ggf. ein temperaturabhängiges Verhalten beschrieben werden können, ohne dass die Temperatur der Umgebung an jedes Teilmodell explizit übergeben wird.

6. **Nicht-funktionelle Eigenschaften:**

   Eingebettete Systeme müssen vielfach zusätzliche Eigenschaften besitzen. Beispiele solcher nicht-funktioneller Eigenschaften sind: Fehlertoleranz, Zuverlässigkeit, maximaler Leistungsverbrauch, Gewicht, Abmessungen, Temperaturbereich, Benutzerfreundlichkeit, Aufrüstbarkeit, Lebensdauer und die Recyclingfähigkeit.

   Es sei bereits an dieser Stelle bemerkt, dass diese Forderung heute von keiner formalen Sprache erfüllt wird, vielleicht auch niemals erfüllt werden kann.

Gajski [GVNG94] stellt folgende zusätzlichen Forderungen:

7. **Hierarchie (strukturell und verhaltensmäßig)**:

   Da der Mensch immer nur eine relativ kleine Zahl von Objekten (Zuständen, Übergängen, Hardwarekomponenten) überblicken kann und da die meisten realen Systeme eine größere Zahl von solchen Objekten erfordern, ist es nur mit Hilfe von Hierarchie möglich, den Überblick zu bewahren. Ohne einen solchen Überblick könnte man sich nicht davon überzeugen, dass die Spezifikation der Intention entspricht. Dabei wird sowohl eine verhaltensmäßige Hierarchie (z.B. in Form von Prozeduren oder Klassenbibliotheken) als auch eine strukturelle Hierarchie (z.B. in Form von Hardwarekomponenten) benötigt.

8. **Programmkonstrukte:**

   In praktischen Anwendungen müssen Ergebnisse vielfach auf komplexe Weise berechnet werden. Hierfür haben sich die üblichen Programmiersprachen sehr bewährt. Arithmetische Operationen, Schleifen und Funktionsaufrufe sollten verfügbar sein (leider wird diese Forderung von verbreiteten Methoden der Spezifikation wie StateCharts nur unzulänglich unterstützt).

9. **Terminierung:**

   Gajski fordert, dass erkennbar sein muss, wann Berechnungsschritte oder ein Zweig nebenläufigen Verhaltens abgeschlosen sind.

Im Hinblick auf die Diskussion über die Software nicht-reaktiver Systeme kann man natürlich noch weitere Forderungen hinzufügen, wie etwa die Unterstützung der Objektorientierung, die Ausführbarkeit der Spezifikation, Maschinenunabhängigkeit sowie die Festlegung einer **exakten Semantik**. Für die Festlegung der Semantik gibt es dabei wieder viele Möglichkeiten. Meist wird leider nur beschrieben, wie ein Simulator zu realisieren ist (prozedurale Semantikbeschreibung).

Ein Sonderheft der Zeitschrift **it+ti** [Ram98] enthält eine Übersicht über den Stand der Technik im Bereich der Spezifikationsmethoden.

Es kann bereits hier festgestellt werden, dass keine der existierenden Sprachen alle Forderungen erfüllt. Im Folgenden werden wir einige der verbreiteten Spezifikationssprachen und ihre Vor- und Nachteile vorstellen.

2.2 StateCharts

2.2.1 Sprachelemente

Die erste Sprache, die hier vorgestellt werden soll, ist eine graphische Beschreibungssprache eines erweiterten Automatenmodells. Bei den klassischen Automaten unterscheiden wir zwischen Moore- und Mealy-Automaten:

- **Moore-Automaten:** Diese lassen sich mathematisch durch die beiden Funktionen λ und δ beschreiben (siehe z.B. [Koh87]):

  $\delta : X \times Z \rightarrow Z$
  $\lambda : Z \rightarrow Y$

  Darin bedeuten:

X	:	Die Menge möglicher Eingabewerte
Y	:	Die Menge möglicher Ausgabewerte
Z	:	Die Menge der inneren Zustände

  Der Zustand $z^+ = \delta(x, z)$ mit $z, z^+ \in Z$ heißt **Folgezustand** des Zustands z bei Eingabe von x.

- **Mealy-Automaten:** Bei Mealy-Automaten ist die Ausgabe nicht nur vom aktuellen Zustand, sondern auch von der Eingabe abhängig:

  $\delta : X \times Z \rightarrow Z$
  $\lambda : X \times Z \rightarrow Y$

Moore- und Mealy-Automaten bilden zusammen die Klasse der **endlichen Automaten** (engl. *finite state machines* (FSM)"). Die Funktionen λ und δ kann man sehr übersichtlich durch den sog. **Zustandsgraphen** darstellen, welche für jeden Zustand genau einen Knoten und für jede Eingabe mit unterscheidbarer Wirkung (anderer Folgezustand oder – bei Mealy-Automaten – andere Ausgabe) eine Kante besitzen.

Die Zustandsdiagramme klassischer Automaten werden bei nicht-trivialen Systemen schnell unübersichtlich. Dies zu vermeiden ist das Ziel der graphischen Sprache StateCharts, die auf Harel zurückgeht [Har87, DH89]. Harel zufolge wurde der Name StateCharts gewählt, weil StateChart *the only unused combination of 'flow' or 'state' with 'diagram' or 'chart'* war.

Die erste wesentliche Erweiterung gegenüber klassischen Zustandsdiagrammen besteht in der Einführung von Hierarchie. Zustände können danach zu **Superzuständen** (engl. *superstates*) zusammengefasst werden. Dies zeigt die Abb. 2.2.

Da aus den Zuständen `A` und `C` nach `B` verzweigt wird, ergibt sich eine Reduktion der Anzahl der Kanten, wenn man `A` und `C` zu einem Superzustand `D` zusammenfasst. Weiter kann es sinnvoll sein, die Reaktion auf `C(P)` in `D` zu verstecken. Die Kante aus `D` heraus bedeutet: `D` wird als Reaktion auf `b` verlassen, unabhängig davon, in welchem der Teilzustände von `D` sich das System aufhält. Bei der hier beschriebenen Form von Superzuständen schließen sich `A` und `C` dabei gegenseitig aus. Das System kann nur in einem der beiden Zustände sein und die beiden Zustände heißen **exklusiv**.

Def.: Die Zustände der untersten Hierarchieebene nennen wir **Basis-Zustände**.

Def.: Die **Vorgänger-Zustände** (engl. *ancestor states*) eines Basis-Zustands z sind jene Superzustände, welche z enthalten.

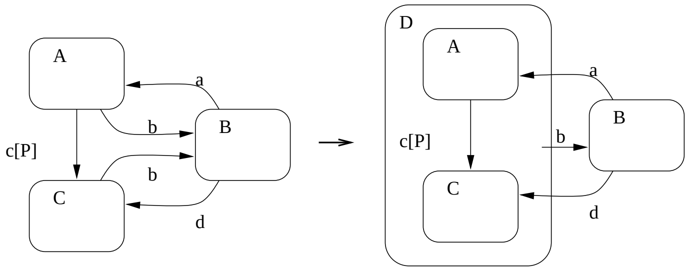


Abbildung 2.2: Zusammenfassen zu hierarchischen Zuständen

In Abb. 2.2 haben wir durch Kanten bis zu den inneren Zuständen hin beschrieben, welcher Zustand eingenommen werden soll. Es gibt weitere Mechanismen, den angesprungenen Zustand innerhalb von Superzuständen zu bestimmen. Die erste Möglichkeit hierzu ist der **Standardzustand** (engl. *default state*), wie in Abb. 2.3 graphisch dargestellt. Der Zustand `A` wird eingenommen, wenn sonst weiter nichts angegeben wird.

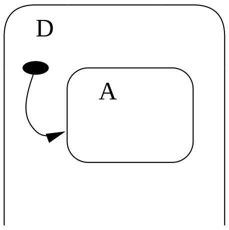


Abbildung 2.3: Graphische Darstellung des Standardzustandes

Eine zweite Möglichkeit besteht in der Benutzung des *history*-Mechanismus. Mit diesem Mechanismus merkt man sich quasi eine Rückkehradresse, d.h. den letzten Zustand vor dem Verlassen des Superzustandes. Der History-Mechanismus wird mit einem `H` dargestellt, siehe Abb. 2.4 (links).

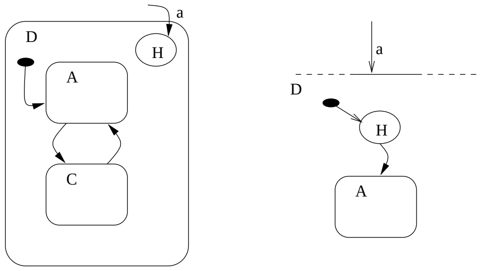


Abbildung 2.4: Graphische Darstellung des History-Zustandes

Beim ersten Betreten von `D` wird dabei `A` angenommen, ansonsten der vor dem Verlassen von `D` zuletzt eingenommene Zustand. Der History-Mechanismus ermöglicht es, von `D` aus andere Zustände wie Prozeduren anzuspringen und zum aufrufenden Zustand zurückzukehren. Abb. 2.4 (rechts) beschreibt eine äquivalente Notation für den History-Mechanismus.

Der History-Mechanismus kann auch hierarchisch eingesetzt werden, wie in Abb. 2.5 zu sehen.

Neben den sich gegenseitig ausschließenden Teilzuständen gibt es auch die Möglichkeit sog. UND-verknüpfter oder **orthogonaler** Zuständen. Bei UND-verknüpften Teilzuständen befindet sich das System gleichzeitig in den Teilzuständen. UND-verknüpfte Teilzustände werden graphisch mit einer durchbrochenen Linie kenntlich gemacht. In Abb. 2.6 ist ein System zu sehen, welches sich gleichzeitig in den Zuständen `B` und `C` befindet, sofern `A` betreten wurde.

Mit UND-verknüpften Zuständen wird die Forderung nach Modellierung von Nebenläufigkeit erfüllt.

Es gibt mehrere Möglichkeiten, zu definieren, welche Teilzustände beim Betreten UND-verknüpfter Teilzustände eingenommen werden. Dies zeigt die Abb. 2.7.

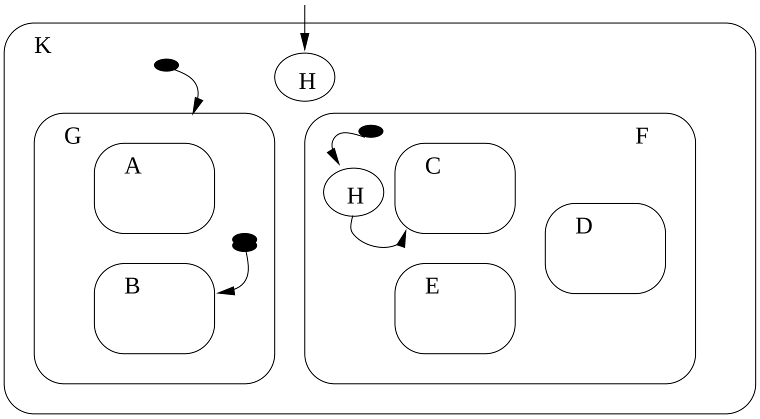


Abbildung 2.5: Hierarchische Verwendung des History-Mechanismus

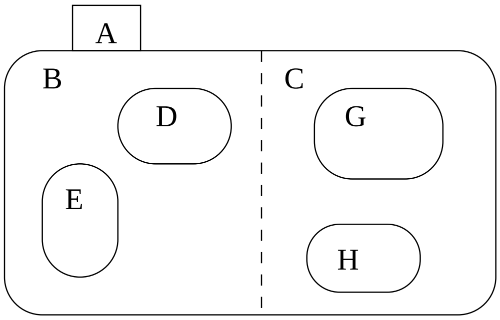


Abbildung 2.6: UND-verknüpfte Zustände

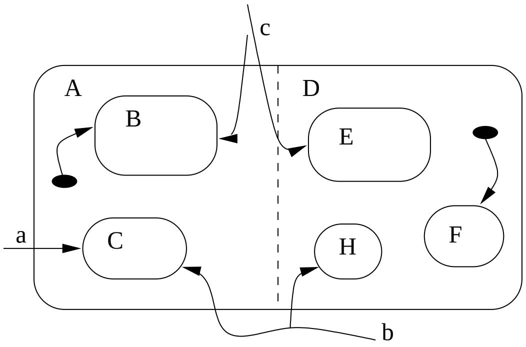


Abbildung 2.7: Verzweigungen in UND-verknüpfte Zustände

Einmal kann man explit angeben, in welche Teilzustände verzweigt wird (siehe Übergang für `c`). Weiter kann für einen Teilzustand der History-Mechanismus benutzt werden (siehe Übergang für `b`, wobei für das erste Betreten noch der default-Mechanismus eingesetzt wird). Schließlich kann für einen Teilzustand ein expliziter Übergang benutzt werden, während für weitere der default-Mechanismus eingesetzt wird (siehe Übergang für `a`).

Auch bezüglich des Verlassens UND-verknüpfter Zustände gibt es mehrere Möglichkeiten (siehe Abb. 2.8).

Man kann explizit angeben, in welcher Kombination von Zuständen man sich befinden muss, um den Superzustand zu verlassen (siehe Übergang für `b`). Man kann den Superzustand verlassen, völlig unabhängig davon, in welchem Teilzustand man sich befindet (siehe Übergang für `e`). Schließlich kann man den Superzustand verlassen, wenn man in einem bestimmten Teilzustand ist (siehe Übergang für `a`).

Insgesamt kann man sagen, dass Zustände in StateCharts entweder UND-Zustände, ODER-Zustände oder Basis-Zustände sind.

Kantenbeschriftungen besitzen die allgemeine Form 3. Vorles.

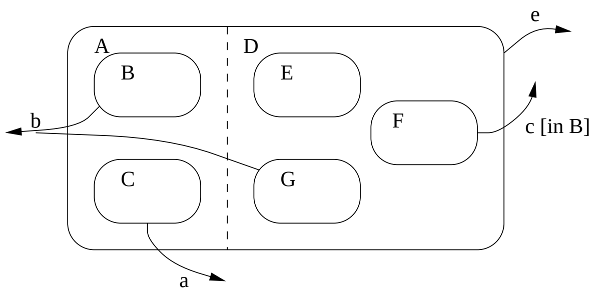


Abbildung 2.8: Verweigungen aus UND-verknüpften Zuständen

$$ev[cond]/react$$

Darin steht *ev* für ein **Ereignis** (engl. *event*), welches stattgefunden haben muss, damit die zugehörige **Reaktion** einschließlich eventueller Zustandsübergänge stattfinden kann. Ein generiertes Ereigniss lebt nur bis zum nächsten Schritt (s.u.). Ereignisse können extern (in der Umgebung) oder intern generiert werden. Ist `S` ein Zustand, so sind *entered*(S) und *exited*(S) spezielle interne Ereignisse. Die werden generiert, sofern der Zustand `S` betreten bzw. verlassen wurde.

cond ist eine **Bedingung** (engl. *condition*). Bedingungen beziehen sich auf den jeweiligen Zustand eines StateChart-Systems und leben typischerweise länger als nur für einen Schritt. Ist `S` ein Zustand, so ist *in*(S) eine spezielle Bedingung. Sie ist erfüllt, sofern sich ein System in `S` befindet.

react ist eine **Reaktion** (engl. *reaction*). Reaktionen können aus der Erzeugung von Ereignissen bestehen. Sie können auch Operationen auf Daten beschreiben. Zu diesem Zweck unterstützt StateChart **Variable**. Reaktionen sind entweder **Aktionen** (engl. *actions*) oder **Aktivitäten** (engl. *activities*). Aktionen sind einmalig und benötigen keine (nennenswerte) Zeit. Beispiele dafür sind das Erzeugen eines Ereignisses oder die Wertzuweisung zu einer Variablen. Aktivitäten dauern längere Zeit an.

Benutzt man als Reaktion verschiedene Operationen auf Variablen oder die Erzeugung von Ereignissen, so kann man damit eine Synchronisation und Kommunikation innerhalb eines StateChart-Modells beschreiben. Eine Anwendung dafür ist in der Abb. 2.9 zu sehen.

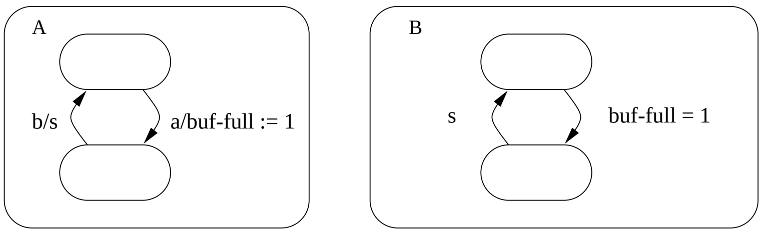


Abbildung 2.9: Globale Kommunikation und Synchronisation

In einem Modell wird mittels einer Zuweisung an die Variable `buf-full` mitgeteilt, dass nach Eintreffen des Signals `a` der Puffer voll ist. Dies wird im rechten Teil in der Formulierung einer Bedingung ausgenutzt. Neben der Kommunikationsmöglichkeit über Variable besteht auch eine Synchronisationsmöglichkeit über Signale, wie z.B. das Signal `s`. Signale unterscheiden sich in StateCharts von Variablen v.a. dadurch, dass sie nicht selbst Werte tragen können. Die StateChart-Semantik basiert auf der Annahme, dass Änderungen von Signalen und Variablen mit einem *broadcast*-Mechanismus dem ganzen Modell mitgeteilt werden. Dies macht StateCharts für verteilte Anwendungen nur bedingt einsetzbar, denn diese globale Kommunikation kann man dort nicht voraussetzen, bzw. sie wäre mit nennenswerten Laufzeiten behaftet.

Statische Reaktionen haben syntaktisch die Form von Kantenbeschriftungen, die aber innerhalb von Zuständen notiert werden. Sie beschreiben Reaktionen, welche erfolgen sollen, wenn das System in dem betreffenden Zustand ist. Semantisch können sie durch Einführung eines Teils eines UND-Zustandes erklärt werden, in dem die statische Reaktion eine Kantenbeschriftung darstellt.

Beispiel:

Linke und rechte Seite der Abb. 2.10 besitzen dieselbe Semantik.

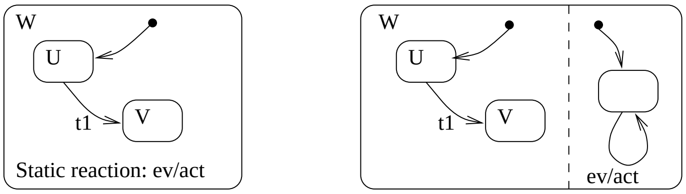


Abbildung 2.10: Semantik statischer Reaktionen

Gemäß der Forderung, zeitliches Verhalten zu beschreiben, können in StateCharts auch **Timer** dargestellt werden. Dies geschieht wie in Abb. 2.11 mittels einer gezackten Linie, der Angabe von Zeitschranken und eines mit 'timeout' beschrifteten Übergangs. Bei nur einer Zeitgrenze erfolgt der Übergang nach Ablauf der entsprechenden Zeit, ansonsten innerhalb des angegebenen Intervalls.

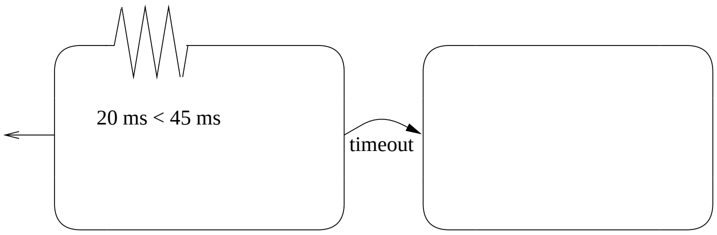


Abbildung 2.11: Graphische Darstellung von Timern

Zeitbedingungen können auch über mehrere Ebenen hinweg formuliert werden (siehe Abb. 2.12).

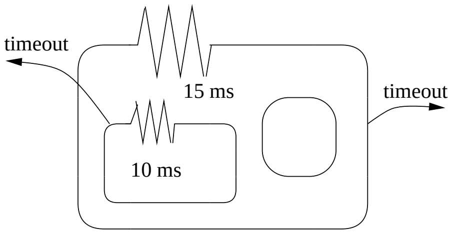


Abbildung 2.12: Hierarchische Benutzung von Timern

Mittels Konnektoren ist es möglich, die Übersicht zu erhöhen, vgl. Abb. 2.13.

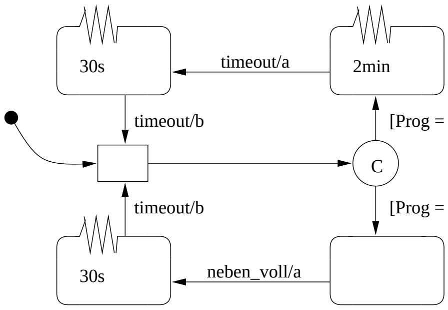


Abbildung 2.13: Beispielhafte Verwendung von Konnektoren

Die Semantik der mit C beschrifteten Kreise kann durch die direkte Führung von Kanten von den Quellen zu den Senken erklärt werden.

2.2.2 STATEMATE und die STATEMATE-Semantik von StateCharts

Die übersichtliche Darstellung von StateChart hat zu einer großen Beliebtheit dieser Beschreibungsform geführt, wobei allerdings eine Vielzahl von Varianten in Gebrauch sind.

STATEMATE ist wohl die meisten verbreitete kommerzielle Implementierung[1] von StateCharts mit zusätzlichen Möglichkeiten der Datenmanipulation. STATEMATE erlaubt über den Zwischenschritt der Erzeugung von VHDL (siehe Abschn. 2.4) eine Realisierung der Spezifikation in Hardware. Alternativ kann auch C-Code ausgegeben werden, womit dann eine Realisierung des Systems in Software möglich ist. STATEMATE ist in der Industrie sehr verbreitet[2]. Beispielsweise wird es bei BMW (in Kombination mit einem weiteren Werkzeug zur vereinfachten Beschreibung von Berechnungen) beim Entwurf von Automobilelektronik eingesetzt.

2.2.2.1 Activity-Charts und Module-Charts

Zusätzlich zu den StateCharts unterstützt STATEMATE auch noch zwei andere Diagramm-Formen, nämlich *activity charts* und *module charts*.

Die Bedeutung von *activity charts* beschreibt die Dokumentation von STATEMATE so:

Activity charts can be viewed as multi-level data-flow diagrams. They capture functions, or activities, all organized into hierarchies and connected via the information that flows between them. We adopt extensions that distinguish between data and control information in the arrow types, and also provide several kinds of graphical connectors, as well as a semantics for information that flows to and from non-basic activities.

Figure 2.14 illustrates some of these notions ... We see internal activities, such as `GET_INPUT` *... and external activities such as* `OPERATOR` *... data flows such as* `RANGE_LIMITS` *... control flows, such as* `COMMANDS` *and the control activity* `EWS_CONTROL`.

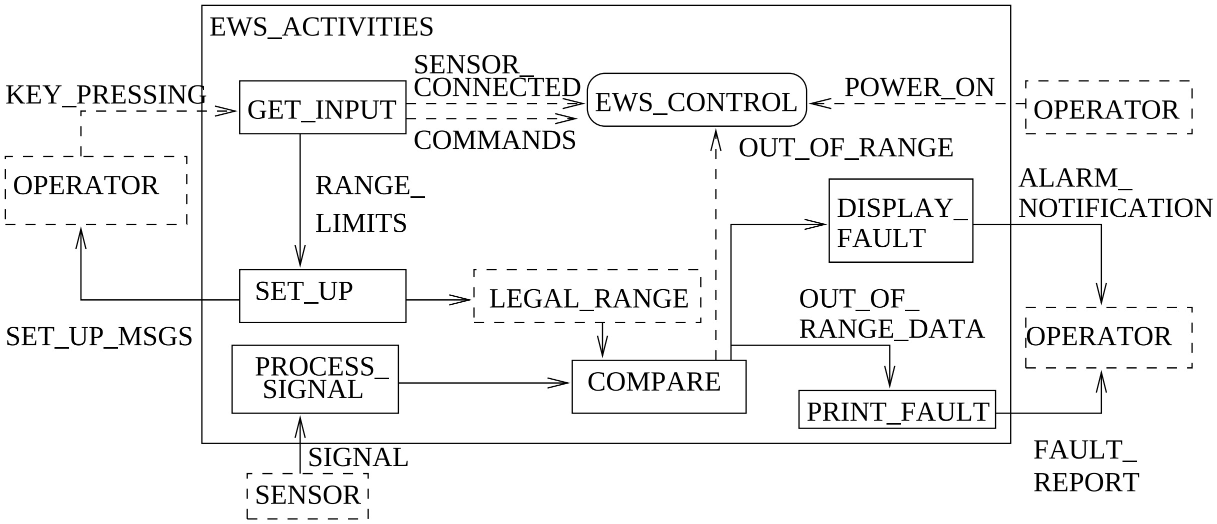


Abbildung 2.14: Activity chart

Die Bedeutung von *module charts* beschreibt die Dokumentation von STATEMATE so:

A module-chart can also be regarded as a certain kind of data-flow diagram or block diagram. Module-charts are used to describe the modules that constitute the implementation of the system, its division into hardware and software blocks and their inner components, and the communication between them.

Fig. 2.15 shows a module chart for the EWS. It contains internal moduels such as the control and computation unit (`CCU`*), ...*

Laut Dokumentation von STATEMATE ist ein Abgleich der einzelnen Diagrammformen vorgesehen.

[1] Vertrieb über die Fa. i-Logix.

[2] Offenbar wird z.Z. (1998) vermehrt auf neue Versionen von MATLAB umgestellt, welche nunmehr ein Werkzeug STATEFLOW enthält. STATEFLOW erlaubt ebenfalls die Spezifikation hierarchischer Zustände, ist aber hinsichtlich der Spezifikationsmöglichkeiten des Verhaltens reichhaltiger.

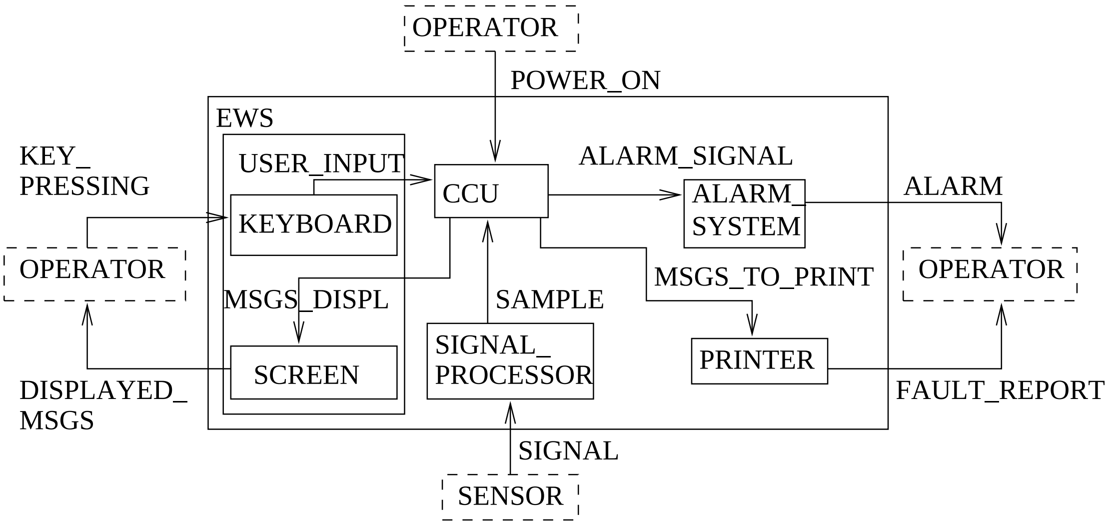


Abbildung 2.15: Module chart

2.2.2.2 Die Begriffe Status und Schritt

Eine sinnvolle kommerzielle Implementierung war nur auf der Basis einer präzisen Semantik der Spezifikationen möglich. Die ursprüngliche Fassung von StateCharts enthielt hier Unklarheiten, z.B. was die präzise Abhängigkeit vom einem Takt anbelangt. Harel hat diese Unklarheiten mit einer neueren Veröffentlichung [HN96] ausgeräumt.

Jede aktuelle Situation eines STATEMATE-Systems ist danach durch den jeweiligen **Status** beschrieben. Von einem Status zum nächsten kommt man durch Ausführen eines **Schritts**.

Abbildung 2.16: Begriffe Status und Schritt

Es wird angenommen, dass es keine Zeit kostet, einen Schritt auszuführen. Der zeitliche Abstand zwischen zwei Schritten ist nicht Teil der Semantikdefinition, sondern wird durch die Umgebung bestimmt. Die Beschreibung der Semantik basiert auf folgenden Prinzipien:

1. Reaktionen auf im Schritt n erzeugte externe oder interne Ereignisse können erst nach Beendigung des Schritts beobachtet werden.
2. Ereignisse existieren nur bis zur Ausführung des Schrittes, nach dem sie aufgetreten sind.
3. Berechnungen in einem Schritt basieren auf der Situation vor Ausführung des Schrittes.
4. Es wird stets eine maximale Menge nicht-konflikt-behafteter Transitionen und statischer Reaktionen (SRs, Reaktionen aufgrund eines konkreten Zustands) ausgeführt.

Zu beachten ist, dass STATEMATE Zeitmodelle unterstützt, bei denen mehrere Schritte zu derselben Zeit t ausgeführt werden. Auch in diesem Fall kann ein im Schritt n erzeugtes Ereignis nur im Schritt $n+1$ beobachtet werden.

Def. : Eine **Konfiguration** ist die maximale Menge von Zuständen, in denen sich ein System befinden kann. Konfigurationen sind durch die Basis-Zustände definiert, in denen sich ein System befindet.

Beispiel:

In Abb.2.17 ist es möglich, dass sich das System in den Basis-Zuständen `B1`, `C1` und `D1` befindet. Die Konfiguration enthält in diesem Fall noch die Vorgänger-Zustände `A` und `S`.

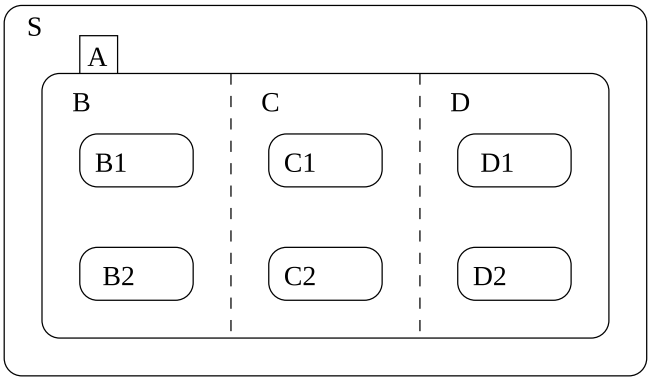


Abbildung 2.17: Erläuterung des Begriffs **Konfiguration**

2.2.2.3 Einfache Transitionen

Die Semantik von STATEMATE für einfache Übergänge ist nun wie folgt erklärt:

Ein System möge sich in einem Basis-Zustand `A` befinden und das Ereignis `ev` möge gerade stattgefunden haben (vgl. Abb. 2.18).

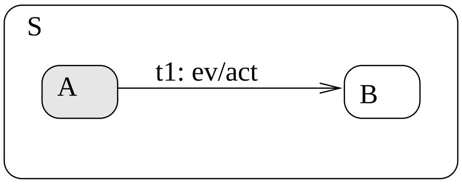


Abbildung 2.18: Zur Semantik einfacher Transitionen

Dann wird Folgendes ausgeführt:

- Die Transition `t1` kann schalten. `t1` findet daher statt, das System geht in den Zustand `B`. Die Aktion `act` wird ausgeführt.
- Die Ereignisse `exited (A)` und `entered (B)` werden erzeugt und können im nächsten Schritt beachtet werden.
- Die Bedingung `in (A)` und wird falsch, `in (B)` wird wahr.
- Die Aktionen, die beim Betreten von `B` ausgeführt werden sollen, werden ausgeführt.
- Alle Statischen Reaktionen (SRs) von `S`, deren Bedingungen erfüllt sind, werden ausgeführt. (weil sich das System in `S` befand und `S` nicht verlässt.
- Alle Aktivitäten, die als `within(A)` angegeben wurden, werden deaktiviert. Jene, die als `throughout (B)` angegeben wurden, werden aktiviert.

Von diesen Änderungen über Funktionen abhängige Ereignisse und Bedingungen werden ebenfalls realisiert.

Die o.a. Änderungen werden erst in dem folgenden Schritt wirksam. So wird `exited (A)` zwar generiert, eine Wirkung kann es jedoch erst beim folgenden Schritt haben. Als Folge davon kann eine System einen bestimmtem Zustand nicht innerhalb eines Schrittes betreten und wieder verlassen.

2.2.2.4 Verbundtransitionen

4. Vorles.

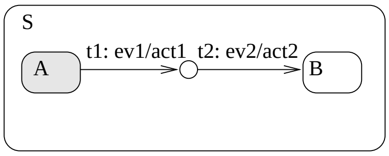
Im Falle hierarchischer Zustände und der Verwendung von Konnektoren gibt es zwischen den Zuständen Wege über Kanten.

Def.: Jede einzelne Kante entlang eines solchen Weges heißt **Transitionssegment**.

Def.: Eine **Verbundtransition** (engl. *compound transition*, (CT)) ist eine Kette von Transitionssegmenten.

Def.: Eine **Basis-Verbundtransition** ist eine maximale Kette von Transitionssegmenten, die gleichzeitig ausführbar sind. Die Ausführungsbedingung ist die Konjunktion der Ausführungsbedingungen der einzelnen Segmente.

Beispiel:



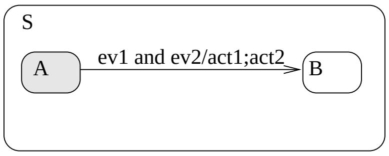


Abbildung 2.19: Beispiel einer Verbundtransition

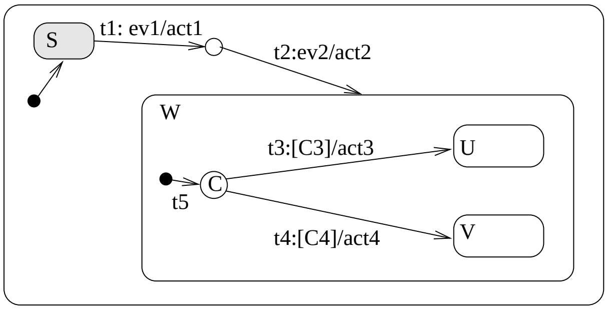
In Abb. 2.19 besitzen die linke und die rechte Seite der Abbildung dieselbe Bedeutung.

Es gibt zwei Sorten von Basis-CTs:

- Anfangs-CTs, welche von einem Zustand ausgehen,
- Fortsetzungs-CTs, die von einem Default- oder History-Connector ausgehen.

Def.: Eine **volle Verbundtransition** besteht aus einer Anfangs-CT und möglicherweise mehreren Fortsetzungs-CTs, deren Ausführung zu einer gültigen Konfiguration führt.

Beispiel:



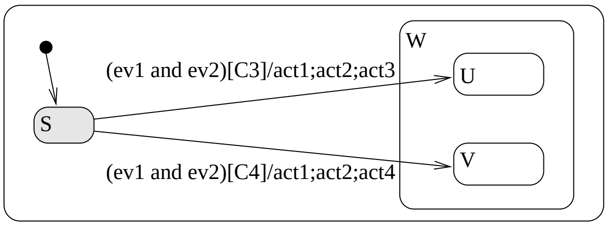


Abbildung 2.20: Zum Begriff der vollen Verbundtransition

Eine Übersicht über die verschiedenen Transitionstypen zeigt die Abb. 2.21.

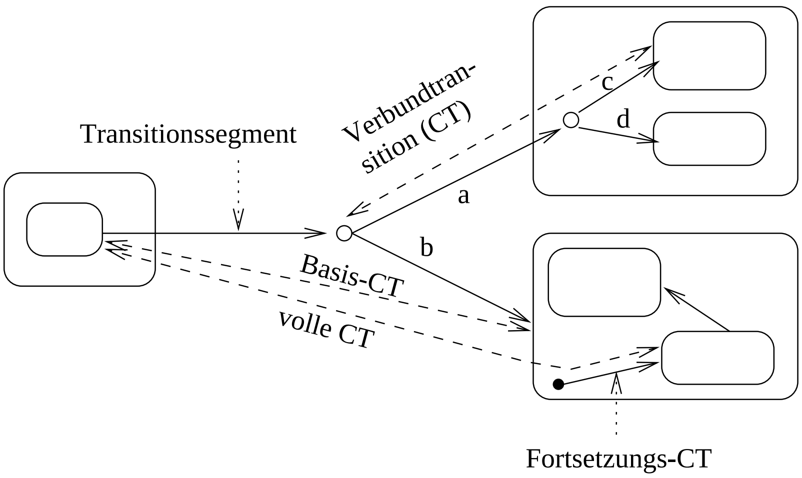


Abbildung 2.21: Verschiedene Klassen von Transitionen

Linke und rechte Seite der Abbildung 2.20 besitzen dieselbe Bedeutung: Volle Verbundtransitionen sind {`t1, t2, t5, t3`} sowie {`t1, t2, t5, t4`}. Falls `C3` und `C4` falsch sind, so bleibt das System in jedem Fall im Zustand `S`.

Eine komplizierte Situation zeigt die Abb. 2.22.

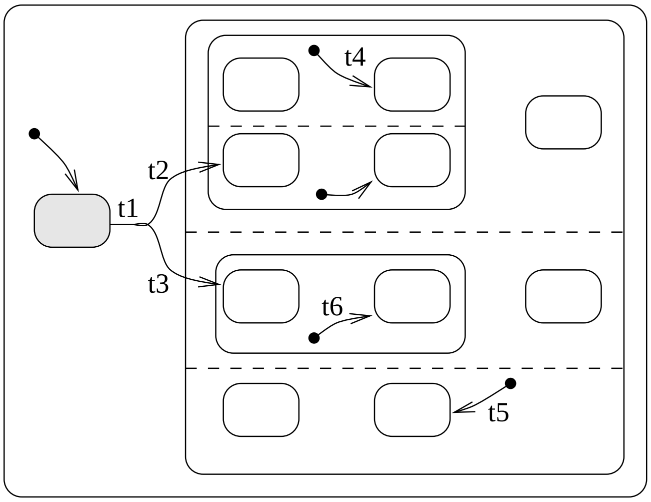


Abbildung 2.22: Komplexe Verbundtransition

Die Anfangs-CT {`t1, t2, t3`} muss von den beiden Fortsetzungs-CTs `t4` und `t5` begleitet werden, um eine volle CT zu werden.

2.2.2.5 Der Scope von Verbundtransitionen

Def.: Der **Scope** einer Verbundtransition `tr` ist der in der Hierarchie äußerste ODER-Zustand (Vorgänger-Zustand), der alle Quellen und Ziele von `tr` enthält.

Wenn `tr` ausgeführt wird, so werden innerhalb des Scopes von `tr` alle Quellen von `tr` verlassen und alle Ziele von `tr` betreten.

Beispiel:

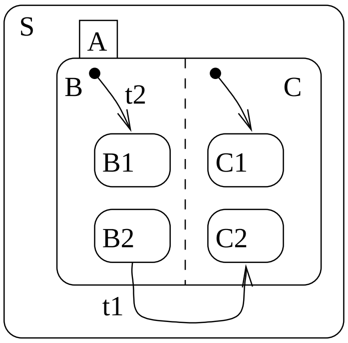


Abbildung 2.23: Zum Verlassen und Betreten von Super-Zuständen

In Abb. 2.23 ist S der Scope von `t1` , da `A` ein UND-Zustand ist. Wenn `t1` ausgeführt wird, dann werden `B2, B, A` und `C` sowie entweder `C1` oder `C2` verlassen und `A, B, B1, C` und `C2` betreten.

2.2.2.6 Konfliktbehaftete Transitionen

Zwei CTs heißen **konfliktbehaftet**, wenn es einen gemeinsamen Zustand gibt, der verlassen wird sofern eine der CTs ausgeführt wird.

Beispiel:

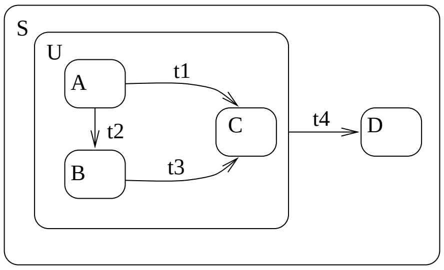


Abbildung 2.24: Konfliktbehaftete Transitionen

In Abb. 2.24 stehen `t1` und `t2` miteinander in Konflikt. `t4` steht in Konflikt mit `t1` , `t2` und `t3`.

Wenn konfliktbehaftete Transitionen auf derselben Hierarchieebene ausgeführt werden können, dann erfolgt die Auswahl der ausgeführten Transitonen nicht deterministisch. Im Falle unterschiedlicher Hierarchieebenen hatte ursprünglich die höhere Ebene Priorität (im Beispiel : `t4`). Diese Priorität soll offenbar umgekehrt werden [HN96].

2.2.2.7 Simulationsalgorithmus

Den Kern der Semantikbeschreibung von STATEMATE bildet eine Darstellung der Vorgänge während der Ausführung eines Schrittes. Der entsprechende Teil des Simulationsalgorithmus geht aus

- von dem Status,
- der gegenwärtigen Zeit, und
- einer Liste der externen Änderungen seit dem letzten Schritt

und liefert einen neuen Status.

Die Berechnung des neuen Status erfolgt in drei Stufen:

1. Ausführung aller Aktionen, welche durch externe Veränderungen (z.B. aufgrund externer Signale) erforderlich sind. Dies beinhaltet neue Werte von Bedingungen und Daten, aber keine neuen Zustände.
2. Berechnung einer maximalen Menge von CTs, welche ausführbereit und nicht konfliktbehaftet sind. In dieser Stufe werden die neuen Werte von Variablen (einschließlich der neu einzunehmenden Zustände) berechnet, aber zunächst nur einem internen Zwischenspeicher zugewiesen.
3. Ausführung der CTs. In dieser Stufen werden die zwischengespeicherten Werte als neuer Status übernommen.

Die Trennung in die Stufen 2 und 3 bewirkt, dass sich Änderungen aufgrund ausgeführter CTs nicht in demselben Schritt bei der Ausführung einer anderen CT auswirken können. Diese Trennung ist auch **zentral für die Vermeidung des Nichtdeterminismus**, der sich sonst bei unbekannter Reihenfolge der Simulation von Teilen der Beschreibung ergeben würde. Dieses **entscheidende Problem** der Simulation (d.h. der Semantik) paralleler Systme zeigt die Abb. 2.25.

Wird in einer einschrittigen Simulation zunächst der linke Zustand und dessen Übergänge berechnet, so werden `a, b` beide zu `1`. Wird der rechte Zustand zuerst evaluiert, so werden `a, b` beide zu `0`. Um diese Unsicherheit zu vermeiden, werden in einer ersten Phase (dem o.a. Schritt 2) zunächst alle neuen Werte von Variablen und die neu einzunehmenden Zustände berechnet und konzeptuell Hilfsvariablen zugewiesen. In einer weiteren Phase (dem o.a. Schritt 3) erfolgt dann die Zuweisung an die eigentlichen Variablen. Dadurch erhalten `a,b` in dem o.a. Beispiel stets denselben Wert, der sich durch eine Vertauschung der Werte der beiden Variablen ergibt.

Ähnliches gilt für die Abhängigkeit der Transitionen von Zuständen (vgl. Abb. 2.26).

Aufgrund der Trennung in die Stufen 2 und 3 können die Übergänge aus `s1` und `s2` in einem Schritt stattfinden. Ansonsten hinge es von der Simulationsreihenfolge ab, welcher der beiden Übergänge ausgeführt

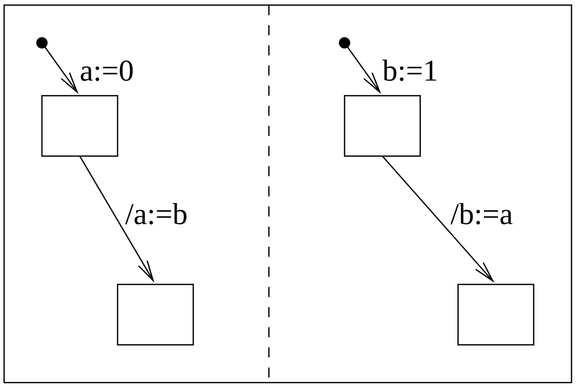


Abbildung 2.25: Problem des Nichtdeterminismus einschrittiger Simulation paralleler Systeme

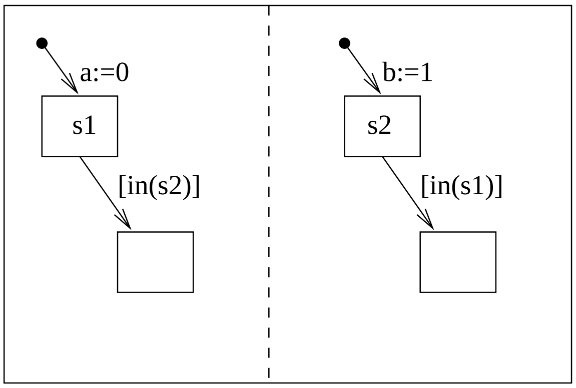


Abbildung 2.26: Zum Problem des Nichtdeterminismus bei Zustandswechseln

werden würde. Die beschriebene Simulationssemantik entspricht auch in natürlicher Weise der Funktion synchroner Schaltungen, läßt sich also mit diesen auch effizient realisieren.

Wir werden sehen, dass eine Form der zweistufigen Simulation auch bei einer anderen parallelen Sprache, nämlich VHDL, benutzt wird.

Die benutzte Form der Semantikbeschreibung ist operational. Sie beseitigt sicher viele Unklarheiten der ursprünglichen Beschreibung von STATEMATE. Weitere Details finden sich bei Harel [HN96]. Für die formale Verifikation wäre aber eine mathematisch präzisere Form der Semantikbeschreibung (z. B. mittel sog. denotatieller Semantik) wünschenswert.

2.2.3 Anwendungsbeispiel Ampelsteuerung

Als Beispiel einer Modellierung betrachten wir eine Ampelsteuerung. Die Ampel soll den Verkehr auf einer Kreuzung zwischen Haupt- und Nebenstraße steuern. Es soll zwei Programme geben: Bei Programm A erhalten Haupt- und Nebenstraße abwechselnd jeweils 2 Minuten bzw. eine halbe Minute grün. Beim Programm B erhält die Hauptstraße grün, bis das Signal 'Nebenstraße voll' eintrifft. Die einzelnen Ampeln werden durch die Variablen $\texttt{Rot}_H$, $\texttt{Gelb}_H$, $\texttt{Grün}_H$, $\texttt{Rot}_N$, $\texttt{Gelb}_N$ und $\texttt{Grün}_N$ dargestellt. Im StateCharts-Modell werden auf oberster Ebene zunächst die Lichter, die Kontrolle und der Fehlerzustand beschrieben (siehe Abb. 2.27).

Im Fehlerzustand blinken dabei die gelben Ampeln (siehe Abb. 2.28).

Der Ablauf im Normalbetrieb (siehe Abb. 2.29) beginnt mit einer Gelbphase für alle. Danach werden die Übergänge ausschließlich von Timern gesteuert, bis auf die Übergänge in die Gelbphase von Haupt- und Nebenstraße. Diese Übergänge hängen über die Signale `a` und `b` davon ab, welches Programm läuft.

Die Signale `a` und `b` werden im hierarchischen Zustand `Kontrolle` erzeugt (siehe Abb. 2.30). Der in dieser Graphik mit `C` beschriebene Kreis dient lediglich der Übersichtlichkeit. Er kann entfallen, wenn die beiden vertikalen Kanten bereits im linken kleinen Rechteck beginnen.

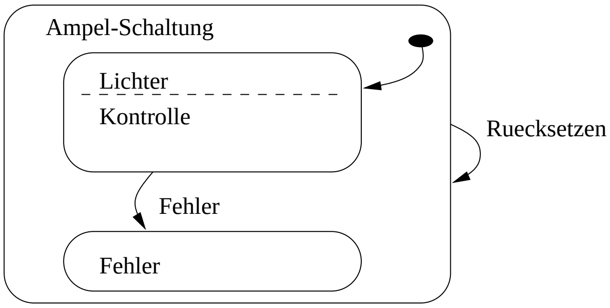


Abbildung 2.27: Globale Sicht der Ampelsteuerung

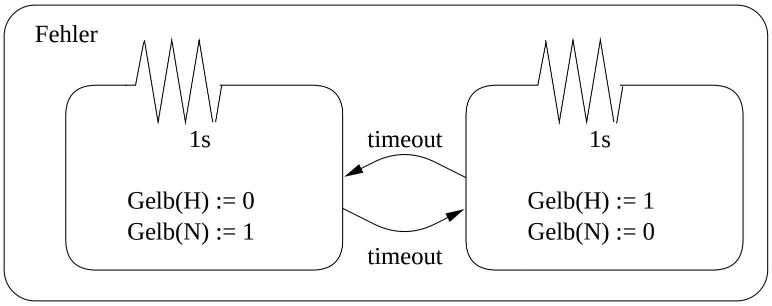


Abbildung 2.28: Fehlerzustand der Ampelsteuerung

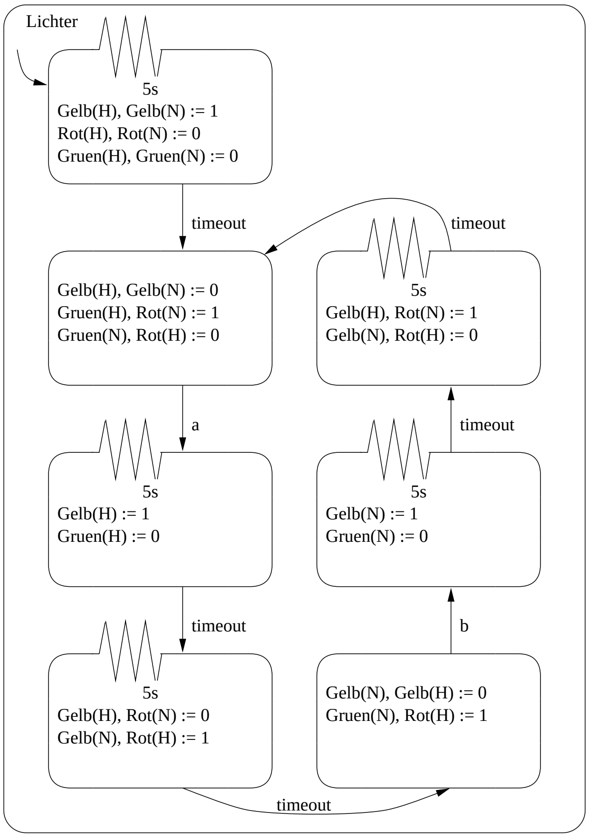


Abbildung 2.29: Teilsystem 'Lichter'

2.2.4 Bewertung

Positiv zu bewerten ist bei StateCharts die Möglichkeit einer beliebig tiefen Hierarchie von Zuständen, verbunden jeweils mit der Wahl zwischen exklusiven oder mit UND-verknüpften Teilzuständen.

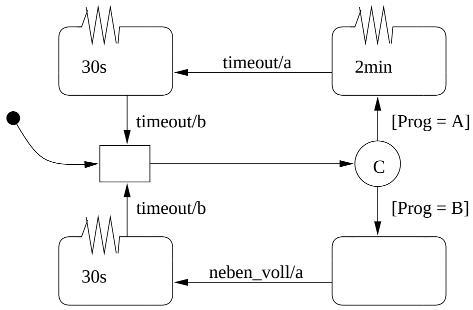


Abbildung 2.30: Teilsystem 'Kontrolle'

Nachteilig ist, dass keine Programm-Konstrukte zur Beschreibung komplexer Berechnungen existieren. Weiter gibt es keine Möglichkeit, Hardware-Strukturen zu darzustellen. Außerdem können nicht-funktionelle Eigenschaften nicht spezifiziert werden.

Semantische Unklarheiten, welche ursprünglich ein Nachteil von StateCharts waren, sind durch die o.a. Semantik von STATEMATE beseitigt.

2.3 VHDL

2.3.1 Übersicht

Alternativ zur Verwendung graphischer Spezifikationen kann man auch textuelle Beschreibungen in maschinenlesbaren Sprachen benutzen. Zu diesem Zweck sind verschiedene Hardware-Beschreibungssprachen (engl. *hardware description languages*, HDLs) entwickelt worden. Sehr verbreitet ist die Sprache VHDL (VHSIC Hardware Description Language) [IEE88, IEE92]. VHSIC wiederum steht für *very high speed integrated circuit*. Das VHSIC-Progamm geht auf eine Initiative des amerikanischen Verteidigungsministeriums (DoD) zurück. 1980 wurde dieses Programm initiiert, 1983 wurde ein Konsortium der Firmen IBM, Intermetrics und Texas Instruments mit der Sprachdefinition beauftragt. 1984 wurde die erste Sprachdefinition abgeschlossen. 1987 wurde eine überarbeitete Version als IEEE Standard 1076 beschlossen (publiziert: 1988). Da IEEE eine Überprüfung der Standards im Abstand von fünf Jahren vorschreibt, erfolgte 1992 die Verabschiedung eines ergänzten Standards (publiziert: 1993).

Eine der wesentlichsten Aufgaben einer Hardwarebeschreibungssprache ist die Modellierung der Parallelarbeit in den verschiedenen Hardwarebausteinen. VHDL benutzt hierfür **Prozesse**. Jeder Prozess modelliert einen Teil der Parallelarbeit in der Hardware. Für einfache Bausteine reicht vielfach ein einziger Prozess aus; komplexe Hardwarebausteine müssen mit mehreren Prozessen modelliert werden. Prozesse kommunizieren untereinander über Signale, die im Wesentlichen den physikalischen Leitungen entsprechen sollen. Da Signale in VHDL aber bei Abwesenheit neuer Zuweisungen beliebig lange ihren Wert behalten, werden sie vielfach wie Variable benutzt. Da Signale von mehreren Hardwarebausteinen getrieben werden können (z.B. im Falle von TriState- oder Open-Collector-Bausteinen), ist die Möglichkeit der Berechnung des resultierenden effektiven Wertes über eine spezielle Sorte von Funktionen, den sog. *resolution functions* gegeben.

Die Sichtbarkeit von Signalen entspricht im Wesentlichen der Sichtbarkeit von globalen Variablen, wie man sie im Zusammenhang mit der Kommunikation und Synchronisation von Prozessen in Betriebssystemen betrachtet. Aus diesem Kontext ist bekannt, dass eine unkoordinierte Ausführung von Prozessen, die auf globale Variable zugreifen können, je nach Ausführungsreihenfolge zu unterschiedlichen Ergebnissen führen kann, also zu **Nichtdeterminismus**. Da verschiedene Simulationen und Simulatoren möglichst zu demselben Ergebnis kommen sollen, versucht man, dies zu vermeiden. Interessanterweise entspricht die Lösung, die man bei VHDL benutzt, weitgehend der von StateCharts: in einer zweiphasigen Simulation werden zunächst die neuen Werte globaler Daten berechnet und anschließend in einer zweiten Phase zugewiesen. Diese Phasen werden wiederholt, bis sich keine Änderungen mehr ergeben. Im Kontext von VHDL wird die Verzögerung zwischen zwei Phasen *delta delay* genannt.

Eine weitere wichtige Aufgabe einer Hardwarebeschreibungssprache ist die explizite **Modellierung der Zeit**. Vielfach bieten Sprachen hier dimensionslose Einheiten (z.B. normale *integer*) an, deren Interpretation dem Benutzer überlassen ist. VHDL geht hier einen Schritt weiter und unterstützt eine dimensionsbehaftete Zeit einschließlich der Vereinbarung der Umrechnung zwischen den verschiedenen Einheiten wie Millisekunden, Nanosekunden usw. Problematisch ist allerdings die Entscheidung, dass eine Implementierung nur 32 Bit zur Darstellung von Zeiten zu verwenden braucht.

VHDl ist weiterhin in dem Bestreben entworfen, möglichst wenig Einschränkungen hinsichtlich möglicher Modelle zu besitzen. Vielfach gibt es daher Möglichkeiten, Modellierungsstile in VHDL selbst festzulegen, wie beispielsweise die Anzahl der benutzten Logikwerte.

U.a. als Folge davon ist VHDL ist eine sehr komplexe Sprache. VHDL baut auf der Sprache ADA (und damit indirekt auf PASCAL) auf und bietet eine Vielzahl von Sprachelementen, was den Blick auf die soeben beschriebenen wesentlichen Elemente leicht versperrt. Es ist ratsam, die übrigen Sprachelemente im Kontext dieser wesentlichen Elemente zu sehen und sich von den syntaktischen Elementen nicht zu sehr ablenken zu lassen[3].

2.3.2 Ein einführendes Beispiel: der Volladdierer

5. Vorles.

In VHDL wird jede zu modellierende Einheit (jeder Hardware-Baustein) als *design entity* oder *VHDL entity* bezeichnet. Eine Einheit setzt sich aus zwei Bestandteilen zusammen, der **Bausteindeklaration** (*entity declaration*) und (ggf. mehreren) **Architekturen** (*architectures*).

[3]Die VHDL-Syntax ist im Übrigen nicht prüfungsrelevant.

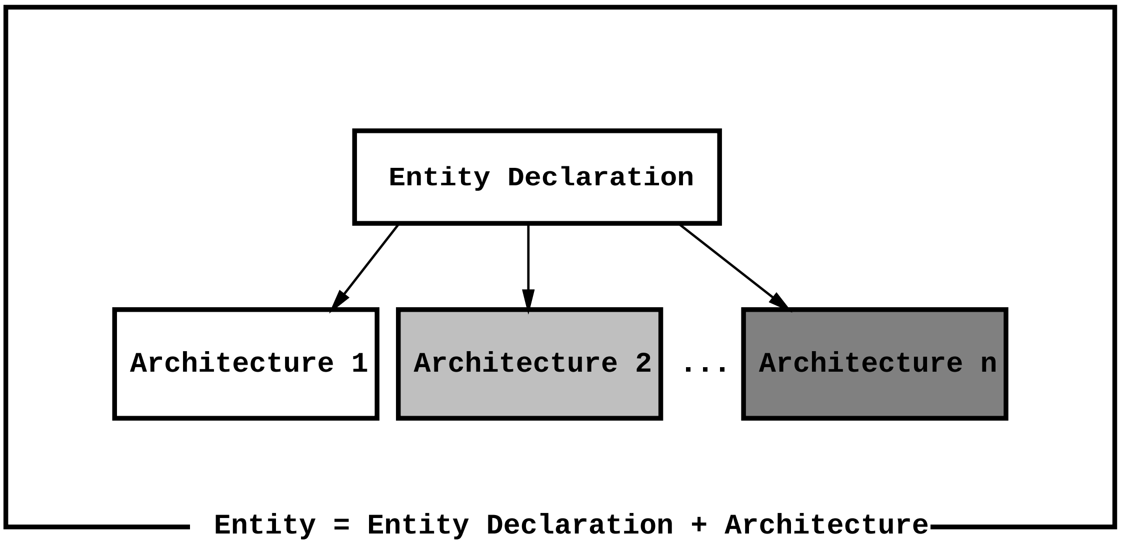


Abbildung 2.31: Entity

Standardmäßig wird in der Bearbeitung die zuletzt analysierte Architektur benutzt. Mittels expliziter Angabe der zu verwendenden Architektur kann man davon abweichen.

Als Beispiel betrachten wir den Volladdierer mit Ein- und Ausgabesignalen nach Abb. 2.32.

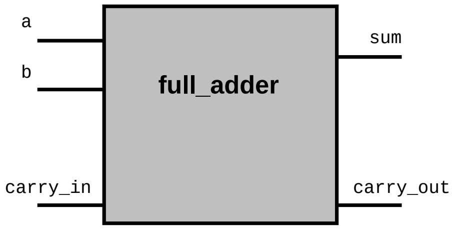


Abbildung 2.32: Volladdierer

Die zur Abbildung 2.32 korrespondierenden Signale werden in der *entity declaration* nach dem Schlüsselwort `PORT` beschrieben:

```
ENTITY full_adder IS               -- entity declaration
  PORT (a, b, carry_in: IN Bit;    -- input ports
        sum, carry_out: OUT Bit);  -- output ports
END full_adder;
```

Architekturbeschreibungen bestehen in VHDL aus einem Kopf und einem Rumpf. Bei den Rümpfen kann man zwischen verschiedenen Beschreibungsstilen unterscheiden, nämlich zwischen Verhaltens-Rümpfen, Struktur-Rümpfen und Mischungen von diesen beiden Stilen. Am Beispiel eines Volladdierers wollen wir im Folgenden beide Formen von Rümpfen vorstellen.

Ein **Verhaltensrumpf** beschreibt die Bildung der Ausgangssignale in Abhängigkeit von den Eingangssignalen (und bei sequentiellen Schaltungen auch den Einfluss interner Zustände). Die Zuweisung zu Signalen wird mit <= kenntlich gemacht:

```
ARCHITECTURE behavior OF full_adder IS  -- architecture
 SIGNAL s: Bit;
 BEGIN
  s          <= a xor  b AFTER 5 Ns;
  sum        <= s xor carry_in AFTER 5 Ns;
  carry_out  <= (a and b) or (s and carry_in) after 10 Ns;
 END behavior;
```

Im Unterschied dazu beschreiben **Strukturrümpfe** den inneren Aufbau von Einheiten, z.B. durch Gatter (siehe Abb. 2.33).

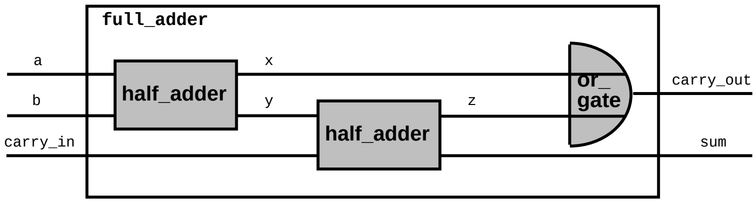


Abbildung 2.33: Struktur-Beschreibung eines Bausteins

Diese Gatter müssen (zumindest in VHDL'87) als *components* bekannt gemacht werden. Auf diese Weise können die Schnittstellen der Gatter auch dann definiert werden, wenn sie (wie es beim *top-down*-Entwurf vorkommen kann) noch nicht in der Bibliothek eingetragen sind. *component*-Beschreibungen ähneln daher *forward*-Deklarationen anderer Programmiersprachen.

Über sog. *port maps* kann dann die Verschaltung der Komponenten untereinander erklärt werden:

```
ARCHITECTURE structure OF full_adder IS
  COMPONENT half_adder
    PORT (i1, i2 : IN Bit; carry  :OUT Bit; sum :OUT Bit);
  END component;
  COMPONENT or_gate
    PORT (i1, i2:IN Bit; o:OUT Bit);
  END component;
  SIGNAL x, y, z: Bit;
 BEGIN
  s1: half_adder PORT MAP (a, b, x, y);
  s2: half_adder PORT MAP (y, carry_in, z, sum);
  s3: or_gate    PORT MAP (x, z, carry_out);
 END structure;
```

VHDL ist v.a. zur Simulation entwickelt worden. Abb. 2.34 zeigt das Beispiel eines Simulationslaufs für den Volladdierer.

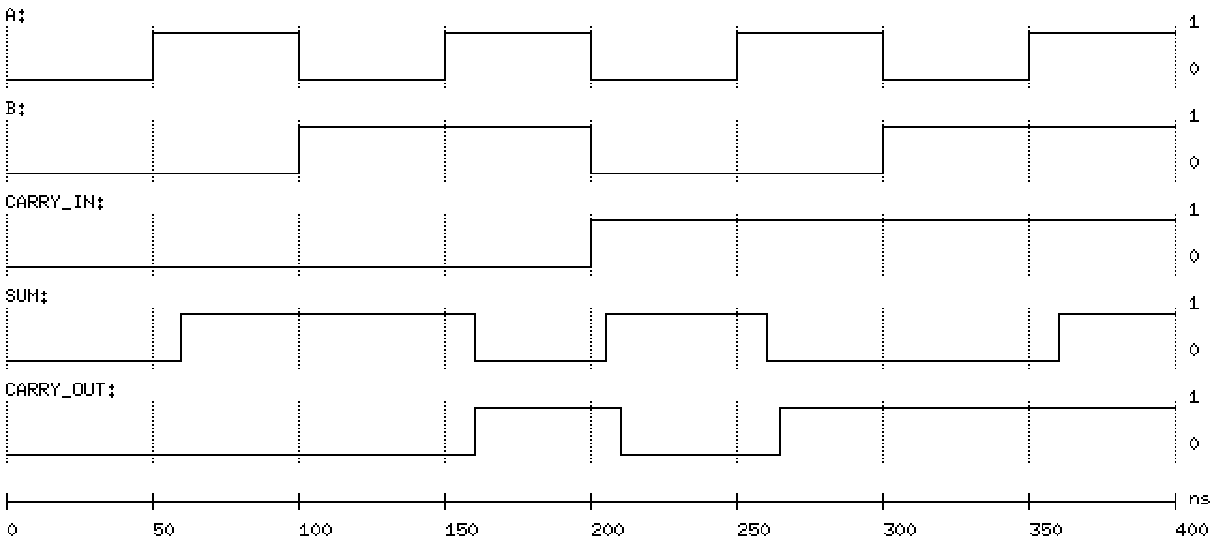


Abbildung 2.34: Simulationsergebnisse als Ablaufdiagramm

Nach diesem einführenden Beispiel kommen wir jetzt zu einer detaillierteren Betrachtung der einzelnen Sprachelemente.

2.3.3 Syntax

Eine Erläuterung der Syntax von VHDL kann leicht umfangreich und wenig interessant werden. Wir benötigen jedoch eine Kenntnis der Syntax und der Semantik, u.a. um den Zwischenschritt bei der Abbildung von STATEMATE in Hardware-Schaltungen zu erklären. Wir werden dabei ein Schwergewicht auf jene Sprachelemente legen, die sich von Softwaresprachen unterscheiden.

2.3.3.1 Lexikalische Elemente

Für Bezeichner gelten weitgehend übliche Standards von Programmiersprachen. Folgende Regeln gelten für **Bezeichner** in VHDL:

- Das erste Zeichen eines Bezeichners muß ein Buchstabe sein.
- Alle anderen Zeichen sind Buchstaben, Ziffern oder der Unterstrich (_).
- Zwei Unterstriche dürfen nicht aufeinander folgen.
- Das letzte Zeichen darf kein Unterstrich sein.
- Zwischen Groß- und Kleinbuchstaben wird nicht unterschieden.

Beispiele: `A_b, was_soll_das_bedeuten`

Kommentare werden durch zwei aufeinanderfolgende "–"-Zeichen eingeleitet und reichen bis zum Ende der Zeile.

Beispiel: `a := 4712; -- dies ist keine Schleichwerbung`

Literale enhalten einige typische Besonderheiten von Hardwaresprachen: die Unterstützung verschiedener Zahlenbasen, die komfortable Unterstützung von Bitvektoren, der unterschiedlichen Konventionen zur Nummerierung von Bits sowie physikalische Größen. In VHDL gibt es konkret die folgenden Literale:

- **Integer–Literale** werden durch eine Folge von Ziffern gebildet und können einen Exponenten zur Basis 10 beinhalten. Außerdem dürfen Unterstriche zur Verbesserung der Lesbarkeit eingefügt werden.

  Beispiele: `2, 22E3, 23_456_789, 2#100110#` (Integer-Literal zur Basis 2)

- **Fließkommaliterale** müssen einen Punkt enthalten. Der Exponent darf auch negativ sein.

  Beispiele: `1.0` (Fließkommaliteral zur Basis 10), `33.6e-4` (Fließkommaliteral zur Basis 10)

- **Zeichenliterale** bestehen aus einzelnen Zeichen umgeben von einfachen Anführungsstrichen.

  Beispiel: `'c'`

- **Zeichenfolgenliterale** sind von doppelten Anführungsstrichen eingeschlossene Folgen von ASCII-Zeichen.

  Beispiel: `"ABC"` (Zeichenfolgenliteral).

- **Bitfolgenliterale** sind Zeichenfolgenliterale mit einem eingeschränkten Zeichensatz. Ein Indikator für die Basis und Unterstriche sind erlaubt.

  Beispiele:

  `B"1111_1111"` (binäres Bitfolgenliteral),
  `"1111_1111"` (binäres Bitfolgenliteral),
  `X"F0A7"` (hexadezimales Bitfolgenliteral)

- **Physikalische Literale** bestehen aus einem Integer– oder Fließkommaliteral und einer Maßeinheit.

  Beispiel: `35 Ns` (physikalisches Literal)

2.3.3.2 Objekte und Typen

2.3.3.2.1 Objekte: Im Bereich der Objekte kommen zu den üblichen Konstanten und Typen noch Signale, welche Leitungen darstellen sollen. VHDL unterscheidet daher im wesentlichen zwischen drei Klassen von Objekten, den "Behältern" für Daten, nämlich

- **Konstanten:** Für diese ist eine einmalige Wertzuweisung bei der Deklaration erlaubt.
- **Signale:** Diese stellen ein Modell von Leitungen dar, können allerdings im Gegensatz zu echten Leitungen Werte beliebig lange speichern.
- **Variable:** Diese besitzen eine Bedeutung wie in üblichen Programmiersprachen. Die Zuweisung an Variable erfolgt sofort. In VHDL'88 sind sie nur lokal für VHDL-Prozesse erlaubt.

Exemplare der angeführten Objektklassen können deklariert werden. In solchen Deklarationen sind verschiedene Typangaben zulässig. Neben den typischen PASCAL-artigen Typen sind v.a. physikalische Typen erlaubt.

Beispiele:

```
CONSTANT pi : Real := 3.14;
VARIABLE mem : neu;
SIGNAL s, u: Bit := '0';
```

2.3.3.2.2 Skalare Typen: Es gibt in VHDL die folgenden skalaren Typen:

Skalarer Typ

- **Aufzählungstypen**

  Beispiele:
  ```
  TYPE Bit is ('0', '1');
  TYPE Boolean is (False, True);
  ```
- **Integer-Typen**: Diese stellen ganzen Zahlen in einem implementationsabhängigen Intervall dar. Weitere Integer-Typen sind durch Einschränkung dieses Intervalls zu bilden.

  Beispiele:
  ```
  TYPE byte_int IS RANGE 0 TO 255;
  TYPE bit_index IS RANGE 31 DOWNTO 0;
  ```
- **Physikalische Typen**

  Beispiel:
  ```
  TYPE Time IS range -2147483647 TO 2147483647
  UNITS
       Fs;                -- femtosecond
       Ps   =  1000 Fs;   -- picosecond;
       Ns   =  1000 Ps;   -- nanosecond
  END units ;
  ```
- **Fließkommatypen**

  Beispiel: `TYPE probability IS RANGE 0.0 TO 1.0 ;`

2.3.3.2.3 Zusammengesetzte Typen: In VHDL gibt es, wie üblich, Feld- und Strukturtypen.

Zusammen-gesetzte Typen

- **Feldtypen:** Felder mit festen Grenzen werden weitgehend wie in üblichen Programmiersprachen angegeben.

  Beispiel: `TYPE word IS ARRAY (15 DOWNTO 0) OF Bit;`

  Eine Besonderheit sind Feldtypen, bei denen die Größe des Feldes noch offen gelassen ist. Die beiden vordefinierten Feldtypen sind von dieser Art:

  ```
  TYPE String IS ARRAY (Positive RANGE <>) OF Character;
  TYPE Bit_Vector IS ARRAY (Natural RANGE <>) OF Bit;
  ```

"Positive" und "Natural" sind vordefinierte Subtypen.

Der Typ `Bit_Vector` wird der Anwendung im Hardware-Bereich wegen unterstützt und im Folgenden noch häufiger benötigt.

Die Zeichenfolge "<>" ist ein Platzhalter für ein Indexintervall, das erst dann angegeben wird, wenn der entsprechende Datentyp benutzt wird. Wenn man z.B. ein Feld von 16 Bit-Objekten benötigt, kann man dies wie folgt angeben:

```
SIGNAL vector : Bit_Vector (0 TO 15) ;
```

- **Strukturtypen:** Auch Strukturtypen werden weitgehend wie in üblichen Programmiersprachen angegeben.

  Beispiel:

```
TYPE  register_bank IS RECORD
      F0, F1: Real;
      R0, R1: Integer;
      A0, A1: Address;
      IR: Instruction;
END   RECORD;
```

2.3.3.3 Der Alias–Mechanismus

Bei der Beschreibung von Hardware muss häufiger auf einzelne Teile eines Bitvektor zugegriffen werden. VHDL stellt hierfür einen Mechanismus zur Verfügung, der es ermöglicht, Teile eines Datenobjekts mit verschiedenen Namen anzusprechen. Dies ist z.B. dann sinnvoll, wenn man häufig die verschiedenen Teile eines Befehlswortes einzeln referenzieren muß.

Sei z.B. ein Instruktionsregister `ir` als `SIGNAL` wie folgt deklariert

```
SIGNAL ir: Bit_Vector (15 DOWNTO 0) ;
```

und ein Befehlswort gemäß Abbildung 2.35 unterteilt.

Bits:	15....13	12............11	10......0
Bedeutung:	Opcode	Registernummer	Distanz

Abbildung 2.35: Befehlsformat

Dann kann man die einzelnen Teile des Instruktionsregisters mit den Bezeichnern `ir_opcode`, `ir_reg` und `ir_dist` ansprechen, wenn im entsprechenden Deklarationsteil die folgenden Anweisungen stehen:

```
ALIAS ir_opcode:  Bit_Vector (2 DOWNTO 0) IS ir (15 DOWNTO 13) ;
ALIAS ir_reg:     Bit_Vector (1 DOWNTO 0) IS ir (12 DOWNTO 11) ;
ALIAS ir_dist:    Bit_Vector (10 DOWNTO 0) IS ir (10 DOWNTO 0) ;
```

2.3.3.4 Attribute

Manchen Elementen einer VHDL–Beschreibung können sogenannte Attribute zugeordnet werden. Attributierbare Elemente sind:

Typen und Subtypen
Prozeduren und Funktionen
Signale, Variable und Konstanten
Entities, Architekturen, Konfigurationen und Pakete
Komponenten
Anweisungsmarken

Attribute beschreiben Eigenschaften des zugehörigen Elements und sind entweder vordefiniert oder vom Benutzer definiert. Ein Element kann beliebig viele Attribute haben. Die Schreibweise für Attribute ist:

`elementname'attributname`

Benutzerdefinierte Attribute werden mit der `ATTRIBUTE`-Anweisung definiert und mit Werten versehen:

```
ATTRIBUTE Cost :     Integer ;
ATTRIBUTE Cost alu:  ENTITY IS 22;
```

Einige wichtige vordefinierte Attribute sind:

F'Left(i) : Dieses Attribut enthält die linke Schranke der i-ten Dimension des Feldes (der Parameter ist optional und default ist 1).

F'Right(i) : Dieses Attribut enthält die rechte Schranke der i-ten Dimension des Feldes (der Parameter ist optional und default ist 1).

F'High(i) : Dieses Attribut enthält die obere Schranke der i-ten Dimension des Feldes F (der Parameter ist optional und default ist 1).

F'Low(i) : Dieses Attribut enthält die untere Schranke der i-ten Dimension des Feldes F (der Parameter ist optional und default ist 1).

S'Stable : Dieses Attribut liefert den Wert `True`, falls das Signal `S` zum gegenwärtigen Simulationszeitpunkt denselben Wert wie beim vorhergehenden Zeitpunkt hat.

S'Event : Dieses Attribut liefert den Wert `True`, falls das Signal `S` zum gegenwärtigen Simulationszeitpunkt einen anderen Wert als beim vorhergehenden Zeitpunkt hat.

2.3.3.5 Vordefinierte Operatoren

Tabellen 2.1 zeigt in VHDL vordefinierte arithmetische Operatoren.

Gruppe	Symbol	Funktion	Argument-Typen
Arithmetik (binär)	+	Addition	Integer, Fließkomma, physik. Typ
	−	Subtraktion	
	*	Multiplikation	Integer, Fließkomma, 1 Op. physik.
	/	Division	
	mod	Modulus	
	rem	Rest	
	**	Exponentiation	
Arithmetik (unär)	+	Unäres Plus	
	−	Unäres Minus	
	abs	Absolutwert	

Tabelle 2.1: Arithmetische Operationen in VHDL

Zu beachten ist, dass die arithmetischen Operationen nur auf *integers*, Fließkommazahlen und physikalischen Typen definiert sind. Damit gelten für diese Operationen die Einschränkungen des Zahlenbereichs der konkreten implementationsabhängigen Realisierung von *integers*. Arithmetische Operationen auf Bitvektoren, für die eine größere Wortlänge möglich ist, bedürfen der Definition entsprechender Arithmetikpakete.

Für physikalische Typen sind nur solche Operatoren anwendbar, welche die Dimension erhalten.

Die Tabelle 2.2 zeigt logische Operatoren.

Schließlich zeigt die Tabelle 2.3 Vergleichsoperatoren und die Konkatenation.

Für Bitvektoren sind damit keine Arithmetikoperationen vordefiniert. Diese müssen erst in entsprechenden Paketen definiert werden.

2.3.3.6 Bausteindeklarationen

Wir kommen jetzt zur syntaktischen Beschreibung von Bausteindeklarationen, die wir am Beispiel bereits kennengelernt haben. Diese besitzen die folgende Syntax:

Gruppe	Symbol	Funktion	Argument-Typen
Logik (binär)	**and**	log. und	Bit, Boolean, 1-dim. Felder
	or	log. oder	
	nand	**not and**	
	nor	**not or**	
	xor	exkl.–oder	
Logik (unär)	**not**	Komplement	

Tabelle 2.2: Logische Operationen in VHDL

Gruppe	Symbol	Funktion	Argument-Typen
Vergleiche	=	Gleichheit	beliebiger Datentyp außer Dateityp
	/=	Ungleichheit	
	<	kleiner als	
	>	größer als	Skalar, 1-dim. Feld e. Aufz.-typs
	<=		
	>=		
	&	Konkatenation	1-dim. Feldtyp oder Element davon

Tabelle 2.3: Vergleichsoperationen und die Konkatenation in VHDL

`ENTITY` *bausteinname* `IS GENERIC (` *Parameterliste* `); PORT (` *Schnittstellenliste* `);` *Deklarationen* `BEGIN` *Anweisungen* `END` *bausteinname*;

Parameterlisten erlauben die Definition von **generischen** Bausteinen. Diese Bausteine enthalten Parameter (z.B. Wortbreiten und Verzögerungszeiten), die erst bei der Instantiierung festgelegt werden.

Beispiel:

```
ENTITY bausteinname IS
GENERIC (wordlength : Integer; delay1: Time) ;
PORT    (base : IN Real; exp : IN Positive; result : OUT Real) ;
END;
```

Schnittstellenlisten beschreiben Ein- und Ausgänge. Beschriebene Eigenschaften sind die folgenden:

- Die Objektart des Schnittstellenobjekts (`SIGNAL`, `CONSTANT` oder `VARIABLE`).
- Die Richtung des Datenflusses durch die Schnittstelle (`IN`, `OUT`, `INOUT` oder `BUFFER`). `BUFFER` muss benutzt werden, wenn man auf einen Ausgang intern auch lesend zugreifen möchte.
- Der Datentyp der Schnittstelle.

Die Syntax von **Schnittstellenlisten** -einschließlich optionaler Anteile- ist die folgende:

Objektart Bezeichnerliste : Richtung Datentyp := Default-Wert

Beispiele:

```
carry_in: Bit;
a, b: Bit;
CONSTANT pi: Real := 3.14
SIGNAL a,b: in Bit := '0'
```

Die **Deklarationen** dürfen in diesem Fall (zumindest in VHDL'87) keine Variablen enthalten. Diese können daher nicht zur Kommunikation der verschiedenen Prozesse innerhalb eines Bausteins benutzt werden. Dies ist nur über Signale möglich, für die der Zuweisungsmechanismus ein deterministisches Verhalten sicherstellt. In VHDL'92 dürfen die Deklarationen 'globale' Variable enthalten, Zugriffe darauf bilden (wie von Betriebssystemen her bekannt) einen **kritischen Abschnitt**. Um ein deterministisches Verhalten des Simulators sicherzustellen, muss auf globale Variablen über **Monitore** zugegriffen werden.

Eine Bausteindeklaration kann nach dem Schlüsselwort `BEGIN` sogar gewisse Anweisungen enthalten. Diese Anweisungen sollen aber nicht das dynamische Verhalten des Bausteins beschreiben. Hier kann man Anweisungen unterbringen, die bestimmte Bedingungen überprüfen. Wenn z.B. zwei Eingangsleitungen eines Bausteins nicht gleichzeitig gesetzt sein dürfen, dann kann diese Überprüfung bereits hier notiert werden. Dies hat wiederum den Vorteil, daß nicht jeder einzelne Architekturrumpf eine solche Überprüfung enthalten muß.

2.3.3.7 Architekturrümpfe

Der Aufbau von Architekturrümpfen sieht wie folgt aus:

```
ARCHITECTURE rumpfname OF bausteinname IS
Deklarationen
BEGIN
Anweisungen
END rumpfname ;
```

Für die Benutzung von Variablen in den Deklarationen gilt dasselbe wie bei *entities*.

Architekturrümpfe können Strukturrümpfe, Verhaltensrümpfe oder eine Mischung aus beidem sein. Zunächst beschreiben wir hier die Sprachelemente zur Modellierung von Struktur.

2.3.3.8 Strukturrümpfe

Da bei einem Top-Down-Design echte Bibliotheksbausteine ggf. nicht bekannt sind, spezifiziert die *component*-Anweisung nur ein Muster einer Schnittstelle. Erst später erfolgt eine Assoziierung mit echten Bibliotheksbausteinen.

Syntax:

```
COMPONENT comp_typname
GENERIC Parameterliste ;
PORT Schnittstellenliste ;
END COMPONENT;
```

Signal-Definitionen dienen der Beschreibung der Verbindungsleitungen.

Beispiel: `SIGNAL l1, l2: Bit ;`

Die **Komponenteninstantiierung** dient der Beschreibung verfügbarer Bausteinexemplare.

Syntax:

```
comp_name : comp_typname
GENERIC MAP Assoziationsliste ;
PORT MAP Assoziationsliste ;
```

Innerhalb der **Assoziationsliste** kann sowohl eine Names- als auch eine Stellungsassoziation von E/A-Signalen erfolgen.

Beispiel:

```
COMPONENT and_gate
PORT (in1, in2 : IN Bit; result: OUT Bit);
END COMPONENT;
SIGNAL a, b, c, d, e, f : Bit ;
and1 : and_gate PORT MAP (a, b, c) ;
and2 : and_gate PORT MAP (result => d, in1 => e, in2 = > f) ;
```

2.3.3.9 Verhaltensrümpfe

Das wesentliche Element eines Verhaltensrumpfes ist der Prozess. Andere Sprachelemente sind vielfach nur abkürzende Schreibweisen für Prozesse. Prozesse sollen die nebenläufige Ausführung von Aktionen in verschiedenen Teilen von Hardware modellieren.

Syntax:

prozessmarke `: -- optional`
`PROCESS`
Deklarationen `-- optional`
`BEGIN` *Anweisungen* `-- optional`
`END PROCESS ;`

Innerhalb von Prozessen dürfen Deklarationen in jeder Version von VHDL Variable enthalten.

Anweisungen in Prozessen werden sequentiell abgearbeitet bis der Prozeß auf eine `WAIT`–Anweisung trifft. Wenn die letzte Anweisung in einem Prozeß abgearbeitet wurde, dann wird danach wieder zur ersten Anweisung gesprungen.

`WAIT`-Anweisungen erlauben es, Prozesse zu suspendieren, bis eine spezifizierte Situation eintritt. Es gibt die folgenden Möglichkeiten:

1. `WAIT ON` *Signalliste* ; Suspendierung bis sich eine Signal in einer Liste ändert.
2. `WAIT UNTIL` *Bedingung* ; Suspendierung bis eine Bedingung erfüllt ist, z.B. `enable = '1';`
3. `WAIT FOR` *Dauer* ; Suspendierung für eine bestimmte Zeit.
4. `WAIT` ; Unbegrenzte Suspendierung des Prozesses

Anstelle von expliziten `WAIT`-Anweisungen kann eine Liste von Signalen im Prozesskopf benutzt werden. Dann werden Prozesse aktiviert, wenn sich ein Signal dieser Liste ändert.

Beispiel:

```
PROCESS (x, y) BEGIN
prod <= x AND y ;
END PROCESS;
```

Der Rumpf dieses Prozesses wird nach jeder Änderung der `x` oder `y` zugewiesenen Werte ausgeführt.

2.3.3.10 Zuweisungen

Es gibt zwei Formen von Zuweisungen:

1. **Variablenzuweisungen**: Variablenzuweisungen haben dieselbe Wirkung wie in üblichen Programmiersprachen.

   Syntax: `variablen_name :=` *Ausdruck* ;

2. **Signalzuweisungen**: Diese Zuweisungen werden entsprechend der Durchlaufzeiten in realer Hardware verzögert ausgeführt (siehe Abschnitt "Semantik"). Signale behalten in VHDL ihren Wert bis zur nächsten Zuweisung. Reale Leitungen haben diese Speichereigenschaft nicht. Aufgrund der Speichereigenschaft werden Signale in VHDL häufig wie Variable benutzt.

   Syntax:

   signal_name `<=` *Ausdruck* `;`
   signal_name `<=` *Ausdruck* `AFTER` *Dauer* `;`
   signal_name `<= TRANSPORT` *Ausdruck* `AFTER` *Dauer* `;`
   signal_name `<= REJECT` *Zeit* `INERTIAL` *Ausdruck* `AFTER` *Dauer* `;`

   Ausdruck muss ein Wert vom Datentyp des Signals *signal_name* und *Dauer* muß vom Datentyp `Time` sein.

   Mehrere Zuweisungen innerhalb **eines** Prozesses an dasselbe Signal sind zulässig. Zu jedem Signal existiert je Prozess ein sog. **Treiber** (engl. *driver*). Jeder Teiber speichert den für die Zukunft bekannten Teil des **Werteverlaufs** (sog. *projected waveform*). Dieser Werteverlauf wird durch Tupel aus Signalwerten und zugehörigen Zeitpunkten beschrieben.

   Trifft ein Simulator während der Ausführung eines Prozesses auf eine Signalzuweisung, so wird zunächst nur der aus dem *Ausdruck* resultierende Wert in den Werteverlauf übertragen. Bei fehlender `AFTER`-Klausel wird die aktuelle Zeit benutzt, sonst die aus der `AFTER`-Klausel berechnete.

Def.: Ein Treiber ist während eines Simulationszyklus **aktiv**, wenn ihm in diesem Zyklus ein Wert aus dem projektierten Werteverlauf des Signals zugewiesen wird, unabhängig davon, ob der neue Wert auch schon der alte Wert war.

Def.: Ein Signal ist während eines Simulationszyklus **aktiv**, wenn einer seiner Treiber aktiv ist.

Def.: Es entsteht ein **Ereignis** (engl. *event*), wenn einem in einem Simulationszyklus einem aktiven Signal ein Wert zugewiesen wird, der vom alten Wert verschieden ist.

2.3.3.11 Transportverzögerung und träge Verzögerung

Bezüglich der Wirkung einer Signalzuweisung auf den vorhergesagten Werteverlauf muss noch hinsichtlich des Zeitmodells unterschieden werden:

1. Bei einer reinen **Transportverzögerung** (engl. *transport delay*) wird das Verhalten einer Schaltung modelliert, welche beliebig kurze Impulse übertragen kann, wie dies annähernd bei einem Kabel der Fall ist (siehe Abb. 2.36).

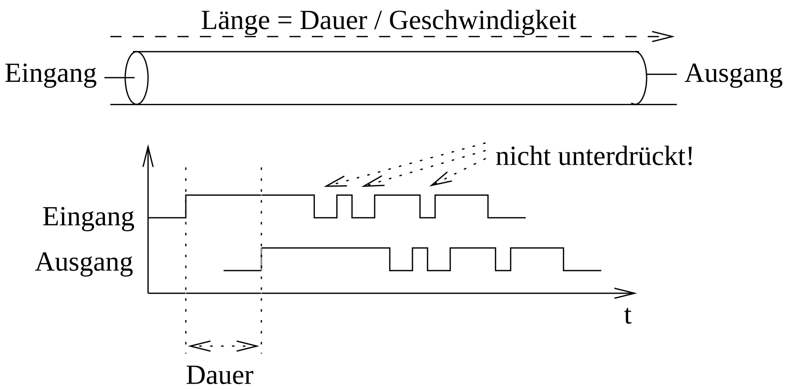


Abbildung 2.36: Transportverzögerung

Diese Form der Verzögerung kann in VHDL durch das reservierte Wort `TRANSPORT` ausgedrückt werden.

Beispiel:

signal_name `<= TRANSPORT` *Ausdruck* `AFTER` *Dauer* `;`

2. Bei einer **trägen Verzögerung** (engl. *inertial delay*) werden Impulse, die kürzer als eine gewisse Mindestzeit andauern, unterdrückt (siehe Abb. 2.37).

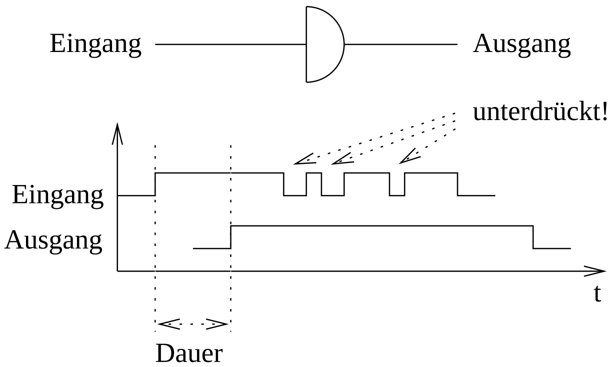


Abbildung 2.37: Träge Verzögerung

Standardmäßig ist diese Mindestzeit gleich der Verzögerungszeit der `AFTER`-Klausel. Eine kürzere Zeit kann durch den Zusatz `REJECT` in Kombination mit der expliziten Spezifikation des `INERTIAL`-Delay angegeben werden.

Beispiel:

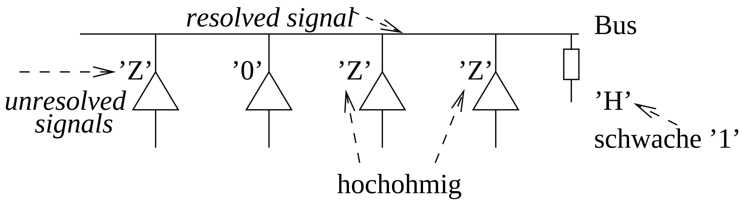
signal_name <= `REJECT` *Zeit* `INERTIAL` *Ausdruck* `AFTER` *Dauer* ;

Diese Modelle in entsprechende Operationen auf den Tupeln des vorhergesagten Werteverlaufs umzusetzen, ist recht subtil. Erst kürzlich erschien eine systematische Analyse dieses eigentlich klassischen Aspekts von Simulatoren [YH97].

2.3.3.12 Resolved Signals

Mehrere Zuweisungen in **verschiedenen** Prozessen sind für die bislang definierten Datentypen unzulässig. Diese Konstruktion ist nur dann zulässig, wenn das Signal ein **berechnetes Signal** (engl. *resolved signal*) ist. Prozesse können die verschiedenen Treiber für ein Signal modellieren.

Beispiel:



Sofern ein Signal mehrere Treiber besitzt, muss aus den Werten für die einzelnen Treiber ein Wert für das Signal ausgerechnet werden. Zu diesem Zweck können entsprechende Berechungsfunktionen (*resolution functions*) in Subtyp-Vereinbarungen angegeben werden.

Beispiel:

Die folgende Vereinbarung erklärt `std_logic` zu einem Untertyp, der sich durch Aufruf der Funktion `resolved` aus dem Typ `std_ulogic` ergibt.

```
SUBTYPE std_logic IS resolved std_ulogic;
```

Die Berechnungsfunktion selbst erhält die verschiedenen Signalquellen in Form eines Vektors:

```
TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;
...
FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_logic;
```
[4]

2.3.3.13 IF-Statements

Die Kontrollstrukturen innerhalb von Prozessen entsprechen weitgehend denen üblicher sequentieller Programmiersprachen.

`IF`-Statements werden in VHDL stets mit `END IF` beendet. In geschachtelten `IF`-Statements werden weitere Tests mit `ELSIF` eingeleitet.

Beispiel: `IF a=3 THEN b:=z; d:=e ELSIF a=5 THEN b:=z ELSE b:=e END IF;`

2.3.3.14 CASE-Statements

`CASE`-Statements klammern in VHDL den selektierenden Wert mit `WHEN` und `=>`.

Beispiel:

```
CASE opcode IS
WHEN 1 => result <= a + b ;
WHEN 2 => result <= a - b ;
WHEN OTHERS => result <= b ;
END CASE;
```

[4] Tatsächlich gibt das VHDL-Paket, welches `std_logic` enthält, den Typ `std_ulogic` zurück, wobei aber nur die Elemente von `std_logic` tatsächlich benutzt werden.

2.3.3.15 Schleifen

Schleifen basieren in VHDL auf einer allgemeinen `LOOP`-Konstruktion, welche durch Präfixe optional zu `WHILE`- oder `FOR`-Schleifen werden können. Optional kann stets eine Schleifenmarke vorangestellt werden, welche bei geschachtelten Schleifen über `NEXT` und `EXIT`-Statements die nächste Schleifeniteration bzw. ein Verlassen einer bestimmten Schleife bewirken kann.

Beispiele:

`marke: LOOP`	`m: WHILE` *Bedingung* `LOOP`	`marke: FOR i IN 0 TO 10 LOOP`
Anweisungen;	*Anweisungen*;	*Anweisungen*;
`EXIT WHEN` *Bedingung*;	`EXIT WHEN` *Bedingung*;	`EXIT WHEN` *Bedingung*;
`NEXT marke WHEN` *Bedingung*;	`NEXT m WHEN` *Bedingung*;	`NEXT marke WHEN` *Bedingung*;
Anweisungen;	*Anweisungen*;	*Anweisungen*;
`END LOOP marke;`	`END LOOP m;`	`END LOOP marke;`

2.3.3.16 Funktionen und Prozeduren

Funktionen werden in VHDL weitgehend in ADA-Syntax notiert.

Beispiele:

```
FUNCTION hauptstr_farbe(z: IN zustände) RETURN farben IS
 BEGIN
  Statements
  RETURN Wert
 END hauptstr_farbe

FUNCTION Nat (a: IN Bit_Vector) RETURN integer IS
 -- Annahme: absteigender Index-Bereich für a;
 --          a(i) besitzt Gewicht 2**i
 VARIABLE res: integer:=0;
 VARIABLE factor : integer := 1;
  BEGIN
   FOR i IN a'low TO a'high LOOP
    IF a(i)='1' THEN
     res := res + factor;
    END IF;
    factor:=factor*2;
   END LOOP;
   RETURN res;
  END Nat;
```

Prozeduren können durch enthaltene `WAIT`-Statements suspendiert werden, was die interne Implementierung sehr komplex macht (es kann mehrere Keller mit Informationen über gerufene, aber suspendierte Prozeduren geben).

2.3.3.17 Pakete

Häufig genutzte Prozeduren und Funktionen können in Paketen abgelegt werden. Pakete enthalten wie *entities* eine Paketdeklaration. Im Unterschied zu *entities* ist nur ein Rumpf erlaubt.

Syntax:

```
PACKAGE paketname IS
Deklarationen
END paketname ;

PACKAGE BODY paketname IS
Definitionen
END paketname ;
```

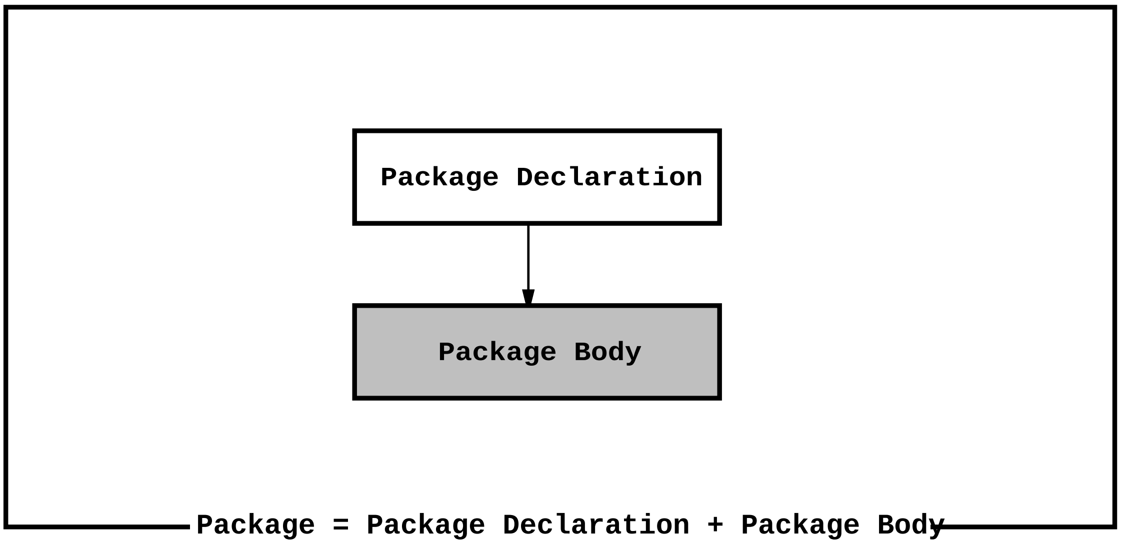


Abbildung 2.38: Bestandteile von Paketen

In einer Paketdeklaration dürfen unter anderem Typen, Konstanten, Signale, Unterprogramme und Komponenten deklariert werden. Der Paketrumpf hingegen enthält die Realisierungen der in der Paketdeklaration vereinbarten Unterprogramme.

Beispiel:

```
PACKAGE math IS
 FUNCTION Nat(a: IN Bit_Vector) RETURN integer;
  ...
END math;

PACKAGE BODY math IS
 Implementierung von Nat;
 ...
END math;
```

2.3.3.18 Konfigurationen

Da VHDL mehrere Architekturrümpfe zu einer Bausteindeklaration zuläßt, muß dem Anwender die Möglichkeit gegeben werden, das zu simulierende Sytem zu konfigurieren. D.h. er muß festlegen können, welche Architektur benutzt werden soll und welcher Bausteinentwurf zur Realisierung einer Komponente herangezogen werden soll.

Standardmäßig wird immer der zuletzt analysierte Architekturrumpf verwendet. Zur Realisierung einer Instanz einer Komponente wird nachgesehen, ob eine Entwurfseinheit mit dem Komponentennamen sichtbar ist.

Konfigurationen können eingesetzt werden, um vom Standard abweichende Zuordnungen vornehmen zu können,

Eine Konfigurationsdeklaration hat die folgende Form:

`CONFIGURATION konfigname OF bausteinname IS`
Deklarationen
Konfiguration
`END konfigname ;`

Ein Beispiel wäre das folgende:

```
CONFIGURATION beispiel OF testumgebung IS
 FOR struktur
```

```
  FOR  baustein1:anforderungsgenerator
   USE entity ampelbib.anforderungsgenerator;
  END FOR;
  FOR baustein2:ampelschaltung
   USE entity ampelbib.ampelschaltung(zwei);
  END FOR;
 END FOR;
END beispiel;
```

Die Konfiguration muß jedoch nicht direkt innerhalb eines Architekturrumpfes festgeschrieben werden. Um die Konfiguration eines Systems flexibel zu halten, kann man diese auch zunächst offen lassen und dann in einer anderen Entwurfseinheit — einer Konfigurationsdeklaration — beschreiben. So kann man völlig verschiedene Konfigurationen zusammenschalten und simulieren lassen, ohne die zugrundeliegenden Entwurfseinheiten verändern zu müssen.

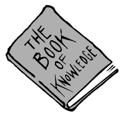
2.3.3.19 Entwurfseinheiten

Unter einer Entwurfseinheit (engl. *design unit*) versteht man einen zusammenhängenden Block von Anweisungen, der vom Analyseprogramm einzeln bearbeitet werden kann. Es gibt in VHDL zwei Sorten von Entwurfseinheiten:

- Primäre Entwurfseinheiten
  - Bausteindeklaration
  - Paketdeklaration
  - Konfigurationsdeklaration
- Sekundäre Entwurfseinheiten
  - Architekturrumpf
  - Paketrumpf

Entwurfseinheiten werden in **Bibliotheken** gespeichert.

Regeln:



- In einer VHDL–Bibliothek muß jede primäre Entwurfseinheit einen eindeutigen Namen haben.
- Es darf mehrere sekundäre Entwurfseinheiten mit demselben Namen geben. Insbesondere darf eine sekundäre Entwurfseinheit auch einen bereits für eine primäre Einheit verwendeten Namen annehmen.
- Zusammengehörige primäre und sekundäre Entwurfseinheiten müssen in derselben Bibliothek stehen — dürfen also nicht getrennt werden.

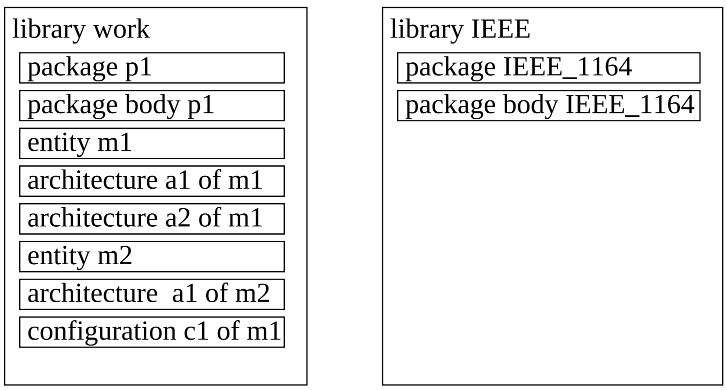


Abbildung 2.39: Speicherung von primären und sekundären Entwurfseinheiten

2.3.3.20 Sichtbarkeit von Entwurfseinheiten

Eine Entwurfseinheit und die darin enthaltenen Vereinbarungen sind für andere Entwurfseinheiten nicht automatisch sichtbar. Dies wird mit Hilfe der `USE`–Anweisung gemacht. Eine `USE`–Anweisung gilt immer nur für die folgende Entwurfseinheit.

```
USE bibname.unitname ;
USE unitname.objekt ;
USE bibname.unitname.objekt ;
USE bibname.unitname.ALL;
```

Ein besonderer Bibliotheksname ist `work`. `work` **bezeichnet immer die Bibliothek, in die das Analyseprogramm die erfolgreich analysierten Einheiten ablegt.**

Ein besonderer Objektname ist `ALL`. `ALL` steht für alle sichtbaren Objekte der Einheit.

2.3.3.21 Sichtbarkeit von Bibliotheken

Genau wie primäre Entwurfseinheiten sind auch Bibliotheken nicht automatisch sichtbar. Dies ist mit der `LIBRARY`–Anweisung möglich. Beispiel:

```
LIBRARY paketbib ;
USE paketbib.ampelpaket.ALL;
```

Vor jeder Entwurfseinheit steht implizit die Anweisung

```
LIBRARY work ;
```

sowie

```
LIBRARY std ;
USE std.standard.ALL;
```

2.3.4 Semantik

Bei der Erklärung der Bedeutung einer VHDL-Beschreibung interessiert uns v.a. wie für eine vorliegende VHDL-Beschreibung Eingaben zu bestimmten Zeitpunkten in Ausgaben zu anderen Zeitpunkten transformiert werden, d.h. uns interessiert das **dynamische** Verhalten. **Statische** Eigenschaften spiegeln v.a. die Übersetzung von VHDL-Beschreibungen wider und erfassen Fragen der Syntax, der Sichtbarkeit, der Organisation von Bibliotheken usw., die für das dynamische Verhalten weniger wichtig sind.

Der für dynamische Eigenschaften von VHDL wichtigste Begriff ist der des **Prozesses**. Prozesse sollen die nebenläufigen Aktivitäten in Hardware-Einheiten nachbilden.

VHDL-Prozesse sind mit zwei wichtigen Eigenschaften verknüpft:

- VHDL-Prozesse sind nicht-hierarchisch. VHDL geht von einem völlig "flachen System" von Prozessen aus. Eine Vater-Sohn-Beziehung wie in UNIX gibt es nicht.
- VHDL-Prozesse können nicht dynamisch erzeugt werden. Die Anzahl der Prozesse kann aus dem Quelltext abgelesen werden. Die einzige Möglichkeit, die Anzahl der zur Laufzeit relevanten Prozesse zur Laufzeit zu beeinflussen ist es, Prozesse mittels `WAIT`-Statements auf nicht eintretende Bedingungen warten und damit irrelevant werden zu lassen.

Das exakte Verfahren der Simulation von VHDL ist recht subtil. Aus diesem Grund soll hier ein Auszug aus dem Originaltext [IEE92] wiedergegeben werden:

The execution of a model consists of an initialization phase followed by the repetitive execution of process statements in the description of that model. Each such repetition is said to be a **simulation cycle**. *In each cycle, the values of all signals in the description are computed. If as a result of this computation an event occurs on a given signal, process statements that are sensitive to that signal will resume and will be executed as part of the simulation cycle.*

At the beginning of initialization, the current time, T_c is assumed to be 0 ns.

The initialization phase consists of the following steps:

- *The driving value and the effective value of each explicitly declared signal are computed, and the current value of the signal is set to the effective value. This value is assumed to have been the value of the signal for an infinite length of time prior to the start of the simulation.*
- *... implicit signals ...*
- *Each nonpostponed process in the model is executed until it suspends.*
- *Each postponed process*[5] *in the model is executed until it suspends.*
- *The time of the next simulation cycle (which in this case is the first simulation cycle), T_n is calculated according to the rules of step f of the simulation cycle, below.*

A simulation cycle consists of the following steps:

a) *The current time, T_c is set equal to T_n. Simulation is complete when $T_n = TIME'HIGH$ and there are no active drivers or process resumptions at T_n.*

b) *Each active*[6] *explicit signal in the model is updated. (Events*[7] *may occur as a result.)*

c) *... implicit ...*

d) *For each process P, if P is currently sensitive to a signal S and if an event has occured on S in this simulation cycle, then P resumes.*

e) *Each nonpostponed process that has resumed in the current simulation cycle is executed until it suspends.*

f) *The time of the next simulation cycle, T_n is determined by setting it to the earliest of*

   1. *TIME'HIGH*
   2. *The next time at which a driver becomes active, or*
   3. *The next time at which a process resumes.*

   If $T_n = T_c$, then the next simulation cycle (if any) will be a delta cycle.

g) *If the next simualtion cycle will be a delta cycle, then the remainder of this step is skipped. Otherwise, each postposed process that has resumed but has not been executed since the last resumption is executed until it suspends. Then, T_n is recalculated according to the rules of step f. It is an error if the execution of any postponed process causes a delta cycle to occur immediately after the current simulation cycle.*

Eine übliche Kurzfassung dieser Beschreibung ist die folgende [KB95]:

The standard VHDL simulation cycle (following the .. relatively simple ... description in the text of the standard ..) runs each process until it becomes blocked in a wait statement. Zero logical time has elapsed to this point. Logical time is then advanced to the next pending assignment time, and the assignment is executed. The conditions of each wait statement are then checked, and if any are satisfied the appropriate process bodies are allowed to execute (in zero logical time) until they all become blocked again. Then logical time is advanced again, and so on.

Die simulierte Zeit ist also zunächst die Zeit $t = 0ns$. Signale und Variable werden mit dem Default-Wert initialisiert, sofern dieser angegebenen ist. Zu Beginn einer VHDL-Simulation sind weiterhin zunächst alle Prozesse aktiv.

Für jeden simulierten Zeitpunkt t werden jetzt die folgenden zwei Phasen und der daran anschließende Test wiederholt (s.a. Abb. 2.40).

[5] Dies sind in VHDL'93 eingeführte Prozesse, die für jeden Zeitpunkt t einmalig ausgeführt werden, nachdem alle Delta-Zyklen abgeschlossen sind.

[6] Siehe Definition auf Seite 38.

[7] Siehe Definition auf Seite 38.

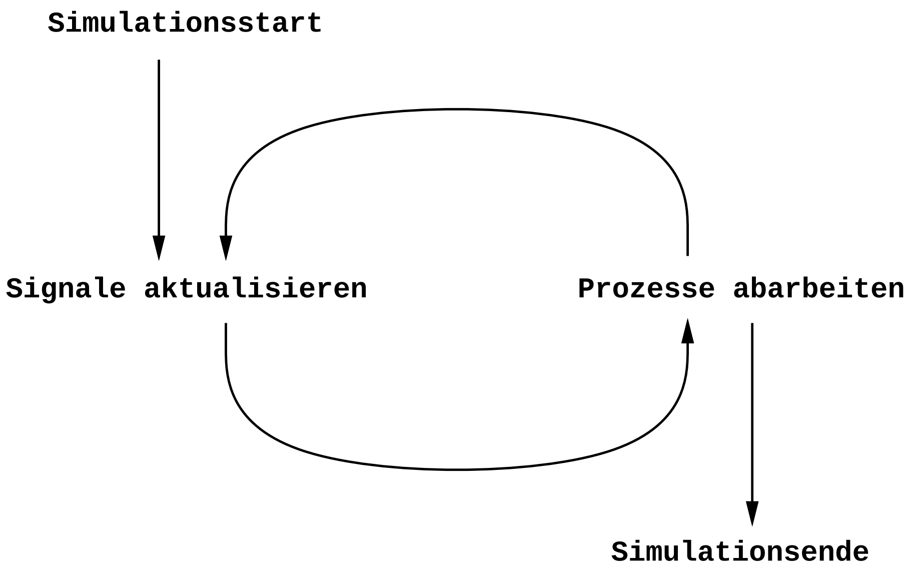


Abbildung 2.40: Die Simulationsphasen von VHDL

1. Die jeweils aktiven Prozesse werden ausgeführt. Dabei werden alle Variablenzuweisungen sofort realisiert. Signalzuweisungen führen zu einem Eintrag des Wertes der rechten Seite der Zuweisung in eine Liste zukünftiger Ereignisse[8]. Falls keine `AFTER`-Klausel angegeben ist, ist die zukünftige Zeit gleich der aktuellen Zeit t.

2. In einer zweiten Phase werden alle für den aktuellen Zeitpunkt in die Liste eingetragenen Ereignisse durch Zuweisung der berechneten Werte an die linken Seiten der Signalzuweisungen realisiert.

Test Falls sich dadurch Änderungen von Signalen ergeben, auf die Prozesse **sensitiv** sind, wird ein weiteres Mal die erste Phase ausgeführt. Dabei wird die Zeit um eine kleine Zeit δ erhöht, die kleiner ist als alle echten Zeiten. Die beiden Phasen werden wiederholt, bis für den Zeitpunkt t $(+\ n * \delta)$ keine Ereignisse mehr in der Ereignisliste eingetragen sind. In diesem Fall wird die simulierte Zeit auf die nächstgrößere Zeit t' in der Liste erhöht und wie oben fortgefahren.

Die obige Beschreibung der Bedeutung von VHDL ist nicht sehr präzise und basiert auf dem Modell der Ereignisliste. Es gibt inzwischen Arbeiten, (siehe z.B. [KB95]), in denen die Semantik von VHDL mit größerer Präzision beschrieben wird. Allerdings hat sich gezeigt, dass es nicht eine einzige formale Semantik-Beschreibung von VHDL geben kann, die alle Zwecke erfüllt. Vielmehr muss je nach Anwendung eine geeignete formale Beschreibung benutzt werden.

Die Vorstellung der δ-Zeiteinheiten soll die Abbildung 2.41 verdeutlichen.

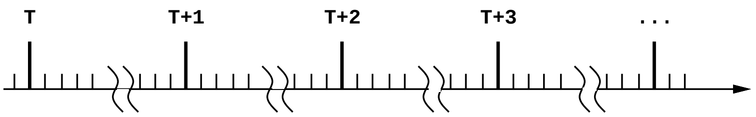


Abbildung 2.41: Anordnung von Delta-Einheiten auf der Zeitachse

In der normalen Anwendung des *delta delay* terminiert das Alternieren der beiden Simulationsphasen, wie an folgenden Beispielen zu sehen.

1. ```
   SIGNAL a,b,c : Bit := '0';
   PROCESS BEGIN
    b <= '1';
    c <= '1' AFTER 1 Ns;
    WAIT for 1 Ns;
   ```

[8]In gewissen Fällen müssen bei dieser Gelegenheit auch bereits eingetragene Ereignisse wieder entfernt werden.
   ```

```
 a <= b;
 b <= a;
 WAIT for 1 Ns;
END PROCESS ;
```

Nach jeweils einer Iteration wird die Delta-Simulationsschleife beendet:

	0 ns	0 ns + 1δ	1 ns	1 ns + 1δ	2 ns
a	0	0	0	1	Wiederholung
b	0	1	1	0	ab Prozess-
c	0	0	1	1	beginn

Im Gegensatz dazu terminiert die Schleife nie, falls eine Zuweisung der Art `a <= NOT a;` in einem Prozess enthalten ist.

2. Als etwas komplexeres Beispiel betrachten wir eine sog. *end-around carry*-Schaltung, wie sie bei der Benutzung des Einer-Komplements eingesetzt wird. Bei der Addition im Einerkomplement ist ein bei der Addition entstehender Übertrag in der in Form einer zusätzlichen Addition einer `1` zu berücksichtigen (siehe Abb. 2.42). Bekanntlich kann es durch die Addition nicht zu einem erneuten Überlauf kommen.

x	int1k(x)
"1110"	-1
+ "1110"	-1
"11100"	-3
+ "1"	Korrektur
"1101"	-2

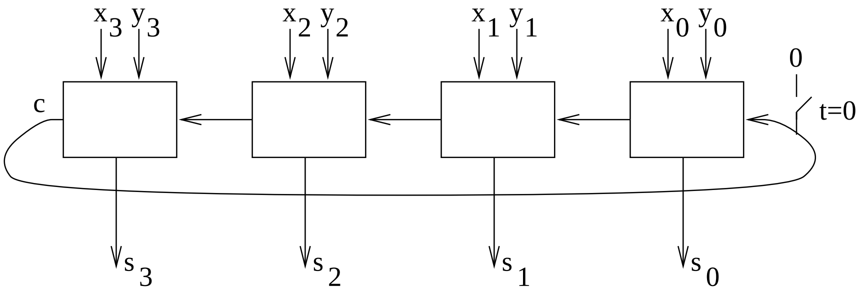


Abbildung 2.42: Addition im Einerkomplement

Die Schaltung enthält eine asynchrone Rückkopplung, deren Simulation nicht-trivial ist.

In VHDL könnte man die Schaltung wie folgt beschreiben:

```
PROCESS eac (x,y)
 SIGNAL c: Bit:='0';
 BEGIN
 WAIT FOR 1 Ns;
 s <= x + y + c;                  -- Annahme: + für Bitvektoren und Bits definiert
 c <= ('0' & x) + ('0' & y) (4)  -- Bit 4
 END;
```

Tabelle 2.4 zeigt den Simulationsablauf

	1 ns	1 ns + δ	1 ns + $2*\delta$
x	"1110"	"1110"	"1110"
y	"1110"	"1110"	"1110"
c	0	1	1
s	?	"1100"	"1101"

Tabelle 2.4: Ausnutzung des *delta delay* beim *end-around carry*

2.3.5 Übersetzung von STATEMATE nach VHDL

Es gibt kommerzielle Programme, welche VHDL-Beschreibungen unter gewissen Einschränkungen hinsichtlich des Sprachumfangs in Hardware abbilden (z.B. von den Firmen Synopsys, Mentor und Viewlogic).

Sofern es gelingt, StateChart-ähnliche Spezifikationen in VHDL zu übersetzen, ist damit ein vollständiger Weg von der Spezifikation zur Hardware gegeben. Leider liefern derartige Übersetzer bislang VHDL-Beschreibungen, die nur schwer in effiziente Schaltungen zu transformieren sind.

Als Beispiel betrachten wir hier eine Schaltung, der sequentiell ein Bitvektor übergeben werden kann. Dieser Bitvektor wird in das Zweierkomplement konvertiert, indem ab der ersten erhaltenen '1' alle Bits komplementiert werden.

Abb. 2.43 zeigt das Zustandsdiagramm dieses Automaten.

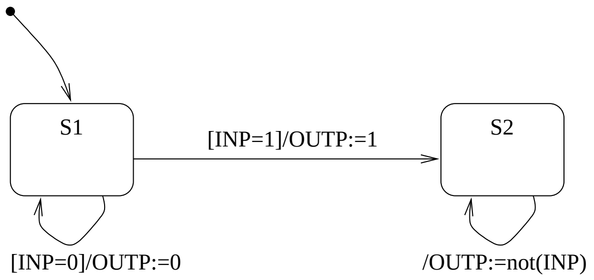


Abbildung 2.43: Zustandsgraph einer Schaltung zur sequentiellen Bildung des Zweierkomplements

Die folgende VHDL-Beschreibung zeigt, wie STATEMATE eine solche Beschreibung nach VHDL übersetzt.

```
-- Created: Tue Oct  6 17:34:29 1998
-- File Name: /home/ls12r/landwehr/Statemate/uebung_wa/vhdl/prof2/prof2_ENTITY2.vhdl
library WORK;
use WORK.SYNOPSYS.all;
use WORK.STC_SYNOPSYS_PACKAGE.all;
entity ENTITY2  is
    port (
       CLOCK:                  in      bit;
       INP:                    in      bit;
       OUTP:                   out     bit  );
end ENTITY2;
architecture Arch_ENTITY2 of ENTITY2 is
  type tpChart_TWOS_COMPLEMENT_states is (S1, S2);
  signal  Chart_TWOS_COMPLEMENT_isin: tpChart_TWOS_COMPLEMENT_states;
begin
  exec_all_process  :  process
  begin
    wait until CLOCK'event and CLOCK='1';
    case Chart_TWOS_COMPLEMENT_isin is
      when  S1 =>
        if CONV_INTEGER(INP) = 0 then
          OUTP <= '0';
          Chart_TWOS_COMPLEMENT_isin <= S1;
        elsif CONV_INTEGER(INP) = 1 then
          OUTP <= '1';
          Chart_TWOS_COMPLEMENT_isin <= S2;
        end if;
      when  S2 =>
        OUTP <= not INP;
        Chart_TWOS_COMPLEMENT_isin <= S2;
    end case;
  end process exec_all_process;
end Arch_ENTITY2;
```

Die `wait`-Anweisung beschreibt die Taktabhängigkeit des Automaten, die `case`-Anweisung die jeweiligen Zustandsübergänge und Ausgaben. Die Funktion `CONV_INTEGER` ist erforderlich, da die Spezifikation sich nicht ausschließlich auf die Benutzung des Datentyps `bit` beschränkt, sondern diesen Typ mit ganzen Zahlen mischt.

Die generierte Ausgabe beschränkt sich auf die Untermenge von VHDL, die von SYNOPSYS-Synthesewerkzeugen akzeptiert wird, erlaubt also mit Hilfe dieser Werkzeuge die Generierung von Hardware.

2.3.6 Mehrwertige Simulation und der IEEE-Standard 1164

6. Vorles.

Simulationsverfahren unterscheiden sich stark hinsichtlich der Menge möglicher Signalwerte. Wir wollen uns im Folgenden auf diskrete Werte, die für eine Simulation von digitalen Systemen ausreichen, beschränken [9].

2.3.6.1 2-wertige Simulation (1 Signalstärke)

Naheliegend ist zunächst einmal die 2-wertige Simulation mit den Werten '0' und '1'. Diese Simulation ist aber für die meisten mikroelektronischen Systeme nicht ausreichend, da z.B. die in der Praxis vorkommenden hochohmigen Ausgänge von Tristate- und Open-Collector-Schaltkreisen nicht angemessen modelliert werden können.

2.3.6.2 3- und 4-wertige Simulation (2 Signalstärken)

Um elektrisch hochohmige Signale darstellen zu können, ergänzt man die Menge der Signalwerte um den Wert 'Z'. Dieser Wert bedeutet, daß das betreffende Signal nicht durch irgendwelche Schaltungsteile zur Annahme eines logischen Signalwertes gezwungen wird.

Kommen allerdings Signalquellen (in VHDL *Treiber* genannt) zusammen, von denen eine den Wert `'Z'` und eine andere `'0'` oder `'1'` liefert, so ergibt sich als Resultat dieser andere Wert. `'Z'` ist damit schwächer als die beiden anderen Werte und wir haben es mit insgesamt 2 Signalstärken zu tun.

Die resultierende 3-wertige Simulation ist in verschiedenen Simulatoren realisiert worden.

In den meisten Fällen wird die Menge {'0', '1', 'Z'} von Signalwerten um einen weiteren Wert 'X' zur Darstellung unbestimmter Signale erweitert. Hierfür gibt es im wesentlichen drei Interpretations-Möglichkeiten, nämlich

- die Interpretation "entweder '0' oder '1' ",
- die Interpretation "elektrisch weder '0' noch '1' " sowie
- die Interpretation "'0', '1' oder ein anderer, elektrisch unbestimmter Wert".

Wir wollen uns hier der dritten Auffassung anschließen. Einen systematische Betrachtung von Wertemengen ermöglicht die sog. CSA-Theorie nach Hayes [Hay82]. Ziel der CSA–Theorie ist die geschlossene Darstellung der Schalterebene. Gemischte Darstellungen z.B. aus Transistoren und Stick–Diagrammen (wie bei Mead und Conway [MC80]) sollen vermieden werden. Die wesentlichen Komponenten der Theorie sind die Verbindung (connector, C), der Schalter (switch, S), der Abschwächer (attenuator, A), sowie ein Element zur Repräsentation von Kondensatoren.

Die Menge der Signalwerte besteht aus vier Elementen: '0', '1', undefiniert ('X') und hochohmig ('Z'):

$$V_4 = \{\texttt{'0', '1', 'X', 'Z'}\}$$

Abb. 2.44 a) beschreibt, wie diese Werte durch eine Verbindung verknüpft werden.

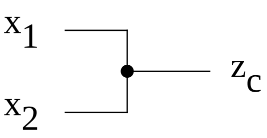
Die Systematik hinter dieser Verknüpfung wird erkennbar, wenn man die Signalwerte nach Signalstärken ordnet. Offensichtlich gilt:

$$'0' >' Z','1' >' Z','X' >' 1','X' >' 0'$$

Diese partielle Ordnung stellt Abb. 2.44 b) dar. Mit der zugehörigen reflexiven Ordnung "$\geq$" ist $\{V_4, \geq\}$ ein Verband [Hay82]. `'X'` und `'Z'` stellen das Eins– bzw. Nullelement dieses Verbandes dar. Das Verhalten der Verbindung entspricht der Supremumbildung im Verband:

$$z_c = \sup(x_1, \ldots, x_n)$$

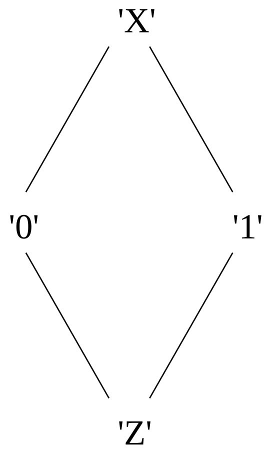
[9] Die folgenden Abschnitte sind aus dem Buch "Synthese und Simulation von VLSI-Systemen" [Mar93] in den vorliegenden Text übertragen worden. Dabei erfolgte eine Anpassung an die Bezeichnungen in üblichen VHDL-Bibliotheken.



x_1	x_2	z_c
'0'	'0'	'0'
'0'	'1'	'X'
'0'	'Z'	'0'
'1'	'0'	'X'
'1'	'1'	'1'
'1'	'Z'	'1'
'Z'	'0'	'0'
'Z'	'1'	'1'
'Z'	'Z'	'Z'
'X'	d	'X'
d	'X'	'X'

d ∈ {'0','1','X','Z'}

a)



b)

Abbildung 2.44: Verhalten einer Verbindung (a), Partielle Ordnung (b)

Abb. 2.45 zeigt die zwei Formen möglicher Schalter und das zugehörige Verhalten. Dabei gilt normalerweise: $K \in \{\texttt{'0'},\texttt{'1'}\} = V_4 - \{\texttt{'X'},\texttt{'Z'}\}$.

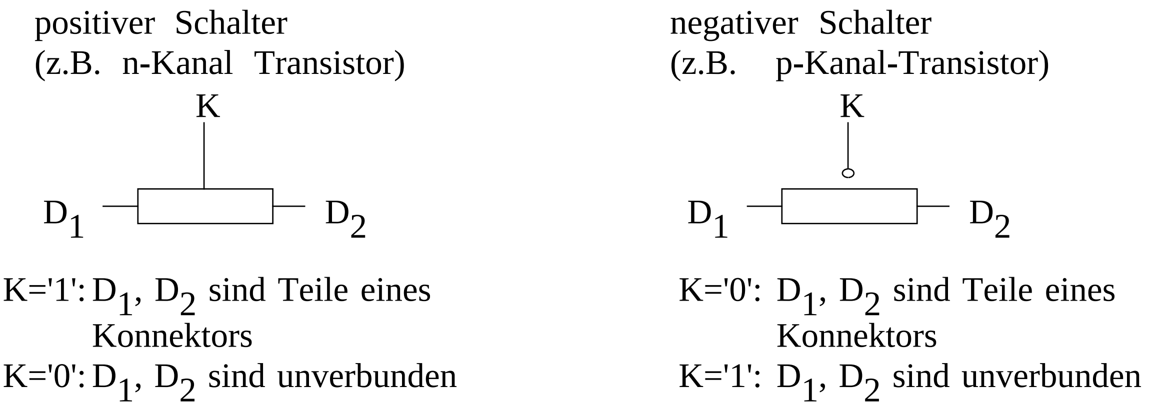


Abbildung 2.45: Schalter in der CSA-Theorie

2.3.6.3 7-wertige Simulation (3 Signalstärken)

NMOS-Schaltkreise enthalten neben den "normalen" geschalteten Transistoren noch Transistoren vom Verarmungstyp (engl. *depletion transistor*). Diese Transistoren werden quasi wie Widerstände eingesetzt. Sie schwächen Signale so ab, daß sie schwächer sind als die Signale an den Ausgängen von normalen Transistoren.

Um die Wechselwirkung zwischen schwachen und starken Signalenwerten beschreiben zu können, führen wir schwache Werte `'L', 'H'` und `'W'` ein. Die Wirkung des Zusammenkommens von Signalen kann wieder übersichtlich über die *sup*-Funktion dargestellt werden. Es gilt:

$$\text{sup('Z','L') ='L',}\quad \text{sup('Z','H') = 'H',}\quad \text{sup('L','H') = 'W',}$$

$$\text{sup('1','L') ='1',}\quad \text{sup('0','H') = '0',}\quad \text{sup('W','Z') = 'W'}$$

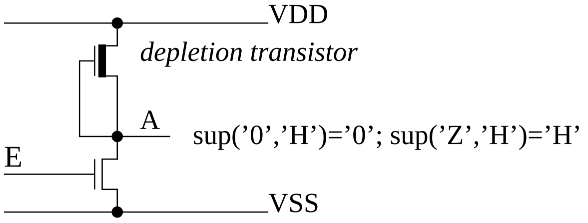


Abbildung 2.46: NMOS-Inverter mit Depletion-Transistor

Graphisch kann man die Beziehungen zwischen den Signalstärken in einem Verbandsdiagramm darstellen (siehe Abb. 2.47).

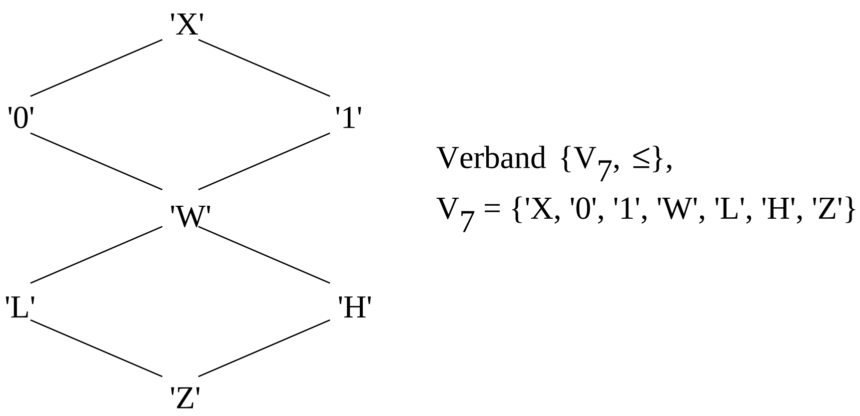


Abbildung 2.47: Verband $\{V_7, \leq\}$

Bausteine, die aus starken Signalen schwache generieren, heißen in der CSA-Theorie Abschwächer (engl. *attenuators*). Verarmungstransistoren sind also ein Spezialfall solcher Bausteine. Bausteine, die aus schwachen Signalen starke generieren, heißen Verstärker (engl. *amplifier*).

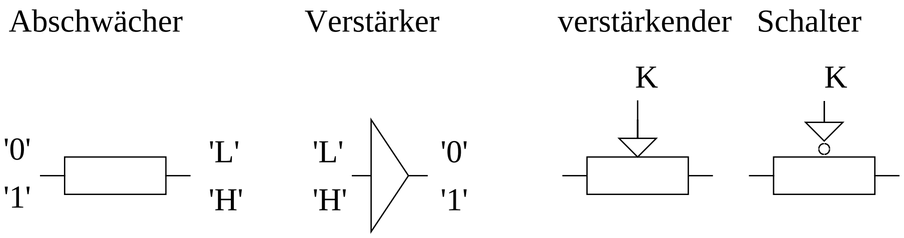


Abbildung 2.48: Bausteine zur Wandlung der Signalstärke

2.3.6.4 10-wertige Simulation (4 Signalstärken)

In MOS-Schaltkreisen können logische Signalwerte auf Leitungen (zumindest für gewisse Zeit) erhalten bleiben, wenn alle angeschlossenen Transistoren gesperrt sind. Die Kapazität der Leitung kann damit zur Speicherung genutzt werden. Ein Beispiel zeigt die Abbildung 2.49.

Es handelt sich um eine Stufe einer sog. *Manchester carry chain*, die zur Addition eingesetzt werden kann [MC80]. In einer ersten Phase, während der das Signal ϕ='1' ist, wird die carry-Leitung für c_{i+1} aufgeladen. In einer zweiten Phase wird diese Leitung entladen, falls beide Eingangstellen gleich '0' sind und mit dem eingehenden Übertrag c_i verbunden, falls genau eine Eingangsstelle gleich '1' ist. Motiviert wird diese Schaltung durch die unsymmetrischen Anstiegs- und Abfallzeiten der NMOS-Technologie. Aufgrund der Verwendung von Depletion-Transistoren dauert es wesentlich länger, eine Leitung auf '1' zu setzen, als sie auf '0' zu entladen. Die hohe Anstiegszeit kann durch das Precharging zu einer Zeit, zu der K_i und P_i ohnehin vielleicht noch nicht bekannt sind, eingespart werden. Precharging wird mit Vorliebe in Fällen eingesetzt, in

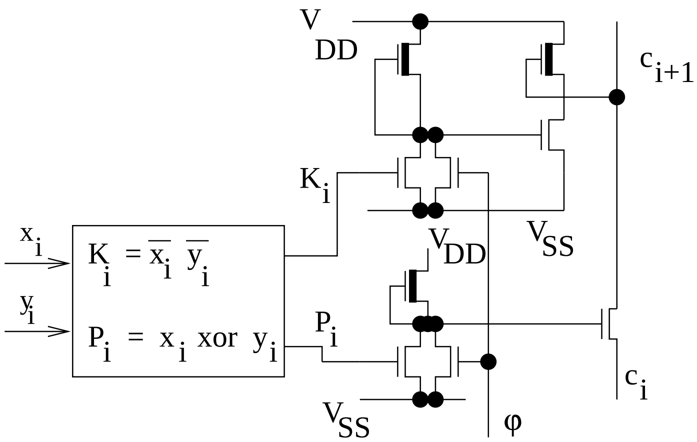


Abbildung 2.49: Addiererstufe mit Precharging

denen es derartige zeitliche Asymmetrien gibt, insbesondere wenn die aufzuladenden Leitungen eine große Kapazität besitzen (beispielsweise bei Bussen).

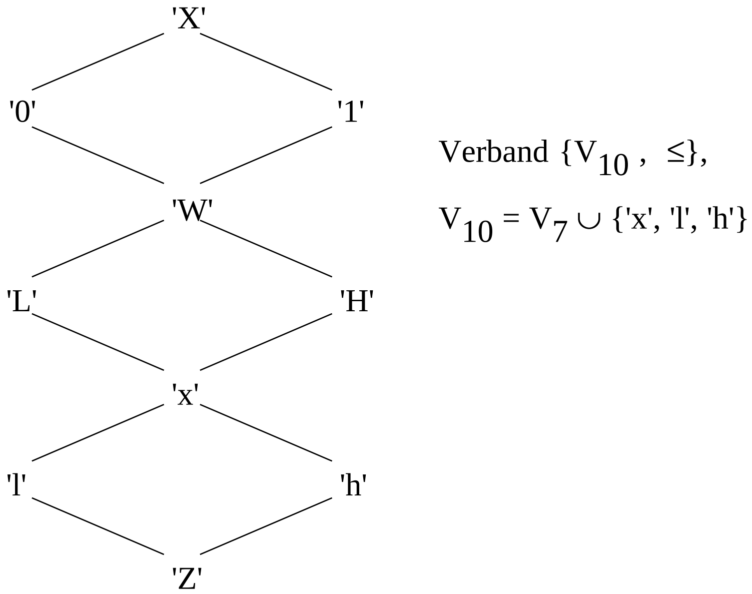


Abbildung 2.50: Verband $\{V_{10}, \leq\}$

Um Schaltungen zu modellieren, welche diesen Effekt ausnutzen, werden weitere, noch schwächere Signale benötigt. Solche noch schwächeren Signale werden z.B. im CAP-Simulator benutzt (siehe [LR83]). Diese Signalwerte werden hier als `'l'` und `'h'` bezeichnet. Bei einer der Abbildung 2.50 entsprechenden Ordnung entspricht die Wirkung einer Verbindung wieder der Supremumbildung, d.h. der Bestimmung der größten Signalstärke.

Sinnvollerweise ändern Simulatoren besonders schwache Signalwerte nach einer bestimmten Zeit auf 'Z' ab.

2.3.6.5 Modellierung von Unsicherheit

Das bislang benutzte 'X' besitzt im allgemeinen die Bedeutung einer unbekannten Spannung. Zur expliziten Darstellung der jeweils möglichen Werte reicht es allein nicht aus. Dies wird klar, wenn man wie in Abb. 2.51 'X'–Werte an den Gates betrachtet.

In diesem Fall ist unsicher, ob eine schwache Null oder eine starke Eins zuzuweisen ist. Um Unsicherheit darzustellen, muß daher im allgemeinen mit **Mengen von Signalwerten** gearbeitet werden. Eine systematische Methode zu einer derartigen Erweiterung beschreibt Hayes in einem zweiten Artikel [Hay86].

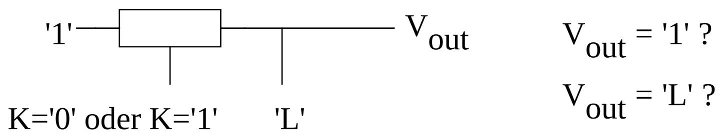


Abbildung 2.51: Zum Problem der undefinierten Gates

Ausgangspunkt der Theorie bildet zunächst die Menge möglicher Basiswerte, im Folgenden A genannt. Unter den **Basiswerten** kann man sich logisch differenzierte Pegelbereiche der elektrischen Signale vorstellen. Im Falle von reinen Gatternetzen benötigt man lediglich zwei Werte: '0' und '1'. Für Open-Collector–Schaltungen werden drei Werte benutzt: '0', '1' und 'Z'. Im Switch–Level–Bereich kommt noch 'X' hinzu:

$A = B_2$	$= \{'0', '1'\}$	für Gatterschaltungen,
C_3	$= \{'0','1','Z'\}$	für Open–Collector-Schaltungen,
T_3	$= \{'0','1','Z','X'\}$	für Switch–Level-Schaltungen,

Tabelle 2.5: Wertemengen für verschiedene Simulationen

Um Unsicherheit zu berücksichtigen, muß, wie bereits oben angedeutet, zu beliebigen Mengen von Basiswerten, also zu **Potenzmengen**[10] **von Basiswerten** übergegangen werden. Für die 2–wertigen Basiswerte kommt man so zu:

$$\wp(A) = \{\{'0'\}, \{'1'\}, \{'0', '1'\} = '?'\}$$

Eine derartige Erweiterung für die Switch–Level–Simulation mit den 7 Signalwerten aus $(V_{10} - \{'X','W','x'\})$ liefert 128 Werte. In dieser Form ist dies auch im CAP-Simulator implementiert [LR83].

2.3.6.6 46-wertige Simulation nach Coelho (5 Signalstärken)

Die Darstellung aller möglichen Teilmengen von Signalwerten erfordert einen hohen Speicherplatz-Bedarf. Gelegentlich geht man daher dazu über, mögliche **Wertebereiche** der oben angegebenen Verbandsdiagramme zu kodieren. Für VHDL beschreibt Coelho ein solches Modell in seinem Standardwerk [Coe89] über VHDL.

Coelho benutzt als Werte, die noch stärker sind als '0' und '1', die Werte F0 und F1 (forced-0 bzw. forced-1). Diese entprechen der Stärke der Stromversorgung. Insgesamt modelliert er damit Unsicherheiten bezüglich der Werte. Statt aller Kombinationen dieser Werte betrachtet Coelho die $n = 9$ Werte als in der Reihenfolge F0, '0', 'L', 'l', 'Z', 'h', 'H', '1', 'F1' geordnet und arbeitet mit allen Intervallen von diesen Werten, nämlich mit

[F0..F0],	[F0..'0'],	[F0..'L'],	...	[F0..F1],
-	['0'..'0'],	['0'..'L'],	...	['0'..F1]
-	-	['L'..'L'],	...	['L'..F1]
...	...	...	...	...
-	-	-	-	[F1..F1]

Es ergeben sich 9+8+7+6+5+4+3+2+1=$n * (n + 1)/2 = 45$ Intervalle. Zusätzlich benutzt er noch einen gesonderten Wert 'U' für "im Simulator uninitialisiert" und kommt so auf 46 mögliche Werte. Das Buch enthält eine ausführliche Beschreibung der benötigten Typ- und Funktionsdefinitionen.

2.3.6.7 Modellierung von Signalwechseln

In manchen Fällen möchte man steigende und fallende Flanken, kurzzeitige Pegeleinbrüche u.s.w. als spezielle Signalwerte erkennen. Dieses ist möglich, indem man von den bislang möglichen Signalwerten zu Sequenzen von solchen Werten übergeht. Man spricht in diesem Fall von der **Produktbildung der Basiswerte**. Formal notiert man die Produktbildung wie folgt:

[10] Potenzmenge einer Menge = Menge aller Teilmengen; die leere Menge ist hier bedeutungslos.

$$P = A \times \ldots\ldots \times A \quad \text{bzw.} \quad P = \wp(A) \times \ldots \wp(A).$$

Ziele der Produktbildung sind z.B. die Darstellung einer zeitlichen Abfolge von Signalwerten[11].

Beispiel:
"rising" = ('0','1')

Die Möglichkeit zur Darstellung von zeitlichen Abfolgen soll nunmehr genutzt werden, um Hazards zu repräsentieren.

Def.: Ein static–1– (0–) Hazard ist ein '1'– ('0'–) Impuls auf einem Signal, welches aufgrund der logischen Funktion ständig '0' ('1') sein sollte.

Die Simulationssequenz eines solchen Hazards enthält eine Folge ('0','1','0') bzw. ('1','0','1'). Die möglichen Tripel von Signalwerten besitzen dabei die Bedeutung:

'N' = ('0','0','0')		; 'E' = ('1','1','1')	
'r' = ('0','X','1')	(rising)	; 'f' = ('1','X','0')	(falling)
'h' = ('0','X','0')	(1-Hazard)	; 'H' = ('1','X','1')	(0-Hazard)

Mit dem vollkommen undefinierten Wert ('U','U','U') kommt man zu 7 Simulationswerten. Diese Werte sind gegen AND, OR und NOT nicht abgeschlossen:

('U','U','U') AND ('0','U','1') = ('0','U','U') (Übergang '0' $\rightarrow$ unsicher)

Für einen Abschluß werden 4 weitere Simulationswerte gebraucht, insgesamt also 11:

('0','U','U')	(Übergang	'0' $\rightarrow$ unsicher)
('1','U','U')	(Übergang	'1' $\rightarrow$ unsicher)
('U"U','0')	(Übergang	unsicher $\rightarrow$ '0')
('U"U','1')	(Übergang	unsicher $\rightarrow$ '1')

Als dynamische Hazards bezeichnet man Übergänge '0'$\rightarrow$'1' bzw. '1'$\rightarrow$'0' mit "unsauberer" Flanke. Zur Darstellung benötigt man 4–Tupel wie z.B. ('0','1','0','1'). Insgesamt kann man dann zwischen 13 Simulationswerten unterscheiden [Hay86].

2.3.6.8 9-wertige Logik nach IEEE-Standard 1164

VHDL basiert nicht auf einer festen Anzahl von Signalwerten, sondern erlaubt es, Signalwerte in der Sprache selbst zu definieren. Um nun eine gewisse Austauschbarkeit von VHDL-Modellen zu erreichen, basieren viele Modelle auf einer 9-wertigen Logik

TYPE std_ulogic IS ('U','X','0','1','Z','W','L','H','-')

die im IEEE-Standard 1164 festgelegt wurde.

Die Bedeutung dieser Werte ist nach der vorangehenden Erläuterung von mehrwertigen Logiksystemen leicht zu erklären: IEEE 1164 basiert auf den 7 Werten der o.a. 7-wertigen Logik. 'U' steht gemäß diesem Standard für 'im Simulator uninitialisiert'.

'-' steht für ein sog. 'input don't care', welches der Erläuterung bedarf.

Mit einer Hardwarebeschreibungssprache wie VHDL möchte man vielfach auch Boolesche Funktionen spezifizieren. Eine relativ bequeme Form dafür bietet das VHDL-SELECT-Statement, welches eine Erweiterung üblicher CASE-Statements ist.

Beispiel: Darzustellen sei die Boolesche Funktion

$f(a,b,c) = a\bar{b} + bc$, wobei deren Wert für $a = b = c =$ `'0'` undefiniert sei.

Eine bequeme Art der Notation in VHDL wäre die Schreibweise als

```
f <= SELECT a & b & c
```

[11] Die Produktbildung kann auch der Darstellung von fehlerfreiem und fehlerhaftem Fall dienen

```
'1' WHEN "10-"    -- beabsichtigt: f = '1', falls a & b  = "10"
'1' WHEN "-11"
'X' WHEN "000"
```

Dabei soll '-' bedeuten, dass die entsprechende Variable des selektierenden Bitvektors `a & b & c` redundant ist. Man nennt '-' auch *input don't care*. Dagegen bedeutet `'X'`, dass die Ausgabe unbekannt bleiben kann, also redundant ist. Man nennt es daher auch *output don't care*.

Leider spezifiziert obiges SELECT-Statement etwas ganz anderes. Dazu muss man wissen, dass VHDL ursprünglich für die Simulation konzipiert wurde. In einer Simulation kommen Werte wie `'U'` und `'X'` während der Simulation vor, wenn die entsprechenden Bits undefiniert sind. Mit einer Abfrage `IF signal = 'X'` möchte man auf solche Fälle reagieren können und beispielsweise eine Fehlermeldung erzeugen. In diesem Zusammenhang muss die Operation '=' stets prüfen, ob `signal` den Wert `'X'` besitzt. Da VHDL eine um Signalwerte erweiterbare Sprache ist, kann die Operation = auch für andere Signalwerte (wie z.B. `'-'`) nur das Abprüfen auf diesen Wert bedeuten[12].

Da der implizite Test auf Gleichheit des SELECT-Statements nicht umdefiniert werden kann, benutzen Synthesesysteme eine Funktion `std_match`, welche `'-'` tatsächlich als *input don't care* behandelt. Leider ist diese Funktion aber nur in IF-Statements sinnvoll einzusetzen.

Als Schlussfolgerung kann man festhalten, dass VHDL aufgrund der Erweiterbarkeit der Wertemenge Abstriche beim Beschreibungskomfort hinnehmen muss.

Im Folgenden sei ein Auszug aus dem Quelltext des IEEE 1164-Standards wiedergegeben:

```
-- --------------------------------------------------------------------
--
--   Title     :  std_logic_1164 multi-value logic system
--   Library   :  This package shall be compiled into a library
--             :  symbolically named IEEE.
--             :
--   Developers:  IEEE model standards group (par 1164)
--   Purpose   :  This packages defines a standard for designers
--             :  to use in describing the interconnection data types
--             :  used in vhdl modeling.
--             :
--   Limitation:  The logic system defined in this package may
--             :  be insufficient for modeling switched transistors,
--             :  since such a requirement is out of the scope of this
--             :  effort. Furthermore, mathematics, primitives,
--             :  timing standards, etc. are considered orthogonal
--             :  issues as it relates to this package and are therefore
--             :  beyond the scope of this effort.
--             :
--   Note      :  ..
PACKAGE std_logic_1164 IS
-------------------------------------------------------------------
-- logic state system  (unresolved)
-------------------------------------------------------------------
TYPE std_ulogic IS ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
-------------------------------------------------------------------
-- unconstrained array of std_ulogic for use with the resolution function
-------------------------------------------------------------------
TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;
```

[12] Da `'-'` als *input don't care* nie einem Signal oder einer Variablen zugewiesen wird, ist obiges SELECT-Statement sinnlos. Man vereinbart, dass ein Simulator vor der Simulation `'-'` durch `'X'` ersetzt.

```
-------------------------------------------------------------------
-- resolution function
-------------------------------------------------------------------
FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;

-------------------------------------------------------------------
-- *** industry standard logic type ***
-------------------------------------------------------------------
SUBTYPE std_logic IS resolved std_ulogic;

-------------------------------------------------------------------
-- unconstrained array of std_logic for use in declaring signal arrays
-------------------------------------------------------------------
TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;

-- common subtypes
-------------------------------------------------------------------
SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1'; -- ('X','0','1')
...
-------------------------------------------------------------------
-- overloaded logical operators
-------------------------------------------------------------------

FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
...

-------------------------------------------------------------------
-- vectorized overloaded logical operators
-------------------------------------------------------------------
FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
...
-------------------------------------------------------------------
-- conversion functions
-------------------------------------------------------------------
FUNCTION To_bit       ( s : std_ulogic;        xmap : BIT := '0') RETURN BIT;
FUNCTION To_bitvector ( s : std_logic_vector ; xmap : BIT := '0') RETURN BIT_VECTOR;
FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT := '0') RETURN BIT_VECTOR;

FUNCTION To_StdULogic       ( b : BIT               ) RETURN std_ulogic;
...
-------------------------------------------------------------------
-- edge detection
-------------------------------------------------------------------
FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;
END std_logic_1164;

PACKAGE BODY std_logic_1164 IS
-------------------------------------------------------------------
-- local types
-------------------------------------------------------------------
TYPE stdlogic_1d IS ARRAY (std_ulogic) OF std_ulogic;
TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;

-------------------------------------------------------------------
-- resolution function
-------------------------------------------------------------------
CONSTANT resolution_table : stdlogic_table := (
--      ---------------------------------------------------------
--      | U    X    0    1    Z    W    L    H    -        |   |
--      ---------------------------------------------------------
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
```

```
FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS

-- tables for logical operations
-- truth table for "and" function
CONSTANT and_table : stdlogic_table := (
--      ----------------------------------------------------
--      | U    X    0    1    Z    W    L    H    -         |   |
--      ----------------------------------------------------
        ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | X |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | 0 |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 1 |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | Z |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | W |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | L |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | H |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - |
);

-------------------------------------------------------------------
-- overloaded logical operators ( with optimizing hints )
-------------------------------------------------------------------

FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
BEGIN
    RETURN (and_table(l, r));
END "and";

-------------------------------------------------------------------
-- conversion functions
-------------------------------------------------------------------
FUNCTION To_bit       ( s : std_ulogic;
          xmap : BIT := '0') RETURN BIT IS
BEGIN
        CASE s IS
            WHEN '0' | 'L' => RETURN ('0');
            WHEN '1' | 'H' => RETURN ('1');
            WHEN OTHERS => RETURN xmap;
        END CASE;
END;
--------------------------------------------------------------------
FUNCTION To_bitvector ( s : std_logic_vector ;
                        xmap : BIT := '0') RETURN BIT_VECTOR IS
    ALIAS sv : std_logic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
    VARIABLE result : BIT_VECTOR ( s'LENGTH-1 DOWNTO 0 );
BEGIN
    FOR i IN result'RANGE LOOP
        CASE sv(i) IS
            WHEN '0' | 'L' => result(i) := '0';
            WHEN '1' | 'H' => result(i) := '1';
            WHEN OTHERS => result(i) := xmap;
        END CASE;
    END LOOP;
    RETURN result;
END;
--------------------------------------------------------------------
FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT := '0') RETURN BIT_VECTOR IS
    ALIAS sv : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
    VARIABLE result : BIT_VECTOR ( s'LENGTH-1 DOWNTO 0 );
BEGIN
    FOR i IN result'RANGE LOOP
        CASE sv(i) IS
            WHEN '0' | 'L' => result(i) := '0';
            WHEN '1' | 'H' => result(i) := '1';
```

```
            WHEN OTHERS => result(i) := xmap;
        END CASE;
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------------------
-- edge detection
-------------------------------------------------------------------
FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN IS
BEGIN
    RETURN (s'EVENT AND (To_X01(s) = '1') AND
                        (To_X01(s'LAST_VALUE) = '0'));
END;
END std_logic_1164;
```

2.4 SpecCharts und SpecC

2.4.1 Aufbau von SpecCharts

SpecCharts [GVNG94] ist eine StateChart-Variante, die sich stärker an VHDL anlehnt, um auch programmiersprachliche Konstrukte zur Verfügung zu haben. SpecCharts ist für zwei verschiedene Ausprägungen der Sprache konzipiert: für eine graphische und für eine textuelle Ausprägung[13].

Die wesentlichen Eigenschaften von SpecCharts beschreibt die Abb. 2.52.

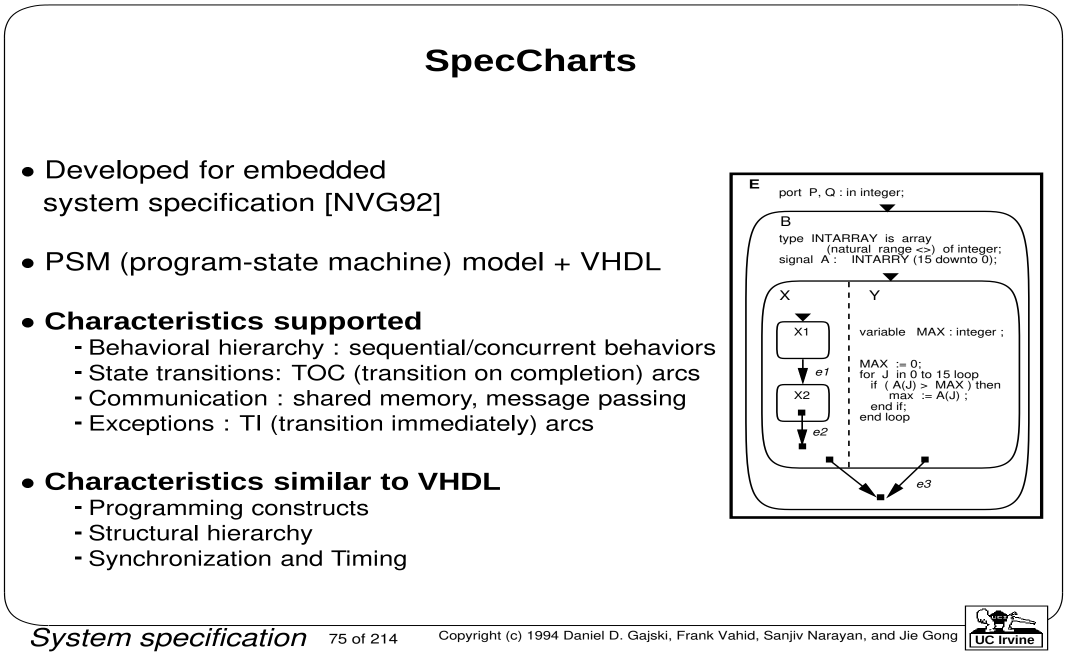


Abbildung 2.52: Wesentliche Eigenschaften von SpecCharts

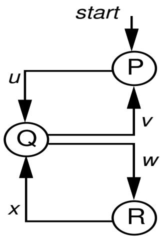
Die textuelle Beschreibung selbst kann relativ einfach in VHDL übersetzt werden.

SpecCharts kennt zwei Arten von Kanten in Zustandsgraphen: *transition on completion* und *transition immediately*. Erstere wird erst ausgeführt, wenn das verhalten abgearbeitet wurde (siehe Abb. 2.53). Letztere wird sofort ausgeführt und kann zum Beispiel zur Modellierung von Unterbrechungen genutzt werden (siehe Abb. 2.54). Verhalten kann auch hierarchisch beschrieben werden (siehe z.B. Abb. 2.55). Für weitere Erklärungen sei auf das Buch von Gajski verwiesen.

[13]Die graphische Ausprägung wird z.Z. offenbar noch nicht durch CAD-Tools unterstützt, sondern dient nur der übersichtlichen Beschreibung.

SpecCharts : state transitions

- State transitions represented by TOC and TI arcs between behaviors




```
behavior MAIN type sequential subbehaviors is
begin
    P  :  (TOC, u, Q) ;
    Q  :  (TOC, v, P), (TOC, w, R);
    R  :  (TOC, x, Q);

    behavior  P .....
    behavior  Q .....
    behavior  R .....

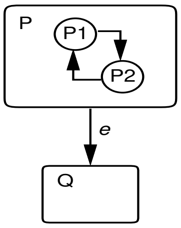
end MAIN;
```

System specification 76 of 214 UC Irvine

Abbildung 2.53: Übergänge bei Beendigung eines Verhaltens (transition on completion=TOC)

SpecCharts : exceptions

- Exceptions represented by TI (transition immediately) arcs




```
behavior MAIN  type sequential subbehaviors is
begin
    P  :  (TI, e, Q);
    Q  :  ;

    behavior  P
          behavior P1
               .......
          behavior P2
               .......

    behavior  Q
          ......

end MAIN;
```

System specification 78 of 214 UC Irvine

Abbildung 2.54: Beschreibung von Ausnahmen mittels transition immediately (TI)-Kanten

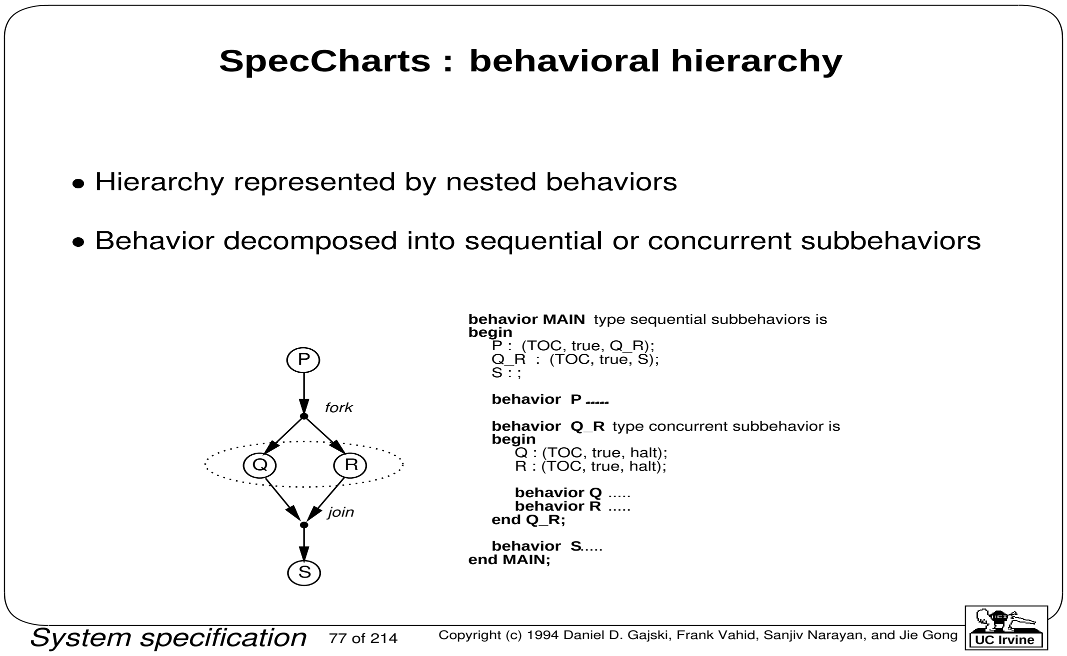


Abbildung 2.55: Verhaltenshierarchie

2.4.2 Beispiel: Anrufbeantworter

7. Vorles.

Als Beispiel der Beschreibung eines Systems in SpecCharts betrachten wir einen Anrufbeantworter. Ziel dieser Betrachtung ist es, die Vorteile von SpecCharts zu beleuchten (siehe Abb. 2.56).

Der Anrufbeantworter verfügt über eine eine Bandeinheit, einen Leitungseingang, eine Anzeige, diverse Tasten und eine Steuerheit für alle diese Komponenten (siehe Abb. 2.57).

Die folgenden Abbildungen beschreiben die Funktion dieser Steuereinheit in hierarchischer Weise.

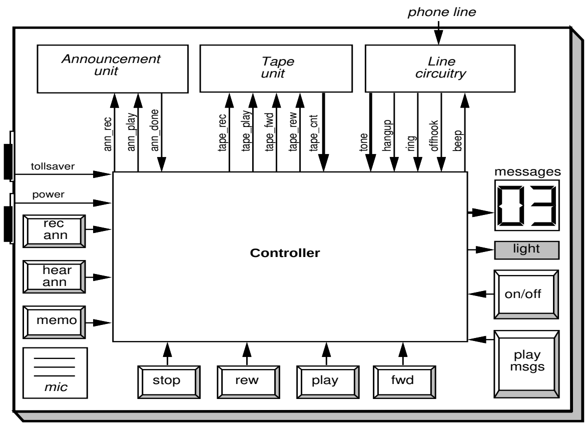
Outline

- Capture an example's model in a particular language
  - PSM model in the SpecCharts language
- Point out the benefits of a good language/model match
- Highlight experiments that demonstrate those benefits

Specification example 81 of 214 UC Irvine

Abbildung 2.56: Übersicht über die Abbildungen des Beispiels

Answering machine controller's environment



Specification example 82 of 214 UC Irvine

Abbildung 2.57: Globale Sicht des Anrufbeantworters

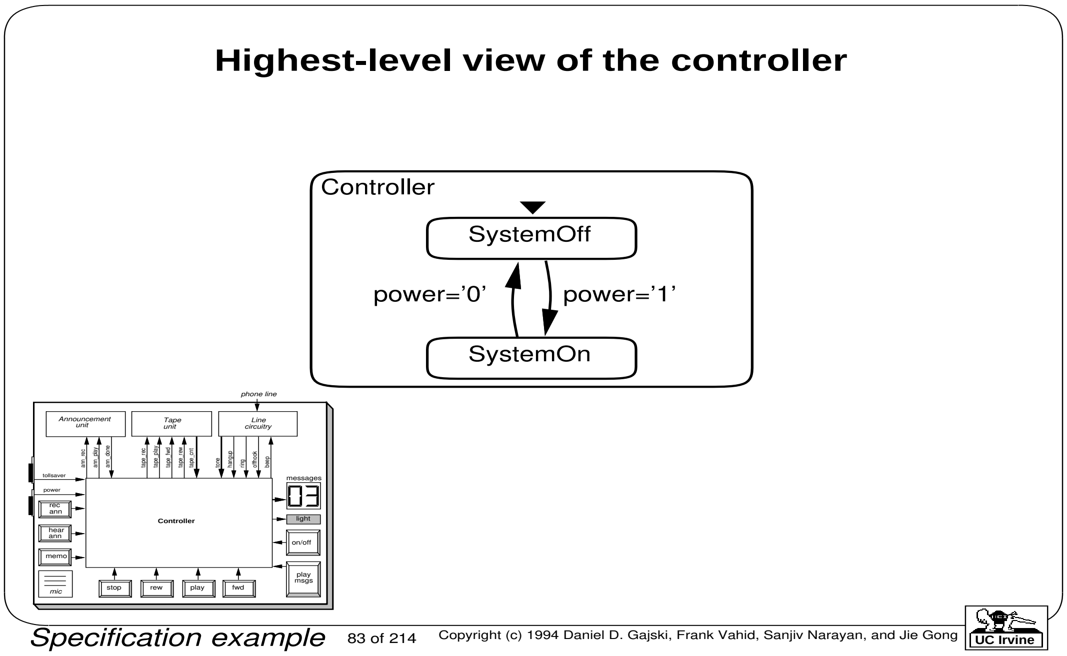


Abbildung 2.58: Oberste Hierarchiebene des Controllers

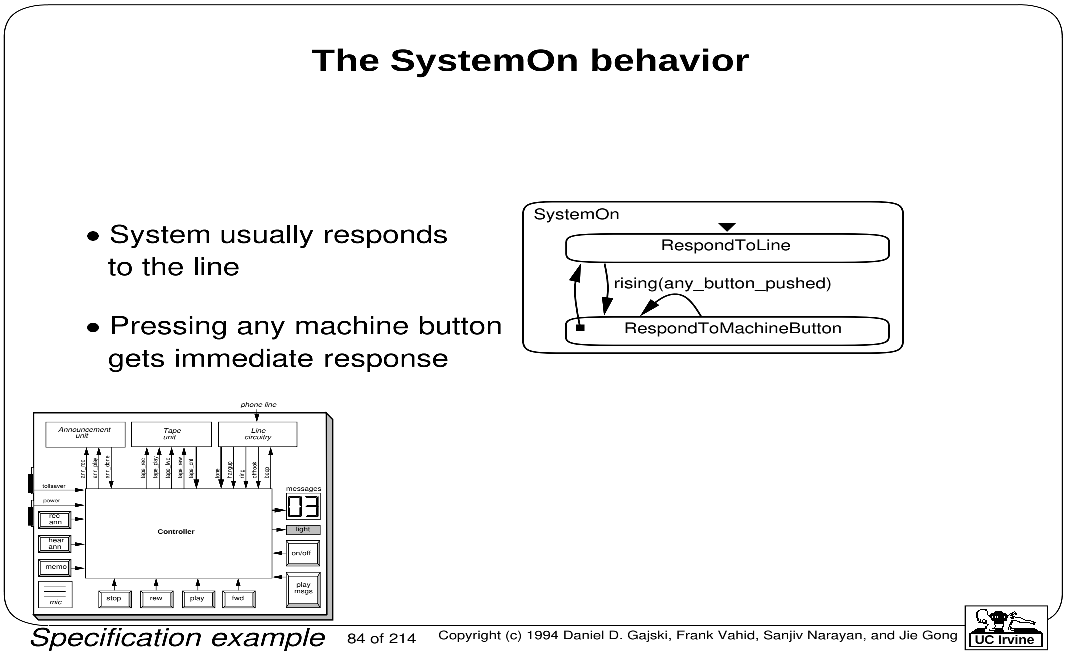


Abbildung 2.59: Einschaltvorgang

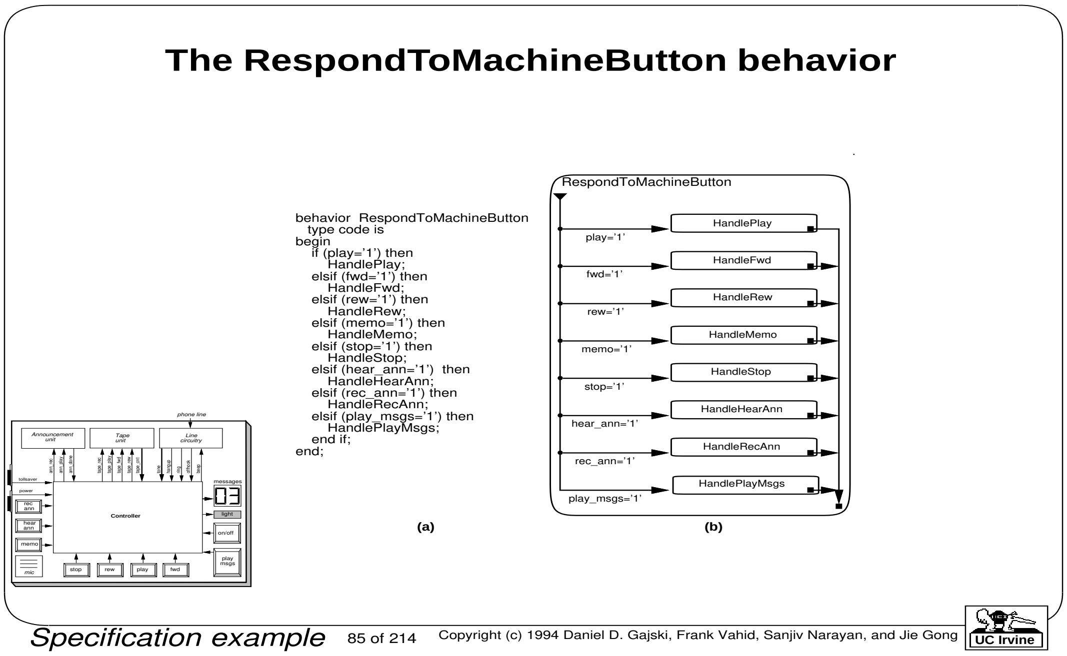


Abbildung 2.60: Reaktion auf die Tasten

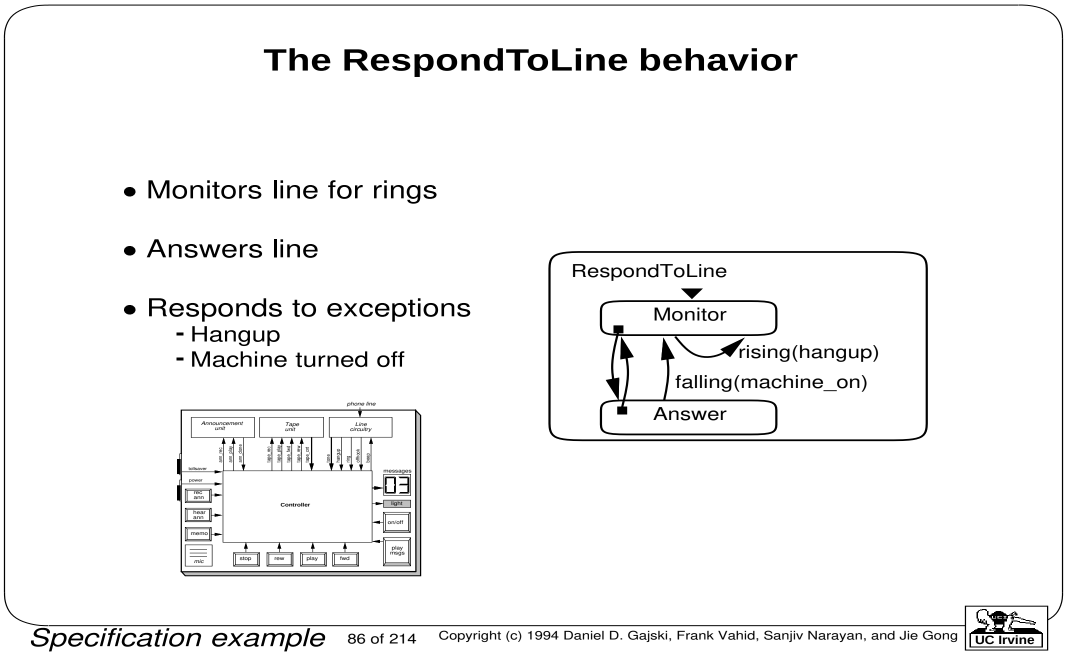


Abbildung 2.61: Reaktion auf die externe Leitung

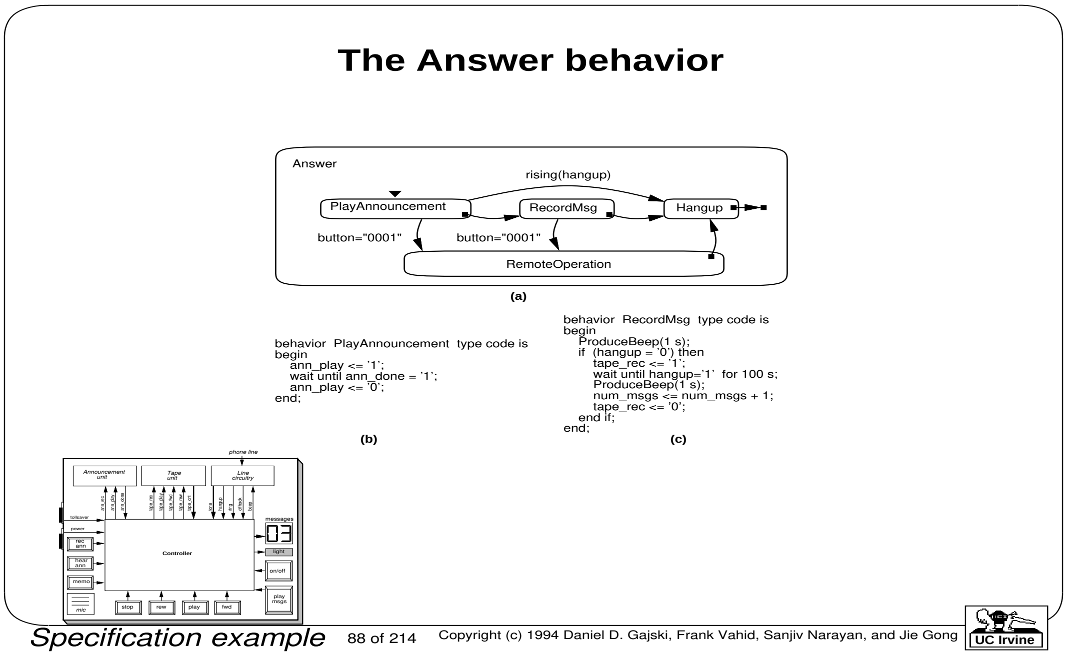


Abbildung 2.62: Antwortverhalten der Maschine

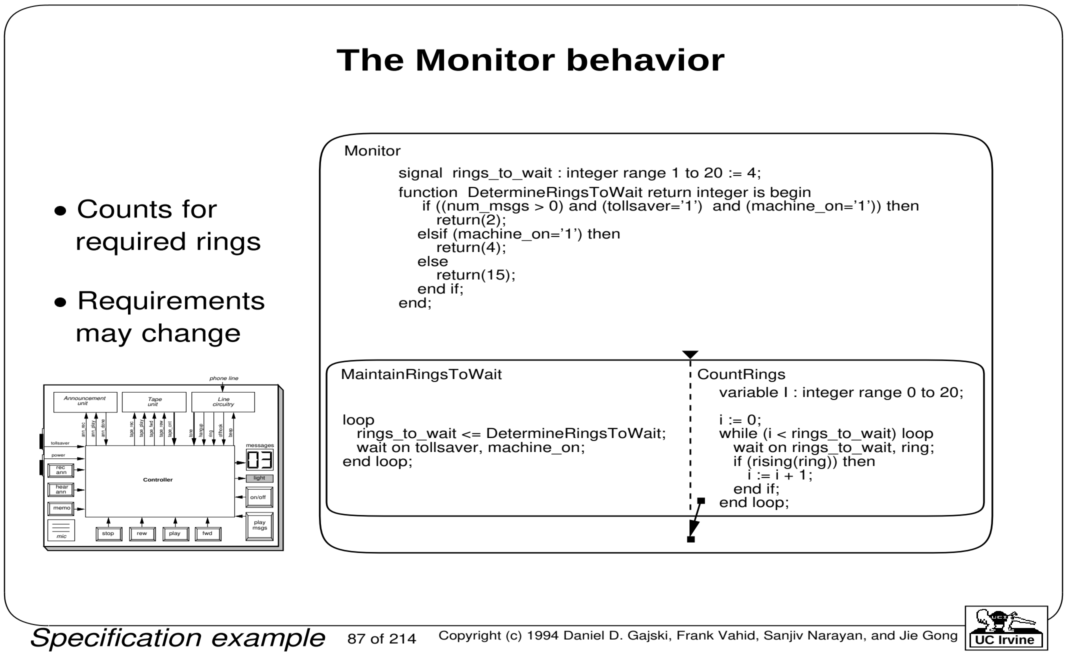


Abbildung 2.63: Überwachungsverhalten der Maschine

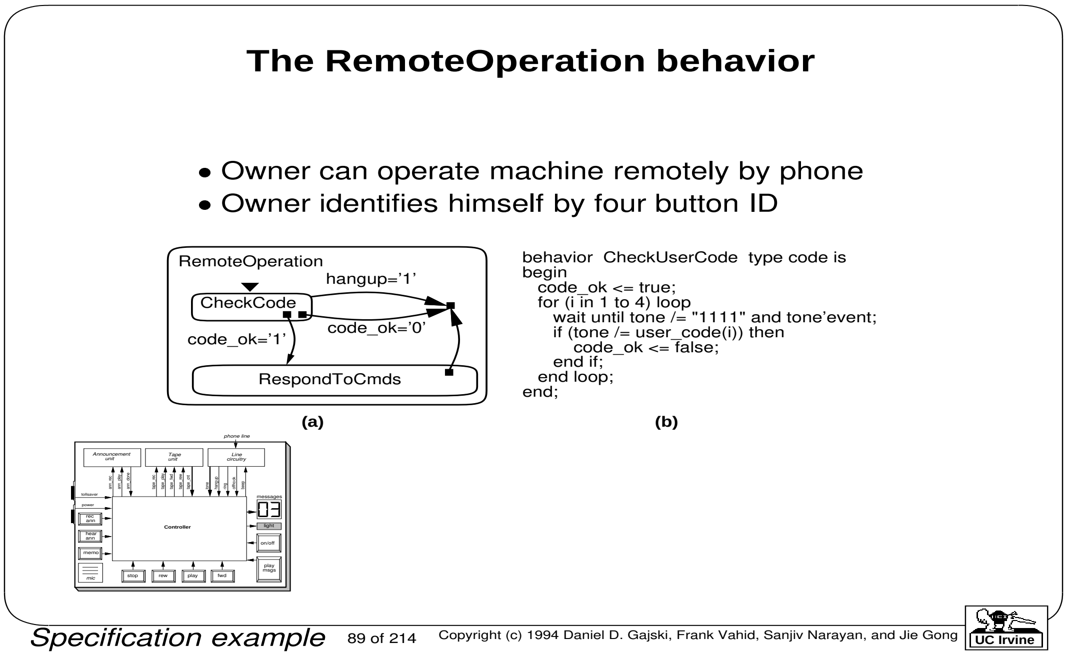


Abbildung 2.64: Fernabfrage

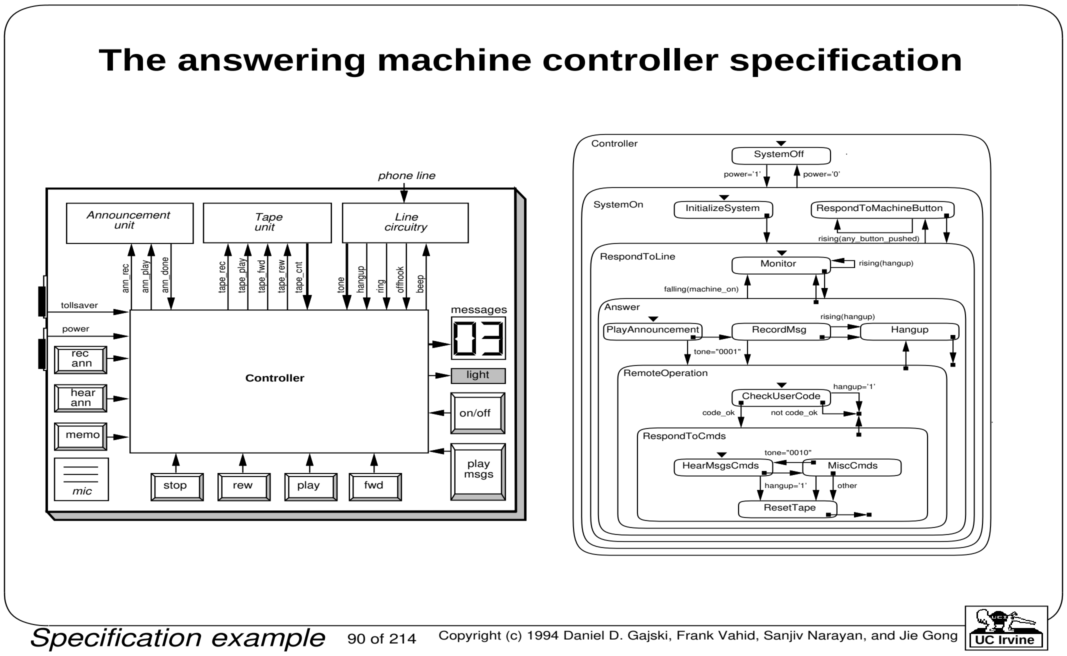


Abbildung 2.65: Spezifikation des Controllers

Executable specification use

- Precision
  - Readability/precision compete in a natural language
  - Executable specification encourages precision
  - Designer asks questions, specification answers them
- Language/model match (SpecCharts/PSM):
  - Hierarchy
  - State-transitions
  - Programming constructs
  - Concurrency
  - Exceptions
  - Completion
  - Equivalence of states and programs

Specification example 91 of 214 UC Irvine

Abbildung 2.66: Vorteile von SpecCharts

Specification capture experiment

	VHDL	SpecCharts
Average specification–time in minutes	40	16
Number of modelers	3	3
Number of incorrect specifications first time	2	0
Number of incorrect specifications second time	1	0

- VHDL modelers required 2.5 times longer
- Two VHDL specifications possessed control errors
- SpecCharts were effective for state-transitions and exceptions

Specification example 92 of 214 UC Irvine

Abbildung 2.67: Ergebnisse des Experiments

2.4.3 Bewertung

Comparison of SpecCharts, VHDL and Statecharts

Answering machine example

		Conceptual model	SpecCharts	VHDL (hierarch.)	VHDL (flat)	Statecharts
Specification attributes	Program–states	42	42	42	32	80
	Arcs	40	40	40	152	135
	Control signals	––	0	84	1	0
	Lines/leaf	––	7	27	29	––
	Lines	––	446	1592	963	––
	Words	––	1733	6740	8088	––
Shortcomings	No sequential program constructs					X
	No hierarchy			X	X	
	No exception constructs			X	X	
	No hierarchical events				X	
	No state–transition constructs			X	X	

Specification example 93 of 214 UC Irvine

Abbildung 2.68: Vergleich mit StateCharts und VHDL

Design quality experiment

Design attribute	Designed from English	Designed from SpecCharts
Control transistors	3130	2630
Datapath transistors	2277	2251
Total transistors	5407	4881
Total pins	38	38

- No loss in design quality with an executable language

Specification example 94 of 214 UC Irvine

Abbildung 2.69: Qualität des Entwurfs

2.4.4 SpecC

SpecCharts erfüllt ganz offensichtlich noch nicht alle Erwartungen an eine Systembeschreibungssprache. SpecC ist eine neuere Entwicklung[14] SpecC besitzt die folgenden wesentlichen Merkmale []:

- Die Unterscheidung zwischen zwei Sorten von Transitionen wird beibehalten.
- SpecC basiert auf C statt auf VHDL. Ein Grund dafür ist die wesentlich stärkere Verbreitung von C in der Industrie. Viele Standards, z.B. für die Bilddatenkompression oder die Kryptographie liegen in C vor. Ihre Umsetzung auf VHDL erzeugt einen erheblichen Aufwand.
- Kanäle usw.

[14] An dieser Entwicklung wirkte mit Rainer Dömer ein Dortmunder Doktorand im Rahmen seiner Promotion im sonnigen Kalifornien mit.

2.5 SDL

2.5.1 Sprachelemente

SDL [Hog89] ist eine Sprache zur Spezifikation von verteilten Systemen, v.a. in der Telekommunikation. Die Entwicklung erfolgte seit Anfang der 70'er Jahre. Erst ab Ende der 80'er Jahre wurde für SDL eine formale Semantik entwickelt. Die Normierung von SDL erfolgte im CCITT (Committee Consultativ International Telegraphique et Telephonique). Für SDL gibt es zwei syntaktische Formen: das graphische Format SDL/GR und das Text-Format SDL/PR.

Das Grundelement von SDL sind Prozesse. Ein Proze stellt jeweils einen erweiterten endlichen Automaten dar. Die Erweiterung besteht dabei aus einer verbesserten Beschreibung von Operationen auf Daten.

Als Beispiel betrachten wir das Zustandsdiagramm der Abbildung 2.70. Es handelt sich um die Beschreibung eines einfachen Automaten zum Aufbau von Verbindungen.

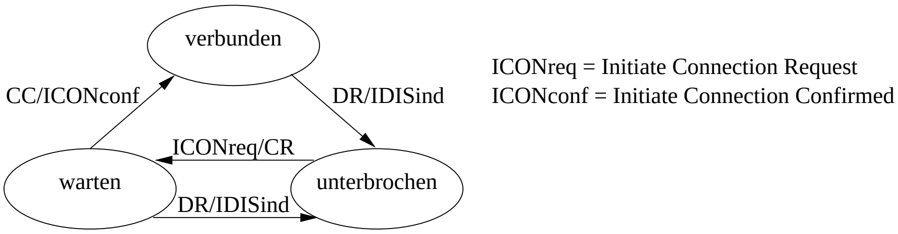


Abbildung 2.70: Zustandsdiagramm

Für dieses Zustandsdiagramm zeigt die Abbildung 2.71 die entsprechende Beschreibung in SDL/GR.

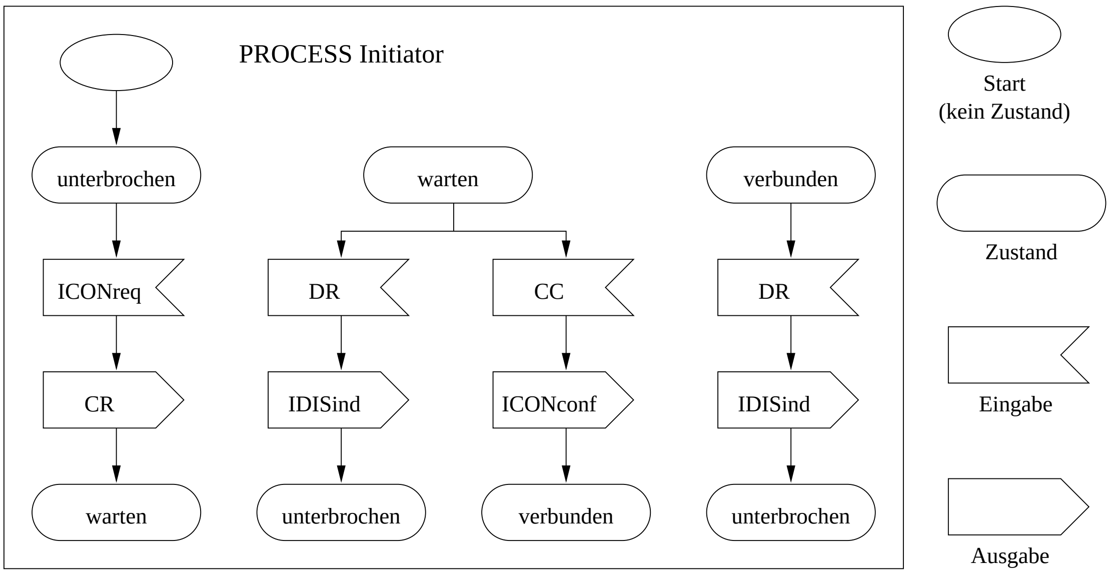


Abbildung 2.71: SDL Beschreibung des Prozesses 'Initiator'

In diesem Beispiel sind rechts die verschiedenen graphischen Symbole für Zustände, Eingaben, Ausgaben und die Identifizierung des initialen Zustands eingezeichnet. Die direkte Zuordnung zu den Kanten des Zustandsdiagramms ist klar zu sehen.

Die **Semantik der Prozesskommunikation** basiert auf einer impliziten Warteschlange (siehe Abb. 2.72).

In jedem gegebenen Zustand wird das erste Signal aus der Warteschlange entfernt und geprüft, ob es einen Zustandsübergang bewirken kann. In jedem Fall wird es nicht weiter gespeichert (Ausnahme: SAVE-Mechanismus). Von Konzept her besitzen die Warteschlangen eine beliebig große Kapazität. Bei der Abbildung auf reale Systeme taucht die Frage auf, wie groß die Warteschlangen sein müssen, damit sie nie überlaufen. Ein weiteres Problem ist die Abbildung auf eine Implementierung, die nicht durch unnötig große Warteschlangen Effizienz verliert. Dies trifft v.a. bei der Abbildung auf Hardware zu, in der ggf. ausreichend viele Register vorhanden sein müssen.

Aufgrund der geforderten Realzeitfähigkeit erlaubt SDL auch die Benutzung von **Timern**. Timer können zusammen mit Prozessen deklariert werden (siehe Abb. 2.73).

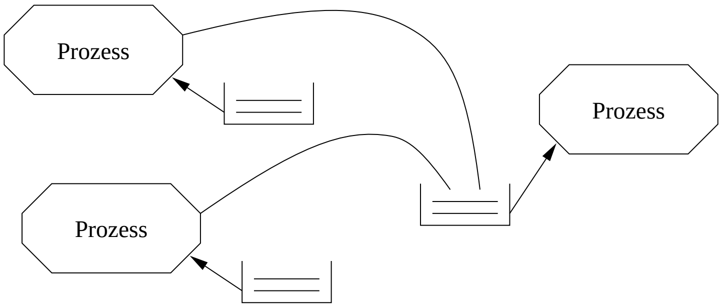


Abbildung 2.72: Implizite Warteschlangen

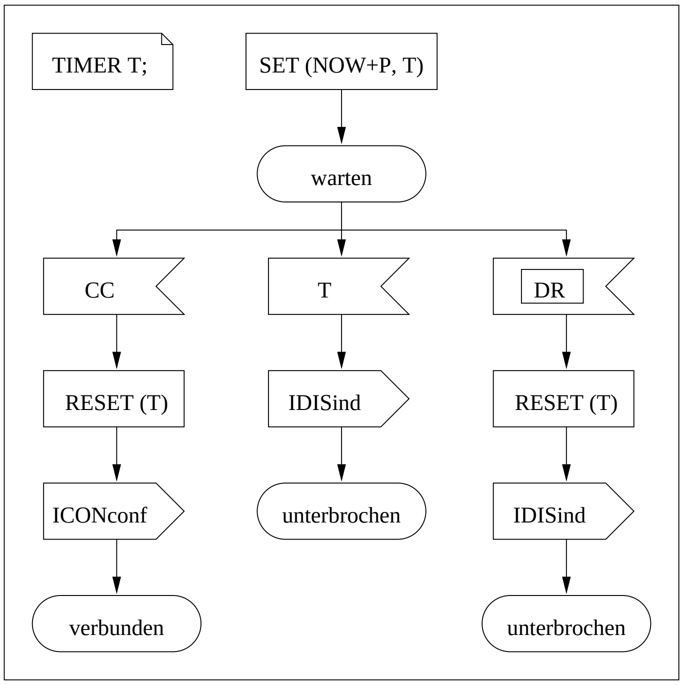


Abbildung 2.73: Deklaration eines Timers T in einem Prozess

Mit dem Befehl `SET(NOW+P,T)` kann der Timer auf die Zeit 'jetzt + P' gesetzt werden. Es ist eine Besonderheit von SDL, dass mit dem Ablauf der gesetzten Zeit nicht unmittelbar eine Aktion des Prozesses verbunden werden kann. Vielmehr besitzt jeder Prozess eine FIFO-Schlange noch nicht bearbeiteter Eingabeereignisse. In diese Schlange wird nach Ablauf der spezifizierten Zeit ein Eingabeereignis `T` eingereiht. Wenn dieses Ereignis das nächste zu bearbeitende Ereignis ist, wird der Übergang in der Mitte der Abb. 2.73 ausgelöst. Die beiden übrigen Übergänge löschen den Timer mittels `RESET(T)`. Wenn der Timer mit `RESET` zurückgesetzt wird, solange er sich noch in der Warteschlange befindet, wird er wieder aus der Schlange entfernt. Der beschriebene FIFO-Mechanismus reicht für die meisten Anwendungen in der Telekommunikation aus, bereitet aber eventuell Probleme beim Einsatz von SDL bei 'ganz harten' Zeitbedingungen.

SDL bietet auch eine Unterstützung bei der **Verwendung von Daten**. Innerhalb von Prozessen können Variable deklariert und genutzt werden (siehe Abb. 2.74). Sie können auch bei Ein- und Ausgaben übergeben werden (siehe Abb. 2.74 rechts).

Das Datenkonzept von SDL basiert auf abstrakten Datentypen (ADTs). In Textflächen von Prozessen können Operationen auf Daten wie in üblichen Programmiersprachen notiert werden.

Zur Erleichterung des Verständnisses im Falle einer großen Zahl von Prozessen kann deren Interaktion mit sog. Blöcken beschrieben werden. Diese Blöcke beschreiben die Kommunikationsstruktur. Kanten können mit Kanalnamen sowie (in eckigen Klammern) Signalnamen beschriftet sein. Derartige Blöcke können hierarchisch strukturiert sein. Der Block auf oberster Ebene heißt **System** (siehe Abb. 2.75).

Blöcke auf unterster Ebene heißen Prozessinteraktionsdiagramme (siehe Abb. 2.76).

Die Hierarchie insgesamt zeigt die Abb. 2.77.

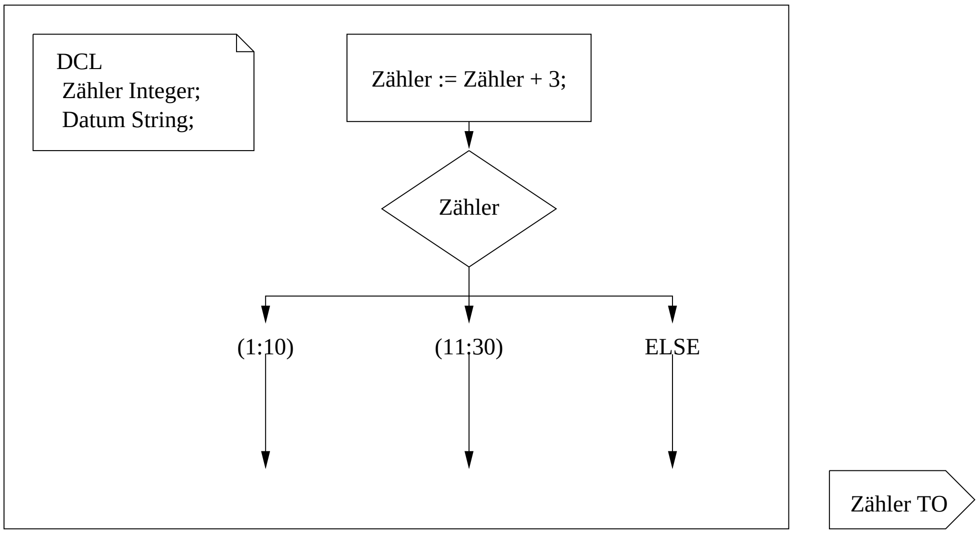


Abbildung 2.74: Benutzung von Daten in Prozessen

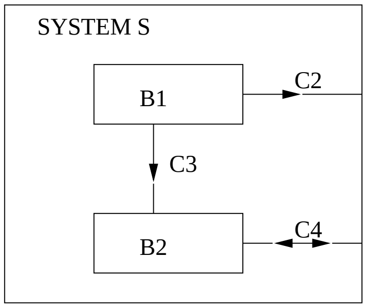


Abbildung 2.75: Beispiel eines Systems

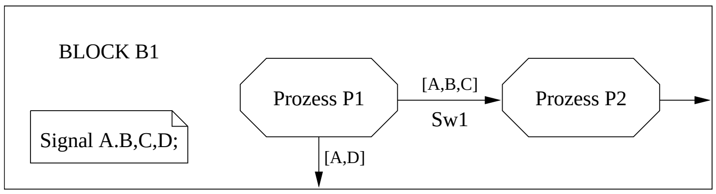


Abbildung 2.76: Beispiel eines Prozessinteraktionsdiagramms

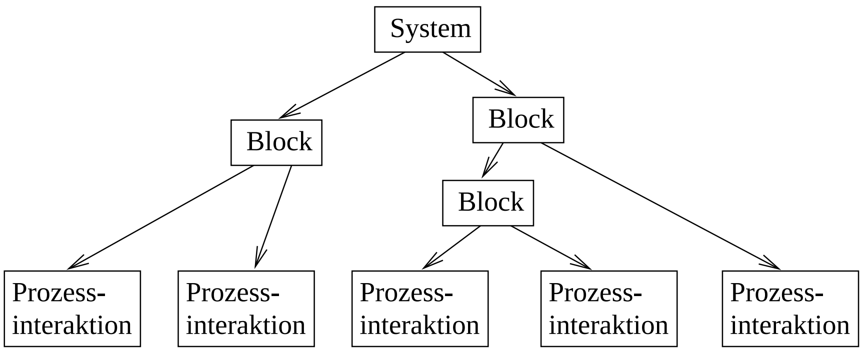


Abbildung 2.77: SDL-Hierarchie

In der Interprozesskommunikation gibt es mehrere Methoden der Adressierung des Empfängers:

1. **Durch die Angabe einer Zieladresse**: Die Nummern von Kindes- und Elternprozessen können erfragt

werden (siehe Abb. 2.78).

Abbildung 2.78: Adressierung von Kindesprozessen

2. **Durch indirekte Adressierung**: Der Empfänger ergibt sich in diesem Fall aus dem Kontext. So ist beispielsweise in der Abb. 2.79 der Weg von Signal `B` aus `P1` eindeutig bestimmt.

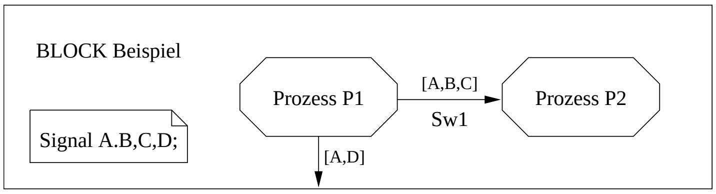


Abbildung 2.79: Indirekte Adressierung

3. **Durch Adressierung eines Kanalwegs**: Beispielsweise wird in Abb. 2.80 der Kanal direkt angegeben.

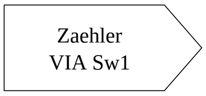


Abbildung 2.80: Adressierung eines Kanalwegs

2.5.2 Bewertung

Die folgenden Eigenschaften ragen bei einer globalen Bewertung von SDL hervor:

- Prozesse können dynamisch erzeugt werden.
- Es gibt keine hierarchischen Prozesse, insbesondere keine Kapselung einer Prozesshierarchie.
- Prozesse können sich selbst beenden
- Nichtdeterminismus ist bei gleichzeitigem Eintreffen an einer Warteschlange vorhanden
- SDL zählt (i.Ggs. zu VHDL) zu den synchronen Sprachen
- es gibt keinen allgemeinen Broadcast-Mechanismus, der bei verteilten Systemen schwierig zu realisieren wäre.

2.5.3 Übersetzung von SDL nach VHDL

Figueiredo und Bonatti beschreiben in [BLR95] ein Verfahren zur Übersetzung einer SDL-Beschreibung nach VHDL. Ziel ist auch hier, auf diese Weise einen Weg zur Realisierung in Hardware zu etablieren. Die Tabelle 2.6 zeigt, wie SDL-Elemente auf VHDL-Elemente abgebildet werden.

Das erzeugte VHDL-Modell hat weiterhin die folgenden Eigenschaften:

- es wird stets eine synchrone Realisierung erzeugt,
- alle Prozesse sind für Takt und Reset sensitiv,
- für die Interprozesskommunikation gibt es drei Optionen:
  1. **"no queue"**: es wird lediglich ein einziges Port-Register realisiert. Diese Lösung ist zwar effizient, führt aber evtl. zum Ignorieren von Events oder Signalen.
  2. **ein Register pro Port**: diese Lösung besitzt ein geringeres Risiko, Events zu ignorieren, ist aber bereits aufwendiger.
  3. **FIFO endlicher Länge**: diese Lösung bietet ein noch geringes Risiko, Events zu verlieren, ist aber noch aufwendiger

SDL	**VHDL**
SYSTEM	VHDL *entity*
BLOCK	VHDL *entity*
PROZESS	1 Entity mit 2 Prozessen:
(nur eine Instanz,	a) SDL-Prozess
keine dynamische Erzeugung)	b) SDL-Kommunikationsschema
Datenoperationen	VHDL-Zuweisungen und Arithmetik-Operationen
Bedingungen	IF .. THEN ... ELSE
KANAL, SIGNALWEG	VHDL-Signale und Verbindungen
VARIABLE	VHDL-Variable (oder -Signale)
ADTs	Bereiche ganzer Zahlen, Integer, Bit-Vektoren
SORTEN	VHDL-Packages
ZUSTÄNDE	Aufzählungstypen
ZUSTANDSÜBERGÄNGE	CASE, WHEN

Tabelle 2.6: Übersetzung von SDL nach VHDL

An diesen Optionen ist zu erkennen, dass bei der Realisierung einer SDL-Spezifikation stets das Problem auftaucht, dass man die maximal mögliche Länge der FIFO-Queues berechnen können muss, um die Spezifikation tatsächlich korrekt zu implementieren.

- Das Senden von Signalen wird durch Invertieren vermittelt.

Als Beispiel der Übersetzung von SDL nach VHDL betrachten wir einen (amerikanischen) Verkaufautomaten für Kekse, Kartoffelchips, Doughnuts und Bretzel [BLR95]. Abbildung 2.81 zeigt das Gesamtsystem dieses Automaten.

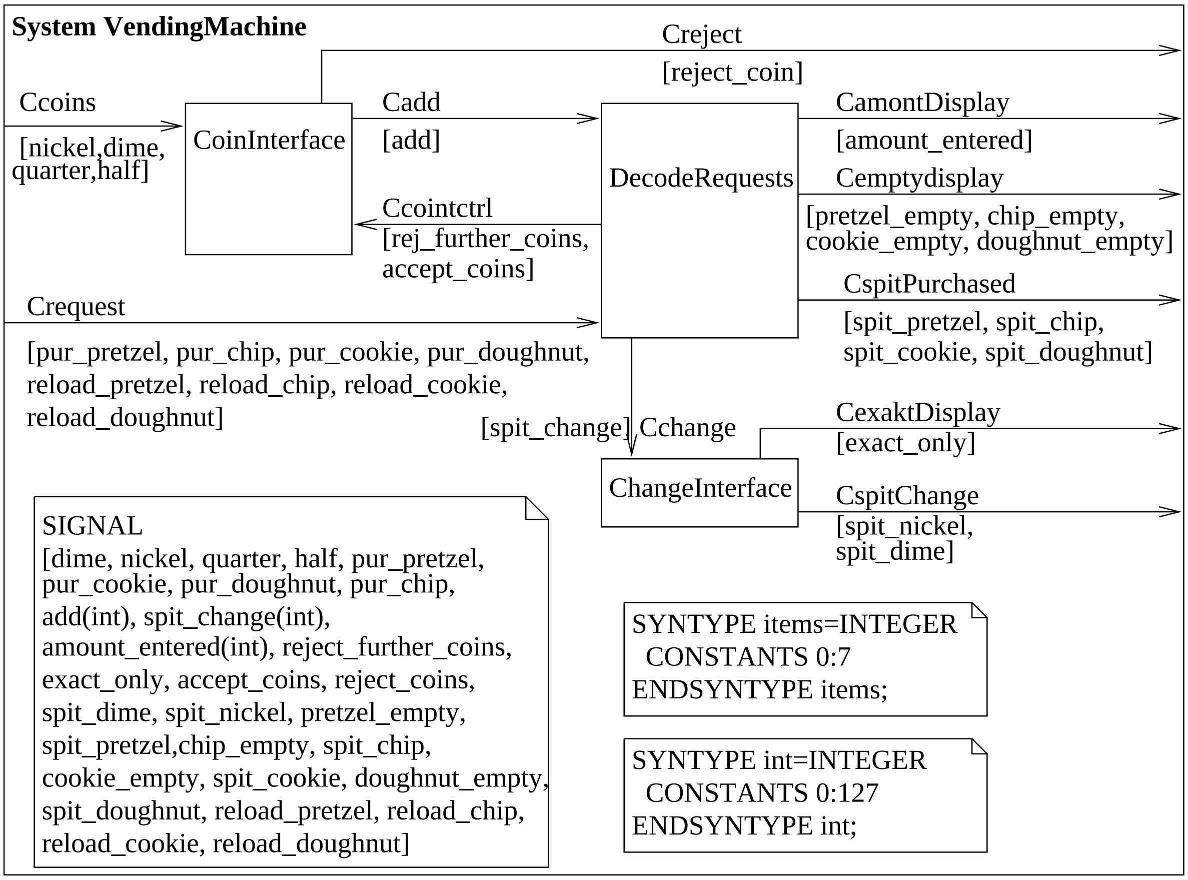


Abbildung 2.81: Gesamtsystem des Verkaufsautomaten

Der Automat akzeptiert Münzen zu je 5, 10, 25 und 50 Cents (*nickels, dimes, quarters, half dollars*). Weiter können die 4 Artikel entweder angefordert oder aufgefüllt werden. Angezeigt wird der eingegebene Betrag, die Liste der z.Z. ausverkauften Artikel und ggf. die Mitteilung, dass die Artikel passend bezahlt werden müssen. Gesteuert wird die Ausgabe von Wechselgeld, die Rückgabe überzahlter Beträge und der gewünschten Artikel.

Abb. 2.82 zeigt beispielhaft einen der Blöcke von `VendingMachine` genauer, nämlich den Block `DecodeRequests`.

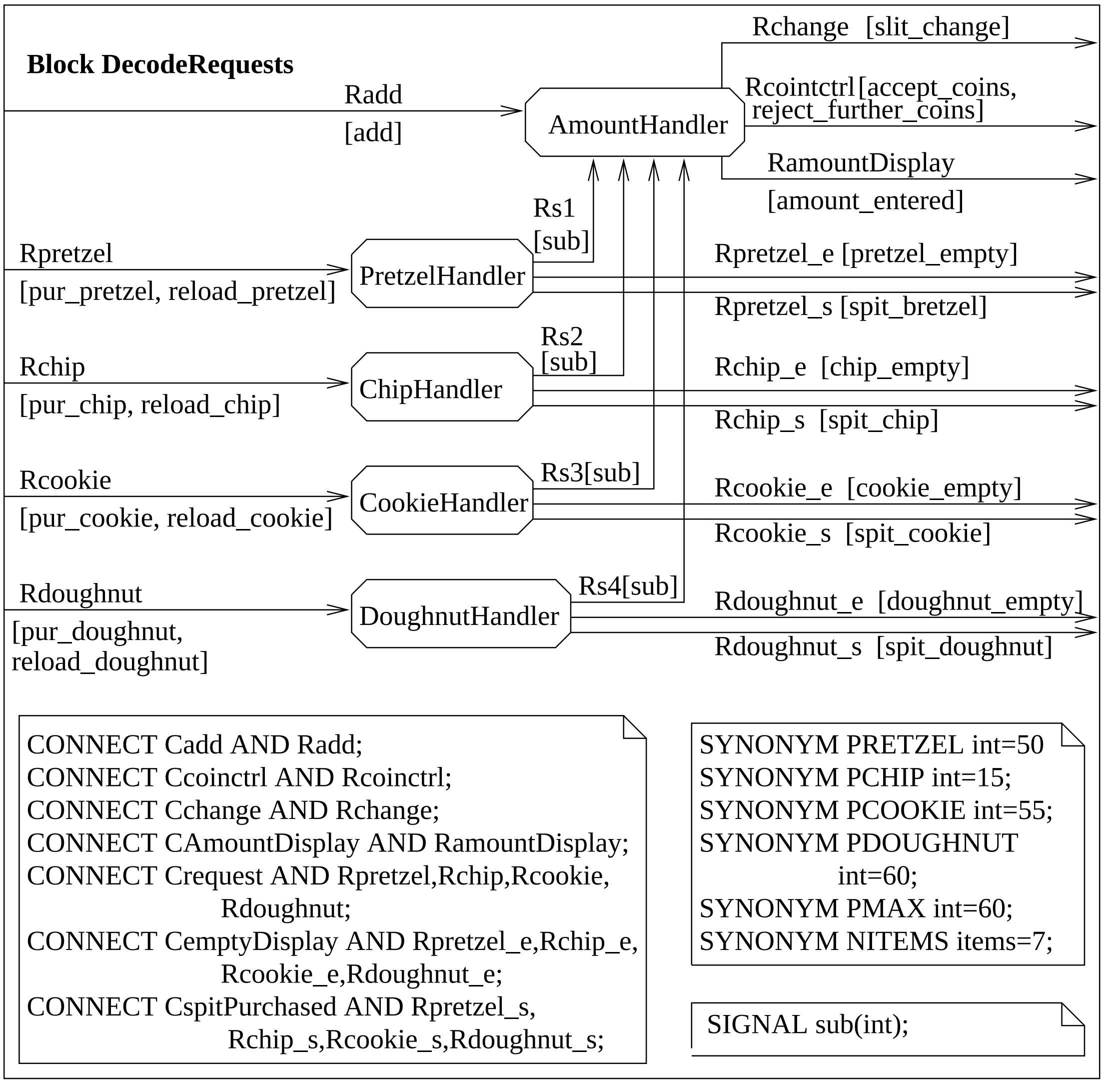


Abbildung 2.82: Block 'DecodeRequests'

Dieser Block enthält einen Prozess zur Berechnung der bereits eingegebenen Summe sowie für jeden der Artikel jeweils einen Prozess. Über `CONNECT`-Statements werden interne mit externen Signalen verbunden.

Als weitere Verfeinerung betrachten wir den Prozess `ChipHandler`. Der entsprechende Automat wird durch die Abb. 2.83 beschrieben.

Dieser Proze wartet im wesentlichen im Zustand `pur_wait` auf einen Verkaufsvorgang. Sofern der Verkauf von Kartoffelchips angefordert wird, wird der Preis von Kartoffelchips mit dem gegenwärtigen Restbetrag an Münzen verglichen. Zu diesem Zweck wird über `VIEWED` der Zugriff auf die globale Variable `current` angefordert. Falls das Geld ausreicht, wird eine Tüte Chips ausgegeben (`spit_chip`) und die Anzahl vorhandener Chip-Tüten dekrementiert. Wird dabei der Wert 0 erreicht, so erfolgt die Anzeige 'Kartoffel-Chips ausverkauft' (`chip_empty`) und der Teilautomat geht in den Zustand `empty`. Nach Eingabe neuer Chips wird die Anzahl der Chip-Tüten wieder initialisiert und der Zustand `pur_wait` eingenommen.

Abbildung 2.84 zeigt die Übersetzung dieses Automaten bzw. Prozesses nach VHDL. Die o.a. Prinzipien der Übersetzung sind zu erkennen.

Derartige VHDL-Beschreibungen können wiederum mittels Synthesewerkzeugen auf Hardware abgebildet werden. Die Originalquelle [BLR95] enthält den entsprechenden Schaltplan. Ein Vergleich zeigt, dass die Beschreibung in der graphischen Variante von SDL mit der geringsten Anzahl von Beschreibungselementen auskommt und damit wohl auch am leichtesten zu überblicken ist (siehe Tabelle 2.7).

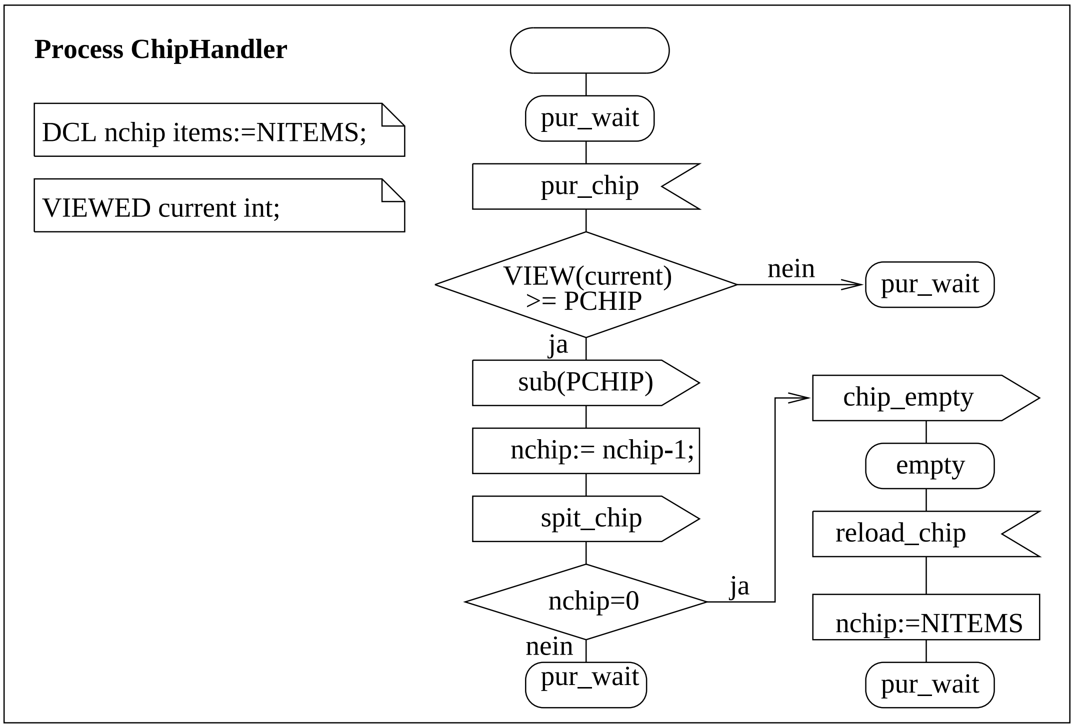


Abbildung 2.83: Prozess 'ChipHandler'

```
p_chiphandler: PROCESS(clock, reset)
  VARIABLE nchip : items;
 BEGIN
  IF (reset='1') THEN  cstate_chiphandler <= pur_wait;  -- asynchr. Reset
                       nchip := NITEMS;
  ELSIF (clock'EVENT AND clock='1' AND clock'LAST_VALUE='0') THEN
   CASE cstate_chiphandler IS
    WHEN pur_wait =>
     IF (rec_pur_chip = '1') THEN
       IF (revealed_current >= PCHIP) THEN
        send_param1_sub <= PCHIP;
        send_sub        <= NOT (send_sub);
        nchip := nchip-1;
        send_spit_chip <= NOT (send_spit_chip);
         IF (nchip=0) THEN
           send_chip_empty <= NOT (send_chip_empty);
           cstate_chiphandler <= empty;
         END IF;
       END IF;
     END IF;
    WHEN empty =>
     IF (rec_reload_chip='1') THEN nchip : NITEMS; cstate_chipchandler <= pur_wait;
     END IF;
   END CASE;
  END IF;
 END PROCESS p_chiphandler;
```

Abbildung 2.84: VHDL-Beschreibung des Prozesses 'ChipHandler'

SDL-GR [graphische Symbole]	SDL-PR [Zeilen Code]	VHDL [Zeilen Code]	Flip-Flops [Stück]	Gatter [Stück]
160	610	1470	140	570

Tabelle 2.7: Anzahl der Beschreibungselemente

2.6 Petri-Netze

8. Vorles.

2.6.1 Einführung

Petri-Netze sind von Carl Adam Petri 1962 entwickelt worden, um kausale Abhängigkeiten zu modellieren. Petri-Netze enthalten keinen expliziten Bezug auf die Zeit und setzen keine globale Synchronisation voraus. Sie sind daher v.a. zur Modellierung verteilter Systeme geeignet.

Petri-Netze basieren ganz wesentlich auf den Begriffen **Bedingung** und **Ereignis**. Bedingungen werden in der graphischen Darstellung von Petri-Netzen in der Regel als Kreise dargestellt. Bedingungen können erfüllt sein oder nicht erfüllt sein. In der graphischen Darstellung werden erfüllte Bedingungen durch Marken innerhalb der Kreise repräsentiert.

Ereignisse werden in der graphischen Darstellung durch Rechtecke symbolisiert. Ereignisse können nur stattfinden, wenn gewisse Vorbedingungen erfüllt sind. Dies wird durch Kanten von jeweiligen Bedingungen zu Ereignissen dargestellt. Stattfindende Ereignisse führen dazu, dass neue Bedingungen erfüllt sind. Für diese Bedingungen werden in graphischen Darstellungen Kanten von den Ereignissen zu den Bedingungen eingezeichnet. Nach Ablauf der Ereignisse sind die Vorbedingungen im allgemeinen nicht mehr erfüllt.

Als Beispiel betrachten wir die Modellierung der Synchronisation von Zügen auf einer eingleisigen Strecke. Die Berechtigung zum Befahren der eingleisigen Strecke wird durch eine Marke in der Mitte dargestellt. Trifft beispielsweise ein Zug ein, der die Strecke nach rechts befahren möchte, so wird dies durch eine erfüllte Bedingung links oben repräsentiert (Abb. 2.85).

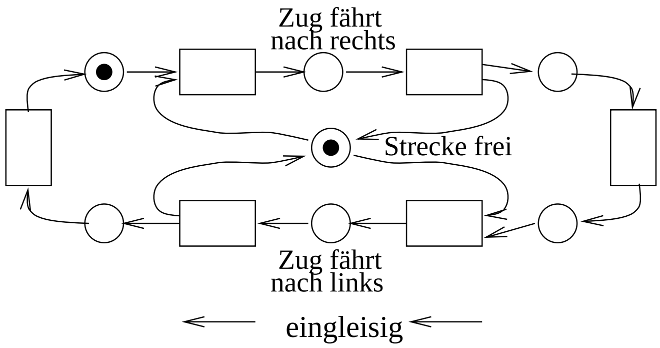


Abbildung 2.85: Synchronisation von Zügen auf einer eingleisigen Strecke

Ist die Strecke frei, so werden beide Marken abgezogen (siehe Abb. 2.86).

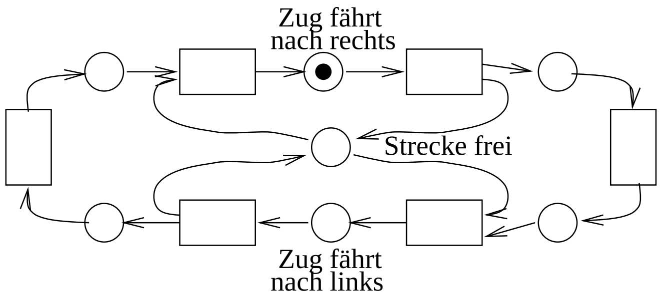


Abbildung 2.86: Fahrt eines Zuges nach rechts

Erst nach dem Verlassen der Strecke erhält die mittlere Bedingung wieder eine Marke (siehe Abb. 2.87).

Das beschriebene Ändern der Markierung nennt man auch das **Marken-Spiel** (engl. *token game*).

2.6.2 Bedingungs/Ereignis-Systeme

Nach der intuitiven Einführung wollen wir uns jetzt mit einigen präziseren Definitionen beschäftigen [Rei86].

Def.: Ein Tripel $N = (B, E, F)$ heißt **Netz**, falls gilt:

1. B und E sind disjunkte Mengen

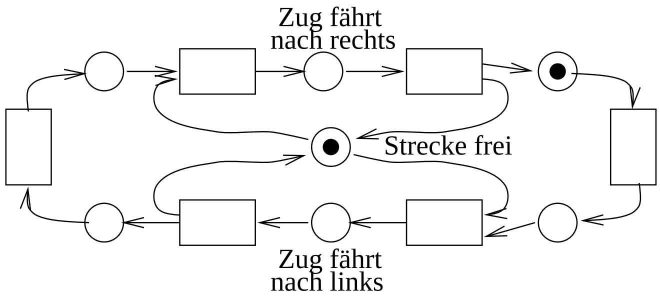


Abbildung 2.87: Zug hat eingleisige Strecke verlassen

2. $F \subseteq (E \times B) \cup (B \times E)$ ist eine zweistellige Relation, die Flussrelation von N.

Def.: Sei N ein Netz. Für $x \in (B \cup E)$ heißt $^\bullet x := \{y|yFx\}$ der **Vorbereich** und $x^\bullet := \{y|xFy\}$ der Nachbereich.

Def.: Ein Paar $(b, e) \in B \times E$ heißt Schlinge, falls $bFe \wedge eFb$. N heißt rein, falls F keine Schlingen enthält (siehe Abb. 2.88 links).

Def.: Ein Netz heißt **schlicht**, falls verschiedene Elemente nicht denselben Vor- und Nachbereich haben (siehe Abb. 2.88 rechts).

Abbildung 2.88: Nicht reine bzw. nicht schlichte Netze

Bedingungs/Ereignis-Systeme (B/E-Systeme) sind schlichte Netze ohne isolierte Elemente, die zusätzlichen Einschränkungen hinsichtlich der Struktur genügen.

Aufgrund der wechselseitigen Verbindung von Bedingungen und Ereignissen handelt es sich bei Petri-Netzen formal um **bipartite Graphen**. In B/E-Systemen ist stets maximal eine Marke pro Bedingung erlaubt. Da wir ohnehin allgemeinere Netze betrachten werden, sind die o.a. Einschränkungen hier nicht relevant.

Im Folgenden wollen wir diese allgemeineren Systeme betrachten.

2.6.3 Stellen/Transitionen-Systeme

Stellen/Transitionen-Systeme (S/T-Systeme) sind derartige Systeme. Im Gegensatz zu B/E-Systemen erlauben sie mehrere Marken pro Bedingung (hier Stelle genannt) und gewichtete Kanten zu und von Ereignissen (hier Transitionen genannt). Die Anzahl der Marken pro Bedingung heißt **Markierung**.

Def.: Eine **Markierung** ist eine Abbildung $M : S \to \mathbb{N} \cup \{\omega\}$.

Für Stellen/Transitionen-Systeme sind neben den aktuellen Markierung noch die jeweiligen Anfangsmarkierungen, die Kapazitäten der Stellen und die Anzahl der über Kanten fließenden Marken zu definieren.

Def.: Ein Tupel (S, T, F, K, W, M_0) heißt Stellen/Transitionen-System (S/T-System) $\Longleftrightarrow$

1. $N = (S, T, F)$ ist ein Netz mit Stellen (*places*) $s \in S$ und Transitionen (*transitions*) $t \in T$.
2. Die Abbildung $K : S \to (\mathbb{N} \cup \{\omega\}) \setminus \{0\}$ ist die Kapazität der Stellen (ω entspricht unbegrenzter Kapazität).
3. Die Abbildung $W : F \to (\mathbb{N} \setminus \{0\})$ ist das Gewicht der Kanten des Graphen.
4. Die Abbildung $M_0 : S \to \mathbb{N} \cup \{\omega\}$ ist die Anfangsmarkierung der Stellen.

Falls eine Transition $t \in T$ stattfindet, verändert sich die Markenzahl entsprechend der Kantengewichte W. Das Gewicht zur Transition hinführender Kanten beschreibt die Anzahl der im Vorbereich zu reduzierenden Marken. Das Gewicht von der Transition wegführender Kanten beschreibt die Anzahl der im Nachbereich zu erzeugender Marken.

Def.: Eine schaltende Transistion bestimmt eine Folgemarkierung M' einer aktuellen Markierung M durch:

$$M'(s) = \begin{cases} M(s) - W(s,t), & \text{falls } s \in {}^\bullet t \setminus t^\bullet \\ M(s) + W(t,s), & \text{falls } s \in t^\bullet \setminus {}^\bullet t \\ M(s) - W(s,t) + W(t,s), & \text{falls } s \in {}^\bullet t \cap t^\bullet \\ M(s) & \text{sonst} \end{cases}$$

Abbildung 2.89 zeigt die Markenveränderung aufgrund einer schaltenden Transition am Beispiel.

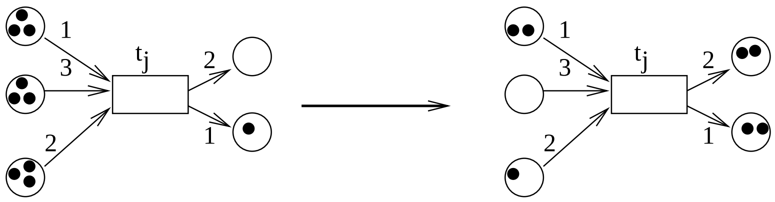


Abbildung 2.89: Veränderung der Markierung

Für nicht beschriftete Kanten bzw. Stellen gilt gegebenenfalls $W(f) = 1$ bzw. $K(s) = \omega$.

Transitionen **können** schalten, wenn für alle Stellen im Vorbereich eine mindestens dem Kantengewicht entsprechende Markenzahl vorhanden ist und wenn alle Stellen im Nachbereich noch ausreichend Kapazität besitzen, um die dem Kantengewicht entsprechende Markenzahl zusätzlich aufzunehmen. Transistionen, welche diese Bedingung erfüllen, heißen **aktiviert**.

Def.: Eine Transition $t \in T$ heißt M-aktiviert $\iff$

$$(\forall s \in {}^\bullet t : M(s) \geq W(s,t)) \wedge (\forall s \in t^\bullet : M(s) + W(t,s) \leq K(s))$$

Aktivierte Transistionen können schalten, d.h. zu einer Veränderung der Markenzahlen führen, müssen dies aber nicht.

Def.: Für alle Transitionen $t \in T$ wird jeweils eine Funktion $\underline{t} : S \to \mathbb{Z}$ definiert durch:

$$\underline{t}(s) = \begin{cases} -W(s,t), & \text{falls } s \in {}^\bullet t \setminus t^\bullet \\ +W(t,s), & \text{falls } s \in t^\bullet \setminus {}^\bullet t \\ -W(s,t) + W(t,s), & \text{falls } s \in {}^\bullet t \cap t^\bullet \\ 0 & \text{sonst} \end{cases}$$

Eine M-aktivierte Transition t bestimmt damit eine Folgemarkierung M' durch

$$\forall s \in S : M'(s) = M(s) + \underline{t}(s)$$

Versteht man unter M und $\underline{t}$ Vektoren und unter '+' die Vektoraddition, so gilt

$$M' = M + \underline{t}$$

Aus den Vektoren $\underline{t}$ bilden wir nun als nächstes die sog. Inzidenzmatrix.

Def.: Die Inzidenzmatrix $\underline{N}$ eines Netzes N ist eine Abbildung $\underline{N} : S \times T \to \mathbb{Z}$
mit $\forall t \in T : \underline{N}(s,t) = \underline{t}(s)$

Für reine Netze kann aus $\underline{N}$ wieder W berechnet werden; für andere Netze ist dies aufgrund der Bildung der Differenzen von zu- und abfließenden Marken nicht mehr eindeutig möglich.

Ein Netz N heißt **kontaktfrei**, falls die Aktivierung von Transitionen nie an unzureichenden Kapazitäten scheitert. Bei kontaktfreien Netzen spielen die Kapazitäten keine Rolle.

Im Buch von Reisig [Rei86] (S. 73) wird gezeigt, dass jedes S/T-Netz in ein äquivalentes kontaktfreies Netz transformiert werden kann. Allerdings kann dieses Netz größer als das ursprüngliche sein.

Für reine und kontaktfreie Netze beschreiben $\underline{\text{N}}$ und M_0 zusammen das Netz vollständig.

2.6.4 Invarianten

Einer der wesentlichen Vorteile von Petri-Netzen ist es, dass einige formale Beweise von Systemeigenschaften verhältnismäßig einfach geführt werden können. Dies gelingt insbesondere mit Hilfe der sog. **Invarianten**.

Wir beginnen hier mit den S-Invarianten. Wir suchen dazu **Stellenmengen konstanter Markensumme**. Beim Schalten einer Transition $t_j \in T$ ist die Markensumme innerhalb von $R \subseteq S$ konstant, wenn

$$\sum_{s \in R} \underline{t}_j(s) = 0$$

ist.

Beispiel:

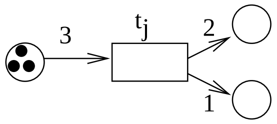


Abbildung 2.90: Drei Stellen mit konstanter Markensumme

Zur Vereinfachung der Schreibarbeit führen wir als nächstes den **charakteristischen Vektor** ein. Der **charakteristische Vektor** einer Menge $R \subseteq S$ ist für jede Komponente $s \in S$ definiert durch:

$$\underline{c}_R(s) = \begin{cases} 1 & \text{falls } s \in R \\ 0 & \text{falls } s \notin R \end{cases}$$

Damit läßt sich eine Summe über Vektoren t_j als Skalarprodukt $\cdot$ schreiben:

$$\sum_{s \in R} \underline{t}_j(s) = \underline{t}_j \cdot \underline{c}_R = \sum_{s \in S} \underline{(t)}_j(s) * c_R(s)$$

Falls die Markensumme für alle $\underline{t}_j \in T$ konstant ist, muss gelten:

$$\begin{array}{rcl} \underline{t}_1 \cdot \underline{c}_R & = & 0 \\ \dots & .. & .. \\ \dots & .. & .. \\ \underline{t}_n \cdot \underline{c}_R & = & 0 \end{array}$$

Daraus folgt, dass die Markensumme für jene Stellenmengen konstant ist, für welche die charakteristischen Vektoren das lineare homogene Gleichungssystem

$$\underline{N}^T \cdot \underline{c}_R \;=\; 0 \qquad (2.1)$$

mit der transponierten Inzidenzmatrix

$$\underline{N}^T = \begin{pmatrix} \underline{t}_1 \\ .. \\ .. \\ \underline{t}_n \end{pmatrix}$$

erfüllen.

Ausgeschrieben lautet die Gleichung 2.1:

$$\begin{pmatrix} \underline{t}_1(s_1)...\underline{t}_1(s_n) \\ \underline{t}_2(s_1)...\underline{t}_2(s_n) \\ ... \\ \underline{t}_m(s_1)...\underline{t}_m(s_n) \end{pmatrix} \begin{pmatrix} c_R(s_1) \\ .. \\ .. \\ c_R(s_n) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ .. \\ 0 \end{pmatrix}$$

In der ausführlichsten Form kann man dies schreiben als:

$$\begin{aligned} \underline{t}_1(s_1) + ... + \underline{t}_1(s_n) &= 0 \\ \underline{t}_2(s_1) + ... + \underline{t}_2(s_n) &= 0 \\ ... + ... + ... &= 0 \quad (2.2) \\ \underline{t}_m(s_1) + ... + \underline{t}_m(s_n) &= 0 \end{aligned}$$

(2.3)

Wir können also S-Invarianten (Stellenmengen konstanter Markensumme) finden, indem wir dieses lineare Gleichungssystem lösen. Dabei ist $\underline{N}$ vorgegeben und die Lösungsvektoren $\underline{c}_R$ sind zu bestimmen, wobei zu berücksichtigen ist, dass Lösungsvektoren nur die Komponenten 0 und 1 besitzen dürfen.

Generell nennt man Gleichungssysteme, deren Lösungen ganzzahlig sein müssen, **diophantische** Gleichungssysteme. Durch die Forderung der Ganzzahligkeit wird die Komplexität von Lösungsverfahren im allgemeinen deutlich erhöht.

Wir wollen die Berechnung von S-Invarianten hier an einem Beispiel betrachten, dem sog. Eisenbahnbeispiel.

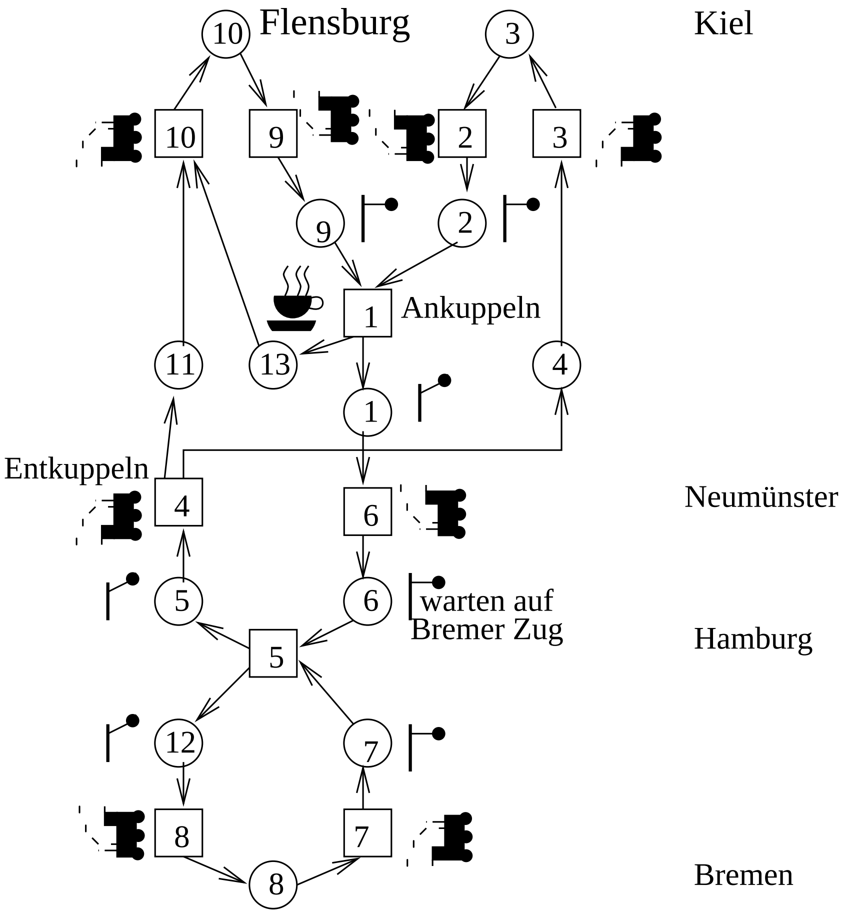


Abbildung 2.91: Modell des Zugverkehrs zwischen Hamburg und Flensburg bzw. Kiel

Das Beispiel modelliert die Organisation des Zugverkehrs im Falle der Trennung eines Zuges in zwei Teilzüge, die in unterschiedliche Richtungen weiterfahren. Eine solche Situation gibt es beispielsweise für den Zugverkehr zwischen Hamburg und Flensburg bzw. Kiel. In Neumünster werden von Hamburg kommende Züge in Teilzüge für die beiden Städte getrennt und bei der Rückfahrt wieder zusammengefügt. Um die Situation interessanter zu gestalten, wird in Hamburg noch die Synchronisation mit den Zügen von und nach Bremen modelliert. Ferner beschreibt eine Stelle den Lokomotivführer, der jeweils in Neumünster warten muss (siehe Abb. 2.91). Offensichtlich sind alle Kantengewichte und alle Kapazitäten gleich 1.

Die Tabelle 2.8 zeigt die zugehörige Inzidenzmatrix.

Unter den Zeilen gibt eine eine lineare Abhängigkeit (beispielsweise ist die Summe der Zeilen 1 bis 9 gleich dem Negativen der Zeile 10), der Rang der Matrix ist folglich 9. Folglich ergibt sich ein (13-9)=4-dimensionaler Lösungs-

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}
t_1	1	-1							-1				1
t_2		1	-1										
t_3			1	-1									
t_4				1	-1						1		
t_5					1	-1	-1					1	
t_6	-1					1							
t_7							1	-1					
t_8								1				-1	
t_9									1	-1			
t_{10}										1	-1		-1

Tabelle 2.8: Transponierte Inzidenzmatrix des Eisenbahnbeispiels

raum, in dem sich charakteristische Vektoren befinden können. Da wir bereits bei der Numerierung der Stellen und Transitionen für eine übersichtliche Anordnung von 0 verschiedener Matrixelemente gesorgt haben, können wir im Folgenden vier linear unabhängige Vektoren leicht bestimmen. Dazu bilden wir vier Zeilenvektoren b_1 bis b_4, in die wir in den Spalten 6, 11, 12, und 13 jeweils genau eine 1 und sonst Nullen einsetzen. Die Werte für die übrigen Spalten ergeben sich dann aus den Gleichungen 2.2. Die vier Vektoren werden sicher linear unabhängig sein.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}
$b_1 = \underline{c}_{R_1}$	1	1	1	1	1	**1**	0	0	0	0	**0**	**0**	**0**
b_2	0	-1	-1	-1	0	**0**	0	0	1	1	**1**	**0**	**0**
$b_3 = \underline{c}_{R_3}$	0	0	0	0	0	**0**	1	1	0	0	**0**	**1**	**0**
$b_4 = \underline{c}_{R_4}$	0	0	0	0	0	**0**	0	0	1	1	**0**	**0**	**1**
$b_1 + b_2 = \underline{c}_{R_2}$	1	0	0	0	1	1	0	0	1	1	1	0	0

Tabelle 2.9: Basisvektoren des Lösungsraums

Wir haben Glück: drei der erzeugten Basisvektoren sind bereits auch charakteristische Vektoren. Für den vierten benötigen wir eine Linearkombination mit b_2. Durch eine einfache Kombination mit b_1 erhalten wir bereits einen vierten charakteristischen Vektor. Als nächstes sehen wir uns an, welche Stellenmengen durch diese charakteristischen Vektoren beschrieben werden (siehe Abb. 2.92).

Die vier Invarianten bedeuten, dass die Anzahl der Kieler Teilzüge, der Flensburger Teilzüge. der Bremer Züge und der Flensburger Lokomotivführer ($\underline{c}_{R_4}$) konstant ist (das wird man wohl auch erwarten). Vereinigungen von disjunkten Stellenmengen konstanter Markensumme sind natürlich auch wieder Stellenmengen konstanter Markensumme. Derartige Stellenmengen bieten aber im Beispiel keinerlei neue Information.

Das von uns benutzte Netz ist ein sog. **Synchronisationsgraph**. Man kann allgemein zeigen, dass Kreise in Synchronisationsgraphen S-Invarianten bilden [Rei86].

Da die Markensummen für die bezeichneten Stellenmengen konstant sind, muss jede solche Menge auch bereits bei der Initialisierung mindestens eine Marke enthalten, sonst wäre der entsprechende Kreis tot. Man kann leicht sehen, dass mindestens drei Marken benötigt werden, damit das Netz nicht tot ist.

Nachweise von der gezeigten Form kann man vielfach einsetzen, um Eigenschaften von Systemen formal nachzuweisen. Ein Standardfall ist der Nachweis für Modelle kritischer Abschnitte[15], dass jeweils maximal eine Marke in den Stellen des kritischen Abschnitts existiert.

Neben den S-Invarianten gibt es noch die T-Invarianten. Diese beschreiben, wie häufig Transitionen schalten müssen, damit eine gegebene Markierung reproduziert werden kann. Vektoren v, welche diese Häufigkeiten beschreiben, erhält man über das Gleichungssystem [Rei86]:

$$\underline{N} \cdot \underline{v}_R = 0 \qquad (2.4)$$

[15] Siehe Vorlesung "Betriebssysteme".

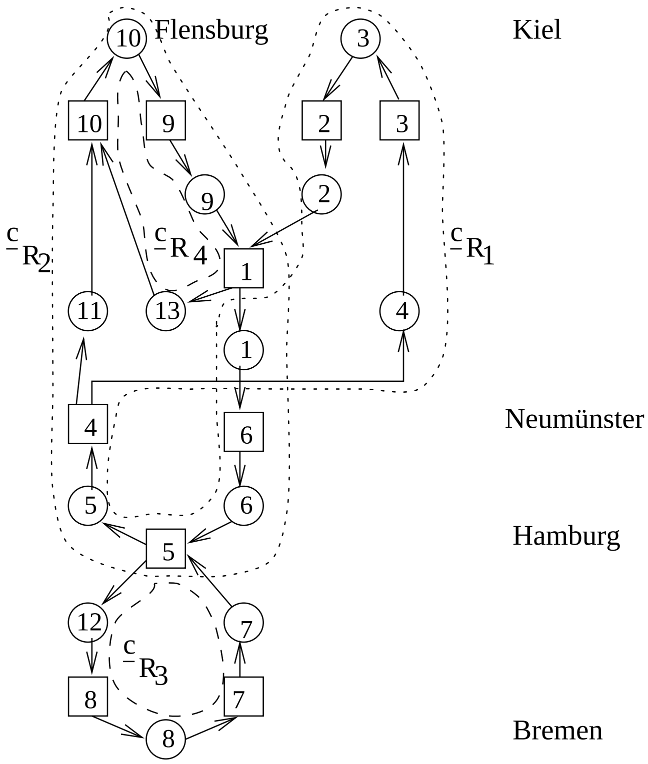


Abbildung 2.92: S-Invarianten des Eisenbahnbeispiels

2.6.5 Prädikat/Ereignis-Netze

Mit einer Erweiterung von S/T-Netzen bzw. B/E-Netzen ist es möglich, große Netze kompakter zu beschreiben und dennoch nicht auf die klare Semantik von S/T-Netzen zu verzichten.

Wir betrachten dazu als einführendes Beispiel das **Problem der dinierenden Philosophen**. Man stellt sich dabei vor, dass um einen Tisch herum beispielsweise drei Philosophen sitzen, welche entweder essen oder denken können. Vor ihnen befindet sich jeweils ein Teller mit Spaghetti. Zwischen je zwei Tellern befindet sich eine Gabel. Ein Philosoph benötigt sowohl seine linke als auch seine rechte Gabel, um essen zu können (siehe Abb. 2.93).

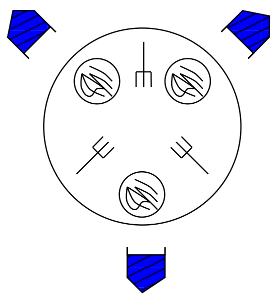


Abbildung 2.93: Das Problem der dinierenden Philosophen

Im Folgenden sollen die Booleschen Variablen d_i, e_i und g_i darstellen, ob der Philosoph i denkt, ißt bzw. ob seine rechte Gabel frei ist. Die Abb. 2.94 beschreibt die Situation als B/E-Netz.

Dieses Netz ist unübersichtlich und nicht leicht auf den Fall n dinierender Philosophen zu verallgemeinern. Abhilfe bietet die Einführung von Prädikaten. Wir definieren dazu Funktionen $l(x)$ und $r(x)$, die jeweils zu einem Philosophen x die linke bzw. die rechte Gabel darstellen sollen. Mit diesen können wir das Problem der dinierenden Philosophen wie in Abb. 2.94 darstellen.

In einem solchen Netz sind die Marken jetzt unterscheidbare Individuen. Mit der Variablen x beschriftete Kanten bedeuten, dass ein Individuum x eine Transition auslöst, bzw. durch eine solche erzeugt wird. Funktionen an den Kanten bedeuten, dass dem Funktionsergebnis entsprechende Individuen vorhanden sein müssen, damit die Ereignisse (hier *events* genannt) stattfinden können bzw. dass dem Funktionsergebnis entsprechende Individuen erzeugt werden. Das *event* `u` beispielsweise entsteht, wenn ein Philosoph x den essenden Zustand `e` verläßt. Es

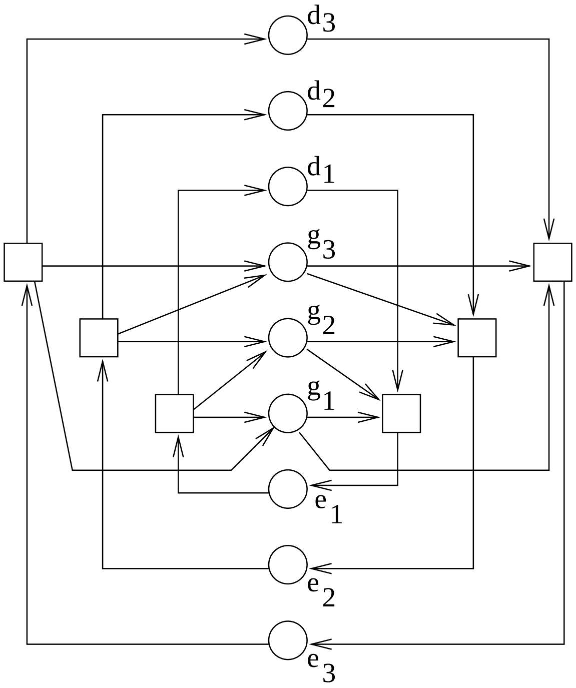


Abbildung 2.94: P/E-Netz der dinierenden Philosophen

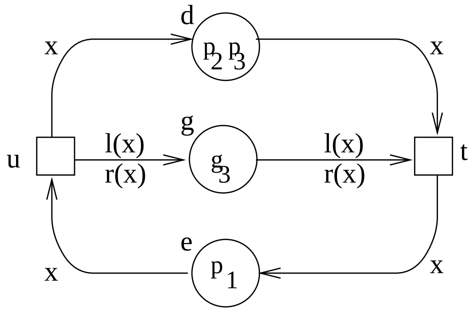


Abbildung 2.95: P/E-Netz der dinierenden Philosophen

werden dann die Gabeln $l(x)$ und $r(x)$ frei und er selbst geht in den Zustand des Denkens über. Ein Übergang in den Zustand des Essens durch den Philosophen x setzt voraus, dass die Gabeln $l(x)$ und $r(x)$ frei sind.

Dieses Modell kann jetzt auf beliebig viele Philosophen verallgemeinert werden. Seine Semantik kann durch Duplizierung des Netzes für die einzelnen Individuen erklärt werden.

2.6.6 Bewertung

Der Vorteil von Petri-Netzen liegt in der Benutzung rein lokaler Entscheidungen. Sie setzen keine globale Uhr voraus. Sie sind daher v.a. zur Beschreibung verteilter Applikationen geeignet.

Weiter existiert eine ausgebaute Theorie zum formalen Nachweis von Eigenschaften. Die Semantik der Netze ist recht klar. Dies gilt auch für P/E-Netze, deren Bedeutung sich durch Rückführung auf B/E- bzw. S/T-Netze erklären läßt.

Standard-Netze besitzen den Nachteil, dass keine Programmiersprachen-Elemente unterstützt werden, dass keine explizite Zeitmodellierung vorgesehen ist (auch nicht lokal) und dass derartige Netze aufgrund der fehlenden Hierarchie leicht unübersichtlich werden.

2.7 Message sequencing charts

Message sequencing charts werden häufig zur Darstellung von zeitlichen Abläufen bei räumlich verteilten Objekten benutzt. Abb. 2.96 zeigt ein Beispiel [Sta94].

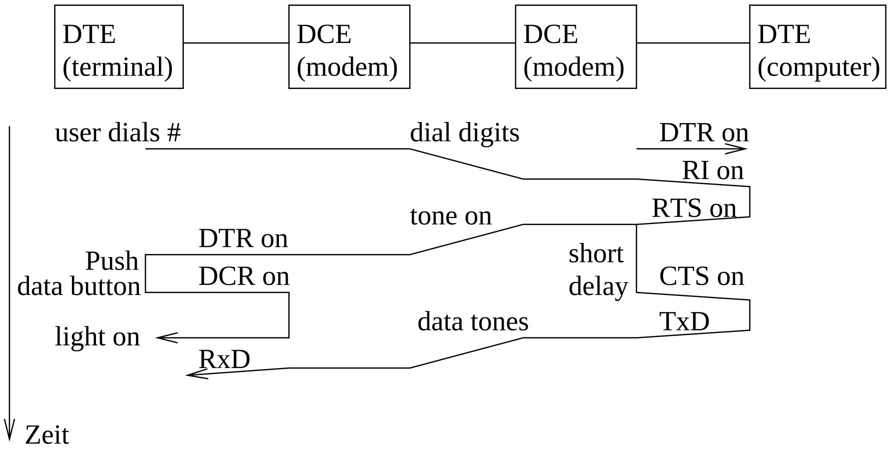


Abbildung 2.96: Message sequencing chart (Aufbau einer RS 232-D-Daten-Verbindung)

Message sequencing charts werden u.a. auch zur Darstellung von Fahrplänen eingesetzt. Diese Darstellungen zeigen allerdings immer nur beispielhafte zeitliche Abläufe und keine exakte Spezifikation zeitlicher Spielräume und Randbedingungen. Für formale Techniken sind sie nicht geeignet.

2.8 UML

UML (Unified Modelling Language) [Gro97, Oes97, FS98] ist das Akronym für die umfassendste Sprache zur Modellierung objektorientierter Softwareentwurfsprozesse. Mit der zunehmenden Bedeutung der Sofware in eingebetteten Systemen wird auch die ojektorientierte Softwareentwicklung für den Mikroelektronikentwurf wichtiger. Aus diesem Grund wollen wir hier die wesentlichen Elemente von UML anführen. UML enthält eine Vielzahl von grafischen Beschreibungsmöglichkeiten. Glücklicherweise haben wir viele davon in dieser Vorlesung schon kennengelernt. UML enthält:

1. **Zustandsdiagramme**:

   UML enthält StateCharts als eine der Beschreibungsmöglichkeiten. StateCharts werden v.a. eingesetzt, um reaktive Systeme zu modellieren.

2. **Aktivitätsdiagrammme** (engl. *activity diagrams*):

   Aktivitätsdiagramme stellen im Wesentlichen erweiterte Petri-Netze dar. Zu den Erweiterungen zählt beispielsweise die Möglichkeit, Entscheidungsrauten üblicher Flussdiagramme zu verwenden. In UML werden Netzgraphen darüber hinaus ähnlich wie Zustandsgraphen in SDL angeordnet.

   UML-Aktivitätsdiagramme und die Datenflussbeschreibungen des Abschnitts 2.2.1, die *activity charts* genannt werden, sind voneinander verschieden.

3. **Interaktionsdiagramme** (engl. *interaction diagrams*):

   Interaktionsdiagramme in UML entsprechen im Wesentlichen den uns bekannten *message sequencing charts*.

4. **Klassendiagramme**

   Diese Diagramme geben wider, wie die Klassen der objektorientierten Programmierung voneinander abgeleitet werden.

5. *package diagrams*:

   Diese Diagramme beschreiben die Zergliederung der Software in Komponenten und entsprechen den o.a. *module charts*.

6. *use cases*:

   use cases beschreiben typische Anwendungen geplanter Software.

Eine enge Integration dieser Vielzahl von Beschreibungsmöglichkeiten und ein automatischer Abgleich der Darstellungsformen untereinander ist bislang offenbar nicht vorhanden.

Im Rahmen dieser Vorlesung gibt es keine Notwendigkeit, auf UML näher einzugehen, denn alle Diagrammformen wurden entweder bereits vorgestellt oder sind Teil der Vorlesung "Software-Technologie". Die Aktualität von UML kann aber dennoch dazu dienen, die Beschäftigung mit den bislang vorgestellten Sprachen zusätzlich zu motivieren. UML zeigt auch, dass manche Gebiete der Informatik, wie z.B. Petri-Netze, über Jahre hinweg v.a. theoretisch betrachtet werden und plötzlich verstärkte praktische Bedeutung bekommen.

2.9 Weitere Sprachen

1. **Verilog**
   Verilog ist eine weitere Hardwarebeschreibungssprache. Sie ist etwas weniger flexibel als VHDL, bietet dafür aber in der Regel höhere Simulationsgeschwindigkeiten. Verilog ist v.a. in den USA populär. Hardware-Modelle sind vielfach sowohl in VHDL als auch in Verilog erhältlich.
2. **Estelle**
   Ziel der Entwicklung von **Estelle** ab 1981 war die Bereitstellung von Methoden zur Beschreibung von Protokollen. Aus der Arbeit einer Untergruppe des Standardisierungsgremiums ISO entstand Estelle, eine Sprache, die heute in vielen Anwendungen erprobt ist [Hog89].
   Wegen der zu SDL ähnlichen Konzepte gab es Bemühungen, SDL und Estelle auf eine einheitliche Basis zu stellen. Die Bemühungen haben aber nicht zu dauerhaftem Erfolg geführt.
   Ebenso wie bei SDL basiert die Kommunikation bei Estelle auf Kanälen und FIFO-Puffern. Ebenso werden erweiterte endliche Automaten als Modell benutzt, für das allerdings nur eine Textnotation zur Verfügung steht.
   Beispiel:
```
state unterbrochen, warten, verbunden;
trans
{1} from unterbrochen to warten
         when Benutzer.ICONreq
         begin output PDU.CR end;
{2} from warten to verbunden
         when PDU.CC
         begin output Benutzer.ICONconf end;
{3} from warten to unterbrochen
         when PDU.DR
         begin output Benutzer.IDISind end;
{4} from unterbrochen to verbunden ...
```
3. **CSP**
   CSP (Hoare) ist eine der ersten Sprachen, welche die Kommunikation zwischen verschiedenen Prozessen explizit zu beschreiben gestattet. Die Kommunikation basiert auf benannten Nachrichtenkanälen.
   Beispiel (Prinzip):
```
PROCESS A                                    PROCESS B
.....                                        ......
VAR a ..                                     VAR b ..
 a := 3;                                     ...
 c!a   ;  --schreibe a auf Kanal c           c?b   -- lese b ueber Kanal c
END;                                         END;
```
   Beide Prozesse warten ggf. darauf, dass der andere Prozess ebenfalls am Treffpunkt ankommt ("rendez-vous"-Konzept). Man nennt diese Form der Kommunikation auch *blockierende* Kommunikation.
   CSP ist die Grundlage für die von der Herstellerfirma `INMOS` entwickelte Sprache `OCCAM` und die dafür entwickelten speziellen Prozessoren, die Transputer.
   Ältere Transputer besitzen 4 Hardware-Kanäle. Die Kommunikation ist mit den jeweils daran angeschlossenen Transputern möglich. Innerhalb eines Prozessors können (vom Betriebssystem simulierte) Kanäle zur Interprozesskommunikation benutzt werden.
   Neuere Transputer benutzen die Kommunikation über **virtuelle Kanäle**. Nachrichten über diese Kanäle werden für den Benutzer unsichtbar über Zwischenstationen geschickt. Damit ist die Kommunikation zwischen beliebigen Transputern leicht programmierbar.
4. **Z**
   Z ist eine Sprache zur algebraischen Spezifikation
5. **LOTOS**
   LOTOS ist ebenfalls eine Sprache zur algebraischen Spezifikation.

6. **Esterel**

   Esterel ist eine dritte Sprache zur algebraischen Spezifikation.

7. **Silage**

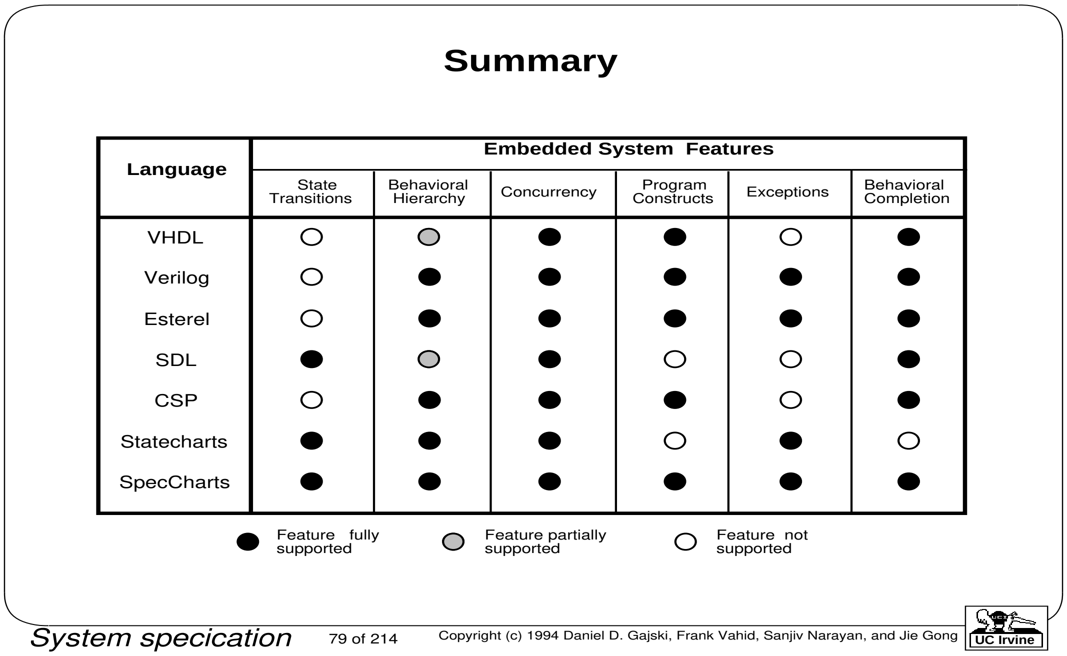
   Silage ist eine spezielle Sprache für die digitale Signalverarbeitung. Es handelt sich um eine funktionale Sprache, die in Datenflussgraphen transformiert werden kann. Die Sprache enthält dem Anwendungsbereich entsprechend spezielle Sprachelemente, beispielsweise zur Angabe der arithmetischen Genauigkeit, eine Unterstützung von Festkommazahlen und von verzögerten Signalen.

8. *HardwareC*

   HardwareC ist eine weitere Hardware-Beschreibungssprache. Die Syntax von HardwareC geht von C aus. HardwareC besitzt relativ einfache Erweiterungen und spezielle Interpretationen von C. Integers entsprechen generischen Parametern von VHDL, Arrays Boolescher Werte entsprechen Bitvektoren. Neben den geschweiften Klammern zur Darstellung sequentieller Ausführung gibt es spitze und eckige Klammern zur Darstellung von Nebenläufigkeit und Parallelität.

2.10 Querbezüge zwischen den Spezifikationssprachen

Gajski vergleicht in seinem Buch die verschiedenen Spezifikationssprachen (siehe Abb. 2.97).



Summary

Language	Embedded System Features					
	State Transitions	Behavioral Hierarchy	Concurrency	Program Constructs	Exceptions	Behavioral Completion
VHDL	○	◐	●	●	○	●
Verilog	○	●	●	●	●	●
Esterel	○	●	●	●	●	●
SDL	●	◐	●	○	○	●
CSP	○	●	●	●	○	●
Statecharts	●	●	●	○	●	○
SpecCharts	●	●	●	●	●	●

● Feature fully supported   ◐ Feature partially supported   ○ Feature not supported

System specication   79 of 214   Copyright (c) 1994 Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong   UC Irvine

Abbildung 2.97: Vergleich von Spezifikationssprachen

Viele dieser Einträge ergeben eine sinnvolle Bewertung. Allerdings ist Esterel sicherlich zur Beschreibung von *state transitions* geeignet und SDL erlaubt Programmiersprachen-Konstrukte. Schließlich sind nur solche Kriterien angeführt, bei denen SpecCharts gut abschneidet. SpecCharts ist im Hinblick auf die Beschreibung einer strukturellen Hierarchie durchaus schwach[16] und die Nebenläufigkeit ist auf flache, statisch vorgegebene Prozesse beschränkt.

Einen etwas umfangreicheren Vergleich enthält die Abb. 2.98 [Nie97].

Die Ziffern in Abb. 2.98 beziehen sich auf Beurteilungen der jeweiligen Sprache, bei denen man auch anderer Meinung sein kann:

1. StateCharts ist einer formalen Analyse zwar nicht leicht zugänglich, dennoch ist diese inzwischen auch möglich [HN96].

[16] Aus diesem Grund arbeitet ein Dortmunder Diplomand z.Z. gemeinsam mit D. Gajski an einer Nachfolgesprache von SpecCharts.

	Standardization	Availability of Tools	Model Executability	Formal Analysis	Non-determinism	Timing Aspects	Exceptions	Synchron./Communication	Concurrency	Behavioural Completion	State-Transitions	Programming Constructs	Behavioural Hierarchy	Structural Hierarchy
Lotos	+	0	-	+	+	-	+	+	+	+	+	+	+	+
SDL	+	0	+	+	+	0	-	+	+	+	+	-	0	+3
Esterel	-	0	+	+	-	-	+	+	+	+	-	+	+	+
StateCharts	-	0	+	-1	-	0	+	+	+	-	+	-	+	-
High-Level Petri Net	-	0	+	+	-4	+6	+	+	+	-	+	-	-	+5
VHDL	+	+	+	-2	-	+	-	+	+	+	-	+	0	+
Verilog	-7	+	+	-	-	+	+	+	+	+	-	+	+	+
SpecCharts	-	0	+	-	-	+	+	+	+	+	+	+	+	-
HardwareC	-	0	+	-	-	0	-	+	+	+	-	+	0	+
CSP/Occam	-	0	+	+	-	-	-	+	+	+	-	+	+	-
Silage	-	0	0	-	-	0	-	-	+	+	-	-	-	-

Abbildung 2.98:

2. Auch VHDL ist einer formalen Analyse nicht leicht zugänglich. Delgado-Kloos [KB95] hat trotzdem einige Ansätze zusammengestellt.
3. Strukturelle Hierarchie bei SDL ist nur in Kombination mit der verhaltensmäßigen Hierarchie unterstützt.
4. Nicht-Determinismus existiert bei Petri-Netzen zwar nicht als separates Element, ist aber im Modell inhärent. Sofern für eine bestimmte Markierung mehrere Transitionen schalten können, erfolgt nichtdeterministisch die Auswahl der nächsten Transition.
5. Für die strukturelle Hierarchie bei Petri-Netzen gilt das Gleiche wie für SDL.
6. Timing wird nur in bestimmten Erweiterungen von Petri-Netzen unterstützt. Für diese ist dann aber die formale Analyse schwierig. Aus der Anzahl der zweifelhaften Einstufungen von Petri-Netzen kann man erkennen, dass (gerade im Ausland) nur sehr unvollständiges Wissen über Petri-Netze bekannt ist.
7. Verilog ist inzwischen auch über IEEE standardisiert worden.

Leider gibt es somit nicht eine einzige Sprache, die alle Anforderungen erfüllt und in praktischen Anwedendungen ist es häufig erforderlich, mehrere Sprachen einzusetzen und Simulationssysteme zu koppeln oder zwischen diesen zu konvertieren.

Bei Bergé [BLR95] findet man Informationen darüber, wie man eine der Sprachen in eine andere übersetzen kann.

Kapitel 3

Zieltechnologien

3.1 Übersicht

Nachdem wir verschiedene Spezifikationstechniken kennengelernt haben, wollen wir jetzt die möglichen Zieltechnologien betrachten, d.h. wir wollen Alternativen für die technische Realisierung von eingebetteten Systemen vorstellen (siehe auch Abb. 3.1).

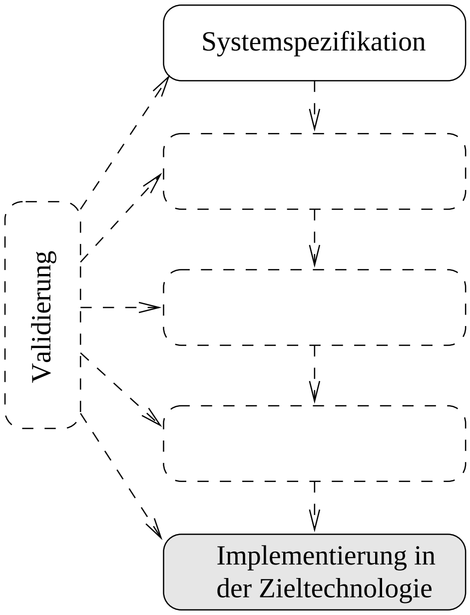


Abbildung 3.1: Kontext der Zieltechnologien

Eine Klassifikation der hardwaretechnischen Möglichkeiten zeigt die Abb. 3.2.

Der größte Aufwand ergibt sich dabei für voll-kundenspezifische Schaltungen (ASICs). Bei geringen Performance-Anforderungen genügt evtl. auch eine reine Compilation auf einen vorhandenen Prozessor. In diesem großen Bereich an Möglichkeiten muss aufgrund der vorgegebenen Randbedingungen jeweils eine geeignete Realisierung gefunden werden.

3.2 Applikations-spezifische Schaltungen (ASICs)

Voll-kundenspezifische Schaltungen (ASICs) [RC89] basieren auf weitgehend unbeschränkter Entwurfsfreiheit. Es werden in diesem Fall integrierte Schaltkreise entwickelt, in denen jeder einzelne Transistor im Prinzip aufgrund der jeweiligen Anforderungen individuell dimensioniert werden kann. Das Layout der Schaltung ist nicht auf Rechteckstrukturen eingeschränkt, die Breiten- zu Längenverhältnisse der Transistoren können den jeweiligen Verhältnissen optimal angpasst werden. Auch schlangenförmige Strukturen zur Realisierung großer Widerstände oder Kondensatoren sind denkbar. Auch können evtl. Sensoren oder optische Schaltkreis-Elemente mit auf dem Chip integriert

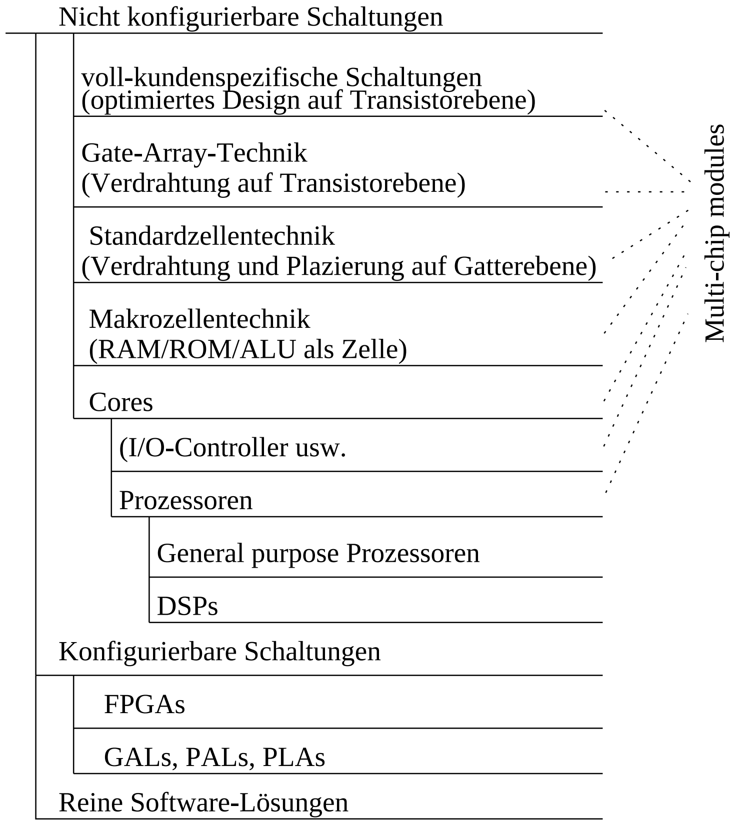


Abbildung 3.2: Zieltechnologien der Mikroelektronik

werden. Diesen Vorteilen steht der Nachteil einer großen Entwicklungszeit, entsprechend hohen Entwurfskosten und eine hohen Risikos bezüglich der Funktionsfähigkeit gegenüber. Derartige Schaltungen werden also daher nur dann eingesetzt, wenn es aufgrund hoher Stückzahlen wirtschaftlich sinnvoll (Beispiel: Pentium-Prozessor) oder wegen spezieller Eigenschaften technisch erforderlich ist (Beispiel: Chip im Auge von Blinden).

3.3 Gate Arrays

Weniger Entwurfsflexibilität gibt es bei den sog. *Gate Arrays*. In diesem Fall werden auf Chips Transistoren unabhängig von der konkreten Anwendung in Reihen angeordnet und meist auch unabhängig von der Anwendung in den unteren Ebenen bereits vorfabriziert. Zwischen den Transistoren gibt es separate Flächen für die Verdrahtung. Die Abb. 3.3 [RC89] zeigt zwei Alternativen für die Anordnung von Transistorzellen und Verdrahtungsbereichen, nämlich das sog. Streifen-Layout und das Insel-Layout.

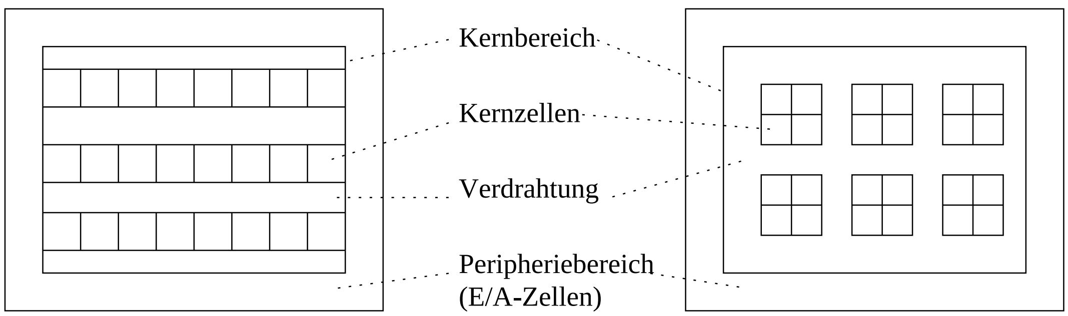


Abbildung 3.3: Layout von Gate Arrays

Die Verdrahtung erfolgt in den oberen Schaltkreis-Ebenen in Aluminium abhängig von der konkreten Anwendung. In den unteren Schaltkreis-Ebenen, in denen die Transistoren realisiert sind, werden in den Verdrahtungsflächen zusätzlich auch nutzbare Leitungssegmente vorfabriziert, meist in Polysilizium.

Ein Problem bei Gate Arrays ist die vorgegebene Breite der Verdrahtungskanäle. Bei zu knapp bemessenen Breiten besteht ein hohes Risiko, dass die Zellen nicht voll genutzt werden können, weil die Leitungen den Zellen nicht zugeführt werden können. Bei reichlich bemessenen Verdrahtungskanälen wird für viele Anwendungen Fläche vergeudet.

3.4 Sea of Gates

Um von der starren Struktur von Gate Arrays herunterzukommen, sind sog. *sea of gates*-Strukturen vorgeschlagen worden. Bei dieser Technik werden Chips gleichmäßig und vollständig mit Transistoren bedeckt; für die Verdrahtung bleibt keine separate Fläche frei. Die Verdrahtung kann dann nur noch in den Metallebenen über den Transistoren erfolgen.

3.5 Standardzellen-Technik

Die Standardzellentechnik basiert auf Zellen gleicher Höhe und variabler Breite. Diese Zellen liegen vorentworfen in einer Zellenbibliothek vor. Diese werden anwendungsabhängig ausgewählt und in Reihen plaziert. Der Abstand der Zellreihen ist dabei variabel. Es kann also jeweils für die Verdrahtungskanäle die Breite vorgesehen werden, die auch tatsächlich benötigt wird. Standardzellen sind daher kompakter als Gate Arrays.

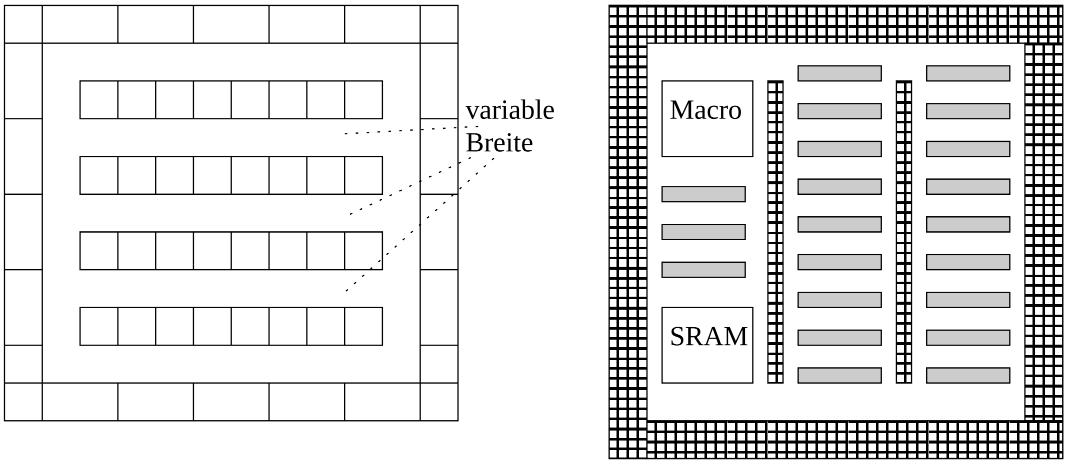


Abbildung 3.4: Standardzellen

Standardzellen werden gelegentlich mit den nachfolgend beschriebenen Makrozellen kombiniert.

3.6 Makrozellen

Makrozellen sind komplexe Zellen unterschiedlicher Größe. Im Gegensatz zu Standardzellen werden sie nicht in Reihen angeordnet. Standardzellen realisieren beispielsweise RAM- oder ROM-Blöcke, eventuell auch arithmetisch/logische Einheiten (ALUs). Mit Makrozellen wird ein kompakteres Layout erzielt als mit Standardzellen.

3.7 Vergleich der Technologien

Die Tabelle 3.1 [Pos89] stellt die Eigenschaften der verschiedenen Zieltechnologien noch einmal anhand der Daten von 1989 gegenüber (natürlich müssten alle diese Zahlen auf die Werte von heute skaliert werden).

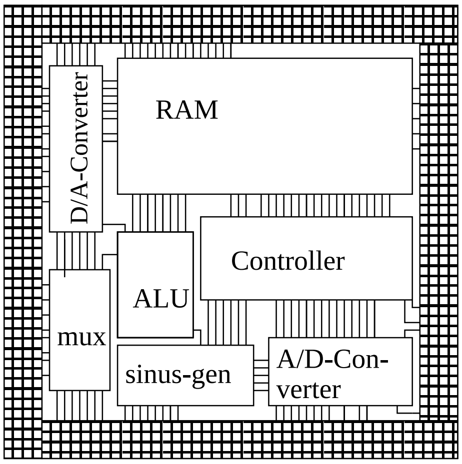


Abbildung 3.5: Makrozellen

	Gate Arrays	Standardzellen	Makrozellen	ASIC (Handlayout)
Gatterdichte [Gatter/mm^2]	100-200	100-200	300-500	
Entwurfsdauer [Wochen]	4-8	6-24	6-24	
Überführung in neue Technologie	direkt	1 Jahr	1-2 Jahre	
Vorteile	schneller Entwurf	einfacher Entwurf	gute Flächen-ausnutzung	hohe Ausbeute
Mindeststückzahl	> 500	> 10.000	> 10.000	> 100.000

Tabelle 3.1: Vergleich von Zieltechnologien

3.8 Cores

3.8.1 Allgemeine Cores

Aufgrund der wachsenden Integrationsdichte und den sich immer weiter verkürzenden Zyklen bis zur Markteinführung, insbesondere im *consumer*-Bereich, entstand die Notwendigkeit, mit größeren Blöcken zu arbeiten. Seit ca. 1996 sind dies die sog. *cores*. Cores sind vorentworfene Blöcke, die in ihrer Komplexität über den Makrozellen liegen. Zu den Cores gehören *peripheral cores*, wie etwa Ethernet-Controller oder PCI-Interfaces

Die Fa. LSI Logic beispielsweise bietet an:

MPEG-, JPEG-, Ethernet-, ATM-, Viterbi-, PCI-, Fibre-Channel- und VGA-Cores.

3.8.2 Prozessorkerne

Zu den Cores gehören auch vollständige Prozessoren (*processor cores*).

3.8.2.1 Allgemeine Prozessoren

Als Beispiel betrachten wir den Prozessorkern MiniRISC CW4001, der von der Fa. LSI Logic verfügbar ist. Dieser besitzt die folgenden Eigenschaften:

- Befehlssatzkompatibilität mit dem Befehlssatz der MIPS R4000 (MIPS-II-Befehlssatz)
- 4 mm^2 groß bei Fertigung in 0.5 μ-CMOS-Technologie
- die Leistungsaunahme liegt bei nur 2mW/MHz
- 60 MHz Takt, 60 MIPS Spitze, 45 MIPS andauernd
- 3-stufiges Fliessband
- Bus mit gemeinsamen Leitungen für Daten/Adressen (kompakt, leistungsarm, aber nicht schnell)
- verschiedene Blöcke konfigurierbar (Cache usw.)

- wird mit VHDL/Verilog-Modellen geliefert.
- MMU verfügbar
- Testbarkeit über 'Full-Scan' realisiert

Der neuere Prozessor TinyRISC TR 4101 benötigt ohne Multiplizierer/Dividierer-Einheit nur noch 2 mm^2. Zur Reduktion der Codedichte bietet er neben 32-Bit-Befehlen auch 16-Bit-Befehle-

3.8.2.2 DSP-Cores

Im Bereich der eingebetteten Systeme sind die Anforderungen muss besonderer Wert auf die Effizienz der benutzten Hardware gelegt werden. Neben einer hohen Codedichte sind eine geringe Leistungsaufnahme und eine an die Verarbeitungsaufgabe angepasste Architektur sehr wichtig.

Eine wichtige Teilaufgabe von eingebetteten Systemen besteht in der digitalen Signalverarbeitung (DSP). Für diese Aufgaben sind spezielle, DSP-Aufgaben effizient verarbeitende Prozessoren entwickelt worden. DSP-Prozessoren besitzen die folgenden spezifischen Eigenschaften:

- *saturating arithmetic:*
  Bei DSP-Aufgaben treten gelegentlich Überschreitungen des darstellbaren Zahlenbereichs auf. Bei Standard-Arithmetik (siehe Anfang des Kapitels 2) werden in diesem Fall die zurückgegebenen Bitvektoren nicht auf spezielle Werte gesetzt (außer bei der Gleitkomma-Arithmetik). Bei DSP-Algorithmen ist es dagegen häufig am besten, wenn ein der größten bzw. der kleinsten Zahl entsprechender Bitvektor abgeliefert wird, der dann z.B. die größte Helligkeit auf dem Bildschirm oder die größte Lautstärke bezeichnet. Auf eine solche Arithmetik kann bei den meisten DSP-Prozessoren umgeschaltet werden.
- *spezielle Adressierungsarten:*
  Aufgrund der Anwendung in üblichen DSP-Algorithmen sehen DSP-Prozessoren häufig spezielle Adressierungsarten vor. Die Modulo-Adressierung beispielsweise erlaubt die effiziente Realisierung von verzögerten Signalen mit Hilfe von Ringpuffern.
- *Eingeschränkte Parallelität:*
  Die Rechenwerke der meisten DSP-Prozessoren erlauben es, in einem Takt Zuweisungen zu mehreren Registern gleichzeitig auszuführen. Diese Prozessoren stellen diese Form der begrenzten Parallelität dann meist auch an der Befehlsschnittstelle zur Verfügung. Gängig sind z.B. *parallel moves* genannte Befehle, die gleichzeitig eine Arithmetik-Operation und einen Datentransport veranlassen.
- *Heterogene Registersätze*
- *multiply/accumulate-Befehle*
- *Realtime-Fähigkeit:*
  Es ist wichtig, die maximale Laufzeit eines Programms möglichst exakt und nicht nur im Mittel angeben zu können. Deshalb wird z.B. vielfach auf Caches, die eine datenabhängige Laufzeit bewirken würden, verzichtet.

DSP-Prozessoren bieten v.a. für DSP-Applikationen mehr Rechenleistung pro Watt Leistungsverbrauch, weshalb sie v.a. in portablen Geräten vielfältig Einsatz finden.

3.9 Multi-Chip-Module (MCMs)

Multi-Chip-Module (MCMs) dienen der Verbindung mehrerer Chips auf möglichst kompakte Weise. Es gibt dazu die folgenden Varianten [She98, IEE93]:

MCM-L Dies ist eine fortgeschrittene Platinen-Technologie, in der Chips unverpackt auf der Platine befestigt werden. Es ist eine Standard-Technik mit geringem Risiko. Sie ist v.a. für niedrige Verbindungsdichten geeignet. Bei wenigen Lagen sind auch die Kosten gering.

MCM-C Bei dieser Technik werden Glas- bzw. Keramik-Träger benutzt. Bis zu 60 Ebenen sind möglich (IBM-Großrechner). Die Träger besitzen eine gute thermische Leitfähigkeit und eine geringe Wärmeausdehnung.

MCM-D Dies ist eine Erweiterung der klassischen VLSI-Technologie, bei der Silizium als Trägermaterial benutzt wird. Es sind mehrere Metallebenen möglich, bei Isolation durch Material einer niedrigen Dielektrizitätskonstanten.

PMCM In diesem Fall werden vorgefertigte MCMs benutzt, bei denen über *anti fuses* Verbindungen herstellbar sind. Es gibt ähnliche Flexibilitätsvorteile wie bei FPGAs (s.u.).

Es gibt verschiedene Befestigungsmöglichkeiten der Chips:

1. *Standard wire bonding*: In diesem Fall werden die Chips mit dem Träger wie sonst innerhalb von Chips mittels Golddrähten verbunden.
2. *Tape automatted bonding*: Bei dieser Technik wird passend geätzte Goldfolie aufgepresst.
3. *Flip chip bonding*: In diesem Fall werden die Chips mit der Schaltungsseite "nach unten" über winzige Lötzinnperlen mit dem Trägermaterial verbunden. Auf diese Weise ist die höchste Packungsdichte möglich.

MCMs besitzen die folgenden Vorteile:

1. MCMs überwinden die Probleme der Wafer-Scale Integration (WSI), wie z.B. schlechte Testbarkeit, hohe Kosten, geringe Ebenenanzahl und thermische Probleme.
2. Die Verdrahtung bildet einen günstigen Kompromiss zwischen der Verdrahtung innerhalb eines Chips bzw. Gehäuses und innerhalb einer Platine.
3. Aufgrund des Fortfalls der Gehäuse kann eine hohe Packungsdichte erzielt werden.
4. Es kann für verschiedene Schaltungskomponenten jeweils die optimale Technologie benutzt werden, also z.B. für Prozessoren und Speicher der jeweils optimale Prozess. Dies ist wichtig, weil Speicher, die zusammen mit Logikkomponenten auf einem Chip integriert werden, weniger kompakt sind als Speicher auf Speicherchips. Der Grund dafür sind Störströme durch die Logikomponenten, welche eine besondere Abschirmung der mit kleinen Spannungen arbeitenden Speicher durch speziell dotierte Wannen erfordern.

   Dies mag mit dazu geführt haben, dass beispielsweise Pentium-Prozessoren und Cache-Speicher als separate Chips in einem MCM-Gehäuse montiert werden.
5. Aufgrund kleinerer Kapazitäten steigt im Verhältnis zur Integration auf einer Platine die Arbeitsgeschwindigkeit.

   Tabelle 3.2 und Abb. 3.6 zeigen hierfür ein Beispiel der Fa. AT&T.

	Platine	MCM I/O
Größe [cm^2]	92	8
Leitungslänge [cm]	3.200	600
Takt [MHz]	40	50
Leistung [W]	5	2,1

Tabelle 3.2: Verbesserung von Kenndaten ohne Neuentwurf von Chips

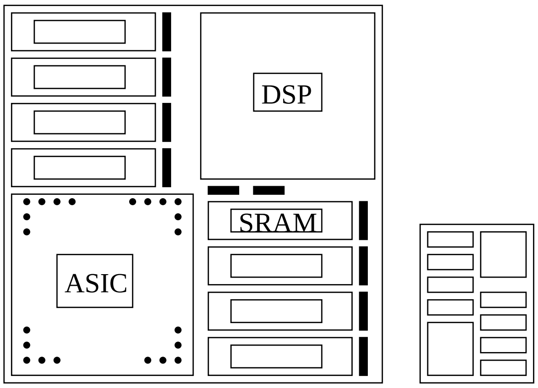


Abbildung 3.6: Flächenvergleich zwischen Platinen- und MCM-Entwurf

6. Bei Neuentwurf der Chips für den Einsatz in MCMs können noch weitere Veränderungen vorgenommen werden:
   - Wegen der großen Kapazitäten auf Platinen wird die Anstiegsgeschwindigkeit der Signale (die sog. *slew-rate*) von Standard-ICs zur Vermeidung hoher Stromimpulse begrenzt. Diese Begrenzung kann entfallen.
   - Da die Pin-Anzahl insbesondere beim *flip chip bonding* sehr groß ist, kann mit differentiellen Signalen und niedrigeren Spannungspegeln gearbeitet werden. Dadurch wird die Leistungsaufnahme reduziert.
   - Der Überspannungsschutz der Eingänge kann fortgelassen werden. Auch dadurch werden die Kapazitäten und die Leistungsaufnahme reduziert.

   Tabelle 3.3 zeigt die resultierende Unterschiede zwischen einem normalen CMOS-Puffer und einem MCM-Puffer.

	CMOS	MCM I/O
Anstiegszeit, simuliert [ns]	9	3,4
Abfallzeit, simuliert [ns]	6	3,7
Max. Freq., simuliert [MHz]	80	250
Max. Freq., gemessen [MHz]	140	400
Leistungsaufnahme [mW]	16.3	3,8

Tabelle 3.3: Vergleich konventieller CMOS- und differentieller MCM-Puffer

Ein Nachteil von MCMs besteht in den verhältnismäßig hohen Kosten. Um diese zu drücken, müsste eine Anwendung in einem Massenmarkt gefunden werden, die wegen der großen Stückzahlen die Preise drückt. Die Verteilung der Absatzzahlen im Konsummarkt (siehe Abb. 3.7) zeigt leider, dass MCMs in Marktbereichen mit großen Stückzahlen bislang zum Einsatz kamen.

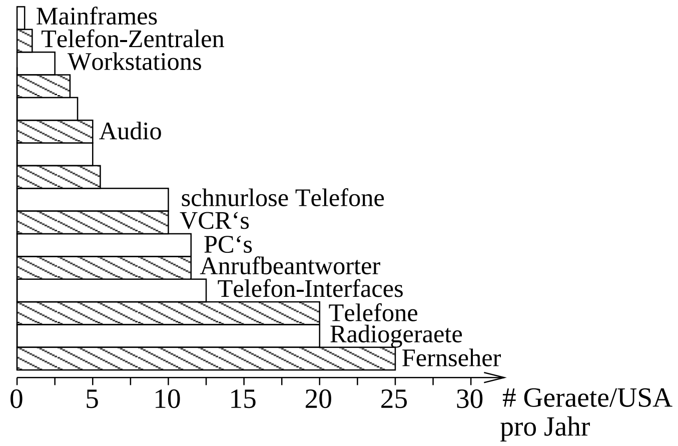


Abbildung 3.7: Verteilung des Konsummarktes

Es gibt daher die Idee, künftig MCMs auch in Massengeräten einzusetzen. Interessant wäre hier beispielsweise der Einsatz bei gemischt digital/analogen Geräten, da sich Analogschaltungen nur mit Zusatzaufwand gemeinsam mit Digitalschaltungen auf einem Chip integrieren lassen.

3.10 (Re-) konfigurierbare Logikbausteine

3.10.1 Field programmable gate arrays (FPGAs)

Großer Beliebtheit erfreuen sich in jüngster Zeit Variationen von Gate Arrays, welche noch im Gehäuse programmierbar sind, die sog. *field programmable gate arrays* (FPGAs).

FPGAs bestehen aus konfigurierbaren logischen Blöcken (*configurable logic blocks, CLBs*), die über programmierbare Leitungen miteinander verbunden werden können. Hierbei gibt es verschiedene Alternativen:

- Multiplexer-basierte CLBs (siehe Abb. 3.8).

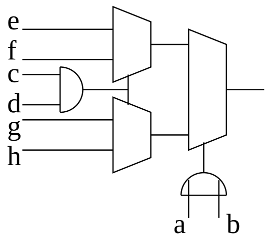


Abbildung 3.8: Multiplexer-basierter CLB der Fa. ACTEL

  Die verschiedenen Eingänge können mit festen Signalwerten konfiguriert werden.

- SRAM-basiert
  Ein Beispiel dafür sind die CLBs der Fa. Xilinx, in Abb. 3.9 gezeigt anhand der CLB-Struktur der FPGA-Serie XC4000.

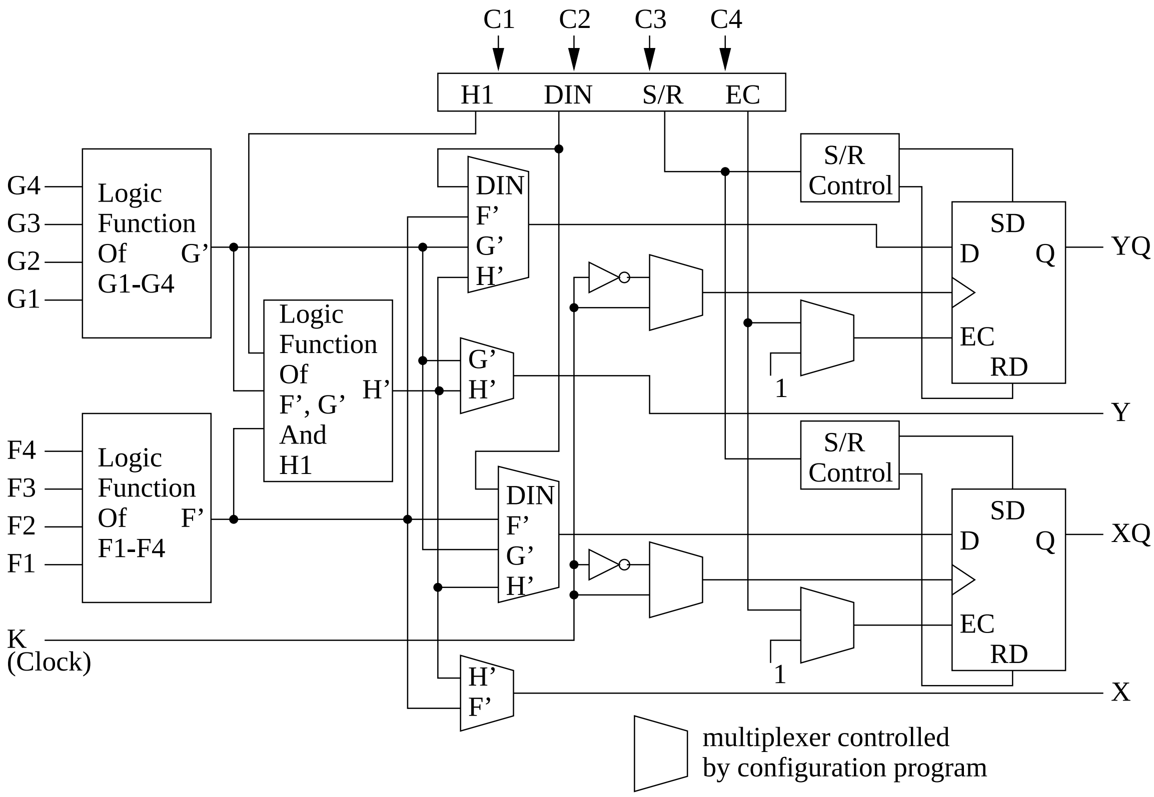
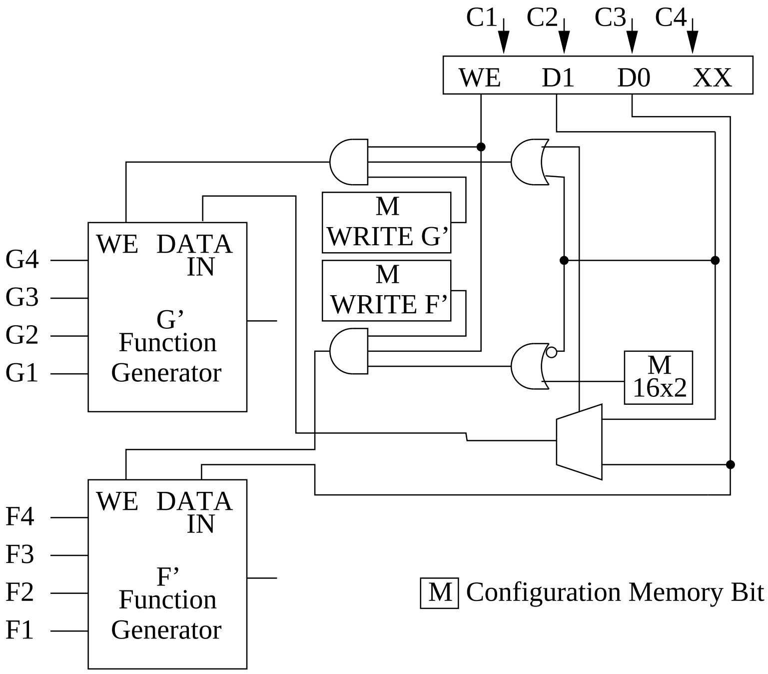


Abbildung 3.9: Vereinfachtes Blockdiagramm des XC4000 CLB

Diese CLBs einthalten jeweils zwei Speicher von 16 Bit, welche alle Booleschen Funktionen von 4 Variablen realisieren können. Weiterhin gibt es noch einen dritten Speicherblock, mit Hilfe dessen auch einige Funktionen von 5 Variablen direkt realisiert werden können. Diese Speicher werden beim Start des FPGAs oder auch später geladen. Die Speicher können auch zur Realisierung von kleinen Speicher benutzt werden (siehe Abb. 3.10). Über eine schnelle Carry-Logik kann ein CLB auch zur Addition von zwei Bit verwendet werden (siehe Abb. 3.11).



Vorlage ist rechts abgechnitten... XX nicht lesbar !

Abbildung 3.10: Benutzung von CLBs als Speicher

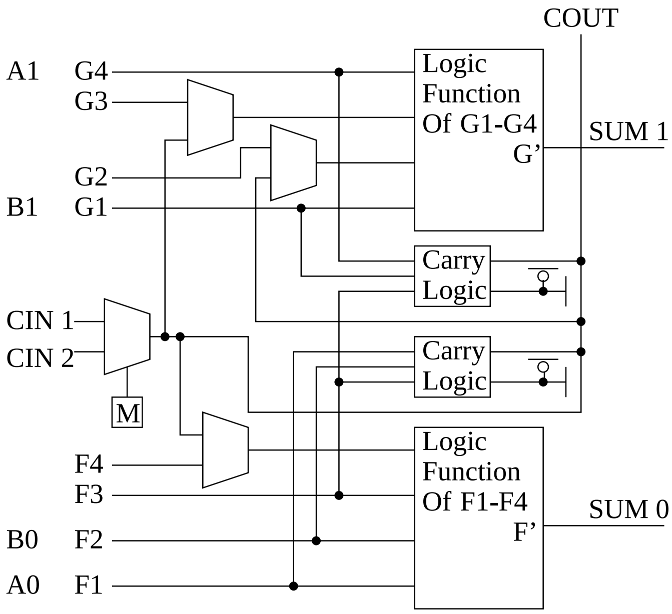


Abbildung 3.11: Carry-Logik

- EPROM-basierte CLBs.

Die Verbindungen der Bausteine untereinander können über zwei Methoden realisiert werden:

- Mittels *anti fuses*:
  In diesem Fall wird die Isolierung zwischen zwei eng benachbarten Leitungen durch einen kurzen Überspannungsimpuls geschmolzen.
  Diese Vorgang ist nicht reversibel; er führt aber dafür zu einem schnellen Pfad.
- Mittels RAM-gesteuerter MOS-Schalter:

Dieser Vorgang ist reversibel (und daher an Universitäten aus Kostengründen beliebt). Er führt allerdings zu flüchtigen und nicht sehr schnellen Verbindungen.

Speziell bei der Serie XC400ist zwischen mehreren Verbindungsformen zu unterscheiden, wie in den Abbildungen 3.12 bis 3.15 zu sehen.

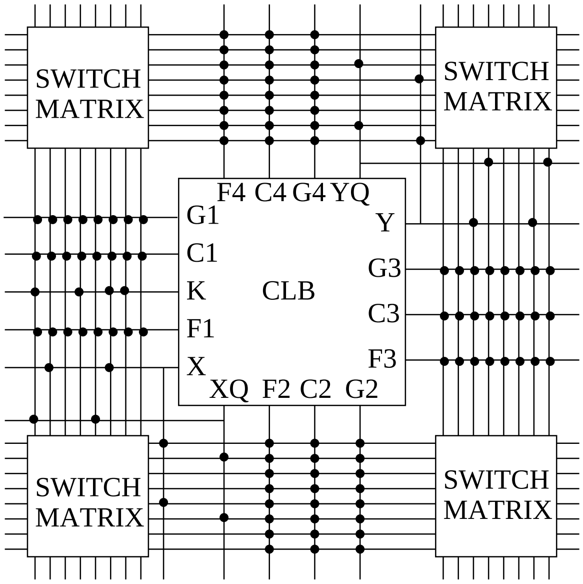


Abbildung 3.12: Typische Verbindung zu benachbarten einfach-langen Leitungen

Die verschiedenen Signale der CLBs stehen überwiegend nur an einer der Seiten zur Verfügung und können dort mit Leitungen programmierbar verschaltet werden.

Über kurze Entfernungen können die einfach-langen Leitungen benutzt werden, die zwischen allen CLBs über eine Schaltmatrix miteinander verbunden werden können (siehe Abb. 3.13).

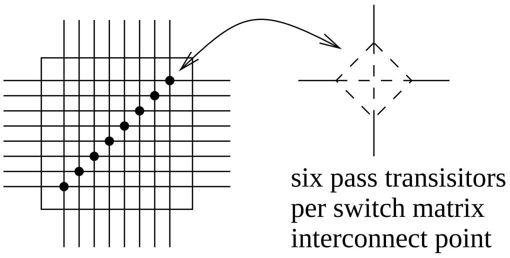


Abbildung 3.13: Schaltmatrix an den Kreuzungspunkten

Für etwas größere Entfernungen gibt es Leitungen, die nur noch an jedem zweiten CLB angeschlossen werden können (siehe Abb. 3.14).

Schließlich gibt es noch globale *longlines* (siehe Abb. 3.15).

In der Serie XC4000 sind FPGAs mit 64 bis 1.024 CLBs erhältlich. Eine typische Verschaltung zur Konfiguration mit Daten aus einem EPROM zeigt die Abb. 3.16.

Die Konfigurationsdaten werden in diesem Fall byteseriell eingelesen (siehe Abb. 3.17).

Die XILINX-Serie XC6000 erlaubt ein dynamisches Rekonfigurieren eines Teils der Zellen[1].

[1] XILINX scheint den Vertrieb dieser Serie wieder einzustellen.

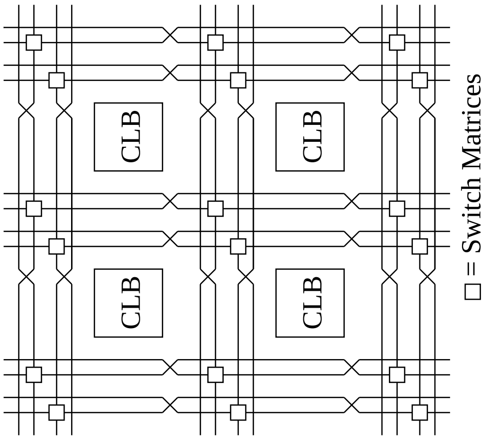


Abbildung 3.14: Doppelt-lange Leitungen

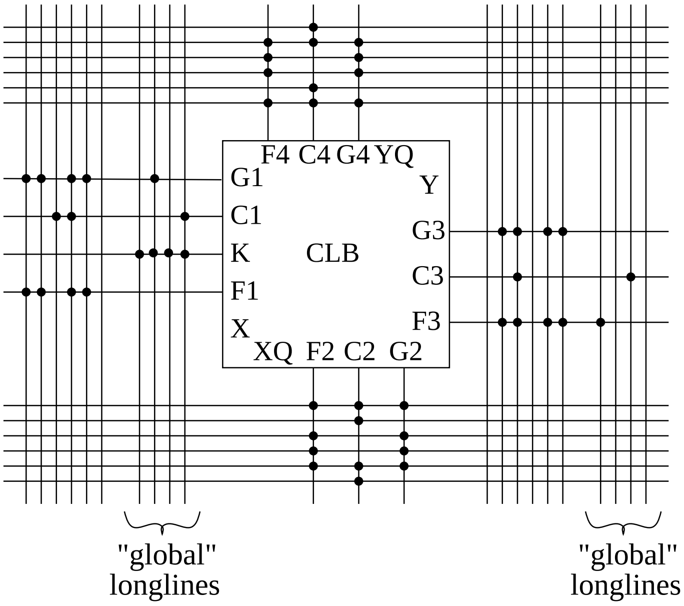


Abbildung 3.15: Longlines

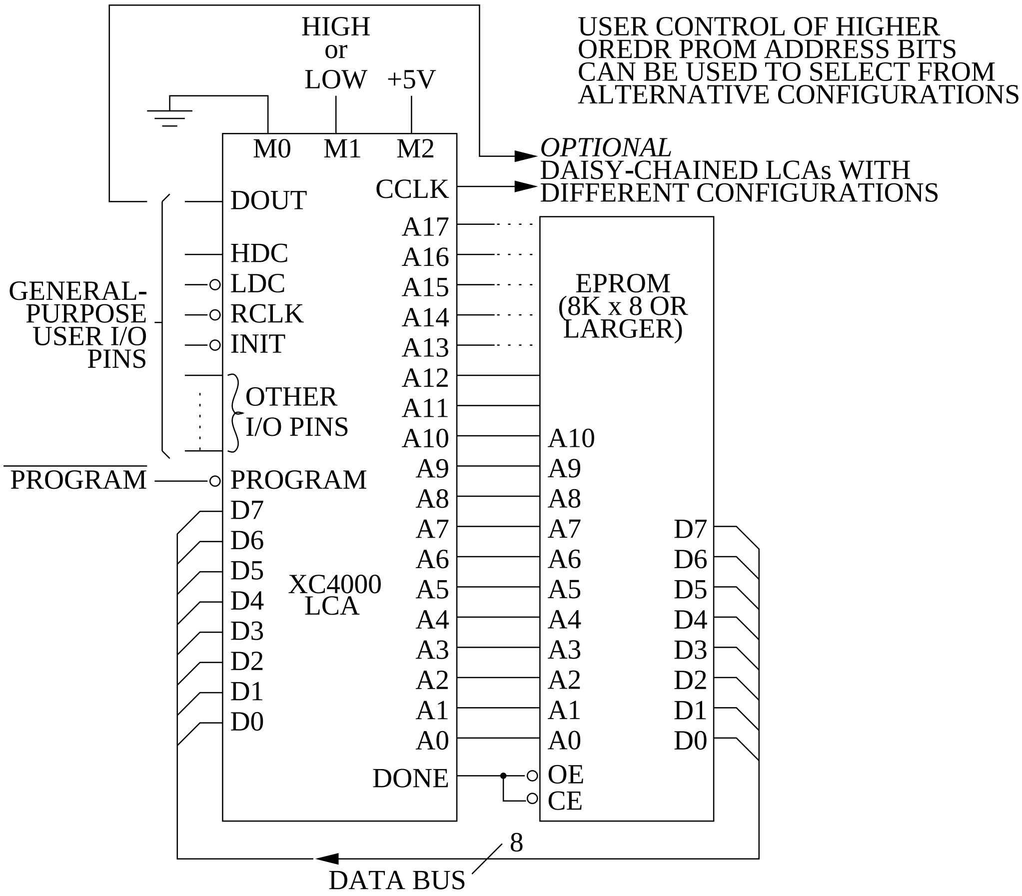


Abbildung 3.16: Laden von Konfigurationsdaten aus einem EPROM

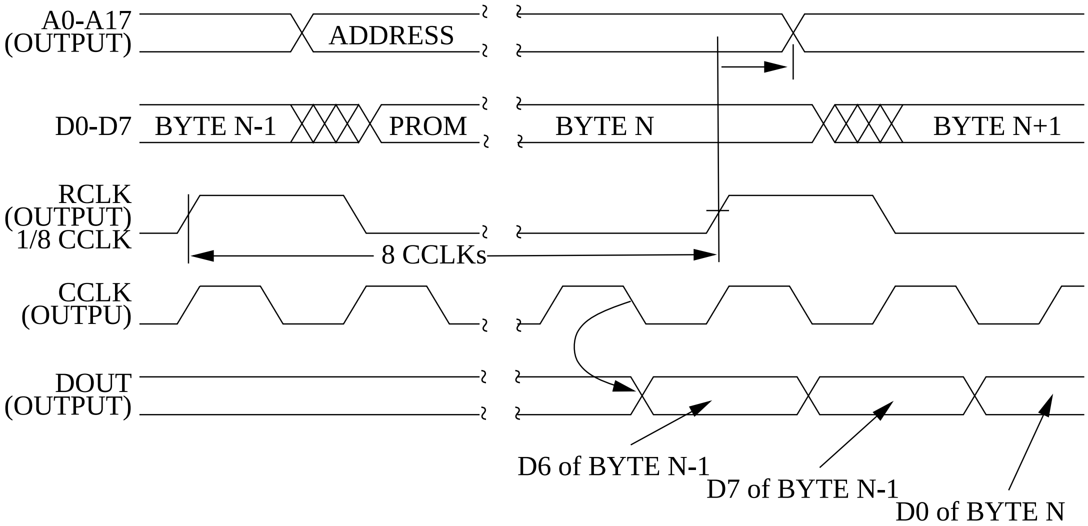


Abbildung 3.17: Zeitlicher Ablauf des Ladevorgangs

3.10.2 PALs und PLDs

PALs sind programmierbare Bausteine, bei denen im Unterschied zu PROMs und PLAs nur die UND-Ebene programmierbar ist (siehe Abb. 3.19 bis 3.20).

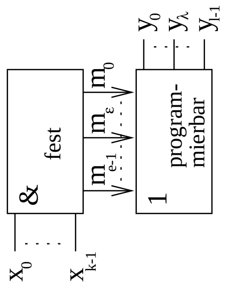


Abbildung 3.18: PROM (feste UND-Struktur)

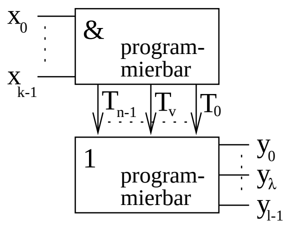


Abbildung 3.19: PLA (programmierbare UND- und ODER-Strukturen

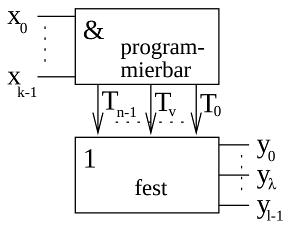


Abbildung 3.20: PAL (programmierbare UND-Struktur, feste ODER-Struktur

Abb. 3.21 zeigt, wie eine Schaltfunktion mittels eines PLAs realisiert werden kann.

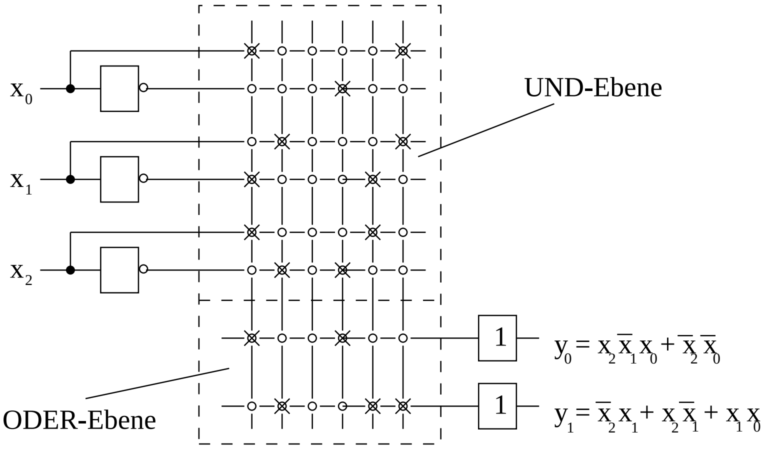


Abbildung 3.21: Beispiel einer Realisierung einer Schaltfunktion mittels eines PLAs

Im Vergleich dazu zeigt Abb. 3.22 eine Beschreibung eines PALs und Abb. 3.23 die Verwendung eines solchen PALs zur Realisierung von Schaltfunktionen.

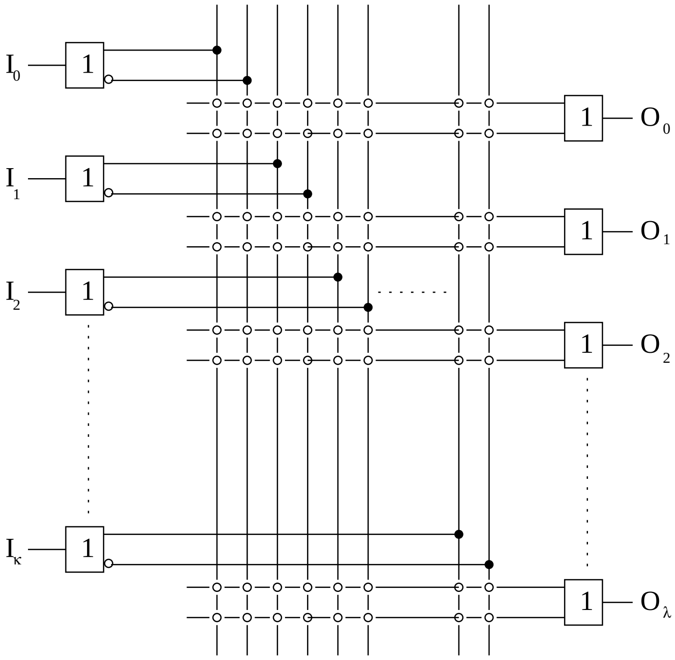


Abbildung 3.22: Einfache PAL-Architektur

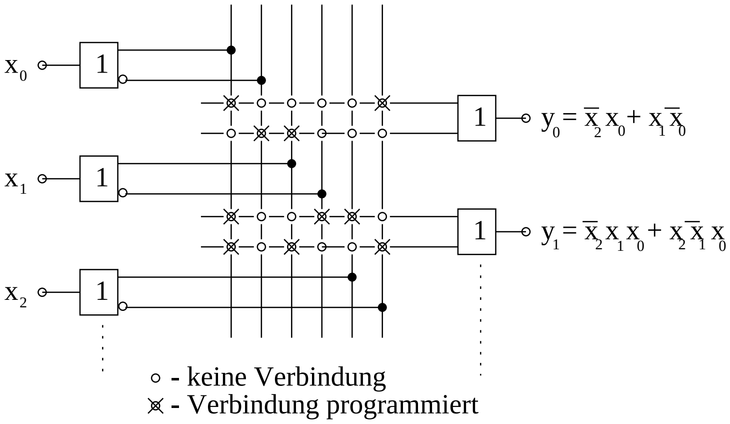


Abbildung 3.23: Beispiel der Realisierung zweier Schaltfunktionen mittels eines PALs

Für die Programmierung von PALs und anderen Bausteinen stehen die in den Abbildungen 3.24 bis 3.28 beschriebenen Techniken zur Verfügung.

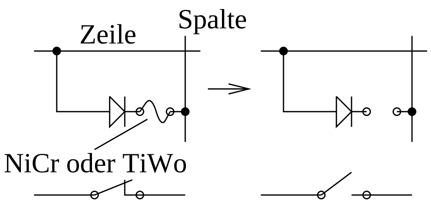


Abbildung 3.24: Verbindungsprogrammierung mittels Schmelzsicherung

Abb. 3.29 zeigt die Komplexität eines Blockes eines handelsüblichen PALs.

In der Serie MACH 435 der Fa. AMD sind beispielsweise 8 von diesen Blöcken über Schaltmatrizen miteinander verschaltbar.

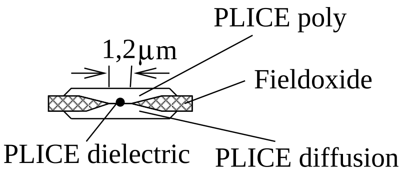


Abbildung 3.25: Verbindungsprogrammierung mittels Anti-Fuse

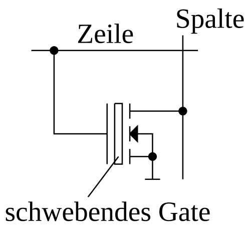


Abbildung 3.26: Verbindungsprogrammierung mittels EPROM-Transistor

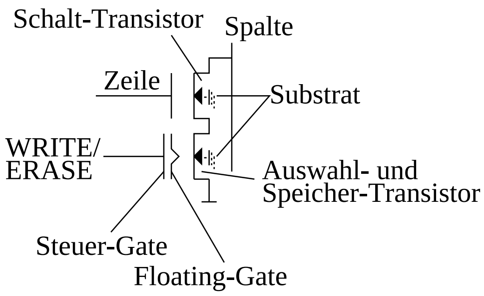


Abbildung 3.27: Verbindungsprogrammierung mittels E^2CMOS-Technologie

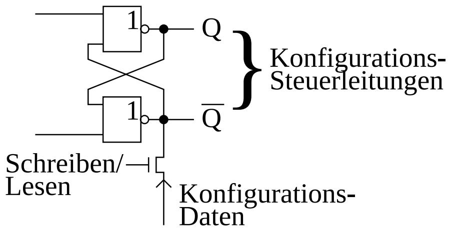


Abbildung 3.28: Verbindungsprogrammierung mittels SRAM-Zelle

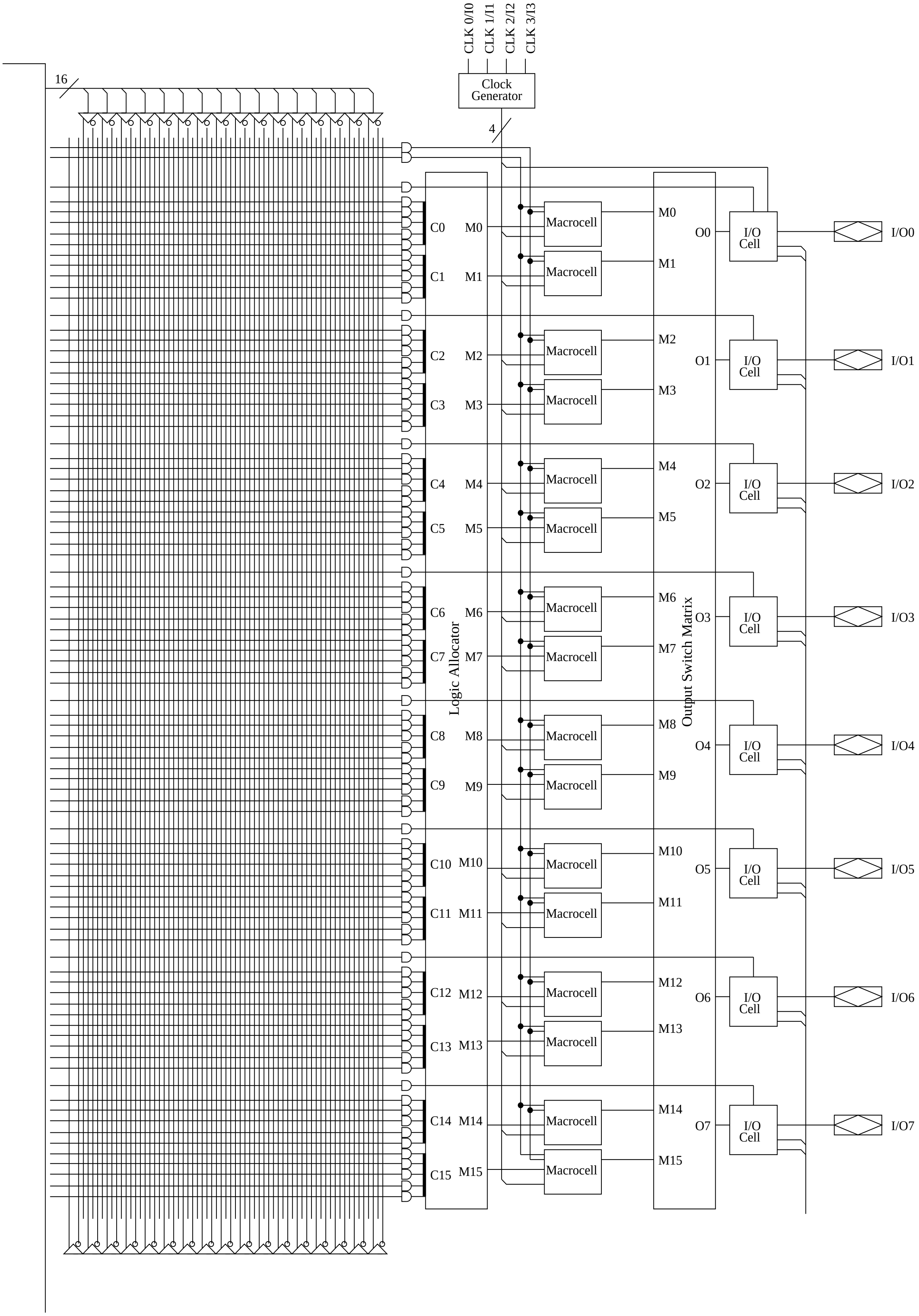


Abbildung 3.29: MACH435 PAL-Block

3.11 Prozessorcode

Falls die Leistungsanforderungen ausreichen, kann die Funktionalität auch vollständig durch Programmierung eines Prozessors realisiert werden. Von der Klassifikation her müssen die Programme in Kombination mit den oben beschriebenen Prozessoren benutzt werden.

Werkzeuge wie StateMate erlauben die Übersetzung der Spezifikation in C-Code. Da StateMate bislang nicht zur Verfügung steht, kann an dieser Stelle keine Aussage dazu gemacht werden.

3.12 Erwartetete Entwicklung der Fertigungstechnologie

Für die zukünftige Planung ist es sehr interessant, die erwartete Entwicklung der Halbleitertechnologie zu kennen. Zu diesem Zweck werden seit einigen Jahren sog. *Roadmaps* von der amerikanischen *semiconductor industries association* (SIA) erstellt. Z.Z. (1997) ist als neueste Fassung die *Roadmap* von 1994 verfügbar [Ass94]. Die Tabellen 3.4 bis 3.9 enthalten eine Zusammenfassung der Aussagen dieser *Roadmap* (©SIA).

Year of the first DRAM Shipment	1995	1998	2001	2004	2007	2010	Driver
Minimum Feature size [μm]	0.35	0.25	0.18	0.13	0.10	0.07	
Memory							D
Bits/Chip(DRAM/Flash)	64M	256M	1G	4G	16G	64G	
Cost/Bitvolume [millicents]	0.017	0.007	0.003	0.001	0.0005	0.0002	

Tabelle 3.4: Wachstum der DRAM-Technologie

Alle Tabellen zeigen zunächst in den Kopfzeilen die abnehmenden minimalen Abmessungen z.B. von Transistor-Gates. In der Spalte *Driver* wird gezeigt, welche Chipklasse die Parameter der jeweiligen Zeilen weiter voran treibt (A=ASIC, D=DRAM, L=Logik).

Für die Größe von DRAMs sagt Tabelle 3.4 ein weiteres Wachstum um den Faktor 4 alle 3 Jahre voraus. Im Jahre 2010 wird die beachtliche Größe von 1 GBit pro Chip erwartet! Dabei sollen die Kosten pro Bit noch einmal dramatisch fallen.

Gleichzeitig soll die Anzahl der Transistoren pro cm^2 auf 90 Millionen steigen (siehe Tabelle 3.5 und Abb. 3.30)[2].

Year of the first DRAM Shipment	1995	1998	2001	2004	2007	2010	Driver
Minimum Feature size [μm]	0.35	0.25	0.18	0.13	0.10	0.07	
Logic (High-volume:Microprocessor)							L(μP)
Logic Transistors/cm^2 (packed)	4M	7M	13M	25M	59M	90M	
Bits/cm^2 (cache SRAM)	2M	6M	20M	50M	100M	300M	
Logic (Low-Volume:ASIC)							L (A)
Transistors/cm^2 (auto layout)	2M	4M	7M	12M	25M	40M	
Non-recurring engineering cost/transistor [millicents]	0.3	0.1	0.05	0.03	0.02	0.01	

Tabelle 3.5: Anzahl der Transistoren

Unter *non-recurring engineering costs* versteht man die einmaligen Kosten bis zum Anlaufen der Fertigung.

Tabelle 3.6 zeigt, wie die Anzahl der Anschlüsse, die Taktfrequenz, die Größe der Chips und die Anzahl der Verdrahtungsebenen wachsen sollen.

4000 Anschlüsse pro ASIC sind sicher eine enorm große Zahl. Das Gleiche gilt für die Taktfrequenz von 1,1 GHz, die Kantenlänge von ca. 3.7 cm pro Chip und die bis zu 8 Verdrahtungsebenen. Abb. 3.31 zeigt die Entwicklung der Taktfrequenzen noch einmal in übersichtlicher Form.

Die angegebenen Daten sind nur erreichbar, wenn die Anzahl von Fehlern pro Siliziumflächeneinheit weiter sinkt (siehe Tabelle 3.7).

Ein großes Problem wird die Leistungsaufnahme sein. Es wird erwartet, dass Mikroprozessoren bis zu 180 Watt aufnehmen werden[3] (siehe Tabelle 3.8).

Gleichzeit wird die Betriebsspannung absinken müssen, damit nicht noch höhere Werte erreicht werden (siehe Abb. 3.32).

[2]Nach der neueren SIA-Roadmap von 1997 sollen 2007 bereits 300 Millionen Transistoren erreicht sein.
[3]Dieder Wert kann sicher zu langen Diskussionen Anlass geben.

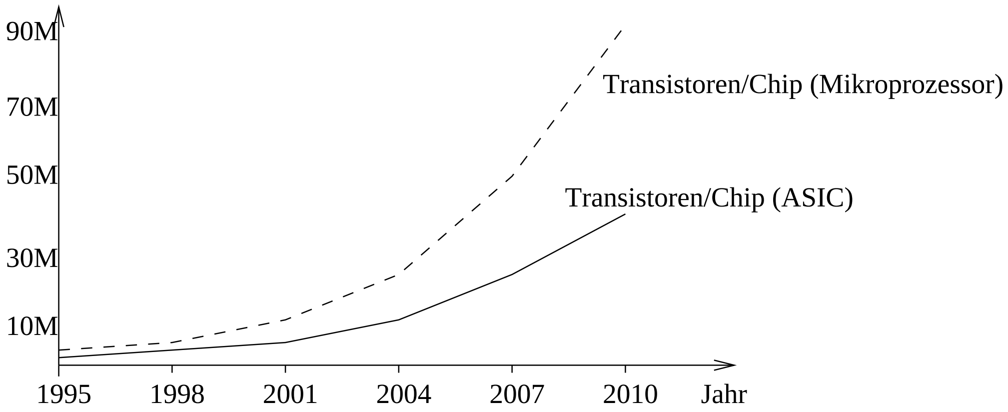


Abbildung 3.30: Anzahl der Transistoren pro cm^2

Year of the first DRAM Shipment Minimum Feature size [μm]	1995 0.35	1998 0.25	2001 0.18	2004 0.13	2007 0.10	2010 0.07	Driver
Number of Chip I/Os							
Chip-to-package(pads)high-performance	900	1350	2000	2600	3600	4800	L,A
Number of Package Pins/Balls							
Microprocessor/controller	512	512	512	512	800	1024	μP
ASIC (high-performance)	750	1100	1700	2200	3000	4000	A
Package cost [cents/pin]	1.4	1.3	1.1	1.0	0.9	0.8	A
Chip Frequency [MHz]							
On-chip clock, cost/performance	150	200	300	400	500	625	μP
On-chip clock, high-performance	300	450	600	800	1000	1100	
Chip-to-board speed, high-performance	150	200	250	300	375	475	L
Chip Size [mm^2]							
DRAM	190	280	420	640	960	1400	D
Microprocessor	250	300	360	430	520	620	μP
ASIC	450	660	750	900	1100	1400	A
Maximum Number Wiring Levels (logic)							
On-chip	4-5	5	5-6	6	6-7	7-8	μP

Tabelle 3.6: Overall Roadmap Technology Characteristics Major Markets

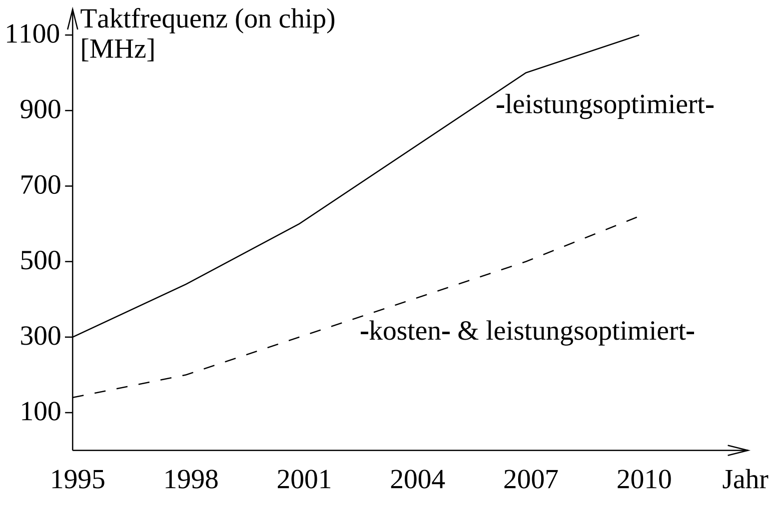


Abbildung 3.31: Taktfrequenz innerhalb der Chips

Der Anzahl der Transistoren pro Mikroprozessor wird auf bis zu 800 Millionen steigen (siehe Tabelle 3.9).

Es ist nicht ganz klar, wozu man diese Anzahl von Transistoren eigentlich verwenden will. Mehr als die heute

Year of the first DRAM Shipment Minimum Feature size [μm]	1995 0.35	1998 0.25	2001 0.18	2004 0.13	2007 0.10	2010 0.07	Driver
Electrical defect density [d/m^2]	240	160	140	120	100	25	A
Minimum mask count	18	20	20	22	22	24	L
Cycle time (days theoretical)	9	10	10	11	11	12	L
Maximum substrate diameter [mm]							
Bulk or epitaxial or SOI* wafer	200	200	300*	300	400*	400	D

Tabelle 3.7: Overall Roadmap Technology Characteristics: Fabrication

Year of the first DRAM Shipment Minimum Feature size [μm]	1995 0.35	1998 0.25	2001 0.18	2004 0.13	2007 0.10	2010 0.07	Driver
Power Supply Voltage [V]							
Desktop	3.3	2.5	1.8	1.5	1.2	0.9	μP
Battery	2.5	1.8-2.5	0.9-1.8	0.9	0.9	0.9	A
Maximum Power							
High-performance with							
heatsink [W]	80	100	120	140	160	180	μP
Logic without heatsink [W/cm^2]							
heatsink [W]	5	7	10	10	10	10	A
Battery [W]	2.5	2.5	3.0	3.5	4.0	4.5	L
Design and Test							
Volume Tester cost/pin [k$]	3.3	1.7	1.3	0.7	0.5	0.4	L
Number of test vectors (μP) [10^6]	16-32	16-32	16-32	8-16	4-8	4	L
Built-in self test/ design for testability	25	40	50	70	90	90+	L

Tabelle 3.8: Electrical Design and Test Metrics

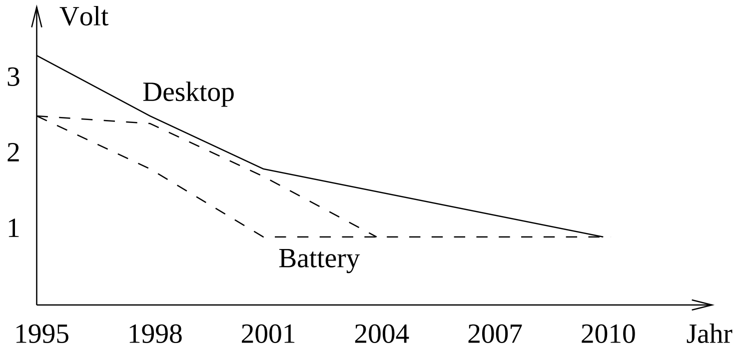


Abbildung 3.32: Betriebsspannung

Year of the first DRAM Shipment Minimum Feature (μm)	1995 0.35	1998 0.25	2001 0.18	2004 0.13	2007 0.10	2010 0.07
DRAM Cell [μm^2] (0.4X/gen)	1.50	0.60	0.24	0.096	0.038	0.015
SRAM Chip Size [mm^2] (1.5X/gen)	220	330	490	740	1100	1600
SRAM Cell [μm^2](0.4X/gen)	8	3.2	1.3	0.52	0.21	0.08
Microprocessor Transistor per Chip (2.3X/gen)	12M	28M	64M	150M	350M	800M
ASIC (gate per chip)	5M	14M	26M	50M	210M	430M

Tabelle 3.9: Principal Wafer Fabrication Characteristics

vorkommenden 4 Fließbänder mit max. 13 Pipelinestufen kann man kaum sinnvoll einsetzen. Viele der Transistoren werden wohl Caches realisieren.

Kapitel 4

Von der Spezifikation zur Implementierung

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 4.1.

	Domäne		
Ebene	Verhalten	Struktur	Geometrie, Layout
System-Ebene (PMS-Ebene)	Kommunizierende Prozesse	Prozessoren (P), Speicher ("memories", M), Schalter (S)	geometrisch/physikalische Information über ein System (Gehäuse)
Algorithmische Ebene (Befehlssatz)	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
Schalter-Ebene	Gleichungen für Ausgaben als Funktion der Eingaben	Schalter,Abschwächer, Verbindungen	"Sticks" (symbolisches Layout)
Schaltkreis-Ebene	Netzwerk-Gleichungen	Kondensatoren, Transistoren, Stromquellen, usw.	Schaltkreis-Layout
Bauelement-Ebene	Gleichungen für Anschluß-Ströme und -Spannungen	Gates, Kanäle, Source-Anschlüsse	Bauelement-Layout (symbolisch oder detailliert)
Prozeß-Ebene	Diffusions-Verh., Ätz-Verhalten	chemische Elemente, Kristallgitter	Masken

Tabelle 4.1: Verschiedene 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 4.1). Die oberste Ebene ist die der Spezifikation, welche bei den heute üblichen Spezifikationstechniken vielfach aus nebenläufigen Prozessen besteht. 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 4.1 entspricht.

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. 4.1).

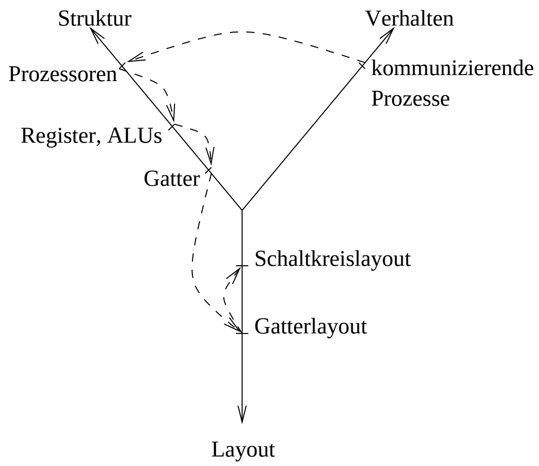


Abbildung 4.1: Y-Diagramm nach Gajski

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. 4.2).

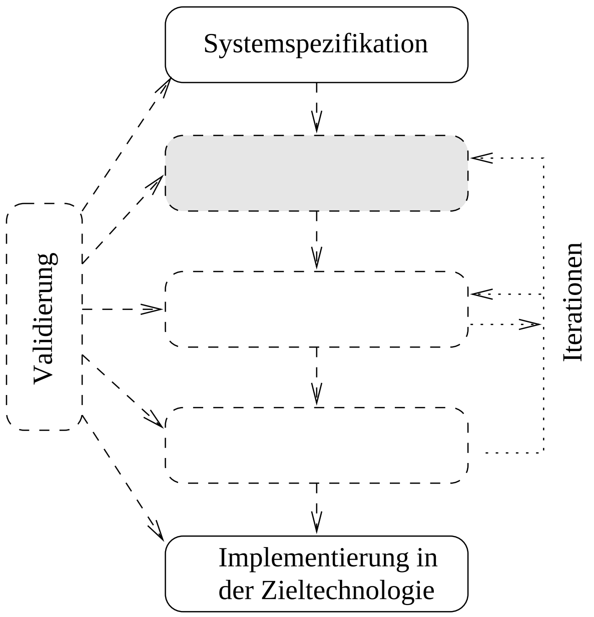


Abbildung 4.2: Erste Schritte zur Implementierung

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 [N.N]).

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 Folgenden mit den Grundlagen solcher Syntheseverfahren beschäftigen.

4.1 Hardware/Software-Codesign

4.1.1 Aufgabe

Im allgemeinen müssen wir dabei zunächst entscheiden, ob eine bestimmte Funktionalität in Hardware oder in Software zu realisieren ist. Bei konstanten Leistungsanforderungen wird dabei im Laufe der Jahre im mehr Funktionalität in Software realisiert, da dies flexibler ist (siehe Abb. 4.3).

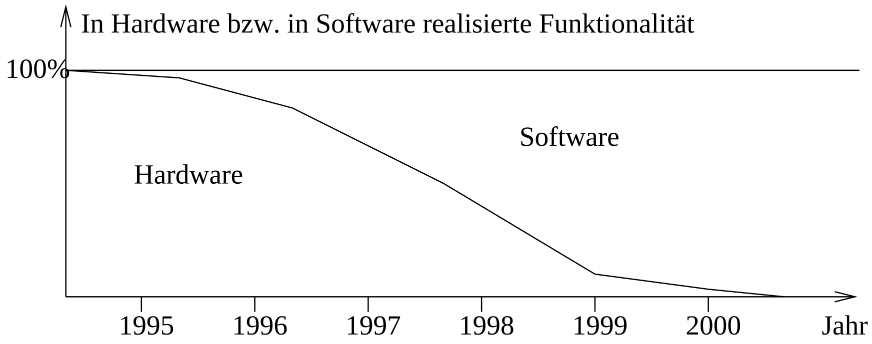


Abbildung 4.3:

In der Praxis gibt es jedoch ständig steigende Leistunsganforderungen (beispielsweise beim Übergang von MPEG-1 über MPEG-2 auf MPEG-4). Der gemischte Entwurf von Hardware und Software wird daher wohl für eine Reihe von Jahren weiter erforderlich sein.

Hierzu ist zunächst die Frage zu beantworten: welcher Anteil der Spezifikation soll in Hardware und welcher soll in Software realisiert werden? Diese Frage soll in Verfahren zur Hardware/Software-Partitionierung beantwortet werden. Nach einer solchen Partitionierung können Hardware- und Software-Realisierung dann separat erzeugt und zur Kontrolle anschließend gemeinsam simuliert werden (siehe Abb. 4.4).

Da die Realisierung nicht notwendigerweise alle Randbedingungen einhält, muss die Partitionierung mit anderen Vorgaben eventuell neu gestartet werden.

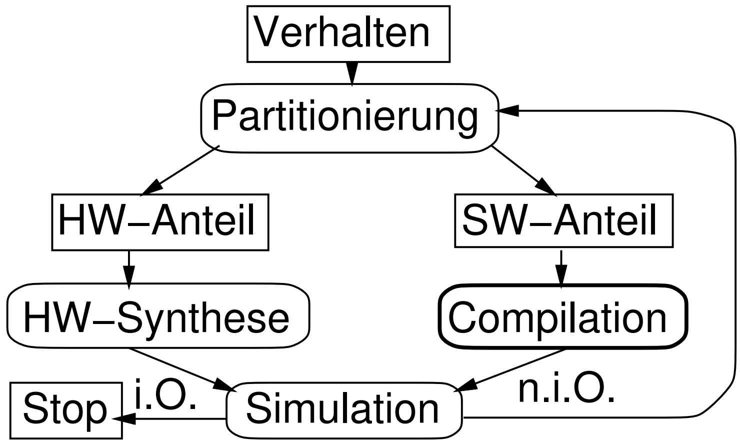


Abbildung 4.4:

4.1.2 COOL

Als Beispiel eines HW/SW-Codesign-Systems betrachten wir das Werkzeug COOL[1].

4.1.2.1 Introduction

Hardware/software codesign deals with the problem of designing embedded systems, where automatic partitioning is one key issue. This paper describes a new approach in hardware/software partitioning for multi-processor systems working fully automatic. The approach is based on integer programming (IP) to solve the partitioning problem. A formulation of the IP-model will be introduced in detail. The drawback of solving IP-models often is a high computation time. To reduce the computation time, an algorithm using IP has been developed which splits the partitioning approach in two phases. In a first phase, a mapping of nodes to hardware or software is calculated by estimating the schedule times for each node with heuristics. During the second phase a correct schedule is calculated for the resulting HW/SW-mapping of the first phase. It will be shown that this *heuristic scheduling* approach strongly reduces the computation time while the results are nearly optimal for the chosen objective function.

Another new feature of our approach is the cost estimation technique. The cost model is not calculated by estimators like in other approaches, because the quality of estimations is often poor and estimators do not consider compiler effects. In our approach, a compiler and a high-level synthesis tool are used instead of special estimators. The disadvantage of an increased runtime for calculating values for the cost metrics is compensated by a higher precision of these values. A higher precision leads to fewer partitioning iterations.

Our system specification method is introduced in section 4.1.2.2. In section 4.1.2.3 our approach to partitioning is presented. A formulation of the hardware/software partitioning problem follows in section 4.1.2.4. Section 4.1.2.5 describes the IP-model of the problem. Experimental results of solving these IP-models are presented in section 4.1.2.6.

4.1.2.2 System Specification

One of the key problems in hardware/software codesign is specification of large systems. Many system specification languages have been developed in the last years. One of the most frequently used ones is VHDL, because many CAD tools supporting VHDL exist. In our approach, we also specify systems in VHDL.

To specify a system that has to be partitioned, the designer has to define the following:

1. The **target technology** has to be specified by defining the set of processors for the software parts and the component library for synthesizing the hardware parts of the embedded system.
2. The **system** has to be defined in VHDL as a set of interconnected instances of components (behavioural VHDL-entities).
3. The **design constraints** have to be defined, including performance constraints (timing) and resource constraints (area, memory).

[1] Der nachfolgende englische Text und die zugehörigen Abbildungen sind dem Aufsatz *R. Niemann, P. Marwedel: An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming, Journal of Embedded Systems, 1997* (©Kluwer, 1997) entnommen.

In our approach the target technology, the system, and the design constraints are specified by using the specification tool *COSYS*[2] which is part of the codesign tool *COOL*[3]. *COSYS* is a graphical VHDL-based interface for hierarchical system specification. In the following, specifying systems with *COSYS* will be illustrated by an MPEG audio system.

First, the designer defines **behavioural entities** using VHDL source code. These behavioural entities, called **components**, are instantiated to form **structural entities**. This is done by connecting **instances** of these components by wires (VHDL signals). Structural entities may also be instantiated. Thus, *COSYS* allows the designer to describe the system **hierarchically**. In figure 4.5 the specification of a hierarchical system is illustrated.

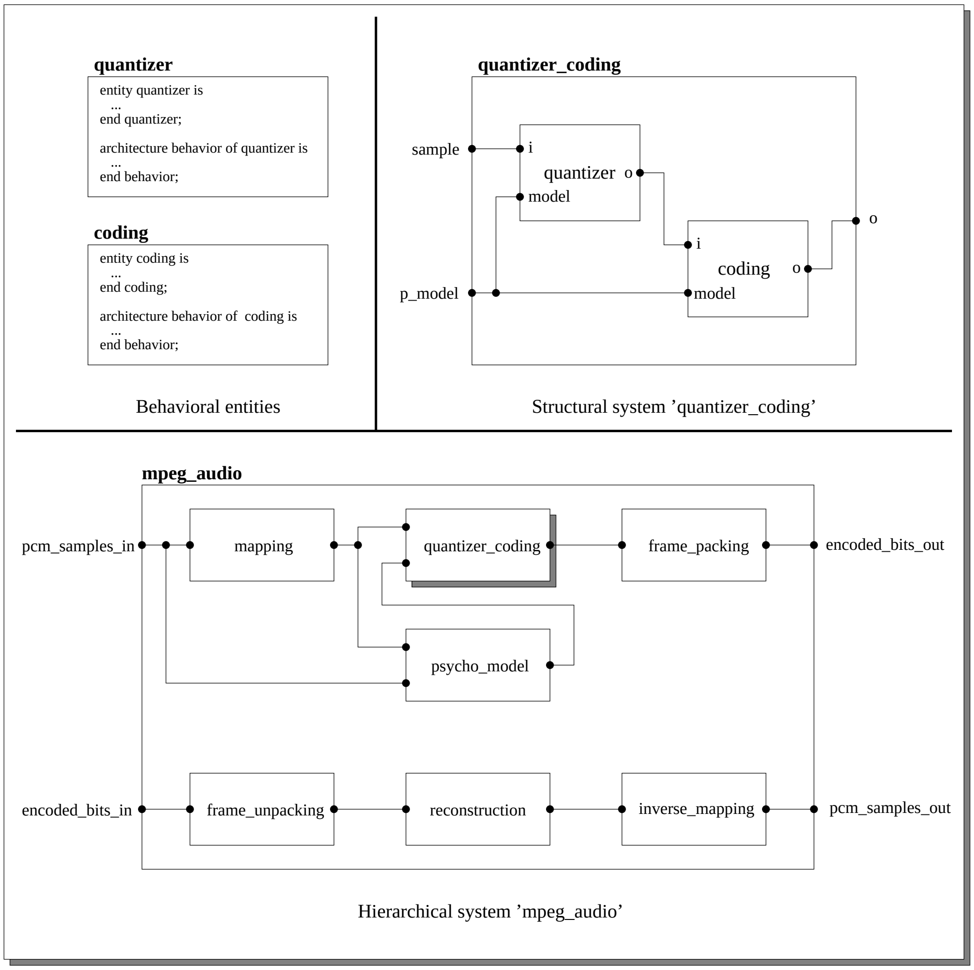


Abbildung 4.5: Hierarchical system specification of an MPEG audio system

The system *mpeg_audio* depicted in figure 4.5 realizes an MPEG audio encoder and decoder. The upper part of the system represents the encoder that encodes incoming PCM audio samples. The result is an encoded bit-stream of the MPEG audio format. The lower part of the system realizes the decoder that decodes MPEG audio bit-streams into PCM samples. The structural entity *quantizer_coding* is instantiated in this hierarchical specification. *Quantizer_coding* is defined with help of two behavioural entities. It contains an instance of component *quantizer* and another instance of component *coding* which have been specified in VHDL code.

The partitioning approach for these specified systems will be described in the following sections.

4.1.2.3 Hardware/Software Partitioning Approach

After the system has been specified with *COSYS*, the VHDL specification is compiled into an internal syntax graph model. For each component (behavioural VHDL-entity) of this model, software source code (C or DFL) and hard-

[2] *COSYS*: (Codesign System specification tool)
[3] *COOL*: (Codesign Tool)

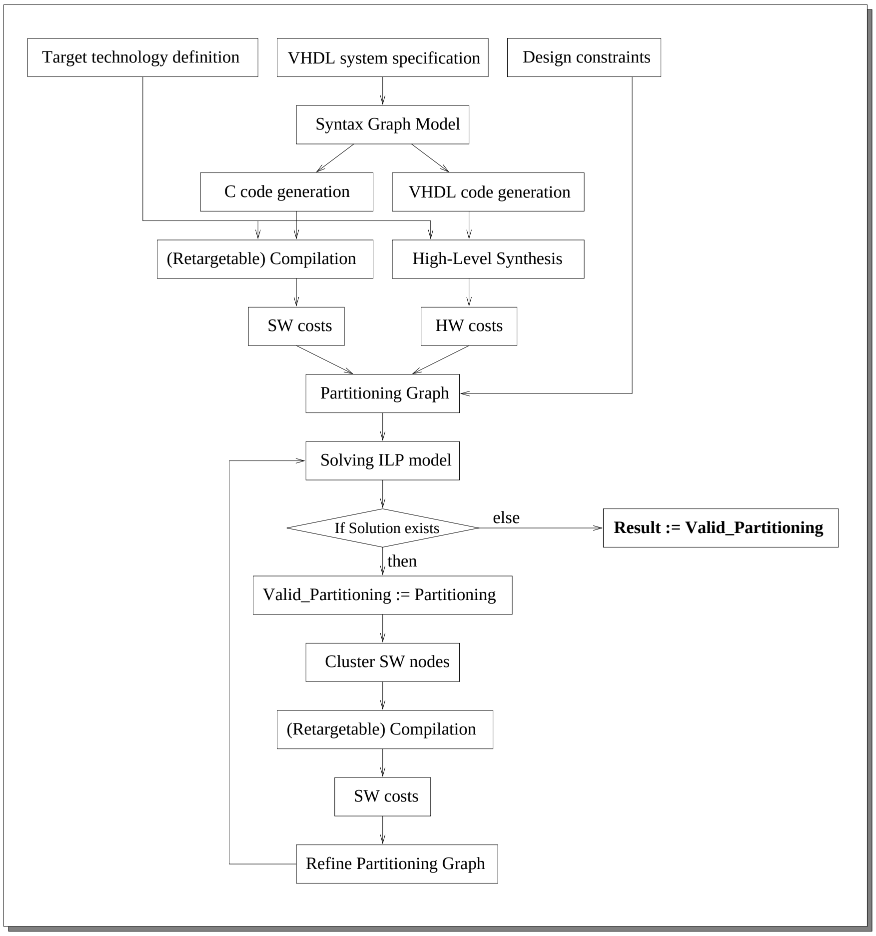


Abbildung 4.6: Hardware/Software Partitioning

ware source code (VHDL) is generated. The software parts are compiled and the hardware parts are synthesized by a high-level synthesis tool (OSCAR [LMD94]). The results are values for software cost metrics (software execution time, memory usage) and values for hardware cost metrics (hardware execution time, area) for the components. The disadvantage of an increased runtime for calculating the cost metrics by running compilers and synthesizers is compensated by a better quality. Moreover, a higher precision of the cost values leads to fewer partitioning iterations. After the compilation/synthesis phase, a partitioning graph is generated in two steps. First, the hierarchy of the system is flattened. Then, a partitioning graph is created in which each node of the graph represents an instance of a component in the flattened system. Edges of the partitioning graph represent the wires between these instances. In figure 4.7 the partitioning graph is calculated for a hierarchical system.

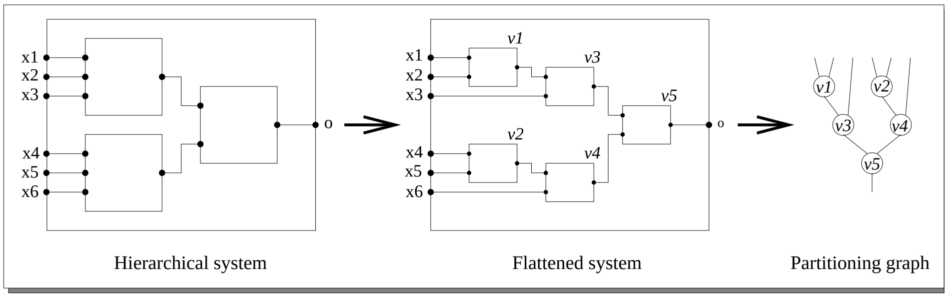


Abbildung 4.7: Calculation of the partitioning graph

In the first step (see fig. 4.7), the structural entities are flattened resulting in a set of instances of components. Then for each instance ($v_1 \ldots v_5$) a node is added to the partitioning graph. The edges between the nodes represent the wires between the instances. Nodes are weighted with hardware and software costs, edges are weighted with interface costs which reflect the cost of hardware/software interfaces. Interface costs are approximated by the number and type of data flowing between both nodes. User-defined design constraints are also attached to the graph. Thus, the partitioning graph includes all information needed for partitioning.

The partitioning graph is then transformed into an IP-model. Afterwards, the model is solved by an IP-solver. The calculated design is optimal for the chosen objective function using the generated cost model, but nevertheless it is possible to improve the design, because although sharing effects between different instances of the same components are considered, sharing effects between different components are not. This limitation can be removed by an iterative partitioning approach. We use a software oriented approach, because compilation is faster than synthesis and software oriented approaches seem to be superior to hardware oriented approaches (see [VGG94]).

Sets of nodes which have been mapped on the same processor are clustered. For each cluster, a new cost metric is calculated by compiling all nodes of the cluster together. Then, the partitioning graph is transformed by replacing each cluster by a new node with the new cost metric attached. Finally, the redefined graph is repartitioned. This iteration will be repeated until no solution is found. The last valid partitioning represents the resulting design. The clustering technique is illustrated in figure 4.8.

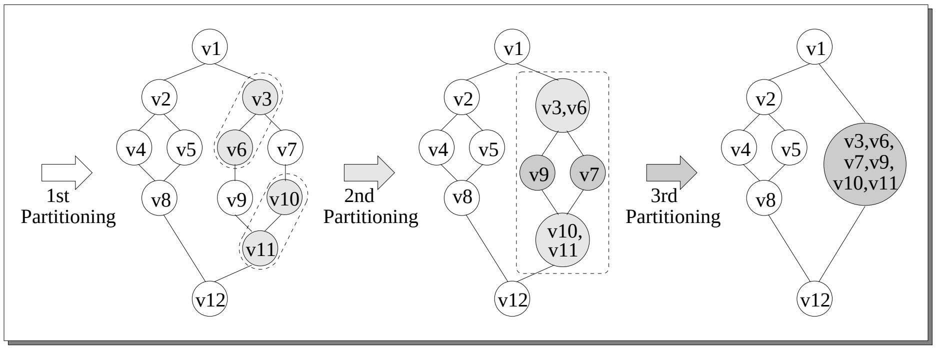


Abbildung 4.8: Partitioning refinement

The first partitioning iteration results in 4 software nodes (v_3,v_6,v_{10},v_{11}) (see fig. 4.8). The nodes v_3, v_6 and v_{10}, v_{11} are clustered. After the second iteration it is now possible to execute v_7,v_9 on the processor, so the new cluster contains v_3,v_6,v_7,v_9,v_{10},v_{11}. In the third iteration no more nodes can be moved from hardware to software.

4.1.2.4 Formulation of the HW/SW Partitioning Problem

This section introduces a formulation of the hardware/software partitioning problem. This formulation is necessary to simplify the description of the problem with the help of an IP-model. We have to define the system which has to be partitioned and the target technology used to implement the system.

4.1.2.4.1 Target Technology and System Specification

Definition 1 *A* **target technology** $\mathcal{T}$ *is defined as a tuple*

$$\mathcal{T} = (\mathcal{V}, \mathcal{E}), \qquad \mathcal{V} = \mathcal{H} \cup \mathcal{P} \cup \mathcal{M}, \qquad \mathcal{E} \subseteq \mathcal{PS}(\mathcal{V}) \setminus \big\{\{v\} \mid v \in \mathcal{V}\big\}$$

containing all target technology components and interconnections. The target technology components $\mathcal{V}$ *are defined as a set of hardware components (ASICs)* $\mathcal{H} = \{h_1, \ldots, h_{n_\mathcal{H}}\}$, *processors* $\mathcal{P} = \{p_1, \ldots, p_{n_\mathcal{P}}\}$ *and memories* $\mathcal{M} = \{m_1, \ldots, m_{n_\mathcal{M}}\}$. *The target technology interconnections* $\mathcal{E}$ *are defined as a set of busses* $\mathcal{E} = \{e_1, \ldots, e_{n_\mathcal{E}}\}$ *connecting these components (at least 2) where* $\mathcal{PS}(\mathcal{V})$ *represents the power set of* $\mathcal{V}$.

In figure 4.9 an example for a target technology is given. It contains a processor p_1, a hardware component h_1, external memory m_1 and a bus b_1 connecting p_1, h_1 and m_1.

A system that has to be mapped to the target technology consists of several instances of different system components and interconnections between them. The formal definition is as follows:

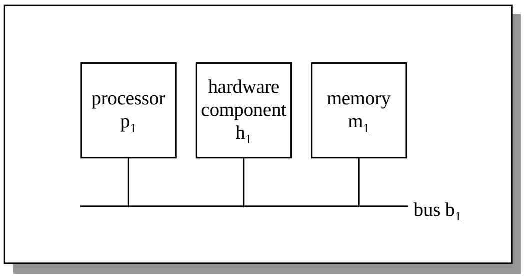


Abbildung 4.9: Target technology

Definition 2 *A* **system** $\mathcal{S}$ *is defined as a 4-tuple*

$$\mathcal{S} = (C, V, E, I)$$

with the following definitions:

$C = \{c_1, \ldots, c_{n_C}\}$	*set of system components,*
$V = \{v_1, \ldots, v_{n_V}\}$	*set of nodes, representing instances of system components,*
$E \subseteq V \times V$	*set of edges, representing interconnections between nodes,*
$I : V \to C$	$I(v_i) = c_l$ *defines that* v_i *is an instance of component* c_l.

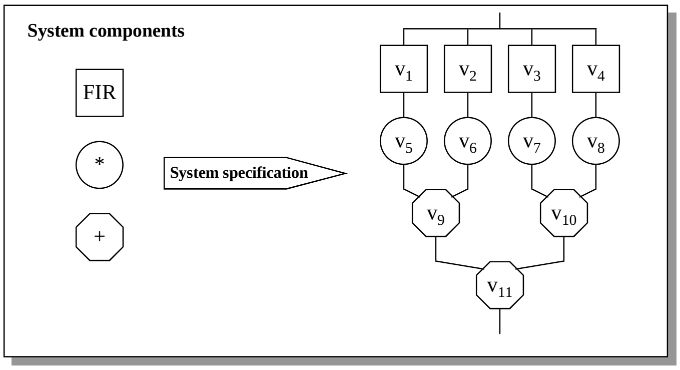


Abbildung 4.10: System specification

In figure 4.10 a 4-band-equalizer is specified. It consists of 3 system components: an FIR-filter, a multiplier and an adder. The equalizer is specified by using 4 instances of the FIR-filter ($v_1 \ldots v_4$), 4 instances of the multiplier ($v_5 \ldots v_8$) and 3 instances of the adder ($v_9 \ldots v_{11}$). This 4-band-equalizer is a well suited example to demonstrate the scheduling, hardware sharing and interfacing problem. Therefore, it will be used in the rest of the paper as a demonstrator example.

4.1.2.4.2 Cost Model Hardware/software partitioning algorithms need **cost metrics** for the nodes and the edges of the system to evaluate different partitionings. The values for these cost metrics are calculated for the possible hardware and software implementations of system components. This is done during the compilation and synthesis phase, described in section 4.1.2.3.

According to the cost metrics definition for system components, we can define the resource costs for each target technology component.

Definition 3 *The* **design costs** *of a system* $\mathcal{S}$ *are defined as follows:*

$C^{dm}(S)$	*represents the used software data memory,*
$C^{pm}(S)$	*the used software program memory,*
$C^{a}(S)$	*the hardware area and*
$C^{t}(S)$	*the total execution time.*

The total cost is a weighted average over these values. The design costs may also be constrained.

4.1.2.5 The IP-Model

Many optimization problems can be solved optimally by using integer programming (IP). This paper will show that our IP-model allows us to solve the hardware/software partitioning problem with the following characteristics:

- optimal solution for an objective function,
- support for multiprocessor and multi-ASIC target technologies,
- timing constraints are guaranteed by scheduling the nodes,
- bus conflicts are prevented by scheduling communication events on edges,
- interface costs are considered,
- instances of the same system component can share their implementation on hardware,
- interactive support for user-defined constraints.

The following paragraphs describe a subset of the IP-model for performing hardware/software partitioning with these characteristics.

Definition 4 *Indices*

$L = \{1, \ldots, n_{\mathcal{C}}\}$	*indices for system components* $c_l \in \mathcal{C}$,
$I = \{1, \ldots, n_V\}$	*indices for nodes* $v_i \in V$,
$J \in N_0^+$	*indices for hardware instances of system components,*
$KH = \{1, \ldots, n_{\mathcal{H}}\}$	*indices for hardware components* $h_k \in \mathcal{H}$,
$KP = \{1, \ldots, n_{\mathcal{P}}\}$	*indices for processors* $p_k \in \mathcal{P}$,

4.1.2.5.1 The Decision Variables

The IP-model needs decision variables for defining mapping, scheduling, sharing and interfacing constraints. Thus, the solution of the IP-model is driven by the following variables:

Definition 5

$$
\begin{aligned}
x_{i,j,k} &= \begin{cases} 1 & : \; v_i \textit{ is mapped to the } j-th \textit{ hardware instance of } c = I(v_i) \\ & \;\;\; \textit{on hardware component } h_k, \\ 0 & : \; \textit{otherwise.} \end{cases} \\
X_{i,k} &= \begin{cases} 1 & : \; v_i \textit{ is mapped to hardware component } h_k, \\ 0 & : \; \textit{otherwise.} \end{cases} \\
Y_{i,k} &= \begin{cases} 1 & : \; v_i \textit{ is mapped to processor } p_k, \\ 0 & : \; \textit{otherwise.} \end{cases} \\
NY_{l,k} &= \begin{cases} 1 & : \; \textit{at least 1 instance of } c_l \textit{ is mapped to processor } p_k, \\ 0 & : \; \textit{otherwise.} \end{cases} \\
NX_{l,k} &: \; \textit{number of hardware instances of } c_l \textit{ realized on } h_k.
\end{aligned}
\tag{4.1}
$$

To visualize usage of these variables, figure 4.11 shows a partitioning of the 4-band-equalizer.

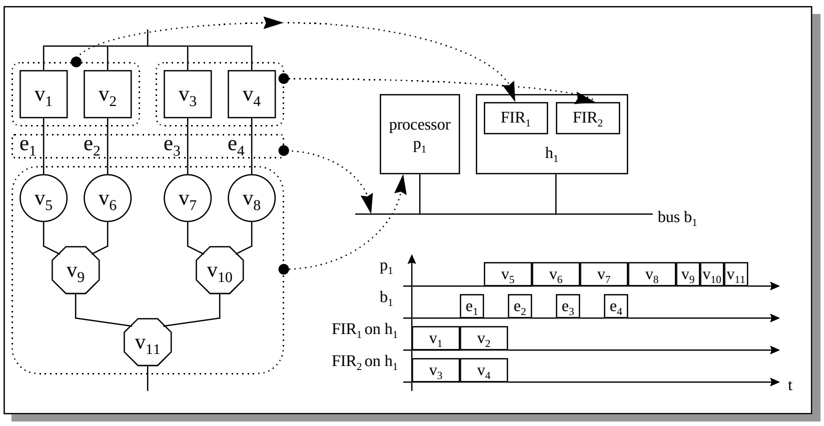


Abbildung 4.11: Hardware sharing

$v_1 \ldots v_4$ are mapped to hardware component h_1 ($X_{1,1} = \ldots = X_{4,1} = 1$). All other nodes $v_5 \ldots v_{11}$ are mapped to processor p_1 ($Y_{5,1} = \ldots = Y_{11,1} = 1$). This mapping forces interfaces for edge $e_1 = (v_1, v_5), \ldots, e_4 = (v_4, v_8)$ to be needed, because data has to be transported from h_1 to p_1. Therefore, $e_1 \ldots e_4$ are mapped to bus b_1. To reduce the amount of hardware area, v_1 and v_2 share the same hardware instance of an FIR-filter on h_1 ($x_{1,1,1} = x_{2,1,1} = 1$). v_3 and v_4 share the second one ($x_{3,2,1} = x_{4,2,1} = 1$).
The variables get the following values: $nx_{1,1,1} = nx_{1,2,1} = 1, nx_{1,3,1} = nx_{1,4,1} = 0$, because only the first two hardware instances of four possible FIR-filters (c_1) are required on h_1 ($NX_{1,1} = 2$). All multiplications (c_2) and adders (c_3) are realized on processor p_1. Therefore, one function is needed for multiplying ($NY_{2,1} = 1$) and another function is needed for adding ($NY_{3,1} = 1$) incoming values. The timing diagram shows a possible schedule for this partitioning.

4.1.2.5.2 The Constraints

The following constraints have to be fulfilled:

1. **General Constraints**: Each node v_i is executed exactly on one target technology component t_k, a processor or a hardware component (eq.4.4). If a system component c_l has been realized on a processor p_k, then it is not necessary to implement it more than once (eq.4.5), because it can be implemented as one function and several function calls. Therefore, the number $NY_{l,k}$ is calculated by equations 4.6 and 4.7. In contrast to the binary variable $NY_{l,k}$, the number of hardware instances $NX_{l,k}$ of a system component c_l realized on a hardware component h_k may be greater than one. If no hardware sharing is considered, then $NX_{l,k}$ is equal to the sum of system instances of c_l that have been mapped to h_k (eq.4.8).

$$\forall i \in I : \forall k \in KH : \quad X_{i,k} \leq 1 \tag{4.2}$$
$$\forall i \in I : \forall k \in KP : \quad Y_{i,k} \leq 1 \tag{4.3}$$
$$\forall i \in I : \quad \sum_{k \in KH} X_{i,k} + \sum_{k \in KP} Y_{i,k} = 1 \tag{4.4}$$

$$\forall l \in L : \forall k \in KP : \quad NY_{l,k} \leq 1 \tag{4.5}$$
$$\forall l \in L, \forall i : I(v_i) = c_l, \forall k \in KP : \quad NY_{l,k} \geq Y_{i,k} \tag{4.6}$$
$$\forall l \in L, \forall k \in KP : \quad NY_{l,k} \leq \sum_{i:I(v_i)=c_l} Y_{i,k} \tag{4.7}$$
$$\forall l \in L, \forall k \in KH : \quad NX_{l,k} = \sum_{i:I(v_i)=c_l} X_{i,k} \tag{4.8}$$

2. **Resource Constraints**: The area C_k^a used on a hardware component h_k is calculated by accumulating the costs for all hardware instances of system components realized on h_k. The amount of used memory on a processor p_k is calculated by summing up the costs for implementing these system components as functions. The resource costs may not violate their resource constraints.

3. **Design Constraints**: The design costs for the complete system are calculated by accumulating the resource costs required by the components of the target technology. These design costs may not exceed their given design constraints. The required hardware area includes additional hardware CI_k^a used for interfaces. If interfacing is not considered, $CI_k^a = 0$ for all busses. The design costs C^t will be described separately.

4. **Timing Constraints**:
   The timing costs cannot be calculated by accumulating the execution time of the nodes, because two nodes v_1, v_2 can be executed in parallel if they do not share the same resources and if there is no path from v_1 to v_2 and vice versa. To determine the starting time and ending time for each node, scheduling has to be performed. The execution time T_i^D is either a hardware or a software execution time. The ending time T_i^E of v_i is the sum of starting time T_i^S and execution time T_i^D. The system execution time C^t is the maximum of the ending times of all nodes v_i and may not violate the global design timing constraint. Data dependencies have to be considered for all edges $e = (v_{i_1}, v_{i_2})$ including interface communication time $TI^D_{i_1,i_2}$. If interfacing is not considered, $TI^D_{i_1,i_2} = 0$. The starting times T_i^S of nodes have to be in their ASAP/ALAP-range which can be calculated in a preprocessing step.

4.1.2.5.3 Hardware Sharing

If hardware sharing is considered, then it is not sufficient to model bindings between nodes v_i and hardware components h_k with help of the binary variable $X_{i,k}$. In order to consider hardware sharing, the binding of v_i to the j-th hardware instance (of system component $c_l = I(v_i)$) contained in h_k has to be modelled.

4.1.2.5.4 Interfacing An interface has to be realized for an edge $e = (v_{i_1}, v_{i_2})$, if v_{i_1} and v_{i_2} are realized on different target technology components.

4.1.2.5.5 Scheduling Two nodes v_{i_1}, v_{i_2} which can be executed in parallel have to be sequentialized, if

- v_{i_1} and v_{i_2} are executed on the same processor or
- v_{i_1} and v_{i_2} share the same hardware instance on the same hardware component.

To sequentialize two nodes v_{i_1}, v_{i_2}, the binary decision variable b_{i_1,i_2} is used.

Two edges e_{i_1}, e_{i_2} have to be sequentialized, if e_{i_1} and e_{i_2} represent interfaces and both edges use the same bus to realize the communication. In this case, the binary decision variable bi_{i_1,i_2} is used to schedule e_{i_1} and e_{i_2}.

4.1.2.5.6 Heuristic Scheduling Resource constrained scheduling is a NP-complete problem [GJ75]. Therefore, it is clear that solving the scheduling problem optimally can not be done efficiently. For this reason, we have developed an algorithm using integer programming that solves the partitioning problem while iterating the following steps:

1. Solve an IP-model for the hardware/software mapping with help of approximated time values.
2. Solve an IP-model for calculating a valid schedule with nodes mapped to hardware or software.
3. If the resulting total time violates the timing constraint, repeat the first two steps with a timing constraint that is tighter than the approximated total time of step 1. (see figure 4.12).

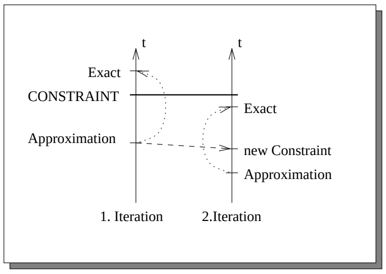


Abbildung 4.12: Heuristic scheduling

The first partitioning results in an approximated execution time which fulfills the given timing constraint. However, the exact execution time violates this constraint. For this reason, a second partitioning with a new timing constraint is executed. This new constraint is tighter than the approximation of the first partitioning. The second partitioning results in a decreased approximated execution time. The exact execution time of the second partitioning fulfills the original timing constraint. Therefore, the second partitioning represents the solution.

4.1.2.6 Results

To evaluate the quality of our partitioning approach we have done an application study in the area of audio algorithms. We have implemented systems between 6 and 29 nodes (between 6 and 37 edges) which have to be partitioned. The partitioning results in this section have been calculated for the following systems:

- n-band-equalizers with $n \leq 7$,
- a system called *audiolab* including a mixer, a fader, an echo, an equalizer and a balance ruler, and
- an MPEG audio encoder (layer II).

These systems were mapped to a target architecture (see figure 4.13) containing a SPARC processor, an ASIC manufactured in a 1μ CMOS technology (COMPASS library) and external memory. A bus connects these components.

All calculated partitionings consider interface costs and hardware sharing effects between nodes. The IP-models were solved by using the IP-solver package OSL [4] from IBM. The computation times of the examples represent CPU seconds on a RS6000. The heuristic partitioning approach can be evaluated by examining

[4] OSL : Optimal Subroutine Library

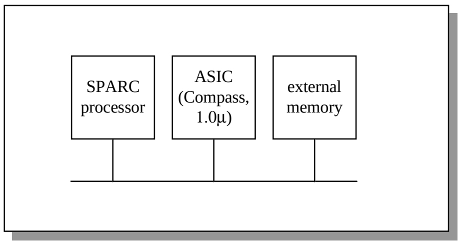


Abbildung 4.13: Target architecture

- the quality and
- the computation time

compared to optimal solutions.

The quality of the heuristic approach can be derived from the deviation between the exact and the approximated solutions. Two equalizers, a 2-band- and a 3-band-equalizer, were partitioned with the optimal and the heuristic approach. For each system, solutions were calculated for a set of 8 timing constraints, resulting in a set of designs ranging from a complete software to a complete hardware solution.

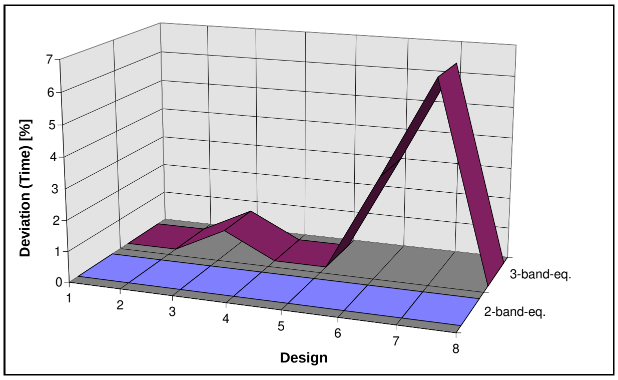


Abbildung 4.14: Deviation System Execution Time (exact/heuristic approach)

The hardware area deviation is zero for both benchmarks. The total system execution time differs between both approaches (see figure 4.14). The optimal approach calculates a mapping first using a minimal amount of hardware area, and then minimizes the system execution time. The heuristic approach tries to minimize the hardware area and finding a valid schedule in a second step. For this reason, the resulting system execution times may differ. The deviation in our experiments is not greater than 6.4%. The average deviation is smaller than 1%.

The main difference of both approaches is the computation time (see figure 4.15). The computation time for both

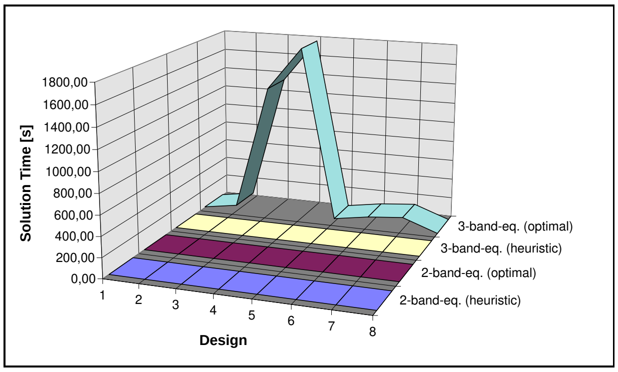


Abbildung 4.15: Computation Time (exact/heuristic approach)

benchmarks is below one second for all designs using the heuristic approach. The computation time calculating the optimal solution is 1773 seconds in the worst case. It becomes clear that solving the hardware/software partitioning problem optimally is not applicable to systems with a larger number of instances.

In figure 4.16 an overview of partitioning all benchmarks using the heuristic approach is given.

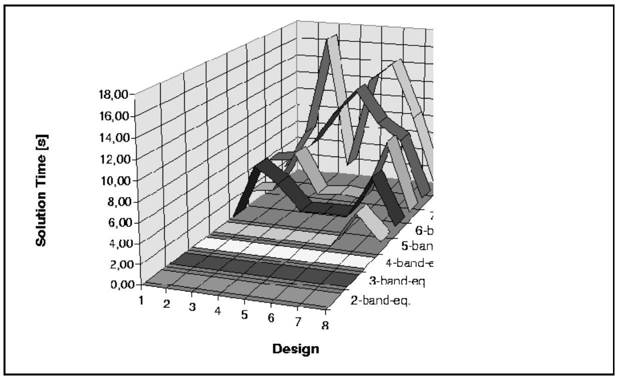


Abbildung 4.16: Computation Time (heuristic approach)

The largest computation time is 16.9 seconds for partitioning the *mpeg* system (containing 29 nodes and 37 edges).

Clearly, the heuristic approach is more practical than the optimal approach, because the results are always nearly optimal and the computation times are significantly lower.

Finally, the trade-off between hardware area and system execution time is demonstrated for the *audiolab* system (containing 25 nodes and 31 edges). 8 different partitionings (see figure 4.17) have been calculated for 8 different timing constraints.

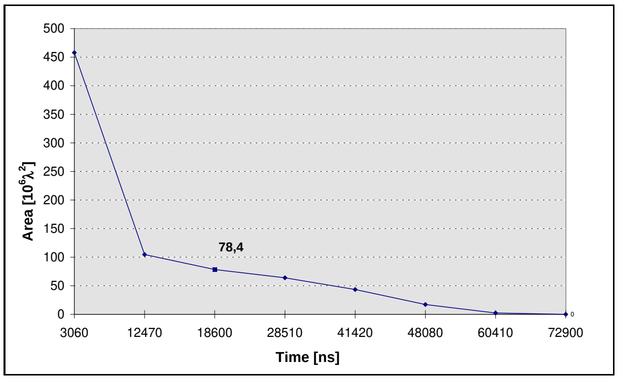


Abbildung 4.17: Area/Time Curve for the Audiolab System

A pure software realization of the *audiolab* system would result in a system execution time of 72900 ns, but this solution is too slow. The fastest realization would have a system execution time of 3060 ns, but it would be a complete hardware realization using $457,9 * 10^6 \lambda^2$ chip area. This solution is a too expensive. The best solution is a hardware/software solution which fulfills the timing constraint of 22675 ns (44.1 kHz sample frequency) with a minimal amount of hardware. The calculated solution has a system execution time of 18600 ns and would require $78,4 * 10^6 \lambda^2$ chip area.

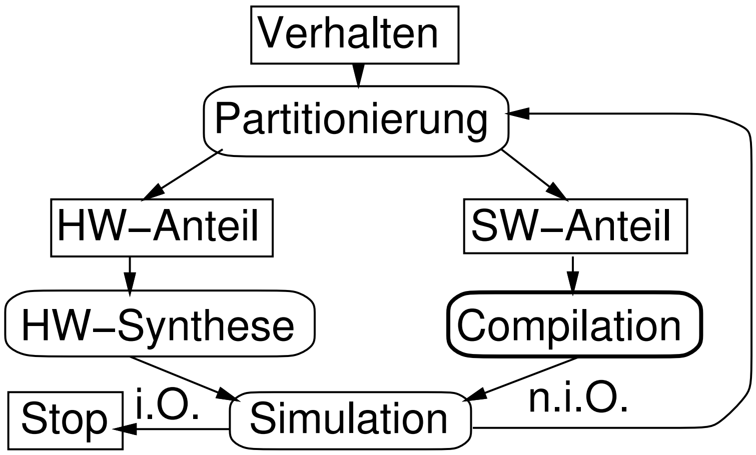
4.2 Codeerzeugung für eingebettete Systeme

Im Falle der softwaremäßigen Realisierung einer Spezifikation muss man sich mit Prozessoren und Codeerzeugungstechniken beschäftigen. Diese werden in dem im Folgenden beschrieben[5].

4.2.1 Embedded Processors, Core Processors and Embedded Systems

There has recently been a huge amount of interest in embedded processors in general and in embedded core processors in particular. What is the main reason behind this huge interest?

[5]Der nachfolgende englische Text und die zugehörigen Abbildungen sind dem Aufsatz *P. Marwedel: Code Generation for Embedded Processors, SASIMI'97, Osaka, 1997* (©SASIMI, 1997) entnommen.



The main reason is *flexibility*. It is possible to change the overall behavior of a processor-based design by changing the program that is executed on the processor. This way, it is possible to accommodate late changes of the specifications. These days, specifications change even after the designers have already started to generate a design. If this happens, ASIC designers may have to start all over again. Designers of processor-based systems are in a better position: they modify the program, re-compile and re-load it.

Flexibility also simplifies upgrading a design. Upgrading does in fact include a number of things: it includes, for example, both the generation of a more enhanced product and downloading new *firmware code* into some product already delivered to the customer.

In the special case of processors cores and other off-the-shelf processors, there is one additional advantage and that is *reuse*. Future chips, which are expected to contain more than 10^8 transistors can only be designed in acceptable time frames if complex components are reused.

Reuse of complex components has a number of advantages:

- It cuts down design time to the required level.
- Reuse of cores also improves the efficiency of the design, since cores are usually highly optimized.
- Testing is simplified because test engineers know the components they have to test from previous designs.

Normally, it is very difficult to provide flexibility and reuse at the same time. These two features are in fact in many cases mutually exclusive. Processors cores, however, exhibit both features at the same time. It is this combination of features, which makes them so popular[6].

Where will those processors and processor cores be used? The various types of processors are intended to be used in so-called *systems-on-a-chip*. In such systems, there is a variety of different components: processors, RAM and ROM memory, various converters and possibly some full custom glue logic. A very important consequence of chip-level integration is that the size of RAM and ROM is extremely important. This, in turn, means that the code generated from application programs has to be very compact and has to use RAM locations for variables very efficiently. This is different from board-level integration, where the size of ROMs and RAMs does not matter that much.

What are actually the main applications areas for systems-on-a-chip? According to market analysts [Rya95], the fastest growing market in general and for systems-on-a-chip in particular is the *embedded systems market*. In this context, *embedded systems* can be defined as *systems reading, processing and controlling physical quantities*. For embedded systems, market analysts are predicting so-called double-digit growth rates, or growth rates beyond ten percent. This indicates that the embedded systems market should receive an adequate amount of attention. Information processing is not restricted to what people come into contact with at the universities and at their desk. Embedded systems are frequently not realized as being present, because they do not come with a screen and a keyboard.

Most of the functionality of embedded systems can be implemented with software and embedded processors. Due to the flexibility of software solutions and the trend towards faster and faster processors, special hardware solutions will become less and less important. The trend towards software solutions does already exist for some time and has already led to the wide-spread use of processors in embedded systems. Two examples show that this use has already exceeded the level that is expected by most people:

1. According to Camposano [CW96], the New York Times has estimated (in 1995) that the average American

[6]FPGAs are the only others component providing the same combination of features. But FPGAs are by far not as efficient as processors.

comes into contact with about 60 microprocessors every day. Even today (1997) most people believe this number to be lower.

2. Due to personal information, some top-level cars include at least 100 microprocessors.

These numbers indicate that quite some fuctionality of embedded systems has already been implemented by software and processors.

The different processors that are used for embedded systems can be classified by using three main characteristics of processors.

The first characteristic (see fig. 4.18) is the form in which processors are available. Processors are called core processors, if they are available as an entry in a CAD library. There are, of course, other forms in which processors are available (for example, in packaged form).

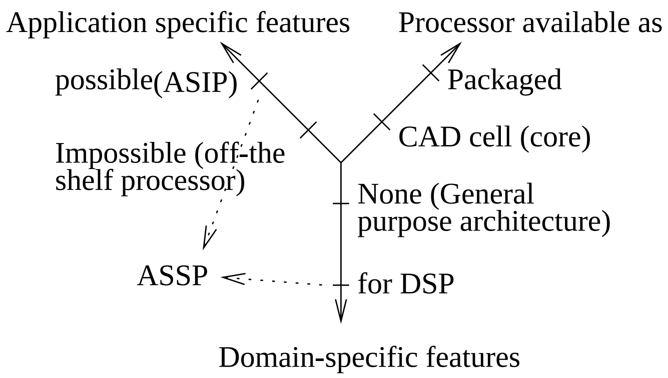


Abbildung 4.18: Cube of processor types

The second characteristic is the availability of domain-specific features. Of course, some processors do not have any of these domain-dependent features. They are called *general purpose processors*. Some processors come with domain-specific features. The goal of adding such features is to find a good match between application domain and processor architecture, leading to a high overall efficiency. For example, processors for digital signal processing include saturating arithmetic, multiply/accumulate instructions and special addressing modes for implementing digital filters and Fourier transforms. Other processors exhibit special features for microcontroller applications, such as sophisticated bit manipulation instructions.

Even more efficient designs are feasible with application-specific features. If such features are present, the corresponding processor is called an ASIP (*application-specific instruction set processor*). There is some research on how optimized ASIPs can be designed for given applications [ANH$^+$93, HD94]. It has also been proven, that ASIPs can in fact be more efficient than domain-specific or general purpose processors [Hua96]. *Off-the-shelf processors* are processors which do not include any application-specific features. *Application-specific signal processors* (ASSPs) are application-specific processors including special features for digital signal processing. In a 3D representation of processors by a cube, ASSPs correspond to one of the edges.

In addition to the three coordinates, there are of course other criteria for classifying processors.

4.2.2 Efficiency of Compilers for Embedded Processors

4.2.2.1 Existing Compilers

Now, the next question is: is it really a problem if future systems will be implemented in software? There are many mature software design environments which now could also be used for the design of embedded systems. The customer base for these environments is larger than that for ECAD tools and hence more money is available for their development. This could be used to make them more reliable and robust than currently used hardware design systems, for which the number of copies sold is usually quite small. Does this mean than designers of embedded systems will find all the tools they need on the market?

This would be nice, but this will most likely not be the case. Development tools for processor-based systems currently have some severe limitations. This applies to various kinds of development tools, but in this contribution we will focus on compilers.

Problems with using current compiler technology for embedded processors have been mentioned by quite a number of industrial designers. Detailed numerical data have been published by a group of reseachers working at the University of Aachen. Researchers at Aachen have compared the size and the speed of assembly language library routines with the size and the speed of compiled code. According the results of this DSPStone benchmark project [Ze94], overhead of compiled code (in terms of code size and clock cycles) typically ranges between 2 and 8.

As an example, we consider the results that have been found for the ADPCM algorithm, which is a very common algorithm for speech encoding. For this algorithm and three different processors, the data memory overhead ranges between about 170 % and almost 400 %. Thus, if compilers are used, the data memory has to be up to almost 5 times larger than for assembly code.

For the same benchmark, the situation is even worse if the speed overhead is taken into account. This overhead ranges between about 500 and almost 700 %. This means that up to 7/8th of all processor cycles are wasted if compilers are used. This translates into a huge loss of performance and electrical power and is clearly not acceptable for embedded systems. *Optimizations for low-power design should not be constrained to the hardware level. From an overall point of view, a highly optimizing compiler is one of the most important contributions to low power design.*

Due to the current lack of highly optimizing compilers, a major amount of applications are implemented in assembly languages. The exact percentage varies from company to company and from application to application. Paulin has computed this percentage for a closed set of applications [PLMS95]. He found that a major percentage of DSP applications and of controller applications is written in assembly languages.

Implementing complex systems using assembly languages has all the well-known disadvantages, for example a long time to the market, a low product quality and the inability of retargeting the application to new processors.

Before we try to change this situation, we should analyze the reasons behind this poor performance of current compilers. These reasons can be understood, if we look at the how currently available architectures for embedded processors were designed.

For most of these processors, the design goal was to have very efficient processor hardware. This hardware was designed to fit typical applications in the application domain as good as possible. Ease of compilation was never really a design goal.

As an example let us consider the TMS320C25, a very popular digital signal processor. The microarchitecture of this processor contains a variety of registers. There are registers at the input and the output of the multiplier and a special accumulator at the output of the adder. Furthermore, there are special address registers, a special address register pointer register and there is a small data RAM. All these registers have a different functionality. Due to this, the register set is said to be *heterogenous*. This constrasts with the homogenous register files found in most standard RISC architectures.

Having a heterogenous register set does indeed make sense for DSP architectures. The registers at the multiplier and the adder can very efficiently be used to implement digital filters. The accumulator is appropriate for storing partial sums during the filtering operation. Replacing these specialized registers by homogenous registers files would possibly extend the critical path for multiplications, would not speed up digital filtering, would increase the chip area and it would restrict the maximum number of accesses to registers in any clock cycle. The only 'advantage' would be to make the design of a compiler easier.

Since embedded systems have to be efficient, processors with special features for embedded systems, in particular for DSP and microcontroller applications, will probably be used for many years. This should generate interest in compiler techniques supporting those features.

4.2.2.2 New Optimization Techniques

4.2.2.2.1 Overview As will be shown in the next sections, it is actually possible to significantly improve the performance of compilers for embedded processors. Some researchers have recently designed new optimization algorithms exploiting special features of embedded processors.

Wess[Wes95], Araujo and co-workers [AM95] have proposed register assignment techniques for heterogenous register sets. These techniques extend register assignment techniques for homogenous register sets that are used for standard RISC- and CISC-processors.

Different operation *modes* are another typical feature of embedded processors. Many DSP processors provide a choice between saturating and wrap-around arithmetic modes and also use modes for selecting significant bits of fixed point numbers after multiplication. The instruction sets of these processors include instructions for switching between these modes (in microprogramming, the same concept was called *residual control*). Compilers should try to generate a minimized number of mode switching instructions. Liao has published an algorithm performing this type of optimization [LDKT95].

The recently announced 'C60 processor of Texas Instruments offers a significant amount of instruction-level parallelism. Up to 8 instructions of 32 bits each can be executed in parallel. The parallelism of this and similar machines can only be exploited if techniques for globally rearranging code exist. These techniques are called global scheduling techniques. *Mutation scheduling* (by Nicolau) [NND95] is possibly the most sophisticated global scheduling technique. It incorporates global code movement, consideration of available hardware resources and the application of algebraic rules.

Nicolau and his group have also worked on the integration of loop unrolling and register allocation [KN96], a technique required for finding a compromise between code density and execution speed for loops.

Another feature, which is commonly found in many processors is the presence of an address generation unit. Such units allow address computations to be performed concurrently with other arithmetic operations. Bartley [Bar92], Liao [LDK+95], Leupers [Leu96], and Sudarsanam [SLD97] have recently proposed algorithms exploiting this extra hardware.

Two optimizations will be explained in a little more detail in the next section.

4.2.2.2.2 Exploitation of instruction level parallelism A number of architectures (such as the popular TMS 320C25) allow several register transfers to be encoded into a single instruction [Tim95]. This can actually be exploited in practical examples. For a simple set of two C-statements, we can generate a sequence of register transfers that are legal for the 'C25. Already in this small example, a set of three transfers can be identified, which can be encoded in the same instruction and executed in one cycle. The result is a reduction from 9 to 7 cycles.

```
u(n) = u(n-1) + K0 e(n) + K1 e(n-1);      e(n-1) e(n)

ACCU:= u(n-1);              ACCU:= u(n-1);
TR := e(n-1);               TR := e(n-1);
PR := TR * K1;              PR := TR * K1;
TR := e(n);                 TR := e(n);||ACCU := ACCU + PR;
e(n-1) := e(n);                        || e(n-1) := e(n);
ACCU := ACCU + PR;
PR := TR * K0;              PR := TR * K0;
ACCU := ACCU + PR;          ACCU := ACCU + PR;
u(n) := ACCU;               u(n) := ACCU;

9 cycles                    7 cycles
```

Abbildung 4.19: Compaction

Techniques for generating such compacted code have already been studied in the context of microprogramming. In the meantime, however, the speed of available computers, the quality of integer programming packages and the knowledge about modeling techniques have been improved to an extent making optimal algorithms applicable for not too small basic blocks. The technique developed by Leupers [LM95] can be applied to basic blocks of up to about 50 transfers. This is sufficient for most practical examples. Partitioning of larger basic blocks into blocks of this size is not required except for few exceptional cases.

The results for compaction are excellent. For various entries in the DSPStone benchmark suite, code size reductions between about 5 and more than 30% have been observed. There is only a single case (lattice2), in which the reduction does not exceed 10 %. The execution times for the compaction algorithm are very moderate: they range from 2 to 35 seconds.

4.2.2.2.3 Exploitation of multiple memory banks In order to study another optimization technique, we consider the microarchitecture of the Motorola 56k processor containing two memory banks. Each bank comes with its own address generation unit (AGU). A move between the X-memory bank and the the X-registers and another move between the Y-memory and the Y-registers can be performed in parallel. In addition, there are cases in which yet another operation can be performed in parallel.

Techniques for allocating variables to memory such that the total number of parallel operations is maximized have been proposed by Sudarsanam and Malik [SM95]. They are based on conflict graphs. In such graphs, nodes denote variables and symbolic registers. Edges between any pair of nodes model potential parallel accesses to the corresponding values and hence indicate that these values should be held in different memory banks or register sets.

An algorithm based on simulated annealing tries to respect as many of these edges as possible. The results are quite impressive: code size reductions for the DSPStone benchmark range between 20 and almost 60 %.

These optimization techniques show that substantial improvements of available compilers are feasible. There is some hope that future compilers for embedded processors will be much better than those that are currently available and that eventually this will lead to a replacement of assembly language programming for embedded systems.

This concludes the description of new optimization techniques. Next, we turn our attention towards retargetability.

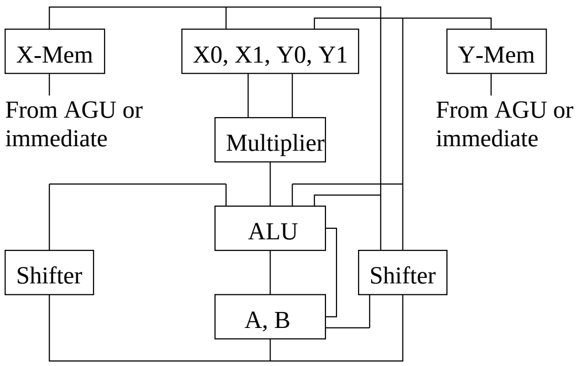


Abbildung 4.20: Mototorola 56k processor architecture

4.2.3 Retargetable Compilers

4.2.3.1 What is retargetability?

Let us first try to define the term. A compiler is said to be retargetable, if it can be applied to a range of target processors.

Actually, we can distinguish between different levels of effort for switching to a new target:

- A high effort is required for so-called *portable* compilers. Porting a compiler to a new target possibly includes rewriting some parts of the compiler.
- More precisely than above, we say that a compiler is retargetable if it includes almost no processor specific code. The characteristics of the target processor msut be captured in a separate *target description*.

4.2.3.2 Why retargetability?

Retargetability is difficult to implement. Why then do people investigate techniques for generating retargetable compilers?

We would like to mention four different reasons:

1. Retargetable compilers are required to support the use of ASIPs. Many different instruction sets can be defined by choosing values for the generic parameters of ASIPs. For each set of values, there will possibly be only a small number of designs. For this small number, it will not be economically feasible to design a specialized compiler. So, there should be a retargetable compiler which can generate code for all legal value sets of generic parameters.

   As far as ASIPs are concerned, no retargetability for very different processors is required. Retargetability within the range of parameters is sufficient.

   Why do we need ASIPs? The main reason for ASIPs is that embedded systems require maximum efficiency and maximum efficiency can only be obtained through customization. It has been observed that customization of processors results in more computations per Watt than can be achieved for standard processors. Fully application-specific hardware (ASICs) would achieve a even higher number of computations per Watt, but ASICs lack flexibility. Hence, ASIPs are a good compromise between power consumption and flexibility. Due to their low power consumption, it has been observed that first generation products are sometimes implemented with standard processors and these are later replaced by ASIPs in second generation products.
2. For embedded systems, there is a large range of applications. This range includes, for example, health care applications in which the systems are inserted into the human body, applications in telecommunications and applications in transportation systems.

   Due to the large variety of applications, a large variety of processors are also required. Due to the mutual dependencies between instruction set and microarchitecture, we believe that there will be different instruction sets, each of which will provide a good match for some applications.

   This means that we also need compilers for different instruction sets which are used for small or medium number of applications. This will be economically feasible only if compilers can be easily retargeted to different instruction sets.
3. With retargetable compilers, it is possible to analyse tradeoffs between adding more processor features and the resulting size, speed and possibly also the power consumption of the processor. At a high level, these

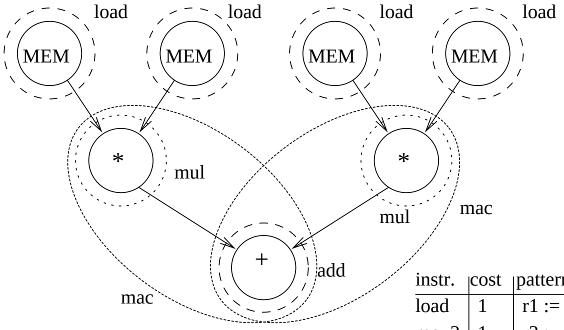
features may be instructions that can be added or deleted. At a lower level, one could also experiment with hardware features, provided that the compiler can be generated from a hardware description.

4. Working on retargetable compilers also provides some insight into mutual dependencies between compilers and architectures. This insight can also help in designing a target-specific compiler. Hence, there may be a benefit from research on retargetable compilers even if only target-specific compilers will be used.

4.2.3.3 Code selection

4.2.3.3.1 Objective Let us now discuss one of the key operations of any compiler and let us see how it can be implemented in a retargetable compiler. Possibly the most important operation in any compiler is *code selection*. In code selection, machine instructions are selected.

Let us consider an example in which the program to be compiled requires the computation of an expression including two multiplies and one add (see fig. 4.21). This expression includes references to four variables. In fig. 4.21, they are denoted by nodes labeled 'MEM'.



instr.	cost	pattern
load	1	r1 := MEM
mov2	1	r2 := r1
mov3	1	r3 := r1
add	2	r1:= r1 + r2
mul	2	r1 := r1 * r2
mac	3	r1 :=(r1*r2)+r3

Abbildung 4.21: Expression (a * b) + (c * d) and instruction patterns

Each of the nodes in fig. 4.21 has to be implemented by some instruction of the available instruction set. According to fig. 4.21, we have one `add` instruction which adds the contents of registers 1 and register 2 and stores the result into register 1. There is also a `multiply` instruction using the same input and output registers. In addition, there is a `mac` instruction and a `load` instruction. Finally, there are `move` instructions for moving data from register 1 to registers 2 and 3. In fig. 4.21, the dotted ovals and circles represent instructions. These do not yet include `move` instructions that are required for moving data from the output register of one instruction to the input register of the next instruction.

Implementing all operations of the expression is equivalent to finding a cover of that expression by instructions. Even our small example allows different covers to be generated and the question is: how can we generate the cover for which the cost (representing the total number of instructions or cycles) is minimal?

A number of techniques for generating optimum covers for data-flow trees have been published in the late eighties[7].

4.2.3.3.2 Code Selection by Tree Parsing As an example, we will consider cover generation using `iburg`[FHP92]. `iburg` is publicly available from Princeton University. `iburg` models cover generation as a language parsing problem. For this purpose, the instruction set has to be described as a grammar. In this grammar, permanent memory and operators are terminals and transient registers are non-terminals. The start symbol is actually not very important and can be defined in a number of ways. Instructions correspond to rewrite rules of the grammar.

`iburg` computes an optimum cover by first annotating data-flow trees with triples representing target registers, instructions used for storing results into those target registers and the cost for storing results into target registers.

For example, MEM nodes are annotated with a triple describing the `load` operation into target register 1. Furthermore, there is another triple using register 2 as the target register and using `mov2` as the last instruction. Its cost is 2, since both the `load` and the `mov2` instruction are required. A similar triple exists for target register 3.

[7]The problem of covering data-flow graphs resulting from expressions containing common subexpressions is more complex. In this text, we will restrict ourselves to data-flow trees.

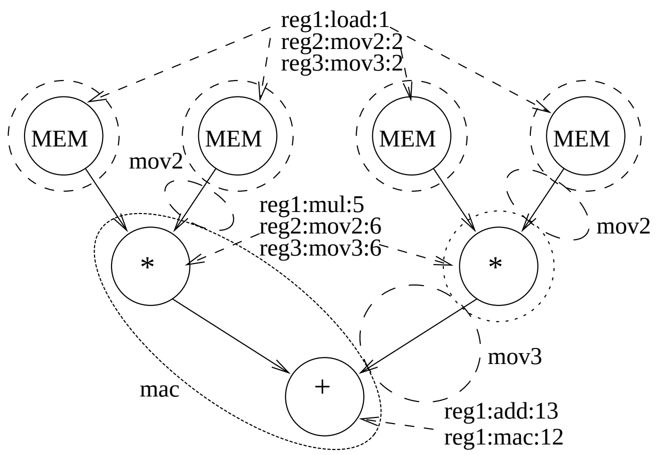


Abbildung 4.22: Annotation and final cover generated by iburg

Annotating the tree is also done for the multiply operation. The result of this operation can be stored in register 1, 2 or 3, resulting in costs of 5, 6 and 6 respectively.

Finally, there are two triples for the root node. The triple using the `mac` instruction has lowest cost and is seleted by `iburg` for the tree cover.

Interestingly enough, this cover includes ovals representing `mov2` and `mov3` instructions. These are automatically selected by `iburg`. This is an improvement over earlier approaches which used separate *data routing* phases to cope with heterogenous register sets.

4.2.3.4 Retargetable Code Generator RECORD

`iburg` alone does not make a full retargetable compiler, but it is a good building block. `iburg` is used in the retargetable compiler RECORD designed by Leupers [Leu97a].

RECORD reads the program to be compiled and the instruction set and then uses the `iburg` pattern matcher generator to generate instruction covers for this program.

In contrast to `iburg`, RECORD exploits various algebraic rules in order to find the best match between the instruction set and the program. For this purpose, algebraic rules are applied to the original expression trees. Transformed trees are also used as input to the pattern matcher and the tree leading to the cover with minimum cost is selected.

RECORD also includes comprehensive *post-processing*. It includes code compaction and address generation unit support as described in section 4.2.3 of this text. Also, there are special optimizations for digital signal processing, such as optimization for delayed signals that are needed for digital filters.

The code generated by RECORD is surprisingly good. It generates code that is more compact than the code produced by the target-specific TI compiler for the 'C25, even though RECORD does not yet include any of the standard optimizations found in compiler text books and even though it is a retargetable compiler. Using a set of DSPStone examples, RECORD outperforms the TI compiler in six cases, there are two cases in which both compilers produce the same amount of code and two cases in which the TI compiler generates more compact code.

4.2.3.5 Target Models

Retargetable compilers use a variety of target models.

RECORD is based on an HDL model of the target processor. The HDL model can be a model either of the instruction set or of the register transfer architecture or any mixture thereof. From this HDL model, RECORD computes the information that is required for the *iburg* pattern matcher. For this purpose, *instruction set extraction* [Leu97a] is used.

Due to this approach, no separate target modeling language is needed and the gap between ECAD and compiler worlds has been bridged.

4.3 Mikroarchitektur-Synthese

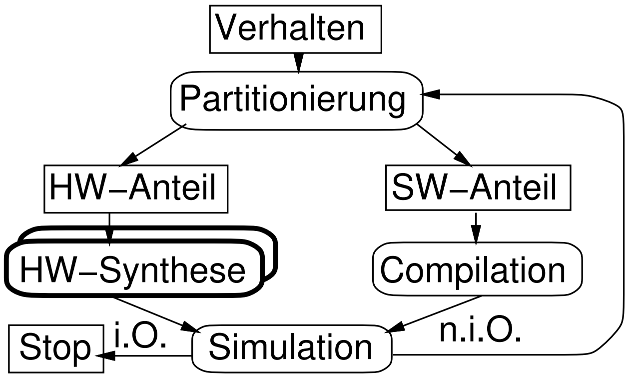
18. Vorles.

4.3.1 Kontext

Synonyme: *behavioral synthesis, high-level synthesis*[8]

Bei Beschränkung auf das Rechenwerk: *data path synthesis*

Kontext:



Ursprünge: Universität Kiel (Zimmermann et al.),
Carnegie Mellon Universität Pittsburgh (Barbacci, Thomas, Parker).

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.

4.3.2 Daten- und Kontrollflussgraphen

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:

Kommen in der Spezifikation auch Verzweigungen vor, so verwendet man einen separaten *Kontrollfluss-Graphen* zur Darstellung der Programm-Kontrolle.

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

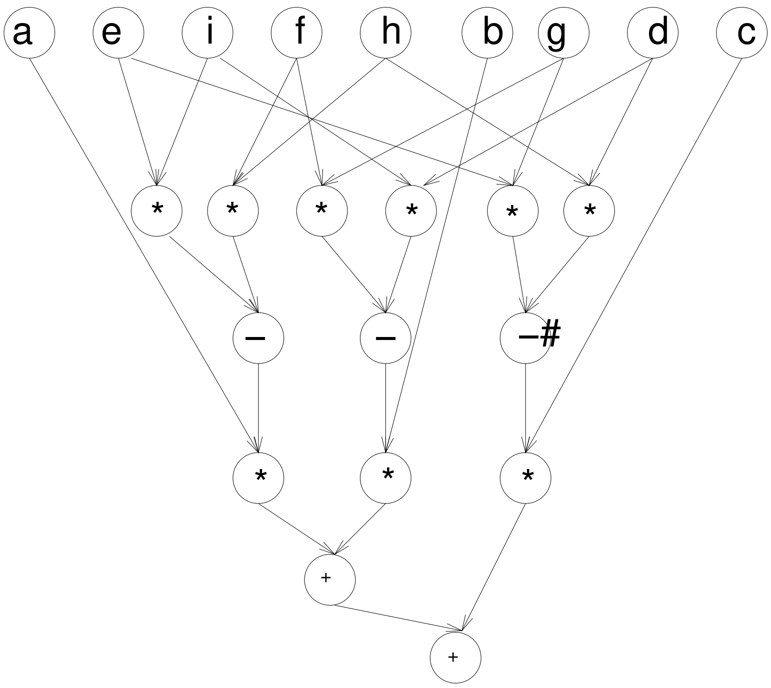


Abbildung 4.23: Datenfluss-Graph (-# = - mit vertauschten Eingängen)

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 folgende CDFG der Abb. 4.24.

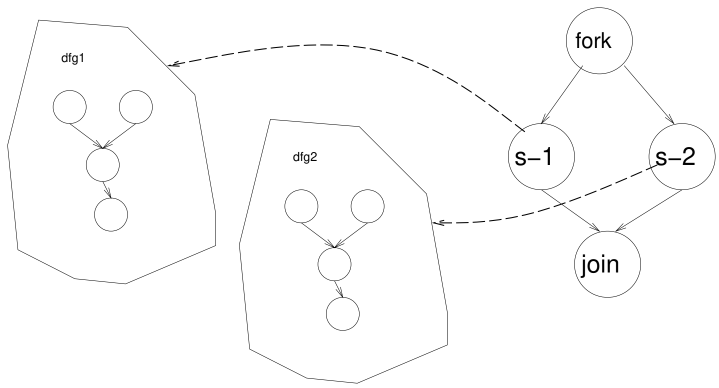


Abbildung 4.24: Kontroll/Datenflussgraph

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.

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.

4.3.3 Scheduling

Die einfachsten Scheduling-Verfahren sind *as soon as possible* (ASAP) und *as late as possible* (ALAP) scheduling.

Für das Beispiel der Determinantenberechnung führen diese Verfahren zu folgenden Ergebnissen:

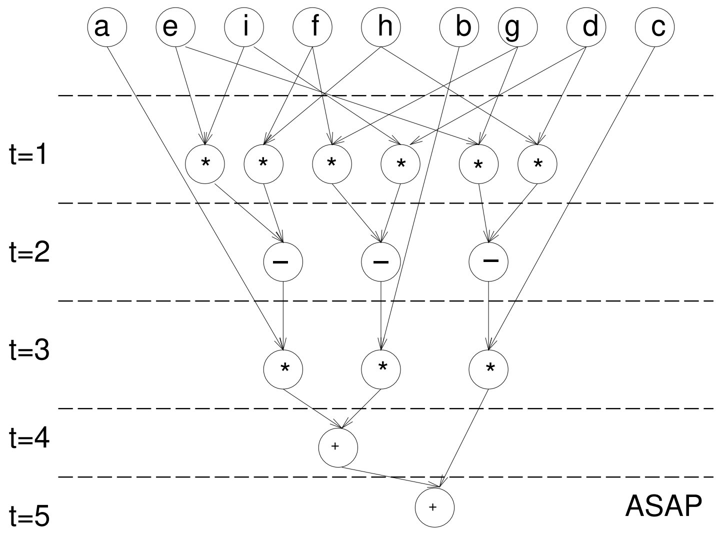


Abbildung 4.25: As-soon-as-possible Scheduling

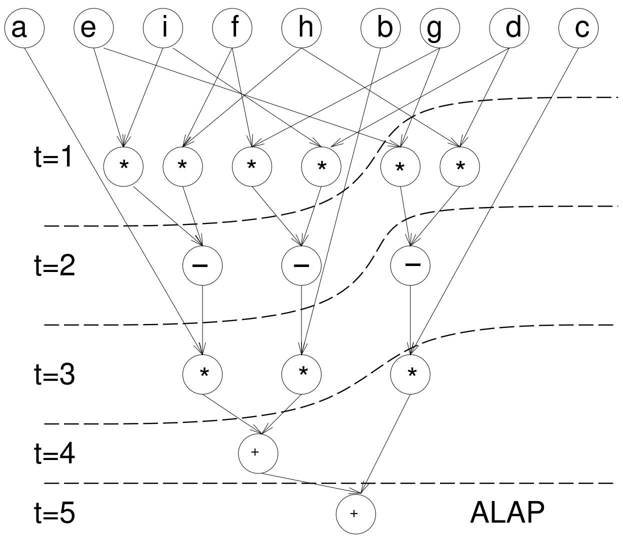


Abbildung 4.26: As-late-as-possible Scheduling

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**, da zu jedem Intervall ein Zustand des Steuerwerks gehört, dass für eine geeignete Ansteuerung aller Bausteine des Rechenwerks sorgen muss.

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

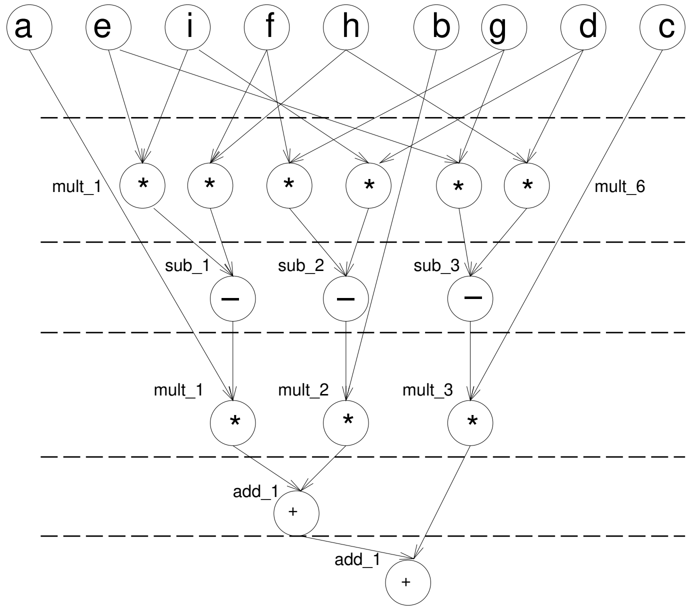
4.3.4 Allokation

Das einfachste Allokationsverfahren besteht darin, für jeden Operationstyp die maximale Anzahl von Operationen pro Kontrollschritt zu zählen. Im Beispiel 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 klar zum Ausdruck.

Komplexere Allokationsverfahren müssen u.a. Komponenten in Baustein-Bibliotheken berücksichtigen.

4.3.5 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. 4.27).

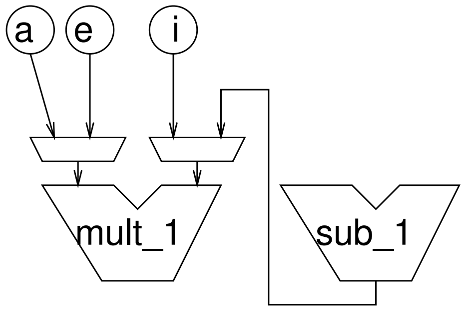


Abbildung 4.27: Verbindungen als Folge einer Zuordnung von Bausteinen zu Operationen

Komplexere Zuordnungsverfahren müssen versuchen, die entstehenden Verbindungen zu optimieren und dabei auch komplexere Komponenten (ALUs usw.) berücksichtigen.

Ein großer Schritt..... wir versuchen, alles auf einmal zu lösen!

4.3.6 Integration von Scheduling, Allokation und Assignment in OSCAR

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

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. 4.28).

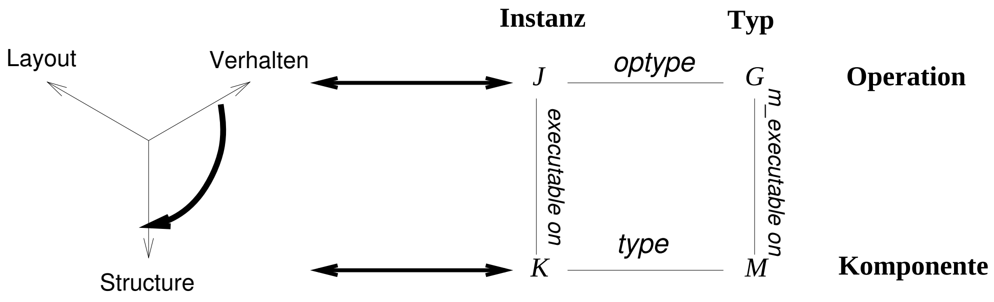


Abbildung 4.28: Notation in OSCAR-Modellen

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 Relation *m_executable on* beschrieben:

$\forall m \in M, g \in G : g$ *m_executable on* $m \iff$ Bausteintyp m kann g ausführen.

Aus dieser Relation leiten wir die folgende Relation ab:

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

Most synthesis tools do not only generate structure. They also generate a binding between operations and *control steps* in which they are started. This is also the case for all architectural synthesis tools we are aware of. We will use i to denote a certain control step and I to denote the set of all control steps. Note that we consider *multi-cycle operations* (operations which do not terminate in the same control step).

Die meisten Werkzeuge generieren nicht nur Struktur. Sie generieren auch eine Bindung zwischen Operationen und **Kontrollschritten** (engl. *control steps*), in denen sie gestartet werden. Dies trifft auf alle uns bekannten Mikrosynthesewerkzeuge zu. Wir werden Indizes $i \in I$ benutzen, um Kontrollschritte zu bezeichnen. Modell lassen in der Regel zu, sog. *multi-cycle operations* zu verwenden, welche mehrere Kontrollschritte benötigen. Der Einfachheit halber nehmen wir an, dass $l(k)$ für jeden Baustein die benötigte Anzahl an Kontrollschritten bezeichnet (d.h., vereinfachend nehmen wir an, dass alle von einem Baustein k ausführbaren Operationen dieselbe Anzahl an Kontrollschritten benötigen; OSCAR kommt ohne diese Annahme aus).

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

$$(4.9) \quad x_{i,j,k} = \begin{cases} 1, & \text{Wenn Operation } j \text{ auf der Baustein-Instanz } k \text{ im Kontrollschritt } i \text{ gestartet wird} \\ 0, & \text{sonst} \end{cases}$$

Als Hilfvariablen werden die folgenden b_k benötigt:

$$(4.10) \qquad b_k = \begin{cases} 1, & \text{falls die Bausteinintanz } k \text{ benötigt wird} \\ 0, & \text{sonst} \end{cases}$$

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:

$$\forall j \in J : \sum_{i \in R(j)} \sum_{\substack{k \in K \\ j \text{ executable on } k}} x_{i,j,k} = 1$$

2. *Resource assignment constraints*: Diese stellen sicher, dass jede Komponente k höchstens alle $l(k)$ 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.

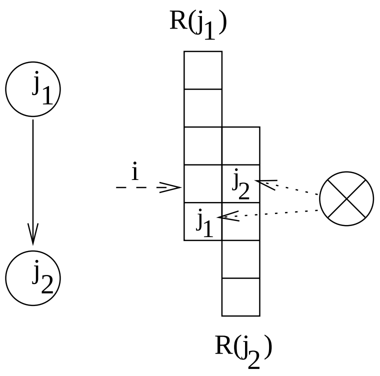


Abbildung 4.29: Verbotene Bindung bei datenabhängigen Operationen

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:

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

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

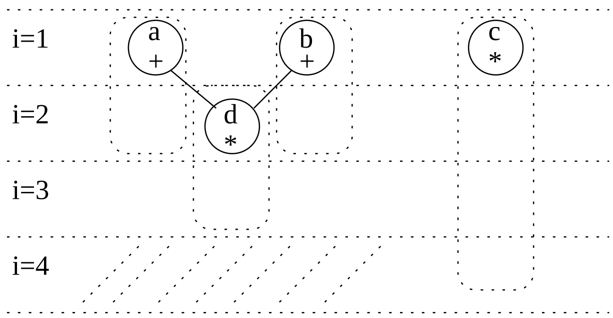


Abbildung 4.30: Einfacher Datenfluss-Graph

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

Für die erste Zuweisung seien drei, für die zweite vier Kontrollschritte zulässig (siehe Abb. 4.30). Wir benutzen dabei im Folgenden die Indizes a, b, c und d zur Indizierung der Operationen, wie in der Abbildung angedeutet.

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 4.2).

Typ	Operationen
1	+
2	*
3	+, *

Tabelle 4.2: Einfache Bausteinbibliothek

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

Wir nehmen an, dass wir maximal zwei Exemplare pro Bausteintyp benötigen und ordnen diesen gemäß Abb. 4.31 Indizes zu.

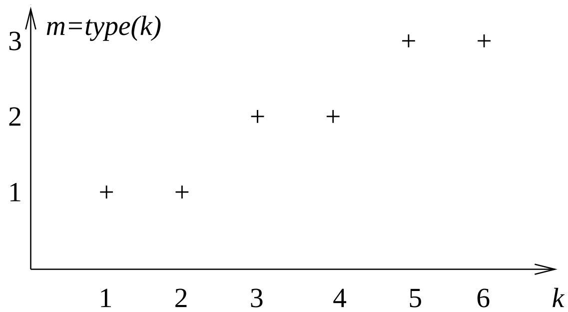


Abbildung 4.31: Indexreservierung

Da diese Programme zur ganzzahligen Programmierung die Zielfunktionen meist 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):

$$\begin{array}{l|llll|llll}
i=1 & x_{1a1} & +x_{1b1} & \le b_1 & & x_{1a2} & +x_{1b2} & \le b_2 \\
i=2 & x_{2a1} & +x_{2b1} & \le b_1 & & x_{2a2} & +x_{2b2} & \le b_2
\end{array}$$

Für $k = 3, 4$ (Multiplizierer):

$$\begin{array}{l|llll|lll}
i=1 & x_{1c3} & & \le b_3 & & x_{1c4} & & \le b_4 \\
i=2 & x_{2c3} & +x_{2d3} & \le b_3 & & x_{2c4} & +x_{2d4} & \le b_4 \\
i=3 & x_{3c3} & +x_{3d3} & \le b_3 & & x_{3c4} & +x_{3d4} & \le b_4 \\
i=4 & +x_{4c3} & & \le b_3 & & x_{4c4} & & \le b_4
\end{array}$$

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

$$\begin{array}{l|lllll|lllll}
i=1 & x_{1a5} & +x_{1b5} & +x_{1c5} & & \le b_5 & +x_{1a6} & +x_{1b6} & +x_{1c6} & & \le b_6 \\
i=2 & x_{2a5} & +x_{2b5} & +x_{2c5} & +x_{2d5} & \le b_5 & +x_{2a6} & +x_{2b6} & +x_{2c6} & +x_{2d5} & \le b_6 \\
i=3 & x_{3c5} & +x_{3d5} & & & \le b_5 & +x_{3c6} & +x_{3d6} & & & \le b_6 \\
i=4 & x_{4c5} & & & & \le b_5 & +x_{4c6} & & & & \le b_6
\end{array}$$

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. $x_{2d3} + x_{2d4} + x_{2d5} + x_{2d6} + x_{2a1} + x_{2a2} + x_{2a5} + x_{2a6} \le 1$ (Schritt 2 darf nicht a und d enthalten);
2. $x_{2d3} + x_{2d4} + x_{2d5} + x_{2d6} + x_{2b1} + x_{2b2} + x_{2b5} + x_{2b6} \le 1$ (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 Verwendund 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 4.3.

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 4.3: 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.

Kapitel 5

Logik- und Controllersynthese

5.1 Controller-Synthese

5.1.1 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. 5.1) benötigt.

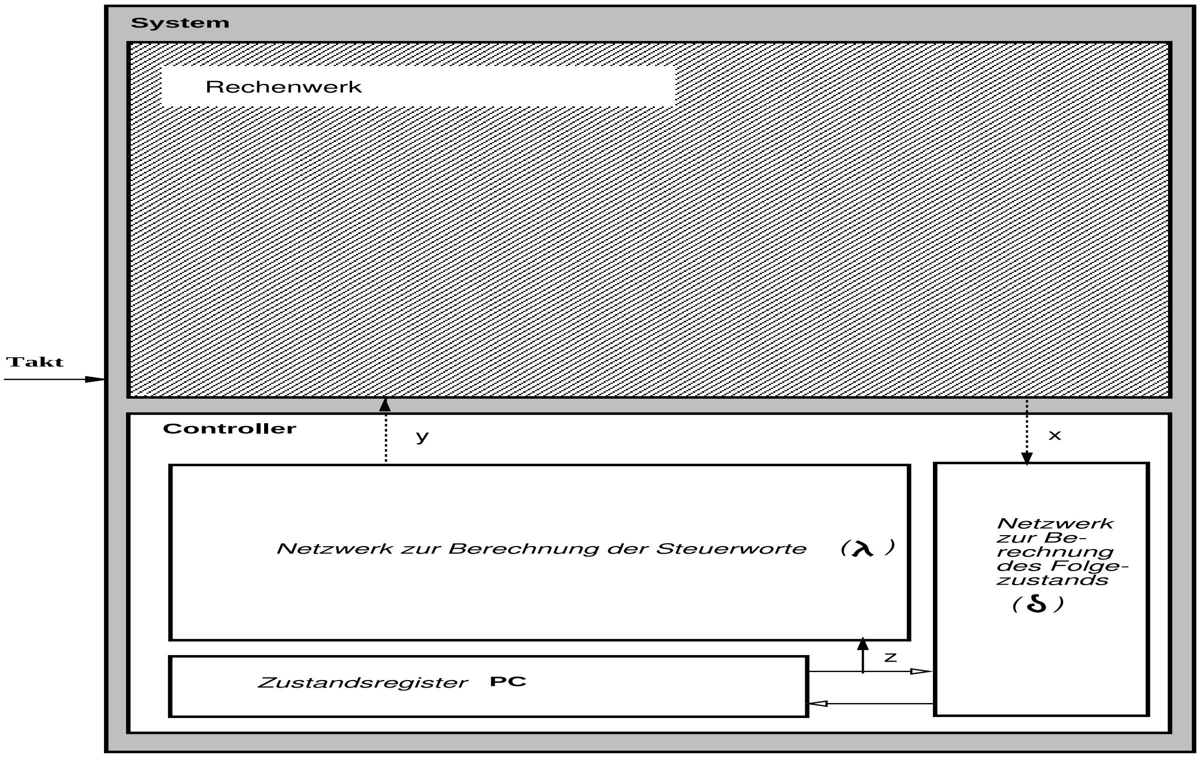


Abbildung 5.1: Aufteilung des Entwurfs in Rechenwerk und Controller

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.

5.1.2 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. 5.2 sind die Zustände `a` und `b` äquivalent.

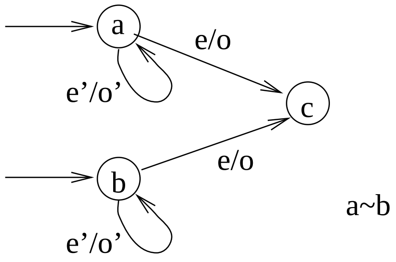


Abbildung 5.2: Äquivalente Zustände

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. 5.3 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.

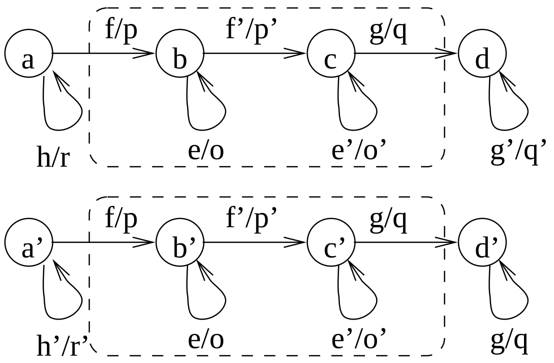


Abbildung 5.3: Unterprogramme in Automaten

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.

5.1.3 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. 5.4).

   $(a_n, .., a_1)$, $(b_n, .., b_1)$ 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 $e(a_n...a_2a_1 \vee$

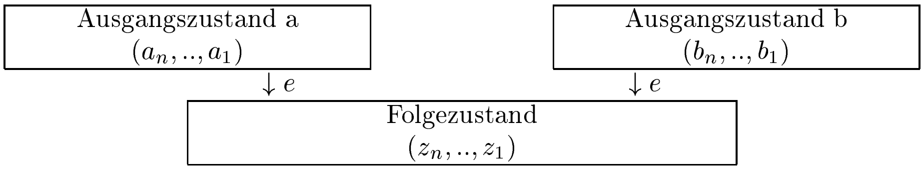


Abbildung 5.4: Kodierung bei “Joins”

$b_n..b_2b_1)$ 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. 5.5 angegebenen Kodierung der Fall.

Abbildung 5.5: 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. 5.9).

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

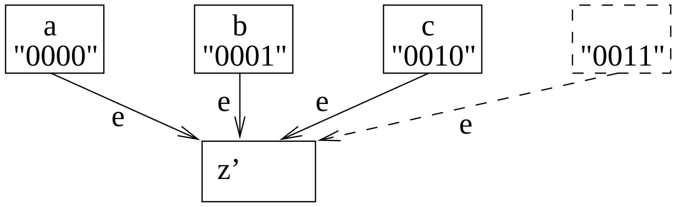


Abbildung 5.6: m=3 Ausgangszustände

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. 5.6).

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. 5.7).

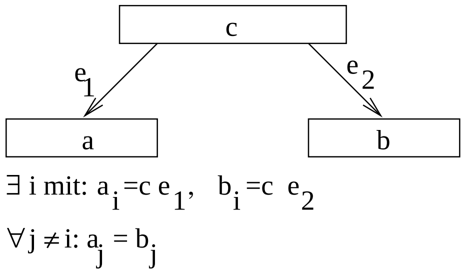


Abbildung 5.7: Fork-Regel

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.

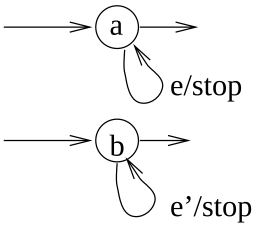


Abbildung 5.8: Zustände, welche die Ausgabe Stop erzeugen

> Beispiel: In Abb. 5.8 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.

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. 5.9). Anschließend wird allen mit diesem Knoten verbundenen Knoten ein Code zugeordnet. Dieser Prozeß wird fortgeführt, bis alle Knoten erreicht sind ("breadth-first search").

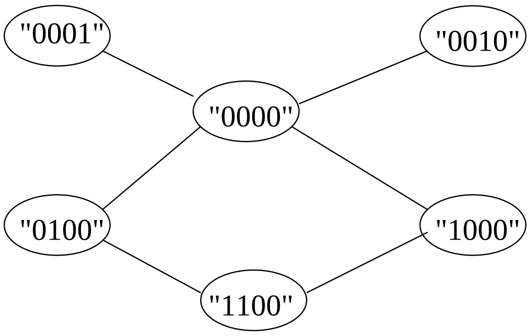


Abbildung 5.9: Graph der gewünschten einschrittigen Kodierungen

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 eines der derzeit besten Werkzeuge 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 Buches 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**

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

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 entwickelte **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].

5.1.4 Realisierung der Ausgabefunktion

5.1.4.1 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. Eine beträchtliche Einsparung ergibt sich häufig schon durch einfache Techniken wie z.B.: Erkennen konstanter Ausgabesignale, Erkennen gleicher oder invertierter Ausgabesignale, Erkennen von Ausgabesignalen, die sich mit logischen Operationen ($\wedge$, $\vee$) 1-stufig auseinander ableiten lassen. Durch einfache Schaltungen am Ausgang des δ-Netzwerks läßt sich das δ-Netzwerk selbst so vereinfachen.
3. Ferner lassen sich symbolische Minimierungstechniken einsetzen, um eine Kodierung der Ausgabewerte zu erreichen.

5.1.4.2 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.

   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 "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]). In der Praxis kommt sie aber wegen großen Hardwareaufwandes nicht vor.
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 Übersetzern.
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

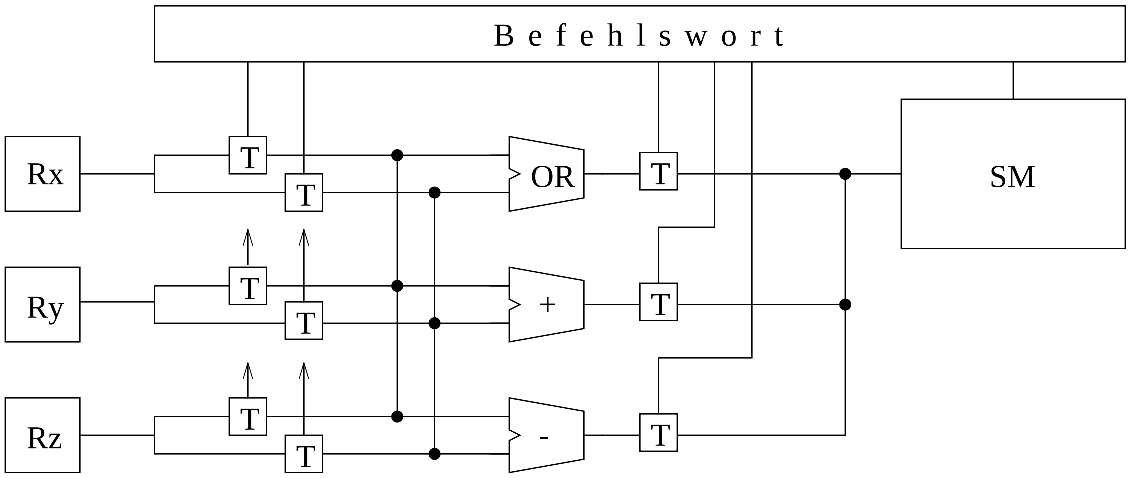


Abbildung 5.10: Direct control (T:Treiber, z.B.Tristate-Treiber)

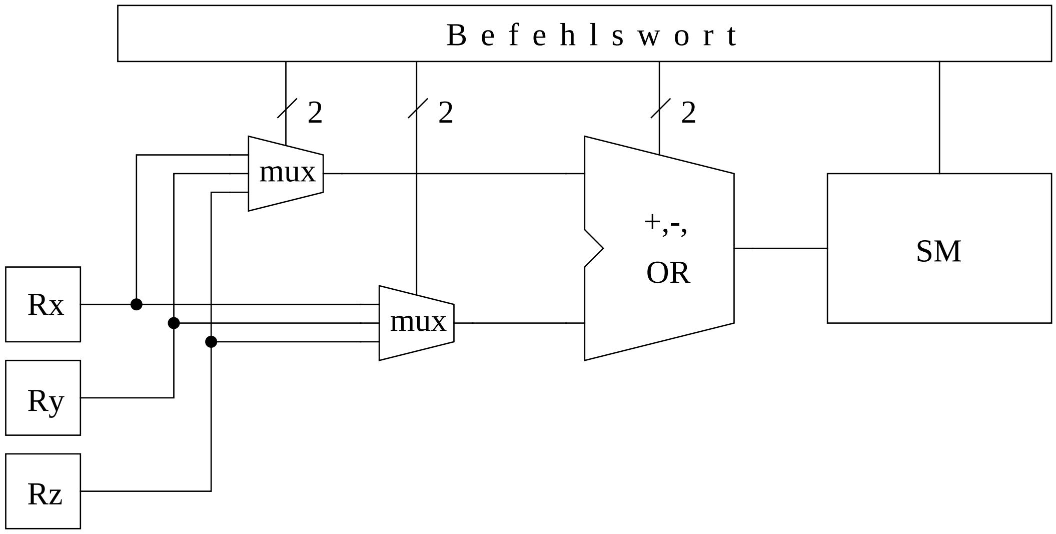


Abbildung 5.11: Direct encoding

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.

4. *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. 5.13).

   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.

5. *Two level encoding*

   Unter dem Begriff "two level encoding" faßt man die beiden Fälle "bit steering" und "format shifting" zusammen. Hängt die Bedeutung eines Befehlsfeldes von einem anderen Befehlsfeld ab, so spricht man von "bit steering".

   Hängt sie von der Belegung eines "residual control" – Registers ab, so bezeichnet man dies als "format shifting" (vgl. Abb. 5.14).

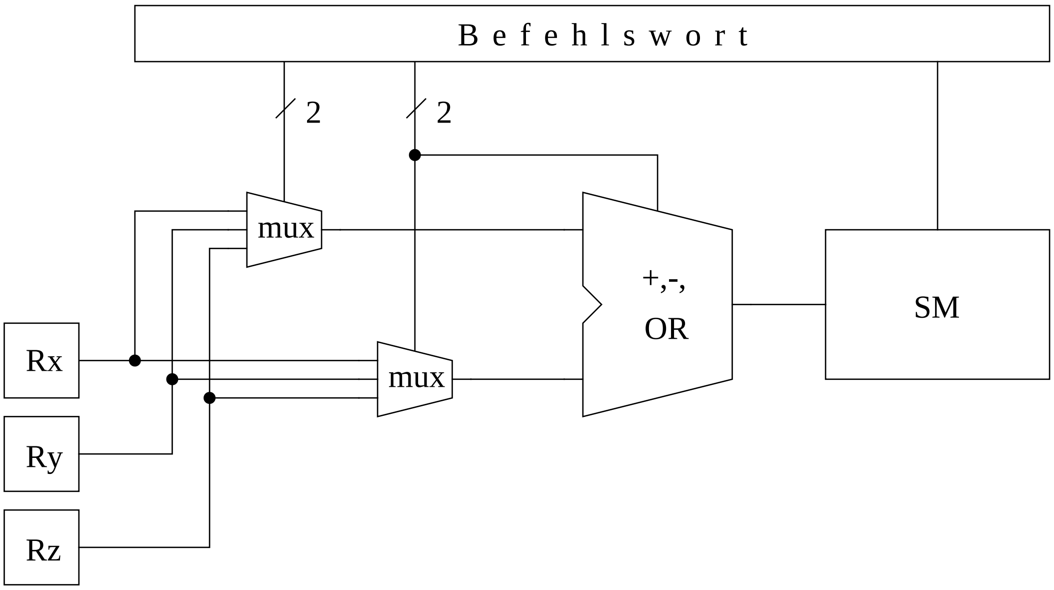


Abbildung 5.12: Direct encoding mit sharing

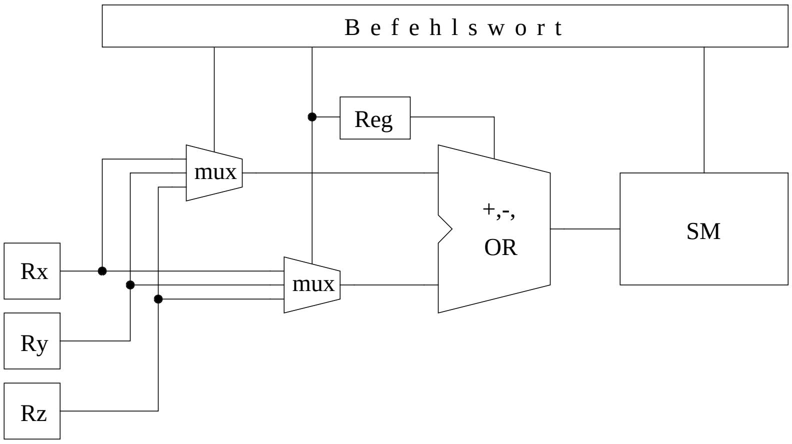


Abbildung 5.13: Residual control

6. *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. 5.15 zeigt die Hardware-Organisation unter Einschluß der Logik zur Bestimmung des Folgezustandes.

   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. 5.16 zeigt die Benutzung dieser Hardware zur Realisierung eines konkreten Flußgraphen.

[2]Eine ähnliche Idee stellt die indirekte Adressierung von Farbwerten in einer *color lookup table* dar (siehe Stammvorlesungen Rechnerarchitektur oder Graphische Systeme).

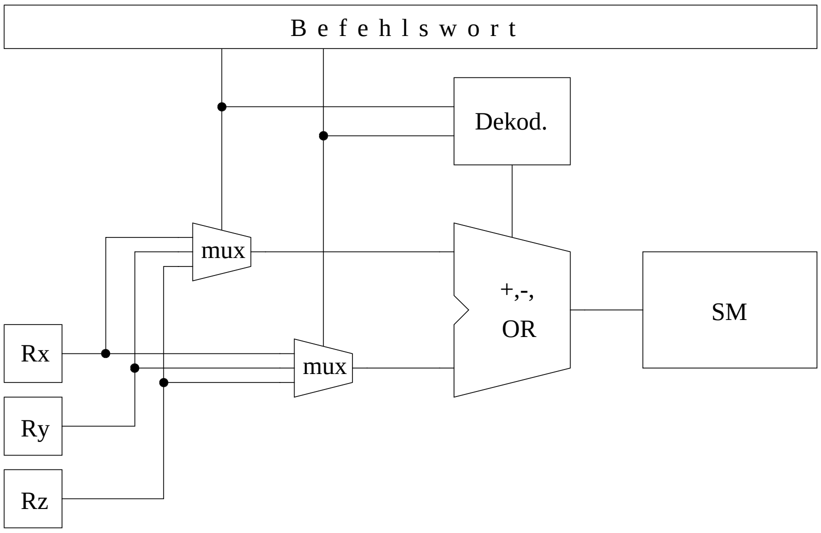


Abbildung 5.14: Bit steering

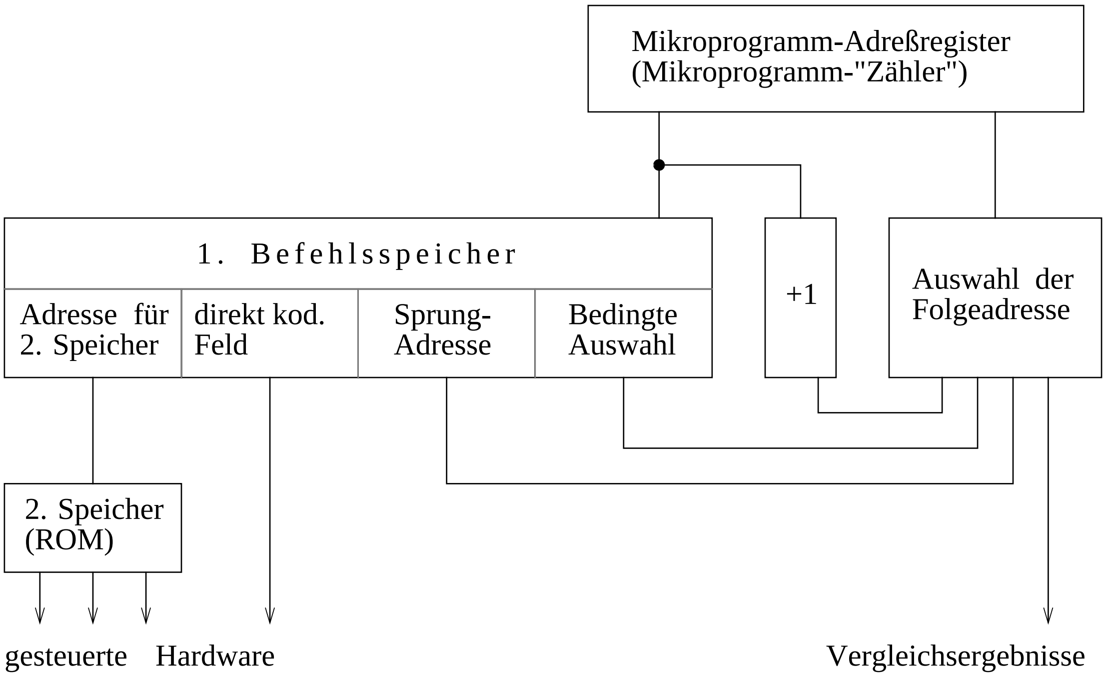


Abbildung 5.15: Two level control store

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.

5.1.4.3 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

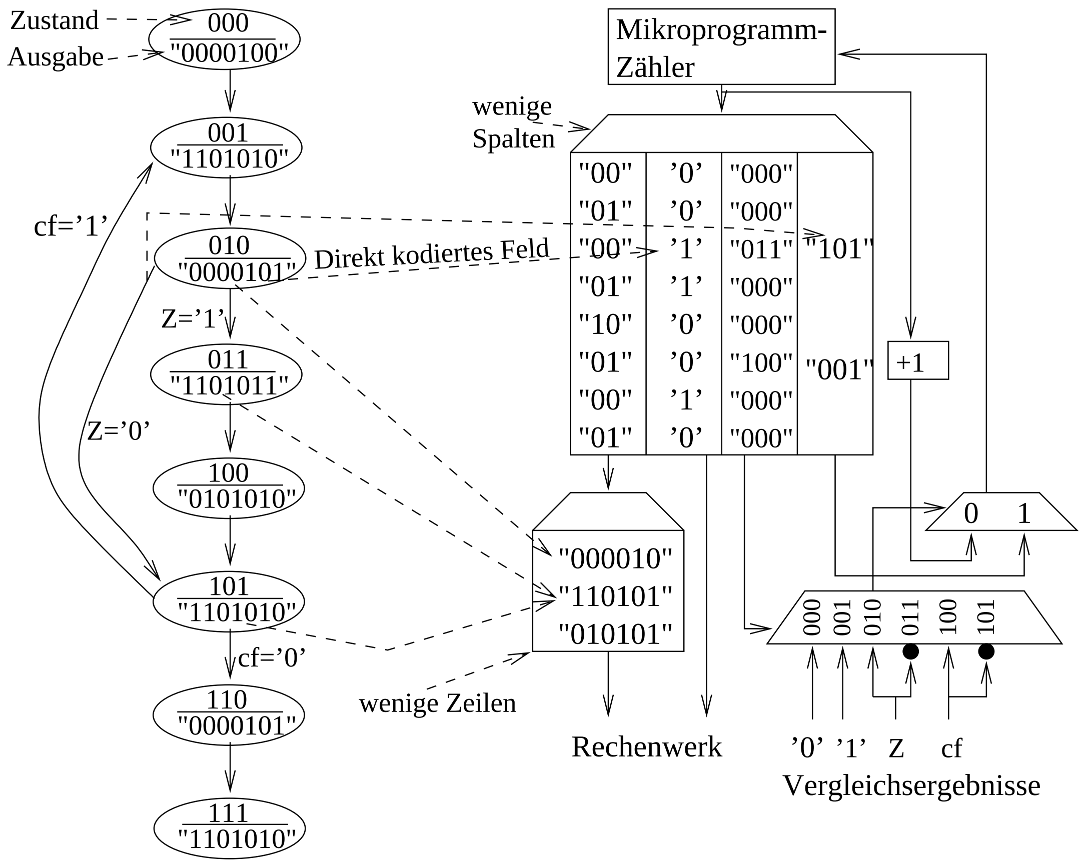


Abbildung 5.16: Realisierung eines konkreten Flußgraphen

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. 5.17.

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 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.

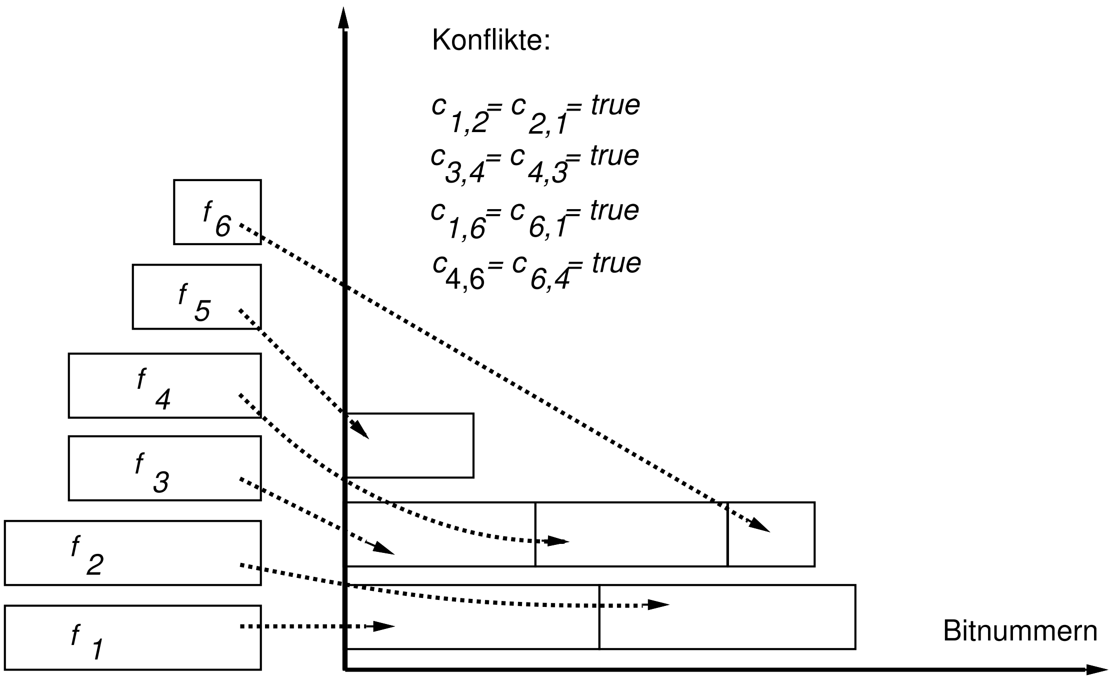


Abbildung 5.17: Wortbreiten-Reduktion bei TODOS

5.1.5 Realisierung von StateChart-Spezifikationen mit kommunizierenden Automaten

StateChart erlaubt die Beschreibung von nebenläufigen (nach StateChart-Sprechweise "orthogonalen") Automaten. Die Realisierung in Form eines einzigen Automaten führt durch die Bildung des Kreuzproduktes der Zustände zur Zustandsexplosion.

Als Alternative hat Harel [DH89] vorgeschlagen, kommunizierende (nebenläufige) Automaten zuu verwenden (engl. *communicating finite state machines*, CFSMs).

Zu diesem Zweck werden vier Typen von Kontrollsignalen definiert:

- entered x: Eine (FSM-) Maschine zeigt einer Maschine x auf dem nächstniedrigeren Niveau an, daß sie betreten werden soll.
- left x: Eine FSM-Maschine zeigt einer Maschine auf nächstniedrigerem Niveau an, daß sie terminieren soll.
- leave x: Eine (FSM-) Maschine x zeigt an, daß sie von sich aus terminiert.
- enter x: Eine (FSM-) Maschine zeigt einer Maschine auf dem nächstniedrigeren Niveau an, daß der konkrete Zustand x betreten werden soll.

Beispiel 1:

Abb. 5.18 zeigt ein hierarchisches Zustandsdiagramm.

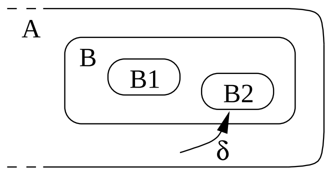


Abbildung 5.18: Hierarchisches StateChart-Diagramm

Zwischen diesen Zuständen sind die in Abb. 5.19 gezeigten Signale auszutauschen.

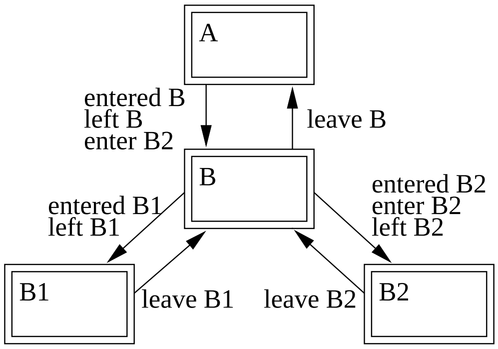


Abbildung 5.19: Austausch von Signalen im Beispiel 1

Beispiel 2:

Abb. 5.20 zeigt ein weiteres hierarchisches Zustandsdiagramm.

Den Automaten D zeigt die Abb. 5.21.

Exemplarisch seien die Übergänge des Automaten D erläutert:

- Kanten left D: wann immer D von außen signalisiert bekommt, daß D zu verlassen ist, werden alle Teilzustände von D verlassen.
- Kante entered D: wenn D betreten wird, sorgt diese Kante für die Transition in den Default-Zustand C.
- Kante $\delta 2$: Transition eines auf der Ebene von D bekannten Signals.
- Kante leave C_b: Internes Signal, erzeugt im Teilzustand C und dort wiedrum erzeugt im Teilzustand A1, und zwar aufgrund des externen Signals $\delta 3$. leave C_b führt zur Erzeugung von enter B12, was in Zustand B1 genutzt wird.
- Kante leave C_a: Übergang aufgrund des Signals $\delta 1$ in FSM C.

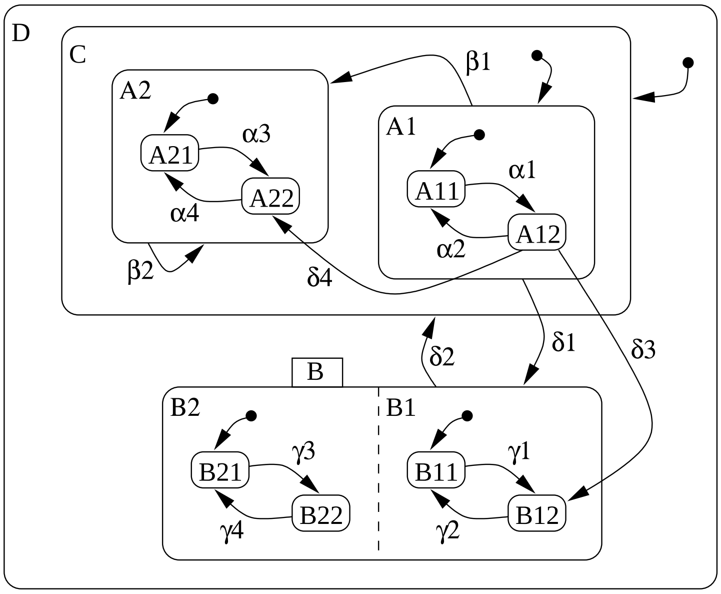


Abbildung 5.20: Hierarchisches StateChart-Diagramm

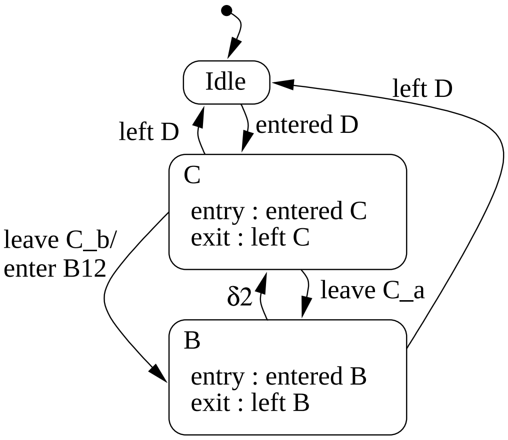


Abbildung 5.21: Automat D

Ausgaben:

- Ausgaben an den Übergängen wie oben am Beispiel leave C_b zu sehen.
- Enter- und Exit-Aktionen in höherer Hierarchie signalisieren jeweils Start/Terminierung auf der darunter liegenden Ebene.

Zu diesem Beispiel gehören insgesamt sechs weitere endliche Automaten mit mehr als einem Zustand (siehe Abb. 5.22 bis 5.27) sowie acht Automaten der Zustände $B_{i,j}$, $A_{,j}$.

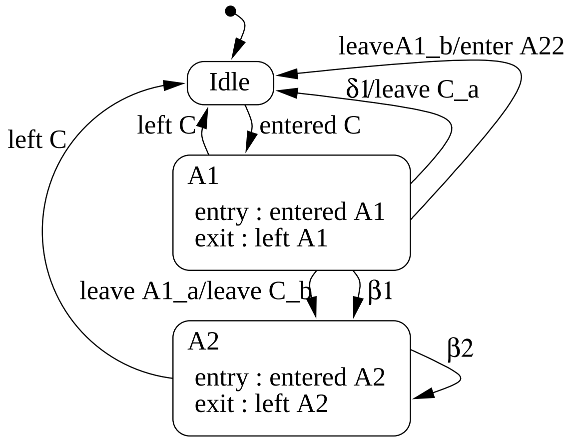


Abbildung 5.22: Automat C

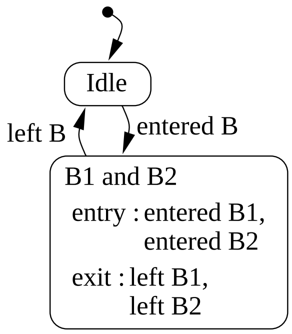


Abbildung 5.23: Automat B

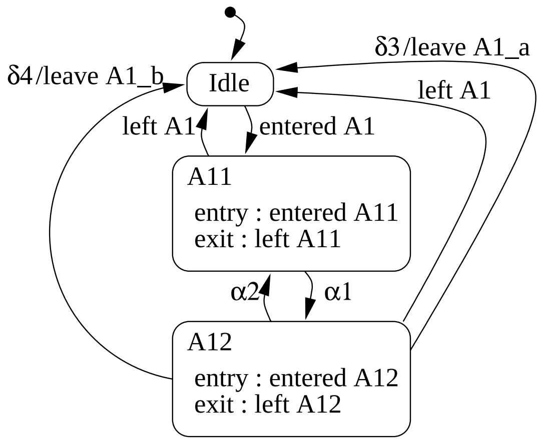


Abbildung 5.24: Automat A1

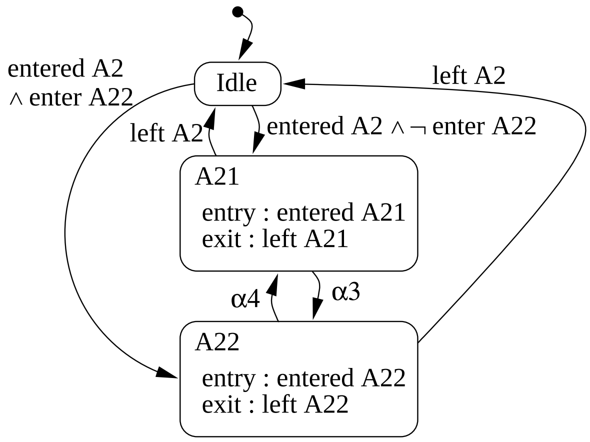


Abbildung 5.25: Automat A2

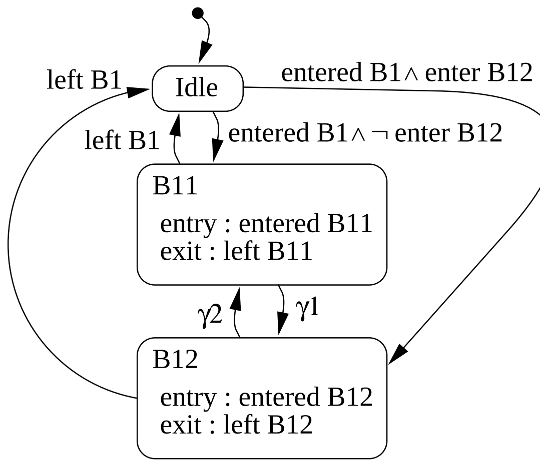


Abbildung 5.26: Automat B1

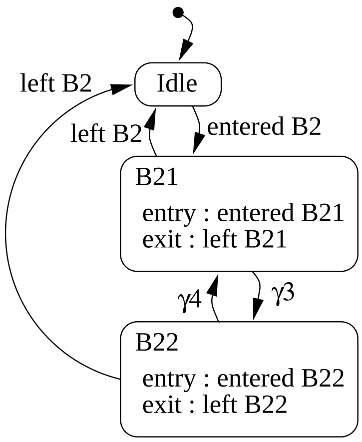


Abbildung 5.27: Automat B2

Den Austausch von Signalen und die Baumstruktur der Automaten zeigt die Abb. 5.28.

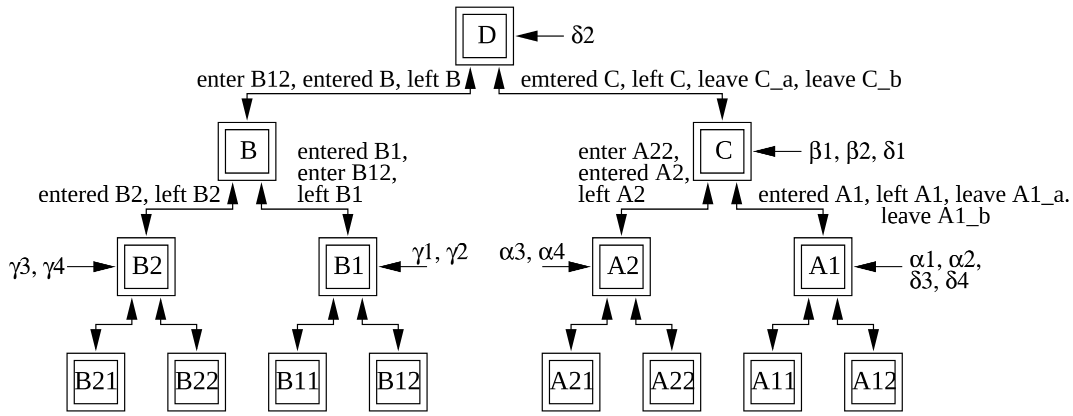


Abbildung 5.28: Baumstruktur der Automaten

Diese Automaten lassen sich durch das in Abb. 5.29 gezeigte Layout realisieren.

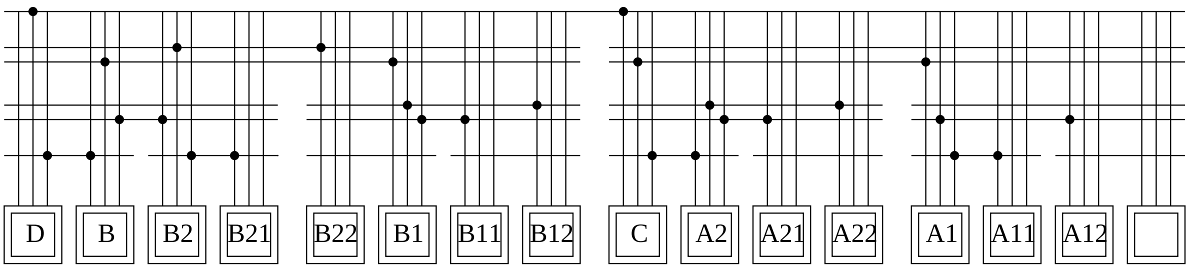


Abbildung 5.29: Layout der Automaten

Effizienter ist das optimierte Layout nach Abb. 5.30.

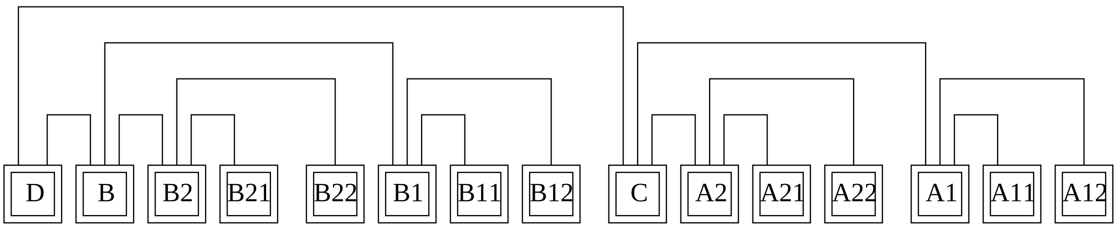


Abbildung 5.30: Optimiertes Layout der Automaten

Bei Beschränkung auf Moore-Automaten gibt es allerdings noch ein Timing-Problem: Signale werden pro Takt um eine Hierarchiestufe weitergeleitet. Im Falle der Abb. 5.31

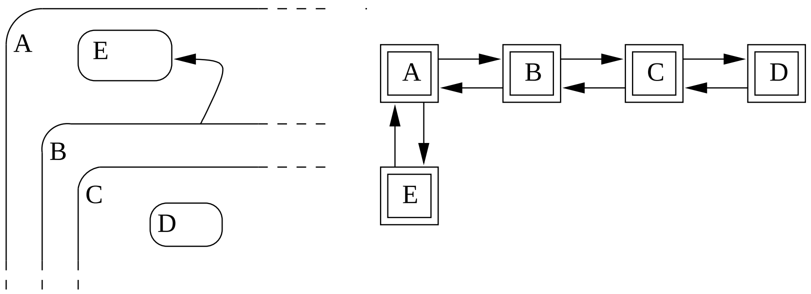


Abbildung 5.31: Timing-Problem der Realisierung

könnte also E bereits aktiv sein, obwohl D noch nicht terminiert hat. Als Abhilfe können die Kommunikationssignale wie bei einem Mealy-Automaten direkt Übergänge auslösen.

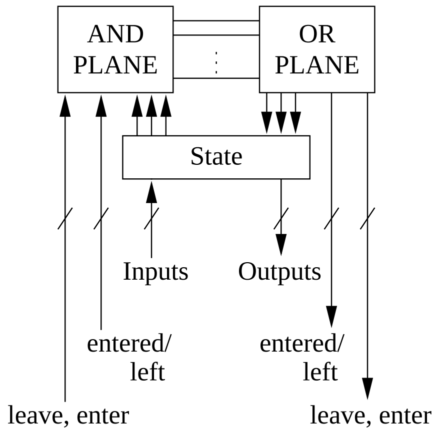


Abbildung 5.32: Mealy-Verhalten der Hierarchie-Signale

Zusammenfassend kann man festhalten, daß mit kommunizierenden endlichen Automaten die Zustandsexplosion vermieden werden kann. Im allgemeinen (insbesondere bei der Realisierung orthogonaler StateCharts-Zustände) sind dabei mehrere Automaten gleichzeitig aktiv.

5.2 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.

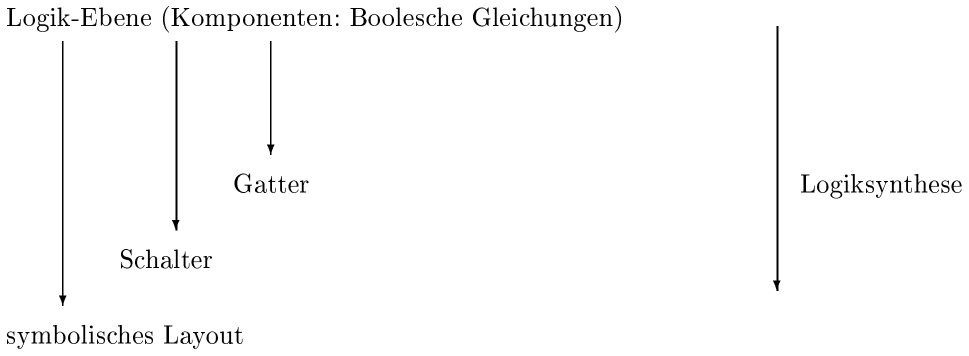


Abbildung 5.33: Zum Begriff "Logiksynthese"

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. 5.33).

5.2.1 Klassische Minimierungstechniken für 2-stufige Logik

5.2.1.1 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:

$$\begin{aligned} and(i,j) &= 0\ , && \text{falls in Produktterm } i \text{ die Variable } x_j \text{ invertiert vorkommt } ^3, \\ and(i,j) &= 1\ , && \text{falls in Produktterm } i \text{ die Variable } x_j \text{ nicht-invertiert vorkommt,} \\ and(i,j) &= 2\ , && \text{falls in Produktterm } i \text{ die Variable } x_j \text{ nicht vorkommt.} \end{aligned}$$

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

$$\begin{aligned} or(i,j) &= 0\ , && \text{falls der Produktterm } i \text{ in die Berechnung von } y_j \text{ nicht eingeht,} \\ or(i,j) &= 1\ , && \text{falls der Produktterm } i \text{ in die Berechnung von } y_j \text{ eingeht.} \end{aligned}$$

Beispiel:

Gegeben sei das PLA mit der Individualität

and	*or*
200	11
121	01
122	10

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

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:

$$\begin{aligned} y_1 &:= \overline{x_2}\,\overline{x_3} + x_1 \\ y_2 &:= \overline{x_2}\,\overline{x_3} + x_1\, x_3 \end{aligned}$$

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.

5.2.1.2 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.

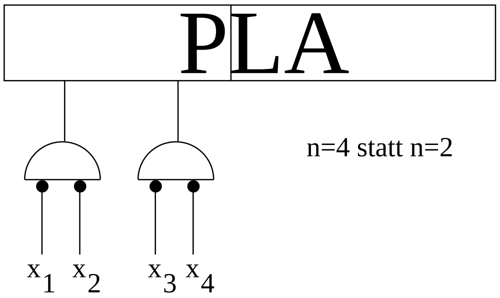


Abbildung 5.34: Reduktion der Spaltenzahl, falls Funktion von x_1x_2 und x_3x_4 abhängt

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

5.2.1.3 Streichen von Variablen in Termen

Eine der schnellen Vereinfachungstechniken besteht in der Suche nach Variablen, welche in einzelnen Termen redundant sind[4]. 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] Dies entspricht in KV-Diagrammen dem Übergang auf jeweils größere Blöcke von Einsen.

5.2.1.4 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[5]. Formal wird dies durch das folgende Lemma ausgedrückt:

Lemma: Zeile row ist redundant $\Longleftrightarrow$
$\forall c \in [1..m]$ 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.

5.2.1.5 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[6]:

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

[5] Dies entspricht in KV-Diagrammen dem Weglassen eines Blockes von Einsen, welche schon durch andere Blöcke überdeckt werden.
[6] Darstellung nach Pusch [Pus90].

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. 5.35 zeigt diesen Vorgang für ein Beispiel anhand eines KV-Diagramms (das Diagramm wird nur zur Verdeutlichung, nicht zur Minimierung benutzt).

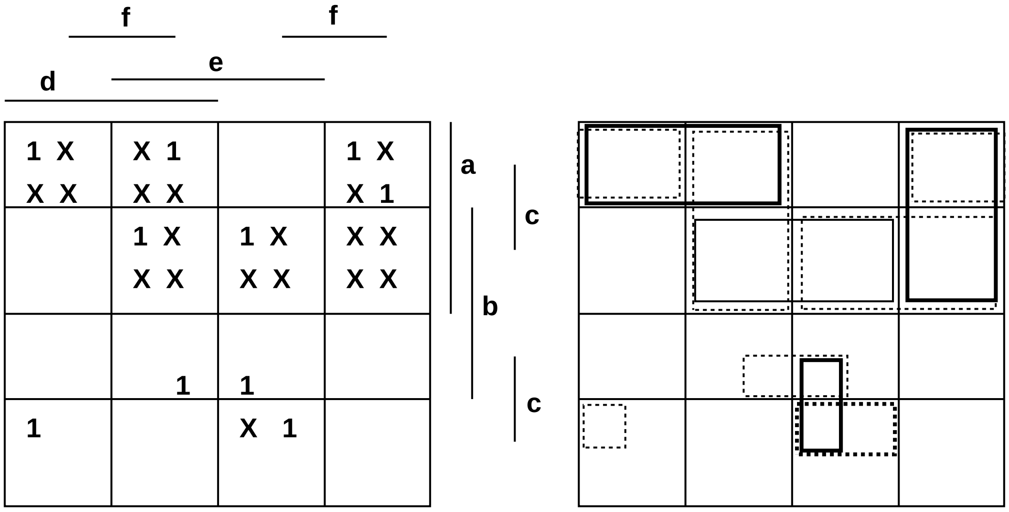


Abbildung 5.35: ESPRESSO: a) Ausgangsfunktion; b) Expansion

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 Überdeckung sorgt, heißt **essentieller Primterm**.
   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. 5.36).

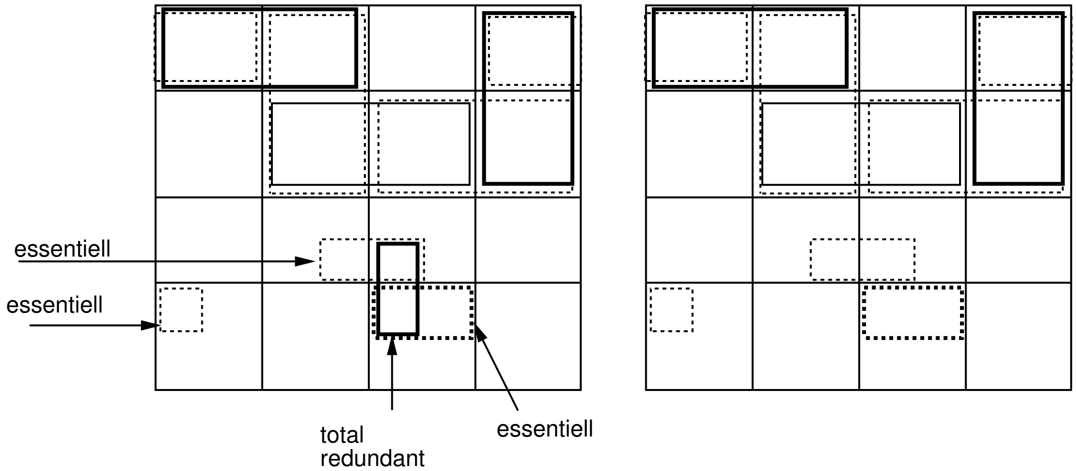


Abbildung 5.36: ESPRESSO: a) Essentielle und total redundante Terme; b) Elimination

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. 5.37 a)).

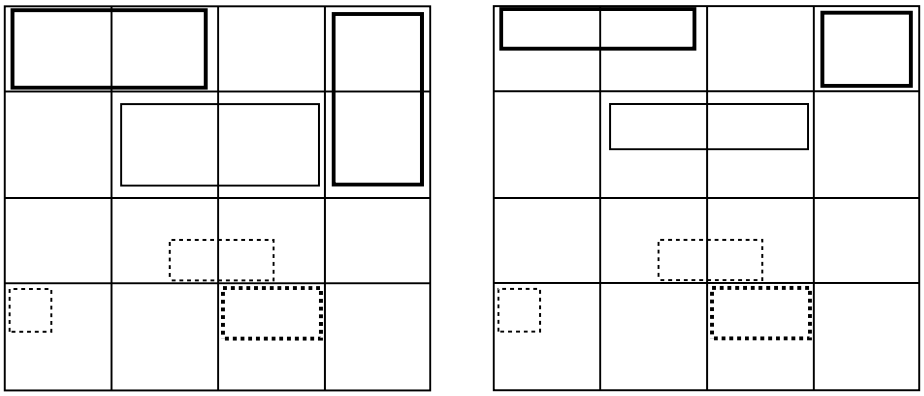


Abbildung 5.37: ESPRESSO: a) Überdeckung; b) Reduktion

(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. 5.37 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].

5.2.1.6 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.

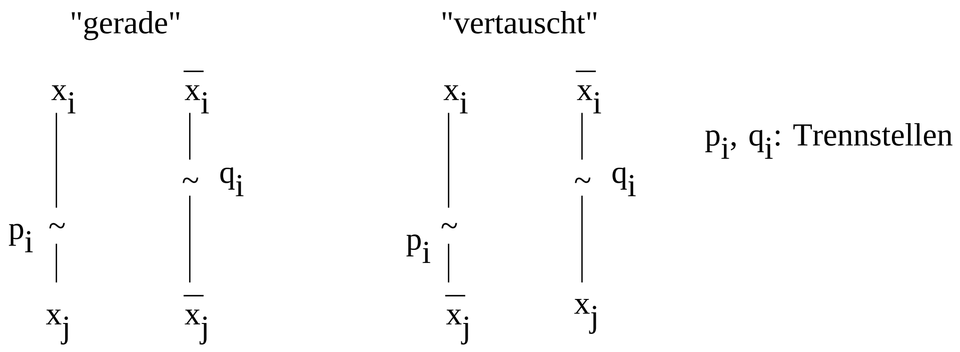


Abbildung 5.38: Möglichkeiten der Zuführung von zwei Variablen

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.

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. 5.39 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 5.39: 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.5.40 (links).

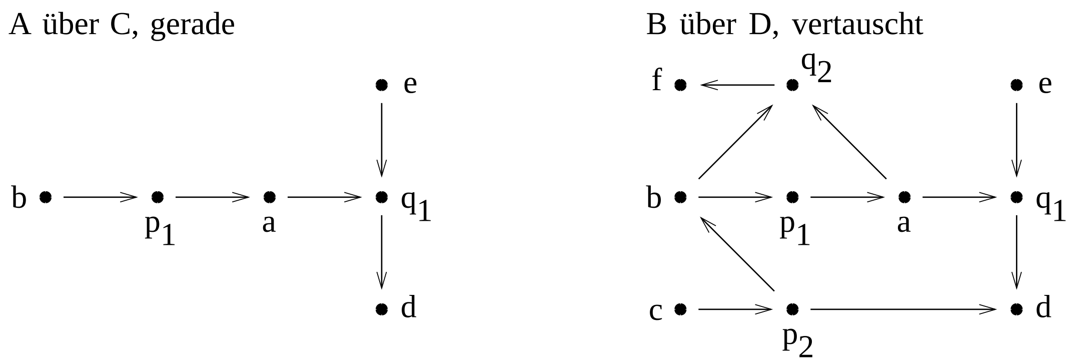


Abbildung 5.40: Graph G nach Paarung von A und C

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.5.40.

Weitere Paarungen sind mangels weiterer Eingangsvariablen nicht möglich. Abb.5.41 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[7] von Lewandowski und C.L.Liu [LC84] sowie von Grass [Gra82].

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

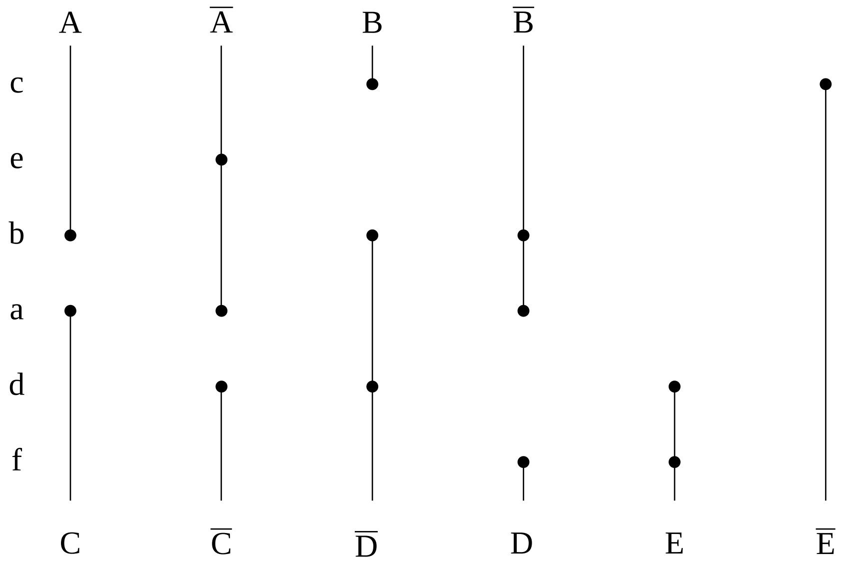


Abbildung 5.41: PLA nach Faltung der AND–Plane

5.2.1.7 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.5.42).

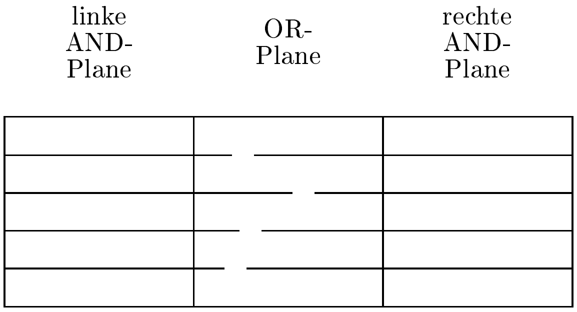


Abbildung 5.42: Faltung der OR–Plane

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

5.2.1.8 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.

5.2.2 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.

5.3 Binary Decision Diagrams (BDDs)

5.3.1 Begriffe

Ein gravierender Nachteil klassischer Funktionsdarstellungen liegt in den benutzten Datenstrukturen. Üblicherweise sind dies disjunktive Normalformen (engl. *sum of products*). Der Nachteil disjunktiver Normalformen liegt in ihrer fehlenden Eindeutigkeit: es gibt üblicherweise sehr viele disjunktive Normalformen, welche dieselbe Boolesche Funktion darstellen. Als Folge davon ist schon der Test, ob zwei disjunktive Normalformen dieselbe Boolesche Funktion darstellen, NP-vollständig[8]. Wahrheitstabellen und KV-Diagramme wachsen exponentiell mit der Anzahl der Variablen und sind daher für praktisch relevante Problemgrößen kaum zu gebrauchen.

Unter gewissen Voraussetzungen eindeutige Darstellungen von Booleschen Funktionen sind die binären Entscheidungsgraphen (engl. *binary decision diagrams*, BDDs). Die heutige Verwendung solcher Diagramme geht auf eine Wiederbelebung älterer Ideen durch Bryant [Bry86, Bry92] zurück.

Wir erläutern BDDs zunächst anhand eines Beispiels. Die durch die Formel $f = bc + a\overline{b}\overline{c}$ dargestellte Boolesche Funktion kann wie in Abb. 5.43 repräsentiert werden.

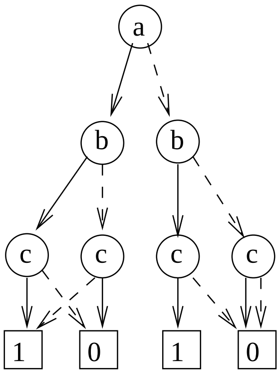


Abbildung 5.43: BDD für $f = bc + a\overline{b}\overline{c}$

Solche Entscheidungsgraphen sind gerichtet, azyklisch und besitzen genau einen Knoten ohne Vorgänger, die Wurzel. Jeder nichtterminale Knoten ist mit einer Variablen beschriftet und hat zwei ausgehende Kanten: eine durchgezogen gezeichnete 1-Kante und eine gestrichelt gezeichnete 0-Kante. Jeder terminale Knoten ist mit einer der Konstanten 0 oder 1 beschriftet und wird Senke genannt.

Entscheidungsgrpahen repräsentieren Boolesche Funktionen in naheliegender Weise: Jede Belegung der Eingabevariablen definiert einen eindeutigen Pfad von der Wurzel zu einer Senke. Der Wert dieser Senke gibt den Funktionswert zu dieser Eingabe an [MT97].

Abb. 5.44 enthält zwei BDDs für dieselbe Boolesche Funktion.

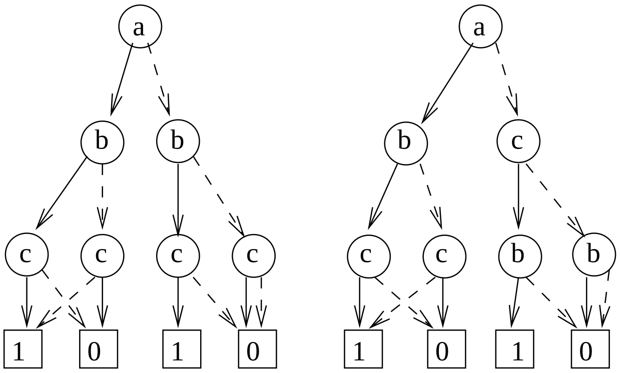


Abbildung 5.44: Zwei BDDs für $f = bc + a\overline{b}\overline{c}$

In diesen kommen die Variablen entlang der Pfade in unterschiedlicher Reihenfolge vor. Offensichtlich sind weitere Einschränkungen notwendig, um zu einer eindeutigen Darstellung einer Booleschen Funktion zu kommen und die erste Einschränkung besteht in der Wahl einer Ordnung, in der die Variablen entlang der Pfade vorkommen können.

[8] Quellen zu diesen und anderen Ergebnissen, die im Zusammenhang mit diesem Abschnitt wichtig sind, findet man beispielsweise bei Meinel [MT97] oder Wegener [Weg].

Ein Entscheidungsgraph heißt geordnet, wenn die Reihenfolgen der Variablen auf jedem Pfad von der Wurzel zu den Senken einer festen Ordnung genügen. Offensichtlich läßt sich für jede vorgegebene Variablenordnung ein solcher geordneter binärer Entscheidungsbaum konstruieren, z.B. in Form eines vollständigen Baumes. [MT97]. Entsprechend eingeschränkte BDDs nennen wir geordnete BDDs (OBDDs).

Abb. 5.45 zeigt, daß auch OBDDs noch nicht eindeutig sind.

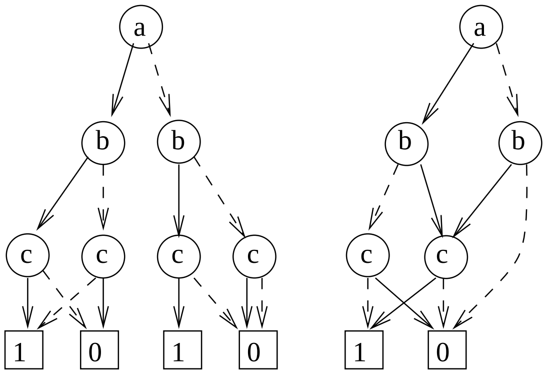


Abbildung 5.45: Zwei OBDDs für $f = bc + a\overline{b}\overline{c}$

Als weitere Einschränkung verlangen wir, daß unsere Graphen mit Hilfe der folgenden drei Regeln stets so weit wie möglich vereinfacht werden:

1. Terminalregel: Lösche alle Terminalknoten mit einer gegebenen Markierung bis auf einen, und lenke alle in die eliminierten Knoten gerichteten Kanten auf den entsprechenden verbliebenen um. Die Wirkung auf unser Beispiel zeigt die Abb. 5.46.

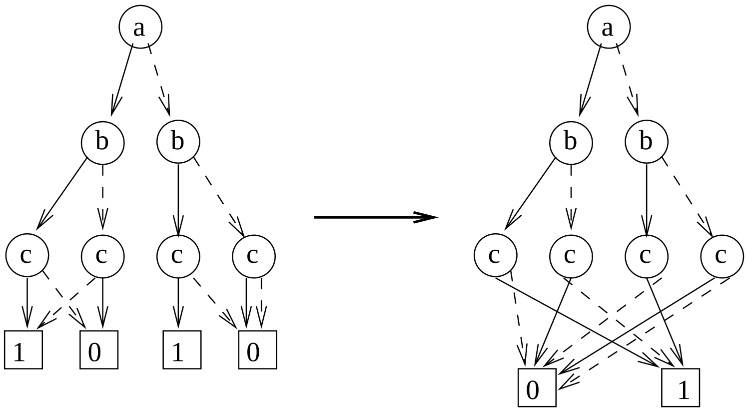


Abbildung 5.46: Auswirkung der Eliminationsregel beim Standardbeispiel

2. Eliminatiosnregel: Wenn die 1- und die 0-Kante eines Knotens v auf den gleichen Knoten u zeigen, dann eliminiere v und lenke alle eingehenden Kanten auf u um (siehe Abb. 5.47).

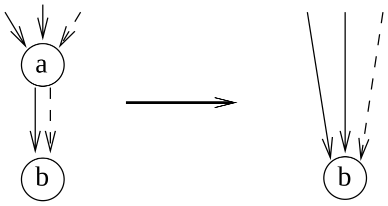


Abbildung 5.47: Eliminationsregel

   Für das vorhergehende Beispiel ergibt sich damit die Abb. 5.48.

3. Isomorphieregel:Wenn die Nichtterminalknoten u und v mit der gleichen Variablen markiert sind, ihre 1-Kanten zum gleichen Knoten führen und ihre 0-Kanten zum gleichen Knoten führen, dann eliminieren einen der beiden Knoten u, v und lenke alle eingehenden Kanten auf den anderen Knoten um (siehe Abb. 5.49).

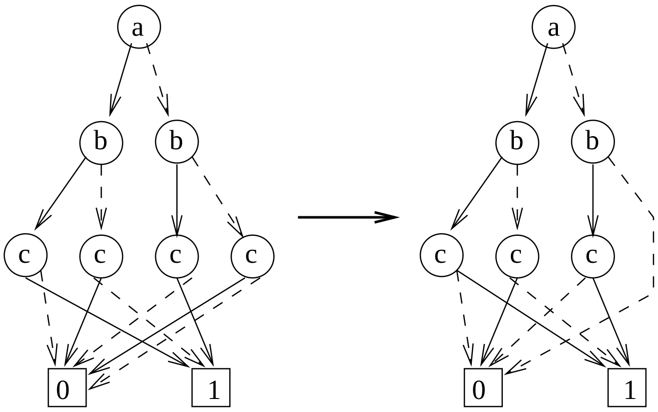


Abbildung 5.48: Elimination beim Standardbeispiel

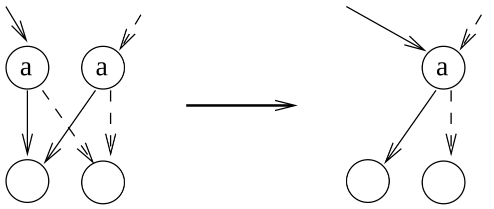


Abbildung 5.49: Isomorphieregel

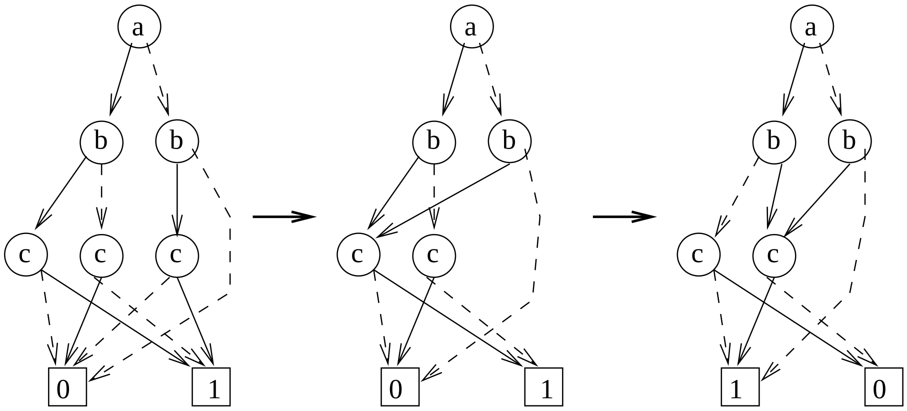


Abbildung 5.50: Auswirkung der Isomorphieregel beim Standardbeispiel

Für das vorhergehende Beispiel ergibt sich die Abb. 5.50.

Def.: Ein OBDD heißt reduziert (abgekürzt: ROBDD), wenn keine der drei Reduktionsregeln anwendbar ist.

Das OBDD im rechten Teil der Abb. 5.45 ist ein ROBDD.

ROBDDs besitzen die folgenden wesentlichen Eigenschaften [Weg, MT97] :

- ROBDDs bilden eine kanonische Darstellung Boolescher Funktionen
- Der Test auf **Äquivalenz** zweier durch Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ dargestellter Booleschen Funktionen ist dadurch in der Zeit $O(|V_1| + |V_2|)$ möglich.
- Es kann in der Zeit $O(V)$ geprüft werden, ob x_i in einer durch $G = (V, E)$ dargestellten Funktion wesentlich ist.
- Die Ersetzung von Variablen durch Konstanten kann in der Zeit $O(V)$ ausgeführt werden.
- Die Frage, ob zu einer Funktion f von Booleschen Variablen eine Belegung existiert, für die $f = 1$ ist (**Erfüllbarkeit**), kann bei der Darstellung von f durch OBDD $G = (V, E)$ in der Zeit $O(|V|)$ entschieden werden.

  Man denke an die völlig andere Situation bei disjunktiven Normalformen!

5.3.2 Bedeutung der Variablenordnung

Die Größe eine OBDDs und damit die Komplexität der Manipulation hängt von der zugrundeliegenden Variablenordnung ab - diese Abhängigkeit kann sehr stark sein. Ein Extrembeispiel ist das folgende: die Funktion

$$x_1x_2 + x_3x_4 + ... + x_{2n-1}x_{2n}$$

wird bezüglich der Variablenordnung $x_1, x_2, ..., x_{2n-1}, x_{2n}$ durch einen OBDD linearer Größe repräsentiert. Für die Variablenordnung $x_1, x_3, ..., x_{2n-1}, x_2, ..., x_{2n}$ hingegen wächst der reduzierte OBDD exponentiell in n (vgl. Abb. 5.51).

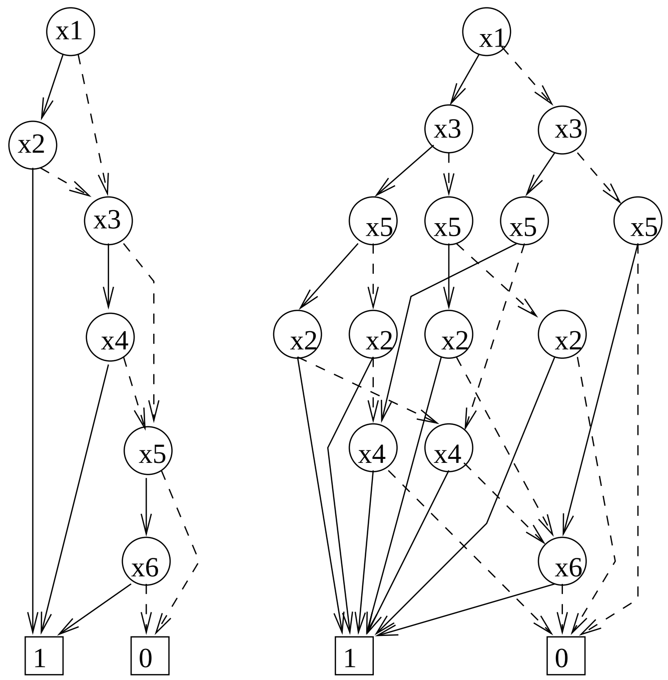


Abbildung 5.51: Einfluß der Variablenordnung

Der gleiche Effekt tritt im Fall von Addierfunktionen auf - auch hier variiert die OBDD-Größe je nach Variablenordnung von linear bis exponentiell in der Anzahl der Eingabebits. Andere wichtige Funktionen wie z.B. die Multiplikation zweier n-Bit-Zahlen haben für jede Variablenordnung OBDDs exponentieller Größe.

Wegen der starken Abhängigkeit der OBDD-Größe von der gewählten Variablenordnung ist es eines der wichtigsten Probleme im Umgang mit OBDDs, gute Ordnungen zu konstruieren. Es ist allerdings bekannt, daß das Problem, die optimale Ordnung für einen gegebenen OBDD zu bestimmen, NP-hart ist ... Die praktisch relevanten Optimierungsstrategien lassen sich in zwei Kategorien einteilen: Heuristiken und dynamisches Reordering.

`Heuristiken`. *Hierbei wird versucht, aus der Anwendung a priori Informationen abzuleiten, die zur Bestimmung einer guten Variablenordnung nützlich sind. Für den Fall der symbolischen Simulation wurden zahlreiche Verfahren entwickelt, um aus der Kenntnis der topologischen Struktur des Schaltkreises eine gute Ordnung zu gewinnen.*

`Dynamisches Reordering`. *Eine andere Technik, die OBDD-Größe zu minimieren, besteht darin, die Variablenordnung während der Bearbeitung dynamisch zu verbessern. Die bisher beste Reordering-Strategie geht auf Richard Rudell zurück und trägt den Namen* `sifting`. [MT97].

Insgesamt gibt es 2^{2^n} Boolesche Funktionen von n Variablen. Es ist klar, daß ein exponentieller Aufwand benötigt wird, um diese untereinander zu unterscheiden. Für die "meisten" Funktionen wird daher auch ein exponentieller Aufwand bei Darstellung als BDD erforderlich sein. Glücklicherweise existieren aber für sehr viele praktisch relevante Funktionen BDDs mit subexponentiellem Aufwand. Eine bekannte Ausnahme ist die Multiplikation, für die daher spezielle Varianten von BDDs entwickelt worden sind.

5.3.3 Shared BDDs

Eine Menge von BDDs, welche mehrere Boolesche Funktionen (sog. Funktionenbündel) repräsentiert, kann in einem einzigen BDD zusammengefaßt werden (siehe Abb. 5.52 [iM96].

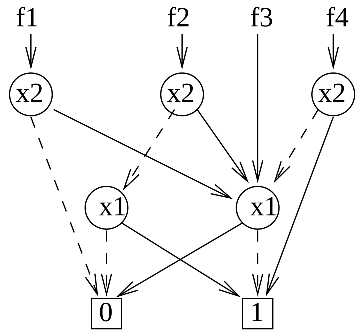


Abbildung 5.52: Shared BDD

Wenn gleichstrukturierte BDDs stets zusammengefaßt werden, nennen wir die entstehenden Graphen *shared BDDs* oder *multi-rooted BDDs*. Shared BDDs bieten die Möglichkeit einer kompakten Darstellung von Funktionenbündeln.

Durch Einführung von Pseudo-Variablen können mehrere Wurzeln wieder zu einer zusammengefaßt werden. Damit können Funktionenbündel mit den üblichen BDD-Paketen zur Darstellung einzelner Funktionen bearbeitet werden. Gemeinsame Teilausdrücke werden dabei leicht erkannt.

5.3.4 Partiell definierte Funktionen

In vielen Fällen ist der Wert von Booleschen Funktionen für gewisse Variablenbelegungen redundant, die Funktion also partiell definiert. Auf Seite 54 wurde dies mit `X` symbolisiert. `X` läßt sich mit 3-wertigen BDDs darstellen (siehe Abb. 5.53 [iM96]).

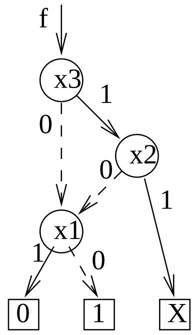


Abbildung 5.53: 3-wertiges BDD

Eine Alternative dazu ist Darstellung einer Funktion f durch ein Funktionenpaar $[f_0, f_1]$. $f = 0$ wird dann durch $f_0 = 0, f_1 = 0$ kodiert, $f = 1$ durch $f_0 = 1, f_1 = 1$ und schließlich $f = X$ durch $f_0 = 0, f_1 = 1$. Ein solches Funktionenpaar kann durch Einführung einer Variablen D wieder in einem einzigen BDD dargestellt werden (siehe Abb. 5.54 (links)).

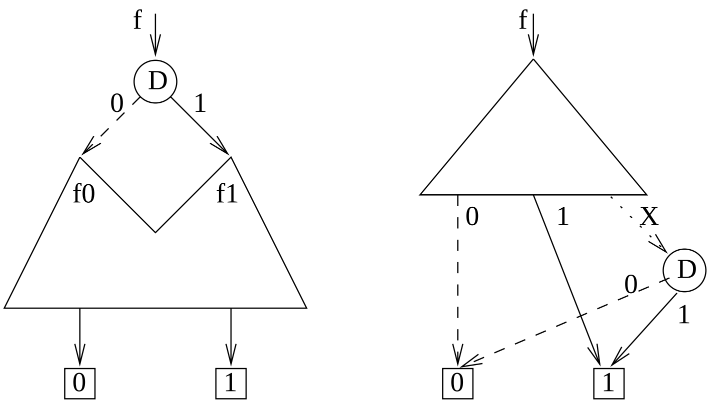


Abbildung 5.54: Benutzung der D-Variablen

Für vollständig definierte Funktionen sind f_0 und f_1 identisch und der Mehraufwand für die Beschreibung von Redundanzen wird zu Null.

Die Ordnung der Variablen kann im Prinzip auch so verändert werden, daß die D-Variable als letztes im Pfad auftaucht (siehe Abb. 5.54 (rechts)). In einer solches Darstellung entspricht jeder Pfad über den D-Knoten zur `1` einer Redundanz. Durch Umdeutung des D-Knotens als `X`-Terminalknoten kommen wir wieder zur Beschreibung wie in Abb. 5.53. BDDs mit D-Knoten an beliebiger Stelle innerhalb des Graphen stellen also den allgemeinen Fall dar.

5.3.5 Darstellung von Mengen mit Hilfe von ZBDDs

In vielen kombinatorischen Optimierungsproblemen müssen *Mengen* von Kombinationen dargestellt werden. Beispielsweise erfordert das Minimierungsverfahren von Quine/McCluskey die Behandlung von Mengen von Prim- und Mintermen. Falls n Elemente möglich sind, können Mengen von solchen Elemnten durch Bitvektoren der Länge n repräsentiert werden. Enthält der Bitvektor unter dem Index i eine `1`, so ist das Element i in der Menge vertreten, sonst nicht (siehe auch Vorlesung "Datenstrukturen").

Diese Bitvektoren können wieder als Boolesche Funktionen von n Variablen betrachtet werden. Derartige Boolesche Funktionen heißen **charakteristische Funktionen**. Sie können mit BDDs dargestellt werden (vgl. Abb. 5.55).

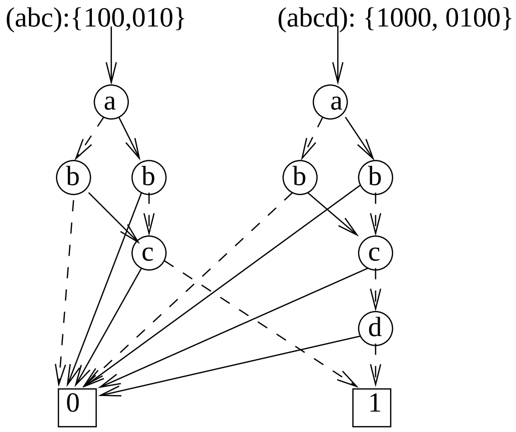


Abbildung 5.55: BDDs zur Darstellung von Mengen

Wegen der Ausnutzung gemeinsamer Teile von Graphen stellen BDDs eine sehr kompakte Repräsentation einer Menge von Basismengen dar. Vielfach entspricht die Komplexität von Operationen auf dieser Datenstruktur in grober Näherung der Größe des BDD und nicht der möglichen Größe der Potenzmenge der Basismengen. Allerdings hat diese Datenstruktur noch einen Nachteil: nicht vorkommende Variablen (wie c und d im Beispiel der Abb. 5.55) können nicht aus dem Graphen entfernt werden. Die o.a. Eliminationsregel führt in diesem Beispiel zu keiner Ersparnis.

Eine Lösung bietet der Übergang auf *zero suppressed BDDs* (ZBDDs), welche für die Darstellung von Mengen effizienter sind als Standard-BDDs. ZBDDs ergeben sich, wenn man die in Abb. 5.56 beschrieben Reduktiosregel anwendet: `1`-Zweige, die zum `0`-Terminal führen, sind offensichtlich redudant und der komplette Teilgraph kann durch den `0`-Zweig ersetzt werden.

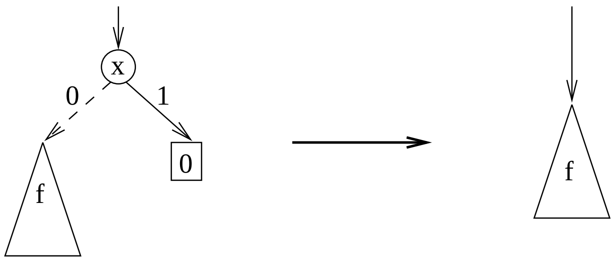


Abbildung 5.56: Reduktionsregel für ZBDDs

Mit Hilfe dieser Reduktionsregel können die Mengen der Abb. 5.55 als ZBDD wie in Abb. 5.57 dargestellt werden.

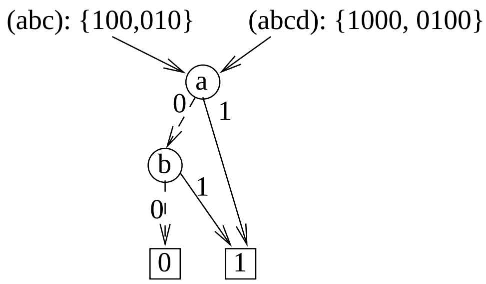


Abbildung 5.57: ZBDD zur Darstellung der o.a. Mengen

5.3.6 Minimierung mit BDDs

Minato hat gezeigt, wie man den Algorithmus von Morreale zur Generierung nicht-redundanter Überdeckungen von der Verwendung disjunktiver Normalformen auf die Verwendung von BDDs umstellen kann [iM96]. Auf diese Weise gelingt die Erzeugung kostengünstiger (wenngleich nicht notwendig minimaler) Realisierungen Boolescher Funktionen. Die Qualität der Lösungen ist etwas schlechter als bei ESPRESSO; dafür können allerdings auch in solchen Fällen noch Lösungen generiert werden, in denen ESPRESSO innerhalb von 10 Stunden Rechenzeit keine Lösung liefert.

Kapitel 6

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.

6.1 Platzierung

6.1.1 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:
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

[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.

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 $\rightarrow$ Min,

b) Maximum der von Schnittlinien durchschnittenen Netze $\rightarrow$ Min; man betrachtet die Anzahl der Netze, die von einer senkrecht dazu geführten Linie geschnitten werden (siehe Abb. 6.1).

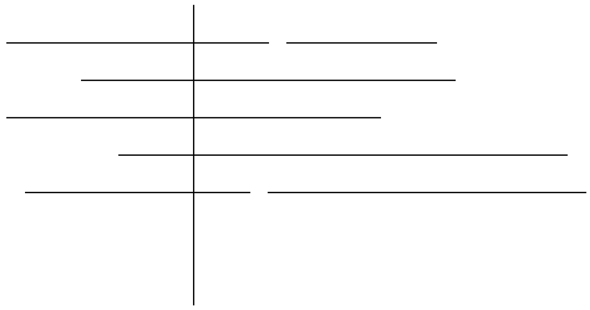


Abbildung 6.1: Schnitt von Netzen

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

c) Maximale Dichte von Leitungen pro Fläche $\rightarrow$ Min.

6.1.2 Platzierung nach dem Kräftemodell

Frühe Veröffentlichungen über Platzierungsverfahren basierten häufig auf dem Kräftemodell idealer Federn (siehe z.B. [HK72]). Man betrachte dazu Abb. 6.2.

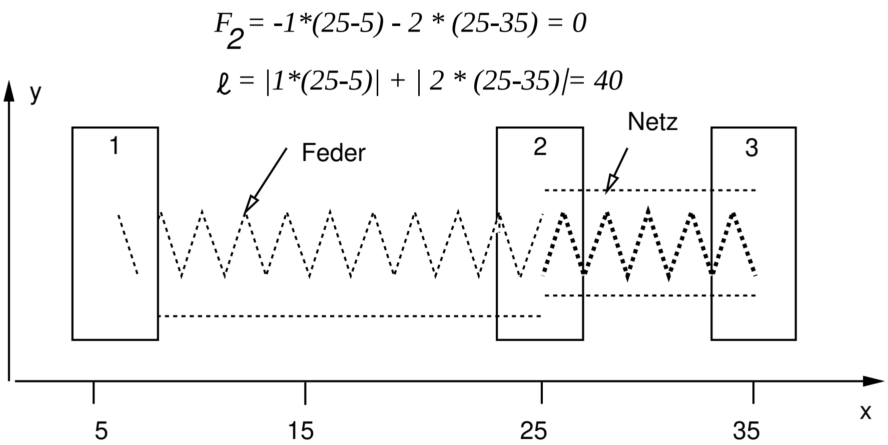


Abbildung 6.2: Platzierung nach dem Kräftemodell

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

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

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 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. 6.3. Die Verdrahtungslänge ist hier gegenüber der Abb. 6.2 reduziert.

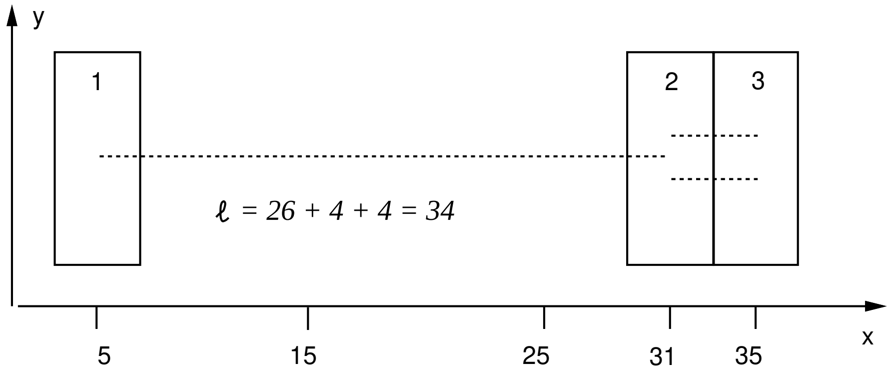


Abbildung 6.3: Platzierung mit minimaler Verdrahtungslänge

6.1.3 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. 6.4 zeigt ein Beispiel für eine Platzierung von drei Zellen *1, 2* und *3* auf Plätzen im Bereich von 1 bis 15.

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

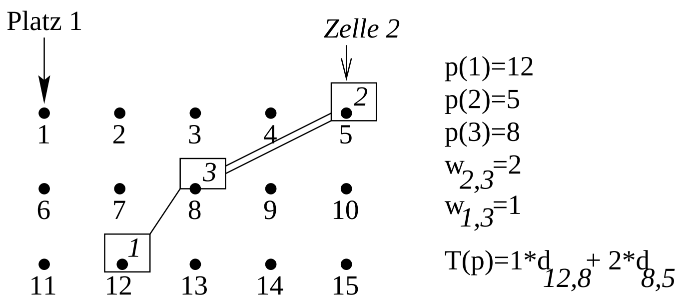


Abbildung 6.4: Beispiel für eine Platzierung

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. 6.5.

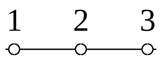


Abbildung 6.5: 3-Punkt-Netz

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 $w_{i,j}$ 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.

---

[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].

6.1.4 Platzierung mittels Partitionierung

6.1.4.1 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

$$k : V \to \mathbb{R}^+$$
$$w : E \to \mathbb{R}^+$$

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

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

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

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.

6.1.4.2 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
```

[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].

`Algorithmus 4.1: Grundidee der konstruktiven Partitionierung`

Beispiel:

Gegeben sei der Graph nach Abb. 6.6 (links). Eine Beschränkung möge bewirken, dass etwa die Hälfte aller Zellen (Knoten) einem Block zugeordnet werden.

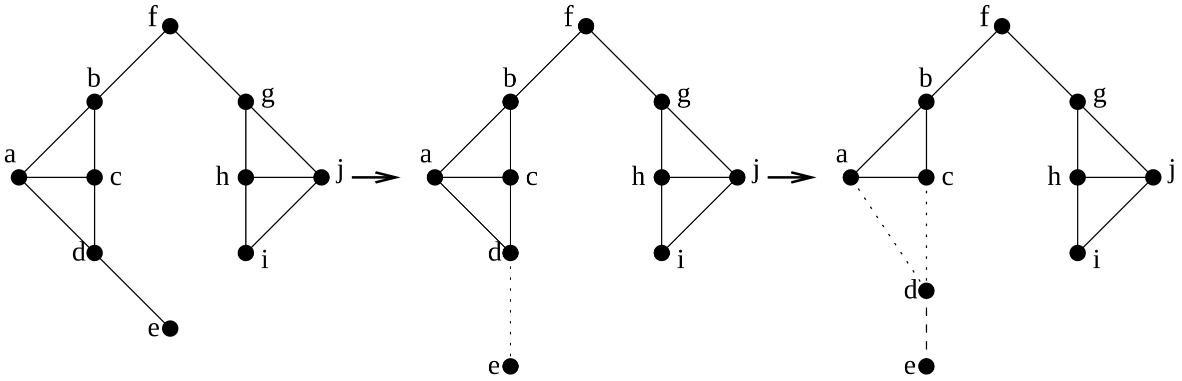


Abbildung 6.6: Interatives Partitionieren eines Graphen

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. 6.6 (mitte) bis Abb. 6.7 (rechts) zeigen, wie der Graph iterativ partitioniert wird.

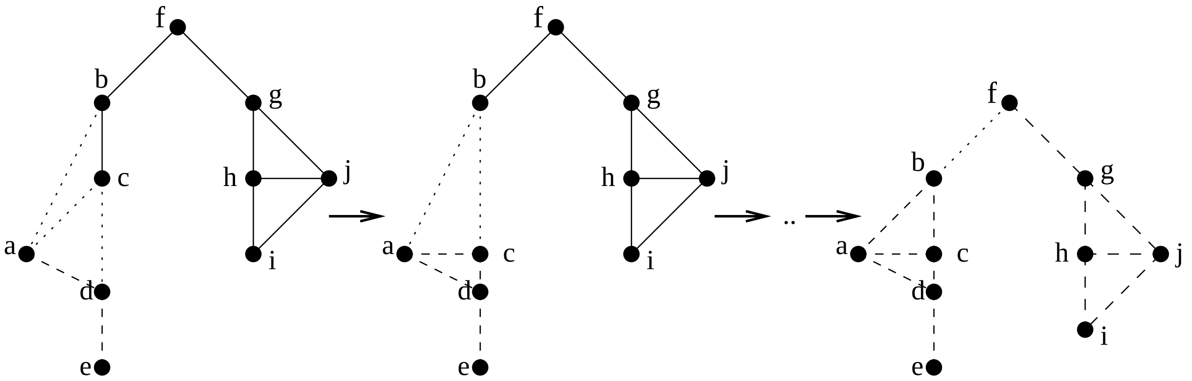


Abbildung 6.7: Interatives Partitionieren eines Graphen

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].

6.1.4.3 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.

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. 6.8 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.

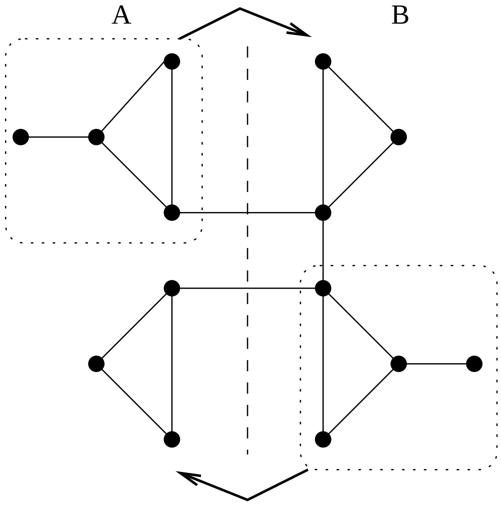


Abbildung 6.8: Graph, welcher ein nicht-lokales Verhalten der Partitionierung erfordert

6.1.4.4 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 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.

6.1.4.5 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:

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

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. 6.9).

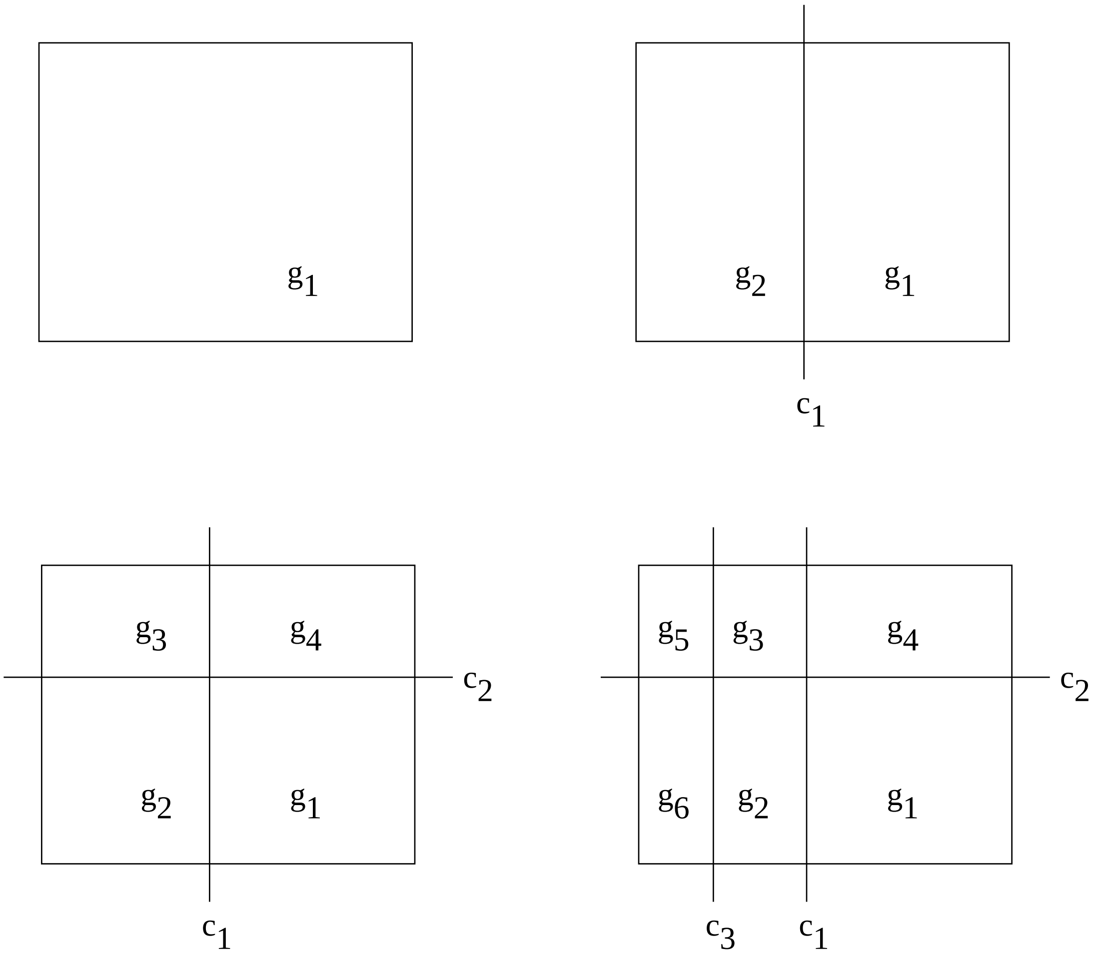


Abbildung 6.9: Min-Cut-Verfahren nach Breuer

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

[6] Gesprochen: Bru–er.

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.

6.1.4.6 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. 6.10 ist einer der Graphen mit gestrichelten, der andere mit einer durchgezogenen Kante dargestellt.

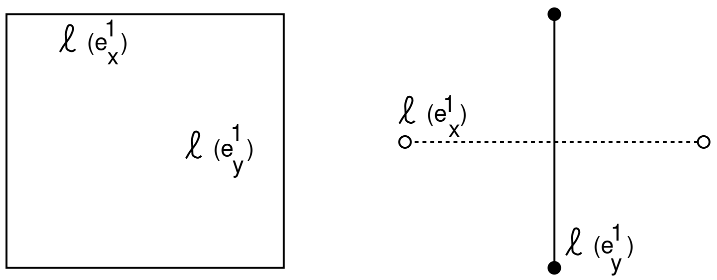


Abbildung 6.10: Darstellung einer einzelnen Fläche durch 2 Graphen

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 6.11 liegen die Flächen 1 und 2 untereinander, folglich sind auch die Kanten e_y^1 und e_y^2 entsprechend geordnet.

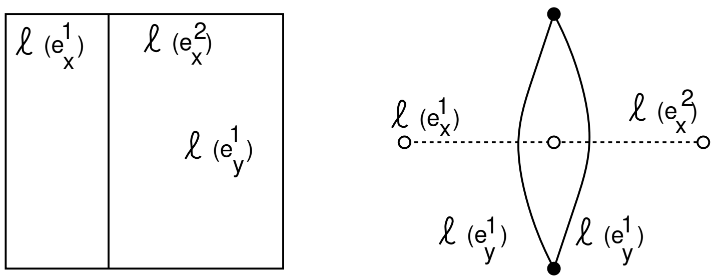


Abbildung 6.11: Darstellung der relativen Lage zweier Flächen

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. 6.10.

2. Nunmehr wird der Netzgraph (zur Definition siehe Seite 167) 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 6.11.

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. 6.11 und bei mehrfachem Schneiden die Abb. 6.12 entstehen[7]. Man beachte die unterschiedliche Größe der Flächen.

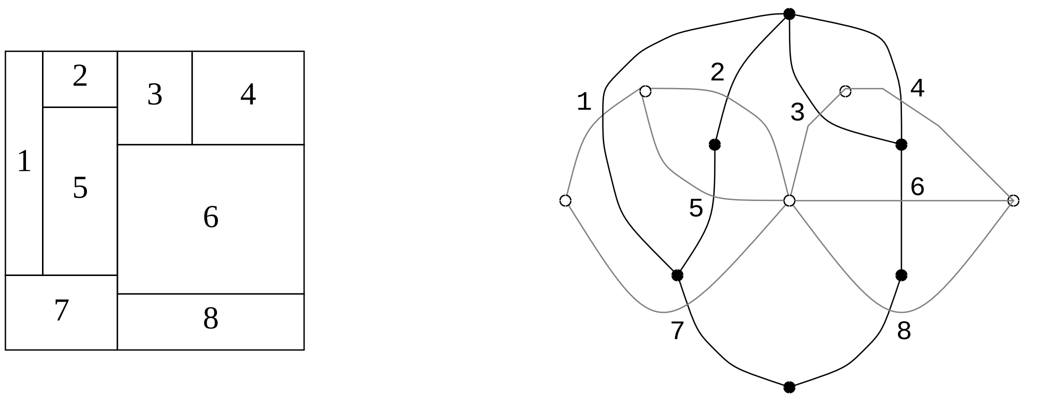
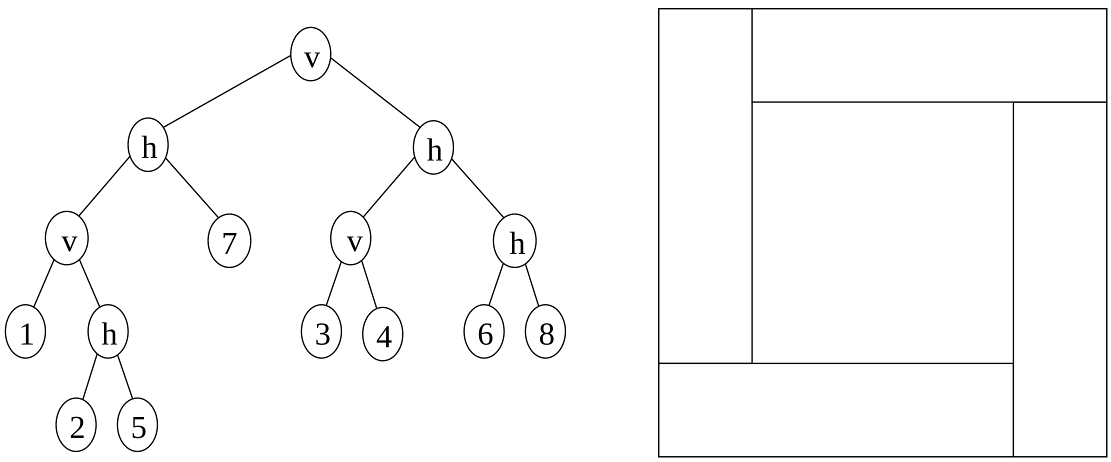


Abbildung 6.12: Platzierung mit unterschiedlich großen Zellen

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. 6.13 a) gibt den Slicing–tree der Flächenaufteilung in Abb. 6.12 wider.



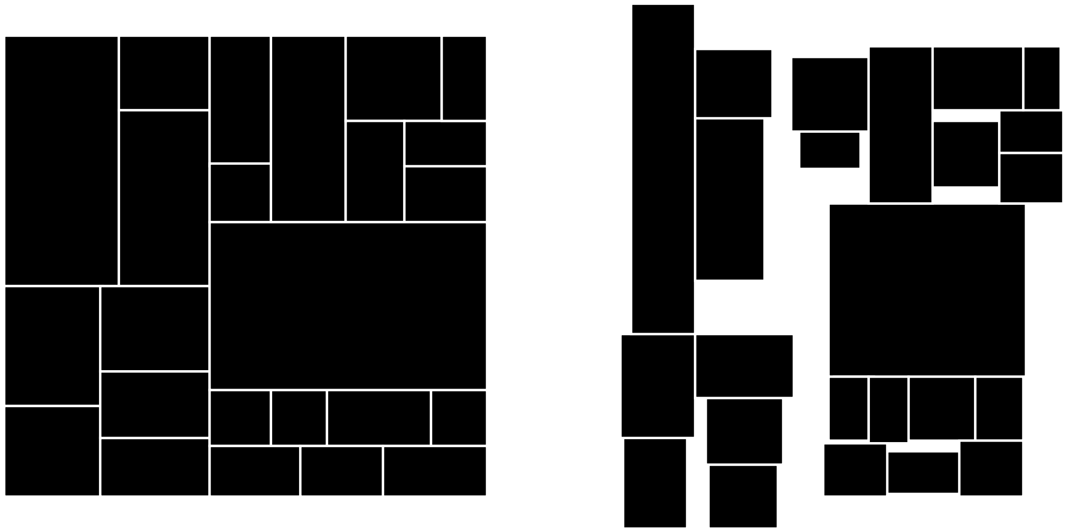
a) b)

Abbildung 6.13: Slicing-Tree (a) und Pin-Wheel (b)

Slicing–trees sind keine allgemeine Darstellung von Flächenaufteilungen, da sie sog. *pin–wheels* nach Abb. 6.13 b) nicht beschreiben können.

[7]Um Abb. 6.12 zu erhalten, wurde zweimal nacheinander horizontal geschnitten.

Die Teilflächen der Abb. 6.12 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. 6.14 a).



a) Initiales Placement b) Berücksichtigung der Zellabmessungen

Abbildung 6.14: Verfahren von Lauther (©1979 IEEE)

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. 6.14 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. 6.14 b).

Verbesserungen

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

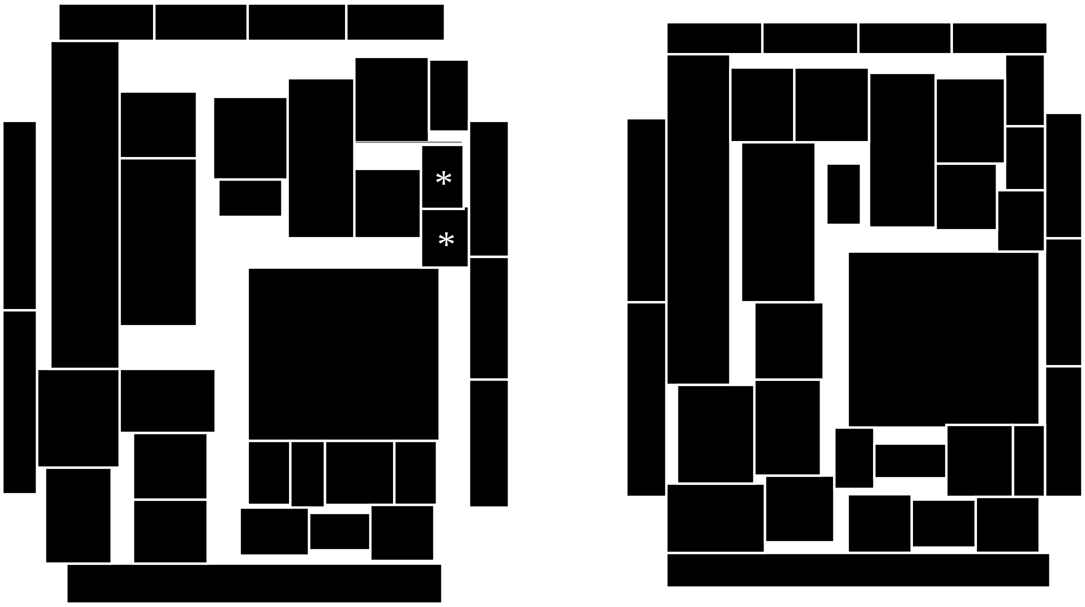
a) Eine Vertauschung von $\ell(e_x^i)$ und $\ell(e_y^i)$ (**Rotation**) kann zu einer Flächenreduktion führen. Aufgrund der vorangegangenen Orientierung ist dies aber selten der Fall. Abb. 6.15 a) zeigt ein Beispiel für einen solchen Fall. Die mit einem Stern gekennzeichneten Zellen wurden gedreht. Damit wurde die Ausdehnung in x-Richtung reduziert. Man beachte, dass die Abbildung zusätzlich am Rand Zellen für die äußeren Anschlüsse enthält.

b) Abb. 6.16 a) zeigt einen kritischen Pfad durch einen der Polargraphen. In Abb. 6.16 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. 6.17). Aus der Abb. 6.15 a) entsteht durch diese Optimierung die Abb. 6.15 b).

6.1.4.7 Berücksichtigung von Netzanschlüssen außerhalb der gegenwärtigen Fläche

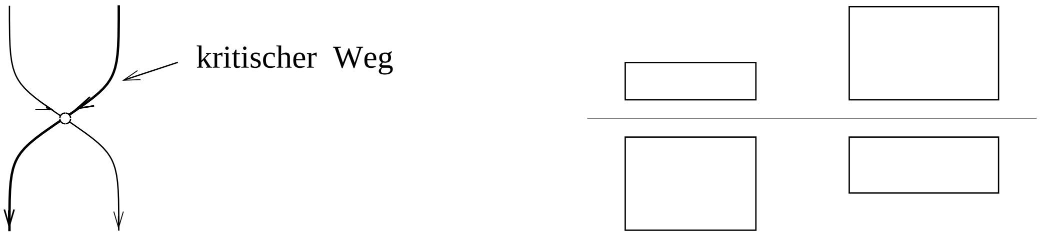
Ein Grundproblem der bislang vorgestellten fortgesetzten Bipartitionierung besteht darin, dass Netzanschlüsse außerhalb der zu teilenden Fläche nicht berücksichtigt werden. Man betrachte dazu Abb. 6.18.

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. 6.19).



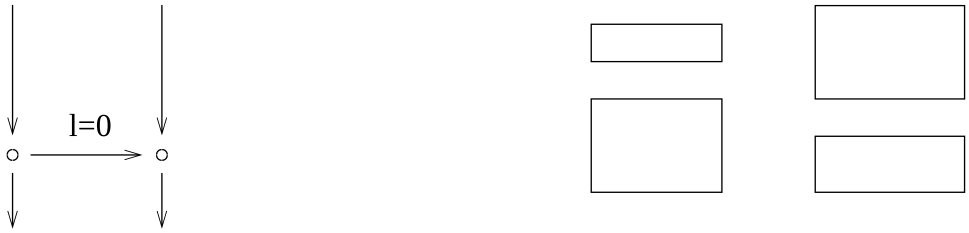
a) Nach Rotation (E/A-Pins eingefügt) b) Nach "Squeezing"

Abbildung 6.15: Ergebnisse des Verfahrens von Lauther (©1979 IEEE)



a) Polargraph b) Flächenaufteilung

Abbildung 6.16: Squeezing : Ausgangssituation



a) Polargraph b) Flächenaufteilung

Abbildung 6.17: Squeezing : Ergebnis

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

6.1.4.8 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

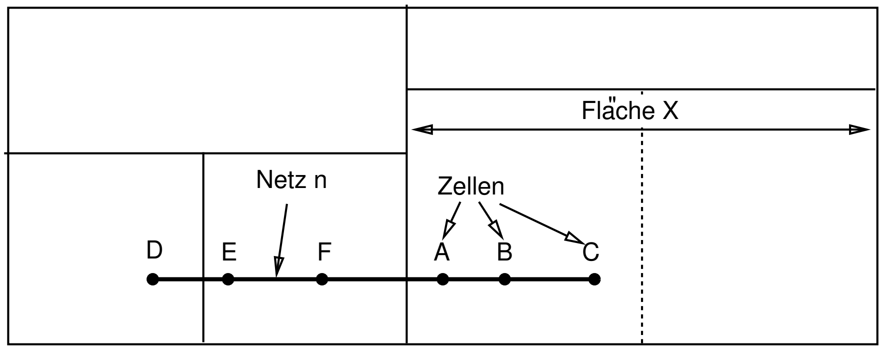


Abbildung 6.18: Zum Problem der Anschlüsse außerhalb der aktuellen Fläche `X`

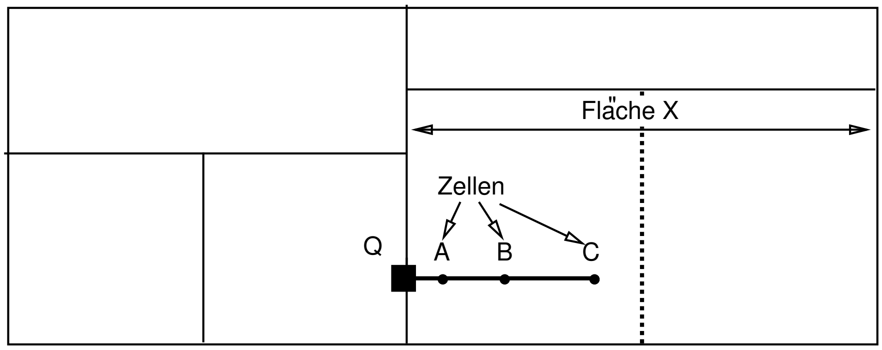


Abbildung 6.19: Einführung einer "Dummy"-Zelle `Q`

und ihre Zuordnung zu vier Quadranten vorgeschlagen.

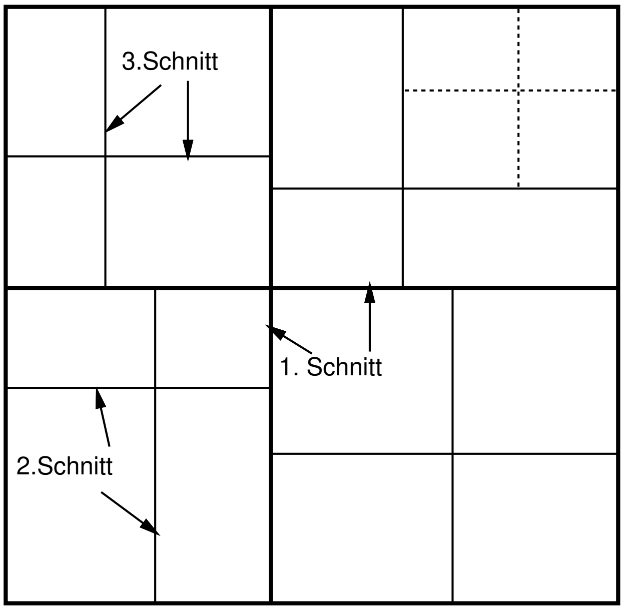


Abbildung 6.20: Aufteilung einer Fläche durch Quadripartitionierung

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.

6.1.5 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 [vLA87a], 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.

6.1.6 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.

6.1.7 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.

6.2 Globale Verdrahtung

6.2.1 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 $w : E \to I\!R^+$.

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 6.21. Alle Kantengewichte sind als 1 angenommen.

Das allgemeine Minimierungsproblem heißt **Steiner tree on graph problem** (STOGP) (siehe z.B. Abb. 6.22).

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.

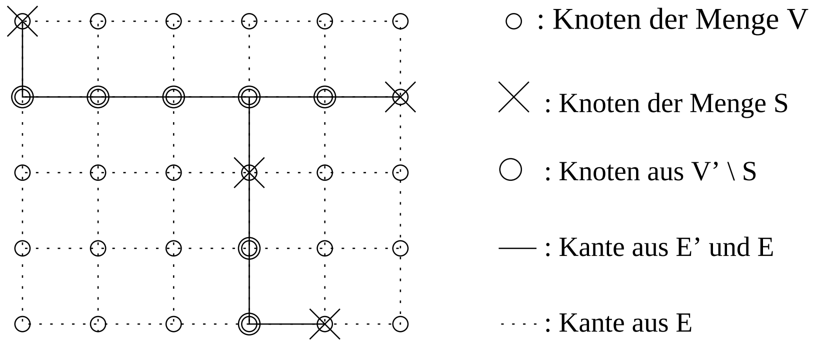


Abbildung 6.21: Minimaler Steiner-Baum in einem Rechteckraster

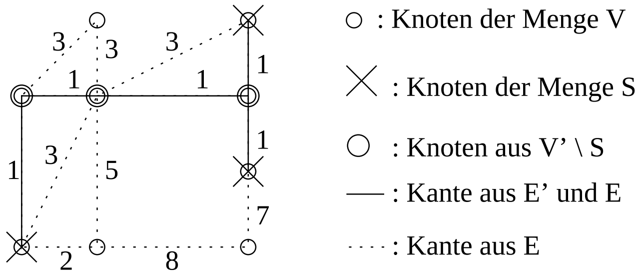


Abbildung 6.22: Minimaler Steiner-Baum in einem Graphen

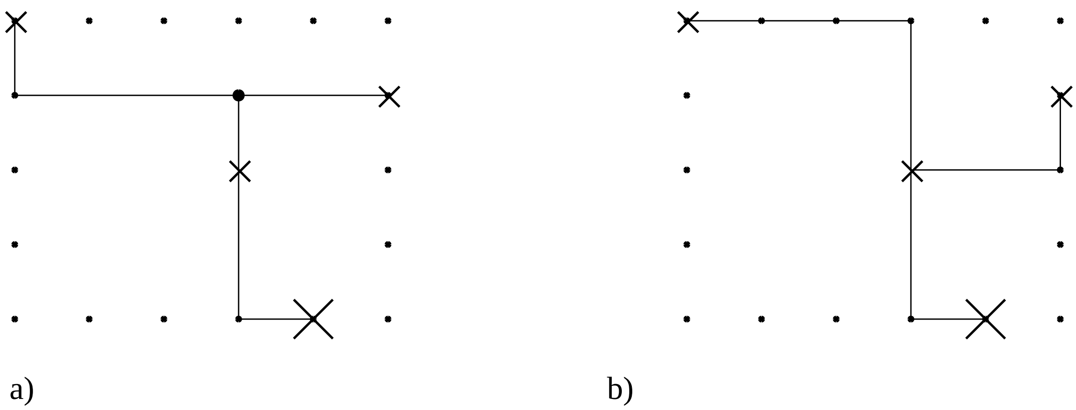


Abbildung 6.23: a) Minimaler Steinerbaum ($\ell = 10$) b) Minimaler Spannbaum ($\ell = 11$) (.=Rasterpunkte, x=Eingänge, X=Ausgang, o=Verdrahtungsknoten)

1. Steiner-Bäume:
   Der minimale **Steiner–Baum** ist ein Baum mit minimaler Verdrahtungslänge, der sich an beliebigen Knoten verzweigt (siehe Abb. 6.23 a, nach [Oht86]).
2. Spannbäume:
   Der minimale **Spannbaum** ist ein Baum mit minimaler Verdrahtungslänge, der sich nur an den Netzknoten verzweigt (siehe Abb. 6.23 b).
3. Minimale Ketten:
   Die minimale **Kette** ist ein Baum mit minimaler Verdrahtungslänge, der sich nirgends verzweigt (Abb. 6.24 a).
4. Minimale direkte Ein/Ausgangsverbindung:
   Die minimale **direkte Ein/Ausgangsverbindung** (siehe Abb. 6.24 b) ist ein Baum mit minimaler Verdrahtungslänge, in dem alle Eingänge direkt mit dem Ausgang verbunden sind. In den Abbildungen 6.23 und 6.24 ist die jeweilige Verdrahtungslänge mit ℓ bezeichnet.

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 Recht-

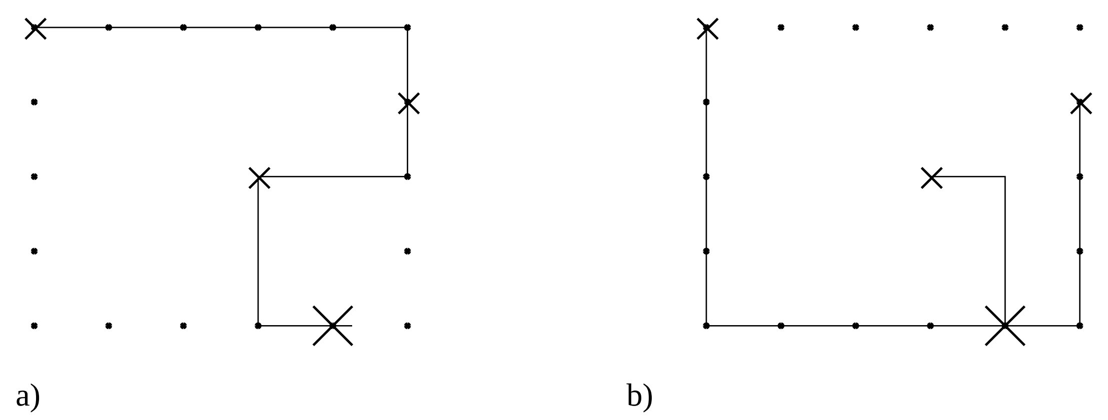


Abbildung 6.24: a) Kette ($\ell = 12$) b) direkte E/A-Verbindung ($\ell = 15$)

ecks, in dem alle zu verdrahtenden Knoten liegen, abgeschätzt (siehe Abb. 6.25). Bei 2- und 3-Punkt-Netzen ist die Abschätzung exakt gleich der Länge des minimalen Steiner–Baumes (Übungsaufgabe!).

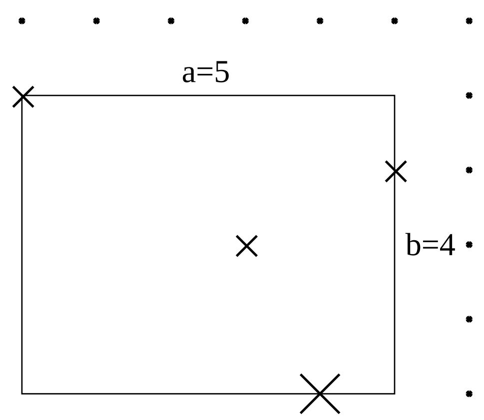


Abbildung 6.25: Umschreibendes Rechteck, (ℓ=9)

6.2.2 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. 6.26 zeigt eine mögliche Einteilung einer Fläche in solche Regionen.

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. 6.26 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. 6.27 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

[8] Eine genaue Definition des Problems der Verdrahtung innerhab von Kanälen erfolgt im Abschnitt **Kanalverdrahtung**.

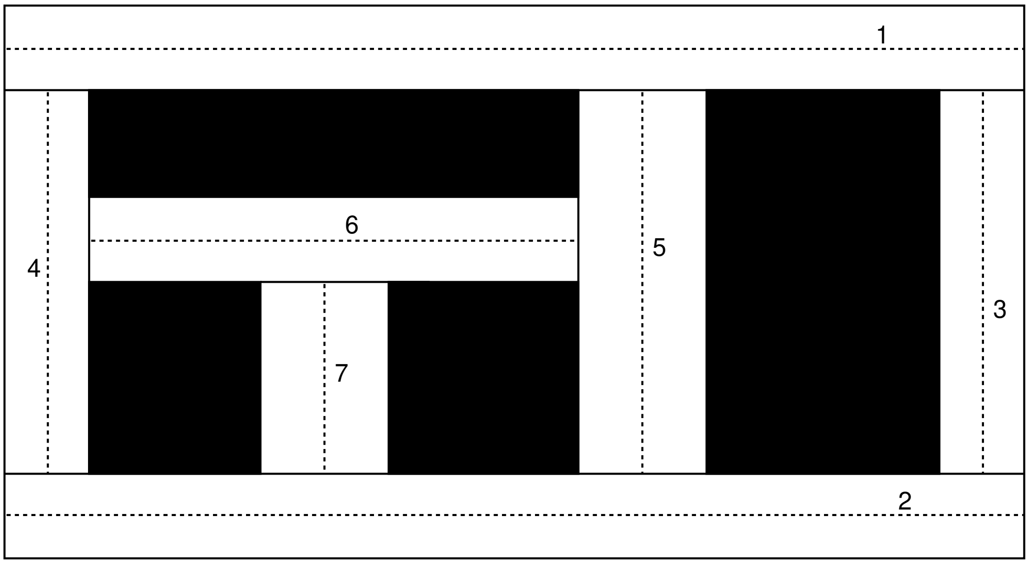


Abbildung 6.26: Verdrahtungskanäle

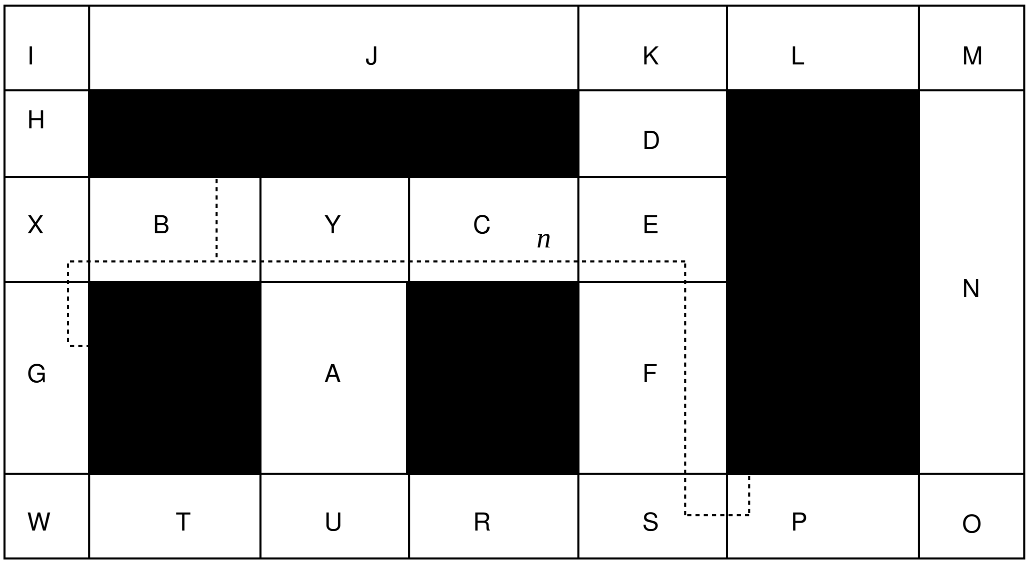


Abbildung 6.27: Minikanäle , Verdrahtung eines Netzes `n`

z.B. die Verdrahtung des Netzes `n` zu führen ist, für das die Abb. 6.27 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. 6.28 zeigt einen Nachbarschaftsgraphen für die Minikanäle der Abb. 6.27.

[9] In einer anderen möglichen Definition werden die Kanäle als Kanten und die Zellen als Knoten modelliert.

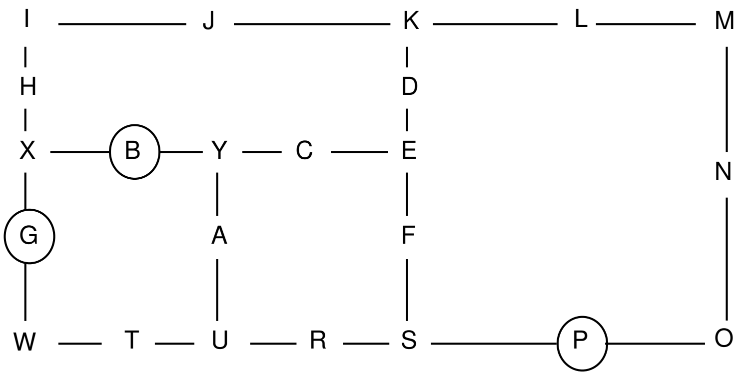


Abbildung 6.28: Nachbarschaftsgraph (Kreise kennzeichnen die Anschlüsse des Netzes `n`)

6.2.3 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. 6.27. 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. 6.28 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.

6.2.4 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:

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

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

```
S' := {v_0};
D[v_0] := 0;
FOR each v ∈ V − {v_0} DO D[v] := ℓ(v_0, 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 v_0 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:

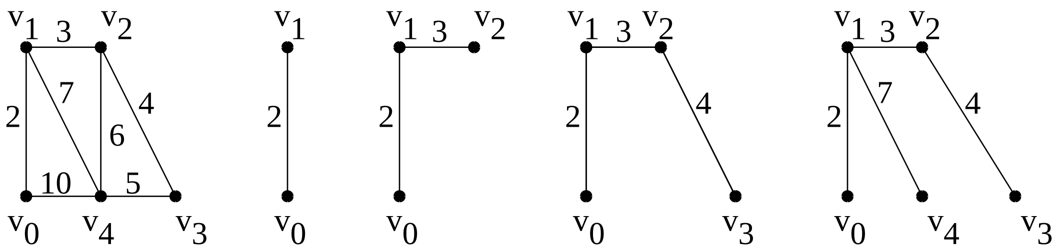


Abbildung 6.29: a) Kantengewichteter Graph $G = (V, E)$ und b) Expansionen

Die Tabelle 6.30 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.

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 6.30: Bestimmung kürzester Wege nach Dijkstra

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!).

6.2.5 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.

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. 6.32 die Entfernungen von den Knoten `a`, `b` und `c` als Tripel in runden Klammern.

[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.

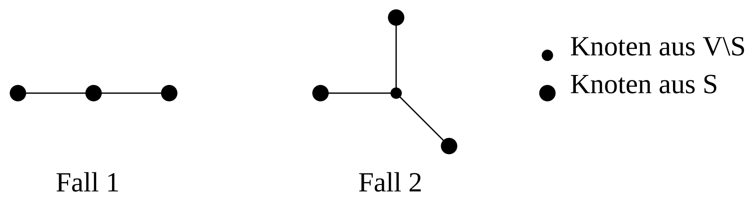


Abbildung 6.31: Verzweigungsknoten bei 3-Punkt-Steiner-Bäumen

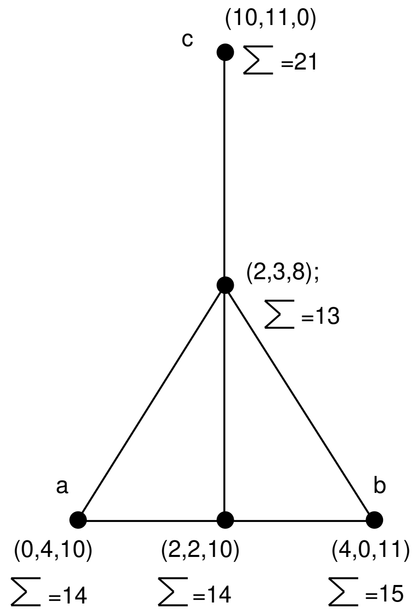


Abbildung 6.32: Lösung des obigen Beispiels mit dem optimalen Algorithmus

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.

6.2.6 *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:

```
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. 6.33 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 $\ell = 13$.

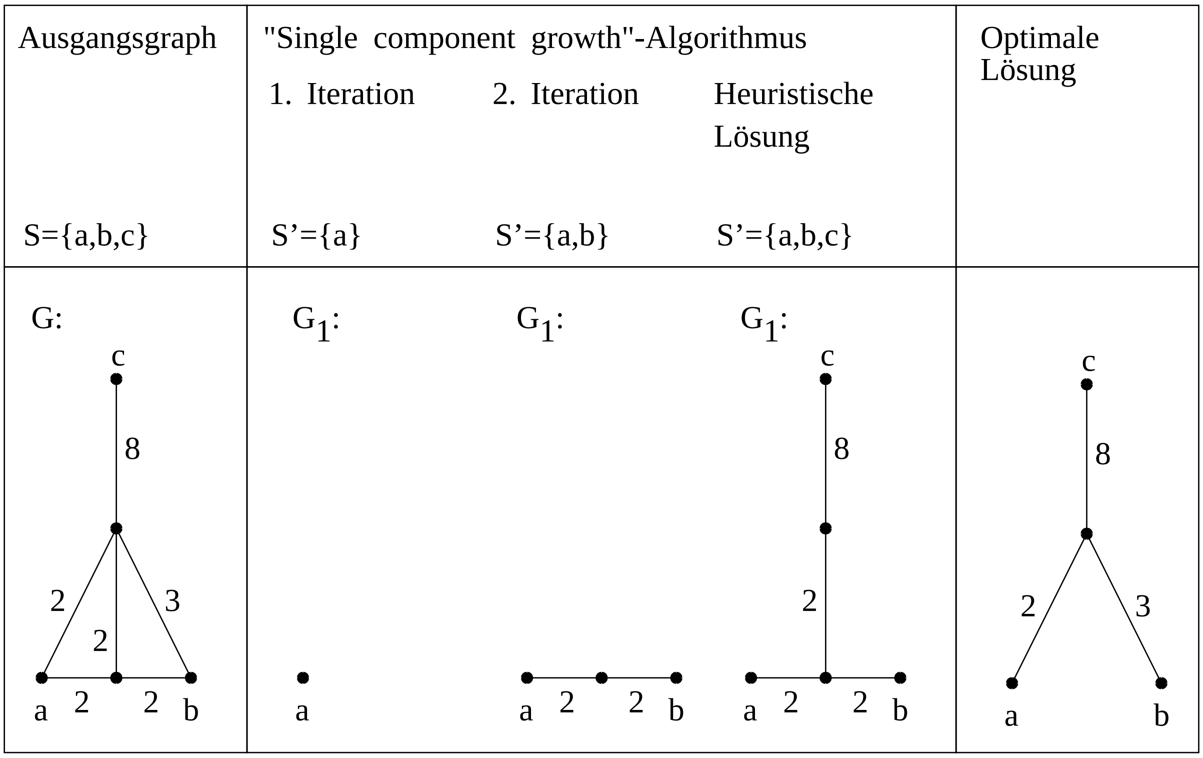


Abbildung 6.33: Beispiel für den *single component growth*-Algorithmus

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.

6.2.7 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. 6.34 a) zeigt einen Ausgangsgraphen G und Abb. 6.34 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 Steiner-Baums. Abb. 6.35 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. 6.35 b) zeigt die Pfade, die in den Graphen G_3 übernommen werden, für unser Beispiel. Die zugehörigen Kanten in G_2 sind gestrichelt eingezeichnet.

4. Berechne einen minimalen Spannbaum G_4 von G_3. Abb. 6.36 a) zeigt den Graphen G_4 für unser Beispiel.

5. Entferne in G_4 die Blätter, die nicht zu S gehören. Der erhaltene Graph heiße G_5. Abb. 6.36 b) zeigt das Ergebnis für unser Beispiel.

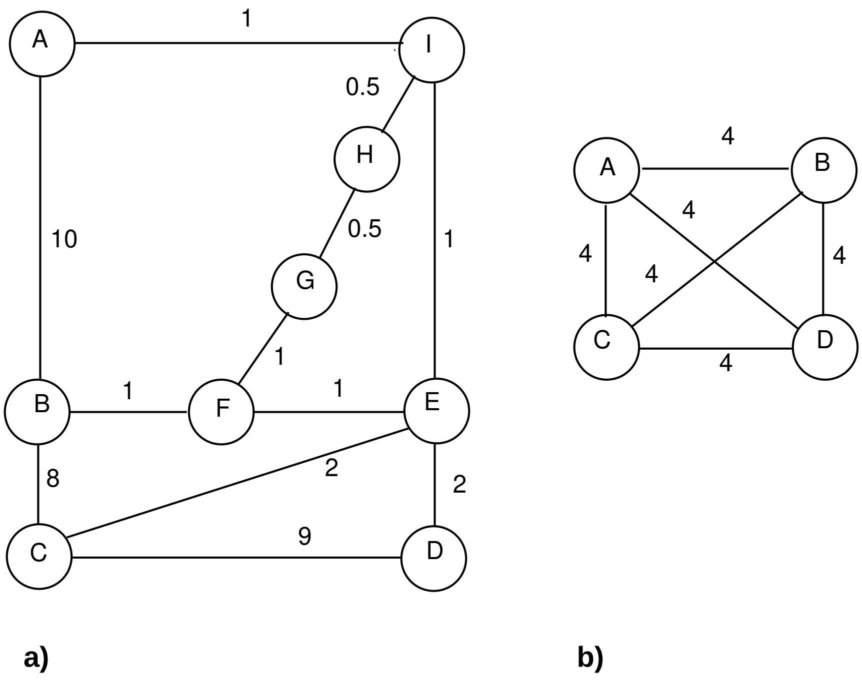


Abbildung 6.34: a) Ausgangsgraph G mit $S = \{A, B, C, D\}$; b) Distanzgraph G_1

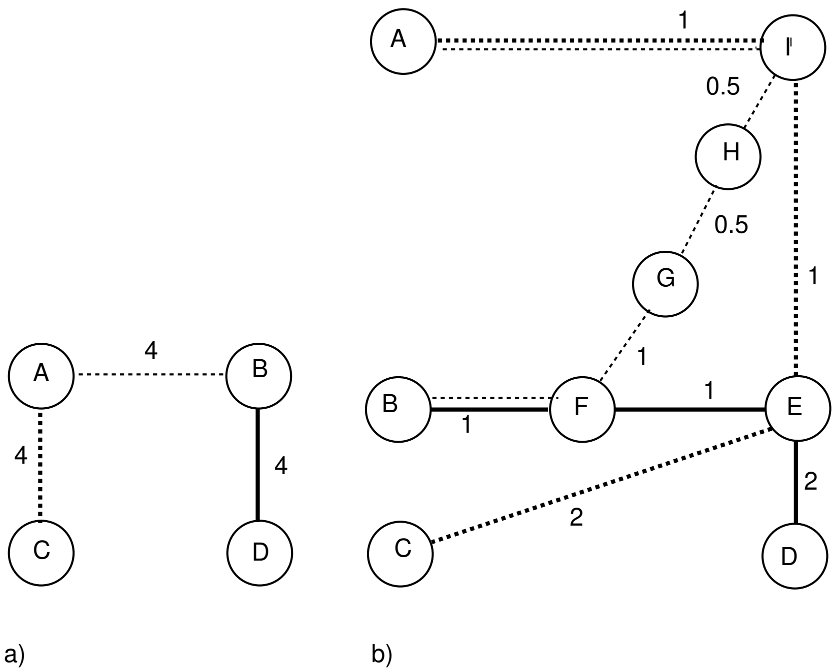


Abbildung 6.35: a) Spannbaum G_2; b) Zum Spannbaum gehörende Pfade (Löschen doppelter Kanten ergibt G_3)

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:

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

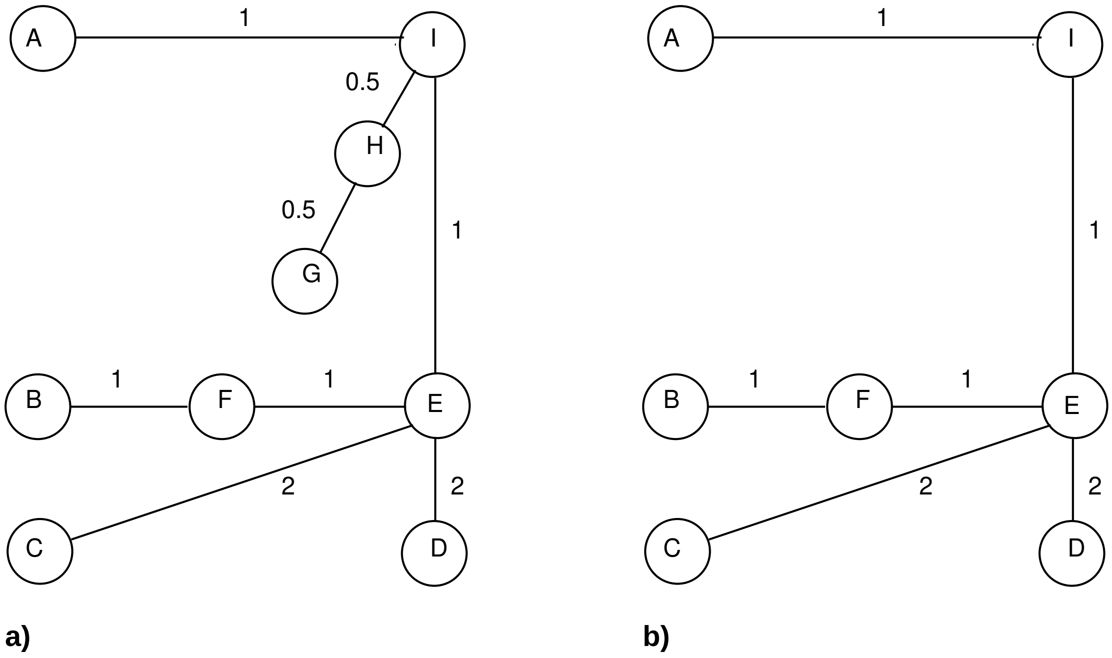


Abbildung 6.36: a) Graph G_4; b) Graph G_5

Die Beweisidee ist folgende: Man nehme dazu an, dass der in Abb. 6.37 gezeigte Baum der minimale Steiner-Baum eines Steiner-Baum-Problems sei.

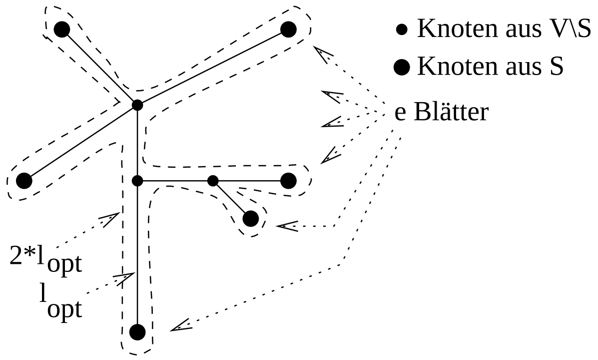


Abbildung 6.37: Zur oberen Schranke für die Länge des Steiner-Baumes nach der o.a. Methode

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.

6.2.8 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.

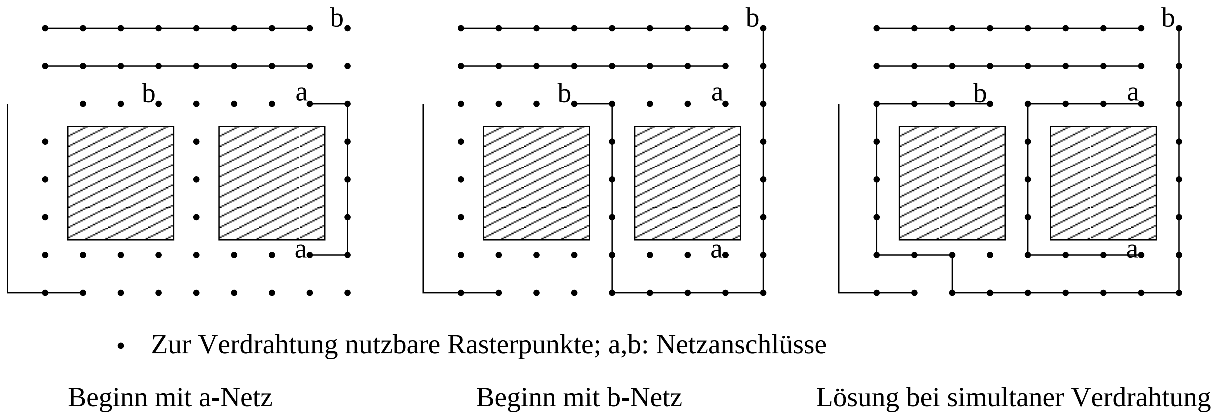


Abbildung 6.38: Problem, welches bei *net at a time*-Verdrahtung unlösbar ist

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.

6.2.9 Sortierung der Verdrahtungsregionen

Während der im Abschnitt 6.3 beschriebenen detaillierten Verdrahtung werden wir vor allem eine Verdrahtung innerhalb der Kanäle betrachten, wie sie in Abb. 6.26 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. 6.39. 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.

Als Konsequenz muß die Verdrahtung einer Slicing–Struktur in einer gegenüber der Schnittabfolge umgekehrten Reihenfolge durchgeführt werden (siehe Abb. 6.40).

6.3 Detaillierte Verdrahtung

6.3.1 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.

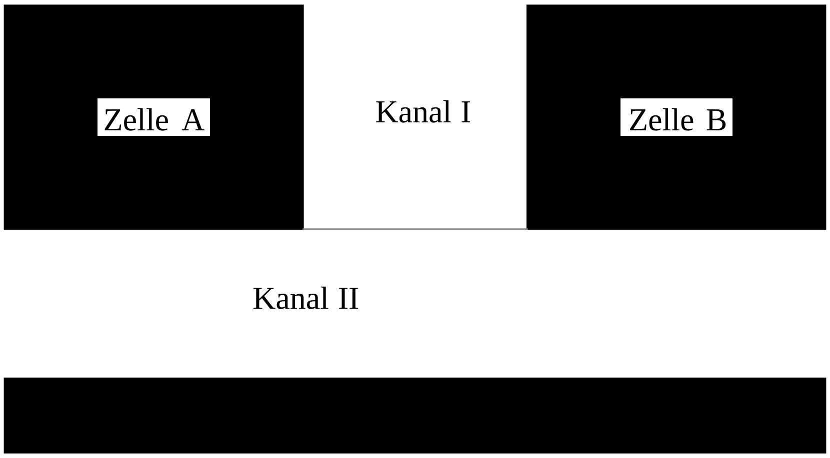


Abbildung 6.39: Zur Reihenfolge der detaillierten Verdrahtung

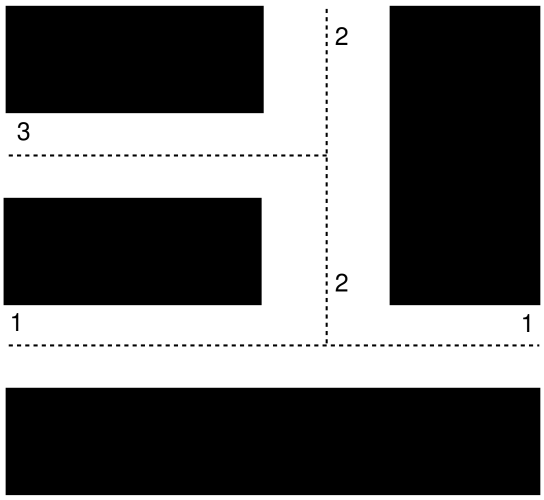


Abbildung 6.40: Reihenfolge der Schnitte bei Slicing-Trees

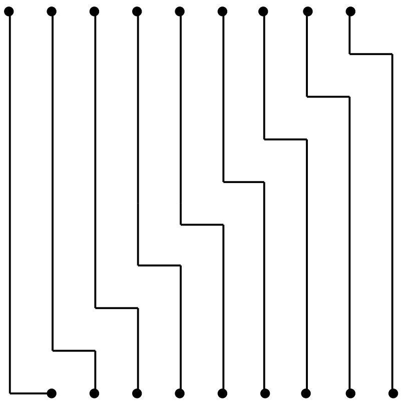


Abbildung 6.41: River-Routing (besonders ungünstiger Fall)

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

6.3.2 Kanalverdrahtung in zwei Ebenen

6.3.2.1 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. 6.42):

- 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].

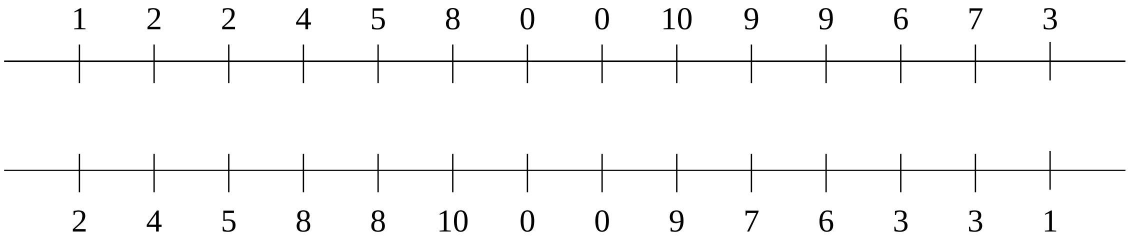


Abbildung 6.42: Beispiel zur Kanalverdrahtung

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. 6.42 [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. 6.42 werden die Intervalle in Abb. 6.43 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. 6.44 enthält den HCG für das Beispiel aus Abb. 6.42.

[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.

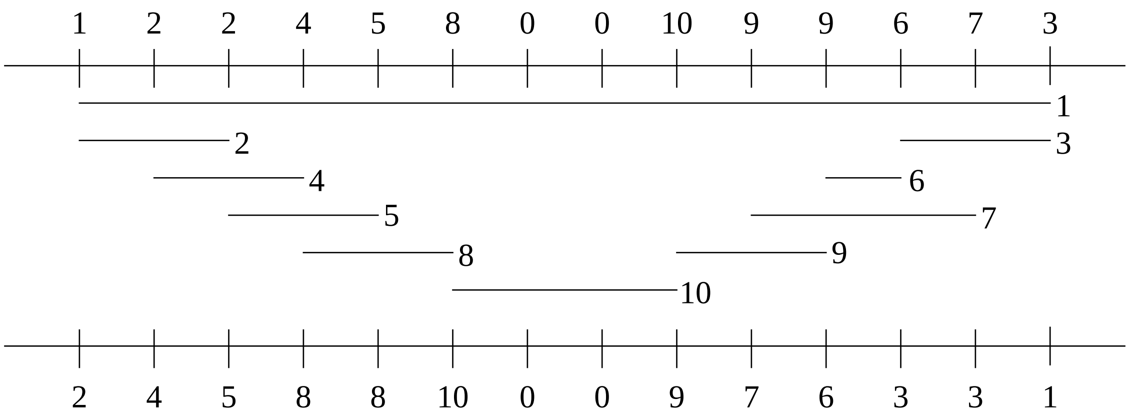


Abbildung 6.43: Intervalle der Netze für obiges Beispiel

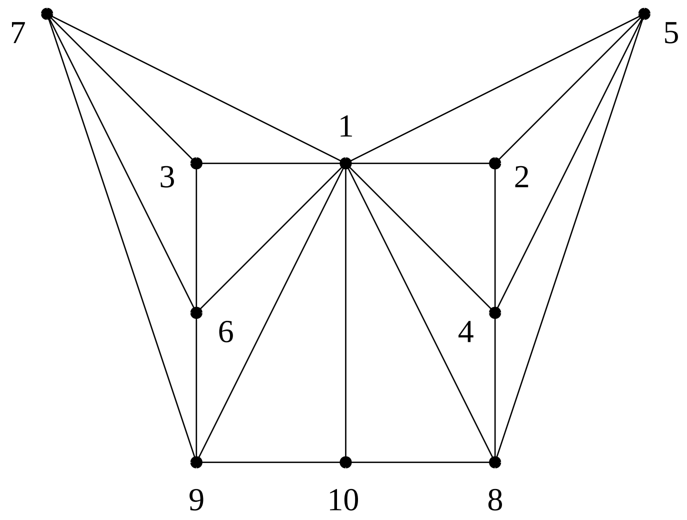


Abbildung 6.44: HCG für obiges Beispiel

6.3.2.2 *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.

6.3.2.2.1 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. 6.42 ergibt sich eine Zuordnung mit 4 benötigten Spuren (siehe Abb. 6.45).

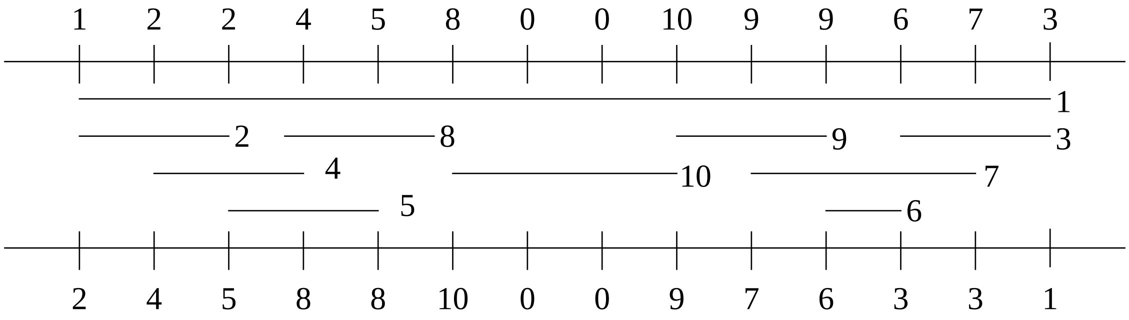


Abbildung 6.45: Ergebnis der Spurzuordnung mit dem *left edge*-Algorithmus

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.

6.3.2.2.2 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 Top$[k] = i$ und 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. 6.42 ist in Abb. 6.46 dargestellt.

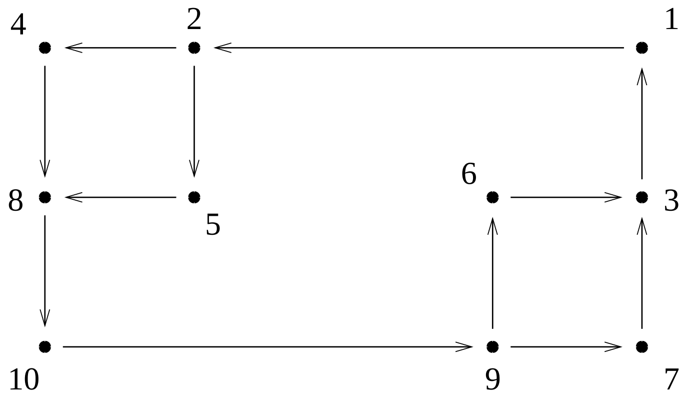


Abbildung 6.46: VCG des obiges Beispiels

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. 6.47 zeigt das Ergebnis des Beispiels nach Abb. 6.42.

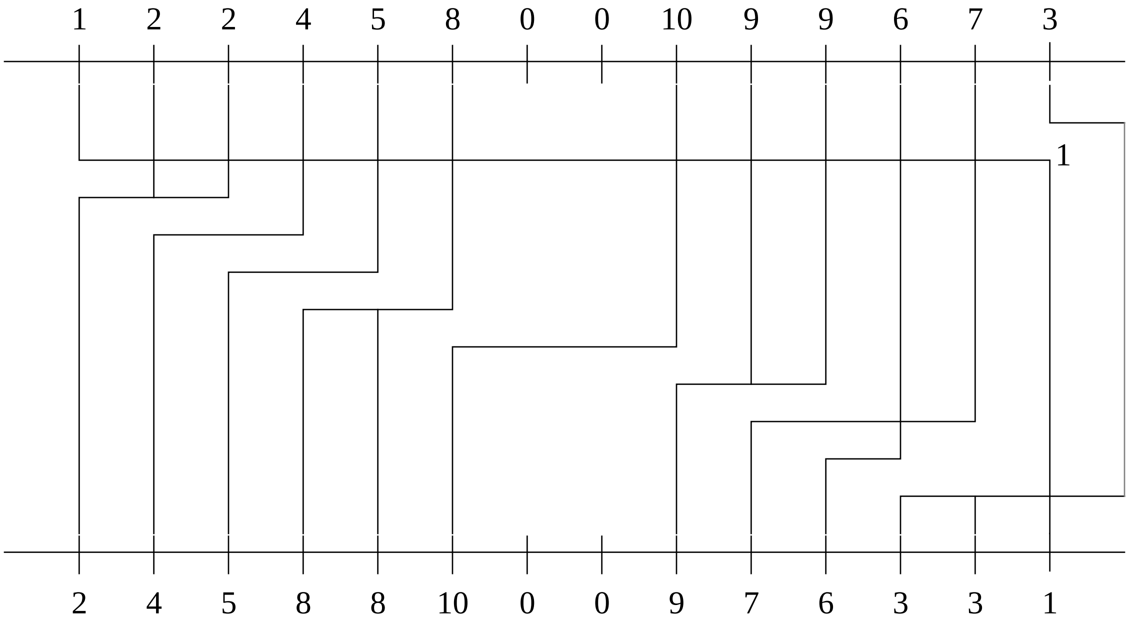


Abbildung 6.47: Spurzuordnung mit dem *constrained left edge*-Algorithmus

Wie aufgrund des Zyklus im Graphen der Abb. 6.46 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. 6.47) 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.

6.3.2.3 *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.
- 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. 6.48 (nach [Bur86]) zeigt je ein Beispiel für einen zu lösenden und einen nicht zu lösenden Zyklus.

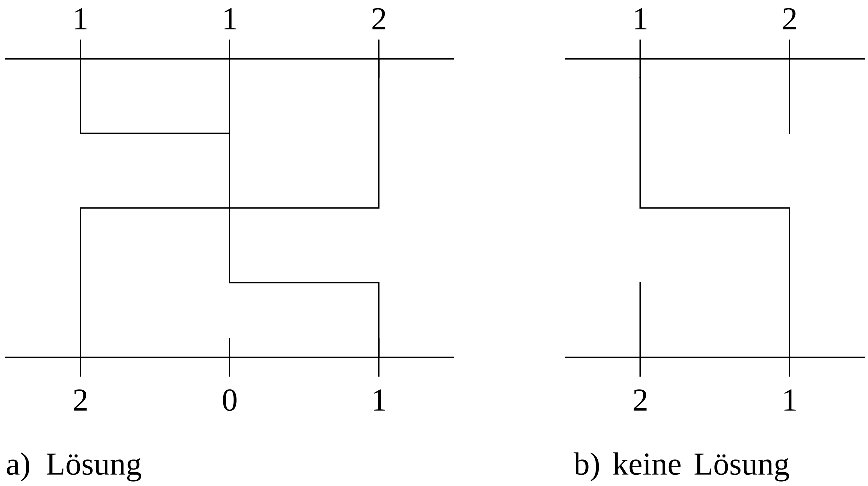


Abbildung 6.48: Verdrahtung bei zyklischen vertikalen Randbedingungen

Um Zyklen allgemein aufbrechen zu können, wird nunmehr die Beschränkung auf ein horizontales Segment pro Netz und Spalte aufgehoben (siehe z.B. Abb. 6.49).

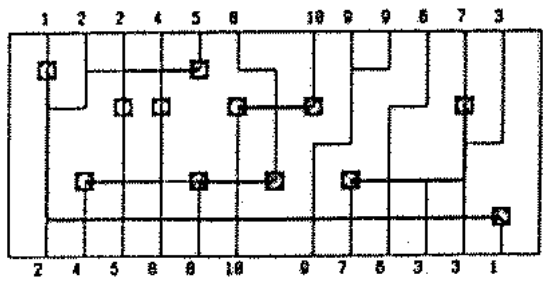


Fig. 15. Burstein's channel.

Abbildung 6.49: Kanalbeispiel nach Burstein (©IEEE)

6.3.2.4 *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 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. 6.50).

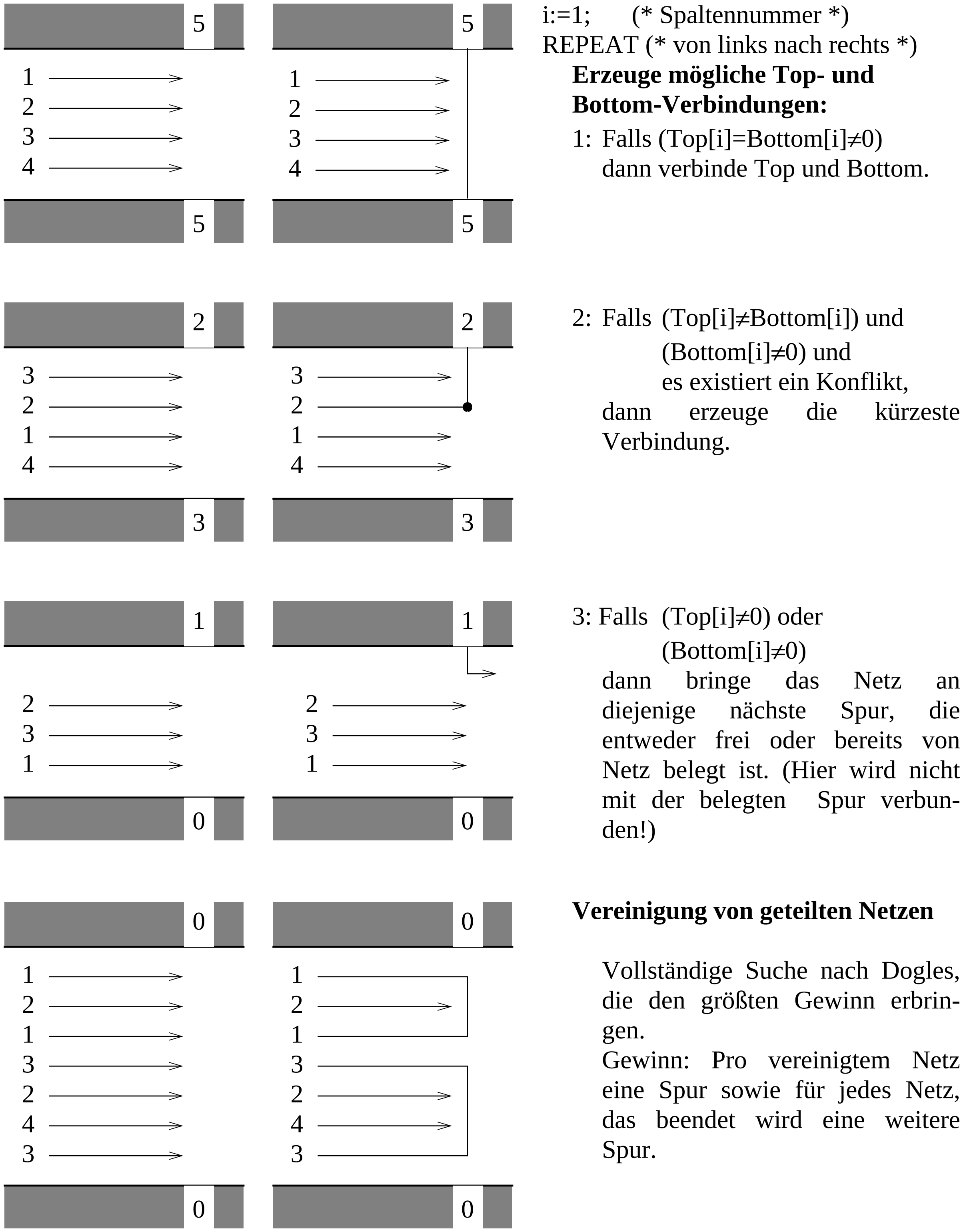


Abbildung 6.50: Greedy Channel Router: Anschluß-Verbindungen und Netzvereinigung

Ist ein Netz sowohl am oberen als auch am unteren Kanalrand anzuschließen, so erfolgt dies als erstes. Im zweiten Fall der Abb. 6.50 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. 6.50 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 6.51).

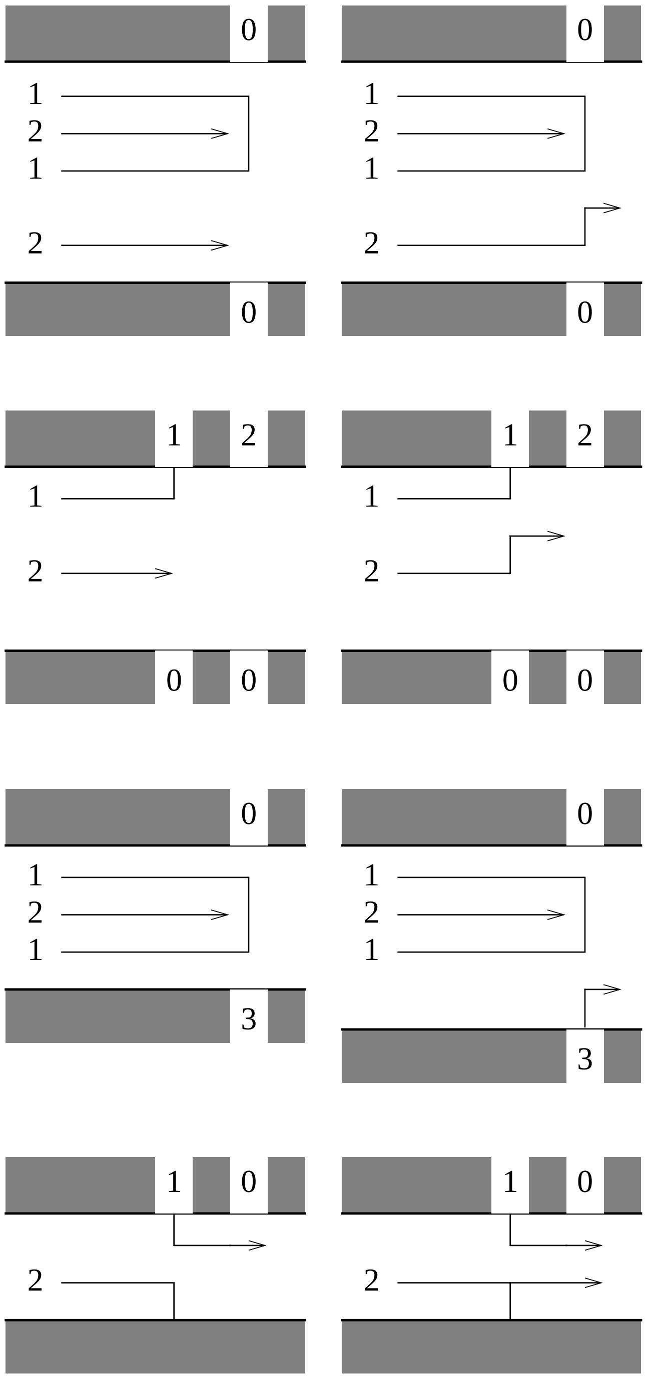


Abbildung 6.51: Greedy Channel Router: Annäherung und Kanalverbreiterung

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. 6.52 und Abb. 6.53 zeigen eine Lösung dieses Beispiels mit dem Greedy Channel Router. Die geteilten Netze und die vorsorglichen Spurwechsel nach oben und unten sind mehrfach vorhanden.

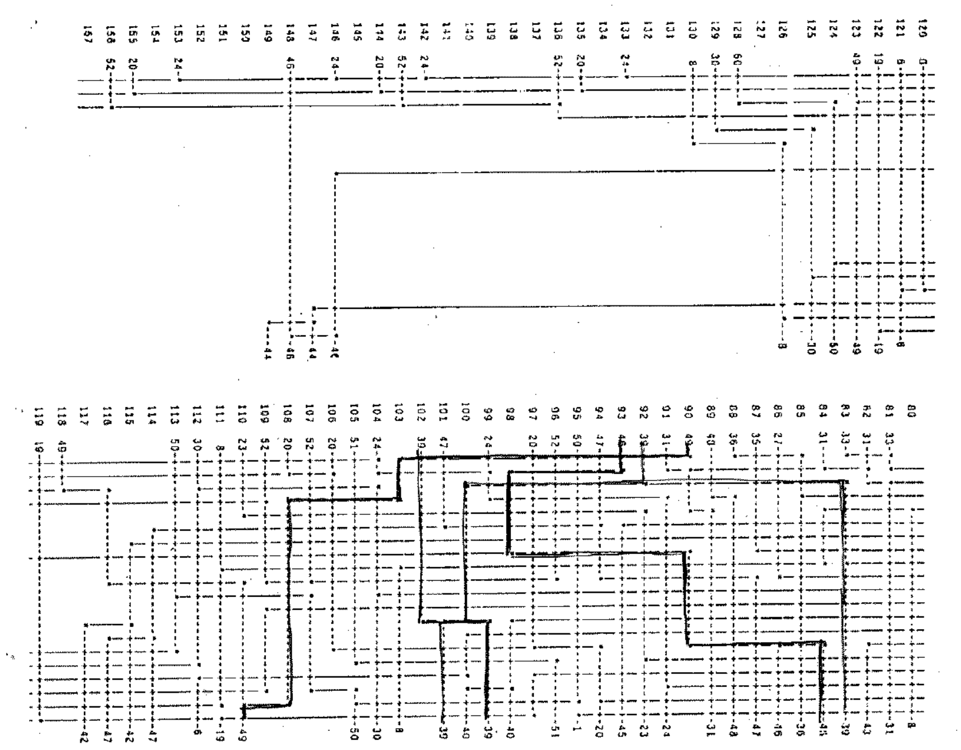


Abbildung 6.52: Lösung des schwierigen Beispiels von Deutsch mit dem Greedy Channel Router (20 Spuren; optimale Lös.:19 Spuren; ©1982 IEEE)

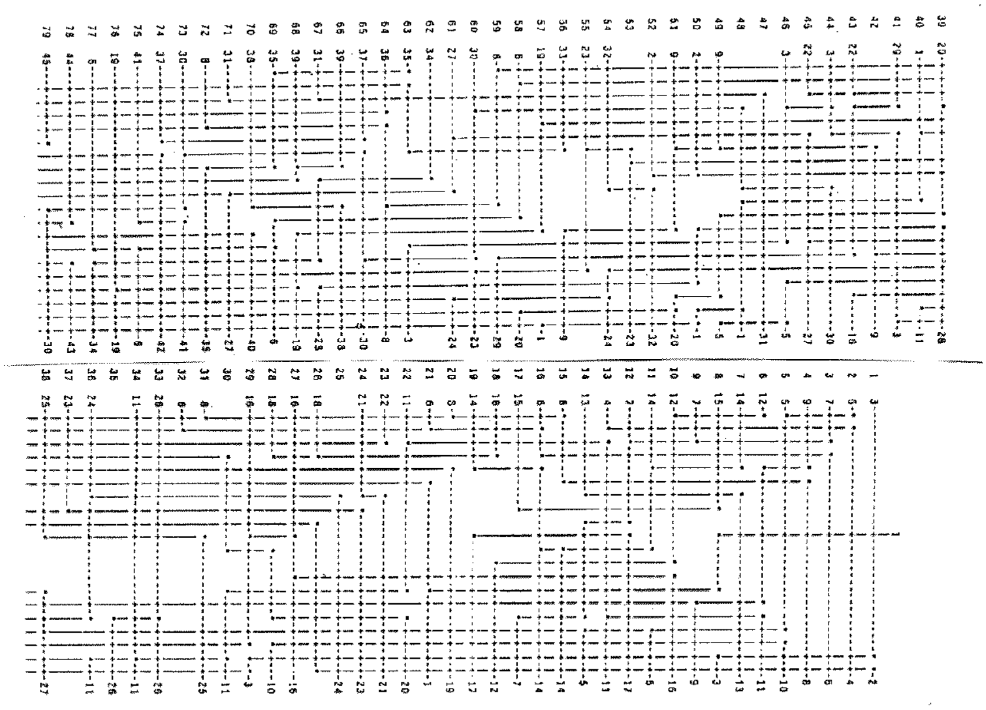


Abbildung 6.53: Beispiel von Deutsch (Fortsetzung; ©1982 IEEE)

6.3.2.5 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. 6.54 und Abb. 6.55).

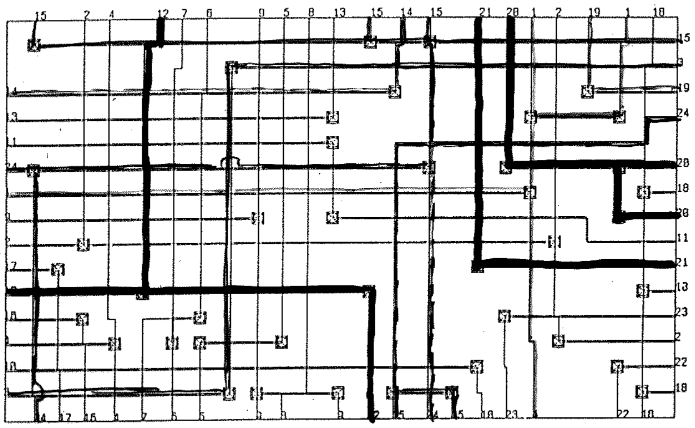


Fig. 13. Burstein's switchbox.

Abbildung 6.54: *Switchbox* nach Burstein (©IEEE)

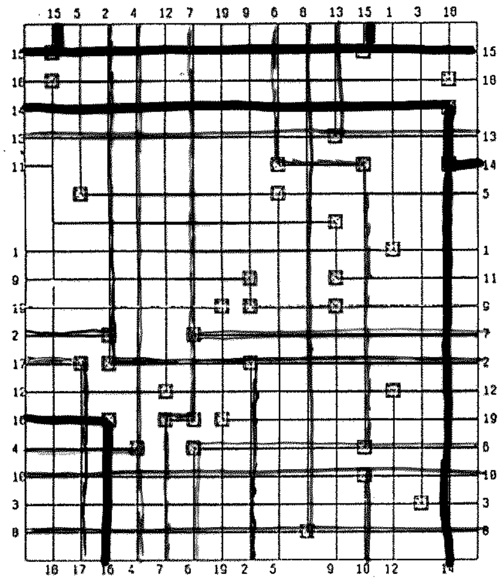


Fig. 14. The dense switchbox.

Abbildung 6.55: Sog. 'dichte' *Switchbox* (©IEEE)

6.3.2.6 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 6.56 zeigt die unterschiedlichen Kanalbreiten für insgesamt drei verschiedenen Verfahren.

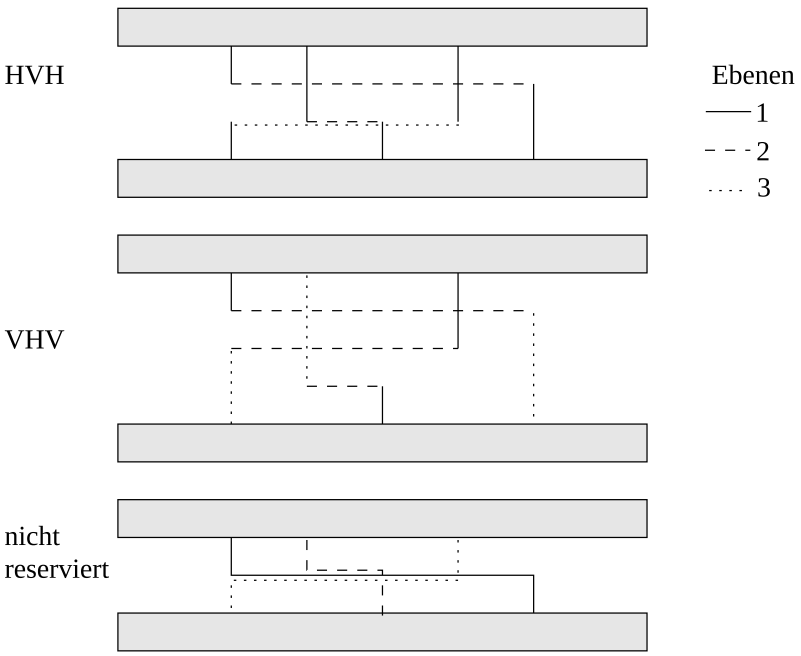


Abbildung 6.56: Kanalverdrahtung in drei Ebenen

6.3.3 Der Lee-Algorithmus

6.3.3.1 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. 6.57 bis 6.59):

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 169). Ein Weg entlang streng absteigender Marken ist also immer auch ein Weg minimaler Länge.

Bei einer Kantenlänge von n Feldern benötigt der Lee–Algorithmus $\mathrm{O}(n^2)$ 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 $\ell =$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.

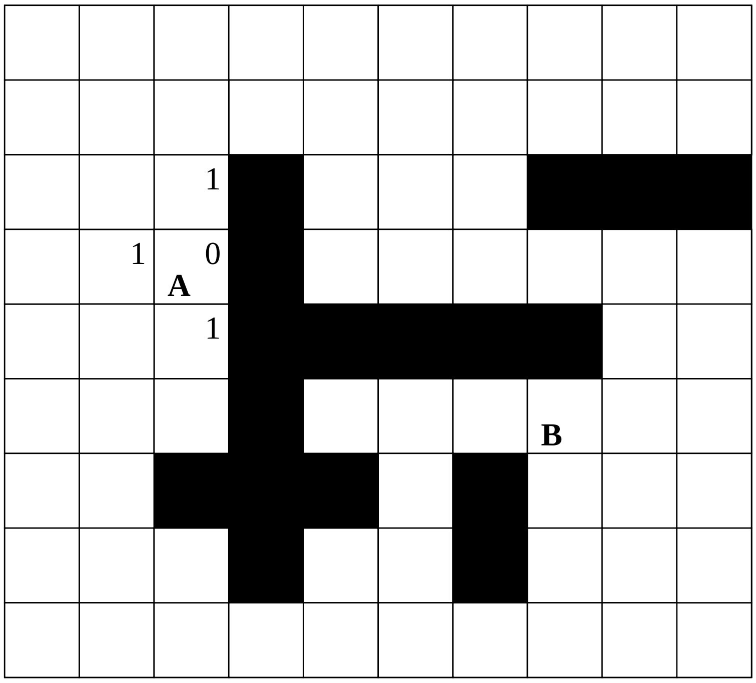


Abbildung 6.57: Markierung mit i=1

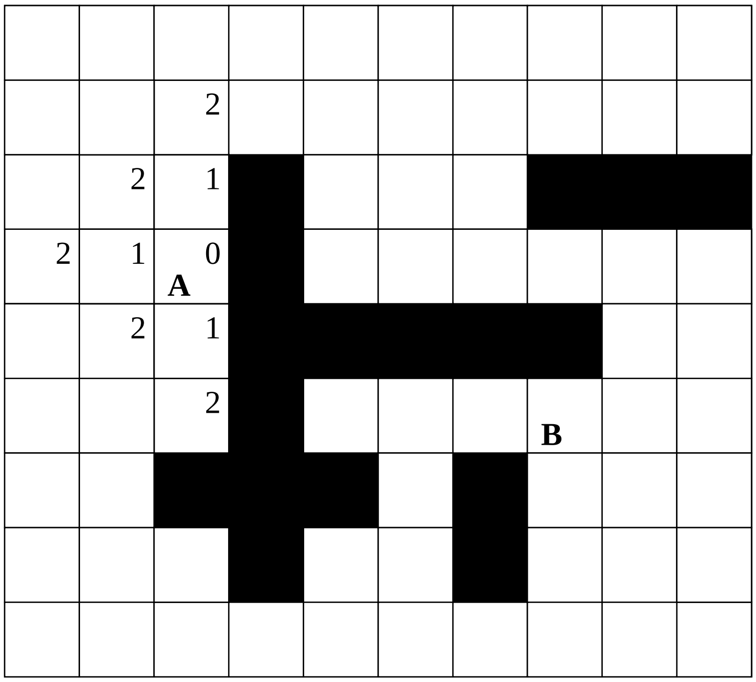


Abbildung 6.58: Markierung mit i=2

6.3.3.2 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

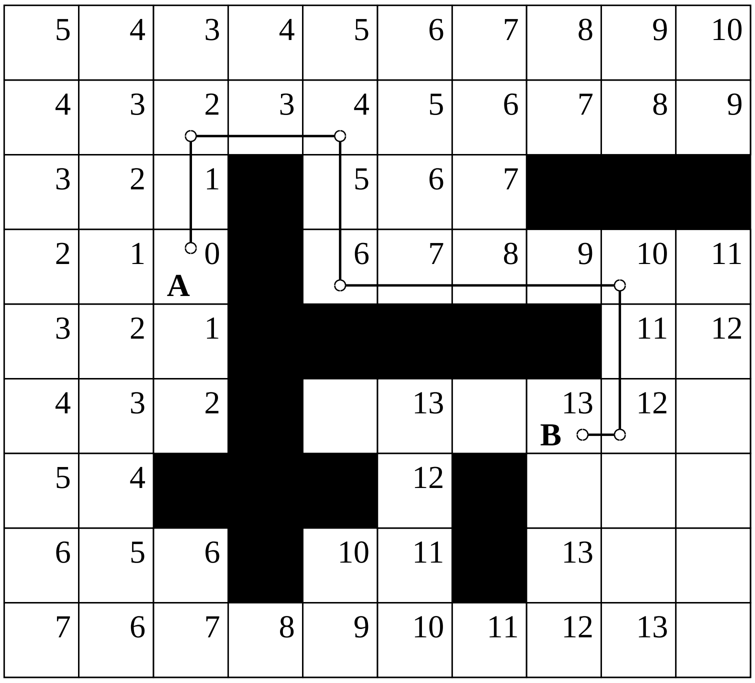


Abbildung 6.59: *backtrace*

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.

6.3.4 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.

6.3.4.1 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.

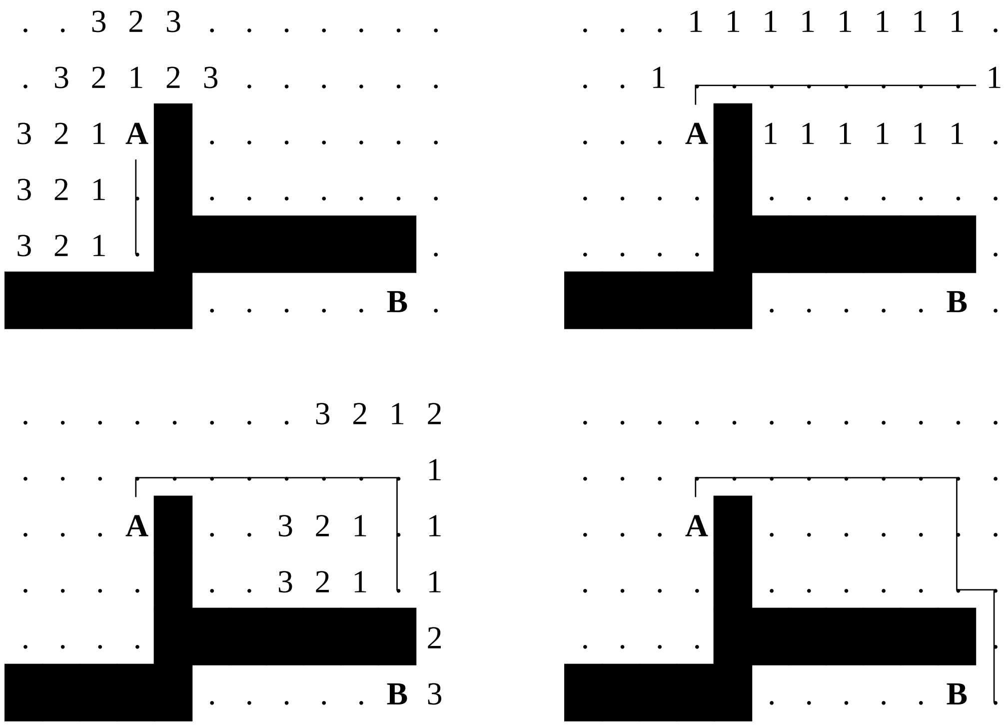


Abbildung 6.60: Beispiel für Soukup's Algorithmus (Zielpunkt ist **B**)

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. 6.60).

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.

6.3.4.2 *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.

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

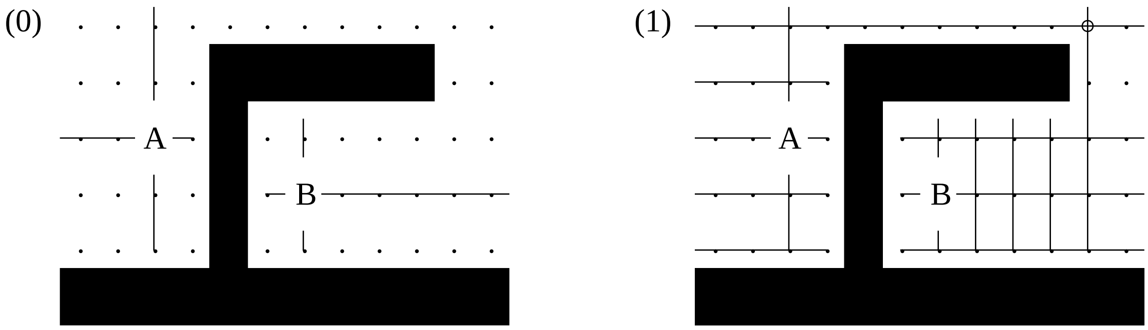


Abbildung 6.61: Zum Liniensuchverfahren von Mikami und Tabuchi

gespeichert werden.

6.3.4.3 *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.

6.3.4.4 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. 6.62).

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.

```
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

6.3.5 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

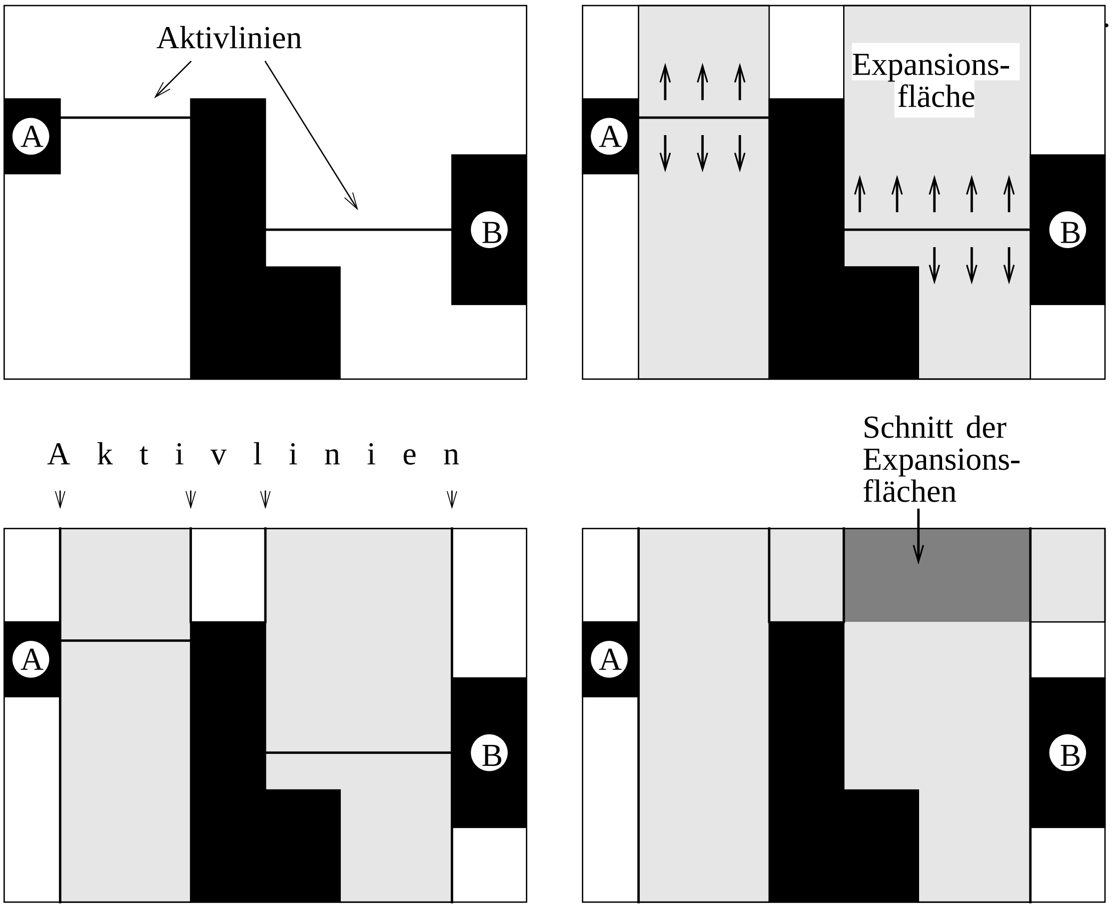


Abbildung 6.62: Zum Linienerweiterungs-Algorithmus

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.

Kapitel 7

Entwurfsüberprüfung (Validierung)

7.1 Methoden der Entwurfsüberprüfung

Eine der zentralen Aufgaben im Entwurfsprozess ist die **Überprüfung** (Validierung) der Ausgangsspezifikation sowie des vollständigen Weges bis hin zur endgültigen Implementierung (siehe Abb. 7.1). Alle Techniken, welche hierbei helfen können, werden in der Praxis dankbar angenommen.

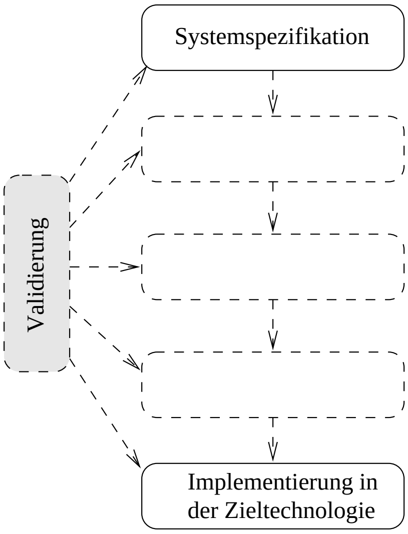


Abbildung 7.1: Kontext der Validierung

Es wäre schön, wenn man formal zeigen könnte, dass das Modell am Endpunkt des Weges jenes am Anfangspunkt realisiert. Leider sind für diese komplexe Aufgabe keine Verifikationstechniken bekannt. Ersatzweise muss daher auf eine Überprüfung auf allen Zwischenebenen erfolgen.

Wir können dabei zwischen verschiedenen Überprüfungstechniken unterscheiden:

1. Die **Simulation** basiert auf dem Ausführen von Systembeschreibungen. Meist wird mit dem Begriff der Simulation das Studium von Fallbeispielen verbunden. Sofern aus Zeitgründen nicht alle Fälle simuliert werden können, kann nach Dijkstra damit nicht die Abwesenheit von Fehlern nachgewiesen werden. Eine vollständige Simulation könnte Aussagen mit 100%-iger Sicherheit garantieren. Wegen der Vielzahl möglicher Eingabewerte gelingt sie aber für komplexe Systeme nicht. Simulation kann bei komplexen Systemen also nur die Anwesenheit, aber nicht die Abwesenheit von Fehlern nachweisen.
2. Die **Verifikation** versucht auf der Basis formaler Analysen Eigenschaften eines Systems, wie z.B. die Äquivalenz mit einem anderen Modell oder die Gültigkeit bestimmter Aussagen nachzuweisen. Die Verifikation erfordert zwar nicht unbedingt die Betrachtung aller Eingabemuster. Sie gelingt aber bislang nur für Teilschaltungen oder für einfache Prozessoren. Die (formale) Verifikation gewinnt in jüngster Zeit zunehmends an Bedeutung, insbesondere aufgrund des Pentium-Gleitkommafehlers. Der Stand der Technik wir in einem Schwerpunktheft der Zeitschrift **it+ti** beschrieben [Bro97].

3. Die Validierung kann erleichert werden, wenn ein System Selbsttest-Maßnahmen enthält (wie z.B. die Verbindung aller Register untereinander in Form eines langen Schieberegisters. Weiter kann man auch versuchen, per Testmuster-Erzeugung generierte Eingabemuster in der Simulation zu verwenden und so die klassischerweise getrennten Bereiche der Entwurfs- und der Testverfahren miteinander zu kombinieren.

Zunächst werden wir uns genauer mit der Simulation beschäftigen.

7.2 Simulation

7.2.1 Simulationsebenen

Simulationen werden während des Systementwurfs und der Schaltungsfertigung auf unterschiedlichen Abstraktionsebenen angewandt.

7.2.1.1 Simulation der Spezifikation

Vielfach möchte man bereits die Spezifikation simulieren. Bei sog. **ausführbaren Spezifikationen** gelingt dies sehr leicht. Im Falle reiner algebraischer Spezifikationen ist heir besonderer Aufwand erforderlich. Simulationsverfahren haben wir im vorigen Kapitel in einigen Fällen bereits zusammen mit der Semantik der jeweiligen Sprache beschrieben.

7.2.1.2 Verhaltenssimulation

Als Ebene darunter können wir die sog. algorithmische Ebene verstehen, bei der wir typischerweise einzelne sequentielle Prozesse betrachten. Diese Ebene kann auch noch weiter aufgeteilt werden: so kann man diese Prozesse durch Compilation und Ausführung der entsprechenden Programme realisieren, ohne damit Informationen über die Verwendung des Befehlssatzes der später zu verwendenden Zielmaschine zu erhalten. Man kann für diese Maschine aber auch einen Befehlssatzsimulator erstellen und diesen den Maschinencode ausführen lassen.

In der Verhaltenssimulation sind die Operationen noch nicht an Hardware-Bausteine gebunden. Es wird lediglich das Input/Output–Verhalten beschrieben. Dies kann u.a. mit gewöhnlichen imperativen Sprachen geschehen. Programmiersprachen wie PASCAL, C, SIMULA u.s.w. sind hierfür in Benutzung. Mehr Unterstützung z.B. hinsichtlich der interaktiven Arbeitsweise, möglicher Signalwerte u.s.w. bieten in der Regel spezielle Hardware-Beschreibungssprachen wie VHDL oder Verilog.

Aufgrund der Abstraktion von der realen Hardware ist es nicht mehr immer klar, ob eine Beschreibung in einer Hochsprache der realen Hardware wirklich entspricht, wie z.B. im Falle von reinen Typkonvertierungen.

7.2.1.3 Simulation von RT-Strukturen

Die nächste Abstraktionsstufe bildet die Simulation von RT-Strukturen. Die Simulation besteht aus der Nachbildung der Signalausbreitung auf den Verbindungen und der Nachbildung der Verknüpfung der Signale in Hardware-Moduln. Die Breite der Bitvektoren ist nicht auf 1 beschränkt und neben den logischen werden auch arithmetische Verknüpfungen durchgeführt. Alle Verknüpfungen (Operationen) sind an Hardware-Bausteine gebunden.

7.2.1.4 Logiksimulation

Die **Logiksimulation** (engl. *logic simulation*) benutzt als Modell Boolesche Funktionen und berechnet aus Eingaben die resultierenden Ausgaben, die ggf. mit einer gewissen Verzögerung den Signalen zugewiesen werden. Logiksimulatoren liefern also eine konkrete Aussage über Zeitabläufe.

Anwendbar sind sie nur dann, wenn zwischen Eingaben und Ausgaben unterschieden werden kann. Werden Schalter bidirektional benutzt (was natürlich möglich ist), so muss z.B. die Schaltersimulation benutzt werden (siehe unten).

Häufig wird diese Simulation auch als **Gattersimulation** (engl. *gate (level) simulation*) bezeichnet. Mit dieser zweiten Bezeichnung wird aber nicht mehr zwischen einer abstrakten Booleschen Funktion und deren Implementierung unterschieden.

7.2.1.5 Schaltersimulation

Auf der nächsten Abstraktionsstufe werden Transistoren als Schalter modelliert. Daher heißt diese Ebene der Art der Simulation **Schaltersimulation** (engl. *switch level simulation*). Wesentliches Ziel der Schaltersimulation ist die Behandlung von Schaltungen mit bidirektional betriebenen Transistoren, für die kein äquivalentes Gattermodell existiert oder bekannt ist.

Im Unterschied zur Gattersimulation muss auch das Speichern von Information auf Kapazitäten berücksichtigt werden. Bezüglich der Kapazitäten (und der Kanalwiderstände) wird nur zwischen einer kleinen Zahl von Stärkeklassen unterschieden. Eine größere Stärke setzt ihren Wert stets gegenüber einer solchen mit geringerer Stärke durch.

Es werden immer noch digitale Signale betrachtet.

Statt eines exakten Zeitverhaltens ist aufgrund üblicher Simulationsalgorithmen nur der aufgrund einer Eingabe resultierende stabile Zustand bekannt. Falls kein stabiler Zustand existiert (z.B. bei Oszillatoren), liefert die Simulation evtl. kein sinnvolles Ergebnis.

Die Schaltersimulation ist u.a. wegen der Bidirektionalität langsamer als die Logiksimulation. Man versucht daher, für möglichst große Teile einer Schaltung die Richtung der Signalausbreitung vorab zu bestimmen und für diese dann wie bei der Logiksimulation nur die Boolesche Funktion zu berechnen.

7.2.1.6 Timing-Simulation

Ziel der Timing–Simulation sind Aussagen über den zeitlichen Verlauf von Strömen und Spannungen in einer Schaltung. Von der Schaltkreissimulation unterscheidet sich die Timing–Simulation durch stark vereinfachende Transistor–Modelle [WE85]. So werden beispielsweise alle Kapazitäten als konstante Kapazitäten gegenüber Masse betrachtet. Die Timing–Simulation ist bis zu zwei Zehnerpotenzen schneller als die Schaltkreissimulation. Es können jedoch nicht alle Schaltkreise damit simuliert werden. Die Genauigkeit liegt deutlich unter der von SPICE.

7.2.1.7 Schaltungssimulation

Ziel der **Schaltungssimulation** (engl. *circuit simulation*, [Rue86]) ist die Bestimmung von Strömen und Spannungen in Schaltungen aus mehreren Bauelementen in Abhängigkeit von der Zeit. Basis der Simulation sind die beiden Kirchhoff'schen Gesetze sowie die charakteristischen Gleichungen der Bauelemente. Das resultierende Differentialgleichungssystem ist nichtlinear, falls vorhandene Halbleiter nichtlinear modelliert werden.

Transistoren werden häufig durch Ersatzschaltbilder beschrieben, deren Parameter z.B. mittels Bauelementsimulation bestimmt werden können. Abbildung 7.2 zeigt ein Ersatzschaltbild eines realen MOS–Transistors. Der zeitabhängige Anteil wird hier durch Kapazitäten dargestellt und der verbleibende Transistor wird als verzögerungsfrei betrachtet.

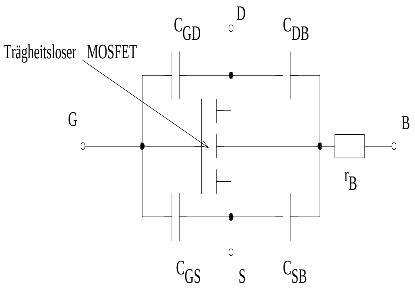


Abbildung 7.2: Einfaches Ersatzschaltbild eines MOS-Transistors

Die Schaltungssimulation ist für komplexe Schaltungen in der Regel nicht durchführbar. Die Grenze liegt z.Zt. etwa bei gut hundert Transistoren.

Abb. 7.3 zeigt als Ergebnis einer solchen Simulation die Reaktion eines Inverters auf ein Eingangssignal [Ban85]. Die drei Kurven entsprechen einer Belastung des Ausgangs mit 0, 10 bzw. 50 pF. Letzteres gilt als typisch für eine Belastung mit 5 Eingängen.

Die wohl größte Verbreitung besitzt der **SPICE**–Simulator. Ein CMOS–NAND–Gatter mit einer Schaltung nach der Abbildung 7.4 würde für SPICE wie in Abb. 7.5 beschrieben.

`MN1, MN2` usw. sind Komponenten (Parts) des Bausteins `NAND2` mit den E/A–Leitungen `VDD, VSS, A, B` und `OUT`. "`M`" als erstes Zeichen des Komponentennamens definiert einen MOS–Transistor, "`C`" einen Kondensator. Für jeden

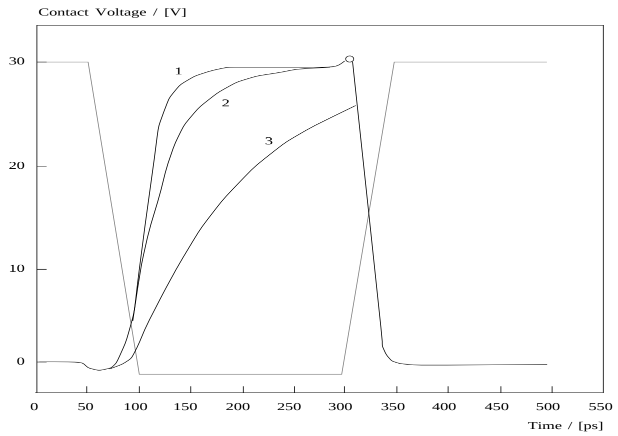


Abbildung 7.3: Verlauf der Ausgangsspannung eines Inverters (©IEEE)

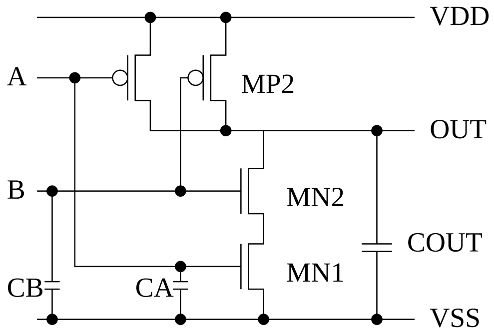


Abbildung 7.4: CMOS-NAND-Gatter

```
.SUBCKT NAND2 VDD VSS A B OUT
MN1 I1 A VSS VSS NFET W=8U L=4U AD=64P AS=64P
MN2 OUT B I1 VSS NFET W=8U L=4U AD=64P AS=64P
MP1 OUT A VDD VDD PFET W=16U L=4U AD=128P AS=128P
MP2 OUT B VDD VDD PFET W=16U L=4U AD=128P AS=128P
CA A VSS 50fF
CB B VSS 50fF
COUT OUT VSS 100fF
.ENDS
```

Abbildung 7.5: Beschreibung eines 2-fach NAND-Gatters in SPICE

Transistor wird die Kanallänge und –breite in μm, sowie die Source– und Drain–Fläche (“`AD`” und “`AS`”) zwecks Berechnung der Kapazitäten angegeben. `CA, CB` und `COUT` stellen zusätzliche Leitungskapazitäten an den Ein– und Ausgängen dar.

Die Simulation von Transistoren erfolgt in SPICE mit drei verschiedenen Methoden:

1. Durch Berechnung von Strömen und Spannungen nach den Kirchhoff'schen Regeln.
2. Analytisch aufgrund detaillierter Information über den Aufbau des Transistors. Es können bis zu 39 (!) Parameter gesetzt werden [Vla87b].
3. “Halb–empirisch” mit Anpassungen an gemessene Werte.

7.2.1.8 Bauelementsimulation

Ziel der **Bauelementsimulation** (engl. *device simulation*, [Eng85, Dir87]) sind Aussagen über Ströme und Spannungen innerhalb einzelner, extern mit Spannungen versorgter Bauelemente, vor allem von Transistoren. Ein weiteres Ergebnis bilden die Werte von Widerständen, Kondensatoren, Strom– und Spannungsquellen, die in Ersatzdarstellungen von Transistoren verwendet werden.

Vorausgesetzt wird eine Kenntnis des räumlichen Aufbaus der Transistoren, z.B. aufgrund einer Prozeßsimulation.

Wie bei der Prozeßsimulation sind Systeme von nichtlinearen Differentialgleichungen zu lösen. Stand der Technik ist die Berücksichtigung von 2 Dimensionen mit einigen Tausend Stützwerten. Die Simulation in 3 Dimensionen und die Nutzung von Feldrechnern sind Gegenstand der Forschung.

Abb. 7.6 zeigt eine per Simulation bestimmte Transistor–Kennlinie [IME86].

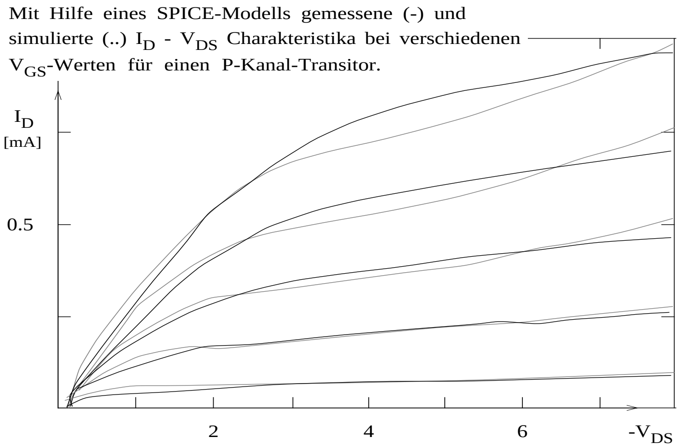


Abbildung 7.6: Kennlinie eines p-Kanal Transistors

7.2.1.9 Prozesssimulation

Die unterste Ebene bildet die **Prozesssimulation** (engl. *process simulation*). Wesentliches Ziel der Prozeßsimulation ist die Bestimmung der Dotierungsprofile von Halbleitern. Zu berechnen ist hierbei vor allem der Transport der Dotierungsmaterialien innerhalb des Kristallverbandes während und nach der Fertigung. In der Regel führen die physikalischen Modelle auf nichtlineare Differentialgleichungen, für die verschiedene spezielle Lösungswege entwickelt wurden. Als Ergebnis liefert die Prozesssimulation eine Beschreibung der in einem bestimmten Herstellungsprozeß erzielten räumlichen Verteilung der chemischen Elemente.

Abb. 7.7 zeigt die Konzentration von Bor–Atomen in Abhängigkeit von der Entfernung zur Oberfläche [IME86], und zwar zum Vergleich gemessene und durch Simulation gewonnene Werte.

Zum Bereich der Prozeßsimulation gehören ferner Untersuchungen zum Verhalten verschiedener Fotolacke sowie verschiedener Ätzungsprozesse.

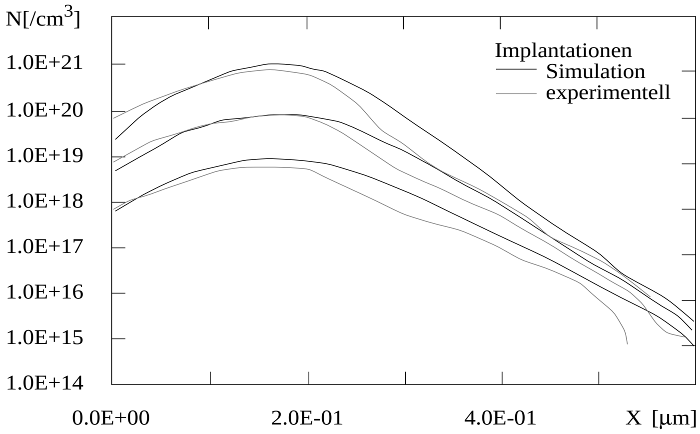


Abbildung 7.7: Dotierungsprofil nach Diffusionsdotierung

7.2.2 Emulation

Als spezielle Form der Simulation wird die Emulation benutzt. Das zu realisierende System wird hierbei durch andere Hardware derartig nachgebildet, dass eine Ausführungsgeschwindigkeit in der Nähe der späteren Arbeitsgeschwindigkeit erreicht werden kann. Beliebt ist in diesem Zusammenhang die Benutzung von FPGAs, die über ein CAD-System entsprechend konfiguriert werden [Qui].

Allerdings muss bei dieser Methode aus der Spezfikation nicht nur die spätere echte Realisierung erzeugt werden, sondern auch das Emulationsmodell. Dies kann einen erheblichen Zusatzaufwand bedeuten, was im Extremfall zu einer Verlangsamung statt zu einer Beschleunigung des Entwurfsprozesses führt.

7.3 Test

7.3.1 Überblick

Die Validierung kann erleichert werden, wenn ein System Selbsttest-Maßnahmen enthält (wie z.B. die Verbindung aller Register untereinander in Form eines langen Schieberegisters. Weiter kann man auch versuchen, per Testmuster-Erzeugung generierte Eingabemuster in der Simulation zu verwenden und so die klassischerweise getrennten Bereiche der Entwurfs- und der Testverfahren miteinander zu kombinieren[1] Wir werden uns daher mit einigen Grundzügen von Testverfahren beschäftigen.

Zum Test digitaler Systeme müssen folgende Teilaufgaben gelöst werden:

1. die Erzeugung von Prüfmustern,
2. die Anwendung der Prüfmuster,
3. die Bestimmung der Fehlerabdeckung.

Für die Testanwendung gibt es interne und externe Methoden. Den internen Testverfahren ist gemeinsam, dass sie in der normalen Systemumgebung durchgeführt werden können und dass keine oder nur eine einfache Apparatur zur Überwachung des Tests erforderlich ist. Im Extremfall dringt nach außen nur die Meldung "defekt" oder "Test bestanden". Voraussetzung dafür ist der Einbau der benötigten Testfunktionen in die Schaltung. Der Ablauf eines internen Tests kann entweder simultan zum normalen Betrieb (*concurrent, on line*), oder in einem gesonderten Testlauf (*off line*) erfolgen.

Ein simultan zum normalen Betrieb laufender Test benötigt Redundanz in der Schaltung oder in der Codierung von Daten und Steuersignalen. Mit redundanten Codes (z.B. Hamming–Codes) lassen sich Hardware-Fehler durch das Auftreten ungültiger Codeworte erkennen. Redundante Schaltungsteile werden in Form von duplizierten Verarbeitungszweigen, etwa als doppelte arithmetisch–logische Einheiten oder mehrfache ganze Prozessoren eingesetzt. Ebenfalls zusätzliche Vergleichsschaltungen (Komaratoren)zeigen bei Nichtübereinstimmung der Ergebnisse einen Fehler an.

Schaltungen, die sowohl die Gültigkeit ihrer Eingabedaten als auch ihre eigene korrekte Arbeitsweise überprüfen, werden *totally self–checking circuits* gennant. Sie nutzen Redundanz in Hardware und Codierung, um in Bezug auf eine gewisse Klasse von Fehlern (meist nur Einzelfehler) vollständig selbsttestend zu sein. Von totaler Fehlersicherheit kann natürlich nicht gesprochen werden.

Bei Normalbetrieb laufende Tests sind besonders wertvoll, wenn es darum geht, intermittierende oder transiente Fehler zu erfassen, die in einem gesonderten Testlauf vielleich selten oder nie auftreten würden.

Off line-Testverfahren laufen entweder in Betriebspausen der normalen Operation (*idle loop*), oder in einem speziellen Testmdous. Für Prozessor–Systeme gibt es reine Software– Lösungen, die als Selbsttest–Programme primär den fehlerfreien Zustand des Geamtsystems bestätigen sollen, oder als Diagnose–Routinen erkannte Fehler in bestimmten Schaltungesteilen zu lokalisieren versuchen. Andere Methoden rekonfigurieren das System für einen speziellen Testbetrieb und nutzen dann zusätzliche Schaltungsteile und Verbindungen. Beispiele dafür sind der *scan-path*– und der BILBO–Ansatz. Anwendungen, die über spezielle Testhardware verfügen und auch die Testauswertung intern vornehmen (z.B. BILBO), werden als eingebauter Test (*built–in–test*) bezeichnet.

Externe Testverfahren werden hauptsächlich zur Prüfung von einzelnen Baulelenten, integrierten Schaltungen und Baugruppen eingesetzt. Hier ersetzt eine spezielle Prüfschaltung oder ein meist kommerzieller Testautomat (*automatic test equipment*=ATE) die normale Systemumgebung. Zusätzlich wird, soweit vorhanden, auf weitere Mess– und Einspeisungspunkte zugegriffen.

Ein Testautomat sorgt für die Zuführung von Testmustern und die Auswertung der Antworten mit einer für den Prüfling nach Möglichkeit maximalen Geschwindigkeit. Obwohl für die Programmierung der Automaten vom Hersteller meist eine komfortable Schnittstelle geboten wird, muss man beim Test einer mit VLSI–Bausteinen bestückten

[1] Derartige Kombinationsmethoden sind Gegenstand des *High-Level Design Verifikation and Test Workshop* [HLD98].

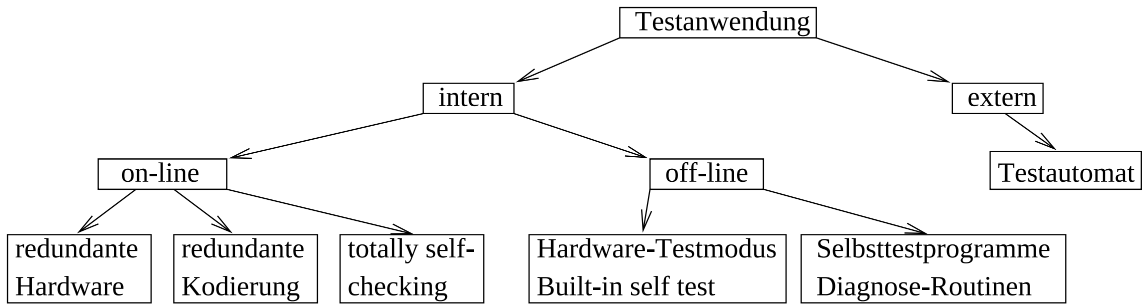


Abbildung 7.8: Testanwendungsmethoden

Platine mit 6–9 Monaten Programmierzeit rechnen. Bei Testerzeugung kann der Automat nur auf die bekannten Algorithmen oder manuelle Eingaben zurückgreifen.

Beim sogenannten *in–circuit–test* für Baugruppen (*boards*) können die Testautomaten über eine Vielzahl von Testköpfen auf Leitungspunkte innerhalb der Schaltung zugreifen. Durch kurzzeitiges Induzieren von teilweise hohen Strömen (*overdrive*) werden ohne Rückssicht auf die Umgebung die gewünschten Spannungspegel in der Schaltung eingestellt und die Bauteile einzeln und unabhänging geprüft. Nachteilige Folgen für die Lebensdauer der Bauelemente sind nicht messbar.

7.3.2 Built–in self test

7.3.2.1 Scan–Path

Beim Test von Schaltungen mit internen Zuständen besteht das größte Problem darin, vor Durchführung eines Tests die Schaltung zunächst einmal in einem bestimmten Zustand Z zu bringen und nach Durchführung des Tests (d.h. nach Eingabe eines Eingangsvektors X) den neuen Zustand von Z^+ von außen anhand von Ausgaben Y zu beobachten. Konnte man alle zustandsspeichernden Register parallel Setzen und auch Auslesen, so hätte man diese Problem beseitigt. Für diese Lösung würde man aber zu viele Leitungen (insbesondere zu viele Leitungen zwischen Chip und Umgebung) benötigen. Ein möglicher Kompromiss ist das bitserielle Setzen und Lesen der Register. Zu diesem Zweck werden die Register als ein langes Schieberegister realisiert:

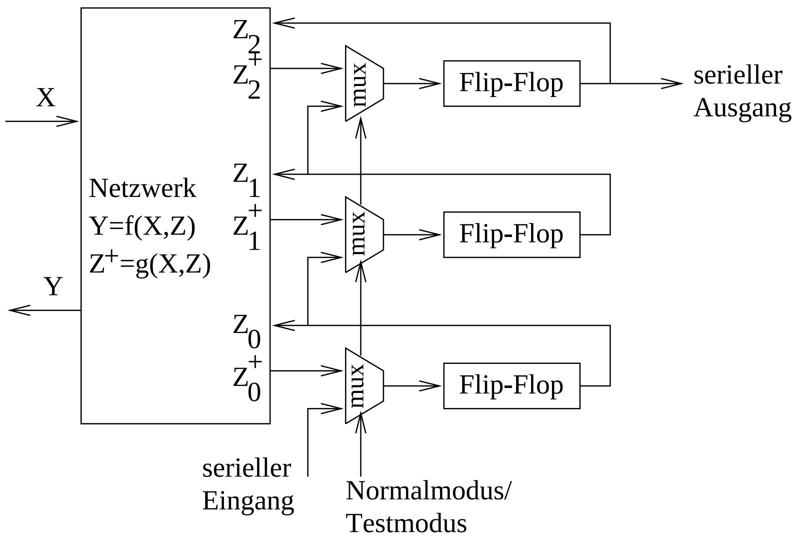


Abbildung 7.9: Scan Path

Um Hardware zu testen, wird beim Scan–Path–Prinzip zunächst durch bitserielles Laden der gwünschte Zustand hergestellt, ein Testvektor X angelegt und anschließend der neue Zustand bitseriell ausgelesen. Der Vorteil der Methode besteht u.a. darin, dass die Testmuster–Erzeugung für kombinatorische Schaltungen anwendbar ist. Wegen der u.U.recht langen Schieberegister kann man aber nicht immer vor und nach jedem Test einen vollständigen Schiebezyklus vorsehen. Der zusätzliche Hardware–Aufwand soll zwischen 4% und 20% betragen, je nachdem, welche Design–Regeln ohne Scan–Path benutzt wurden.

Bei IBM ist diese Technik in der Variante LSSD (*Level Sensitive Scan Design*) für alle Entwürfe zwingend vorgeschrieben. Bei Siemens ist diese Technik unter dem Namen **Prüfbus** im Einsatz. Zur Reduktion der Lade– und Auslesezeit ist ferner die Technik *random access scan* entwickelt worden, in der adressierbare Flip–Flops verwendet werden.

7.3.2.2 Signaturanalyse

Bei jeder Testanwendung muss das Testergebnis mit dem Sollwert verglichen werden. Wegen der großen Menge zu testender Ergebnisse sind Methoden zur Datenreduktion interessant. Allen anderen Methoden ist die sog. Signaturanalyse. Sie basiert im Prinzip auf der Erzeugung von CRC (*cyclic redundancy check*) Prüfzeichen, d.h. auf zyklischen Codes. Während der Durchführung des Tests mit einer bestimmten Prüfmuster–Sequenz beobachtet man eine bestimmte Leitung einer Schaltung und berechnet für die Bitfolge auf dieser Leitung das zugehörige CRC–Zeichen. Das CRC–Zeichen ist für diese Bitfolge charakteristisch "wie eine Unterschrift". Von der Sollfolge abweichende Bitfolgen erzeugen mit hoher Wahrscheinlichkeit vom Soll-CRC–Zeichen abweichende CRC–Zeichen. Während bei der Datenübertragung die CRC–Zeichen mit übertragen werden, werden die korrekten CRC–Zeichen bei der Signaturanalyse für eine bestimmte Testmuster–Folge berechnet und z.B. in den Schaltplan eingetragen. Der Wartungsingenieur kann das Ist–CRC–Zeichen aufnehmen und mit dem Schaltplan vergleichen.

Die Signatur-Analyse wird durch ein linear rückgekoppeltes Schieberegister realisiert. Wenn alle Testantworten über den seriellen Eingang in das Schieberegister übernommen worden sind, steht im Register die Signatur $f(A)$ des Tests.

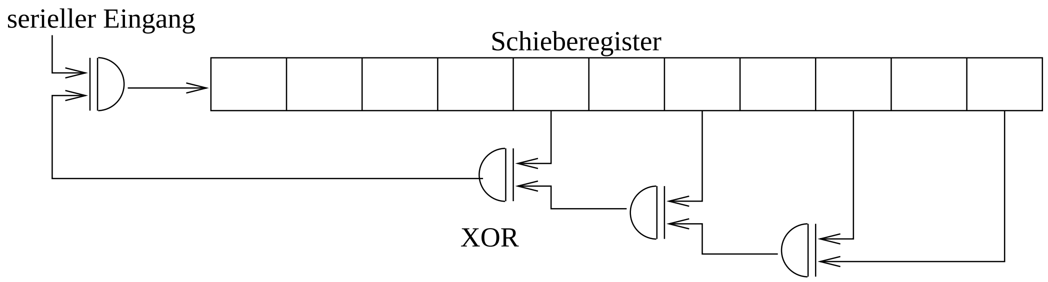


Abbildung 7.10: Bildung einer Signatur

Derartige Schieberegister kann man z.B. in Gehäuse ähnlich demjenigen der digitalen Fieberthermometer einbauen. Bei einer Länge der Eingabefolge von n Bit und einem 16 Bit Schieberegister gibt es 2^n mögliche Eingaben, die auf 2^16 verschiedene Signaturen abgebildet werden. Die lineare Rückkopplung bewirkt eine gleichmäßige Verteilung über alle Signaturen. Dies bedeutet:

Für n=16: Jede mögliche Eingabesequenz besitzt eine eindeutige Signatur.

Für n=17: Je genau zwei Eingaben erhalten die gleiche Signatur. Bei der Fehlersuche wird genau eine von $2^{17}-1$ existierenden falschen Antworten nicht erkannt.

Für n=18: Bei Gleichverteilung erhalten je vier Eingaben erhalten dieselbe Signatur, davon entspricht nur eine Eingabe der erwarteten Testantwort.

Allgemein gilt: Je $2^{(n-16)}$ Eingaben erhalten eine identische Signatur, davon sind $2^{(n-16)}-1$ fehlerhaft. Insgesamt gibt 2^n mögliche Eingaben der Länge n, davon sind 2^n-1 falsch. Die Wahrscheinlichkeit, eine unkorrekte Eingabe nicht zu erkennen, weil sie dieselbe Signatur wie das korrekte Ergebnis erzeugt, ist:

$$\frac{\text{nicht erkannte Fehler}}{\text{mögliche Fehler}} = \frac{2^{(n-16)}-1}{2^n-1} \sim 2^{-16} \text{ für } n >> 16$$

Problematisch bleibt die Fehlerlokalisation bei Fehlern in rückgekoppelter Hardware.

7.3.2.3 Built–in Logic Block Observer

Um neben der Testauswertung auch die Testmuster–Erzeugung zu beschleunigen, schlugen Könemann, Mucha und Zwiehoff [KMZ79] vor, die Register auch zur Erzeugung von Zufallszahlen zu benutzen und ihnen insgesamt 4 Betriebsmodi zu geben, z.B. :

1. Normal Mode,
2. Shift (Scan Path)Mode,
3. (Multi Input) Linear Feedback Shift Regsiter Mode (LFSR Mode)

4. Reset (Alternativ: LFSR Mode, ohne Eingang, Zufallszahlen–Generator)

Diese Form von Registern wird als **BILBO** (*built–in logic block observer*) bezeichnet. Eine mögliche Realisierung zeigt Abb. 7.11. Für diese gilt: $\forall i \neq 0 : D_i = (C1 \text{ AND } Z_i) \text{ XOR } (C2 \text{ NOR } Q_{i-1})$.

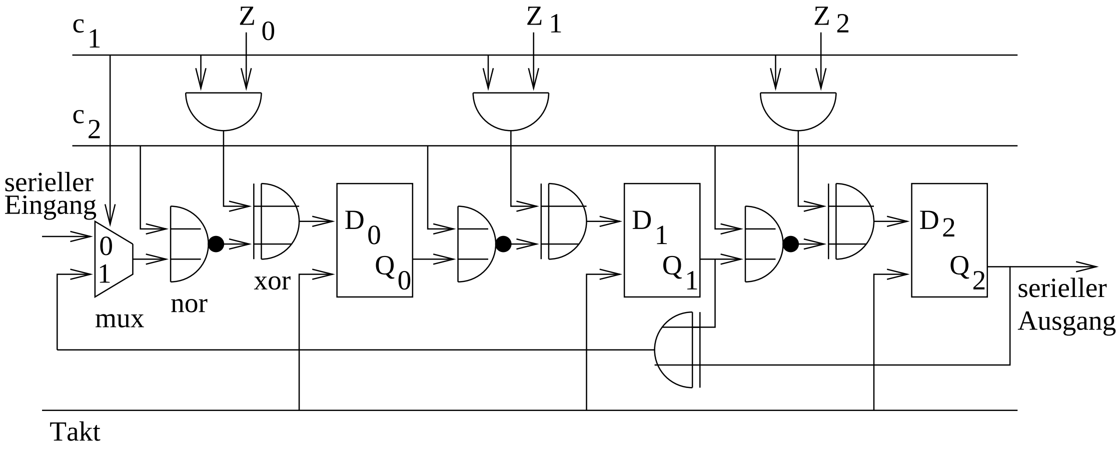


Abbildung 7.11: Bilbo

Daraus ergeben sich die speziellen Fälle in der Tabelle 7.1.

c_1	c_2		
'0'	'0'	$'0' \oplus \overline{Q_{i-1}} = \overline{Q_{i-1}}$	Scan Path Mode
'0'	'1'	$'0' \oplus \overline{'1'} = '0'$	Reset
'1'	'0'	$Z_i \oplus \overline{Q_{i-1}}$	Multi Input LFSR
'1'	'1'	$Z_i \oplus \overline{'1'} = Z_i$	Normaler Modus

Tabelle 7.1: Betriebsmodi der BILBO-Register

Angewandt werden BILBO–Register meist in Paaren, siehe Abb. 7.12.

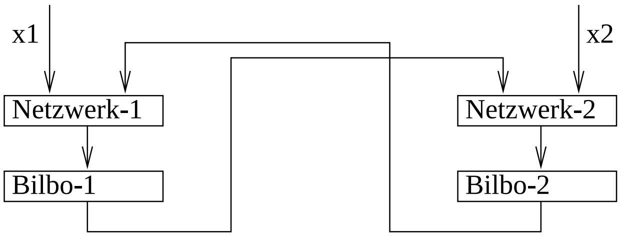


Abbildung 7.12: Anwendung von Bilbo-Registern

Zunächst wird in das Register Bilbo–1 der Anfangszustand für die Zufallszahlen–Generierung seriell eingeschoben. Dann wird dieses Register im LFSR–Modus bei abgeschalteten parallelen Eingängen als Zufallsgenerator benutzt. Gleichzeitig wird auch Register Bilbo–2 in den LFSR– Modus versetzt, um bei eingeschalteten parallelen Eingängen als CRC–Auffangregister zu dienen. Während der Erzeugung von Zufallszahlen muss der Eingang $x2$ mit den Werten beschaltet werden, die bei der Berechnung des Soll–CRC–Zeichens benutzt wurden. Nach Abaluf des Tests wir das berechnete CRC–Zeichen aus dem Register Bilbo–2 seriell ausgeschoben und mit dem Soll–CRC– Zeichen verglichen. Anschließend werden die Rollen der beiden Register vertauscht. Eine Variante dieser Technik wird häufig bei der Überprüfung des Inhaltes von ROM–Speichern eingesetzt. Als Mustergenerator dient dann ein als Zähler betriebendes Adressregister, als CRC– Auffangregister das Datenregister. In Mikrocomputer–Systemen kann das Datenregister dann parallel über dem Bus ausgelesen werden. Gerade der Inhalt von ROM–Speichern lässt sich mit anderen Methoden schlecht testen.

7.3.3 Testmustererzeugung für das Gatterniveau

Für alle aufgrund eines Fehlermodells möglichen Fehler möchte man Testmuster berechnen, aufgrund derer man die An– bzw. Abwesenheit der Fehler feststellen kann. Üblicherweise geht man dazu vom Gatter–Niveau aus und betrachtet als Fehlermodell das sog. Haftfehler–Modell. In diesem Modell nimmt man an, dass Leitungen im Fehlerfall entweder ständig logisch 0 (*stuck–at– zero*) oder ständig mit logisch 1 verbunden sind (*stuck–at–one*). Der große Vorteil dieses Modells liegt in der guten theoretischen Handhabbarkeit. Es gibt in der Hardware zwar eine Vielzahl anderer Fehler, diese lassen sich jedoch meist in äquivalente Haftfehler transformieren. Eine Ausnahme bildet die CMOS–Technologie, in der kombinatorische Schaltkreise im Fehlerfall in sequentielle Schaltkreise, d.h. endliche Automaten übergehen können. Das Fehlermodell ist in diesem Fall um *stuck–at–open*–Fehler zu erweitern [Lal85].

Vorübergehende Fehler werden von diesen Modellen nicht behandelt.

Es sei an dieser Stelle noch auf einen fundamentalen Unterschied zwischen dem Testen im Bereich der Software und dem Testen im Bereich der Hardware hingewiesen: Software ließe sich im Prinzip per Konstruktion oder per Verifikation korrekt herstellen. Von korrekten Programmen kann man beliebig viele korrekte Kopien herstellen. Hardware lässt sich jedoch nocht so einfach "kopieren". Jedes neue Exemplar eines Hardware–Systems muss getestet werden, da es sich von den bislang gefertigten Systemen immer noch unterscheiden kann. Dafür sind Hardware–Tests aber recht rigoros: Man verlangt in der Regel, dass 95% bis 99 % aller aufgrund eines Fehlermodells möglichen Fehler im Test erkannt werden können.

Die Testmuster–Erzeugung auf Gatter–Niveau sei zunächst anhand eines einführenden Beispiels erläutert. Zu erzeugen sei ein Test für den Fehler *stuck–at–one* (**s–a–1**) der Leitung f in der Abb. 7.13.

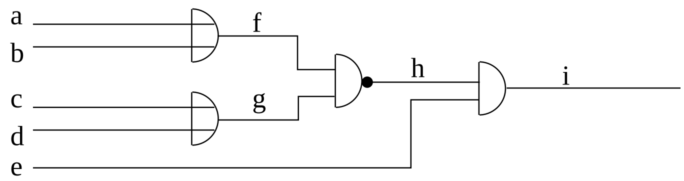


Abbildung 7.13: Beispielschaltung

Die Erzeugung eines Tests für s-a-1 bei f könnte wie folgt geschehen:

$a =' 0', b =' 0'$, damit $f =' 0'$ im fehlerfreien Fall ist;
$g =' 1'$, damit der Fehler fortgeschaltet werden kann,
$c =' 1'$, "–" "–",
$e =' 1'$, "–" "-".

Als Ergebnis erhält man $i =' 1'$ im fehlerfreien und $i =' 0'$ im fehlerhaften Fall. Um sich derartige Fallunterscheidungen zu sparen, führt man die Variable D ein, die wie folgt definiert ist:

$$D := \begin{cases} '1' & \text{im fehlerfreien Fall} \\ '0' & \text{im fehlerhaften Fall} \end{cases}$$

$$\overline{D} := \begin{cases} '0' & \text{im fehlerfreien Fall} \\ '1' & \text{im fehlerhaften Fall} \end{cases}$$

Da lediglich eine Variable definiert wird, impliziert dies, dass nur **Einzelfehler** betrachtet werden. Mehrfachfehler werden angeblich aufgrund praktischer Erfahrungen durch Tests auf Einzelfehler in der Regel ebenfalls gefunden.

Die Darstellung der Funktionen der Gatter geschieht im folgenden durch sog. *cubes*, die so definiert werden:

Def.: *primitive cubes* (pc's) sind **Zeilen** in der Darstellung einer Funktion f durch Primimplikanten für f und $\overline{f}$.

Beispiel: 2-fach NAND–Gatter `C <= A NAND B`:

fehlerfrei				fehlerhaft			s-a-1(A)			s-a-0(A)			s-a-1(B)			s-a-0(B)		
	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C
β_1	0	X	1		α_1		X	0	1	X	X	1	0	X	1	X	X	1
	X	0	1															
β_0	1	1	0		α_0		X	1	0	-	-	-	1	X	0	-	-	-

Tabelle 7.2: *primitive cubes*

Def.: *primitive D-cubes of a fault* (pdcf's) sind "Zeilen", die eine Minimalbedingung für das Erscheinen des Fehlers angeben.

Eine Eingangsbelegung erzeugt eine fehlerhafte Ausgabe $D(\overline{D})$, wenn sie in einem Primterm β_1 und α_0 (β_0 und α_1) enthalten ist.

Man definiert daher den Schnitt von Primtermen bzw. *cubes*:

$$X \cap' 0' =' 0',' X' \cap' 1' =' 1',' 1' \cap' 0' = \emptyset \text{ (leer)}\ ,\ \text{usw.}$$

Für das 2-fach NAND-Gatter gilt die Tabelle 7.3.

	s-a-1(A)			s-a-0(A)			s-a-1(B)			s-a-0(B)			s-a-1(C)+			s-a-0(C)++		
	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C
$\beta_0 \cap \alpha_1$		$\emptyset$		D	1	$\overline{D}$		$\emptyset$		1	D	$\overline{D}$	1	1	$\overline{D}$	0	X	D
$\beta_1 \cap \alpha_0$	0*	1	D		$\emptyset$		1	$\overline{D}$	D		$\emptyset$					X	0	D

Tabelle 7.3: *primitive D-cubes of a fault* (*: im fehlerhaften Fall 1 und sonst 0; +: $\beta_0/_{C=\overline{D}}$; ++: $\beta_1/_{C=D}$)

Def.: *propagation D-cubes of a logic element* sind "Zeilen", die eine Minimalbedingung zum Fortschalten des Fehlers vom Ein– zum Ausgang angeben.

Ein Fehler $D(\overline{D})$ an einem Eingang r erscheint als $f = D(\overline{D})$ am Ausgang wenn gilt: $r =' 0'$ impliziert $f =' 0'$ und $r =' 1'$ impliziert $f =' 1'$ (nichtinvertierend).

Ein Fehler $D(\overline{D})$ an einem Eingang r erscheint als $f = \overline{D}(D)$ am Ausgang wenn gilt: $r =' 0'$ impliziert $f =' 1'$ und $r =' 1'$ impliziert $f =' 0'$ (invertierend).

Zur Berechnung der *propagation cubes* betrachtet man daher den Schnitt von β_1 und β_0 unter Ignorieren des Eingangs r.

Für das 2–fach NAND–Gatter gilt die Tabelle 7.4.

	A	B	C		A	B	C		A	B	C		A	B	C
$\beta_{1/A=1}$	1	0	1	$\beta_{1/A=0}$	0	X	1	$\beta_{1/B=1}$	0	1	1	$\beta_{1/B=0}$	X	0	1
$\beta_{0/A=0}$		$\emptyset$		$\beta_{0/A=1}$	1	1	0	$\beta_{0/B=0}$		$\emptyset$		$\beta_{0/B=1}$	1	1	0
$\bigcap_\Gamma$		$\emptyset$			$\overline{D}$	1	D			$\emptyset$			1	D	$\overline{D}$
					D	1	$\overline{D}$						1	$\overline{D}$	D

Tabelle 7.4: Berechnung der *propagation cubes*

Bei Erscheinen des Fehlersignals am mehreren Eingängen (**rekonvergente Signale**) sind noch die propagation cubes ($DD\overline{D}$) und ($\overline{D}\overline{D}D$) nützlich. Das Abspeichern von zwei cubes mit zueinander negierten Fehlersignalen kann entfallen, wenn dies im folgenden Algorithmus als implizit gegeben betrachtet wird.

Der folgende Algorithmus findet stets ein gesuchtes Testmuster, sofern dieses existiert.

D-Agorithmus:

1. Wähle einen "D-cube" (pcdf) für den betrachtenden Fehler aus.
2. **Implikation**: Die durch den vorangegangenen Schritt implizierten Signale an den Eingängen und Ausgängen von Schaltgliedern werden rückwarts und vorwärts durch die Schaltung verfolgt. Die logischen Pegel auf den einzelnen Leitungen werden festgestellt, soweit dies ohne weitere Annahme über Signalpegel möglich ist. Dazu werden Schnittmengen zwischen dem "Test– cube" (dem jeweiligen Stand der bekannten Signale)und den "primitiv cubes" der zu durchlaufenden Schaltglieder gebildet. Wenn eine Inkonsistenz auftritt, d.h. wenn die Schnittmengen leer oder undefiniert sind, kehrt der Algorithmus zum letzten Schritt zurück, der eine alternative zu der getroffenen Auswahl bietet (backtracking).
3. *D–drive*: Die D-Grenze (*D-frontier*) besteht aus Menge aller Schhaltglieder, deren Ausgänge unspezifiziert sind und an deren Eingängen ein Fehlersignal D oder $\overline{D}$ auftritt. Ein Element aus dieser Menge wird ausgewählt und mittels Schnittbildung zwischen *test cube* und *propagation D cube* das Signal zum Ausgang des Elements weitergeleitet. Wenn kein nichtleerer Schnitt mit einem Schaltglied an der D–Grenze existiert, erfolgt wieder ein *backtracking*.
4. Für die beim D-drive implizierten Signale wird jetzt die Implikation Schritt (2) durchgeführt. Schritt (2) und (3) werden solange wiederholt, bis ein Fehlersignal einen primären Schaltungsausgang erreicht hat.
5. *line justification*: Bei den vorangegangenen Schritten unspezifiziert gebliebene Engänge von Schaltgliedern mit implizierten Ausgang werden nun durch Auswahl eines Schnittes zwischen dem *test cube* und einem *primitive cube* belegt. Erneut kann ein *backtracking* erforderlich sein.

a	b	c	d	e	f	g	h	i
X	1				1			
1	X				1			
0	0				0			
		X	1			1		
		1	X			1		
		0	0			0		
					0	X	1	
					X	0	1	
					1	1	0	
				0			X	0
				X			0	0
				1			1	1
					0	1	D	
0	0				0	1	D	
0	0			1	0	1	D	D
0	0	1	X	1	0	1	D	D

Tabelle 7.5: Testmusterberechnung

Für die Abb. 7.13 führt der Algorithmus zu der Testmusterberechnung nach Tabelle 7.5.

Trotz des umfangreichen Backtracking soll die Laufzeit des Verfahrens in der Praxis etwa mit dem Quadrat der Zahl der Gatter wachsen. Der D-Algorithmus erzeugt pro Fehler jeweils ein Testmuster. Industriell werden Weiterentwicklungen eingesetzt, die möglichst viele Fehler auf einmal testen.

7.3.4 Testverifizierung

Um die Güte von Hardware–Tests zu überprüfen, gehört zu jedem Test auch eine Berechnung des Anteils der erkennbaren Fehler. Dieser Anteil wird als **Fehlerüberdeckung** oder *fault coverage* bezeichnet:

Def.:

$$\text{Fehlerüberdeckung} := \frac{\text{Zahl der durch einen Test erkennbaren Fehler}}{\text{Zahl der laut Fehlermodell möglichen Fehler}}$$

Theoretische Berechnungen der Fehlerüberdeckung scheiden meist aus Aufwandsgründen aus. Eine Messung durch Probeläufe an Systemen mit künstlich eingebauten Fehlern ist u.a. wegen fehlender Prototypen oder (bei ICs) wegen fehlender Änderungmöglichkeiten nicht durchführbar. In der Praxis wird daher fast ausschließlich die Fehlersimulation durchgeführt. Bei diesem Ansatz wird die Reaktion eines fehlerfreien und eines fehlerhaften Modells der Hardware auf den Testmustersatz per Simulation miteinander verglichen.

Bei diesem intuitven Ansatz muss also für jeden möglichen Fehler der gesamte Testmustersatz simuliert werden. Rechenzeiten von mehreren Tagen sind daher durchaus gängig.

Simuliert man Gatter–Niveau, so kann man durch sog. parallele Fehlersimulation eine wesentliche Beschleunigung erreichen. Dabei nutzt man aus, dass bei üblichen Gattersimulationen nur ein Bit eines Maschinenwortes (oder eines Bytes) verwendet wird. Man nutzt nun alle Bits eines Maschinenwortes, indem man jedes Testmuster einem anderen Bit des Maschinenwortes zuordnet. Bei einer größeren Zahl von Testmustern kann man statt der Maschinenworte auch lange Bytestrings verwenden, wie sie z.B. in PASCAL durch das Mengenkonzept verfügbar sind. Operationen auf Bytestrings können beim Befehlssatz der IBM–Mainframes effizient durch Blockbefehle implementiert werden.

Bei der *deductive fault simulation* bzw. bei der *concurrent fault simulation* versucht man, in aufwendigen Verfahren mehrere Fehler für ein vorgegebenses Testmuster zu simulieren. Im Gebrauch sind ferner Hardwarebeschleuniger (*accelerators*).

7.3.5 Erzeugung von Selbsttest–Programmen

Eine Alternative zu teuren Testautomaten und zur eingebauten Testhardware besteht bei programmierbaren digitalen Systemen in der Verwendung von Selbsttestprogrammen, d.h. von "diagnostics", die auf dem zu testenden System selbst ausgeführt werden. Der Vorteil liegt bei diesem Ansatz in der weitgehenden Einsparung von Testhardware. Als Nachteil galt bislang der Aufwand bei der Erstellung von Selbsttest–Programmen. Ziel u.a. von Arbeiten am Lehrstuhl 12 war es, diesem Nachteil durch Automatisierung der Programmerzeugung auszuräumen. Es wurde das Programm RESTART entwickelt, das zu einer vorliegenden Hardware–Strukturbeschreibung automatisch Testprogramme generiert.

Voraussetzung für RESTART sind:

1. die Speicherprogrammierbarkeit,
2. das Vorhandensein entweder der Operationen ”=, ”<>”, ”XOR” oder von ”+”, ”-” in Kombination mit dem Test auf Null (”=0”)
3. die Möglichkeit, abhängig vom Vergleichsergebnis bedingt zu springen.

RESTART generiert binären Programmcode für Befehle der Art

```
PC:= (IF Sollwert = IstWert THEN PC+1 PC ELSE ErrorExit;
```

Beispielsweise würde die Speicherzelle `SR[5]` getestet werden durch die Sequenz:

```
SR[5]:=5;
PC:=(IF SR[5]=5 THEN (PC+1) ELSE ErrorExit;
```

Es wird angenommen, dass ein Sprung an das Label `ErrorExit` von außen erkennbar ist.

Die Codererzeugung für diese Testbefehle basiert auf einer Analyse der Hardware–Module und ihrer Verbindungen untereinander. Eine detailliertere Beschreibung des Einsatzes von RESTART in einer Umgebung, in der auch die Fehlerüberdeckung berechnet werden kann, liefert der folgende englischsprachige Text[2].

7.3.5.1 Introduction

Test generation methods have traditionally focussed very much on gate level descriptions. Advantages of this approach include the availability of well-defined fault models and a good fault coverage. Disadvantages include the growing complexity of gate-level descriptions, resulting in large tool execution times and the lack of applicability of these techniques during early design phases.

As another extreme case, test generation methods based on purely behavioural circuit descriptions (for example, on instruction set descriptions) have a rather weak relation to possible physical defects.

Consequently, in our earlier work we have focussed on test generation methods based on circuit descriptions at an intermediate level. More precisely, we have chosen the RT structural level since physical defects can easily be represented at this level. In particular, we have focussed on exploiting programmability of processors such as DSP processors [Lee88], core processors [Adv95, LSI96], and ASIPs [Wou94, HD95]. Such processors are amenable to self-testing by running self-test programs on the processors themselves. It has been shown, that such self-test programs can be generated automatically from test specifications for each of their components [Krü91, BM95].

So far, there has been neither a full integration of self-test program generation into standard test environments nor has there been any proof of the good fault coverage expected for this approach. Introducing such a full integration and demonstrating the achievable fault coverage are the main goals of this paper. We will show that self-test programs provide a good fault coverage not only for the data path (which was expected) but also for the controller. For the testing the ALU of processor having just a single ALU, the same ALU has to perform also the result comparison. This could potentially lead to fault masking effects. Our experimental data shows that fault masking can be avoided, even for the simple processor which we will consider.

Being able to generate full self-test programs performing all the essential operations is the main strength of RESTART [BM95] and its predecessor MSST [Krü91]. RESTART (retargetable compilation of self-test programs using constraint logic programming) is the tool used for this paper.

7.3.5.2 Structure of STAR-DUST test generation process

The work described in this paper is directed at using RESTART for the automatic generation of a comprehensive processor test program based on realistic fault models. This requires the availability of both RT-level and gate-level models for the same processor. These models are used by the tools implementing the essential tasks of our current approach:

1. synthesis of gate-level descriptions for each of the RT-level components,
2. test pattern generation for each of the RT-level components from their gate-level descriptions,

[2]Dieser Text ist dem Bericht Ulrich Bieker, Martin Kaibel, Peter Marwedel, Walter Geisselhardt: STAR-DUST: Hierarchical Test of Embedded Processors by Self-Test Programs, *Interner Bericht, FB Informatik, Universität Dortmund, 1998* entnommen.

3. generation of machine instructions implementing justification and response propagation for each of the test patterns,
4. fault simulation computing the fault coverage achieved by running self-test programs.

The tools and data formats currently used for these four tasks are shown in fig. 7.14.

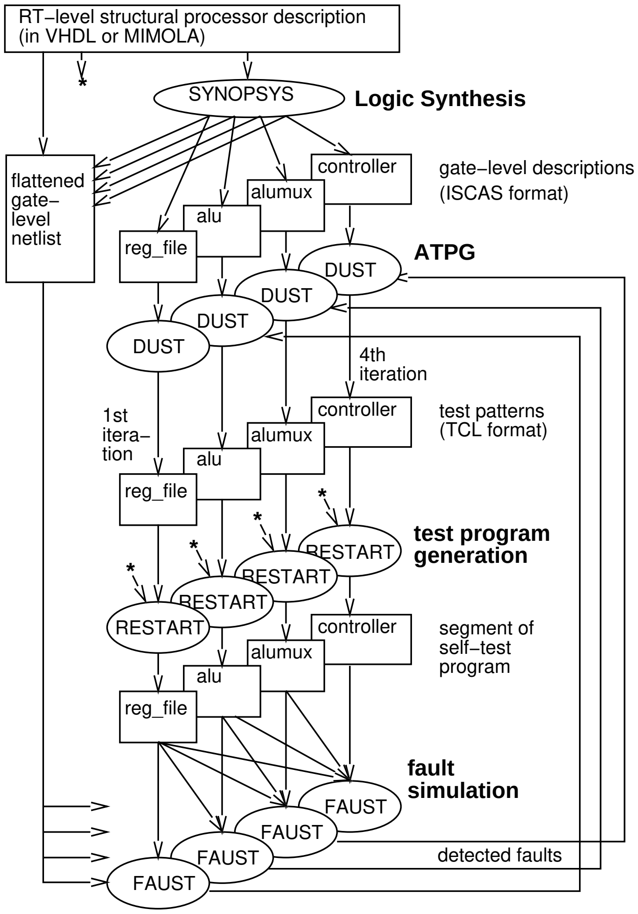


Abbildung 7.14: STAR-DUST test generation process

We start with an RT-level structural description of the processor under test. Input formats currently available for this purpose include VHDL [IEE92] or MIMOLA [BBH+94].

From this specification, we have to obtain equivalent gate level descriptions. For this purpose, we run the SYNOPSYS logic synthesis tools [Syn95] for each of the RT-level components. In fig. 7.14 it is assumed that our processor consists of just four components: register file `reg_file`, an ALU, an ALU multiplexer, and a controller.

Gate-level descriptions for each of the components are required for the tool implementing task 2. An overall flattened netlist for the entire processor is required for fault simulation (task 4). Gate-level descriptions generated by SYNOPSYS tools have to be translated into the ISCAS benchmark format required for ATPG and fault simulation tools.

Next, we select a heuristic order for processing RT-level components. (in figure 7.14, components drawn in front are processed first). Respecting that order, we do the following for each RT-level component:

- We use the DUisburg Sequential Test generator DUST [GK91a] for obtaining test pattern sets for the current component (e.g. the register file). DUST is an enhanced implementation of the BACK algorithm [Che88].

  Concerning a certain RT component, there may be some restrictions on the input vectors X(t) and state vectors Z(t) of the current component which can be justified. These restrictions may be imposed e.g., by encoding restrictions or the connections of the processor in which the RT component is embedded. For all but the first iteration of the loop, faults covered by already generated test program segments are also taken into account. If all faults are covered by such segments, the component is effectively skipped.

  Test pattern produced by DUST have to be translated into the test code language (TCL) accepted by the next tool. This translation is straightforward.

- Self-test programs are synthesized using RESTART. RESTART produces programs justifying test patterns and evaluating test responses. Evaluation of test responses is done by conditional jumps.

  Example: Consider a test pattern for an ALU. Suppose control code "10" activates the add operation of the alu, and assume that binary values "0111" and "0001" have been selected as test patterns by DUST. Then, a test for the add operation is specified in TCL by the following statement:

  TEST alu(0111,0001,10);

  The result should be 1000. RESTART generates code, which activates (i.e. justifies) the + operation and checks the result by a conditional jump:

  ```
  IF 0111 + 0001 = 1000
  THEN increment program counter
  ELSE jump to error label;
  ```

  Every TEST statement is compiled to a conditional jump. If no error occurs, the program continues with the execution of the next instruction of the self-test program, otherwise a jump to an error procedure is performed. TCL allows the specification of all kinds of tests including memory test loops.

  The following requirements must be met by processors to be tested with code generated by RESTART:

  1. The processor to be tested must be able to perform a comparison operation and to perform a conditional jump.
  2. The processor must be programmable.
  3. The program counter must be observable (on the other hand, a scan path is not necessary).
  4. A single instruction cycle is required.

  Details on the code generation technique in RESTART can be found in recent papers and books [BM95, Bie95b, Bie95a].

- Programs generated by RESTART are then used as initial stimuli for fault simulation at the gate-level. This way, the coverage of the programs produced by RESTART can be computed. Fault simulation is based on the gate-level stuck-at fault model.

  In our first approach we use FAUST (FAult Simulation Tool), a very efficient single pattern, single fault propagation method. FAUST has been extended to handle a ROM, assuming faults only on the data and address line of the ROM. The instruction memory itself is assumed to be fault free. To speed up the fault simulation process we will replace FAUST by PARIS (PARallel Iterative Simulator), a parallel pattern single fault propagation simulator [GM93, GK91b].

  Information about covered faults is exploited in the next cycle of the loop in order to reduce the size of the required test pattern set.

The loop terminates if all RT-level components have been considered.

RESTART, DUST and FAUST are the essential ingredients of this test generation process. Hence, we call this process STAR-DUST.

7.3.5.3 An Example

In order to demonstrate the test generation flow, let us use the CPU `SIMPLECPU` depicted in fig. 7.15. `SIMPLECPU` is a small programmable microprocessor consisting of eight modules. The `SIMPLECPU` controller (shaded area) consists of a program counter, an instruction memory, an incrementer, and a multiplexer.

A 16 x 4 register file, a 4-bit ALU and a second multiplexer make up the datapath. The register file and the program counter are synchronized by a clock. Control signals are denoted by "c" followed by an index range. The 8-bit program counter addresses the 256 x 22 bit instruction memory. The ALU computes a condition output that enables the controller multiplexer to perform conditional jumps.

The gate-level description of `SIMPLECPU` can be synthesized. The resulting circuit consists of 467 gates and 72 D-flip-flops.

Let us now explain the experimental procedure for generating a test program for `SIMPLECPU`. We start by developing the self-test program for testing the register file (1st iteration) and catenate self-test program segments for the other RT-components.

1st iteration: We assume D-flip-flops have neither a set nor a reset input. Therefore, we can not detect faults which prevent the initialization of the D-flip-flops. DUST marks such faults as **untestable**. In total, `SIMPLECPU` has 6 untestable faults.

Using test frames [GK91a], we prune the search space of the test generator. Faults which can not be detected because of a test frame are called **functionally untestable**. We found 4 faults in the register file and 2 faults in the controller to be functionally untestable.

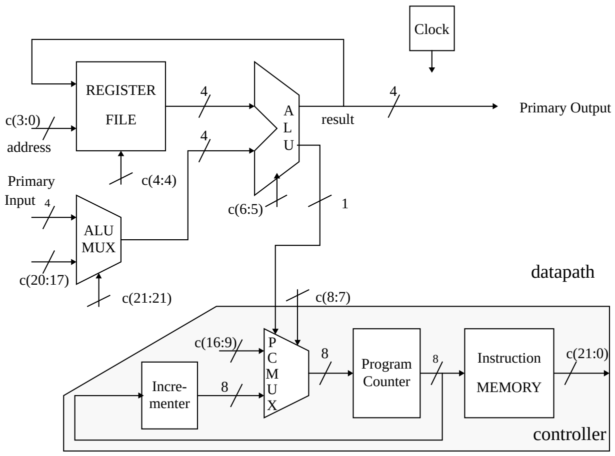


Abbildung 7.15: SIMPLECPU

As example for a test frame we consider the register file, which is embedded in the complete circuit. If a certain register cell should be tested, the signal `wr_enable` must be '0' (read mode) and the ALU must perform a comparison operation in order to compare the output value of the register file with the expected value. The comparison is done by a subtract operation and a comparison with zero, i.e. (`outdata` - expected value) = '0'. Therefore, in the non-faulty case the output of the ALU which feeds the data input of the register file is restricted to the binary value 0000. To summarize, in the context of self-test programs generated by RESTART, `wr_enable` = '0' implies `indata` = 0000. This restriction concerning test patterns of the register file is specified as test frame.

2nd iteration: By applying the abovementioned 193 test patterns to the register file we achieve a fault coverage of 49.08% for `alu`. To detect the remaining faults of `alu`, DUST generates 17 test patterns. The resulting self-test code consists of 51 instructions, i.e., 3 instructions are necessary to apply an ALU test pattern and to check the test response. We illustrate such a test cycle by an example.

One of the 17 generated test patterns activates the + operation of the ALU with data "0000" and "1111". The according TCL statement is:

```
TEST alu(0000,1111,10);
```

The expected test response of the ALU output signal `result` is "1111". Table 7.6 shows the resulting code for above TCL statement.

21	20:17	16:9	8:7	6:5	4	3:0	Comment
1	0000	X···X	00	01	1	0000	REG[0] := 0000
0	XXXX	X···X	00	10	1	0000	REG[0] := REG[0] + 1111
0	XXXX	11111111	10	11	0	0000	IF (REG[0] - 1111) = 0 THEN PC := PC+1 ELSE PC := 255

Tabelle 7.6: Code for an `alu` test cycle

Additionally, we expect the test pattern "1111" at every instruction cycle at the primary input. The first instruction loads register 0 with the binary value "0000" which has to be applied to the first input port of the ALU. Value "0000" directly originates from the instruction memory (signal `c`(21:17)). The second instruction applies the generated test pattern and activates the `+` operation of the ALU ("1111" originates from the primary input). Instruction 3 finally checks the result of the ALU by activating the comparison operation of the ALU and performing a conditional jump. In case of a fault, the program jumps to the user defined error addres 255 (the program counter must be observable). Otherwise the self-test program continues with the execution of the next test statement. A fault in the comparison unit of the ALU is detected by different conditional jumps and therefore can be observed at the program counter.

An additional reset pin exists to initialize the program counter with 0. Therefore, the first instruction of the generated self-test program is located at address 0.

3rd iteration: Test generation for `alumux` is effectively skipped, because the test program segments generated for `reg_file` and `alu` already provide a 100 % fault coverage for this component.

4th iteration: Testing the controller is mainly implicitly done by the self-test program developed so far. Adding 6 (conditional and unconditional) jump instructions results in a fault coverage of 98.36% for the controller. Currently, these 6 TCL statements have to be added manually, but automation of this step can be added to RESTART.

7.3.5.4 Results

7.3.5.4.1 Results for SIMPLECPU The entire self-test program for `SIMPLECPU` consists of 250 instructions and the achieved fault coverage for `SIMPLECPU` (including data path and controller) is 99.63%.

Table 7.7 gives information and results concerning the example processor `SIMPLECPU`.

RT-component	gates	DFF's	flt's	det	untest	f_untest	aborted	fault coverage	efficiency
`reg_file`	316	64	2256	2252	0	4	0	99.82%	100.00%
`alu`	70	0	436	436	0	0	0	100.00%	100.00%
`alumux`	13	0	76	76	0	0	0	100.00%	100.00%
`controller`	68	8	488	480	6	2	0	98.36%	100.00%
`SIMPLECPU`	467	72	3256	3244	6	6	0	99.63%	100.00%

Tabelle 7.7: `SIMPLECPU` results

For every RT component and the entire processor, table 7.7 shows the number of gates, the number of D-flip-flops, the number of stuck-at faults, the number of detected faults, the number of untestable faults, the number of functional untestable faults, the number of aborted faults, the fault coverage for the stuck-at fault model, and the efficiency.

To illustrate the abovementioned iterations consider table 7.8. It shows the number of generated test patterns and the resulting fault coverage for the different components iteration by iteration. Initially, we generated 193 test patterns for the register file. The resulting test code for these 193 patterns already detects e.g., 81.35% of the controller faults. The final self-test program consists of $193 + 3 * 17 + 6 = 250$ microinstructions. With the help of the 6 jump instructions of step 4, we detected 82 remaining faults of the controller.

	additional patterns	`reg_file`	`alu`	`alumux`	`controller`
1st iteration	193	$\boxed{99{,}82\ \%}$	49,08 %	100 %	81,35 %
2nd iteration	17	99,82 %	$\boxed{100\ \%}$	100 %	81,55 %
3rd iteration	0 (skipped)	99,82 %	100 %	$\boxed{100\ \%}$	81,55 %
4th iteration	6 branches	99,82 %	100 %	100 %	$\boxed{98{,}36\ \%}$

Tabelle 7.8: Course of development of self-test program and fault coverage

The results show, that the self-test program approach can achieve high fault coverages for the data path as well as for the controller. Interestingly enough, fault masking can be avoided even for this simple processor.

	DUST	RESTART	FAUST
CPU seconds	4.12	93.56	5.77

Tabelle 7.9: CPU times measured on a Sparc 20

Table 7.9 shows the CPU times for the different tools for SIMPLECPU. For DUST the sum of CPU times required to generate test patterns for all RT components is given. The hierachical test approach divides the complex test problem for the entire processor in several less complex subproblems. Therefore, we are able to generate a test within minutes.

For this example, RESTART is the tool consuming most the computing time. To some extend, this is caused by focussing on the functionality of RESTART, rather than on its speed. RESTART is implemented in a non-standard programming paradigm, called *constraint logic programming*, which might have caused some runtime penalty.

7.3.5.4.2 Results for Tanenbaum Example As a slightly more complex example, we have studied an architecture described by Tanenbaum [Tan90]. For this example, the gate-level description contains 1936 gates and 296 D-type flip-flops. There are 6803 possible stuck-at faults, of which 38 are untestable. Using the STAR-DUST procedure, a fault coverage of 95.88 % is obtained, corresponding to an efficiency of 96.44 %. The resulting self-test program requires 246 instructions. The architecture allows up to 255 instructions. A higher fault coverage could have been achieved without the limit on the number of instructions.

For this example, automatic code generation for loops was not possible due to the limited support of loops in the particular architecture. Hence, a loop for testing the register file was manually specified leading to a total of 54 static instructions. All other instructions were synthesized from automatically generated TCL statements.

7.3.5.5 Summary: Characteristics of the STAR-DUST approach

STAR-DUST meets a large number of different objectives:

- STAR-DUST is the first test generation process computing the fault coverage which can be achieved by self-test programs.

  Computed coverage is implicitly based on the assumption that the synthesized gate-level description is the one actually implemented.
- STAR-DUST is a hierarchical test generation process reducing the complexity of generating test patterns for the entire processor to that of generating test patterns for each of the components.

  Test generation proceeds at a high level of abstraction (using RESTART) while at the same time preserving the high fault coverage through gate level fault modelling.
- STAR-DUST reduces the length of the test program by considering faults covered by tests generated for one component during subsequent generation of test patterns.
- Errors during logic synthesis usually result in error reports during fault simulation. Hence, fault simulation is effectively used for validating the consistency of RT-level and gate-level models. STAR-DUST can be employed for model validation! In particular, logic synthesis results can be validated this way.
- Processor testability can be improved with the help of additional redesign cycles.

7.4 Formale Verifikation

Für diesen Abschnitt sei auf den Gastvortrag von Tiziana Margaria-Steffen verwiesen.

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.

[Adv95] Advanced RISC Machines Ltd. ARM. web pages. *http://www.arm.com/*, 1995.

[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.

[AM95] G. Araujo and S. Malik. Optimal code generation for embedded memory no-homogenous register architectures. *8th Int. Symp. on System Synthesis (ISSS)*, pages 36–41, 1995.

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

[ANH+93] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi. An ASIP instruction set optimization algorithm with functional module sharing constraint. *Int. Conf. on Computer-Aided Design (ICCAD)*, pages 526–532, 1993.

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

[Ass94] Semiconductor Industry Association. The national technology roadmap for semiconductors. *www*, 1994.

[Ban85] R.E. Bank. Transient simulation of silicon devices and circuits. *IEEE Trans. on CAD, Vol. CAD-4*, pages 436–451, 1985.

[Bar92] D.H. Bartley. Optimizing stack frame accesses with restricted addressing modes. *Software - Practice and Experience*, 22:101–110, 1992.

[BBH+94] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA language 4.1. *Universität Dortmund, Informatik 12, http://ls12-www.informatik.uni-dortmund.de*, 1994.

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

[Bie95a] U. Bieker. Retargetable compilation of self-test programs using constraint logic programming. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer Academic Publishers*, 1995.

[Bie95b] U. Bieker. *Retargierbare Compilierung von Selbsttestprogrammen digitaler Prozessoren mittels Constraint-logischer Programmierung (in German)*. Shaker Verlag, Aachen, ISBN 3-8265-10925, 1995.

[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.

[BLR95] J.-M. Bergé, O. Levia, and J. Rouillard. *High-Level System Modeling*. Kluwer Academic Publishers, 1995.

[BM95] U. Bieker and P. Marwedel. Retargetable self-test program generation using constraint logic programming. *32nd Design Automation Conference*, 1995.

[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.

[Bro97] M. Broy. Schwerpunktthema: Formale Methoden in der Praxis. *Sonderheft der Zeitschrift it+ti*, (3), 1997.

[Bry86] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, pages 677–691, 1986.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[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.

[Che88] W. Cheng. The back algorithm for sequential test generation. *Proceedings International Conference on Computer Design*, pages 66–69, 1988.

[Coe89] D. R. Coelho. The VHDL handbook. *Kluwer Academic Publishers*, 1989.

[CW96] R. Camposano and W. Wolf. Message from the editors-in-chief. *Design Automation for Embedded Systems*, 1996.

[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.

[DH89] D. Drusinsky and D. Harel. Using statecharts for hardware description and synthesis. *IEEE Trans. on Computer Design*, 1989.

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

[Dir87] H. K. Dirks. The impact of computers on semiconductor device modeling. *Int. Conf. VLSI and Computers*, pages 327–332, 1987.

[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.

[EED] EEDesign. Tables of hard cores and soft cores. *http: //www.eedesign/Resources.html*.

[Eng85] W.L. Engl. Process and device modeling. *Advances in CAD for VLSI, Vol. 1, North Holland*, 1985.

[FHP92] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems, volume 1*, pages 213–226, 1992.

[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.

[FS98] M. Fowler and K. Scott. *UML Distilled - Applying the Standard Object Modeling Language*. Addison-Wesley, 1998.

[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.

[GJ75] M.R. Garey and D.S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM J. Comput.*, pages 397–411, 1975.

[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.

[GK91a] N. Gouders and R. Kaibel. Advanced techniques for sequential test generation. *Proc. ETC*, pages 293–300, 1991.

[GK91b] N. Gouders and R. Kaibel. PARIS: A parallel pattern fault simulator for synchronous sequential circuits. *Proc. ICCD*, pages 542–545, 1991.

[GM93] W. Geisselhardt and M. Mojtahedi. New methods for parallel pattern fast fault simulation for synchronous sequential circuits. *ICCAD*, 1993.

[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.

[Gro97] Rational Software Group. Uml summary. *www.rational.com*, 1997.

[GVNG94] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and design of embedded systems.* Prentice Hall, 1994.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.

[Hay82] J.P. Hayes. A unified switching theory with applications to VLSI design. *Proceedings of the IEEE, Vol.70*, pages 1140–1151, 1982.

[Hay86] J.P. Hayes. Digital system simulation with multiple logic values. *IEEE Trans. on CAD, Vol. 5*, pages 274–283, 1986.

[HD94] I.-J. Huang and A. Despain. Generating instruction sets and microarchitectures from applications. *Int. Conf. on CAD (ICCAD)*, pages 391–396, 1994.

[HD95] I.-J. Huang and A. Despain. Synthesis of application specific instruction sets. *IEEE Trans. on CAD*, pages 663–675, 1995.

[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.

[HLD98] publication chairman HLDVT. High-level design verifikation and test workshop. *web-page: http://www.ee.ucsd.edu/hldvt???*, 1998.

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Software Engineering Methods*, 1996.

[Hog89] D. Hogrefe. *Estelle, LOTOS und SDL.* Springer-Verlag, 1989.

[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.

[Hua96] I.-J. Huang. Synthesis and exploration of instruction set design for application specific symbolic computing. *2nd Workshop on Code Generation for Embedded Processors (unpublished)*, 1996.

[IEE88] Design Automation Standards Subcommittee of the IEEE. IEEE standard VHDL language reference manual (IEEE Std. 1076-87). *IEEE Inc., New York*, 1988.

[IEE92] Design Automation Standards Subcommittee of the IEEE. Draft standard VHDL language reference manual. *IEEE Standards Department, 1992*, 1992.

[IEE93] IEEE. Sonderheft mcms. *IEEE Design & Test, Dec.*, 1993.

[iM96] Shin ichi Minato. *Binary Decision Diagrams and Applications for VLSI CAD.* Kluwer Academic Publishers, 1996.

[IME86] IMEC. Jahresbericht der IMEC. *Interuniversitair Micro-Elektronica Centrum, Leuven*, 1986.

[KB95] C. Delgado Kloos and P. T. Breuer. *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995.

[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.

[KMB81] L. Kou, G. Markowsky, and L. Berman. A fast algorithms for Steiner trees. *Acta Informatica*, pages 141–145, 1981.

[KMZ79] B. Könemann, J. Mucha, and G. Zwiehoff. Built-in logic block observer. *Proc. IEEE Intern. Test Conf.*, pages 261–266, 1979.

[KN96] D.J. Kolson and A. Nicolau. Optimal register assignment to loops for embedded code generation. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 1996.

[Koh87] Z. Kohavi. Switching and finite automata theory. *Tata McGraw-Hill Publishing Company, New Delhi, 9th reprint*, 1987.

[Kop97] H. Kopetz97. *Real-Time Systems –Design Principles for Distributed Embedded Applications–*. Kluwer, 1997.

[Krü91] G. Krüger. A tool for hierarchical test generation. *IEEE Trans. on CAD, Vol. 10*, pages 519–524, 1991.

[Lal85] P.K. Lala. *Fault tolerant and Fault Testable Hardware Design*. Prentice Hall, 1985.

[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.

[LDK+95] S. Liao, S. Devadas, K. Keutzer, S. Tijang, and A. Wang. Storage assignment to decrease code size. *Programming Language Design and Implementation (PLDI)*, 1995.

[LDKT95] S. Liao, S. Devadas, K. Keutzer, and S. Tijang. Code optimization techniques for embedded DSP microprocessors. *32nd Design Automation Conference*, pages 599–604, 1995.

[Lee61] C.Y. Lee. An algorithm for path connections and its application. *IRE Trans. on Electronic Computers, Vol.EC–10*, pages 346–365, 1961.

[Lee88] E. Lee. Programmable DSP architectures, parts i and ii. *IEEE ASSP Magazine*, Oct. 1988 & Jan. 1989, 1988.

[Len90] T. Lengauer. Combinatorial algorithms for integrated circuit layout. *Wiley-Teubner*, 1990.

[Leu96] R. Leupers. Algorithms for address assigment in DSP code generation. *ICCAD*, 1996.

[Leu97a] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997.

[Leu97b] R. Leupers. Retargetable generator of code selectors from HDL processor models. *European Design and Test Conference (ED &TC)*, 1997.

[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.

[LM95] R. Leupers and P. Marwedel. Time-constrained code compaction for DSPs. *Int. Symp. on System Synthesis (ISSS)*, 1995.

[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.

[LMD94] B. Landwehr, P. Marwedel, and R. Dömer. OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming. *Proceedings of the EURO-DAC*, pages 90–95, 1994.

[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.

[LNV+97] C. Liem, F. Nacabal, C. Valderrama, P. Paulin, and A. Jerraya. Co-simulation and software compilation methodologies for the system-on-a-chip in multimedia. 1997.

[LR83] K.-D. Lewke and F.J. Rammig. Description and simulation of MOS in register transfer languages. *in: F. Anceau, E.J. Aas(Hrsg.): VLSI'83, North Holland*, 1983.

[LSI96] LSI Logic Inc. web pages. *http://www.lsil.com/products/unit5_5.html*, 1996.

[Mar86] P. Marwedel. A new synthesis algorithm for the MIMOLA software system. *23rd Design Automation Conf.*, pages 271–277, 1986.

[Mar93] P. Marwedel. *Synthese und Simulation von VLSI-Systemen*. Hanser, 1993.

[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.

[MG95] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[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.

[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.

[MT97] C. Meinel and T. Theobald. Geordnete binäre entscheidungsgraphen und ihre bedeutung im rechnergestützten entwurf hochintegrierter schaltkreise. *Informatik-Spektrum*, pages 268–275, 1997.

[Nie97] Ralf Niemann. Harware/software-codesign mit cool. *Vortragsfolien, LS Informatik 12*, 1997.

[NM96] R. Niemann and P. Marwedel. Hardware/software partitioning using integer programming. *European Design & Test Conference*, 1996.

[N.N] N.N. *The Advanced learner's dictionary of contemporary English*. Oxford University Press.

[NND95] S. Novack, A. Nicolau, and N. Dutt. A unified code generation approach using mutation scheduling. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer Academic Publishers*, 1995.

[Oes97] B. Oesterreich. *Objektorientierte Softwareentwicklung mit der Unified Modeling Language*. Oldenbourg, 1997.

[Oht86] T. Ohtsuki. Layout design and verification. *Advances in CAD for VLSI, Vol. 4, North–Holland*, 1986.

[Pag98] J. Pagni. Wie man gebäude wachsam und sparsam macht -integrierte beleuchtung, heizung und klimatisierung-. *in: Zentrum für Technologische Entwicklung Tekes, Im Blickpunkt: Finnische Technologie*, 1998.

[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.

[PLMS95] P. Paulin, C. Liem, T. May, and S. Sutarwala. DSP design tool requirements for embedded systems: A telecommunications industrial perspective. *Journal of VLSI Signal Processing*, pages 23–47, 1995.

[Pos89] H.-U. Post. *Entwurf und Technologie hochintegrierter Schaltungen*. Teubner, 1989.

[Pus90] D. Pusch. Optimierung von PLAs für Mikroprogrammsteuerwerke. *Diplomarbeit, Inst. f. Informatik &P.M., Univ. Kiel*, 1990.

[Qui] Fa. Quickturn. http://www.quickturn.com. *(web pages)*.

[Ram98] F.J. Rammig. Schwerpunktthema: Systemspezifikation - Methoden, Werkzeuge und Anwendungen. *Sonderheft der Zeitschrift it+ti*, (3), 1998.

[RC89] W. Rosenstiel and R. Camposano. *Rechnergestützter Entwurf hochintegrierter Schaltungen*. Springer-Verlag, 1989.

[Rei86] W. Reisig. *Petrinetze - Eine Einführung*. Springer-Verlag, 1986.

[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.

[Rue86] A.E. Ruehli. Circuit analysis, simulation and design. *Advances in CAD for VLSI, Vol. 3, North Holland*, 1986.

[Rya95] M. Ryan. Market focus – insight into markets that are making the news in EE Times. *http://techweb.cmp.com/techweb/eet/embedded/embedded.html (Sept. 11)*, 1995.

[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.

[SLD97] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. *Design Automation Conference*, 1997.

[SM91] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *acm computing surveys, Vol. 23*, pages 143–221, 1991.

[SM95] A. Sudarsanam and S. Malik. Memory bank and register allocation in software synthesis for ASIPs. *Intern. Conf. on Computer-Aided Design (ICCAD)*, pages 388–392, 1995.

[Sou78] J. Soukup. A fast maze router. *15th Design Automation Conf.*, pages 100–101, 1978.

[Spe93] Special Interest Group on Synthesis from VHDL. VHDL arithmetic package for synthesis. *repository at INTERNET host "vhdl.org", login "anonymous", file "vi/vhdlsynth/numeric_bit.vhd"*, 1993.

[Sta94] W. Stallings. *Data and Computer Communications.* MacMillan, 1994.

[Syn95] Inc Synopsys. Synopsys version v3.3a. *700 East Middlefield Road. Mountain View, CA 94043-4033, USA*, 1995.

[Tan90] A.S. Tanenbaum. Structured computer organization. *Prentice Hall, (3rd edition)*, pages 161–186, 1990.

[Tei97] J. Teich. *Digitale Hardware/Software-Systeme.* Springer, 1997.

[Tim95] E. Timmer. Conflict modelling and instruction scheduling in code generation for in-house DSP cores. *32th Design Automation Conference*, 1995.

[Ull84] J.D. Ullman. Computational aspects of VLSI. *Computer Science Press*, 1984.

[VGG94] F. Vahid, J. Gong, and D. Gajski. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. *European Design Automation Conference (EURO-DAC)*, pages 214–219, 1994.

[vLA87a] P.J.M. van Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications.* D. Riedel, Dortrecht, 1987.

[Vla87b] A. Vladimirescu. SPICE user's guide. *Northwest Laboratory for Integrated Systems, Seattle*, 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.

[WE85] N. Weste and K. Eshraghian. Principles of CMOS VLSI design, a systems perspective. *Addison Wesley, Reading*, 1985.

[Weg] Ingo Wegener. *Theorie des Logikentwurfes, Skript zur Vorlesung.* Fachbereich Informatik der Universität Dortmund.

[Wes95] B. Wess. Code generation based on Trellis diagrams. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer Academic Publishers*, 1995.

[Wou94] R. Woudsma. EPICS, a flexible approach to embedded DSP cores. *Int. Conf. on Signal Processing and Applications and Technology*, 1994.

[YH97] H. Yalcin and J. P. Hayes. Event propagation conditions in circuit delay computation. *ACM Trans. on Design Automation of Electronic Systems*, 2:249–280, 1997.

[Ze94] V. Zivojnovic and et al. DSPstone: A DSP-oriented benchmarking methodology. *Proc. of the Intern. Conf. on Signal Processing and Technology*, 1994.

[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