

Einleitung

In diesem einleitenden Kapitel soll der automatisierte Entwurf von digitalen Hardware/Software-Systemen motiviert werden. Dabei werden die wesentlichen Synthesaufgaben vorgestellt.

1.1 Motivation

Betrachtet wird der automatisierte Entwurf (engl. *design automation*) und die Optimierung komplexer digitaler Systeme, die aus Hardware- und Softwarekomponenten bestehen, sog. *Hardware/Software-Systeme*. Obwohl solche Systeme bereits seit vielen Jahren von Ingenieuren und Technikern konzipiert und gebaut werden, ist man sich heutzutage darüber einig, dass man nur durch den Einsatz rechnergestützter Entwurfsmethoden (engl. *computer-aided design* (CAD)) die Komplexität heutiger Systeme bewältigen und bessere Entwürfe (Performanz, Kosten) in kürzerer Zeit ermöglichen kann. Dies erklärt das wachsende Interesse der Industrie und der Forschung an rechnergestützten Entwurfsmethoden.

Die Komplexität, so wie sie hier verstanden wird, entsteht *nicht nur* durch die Anzahl der Einzelkomponenten, aus denen ein System zusammengesetzt ist, sondern vor allem durch *Heterogenität*. In Zukunft liegen die Anforderungen gerade bei der Beherrschung heterogener technischer Systeme, die sich durch verschiedenartige Komponenten und Interaktionen auszeichnen, die auf einen ganz bestimmten Anwendungsbereich zugeschnitten und die in einem technischen Kontext eingebettet sind, sog. *eingebettete Systeme*. Eingebettete Systeme umgeben uns in unserem täglichen Leben. Dazu zählen beispielsweise die Bereiche Avionik, Medizintechnik, Automobiltechnik, Prozess- und Industriesteuerungen, digitale Netzwerke, Telekommunikation sowie zunehmend der Bereich der Unterhaltungselektronik. Heutzutage sind bereits mehr als 99% aller Prozessoren in eingebetteten Systemen verbaut [417] (und nicht, wie anzunehmen, in Vielweckrechnern). Bemerkenswerterweise lag 2004 der Marktanteil eingebetteter Systeme mit einem Marktvolumen von ca. 46 Milliarden US-Dollar bereits in der gleichen Größenordnung wie der gesamte Markt für Vielweckrechner (PCs, Laptops, Workstations), was man sich vor einigen Jahren

noch nicht vorstellen konnte. Erstaunlich ist auch die von der Marktforschungsfirma BCC [25] im Jahre 2005 vorhergesagte und bis 2009 prognostizierte mittlere jährliche Wachstumsrate von über 14 % für eingebettete Systeme.

Ein konkretes Beispiel eines eingebetteten Systems ist die in Abb. 1.1 dargestellte integrierte Schaltung, die innerhalb eines mobilen Telefonterminäls (GSM-Standard) eingesetzt wird.

Beispiel 1.1.1. Abbildung 1.1 (aus [137]) zeigt ein typisches heterogenes Hardware-/Software-System, bestehend aus einem Prozessor (DSP), anwendungsspezifischer Hardware und Peripherie. Es handelt sich um eine Ein-Chip-Realisierung des GSM-Standards für ein zelluläres Telefon (in Deutschland auch „Handy“ genannt). Dargestellt ist die Aufteilung der Blöcke eines Blockdiagramms auf die Architektur. Der Prozessor wird eingesetzt, um Aufgaben mit niedrigen bis mittleren Datenraten (Codierung/Decodierung) sowie Steuerungsfunktionen zu übernehmen. Die Blöcke, die höhere Rechenleistungen erfordern (Modulation und Demodulation), werden in anwendungsspezifischer Hardware realisiert.

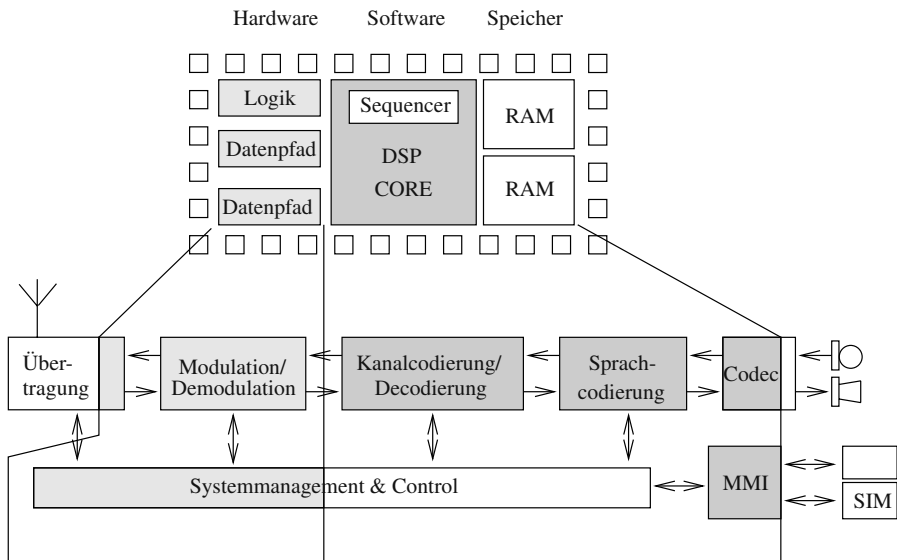


Abb. 1.1. Blockdiagramm der Funktion eines zellulären Telefons nach dem GSM-Standard (unten) und Abbildung der Blöcke auf eine Ein-Chip-Realisierung mit einem DSP-Core (oben)(nach [137])

Abbildung 1.2 zeigt den schematischen Aufbau eines *eingebetteten Systems*. Eingebettete Systeme sind dazu bestimmt, Funktionen als Antwort auf bestimmte Stimuli auszuführen und Daten informationstechnisch zu verarbeiten. Neben dem Heterogenitätsaspekt Hardware-Software treten hier die Aspekte mechanisch-elektrisch

(Sensoren, Aktoren) und analog-digital (Analog-Digital-Umsetzung und Digital-Analog-Umsetzung) auf (siehe auch Abb. 1.1).

In diesem Buch werden jedoch vornehmlich die Komponenten der digitalen Informationsverarbeitung innerhalb eines eingebetteten Systems behandelt, insbesondere die Gebiete des Entwurfs und der Programmierung dieser Komponenten.

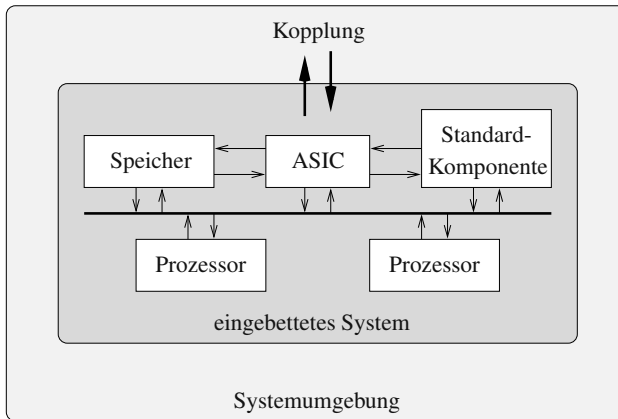


Abb. 1.2. Schematische Darstellung eines eingebetteten Systems

Das große Interesse am systematischen Entwurf von eingebetteten Systemen ist maßgeblich verursacht durch

- Fortschritte in Schlüsseltechnologien (Mikroelektronik, formale Methoden). Dadurch ergibt sich
- eine steigende Vielfalt von Anwendungen und Leistungsanforderungen und damit verbunden
- die Notwendigkeit, Entwurfs- und Testkosten zu senken.

Um die Zielsetzungen dieses Bereiches genauer zu verstehen, ist es sinnvoll, sich dessen historische Entwicklung kurz vor Augen zu führen. Insgesamt lässt sich diese in drei Abschnitte unterteilen:

- Nachdem die Zahl der Objekte in den unteren Entwurfsebenen (Geometrie, physikalische Ebene) aufgrund des Zeitaufwands und der Fehleranfälligkeit ohne Automatisierung nicht mehr handhabbar war, wurden in Industrie und Universitäten Modelle und Methoden entwickelt, die z. B. auf Schaltungssimulation, Platzierung und Verdrahtung abzielten.
- In einem weiteren Schritt wurden dann auch höhere Abstraktionsebenen in die Automatisierung einbezogen, z. B. die Simulation von Schaltungen auf Logikebene oder auch die Logiksynthese.

- Neue Anforderungen bezüglich der Systemkomplexität, der Zeitspanne zwischen Produktidee und Markteinführung, sowie der Zuverlässigkeit und Güte führen nun zur Entwurfsautomatisierung auf der noch abstrakteren *Systemebene*.

Auf der Systemebene besteht eine zentrale Aufgabe darin, eine Aufteilung der Funktionalität in Hard- und Softwarekomponenten vorzunehmen (sog. *Hardware/Software-Partitionierung*).

Beispiel 1.1.2. Ein Netzwerkcontroller soll entworfen werden, der einen Speicher mit einer seriellen Schnittstelle koppelt (siehe Abb. 1.3 aus [146]). Seine Aufgabe besteht darin, Daten über die serielle Schnittstelle zu senden und zu empfangen und dabei ein bestimmtes Protokoll einzuhalten (z. B. CS/CD für Ethernet oder Infini-Band [358, 191]). Dabei ist die maximale Datenübertragungszeit T_{\max} (in ns) für ein Kilobyte an Daten zu unterschreiten. Das System ist ferner einer Kostenschranke K_{\max} (in Dollar) unterworfen und soll nicht mehr als P_{\max} (in mW) Leistung verbrauchen. Offensichtlich muss man zunächst die Entscheidung treffen, welche Aufgaben in Software und welche in Hardware realisiert werden. Eine exemplarische Aufteilung ist in Abb. 1.3 dargestellt.

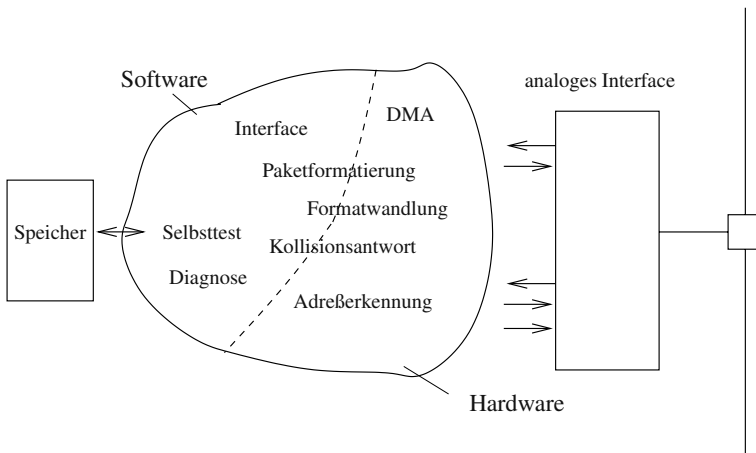


Abb. 1.3. Netzwerkcontroller: exemplarische Partitionierung der zu implementierenden Funktionalität in Hardware und Software (aus [146])

In den meisten Fällen erfolgt die Hardware/Software-Partitionierung durch Abschätzung von Kosten und Performanzanforderungen nach dem jeweiligen Erfahrungswissen des Entwicklers/der Entwicklerin. Da diese Entwurfsentscheidung nun aber auf groben Schätzungen beruht, ist keine Gewährleistung gegeben, dass das realisierte, fertige System alle Entwurfsbeschränkungen erfüllt bzw. in einer gewissen Hinsicht optimal ist. Solche Systeme sind üblicherweise in mindestens einer Eigenschaft unter- oder überdimensioniert, wie folgendes Beispiel zeigt.

Beispiel 1.1.3. Abbildung 1.4 zeigt verschiedene Lösungen zur Realisierung des Netzwerkcontrollers aus Beispiel 1.1.2. Die Menge von Entwurfsbeschränkungen

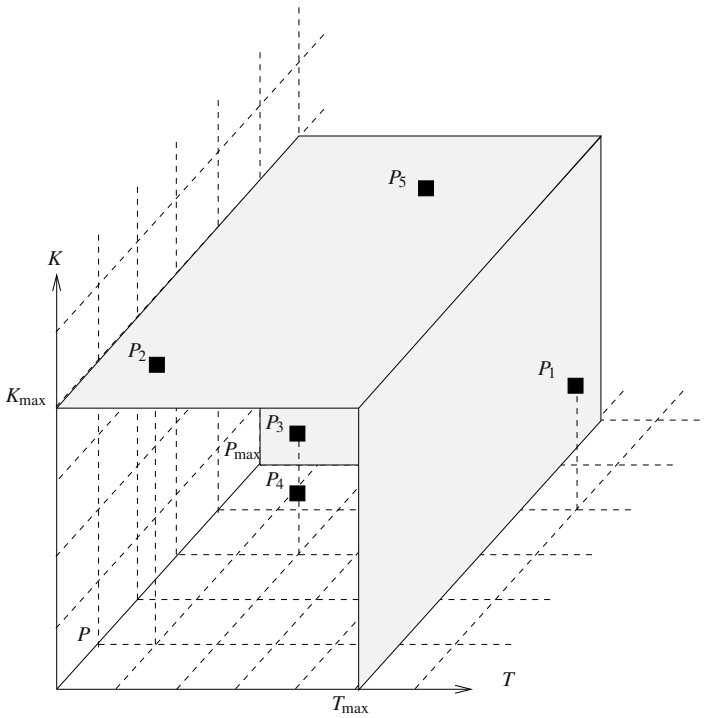


Abb. 1.4. Verschiedene Realisierungsvarianten eines Netzwerkcontrollers aus Beispiel 1.1.2

gen kennzeichnet hier einen dreidimensionalen *Entwurfsraum* mit den Achsen Datenübertragungszeit/kByte T , Kosten K und Leistungsverbrauch P . Ein/e mit der Entwicklung beauftragter Softwareingenieur/in konstruierte ein System mit einem Mikroprozessor und erhielt ein System mit den durch den Entwurfspunkt P_1 gekennzeichneten Eigenschaften. Diese Lösung erfüllt zwar die Kostenanforderungen, aber nicht die Datenratenbeschränkung. Ein/e Hardwareingenieur/in entwickelte eine dedizierte integrierte Schaltung, deren Eigenschaften durch den Punkt P_2 dargestellt sind. Offensichtlich erfüllt diese Realisierung die Performanzanforderungen und Leistungsverbrauchsanforderungen, nicht aber die Kostenbeschränkung. Das System wurde überdimensioniert.

Der Punkt P_3 entspricht einer gemischten Hardware/Software-Lösung. Leider ist diese Lösung nicht optimal, da der Punkt P_4 , der der in Abb. 1.3 dargestellten Aufteilung entspricht, in allen Eigenschaften besser oder gleich gut ist. Viele Lösungen, bei denen die Hardware/Software-Partitionierung „ad hoc“ bestimmt worden ist, erfüllen die Entwurfsbeschränkungen nicht oder sind suboptimal.

Aktuelle Systeme sind weitaus komplexer als die bereits vorgestellten Beispiele. Als Beispiel sei der Funktionsumfang heutiger Handys genannt. Abbildung 1.5 zeigt diese Funktionen in einem Blockschaltbild. Die Umsetzung als Multi-Chip-

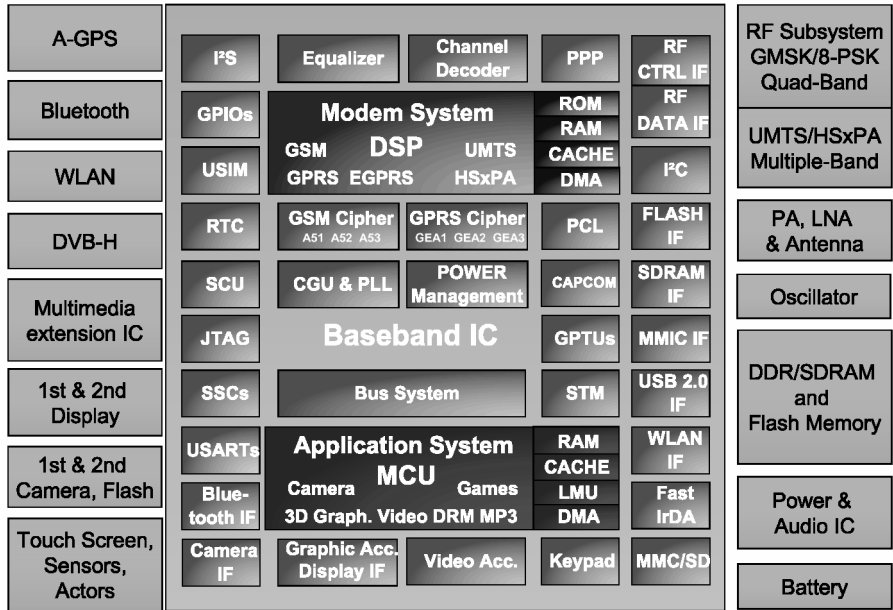


Abb. 1.5. Funktionen eines modernen Mobilfunkgerätes. Quelle: Dietmar Wenzel, Infineon Technologies AG

Lösung ist in Abb. 1.6 zu sehen. Für die Basisbandfunktionalität wird ein speziell gefertigter Prozessor verwendet, der neben einer RISC-CPU und einem Digitalen Signalprozessor (DSP) auch ein analoges Interface besitzt. Als Interfaces besitzt ein Handy heutzutage neben einigen Video- und Audio-Codecs (u. a. MPEG-4, H.263, H.264, ACC, MP3) zahlreiche weitere Schnittstellen (u. a. USB, Kamera, Display, A/D-Umsetzer, IrDA, Bluetooth etc).

Aus diesen Beispielen sollte deutlich geworden sein, dass automatische Syntheseverfahren für eingebettete Hardware/Software-Systeme nützlich und erforderlich sind, damit die Systementwicklung mit dem Technologiefortschritt standhalten kann und der Entwurf effizienter Systeme möglich ist. Die Verfahren der Synthese und Optimierung stehen dabei im Vordergrund dieses Buches.

Der folgende Abschnitt 1.2 enthält eine historische und inhaltliche Gliederung, die den Systementwurf, so wie er hier verstanden wird, in die heutige Praxis einbet-

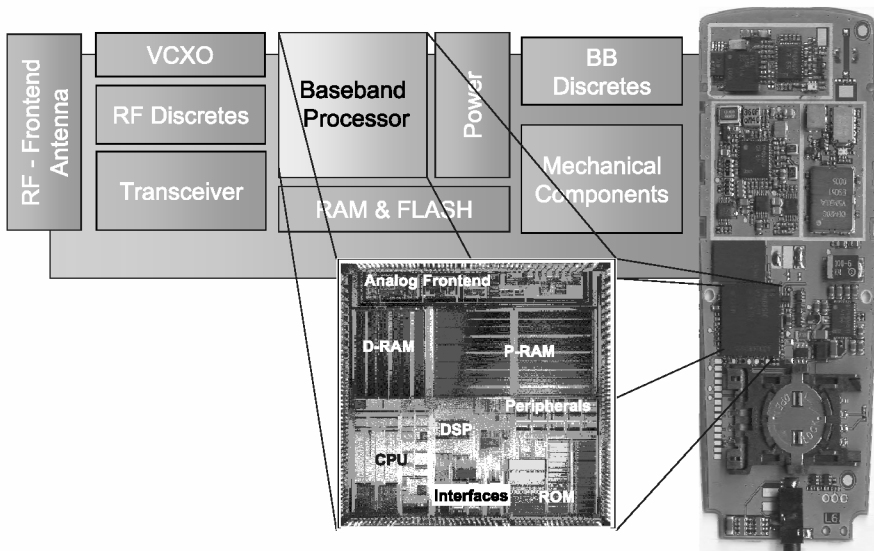


Abb. 1.6. Hardware/Software-Implementierung des Mobilfunkgerätes aus Abb. 1.5. Quelle: Dietmar Wenzel, Infineon Technologies AG

tet. Abschnitt 1.3 dient der Erläuterung der wesentlichen Grundbegriffe Abstraktion und Entwurfsrepräsentationen.

1.2 Entwurfsmethodik

Historisch gesehen durchlief die Entwurfsautomatisierung drei Evolutionsstufen, die im Folgenden beschrieben werden.

1.2.1 Erfassen und simulieren

Gajski et al. [122] beschreiben die bis heute am meisten eingesetzte Entwurfsmethodik für integrierte Schaltungen als eine Methodik des „Erfassens und Simulierens“: Man startet mit einer nichtformalen, umgangssprachlichen Spezifikation des Produktes, die noch keine Informationen über die konkrete Implementierung enthält. Es wird nur die Funktionalität, nicht aber die Art und Weise ihrer Realisierung, bestimmt. Anschließend wird eine grobe Blockstruktur der Architektur entworfen, die eine verfeinerte, aber immer noch unvollständige Spezifikation des Gesamtsystems darstellt. In weiteren Verfeinerungsschritten werden die einzelnen Blöcke

dann in Logik- oder sogar Transistordiagramme umgesetzt und in existierende CAD-Entwurfssysteme eingegeben. Auf dieser Basis lassen sich dann umfangreiche Simulationen der Funktionalität und des Zeitverhaltens sowie Untersuchungen der Testbarkeit durchführen. Die erfassten Diagramme dienen zudem möglicherweise dazu, Zellen auf physikalischer Ebene zu platzieren und miteinander zu verdrahten oder auch das Layout einer integrierten Schaltung automatisch zu generieren.

Dieser Ansatz des „Erfassens und Simulierens“ wird übrigens nicht nur im Bereich des Hardwareentwurfs, sondern auch bei der Programmierung von Mikroprozessorsystemen, also beim Softwareentwurf, eingesetzt. Eine nichtformale Spezifikation wird in Blockstrukturen und anschließend in Assemblerprogramme verfeinert. Bevor das endgültige Programm in Maschinensprache generiert wird, erfolgen Simulationen und Emulationen zur Validierung von Funktionalität und Zeitverhalten.

1.2.2 Beschreiben und synthetisieren

Seit vielen Jahren ist die Logiksynthese fester Bestandteil der Entwurfsmethodik von Hardwaresystemen. Ein Entwurf wird hier bezüglich seines Verhaltens in einer simulierbaren, d. h. ausführbaren Beschreibungssprache spezifiziert; so kann man z. B. ein Steuerungssystem durch Boolesche Gleichungen und Zustandsdiagramme beschreiben. Die Struktur der Implementierung wird dann automatisch durch entsprechende Syntheseverfahren generiert, im Vergleich zum Handentwurf eine sehr viel schnellere und vor allem sichere Entwurfsart.

Dieser Weg des „Beschreibens und Synthetisierens“ [122] kann nun bei verschiedenen Teilproblemen eines gesamten Entwurfs durchgeführt werden: Auf der *Logikebene* werden funktionale Einheiten (z. B. Multiplizierer, arithmetisch-logische Einheiten (engl. *arithmetic logical units* (ALUs)) etc.) und Steuerungseinheiten (z. B. Zustandsmaschinen) durch die Logiksynthese automatisch generiert. Hierzu gehören Verfahren zur Minimierung Boolescher Ausdrücke, Zustandsminimierungen und die *Technologieabbildung* [258, 26], d. h. die Implementierung der minimierten Funktionen mit Gattern aus einer speziellen Entwurfsbibliothek.

Ein anderes Beispiel ist die *Architektursynthese* (engl. *high-level synthesis* oder auch *behavioral synthesis*) [94]. Darunter versteht man die Synthese von integrierten Schaltungen, die aus einem Datenpfad (Operationswerk) und einem Kontrollpfad (Steuerwerk) bestehen. Charakteristisch für die Beschreibung des Verhaltens solcher synthetisierbarer Systeme sind Datenfluss- und Kontrollflussgraphen. Die Transformation in eine strukturelle Beschreibung erfolgt durch Lösen der drei Syntheseaufgaben *Allokation*, *Ablaufplanung* und *Bindung*.

- Aufgabe der *Allokation* ist es, die Zahl und Art der Komponenten zu bestimmen, die in der Implementierung verwendet werden sollen, also zum Beispiel die Zahl der Register und Speicherbänke, die Zahl und Arten der internen Busse zur Datenkommunikation sowie die verwendeten funktionalen Einheiten wie Multiplizierer oder ALUs. Daher dient die Allokation im Wesentlichen der Abwägung von Kosten und Leistungsfähigkeit einer Schaltung.

- Die *Ablaufplanung* weist den spezifizierten Aufgaben Zeitintervalle der Abarbeitung zu, so dass anschließend in jedem Zeitschritt bekannt ist, welche Operationen wann ausgeführt werden.
- Die *Bindung* ordnet abschließend den Daten entsprechende Speicherzellen, den Operationen funktionale Einheiten und den Datenkommunikationen einen Bus oder eine Verbindungsleitung zu.

Auch hier gibt es wieder eine direkte Parallele zum *Softwareentwurf*. Im Gegensatz zum „Erfassen und Simulieren“ wird die Funktionalität des Systems durch eine ablauffähige *Programmiersprache* spezifiziert, z. B. C, C++, JAVA. Aufgabe des Übersetzers ist dann die automatische Generierung eines Maschinenprogramms. Wenn es sich um eine parallele Zielarchitektur handelt, sind wiederum die drei wesentlichen Aufgaben der Allokation, Ablaufplanung und Bindung bei der Übersetzung auszuführen. Auch wenn beim Softwareentwurf die Zielarchitektur i. Allg. gegeben ist, sind in beiden Fällen fast die gleichen Ablaufplanungs-, Bindungs- und Optimierungsproblemstellungen zu lösen. So sind auch bei der Softwareübersetzung den Operationen und Datentransporten Zeitschritte unter Berücksichtigung der zur Verfügung stehenden Ressourcen zuzuordnen wie beispielsweise Busbandbreite, Zahl der Busse, Zahl der internen Register, Speicherbedarf und Zahl und Art der parallelen arithmetischen Einheiten. Ziel der Bindung ist hier ebenfalls, die Variablen, Operationen und Datentransporte den vorhandenen physikalischen Einheiten zuzuordnen. Diese Verfahren werden vor allem in Übersetzern für die heutigen superskalaren RISC-Prozessoren (z. B. PowerPC, Alpha-Prozessor), für VLIW-Rechner (engl. *very long instruction word computer*) oder auch für Signalprozessoren eingesetzt, da sie alle durch interne Parallel- und Fließbandverarbeitung (engl. *pipelining*) ausgezeichnet sind.

1.2.3 Spezifizieren, explorieren und verfeinern

Auf der Abstraktionsebene komplexer Systeme ist die Entwurfsmethodik bei weitem noch nicht so ausgereift wie in den bisher beschriebenen Bereichen. Dennoch hat sich hier ein Paradigma durchgesetzt, das sich durch die ebenfalls von Gajski et al. [122] geprägten Stichworte „Spezifizieren, explorieren und verfeinern“ beschreiben lässt: In der *Spezifikationsphase* wird in einem sehr frühen Stadium des Entwurfsprozesses eine Spezifikation des Gesamtsystems erstellt. Sie ist Ausgangspunkt und Grundlage für

- die Beschreibung der Funktionalität eines Systems (z. B. um die Wettbewerbsfähigkeit eines Produktes abzuschätzen),
- die Dokumentation des Entwurfsprozesses in allen Schritten,
- die automatische Verifikation kritischer Systemeigenschaften,
- die Untersuchung und Exploration verschiedener Realisierungsalternativen,
- die Synthese der Teilsysteme und
- die Veränderung und Nutzung bereits bestehender Entwürfe.

Die *Explorationsphase* dient dazu, Realisierungsalternativen eines durch verschiedenste Anforderungen und Metriken beschränkten Entwurfsraums miteinander zu vergleichen. Eine der Hauptaufgaben dabei besteht darin, die zu implementierende Funktionalität auf mögliche Komponenten eines heterogenen Systems zu verteilen. Diese Teilsysteme können nun anwendungsspezifische integrierte Schaltungen, vorgefertigte Mikroprozessoren, aber auch vorhandene Spezialbausteine sein. Da jede neue Partitionierungsvariante einer unterschiedlichen Systemrealisierung entspricht, verlangt die Bewertung eine Vorausschätzung wesentlicher Eigenschaften wie Verarbeitungsleistung, Kosten, Leistungsverbrauch und Testbarkeit.

In der anschließenden *Verfeinerungsphase* wird die Spezifikation entsprechend der Partitionierung und Allokation in verschiedene Hardware- und Softwarekomponenten aufgeteilt. Die Ausgangslage ist vergleichbar mit der nach der Bestimmung eines Blockdiagramms auf der Grundlage einer nichtformalen Spezifikation, siehe Abschnitt 1.2.1. Im Unterschied dazu wurde diese Aufteilung aber nach der Exploration eines großen Entwurfsraums erhalten. Zudem steht die Verfeinerung auf „sicheren Füßen“, da sie formal aus der gegebenen Spezifikation abgeleitet wurde. In weiteren Verfeinerungsschritten kann dann der gesamte Prozess der „Exploration und Verfeinerung“ wiederholt werden, bis eine vollständig strukturelle Beschreibung als Implementierung des Systems vorliegt. Durch ein solches Vorgehen werden nicht nur frühzeitig mögliche Entwurfsalternativen (z. B. Software statt Hardware, anwendungsspezifische Schaltungen statt Standardkomponenten) geprüft, sondern es entfallen auch teure und zeitraubende Entwurfsiterationen.

1.3 Abstraktion und Entwurfsrepräsentationen

Die folgenden Abschnitte enthalten eine kurze Darstellung der verschiedenen Abstraktionsebenen und Sichten eines Systems sowie eine Klassifizierung der Synthese- und Optimierungsprobleme beim Systementwurf.

1.3.1 Modelle

Unter einem *Modell* versteht man die formale Beschreibung eines Systems oder Teilsystems. Hierbei wird das zu modellierende Objekt unter einem ganz bestimmten „Blickwinkel“ betrachtet, d. h., es werden nur bestimmte Eigenschaften ohne die zugehörigen Details gezeigt. Diesen Vorgang nennt man *Abstraktion*. Die vorangegangenen Abschnitte sollten deutlich gemacht haben, dass der Entwurf eines Systems auf dem Prinzip der *Verfeinerung* beruht: Der Grad an Detailliertheit wird beim Entwurf schrittweise erhöht. Daher kann man Modelle anhand des Grades Ihrer Verfeinerung nach Abstraktionsebenen klassifizieren.

Auf der anderen Seite gibt es auch unterschiedliche *Sichten* eines Objektes. So kann man eine Schaltung als Verbindung von Einzelkomponenten betrachten oder als eine Einheit mit bestimmtem Verhalten. Abstraktionsebenen und Sichten sind in gewisser Weise orthogonal zueinander.

Obwohl man natürlich (fast) beliebig viele Schichten an Sichten und Abstraktion einführen kann, werden hier vor allem die Abstraktionsebenen der *Architektur* und *Logik* beim Hardwareentwurf, die Ebenen *Modul* und *Block* beim Softwareentwurf und die Ebene *System* beim Entwurf ganzer Systeme behandelt. An Sichten werden dabei *Verhalten* und *Struktur*, siehe auch die graphische Darstellung in Abb. 1.7, betrachtet.

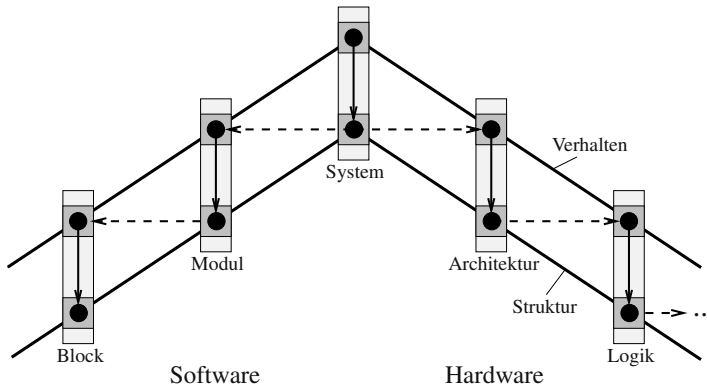


Abb. 1.7. Das Doppeldachmodell: Graphische Darstellung einiger wichtiger Abstraktionsebenen und Sichten, die beim Entwurf eingebetteter Systeme durchlaufen werden. Idealerweise erfolgt der Entwurf *Top-down*, d. h. von höheren zu niedrigeren Abstraktionsebenen. Die vertikalen Pfeile stellen Synthese- und Verfeinerungsschritte dar.

Die folgende Aufstellung soll die unterschiedlichen Abstraktionsebenen stichwortartig beschreiben:

System: Die Modelle der Systemebene beschreiben das zu entwerfende Gesamtsystem auf der Ebene von Netzwerken aus komplexen, miteinander kommunizierenden Teilsystemen, die jeweils komplexe Aufgaben (z. B. Algorithmen, Tasks) berechnen.

Architektur: Die Architekturebene gehört zum Bereich des Hardwareentwurfs. Modelle auf dieser Ebene beschreiben kommunizierende funktionale Blöcke, die komplexe arithmetische und logische Operationen ausführen.

Logik: Die Logikebene gehört ebenfalls zum Hardwarebereich. Die Modelle dieser Ebene beschreiben verbundene Gatter und Register, die Boolesche Funktionen berechnen.

Modul: Die Modulebene gehört zum Softwarebereich. Die entsprechenden Modelle beschreiben Funktion und Interaktion komplexer Module (z. B. kommunizierende Prozesse oder Tasks).

Block: Die Blockebene gehört ebenfalls zum Softwarebereich. Die entsprechenden Modelle beschreiben Programme mit Daten- und Kontrollabhängigkeiten zwi-

schen Grundblöcken, welche aus Instruktionen aufgebaut sind, die auf der zugrunde liegenden Rechnerarchitektur elementare Operationen ausführen.

Neben dieser Klassifizierung von Modellen nach ihrem Grad der Abstraktion unterscheidet man zudem verschiedene *Sichten* innerhalb einer Abstraktionsebene.

Verhalten: In der Verhaltenssicht werden Funktionen unabhängig von ihrer konkreten Implementierung beschrieben.

Struktur: In der strukturellen Sicht hingegen werden kommunizierende Komponenten beschrieben. Aufteilung und Kommunikation entsprechen der tatsächlichen Implementierung.

Anhand dieser Klassifizierung lässt sich (sehr vereinfacht dargestellt) der Entwurf eines komplexen Systems als Abfolge von *Verfeinerungsschritten* verstehen, bei denen einer Verhaltensbeschreibung strukturelle Informationen über die Implementierung zugefügt werden und die entstehenden Teilmodule dann wieder Ausgangspunkte von Verfeinerungen (horizontale Pfeile in Abb. 1.7) auf der nächst niedrigeren Abstraktionsebene sind. Da die beiden Sichten Verhalten und Struktur jeweils ein Dach über die unterschiedlichen Abstraktionsebenen aufspannen, sprechen wir im Folgenden auch von dem *Doppeldachmodell*.

Bei dieser idealisierten Darstellung des Entwurfsprozesses wird allerdings stark vereinfachend außer Acht gelassen, dass bei einem konkreten Entwurf viele Iterationen zwischen den Abstraktionsebenen notwendig werden, also i. Allg. weder ein reiner *Top-down-Entwurf* (engl. *top down* = von oben nach unten) noch ein reiner *Bottom-up-Entwurf* (engl. *bottom up* = von unten nach oben) vorliegt. Einige Systemkomponenten werden zudem direkt auf unteren Abstraktionsebenen entworfen, so dass zu einem bestimmten Zeitpunkt im Entwurfsprozess nicht alle Systemkomponenten den gleichen Abstraktions- bzw. Verfeinerungsgrad aufweisen.

Aufgabe der *Synthese* (vertikale Pfeile in Abb. 1.7) ist nun die (teil-)automatische Transformation einer Beschreibung auf Verhaltenssicht in eine Beschreibung auf Struktursicht. Um die Zusammenhänge etwas deutlicher zu machen, sollen nun anhand von Synthesebeispielen zumindest einige der besprochenen Ebenen und Sichten näher erläutert werden.

1.3.2 Synthese

In diesem Kapitel werden die Syntheseschritte auf den einzelnen Abstraktionsebenen anhand von Beispielen vorgestellt. Besonders ausführlich wird hierbei auf die *Systemebene* eingegangen, da diese eine zentrale Rolle im Entwurf von digitalen Hardware/Software-Systemen einnimmt.

Systemsynthese

Wie auch in den anderen Abstraktionsebenen, ist die Systemebene durch charakteristische Beschreibungsformen bezüglich des Verhaltens und der Struktur gekennzeichnet. Das *Verhalten* wird durch Leistungsanforderungen beschrieben, wie funktionale Spezifikation, Zeitverhalten und nichtfunktionale Eigenschaften. Die *strukturelle*

Beschreibung zeigt das System als Netzwerk aus Prozessoren, Standardkomponenten, anwendungsspezifischen integrierten Schaltungen, Verbindungsstrukturen und Speicherbausteinen.

Beispiel 1.3.1. Abbildung 1.8 zeigt eine physikalische Sicht eines Hardware/Software-Systems auf Systemebene. Es handelt sich um eine *Ein-Chip-Realisierung* oder ein sog. *System-on-a-Chip (SoC)*, die einen Prozessor (rechts) als Makrozelle (engl. *processor core*) sowie eine Menge von Logik- (u. a. ein Gatearray links), Speicher- (Mitte) und Peripherieblöcken (z. B. Timer, Digital-Analog-Umsetzer und Analog-Digital-Umsetzer) auf dem Chip integriert. Mit dem Fortschritt der Miniaturisierung in der Halbleitertechnologie ist es heutzutage sogar möglich, dass mehrere vernetzte Prozessoren auf einem einzigen Chip integriert werden. Man spricht dann von einem *MPSoC*, einem sog. *Multiple Processor System-on-a-Chip*. In vielen Fällen sind die einzelnen Komponenten in Anzahl und Größe individuell konfigurierbar. Andere Realisierungsformen von hier betrachteten Hardware/Software-Systemen sind Ein- und Mehrplatinenentwürfe.

Eine entscheidende Entwurfsaufgabe auf Systemebene ist offensichtlich die Partitionierung der Verhaltensbeschreibung in Teilsysteme. Hierbei spielen sehr unterschiedliche Optimierungskriterien eine Rolle wie Kosten, Verarbeitungsleistung und Leistungsverbrauch, aber auch andere nichtfunktionale Kriterien. Dazu gehören beispielsweise die Wiederverwendbarkeit des Entwurfs in zukünftigen Produktlinien und die „Time-to-market“. Dies ist die Zeit von der Konzeptualisierung eines Systems bis zur Auslieferung des Produktes an den Kunden. Die Flexibilität eines Produktes (gegenüber kleinen Produktänderungen) ist eine weitere denkbare nichtfunktionale Eigenschaft.

Das Interesse an automatisierten Syntheseverfahren auf der Systemebene lässt sich vor allem auf die folgenden Beweggründe zurückführen:

Kurze Entwurfszyklen: Ein automatisiertes Entwurfssystem ist in der Lage, einen Entwurf schneller durchzuführen, als dies ein Entwickler/eine Entwicklerin ohne Unterstützung von CAD-Werkzeugen könnte. Man denke nur an die Zeiterparnis durch Werkzeuge zum automatisierten Platzieren und Verdrahten von Leiterplatten und integrierten Schaltungen. In fast allen Bereichen der Technik ist in den vergangenen Jahren eine enorme Reduktion der Produktlebensdauer und somit der „Time-to-market“ festzustellen. Mit dieser Entwicklung kann man nur durch den Einsatz geeigneter CAD-Verfahren Schritt halten.

Reduzierte Entwurfsfehler: Um kostspielige Iterationen aufgrund von Fehlern im Entwurf zu vermeiden, wird auf die Entwicklung von Synthesewerkzeugen Wert gelegt, die „beweisbar“ korrekte Entwürfe liefern. Dies gelingt einerseits dadurch, dass der Verfeinerungsvorgang von einer Verhaltensbeschreibung hin zu einer strukturellen Beschreibung als eine Sequenz von Programmtransformationen verstanden wird, oder andererseits dadurch, dass formale Verifikationsverfahren eingesetzt werden.

Exploration des Entwurfsraums: Gerade auf den obersten Entwurfsebenen werden grundlegende Entwurfsentscheidungen getroffen, die die Leistungsfähigkeit und

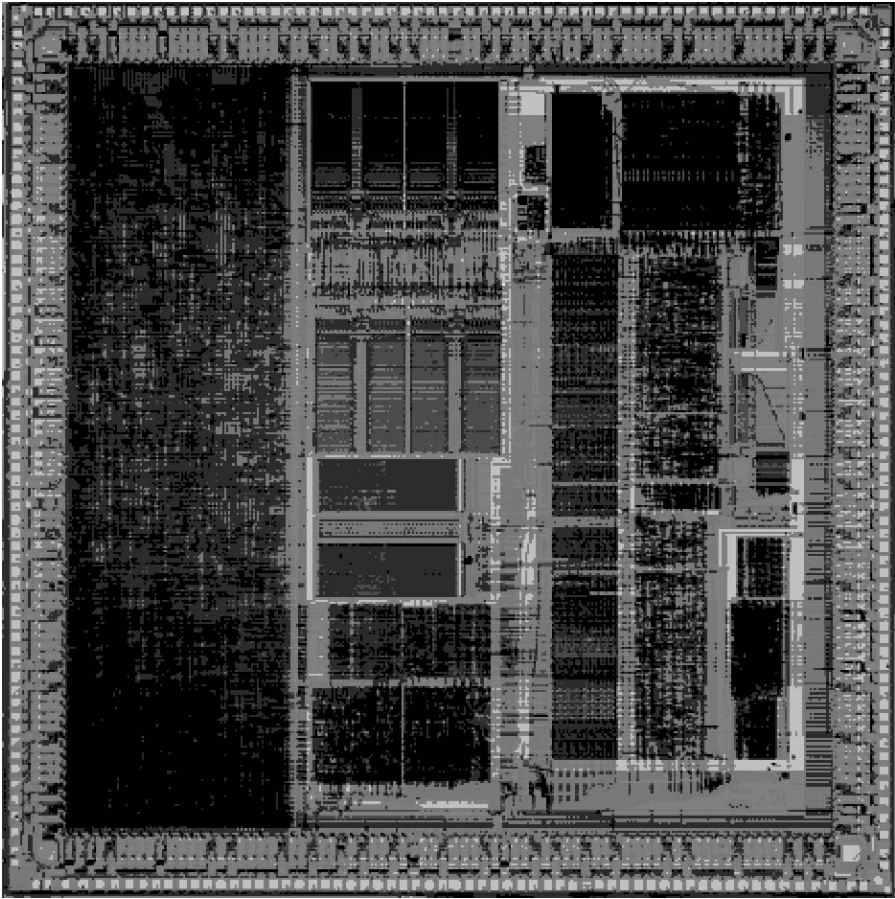


Abb. 1.8. Physikalische Sicht einer Ein-Chip-Realisierung eines Hardware/Software-Systems.
Quelle: Texas Instruments, cDSP

Kosten des implementierten Systems bestimmen. Die Konsequenzen von Fehlentscheidungen werden somit in einem frühen Entwurfsstadium deutlich, z. B. die Verletzung von Zeitbeschränkungen. Man kann sich auf der Systemebene ein Entwurfswerkzeug vorstellen, mit dem unterschiedliche Realisierungsarten einer Spezifikation schnell bewertet werden können, also eine Exploration des Entwurfsraums unter Optimierungsgesichtspunkten unterstützt wird. Ein solches Werkzeug mit dem Namen *SystemCoDesigner* [162] wird im Rahmen dieses Buches vorgestellt.

Beispiel 1.3.2. Das folgende Beispiel zeigt einige typische Probleme, die bei einem Entwurf auf Systemebene entstehen. In digitalen Video-Anwendungen ist es oft notwendig, die erforderlichen Übertragungsbandbreiten durch eine geeignete Daten-

kompression zu reduzieren. Das folgende Beispiel beschreibt einen Hybrid-Codierer, der Transformationscodierung und prädiktive Codierung kombiniert. Der Kompressionsfaktor einer reinen Bildcodierung wird durch ein prädiktives Schema für Bildfolgen verbessert. Ein Block innerhalb eines Bildes wird geschätzt aus einem Block innerhalb des vorangegangenen Bildes. Abbildung 1.9 zeigt eine Darstellung eines solchen Hybrid-Codierers auf Verhaltensebene.

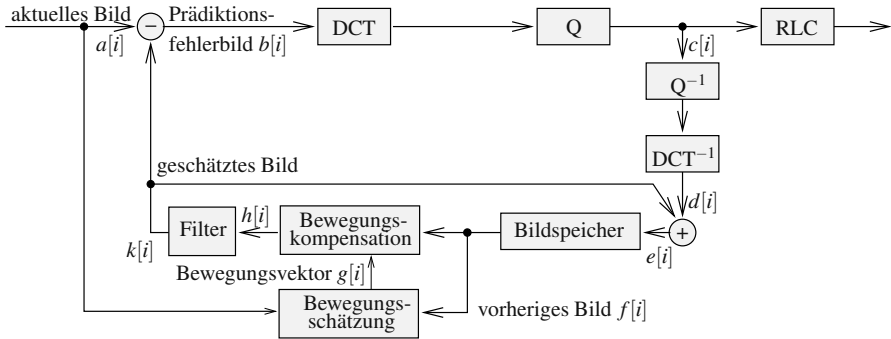


Abb. 1.9. Darstellung eines Hybrid-Codierers für Bildsequenzen

Aus der folgenden Beschreibung wird deutlich, dass die einzelnen Blöcke der Darstellung komplexe Teiloperationen beschreiben und die Kommunikation mittels komplexer Datentypen erfolgt (hier Bildsequenzen, wobei die Bilder ihrerseits wieder aus Blöcken, Makroblöcken und einzelnen Pixeln zusammengesetzt sind). Die zweidimensionale diskrete Kosinustransformation (DCT) wird auf nichtüberlappende Blöcke des Prädiktionsfehlerbildes $b[i]$ angewendet. Die transformierten Blöcke repräsentieren den räumlichen Frequenzinhalt des entsprechenden Blocks. Durch die Transformation der Bilddaten in den Frequenzbereich kann die Bildinformation nach der Quantisierung (Q) durch einige wenige Koeffizienten dargestellt werden, ohne dass sich subjektiv eine Bildverschlechterung ergibt (Irrelevanzreduktion). Die abschließende Codierung (RLC) benutzt die Dynamik der zu übertragenden Werte zur Reduktion der Datenrate. Statt aufeinander folgende Bilder einer Sequenz unabhängig voneinander zu codieren, kann man eine höhere Reduktion erzielen, in dem man eine Bewegungsschätzung und Bewegungskompensation durchführt und zusätzlich Differenzbilder codiert (Ausnutzung örtlicher und zeitlicher Redundanzen der Bildinhalte). Ein Block im Bild $a[i]$ wird verglichen mit Nachbarblöcken des vorangegangenen Bildes $f[i]$. Hieraus wird ein Bewegungsvektor $g[i]$ bestimmt. Als Resultat der Bewegungskompensation erhält man ein geschätztes Bild $k[i]$. In der Darstellung nach Abb. 1.9 werden Teilalgorithmen als Blöcke dargestellt. Hier gibt es noch keine Spezifikation des zeitlichen Ablaufs, der Abbildung auf eine Zielarchitektur, der Speichergrößen und der Partitionierung von Bildern in Blöcke oder Makroblöcke.

Abbildung 1.10 zeigt eine mögliche Systemarchitektur, die aus den Komponenten BUS, CM (Steuerungsprozessor für die Speicherzugriffe und Busarbitrierung), FC (Bildspeicher), BM (Spezialmodul für die Bewegungsschätzung), BC (lokaler Speicher für das BM), PM (allgemein programmierbarer Prozessor) und GC (lokaler Speicher für die PM-Module) besteht.

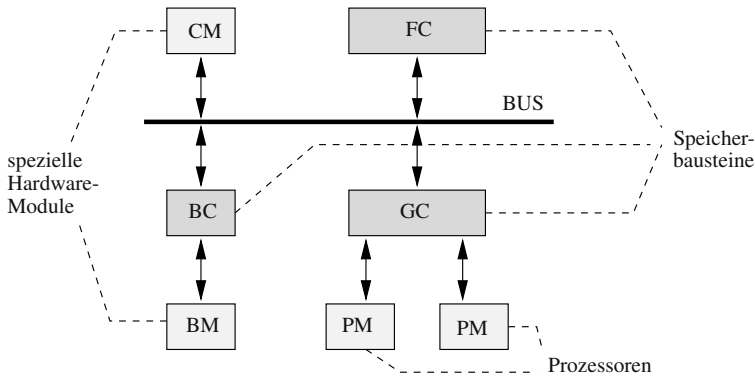


Abb. 1.10. Strukturelle Sicht einer möglichen Implementierung eines Video-Codex

Neben den Beschreibungsformen unterscheiden sich die verschiedenen Abstraktionsebenen vor allem in den Freiheitsgraden, die bei der Verfeinerung von der Verhaltenssicht auf eine strukturelle Sicht bestehen. Die Entwurfsfreiheit nimmt von den oberen Abstraktionsebenen zu den niedrigeren Abstraktionsebenen immer weiter ab. Auf der Systemebene ist es sicherlich müßig, sich über Details der Implementierung Gedanken zu machen, bevor nicht grundlegende Überlegungen bezüglich der Systemarchitektur getroffen sind. Insbesondere können die folgenden Implementierungsentscheidungen getroffen und die resultierenden Kosten- und Leistungsfaktoren gegeneinander aufgewogen werden:

- Festlegung der Komponententypen, die in der Implementierung verwendet werden (*Allokation*), z. B. Mikroprozessor, ASIC, Speicherbausteine.
- Festlegung der Anzahl der jeweiligen Komponenten, Auswahl und Dimensionierung der Verbindungsstruktur.
- Zuordnung der Variablen zu Speicherbausteinen, Operationen zu Funktionsbausteinen und Kommunikationen zu Bussen (*Partitionierung, Bindung*). Hierbei sind auch Realisierungen in Hardware und in Software gegeneinander abzuwägen.
- *Ablaufplanung* der Teilalgorithmen auf den allozierten Komponenten.

Beispiel 1.3.3. In Zusammenhang mit dem vorangegangenen Beispiel gibt es verschiedene Zielarchitekturen, auf die das gesamte System abgebildet werden kann, z. B.

- einen Mikroprozessor, Signal- oder Bildprozessor,
- mehrere parallel arbeitende programmierbare Prozessoren,
- eine Erweiterung der oben genannten Architekturen mit spezialisierten funktionalen Einheiten, z. B. für die diskrete Kosinustransformation oder die Bewegungsschätzung,
- eine reine spezialisierte Hardwarelösung, die an den Algorithmus genau angepasst ist.

Zu jeder dieser Implementierungen existieren wiederum verschiedene Möglichkeiten der Zuordnung von Daten zu Speichern und Teilalgorithmen zu Modulen, der Wahl von Kommunikationsstruktur und Busbandbreiten sowie der Ablaufplanung der einzelnen Teilalgorithmen. Als Beispiel einer nur graduellen Änderung könnte man die Kommunikation zu den lokalen Cache-Speichern verändern und einen der allgemein programmierbaren Prozessoren durch einen Spezialbaustein DM mit privatem Cache-Speicher DC ersetzen, der effizient die diskrete Kosinustransformation berechnen kann, siehe Abb. 1.11.

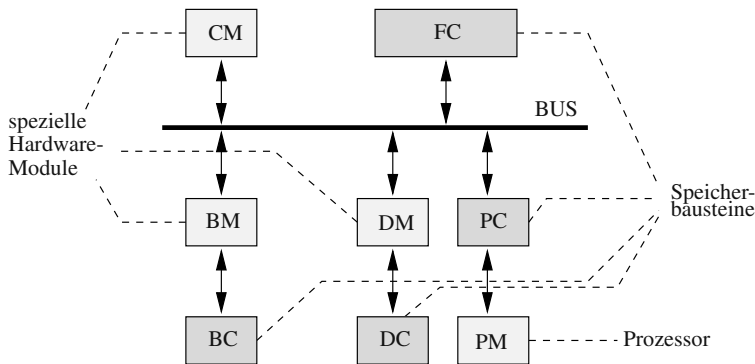


Abb. 1.11. Strukturelle Sicht auf eine leicht veränderte Implementierung eines Bildcodierers

Einige der im Rahmen der Synthese auf Systemebene erforderlichen Aufgaben sind in Abb. 1.12 dargestellt.

Die gesamte *Entwurfsmethodik* auf Systemebene sollte eine einfache und effiziente Möglichkeit bieten, verschiedene Entwurfsalternativen zu untersuchen. Voraussetzung ist zunächst eine (ausführbare) *Spezifikation* des gewünschten Systemverhaltens. Anforderungen an eine solche Spezifikation sind Simulierbarkeit, Möglichkeit zur formalen Verifikation, Verständlichkeit, Möglichkeit zur Anbindung an CAD-Werkzeuge und Vollständigkeit (Beschreibung aller Systemeigenschaften).

Die folgenden Schritte hängen eng miteinander zusammen. In einer *Allokationsphase* müssen zunächst die Komponenten der Architektur ausgewählt werden, z. B. Prozessoren, Speicher und anwendungsspezifische integrierte Schaltungen. Diese Ressourcen sind charakterisiert durch Instruktionsatz, Zahl der Instruktionen pro

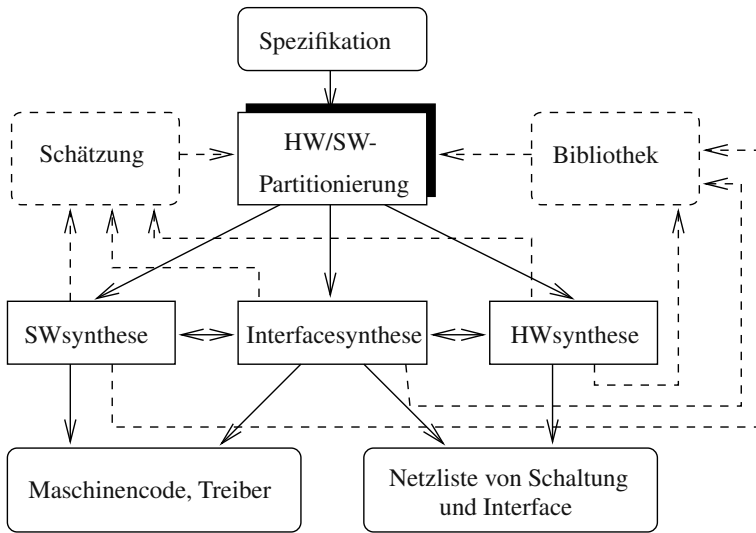


Abb. 1.12. Grobe Darstellung eines möglichen Entwurfsablaufs auf Systemebene

Zeiteinheit (Prozessoren), Zahl der möglichen Gatter, Verzögerungszeiten der Gatter, Leistungsverbrauch (ASICs), Speichergröße, Schreib-Lese-Protokolle und Zugriffszeiten (Speicher).

Die Komponenten der Architektur, die in der Allokationsphase Berücksichtigung finden, hängen oftmals stark von den Erfahrungen des Entwicklers/der Entwicklerin ab. Eine Übersicht über Spezialisierungsformen von Komponenten auf der Systemebene ist in Abb. 1.13 zu sehen. Diese unterscheiden sich im Wesentlichen in ihrer Performanz und Flexibilität. Am flexibelsten, aber leider auch oftmals am ungeeignetsten, z. B. aufgrund von Kosten, Leistungsverbrauch und Grad der Parallelisierung, sind Vielzweckprozessoren (engl. *general purpose processors*). Ein ASIC hingegen ist oft zu teuer, nicht flexibel genug oder bedarf einer zu hohen Entwicklungszeit. Alternativen stellen hierzu Spezialprozessoren dar. Die wohl bekanntesten Klassen an Spezialprozessoren sind *Digitale Signalprozessoren (DSPs)* und *Mikrocontroller*. In Richtung Hardware nimmt die Performanz zu, aber gleichzeitig die Flexibilität ab. Spezialisierte Komponenten an der Schnittstelle zwischen Hard- und Software sind sog. *Anwendungsspezifische Instruktionssatz-Prozessoren* (engl. *application specific instruction set processor, ASIPs*) und sog. *FPGAs* (engl. *field programmable gate arrays*), welche eine Nahtstelle bezüglich Flexibilität und Performanz zwischen Software und Hardware herstellen. Aus Kostengründen ist ein ASIP oft nur ein „abgespeckter“ Prozessor und damit günstiger als ein Vielzweckprozessor, aber aufgrund der (wenn auch beschränkten) Programmierbarkeit immer noch flexibler als dedizierte Hardware. SRAM-basierte FPGAs erlauben die Implementierung von Logik und Speicher durch Programmierung durch einen Bitstrom. FPGAs als Hardwarerealisierungsvariante besitzen damit auch die Flexibilität von Softwa-

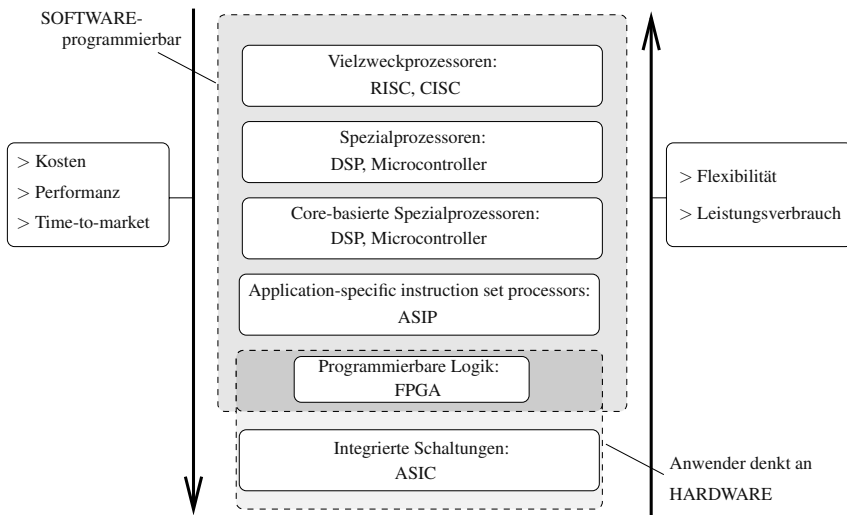


Abb. 1.13. Spezialisierungsformen und Kriterien für Hardware/Software-Entscheidungen

relösungen mit hoher Performanz. Jedoch ist die Performanz und die Auslastung der Ressourcen oft lange nicht so hoch wie bei ASICs.

Die Spezifikation wird anschließend in Hardware- und Softwarekomponenten aufgeteilt (Hardware/Software-*Partitionierung* in Abb. 1.12). Dabei wird die Entscheidung getroffen, welche Komponenten der zu implementierenden Funktionalität in Software verfeinert werden (um dann auf einem oder mehreren der allozierten Prozessoren ausgeführt zu werden) und welche Komponenten in Hardware verfeinert werden. Der Softwareanteil kann somit weiter in verschiedene Teile zerlegt werden, von denen jeder auf einem eigenen Prozessor abläuft, z. B. auf einem langsamen Prozessor für unkritische Systemteile und auf Spezialprozessoren für schnelle Datentransformationen. Für die in Hardware zu implementierenden Komponenten werden eine oder mehrere Schaltungen synthetisiert.

Da jede neue Allokation und jede neue Partitionierung eine mögliche Systemimplementierung erzeugt, erfordert ein Vergleich dieser Optionen die *Schätzung* von Systemeigenschaften. Jeder Satz von Schätzwerten wird anschließend mit den gegebenen Anforderungen verglichen und eine optimale Implementierung ausgewählt.

Nach dieser Auswahl muss die Spezifikation soweit verfeinert werden, dass sie die strukturellen Eigenschaften der Implementierung auf Systemebene charakterisiert. Speziell sind die verschiedenen Teilsysteme den allozierten Systemkomponenten zuzuordnen und die notwendigen Kommunikationskanäle und Protokolle zu spezifizieren (Hardwaresynthese, Softwaresynthese und Interfacesynthese).

Architektursynthese

Architektursynthese beschreibt den Verfeinerungsschritt im Entwurf eines Hardwaresystems, bei dem eine Verhaltensbeschreibung mit Aufgaben der Granularität von elementaren arithmetisch/logischen Operationen (z. B. Addition, Multiplikation) auf eine strukturelle Beschreibung eines *Operationswerks* (Datenpfad) und eines *Steuerwerks* (Kontrollpfad) abgebildet wird. Wesentliche Aufgaben sind (siehe Abschnitt 1.2.2)

- Identifikation von Hardwarekomponenten, die die spezifizierten Operationen ausführen können (*Allokation*),
- *Ablaufplanung* zur Bestimmung der Zeitpunkte, an denen die Operationen ausgeführt werden,
- Zuordnung von Variablen zu Speichern, Operationen zu funktionalen Einheiten und Kommunikationskanälen zu Bussen (*Bindung*).

Die makroskopischen Eigenschaften, wie z. B. Schaltungsfläche und Verarbeitungsleistung, hängen wesentlich von der Optimierung auf dieser Abstraktionsebene ab.

Beispiel 1.3.4. Das folgende Beispiel soll eine Schaltung modellieren, die eine Differentialgleichung der Form $y'' + 3xy' + 3y = 0$ im Intervall $[x_0, a]$ mit der Schrittweite dx und Anfangswerten $y(x_0) = y$, $y'(x_0) = u$ mit Hilfe der Euler-Methode numerisch löst. Dabei wird für ein Anfangswertproblem der Form $y' = f(x, y(x))$ mit $y(x_0) = y_0$ die Näherungsformel $y(x + dx) \approx y(x) + dx f(x, y(x))$ eingesetzt, um beginnend mit x_0 die Werte von $y(x_0 + i dx)$, $i = 1, 2, \dots$, sukzessiv zu bestimmen. Im Beispiel lautet die Differentialgleichung $y'' = f(x, y(x), y'(x)) = -3xy'(x) - 3y(x)$. Die zweifache Anwendung der Euler-Methode liefert hier die Lösung: $y'(x + dx) = y'(x) + dx (-3xy'(x) - 3y(x))$ und $y(x + dx) = y(x) + dx y'(x)$. Eine Verhaltensspezifikation in einer Systembeschreibungssprache (hier SystemC [144, 35, 188]) würde etwa folgendermaßen aussehen (x_0 entspricht dem Portsignal `x_in`, $y(x_0)$ entspricht `y_in`, $y'(x_0)$ entspricht `u_in`, dx entspricht `dx_in` und a entspricht `a_in`):

```
#include "systemc.h"

class dgl : sc_module {
public:
    sc_in<bool> activate;
    sc_in<double> x_in;
    sc_in<double> y_in;
    sc_in<double> u_in;
    sc_in<double> a_in;
    sc_in<double> dx_in;
    sc_out<double> y_out;

    SC_HAS_PROCESS(dgl);

    dgl(sc_module_name module_name) :
        sc_module(module_name) {
```

```

        SC_METHOD(algorithm);
        sensitive << activate;
    }

private:
    void algorithm() {
        double x = x_in;
        double y = y_in;
        double u = u_in;
        double a = a_in;
        double dx = dx_in;

        double x1, u1, y1;

        while( a <= x ) {
            x1 = x + dx;
            u1 = u - (3 * x * u * dx) - (3 * y * dx);
            y1 = y + (u * dx);
            x = x1; u = u1; y = y1;
        }
        y_out = y;
    }
};

```

Die Klasse `dgl` ist von `sc_module` abgeleitet und stellt eine Beschreibung der Ein- und Ausgänge für den Algorithmus `algorithm` dar. Eingangssignale sind `x_in`, `y_in`, `u_in`, `dx_in`, `a_in` sowie das Signal `activate`, das zum Starten der numerischen Integration dient. Das Signal `y_out` enthält die Lösung $y(a)$. Die Beschreibung des Verhaltens besteht im Wesentlichen aus einem Prozess `SC_METHOD`, der durch eine Änderung des Signals `activate` gestartet wird. Dieser intern sequentielle Prozess definiert die lokalen Variablen `x`, `y`, `u`, `dx`, `a`, `x1`, `y1`, `u1` und realisiert die eigentliche Iteration.

Nach der Architektursynthese könnte ein Blockschaltbild, wie in Abb. 1.14 gezeigt, entstehen: Das *Operationswerk* (Datenpfad) der Schaltung enthält als Ressourcen einen Multiplizierer und eine ALU (arithmetisch-logische Funktionseinheit), die Addition, Subtraktion und Vergleiche ausführen kann. Des Weiteren enthält die Architektur einen Speicher, eine Einheit zur Verteilung der Daten auf die Funktionsblöcke des Datenpfades sowie ein *Steuerwerk* (Kontrollpfad). Diese Sicht könnte auch durch eine strukturelle Beschreibung in einer entsprechenden Beschreibungssprache, z. B. VHDL, SystemVerilog, oder SystemC dargestellt werden.

Logiksynthese

Aufgabe der *Logiksynthese* ist die Generierung einer strukturellen Sicht auf Logikebene. Demzufolge bestimmt sie also die Struktur einer Schaltung auf Gatterebene. Ausgangspunkte der Logiksynthese können z. B. Boolesche Gleichungen oder endliche Zustandsautomaten sein, die entweder durch graphische Methoden oder mit

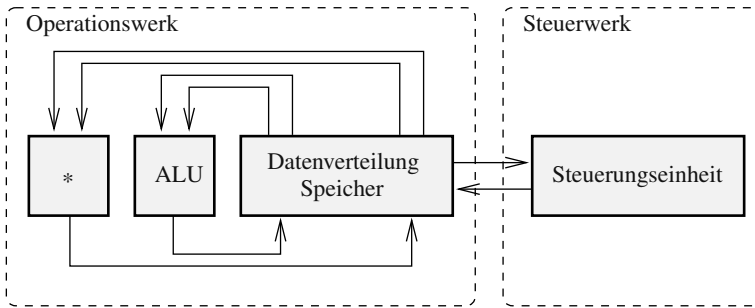


Abb. 1.14. Beispiel einer strukturellen Sicht auf Architekturebene

Hilfe eines Programms in einer Hardwarebeschreibungssprache oder SystemC spezifiziert wurden.

Unter anderem werden durch die Logiksynthese die folgenden Teilprobleme gelöst:

- Optimierung Boolescher Ausdrücke,
- Zustandsminimierung und Zustandszuordnung,
- Bindung an eine Bibliothek von Zellen, d. h., das logische Modell wird in eine Verbindung von Instanzen der Bibliothekszellen transformiert.

Optimierungsverfahren spielen auch hier eine zentrale Rolle, da die mikroskopischen Eigenschaften einer Implementierung festgelegt werden [258, 26]. Ergebnis der Logiksynthese ist eine strukturelle Repräsentation, die zum Beispiel Gatter, Register sowie ihre Verbindungen charakterisiert (Netzliste).

Beispiel 1.3.5. Das Steuerwerk in Abb. 1.14 hat die Aufgabe, die Operationen im Datenpfad sequentiell ablaufen zu lassen, indem die entsprechenden Steuerungssignale generiert werden. Dies ist ein typisches Beispiel, in dem eine Verhaltensbeschreibung in Form eines Zustandsdiagramms angebracht ist. Aufgabe der Logiksynthese ist es nun, eine Schaltung zu generieren, die diese Spezifikation implementiert. Als Beispiel der Sichten auf der Logikebene zeigt Abb. 1.15 das Zustandsdiagramm eines endlichen Zustandsautomaten, der zwei oder mehrere aufeinander folgende Einsen im Eingangsstrom einer Folge von Bits (Eingabe *in*) erkennt. Dargestellt sind eine Verhaltensbeschreibung in Form eines Zustandsdiagramms und eine Schaltungsrealisierung.

Auch diese Sichten lassen sich in einer Beschreibungssprache formulieren. Eine Verhaltensbeschreibung in SystemC, aus der die in Abb. 1.15 dargestellte Schaltung direkt synthetisiert werden kann, könnte wie folgt aussehen:

```
#include "systemc.h"

class rec : sc_module {
public:
    sc_in<bool> clk;
```

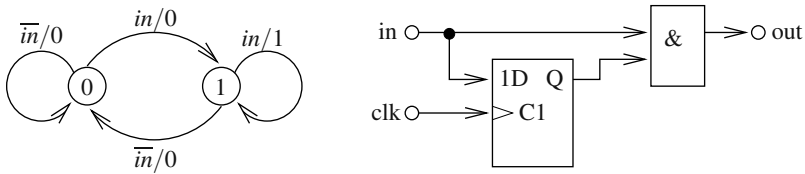


Abb. 1.15. Beispiel einer Verhaltenssicht (Zustandsdiagramm) und einer strukturellen Sicht auf Logikebene

```

sc_in<bool> in;
sc_out<bool> out;

SC_HAS_PROCESS(rec);

rec(sc_module_name module_name) :
    sc_module(module_name) {
    state.write(zero);
    SC_METHOD(behavior);
    sensitive_pos << clk;
}

private:
    enum state_type {zero, one};

    sc_signal<state_type> state;

    void behavior() {
        if (in.read() == 1) {
            switch (state.read()) {
                case zero:
                    state.write(one);
                    out.write(0);
                    break;
                case one:
                    state.write(one);
                    out.write(1);
                    break;
            }
        } else {
            state.write(zero);
            out.write(0);
        }
    }
};

```

In diesem Modell ist das Signal `state` vom Aufzählungstyp `state_type` und speichert den Zustand des endlichen Automaten. Der Prozess wird jedes mal dann

neu ausgeführt, wenn sich das Taktsignal `clk` auf den Wert '1' (`sensitive_pos << clk`) ändert.

Die Schaltungsstruktur lässt sich auch in SystemC modellieren:

```
#include "systemc.h"

class rec : sc_module {
public:
    sc_in<bool> clk;
    sc_in<bool> in;
    sc_out<bool> out;

    rec(sc_module_name module_name) :
        sc_module(module_name) {
        and1 = new and_component("and1");
        dff1 = new dff_component("dff1");
        and1->i1(in);
        and1->i2(tmp);
        and1->out(out);
        dff1->in(in);
        dff1->clk(clk),
        dff1->out(tmp);
    }

private:
    sc_signal<bool> tmp;
    and_component* and1;
    dff_component* dff1;
};
```

Abbildung 1.16 zeigt eine Schaltungsrepräsentation, die direkt dem vorangegangenen SystemC-Modell entspricht.

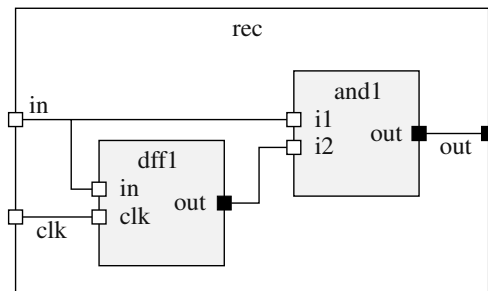


Abb. 1.16. Schaltungsdiagramm zu einem strukturellen SystemC-Modell auf Logikebene

Softwaresynthese

In Abb. 1.7 wurden auf der Seite der Software die Abstraktionsebenen „Modul“ und „Block“ unterschieden. Auf der Modulebene könnte eine Verhaltensbeschreibung z. B. in Form einer algebraischen Spezifikation vorliegen, die die Eigenschaften des zu entwickelnden Softwaresystems in Form „mathematischer Sätze und Axiome“ beschreibt oder als Netzwerk kommunizierender Prozesse bzw. Tasks. Aufgabe der Verfeinerung ist es nun, eine in gewissem Sinne äquivalente strukturelle Darstellung zu erzeugen, z. B. formuliert in einer höheren Programmiersprache (z. B. in C, C++ oder JAVA) oder einer echtzeitfähigen Sprache (z. B. ESTEREL) oder durch Abbildung auf Betriebssystemroutinen eines Echtzeitbetriebssystems (engl. *Real-Time Operating System*, RTOS).

Auf Modulebene lassen sich die wichtigsten Grundaufgaben wie folgt skizzieren:

- Ablaufplanung der verschiedenen Softwareprozesse auf (Echtzeit-)Betriebssystemebene,
- Einbindung von Betriebssystemroutinen, z. B. zur Interruptsteuerung und zur Ein- und Ausgabe von Daten.

In der nächst tieferen Blockebene wird nun hieraus durch einen Übersetzungsvorgang ein Assembler- oder Maschinenprogramm erzeugt. Die folgende Aufzählung fasst einige der wesentlichen Transformations- und Optimierungsvorgänge auf der Blockebene zusammen:

- Programmtransformationen zur optimalen Ausnutzung von Fließbandverarbeitung innerhalb von Daten- und Kontrollpfad des Zielprozessors,
- Optimierung des Speicherplatzbedarfs,
- Parallelisierung auf Instruktionsebene, um parallele funktionale Einheiten im Zielprozessor ausnutzen zu können,
- Festlegung der Ablaufplanung von Instruktionen bzw. der Reihenfolge der Abarbeitung der Instruktionen,
- Optimierung der Befehlsauswahl und Registervergabe.

Im Gegensatz zu einem Programm in einer höheren Programmiersprache ist ein Assemblerprogramm i. Allg. nicht nur erheblich länger, sondern auch abhängig von der jeweiligen Zielarchitektur. Ferner fehlen Möglichkeiten zur Typüberprüfung und zum strukturierten Kontrollfluss.

Beispiel 1.3.6. Als Beispiel für die beiden Sichten auf der Blockebene wird als Verhaltensbeschreibung ein C++-Programm betrachtet, das $\sum_{i=0}^{100} i^2$ berechnet.¹

```
#include <iostream>

int main(int argc, char *argv[])
{
```

¹ Es wird im Folgenden angenommen, dass der Datentyp `int` mit einer Genauigkeit von 32 Bits dargestellt wird.

```

int sum = 0;
for (int i = 0; i <= 100; i++)
    sum += i*i;
std::cout << "The sum of i*i from 0 ... 100 is "
           << sum << std::endl;
}

```

Nach der Übersetzung für den RISC-Prozessor MIPS R2000 könnte das folgende Assemblerprogramm entstehen:

```

...
main:
    subu    $29, $sp, 32
    sw      $31, 20($29)
    sd      $4, 32($29)
    sw      $0, 24($29)
    sw      $0, 28($29)
loop:
    lw      $14, 28($29)
    mul     $15, $14, $14
    lw      $24, 24($29)
    addu    $25, $24($29)
    ...

```

Dieses Assemblerprogramm hat insgesamt 29 Zeilen und muss anschließend noch in ein binäres Maschinenprogramm umgesetzt werden.

1.3.3 Optimierung

Optimierung ist ein entscheidender Gesichtspunkt von Entwurfsverfahren auf allen Abstraktionsebenen. Die unterschiedlichen strukturellen Implementierungen eines Systems definieren seinen *Entwurfsraum*. Der Entwurfsraum ist somit eine endliche Menge von *Entwurfspunkten*. Mit jedem dieser Entwürfe sind Werte der Zielfunktionen verbunden, z. B. Kosten und Verarbeitungsleistung.

Aufgabe der Optimierung ist es daher, den „besten“ Entwurf zu finden, d. h. diejenige Implementierung, die unter mehreren Optimierungszielen optimal ist. Da das Optimierungsproblem aber mehrere verschiedene Kriterien beinhaltet, sollte man sich die Definition eines Optimums etwas genauer ansehen.

Ein Entwurfspunkt heißt *nichtdominiert*, wenn es im Entwurfsraum keinen anderen Punkt gibt, der in allen betrachteten Optimierungskriterien mindestens gleich gut ist. Optimale Entwurfspunkte zeichnen sich nun dadurch aus, dass sie von keinem anderen Punkt des Entwurfsraums dominiert werden. Man nennt diese Menge nichtdominierter Entwurfspunkte *Pareto-Punkte* [50, 304]. Diese Definition optimaler Lösungen entspricht einer Erweiterung des Begriffs des globalen Optimums bei monodimensionalen Optimierungsproblemen.

Beispiel 1.3.7. Das Beispiel der Implementierung eines Video-Codierers wird hier fortgesetzt. Als mögliche Kriterien (unter vielen anderen!) für eine Exploration des

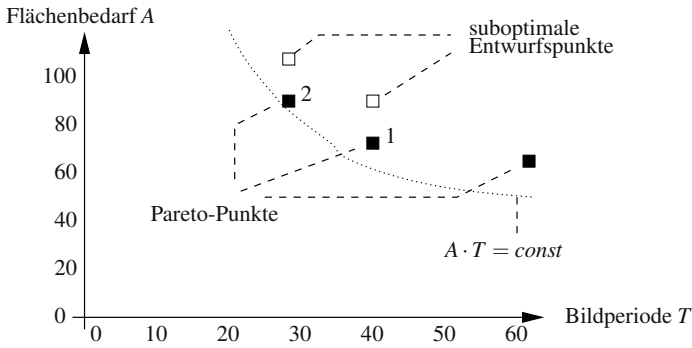


Abb. 1.17. Beispiel eines Entwurfsraums mit Pareto-Punkten; Implementierungen 1 und 2 entsprechen den Strukturen von Abb. 1.10 bzw. Abb. 1.11

Entwurfsraums kann man die Chipfläche bei einer Implementierung als integrierte Schaltung (Ein-Chip-Realisierung) betrachten sowie die Verarbeitungsrate (Bildperiode T). In den Flächenbedarf A gehen nicht nur die Zahl und Art der Teilsysteme ein (allgemein programmierbare Prozessoren, Spezialbausteine), sondern auch die Größe und Art der Speicher (Bildspeicher, schnelle lokale Cache-Speicher) sowie die Organisation und Breite der Busse.

Die Komplexität der Synthesaufgabe wird deutlich, wenn man bedenkt, dass für jede betrachtete und in Bezug auf Busbandbreite, Signaldarstellung und Speichergröße parametrisierte Architektur eine für die Kriterien jeweils optimale Ablaufplanung und Bindung bestimmt werden muss.

In Abb. 1.17 wird ein kleiner Ausschnitt des Entwurfsraums gezeigt, der lediglich durch die normierten Kriterien „Flächenbedarf“ und „Verarbeitungszeit pro Bild“ parametrisiert wird. Den Vorgang der Suche bzw. Ermittlung der Pareto-Menge oder einer möglichst guten Approximation dieser Menge bezeichnet man auch als *Entwurfsraumexploration*. Es wird also deutlich, dass im Sinne der gewählten Kriterien jeder Pareto-Punkt eine optimale Lösung darstellt. Durch eine an die Entwurfsraumexploration anschließende benutzergersteuerte Abwägung der unterschiedlichen Kriterien kann dann der Entwickler/die Entwicklerin eine der gefundenen Pareto-optimalen Lösungen auswählen und implementieren.



<http://www.springer.com/978-3-540-46822-6>

Digitale Hardware/Software-Systeme

Synthese und Optimierung

Teich, J.; Haubelt, C.

2007, XV, 594 S., Softcover

ISBN: 978-3-540-46822-6