Eingebettete Systeme

Materialien zum Kapitel Architekturentwurf

Lothar Thiele, Jürgen Teich[1]

ETH Zürich
Institut für Technische Informatik und Kommunikationsnetze
Fachgruppe Technische Informatik

©bei den Autoren

---

[1] Universität Erlangen Nürnberg, Lehrstuhl für Informatik

Vorwort

Begründung

Inhalt der Vorlesung sind Modelle und Methoden zum Entwurf komplexer Systeme. Die Komplexität, so wie sie hier verstanden wird, entsteht *nicht nur* durch die Zahl der Einzelkomponenten, aus denen das 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. Erfolgreiche Produkte in dieser Hinsicht sind in zahlreichen Bereichen zu finden, etwa der Verkehrstechnik mit ABS und Airbag. Hier zum Beispiel werden die wesentlichen Eigenschaften solcher Systeme deutlich: Heterogenität (analog–digital, mechanisch–elektronisch, Hardware–Software, Sensorik, Aktorik, Mikrosystemtechnik), hohe Verfügbarkeit, Selbstdiagnose (Anzeige bei Ausfall des ABS), Fehlertoleranz.

Selbstverständlich können im Rahmen der Vorlesung nicht beliebige komplexe technische Systeme betrachtet werden. Wir werden uns hier beschränken auf Systeme, die aus miteinander verbundenen integrierten Schaltungen bestehen. Beispiele sind die sogenannten eingebetteten Systeme, siehe Abb. 1. Sie sind dazu bestimmt, Funktionen als Antwort auf bestimmte Stimuli auszuführen und fallen somit in die Klasse reaktiver Systeme. Es handelt sich dabei meist um zeitkritische Anwendungen, bei denen die Antwort innerhalb bestimmter Zeitschranken erforderlich ist (Echtzeitsysteme). Echtzeit-Steuerungssysteme müssen also kritische Informationen zwischen autonomen Teilsystemen unter Einhaltung von Zeitschranken zuverlässig austauschen. Diese Anforderungen werden zum Beispiel offensichtlich in Zusammenhang mit (öffentlichen) Transportsystemen wie Flugzeug, Bahn oder auch Automobil. Man sollte sich zum Beispiel vorstellen, dass jeweils autonome Sensor-Elektronik-Aktor-Systeme verantwortlich sind für Bremsfunktion, Motorsteuerung, Federungssysteme, Kupplung, Getriebe und vieles andere mehr. Hier werden bereits analoge und digitale elektronische Hardware-Software-Systeme mit mechanischen Komponenten verbunden. Alle Teilsysteme stehen zusätzlich untereinander über ein Kommunikationsnetzwerk miteinander in Verbindung. Andere Beispiele dieser Art finden sich in der Haustechnik, bei verteilten Produktionssystemen (verteilte Sensor-Aktor-Systeme, siehe oben) sowie im Telekommunikationsbereich (mobile Endgeräte, Kommunikationsnetze).

Das grosse Interesse am systematischen Entwurf von heterogenen Systemen, die mechanisch-elektronische, analog-digitale sowie Hardware-Software-Komponenten enthalten, ist massgeblich verursacht durch

- die steigende Vielfalt und Komplexität von Anwendungen für eingebettete Systeme,
- die Notwendigkeit, Entwurfs- und Testkosten zu senken, sowie durch
- Fortschritte in Schlüsseltechnologien (Mikroelektronik, formale Methoden).

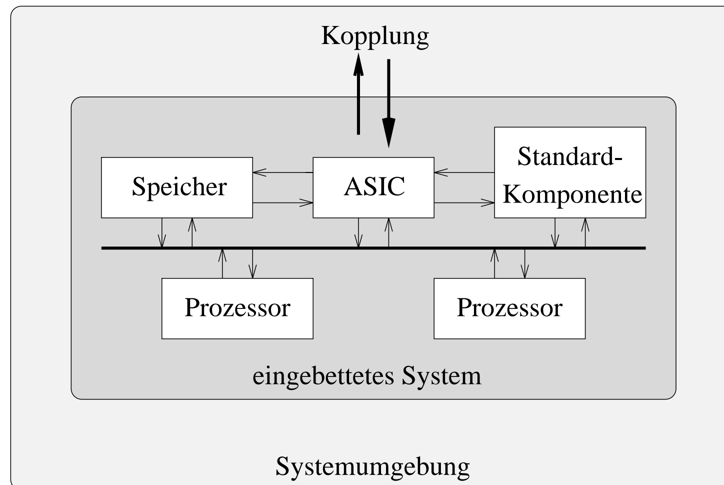


Abbildung 1: Schematische Darstellung eines eingebetteten Systems

Um die Entwicklung in diesem Bereich genauer zu verstehen, ist es sinnvoll, sich die historische Entwicklung kurz vor Augen zu führen.

Als Beispiel eines Systems, das seine Komplexität vor allem durch die riesige Anzahl seiner Komponenten erhält, können integrierte Schaltungen dienen: Eine integrierte Schaltung kann aus vielen Millionen Einzelelementen bestehen, die jedoch alle in ähnlicher Weise arbeiten und in einem homogenen technologischen Prozess (aus Silizium) hergestellt werden. Die Forschritte in der Technologie hätten alleine jedoch nicht zu dem riesigen Erfolg in diesem Bereich geführt. Entscheidenden Einfluss hatte die Entwicklung rechnergestützter Entwurfsverfahren, mit deren Hilfe wesentliche Schritte automatisiert werden. Heutige integrierte Schaltungen könnten nicht mehr ohne den Einsatz von CAD (computer-aided-design)-Methoden entworfen werden. Insgesamt lässt sich diese Entwicklung in drei Abschnitte unterteilen:

- Nachdem die Zahl der Objekte in den unteren Entwurfsebenen (Geometrie, physikalische Ebene) aufgrund des Zeitaufwandes und der Fehleranfälligkeit ohne Automatisierung nicht mehr handhabbar war, wurden in Industrie und Forschung Modelle und Methoden entwickelt, die zum Beispiel auf Schaltungssimulation, Platzierung und Verdrahtung abzielten.

- In einem weiteren Schritt wurden dann auch höhere Abstraktionsebenen in die Automatisierung einbezogen, wie zum Beispiel die Simulation von Schaltungen auf Logik-Ebene oder auch die Logiksynthese.

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

Eine Betrachtung auf Systemebene ist nicht nur näher an der üblichen Vorstellung eines Systementwicklers. Es wäre zudem kaum vorstellbar, wie ein System, das aus mehreren verteilten Mikroprozessoren einschliesslich der zugehörigen Software sowie aus integrierten Schaltungen mit jeweils mehr als 100.000 Gattern besteht, auf andere Weise

spezifiziert, entworfen, getestet und dokumentiert werden kann. Je komplexer ein Entwurf ist, desto schwieriger wird es, die Funktionalität aus Assembler-Programmen und Logik-Diagrammen auf Gatter-Ebene zu verstehen. Auf der anderen Seite, wenn das System als eine Folge von komplexen Operationen beschrieben wird, die auf abstrakten Datentypen operieren und ihre Ergebnisse über abstrakte Kanäle kommunizieren, so wird es dem Entwickler leichter fallen, die korrekte Funktionalität des Entwurfs zu testen oder zu verifizieren und verschiedene Realisierungsalternativen zu evaluieren.

Zusammenfassend haben die folgenden Kernpunkte entscheidend zu den riesigen Fortschritten auf dem Gebiet der Entwurfsautomatisierung beigetragen:

- Modellierung und formale Spezifikation,
- Hierarchie und Abstraktion,
- Effiziente Synthesealgorithmen.

Sie werden auch im Vordergrund der Vorlesung stehen.

Die Vorlesung wird selbstverständlich nicht alle Aspekte des Systementwurfs abdecken können. Das liegt unter anderem daran, dass das gesamte Gebiet noch stark in Bewegung ist. Aus Zeitgründen werden wir uns speziell nicht mit formaler Verifikation und Simulationsverfahren auseinandersetzen.

Die Materialien

Die Vorlesung gliedert sich in Aspekte, die sich auf die Architektur und Hardware eingebetteter Systeme beziehen und solche, die vor allem die Software betreffen. Einen guten Überblick über den gesamten Bereich der eingebetteten Systeme geben die Bücher von Wayne Wolf [1] und Peter Marwedel [2].

Architektur, Entwurfsmethodik und Architektursynthese

Die vorliegenden Materialien wurden zur Begleitung der Vorlesung im Bereich der Systemarchitektur, Entwurfsmethodik und Hardwaresynthese erstellt.

- Das Skript stellt nur eine Form der Unterlagen für diese Vorlesung dar. Weiterhin werden Kopien der Vorlesungsfolien oder auch ausgewählte Publikationen zur Verfügung gestellt.
- Das Skript soll als Referenz über einige Themenbereiche dienen, die in der Vorlesung behandelt werden. Wir stellen an uns nicht den Anspruch, ein Buch vorzulegen.
- Das Skript befindet sich in einem dynamischen Zustand. Daher werden wesentliche Teile im Verlauf der Vorlesung ergänzt und an die Hörer verteilt.

In einigen Teilen bezieht sich dieser Teil der Vorlesung auf die Bücher von Giovanni De Micheli [3] und Daniel Gajski et al. [4]. Weiterhin wurden wesentlich Teile dieser Vorlesung von Jürgen Teich in sein Buch [5] übernommen.

Weitere Literaturhinweise findet man im jeweiligen Zusammenhang.

Einbettung

Vielleicht stellt sich die Frage, wie denn die Vorlesung in das bestehende Angebot (speziell des Instituts für Technische Informatik und Kommunikationsnetze) einzuordnen ist. Diese Frage drängt sich vor allem deshalb auf, da sich ja schon in verschiedenen anderen Veranstaltungen mit eingebetteten Systemen auseinandergesetzt wird.

- In den Veranstaltungen „Technische Informatik I/II“ werden die wesentlichen Grundlagen zum Verständnis von Hardware- und Softwarearchitekturen gelegt. Sie sind daher Voraussetzung für diese Lehrveranstaltung.
- Die Kernfachvorlesung „Diskrete Ereignissysteme“ legt theoretische Grundlagen zum Verständnis von Modellen und Methoden, die bei eingebetteten Systemen verwendet werden. Einige dieser Grundbegriffe werden im Rahmen dieser Vorlesung wiederholt. Diese Kernfachvorlesung ist demzufolge keine unbedingte Voraussetzung.

Inhaltsverzeichnis

1 Architekturen für eingebettete Systeme 7
1.1 Grundstrukturen von eingebetteten Systemen . . . 7
1.2 Implementierungstypen . . . 7
1.2.1 Ein-Chip-Lösungen . . . 7
1.2.2 Board-Level-Systeme . . . 12
1.3 Prozessoren . . . 13
1.3.1 Mikrocontroller . . . 13
1.3.2 Mikroprozessoren . . . 16
1.3.3 Anwendungsspezifische Prozessoren (ASIP) . . . 19
1.3.4 Digitale Signalprozessoren – DSPs . . . 21
1.4 Hardwareimplementierungen . . . 24
1.4.1 ASIC-Entwurf . . . 24
1.4.2 Programmierbare Hardware . . . 27

2 Entwurfsmethodik 31
2.1 Entwurfsmethodik . . . 31
2.1.1 Erfassen und Simulieren . . . 31
2.1.2 Beschreiben und Synthetisieren . . . 31
2.1.3 Spezifizieren, Explorieren und Verfeinern . . . 33
2.2 Abstraktion und Entwurfsrepräsentationen . . . 34
2.2.1 Modelle . . . 34
2.2.2 Synthese . . . 35
2.2.3 Optimierung . . . 44

3 Spezifikation und Modellierung 47
3.1 Einleitung . . . 47
3.2 Klassifikation von Modellen . . . 49
3.3 Petri-Netz-Modell . . . 50
3.3.1 Dynamische Eigenschaften von Petri-Netzen . . . 52
3.4 Zustandsorientierte Modelle . . . 55
3.4.1 Endliche Automaten (FSM) . . . 55
3.4.2 Erweiterte Zustandsmaschinenmodelle . . . 56
3.4.3 Hierarchische, nebenläufige Zustandsmaschinen . . . 57
3.4.4 Statecharts . . . 57
3.5 Aktivitätsorientierte Modelle . . . 62
3.5.1 Datenflussgraphen . . . 62
3.5.2 Markierte Graphen . . . 62
3.5.3 Synchrone Datenflussgraphen (SDF) . . . 64

3.6 Strukturorientierte Modelle . . . 67
3.6.1 Komponenten-Verbindungsdiagramm (CCD) . . . 67
3.7 Heterogene Modelle . . . 69
3.7.1 Kontroll/Datenflussgraphen (CDFGs) . . . 69
3.8 Können Programmiersprachen mehr? . . . 73

4 Synthese **75**
4.1 Fundamentale Syntheseprobleme . . . 75
4.1.1 Ablaufplanung . . . 76
4.1.2 Allokation . . . 76
4.1.3 Bindung . . . 77
4.2 Algorithmen zur Ablaufplanung . . . 81
4.2.1 Ablaufplanung ohne Ressourcebeschränkungen . . . 81
4.2.2 Ablaufplanung mit Zeitbeschränkungen . . . 84
4.2.3 Ablaufplanung mit Ressourcenbeschränkungen . . . 86
4.2.4 Iterative Verfahren zur Ablaufplanung . . . 104

5 Beispiele zur Architektursynthese **107**
5.1 Architekturbewertungen . . . 107
5.1.1 Verarbeitungsgeschwindigkeit . . . 107
5.1.2 Auslastung von Ressourcen . . . 107
5.1.3 Auslastung von Architekturen . . . 108
5.1.4 Fliessbandtiefe . . . 108
5.2 Architekturentwurf für ein digitales Filter . . . 109
5.3 Speicher . . . 111
5.3.1 Lebenszeit von Variablen . . . 112
5.3.2 Modellierung des Speicherbedarfs . . . 113
5.4 Anwendung der Verfahren auf einen Videokodierer . . . 114
5.4.1 Eine einfache Zielarchitektur . . . 116
5.4.2 Ein realistisches Architekturbeispiel . . . 122

Literatur **127**

Kapitel 1

Architekturen für eingebettete Systeme

Es folgt eine Übersicht über die Grundstrukturen von eingebetteten Systemen. Zum einen werden unterschiedliche *Implementierungsarten* beschrieben. Hierzu gehören Ein-Chip-Systeme, Multi-Chip-Module und Boardsysteme. Was die softwareprogrammierbaren Komponenten angeht, werden verschiedene Prozessortypen klassifiziert. Gerade im Bereich von eingebetteten Systemen spielen Spezialprozessoren eine dominante Rolle. Dies sind zum Beispiel *Mikrocontroller* und *Digitale Signalprozessoren (DSPs).* Wenn Prozessoren für einen einzelnen Anwendungsbereich zugeschnitten sind, spricht man auch von sog. ASIPs (engl. *Application-Specific Instruction set Processors*).

Was die Realisierung von Hardware angeht, unterscheidet man auch verschiedene Implementierungsarten, wobei zum einen der *ASIC-Entwurf* (voll- bzw. halbkundenspezifisch) und zum anderen das Prototypisieren auf *konfigurierbarer Hardware* gehört. Abb. 1.1 gibt einen Überblick über die einzelnen Realisierungsformen zur Implementierung eines Problems.

1.1 Grundstrukturen von eingebetteten Systemen

Abbildung 1.2 zeigt eine Übersicht über den Aufbau eines typischen eingebetteten Systems, bestehend aus Sensoren, Interfaces, dem digitalen System und Aktoren.

In Tabelle 1.1 sind einige Beispiele dargestellt.

1.2 Implementierungstypen

1.2.1 Ein-Chip-Lösungen

Im Folgenden sollen die Gründe für die Realisierung eines eingebetteten Systems als Ein-Chip-Lösung motiviert werden.

- *Kosten:* Im Allgemeinen gilt die Regel, dass Ein-Chip-Systeme aufgrund hoher Entwicklungskosten nur in hohen Stückzahlen rentabel sind, sonst teurer als beispielsweise Platinenentwürfe.

- *Gewicht, Grösse:* Gewisse Anwendungsbereiche stellen Anforderungen an maximales Gewicht, maximale Grösse, insb. der Bereich mobiler Geräte (Laptops, Chipcards,

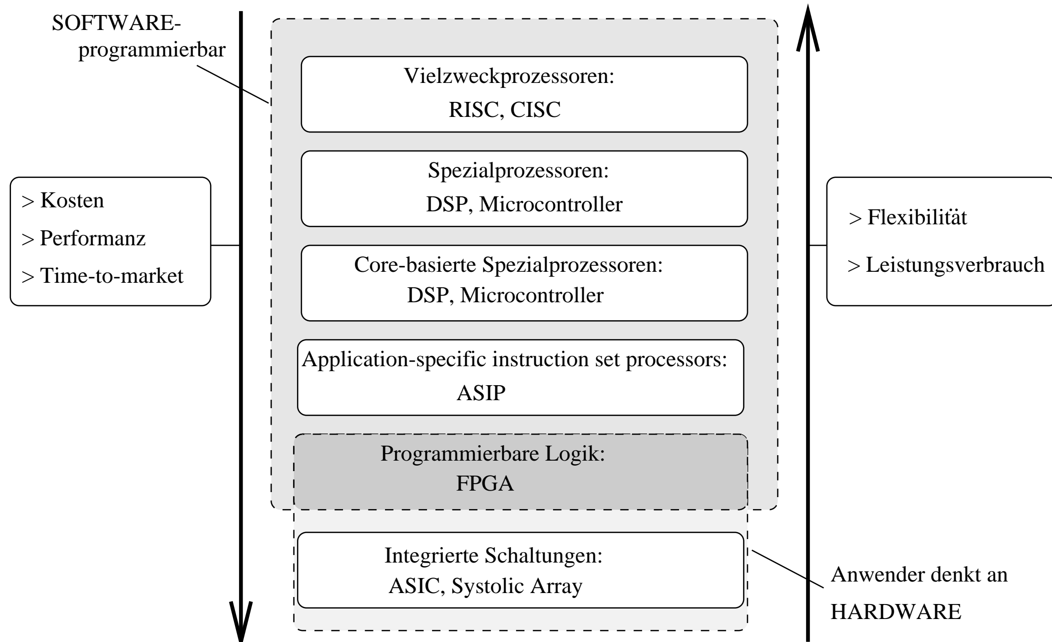


Abbildung 1.1: Spezialisierungsformen und Kriterien für Hardware/Software-Entscheidungen

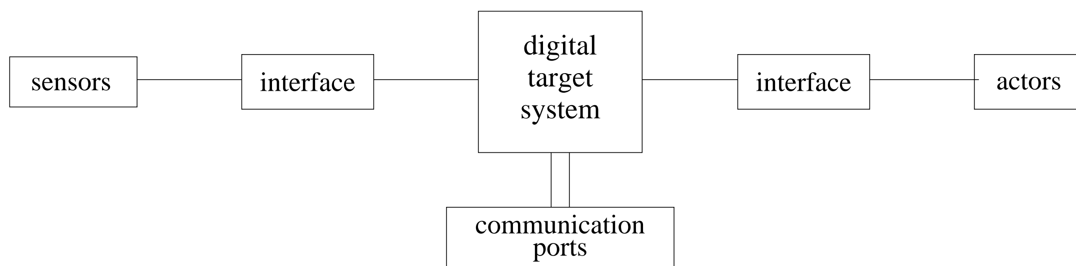


Abbildung 1.2: Grundaufbau eines typischen eingebetteten Systems

Beispiel/ Sensoren	Interface	Komm.ports	Interface	Aktoren
PC Maus, Tastatur	Ser./Par.- Wandler	Ethernet-Transc. DMA	D/A-Wandler Video-Signal-Generator	Bildschirm-steuerung
Laserdrucker Hitzesensoren ser. Schnittstelle Tastenfeld, Schalter	A/D-Wandler DMA parallel	Ethernet-Transc. par./ser.	D/A-Wandler Pulsformer (PWM)	Leistungs-elektronik, Motoren
Autosteuerung Motordrehzahl Position Kurbel-welle etc.	A/D-Wandler Komparatoren	CAN-Bus Transc. parallel I/O	PWM D/A-Wandler	Zündung Einspritzung Gangschalt.

Tabelle 1.1: Typische Struktur von eingebetteten Systemen

Mobiltelefone).

- *Verlässlichkeit:* Ein-Chip-Systeme sind im Allgemeinen verlässlicher als Platinenentwürfe, da Konnektoren, Kabel, Anschlüsse besonders kurz sind. Auch die Störanfälligkeit und elektromagnetische Verträglichkeit (EMV) sind höher aufgrund kleinerer Dimensionen.
- *Leistungsverbrauch:* gleiche Argumente wie für Grösse und Gewicht
- *Schutz intellektuellen Besitzes*

Beispiel 1.1 *Abbildung 1.3 zeigt das Layout eines konfigurierbaren Ein-Chip-Systems von Texas Instruments (TI-cDSP). Neben einem Prozessorkern und einem digitalen Signalprozessor enthält der Chip konfigurierbaren RAM- und ROM-Bereich sowie ein Gatearray (links) zum Entwurf anwendungsspezifischer Hardware. Häufig kommen zu diesen Komponenten auch periphere Komponenten dazu, z. B. eine A/D-Wandler-Zelle oder Schnittstellenzellen (seriell/parallel).*

Beispiel 1.2 *Neben der Entscheidung „Hardware oder Software“ betrifft ein weiterer Heterogenitätsaspekt eingebetteter Systeme die Frage analog oder digital. Abbildung 1.4 zeigt ein Ein-Chip-System, das einen SMARTPOWER Mikrocontroller mit einer 3 A DC-Motorbrücke koppelt. Die vier in der rechten Hälfte dargestellten regelmässigen Zellen des Layouts stellen DMOS-Leistungstransistoren dar. Neben dem Mikrocontroller zur Steuerung der Motorbrücke existiert ein EPROM zur Programmierung von Funktionen wie beispielsweise der Einstellung von Spannungsreferenzen und Verstärkereinstellungen.*

Beispiel 1.3 *Tabelle 1.2 zeigt unterschiedliche, vom Halbleiterhersteller Philips stammende Konfigurationen von Ein-Chip-Systemen, die den Prozessor 8051 verwenden.*

Eine typische Konfiguration eines Ein-Chip-Systems mit einem 8051-Prozessor ist in Abb. 1.5 dargestellt.

Zusammenfassung Ein-Chip-Lösungen:

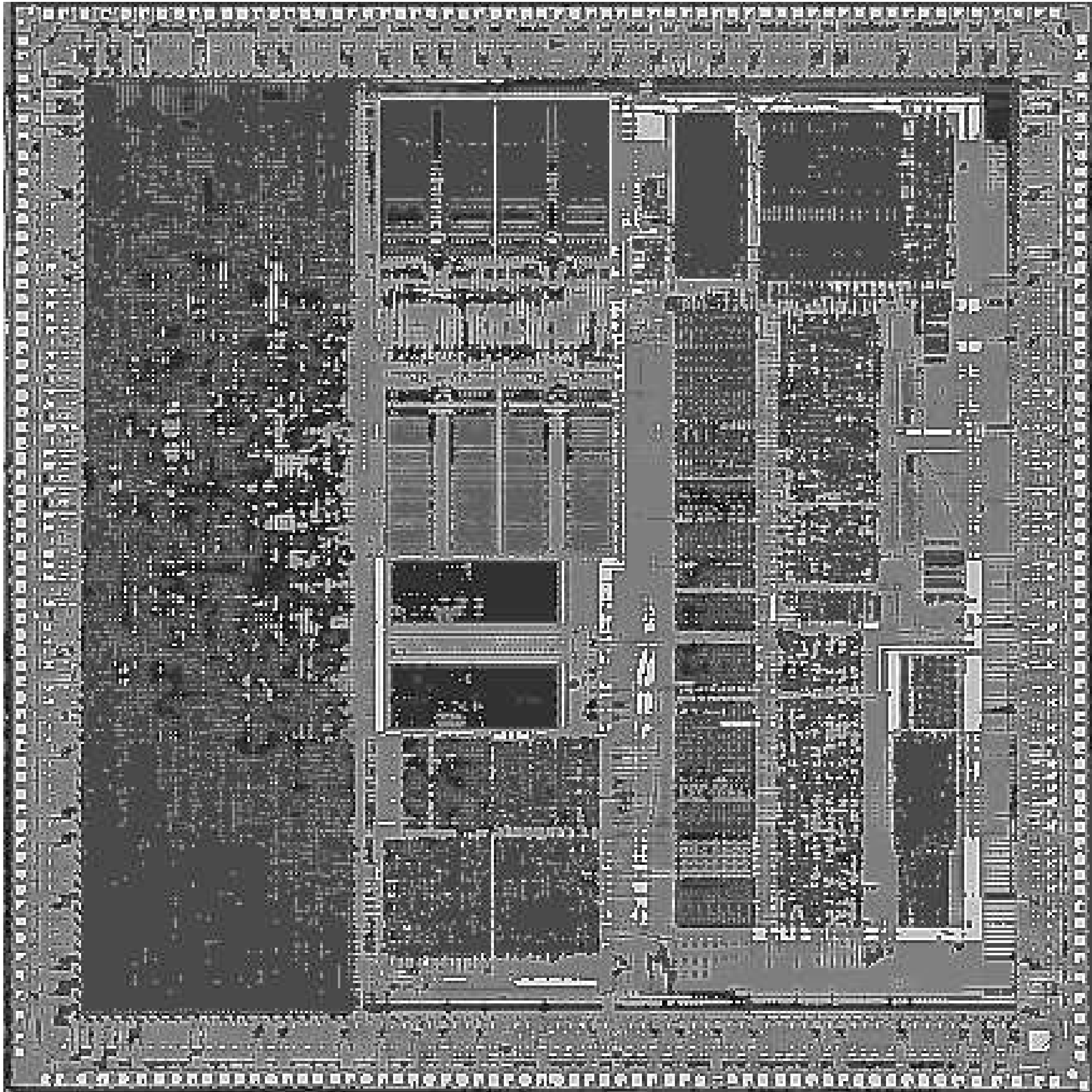


Abbildung 1.3: Layout eines Ein-Chip-Systems von Texas Instruments (cDSP)

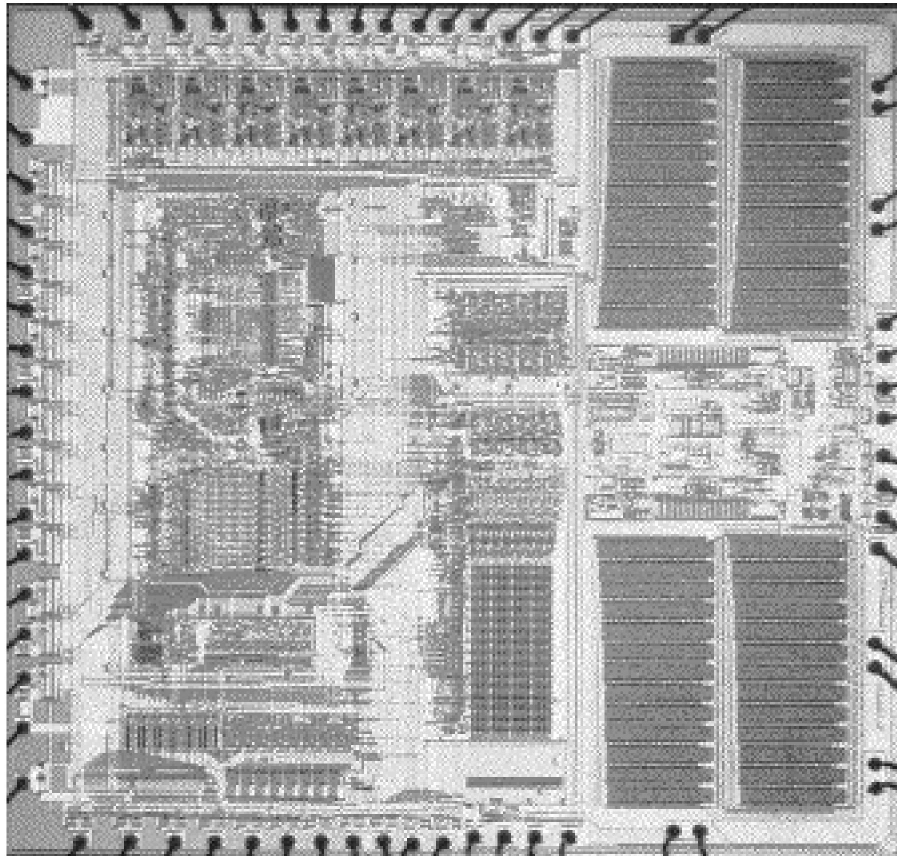


Abbildung 1.4: Layout des Ein-Chip-Systems SMARTPOWER

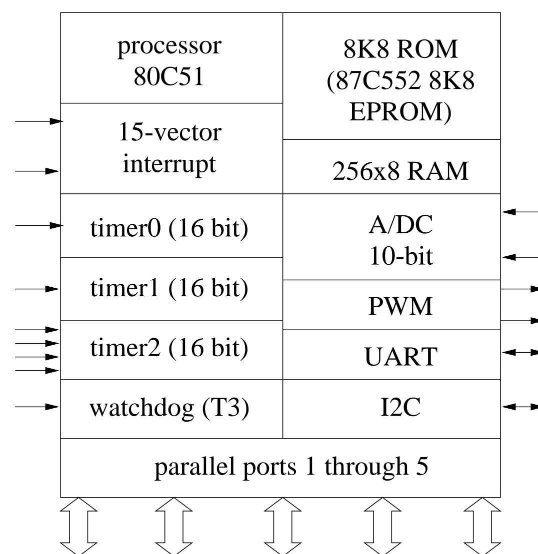


Abbildung 1.5: Philips 83 C552: 8-Bit-basierter Mikrocontroller

Eigenschaft	83/87 C552	83/87 C562	83/87 C652	83/87 C662	83/87 C654	83/87 C528	83 C851	83/87 C592
Rom EPROM	8K8	8K8	8K8	8K8	16K8	32K8	4K8	8K8
Ram [Bytes]	256	256	256	256	256	256	128	256
EEPROM [Bytes]							256	
I/O parallel	6x8	6x8	4x8	4x8	4x8	4x8	4x8	6x8
I/O UART	x	x	x	x	x	x	x	x
I/O I^2C	x		x		x			CAN
Timer 0,1	x	x	x	x	x	x	x	x
Timer 2,3	x	x						x
ADC 8-Eing.	10 Bit	8 Bit						10 Bit
PWM 2-Ausg.	x	x						x
Int.vektoren	15	14	6	5	6	6	5	15
Gehäuse								
Dil40			x	x	x	x	x	
PLCC-44			x	x	x	x	x	
QFP-44			x	x	x	x	x	
PLCC-68	x	x						x

Tabelle 1.2: Konfigurationen des 8051. Quelle: Philips Halbleiter

- Prozessorhersteller bieten Makrozellen bekannter Prozessortypen (sog. *Cores*) an, die mit unterschiedlichsten anderen Komponenten (z. B. Speicher) in Art und Grösse beliebig konfiguriert werden können.
- Cores erleichtern die Migration zwischen Versionen und Halbleiterherstellern und senken damit das Entwurfsrisiko.
- Neben Speicher gibt es zahlreiche andere periphere Einheiten.
- Unterstützt werden häufig spezielle Powerdown-Funktionen.
- Die Architekturen verzichten häufig auf Caching.
- Der Datenspeicher ist klein gegenüber dem Programmspeicher.
- Baukastensysteme unterstützen die Wiederverwendbarkeit von Komponenten.

1.2.2 Board-Level-Systeme

Abbildung 1.6 zeigt den typischen Aufbau eines Platinenentwurfs. Als Gründe für den Platinenentwurf lassen sich folgende Kriterien aufführen:

- Vorteile gegenüber der Ein-Chip-Lösung:
  - *Erfüllbarkeit:* Das System passt nicht auf einen einzelnen Chip.
  - *Kosten:* Die Benutzung von Standardchips ist kostengünstiger.
  - *Entwurfszeit:* häufig kleiner als für einen ASIC (1 Tag bis 1 Woche), da die Benutzung von existierenden Standardkomponenten möglich ist.

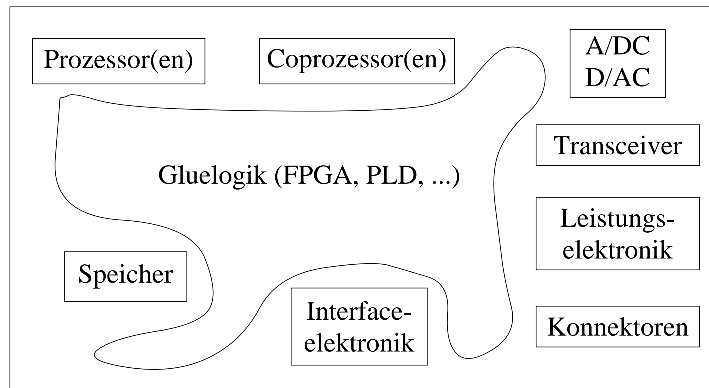


Abbildung 1.6: Typischer Aufbau eines Platinenentwurfs eines eingebetteten Systems

  - Höhere *Flexibilität* bei späten Änderungen.
  - *Verlässlichkeit:* Ein-Chip-Entwürfe sind technologiebedingt anfällig gegen eine Reihe von Defekten (Oxidation, heisse Ladungsträger), Beschädigung des Gehäuses. Hochsichere Systeme sind typischerweise verteilt.

- Vorteile gegenüber Lösungen bestehend aus mehreren Platinen:
  - *Performanz:* höhere Kommunikationsbandbreite möglich,
  - *Kosten:* niedriger,
  - *Grösse, Gewicht, EMV, Leistungsverbrauch*: vgl. Ein-Chip-Lösung.

Für Multi-Chip-Module sprechen Argumente von sowohl Platinen- als auch Ein-Chip-Systemen.

Für Systeme, die aus mehreren Platinen bestehen (*Multiboard-Systeme*), sprechen schliesslich Gründe der *Erfüllbarkeit* (z. B. wenn das System nicht auf eine Platine passt) sowie Gründe der *Flexibilität*, *Skalierbarkeit*, *Wartbarkeit* und *Erweiterbarkeit*. Nachteile sind hier vor allem die komplexere Kommunikationsstruktur der Teilsysteme untereinander und die dadurch häufig bedingte Einbusse in der Kommunikationsrate zwischen Teilsystemen.

1.3 Prozessoren

Tabelle 1.3 gibt eine Übersicht über die Marktanteile von Mikrocontrollern und Mikroprozessoren unterschiedlicher Leistungsfähigkeiten von 1993 und eine Vorhersage für 1998.

1.3.1 Mikrocontroller

Mikrocontroller kennzeichnen eine Klasse von Prozessoren, die für den speziellen Anwendungsbereich der *Steuerung* von Prozessen zugeschnitten sind.

Die klassischen Mikrocontroller besitzen eine Wortbreite von 4-8 Bit, neuere Generationen mittlerweile Wortbreiten von 16-64 Bit. Erstere sind optimiert für Anwendungen mit steuerungsorientierten Systemfunktionen. Die Eigenschaften lassen sich wie folgt charakterisieren:

	1993	1998 (FCST)
Mikrocontroller	**44 % ($6.8B)**	**38 % ($13.1B)**
4-Bit	11 % ($1.8B)	5 % ($1.7B)
8-Bit	25 % ($3.8B)	18 % ($6.4B)
16- und 32-Bit	3 % ($0.5B)	7 % ($2.3B)
DSP	5 % ($0.7B)	8 % ($2.7B)
Mikroprozessoren	**56 % ($8.7B)**	**62 % ($21.9B)**
8-Bit und 16-Bit	5 % ($0.7B)	1 % ($0.3B)
32-Bit und 64-Bit	51.6 % ($8.0B)	
32-Bit		33 % ($11.6B)
64-Bit		28 % ($10.0B)
	total $15.5B	total $35.2B

Tabelle 1.3: Markt von Mikrocontrollern und Mikroprozessoren. (Quelle: ICE, 94)

- Integration von Zugriffsfunktionen auf Peripherie (I/O) im Instruktionssatz.
- Bit- und Logikoperationen.
- Register sind häufig im RAM realisiert. Dadurch kann ein Kontextwechsel durch einfache Zeigeranweisungen bewerkstelligt werden. Diese Prozessoren erreichen *Interruptlatenzen* im Bereich von 1μs und genauso schnelle Kontextwechselzeiten.
- Geringe Performanz (ca. 1μs/Instruktion).

Beispiel 1.4 *Der Prozessor 8051 ist einer der in steuerungsdominanten Anwendungen am häufigsten eingesetzten Mikocontroller. Er besitzt eine Wortbreite von 8 Bit. Die weiteren Eigenschaften lassen sich wie folgt zusammenfassen:*

- *Akkumulatormaschine (1-Adress-Maschine), CISC, 8-Bit-Register,*
- *8 Bänke mit jeweils 8 Registern, realisiert als RAM, Umschaltung der Bänke durch Interrupts oder Unterprogramme (sog.* Registerwindowing*).*
- *Adressierungsarten direkt, indirekt, immediat, relativ, m–m-, m–r- und r–r-Transportbefehle.*
- *I/O-Ports haben separaten Adressraum, insb. gibt es Spezialbefehle für Zugriff auf I/O-Ports, sogar auf einzelne Bits.*
- *Dichte Instruktionscodierung: 1-3 Bytes/Instruktion.*
- *Mehrere Power-down-Modi.*

Es gibt jedoch heute auch Mikrocontrollergenerationen, die eine Wortbreite von 16-64 Bit besitzen. Als Beispiele lassen sich die Prozessorfamilien Motorola MC683xx, Siemens x166 und Intel x196 nennen. Die Anwendungsbereiche solcher Mikrocontroller lassen sich wie folgt beschreiben:

- *Systeme mit höheren Berechnungsanforderungen*, z. B. aus Bereichen der Signalverarbeitung und Regelungstechnik,
- *Systeme mit hohen Datenraten und vielen Datenmanipulationsoperationen*, z. B. Anwendungen der Telekommunikation,

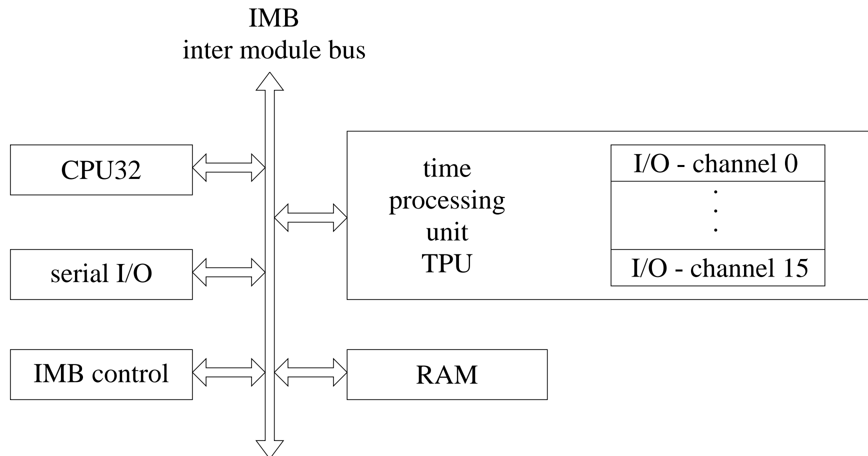


Abbildung 1.7: Architektur des Motorola MC68322-Prozessors

- *Systeme mit hohen Datenraten und steuerungsdominanten Aufgaben*, z. B. bei der Automobiltechnik.

Beispiel 1.5 *Als Beispiel wird die Familie MC683xx von Motorola betrachtet. Die CPU besitzt eine Wortbreite von 32 Bit (CPU 32). Die weiteren Eigenschaften lassen sich wie folgt zusammenfassen:*

- *68000-Prozessor, erweitert durch die meisten der Eigenschaften des 68030.*
- *CISC-Prozessor: erreicht hohe Codedichte!*
- *Fliessbandverarbeitung*
- *Standardregister (nicht in RAM!). Damit ist der Kontextwechsel langsamer als bei den 4- bis 8-Bit-Mikrocontrollern.*
- *virtuelles Speichermodell*
- *Benutzer- (User-) und privilegierter (Supervisor-)Programmiermodus; die letzten beiden Eigenschaften zielen auf die Benutzung eines Betriebssystems.*

Abb. 1.7 zeigt als Beispiel die Architektur des MC68332, der auf Anwendungsbereiche zielt, bei denen eine Mischung von berechnungsintensiven Aufgaben und komplexen I/O-Operationen vorliegt. Die dargestellte Einheit TPU (engl. time processing unit*) dient dazu, die CPU bei häufigen I/O-Operationen zu entlasten, um die Performanz für Berechnungen wegen Kontextwechseln nicht zu erniedrigen. Die TPU besitzt 16 Kanäle, die intern aus einem Zähler und einem Komparator (*capture & compare*) bestehen. Die Zähler können über externe Ereignisevents bzw. in konstanten Zeitabständen getriggert werden und generieren beim Nulldurchgang ein Ereignisevent an eine zusätzlich existierende mikroprogrammierte Steuerungseinheit, die zyklisch (Round-robin) alle Kanäle überwacht und I/O-Operationen durchführt. Diese können einen oder mehrere Kanäle betreffen. Da die 16 Kanäle zyklisch von einer einzigen Steuerungseinheit bedient werden, ergeben sich hohe Latenzen.*

Zusammenfassend kann die TPU als ein peripherer Coprozessor *aufgefasst werden. Die Komponenten kommunizieren über einen Intermodulbus (IMB). Neben den hohen Latenzen für I/O-Operationen besteht eine weitere Schwierigkeit bei peripheren Coprozessoren wie der TPU in deren Programmierung (Codegenerierung) und in der Schwierigkeit, eigene I/O-Funktionen zu definieren. Dies ist beim MC68332 zwar vorgesehen, aber beispielsweise nur über Anschluss von externem RAM möglich.*

Die Familien MC68332 und Siemens x166 sind Mitglieder von *Baukastensystemen*, bestehend aus

- *Modulbus* inkl.
  - Bussystem (Motorola IMB, Siemens X-Bus), nach aussen geführt zur Erweiterung,
  - Interruptsystem (Vektor, flexible Priorisierung),
  - Kommunikationsmodell.
- *Prozessorkern* (Cores): 16 Bit, 32 Bit, 64 Bit
- *Speicherkomponenten*: ROM, RAM, EPROM
- *peripheren Einheiten*: TPU, SIO, DMA etc.
- *Coprozessoren*, z. B. Fuzzycontrol, Graphik etc.
- *benutzerdefinierten Standardzell/Gatearray-Blöcken*.

Jedes Systemhaus bietet unterschiedliche Familien an.

1.3.2 Mikroprozessoren

Beispiele: PowerPC, Alpha, P6 etc.

Anwendungsgebiete:

- Systeme mit hohen Performanzanforderungen (z. B. PCs, Workstations),
- Systeme mit hohen Anforderungen an Berechnungen und Datenmanipulationen (z. B. Graphik, Multimedia, Telekommunikation),
- Anwendungen mit hohem Mass an Parallelität.

Gegenüber bisher beschriebenen Prozessoren besitzen diese Prozessoren folgende Eigenschaften:

- *Parallelität:* mehrere Datenpfade (funktionale Einheiten) und Multiportspeicher (Registerfiles),
- *Arten der Parallelität:*
  - VLIW (engl. *very long instruction word*): Starten mehrerer Operationen in einer Instruktion; speicherintensiv, Arrangierung der Abarbeitung zur Übersetzerzeit.
  - superskalar: dyn. Arrangierung nebenläufiger Instruktionen (zur Laufzeit); komplexe Steuerung.
  - Instruktionspipelining.

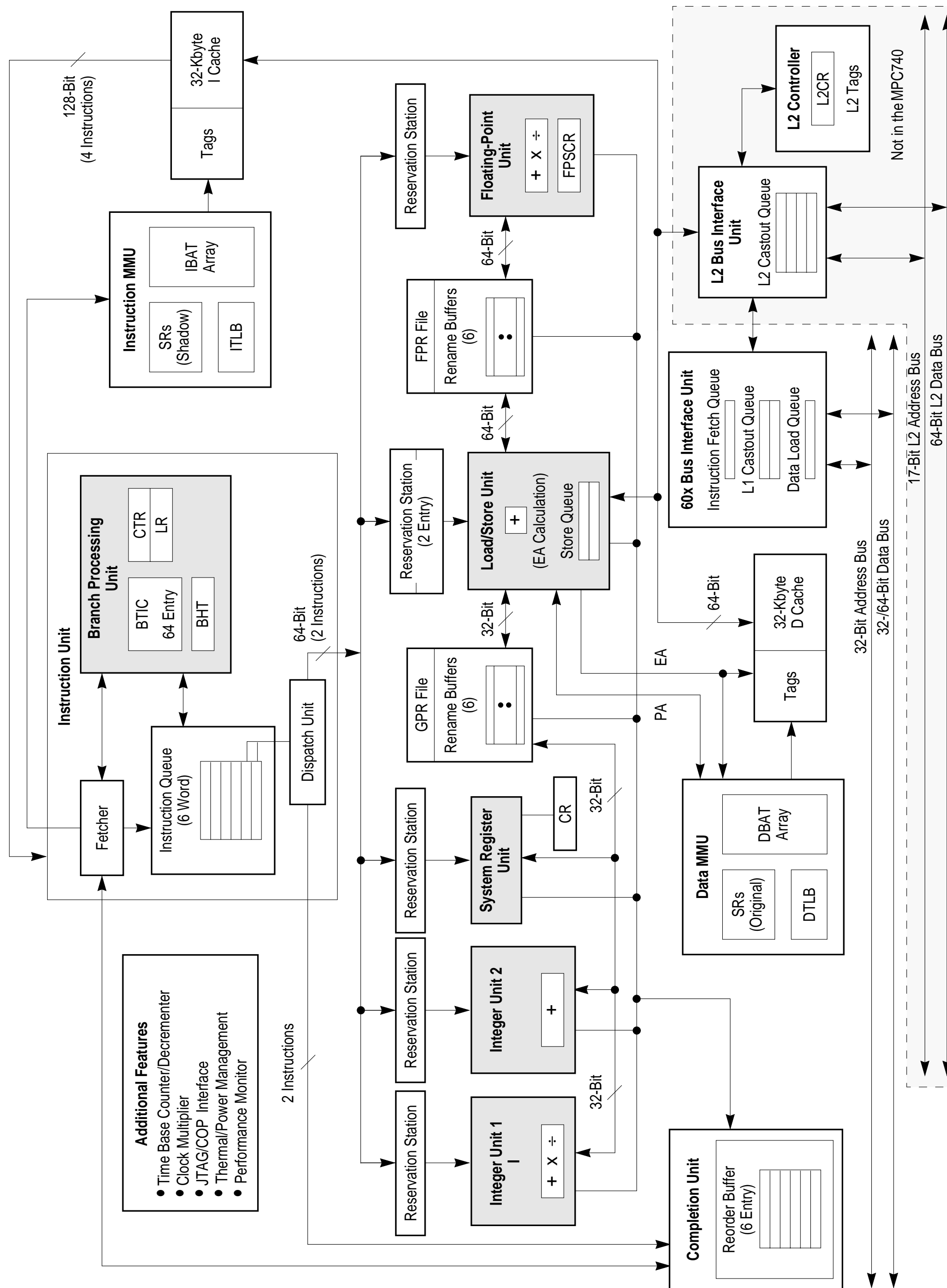


Abbildung 1.8: Aufbau eines PowerPC750

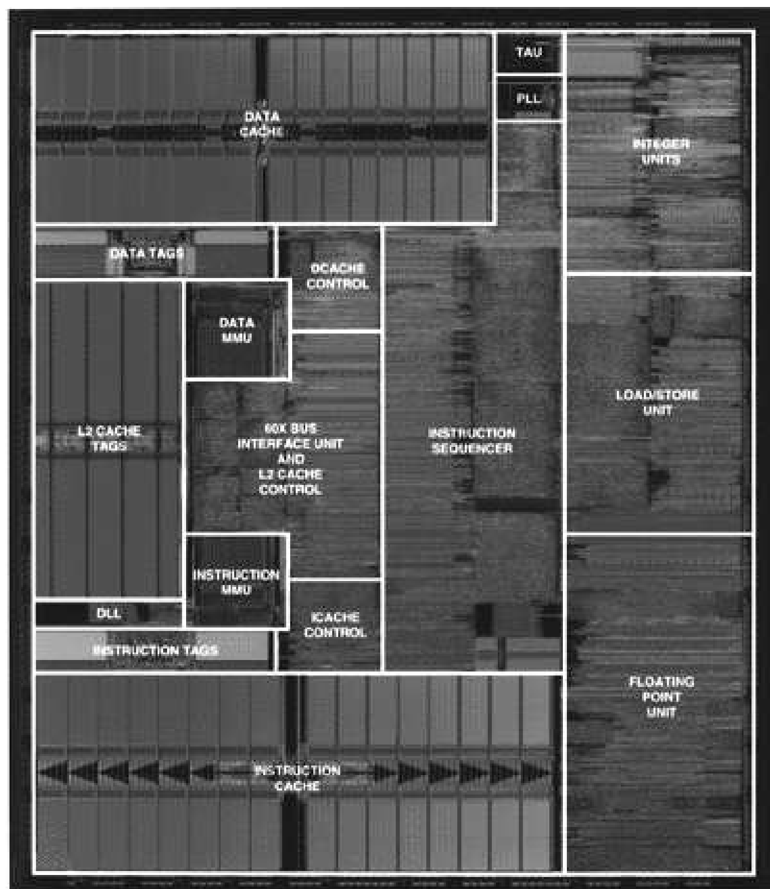


Abbildung 1.9: Layout des PowerPC750

Beispiel 1.6 *Als Beispiel wollen wir nun exemplarisch den Prozessor PowerPC750 betrachten, dessen Aufbau in Abb. 1.8 beschrieben und dessen Layout in Abb. 1.9 dargestellt ist.*

Es handelt sich um eine RISC-Architektur. Der Prozessor kann mit Taktfrequenzen von 200-266 MHz getaktet werden. Er besitzt eine Versorgungsspannung von 3.3V. Das Befehlsfliessband ist 4-stufig (fetch, decode/dispatch, execute, complete/write back) Die superskalare Architektur besitzt 6 funktionale Einheiten (Branch (BPU), 2 Integer-Einheiten (IUs), 1 Gleitkommaeinheit (FPU), 1 Load-Store-Einheit (LSU) und eine Systemregistereinheit (SRU)). Bis zu 4 Instruktionen können gleichzeitig pro Zyklus gefetched werden.

Am Layout in Abb. 1.9 erkennt man, dass bei heutigen Prozessoren das Steuerwerk einen beträchtlichen Anteil der Chipfläche belegt. Weiterhin ist für diese Klasse von Prozessoren typisch, dass die Speicherorganisation hierarchisch ist. So gibt es neben den Registern eine Cache-Hierarchie, die sowohl Instruktions-, als auch Datencache betrifft.

Beispiel 1.7 *Als weiteres Beispiel betrachten wir den Prozessor Intel i960. Es handelt sich hier um eine* Harvard-Architektur, *bei der Daten- und Adressberechnungspfade getrennt sind. Die Wortbreite beträgt 32 Bit. Charakteristisch für die Ausnutzbarkeit von Befehlsfliessbandverarbeitung sind gleich lange Befehle. Dies ist der Fall bei sog. RISC-Prozessoren (Einwortbefehle). Alle Befehle (ausser Lade- und Speicherbefehl) arbeiten auf Registeroperanden. Der Prozessor besitzt ebenfalls Cache, der auf dem Chip realisiert ist.*

Zusammenfassend lässt sich für die vorgestellten „High-Performance“-Prozessoren sagen, dass sie komplexeste IC-Technologie besitzen bei höchstoptimierter Schaltungsstruktur, wobei der Entwicklungsaufwand heute schon in den Bereich mehrerer 100 Mannjahre geht. Die Architektur ist nicht auf spezielle Anwendungsgebiete ausgelegt und damit breit einsetzbar. Diese hohen Performanz- und Flexibilitätsanforderungen verbunden mit der Notwendigkeit, am Rande des technologisch Möglichen zu fertigen, sind Quellen für eine grosse Anzahl von Entwurfsproblemen und Kosten. Der Einsatz der Prozessoren ist wegen ihrer Grösse (mehrere hundert Pins) auf Platinenentwürfe beschränkt.

Aus folgenden Gründen ist der Einsatz beschriebener Mikroprozessoren im Bereich eingebetteter Systeme schwierig bzw. nicht sinnvoll:

- Akkurate Analyse des Zeitverhaltens schwierig wegen
  - Caches,
  - dynamischer Ablaufplanung (Umordnen der Ausführung der Befehle durch den Prozessor),
  - gleichzeitiger Abarbeitung mehrerer Anweisungsblöcke.
- Komplexe Interfaces:
  - hohe Anforderungen an Speichersystem,
  - Caches, Kohärenzprotokolle,
  - Buskommunikationsbandbreite schwer abschätzbar.

1.3.3 Anwendungsspezifische Prozessoren (ASIP)

ASIPs besitzen spezielle Instruktionssätze, funktionale Einheiten, Register und spezielle Verbindungsstrukturen. Die Spezialisierung führte zu Architekturen, die sich von denen eines Mikroprozessors stark unterscheiden.

ASIPs stellen offensichtlich bezüglich Flexibilität und Performanz die Nahtstelle von der Softwareseite zur Hardwareseite her. Aus Kostengründen ist ein ASIP oft nur ein „abgespeckter“ Prozessor und damit günstiger als ein Vielzweckprozessor, aber aufgrund seiner (wenn auch beschränkten) Programmierbarkeit immer noch flexibler als dedizierte Hardware.

Gründe zur Entwicklung von anwendungsspezifischen Prozessoren (ASIPs) sind damit

- *Flexibilität:* Allgemein programmierbare Prozessoren sind zu langsam, zu gross, haben zu viele Pins, passen nicht auf einen Chip etc.
- *Kosten:* Sparen von Pins, kleineres Gehäuse, einfachere Interface- und Speicherarchitektur ausreichend.
- *Leistungsverbrauch:* Mobile Systeme, thermische Probleme, usw.

Die Unterschiede zu allgemein programmierbaren Prozessoren beziehen sich insbesondere auf folgende Merkmale:

- *Instruktionssatz:*
  - Operatorverkettung (z. B. Multipliziere/Akkumuliere, Vektoroperationen), dadurch weniger Instruktionsfetchzyklen und resultierend höhere Performanz.
  - Ausnutzung von Parallelität, z. B. nebenläufige Daten- und Adressberechnungen. Dies können VLIW- und superskalare Maschinen zwar auch, aber bei ASIPs ist ein geringerer Steuerungsaufwand erforderlich.

  Man erwartet also durch einen spezialisierten Instruktionssatz einen niedrigeren Steuerungsaufwand, niedrigere Kosten und höhere Codedichte, allerdings keine Ersparnis im Leistungsverbrauch.
- *Funktionseinheiten* (FUs):
  - Spezialisierte Funktionen, z. B. Operationen auf Zeichenketten, Pixeloperationen, Verkettung von Operationen (DSP).
  - Adaption der Wortlänge bringt Gewinn in Kosten, Leistungsverbrauch, Programmausführungszeit, allerdings kann möglicherweise die Zykluszeit einer Instruktion grösser werden.
- *Speicherstruktur:*
  - Anzahl und Grösse der Speicherbänke,
  - Anzahl und Wortbreite der Speicherports,
  - Zugriffsarten (nx read, mx write, read-modify-write, page mode etc.),
  - Funktion (RAM, FIFO etc.)

  Der Gewinn zeichnet sich hier in den Kosten (Fläche), in höherer Parallelität und dadurch höherem Leistungsverbrauch aus.
- *Verbindungsstruktur:*
  - optimierter Datenpfad mit reduzierter Verbindungsstruktur

  - Spezialregister (MUL/ACC, Zwischenergebnisse)

  Der Einfluss betrifft die Komplexität der Verdrahtung und Steuerung und die Zykluszeit von Instruktionen.

- *Steuerwerk:*
  - Fliessbandverarbeitung,
  - spekulative Ausführung,
  - mikroprogrammiertes Steuerwerk oder FSM.

1.3.4 Digitale Signalprozessoren – DSPs

Zu der Klasse anwendungsspezifischer Prozessoren gehört die Klasse der digitalen Signalprozessoren – DSPs.

Die Anwendungsgebiete sind charakterisiert durch

- Dominanz von Datenfluss,
- komplexe, regelmässige arithmetische Operationen auf Feldern, z. B. Filter, FFT, Matrizenoperationen,
- hohe Nebenläufigkeit,
- spezielle Adressiersequenzen, z. B. linear ($a_1x_1+a_2x_s+\cdots+a_nx_n$), modulo, bitrevers.

Die wesentlichen Merkmale von DSPs lassen sich wie folgt zusammenfassen:

- Festkomma- und Gleitkommaarithmetik,
- Wortlängen 16 bis 64 Bit,
- Leistungsverbrauch im mW- bis W-Bereich,
- spezielle Speicher- und Busstruktur,
- Mehrprozessorsysteme und Architekturen mit Coprozessoren.

Beispiele für käufliche Signalprozessoren sind ADSP21060 (Analog Devices), TMS320 C80 (Texas Instruments), Motorola M56000 (Festkomma) und M96000 (Gleitkomma). Die ersten beiden sind sehr unterschiedliche Architekturen, die jedoch die wesentlichen für DSPs typischen Architekturmerkmale besitzen.

Beispiel 1.8 *Als Beispiel wird der Signalprozessor ADSP21060 (SHARC) von Analog Devices betrachtet. Es handelt sich um eine Load-Store-Architektur mit einer 32-Bit-Gleitkomma-ALU (siehe Abb. 1.10).*

Das Operationswerk (links) besitzt 3 parallele Einheiten: eine ALU, eine Schiebeeinheit (Multi-Bit) und eine Multipliziereinheit. Die entsprechenden Befehle des Instruktionssatzes sind Ein-Zyklen-Befehle. Die Prozessorfrequenz beträgt laut Hersteller 40 MHz. Weiterhin unterstützt der Instruktionssatz spezielle Instruktionen, z. B. zur Betragsbildung. In einer Instruktion können mehrere nebenläufige Operationen initiiert werden. An Datenregistern liegen zwei Bänke mit jeweils 16 40-Bit-Registern (interne Wortbreite) vor. Die zweite Bank kann z. B. bei einem Kontextwechsel zwischen zwei Tasks eingesetzt werden.

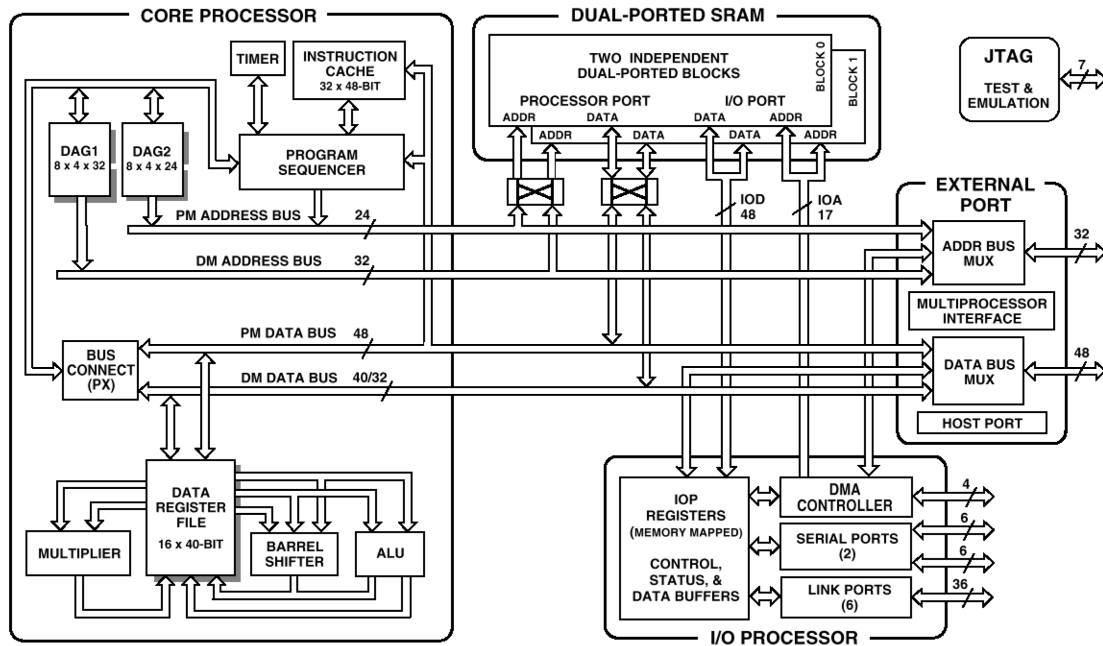


Abbildung 1.10: Signalprozessor ADSP21060 (SHARC)

Spezielle Befehle zur Schleifenabarbeitung, insbesondere von geschachtelten Schleifen, werden im Programmsequenzer unterstützt. Der kleine Instruktionscache mit 32 Einträgen lässt auch darauf schliessen, dass die Abarbeitung von Blöcken, die aus einzelnen Instruktionen bestehen, beschleunigt werden soll. Der Cache wirkt damit performanzsteigernd bei kleinen, häufig iterierten Schleifen. Es existieren zwei unabhängige Adressgenerierungseinheiten. Das Speichersystem besteht aus einem grossen Zwei-Port-Speicher, organisiert in zwei Bänken, was typisch für DSPs ist. Der erste Port wird über einen Crossbar-Switch für Daten- und Adressbus gemultiplext. Der zweite Port ist exklusiv für den dargestellten I/O-Prozessor reserviert.

Desweiteren besitzt der SHARC-Prozessor 6 4-Bit-Link Ports, die eine Datenübertragung zu 6 mit einem Prozessor verbindbaren anderen gleichartigen Prozessoren mit einer Datenrate von 40 MBytes/s pro Kanal erlauben. Damit lassen sich Punkt-zu-Punkt-Verbindungen erzeugen. Alle Instruktionen sind 48-Bit-Wörter, wobei wie bereits angedeutet in einer Instruktion maximal 3 parallele Operationen initiiert werden können. Schliesslich verfügt der Prozessor auch über eine komplexe DMA-Einheit. Der Aufbau des I/O-Prozessors zielt folglich auf die Beschleunigung von Speicher- und Datentransferoperationen.

Als weiteres Beispiel wird der Prozessor TMS320C80 von Texas Instruments betrachtet.

Beispiel 1.9 *Beim Prozessor TMS320C80 von Texas Instruments handelt es sich um eine Mehrprozessorarchitektur mit 4 32-Bit-Festkomma-Prozessoren und einem 32-Bit-Gleitkommaprozessor (Master DSP) auf einem Chip, siehe Abb. 1.11. Die Speicherarchitektur ist äusserst komplex (lokale Register in den DSPs, 4x2-KB-RAM-Bänke (512 Worte/Bank), 2 KB Instruktionscache pro Prozessor (256 Worte)). Damit existiert gegenüber dem SHARC eine höhere Nebenläufigkeit im Speicherzugriff, allerdings ist der Speicher viel kleiner.*

Zur Verbindung der Prozessoren existiert ein nahezu vollständiger Crossbar-Switch. Dieser reduziert Kommunikationsbeschränkungen und Zugriffskonflikte des Shared Memory. Dadurch soll eine hohe interne Speicher- und Kommunikationsbandbreite erreicht werden, was offensichtlich das untätige Warten der DSPs auf Daten verhindern soll.

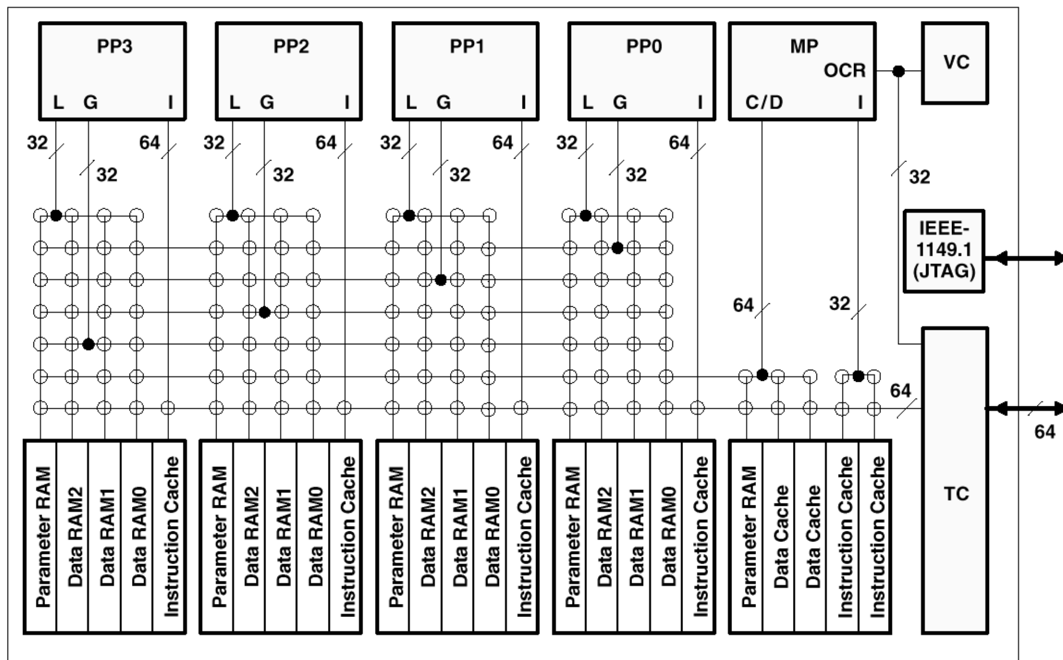


Abbildung 1.11: Aufbau eines TMS320C80 (MVP) von Texas Instruments

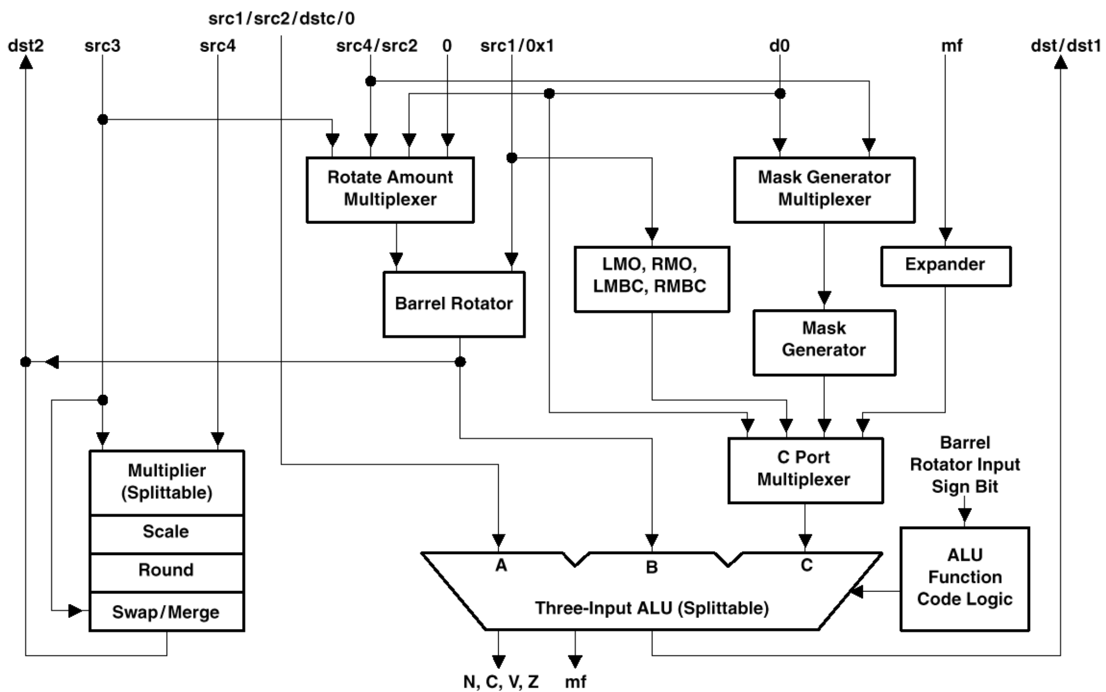


Abbildung 1.12: Datenpfad des TMS320C80

Als Controller dient ein 32-Bit-RISC-Gleitkommaprozessor. Die 4 Festkommaprozessoren (Advanced DSP) besitzen folgende Eigenschaften (siehe Abb. 1.12):

- *Multiplizierer, Schiebeeinheit, ALU mit 3 Eingängen, die in kleinere 8-Bit-Einheiten aufgespalten werden kann.*
- *Unterstützung spezieller Operationen auf Bitebene (u. A. Pixelexpander),*
- *2 Adress-ALUs für Indexberechnungen,*
- *Instruktionswortbreite 64 Bit (horizontal) für parallele Datenoperation, Datentransfer und einen globalen Datentransfer.*

Gebaut wurde der Prozessor für Video- und Multimediaapplikationen. Die Architektur erlaubt die beschleunigte Ausführung vor allem kleiner (256 Worte), berechnungsintensiver Programme auf kleinen Datenbereichen bei einfachen Operationen.

1.4 Hardwareimplementierungen

1.4.1 ASIC-Entwurf

Zum ASIC-Entwurf gehört der voll kundenspezifische Entwurf integrierter Schaltungen, bei dem die Schaltung bis auf das Layout vom Entwerfer gestaltet wird. Desweiteren gehört dazu der Standardzellenentwurf, der vom Fertigungsaufwand gleich aufwendig ist wie der voll kundenspezifische Entwurf. Jedoch ist der Entwurfsprozess dahingehend einfacher, dass für die verwendeten elementaren Gatter Bibliotheken mit Standardzellenlayouts vorliegen, die dann platziert und verdrahtet werden müssen. Beide Entwurfstypen sind aus Kostengründen nur bei hohen Stückzahlen empfehlenswert. Jedoch gibt es auch ASICs, die bereits eine vorgefertigte Halbleiterstruktur besitzen, so dass bei der Fertigung nur noch die Verdrahtung der Zellen untereinander als Fertigungsschritt zu vollziehen ist. Hierzu gehören beispielsweise Gatearrays und Seas-of-Gates. Bei Gatearrays liegen vorgefertigte Zellen in Reihen fest, die durch dedizierte Verdrahtungskanäle miteinander verdrahtet werden können. Bei Seas-of-Gates verrät der Name bereits, dass hier auch Zellen als Verdrahtungszellen zwischen Gates konfiguriert werden können. Aufgrund der Vorfertigung ist die Flächenausbeute bei diesen Typen von ASICs verständlicherweise geringer als beim voll kundenspezifischen Entwurf.

Beispiel 1.10 *Als weiteres Beispiel zeigt Abb. 1.13 ein Ein-Chip-System, bei dem ein Mikrocontroller mit einer kundenspezifischen Gatearray-Schaltung (siehe oben) über einen chipinternen Bus verbunden ist. Das System C167 der Firma Siemens stellt einen Chip zur Motorregelung dar.*

Beispiel 1.11 *Abbildung 1.14 zeigt ein Ein-Chip-System der Firma Motorola. Es handelt sich um einen Rastergraphikprozessor, der einen MC68332 als Makrozelle in einem Standardzellentwurf integriert.*

Andere Beispiele von eingebetteten Ein-Chip-Systemen, bei denen ein Standardprozessor in anwendungsspezifischer Hardware eingesetzt wird, sind Prozessoren für Laserdrucker (z. B. Motorola Flexcore) oder PCMCIA-Controller-Chips (z. B. SUNDISK).

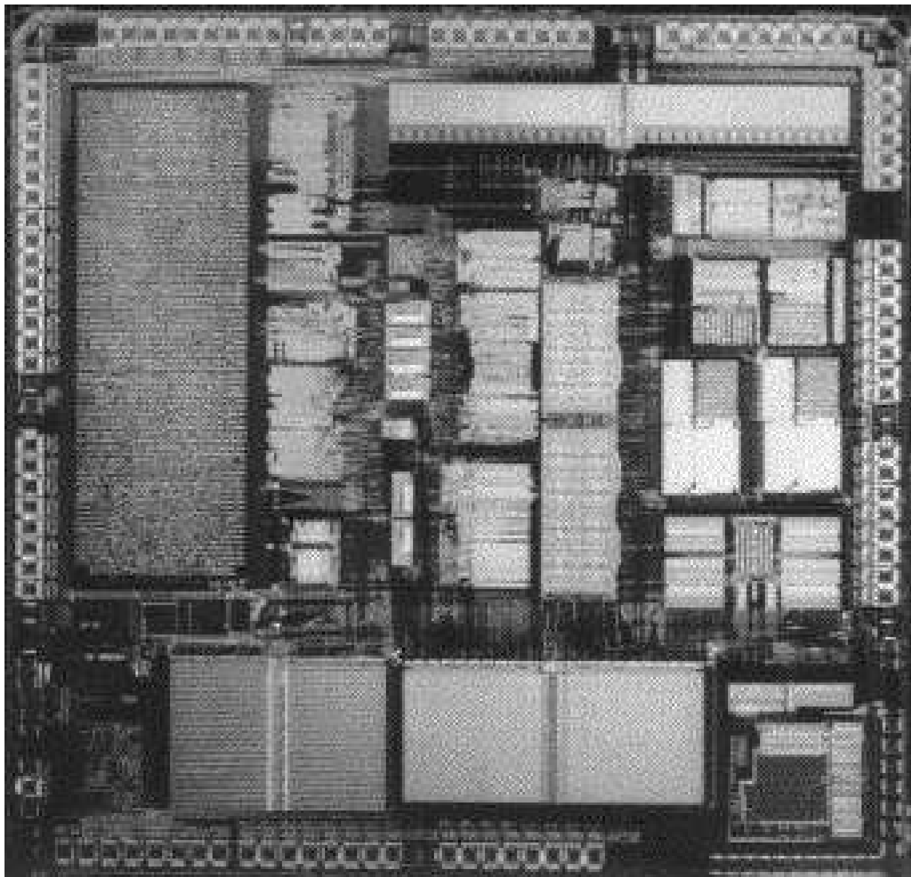


Abbildung 1.13: C167-Mikrocontroller mit kundenspezifischem Gatearray

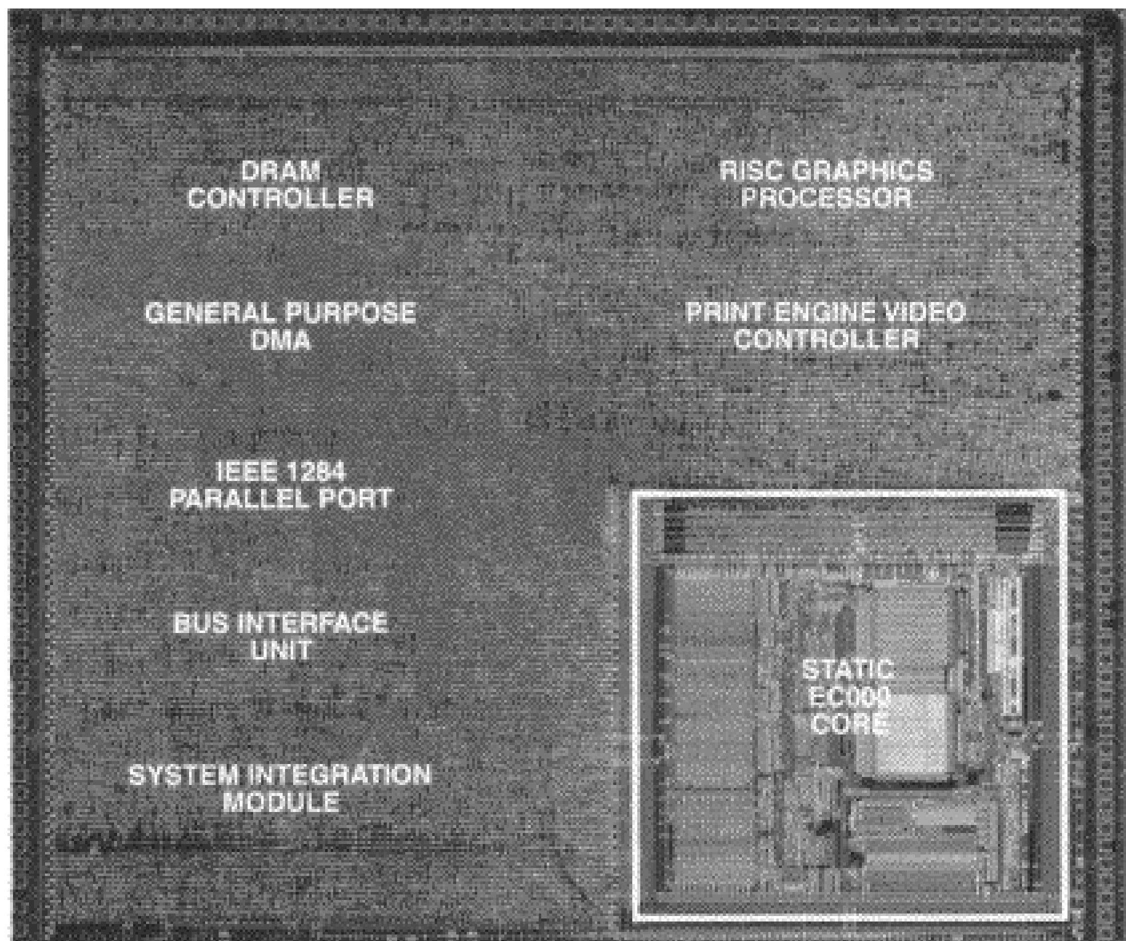


Abbildung 1.14: Rastergraphikprozessor mit Standardzellenentwurf und MC68322-Prozessor

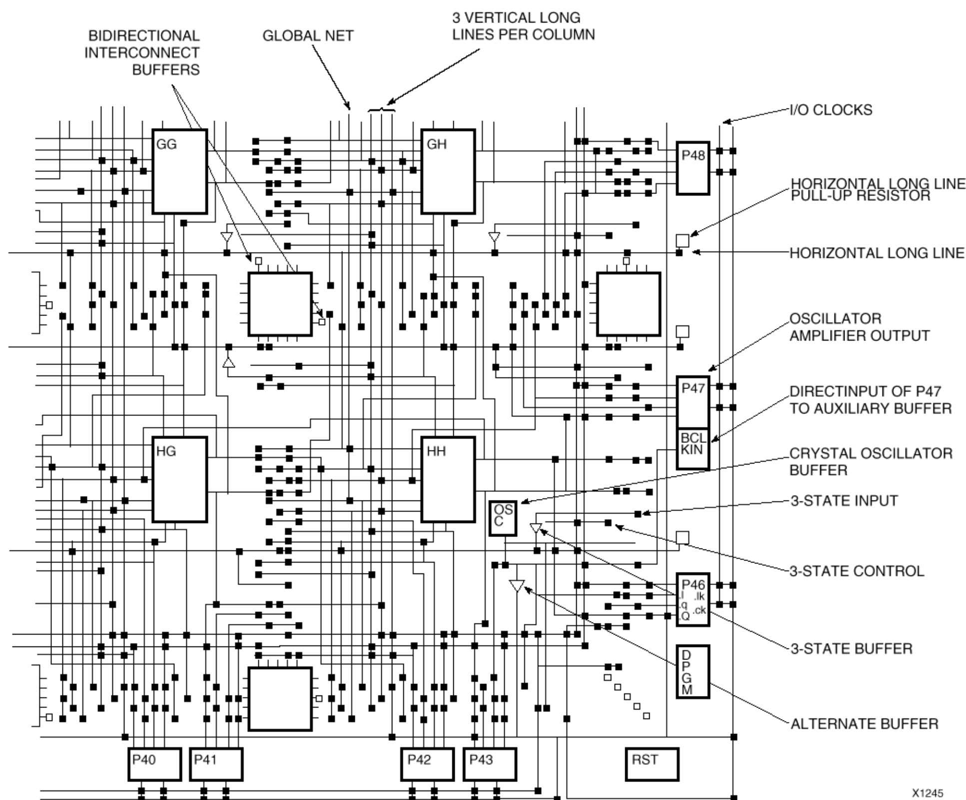


Abbildung 1.15: Aufbau eines FPGA (Xilinx)

1.4.2 Programmierbare Hardware

Zur programmierbaren Hardware gehören u. A. FPGAs (engl. *field programmable gate arrays*), und PLAs (engl. *programmable logic arrays*). Während PLAs nur Boolesche Funktionen einer bestimmten Form (zweistufige Form) implementieren können, lassen sich mit FPGAs auch Speicherzellen realisieren. FPGAs eignen sich dazu hervorragend, Steuerwerke (FSMs) zu realisieren.

Abbildung 1.15 zeigt den prinzipiellen Aufbau eines Xilinx FPGA (XC4005PG156). Es können 5000 Gatteräquivalente realisiert werden. Das FPGA wird durch einen Strom von Daten konfiguriert; dies kann auf unterschiedliche Art erfolgen. Ferner gibt es eine Funktion, mit der man den inneren Zustand sämtlicher Speicherzellen des FPGA auslesen kann (sog. Readback-Funktion). Der beschriebene Baustein besitzt 112 I/O-Pins und 196 (14x14) Logikblöcke (sog. CLBs, engl. *configurable logic blocks*), siehe Abb. 1.16. Insgesamt können 6272 Bit RAM verwendet und 616 Flipflops realisiert werden.

Ein CLB kann jede Boolesche Funktion von 5 Eingängen bzw. 2 Boolesche Funktionen von 4 Eingängen implementieren.

IOBs (*input/output blocks*, siehe Abb. 1.17) sind spezielle Blöcke, die das Ein- und Auslesen von Registerwerten über die Pins von bzw. nach aussen ermöglichen. Auch diese können in ihrer Funktion (Richtung, Modus) konfiguriert werden. Für das Platzieren und Verdrahten einer Netzliste auf einem FPGA gibt es spezielle Software, die als Eingabe die Netzliste nimmt und als Ergebnis dis Konfigurationsdatei ausgibt, die in das FPGA eingelesen wird und die Zellen konfiguriert.

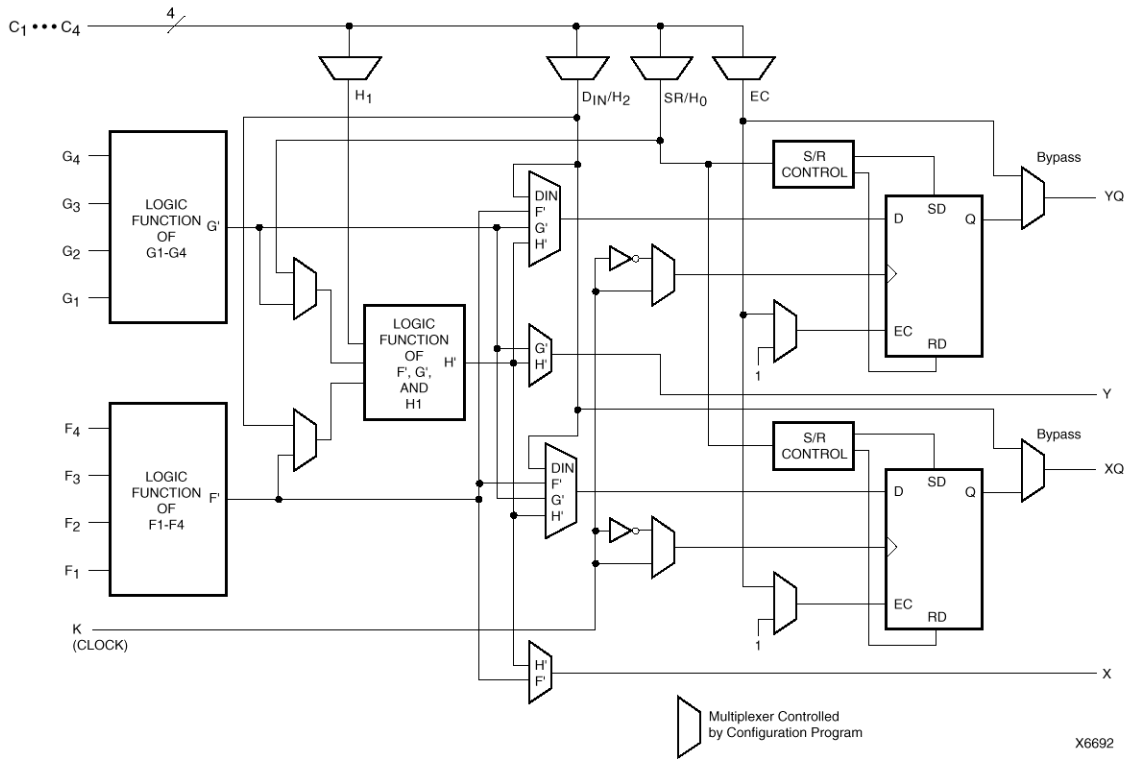


Abbildung 1.16: CLB

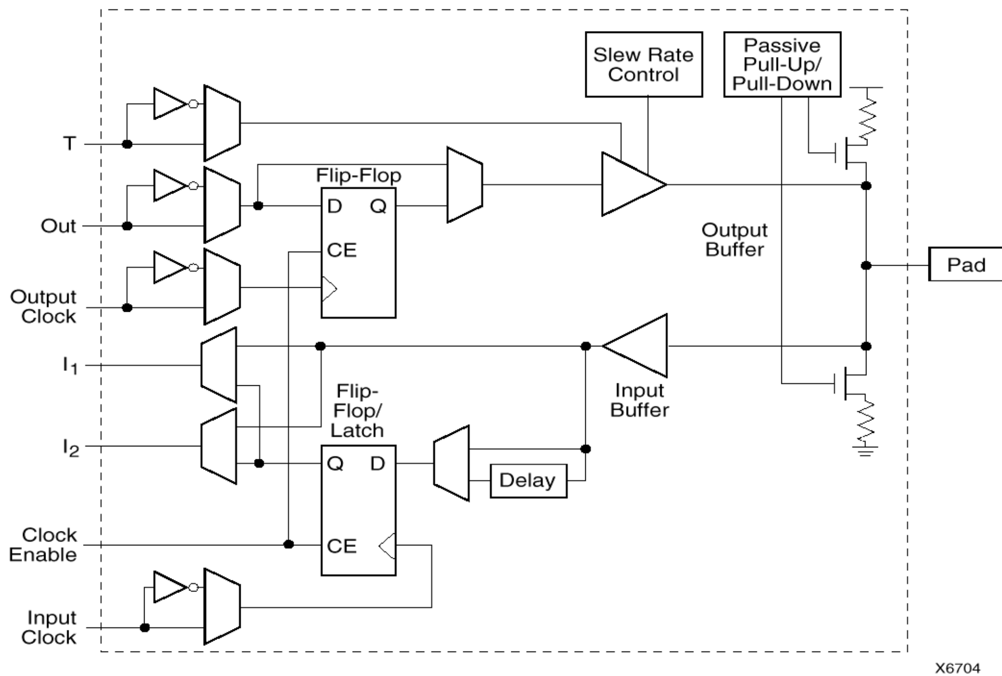


Abbildung 1.17: Aufbau eines IOBs

Heutzutage gibt es FPGAs mit mehreren 100 000 Gatteräquivalenten.

Die Einsatzgebiete von FPGAs im Bereich des Entwurfs komplexer eingebetteter Systeme lassen sich zusammenfassen als Applikationsgebiete, bei denen

- Entwurfszeit hohe Priorität hat,
- Softwarelösungen nicht die erforderliche Performanz haben,
- ASIC-Lösungen aufgrund der Kosten nicht geeignet sind (Stückzahl!),
- Grösse und Leistungsverbrauch zweitrangig sind,
- Flexibilität wichtig ist (für spätere Änderungen, Anpassungen an andere Systeme, Erweiterungen der Funktionalität).

Kapitel 2

Entwurfsmethodik

Dieses Kapitel dient dazu, die Vorlesung in unterschiedlicher Art und Weise einzubetten. Abschnitt 2.1 enthält eine historische und inhaltliche Einbettung, die den Systementwurf, so wie er hier verstanden wird, in die heutige Praxis einbettet. Der folgende Abschnitt 2.2 dient der Erläuterung der wesentlichen Grundbegriffe: Abstraktion und Entwurfsrepräsentationen.

2.1 Entwurfsmethodik

2.1.1 Erfassen und Simulieren

Bis heute werden die meisten integrierten Schaltungen auf der Basis einer Entwurfsmethodik entworfen, die sich aus den beiden Hauptkomponenten „Erfassen und Simulieren“ zusammensetzt. Man startet mit einer informalen, umgangssprachlichen Spezifikation des Produktes, die noch keine Informationen über die konkrete Implementierung enthält. Es wird also nur die Funktionalität bestimmt, aber nicht die Art und Weise ihrer Realisierung. Anschliessend wird dann 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 Transistor-Diagramme umgesetzt und in entsprechende Software-Systeme eingegeben. Auf dieser Basis lassen sich dann umfangreiche Simulationen der Funktionalität und des Zeitverhaltens sowie Untersuchungen der Testbarkeit durchführen. Diese 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 nicht nur im Bereich des Hardware-Entwurfs sondern auch bei der Programmierung von Mikroprozessorsystemen, also beim Software-Entwurf, eingesetzt. Eine informale Spezifikation wird in Blockstrukturen und anschliessend in Assembler-Programme verfeinert. Es folgen Simulationen und Emulationen zur Validierung von Funktionalität und Zeitverhalten, bevor das endgültige Programm in Maschinensprache generiert wird.

2.1.2 Beschreiben und Synthetisieren

In den letzten Jahren wurde die Logik-Synthese integraler Teil der Entwurfsmethodik von Hardware-Systemen. Ein Entwurf wird hier bezüglich seines Verhaltens in einer si-

mulierbaren, das heisst ausführbaren Beschreibungssprache spezifiziert; so kann man zum Beispiel ein Steuerungssystem durch Boole'sche Gleichungen und Zustandsdiagramme beschreiben. Die Struktur der Implementierung wird dann automatisch durch entsprechende Syntheseverfahren generiert, im Vergleich zum Hand-Entwurf eine sehr viel schnellere und vor allem sichere Entwurfsart.

Dieser Weg des „Beschreibens und Synthetisierens“ kann nun bei verschiedenen Teilproblemen eines gesamten Entwurfs durchgeführt werden. Auf der *Logik-Ebene* werden funktionale Einheiten (z. B. ALUs, Vergleicher, Multiplizierer) und Steuerungseinheiten (z. B. Zustandsmaschinen) durch die Logiksynthese automatisch generiert. Hierzu gehören Verfahren zur Minimierung Boole'scher Ausdrücke, Zustandsminimierungen und die Technologie-Abbildung, d. h. Implementierung der minimierten Funktionen mit Gattern aus einer speziellen Entwurfsbibliothek.

Ein anderes Beispiel ist die *Architektur-Synthese*. Hier können integrierte Schaltungen synthetisiert werden, die aus Speicherbausteinen, Steuerungslogik und funktionalen Bausteinen bestehen. Das Verhalten dieser Prozessoren kann durch Algorithmen, Flussdiagramme, Datenflussgraphen, Instruktionssätze oder auch durch verallgemeinerte Zustandsmaschinen beschrieben werden, bei denen mit jedem Zustand eine beliebig komplexe Berechnung verbunden sein kann. Die Transformation in eine strukturelle Beschreibung erfolgt anschliessend durch die 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 Art der internen Busse zur Datenkommunikation sowie die verwendeten funktionalen Einheiten wie Multiplizierer oder ALUs. Daher befasst sich die Allokation auch wesentlich mit der Balancierung von Kosten gegenüber Leistungsfähigkeit der Schaltung.

- Die *Ablaufplanung* teilt dem spezifizierten Verhalten Zeitintervalle zu, so dass anschliessend in jedem Zeitschritt bekannt ist, welche Daten von einem Register zu einem anderen transportiert werden und wie sie dabei von den funktionalen Einheiten transformiert werden.

- Die *Bindung* ordnet abschliessend jeder Variablen eine entsprechende Speicherzelle, jeder Operation eine funktionale Einheit und jeder Datenkommunikation einen Bus oder eine Verbindungsleitung zu.

Auch hier gibt es wieder eine direkte Parallele zum *Software-Entwurf*. Im Gegensatz zum „Erfassen und Simulieren“ wird die Funktionalität des Systems durch eine ablauffähige *Hochsprache* spezifiziert, z. B. C, C++, Oberon. Aufgabe des Übersetzers ist dann die automatische Generierung des 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 Software-Entwurf die Zielarchitektur im Allgemeinen gegeben ist, sind in beiden Fällen fast die gleichen Ablaufplanungs-, Bindungs- und Optimierungs-Problemstellungen zu lösen. So sind auch bei der Software-Übersetzung die Operationen und Datentransporte Zeitschritten zuzuordnen unter Berücksichtigung der zur Verfügung stehenden Ressourcen, wie Busbandbreite, Zahl der Busse, Zahl der internen Register, Speicherbedarf und Zahl und Art der parallelen arithmetischen Einheiten. Ziel der Bindung ist dann auch hier, 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 (Very Long Instruction Word) oder auch für Signalprozessoren eingesetzt, da sie alle durch interne Parallel- und Fliessbandverarbeitung (Pipelining) ausgezeichnet sind.

2.1.3 Spezifizieren, Explorieren und Verfeinern

Auf der Ebene komplexer Systeme ist die Entwurfsmethodik bei weitem noch nicht so ausgereift wie in den bisher beschriebenen Bereichen. Dennoch scheint sich hier ein Paradigma durchzusetzen, das durch die Stichworte „Spezifizieren“, „Explorieren“ und „Verfeinern“ beschrieben werden kann.

In der *Spezifikationsphase* wird in einem sehr frühen Stadium des Entwurfsprozesses eine ausführbare 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, verschiedene Realisierungsalternativen bezüglich Ihrer Kosten und Leistungsfähigkeit miteinander zu vergleichen. Wesentliche Aufgabe ist es hier, die Systemfunktionen 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 auf die verschiedenen Hardware- und Softwarekomponenten verteilt und die korrekte Kommunikation zwischen diesen Einheiten sichergestellt. Die Ausgangslage ist also vergleichbar mit der nach der Bestimmung eines Block-Diagramms auf der Grundlage einer informalen Spezifikation, siehe Abschnitt 2.1.1. Im Unterschied dazu wurde diese Aufteilung aber nach der Exploration eines grossen Entwurfsraumes erhalten und die Verfeinerung steht 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 zur Implementierung des Systems vorliegt. Durch eine 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.

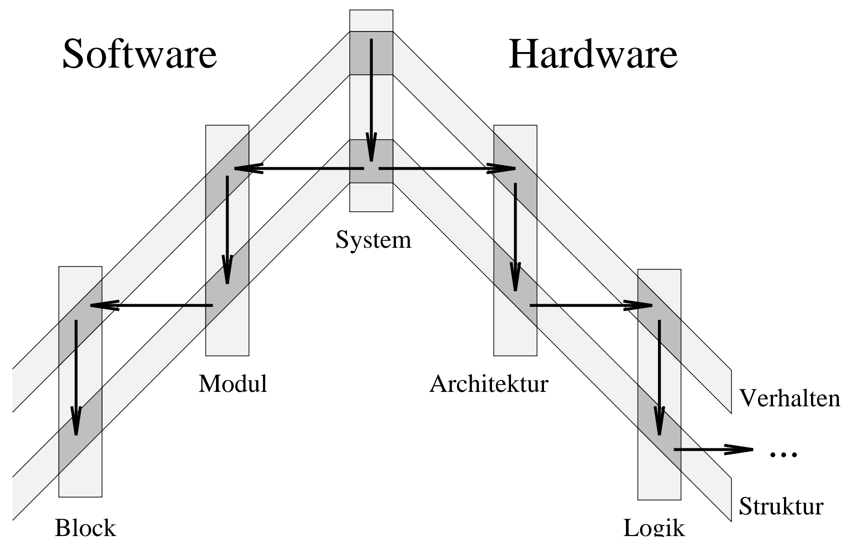


Abbildung 2.1: Graphische Darstellung einiger wichtiger Abstraktionsebenen und Sichten beim Systementwurf

2.2 Abstraktion und Entwurfsrepräsentationen

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

2.2.1 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 vorangegangene 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. Man kann also Modelle anhand des Grades Ihrer Verfeinerung nach Abstraktionsebenen klassifizieren.

Auf der anderen Seite gibt es aber auch unterschiedliche *Sichten* eines Objektes. So kann man eine Schaltung als Verbindung von Einzelkomponenten betrachten oder auch 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 wir uns im Rahmen der Vorlesung vor allem mit den Abstraktionsebenen *Architektur und Logik* beim Hardware-Entwurf, *Modul und Block* beim Software-Entwurf und *System* beim Entwurf heterogener Systeme sowie mit den Sichten *Verhalten und Struktur* auseinandersetzen, siehe auch die graphische Darstellung in Abb. 2.1.

Die folgende Aufstellung soll die unterschiedlichen Abstraktionsebenen stichwortartig beschreiben:

System: Die Modelle der System-Ebene beschreiben das zu entwerfende Gesamtsystem

auf der Ebene von Netzwerken aus komplexen, miteinander kommunizierenden Teilsystemen.

Architektur: Die Architektur-Ebene gehört zum Bereich des Hardware-Entwurfs. Modelle auf dieser Ebene beschreiben kommunizierende funktionale Blöcke, die komplexe Operationen ausführen.

Logik: Die Logik-Ebene gehört ebenfalls zum Hardware-Bereich. Die Modelle dieser Ebene beschreiben verbundene Gatter und Register, die Boole'sche Funktionen berechnen.

Modul: Die Modul-Ebene gehört zum Software-Bereich. Die entsprechenden Modelle beschreiben Funktion und Interaktion komplexer Module.

Block: Die Block-Ebene gehört ebenfalls zum Software-Bereich. Die entsprechenden Modelle beschreiben Programme bis hin zu Instruktionen, die auf der zugrundeliegenden 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 Verhaltens-Sicht werden Funktionen unabhängig von ihrer konkreten Implementierung beschrieben.

Struktur: In der strukturellen Sicht hingegen werden kommunizierende Komponenten beschrieben. Die 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 auf der nächstniedrigeren Abstraktionsebene sind, siehe auch Abb. 2.1.

Bei dieser Darstellung wird allerdings stark vereinfachend ausser Acht gelassen, dass bei einem konkreten Entwurf viele Iterationen zwischen den Abstraktionsebenen notwendig werden, also nicht nur „top-down“, sondern auch „bottom-up“ in den Abstraktionsebenen vorgegangen wird. Einige Systemteile werden zudem direkt auf unteren Abstraktionsebenen entworfen, so dass zu einem bestimmten Zeitpunkt im Entwurfsprozess nicht alle Systemteile den gleichen Abstraktions- oder Verfeinerungsgrad aufweisen.

Aufgabe der *Synthese* ist nun die (teilweise) automatische Transformation zwischen den verschiedenen Abstraktionsebenen und Sichten. Um die Zusammenhänge etwas deutlicher zu machen, sollen nun anhand von Synthesebeispielen einige der besprochenen Ebenen und Sichten näher erläutert werden.

2.2.2 Synthese

2.2.2.1 Architektur-Synthese

Aufgabe der Synthese auf Architektur-Ebene ist es, eine strukturelle Sicht aus einer Verhaltensbeschreibung zu generieren. Demzufolge wird hier der Grad der Parallelität von Operationen bestimmt. Wesentliche Aufgaben sind (siehe Abschnitt 2.1.2)

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

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

Beispiel 2.1 *Das folgende Beispiel soll eine Schaltung modellieren, die eine Differentialgleichung der Form* $y'' + 3xy' + 3y = 0$ *im Intervall* $[x, a]$ *mit der Schrittweite* dx *und Anfangswerten* $y(0) = y$, $y'(0) = u$ *mit Hilfe der Euler-Methode numerisch löst. Eine Verhaltensspezifikation in einer Beschreibungssprache (hier VHDL) würde etwa folgendermassen aussehen:*

```
ENTITY dgl IS
  PORT(x_in,y_in,u_in,dx_in,a_in: IN REAL; activate: IN BIT; y_out: OUT REAL);
END dgl;

ARCHITECTURE behavioral OF dgl IS BEGIN
  PROCESS (activate)
    VARIABLE x, y, u, dx, a, x1, u1, y1: REAL;
  BEGIN
    x := x_in; y := y_in; u := u_in; dx := dx_in; a := a_in;
    LOOP
      x1 := x + dx;
      u1 := u - (3 * x * u * dx) - (3 * y * dx);
      y1 := y + (u * dx);
      x := x1; u := u1; y:= y1;
      EXIT WHEN x1 > a;
    END LOOP;
    y_out <= y;
  END PROCESS;
END behavioral;
```

Der Block mit dem Schlüsselwort `ENTITY` *stellt eine Beschreibung der Ein- und Ausgänge der* `ARCHITECTURE` *mit dem Namen* `dgl` *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* `PROCESS`*, der durch eine Änderung des Signals* `activate` *gestartet wird. Dieser intern sequentielle Prozess definiert die lokalen Variablen* `x, u, u, dx, a, x1, y1, u1` *und realisiert die eigentliche Iteration.*

Nach der Architektur-Synthese könnte ein Blockschaltbild wie in Abbildung 2.2 gezeigt entstehen: Der Datenpfad der Schaltung enthält als Ressourcen einen Multiplizierer und eine ALU (arithmetisch-logische Funktionseinheit), die Addition, Subtraktion und Vergleiche ausführen kann. Desweiteren enthält die Architektur einen Speicher, eine Einheit zur Verteilung der Daten auf die Funktionsblöcke des Datenpfades sowie eine Steuerungseinheit. Diese Sicht könnte auch durch eine strukturelle Beschreibung in einer entsprechenden Beschreibungssprache, z. B. VHDL, dargestellt werden.

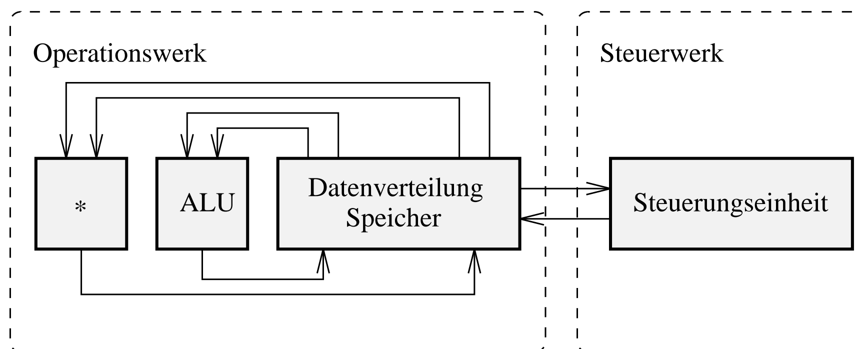


Abbildung 2.2: Beispiel einer strukturellen Sicht auf Architektur-Ebene

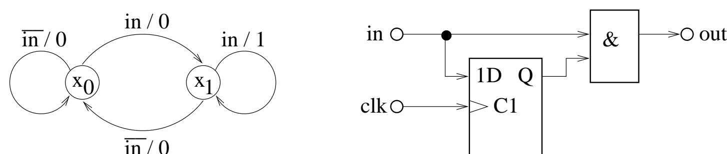


Abbildung 2.3: Beispiel einer Verhaltenssicht (Zustandsdiagramm) und einer strukturellen Sicht auf Logik-Ebene

2.2.2.2 Logik-Synthese

Aufgabe der *Logik-Synthese* ist die Generierung einer strukturellen Sicht auf Logik-Ebene. Demzufolge bestimmt sie also die Struktur einer Schaltung auf Gatter-Ebene. Ausgangspunkte der Logik-Synthese können zum Beispiel Boole'sche Gleichungen oder endliche Zustandsautomaten sein, die entweder durch graphische Methoden oder mit Hilfe eines Programms in einer Hardware-Beschreibungssprache spezifiziert wurden.

Unter anderem werden durch die Logik-Synthese die folgenden Teilprobleme gelöst:

- Optimierung Boole'scher Ausdrücke,
- Zustandsminimierung und Zustandszuordnung,
- Bindung an eine Bibliothek von Zellen, das heisst das logische Modell wird in eine Verbindung von Instanzen der Bibliotheks-Zellen transformiert.

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

Beispiel 2.2 *Die Steuerungseinheit in Abb. 2.2 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 Logik-Synthese ist es nun, eine Schaltung zu generieren, die diese Spezifikation implementiert. Als Beispiel der Sichten auf der Logik-Ebene zeigt Abbildung 2.3 einen endlichen Zustandsautomaten, der zwei oder mehr aufeinanderfolgende '1' im Eingangsstrom erkennt.*

Auch diese Sichten lassen sich in einer Beschreibungssprache formulieren. In VHDL lautet das Zustandsdiagramm

```
ENTITY rec IS
  PORT (in, clk: IN BIT; out: OUT BIT);
END rec;

ARCHITECTURE behavior OF rec IS
  TYPE state_type IS (zero, one);
  SIGNAL state: state_type := zero;
  PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1';
    IF (in = '1') THEN
      CASE state IS
        WHEN => zero
          state <= one;
          out <= '0';
        WHEN => one
          state <= one;
          out <= '1';
      END CASE;
    ELSE
      state <= zero;
      out <= '0';
    END IF;
  END PROCESS;
END behavior;
```

In diesem Modell ist das Signal `state` *vom Aufzählungstyp und speichert den Zustand des endlichen Automaten. Der Prozess wird jedesmal dann ausgeführt, wenn sich* `clk` *auf den Wert '1' ändert. Die* `WAIT`*-Anweisung synchronisiert das Modell auf den Takt* `clk`*.*

Auch das strukturelle Modell lässt sich in VHDL modellieren:

```
ARCHITECTURE structure OF rec IS
  COMPONENT and PORT (i1, i2: IN BIT; o1: OUT BIT); END COMPONENT;
  COMPONENT dff PORT (d1, cl1: IN BIT; q1: OUT BIT); END COMPONENT;
  SIGNAL int: BIT;
BEGIN
  g1: and PORT MAP (in, int, out);
  g2: dff PORT MAP (in, clk, int);
END structure;
```

Abb. 2.4 zeigt eine Schaltungsrepräsentation, die direkt dem vorangegangenen VHDL-Modell entspricht.

2.2.2.3 Software-Synthese

In Abb. 2.1 haben wir auf der Seite der Software die Abstraktionsebenen „Modul“ und „Block“ unterschieden. Auf der Modulebene könnte eine Verhaltensbeschreibung zum Beispiel in Form einer algebraischen Spezifikation vorliegen, die die Eigenschaften des zu entwickelnden Software-Systems in Form „mathematischer Sätze und Axiome“ beschreibt. Aufgabe der Verfeinerung ist es nun, eine in gewissem Sinne äquivalente strukturelle Darstellung zu erzeugen, zum Beispiel formuliert in einer höheren Programmiersprache (C,

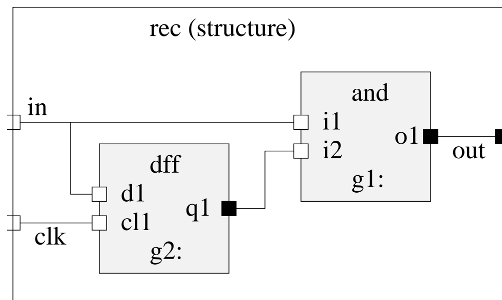


Abbildung 2.4: Schaltungsdiagramm zu einem strukturellen VHDL-Modell auf Logik-Ebene

C++, Oberon, usw.) oder einer echtzeitfähigen Sprache (zum Beispiel Esterel, Pearl, Ada 9X).

In der nächsttieferen Blockebene wird nun hieraus durch einen Übersetzungsvorgang ein Assembler- oder Maschinenprogramm erzeugt. Die folgende Aufzählung enthält einige der wesentlichen Transformations- und Optimierungsvorgänge:

- 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.
- Ablaufplanung der verschiedenen Software-Prozesse bei Echtzeitsystemen.
- Einbindung von Betriebssystemroutinen, zum Beispiel zur Interrupt-Steuerung und zur Ein- und Ausgabe von Daten.

Im Gegensatz zu einem Programm in einer höheren Programmiersprache ist ein Assemblerprogramm im Allgemeinen nicht nur erheblich länger, sondern es ist abhängig von der jeweiligen Zielarchitektur, und es fehlen Möglichkeiten zur Typ-Überprüfung und zum strukturierten Kontroll-Fluss.

Beispiel 2.3 *Als Beispiel für die beiden Sichten auf der Blockebene betrachten wir als Verhaltensbeschreibung ein C-Programm, das* $\sum_{i=0}^{i=100} i^2$ *berechnet:*

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  int i;
  int sum = 0;
  for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
  printf ("The sum of i*i from 0 ... 100 is %d\n", sum);
}
```

Nach der Übersetzung für den RISC-Prozessor MIPS R2000 könnte das folgende Assembler-Programm 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 Assembler-Programm hat insgesamt 29 Zeilen und muss anschliessend noch in ein binäres Maschinenprogramm umgesetzt werden.

2.2.2.4 System-Synthese

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

Kurze Entwurfszyklen: Ein automatisiertes Entwurfssystem ist in der Lage, einen Entwurf schneller durchzuführen, als dies ein Entwickler ohne Unterstützung von CAD-Werkzeugen könnte. Man denke nur an die Zeitersparnis 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 Produktlebensdauern und somit der „time-to-market“ festzustellen. Mit dieser Entwicklung kann man nur durch den Einsatz geeigneter CAD-Verfahren schritthalten.

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 und andererseits dadurch, dass formale Verifikationsverfahren eingesetzt werden.

Exploration des Entwurfsraumes: Gerade auf den obersten Entwurfsebenen werden grundlegende Entwurfsentscheidungen getroffen, die Leistungsfähigkeit und Kosten des implementierten Systems bestimmen. Die Konsequenzen von Fehlentscheidungen werden somit in einem frühen Entwurfsstadium deutlich, zum Beispiel die Verletzung von Zeitbeschränkungen. Man kann sich also auf der Systemebene ein Entwurfswerkzeug vorstellen, mit dem unterschiedliche Realisierungsarten einer Spezifikation schnell bewertet werden können, also eine Exploration des Entwurfsraumes unter Optimierungsgesichtspunkten unterstützt wird.

Da ein Schwerpunkt dieser Vorlesung auf Entwurfsverfahren der *Systemebene* liegt, werden die Problemstellungen auf dieser Ebene hier genauer dargestellt. Wie auch in den anderen Abstraktionsebenen, ist die Systemebene durch charakteristische Beschreibungsformen bezüglich des Verhaltens und der Struktur gekennzeichnet. Das *Verhalten*

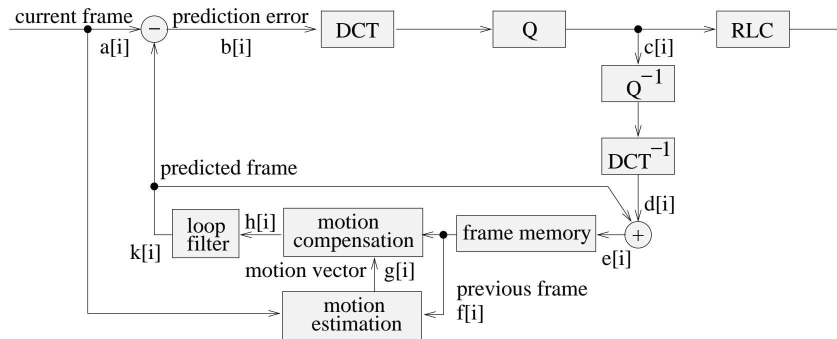


Abbildung 2.5: Darstellung eines Hybrid-Kodiereres für Bildsequenzen

wird durch Leistungsanforderungen beschrieben, wie funktionale Spezifikation, Zeitverhalten sowie nicht-funktionale Eigenschaften. Die *strukturelle Beschreibung* zeigt das System als Netzwerk aus Prozessoren, Standardkomponenten, anwendungsspezifischen integrierten Schaltungen, Verbindungsstrukturen und Speicherbausteinen. Entscheidende Entwurfsaufgabe ist auf dieser Ebene also die Partitionierung der Verhaltensbeschreibung in Teilsysteme. Hierbei spielen sehr unterschiedliche Optimierungskriterien eine Rolle, wie Kosten, Verarbeitungsleistung und Leistungsverbrauch, aber auch nicht-funktionale Kriterien wie Wiederverwendbarkeit des Entwurfs in zukünftigen Produktlinien, Vorlaufzeit („time-to-market", Produkteinführungszeit) und Flexibilität („Änderungsfreundlichkeit").

Beispiel 2.4 *Das folgende Beispiel zeigt einige typische Probleme, die bei einem Entwurf auf Systemebene entstehen. In digitalen Video-Anwendungen ist es oft erforderlich, die nötigen Übertragungsbandbreiten durch eine geeignete Datenkompression zu reduzieren. Das folgende Beispiel ist ein Hybrid-Kodierer, der Transformationskodierung und prädiktive Kodierung kombiniert. Der Kompresionsfaktor einer reinen Bildkodierung wird durch ein prädiktives Schema für Bildfolgen verbessert. Ein Block innerhalb eines Bildes wird geschätzt aus einem Block innerhalb des vorangegangenen Bildes. Abbildung 2.5 zeigt eine Darstellung eines solchen Hybrid-Kodierers auf Verhaltensebene.*

Aus der nun 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 Kosinus-Transformation (DCT) wird auf nichtüberlappende Blöcke des Prädiktionsfehler-Bildes $b[i]$ *angewendet. Die transformierten Blöcke repräsentieren den räumlichen Frequenzinhalt des entsprechenden Blocks. Die nachfolgende Quantisierung (Q) benutzt die räumliche Redundanz innerhalb eines Bildes und die abschliessende Kodierung (RLC) die Dynamik der zu übertragenden Werte. Die Bewegungsschätzung und Bewegungskompensation werden für die Kodierung zwischen aufeinanderfolgenden Bildern einer Sequenz benutzt. Ein Block im Bild* $a[i]$ *wird verglichen mit Nachbarblöcken des vorangegangenen Bildes* $f[i]$ *und hieraus wird ein Bewegungsvektor* $g[i]$ *bestimmt. Als Resultat der Bewegungskompensation erhält man ein geschätztes Bild* $k[i]$*. In der Darstellung nach Abb. 2.5 werden Teilalgorithmen als Blöcke dargestellt. Hier gibt es also noch keine Spezifikation des zeitlichen Ablaufs, der Abbildung auf eine Zielarchitektur, der Speichergrössen, der Partitionierung von Bildern in Blöcke oder Makroblöcke.*

Abb. 2.6 zeigt eine mögliche Systemarchitektur, die aus den Komponenten BUS, CM (Steuerungsprozessor für die Speicherzugriffe und Bus-Arbitrierung), 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.

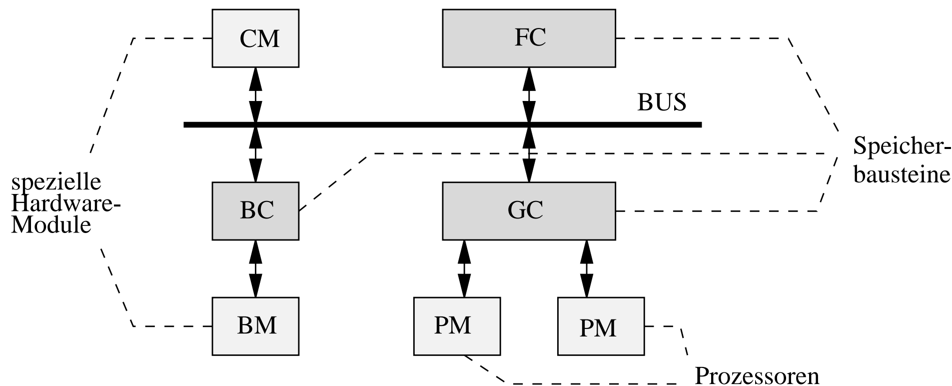


Abbildung 2.6: Strukturelle Sicht einer möglichen Implementierung eines Video-Codecs

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 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 werden 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*). Hierbei sind auch Realisierungen in Hardware und in Software gegeneinander abzuwägen.

Beispiel 2.5 *In Zusammenhang mit dem vorangegangenen Beispiel gibt es nun verschiedene Zielarchitekturen, auf die das gesamte System abgebildet werden kann, z. B.*

- *ein einziger Mikroprozessor, Signal- oder Bildprozessor,*
- *mehrere parallel arbeitende programmierbare Prozessoren,*
- *eine Erweiterung der oben genannten Architekturen mit spezialisierten funktionalen Einheiten, z. B. für die diskrete Kosinus-Transformation oder die Bewegungsschätzung,*
- *eine reine spezialisierte Hardware-Lösung, 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 Kosinus-Transformation berechnen kann (s. Abb. 2.7).

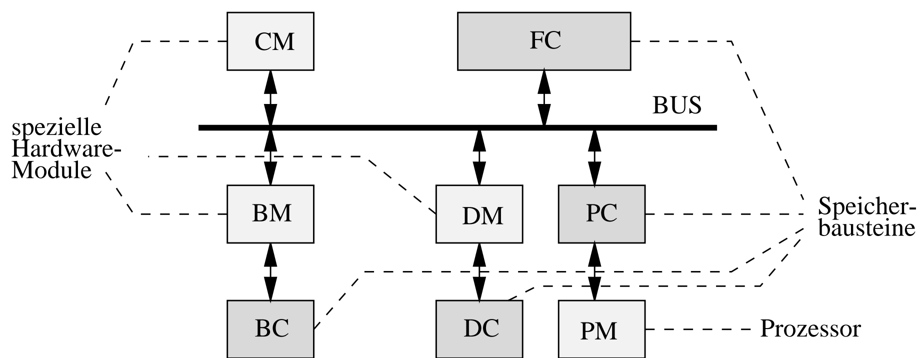


Abbildung 2.7: Strukturelle Sicht auf eine leicht veränderte Implementierung eines Video-Codecs

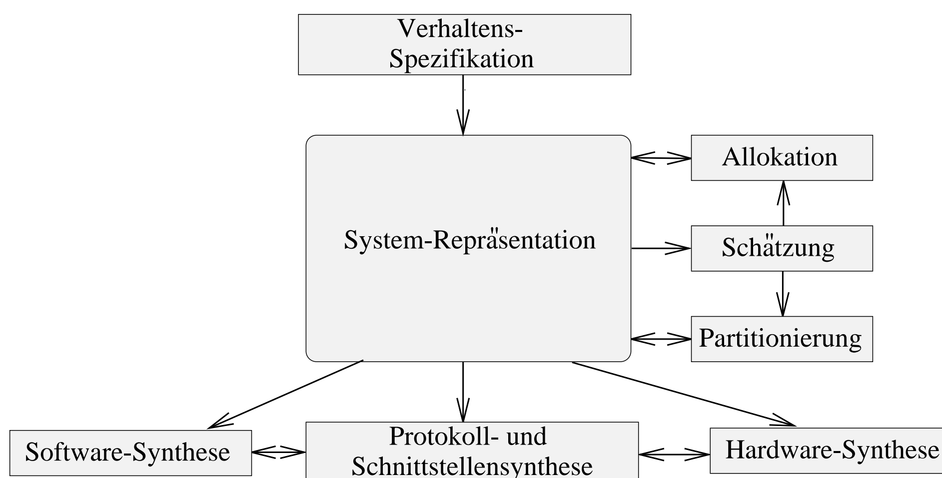


Abbildung 2.8: Grobe Darstellung des Entwurfsablaufs auf Systemebene

Einige der im Rahmen der Synthese auf Systemebene erforderlichen Aufgaben sind in Abbildung 2.8 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 Sprache 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, wie Prozessoren, Speicher und anwendungsspezifische integrierte Schaltungen. Diese Komponenten sind charakterisiert durch Instruktionssatz, Zahl der Instruktionen pro Zeiteinheit (Prozessoren), Zahl der möglichen Gatter, Verzögerungszeiten der Gatter, Leistungsverbrauch (ASICs) und Speichergrösse, Schreib-Lese-Protokolle, Zugriffszeiten (Speicher).

Die Spezifikation wird anschliessend in Hardware- und Software-Komponenten aufgeteilt werden (Hardware-Software-*Partitionierung*). Der Software-Anteil wird also auf einem

oder mehreren der zugewiesenen Prozessoren ausgeführt, der Hardware-Anteil wird mit einer oder mehreren integrierten Schaltungen synthetisiert. Der Software-Anteil kann also weiter in verschiedene Teile zerlegt werden, von denen jeder auf einem eigenen Prozessor abläuft, zum Beispiel auf einem langsamen Prozessor für unkritische Systemteile und Spezialprozessoren für schnelle Datentransformationen.

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 wird ausgewählt.

Nach dieser Auswahl muss die Spezifikation so weit verfeinert werden, dass sie die strukturellen Eigenschaften der Implementierung auf Systemebene charakterisiert. Speziell sind die verschiedenen Teilsysteme den zugewiesenen Systemkomponenten zuzuordnen und die notwendigen Kommunikationskanäle und Protokolle zu spezifizieren.

2.2.3 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 (z. B. Kosten und Verarbeitungsleistung) verbunden.

Aufgabe der Optimierung ist es, den „besten“ Entwurf zu finden, das heisst diejenige Implementierung, die alle Zielfunktionen optimiert. Da unser Optimierungsproblem aber verschiedene Kriterien beinhaltet, sollte man sich die Definition von Optimalität etwas genauer ansehen.

Ein Entwurfspunkt wird *Pareto-Punkt* genannt, falls er von keinem anderen Punkt des Entwurfsraums in allen Eigenschaften übertroffen wird. Im Gegensatz zum Begriff des globalen Optimums gibt es im Allgemeinen mehrere Pareto-Punkte.

Beispiel 2.6 *Das Beispiel der Implementierung eines Video-Kodierers wird hier fortgesetzt. Als mögliche Kriterien (unter vielen anderen!) für eine Exploration des Entwurfsraumes kann man die Integrationsfläche bei einer Implementierung auf einer einzigen integrierten Schaltung betrachten sowie die Verarbeitungsrate. In die Integrationsfläche gehen nicht nur die Zahl und Art der Teilsysteme (allgemein programmierbare Prozessoren, Spezialbausteine) ein, sondern auch die Grösse und Art der Speicher (Bildspeicher, schnelle lokale Cache-Speicher) sowie die Organisation und Breite der Busse.*

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

Abb. 2.9 wird ein kleiner Ausschnitt des Entwurfsraumes gezeigt, der lediglich durch die normierten Kriterien „Integrationsfläche“ und „Verarbeitungszeit pro Bild“ parametrisiert wird. Hier wird deutlich, dass im Sinne der gewählten Kriterien optimale Implementierungen nur Pareto-Punkte sein können. Durch eine Gewichtung der unterschiedlichen Kriterien kann dann eine der entsprechenden Implementierungen ausgewählt werden.

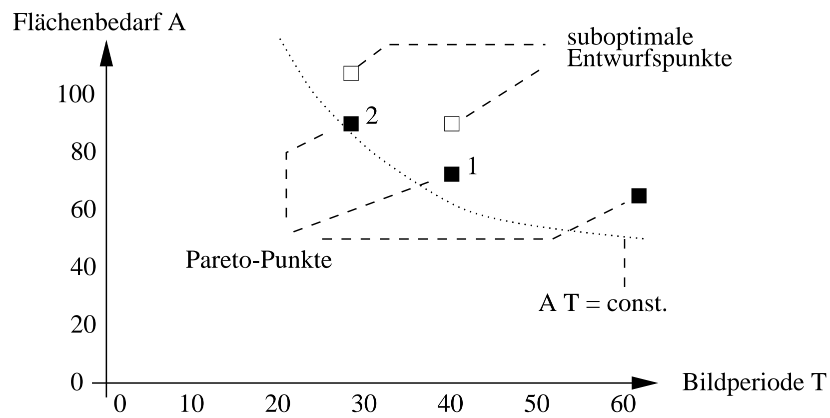


Abbildung 2.9: Beispiel eines Entwurfsraumes mit Pareto-Punkten; Implementierungen 1 und 2 entsprechen den Strukturen von Abb. 2.6 bzw. 2.7

Kapitel 3

Spezifikation und Modellierung

In diesem Kapitel wollen wir uns mit den Aufgaben der Spezifikation und der Modellierung beschäftigen. Dabei sollen die wichtigsten konzeptionellen Modelle für eingebettete Hardware- und Softwaresysteme vorgestellt werden.

3.1 Einleitung

Zu Beginn des Entwurfs eines Systems bedarf es einer Beschreibung der gewünschten Funktionalität. Um von unwichtigen Details eines Systems zu abstrahieren, bedarf es einer sogenannten Modellbildung. Unter *Modellbildung* (=Modellierung) versteht man das Definieren von Basisobjekten und Zusammensetzungsregeln, mit denen die Charakteristika eines Systems beschrieben werden. Diese Objekte und Regeln definieren ein formales System, ein sogenanntes *Modell*. Ein Modell sollte in erster Linie eindeutig und vollständig sein. Weitere wichtige Aspekte sind leichte Verständlichkeit und leichte Änderbarkeit.

Beispiel 3.1 *Abb. 3.1 zeigt unterschiedliche Modelle zur Spezifikation einer Fahrstuhlsteuerung, insbesondere eine Verhaltensbeschreibung in natürlicher Sprache a), eine algorithmische Beschreibung mit einer Prozedur b) und durch ein Zustandsdiagramm (Mealy-Automatenmodell).*

Verschiedene Modelle können dazu dienen, verschiedene *Sichten* eines Systems darzustellen. Sie betonen dabei verschiedene Charakteristika eines Systems (z. B. Verhalten und Struktur). Ein System *spezifiziert* man in einem bestimmten Modell. Die Spezifikation kann entweder graphisch oder sprachlich erfolgen. Es sollte hier betont werden, dass eine Sprache (z. B. C, VHDL) im Allgemeinen verschiedene Sichten repräsentieren kann genauso wie eine Sicht in verschiedenen Sprachen formuliert werden kann. Für den Entwerfer eines Systems kann es nützlich sein, in verschiedenen Entwurfsphasen unterschiedliche Modelle benutzen zu können. Genauso ist es oft nützlich, Teilaspekte eines Systems in verschiedenen Modellen zu beschreiben. Dazu benötigt man *heterogene Modelle*.

Nach der Modellbildung geht es darum, das Modell zu verfeinern und eine Implementierung zu finden, vergleiche Abschnitt 2.2.2. Diese Beschreibung umfasst die Anzahl und die Typen der Systemkomponenten sowie deren Verbindungsstruktur (Architekturebene) oder den Instruktionssatz und das Programm- und Speichermodell eines Mikroprozessors (Blockebene), siehe Kapitel 2.2.

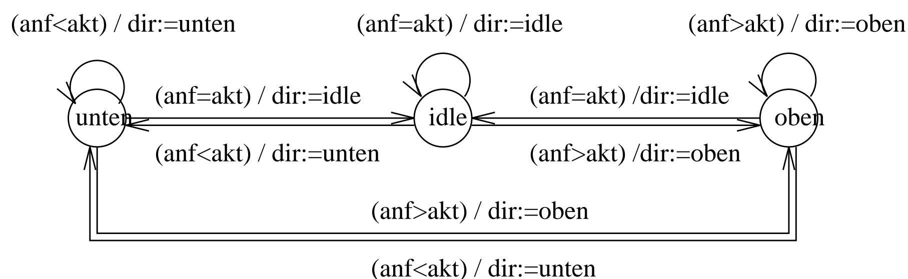
Im Folgenden wollen wir Modelle klassifizieren, die heutzutage in Hardware- und Softwareentwurf von Systemen eine wichtige Rolle spielen.

„Steht der Fahrstuhl und ist die angefragte Etage gleich der aktuellen Etage, dann tue nichts.
Steht der Fahrstuhl und ist die angefragte Etage niedriger als die aktuelle Etage, dann bewege den Fahrstuhl nach unten zur angefragten Etage.
Steht der Fahrstuhl und ist die angefragte Etage weiter oben als die aktuelle Etage, dann bewege den Fahrstuhl nach oben zur angefragten Etage.‘‘

(a)

```
LOOP
 IF anf = akt THEN
  dir:=idle;
 ELSIF anf < akt THEN
  dir:=unten;
 ELSIF anf > akt THEN
  dir:=oben;
 END IF;
END LOOP;
```

(b)



(c)

Abbildung 3.1: Modelle zur Spezifikation einer Fahrstuhlsteuerung: a) natürliche Sprachbeschreibung, b) Algorithmus und c) Zustandsdiagramm (Mealy-Automatenmodell).

3.2 Klassifikation von Modellen

Im Folgenden treffen wir eine Klassifikation von Modellen, die orthogonal zu der bereits eingeführten Klassifikation in Abstraktionsebenen (System, Architektur und Logik) und Sichtweisen (insb. Verhalten und Struktur, siehe Kapitel 2.2) zu verstehen ist. Modelle sind

- *zustandsorientiert*,
- *aktivitätsorientiert*,
- *strukturorientiert*,
- *zeitorientiert*,
- *datenorientiert*.

Ein *zustandsorientiertes* Modell, wie beispielsweise das Modell des endlichen Automaten (im Weiteren FSM (=finite state machine) genannt), repräsentiert ein System als eine Menge von Zuständen und Zustandstransitionen. Ein zustandsorientiertes Modell ist am besten zur Modellierung von steuerungsdominanten Problemen geeignet wie sie beispielsweise in reaktiven Echtzeitsystemen vorkommen. Die Orthogonalität der hier eingeführten Klassifikation zu Architekturebene und Sichtweise soll an einem Beispiel deutlich gemacht werden. Auf Systemebene können die Aktionen von Zuständen bzw. Zustandsübergängen eines Zustandsmodells komplexerer Natur sein (z. B. Tasks). Auf der Logikebene sind dies Boole'sche Gleichungen. Während ein Zustandsdiagramm üblicherweise eine Verhaltenssicht darstellt, entspricht ein Blockschaltbild eines Steuerwerks einer strukturellen Sichtweise.

Als Beispiel eines *aktivitätsorientierten* Modells werden wir Datenflussgraphen kennenlernen. In einem Datenflussgraphen werden die Aktivitäten und deren Datenabhängigkeiten und Ausführungsabhängigkeiten dargestellt. Dieses Modell ist am besten zur Modellierung *transformationaler Systeme* geeignet wie sie beispielsweise in der digitalen Signalverarbeitung vorkommen. Dort werden Daten bei meist fester Datenrate einer Reihe von Transformationen unterworfen. Strukturorientierte Modelle, wie beispielsweise Blockschaltbilder, dienen der Beschreibung der physikalischen Komposition des Systems (physikalische Module und Verbindungen).

Datenorientierte Modelle beschreiben ein System als eine Kollektion von Datenobjekten mit Attributen und Relationen. Diese Modelle werden meist in Informationssystemen (z. B. Datenbanksystemen) angewendet. Dort ist die Beschreibung der Datenorganisation meist wichtiger als deren Funktionalität.

Wir werden sehen, dass auf Systemebene *heterogene* Modelle eine grosse Rolle spielen. Verschiedene Sichten eines Systems können mit heterogenen Modellen gleichzeitig dargestellt werden.

Im Folgenden werden verschiedene Modelle vorgestellt, die im Systementwurf weit verbreitet sind. Viele dieser Modelle basieren auf dem Grundmodell von *Petri-Netzen*. Wir werden deshalb zunächst diese wichtige Klasse von Modellen einführen. Verschiedenartige Modelle ergeben sich dann durch unterschiedliche *Interpretation* eines Petri-Netzes oder durch spezielle Eigenschaften von Petri-Netzen. Nach einer allgemeinen Einführung von Petri-Netzen werden wir deren spezielle Eigenschaften definieren und die für den Systementwurf wichtigsten interpretierten Petri-Netz-Typen kennenlernen.

3.3 Petri-Netz-Modell

Ein *Petri-Netz* (nach dem Mathematiker C. A. Petri [7] benannt), ist ein formales Modell zur Beschreibung von Systemen mit dem Schwerpunkt der Modellierung von asynchronen, nebenläufigen Vorgängen. Petri-Netze werden benutzt zur Modellierung von Systemen im Hardwareentwurf (insbesondere asynchroner Schaltungen), in Softwaresystemen sowie zur Beschreibung von Ablaufplänen. Abb. 3.2 zeigt ein Beispiel eines Netzgraphen, einer graphischen Darstellung eines Petri-Netzes. Ein Petri-Netz besteht aus einem Netzgraphen und einer Dynamisierungsvorschrift. Wir werden allerdings im Weiteren nicht unterscheiden zwischen einem Petri-Netz und seinem Netzgraphen. Formal definiert man ein Petri-Netz wie folgt:

Definition 3.1 *Ein Petri-Netz ist ein 6-Tupel* $G = (P, T, F, K, W, M_0)$ *mit*

$$P \cap T = \emptyset$$

und

$$F \subseteq (P \times T) \cup (T \times P)$$

Darin bedeuten:

- $P = \{p_1, p_2, \cdots, p_m\}$ *die Menge der* Plätze *(oder Stellen),*
- $T = \{t_1, t_2, \cdots, t_n\}$ *die Menge der* Transitionen*,*

Beide Mengen zusammen bilden die Menge von Knoten.

- F *heisst* Flussrelation*. Die Elemente von* F *heissen* Kanten*.*
- $K : P \rightarrow \mathbf{N} \cup \{\infty\}$ *(*Kapazitäten *der Stellen - evtl. unbeschränkt),*
- $W : F \rightarrow \mathbf{N}$ *(*Kantengewichte *der Kanten),*
- $M_0 : P \rightarrow \mathbf{N_0}$ Anfangsmarkierung*, wobei* $\forall p \in P : M_0(p) \leq K(p)$*.*

Ein Petri-Netz ist also ein *gerichteter bipartiter Graph.* In der üblichen Darstellung werden die Stellen als Kreise, die Transitionen als Rechtecke und die Kanten als Pfeile notiert. Im Zusammenhang mit einem Netzknoten x hat man häufig mit zwei bestimmten Mengen von Nachbarknoten zu tun. Dies ist zum einen der *Vorbereich*

$$\bullet x \ = \ \{y | (y, x) \in F\}$$

aller Eingangsknoten (zum Beispiel *Eingangsstellen*, falls x Transitionen sind). Zum anderen ist dies der *Nachbereich* aller Ausgangs- oder Outputknoten von x:

$$x \bullet \ = \ \{y | (x, y) \in F\}$$

Beispiel 3.2 *In Abbildung 3.2 sind ebenfalls die Vor- und Nachbereiche der Transitionen dargestellt sowie die Anfangsmarkierung* M_0*. Es gibt 12 Kanten (*$F = \{f_1, f_2, \cdots, f_{12}\}$*), zum Beispiel ist* $f_1 = (p_1, t_1)$*. In der graphischen Darstellung wird die Markenzahl* $M(p)$ *einer Stelle* $p \in P$ *unter der Markierung* M *durch* $M(p)$ *Punkte dargestellt, siehe z. B. Stelle* p_3*. Kapazitäten* $\neq \infty$ *bzw. Gewichte* $\neq 1$ *werden an die betreffenden Stellen bzw. Pfeile geschrieben. Die Kapazitäten aller Stellen in Abb. 3.2 sind also unbeschränkt, die Gewichte aller Kanten* $= 1$*.*

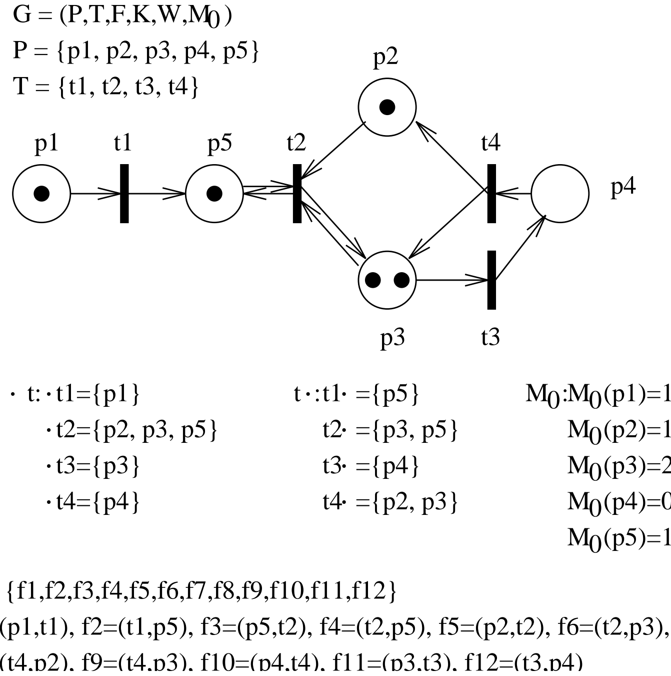


Abbildung 3.2: Beispiel eines Petri-Netzes G.

Marken sind Attribute von Stellen und zirkulieren im Petri-Netz. Der Zustand eines Netzes wird über die *Markierungsfunktion* $M : P \rightarrow \mathbf{N_0}$ (im Weiteren oft einfach als *Markierung* bezeichnet) definiert, wobei $M(p)$ die Zahl der Marken einer Stelle $p \in P$ darstellt. $\mathbf{N_0}$ bezeichne die Menge der natürlichen Zahlen inklusive der Zahl 0.

Es fehlt nun noch eine *Dynamisierungsvorschrift*, die angibt, unter welchen Umständen Marken im Petri-Netz bewegt werden können. Jeweils zulässige Markierungswechsel ergeben sich aus der sogenannten *Schaltregel* (firing rule), worin die *Schaltbereitschaft* und das *Schalten* einer Transition wie folgt definiert werden:

Definition 3.2 *Eine Transition $t \in T$ eines Petri-Netzes $G = (P, T, F, K, W, M_0)$ heisst* schaltbereit *(unter der Markierung M), geschrieben $M[t\rangle$, wenn*

1. $\forall p \in \bullet t \setminus t\bullet : M(p) \geq W(p, t)$;
2. $\forall p \in t \bullet \setminus \bullet t : M(p) \leq K(p) - W(t, p)$;
3. $\forall p \in t \bullet \cap \bullet t : M(p) \leq K(p) - W(t, p) + W(p, t)$.

Damit eine Transition *schalten* (oder feuern) kann, muss sie schaltbereit sein. Das Schalten führt zu dem Ergebnis, dass von jeder Eingangsstelle $p \in \bullet t$ genau $W(p, t)$ Marken abgezogen (oder konsumiert) werden und jeweils $W(t, p)$ Marken an Ausgangsstellen $p \in t\bullet$ gelegt (sprich produziert) werden:

Definition 3.3 *Eine Transition $t \in T$ eines Petri-Netzes $G = (P, T, F, K, W, M_0)$* schaltet *von M auf M' geschrieben $M[t\rangle M'$, wenn t unter M schaltbereit ist und M' aus M wie folgt gebildet wird.*

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

M' heisst Folgemarkierung von M unter t. Das Entfernen der Marken der Eingangsstellen und das Produzieren der Marken an den Ausgangsstellen der Transition erfolgt simultan. Die Bedingungen an die Schaltbereitschaft und die Definition der Folgemarkierung bezeichnet man auch als *Schaltregel*.

Beispiel 3.3 *In dem Petri-Netz in Abb. 3.2 ist Transition t_2 schaltbereit. Nach Feuern von t_2 ist die neue Markierung M' mit $M'(p1) = M'(p5) = 1$, $M'(p2) = M'(p4) = 0$ und $M'(p3) = 2$.*

Petri-Netze sind weitverbreitet zur Modellierung dynamischer zustandsdiskreter Systeme. Zunächst betrachten wir nur Petri-Netze mit unbeschränkter Kapazität und Einheitsgewichten ($K = \infty, W = 1$). Abb. 3.3a zeigt, wie man *Sequentialität* mit Hilfe eines Petri-Netzes modellieren kann. Abb. 3.3b stellt eine nichtdeterministische Verzweigung dar. Transitionen t_1 und t_2 sind in einem *Konflikt*. Ein Konflikt liegt dann vor, wenn zwei oder mehrere Transitionen schaltbereit sind, die mindestens eine gemeinsame Eingangsstelle haben und das Schalten der einen Transition die Schaltbereitschaft der anderen zerstört. Auch Abb. 3.3d stellt einen Konflikt dar. Transitionen t_1 und t_2 streiten sich um eine Ressource, die durch die Marke der mittleren Stelle dargestellt ist.

Definition 3.4 *Zwei Transitionen t_1 und t_2 eines Petri-Netzes mit $K = \infty$ sind im Konflikt, wenn $M[t_1\rangle$ und $M[t_2\rangle$ aber nicht $M[\{t_1, t_2\}\rangle$. Dabei bezeichne $M[\{t_1, t_2, \cdots, t_n\}\rangle$ die gleichzeitige, nebenläufige Schaltbereitschaft aller Transitionen $t_1, t_2, \cdots, t_n$.*

In Abb. 3.3c ist *Synchronisation* mit Hilfe eines Petri-Netzes modelliert.

Die Modellierung von Nebenläufigkeit wird in Abb. 3.3e deutlich. Transitionen t_2 und t_3 können simultan feuern. Dieses Petri-Netz stellt ein Erzeuger-Verbraucher-System (producer–consumer) dar. Die Transitionen können z. B. betriebliche Vorgänge modellieren.

3.3.1 Dynamische Eigenschaften von Petri-Netzen

Im Folgenden werden wir spezielle Eigenschaften von Petri-Netzen definieren. Diese Eigenschaften erlauben die Klassifikation bekannter Modelle als Teilklassen von Petri-Netzen. Die wichtigsten Eigenschaften sind Sicherheit und Lebendigkeit.

Sicherheit

Zu jeder Markierung M eines Petri-Netzes gehört eine *Markierungsklasse*. Dies ist diejenige Menge von Markierungen, die neben M selbst noch alle diejenigen Markierungen enthält, die ausgehend von M durch Schaltfolgen erreichbar sind. Die Markierungsklasse der Anfangsmarkierung M_0 eines Petri-Netzes G heisst auch *Erreichbarkeitsmenge* von G, geschrieben $[M_0\rangle$. Damit können wir die Sicherheit eines Petri-Netzes wie folgt definieren:

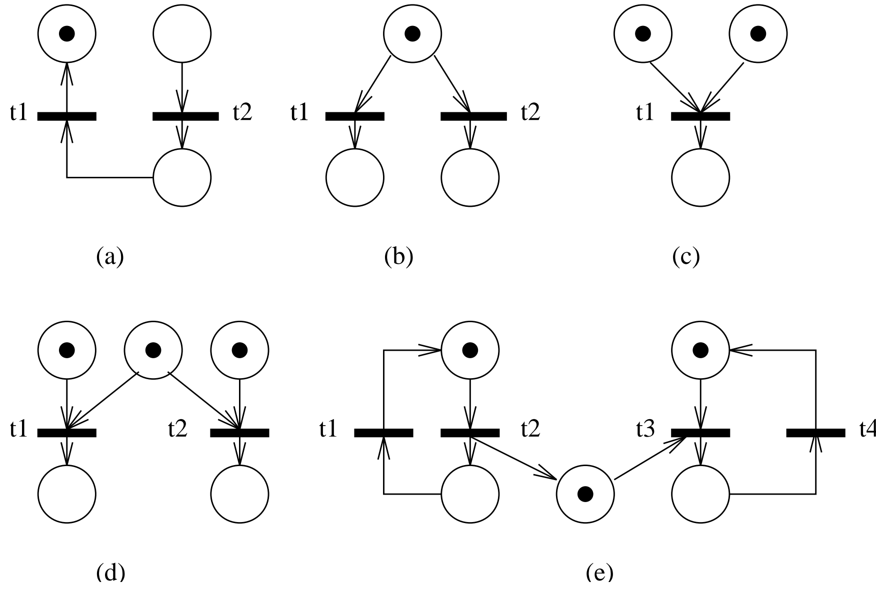


Abbildung 3.3: Petri-Netze, die (a) Sequentialität, (b) Verzweigung, (c) Synchronisation, (d) Ressourcenkonflikt und (e) Nebenläufigkeit modellieren.

Definition 3.5 *Seien $G = (P, T, F, K, W, M_0)$ ein Petri-Netz und $B : P \to \mathbf{N_0} \cup \{\infty\}$ eine Abbildung, die jeder Stelle eine 'kritische Markenzahl' zuordnet. G heisst B-*sicher *bzw. B-*beschränkt*, wenn*

$$\forall M \in [M_0\rangle, p \in P : M(p) \leq B(p).$$

G heisst beschränkt*, wenn es eine natürliche Zahl b gibt, für die G b-beschränkt ist.*

Während Kapazität eine Begrenzung der Markenzahlen darstellt, die *a priori* gegeben ist, stellt Sicherheit eine Begrenzung *a posteriori* dar, eine Beobachtung. Interessant bei der Analyse von durch Petri-Netzen beschriebenen Systemen ist der wichtige Satz:

Theorem 3.1 ([8]) *Ein Petri-Netz ist genau dann beschränkt, wenn seine Erreichbarkeitsmenge endlich ist.*

Beweis: Sei $|P| = m$. Ist das System beschränkt, dann ist es auch für ein $b \in \mathbf{N}$ b-sicher, so dass es nur $(b+1)^m$ verschiedene erreichbare Markierungen geben kann. Ist das System hingegen unbeschränkt, so ist die Markenzahl auf mindestens einer Stelle p_0 unbeschränkt; sonst wäre das System b-beschränkt mit $b = \max\{M(p)|M \in [M_0\rangle, p \in P\}$. Somit gibt es unendlich viele verschiedene auf p_0 angenommene Markenzahlen, und dazu gehören unendlich viele verschiedene erreichbare Markierungen.

Lebendigkeit

Definition 3.6 *Eine Transition t eines Petri-Netzes $G = (P, T, F, K, W, M_0)$ heisst* tot*, wenn sie unter keiner erreichbaren Markierung schaltbereit ist:*

$$\forall M \in [M_0\rangle : \neg M[t\rangle.$$

Definition 3.7 *Eine Transition t eines Petri-Netzes $G = (P, T, F, K, W, M_0)$ heisst* aktivierbar*, wenn sie mindestens unter einer Folgemarkierung schaltbereit ist:*

$$\exists M_1 \in [M_0\rangle : M_1[t\rangle.$$

Sie heisst lebendig*, wenn sie unter allen Folgemarkierungen aktivierbar ist:*

$$\forall M_1 \in [M_0\rangle : \exists M_2 \in [M_1\rangle : M_2[t\rangle.$$

Wir beachten, dass in dieser Definition tot nicht das Gegenteil von lebendig ist. Aufbauend auf dem Lebendigkeitsbegriff für Transitionen definiert man die Lebendigkeit eines Petri-Netzes wie folgt:

Definition 3.8 *Ein Petri-Netz $G = (P, T, F, K, W, M_0)$ heisst* tot *in einer Markierung M, wenn alle seine Transitionen tot sind:*

$$\forall t \in T : \neg M[t\rangle.$$

G heisst verklemmungsfrei *oder* schwach lebendig*, wenn es unter keiner Folgemarkierung tot ist:*

$$\forall M_1 \in [M_0\rangle : \exists t \in T : M_1[t\rangle.$$

G heisst lebendig *oder* stark lebendig*, wenn alle seine Transitionen lebendig sind:*

$$\forall t \in T, M_1 \in [M_0\rangle : \exists M_2 \in [M_1\rangle : M_2[t\rangle.$$

Das Petri-Netz in Abb. 3.3e) ist stark lebendig. Eng verbunden mit dem Begriff der toten Transition ist der Begriff der *Verklemmung* (Deadlock) ($\exists M_1 \in [M_0\rangle : \forall t \in T : \neg M_1[t\rangle$). Der Begriff der Verklemmung ist vor allem dann von Bedeutung, wenn das Petri-Netz eine Verhaltensspezifikation für ein zu konstruierendes System darstellt, bei dem Verklemmungen als Störfälle einzustufen sind.

Ein Petri-Netz heisst schliesslich *konservativ*, wenn die Anzahl der Marken im Netz konstant ist unter allen Schaltfolgen von Transitionen. Das impliziert, dass alle Transitionen eines konservativen Petri-Netzes konservativ sein müssen, was wiederum bedeutet, dass in einem konservativen Petri-Netz bei jeder Transition die Summe der Gewichte aller Eingangsstellen gleich der Summe der Gewichte aller Ausgangsstellen ist.

Definition 3.9 *Sei $w : P \rightarrow \mathbf{N_0}$. Ein Petri-Netz $G = (P, T, F, K, W, M_0)$ heisst* konservativ*, wenn es für einen strikt positiven Vektor w die Bedingung*

$$\forall M \in [M_0\rangle : wM = wM_0$$

erfüllt, wobei

$$wM := \sum_{p \in P} w(p)M(p).$$

Während wir bislang den Stellen und Transitionen eines Petri-Netzes keine Bedeutung zugeordnet haben (Petri-Netze sind sog. *uninterpretierte Netze*), spielen in der Realität spezielle *interpretierte Netze* eine wichtige Rolle. Modelle, die entweder zustandsorientiert, aktionsorientiert oder datenflussorientiert sind, ergeben sich als spezielle Netztypen.

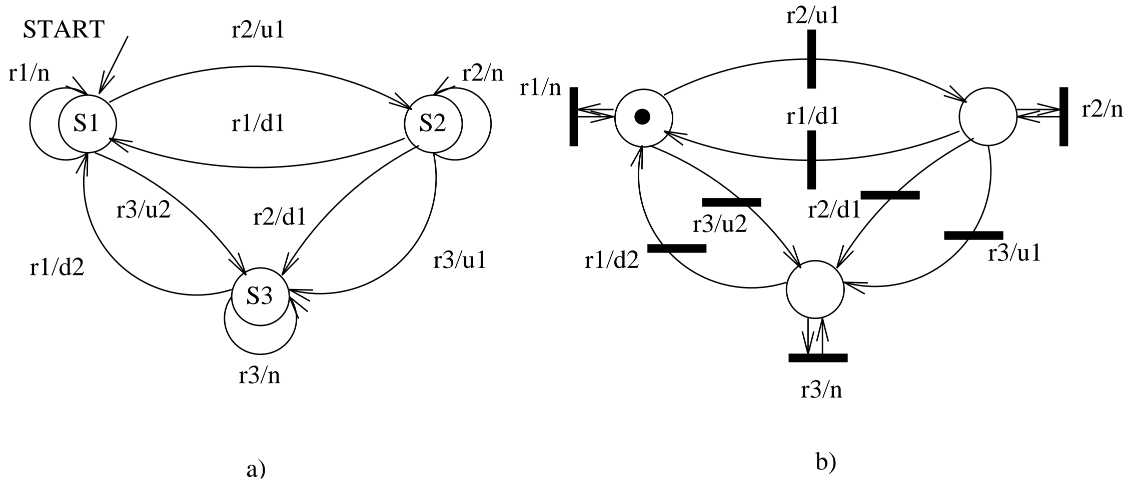


Abbildung 3.4: Zustandsdiagramm und äquivalentes Petri-Netz-Modell für eine Fahrstuhlsteuerung.

3.4 Zustandsorientierte Modelle

3.4.1 Endliche Automaten (FSM)

Endliche Automaten spezifiziert man häufig über Zustandsdiagramme (siehe z. B. Abb. 3.4a). Wir wollen nun zeigen, dass Zustandsdiagramme auch mit Petri-Netzen dargestellt werden können. Man kann zeigen, dass Petri-Netze, in denen jede Transition genau einen Eingang und genau einen Ausgang besitzt und die mit genau einer Marke auf einer der Stellen anfangsmarkiert sind, äquivalent sind mit dem Modell des *(nichtdeterministischen) endlichen Automaten.*

Definition 3.10 *Ein nichtdeterministischer endlicher Automat ist ein Petri-Netz* $G = (P, T, F, K, W, M_0)$ *mit den Eigenschaften:*

1. $\forall t \in T : |\bullet t| = |t \bullet| = 1,$
2. $\sum_{p \in P} M_0(p) = 1,$
3. $K = W = 1,$
4. *Jeder Transition ist ein Prädikat zugewiesen, das eine Bedingung beschreibt, ohne deren Erfüllung die Transition nicht schalten kann (Zustandsübergangsbedingung).*

Dies wird aus folgender Überlegung deutlich: Sei der *Zustand* eines Petri-Netzes gleich seiner Markierung, dann ist ein Petri-Netz mit genannter Eigenschaft offensichtlich konservativ und damit die Menge erreichbarer Markierungen (= Zustandsraum) endlich. Es gibt genau einen Anfangszustand (es gibt genau eine Stelle, die eine Marke besitzt). Ein Zustandsdiagramm (siehe Abb. 3.4a) konvertiert man in ein Petri-Netz, indem man jeden Zustand durch eine Stelle und jeden Zustandsübergang durch eine entsprechende Stellen verbindende Transition modelliert (siehe Abb. 3.4b). Zustandsdiagramme sind also interpretierte Petri-Netze und bilden ein zustandsorientiertes Modell.

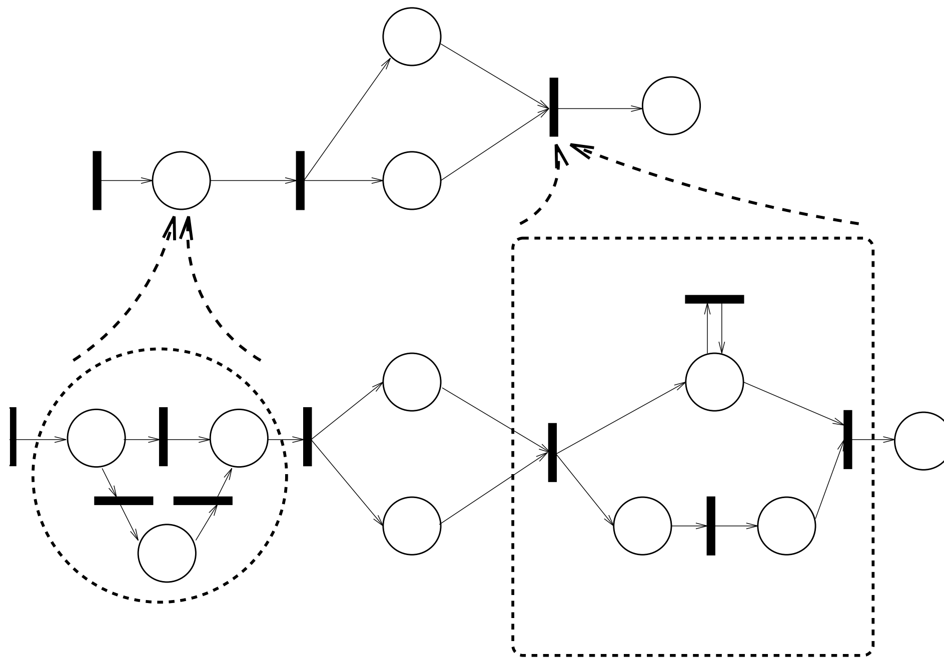


Abbildung 3.5: Hierarchische Modellierung in Petri-Netzen durch Ersetzung der Stellen oder Transitionen durch Subnetze (oder umgekehrt).

3.4.2 Erweiterte Zustandsmaschinenmodelle

Eine wichtige Anforderung bei der Modellierung komplexer Systeme ist die Möglichkeit zur Einführung von Hierarchieebenen. Hierarchie ist ein geeignetes Mittel zur Reduktion der Entwurfskomplexität und schafft effizientere Beschreibungsformen. Im Folgenden wollen wir zeigen, wie man Hierarchie im Petri-Netz-Modell darstellen kann. Beispielsweise können Stellen oder Transitionen durch Teilnetze ersetzt werden. Ein Beispiel ist in Abb. 3.5 dargestellt. Falls verschiedene, nebenläufige Transitionen auf das gleiche Subnetz verweisen, dann ist die Konvention, dass die beiden Teilnetze auch nebenläufig ablaufen. Es werden also nebenläufige Instanzen kreiert. Allerdings ist die Semantik hierarchischer Petri-Netze bislang unklar.

Das Modell des endlichen Automaten ist in vielerlei Hinsicht restriktiv, z. B. können keine nebenläufigen Zustände dargestellt werden. Ansätze, die das Modell des endlichen Automaten erweitern, sind beispielsweise kommunizierende endliche Automaten (z. B. in der Sprache SDL). Nebenläufigkeit findet hier genau auf einer Hierarchieebene statt. Andere Ansätze sind Prozesskalküle. Milners CCS [9] (A calculus of communicating systems) ist eine formale Sprache, die auch die Darstellung verschachtelter Hierarchien auf der Ebene von Prozessen zulässt. CCS oder CSP [10] (Communicating sequential processes) erlauben Kommunikation von Prozessen z. B. über einen *Rendezvous*-Mechanismus (CSP), der den sendenden Prozess blockiert, so lange der Empfänger nicht bereit ist, die Mitteilung entgegen zu nehmen (siehe Technische Informatik II). Auch sei in diesem Zusammenhang die Methode CIP erwähnt, die in der Vorlesung Prozessrechentechnik vorgestellt wird. CIP erlaubt sowohl asynchrone als auch synchrone Kommunikation zwischen Zuständen verschiedener Hierarchieebenen. Ein anderes wichtiges Kommunikationsschema ist der sogenannte *Broadcast*, bei dem der Sender nicht blockiert wird und die Meldung an alle

anderen Prozesse gleichzeitig gesendet wird.

3.4.3 Hierarchische, nebenläufige Zustandsmaschinen

Ein zur Spezifikation von eingebetteten Systemen häufig verwendetes zustandsorientiertes Modell, welches die Eigenschaften der Nebenläufigkeit und Hierarchie dem Modell der Zustandsdiagramme hinzufügt, ist das Modell des *hierarchischen, nebenläufigen endlichen Automaten* (HCFSM = hierarchical concurrent finite-state machine). Eine Implementierung des Modells des hierarchischen nebenläufigen endlichen Automaten mit graphischer Eingabe sind Statecharts [11].

3.4.4 Statecharts

Wir wollen im Folgenden die Eigenschaften von Statecharts näher kennenlernen.

$$\begin{aligned}\text{Statecharts} &= \text{Zustandsdiagramme} + \\ &\quad \text{Hierarchie} + \\ &\quad \text{NebenläufigeZustände} + \\ &\quad \text{Broadcastkommunikation}\end{aligned}$$

Es liegt uns fern, Statecharts hier in vollem Umfang inklusive einer Beschreibung der formalen Semantik einzuführen. Statt dessen wollen wir die wichtigsten Konzepte informell vorstellen.

Hierarchische Verfeinerung

Zustände werden durch abgerundete Rechtecke (Boxen) dargestellt. Pfeile bezeichnen Zustandsübergänge. Hierarchie wird durch Enthaltensein von Zuständen in anderen Zuständen dargestellt. Zustände einer höheren Hierarchiestufe werden als *Superzustände* bezeichnet. Innerhalb eines Superzustands bezeichnet man alle Zustände als *Subzustände*. Die Aktivierung eines Superzustands entspricht dem Eintrit in einen seiner Subzustände. Die Semantik eines Superzustands ist dabei *eXclusiv-OR (Dekomposition)* und ein Superzustand ist eine Abstraktion seiner Subzustände. Ein Zustandübergang kann bei Zuständen unterschiedlicher Hierarchieebene beginnen bzw. enden. Ein Zustandübergang wird gekennzeichnet mit einem ihn auslösenden Ereignis α und optional mit einer Kondition K, geschrieben $\alpha(K)$, wobei K ein Boole'scher Ausdruck ist. Ein Zustandsübergang bei Vorliegen des Ereignisses α erfolgt nur dann, wenn K wahr ist. Die Notation soll an einem Beispiel deutlicher gemacht werden.

Beispiel 3.4 *Abb. 3.6a zeigt einen Statechart, bestehend aus einem Superzustand D, der die beiden Subzustände A und C beinhaltet. Die Semantik von Zustand D ist ein XOR seiner Subzustände, d. h. im Zustand D zu sein bedeutet entweder in A oder in C zu sein, aber nie in beiden gleichzeitig. Der Übergang β bedeutet, dass unabhängig davon, ob die Maschine sich in Zustand A oder C befindet, sie beim Auftreten von Ereignis β in den Zustand B wechselt. Es handelt sich folglich um eine Zusammenfassung von Kanten. Ein Zustandswechsel von A nach C erfolgt aus Zustand A bei Ereignis γ nur dann, wenn die Bedingung K wahr ist. Anfangszustände sind durch besondere Pfeile gekennzeichnet (siehe Abb. 3.6b) und entsprechen den Anfangszuständen in herkömmlichen FSM.*

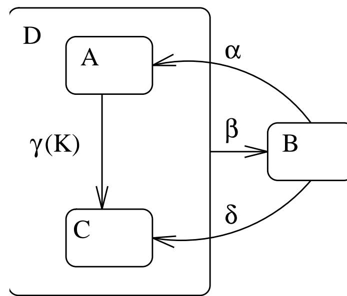


Abbildung 3.6: Hierarchie in Statecharts.

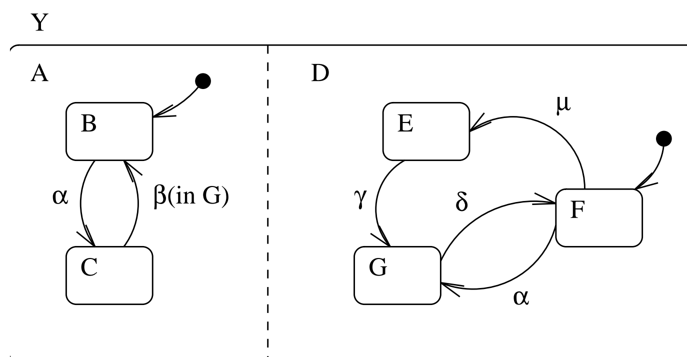


Abbildung 3.7: Modellierung von nebenläufigen Zuständen in Statecharts.

Nebenläufigkeit und Unabhängigkeit

Die Einführung von Nebenläufigkeit erfolgt in Statecharts durch Definition der sogenannten *AND-Dekomposition*. Befindet sich ein System in einem Subzustand eines Superzustands, muss es sich auch in einem Subzustand all seiner AND-Komponenten befinden. In der Statecharts-Notation wird ein Diagramm physisch durch gestrichelte Linien in AND-Komponenten unterteilt. Der Superzustand ist dann das *orthogonale Produkt* all seiner Subzustände (siehe Abb. 3.7). Schliesslich wird ein Superzustand verlassen, wenn ein Zustandsübergang irgendeiner seiner Subzustände zum Verlassen des Superzustands führt. Das Ganze wollen wir wieder an einem Beispiel erläutern.

Beispiel 3.5 *Abb. 3.7 zeigt, wie Nebenläufigkeit im Statecharts-Modell dargestellt wird. In Zustand Y zu sein bedeutet in Zustand A und Zustand D gleichzeitig zu sein, also in einer Kombination von B oder C mit E, F oder G. Y ist das orthogonale Produkt von A und D. Der Eintritt in den Superzustand Y entspricht dem Eintritt in die beiden Subzustände B und F, man schreibt (B, F) und bezeichnet (B, F) auch als Konfiguration. Grundsätzlich sind die einzelnen Komponenten eines Superzustands unabhängig voneinander. Verwenden die sogenannten Subdiagramme jedoch gleiche Ereignisse bei Zustandsübergängen, müssen die beiden Teilautomaten in diesen Ereignissen* synchronisieren. *Beispielsweise erfolgt aus der Konfiguration (B, F) bei Auftreten von Ereignis α ein Zustandswechsel in die Konfiguration (C, G), d. h. das Ereignis α synchronisiert den Zustandswechsel. Erfolgt Ereignis μ in Konfiguration (B, F), dann ist nur der Superzustand D von einem Zustandswechsel betroffen (Unabhängigkeit). Das Statechart in Abb. 3.7 ist äquivalent zu dem Zustandsdiagramm in Abb. 3.8. Die Abbildung erfolgt durch Bilden des Produktautomaten. Betrachtet man die Zustandsmaschine A mit $|A| = 2$ Zuständen und D mit $|D| = 3$ Zuständen,*

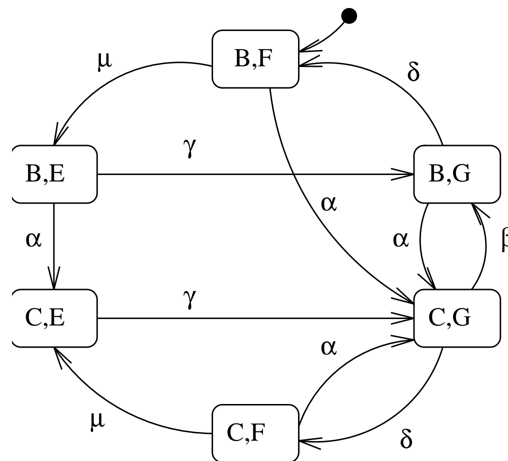


Abbildung 3.8: Produktautomat des Statecharts in Abb. 3.7.

so hat der Produktautomat $|A| \cdot |D| = 6$ *Zustände, nämlich genau die Anzahl aller möglichen Konfigurationen seiner Zustände. Formell ist das orthogonale Produkt in Statecharts eine Verallgemeinerung der üblichen Produktautomatenbildung, denn es kann hier Abhängigkeiten zwischen den orthogonalen Komponenten geben durch a) gemeinsame Ereignisse (üblich) und b) durch Konditionen. Zum Beispiel hat die Bedingung (in G) in dem Statechart in Abb. 3.7 die Konsequenz, dass es im Produktautomaten lediglich einen Übergang mit Ereignis* β *gibt, denn der Übergang bedeutet, dass im Produktautomat ein Übergang aus der Konfiguration* (C, G) *durch* β *ausgelöst werden kann, nicht aber in den Konfigurationen* (C, E) *und* (C, F)*.*

Bislang wurde nur das Zustandsübergangsverhalten in Statecharts beschrieben. Was noch fehlt, ist offensichtlich die Beschreibung vom Ausgangsverhalten. In Statecharts erfolgt dies durch sogenannte *Aktionen*. Ein allgemeiner Zustandsübergang ist der Form $\alpha(P)/A$, wobei A eine Aktion darstellt. Aktionen können externe Ereignisse sein, aber auch interne Ereignisse, die in orthogonalen Komponenten andere Zustandsänderungen auslösen können.

Beispiel 3.6 *Abb. 3.9 stellt unterschiedliche Zustandsübergänge in Statecharts dar. Zu Abb. 3.9a: Ein Ereignis kann elementar sein, der Wahrheitswert einer Kondition sowie disjunktive und konjunktive Verknüpfungen von Ereignissen. Zu Abb. 3.9b: Eine Kondition kann ein Vergleichsausdruck sein. Eine besondere Kondition ist* $\mathrm{in}(S)$*, wobei* S *ein Zustand ist. Auch hier sind disjunktive und konjunktive Verknüpfungen von Konditionen gültige Konditionen. Zu Abb. 3.9c: Eine Aktion ist beispielsweise die Zuweisung eines Ausdrucks an eine Variable oder eine Kondition, deren Wahrheitswert als ein Ereignis andere Zustandsänderungen auslösen kann.*

Beiläufig sei noch erwähnt, dass Statecharts auch die Fähigkeit besitzen, sich beim Verlassen von Zuständen den verlassenen Zustand zu merken (sog. History-Mechanismus). Ferner können auch Zeitbedingungen spezifiziert werden über die minimalen und maximalen Verweildauern in Zuständen (sog. Timeouts).

Zum Schluss wollen wir noch ein grösseres Beispiel einer HCFSM in Statecharts darstellen. Es handelt sich um die Steuerung der Betriebsarten einer Armbanduhr mit Alarm- und Stoppuhrfunktionen.

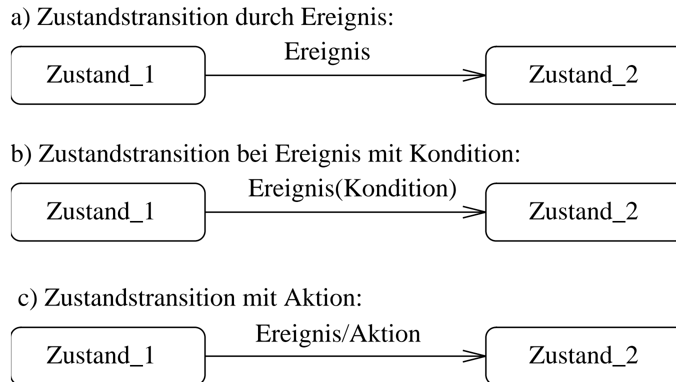


Abbildung 3.9: Arten von Zustandsübergängen in Statecharts.

Beispiel 3.7 *Abb. 3.10 zeigt ein Beispiel eines in Statecharts spezifizierten Systems zur Steuerung der Betriebsarten einer Armbanduhr (aus [12]). Die Uhr besitzt vier Knöpfe ul, ur, ll, lr (upper-left, upper-right, lower-left, lower-right) und fünf Betriebsarten, darunter:*

- *TIMER: In dieser Betriebsart (Grundzustand) wird nur die Uhrzeit dargestellt. In diesem Zustand bedeuten Knopfdruck*
  - *ll: Wechsel in den Zustand STOPWATCH;*
  - *ul: Wechsel in den Zustand TIME_UPDATE;*
  - *lr: Wechsel des Displayzustands (24H oder AM-PM); und*
  - *ur: Einschalten der Beleuchtung.*
- *TIME_UPDATE (Ändern der Zeit):*
  - *ll: Umschalten des zu verändernden Datums (Sekunden, Stunden, Minuten, Tag, etc.);*
  - *lr: Update des selektierten Datums;*
  - *ul: Zurück in den Zustand TIMER;*
- *STOPWATCH (Stoppuhrfunktion):*
  - *ll: Wechsel in Zustand ALARM;*
  - *lr: Start-Stop der Stoppuhr;*
  - *ur: Reset der Stoppuhr;*
- *ALARM (Weckfunktion):*
  - *ll: Wechsel in Zustand TIMER;*
  - *ul: Wechsel in Zustand ALARM_UPDATE;*
  - *lr: Ein- und Ausschalten des Stundensignals (chime);*
  - *ur: Ein- und Ausschalten des Alarms.*

Der Zustand ALARM_UPDATE besitzt die gleichen Funktionen wie der Zustand TIME_UPDATE. In jedem beliebigen Zustand soll der Knopfdruck ur den Alarm ein- bzw. ausschalten. Diese Spezifikation ist als Statechart in Abb. 3.10 dargestellt.

Ein Nachteil von nebenläufigen hierarchischen Zustandsmaschinen ist ohne Zweifel deren eingeschränkte zustandsorientierte Sicht.

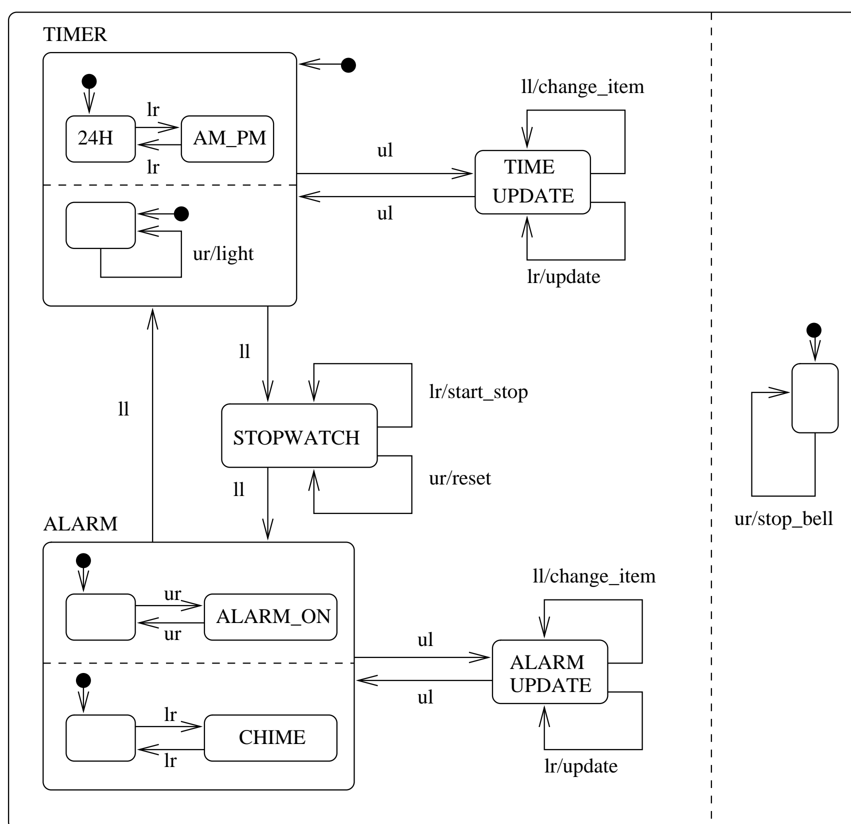


Abbildung 3.10: Statechart für eine Armbanduhr mit Alarm- und Stoppuhrfunktion.

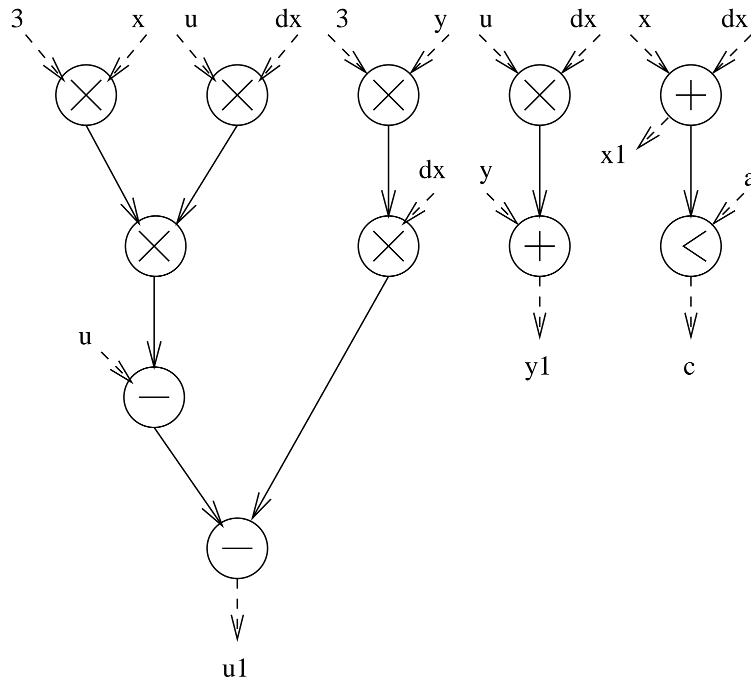


Abbildung 3.11: Datenflussgraph der Spezifikation zur Lösung von Differentialgleichungen nach der Euler-Methode aus Beispiel 2.1. Es gibt keine explizite Ausführungsreihenfolge der Aktivitäten. Aktivitäten sind Module, Blöcke, oder einfache Instruktionen.

3.5 Aktivitätsorientierte Modelle

3.5.1 Datenflussgraphen

Datenflussgraphen sind gerichtete Graphen, bestehend aus Knoten und Kanten. Die Knotenmenge stellt üblicherweise eine Menge von *Aktivitäten* dar. Die Kanten repräsentieren den gerichteten *Datenfluss*. In einem Datenflussgraphen werden die Berechnungen allein durch die Verfügbarkeit von Daten gesteuert.

Beispiel 3.8 *Abb. 3.11 zeigt den Datenflussgraphen der in Kapitel 2, Beispiel 2.1 eingeführten Spezifikation, die eine Lösungsmethode für Differentialgleichungen nach der Euler-Methode spezifiziert. Achtung: Kommutativität und Assoziativität in Ausdrücken führen im Allgemeinen zu unterschiedlichen Datenflussgraphen.*

Offensichtlich lässt sich ein Datenflussgraph direkt als Petri-Netz darstellen, wenn man die Knoten als Transitionen modelliert und die Kanten als mit den Transitionen verbundene Stellen auffasst. Eingangskanten von Knoten ohne Vorgänger werden durch Stellen ohne Vorbereich ersetzt, Ausgangskanten von Knoten ohne Nachfolger werden durch Stellen ohne Nachbereich ersetzt.

3.5.2 Markierte Graphen

Markierte Graphen [13] sind Datenflussgraphen mit speziellen Eigenschaften. Ein Beispiel eines markierten Graphen ist in Abb. 3.12 dargestellt. Ein markierter Graph lässt sich

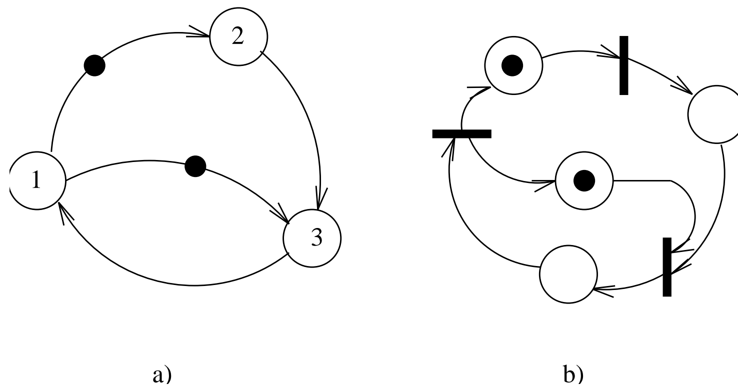


Abbildung 3.12: Modell eines markierten Graphen a) und äquivalente Darstellung in Petri-Netz-Notation b).

durch ein Petri-Netz beschreiben, in dem jede Stelle genau eine Eingangstransition und genau eine Ausgangstransition hat.

Definition 3.11 *Ein markierter Graph ist ein Petri-Netz $G = (P, T, F, K, W, M_0)$ mit den Eigenschaften:*

1. $\forall p \in P : |\bullet p| = |p \bullet| = 1$,
2. $K = \infty, W = 1$.

Sieht man von der Markierung ab, so sind markierte Graphen dual zu Zustandsdiagrammen. Dies wird z. B. an dem in Abb. 3.12 dargestellten Beispiel deutlich.

Die übliche Interpretation dieser Klasse von Netzen ist die Assoziation von Aktionen (z. B. Signale, Berechnungen, Tasks, etc.) mit Transitionen und die Assoziation von Stellen mit einem Datenpuffer mit der Semantik eines FIFOs (First-in-First-out-Speicher). Markierte Graphen sind also aktions- bzw. datenflussorientierte Modelle. Markierte Graphen besitzen zwar eine geringere Modellierungskraft als allgemeine Petri-Netze. So können Verzweigungen nicht dargestellt werden. Sie sind konfliktfrei und können damit nur deterministisches Verhalten beschreiben. Allerdings lassen sich Eigenschaften wie Lebendigkeit und Sicherheit algorithmisch effizient bestimmen. Commoner et al. zeigten in [13], dass für markierte Graphen folgende Eigenschaften gelten:

- Die Summe der Marken innerhalb eines gerichteten Zyklus C ändert sich nicht durch Feuern von Transitionen:

$$\forall M \in [M_0\rangle : \sum_{p \in C} M_0(p) = \sum_{p \in C} M(p)$$

- Eine Markierung M_0 ist lebendig genau dann, wenn die Summe der Marken eines jeden gerichteten Zyklus C positiv ist:

$$M_0 \text{ lebendig} \Longleftrightarrow \forall C \in Zyklen(G) : \sum_{p \in C} M_0(p) > 0$$

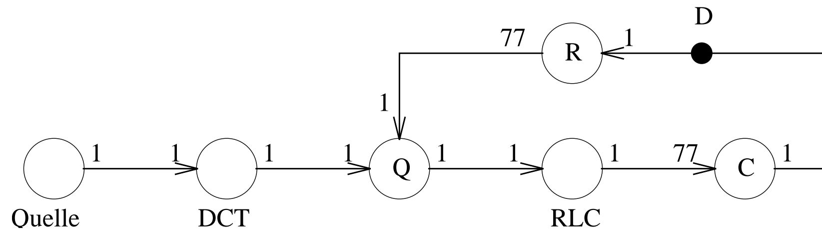


Abbildung 3.13: Darstellung eines Algorithmus zur Bilddatenkompression mittels eines SDF-Graphen.

- Eine lebendige Markierung M_0 ist genau dann sicher, wenn jede Transition in einem gerichteten Zyklus C mit Markensumme 1 ist:

$$M_0 \text{ lebendig} \wedge M_0 \text{ sicher} \Longleftrightarrow \forall t \in T : t \in C, C \in Zyklen(G) \wedge \sum_{p \in C} M(p) = 1$$

- Für jeden stark zusammenhängenden markierten Graphen [1] gibt es eine lebendige und sichere Anfangsmarkierung M_0.

Zur Analyse der Performanz von Systemen fügt man den Knoten oder den Kanten eines markierten Graphen oft noch ein Attribut der Rechenzeit zu. Solche Graphen heissen *zeitbehaftete markierte Graphen* (timed marked graphs).

3.5.3 Synchrone Datenflussgraphen (SDF)

Interessant für Anwendungen im Bereich der Signalverarbeitung sind Systeme, deren Teilsysteme mit mehreren unterschiedlichen Datenraten arbeiten und gewisse Teilaufgaben realisieren.

Beispiel 3.9 *Abb. 3.13 zeigt einen Teil eines Bildkodierers. Dargestellt ist ein Eingabeknoten, der die Bildquelle darstellt, gefolgt von einem DCT (Diskrete Kosinustransformation) und einer über eine Rückführungsschleife gesteuerten Quantisierungsstufe (Q) mit abschliessender Kodierung (RLC). Nach dem ETSI-Standard wird ein neuer Steuerwert des Quantisierers immer dann gebildet, wenn eine neue Bildzeile anliegt. Bei einer Zeilenlänge von 616 Pixeln pro Zeile und einer Blockbreite von 8 Pixeln muss ein neues Steuerwort alle 616/8 = 77 Blöcke gebildet werden. Der Knoten C (Control) bildet aus 77 Eingangsdaten ein neues Steuerdatum. Der Operator R (Replikator) liest ein Steuerwort ein, dass aus der Information von 77 Blöcken gebildet wird, und repliziert diesen Wert 77 mal zur Steuerung des Quantisierers für die nächste Bildzeile. Daraus ergeben sich unterschiedliche Raten der einzelnen Knoten. Z.B. arbeiten die Knoten DCT, Q und RLC 77-mal in dem Zeitrahmen, in dem Knoten C und Knoten R einmal arbeiten.*

Zur Modellierung von Anwendungen der digitalen Signalverarbeitung definiert Lee [14] das Modell des *synchronen Datenflussgraphen*, im Folgenden SDF-Modell genannt. Ein SDF-Graph ist ein um die Eigenschaft, dass die Gewichte von eins verschieden sein können, erweiterter markierter Graph und damit auch einer Teilklasse von Petri-Netzen äquivalent:

[1] Ein gerichteter Graph heisst stark zusammenhängend, wenn es von jedem Knoten zu jedem anderen Knoten einen gerichteten Pfad gibt.

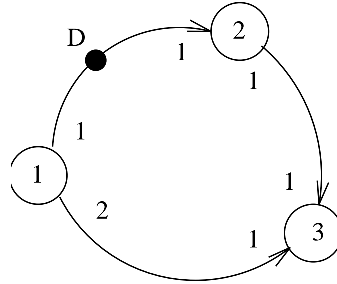


Abbildung 3.14: SDF-Graph; Markierungen sind entweder durch schwarze Punkte oder explizite Bezeichung (D) dargestellt.

Definition 3.12 *Ein SDF-Graph (Synchroner Datenflussgraph) ist ein Petri-Netz* $G = (P, T, F, K, W, M_0)$ *mit den Eigenschaften:*

1. $\forall p \in P : |\bullet p| = |p \bullet| = 1$,
2. $K = \infty$,
3. *Jeder Eingangsstelle* p *einer Transition* t *ist eine Zahl* $cons(p,t) := W(p,t) \in \mathbf{N}$ *zugewiesen. Jeder Ausgangsstelle* p *einer Transition ist eine Zahl* $prod(t,p) := W(t,p) \in \mathbf{N}$ *zugewiesen.*

Im SDF-Graphen sind die Gewichte *prod* als Zahlen am Ursprung einer Kante, die Gewichte *cons* als Zahlen am Ziel einer Kante dargestellt, siehe z. B. Abb. 3.13. Vorsicht: Die Bezeichnung „synchroner Datenfluss“ basiert auf der Motivation, eine Berechnung synchron zu nennen, wenn die Grössen *cons* und *prod* Konstanten sind, die vorgegeben sind. Das Modell hat nichts mit Synchronismus zu tun wie man ihn im Sinne einer Schaltkreisrealisierung versteht! Abb. 3.14 zeigt ein weiteres Beispiel eines SDF-Graphen. Algebraisch lässt sich ein SDF-Graph durch folgende Struktur beschreiben:

- *SDF-Graph* $G = (V, A, cons, prod, del)$. Hierin stellt V die Menge der Knoten (d. h. Berechnungen) dar, $A \subseteq V \times V$ die Menge der Kanten (FIFO-Puffer), $cons : A \rightarrow \mathbf{N}$ die Anzahl der beim Feuern konsumierten Marken (Datensamples), $prod : A \rightarrow \mathbf{N}$ die Anzahl der beim Feuern erzeugten Marken und $del : A \rightarrow \mathbf{N}$ die Anzahl der initialen Marken. Eine Markierung wird als Vektor $del \in \mathbf{Z}^{1 \times |A|}$ dargestellt.
- *Topologiematrix* $C \in \mathbf{Z}^{|V| \times |A|}$: Die Graphstruktur lässt sich mit der Topologiematrix darstellen, die ähnlich der Inzidenzmatrix eines gerichteten Graphen ist. Verläuft Kante a_j von Knoten v_i nach Knoten v_k, dann sind alle Einträge der Spalte c_j bis auf $c_{ij} = -prod(j)$ und $c_{kj} = cons(j)$ gleich null. Für eine Selbstschleife sind alle Einträge der entsprechenden Spalte gleich null.

Beispiel 3.10 *Für den SDF-Graphen in Abb. 3.14 erhalten wir folgende algebraische Modellierung:* $V = \{v_1, v_2, v_3\}$, $A = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$, $del = (1, 0, 0)$ *und die Topologiematrix*

$$C = \begin{pmatrix} -1 & 0 & -2 \\ 1 & -1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Verbindungen eines SDF-Graphen zur Aussenwelt werden nicht explizit darstellt. Die Dynamik der Markierung lässt sich durch folgende lineare Gleichung beschreiben:

$$del' = del - \gamma C$$

mit $\gamma \in \mathbf{N}^{1\times|V|}$. Dabei stellt *del* die Anfangsmarkenverteilung dar und *del'* die neue Markenverteilung, wenn jeder Knoten v_i genau γ_i-mal gefeuert hat.

Beispiel 3.11 *Betrachten wir erneut den SDF-Graphen in Abb. 3.14. Wenn wir die Markenverteilung wissen wollen, die entsteht, nachdem Knoten v_2 gefeuert hat, so erhalten wir mit $\gamma = (0, 1, 0)$ die neue Markenverteilung $del' = del - \gamma C = (1, 0, 0) - (1, -1, 0) = (0, 1, 0)$.*

Bei signalverarbeitenden Systemen hat man im Allgemeinen unendliche Eingangsströme. Folglich ist man an der Analyse von periodischen Abläufen (Feuern der Knoten) interessiert. Offensichtlich ist es wichtig, dass in dem spezifizierten System die Menge der Daten endlich bleibt (bounded memory). Da die Anzahl der Marken, die von den einzelnen Knoten produziert und konsumiert werden, unterschiedlich sind, ist die Summe der Daten nicht unbedingt konstant. Damit kommt es zu der Frage, wie ein SDF-Graph spezifiziert sein muss, damit Beschränktheit garantiert ist. Unter der Annahme, dass der gegebene SDF-Graph zusammenhängend ist, ist man z. B. daran interessiert, ob es einen periodischen Ablauf gibt, in dem die ursprüngliche Markierung wieder eingenommen wird und wie die Markierungen innerhalb eines solchen periodischen Ablaufs aussehen. Eine notwendige Bedingung dafür ist, dass die Topologiematrix die Bedingung $rang(C) = |V| - 1$ erfüllt. Der Beweis dafür ist einfach: Ist der Graph wie angenommen zusammenhängend, so kann der Rang von C nur entweder $|V|$ oder $|V| - 1$ sein. Um die initiale Markenverteilung wieder zu erreichen, muss gelten: $del' = del - \gamma C = del$ und damit $\gamma C = 0$. Diese Gleichung muss eine nichttriviale Lösung haben. Dies ist nur dann möglich, falls C nicht vollen Rang hat und damit muss gelten $rang(C) = |V| - 1$. Falls diese Bedingung nicht erfüllt ist, heisst der SDF-Graph *inkonsistent*.

Beispiel 3.12 *Der Graph in Abb. 3.15a ist inkonsistent, da $rang(C) = 3$. Man erkennt leicht, dass nachdem Knoten v_1, v_2 und v_3 jeweils einmal gefeuert haben, eine Marke auf der Kante (v_1, v_3) akkumuliert. Für den Graphen in Abb. 3.15b hingegen gilt $rang(C) = |V| - 1 = 2$. In einem periodischen Ablauf feuern Knoten v_1 und Knoten v_2 jeweils einmal, während Knoten v_3 zweimal feuert. Der Vektor $\gamma = (1, 1, 2)$ erfüllt $\gamma C = 0$ und führt damit auf die initiale Markenverteilung zurück.*

Zum Schluss sei noch erwähnt, wie man die relative Häufigkeit von Knotenfeuerungen, die zur gleichen Markenverteilung führen, bestimmen kann. Sei $rang(C) = |V| - 1$. Dann sucht man den kleinsten positiven Vektor γ, der im ganzzahligen Nullraum der Matrix C liegt.

Beispiel 3.13 *Für den SDF-Graphen in Abb. 3.15 ist der Nullraum von C gegeben durch $\alpha \times (1, 1, 2)$ mit $\alpha \in \mathbf{N}$. D.h. mit $\alpha = 1$ erhält man den kleinsten ganzzahligen Vektor, der zur initialen Markenverteilung führt.*

Im Zusammenhang mit SDF-Graphen ist weiterhin interessant, ob ein so spezifiziertes System in eine Verklemmung geraten kann, sowie die Abbildung von SDF-Graphen auf dedizierte Hardware, Software oder beides. Hierzu kommt auch die Frage hinzu, ob und wie man gültige Ablaufpläne für Softwarerealisierungen auf Mikroprozessoren und Parallelrechnern finden kann. Wir verweisen hier auf Übungen und Praktikum.

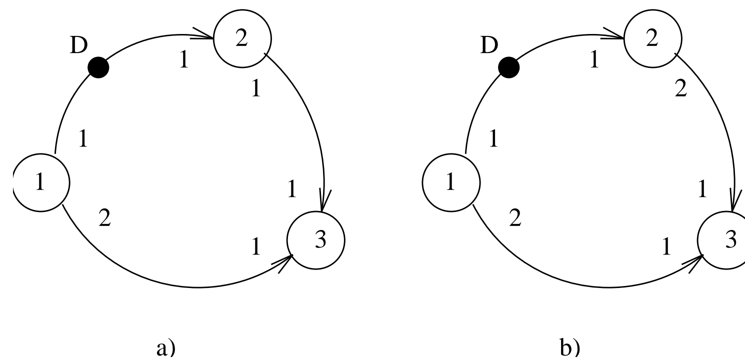


Abbildung 3.15: Konsistenz in SDF-Graphen: der SDF-Graph in a) ist nicht konsistent, der Graph in b) ist konsistent.

3.6 Strukturorientierte Modelle

Wir haben gezeigt, dass Petri-Netze sehr unterschiedliche Aspekte von Systemen modellieren können. Darunter fallen zustandsorientierte Modelle (FSM, HCFSM), aktivitätsorientierte Modelle (Markierte Graphen, Datenflussgraphen) und zeitorientierte Modelle (zeitbehaftete markierte Graphen). Obige Modelle dienen alle schwerpunktmässig der Modellierung des Verhaltens von Systemen. Im Folgenden wollen wir noch einige strukturorientierte Modelle kennenlernen. Im Gegensatz zu Verhaltensmodellen liegt bei strukturorientierten Modellen der Schwerpunkt in der Modellierung der *physikalischen Struktur* eines Systems.

3.6.1 Komponenten-Verbindungsdiagramm (CCD)

Als *Komponten-Verbindungsdiagramme* (Component-connectivity diagrams (CCDs)) bezeichnet man eine Klasse von strukturorientierten Modellen, die ein System als eine Menge von Komponenten und deren Verbindungen beschreiben. CCDs stellen also ein System aus struktureller Sichtweise dar. In einem CCD entsprechen Knoten Komponenten mit definierter Menge von Ein- und Ausgängen. Diese Komponenten sind beispielsweise Prozessoren, ALUs, Gatter, Transistoren bei der Modellierung von Hardware oder Programme, Module, Funktionen bei der Modellierung von Software. Den Kanten entsprechen Verbindungen der Komponenten. Auf der Ebene von Hardware sind dies beispielsweise Leitungen und Busse. Auf der Softwareebene können dies Variablen bis hin zu komplexen Datenobjekten sein.

CCDs spielen auf allen Abstraktionsebenen des Entwurfs eine wichtige Rolle. So ist z. B. in Abb. 3.16 ein System auf a) Systemebene, b) Architekturebene und c) Logikebene dargestellt mit Hilfe von CCDs.

Auf Systemebene sind die Komponenten beispielsweise Prozessoren, Speicher oder ein ASIC. Verbindungen sind auf dieser Ebene nur partiell spezifiziert. Auf Architekturebene sind die strukturellen Objekte ALUs, Register, Selektoren, Busse. Die Verbindungen zeigen, wo die Daten zwischen den arithmetischen Einheiten und den Speicherelementen fliessen. Auf Logikebene sind die strukturellen Objekte logische Gatter. Hier stellen die Verbindungen physikalische Leitungen dar.

Auf die Beschreibung von datenorientierten Modellen wollen wir im Rahmen dieser Vorlesung verzichten. Statt dessen wollen wir uns zum Abschluss dieses Kapitels einigen

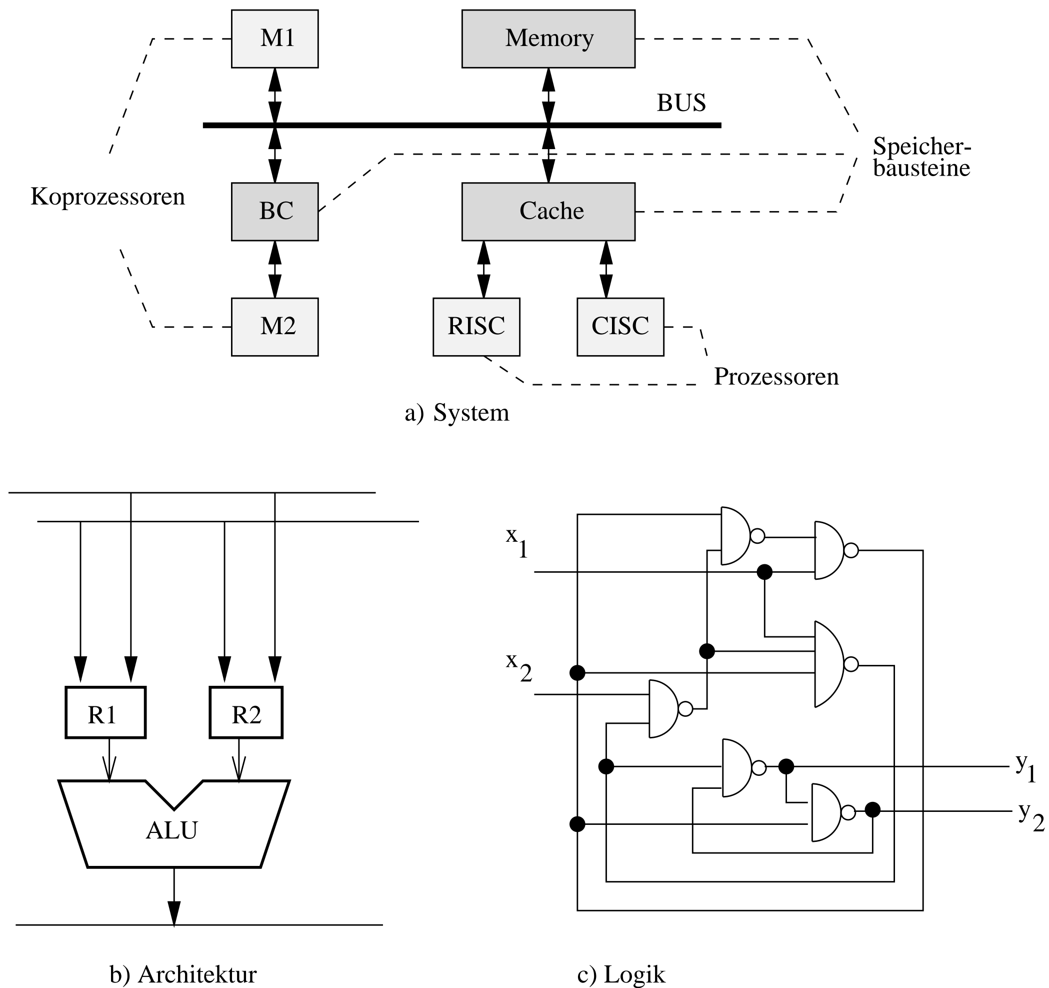


Abbildung 3.16: Strukturorientierte Modellierung: a) CCD auf Systemebene, b) auf Architekturebene und c) auf Logikebene.

heterogenen Modelle widmen. Basierend auf diesen Modellen wollen wir in den nächsten Kapiteln die essentiellen Entwurfsschritte beim Entwurf eingebetteter Systeme erläutern.

3.7 Heterogene Modelle

Die bisher vorgestellten Modelle haben den Vorteil, dass sie eine bestimmte Eigenschaft eines Systems bzw. Sichtweise auf ein System gut beschreiben. Während es zwar leichter ist, solche spezifischen Modelle zu analysieren bzw. aus ihnen funktionierende Systeme zu synthetisieren, leiden sie unter ihrer eingeschränkten Modellierungskraft. Um ein komplexes System zu beschreiben, benutzt man im Allgemeinen heterogene Modelle.

3.7.1 Kontroll/Datenflussgraphen (CDFGs)

Das Modell von Kontroll/Datenflussgraphen (control/data flow graph (CDFG)) ist ein heterogenes Modell, das eine Aufteilung einer Systemspezifikation in steuerungsorientierte Komponenten und datenflussorientierte Komponenten vornimmt und damit beide Aspekte in einem Modell vereinigen kann. Offensichtlich können im Datenflussgraphen keine *Kontrollstrukturen*, wie z. B. *Verzweigungen* und *Iterationen* (z. B. Schleifenkonstrukte) modelliert werden. Eine einfache Möglichkeit besteht darin, das Modell des Datenflussgraphen durch Einführung sogenannter *Verzweigungsknoten* zu erweitern. Ein Verzweigungsknoten stellt den Ursprung einer Menge alternativer Pfade dar, die den einzelnen Verzweigungsalternativen entsprechen. Verzweigungsknoten berechnen Verzweigungsentscheidungen. Ein Schleifenkonstrukt lässt sich dann einfach durch eine Verzweigung mit Test auf die Abbruchbedingung der Iteration modellieren. Dies werden wir im Folgenden zeigen.

Wir wollen uns im Weiteren mit einer erweiterten Klasse von hierarchischen CDFGs beschäftigen, sogenannten *Sequenzgraphen* [3] und führen die Eigenschaften von CDFGs anhand dieser Erweiterung direkt ein.

Beispiel 3.14 *Abb. 3.17 stellt den dem in Abb. 3.11 gezeigten Datenflussgraphen äquivalenten (nichthierarchischen) Sequenzgraphen dar.*

Definition 3.13 *Ein* Sequenzgraph *bezeichnet eine Hierarchie von gerichteten Graphen. Ein generisches Element des Graphen heisst* Einheit eines Sequenzgraphen. *Eine Einheit ist ein erweiterter Datenflussgraph $G_S(V, A)$ mit Knotenmenge V und Kantenmenge A sowie folgenden Eigenschaften:*

- *Eine Einheit besitzt zwei Arten von Knoten: a)* Operationen *oder Tasks und b)* Hierarchieknoten. *Hierarchieknoten dienen der Verbindung von Einheiten in der Hierarchie.*

- *Eine Einheit stellt einen* azyklischen *und* polaren *Graphen dar, d. h. es gibt zwei ausgezeichnete Knoten, den sog.* Startknoten *und den* Endknoten. *Beide Knoten sind Hierarchieknoten und stellen die Operation NOP (= keine Operation) dar. Neben Start- und Endknoten gibt es drei weitere Hierarchieknoten, nämlich Modulaufruf (CALL), Verzweigung (BR) und Iteration (LOOP).*

- *Sequenzgrapheinheiten, die die Blätter der Hierarchie darstellen, besitzen ausser dem Start- und Endknoten keine Hierarchieknoten.*

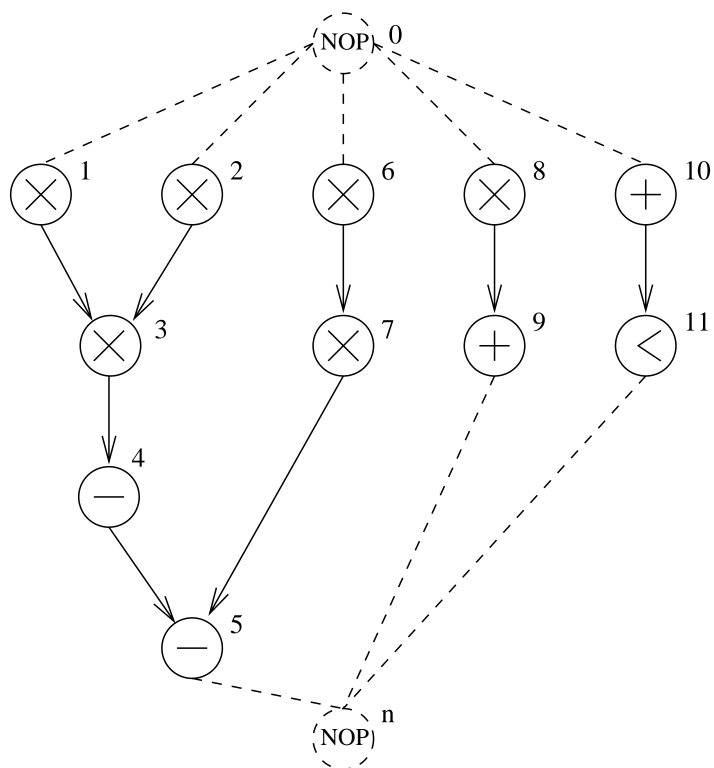


Abbildung 3.17: Sequenzgraph der Spezifikation zur Lösung von Differentialgleichungen nach der Euler-Methode aus Beispiel 2.1.

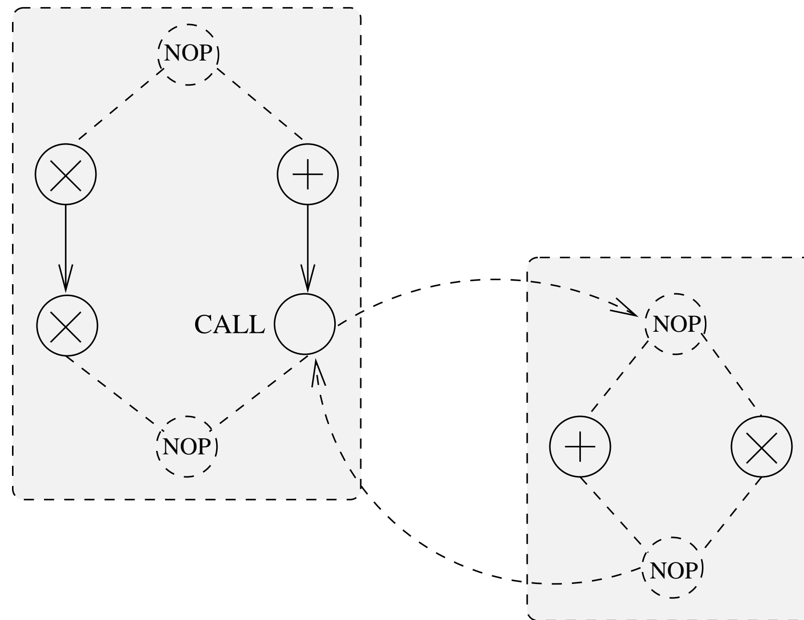


Abbildung 3.18: Modellierung von Modulaufrufen im Modell des Sequenzgraphen.

Beispiel 3.15 *In Abb. 3.17 sind Knoten* 0 *und Knoten* n *Hierarchieknoten. Knoten* 0 *ist der Startknoten und Knoten* n *der Endknoten. Aus einem gegebenen Datenflussgraphen erhält man den Sequenzgraphen durch 1) Entfernen aller Eingangskanten, die zu Knoten führen, die keine Vorgängerknoten besitzen, 2) Einfügen des Startknotens und Einfügen von je einer Kante vom Startknoten zu allen Knoten, die keine Eingangskanten besitzen, 3) Entfernen aller Ausgangskanten, die von Knoten wegführen, die keine Nachfolgerknoten besitzen, und 4) Verbinden aller Knoten, die keine Ausgangskanten besitzen, mit dem einzuführenden Endknoten.*

Auf die Betrachtung zyklischer Graphen kann man offensichtlich verzichten, wenn Iterationen über Hierarchiebildung darstellt werden können. Im Folgenden wird gezeigt, wie Modulaufrufe (siehe Abb. 3.18), Verzweigungen (siehe Abb. 3.19) und Iterationen (siehe Abb. 3.20) in Sequenzgraphen darstellt werden.

Ein Modulaufrufknoten ist ein Zeiger auf eine andere Einheit eines Sequenzgraphen auf niederer Hierarchiestufe. Er modelliert a) eine Menge von Abhängigkeiten der unmittelbaren Vorgängerknoten vom Startknoten des aufgerufenen Moduls und b) eine Menge von Abhängigkeiten des Endknotens des aufgerufenen Moduls von seinen unmittelbaren Nachfolgeknoten.

Verzweigung wird im Modell des Sequenzgraphen durch einen Verzweigungsknoten (BR) und eine Menge von Verzweigungsgraphen modelliert. Ein Verzweigungsgraph ist wiederum ein Sequenzgraph. Der Verzweigungsknoten berechnet einen *Verzweigungsausdruck* und wählt je nach dessen Wert einen Sequenzgraphen zur Ausführung aus. Die Anzahl unterschiedlicher Verzweigungsgraphen entspricht dem Wertebereich des Verzweigungsausdrucks. Die Ausführung eines Verzweigungsgraphen schliesst die Ausführung jedes anderen Verzweigungsgraphen aus.

Beispiel 3.16 *Abb. 3.18 zeigt die Modellierung von Modulaufrufen innerhalb von Sequenzgraphen. Der dargestellte Sequenzgraph entspricht z. B. den durch folgende Gleichung beschriebenen Abhängigkeiten:*

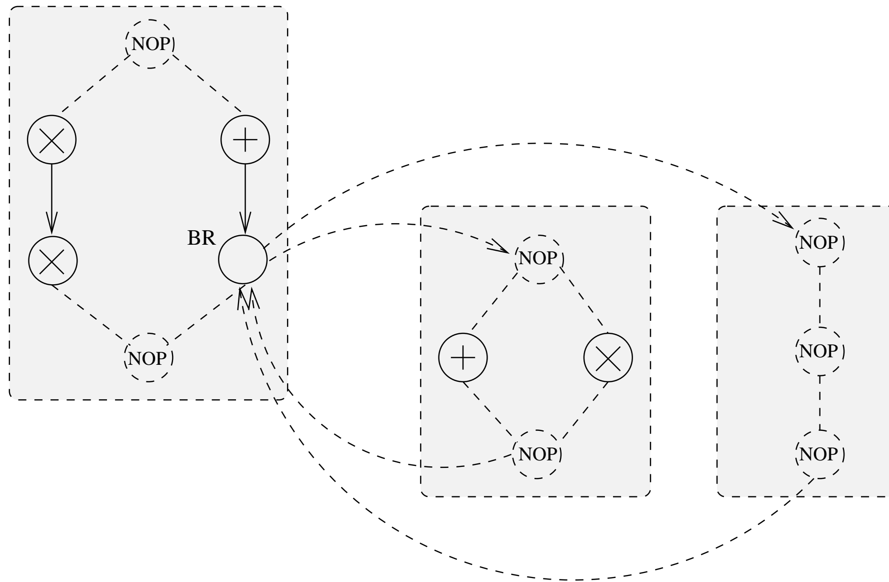


Abbildung 3.19: Modellierung von Verzweigung im Modell des Sequenzgraphen.

```
x := a * b; y := x * c; z := a + b; submodul(a,z);
```

mit

```
PROCEDURE submodul (m,n) IS
 p := m + n; q := m * n;
END submodul;
```

In 3.19 ist die Modellierung von Verzweigungen dargestellt. Der dargestellte Sequenzgraph entspricht z. B. dem Code:

```
x := a * b; y := x * c; z := a + b;
IF z > 0 THEN
  p := m + n; q := m * n;
END IF;
```

Ähnlich wie beim Modulaufruf werden Iterationen über Hierarchiebildung modelliert. Der Hierarchieknoten LOOP modelliert die Auswertung eines Ausdrucks zum Iterationsabbruch. Jede Iteration entspricht dem Aufruf des *Iterationsrumpfes*, der durch einen Sequenzgraphen repräsentiert wird.

Beispiel 3.17 *Abb. 3.20 zeigt eine Modellierung der Euler-Methode zur Lösung von Differentialgleichungen mittels Sequenzgraphen. Das System führt offensichtlich drei Aufgaben durch: a) Einlesen der Eingangsdaten, b) Iterieren, c) Ausgabe der Ausgangsdaten. Die Steuerung der Iteration erfolgt im Knoten LOOP, der die Variable c auswertet. Der Schleifenrumpf ist der in Abb. 3.17 dargestellte Graph.*

Letztlich gilt noch eine Bemerkung Attributen, die man den Knoten und Kanten eines Sequenzgraphen zuweisen kann. Dies können Messwerte bzw. Abschätzungen der entsprechenden Flächen und/oder Berechnungszeiten sein. Die Berechnungszeiten können *datenunabhängig* oder *datenabhängig* sein. Offensichtlich können nur datenunabhängige Berechnungszeiten vor der Synthese abgeschätzt werden. Datenabhängige Operationen sind beispielsweise Iterationen und Verzweigungen. Ferner können datenabhängige Berechnungen

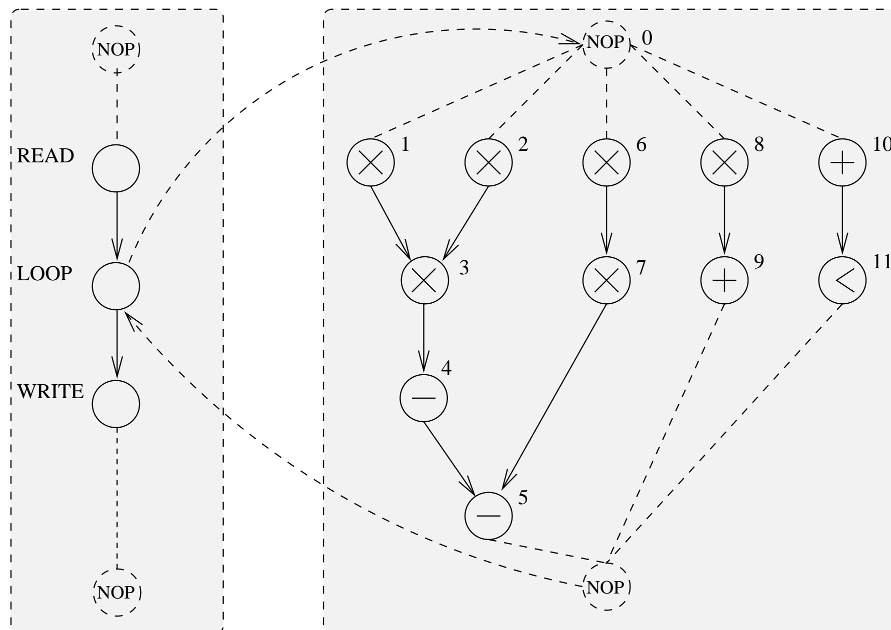


Abbildung 3.20: Modellierung von Iteration im Modell des Sequenzgraphen.

beschränkt (hier können untere und obere Schranken berechnet werden) oder *unbeschränkt* (z. B. das Warten auf externe Ereignisse) sein.

3.8 Können Programmiersprachen mehr?

Programmiersprachen sind heterogene Modelle, die gleichzeitig daten-, aktivitäts- sowie steuerungsorientiert sein können. Grundsätzlich unterscheidet man zwei Arten von Programmiersprachen: *imperative* und *deklarative* Sprachen. Imperative Programmiersprachen wie beispielsweise C und Pascal besitzen ein Ausführungsmodell, in dem Anweisungen in der Reihenfolge ausgeführt werden, wie sie im Programmtext erscheinen (= control-driven). LISP und PROLOG hingegen sind deklarative Sprachen. Für diese Sprachen ist charakteristisch, dass sie keine explizite Ausführungsreihenfolge spezifizieren (= demand-driven). Statt dessen wird das Ziel der Berechnung durch eine Menge von Funktionen oder von logischen Regeln ausgedrückt.

Imperative Programmiersprachen wie C bieten den Vorteil, dass komplexe Datenstrukturen wie Verbundtypen (Arrays, Records, etc.) leicht modelliert werden können. Prozeduren und Funktionen erlauben die Bildung von Hierarchie. Im Weiteren besitzen diese Sprachen zahlreiche Kontrollstrukturen wie Sequenzen von Anweisungen, Verzweigungen (z. B. IF, CASE), Schleifenkonstrukte (WHILE, FOR, REPEAT) und Unterprogrammaufrufe. Weiterhin haben imperative Programmiersprachen den Vorteil, dass sie weit verbreitet und *ausführbar* sind durch Compilierung auf einem Mikrocomputer. Die meisten programmierbaren Rechnerarchitekturen, darunter die meisten Mikrocomputer, werden in imperativen Programmiersprachen programmiert. Imperative Programmiersprachen haben jedoch den Nachteil, dass sie nebenläufige Operationen nicht beschreiben können. Dieses Problem führte zur Einführung zahlreicher neuer Programmiersprachen, darunter das Modell CSP (communicating sequential processes) [10], ADA, ParallelC und VHDL [15]. VHDL hat sich

dabei als Hardwarebeschreibungssprache durchgesetzt. Zahlreiche verschiedene Kommunikationsmechanismen wie das sog. *message passing* in CSP, der *Rendezvous*-Mechanismus in ADA und die Kommunikation über globalen Speicher können in VHDL modelliert werden. Während oben genannte Sprachen sehr allgemein sind, soll hier erwähnt werden, dass zur Spezifikation von reaktiven Echtzeitsystemen weitere Sprachen entwickelt wurden, darunter ESTEREL, LUSTRE, LUCID und Argos. Diese Sprachen ermöglichen nicht nur die Synthese von Hardware ausgehend von einer Verhaltensspezifikation sondern bieten mittlerweile auch elegante formale Verifikationsmethoden zur Überprüfung der Korrektheit von Spezifikationen. Man bezeichnet obige Sprachen als *synchron*, basierend auf der Vorstellung, dass die Antwort eines Systems auf externe Ereignisse ohne zeitliche Verzögerung erfolgt. Diese Annahme, wenn auch oft unrealistisch, ist begründet durch die Forderung, dass ein reaktives System die Reihenfolge aller externen Ereignisse verfolgen können muss. In der Realität hat man jedoch häufig genauere Vorstellungen, in welchen Zeitabständen externe Ereignisse kommen können. Solche gegebenen Daten (z. B. welche Datenrate ein System verarbeiten können muss oder wieviel Zeit verstreichen darf, bis ein System auf ein externes Ereignis reagiert) erlauben eine bessere Optimierung der zu synthetisierenden Systemkomponenten. Es bietet sich also besser ein zeitbehaftetes Modell an wie beispielsweise zeitbehaftete markierte Graphen oder Sequenzgraphen.

Kapitel 4

Synthese

Im ersten Teil der Vorlesung haben wir uns vor allem mit der Modellierung von Systemen und deren Spezifikation beschäftigt. Nach erfolgter Spezifikation ist es Aufgabe der *Synthese*, eine funktionale Spezifikation auf eine strukturelle Beschreibung abzubilden. Die wichtigsten Aufgaben der Synthese wurden in Kapitel 2 kurz als *Allokation* der Ressourcen, *Ablaufplanung*, und *Bindung* der Operationen an Ressourcen vorgestellt. Wir wollen zeigen, dass diese Grundaufgaben unabhängig von der Implementierung als Hardware- und/oder Softwaresystem verstanden werden können. Im Weiteren sind diese Grundaufgaben auch unabhängig vom Grad der bereits erzielten Abstraktion, also System-, Architektur- und Logiksynthese bei einer Hardwareimplementierung bzw. Programm- und Instruktionsebene bei einer Softwareimplementierung.

Als Grundmodell einer Ausgangsspezifikation werden wir das in Kapitel 3 eingeführte Modell des Sequenzgraphen zugrundelegen. An diesem Modell werden wir die Komplexität der Grundprobleme der Synthese untersuchen. Die diskrete Natur dieser Probleme bedarf der Lösung kombinatorischer Entscheidungs- und Optimierungsprobleme. Da Effizienz und Exaktheit von Algorithmen oft gegensätzliche Forderungen sind, werden wir einerseits *Heuristiken* und andererseits *exakte Verfahren* kennenlernen.

4.1 Fundamentale Syntheseprobleme

Wir betrachten nun die Hauptprobleme der Synthese von Systemen, nämlich das Festlegen der zeitlichen Abläufe von Operationen (Ablaufplanung, *Scheduling*) und deren Bindung an Ressourcen (*Binding*). Als Grundmodell betrachten wir eine durch einen nichthierarchischen Sequenzgraphen gegebene Spezifikation. Die Erweiterungen auf hierarchische Graphen sowie die Betrachtung allgemeinerer Graphen (u. a. zyklische Graphen) werden später behandelt. Im Weiteren nehmen wir zur Einfachheit an, dass die Ausführungszeiten aller Operationen, die Typen aller zur Verfügung stehenden Ressourcen sowie deren Kosten bekannt sind.

Unsere weiteren Annahmen beruhen also auf folgenden Gegebenheiten:

- Die Operationen und deren Abhängigkeiten werden in einem nichthierarchischen Sequenzgraphen $G_S = (V_S, E_S)$ mit Knotenmenge $V_S = \{v_0, v_1, \cdots, v_n\}$ beschrieben. v_0 heisse Quellknoten, v_n heisse Endknoten. G_S habe $n_{ops} = n - 1$ Operationen.

- Der Zusammenhang zwischen Operationen und Ressourcentypen wird in einem Ressourcengraphen $G_R = (V_R, E_R)$ beschrieben, der wie folgt definiert ist.

Definition 4.1 (Ressourcengraph) *Ein bipartiter Ressourcengraph $G_R = (V_R, E_R)$ besteht aus den Knoten $V_R = V_S \cup V_T$, wobei V_S die Knoten des Sequenzgraphen G_S sind. Die Knoten $v_t \in V_T$ repräsentieren einen bestimmten Ressourcentyp. Die Kanten $(v_s, v_t) \in E_R$ mit $v_s \in V_S$ und $v_t \in V_T$ modellieren die Verfügbarkeit einer Instanz einer Ressource des Typs v_t für eine Operation v_s.*

- Die *Kostenfunktion* $c : V_T \to \mathbf{Z}$ ordnet jedem Ressourcentyp $v_t \in V_T$ die Kosten $c(v_t)$ zu.
- Die Funktion $w : E_R \to \mathbf{Z}^+$ ordnet jeder Kante (v_s, v_t) des Ressourcengraphen die Ausführungszeit zu, die Operation v_s auf Ressourcentyp v_t benötigt.

Bei unseren ersten Untersuchungen werden wir uns auf den einfachen Fall beschränken, dass einer Operation v_s genau ein Ressourcentyp zugeordnet ist. Daraus folgt, dass im Ressourcengraph an jedem Knoten in V_S genau eine Kante beginnt und jeder Operation $v_s \in V_S$ genau eine Ausführungszeit w_s zugeordnet ist mit $w_s = w((v_s, v_t))$, $(v_s, v_t) \in E_R$.

Sequenzgraph und Ressourcengraph zusammen stellen unser mathematisches Modell zur Beschreibung der Syntheseprobleme dar. Anhand dieses Modells werden wir in folgenden Abschnitten wichtige Synthesalgorithmen kennenlernen. Zunächst definieren wir die Probleme der Ablaufplanung, Allokation und Bindung.

4.1.1 Ablaufplanung

Unter *Ablaufplanung* versteht man das Festlegen von Startzeiten der Operationen unter Berücksichtigung aller Datenabhängigkeiten im Sequenzgraphen. Dabei sei die *Startzeit* $\tau(v_s)$ einer Operation $v_s \in V_S$ der Zeitpunkt, zu dem die Ausführung der Operation beginnt. Damit erhalten wir folgende Definition:

Definition 4.2 (Ablaufplan) *Ein* Ablaufplan *(Schedule) eines Sequenzgraphen $G_S = (V_S, E_S)$ ist eine Funktion $\tau : V_S \to \mathbf{Z}^+$, die die Bedingungen*

$$\tau(v_j) - \tau(v_i) \geq w_i \quad \forall (v_i, v_j) \in E_S \tag{4.1}$$

erfüllt.

Definition 4.3 (Latenz) *Unter der* Latenz L *eines Sequenzgraphen verstehen wir die Zeitdifferenz der Startzeitpunkte von Quell- und Zielknoten, also $L = \tau(v_n) - \tau(v_0)$.*

Beispiel 4.1 *Als Beispiel betrachten wir den Sequenzgraphen des bereits in Kapitel 2 in Bsp. 2.1 eingeführten und in Abb. 3.11 dargestellten Sequenzgraphen eines Integrators zur Lösung einer Differentialgleichung nach der Euler-Methode. Unter der Annahme, dass die Ausführungszeit aller Operationen gleich eins sei, erhalten wir den in Abb. 4.1 dargestellten Ablaufplan $\tau(v_1) = \tau(v_2) = \tau(v_6) = \tau(v_8) = \tau(v_{10}) = 1$, $\tau(v_3) = \tau(v_7) = \tau(v_9) = \tau(v_{11}) = 2$, $\tau(v_4) = 3$ und $\tau(v_5) = 4$. Die Latenz L dieses Ablaufplans beträgt $L = \tau(v_n) - \tau(v_0) = 5 - 1 = 4$.*

4.1.2 Allokation

Definition 4.2 berücksichtigt bei der Formulierung von Ablaufplänen nicht die Tatsache, dass oft nur eine begrenzte Anzahl von Ressourcen eines jeden Typs zur Verfügung stehen.

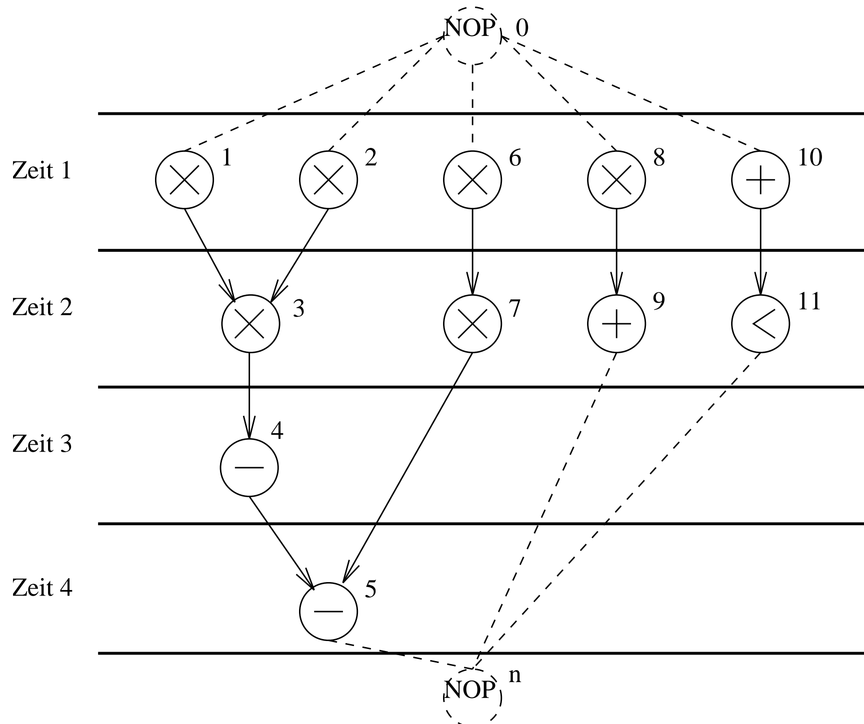


Abbildung 4.1: Ablaufplan eines Sequenzgraphen.

Beispiel 4.2 *Betrachten wir erneut den Ablaufplan in Beispiel 4.1, so bräuchte man im Zeitschritt* 1 *mindestens* 4 *Ressourcen, die eine Multiplikation durchführen können und eine Ressource, die eine Addition durchführen kann. Oft stehen nur begrenzt viele Ressourcen gleichzeitig zur Verfügung. Abb. 4.2 zeigt einen Ablaufplan des gleichen Sequenzgraphen, der die Nebenbedingungen erfüllt, dass es nur jeweils eine Ressource gibt, die eine Multiplikation bzw. Addition, Subtraktion und Vergleichsbildung durchführen kann. Die Latenz beträgt in diesem Falle* $L = \tau(v_n) - \tau(v_0) = 8 - 1 = 7$.

Die Festlegung von Mengenangaben aller zur Verfügung stehenden Ressourcentypen bezeichnen wir im Folgenden als *Allokation* von Ressourcen:

Definition 4.4 (Allokation) *Eine* Allokation *ist eine Funktion* $\alpha : V_T \rightarrow \mathbf{Z}^+$, *die jedem Ressourcentypen* $v_t \in V_T$ *die Anzahl* $\alpha(v_t)$ *verfügbarer Instanzen zuordnet.*

Beispiel 4.3 *Wir betrachten erneut den Ablaufplan in Abb. 4.1. Der Ressourcengraph zur Modellierung der Ressourcebeschränkungen in Beispiel 4.2 ist in Abb. 4.3 dargestellt. Sei* $V_T = \{r_1, r_2\}$. *Es gibt also zwei Ressourcentypen* (r_1: *Multiplizierer,* r_2: *ALU (Addierer, Subtrahierer und Vergleichsbildung)). Im Weiteren gelte* $\alpha(r_1) = 1, \alpha(r_2) = 1$, *d. h. es sind genau eine Instanz des Ressourcentyps Multiplizierer und eine Instanz des Typs ALU verfügbar. Die Kanten entsprechen den Ressourcentypzugehörigkeiten, die Kantengewichte stellen hier die Rechenzeiten* $w_s = w((v_s, v_t))$ *der Operationen* v_s *auf entsprechenden Ressourcentypen dar.*

4.1.3 Bindung

Bindung spezifiziert, welche Instanz eines Ressourcentypen welche Operationen implementieren wird.

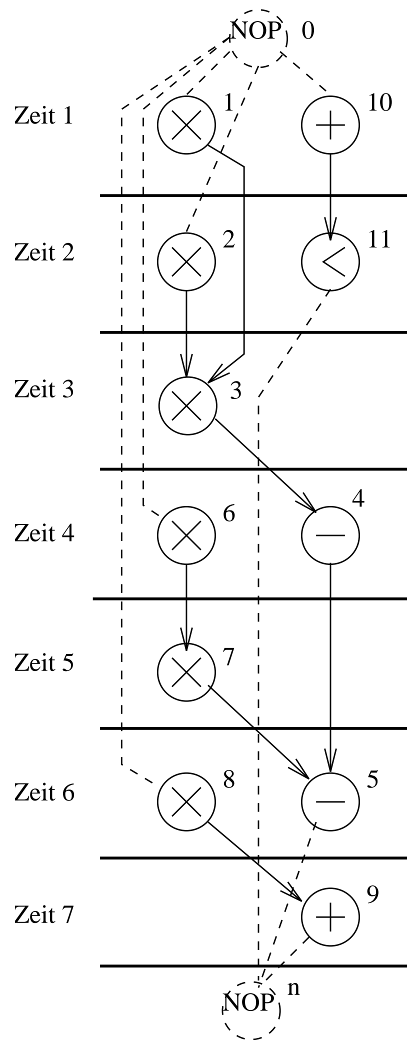


Abbildung 4.2: Ablaufplan eines Sequenzgraphen bei beschränkter Anzahl von Ressourcen.

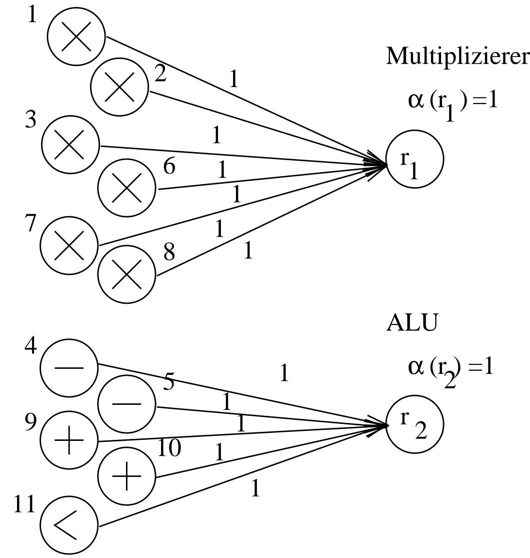


Abbildung 4.3: Ressourcengraph für den Sequenzgraphen und Ressourcebeschränkungen aus Beispiel 4.2.

Definition 4.5 (Bindung) *Die* Bindung *eines Sequenzgraphen* $G_S = (V_S, E_S)$ *ist gegeben durch Funktionen* $\beta : V_S \to V_T$ *und* $\gamma : V_S \to \mathbf{Z}^+$*, wobei* $\beta(v_s) = v_t$ *und* $\gamma(v_s) = r$ *bedeutet, dass Operation* $v_s \in V_S$ *durch die* r*-te Instanz des Ressourcentyps* $v_t \in V_T$ *implementiert wird.*

Offenbar müssen Start- und Endknoten eines Sequenzgraphen nicht an Ressourcen gebunden werden. Nun gibt es Ressourcentypen, die unterschiedliche Operationen implementieren können. Z.B. kann ein Prozessor sämtliche in seinem Instruktionssatz definierten Operationen ausführen. Als Beispiel auf Hardwareebene hatten wir bereits das Beispiel einer arithmetisch-logischen Einheit (ALU) vorgestellt, die die Operationen Addition, Subtraktion und Vergleich implementieren kann. Nun muss die Eigenschaft erfüllt sein, dass jede Operation eines gegebenen Sequenzgraphen auf mindestens einem Ressourcentyp implementierbar ist. Im Ressourcengraphen entspricht dies der Eigenschaft, dass von jedem Knoten $v_i \in V_S$ mindestens eine Kante ausgeht. Der Fall, dass eine Operation $v_i \in V_S$ auf verschiedenen Ressourcentypen implementiert werden kann (z. B. eine Addition auf einem Carry-Lookahead-Addierer oder einem Carry-Ripple-Addierer), entspricht dem Fall, dass im Ressourcengraphen mehrere Kanten im Knoten v_i beginnen. Dann spricht man von *Modulselektion.*

Ein einfacher Fall der Bindung entspricht dem Fall *dedizierter Ressourcen.* Jede Ressource bekommt maximal eine Operation zugewiesen.

Beispiel 4.4 *Betrachten wir erneut den Sequenzgraphen in Abb. 4.1. Im Falle dedizierter Ressourcen benötigt man genau* $n_{ops} = 11$ *Ressourcen. Mit der durch den Ressourcengraphen in Abb. 4.3 beschriebenen Ressourcenzugehörigkeit werden sechs Instanzen des Typs Multiplizierer und fünf Instanzen des Typs ALU benötigt. Eine Bindung ist gegeben durch* $\beta(v_1) = r_1, \gamma(v_1) = 1$, $\beta(v_2) = r_1, \gamma(v_2) = 2$, $\beta(v_3) = r_1, \gamma(v_3) = 3$, $\beta(v_4) = r_2, \gamma(v_4) = 1$, $\beta(v_5) = r_2, \gamma(v_5) = 2$, $\beta(v_6) = r_1, \gamma(v_6) = 4$, $\beta(v_7) = r_1, \gamma(v_7) = 5$, $\beta(v_8) = r_1, \gamma(v_8) = 6$, $\beta(v_9) = r_2, \gamma(v_9) = 3$, $\beta(v_{10}) = r_2, \gamma(v_{10}) = 4$, $\beta(v_{11}) = r_2, \gamma(v_{11}) = 5$.

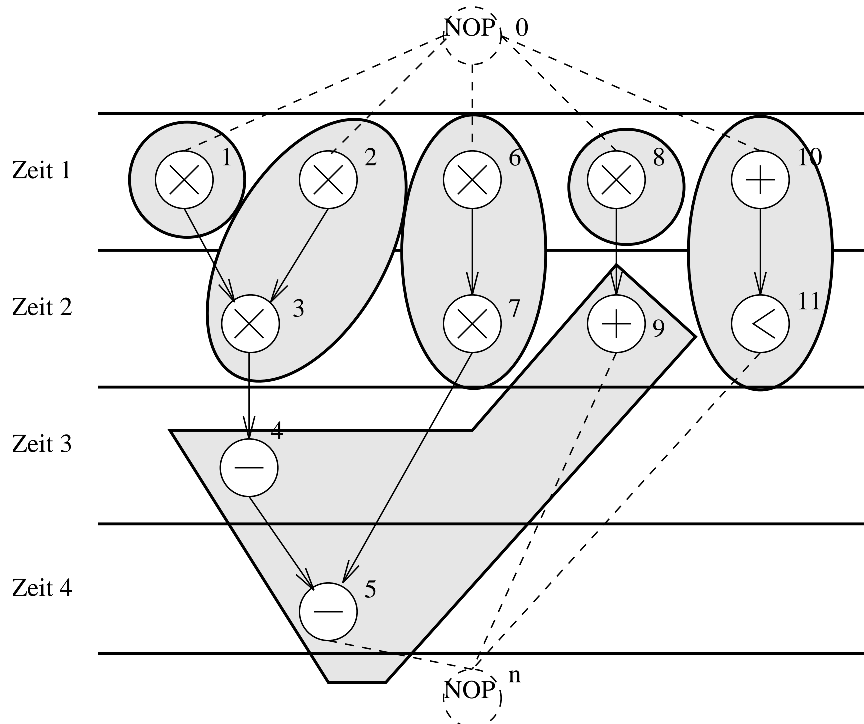


Abbildung 4.4: Ablaufplanung und Ressourcebindung eines Sequenzgraphen.

Beispiel 4.5 *Die Bindung in Beispiel 4.4 ist offensichtlich nicht besonders effizient. Die minimal benötigten Ressourcen zur Implementierung des in Abb. 4.1 gezeigten Ablaufplans erhält man durch Bestimmung der maximalen Anzahl gleichzeitig ablaufender Operationen eines Ressourcentyps über alle Zeitschritte. So werden nur vier Multiplizierer und zwei ALUs gebraucht. Die Ressourcebindung gegeben durch* $\beta(v_1) = r_1, \gamma(v_1) = 1$, $\beta(v_2) = r_1, \gamma(v_2) = 2$, $\beta(v_3) = r_1, \gamma(v_3) = 2$, $\beta(v_4) = r_2, \gamma(v_4) = 2$, $\beta(v_5) = r_2, \gamma(v_5) = 2$, $\beta(v_6) = r_1, \gamma(v_6) = 3$, $\beta(v_7) = r_1, \gamma(v_7) = 3$, $\beta(v_8) = r_1, \gamma(v_8) = 4$, $\beta(v_9) = r_2, \gamma(v_9) = 2$, $\beta(v_{10}) = r_2, \gamma(v_{10}) = 1$, $\beta(v_{11}) = r_2, \gamma(v_{11}) = 1$ *ist in Abb. 4.4 graphisch dargestellt.*

Damit haben wir die Grundaufgaben der Synthese vorgestellt. Offensichtlich sind die Probleme Ablaufplanung und Bindung eng miteinander verwoben. Zunächst sollte klar sein, dass es keine zwingende Reihenfolge geben muss, die Ablaufplanung vor der Bindung durchzuführen. Zum Beispiel kann eine *partielle Bindung* Teil einer anfänglichen Spezifikation sein. Eine gegebene Bindung (voll oder partiell) beeinflusst die Ablaufplanung, da nie zwei Operationen gleichzeitig auf einer Ressource implementiert werden können. Genauso haben wir gesehen, dass eine gegebene Ablaufplanung die nachfolgende Bindung beeinflusst.

Im Folgenden wollen wir Algorithmen untersuchen, die Ablaufplanung und Bindung durchführen sollen. Offensichtlich ist man nicht an beliebigen Lösungen interessiert, sondern an Lösungen, die eine Implementierung unter einem oder unter mehreren Gesichtspunkten optimieren. Beispielsweise versucht man bei der Ablaufplanung die Latenz eines Sequenzgraphen zu minimieren. Eine optimale Bindung minimiert die Kosten einer Implementierung, gegeben beispielsweise durch die Anzahl der benötigten Ressourcen. Während Latenzoptimalität und Kostenminimalität im Allgemeinen gegensätzliche Faktoren darstellen, ist man oft an Implementierungen interessiert, die latenzoptimal bei

gegebenen Ressourcenbeschränkungen beziehungsweise kostenminimal bei gegebenen Zeitbeschränkungen sind.

Um die Lösung solcher Optimierungsprobleme auch für komplexe Aufgabenstellungen rechnergestützt durchführen zu können, bedarf es effizienter Algorithmen. Bevor wir die wichtigsten Algorithmen zur Systemsynthese vorstellen, möchten wir informell definieren, was wir unter *Effizienz* bzw. *Optimalität* eines Algorithmus verstehen.

4.2 Algorithmen zur Ablaufplanung

Im Folgenden wollen wir einige Optimierungsprobleme im Zusammenhang mit der Ablaufplanung vorstellen, die Komplexität dieser Probleme analysieren und dann Algorithmen zu deren Lösung vorstellen. Um eine bessere Übersicht über die Verfahren zu erzielen, wird eine Einteilung der Optimierungsprobleme nach deren Komplexitätsklasse vorgenommen.

- *Unbeschränkte Ressourcen:* Es gibt keine Beschränkungen bezüglich Anzahl und Typ der zur Verfügung stehenden Ressourcen.
- *Beschränkte Ressourcen:* Der Lösungsraum ist durch Vorgaben über Anzahl und Typ der zur Verfügung stehenden Ressourcen eingeschränkt.

Eine weitere in der Literatur [16] bekannte Dichotomie unterteilt die Verfahren nach dem benutzen Lösungsansatz:

- *Transformatorische Algorithmen:* Ausgangspunkt ist ein Ablaufplan, der mit einem Verfahren für unbeschränkte Ressourcen berechnet worden ist. Dieser wird dann so lange transformiert, bis alle Beschränkungen der Ressourcen erfüllt sind und das gewünschte Optimalitätskriterium erfüllt ist.
- *Iterativ-konstruktive Algorithmen:* Diese Algorithmen betrachten in jedem Schritt die Ablaufplanung und Bindung genau einer Operation und bauen somit schrittweise konstruktiv den Ablaufplan auf.

4.2.1 Ablaufplanung ohne Ressourcebeschränkungen

Das Problem der Ablaufplanung ohne Ressourcebeschränkungen ist relevant in folgenden Fällen:

- *Bestimmung unterer Schranken*: Offensichtlich kann man eine untere Schranke für die Latenz aller Ablaufpläne mit Ressourcebeschränkungen finden, wenn man das Latenzminimierungsproblem ohne Ressourcenbeschränkungen gelöst hat.
- *Bindung mit dedizierten Ressourcen*: Praktisch gesehen liegt ein solcher Fall vor, falls fast alle Operationen unterschiedlicher Art sind, so dass jede Operation auf einem anderen Ressourcentyp implementiert werden muss oder wenn die Kosten der dedizierten Komponenten vernachlässigbar sind.
- *Lösen der Ablaufplanung nach der Bindung*: Den Einfluss der Bindung kann man durch zusätzliche Sequentialisierungskanten im Sequenzgraphen modellieren. Damit wird verhindert, dass Operationen, die an eine Ressource gebunden sind, nebenläufig ablaufen können. Die Kosten der Implementierung sind in diesem Fall unabhängig von der Ablaufplanung.

Damit kann man folgendes Optimierungsproblem formulieren:

Definition 4.6 (Latenzminierung ohne Ressourcenbeschränkung) *Gegeben seien ein nichthierarchischer Sequenzgraph $G_S = (V_S, E_S)$ und ein Ressourcengraph $G_R = (V_R, E_R)$ mit der Gewichtsfunktion $w : E_R \rightarrow \mathbf{Z}^+$, die jeder Kante (v_s, v_t) in G_R die Ausführungszeit $w_s = w((v_s, v_t))$ zuordnet. Unter* Latenzminimierung ohne Ressourcenbeschränkung *versteht man das Problem*

$$\min\{\ \tau(v_n) \mid \tau(v_j) - \tau(v_i) \geq w_i \ \ \forall (v_i, v_j) \in E_S\}$$

Der ASAP-Algorithmus

Die einfachste Möglichkeit, einen latenzoptimalen Ablaufplan zu bestimmen, ist, jede Operation so früh wie möglich ([17], *as soon as possible (ASAP)*) auszuführen. Das Problem der Latenzminimierung ohne Ressourcebeschränkungen kann in polynomieller Zeit exakt gelöst werden durch topologische Sortierung der Knoten des Sequenzgraphen. Bezeichnen wir die durch den Algorithmus bestimmten Startzeitpunkte der Operationen mit $\tau^S = \{\tau^S(v_0), \tau^S(v_1), \ldots, \tau^S(v_n)\}$, dann sieht der Algorithmus wie folgt aus:

ASAP($G_S(V_S, E_S), w$) {
    $\tau^S(v_0) := 1$;
    REPEAT {
        Wähle einen ungeplanten Knoten v_i aus, dessen Vorgänger alle geplant sind;
        $\tau^S(v_i) := \max_j\{\tau^S(v_j) + w_j \mid (v_j, v_i) \in E_S\}$
    }
    UNTIL (v_n geplant);
    RETURN (τ^S)
}

Beispiel 4.6 *Die in Abb. 4.1 dargestellte Ablaufplanung entspricht der durch den Algorithmus ASAP bestimmtem Ablaufplanung für die Annahme, dass $w_i = 1 \ \forall i \in \{1, \ldots, 11\}$. Im ersten Schritt wird $\tau(v_0)^S = 1$ gesetzt. Die Knoten ohne ungeplante Vorgänger sind v_1, v_2, v_6, v_8 und v_{10}. Deren Startzeit wird auf $\tau(v_0)^S + w_0 = 1$ gesetzt. So erhält man $\tau^S(v_i) = 1 \ \forall i \in \{1, 2, 6, 8, 10\}$, $\tau^S(v_i) = 2 \ \forall i \in \{3, 7, 9, 11\}$, $\tau^S(v_4) = 3$ und $\tau^S(v_5) = 4$ und damit die minimale Latenz $L^S = \tau^S(v_n) - \tau^S(v_0) = L = 5 - 1 = 4$.*

Der ALAP-Algorithmus

Wir betrachten nun den Fall, dass ein Ablaufplan eine sogenannte *Deadline* $\bar{L}$ erfüllen muss. Eine Deadline stellt eine obere Schranke für die Latenz L dar. Eine interessante Frage nach Vorgabe einer Deadline $\bar{L}$ ist, in welchen Intervallen die Startzeiten der Operationen schwanken können, um die Deadline zu erfüllen. Zum ASAP-Algorithmus, der jeweils die frühesten Startzeitpunkte aller Operationen ermittelt, gibt es einen komplementären Algorithmus, den sogenannten *ALAP-Algorithmus* (*as late as possible*), der die spätesten Startzeitpunkte ermittelt. Der Algorithmus sieht wie folgt aus:

ALAP($G_S(V_S, E_S), w, \bar{L}$) {
    $\tau^L(v_n) := \bar{L} + 1$;
    REPEAT {
        Wähle einen ungeplanten Knoten v_i aus, dessen Nachfolger alle geplant sind;

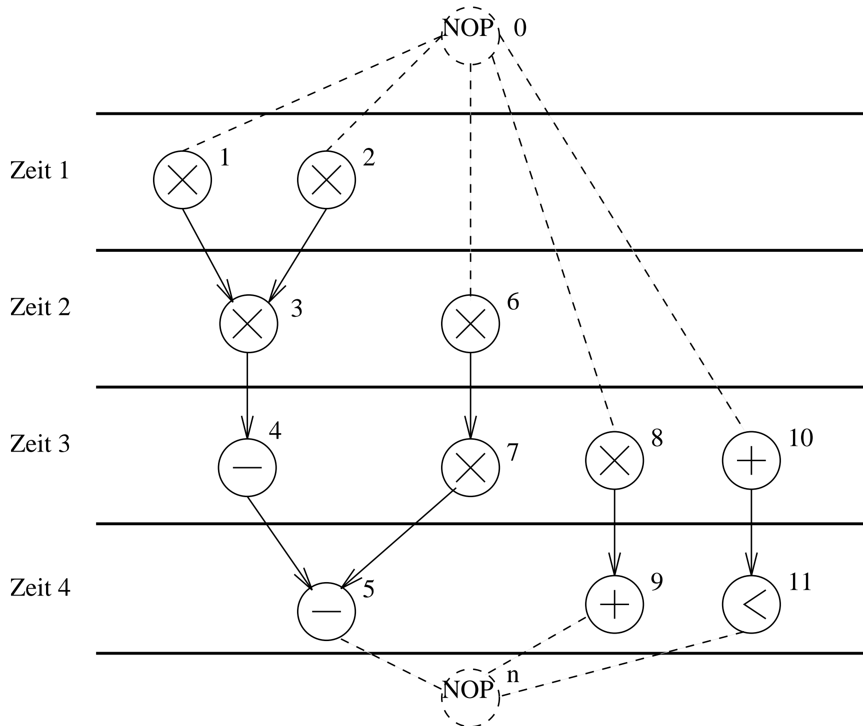


Abbildung 4.5: Ablaufplanung nach dem ALAP- Algorithmus mit Deadline $\bar{L} = 4$.

$\tau^L(v_i) := \min_j\{\tau(v_j)^L \mid (v_i, v_j) \in E_S\} - w_i$
}
UNTIL (v_0 geplant);
RETURN (τ^L)
}

Die durch den Algorithmus bestimmten Startzeitpunkte der Operationen bezeichnen wir mit $\tau^L = \{\tau^L(v_0), \tau^L(v_1), \ldots, \tau^L(v_n)\}$. Als Deadline $\bar{L}$ wählt man üblicherweise $\bar{L} = L = \tau^S(v_n) - \tau^S(v_0)$. Mit Hilfe des ALAP-Algorithmus kann man dann die *Mobilität* μ (engl. *slack*) der Operationen bestimmen als Differenz der Startzeitpunkte von ASAP- und ALAP-Algorithmus. Folglich gilt $\mu(v_i) = \tau^L(v_i) - \tau^S(v_i)\ \forall i \in \{0, 1, \ldots, n\}$. Operationen, für die die Mobilität null ist, können nur zu genau einem Zeitpunkt gestartet werden, damit die Deadline erfüllt werden kann. Solche Operationen heissen auch *kritisch*. Falls die Mobilität einer Operation grösser null ist, dann gibt sie die Grösse des Intervalls an, in dem die Operation gestartet werden kann, ohne die Deadline zu verletzen.

Beispiel 4.7 *Abb. 4.5 zeigt die Ablaufplanung des Sequenzgraphen aus Beispiel 4.6 mit dem ALAP-Algorithmus. Als Deadline $\bar{L}$ wurde die durch den ASAP-Algorithmus bestimmte Latenz $\bar{L} = 4$ gewählt. Zuerst wird $\tau(v_n)^L = 5$ gesetzt. Die Knoten, deren Nachfolger geplant sind, sind v_5, v_9 und v_{11}. Ihre Startzeit wird auf vier gesetzt und so weiter. Ein Vergleich der Ablaufpläne zeigt, dass die Mobilität der Operationen v_1, v_2, v_3, v_4 und v_5 null ist, d. h. sie befinden sich auf einem kritischen Pfad. Die Mobilität der anderen Knoten beträgt $\mu(v_6) = \mu(v_7) = 1$ und $\mu(v_8) = \mu(v_9) = \mu(v_{10}) = \mu(v_{11}) = 2$.*

Die Komplexität der ASAP- und ALAP-Algorithmen hängt von deren Implementierung ab, insbesondere von der Datenstuktur, mit der der Sequenzgraph dargestellt wird.

Im besten Fall ist die Komplexität gleich der des Problems der topologischen Sortierung (Tiefensuche in gerichteten, azyklischen Graphen) und damit $\mathcal{O}(|V_S| + |E_S|)$ [18].

4.2.2 Ablaufplanung mit Zeitbeschränkungen

Häufig möchte man zusätzlich zu den dem Problem inhärenten Datenabhängigkeiten weitere zeitliche Beschränkungen ausdrücken. Diese kann man unterteilen in

- *absolute Zeitbeschränkungen*: Hierzu zählen sogenannte *Deadlines* (= späteste Startzeitpunkte von Operationen) und *Releasezeiten* (= früheste Startzeitpunkte von Operationen). Deadlines und Releasezeiten sind also absolute Beschränkungen der Startzeiten von Operationen.

- *relative Zeitbeschränkungen*: Relative Beschränkungen drücken die zeitlichen Relationen zwischen Paaren von Operationen aus und sind unabhängig von deren absoluten Werten.

Beispielsweise garantiert eine relative Minimumsbeschränkung, dass eine Operation einer anderen erst nach einer gewissen Anzahl von Zeitschritten folgen darf. Dies gilt unabhängig davon, ob zwischen beiden Operationen Datenabhängigkeiten existieren. Eine Maximumsbeschränkung drückt aus, dass nur eine gewisse Anzahl von Zeitschritten zwischen den Startzeitpunkten zweier Operationen verstreichen darf. Kombinationen von Minimums- und Maximumsbeschränkungen erlauben die Formulierung exakter Distanzen zwischen Startzeitpunkten zweier Operationen, z. B. Gleichzeitigkeitsbeschränkungen. Absolute Zeitbeschränkungen können übrigens auch als Spezialfälle von relativen Zeitbeschränkungen zum Startzeitpunkt des Quellknotens gesehen werden.

Definition 4.7 (Relative Zeitbeschränkungen) *Eine relative Zeitbeschränkung zwischen zwei Operationen v_i und v_j, $v_i, v_j \in V_S$ eines Sequenzgraphen $G_S(V_S, E_S)$ drückt man durch Zahlen $l_{ij} \in \mathbf{Z}^+$ wie folgt aus:*

- *Minimumsbeschränkung:* $\tau(v_j) \geq \tau(v_i) + l_{ij}$,

- *Maximumsbeschränkung:* $\tau(v_j) \leq \tau(v_i) + l_{ij}$.

Beispiel 4.8 *Ein Busprotokoll verlange, dass eine Einheit genau drei Takte nach dem Einlesen eines Eingangsdatums ein entsprechendes Ausgangsdatum auf den Bus schreibt. Modelliert man Lese- und Schreiboperation als Knoten v_i und v_j, so kann man die Zeitbeschränkung der Spezifikation durch eine Minimums- und eine Maximumsbeschränkung gemäss Definition 4.7 mit $l_{ij} = l_{ji} = 3$ ausdrücken.*

Beispiel 4.9 *Gegeben sei das Komponentenverbindungsdiagramm in Abb. 4.6, das den Zugriff einer Schaltung auf einen Speicher darstellen soll. Spezifikation des Speichers:*

- *Adresse muss mindestens 1 Takt, darf aber höchstens 2 Takte anliegen.*
- *Daten erscheinen 1 Takt nach Anlegen der Adresse und sind 1 Takt lang gültig.*

Die zeitliche Spezifikation einer Schaltung für dieses Protokoll ist in Abb. 4.7 dargestellt.

Eine konsistente Modellierung von Minimums- und Maximumsbeschränkungen erhält man durch Definition eines sogenannten *Constraintgraphen* G_C.

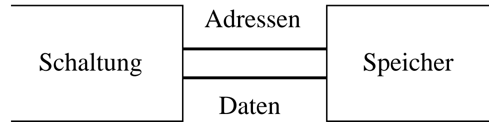


Abbildung 4.6: CCD eines Speicherzugriffs

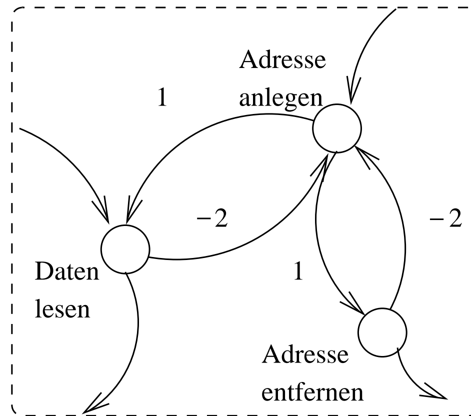


Abbildung 4.7: Modellierung eines Speicherzugriffprotokolls

Definition 4.8 (Constraintgraph) *Gegeben sei ein Sequenzgraph $G_S(V_S, E_S)$, ein Ressourcengraph $G_R(V_R, E_R)$ mit der Gewichtsfunktion $w : E_R \rightarrow \mathbf{Z}^+$, die jeder Kante $(v_s, v_t) \in E_R$ die Ausführungszeit $w_s = w((v_s, v_t))$ zuordnet, sowie eine Menge von n Minimums- und Maximumsbeschränkungen, gegeben durch Zahlen $l_{ij} \in \mathbf{Z}^+$. Ein* Constraintgraph *$G_C(V_C, E_C, d)$ ist ein kantengewichteter, gerichteter Graph mit Gewichtsfunktion $d : E_C \rightarrow \mathbf{Z}$, den man aus G_S erhält wie folgt:*

- *$V_C = V_S$; $E_C = E_S \cup E'$; E' beinhaltet jeweils eine Kante pro Minimums- bzw. Maximumsbeschränkung, also $|E'| = n$.*
- *E' bildet man wie folgt: Pro Minimumsbeschränkung l_{ij} gibt es eine Kante $(v_i, v_j) \in E'$ mit Gewicht $d((v_i, v_j)) = l_{ij}$; pro Maximumsbeschränkung l_{ij} gibt es eine Kante $(v_j, v_i) \in E'$ mit Gewicht $d((v_j, v_i)) = -l_{ij}$. Für die übrigen Kanten, d. h. Kanten $(v_i, v_j) \in E_S \subseteq E_C$, gilt $d((v_i, v_j)) = w_i$.*

Beispiel 4.10 *Betrachten wir den Sequenzgraphen G in Abb. 4.8 und die dort gegebene Knotennummerierung. Die Ausführungszeit einer Multiplikation betrage zwei Zeitschritte, die der Addition einen Schritt. Ferner soll Operation v_4 mindestens $l_{04} = 4$ Zeitschritte nach Operation v_0 starten und v_2 soll höchstens $l_{12} = 3$ Schritte nach v_1 starten. Der aus G nach der Konstruktionsvorschrift in Definition 4.8 gebildete Constraintgraph G_C ist in Abb. 4.8 dargestellt.*

Die Hinzunahme von Minimums- und Maximumsbeschränkungen führt zu dem Problem, feststellen zu können, ob es überhaupt einen Ablaufplan gibt, der sämtliche Bedingungen erfüllt. Ein einfaches Testkriterium besteht darin, dass man für jede Maximumsbeschränkung l_{ij} testet, ob der längste gewichtete Pfad zwischen Knoten v_i und Knoten v_j, der die minimale Zeitdistanz dieser beiden Knoten ausdrückt, kleiner oder gleich l_{ij} ist. Folglich ist eine notwendige Bedingung für die Existenz eines Ablaufplans, dass der

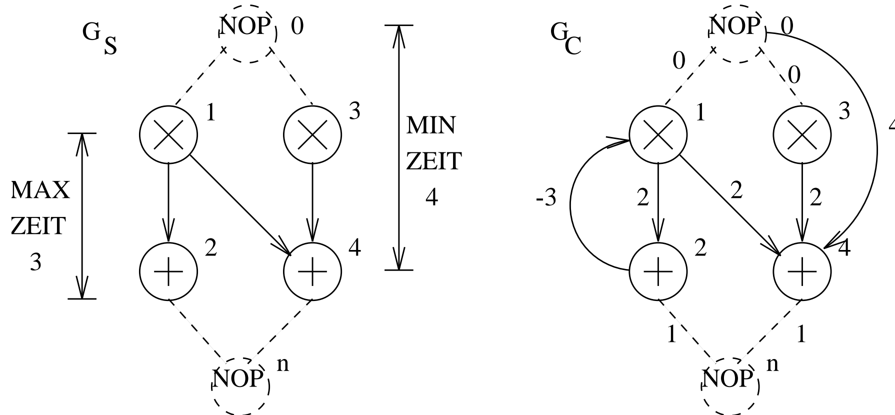


Abbildung 4.8: Beispiel eines Constraintgraphen mit Minimums- und Maximumsbeschränkungen.

Constraintgraph frei von positiven Zyklen ist. Man kann zeigen, dass diese Bedingung auch hinreichend ist [18]. Die Existenz eines Ablaufplans kann man also mit dem Bellman-Ford-Algorithmus in polynomieller Zeit ($\mathcal{O}(|V_S| \cdot |E_S|)$) feststellen.

Beispiel 4.11 *Für den Constraintgraphen in Abb. 4.8 erhält man den Ablaufplan $\tau(v_0) = \tau(v_1) = \tau(v_3) = 1$, $\tau(v_2) = 3$, $\tau(v_4) = 5$ und $\tau(v_n) = 6$. Knoten v_4 wird gegenüber dem Fall ohne Minimums- und Maximumsbeschränkungen um zwei Zeitschritte verzögert. Die Zeitdifferenz der Startzeitpunkte von v_1 und v_2 beträgt zwei und ist damit kleiner als die obere Schranke. Die Latenz beträgt $L = \tau(v_n) - \tau(v_0) = 5$.*

Zusammenfassung: Wir haben gesehen, dass im Falle unbeschränkter Ressourcen das Problem der Latenzminimierung in polynomieller Zeit gelöst werden kann für Sequenzgraphen (topologische Sortierung $\mathcal{O}(|V_S| + |E_S|)$). Wir haben dann absolute und relative Zeitbeschränkungen eingeführt und gezeigt, dass auch in diesem Fall die Existenz eines Ablaufplans sowie das Problem der Latenzminimierung in polynomieller Zeit gelöst werden kann durch Lösen des „Längster-Pfad-Problems“ (z. B. Bellman-Ford-Algorithmus $\mathcal{O}(|V_S| \cdot |E_S|)$).

Im Folgenden werden wir unsere Problemstellungen durch Hinzunahme von Ressourcebeschränkungen erweitern.

4.2.3 Ablaufplanung mit Ressourcenbeschränkungen

Das Problem der Ablaufplanung mit Ressourcebeschränkungen ist nicht nur im Bereich der Systemsynthese relevant, sondern wurde auch bereits intensiv im Bereich des Operations Research untersucht. Es wurde gezeigt, dass das Problem der latenzoptimalen Ablaufplanung eines Sequenzgraphen schon im einfachen Fall nur eines vorhandenen Ressourcentyps NP-hart ist [19]. Dieses Resultat gilt sogar für den Fall, dass die Berechnungszeiten aller Operationen 1 sind.

Um die exponentielle Laufzeit exakter Verfahren zu vermeiden, wurden zahlreiche heuristische Verfahren mit polynomieller Laufzeit entwickelt. Wir wollen zunächst einige Heuristiken untersuchen und anschliessend als Alternative ein exaktes Verfahren vorstellen, das auf der Formulierung als ganzzahliges lineares Programm (integer linear program (ILP)) beruht.

Im Rahmen der ressourcenbeschränkten Ablaufplanung kann man grundsätzlich zwei Arten der Planung betrachten:

- *Latenzminimierung mit Ressourcebeschränkungen:* Dieses Problem kann nach unserer Modellierung wie folgt formuliert werden:

  Definition 4.9 (Latenzminierung mit Ressourcenbeschränkungen) *Gegeben sind ein nichthierarchischer Sequenzgraph $G_S = (V_S, E_S)$ und ein Ressourcengraph $G_R = (V_R, E_R)$ mit der Gewichtsfunktion $w : E_R \rightarrow \mathbf{Z}^+$, die jeder Kante $(v_s, v_t) \in E_R$ die Ausführungszeit $w_s = w((v_s, v_t))$ zuordnet. Unter* Latenzminimierung mit Ressourcenbeschränkungen *versteht man das Problem*

$$\begin{aligned}\min\{\tau(v_n) - \tau(v_0) \quad | \quad & \tau(v_j) - \tau(v_i) \geq w_i \;\; \forall (v_i, v_j) \in E_S \;\wedge \\ & |\{v_s : \beta(v_s) = v_k \;\wedge\; \tau(v_s) \leq t \leq \tau(v_s) + w_s\}| \leq \alpha(v_k) \\ & \forall v_k \in V_T, \forall 1 \leq t \leq \bar{L}\}\end{aligned}$$

- *Kostenminimierung unter Latenzbeschränkungen:* Hier wird ein Ablaufplan gesucht, der bei Vorgabe einer Latenzbeschränkung $\bar{L}$ minimale Kosten (ausgedrückt beispielsweise durch die Anzahl benötigter Ressourcen) verursacht.

Wir betrachten hier nur das erste Problem der Latenzminimierung mit Ressourcenbeschränkungen. Es sei jedoch bemerkt, dass sich für die folgenden Heuristiken auch ein Pendant zur Lösung des zweiten Problems in entsprechender Weise formulieren lässt (Uebung!).

Erweiterte ASAP- und ALAP-Verfahren

Verfeinerungen von ASAP- und ALAP-Algorithmen führten zu modifizierten Verfahren [20, 21, 22], die auch Ressourcebeschränkungen berücksichtigen können. Der Ausgangspunkt ist jeweils ein Ablaufplan, der mit dem ASAP- oder ALAP-Verfahren ermittelt wurde. In jedem Schritt des Algorithmus wird anschliessend überprüft, ob die Ressourcebeschränkungen eingehalten werden. Wird eine Ressourcebeschränkung verletzt, so werden entsprechend viele Operationen auf einen späteren (ASAP) bzw. einen früheren Zeitschritt (ALAP) verlagert. Diese Erweiterung nennt man auch *ASAP/ALAP-Planung mit bedingter Verschiebung*, ein iterativ konstruktives Verfahren.

Beispiel 4.12 *Das erweiterte ASAP-Verfahren angewendet auf den Sequenzgraphen in Beispiel 4.3 mit den Ressourcebeschränkungen $\alpha(r_1) = 2$, $\alpha(r_2) = 2$ führt zu dem Ablaufplan in Abb. 4.9.*

List Scheduling

Auf Listen basierte Verfahren zur Ablaufplanung wurden zuerst im Bereich der Kompaktierung von Mikrocode [23] entwickelt. In diesem Anwendungsbereich werden mehrere Mikroinstruktionen parallel als Mikrooperation ausgeführt, um möglichst kompakte und damit schnelle Mikroprogramme zu erzeugen.

Diese Verfahren können auch als Weiterentwicklungen der ASAP-Verfahren aufgefasst werden. Im Gegensatz zu diesen werden jedoch globale Kriterien benutzt, um die Reihenfolge der Operationen zu bestimmen, für die der Ablaufplan festgelegt wird. Wie beim

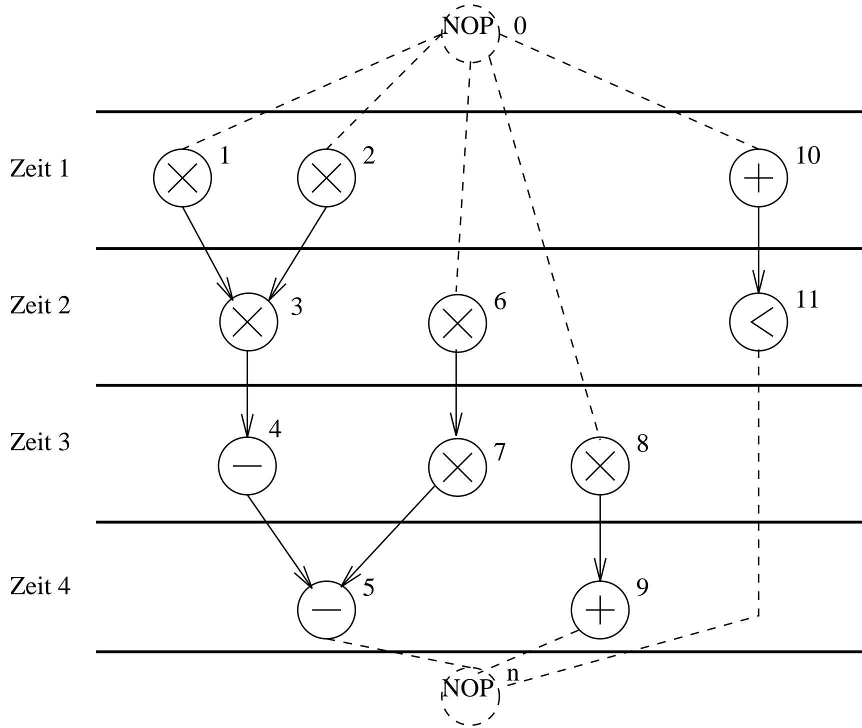


Abbildung 4.9: Modifizierte ASAP-Ablaufplanung zur Berücksichtigung von Ressourcebeschränkungen.

ASAP-Verfahren werden die Operationen schrittweise vom ersten bis zum letzten Zeitschritt geplant. Es handelt sich folglich um ein iterativ konstruktives Verfahren.

Zu Beginn des Verfahrens werden die Knoten des Sequenzgraphen topologisch sortiert. Anschliessend wird für jeden Knoten eine Priorität berechnet. Als Kriterien hierfür wurde die Mobilität einer Operation als Differenz ihrer Startzeitpunkte nach ASAP- und ALAP-Planung [24, 25] vorgeschlagen. Nach dieser Rechnung wird das eigentliche Planungsverfahren durchgeführt. In jedem Iterationsschritt t wird die Menge $U_{t,k}$ an Kandidaten des Ressourcentyps k bestimmt, deren Vorgänger im Sequenzgraphen alle zum Zeitpunkt t beendet sind:

$$U_{t,k} = \{v_i \in V_S : \beta(v_i) = v_k \ \wedge \forall j : (v_j, v_i) \in E_S : t \geq \tau(v_j) + w_j\}$$

Im gleichen Schritt bestimmt man die Menge $T_{t,k}$ an Operationen des Ressourcentyps k, die im Zeitschritt t ausgeführt werden, also die zu einem Zeitpunkt vor t gestartet worden sind, aber noch nicht beendet sind und damit eine Ressource belegen zum Zeitpunkt t:

$$T_{t,k} = \{v_i \in V_S : \beta(v_i) = v_k \ \wedge t > \tau(v_i) > t - w_i\}$$

Dann erfolgt die Ablaufplanung: Für jeden Ressourcentyp k wird basierend auf der festgelegten Prioritätsliste der Operationen, beschreibbar durch eine Funktion $p : V_S \rightarrow \mathbf{Z}^+$, eine Menge S_t an Operationen mit maximaler Priorität aus der Kandidatenmenge $U_{t,k}$ ausgewählt und geplant. Dabei kann S_t maximal so viele Operationen enthalten, wie es die Ressourcenbeschränkung $|S_t| + |T_{t,k}| \leq \alpha(v_k)$ erlaubt. Der ganze Algorithmus lässt sich damit wie folgt formulieren:

```
LIST(G_S(V_S, E_S),G_R(V_R, E_R),a,p){
    t := 1;
    REPEAT {
        FOR k=1 to |V_R| {
            Bestimme Kandidatenmenge U_{t,k};
            Bestimme Menge nicht beendeter Operationen T_{t,k};
            Wähle eine Menge maximaler Priorität S_t ⊆ U_{t,k}:
            |S_t| + |T_{t,k}| ≤ α(v_k)
            τ(v_i) := t  ∀i : v_i ∈ S_t;
        }
        t := t + 1;
    }
    UNTIL (v_n geplant);
    RETURN (τ);
}
```

Die Berechnungskomplexität dieses Algorithmus ist $\mathcal{O}(|V_S|)$. Der Algorithmus konstruiert einen Ablaufplan, der per Definition die Ressourcebeschränkungen erfüllt. Das heuristische Dringlichkeitsmass wird statisch vor Ablauf des Algorithmus bestimmt.

Beispiel 4.13 *Wir betrachten erneut den Ablaufplan in Abb. 4.1. Der Ressourcengraph zur Modellierung der Ressourcebeschränkungen wurde in Beispiel 4.2 beschrieben und ist in Abb. 4.3 dargestellt. Es gibt zwei Ressourcentypen (r_1: Multiplizierer, r_2: ALU (Addition, Subtraktion und Vergleichsbildung)) mit Allokation $\alpha(r_1) = 1$, $\alpha(r_2) = 1$. Die in Abb. 4.2 dargestellte Ablaufplanung entspricht der durch List-Scheduling berechneten Ablaufplanung mit der durch den längsten Pfad bestimmten Prioritätsliste $p(v_1) = p(v_2) = 4$, $p(v_3) = p(v_6) = 3$, $p(v_4) = p(v_7) = p(v_8) = p(v_{10}) = 2$ und $p(v_5) = p(v_9) = p(v_{11}) = 1$. In diesem Beispiel gilt $T_{t,k} = 0 \; \forall t, k$, weil $w_i = 1 \; \forall v_i \in V_S$. Die erzielte Latenz ist in diesem Fall optimal.*

Um zu zeigen, dass es bereits einfache Beispiele gibt, bei denen List-Scheduling suboptimale Lösungen erzeugt, wollen wir ein weiteres Beispiel anfügen.

Beispiel 4.14 *Wir betrachten den Sequenzgraph in Abb. 4.10a und den zugehörigen Ressourcengraphen in Abb. 4.10b mit der Allokation $\alpha(r_1) = 1$, $\alpha(r_2) = 1$. In Abb. 4.11a sehen wir den mit List-Scheduling gefundenen Ablaufplan. Abb. 4.11b zeigt, dass es einen Ablaufplan gibt, der eine um einen Zeitschritt geringere Latenz aufweist.*

Man kann zeigen, dass der List-Scheduling-Algorithmus exakt ist, wenn der Sequenzgraph (nach Streichen des Anfangsknotens) ein Baum ist und alle Berechnungszeiten 1 sind.

Ganzzahlige lineare Programmierung

Eine weitere Möglichkeit, Ablaufpläne unter Berücksichtigung endlicher Ressourcen zu bestimmen, besteht in der Formulierung als ganzzahlige lineares Programm. Eine der ersten Arbeiten, in der formale Methoden beschrieben wurden, um digitale Logik mit algebraischen Relationen auf der Ebene von Datenpfadsystemen zu modellieren und Register-Transfer-Logik zu synthetisieren, wurde von Hafer und Parker ([27]) veröffentlicht. Zahlreiche weitere Arbeiten bauen direkt oder indirekt auf dem von den Autoren vorgeschlagenen Ansatz auf, z. B. [28, 29, 30, 31].

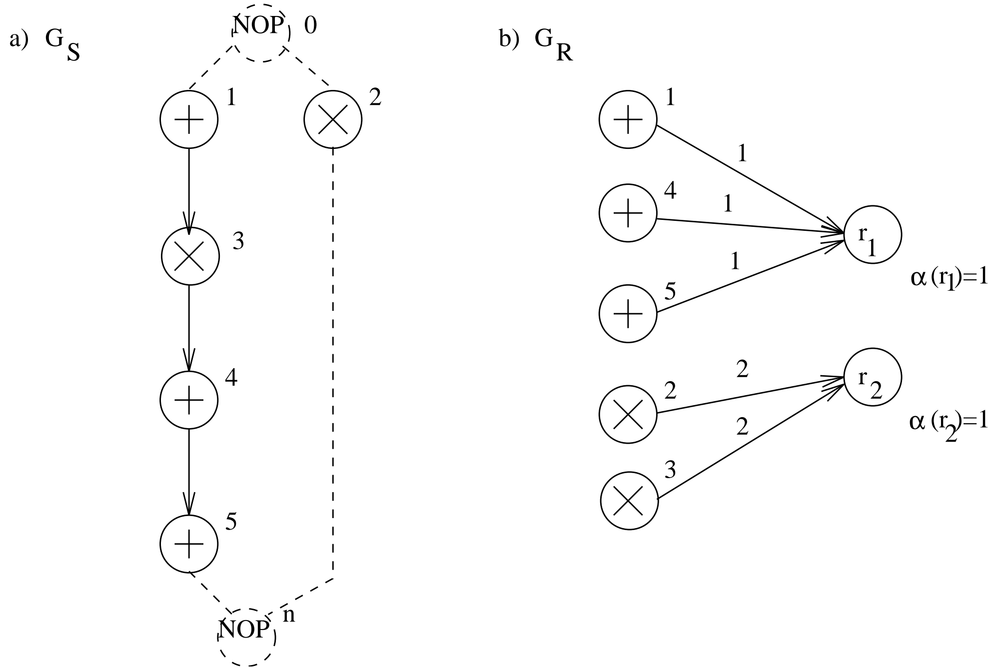


Abbildung 4.10: Sequenzgraph und Ressourcengraph aus Beispiel 4.14.

Theorem 4.1 (Ablaufplanung mit Ressourcenbeschränkungen) *Gegeben seien ein Sequenzgraph G_S, ein Ressourcengraph G_R, die Zahl der verfügbaren Instanzen $\alpha(v_t)$ des Ressourcentypen v_t, die Ausführungszeiten w_s der Operationen v_s und die mit ASAP- und ALAP-Planungsverfahren berechneten frühest- bzw. spätestmöglichen Ausführungszeitpunkte $l_s := \tau^S(v_s)$ und $h_s := \tau^L(v_s)$ der Operationen $v_s \in V_S$. Eine zulässige Lösung des folgenden Ungleichungssystems ist dann eine Lösung des zulässigen Planungsproblems:*

$$x_{i,t} \in \{0,1\} \qquad \forall v_i \in V_S,\ \forall t : l_i \leq t \leq h_i$$

$$\sum_{t=l_i}^{h_i} x_{i,t} = 1 \qquad \forall v_i \in V_S \tag{4.2}$$

$$\sum_{t=l_i}^{h_i} t \cdot x_{i,t} = \tau(v_i) \qquad \forall v_i \in V_S \tag{4.3}$$

$$\tau(v_j) - \tau(v_i) \geq w_i \qquad \forall (v_i, v_j) \in E_S \tag{4.4}$$

$$\sum_{i:(v_i,v_k)\in E_R} \sum_{p'=\max\{0,t-h_i\}}^{\min\{w_i-1,t-l_i\}} x_{i,t-p'} \leq \alpha(v_k)$$

$$\forall v_k \in V_T, \forall t : 1 \leq t \leq \max\{h_i : v_i \in V_S\} \tag{4.5}$$

Es folgt eine kurze Erläuterung zu den einzelnen Gleichungen und Ungleichungen:

- Die binäre Variable $x_{i,t}$ drückt die Ablaufplanung der Operationen $v_i \in V_S$ aus:

$$x_{i,t} = \begin{cases} 1 & \text{falls } v_i \text{ zum Zeitpunkt } \tau(v_i) = t \text{ ausgeführt wird} \\ 0 & \text{sonst} \end{cases}$$

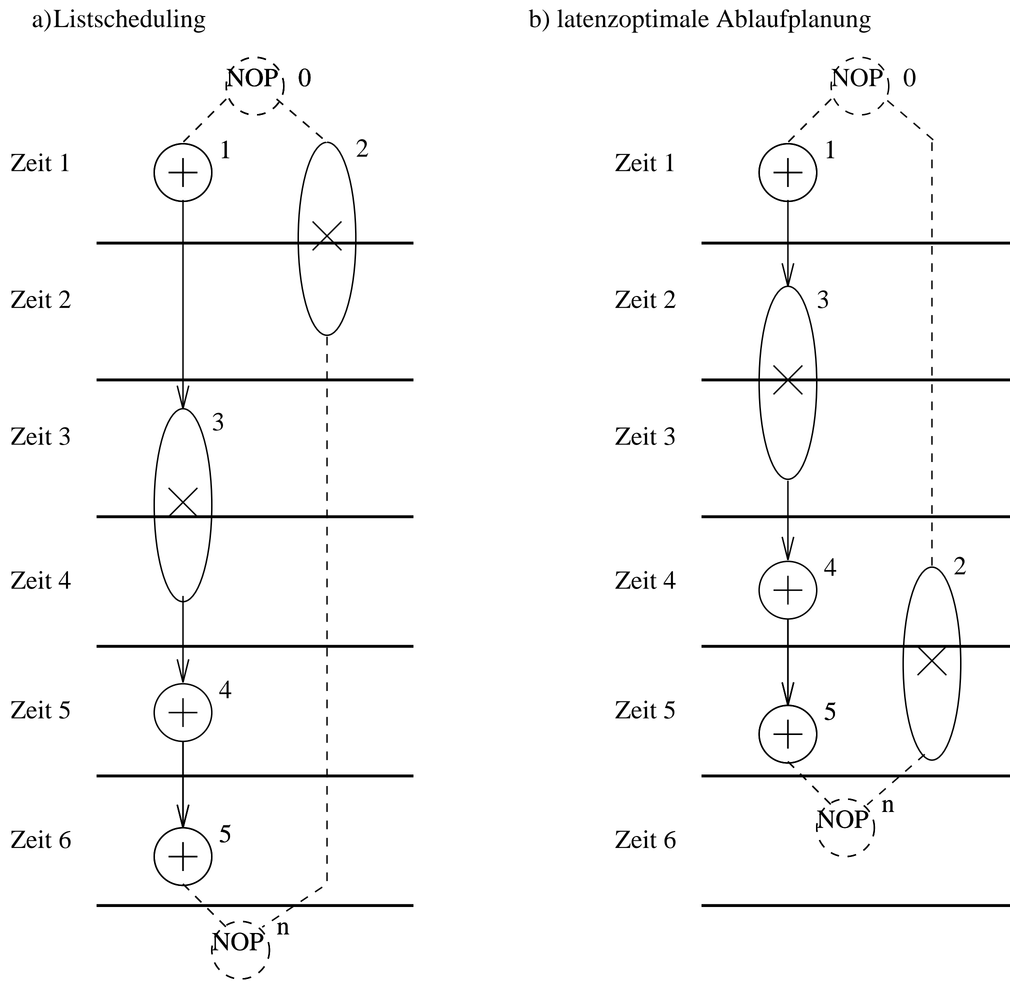


Abbildung 4.11: List-Scheduling ist im Allgemeinen suboptimal.

Der Gültigkeitsbereich der Variablen $x_{i,t}$ erstreckt sich jeweils vom frühest- bis zum spätestmöglichen Kontrollschritt, an dem die Ausführung von Operation v_i begonnen werden kann.

- Da $x_{i,t}$ den Wert 1 genau zu dem Zeitpunkt annimmt, an dem v_i ausgeführt wird, wird durch die Summation von $x_{i,t}$ in (4.2) über alle möglichen Ausführungszeitpunkte t mit $l_i \leq t \leq h_i$ gewährleistet, dass v_i genau einmal während einer Iteration ausgeführt wird.

- Die Eigenschaft, dass $x_{i,t}$ den Wert 1 am Ausführungszeitpunkt $\tau(v_i) = t$ annimmt, wird in (4.3) ausgenutzt, um $\tau(v_i)$ explizit zu berechnen. Das Produkt $x_{i,t} \cdot t$ hat zum Zeitpunkt $t = \tau(v_i)$ den Wert $\tau(v_i)$ und ansonsten den Wert 0. Die Summation über das Produkt $x_{i,t} \cdot t$ liefert demzufolge genau $\tau(v_i)$.

- Das Planungsverfahren muss die durch den Sequenzgraphen G_D implizierten Datenabhängigkeiten gewährleisten: Für alle Kanten $(v_i, v_j) \in E_S$ muss gelten, dass die Ausführung von Operation v_j erst begonnen werden kann, wenn die Berechnung von Operation v_i beendet ist. Die Differenz der Ausführungszeitpunkte von v_j und v_i muss also mindestens so gross wie die Dauer der Ausführung von v_i sein (Gleichung 4.4).

- Schliesslich muss garantiert werden, dass zu keinem Zeitpunkt mehr als die zur Verfügung stehenden Instanzen der Ressourcentypen benutzt werden. Für die innere Summe in (4.5) gilt durch die Indexverschiebung mit p':

  $$\sum_{p'=0}^{w_i-1} x_{i,t-p'} = \begin{cases} 1 & : \quad \forall t : \tau(v_i) \leq t \leq \tau(v_i) + w_i - 1 \\ 0 & : \quad \text{sonst} \end{cases}$$

  Diese Summe hat also genau während der Ausführungszeit einer Operation v_i, das heisst für alle Kontrollschritte t mit $\tau(v_i) \leq t \leq \tau(v_i) + w_i - 1$, jeweils den Wert 1. Durch Summation über alle Operationen, die denselben Ressourcentypen v_k beanspruchen, erhält man für einen festen Zeitpunkt t die Anzahl der Operationen, die auf den Ressourcentypen v_k zu dem betreffenden Zeitpunkt zugreifen. Es seien jeweils $\alpha(v_k)$ Instanzen von dem Ressourcentypen $v_k \in V_T$ vorhanden. Mithin gewährleistet das Ungleichungssystem, dass für alle Kontrollschritte t und für alle Ressourcentypen $v_k \in V_T$ niemals mehr als die zur Verfügung stehende Anzahl von Instanzen eines Ressourcentyps benötigt wird. Die etwas komplexeren Summengrenzen $\max\{0, t-h_i\}$ bzw. $\min\{w_i-1, t-l_i\}$ stellen sicher, dass nur definierte Variablen $x_{i,t-p'}$ aufsummiert werden.

Durch Auswertung von Ungleichungssystem (4.5) für alle Kontrollschritte t mit $1 \leq t \leq \gamma$ wird ein Benutzungsprofil eines Ressourcentyps berechnet. Ein derartiges Benutzungsprofil wird im folgenden Beispiel dargestellt.

Beispiel 4.15 *Gegeben sei der Ressourcengraph nach Abbildung 4.12 mit den Operationen v_1 und v_2, die die Berechnungszeiten von 4 und 3 Kontrollschritten auf Ressource r haben. Die Startzeitpunkte der Ausführung der Operationen seien $\tau(v_1) = 2$ und $\tau(v_2) = 4$. Durch Auswertung der inneren Summe des Ungleichungssystems (4.5) erhält man die beiden linken Diagramme in Abbildung 4.13, während die Summe für jeden Kontrollschritt im rechten Teil zu sehen ist. Offensichtlich gibt der zu jedem Kontrollschritt zugehörige Wert die Anzahl der benötigten Ressourcen an.*

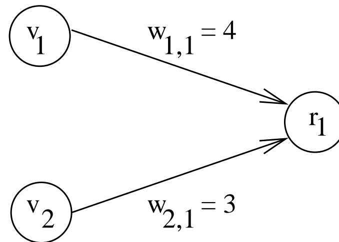


Abbildung 4.12: Ressourcengraph aus Beispiel 4.15.

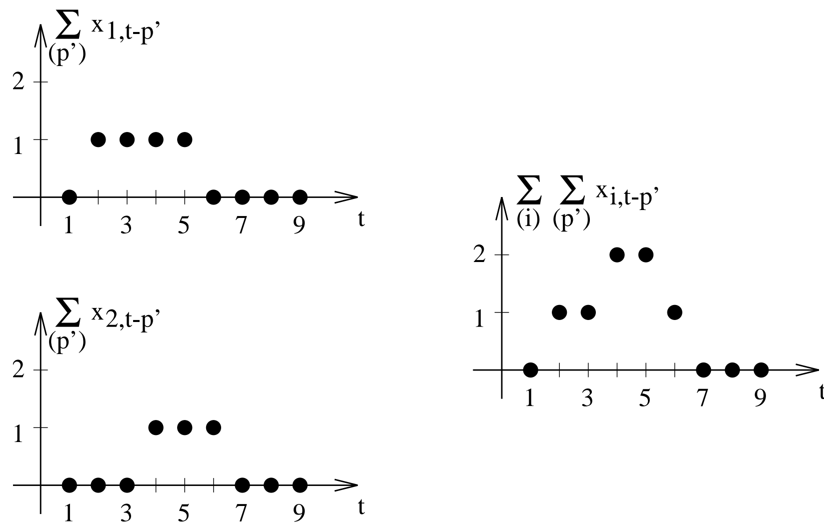


Abbildung 4.13: Berechnung der Ressourceausnutzung in Beipspiel 4.15.

Iterative Algorithmen

Die bisher betrachtete Klasse von Algorithmen konnte durch (azyklische) Sequenzgraphen beschrieben werden. Das heisst, alle Operationen wurden genau einmal ausgeführt. Aber zum Beispiel Algorithmen der Bild- und Signalverarbeitung sind dadurch gekennzeichnet, dass bestimmte Operationen immer wieder auf jeweils unterschiedliche Daten angewendet werden. Beim Beispiel des Hybridcoders in der Einleitung hiess dies, dass für jedes Bild einer Videosequenz die Kodierschleife zu durchlaufen war.

Es liegt daher nahe, die Klasse der Algorithmen wie folgt zu erweitern:

Definition 4.10 (Iterative Algorithmen) *Ein iterativer Algorithmus besteht aus einer Menge von V quantifizierten, linear indizierten Gleichungen $S_i[l]$:*

$$S_1[l] \ldots S_i[l] \ldots S_V[l] \qquad \forall l \geq 0$$

Jede Gleichung $S_i[l]$ ist von der Form

$$x_i[l] = \mathbf{F}_i[\ldots, x_j[l - d_{ji}], \ldots]$$

wobei der Index l die Iteration des Algorithmus angibt. Die Variablen $x_i[l]$ sind skalar indiziert und $\mathbf{F}_i$ sind beliebige Funktionen. Zwischen der Berechnung von Variablen $x_i[l]$ und $x_j[l]$ können konstante Indexverschiebungen $d_{ji} \in \mathbf{N}$ bestehen.

Die Gleichungen $S_i[l]$ werden iterativ für alle $l \in \mathbf{Z}^+$ ausgewertet. Eine Indexverschiebung d_{ji} bewirkt, dass die Berechnung der Variablen x_i um mindestens d_{ji} Iterationen gegenüber der Berechnung der Variablen x_j verzögert ist.

Beispiel 4.16 *Bei dem einfachen iterativen Algorithmus*

$$\begin{aligned} x_2[l] &= f_2(x_1[l]) && \forall l \geq 0 \\ x_3[l] &= f_3(x_1[l]) && \forall l \geq 0 \\ x_4[l] &= f_4(x_2[l], x_3[l]) && \forall l \geq 0 \end{aligned}$$

besteht zwischen der Berechnung der Variablen x_2 und x_1 eine Indexverschiebung um eine Iteration. Dieser Algorithmus lässt sich natürlich auch in Form eines imperativen Schleifenprogramms darstellen, zum Beispiel in VHDL:

```
...
LOOP
  x2(l) := f2(x1(l));
  x3(l) := f3(x1(l));
  x4(l) := f4(x2(l), x3(l));
  l := l+1
END LOOP;
...
```

Die Struktur und Datenabhängigkeiten eines iterativen Algorithmus lassen sich wiederum durch einen Sequenzgraphen darstellen. Zusätzlich sind jedoch noch die den Kanten zugeordneten Indexverschiebungen zu definieren. Die Funktion $d : E_S \rightarrow \mathbf{Z}$ ordnet jeder Kante (v_j, v_i) des Sequenzgraphen die zugehörige Indexverschiebung d_{ji} zu.

Beispiel 4.17 *Der um die Indexverschiebungen erweiterte Sequenzgraph des im Beispiel 4.16 eingeführten Algorithmus ist in Abb. 4.14 dargestellt.*

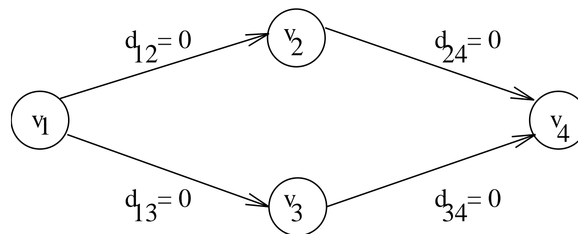


Abbildung 4.14: Beispiel eines erweiterten Sequenzgraphen.

Es folgen noch einige weitere Definitionen, die für das Verständnis iterativer Algorithmen und deren Implementierung wichtig sind. Zunächst soll der Begriff der Iteration genauer erläutert werden.

Definition 4.11 (Iteration) *Eine Iteration ist die Berechnung aller Variablen $x_i[l]$ eines nach Definition 4.10 gegebenen iterativen Algorithmus für ein festes l.*

Während einer Iteration wird daher der entsprechende Algorithmus genau einmal vollständig abgearbeitet. Es werden die zu einem vollständigen Satz von Eingabedaten gehörigen Ausgangsdaten berechnet. Unter einer Iteration versteht man demnach die Ausführung einer Einheits-Berechnungsaufgabe.

Bei der Abbildung eines Algorithmus auf eine Architektur ist die Verarbeitungsgeschwindigkeit ein wichtiges Leistungsmerkmal. Das Iterationsintervall und die Latenz sind zwei wichtige Begriffe, die die Ausführungszeit von Algorithmen betreffen.

Definition 4.12 (Iterationsintervall) *Das* Iterationsintervall *P bezeichnet die Anzahl von Taktzyklen zwischen dem Beginn zweier aufeinanderfolgender Iterationen.*

Da zu jeder Iteration ein neuer Satz von Eingangsdaten verarbeitet wird, ist das Iterationsintervall demnach das Inverse der Durchsatzrate. Es ist mithin ein äusserst wichtiges Mass für den Datendurchsatz und die Verarbeitungsgeschwindigkeit einer Implementierung.

Neben der Rate, mit der neue Daten einem System zugeführt werden können, ist auch die Latenz wichtig, die Zeit, die zwischen der ersten und letzten Operation einer Iteration vergeht.

Beispiel 4.18 *Die Begriffe Iteration, Iterationsintervall und Latenz werden anhand von Abb. 4.15 veranschaulicht. Es wird vorausgesetzt, dass die Ausführungszeit der Operationen jeweils einen Taktzyklus benötigt. Die Iterationen werden mit einem Iterationsintervall von 1 Taktzyklus abgearbeitet, da zu jedem Taktzeitpunkt eine neue Iteration gestartet wird. Da die Abarbeitung einer Iteration 4 Taktzyklen erfordert, beträgt die Latenz 4 Taktzyklen.*

Funktionale Fliessbandverarbeitung und Schleifenfaltung

Die so beschriebenen iterativen Algorithmen zeichnen sich durch inhärenten Parallelismus auf zwei Ebenen aus. Zum einen können die Werte zweier Variablen $x_i[l]$ und $x_j[l]$ innerhalb einer Iteration l parallel berechnet werden, wenn keine direkten Datenabhängigkeiten zwischen den Operationen besteht. Zum anderen kann, bevor eine Iteration beendet ist, eine neue Iteration begonnen werden. Dies führt dazu, dass Variablen aus verschiedenen Iterationen parallel berechnet werden können und mehrere Operationen gleichzeitig

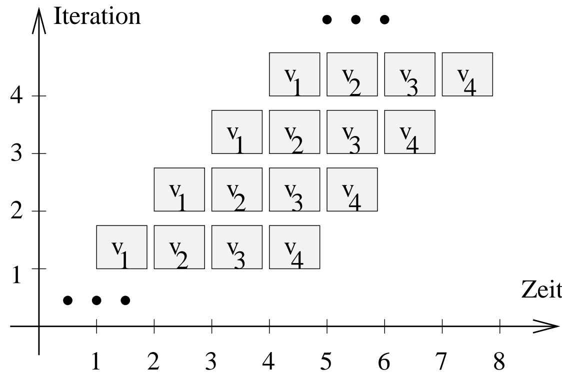


Abbildung 4.15: Iterationsintervall und Latenz.

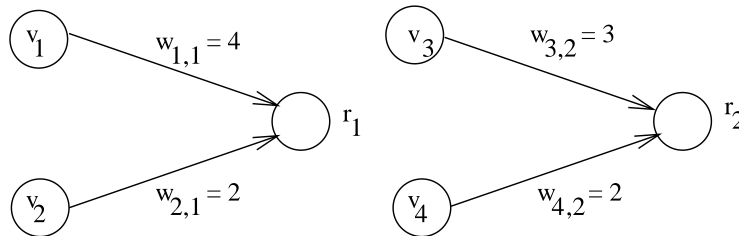


Abbildung 4.16: Ressourcengraph aus Beispiel 4.19.

berechnet werden. Dies ist auch unter dem Begriff *funktionale Fliessbandverarbeitung* bekannt; in Abb. 4.15 sieht man, dass Operation v_2 der ersten Iteration parallel zu Operation v_1 der zweiten Iteration ausgeführt wird.

Definition 4.13 (Funktionale Fliessbandverarbeitung) *Gegeben seien ein iterativer Algorithmus nach Definition 4.10 und ein Iterationsintervall P nach Definition 4.12. Unter funktionaler Fliessbandverarbeitung (Makrofliessbandverarbeitung, functional pipelining, macro pipelining) versteht man die gleichzeitige Bearbeitung von zu unterschiedlichen Iterationen gehörenden Datensätzen. Es gibt also Kontrollschritte, an denen Operationen mit Daten aus verschiedenen Iterationen ausgeführt werden.*

Im Fall unbegrenzter Ressourcen ist das minimal erreichbare Iterationsintervall (und damit die maximal erreichbare Durchsatzrate) durch das Maximum aller Rechenzeiten gegeben: $P_{\min} = \max\{w((v_s, v_t))\} \;\; \forall (v_s, v_t) \in E_R$. Ein Beispiel für funktionale Fliessbandverarbeitung wird im Folgenden gegeben.

Beispiel 4.19 *Gegeben seien der Abhängigkeitsgraph nach 4.14 und der zugehörige Ressourcengraph nach Abbildung 4.16. Ein nach Ungleichungssystem (4.2,4.4,4.5) berechneter zulässiger Ablaufplan für diese Konfiguration ist in Tabelle 4.1 dargestellt. Nach diesem Ablaufplan beträgt das Iterationsintervall $P = 9$ Kontrollschritte; das heisst alle 9 Kontrollschritte kann die Berechnung eines neuen Datensatzes begonnen werden. Aus Abbildung 4.17, die die Belegung der Ressourcen durch die Operationen darstellt, geht hervor, dass Ressource r_2 während der Kontrollschritte 0 bis 3 jeder Iteration nicht arbeitet. Nach Berechnung von Operation v_2 auf Ressource r_1 ist es bei geeigneter Speicherung der Zwischenergebnisse offensichtlich möglich, gleichzeitig mit dem Beginn der Berechnung von Operation v_4 auf Ressource r_2, die Berechnung eines* neuen Daten*satzes durch Operation v_1 auf Ressource r_1 zu beginnen. Anders ausgedrückt bedeutet dies, dass*

P	9			
v_i	v_1	v_2	v_3	v_4
$\beta(v_i)$	r_1	r_1	r_2	r_2
$\tau(v_i)$	0	4	4	7

Tabelle 4.1: Ein einfacher Ablaufplan.

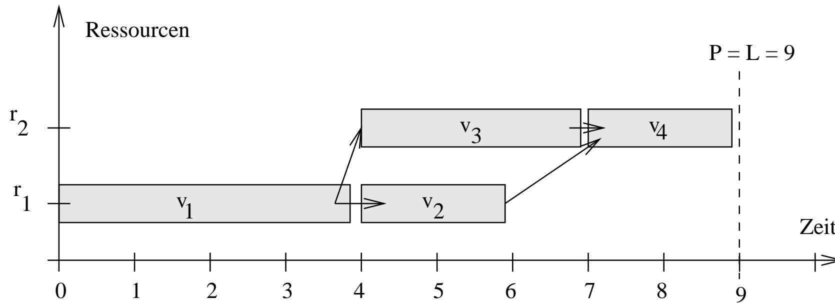


Abbildung 4.17: Ablaufplanung und Bindung.

die Operationen v_1, v_2, v_3 in Iteration l und Operation v_4 in Iteration $l+1$ berechnet werden. Ein Ablaufplan, der funktionale Fliessbandverarbeitung berücksichtigt, ist in Tabelle 4.2 dargestellt. Das Iterationsintervall beträgt jetzt $P = 7$ Kontrollschritte. Abbildung 4.18 zeigt die entsprechende Auslastung der Ressourcen.

P	7			
v_i	v_1	v_2	v_3	v_4
$\beta(v_i)$	r_1	r_1	r_2	r_2
$\tau(v_i)$	0	4	4	7/0

Tabelle 4.2: Ein Ablaufplan mit funktionaler Fliessbandverarbeitung.

Eine Erweiterung von funktionaler Fliessbandverarbeitung stellt die Schleifenfaltung dar.

Definition 4.14 (Schleifenfaltung) *Bei einem Ablaufplan mit Schleifenfaltung und funktionaler Fliessbandverarbeitung können Anfangs- und Endzeitpunkte der Ausführung einer Operation in aufeinanderfolgenden Iterationen liegen.*

Ein Beispiel für funktionale Fliessbandverarbeitung mit Schleifenfaltung sei im Folgenden gegeben:

Beispiel 4.20 *Gegeben seien der Abhängigkeitsgraph nach 4.14 und der zugehörige Ressourcengraph nach Abbildung 4.16. Das Iterationsintervall kann auf $P = 6$ Kontrollschritte gesenkt werden, wenn der Beginn der Ausführung von Operation v_3 an Kontrollschritt $t = 4$ beginnt und an Kontrollschritt $t = 1$ des darauffolgenden Abarbeitungszyklus endet. Entsprechend verschiebt sich der Beginn von Operation v_4 auf Kontrollschritt $t = 1$. Der Ablaufplan ist in Tabelle 4.3 dargestellt, die entsprechende Belegung der Ressourcen in Abbildung 4.19.*

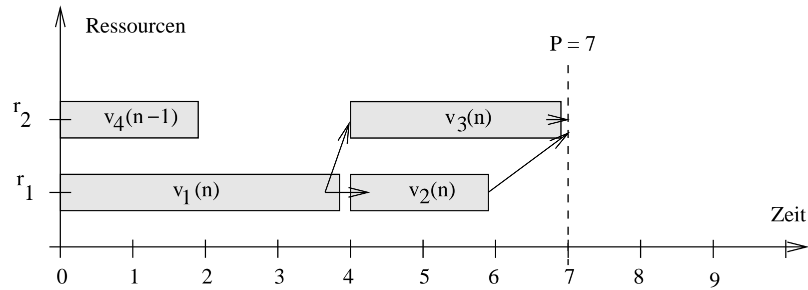


Abbildung 4.18: Ablaufplanung mit funktionaler Fliessbandverarbeitung.

P	6			
v_i	v_1	v_2	v_3	v_4
$\beta(v_i)$	r_1	r_1	r_2	r_2
$\tau(v_i)$	0	4	4	7/1

Tabelle 4.3: Ein Ablaufplan mit funktionaler Fliessbandverarbeitung und Schleifenfaltung

Ganzzahlige lineare Programmierung zur Ablaufplanung iterativer Algorithmen

Das in Theorem 4.1 beschriebene Verfahren zur Ablaufplanung mit endlichen Ressourcen wird nun auf iterative Algorithmen erweitert. Zunächst wollen wir jedoch davon ausgehen, dass es keine Datenabhängigkeiten zwischen Iterationen gibt, also $d_{ij} = 0$ gilt für alle Kanten $(v_i, v_j) \in E_S$ des Sequenzgraphen G_S.

Theorem 4.2 (Fliessbandverarbeitung und Schleifenfaltung) *Gegeben seien ein Sequenzgraph G_S mit $d_{ij} = 0$ für alle $(v_i, v_j) \in E_S$, ein Ressourcengraph G_R, das Iterationsintervall P, die maximal erlaubte Latenz L mit $P < L$ und die mit ASAP- und ALAP-Planungsverfahren berechneten frühest- beziehungsweise spätestmöglichen Ausführungszeitpunkte l_i und h_i der Operationen $v_i \in V_S$. Wird in dem Ungleichungssystem (4.2, 4.4, 4.5) das Ungleichungssystem (4.5) durch das folgende System (4.6) ersetzt, so ist eine zulässige Lösung des resultierenden Ungleichungssystems (4.2,4.4,4.6) eine Lösung des Planungsproblems mit funktionaler Fliessbandverarbeitung und Schleifen-*

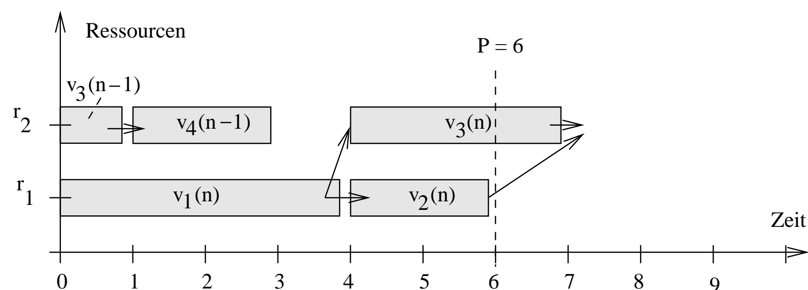


Abbildung 4.19: Ablaufplanung bei Fliessbandverarbeitung und Schleifenfaltung.

faltung.

$$\sum_{i:(v_i,v_k)\in E_R} \sum_{p'=0}^{w_i-1} \sum_{p:l_i\leq t-p'+p\cdot P\leq h_i} x_{i,t-p'+p\cdot P} \leq \alpha(v_k) \qquad \forall 1 \leq t \leq P,\ \forall v_k \in V_T \tag{4.6}$$

Der Beweis dieses Theorems stützt sich auf die Ausführungen in Zusammenhang mit Theorem 4.1.

- Zur mathematischen Modellierung des Planungsverfahrens mit Berücksichtigung von funktionaler Fliessbandverarbeitung und Schleifenfaltung wird die in Theorem 4.1 definierte binäre Variable $x_{i,t}$ benutzt.

- Ebenso wie im rein azyklischen Fall muss gewährleistet sein, dass jede Operation genau einmal in jedem Iterationsintervall ausgeführt wird, siehe (4.2). Durch (4.4) wird gewährleistet, dass die Datenabhängigkeiten zwischen den Operationen erhalten bleiben.

- Etwas schwieriger sind nun die Ressourcenbeschränkungen zu modellieren, da sich Operationen über die Grenzen eines Ausführungszyklus erstrecken können. Die Ausführung der Operationen v_i mit dem Index l eines iterativen Algorithmus beginnen an den Kontrollschritten $t_i = \tau(v_i) + l \cdot P$. Wird die Ausführung einer Operation v_i zum Kontrollschritt $\tau(v_i)$ begonnen, belegt sie entsprechend Ressourcen während der Kontrollschritte t mit

$$t = \tau(v_i) + p' - p \cdot P \tag{4.7}$$
$$\forall p' : 0 \leq p' \leq w_i - 1 \tag{4.8}$$
$$\forall p : l_i \leq t - p' + p \cdot P \leq h_i \tag{4.9}$$

  Für $p = 0$ ergibt die Auswertung von (4.8) die Kontrollschritte t mit $\tau(v_i) \leq t \leq \tau(v_i)+w_i-1$; dies entspricht den Ausführungszeitpunkten von v_i im azyklischen Fall. Für $p \neq 0$ sind dies die jeweils um $p \cdot P$ Kontrollschritte verschobenen Ausführungszeitpunkte von Operation v_i. Die Grenzen für p ergeben sich aus der Forderung, dass sich alle Kontrollschritte innerhalb des zulässigen Bereichs für v_i befinden müssen. Die Auswertung der beiden inneren Summen von Ungleichung (4.6) für eine gegebene Operation v_i ergibt, dass die Summe

$$\sum_{p'=0}^{w_i-1} \sum_{p:l_i\leq t-p'+p\cdot P\leq h_i} x_{i,t-p'+p\cdot P}$$

  für alle t, zu denen Operation v_i innerhalb einer Iteration $1 \leq t \leq P$ eine Ressource belegt, den Wert 1 hat, sonst den Wert 0. Wiederum wird durch Summation über alle Operationen v_i, die auf demselben Ressourcentypentypen v_k ausgeführt werden, sichergestellt, dass zu keinem Kontrollschritt mehr als die von diesem Typ zur Verfügung stehenden Instanzen benutzt werden.

Das bis jetzt vorgestellte Verfahren zur Ablaufplanung setzt voraus, dass keine Datenabhängigkeiten zwischen den Iterationen bestehen. In zahlreichen Algorithmen der Signalverarbeitung werden jedoch Daten, die in Iteration l berechnet werden, in einer der

folgenden Iterationen benutzt. Ein bekanntes Beispiel ist ein Algorithmus [1] für ein FIR-Filter:

$$y_t = \sum_{j=0}^{N-1} a_j \cdot u_{t-j}$$

Der Ausgangswert y_t zum Zeitpunkt t wird als Summe der mit a_j gewichteten Eingangswerte u der Zeitpunkte $t \ldots t - N + 1$ berechnet.

Es ist daher sinnvoll, als Erweiterung des bisherigen Modells auch iterative Algorithmen mit Datenabhängigkeiten zwischen den Iterationen zu betrachten. Weiterhin wird zugelassen, dass die von den iterativen Algorithmen implizierten Sequenzgraphen Zyklen aufweisen. Die Berechnung der frühest- bzw. spätestmöglichen Ausführungszeitpunkte der Operationen mit ASAP- und ALAP-Verfahren kann unter Vernachlässigung aller Datenabhängigkeiten mit $d_{ij} \neq 0$ durchgeführt werden.

Theorem 4.3 (Datenabhängigkeiten zwischen Iterationen) *Gegeben seien ein möglicherweise zyklischer Sequenzgraph G_S, ein Ressourcengraph G_R, das Iterationsintervall P, die maximal zulässige Latenz L mit $P < L$ und die mit ASAP- und ALAP-Planungsverfahren berechneten frühest- bzw. spätestmöglichen Ausführungszeitpunkte l_i und h_i der Operationen $v_i \in V_S$. Wird in dem Ungleichungssystem (4.2,4.4,4.6) die Ungleichung (4.4) durch das folgende Ungleichungssystem (4.10) ersetzt, so ist eine zulässige Lösung des resultierenden Ungleichungssystems eine Lösung des Planungsproblems mit Datenabhängigkeiten zwischen Iterationen sowie funktionaler Fiessbandverarbeitung und Schleifenfaltung.*

$$\tau(v_j) - \tau(v_i) \geq w_i - d_{ij} \cdot P \qquad \forall (v_i, v_j) \in E_S \tag{4.10}$$

Zur mathematischen Modellierung des Planungsverfahrens mit Berücksichtigung der Datenabhängigkeiten zwischen Iterationen sowie funktionaler Fliessbandverarbeitung und Schleifenfaltung wird die in Theorem 4.1 definierte binäre Variable $x_{i,t}$ benutzt. Die Argumentation für die Gültigkeit der Ungleichungen (4.2,4.6) bleibt unverändert.

Für die Knoten $v_i, v_j \in V_S$ existiere eine Kante $(v_i, v_j) \in E_S$, der ein Gewicht d_{ij} zugeordnet sei. Die in Iteration l durch die Ausführung von Operation v_i erzeugten Daten werden in Iteration $l + d_{ij}$ von Operation v_j benötigt. Die Ausführung von Operation v_i beginne an Kontrollschritt $t = \tau(v_i)$. Für den frühest möglichen Kontrollschritt $t = \tau(v_j)$, an dem die Ausführung von Operation v_j beginnen kann, gilt also

$$\tau(v_j) + d_{ij} \cdot P \geq \tau(v_i) + w_i.$$

Zur Veranschaulichung der Ergebnisse wollen wir nun ein konkretes Beispiel betrachten.

Beispiel 4.21 *Gegeben sei ein FIR-Filter zweiter Ordnung, der durch den iterativen Algorithmus*

$$y(t) = a_0 x(t) + a_1 x(t-1) + a_2 x(t-2) \qquad \forall t \geq 0 \tag{4.11}$$

gegeben ist. Eine direkte Implementierung dieser Differenzengleichung ist der in Abb. 4.20 dargestellte Signalflussgraph.

Um eine Implementierung mit beschränkten Ressourcen (≤ 3 Multiplizierer, ≤ 2 Addierer) zu erzielen, modellieren wir den Filter mit dem Sequenzgraphen in Abb. 4.21, den man aus Abb. 4.20 erhält durch Auftrennen der Rückkopplung an Verzögerungselementen (gekennzeichnet mit D) und nach Einführen eines Startknotens v_0 und eines Endknotens v_n.

[1] Es handelt sich hier nur um eine kompakte Schreibweise für einen iterativen Algorithmus

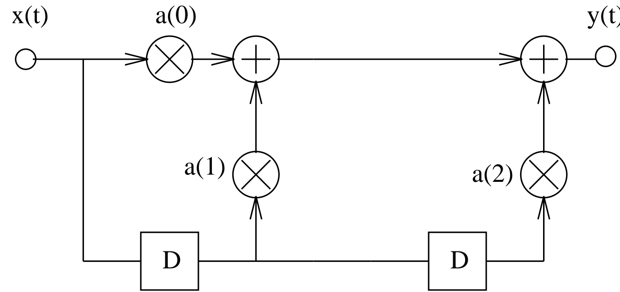


Abbildung 4.20: Signalflussgraph eines FIR-Filters zweiter Ordnung

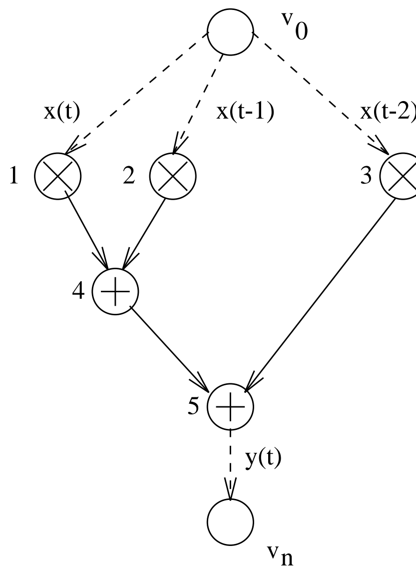


Abbildung 4.21: Sequenzgraph FIR-Filter

Beispiel 4.22 *Der Sequenzgraph in Abb. 4.21 erlaubt die Ablaufplanung des FIR-Filters mit beschränkten Ressourcen.*

Nun wollen wir das Problem der Latenzminimierung mit Ressourcenbeschränkungen mit Hilfe eines ganzzahligen linearen Programms beschreiben.

Beispiel 4.23 *Gegeben sei der Sequenzgraph des FIR-Filters in Abb. 4.21. Wir nehmen an, dass zwei Ressourcen des Typs r_1 (Multiplizierer) ($\alpha(r_1) = 2$) und eine Ressource des Typs r_2 (Addierer) ($\alpha(r_2) = 1$) verfügbar seien. Um die Anzahl der Variablen des ganzzahligen linearen Programms möglichst klein zu halten, berechnen wir für eine gewählte Latenzschranke $\bar{L} = 5$ die frühesten und spätesten Startzeitpunkte von Operationen mit Hilfe der Algorithmen ASAP und ALAP. Das Ergebnis sind Zeitintervalle, die in Abb. 4.22 dargestellt sind.*

Beispiel 4.24 *Unter den Vorgaben in Beispiel 4.23 erhalten wir folgendes System von Beschränkungen für unser ganzzahliges lineares Programm:*

1. Einführung binärer Variablen:

$$\tau(v_1) = 1x_{1,1} + 2x_{1,2}, \qquad x_{1,1} + x_{1,2} = 1$$

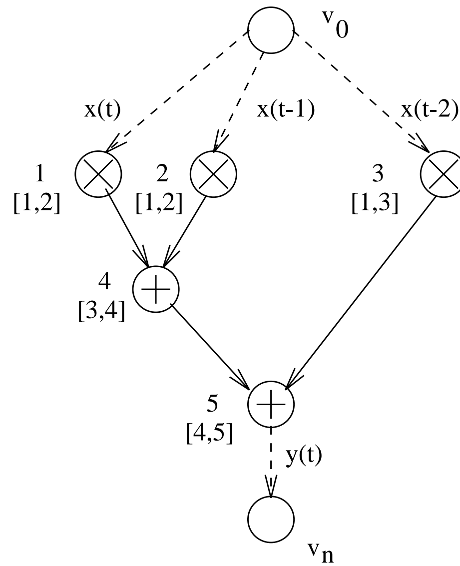


Abbildung 4.22: ASAP und ALAP mit $\bar{L} = 5$ am Beispiel des FIR-Filters

$$\tau(v_2) = 1x_{2,1} + 2x_{2,2}, \qquad x_{2,1} + x_{2,2} = 1$$
$$\tau(v_3) = 1x_{3,1} + 2x_{3,2} + 3x_{3,3}, \qquad x_{3,1} + x_{3,2} + x_{3,2} = 1$$
$$\tau(v_4) = 3x_{4,3} + 4x_{4,4}, \qquad x_{4,3} + x_{4,4} = 1$$
$$\tau(v_5) = 4x_{5,4} + 5x_{5,5}, \qquad x_{5,4} + x_{5,5} = 1$$

2. Datenabhängigkeiten:

$$\tau(v_4) - \tau(v_2) \geq 2$$
$$\tau(v_4) - \tau(v_2) \geq 2$$
$$\tau(v_5) - \tau(v_3) \geq 2$$
$$\tau(v_5) - \tau(v_4) \geq 1$$
$$\tau(v_1), \tau(v_2), \tau(v_3) \geq 1$$

3. Berechnung der Belegungen durch Faltung und Summation der Belegungen:

$$x_{1,1} + x_{2,1} + x_{3,1} \leq 2 \qquad (t = 1)$$
$$x_{1,1} + x_{1,2} + x_{2,1} + x_{2,2} + x_{3,1} + x_{3,2} \leq 2 \qquad (t = 2)$$
$$x_{1,2} + x_{2,2} + x_{3,2} + x_{3,3} \leq 2 \qquad (t = 3)$$
$$x_{3,3} \leq 2 \qquad (t = 4)$$

Eine Lösung des ganzzahligen linearen Programms führt zu der Lösung in Abb. 4.23. Die Latenz beträgt $L = 5$ und entspricht dem Iterationsintervall.

Offensichtlich sind die Ressourcen schlecht ausgenutzt, da aufeinanderfolgende Iterationen nicht überlappen. Eine bessere Formulierung des Algorithmus ist der in Abb. 4.24 dargestellte Abhängigkeitsgraph, der nicht nur eine Iteration des Algorithmus darstellt.

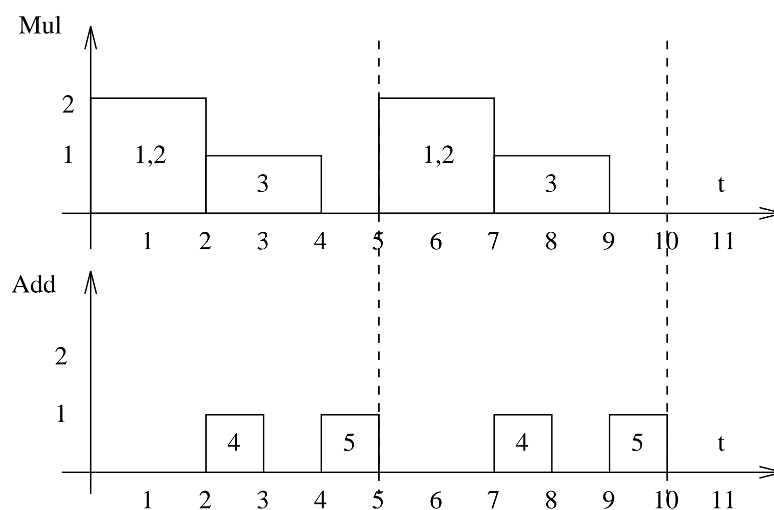


Abbildung 4.23: Ablaufplan FIR-Filter

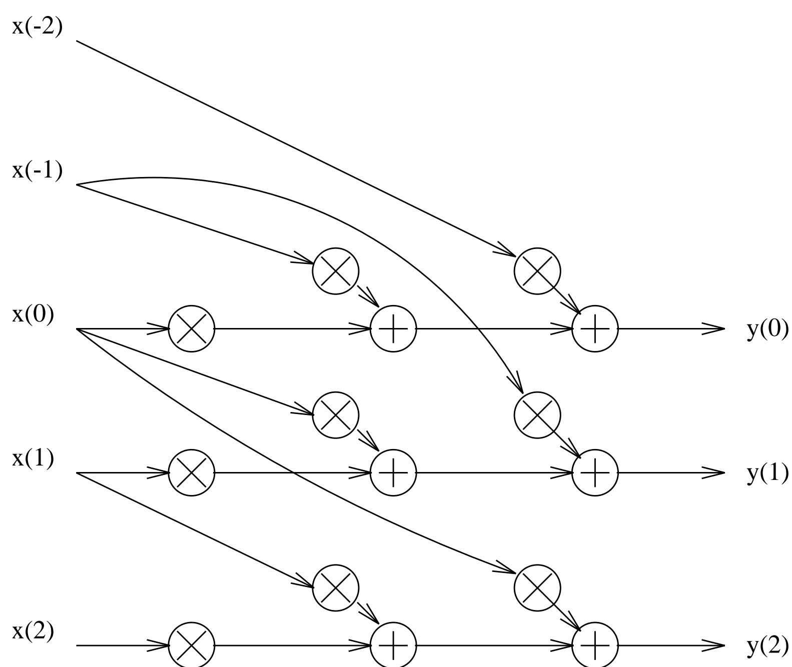


Abbildung 4.24: Abhängigkeitsgraph des FIR-Filters

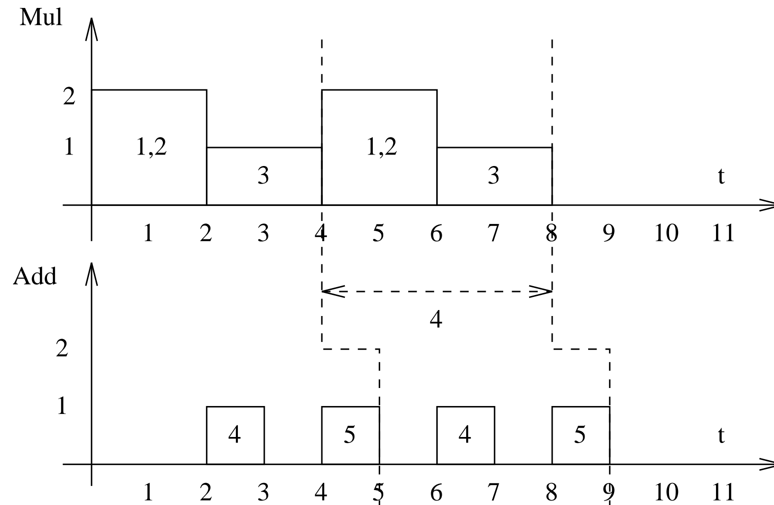


Abbildung 4.25: Ablaufplan FIR-Filter

Beispiel 4.25 *Abb. 4.24 zeigt eine äquivalente Darstellung des Algorithmus in Glg. (4.11), die den Vorteil hat, dass eine bessere Ablaufplanung möglich ist durch gleichzeitiges Planen mehrerer Iterationen. Ein besserer Ablaufplan mit Iterationsintervall* 4 *ist in Abb. 4.25 dargestellt.*

Diesen Vorteil kann man nun auch erzielen, indem man funktionale Fliessbandverarbeitung und Schleifenfaltung erlaubt.

Beispiel 4.26 *Die in Abb. 4.25 dargestellte Lösung kann man auch durch Betrachtung unseres erweiterten Sequenzgraphmodells erzielen. Dieser ist in Abb. 4.26 dargestellt.*

4.2.4 Iterative Verfahren zur Ablaufplanung

Zur Vervollständigung unserer Verfahren möchten wir hinzufügen, dass es auch Verfahren zur iterativen Verbesserung von Abaufplänen gibt. Hierzu gehört beispielsweise Simulated Annealing [32]. Ein anfänglicher Ablaufplan wird iterativ verbessert durch Umplanung einzelner Operationen. Die Umplanungsschritte erfolgen zufallsgesteuert. Schritte, die die aktuelle Latenz verkleinern, werden generell akzeptiert. Um jedoch die Möglichkeit zu vermeiden, dass man ein lokales Minimum erzielt, werden auch Umplanungen mit schlechterer Latenz mit einer Wahrscheinlichkeit akzeptiert. Diese Wahrscheinlichkeit nimmt im Verlaufe des Verfahrens ab. Simulated Annealing ist ein exaktes Verfahren. Es zeichnet sich aber durch eine im Allgemeinen hohe Laufzeit aus. Ein anderes Verfahren zur iterativen Verbesserung von Ablaufplänen sind Genetische Algorithmen [33, 34]. Aus Zeitgründen können wir hier nicht weiter auf diese Verfahren eingehen.

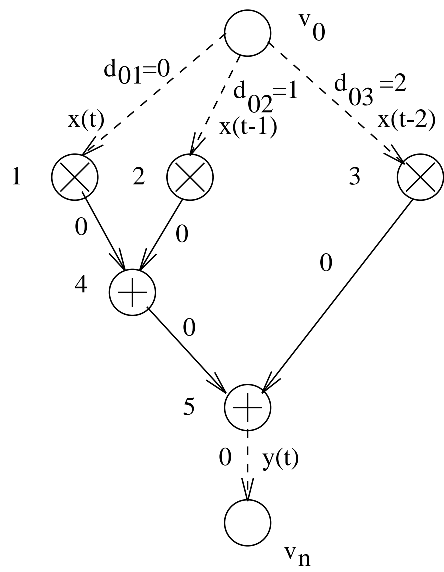


Abbildung 4.26: Erweiterter Sequenzgraph FIR-Filter

Kapitel 5

Beispiele zur Architektursynthese

5.1 Architekturbewertungen

In den vergangenen Kapiteln wurden graphische und algorithmische Modelle zur Beschreibung von Schaltungen vorgestellt. Es wurden auf ganzzahliger linearer Programmierung basierende Planungsverfahren entwickelt, mit denen die Synthese solcher Systeme durchgeführt werden kann. Es ist jetzt notwendig, die Konzepte einzuführen, mit denen es möglich ist, Architekturen auf ihre Leistungsfähigkeit hin zu untersuchen und entsprechend zu bewerten.

Architekturen können mit unterschiedlichen Leistungsmassstäben bewertet werden. Ein in vielen Optimierungsproblemen auftretendes Dilemma besteht darin, dass die Leistungsverbesserung einer Architektur hinsichtlich eines Kriteriums κ_1 die Verschlechterung hinsichtlich eines anderen Kriteriums κ_2 bedeutet. Ein Beispiel ist die Verbesserung der Durchsatzrate einer CMOS-Schaltung durch Erhöhung der Taktfrequenz, was jedoch eine Erhöhung des dynamischen Leistungsverbrauchs bewirkt.

5.1.1 Verarbeitungsgeschwindigkeit

Zwei wichtige Kriterien zur Beurteilung der Verarbeitungsgeschwindigkeit einer Schaltung sind das *Iterationsintervall* P und die *Latenz* L. Das Iterationsintervall P ist ein direktes Mass für die Geschwindigkeit, mit der eine Schaltung einen Strom von Eingangsdaten verarbeiten kann. Die Latenz L beschreibt die Geschwindigkeit, mit der ein Satz von Eingabedaten vollständig bearbeitet wird.

Zum Kontrollschritt $t = t_i$ beginnt die Verarbeitung der zur i-ten Iteration gehörigen Menge von Eingangsdaten $\delta(i)$. Die Verarbeitung der nächsten Menge von Eingangsdaten $\delta(i+1)$ beginnt entsprechend zum Zeitpunkt $t_i + P$. Zum Zeitpunkt $t_i + L$ stehen alle $\delta(i)$ entsprechenden Ausgangsdaten zur Verfügung.

5.1.2 Auslastung von Ressourcen

Zur Beurteilung der Leistungsfähigkeit einer Architektur gehört neben der Beurteilung der Verarbeitungsgeschwindigkeit auch die Frage, wie gut die eingesetzten Ressourcen einer Implementierung ausgenutzt werden.

Definition 5.1 (Effizienz eines Ressourcetypen) *Die Effizienz $E(r_k)$ eines Ressourcetyps r_k ist die mittlere Zahl belegter Ressourcen durch die Zahl allozierter Ressourcen $\alpha(r_k)$.*

Entsprechend ist auch die Effizienz einer einzelnen Ressource definiert. So ist zum Beispiel die Effizienz $E(p)$ eines Prozessors p der Bruchteil des Iterationsintervalls, während dem der Prozessor p Daten verarbeitet. Die Effizienz ist mithin ein wichtiges Entscheidungskriterium für die Zuordnung von Operationen zu Prozessoren. Es kann etwa sinnvoll sein einen schlecht ausgelasteten Spezialprozessor, der viel Fläche benötigt, durch einen besser ausgelasteten Universalprozessor zu ersetzen, falls ein damit verbundener Verlust an Verarbeitungsgeschwindigkeit tolerierbar ist.

5.1.3 Auslastung von Architekturen

Der Lastausgleichsfaktor Z ist ein Mass zur Bewertung der Auslastungen der einzelnen Ressourcen einer Architektur.

Definition 5.2 (Lastausgleichsfaktor eines Ressourcetypen) *Gegeben seien die entsprechend Definition 5.1 definierten Effizienzen einer Ressource. Dann gilt für den Lastausgleichsfaktor eines Ressourcetypen r_k:*

$$Z(r_k) := \frac{\min\{E(p): \text{ alle Ressourcen } p \text{ eines Ressourcetypen } r_k\}}{\max\{E(p): \text{ alle Ressourcen } p \text{ eines Ressourcetypen } r_k\}} \tag{5.1}$$

Der Lastausgleichsfaktor Z gibt daher das Verhältnis von minimaler zu maximaler Auslastung der Ressourcen eines Ressourcetypen an. Falls zum Beispiel alle Prozessoren eines Prozessortyps r gleichmässig ausgelastet sind, ist mit $Z(r) = 1$ der Lastausgleichsfaktor maximal. Die Effizienz $E(r)$ dieses Prozessortyps r ist dann durch die Effizienz der einzelnen Prozessoren p gegeben

$$E(r) = E(p) \qquad \forall \quad \text{Prozessoren } p \text{ des Typs } r \tag{5.2}$$

In allen anderen Fällen ist die Effizienz $E(r)$ wie folgt begrenzt:

$$E(p) \cdot Z \leq E(r) \leq E(p)/Z \tag{5.3}$$

Bedingung zur guten Auslastung einer Schaltung sind also sowohl gleichmässige als auch hohe Auslastung der Prozessoren.

5.1.4 Fliessbandtiefe

Wie im vergangenen Kapitel dargestellt wurde, kann durch funktionale Fliessbandverarbeitung die Durchsatzrate einer Schaltung gesteigert werden. Das Verhältnis aus Latenz L und Iterationsintervall P wird als (funktionaler) Fliessbandfaktor F bezeichnet.

Definition 5.3 (Fliessbandfaktor) *Der Fliessbandfaktor F wird als Quotient aus Latenz L und Iterationsintervall P berechnet:*

$$F = \frac{L}{P} \tag{5.4}$$

Es gibt dementsprechend Zeitschritte, an denen Berechnungen und Speicheroperationen auf Daten in $\lceil F \rceil$ verschiedenen Iterationen durchgeführt werden. Werden nun die Daten einer Operation v_i mit $(v_i, v_j) \in E_S$ über mehrere Iterationen gespeichert bis sie von v_j benutzt werden, so muss der Speicher entsprechend vergrössert werden. In einer rein seriellen Anwendung mit $L = P$ ist der Speicherbedarf nach unten durch die Operation mit dem grössten Bedarf begrenzt. Der Fliessbandfaktor F hat mithin Einfluss auf die Grösse des notwendigen Speichers und auf die Komplexität des Kontrollbausteins für den Speicher.

5.2 Architekturentwurf für ein digitales Filter

In diesem Abschnitt wird die Anwendung der Verfahren auf den Entwurf eines digitalen elliptischen Filters 5. Ordnung nach [37] gezeigt. Der Sequenzgraph des Filters ist in Abbildung 5.1 dargestellt. Die Knoten A und O stellen Ein- und Ausgang der Schaltung dar, während die Knoten $1, \ldots, 27$ Additionen und die Knoten $a, \ldots, h$ Multiplikationen repräsentieren. Die dicker dargestellten Querstriche in den Kanten stellen Indexverschiebungen um eine Iteration dar.

Zur Ausführung der Operationen stehen Addierer und Multiplizierer zur Verfügung, deren Rechenzeiten eine beziehungsweise zwei Taktzyklen betragen. Die folgenden beiden Konfigurationen werden untersucht:

- Für die ersten Untersuchungen wurde nur der azyklische Abhängigkeitsgraph betrachtet; er geht aus dem Abhängigkeitsgraph nach Abbildung 5.1 hervor, indem die Kanten, die eine Indexverschiebung enthalten, entfernt wurden und anschliessend die Knoten $3, 23, 24, 14, 26, 27, 21$ mit dem Schaltungsausgang und die Knoten $2, 22, 23, 6, 10, 17, 25, 26, 19, 20$ mit dem Eingang verbunden wurden. Für diese Konfiguration wurden Ablaufpläne ermittelt, wobei für ein vorgegeben Iterationsintervall mit der zugehörigen minimalen Latenz der Bedarf an arithmetischen Einheiten optimiert wurde. Demzufolge fehlen also die Datenabhängigkeiten *zwischen* Iterationen.

- In einer weiteren Untersuchungsreihe wurden Ablaufpläne für den zyklischen Abhängigkeitsgraphen nach Abb. 5.1 berechnet. Auch hier wurden für vorgegebene Werte des Iterationsintervalls der Verbrauch an Ressourcen unter Annahme minimaler Latenz optimiert.

In Tabelle 5.1 sind die Ergebnisse zusammengefasst. Die Spalten P und L stellen die Werte des Iterationsintervalls P und der zugehörigen minimalen Latenz L dar, für die Planungen der Filters berechnet wurden. Die Spalten $+$ und $\times$ geben die mindestens notwendige Anzahl von Addierern und Multiplizierern zur Realisierung der Ablaufpläne an. Die Geschwindigkeit, mit der die Lösungen ermittelt wurden, wird in der Spalte I angegeben; es ist die Anzahl von Iterationen des Simplex Algorithmus, die das verwendete Programmpaket durchführte. Die in Tabelle 5.1 dargestellten Ergebnisse zeigen, dass bei Erhöhung

P	L	zyklisch			azyklisch		
		$+$	$\times$	I	$+$	$\times$	I
16	19	3	2	3440	2	2	288
17	19	2	2	638	2	2	481
18	19	2	2	211	2	2	153
19	19	2	2	294	2	2	306
20	20	2	2	322	2	2	282
21	21	2	1	349	2	1	293

Tabelle 5.1: Ergebnisse für das elliptische Filter

des Iterationsintervalls der Hardwareverbrauch gesenkt werden kann. Man kann also auf Kosten der Durchsatzrate den Bedarf an Hardware reduzieren. Bei konstant gehaltenem Iterationsintervall kann man durch Erhöhen der Latenz ebenfalls den Hardwareverbrauch

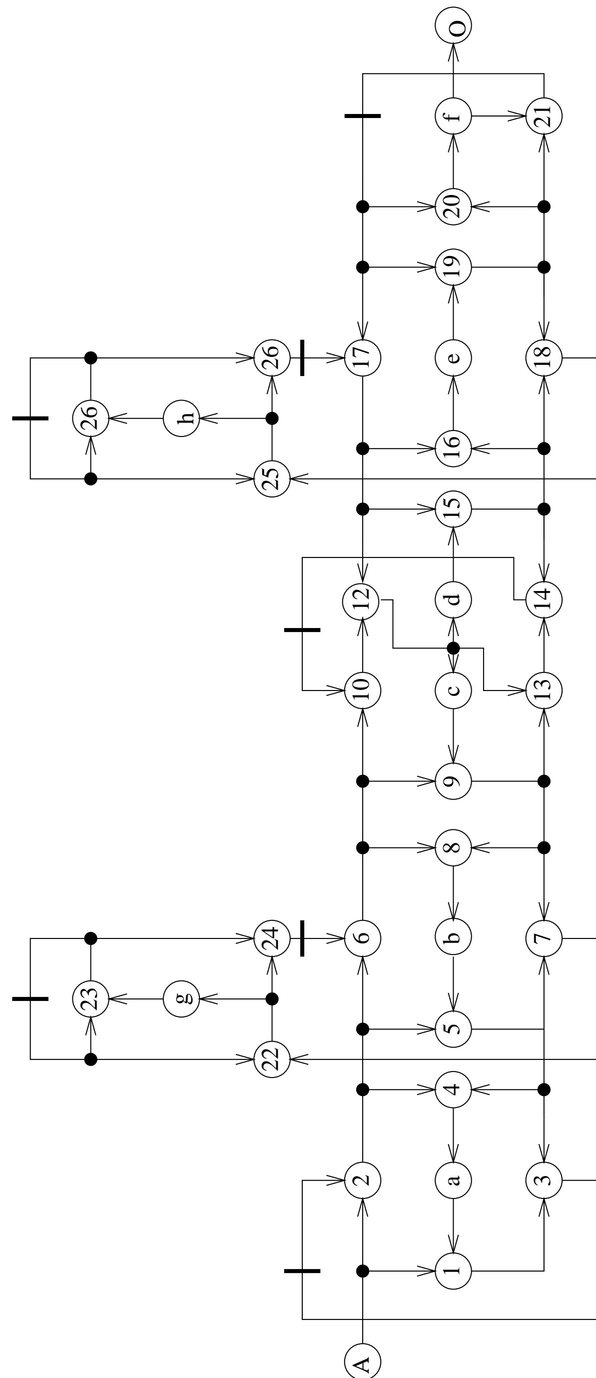


Abbildung 5.1: Digitales elliptisches Filter fünfter Ordnung

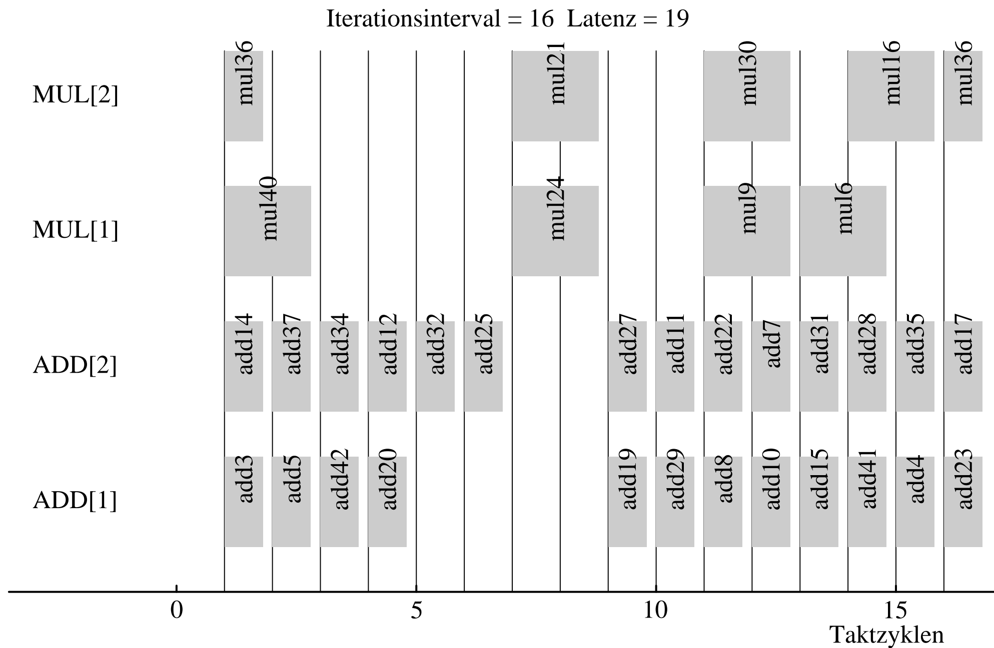


Abbildung 5.2: Ablaufplan eines digitalen elliptischen Filters

senken. Im Fall des azyklischen Abhängigkeitsgraphen wurden für unterschiedliche Werte des Iterationsintervalls Ablaufpläne mit grösserer Latenz als in Tabelle 5.1 dargestellt berechnet. Die Resultate sind in Tabelle 5.2 angegeben. Bei einem Iterationsintervall von $P = 17$ Kontrollschritten kann bei einer Erhöhung der zulässigen Latenz L von 19 auf 20 Kontrollschritte eine Realisierung gefunden werden, die nur einen Multiplizierer benötigt. In Abbildung 5.2 ist der Vollständigkeit halber noch ein Ablaufplan für das digitale Filter

P	L	$+$	$\times$
16	19	2	2
17	20	2	1
18	21	2	1
19	22	2	1

Tabelle 5.2: Ergebnisse für das elliptische Filter, azyklischer Sequenzgraph

dargestellt; es ist ein Ergebnis für den azyklischen Fall bei einem Iterationsintervall von $P = 16$ Kontrollschritten und einer Latenz von $L = 19$ Kontrollschritten, wobei die Anzahl benötigtet Addierer und Multiplizierer minimiert wurde.

5.3 Speicher

Als Voraussetzung für das folgende Beispiel wird noch kurz auf die Modellierung des Speicherbedarfs einer Schaltungsarchitektur eingegangen. Eine zentrale Rolle beim Entwurf von Schaltungen auf Architekturebene ist die korrekte Modellierung, Synthese und Optimierung des Speicherbedarfs. Die folgenden Punkte motivieren den Bedarf an spezi-

ellen Entwurfsmöglichkeiten für Speicher und grenzen die Anforderungen an Entwurfssysteme der Systemebene gegenüber denen anderer Entwurfsebenen ab.

- Beim Entwurf von Datenpfadsystemen auf algorithmischer Ebene wird der Speicher als Register in den Datenpfaden modelliert ([29, 31]). Beim Entwurf von Multiprozessorsystemen auf der Systemebene wird neben den Registern der einzelnen Prozessoren schneller Speicher benötigt, in dem Berechnungsergebnisse abgelegt werden können. Dies kann zur Zwischenspeicherung während der Bearbeitung einer Operation notwendig sein oder um Ergebnisdaten abzulegen, die später als Eingangsdaten von anderen Operationen benötigt werden. Derartige „Cache"-Speicher werden als Speicherbänke auf dem Prozessor selbst oder als externe Module realisiert. Neben dem Speicher müssen geeignete Verbindungsstrukturen entworfen werden, mit deren Hilfe der Datenaustausch abgewickelt wird.

- Eine grundlegende Grösse bei der Betrachtung des Speicherbedarfs ist die Fläche, da sie einen nicht zu vernachlässigenden Anteil an der gesamten Schaltungsfläche einnimmt. Der Optimierung des Speicherbedarfs kommt daher eine wichtige Rolle bei der Schaltungssynthese zu.

- Speichermodule können für einzelne Prozessoren reserviert sein oder mehrere Prozessoren können sich ein Speichemodul teilen. Das Teilen von Speichermodulen (*shared memory*) bietet den Vorteil besserer Ausnutzung und des Wegfalls von Kopieroperationen zwischen Speichermodulen, wenn Daten zwischen Prozessoren ausgetauscht werden. In diesem Fall müssen allerdings Zugriffskonflikte durch gemeinsamen Zugriff von Prozessoren verhindert werden.

- Eine weitere wichtige Anforderung an ein Synthesewerkzeug ist neben der reinen Optimierung der Grösse des Speichers die Möglichkeit, Operationen bestimmten Speicherbereichen zuweisen zu können.

- Da die Daten der Operationen auf Speicherbänken abgelegt werden, ist die genaue Berechnung der Lebenszeit von Variablen von besonderer Wichtigkeit.

5.3.1 Lebenszeit von Variablen

Die exakte Berechnung der Lebenszeit von Variablen ist von essentieller Bedeutung für die korrekte Berechnung des Speicherbedarfs. Für jede Kante $(v_i, v_j) \in E_S$ kann ein Transfer von Daten von der generierenden Operation v_i zu der benutzenden Operation v_j stattfinden. Die von v_i erzeugten Daten müssen im Speicher verweilen, bis sie von v_j benutzt werden. Im hier benutzten Modell für den Transfer von Daten werden die folgenden Annahmen getroffen:

- Gültige Ausgangsdaten einer Operation v_i werden vom Kontrollschritt $t = \tau(v_i)$ an erzeugt, an dem die Ausführung von v_i beginnt.

- Alle Operationen v_j, die direkt datenabhängig von v_i sind, benutzen den gleichen Satz von Ausgangsdaten.

- Die von v_i erzeugten Ausgangsdaten verweilen bis zum Kontrollschritt $t = \tau(v_j)$ im Speicher, an dem die Ausführung der letzten Operation v_j beginnt, die direkt von v_i datenabhängig ist.

- Die Grösse des für eine Operation reservierten Speicherbereichs ändert sich nicht während der Kontrollschritte, an denen auf den Bereich zugegriffen wird.
- Speicherbereiche, die nicht mehr benötigt werden, um Daten zu speichern, können anschliessend von anderen Operationen benutzt werden.

Definition 5.4 (Lebenszeit von Variablen) *Gegeben sei ein Sequenzgraph G_S. Es seien $\tau(v_i)$ und $\tau(v_j)$ die Kontrollschritte, an denen die Ausführung einer Operation v_i beziehungsweise einer von v_i datenabhängigen Operation v_j mit $(v_i, v_j) \in E_S$ beginnen. Die Lebenszeit der Variablen, die von der Operation v_i generiert werden, ist die Anzahl von Kontrollschritten $\tau(v_k) - \tau(v_i)$, für die gilt: $\tau(v_k) = \max\{\tau(v_j) \ : \ (v_i, v_j) \in E_S\}$*

Die Lebenszeit der von einer Operation v_i erzeugten Variablen ist daher das Intervall zwischen dem Kontrollschritt $\tau(v_i)$, an dem die Ausführung von v_i beginnt, und dem letzten Kontrollschritt $\tau(v_k)$, an dem die Ausführung einer direkt von v_i datenabhängigen Operation v_j beginnt. Es wird demnach von der Annahme ausgegangen, dass mit dem Beginn der Ausführung einer Operation gültige Daten für nachfolgende Operationen zur Verfügung stehen.

Es ist hier lediglich angemerkt, dass sich auch die Lebenszeiten von Variablen in Form linearer Gleichungen und Ungleichungen darstellen lassen. Hierzu werden neue Variablen $z_{i,t}$ bestimmt mit

$$z_{i,t} = \begin{cases} 1 & : \text{ während der Lebenszeit der von } v_i \text{ erzeugten Variablen} \\ 0 & : \text{ sonst} \end{cases}$$

5.3.2 Modellierung des Speicherbedarfs

Nachdem im letzten Abschnitt die Lebenszeit von Variablen definiert wurde und Methoden zur ihrer exakten Berechnung angedeutet wurden, kann jetzt der Speicherbedarf von Operationen modelliert und berechnet werden.

Der Speicherbedarf von Operationen wird nach *Arbeitsspeicher* und *Transferspeicher* unterschieden. Diese rein konzeptionelle Unterteilung betrifft nur die Modellbildung. Der errechnete Bedarf einer Operation an Arbeitsspeicher beziehungsweise Transferspeicher kann auf dem gleichen Speicherbaustein zur Verfügung gestellt werden. Ein geeignetes Werkzeug zur Modellierung des Speicherbedarfs von Operationen bezüglich Transfer- und Arbeitsspeicher sind wieder die zur Modellierung des Prozessorbedarfs verwendeten Ressourcegraphen. Die Knoten stellen nun die Operationen des Sequenzgraphen sowie Speichermodule dar, die Kanten ordnen den Operationen die Speichermodule zu. Die Kantengewichte geben die Grösse der von den Operation erzeugten Datenmenge an, die dann auf dem zugeordneten Speichermodul abgelegt wird. Die Grösse des auf einem Modul zur Verfügung stehenden Speicherbereichs (Allokation) wird den Knoten der Speichermodule zugeordnet.

Die Konzepte von Arbeits- und Transferspeicher werden nun näher erläutert.

Transferspeicher

Die Ergebnisdaten einer Operation v_i müssen allen von ihr direkt datenabhängigen Operationen v_j mit $(v_i, v_j) \in E_S$ zur Verfügung stehen. Es wird davon ausgegangen, dass gültige Ausgangsdaten vom Beginn der Ausführung einer Operation v_i an produziert werden; sie

werden solange im Speicher gehalten, bis die Ausführung der letzten Operation v_j, die direkt von v_i abhängt, beginnt. Die Definition der Lebenszeiten von Variablen impliziert direkt, dass der Transferspeicher mit Hilfe der Variablen $z_{i,t}$ mit Hilfe linearer Gleichungen berechnet werden kann.

Arbeitsspeicher

Während der Ausführung einer Operation werden nicht nur Ergebnisdaten für nachfolgende Operationen erzeugt; ebenso werden Daten und Zwischenresultate, die später in der Berechnung benötigt werden, gespeichert und gelesen. Der Speicherbereich, der für diesen Zweck reserviert ist, wird Arbeitsspeicher genannt. Operationen benötigen Arbeitsspeicher nur während ihrer Ausführungszeit; es liegt daher nahe, dass der Bedarf an Arbeitsspeicher auf gleiche Weise wie die Belegung einer Prozessorressource abgleitet werden kann.

Zugriffskonflikte

Beim konkurrierenden Zugriff mehrerer Prozessoren auf gemeinsame Speichermodule kann es zu Zugriffskonflikten kommen. Zugriffskonflikte auf Speichermodule lassen sich durch Verwendung von n-Tor Speicher vermeiden, wenn n genügend gross ist. Der Aufwand zur Realisierung und Steuerung von Speicher mit vielen Toren, auf die gleichzeitig gelesen und geschrieben wird, ist jedoch sehr gross. Es ist daher vernünftig, anzunehmen, dass die Tore von Speicher als eigene Ressource anzusehen sind; Zugriffskonflikte können dann vermieden werden, wenn zu keinem Kontrollschritt mehr als die jeweils zur Verfügung stehende Anzahl von Toren für Schreib- und Lesezugriffe benutzt wird. Die geeignete Modellierung ist wieder ein Ressourcegraph. Hierbei modellieren die Knoten die Operationen sowie die Tore der Speichermodule. Eine Kante stellt den Zugriff einer Operation auf das entsprechende Speichertor dar. Die auf einem Speichermodul zur Verfügung stehende Anzahl von Toren ist den Tor-Knoten zugeordnet.

5.4 Anwendung der Verfahren auf einen Videokodierer

Die im vergangenen Abschnitt dargestellten Verfahren und Methoden werden nun benutzt, um verschiedene Möglichkeiten der Implementierung eines hybriden Videokodierers nach CCITT Empfehlung H.261 zu entwerfen und zu vergleichen. Die zugrundegelegten Architekturparameter wurden [38] entnommen. Der Vollständigkeit halber soll hier nochmals der Algorithmus erläutert werden.

Abb. 5.3 zeigt das vereinfachte Blockschaltbild der Kodierschleife. Der Bildaufbau des in Standard CCITT H.261 unterstützten *Common Intermediate Format* ist in Abbildung 5.4 dargestellt. In Abbildung 5.5 ist die Anordnung der Informationen für Luminanz Y und Chrominanz C_r und C_b dargestellt. Das grundlegende Datenformat in der Kodierschleife sind die *Makroblöcke*. Jeder Makroblock besteht aus einem Block von 16×16 Bildpunkten, der die Luminanzinformation enthält sowie einem 8×8 Block, der die Chrominanzinformationen enthält.

Die wichtigsten Funktionsblöcke der Kodierschleife werden kurz beschrieben, um ihre Implementierungsmöglichkeiten in Hardware besser verstehen zu können ([39]). Die Schlüsseltechniken eines Videokodierers nach H.261 sind diskrete Cosinustransformation, Bewegungsschätzung (*motion estimation*) durch Blockvergleich (*block matching*) und

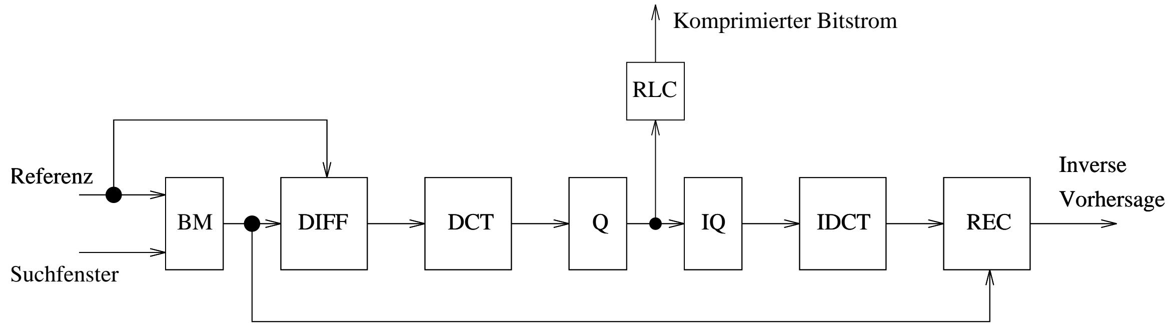


Abbildung 5.3: Kodierschleife nach CCITT H.261

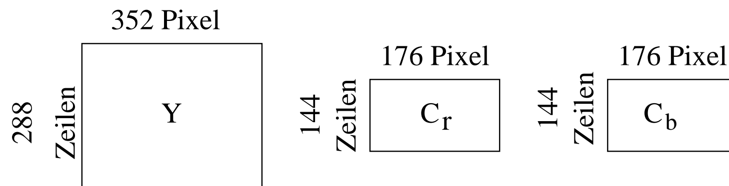


Abbildung 5.4: Bildaufbau des Common Intermediate Format für Video Quellen nach H.261

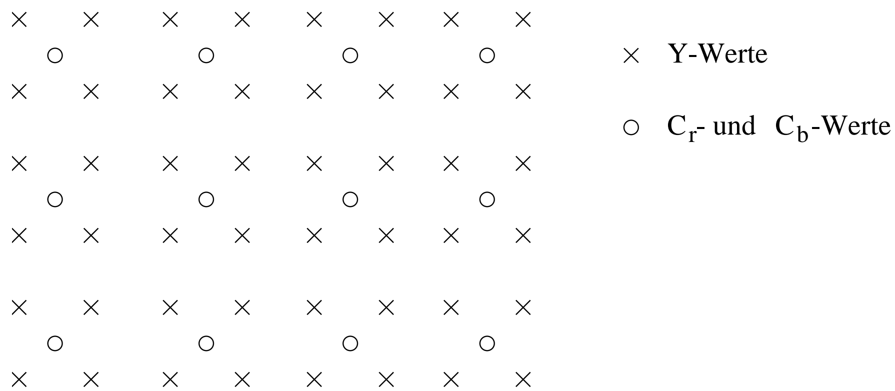


Abbildung 5.5: Anordnung der Luminanz- und Chrominanzinformation

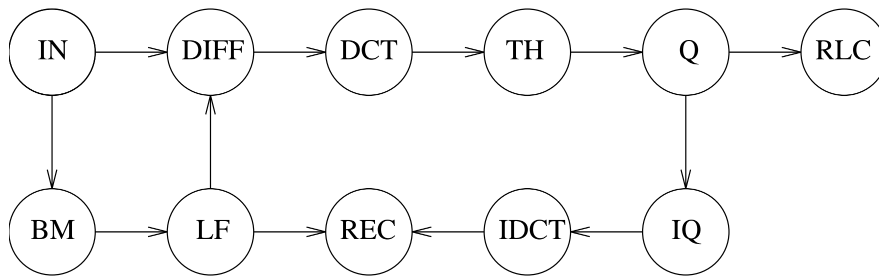


Abbildung 5.6: Abhängigkeitsgraph eines einfachen Videokodierers

zweidimensionale variable Lauflängenkodierung (*run length coding*). Mit diskreter Cosinustransformation (DCT), Quantisierung (Q) und Lauflängenkodierung (RLC) werden die Daten *innerhalb* eines Makroblocks komprimiert, durch die Bewegungsschätzung (ME) werden die Daten *zwischen* zeitlich aufeinanderfolgenden Makroblöcken komprimiert. Zur Bewegungsschätzung wird ein zu übertragender Makroblock mit den Makroblöcken eines Suchfensters verglichen und nur der Verschiebungsvektor übertragen, für den ein Ähnlichkeitskriterium zwischen Makroblöcken maximal wird. In Abbildung 5.3 wird der zu übertragende Makroblock als Referenz bezeichnet. Der in dieser Implementierung eingesetzte Algorithmus für den Blockvergleich besteht im wesentlichen aus der Berechnung von Differenzen und Absolutwerten; der Einsatz eines dedizierten Prozessors liegt also nah. Im Gegensatz zu den anderen Operationen wird die Bewegungsschätzung nur auf der Luminanzinformationen ausgeführt. Mit den inversen Operation IDCT und IQ sowie der Rekonstruktion REC wird der Bewegungsvektor wieder hergestellt. Zwischen den Operationen DCT und IDCT wird gegenüber den restlichen Operationen der Kodierschleife mit doppelter Genauigkeit gerechnet.

5.4.1 Eine einfache Zielarchitektur

Die ersten Untersuchungen wurden an einem einfachen Grundalgorithmus und einer einfachen Zielarchitektur durchgeführt. In Abbildung 5.6 ist der zugrunde liegende Datenabhängigkeitsgraph G_D dargestellt; er lässt sich direkt aus dem Blockschaltbild des Videokodierers nach Abbildung 5.3 herleiten. Gegenüber Abbildung 5.3 wurde die Bewegungsschätzung (ME) in Blockvergleich (BM) und Schleifenfilterung LF (Loop Filter) verfeinert. Weiterhin wurde die Quantisierung Q in die eigentliche Quantisierung und die variable Schwellwertbildung TH (Threshold) unterteilt. Die Zielarchitektur, auf die der Abhängigkeitsgraph nach Abbildung 5.6 abgebildet werden soll, ist in Bild 5.7 dargestellt.

Die Architektur enthält die drei Prozessoren BMM, DCTM und PUM. BMM und DCTM sind Spezialprozessoren, die zur Ausführung des Blockvergleichs BM beziehungsweise der diskreten Cosinustransformationen DCT und IDCT vorgesehen sind. Die restlichen Operationen werden auf dem allgemeinen Prozessor PUM abgearbeitet. Die Bewegungsschätzung wurde in Blockvergleich BM und Schleifenfilter LF aufgespalten; da nur BM auf BMM ausgeführt wird, reduziert sich auf diesem Prozessor die Rechenzeit. Alle Prozessoren sind mit einem gemeinsamen Zwischenspeicher verbunden, der die während einer Iteration benötigten Daten speichert. Der Zwischenspeicher seinerseits ist über einen Bus mit dem Bildspeicher verbunden, aus dem über die IN-Operation die Daten für Such- und Referenzgebiet entnommen werden.

In Tabelle 5.3 sind die Operationen v_i mit den entsprechenden Ausführungszeiten w_i

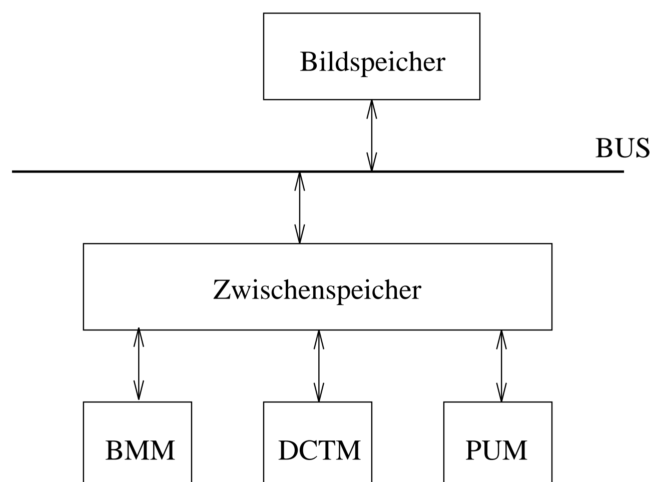


Abbildung 5.7: Zielarchitektur eines einfachen Videokodierers

auf den ihnen zugeordneten Prozessoren sowie dem Bedarf an Transferspeicher μ_i und Arbeitsspeicher ν_i aufgeführt. Die Ausführungszeiten sind in Vielfachen von dimensionslosen Kontrollschritten angegeben, die Einheiten für den Speicherbedarf sind 8×8 Pixelblöcke. Der Speicherbedarf von BM resultiert aus der Verarbeitung von 10 Blöcken des Suchfensters, die die Luminanzinformation enthalten. Da zwischen DCT und IDCT die Daten mit doppelter Rechengenauigkeit verarbeitet werden, ist dort der Speicherbedarf entsprechend doppelt so gross wie bei den sonstigen Operationen. Die Aufspaltung des Speichers in Arbeits- und Transferspeicher lässt eine genaue Modellierung des Speicherbedarfs von Operationen zu. Für DCT beträgt der Bedarf an Arbeitsspeicher 6 Einheiten, während zur nachfolgenden Operation 12 Einheiten transferiert werden; IDCT arbeitet intern auf 12 Einheiten und überträgt nur 6 Einheiten zu REC.

v_i	w_i	μ_i	ν_i	Prozessor
IN	0	0	0	-
BM	22	40	6	BMM
LF	9	6	6	PUM
DIF	1	6	6	PUM
DCT	8	6	12	DCTM
Q	1	12	12	PUM
IQ	1	12	12	PUM
IDCT	8	12	6	DCTM
REC	1	6	6	PUM
TH	8	12	12	PUM
RLC	8	12	0	PUM

Tabelle 5.3: Ausführungszeiten und Speicherbedarf im einfachen Architekturmodell

Die Architektursynthese wurde von einem Planungsverfahren für funktionale Fliessbandverarbeitung mit Schleifenfaltung und Speicherberechnung durchgeführt. Modulauswahl wurde nicht betrachtet. Bei dem minimal möglichen Iterationsintervall von $P = 29$ Kon-

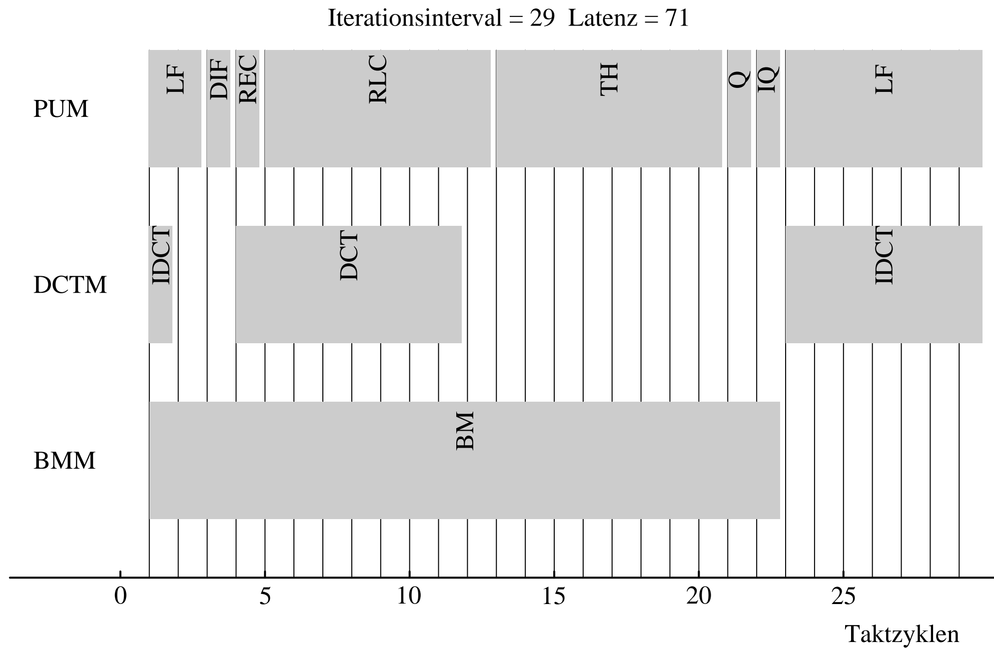


Abbildung 5.8: Abfolge der Operationen

trollschritten und einer zugehörigen minimalen Latenz von $L = 71$ Kontrollschritten wurde eine Planung der Operationen ermittelt, wobei zusätzlich der Speicherbedarf minimiert wurde. Die Abbildungen 5.8 und 5.9 stellen die Abarbeitung der Operationen auf den entsprechenden Prozessoren aus unterschiedlichen Blickwinkeln dar.

In Abb. 5.8 werden funktionale Fliessbandverarbeitung und Schleifenfaltung illustriert: Aus dem Abhängigkeitsgraph nach Abbildung 5.6 geht hervor, dass LF vor IDCT ausgeführt werden muss, wenn diese Operationen zur selben Iteration gehören. Da nach Abbildung 5.8 die Operationen LF und IDCT gleichzeitig auf unterschiedlichen Prozessoren ausgeführt werden, müssen diese Operationen zu verschiedenen Iterationen gehören. Der Ablaufplan realisiert weiter Schleifenfaltung, wie aus der Tatsache hervorgeht, dass Beginn und Ende der Ausführung von LF und IDCT in aufeinanderfolgenden Iterationen liegen. In Abbildung 5.9 ist hingegen die Abfolge der Operationen während der Periode einer Latenz dargestellt. Es wird ersichtlich, dass die durch den Abhängigkeitsgraphen 5.6 implizierten Datenabhängigkeiten eingehalten werden. Die wichtigsten der in Abschnitt 5.1 definierten Leistungsmerkmale dieser einfachen Architektur sind in der folgenden Tabelle 5.4 zusammengestellt.

Im hier betrachteten einfachen Architekturmodell greifen alle Verarbeitungselemente auf das gleiche Speichermodul zu. Der Beitrag von LF zum Gesamtspeicherbedarf ist exemplarisch in Abbildung 5.10 illustriert.

Die Ausführung von Operation LF beginnt an Kontrollschritt $t = 23$. Von diesem Kontrollschritt an muss Arbeitsspeicher in der Grösse von 6 Einheiten zur Ablage interner Daten reserviert werden. Gleichzeitig wird begonnen, Transferspeicher ebenfalls in Grösse von 6 Einheiten für den Datentransport zu nachfolgenden Operationen zu reservieren. Die Ausführung von LF dauert 9 Kontrollschritte; von Kontrollschritt $t = 23$ bis $t = 31$

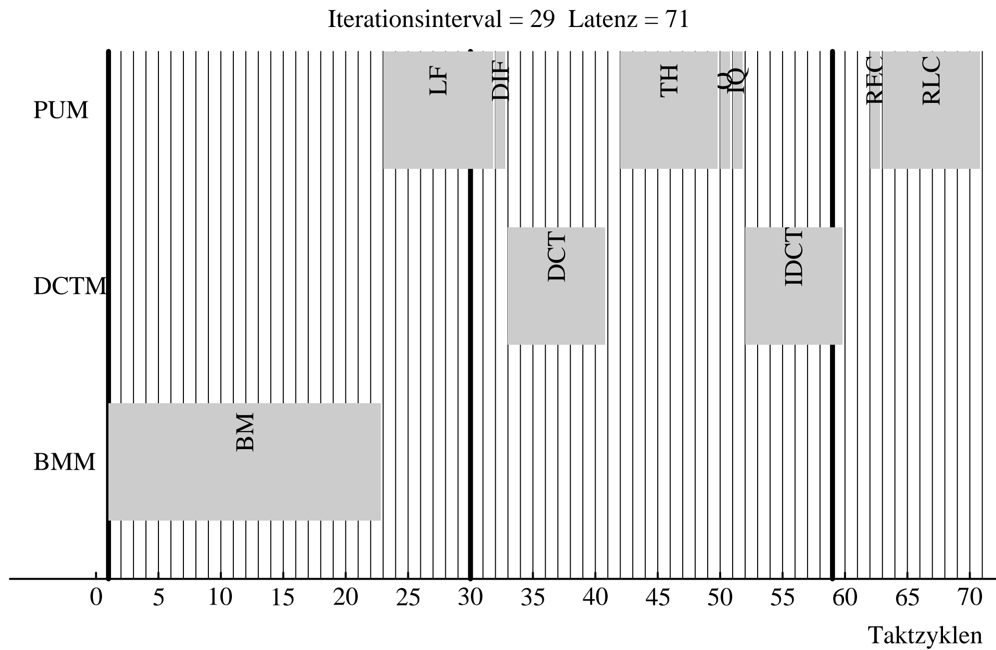


Abbildung 5.9: Entfaltete Abfolge der Operationen

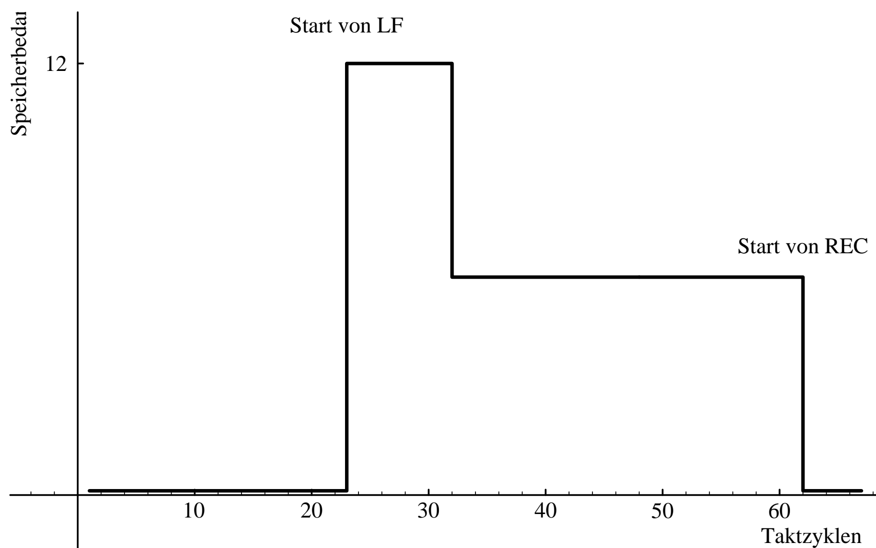


Abbildung 5.10: Speicherbedarf der Operation LF

Merkmal	Wert
Iterationintervall P	29
Latenz L	71
Effizienz $E(BM)$	0.76
Effizienz $E(DCTM)$	0.27
Effizienz $E(PUM)$	1
Lastausgleichsfaktor Z	0.27
Fliessbandfaktor F	2.5

Tabelle 5.4: Bewertung der einfachen Beispielarchitektur

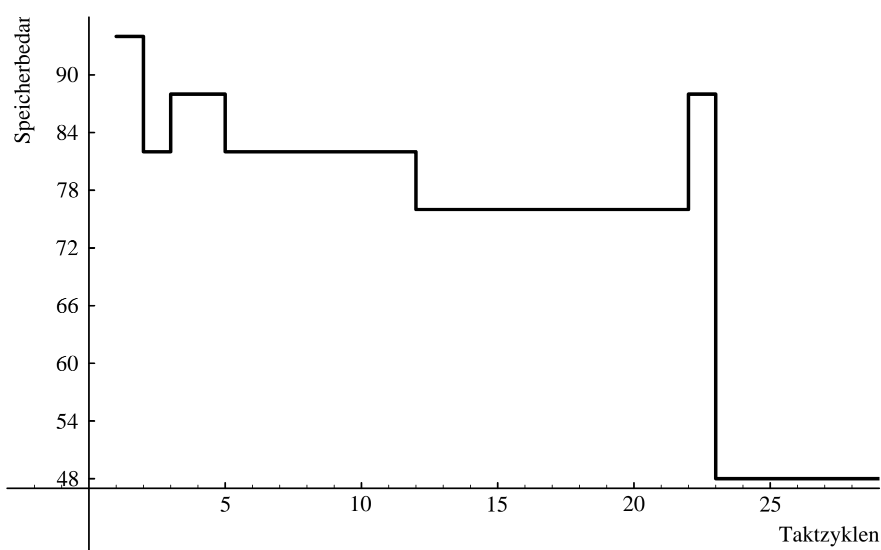


Abbildung 5.11: Gesamtspeicherbedarf

beträgt der gesamte Speicherbedarf also 12 Einheiten. An Kontrollschritt $t = 32$ wird der für Arbeitsspeicher reservierte Bereich freigegeben und nur Transferspeicher gehalten. Die Ausführung der von LF datenabhängigen Operationen DIFF und REC beginnt an den Kontrollschritten $t = 32$ beziehungsweise $t = 62$. Da der Transferspeicher von LF reserviert bleibt, bis die Ausführung der letzten von LF datenabhängigen Operation beginnt, muss dieser Bereich bis zum Kontrollschritt $t = 61$ erhalten bleiben. Aus Abbildung 5.8 geht hervor, dass der Speicher an Kontrollschritt $t = 62$ wieder freigegeben ist.

In diesem einfachen Architekturmodell wird angenommen, dass alle Operationen auf eine gemeinsame Speicherbank GC zugreifen. Der Gesamtspeicherbedarf ergibt sich daher als Summe der Erfordernisse der einzelnen Operationen, wie in Abb. 5.11 für die Dauer eines Iterationsintervalls dargestellt ist. Mit dem Iterationsintervall von $P = 29$ Kontrollschritten und einem maximalen Speicherbedarf von 94 Einheiten ergibt sich eine Effizienz des Speicher von $E(GC) = 0.625$

Die bis jetzt diskutierten Architekturparameter wie Ablaufplanung, Modulzuweisung, Latenz und Iterationsintervall beeinflussen stark den Speicherbedarf einer Implementie-

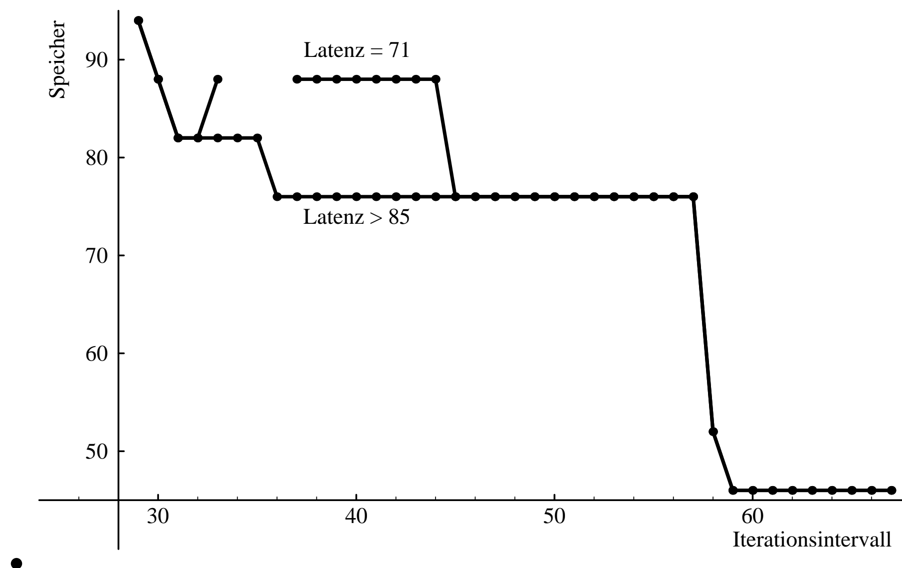


Abbildung 5.12: Speicherbedarf als Funktion des Iterationsintervalls

rung. Die folgenden Überlegungen dienen der Erklärung dieser Abhängigkeiten.

- Das Erhöhen des Iterationsintervalls bei einer gegebenen Latenz reduziert den Fliessbandfaktor. Die daraus resultierende reduzierte Überlappung von Iterationen kann zu einer Reduktion des benötigten minimalen Speicherbedarfs führen. Der minimale Speicherbedarf des Videokodierers nach Abbildung 5.6 mit den Kenndaten nach Tabelle 5.3 wurde für zwei feste Werte der Latenz L bei veränderlichem Iterationsintervall bestimmt. Die Ergebnisse der Versuchsreihe sind in Abbildung 5.12 dargestellt. Die obere Kurve entspricht einer maximalen Latenz von $L = 71$ Kontrollschritten; dies ist die minimal erreichbare Latenz bei dem kleinst möglichen Iterationsintervall von $P = 29$ Kontrollschritten. Da es bei diesen Werten nur einen kleinen Freiheitsgrad bei der Ablaufplanung der Operationen gibt, kann es sogar zu einem Anstieg des benötigten Speicherbedarfs kommen. Wie aus Abbildung 5.12 ferner hervorgeht, konnte für Werte des Iterationsintervalls von $P = 33, 34, 35$ Kontrollschritten kein zulässiger Ablaufplan berechnet werden. Die sprungartige Abnahme im Speicherbedarf bei einem Iterationsintervall von $P = 59$ erklärt sich aus der Tatsache, dass in den entsprechenden Ablaufplänen die Operation BM vor allen anderen Operationen ausgeführt werden kann. Der Arbeitsspeicher von BM überlappt sich daher nicht mehr mit dem Speicherbedarf der anderen Operationen.

- Eine Vergrösserung der Latenz L bei festem Iterationsintervall führt im Allgemeinen zu erhöhtem Speicherbedarf. Dies gilt dann, wenn die Latenz ausgenutzt wird, so dass der rechnerische Fliessbandfaktor $F = L/P$ mit dem tatsächlichen übereinstimmt. Das Planungsverfahren sollte jedoch in der Lage sein, bei entsprechenden Zielfunktionen einen Ablaufplan so zu bestimmen, dass Optimalität erzielt wird und die Anfangszeiten der Operationen innerhalb der Latenz entsprechend verteilt werden. So wird bei Optimierung des Speicherbedarf nach Abbildung 5.12 bei einem Iterationsintervall von $P = 29$ Kontrollschritten die Latenz von $L = 71$ Kontrollschritten benötigt: Eine kleinere Latenz ist nicht möglich und es ergibt sich ein Fliessbandfaktor von $F = 2.5$. Bei einem Iterationsintervall von $P = 60$ kann die

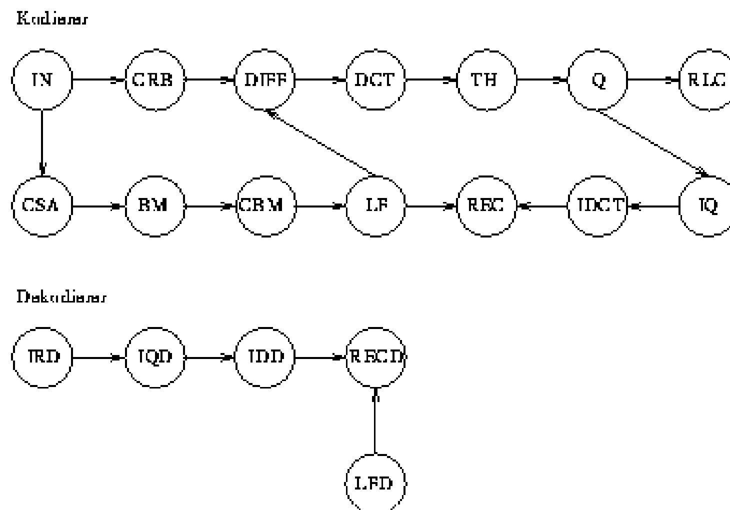


Abbildung 5.13: Vollständiger Abhängigkeitsgraph eines Videokodierers

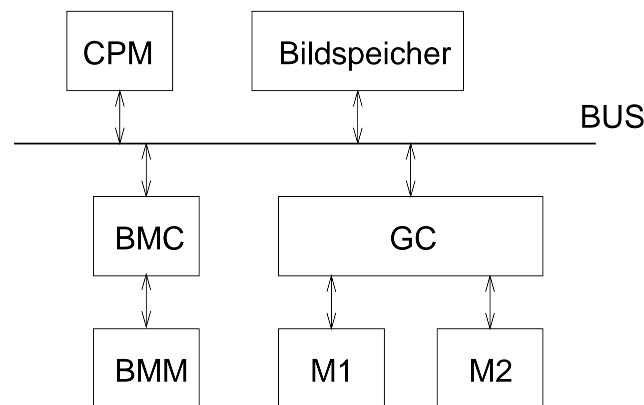


Abbildung 5.14: Realistische Zielarchitektur eines Videokodierers

Ausführung der letzten Operation schon an Kontrollschritt $t = 60$ begonnen werden, so dass sich ein Fliessbandfaktor von $F = 1$ ergibt. Entsprechend klein ist dann auch der Speicherbedarf. Andererseits geht aus Abbildung 5.12 auch hervor, dass es bei kleinem Iterationsintervall manchmal sinnvoll sein kann, die Latenz zu erhöhen, da dann dann die Anzahl von Kontrollschritten t erhöht wird, an denen die Ausführung einer Operation begonnen werden kann.

5.4.2 Ein realistisches Architekturbeispiel

In diesem Abschnitt werden ein genaueres Algorithmenmodell und realistischere Architekturmodelle des Videokodierers vorgestellt. Abbildung 5.13 und 5.14 stellen den Abhängigkeitsgraph und die Zielarchitektur dar. Dieses Beispiel zeigt, wie sich Algorithmen- und Architekturentwurf gegenseitig beeinflussen.

Im Allgemeinen wird bei Videokonferenzsystemen Sende- und Empfangsbetrieb gewünscht. Der Videokodierer muss daher neben der Kodierschleife auch den umgekehrten

Prozess gestatten. Der Algorithmus wurde daher um die Operationen des Dekodierers erweitert; es sind dies die inverse Lauflängenkodierung IRD, die inverse Quantisierung IQD, die inverse diskrete Cosinustransformation IDD, die Signalrekonstruktion RECD und die Schleifenfilterung LFD. Zusätzlich eingeführt wurden die Kopieroperationen CRB (Copy Reference Block), CSA (Copy Search Area) und CBM (Copy Best Match).

Gegenüber der in Abb. 5.7 gezeigten Architektur sind die folgenden Veränderungen vorgenommen worden: Zur Abarbeitung der Prozessoren sind drei Prozessoren BMM, M1 und M2 vorhanden. Der Algorithmus nach Abb. 5.13 wurde auf zwei Zielarchitekturen nach Abbildung 5.14 abgebildet und die unterschiedlichen Leistungs- und Kostenparameter untersucht. Die Architekturen unterscheiden sich bezüglich der Typen der Prozessoren M1 und M2 und der Zuweisung von Operationen zu den Prozessoren. Gemeinsam ist die Abarbeitung des Blockvergleichs BM auf einem Spezialprozessor BMM.

- **Modell 1** Die Zielarchitektur nach Modell 1 benutzt als Prozessorelemente M1 und M2 einen Vielzweckprozessor PUM und einen Spezialprozessor DCTM zur Ausführung der diskreten Cosinustransformationen.

- **Modell 2** In der Zielarchitektur nach Modell 2 werden zwei Vielzweckprozessoren PU1 und PU2 eingesetzt. Die Zuweisung der Operationen zu den Prozessoren ist nicht festgelegt; sie erfolgt durch Modulauswahl nach dem Kriterium der Minimierung von Iterationsintervall (gleichbedeutend mit Maximierung der Durchsatzrate) und der Optimierung des Speicherbedarfs.

In Tabelle 5.5 sind die Parameter der Architekturen nach Modell 1 und Modell 2 aufgeführt.

Im Architekturmodell nach Abbildung 5.7 greifen alle Prozessoren auf einen gemeinsamen Speicher GC zu. Die realistischeren Architekturmodelle sehen zwei Speichermodule vor: Der Spezialprozessor BMM hat einen eigenen Speicher BMC, die Prozessoren M1 und M2 teilen sich einen gemeinsamen Speicher GC. Da die Prozessorelemente nur auf die ihnen zugeteilten Speicherelementen zugreifen können, muss der Algorithmus so verändert werden, dass zusätzliche Kopieroperationen die Daten zwischen den Speicherblöcken kopieren, wenn die Daten zweier Operationen v_i und v_j mit $(v_i, v_j) \in E_D$ in verschiedenen Speicherblöcken abgelegt werden. Dies sind die Operationen CRB (Copy Reference Block), CSA (Copy Search Area) und CBM (Copy Best Match). Da diese Operationen von dem Kontrollbaustein des Speichers ausgeführt werden, wird ihnen der dedizierte Prozessor CPM (Copy Module) zugeordnet.

Mit den beschriebenen Planungsverfahren wurden Ablaufpläne für die obigen Architekturmodelle berechnet. In den Abbildungen 5.15 und 5.16 sind Ergebnisse für Ablaufpläne mit minimalem Iterationsintervall und der jeweils minimalen Latenz dargestellt. Die Ablaufpläne beinhalten die Optimierung des Speicherbedarfs.

Wie bereits erläutert, kann es bei konkurrierendem Zugriff von Prozessoren auf Speichermodule zu Zugriffskonflikten kommen. Im vorliegenden Fall greifen die Prozessoren CPM, M1 und M2 auf den gemeinsamen Speicher GC zu. Als Realisierungsform für den Speicher sind Bausteine vorgesehen, die jeweils gleichzeitiges Lesen und Schreiben auf zwei Toren gestatten (dual ported RAM). Um den gemeinsamen Zugriff von CPM, M1 und M2 zu verhindern, werden die beiden Tore als Ressource betrachtet, von der nur zwei Exemplare zur Verfügung stehen. Wie aus Abbildung 5.16 hervorgeht, verhindert diese zusätzliche Bedingung, dass für Modell 2 ein Ablaufplan mit dem theoretischen Minimum des Iterationsintervalls von $\gamma = 27$ Kontrollschritten berechnet werden kann. Andererseits müsste

$v_i \in E_S$	$w_{i,k}$	$\mu_{i,GC}$	$\nu_{i,GC}$	$\mu_{i,BMC}$	$\nu_{i,BMC}$	Ressourcetypen
IN	0	0	0	0	0	-
BM	22	0	0	40	6	BMM
LF	9	6	6	0	0	PUM
DIF	1	6	6	0	0	PUM
DCT	8/4	6	12	0	0	DCTM/PUM
Q	1	12	12	0	0	PUM
IQ	1	12	12	0	0	PUM
IDCT	8/4	12	6	0	0	DCTM/PUM
REC	1	6	6	0	0	PUM
CRB	1	0	6	0	0	PUM
CSA	1	0	0	0	6	PUM
TH	8	12	12	0	0	PUM
RLC	8	12	0	0	0	PUM
IRD	1	6	6	0	0	PUM
IQD	1	6	6	0	0	PUM
IDD	8/4	6	6	0	0	DCTM/PUM
CBM	1	0	6	6	0	PUM
RECD	1	6	0	0	0	PUM
LFD	9	6	6	0	0	PUM

Tabelle 5.5: Parameter der Architekturen nach Modell 1 und 2

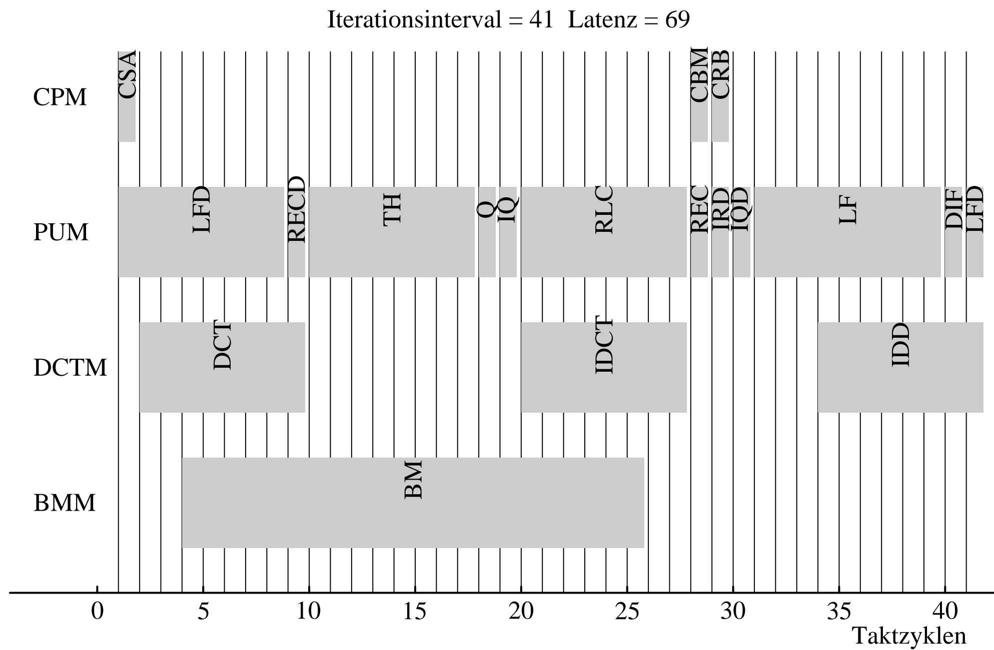


Abbildung 5.15: Ablaufplan für Modell 1

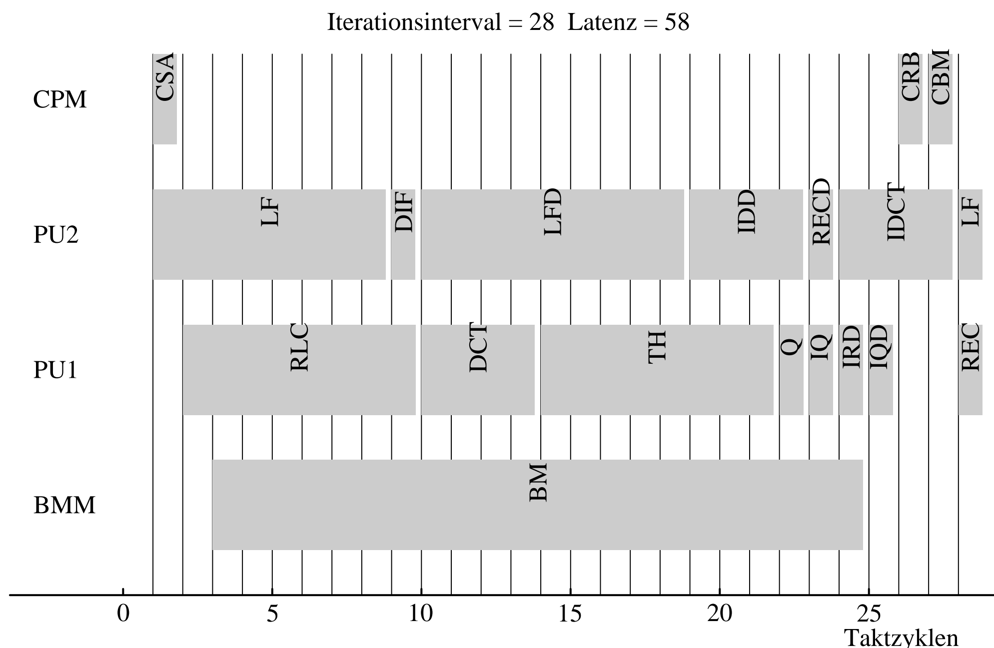


Abbildung 5.16: Ablaufplan für Modell 2

zur Realisierung dieses unwesentlich schnelleren Ablaufplans ein bedeutend aufwendigerer Speicherbaustein eingesetzt werden. Abb. 5.17 zeigt den Verlauf der Speicherbelegung für Modell 2.

Die Leistungsbewertungen für die Architekturen nach Modell 1 und Modell 2 sind in Tabelle 5.6 zusammengestellt. Es ergibt sich direkt, dass Modell 2 hinsichtlich Verarbei-

Parameter	Modell 1	Modell 2
Iterationsintervall P	41	28
Latenz L	69	58
Speicherbedarf	88	100
Effizienz E	0.55	0.7
Lastausgleichsfaktor Z	0.07	0.11
Fliessbandfaktor F	1.68	2.07

Tabelle 5.6: Leistungsbewertungen der Architekturen nach Modell 1 und Modell 2

tungsgeschwindigkeit die überlegene Implementierung darstellt. Ausserdem ist offensichtlich, dass die Effizienz von Modell 2 höher ist, da die Operationen gleichmässig auf die Module PU1 und PU2 aufgeteilt werden. Wie andererseits aus Tabelle 5.7 hervorgeht, ist der Flächenverbrauch eines Universalprozessors PU höher als der eines Spezialprozessors DCTM. Der Speicherbedarf von Modell 2 ist ebenfalls höher als der von Modell 1; dies lasst sich mit dem höheren Fliessbandfaktor P erklären. Der erhöhte Speicherbedarf wirkt sich natürlich auf die Fläche von einer Implementierung von Modell 2 aus. Die Optimierung dieser Kriterien kann einfach in das ganzzahlige lineare Programm übernommen

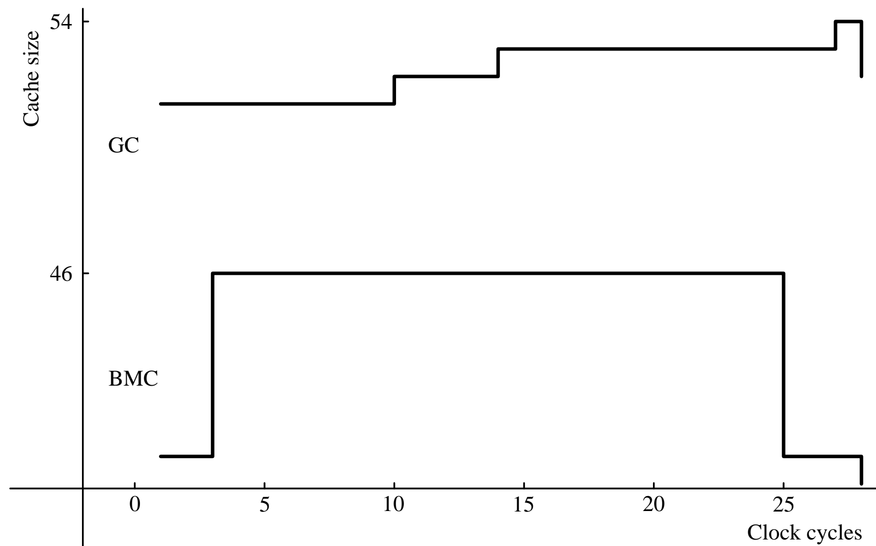


Abbildung 5.17: Speicherbedarf des Videokodierers

Modul	Fläche
BMM	4
DCTM	8
PUM	20
Überhang	20
Speicher	0.25 / (8 × 8 Block)

Tabelle 5.7: Flächenbedarf der Module

werden, indem man dem Speicherbedarf und den Verarbeitungsmodulen Kosten zuweist, die den Flächenbedarf messen. Falls für den Flächenbedarf der Prozessoren die Angaben aus Tabelle 5.7 angenommen werden, folgt für den Flächenbedarf A der Architekturen nach Modell 1 und Modell 2:

$$
\begin{aligned}
A(\text{Modell 1}) &= 74 \\
A(\text{Modell 2}) &= 89
\end{aligned}
$$

Um nun Verarbeitungsgeschwindigkeit und Flächenbedarf einer Schaltung gegeneinander abwiegen zu können, wurde das AT-Mass als das Produkt von Fläche A und Durchsatzrate T einer Schaltung eingeführt. Da die Durchsatzrate T mit dem Iterationsintervall P identisch ist, ergibt sich für den Vergleich

$$
\begin{aligned}
AT(\text{Modell 1}) &= 3034 \\
AT(\text{Modell 2}) &= 2492
\end{aligned}
$$

Da ein möglichst kleines AT-Produkt angestrebt wird, ist eine Realisierung mit dem Architekturmodell 2 vorzuziehen.

In Abbildung 5.18 werden die Fliessband-Faktoren der beiden Architekturmodelle verglichen. Obwohl Modell 2 ein geringeres Iterationsintervall zulässt, hat es einen erhöhten

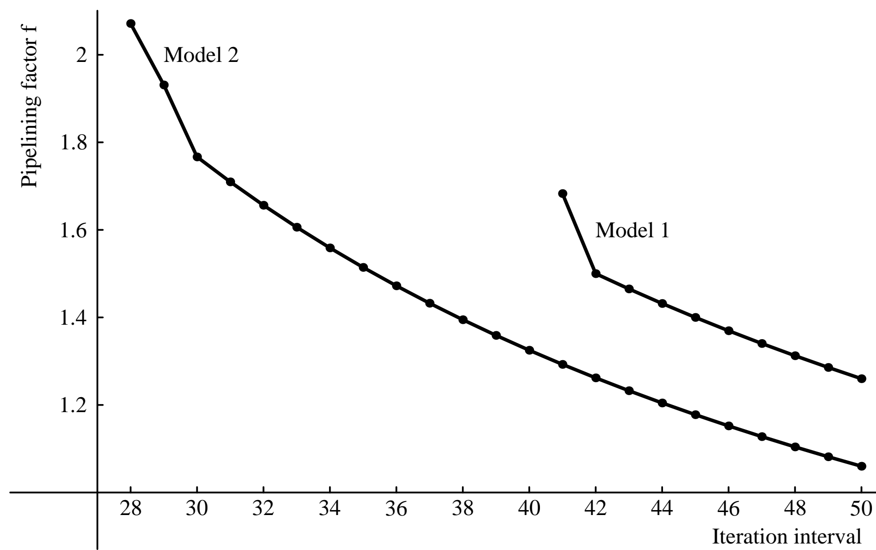


Abbildung 5.18: Vergleich der Fliessband-Faktoren

Speicherbedarf. Dies, verbunden mit dem höheren Fliessband-Faktor, kann zu einem erhöhten Aufwand bei der Adress-Generierung führen.

Literaturverzeichnis

[1] W. Wolf, *Computers as Components.* Morgan Kaufmann Publishers, 2001.

[2] P. Marwedel, *Embedded System Design.* Kluwer Academic Publishers, 2003.

[3] G. DeMicheli, *Synthesis and Optimization of Digital Circuits.* New York: Mc Graw Hill, 1994.

[4] D. Gajski, F. Vahid, S. Naranyan, and J. Gong, *Specification and Design of Embedded Systems.* Englewood Cliffs, NJ: Prentice Hall, 1994.

[5] J. Teich, *Digitale Hardware/Software-Systeme.* Berlin, Heidelberg: Springer Verlag, ISBN 3-540-62433-3, 1997.

[6] G. Buttanzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, 1997.

[7] C. A. Petri, "Interpretations of a net theory," Tech. Rep. 75–07, GMD, Bonn, W. Germany, 1975.

[8] B. Baumgarten, *Petri-Netze Grundlagen und Anwendungen.* Mannheim: BI Wissenschaftsverlag, 1990.

[9] R. Milner, "A calculus of communicating systems," *Lecture Notes in Computer Science, Springer Verlag*, vol. 92, 1980.

[10] C. Hoare, *Communicating Sequential Processes.* Englewood Cliffs, NJ: Prentice Hall, 1985.

[11] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, 1987.

[12] G. Berry, "Programming a digital watch in ESTEREL," Tech. Rep. 08–91, Centre de Mathematiques Appliquees, Ecole des Mines de Paris, Sophia-Antipolis, 1991.

[13] F. Commoner and A. Holt, "Marked directed graphs," *Journal of Computer and System Sciences*, vol. 5, pp. 511–523, 1971.

[14] E. Lee and D. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[15] IEEE, *IEEE Standard VHDL Language Reference Manual.* IEEE STd. 1076-1987: IEEE, 1987.

[16] M. McFarland, A. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, vol. 78, pp. 301–318, February 1990.

[17] C. Tseng and D. P. Siewiorek, "Automated synthesis of datapaths in digital systems," *IEEE Trans. on CAD*, vol. 5, pp. 274–277, July 1986.

[18] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms.* Cambridge, MA: McGraw-Hill, 1990.

[19] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: Freeman, 1976.

[20] P. Marwedel, "A new synthesis algorithm for the mimola software system," in *Proc. of the 23rd Design Automation Conference*, (ACM, IEEE), pp. 271–277, 1986.

[21] H. J. Whitehouse, S. Y. Kung, and T. Kailath, *VLSI and Modern Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1985.

[22] H. Trickey, "Flamel: A high level hardware compiler," *IEEE Trans. on CAD*, vol. 6, no. 2, pp. 259–269, 1987.

[23] S. Davidson, D. Landskov, B. Shriver, and P. Mallett, "Some experiments in local microcode compaction for horizontal machines," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 460–477, 1981.

[24] B. Pangrle and D. Gajski, "Design tools for intelligent silicon compilation," *IEEE Trans. on CAD*, vol. CAD-6, pp. 1098–1112, November 1987.

[25] G. Goosens and other, "An efficient microcode compiler for custom microprocessor DSP systems," in *Proc. of the Int. Conf. on CAD*, pp. 24–27, 1987.

[26] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis," *IEEE Transactions on Computer-Aided Design*, vol. CAD-8, pp. 661–679, July 1989.

[27] L. Hafer and A. Parker, "A formal method for the specification, analysis and design of register-transfer digital logic," *IEEE Transactions on Computer-Aided Design*, vol. CAD-2, pp. 4–18, 1983.

[28] C. Hwang, J. Lee, and Y. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Transactions on Computer-Aided Design*, vol. CAD-10, pp. 464–475, April 1991.

[29] R. Kaneshiro, K. Konstantinides, and J. Tani, "Task allocation and scheduling models for multiprocessor digital signal processing," *IEEE Trans. on Accoustics, Speech and Signal Processing*, vol. 38, pp. 2151–2161, December 1990.

[30] S. Prakash and A. C. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *Journal Parallel and Distributed Computing*, vol. 16, pp. 338–351, 1992.

[31] C. Gebotys and M. I. Elmasry, "Global optimization approach for architectural synthesis," *IEEE Journal on CAD*, vol. 12, pp. 1266–1278, September 1993.

[32] S. Devadas and R. Newton, "Algorithms for hardware allocation in data-path synthesis," *IEEE Trans. on CAS*, vol. CAD-8, no. 7, pp. 768–781, 1989.

[33] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning.* Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1989.

[34] J. H. Holland, *Adaption in Natural and Artificial Systems.* Cambridge, Massachusetts: Bradford Book, MIT Press, 1992.

[35] F. Harary, *Graph Theory.* Reading, MA: Addison-Wesley, 1972.

[36] A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assignment within large apertures," in *Proc. of the 8th Design Automation Workshop*, pp. 155–163, 1971.

[37] H. J. W. S. Y. Kung and T. Kailath, *VLSI and Modern Signal processing.* Eaglewood Cliffs, NJ: Prentice Hall, 1985.

[38] M. S. J. Wilberg and P. Pirsch, "Hierarchical multiprocessor system for video signal processing," in *Proc. SPIE*, vol. 1818, pp. 1076–1087, 1992.

[39] P. Pirsch, ed., *VLSI Implementations for Image Communications.* No. 2 in Advances in Image Communication, Elsevier, 1993.