eXamen.press

eXamen.press ist eine Reihe, die Theorie und Praxis aus allen Bereichen der Informatik für die Hochschulausbildung vermittelt.

Peter Marwedel

Eingebettete Systeme

Übersetzt aus dem Englischen von Lars Wehmeyer

Springer

Peter Marwedel
Fakultät für Informatik
Technische Universität Dortmund
Otto-Hahn-Str. 16
44221 Dortmund
peter.marwedel@tu-dortmund.de

Übersetzer

Lars Wehmeyer

korrigierter Nachdruck 2008

ISBN 978-3-540-34048-5 e-ISBN 978-3-540-34049-2

DOI 10.1007/978-3-540-34049-2

ISSN 1614-5216

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Einbandgestaltung: KünkelLopka Werbeagentur, Heidelberg

Gedruckt auf säurefreiem Papier

9 8 7 6 5 4 3 2 1

springer.com

Dieses Buch ist meiner Familie gewidmet.

Vorwort

Relevanz eingebetteter Systeme

Eingebettete Systeme können definiert werden als informationsverarbeitende Systeme, die in ein umgebendes Produkt wie z.B. Autos, Telekommunikationsgeräte oder Produktionsmaschinen eingebettet sind. Solche Systeme haben viele gemeinsame Charakteristiken, u.a. das Einhalten von Zeitbedingungen, Zuverlässigkeit und Effizienz. Die Technologie eingebetteter Systeme ist eine Grundvoraussetzung, um das Paradigma der allgegenwärtigen Information (*ubiquitous information*) zu gewährleisten – eines der Hauptziele der modernen Informationstechnik (IT).

Nach dem Erfolg der IT im Bereich der Büroanwendungen und Geschäftsprozesse werden eingebettete Systeme in den kommenden Jahren **das** wichtigste Anwendungsgebiet für Informationstechnik sein. Aufgrund dieser Erwartungen wurde bereits der Begriff ***Post-PC Era*** geprägt. Dieser Begriff drückt aus, dass Standard-PCs in Zukunft weniger dominant sein werden. Prozessoren und Sofware werden zunehmend in viel kleineren Systemen verwendet werden, und in vielen Fällen werden sie sogar völlig unsichtbar sein, was zum Begriff des **verschwindenden Computers** geführt hat. Es ist heute bereits offensichtlich, dass technische Produkte sehr hochentwickelt sein müssen, um das Interesse der Kunden zu wecken. Autos, Kameras, Fernseher, Handys usw. lassen sich heute ohne clevere integrierte Softwarelösungen kaum noch verkaufen. Bereits heute übersteigt die Anzahl von Prozessoren in eingebetteten Systemen die Anzahl von Prozessoren, die in PCs verbaut sind – und dieser Trend wird sich weiter fortsetzen. Nach aktuellen Vorhersagen wird die Größe und Komplexität eingebetteter Software in den kommenden Jahren sehr stark wachsen. Es gibt auf diesem Gebiet eine neue Form des „Mooreschen Gesetzes“: *In vielen Produkten im Bereich der Konsumelektronik verdoppelt sich die Größe der Software alle zwei Jahre* [Vaandrager, 1998].

Diese Relevanz eingebetteter Systeme wird in den aktuellen Studienplänen nicht ausreichend berücksichtigt. Dieses Buch soll dazu beitragen, die Situation zu verbessern. Es stellt Materialien für einen einführenden Kurs zu eingebetteten Systemen zur Verfügung, kann aber auch außerhalb von Vorlesungen eingesetzt werden.

Zielgruppe dieses Buches

Dieses Buch wurde für die folgenden Leser konzipiert:

- Informatik-, Ingenieur-Informatik- und Elektrotechnik-Studenten, die sich im Bereich der eingebetteten Systeme spezialisieren wollen. Das Buch ist geeignet für Studierende ab dem dritten Studienjahr, die bereits ein grundlegendes Verständnis von Rechner-Hard- und Software haben. Es ebnet den Weg zu fortgeschrittenen Themen, die in Folgekursen vertieft werden sollten.
- Ingenieure, die bislang im Bereich der Hardware für Rechensysteme gearbeitet haben und die sich intensiver mit dem Bereich der Software für eingebettete Systeme beschäftigen wollen. Dieses Buch sollte ein ausreichendes Hintergrundwissen zur Verfügung stellen, um aktuelle relevante Publikationen auf dem Gebiet zu verstehen.
- Professoren, die einen Studiengang oder eine Studienrichtung für eingebettete Systeme konzipieren.

Eingliederung eingebetteter Systeme in einen Studienplan

Dieses Buch setzt Grundwissen in den folgenden Gebieten voraus (s. Abb. 0.1);

- Elektrische Schaltnetze auf Oberstufen-Niveau (z.B. Kirchhoffsche Gesetze),
- Operationsverstärker (optional),
- Rechner-Hardware, z.B. anhand des einführenden Buches von J.L. Hennessy und D.A. Patterson [Hennessy und Patterson, 1995],
- Grundwissen digitaler Schaltungen wie etwa Gatter und Register,
- Programmierung,
- Endliche Automaten,
- Fundamentale mathematische Konzepte wie Tupel, Integrale und lineare Gleichungssysteme, möglichst auch die Grundlagen der Wahrscheinlichkeitstheorie,

- Algorithmen (Graphenalgorithmen und Optimierungsalgorithmen wie z.B. *Branch-and-Bound*),
- NP-Vollständigkeit und Berechenbarkeit.

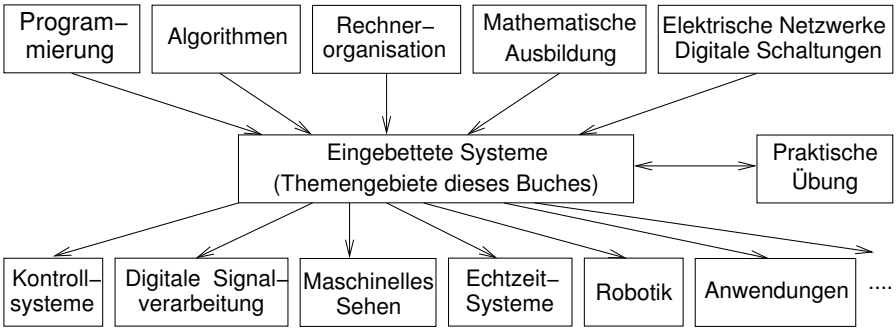


Abb. 0.1. Positionierung der Themen dieses Buches

Eines der Hauptziele dieses Buches ist es, einen Überblick über den Entwurf eingebetteter Systeme zu geben und die wichtigsten Themen im Bereich der eingebetteten Systeme zueinander in Beziehung zu setzen. Es soll sowohl Studierende als auch Lehrende dazu motivieren, sich weitere Themen zu erarbeiten. Einige Themen werden in diesem Buch erschöpfend behandelt, andere nur kurz angerissen. Diese kürzeren Abschnitte sollen helfen, verwandte Themengebiete in den Gesamtkontext einordnen zu können. Dadurch können Lehrende entsprechend ihrer eigenen Vorlieben Zusatzmaterial in den Kurs einbauen. Das Buch sollte durch weiterführende Kurse ergänzt werden, die vertiefendes Wissen in den folgenden Themengebieten vermitteln:

- Digitale Signalverarbeitung,
- Robotik,
- Maschinelles Sehen,
- Sensoren und Aktuatoren,
- Echtzeitsysteme, Echtzeitbetriebssysteme und Echtzeit-Scheduling,
- Regelungstechnik,
- Spezifikationssprachen für eingebettete Systeme,
- Rechnergestützte Entwurfsumgebungen für applikationsspezifische Hardware,
- Formale Verifikation von Hardware,
- Systematisches Testen von Hardware und Software,
- Leistungsbewertung von Rechensystemen,

- Entwurfsmethoden für energiesparende Systeme,
- Sicherheit und Verlässlichkeit von Computersystemen,
- *Ubiquitous Computing* (allgegenwärtiges Rechnen),
- Anwendungsgebiete in der Telekommunikation, im Automobilbereich, in medizinischen Geräten und intelligenten Häusern,
- Konsequenzen des Einsatzes eingebetteter Systeme.

Ein auf diesem Buch aufbauender Kurs sollte durch eine Übung vervollständigt werden, in der beispielsweise programmierbare Roboter wie Lego Mindstorms^{TM} oder ähnliche verwendet werden. Außerdem sollten Studierende praktische Erfahrungen mit StateCharts-basierten Werkzeugen machen können.

Zusatzinformationen zu diesem Buch sind von der folgenden Webseite erhältlich:

http://ls12-www.cs.tu-dortmund.de/~marwedel/es-book

Diese Seite beinhaltet Verweise zu Präsentationsfolien, Übungen, Hinweise zum Übungsbetrieb, Referenzen zu einer Auswahl von aktuellen Publikationen sowie Fehlerkorrekturen. Leser, die Fehler entdeckt haben oder die Vorschläge zur Verbesserung des Buches beitragen möchte, wenden sich bitte per E-Mail an Peter.Marwedel@tu-dortmund.de.

Übungen können auch anhand von Informationen aus der folgenden Auswahl von Büchern konzipiert werden: [Ganssle, 1992], [Ball, 1996], [Ball, 1998], [Barr, 1999], [Ganssle, 2000], [Wolf, 2001], [Buttazzo, 2002].

Die Verwendung von Namen in diesem Buch ohne Hinweise auf Urheberrechte oder eingetragene Warenzeichen bedeutet nicht, dass diese Namen nicht evtl. durch Dritte geschützt sind.

Viel Spaß beim Lesen dieses Buches!

Dortmund, September 2003

P. Marwedel

Herzlich willkommen zur aktualisierten Version dieses Buches! Die Verschmelzung des Kluwer- und Springer-Verlags macht es möglich, weniger als zwei Jahre nach dem Erscheinen der Erstausgabe 2003 eine neue Version dieses Buches herauszubringen. In dieser aktualisierten Version wurden alle gefundenen Tipp- und sonstigen Fehler der ersten Ausgabe korrigiert. Außerdem wurden alle Internet-Referenzen überprüft und aktualisiert. Abgesehen von diesen Änderungen ist der Inhalt des Buches gleich geblieben. Eine Liste von korrigierten Fehlern ist auf der oben genannten Webseite verfügbar.

Viel Spaß beim Lesen dieses aktualisierten Buches!

Dortmund, August 2005

P. Marwedel

Die vorliegende deutsche Fassung des Buches basiert auf der englischen Version des Buches aus dem Jahr 2005. Gegenüber dem englischen Original wurden mehrere Themen etwas ergänzt. Insbesondere wurde die Beschreibung von Scheduling-Verfahren um Beweise ergänzt, die Darstellung von Berechnungsmodellen (*models of computation*) überarbeitet und das Kapitel über Validierung zu einem Kapitel über Evaluation und Validierung erweitert. Dennoch sollten in der Regel die deutsche und die englische Fassung nebeneinander benutzt werden können.

Dortmund, Januar 2007
P. Marwedel (Autor des englischen Originals)
L. Wehmeyer (Übersetzer der deutschen Ausgabe)

Die vorliegende korrigierte Fassung der ersten Auflage der deutschen Ausgabe unterscheidet sich von der ursprünglichen Fassung der ersten Auflage nur durch die Beseitigung von Druckfehlern und die Überprüfung aller Referenzen auf das *World Wide Web*. Fehlermeldungen von Melanie Schmidt, Jan-Philipp Kappmeier und Matthias Steinkamp wurden berücksichtigt. Reinhard von Hanxleden (Universität Kiel) schlug neue Formulierungen im Abschnitt über Esterel vor. Beide Fassungen können nebeneinander benutzt werden.

Dortmund, April 2008
P. Marwedel

Danksagung

Meine Doktoranden, besonders Lars Wehmeyer, haben eine Vorabversion des Buches gründlich Korrektur gelesen. Die Teilnehmer des Kurses „Introduction to Embedded Systems“ im Sommer 2003, insbesondere Lars Bensmann, haben ebenfalls zum Buch beigetragen. Außerdem haben die folgenden Kollegen und Studierenden Kommentare und Hinweise gegeben, die in das Buch eingearbeitet wurden: W. Müller, F. Rammig (U. Paderborn), W. Rosenstiel (U. Tübingen), R. Dömer (UC Irvine) und W. Kluge (U. Kiel). Zur Vorbereitung und Zusammenstellung des Buches wurden Materialien der folgenden Personen verwendet: G. C. Buttazzo, D. Gajski, R. Gupta, J. P. Hayes, H. Kopetz, R. Leupers, R. Niemann, W. Rosenstiel und H. Takada. Korrekturen zur ersten Ausgabe dieses Buches wurden beigetragen von David Hec, Thomas Wiederkehr und Thorsten Wilmer. Eine Vorabversion der deutschen Fassung des Buches wurde von Martin Faßbach, Fatih Gedikli, Gordon Schlechter, Karl-Heinz Temme und Matthias Wiedenhorst Korrektur gelesen. Studierende des Kurses des Wintersemesters 2007/2008 haben ebenfalls zu den Korrekturen beigetragen. Selbstverständlich ist der Autor selbst verantwortlich für alle verbleibenden Fehler.

Dank gilt all denen, die die erhöhte Arbeitsbelastung des Autors während der Entstehung dieses Buches geduldig akzeptiert haben, insbesondere die dadurch oft fehlende Zeit sowohl für berufliche als auch persönliche Partner.

Schließlich sollte noch erwähnt werden, dass Kluwer Academic Publishers die Veröffentlichung dieses Buches vom ersten Konzept an unterstützt hat. Durch die Unterstützung des Verlags wurde die Arbeit an diesem Buch vorangetrieben.

Nach der Verschmelzung von Kluwer mit dem Springer-Verlag hat der Springer-Verlag großes Interesse an der Veröffentlichung einer deutschen Ausgabe gezeigt.

Inhaltsverzeichnis

1

Einleitung

1.1 Begriffe und Abgrenzung

Bis in die späten 80er Jahre war die Informationsverarbeitung mit großen *Mainframe*-Rechnern und riesigen Bandlaufwerken verbunden. Während der 90er Jahre hat sich die Informationsverarbeitung zu den *Personal Computern*, den PCs, verlagert. Dieser Trend zur Miniaturisierung geht weiter, und die Mehrzahl informationsverarbeitender Geräte werden in naher Zukunft kleine, teilweise auch tragbare Computer sein, die in größere Produkte integriert sind. Das Vorhandensein von Prozessoren in diesen umgebenden Produkten, wie z.B. in Telekommunikationsgeräten, wird weniger sichtbar sein als beim klassischen PC. Daher wird dieser Trend als der **verschwindende Computer** bezeichnet. Allerdings bedeutet dieser Begriff nicht, dass die Computer tatsächlich verschwinden werden, sondern vielmehr, dass sie überall sein werden. Diese neue Art von Anwendungen der Informationsverarbeitung wird auch *ubiquitous computing* (**allgegenwärtiges Rechnen**), *pervasive computing* [Hansmann, 2001], [Burkhardt, 2001] oder *ambient intelligence* [Marzano und Aarts, 2003] genannt. Diese drei Begriffe beschäftigen sich mit unterschiedlichen Nuancen der zukünftigen Informationsverarbeitung. *Ubiquitous Computing* konzentriert sich auf die langfristige Zielsetzung, Informationen jederzeit und überall zur Verfügung zu stellen, wohingegen *pervasive computing* sich mehr mit praktischen Aspekten, wie etwa der Ausnutzung bereits vorhandener Technologie, beschäftigt. Im Bereich *ambient intelligence* findet man einen Schwerpunkt auf der Kommunikationstechnologie im Wohnbereich der Zukunft sowie im Bereich der intelligenten Gebäudetechnik. Eingebettete Systeme sind einer der Ausgangspunkte dieser drei Gebiete und sie steuern einen Großteil der notwendigen Technologie bei. **Eingebettete Systeme sind informationsverarbeitende Systeme, die in ein größeres Produkt integriert sind**, und die normalerweise nicht direkt vom Benutzer wahrgenommen werden. Beispiele für eingebettete Systeme sind informationsverarbeitende Systeme in Telekommunikationsgeräten, in Transportsystemen

wie Autos, Zügen, Flugzeugen, in Fabriksteuerungen und in Unterhaltungsgeräten. Diese Systeme haben die folgenden gemeinsamen Charakteristiken:

- Häufig sind eingebettete Systeme mit der physikalischen Umwelt verbunden. **Sensoren** sammeln Informationen über die Umgebung und **Aktuatoren**[1] nehmen Einfluss auf die Umwelt.
- Eingebettete Systeme müssen **verlässlich** sein.

  Viele eingebettete Systeme sind sicherheitskritisch und müssen deshalb verlässlich arbeiten. Atomkraftwerke sind ein Beispiel für extrem sicherheitskritische Systeme, die zumindest zum Teil von Software gesteuert werden. Verlässlichkeit ist auch in anderen Systemen wichtig, so etwa in Autos, Zügen, Flugzeugen usw. Ein Hauptgrund, warum diese Systeme sicherheitskritisch sind, ist die Tatsache, dass sie direkt mit ihrer Umgebung in Verbindung stehen und einen unmittelbaren Einfluss auf diese Umgebung haben.

  Verlässlichkeit beinhaltet die folgenden Eigenschaften eines Systems:

  1. **Zuverlässigkeit:** Die Zuverlässigkeit ist die Wahrscheinlichkeit, dass ein System nicht ausfällt.
  2. **Wartbarkeit:** Die Wartbarkeit ist die Wahrscheinlichkeit, dass ein ausgefallenes System innerhalb einer bestimmten Zeitspanne wieder repariert werden kann.
  3. **Verfügbarkeit:** Die Verfügbarkeit ist die Wahrscheinlichkeit, dass ein System korrekt arbeitet. Sowohl Zuverlässigkeit als auch Wartbarkeit müssen hoch sein, um eine hohe Verfügbarkeit zu erreichen.
  4. **Sicherheit:** Dieser Ausdruck beschreibt die Eigenschaft, dass ein ausfallendes System keinen Schaden verursacht.
  5. **Integrität:** Dieser Begriff beschreibt die Eigenschaft, dass vertrauliche Daten geheim bleiben und dass die Authentizität der Kommunikation gewährleistet ist.

- Eingebettete Systeme müssen **effizient** sein. Die folgenden Größen können dazu verwendet werden, die Effizienz von eingebetteten Systemen zu beschreiben:

  1. **Energie:** Viele eingebettete Systeme sind in tragbare Geräte integriert, die ihre Energie über Batterien beziehen. Nach aktuellen Vorhersagen [SEMATECH, 2003] wird die Kapazität von Batterien nur langsam wachsen. Allerdings wachsen die Anforderungen an die Rechenkapazität, insbesondere für Multimedia-Anwendungen, sehr stark an, und die Kunden erwarten trotzdem lange Batterielaufzeiten. Daher muss die verfügbare elektrische Energie sehr effizient eingesetzt werden.

[1] Aktuatoren sind in diesem Kontext Geräte, die numerische Werte in physikalische Größen umwandeln.

2. **Codegröße:** Der Code, der auf einem eingebetteten System ausgeführt werden soll, muss innerhalb des Systems selbst gespeichert werden. Typischerweise gibt es in solchen Systemen keine Festplatte, auf der das Programm abgelegt werden kann. Das dynamische Hinzufügen von Code ist (noch) eine Ausnahmeerscheinung und tritt nur z.B. bei Java-Telefonen oder Set-Top-Boxen auf. Aufgrund der Rahmenbedingungen muss die Größe des Codes so klein wie möglich sein, er muss aber dennoch das gewünschte Verhalten zeigen. Dies gilt insbesondere für *Systems On a Chip* (SOCs), bei denen sich alle informationsverarbeitenden Schaltkreise auf einem einzigen Chip befinden. Wenn der Befehlsspeicher auf diesem Chip integriert werden soll, muss er zwangsläufig sehr effizient genutzt werden.
3. **Laufzeit-Effizienz:** Die gewünschte Funktionalität eines eingebetteten Systems sollte mittels eines minimalen Aufwands an Ressourcen zur Verfügung gestellt werden. Auch Zeitbedingungen sollten unter Verwendung minimaler Hardware- und Energie-Ressourcen sicher eingehalten werden können. Außerdem ist es wichtig, dass wirklich nur notwendige Hardware-Komponenten vorhanden sind. Komponenten, die nicht in der Lage sind, die maximal benötigte Zeit zur Ausführung einer Aufgabe (die sog. *Worst Case Execution Time* (WCET)) zu verbessern, wie z.B. viele Caches können in zeitkritischen Systemen weggelassen werden.
4. **Gewicht:** Alle tragbaren Geräte müssen ein möglichst geringes Gewicht aufweisen, da dies oft einen Kaufanreiz für ein bestimmtes System darstellt.
5. **Preis:** Für eingebettete Systeme, die in großen Stückzahlen hergestellt werden, insbesondere solche in Konsumelektronik, ist der Wettbewerb auf dem Markt ein sehr wichtiger Aspekt. Daher müssen sowohl Hardware-Komponenten als auch das Software-Entwicklungs-Budget sorgfältig und effizient genutzt werden.

- Diese Systeme werden meistens **für eine bestimmte Applikation entworfen.** Beispielsweise wird ein Prozessor im Steuergerät eines Autos oder eines Zuges nur immer genau seine Steuerungs-Software ausführen. Niemand würde versuchen, auf einem solchen System ein Computerspiel oder eine Tabellenkalkulation auszuführen. Dafür gibt es zwei Hauptgründe:

  1. Würde man die Ausführung anderer Programme auf diesen Systemen zulassen, würden sie dadurch weniger verlässlich.
  2. Die Ausführung anderer Programme ist nur möglich, wenn entsprechende Ressourcen wie z.B. Speicher ungenutzt sind. In einem effizienten System sollten allerdings alle Ressourcen effizient genutzt werden.

- Die meisten eingebetteten Systeme haben keine Tastatur, keine Maus und keine großen Bildschirme als Benutzerschnittstelle. Stattdessen besitzen sie

ein spezialisiertes Benutzer-Interface, das z.B. aus Knöpfen, einem Lenkrad, Pedalen usw. bestehen kann. Aus diesem Grund nimmt der Benutzer das informationsverarbeitende System im Hintergrund kaum wahr. Diese neue Computer-Ära wurde deshalb auch durch das Schlagwort des **verschwindenden Computers** charakterisiert.

- Viele eingebettete Systeme müssen **Echtzeit-Bedingungen** einhalten. Wenn die Berechnungen nicht innerhalb einer festgelegten Zeitspanne durchgeführt werden, kann dies zu schweren Qualitätseinbußen führen (wenn z.B. die Audio- oder Videoqualität durch Aussetzer oder Sprünge leidet), oder es kann sogar zu körperlichem Schaden des Benutzers führen (wenn sich z.B. Systeme in Autos, Zügen oder Flugzeugen nicht wie vorgesehen verhalten). Zeitbedingungen, deren Nichteinhalten zu einer Katastrophe führen kann, heißen **harte Zeitbedingungen** [Kopetz, 1997]. Alle anderen Zeitbedingungen heißen **weiche Zeitbedingungen**.

  Viele aktuelle informationsverarbeitende Systeme verwenden Techniken, um die durchschnittliche Geschwindigkeit zu erhöhen. So verbessern etwa Caches die durchschnittliche Leistungsfähigkeit eines Systems. In anderen Fällen wird eine zuverlässige Kommunikation dadurch erreicht, dass bestimmte Nachrichten wiederholt werden. Pakete werden etwa von Internet-Protokollen erneut verschickt, wenn die Originalnachricht verlorengegangen ist. Im Durchschnitt führen solche wiederholten Sendevorgänge (hoffentlich) nur zu einer geringen Verringerung der Übertragungsgeschwindigkeit, obwohl die Verzögerung für eine bestimmte Nachricht Größenordnungen über der normalen Verzögerungszeit liegen kann. Im Bereich der echtzeitfähigen Systeme ist eine Argumentation, die auf durchschnittlicher Leistungsfähigkeit aufbaut, nicht akzeptabel. **Eine garantierte System-Antwortzeit muss ohne statistische Argumente erklärt werden können** [Kopetz, 1997].

- Viele eingebettete Systeme sind **hybride Systeme**. Das bedeutet, sie enthalten sowohl analoge als auch digitale Schaltungsteile. Analoge Teile verarbeiten kontinuierliche Signalwerte in kontinuierlicher Zeit, wohingegen digitale Teile diskrete Signalwerte in diskreten Zeitschritten verarbeiten.

- Typischerweise sind eingebettete Systeme **reaktive Systeme**, die man wie folgt definieren kann: *Ein reaktives System ist ein System, das in kontinuierlicher Interaktion mit seiner Umgebung steht und mit einer Geschwindigkeit arbeitet, die von seiner Umwelt vorgegeben wird* [Bergé et al., 1995]. Reaktive Systeme befinden sich in einem bestimmten Zustand und warten auf eine Eingabe. Nach jeder Eingabe führen sie eine Rechnung durch, erzeugen eine Ausgabe und wechseln in einen neuen Zustand. Aus diesem Grunde lassen sich solche Systeme gut als endliche Automaten modellieren. Reine mathematische Funktionen, die lediglich die von den Algorithmen gelösten Probleme beschreiben, wären keine geeigneten Modelle zur Beschreibung reaktiver Systeme.

- Eingebettete Systeme sind **in der Lehre und in öffentlichen Diskussionen unterrepräsentiert**. *Um eingebettete Prozessoren wird kein Wirbel im Fernsehen und in Zeitschriftenartikeln gemacht...* [Ryan, 1995]. Ein Problem bei der Vermittlung von Wissen über eingebettete Systeme ist die Ausstattung, die benötigt wird, um das Thema interessant und praktisch erfahrbar zu machen. Außerdem erschwert die hohe Komplexität von praktisch eingesetzten eingebetteten Systemen die Wissensvermittlung.

Wegen dieser gemeinsamen Charakteristiken eingebetteter Systeme erscheint es sinnvoll, gemeinsame Ansätze zum Entwurf solcher Systeme zu betrachten, statt sich nur auf einzelne Anwendungsgebiete und deren Lösung zu konzentrieren.

Natürlich hat nicht jedes eingebettete System alle oben genannten Eigenschaften. Wir können „eingebettete Systeme“ also wie folgt definieren: **Informationsverarbeitende Systeme, welche die meisten der oben aufgezählten Eigenschaften erfüllen, heißen eingebettete Systeme**. Diese Definition beinhaltet eine gewisse Unschärfe. Allerdings erscheint es weder notwendig noch möglich, diese Unschärfe zu vermeiden.

Die meisten der genannten charakteristischen Eigenschaften eingebetteter Systeme finden sich auch in den kürzlich eingeführten Gebieten des *ubiquitous computing* oder *pervasive computing*, auch bekannt als *ambient intelligence*. Diese Begriffe lassen sich auf deutsch mit **allgegenwärtiges Rechnen** übersetzen. Das Hauptziel dieser Gebiete ist es, Informationen **überall und jederzeit** verfügbar zu machen, weswegen sie sich auch mit Kommunikationstechnik beschäftigen. Abb. 1.1 zeigt eine graphische Darstellung, wie *ubiquitous computing* von eingebetteten Systemen und Kommunikationstechnik beeinflusst wird.

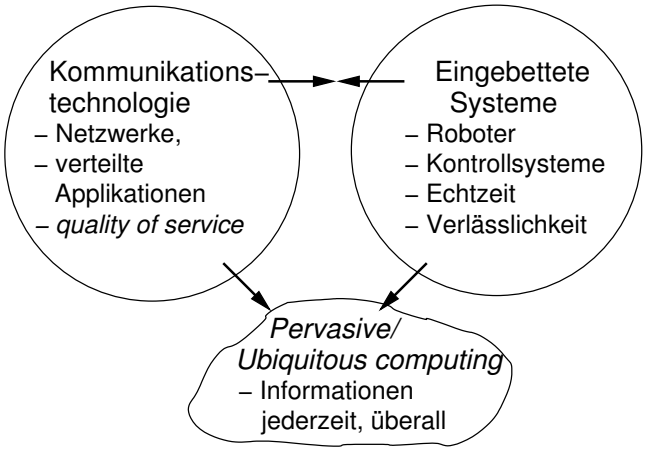


Abb. 1.1. Einfluss eingebetteter Systeme auf *ubiquitous computing*

Beispielsweise müssen beim *ubiquitous computing* Echtzeit- und Verlässlichkeitsbedingungen von eingebetteten Systemen eingehalten werden, während fundamentale Techniken der Kommunikationstechnik, wie z.B. Netzwerke, verwendet werden.

1.2 Anwendungsgebiete

Die folgende Liste beinhaltet Anwendungsbereiche, in denen eingebettete Systeme eingesetzt werden:

- **Automobilbereich:** Moderne Autos können nur noch verkauft werden, wenn sie einen beträchtliche Anteil an Elektronik enthalten, z.B. Airbag-Steuerungs-Systeme, elektronische Motorsteuerungen, ABS, Klimaanlagen, Navigationsgeräte mit GPS-Unterstützung und viele andere.
- **Bordelektronik im Flugzeug:** Einen Großteil des Gesamtwertes eines Flugzeugs machen heute die informationsverarbeitenden Systeme aus. Dazu gehören Flugkontrollsysteme, Anti-Kollisions-Systeme, Piloten-Informations-Systeme und andere. Die Verlässlichkeit der Systeme ist in diesem Bereich von allerhöchster Wichtigkeit.
- **Eisenbahntechnik:** Bei Zügen, Lokomotiven und stationären Sicherheitssystemen ist die Situation ähnlich wie bei Autos und Flugzeugen. Auch hier tragen die Sicherheitssysteme einen Großteil zum gesamten Wert bei, und die Verlässlichkeit hat eine ähnlich hohe Priorität.
- **Telekommunikation:** Die Verkaufszahlen von Handys sind in den vergangenen Jahren so stark gestiegen wie in kaum einem anderen Bereich. Schlüsselaspekte bei der Entwicklung von Handys sind das Beherrschen der Sendetechnik, digitale Signalverarbeitung und ein geringer Energieverbrauch.
- **Medizinische Systeme:** Im Bereich der medizinischen Geräte gibt es durch die Verwendung von informationsverarbeitenden Geräten ein großes Innovationspotential.
- **Militärische Anwendungen:** Informationsverarbeitung wird seit vielen Jahren in militärischen Ausrüstungsgegenständen verwendet. Eine der ersten Computeranwendungen war die automatische Auswertung von Radarsignalen.
- **Authentifizierungs-Systeme:** Eingebettete System können zur Benutzer-Authentifizierung verwendet werden. Beispielsweise können neuartige Zahlungssysteme die Sicherheit gegenüber klassischen Systemen deutlich erhöhen. Ein Beispiel für ein solches System ist etwa der SMARTpen® [IMEC, 1997] (s. Abb. 1.2).

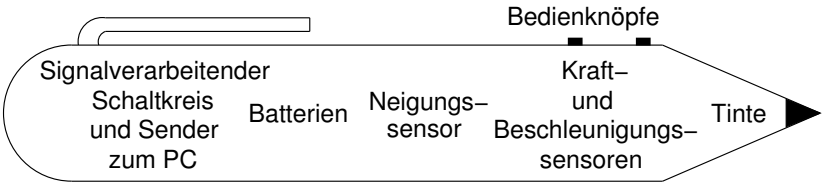


Abb. 1.2. SMARTpen

Der SMARTpen sieht aus wie ein Stift und analysiert während der Unterschrift physikalische Parameter wie z.B. Neigung, Anpressdruck und Beschleunigung. Diese Werte werden an einen Rechner übertragen und mit den Referenzdaten verglichen, die zu dem betreffenden Benutzer gespeichert sind. So kann man sowohl das Aussehen der Unterschrift als auch die Art und Weise, wie sie zu Papier gebracht wurde, mit der gespeicherten Information vergleichen.

Andere Authentifizierungssyteme sind etwa Fingerabdrucksensoren oder Gesichtserkennungs-Systeme.

- **Unterhaltungselektronik:** Video- und Audio-Geräte sind ein besonders wichtiger Sektor der Elektronikindustrie. Die Anzahl informationsverarbeitender Systeme auf diesem Gebiet erhöht sich ständig. Neue Dienste und bessere Qualität werden durch moderne Methoden der digitalen Signalverarbeitung erreicht. Viele Fernseher, Multimedia-Handys und Spielekonsolen beinhalten Hochleistungsprozessoren und -Speichersysteme. Diese stellen eine besondere Gattung eingebetteter Systeme dar.
- **Fabriksteuerungen:** Im Bereich der Fabriksteuerungen werden eingebettete Systeme traditionell seit Jahrzehnten eingesetzt. Die Sicherheit solcher Systeme ist sehr wichtig, wohingegen der Energieverbrauch ein weniger wichtiges Kriterium ist. Das Beispiel in Abb. 1.3 (entnommen aus Kopetz [Kopetz, 1997]) zeigt einen Flüssigkeitsbehälter, der mit einem Abflussrohr verbunden ist. Das Rohr ist mit einem Ventil und einem Sensor versehen. Ein Computer kann die Daten des Sensors verwenden, um die Flüssigkeitsmenge zu steuern, die durch das Rohr abfließt.

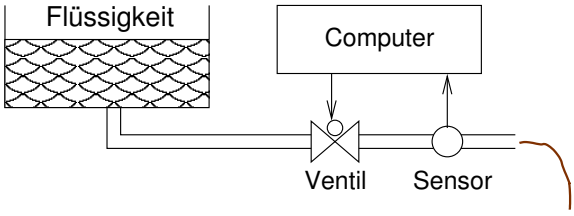


Abb. 1.3. Regelung eines Ventils

Das Ventil ist ein Beispiel für einen Aktuator (Definition s. Seite 2).

- **Intelligente Gebäude:** Informationsverarbeitende Systeme können verwendet werden, um den Komfort in Gebäuden zu verbessern, deren Energieverbrauch zu verringern oder um die Sicherheit zu erhöhen. Vorher unabhängige Teilsysteme müssen zu diesem Zweck miteinander verbunden werden. Der Trend geht dahin, Klimaanlagen, Lichtsteuerungen, Zugangskontrollen, Abrechnungssysteme sowie Informationsverteilung und -bereitstellung in ein einziges System zu integrieren. So kann man z.B. Energie sparen, indem man Klimaanlage, Heizung und Beleuchtung herunterfährt, wenn die betreffenden Räume leer sind. Verfügbare Räume können an geeigneten Stellen angezeigt werden, wodurch sowohl die Suche nach einem Raum für ein spontanes Meeting als auch die Aufgabe der Reinigungskräfte vereinfacht wird. Der Geräuschpegel der Klimaanlage kann an die aktuelle Situation im Raum angepasst werden. Eine intelligente Steuerung der Jalousien kann die Beleuchtung und die Nutzung der Klimaanlage optimieren. Für leere Räume können größere Temperaturschwankungen akzeptiert werden, außerdem kann die Beleuchtung entsprechend reduziert werden. Im Notfall kann eine Liste der belegten Räume am Eingang des Gebäudes angezeigt werden (vorausgesetzt, der notwendige Strom steht noch zur Verfügung).

  Anfangs werden solche Systeme wohl hauptsächlich in modernen Bürogebäuden zu finden sein.

- **Robotik:** Im Bereich der Robotik werden eingebettete Systeme ebenfalls seit langem eingesetzt. Ein wichtiger Bereich auf diesem Gebiet ist die Mechanik. Viele der oben genannten Charakteristiken treffen auch auf die Robotik zu. Seit kurzem gibt es eine neue Art von Robotern, die Tieren oder sogar dem Menschen nachempfunden sind. Abb. 1.4 zeigt ein Beispiel.

Diese Beispiele zeigen die vielfältigen Formen und Ausprägungen eingebetteter Systeme. Warum ist es sinnvoll, all diese verschiedenen Arten von eingebetteten Systemen in einem einzigen Buch zu beschreiben? Es ist sinnvoll, weil die Informationsverarbeitung in all diesen Systemen sehr ähnlich ist, obwohl die Systeme physikalisch vollkommen unterschiedlich sind.

1.3 Wachsende Relevanz von eingebetteten Systemen

Die Größe des Marktes für eingebettete Systeme kann aus einer Reihe von Perspektiven analysiert werden. Wenn man z.B. die Anzahl der momentan im Betrieb befindlichen komplexen Prozessoren betrachtet, so wurde geschätzt, dass mehr als 90% dieser Prozessoren in eingebetteten Systemen verwendet werden. Viele dieser eingebetteten Prozessoren sind 8-Bit-Prozessoren, trotzdem sind 75% aller 32-Bit-Prozessoren ebenfalls in eingebettete Syste-

Abb. 1.4. Roboter „Johnnie“ (mit freundlicher Genehmigung von H. Ulbrich, F. Pfeiffer, Lehrstuhl für Angewandte Mechanik, TU München), ©TU München

me integriert [Stiller, 2000]. Bereits 1996 wurde geschätzt, dass der durchschnittliche US-Amerikaner jeden Tag mit 60 Mikroprozessoren in Berührung kommt [Camposano und Wolf, 1996]. Einige Autos der Luxusklasse beinhalten mehr als 100 Prozessoren[2]. Diese Zahlen sind viel größer als man normalerweise annimmt, da vielen Menschen nicht bewusst ist, dass sie Prozessoren verwenden. Wie wichtig eingebettete Systeme sind, ist auch von der Journalistin Mary Ryan geäußert worden [Ryan, 1995]:

„... eingebettete Chips bilden das Rückgrat der von Elektronik getriebenen Welt, in der wir leben. ... sie sind in fast allem enthalten, was mit Elektrizität betrieben wird.“

Vielen Vorhersagen zufolge wird der Markt für eingebettete Systeme bald viel größer sein als der Markt für PC-ähnliche Systeme. Außerdem wird die Menge an Software, die in eingebetteten Systemen verwendet wird, stark zunehmen. Nach Vaandrager wird sich *„für viele Produkte im Bereich der Unterhaltungselektronik die Größe des Codes* ***alle zwei Jahre verdoppeln*** *“* [Vaandrager, 1998].

[2] Quelle: persönliche Kommunikation

Eingebettete Systeme bilden die Basis der sogenannten *post PC era*, in der sich die Informationsverarbeitung immer weiter weg von PCs, hin zu eingebetteten Systemen verlagert.

Die wachsende Zahl von Anwendungen macht es notwendig, Methoden zu entwickeln, die den Entwurf eingebetteter Systeme unterstützen. Die momentan verwendeten Technologien und Programme weisen noch starke Einschränkungen auf. So besteht ein großer Bedarf an besseren Spezifikationssprachen, an Programmen, die aus einer Spezifikation eine Implementierung ableiten, an Programmen, welche die Einhaltung von Zeitbedingungen prüfen können, an Echtzeitbetriebssystemen, an Entwurfsmethoden für energiesparende Systeme sowie an Entwurfsmethoden für verlässliche Systeme. Dieses Buch soll Grundlagen zu den wichtigsten Themen vermitteln und ein Ausgangspunkt für weitere Forschungsarbeiten auf diesem Gebiet sein.

1.4 Struktur dieses Buches

Viele der vorhandenen Bücher über eingebettete Systeme erläutern und beschreiben den Einsatz von Mikrocontrollern, sowie die in Mikrocontrollern verwendeten Speicher, die Ein- und Ausgabeeinheiten sowie die Struktur der Interrupts. Es gibt zahlreiche solcher Bücher, etwa [Ganssle, 1992], [Ball, 1996], [Ball, 1998], [Barr, 1999] und [Ganssle, 2000].

Aufgrund der steigenden Komplexität eingebetteter Systeme muss dieser Blickwinkel erweitert werden und zumindest verschiedene Spezifikationssprachen, Hardware-/Software-Codesign, Compiler-Techniken, Scheduling (Ablaufplanung) sowie Validierungstechniken enthalten. Diese Gebiete werden im vorliegenden Buch behandelt. Das Ziel dieses Buches ist es, Studierenden eine Einführung in das Gebiet der eingebetteten Systeme zu geben und ihnen damit die Möglichkeit zu eröffnen, die verschiedenen Teilgebiete einzuordnen.

Als weiterführende Literatur empfehlen wir folgende Quellen (die teilweise auch während der Vorbereitung dieses Buches herangezogen wurden):

- Es gibt eine Vielzahl von Informationsquellen zu Spezifikationssprachen. Zu nennen sind frühe Bücher von Young [Young, 1982], Burns und Wellings [Burns und Wellings, 1990] sowie von Bergé [Bergé et al., 1995]. Über neuere Sprachen wie beispielsweise Java, SystemC [Müller et al., 2003], SpecC [Gajski et al., 2000] usw. gibt es sehr viele Bücher.
- Einige Ansätze zum Entwurf und zum Einsatz von Echtzeitbetriebssystemen (*Real-Time Operating Systems* (RTOS)) werden im Buch von Kopetz [Kopetz, 1997] vorgestellt.
- Echtzeit-Scheduling (Ablaufplanung) wird umfassend in den Büchern von Buttazzo [Buttazzo, 2002] und Krishna und Shin [Krishna und Shin, 1997] abgehandelt.

- Die Vorlesungsskripte von Rajiv Gupta [Gupta, 2002] geben einen Überblick über eingebettete Systeme.
- Das Gebiet der Robotik ist eng mit den eingebetteten Systemen verwandt. Wir empfehlen für Informationen auf diesem Gebiet die Bücher von Fu, Gonzalez und Lee [Fu et al., 1987].
- Weitere Informationen findet man in einem Buch von Vahid [Vahid, 2002], in der ARTIST Roadmap [Bouyssounouse und Sifakis, 2005] sowie im forschungsorientierten *Embedded Systems Handbook* [Zurawski, 2006].

Die Struktur dieses Buches entspricht dem vereinfachten Informationsfluss beim Entwurf eingebetteter Systeme, wie er in Abb. 1.5 abgebildet ist.

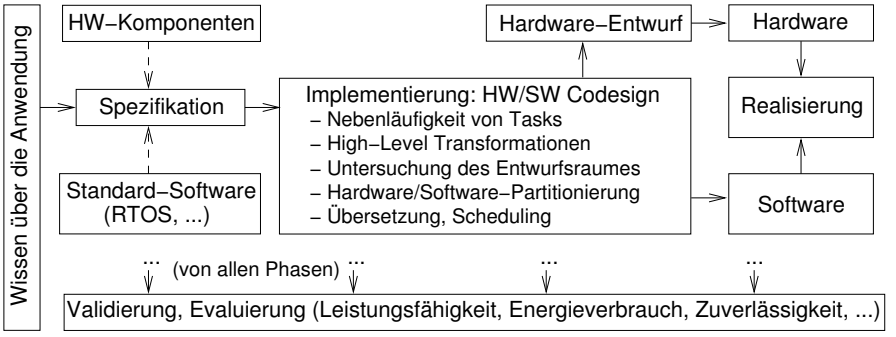


Abb. 1.5. Vereinfachter Informationsfluss beim Entwurf eingebetteter Systeme

Der Informationsfluss beginnt mit einer Idee. Diese Idee muss in einer Entwurfsspezifikation beschrieben werden. Hierbei sollten zur Verfügung stehende Hard- und Software-Komponenten wiederverwendet werden.

Der eigentliche Entwurf beginnt bei der Spezifikation. Typischerweise müssen beim Entwurf eingebetteter Systeme sowohl Hardware- als auch Software-Komponenten in Betracht gezogen werden. Die Entwurfsaktivitäten beinhalten das Abbilden von durchzuführenden Operationen auf nebenläufige Tasks, *High-Level*-Transformationen (etwa komplexe Schleifentransformationen), das Abbilden von Aufgaben auf entweder Hardware oder Software (die sogenannte Hardware/Software-Partitionierung), Untersuchen und Evaluieren des Entwurfsraumes, Software-Übersetzung mit Hilfe von Compilern, sowie Scheduling (Ablaufplanung). Eventuell ist es notwendig, spezialisierte Hardware oder optimierte Prozessorarchitekturen zu verwenden. Der Entwurf dedizierter Hardware wird in diesem Buch allerdings nicht behandelt. Zur Übersetzung der Software in die Maschinensprache können Standard-Compiler verwendet werden. Allerdings bieten diese häufig keine spezielle Unterstützung für eingebettete Systeme. Daher werden wir auch kurz auf Compiler-Techniken eingehen, mit denen die gewünschte Effizienz der generierten Software er-

reicht werden kann. Sobald der Maschinen-Code für jede Teilaufgabe generiert wurde, kann das Scheduling präzise durchgeführt werden. Schließlich werden Software- und Hardware-Beschreibungen zu einer kompletten Beschreibung des Systems zusammengefügt. Anhand dieser Beschreibung kann dann die Produktion des Systems erfolgen.

Nach momentanem Stand der Technik kann man für keinen dieser Schritte garantieren, dass er immer korrekte Ergebnisse liefert. Daher ist es notwendig, den Entwurf zu validieren. Die Validierung besteht darin, die Beschreibung von Zwischenschritten oder des endgültigen Designs mit den anderen Beschreibungen zu vergleichen und auf Abweichungen zu überprüfen. Außerdem muss zu verschiedenen Zeitpunkten der Entwurf evaluiert werden, z.B. in Bezug auf seine Leistungsfähigkeit, seine Verlässlichkeit, Energieverbrauch, Herstellbarkeit usw.

Zu beachten ist, dass Abb. 1.5 den Fluss von **Informationen über das Entwurfsobjekt** darstellt. Die Abfolge von **Entwurfsaktivitäten** muss zu diesem Fluss konsistent sein. Das bedeutet allerdings nicht, dass die Entwurfsaktivitäten einem einfachen Pfad von der Idee hin zum fertigen Produkt entsprechen. In der Praxis müssen einige Aktivitäten wiederholt werden. So kann es notwendig sein, zu einem späteren Zeitpunkt auf die Ebene der Spezifikation zurückzugehen, um zusätzliche Informationen über die Anwendung zu erhalten. Es kann auch vorkommen, dass andere Echtzeit-Betriebssysteme in Betracht gezogen werden müssen, da das ursprünglich vorgesehene Betriebssystem die Anforderungen nicht erfüllt.

Die Kapitel dieses Buches sind analog zum Informationsfluss strukturiert: in Kapitel 2 werden Spezifikationssprachen behandelt. Wichtige Hardware-Komponenten eingebetteter Systeme werden in Kapitel 3 betrachtet. Kapitel 4 widmet sich der Beschreibung von Echtzeit-Betriebssystemen und sogenannter *Middleware* sowie Scheduling-Techniken. Standard-Entwurfstechniken zur Implementierung eingebetteter Systeme, u.a. Compiler-Techniken, werden in Kapitel 5 behandelt. Das letzte Kapitel schließlich befasst sich mit der Evaluation und Validierung.

2

Spezifikationssprachen

2.1 Anforderungen

Gemäß dem vereinfachten Informationsfluss-Diagramm (s. Abb. 1.5) beschreibt dieses Kapitel die Anforderungen an Ansätze zur Spezifikation eingebetteter Systeme.

Es mag immer noch Fälle geben, in denen die Spezifikation eines eingebetteten Systems in einer natürlichen Sprache, wie z.B. Englisch, festgehalten wird. Dieser Ansatz ist aber vollkommen ungeeignet, denn ihm fehlen wichtige Anforderungen an Spezifikationssprachen: es muss möglich sein, eine Spezifikation auf Vollständigkeit sowie auf Widerspruchsfreiheit zu prüfen, außerdem möchte man aus der Spezifikation mit einer systematischen Vorgehensweise eine Implementierung des Systems herleiten können. Deshalb sollte die Spezifikation in Form einer maschinenlesbaren, formalen Sprache erfolgen. Spezifikationssprachen für eingebettete Systeme sollten in der Lage sein, die folgenden Sachverhalte darzustellen[1]:

- **Hierarchie:** Der Mensch ist im Allgemeinen nicht in der Lage, Systeme zu verstehen, die viele Objekte (wie z.B. Zustände, Komponenten) enthalten, die in komplexem Zusammenhang miteinander stehen. Um ein reales System zu beschreiben, benötigt man mehr Objekte als ein Mensch erfassen kann. Die Einführung von Hierarchie-Ebenen ist der einzige Weg, dieses Problem zu lösen. Hierarchie kann so eingesetzt werden, dass ein Mensch immer nur eine kleine Anzahl von Objekten auf einmal überschauen muss.

  Es gibt zwei Arten von Hierarchien:

  – **Verhaltens-Hierarchien:** Verhaltenshierarchien enthalten Objekte, die notwendig sind, um das Verhalten des Gesamtsystems zu beschrei-

[1] Zur Erstellung dieser Liste wurden Informationen aus den Büchern von Burns et al. [Burns und Wellings, 1990], Bergé et al. [Bergé et al., 1995] und Gajski et al. [Gajski et al., 1994] verwendet.

ben. Hierarchische Zustände und verschachtelte Prozeduren sind Beispiele solcher Hierarchien.

  - **Strukturelle Hierarchien:** Strukturelle Hierarchien beschreiben, wie das Gesamtsystem aus einzelnen physikalischen Komponenten zusammengesetzt ist.

    Eingebettete Systeme können z.B. aus Prozessoren, Speichern, Aktuatoren und Sensoren bestehen. Prozessoren wiederum enthalten Register, Multiplexer und Addierwerke. Multiplexer bestehen aus Gattern.

- **Zeitverhalten:** Da das Einhalten von Zeitbedingungen eine grundlegende Anforderung an eingebettete Systeme ist, muss es möglich sein, diese Zeitbedingungen explizit in der Spezifikation zu erfassen.
- **Zustandsorientiertes Verhalten:** Es wurde bereits in Kapitel 1 erwähnt, dass Automaten eine gute Darstellungsmöglichkeit für reaktive Systeme sind. Aus diesem Grund sollte es einfach sein, zustandsorientiertes Verhalten, wie es endliche Automaten zeigen, zu beschreiben. Klassische Automatenmodelle sind allerdings nicht ausreichend, da sie weder Zeitbedingungen noch Hierarchie unterstützen.
- **Ereignisbehandlung:** Da eingebettete Systeme oft reaktive Systeme sind, müssen Mechanismen zur Beschreibung von Ereignissen existieren. Solche Ereignisse können externe (von der Umwelt erzeugte) oder interne (von Komponenten des Systems erzeugte) Ereignisse sein.
- **Keine Hindernisse bei der Erzeugung von effizienten Implementierungen:** Da eingebettete Systeme effizient sein müssen, sollte die Spezifikationssprache eine effiziente Realisierung des Systems nicht behindern oder unmöglich machen.

  Beispiel: Angenommen, eine Spezifikationstechnik setze die Existenz einer virtuellen Maschine oder eines umfangreichen Simulators voraus. Dann müssen diese selbst bei kleinen Spezifikationen implementiert werden, was den Entwurf insgesamt nicht mehr effizient werden lassen kann.
- **Unterstützung für den Entwurf verlässlicher Systeme:** Spezifikationstechniken sollten den Entwurf von verlässlichen Systemen unterstützen. Beispielsweise sollte die Spezifikationssprache eine eindeutige Semantik haben und den Einsatz formaler Verifikationstechniken erlauben. Außerdem sollte es möglich sein, Sicherheits- und Authentizitäts-Anforderungen zu beschreiben.
- **Ausnahmeorientiertes Verhalten:** In vielen praktischen Systemen treten Ausnahmen auf. Um verlässliche Systeme entwerfen zu können, muss die Behandlung solcher Ausnahmesituationen einfach zu beschreiben sein. Es ist nicht ausreichend, wenn man die Ausnahmebehandlung z.B. für jeden einzelnen Zustand angeben muss, wie das etwa bei klassischen Automatenmodellen der Fall ist. Beispiel: In Abb. 2.1 soll die Eingabe `k` das Auftreten einer Ausnahme darstellen.

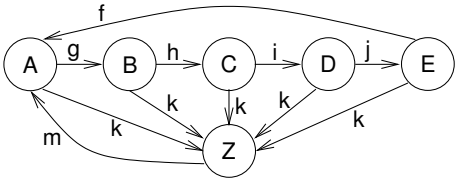


Abb. 2.1. Zustandsdiagramm mit Ausnahme k

Die Angabe einer solchen Ausnahme für jeden Zustand macht das Zustandsdiagramm sehr komplex und unübersichtlich, besonders bei großen Diagrammen mit vielen Transitionen. Wir werden später zeigen, wie man all diese Ausnahme-Transitionen durch eine einzige Transition ersetzen kann.

- **Nebenläufigkeit:** Viele praktisch relevante Systeme sind verteilte, nebenläufige Systeme. Aus diesem Grunde muss Nebenläufigkeit oder Parallelität einfach zu spezifizieren sein.
- **Synchronisation und Kommunikation:** Zeitgleich ablaufende Aktivitäten innerhalb eines Systems müssen miteinander kommunizieren und sich über die Verwendung von Ressourcen absprechen können. So ist es zum Beispiel häufig notwendig, gegenseitigen Ausschluss zu realisieren.
- **Programmiersprachenelemente:** Gewöhnliche Programmiersprachen haben sich als gängige Methode zur Beschreibung von Berechnungsvorschriften etabliert. Daher sollten Elemente von Programmiersprachen in der Spezifikation verwendet werden können. Klassische Zustandsdiagramme erfüllen diese Anforderung nicht.
- **Ausführbarkeit:** Eine Spezifikation ist nicht automatisch äquivalent zur ursprünglichen Idee des Designers. Das Ausführen der Spezifikation stellt eine Möglichkeit der Plausibilitätsprüfung für den Entwurf dar. Spezifikationen, die Programmiersprachenelemente verwenden, sind in diesem Kontext von Vorteil.
- **Unterstützung für den Entwurf großer Systeme:** Der Trend geht hin zu immer größeren und komplexeren Programmen, die auf eingebetteten Systemen ablaufen. In der Software-Technologie gibt es Mechanismen, um solche großen Systeme handhabbar zu machen, z.B. die Objektorientierung. Solche Mechanismen sollten auch in der Spezifikationsmethode Einsatz finden.
- **Unterstützung von spezifischen Anwendungsbereichen:** Es wäre natürlich schön, wenn man ein und dieselbe Spezifikationstechnik für alle möglichen eingebetteten Systeme verwenden könnte, da dies den Aufwand für die Entwicklung von Techniken minimieren würde. Allerdings ist die mögliche Bandbreite von Anwendungsgebieten so groß, dass man kaum hoffen kann, dass eine einzige Sprache alle Belange aller Anwen-

dungsbereiche gleichermaßen gut abdecken kann. Beispielsweise können kontrollflussdominierte, datenflussdominierte, zentralisierte oder verteilte Anwendungsgebiete von spezifischer *Tool*-Unterstützung für den jeweiligen Bereich profitieren.

- **Lesbarkeit:** Selbstverständlich muss eine Spezifikation von Menschen gelesen werden können. Nach Möglichkeit sollte sie auch maschinenlesbar sein, damit sie von einem Rechner verarbeitet werden kann.
- **Portierbarkeit und Flexibilität:** Spezifikationen sollten unabhängig von der für die Implementierung verwendeten spezifischen Hardware-Plattform sein, so dass man sie leicht für eine Auswahl von Zielplattformen einsetzen kann. Sie sollten so flexibel sein, dass eine kleine Änderung am Design auch nur eine kleine Änderung in der Spezifikation erfordert.
- **Terminierung:** Es sollte möglich sein, anhand der Spezifikation Prozesse zu identifizieren, die terminieren. Auf diese Weise wird erreicht, dass Prozessen, die keine Aufgaben mehr erfüllen, auch ihre Ressourcen entzogen werden können und dass man sich bei der Validierung mit diesen Prozessen nicht mehr beschäftigen muss.
- **Unterstützung für Nicht-Standard-Ein-/Ausgabe-Geräte:** Viele eingebettete Systeme verwenden andere Ein- und Ausgabegeräte als die vom PC her bekannten Tastaturen und Mäuse. Es sollte möglich sein, die Eigenschaften solcher Geräte in einfacher Weise zu spezifizieren.
- **Nicht-funktionale Eigenschaften:** Reale Systeme besitzen eine Reihe nicht-funktionaler Eigenschaften, wie etwa Ausfallsicherheit, Größe, Erweiterbarkeit, Lebenserwartung, Energieverbrauch, Gewicht, Recycling-Fähigkeit, Benutzerfreundlichkeit, elektromagnetische Verträglichkeit und so weiter. Es gibt keine Hoffnung, dass man all diese Eigenschaften in irgendeiner Weise formal definieren könnte.
- **Angemessenes Berechnungsmodell:** Zur Beschreibung von Berechnungen benötigt man ein Berechnungsmodell. Solche Modelle werden im nächsten Abschnitt behandelt.

An dieser Liste von Anforderungen kann man erkennen, dass es wohl nie eine einzige formale Sprache geben wird, die alle diese Anforderungen erfüllt. In der Praxis muss man daher in der Regel mit Kompromissen leben. Die Auswahl der Spezifikationssprache, die für ein bestimmtes Projekt verwendet wird, hängt hauptsächlich vom Anwendungsbereich und von der Umgebung, in der das System entwickelt werden soll, ab. Wir stellen im Folgenden einige Sprachen vor, die für den Entwurf eines praxisnahen Systems in Frage kommen.

2.2 Berechnungsmodelle

Die Anwendungen der Informationstechnologie waren bislang sehr stark an das Von-Neumann-Paradigma des sequentiellen Rechnens angelehnt. Dieses Paradigma ist für eingebettete Systeme, insbesondere für Realzeitsysteme, ungeeignet, da es beim Von-Neumann-Modell keine Möglichkeit gibt, Zeit auszudrücken. Andere Berechnungsmodelle sind für diesen Bereich eher geeignet.

Berechnungsmodelle definieren (in Anlehnung an [Lee, 1999]):

- **Komponenten** und die Organisation des Ablaufs von Berechnungen in diesen Komponenten: Prozeduren, Prozesse, Funktionen, endliche Automaten sind mögliche Komponenten.
- **Kommunikations-Protokolle**: Diese Protokolle beschreiben die Kommunikationsmöglichkeiten zwischen den Komponenten. Asynchroner Nachrichtenaustausch und *Rendez-Vous*-basierte Kommunikation sind Beispiele für solche Protokolle.

Beispiele für Berechnungsmodelle (siehe Lee [Lee, 1999], Janka [Janka, 2002] und Jantsch [Jantsch, 2003], die auch Nebenläufigkeit ausdrücken, können nach dem Modell der Kommunikation und dem Modell der Berechnungen in den Komponenten klassifiziert werden:

- **Modelle der Kommunikation von Komponenten**:
  - **Gemeinsamer Speicher** (*shared memory*)

    Bei diesem Modell greifen kommunizierende Komponenten auf gemeinsamen Speicher zu. Eine akzeptable Geschwindigkeit ergibt sich dabei nur für Komponenten, die sich in räumlicher Nähe zum Speicher befinden. Für verteilte Systeme ist dieses Modell also nicht geeignet.
  - **Nachrichtenaustausch** (*message passing*)

    Beim Nachrichtenaustausch kommunizieren Prozesse durch das Versenden von Nachrichten miteinander. Der Versand erfolgt über **Kanäle**. Wir unterscheiden zwischen asynchronem Nachrichtenaustausch und synchronem Nachrichtenaustausch.

    Beim **asynchronen Nachrichtenaustausch** muss der Absender nicht warten, bis der Empfänger bereit ist, die Nachricht zu empfangen. Er kann vielmehr in seinen Berechnungen fortfahren. Im Alltag entspricht das etwa dem Abschicken eines Briefes. In diesem Fall erfolgt der Versand über Kanäle, die in der Lage sind, Nachrichten zu puffern. Die meist notwendige Zwischenspeicherung von Nachrichten ist problematisch, da es zu Überläufen der Nachrichtenpuffer kommen kann.

    Bei **synchronem Nachrichtenaustausch** müssen Sender und Empfänger aufeinander warten (man spricht deshalb auch von der *rendez-*

vous-Technik). Kanäle müssen hierbei nicht puffern. Dafür ergibt sich dann allerdings häufig eine Leistungseinbuße in den Komponenten.

- **Modelle der Berechnungen in den Komponenten**
  - **Von Neumann-Modell**

    Als Modell der Berechnung in den Komponenten können wir z.B. das von-Neumann-Modell einer sequentiellen Abarbeitung eines Befehlsstroms benutzen. Hierbei wird die Existenz von abzuarbeitenden Maschinenbefehlen vorausgesetzt.
  - **Diskretes Ereignismodell**

    In diesem Modell tragen alle Ereignisse einen total geordneten Zeitstempel, der die Zeit angibt, zu der das Ereignis stattfindet. Simulatoren für diskrete Ereignismodelle verfügen über eine nach der Zeit sortierte globale Ereignis-Warteschlange. Beispiele sind u.a. VHDL (s. Seite 64) und Verilog (s. Seite 81).
  - **Endliche Automaten**

    Klassische endliche Automaten bilden ein weiteres Modell der Berechnung in Komponenten. Ihr Verhalten wird durch die üblichen Übergangs- und Ausgabefunktionen beschrieben.
- **Kombinierte Modelle**

  Manche Modelle verbinden ein bestimmtes Modell der Berechnung in den Komponenten mit einem Modell der Kommunikation. So kombiniert SDL (s. Seite 32) beispielsweise das Modell eines endlichen Automaten für die Berechnung in den Komponenten mit dem Modell des asynchronen Nachrichtenaustauschs für die Kommunikation. StateCharts dagegen kombiniert das Modell des endlichen Automaten für die einzelne Komponente mit dem Modell des gemeinsamen Speichers für die Kommunikation. CSP (s. Seite 60) und ADA (s. Seite 61) kombinieren das Modell der von-Neumann-Befehlsausführung mit dem synchronen Nachrichtenaustausch.

Verschiedene Anwendungen können die Verwendung unterschiedlicher Modelle notwendig machen. Während einige der verfügbaren Spezifikationssprachen nur eines der Modelle implementieren, erlauben andere eine Mischung verschiedener Modelle. Für die Mehrzahl der Modelle werden nachfolgend exemplarisch Beispielsprachen vorgestellt.

2.3 StateCharts

Die erste praktisch eingesetzte Sprache, die hier vorgestellt werden soll, ist StateCharts. StateCharts wurde 1987 von David Harel [Harel, 1987] vorgestellt und später detaillierter beschrieben [Drusinsky und Harel, 1989]. StateCharts beschreibt kommunizierende endliche Automaten, basierend auf dem

shared memory Kommunikations-Konzept. Den Namen hat Harel angeblich so gewählt, weil es „*die einzige unbenutzte Kombination von 'flow' oder 'state' mit 'diagram' oder 'chart'* " war.

Im Abschnitt 2.1 wurde erwähnt, dass es oft notwendig ist, zustandsorientiertes Verhalten zu modellieren. Zustandsdiagramme sind dafür die klassische Methode. Abb. 2.2 (identisch mit Abb. 2.1) zeigt ein Beispiel für ein klassisches Zustandsdiagramm, das einen **endlichen Automaten** darstellt.

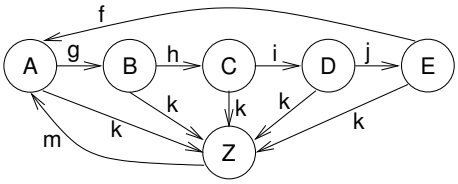


Abb. 2.2. Zustandsdiagramm

Kreise bezeichnen Zustände. **Deterministische** endliche Automaten, die wir hier betrachten wollen, können sich immer nur in einem ihrer Zustände befinden. Kanten stellen die Transitionen oder Übergänge zwischen den Zuständen dar. Die Kantenbeschriftung bezeichnet Ereignisse. Wenn eine Eingabe stattfindet, ändert der endliche Automat seinen Zustand der jeweiligen Kante entsprechend. Endliche Automaten können auch eine Ausgabe erzeugen (dies ist in Abb. 2.2 nicht gezeigt). Nähere Informationen zu klassischen endlichen Automaten findet man z.B. in [Kohavi, 1987].

2.3.1 Modellierung von Hierarchie

StateCharts beschreiben erweiterte endliche Automaten und sind daher gut geeignet, zustandsorientiertes Verhalten abzubilden. Die wichtigste Erweiterung gegenüber klassischen Automaten ist das Konzept der **Hierarchie**. Hierarchie wird mit Hilfe von **Superzuständen** (*superstates*) eingeführt.

Definitionen:

- Zustände, die andere Zustände enthalten, heißen **Superzustände**
- Zustände, die in anderen Zuständen enthalten sind, heißen **Unterzustände** der Superzustände

Abb. 2.3 zeigt ein StateCharts-Beispiel. Es ist eine hierarchische Version von Abb. 2.2.

Superzustand S beinhaltet die Zustände A, B, C, D und E. Angenommen, der Automat befindet sich in Zustand Z (wir bezeichnen Z in diesem Fall auch als **aktiven Zustand**). Wenn dann die Eingabe m am Automaten erfolgt, ist A

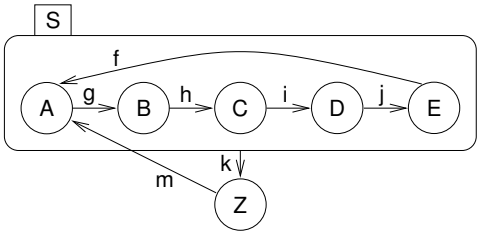


Abb. 2.3. Hierarchisches Zustandsdiagramm

der nächste Zustand. Wenn sich der Automat in Zustand S befindet und es gibt die Eingabe k, so ist Z der nächste Zustand, unabhängig davon, in welchem der Unterzustände A, B, C, D oder E sich der Automat tatsächlich befindet. In diesem Beispiel sind alle in S enthaltenen Zustände nicht-hierarchische Zustände. Im Allgemeinen könnten die Unterzustände von S selbst wieder Superzustände sein, die weitere Unterzustände enthalten.

Definitionen:

- Jeder Zustand, der nicht aus anderen Zuständen besteht, heißt **Basiszustand**.
- Ist t ein Unterzustand von s, so heißt s **Oberzustand** von t.

Der endliche Automat in Abb. 2.3 kann sich zu einem bestimmten Zeitpunkt nur in genau einem der Unterzustände des Superzustands S befinden. Solche Superzustände heißen **ODER-Superzustände**. Gemeint ist hier immer ein sich gegenseitig ausschließendes ODER, da sich der Automat nur in **einem** der Zustände A, B, C, D oder E befinden kann.

Definition: Superzustände S heißen **ODER-Superzustände**, wenn das System, das S enthält, sich zu jedem Zeitpunkt lediglich in einem einzigen Unterzustand von S befinden kann, solange es sich in S befindet.

In Abb. 2.3 könnte die Eingabe k einer Ausnahme entsprechen, wegen der Zustand S verlassen werden muss. Das Beispiel zeigt bereits, wie man solche Ausnahmebehandlungen durch Verwendung von Hierarchie kompakt darstellen kann.

StateCharts erlaubt es, Systeme hierarchisch zu beschreiben, indem eine Beschreibung des Systems Untersysteme enthält, die wiederum Beschreibungen von Unterzuständen enthalten können und so weiter. Das Gesamtsystem kann somit in Form eines **Baumes** dargestellt werden. Die **Wurzel** dieses Baumes entspricht dem gesamten System, und die inneren Knoten entsprechen hierarchischen Beschreibungen (die im Falle von StateCharts Superzustände heißen). Die **Blätter** der Hierarchie sind nicht-hierarchische Beschreibungen (die im Falle von StateCharts Basiszustände heißen).

Bisher haben wir explizite, direkte Kanten verwendet, die zu Basiszuständen führen, um den neuen Zustand zu bestimmen. Der Nachteil dieser Art der Modellierung liegt in der Tatsache, dass man die internen Strukturen der Superzustände nicht vor der Umgebung verstecken kann. In einem echten hierarchischen Modell sollte es möglich sein, die interne Struktur zu verstecken, so dass diese später beschrieben oder sogar geändert werden kann, ohne eine Auswirkung auf die Umgebung zu haben. Dies wird durch zusätzliche Mechanismen ermöglicht, die den neuen Zustand bestimmen.

Der erste zusätzliche Mechanismus ist der **Standardzustand** (*default state*). Er kann in Superzuständen verwendet werden, um anzuzeigen, welche der Unterzustände betreten werden, wenn der Superzustand betreten wird. In den Diagrammen wird der Standardzustand mit Hilfe einer Kante bezeichnet, die von einem kleinen ausgefüllten Kreis zum jeweiligen Zustand geht. Abb. 2.4 zeigt ein Zustandsdiagramm, das den Standardzustands-Mechanismus verwendet. Es ist äquivalent zum Diagramm in Abb. 2.3. Man beachte, dass der kleine ausgefüllte Kreis selber keinen Zustand darstellt.

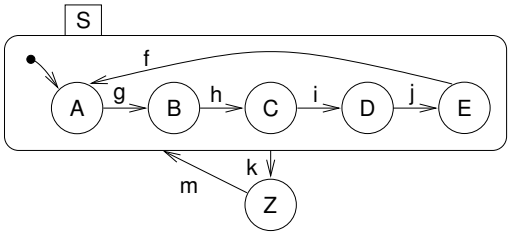


Abb. 2.4. Zustandsdiagramm mit Standardzustand

Ein weiterer Mechanismus, um den nächsten Zustand anzugeben, ist der sogenannte ***History*-Zustand**. Mit Hilfe dieses Konstrukts ist es möglich, in den letzten Unterzustand zurückzukehren, der aktiv war, bevor der Superzustand verlassen wurde. Der *History*-Mechanismus wird durch den Buchstaben H in einem Kreis dargestellt. Um den aktiven Zustand für den ersten Wechsel in einen Superzustand zu kennzeichnen, wird der *History*-Mechanismus oft mit einem Standardzustand kombiniert. Abb. 2.5 zeigt ein Beispiel.

Das Verhalten des endlichen Automaten hat sich nun verändert: Sei der Automat zunächst im Zustand Z, und es ereigne sich die Eingabe m. Dann ist A der nächste Zustand, wenn der Superzustand S zum ersten Mal betreten wird. Ansonsten wird der zuletzt aktive Unterzustand betreten. Dieser Mechanismus hat viele Anwendungen. Wenn z.B. die Eingabe k eine Ausnahme darstellt, könnte die Eingabe m verwendet werden, um in den Zustand vor der Ausnahme zurückzukehren. Die Zustände A, B, C, D und E könnten Zustand Z auch wie eine Prozedur aufrufen. Nachdem diese „Prozedur“ Z abgearbeitet ist, kehrt der Automat zum aufrufenden Zustand zurück.

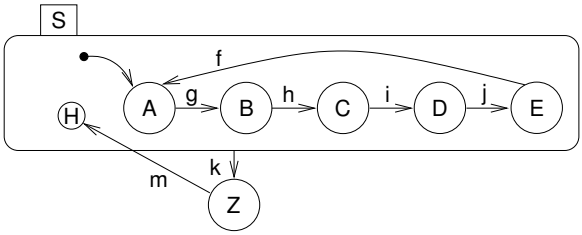


Abb. 2.5. Zustandsdiagramm mit *History*-Mechanismus und Standardzustand

Der Automat aus Abb. 2.5 kann auch wie in Abb. 2.6 dargestellt werden. In diesem Fall wurden die Darstellungen für den Standardzustand und den *History*-Mechanismus kombiniert.

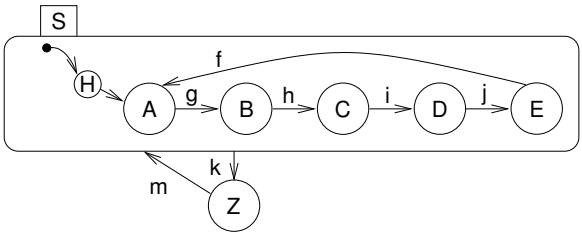


Abb. 2.6. Kombination der Symbole für *History*- und Standardzustand

Eine Spezifikationstechnik muss auch in der Lage sein, Nebenläufigkeit und Parallelität darzustellen. Zu diesem Zweck gibt es in StateCharts eine zweite Art von Superzuständen, die sogenannten **UND-Superzustände**.

Definition: Superzustände S heißen **UND-Superzustände**, wenn das System, das S enthält, sich in allen Unterzuständen von S gleichzeitig befindet, solange es sich in S befindet.

Das Diagramm für einen Anrufbeantworter in Abb. 2.7 enthält einen UND-Superzustand.

Ein Anrufbeantworter muss normalerweise zwei Aufgaben parallel ausführen: er wartet auf ankommende Anrufe und überprüft gleichzeitig, ob der Benutzer etwas auf den Eingabetasten eingegeben hat. In Abb. 2.7 heißen die entsprechenden Zustände Lwait und Kwait. Ankommende Anrufe werden im Zustand Lproc abgearbeitet, während Reaktionen auf die Eingabetasten im Zustand Kproc erzeugt werden. Der Ein-/Aus-Schalter wird fürs Erste so modelliert, dass die von ihm erzeugten Ereignisse (Einschalten und Ausschalten) separat dekodiert werden und somit nicht in den Zustand Kproc gewechselt wird. Wird der Anrufbeantworter ausgeschaltet, so werden die beiden im ein-Zustand enthaltenen Zustände verlassen. Sie werden erst wieder betreten, wenn der Anrufbeantworter wieder eingeschaltet wird. Zu diesem Zeitpunkt werden dann die

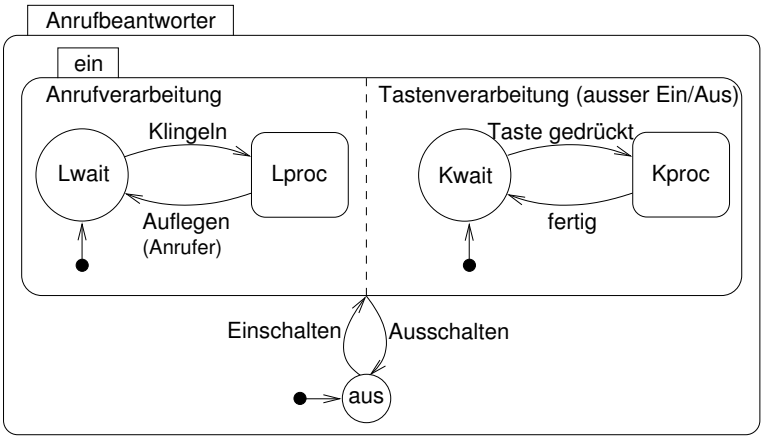


Abb. 2.7. Anrufbeantworter

Standardzustände Lwait und Kwait betreten. Solange die Maschine eingeschaltet ist, befindet sie sich immer in den beiden Unterzuständen des Zustands ein.

Für UND-Superzustände können die Unterzustände, die als Reaktion auf ein Ereignis betreten werden, unabhängig voneinander beschrieben werden. Jede beliebige Kombination von *History*-, Standardzuständen und expliziten Transitionen ist möglich. Für das Verständnis von StateCharts ist es wichtig zu begreifen, dass immer **alle** Unterzustände betreten werden, auch wenn es nur eine einzige explizite Transition zu einem der Unterzustände gibt. Analog dazu gilt, dass eine Transition aus einem UND-Superzustand heraus immer dazu führt, dass **alle** seine Unterzustände verlassen werden.

Als Beispiel modifizieren wir unseren Anrufbeantworter so, dass der Ein-/Aus-Schalter wie alle anderen Bedientasten im Zustand Kproc dekodiert und behandelt wird (s. Abb. 2.8).

Wenn der Anrufbeantworter ausgeschaltet wird, findet eine Transition in den aus-Zustand statt. Dieser Übergang führt dazu, dass auch der Zustand, der auf ankommende Anrufe wartet, verlassen wird. Das Wiedereinschalten der Maschine führt dazu, dass eben dieser Zustand auch wieder mit betreten wird.

Zusammenfassend können wir festhalten: **Zustände in StateCharts-Diagrammen sind entweder UND-Superzustände, ODER-Superzustände oder Basiszustände.**

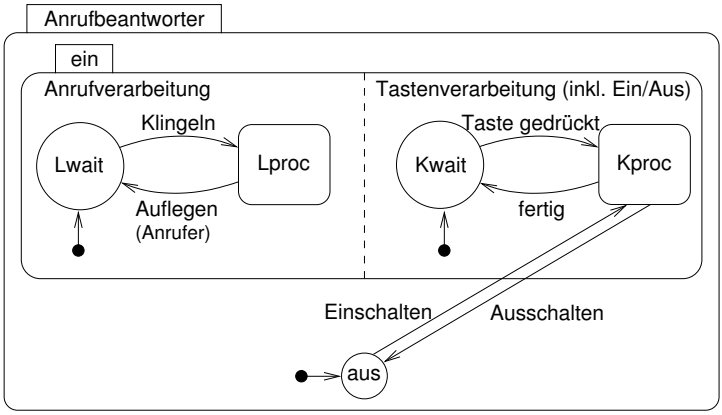


Abb. 2.8. Anrufbeantworter mit veränderter Ein-/Ausschalter-Modellierung

2.3.2 Zeitbedingungen

Da es notwendig ist, in eingebetteten Systemen Zeitbedingungen zu modellieren, bietet StateCharts die sogenannten *Timer* an. Zeitbedingungen werden durch das gezackte Symbol im linken Teil von Abb. 2.9 dargestellt.

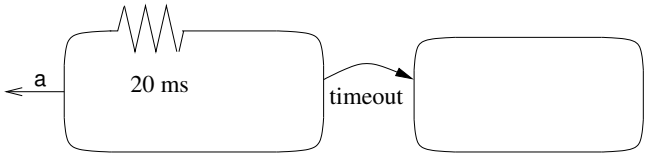


Abb. 2.9. Zeitbedingungen in StateCharts

Wenn das System für die im *Timer* angegebene Zeitdauer im *Timer*-Zustand war, wird ein *Time-Out* ausgelöst und das System verlässt diesen Zustand. *Timer* können auch hierarchisch verwendet werden.

Ein *Timer* könnte beispielsweise in einer tieferen Hierarchiestufe des Anrufbeantworters verwendet werden, um das Verhalten des Zustands Lproc zu beschreiben. Abb. 2.10 zeigt eine mögliche Beschreibung für diesen Zustand.

Da das Auflegen des Anrufers in Abb. 2.7 in Form einer Ausnahmebehandlung realisiert ist, wird der Zustand Lproc immer erst dann verlassen, wenn der Anrufer auflegt. Wenn allerdings der Angerufene auflegt, hat der Entwurf des Zustands Lproc einen Schönheitsfehler: wenn der Angerufene zuerst auflegt, ist das Telefon solange tot (und still), bis der Anrufer ebenfalls aufgelegt hat.

StateCharts beinhalten noch weitere Sprachelemente, die beispielsweise im Buch von Harel [Harel, 1987] zu finden sind. Zusammen mit Drusinsky gibt

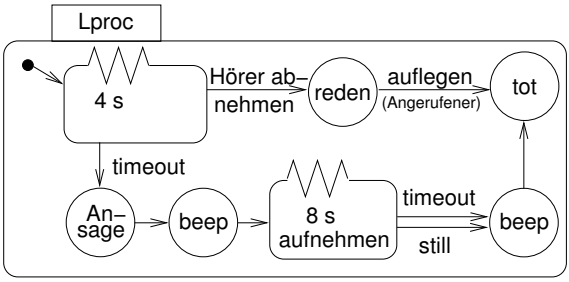


Abb. 2.10. Behandlung von eingehenden Anrufen in Lproc

Harel [Drusinsky und Harel, 1989] in einem Artikel eine genauere Beschreibung der Semantik von StateMate, einer StateCharts-Implementierung.

2.3.3 Kantenbeschriftungen und StateMate-Semantik

Die Ausgabe der erweiterten Automatenmodelle wurde bislang noch nicht betrachtet. Solche Ausgaben können mit Hilfe von Kantenbeschriftungen realisiert werden. Die allgemeine Form einer Kantenbeschriftung ist „Ereignis [Bedingung] / Reaktion“. Alle drei Bestandteile der Beschriftung sind optional. Die **Reaktion** beschreibt die Reaktion des Automaten auf den Zustandsübergang. Mögliche Reaktionen beinhalten das Erzeugen von Ereignissen oder die Zuweisung von Variablenwerten. Die Anteile Bedingung und Ereignis beschreiben zusammen die Eingaben an den Automaten. Die **Bedingung** beschreibt das Überprüfen von Werten von Variablen oder des Zustands des Gesamtsystems. Der **Ereignis**teil symbolisiert, auf welches Ereignis zu prüfen ist. Solche Ereignisse können entweder intern oder extern erzeugt werden. Interne Ereignisse werden als Ergebnis von Zustandsübergängen erzeugt und werden in der Reaktion des Übergangs beschrieben. Externe Ereignisse werden üblicherweise in der Systemumgebung beschrieben.

Beispiele:

- Einschalten / Ein:=1 (Test auf ein Ereignis und Wertzuweisung an eine Variable),
- [Ein=1] (Überprüfung eines Variablenwerts),
- Ausschalten [not in Lproc] / Ein:=0 (Ereignistest, Überprüfung eines Zustands, Variablenzuweisung. Die Zuweisung wird nur durchgeführt, wenn das Ereignis stattgefunden hat und die Bedingung erfüllt ist).

Die Menge aller Variablenwerte zusammen mit der Menge von erzeugten Ereignissen (und die aktuelle Zeit) ist definiert als der **Status**[2] eines StateCharts-

[2] Normalerweise würde man den Begriff „Zustand“ statt „Status“ verwenden, aber der Begriff „Zustand“ hat in StateCharts eine andere Bedeutung.

Modells. Ereignisse werden, wie bereits erwähnt, entweder intern oder extern erzeugt.

Die Semantik der Kantenbeschriftungen kann nur im Zusammenhang mit der generellen Semantik von StateCharts erklärt werden. Die StateMate-Implementierung von StateCharts [Drusinsky und Harel, 1989] geht von einer schrittweisen Ausführung von StateCharts-Beschreibungen aus. Abhängig vom aktuellen Status werden Übergänge von einem Status zum nächsten ausgelöst. Diese Übergänge werden als ein **Schritt** bezeichnet (s. Abb. 2.11).

Abb. 2.11. Schritte während der Ausführung eines StateCharts-Modells

Jeder Schritt besteht aus drei Phasen:

1. In der ersten Phase wird der Einfluss von externen Bedingungen und Ereignissen ausgewertet. Das beinhaltet auch die Auswertung von Funktionen, die von externen Ereignissen abhängen. In dieser Phase gibt es keine Zustandsübergänge. In unseren einfachen Beispielen wird diese Phase nicht unbedingt benötigt.
2. Die nächste Phase besteht darin, die Menge der Zustandsübergänge zu bestimmten, die im aktuellen Schritt ausgeführt werden sollen. Variablenzuweisungen werden ausgewertet, aber die neuen Werte werden nur temporären Zwischenvariablen zugewiesen.
3. In der dritten Phase finden die Zustandsübergänge statt und Variablen erhalten ihre neuen Werte.

Die Aufteilung in die Phasen 2 und 3 ist besonders wichtig, um ein deterministisches und reproduzierbares Verhalten von StateCharts-Modellen zu erreichen. Als Beispiel betrachten wir das StateCharts-Modell in Abb. 2.12.

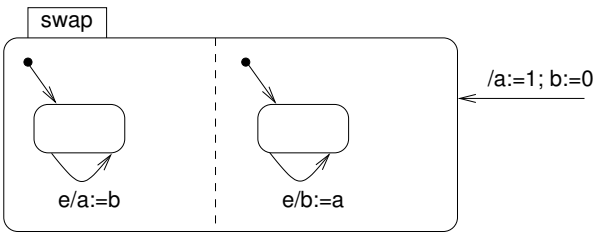


Abb. 2.12. Gegenseitige, abhängige Wertzuweisung

Wegen der Aufteilung in zwei Phasen werden bei Eintreten des Ereignisses `e` die neuen Werte für `a` und `b` zuerst in temporären Variablen, z.B. `a'` und `b'`,

zwischengespeichert. In der letzten Phase werden diese Zwischenwerte dann in die eigentlichen Variablen kopiert:

Phase 2: a':=b; b':=a;

Phase 3: a:=a'; b:=b'

Dadurch werden die Werte der beiden Variablen jedesmal vertauscht, wenn das Ereignis e eintritt. Dieses Verhalten entspricht zwei über Kreuz verbundenen Registern (eines für jede Variable), die an der gleichen Taktleitung angeschlossen sind (s. Abb. 2.3.3) und modelliert das Verhalten eines getakteten Schaltwerks, das diese zwei Register enthält.

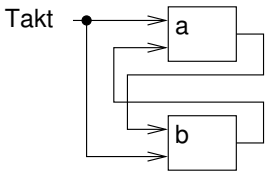


Abb. 2.13. über Kreuz verbundene Register

Ohne die Aufteilung in zwei Phasen würde das Ergebnis von der Reihenfolge der Ausführung der Wertzuweisungen abhängen. In jedem Fall würde aber beiden Variablen der gleiche Wert zugewiesen werden. Diese Aufteilung in (mindestens) zwei Phasen ist typisch für Sprachen, die das Verhalten von synchroner Hardware nachbilden. Eine ähnliche Aufteilung findet man auch in VHDL (s. Seite 80).

Die gerade beschriebenen drei Phasen werden jeweils in jedem **Schritt** ausgeführt. Schritte werden jedesmal ausgeführt, wenn ein Ereignis eintritt oder sich ein Variablenwert geändert hat. Die Ausführung eines StateCharts-Modells besteht aus einer Folge von Schritten (s. Abb. 2.11), wobei jeder Schritt aus den beschriebenen drei Phasen besteht.

Das Konzept der Schritte erlaubt es, die Semantik von **Ereignissen** genauer zu definieren. **Die Sichtbarkeit von Ereignissen ist auf denjenigen Schritt beschränkt, der auf den Schritt folgt, in dem das Ereignis erzeugt wurde.** Somit verhalten sich Ereignisse wie einzelne Bitwerte, die bei einer Taktflanke in Registern abgelegt werden und einen Einfluss auf die bei der nächsten Taktflanke gespeicherten Werte haben. Ereignisse „leben“ nur bis zum nächsten Schritt bzw. bis zur nächsten Taktflanke.

Im Gegensatz dazu behalten Variablen ihren Wert, bis ihnen ein neuer Wert zugewiesen wird. Nach der StateMate-Semantik sind neue Werte von Variablen in allen Teilen des Modells sichtbar, und zwar ab demjenigen Schritt, der auf den Schritt folgt, in dem die Wertzuweisung erfolgt ist. Das bedeutet, dass die Semantik von StateMate neue Variablenwerte zwischen zwei Schritten überall im Modell propagiert und sichtbar macht. Implizit geht StateCharts

gemäß StateMate-Semantik also von einem *Broadcast*-Mechanismus zum Aktualisieren von Variablenwerten aus. Für verteilte Systeme ist es sehr schwierig, alle Variablenwerte zwischen zwei Schritten zu aktualisieren. Aufgrund dieses *Broadcast*-Mechanismus sind StateCharts mit der beschriebenen Semantik nicht geeignet, um verteilte Systeme zu beschreiben.

2.3.4 Bewertung und Erweiterungen

Das Hauptanwendungsgebiet von StateCharts sind lokale, kontrolldominierte Systeme. Die Möglichkeit, Hierarchie-Ebenen beliebig zu verschachteln und mit frei wählbaren UND- und ODER-Zuständen zu versehen, ist der Hauptvorteil von StateCharts. Ein weiterer Vorteil ist die ausreichend präzise Definition der Semantik von StateCharts [Drusinsky und Harel, 1989] gemäß StateMate-Implementierung. Außerdem ist eine Vielzahl von kommerziellen Programmen erhältlich, die StateCharts verwenden. StateMate[3] und StateFlow[4] sind Beispiele für solche kommerziellen Anwendungen. Viele dieser Programme sind in der Lage, aus StateCharts-Modellen äquivalente Beschreibungen in C oder VHDL (s. Seite 64) zu erzeugen. Aus VHDL kann mit Hilfe von Synthese-Tools direkt Hardware erzeugt werden. Aus diesem Grund bieten Programme, die auf StateCharts basieren, einen vollständigen Pfad von der StateCharts-Beschreibung bis zur fertigen Hardware. Generierte C-Programme können übersetzt und ausgeführt werden, womit auch ein Weg zu Software-Realisierungen existiert.

Leider ist die Effizienz dieser automatisch erzeugten Realisierungen häufig ein Problem. So können in einer automatisch erzeugten Software etwa Unterzustände von UND-Zuständen auf Unix-Prozesse abgebildet werden. Dieses Konzept ist für die Simulation von StateCharts geeignet, nicht jedoch für eine effiziente Realisierung auf kleinen, leistungsschwachen Prozessoren. Der Produktivitätsgewinn durch objektorientierte Programmierung kann mit StateCharts nicht ausgeschöpft werden, da es keine Objektorientierung unterstützt. Außerdem ist es durch den *Broadcast*-Mechanismus kaum für verteilte Anwendungen geeignet. StateCharts unterstützen keine Programmiersprachen-Konstrukte, um komplexe Berechnungsvorschriften zu realisieren. Außerdem können sie keine Hardware-Strukturen oder nicht-funktionales Verhalten beschreiben.

Kommerzielle Implementierungen von StateCharts bieten üblicherweise einige Mechanismen, um die Nachteile von StateCharts zu umgehen. So kann etwa C-Code verwendet werden, um Programmiersprachenkonstrukte zu unterstützen. In StateMate stellen die sogenannten *Module Charts* Hardwarestrukturen dar.

[3] http://www.ilogix.com
[4] http://www.mathworks.com/products/stateflow

2.4 Allgemeine Spracheigenschaften

Der vorhergehende Abschnitt hat Beispiele für Eigenschaften von Spezifikationssprachen für eingebettete Systeme gezeigt. Diese Beispiele sind hilfreich, um in diesem Abschnitt einige allgemeine Charakteristiken von Sprachen zu besprechen, bevor in den nachfolgenden Kapiteln weitere Sprachen vorgestellt werden. Es gibt einige Unterscheidungsmerkmale, nach denen wir verschiedene Eigenschaften von Sprachen vergleichen können. Die erste Eigenschaft hängt mit der Unterscheidung zwischen deterministischen und nichtdeterministischen Modellen zusammen, die bereits in der Besprechung von StateCharts erwähnt wurde.

2.4.1 Synchrone und asynchrone Sprachen

Einige Sprachen verwenden als Grundlage kommunizierende endliche Automaten oder Mengen von kommunizierenden Prozessen, die in ADA oder in Java beschrieben werden. Diese Sprachen haben das Problem, dass sie nicht deterministisch sind, da die Reihenfolge der Ausführung von Prozessen nicht spezifiziert ist. Die Ausführungsreihenfolge kann einen Einfluss auf das Ergebnis einer Berechnung haben. Dieser Effekt hat einige negative Auswirkungen, etwa bei der Validierung eines Entwurfs. Dieser **Nicht-Determinismus** kann durch die Verwendung von sogenannten synchronen Sprachen vermieden werden. Synchrone Sprachen beschreiben nebenläufige Automaten. „*... wenn Automaten nebenläufig arbeiten, ist ein Zustandswechsel des Gesamtsystems zusammengesetzt aus 'gleichzeitigen' Zustandsübergängen aller beteiligter Automaten*“ [Halbwachs, 1998]. Das bedeutet, dass man nicht alle möglichen Abfolgen von Zustandsübergängen aller Automaten betrachten muss, wie dies bei unabhängigen Takten in den Unterautomaten der Fall wäre. Stattdessen kann man die Existenz eines einzigen globalen Taktes annehmen. Bei jedem Taktsignal werden alle Eingaben berücksichtigt, neue Ausgaben und Zustände werden berechnet und die entsprechenden Zustandsübergänge ausgeführt. Dazu ist ein schneller *Broadcast*-Mechanismus notwendig, der alle Teile des Modells erreicht. Diese idealisierte Betrachtung von Gleichzeitigkeit hat den Vorteil, dass dadurch **deterministisches Verhalten** garantiert wird. Es stellt eine Einschränkung des allgemeinen Modells kommunizierender Automaten dar, bei dem jeder Automat seinen eigenen Takt haben darf. Synchrone Sprachen stellen das Prinzip der Gleichzeitigkeit in synchroner Hardware dar und repräsentieren die Semantik von Sprachen für Industriesteuerungen wie IEC 60848 und STEP 7 (s. Seite 85). Ein deterministisches Verhalten für alle Sprachkonstrukte zu garantieren war eines der Hauptziele bei der Entwicklung der synchronen Sprachen Esterel (s. Seite 86) [Esterel, 2006] und Lustre [Halbwachs et al., 1991]. Aufgrund der drei Simulationsphasen ist auch StateCharts mit StateMate-Semantik eine synchrone Sprache (und entsprechend deterministisch). Genau wie StateCharts sind synchrone Sprachen eher

schlecht für die Modellierung von verteilten Systemen geeignet, da das Konzept eines einzigen, globalen Taktes bei solchen Systemen einige Schwierigkeiten bereitet.

2.4.2 Prozess-Konzepte

Man kann das Prozess-Konzept verschiedener Programmiersprachen anhand einiger Kriterien vergleichen:

- Die **Anzahl der Prozesse** kann entweder **statisch** oder **dynamisch** sein. Eine statische Anzahl von Prozessen vereinfacht die Implementierung und ist ausreichend, wenn jeder Prozess ein Stück Hardware modelliert und wenn man *„Hotplugging“* (das dynamische Verändern der Hardware) außer Acht lässt. In allen anderen Fällen sollte die dynamische Erzeugung (und Entsorgung) von Prozessen betrachtet werden.
- Prozesse können entweder statisch **verschachtelt** oder alle auf der gleichen Ebene deklariert werden. StateCharts erlaubt zum Beispiel verschachtelte Prozessdeklarationen, während SDL (s. Seite 32) dies nicht unterstützt. Durch die Verschachtelung wird auch eine Kapselung ermöglicht.
- Es gibt verschiedene Techniken zur **Prozesserzeugung**. Das Erzeugen eines Prozesses kann durch die Abarbeitung einer Prozessdeklaration im Quelltext der Anwendung, durch *„fork“* und *„join“*-Operationen des Betriebssystems (wie etwa in Unix) oder durch explizite Prozesserzeugungs-Aufrufe realisiert werden.

StateCharts unterstützt nur eine statische Anzahl von Prozessen. Prozesse können verschachtelt werden. Die Erzeugung der Prozesse geschieht durch die Abarbeitung des Quelltextes.

2.4.3 Synchronisation und Kommunkation

Es gibt zwei grundsätzliche Kommunikations-Paradigmen: gemeinsamer Speicher (***shared memory***) und **Nachrichtenaustausch**.

Bei Verwendung von *shared memory* können alle Prozesse auf Variablen zugreifen. Der Zugriff auf einen gemeinsamen Speicher sollte geschützt werden, wenn auch Schreibzugriffe erlaubt werden sollen. In diesem Fall muss der exklusive Zugriff garantiert werden, wenn mehrere Prozesse auf den gemeinsamen Speicher zugreifen. Codesegmente, für die ein exklusiver Zugriff garantiert werden muss, heißen **kritische Abschnitte**. Es gibt verschiedene Mechanismen, um den exklusiven Zugriff auf Ressourcen zu garantieren, darunter Semaphore, bedingte kritische Regionen und Monitore. Genauere Informationen findet man in der Literatur zu Betriebssystemen. Kommunikation

auf der Basis von *shared memory* kann sehr schnell sein, ist aber insbesondere bei Multiprozessor-Systemen, die keinen gemeinsamen physikalischen Speicher haben, schwierig umzusetzen.

Beim Nachrichtenaustausch werden Nachrichten ähnlich wie E-Mails im Internet versendet und empfangen. Der Austausch von Nachrichten kann leicht implementiert werden, auch wenn kein gemeinsamer Speicherbereich zur Verfügung steht. Allerdings ist die Kommunikation durch Nachrichtenaustausch in der Regel langsamer als bei Verwendung von *Shared Memory*. Beim Nachrichtenaustausch kann man grundsätzlich die folgenden drei Techniken unterscheiden:

- **Asynchroner Nachrichtenaustausch** oder **nicht-blockierende Kommunikation** (s. Seite 17),
- **Synchroner Nachrichtenaustausch**, auch **blockierende, *Rendez-Vous*-basierte Kommunikation** genannt (s. Seite 17),
- **Erweitertes *Rendez-Vous*:** der Sender darf erst weiterarbeiten, wenn er eine Bestätigungsnachricht vom Empfänger erhalten hat. Der Empfänger einer Nachricht muss diese Bestätigung nicht sofort nach dem Erhalt der Nachricht abschicken, sondern kann z.B. den Typ der Nachricht überprüfen, bevor er die Bestätigung schickt.

StateCharts erlaubt die Verwendung von globalen Variablen und benutzt folglich das *shared memory* Modell.

2.4.4 Spezifikation von Zeitbedingungen

Burns und Wellings [Burns und Wellings, 1990] definieren die folgenden vier Anforderungen für die Angabe von Zeiten in Spezifikationssprachen:

- Zugriff auf einen *Timer*, der die Möglichkeit bietet, die **vergangene Zeit zu messen**:

  Die Sprache CSP (s. Seite 60) erfüllt diese Anforderung beispielsweise durch Kanäle, die eigentlich *Timer* sind. Leseoperationen auf solchen Kanälen liefern die aktuelle Zeit.

- Eine Möglichkeit, **Prozesse um eine bestimmte Zeit zu verzögern**:

  Echtzeitsprachen stellen üblicherweise ein Konstrukt zur Verzögerung zur Verfügung. In VHDL kann etwa die wait for-Anweisung (s. Seite 76) verwendet werden.

- Eine Möglichkeit, ***Timeouts*** zu spezifizieren:

  Die meisten Echtzeitsprachen haben ein *Timeout*-Sprachelement.

- Methoden, um **Zeitschranken** und **Abläufe (Schedules)** anzugeben:

Leider erlauben es die meisten Sprachen nicht, Zeitschranken anzugeben. Wenn sie angegeben werden können, dann zumeist in separaten Steuerungs-Dateien oder in Dialogfenstern.

StateCharts unterstützt die Angabe von *Timeouts*. Andere Zeitbedingungen können nicht ohne weiteres angegeben werden.

2.4.5 Nicht-Standard-Ein-/Ausgabegeräte

Einige Sprachen bieten spezielle Funktionen an, um Ein- und Ausgabegeräte direkt ansprechen zu können. In ADA kann man Variablen festen Speicheradressen zuordnen. Diese Adressen können auch von externen Geräten verwendet werden, die über *memory-mapped I/O* angebunden sind. Auf diese Weise kann man alle Ein- und Ausgabefunktionen in ADA realisieren. In ADA können auch Prozeduren an Interrupt-Adressen abgelegt werden.

StateCharts bietet keine spezielle Unterstützung für Ein- und Ausgabegeräte in Form der Verwendung von E/A-Speicheradressen. Kommerzielle Implementierungen ergänzen StateCharts aber üblicherweise um diese Funktionalität.

2.5 SDL

Wegen der Kommunikation über gemeinsamen Speicher und aufgrund des *Broadcast*-Mechanismus sind StateCharts für die Modellierung verteilter Systeme ungeeignet. Wir wenden uns nun einer zweiten Sprache zu, die genau für diese Systeme prädestiniert ist: SDL. SDL wurde speziell für verteilte Anwendungen entwickelt und basiert auf asynchronem Nachrichtenaustausch. Die erste Entwicklung begann in den siebziger Jahren, die formale Semantik wurde in den späten achtziger Jahren definiert. Die Sprache ist von der ITU (International Telecommunication Union) standardisiert worden. Der erste Standard, „Z.100 Recommendation“ wurde 1980 veröffentlicht und in den Jahren 1984, 1988, 1992 (SDL-92), 1996 und 1999 aktualisiert. Wichtige Versionen sind u.a. SDL-88, SDL-92 und SDL-2000 [SDL Forum Society, 2003a].

Viele Anwender bevorzugen graphische Spezifikationssprachen, während andere textuelle Spezifikationen vorziehen. SDL bietet beide Möglichkeiten. Die Basiselemente von SDL sind Prozesse. Prozesse stellen erweiterte endliche Automaten dar. Die Erweiterungen beinhalten zum Beispiel Operationen auf Daten. Abb. 2.14 zeigt die Symbole, die in der graphischen Repräsentation von SDL verwendet werden.

Als Beispiel soll gezeigt werden, wie man das Zustandsdiagramm in Abb. 2.15 in SDL darstellen kann. Abb. 2.15 ist äquivalent zu Abb. 2.4 auf Seite 21, bis auf das Hinzufügen von Ausgabewerten, das Entfernen des Zustands Z

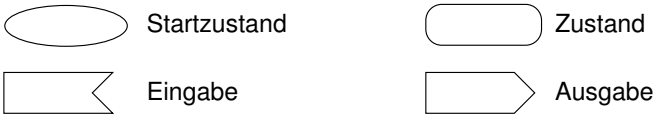


Abb. 2.14. Symbole in der graphischen Darstellung von SDL

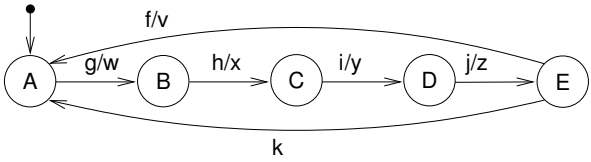


Abb. 2.15. Ein endlicher Automat

und der veränderten Wirkung des Signals k. Abb. 2.16 zeigt die zugehörige graphische SDL-Darstellung. Offensichtlich ist die Darstellung äquivalent zum Zustandsdiagramm in Abb. 2.15.

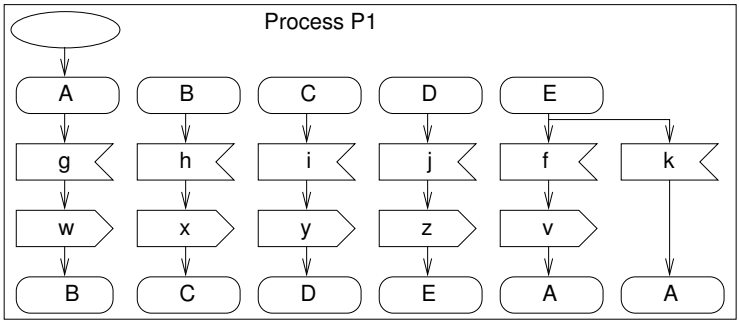


Abb. 2.16. SDL-Darstellung von Abb. 2.15

Als Erweiterung gegenüber klassischen endlichen Automaten können SDL-Prozesse Operationen auf Daten ausführen. Variablen können lokal für einen Prozess deklariert werden. Ihr Typ kann entweder vordefiniert werden oder er wird durch die SDL-Operation festgelegt. SDL unterstützt abstrakte Datentypen (ADTs). Die Syntax der Deklarationen und Operationen ist anderen Programmiersprachen sehr ähnlich. Abb. 2.17 zeigt Beispiele für Deklarationen, Zuweisungen und Entscheidungen in SDL.

SDL enthält auch Programmiersprachen-Elemente wie Prozeduren. Prozeduraufrufe können auch graphisch dargestellt werden. Objektorientierte Elemente wurden in SDL-1992 in die Sprache integriert und in SDL-2000 erweitert.

Erweiterte endliche Automaten sind nur die Basis-Elemente von SDL-Beschreibungen. Im Allgemeinen besteht eine SDL-Beschreibung aus einer Menge von interagierenden Prozessen oder Automaten. Prozesse können Signale an ande-

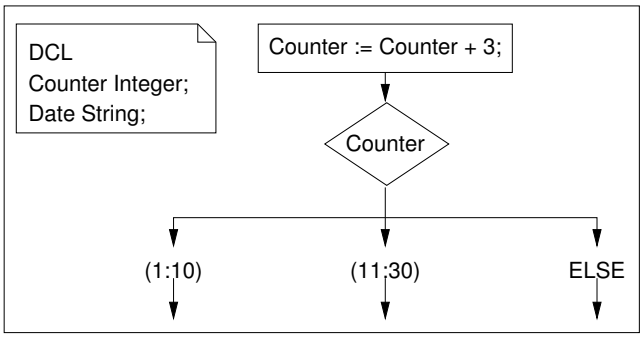


Abb. 2.17. Deklarationen, Zusweisungen und Entscheidungen in SDL

re Prozesse schicken. Die Semantik der Interprozess-Kommunikation in SDL basiert auf *first-in first-out* (FIFO) **Warteschlangen** die mit jedem Prozess verbunden sind. Ein Signal, das an einen bestimmten Prozess geschickt wird, landet in dessen FIFO-Warteschlange (s. Abb. 2.18). Diese Art der Kommunikation entspricht asynchronem Nachrichtenaustausch.

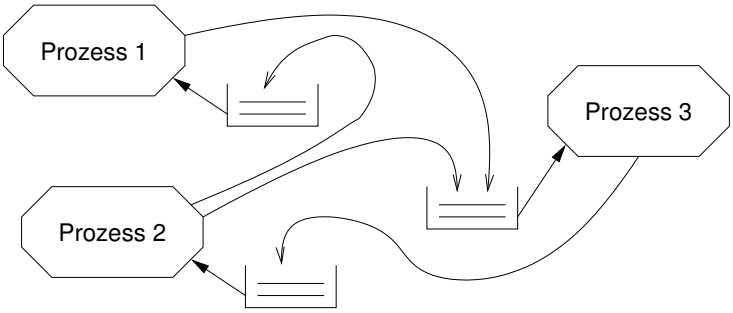


Abb. 2.18. SDL Interprozess-Kommunikation

Jeder Prozess entnimmt den nächsten verfügbaren Eintrag aus der FIFO-Warteschlange und prüft, ob er einem der Eingabewerte für den aktuellen Prozess und Zustand entspricht. Wenn das so ist, wird die zugehörige Transition ausgeführt und eine entsprechende Ausgabe generiert. Ein Eintrag in der FIFO-Warteschlange wird ignoriert, wenn er keinem der Eingabewerte entspricht (es sei denn, der sogenannte SAVE-Mechanismus wird verwendet). Die FIFO-Warteschlangen haben im Modell eine unendliche Kapazität. Das bedeutet, dass in der Semantik von SDL-Beschreibungen keine FIFO-Überläufe betrachtet werden. In echten Systemen dagegen müssen die FIFO-Warteschlangen natürlich eine endliche Länge haben. Dies ist eines der Probleme von SDL: um korrekte Realisierungen von einer SDL-Spezifikation ab-

zuleiten, muss man garantierte sichere obere Schranken für die Länge der FIFO-Warteschlangen bestimmen.

Prozess-Interaktionsdiagramme können verwendet werden, um zu visualisieren, welche Prozesse mit welchen anderen kommunizieren. Prozess-Interaktionsdiagramme beinhalten **Kanäle**, die zum Senden und Empfangen von Signalen verwendet werden. Bei SDL beschreibt der Begriff „Signal“ Ein- und Ausgaben der modellierten Automaten. Prozess-Interaktionsdiagramme sind Sonderfälle von **Blockdiagrammen**, die weiter unten näher beschrieben werden.

Beispiel: Abb. 2.19 zeigt ein Prozess-Interaktionsdiagramm B1 mit den Kanälen Sw1 und Sw2. In Klammern stehen die Namen der Signale, die über den jeweiligen Kanal transportiert werden.

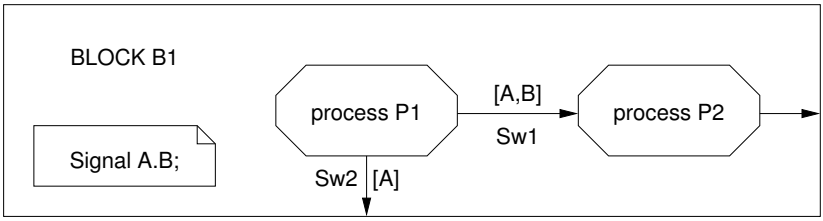


Abb. 2.19. Prozess-Interaktionsdigramm

Es gibt drei Möglichkeiten, den Empfänger eines Signals anzugeben:

1. **Durch Prozess-Identifikatoren:** durch die Angabe eines empfangenden Prozesses im graphischen Ausgabe-Symbol (s. linke Seite von Abb. 2.20).

Abb. 2.20. Angabe des Empfängers eines Signals

   Da Prozesse dynamisch generiert werden können, die Anzahl von Prozessen also nicht bereits zur Übersetzungszeit festgelegt werden muss, gibt der Wert OFFSPRING Prozesse an, die von einem anderen Prozess dynamisch generiert wurden.
2. **Explizit:** durch die Angabe eines Kanal-Namens (s. rechte Seite von Abb. 2.20). Sw1 ist der Name eines Kanals.
3. **Implizit:** Wenn es eine direkte Zuordnung von Signalen zu Kanälen gibt, wird bei Angabe eines Signals der jeweilige Kanal verwendet. Beispiel: in Abb. 2.19 wird das Signal B implizit immer über Kanal Sw1 laufen.

Ein Prozess kann nicht innerhalb eines anderen Prozesses definiert werden, Prozesse können also nicht verschachtelt werden. Sie können aber hierarchisch in sogenannten **Blöcken** gruppiert werden. Blöcke auf der höchsten Hierarchiestufe heißen **Systeme**, Blöcke auf der untersten Hierarchiestufe heißen **Prozess-Interaktionsdiagramme**. B1 kann in dazwischenliegenden Blöcken (etwa in B in Abb. 2.21) verwendet werden.

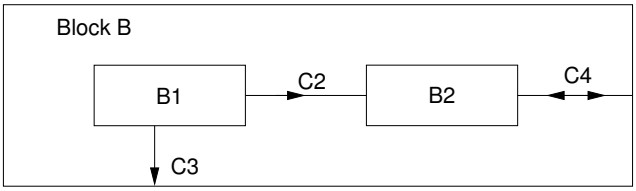


Abb. 2.21. SDL Block

Auf der höchsten Ebene innerhalb der Hierarchie befindet sich das System (s. Abb. 2.22). Ein System hat keine Kanäle nach außen, wenn dessen Umwelt auch als Block modelliert wird.

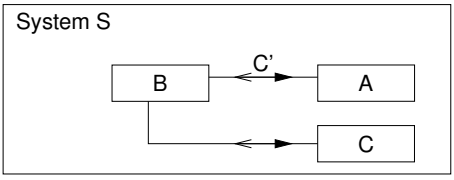


Abb. 2.22. SDL System

Abb. 2.23 zeigt die Hierarchie, die durch die Block-Diagramme 2.19, 2.21 und 2.22 modelliert wird. Prozess-Interaktionsdiagramme befinden sich eine Stufe oberhalb der **Blätter** der hierarchischen Beschreibung. Die System-Beschreibung bildet die **Wurzel**.

Einige Einschränkungen bei der Modellierung von Hierarchie wurden in SDL-2000 beseitigt. In SDL-2000 wurde die Ausdruckskraft von Blöcken und Prozessen angeglichen und durch ein allgemeines **Agenten**-Konzept ersetzt.

Zur Modellierung von Zeit enthält SDL ***Timer***. *Timer* können lokal für Prozesse deklariert werden. Sie können mit Hilfe von SET und RESET-Anweisungen gesetzt und zurückgesetzt werden. Abb. 2.24 zeigt die Verwendung eines *Timers* T. Das Diagramm entspricht dem in Abb. 2.16, mit dem Unterschied, dass der *Timer* T beim Übergang von Zustand D in Zustand E auf den Wert der aktuellen Zeit now plus p gesetzt wird. Für die Transition von E nach A ist damit ein *Timeout* von p Zeiteinheiten definiert. Wenn diese

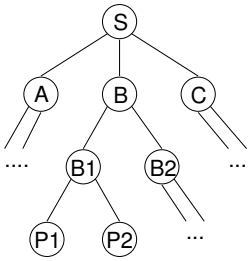


Abb. 2.23. SDL-Hierarchie

Zeit verstrichen ist, bevor das Signal f eintrifft, findet ein Übergang in den Zustand A statt, bei dem kein Ausgabesignal v erzeugt wird.

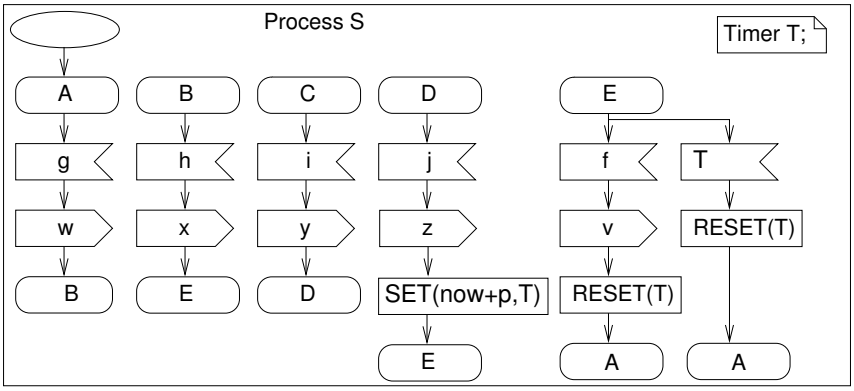


Abb. 2.24. Verwendung eines Timers T

SDL kann beispielsweise verwendet werden, um Protokoll-Stapel in Rechner-Netzwerken zu beschreiben. Abb. 2.25 zeigt drei Prozessoren, die durch einen Router verbunden werden. Die Kommunikation zwischen Prozessoren und Router basiert auf FIFO-Warteschlangen.

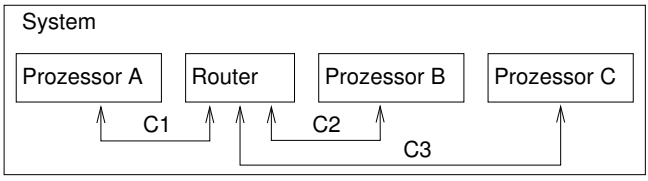


Abb. 2.25. Kleines Computernetzwerk in SDL

Sowohl die Prozessoren als auch der Router implementieren Protokolle mit verschiedenen Ebenen (s. Abb. 2.26).

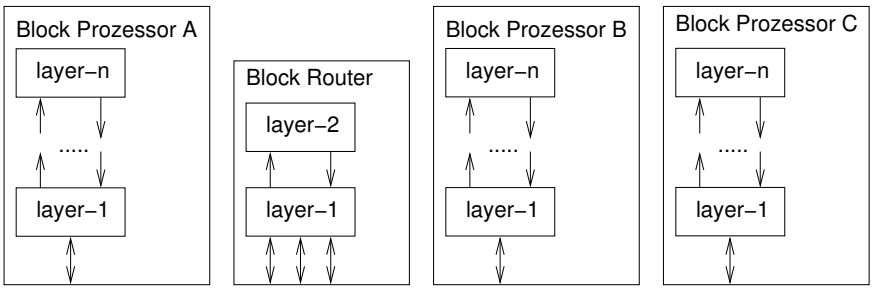


Abb. 2.26. Protokoll-*Stacks* in SDL

Jede Ebene beschreibt die Kommunikation mit einer abstrakteren Sichtweise. Das Verhalten jeder Ebene wird üblicherweise als endlicher Automat modelliert. Die genaue Beschreibung dieser Automaten hängt vom verwendeten Netzwerk-Protokoll ab und kann sehr komplex werden. Typischerweise wird das Überprüfen und Behandeln von Fehlerbedingungen sowie das Sortieren und Weiterleiten von Informationspaketen realisiert.

Es gibt kommerzielle Programme, die SDL unterstützen. Verfügbare Programme für SDL von Firmen wie Telelogic [Telelogic AB, 2003], Cinderella [Cinderella ApS, 2003] und SINTEF enthalten Schnittstellen zu UML (s. Seite 51), MSCs (s. Seite 49) und CHILL (s. Seite 85). Eine vollständige Liste verfügbarer Programme wird im SDL-Forum zur Verfügung gestellt [SDL Forum Society, 2003b].

SDL ist hervorragend zur Modellierung von verteilten Anwendungen geeignet und wurde zum Beispiel für die Spezifikation von ISDN verwendet. SDL ist nicht in allen Situationen deterministisch, da die Reihenfolge, in der gleichzeitig ankommende Signale in die FIFO-Warteschlangen eingereiht werden, nicht spezifiziert ist. Verlässliche Implementierungen erfordern das manchmal schwierige Bestimmen einer oberen Schranke für die Länge der FIFO-Warteschlangen. Die Modellierung von Zeitbedingungen ist ausreichend für weiche Zeitschranken, aber nicht für harte Echtzeitbedingungen. Hierarchien werden nicht so wie in StateCharts unterstützt. Es gibt keine vollständige Unterstützung für Programmiersprachenkonstrukte, obwohl dies von den aktuellen Standards angestrebt wird, und keine Beschreibung von nicht-funktionalen Eigenschaften.

2.6 Petrinetze

2.6.1 Einführung

1962 veröffentlichte Carl Adam Petri seine Methoden zur Darstellung kausaler Abhängigkeiten [Petri, 1962], die unter dem Namen **Petrinetze** bekannt wurden. Die Stärke von Petrinetzen liegt in dieser Konzentration auf kausale Abhängigkeiten. Petrinetze gehen nicht von einer globalen Synchronisation aus und sind daher besonders gut zur Modellierung verteilter Systeme geeignet.

Bedingungen, **Ereignisse** und eine **Flussrelation** sind die Kernelemente von Petrinetzen. Bedingungen sind entweder erfüllt oder nicht erfüllt. Ereignisse können eintreten. Die Flussrelation beschreibt die Bedingungen, die erfüllt sein müssen, damit Ereignisse eintreten können. Außerdem beschreibt sie, welche Bedingungen wahr werden, wenn ein Ereignis eintritt.

Graphische Darstellungen von Petrinetzen verwenden typischerweise Kreise, um Bedingungen, und Rechtecke, um Ereignisse darzustellen. Pfeile stellen die Flussrelation dar. Abb. 2.27 zeigt ein erstes Beispiel.

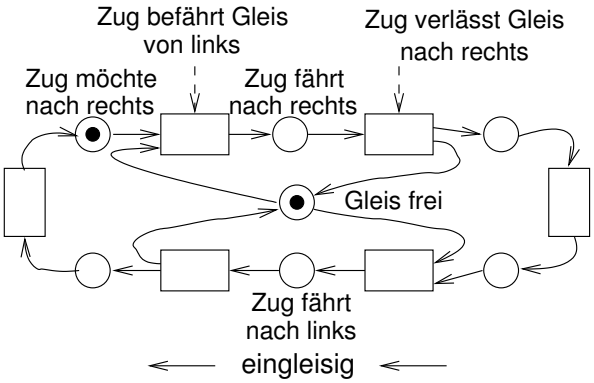


Abb. 2.27. Eingleisiger Bahnstrecken-Abschnitt

Dieses Beispiel beschreibt den gegenseitigen Ausschluss von Zügen auf einem eingleisigen Streckenabschnitt, der in beide Richtungen befahren werden muss. Ein sogenanntes *Token* oder eine Marke wird verwendet, um die Kollision von Zügen zu vermeiden, die in unterschiedliche Richtungen fahren. Im Beispiel-Petrinetz wird das *Token* durch eine erfüllte Bedingung in der Mitte des Modells dargestellt. Der kleine, im großen Kreis enthaltene ausgefüllte Kreis symbolisiert die Situation, in der die Bedingung erfüllt ist (im Beispiel: der Abschnitt ist frei). Generell werden in Petrinetzen erfüllte Bedingungen durch **Marken** repräsentiert und in graphischen Darstellungen durch kleine, gefüllte Kreise symbolisiert. Dementsprechend kennzeichnet eine weitere

Marke die Tatsache, dass ein Zug nach rechts fahren möchte. Damit sind die beiden Bedingungen zum Schalten des Ereignisses „Zug befährt den Abschnitt von links“ erfüllt. Wir nennen diese Bedingungen **Vorbedingungen**. Wenn die Vorbedingungen eines Ereignisses erfüllt sind, kann das Ereignis eintreten. Nach dem Eintreten des Ereignisses sind die Vorbedingungen nicht mehr erfüllt: es wartet kein Zug mehr auf den eingleisigen Abschnitt und das *Token* zur Nutzung des Streckenabschnitts ist ebenfalls nicht mehr verfügbar. Folglich sind die beiden Marken aus Abb. 2.27 in Abb. 2.28 nicht mehr vorhanden.

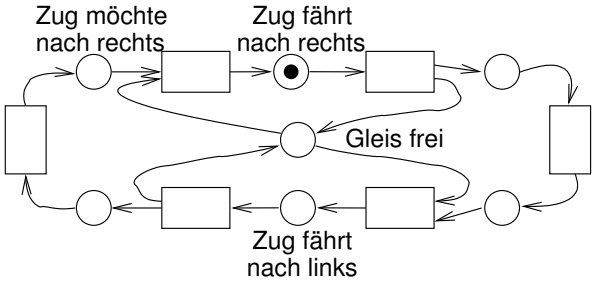


Abb. 2.28. Verwendung der Ressource „Gleis-Abschnitt“

Andererseits fährt jetzt ein Zug auf dem Abschnitt von links nach rechts. Somit ist die zugehörige Bedingung erfüllt (s. Abb. 2.28). Eine Bedingung, die nach dem Eintreten eines Ereignisses erfüllt ist, heißt **Nachbedingung**. Im Allgemeinen kann ein Ereignis nur dann eintreten, wenn alle seine Vorbedingungen erfüllt sind. Wenn das Ereignis eingetreten ist, sind die Vorbedingungen nicht mehr erfüllt, stattdessen werden die Nachbedingungen erfüllt. Entsprechend verändert sich die Kennzeichnung von Bedingungen mit Marken. Die Vor- und Nachbedingungen eines Ereignisses werden durch Pfeile dargestellt. In unserem Zugbeispiel sieht man, dass ein Zug, der den eingleisigen Streckenabschnitt verlässt, die Marke in der Mitte des Modells wiederherstellt (s. Abb. 2.29).

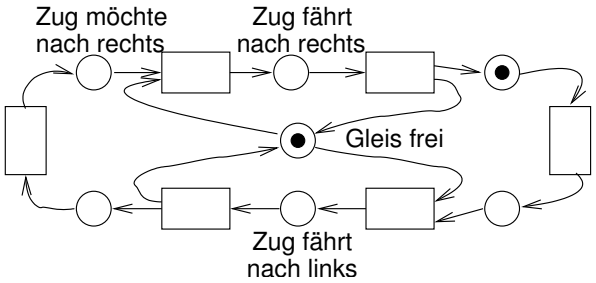


Abb. 2.29. Freigabe der Ressource „Gleis-Abschnit“

Wenn zwei Züge gleichzeitig den eingleisigen Abschnitt benutzen wollen (s. Abb. 2.30), kann nur einer der beiden in den Abschnitt eintreten.

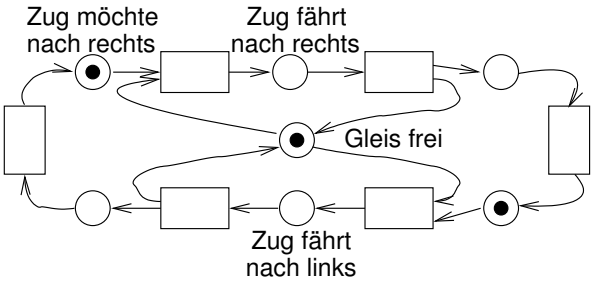


Abb. 2.30. Konflikt um die Ressource „Gleis-Abschnitt“

Wir betrachten nun ein größeres Beispiel: Wieder geht es um die Synchronisation von Zügen, in diesem Fall werden die Hochgeschwindigkeitszüge vom Typ „Thalys“ modelliert, die zwischen Amsterdam, Köln, Brüssel und Paris verkehren. Es gibt unabhängige Zugteile, die von Amsterdam und Köln nach Brüssel fahren. Dort werden die Zugteile verbunden und fahren dann gemeinsam nach Paris weiter. Auf dem Rückweg von Paris werden die Züge in Brüssel wieder getrennt. Wir nehmen an, dass die Züge in Paris Anschlüsse an Züge aus dem Pariser Süden sicherstellen müssen. Das zugehörige Petrinetz zeigt Abb. 2.31.

Erfüllte Bedingungen 3 und 10 stellen Züge dar, die in Amsterdam bzw. Köln warten. Die Transitionen 9 und 2 modellieren Züge, die von diesen Städten aus nach Brüssel fahren. Nach der Ankunft in Brüssel enthalten die Stellen 9 und 2 jeweils eine Marke. Transition 1 symbolisiert das Verbinden der beiden Zugteile. Die Tasse steht für einen der beiden Lokführer, der in Brüssel eine Pause hat, während der andere nach Paris weiterfährt. Transition 5 stellt die Anschlussbedingungen mit anderen Zügen im Gare du Nord von Paris dar. Diese Anschlüsse verbinden den Gare du Nord mit anderen Pariser Bahnhöfen (wir haben hier lediglich den Gare de Lyon als Beispiel gezeigt, obwohl es in Paris noch weitere Bahnhöfe gibt). Selbstverständlich fahren die Thalys-Züge nicht mit Dampflokomotiven – die verwendeten Symbole sind aber eingängiger als die von modernen Hochgeschwindigkeitszügen.

Einer der Hauptvorteile von Petrinetzen ist ihre Eignung für formale Beweise über Systemeigenschaften, sowie die Tatsache, dass es standardisierte Methoden gibt, solche Beweise zu erzeugen. Um solche Beweise zu ermöglichen, benötigen wir eine formale Definition von Petrinetzen.

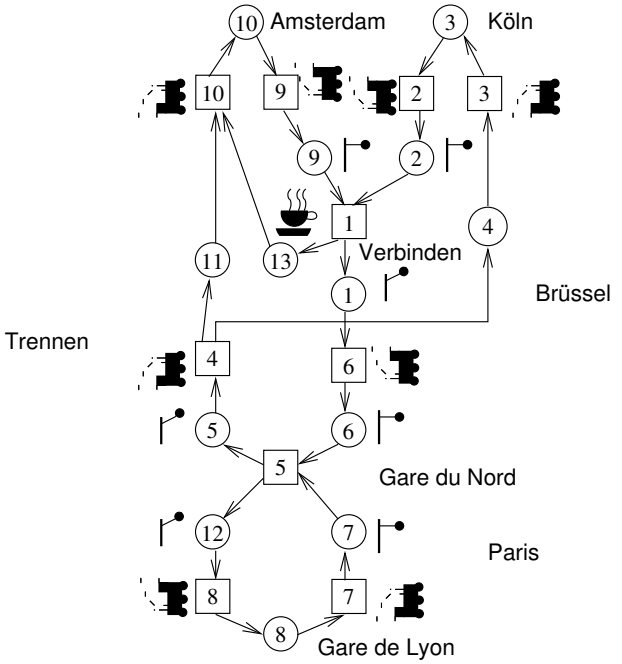


Abb. 2.31. Modell der Thalys-Züge zwischen Amsterdam, Köln, Brüssel und Paris

2.6.2 Bedingungs-/Ereignisnetze

Als erste Klasse von Petrinetzen werden wir die Bedingungs-/Ereignisnetze formal definieren.

Definition: $N = (C, E, F)$ heißt **Netz**, genau dann wenn die folgenden Bedingungen erfüllt sind:

1. C und E sind disjunkte Mengen.
2. $F \subseteq (E \times C) \cup (C \times E)$ ist eine zweistellige Relation, genannt Flussrelation.

Die Menge C heißt die Menge der Bedingungen und die Menge E ist die Menge der Ereignisse.

Definition: Sei N ein Netz und sei $x \in (C \cup E)$. Dann heißt ${}^\bullet x := \{y | yFx\}$ die Menge der Vorbedingungen (oder der Vorbereich) von x und $x^\bullet := \{y | xFy\}$ ist die Menge der Nachbedingungen (oder der Nachbereich) von x.

Diese Definition wird zumeist für den Fall $x \in E$ verwendet, gilt aber auch für $x \in C$.

Definition: Sei $(c, e) \in C \times E$.

1. (c, e) heißt **Schleife**, wenn $cFe \wedge eFc$.

2. N heißt **rein**, wenn F keine Schleifen enthält

Definition: Ein Netz heißt einfach, wenn keine zwei Transitionen t_1 und t_2 die gleiche Menge von Vor- und Nachbedingungen haben.

Abb. 2.32. Netze, die nicht rein (links) und nicht einfach sind (rechts)

Einfache Netze ohne isolierte Elemente, die einige zusätzliche Anforderungen erfüllen, heißen **Bedingungs-/Ereignisnetze**. Bedingungs-/Ereignisnetze sind ein Sonderfall von bipartiten Graphen (Graphen mit zwei unterschiedlichen, disjunkten Mengen von Knoten). Die zusätzlich notwendigen Bedingungen werden wir hier nicht weiter vertiefen, da wir uns im Weiteren mit allgemeineren Klassen von Netzen beschäftigen.

2.6.3 Stellen-/Transitionen-Netze

Bei Bedingungs-/Ereignisnetzen gibt es höchstens eine Marke pro Bedingung. Für viele Anwendungen ist es nützlich, diese Einschränkung aufzuheben und mehrere Marken pro Bedingung zu erlauben. Netze, die mehrere *Token* pro Bedingung erlauben, heißen Stellen-/Transitionen-Netze (*place/transition nets*). Stellen entsprechen den bisher verwendeten Bedingungen, Transitionen entsprechen den Ereignissen. Die Anzahl von Marken pro Stelle heißt **Belegung**. Mathematisch betrachtet ist eine Belegung eine Abbildung von der Menge der Stellen auf die Menge der natürlichen Zahlen, die um ein besonderes Symbol erweitert wird: ω steht für unendlich.

Sei $\mathbb{N}_0$ die Menge der natürlichen Zahlen inklusive der 0. Dann kann ein Stellen-/Transitionen-Netz formal wir folgt definiert werden:

Definition: (P, T, F, K, W, M_0) heißt **Stellen-/Transitionen-Netz** genau dann wenn

1. $N = (P, T, F)$ ein Netz ist, mit den Stellen $p \in P$ und den Transitionen $t \in T$,
2. die Abbildung $K : P \to (\mathbb{N}_0 \cup \{\omega\}) \setminus \{0\}$ die Kapazität der Stellen darstellt (wobei ω eine unbeschränkte Kapazität symbolisiert),
3. die Abbildung $W : F \to (\mathbb{N}_0 \setminus \{0\})$ das Gewicht der Kanten des Graphen darstellt,
4. die Abbildung $M_0 : P \to \mathbb{N}_0 \cup \{\omega\}$ die initiale Belegung der Stellen mit Marken darstellt.

Kantengewichte haben einen Einfluss auf die Anzahl der Marken, die benötigt werden, bevor eine Transition schalten kann und geben auch die Anzahl von Marken an, die von einer schaltenden Transition erzeugt werden. Sei $M(p)$ die aktuelle Belegung der Stelle $p \in P$ und sei $M'(p)$ die Markierung nach dem Schalten einer Transition $t \in T$. Das Gewicht der Kanten, die zu Vorbedingungen von t gehören, gibt die Anzahl der Marken an, die aus den Vorbedingungs-Stellen abgezogen werden. Entsprechend stellt das Gewicht der Kanten, die zu Nachbedingungen führen, die Anzahl der Marken dar, die den Stellen in den Nachbedingungen hinzugefügt werden. Formal wird die neue Belegung M' wie folgt berechnet:

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

Abb. 2.33 zeigt ein Beispiel, wie sich das Schalten von Transition t_j auf die aktuelle Belegung auswirkt.

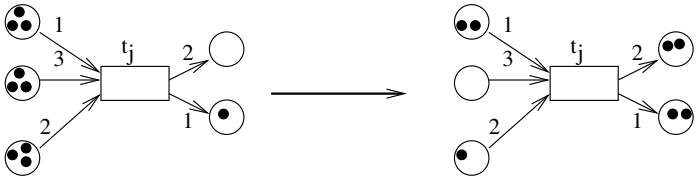


Abb. 2.33. Erzeugen einer neuen Belegung

Für unbeschriftete Kanten vereinbart man ein Gewicht von 1, und Knoten ohne entsprechende Angabe haben eine unbeschränkte Kapazität ω.

Jetzt müssen noch die beiden Bedingungen beschrieben werden, die erfüllt sein müssen, bevor eine Transition $t \in T$ schalten kann:

- in allen Stellen p in der Menge der Vorbedingungen muss die Anzahl der Marken mindestens so groß sein wie das Gewicht der Kante von p nach t, und
- in allen Stellen p in der Menge der Nachbedingungen muss die Kapazität groß genug sein, um die von t neu erzeugten Marken aufzunehmen.

Transitionen, die diese beiden Bedingungen erfüllen, heißen **M-aktiviert**. Formal kann dies wie folgt definiert werden:

Definition: Eine Transition $t \in T$ heißt M-aktiviert genau dann wenn

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

Aktivierte Transitionen können schalten, müssen dies aber nicht zwangsläufig. Wenn mehrere Transitionen aktiviert sind, ist die Reihenfolge ihres Schaltens nicht deterministisch definiert.

Der Einfluss einer schaltenden Transition t auf die Markenzahl kann bequem durch einen der Transition zugeordneten Vektor $\underline{t}$ beschrieben werden, der wie folgt definiert ist:

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

Beim Schalten einer Transition t ergibt sich dann die neue Markenzahl M' für alle Stellen p wie folgt:

$$M'(p) = M(p) + \underline{t}(p)$$

Unter Benutzung der Vektoraddition können wir verkürzt schreiben:

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

Aus der Menge aller Vektoren $\underline{t}$ können wir eine sogenannte Inzidenzmatrix $\underline{N}$ bilden, welche als Spalten die Vektoren der verschiedenen Transitionen enthält:

$$\underline{N} : P \times T \to \mathbb{Z}; \qquad \forall t \in T : \underline{N}(p,t) = \underline{t}(p)$$

Mit Hilfe dieser Matrix kann man auf standardisierte Weise formale Beweise von Systemeigenschaften führen. Beispielsweise kann es Teilmengen der Stellen geben, in denen sich die Gesamtzahl der Marken unabhängig von den schaltenden Transitionen nicht verändert [Reisig, 1985]. Solche Stellenmengen konstanter Markensumme nennen wir **S-Invarianten**. Um solche S-Invarianten zu finden, betrachten wir zunächst eine Transition t_j und suchen Stellenmengen $R \subseteq P$ für welche das Schalten der Transition die Markenzahl nicht verändert. Für diese muss gelten:

$$\sum_{p \in R} \underline{t}_j(p) = 0 \tag{2.1}$$

Abbildung 2.34 zeigt ein Beispiel für eine Transition, bei der für die drei Stellen die Markensumme konstant bleibt.

Zur Vereinfachung der Summenschreibweise in Gleichung 2.1 führen wir nunmehr den sogenannten charakteristischen Vektor $\underline{c}_R$ einer Stellenmenge R ein:

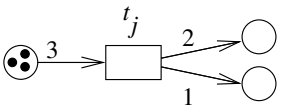


Abb. 2.34. Transition mit konstanter Markensumme

$$\underline{c}_R(p) = \begin{cases} 1 \text{ wenn } p \in R \\ 0 \text{ wenn } p \notin R \end{cases}$$

Damit können wir Gleichung 2.1 umschreiben in:

$$\sum_{p \in R} \underline{t}_j(p) = \sum_{p \in P} \underline{t}_j(p)\underline{c}_R(p) = \underline{t}_j \cdot \underline{c}_R = 0 \tag{2.2}$$

Dabei kennzeichnet $\cdot$ das Skalarprodukt. Wir suchen jetzt Stellenmengen, für die das Schalten **aller** Transitionen die Markensumme konstant lässt. Dann muss die Gleichung 2.2 statt für eine Transition t_j für alle Transitionen gelten:

$$\begin{aligned} \underline{t}_1 \cdot \underline{c}_R &= 0 \\ \underline{t}_2 \cdot \underline{c}_R &= 0 \\ &\ldots \\ \underline{t}_n \cdot \underline{c}_R &= 0 \end{aligned} \tag{2.3}$$

Das Gleichungssystem 2.3 lässt sich mit der transponierten Inzidenzmatrix $\underline{N}^T$ zusammenfassen zu:

$$\underline{N}^T \underline{c}_R = 0 \tag{2.4}$$

Das Gleichungssystem 2.4 ist ein lineares homogenes Gleichungssystem. Die Matrix $\underline{N}$ beschreibt die Kantengewichte des Petrinetzes. Gesucht sich Vektoren $\underline{c}_R$, welche dieses Gleichungssystem lösen. Da die Lösungsvektoren charakteristische Vektoren sein müssen, können wir als Komponenten der Vektoren nur 0 und 1 erlauben[5]. Das Lösen derartiger Gleichungssysteme ist komplexer als das Lösen von Gleichungssystemen mit reellwertigen Lösungsvektoren. Dennoch können durch das Lösen der Gleichung 2.4 Aussagen über Eigenschaften eines Petrinetzes gewonnen werden. In unserem Beispiel ändert sich

[5] Wenn wir gewichtete Markensummen betrachten, können wir natürliche Zahlen als Lösungen erlauben.

die Anzahl der Züge, die zwischen Köln und Paris verkehren, nicht. Das gleiche gilt für die Züge zwischen Amsterdam und Paris. In Modellen, die den Zugriff auf gemeinsame Ressourcen modellieren, kann beispielsweise der gegenseitige Ausschluss nachgewiesen werden.

2.6.4 Prädikat-/Ereignis-Netze

Sowohl Bedingungs-/Ereignisnetze als auch Stellen-/Transitionen-Netze können für größere Beispiele schnell sehr groß und unübersichtlich werden. Eine Verringerung der Größe ist häufig durch den Einsatz von Prädikat-/Ereignis-Netzen möglich. Wir werden dies am Beispiel der „speisenden Philosophen“ demonstrieren. Das Problem geht von einer Anzahl von Philosophen aus, die an einem runden Tisch essen. Vor jedem Philosophen steht ein Teller mit Spaghetti. Zwischen den Tellern befindet sich jeweils nur eine Gabel (s. Abb. 2.35). Jeder Philosoph ist entweder mit Essen oder mit Nachdenken beschäftigt. Essende Philosophen benötigen dafür die beiden Gabeln, die neben ihrem Teller liegen. Folglich kann ein Philosoph nur dann essen, wenn seine beiden Nachbarn gerade nicht essen.

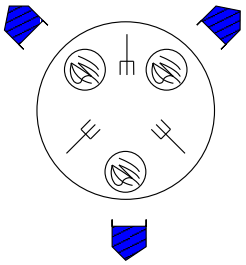


Abb. 2.35. Das Problem der speisenden Philosophen

Diese Situation kann, wie in Abb. 2.36 gezeigt, in einem Bedingungs-/Ereignisnetz modelliert werden. Die Bedingungen t_j entsprechen dem „denkenden“, die Bedingungen e_j dem „essenden“ Zustand, und die Bedingungen f_j stellen die verfügbaren Gabeln dar.

In Anbetracht der geringen Größe des zugrundeliegenden Problems ist dieses Netz bereits recht groß. Das Netz lässt sich verkleinern, wenn man ein Prädikat-/Ereignis-Netz verwendet. Abb. 2.37 zeigt ein Modell des Philosophen-Problems als Prädikat-/Ereignis-Netz.

Bei Prädikat-/Ereignis-Netzen haben Marken eine Identität und können unterschieden werden. Dies wird in Abb. 2.37 benutzt, um die drei Philosophen p_1 bis p_3 zu unterscheiden und um die Gabel f_3 zu identifizieren[6]. Desweiteren können Kanten mit Beschriftungen versehen werden, die Variablen und

[6] In diesem Beispiel ist f_3 die dem Philosophen p_1 gegenüberliegende Gabel.

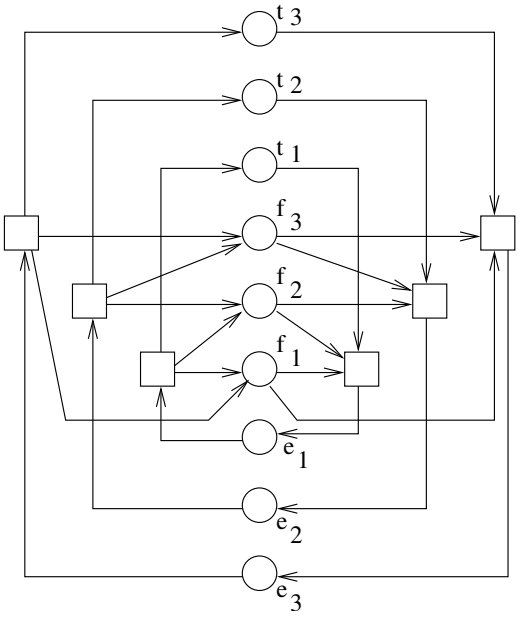


Abb. 2.36. Stellen-/Transitionen-Netz des Philosophen-Problems

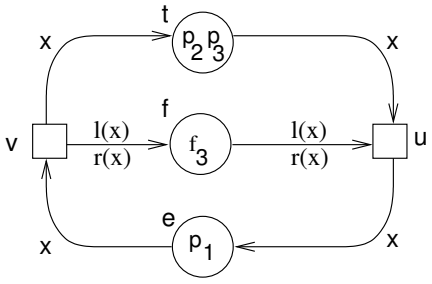


Abb. 2.37. Prädikat-/Ereignis-Netz des Philosophen-Problems

Funktionen repräsentieren. Im Beispiel werden Variablen verwendet, um die Identität der Philosophen zu beschreiben. Die Funktionen l(x) bzw. r(x) beschreiben die linke bzw. rechte Gabel von Philosoph x. Diese beiden Gabeln werden als Vorbedingung für die Transition u benötigt, und werden als Nachbedingung beim Schalten der Transition v wieder zurückgegeben. Dieses Modell kann einfach durch das Hinzufügen weiterer Marken auf den Fall von $n > 3$ Philosophen erweitert werden. Im Gegensatz zu dem Netz in Abb. 2.36 muss die eigentliche Struktur des Netzes dazu nicht verändert werden.

2.6.5 Bewertung

Der Hauptvorteil von Petrinetzen ist ihre Stärke bei der Modellierung kausaler Abhängigkeiten. Standard-Petrinetze bieten keine Unterstützung für Zeitbedingungen. Alle Entscheidungen können lokal getroffen werden, indem Transitionen mit ihren Vor- und Nachbedingungen analysiert werden. Aus diesem Grund können sie zur Modellierung von geographisch verteilten Systemen verwendet werden. Außerdem gibt es starke theoretische Grundlagen für die Betrachtung von Petrinetzen, was formale Beweise von Systemeigenschaften vereinfacht.

In manchen Zusammenhängen sind die Stärken von Petrinetzen aber auch ihre Schwächen. Wenn Zeitbedingungen zu berücksichtigen sind, können Standard-Petrinetze nicht verwendet werden. Außerdem bieten diese kein Hierarchie-Konzept und keine Programmiersprachen-Konstrukte an, von objektorientierten Konzepten ganz abgesehen. In der Regel ist es schwierig, Daten in Petrinetzen darzustellen.

Es gibt Erweiterungen von Petrinetzen, die einige dieser Schwächen beheben. Allerdings gibt es keine universelle Petrinetz-Erweiterung, die alle Anforderungen, die am Anfang dieses Kapitels aufgestellt wurden, erfüllt. Trotzdem haben sich Petrinetze aufgrund der zunehmenden Anzahl verteilter Systeme in der ganzen Welt zunehmend verbreitet.

2.7 *Message Sequence Charts*

Message Sequence Charts(MSCs) stellen eine Möglichkeit dar, Ablaufpläne graphisch darzustellen. In MSCs stellt eine Dimension (üblicherweise die vertikale) die Zeit dar, die andere Dimension steht für eine geographische Verteilung.

MSCs sind gut geeignet, um Fahrpläne von Zügen oder Bussen darzustellen. Abb. 2.38 zeigt ein Beispiel. Es bezieht sich wieder auf Züge, die zwischen Amsterdam, Köln, Brüssel und Paris verkehren. Aachen ist als zusätzlicher Halt auf der Strecke zwischen Köln und Brüssel dargestellt. Vertikale Segmente zeigen die Zeit, die der Zug in einem Bahnhof verbringt. In Brüssel gibt es eine zeitliche Überlappung zwischen den Zügen aus Richtung Köln und aus Richtung Amsterdam. Es gibt einen zweiten Zug zwischen Paris und Köln, der keine Umsteigebeziehung zu dem Zug von/nach Amsterdam hat.

Ein realistischeres Beispiel ist in Abb. 2.39 dargestellt. Es beschreibt den Zugverkehr im Bereich des Schweizer Lötschbergs [Huerlimann, 2003]. Schnelle und langsame Züge können anhand der Steigung der Linien unterschieden werden. Die Abbildung enthält zusätzlich Informationen zur genauen Zeit. In diesem Zusammenhang spricht man von einem **Weg-Zeit-Diagramm**. Weg-Zeit-Diagramme können wir als Variante von MSCs auffassen.

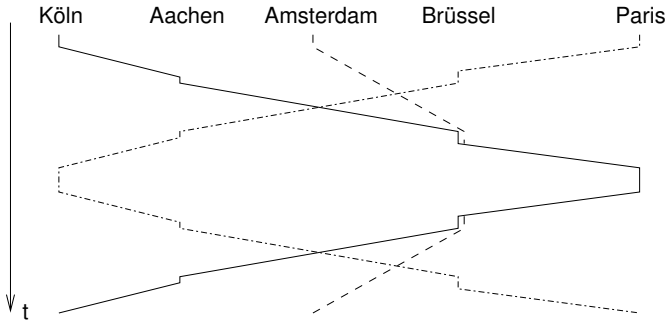


Abb. 2.38. *Message Sequence Chart*

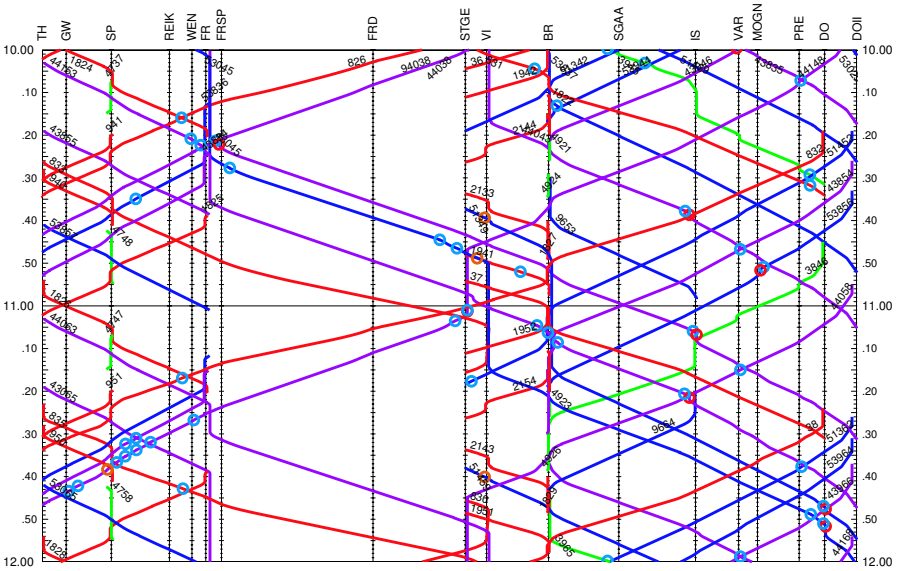


Abb. 2.39. Zugverkehr, dargestellt mit Hilfe eines *Message Sequence* Diagramms (mit freundlicher Genehmigung von H. Brändli, IVT, ETH Zürich), ©ETH Zürich

MSCs sind geeignet, um typische Ablaufpläne, wie etwa Fahrpläne, darzustellen. Allerdings geben Sie keinerlei Informationen zur eventuell notwendigen Synchronisation. So ist zum Beispiel nicht klar, ob die zeitliche Überlappung in Brüssel zufällig passiert, oder ob eine echte zeitliche Abstimmung zwischen den Zügen notwendig ist, um den Anschluss für umsteigende Reisende zu sichern. Außerdem können erlaubte Abweichungen von einem vorgegebenen Ablauf (Min-/Max-Timing) nicht in MSCs dargestellt werden. Man weiß auch nicht, ob ein bestimmtes Diagramm ausdrücken soll, dass die gezeigte Sequenz existiert (*existential interpretation*) oder ob sie ausdrücken soll, dass alle Abläufe genau gemäß der gezeigten Sequenz stattfinden sol-

len (*universal interpretation*). Damm und Harel haben als Erweiterung von MSCs sogenannte ***Life Sequence Charts*** eingeführt, bei denen zwischen verpflichtendem Verhalten und möglichem Verhalten unterschieden werden kann [Damm und Harel, 2001].

2.8 UML

Alle bisher vorgestellten Sprachen gehen von einer recht genauen Kenntnis des Verhaltens des zu spezifizierenden Systems aus. Häufig, insbesondere während der frühen Phasen der Spezifikation, sind solche detaillierten Informationen jedoch nicht verfügbar. Allererste Entwürfe solcher Systeme entstehen oft auf Servietten oder auf benutzten Briefumschlägen. Diese erste Planungsphase zu systematisieren ist das Ziel von UML. UML [OMG, 2005], [Fowler und Scott, 1998] ist die Abkürzung von „*Unified Modeling Language*“. Es wurde von führenden Software-Technologie-Experten entwickelt und wird von vielen kommerziellen Tools unterstützt. UML unterstützt hauptsächlich den Prozess der Software-Entwicklung. Durch die vielen verwendeten Diagrammtypen ist UML eine komplexe graphische Sprache. Glücklicherweise sind die meisten Diagrammtypen Abwandlungen von graphischen Darstellungen, die bereits in diesem Buch vorgestellt wurden.

Die Version 1.4 von UML wurde nicht für eingebettete Systeme entworfen. Daher fehlen einige der Anforderungen an Sprachen zur Modellierung von eingebetteten Systemen (s. Seite 13). Insbesondere fehlen die folgenden Eigenschaften [McLaughlin und Moore, 2001]:

- Die Partitionierung der Software in Tasks und Prozesse kann nicht dargestellt werden,
- Zeitverhalten kann überhaupt nicht darstellt werden,
- die Verfügbarkeit von benötigten Hardware-Komponenten kann nicht beschrieben werden.

Aufgrund der zunehmenden Menge an Software in eingebetteten Systemen wird UML auch in diesem Bereich zunehmend wichtiger. Aus diesem Grunde wurden verschiedene Vorschläge zur Erweiterung von UML gemacht, welche auch die Unterstützung von Echtzeit-Applikationen ermöglichen sollen [McLaughlin und Moore, 2001], [Douglass, 2000]. Während der Entwicklung von UML 2.0 wurden diese Erweiterungen berücksichtigt. UML 2.0 beinhaltet 13 Diagrammtypen (im Gegensatz zu neun in UML 1.4) [Ambler, 2005]. Die deutsche Benennung der UML-Diagramme orientiert sich an den Vorgaben von Jeckle et al. [Jeckle, 2004].

- **Sequenz-Diagramme**: Sequenz-Diagramme sind im Prinzip Abwandlungen von *Message Sequence Charts*. Abb. 2.40 zeigt ein Sequenzdiagramm, das auf einem Beispiel der Gentleware AG [Poseidon, 2003] basiert.

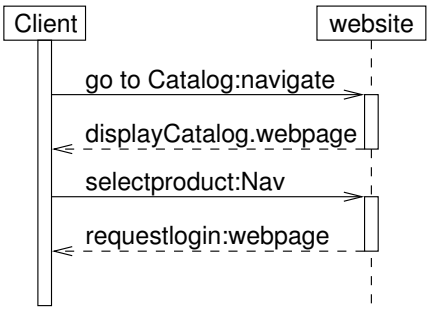


Abb. 2.40. Teil aus einem UML Sequenz-Diagramm

Der Hauptunterschied zwischen den Diagrammen in Abb. 2.39 und 2.40 ist die Tatsache, dass Abb. 2.40 keine konkreten Zeitangaben enthält, da UML 1.4 nicht für Echtzeitanwendungen konzipiert wurde. Einige der Einschränkungen von UML 1.4 wurden in UML 2.0 behoben.

- **Automaten-Diagramme** (in Version 1 von UML: **Zustandsdiagramme**): UML enthält eine Variante von StateCharts und unterstützt somit die Modellierung von endlichen Automaten.
- **Aktivitäts-Diagramme**: Im Prinzip sind Aktivitäts-Diagramme erweiterte Petrinetze. Die Erweiterungen umfassen z.B. Symbole für Entscheidungen (wie in den üblichen Fluss-Diagrammen). Die Platzierung von Symbolen ist ähnlich wie in SDL. Abb. 2.41 zeigt ein Beispiel.

  Das Beispiel zeigt die Durchführung einer Standardisierung. Verzweigungen und Zusammenführungen des Kontrollflusses entsprechen Transitionen in Petrinetzen, deren Symbole (horizontale Balken) hier auch verwendet werden. Die Raute am unteren Rand ist das Symbol, das für Entscheidungen verwendet wird. Aktivitäten können zu „Bahnen“ (die Bereiche zwischen den vertikalen gepunkteten Linien) zusammengefasst werden, so dass verschiedene Zuständigkeiten und auszutauschende Dokumente visualisiert werden können.
- **Verteilungsdiagramme**: Diese Diagramme sind für eingebettete Systeme wichtig: sie beschreiben die „Ausführungs-Architektur“ der Hardware- und Software-Knoten eines Systems.
- **Paket-Diagramme**: Paketdiagramme zeigen die Partitionierung der Software in verschiedene Pakete. Sie ähneln den *Modul-Charts* in StateMate.

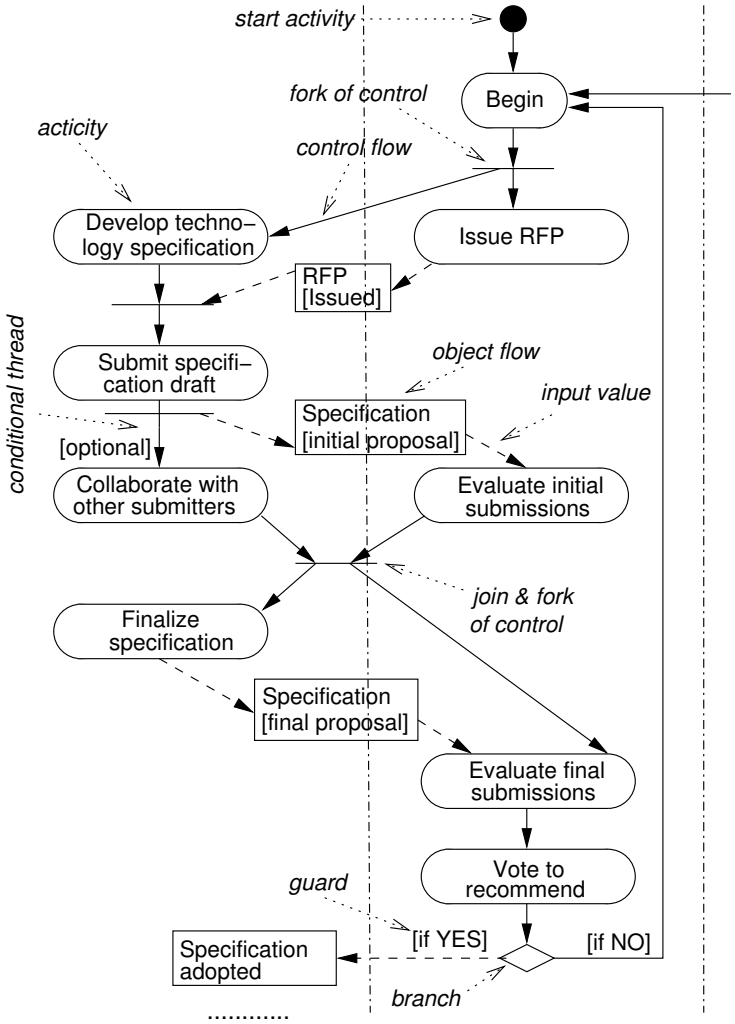


Abb. 2.41. Aktivitäts-Diagramm [Kobryn, 2001]

- ***Use-Case*-Diagramme**: Diese Diagramme stellen typische Anwendungsszenarien des zu entwickelnden Systems dar. Abb. 2.42 zeigt Szenarien für die Kunden einer Bank.
- **Klassen-Diagramme**: Diese Diagramme beschreiben die Vererbungsrelationen von Klassen.
- **Zeitverlaufs-Diagramme** werden verwendet, um Zustandsänderungen eines Objekts über der Zeit darzustellen.

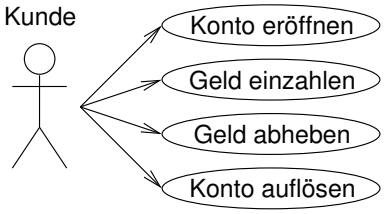


Abb. 2.42. Beispiel für einen *Use-Case*

- **Kommunikations-Diagramme** (in UML 1.x: *Collaboration diagram*): Diese Graphen stellen Klassen und deren Beziehungen sowie die zwischen ihnen ausgetauschten Nachrichten dar.
- **Komponenten-Diagramme** stellen die Komponenten dar, die in Anwendungen oder in Systemen verwendet werden.
- **Objekt-Diagramme**, **Interaktions-Übersichts-Diagramme**, **Kompositions-Struktur-Diagramme**: Diese drei Diagrammtypen werden seltener verwendet. Einige sind eigentlich nur Sonderfälle von anderen Diagrammen.

Aktuell verfügbare Programme, z.B. von ILogix (http://www.ilogix.com) erlauben eine gewisse Konsistenzüberprüfung zwischen den verschiedenen Diagrammtypen. Eine vollständige Überprüfung scheint allerdings unmöglich zu sein. Ein Grund dafür liegt in der ursprünglich nicht genau definierten Semantik von UML. Es wurde argumentiert, dass dies bewusst getan wurde, da man sich in der frühen Entwurfsphase eines Systems nicht mit präzisen Semantikfragen auseinandersetzen möchte. Als Folge kann man in UML nur dann präzise und ausführbare Spezifikationen erstellen, wenn man es mit einer anderen, ausführbaren Sprache kombiniert. Es gibt solche Kombinationen von UML mit SDL [Telelogic, 1999] und mit C++. Parallel dazu gibt es allerdings auch erste Versuche, eine genaue Semantik für UML zu definieren.

In diesem Buch soll UML nicht weiter besprochen werden, da bereits alle wichtigen Diagrammtypen erwähnt wurden und da ein gewisses Risiko der Überlappung mit dem Bereich der Software-Technologie besteht. Trotzdem ist es interessant, dass z.B. Petrinetze, die ursprünglich alles andere als populär waren, Jahrzehnte nach ihrer Erfindung plötzlich häufig verwendete Hilfsmittel bei der Spezifikation von Systemen sind, weil sie in UML aufgenommen wurden.

2.9 Prozessnetze

2.9.1 Taskgraphen

Prozessnetze wurden bereits im Zusammenhang mit Berechnungsmodellen erwähnt. Prozessnetze werden mit Hilfe von Graphen modelliert. Wir werden die beiden Bezeichnungen **Taskgraph** und **Prozessnetz** synonym verwenden, obwohl die Begriffe ursprünglich von verschiedenen Gruppen geprägt wurden. Knoten in Graphen repräsentieren Prozesse, die bestimmte Operationen ausführen. Prozesse bilden Eingabedatenströme auf Ausgabedatenströme ab. Sie werden häufig in Hochsprachen implementiert. Typische Prozesse enthalten (eventuell nicht-terminierende) Schleifen. In jedem Durchlauf einer solchen Schleife lesen sie Daten von ihren Eingängen, verarbeiten diese Daten und erzeugen Ausgabedatenströme. Kanten beschreiben die Relationen zwischen Prozessen. Wir wollen diese Graphen nun etwas genauer definieren.

Die offensichtlichste Beziehung zwischen Prozessen ist die kausale Abhängigkeit: Viele Prozesse können erst dann ausgeführt werden, wenn andere Prozesse beendet worden sind. Diese Abhängigkeit wird typischerweise in **Abhängigkeitsgraphen** dargestellt. Abb. 2.43 zeigt einen Abhängigkeitsgraphen für eine Menge von Prozessen oder Tasks.

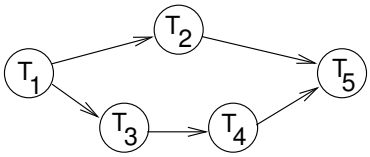


Abb. 2.43. Abhängigkeitsgraph

Definition: Ein Abhängigkeitsgraph ist ein gerichteter Graph $G = (V, E)$ wobei $E \subseteq V \times V$ eine partielle Ordnung darstellt. Wenn $(v_1, v_2) \in E$, dann heißt v_1 ein **direkter Vorgänger** von v_2, und v_2 heißt **direkter Nachfolger** von v_1. Angenommen E^* sei die transitive Hülle von E. Wenn $(v_1, v_2) \in E^*$, dann heißt v_1 **Vorgänger** von v_2 und v_2 heißt **Nachfolger** von v_1.

Solche Abhängigkeitsgraphen bilden spezielle Fälle von Taskgraphen. Taskgraphen stellen die Relationen zwischen einer Menge von Prozessen dar. Taskgraphen können mehr Informationen enthalten als der in Abb. 2.43 gezeigte. Beispielsweise können Taskgraphen die folgenden Erweiterungen von Abhängigkeitsgraphen enthalten:

1. ***Timing*-Informationen**: Tasks können Ankunftszeiten, *Deadlines*, Perioden und Ausführungszeiten haben. Um diese bei der Ablaufplanung, dem Scheduling, zu berücksichtigen, ist es hilfreich, wenn diese Informationen in den Taskgraphen eingefügt werden. Wir verwenden die Notation

aus dem Buch von Liu [Liu, 2000], um in Abb. 2.44 mögliche Ausführungsintervalle darzustellen. Die erste Zahl bezeichnet die Ankunftszeit, die zweite die Zeit für die spätestmögliche Beendigung. Die Tasks T_1 bis T_3 werden dabei als unabhängig angenommen.

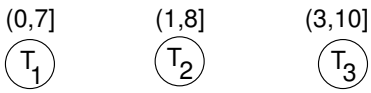


Abb. 2.44. Taskgraphen mit *Timing*-Information

Es kann wesentlich komplexere Kombinationen von Timing und Abhängigkeitsrelationen geben.

2. **Unterscheidung von verschiedenen Relations-Typen** zwischen den Tasks: Präzedenz-Relationen modellieren lediglich Einschränkungen bezüglich der Ausführungsreihenfolge. Auf einer detaillierteren Ebene kann es hilfreich sein, das Scheduling von der Kommunikation zwischen den Tasks zu trennen. Kommunikation kann auch durch Kanten beschrieben werden, es können allerdings weitere Informationen zu jeder dieser Kanten bekannt sein, etwa der Zeitpunkt der Kommunikation und die ausgetauschte Informationsmenge. Präzedenzrelationen können als Kanten eines anderen Typs beibehalten werden, da es Situationen geben kann, in denen Prozesse nacheinander ausgeführt werden müssen, obwohl sie keine Informationen miteinander austauschen.

   In Abb. 2.43 sind Ein- und Ausgabe (I/O) nicht explizit dargestellt. Implizit wird angenommen, dass Knoten ohne Vorgänger im Taskgraphen zu einer bestimmten Zeit eine Eingabe erhalten. Weiter wird angenommen, dass sie nach einer gewissen Zeit eine Ausgabe für den Nachfolgerknoten erzeugen. Diese Ausgabe ist erst dann verfügbar, wenn die Ausführung der Task abgeschlossen ist. Es ist oft nützlich, Ein- und Ausgabe genauer und explizit zu beschreiben. Um dies zu erreichen, wird ein dritter Relationstyp benötigt. Wir verwenden die Notation von Thoen [Thoen und Catthoor, 2000]. Danach stellen Kreise mit Punkt Ein- und Ausgabe dar, wie dies in Abb. 2.45 zu sehen ist. Die Kanten von bzw. zu diesen Knoten beschreiben die erwähnte dritte Relation.

3. **Exklusiver Zugriff auf Ressourcen:** Tasks können einen exklusiven Zugriff auf eine Ressource anfordern, etwa auf ein Ein-/Ausgabegerät oder auf einen Speicherbereich zur Kommunikation. Informationen über eventuell notwendige exklusive Zugriffe sollten beim Scheduling berücksichtigt werden. Durch die Verwendung dieser Information kann man z.B. das Problem der Prioritätsumkehr vermeiden (s. Seite 156). Solche Informationen über exklusiven Zugriff können auch in Taskgraphen dargestellt werden.

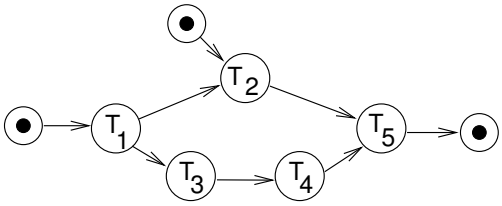


Abb. 2.45. Taskgraph mit Eingabe- und Ausgabe-Knoten und -Kanten

4. **Periodische Abläufe:** Viele Tasks, insbesondere im Bereich der digitalen Signalverarbeitung, sind periodisch. Man muss also genauer unterscheiden zwischen einer Task und ihrer Ausführung – letztere wird häufig als **Job** bezeichnet [Liu, 2000]. Taskgraphen für solche Schedules sind unendlich. Abb. 2.46 zeigt einen Taskgraphen mit den Jobs J_{n-1} bis J_{n+1} einer periodischen Task.

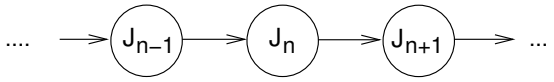


Abb. 2.46. Taskgraph mit Jobs (Ausführungen) einer periodischen Task

5. **Hierarchische Knoten**: Die Komplexität der Berechnungen, die in einem Knoten ausgeführt werden, kann sehr unterschiedlich sein. Einerseits können die beschriebenen Programme sehr groß sein und Tausende von Codezeilen enthalten. Andererseits kann man Programme in kleine Programmteile aufteilen, so dass im Extremfall jeder Knoten nur genau einer Operation entspricht. Die Komplexität von Knoten im Graphen heißt **Granularität**. Auf die Frage, welche Granularität man verwenden sollte, gibt es keine allgemeingültige Antwort. Für einige Zwecke sollte die Granularität so grob wie möglich sein, etwa wenn die Knoten die Prozesse in einem Echtzeitbetriebssystem darstellen. In diesem Fall sollten die Knoten große Programmteile enthalten, um die Anzahl der Kontextwechsel zwischen den Prozessen gering zu halten. Für andere Anwendungen kann es sinnvoll sein, wenn jeder Knoten nur eine Operation enthält, etwa wenn die Knoten entweder auf Hard- oder auf Software ausgeführt werden sollen. Wenn eine bestimmte Operation (z.B. die häufig vorkommende Diskrete Cosinus-Transformation oder DCT) auf eine Spezialhardware abgebildet werden kann, dann sollte sie nicht in einem großen Knoten mit vielen anderen Operationen versteckt sein. Vielmehr sollte die DCT als einzelner Knoten modelliert werden. Um häufige Wechsel der Granularität zu vermeiden, sind hierarchische Knoten nützlich. Auf einer hohen hierarchischen Ebene können die Knoten dann z.B. komplexe Aufgaben beschreiben, auf einer niedrigeren Ebene z.B. lineare Codesequenzen oder, noch

tiefer, einzelne arithmetische Operationen. Abb. 2.47 zeigt eine hierarchische Version des Abhängigkeitsgraphen aus Abb. 2.43, wobei hierarchische Knoten durch Rechtecke dargestellt werden.

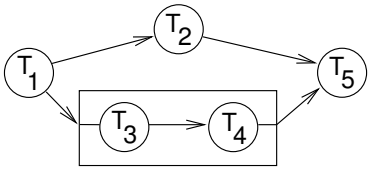


Abb. 2.47. Hierarchischer Taskgraph

Von Thoen [Thoen und Catthoor, 2000] wurde ein sehr umfassendes Taskgraph-Modell, der sogenannte ***Multi-Thread Graph*** (MTG) vorgeschlagen. MTGs lassen sich so definieren:

Definition: Ein *Multi-Thread Graph* M ist definiert als 11-Tupel $(O, E, V, D, \vartheta, \iota, \Lambda, \mathcal{E}^{lat}, \mathcal{E}^{resp}, \nabla^{i}, \nabla^{av})$, wobei

- O ist die Menge von Operationsknoten. Diese können verschiedene Typen haben, u.a. *thread, hierarchical thread, or, even, synchro, semaphore, source* und *sink*. MTGs haben einzelne Quellknoten vom Typ *source* und Senken vom Typ *sink*. Mit Knoten vom Typ *or* kann die Situation modelliert werden, dass nur eine Task aus einer Menge von Tasks benötigt wird, um die nächste Task zu starten. *Event*-Knoten modellieren externe Eingaben. Mit *Semaphoren* kann der gegenseitige Ausschluss modelliert werden. *Synchro*-Knoten dienen der Rückmeldung an die Umgebung.
- E ist die Menge der Kontrollflusskanten. Diese Kanten können u.a. Informationen über zeitliche Abläufe enthalten.
- V, D und ϑ bezeichnen Variablenzugriffe, auf die wir hier nicht näher eingehen.
- ι ist die Menge der Ein-/Ausgabeknoten.
- Λ gibt Ausführungslatenz-Intervalle für alle Threads an.
- $\mathcal{E}^{lat}$, $\mathcal{E}^{resp}$, ∇^{i} und ∇^{av} sind Zeitbedingungen.

Wie man anhand der Definition sehen kann, sind in MTGs fast alle vorgestellten Erweiterungen von einfachen Präzedenzgraphen enthalten.

2.9.2 Asynchroner Nachrichtenaustausch

Beim asynchronen Nachrichtenaustausch findet die Kommunikation zwischen den Prozessen über Puffer statt. Diese Puffer werden üblicherweise als FIFO-Warteschlangen mit theoretisch unendlicher Größe modelliert.

Kahn Prozessnetzwerke

Kahn Prozessnetzwerke (KPN) [Kahn, 1974] sind ein Spezialfall solcher Prozessnetzwerke. Schreibvorgänge in KPNs sind nicht-blockierend. Leseoperationen werden blockiert, wenn ein Prozess versucht, etwas aus einer leeren FIFO-Warteschlange zu lesen. Diese FIFOs sind die einzige Kommunikationsmöglichkeit zwischen Prozessen. Nur ein einziger Prozess darf aus einer bestimmten Warteschlange lesen. Wenn ein Prozess seine Ausgabedaten also an mehrere Nachfolger schicken will, müssen die Daten innerhalb des Prozesses dupliziert werden. Im Allgemeinen muss die Ablaufplanung bei Kahn-Prozessen zur Laufzeit vorgenommen werden, da ihr Laufzeitverhalten schwer vorherzusagen ist. Die Frage, ob alle realen, in der Größe beschränkten FIFOs für ein bestimmtes KPN-Modell wirklich ausreichend dimensioniert sind, ist im allgemeinen Fall unentscheidbar. Trotzdem gibt es praktisch anwendbare Algorithmen [Kienhuis et al., 2000].

Synchroner Datenfluss

Das synchrone Datenfluss-Modell (SDF) [Lee und Messerschmitt, 1987] lässt sich am besten anhand seiner graphischen Darstellung erklären. Abb. 2.48 (links) zeigt einen synchronen Datenflussgraphen. Der Graph ist gerichtet, die Knoten A und B stellen die Berechnungen * und + dar. SDF-Graphen, wie alle Datenflussgraphen, zeigen die auszuführenden Operationen und ihre Abhängigkeiten, aber nicht die Reihenfolge, in der die Berechnungen ausgeführt werden müssen – im Gegensatz zu sequentiellen Sprachen, wie etwa der Programmiersprache C. Eingaben für SDF-Graphen bestehen aus einem unendlichen Strom von Daten. Kanten müssen immer dann eingefügt werden, wenn zwischen zwei Knoten eine Datenabhängigkeit besteht.

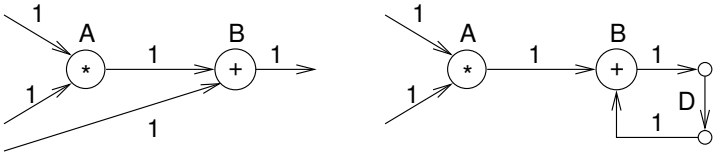


Abb. 2.48. Graphische Darstellung von synchronem Datenfluss

Die Ausführung einer Berechnung innerhalb eines Knotens wird als „feuern“ bezeichnet. Jedesmal, wenn eine Berechnung feuert, wird eine Anzahl von Daten (repräsentiert durch sogenannte *Token* oder Marken) verbraucht und erzeugt. Beim synchronen Datenfluss ist die Anzahl von erzeugten oder verbrauchten Marken fest und zur Entwurfszeit bekannt.

Konstante Kantenbeschriftungen geben die zugehörige Anzahl von Marken an. Diese Konstanten erlauben die Modellierung von Signalverarbeitungsan-

wendungen, in denen mehrere Datenraten vorkommen (bei denen also einige Signale mit einer ganzzahligen vielfachen Frequenz von anderen Signalen erzeugt werden). Der Ausdruck **synchroner** Datenfluss drückt aus, dass die Marken von ankommenden Kanten auf synchrone Art und Weise verbraucht werden – also **alle genau zum selben Zeitpunkt**. Der Ausdruck **asynchroner** Nachrichtenaustausch zeigt dagegen, dass die Marken in FIFO-Puffern zwischengespeichert werden. Die Tatsache, dass beim Feuern eine feste Anzahl von Marken erzeugt und verbraucht wird, erlaubt die Bestimmung einer Ausführungsreihenfolge und des Speicherbedarfs zur Übersetzungszeit. So kann ein komplexes Scheduling zur Laufzeit vermieden werden. SDF-Graphen können Verzögerungen enthalten, die mit dem Symbol D auf einer Kante gekennzeichnet werden (s. Abb. 2.48 rechts). SDF-Graphen können in periodische Schedules für Einzel- und auch für Multi-Prozessorsysteme umgesetzt werden (s. etwa [Pino und Lee, 1995]). Eine erlaubte Ausführungsreihenfolge für das einfache Beispiel in Abb. 2.48 besteht aus der Sequenz (A, B) (unendlich oft wiederholt). Die Sequenz (A,A,B) (A wird doppelt so oft ausgeführt wie B) wäre nicht erlaubt, da sich in diesem Fall eine unendliche Anzahl von Marken im impliziten Puffer zwischen A und B ansammeln würden.

2.9.3 Synchroner Nachrichtenaustauch

CSP

CSP [Hoare, 1985] (*communicating sequential processes*, kommunizierende sequentielle Prozesse) ist eine der ersten Sprachen, die Mechanismen für Interprozess-Kommunikation enthalten. Die Kommunikation basiert auf Kanälen.

Beispiel:

```
process A                          process B
.....                              ......
var a ..                           var b ...
 a := 3;                            ...
 c!a; -- Ausgabe auf Kanal c       c?b; -- Eingabe von Kanal c
end;                               end;
```

Beide Prozesse warten, bis der jeweils andere Prozess bei der Eingabe- bzw. Ausgabe-Anweisung angekommen ist. Diese Form der Kommunikation bezeichnet man als ***Rendez-Vous*-Konzept** oder als **blockierende Kommunikation**.

CSP war die Grundlage für die Sprache OCCAM, die als Programmiersprache für **Transputer** [Thiébaut, 1995] eingeführt wurde.

ADA

In den achtziger Jahren des zwanzigsten Jahrhunderts bemerkte das US-amerikanische Verteidigungsministerium, dass sowohl die Verlässlichkeit als auch die Wartbarkeit der Software in den militärischen Ausrüstungsgegenständen zu einem großen Problem werden könnten, wenn nicht strenge Regelungen eingeführt würden. Es wurde entschieden, dass alle Software in der gleichen Echtzeitsprache geschrieben werden sollte. Die Anforderungen an solch eine Sprache wurden formuliert. Da keine existierende Sprache alle Anforderungen erfüllte, wurde eine neue Sprache entworfen. Die letztlich angenommene Sprache basiert auf PASCAL und heißt ADA (nach Ada Lovelace, die als die erste Programmiererin gilt). ADA'95 [Kempe, 1995], [Burns und Wellings, 2001] ist eine objektorientierte Erweiterung des Original-Sprachstandards.

ADA hat die interessante Eigenschaft, dass man verschachtelte Prozesse (die in ADA Tasks heißen) deklarieren kann. Eine Task wird gestartet, wenn der Kontrollfluss in den Bereich verzweigt, in dem die Task deklariert wurde. Das folgende Beispiel ist aus Burns et al. [Burns und Wellings, 1990] entnommen:

```
procedure beispiel1 is
   task a;
   task b;
   task body a is
     -- lokale Deklarationen für a
     begin
       -- Anweisungen für a
     end a;
   task body b is
     -- lokale Deklarationen für b
     begin
       -- Anweisungen für b
     end b;
  begin
   -- Tasks a und b starten, bevor die erste Anweisung
   -- von beispiel1 ausgeführt wird
  end;
```

Das Kommunikationskonzept von ADA ist ein weiteres Schlüssel-Konzept. Es basiert auf dem *Rendez-Vous* Paradigma. Wenn zwei Tasks Informationen austauschen wollen, muss diejenige Task, die den „Treffpunkt" zuerst erreicht, warten bis auch der Kommunikationspartner den entsprechenden Punkt im Kontrollfluss erreicht hat. In der ADA-Syntax werden Prozeduren verwendet,

um Kommunikation zu beschreiben. Prozeduren, die von anderen Tasks aufgerufen werden können, müssen mit dem Schlüsselwort **entry** gekennzeichnet werden. Beispiel [Burns und Wellings, 1990]:

```
task screen_out is
  entry call (val : character; x, y : integer);
end screen_out;
```

Task screen_out enthält eine Prozedur call, die von anderen Prozessen aufgerufen werden kann. Eine andere Task kann diese Prozedur aufrufen, indem sie ihr den Namen der Task voranstellt:

```
screen_out.call('Z',10,20);
```

Die aufrufende Task muss warten, bis die aufgerufene Task einen Punkt im Kontrollfluss erreicht hat, an dem sie Aufrufe von anderen Tasks annimmt. Dieser Punkt wird mit dem Schlüsselwort **accept** markiert:

```
task body screen_out is
  ...
  begin
    accept call (val : character; x, y : integer) do ...
  end call;
  ...
end screen_out;
```

Offensichtlich sollte eine Task screen_out auch auf mehrere Aufrufe gleichzeitig warten können. Hierzu wird die ADA **select**-Anweisung verwendet. Beispiel:

```
task screen_output is
  entry call_ch(val:character; x, y: integer);
  entry call_int(z, x, y: integer);
end screen_out;
task body screen_output is
...
  select
    accept call_ch ...  do...
    end call_ch;
  or
    accept call_int ... do ...
    end call_int;
  end select; ...
```

In diesem Fall wartet screen_out, bis entweder call_ch oder call_int aufgerufen wird.

ADA ist die Sprache der Wahl für die Programmierung vieler militärischer Ausrüstungsgegenstände, die in der westlichen Hemisphäre produziert wurden.

Auch hier werden Prozessnetzwerke nicht explizit als Graphen dargestellt, aber man kann diese Graphen aus der textuellen Darstellung erzeugen.

2.10 Java

Java wurde als plattformunabhängige Sprache entworfen. Sie kann auf jeder Maschine ausgeführt werden, für die ein Interpreter der internen Bytecode-Darstellung von Java-Programmen verfügbar ist. Dieser Bytecode ist sehr kompakt, was im Vergleich zu den üblichen binären Maschinencodes zu geringeren Speicheranforderungen führt. Dies ist ein offensichtlicher Vorteil für *System on a Chip*-Anwendungen, bei denen nur sehr wenig Speicherplatz zur Verfügung steht.

Außerdem wurde Java als sichere Sprache entworfen. Viele potentiell gefährliche Eigenschaften und Fähigkeiten von C oder C++ (wie etwa Zeiger-Arithmetik) sind in Java nicht vorhanden. Daher erfüllt Java die Sicherheitsanforderungen zum Entwurf eingebetteter Systeme. Java unterstützt Ausnahmebehandlungen, wodurch die Behandlung von Laufzeitfehlern vereinfacht wird. Es gibt keine Gefahr von Speicherlecks aufgrund fehlender Speicherfreigaben, da Java den Speicherplatz automatisch verwaltet und organisiert (*Garbage-Collection*). Dieser Mechanismus verhindert potentielle Probleme bei Systemen, die über Monate oder sogar Jahre hinweg ohne Neustart durchlaufen müssen. Java erfüllt auch die Anforderungen an Nebenläufigkeit, da es *Threads* unterstützt[7].

Zusätzlich lassen sich Java-Applikationen relativ schnell implementieren, da Java eine objektorientierte Sprache ist und Java-Entwicklungsumgebungen mit mächtigen Bibliotheken ausgeliefert werden.

Allerdings ist Standard-Java nicht hauptsächlich für Echtzeitanwendungen entworfen worden, und einige Merkmale, die es zu einer idealen Programmiersprache für eingebettete Systeme machen würden, fehlen:

- Die Größe der Java Runtime-Bibliotheken muss zur eigentlichen Größe der Anwendung hinzuaddiert werden. Diese Bibliotheken können recht groß sein. Folglich profitieren nur wirklich große Anwendungen von der kompakten Bytecode-Darstellung der Applikation.

[7] Allerdings zeigt Lee [Lee, 2006], dass *Threads* ein sehr problematisches Mittel zur Beschreibung von Nebenläufigkeit sind.

- Für viele eingebettete Anwendungen ist es notwendig, direkten Einfluss auf die Ein- und Ausgabe-Geräte zu haben (s. Seite 16). Aus Sicherheitsgründen gibt es diese direkte Kontrolle über Geräte in Java nicht.
- Die automatische Speicherverwaltung (*Garbage Collection*) benötigt Rechenzeit. In Standard-Java gibt es keine Möglichkeit, festzustellen, wann die *Garbage Collection* gestartet wird. Das macht es schwierig, die längstmögliche Ausführungszeit (*worst case execution time*) zu bestimmen. Es sind lediglich sehr konservative, pessimistische Aussagen über die maximale Laufzeit möglich.
- Java trifft keine Aussage darüber, in welcher Reihenfolge mehrere ausführbereite Threads gestartet werden. Dadurch werden die *worst case execution time*-Schätzungen noch konservativer.

Erste Vorschläge, um diese Probleme zu lösen, kamen von Nilsen [Nilsen, 2004] und sahen u.a. eine Hardware-Unterstützung für die *Garbage Collection*, ein Austauschen des Laufzeit-Schedulers sowie das Markieren einiger Speichersegmente vor.

Die wichtigsten Java Entwicklungsumgebungen sind die *Java Enterprise Edition* (J2EE), die *Java Standard Edition* (J2SE), die *Java Micro Edition* (J2ME) sowie *CardJava*. Letzteres ist eine stark reduzierte Java-Version mit besonderer Berücksichtigung der Sicherheit von *SmartCard*-Anwendungen. J2ME ist die relevante Java-Umgebung für alle anderen Typen von eingebetteten Systemen. Zwei Bibliotheks-Profile wurden für J2ME definiert: CDC und CDLC. CDLC wird für Mobiltelefone verwendet und unterstützt das sogenannte MIDP 1.0/2.0 *Application Programming Interface* (API). CDC wird beispielsweise in Fernsehern und in modernen Handys mit vielen Funktionen verwendet. Eine Echtzeit-Erweiterung für Java heißt „*Real-time specification for JAVA* (JSR-1)“ [Java Community Process, 2002] und wird von TimeSys [TimeSys Inc., 2003] unterstützt. Eine weitere ist „*High Integrity Java*“ (HIJA) (http://www.hija.info).

2.11 VHDL

2.11.1 Einführung

Sprachen, die Hardware beschreiben, wie etwa VHDL, werden **Hardware-Beschreibungssprachen** (*Hardware Description Languages* (HDL) genannt. Bis in die achtziger Jahre wurden die meisten Systeme mit Hilfe graphischer HDLs beschrieben. Der meistverwendete Baustein einer solchen Darstellung war ein einzelnes Gatter. Zusätzlich zu einer graphischen Darstellung kann man auch eine textuelle HDL-Beschreibung erstellen. Die Stärke

von textuellen Darstellungen liegt in der Möglichkeit, relativ leicht komplexe Berechnungen mit Variablen, Schleifen, Funktionsparametern und Rekursionen darstellen zu können. Folglich wurden graphische Darstellungen fast vollständig von textuellen HDLs ersetzt, als die digitalen Systeme im Lauf der achtziger Jahre immer komplexer wurden. Die textuellen HDL-Darstellungen waren ursprünglich ein Forschungsthema an Universitäten. Mermet et al. [Mermet et al., 1998] geben einen Überblick über die in Europa im Laufe der achtziger Jahre entwickelten Sprachen. MIMOLA war eine dieser Sprachen, und der Autor dieses Buches hat zu ihrem Entwurf und ihrer Anwendung beigetragen [Marwedel und Schenk, 1993]. Textuelle Sprachen wurden beliebt, als VHDL und der Konkurrent Verilog (s. Seite 81) eingeführt wurden. VHDL wurde im Rahmen des VHSIC-Programms des US-amerikanischen Verteidigungsministeriums entwickelt. VHSIC steht für *Very High Speed Integrated Circuits*[8]. Ursprünglich wurde der Entwurf von VHDL (*VHSIC hardware description language*) von drei Firmen durchgeführt: IBM, Intermetrics und Texas Instruments. Eine erste Version von VHDL wurde 1984 veröffentlicht. Später wurde VHDL dann von der IEEE unter dem Namen IEEE 1076 standardisiert. Die erste IEEE-Version von VHDL wurde im Jahre 1987 standardisiert und in den Jahren 1992, 1997 und 2002 aktualisiert.

Ein Hauptunterscheidungsmerkmal zwischen üblichen Software-Sprachen und Hardwarebeschreibungssprachen ist die Notwendigkeit zur Beschreibung der Nebenläufigkeit verschiedener Hardwarekomponenten. VHDL verwendet **Prozesse**, um diese parallele Ausführung zu beschreiben. Jeder Prozess modelliert eine Komponente einer möglicherweise nebenläufigen Hardware. Für einfache Komponenten kann ein einzelner Prozess ausreichend sein. Um komplexere Hardwarekomponenten zu modellieren, benötigt man in der Regel mehrere Prozesse. Prozesse kommunizieren über **Signale** miteinander. Signale entsprechen in etwa den physikalischen Verbindungen in der Hardware, also z.B. Leitungen oder Drähten. Ein weiterer Unterschied zwischen Software-Programmiersprachen und HDLs ist die Modellierung der Zeit. VHDL unterstützt, ebenso wie alle anderen HDLs, die Modellierung der Zeit.

Der Entwurf von VHDL nahm ADA als Ausgangspunkt, da beide Sprachen für das US-amerikanische Verteidigungsministerium entworfen wurden. Weil ADA auf PASCAL basiert, hat VHDL einige syntaktische Eigenheiten von PASCAL übernommen. Allerdings ist die Syntax von VHDL im Allgemeinen sehr viel komplexer, so dass man aufpassen muss, um sich nicht verwirren zu lassen. In diesem Buch werden nur einige Konzepte von VHDL vorgestellt, die auch in anderen Sprachen vorkommen oder die einfach nützlich sind. Eine vollständige Beschreibung des VHDL-Sprachstandards ist nicht Bestandteil dieses Buches. Ihn kann man von der IEEE erhalten (s. [IEEE, 1992]).

[8] Die Entwicklung des Internets war auch ein Teil dieses VHSIC-Programms.

2.11.2 *Entities* und *Architectures*

In VHDL heißt jeder modellierte Baustein ***design entity*** oder **VHDL *entity***. Entities bestehen aus zwei Bestandteilen: einer ***Entity*** **Deklaration** und einer (oder mehrerer) **Architekturen** (***architectures*** (s. Abb. 2.49)). Für jede *Entity* wird im Standardfall die zuletzt analysierte Architektur verwendet. Die Verwendung von anderen Architekturen kann auch angegeben werden.

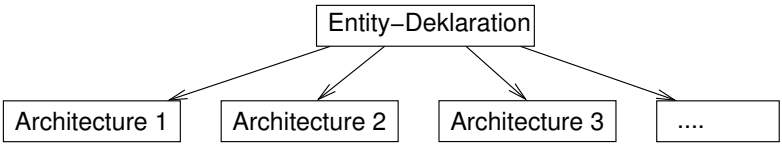


Abb. 2.49. Eine *Entity* besteht aus einer *Entity*-Deklaration und Architekturen

Als Beispiel betrachten wir einen Volladdierer. Volladdierer haben drei Eingänge und zwei Ausgänge (s. Abb. 2.50).

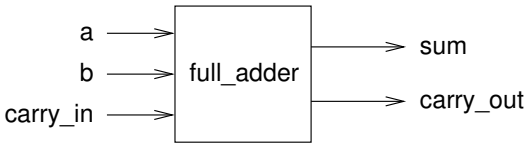


Abb. 2.50. Ein Volladdierer und seine Schnittstellen-Signale

Eine *Entity*-Deklaration, die Abb. 2.50 entspricht, könnte etwa wie folgt aussehen:

```
entity full_adder is                    -- Entity-Deklaration
   port (a, b, carry_in: in Bit;        -- Eingabe-Ports
         sum, carry_out: out Bit);      -- Ausgabe-Ports
end full_adder;
```

Architekturen bestehen aus Architektur-Köpfen und Architektur-Rümpfen. Man kann verschiedene Stile von Rümpfen unterscheiden, hauptsächlich strukturelle Rümpfe und Verhaltens-Rümpfe. Wir zeigen die Unterschiede zwischen diesen beiden Modellierungsarten am Beispiel des Volladdierers. Verhaltensrümpfe enthalten nur die Informationen, die benötigt werden, um aus den Eingabesignalen die Ausgabesignale und den internen Zustand (wenn es einen gibt) zu berechnen. Dies beinhaltet auch das zeitliche Verhalten. Das folgende Beispiel zeigt einen Verhaltensrumpf („<=“ kennzeichnet in VHDL die Signal-Wertzuweisung):

```
architecture behavior of full_adder is  -- Architektur
 begin
   sum       <= (a xor b) xor carry_in after 10 Ns;
   carry_out <= (a and b) or (a and carry_in) or
                (b and carry_in) after 10 Ns;
 end behavior;
```

VHDL-basierte Simulatoren können die Ausgabesignale graphisch als Wellenformen darstellen. Diese ergeben sich, wenn Werte an die Eingabeports des Volladdierers gelegt werden.

Im Gegensatz zu Verhaltensrümpfen beschreiben strukturelle Architektur-Rümpfe die Art und Weise, in der Entities aus einfacheren Entities zusammengebaut sind. Beispielsweise kann der Volladdierer als *Entity* modelliert werden, die aus drei Komponenten besteht (s. Abb. 2.51). Diese Komponenten heißen i1, i2 und i3 und sind vom Typ half_adder oder or_gate.

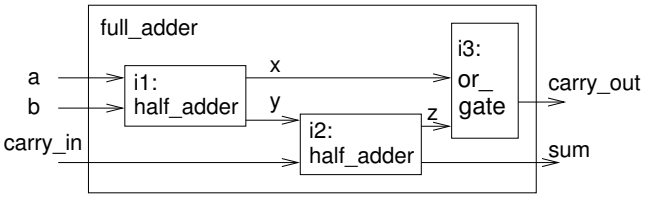


Abb. 2.51. Schematische Darstellung des Struktur-Rumpfes des Volladdierers

In der 1987er-Version von VHDL mussten diese Komponenten in einer sogenannten *component declaration* deklariert werden. Diese Deklaration ist der *Forward*-Deklaration anderer Programmiersprachen sehr ähnlich und dient auch genau dem gleichen Ziel: sie stellt genügend Informationen über eine Komponente zur Verfügung, die zu dem Zeitpunkt evtl. noch nicht in der VHDL-Datenbank gespeichert ist (was bei sogenannten *Top-Down-Designs* passieren kann). Ab der 1992er-Version von VHDL sind solche Deklarationen nicht mehr notwendig, wenn die benötigten Komponenten bereits in der Komponenten-Datenbank abgelegt wurden.

Verbindungen zwischen lokalen Komponenten und den Ports (Schnittstellen) der *Entity* werden mit Hilfe von ***Port Maps*** beschrieben. Der folgende VHDL-Code beschreibt den Struktur-Rumpf aus Abb. 2.51:

```
architecture structure of full_adder is  -- Architektur-Kopf
   component half_adder
     port (in1, in2 : in Bit; carry  :out Bit; sum :out Bit);
   end component;
```

```
 component or_gate
   port(in1, in2:in Bit; o:out Bit);
 end component;
 signal x, y, z: Bit;            -- lokale Signale
begin                            -- Port-Map
 i1: half_adder                  -- Instantiierung von half_adder i1
     port map (a, b, x, y);      -- Verbindungen zwischen Ports
 i2: half_adder port map (y, carry_in, z, sum);
 i3: or_gate    port map (x, z, carry_out);
end structure;
```

2.11.3 Mehrwertige Logik und IEEE 1164

In diesem Buch beschränken wir uns auf die Beschreibung von eingebetteten Systemen, die mit binärer Logik implementiert werden. Trotzdem ist es ratsam oder sogar notwendig, mehr als zwei Signalwerte zur Modellierung solcher Systeme zu verwenden. Beispielsweise könnte ein System elektrische Signale unterschiedlicher Stärke enthalten, und die Stärke und der Wert des Signals, das entsteht, wenn man mehrere solcher Signale zusammenschaltet, müssen berechnet werden. Im Folgenden werden wir deshalb zwischen dem **Wert** und der **Stärke** eines Signals unterscheiden. Während Ersteres eine Abstraktion der Signalspannung darstellt, ist Letzteres eine Abstraktion der Impedanz (des Innenwiderstands) der Spannungsquelle. Wir werden diskrete Wertemengen verwenden, um die Signalwerte und Signalstärken zu beschreiben. Unbekannte elektrische Signale werden durch spezielle Signalwerte modelliert.

In der Praxis verwenden elektronische Entwurfssysteme eine Vielzahl von Wertemengen. Einige Systeme erlauben nur zwei Werte, andere 9 oder sogar 46. Das Ziel bei der Entwicklung solcher diskreter Wertemengen ist es, das Lösen von Netzwerk-Gleichungen (z.B. nach Kirchhoff) zu vermeiden, aber existierende Systeme trotzdem ausreichend genau zu modellieren. Im Folgenden zeigen wir einen Weg, wie man systematisch zu Wertemengen kommt und diese zueinander in Relation setzt. Wir werden die Stärke der elektrischen Signale als Hauptunterscheidungsmerkmal für verschiedene Wertemengen verwenden. Eine systematische Methode, wie man solche Wertemengen aufbauen kann, die CSA-Theorie, wurde von Hayes vorgestellt [Hayes, 1982]. Später werden wir zeigen, wie die normalerweise von VHDL-Modellen verwendete Wertemenge als Spezialfall erzeugt werden kann.

Zwei Logikwerte (1 Signalstärke)

Im einfachsten Fall werden lediglich zwei Logikwerte verwendet. Sie heißen '0' und '1'. Diese Werte werden als gleich stark angenommen. Das bedeutet, wenn zwei Kabel miteinander verbunden werden, von denen eines den Wert '0', das andere den Wert '1' hat, so kann man den Wert des entstehenden Signals nicht bestimmen.

Eine einzige Signalstärke kann ausreichend sein, um ein System zu modellieren, bei dem nie zwei Signale mit unterschiedlichen Werten verbunden werden und in dem sich keine Signale mit unterschiedlichen Stärken in einer bestimmten Hardware-Einheit treffen.

Drei und vier Logikwerte (2 Signalstärken)

In vielen Schaltungen gibt es den Fall, dass ein elektrisches Signal nicht aktiv von einem Ausgang getrieben wird. Dies kann vorkommen, wenn ein Draht weder mit Masse noch mit der Versorgungsspannung oder einem anderen Knoten der Schaltung verbunden ist.

Beispielsweise können Systeme sogenannte *Open-Collector*- (s. Abb. 2.52, links) oder *Tristate*-Ausgänge haben (s. Abb. 2.52, rechts). Durch das Anlegen entsprechender Eingabesignale können solche Ausgänge effektiv von der Leitung abgeklemmt werden[9].

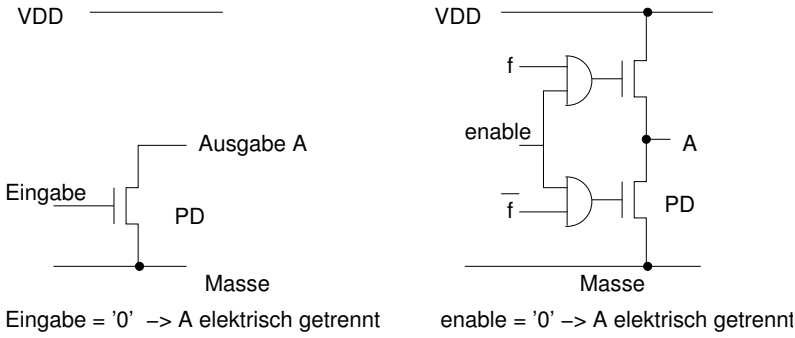


Abb. 2.52. Ausgänge, die effektiv von der Leitung abgeklemmt werden können

Offensichtlich ist die Signalstärke eines solchen abgeklemmten Signals die kleinste Signalstärke, die man sich vorstellen kann. Insbesondere ist diese Signalstärke kleiner als die von '0' und '1'. Außerdem ist der Signalwert eines solchen Signals unbekannt. Diese Kombination von Signalstärke und Signalwert

[9] In der Praxis sind die *Tristate*-Ausgänge meist invertiert.

wird durch den Logikwert mit dem Namen 'Z' dargestellt. Wenn ein Signalwert 'Z' mit irgendeinem anderen Signal zusammengeschaltet wird, dominiert immer das andere Signal. Wenn also beispielsweise zwei *Tristate*-Ausgänge an denselben Bus angeschlossen sind und einer der Ausgänge steuert ein 'Z' bei, so ist der Ergebnis-Wert auf dem Bus immer der Wert des zweiten Ausgangs (s. Abb. 2.53).

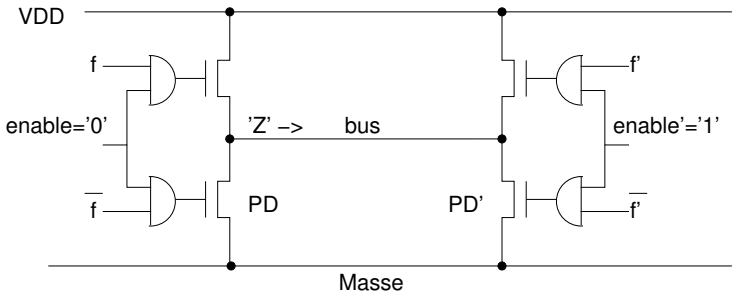


Abb. 2.53. Der rechte Ausgang dominiert den Bus

In VHDL ist jeder Ausgang mit einem sogenannten **Signaltreiber** verbunden. Die Berechnung des Ergebniswertes, wenn mehrere Signaltreiber zu einem Signal beitragen, heißt **Funktionswert-Auflösung (*resolution*)**, die entstehenden Werte werden von **Auflösungs-Funktionen (*resolution functions*)** berechnet.

In den meisten Fällen werden dreiwertige Logikmengen {'0', '1', 'Z'} durch einen vierten Wert mit dem Namen 'X' ergänzt. 'X' beschreibt den unbekannten Signalwert der gleichen Stärke wie '0' und '1'. Genauer ausgedrückt wird der Wert 'X' verwendet, um unbekannte Signale darzustellen, die entweder einen der Werte '0' oder '1' haben oder aber eine Spannung darstellen, die keinem dieser beiden Werte entspricht[10].

Wenn zwei Signaltreiber verbunden werden, lässt sich die benötigte Funktionswert-Auflösungsfunktion leicht berechnen, wenn wir die partielle Ordnung der vier Signalwerte '0', '1', 'Z' und 'X' ausnutzen. Diese partielle Ordnung ist in Abb. 2.54 dargestellt:

Die Kanten in der Abbildung zeigen die Dominanz von Signalwerten. Die Kanten definieren eine Relation $>$ auf den Elementen. Wenn $a > b$, dann dominiert a b (oder b wird von a dominiert). '0' und '1' dominieren 'Z', wohingegen 'X' alle anderen Signalwerte dominiert. Ausgehend von der Relation $>$ definieren wir eine Relation $\geq$. Die Aussage $a \geq b$ gilt genau dann, wenn $a > b$ oder wenn $a = b$.

[10] Es gibt auch andere Interpretationen von 'X', aber die oben genannte Interpretation ist für unsere Zwecke am Besten geeignet.

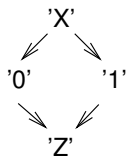


Abb. 2.54. Partielle Ordnung der Wertemenge {'0', '1', 'Z', 'X'}

Wir definieren weiter eine Operation *sup* auf zwei Signalen, die das **Supremum** der beiden Signalwerte zurückliefert. Das Supremum c zweier Signalwerte a und b ist das schwächste Signal, für das $c \geq a$ und $c \geq b$ gilt. So ist z.B. *sup* ('Z','0') = '0', *sup*('Z','1')='1' usw. Die interessante Beobachtung liegt darin, dass Auflösungsfunktionen die Supremumsfunktion *sup* nach der obigen Definition berechnen sollten.

Sieben Signalwerte (3 Signalstärken)

Für viele Schaltungen sind zwei Signalstärken nicht ausreichend. Ein häufig vorkommender Fall, der mehr Werte benötigt ist die Verwendung von sogenannten *depletion transistors* oder Verarmungstransistoren (s. Abb. 2.55).

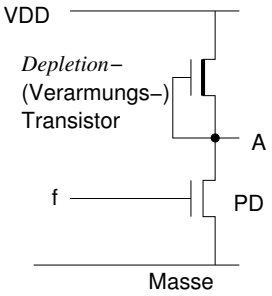


Abb. 2.55. Ausgang mit Verarmungstransistoren

Der Effekt von Verarmungstransistoren ist vergleichbar mit der Verwendung eines Widerstands, der eine hochohmige Verbindung zur Versorgungsspannung VDD herstellt. Verarmungstransistoren dienen genau wie der *Pull-Down-Transistor* PD als Signaltreiber für den Knoten A der Schaltung, und der Signalwert am Knoten A kann mit Hilfe einer Signalwert-Auflösungsfunktion berechnet werden. Der *Pull-Down*-Transistor liefert einen Treiberwert von '0' oder 'Z', je nachdem, wie der Eingabewert von PD ist. Der Verarmungstransistor liefert ein Signal, das schwächer ist als '0' und '1'. Der Signalwert entspricht jedoch einem Wert von '1'. Wir bezeichnen den Signalwert, den der Verarmungstransistor liefert mit 'H', und wir nennen ihn „schwache logische

Eins“. Analog kann es auch „schwache logische Nullen“ geben, die durch 'L' gekennzeichnet werden. Der Wert, der entsteht, wenn man eine Verbindung zwischen 'H' und 'L' herstellt, heißt „schwach logisch undefiniert“, geschrieben als 'W'. Daraus ergeben sich drei Signalstärken und die sieben Signalwerte {'0', '1', 'Z', 'X', 'H', 'L', 'W'}. Die Signalwert-Auflösungsfunktion basiert wieder auf der partiellen Ordnung auf diesen sieben Signalwerten, die in Abb. 2.56 gezeigt wird.

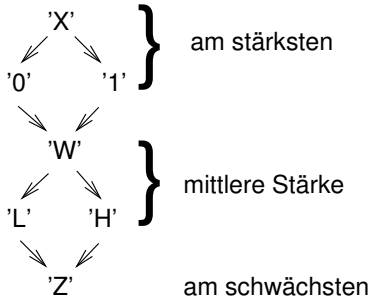


Abb. 2.56. Partielle Ordnung für die Wertemenge {'0', '1', 'Z', 'X', 'H', 'L', 'W'}

Diese Ordnung definiert wieder eine Operation *sup*, die das schwächste Signal zurückliefert, das mindestens so stark ist wie eines der beiden Argumente. Beispielsweise gilt *sup*('H','0') = '0', *sup*('H','Z') = 'H', *sup*('H','L') = 'W'.

'0' und 'L' kennzeichnen den gleichen Signalwert, aber mit unterschiedlicher Signalstärke. Das gleiche gilt für '1' und 'H'. Geräte, welche die Stärke eines Signals erhöhen, heißen **Verstärker**. Geräte, welche die Stärke eines Signals erniedrigen, heißen **Abschwächer** (*attenuator*).

Zehn Signalwerte (4 Signalstärken)

In einigen Fällen sind auch drei Signalstärken nicht ausreichend. Beispielsweise gibt es Schaltungen, die Verbindungsleitungen als Ladungsspeicher verwenden. Solche Leitungen werden während einer bestimmten Betriebsphase mit den Logikwerten '0' oder '1' vorgeladen. Diese Ladungen können den (hochohmigen) Eingang einiger Transistoren steuern. Werden diese Leitungen mit einer Signalquelle (außer einer mit dem Wert 'Z') verbunden, so verlieren sie ihre Ladung und das Signal der zweiten Quelle dominiert.

In Abb. 2.57 wird beispielsweise ein Bus von einem speziellen Ausgang getrieben. Der Bus verfügt über einen Kondensator C mit hoher Kapazität. Während die Funktion f noch auf '0' ist, wird ϕ auf den Wert '1' gesetzt, wodurch der Kondensator C aufgeladen wird. Danach wird ϕ auf '0' gesetzt. Dieses sogenannte Vorladen oder *Precharging* wird verwendet, da das Aufladen eines Busses mit einem Ausgang wie dem in Abb. 2.55 gezeigten ein

langsamer Prozess ist, da der Widerstand des Verarmungstransistors relativ groß ist. Wenn der tatsächliche Wert der Funktion f bekannt wird, und es ist der Signalwert '1', so wird der Bus entladen. Diese Entladung durch normale „*Pull-Down*-Transistoren“ PD erfolgt schneller als das *Precharging*.

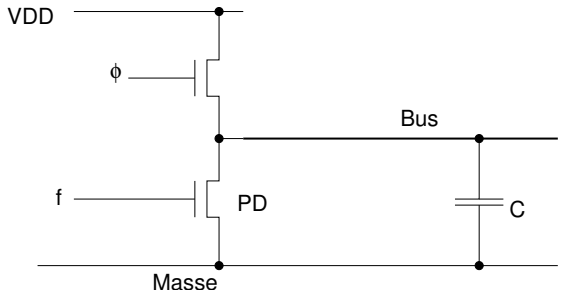


Abb. 2.57. *Pre-charging* (Vorladen) eines Busses

Um solche Fälle modellieren zu können, brauchen wir Signalwerte, die schwächer sind als 'H' und 'L', aber stärker als 'Z'. Wir nennen solche Werte „sehr schwache Signalwerte“ und schreiben sie als 'h' und 'l'. Der zugehörige sehr schwache undefinierte Signalwert heißt 'w'. Damit erhalten wir zehn Signalwerte {'0', '1', 'Z', 'X', 'H', 'L', 'W', 'h', 'l', 'w'}. Mit Hilfe der Signalstärken kann wieder eine partielle Ordnung auf diesen Werten definiert werden (s. Abb. 2.58).

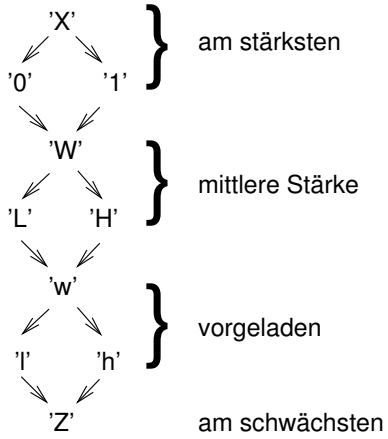


Abb. 2.58. Partielle Ordnung der Wertemenge {'0', '1', 'Z', 'X', 'H', 'L', 'W', 'h', 'l', 'w'}

Fünf Signalstärken

Bislang haben wir die Versorgungsspannungen nicht betrachtet. Diese sind stärker als alle bisher betrachteten Signale. Signalwertemengen, welche die Versorgungsspannungen beinhalten, haben zur Definition einer 46-elementigen Signalwertmenge geführt [Coelho, 1989]. Solche Modelle sind allerdings nicht sehr weit verbreitet.

IEEE 1164

In VHDL gibt es keine vordefinierte Anzahl von Signalwerten, wenn man von der grundsätzlichen Unterstützung für zwei Signalwerte absieht. Stattdessen kann man die zu verwendende Wertemenge in VHDL selbst definieren, und unterschiedliche VHDL-Modelle können unterschiedliche Signal-Wertemengen verwenden.

Würde diese Fähigkeit von VHDL allerdings in dieser Weise eingesetzt, wäre die Übertragbarkeit und Austauschbarkeit von VHDL-Modellen stark eingeschränkt. Um das Austauschen von Modellen zu vereinfachen, wurde eine Standard-Wertemenge definiert und von der IEEE standardisiert. Dieser Standard heißt IEEE 1164 und wird von vielen Systemen eingesetzt. IEEE 1164 hat neun Werte: {'0', '1', 'Z', 'X', 'H', 'L', 'W', 'U', '-'}. Die ersten sieben Werte entsprechen den sieben oben beschriebenen Signalwerten. 'U' steht für einen nicht-initialisierten Wert. Er wird von Simulatoren für Signale verwendet, die noch nicht explizit definiert wurden.

'-' steht für eine beliebige Eingabe (*input don't care*). Dieser Wert muss näher erläutert werden. Hardwarebeschreibungsspachen werden häufig zur Beschreibung Boolescher Funktionen verwendet. Mit Hilfe der VHDL-**select**-Anweisung ist das bequem möglich. Die **select**-Anweisung entspricht den **switch**- und **case**-Anweisungen anderer Programmiersprachen. Die Bedeutung des VHDL- **select** ist jedoch anders als die Bedeutung der **select**-Anweisung in ADA (s. Seite 62).

Beispiel: Angenommen, wir wollen die Boolesche Funktion

$$f(a,b,c) = a\bar{b} + bc$$

beschreiben. Weiterhin nehmen wir an, dass f für den Fall $a = b = c =$'0' undefiniert sein soll. Eine sehr bequeme Methode, dies in VHDL zu beschreiben, zeigt der folgende Code-Ausschnitt:

```
f <= select a & b & c  -- & ist die Konkatenation
     '1' when "10-"    -- erster Term
     '1' when "-11"    -- zweiter Term
     'X' when "000"
```

So könnten gegebene Funktionen leicht in VHDL umgesetzt werden. Leider drückt die select-Anweisung hier etwas vollkommen Anderes aus: Da IEEE 1164 nur eine von vielen möglichen Wertemengen darstellt, kann VHDL kein Wissen über die „Bedeutung“ von '-' haben. Wenn VHDL-Compiler Konstrukte wie die oben gezeigte select-Anweisung übersetzen, prüfen sie, ob der zu betrachtende Ausdruck (im Beispiel a & b & c) den Werten in den *when*-Teilen entspricht. Genauer gesagt überprüfen sie, ob z.B. a & b & c gleich "10-" ist. In diesem Zusammenhang verhält sich '-' wie jeder andere Logikwert auch: VHDL-Systeme überprüfen, ob c den Wert '-' hat. Da der Wert '-' niemals irgendeiner der Variablen zugewiesen wird, werden diese Bedingungen nie erfüllt. Daher ist der Wert '-' nur von begrenztem Nutzen. Die Nichtverfügbarkeit eines praktischen Wertes für eine beliebige Eingabe ist der Preis, den man für die Flexibilität bei der Definition von Wertemengen in der Programmiersprache VHDL bezahlen muss.

Eine positive Eigenschaft der beschriebenen allgemeinen Betrachtungen von Seite 69 bis Seite 74 ist die Tatsache, dass man daraus direkt Schlussfolgerungen bezüglich der Ausdrucks- und Modellierungsfähigkeit des Standards IEEE 1164 ziehen kann. Der IEEE-Standard basiert auf der 7-elementigen Signalwertmenge von Seite 71 und kann daher Schaltungen mit Verarmungstransistoren modellieren. Eine Modellierung von vorgeladenen Busleitungen ist damit allerdings nicht möglich[11].

2.11.4 VHDL-Prozesse und Simulations-Semantik

VHDL behandelt die oben beschriebenen Komponenten wie Prozesse. Streng genommen ist die oben verwendete Syntax nur eine Kurzform für die Beschreibung von Prozessen. Die allgemeine Syntax für Prozesse ist wie folgt:

Name : -- optional
process
Deklarationen -- optional
begin
Befehle-- optional
end process ;

Prozesse können **wait**-Instruktionen enthalten. Solche Instruktionen werden benutzt, um die Ausführung eines Prozesses zu suspendieren. Es gibt folgende Arten von **wait**-Instruktionen:

[11] Abgesehen von dem Fall, in dem man keine Verarmungstransistoren einsetzt. In diesem Fall könnte man die schwachen Werte als auf den Leitungen gespeicherte Ladungen interpretieren. Dies ist allerdings nicht sehr praktisch, da Verarmungstransistoren heute in den meisten Systemen eingesetzt werden.

- **wait on** *Signalliste*; suspendieren, bis sich eines der Signale in der Liste ändert;
- **wait until** *Bedingung*; suspendieren, bis die *Bedingung* erfüllt ist, z.B. a = '1';
- **wait for** *Dauer*; für eine bestimmte Zeitdauer suspendieren;
- **wait**; auf unbestimmte Zeit suspendieren.

Als Alternative zu expliziten **wait**-Instruktionen kann im Kopf des Prozesses eine Liste von Signalen angegeben werden, die sogenannte **Sensitivitätsliste** (*sensitivity list*). In diesem Fall wird der Prozess immer dann aktiviert, wenn eines der Signale in dieser Liste seinen Wert ändert. Beispiel: Das folgende Modell eines AND-Gatters wird zu Beginn der Simulation den Prozess einmal ausführen. Danach wird der Prozess immer dann neu gestartet, wenn eines seiner Eingangssignale den Wert ändert:

```
process(x, y) begin
    prod <= x  AND y ;
end process;
```

Dieses Modell ist äquivalent zu

```
process begin
    prod <= x  AND y ;
    wait on x,y;
end process;
```

Im Original Standard-Dokument [IEEE, 1997] wird die Ausführung eines VHDL-Prozesses wie folgt beschrieben: *Die Ausführung eines Modells besteht aus einer* ***Initialisierungsphase****, danach folgt die* ***wiederholte Ausführung von Prozessen****, die in der Beschreibung des Modells enthalten sind. Jede solche Wiederholung heißt* ***Simulations-Zyklus****. In jedem Zyklus werden die Werte aller Signale in der Prozess-Beschreibung berechnet. Falls aufgrund dieser Ergebnisse ein Ereignis auf einem bestimmten Signal stattfindet, so werden Prozesse, die auf dieses Signal sensitiv sind, aufgeweckt und innerhalb des selben Simulationszyklus ausgeführt.*

Die Initialisierungsphase berücksichtigt die angegebenen Initialisierungswerte für Signale und Variablen und führt jeden Prozess einmal aus. Sie wird im Standard wie folgt beschrieben[12]:

Zu Beginn der Initialisierung ist die aktuelle Zeit T_c 0 ns. Die Initialisierungsphase besteht aus den folgenden Schritten[13]:

[12] Implizit deklarierte Signale und sogenannte „*postponed processes*“, die in der 1997er-Version von VHDL eingeführt wurden, werden hier nicht betrachtet.

[13] Um in der Detailfülle des Standards nicht den Überblick zu verlieren, werden einige Punkte weggelassen (markiert durch „...“).

- *Der treibende Wert und der effektive Wert jedes explizit deklarierten Signals werden berechnet, und der aktuelle Wert des Signals wird auf den effektiven Wert gesetzt. Es wird angenommen, dass das Signal diesen Wert bereits für eine unendlich lange Zeit vor dem Start der Simulation hatte. ...*
- *Jeder ... Prozess des Modells wird ausgeführt, bis er suspendiert wird. ...*
- *Die Zeit des nächsten Simulationszyklus (der in diesem Fall der erste Simulationszyklus ist) T_n wird anhand der Regeln von Schritt e des Simulationszyklus (s.u.) berechnet.*

Jeder Simulationszyklus beginnt damit, dass die aktuelle Zeit auf denjenigen nächsten Zeitpunkt gesetzt wird, an dem Änderungen berücksichtigt werden müssen. Diese Zeit T_n wurde entweder während der Initialisierungsphase oder während der letzten Ausführung eines Simulationszyklus berechnet. Die Simulation wird beendet, wenn die aktuelle Zeit den Maximalwert $TIME'HIGH$ erreicht. Im Originaldokument wird der Simulationszyklus wie folgt beschrieben: *Ein Simulationszyklus besteht aus den folgenden Schritten:*

a) *Die aktuelle Zeit T_c wird auf T_n gesetzt. Die Simulation ist beendet, wenn $T = TIME'HIGH$ ist und wenn es keine aktuellen Treiber oder aufgeweckte Prozesse zur Zeit T_n gibt.*

b) *Jedes aktive explizite Signal im Modell wird aktualisiert. (Daraus können sich Ereignisse ergeben) ...*

   Dieser Satz aus dem Dokument bezieht sich darauf, dass im vorhergehenden Zyklus neue zukünftige Werte für einige der Signale berechnet worden sein können. Wenn T_c dem Zeitpunkt entspricht, an dem diese Werte gültig werden, werden sie jetzt zugewiesen. Man beachte, dass neue Werte für Signale niemals sofort innerhalb des Simulationszyklus zugewiesen werden. Sie werden frühestens vor dem nächsten Simulationszyklus zugewiesen. Signale, die ihren Wert verändern, verursachen sogenannte Ereignisse, die ihrerseits wiederum die Ausführung von Prozessen anstoßen können, die auf diese Signale sensitiv sind.

c) *Jeder Prozess P, der gerade auf ein Signal S sensitiv ist, wird aufgeweckt, wenn in diesem Simulationszyklus ein Ereignis auf dem Signal S stattgefunden hat.*

d) *Jeder ... Prozess, der im aktuellen Simulationszyklus aufgeweckt wurde, wird solange ausgeführt, bis er suspendiert wird.*

e) *Die Zeit des nächsten Simulationszyklus T_n wird bestimmt als frühester Zeitpunkt, an dem*

   1. *TIME'HIGH erreicht wird* (das Ende der Simulationszeit).
   2. *ein Signaltreiber aktiv wird* (der nächste Zeitpunkt, an dem ein Treiber einen neuen Wert angibt), *oder*

3. *ein Prozess aufgeweckt wird* (diese Zeit wird durch **wait for**-Anweisungen bestimmt).

Wenn $T_n = T_c$, dann ist der nächste Simulationszyklus (wenn es einen gibt) ein Delta-Zyklus.

Die iterative Form von Simulationszyklen wird in Abb. 2.59 gezeigt.

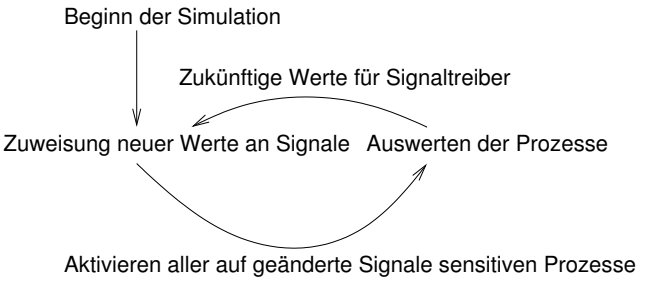


Abb. 2.59. VHDL Simulationszyklen

Delta (δ)-Simulationszyklen sind eine Quelle vieler Diskussionen. Ihr Sinn ist es, eine infinitesimal kleine Verzögerung zu erzeugen, selbst wenn der Benutzer keine Verzögerung angegeben hat. Als Beispiel zeigen wir den Effekt dieser Zyklen anhand eines Flipflops. Abb. 2.60 zeigt die Grundschaltung eines Flipflops.

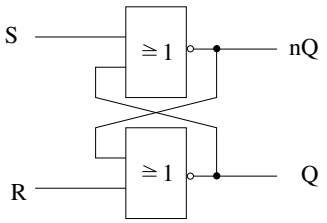


Abb. 2.60. RS-Flipflop

Das Flipflop wird in VHDL wie folgt modelliert:

```
entity RS_Flipflop is
port (R: in    BIT; -- zurücksetzen
      S: in    BIT; -- setzen
      Q: inout BIT; -- Ausgabe
      nQ: inout BIT; -- neg. Ausgabe
      );
```

```
end RS_Flipflop;
architecture one of RS_Flipflop is
begin
process: (R,S,Q,nQ)
    begin
      Q  <= R nor nQ;
      nQ <= S nor Q;
    end process;
end one;
```

Die Ports Q und nQ müssen vom Typ **inout** sein, da sie auch intern gelesen werden, was mit Ports vom Typ **out** nicht möglich ist. Abb. 2.61 zeigt die Simulationszeitpunkte, an denen die Signale dieses Modells aktualisiert werden.

	< 0ns	0ns	0ns+δ	0ns+$2 * \delta$	0ns+$3 * \delta$
R	0	1	1	1	1
S	0	0	0	0	0
Q	1	1	0	0	0
nQ	0	0	0	1	1

Abb. 2.61. δ-Zyklen für das RS-Flipflop

Die Ausgangssituation sei S=R=nQ='0', Q='1'. Zur Zeit t=0 werde dann R='1' gesetzt. Im ersten δ-Zyklus ändert Q den Wert aufgrund der Änderung des Werts von R. Im zweiten δ-Zyklus ändert nQ den Wert aufgrund der Änderung des Werts von Q im ersten δ-Zyklus. Im dritten Zyklus wird die Änderung von nQ zum „oberen“ Eingang des unteren NOR-Gatters propagiert. Der Ausgangswert ändert sich dadurch nicht mehr, weil bereits vorher der „untere“ Eingang unteren Gatters mit '1' belegt war.

δ-Zyklen entsprechen einer infinitesimal kleinen Zeiteinheit, die in Wirklichkeit immer vorhanden ist. δ-Zyklen stellen sicher, dass die Simulation die Kausalität erhält und dass die Ergebnisse nicht von der Reihenfolge abhängen, in der Teile eines Modells während der Simulation ausgewertet werden. Wäre das nicht der Fall, so wäre die Simulation nicht-deterministisch, was man von der Simulation einer Schaltung, die in der Realität deterministisch ist, nicht erwarten würde. Es kann beliebig viele δ-Zyklen geben, bevor die aktuelle Simulationszeit T_c weitergeschaltet wird. Diese Möglichkeit, Endlosschleifen in VHDL-Programme einzubauen, kann sehr verwirrend sein. Um solche Endlosschleifen zu vermeiden, kann man z.B. Wertzuweisungen ohne zeitli-

che Verzögerung, wie wir sie in unserem Flipflop-Modell verwendet haben, verbieten.

Ein sehr wichtiges Konzept in VHDL ist die Trennung zwischen der Berechnung neuer Werte für Signale und die eigentliche Zuweisung der Werte an die Signale. Erst diese Trennung ermöglicht deterministische Simulationsergebnisse. Wenn ein Modell die folgenden Zeilen enthält

```
a <= b;
b <= a;
```

so werden die Signale a und b immer vertauscht. Wenn die Zuweisungen sofort ausgeführt würden, so würde das Ergebnis von der Reihenfolge abhängen, in der die beiden Wertzuweisungen ausgeführt werden (s. dazu auch Seite 26).

2.12 SystemC

Da heute mehr und mehr Funktionalität in Software implementiert wird, besteht eine wachsende Zahl von eingebetteten Systemen aus einer Mischung von Hard- und Software. Die am Häufigsten verwendete Sprache zur Beschreibung von Software in eingebetteten Systemen ist C. Beispielsweise implementieren eingebettete Systeme Standards wie MPEG 1/2/4 oder Dekodierer für Handy-Übertragungsstandards wie GSM. Diese Standards sind häufig in Form einer „Referenz-Implementierung“ verfügbar, zumeist ein C-Programm, das nicht auf Geschwindigkeit optimiert ist, sondern lediglich die gewünschte Funktionalität beschreibt. Der Nachteil von Entwürfen, die mit VHDL oder Verilog gemacht werden, besteht darin, dass solche Standards reimplementiert werden müssen, um sie in Hardware zu realisieren. Um Hardware und Software gemeinsam simulieren zu können, müssen Hard- und Software-Simulatoren außerdem über geeignete Schnittstellen verfügen. Das führt typischerweise zu einer niedrigeren Effizienz und zu inkonsistenten Benutzerschnittstellen. Außerdem müssen die Entwickler mehrere Sprachen beherrschen.

Aus diesen Gründen wurde nach Techniken gesucht, um Hardwarestrukturen in Software-Sprachen abzubilden. Einige grundlegende Probleme müssen gelöst werden, bevor man Hardware mit Hilfe von Software-Sprachen modellieren kann:

- **Nebenläufigkeit**, wie man sie in Hardware findet, muss in Software modelliert werden.
- Es muss eine Darstellungsmöglichkeit für Simulations-**Zeit** geben.
- **Mehrwertige Logik** und **Signalauflösung**, wie weiter oben beschrieben, müssen unterstützt werden.
- Das **deterministische Verhalten** beinahe aller relevanter Hardware-Schaltungen muss garantiert werden.

SystemC^{TM} [SystemC, 2002] ist eine C++-Klassenbibliothek, die zur Lösung dieser Probleme entwickelt wurde. Bei Verwendung von SystemC können Spezifikationen in C oder in C++ geschrieben werden, wobei an den entsprechenden Stellen Referenzen auf die Klassenbibliothek eingesetzt werden.

SystemC beinhaltet die gleichzeitige Ausführung mehrerer Prozesse. Die Semantik der Simulation ist ähnlich wie bei VHDL, auch in SystemC gibt es Delta-Zyklen. Die Ausführung von Prozessen wird über Sensitivitätslisten und Aufrufe von **wait**-Instruktionen gesteuert. Das Konzept der Sensitivitätslisten von VHDL wurde erweitert, so dass in SystemC 2.0 auch dynamische Sensitivitätslisten enthalten sind.

SystemC hat auch ein Modell für die Zeit. SystemC 1.0 verwendet Gleitkommazahlen, um die Zeit darzustellen. In SystemC 2.0 wird eine ganzzahlige Darstellung bevorzugt. SystemC 2.0 unterstützt auch physikalische Einheiten wie Pikosekunden, Nanosekunden, Mikrosekunden usw.

Die Datentypen von SystemC beinhalten alle üblichen Hardware-Typen: vierwertige Logik ('0', '1', 'X' und 'Z') sowie Bitvektoren unterschiedlicher Länge werden unterstützt. Die Beschreibung von Anwendungen aus der digitalen Signalverarbeitung wird durch die Verfügbarkeit von Typen für Festkommazahlen vereinfacht.

Deterministisches Verhalten (s. Seite 29) wird im Allgemeinen nicht garantiert, sondern nur durch Verwendung eines bestimmten Modellierungs-Stils erreicht. Mit Hilfe eines Kommandozeilenschalters kann der Simulator angewiesen werden, Prozesse in unterschiedlichen Reihenfolgen auszuführen. So kann der Benutzer überprüfen, ob die Simulationsergebnisse von der Reihenfolge der Prozessausführungen abhängen. Für Modelle mit realistischer Komplexität kann man allerdings nicht nachweisen, ob sie deterministisch sind – man kann nur zeigen, dass sie nicht-deterministisch sind.

Die Wiederverwendung von Hardwarekomponenten in verschiedenen Kontexten wird durch die Aufspaltung in Berechnung und Kommunikation vereinfacht. SystemC 2.0 bietet *channels*, *ports* und *interfaces* als abstrakte Komponenten für die Kommunikation an.

SystemC hat das Potential, die bestehenden VHDL-Designabläufe zu ersetzen, obwohl Programme zur automatisierten Hardware-Synthese ausgehend von SystemC-Modellen gerade erst verfügbar werden [Herrera et al., 2003a], [Herrera et al., 2003b]. Das Vorgehen und Anwendungen für SystemC-basierte Entwürfe werden in einem Buch über dieses Thema [Müller et al., 2003] beschrieben.

2.13 Verilog und SystemVerilog

Verilog [Thomas und Moorby, 1991] ist eine weitere Hardware-Beschreibungssprache. Ursprünglich war es eine proprietäre Sprache, wurde aber später

als IEEE Standard 1364 vereinheitlicht. Die Versionen heißen IEEE Standard 1364-1995 (Verilog Version 1.0) und IEEE Standard 1364-2001 (Verilog 2.0). Einige Eigenschaften von Verilog sind VHDL sehr ähnlich. Genau wie in VHDL werden Entwürfe als miteinander verbundene *Design-Entities* beschrieben, diese können wiederum verhaltensorientiert beschrieben werden. Prozesse werden zur Modellierung der Nebenläufigkeit von Hardwarekomponenten verwendet. Genau wie in VHDL gibt es Bitvektoren und Zeiteinheiten. In einigen Bereichen ist Verilog allerdings weniger flexibel und konzentriert sich mehr auf bequem verwendbare Grundfunktionen. Standard-Verilog erlaubt es beispielsweise nicht, Aufzählungstypen so flexibel zu definieren wie dies im IEEE 1164-Standard der Fall ist. Die Unterstützung für vierwertige Logik ist direkt in die Verilog-Sprache eingebaut, und der Standard IEEE 1364 stellt auch eine mehrwertige Logik mit acht verschiedenen Signalstärken zur Verfügung. Mehrwertige Logik ist in Verilog enger an die Sprache gebunden als in VHDL. Das Logik-System von Verilog bietet auch mehr Funktionen für die Beschreibung von Schaltungen auf der Transistor-Ebene. Trotzdem ist VHDL im Allgemeinen flexibler. So kann man in VHDL etwa Hardware-Komponenten in Schleifen instantiieren, wodurch man strukturelle Beschreibungen wie z.B. einen N-Bit-Addierer spezifizieren kann, ohne wirklich N Addierer und ihre Verbindungen explizit angeben zu müssen.

Die Anzahl der Benutzer von Verilog ist ähnlich hoch wie die von VHDL. Während VHDL in Europa beliebter ist, wird Verilog in den Vereinigten Staaten bevorzugt.

Die Versionen 3.0 und 3.1 von Verilog sind auch als SystemVerilog bekannt. Sie enthalten zahlreiche Erweiterungen zu Verilog 2.0, u.a. [Accellera, 2005]:

- zusätzliche Sprachelemente, um Verhalten zu modellieren,
- C-Datentypen wie `int` und Methoden zur Typdefinition wie `typedef` und `struct`,
- Definition der Schnittstellen von Hardwarekomponenten als eigenständige *Entities*,
- standardisierte Methoden zum Aufruf von C/C++-Funktionen und, zu einem gewissen Grad, zum Aufruf eingebauter Verilog-Funktionen aus C,
- deutlich verbesserte Unterstützung zur Beschreibung der Umgebung (der *Testbench*) für die entwickelte Hardware (die als CUD bezeichnet wird), und zur Verwendung der *Testbench*, um die CUD durch Simulation zu verifizieren,
- Klassen (wie man sie von der objektorientierten Programmierung her kennt), die in Testbenches verwendet werden können,
- dynamische Erzeugung von Prozessen,
- standardisierte Interprozess-Kommunikation und -Synchronisation, auch mit Semaphoren,

- automatische Anforderung und Freigabe von Speicher,
- Sprachelemente, welche die formale Verifikation vereinheitlichen (s. Seite 213).

Aufgrund der Schnittstellen zu C und C++ ist es auch möglich, Verilog mit SystemC zu verbinden. Verbesserte Simulations- und formale Verifikationsmöglichkeiten sowie die Kombination mit SystemC-Modellen lassen eine steigende Akzeptanz und Beliebtheit von Verilog erwarten.

2.14 SpecC

Die Sprache SpecC [Gajski et al., 2000] basiert auf einer klaren Trennung zwischen Kommunikation und Berechnung, wie sie bei der Modellierung eingebetteter Systeme verwendet werden sollte. Diese Trennung ermöglicht die Wiederverwendung von Komponenten in verschiedenen Kontexten und erlaubt eine *plug-and-play*-Verwendung von Systemkomponenten. SpecC modelliert Systeme als hierarchische Netzwerke von Verhaltensweisen, die über Kanäle miteinander kommunizieren. SpecC-Beschreibungen bestehen aus Verhalten (*behaviors*), Kanälen (*channels*) und Schnittstellen (*interfaces*). *Behaviors* enthalten *Ports*, lokal instantiierte Komponenten, private Variablen und Funktionen sowie eine nach außen sichtbare main-Funktion. *Channels* kapseln die Kommunikation. Sie enthalten Variablen und Funktionen, die zur Definition von Kommunikationsprotokollen verwendet werden. *Interfaces* verbinden *Behaviors* und *Channels*. Sie deklarieren die Kommunikationsprotokolle, die innerhalb eines Kanals definiert werden.

SpecC kann Hierarchien mit Hilfe von geschachtelten *Behaviors* modellieren. Abb. 2.62 [Gajski et al., 2000] zeigt eine Komponente B, die Unterkomponenten b1 und b2 enthält.

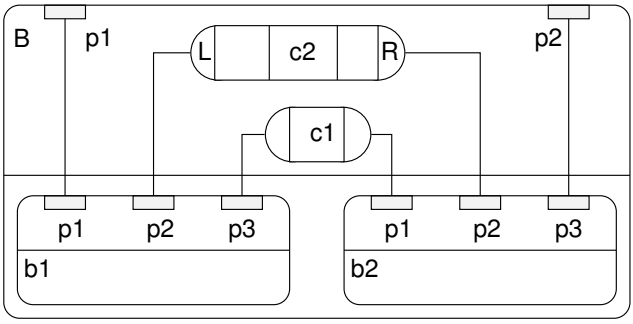


Abb. 2.62. Strukturelle Hierarchie in SpecC

Die Unterkomponenten kommunizieren durch die Variable c1 und durch den Kanal c2. Die strukturelle Hierarchie enthält die Instanzen b1 und b2 als Blattknoten. b1 und b2 werden parallel ausgeführt, was in SpecC durch das Schlüsselwort **par** ausgedrückt wird. Diese strukturelle Hierarchie wird im folgenden SpecC-Modell beschrieben:

```
interface L {void Write(int x); };
interface R {int Read(void); };
channel C implements L,R
  {int Data; bool Valid;
  void Write(int x) {Data=x; Valid=true;}
  int Read (void)
    {while (!Valid) waitfor (10); return (Data);} }
 behavior B1(in int p1, L p2, in int p3)
  {void main (void) {/* ...*/ p2.Write(p1);} };
 behavior B2 (out int p1, R p2, out int p3)
  {void main(void) {/*...*/ p3=p2.Read(); } };
behavior B(in int p1, out int p2)
 {int c1;   C c2;   B1 b1(p1, c2, c1);   B2 b2(c1, c2, p2);
 void main (void)
  {par {b1.main(); b2.main();}}
 };
```

Man beachte, dass man das Schnittstellen-Protokoll im Kanal C, das aus Methoden zum Lesen und Schreiben besteht, verändern kann, ohne damit die *behaviors* B1 und B2 ändern zu müssen. Die Kommunikation kann beispielsweise bit-seriell oder parallel erfolgen – die Auswahl einer dieser Methoden hat keine Auswirkungen auf die Modelle von B1 und B2. Dies ist eine notwendige Eigenschaft, wenn Modelle wiederverwendet werden sollen.

Um den Entwurf von Systemen, die Soft- und Hardware-Komponenten enthalten, zu vereinfachen, basiert die Syntax von SpecC auf C und C++. Tatsächlich werden SpecC-Modelle zur Simulation nach C++ übersetzt.

Das Kommunikationsmodell von SpecC hat die Kommunikation in SystemC 2.0 inspiriert.

2.15 Weitere Sprachen

Eine große Anzahl von Sprachen wurde für eingebettete Applikationen entwickelt. Die folgende Liste führt einige davon auf:

- **Pearl:** Pearl [Deutsches Institut für Normung, 1997] wurde für industrielle Kontroll-Anwendungen entwickelt. Es beinhaltet viele Sprachelemente, um Prozesse zu kontrollieren, auch die Modellierung von Zeit ist möglich. Pearl benötigt ein zugrundeliegendes Echtzeitbetriebssystem. Die Sprache ist in Europa sehr beliebt, was sich in der großen Anzahl von industriellen Steuerungsanwendungen zeigt, die in dieser Sprache implementiert wurden.
- **Chill:** Chill [Winkler, 2002] wurde für Telefonvermittlungsstellen entworfen. Es wurde von der CCITT standardisiert und in Telekommunikations-Geräten verwendet. Chill ist eine Art erweitertes PASCAL.
- **IEC 60848**, **STEP 7:** IEC 60848 [IEC, 2002] und STEP 7 [Berger, 2001] sind zwei Sprachen, die in Steuerungsanwendungen verwendet werden. Beide bieten graphische Elemente, um die Funktionalität des Systems zu beschreiben.
- **SpecCharts:** SpecCharts [Gajski et al., 1994], ein Vorgänger von SpecC, verbindet die Vorteile von StateCharts und VHDL. Es basiert auf dem StateCharts Modellparadigma von endlichen Automaten, erlaubt aber die Beschreibung des Verhaltens in VHDL. Außerdem werden verschiedene Transitionen unterschieden, die entweder sofort oder erst nach dem Ende aller Berechnungen durchgeführt werden. Der erste Transitionstyp vereinfacht die Modellierung von Ausnahmezuständen. Genau wie StateCharts hat auch SpecCharts Schwierigkeiten mit der Beschreibung struktureller Hierarchie.
- **Estelle** [Hogrefe, 1989]: Diese Sprache wurde zur Beschreibung von Kommunikationsprotokollen entworfen. Ähnlich wie SDL wird die Kommunikation durch Kanäle und FIFO-Puffer modelliert. Versuche, SDL und Estelle zusammenzuführen, sind gescheitert.
- **LOTOS, Z:** Diese Sprachen [Jeffrey und Leduc, 1996], [Spivey, 1992] sind algebraische Spezifikationssprachen, mit deren Hilfe präzise Spezifikationen und formale Beweise ermöglicht werden. Leider sind sie nicht ausführbar und daher nicht für die Validierung in frühen Entwurfsphasen geeignet.
- **Silage** [Genin et al., 1990]: Diese Sprache und die kommerzielle Variante DFL [Mentor Graphics, 1996] sind auf die digitale Signalverarbeitung zugeschnitten. Es handelt sich um funktionale Sprachen, wodurch man den sequentiellen, von-Neumann-artigen Programmierstil vermeiden kann.
- **Rosetta:** Die Entwicklung von Rosetta ist Teil der Aktivitäten der Accellera-Initiative. *„Die Mission von Accerella besteht darin, die weltweite Entwicklung und Verwendung von Standards, die von System-, Halbleiter- und Entwicklungswerkzeug-Firmen benötigt werden, voranzutreiben. Dadurch soll der sprachbasierte Entwurfsautomatisierungs-Prozess verbessert*

werden“ [Accellera, 2002]. Die Accellera-Initiative arbeitet auch an neuen Versionen der VHDL- und Verilog-Standards.

- **Esterel:** Die Definition von Esterel verfügt über die folgenden besonderen Eigenschaften [Boussinot und de Simone, 1991]: Esterel ist eine reaktive Sprache: wenn Esterel-Modelle mit einem Eingabe-Ereignis aktiviert werden, reagieren sie mit der Erzeugung eines Ausgabe-Ereignisses. Esterel ist eine synchrone Sprache: es wird angenommen, dass alle Reaktionen ohne Zeitverzögerung abgeschlossen werden, und dass es ausreichend ist, das Verhalten zu diskreten Zeitpunkten zu analysieren. Dieses idealisierte Modell vermeidet die Probleme von überlappenden Zeiträumen und Ereignissen, die ankommen, während die vorhergehende Reaktion noch nicht abgeschlossen war. Wie andere Sprachen auch, hat Esterel einen Parallelisierungs-Operator, der als || geschrieben wird. Wie in StateCharts findet Kommunikation über einen *Broadcast*-Mechanismus statt. Im Gegensatz zu StateCharts findet die Kommunikation allerdings augenblicklich, ohne jede Verzögerung, statt. Das bedeutet, dass alle Signale, die zu einem bestimmten Zeitpunkt erzeugt werden, in genau diesem Zeitpunkt auch von allen anderen Teilen des Modells gesehen werden. Wenn diese anderen Teile sensitiv auf die erzeugten Signale reagieren, reagieren sie auch in genau diesem einen Zeitpunkt. Diese Weiterleitung von Werten während ein und derselben makroskopischen Zeiteinheit entspricht den δ-Zyklen von VHDL und der Erzeugung des nächsten Zustands für den gleichen Zeitpunkt in StateCharts, wobei der *Broadcast* hier augenblicklich geschieht. Das synchrone Ausführungsmodell führt dazu, dass Esterel-Programme widersprüchlich oder mehrdeutig (nicht deterministisch) sein können. Solche Programme müssen vom Compiler erkannt und zurückgewiesen werden. Für mehr und aktuellere Informationen zu Esterel verweisen wir auf die Web-Seite [Esterel, 2006] und zwei Bücher [Potop-Butucaru et al., 2007], [von Hanxleden, 2005].
- **MATLAB/Simulink:**

  MATLAB/Simulink [Tewari, 2001] ist ein Modellierungs- und Simulations-Werkzeug, das auf mathematischen Prinzipien beruht. Systeme können etwa in Form von partiellen Differentialgleichungen beschrieben werden. Dieser Ansatz ist gut geeignet, um physikalische Systeme, z.B. Autos oder Züge, zu modellieren und dann deren Verhalten zu simulieren. Auch digitale Signalverarbeitungssysteme können bequem in MATLAB modelliert werden. Um eine reale Implementierung eines MATLAB/Simulink-Modells zu erhalten, muss es erst in eine Sprache übersetzt werden, die von Hardware- oder Software-Entwurfssystemen unterstützt wird, also beispielsweise nach C oder VHDL.

2.16 Ebenen der Hardware-Modellierung

In der Praxis werden Entwürfe auf unterschiedlichen Abstraktionsebenen begonnen. In einigen Fällen wird auf einer hohen Abstraktionsebene das Gesamtverhalten des zu entwerfenden Systems beschrieben. In anderen Fällen beginnt die Spezifikation mit der Beschreibung einer elektrischen Schaltung auf einer entsprechend niedrigeren Abstraktionsstufe. Für jede Abstraktionsebene gibt es eine Anzahl von Sprachen, und einige Sprachen decken mehrere Abstraktionsebenen ab. Im Folgenden wird eine Menge von möglichen Abstraktionsebenen beschrieben. Einige der niedrigeren Abstraktionen werden hier nur der Vollständigkeit halber erwähnt – eine Spezifikation sollte nicht auf einer dieser Ebenen beginnen. Es folgt eine Liste von häufig verwendeten Bezeichnungen und Eigenschaften von Abstraktionsebenen:

- **Modelle auf Systemebene:** Der Begriff Systemebene ist nicht klar definiert. Er wird hier verwendet, um das gesamte eingebettete System und das System, in das die Informationsverarbeitung integriert ist („das Produkt“) zu beschreiben, und möglicherweise auch die Umwelt (die physikalischen Eingaben an das System, die z.B. die Straßenverhältnisse oder Wetterbedingungen darstellen können). Offensichtlich beinhalten solche Modelle sowohl Mechanik- als auch Informationsverarbeitungs-Aspekte, und es kann schwierig sein, dafür passende Simulatoren zu finden. Mögliche Lösungen sind die Verwendung von VHDL-AMS (die analoge Erweiterung von VHDL), SystemC oder MATLAB. MATLAB und VHDL-AMS unterstützen die Modellierung partieller Differentialgleichungen, eine der wichtigsten Anforderungen bei der Simulation mechanischer Systeme. Es ist eine große Herausforderung, die informationsverarbeitenden Bestandteile des Systems so zu modellieren, dass das Simulationsmodell auch zur Synthese des eingebetteten Systems verwendet werden kann. Wenn das nicht möglich ist, muss unter Umständen eine fehleranfällige manuelle Übersetzung zwischen den verschiedenen Modellen durchgeführt werden.
- **Algorithmische Ebene:** Auf dieser Ebene werden Algorithmen simuliert, die innerhalb des eingebetteten Systems zum Einsatz kommen sollen. Beispielsweise kann ein MPEG Video-Encoder simuliert werden, um die Ausgabequalität der Videos zu bestimmen. Bei der Verwendung solcher Simulationen gibt es keinen Bezug zum Befehlssatz des Zielprozessors.

  Datentypen können in der Simulation durchaus noch eine höhere Wortbreite haben als in der endgültigen Implementierung. Beispielsweise verwenden die Referenzimplementierungen des MPEG-Standards doppeltgenaue Gleitkommazahlen. Das endgültige eingebettete System wird solche Datentypen kaum verarbeiten können. Wenn die Datentypen so gewählt wurden, dass jedes Bit in der Simulation genau einem Bit in der Implementierung entspricht, so spricht man von einem **bitgenauen Modell**.

Die Übersetzung von nicht-bitgenauen in bitgenaue Modelle sollte durch Hilfsprogramme unterstützt werden (s. Seite 176).

Modelle auf dieser Ebene können aus einem einzelnen Prozess oder aus einer Menge von kooperierenden Prozessen bestehen.

- **Befehlssatz-Ebene:** In diesem Fall wurden die Algorithmen bereits für den Befehlssatz des zu verwendenden Prozessors (oder der Prozessoren) übersetzt. Simulationen auf dieser Ebene erlauben das Zählen der ausgeführten Instruktionen. Es gibt verschiedene Varianten der Befehlssatzebene:
  - In einem grobkörnigen Modell wird nur die Wirkung von Instruktionen simuliert, das Zeitverhalten wird vernachlässigt. Die Informationen aus Assembler-Referenz-Handbüchern (die Befehlssatz-Architektur, *Instruction Set Architecture* (ISA)) sind für die Definition eines solchen Modells ausreichend.
  - **Modellierung auf Transaktions-Ebene:** Bei der Modellierung auf Transaktionsebene werden Transaktionen, wie z.B. Bus Lese- oder Schreiboperationen, sowie Kommunikationsvorgänge zwischen verschiedenen Komponenten modelliert. Diese Art der Modellierung enthält weniger Details als die zyklengenaue Modellierung (s.u.), daher können hier deutliche Vorteile bei der Simulationsgeschwindigkeit erzielt werden [Clouard et al., 2003].
  - In einem feinkörnigeren Modell kann man eine **zyklengenaue Befehlssatz-Simulation** erreichen. In diesem Fall kann die genaue Zyklenzahl, die zur Ausführung einer Applikation benötigt wird, bestimmt werden. Zum Aufstellen von zyklengenauen Modellen braucht man sehr genaue Informationen über die Prozessor-Hardware um z.B. *Pipeline-Stalls*, Ressourcen-Konflikte und Speicher-Wartezyklen richtig modellieren zu können.
- **Register-Transfer-Ebene (RTL):** Auf dieser Ebene werden alle Komponenten auf der Register-Transfer-Ebene modelliert. Das beinhaltet arithmetisch/logische Einheiten (ALUs), Register, Speicher, Multiplexer und Dekodierer. Modelle auf dieser Ebene sind immer zyklengenau. Die automatische Synthese solcher Modelle ist keine große Herausforderung.
- **Modelle auf Gatter-Ebene:** Hier enthalten die Modelle Gatter als Basis-Bausteine. Modelle auf Gatterebene erlauben genaue Aussagen über die Wahrscheinlichkeiten einer Wertänderung von Signalen und können deshalb zur Energiebestimmung herangezogen werden. Die Berechnung von Verzögerungen kann hier genauer durchgeführt werden als bei einem Modell auf RTL-Ebene. Allerdings ist während der Entwurfsphase üblicherweise keine Information über die Länge von Verbindungsleitungen und somit auch nicht über deren Kapazitäten verfügbar. Daher sind

Verzögerungs- und Energiewerte auch auf dieser Ebene lediglich Schätzungen.

Der Begriff „Gatterebene“ wird manchmal auch in Situationen verwendet, wo die Gatter lediglich dazu verwendet werden, um Boolesche Funktionen darzustellen. Gatter in einem solchen Modell repräsentieren aber nicht unbedingt physikalische Gatter. Es wird dann nur das Verhalten der Gatter betrachtet, nicht die Tatsache, dass sie in der späteren Realisierung physikalische Komponenten darstellen. Genauer sollten solche Modelle eigentlich „Boolesche Funktionsmodelle“ heißen[14], dieser Begriff ist aber nicht sehr weit verbreitet.

- **Modelle auf Schalter-Ebene:** Modelle auf Schalterebene verwenden Schalter (Transistoren) als Grundbausteine. Solche Beschreibungen verwenden Modelle digitaler Signalwerte (auf Seite 68 findet sich eine Beschreibung möglicher Wertemengen). Im Gegensatz zu Modellen auf Gatterebene können diese Modelle auf Schalterebene den bidirektionalen Transfer von Informationen in Schaltungen abbilden.
- **Modelle auf Schaltungs-Ebene:** Die Schaltungstheorie und ihre Bestandteile (Strom- und Spannungsquellen, Widerstände, Kondensatoren, Spulen und möglicherweise Makromodelle von Halbleitern) bilden die Basis der Simulation auf dieser Ebene. Simulationen beinhalten partielle Differentialgleichungen. Diese Gleichungen sind nur dann linear, wenn das Verhalten der Halbleiter linearisiert (approximiert) wird. Der am häufigsten verwendete Simulator auf der Schaltungsebene ist SPICE [Vladimirescu, 1987] und dessen Varianten.
- **Modelle auf Layout-Ebene:** Modelle auf Layoutebene zeigen das tatsächliche Layout der Schaltung. Solche Modelle beinhalten **geometrische** Informationen. Sie können nicht direkt simuliert werden, da die Geometrie keine konkreten Anhaltspunkte für das Verhalten gibt. Das Verhalten kann erschlossen werden, wenn man das Modell auf Layoutebene gemeinsam mit einem verhaltensorientierten Modell auf einer höheren Ebene kombiniert, oder indem aus dem Layout die Schaltung extrahiert wird, wobei man Wissen über die Darstellung von Komponenten auf der Layoutebene benötigt. In einem typischen Entwurfs-Ablauf werden die Länge von Verbindungsleitungen und deren entsprechende Kapazitäten aus dem Layoutmodell extrahiert und in die Beschreibungen auf höherer Ebene **zurückannotiert (*back-annotated*)**. Auf diese Weise kann die Genauigkeit von Verzögerungs- und Energieschätzungen verbessert werden.
- **Modelle auf Prozess- und Schaltungs-Ebene:** Auf einer noch niedrigeren Stufe kann man den Herstellungsprozess modellieren. Aufgrund der Information aus diesen Modellen kann man Parameter (z.B. Verstärkung, Kapazität) für die verwendeten Geräte (Transistoren) berechnen. Man be-

[14] Diese Modelle könnten mit Hilfe von binären Entscheidungsdiagrammen (*Binary Decision Diagram* (BDD)) [Wegener, 2000] dargestellt werden.

achte, dass die Verwendung des Begriffs „Prozess“ in der Fertigungstechnologie mit unserer bisherigen Benutzung nicht kompatibel ist.

2.17 Vergleich der Sprachen

Keine der bisher präsentierten Sprachen erfüllt alle Anforderungen an Spezifikationssprachen für eingebettete Systeme. Abb. 2.63 gibt einen Überblick über die Haupteigenschaften einiger Sprachen. Ausnahmen und dynamische Prozesserzeugung sollen ab SystemC 3.0 unterstützt werden.

Sprache	Verhaltens-Hierarchie	strukturelle Hierarchie	Programmiersprachenelemente	Ausnahmeunterstützung	dynamische Prozess-Erzeugung
StateCharts	+	-	-	+	-
VHDL	+	+	+	-	-
SpecCharts	+	-	+	+	-
SDL	+-	+-	+-	-	+
Petrinetze	-	-	-	-	+
Java	+	-	+	+	+
SpecC	+	+	+	+	+
SystemC	+	+	+	- (2.0)	- (2.0)
ADA	+	-	+	+	+

Abb. 2.63. Vergleich der Sprachen

Die Sprachen können auch hinsichtlich des benutzten Berechungsmodells (*Model of Computation*, (MoC)) verglichen werden. Das benutzte Berechungsmodell kann einen starken Einfluss darauf haben, mit welcher Klarheit und welchem Aufwand eine Anwendung implementiert werden kann. Abb. 2.64 zeigt eine mögliche grobe Einordnung von einigen der vorgestellten Sprachen.

Die Einordnung von ADA erfolgte dabei wegen der synchronen Kommunikation und trotz des Schwergewichts bei der Modellierung von Monoprozessor-Anwendungen. VHDL und Verilog wurden der *shared memory* Kategorie zugeordnet, da verteilte Simulatoren für diese Sprachen aufgrund der Simulationssemantik nur ineffizient zu implementieren sind. Standardsprachen wie C, C++ oder Java können um die Behandlung nachrichtenbasierter Kommunikation erweitert werden (siehe dazu Seite 165).

Die Modellierung in verschiedenen Berechnungsmodellen und deren Kombination sind Ziel des Ptolemy-Projekts [Davis et al., 2001]. Ein besonderes Augenmerk gilt der Erzeugung von eingebetteter Software. Die Kernidee besteht

Kommunikation/ Berechungsmodell	*Shared memory*	Nachrichtenbasierte Kommunikation	
		synchron	asynchron
Kommunizierende endliche Automaten	StateCharts		SDL
Datenflussmodell	(nicht sinnvoll)		Kahn-Netzwerke SDF
Von Neumann-Modell	C, C++, Java	C, C++, Java, ... mit Bibliotheken CSP, ADA	
Diskretes Ereignismodell (DE)	VHDL, Verilog SystemC	(Nur experimentelle Versuche) Verteiltes DE in Ptolemy	

Abb. 2.64. Berechnungsmodelle einiger der vorgestellten Sprachen

darin, die Software aus dem Berechnungsmodell zu erzeugen, das für eine bestimmte Anwendung am geeignetsten ist. Version 2 von Ptolemy (Ptolemy II) kennt die folgenden Berechnungsmodelle und zugehörigen Anwendungsgebiete:

1. endliche Automaten,
2. kommunizierende sequentielle Prozesse (Unterstützung von CSP),
3. diskretes Ereignismodell,
4. verteilte diskrete Ereignisse,
5. Prozessnetzwerke, unter Verwendung von Kahn-Prozessnetzwerken (s. Seite 59),
6. Synchroner Datenfluss (SDF, s. Seite 32),
7. kontinuierliche Zeit (angemessen für mechanische Systeme und analoge Schaltungen),
8. synchrone/reaktive Berechnungsmodelle (Esterel (s. Seite 86) ist eine Sprache, die dieses Modell verwendet).

Es ist nicht sehr wahrscheinlich, dass eine einzige Sprache oder ein einzelnes Berechungsmodell irgendwann einmal alle Anforderungen erfüllen wird, da einige dieser Anforderungen sich prinzipiell widersprechen. Eine Sprache, die echte Realzeit-Bedingungen erfüllt, kann für weniger strenge Echtzeitbedingungen eher ungeeignet sein. Eine Sprache, die für verteilte kontrolldominierte Anwendungen entworfen wurde, kann für lokale datenflussdominierte Anwendungen eher schlecht eingesetzt werden. Daher wird man wohl weiterhin mit Kompromissen leben müssen.

Welche Kompromisse werden in der Praxis tatsächlich verwendet? Die Programmierung in Assembler war in den frühen Jahren der Programmierung eingebetteter Systeme sehr weit verbreitet. Die Programme waren klein genug, um die Komplexität der Probleme tatsächlich noch in Assembler handhaben zu können. Der nächste Schritt besteht in der Verwendung der Programmiersprache C oder einem von ihr abgeleiteten Dialekt. Da sich die Komplexität der Software in eingebetteten Systemen stetig erhöht (s. Seite 9), werden der Sprache C höhere Programmiersprachen folgen. Objektorientierte Sprachen wie SDL bieten die nächste Abstraktionsstufe. Sprachen wie UML werden in frühen Entwurfsphasen benötigt, um die Spezifikation festzuhalten. In der Praxis können alle diese Sprachen so wie in Abb. 2.65 verwendet werden.

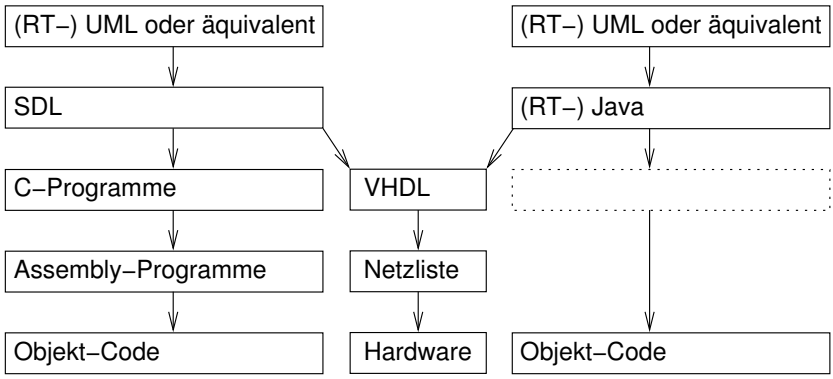


Abb. 2.65. Kombination mehrerer Sprachen zur Erzeugung einer Realisierung

Nach Abb. 2.65 können Sprachen wie SDL oder StateCharts in die Sprache C übersetzt werden. Diese C-Beschreibungen werden dann übersetzt. SDL oder StateCharts als Ausgangspunkt eröffnen auch den Weg, Funktionalität in Hardware zu implementieren, wenn es Übersetzer von diesen Sprachen nach VHDL gibt. Sowohl C als auch VHDL werden als Zwischensprachen sicherlich noch die nächsten Jahre überleben. Java benötigt diese Zwischenschritte nicht, profitiert aber auch von guten Übersetzungskonzepten in Assemblersprachen.

2.18 Verlässlichkeitsanforderungen

Die vorhergehenden Abschnitte haben sich hauptsächlich mit der Spezifikation des funktionalen Verhaltens des zu entwerfenden Systems beschäftigt. Insbesondere für sicherheitskritische Systeme können die Sicherheitsanforderungen aber alles andere dominieren. Sicherheitsanforderungen können nicht erst nachträglich eingebracht werden, sondern müssen von Anfang an in die Überlegungen und den Entwurf einbezogen werden. Der Entwurf von sicheren

und verlässlichen Systemen ist ein Thema für sich. Dieses Buch kann lediglich Hinweise in diese Richtung geben.

Nach Kopetz [Kopetz, 2003] muss man in Betracht ziehen, dass für sicherheitskritische Systeme das gesamte System verlässlicher sein muss als alle seine Bestandteile. Um diese Herausforderung zu lösen, schlägt Kopetz die folgenden zwölf Entwurfsprinzipien vor:

1. Sicherheitsüberlegungen müssen als **der** wichtige Teil der Spezifikation verwendet werden, sie treiben den gesamten Entwurfsprozess.
2. Präzise Spezifikationen von Entwurfshypothesen müssen ganz zu Beginn erstellt werden. Diese enthalten erwartete Fehler und ihre Wahrscheinlichkeit.
3. Fehler-Eingrenzungs-Regionen (*fault containment regions*) müssen betrachtet werden. Fehler in einer abgegrenzten Region sollten Fehler in anderen Regionen nicht beeinflussen.
4. Eine konsistente Verwendung von Zeit und Zuständen muss sichergestellt werden. Ansonsten ist es unmöglich, zwischen Ursprungs- und Folgefehler zu differenzieren.
5. Wohldefinierte Schnittstellen müssen die Interna der Komponenten verstecken.
6. Es muss sichergestellt sein, dass keine Abhängigkeiten zwischen dem Ausfallen der Komponenten bestehen.
7. Komponenten sollten sich selbst als korrekt funktionierend ansehen, es sei denn, zwei oder mehr andere Komponenten sind der Meinung, dass das Gegenteil der Fall ist (Prinzip des Selbstvertrauens).
8. Fehlertoleranzmechanismen müssen so entworfen sein, dass sie bei der Erklärung des Systemverhaltens keine zusätzliche Schwierigkeit darstellen. Fehlertoleranzmechanismen sollten von der normalen Funktion entkoppelt sein.
9. Das System muss Diagnose-freundlich entworfen sein. Beispielsweise muss es möglich sein, vorhandene (aber verdeckte) Fehler zu finden.
10. Die Mensch-Maschine-Schnittstelle muss intuitiv und nachsichtig sein. Die Sicherheit muss trotz menschlicher Fehler gewährleistet bleiben.
11. Jede Anomalie sollte aufgezeichnet werden. Diese Anomalien können auf dem normalen Schnittstellen-Niveau unsichtbar sein. Die Aufzeichnung sollte interne Effekte einschließen, da die Anomalien sonst von Fehlertoleranzmechanismen maskiert werden können.
12. Verwenden einer Niemals-Aufgeben-Strategie. Eingebettete Systeme müssen unter allen Umständen ununterbrochene Dienste leisten. Die Erzeugung beispielsweise eines *Pop-Up*-Fensters oder das einfache Abschalten sind nicht akzeptabel.

3

Hardware eingebetteter Systeme

3.1 Einführung

Beim Entwurf eingebetteter Systeme ist es erforderlich, sowohl die Hardware als auch die Software zu betrachten. Die Wiederverwendung vorhandener Hard- und Softwarekomponenten ist der Kernpunkt der aktuellen **Plattform-basierten Entwurfsmethodik**. Diese wird ab Seite 170 beschrieben. Mit dem Wissen, dass vorhandene Komponenten wiederverwendet werden müssen, beziehen wir uns in diesem Kapitel wieder auf das Informationsflussdiagramm aus Abbildung 3.1 und beschreiben einige wichtige Eigenschaften der in eingebetteten Systemen eingesetzten Hardware.

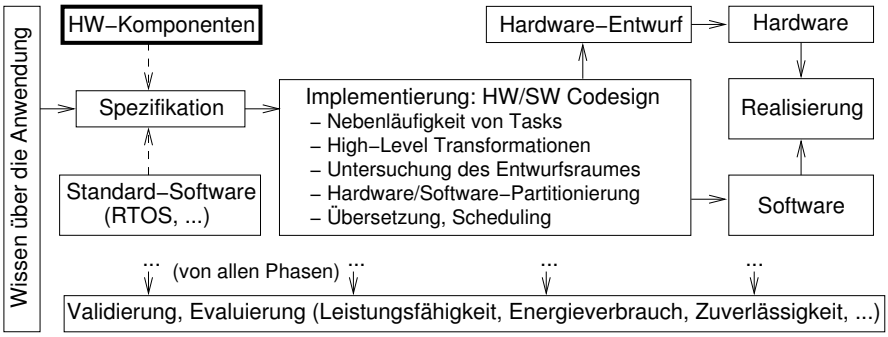


Abb. 3.1. Vereinfachter Informationsfluss beim Entwurf eingebetteter Systeme

Die Hardware eingebetteter Systeme ist viel weniger standardisiert als z.B. die von PCs. Aufgrund der riesigen Auswahl von unterschiedlichen Hardwarekomponenten für eingebettete Systeme ist es unmöglich, einen umfassenden Überblick über alle Arten von Komponenten zu geben. Trotzdem werden wir

versuchen, einen Überblick über die wichtigsten und am häufigsten verwendeten Komponenten zu geben.

In vielen eingebetteten Systemen, besonders in Regelungssystemen, wird eingebettete Hardware wie in Abb. 3.2 gezeigt in einer „Schleife“ verwendet (*Hardware in the loop*).

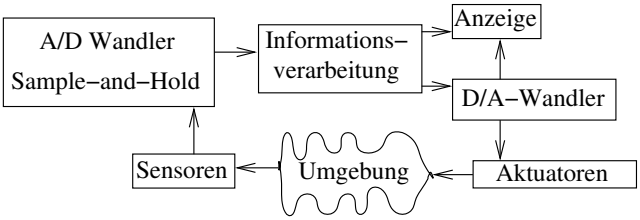


Abb. 3.2. *Hardware in the loop*

In dieser Schleife werden Informationen aus der physikalischen Umgebung durch **Sensoren** verfügbar gemacht. Typischerweise erzeugen Sensoren kontinuierliche Folgen analoger Werte. In diesem Buch werden wir uns auf die Betrachtung von digitalen Systemen beschränken, die diskrete Folgen von Werten verarbeiten. Zwei Arten von Schaltungen führen diese Konvertierung durch: *Sample-and-hold*-Schaltungen und Analog-Digital-Wandler (A/D-Wandler). In der Regel benutzt man dabei zunächst *sample-and-hold*-Schaltungen, um von zeitkontinuierlichen Wertemengen auf zeitdiskrete Folgen von Werten überzugehen. Anschließend erfolgt dann eine Wertediskretisierung im A/D-Wandler. Nach der Umwandlung können die Informationen digital weiterverarbeitet werden. Die erzeugten Ergebnisse können z.B. angezeigt, aber auch zur Steuerung der physikalischen Umgebung verwendet werden. Letzteres geschieht mit der Hilfe von **Aktuatoren**. Da die meisten Aktuatoren analog sind, benötigt man in der Regel auch eine Digital-Analog-Wandlung.

Dieses Modell ist offensichtlich geeignet für Regelungsanwendungen. Für andere Anwendungen kann man es immerhin als erste Näherung für ein Modell verwenden. In Handys werden beispielsweise Antennen als Sensoren und Lautsprecher als Aktuatoren verwendet. Im Folgenden werden die wichtigsten Hardwarekomponenten anhand der Schleifenstruktur aus Abb. 3.2 beschrieben.

3.2 Eingabe

3.2.1 Sensoren

Wir beginnen mit einer kurzen Betrachtung von Sensoren. Sensoren können für nahezu jede physikalische Größe entworfen werden, etwa für Gewicht,

Geschwindigkeit, Beschleunigung, elektrischen Strom, Spannung, Temperatur usw. Viele physikalische Effekte können zur Konstruktion von Sensoren ausgenutzt werden [Elsevier B.V., 2003a]. Beispiele hierfür sind die Induktion (die Entstehung elektrischer Ströme in einem magnetischen Feld) oder licht-elektrische Effekte. Außerdem gibt es auch Sensoren für chemische Substanzen [Elsevier B.V., 2003b].

In den letzten Jahren wurden viele neue Sensoren entwickelt, und ein Großteil des Fortschritts beim Entwurf von intelligenten Systemen kann auf die moderne Sensor-Technologie zurückgeführt werden. Es ist unmöglich, diesen Bereich der Hardware eingebetteter Systeme umfassend zu beschreiben. Daher geben wir einige charakteristische Beispiele:

- **Beschleunigungs-Sensoren:** Abbildung 3.3 zeigt einen kleinen Sensor, der mit Hilfe der Mikrosystem-Technologie hergestellt wurde. Der Sensor besitzt in seiner Mitte eine kleine Masse. Bei Beschleunigung wird diese Masse aus ihrer Mittelposition abgelenkt, was den elektrischen Widerstand der kleinen, an die Masse angeschlossenen Drähte verändert.

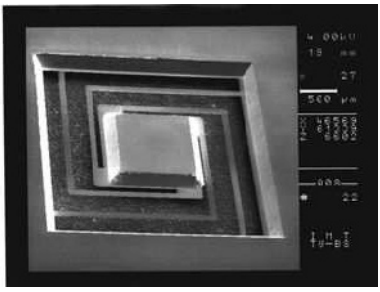


Abb. 3.3. Beschleunigungs-Sensor (mit freundlicher Genehmigung von S. Bütgenbach, IMT, TU Braunschweig), ©TU Braunschweig

- **Regen-Sensoren:** Um eine Ablenkung des Fahrers zu vermeiden, verfügen einige moderne Autos der Oberklasse über Regensensoren. Mit ihrer Hilfe kann die Geschwindigkeit der Scheibenwischer automatisch an die Regenintensität angepasst werden.
- **Bild-Sensoren:** Grundsätzlich gibt es zwei Sorten von Bildsensoren: sogenannte *Charge-Coupled Devices* (CCDs) und CMOS-Sensoren. In beiden Fällen werden gitterförmig angeordnete Lichtsensoren verwendet. Die Architektur von CMOS-Sensor-Gittern ähnelt der von normalen Speichern: einzelne Bildpunkte (*Pixel*) können wahlfrei adressiert und ausgelesen werden. CMOS-Sensoren verwenden die Standard-CMOS-Technologie integrierter Schaltkreise [Dierickx, 2000], was die Integration von Sensoren und logischen Schaltkreisen auf einem Chip erleichtert. Dadurch ist es möglich, einige vorbereitende Verarbeitungsschritte bereits direkt auf

dem Sensorchip zu erledigen. Man spricht in diesem Fall von intelligenten Sensoren (*smart sensors*). CMOS-Sensoren benötigen nur eine einzige Standard-Spannungsversorgung und verfügen im Allgemeinen über einfache Anschlussmöglichkeiten. Aus diesem Grunde können CMOS-basierte Sensoren preisgünstig hergestellt werden. Im Gegensatz dazu ist die CCD-Technologie für optische Anwendungen optimiert. In dieser Technologie müssen Ladungen von einem Pixel zum nächsten transportiert werden, bis sie schließlich am Rande des Gitters ausgelesen werden können. Diese sequentielle Weitergabe von Ladungen gab den CCDs ihren Namen. In Analogie zum Weiterreichen von mit Wasser gefüllten Eimern in einer Menschenkette beim Löschen eines Feuers werden CCDs im Deutschen auch als **Eimerkettenschaltungen** bezeichnet. Bilder, die mit Hilfe von CCD-Sensoren erzeugt wurden, können eine höhere Qualität als die von CMOS-Sensoren aufweisen. Der Anschluss und die Schnittstellen von CCD-Sensoren sind allerdings komplexer. Daher werden CMOS-Sensoren für solche Anwendungen verwendet, bei denen geringe Kosten eine große Rolle spielen und dafür eine niedrige bis mittlere Bildqualität akzeptiert werden kann. CCD-Sensoren werden für teurere Sensoren mit hoher Qualität eingesetzt, wie sie etwa in Videokameras und optischen Teleskopen verwendet werden. Allerdings werden CMOS-Sensoren besser und digitale Spiegelreflex-Kameras benutzen u.a. wegen der größeren Sensoren und der geringeren Bedeutung eines niedrigen Stromverbrauchs[1] die CMOS-Technik.

- **Biometrische Sensoren:** Höhere Sicherheitsstandards und die Notwendigkeit, mobile Ausrüstungsgegenstände zu sichern, haben zu einem verstärkten Interesse an Authentifizierungsmaßnahmen geführt. Wegen der Mängel von Sicherheitssystemen, die auf Passwörtern basieren (z.B. gestohlene und vergessene Passwörter), erfreuen sich *Smartcards*, biometrische Sensoren und biomedizinische Authentifizierungssysteme wachsender Beliebtheit. Bei der biomedizinischen Authentifizierung wird versucht festzustellen, ob eine Person wirklich der- oder diejenige ist, als die er oder sie sich ausgibt. Methoden der biomedizinischen Authentifizierung umfassen etwa Netzhaut-Abtastung, Fingerabdrucksensoren oder Gesichtserkennung. Fingerabdrucksensoren werden üblicherweise mit der gleichen CMOS-Technologie [Weste et al., 2000] hergestellt wie integrierte Schaltkreise. Mögliche Anwendungsgebiete sind Laptops, die den Zugriff nur erlauben, wenn der Fingerabdruck des Benutzers identifiziert werden konnte. Die oben beschriebenen CCD- und CMOS-Bildsensoren werden zur automatischen Gesichtserkennung eingesetzt. Fälschlicherweise akzeptierte oder fälschlicherweise abgelehnte Benutzer stellen ein inhärentes Problem dieser biomedizinischen Authentifizierungsmethoden dar. Im Gegen-

[1] Es entfällt die Notwendigkeit, ein elektronisches *Display* über längere Zeiträume mit Bildinformationen zu versorgen, im Gegensatz zu Kameras mit unzulänglichem optischen Sucher.

satz zur Verwendung von Passwörtern ist eine exakte Übereinstimmung und somit 100%ige Sicherheit hier nicht zu erreichen.

- **Künstliche Augen:** Projekte zum Künstlichen Auge haben sehr viel Aufmerksamkeit erregt. Während einige Projekte sich direkt mit dem Auge beschäftigen, versuchen andere das Sehen auf indirektem Wege zu ermöglichen. Das Dobelle-Institut experimentierte mit einer Versuchsanordnung, bei der eine kleine Kamera in eine Brille eingebaut ist. Diese Kamera wird mit einem Computer verbunden, der die gesehenen Muster in elektrische Pulse umwandelt. Diese Impulse werden mittels einer Elektrode direkt an das Gehirn gesendet. Im Jahr 2003 lag die Auflösung bei etwa 128x128 Pixeln, womit eine blinde Person in einer festgelegten Umgebung Auto fahren konnte [Dobelle, 2003]. Neuere Techniken versuchen, Bildsignale in akustische Signale zu wandeln.
- **Andere Sensoren:** Andere weitverbreitete Sensoren sind u.a. Drucksensoren, Entfernungssensoren, Motorkontrollsensoren, Halleffektsensoren und viele weitere mehr.

3.2.2 Sample-and-Hold-Schaltungen

Alle üblichen digitalen Computer arbeiten mit diskreten Zeitwerten[2]. Das bedeutet, dass sie diskrete Folgen von Werten verarbeiten können. Kontinuierliche Werte müssen folglich in den diskreten Wertebereich umgewandelt werden. Dies wird von sogenannten *Sample-and-Hold-Schaltungen* übernommen. Abbildung 3.4 (links) zeigt eine einfache *Sample-and-Hold*-Schaltung.

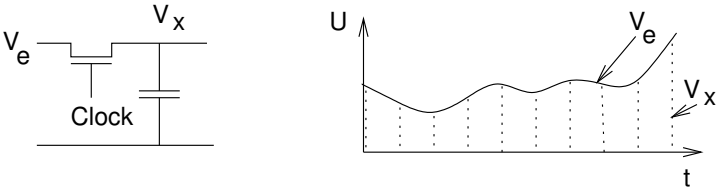


Abb. 3.4. *Sample-and-Hold*-Schaltung

Im Wesentlichen besteht die Schaltung aus einem getakteten Transistor und einem Kondensator. Der Transistor verhält sich wie ein Schalter. Immer, wenn der Schalter durch das Taktsignal geschlossen wird, wird der Kondensator aufgeladen, so dass seine Spannung praktisch der Eingangsspannung V_e entspricht. Wenn der Schalter wieder geöffnet wird, bleibt diese Spannung so lange erhalten, bis der Schalter wieder geschlossen wird. Jede in dem Kondensator gespeicherte Spannung kann als ein Element der diskreten Folge von

[2] Quantencomputer schließen wir hier aus.

Werten V_x angesehen werden, die aus der kontinuierlichen Folge V_e erzeugt wurden (s. Abb. 3.4, rechts).

Eine ideale *Sample-and-Hold*-Schaltung wäre in der Lage, die Spannung des Kondensators in einer beliebig kurzen Zeit zu ändern. So könnte die Eingangsspannung zu einem bestimmten Zeitpunkt auf den Kondensator übertragen werden und jedes Element der diskreten Folge würde der Eingangsspannung zu einem bestimmten Zeitpunkt entsprechen. In der Praxis muss der Transistor allerdings für eine gewisse Zeit leitend bleiben, damit sich der Kondensator tatsächlich laden oder entladen kann.

3.2.3 A/D-Wandler

Da wir uns auf digitale Computersysteme beschränken, muss nach der Diskretisierung der Zeiten auch eine Diskretisierung der Werte (auch **Quantisierung** genannt) vorgenommen werden. Diese Umwandlung von analogen zu digitalen Werten wird von A/D-Wandlern durchgeführt. Es gibt eine Vielzahl von A/D-Wandlern mit verschiedenen Geschwindigkeits- und Genauigkeits-Eigenschaften. Wir stellen hier zwei Varianten vor:

- ***Flash* A/D-Wandler** (Wandler nach der Methode des direkten Vergleichs): Diese Art A/D-Wandler verwendet viele Vergleicher. Jeder Vergleicher hat zwei Eingänge, die als '+' und '−' bezeichnet werden. Wenn die Spannung am '+'-Eingang diejenige am '−'-Eingang übersteigt, ist der Ausgang logisch '1', sonst ist er logisch '0'[3].

  Im A/D-Wandler sind alle '−'-Eingänge mit einem Spannungsteiler verbunden. Wenn die Eingangsspannung V_x die konstante Spannung V_{ref} übersteigt, erzeugt der oberste Vergleicher in Abb. 3.5 eine '1'. Der Kodierer am Ausgang des Vergleichers bestimmt die höchstwertige '1' und kodiert den Fall $V_x > V_{ref}$ als größten möglichen Ausgabewert.

  Wenn die Eingangsspannung V_x niedriger als V_{ref}, aber immer noch größer als $\frac{3}{4}V_{ref}$ ist, erzeugt der oberste Vergleicher in Abb. 3.5 eine '0', wohingegen der folgende Vergleicher immer noch eine '1' generiert. Der Kodierer wird diesen Fall als den zweithöchsten Wert ausgeben.

  Genauso kann man für die Fälle $\frac{2}{4}V_{ref} < V_x < \frac{3}{4}V_{ref}$, $\frac{1}{4}V_{ref} < V_x < \frac{2}{4}V_{ref}$, und $0 < V_x < \frac{1}{4}V_{ref}$ argumentieren, die entsprechend als drittgrößter, viertgrößter und als kleinster Wert kodiert werden. Der Kodierer liefert dabei stets eine Binärkodierung der Nummer des höchstwertigen Eingangs, der eine '1' führt.

[3] In der Praxis ist der Fall von genau identischen Spannungen irrelevant, da das tatsächliche Verhalten bei sehr kleinen Spannungsunterschieden von vielen Faktoren abhängt (etwa von der Temperatur, von Herstellungsprozessen usw).

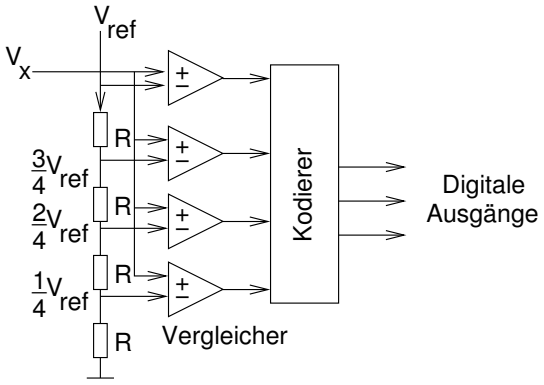


Abb. 3.5. *Flash* A/D-Wandler

Diese Schaltung kann positive analoge Spannungen in digitale Werte umwandeln. Die Wandlung von sowohl positiven als auch negativen Spannungen erfordert einige Erweiterungen.

Der Hauptvorteil dieser Schaltung ist ihre Geschwindigkeit. Sie benötigt keinen Takt. Die Verzögerung zwischen Ein- und Ausgang ist sehr kurz und die Schaltung kann beispielsweise leicht für Hochgeschwindigkeits-Videoanwendungen verwendet werden. Der Nachteil ist ihre hohe Hardware-Komplexität: es werden $n-1$ Vergleicher benötigt, um n Werte unterscheiden zu können. Man stelle sich vor, dieser Wandlertyp solle verwendet werden, um digitale Audiosignale für CD-Rekorder umzuwandeln: Dafür würden $2^{16}-1$ Vergleicher benötigt.

- **Sukzessive Approximation** (Schaltung nach dem Wägeprinzip): Die Unterscheidung vieler digitaler Werte ist mit Hilfe eines A/D-Wandlers nach dem Prinzip der sukzessiven Approximation möglich. Die Schaltung ist in Abb. 3.6 gezeigt.

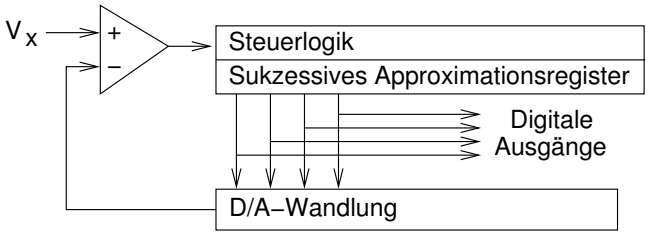


Abb. 3.6. A/D-Wandlerschaltung mit sukzessiver Approximation

Die Grundidee dieser Schaltung basiert auf der binären Suche. Anfangs wird das höchstwertige Ausgabe-Bit auf '1' gesetzt, alle anderen Bits auf

'0'. Dieser digitale Wert wird dann in einen analogen Wert umgewandelt, der $0,5\times$ der maximalen Eingangsspannung entspricht[4]. Wenn V_x diesen analogen Spannungswert übersteigt, bleibt das höchstwertige Bit auf '1', ansonsten wird es auf '0' zurückgesetzt.

Dieser Vorgang wird mit dem nächsten Bit wiederholt. Es bleibt auf '1', wenn die Eingangsspannung entweder im zweiten oder im vierten Viertel des Eingangsspannungsbereichs liegt. Dies wird mit allen weiteren Bits wiederholt.

Der Hauptvorteil der sukzessiven Approximationstechnik liegt in ihrer Hardware-Effizienz. Um n digitale Werte unterscheiden zu können, werden $log_2(n)$ Bits im Ausgaberegister und im D/A-Wandler benötigt. Der Nachteil der Schaltung ist ihre beschränkte Geschwindigkeit, da für jeden Wert $O(log_2(n))$ Schritte benötigt werden. Diese A/D-Wandler werden daher in Anwendungen eingesetzt, bei denen hohe Präzision und mittlere Geschwindigkeit erforderlich sind. Beispiele hierfür sind etwa Audio-Anwendungen.

Es gibt noch einige andere Arten von A/D-Wandlern sowie Techniken, um automatisch den für eine Anwendung am besten geeigneten Wandler auszuwählen [Vogels und Gielen, 2003].

3.3 Kommunikation

Bevor Informationen in einem eingebetteten System verarbeitet werden können, müssen sie erst einmal verfügbar sein. Die Übertragung der Information kann über einen von vielen möglichen sogenannten **Kanälen** erfolgen. Kanäle sind abstrakte Gebilde, die durch die Eigenschaften der Kommunikation charakterisiert werden, etwa durch die maximale Übertragungsgeschwindigkeit oder die Störparameter. Die Wahrscheinlichkeit für Fehler bei der Kommunikation wird mit Hilfe von Techniken aus der Kommunikationstheorie berechnet. Die physikalische Grundlage, auf der die Kommunikation erfolgt, wird als **Medium** bezeichnet. Wichtige Medienklassen sind etwa drahtlose Medien (Radiowellen-Übertragung, Infrarot), optische Medien (Lichtwellenleiter) oder elektrische Leitungen.

Bei der Vielzahl von Klassen eingebetteter Systeme gibt es auch viele unterschiedliche Anforderungen an die Kommunikation. Im Allgemeinen ist das Verbinden von Hardwarekomponenten eingebetteter Systeme alles andere als einfach. Einige häufig vorkommende Anforderungen werden im nächsten Abschnitt aufgezählt.

[4] Glücklicherweise kann die Umwandlung von digitalen zu analogen Werten (D/A-Wandlung) sehr effizient und schnell realisiert werden (s. Seite 132).

3.3.1 Anforderungen

Die folgende Liste beschreibt einige Anforderungen, die bei der Kommunikation eingehalten werden müssen:

- **Echtzeit-Verhalten:** Diese Anforderung hat weitreichende Auswirkungen auf den Entwurf des Kommunikations-Systems. Einige kostengünstige Lösungen, wie etwa Ethernet, erfüllen diese Anforderung nicht.
- **Effizienz:** Die Verbindung zweier Hardwarekomponenten kann recht teuer sein. Beispielsweise ist eine direkte Punkt-zu-Punkt-Verbindung in einem großen Gebäude nahezu unmöglich. Im Automobilbereich hat sich gezeigt, dass separate Kabel, welche die Steuereinheiten mit der externen Peripherie verbinden, sehr teuer und schwer sind. Einzelne Kabel machen es auch schwierig, neue Komponenten anzuschließen. Die Kosten haben auch einen Einfluss auf den Entwurf der Stromversorgung externer Geräte. Häufig wird eine zentrale Stromversorgung eingesetzt, um Kosten zu sparen.
- **Angemessene Bandbreite und Kommunikations-Verzögerung:** Die geforderten Bandbreiten eingebetteter Systeme unterscheiden sich sehr stark. Die zur Verfügung stehende Bandbreite muss den Anforderungen genügen, ohne das System unnötig zu verteuern.
- **Unterstützung für Ereignis-gesteuerte Kommunikation:** Systeme, die auf dem regelmäßigen Abfragen von Geräten (sog. *Polling*) basieren, besitzen ein sehr gut vorhersagbares Echtzeitverhalten. Die Verzögerungen können bei dieser Art der Kommunikation allerdings zu groß sein. Oft wird eine schnelle, ereignisgesteuerte Kommunikation benötigt. Notfallbedingungen sollten beispielsweise sofort weitergeleitet werden und nicht solange unbemerkt bleiben, bis ein zentrales Systeme alle Geräte nach neuen Nachrichten abfragt.
- **Robustheit:** Eingebettete Systeme sollen bei extremen Temperaturen oder in der Nähe von Quellen elektromagnetischer Strahlung eingesetzt werden. Automotoren etwa sind Temperaturen von -20 bis zu +180 °C ausgesetzt. Spannungspegel und Taktfrequenzen können von solch hohen Temperaturschwankungen beeinflusst werden. Trotzdem muss eine verlässliche Kommunikation gewährleistet werden.
- **Fehlertoleranz:** Trotz aller Bemühungen, eine robuste Kommunikation zu erreichen, können Fehler auftreten. Eingebettete Systeme sollten auch nach einem solchen Fehler funktionsfähig bleiben. Neustarts, wie man sie vom PC her gewohnt ist, sind inakzeptabel. Wenn eine Kommunikation also fehlgeschlagen ist, sind Wiederholungsversuche notwendig. Hier entsteht ein Konflikt mit der ersten Anforderung: wenn man mehrere Kommunikationsversuche zulässt, ist es schwierig, das Echtzeitverhalten zu garantieren.
- **Wartbarkeit, Diagnosefähigkeit:** Es ist offensichtlich, dass man eingebettete Systeme in einer annehmbar kurzen Zeit reparieren können muss.

- **Verschlüsselung:** Um sicherzustellen, dass private und vertrauliche Informationen geheim bleiben, kann es notwendig sein, bei der Kommunikation Verschlüsselungstechniken einzusetzen.

Diese Anforderungen an die Kommunikation sind eine direkte Folge der allgemeinen Eigenschaften eingebetteter Systeme, die in Kapitel 1 gezeigt wurden. Wegen der Konflikte zwischen einigen der Anforderungen müssen in der Praxis Kompromisse eingegangen werden. Beispielsweise kann es unterschiedliche Kommunikationsmodi geben: einen Modus mit hoher Bandbreite, der Echtzeitverhalten garantiert, aber keine Fehlertoleranz bietet (diese Übertragungsart wäre etwa geeignet für Multimedia-Daten), und einen weiteren fehlertoleranter Modus mit geringerer Bandbreite, der z.B. für kurze Nachrichten verwendet wird, die nicht verlorengehen dürfen.

3.3.2 Elektrische Robustheit

Es gibt einige grundlegende Techniken, um elektrische Robustheit zu erreichen. Digitale Kommunikation innerhalb eines Chips findet üblicherweise mit **asymmetrischen Signalen** statt (sog. *asymmetric* oder *single-ended signaling*). Dabei werden die Signale über eine einzige Leitung geschickt (s. Abb. 3.7).

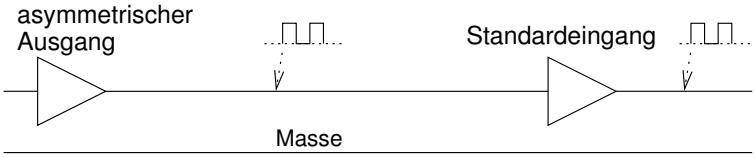


Abb. 3.7. Asymmetrische Signale

Solche Signale werden durch eine Spannung gegenüber einer einheitlichen Masse dargestellt, seltener durch unterschiedliche Ströme. Eine einzige Masseleitung ist ausreichend für eine größere Anzahl solcher Signale. Asymmetrische Signale sind allerdings sehr anfällig für externe Störeinflüsse. Wenn solche Störungen (die z.B. durch einen anlaufenden Motor verursacht werden können) die Spannung beeinflussen, können Nachrichten verfälscht werden. Es ist auch schwierig, einen genau definierten Massepegel zwischen verschiedenen kommunizierenden Einheiten bereitzustellen, da die Masseleitungen selber auch einen gewissen Widerstand und eine Induktivität haben. Dies ist bei der **symmetrischen** oder **differentiellen Signalübertragung** anders. Bei der symmetrischen Übertragung benötigt jedes Signal zwei Leitungen (s. Abb. 3.8).

Bei der symmetrischen Signalübertragung werden binäre Werte wie folgt kodiert: wenn die Spannung auf der ersten Leitung gegenüber der zweiten po-

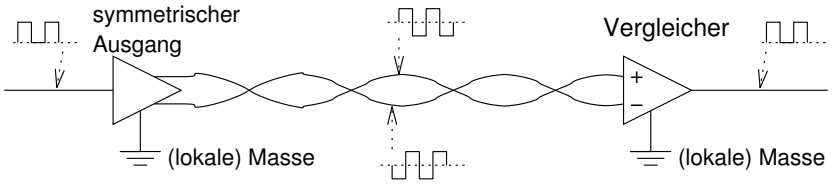


Abb. 3.8. Symmetrische Signalübertragung

sitiv ist, wird dies als '1' dekodiert, sonst ist der übertragene Wert eine '0'. Die beiden Leitungen werden üblicherweise miteinander verdrillt und bilden sogenannte *twisted pairs*. Es gibt lokale Massepegel, aber ein Spannungsunterschied zwischen diesen lokalen Massepegeln stört die Kommunikation nicht. Vorteile der symmetrischen Signalübertragung sind u.a.:

- Störungen wirken prinzipiell auf beide Leitungen in ähnlicher Weise ein. Beim Vergleichen werden daher die meisten Störungen entfernt.
- Der Logikwert hängt ausschließlich von der Polarität der Spannung zwischen den beiden Leitungen ab. Die Stärke der Spannung kann durch Reflexionen oder den Leitungswiderstand verändert werden – dies hat jedoch keinen Einfluss auf den übermittelten Wert.
- Signale verursachen keine Ströme auf den Masseleitungen, wodurch die Qualität des Massepegels weniger kritisch ist.
- Es wird kein gemeinsames Massepotential benötigt. Daher entfällt auch die Notwendigkeit, eine große Anzahl von Kommunikationspartnern mit einem hochqualitativen gemeinsamen Masseanschluss zu versehen. Dies ist einer der Gründe, warum symmetrische Signale bei Ethernet verwendet werden.
- Als Konsequenz der bisher genannten Eigenschaften erreicht die symmetrische Signalübertragung einen höheren Durchsatz als die asymmetrische Übertragung.

Allerdings müssen für differentielle Signale stets zwei Leitungen für jedes Signal zur Verfügung stehen. Außerdem benötigt man negative Spannungen, wenn man nicht nur jeweils entgegengesetzte Logikpegel der asymmetrischen Signale verwendet.

3.3.3 Garantieren von Echtzeitverhalten

Die meisten Rechner-Netzwerke basieren auf den Ethernet-Standards. Für die 10 MBit/s- und 100 MBit/s-Versionen kann es zwischen den Kommunikationspartnern zu Kollisionen bei der Datenübertragung kommen. Das bedeutet, dass mehrere Partner zur gleichen Zeit versuchen, Daten zu übertragen, wodurch sich die Signale auf den Leitungen stören. Jedes Mal, wenn

das passiert, müssen die Kommunikationspartner die Kommunikation einstellen, einige Zeit warten und es dann erneut versuchen. Die Wartezeit wird zufällig bestimmt, um die Wahrscheinlichkeit für einen erneuten Konflikt beim nächsten Versuch zu reduzieren. Diese Methode heißt ***carrier-sense multiple access/collision detect*** (CSMA/CD). Bei Verwendung von CSMA/CD können die Kommunikationszeiten sehr lang werden, da Konflikte sich häufig wiederholen können, obwohl dies nicht sehr wahrscheinlich ist. Daher kann CSMA/CD nicht eingesetzt werden, wenn Echtzeitbedingungen eingehalten werden müssen.

Das Problem kann mit Hilfe von CSMA/CA (***carrier-sense multiple access/collision avoidance***) vermieden werden. Kollisionen werden bei diesem Verfahren komplett vermieden, anstatt sie nur zu entdecken und darauf zu reagieren. Bei CSMA/CA gibt es für jeden Kommunikationspartner eine Priorität. Das Übertragungsmedium wird den Partnern während sogenannter Arbitrierungsphasen zugeordnet, darauf folgen dann die eigentlichen Kommunikationsphasen. Während der Arbitrierungsphase geben die Partner ihren Kommunikationswunsch auf dem Übertragungsmedium bekannt. Wenn ein anderer Teilnehmer mit einer höheren Priorität ebenfalls senden möchte, müssen alle anderen ihren Kommunikationswunsch sofort zurückziehen.

Unter der Annahme, dass es eine obere Schranke für die Zeit zwischen den Arbitrierungsphasen gibt, garantiert CSMA/CA ein vorhersagbares Echtzeitverhalten für den Kommunikationspartner mit der höchsten Priorität. Für die anderen Teilnehmer kann das Echtzeitverhalten nur dann garantiert werden, wenn die höher priorisierten Partner nicht andauernd senden möchten.

Die Hochgeschwindigkeits-Version von Ethernet (1 GBit/s) basiert auch auf der Vermeidung von Kollisionen.

3.3.4 Beispiele

- **Sensor-Aktuator-Busse:** Sensor-Aktuator-Busse ermöglichen die Kommunikation zwischen einfachen Geräten wie etwa Schaltern und Lampen und den datenverarbeitenden Geräten. Die Anzahl der angesteuerten Geräte kann sehr groß werden, daher müssen die Kosten der Verdrahtung in diesem Fall besonders beachtet werden.
- **Feldbusse:** Feldbusse sind den Sensor-Aktuator-Bussen ähnlich, sie unterstützen jedoch im Allgemeinen höhere Datenraten. Beispiele für Feldbusse sind u.a. folgende:
  - ***Controller Area Network* (CAN-Bus):** Dieser Bus wurde 1981 von Bosch und Intel entwickelt, um Steuereinheiten und Peripherie zu verbinden. Er ist insbesondere im Automobilbereich beliebt, da man mit dem CAN-Bus eine große Anzahl von Leitungen durch einen einzigen Bus ersetzen kann. Wegen der Größe des Automobilmarktes sind

CAN-Komponenten verhältnismäßig preiswert und werden daher auch in anderen Bereichen, etwa zur Heim-Automatisierung und in Fabriksteuerungen eingesetzt. CAN hat die folgenden Eigenschaften:

- differentielle Signalübertragung über *Twisted-Pair*-Kabel,
- Arbitrierung mittels CSMA/CA,
- Durchsatz zwischen 10 kBit/s und 1 MBit/s
- Niedrig- und hochpriorisierte Signale,
- maximale Latenz von 134 μs für hochpriorisierte Signale,
- Kodierung der Signale ähnlich wie bei der seriellen (RS-232) Übertragung bei PCs, mit Modifikationen für differentielle Signalübertragung.

– Beim ***Time-Triggered-Protocol* (TTP)** für fehlertolerante Sicherheits-Systeme (etwa Airbags in Autos) [Kopetz und Grunsteidl, 1994], erfolgt bereits zur Entwurfszeit ein Scheduling für die Nachrichtenübertragungen. Damit werden Kollisionen auf Leitungen zur Laufzeit und damit ein schlecht vorhersehbares zeitliches Verhalten während der Laufzeit ausgeschlossen.

– **FlexRay** [FlexRay Consortium, 2002] ist ein sogenanntes TDMA (*Time Divison Multiple Access*)-Protokoll, das vom FlexRay-Konsortium (bestehend aus BMW, Daimler-Chrysler, General Motors, Ford, Bosch, Motorola und Philips Semiconductors) ausgearbeitet wurde. FlexRay ist eine Kombination einer Variante von TTP und dem ***byteflight*-Protokoll** [Byteflight Consortium, 2003].

– **MAP:** MAP ist ein Bus, der für Automobilfabriken entworfen wurde.

– **EIB:** Der *European Installation Bus* (EIB) wurde für die Heimautomatisierung entwickelt.

- **Drahtlose Kommunikation:** Drahtlose Kommunikation wird immer beliebter, allerdings wird die Bandbreite zur knappen Ressource. Aus diesem Grund wurden die UMTS Mobilfunk-Frequenzen zu extrem hohen Preisen verkauft (ca. 500 Euro pro Einwohner Deutschlands).

  Bluetooth ist ein Standard, um Geräte wie Mobiltelefone und Kopfhörer miteinander zu verbinden.

  Die drahtlose Ethernet-Version wurde von der IEEE als 802.11 standardisiert. Dieser Standard wird in lokalen Netzwerken (*LANs*) verwendet.

  DECT ist ein Standard, der in Europa für drahtlose Telefone verwendet wird.

3.4 Verarbeitungseinheiten

3.4.1 Überblick

Zur Informationsverarbeitung werden wir anwendungsspezifische integrierte Schaltkreise (*Application-Specific Integrated Circuits*, (ASICs)) mit festverdrahteter Funktionalität, rekonfigurierbare Logik sowie Prozessoren betrachten. Diese drei Technologien unterscheiden sich recht stark, z.B. bezüglich ihres Energiebedarfs. Abbildung 3.9 (in Anlehnung an de Man [De Man, 2002]) zeigt die Anzahl von Operationen pro Watt, die mit einer bestimmten Hardware-Technologie erreicht werden kann:

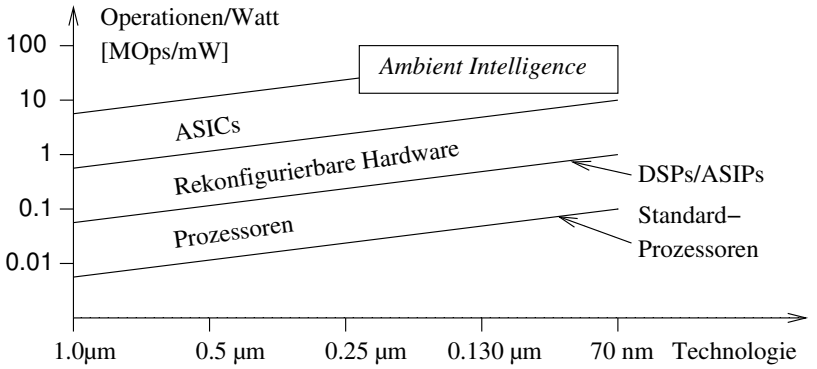


Abb. 3.9. Hardware-Effizienz

Offenbar erreicht man mehr Operationen pro Watt, wenn die verwendete Technologie immer kleinere Strukturgrößen der integrierten Schaltkreise erlaubt. Allerdings ist für eine bestimmte Technologie die Anzahl der Operationen pro Watt für einen applikationsspezifischen Schaltkreis am größten. Für rekonfigurierbare Logikbausteine (s. Seite 126) ist dieser Wert eine Zehnerpotenz niedriger. Bei programmierbaren Prozessoren sind es zwei Zehnerpotenzen. Andererseits bieten Prozessoren die größte Flexibilität, die sich direkt aus der Flexibilität der Software ergibt. Auch bei rekonfigurierbarer Logik hat man eine gewisse Flexibilität, aber sie wird durch die maximale Größe der Applikationen, die man auf solche Bauteile abbilden kann, begrenzt. Bei festverdrahteten Entwürfen gibt es keine Flexibilität. Diese Beobachtung gilt auch für Prozessoren: Wenn ein Prozessor für einen bestimmten Anwendungsfall, wie etwa die digitale Signalverarbeitung (*Digital Signal Processing* (DSP)), optimiert wurde, erreichen dessen Energie-Effizienzwerte beinahe die von rekonfigurierbarer Logik. Für allgemeine, nicht spezialisierte Prozessoren sind die Effizienzwerte am schlechtesten.

Die Energie E für eine bestimmte Anwendung hängt eng mit der Leistung P zusammen, die pro Operation verbraucht wird, denn es gilt:

$$E = \int P dt$$

Eine Reduktion des Leistungsverbrauchs reduziert also auch den Energieverbrauch, wenn das Integral über den gleichen Zeitraum bestimmt wird. In einigen Fällen kann allerdings eine starke Reduktion der Laufzeit bei nur geringer Erhöhung der Leistung erreicht werden (z.B. aufgrund einer Erhöhung der Speichergeschwindigkeit). Dies kann den Energieverbrauch minimieren. In einigen Fällen entspricht also eine minimierte Leistung auch einer minimierten Energie. Dieser einfache Zusammenhang ist im Allgemeinen aber nicht gegeben.

Sowohl die Minimierung der Leistung als auch der Energie sind wichtige Ziele. Der Leistungsverbrauch hat einen Einfluss auf die Größe der Spannungsversorgung, auf den Entwurf der Spannungsregler, auf die Dimensionen der Verbindungsleitungen und auf den kurzzeitigen Kühlungsbedarf. Das Minimieren des Energieverbrauchs ist insbesondere für mobile Geräte wichtig, da sich die Kapazitäten der verfügbaren Batterien nur langsam verbessern [SEMATECH, 2003] und die Energiekosten daher sehr hoch werden können. Eine verringerte Energieaufnahme verringert auch den Kühlungsaufwand und verbessert die Zuverlässigkeit (da die Lebensdauer von elektronischen Schaltungen bei erhöhter Temperatur reduziert wird).

Abbildung 3.9 zeigt den Effizienz-/Flexibilitätskonflikt der aktuell verfügbaren Hardware-Technologien: wenn man auf sehr gute Leistungs- und Energieeffizienz Wert legt, sollte man keine flexiblen Entwürfe auf Basis von Prozessoren oder programmierbarer Logik verwenden. Wenn man sehr flexibel sein möchte, ist man nicht energieeffizient. Wir betrachten im Folgenden zuerst die anwendungsspezifischen Schaltkreise.

3.4.2 Anwendungsspezifische integrierte Schaltkreise (ASICs)

Für Hochleistungsanwendungen und für große Märkte können anwendungsspezifische integrierte Schaltkreise (*Application-Specific Integrated Circuits* (ASICs)) hergestellt werden. Die Kosten eines solchen ASIC-Entwurfs und der Fertigung sind allerdings sehr hoch. Diese Kosten wachsen ebenso exponentiell wie die Komplexität der gefertigten Schaltkreise. Die Kosten für einen vollständigen Maskensatz eines Chips liegen bereits jenseits einer Million Dollar. Daher sind ASICs nur einsetzbar, wenn maximale Energieeffizienz benötigt wird, wenn der Markt die entsprechenden Kosten akzeptiert, oder wenn eine sehr große Anzahl solcher Systeme verkauft werden kann.

3.4.3 Prozessoren

Der Hauptvorteil von Prozessoren ist ihre Flexibilität. Wenn Prozessoren verwendet werden, kann man das Verhalten eines eingebetteten Systems komplett verändern, indem man einfach nur die Software austauscht, die von den Prozessoren ausgeführt wird. Solche Verhaltensänderungen können notwendig werden, um Entwurfsfehler auszubessern, um das System auf einen aktualisierten oder verbesserten Stand zu bringen oder um dem System zusätzliche Fähigkeiten zu geben. Aufgrund dieser Möglichkeiten sind Prozessoren in letzter Zeit sehr populär geworden. Diese Beliebtheit ist auch in der Presse betont worden:

„Auf der Ebene der Chips enthalten eingebettete Chips Mikrocontroller und Mikroprozessoren. Mikrocontroller sind die wahren Arbeitspferde der eingebetteten Systeme. Sie sind die ursprünglichen 'eingebetteten Chips' und umfassen auch die Chips, die zuerst als Regelungseinheiten in Aufzügen und Thermostaten eingesetzt wurden“ [Ryan, 1995]

Eingebettete Prozessoren müssen effizient sein, auch wenn sie die Effizienz von ASICs in der Regel nicht erreichen können. Außerdem haben sie nicht den Zwang zur Befehlssatz-Kompatibilität mit den allgemein verwendeten PCs. Aus diesem Grunde können ihre Architekturen sich von denen der PCs unterscheiden. Aus der Notwendigkeit der Effizienz ergibt sich eine Anzahl von Aspekten (s. Seite 2):

- **Energie-Effizienz:** Architekturen müssen in Bezug auf die Energieeffizienz optimiert werden. Gleichzeitig muss sichergestellt werden, dass während der Erzeugung der Software keine Effizienz verschenkt wird. Compiler, die beispielsweise die benötigte Anzahl von Zyklen einer Anwendung um 50% erhöhen, führen das System weit weg von der Energieeffizienz von ASICs, möglicherweise sogar um mehr als eben diese 50%, da ja evtl. die Spannung und die Taktfrequenz angepasst werden müssen, um die Zeitbedingungen einhalten zu können.

  Es gibt eine Vielzahl von Techniken, die Prozessoren energieeffizient machen, und die Energieeffizienz sollte auf verschiedenen Abstraktionsebenen berücksichtigt werden, vom Entwurf des Befehlssatzes bis hin zum Herstellungsprozess des eigentlichen Chips [Burd und Brodersen, 2003]. Die Verwendung von sogenannten *gated clocks* ist eine Möglichkeit. Hierbei werden Teile des Prozessors vom Takt abgekoppelt, wenn sie längere Zeit nicht benötigt werden. So wird etwa kein Taktsignal an die Multiplizierereinheit herangeführt, wenn gerade keine Multiplikation durchzuführen ist. Es gibt auch Ansätze, den zentralen Takt für große Teile eines Prozessors insgesamt abzuschaffen. Dabei gibt es zwei gegensätzliche Ansätze: global synchrone, lokal asynchrone Prozessoren auf der einen Seite und global asynchrone, lokal synchrone (GALS) [Iyer und Marculescu, 2002] auf der anderen Seite.

Zwei Techniken können auf einer höheren Abstraktionsebene angewendet werden:

– **Dynamisches *Power-Management* (DPM):** Bei diesem Ansatz haben die Prozessoren neben dem normalen Betriebszustand verschiedene Energiesparzustände. Jeder dieser Energiesparzustände verfügt über einen unterschiedlichen Energieverbrauch und über eine unterschiedliche Zeit, bis der normale Betriebszustand wieder erreicht wird. Abbildung 3.10 zeigt drei Zustände des StrongArm SA 1100 Prozessors.

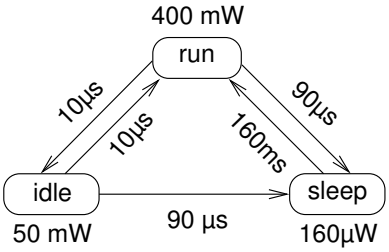


Abb. 3.10. Dynamische *Power-Management*-Zustände des StrongArm Prozessors SA 1100

Der Prozessor ist im Zustand run voll funktionsfähig. Im idle-Zustand werden ausschließlich die Interrupt-Eingänge überwacht. Im sleep-Zustand ist jegliche Aktivität auf dem Chip abgeschaltet. Man beachte den großen Unterschied des Energiebedarfs zwischen dem sleep- und den beiden anderen Zuständen, ebenso wie die große Verzögerungszeit, um vom sleep-Zustand wieder in den run-Zustand zu wechseln.

– **Dynamische Anpassung der Versorgungsspannung:** Der Ansatz der dynamischen Anpassung der Versorgungsspannung (*Dynamic Voltage Scaling* (DVS)) nutzt die quadratische Reduktion des Energieverbrauchs von CMOS-Schaltkreisen bei linearer Reduktion der Versorgungsspannung V_{dd}. Die Leistung P von CMOS-Schaltkreisen wird wie folgt berechnet [Chandrakasan et al., 1992]:

$$P = \alpha \;\; C_L \; V_{dd}^2 \; f \tag{3.1}$$

wobei α die Schaltaktivität, C_L die Leitungskapazität, V_{dd} die Versorgungsspannung und f die Taktfrequenz ist. Die Verzögerung von CMOS-Schaltkreisen kann durch die folgende Gleichung angenähert werden [Chandrakasan et al., 1992], [Chandrakasan et al., 1995]:

$$\tau = k \cdot C_L \cdot \frac{V_{dd}}{(V_{dd} - V_t)^2} \tag{3.2}$$

wobei k eine Konstante und V_t die Schwellspannung (*threshold voltage*) ist. V_t hat einen Einfluss auf die Spannung, die benötigt wird, um den Transistor einzuschalten. Beispielsweise kann V_t für eine maximale Versorgungsspannung $V_{dd,max}$ von 3,3 Volt 0,8 Volt betragen. Aus Gleichung 3.2 folgt, dass die maximale Taktfrequenz eine Funktion der Versorgungsspannung ist. Das Verringern der Versorgungsspannung reduziert die Leistungsaufnahme allerdings quadratisch, während die Laufzeit von Programmen nur linear zunimmt (wenn man den Einfluss des Speichersystems vernachlässigt). Dies wird beim sogenannten *Dynamic Voltage Scaling* (DVS) ausgenutzt. Der Crusoe^{TM}-Prozessor von Transmeta verfügt beispielsweise über 32 Spannungswerte zwischen 1,1 und 1,6 Volt. Die Taktfrequenz kann zwischen 200 und 700 MHz in Schritten von 33 MHz verändert werden. Übergänge von einem Spannungs-/Frequenz-Paar zu einem anderen benötigen ca. 20 ms. Zwei unterschiedliche Geschwindigkeits/Spannungs-Paare gibt es bei der SpeedStep^{TM}-Technologie von Intel® für den Mobile Pentium® III Prozessor. Einige Aspekte, die beim Entwurf von DVS-fähigen Prozessoren beachtet werden müssen, werden von Burd und Brodersen [Burd und Brodersen, 2000] beschrieben.

- **Codegrößen-Effizienz:** Die Minimierung der Codegröße ist ein sehr wichtiger Aspekt für eingebettete Systeme, da diese üblicherweise keine Festplatte haben und somit die Speicherkapazität dieser Systeme stark beschränkt ist. Das gilt um so mehr für sogenannte *Systems On a Chip* (SoCs). Bei SoCs werden Prozessor und Speicher auf dem selben Chip untergebracht. In diesem Fall nennt man den Speicher **eingebetteten Speicher**. Eingebettete Speicher können in der Herstellung teurer sein als separate Speicherchips, da die Herstellungsverfahren für Prozessor und Speicher kompatibel sein müssen. Trotzdem wird ein Großteil der Chipfläche von Speichern eingenommen. Es gibt verschiedene Techniken, um die Codegrößen-Effizienz zu verbessern:
  - **CISC-Maschinen:** Standard RISC-Prozessoren werden mit dem Ziel einer hohen Geschwindigkeit und nicht Codegrößeneffizienz entworfen. Frühere Prozessoren mit komplexem Befehlssatz (*Complex Instruction Set Computers* (CISC)) wurden tatsächlich im Hinblick auf Codegrößen-Effizienz entwickelt, da sie mit sehr langsamen Speichern verbunden waren, und nur selten Caches verwendet wurden. Daher finden „altmodische“ CISC-Rechner in heutigen eingebetteten Systemen Verwendung. Motorolas *ColdFire*-Prozessoren, die auf der Motorola 68000-Familie basieren, sind Beispiele für solche CISC-Prozessoren.
  - **Kompressions-Techniken:** Um den Platzbedarf für Instruktionen auf dem Chip zu reduzieren, und um die Energie für das Laden dieser Instruktionen zu verringern, werden Befehle häufig in komprimierter Form im Speicher abgelegt. Wegen der daraus entstehenden Verringerung der benötigten Bandbreite kann das Laden der Befehle mögli-

cherweise sogar schneller erfolgen. Ein (hoffentlich kleiner und schneller) Dekodierer wird zwischen den Prozessor und den (Instruktions-) Speicher platziert und erzeugt die eigentlichen Instruktionen ohne weitere Verzögerung (s. Abb. 3.11, rechts). Statt einen potentiell großen Speicher zum Speichern von unkomprimierten Befehlen zu verwenden, werden diese also in komprimierter Form abgelegt.

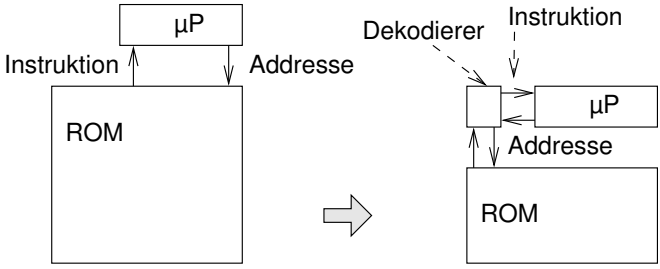


Abb. 3.11. Dekodierung von komprimierten Instruktionen

Die Ziele der Komprimierung können wie folgt zusammengefasst werden:

- Einsparen von ROM- und RAM-Bereichen, da diese potentiell teurer sein können als der Prozessor selbst.
- Verwenden von Kodierungstechniken für Instruktionen, möglicherweise auch für Daten, mit den folgenden Eigenschaften:
  - Wenig oder kein Laufzeit-Zuwachs für diese Techniken.
  - Das Dekodieren sollte auch mit begrenztem Kontext funktionieren (beispielsweise sollte man nicht erst das ganze Programm lesen müssen, um ein Sprungziel zu bestimmen).
  - Die Wort-Größe des Speichers, der Instruktionen und der Adressen müssen berücksichtigt werden.
  - Sprung-Instruktionen, die zu beliebigen Adressen verzweigen, müssen unterstützt werden.
  - Eine schnelle Kodierung wird nur benötigt, wenn zu schreibende Daten komprimiert werden sollen. Ansonsten ist eine schnelle Dekodierung ausreichend.

Es gibt verschiedene Variationen dieser Verfahren:

- Bei einigen Prozessoren gibt es einen **zweiten Befehlssatz**. Dieser zweite Befehlssatz hat ein kürzeres Befehlsformat. Ein Beispiel hierfür ist die ARM-Prozessorfamilie. Der ARM-Befehlssatz ist ein 32-Bit-Befehlssatz. Er unterstützt u.a. die bedingte Ausführung von Befehlen (*predicated execution*). Das bedeutet, dass eine Instruktion

nur dann ausgeführt wird, wenn eine bestimmte Bedingung erfüllt ist (s. Seite 123). Diese Bedingung wird in den ersten vier Bits des Befehlswortes kodiert. Die meisten ARM-Prozessoren verfügen über einen zweiten Befehlssatz, der nur 16 Bit breite Befehlsworte, die sogenannten THUMB-Befehle, verwendet. THUMB-Instruktionen sind kürzer, da sie keine bedingte Ausführung unterstützen, kürzere und weniger Registerfelder und kürzere Immediate-Felder verwenden (s. Abb. 3.12).

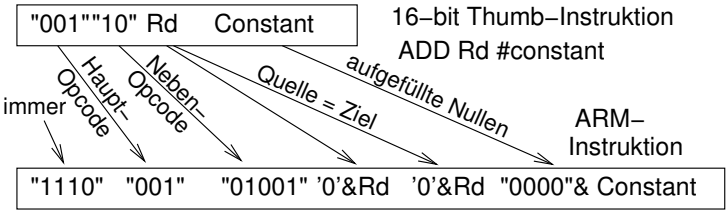


Abb. 3.12. Dekodieren von THUMB-Instruktionen zu ARM-Instruktionen

THUMB-Instruktionen werden dynamisch in ARM-Instruktionen umgewandelt, während das Programm abläuft. THUMB-Instruktionen können nur die Hälfte der Register in arithmetischen Befehlen ansprechen. Daher werden die Registerfelder in THUMB-Instruktionen mit einem '0'-Bit konkateniert[5]. Im THUMB-Befehlssatz sind Quell- und Zielregister häufig identisch und die maximale Länge von direkt im Befehlswort kodierten Konstanten ist je nach Befehl auf 4 oder 10 Bit begrenzt. Bei der Dekodierung wird *Pipelining* verwendet, um die zusätzlich benötigte Zeit zu minimieren.

Ähnliche Techniken existieren auch für andere Prozessoren. Der Nachteil dieses Ansatzes besteht darin, dass alle Software-Tools (Compiler, Assembler, Linker, Debugger usw.) den zweiten Befehlssatz unterstützen müssen. Daher kann dieser Ansatz unter Berücksichtigung der Softwareentwicklungskosten recht teuer werden.

- Ein zweiter Ansatz ist die Verwendung eines **Wörterbuchs**, eines sogenannten *Dictionaries*. Bei diesem Ansatz wird jede Instruktion nur einmal in einer Tabelle abgespeichert. Für jeden Wert des Programmzählers liefert eine zweite Tabelle einen Zeiger auf die entsprechende Instruktion im Wörterbuch (s. Abb. 3.13).

  Dieser Ansatz basiert auf der Annahme, dass es nur wenige unterschiedliche Instruktionsmuster gibt. Daher werden nur wenige Ein-

[5] Bei Verwendung der VHDL-Notation (s. Seite 64) wird die Konkatenation durch das Zeichen '&' dargestellt, und Konstanten werden in Abb. 3.12 von Anführungszeichen umschlossen.

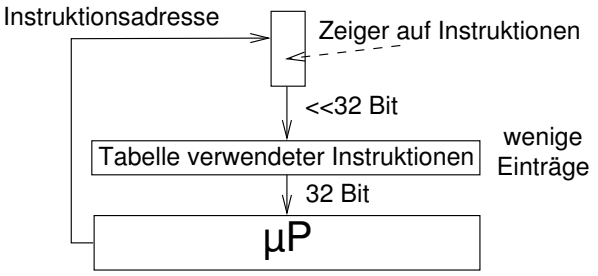


Abb. 3.13. *Dictionary*-Ansatz zur Instruktions-Kompression

träge im Wörterbuch benötigt. Folglich kann die Bitbreite der Zeiger auf diese Einträge recht klein sein. Es gibt viele Variationen dieses Ansatzes. Einige heißen *two-level control store* [Dasgupta, 1979] oder *nanoprogramming* [Stritter und Gunter, 1979]. *Procedure exlining* [Vahid, 1995] ist ein verwandter Ansatz, bei dem gemeinsame Codesequenzen nur einmal im Speicher abgelegt werden und der originale Code durch einen Sprung an den gemeinsamen Code ersetzt wird.

Eine umfassende Zusammenfassung bekannter Kompressionstechniken gibt es im Internet [Latendresse, 2004].

- **Laufzeit-Effizienz:** Um Echtzeitanforderungen erfüllen zu können, ohne hohe Taktfrequenzen zu verwenden, können Architekturen an bestimmte Anwendungsgebiete, wie etwa die digitale Signalverarbeitung (*Digital Signal Processing* (DSP)) angepasst werden. Man kann sogar noch einen Schritt weitergehen und Prozessoren mit einem anwendungsspezifischen Befehlssatz entwerfen (*Application Specific Instruction Set Processors* (ASIPs)). Als Beispiel für Anwendungsbereichs-spezifische Prozessoren betrachten wir DSP-Prozessoren. In der digitalen Signalverarbeitung ist das Filtern von Signalen eine wichtige und häufig verwendete Funktion. Gleichung 3.3 beschreibt einen digitalen Filter, der aus einer Eingabesequenz $x = (x_0, x_1, ...)$ eine Ausgabesequenz $y = (y_0, y_1, ...)$ bestimmt.

$$y_i = \sum_{j=0}^{n-1} x_{i-j} * a_j \tag{3.3}$$

Ein Element y_i der Ausgabesequenz entspricht dem gewichteten Mittelwert der letzten n Sequenzelemente von x und kann iterativ unter Verwendung der Gleichungen 3.4 bis 3.6 berechnet werden.

$$y_{i,j} = y_{i,j-1} + x_{i-j} * a_j \tag{3.4}$$

$$\text{mit} \ : \ y_{i,-1} = 0 \tag{3.5}$$

$$\text{und} \ : \ y_i = y_{i,n-1} \tag{3.6}$$

DSPs werden so entworfen, dass jede Iteration als eine einzige Instruktion kodiert werden kann. Als Beispiel betrachten wir den in Abb. 3.14 gezeigten ADSP 2100 DSP-Prozessor.

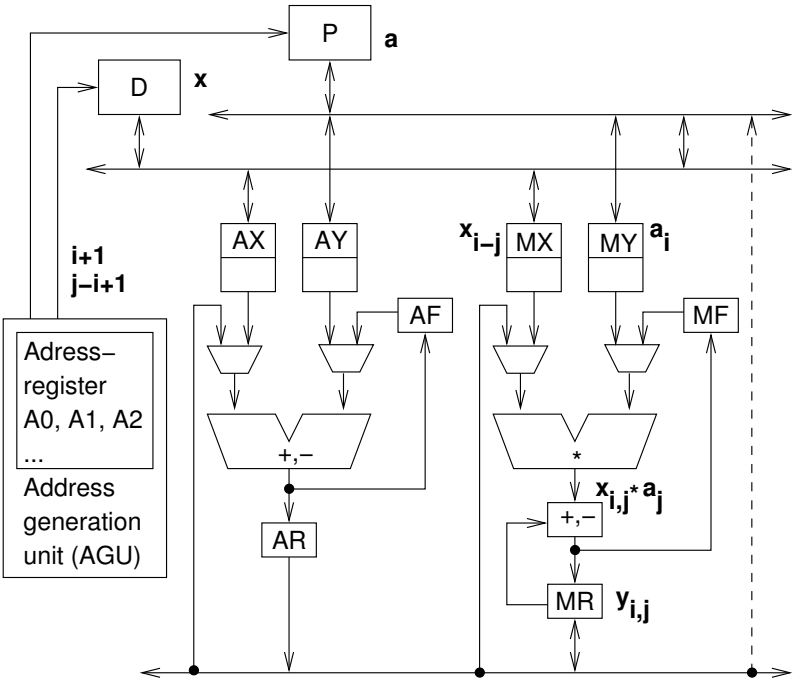


Abb. 3.14. Interne Architektur des ADSP 2100 Prozessors

Der Prozessor hat zwei als D und P bezeichnete Speicher. Eine spezielle Einheit, die Adressen berechnet (*Address Generation Unit* (AGU)), kann Zeiger auf diese beiden Speicher in ihren Registern A0, A1, A2, ... zur Verfügung stellen. Es gibt zusätzlich separate Einheiten für Additionen und Multiplikationen, jeweils mit eigenen Argumentregistern AX, AY, AF, MX, MY und MF. Der Multiplizierer ist mit dem zweiten Addierer verbunden, um eine Folge von Multiplikationen und Additionen schnell berechnen zu können.

Bei diesem Prozessor wird die Aktualisierung einer Partialsumme in einem Takt berechnet. Dazu werden die beiden Speicher so belegt, dass sie die beiden Arrays x und a enthalten, und die Adressregister werden so initialisiert, dass die wichtigen Zeiger auf die Daten leicht von der AGU berechnet werden können. Partialsummen $y_{i,j}$ werden in MR gespeichert.

Die Berechnung, die eine *Pipeline* benutzt, verwendet die Register A1, A2, MX und MY, wie man in der folgenden Implementierung des Filters sehen kann.

```
MR:=0;  A1:=1;  A2:=n-2;  MX:=x[n-1];  MY:=a[0];
for (j=1;  j<=n; j++)
  {MR:=MR + MX * MY;   MX:=x[A2];   MY:=a[A1];
    A1++; A2-- }
```

Eine einzige Instruktion kodiert den Schleifenkörper. Sie besteht aus den folgenden Operationen:

- Lesen der beiden Argumente aus den Argumentregistern MX und MY, multiplizieren der beiden Werte und Addition des Produkts zu Register MR, das die Werte $y_{i,j}$ enthält,
- Laden der nächsten Elemente der Arrays a und x aus den Speichern P und D und Speichern dieser Werte in den Argumentregistern MX und MY,
- Aktualisieren der Zeiger auf die nächsten Argumente, die in den Adressregistern A1 und A2 gespeichert werden,

Auf diese Weise benötigt jede Iteration nur einen einzigen Befehl. Um das zu erreichen, müssen mehrere Operationen parallel ausgeführt werden. Bei gegebenen Rechenkapazitäts-Anforderungen führt diese (eingeschränkte) Form der Parallelität zu relativ niedrigen Taktfrequenzen. Desweiteren erfüllen die Register bei dieser Architektur unterschiedliche Aufgaben. Man nennt sie daher **heterogene** Register. Heterogene Registersätze sind ein typisches Merkmal von DSP-Prozessoren. Um zusätzliche Zyklen für das Überprüfen des Schleifenendes einzusparen, werden in vielen DSP-Prozessoren sogenannte *zero-overhead loop instructions* eingesetzt. Mit solchen Befehlen kann ein einziger Befehl oder eine kleine Anzahl Befehle über eine bestimmte Anzahl von Durchläufen ausgeführt werden. Prozessoren, die nicht für DSP-Anwendungen spezialisiert sind, würden mehrere Instruktionen pro Iteration und damit eine höhere Taktfrequenz benötigen (die evtl. gar nicht möglich ist).

DSP-Prozessoren

Zusätzlich zur Realisierung von Filtern mit Hilfe einer einzigen Instruktion im Schleifenkörper bieten DSP-Prozessoren noch weitere Merkmale, die spezifisch für den DSP-Anwendungsbereich sind:

- **Spezialisierte Adressierungsarten:** In der oben beschriebenen Filter-Anwendung müssen nur die letzten n Werte der Folge x verfügbar sein.

Dazu können Ringpuffer verwendet werden. Diese können leicht mittels Modulo-Adressierung realisiert werden. Bei der Modulo-Adressierung können Adressen inkrementiert und dekrementiert werden, bis das erste oder das letzte Element des Puffers erreicht wird. Weiteres Erhöhen oder Verringern führt dann zu Adressen, die zum anderen Ende des Puffers zeigen.

- **Separate Adressgenerierungseinheiten:** Adressgenerierungseinheiten (*Address Generation Units* (AGUs)) werden üblicherweise direkt an die Adresseingänge des Datenspeichers angeschlossen (s. Abb. 3.15).

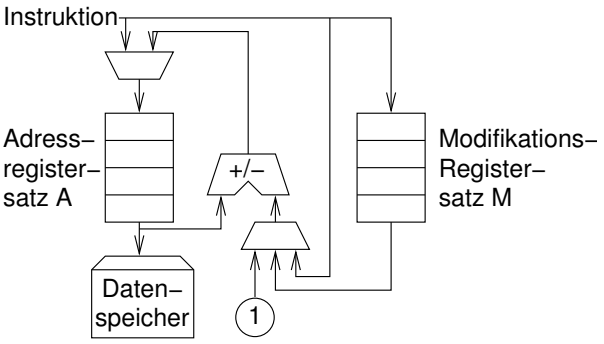


Abb. 3.15. AGU mit speziellen Adressregistern

Adressen, die in Adressregistern vorliegen, können in Register-indirekten Adressierungsmodi verwendet werden, wodurch Maschinenbefehle, Zyklen und Energie eingespart werden. Um die Nützlichkeit dieser Adressregister zu erhöhen, stellen die Befehlssätze häufig Auto-Inkrement- und Auto-Dekrement-Optionen für die meisten Befehle zur Verfügung, welche die Adressregister nutzen.

- **Sättigungsarithmetik:** Sättigungsarithmetik verändert die Behandlung von Über- und Unterläufen. Die Standard-Binär-Arithmetik verwendet ein sogenanntes *wrap-around* bei Über- oder Unterläufen. Dabei werden die einzelnen Bits innerhalb der darstellbaren Stellen des Ergebnisses so berechnet, als gäbe es keinen Über- bzw. Unterlauf. Auftretende Überträge werden einfach ignoriert. Abbildung 3.16 zeigt ein Beispiel, bei dem zwei vorzeichenlose Vier-Bit-Zahlen addiert werden. Dabei wird ein Übertrag erzeugt, der nicht in einem der Standardregister zurückgegeben werden kann. Das Ergebnisregister enthält bei Verwendung von *wrap-around* ein Muster, das nur aus Nullen besteht. Kein Ergebnis könnte weiter vom gewünschten Ergebnis entfernt sein als dieses.

  Bei der Sättigungsarithmetik wird ein Ergebnis erzeugt, das so nahe wie möglich am wahren Ergebnis liegt. **Die Sättigungsarithmetik gibt den größten möglichen Wert zurück, wenn ein Überlauf stattgefun-**

	0111
+	1001
wrap-around Arithmetik	10000
Sättigungsarithmetik	1111

Abb. 3.16. *Wrap-around* gegenüber Sättigungsarithmetik für vorzeichenlose Ganzzahlen

den hat, und den kleinsten möglichen Wert im Falle eines Unterlaufs. Dieser Ansatz ist insbesondere für Video- und Audioanwendungen sinnvoll. Der Benutzer wird im obigen Beispiel wohl kaum den Unterschied zwischen dem wahren Ergebniswert und dem größten darstellbaren Wert erkennen können. Es wäre auch sinnlos, bei einem Überlauf eine Ausnahme zu signalisieren, da es schwierig wäre, eine solche Ausnahme in Echtzeit zu behandeln. Man muss allerdings wissen, ob vorzeichenlose oder vorzeichenbehaftete Zahlen verwendet werden, um den richtigen Ergebniswert zu bestimmen.

- **Festkomma-Arithmetik:** Gleitkomma-Hardware erhöht die Kosten und die Leistungsaufnahme von Prozessoren. Geschätzte 80% der DSP-Prozessoren haben daher keine Gleitkomma-Hardware [Aamodt und Chow, 2000]. Zusätzlich zur Ganzzahl-Verarbeitung bieten viele solcher Prozessoren aber die Verarbeitung von Festkomma-Zahlen an. Festkomma-Datentypen können mit Hilfe eines 3-Tupels $(wl, iwl, sign)$ spezifiziert werden, wobei wl die Gesamtwortlänge ist, iwl die Ganzzahlwortlänge (also die Anzahl der Stellen links des Dezimalkommas) und $sign \in \{s, u\}$ angibt, ob es sich um vorzeichenbehaftete oder vorzeichenlose Zahlen handelt (s. Abb. 3.17). Weiterhin gibt es verschiedene Rundungsmodi (z.B. Abschneiden) und Überlaufmodi (z.B. Sättigungs- und *Wrap-around*-Arithmetik).

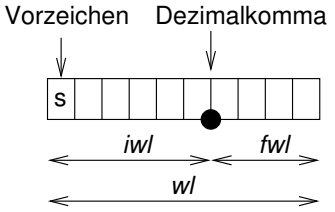


Abb. 3.17. Parameter eines Festkomma-Zahlensystems

Bei Festkommazahlen bleibt die Position des Dezimalkommas nach der Multiplikation erhalten (einige der niederwertigen Bits werden abgeschnitten oder gerundet). Bei Festkomma-Prozessoren wird diese Operation direkt von der Hardware unterstützt.

- **Echtzeit-Fähigkeiten:** Einige der in modernen Prozessoren verwendeten Bestandteile dienen dazu, die durchschnittliche Ausführungszeit von Programmen zu verbessern. In vielen Fällen ist es schwierig bis unmöglich, formal nachzuweisen, dass diese Vorkehrungen auch die längste mögliche Ausführungszeit (die *Worst Case Execution Time* (WCET)) verbessern. In solchen Fällen kann es besser sein, auf den Einsatz dieser Techniken zu verzichten. Beispielsweise ist es schwierig (aber nicht unmöglich, siehe [Absint, 2002]), eine bestimmte Leistungssteigerung durch den Einsatz eines Caches zu garantieren. Aus diesem Grunde verzichten viele eingebettete Prozessoren auf den Einsatz von Caches. Ebenso wird man virtuelle Adressierung und *demand paging* [Hennessy und Patterson, 1995] üblicherweise nicht in eingebetteten Systemen finden.
- **Speicher mit mehreren Bänken oder mehrere Speicher:** Der sinnvolle Einsatz mehrerer Speicher wurde anhand des ADSP 2100-Beispiels bereits gezeigt: Die beiden Speicher D und P erlauben einen gleichzeitigen Zugriff auf beide Argumente. Einige DSP-Prozessoren verfügen über zwei Speicherbänke.
- **Heterogene Registersätze:** Heterogene Registersätze wurden bei der Besprechung der Filter-Anwendung bereits erklärt.
- **Multiply-Accumulate-Befehle:** Diese Befehle führen eine Multiplikation gefolgt von einer Addition aus. Auch dieser Befehl wurde bereits bei der Filter-Anwendung verwendet.

Multimedia-Prozessoren

Die Register und Rechenwerke moderner Prozessorarchitekturen haben häufig eine Breite von 64 Bit. Man kann also zwei 32 Bit Datentypen (*double words*), vier 16 Bit Datentypen (*words*) oder acht 8 Bit Datentypen (*bytes*) in ein einziges Register laden (s. Abb. 3.18).

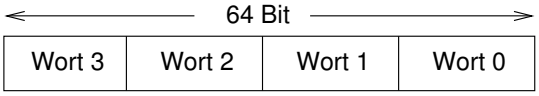


Abb. 3.18. Verwenden eines 64 Bit Registers für gepackte Datentypen

Arithmetische Einheiten können so entworfen werden, dass die Überlauf-Bits an den *double word*, *word* oder *byte*-Grenzen unterdrückt, also nicht weitergeleitet, werden. Multimedia-Befehlssätze nutzen diese Eigenschaft aus, um Operationen auf sogenannten gepackten Datentypen durchzuführen. Solche Befehle werden manchmal SIMD-Befehle genannt (*Single Instruction, Multiple Data*), da eine einzige Instruktion die auszuführende Operation auf mehreren Datenelementen vorgibt [Hennessy und Patterson, 1996]. Bei der Verwen-

dung von 8 Bit langen Datentypen in einem 64 Bit-Register sind Geschwindigkeitssteigerungen bis zu einem Faktor von acht gegenüber nicht-gepackten Datentypen möglich. Multimedia-Befehle können außerdem mit Sättigungsarithmetik kombiniert werden und damit eine effizientere Behandlung von Überläufen erlauben als Standard-Befehle. Die Daten werden üblicherweise im Speicher gepackt abgelegt. Das Entpacken und Packen kann man vermeiden, wenn man arithmetische Operationen direkt auf den gepackten Daten durchführt. Der Geschwindigkeitsvorteil bei der Verwendung von Multimedia-Befehlen kann also durchaus größer sein als der Faktor, der sich nur aus der Anzahl gleichzeitig bearbeiteter Datenelemente in gepackten Datentypen ergibt.

Very Long Instruction Word (VLIW)-Prozessoren

Die Anforderungen an die Rechenleistung eingebetteter Systeme steigen, insbesondere im Multimedia-Bereich, in dem moderne Kodierungs-Techniken und Kryptographie eingesetzt werden, stark an. Die in üblichen Hochleistungsrechnern verwendeten leistungssteigernden Techniken sind für eingebettete Systeme ungeeignet: da beispielsweise bei PCs die Befehlssatz-Kompatibilität ein wichtiger Faktor ist, verwenden die in diesen Systemen eingebauten Prozessoren viele ihrer Ressourcen und einen großen Teil der Energie dazu, parallel ausführbare Befehle in einer Anwendung zu finden. Trotzdem ist ihre Leistungsfähigkeit oft nicht ausreichend. In eingebetteten Systemen kann man ausnutzen, dass eine Befehlssatz-Kompatibilität mit PCs nicht erforderlich ist. Daher können Befehle verwendet werden, welche die parallel ausführbaren Instruktionen explizit angeben. Diese Prozessoren heißen *Explicit Parallelism Instruction Set Computers* (EPICs). Bei EPICs wird die Identifizierung von parallel ausführbaren Instruktionen von der Hardware zum Compiler verschoben[6]. Man kann sowohl Silizium als auch Energie einsparen, wenn mögliche Parallelität nicht mehr zur Laufzeit analysiert werden muss. Als Sonderfall betrachten wir hier den Fall von sehr langen Befehlswörtern (*Very Long Instruction Words* (VLIW)). Bei VLIW-Prozessoren werden mehrere Operationen oder Instruktionen in einem sehr langen Befehlswort (das manchmal **Befehlspaket** genannt wird) kodiert und dann parallel ausgeführt. Jede Operation wird in ein bestimmtes Feld des Befehlspakets geschrieben. Jedes dieser Felder steuert bestimmte Hardware-Einheiten an. In Abb. 3.19 werden vier solcher Felder verwendet, von denen jedes eine der Hardware-Einheiten steuert.

Der Compiler muss für VLIW-Architekturen Befehlspakete erzeugen. Daher muss der Compiler auch Informationen über die zur Verfügung stehenden Hardware-Einheiten haben, um diese optimal auszulasten.

[6] EPICs werden manchmal auch in PCs eingesetzt [Transmeta, 2005, Intel, 2008]. Die Eigenschaften vorhandener Programme schränken die Nützlichkeit dieses Vorgehens allerdings stark ein.

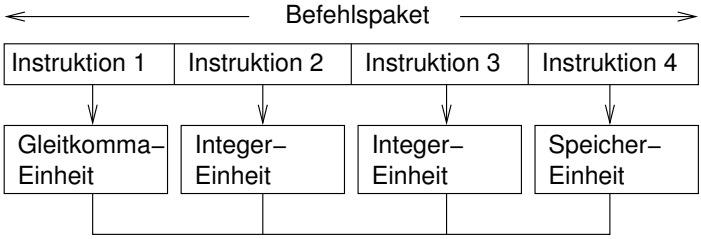


Abb. 3.19. VLIW-Architektur (Beispiel)

Die Felder in den Befehlen sind fest vorgegeben und müssen vorhanden sein, unabhängig davon, ob in einem bestimmten Zyklus tatsächlich ein Befehl auf der entsprechenden Einheit ausgeführt werden soll. Wenn nicht genügend Parallelität vorhanden ist, um alle Einheiten auszulasten, kann die Codedichte von VLIW-Prozessoren daher relativ niedrig sein. Dieses Problem kann durch mehr Flexibilität vermieden werden. Beispielsweise verwendet die Familie der Texas Instruments TMS 320C6xx-Prozessoren eine variable Befehlspaket-Größe von bis zu 256 Bit. In jedem Instruktionsfeld ist ein Bit reserviert, das angibt, ob die im nächsten Feld angegebene Operation parallel ausgeführt werden soll (s. Abb. 3.20). Auf diese Weise werden keine Bits für nicht benutzte Funktionseinheiten verschwendet.

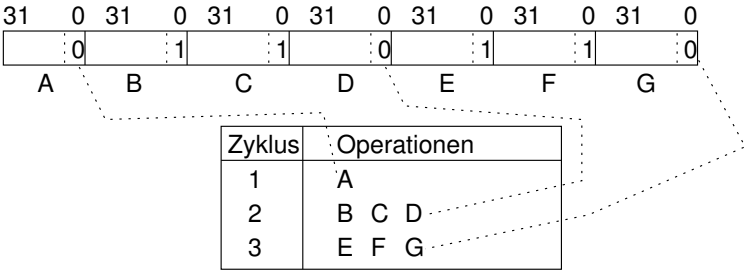


Abb. 3.20. Befehlspakete für TMS 320C6xx

Wegen der variablen Länge ihrer Befehlspakete sind die TMS 320C6xx-Prozessoren keine klassischen VLIW-Prozessoren. Aufgrund der explizit beschriebenen Parallelität sind sie aber auf jeden Fall EPICs.

Partitionierte Registersätze

Die Implementierung der Registersätze für VLIW- und EPIC-Prozessoren ist alles andere als trivial. Wegen der großen Anzahl parallel auszuführender Operationen wird eine große Anzahl paralleler Registerzugriffe benötigt. Man benötigt also viele Schreib- und Leseports für die Register. Allerdings

werden Registersätze mit vielen Ports zunehmend langsamer, größer und verbrauchen mehr Energie, sind also ineffizient. Daher verwenden viele VLIW-/EPIC-Architekturen partitionierte Registersätze. Funktionseinheiten werden dann nur an eine Teilmenge der Register angeschlossen. Als Beispiel zeigt Abb. 3.21 die interne Struktur des TMS 320C6xx-Prozessors. Diese Prozessoren haben zwei Registersätze, die jeweils mit der Hälfte der Funktionseinheiten verbunden sind. Bei jedem Takt ist nur eine einzige Verbindung zwischen einem Registersatz und den Funktionseinheiten, die am anderen Registersatz angeschlossen sind, möglich.

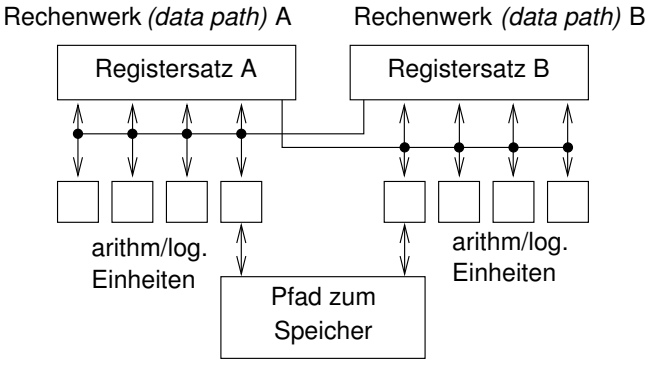


Abb. 3.21. Partitionierte Registersätze beim TMS 320C6xx

Andere Partitionierungen werden von Lapinskii et al. [Lapinskii et al., 2001] vorgestellt.

Viele DSP-Prozessoren sind tatsächlich VLIW-Prozessoren. Als Beispiel betrachten wir den M3-DSP [Fettweis et al., 1998]. Der M3-DSP ist ein VLIW-Prozessor mit (bis zu) 16 parallelen Datenpfaden. Diese Datenpfade sind an einen Gruppenspeicher angeschlossen, der die benötigten Argumente parallel zur Verfügung stellen kann (s. Abb. 3.22).

Bedingte Ausführung

Ein potentielles Problem von VLIW- und EPIC-Architekturen ist ihre möglicherweise große *delay penalty*. Darunter versteht man Prozessorzyklen, die nicht für nützliche Operationen genutzt werden können, weil benötigte Speicherworte nicht schnell genug bereitgestellt werden können (z.B. bei Sprungbefehlen). Üblicherweise werden alle Befehlspakete in einem Fließband (*Pipeline*) [Hennessy und Patterson, 1996] verarbeitet. Jede der Fließbandstufen führt nur einen bestimmten Teil der Operationen der jeweiligen Instruktion aus. Die Tatsache, dass ein Sprungbefehl in einem Befehlspaket enthalten ist, kann nicht direkt in der ersten Fließbandstufe entdeckt werden. Wenn die Ausführung des Sprungbefehls endgültig abgeschlossen ist, wurden bereits

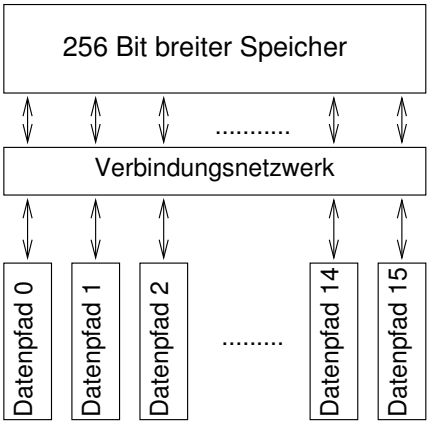


Abb. 3.22. M3-DSP (vereinfacht)

weitere Instruktionen im Fließband bearbeitet (s. Abb. 3.23). Man würde in der Regel erwarten, dass die Bearbeitung dieser Befehle wirkungslos bleibt, wenn die Sprungbedingung erfüllt ist. Allerdings würde dann Rechenleistung verlorengehen.

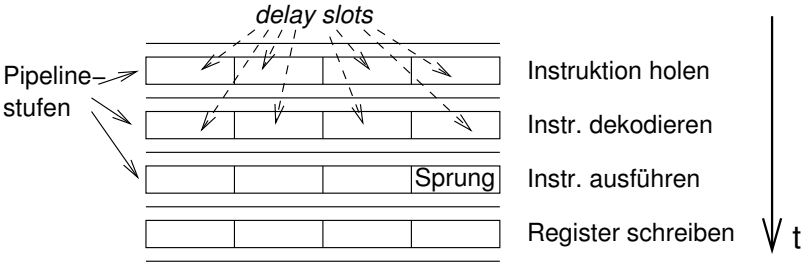


Abb. 3.23. Sprungbefehl und *delay slots*

Es gibt im Wesentlichen zwei Möglichkeiten, mit diesem potentiellen Leistungsverlust umzugehen:

1. Die Befehle werden so ausgeführt, als wäre gar kein Sprungbefehl vorhanden. Dieser Fall heißt **verzögerter Sprung** (*delayed branch*). Diejenigen Befehle des Befehlspaketes, die nach dem Sprung noch ausgeführt werden, heißen ***branch delay slots***. Diese *delay slots* können mit solchen Befehlen gefüllt werden, die ohne *delay slots* eigentlich vor dem Sprung ausgeführt werden würden. Allerdings ist es im Allgemeinen schwierig, alle *delay slots* mit sinnvollen Befehlen zu füllen, und so müssen einige mit *no-operation* (NOP))-Instruktionen aufgefüllt werden. Der Ausdruck

branch delay penalty gibt den Geschwindigkeitsverlust aufgrund dieser NOPs an.

2. Das Fließband wird angehalten, bis die ersten Instruktionen vom Sprungziel des Sprunges geladen worden sind. In diesem Fall gibt es keine *branch delay slots*. Bei dieser Vorgehensweise werden die *branch delay penalty*-Zyklen durch das Anhalten der Pipeline verursacht.

Die *branch delay penalties* können sich sehr deutlich auswirken. Die TMS 320C6xx-Prozessorfamilie hat beispielsweise bis zu 40 *delay slots*. Die Geschwindigkeit kann daher durch die Vermeidung von Sprüngen stark verbessert werden. Um Sprünge, die von if-Befehlen erzeugt werden, zu vermeiden, kann man die **bedingte Ausführung** (*predicated execution*) von Befehlen ausnutzen. Jede bedingte Anweisung verfügt über eine Bedingung, die in wenigen Bits kodiert und zur Laufzeit ausgewertet werden kann. Wenn die Bedingung erfüllt ist, wird die Anweisung ausgeführt. Wenn nicht, wird die Anweisung in ein NOP umgewandelt. Bedingte Ausführung findet man auch in einigen RISC-Maschinen wie etwa den ARM-Prozessoren. Beispielsweise verfügen ARM-Befehle, wie sie auf Seite 113 vorgestellt wurden, über ein 4 Bit breites Bedingungsfeld, in dem Ausdrücke über die *Condition Code* Register kodiert werden können. Die Werte in diesen Registern werden zur Laufzeit überprüft. Sie bestimmen, ob ein Befehl ausgeführt oder in ein NOP umgewandelt wird.

Die bedingte Ausführung kann verwendet werden, um kleine if-Blöcke effizient zu implementieren: die Bedingung wird in den *Condition Code* Registern gespeichert. Die Anweisungen des *if*-Blocks werden mit Hilfe bedingter Befehle implementiert, die von dieser Bedingung abhängen. So kann der *if*-Block parallel mit anderen Operationen ausgewertet und ausgeführt werden, ohne eine *delay penalty* zu verursachen.

Mikrocontroller

Viele in eingebetteten Systemen eingesetzte Prozessoren sind Mikrocontroller. Mikrocontroller sind normalerweise nicht sehr komplex und relativ leicht in ein System zu integrieren. Aufgrund der Relevanz von Mikrocontrollern beim Entwurf von Steuerungssystemen stellen wir hier einen der am häufigsten verwendeten Prozessoren vor: den Intel 8051. Dieser Prozessor hat die folgenden Eigenschaften:

- Eine 8 Bit CPU, optimiert für Steuerungs-Anwendungen,
- viele Operationen auf Booleschen Datentypen,
- Programmspeicher bis zu 64 kB,
- separater Datenspeicher mit bis zu 64 kB,

- 4 kB Programmspeicher direkt auf dem Chip, 128 Byte Datenspeicher direkt auf dem Chip,
- 32 Ein-/Ausgabeleitungen, die individuell adressiert werden können,
- 2 Zähler auf dem Chip,
- serielle asynchrone Schnittstelle (*UART*) auf dem Chip,
- Takterzeugung auf dem Chip,
- viele Varianten sind kommerziell erhältlich.

Alle diese Eigenschaften sind recht charakteristisch für Mikrocontroller.

3.4.4 Rekonfigurierbare Logik

Für viele Anwendungsfälle ist ein vollständiger Hardware-Entwurf und der Einsatz von anwendungsspezifischen integrierten Schaltkreisen (*ASICs*) zu teuer. Andererseits sind Lösungen, die auf Software basieren, häufig zu langsam oder verbrauchen zu viel Energie. Rekonfigurierbare Logik kann dann eine Lösung darstellen, wenn die verwendeten Algorithmen effizient in speziell angepasster Hardware implementiert werden können. Diese Lösung kann beinahe so schnell sein wie der Einsatz von ASICs. Im Gegensatz zu diesen kann die ausgeführte Funktion allerdings durch das Ändern von Konfigurationsdaten verändert werden. Wegen dieser Eigenschaften findet die rekonfigurierbare Logik in den folgenden Gebieten Anwendung:

- **Prototypen:** Moderne ASICs können sehr komplex sein, und der Entwurfsprozess ist langwierig und teuer. Daher möchte man häufig einen Prototypen zur Verfügung haben, der für Versuche mit dem System verwendet werden kann und der sich „fast“ wie das endgültige System verhält. Der Prototyp darf teurer und größer als das endgültige System sein. Auch darf sein Stromverbrauch durchaus noch höher sein, und Zeitbedingungen können entschärft werden. Es ist auch möglich, dass der Prototyp nur die wichtigsten Funktionen implementiert. Solch ein Prototyp kann dazu verwendet werden, das grundlegende Verhalten des zu entwickelnden Systems zu überprüfen.
- **Kleine Stückzahlen:** Wenn das zu erwartende Marktvolumen zu klein ist, um die hohen Entwicklungskosten für ASICs zu amortisieren, kann rekonfigurierbare Logik der richtige Ansatz für Anwendungen sein, die sich nicht allein in Software implementieren lassen.

Rekonfigurierbare Hardware enthält typischerweise einen Speicher (*RAM*), um die Konfigurationsinformation während des normalen Betriebs der Hardware abzuspeichern. Diese Speicher sind aber in der Regel **flüchtig** (sie behalten die gespeicherte Information also nur, solange die Versorgungsspannung

anliegt). Darum muss die Konfiguration beim Starten des Systems zuerst in diesen Speicher kopiert werden. **Persistente** Speichertechnologien wie etwa *ROM*- und *Flash*-Speicher können die Konfigurationsdaten für diesen Zweck vorhalten.

***Field Programmable Gate Arrays* (FPGAs)** sind die am häufigsten eingesetzte Form rekonfigurierbarer Logik. FPGAs können nach der Herstellung beim Kunden programmiert werden. Sie bestehen aus Feldern von Prozessor-Elementen. Als Beispiel zeigt Abb. 3.24 die Struktur eines Xilinx Virtex-II FPGAs (siehe http://www.xilinx.com).

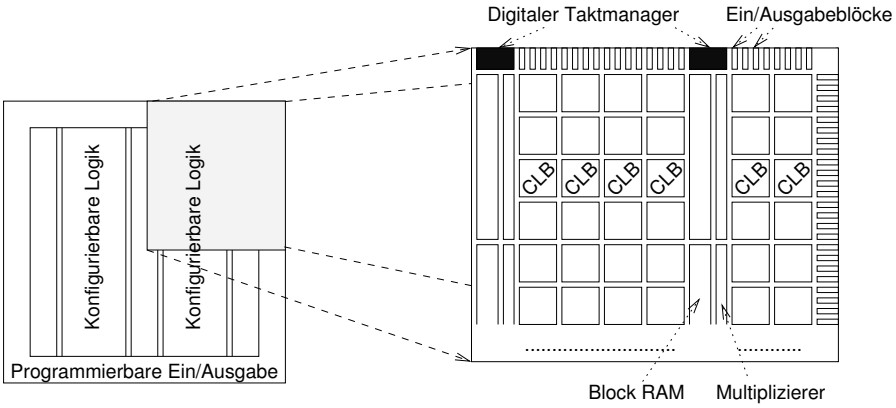


Abb. 3.24. Struktur eines Virtex II FPGAs

Die FPGAs der Virtex II-Baureihe enthalten **konfigurierbare Logikblöcke** (*Configurable Logic Blocks* (CLBs)). Diese können mit Hilfe einer programmierbaren Verbindungsstruktur miteinander verschaltet werden. Ein FPGA kann auch mehr als Tausend Ein-/Ausgabeleitungen haben und über spezielle taktverarbeitende Blöcke verfügen. Zusätzlich gibt es 18×18 Bit Multiplizierer und Speicher (*Block-RAM*). Jeder CLB besteht aus 4 sogenannten *Slices* (s. Abb. 3.25).

Jede dieser *Slices* enthält zwei 16 Bit Speicher F und G. Diese Speicher können als Wertetabellen (*Look-Up-Tables* (LUTs)) verwendet werden, um alle 2^{16} Booleschen Funktionen von vier Variablen darzustellen. Unter Verwendung von Multiplexern (MUXF5, MUXFx) können einige dieser Speicher auch miteinander verbunden werden, so dass Tabellen für Funktionen mit bis zu acht Variablen möglich sind. Die Speicher können aber auch als normaler RAM-Speicher oder als Schieberegister (*SRLs*) verwendet werden. Jede *Slice* verfügt auch über zwei Ausgabe-Register und Spezial-Logik (ORCY, CY, usw.) für Additionen (s. Abb. 3.26).

Konfigurationsdaten bestimmen die Einstellung der Multiplexer in den *Slices*, das Takten von Registern und Speicher, den Inhalt der Speicherkomponenten

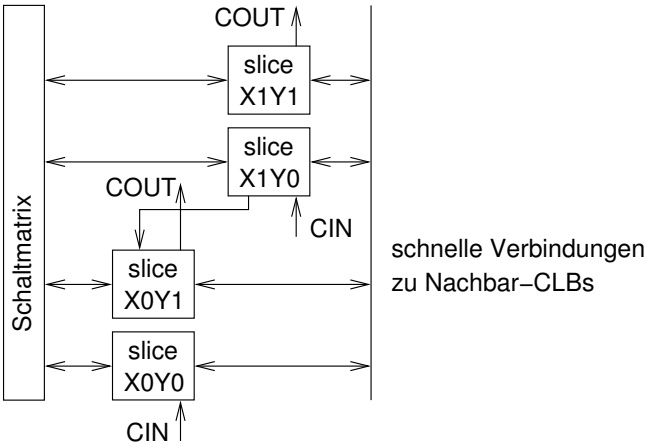


Abb. 3.25. Virtex II CLB

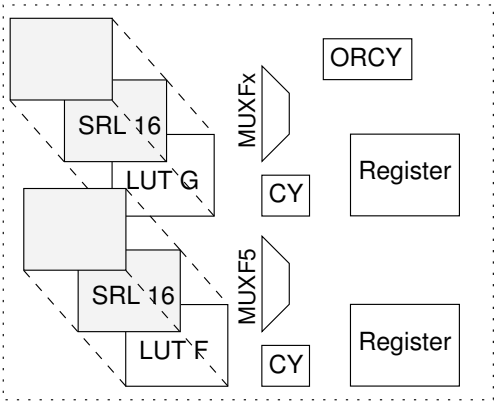


Abb. 3.26. Virtex II Slice (vereinfacht)

und die Verbindungen zwischen den CLBs. Typischerweise werden diese Konfigurationsinformationen aus einer *High-Level* Beschreibung der Funktionalität der Hardware generiert, beispielweise aus einem VHDL-Modell. Idealerweise kann die gleiche Beschreibung auch zur automatischen Erzeugung von ASICs verwendet werden. In der Praxis sind hierzu jedoch manuelle Eingriffe nötig.

2006 waren in der Xilinx Virtex II Pro-Baureihe FPGAs mit bis zu 44.096 *Slices* verfübar.

Die Integration von rekonfigurierbarer Logik mit anderen Prozessoren und Software wird bei der Virtex II Pro-Serie von Xilinx vereinfacht. Diese FPGAs enthalten bis zu vier Power-PC-Prozessoren und schnelle Ein-/ Ausgabeblöcke.

3.5 Speicher

Daten, Programme und FPGA-Konfigurationen müssen effizient in einem Speicher abgelegt werden können. Effizient bedeutet hier Laufzeit-, Codegrößen- und Energie-effizient. Codegrößeneffizienz setzt einen guten Compiler voraus und kann mit Hilfe von Techniken zur Codekompression verbessert werden (s. Seite 113). Speicherhierarchien können ausgenutzt werden, um eine gute Laufzeit- und Energieeffizienz zu erzielen. Dies basiert auf der Tatsache, dass große Speicher mehr Energie pro Zugriff benötigen und auch langsamer sind als kleine Speicher.

Abbildung 3.27 zeigt die Speicher-Zykluszeit und die Leistung als Funktion der Größe des Speichers [Rixner et al., 2000]. Das gleiche Verhalten kann auch bei größeren Speichern beobachtet werden.

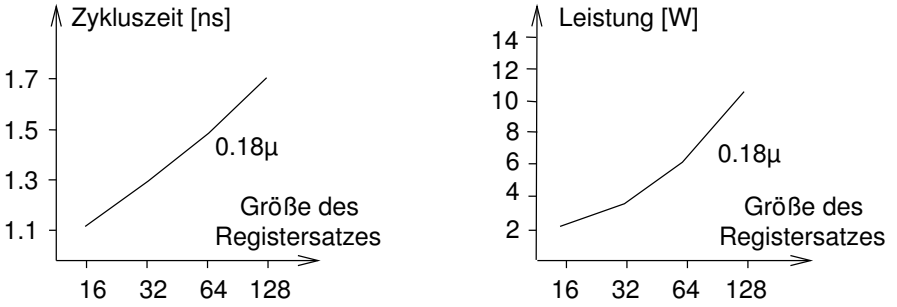


Abb. 3.27. Zykluszeit und Leistung als Funktion der Speichergröße

Es ist beobachtet worden, dass der Geschwindigkeitsunterschied zwischen Prozessoren und Speichern angestiegen ist (s. Abb. 3.28), und es ist zu vermuten, dass sich dieser Trend fortsetzt.

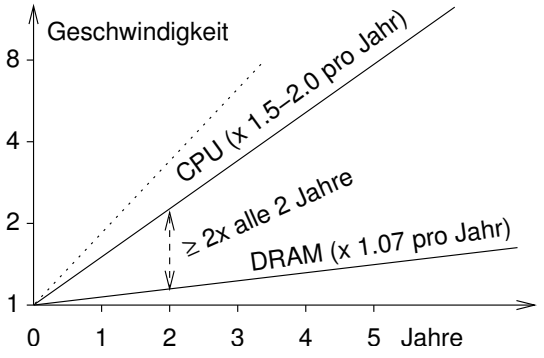


Abb. 3.28. Zunehmende Differenz zwischen Prozesor- und Speichergeschwindigkeit

Während die Geschwindigkeit von Speichern sich um einen Faktor von ca. 1,07 pro Jahr verbessert hat, wurden Prozessoren bisher mit einem Faktor zwischen 1,5 und 2 pro Jahr schneller [Machanik, 2002]. Das bedeutet, dass man eine immer größer werdende die Lücke zwischen Prozessor- und Speichergeschwindigkeiten erwarten kann.

Aus diesem Grund ist es wichtig, kleine und schnelle Speicher als Puffer zwischen dem Hauptspeicher und dem Prozessor zu verwenden. Im Gegensatz zu PC-Systemen muss die Architektur dieser kleinen Speicher ein vorhersagbares Echtzeitverhalten garantieren. Eine Kombination aus kleinen Speichern (die häufig benötigte Daten und Befehle enthalten) mit einem großen Speicher (der die restlichen Speicherobjekte aufnimmt) ist im Allgemeinen energieeffizienter als ein einziger großer Speicher.

Caches wurden ursprünglich eingeführt, um eine gute Laufzeiteffizienz zu ermöglichen. Im Zusammenhang mit dem rechten Teil der Abb. 3.27 wird allerdings klar, dass Caches auch die Energieeffizienz von Speichersystemen verbessern. Zugriffe auf den Cache sind Zugriffe auf kleine Speicher und benötigen daher weniger Energie pro Zugriff als Hauptspeicherzugriffe. Bei Caches muss die Hardware aber immer überprüfen, ob eine gültige Kopie der gewünschten Information im Cache vorliegt oder nicht. Bei dieser Prüfung werden die sogenannten *Tags* des Caches abgefragt, die eine Teilmenge der relevanten Adressbits enthalten [Hennessy und Patterson, 1996]. Das Lesen dieser *Tags* verbraucht zusätzliche Energie. Die Vorhersagbarkeit des Echtzeitverhaltens von Caches ist in der Regel sehr schlecht.

Als Alternative können kleine Speicher in den Adressbereich des Systems eingeblendet werden (s. Abb. 3.29).

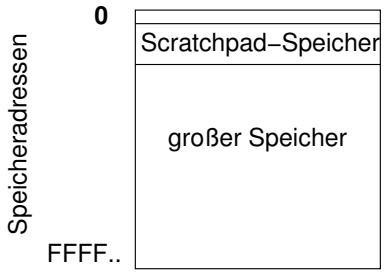


Abb. 3.29. Speicherbelegung mit einem Scratchpad-Speicher

Solche Speicher werden ***Scratchpad*-Speicher** (*Scratch Pad Memories* (SPMs)) genannt. Häufig verwendete Variablen und Instruktionen sollten in den Adressbereich des SPM gelegt werden. In diesem Fall sind keine Gültigkeits-Überprüfungen in Hardware zur Laufzeit notwendig, was den Energieverbrauch reduziert. Abbildung 3.30 zeigt einen Vergleich zwischen der Energie pro Zugriff auf ein *Scratchpad* und einen Cache.

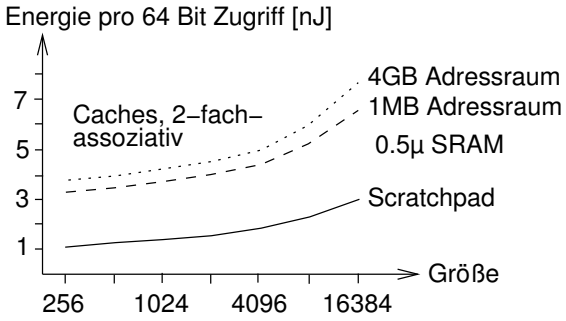


Abb. 3.30. Energieverbrauch pro Scratchpad- und Cache-Zugriff

Für einen zweifach assoziativen Cache unterscheiden sich die Werte in etwa um einen Faktor drei. Die Energiewerte dieses Beispiels wurden mit Hilfe des *CACTI*-Programms berechnet [Wilton und Jouppi, 1996], das eigentlich zur Abschätzung der Energie von Caches dient. Aus den Energiewerten für die Speicherkomponenten eines Caches können auch Energiewerte für *Scratchpad*-Speicher bestimmt werden.

Scratchpad-Speicher können die Vorhersagbarkeit des Zeitverhaltens eines Speicherzugriffs verbessern, wenn der Compiler dafür sorgt, dass die häufig benötigten Variablen im Scratchpad gehalten werden (s. Seite 199).

3.6 Ausgabe

Ausgabegeräte eingebetteter Systeme beinhalten

- **Anzeigegeräte:** Die Technologie von Anzeigegeräten (*display technology*) ist sehr wichtig, deshalb gibt es darüber sehr viele Informationen [Society for Display Technology, 2003]. Große Forschungs- und Entwicklungsanstrengungen haben zu neuen Anzeige-Technologien wie etwa organischen Anzeigen [Gelsen, 2003] geführt. Organische Anzeigen (*organic displays*) können mit einer sehr hohen Dichte hergestellt werden. Im Gegensatz zu *Liquid Crystal Displays* (LCDs) benötigen sie keine Hintergrundbeleuchtung und polarisierende Filter, da sie selber Licht aussenden können. Aufgrund dieser Eigenschaften werden grundlegende Veränderungen des Anzeigegeräte-Marktes erwartet.
- **Elektromechanische Geräte:** Diese Geräte beeinflussen ihre Umgebung durch Motoren und andere elektromechanische Bauteile.

Es werden sowohl analoge als auch digitale Ausgabegeräte verwendet. Im Falle analoger Ausgabegeräte muss die digitale Information zuerst mittels eines Digital-Analog-Wandlers (D/A-Wandler) konvertiert werden.

3.6.1 D/A-Wandler

D/A-Wandler sind nicht sehr kompliziert. Abbildung 3.31 zeigt den Schaltplan eines einfachen D/A-Wandlers.

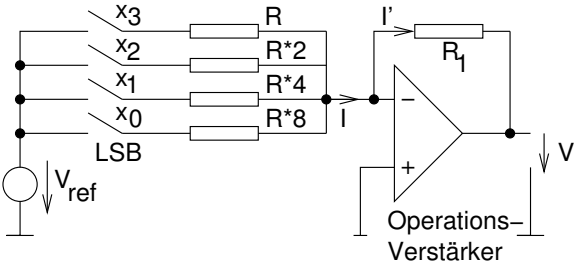


Abb. 3.31. D/A-Wandler

Der Operationsverstärker in Abb. 3.31 verstärkt die Spannungsdifferenz zwischen seinen beiden Eingängen um einen sehr großen Faktor (mehrere Zehnerpotenzen). Über den Widerstand R_1 wird die entstehende Spannung wieder in den Eingang '−' eingespeist. Wenn zwischen den Eingängen eine kleine Spannungsdifferenz anliegt, wird sie invertiert, verstärkt und wieder an den Eingang gelegt, wodurch die Eingangsspannungdifferenz reduziert wird. Aufgrund des großen Verstärkungsfaktors wird die Spannung zwischen den Eingängen praktisch auf Null reduziert. Da der Eingang '+' mit Masse verbunden ist, ist die Spannung zwischen Eingang '−' und Masse praktisch gleich Null. Man sagt, der '−'-Eingang bilde eine virtuelle Masse. Eine virtuelle Masse kann man dabei als einen Leitungsknoten interpretieren, dessen Spannung in Bezug auf die Masseleitung praktisch gleich Null ist, der mit dieser Leitung aber nicht direkt verbunden ist.

Die Idee des D/A-Wandlers besteht nun darin, einen Strom zu erzeugen, der proportional zum Wert ist, der von dem Bitvektor x dargestellt wird. Dieser Strom wird dann in eine äquivalente Spannung umgewandelt.

Nach den Kirchhoffschen Gesetzen ist der Strom I hier die Summe aus den Strömen durch die Widerstände. Der Strom durch einen der Widerstände ist gleich Null, wenn das entsprechende Vektorelement von x gleich '0' ist. Wenn es dagegen '1' ist, entspricht der Strom wegen der entsprechend gewählten Widerstandswerte der Gewichtung dieses Bits.

$$\begin{aligned} I &= x_3 * \frac{V_{ref}}{R} + x_2 * \frac{V_{ref}}{2*R} + x_1 * \frac{V_{ref}}{4*R} + x_0 * \frac{V_{ref}}{8*R} \\ &= \frac{V_{ref}}{R} * \sum_{i=0}^{3} x_i * 2^{i-3} \end{aligned} \tag{3.7}$$

Nach Kirchhoff und wegen der virtuellen Masse am Eingang '−' gilt auch: $V + R_1 * I' = 0$.

Der Strom, der in die Eingänge des Operationsverstärkers fließt, ist praktisch gleich Null. Somit sind die beiden Ströme I und I' identisch: $I = I'$. Daher:

$$V + R_1 * I = 0 \tag{3.8}$$

Aus den Gleichungen 3.7 und 3.8 ergibt sich:

$$-V = V_{ref} * \frac{R_1}{R} * \sum_{i=0}^{3} x_i * 2^{i-3} = V_{ref} * \frac{R_1}{8 * R} * nat(x) \tag{3.9}$$

nat ist die natürliche Zahl, die durch den Bitvektor x dargestellt wird. Offensichtlich ist die Spannung am Ausgang proportional zur durch x dargestellten Zahl. Positive Ausgangsspannungen und Bitvektoren, die Zahlen im Zweierkomplement darstellen, erfordern kleine Erweiterungen des D/A-Wandlers.

Es stellt sich die Frage, ob es möglich ist, die ursprünglichen analogen Werte von den Sensor-Ausgängen an den Ausgängen der D/A-Wandler zu rekonstruieren, wenn der Prozessor in der Hardware-Schleife (s. Abb. 3.2) die Werte der A/D-Wandler unverändert an die D/A-Wandler weiterreicht. Nach dem Theorem von Nyquist (s. [Oppenheim et al., 1999]) ist dies in der Tat möglich, vorausgesetzt, die Taktfrequenz der *Sample-and-Hold*-Schaltung ist wenigstens doppelt so groß wie die höchste Frequenz, die in den Eingabespannungen vorkommt. Das gilt allerdings nur, wenn man eine beliebig hohe Genauigkeit der Digitalwerte zugrundelegt. Die in der Praxis begrenzte Genauigkeit sorgt für Abweichungen (*quantization noise*) [Oppenheim et al., 1999], die man nicht auf Null reduzieren kann.

3.6.2 Aktuatoren

Es gibt eine Vielzahl von Aktuatoren [Elsevier B.V., 2003a] - von riesigen Aktuatoren, die in der Lage sind, Gewichte von mehreren Tonnen zu bewegen, bis hin zu winzigen Aktuatoren mit einer Größe im μm-Bereich, wie der in Abb. 3.32 gezeigte.

Es ist unmöglich, einen Überblick über alle verfügbaren Aktuatoren zu geben. Als Beispiel nennen wir hier eine bestimmte Art von Aktuatoren, die zukünftig an Bedeutung gewinnen werden: die Mikrosystem-Technologie erlaubt die Herstellung von winzigen Aktuatoren, die beispielsweise in den menschlichen Körper eingepflanzt werden können.

Mit Hilfe solcher Mini-Aktuatoren kann die Menge von Medikamenten, die in den Körper eingebracht werden, genau an den aktuellen Bedarf angepasst

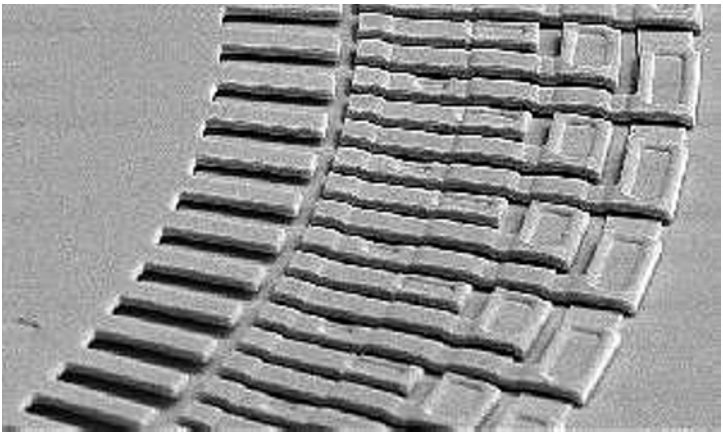


Abb. 3.32. Aktuator-Motor basierend auf Mikrosystemtechnologie (Teilansicht; mit freundlicher Genehmigung von E. Obermeier, MAT, TU Berlin), ©TU Berlin

werden. Damit ist eine viel bessere Medikamentenversorgung möglich als mit herkömmlichen Injektionen. Abbildung 3.32 zeigt einen Miniatur-Motor, der mit Mikrosystemtechnologie hergestellt wurde. Seine Größe bewegt sich im μm-Bereich. Das drehbare Teil in der Mitte wird durch elektrostatische Kräfte gesteuert.

4

Eingebettete Betriebssysteme, *Middleware* und Scheduling

Nicht alle Bestandteile eines eingebetteten Systems müssen von Grund auf neu entworfen werden. Es gibt Standardkomponenten, die wiederverwendet werden können. Diese Komponenten enthalten das Wissen früherer Entwicklungen und stellen sogenanntes geistiges Eigentum (*Intellectual Property* (IP)) dar. Die Wiederverwendung von IP ist eine der Haupttechniken, um die zunehmende Komplexität aktueller Systeme in den Griff zu bekommen. Die Wiederverwendung verfügbarer Softwarekomponenten ist die Basis der Plattform-basierten Entwurfsmethodik, die ab Seite 170 kurz vorgestellt wird.

Standard-Softwarekomponenten, die wiederverwendet werden können, beinhalten etwa eingebettete Betriebssysteme (*Operating Systems* (OS)), Echtzeit-Datenbanken und andere Arten von *Middleware*. Dieser letzte Begriff beschreibt Software, die eine Zwischenschicht zwischen dem Betriebssystem und der Anwendungssoftware zur Verfügung stellt, etwa Kommunikations-Bibliotheken. Aufrufe an solche Standard-Softwarekomponenten müssen unter Umständen bereits bei der Spezifikation eines Systems berücksichtigt werden. Daher sind Informationen über die verwendeten Schnittstellen (*Application Programming Interface* (API)) bereits für die Erstellung einer ausführbaren Spezifikation notwendig.

Desweiteren gibt es einige Standardtechniken zur Ablaufplanung (*scheduling*), die der Entwickler kennen und berücksichtigen sollte. Bestimmte Scheduling-Techniken werden möglicherweise von einem bestimmten Betriebssystem nicht unterstützt. Auch diese Einschränkungen müssen in Betracht gezogen werden.

Gemäß dem vereinfachten Informationsfluss-Diagramm beschreiben wir in diesem Kapitel eingebettete Betriebssysteme, *Middleware* und Scheduling (s. auch Abb. 4.1).

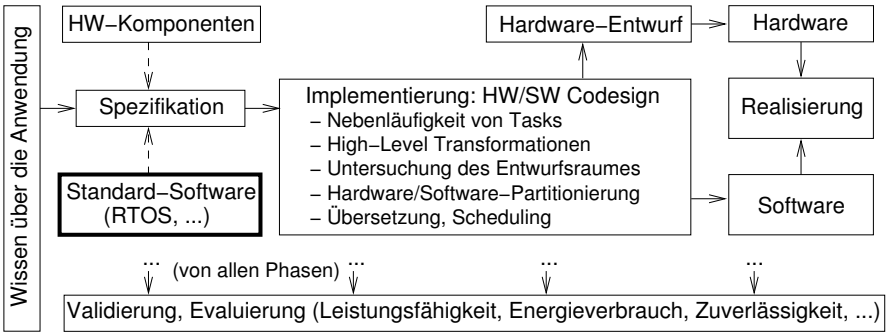


Abb. 4.1. Vereinfachter Informationsfluss beim Entwurf

4.1 Schranken von Ausführungszeiten

Die Ablaufplanung oder das *Scheduling* von Tasks erfordert Wissen über die Ausführungsdauer von Tasks, insbesondere wenn das Einhalten von Zeitbedingungen garantiert werden muss, wie etwa in Echtzeitsystemen. Die längste mögliche Ausführungszeit (*Worst Case Execution Time* (WCET)) ist die Ausgangsbasis der meisten Scheduling-Algorithmen. Die tatsächlich mögliche maximale Ausführungszeit ist dabei in vielen Fällen schwierig zu bestimmen. Man arbeitet daher statt mit möglichen Ausführungszeiten häufig mit sicheren oberen Schranken für diese Zeiten. Leider sind die Bezeichnungen an dieser Stelle in der Literatur uneinheitlich:

1. In der ersten Variante bezeichnet der Begriff WCET obere Schranken, obwohl der Begriff *bound* nicht explizit vorkommt. Tatsächlich mögliche größtmögliche Ausführungszeiten werden bei dieser Variante als **tatsächliche WCET** (*actual WCET*) bezeichnet (s. Abb. 4.2).

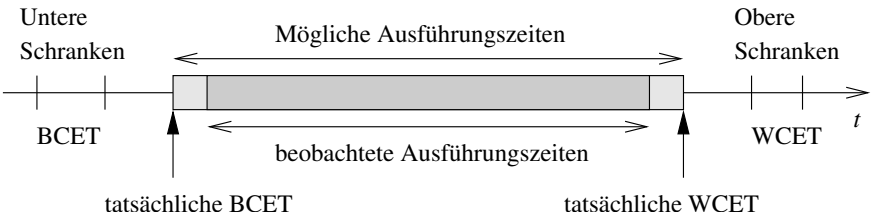


Abb. 4.2. Mögliche Verwendung des WCET-Begriffs

Entsprechendes gilt für die kleinstmögliche Ausführungszeit (*Best Case Execution Time* (BCET)). Auch hier müssen wir zwischen den Schranken und den tatsächlich möglichen Zeiten unterscheiden. Schließlich können

wir noch berücksichtigen, dass sich gemessene bzw. beobachtete Ausführungszeiten innerhalb des Intervalls der möglichen Ausführungszeiten bewegen.

Diese Variante der Benennung schien bis vor Kurzem die übliche zu sein. Sie wird daher auch im englischen Original des Buches benutzt. Allerdings ist die Bezeichnung aufgrund des fehlenden Zusatzes *bound* potenziell irreführend.

2. In der zweiten Variante bezeichnet WCET die tatsächliche größtmögliche Ausführungszeit von Programmen. Sichere obere Schranken für die Ausführungszeit werden bei dieser Variante als **geschätzte WCETs** (*estimated WCETs*) bezeichnet. Es muss dann vielfach explizit hervorgehoben werden, dass es sich trotz der Verwendung des Begriffs „geschätzte“ um sichere Schranken handelt. Abb. 4.3 zeigt diese Verwendung der Begriffe. Diese Verwendung der Begriffe scheint sich derzeit stärker durchzusetzen.

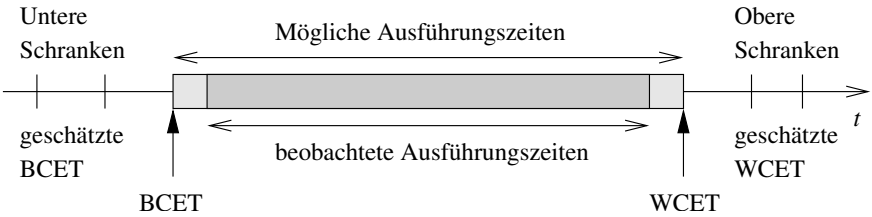


Abb. 4.3. Alternative Verwendung des WCET-Begriffs

In dieser Übersetzung schließen wir uns der neueren, zweiten Variante an. Sofern die Unterscheidung zwischen tatsächlich möglichen Ausführungszeiten und Schranken wichtig ist, werden wir durch Zusätze aber immer deutlich machen, welche Interpretation gemeint ist.

Einige Angaben zur Berechnung von Laufzeitschranken werden in diesem Buch ab Seite 223 gemacht.

4.2 Scheduling in Echtzeitsystemen

Wie oben bereits angedeutet ist die Ablaufplanung (*Scheduling*) eines der Hauptprobleme bei der Implementierung von Echtzeitsystemen. Scheduling-Algorithmen können beim Entwurf eines Echtzeitsystems an verschiedenen Stellen benötigt werden. Sehr grobe Schätzungen können bereits beim Erstellen der Spezifikation notwendig sein. Während der Hardware/Software-Partitionierung sind genauere Vorhersagen über Schedules nötig. Nach der

Übersetzung der Software sind noch genauere Informationen über die Ausführungszeiten von Tasks verfügbar, womit das Scheduling präzisiert werden kann. Schließlich ist es notwendig, zur Laufzeit zu entscheiden, welche Task als nächstes ausgeführt wird. Scheduling ist auch mit der Leistungsbewertung verbunden, wie in Abb. 4.1 unten angedeutet. Wie die Leistungsbewertung kann auch das Scheduling nicht auf einen einzigen Schritt beschränkt werden. Wir behandeln das Scheduling in diesem Kapitel, weil es eng mit Echtzeitbetriebssystemen (*Real-Time Operating Systems* (RTOS)) zusammenhängt. Trotzdem sind einige der Scheduling-Techniken vom verwendeten RTOS vollkommen unabhängig. Wenn das Scheduling zur Entwurfszeit durchgeführt wird, kann sich das RTOS-Scheduling auf einfache *Look-Up*-Tabellen beschränken, die beschreiben, welche Task wann ausgeführt wird.

4.2.1 Klassifikation von Scheduling-Algorithmen

Scheduling-Algorithmen können nach verschiedenen Kriterien unterschieden werden. Abb. 4.4 zeigt eine mögliche Klassifizierung von Algorithmen (ähnliche Vorschläge werden in den entsprechenden Büchern zum Thema vorgestellt [Balarin et al., 1998], [Kwok und Ahmad, 1999], [Stankovic et al., 1998], [Liu, 2000], [Buttazzo, 2002]).

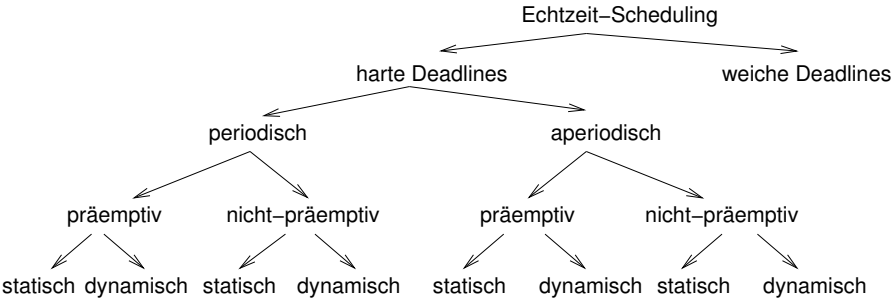


Abb. 4.4. Klassen von Scheduling-Algorithmen

- **Weiche und harte Zeitbedingungen:** Das Scheduling für weiche Zeitbedingungen basiert häufig auf Erweiterungen von Standard-Betriebssystemen. Beispielsweise können Task- und Betriebssystemaufruf-Prioritäten für ein System mit weichen Zeitbedingungen ausreichend sein. Solche Systeme werden wir in diesem Buch allerdings nicht weiter beschreiben. Für harte Echtzeitanforderungen ist deutlich mehr Aufwand und eine detailliertere Analyse notwendig. Für solche Systeme kann man dynamische und statische Scheduler einsetzen.

- **Scheduling für periodische und aperiodische Tasks**: Im Folgenden werden wir zwischen periodischen, aperiodischen und sporadischen Tasks unterscheiden.

  Definition: Tasks, die alle p Zeiteinheiten ausgeführt werden müssen, heißen **periodische Tasks**, und p heißt ihre **Periode**. Jede Ausführung einer periodischen Task heißt **Job**.

  Definition: Tasks, die nicht periodisch sind, heißen **aperiodisch**.

  Definition: Aperiodische Tasks, die den Prozessor zu unvorhersehbaren Zeiten anfordern, heißen **sporadisch**, wenn eine minimale Zeit zwischen den Zeitpunkten vergeht, zu denen die Tasks den Prozessor anfordern.

- **Präemptives und nicht-präemptives Scheduling:** Nicht-präemptive Scheduler basieren auf der Annahme, dass Tasks solange ausgeführt werden, bis ihre Ausführung abgeschlossen ist. Folglich kann die Reaktion auf externe Ereignisse recht spät erfolgen, wenn einige Tasks eine lange Ausführungszeit haben. Präemptive Scheduler, bei denen die Ausführung einer Task vom Scheduler unterbrochen werden kann, müssen verwendet werden, wenn einige Tasks lange Ausführungszeiten haben oder wenn die Reaktionszeit auf externe Ereignisse kurz sein muss.

- **Statisches und dynamisches Scheduling:** Dynamische Scheduler treffen ihre Entscheidungen zur Laufzeit. Sie sind flexibel, erfordern aber zusätzlichen Aufwand zur Laufzeit. Man nennt sie daher auch *on line*-Scheduler. Außerdem sind dynamischen Schedulern normalerweise keine globalen Kontexte bekannt, wie etwa Ressourcen-Anforderungen oder Abhängigkeiten zwischen Tasks. Bei eingebetteten Systemen sind solche globalen Kontextinformationen üblicherweise zum Entwurfszeitpunkt bekannt. Folglich sollte diese Information auch genutzt werden.

  Statische Scheduler treffen Entscheidungen zur Entwurfszeit, man spricht daher auch von *off line*-Scheduling. Sie planen die Startzeiten der Tasks und erzeugen entsprechende Tabellen, in denen diese Startzeiten vermerkt sind. Die Tabellen werden an einen einfachen *Dispatcher* weitergeleitet, der keine Entscheidungen trifft, sondern lediglich Tasks zu den in den Tabellen angegebenen Zeiten startet. Der *Dispatcher* kann über einen *Timer* gesteuert werden, der ihn zum Analysieren der Tabelle auffordert. Systeme, die vollkommen von einem solchen *Timer* gesteuert werden, heißen **vollkommen zeitgesteuert** (*entirely Time Triggered* (TT systems)). Solche Systeme werden im Buch von Kopetz [Kopetz, 1997] ausführlich erklärt:

  „In einem vollkommen zeitgesteuerten System wird die zeitliche Steuerung aller Tasks **a priori** *von Programmen vorgenommen, die nicht erst zur Laufzeit aktiv werden. Die zeitliche Steuerungsdatenstruktur wird in einer sogenannten* **Task-Beschreibungs-Liste** (*Task-Descriptor List* (TDL)*) beschrieben, die das zyklische Schedule für alle Aktivitäten des betreffenden Knotens enthält (Abb. 4.5). Dieses Schedule berücksichtigt die benötigten*

Abhängigkeits- und Ausschluss-Relationen zwischen den Tasks, so dass eine explizite Koordinierung der Tasks durch das Betriebssystem zur Laufzeit nicht notwendig ist.“

Abb. 4.5 enthält Task-Start-, Task-Ende- und Nachrichten (sende)-Aktivitäten.

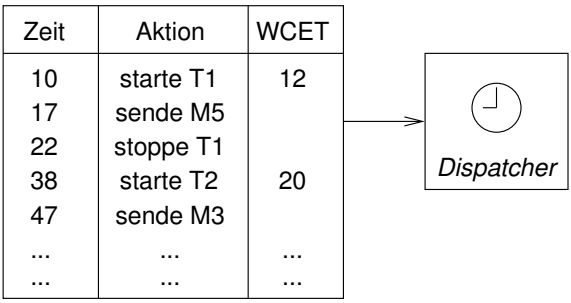


Abb. 4.5. Task-Beschreibungsliste in einem vollkommen zeitgesteuerten Betriebssystem

„Der Dispatcher wird durch den synchronisierten Takt aktiviert. Er liest die TDL und führt die Aktion durch, die für diesen Zeitpunkt geplant ist ...“

Der Hauptvorteil des statischen Schedulings ist die leichte Überprüfbarkeit von Zeitbedingungen:

„Um Zeitbedingungen in Echtzeitsystemen zu erfüllen, ist die Vorhersagbarkeit des Systemverhaltens der wichtigste Punkt; ein vor der Laufzeit durchgeführtes Scheduling ist häufig die einzig praktisch anwendbare Methode, um ein vorhersagbares Verhalten eines komplexen Systems zu erhalten“ (Xu und Parnas, zitiert nach Kopetz).

Der Hauptnachteil ist die möglicherweise schlechte Antwortzeit auf sporadisch eintretende Ereignisse.

- **Zentralisiertes und verteiltes Scheduling:** Multiprozessor-Schedulingalgorithmen können entweder lokal auf einem Prozessor oder verteilt auf mehreren Rechnern ausgeführt werden.
- **Art und Komplexität von *Schedulability*-Tests:**

  In der Praxis ist es wichtig, festzustellen, ob für eine gegebene Taskmenge und zugehörige Einschränkungen überhaupt ein Schedule existiert.

  Eine Menge von Tasks heißt ***schedulable*** unter gegebenen Einschränkungen, wenn es für diese Taskmenge und die Einschränkungen ein Schedule gibt. Für viele Anwendungen ist ein ***Schedulability*-Test** wichtig. Tests, die nie ein falsches Ergebnis liefern (exakte Tests) sind in vielen Situatio-

nen NP-hart [Garey und Johnson, 1979]. Daher werden hinreichende und notwendige Tests verwendet. Bei ersteren werden hinreichende Bedingungen überprüft, unter denen ein Schedule existiert. Es gibt eine gewisse Wahrscheinlichkeit, dass ein solcher Test anzeigt, dass ein Schedule nicht garantiert ist, obwohl eines gefunden werden kann. Notwendige Tests überprüfen notwendige Bedingungen. Sie können zeigen, dass kein Schedule existiert. Es kann aber auch Fälle geben, in denen kein Schedule existiert, die Tests dies aber nicht beweisen können.

- **Mono- und Multiprozessor-Scheduling:** Einfache Schedulingalgorithmen behandeln lediglich Ein-Prozessorsysteme, wohingegen kompexere Algorithmen auch mit Systemen umgehen können, die aus mehreren Prozessoren bestehen. In diesem Fall kann man zwischen Algorithmen für homogene Multiprozessorsysteme und solchen für heterogene Multiprozessorsysteme unterscheiden. Letztere berücksichtigen Prozessor-abhängige Unterschiede bei der Task-Laufzeit und können auch in gemischten Hardware-/Software-Systemen eingesetzt werden, wobei einige Tasks auf Hardware abgebildet werden.
- **Unabhängige und abhängige Tasks:** Man kann unterscheiden zwischen Tasks ohne jegliche Inter-Task-Kommunikation (im Weiteren einfache (*simple*) oder S-Tasks genannt) und anderen Tasks, die komplexe Tasks genannt werden. S-Tasks können sich in einem von zwei möglichen Zuständen befinden: bereit (*ready*) oder in der Ausführung (*running*).

4.2.2 Aperiodisches Scheduling

Scheduling ohne Task-Abhängigkeiten

Sei $\{T_i\}$ eine Menge von Tasks. Sei weiterhin (s. Abb. 4.6):

- c_i die Ausführungszeit von T_i,
- d_i das ***Deadline*-Intervall**, das heißt die Zeit zwischen dem Zeitpunkt, an dem T_i verfügbar wird und dem Zeitpunkt, zu dem T_i beendet sein muss.
- l_i der **Spielraum** (***laxity* oder Schlupf**), definiert als

$$l_i = d_i - c_i \tag{4.1}$$

  Wenn $l_i = 0$, dann muss T_i sofort nach seiner Ankunft gestartet werden.
- f_i ist der Zeitpunkt, an dem die Ausführung von T_i beendet ist.

Scheduling-Algorithmen versuchen, verschiedene Kostenfunktionen zu minimieren. Eine mögliche Kostenfunktion ist die maximale Verspätung.

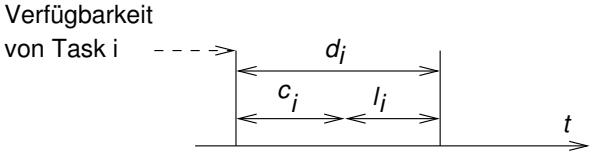


Abb. 4.6. Definition des Schlupfes einer Task

Definition: Die **Maximale Verspätung** (*maximum lateness*) ist definiert als die Differenz zwischen dem Ausführungsende und der *Deadline*, maximiert über alle Tasks. Die Maximale Verspätung ist negativ, wenn alle Tasks vor der *Deadline* abgeschlossen werden können.

Eine andere mögliche Kostenfunktion wäre die **durchschnittliche Bearbeitungszeit** der Tasks.

Zuerst betrachten wir[1] den Fall eines Einzel-Prozessorsystems, bei dem alle Tasks zur gleichen Zeit ankommen. Wenn alle Tasks gleichzeitig verfügbar werden, ist es offensichtlich sinnlos, Tasks vorzeitig zu beenden.

Earliest Due Date (EDD)-Algorithmus

Ein sehr einfacher Schedulingalgorithmus für diesen Fall wurde 1955 von Jackson vorgestellt [Jackson, 1955]. Der Algorithmus basiert auf Jacksons Regel: *„Wenn eine Menge von n unabhängigen Tasks gegeben ist, so ist jeder Algorithmus, der diese Tasks in der Reihenfolge nicht-abnehmender Deadlines ausführt, optimal in Bezug auf die Minimierung der maximalen Verspätung."*

Der Algorithmus heißt ***Earliest Due Date* (EDD)**, weil er die Tasks mit der frühesten *Deadline* zuerst ausführt. Für EDD müssen die Tasks nach ihren *Deadlines* sortiert sein. Wenn die *Deadlines* im Voraus bekannt sind, kann EDD als statischer Schedulingalgorithmus implementiert werden. Seine Komplexität ist aufgrund des Sortierens $O(n \log(n))$.

Beweis der Optimalität von EDD:

Sei σ ein Schedule, das durch irgendeinen Algorithmus A erzeugt wurde. Wenn A nicht die Wirkung von EDD hat, dann gibt es Tasks T_a und T_b derart, dass in σ die Ausführung von T_b derjenigen von T_a vorangeht, obwohl die Deadline von T_a kleiner ist als die von T_b ($d_a < d_b$). Wir betrachten nun ein Schedule σ' welches aus σ durch Vertauschen der Ausführungsreihenfolge von T_a und T_b entsteht (s. Abb. 4.7).

Die maximale Verspätung für T_a und T_b in σ ist: $L_{max}(a, b) = f_a - d_a$. Die maximale Verspätung für T_a und T_b in σ' ist das Maximum der Verspätungen beider Tasks: $L'_{max}(a, b) = max(L'_a, L'_b)$. Es gibt nun zwei mögliche Fälle:

[1] Wir verwenden in diesem Abschnitt Material aus dem Buch von Buttazzo [Buttazzo, 2002], das auch weitergehende Informationen und Referenzen enthält.

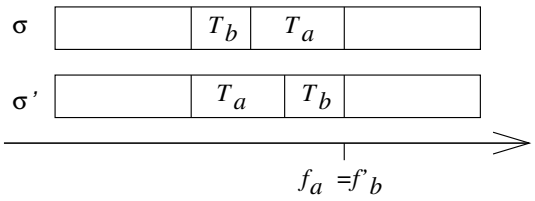


Abb. 4.7. Schedules σ und σ'

1. $L'_a > L'_b$: Dann ist
   $L'_{max}(a,b) = f'_a - d_a$
   Da Task T_a im neuen Schedule früher beendet wird, gilt
   $L'_{max}(a,b) = f'_a - d_a < f_a - d_a$.
   Die rechte Seite der Ungleichung ist aber die maximale Verspätung im Schedule σ, sodass folgt:
   $L'_{max}(a,b) < L_{max}(a,b)$
2. $L'_a \leq L'_b$:
   Dann ist
   $L'_{max}(a,b) = f'_b - d_b = f_a - d_b$ (s. Abb. 4.7).
   Da die *Deadline* von T_a vor der *Deadline* von T_b liegen soll, folgt
   $L'_{max}(a,b) < f_a - d_a$
   Es folgt also wieder
   $L'_{max}(a,b) < L_{max}(a,b)$

Mithin kann jedes von einem EDD-Schedule verschiedene Schedule in endlich vielen Vertauschungen in ein EDD-Schedule transformiert werden, wobei die maximale Verspätung nur kleiner werden kann. Also ist EDD optimal unter allen möglichen Schedulingverfahren. q.e.d.

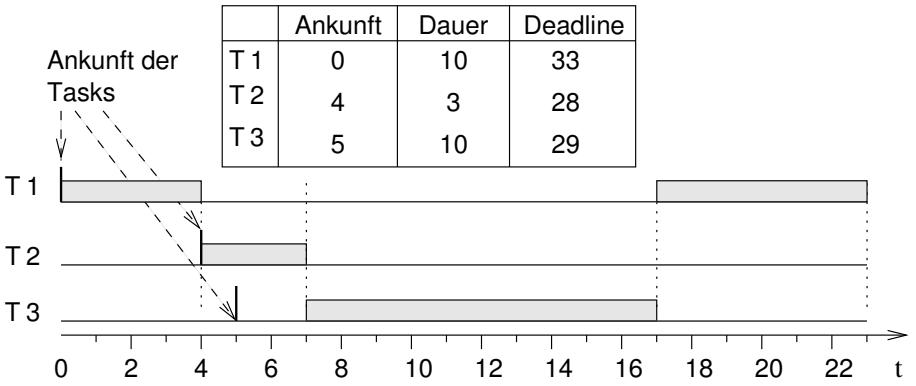
Earliest Deadline First (EDF)-Algorithmus

Betrachten wir nun den Fall von unterschiedlichen Ankunftszeiten für ein Ein-Prozessorsystem. Bei diesem Szenario kann ein Unterbrechen von Tasks (*preemption*) potentiell die maximale Verspätung verbessern.

Der ***Earliest Deadline First* (EDF)**-Algorithmus ist optimal in Bezug auf die Minimierung der maximalen Verspätung. Er basiert auf dem folgenden Theorem [Horn, 1974]: *„Wenn eine Menge von n Tasks mit beliebigen Ankunftszeiten gegeben ist, so ist ein Algorithmus, der zu jedem Zeitpunkt diejenige ausführungsbereite Task mit der frühesten absoluten Deadline ausführt, optimal in Bezug auf die Minimierung der maximalen Verspätung.“*

Bei EDF muss jede ankommende ausführbereite Task in eine Warteschlange von ausführbereiten Tasks eingefügt werden. Die Tasks in der Warteschlange sind dabei nach ihrer *Deadline* sortiert. Daher ist EDF ein dynamischer Schedulingalgorithmus. Wenn eine neu angekommene Task als erstes Element in die Warteschlange eingefügt wird, muss die momentan ausgeführte Task beendet werden. Wenn für die Warteschlange sortierte Listen verwendet werden, ist die Komplexität von EDF $O(n^2)$. Sogenannte *Bucket*-Arrays können zur Reduzierung der Laufzeit beitragen.

Abb. 4.8 zeigt ein Schedule, das mit Hilfe des EDF-Algorithmus erzeugt wurde. Die vertikalen Striche deuten die Ankunft einer neuen Task an.



	Ankunft	Dauer	Deadline
T1	0	10	33
T2	4	3	28
T3	5	10	29

Abb. 4.8. EDF-Schedule

Zum Zeitpunkt 4 hat Task T2 eine frühere *Deadline*. Daher wird Task T1 beendet. Zum Zeitpunkt 5 kommt Task T3 an. Wegen ihrer späteren *Deadline* wird T2 hier nicht unterbrochen.

Beweis der Optimalität von EDF:

Sei σ ein Schedule, welches durch einen von EDF verschiedenen Algorithmus A erzeugt wird. Sei σ_{EDF} ein durch EDF erzeugtes Schedule. Wir zerlegen nunmehr die Zeitachse in diskrete Zeitabschnitte von jeweils einer Zeiteinheit. Jeder Zeitabschnitt enthält die Zeiten im Intervall $[t, t{+}1)$. Sei $\sigma(t)$ die Task, die gemäß Schedule σ im Zeitabschnitt $[t, t{+}1)$ ausgeführt wird. Sei $E(t)$ die Task, welche zur Zeit t unter allen verfügbaren Tasks die früheste *Deadline* besitzt. Sei ferner $t_E(t)$ die Zeit ($\geq t$), zu welcher die Task $E(t)$ im Schedule σ ihre Ausführung beginnt.

Wenn das Schedule σ kein EDF-Schedule ist, dann gibt es eine Zeit t, zu der nicht die Task mit der frühesten *Deadline* ausgeführt wird, für die also $\sigma(t) \neq E(t)$ gilt (s. Abb. 4.9). Deadlines sind durch Pfeile nach unten dargestellt.

Die Grundidee des Beweises ist jetzt, dass das Vertauschen von $\sigma(t)$ und $E(t)$ (s. Abb. 4.10) die maximale Verspätung nicht erhöhen kann.

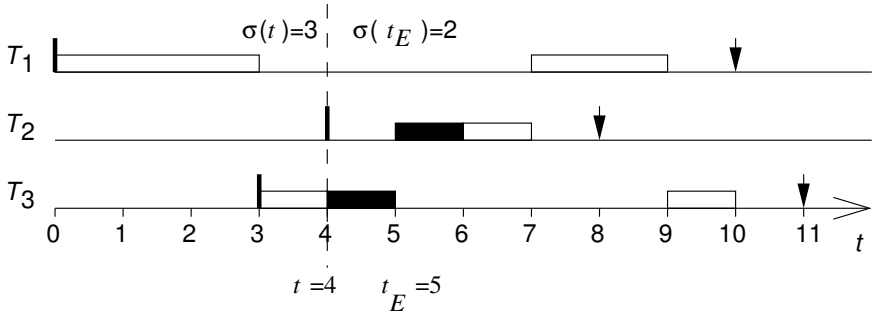


Abb. 4.9. Schedule σ

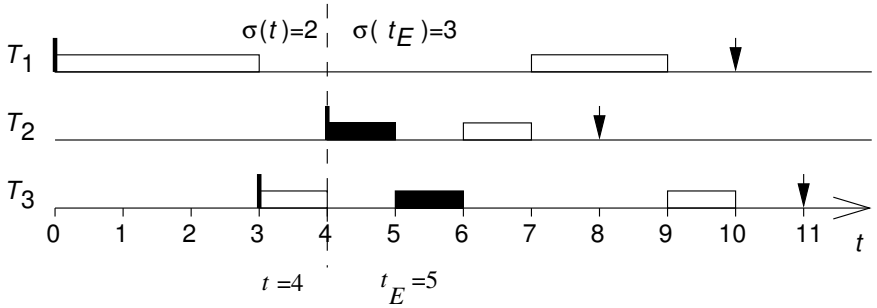


Abb. 4.10. Schedule nach Vertauschung der Tasks $\sigma(t)$ und $E(t)$

Sei D die späteste *Deadline*. Dann kann σ_{EDF} aus σ durch maximal D Vertauschungen mittels des folgenden Algorithmus erhalten werden:

for (t=0 **to** D-1)
  if $(\sigma(t) \neq E(t))$
    $\sigma(t_E) = \sigma(t)$;
    $\sigma(t) = E(t)$;

Mit demselben Argument wie beim Beweis von Jackson's Regel kann gezeigt werden, dass das Vertauschen die maximale Verspätung nicht erhöht. Damit kann aus jedem von einem EDF-Schedule verschiedenen Schedule ein EDF-Schedule erzeugt werden, wobei sich die maximale Verspätung nicht erhöht. Damit ist EDF unter allen möglichen Schedulingverfahren optimal.

Wir können zeigen, dass das Vertauschen die *Deadlines* der einzelnen Tasks einhält, sofern sie in σ eingehalten wurden. Zunächst betrachten wir die Task $E(t)$. Da sie im neuen Schedule früher ausgeführt wird als im alten, hält sie ihre *Deadline* ein, wenn sie im alten Schedule eingehalten wurde. Als nächstes betrachten wir die Task $\sigma(t)$. Task $\sigma(t)$ muss eine spätere *Deadline* haben als Task $E(t)$. Also hält Task $\sigma(t)$ die *Deadline* im neuen Schedule ein, wenn $E(t)$ die *Deadline* im alten Schedule eingehalten hat.

Least Laxity (LL)

Least Laxity (LL), *Least Slack Time First* (LST) und *Minimium Laxity First* (MLF) sind drei Namen für eine weitere Scheduling-Strategie [Liu, 2000]. Beim LL-Scheduling sind die Prioritäten der Tasks eine monoton fallende Funktion ihres Schlupfs (s. Gleichung 4.1; je weniger Spielraum eine Task also hat, desto höher ihre Priorität). Der Schlupf verändert sich dynamisch. LL ist auch ein präemptives Schedulingverfahren.

Abb. 4.11 zeigt ein Beispiel eines LL-Schedules zusammen mit den berechneten *Laxity*-Werten.

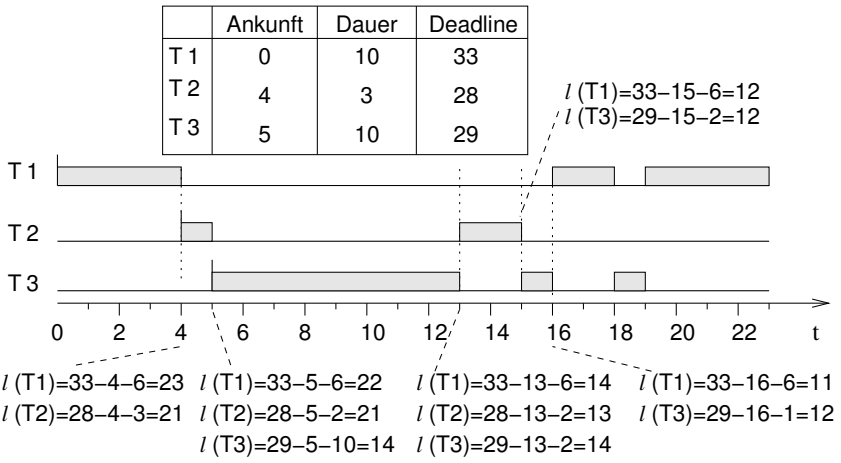


Abb. 4.11. *Least-Laxity*-Schedule

Zum Zeitpunkt 4 wird Task T1 wie oben unterbrochen. Zum Zeitpunkt 5 wird Task T2 nun auch unterbrochen, weil sie einen höheren Schlupf hat als Task T3.

Man kann zeigen (in [Liu, 2000] ist dies als Übung dem Leser überlassen), dass LL auch ein optimales Schedulingverfahren für Ein-Prozessorsysteme in dem Sinne ist, dass es ein Schedule findet, wenn ein Schedule existiert. Wegen seiner dynamischen Prioritäten kann man es nicht in Standard-Betriebssystemen einsetzen, da diese überlicherweise nur statische Task-Prioritäten verwalten können. LL-Scheduling erfordert ein periodisches Überprüfen des Schlupfes und benötigt und berücksichtigt (im Gegensatz zu EDF) Wissen über die Ausführungszeit. Es ist in Lage, bereits frühzeitig zu erkennen, dass eine *Deadline* nicht mehr eingehalten werden kann.

Verfahren ohne Unterbrechung

Wenn die Unterbrechung von Tasks nicht erlaubt ist, dürfen optimale Schedulingverfahren den Prozessor möglicherweise nicht vollständig auslasten, damit spät ankommende Tasks mit knappen *Deadlines* noch ausgeführt werden können.

Beweis: Nehmen wir an, dass ein optimaler nicht-präemptiver Scheduler (der kein Wissen über die Zukunft hat) den Prozessor immer auslastet. Dieser Scheduler muss das Schedule in Abb. 4.12 optimal erzeugen (d.h. er muss ein Schedule finden, wenn eines existiert).

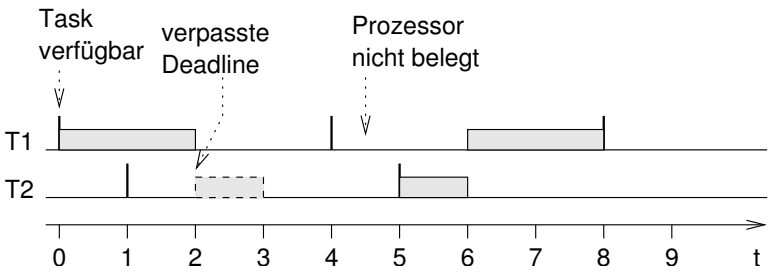


Abb. 4.12. Scheduler darf den Prozessor nicht vollständig auslasten

Für das Beispiel in Abb. 4.12 nehmen wir zwei Tasks an. Sei T1 ein periodischer Prozess mit einer Ausführungszeit von 2, einer Periode von 4 und einem *Deadline*intervall von 4. Sei T2 eine Task, die nur zu den Zeitpunten $4 * n + 1$ ausführungsbereit ist und eine Ausführungszeit und ein *Deadline*intervall von 1 hat. Wir nehmen an, dass die gleichzeitige Ausführung von T1 und T2 aufgrund von Ressourcenkonflikten nicht möglich ist. Unter diesen Annahmen muss der Scheduler die Ausführung von Task T1 zum Zeitpunkt 0 starten, da er ja den Prozessor voll auslasten soll. Da der Scheduler nicht-präemptiv ist, kann er T2 nicht starten, wenn sie zum Zeitpunkt 1 ausführbereit ist. Daher verpasst T2 ihre *Deadline*. Wenn der Scheduler den Prozessor nicht sofort belegt hätte, (wie in Abb. 4.12 zum Zeitpunkt 4 gezeigt), hätte er ein zulässiges Schedule gefunden. Daher ist dieser Scheduler nicht optimal. Dies ist ein Widerspruch zur Annahme, dass optimale Scheduler den Prozessor immer voll auslasten. q.e.d.

Abschließend stellen wir fest: um verpasste *Deadlines* zu vermeiden, muss der Scheduler Wissen über die Zukunft haben. Wenn a priori kein Wissen über Ankunftszeiten von Tasks verfügbar ist, kann kein *on line*-Schedulingalgorithmus entscheiden, ob der Prozessor im Leerlauf bleiben soll oder nicht. Es wurde gezeigt, dass EDF optimal ist unter allen Algorithmen, welche den Prozessor nicht unbeschäftigt lassen, wenn ausführbereite Tasks vorliegen. Wenn die Ankunftszeiten a priori bekannt sind, wird das Schedulingproblem im

nicht-unterbrechenden Fall im Allgemeinen NP-hart und *Branch-and-Bound*-Techniken werden typischerweise zur Erzeugung von Schedules verwendet.

Scheduling bei abhängigen Tasks

Wir beginnen mit einem Taskgraphen, der Abhängigkeiten zwischen den Tasks darstellt (s. Abb. 4.13). Task T3 kann erst ausgeführt werden, wenn Tasks T1 und T2 abgearbeitet sind und jeweils eine Nachricht an T3 gesendet haben.

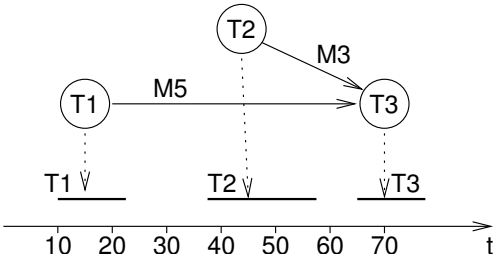


Abb. 4.13. Präzedenzgraph und Schedule

Die Abbildung zeigt auch ein gültiges Schedule. Bei statischem Scheduling kann dieses Schedule in einer Tabelle abgelegt werden, die dem *Dispatcher* anzeigt, zu welchen Zeitpunkten welche Task gestartet werden muss und welche Nachrichten gesendet werden müsssen.

Ein optimaler Algorithmus, der die maximale Verspätung im Falle gleichzeitig ankommender Tasks minimiert, wurde von Lawler vorgestellt [Lawler, 1973]. Dieser Algorithmus heißt ***Latest Deadline First* (LDF)**. LDF basiert auf einer totalen Ordnung der Tasks. Diese Ordnung muss mit der partiellen Ordnung kompatibel sein, die der Taskgraph definiert. LDF liest den Taskgraphen und speichert diejenige Task, welche die späteste *Deadline* und keine Nachfolger im Taskgraphen hat, in einem Stapel. Dies wird für alle verbleibenden Tasks wiederholt. Wenn es nur eine globale *Deadline* gibt, führt LDF prinzipiell eine **topologische Sortierung** durch [Sedgewick, 1988]. Zur Laufzeit werden die Tasks in der so erzeugten Reihenfolge ausgeführt. LDF ist nichtpräemptiv und optimal für Ein-Prozessorsysteme.

Im Falle von asynchronen Ankunftszeiten der Tasks kann man mit einem modifizierten EDF-Algorithmus ein gültiges Schedule bestimmen. Die Grundidee besteht darin, das Problem mit der gegebenen Menge abhängiger Tasks in eine äquivalente Menge unabhängiger Tasks mit unterschiedlichen Zeitparametern umzuwandeln [Chetto et al., 1990]. Dieser Algorithmus ist wiederum optimal für Ein-Prozessorsysteme.

Wenn Tasks nicht unterbrochen werden dürfen, kann der heuristische Algorithmus aus [Stankovic und Ramamritham, 1991] verwendet werden.

4.2.3 Periodisches Scheduling

Notation

Jetzt betrachten wir den Fall periodischer Tasks. Ein periodischer Scheduling-Algorithmus wird als **optimal** bezeichnet, wenn er immer ein Schedule findet, wenn eines existiert.

Sei $\{T_i\}$ eine Menge von Tasks. Jede Ausführung einer Task T_i heißt **Job**. Die Ausführungszeit aller Jobs, die zu einer Task gehören, wird als immer gleich angenommen. Sei weiterhin (s. Abb. 4.14)

- p_i die Periode von Task T_i,
- d_i das ***Deadline*intervall**, also die Zeit zwischen dem Zeitpunkt, zu dem T_i verfügbar wird und dem Zeitpunkt, zu dem die Ausführung beendet sein muss,
- c_i die Ausführungszeit von T_i,
- l_i die ***Laxity*** oder der **Schlupf**, definiert als

$$l_i = d_i - c_i \tag{4.2}$$

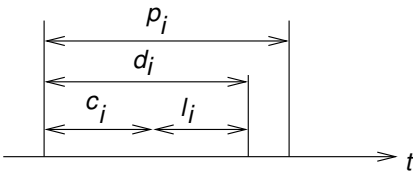


Abb. 4.14. verwendete Notation für Zeitintervalle

Wenn $l_i = 0$, dann muss T_i sofort nach seiner Verfügbarkeit gestartet werden.

Sei μ die **durchschnittliche Prozessorauslastung** für eine Menge von n Prozessen, also die akkumulierte Ausführungszeit der Prozesse dividiert durch ihre Periode:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \tag{4.3}$$

Unter der Annahme, dass die Ausführungszeiten auf einer Menge von m Prozessoren gleich groß sind, ergibt Gleichung 4.4 eine notwendige Bedingung für die Existenz eines Schedules:

$$\mu \leq m \tag{4.4}$$

Unabhängige Tasks

Zunächst betrachten wir den Fall unabhängiger Tasks.

Rate Monotonic Scheduling

Rate Monotonic Scheduling (RMS) [Liu und Layland, 1973] ist wohl der bekannteste Schedulingalgorithmus für unabhängige periodische Tasks. *Rate Monotonic Scheduling* basiert auf den folgenden Annahmen („**RM-Annahmen**“):

1. Alle Tasks mit harter *Deadline* sind periodisch.
2. Alle Tasks sind voneinander unabhängig.
3. $d_i = p_i$ für alle Tasks.
4. c_i ist konstant und für alle Tasks bekannt.
5. Die Zeit für einen Kontextwechsel ist vernachlässigbar.
6. Für einen Prozessor und n Tasks gilt die folgende Gleichung bzgl. der durchschnittlichen Auslastung μ:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \quad \leq \quad n(2^{\frac{1}{n}} - 1) \tag{4.5}$$

Abb. 4.15 zeigt Werte der rechten Seite der Gleichung 4.5.

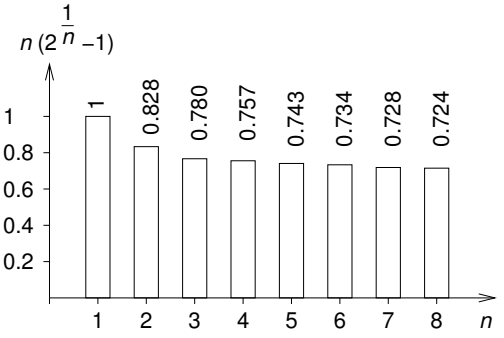


Abb. 4.15. Werte der rechten Seite von Gleichung 4.5

Die rechte Seite der Gleichung hat für große Werte von n einen Wert von ungefähr 0,7:

$$\lim_{n\to\infty} n * (2^{\frac{1}{n}} - 1) = ln(2) \ \ (\approx 0,7) \tag{4.6}$$

Beim *Rate Monotonic Scheduling* ist **die Priorität der Tasks eine monoton fallende Funktion ihrer Periode bzw. eine monoton steigende Funktion der Ausführungsrate**. Anders ausgedrückt haben Tasks mit einer kurzen Periode eine höhere Priorität, wohingegen Tasks mit langer Periode eine geringere Priorität haben. RM Scheduling ist **präemptiv** mit **festen Prioritäten**. Aus Gleichung 4.5 ergibt sich, dass ein gewisser Teil der Rechenleistung des Prozessors nicht genutzt werden kann, um sicherzustellen, dass keine Task ihre *Deadline* verpasst.

Abb. 4.16 zeigt ein Beispiel eines mit RM Scheduling erzeugten Schedules.

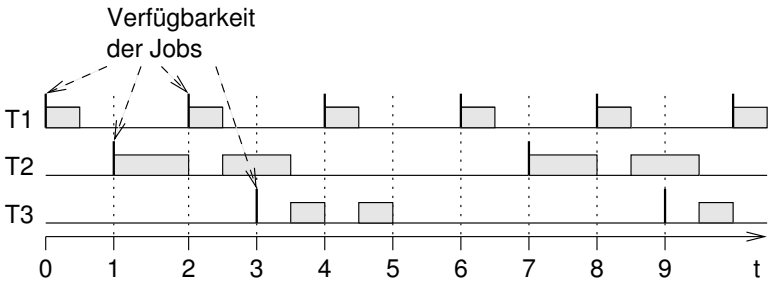


Abb. 4.16. Beispiel eines RM-Schedules

Vertikale Striche deuten die Ankunftszeiten der Tasks an. Tasks 1, 2 und 3 haben jeweils eine Periode von 2, 6 und 6. Die Ausführungszeiten betragen 0,5, 2 und 1. Task 1 hat die kürzeste Periode und daher die höchste Priorität. Jedesmal, wenn Task 1 ausführbereit ist, unterbricht ihr Job die gerade aktive Task. Task 2 hat die gleiche Periode wie Task 3, daher unterbrechen sich diese beiden Tasks nicht.

Rate Monotonic Scheduling (RMS) hat die folgenden wichtigen Vorteile:

- RMS basiert auf **statischen** Prioritäten. Das verringert die Anforderungen an das Betriebssystem und ermöglicht die Verwendung von RMS in Standard-Betriebssystemen, die feste Prioritäten unterstützen, wie etwa Windows NT (s. [Ramamritham et al., 1998], [Ramamritham, 2002]).
- Wenn die oben genannten sechs RM-Annahmen (s. Seite 150) erfüllt sind, werden alle *Deadlines* eingehalten (s. Buttazzo [Buttazzo, 2002]). Man kann beweisen, dass *Rate Monotonic Scheduling* für Ein-Prozessorsysteme optimal ist.

RMS ist die Basis vieler formaler *Schedulability*-Beweise.

Abb. 4.17 zeigt einen Fall, bei dem Gleichung 4.5 nicht eingehalten wird, also nicht genügend Leerlaufzeit existiert, um die Existenz eines gültigen RM-Schedules zu garantieren. Eine Task hat eine Periode von 5 und eine Ausführungszeit von 3, wohingegen die zweite Task eine Periode von 8 und eine Ausführungszeit von 3 hat. Task T2 wird mehrfach unterbrochen.

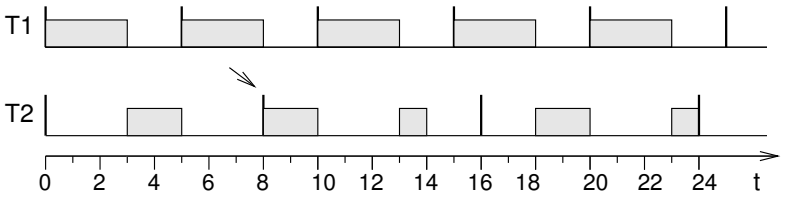


Abb. 4.17. RM-Schedule verpasst die *Deadline* zum Zeitpunkt 8

Dieser spezielle Fall hat eine durchschnittliche Auslastung von $\mu = \frac{3}{5} + \frac{3}{8} = \frac{39}{40} = 0,975$. Andererseits ist $2 * (2^{\frac{1}{2}} - 1) \approx 0{,}828$. Daher ist bei Verwendung von RM-Scheduling nicht garantiert, dass ein Schedule existiert. Die *Deadline* wird in diesem Beispiel zum Zeitpunkt 8 verpasst. Wir nehmen hier an, dass die Berechnungen, die ihre *Deadline* verpasst haben, nicht in der folgenden Periode nachgeholt werden.

Allerdings ist die Leerlaufzeit oder Rechenzeitreserve des Prozessors nicht unbedingt notwendig. Man kann zeigen, dass RM-Scheduling auch genau dann optimal ist, wenn statt Gleichung 4.5 folgende Gleichung gilt:

$$\mu \leq 1 \tag{4.7}$$

unter der zusätzlichen Voraussetzung, dass die Periode aller Tasks ein ganzzahliges Vielfaches der Periode der Task mit der jeweils höheren Priorität ist.

Die Gleichungen 4.5 oder 4.7 sind gut geeignet, um in Kombination mit den RM-Annahmen hinreichende Bedingungen für die Existenz eines gültigen RM-Schedules zu überprüfen.

Bei der Konstruktion von Beispielen und beim Führen von Beweisen ist es hilfreich, zu wissen, welche Situationen hinsichtlich des Findens von Schedules mit RMS besonders kritisch sind.

Definition: Ein Zeitpunkt t heißt **kritischer Zeitpunkt** einer Task, wenn eine Bereitstellung der Task zu diesem Zeitpunkt deren Antwortzeit maximiert.

Lemma: Für jede Task entstehen die kritischen Zeitpunkte, wenn sie gleichzeitig mit allen Tasks einer höheren Priorität bereitgestellt wird.

Beweis: Sei $T = \{T_1, ..., T_n\}$ eine Menge periodischer Tasks mit: $\forall i : p_i \leq p_{i+1}$. Wir betrachten Task T_n und eine Task T_i höherer Prorität. Dann wird die Antwortzeit von T_n durch Unterbrechungen durch Tasks einer höheren Priorität verlängert (s. Abb. 4.18).

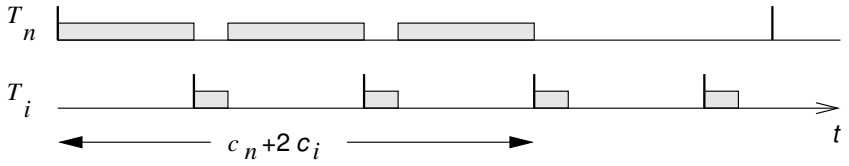


Abb. 4.18. Verzögerung von Task T_n durch eine Task T_i höherer Priorität

Die Anzahl der überlappten Ausführungen nimmt potenziell zu, wenn T_i früher ausführbereit ist (s. Abb. 4.19). Die maximale Verzögerung ergibt sich offenbar, wenn T_i und T_n gleichzeitig ausführbereit sind.

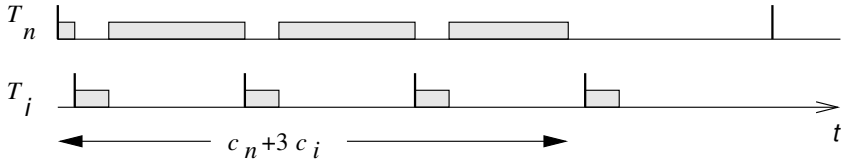


Abb. 4.19. Wachsende Verzögerung von Task T_n

Die Argumente für T_n und T_i können für alle übrigen Paare von Tasks wiederholt werden. Damit ergibt sich der kritische Zeitpunkt einer Task dann, wenn sie gleichzeitig mit allen Tasks einer höheren Priorität bereitgestellt wird. q.e.d.

Beim Beweis der Optimalität von RMS muss daher nur der Fall betrachtet werden, in dem die Tasks gleichzeitig ausführbereit werden.

Earliest Deadline First Scheduling

EDF kann auch auf Mengen periodischer Tasks angewendet werden. Aus der Optimalität von EDF für nicht-periodische Schedules folgt, dass EDF auch für periodische Tasks optimal ist. Es müssen keine zusätzlichen Bedingungen eingehalten werden, um diese Optimalität zu erreichen. Daraus folgt insbesondere, dass EDF auch für den Fall $\mu = 1$ optimal ist. Daher tritt bei Verwendung von EDF im Beispiel 4.17 keine verpasste *Deadline* auf (s. Abb. 4.20). Zum Zeitpunkt 5 unterscheidet sich das Verhalten vom RMS: wegen seiner früheren *Deadline* wird Task T2 nicht unterbrochen.

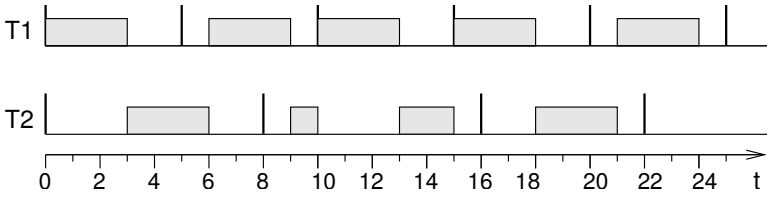


Abb. 4.20. EDF-Schedule für das Beispiel aus Abb. 4.17

Da EDF dynamische Prioritäten verwendet, kann es nicht in solchen Standard-Betriebssystemen verwendet werden, die nur feste Prioritäten für die Tasks vorsehen. Das Verfahren kann auch für den Fall angepasst werden, dass die *Deadlines* sich von den Perioden unterscheiden.

Abhängige Tasks

Das Scheduling abhängiger Tasks ist schwieriger als jenes unabhängiger Tasks. Das Problem, zu entscheiden, ob ein Schedule für eine gegebene Menge abhängiger Tasks und für eine gegebene *Deadline* existiert, ist NP-vollständig [Garey und Johnson, 1979]. Es gibt verschiedene Ansätze, um den Aufwand beim Scheduling zu reduzieren:

- Hinzufügen weiterer Ressourcen, um das Scheduling einfacher zu machen, und
- Partitionieren des Schedulings in einen statischen und einen dynamischen Teil. Mit diesem Ansatz können viele Entscheidungen bereits zur Entwurfszeit getroffen werden, und nur noch eine minimale Anzahl von Entscheidungen verbleibt zur Laufzeit.

Sporadische Ereignisse

Prinzipiell könnte man sporadische Ereignisse mit Unterbrechungen (*Interrupts*) verbinden und sie jedesmal sofort ausführen, wenn die Interrupt-Priorität die höchste im gesamten System ist. Das hätte allerdings ein unvorhersagbares Verhalten für alle anderen Tasks zur Folge. Daher werden besondere *sporadic task servers* verwendet, die regelmäßig ausgeführt werden und dabei prüfen, ob ausführbereite sporadische Ereignisse existieren. So können sporadische Ereignisse praktisch in periodische Tasks umgewandelt werden, wodurch die Vorhersagbarkeit des Gesamtsystems deutlich verbessert wird.

4.2.4 Ressourcen-Zugriffs-Protokolle

Prioritäts-Umkehr

In manchen Fällen müssen Tasks exklusiven Zugriff auf Ressourcen wie etwa gemeinsam genutzte globale Variablen oder Geräte erhalten, um ein nichtdeterministisches oder allgemein ein unerwünschtes Programmverhalten zu verhindern. Programmteile, in denen solch ein exklusiver Zugriff notwendig ist, heißen **kritische Abschnitte**. Betriebssysteme bieten normalerweise Möglichkeiten (sogenannte *primitives*) an, um exklusiven Zugriff auf Ressourcen anzufordern und um die Ressourcen wieder freizugeben. Diese heißen ***Mutex*-Primitive** (von *Mutual exclusion*, gegenseitiger Ausschluss). Tasks, die keinen exklusiven Zugriff haben, müssen warten, bis die Ressource freigegeben wird. Daher muss die Freigabe-Operation überprüfen, ob gerade eine andere Task auf die Ressource wartet und diejenige wartende Task mit der höchsten Priorität aufwecken. Wir nennen die Anforderung einer Ressource P(S) und die Freigabe-Operation V(S), wobei S der jeweiligen Ressource entspricht. Kritische Abschnitte sollten möglichst kurz gehalten werden.

Bei Tasks mit kritischen Abschnitten kann die sogenannte **Prioritätsumkehr** (*priority inversion*) auftreten. Ein Beispiel einer solchen Prioritätsumkehr ist in Abb. 4.21 dargestellt. Task T1 habe eine höhere Priorität als Task T2.

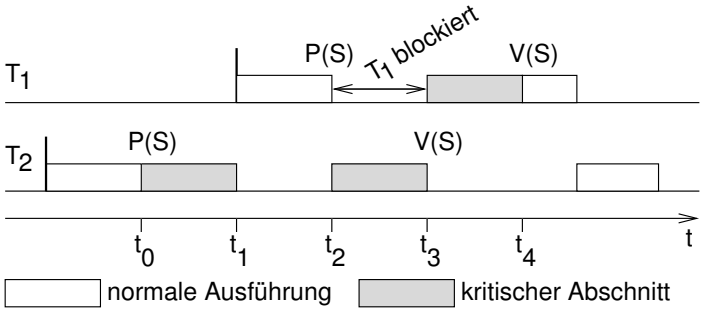


Abb. 4.21. Prioritätsumkehr für zwei Tasks

Zum Zeitpunkt t_0 betritt Task T2 einen kritischen Abschnitt, nachdem sie den exklusiven Zugriff durch eine P-Operation angefordert hat. Zum Zeitpunkt t_1 wird Task T1 ausführbereit und unterbricht T2. Bei t_2 erhält T1 keinen exklusiven Zugriff auf die Ressource, die ja noch von T2 verwendet wird. Folglich wird Task T1 blockiert. Task T2 wird also fortgesetzt und gibt nach einiger Zeit die Ressource frei. Diese Freigabe-Operation prüft, ob Tasks mit höherer Priorität auf die Ressource warten und unterbricht Task T2. Effektiv wurde somit T1 mit ihrer hohen Priorität von T2, einem Prozess mit niedrigerer Priorität, blockiert. Dieser Effekt heißt **Prioritätsumkehr**.

Im Allgemeinen tritt Prioritätsumkehr auf, wenn eine Task mit niedriger Priorität eine Task mit höherer Priorität blockiert, weil erstere exklusiven Zugriff auf eine Ressource hat. Die Notwendigkeit des exklusiven Zugriffs auf einige Ressourcen ist der Hauptgrund für das Auftreten einer Prioritätsumkehr.

Im speziellen Fall von Abb. 4.21 kann die Dauer der Blockierung die Länge des kritischen Abschnitts von Task T2 nicht überschreiten. Leider gibt es im allgemeinen Fall keine solche obere Schranke. Dies wird in Abb. 4.22 gezeigt.

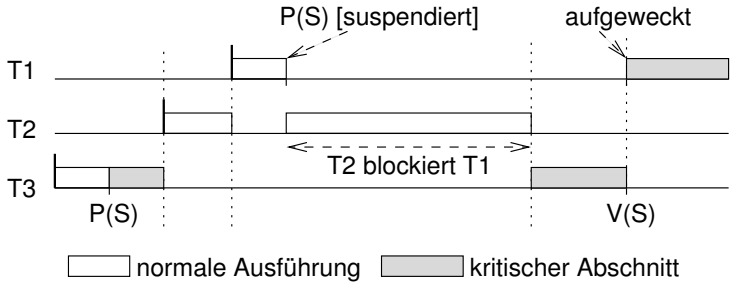


Abb. 4.22. Prioritätsumkehr mit potentiell langer Blockade

Die Tasks T1, T2 und T3 seien gegeben. T1 hat die höchste Priorität, T2 hat eine mittlere Priorität und T3 hat die niedrigste Priorität. Außerdem nehmen wir an, dass T1 und T3 mittels einer P(S)-Operation exklusiven Zugriff auf eine Ressource anfordern. Sei nun T3 in ihrem kritischen Abschnitt, wenn sie von T2 unterbrochen wird. Wenn T2 ihrerseits von T1 unterbrochen wird, die die Ressource verwenden möchte, die gerade von T3 verwendet wird, wird T1 blockiert und lässt T2 weiterarbeiten. Solange T2 arbeitet, kann T3 ihre Ressource nicht freigeben. Daher wird T1 effektiv von T2 blockiert, obwohl T1 eine höhere Priorität als T2 hat. In diesem Beispiel dauert die Prioritätsumkehr so lange an, wie T2 ausgeführt wird. Daher ist die Länge der Prioritätsumkehr nicht durch die Länge eines kritischen Abschnitts nach oben beschränkt.

Der wohl bekannteste Fall von Prioritätsumkehr geschah im *Mars Pathfinder*: ein exklusiver Zugriff auf einen gemeinsam genutzten Speicherbereich führte zu einer Prioritätsumkehr auf dem Mars [Jones, 1997].

Prioritätsvererbung

Eine Möglichkeit, die Prioritätsumkehr zu vermeiden, besteht in der Verwendung des Prioritätsvererbungs-Protokolls (*priority inheritance protocol*). Dieses funktioniert wie folgt:

- Tasks werden gemäß ihrer aktiven Prioritäten eingeplant. Tasks mit der gleichen Priorität werden in der Reihenfolge eingeplant, in der sie ausführbereit werden.

- Wenn eine Task T1 eine P(S)-Operation ausführt und die Ressource ist bereits exklusiv einer anderen Task T2 zugeordnet, so wird T1 blockiert. Wenn die Priorität von T2 niedriger ist als die von T1, erbt T2 die Priorität von T1. Daher führt T2 die Ausführung fort. Im Allgemeinen erbt eine Task die höchste Priorität der Tasks, die von ihr blockiert werden.
- Wenn eine Task T2 eine V(S)-Operation ausführt, wird ihre Priorität herabgesetzt auf die höchste Priorität der Tasks, die sie blockiert[2]. Wenn keine weitere Task von T2 blockiert wird, so wird ihre Priorität auf den ursprünglichen Wert zurückgesetzt. Außerdem wird diejenige Task mit der höchsten Priorität fortgesetzt, die wegen der Ressource S blockiert war.
- Prioritätsvererbung ist transitiv: wenn T0 von T1 blockiert wird, und T1 wird von T2 blockiert, so erbt T2 die Priorität von T0.

Im Beispiel von Abb. 4.22 würde T3 die Priorität von T1 erben, wenn T1 die Operation P(S) ausführt. Das vermeidet das oben erwähnte Problem, da T3 folglich nicht von T2 unterbrochen werden kann (s. Abb. 4.23).

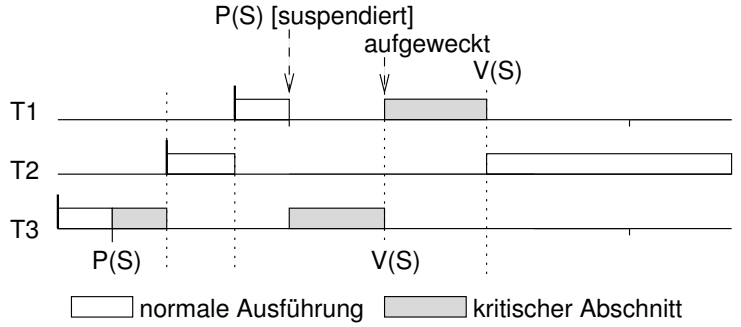


Abb. 4.23. Prioritätsvererbung für das Beispiel aus Abb. 4.22

Prioritätsvererbung wird auch in ADA verwendet: während eines *Rendez-Vous* wird die Priorität der beiden Tasks auf das Maximum ihrer Prioritäten gesetzt.

Die Prioritätsvererbung löste auch das Problem des *Mars Pathfinder*: das VxWorks-Betriebssystem, das im Pathfinder eingesetzt wurde, hat ein *Flag* für den Aufruf von *Mutex*-Primitiven, welches die Verwendung der Prioritätsvererbung aktiviert. Als die Software ausgeliefert wurde, war das entsprechende *Flag* deaktiviert. Das auf dem Mars aufgetretene Problem wurde durch die *Debugging*-Schnittstelle von VxWorks gelöst: das Prioritätsvererbungs-*Flag* wurde aktiviert, während der Pathfinder sich schon auf dem Mars befand [Jones, 1997].

[2] Es kann nur bei Verwendung von mehr als einer Ressource auftreten, dass dies nicht die ursprüngliche Priorität der Task ist.

Prioritätsvererbung ist also in der Lage, einige auftretende Probleme zu lösen, leider allerdings nicht alle. Es kann eine große Anzahl von Tasks mit hoher Priorität geben, und es kann zu sogenannten *Deadlocks* kommen. Insbesondere die Transitivität der Prioritätsvererbung kann zu Schwierigkeiten führen. Yodaiken [Yodaiken, 2004] argumentiert, dass Systeme möglichst so entworfen werden sollten, dass die Prioritätsumkehr vermieden wird. Lee [Lee, 2006] zeigt, dass manche der Probleme durch die Wahl von nebenläufigen Prozessen als Berechnungmodell entstehen. Er schlägt die Nutzung alternativer Berechnungsmodelle vor. Allerdings sind diese derzeit für komplexe Anwendungen unzureichend entwickelt. Für nebenläufige Prozesse kann das komplexere *Priority Ceiling Protocol* [Sha et al., 1990] eingesetzt werden.

4.3 Eingebettete Betriebssysteme

4.3.1 Allgemeine Anforderungen

Die Aufgaben des Scheduling, der Taskumschaltung und der Ein- und Ausgabe sind, außer bei sehr einfachen Systemen, auf die Unterstützung eines Betriebssystems angewiesen, das für eingebettete Anwendungen geeignet ist. Algorithmen zum Taskwechsel (oder der sogenannte *dispatch*) wirken wie Multiplexer zwischen Tasks und Prozessor, so dass jede Task ihren eigenen (virtuellen) Prozessor zu haben scheint. Die folgenden Eigenschaften sind für Echtzeit-Betriebssysteme und für Betriebssysteme in eingebetteten Systemen unerlässlich:

- Wegen der großen Anzahl unterschiedlicher eingebetteter Systeme gibt es auch viele Anforderungen an die Funktionalitäten eingebetteter Betriebssysteme. Aufgrund der Effizienzanforderungen ist es nicht möglich, mit einem Betriebssystem zu arbeiten, dass die Vereinigungsmenge aller überhaupt geforderten Funktionalitäten erfüllt. Aus diesem Grunde werden Betriebssysteme benötigt, die sich an die jeweiligen Bedürfnisse des betrachteten Anwendungsgebiets **flexibel anpassen** lassen. Die **Konfigurierbarkeit** ist somit eine der Haupteigenschaften eingebetteter Betriebssysteme. Im einfachsten Fall bedeutet Konfigurierbarkeit die Möglichkeit, nicht benötigte Funktionen zu entfernen (dies kann bis zu einem gewissen Grad z.B. von einem Linker erledigt werden). Eine komplexere Möglichkeit ist die bedingte Übersetzung (unter Ausnutzung der Präprozessor-Befehle `#if` und `#ifdef`). Aufwendig zu verwaltende dynamische Daten können durch einfachere, an den Anwendungsfall angepasste statische Daten ersetzt werden. Weitergehende Analysen und Compiler-Optimierungen können in diesem Zusammenhang auch die Effizienz steigern. Der Einsatz objektorientierter Programmierung kann zur sauberen Ableitung von Unterklassen genutzt werden. Ein potentielles Problem bei der Verwendung

eines Systems mit einer Vielzahl individuell angepasster Betriebssysteme ist die Validierung. Jedes einzelne Betriebssystem muss gründlich getestet werden. Takada sieht dies als potentielles Problem von eCOS, einem *Open-Source* Echtzeitbetriebssystem von Red Hat, das 100 bis 200 konfigurierbare Eigenschaften besitzt [Takada, 2001].

- In eingebetteten Systemen werden viele unterschiedliche Peripheriegeräte eingesetzt. Viele eingebettete Systeme verfügen nicht über eine Festplatte, eine Tastatur, einen Bildschirm oder eine Maus. Es gibt praktisch **kein Gerät, das von allen Versionen des Betriebssystems unterstützt werden müsste**, ausgenommen vielleicht der System-*Timer*. Folglich erscheint es sinnvoll, relativ langsame Peripheriegeräte wie Platten oder Netzwerkschnittstellen mittels eigener Tasks zu bearbeiten, anstatt die entsprechenden Treiber in den Kern des Betriebssystems zu integrieren.
- **Schutzmechanismen** werden nicht immer benötigt, da eingebettete Systeme typischerweise für einen bestimmten Einsatzzweck entworfen werden und somit kaum ungetestete Programme geladen werden dürften. Nachdem die Software getestet worden ist, kann angenommen werden, dass sie verlässlich arbeitet. Schutzmechanismen können trotzdem aus Sicherheits- und Authentifizierungsgründen notwendig sein. In den meisten Fällen haben eingebettete Systeme aber keine speziellen Schutzmechanismen. Das gilt auch für die Ein- und Ausgabe. Im Gegensatz zu Desktop-Anwendungen sollen Ein- und Ausgabe-Instruktionen keinen Sonderstatus einnehmen, sondern die Tasks sollen sich selber um ihre Ein- und Ausgabe kümmern können. Das passt gut zum vorhergehenden Aufzählungspunkt und verringert den Aufwand von Ein-/Ausgabe-Operationen.

  Beispiel: Sei switch eine auf den Speicher abgebildete (*memory mapped*) Ein-/Ausgabe-Adresse eines Schalters, der von einem Programm überprüft werden soll. Wir können einfach eine Instruktion der Form

  ```
  load register,switch
  ```

  verwenden, um den Schalter abzufragen. Somit ist kein Aufwand für einen Betriebssystemaufruf notwendig, der zuerst aufwendig den Task-Kontext (Registerinhalte usw.) sichern und wiederherstellen müsste.
- **Unterbrechungen** (*interrupts*) können von jedem Prozess verwendet werden. Bei Desktop-Anwendung würde es die Verlässlichkeit des Systems stark einschränken, wenn jeder Prozess direkt Interrupts verwenden dürfte. Da eingebettete Anwendungen als gründlich getestet gelten und Schutzmechanismen nicht notwendig sind, und da die Ansteuerung von einer Vielzahl von Geräten benötigt wird, ist es möglich, durch Interrupts direkt Tasks zu starten oder anzuhalten (z.B. durch das Ablegen der Tasks in der Interruptvektor-Adresstabelle). Das ist wesentlich effizienter als die Verwendung von Betriebssystemaufrufen. Allerdings kann das Zusammensetzen der Komponenten des Betriebssystems komplexer werden: Wenn eine Task fest an einen bestimmten Interrupt gebunden ist, kann es schwierig

sein, eine andere Task hinzuzufügen, die auch von einem Ereignis gestartet werden soll.

- Viele eingebettete Systeme sind Echtzeit-Systeme. Daher müssen die in diesen Fällen verwendeten Betriebssysteme **Echtzeitbetriebssysteme** (*Real-Time Operating Systems* (RTOS)) sein.

4.3.2 Echtzeitbetriebssysteme

Definition: (Ein) *„Echtzeitbetriebssystem ist ein Betriebssystem, das die Konstruktion eines Echtzeit-Systems erlaubt“* [Takada, 2001].

Welche Eigenschaften werden benötigt, damit ein Betriebssystem ein Echtzeitbetriebssystem wird? Die folgenden Punkte sind drei Hauptanforderungen an Echtzeitbetriebssysteme[3]:

- **Das Zeitverhalten des Betriebssystems muss vorhersagbar sein.** Für jeden Betriebssystemaufruf muss eine obere Schranke für die Laufzeit garantiert werden. In der Praxis gibt es verschiedene Stufen von Vorhersagbarkeit. Beispielsweise kann es Mengen von Betriebssystemaufrufen geben, für die eine solche obere Schranke existiert und bei denen es keine großen Abweichungen bei der Ausführungszeit gibt. Aufrufe wie „bestimme die aktuelle Uhrzeit“ könnten in diese Kategorie fallen. Bei anderen Aufrufen gibt es sehr große Laufzeitunterschiede. Ein Aufruf wie „reserviere mir 4MB Speicherplatz“ könnte in diese zweite Kategorie fallen. Insbesondere müssen die Scheduling-Entscheidungen des Betriebssystems deterministisch sein (Standard-Java erfüllt dieses Kriterium nicht, da beim Starten von ausführbaren *threads* keine Reihenfolge angegeben werden kann). Ein weiterer erwähnenswerter Fall ist die sogenannte *Garbage Collection*. Im Zusammenhang mit Java wurden verschiedene Versuche unternommen, eine zeitlich vorhersagbare *Garbage Collection* zu implementieren (s. Seite 64).

  Es kann auch vorkommen, dass Interrupts zu bestimmten Zeiten deaktiviert werden müsssen, um zu vermeiden, dass sich zwei Tasks bei der Ausführung behindern. Dies ist eine sehr einfache Möglichkeit, gegenseitigen Ausschluss auf einem Ein-Prozessor-System zu erreichen. Die Zeitdauer, während der die Interrupts abgeschaltet sind, sollte dabei kurz gehalten werden, um unvorhersagbare Wartezeiten beim Verarbeiten kritischer Ereignisse zu vermeiden.

  Wenn Echtzeitbetriebssysteme auch Dateisysteme implementieren, kann es notwendig sein, die Daten in zusammenhängenden Dateien (*contiguous fi-*

[3] Dieser Abschnitt verwendet zur Beschreibung von Echtzeitbetriebssystemen Informationen aus Hiroaki Takadas Tutorial [Takada, 2001] zu Echtzeitbetriebssystemen, das er auf der Asian South-Pacific Design Automation Conference (ASP-DAC) im Jahr 2001 gehalten hat.

les) zu speichern, um unvorhersagbare Bewegungen des Schreib-/Lesekopfs der Festplatte zu vermeiden.

- **Das Betriebssystem muss sich um die Zeitverwaltung und das Scheduling von Tasks kümmern**. Das Betriebssystem muss möglicherweise auch gegebene Zeitschranken (*Deadlines*) berücksichtigen, so dass geeignete Schedulingtechniken zum Einsatz kommen. Es gibt allerdings auch Fälle, in denen das Scheduling vollkommen *off line* durchgeführt wird. Dann muss das Betriebssystem nur in der Lage sein, Dienste zu bestimmten Zeiten und mit bestimmten Prioritäten zu starten.

  Das Betriebssystem muss einen Zeitdienst mit hoher Präzision zur Verfügung stellen. Solche Zeitdienste werden beispielsweise benötigt, um zwischen ursprünglichen Fehlern und Folgefehlern zu unterscheiden. Mit ihrer Hilfe kann man z.B. diejenigen Kraftwerke feststellen, die für einen Stromausfall wie den an der amerikanischen Ostküste im Jahr 2003 verantwortlich sind. Ähnliches gilt für die Ursachenforschung bei europäischen Stromausfällen. Zeitdienste und globale Synchronisation von Uhren werden detailliert im Buch von Kopetz [Kopetz, 1997] beschrieben.
- **Das Betriebssystem muss schnell sein**. Zusätzlich zur Vorhersagbarkeit muss das Betriebssystem in der Lage sein, Anwendungen mit sehr kurzen *Deadlines* (im Bereich von Bruchteilen von Sekunden) zu unterstützen.

Jedes Echtzeitbetriebssystem verfügt über einen sogenannten echtzeitfähigen Betriebssystem-**Kern** (*Kernel*). Dieser Kern verwaltet die Ressourcen des Systems, unter anderem den Prozessor, den Speicher und den System-*Timer*. Schutzmechanismen (ausser für Verlässlichkeit, Sicherheit und Authentifizierung) werden häufig nicht benötigt.

Es gibt zwei Arten von Echtzeitbetriebssystemen:

- **Allzweck-Echtzeitbetriebssysteme:** Bei diesen Betriebssystemen werden einige Treiber implizit als vorhanden vorausgesetzt. Diese Treiber werden in den Kernel eingebettet. Die Anwendungs- oder *Middleware*-Software wird auf dem *Application Programming Interface* implementiert, das für alle Anwendungen standardisiert ist (s. Abb. 4.24 links).

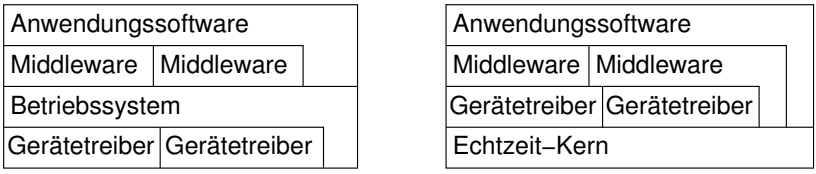


Abb. 4.24. Allzweck-Betriebssystem (links) gegenüber Echtzeitbetriebssystemkern (rechts)

- **Echtzeitkern-Betriebssysteme:** Da es kaum ein Standardgerät in eingebetteten Systemen gibt, werden Gerätetreiber nicht tief im Betriebssystemkern verankert, sondern außerhalb des Kerns implementiert. Nur die wirklich benötigten Treiber werden eingebunden. Anwendungen und *Middleware* können statt auf einem standardisierten *API* des Betriebssystems auf geeigneten Treibern implementiert werden (s. Abb. 4.24 rechts).

Die Hauptfunktionen des Betriebssystemkerns sind die Taskverwaltung, die Intertask-Synchronisation und -Kommunikation, die Zeit- und die Speicherverwaltung.

Während einige Echtzeitbetriebssysteme für allgemeine Anwendungen entwickelt wurden, konzentrieren sich andere auf bestimmte Anwendungsgebiete. Beispielsweise ist das OSEK/VDX-Betriebssytem auf den Automobilbereich spezialisiert. Wegen dieser Konzentration auf einen Bereich ist es ein sehr kompaktes Betriebssystem.

Einige Betriebssysteme bieten Standard-Schnittstellen in Form eines *API*, andere haben eigene, proprietäre *API*s. Beispielsweise sind einige Echtzeitbetriebssysteme mit den POSIX RT-Erweiterungen für UNIX [Harbour, 1993] kompatibel, andere mit dem OSEK/VDX Betriebssystem oder mit der ITRON-Spezifikation, die in Japan entwickelt wurde (ITRON ist ein bewährtes Echtzeitbetriebssytem, das beim Linken konfiguriert werden kann). Viele Echtzeitbetriebssysteme mit Echtzeitkern haben ein eigenes *API*.

Verfügbare Echtzeitbetriebssysteme können weiter in die folgenden drei Kategorien eingeteilt werden [Gupta, 2002]:

- **Schnelle proprietäre Kerne:** Nach Gupta sind *„diese Kerne für komplexe Systeme ungeeignet, da sie entworfen wurden, um schnell statt unter allen Umständen vorhersagbar zu sein“*. Beispiele sind etwa QNX, PDOS, VCOS, VTRX32 und VxWorks.
- **Echtzeit-Erweiterungen von Standard-Betriebssystemen:** Um von bequemen weit verbreiteten Standard-Betriebssystemen profitieren zu können, wurden hybride Systeme entwickelt. Dabei sollen sowohl Echtzeit-Tasks wie auch andere Tasks unterstützt werden. Für beide Klassen von Tasks sollen möglichst die üblichen Systemaufrufe nutzbar sein. Es gibt zwei Ansätze, um dieses Ziel zu erreichen:

  1. Beim ersten Ansatz werden einige Komponenten eines Standardbetriebssystems ausgetauscht. Insbesondere wird typischerweise der Scheduler ausgetauscht. Möglich ist beispielsweise der Einsatz eines Schedulers, welcher die Realzeiterweiterungen des POSIX-Standards (IEEE Standard 1003.1d) für eine Betriebssystemsschnittstelle unterstützt. Dieser Ansatz hat einige Vorteile: das System bietet eine Standard-Betriebssystemschnittstelle und es kann graphische Benutzerschnittstellen, Dateisysteme usw. verwenden. Außerdem werden Erweiterun-

gen zu Standard-Betriebssystemen schnell in der eingebetteten Welt verfügbar. Der Nachteil ist, dass damit in der Regel der Kern des Betriebssystems immer noch nicht realzeitfähig wird. So kann üblicherweise während der Bearbeitung eines Betriebssystemaufrufs eines Prozesses nicht auf die Bearbeitung einer Realzeit-Task umgeschaltet werden. Wenn allerdings nicht wirklich harte Realzeitbedingungen vorliegen, dann kann mit diesem Ansatz der größte Benutzungskomfort erreicht werden.

Nach Gupta ist der Versuch, eine Abwandlung eines Standard-Betriebssystems zu verwenden *"nicht der richtige Ansatz, da es zu viele zugrunde liegende nicht zutreffende Annahmen gibt, wie etwa die Optimierung für die durchschnittliche (statt für die maximale) Laufzeit, ... das Ignorieren der meisten, wenn nicht aller semantischer Informationen und ein Prozessor-Scheduling unabhängig von der Ressourcen-Allokation"*. In der Tat sind Abhängigkeiten zwischen den Tasks bei Standard-Betriebssystemen nicht sehr häufig anzutreffen und werden von diesen daher häufig komplett ignoriert. Bei eingebetteten Systemen dagegen ist die Situation völlig anders, da Abhängigkeiten zwischen den Tasks durchaus üblich sind und auch berücksichtigt werden sollten. Desweiteren werden das Scheduling und die Ressourcen-Allokation bei Standard-Betriebssystemen sehr selten kombiniert durchgeführt. Um Zeitbedingungen zu garantieren, ist es allerdings unerlässlich, das Scheduling und die Ressourcen-Allokation in einem Schritt durchzuführen.

2. Ein zweiter Ansatz ist radikaler: es wird ein zweites, kleines Betriebssystem erstellt, welches alle Echtzeit-Tasks ausführt. Das Standardbetriebssystem wird als eine Task des Echtzeitsystems ausgeführt (s. Abb. 4.25).

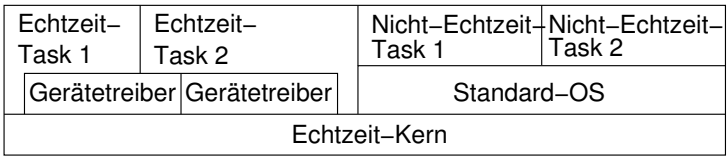


Abb. 4.25. Standard-Betriebssystem als eine Task eines Echtzeitsystems

Probleme mit dem Standard-Betriebssystem und seinen Nicht-Echtzeit-Tasks haben keinen negativen Einfluss auf die Echtzeittasks mehr. Das Standard-Betriebssystem kann sogar komplett abstürzen, ohne die Echzeittasks zu beeinflussen. Bei der Entwicklung des Echtzeitsystems können potenziell alle Nachteile einer reinen Erweiterung eines Standard-Betriebssystems vermieden werden. Andererseits, und dies wird schon aus Abb. 4.25 deutlich, leidet der Programmierkomfort. Es kann es Probleme mit Gerätetreibern geben, da das Standard-

Betriebssystem seine eigenen Gerätetreiber verwendet. Um Störungen zwischen diesen Treibern und denen der Echtzeit-Tasks zu vermeiden, kann es notwendig sein, die Geräte danach aufzuteilen, ob sie von Standard- oder von Echtzeit-Treibern und -Tasks bedient werden. Als weiterer Nachteil können die Echtzeit-Tasks die Fähigkeiten und Dienste des Standard-Betriebssystems nur eingeschränkt nutzen. Dabei gibt es Versuche, die Kluft zwischen den beiden Task-Typen zu überbrücken ohne die Echtzeitfähigkeiten zu verlieren. Ein weiterer Nachteil ist der hohe Aufwand für die Entwicklung eines neuen Betriebssystems. RT-Linux ist ein bekanntes Beispiel eines solchen Betriebssystems [Barabanov, 1997].

- Es gibt eine Anzahl von **Forschungssystemen**, die darauf abzielen, die oben genannten Nachteile zu vermeiden. Dazu gehören etwa Melody [Wedde und Lind, 1998] und, nach Gupta [Gupta, 2002] MARS, Spring, MARUTI, Arts, Hartos und DARK.

Takada [Takada, 2001] erwähnt als Forschungsthemen einen Speicherschutz, der mit geringem Aufwand gewährleistet werden kann, zeitlichen Schutz von Rechner-Ressourcen (um zu verhindern, dass Tasks länger für eine Berechnung brauchen als ursprünglich geplant), Echtzeitbetriebssysteme für *on chip*-Multiprozessoren (insbesondere für heterogene Multiprozessoren und mehrfädige (*multithreaded*) Prozessoren) sowie Unterstützung für kontinuierliche Medienströme und *Quality-of-Service*-Kontrolle.

Aufgrund des zu erwartenden Wachstums auf dem Gebiet der eingebetteten Systeme versuchen Hersteller von Standard-Betriebssystemen verstärkt, Variationen ihrer Produkte zu verkaufen und Marktanteile von traditionellen Herstellern wie Wind River Systems [Wind River Systems, 2003] zu übernehmen. Beispiele hierfür sind *Embedded Windows XP* und *Windows CE*.

4.4 *Middleware*

4.4.1 Echtzeit-Datenbanken

Datenbanken sind eine bequeme und strukturierte Methode zur Speicherung und zum Zugriff auf Informationen. Folglich stellen Datenbanken ein *API* zum Lesen und Schreiben von Informationen zur Verfügung. Eine Folge von Lese- und Schreib-Operationen wird **Transaktion** genannt. Transaktionen müssen unter Umständen aus verschiedenen Gründen abgebrochen werden: Es kann u.a. Hardwareprobleme, Blockaden (*Deadlocks*) oder Probleme mit parallelen Anfragen geben. Eine häufige Anforderung an Transaktionen ist, dass der Zustand der Datenbank nicht verändert werden soll, bis die Transaktion bis zum Ende ausgeführt wurde. Änderungen von Transaktionen werden daher

in der Regel solange nicht als endgültig betrachtet, bis sie **bestätigt** (*commited*) wurden. Die meisten Transaktionen müssen **atomar** (*atomic*) sein. Das bedeutet, dass das Endergebnis der Transaktion (also der neue Zustand der Datenbank) entweder der vollständigen Abarbeitung der Transaktion entspricht, oder dass überhaupt keine Änderung auftritt. Außerdem muss der Zustand der Datenbank nach jeder Transaktion **konsistent** sein. Konsistenz-Bedingungen können zum Beispiel beinhalten, dass Werte, die von ein und derselben Transaktion gelesen wurden, zueinander konsistent sind (also keinen Zustand beschreiben, der in der von der Datenbank modellierten Umgebung nicht vorkommen kann). Weiterhin darf für einen anderen Benutzer kein Zwischenzustand sichtbar sein, der auf die teilweise Abarbeitung einer Transaktion zurückzuführen ist. Die Transaktionen müssen also **isoliert** voneinander ausgeführt werden. Schließlich sollen die Ergebnisse einer Transaktion persistent gespeichert werden. Diese Eigenschaft heißt auch **Dauerhaftigkeit** (*durability*) einer Transaktion. Diese vier Eigenschaften zusammen werden als die ***ACID*-Eigenschaften** bezeichnet (von *Atomic, Consistent, Isolated, Durable*). Siehe dazu Kapitel 5 im Buch von Krishna und Shin [Krishna und Shin, 1997].

Bei einigen Datenbanken müssen weiche Echtzeitbedingungen beachtet werden. So sind etwa Zeitbedingungen bei Flugreservierungen weich. Im Gegensatz dazu kann es aber auch harte Echtzeitbedinungen geben. Die automatische Erkennung von Fußgängern in Automobil-Anwendungen oder die Zielerkennung in militärischen Anwendungen müssen harte Echtzeitbedingungen erfüllen. Dabei möchte man die zu erkennenden Muster gern in Datenbanken ablegen. Die oben genannten Anforderungen machen es sehr schwierig, harte Echtzeitbedingungen zu garantieren. Transaktionen können beispielsweise mehrere Male unterbrochen werden, bis sie dann endgültig ausgeführt und *commited* werden. Alle Datenbanken, die *Demand Paging* und Festplatten einsetzen, haben schwer vorhersagbare Plattenzugriffszeiten. Eine mögliche Lösung ist es, die Datenbank im Hauptspeicher zu halten. Eingebettete Datenbanken sind manchmal klein genug, um diesen Ansatz möglich zu machen. In anderen Fällen kann man die *ACID*-Anforderungen entschärfen. Weitere Informationen hierzu finden sich im Buch von Krishna und Shin.

4.4.2 Zugriff auf entfernte Objekte

Es gibt spezielle Softwarepakete, die den Zugriff auf entfernte Dienste vereinfachen. Ein Beispiel hierfür ist CORBA®(*Common Object Request Broker Architecture*) [Object Management Group (OMG), 2003]. Mit CORBA kann durch standardisierte Schnittstellen auf entfernte Objekte zugegriffen werden. Klienten kommunizieren mit lokalen *Stubs*, die den Zugriff auf entfernte Objekte nachbilden. Diese Klienten schicken sowohl Informationen über das Objekt, auf das zugegriffen werden soll, als auch optionale Parameter an den *Object Request Broker (ORB)* (s. Abb. 4.26). Der *ORB* bestimmt dann, wo sich das

referenzierte Objekt befindet und versendet Informationen gemäß einem standardisierten Protokoll, z.B. dem IIOP-Protokoll, dorthin, wo sich das Objekt befindet. Diese Information wird dann über ein sogenanntes *Skeleton* an das Objekt weitergeleitet und die möglicherweise angefragte Information wird wiederum über den *ORB* zurückgeschickt. Auf diese Weise werden Sprachen, die dem von-Neumannschen Berechnungsmodell folgen, um nachrichtenbasierte Kommunikation erweitert.

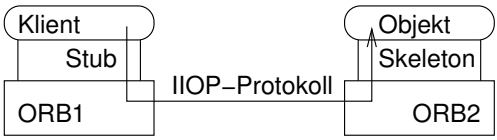


Abb. 4.26. Zugriff auf entfernte Objekte mittels CORBA

Standard-CORBA bietet nicht die Vorhersagbarkeit, die für Echtzeitanwendungen notwendig ist. Daher wurde ein eigener Echtzeit-CORBA-Standard (*RT-CORBA*) definiert [Object Management Group (OMG), 2002]. Eine wichtige Eigenschaft von RT-CORBA ist die *zeitliche Vorhersagbarkeit von Anfragen in einem System fester Prioritäten.* Dies beinhaltet *die Berücksichtigung von Thread-Prioritäten zwischen Client und Server, um Ressourcen-Konflikte aufzulösen* sowie die Beschränkung der Zeit für Operationsaufrufe. Ein besonderes Problem von Echtzeitsystemen ist die Tatsache, dass die Prioritäten von *Threads* unter Umständen nicht berücksichtigt werden, wenn ein *Thread* den exklusiven Zugriff auf eine Ressource erhält. Dieses Problem der Prioritätsumkehr (s. Seite 155) muss auch in RT-CORBA gelöst werden. RT-CORBA verfügt über Mechanismen, um die maximale Zeit, während der eine solche Prioritätsumkehr auftreten kann, nach oben zu beschränken. Außerdem kann RT-CORBA die Prioriäten von *Threads* organisieren. Diese Prioritäten sind unabhängig von den Prioritäten des zugrundeliegenden Betriebssystems, obwohl sie mit den Echtzeit-Erweiterungen des POSIX-Standards [Harbour, 1993] für Betriebssysteme kompatibel sind. Die *Thread*-Priorität der *Clients* kann zum Server weitergeleitet werden. Auch für Primitive, die exklusiven Zugriff auf Ressourcen realisieren, ist ein Prioritäten-Management vorhanden. In einer Implementierung von RT-CORBA muss das auf Seite 156 beschriebene Prioritäts-Vererbungs-Protokoll verfügbar sein. Durch eine Menge vorgefertigter *Threads* wird der zeitlich schwer vorherzusagende Aufwand, neue *Threads* zu erzeugen und alte zu entsorgen, umgangen.

Als Alternative zu CORBA erlaubt das *Message Passing Interface* (MPI), die Kommunikation zwischen verschiedenen Prozessoren. Um die Kommunikation im *MPI*-Stil in Echtzeitsystemen verfügbar zu machen, wurde eine Echtzeitversion von *MPI* (*MPI/RT*) definiert [Skjellum et al., 2002]. *MPI-RT* deckt nicht alle Besoderheiten ab, die von RT-CORBA untertützt werden, wie etwa

die *Thread*-Erzeugung. *MPI/RT* ist als optionale Zwischenschicht zwischen dem Betriebssystem und dem Standard (Nicht-Echtzeit) *MPI* zu sehen.

5

Implementierung eingebetteter Systeme: Hardware-/Software-Codesign

Wenn die Spezifikation eines Systems abgeschlossen ist, kann mit dem eigentlichen Entwurf begonnen werden. Dies ist auch im vereinfachten Informationsfluss in Abb. 5.1 dargestellt.

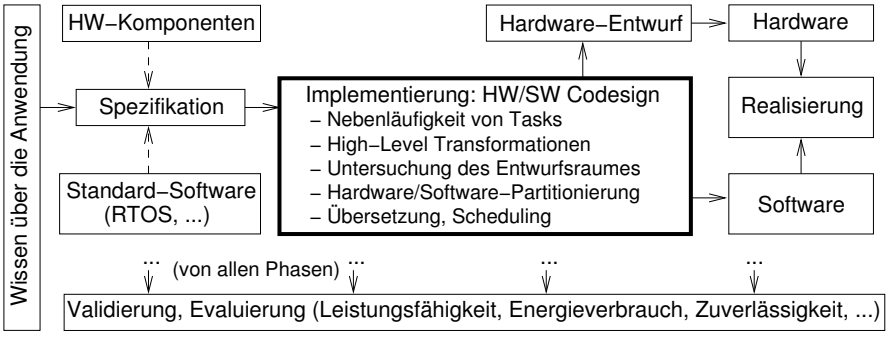


Abb. 5.1. Vereinfachter Informationsfluss beim Entwurf eingebetteter Systeme

Es ist eine charakteristische Eigenschaft des Entwurfs eingebetteter Systeme, dass sowohl Hard- als auch Softwareaspekte betrachtet werden müssen. Dementsprechend heißt diese Art des Entwurfs **Hardware-/Software-Codesign**. Das Ziel besteht darin, die richtige Mischung aus Hardware und Software zu bestimmen, um ein möglichst effizientes Produkt zu erhalten, das alle Anforderungen erfüllt. Aus diesem Grund können eingebettete Systeme nicht direkt aus einem Synthese-Prozess erzeugt werden, der nur die Spezifikation des Verhaltens berücksichtigt. Vielmehr müssen auch die verfügbaren Komponenten berücksichtigt werden. In diesem Kapitel sollen v.a. Optimierungen vorgestellt werden, die helfen, effiziente Hardware/Software-Systeme zu entwickeln. Es gibt noch einen zweiten Grund für die Notwendigkeit, verfügbare Komponenten von Anfang an zu betrachten: die Wiederverwendung von Kom-

ponenten ist unerlässlich, um der steigenden Komplexität eingebetteter Systeme und der immer kürzeren Zeitspanne bis zur Markteinführung (*Time to Market*) zu begegnen. Daraus entstand der Begriff des **Plattform-basierten Designs**:

„Eine Plattform ist eine Familie von Architekturen, die eine Menge von Einschränkungen erfüllen, welche die Wiederverwendung von Hard- und Software-Komponenten erlaubt. Eine Hardware-Plattform ist hierzu allerdings nicht ausreichend. Ein schneller und verlässlicher Entwurf unter Verwendung dieser Komponenten erfordert ein geeignetes Application Programming Interface (API), um die Plattform gemäß der Anwendungssoftware zu erweitern. Im Allgemeinen ist eine Plattform eine Abstraktionsebene, die viele mögliche Implementierungen (also Verfeinerungen in Richtung zu einer niedrigeren Abstraktionsebene) erlaubt. Der Plattform-basierte Entwurf ist ein sogenannter Meet-in-the-Middle-Ansatz: Der Top-Down-Entwurfsteil besteht darin, eine Instanz der abstrakten Plattform auf eine Instanz einer niedrigeren Abstraktionsebene abzubilden und dabei die Entwurfseinschränkungen mitzuführen.“ [Sangiovanni-Vincentelli, 2002].

Die Abbildung ist ein iterativer Prozess, der schrittweise von Programmen zur Leistungs-Abschätzung gesteuert wird. Abb. 5.2 [Herrera et al., 2003a] visualisiert diesen Ansatz.

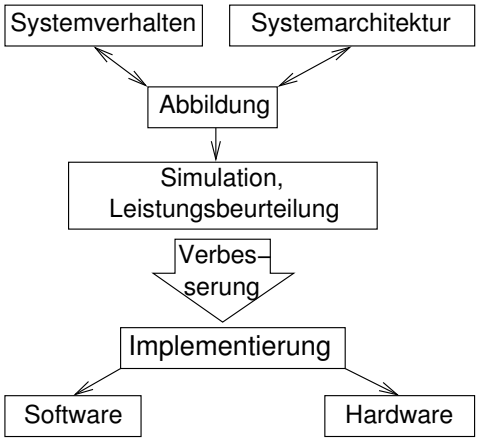


Abb. 5.2. Plattform-basierter Entwurf

Die Entwurfsschritte müssen die verfügbaren Plattformen berücksichtigen. Tatsächlich gibt es viel mehr solcher Entwurfsschritte, als wir in diesem Buch beschreiben können, und die Bezüge zu verfügbaren Plattformen sind nicht immer explizit dargestellt. Die hier vorgestellten Entwurfsschritte sind unter anderem:

- **Organisation der Nebenläufigkeit auf Taskebene:** Dieser Vorgang beschäftigt sich mit der Identifikation derjenigen Tasks, die in der endgültigen Implementierung des eingebetteten Systems vorhanden sein sollen. Diese Tasks müssen nicht zwangsläufig die Tasks aus der Spezifikation sein, da es gute Gründe für das Verschmelzen und Aufteilen von Tasks gibt (s. Abschnitt 5.5.1).
- ***High-Level*-Transformationen**: Es gibt viele optimierende sogenannte *High-Level*-Transformationen, die auf die Spezifikation, also auf einer hohen Abstraktionsebene, angewendet werden können. Beispielsweise können Schleifen so vertauscht werden, dass Speicherzugriffe ein lokaleres Verhalten haben. Gleitkomma-Arithmetik kann häufig ohne signifikanten Genauigkeitsverlust durch Festkomma-Operationen ersetzt werden. Diese *High-Level*-Transformationen werden üblicherweise nicht von den verwendeten Compilern unterstützt und müssen daher vor der Übersetzung manuell durchgeführt werden.
- **Hardware-/Software-Partitionierung:** Wir gehen im Allgemeinen davon aus, dass einige Funktionen aufgrund der steigenden Anforderungen an die Rechenleistung von spezieller Hardware ausgeführt werden müssen [De Man, 2002]. Die Hardware-/Software-Partitionierung bestimmt eine Zuordnung von Funktionen entweder zu Hardware oder zu Software.
- **Übersetzung (*Compilation*):** Diejenigen Teile der Spezifikation, die auf Software abgebildet werden, müssen übersetzt werden. Die Effizienz des erzeugten Codes kann verbessert werden, wenn der Compiler zusätzliches Wissen über den verwendeten Prozessor (und möglicherweise auch die eingesetzten Speicher) hat. Aus diesem Grund gibt es spezielle Compiler für eingebettete Systeme, welche die Eigenschaften der zugrundeliegenden Hardware berücksichtigen.
- **Scheduling:** Das Scheduling (die Abbildung von Operationen auf Anfangszeiten) muss in verschiedenen Zusammenhängen durchgeführt werden. Ein ungefähres Schedule muss während der Hardware-/Software-Partitionierung, während der Organisation der Nebenläufigkeit der Tasks und möglicherweise während der Übersetzung erzeugt werden. Ein genaues Schedule kann für den endgültigen Code generiert werden.
- **Untersuchung des Entwurfsraums (*Design Space Exploration*)**: Meistens erfüllen mehrere Entwürfe die Spezifikationen. Die Untersuchung des Entwurfsraumes analysiert die Menge der möglichen Entwürfe. Aus denjenigen Entwürfen, welche die Spezifikation erfüllen, muss einer ausgewählt werden.

Konkrete Entwurfsabläufe können diese Schritte in unterschiedlichen Reihenfolgen durchführen. Auch die Schritte selbst sind teilweise verschieden.

5.1 Organisation der Nebenläufigkeit auf Task-Ebene

Wie auf Seite 57 erwähnt, ist die **Granularität** des Taskgraphen eine der wichtigsten Eigenschaften. Selbst bei hierarchischen Taskgraphen kann es sinnvoll sein, die Granularität der Knoten zu verändern. Die während der Spezifikation vorgenommene Partitionierung in Tasks oder Prozesse zielt nicht zwangsläufig auf die effizienteste Implementierung. Während der Spezifikationsphase ist eine klare Aufgabentrennung und ein sauberes Softwaremodell wichtiger als Gedanken an eine möglichst effiziente Implementierung. Daher wird es nicht unbedingt eine eins-zu-eins-Beziehung zwischen den Tasks in der Spezifikation und denen in der endgültigen Implementierung geben. Eine Umgruppierung von Tasks kann durchaus sinnvoll sein. Erreicht wird sie durch das Verschmelzen und Aufteilen von Tasks.

Tasks können verschmolzen werden, wenn eine Task $\mathsf{T_j}$ nur einen direkten Vorgänger $\mathsf{T_i}$ hat (s. Abb. 5.3 mit $\mathsf{T_i} = \mathsf{T_3}$ und $\mathsf{T_j} = \mathsf{T_4}$). Diese Umformung kann zu einem verringerten Aufwand bei Kontextwechseln führen, wenn die betroffenen Knoten in Software implementiert werden. Im Allgemeinen kann das Verschmelzen zu erhöhtem Optimierungspotential führen.

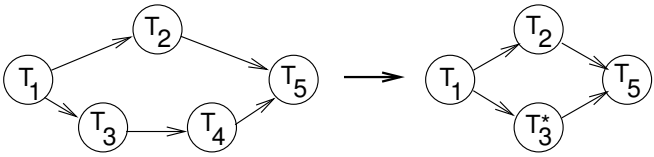


Abb. 5.3. Verschmelzen von Tasks

Andererseits kann das Aufteilen von Task-Knoten aus folgenden Gründen vorteilhaft sein:

Tasks können Ressourcen (wie etwa eine große Menge Speicher) anfordern und in Benutzung haben, während sie etwa auf eine Eingabe warten. Um die Ausnutzung der Ressourcen zu maximieren, kann es vorteilhaft sein, die Reservierung von Ressourcen auf die Zeitintervalle zu beschränken, in denen diese tatsächlich verwendet werden. In Abb. 5.4 nehmen wir an, dass Task $\mathsf{T_2}$ während ihrer Abarbeitung auf eine Eingabe wartet. In der ursprünglichen Version kann die Ausführung von Task $\mathsf{T_2}$ erst dann gestartet werden, wenn diese Eingabe verfügbar ist. Der Knoten kann nun in die Teilknoten $\mathsf{T_2^*}$ und $\mathsf{T_2^{**}}$ aufgeteilt werden, so dass die Eingabe nur für die Ausführung von $\mathsf{T_2^{**}}$ benötigt wird. Somit kann $\mathsf{T_2^*}$ bereits früher gestartet werden, was zu mehr Möglichkeiten beim Scheduling führt. Dies kann wiederum zu einer verbesserten Ausnutzung von Ressourcen führen und somit möglicherweise das Einhalten einer bestimmten Deadline ermöglichen. Es kann auch einen Einfluss auf die benötigte Datenspeichermenge haben, da $\mathsf{T_2^*}$ Teile des reservierten

Speichers vor der Beendigung freigeben kann. Dieser Speicher kann dann von anderen Tasks verwendet werden, während Task T_2^{**} auf ihre Eingabe wartet.

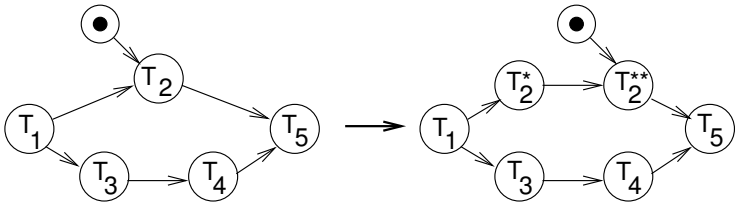


Abb. 5.4. Aufteilen von Tasks

Man könnte argumentieren, dass alle Tasks solche Ressourcen wie große Speicherblöcke immer freigeben sollten, bevor sie z.B. auf Eingaben warten. Die Lesbarkeit der ursprünglichen Spezifikation kann allerdings darunter leiden, wenn man solche Implementierungsfragen bereits in einer frühen Entwurfsphase berücksichtigt.

Recht komplexe Transformationen der Spezifikationen können mit einer Petrinetz-basierten Technik durchgeführt werden, die von [Cortadella et al., 2000] beschrieben wurde. Sie beginnt mit einer Spezifikation, die aus einer Menge von Tasks besteht, die in der Sprache *FlowC* beschrieben werden. FlowC erweitert C um Prozessköpfe und Intertask-Kommunikation in Form von READ- und WRITE-Funktionsaufrufen. Abbildung 5.5 zeigt eine Spezifikation in FlowC.

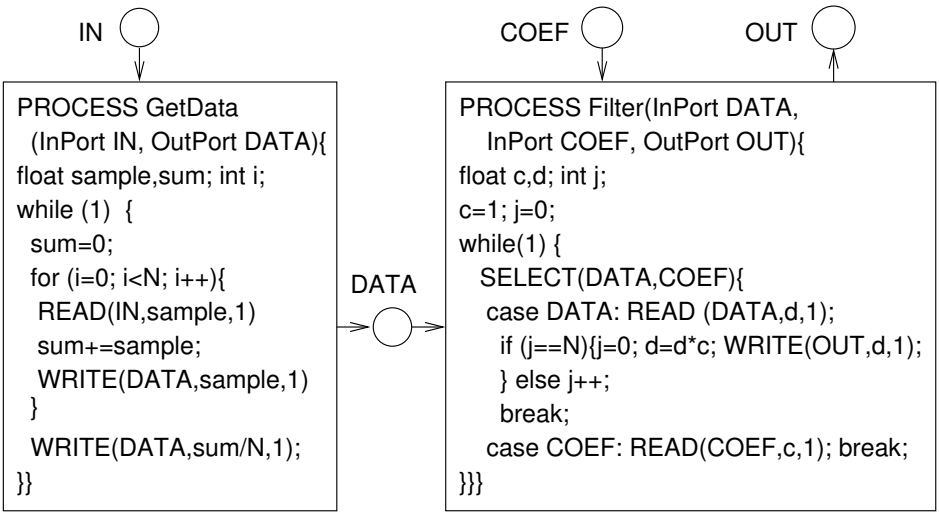


Abb. 5.5. Systemspezifikation

Das Beispiel verwendet die Eingabe-Ports IN und COEF sowie den Ausgabe-Port OUT. Eine direkte Interprozess-Kommunikation wird durch den gepufferten Kanal DATA realisiert. Die Task GetData liest Daten aus der Umgebung und schickt sie an den DATA-Kanal. Wenn N Werte gesendet wurden, wird ihr Durchschnitt berechnet und auch über den Kanal gesendet. Task Filter liest N Werte aus dem Kanal (und ignoriert sie) und liest danach den Mittelwert, multipliziert diesen mit c (c wird aus dem Port COEF gelesen) und schreibt das Ergebnis auf den Ausgabeport OUT. Der dritte Parameter der READ- und WRITE-Funktionsaufrufe ist die Anzahl zu lesender und zu schreibender Werte. Aufrufe von READ blockieren, Aufrufe von WRITE blockieren nur, wenn die Anzahl von Elementen im Kanal eine vordefinierte Anzahl überschreitet. Die SELECT-Anweisung hat die gleiche Semantik wie die entsprechende ADA-Instruktion (s. Seite 62): die Ausführung der Task wird suspendiert, bis eine Eingabe an einem der Eingabeports eintrifft. Dieses Beispiel erfüllt alle Kriterien für das Aufteilen von Tasks, die in Zusanmenhang mit Abb. 5.4 erwähnt wurden. Beide Tasks werden auf Eingaben warten und dabei Ressourcen blockieren. Die Effizienz kann durch eine Umstrukturierung der Tasks verbessert werden. Die in Abb. 5.4 gezeigte einfache Aufteilung ist allerdings nicht ausreichend. Die von Cortadella et al. vorgeschlagene Technik ist umfassender. Mit ihrer Hilfe werden FlowC-Programme in (erweiterte) Petrinetze umgewandelt. Die einzelnen Petrinetze für jede Task werden dann zu einem einzigen Petrinetz zusammengefasst. Mit Hilfe von theoretischen Petrinetz-Methoden werden dann neue Tasks erzeugt. Abb. 5.6 zeigt eine mögliche neue Struktur.

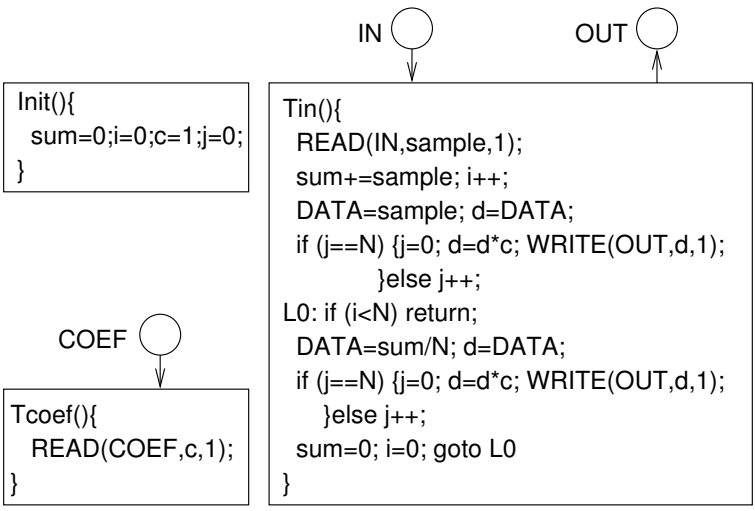


Abb. 5.6. Erzeugte Software-Tasks

In dieser neuen Struktur erledigt eine Task die gesamte Initialisierung. Außerdem gibt es eine Task für jeden Eingabeport. Eine effiziente Implementierung würde bei jeder neuen Eingabe einen eigenen Interrupt für jeden Port erzeugen. Die Tasks könnten dann direkt von diesen Interrupts gestartet werden und es wären hierfür keine Betriebssystemaufrufe mehr notwendig. Die Kommunikation kann einfach über eine gemeinsam genutzte globale Variable realisiert werden (unter der Annahme eines gemeinsamen Adressbereichs für alle Tasks). Das resultierende Betriebssystem wäre sehr klein, wenn es überhaupt noch benötigt wird.

Der Code für Task Tin ist in Abb. 5.6 gezeigt. Er wurde aus der Petrinetz-basierten Intertask-Optimierung erzeugt. Er sollte weiter von Intratask-Optimierungen verbessert werden, da die Abfrage in der ersten if-Instruktion immer false ergibt (j ist in diesem Fall gleich i-1, und i und j werden auf 0 zurückgesetzt wenn i den Wert N erreicht). Für das zweite if ist das Ergebnis immer true, da dieser Punkt des Programms nur erreicht wird, wenn i gleich N ist und wenn i gleich j ist wenn das Label L0 erreicht wird. Auch die Anzahl verwendeter Variablen kann reduziert werden. Eine optimierte Version von Tin könnte so aussehen:

```
Tin () {
  READ (IN, sample, 1);
  sum += sample;  i++;
  d = sample;
L0: if (i < N) return;
  d = sum/N;
  d = d*c;  WRITE(OUT,d,1);
  sum = 0;  i = 0;
  return;
}
```

Diese optimierte Version von Tin könnte von einem sehr geschickten Compiler erzeugt werden. Leider kann kaum einer der verfügbaren Compiler eine solche Optimierung automatisch durchführen. Trotzdem zeigt das Beispiel die benötigten Transformationen, um „gute“ Taskstrukturen zu erhalten. Weitere Details über die Erzeugung von Tasks findet man bei Cortadella et al. [Cortadella et al., 2000].

Optimierungen, die der eben gezeigten ähneln, werden im Buch von Thoen und Catthoor [Thoen und Catthoor, 2000] beschrieben. Eine Liste der Publikationen des IMEC über den Bereich der Task-Organisation kann von der IMEC-Webseite [IMEC Desics group, 2003] heruntergeladen werden.

5.2 *High-Level*-Optimierungen

Es gibt viele *High-Level*-Optimierungen, die potentiell die Effizienz eingebetteter Software verbessern können.

5.2.1 Umwandlung von Gleitkomma- in Festkomma-Darstellung

Die Umwandlung von Gleitkomma- in Festkomma-Darstellungen ist eine häufig verwendete Technik. Der Grund hierfür liegt in der Spezifikation weitverbreiteter Signalverarbeitungs-Algorithmen (etwa MPEG-2 oder MPEG-4) in der Programmiersprache C: diese Spezifikationen verwenden Gleitkomma-Datentypen. Es bleibt dem Entwickler überlassen, effiziente Implementierungen dieser Standards zu erzeugen.

Bei vielen Signalverarbeitungs-Anwendungen ist es möglich, diese Ersetzung von Gleitkomma- durch Festkomma-Datentypen durchzuführen (s. Seite 119). Die Vorteile können beachtlich sein. Beispielsweise wurde beim MPEG-2 Videokompressions-Algorithmus eine Verringerung der benötigten Zyklen um 75% und eine Energiereduktion um 76% erreicht [Hüls, 2002]. Allerdings verliert man in der Regel an Genauigkeit. Genauer gesagt muss man einen Kompromiss finden zwischen den Kosten der Implementierung und der Qualität des Algorithmus. Letztere kann beispielsweise über das Verhältnis von Nutz- zu Störsignalen (*Signal-to-Noise-Ratio* (SNR)) bestimmt werden. Bei kleinen Wortlängen kann die Qualität sehr negativ beeinflusst werden. Gleitkomma-Datentypen können also durch Festkomma-Datentypen ersetzt werden, wenn man den Qualitätsverlust mit analysiert. Die Ersetzung wurde ursprünglich manuell durchgeführt, was allerdings ein aufwendiger und fehleranfälliger Prozess ist.

Aus diesem Grund wurde versucht, diese Ersetzung zu automatisieren bzw. teilweise zu automatisieren. Eines der bekanntesten Werkzeuge hierfür ist FRIDGE (*fixed-point programing design environment*) [Willems et al., 1997], [Keding et al., 1998]. FRIDGE ist kommerziell als Teil der *Synopsys System Studio Tool-Suite* erhältlich [Synopsys, 2005].

Bei FRIDGE beginnt der Entwurfsprozess mit einem in C beschriebenen Algorithmus, der Gleitkommazahlen enthält. Dieser Algorithmus wird dann in eine Darstellung in der Sprache **Fixed-C** umgewandelt. Fixed-C ist eine Untermenge von C++ und stellt mithilfe von C++-Typdefinitionen zwei Festkomma-Datentypen, `fixed` und `Fixed`, zur Verfügung. Festkomma-Datentypen können wie andere Variablen deklariert werden. Die folgende Beschreibung deklariert einen Festkomma-Datentyp als skalare Variable, als Zeiger und als Feld von Festkomma-Datentypen:

```
fixed a,*b,c[8]
```

Die Parameter der Festkomma-Darstellung werden dabei nicht festgelegt.

Die Parameter der Festkomma-Datentypen können in der Deklaration oder bei der Zuweisung angegeben werden:

```
a=fixed(5,4,s,wt,*b)
```

Diese Zuweisung legt die Wortlänge von a auf 5 Bit fest, die Nachkomma-Wortlänge beträgt 4, es gibt ein Vorzeichen-Bit s, Überläufe werden durch Wrap-Around (w) behandelt, und die Rundung geschieht durch Abschneiden (*truncation* (t)). *b beschreibt die Quelle der Zuweisung. Die Parameter von Festkomma-Zahlen, die in einer Zuweisung gelesen werden, werden bei der Zuweisung zu diesen Variablen bestimmt. Der Datentyp Fixed ist fixed sehr ähnlich, bietet aber zusätzlich eine Konsistenz-Überprüfung zwischen den Parametern, die in der Deklaration verwendet wurden und denen, die bei der Zuweisung angegeben wurden. Für jede Variablenzuweisung können die Parameter (also auch die Wortlänge) unterschiedlich sein. Informationen über die Parameter können zum Original-C-Programm hinzugefügt werden, bevor die Anwendung simuliert wird. Die Simulation stellt Wertebereiche für jede Zuweisung zur Verfügung. Basierend auf dieser Information fügt FRIDGE Parameter-Informationen zu allen Zuweisungen hinzu. FRIDGE kann solche Informationen auch aus dem Kontext herauslesen. Beispielsweise ist der Maximalwert einer Addition die Summe der Argumente. Zusätzliche Parameter-Informationen können entweder aufgrund von Simulationsergebnissen oder aus *Worst-Case*-Analysen generiert werden. Da es auf einer Simulation basiert, liefert FRIDGE nicht unbedingt die *Worst-Case*-Werte, die eine formale Analyse ergeben würde. Das erzeugte C++-Programm wird simuliert, um den Qualitätsverlust zu beurteilen. Die Synopsys-Version von FRIDGE verwendet SystemC-Festpunkt-Datentypen, um die erzeugten Informationen über die Datentyp-Parameter darzustellen. Folglich kann auch SystemC verwendet werden, um Festkomma-Datentypen zu simulieren.

Analysen zur Beurteilung des Zusammenhangs zwischen zusätzlichem Störsignal und benötigter Wortlänge wurden von [Shi und Brodersen, 2003] sowie von [Menard und Sentieys, 2002] vorgestellt.

5.2.2 Einfache Schleifentransformationen

Es gibt einige Schleifentransformationen, die auf Spezifikationen angewendet werden können. Zu den Standard-Schleifentransformationen gehören:

- **Schleifen vertauschen** (*loop permutation*): Nach dem C-Standard von Kernighan und Ritchie [Kernighan und Ritchie, 1988] werden zweidimensionale Felder wie in Abb. 5.7 im Speicher angeordnet. Benachbarte Indexwerte des zweiten Index werden in einem zusammenhängenden Speicherblock abgelegt. Diese Anordnung heißt *Row Major Order* [Muchnick, 1997]. In FORTRAN werden solche Felder anders abgespeichert: hier werden benachbarte Werte des ersten Indexwertes auf einen zusammenhängenden

Speicherblock abgebildet (*Column Major Order*). Publikationen, die Optimierungen für FORTRAN beschreiben, können daher leicht verwirrend wirken.

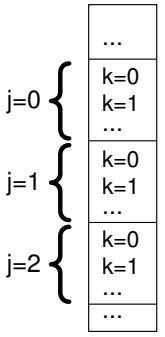


Abb. 5.7. Speicheranordnung für ein zweidimensionales Feld p[j][k] in C

Bei einer *Row Major*-Anordnung ist es normalerweise von Vorteil, die Schleifen so anzuordnen, dass der letzte Index mit der innersten Schleife adressiert wird. Eine entsprechende Vertauschung von Schleifen zeigt das folgende Beispiel:

```
for (k=0; k<=m; k++)        for (j=0; j<=n; j++)
  for (j=0; j<=n; j++)  ⇒     for (k=0; k<=m; k++)
    p[j][k] = ...               p[j][k] = ...
```

Solch eine Permutation kann die Wiederverwendung von Feldelementen im Cache stark verbessern, da die nächste Iteration der Schleife auf die nachfolgende Speicheradresse zugreift. Caches sind in der Regel so aufgebaut, dass ein Zugriff auf aufeinanderfolgende Adressen wesentlich schneller möglich ist als ein Zugriff auf Adressen, die weit von der vorherigen Zugriffsadresse entfernt sind.

- **Schleifen verschmelzen, Schleifen aufspalten** (*Loop Fusion, Loop Fission*): Es gibt Fälle, in denen zwei getrennte Schleifen verschmolzen werden können. Andererseits kann eine einzelne Schleife in zwei Schleifen aufgespalten werden. Hier ein Beispiel:

```
for(j=0; j<=n; j++)        for (j=0; j<=n; j++)
  p[j]= ... ;                {p[j]= ... ;
for (j=0; j<=n; j++) ⇔       p[j]= p[j] + ...}
  p[j]= p[j] + ...
```

Die linke Version kann vorteilhaft sein, wenn der Zielprozessor eine Hardware-Schleifen-Instruktion (*Zero-Overhead Loop Instruction*) besitzt, die für kleine Schleifen ohne zusätzlichen Verwaltungsaufwand z.B. für die Schleifengrenzen auskommt. Die rechte Version kann zu einem besseren

Cache-Verhalten führen, da die Zugriffe auf Feld p eine höhere Lokalität aufweisen. Außerdem wird das Potential für parallele Berechnungen innerhalb des Schleifenkörpers erhöht. Wie bei vielen anderen Transformationen ist es schwierig zu entscheiden, welche Variante letztendlich zum besseren Code führt.

- **Schleifen abrollen** (*Loop Unrolling*): Das Abrollen von Schleifen ist eine Standard-Transformation, die mehrere Instanzen des Schleifenkörpers erzeugt. Im folgenden Beispiel wird die Schleife einmal abgerollt:

```
for (j=0; j<=n; j++)        for (j=0; j<=n; j+=2)
  p[j]= ... ;          ⇒      {p[j]= ... ;
                               p[j+1]= ...}
```

  Die Anzahl der erzeugten Schleifenkörper heißt **Abroll-Faktor** (*Unrolling Factor*). Es sind durchaus größere Abrollfaktoren als zwei möglich. Das Abrollen verringert den Aufwand für die Verwaltung der Schleife (weniger Sprungbefehle pro Ausführung des ursprünglichen Schleifenkörpers) und verbessert aus diesem Grunde normalerweise die Geschwindigkeit. Im Extremfall können Schleifen komplett abgerollt werden, wodurch der Aufwand für Kontrollfluss und Sprünge ganz wegfällt. Andererseits erhöht das Abrollen aber die Codegröße. In der Regel werden nur Schleifen mit einer konstanten Anzahl von Ausführungen abgerollt.

5.2.3 Kachel-/Blockweise Verarbeitung von Schleifen

Es wurde bereits erwähnt, dass sich die Geschwindigkeit von Speichern langsamer entwickelt als die Geschwindigkeit von Prozessoren. Da kleine Speicher schneller sind als große (s. Seite 129), kann es sinnvoll sein, eine Speicherhierarchie einzusetzen. Mögliche Varianten von „kleinen“ Speichern sind Caches oder Scratchpad-Speicher. Eine häufige Wiederverwendung der in diesen kleinen Speichern abgelegten Informationen ist notwendig, da die Speicherhierarchie sonst nicht effizient ausgenutzt werden kann.

Wiederverwendungs-Effekte kann man anhand des folgenden Beispiels erkennen. Wir betrachten die Multiplikation zweier Felder der Größe $N \times N$ [Lam et al., 1991]:

```
for (i=1; i<=N; i++)
  for(k=1; k<=N; k++){
    r=X[i,k]; /* einem Register zugeordnet */
    for (j=1; j<=N; j++)
      Z[i,j] += r* Y[k,j]
  }
```

Betrachten wir nun das Zugriffsmuster für diesen Code: Das selbe Element X[i,k] wird von allen Iterationen der innersten Schleife verwendet. Ein Com-

piler wird üblicherweise in der Lage sein, dieses Element in einem Register abzulegen und es von dort in jedem Schleifendurchlauf zu verwenden. Wir nehmen an, dass die Felder in *Row Major Order* abgelegt sind, wie das in C üblich ist. Das bedeutet, dass Feldelemente mit benachbarten Zeilen (rechtesten Indexwerten) in aufeinanderfolgenden Speicheradressen abgelegt werden. Folglich werden in der innersten Schleife benachbarte Elemente von Z und Y geladen. Diese Eigenschaft ist vorteilhaft, wenn das Speichersystem das sogenannte *prefetching* erlaubt. Beim *prefetching* wird beim Laden eines Wortes automatisch auch das Laden des nachfolgenden Wortes angestoßen. Abbildung 5.8 zeigt das Zugriffsmuster für diesen Code.

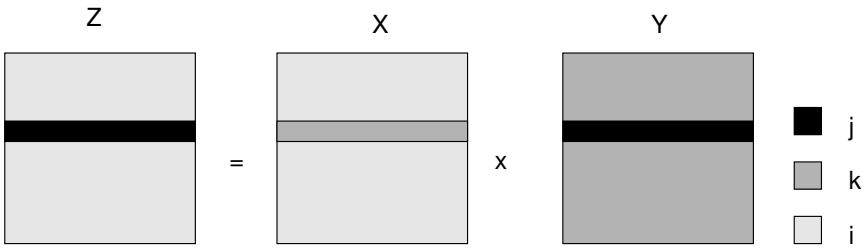


Abb. 5.8. Zugriffsmuster für nicht blockweise Matrixmultiplikation

Während eines Durchlaufs der innersten Schleife über die Laufvariable j werden die schwarzen Flächen der Felder Z und Y verwendet (und folglich in den Cache geladen). Ob diese Informationen beim nächsten Durchlauf der mittleren oder der äußeren Schleife über die beiden anderen Laufvariablen noch im Cache sind, hängt von der Cachegröße ab. Im schlechtesten Fall (wenn N groß oder der Cache klein ist) müssen die Informationen bei jeder Ausführung der innersten Schleife neu in den Cache geladen werden und können folglich nie wiederverwendet werden. Die Gesamtanzahl der Speicherzugriffe kann bis zu $2\ N^3$ (Zugriffe auf Z) + N^3 (Zugriffe auf Y) + N^2 (Zugriffe auf X) betragen.

Arbeiten im Bereich des wissenschaftlichen Rechnens haben zum Entwurf von **blockweisen** oder **gekachelten Zugriffs-Algorithmen** geführt [Xue, 2000], welche die **Lokalität der Speicherzugriffe** verbessern. Das folgende Beispiel zeigt eine gekachelte Version des obigen Algorithmus, und Abbildung 5.9 zeigt das zugehörige Zugriffsmuster.

```
for(kk=1; kk<= N; kk+=B)
  for (jj=1; jj<= N; jj+=B)
    for (i=1; i<= N; i++)
      for (k=kk; k<= min(kk+B-1,N); k++){
        r=X[i][k]; /* einem Register zugeordnet */
        for (j=jj; j<= min(jj+B-1, N); j++)
```

```
        Z[i][j] += r* Y[k][j]
    }
```

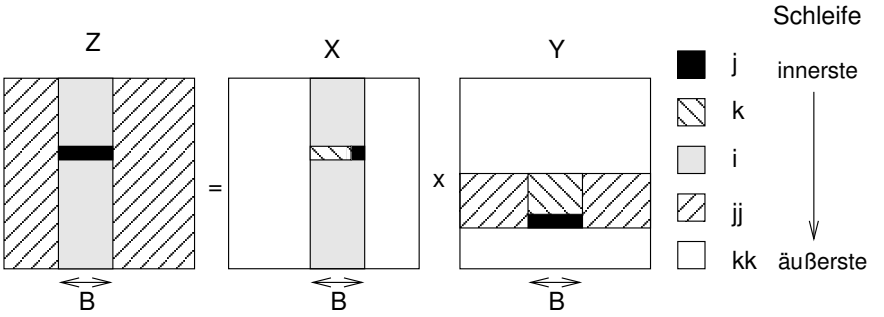


Abb. 5.9. Zugriffsmuster der Matrixmultiplikation bei blockweisem/gekacheltem Zugriff

Der Indexbereich der innersten Schleife wird verkürzt, so dass sie auf weniger Feldelemente zugreift (in der Abbildung schwarz dargestellt). Wenn der richtige *Blocking*-Faktor ausgewählt wurde, befinden sich die Elemente noch im Cache, wenn der nächste Durchlauf der innersten Schleife beginnt. Entsprechend sollte der Blocking-Faktor B so gewählt werden, dass die Elemente der innersten Schleife in den Cache passen. Insbesondere sollte er so gewählt werden, dass eine $B \times B$ große Teilmatrix von Y in den Cache passt. Das entspricht einem Wiederverwendungsfaktor von N für das Feld Y, da die Elemente in der Teilmatrix N mal während eines Durchlaufs der i-Schleife verwendet werden. Außerdem sollte ein Block von B Zeilenelementen von Z in den Cache passen. Dieser kann während der Ausführung der k-Schleife wiederverwendet werden, wodurch sich für Z ein Wiederverwendungsfaktor von B ergibt. Die Gesamtzahl von Speicherzugriffen reduziert sich somit auf maximal $2\,N^3/B$ (Zugriffe auf Z) + N^2 (Zugriffe auf Y) + N^2 (Zugriffe auf X). In der Praxis wird der Wiederverwendungsfaktor kleiner sein. Das Optimieren dieses Wiederverwendungsfaktors bedarf detaillierter Untersuchungen. Das ursprüngliche Ziel der Forschung in diesem Bereich war eine Verbesserung der Laufzeit durch den Einsatz von gekachelten Zugriffsmustern. Die Laufzeit einer Matrixmultiplikation konnte durch den Einsatz dieser Methoden um einen Faktor von 3 bis 4,3 verbessert werden [Lam et al., 1991]. Mit der steigenden Diskrepanz zwischen Speicher- und Prozessorgeschwindigkeiten wird sich das Potential dieser Optimierungstechnik noch weiter erhöhen. Durch gekachelte Speicherzugriffe kann auch der Energiebedarf des Speichersystems reduziert werden [Chung et al., 2001]. Allerdings kann durch die zusätzlichen Schleifen auch zusätzlicher Aufwand entstehen.

5.2.4 Aufteilen von Schleifen

Das Aufteilen von Schleifen (*Loop Splitting*) ist eine weitere Optimierung, die vor der eigentlichen Übersetzung eines Programms ausgeführt werden kann. Potentiell könnte diese Optimierung auch in den Compiler integriert werden.

Viele Bildverarbeitungsalgorithmen führen eine Art der Filterung durch. Dabei werden die Informationen eines bestimmten Bildpunktes (*Pixel*) sowie die Informationen einiger seiner Nachbarn analysiert. Die zugehörigen Berechnungen sind in der Regel immer die gleichen. Wenn sich ein Pixel aber in der Nähe des Bildrandes befindet, existieren eventuell nicht alle Nachbarpixel, und der Algorithmus muss an diesen Stellen entsprechend angepasst werden. In einer intuitiven Implementierung des Filter-Algorithmus könnte diese veränderte Behandlung durch eine Abfrage in der innersten Schleife realisiert werden. Eine effizientere Version des Algorithmus kann erzeugt werden, indem man die Schleifen so aufteilt, dass ein Schleifenkörper die regelmäßigen Fälle behandelt, während der andere die Sonderfälle realisiert. Abbildung 5.10 zeigt eine graphische Darstellung dieser Transformation.

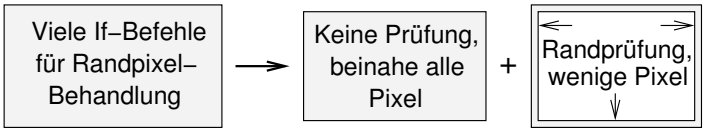


Abb. 5.10. Aufteilen einer Bildverarbeitung in einen regelmäßigen Fall und in Sonderfälle

Die manuelle Durchführung dieser Schleifenaufteilung ist sehr schwierig und fehleranfällig. Falk et al. [Falk und Marwedel, 2003] haben einen Algorithmus zum *Loop Nest Splitting* entwickelt, der die Optimierung auch für mehr als zwei Dimensionen automatisch durchführt. Das Verfahren basiert auf einer komplexen Analyse der Zugriffsmuster auf Feldelemente in den jeweiligen Schleifen. Optimierte Lösungen werden mit Hilfe eines genetischen Algorithmus erzeugt. Der folgende Code-Ausschnitt zeigt verschachtelte Schleifen aus der Bewegungsschätzung des MPEG-4-Standards:

```
for (z=0; z<20; z++)
  for (x=0; x<36; x++) {x1=4*x;
   for (y=0; y<49; y++) {y1=4*y;
    for (k=0; k<9; k++) {x2=x1+k-4;
     for (l=0; l<9; ) {y2=y1+l-4;
      for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
       for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
        if (x3<0 || 35<x3||y3<0||48<y3)
```

```
            then_block_1; else else_block_1;
           if (x4<0|| 35<x4||y4<0||48<y4)
            then_block_2; else else_block_2;
}}}}}}
```

Unter Verwendung von Falks Algorithmus werden diese Schleifen in die folgende Form umgewandelt:

```
for (z=0; z<20; z++)
  for (x=0; x<36; x++) {x1=4*x;
   for (y=0; y<49; y++)
    if (x>=10||y>=14)      // splitting-if
     for (; y<49; y++)
      for (k=0; k<9; k++)
       for (l=0; l<9;l++ )
        for (i=0; i<4; i++)
         for (j=0; j<4;j++) {
          then_block_1; then_block_2}
    else {y1=4*y;
     for (k=0; k<9; k++) {x2=x1+k-4;
      for (l=0; l<9; ) {y2=y1+l-4;
       for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
        for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
         if (0 || 35<x3 ||0|| 48<y3)
          then_block_1; else else_block_1;
         if (x4<0|| 35<x4||y4<0||48<y4)
          then_block_2; else else_block_2;
}}}}}}
```

Statt die komplizierten Abfragen in der innersten Schleife zu überprüfen, gibt es jetzt eine sogenannte *splitting-if*-Instruktion nach der dritten for-Schleife. Alle regulären Fälle werden im then-Fall dieser Anweisung behandelt. Der else-Teil behandelt die relativ kleine Zahl von Ausnahmefällen.

Abbildung 5.11 zeigt anhand von verschiedenen Anwendungen und Zielprozessoren, wieviele Ausführungszyklen durch die Anwendung des *Loop Nest Splitting* eingespart werden können.

Bei der Bewegungs-Schätzung kann die Ausführungszeit um 75% reduziert werden (auf ein Viertel der ursprünglichen Zeit). Offensichtlich kann man mit

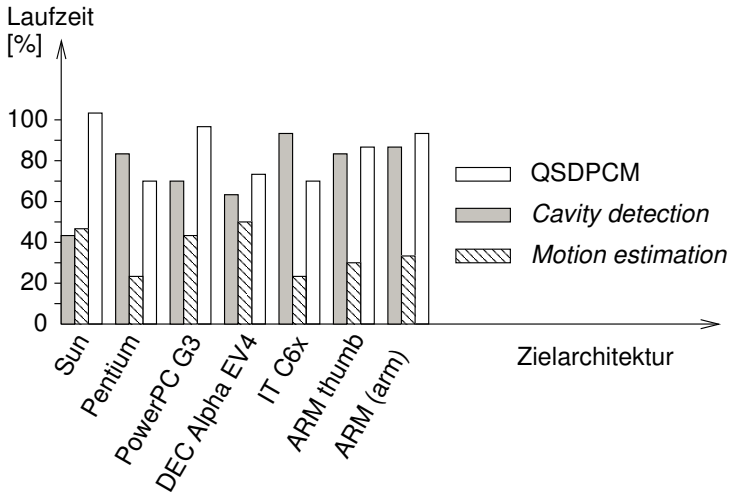


Abb. 5.11. Ergebnisse für *Loop Nest Splitting*

dem Verfahren beträchtliche Einsparungen erzielen. Dieses Potential sollte definitiv nicht außer Acht gelassen werden.

5.2.5 Falten von Feldern

Einige eingebettete Anwendungen, insbesondere aus dem Multimedia-Bereich, verwenden große Felder. Da der Speicherplatz in eingebetteten Systemen beschränkt ist, sollten Möglichkeiten zur Verringerung des Speicherbedarfs betrachtet werden. Abbildung 5.12 stellt die von fünf Feldern verwendeten Adressen als Funktion über der Zeit dar. Zu jedem Zeitpunkt wird nur eine Teilmenge der Feldelemente benötigt. Die maximale Anzahl von benötigten Elementen heißt **Adress-Referenz-Fenster** (*address reference window*) [De Greef et al., 1997a]. In Abb. 5.12 ist das Maximum durch einen Doppelpfeil dargestellt.

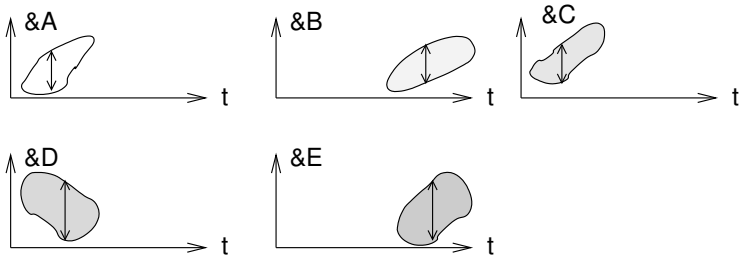


Abb. 5.12. Zugriffsmuster für Felder

Eine klassische Speicherzuordnung für Felder ist im linken Teil von Abb. 5.13 dargestellt. Jedes Feld belegt soviel Platz, wie es während der gesamten Ausführungszeit maximal belegt (unter der Annahme, es handele sich um globale Felder).

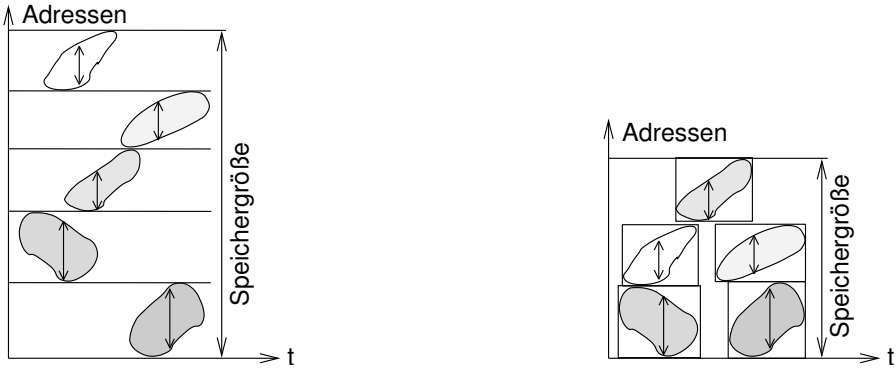


Abb. 5.13. Ungefaltetes (links) und mit *inter-array folding* gefaltetes Feld (rechts)

Eine der möglichen Verbesserungen, das sogenannte *inter-array folding*, ist im rechten Teil von Abb. 5.13 gezeigt. Felder, die während bestimmter überlappender Zeitintervalle nicht gleichzeitig gebraucht werden, können sich den gleichen Speicherplatz teilen. Eine zweite Verbesserung, das *intra-array folding* [De Greef et al., 1997b] ist in Abb. 5.14 dargestellt. Es nutzt aus, dass auch **innerhalb** eines Arrays nur eine bestimmte Anzahl von Elementen gleichzeitig benötigt wird. Speicherplatz kann somit durch komplexere Adressberechnungen eingespart werden.

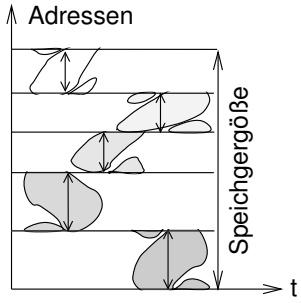


Abb. 5.14. Mittels *Intra-array folding* gefaltetes Feld

Die beiden Arten des Faltens können kombiniert werden.

Andere Formen von *High-Level*-Transformationen wurden von Chung, Benini und De Micheli [Chung et al., 2001], [Tan et al., 2003] analysiert. Es gibt viele weitere Beiträge zu diesem Thema aus dem Bereich des Compilerbaus.

Das *Function Inlining*[1] ersetzt den Aufruf von nicht-rekursiven Funktionen durch den Code der aufgerufenen Funktion. Herkömmliche *Inlining*-Techniken benötigen Informationen vom Benutzer oder Programmierer, um zu entscheiden, welche Funktionen für das *Inlining* in Betracht gezogen werden. Diese Transformation erhöht die Geschwindigkeit des Codes, kann aber auch eine Codevergrößerung zur Folge haben, was zu Problemen führen kann, wenn Systeme einschließlich des Speichers auf einem Chip integriert sind (sogenannte *systems on a chip* (SoC)). Bei SoCs ist die Größe des Instruktionsspeichers eine sehr kritische Größe. Daher muss es eine Möglichkeit geben, die Codegröße zu beschränken. Folglich braucht man eine automatische Analyse, die bestimmt, für welche Funktionen das *Inlining* durchgeführt wird, um diese Grenze nicht zu überschreiten. Bekannte Techniken wurden etwa von [Teich et al., 1999], [Leupers und Marwedel, 1999] und [Palkovic et al., 2002] vorgestellt. Diese Techniken können entweder in einen Compiler integriert werden oder als *Source-to-Source*-Transformation auf dem Quellcode vor dem Übersetzen durchgeführt werden.

5.3 Hardware-/Software-Partitionierung

5.3.1 Einleitung

Während des Entwurfsprozesses muss die Entscheidung getroffen werden, welche Teile der Spezifikation in Hardware und welche in Software, also auf Prozessoren, ausgeführt werden sollen. Dieser Abschnitt beschreibt einige Techniken, mit deren Hilfe eine solche Zuordnung bestimmt werden kann.

Mit Hardware-/Software-Partitionierung ist die Aufgabe gemeint, jeden Knoten des Taskgraphen entweder auf Hardware oder auf Software abzubilden. Abbildung 5.15 zeigt eine Standardvorgehensweise für die Integration der Hardware-/Software-Partitionierung in den allgemeinen Entwurfsablauf. Wir beginnen mit einer einheitlichen Darstellung der Spezifikation, beispielsweise in Form von Taskgraphen, sowie mit Informationen über die verfügbaren Plattformen.

Für jeden Knoten des Taskgraphen werden Informationen benötigt, welcher Aufwand oder welcher Vorteil mit einer bestimmten Art der Implementierung dieses Knotens verbunden ist. So müssen etwa Ausführungszeiten vorausgesagt werden (s. Seite 223). Kommunikationszeiten sind sehr schwer vorherzusagen. Trotzdem sollten Tasks, die viel miteinander kommunizieren müssen,

[1] Das Konzept des *Function Inlining* wird als bekannt vorausgesetzt, da es in Programmierkursen gelehrt wird.

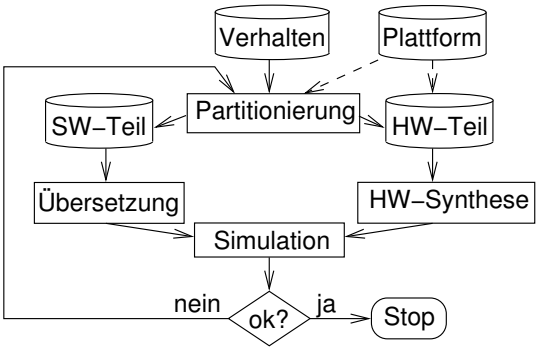


Abb. 5.15. Allgemeiner Überblick über Hardware-/Software-Partitionierung

auf die selbe Komponente abgebildet werden. Häufig werden iterative Verfahren eingesetzt. Dabei wird eine initiale Startlösung für das Partitionierungsproblem erzeugt, analysiert und schrittweise verbessert.

Einige Ansätze für die Partitionierung können Knoten des Taskgraphen lediglich auf Spezial-Hardware oder aber auf Software, die auf einem einzelnen Prozessor läuft, abbilden. Solch eine Partitionierung kann mit Bipartitionierungsalgorithmen auf Graphen [Kuchcinski, 2002] durchgeführt werden.

Umfangreichere Partitionierungsalgorithmen können Taskgraph-Knoten auch auf Multiprozessor-Systeme und auf Hardware abbilden. Im Folgenden werden wir zeigen, wie man dies mit Hilfe einer Standard-Optimierungstechnik aus dem *Operations Research*, der **ganzzahligen Programmierung**, erreichen kann. Das hier vorgestellte Modell ist eine vereinfachte Version der Optimierungen, die für das *codesign tool* COOL [Niemann, 1998] entwickelt wurden.

5.3.2 COOL

Die Eingabe für COOL besteht aus drei Teilen:

- **Zieltechnologie:** Dieser Teil der Eingaben für das COOL-System enthält Informationen über die verfügbaren Komponenten der Hardwareplattform. COOL unterstützt die Verwendung von homogenen Multiprozessor-Systemen. Heterogene Systeme mit unterschiedlichen Prozessoren werden nicht unterstützt, da COOL keine automatische oder manuelle Prozessor-Auswahl anbietet. Der Name des verwendeten Prozessors (und Informationen über den zugehörigen Compiler) muss in der COOL-Eingabe enthalten sein. Für anwendungsspezifische Hardware müssen alle Informationen und Parameter vorhanden sein, die für die Durchführung einer automatischen Hardware-Synthese benötigt werden. Insbesondere sind Informationen über die zu verwendenden Technologie-Bibliotheken notwendig.

- **Entwurfs-Einschränkungen:** Der zweite Teil der Eingabe besteht aus Einschränkungen und Randbedingungen wie etwa benötigter Durchsatz, Latenz, maximale Speichergröße oder maximale Fläche der anwendungsspezifischen Hardware.
- **Verhalten:** Drittens wird schließlich das gewünschte Verhalten spezifiziert. Dies geschieht in Form von hierarchischen Taskgraphen. Man könnte z.B. den hierarchischen Taskgraphen aus Abb. 2.47 auf Seite 58 verwenden.

  COOL verwendet zwei Typen von Kanten: Kommunikationskanten und Zeitkanten. Kommunikationskanten können Informationen über die Menge der auszutauschenden Informationen enthalten. Zeitkanten stellen Zeitbedingungen dar. Bei COOL ist es notwendig, dass das Verhalten der Blattknoten[2] des Taskgraphen bekannt ist. COOL erwartet die Spezifikation des Verhaltens in VHDL[3].

COOL führt die Partitionierung in den folgenden Schritten durch:

1. **Übersetzung** des Verhaltens **in ein internes Graphen-Modell**.
2. **Übersetzung** des Verhaltens jedes Knotens **von VHDL nach C**.
3. **Übersetzung aller C-Programme** für den ausgewählten Zielprozessor, Berechnung der resultierenden Programmgröße sowie Abschätzung der resultierenden Laufzeit. Wenn für die Bestimmung der Laufzeit Simulationen notwendig sind, müssen Simulations-Eingabedaten verfügbar sein.
4. **Synthese von Hardwarekomponenten:** Für jeden Blattknoten wird anwendungsspezifische Hardware synthetisiert. Aufgrund der möglicherweise großen Anzahl zu synthetisierender Komponenten sollte die Hardwaresynthese nicht zu langsam sein. Viele kommerzielle Synthese-Tools, die sich auf die Synthese auf Gatterebene konzentrieren, haben sich in der Praxis als zu langsam für den Einsatz mit COOL erwiesen. *High-Level* Synthese-Tools, die auf der Register-Transfer-Ebene (also mit Addierern, Registern und Multiplexern statt mit Gattern) arbeiten, haben dagegen eine ausreichende hohe Synthese-Geschwindigkeit. Solche Programme können auch ausreichend genaue Werte für Verzögerungszeiten und die benötigte Silizium-Fläche bestimmen. In der tatsächlichen Implementierung von COOL wird das *High-Level*-Synthese-Tool OSCAR [Landwehr und Marwedel, 1997] verwendet.
5. **Ausflachen der Hierarchie:** Der nächste Schritt besteht aus der Extraktion eines flachen Taskgraphen aus dem hierarchischen Taskgraphen.

[2] Def. Blattknoten s. Seite 20.

[3] Rückblickend wäre es geschickter gewesen, hierfür C zu verwenden, da dadurch die Behandlung vieler Standards, die in C spezifiziert werden, einfacher gewesen wäre.

Da keine Knoten verschmolzen oder aufgetrennt werden, bleibt bei diesem Schritt die Granularität erhalten, die der Entwickler gewählt hat. Kosten und Geschwindigkeitsinformationen, die aus der Software-Übersetzung und der Synthese erzeugt wurden, werden an die Knoten annotiert. Tatsächlich ist dies die Hauptidee bei COOL: die **Informationen, die für die Hardware-/Software-Partitionierung benötigt werden, werden im Voraus berechnet, und zwar mit guter Präzision**. Diese Informationen bilden die Basis für kostenminimale Entwürfe, welche die Entwufsvorgaben und -einschränkungen erfüllen.

6. **Erzeugen und Lösen eines mathematischen Modells des Optimierungsproblems:** COOL verwendet die Technik der ganzzahligen Programmierung (*integer programming* (IP)), um das Optimierungsproblem zu lösen. Ein kommerzielles Programm zum Lösen solcher Gleichungssysteme (*IP-Solver*) wird eingesetzt, um Werte für die Entscheidungsvariablen zu bestimmen, welche die Kosten minimieren. Die Lösung ist optimal in Bezug auf die Kostenfunktion, die aus der verfügbaren Information abgeleitet wurde. Allerdings enthält dieses Modell nur eine grobe Annäherung der Kommunikationszeit. Die Kommunikationszeit zwischen zwei Knoten eines Taskgraphen hängt von der Zuordnung dieser beiden Knoten auf Prozessoren und Hardware ab. Wenn beide Knoten auf den selben Prozessor abgebildet werden, dann kann die Kommunikation lokal und somit recht schnell durchgeführt werden. Wenn die Knoten auf unterschiedliche Hardwarekomponenten abgebildet werden, ist die Kommunikation nicht mehr lokal und somit langsamer. Das Modellieren der Kosten für alle möglichen Kombinationen der Abbildung von Taskgraphen auf Komponenten würde das Modell sehr komplex machen. Deshalb werden die Kommunikationskosten durch die iterative Verbesserung einer initialen Startlösung modelliert. Weitere Details zu diesem Schritt werden weiter unten vorgestellt.
7. **Iterative Verbesserung:** Um mit guten Abschätzungen bezüglich der Kommunikationszeiten arbeiten zu können, werden benachbarte Knoten, die auf die selbe Hardwarekomponente abgebildet werden, in diesem Schritt verschmolzen. Diese Verschmelzung wird in Abb. 5.16 gezeigt.

   Wir nehmen an, dass Tasks T_1, T_2 und T_5 auf die Hardwarekomponenten H1 und H2 abgebildet werden, wohingegen T_3 und T_4 auf dem Prozessor P1 ausgeführt werden. Folglich ist die Kommunikation zwischen T_3 und T_4 eine **lokale** Kommunikation. Daher werden T_3 und T_4 verschmolzen und die Kommunikation zwischen diesen beiden Tasks benötigt keinen Kommunikationskanal.

   Die Kommunikationszeit kann nun mit verbesserter Genauigkeit abgeschätzt werden. Der resultierende Graph wird dann als Grundlage für eine erneute mathematische Optimierung verwendet. Die letzten beiden Schrit-

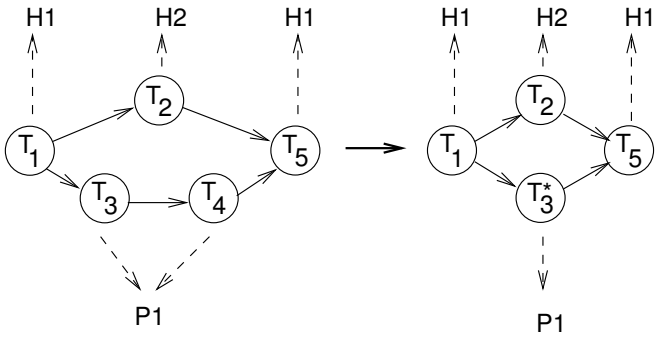


Abb. 5.16. Verschmelzen von Knoten, die auf die selbe Hardwarekomponente abgebildet werden

te, also der vorhergehende und der soeben beschriebene Schritt, werden solange wiederholt, bis keine Graphknoten mehr verschmolzen werden.

8. **Schnittstellen-Synthese:** Nach der Partitionierung wird noch die Verbindungslogik generiert, die benötigt wird, um Prozessoren, anwendungsspezifische Hardwarekomponenten und Speicher miteinander zu verbinden.

Nun wollen wir Schritt 6 etwas genauer beschreiben. IP-Modelle sind ein allgemeiner Ansatz, um Optimierungsprobleme darzustellen. IP-Modelle bestehen aus zwei Teilen: einer Kostenfunktion und einer Menge von Nebenbedingungen. Beide Teile enthalten Verweise auf eine Menge $X = \{x_i\}$ von ganzzahligen Variablen. Kostenfunktionen müssen lineare Funktionen dieser Variablen sein. Sie haben also die allgemeine Form

$$C = \sum_{x_i \in X} a_i x_i, \text{ mit } a_i \in I\!R, x_i \in \mathbf{Z} \tag{5.1}$$

Die Menge J der Nebenbedingungen muss ebenfalls aus linearen Funktionen der ganzzahligen Variablen bestehen. Sie müssen die folgende Form haben:

$$\forall j \in J : \sum_{x_i \in X} b_{i,j} x_i \geq c_j \text{ mit } b_{i,j}, c_j \in I\!R \tag{5.2}$$

Man beachte, dass man $\geq$ in Gleichung (5.2) durch $\leq$ ersetzen kann, wenn die Konstanten $b_{i,j}$ entsprechend angepasst werden.

Definition: Das **Problem der ganzzahligen Programmierung (IP)** besteht aus der Minimierung einer Kostenfunktion (5.1) unter Einhaltung der Nebenbedingungen aus Gleichung (5.2). Wenn der Wertebereich aller Variablen so weit eingeschränkt wird, dass sie nur die Werte 0 oder 1 annehmen

können, heißt das zugehörige Modell **0/1-Modell der ganzzahligen Programmierung**. In diesem Fall heißen die Variablen auch **(binäre) Entscheidungsvariablen**.

Unter der Annahme, dass x_1, x_2 und x_3 nicht negativ und ganzzahlig sind, stellt das folgende Gleichungssystem beispielsweise ein 0/1-IP-Modell dar:

$$C = 5x_1 + 6x_2 + 4x_3 \tag{5.3}$$
$$x_1 + x_2 + x_3 \geq 2 \tag{5.4}$$
$$x_1 \leq 1 \tag{5.5}$$
$$x_2 \leq 1 \tag{5.6}$$
$$x_3 \leq 1 \tag{5.7}$$

Aufgrund der Nebenbedingungen haben alle Variablen entweder den Wert 0 oder 1. Es gibt vier mögliche Lösungen. Diese sind in Tabelle 5.1 aufgelistet. Die Lösung mit den Kosten von 9 ist optimal.

x_1	x_2	x_3	C
0	1	1	10
1	0	1	9
1	1	0	11
1	1	1	15

Tabelle 5.1. Mögliche Lösungen des gezeigten IP-Problems

Anwendungen, bei denen eine **Gewinnfunktion** C' **maximiert** werden muss, können in die obige Form gebracht werden, indem man $C = -C'$ setzt.

IP-Modelle können mit Hilfe mathematischer Programmiertechniken optimal gelöst werden. Leider ist die ganzzahlige Programmierung NP-vollständig und die Ausführungszeiten können sehr lang werden. Trotzdem ist diese Technik nützlich, um Probleme optimal zu lösen, die nicht extrem groß sind. Die Ausführungszeit hängt von der Anzahl der Variablen sowie von der Anzahl und der Struktur der Nebenbedingungen ab. Gute IP-Lösungsprogramme (wie etwa lp_solve [Berkelaar und et al., 2005] oder CPLEX können wohlstrukturierte Gleichungssysteme mit einigen Tausend Variablen in akzeptablen Rechenzeiten (also in Minuten) lösen. Weitere Informationen zum Thema ganzzahlige Programmierung und zur verwandten **linearen Programmierung** findet man in entsprechenden Büchern, z.B. von Wolsey [Wolsey, 1998]). Das Modellieren von Optimierungsproblemen als IP-Modelle ist trotz der potentiell hohen Komplexität der Probleme sinnvoll: Viele Probleme können in akzeptabler Ausführungszeit gelöst werden – und selbst, wenn sie nicht gelöst werden können, liefern IP-Modelle immer noch einen guten Startpunkt für Heuristiken.

Als nächstes wird gezeigt, wie die Partitionierung mit Hilfe eines 0/1-IP-Modells beschrieben werden kann. Die folgenden Indexmengen werden bei der Beschreibung des IP-Modells verwendet:

- Indexmenge I beschreibt die Knoten des Taskgraphen, wobei jedes $i \in I$ einem Knoten entspricht.
- Indexmenge L beschreibt die **Typen der Taskgraph-Knoten**. Jedes $l \in L$ entspricht einem Taskgraph-Knoten-Typen. Beispielsweise kann es Knoten geben, welche die Berechnung der Quadratwurzel, eine diskrete Cosinus-Transformation (DCT) oder eine diskrete Fourier-Transformation (DFT) beschreiben. Jede dieser Knotenarten wird als ein Typ gezählt.
- Indexmenge KH beschreibt die **Typen der Hardwarekomponenten**. Jedes $k \in KH$ entspricht einem Hardware-Komponenten-Typ. Beispielsweise kann es spezielle Hardwarekomponenten geben, welche die DCT oder die DFT berechnen. Dann gibt es einen Indexwert für die DCT-Hardwarekomponenten und einen Indexwert für die DFT-Hardwarekomponente.
- Von jeder Hardwarekomponente gibt es mehrere Kopien oder „Instanzen“. Jede Instanz wird durch einen Index $j \in J$ identifiziert.
- Indexmenge KP beschreibt Prozessoren. Jedes $k' \in KP$ identifiziert einen der Prozessoren (die alle vom gleichen Typ sind).

Das Modell benötigt die folgenden Entscheidungsvariablen:

- $X_{i,k}$: diese Variable wird 1, wenn der Knoten v_i auf den Hardwarekomponenten-Typ $k \in KH$ abgebildet wird, und 0 sonst.
- $Y_{i,k}$: diese Variable wird 1, wenn der Knoten v_i auf den Prozessor $k \in KP$ abgebildet wird, und 0 sonst.
- $NY_{l,k}$: diese Variable wird 1, wenn mindestens ein Knoten vom Typ l auf den Prozessor $k \in KP$ abgebildet wird, und 0 sonst.
- T ist eine Abbildung $I \rightarrow L$ von den Taskgraph-Knoten auf ihre entsprechenden Typen.

In unserem speziellen Fall summiert die Kostenfunktion die Gesamtkosten aller Hardwareeinheiten:

$$C = \text{Prozessorkosten} + \text{Speicherkosten} + \text{Kosten der anwendungsspez. Hardware}$$

Die Kosten können offensichtlich minimiert werden, wenn keine Prozessoren, keine Speicher und keine anwendungsspezifischen Hardwarekomponenten im Entwurf enthalten wären. Aufgrund der Nebenbedingungen ist dies allerdings keine gültige Lösung. Wir beschreiben nun kurz einige der im IP-Modell verwendeten Nebenbedingungen (NB):

- **Operations-Zuweisungs-NB:** Diese Nebenbedinungen garantieren, dass jede Operation entweder in Hardware oder in Software ausgeführt wird. Die zugehörigen Nebenbedingungen können wie folgt formuliert werden:

$$\forall i \in I : \sum_{k \in KH} X_{i,k} + \sum_{k \in KP} Y_{i,k} = 1$$

In Worten bedeutet das: für alle Taskgraph-Knoten i muss folgendes gelten: i wird entweder in Hardware implementiert (indem eine der $X_{i,k}$-Variablen für irgendein k auf 1 gesetzt wird) oder er wird in Software implementiert (indem eine der $Y_{i,k}$-Variablen für irgendein k auf 1 gesetzt wird).

Alle Variablen sind dabei nicht-negative ganze Zahlen:

$$X_{i,k} \in I\!N_0, \tag{5.8}$$
$$Y_{i,k} \in I\!N_0 \tag{5.9}$$

Zusätzliche Nebenbedinungen stellen sicher, dass die Entscheidungsvariablen $X_{i,k}$ und $Y_{i,k}$ den Wert 1 als obere Schranke haben. Somit sind sie faktisch 0/1-Entscheidungsvariablen:

$$\forall i \in I : \forall k \in KH : X_{i,k} \leq 1$$
$$\forall i \in I : \forall k \in KP : Y_{i,k} \leq 1$$

Wenn die Funktionalität eines bestimmten Knoten vom Typ l auf einen Prozessor k abgebildet wird, muss sichergestellt werden, dass der Speicher dieses Prozessors eine Kopie der Software für diese Funktion enthält:

$$\forall l \in L, \forall i : T(v_i) = c_l, \forall k \in KP : \qquad NY_{l,k} \geq Y_{i,k}$$

In Worten bedeutet das: Für alle Typen l eines Taskgraph-Knoten und für alle Knoten i von diesem Typ muss das folgende gelten: Wenn i auf einen Prozessor k abgebildet wird (was dadurch ausgedrückt wird, dass $Y_{i,k}$ gleich 1 ist), dann muss die Funktionalität, die l entspricht, auf Prozessor k ausführbar sein, und die zugehörige Software muss auf diesem Prozessor verfügbar sein (was dadurch ausgedrückt wird, dass $NY_{l,k}$ gleich 1 ist).

Zusätzliche Nebenbedingungen stellen sicher, dass auch die Variablen $NY_{l,k}$ auf den 0/1-Wertebereich eingeschränkt sind:

$$\forall l \in L : \forall k \in KP : \qquad NY_{l,k} \leq 1$$

- **Ressourcen-NB**: Die nächste Menge von Nebenbedingungen stellt sicher, dass nicht „zu viele“ Knoten gleichzeitig auf eine Hardwarekomponente abgebildet werden. Wir nehmen an, dass in jedem Takt maximal eine Operation auf einer Hardwarekomponente ausgeführt werden kann. Unglücklicherweise bedeutet dies, dass der Partitionierungs-Algorithmus auch ein Schedule für die Ausführung der Knoten des Taskgraphen erzeugen muss. Scheduling selbst ist für die meisten relevanten Probleminstanzen bereits NP-vollständig.
- **Abhängigkeits-NB:** Diese Nebenbedinungen stellen sicher, dass das erzeugte Schedule für die Ausführung von Operationen mit den Abhängigkeiten im Taskgraphen kompatibel ist.
- **Entwurfs-NB:** Diese Nebenbedingungen begrenzen die maximalen Gesamtkosten für die Summe der Komponenten einer bestimmten Klasse von Hardwarekomponenten wie etwa Speicher, Prozessoren oder die Fläche von anwendungsspezifischer Hardware.
- **Zeit-NB**: Wenn Zeitbedingungen in der Eingabe zu COOL enthalten sind, werden auch diese in IP-Nebenbedingungen umgewandelt.
- Einige zusätzliche weniger wichtige Nebenbedingungen sind in dieser Liste nicht enthalten.

Beispiel: Im Folgenden zeigen wir, wie die Nebenbedingungen für den Taskgraphen aus Abb. 5.17 (der gleiche wie in Abb. 2.47) erzeugt werden können:

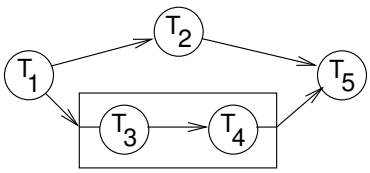


Abb. 5.17. Taskgraph

Wie nehmen an, dass eine Hardware-Komponentenbibliothek mit drei Komponenten-Typen H1, H2 und H3 mit den jeweiligen Kosten 20, 25 und 30 Kosteneinheiten vorhanden ist. Weiterhin sei auch ein Prozessor P mit Kosten 5 verfügbar. Tabelle 5.2 beschreibt die Ausführungszeiten der Tasks auf diesen Komponenten.

Die Tasks T_1 bis T_5 können nur entweder auf dem Prozessor oder auf einer anwendungsspezifischen Hardwareeinheit ausgeführt werden. Offensichtlich sind Prozessoren billig, aber langsam bei der Ausführung der Tasks T_1, T_2 und T_5.

Die folgenden Operations-Zuweisungs-Bedingungen müssen erzeugt werden, unter der Annahme, dass maximal ein Prozessor (P1) verwendet werden soll:

T	H1	H2	H3	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

Tabelle 5.2. Ausführungszeiten der Tasks T_1 bis T_5 auf Komponenten

$X_{1,1} + Y_{1,1} = 1$ (Task 1 wird entweder auf H1 oder auf P1 ausgeführt)
$X_{2,2} + Y_{2,1} = 1$ (Task 2 wird entweder auf H2 oder auf P1 ausgeführt)
$X_{3,3} + Y_{3,1} = 1$ (Task 3 wird entweder auf H3 oder auf P1 ausgeführt)
$X_{4,3} + Y_{4,1} = 1$ (Task 4 wird entweder auf H3 oder auf P1 ausgeführt)
$X_{5,1} + Y_{5,1} = 1$ (Task 5 wird entweder auf H1 oder auf P1 ausgeführt)

Wir nehmen weiter an, dass die Tasks T_1 bis T_5 jeweils die Typen $l = 1, 2, 3, 3$ und 1 haben. Dann werden die folgenden Ressourcen-Bedingungen benötigt:

$$NY_{1,1} \geq Y_{1,1} \tag{5.10}$$
$$NY_{2,1} \geq Y_{2,1}$$
$$NY_{3,1} \geq Y_{3,1}$$
$$NY_{3,1} \geq Y_{4,1}$$
$$NY_{1,1} \geq Y_{5,1} \tag{5.11}$$

Gleichung 5.10 bedeutet: Wenn Task 1 auf dem Prozessor ausgeführt wird, dann muss die Funktion $l = 1$ auf dem Prozessor implementiert werden. Die gleiche Funktion muss auch auf dem Prozessor implementiert werden, wenn Task 5 auf dem Prozessor ausgeführt werden soll (Gleichung 5.11).

Zeitbedingungen sind in diesem Beispiel nicht enthalten. Allerdings sieht man leicht, dass der Prozessor bei der Ausführung einiger Tasks sehr langsam ist und dass anwendungsspezifische Hardware notwendig ist, wenn die Ausführungszeit unter 100 Zeiteinheiten liegen soll.

Die Kostenfunktion lautet:

$$C = 20 * \#(\mathsf{H1}) + 25 * \#(\mathsf{H2}) + 30 * \#(\mathsf{H3}) + 5 * \#(\mathsf{P})$$

wobei #() die Anzahl der Instanzen einer Hardwarekomponente darstellt. Diese Zahl kann aus den bisher eingeführten Variablen berechnet werden, wenn zusätzlich das Scheduling berücksichtigt wird. Für eine Zeitschranke von 100

Zeiteinheiten besteht die Lösung mit minimalen Kosten aus den Komponenten H1, H2 und P. Das bedeutet, dass Tasks T_3 und T_4 in Software implementiert werden, alle anderen dagegen in Hardware.

Allgemein können aufgrund der Komplexität des kombinierten Partitionierungs- und Scheduling-Problems nur kleine Probleminstanzen auf diese Weise in akzeptabler Laufzeit gelöst werden. Daher wird das Problem heuristisch in ein Scheduling- und ein Partitionierungsproblem aufgespalten: eine erste Partitionierung wird aufgrund der anfangs geschätzten Ausführungszeiten bestimmt. Das endgültige Scheduling wird nach der Partitionierung durchgeführt. Sollte sich dabei herausstellen, dass das Scheduling zu optimistisch gewählt war, so muss der gesamte Prozess mit schärferen Zeitschranken wiederholt werden. Experimente mit kleinen Beispielen haben gezeigt, dass die Kosten für eine solche heuristische Lösung nur 1 bis 2% über denen der optimalen Lösung liegen.

Automatische Partitionierung kann dazu verwendet werden, den gesamten Entwurfsraum zu untersuchen. Im Folgenden zeigen wir Ergebnisse für eine Audio-Anwendung, die aus einem Mischer, Fader-, Echo-, Equalizer- und Balance-Einheiten besteht. Dieses Beispiel verwendet ältere Zieltechnologien, um den Effekt der Partitionierung zu demonstrieren. Die Zielhardware besteht aus einem (langsamen) SPARC-Prozessor, externem Speicher und anwendungsspezifischer Hardware, die aus einer (veralteten) 1μ ASIC-Bibliothek erzeugt wird. Die insgesamt zulässige Verzögerung des Systems wird auf 22675 ns gesetzt, was einer Abtastrate von 44,1 kHz entspricht, wie sie auch bei Audio-CDs verwendet wird. Abbildung 5.18 zeigt verschiedene Lösungsmöglichkeiten im Entwurfsraum, die durch Veränderungen der Verzögerungs-Nebenbedingung erzeugt werden können.

Die Einheit λ bezieht sich auf eine Technologie-abhängige Größe. Prinzipiell handelt es sich hierbei um die Hälfte der kürzesten Distanz zwischen den Mitten zweier Metall-Leitungen auf dem Chip (auch als *half-pitch* bezeichnet [SEMATECH, 2003]). Die Lösung links entspricht einer Lösung, die vollkommen in Hardware implementiert wird, die rechte Lösung wird ausschließlich in Software realisiert. Andere Lösungen verwenden eine Mischung aus Hard- und Software. Die Lösung mit einer Fläche von 78,4 λ^2 ist die kostengünstigste, welche die Zeitbedingung erfüllt.

Selbstverständlich hat sich die Technologie weiterentwickelt, so dass heute eine reine Softwarelösung für diese Audio-Anwendung möglich wäre. Trotzdem zeigt das Beispiel die zugrundeliegende Entwurfsmethodik, die auch für anspruchsvollere Anwendungen verwendet werden kann, insbesondere für solche aus dem Hochgeschwindigkeits-Bereich, wie etwa MPEG-4 oder dessen Weiterentwicklungen.

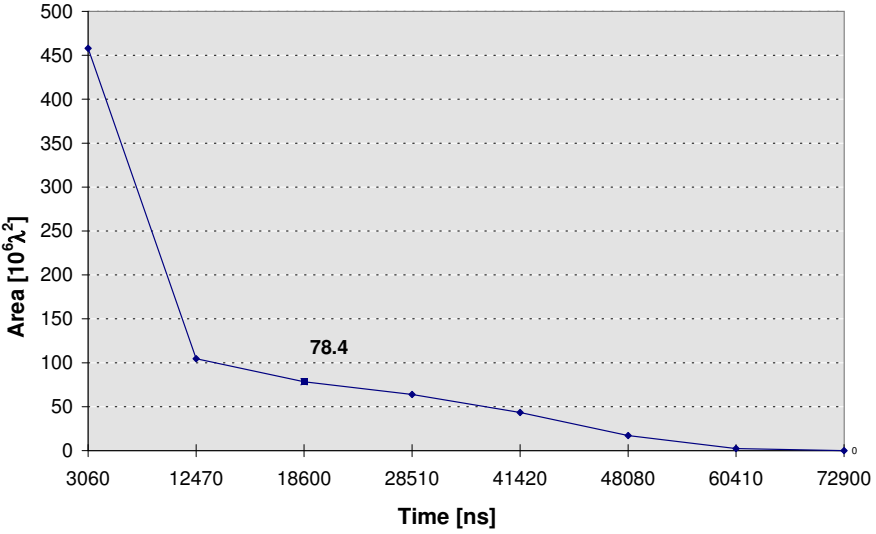


Abb. 5.18. Entwurfsraum für die Audio-Anwendung

5.4 Compiler für eingebettete Systeme

5.4.1 Einführung

Dass es für die in PCs verwendeten Prozessoren Optimierungen und Compiler gibt, ist weitgehend bekannt. Die Entwicklung von Compilern für weit verbreitete 32 Bit Prozessoren ist üblich und gut erforscht. Für eingebettete Systeme werden häufig auch Standard-Compiler verwendet, da sie normalerweise billig oder sogar frei vefügbar sind.

Es gibt allerdings einige gute Gründe, spezielle Optimierungen und Compiler für eingebettete Systeme zu entwerfen:

- Prozessorarchitekturen in eingebetteten Systemen haben spezielle Eigenschaften und Fähigkeiten (s. Seite 110). Diese Fähigkeiten sollten vom Compiler auch ausgenutzt werden, um effizienten Code zu erzeugen.
- Sehr stark optimierter Code ist wichtiger als eine hohe Geschwindigkeit des Compilers.
- Compiler können potentiell mithelfen, Echtzeitbedingungen einzuhalten und zu beweisen. Beispielsweise kann es hilfreich sein, bestimmte Cachezeilen einzufrieren und so zu verhindern, dass häufig ausgeführter Code aus dem Cache verdrängt wird und somit wiederholt geladen werden muss.

- Compiler können helfen, den Energieverbrauch eingebetteter Systeme zu reduzieren. Es sollten Compiler verfügbar sein, die Energieoptimierungen durchführen können.
- Für eingebettete Systeme gibt es eine Vielzahl von Befehlssätzen. Daher gibt es auch mehr Prozessoren, für die Compiler verfügbar sein sollten. Manchmal ist es sogar gewünscht, den Befehlssatz mit Hilfe eines **retargierbaren Compilers** zu optimieren. Bei solchen Compilern ist der Befehlssatz eine Eingabe an das Compiler-Generator-System. Solche Systeme können verwendet werden, um versuchsweise Befehlssätze zu verändern und die resultierenden Veränderungen am erzeugten Maschinencode zu untersuchen. Dies ist eine spezielle Ausprägung der Entwurfsraum-Untersuchung, die z.B. von den Tensilica Tools [Tensilica Inc., 2003] unterstützt wird.

Einige erste Ansätze für retargierbare Compiler werden im ersten Buch zu diesem Thema [Marwedel und Goossens, 1995] beschrieben. Optimierungen findet man in neueren Büchern von Leupers [Leupers, 1997], [Leupers, 2000a]. In diesem Abschnitt zeigen wir Beispiele für Compiler-Techniken für eingebettete Systeme.

Zusätzlich kann es auch sinnvoll sein, wenn Compiler für eingebettete Systeme auch Kompressionstechniken (s. Seite 113 bis 115) unterstützen.

5.4.2 Energieoptimierende Compiler

Viele eingebettete Systeme sind mobile Systeme, die mit Batterien betrieben werden. Während die Anforderungen an die Rechenleistung mobiler Systeme stark wachsen, nimmt die Kapazität der verwendeten Batterien nur langsam zu [SEMATECH, 2003]. Die Verfügbarkeit einer ausreichenden Menge an Energie ist somit ein ernst zu nehmender Engpass für neue Anwendungen.

Energie kann auf verschiedenen Ebenen eingespart werden, beispielsweise beim Fabrikationsprozess, bei der verwendeten Technologie, dem Schaltungsentwurf, dem Betriebsystem und den Anwendungs-Algorithmen. Eine entsprechende Übersetzung der Algorithmen in den Maschinencode kann ebenfalls hilfreich sein. Auch *High-Level*-Optimierungen (s. Seite 176 bis 186) können zu einer Verringerung des Energieverbrauchs beitragen. In diesem Abschnitt werden wir Compiler-Optimierungen betrachten, die den Energieverbrauch reduzieren können. Hierfür werden Energiemodelle benötigt, wie sie in Kapitel 6 beschrieben werden. Unter Verwendung solcher Energiemodelle wurden folgende Optimierungen eingesetzt, um den Energieverbrauch zu reduzieren:

- **Energieoptimierendes Scheduling:** die Befehlsreihenfolge kann verändert werden, solange die Korrektheit des Programms dabei erhalten bleibt. Die Reihenfolge der Befehle kann so verändert werden, dass die Anzahl der

Transitionen auf dem Instruktions-Bus minimiert wird. Diese Optimierung kann auf der Ausgabe des Compilers durchgeführt werden. Somit ist es nicht unbedingt erforderlich, den Compiler zu modifizieren.

- **Energieoptimierende Befehlsauswahl:** Typischerweise gibt es verschiedene Befehlsfolgen, um einen bestimmten Algorithmus auszudrücken. In einem Standardcompiler ist die Anzahl der Instruktionen oder die Anzahl von Ausführungszyklen ein häufig verwendetes Optimierungsziel (Kostenfunktion), um eine gute Befehlsauswahl zu erzeugen. Dieses Ziel kann durch den Energieverbrauch der erzeugten Befehlssequenz ersetzt werden. Steinke und andere haben herausgefunden, dass dadurch eine Reduktion des Energieverbrauchs um einige Prozent möglich ist.
- Das **Ersetzen der Kostenfunktion** ist auch für andere Standard-Compileroptimierungen möglich, wie etwa *Register-Pipelining* oder *Loop Invariant Code Motion* [Appel, 1998]. Verbesserungen von bis zu 28% wurden beobachtet [Steinke et al., 2001b]. Meist sind die Einsparungen jedoch geringer.
- **Ausnutzen der Speicherhierarchie:** Wie auf Seite 129 erklärt, verfügen kleinere Speicher über einen schnelleren Zugriff und benötigen weniger Energie pro Zugriff. Daher kann eine beträchtliche Menge von Energie eingespart werden, wenn die Existenz von kleinen *Scratchpad*-Speichern (SPMs) vom Compiler ausgenutzt werden kann. Zu diesem Zweck wird jeder Basisblock[4] und jede Variable als Speichersegment i modelliert. Für jedes Segment gibt es eine zugehörige Größe s_i. Beispielsweise durch *Profiling*, also die probeweise Ausführung oder Simulation einer Anwendung mit Testdaten, ist es möglich, den Gewinn g_i zu bestimmen, der entsteht, wenn Segment i statt im Hauptspeicher im *Scratchpad*-Speicher abgelegt wird. Sei

$$x_i = \begin{cases} 1 & \text{wenn Segment } i \text{ im } \textit{Scratchpad} \text{ liegt} \\ 0 & \text{sonst} \end{cases} \tag{5.12}$$

Dann besteht das Ziel darin, die Zielfunktion

$$\sum_i g_i \cdot x_i \tag{5.13}$$

zu maximieren und gleichzeitig die Größenbedingung

$$\sum_i s_i \cdot x_i \leq K \tag{5.14}$$

[4] Ein Basisblock ist eine maximal lange Abfolge sequentiell ausgeführter Befehle.

einzuhalten, wobei K die Größe des *Scratchpads* ist.

Dieses Problem ist als Rucksack-Problem bekannt. Die Lösung des Problems ist eine Eins-zu-Eins-Abbildung zwischen Speichersegment und SPM-Bereich. Ein Modell der ganzzahligen Programmierung, das zu einer solchen Abbildung führt, wurde von Steinke vorgestellt [Steinke et al., 2002]. Für einige Beispielprogramme wurde eine Energieeinsparung von bis zu 80% erreicht, obwohl die Größe des *Scratchpad*-Speichers nur einen kleinen Bruchteil der Gesamtcodegröße betrug. Die Ergebnisse für den Sortieralgorithmus *bubblesort* sind in Abb. 5.19 gezeigt.

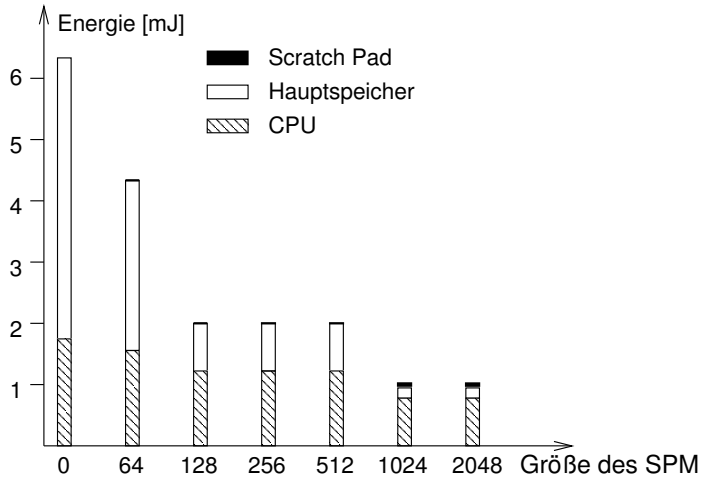


Abb. 5.19. Energiereduktion durch compilergesteuerte Verschiebung ins SPM für *bubblesort*

Offenbar führen größere SPMs zu einem reduzierten Energieverbrauch im Hauptspeicher. Die Energie, die vom Prozessor verbraucht wird, wird ebenfalls reduziert, da weniger Wartezyklen benötigt werden. Die Versorgungsspannung wurde als konstant angenommen. Von allen Compileroptimierungen, die von Steinke analysiert wurden, sind die möglichen Energieeinsparungen durch die Ausnutzung der Speicherhierarchie am größten.

Die Optimierung kann in verschiedene Richtungen erweitert werden. Zunächst einmal kann versucht werden, möglichst alle Klassen von Speichersegmenten als Kandidaten für eine Zuordnung zum *Scratchpad* zu betrachten. Globale Variablen können relativ leicht dem *Scratchpad* zugeordnet werden. Dasselbe gilt für ganze Funktionen. Einzelne Codefragmente innerhalb von Funktionen (wie z.B. Basisblöcke) können nur dann im SPM realisiert werden, wenn explizite Sprünge zwischen den Codefragmenten im Hauptspeicher und jenen im SPM erzeugt werden. Der Laufzeit-*Stack* kann verlagert werden, wenn nachgewiesen werden kann, dass er klein ge-

nug bleibt. Dazu muss ein Werkzeug eingesetzt werden, das Schranken für die Größe des Laufzeit-*Stack*s bestimmt. Aufgrund des Halteproblems ist dies natürlich nur für eingeschränkte Programme möglich.

Die größte Effizienz eines SPMs ergibt sich dann, wenn seine Größe etwa der Anzahl der häufig benötigten Speicherzellen (des sogenannten *working sets*) eines Programmes entspricht. Wenn auf einer Hardwareplattform verschiedene Programme mit verschiedenen *working set*-Größen ausgeführt werden sollen, entsteht so das Problem der Wahl der richtigen SPM-Größe. Dieses Problem kann durch partitionierte SPMs gelöst werden. Werden SPMs in mehrere kleine Speicher aufgeteilt, so kann deren Benutzung auf die Größe des *working sets* begrenzt werden. Wehmeyer et al. haben gezeigt, welche Einsparungen so gegenüber dem Fall eines einzigen SPMs möglich sind [Wehmeyer und Marwedel, 2006]. Sie haben auch demonstriert, dass der Einsatz von SPMs die WCET deutlich verbessern kann, da bereits zur Übersetzungszeit bekannt ist, ob ein Speichersegment sich zur Laufzeit im SPM oder in einem langsameren Speicher befinden wird.

Die erwähnten Optimierungen erlauben noch kein dynamisches Verschieben von Speichersegmenten zur Laufzeit. Bei großen Programmen mit mehreren *hot spots* ist es sinnvoll, die jeweils gerade häufig benutzten Segmente im SPM zu haben und die übrigen ggf. wieder auf langsamere Ebenen der Speicherhierarchie zu verdrängen. Man kommt so zu einem automatischen Ein- und Auslagern zwischen den verschiedenen Hierarchieebenen. Die Speicherbereiche im SPM werden während der Laufzeit immer wieder für andere Speicherobjekte benutzt. Dies entspricht der *Overlay*-Technik, die in Zeiten knappen Speichers durch die Programmierer anzuwenden war, die jetzt aber automatisch erfolgt. Diese Form der SPM-Nutzung kann man daher *overlaying memory allocation* nennen, die vorher beschriebene Form von Steinke et al. heißt daher *non-overlaying memory allocation*[5]. *Overlaying memory allocation* kann auch mit dem *demand paging* zwischen Hauptspeicher und Plattenlaufwerken verglichen werden. Im Unterschied zum *demand paging* erfolgt jetzt das Ein- und Auslagern aber unter Compilerkontrolle statt auf Veranlassung der *Paging*-Hardware. Daraus resultiert eine gute Vorhersagbarkeit des zeitlichen Verhaltens des Systems. Ein Algorithmus zum Einfügen der notwendigen Kopierbefehle wurde von Verma et al. vorgestellt [Verma und Marwedel, 2007]. Eine Erweiterung auf mehrere Speicherhierarchieebenen beschreiben Brockmeyer et al. [Brockmeyer et al., 2003]. Diese Erweiterung wird allerdings (noch) nicht durch automatische Werkzeuge unterstützt.

Werden SPMs in Mehrprozess-Systemen eingesetzt, so gehören deren Inhalte mit zum Prozess-Kontext, müssen also im Allgemeinen bei Pro-

[5] Manchmal wird auch zwischen statischer und dynamischer Allokation unterschieden. Allerdings sind diese beiden Adjektive semantisch mehrdeutig.

zessumschaltung gesichert und zurückgeschrieben werden. Der Aufwand dafür könnte die Einsparungen durch die Nutzung eines SPMs gefährden. Verma et al. beschreiben Algorithmen, mit denen der Aufwand dafür reduziert werden kann [Verma und Marwedel, 2007]. Wenn zur Entwurfszeit nicht bekannt ist, welche Programme gleichzeitig in einem System ausgeführt werden, dann ist eine Unterstützung der SPM-Verwaltung durch ein Betriebssystem erforderlich. In dieser Richtung liegen Ansätze von Pyka et al. vor (s. u.a. [Faßbach, 2006]). Egger et al. haben beschrieben [Egger et al., 2006], wie SPMs in Kombination mit Speicherverwaltungseinheiten (*Memory Management Units* (MMUs)) genutzt werden können. Weitere Arbeiten zu SPMs stammen z.B. von Kandemir [Kandemir et al., 2004].

5.4.3 Compiler für digitale Signalverarbeitung

Eigenschaften und Fähigkeiten von Prozessoren für die digitale Signalverarbeitung (*Digital Signal Processor* (DSP)) wurden auf Seite 117 beschrieben. Diese sollten von Compilern unbedingt ausgenutzt werden. Entsprechende Techniken können beispielsweise anhand der Adressgenerierungseinheiten (*Address Generation Units* (AGU)) gezeigt werden. Diese Einheiten erlauben es, bestimmte Adressen ohne zusätzliche Zyklen parallel zu Operationen im übrigen Teil des Rechenwerks zu erzeugen (s. Seite 116). Die Möglichkeit, Adressen „gratis“ zu erzeugen, hat einen starken Einfluss darauf, wie Variablen im Speicher abgelegt werden sollten. Abbildung 5.20 zeigt ein Beispiel.

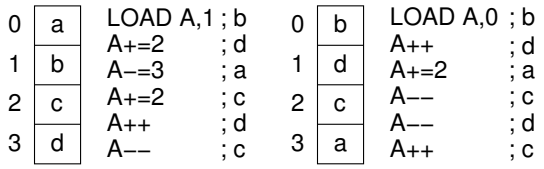


Abb. 5.20. Vergleich der Speicher-Belegungen: Original (links) und optimiert (rechts)

Wir nehmen an, dass auf die Variablen a bis d innerhalb eines Basisblocks in der Reihenfolge (b, d, a, c, d, c) zugegriffen wird. Bei Verwendung von Registerindirekter Adressierung für den Zugriff auf diese Variablen muss zunächst die Adresse der Variablen b in ein Adressregister geladen werden. Wir betrachten zunächst einmal die Speicherbelegung in Abb. 5.20 (links). Hier ist für den Zugriff auf b die Adresse 1 in das Adressregister zu laden. Die eigentliche Instruktion, die auf den Wert von b zugreift, ist in Abb. 5.20 nicht dargestellt, da wir uns auf die Generierung des Adresscodes konzentrieren wollen. Deshalb wird als nächstes die Erzeugung der Adresse für den Zugriff auf die

nächste Variable, nämlich d, betrachtet. Unter der Annahme, dass es nur ein einziges Adressregister A gibt, muss A jetzt aktualisiert werden, so dass es auf Variable d zeigt. Dazu muss der Wert 2 auf das Register addiert werden. Wir ignorieren wiederum die eigentliche Instruktion, die den Wert von d lädt und betrachten jetzt den Zugriff auf a. Diesmal muss der Wert 3 von A subtrahiert werden, und für den nächsten Zugriff muss wieder 2 addiert werden. Angenommen, der Bereich der Autoinkrement- und Autodekrement-Operationen ist auf ± 1 beschränkt, so können nur die letzten beiden Zugriffe in Abb. 5.20 mithilfe dieser Operationen implementiert werden. Insgesamt werden also vier zusätzliche Instruktionen zur Adressberechnung benötigt.

Wenn wir im Gegensatz dazu eine Speicherbelegung wie im rechten Teil von Abb. 5.20 annehmen, können vier Adressberechnungen mithilfe von Autoinkrement- und Autodekrement-Operationen berechnet werden, die parallel mit den Operationen des Haupt-Rechenwerks ausgeführt werden können. Lediglich zwei zusätzliche Befehle werden für diejenigen Adressberechnungen benötigt, die einen größeren *Offset* als eins haben.

Wie kann man nun solch eine geschickte Speicherbelegung berechnen? Algorithmen gehen hierfür üblicherweise von einem Zugriffsgraphen aus, wie er in Abb. 5.21 dargestellt ist.

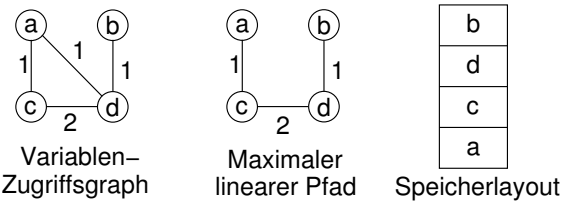


Abb. 5.21. Speicherallokation für die Zugriffssequenz (b, d, a, c, d, c) mit einem Adressregister A

Solche Zugriffsgraphen haben einen Knoten für jede Variable und eine Kante für jedes Paar von Variablen, auf das direkt nacheinander zugegriffen wird. Das Kantengewicht entspricht der Anzahl der direkt aufeinanderfolgenden Zugriffe auf die beiden Variablen.

Variablen, die durch eine Kante mit hohem Gewicht verbunden sind, sollten vorzugsweise in benachbarten Speicherzellen abgelegt werden. Die Anzahl eingesparter separater Adressberechnungs-Instruktionen entspricht dem Gewicht der jeweiligen Kante. Wenn also Variablen c und d in benachbarten Speicherzellen abgelegt werden, können die beiden letzten Zugriffe in der Zugriffs-Sequenz mithilfe von Autoinkrement- und Autodekrement-Operationen durchgeführt werden.

Das Ziel des Algorithmus ist eine lineare Anordnung von Variablen im Speicher, bei der die Verwendung von Autoinkrement- und Autodekrement-

Operationen maximiert wird. Das entspricht der Suche nach einem linearen Pfad mit maximalen Kantengewicht im Zugriffsgraphen. Leider ist dieses Problem NP-vollständig, weswegen zur Berechnung solcher Pfade häufig Heuristiken eingesetzt werden [Liao et al., 1995b], [Sudarsanam et al., 1997]. Die meisten basieren auf Kruskals Spannbaum-Heuristik. Sie beginnen mit einem Graphen, der keine Kanten enthält und fügen der Reihe nach Kanten mit abnehmendem Gewicht hinzu, wobei der Grad jedes Knotens maximal 2 betragen darf und gleichzeitig die Bildung von Zyklen vermieden werden muss. Die Reihenfolge der Variablen im Speicher entspricht dann der Reihenfolge der Variablen entlang dieses linearen Pfads.

Der gerade skizzierte Algorithmus deckt nur einen einfachen Fall ab. Erweiterungen lösen Spezialfälle optimal oder können mit komplexeren Situationen umgehen, so etwa:

- $n > 1$ Adressregister [Leupers und Marwedel, 1996]
- Verwendung von sogenannten *Modifier Registers* in der Adressgenerierungseinheit [Leupers und Marwedel, 1996], [Leupers und David, 1998]
- Erweiterungen für Arrays von Basu et al. [Basu et al., 1999]
- Das Problem der optimalen Vergabe von Adressregistern bei fester Speicherzuordnung wird als *address pointer assignment* (APA) Problem bezeichnet. Gebotys hat gezeigt, dass dieses auf ein *mimimum cost circulation* Problem zurückgeführt und optimal gelöst werden kann [Gebotys, 1997]. Tatsächlich könnte dieser Ansatz die Ergebnisse von Basu et al. verbessern.
- Abdeckung größerer Bereiche durch die Autoinkrement- und Autodekrement-Operationen [Sudarsanam et al., 1997]
- Berücksichtigung der Möglichkeit, mehreren Variablen denselben Speicherplatz zuzuordnen, wenn sich deren Lebensdauern nicht überlappen [Ottoni, 2003]
- Ausnutzen der Möglichkeit, Speicherzugriffe zeitlich neu einzuplanen, wenn die Datenabhängigkeiten dies erlauben [Choi und Kim, 2002]

In [Huynh et al., 2006] wurden die Ergebnisse der verschiedenen Verfahren verglichen. Die oben beschriebene Speicherallokation verbessert sowohl die Codegröße als auch die Laufzeit des erzeugten Codes. Andere Optimierungsalgorithmen nutzen weitere Architekturmerkmale von DSP-Prozessoren aus, etwa:

- mehrere Speicherbänke [Sudarsanam und Malik, 1995],
- heterogene Registersätze [Araujo und Malik, 1995],
- Modulo-Adressierung,
- Möglichkeit der Parallelität auf Befehlsebene (*Instruction Level Parallelism* (ILP)) [Leupers und Marwedel, 1995],

- mehrere Betriebs-Modi [Liao et al., 1995a].

Weitere neue Optimierungen werden in [Leupers, 2000a] beschrieben.

5.4.4 Compiler für Multimedia-Prozessoren

Um gepackte Datentypen, wie sie auf Seite 120 beschrieben wurden, vollständig zu unterstützen, muss der Compiler in der Lage sein, Operationen innerhalb von Schleifen automatisch in Operationen auf gepackten Datentypen umzuwandeln. Das Ausnutzen dieser Datentypen ist für die Erzeugung effizienten Codes notwendig. Es ist eine sehr schwierige Aufgabe, diese Architektureigenschaft im Compiler zu berücksichtigen. Die Algorithmen, die innerhalb von Compilern zum Einsatz kommen, um gepackte Datentypen zu unterstützen, sind Weiterentwicklungen von Vektorisierungs-Algorithmen, die ursprünglich für Supercomputer entwickelt wurden. Nur wenige dieser Algorithmen wurden bisher beschrieben [Fisher und Dietz, 1998], [Fisher und Dietz, 1999], [Leupers, 2000b], [Krall, 2000], [Larsen und Amarasinghe, 2000].

Die automatische Parallelisierung von Schleifen für den M3-DSP (s. Seite 123) erfordert den Einsatz von Vektorisierungs-Techniken, die (im Vergleich zu einer sequentiellen Abarbeitung der Operationen, s. Abb. 5.22) eine signifikante Beschleunigung des Codes erreichen [Lorenz et al., 2002]. Für die Anwendung dot_product_2 ist die Größe der Vektoren zu klein, um zu einer bemerkenswerten Beschleunigung zu führen, und folglich konnte keine Vektorisierung durchgeführt werden. Die Anzahl der Zyklen konnte für den Benchmark example um 94% reduziert werden, wenn die Vektorisierung in Kombination mit der Ausnutzung von Hardwareschleifen (*zero overhead hardware loops*) eingesetzt wird.

5.4.5 Compiler für VLIW-Prozessoren

VLIW-Architekturen (s. Seite 121) machen einige besondere Compiler-Optimierungen erforderlich:

- Eine der wichtigsten Optimierungen für TMS 320C6xx-Prozessoren ist die compilergesteuerte Zuordnung der Funktionseinheit, die eine bestimmte Operation ausführen soll. Aufgrund der zwei Rechenwerke (s. Abb. 3.21 auf Seite 123) müssen die Operationen hierfür in zwei Mengen aufgeteilt werden [Jacome und de Veciana, 1999], [Jacome et al., 2000], [Leupers, 2000c] und außerdem einem der beiden Registersätze zugeordnet werden.
- VLIW-Prozessoren verfügen häufig über sogenannte *Branch Delay Slots* (s. Seite 124). Bei VLIW-Prozessoren ist der Geschwindigkeitsverlust bei einem Sprung sehr viel größer als bei anderen Prozessoren, da jeder *Branch*

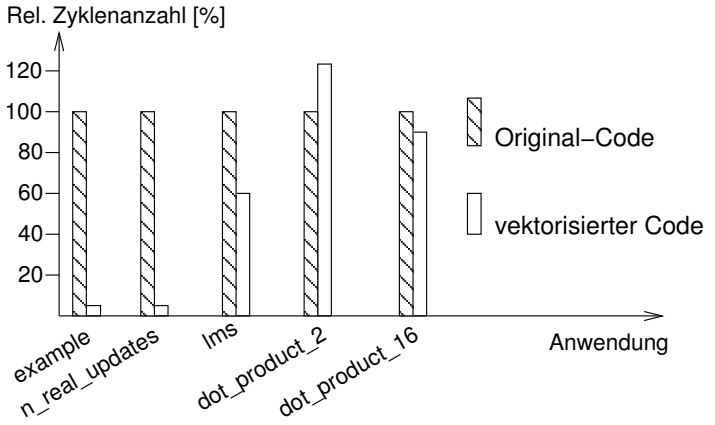


Abb. 5.22. Reduktion der Anzahl der Ausführungszyklen durch Vektorisierung beim M3-DSP

Delay Slot statt einer einzelnen Instruktion ein komplettes Instruktionspaket enthalten kann. Beim TMS 320C6xx beträgt die *Branch Delay Penalty* beispielsweise $5 \times 8 = 40$ Instruktionen. Um diesen Verlust zu vermeiden, unterstützen die meisten VLIW-Prozessoren die bedingte Ausführung von Befehlen für viele *Condition Code*-Register. Die bedingte Ausführung erlaubt die effiziente Implementierung kleiner if-Abfragen. Für größere if-Anweisungen sind bedingte Sprünge allerdings wieder effizienter, da man hier den gegenseitigen Ausschluss der then- und else-Zweige bei der Ressourcen-Zuteilung (z.B. der Funktionseinheiten) berücksichtigen kann. Die genaue Beurteilung dieser beiden Implementierungsvarianten von Fallunterscheidungen kann durch eine entsprechende Optimierungstechnik erreicht werden [Mahlke et al., 1992], [August et al., 1997], [Leupers, 1999].

- Aufgrund der erwähnten hohen *Branch Delay Penalty* ist das Inlining (s. Seite 186) eine weitere sehr nützliche Optimierung für VLIW-Prozessoren.

5.4.6 Compiler für Netzwerkprozessoren

Netzwerkprozessoren sind eine relativ neue Art von Prozessoren. Sie sind für Hochgeschwindigkeits-Internetanwendungen optimiert. Ihr Befehlssatz enthält zahlreiche Befehle, die auf Bitfelder in Informationsströmen zugreifen und diese verändern können. Typischerweise werden diese Prozessoren in Assembler programmiert, da der Durchsatz höchste Priorität hat. Andererseits werden Netzwerkprozessoren zunehmend komplexer, und somit können Compiler für Netzwerkprozessoren den Entwurf neuer Netzwerkkomponenten unterstützen. Die notwendigen Analysen auf der Bit-Ebene wurden von Wagner et al. untersucht [Wagner und Leupers, 2002]. Durch die Ausnutzung von speziellen

Instruktionen auf Bit-Ebene wurde bei einem Netzwerkprozessor eine Geschwindigkeitssteigerung um 28% erreicht.

5.4.7 Compiler-Erzeugung, retargierbare Compiler und Untersuchung des Entwurfsraums

Als die ersten Compiler entwickelt wurden, war dies ein vollkommen manueller Vorgang. Mittlerweile wurden einige der Schritte, die zum Entwurf eines Compilers notwendig sind, automatisiert oder durch Werkzeuge unterstützt. Beispielsweise sind lex und yacc und aktuellere Versionen dieser Werkzeuge[6] Hilfsmittel zum Parsen von Sourcecode. Das Erzeugen des Maschinencodes wird mittlerweile ebenfalls von Werkzeugen unterstützt. So kann die **baumbasierte Musterüberdeckung** (*Tree Pattern Matching*), wie sie etwa in **olive**[7] enthalten ist, zur Codeauswahl verwendet werden. Trotz dieser und weiterer Hilfsmittel ist der Entwurf eines Compilers nach wie vor nicht vollständig automatisierbar.

Trotzdem gibt es viele Ansätze, um retargierbare Compiler zu entwerfen. Wir unterscheiden zwischen verschiedenen Arten von Retargierbarkeit:

- **Entwickler-Retargierbarkeit:** In diesem Fall sind Compiler-Spezialisten dafür verantwortlich, den Compiler an einen neuen Prozessor und dessen Befehlssatz anzupassen.
- **Endanwender-Retargierbarkeit:** In diesem Fall kann der Anwender den Compiler retargieren. Dieser Ansatz ist sehr viel anspruchsvoller.

Weitere Informationen über retargierbare Compiler und ihren Einsatz für die Untersuchung des Entwurfsraumes findet man in einem Buch von Leupers und Marwedel [Leupers und Marwedel, 2001].

5.5 Versorgungsspannungs-Anpassung und Energie-Management

5.5.1 Dynamische Anpassung der Versorgungsspannung

Einige eingebettete Prozessoren unterstützen die dynamische Anpassung der Versorgungsspannung (*Dynamic Voltage Scaling* (DVS)) und dynamisches Power-Management (s. Seite 111). Diese Fähgikeiten können durch einen weiteren Optimierungsschritt ausgenutzt werden. Normalerweise findet diese Optimierung nach der Codererzeugung durch den Compiler statt. Sie erfordert

[6] http://www.combo.org/lex_yacc_page

[7] Olive ist Teil der Software des SPAM-Compilerprojekts der Universität Princeton. Leider wird diese Software derzeit nicht mehr im Internet angeboten.

eine globale Sicht aller im System vorhandenen Tasks sowie deren Abhängigkeiten, Schlupfzeiten und weitere Informationen.

Das Einsparungspotential bei der Anpassung der Versorgungsspannung zeigt das folgende Beispiel [Ishihara und Yasuura, 1998]. Wir gehen von einem Prozessor aus, der die drei verschiedenen Versorgungsspannungen 2,5 V, 4,0 V und 5,0 V unterstützt. Unter der Annahme, dass ein Prozessorzyklus bei 5,0 V eine Energie von 40 nJ verbraucht, kann mithilfe von Gleichung 3.1 auf Seite 111 die Energie bei Verwendung der anderen Versorgungsspannungen berechnet werden (s. Tabelle 5.3, wobei 25 nJ ein gerundeter Wert ist).

V_{dd} [V]	5.0	4.0	2.5
Energie pro Zyklus [nJ]	40	25	10
f_{max} [MHz]	50	40	25
Zykluszeit [ns]	20	25	40

Tabelle 5.3. Charakteristische Daten eines Prozessors mit DVS

Desweiteren nehmen wir an, dass die betrachtete Task 10^9 Zyklen innerhalb von 25 Sekunden ausführen muss. Es gibt verschiedene Möglichkeiten, dies zu erreichen, wie die Abbildungen 5.23 und 5.24 zeigen. Bei Verwendung der maximalen Spannung (Fall a), Abb. 5.23), kann man den Prozessor während der Schlupfzeit für 5 Sekunden abschalten. Wir nehmen an, dass der Prozessor in diesem Zustand keine Energie verbraucht.

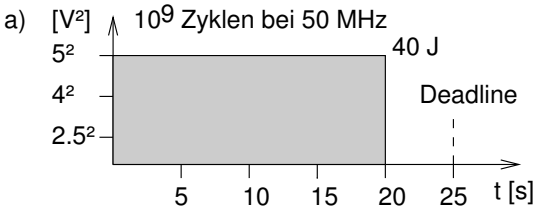


Abb. 5.23. Möglicher Verlauf der Versorgungsspannung

Eine andere Möglichkeit (Fall b), Abb. 5.24 oben) besteht darin, den Prozessor zuerst mit maximaler Geschwindigkeit laufen zu lassen und die Spannung dann so zu reduzieren, dass die restlichen Zyklen mit der geringstmöglichen Versorgungsspannung ausgeführt werden können. Schließlich kann man den Prozessor auch mit einer Versorgungsspannung betreiben, die gerade groß genug ist, damit die Task in der vorgegebenen Zeit beendet werden kann (Fall c), Abb. 5.24 unten).

Die zugehörigen Energieverbräuche berechnen sich wie folgt:

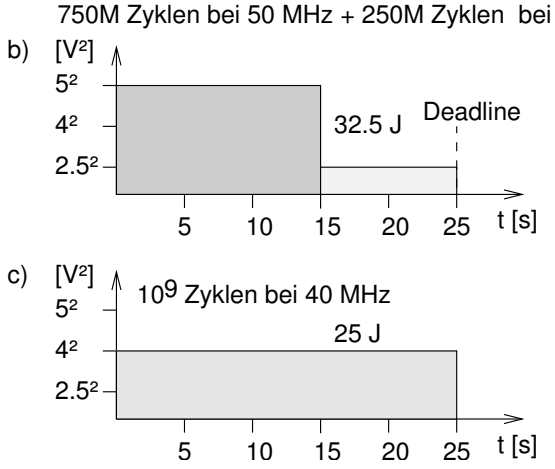


Abb. 5.24. Zwei weitere Verläufe der Versorgungsspannung

$$E_a = 10^9 \times 40 \cdot 10^{-9} = 40\ [J] \tag{5.15}$$

$$E_b = 750 \cdot 10^6 \times 40 \cdot 10^{-9} + 250 \cdot 10^6 \times 10 \cdot 10^{-9} = 32.5\ [J] \tag{5.16}$$

$$E_c = 10^9 \times 25 \cdot 10^{-9} = 25\ [J] \tag{5.17}$$

Der minimale Energieverbrauch wird für die ideale Versorgungsspannung von 4 Volt erreicht. Im Folgenden verwenden wir den Begriff **Prozessor mit variabler Versorgungsspannung** nur für solche Prozessoren, die mit einer **beliebigen** Spannung (bis zu einem bestimmten Maximalwert) betrieben werden können. In der Praxis ist es sehr teuer, wirklich variable Spannungen zu unterstützen. Daher unterstützen reale Prozessoren in der Regel nur einige wenige fest vorgegebene Spannungen.

Die Beobachtungen aus dem obigen Beispiel können wie folgt verallgemeinert werden:

- Wenn ein Prozessor mit variabler Versorgungsspannung eine Task vor ihrer Deadline abschließt, kann der Energieverbrauch reduziert werden[8].
- Wenn ein Prozessor eine einzige Versorgungsspannung V verwendet und eine Task T genau zu ihrer Deadline abschließt, dann ist V diejenige eindeutige Versorgungsspannung, die den Energieverbrauch von T minimiert.

Diese Aussagen können verwendet werden, um Tasks Spannungen zuzuordnen. Wir betrachten nun die Zuordnung von Spannungen an eine Menge von Tasks [Ishihara und Yasuura, 1998] [Okuma et al., 1999]. Wir verwenden die folgende Notation:

[8] Diese Formulierung macht eine implizite Annahme in Lemma 1 des Beitrags von Ishihara und Yasuura explizit.

N : Anzahl der Tasks
EC_j : Anzahl der bereits ausgeführten Zyklen von Task j
L : Anzahl der Versorgungsspannungen des Zielprozessors
V_i : die i. Spannung, mit $1 \leq i \leq L$
F_i : die Taktfrequenz für Versorgungsspannung V_i
T : die globale Deadline, bei der alle Tasks abgeschlossen sein müssen
$X_{i,j}$: die Anzahl von Taktzyklen, die Task j bei Versorgungsspannung V_i ausgeführt wird
SC_j : die durchschnittliche Schalt-Kapazität während der Ausführung von Task j (SC_i enthält die Kapazität C_L und die Schaltaktivität α (s. Gleichung 3.1 auf Seite 111))

Die Anpassung der Versorgungsspannung kann nun als Problem der ganzzahligen Programmierung (*Integer Programming* (IP)) formuliert werden (s. Seite 190). Wir gehen u.a. von den folgenden vereinfachenden Annahmen aus:

- Es gibt einen Zielprozessor, der mit einer festen Anzahl von diskreten Versorgungsspannungen betrieben werden kann.
- Die Zeit für Versorgungsspannungs- und Frequenz-Umschaltungen ist vernachlässigbar kurz.
- Die maximale Anzahl von Taktzyklen für jede Task ist bekannt.

Mit diesen Annahmen kann das IP-Problem wie folgt formuliert werden:

Minimiere

$$E = \sum_{j=1}^{N} \sum_{i=1}^{L} SC_j \cdot x_{i,j} \cdot V_i^2 \tag{5.18}$$

unter den Nebenbedingungen

$$\sum_{i=1}^{L} x_{i,j} = EC_j \tag{5.19}$$

und

$$\sum_{j=1}^{N} \sum_{i=1}^{L} \frac{x_{ij}}{F_i} \leq T \tag{5.20}$$

Das Ziel besteht darin, die Anzahl $x_{i,j}$ von Zyklen herauszufinden, die jede Task j mit einer bestimmten Versorgungsspannung V_i ausgeführt wird. Mit diesem Modell zeigen Ishihara und Yasuura, dass die Effizienz des Systems in der Regel verbessert wird, wenn man mehrere Spannungspegel zur Auswahl hat. Wenn viel Schlupf vohanden ist, kann man bei Verfügbarkeit

von mehreren Spannungspegeln diejenigen bestimmen, die der idealen Versorgungsspannung nahekommen. In der Praxis erreicht man jedoch bereits mit vier Spannungspegeln häufig recht gute Ergebnisse.

In vielen Fällen ist die Ausführung der Tasks schneller als die vorhergesagte maximale Ausführungszeit. Dieses Verhalten kann vom oben skizzierten Algorithmus nicht ausgenutzt werden. Diese Einschränkung lässt sich durch die Verwendung von sogenannten *Checkpoints* beheben, an denen maximale und tatsächliche Laufzeiten verglichen werden. Das Ergebnis des Vergleichs kann dann dazu verwendet werden, die Spannung weiter zu reduzieren [Azevedo et al., 2002]. Eine Erweiterung auf Multiprozessoren beschreibt Jovanovic [Jovanovic, 2006].

5.5.2 Dynamisches Power-Management

Um den Energieverbrauch zu reduzieren, kann man auch Stromsparzustände ausnutzen, wie sie auf Seite 111 eingeführt wurden. Man spricht in diesem Zusammenhang von dynamischem Power-Management (*Dynamic Power Management* (DPM)). Die Kernfrage hierbei lautet: wann sollte man in einen stromsparenden Zustand wechseln? Einfache Ansätze wechseln nach einer festgelegten Zeit in einen Stromsparzustand. Komplexere Methoden modellieren die Zeiten, in denen der Prozessor nicht ausgelastet ist, in Form von stochastischen Prozessen, mit deren Hilfe die Auslastung von Subsystemen genauer vorhergesagt werden kann. Modelle, die auf Exponentialverteilungen basieren, haben sich als ungenau erwiesen. Ausreichend genaue Modelle basieren beispielsweise auf der sogenannten Erneuerungs-Theorie (*Renewal Theory*) [Simunic et al., 2000].

Ein vollständiger Überblick über Energiemanagement wurde von Benini et al. [Benini und Micheli, 1998] veröffentlicht. Es gibt auch Algorithmen, die das dynamische Anpassen der Versorgungsspannung mit DPM in einer einzigen Optimierung kombinieren [Simunic et al., 2001].

Die Zuweisung von Spannungen und die Berechnung der Übergänge in Stromsparzustände gehören zu den letzten Schritten bei der Optimierung eingebetteter Software.

6

Evaluierung und Validierung

6.1 Einleitung

Zwei wichtige Aspekte des Entwurfs eingebetteter Systeme wurden bisher nicht betrachtet: die Evaluierung und die Validierung. Bei der Evaluierung wird überprüft, ob ein Entwurf (möglicherweise auch nur ein Teilentwurf) bestimmte Anforderungen erfüllt. Bei der Validierung wird überprüft, ob ein bestimmter Entwurf oder ein Teil davon seinem Zweck entspricht und sich wie erwartet verhalten wird. Während die Validierung mehr auf die Überprüfung der Korrektheit eines Entwurfs zielt, dient die Evaluierung mehr der Überprüfung quantitativer Parameter, wie der Leistung, des Energieverbrauchs oder der erwarteten Lebensdauer. In vielen Entwurfsprozessen sind Evaluationen und Validierungen miteinander verknüpft.

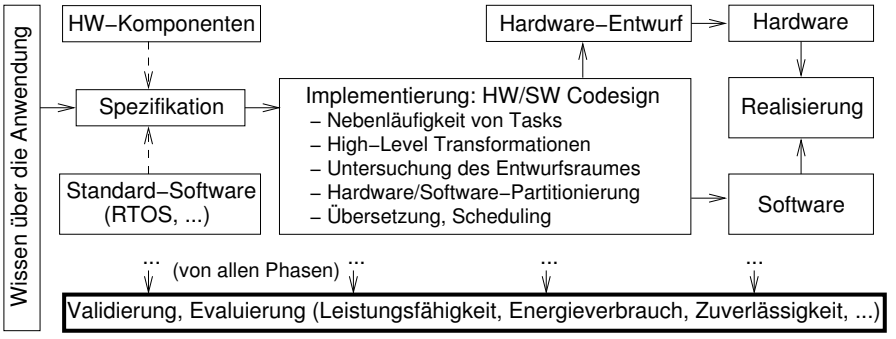


Abb. 6.1. Vereinfachter Informationsfluss beim Entwurf eingebetteter Systeme

Evaluierung und Validierung sind für jeden Entwurf wichtig, und kaum ein System würde sich wie beabsichtigt verhalten, wenn es nicht im Rahmen des Entwurfs auch evaluiert und validiert worden wäre. Für sicherheitskritische

Systeme ist die Validierung unerlässlich. Gemäß der bislang schon verwendeten Darstellung des Flusses von Entwurfsinformationen (s. Abb. 6.1) beschäftigen wir uns in diesem Kapitel mit der Evaluierung und Validierung.

Theoretisch wäre es möglich, vollständig verifizierte Werkzeuge zu entwerfen, die anhand der Spezifikationen immer korrekte Implementierungen erzeugen, welche alle quantitativen Parameter einhalten. In der Praxis werden derartige Werkzeuge nur in sehr einfachen Fällen bereitstehen. Folglich muss jeder Entwurf einzeln überprüft werden. Um die Anzahl der Überprüfungen zu minimieren, könnte man versuchen, erst ganz am Ende des Entwurfsablaufs zu evaluieren und validieren. Leider funktioniert dieses Vorgehen in der Regel nicht, da es zu große Unterschiede zwischen der Abstraktionsebene der Spezifikation und derjenigen der Implementierung gibt. Daher müssen die Überprüfungen in mehreren Phasen während des Entwurfsablaufs durchgeführt werden. Evaluierung, Validierung und Entwurf sollten miteinander verwoben sein und nicht als voneinander unabhängige Aufgaben gesehen werden. Besonders deutlich wird dies am Entwurfsfluss für die SpecC Entwurfsmethodik (siehe Abb. 6.2) [Center for Embedded Computer Systems, 2003].

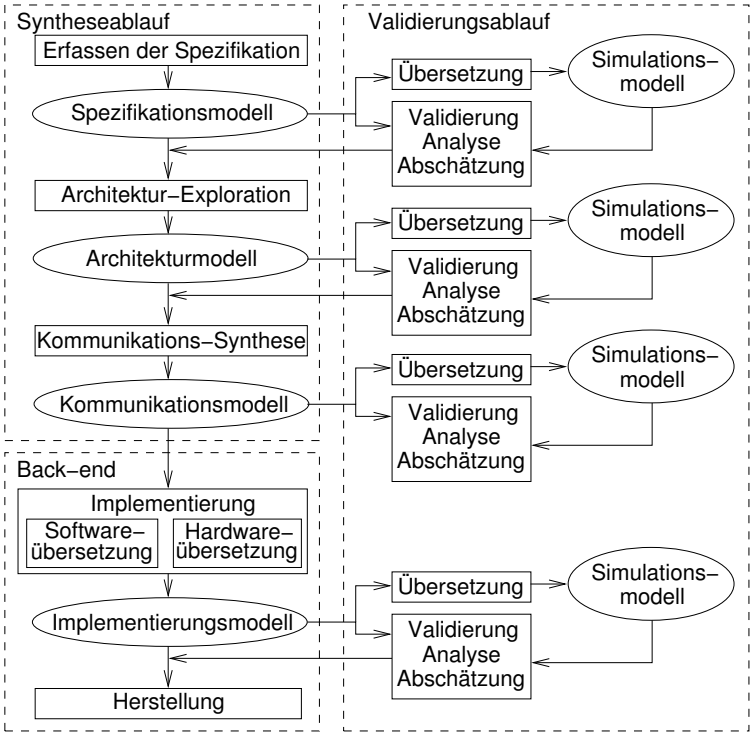


Abb. 6.2. Mögliche Codesign-Methodik bei Verwendung von SpecC

Die Methodik beginnt mit der Spezifikation, die in SpecC festgehalten wird. Das SpecC-Modell ist ausführbar. Folglich können Simulationsläufe zur Evaluierung und Validierung des Modells sowie zur Abschätzung wichtiger Entwurfsparameter verwendet werden. Der nächste Schritt besteht in der Erkundung des Architektur-Entwurfsraumes, bestehend aus Allokation, Partitionierung und Scheduling. Die Allokation besteht aus der Auswahl geeigneter Komponenten (Verarbeitungseinheiten (Prozessoren, IP-Komponenten oder anwendungsspezifische Hardware), Speicher, Busse) aus einer Bibliothek. Die Partitionierung bestimmt eine Zuordnung von Teilen der Systemspezifikation auf die Komponenten. Variablen werden Speichern, Kanäle werden Bussen und Verhalten werden Verarbeitungseinheiten zugeordnet. Das Scheduling dient der Serialisierung der Ausführung. Abbildung 6.2 beschreibt den Fluss der Informationen. Der tatsächliche Entwurf besteht aus einer Vielzahl von Schritten, die mit diesem Informationsfluss konsistent sind. Nach der Erkundung der Architekturparameter erfolgt eine Evaluierung bzw. Abschätzung von Eigenschaften und Validierung des Entwurfs.

Bei der Synthese von Kommunikationsbestandteilen werden gemäß SpecC Methodik abstrakte Busse in einer Reihe von Verbesserungen auf physikalische Leitungen abgebildet. Im sogenannten *Backend* erzeugen Software-Compiler den binären Maschinencode, während Hardware-Synthese-Werkzeuge verwendet werden, um Spezialhardware zu erzeugen.

Allgemein wäre es schön, wenn eine einzige Validierungstechnik auf alle Evaluierungsprobleme angewendet werden könnte. In der Praxis löst aber keine der existierenden Techniken alle Probleme, und somit muss man auf eine Kombination verschiedener Techniken zurückgreifen.

6.2 Pareto-Optimalität

Das Ergebnis der Evaluation besteht in der Regel aus einer Charakterisierung durch mehrere Kriterien, wie z.B. die mittlere und die maximale Laufzeit, den Energieverbrauch, die Codegröße, die Zuverlässigkeit und die Sicherheit. Im Allgemeinen ist es nicht sinnvoll, diese Größen in einer einzigen Zielfunktion zusammenzufassen. Vielmehr sind Designer meist daran interessiert, eine Menge von möglichen Entwürfen zu überblicken und aus einer Menge dann einen geeigneten Entwurf herauszusuchen. Diese Menge sollte allerdings nur „sinnvolle“ Entwürfe enthalten. Mengen „sinnvoller“ Entwürfe zu finden ist das Ziel der **multikriteriellen Optimierung**.

Bei der multikriteriellen Optimierung betrachten wir einen m-dimensionalen Raum X der möglichen Lösungen eines Entwurfsproblems. Auf diesem Raum ist eine Vektorfunktion $f(x) = (f_1(x), ..., f_n(x))$ mit $x \in X$ definiert, welche einen konkreten Entwurf jeweils hinsichtlich unterschiedlicher Kriterien bewertet. Sei F der n-dimensionale Raum der Werte der Kriterien für mögliche

Lösungen des Entwurfsproblems (*objective space*). Bezüglich jedes Kriteriums sei eine Ordnung $<$ bzw. $\leq$ definiert. Wir gehen nachfolgend davon aus, dass wir die *Minimierung* der Kriterien betrachten.

Definition: Ein Vektor $u = (u_1, ..., u_n) \in F$ **dominiert** einen Vektor $v = (v_1, ..., v_n) \in F$ genau dann, wenn u hinsichtlich mindestens eines Kriteriums „besser“ ist als v und hinsichtlich aller übrigen Kriterien zumindest nicht schlechter, d.h. wenn gilt

$$\forall i \in \{1, ...n\} : u_i \leq v_i \quad \wedge \tag{6.1}$$

$$\exists i \in \{1, .., n\} : u_i < v_i \tag{6.2}$$

Definition: Ein Vektor $u \in F$ heißt indifferent zu einem Vektor $v \in F$ wenn kein Vektor den anderen dominiert.

Definition: Eine Lösung $x \in X$ heißt **Pareto-optimal** in Bezug auf X genau dann, wenn es keine Lösung $y \in X$ gibt, so dass $u = f(x)$ durch $v = f(y)$ dominiert wird.

Als Anwendungsbeispiel betrachten wir das *Matador/Task Concurrency Management* (TCM)-Projekt am *Interuniversitair Micro-Electronica Centrum* (IMEC) in Leuven in Belgien. Hier liegt der Schwerpunkt auf der Abbildung von Tasks auf Prozessoren. Verschiedene Konfigurationen von Multi-Prozessor-Systemen werden evaluiert und als Mengen Pareto-optimaler Lösungen dargestellt. Wong et al. [Wong et al., 2001] beschreiben Konfigurationen für ein MPEG-4-Abspielgerät. Die Autoren nehmen an, dass Kombinationen von Strong-Arm-Prozessoren und spezialisierten Beschleunigern verwendet werden sollen. Es werden vier Lösungen präsentiert, welche die Zeitbedingungen von 30 ms erfüllen (s. Tabelle 6.1).

Prozessor-Kombination	1	2	3	4
Anzahl von Beschleunigern	6	5	4	3
Anzahl langsamer Prozessoren	0	3	5	7
Insgesamte Anzahl Prozessoren	6	8	9	10

Tabelle 6.1. Prozessor-Konfigurationen

Bei den Kombinationen 1 und 4 gibt es nur eine Zuordnung von Tasks zu Prozessoren, welche die Zeitbedingungen einhalten kann. Bei den Kombinationen 2 und 3 führen unterschiedliche Zeitbedingungen zu verschiedenen Abbildungen von Tasks auf Prozessoren und zu unterschiedlichen Energieverbräuchen.

Die **Erkundung des Entwurfsraumes** basiert auf dem Konzept der Pareto-Optimalität, wie in Abb. 6.3 für die Konfigurationen 2 und 3 gezeigt. Ober-

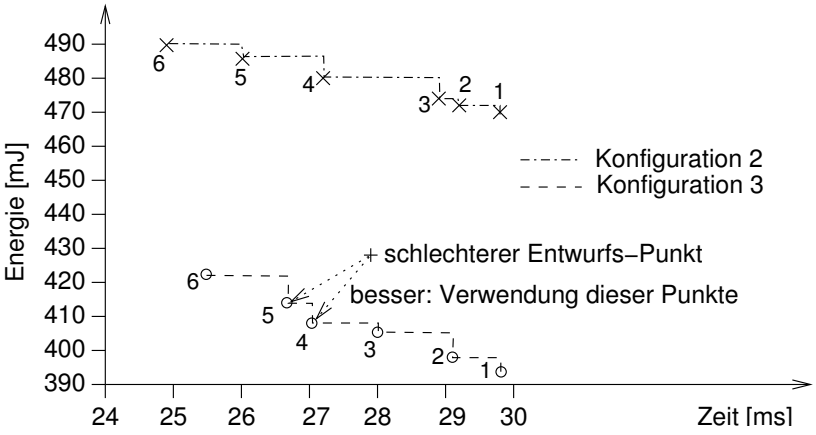


Abb. 6.3. Pareto-Kurven für die Prozessor-Kombinationen 2 und 3

halb der beiden Linien befinden sich die dominierten Lösungen. Beispielsweise entspricht der Bereich oberhalb der unteren, gestrichelten Linie (also z.B. der mit + gekennzeichnete Punkt) den Punkten im Entwurfsraum, die durch die Lösungen auf der gestrichelten Linie dominiert werden. Bei allen Entwürfen in diesem Bereich kann man entweder die Geschwindigkeit, den Energieverbrauch oder sogar beides verbessern, indem man die Lösungen auf der unteren Linie benutzt. Daher werden alle Lösungen in diesem Gebiet, die bei einer Abbildung von Tasks auf Prozessoren gefunden werden, ignoriert. Die obere Treppenfunktion entspricht der Konfiguration 2. Beim TCM-Ansatz des IMEC werden die Pareto-optimalen Lösungen in der Systemimplementierung gespeichert. Zur Laufzeit wird dann eine Lösung aufgrund des noch verfügbaren Schlupfes ausgewählt. Ist nur noch wenig Schlupf verfügbar, so muss viel Energie für die Berechnungen aufgewandt werden. Ist noch ein großer Schlupf vorhanden, dann kann Energie eingespart werden. Auf diese Weise kann berücksichtigt werden, dass die Laufzeit von Tasks oder Jobs in der Praxis nicht konstant ist. Das Beispiel zeigt sehr schön den Vorteil des Verzichts auf die Auswahl **einer** „optimalen“ Lösung zur Entwurfszeit. Durch die Verschiebung der Entscheidung in die Laufzeit kann bei der Entscheidung zusätzlich Information genutzt und Energie eingespart werden. Gleichzeitig kann ggf. durch die Auswahl einer schnellen Lösung noch eine Realzeitbedingung eingehalten werden.

6.3 Simulation

Simulationen sind eine sehr weitverbreitete Technik für die Validierung von Entwürfen. Eine Simulation besteht aus der Ausführung eines Entwurfsmodells auf einer geeigneten Hardware, üblicherweise auf digitalen Standard-

Computern. Bei diesem Ansatz muss das zu validierende Modell ausführbar sein. Alle in Kapitel 2 vorgestellten ausführbaren Sprachen können zur Durchführung von Simulationen verwendet werden, und zwar auf verschiedenen Abstraktionsebenen (wie auf Seite 87 beschrieben).

Die Abstraktionsebene, auf der simuliert wird, ist stets ein Kompromiss zwischen der Simulationsgeschwindigkeit und der Simulationsgenauigkeit. Je schneller die Simulation, desto weniger genau ist sie.

Bisher haben wir den Begriff „Verhalten“ im Sinne des funktionalen Verhaltens des Systems (also seiner Ein-/Ausgabe-Funktionalität) verwendet. Es gibt auch Simulationen von anderen, nicht-funktionalen Verhaltensweisen von Entwürfen, etwa des thermischen Verhaltens oder der elektromagnetischen Verträglichkeit (*Electro-Magnetic Compatibility* (EMC)) mit anderen elektronischen Geräten.

Für die Validierung eingebetteter Systeme haben Simulationen einige schwerwiegende Einschränkungen:

- Simulationen sind typischerweise viel langsamer als der tatsächliche Entwurf. Wenn man also versucht, den Simulator mit der tatsächlichen Umgebung des eingebetteten Systems zu verbinden, würden viele **verletzte Zeitbedingungen** auftreten.
- Simulationen in der realen Umgebung können sogar **gefährlich** sein - wer würde schon ein Flugzeug mit instabiler, ungetesteter Steuerungssoftware fliegen wollen?
- Viele Anwendungen verwenden sehr große Datenmengen, und es ist in der Regel unmöglich, in akzeptabler Zeit eine ausreichende Abdeckung dieser Daten durch Simulation zu erzielen. Multimedia-Anwendungen sind typische Vertreter für diesen Effekt. So benötigt etwa die Simulation der Kompression eines Videostroms sehr viel Zeit.
- Die meisten praktisch eingesetzten Systeme sind heutzutage zu komplex, um alle möglichen Fälle (Eingaben) simulieren zu können. Daher können Simulationen zwar helfen, Fehler im Entwurf zu entdecken, sie können aber die Fehlerfreiheit des Systems nicht garantieren. Es ist nicht möglich, alle Kombinationen von Eingaben und internen Zuständen zu simulieren.

Aufgrund dieser Einschränkungen gewinnt die formale Verifikation zunehmend an Bedeutung (s. Seite 233).

6.4 Rapid Prototyping und Emulation

Häufig sollen sich Entwürfe in realistischen Umgebungen bewähren, bevor ihre endgültige Version hergestellt wird. Steuerungssysteme in Autos sind

hierfür ein hervorragendes Beispiel. Solche Systeme sollten von Fahrern in unterschiedlichen Umgebungen verwendet werden, bevor die Massenproduktion anläuft. Aus diesem Grund verwendet die Automobilindustrie Entwurfs-Prototypen. Diese Prototypen verhalten sich im Wesentlichen wie das endgültige System, aber sie dürfen z.B. größer sein, mehr Energie verbrauchen und andere Eigenschaften aufweisen, die ein Testfahrer noch akzeptieren kann. Solche Prototypen können z.B. mithilfe von FPGAs gebaut werden. Einschubschränke mit mehreren FPGAs werden vor der Testfahrt im Kofferraum des Wagens untergebracht.

Dieser Ansatz ist nicht auf die Automobilindustrie beschränkt. Es gibt andere Einsatzgebiete, in denen Prototypen auf Basis von FPGAs verwendet werden. Kommerziell verfügbare **Emulatoren** bestehen aus einer großen Anzahl FPGAs. Diese werden mit einer speziellen *Mapping*-Software ausgeliefert, die eine Spezifikation auf diese FPGAs abbilden kann. Mithilfe solcher Emulatoren können Experimente mit Systemen durchgeführt werden, die sich beinahe wie die endgültigen Produkte verhalten.

Auch Emulationen können nicht garantieren, dass alle Grenzfälle abgedeckt werden. Daher sollten Analysemethoden eingesetzt werden, die derartige Fälle erfassen.

6.5 Leistungsbewertung

Die Leistungsanalyse hat zum Ziel, präzise Methoden zur Vorhersage der Leistung eines Systems bereitzustellen. Dies ist eine erhebliche Herausforderung, da es für eingebettete Systeme nicht ausreicht, Angaben über ein durchschnittliches Verhalten zu machen. Vielmehr muss insbesondere bei Realzeitsystemen auch das Einhalten der Zeitbedingungen überprüft werden.

Es gibt eine Vielzahl von Ansätzen zur Leistungsbewertung in frühen Entwurfsphasen. Für die Bestimmung erwarteter Leistungs- (und auch Kosten-) Werte wurden u.a. folgende Ansätze vorgeschlagen:

- **Geschätzte Kosten- und Leistungswerte:** Zu diesem Zweck existiert eine große Anzahl von Methoden zur Leistungsbewertung auf der Basis früher, unvollständiger Entwurfsinformationen. Beispiele sind etwa die Arbeiten von Jha und Dutt [Jha und Dutt, 1993] für Hardware, und von Jain et al. [Jain et al., 2001] für Software. Das Erzeugen ausreichend genauer Schätzungen ist sehr aufwendig.
- **Genaue Kosten- und Leistungswerte:** Die Bestimmung genauer Werte ist möglich, wenn aus den frühen Entwurfsphasen heraus prototypische Entwürfe erstellt werden. Dies ist nur möglich, wenn Schnittstellen zu schnellen *Software-Synthese-Tools* (also zu Compilern) und zu schnellen Hardware-Synthese-Tools vorhanden sind. Diese Methode kann zwar ge-

nauer sein als die oben vorgestellte, kann aber dafür auch bedeutend mehr Zeit benötigen (was dazu führen kann, dass diese Technik nicht praktisch einsetzbar ist).

Um gute Schätzungen zu erhalten, muss auch die Kommunikation berücksichtigt werden. Leider ist es sehr schwierig, in frühen Entwurfsphasen bereits die Kommunikationskosten zu bestimmen.

Formale Techniken zur Leistungsbewertung von eingebetteten Systemen wurden u.a. Thiele et al., von Ernst et al. und Wilhelm et al. vorgeschlagen. Diese Techniken bewerten relativ genau bekannte Entwürfe. Insbesondere müssen die Hardwarekomponenten bekannt sein.

Ausgangspunkt der Leistungsanalyse nach dem von Thiele et al. entwickelten ***real time calculus***[1] ist eine Beschreibung der Ankünfte von Ereignissen, die eine Bearbeitung in unserem System erforderlich machen. Dabei sind Fluktuationen zu berücksichtigen. Ankünfte werden daher beschrieben durch Kurven (*arrival curves*) $\overline{\alpha}^u(\Delta), \overline{\alpha}^l(\Delta) \in I\!R \geq 0, \Delta \in I\!R \geq 0$, die jeweils die maximale bzw. die minimale Anzahl von Ereignissen beschreiben, die in einem Intervall der Länge Δ eingehen. Es gibt also höchstens $\overline{\alpha}^u(\Delta)$ und mindestens $\overline{\alpha}^l(\Delta)$ eingehende Ereignisse in einem Intervall $[t, t + \Delta)$ für alle $t \geq 0$. Abb. 6.4 beschreibt die Zahl möglicher Ereignisankünfte für mögliche Modelle von Ereignisströmen.

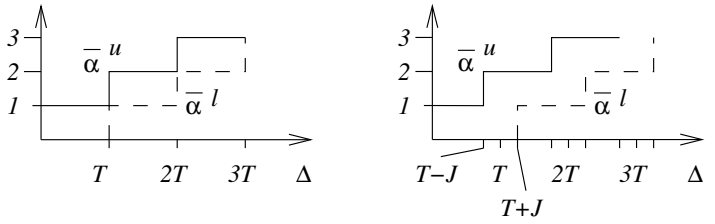


Abb. 6.4. Ankünfte (*arrival curves*)

Bei periodischen Ereignisströmen mit einer Periode T gibt es beispielsweise in einem Zeitintervall von 1,5 T mindestens ein und höchstens zwei ankommende Ereignisse (s. Abb. 6.4 (links)). Bei periodischen Ereignisströmen mit einem Jitter J verschieben sich die Kurven um diesen Betrag J (s. Abb. 6.4 (rechts)). Wir benutzen überstrichene Bezeichnungen wie $\overline{\alpha}$ für alle Größen, die sich auf die eingehenden Ereignisse beziehen.

Auf ähnliche Weise wird die zur Verfügung stehende Bearbeitungs- und Übertragungsleistung durch *service functions* $\beta^u(\Delta), \beta^l(\Delta) \in I\!R \geq 0, \Delta \in I\!R \geq 0$ beschrieben. Damit wird modelliert, dass die Bearbeitungsleistung im Laufe

[1] Die Darstellung des *real time calculus* in diesem Buch richtet sich nach dessen Vorstellung in [Thiele, 2006].

der Zeit schwanken kann. Abb. 6.5 charakterisiert die Übertragungsleistung eines Busses, der nach dem *time division multiple access* (TDMA)-Prinzip immer nur innerhalb gewisser Zeitintervalle s für eine Übertragung genutzt werden kann und dann eine Bandbreite b bietet. Während des Zeitfensters s steigt der Umfang an übertragbaren Informationen linear an.

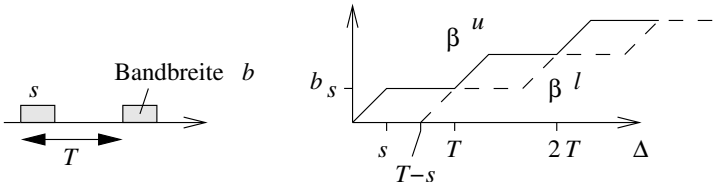


Abb. 6.5. Bearbeitungs- bzw. Übertragungsleistung (*service functions*)

Die Bestimmung der Funktionen $\overline{\alpha}$ und β muss mit den jeweils anwendbaren Techniken erfolgen. Diese Aufgabe wird nicht durch den *real time calculus* selbst gelöst.

Bislang fehlt noch die Information, welchen Bearbeitungsbedarf (*Workload*) ein eintreffendes Ereignis erfordert. Die notwendige Bearbeitungsleistung wird im *real time calculus* durch weitere Funktionen $\gamma^u(e), \gamma^l(e) \in R \geq 0$ für jede Folge von e Ereignissen charakterisiert. Diese Information kann z.B. aus den Schranken für die Ausführungszeit eines Jobs bestimmt werden. Abb. 6.6 zeigt ein Beispiel für diese Funktionen. Dabei wurde angenommen, dass pro eingehendem Event drei bis vier Rechenzeiteinheiten zur Bearbeitung erforderlich sind.

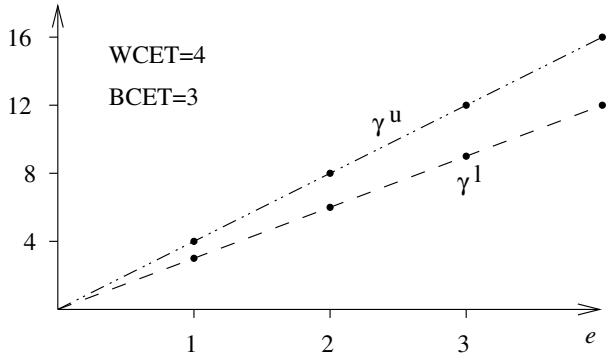


Abb. 6.6. *Workload*-Charakterisierung

Aus diesen Informationen können wir jetzt relativ leicht die *Workload* bestimmen, die für einen eingehenden Ereignisstrom erforderlich ist. Obere und untere Schranken sind charakterisiert durch die Funktionen

$$\alpha^u(\Delta) = \gamma^u(\overline{\alpha}^u(\Delta)) \text{ und } \alpha^l(\Delta) = \gamma^l(\overline{\alpha}^l(\Delta))$$

Umgekehrt ergeben sich die minimal bzw. maximal verarbeitbaren Ereigniszahlen durch

$$\overline{\beta}^u(\Delta) = \gamma^{l\ -1}(\beta^u(\Delta)) \text{ und } \overline{\beta}^l(\Delta) = \gamma^{u\ -1}(\beta^l(\Delta))$$

Mit dieser Information kann bestimmt werden, wie Realzeit-Komponenten einen eingehenden Ereignisstrom $[\overline{\alpha}^l, \overline{\alpha}^u]$ in einen ausgehenden Ereignisstrom $[\overline{\alpha}^{l'}, \overline{\alpha}^{u'}]$ transformieren. Ebenso kann bestimmt werden, welche Verarbeitungsleistung noch für andere Aufgaben bereitsteht. Diese verbleibende Verarbeitungsleistung ergibt sich durch die Transformation der *service curves* $[\overline{\beta}^l, \overline{\beta}^u]$ in *service curves* $[\overline{\beta}^{l'}, \overline{\beta}^{u'}]$ (s. Abb. 6.7). Diese verbleibende Verarbeitungsleistung steht z.B. für Aufgaben zur Verfügung, die auf demselben Prozessor in Tasks mit niedrigerer Priorität gelöst werden.

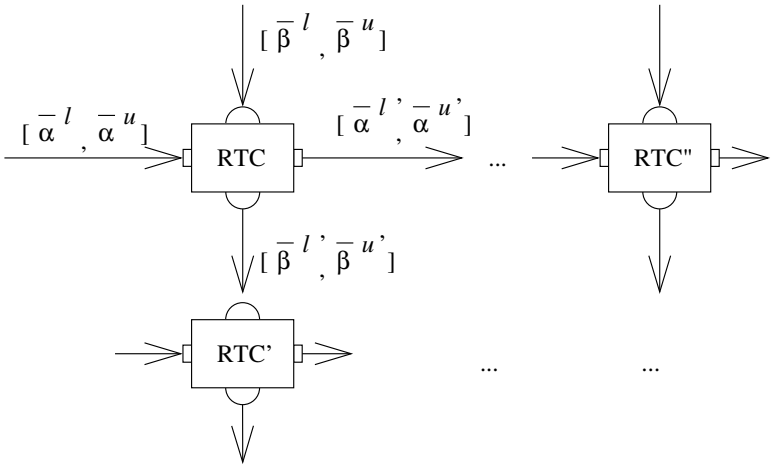


Abb. 6.7. Transformation von Ereignisströmen und Bearbeitungskapazität durch Realzeitkomponenten

Thiele et al. geben an, wie die ausgehenden Ereignisströme und die verbleibende Bearbeitungsleistung ausgerechnet werden können [Thiele, 2006]:

$$\overline{\alpha}^{u'} = [(\overline{\alpha}^u \underline{\otimes} \overline{\beta}^u) \overline{\oslash} \overline{\beta}^l] \wedge \overline{\beta}^u \tag{6.3}$$

$$\overline{\alpha}^{l'} = [(\overline{\alpha}^{l} \underline{\oslash} \overline{\beta}^{u}) \underline{\otimes} \overline{\beta}^{l}] \wedge \overline{\beta}^{l} \tag{6.4}$$

$$\overline{\beta}^{u'} = (\overline{\beta}^{u} - \overline{\alpha}^{l}) \underline{\oslash} 0 \tag{6.5}$$

$$\overline{\beta}^{l'} = (\overline{\beta}^{l} - \overline{\alpha}^{u}) \overline{\otimes} 0 \tag{6.6}$$

Dabei sind die Operatoren definiert durch:

$$(f \underline{\otimes} g)(t) = inf_{0 \le u \le t}\{f(t-u) + g(u)\} \tag{6.7}$$

$$(f \overline{\otimes} g)(t) = sup_{0 \le u \le t}\{f(t-u) + g(u)\} \tag{6.8}$$

$$(f \overline{\oslash} g)(t) = sup_{u \ge 0}\{f(t+u) - g(u)\} \tag{6.9}$$

$$(f \underline{\oslash} g)(t) = inf_{u \ge 0}\{f(t+u) - g(u)\} \tag{6.10}$$

und $\wedge$ bezeichnet den Minimum-Operator.

Auch können die maximale Verzögerung und die Größe des Pufferspeichers ausgerechnet werden. Auf diese Weise können die Leistung und weitere charakteristische Parameter eines Gesamtsystems aus den Informationen über die Komponenten ausgerechnet werden.

Ein zweiter, einflussreicher Ansatz zur Leistungsbewertung ist der SymTA/S-Ansatz von Ernst et al. [Henia et al., 2005]. Im Falle von SymTA/S werden statt der beliebigen Ereignisströme bei Thiele nur sogenannte Standard-Ereignisströme zugelassen. Zu den Standardereignisströmen zählen insbesondere periodische Ereignisströme, periodische Ereignisströme mit Jitter und periodische Ereignisströme mit Bursts. Ergebnisse der Scheduling-Theorie für Standardereignisströme können so direkt genutzt werden. Dabei werden zunächst die Ausgangs-Ereignisströme für alle Komponenten bestimmt, die direkt mit den Systemeingängen verbunden sind. In einem nächsten Schritt werden diese Ströme zu den nächsten Eingängen propagiert. Diese Schritte werden wiederholt, bis alle Ereignisströme bekannt und konsistent sind.

Sowohl der *real time calculus* wie auch SymTA/S benötigen Aussagen zu den Ausführungszeiten auf den einzelnen Verarbeitungseinheiten. Die Berechnung einer oberen Schranke für die Ausführungszeit von Programmen auf Prozessoren ist im Allgemeinen nicht entscheidbar, da es nicht entscheidbar ist, ob ein Programm terminiert oder nicht. Daher kann die Schranke nur für bestimmte Programme oder Tasks berechnet werden. Beispielsweise kann die Schranke für nicht-rekursive Programme mit *for*-Schleifen und konstanten Schleifengrenzen berechnet werden.

Die Berechnung einer möglichst genauen oberen Schranke der Ausführungszeit kann trotzdem schwierig sein. Die in modernen Prozessoren verwendeten *Pipelines* mit ihren verschiedenen *Hazards* sowie Speicherhierarchien mit schwieriger Treffer-Vorhersagbarkeit sind bei der Bestimmung einer präzisen oberen Schranke hinderlich und führen häufig zu einer deutlichen Überschätzung der WCET. Einige dieser Architekturmerkmale reduzieren die durchschnittliche

Ausführungszeit, können aber keine Verbesserung der WCET garantieren. Diese Architekturbestandteile werden in der Regel beim Entwurf von eingebetteten Systemen gar nicht erst ausgewählt (s. Seite 120). Die Berechnung von WCET-Schranken von Systemen mit *Pipelines* und Caches ist ein Forschungsthema (s. beispielsweise [Healy et al., 1999] und die Webseiten von AbsInt [Absint, 2002]). Unterbrechungen (*Interrupts*) und virtueller Speicher sorgen für weitere Komplikationen. Wie man sieht, ist es schon schwierig, anhand eines Assembler-Programms dessen WCET zu bestimmen. Gute obere Schranken für Programme zu bestimmen, die in einer Hochsprache wie etwa C geschrieben sind, ist ohne Kenntnis des generierten Assembler-Codes unmöglich.

Nachfolgend werden wir uns etwas genauer mit der Analyse des Cache-Verhaltens beschäftigen[2]. Wir betrachten dazu einen n-fach mengenassoziativen Cache wie in Abb. 6.8 dargestellt[3].

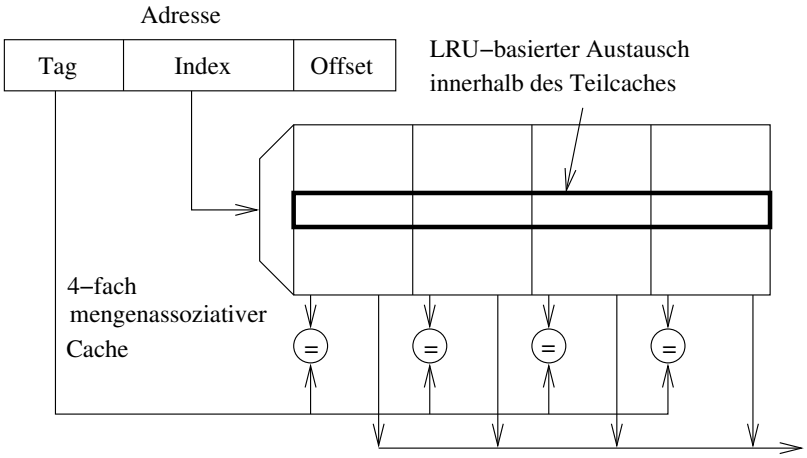


Abb. 6.8. Mengenassoziativer Cache (für n=4)

Wir betrachten den Teilcache, der zu einem bestimmten Index gehört (in Abb. 6.8 fett dargestellt). Wir setzen voraus, dass jeder derartige Teilcache die *least recently used* (LRU)-Verdrängungsstrategie nutzt. Die bedeutet, dass unter allen Einträgen mit einem Index immer die jeweils zuletzt benutzten n Einträge im Teilcache gespeichert werden. Wesentlich ist, dass für die verschiedenen Indizes unabhängige Hardware zur Buchführung über das Alter der Einträge vorhanden ist. Unter dieser Voraussetzung kann die Analyse des

[2] Die hier gegebene Darstellung basiert auf der Beschreibung von R. Wilhelm [Wilhelm, 2006].

[3] Wir setzen hier wie in Kapitel 1 beschrieben voraus, dass Studierende aufgrund vorheriger Kurse mit dem grundsätzlichen Aufbau von Caches vertraut sind.

Cache-Verhaltens für die einzelnen Teilcaches, die zu einem Index gehören, unabhängig voneinander vorgenommen werden, da sich diese untereinander nicht beeinflussen. Ist eine Sequenz von Adressen von Speicherzugriffen gegeben, so müssen zur Analyse des Cacheverhaltens nur jene Speicheradressen betrachtet werden, deren Index einem gerade betrachteten Teilcache entspricht. Betrachten wir nunmehr einen einzelnen Teilcache und nehmen an, dass wir Informationen über die Inhalte der Spalten innerhalb eines Teilcaches haben. Nehmen wir weiterhin an, dass wir rekonvergente Programmpfade (*joins*) haben und bestimmen wollen, wie aus den Informationen über Cacheinhalte vor der Rekonvergenz Informationen über Cacheinhalte nach der Rekonvergenz bestimmt werden können. Dabei müssen wir unterscheiden zwischen der ***may***-Analyse und der ***must***-Analyse. Die ***must***-Analyse gibt Auskunft darüber, welche Informationen sich sicher im Cache befinden. Sie ist geeignet, Zusicherungen bei der Bestimmung von WCET-Schranken zu geben. Die *may*-Analyse resultiert in Aussagen darüber, welche Informationen sich möglicherweise im Cache befinden könnten. Diese Aussagen sind wichtig, um zu wissen, was sich mit Sicherheit nicht im Cache befindet. Mit diesem Wissen können wir BCET-Schranken bestimmen. Betrachten wir zunächst die ***must***-Analyse für rekonvergente Programmpfade. Abbildung 6.9 zeigt eine entsprechende Situation. Das Alter der Einträge wachse innerhalb eines Rechtecks von links nach rechts.

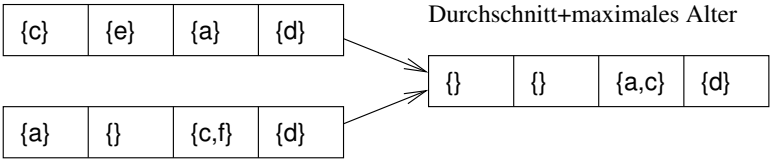


Abb. 6.9. *Must*-Analyse für LRU-Caches bei rekonvergenten Pfaden

Das Speicherobjekt `c` sei das jüngste Element in der Cachezeile sofern wir über einen Programmpfad zur Rekonvergenz gelangen und `a` sei das jüngste Element sofern wir über den anderen Pfad dorthin gelangen. Entsprechendes gilt für die älteren Einträge im Cache. Wir wollen jetzt im Kontext der *must*-Analyse bestimmen, was der „schlechteste“ Fall nach der Rekonvergenz ist. Offensichtlich können wir nach der Rekonvergenz im Cache mit Sicherheit nur solche Speicherobjekte finden, die sich im Durchschnitt der beiden ursprünglichen Cacheinhalte befanden. Als Alter müssen wir im Sinne der *worst case*-Analyse das maximale Alter annehmen. Abb. 6.9 zeigt das Ergebnis. Offensichtlich muss die Analyse für jeden Platz (jede Spalte) im Cache mit Mengen möglicher Einträge arbeiten.

Betrachten wir nun die *may*-Analyse für rekonvergente Programmpfade. Abb. 6.10 zeigt wiederum die Situation.

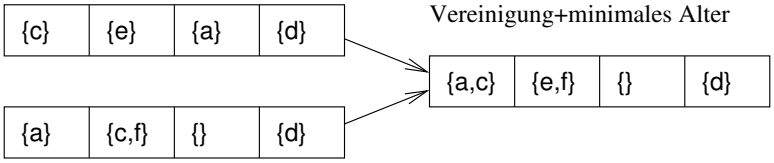


Abb. 6.10. *May*-Analyse für LRU-Caches bei rekonvergenten Pfaden

Im resultierenden Cache können nunmehr offensichtlich Speicherobjekte vorhanden sein, die vor der Rekonvergenz in einem der beiden Programmpfade vorhanden waren. Also müssen wir die Vereinigung der Speicherobjekte betrachten. Im bestmöglichen Fall müssen wir von dem jüngsten Alter der Speicherobjekte ausgehen. Abb. 6.10 zeigt das Ergebnis.

Die hier gezeigte Cache-Analyse bildet allerdings nur einen kleinen Teil der Analysen, die zur Bestimmung guter WCET- und BCET-Schranken erforderlich sind. Der Artikel von Wilhelm [Wilhelm, 2006] enthält eine genauere Beschreibung der notwendigen Analysen.

6.6 Bewertung des Energieverbrauchs

Energiemodelle sind grundlegender Bestandteil aller Energieoptimierungen. Ein allgemeines Problem solcher Energiemodelle ist ihre häufig nur sehr begrenzte Genauigkeit[4].

- Eines der ersten Energiemodelle wurde von Tiwari [Tiwari et al., 1994] vorgestellt. Das Modell enthält sogenannte Basiskosten und Inter-Instruktions-Kosten. Basiskosten einer Instruktion entsprechen dem Energieverbrauch, der bei jeder Ausführung einer Instruktion entsteht, wenn die Instruktion in einer endlosen Folge immer wieder ausgeführt wird. Inter-Instruktions-Kosten modellieren die zusätzlich entstehenden Kosten, wenn unterschiediche Instruktionen ausgeführt werden. Diese zusätzliche Energie wird beispielsweise durch das Aktivieren und Deaktivieren von Funktionseinheiten verursacht. Dieses Energiemodell konzentriert sich auf den Energieverbrauch im Prozessor und vernachlässigt die Energie, die im Speicher oder in anderen Teilen des Systems verbraucht wird.
- Ein weiteres Modell stammt von Simunic et al. [Simunic et al., 1999]. Es basiert auf Datenblättern. Der Vorteil dieses Ansatzes ist die Möglichkeit, den Energiebeitrag aller Komponenten eines eingebetteten Systems berücksichtigen zu können. Allerdings sind die in Datenblättern angegebenen Durchschnittswerte weniger genau als Angaben über minimale und maximale Werte.

[4] In Diskussionen ist häufig von Abweichungen bis zu 50% die Rede.

- Ein drittes Modell wurde von [Russell und Jacome, 1998] vorgestellt. Dieses Modell basiert auf präzisen Messungen zweier fester Konfigurationen. Die Ergebnisse sind für diese Konfigurationen sehr genau, erlauben aber keine Aussagen über die Wirkung von Änderungen am Entwurf.
- Ein weiteres Modell wurde von Lee [Lee et al., 2001] vorgeschlagen: es enthält eine detaillierte Analyse der *Pipeline*-Effekte. Es werden allerdings keine Mehrzyklen-Instruktionen und keine *Pipeline-Stalls* berücksichtigt.
- Der encc (*energy-aware C-compiler*) der Universität Dortmund verwendet das Energiemodell von Steinke et al. [Steinke et al., 2001a]. Es basiert auf genauen Messungen echter Hardware. Sowohl der Energieverbrauch des Prozessors als auch der des Speichers sind in diesem Modell enthalten.
- Der Energieverbrauch von Caches läßt sich mit Hilfe des CACTI-Werkzeugs [Wilton und Jouppi, 1996] bestimmen.

Aus diesen Beispielen wird das generelle Problem deutlich: für eine gegebene, konkrete Hardware kann man präzise Aussagen zum Energieverbrauch machen. Will man aber die Wirkung von Änderungen am Entwurf beurteilen, so ist man auf Modelle angewiesen, deren Genauigkeit sehr leicht angezweifelt werden kann.

6.7 Risiko- und Verlässlichkeits-Analyse

Eingebettete Systeme können (wie viele Produkte) Sach- und Personen-Schäden hervorrufen. Es ist nicht möglich, das Risiko solcher Schäden auf Null zu reduzieren. Man kann lediglich versuchen, die Wahrscheinlichkeit von Schäden so gering wie möglich zu halten, am Besten einige Zehnerpotenzen geringer als andere Risiken. Für viele Anwendungen muss die Wahrscheinlichkeit einer Katastrophe unterhalb von 10^{-9} pro Stunde liegen [Kopetz, 1997], was einem einzigen Zwischenfall bei 100.000 Systemen mit einer Betriebszeit von jeweils 10.000 Stunden entspricht.

Werden derartige Anforderungen an die Zuverlässigkeit gestellt, so muss im Rahmen der Evaluation auch versucht werden, Aussagen über erwartete Ausfallraten, Lebensdauern und ähnliche Größen zu bestimmen. Grundlage der theoretischen Berechnungen hierzu sind in der Regel die Wahrscheinlichkeitsverteilungen von Fehlerereignissen. Im Folgenden bezeichne x die Zeit bis zum ersten Ausfall eines Systems. x ist eine Zufallsvariable. Sei $f(x)$ die Dichtefunktion dieser Zufallsvariablen. Eine sehr häufig benutzte Dichtefunktion ist dabei die Exponentialverteilung mit der Dichtefunktion $f(x) = \lambda e^{-\lambda x}$. Bei dieser Dichtefunktion nehmen die Systemausfälle mit fortschreitender Zeit immer mehr ab. Aus Gründen der einfachen mathematischen Handhabbarkeit benutzt man diese Dichtefunktion als Referenz auch dann, wenn sie die exakte Situation nicht widerspiegelt oder wenn zunächst eine grobe Übersicht über

die Verhältnisse gewünscht wird. Abb. 6.11 (links) zeigt die Dichtefunktion der Exponentialverteilung.

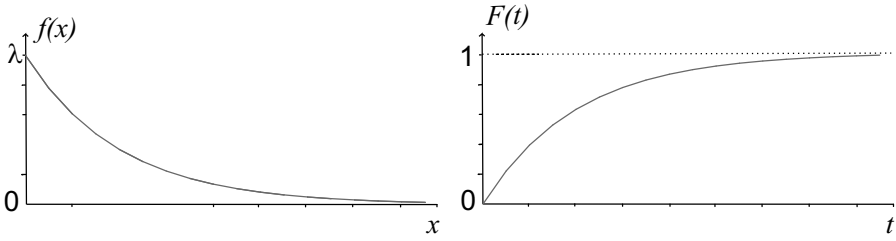


Abb. 6.11. Dichtefunktion und Verteilungsfunktion bei Exponentialverteilung

In vielen Fällen ist man mehr an den Wahrscheinlichkeiten für die Funktionsfähigkeit des Systems interessiert als an den Dichtefunktionen. Allgemein ergibt sich die Wahrscheinlichkeit dafür, dass ein System zum Zeitpunkt t fehlerhaft ist, durch Integration über die Dichtefunktion. Die so bestimmte Funktion ist die **Verteilungsfunktion**:

$$F(t) = Pr(x \leq t) \tag{6.11}$$

$$F(t) = \int_0^t f(x)dx \tag{6.12}$$

Für die Exponentialverteilung ergibt sich beispielsweise

$$F(t) = \int_0^t \lambda e^{-\lambda x} dx = -[e^{-\lambda x}]_0^t = 1 - e^{-\lambda t} \tag{6.13}$$

Abb. 6.11 (rechts) zeigt die entsprechende Funktion. Mit fortschreitender Zeit nähert sich die Wahrscheinlichkeit, dass das System fehlerhaft ist, dem Wert 1.

Definition: Unter der **Zuverlässigkeit** $R(t)$ eines Systems verstehen wir die Wahrscheinlichkeit dafür, dass die Zeit bis zum ersten Fehler größer ist als eine Zeit t:

$$R(t) = Pr(x > t), t \geq 0 \tag{6.14}$$

$$R(t) = \int_t^\infty f(x)dx \tag{6.15}$$

$$F(t) + R(t) = \int_0^t f(x)dx + \int_t^\infty f(x)dx = 1 \tag{6.16}$$

$$R(t) = 1 - F(t) \tag{6.17}$$

$$f(x) = -\frac{dR(t)}{dt} \tag{6.18}$$

Für die Exponentialverteilung ergibt sich $R(t) = e^{-\lambda t}$. Abb. 6.12 zeigt diese Funktion. Nach einer Zeit $t = 1/\lambda$ beträgt die Wahrscheinlichkeit, dass das System funktioniert, noch ca. 37%.

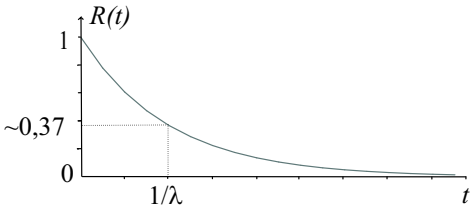


Abb. 6.12. Zuverlässigkeit bei Exponentialverteilung

Definition: Die **Fehlerrate** $\lambda(t)$ ist die Wahrscheinlichkeit dafür, dass ein System im Zeitintervall zwischen t und $t + \Delta t$ ausfällt.

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{Pr(t < x \leq t + \Delta t | x > t)}{\Delta t} \tag{6.19}$$

Dabei ist $Pr(t < x \leq t + \Delta t | x > t)$ die bedingte Wahrscheinlichkeit dafür, dass das System im Zeitintervall ausfällt, unter der Annahme, dass es zur Zeit t noch funktioniert. Für bedingte Wahrscheinlichkeiten gilt die allgemeine Formel $Pr(A|B) = Pr(AB)/Pr(B)$, wobei $Pr(AB)$ die Wahrscheinlichkeit des Verbundereignisses AB ist. In diesem Fall ist $Pr(B)$ die Wahrscheinlichkeit, dass das System zur Zeit t noch funktioniert, also $R(t)$. Damit folgt aus Gleichung 6.19:

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{F(t + \Delta t) - F(t)}{\Delta t R(t)} \tag{6.20}$$

$$= \frac{f(t)}{R(t)} \tag{6.21}$$

Für die Exponentialverteilung ergibt sich[5]:

[5] Diese Beziehung bildet eine gewisse Motivation dafür, die Konstante der Exponentialverteilung und die Ausfallrate mit demselben Buchstaben zu bezeichnen.

$$\lambda(t) = \frac{f(t)}{R(t)} = \frac{\lambda e^{-\lambda t}}{e^{-\lambda t}} = \lambda \tag{6.22}$$

Die Fehlerraten werden häufig in der Einheit **FIT** (*failure in time*) gemessen. Ein FIT entspricht dabei einem erwarteten Fehler in 10^9 Stunden. Die Fehlerraten vieler echter Systeme sind allerdings nicht konstant. Manche folgen der sogenannten „Badewannen"-Kurve (s. Abb. 6.13). Danach sind die Fehlerraten zunächst höher („Frühausfälle"), werden dann annähernd konstant und steigen später aufgrund von Alterungserscheinungen wieder an.

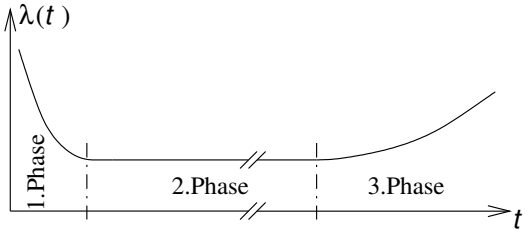


Abb. 6.13. Badewannenkurve der Fehlerraten

Definition: Unter der **mittleren Zeit bis zum Fehler** (*Mean Time To Failure* (**MTTF**)) verstehen wir die mittlere Zeit bis zu einem Fehler unter der Annahme, dass das System zunächst funktioniert. Diese Zeit ergibt sich als Erwartungswert der Zufallsvariablen x:

$$\text{MTTF} = E\{x\} = \int_0^\infty x f(x) dx \tag{6.23}$$

Für die Exponentialverteilung ergibt sich beispielsweise

$$\text{MTTF} = \int_0^\infty x \lambda e^{-\lambda x} dx \tag{6.24}$$

Das Integral können wir mit Hilfe der Produktregel ($\int uv' = uv - \int u'v$ mit $u = x$ und $v' = \lambda e^{-\lambda x}$) bestimmen. Dann ergibt sich aus der Gleichung 6.24:

$$\text{MTTF} = -[xe^{-\lambda x}]_0^\infty + \int_0^\infty e^{-\lambda x} dx \tag{6.25}$$

$$= -\frac{1}{\lambda}[e^{-\lambda x}]_0^\infty = -\frac{1}{\lambda}[0 - 1] = \frac{1}{\lambda} \tag{6.26}$$

Die mittlere Zeit bis zu einem Fehler ist damit sinnvollerweise gleich dem Kehrwert der Fehlerrate.

Definition: Unter der **mittleren Zeit zur Reparatur** (*Mean Time To Repair* (**MTTR**)) verstehen wir die mittlere Zeit bis zur Reparatur eines Fehlers unter der Annahme, dass das System zunächst defekt ist. Die MTTR ist der Erwartungswert der Verteilungsfunktion, welche die für eine Reparatur benötigten Zeiten beschreibt.

Definition: Unter der **mittleren Zeit zwischen Fehlern** (*Mean Time Between Failures*, (**MTBF**)) verstehen wir die mittlere Zeit zwischen zwei Fehlerereignissen.

Die MTBF ergibt sich als Summe der MTTF und der MTTR (s. Abb. 6.14). Die Zeichnung spiegelt dabei nicht wieder, dass es sich bei allen Größen um statistische Werte handelt und dass daher konkrete Fehlerereignisse zu anderen Zeiten auftreten könnten.

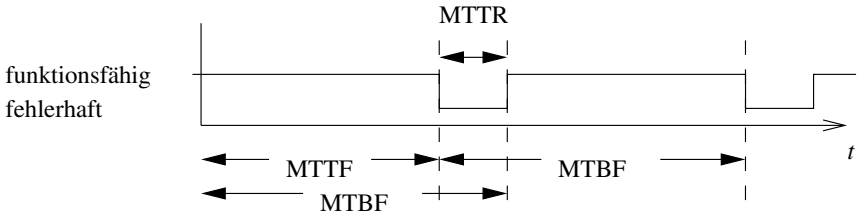


Abb. 6.14. Zur Definition von MTTF, MTTR und MTBF

Bei vielen Systemen werden Reparaturen nicht betrachtet. Außerdem sollte die MTTR sehr viel kleiner sein als die MTTF. Deswegen werden in vielen Publikationen leider die Begriffe MTBF und MTTF weitgehend synonym benutzt.

Definition: Unter der **Verfügbarkeit** (*availability*) verstehen wir den Anteil der Zeit, zu der wir über ein System verfügen können, an der Gesamtzeit.

Die Verfügbarkeit ist im Prinzip von der Zeit abhängig, über welche wir die Nutzung des Systems betrachten und wird daher als Funktion $A(t)$ modelliert. Es wird allerdings häufig nur die Verfügbarkeit $A = lim_{t\to\infty} A(t)$ für große Zeiten betrachtet. A kann einfach aus dem Quotienten von $MTTF$ und $MTBF$ berechnet werden:

$$\text{Availability } A = \lim_{t\to\infty} A(t) = \frac{\text{MTTF}}{\text{MTBF}}$$

Beispielsweise hätte ein System, dass im Mittel 999 Tage benutzt werden kann und dann jeweils einen Tag lang repariert wird, eine Verfügbarkeit von $A = 0,999$.

Erlaubte Ausfälle von Systemen können in der Größenordnung von einem Fehler in 10^9 Stunden (1 FIT) liegen. Das ist bis zu 1000 mal weniger als

die typische Fehlerrate von Chips. Offensichtlich muss man also Fehlertoleranzmaßnahmen verwenden. Wegen der niedrigen akzeptablen Fehlerrate sind die Systeme innerhalb vertretbarer Zeit nicht experimentell hinsichtlich ihrer Fehlerrate überprüfbar. Stattdessen muss die Sicherheit durch eine Kombination aus Experimenten und formalen Argumenten gezeigt werden. Abstraktion muss verwendet werden, um das System mit Hilfe einer hierarchischen Menge von Verhaltensmodellen erklärbar zu machen. Designfehler und menschliche Fehler müssen in Betracht gezogen werden.

Schäden ergeben sich aus dem Auftreten eines Fehlers. Jeder mögliche Schaden hat eine Schadenshöhe (die Kosten) und eine Wahrscheinlichkeit. Das **Risiko** kann als Produkt dieser beiden Größen definiert werden. Das Risiko kann mit verschiedenen Techniken analysiert werden [Dunn, 2002], [Press, 2003]:

- **Fehlerbaum-Analyse:** (*Fault Tree Analysis* (FTA)) Die Fehlerbaumanalyse ist eine *Top-Down*-Methode der Risikoanalyse. Die Analyse beginnt mit einem möglichen Schaden und versucht dann, Szenarien zu finden, die zu diesem Schaden führen. Bei der FTA werden meist graphische Darstellungen möglicher Schäden verwendet. Das umfasst auch Symbole für UND- und ODER-Gatter. ODER-Gatter werden verwendet, wenn ein einziges Ereignis einen Fehler erzeugen kann. UND-Gatter werden verwendet, wenn mehrere Ereignisse oder Bedingungen erfüllt sein müssen, damit der Fehler auftritt. Abbildung 6.15 zeigt ein Beispiel.

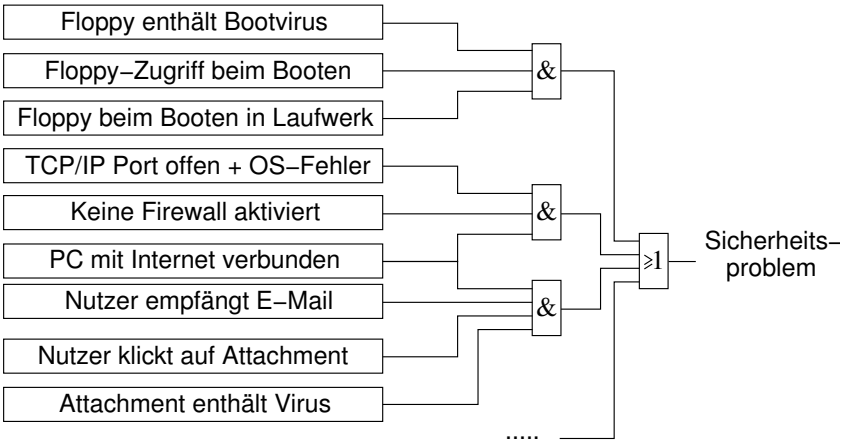


Abb. 6.15. Fehlerbaum

Die einfachen UND- und ODER-Gatter können nicht alle Situationen modellieren. Beispielsweise kann man mit ihnen keine gemeinsam genutzten Ressourcen modellieren, die nur in beschränkter Form vorliegen (beispielsweise Energie oder Speicherzellen). Markov-Modelle [Bremaud, 1999] können verwendet werden, um solche Fälle zu modellieren.

- **Fehler-Modus und Effekt-Analyse:** (*Failure Mode and Effect Analysis* (FMEA)) FMEA beginnt bei den Komponenten und versucht, deren Zuverlässigkeit abzuschätzen. Mit dieser Information wird die Zuverlässigkeit des Systems aufgrund der Zuverlässigkeit seiner Bestandteile bestimmt (entsprechend einer *Bottom-Up*-Analyse). Der erste Schritt besteht im Aufstellen einer Tabelle der Komponenten, möglicher Fehler, Fehlerwahrscheinlichkeiten und Auswirkungen auf das Systemverhalten. Die Risiken für das Gesamtsystem werden dann aus dieser Tabelle berechnet. Tabelle 6.2 zeigt ein Beispiel.

Komponente	*Fehler*	*Konsequenzen*	*Wahrhscheinlichkeit*	*Kritisch?*
Prozessor	Metallwanderung	Ausfall	10^{-7} /h	ja
...	...	...	...	...

Tabelle 6.2. FMEA-Tabelle

Weitere Informationen zu Fehlerraten können z.B. Firmenschriften entnommen werden [Triquint, 2006].

Es sind für beide Ansätze Werkzeuge verfügbar. Sowohl Fehlerbaumanalyse als auch FMEA kann in sogenannten *Safety Cases* eingesetzt werden. Dabei muss eine unabhängige Instanz davon überzeugt werden, dass ein bestimmter technischer Ausrüstungsgegenstand wirklich sicher ist. In diesem Zusammenhang wird häufig gefordert, dass der Ausfall einer einzelnen Komponente nicht zu einer Katastrophe führen darf.

Für weitergehende Informationen über Verlässlichkeit und Sicherheitsaspekte verweisen wir auf die entsprechende Literatur [Laprie, 1992], [Neumann, 1995], [Leveson, 1995], [Storey, 1996], [Geffroy und Motet, 2002].

6.8 Formale Verifikation

Die formale Verifikation beschäftigt sich mit formalen mathematischen Beweisen, welche die Korrektheit eines Systems zeigen. Zur Durchführung einer formalen Verifikation wird zuerst ein formales Modell benötigt. Wenn ein Modell erstellt wurde, können damit bestimmte Eigenschaften bewiesen werden.

Die Techniken der formalen Verifikation können anhand der verwendeten Logik klassifiziert werden:

- **Aussagen-Logik:** In diesem Fall bestehen die Modelle aus Booleschen Gleichungen, die durch Boolesche Variablen und Operatoren wie UND und ODER beschrieben werden. (Zustandslose) Logik-Netzwerke auf Gatterebene können mit dieser Logik bequem beschrieben werden. Die verfügbaren Werkzeuge zielen häufig darauf ab, die Äquivalenz zweier auf diese Art

beschriebener Modelle zu beweisen. Solche Werkzeuge heißen **Tautologie-Checker** oder **Äquivalenz-Checker**. Da die Aussagenlogik entscheidbar ist, kann entschieden werden, ob zwei Darstellungen äquivalent sind oder nicht (es gibt keine unsicheren Fälle). Beispielsweise kann eine Darstellung den Gattern einer realen Schaltung entsprechen, während die andere der Spezifikation entspricht. Der Beweis der Äquivalenz der beiden Darstellungen beweist dann die Korrektheit aller durchgeführter Transformationen (beispielsweise Energie- oder Laufzeit-Optimierungen). Tautologie-Checker können häufig mit Systemen umgehen, die für eine vollständige simulationsbasierte Validierung zu groß sind. Der Hauptgrund für die Mächtigkeit von neueren Tautologie-Checkern liegt in der Verwendung von binären Entscheidungsdiagrammen (*Binary Decision Diagrams* (BDDs)) [Wegener, 2000]. Die Komplexität eines BDD-basierten Äquivalenz-Checkers für Boolesche Funktionen wächst linear mit der Anzahl der Knoten des BDDs. Im Gegensatz dazu ist die Äquivalenzprüfung von Funktionen in einer DNF-Darstellung (also als Summe von Produkten) NP-hart. Trotz allem muss natürlich die Anzahl der benötigten BDD-Knoten berücksichtigt werden. Viele Funktionen lassen sich effizient in Form von BDDs darstellen. Im Allgemeinen wächst jedoch die Anzahl der BDD-Knoten exponentiell mit der Anzahl der Variablen. In den Fällen, in denen BDDs eine Funktion effizient darstellen können, haben BDD-basierte Äquivalenz-Checker die vorher verwendeten Simulatoren ersetzt. Sie werden verwendet, um Gatternetzwerke mit Millionen von Transistoren zu verifizieren. Die Fähigkeit zur Verifikation endlicher Automaten ist dagegen sehr eingeschränkt.

- **Prädikaten-Logik erster Stufe** (*First Order Logic* (FOL)): Die Prädikatenlogik erster Stufe beinhaltet die Quantifizierung mithilfe der Existenz- ($\exists$) und All-Quantoren ($\forall$). Ein gewisses Maß an Automatisierung für die Verifikation durch FOL ist möglich. Da diese Logik im Allgemeinen nicht entscheidbar ist, können unsichere Fälle auftreten.
- **Prädikaten-Logik höherer Stufe** (*Higher Order Logic* (HOL)): Prädikatenlogik höherer Stufe erlaubt es, Funktionen so wie andere Objekte zu manipulieren[6]. Bei Verwendung von Logik höherer Ordnung ist die Automatisierung von Beweisen fast nie möglich und muss in der Regel manuell (evtl. mit Werkzeug-Unterstützung) durchgeführt werden.

Model Checking

Die Verifikation endlicher Automaten kann mithilfe von ***Model Checking*** durchgeführt werden. *Model Checking* verifiziert Eigenschaften endlicher Automaten durch eine Analyse des Zustandsraums des Systems. Die Verifikation benötigt bei diesem Ansatz drei Stufen:

[6] http://archive.comlab.ox.ac.uk/formal-methods/hol.html

1. Erzeugung eines Modells des zu verifizierenden Systems,
2. Definition der erwarteten Eigenschaften und
3. *Model-Checking* (der eigentliche Verifikations-Schritt).

Folglich benötigt ein *Model-Checking*-System das Modell und die Eigenschaften als Eingabe (s. Abb. 6.16).

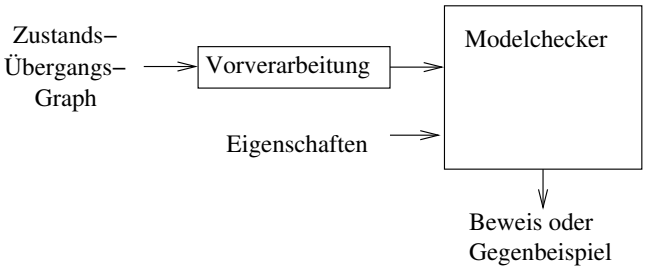


Abb. 6.16. Eingaben für *Model-Checking*

Verifikations-Werkzeuge können Eigenschaften beweisen oder widerlegen. In letzterem Fall können sie Gegenbeispiele liefern. *Model-Checking* ist einfacher zu automatisieren als FOL. Die Sprachen, die zur Definition der Eigenschaften verwendet werden, erlauben normalerweise die Verwendung von Quantoren für die Zustände.

Ein verbreitetes *Model-Checking*-System ist EMC [E. Clarke et al., 2003]. Dieses System erwartet die Eigenschaften in Form von CTL-Formeln. CTL steht für *Computational Tree Logics*. CTL-Formeln bestehen aus zwei Teilen:

- einem **Pfad-Quantifizierer** (dieser Teil spezifiziert den Pfad im Zustandsübergangsdiagramm) und
- einem **Zustands-Quantifizierer** (dieser Teil beschreibt bestimmte Zustände).

Beispiel: $M, s \models AGg$ bedeutet: Für den Zustand s des Transitionsgraphen M gilt die Eigenschaft g für alle Pfade (dargestellt als A) und alle Zustände (dargestellt als G).

Im Jahr 1987 wurde *Model-Checking* mithilfe von BDDs implementiert. Damit wurden einige Fehler in der Spezifikation des *Future Bus*-Protokolls gefunden.

Es werden Erweiterungen benötigt, um Echtzeitverhalten und Zahlen zu behandeln. Weitere Informationen über formale Verifikation findet man in Büchern zu diesem Thema. Wir verweisen hier beispielsweise auf die Bücher von Kropf [Kropf, 1999] und Clarke et al. [Clarke et al., 2000]).

6.9 Testen

6.9.1 Betrachteter Bereich

Nach der Herstellung von eingebetteten Systemen müssen diese auch getestet werden. Beim Testen legt man eine Menge von speziell ausgesuchten Eingabemustern, sogenannte **Testmuster**, an die Eingänge des zu testenden Systems an, beobachtet dessen Verhalten und vergleicht dieses Verhalten mit dem erwarteten Systemverhalten. Der Hauptzweck des Testens besteht darin, Systeme zu entdecken, die entweder nicht richtig hergestellt wurden (Herstellungstest) oder solche, die später ausfallen (Feldtest). Auch sollte die Testbarkeit eines Entwurfs nicht erst nach Abschluss des Entwurfs betrachtet werden, sondern bereits in der Evaluation von Teilentwürfen berücksichtigt werden. Weiterhin wurde vorgeschlagen, Testmethoden bereits während der Entwurfsphase einzusetzen, um die Anzahl der Möglichkeiten zur Systemvalidierung zu erhöhen. So können Testmuster etwa auf Softwaremodelle von Systemen angewendet werden, um zu überprüfen, ob zwei Modelle sich gleich verhalten. Zeitintensivere formale Methoden müssen dann nur in den Fällen verwendet werden, bei denen dieser Äquivalenztest fehlgeschlagen ist.

Das Testen umfasst eine Anzahl verschiedener Tätigkeiten:

1. **Erzeugung der Testmuster,**
2. **Anwendung der Testmuster,**
3. **Beobachtung des Systemverhaltens** und
4. **Vergleich der Ergebnisse**.

Bei der Testmustererzeugung versucht man, solche Testmuster zu finden, die korrekt funktionierende von defekten Systemen unterschieden. Die Erzeugung von Testmustern basiert auf **Fehlermodellen**, die möglicherweise auftretende Fehler modellieren. Beispielsweise ist es möglich, ein *stuck-at*-Fehlermodell zu verwenden, das annimmt, dass ein internes Signal einer Schaltung entweder immer mit `'0'` oder immer mit `'1'` verbunden ist. Man hat festgestellt, dass sich viele Fehler tatsächlich so äußern, als ob ein Signal einen festen, unveränderlichen Wert hätte. Die moderne CMOS-Technologie macht allerdings die Verwendung umfassenderer Fehlermodelle notwendig, um etwa transiente Fehler (die bei Signalübergängen auftreten) oder Verzögerungsfehler (welche die Verzögerung des Systems verändern) zu modellieren. Im Gegensatz zur Software gibt es für die Hardware gute Fehlermodelle. Die Testmustererzeugung versucht, für alle Fehler, die nach einem bestimmten Fehlermodell auftreten können, entsprechende Tests zu erzeugen. Die Qualität der Testmuster kann mit Hilfe der **Fehlerabdeckung** bestimmt werden. Die Fehlerabdeckung ist der Prozentsatz potentieller Fehler, der mit einer gegebenen Testmustermenge gefunden werden kann:

$$Abdeckung = \frac{\text{Anzahl entdeckbarer Fehler für eine Testmustermenge}}{\text{Gesamtzahl möglicher Fehler im Fehlermodell}}$$

In der Praxis müssen Fehlerabdeckungen im Bereich von 98 bis 99% liegen, um eine gute Produktqualität zu erreichen.

6.9.2 Testfreundlicher Entwurf

Wenn das Testen erst nach dem Systementwurf in Betracht gezogen wird, kann es schwierig sein, ein System zu testen. So ist es beispielsweise schwierig, festzustellen, ob ein Automat korrekt implementiert wurde. Hierzu sind in der Regel komplexe Eingabefolgen notwendig, welche den Automaten jeweils wieder in einen definierten initialen Zustand versetzen [Kohavi, 1987]. Wir betrachten dazu wieder ein Beispiel aus Kapitel 2 (siehe Abb. 6.17).

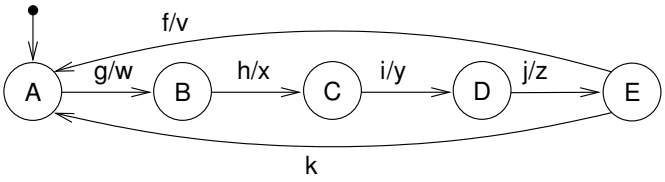


Abb. 6.17. Zu testender Automat

Wenn wir den Übergang vom Zustand C zum Zustand D testen wollen, so müssen wir zunächst den Automaten in den Ausgangszustand C bringen. Anschließend erzeugen wir die Eingabe i und überprüfen, ob die Ausgabe y erzeugt wird. Weiter müssen wir kontrollieren, ob sich der Automat im Zustand D befindet. Es ist relativ aufwendig, den Automaten zunächst in den Ausgangszustand zu bringen und den erreichten Zustand zu kontrollieren. Um das Testen zu vereinfachen, kann man das zu testende System um bestimmte Hardwarekomponenten ergänzen, welche die Durchführung von Tests vereinfachen. Der Entwurfstil, der eine bessere Testbarkeit zum Ziel hat, heißt **testfreundlicher Entwurf** (*Design for Testability* (DfT)). Ein bekanntes Beispiel ist eine Spezialhardware zum Testen endlicher Automaten. Das Erreichen bestimmter Zustände und das Beobachten des Systemverhaltens, wenn eine bestimmte Eingabe angelegt wird, wird durch das sogenannte *Scan-Design* deutlich vereinfacht. Beim *Scan-Design* werden alle Flip-Flops, die Zustände speichern, als serielle Schieberegister miteinander verbunden (s. Abb. 6.18).

Wenn die Multiplexer im *Scan*-Modus arbeiten, kann jeder beliebige Zustand seriell in die drei Flip-Flops geladen werden. In einer zweiten Phase kann man dann Eingaben an den Automaten anlegen, während sich die Multiplexer im normalen Modus befinden. Im Ergebnis befindet sich der Automat danach in einem neuen Zustand. Dieser neue Zustand kann in der dritten Phase seriell aus den Schieberegistern herausgeschoben werden. Insgesamt muss man

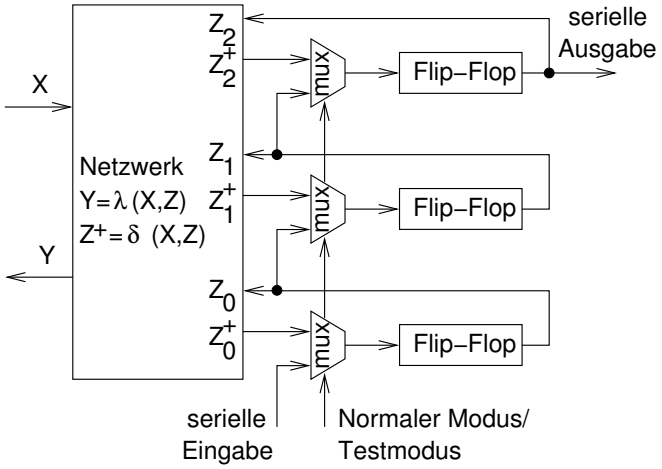


Abb. 6.18. *Scan-Path*-Entwurf

sich auf diese Weise keine Gedanken machen, wie man den Automaten in einen bestimmten Zustand bekommt und wie man beim Testen überprüfen kann, ob die Zustandsübergangsfunktion δ richtig implementiert wurde. Ein sicherer Weg, Zustandsübergänge zu testen, besteht darin, zuerst den „alten" Zustand über die Schieberegister in den Automaten zu schieben, dann die Eingabe anzulegen und den Ergebniszustand wiederum seriell aus den Schieberegistern auszulesen. Somit hat die Tatsache, dass man zustandsbehaftete Systeme testet, nur einen Einfluss auf die beiden (relativ einfachen) Schiebephasen. Ansonsten können Testmuster eingesetzt werden, wie sie auch zum Testen von zustandslosen Booleschen Netzwerken verwendet werden, um die korrekte Ausgabe des Automaten zu überprüfen. Das bedeutet, dass man sich weitestgehend auf den Einsatz von Methoden zur Testmustererzeugung für Boolesche Funktionen (bzw. zustandslose Netzwerke) beschränken kann, statt sich mit komplexen Zustandsfolgen beschäftigen zu müssen.

Scan-Design ist eine Technik, die für einzelne Chips gut funktioniert. Zum Testen mehrerer Komponenten, z.B. auf einer Platine, werden Techniken zur Verbindung der Schieberegister der einzelnen Komponenten benötigt. **JTAG** ist ein Standard, der genau dies erreicht. Der Standard definiert Register an der äußeren Begrenzung jedes Chips sowie eine Menge von Testanschlüssen und Steuerbefehlen, mit deren Hilfe alle Chips mithilfe von Schieberegistern miteinander verbunden werden können. JTAG ist auch als *Boundary Scan* bekannt [Parker, 1992].

Für Chips mit einer größeren Anzahl von Flip-Flops kann das Setzen und Auslesen der Zustände sehr lange dauern. Um den Vorgang der Testmustererzeugung zu beschleunigen, wurde auch die Integration von Hardware zur Erzeugung der Testmuster vorgeschlagen. Typischerweise werden Pseudo-Zufalls-

Muster, die von rückgekoppelten Registern erzeugt werden, als Testmuster verwendet.

Um auch das Herausschieben der Antworten des zu testenden Systems zu vermeiden, werden die Systemantworten komprimiert. Die komprimierten Antworten haben, ähnlich wie CRC-Zeichen (*Cyclic Redundancy Check*), die Eigenschaft, dass nur mit sehr geringer Wahrscheinlichkeit (ca. 2^{-n}, wobei n die Anzahl der Bits in der komprimierten Antwort darstellt) aus einer falschen Antwort eine korrekte komprimierte Antwort erzeugt wird.

Eingebaute Logikblock-Beboachter (*Built-in Logic Block Observer* (BILBO)) wurden als Schaltung vorgeschlagen [Könemann et al., 1979], um die Testmustererzeugung, die Kompaktierung der Antworten und die seriellen Ein-/Ausgabefähigkeiten zu kombinieren. Ein BILBO mit drei D-Flip-Flops ist in Abb. 6.19 gezeigt.

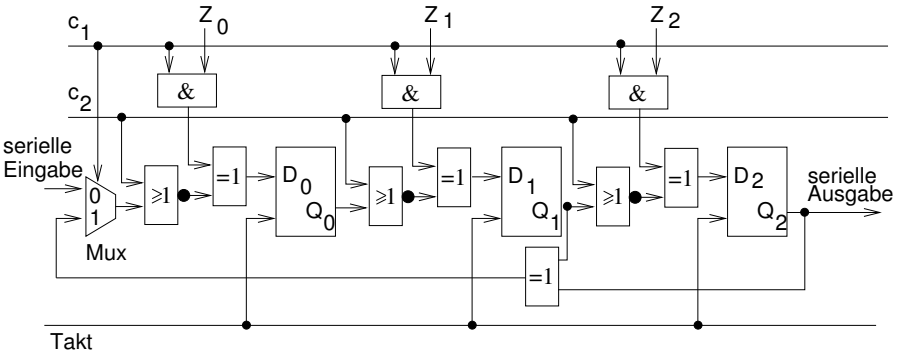


Abb. 6.19. BILBO

Die Betriebsmodi der BILBO-Register sind in Tabelle 6.3 aufgeführt. Das 3-Bit-Register aus Abb. 6.19 kann im *Scan-Path-*, *Reset-*, *Linear Feedback Shift Register* (LFSR)- oder im normalen Modus sein. Diese Modi ergeben sich aus den betreffenden Werten für die Steuereingänge c_1 und c_2. Die Wahl von $c_1 =' 0'$ und $c_2 =' 0'$ hat zur Folge, dass die Logik zwischen einem Eingang D_i und dem vorherigen Ausgang Q_{i-1} nur eine Invertierung bewirkt. Diese Wahl gestattet also die Nutzung des BILBOs als Schieberegister. Entsprechend bewirkt die Wahl $c_1 =' 0'$ und $c_2 =' 1'$, dass die Logik zwischen den Registern eine Konstante '0' erzeugt. Folglich bewirkt diese Wahl ein Rücksetzen der Register. Die Wahl $c_1 =' 1'$ und $c_2 =' 0'$ erzeugt eine XOR-Verknüpfung mit den parallelen Z_i-Eingängen. Für diesen Modus gibt es zwei Anwendungen. Einerseits können die Z_i-Eingänge mit Teilen der zu testenden Schaltung (*Circuit Under Test* (CUT)) verbunden werden. Beim Anlegen von Prüfmustern an die CUT entstehen dann im BILBO-Register Signaturen, die für die Reaktion der Schaltung auf die Prüfmuster typisch sind und zur Unterscheidung zwischen einer falschen und einer vermutlich richtigen Reaktion herangezogen werden

können. Andererseits können die Z_i-Eingänge auch mit den Ausgängen des BILBO-Registers verbunden werden. Bei geeigneten Verbindungen können so Pseudo-Zufallszahlen generiert werden, die als Prüfmuster eingesetzt werden können. Schließlich lässt sich das Register bei der Wahl $c_1 =' 1'$ und $c_2 =' 1'$ wie in normales Register benutzen.

c_1	c_2	D_i	
'0'	'0'	$'0' \oplus \overline{Q_{i-1}} = \overline{Q_{i-1}}$	Scan-Path-Modus
'0'	'1'	$'0' \oplus \overline{'1'} =' 0'$	Reset
'1'	'0'	$Z_i \oplus \overline{Q_{i-1}}$	LFSR Modus
'1'	'1'	$Z_i \oplus \overline{'1'} = Z_i$	normaler Modus

Tabelle 6.3. Modi der BILBO Register

Häufig werden BILBOs paarweise verwendet. Ein BILBO erzeugt Pseudo-Zufallsmuster und leitet diese an ein Boolesches Netzwerk weiter. Die Antwort des Netzwerks wird dann von einem zweiten BILBO komprimiert, das an den Ausgängen des Netzwerks angeschlossen ist. Am Ende der Testsequenz wird die komprimierte Antwort seriell herausgeschoben und mit der erwarteten Systemantwort verglichen.

Zusätzliche Hardware zum testfreundlichen Entwurf ist während der Erstellung von Prototypen und bei der Fehlersuche in der Hardware eine große Hilfe. Es kann auch nützlich sein, solche Schaltungen in das fertige Produkt zu integrieren, da die Herstellung von Hardware nie vollkommen fehlerfrei abläuft. Das Testen der fertigen Hardware trägt maßgeblich zu den Gesamtkosten eines Produktes bei, und Mechanismen, die diese Kosten reduzieren helfen, werden von allen Firmen sehr begrüßt.

6.9.3 Selbstestprogramme

Ein großes Problem beim Testen moderner integrierter Schaltkreise ist ihre begrenzte Anzahl von Anschlüssen. Dadurch wird es immer schwieriger, interne Komponenten der Schaltung anzusprechen und zu erreichen. Es wird auch zunehmend schwieriger, solche Systeme bei voller Betriebsgeschwindigkeit zu testen, da die Tester mindestens so schnell sein müssen wie die Schaltungen selbst. Ein möglicher Ausweg aus diesem Dilemma ergibt sich aus der Tatsache, dass viele eingebettete Systeme Prozessoren verwenden: Ein Prozessor ist in der Lage, ein Test- oder Diagnoseprogramm auszuführen. Solche Diagnoseprogramme werden schon seit Jahrzehnten zum Testen von Großrechnern eingesetzt. Abbildung 6.20 zeigt einige Komponenten eines Prozessors.

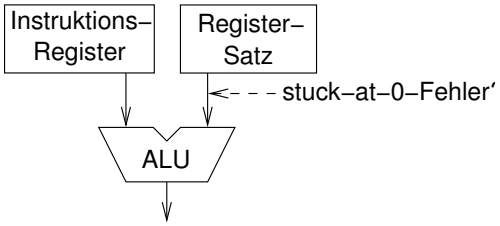


Abb. 6.20. Ausschnitt aus einer Prozessor-Hardware

Die betrachteten Fehlertypen sind Teil des **Fehlermodells**. Um beispielsweise *Stuck-at*-Fehler an den Eingängen der ALU zu testen, kann man ein kleines Testprogramm ausführen:

> Speichere ein Muster aus lauter Einsen in einem Register;
>
> Führe ein XOR zwischen der Konstanten "0000...00" und
>
> dem Register durch,
>
> überprüfe, ob das Ergebnis ein '0'-Bit enthält,
>
> wenn ja, melde einen Fehler;
>
> sonst starte den Test für den nächsten Stuck-at-Fehler

Ähnliche kleine Programme können für andere *Stuck-at*-Fehler erzeugt werden. Leider ist die Erstellung von Diagnoseprogrammen für Großrechner meist eine manuelle Tätigkeit. In der Literatur wurden verschiedentlich Methoden zur automatischen Erzeugung von Diagnoseprogrammen vorgeschlagen [Brahme und Abraham, 1984], [Krüger, 1986], [Bieker und Marwedel, 1995], [Krstic und Dey, 2002].

6.10 Fehlersimulation

Es ist momentan nicht möglich (und es wird wohl auch nicht möglich werden), das Verhalten eines fehlerbehafteten Systems vollständig vorherzusagen. Daher wird das Verhalten von fehlerbehafteten Systemen häufig simuliert. Diese Art der Simulation nennt man **Fehlersimulation**. Bei der Fehlersimulation werden Systemmodelle so verändert, dass sie das Verhalten des Systems beim Auftreten eines bestimmten Fehlers widerspiegeln.

Das Ziel der Fehlersimulation bestehet u.a. darin,

- den Effekt eines Fehlers auf die Komponenten auf der Systemebene festzustellen. Fehler heißen **redundant**, wenn sie das beobachtbare Verhalten des Systems nicht beeinflussen,
- festzustellen, ob Mechanismen zur Erhöhung der Fehlertoleranz wirklich einen positiven Effekt haben.

Bei der Fehlersimulation muss das System mit allen Fehlern, die im Fehlermodell vorkommen können, sowie mit einer großen Anzahl unterschiedlicher Eingabedaten simuliert werden. Folglich ist die Fehlersimulation ein sehr zeitaufwendiger Vorgang. Es wurden verschiedene Techniken vorgeschlagen, um die Fehlersimulation zu beschleunigen, so etwa die **parallele Fehlersimulation**. Diese Technik ist besonders effektiv, wenn das System als Modell auf der Gatterebene vorliegt. In diesem Fall sind interne Signale einfache Bitwerte. Somit kann ein Signal auf ein einzelnes Bit eines Maschinenwortes des Simulationsrechners abgebildet werden. AND- und OR-Maschineninstruktionen können dann verwendet werden, um Boolesche Netzwerke zu simulieren. Allerdings würde so nur ein einziges Bit des Maschinenwortes verwendet werden können. Die Effizienz kann durch parallele Fehlersimulation erhöht werden. Dabei werden n verschiedene Testmuster gleichzeitig simuliert, wobei n die Maschinenwortlänge ist. Der Wert jedes der n Testmuster wird auf eine bestimmte Bitposition im Maschinenwort abgebildet. Die Ausführung der oben erwähnten AND- und OR-Befehle simuliert dann das Verhalten des Netzwerks für n Testmuster gleichzeitig.

6.11 Fehlerinjektion

Die Fehlersimulation kann für reale Systeme zu zeitaufwendig sein. Wenn ein reales System verfügbar ist, kann man stattdessen die Fehlerinjektion verwenden. Dabei wird ein reales System verändert und der Effekt auf das Gesamtverhalten des Systems wird überprüft. Fehlerinjektion verlässt sich nicht auf Fehlermodelle (obwohl sie auch hier verwendet werden können). Daher kann Fehlerinjektion potentiell auch Fehler finden, die von einem Fehlermodell nicht vorhergesagt wurden.

Man kann zwischen zwei Arten von Fehlerinjektion unterscheiden:

- Lokale Fehler im System, und
- Fehler in der Umgebung (Verhalten, das nicht der Spezifikation entspricht). Beispielsweise kann man überprüfen, wie sich das System verhält, wenn es außerhalb der spezifizierten Temperatur oder Strahlung betrieben wird.

Zur Fehlerinjektion können verschiedene Methoden verwendet werden:

- Fehlerinjektion auf Hardware-Ebene: Beispiele sind z.B. die Manipulation von Anschlüssen, elektromagnetische oder Teilchen-Strahlung.
- Fehlerinjektion auf Software-Ebene, beispielsweise das Verändern von Bits im Speicher.

Nach den Experimenten von Kopetz [Kopetz, 1997] ist die Software-basierte Fehlerinjektion prinzipiell genauso effektiv wie die Hardware-basierte. Der

Einsatz von Teilchenstrahlung war hier eine nennenswerte Ausnahme, da diese Strahlung Fehler erzeugt, die mit anderen Methoden nicht hervorgerufen wurden.

Literaturverzeichnis

[Aamodt und Chow, 2000] Aamodt, T. und Chow, P. (2000). Embedded ISA support for enhanced floating-point to fixed-point ANSI C compilation. *3rd ACM Intern. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 128–137.

[Absint, 2002] Absint (2002). Absint: WCET analyses. *http://www.absint.de/wcet.htm*.

[Accellera, 2002] Accellera (2002). EDA industry working groups (for Rosetta). *http://www.eda.org*.

[Accellera, 2005] Accellera (2005). SystemVerilog. *http://www.systemverilog.org*.

[Ambler, 2005] Ambler, S. (2005). The diagrams of UML 2.0. *http://www.agilemodeling.com/essays/umlDiagrams.htm*.

[Appel, 1998] Appel, A. W. (1998). *Modern Compiler Implementation in C*. Cambridge University Press, Cambridge, New York.

[Araujo und Malik, 1995] Araujo, G. und Malik, S. (1995). Optimal code generation for embedded memory non-homogenous register architectures. *8th Int. Symp. on System Synthesis (ISSS)*, pages 36–41.

[August et al., 1997] August, D. I., Hwu, W. W., und Mahlke, S. (1997). A framework for balancing control flow and predication. *Ann. Workshop on Microprogramming and Microarchitecture (MICRO)*, pages 92–103.

[Azevedo et al., 2002] Azevedo, A., Issenin, I., Cornea, R., Gupta, R., Dutt, N., Veidenbaum, A., und Nicolau, A. (2002). Profile-based dynamic voltage scheduling using program checkpoints. *Design, Automation, and Test in Europe (DATE)*, pages 168–175.

[Balarin et al., 1998] Balarin, F., Lavagno, L., Murthy, P., und Sangiovanni-Vincentelli, A. (1998). Scheduling for embedded real-time systems. *IEEE Design & Test of Computers*, pages 71–82.

[Ball, 1996] Ball, S. R. (1996). *Embedded Microprocessor Systems - Real world designs*. Newnes.

[Ball, 1998] Ball, S. R. (1998). *Debugging Embedded Microprocessor Systems*. Newnes.

[Barabanov, 1997] Barabanov, M. (1997). A Linux-based Real-Time Operating System. *Master thesis, New Mexico Institute of Mining and Technology*.

[Barr, 1999] Barr, M. (1999). *Programming Embedded Systems*. O'Reilly.

[Basu et al., 1999] Basu, A., Leupers, R., und Marwedel, P. (1999). Array index allocation under register constraints. *Int. Conf. on VLSI Design, Goa/India*.

[Benini und Micheli, 1998] Benini, L. und Micheli, G. D. (1998). *Dynamic Power Management – Design Techniques and CAD Tools*. Kluwer Academic Publishers.

[Bergé et al., 1995] Bergé, J.-M., Levia, O., und Rouillard, J. (1995). *High-Level System Modeling*. Kluwer Academic Publishers.

[Berger, 2001] Berger, H. (2001). *Automating with STEP 7 in LAD and FBD: SIMATIC S7-300/400 Programmable Controllers*. Wiley.

[Berkelaar und et al., 2005] Berkelaar, M. und et al. (2005). Unix^{tm} manual page of lp_solve. *Eindhoven University of Technology, Design Automation Section, current version of source code available at http:// groups.yahoo.com/ group/ lp_solve/ files*.

[Bieker und Marwedel, 1995] Bieker, U. und Marwedel, P. (1995). Retargetable self-test program generation using constraint logic programming. *32nd Design Automation Conference (DAC)*, pages 605–611.

[Boussinot und de Simone, 1991] Boussinot, F. und de Simone, R. (1991). The Esterel language. *Proc. of the IEEE, Vol. 79, No. 9*, pages 1293–1304.

[Bouyssounouse und Sifakis, 2005] Bouyssounouse, B. und Sifakis, J., editors (2005). *Embedded Systems Design, The ARTIST Roadmap for Research and Development*. Lecture Notes in Computer Science, Vol. 3436, Springer.

[Brahme und Abraham, 1984] Brahme, D. und Abraham, J. A. (1984). Functional testing of microprocessors. *IEEE Trans. on Computers*, pages 475–485.

[Bremaud, 1999] Bremaud, P. (1999). *Markov Chains*. Springer Verlag.

[Brockmeyer et al., 2003] Brockmeyer, E., Miranda, M., Corporaal, H., und Catthoor, F. (2003). Layer assignment techniques for low energy in multi-layered memory organisations. *Design, Automation and Test in Europe (DATE)*, pages 11070–11075.

[Burd und Brodersen, 2000] Burd, T. und Brodersen, R. (2000). Design issues for dynamic voltage scaling. *Intern. Symp. on Low Power Electronics and Design (ISLPED)*, pages 9–14.

[Burd und Brodersen, 2003] Burd, T. und Brodersen, R. W. (2003). *Energy efficient microprocessor design*. Kluwer Academic Publishers.

[Burkhardt, 2001] Burkhardt, J. (2001). *Pervasive Computing*. Addison-Wesley.

[Burns und Wellings, 1990] Burns, A. und Wellings, A. (1990). *Real-Time Systems and Their Programming Languages*. Addison-Wesley.

[Burns und Wellings, 2001] Burns, A. und Wellings, A. (2001). *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley.

[Buttazzo, 2002] Buttazzo, G. (2002). *Hard Real-time computing systems*. Kluwer Academic Publishers, 4th printing.

[Byteflight Consortium, 2003] Byteflight Consortium (2003). Home page. *http:// www.byteflight.com*.

[Camposano und Wolf, 1996] Camposano, R. und Wolf, W. (1996). Message from the editors-in-chief. *Design Automation for Embedded Systems*.

[Center for Embedded Computer Systems, 2003] Center for Embedded Computer Systems (2003). SoC Environment. *http:// www.cecs.uci.edu/ ~cad/ sce.html*.

[Chandrakasan et al., 1992] Chandrakasan, A., Sheng, S., und Brodersen, R. W. (1992). Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):119–123.

[Chandrakasan et al., 1995] Chandrakasan, A. P., Sheng, S., und Brodersen, R. W. (1995). Low power CMOS digital design. *Kluwer Academic Publishers*.

[Chetto et al., 1990] Chetto, H., Silly, M., und Bouchentouf, T. (1990). Dynamic scheduling of real-time tasks under precedence constraints. *Journal of Real-Time Systems, 2.*

[Choi und Kim, 2002] Choi, Y. und Kim, T. (2002). Address assignment combined with scheduling in DSP code generation. *Design Automation Conference.*

[Chung et al., 2001] Chung, E.-Y., Benini, L., und Micheli, G. D. (2001). Source code transformation based on software cost analysis. In *Int. Symp. on System Synthesis (ISSS)*, pages 153–158.

[Cinderella ApS, 2003] Cinderella ApS (2003). home page. *http://www.cinderella.dk*.

[Clarke et al., 2000] Clarke, E. M., Grumberg, O., und Peled, D. A. (2000). *Model Checking.* The MIT Press; Second printing.

[Clouard et al., 2003] Clouard, A., Jain, K., Ghenassia, F., Maillet-Contoz, L., und Strassen, J. (2003). Using transactional models in SoC design flow. *in [Müller et al., 2003]*, pages 29–64.

[Coelho, 1989] Coelho, D. R. (1989). The VHDL handbook. *Kluwer Academic Publishers.*

[Cortadella et al., 2000] Cortadella, J., Kondratyev, A., Lavagno, L., Massot, M., Moral, S., Passerone, C., Watanabe, Y., und Sangiovanni-Vincentelli, A. (2000). Task generation and compile-time scheduling for mixed data-control embedded software. *Design automation conference (DAC)*, pages 489–494.

[Damm und Harel, 2001] Damm, W. und Harel, D. (2001). LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80.

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

[Davis et al., 2001] Davis, J., Hylands, C., Janneck, J., Lee, E. A., Liu, J., Liu, X., Neuendorffer, S., Sachs, S., Stewart, M., Vissers, K., Whitaker, P., und Xiong, Y. (2001). Overview of the Ptolemy project. *Technical Memorandum UCB/ERL M01/11; http://ptolemy.eecs.berkeley.edu.*

[De Greef et al., 1997a] De Greef, E., Catthoor, F., und Man, H. D. (1997a). Array placement for storage size reduction in embedded multimedia systems. *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 66–75.

[De Greef et al., 1997b] De Greef, E., F.Catthoor, und Man, H. (1997b). Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Intern. Parallel Proc. Symp.(IPPS) in Proc. Workshop on "Parallel Processing and Multimedia"*, pages 84–98.

[De Man, 2002] De Man, H. (2002). Keynote session at DATE'02. *http://www.date-conference.com/conference/2002/keynotes/index.htm.*

[Deutsches Institut für Normung, 1997] Deutsches Institut für Normung (1997). *DIN 66253, Programmiersprache PEARL, Teil 2 PEARL 90.* Beuth-Verlag; English version available through http://www.din.de.

[Dierickx, 2000] Dierickx, B. (2000). CMOS image sensors - concepts, Photonics West 2000 short course. *http://epp.fnal.gov/DocDB/0000/000048/001/pw00concepts.pdf.*

[Dobelle, 2003] Dobelle (2003). Wikipedia article. *http://en.wikipedia.org/wiki/William_H._Dobelle.*

[Douglass, 2000] Douglass, B. P. (2000). *Real-Time UML, 2nd edition.* Addison Wesley.

[Drusinsky und Harel, 1989] Drusinsky, D. und Harel, D. (1989). Using statecharts for hardware description and synthesis. *IEEE Trans. on Computer Design*, pages 798–807.

[Dunn, 2002] Dunn, W. (2002). *Practical Design of Safety-Critical Computer Systems*. Reliability Press.

[E. Clarke et al., 2003] E. Clarke et al. (2003). Model checking@CMU. *http://www-2.cs.cmu.edu/~modelcheck/index.html*.

[Egger et al., 2006] Egger, B., Lee, J., und Shin, H. (2006). Scratchpad memory management for portable systems with a memory management unit. *CASES*.

[Elsevier B.V., 2003a] Elsevier B.V. (2003a). Sensors and actuators A: Physical. *An International Journal*.

[Elsevier B.V., 2003b] Elsevier B.V. (2003b). Sensors and actuators B: Chemical. *An International Journal*.

[Esterel, 2006] Esterel, T. I. (2006). Homepage. *http://www.esterel-technologies.com*.

[Faßbach, 2006] Faßbach, C. (2006). Energieoptimierende, betriebssystemunterstützte, online Scratchpad-Allokation für Multiprozess-Applikationen (in German). *Master thesis, CS Dept., TU Dortmund, http://ls12-www.cs.tu-dortmund.de/publications/theses*.

[Falk und Marwedel, 2003] Falk, H. und Marwedel, P. (2003). Control flow driven splitting of loop nests at the source code level. *Design, Automation and Test in Europe (DATE)*, pages 410–415.

[Fettweis et al., 1998] Fettweis, G., Weiss, M., Drescher, W., Walther, U., Engel, F., Kobayashi, S., und Richter, T. (1998). Breaking new grounds over 3000 MMAC/s: a broadband mobile multimedia modem DSP. *Intern. Conf. on Signal Processing Application & Technology (ICSPA), available at http://citeseer.ist.psu.edu/111037.html*.

[Fisher und Dietz, 1998] Fisher, R. und Dietz, H. G. (1998). Compiling for SIMD within a single register. *Annual Workshop on Lang. & Compilers for Parallel Computing (LCPC)*, pages 290–304.

[Fisher und Dietz, 1999] Fisher, R. J. und Dietz, H. G. (1999). The Scc compiler: SWARing at MMX and 3DNow! *Annual Workshop on Lang. & Compilers for Parallel Computing (LCPC)*, pages 399–414.

[FlexRay Consortium, 2002] FlexRay Consortium (2002). Flexray® requirement specification. version 2.01. *http://www.flexray.de*.

[Fowler und Scott, 1998] Fowler, M. und Scott, K. (1998). *UML Distilled - Applying the Standard Object Modeling Language*. Addison-Wesley.

[Fu et al., 1987] Fu, K., Gonzalez, R., und Lee, C. (1987). *Robotics*. McGraw-Hill.

[Gajski et al., 1994] Gajski, D., Vahid, F., Narayan, S., und Gong, J. (1994). *Specification and Design of Embedded Systems*. Prentice Hall.

[Gajski et al., 2000] Gajski, D., Zhu, J., Dömer, R., Gerstlauer, A., und Zhao, S. (2000). *SpecC: Specification Language Methodology*. Kluwer Academic Publishers.

[Ganssle, 1992] Ganssle, J. G. (1992). *Programming Embedded Systems*. Academic Press.

[Ganssle, 2000] Ganssle, J. G. (2000). *The Art of Designing Embedded Systems*. Newnes.

[Garey und Johnson, 1979] Garey, M. R. und Johnson, D. S. (1979). *Computers and Intractability*. Bell Labaratories, Murray Hill, New Jersey.

[Gebotys, 1997] Gebotys, C. (1997). DSP address optimization using minimum cost circulation technique. *ICCAD*.

[Geffroy und Motet, 2002] Geffroy, J.-C. und Motet, G. (2002). *Design of Dependable computing Systems*. Kluwer Academic Publishers.

[Gelsen, 2003] Gelsen, O. (2003). Organic displays enter consumer electronics. *Opto & Laser Europe, June; availabe at http:// optics.org/ articles/ ole/ 8/ 6/ 6/ 1*.

[Genin et al., 1990] Genin, D., Hilfinger, P., Rabaey, J., Scheers, C., und Man, H. D. (1990). DSP specification using the Silage language. *1990 International Conference on Acoustics, Speech, and Signal Processing (ICASSP-90)*, pages 1056–1060.

[Gupta, 2002] Gupta, R. (2002). Introduction to embedded systems. *http:// www. ics.uci.edu/ ~rgupta/ ics212.html, Query: 2003*.

[Halbwachs, 1998] Halbwachs, N. (1998). Synchronous programming of reactive systems, a tutorial and commented bibliography. *Tenth International Conference on Computer-Aided Verification, CAV'98, LNCS 1427, Springer Verlag*.

[Halbwachs et al., 1991] Halbwachs, N., Caspi, P., Raymond, P., und Pilaud, D. (1991). The synchronous dataflow language LUSTRE. *Proc. of the IEEE*, 79:1305–1320.

[Hansmann, 2001] Hansmann, U. (2001). *Pervasive Computing*. Springer Verlag.

[Harbour, 1993] Harbour, M. G. (1993). RT-POSIX: An overview. *http:// www.ctr. unican.es/ publications/ mgh-1993a.pdf*.

[Harel, 1987] Harel, D. (1987). StateCharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274.

[Hayes, 1982] Hayes, J. (1982). A unified switching theory with applications to VLSI design. *Proceedings of the IEEE, Vol.70*, pages 1140–1151.

[Healy et al., 1999] Healy, C., Arnold, R., Mueller, F., Whalley, D., und Harmon, M. (1999). Bounding pipeline and instructions cache performance. *IEEE Transactions on Computers*, pages 53–70.

[Henia et al., 2005] Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., und Ernst, R. (2005). System level performance analysis - the SymTA/S approach. *IEE Proceedings Computers and Digital Techniques*.

[Hennessy und Patterson, 1995] Hennessy, J. L. und Patterson, D. A. (1995). *Computer Organization – The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc.

[Hennessy und Patterson, 1996] Hennessy, J. L. und Patterson, D. A. (1996). *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers Inc.

[Herrera et al., 2003a] Herrera, F., Fernández, V., Sánchez, P., und Villar, E. (2003a). Embedded software generation from SystemC for platform based design. *in [Müller et al., 2003]*, pages 247–272.

[Herrera et al., 2003b] Herrera, F., Posadas, H., Sánchez, P., und Villar, E. (2003b). Systemic embedded software generation from SystemC. *Design, Automation and Test in Europe (DATE)*, pages 10142–10149.

[Hoare, 1985] Hoare, C. (1985). *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science.

[Hogrefe, 1989] Hogrefe, D. (1989). *Estelle, LOTOS und SDL*. Springer-Verlag.

[Horn, 1974] Horn, W. (1974). Some simple scheduling algorithms. *Naval Research Logistics Quarterly, Vol. 21*, pages 177–185.

[Huerlimann, 2003] Huerlimann, D. (2003). Opentrack home page. *http:// www. opentrack.ch*.

[Hüls, 2002] Hüls, T. (2002). Optimizing the energy consumption of an MPEG application (in German). *Master thesis, CS Dept., TU Dortmund, http:// ls12-www. cs.tu-dortmund.de/ publications/ theses*.

[Huynh et al., 2006] Huynh, J., Amaral, J., Berube, P., und Touati, S. (2006). Evaluation of offset assignment heuristics. *http://www.cs.ualberta.ca/TechReports/2006/TR06-04/TR06-04.ps*.

[IEC, 2002] IEC (2002). GRAFCET specification language for sequential function charts. *http://www.iec-normen.de/shop/product.php?artikelnr=200768*.

[IEEE, 1997] IEEE (1997). *IEEE Standard VHDL Language Reference Manual (1076-1997)*. IEEE.

[IEEE, 1992] IEEE, D. (1992). Draft standard VHDL language reference manual. *IEEE Standards Department, 1992*.

[IMEC, 1997] IMEC (1997). LIC-SMARTpen identifies signer. *IMEC Newsletter, http://www.imec.be/wwwinter/mediacenter/en/newsletter_18.pdf, Query: 2003*.

[IMEC Desics group, 2003] IMEC Desics group (2003). Task concurrency management (overview of IMEC activities). *http://www.imec.be/design/tcm/*.

[Intel, 2008] Intel (2008). Intel® itanium® processor 9000 sequence. *http://www.intel.com/products/processor/itanium/index.htm, Query: 2008*.

[Ishihara und Yasuura, 1998] Ishihara, T. und Yasuura, H. (1998). Voltage scheduling problem for dynamically variable voltage processors. *Intern. Symp. on Low Power Electronics and Design (ISLPED)*, pages 197–2002.

[Iyer und Marculescu, 2002] Iyer, A. und Marculescu, D. (2002). Power and performance evaluation of globally asynchronous locally synchronous processors. *Intern. Symp. on Computer Architecture (ISCA)*, pages 158–168.

[Jackson, 1955] Jackson, J. (1955). Scheduling a production line to minimize maximum tardiness. *Management Science Research Project 43, University of California, Los Angeles*.

[Jacome et al., 2000] Jacome, M., de Veciana, G., und Lapinskii, V. (2000). Exploring performance tradeoffs for clustered VLIW ASIPs. *IEEE Int. Conf. on Computer-Aided Design (ICCAD)*, pages 504–510.

[Jacome und de Veciana, 1999] Jacome, M. F. und de Veciana, G. (1999). Lower bound on latency for VLIW ASIP datapaths. *IEEE Int. Conf. on Computer-Aided Design (ICCAD)*, pages 261–269.

[Jain et al., 2001] Jain, M., Balakrishnan, M., und Kumar, A. (2001). ASIP design methodologies : Survey and issues. *Fourteenth International Conference on VLSI Design*, pages 76–81.

[Janka, 2002] Janka, R. (2002). *Specification and Design Methodology for Real-Time Embedded Systems*. Kluwer Academic Publishers.

[Jantsch, 2003] Jantsch, A. (2003). *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann.

[Java Community Process, 2002] Java Community Process (2002). JSR-1 – real-time specification for Java. *http://www.jcp.org/en/jsr/detail?id=1*.

[Jeckle, 2004] Jeckle, M. (2004). UML auf gut deutsch, http://www.jeckle.de/uml.de.

[Jeffrey und Leduc, 1996] Jeffrey, A. und Leduc, G. (1996). E-LOTOS core language. *http://mack.ittc.ku.edu/jeffrey96elotos.html, Query: 2008*.

[Jha und Dutt, 1993] Jha, P. und Dutt, N. (1993). Rapid estimation for parameterized components in high-level synthesis. *IEEE Transactions on VLSI Systems*, pages 296–303.

[Jones, 1997] Jones, M. (1997). What really happened on Mars Rover Pathfinder. *in: P.G.Neumann (ed.): comp.risks, The Risks Digest, Vol. 19, Issue 49*.

[Jovanovic, 2006] Jovanovic, O. (2006). Dynamic voltage selection for energy efficient real-time multiprocessors. Technical report, Diplomarbeit, Fakultät für Informatik, Technische Universität Dortmund.

[Kahn, 1974] Kahn, G. (1974). The semantics of a simple language for parallel programming. *Proc. of the IFIP Congress 74*, pages 471–475.

[Kandemir et al., 2004] Kandemir, M., Ramanujam, J., Irwin, M. J., Vijaykrishnan, N., Kadayif, I., und Parikh, A. (2004). A compiler based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on CAD (TCAD)*, pages 243–260.

[Keding et al., 1998] Keding, H., Willems, M., Coors, M., und Meyr, H. (1998). FRIDGE: A fixed-point design and simulation environment. *Design, Automation and Test in Europe (DATE)*, pages 429–435.

[Kempe, 1995] Kempe, M. (1995). Ada 95 Reference Manual, ISO/IEC standard 8652:1995. *(HTML-version), http://www.adahome.com/rm95/*.

[Kernighan und Ritchie, 1988] Kernighan, B. W. und Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall.

[Kienhuis et al., 2000] Kienhuis, B., Rijjpkema, E., und Deprettere, E. (2000). Compaan: Deriving process networks from Matlab for embedded signal processing architectures. *Proc. 8th Intern. Workshop on Hardware/Software Codesign (CODES)*.

[Könemann et al., 1979] Könemann, B., Mucha, J., und Zwiehoff, G. (1979). Built-in logic block observer. *Proc. IEEE Intern. Test Conf.*, pages 261–266.

[Kobryn, 2001] Kobryn, C. (2001). UML 2001: A standardization Odyssey. *Communication of the ACM (CACM), available at http://www.omg.org/attachments/pdf/UML_2001_CACM_Oct99_p29-Kobryn.pdf*, pages 29–36.

[Kohavi, 1987] Kohavi, Z. (1987). *Switching and Finite Automata Theory*. Tata McGraw-Hill Publishing Company, New Delhi, 9th reprint.

[Kopetz, 1997] Kopetz, H. (1997). *Real-Time Systems –Design Principles for Distributed Embedded Applications–*. Kluwer Academic Publishers.

[Kopetz, 2003] Kopetz, H. (2003). Architecture of safety-critical distributed real-time systems. *Invited Talk; Design, Automation, and Test in Europe (DATE)*.

[Kopetz und Grunsteidl, 1994] Kopetz, H. und Grunsteidl, G. (1994). TTP –a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27:14–23.

[Krall, 2000] Krall, A. (2000). Compilation techniques for multimedia extensions. *International Journal of Parallel Programming*, 28:347–361.

[Krishna und Shin, 1997] Krishna, C. und Shin, K. G. (1997). *Real-Time Systems*. McGraw-Hill, Computer Science Series.

[Kropf, 1999] Kropf, T. (1999). *Introduction to Formal Hardware Verification*. Springer-Verlag.

[Krstic und Dey, 2002] Krstic, A. und Dey, S. (2002). Embedded software-based self-test for programmable core-based designs. *IEEE Design & Test*, pages 18–27.

[Krüger, 1986] Krüger, G. (1986). Automatic generation of self-test programs: A new feature of the MIMOLA design system. *23rd Design Automation Conference (DAC)*, pages 378–384.

[Kuchcinski, 2002] Kuchcinski, K. (2002). System partitioning (course notes). *http://www.cs.lth.se/home/Krzysztof_Kuchcinski/DES/Lectures/Lecture7.pdf, Query: 2008*.

[Kwok und Ahmad, 1999] Kwok, Y.-K. und Ahmad, I. (1999). Static scheduling algorithms for allocation directed task graphs to multiprocessors. *ACM Computing Surveys*, 31:406–471.

[Lam et al., 1991] Lam, M. S., Rothberg, E. E., und Wolf, M. E. (1991). The cache performance and optimizations of blocked algorithms. *Proceedings of ASPLOS IV*, pages 63–74.

[Landwehr und Marwedel, 1997] Landwehr, B. und Marwedel, P. (1997). A new optimization technique for improving resource exploitation and critical path minimization. *10th International Symposium on System Synthesis (ISSS)*, pages 65–72.

[Lapinskii et al., 2001] Lapinskii, V., Jacome, M. F., und de Veciana, G. (2001). Application-specific clustered VLIW datapaths: Early exploration on a parameterized design space. Technical Report UT-CERC-TR-MFJ/GDV-01-1, Computer Engineering Research Center, University of Texas at Austin.

[Laprie, 1992] Laprie, J. C., editor (1992). *Dependability: basic concepts and terminology in English, French, German, Italian and Japanese*. IFIP WG 10.4, Dependable Computing and Fault Tolerance, in: volume 5 of Dependable computing and fault tolerant systems, Springer Verlag.

[Larsen und Amarasinghe, 2000] Larsen, S. und Amarasinghe, S. (2000). Exploiting superword parallelism with multimedia instructions sets. *Programming Language Design and Implementation (PLDI)*, pages 145–156.

[Latendresse, 2004] Latendresse, M. (2004). The code compression bibliography. *http://www.iro.umontreal.ca/~latendre/compactBib/index.html, Query: 2007.*

[Lawler, 1973] Lawler, E. L. (1973). Optimal sequencing of a single machine subject to precedence constraints. *Managements Science, Vol. 19*, pages 544–546.

[Lee, 1999] Lee, E. (1999). Embedded software – an agenda for research. Technical report, UCB ERL Memorandum M99/63.

[Lee und Messerschmitt, 1987] Lee, E. und Messerschmitt, D. (1987). Synchronous data flow. *Proc. of the IEEE, vol. 75*, pages 1235–1245.

[Lee, 2006] Lee, E. A. (2006). The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley.

[Lee et al., 2001] Lee, S., Ermedahl, A., und Min, S. (2001). An accurate instruction-level energy consumption model for embedded risc processors. *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES).*

[Leupers, 1997] Leupers, R. (1997). *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers.

[Leupers, 1999] Leupers, R. (1999). Exploiting conditional instructions in code generation for embedded VLIW processors. *Design, Automation and Test in Europe (DATE).*

[Leupers, 2000a] Leupers, R. (2000a). *Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools*. Kluwer Academic Publishers.

[Leupers, 2000b] Leupers, R. (2000b). Code selection for media processors with SIMD instructions. *Design, Automation and Test in Europe (DATE)*, pages 4–8.

[Leupers, 2000c] Leupers, R. (2000c). Instruction scheduling for clustered VLIW DSPs. *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 291–300.

[Leupers und David, 1998] Leupers, R. und David, F. (1998). A uniform optimization technique for offset assignment problems. *Int. Symp. on System Synthesis (ISSS)*, pages 3–8.

[Leupers und Marwedel, 1995] Leupers, R. und Marwedel, P. (1995). Time-constrained code compaction for DSPs. *Int. Symp. on System Synthesis (ISSS)*, pages 54–59.

[Leupers und Marwedel, 1996] Leupers, R. und Marwedel, P. (1996). Algorithms for address assignment in DSP code generation. *IEEE Int. Conf. on Computer-Aided Design (ICCAD*, pages 109–112.

[Leupers und Marwedel, 1999] Leupers, R. und Marwedel, P. (1999). Function inlining under code size constraints for embedded processors. *IEEE Int. Conf. on Computer-Aided Design (ICCAD)*, pages 253–256.

[Leupers und Marwedel, 2001] Leupers, R. und Marwedel, P. (2001). *Retargetable Compiler Technology for Embedded Systems – Tools and Applications*. Kluwer Academic Publishers.

[Leveson, 1995] Leveson, N. (1995). *Safeware, System safety and Computers*. Addison Wesley.

[Liao et al., 1995a] Liao, S., Devadas, S., Keutzer, K., und Tijang, S. (1995a). Code optimization techniques for embedded DSP microprocessors. *32nd Design Automation Conference (DAC)*, pages 599–604.

[Liao et al., 1995b] Liao, S., Devadas, S., Keutzer, K., Tijang, S., und Wang, A. (1995b). Storage assignment to decrease code size. *Programming Language Design and Implementation (PLDI)*, pages 186–195.

[Liu und Layland, 1973] Liu, C. L. und Layland, J. W. (1973). Scheduling algorithms for multi-programming in a hard real-time environment. *Journal of the Association for Computing Machinery (JACM)*, pages 40–61.

[Liu, 2000] Liu, J. W. (2000). *Real-Time Systems*. Prentice Hall.

[Lorenz et al., 2002] Lorenz, M., Wehmeyer, L., Draeger, T., und Leupers, R. (2002). Energy aware compilation for DSPs with SIMD instructions. *LCTES/SCOPES '02*, pages 94–101.

[Machanik, 2002] Machanik, P. (2002). Approaches to addressing the memory wall. *Technical Report, November, Univ. Brisbane*.

[Mahlke et al., 1992] Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E., und Bringmann, R. A. (1992). Effective compiler support for predicated execution using the hyperblock. *MICRO-92*, pages 45–54.

[Marwedel und Goossens, 1995] Marwedel, P. und Goossens, G., editors (1995). *Code Generation for Embedded Processors*. Kluwer Academic Publishers.

[Marwedel und Schenk, 1993] Marwedel, P. und Schenk, W. (1993). Cooperation of synthesis, retargetable code generation and testgeneration in the MSS. *EDAC-EUROASIC'93*, pages 63–69.

[Marzano und Aarts, 2003] Marzano, S. und Aarts, E. (2003). *The New Everyday*. 010 Publishers.

[McLaughlin und Moore, 2001] McLaughlin, M. und Moore, A. (2001). Real-Time Extensions to UML. *Dr. Dobb's Portal, http://www.ddj.com/dept/architect/184410749*.

[Menard und Sentieys, 2002] Menard, D. und Sentieys, O. (2002). Automatic evaluation of the accuracy of fixed-point algorithms. *Design, Automation and Test in Europe (DATE)*, pages 529–535.

[Mentor Graphics, 1996] Mentor Graphics (1996). DSP architect DFL user's and reference manual.

[Mermet et al., 1998] Mermet, J., Marwedel, P., Ramming, F. J., Newton, C., Borrione, D., und Lefaou, C. (1998). Three decades of hardware description languages in Europe. *Journal of Electrical Engineering and Information Science*, 3:106pp.

[Muchnick, 1997] Muchnick, S. S. (1997). *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, Inc.

[Müller et al., 2003] Müller, W., Rosenstiel, W., und Ruf, J. (2003). *SystemC – Methodologies and Applications*. Kluwer Academic Publications.

[Neumann, 1995] Neumann, P. G. (1995). *Computer Related Risks*. Addison Wesley.

[Niemann, 1998] Niemann, R. (1998). *Hardware/Software Co-Design for Data-Flow Dominated Embedded Systems*. Kluwer Academic Publishers.

[Nilsen, 2004] Nilsen, K. (2004). Real-Time Java. *http://java.sun.com/javase/technologies/realtime.jsp*.

[Object Management Group (OMG), 2002] Object Management Group (OMG) (2002). Real-time CORBA specification, version 1.1, August 2002. *Object Management Group, http://www.omg.org/docs/formal/02-08-02.ps*.

[Object Management Group (OMG), 2003] Object Management Group (OMG) (2003). CORBA®basics. *http://www.omg.org/gettingstarted/corbafaq.htm*.

[Okuma et al., 1999] Okuma, T., Ishihara, T., und Yasuura, H. (1999). Real-time task scheduling for a variable voltage processor. In *ISSS '99: Proceedings of the 12th international symposium on System synthesis*, page 24, Washington, DC, USA. IEEE Computer Society.

[OMG, 2005] OMG (2005). UML^{TM} resource page. *http://www.uml.org*.

[Oppenheim et al., 1999] Oppenheim, A. V., Schafer, R., und Buck, J. R. (1999). *Digital Signal Processing*. Pearson Higher Education.

[Ottoni, 2003] Ottoni, D. (2003). Improving offset assignment through simultaneous variable coalescing. *SCOPES*.

[Palkovic et al., 2002] Palkovic, M., Miranda, M., und Catthoor, F. (2002). Systematic power-performance trade-off in MPEG-4 by means of selective function inlining steered by address optimisation opportunities. *Design, Automation, and Test in Europe (DATE)*, pages 1072–1079.

[Parker, 1992] Parker, K. P. (1992). *The Boundary Scan Handbook*. Kluwer Academic Press.

[Petri, 1962] Petri, C. A. (1962). Kommunikation mit Automaten. *Schriften des Institutes für Instrumentelle Mathematik, Bonn*.

[Pino und Lee, 1995] Pino, J. L. und Lee, E. (1995). Hierarchical static scheduling of dataflow graphs onto multiple processors. *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*.

[Poseidon, 2003] Poseidon (2003). Documentation for Poseidon for UML. *http://www.gentleware.com/index.php?id=documentation*.

[Potop-Butucaru et al., 2007] Potop-Butucaru, D., Edwards, S., und Berry, G. (2007). *Compiling Esterel*. Springer.

[Press, 2003] Press, D. (2003). *Guidelines for Failure Mode and Effects Analysis for Automotive, Aerospace and General Manufacturing Industries*. CRC Press.

[Ramamritham, 2002] Ramamritham, K. (2002). System support for real-time embedded systems. *in: Tutorial 1, 39th Design Automation Conference (DAC)*.

[Ramamritham et al., 1998] Ramamritham, K., Shen, C., Gonzalez, O., Sen, S., und Shirgurkar, S. B. (1998). Using Windows NT for real-time applications: Experimental observations and recommendations. *IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 102–111.

[Reisig, 1985] Reisig, W. (1985). *Petri nets*. Springer Verlag.

[Rixner et al., 2000] Rixner, S., Dally, W. J., Khailany, B., Mattson, P., Kapasi, U. J., und Owens, J. D. (2000). Register organization for media processing. *6th Intern. Symp. on High-Performance Computer Architecture*, pages 375–386.

[Russell und Jacome, 1998] Russell, T. und Jacome, M. F. (1998). Software power estimation and optimization for high performance, 32-bit embedded processors. *Proc. International Conference on Computer Design (ICCD)*, pages 328–333.

[Ryan, 1995] Ryan, M. (1995). Market focus – insight into markets that are making the news in EE Times. *EE Times, Sept. 11th 1995.*

[Sangiovanni-Vincentelli, 2002] Sangiovanni-Vincentelli, A. (2002). The context for platform-based design. *IEEE Design & Test of Computers*, page 120.

[SDL Forum Society, 2003a] SDL Forum Society (2003a). Home page. *http://www.sdl-forum.org.*

[SDL Forum Society, 2003b] SDL Forum Society (2003b). List of commercial tools. *http://www.sdl-forum.org/Tools/Commercial.htm.*

[Sedgewick, 1988] Sedgewick, R. (1988). *Algorithms.* Addison-Wesley.

[SEMATECH, 2003] SEMATECH (2003). International technology roadmap for semiconductors (ITRS). *http://public.itrs.net.*

[Sha et al., 1990] Sha, L., Rajkumar, R., und Lehoczky, J. (1990). Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Trans. on Computers*, pages 1175–1185.

[Shi und Brodersen, 2003] Shi, C. und Brodersen, R. (2003). An automated floating-point to fixed-point conversion methodology. *Int. Conf. on Audio Speed and Signal Processing (ICASSP)*, pages 529–532.

[Simunic et al., 2000] Simunic, T., Benini, L., Acquaviva, A., Glynn, P., und Micheli, G. D. (2000). Energy efficient design of portable wireless devices. *Intern. Symp. on Low Power Electronics and Design (ISLPED)*, pages 49–54.

[Simunic et al., 2001] Simunic, T., Benini, L., Acquaviva, A., Glynn, P., und Micheli, G. D. (2001). Dynamic voltage scaling and power management for portable systems. *Design Automation Conference (DAC)*, pages 524–529.

[Simunic et al., 1999] Simunic, T., Benini, L., und De Micheli, G. (1999). Cycle-accurate simulation of energy consumption in embedded systems. *Design Automation Conference (DAC)*, pages 876–872.

[Skjellum et al., 2002] Skjellum, A., Kanevsky, A., Dandass, Y. S., und Watts, J. (2002). The real-time message passing interface standard (mpi/rt-1.1), http://citeseer.ist.psu.edu/skjellum02realtime.html.

[Society for Display Technology, 2003] Society for Display Technology (2003). Home page. *http://www.sid.org.*

[Spivey, 1992] Spivey, M. (1992). *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition.

[Stankovic und Ramamritham, 1991] Stankovic, J. und Ramamritham, K. (1991). The Spring kernel: a new paradigm for real-time systems. *IEEE Software*, 8:62–72.

[Stankovic et al., 1998] Stankovic, J., Spuri, M., Ramamritham, K., und Buttazzo, G. (1998). *Deadline Scheduling for Real-Time Systems, EDF and related algorithms.* Kluwer Academic Publishers.

[Steinke et al., 2001a] Steinke, S., Knauer, M., Wehmeyer, L., und Marwedel, P. (2001a). An accurate and fine grain instruction-level energy model supporting software optimizations. *Proc. of the International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS).*

[Steinke et al., 2002] Steinke, S., L.Wehmeyer, Lee, B.-S., und Marwedel, P. (2002). Assigning program and data objects to scratchpad for energy reduction. *Design, Automation and Test in Europe (DATE)*, pages 409–417.

[Steinke et al., 2001b] Steinke, S., Schwarz, R., Wehmeyer, L., und Marwedel, P. (2001b). Low power code generation for a RISC processor by register pipelining. Technical Report 754, TU Dortmund, Dept. of CS XII, http://ls12-www.cs.tu-dortmund.de/publications/global_index.html.

[Stiller, 2000] Stiller, A. (2000). Neue prozessoren (in German). *c't*, 22:52.

[Storey, 1996] Storey, N. (1996). *Safety-critical Computer Systems*. Addison Wesley.

[Stritter und Gunter, 1979] Stritter, E. und Gunter, T. (1979). Microprocessor architecture for a changing world: The Motorola 68000. *IEEE Computer*, 12:43–52.

[Sudarsanam et al., 1997] Sudarsanam, A., Liao, S., und Devadas, S. (1997). Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. *Design Automation Conference (DAC)*, pages 287–292.

[Sudarsanam und Malik, 1995] Sudarsanam, A. und Malik, S. (1995). Memory bank and register allocation in software synthesis for ASIPs. *Intern. Conf. on Computer-Aided Design (ICCAD)*, pages 388–392.

[Synopsys, 2005] Synopsys (2005). System studio. *http://www.synopsys.com/products/cocentric_studio*.

[SystemC, 2002] SystemC (2002). Home page. *http://www.SystemC.org*.

[Takada, 2001] Takada, H. (2001). Real-time operating system for embedded systems. *in: M. Imai and N. Yoshida (eds.): Tutorial 2 – Software Development Methods for Embedded Systems, Asia South-Pacific Design Automation Conference (ASP-DAC)*.

[Tan et al., 2003] Tan, T. K., Raghunathan, A., und Jha, N. K. (2003). Software architectural transformations: A new approach to low energy embedded software. *Design, Automation and Test in Europe (DATE)*, pages 11046–11051.

[Teich et al., 1999] Teich, J., Zitzler, E., und Bhattacharyya, S. (1999). 3D exploration of software schedules for DSP algorithms. *CODES'99*, page 168pp.

[Telelogic, 1999] Telelogic (1999). Real-Time Extensions to UML. *http://www.telelogic.com/help/search/index.cfm, Query: 2007*.

[Telelogic AB, 2003] Telelogic AB (2003). Home page. *http://www.telelogic.com*.

[Tensilica Inc., 2003] Tensilica Inc. (2003). Home page. *http://www.tensilica.com*.

[Tewari, 2001] Tewari, A. (2001). *Modern Control Design with MATLAB and SIMULINK*. John Wiley and Sons Ltd.

[Thiébaut, 1995] Thiébaut, D. (1995). Parallel programming in C for the transputer. *http://maven.smith.edu/~thiebaut/transputer/descript.html*.

[Thiele, 2006] Thiele, L. (2006). Performance analysis of distributed embedded systems. *in: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press, 2006*.

[Thoen und Catthoor, 2000] Thoen, F. und Catthoor, F. (2000). *Modelling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems*. Kluwer Academic Publishers.

[Thomas und Moorby, 1991] Thomas, D. E. und Moorby, P. (1991). The Verilog hardware description language. *Kluwer Academic Publishers*.

[TimeSys Inc., 2003] TimeSys Inc. (2003). Home page. *http://www.timesys.com*.

[Tiwari et al., 1994] Tiwari, V., Malik, S., und Wolfe, A. (1994). Power analysis of embedded software: A first step towards software power minimization. *IEEE Trans. On VLSI Systems*, pages 437–445.

[Transmeta, 2005] Transmeta, I. (2005). Support: Technical documentations. *http://www.transmeta.com/developers/techdocs.html, Query: 2007*.

[Triquint, 2006] Triquint (2006). What's the MTBF for gallium arsenide devices? *http://www.triquint.com/company/quality/faqs/faq_11.cfm*.

[Vaandrager, 1998] Vaandrager, F. (1998). Lectures on embedded systems. *in Rozenberg, Vaandrager (eds), LNCS, Vol. 1494.*

[Vahid, 1995] Vahid, F. (1995). Procedure exlining. *Int. Symp. on System Synthesis (ISSS)*, pages 84–89.

[Vahid, 2002] Vahid, F. (2002). *Embedded System Design.* John Wiley& Sons.

[Verma und Marwedel, 2007] Verma, M. und Marwedel, P. (2007). *Advanced Memory Optimization Techniques for Low-Power Embedded Processors.* Springer.

[Vladimirescu, 1987] Vladimirescu, A. (1987). SPICE user's guide. *Northwest Laboratory for Integrated Systems, Seattle.*

[Vogels und Gielen, 2003] Vogels, M. und Gielen, G. (2003). Figure of merit based selection of A/D converters. *Design, Automation and Test in Europe (DATE)*, pages 1190–1191.

[von Hanxleden, 2005] von Hanxleden, R. (2005). Modellierung Reaktiver Systeme - Synchrone Sprachen und Statecharts. *in: P. Liggesmeyer, D. Rombach (Hrsg.): Software Engineering Eingebetteter Systeme, Elsevier Spektrum Akademischer Verlag.*

[Wagner und Leupers, 2002] Wagner, J. und Leupers, R. (2002). Advanced code generation for network processors with bit packet addressing. *Workshop on Network Processors (NP1).*

[Wedde und Lind, 1998] Wedde, H. und Lind, J. (1998). Integration of task scheduling and file services in the safety-critical system MELODY. *EUROMICRO '98 Workshop on Real-Time Systems, IEEE Computer Society Press*, page 18pp.

[Wegener, 2000] Wegener, I. (2000). *Branching programs and binary decision diagrams – Theory and Applications.* SIAM Monographs on Discrete Mathematics and Applications.

[Wehmeyer und Marwedel, 2006] Wehmeyer, L. und Marwedel, P. (2006). *Fast, efficient and predictable memory accesses : optimization algorithms for memory architecture aware compilation.* Springer.

[Weste et al., 2000] Weste, N. H. H., Eshraghian, K., Michael, S., Michael, J. S., und Smith, J. S. (2000). *Principles of CMOS VLSI Design: A Systems Perspective.* Addision-Wesley.

[Wilhelm, 2006] Wilhelm, R. (2006). Determining bounds on execution times. *in: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press, 2006.*

[Willems et al., 1997] Willems, M., Bürsgens, V., Keding, H., Grötker, T., und Meyr, H. (1997). System level fixed-point design based on an interpolative approach. *Design Automation Conference (DAC)*, pages 293–298.

[Wilton und Jouppi, 1996] Wilton, S. und Jouppi, N. (1996). CACTI: An enhanced access and cycle time model. *Int. Journal on Solid State Circuits*, 31(5):677–688.

[Wind River Systems, 2003] Wind River Systems (2003). Web pages. *http://www.windriver.com.*

[Winkler, 2002] Winkler, J. (2002). The CHILL homepage. *http://www1.informatik.uni-jena.de/languages/chill/chill.htm.*

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

[Wolsey, 1998] Wolsey, L. (1998). *Integer Programming.* Jon Wiley & Sons.

[Wong et al., 2001] Wong, C., Marchal, P., Yang, P., Prayati, A., Catthoor, F., Lauwereins, R., Verkest, D., und Man, H. D. (2001). Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform. *9th Intern. Symp. on Hardware/Software Codesign (CODES)*, pages 170–177.

[Xue, 2000] Xue, J. (2000). *Loop tiling for parallelism*. Kluwer Academic Publishers.

[Yodaiken, 2004] Yodaiken, V. (2004). Against priority inheritance. *FSMlabs Technical Report, http://www.fsmlabs.com/pdfs/Priority_Inheritance.pdf*.

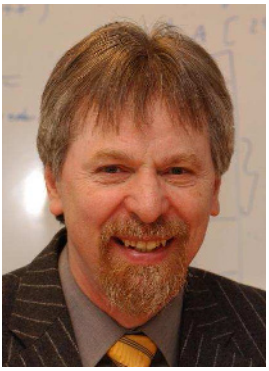
[Young, 1982] Young, S. (1982). *Real Time Languages –design and development–*. Ellis Horwood.

[Zurawski, 2006] Zurawski, R., editor (2006). *Embedded Systems Handbook*. CRC Press.

Sachverzeichnis

Über den Autor

Peter Marwedel



Peter Marwedel wurde in Hamburg geboren. Er erhielt 1974 den Doktortitel in Physik von der Universität Kiel. Von 1974 bis 1989 war er Mitarbeiter des Instituts für Informatik und Angewandte Mathematik an dieser Universität. Seit 1989 ist er Professor an der Technischen Universität Dortmund. Er ist Lehrstuhlinhaber des Lehrstuhls für „Technische Informatik und Eingebettete Systeme“ der Fakultät für Informatik und ist außerdem Vorsitzender des ICD e.V., einer Firma, die sich auf den Technologietransfer spezialisiert hat. Durch das ICD wird das Wissen über Compiler für eingebettete Systeme in kommerzielle Produkte übertragen und so externen Kunden zugänglich gemacht. Peter Marwedel war 1985/1986 als Gastprofessor an der Universität Paderborn und 1995 an der University of California at Irvine. Von 1992 bis 1995 war er Dekan des Fachbereichs Informatik. Er war aktiv am wachsenden Erfolg der DATE-Konferenz beteiligt und hat die Serie der SCOPES-Workshops ins Leben gerufen. 1975 begann er in Zusammenhang mit dem MIMOLA-Projekt mit der Arbeit an der High-Level-Synthese und hat sich auf die Synthese von *Very Long Instruction Word* (VLIW)-Maschinen spezialisiert. Später beschäftigte er sich zusätzlich mit Compilern für eingebettete Prozessoren unter besonderer Berücksichtigung der Retargierbarkeit. Seine Projekte beschäftigten sich auch mit der Synthese von Selbsttest-Programmen für Prozessoren. Seine Arbeit umfasst Codesign und Codegenerierung für energiesparende Systeme. Als Ergebnis der Arbeiten an Multimedia-basierten Lehr-Lernkomponenten entstanden die *levi* Multimedia-Lerneinheiten für Eingebettete Systeme (siehe http://ls12-www.cs.tu-dortmund.de/levi).

Peter Marwedel ist Mitglied von ACM, IEEE und der Gesellschaft für Informatik (GI).

Er ist verheiratet und hat zwei Töchter und einen Sohn. Seine Hobbys sind Skifahren, Modellbahnen und Photographie.

E-mail: Peter.Marwedel@tu-dortmund.de

Webseite: http://ls12-www.cs.tu-dortmund.de/∼marwedel