1. Grundlagen

1.1 Steuerungsaufgaben lösen

Wenn Geräte, Maschinen, Anlagen usw. selbsttätig (automatisch) arbeiten sollen, müssen Steuereinrichtungen vorgesehen werden (Abb. 1.1). Manchmal sind Schnittstellenspezifikationen einzuhalten, manchmal kann man die Schnittstellen selbst festlegen und auf die jeweiligen Anforderungen abstimmen (Entwurfsoptimierung). Das typische Beispiel ist der Verbund von Steuer- und Operationswerk in einem Universal- oder Spezialprozessor.

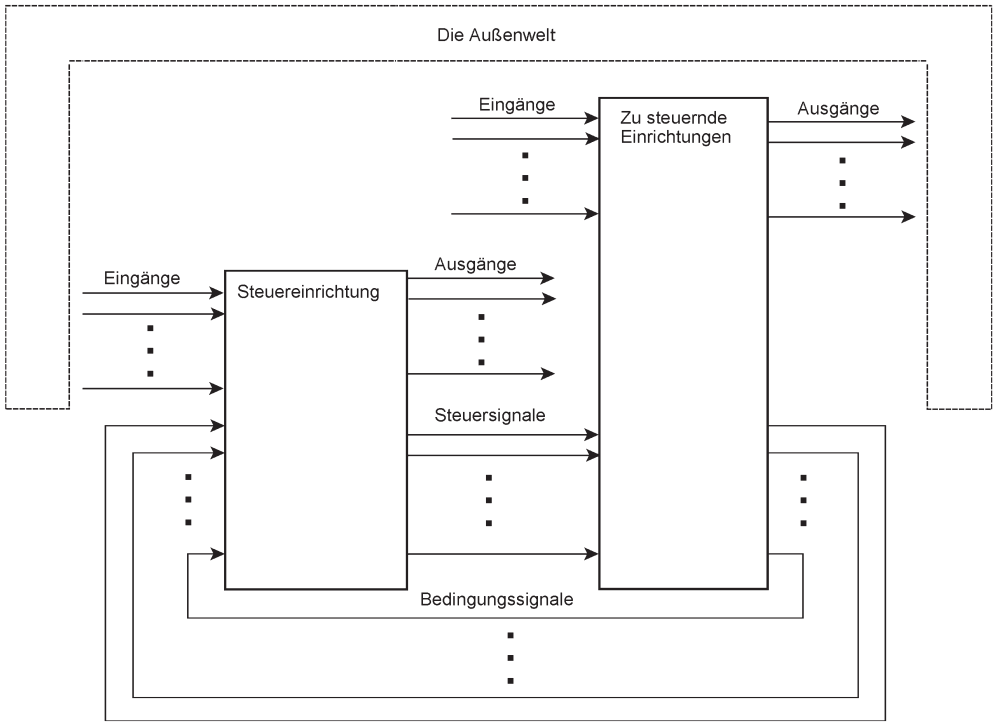

Abb. 1.1 Die Steuereinrichtung in Verbindung mit den zu steuernden Einrichtungen. Alle Einrichtungen können mit der Außenwelt verbunden sein. Die Eingänge der Steuereinrichtung sind beispielsweise Anforderungs- und Meldesignale, die Ausgänge Erlaubnis- und Kommandosignale. Die Ein- und Ausgänge der zu steuernden Einrichtungen dienen vor allem zum Einlesen und Ausgeben von Daten.

Am Anfang steht die funktionelle Absicht. Es wird nach Mitteln gesucht, sie zu verwirklichen. Das betrifft sowohl das gedankliche Vorgehen als auch die Wahl der technischen Mittel, um die gefundene Lösung zu implementieren (Tabelle 1.1).

Vorgehensweisen	Alternativen der Implementierung
• Intuitives, erfinderisches Suchen nach einer anwendungsspezifischen Lösung. • Intuitives Programmieren[1]. • Allgemein übliche, bewährte Verfahren und Entwurfsregeln[2]. • Prinzipien und Modellvorstellungen, die aus der Automatentheorie abgeleitet sind[3]. • Beschreibungsmittel, die vor allem auf der Aussagenlogik bzw. Booleschen Algebra beruhen[4]. • Schaltungsentwurf mittels Verhaltensbeschreibung. Als eine Art Programm entwerfen, aber als Maschine bauen.	• Anwendungsspezifische Hardware. • Schaltungslösung mit fertigen („harten“) Funktionsbausteinen. • Schaltungslösung mit programmierbaren Bausteinen, vor allem FPGAs. • Fertige Plattformen. Die Probleme werden durch Programmieren gelöst. • Es werden eigens programmierbare Plattformen geschaffen, um das Problem durch Programmieren zu lösen[5].

1. Das Problem wird von vornherein als Programmieraufgabe angesehen und ohne Rückgriff auf vorgefertige Lösungsmuster, graphische Entwicklungssysteme, Entwurfssprachen usw. ausprogrammiert.
2. Beispielsweise mit Kontaktplänen (Ladder Diagrams), Ablaufplänen, Funktionsplänen oder Zustandsdiagrammen (State Charts).
3. Die wichtigste Modellvorstellung ist der (endliche) Zustandsautomat (Finite State Machine, FSM).
4. Die so gefundenen Ausdrücke werden in Schaltungsstrukturen umgesetzt oder von programmierbaren Einrichtungen ausgewertet (Bitprozessor).
5. Die programmierbare Plattform kann eine Spezialmaschine sein, beispielsweise ein Boolescher Prozessor, eine Universalmaschine, die für den betreffenden Einsatzzweck optimiert ist, oder ein Mikroprogrammsteuerwerk.

Tabelle 1.1 Steuerungsaufgaben lösen. Grundsätzliche Alternativen im Überblick.

Welche Grundsatzlösung gewählt wird, ergibt sich vor allem aus den zeitlichen Anforderungen der zu steuernden Einrichtungen (Realzeitraster). Kommt es nicht auf die Millisekunde an, wird man die Steuerungsaufgabe oftmals programmseitig erledigen. Schaltungslösungen sind dann erforderlich, wenn die zeitlichen Anforderungen nicht anders erfüllt werden können oder wenn der Schaltungsaufwand auf das Nötigste beschränkt werden muß, man sich also die Infrastruktur der Programmsteuerung (Programmspeicher, Programmablaufsteuerung usw.) gar nicht leisten kann.

Das Realzeitraster

Mit diesem Begriff wollen wir pauschale Anforderungen an das Realzeitverhalten bezeichnen. In welcher Zeit muß die Steuerungsaufgabe erledigt sein? Wie lange darf es dauern, auf Signale aus der Außenwelt oder aus der Anwendungsumgebung zu reagieren, also die jeweiligen Ausgangssignale oder Ergebnisse zu liefern? Aus der Größenordnung des Realzeitrasters und der Komplexität des Anwendungsproblems ergibt sich, auf welchen Ebenen und mit welchen Mitteln die Entwicklungsaufgabe gelöst werden kann (Tabelle 1.2).

Zeitraster	Beispiele	Technische Mittel
10 ns und weniger	Befehlsausführung in Prozessoren, Speicheransteuerung, Hochleistungsschnittstellen, Videodarstellung	Hardware. Register-Transfer-Ebene (RTL), ggf. auch die Ebenen der Technologien (Halbleiter, Transistorschaltungstechnik usw.)
100 ns	Gerätesteuerung	Hardware auf der Register-Transfer-Ebene (RTL), Zustandsautomaten (State Machines), Mikroprogrammierung
1 µs	Gerätesteuerung	Hardware auf der Register-Transfer-Ebene (RTL), Zustandsautomaten (State Machines), Mikroprogrammierung, Maschinenprogrammierung üblicher Universalprozessoren (was sich mit etwa 10...50 Maschinenbefehlen erledigen läßt)
1 ms	Gerätesteuerung, Regelungsaufgaben (Closed Loop)	Leistungsoptimierte Realzeit-Software (was sich mit wenigen tausend Maschinenbefehlen erledigen läßt)
10 ms	Gerätesteuerung, Regelungsaufgaben (Closed Loop), Bedienung/Anzeige	Realzeit-Software
100 ms	Regelungsaufgaben (Closed Loop), Bedienung/Anzeige, Prozeßsteuerung, Informationsbeschaffung (Retrival)	Komplexe Realzeit-Software, Vernetzung
1 s	Prozeßsteuerung, Informationsbeschaffung (Retrival)	Realzeit- bzw. beliebige Software (je nach Hardware-Plattform), Vernetzung
Kommt nicht darauf an (hat Zeit)	Fertigungssteuerung, allgemeine Verwaltung, Informationsbeschaffung (Retrival)	Beliebige Software, Vernetzung einschließlich Internet

Tabelle 1.2 Typische Größenordnungen des Realzeitrasters.

Das allgemeine Entwicklungsziel besteht darin, die geforderten Informationswandlungen in der vorgegebenen Zeit mit kostenoptimalen[1] Mitteln zu erbringen. Beim aktuellen Stand der Technik ist man typischerweise bestrebt, Schaltungsentwicklungen zu vermeiden und möglichst viel durch Programmieren fertiger Hardwareplattformen zu erledigen. Wenn es Millisekunden oder noch länger Zeit hat, ist das zumeist auch möglich[2]. Sind es hingegen Mikrosekunden und weniger, wird man gründlicher nachdenken müssen. Nun gibt es universelle Prozessoren, die auch in

1. Es versteht sich von selbst, die Kosten über alles = über den gesamten Lebenszyklus zu betrachten (Total Cost of Ownership, TCO).
2. Dem könnte aber die Komplexität des Anwendungsproblems entgegenstehen.

solchen Zeitintervallen tausende Maschinenbefehle ausführen können. Es ist aber die Frage, wie die Programme geschrieben werden und in welcher Systemumgebung sie laufen sollen. Harte Realzeitanforderungen –Mikrosekunden und weniger – erfordern maschinennahe Programmierung[3] und eine auf diese Anforderungen abgestimmte Systemumgebung[4]. Typische Grundsatzlösungen für so harte Realzeitanforderungen sind anwendungsspezifische Schaltungen, maschinennah programmierte Universalprozessoren und Maschinen auf Grundlage der Mikroprogrammsteuerung.

Die Übergänge sind fließend. Alle Prinzpien lassen sich miteinander kombinieren, um Kostenvorgaben einzuhalten und Leistungsanforderungen zu erfüllen. Zumeist wird man daran denken, eine fertige Plattform zu programmieren. Wenn die Kosten im vorgegebenen Rahmen bleiben, wird man in den meisten Fällen das Anwendungsproblem dem Prinzip nach als gelöst betrachten und weitere Bemühungen um grundsätzliche Alternativen unterlassen. Wird es aber aufwendiger, muß man sich nach solchen Alternativen umsehen. Nun könnte man universelle Prozessoren außerhalb der üblichen Industriestandards und routinemäßigen Gepflogenheiten einsetzen, beginnend mit einer sehr maschinennahen Programmierung bis hin zum Stand-Alone-Programm, das die Maschine voll und ganz für sich hat, also nicht vom Overhead einer Systemsoftware gleichsam gebremst wird. Darüber hinaus kann man den Prozessor mit anwendungsspezifischen Schaltungen ergänzen (Hardware-Software-Co-Design) oder als ein Bauelement wie alle anderen ansehen, das in selbst entwickelte Anwendungsschaltungen eingebaut und womöglich trickreich angesteuert wird. Mit programmierbaren Schaltkreisen (FPGAs) und Entwicklungssystemen, die den Schaltungsentwurf mittels Verhaltensbeschreibung unterstützen, kommt die anwendungsspezifische Schaltung auch in sozusagen zivilen Projekten[5] als Alternative in Betracht. Das Prinzip der Mikroprogrammsteuerung ergibt Lösungsansätze, die Kompromisse zwischen beiden Extremen darstellen (Abb. 1.2). Das Mikroprogrammsteuerwerk ist einfacher, leistungsfähiger und flexibler als die meisten fertigen Prozessoren, ist leichter zu entwerfen als eine reine Schaltungslösung und – auf der Maschinenebene – praktisch ebenso zu programmieren wie ein üblicher Prozessorkern oder Mikrocontroller.

Automatentakt und Maschinenzyklen

Die Theorie der abstrakten Automaten kennt nur diskrete Taktzeitpunkte, reale Steuereinrichtungen arbeiten hingegen in realen Anwendungsumgebungen. Sie sind somit an reale Zeitraster und zeitliche Anforderungen gebunden. Eingangssignale sind aus der Außenwelt einzulesen, Ausgangssignale zu liefern. In den meisten Anwendungsfällen werden diese Abläufe zyklisch wiederholt. Abb. 1.3 veranschaulicht einen solchen Maschinenzyklus[6].

3. Durchaus auch in höheren Programmiersprachen. Man sollte aber wissen, was in der Maschine abläuft (think low-level, write high-level; [62]). Programmoptimierungsrichtlinien (der Hersteller) beachten!
4. Beispielsweise ein Realzeit-Betriebssystem.
5. Also im Rahmen üblicher Termin- und Kostenvorgaben sowie eines typischen Standes an Fachkenntnissen.
6. Wir reden von Maschinenzyklen im Sinne der Automatentheorie, nicht von den Maschinen- und Taktzyklen der Schaltungen. Ein Maschinenzyklus ist hier die Zeit vom Einlesen der Eingangssignale bis zum Ausgeben der Ausgangssignale, gleichgültig mit welchen Mitteln (Hardware oder Software) diese Funktionen implementiert werden.

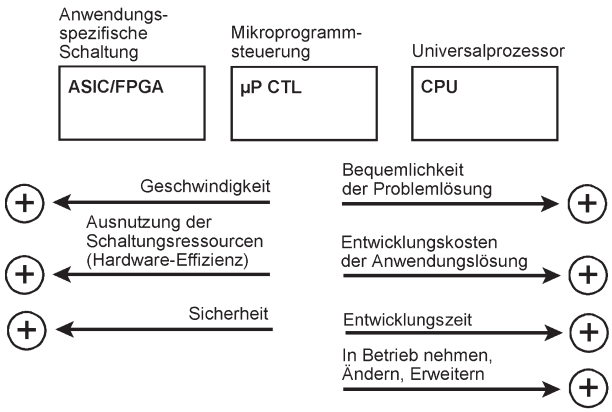

Abb. 1.2 Die Mikroprogrammsteuerung steht – als weitere Alternative der Problemlösung – zwischen der anwendungsspezifischen Schaltung und dem Universalprozessor. Alle drei Prinzipien lassen sich auf vielfältige Weise miteinander kombinieren. Die Pluszeichen deuten an, auf welcher Seite die jeweiligen Vorteile liegen.

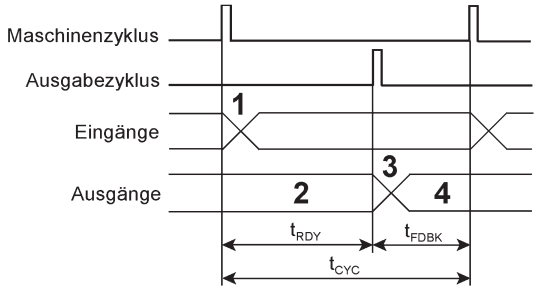

Abb. 1.3 Der typische Maschinenzyklus einer Steuereinrichtung.

Die Eingangssignale werden eingelesen (1). Daraus werden die aktuellen Werte der Ausgangssignale berechnet (2). Die Ausgangssignale werden ausgegeben (3). Nun könnte man den Maschinenzyklus beenden und den nächsten beginnen. Das ist aber nicht immer zweckmäßig. Welche Reaktion der Außenwelt ist zu erwarten, wenn sie noch keine Gelegenheit hatte, auf die ausgegebenen Signale zu reagieren? Also ist es oftmals sinnvoll, eine gewisse Zeit zu warten, bis der nächste Maschinenzyklus beginnt (4). Die Zykluszeit t_{CYC} ergibt sich somit aus der Zeit, die benötigt wird, um die Werte der Ausgangssignale zu bestimmen (t_{RDY}) und der Wartezeit, die eingeräumt wird, damit die Außenwelt reagieren kann (t_{FDBK}). Das Zeitraster der Maschinenzyklen ergibt sich als Vorgabe aus dem Anwendungsproblem. Es versteht sich von selbst, daß das Anwendungsproblem nur dann lösbar ist, wenn es möglich ist, Steuereinrichtungen zu nutzen oder zu entwickeln, die diese Anforderungen einhalten können. Abb. 1.4 veranschaulicht grundsätzliche Anforderungen.

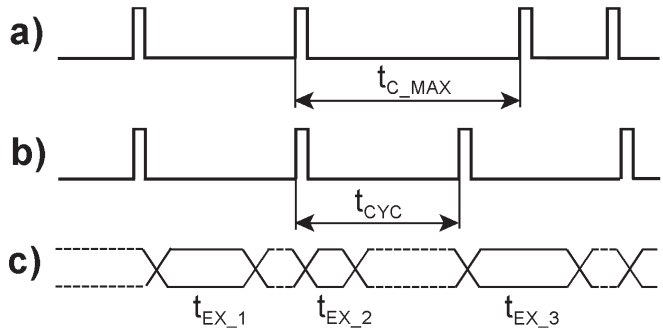

Abb. 1.4 Grundsätzliche Anforderungen an das Realzeitverhalten anhand von drei Beispielen.

a) Die Maschinenzyklen dürfen unterschiedlich lange dauern; es ist kein starres Zeitraster gefordert. In vielen Anwendungsfällen darf aber eine bestimmte maximale Zykluszeit t_{C_MAX} nicht überschritten werden[7].
b) Es ist ein starres Zeitraster gefordert, hier gekennzeichnet durch eine gleichbleibende Zykluszeit t_{CYC}.
c) Ein starres Zeitraster einzuhalten ist dann ein Problem, wenn die Abläufe unterschiedlich lange dauern können, hier veranschaulicht durch Ausführungszeiten t_{EX_1}, t_{EX_2} usw. Vor allem Softwarelösungen weisen ein solches Zeitverhalten auf. In jedem Maschinenzyklus wird eine Programmschleife ausgeführt, die unterschiedlich lange dauert, je nachdem, welche Zweige durchlaufen werden. Manchmal kann man die Laufzeiten der Programmzweige bestimmen und ggf. Wartebefehle oder Warteschleifen einfügen. Das gelingt aber nur bei sehr einfachen Programmabläufen und in Maschinen, deren Befehle exakt gleichbleibende Ausführungszeiten haben[8]. Ansonsten sind zusätzliche Maßnahmen der Zeitkontrolle und Ablaufsteuerung erforderlich.

Latenz- und Reaktionszeiten

Beide Zeitkennwerte betreffen Anforderungen, die in der Außenwelt auftreten. Die Latenzzeit (Latency) ist die Zeit zwischen dem Auftreten der Anforderung und dem Beginn der Erledigung. Die Reaktionszeit ist die Zeit zwischen dem Auftreten der Anforderung und der ersten Reaktion darauf, die sich in der Außenwelt bemerkbar macht (Abb. 1.5).

Manche Latenzzeiten sind schaltungstechnisch bedingt. Sie werden im Datenmaterial der Prozessoren genannt. Das betrifft unter anderem die Zeit zwischen der Unterbrechungsauslösung und dem Beginn der Unterbrechungsbehandlung. Diesen Werten liegen aber immer die günstig-

7. Beispielsweise eine Halbwelle des Netzsinus (< 8 ms bei 60 Hz Netzfrequenz) oder ein Zeitintervall, das sich aus den Anforderungen der Mensch-Maschine-Schnittstellen ergibt, z. B. wenigsten 20mal je Sekunde die Anzeigen zu aktualisieren und die Tasten abzufragen (< 50 ms).
8. Keine Caches, kein virtueller Speicher, keine Interrupts, keine Systemsoftware, die das laufende Anwendungsprogramm unterbricht, um anderen Anwendungen Laufzeit zuzuteilen.

sten Betriebsverhältnisse zugrunde (Beispiel: die Unterbrechung ist zugelassen, es sind keine Unterbrechungen höherer Priorität anhängig, der vorhergehende Befehl hat die kürzeste Ausführungszeit, es gibt keine Wartezustände).

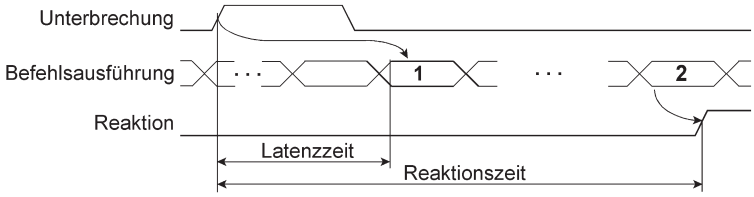

Abb. 1.5 Latenz- und Reaktionszeit am Beispiel einer Unterbrechungsbehandlung. 1 - der erste Befehl der Unterbrechungsbehandlung; 2 - dieser Befehl veranlaßt die erste Reaktion in der Außenwelt, beispielsweise das Einschalten einer Leuchtanzeige.

In der Praxis sind aber stets die jeweils ungünstigsten Verhältnisse anzusetzen. Und da können aus wenigen hundert Nanosekunden ohne weiteres viele Millisekunden werden ... Beispiel: die Unterbrechung ist zeitweise verhindert, weil im Moment gerade der Betriebssystemkern läuft, es sind noch mehrere Unterbrechungen höherer Priorität anhängig, und das betreffende Programm befindet sich nicht im Arbeitsspeicher, muß also erst geladen werden. Das typische Kennzeichen einer guten Realzeitumgebung sind eindeutig voraussehbare Latenz- und Reaktionszeiten. Die typischen universellen Systemplattformen können diese Bedingung prinzipbedingt nicht erfüllen, da sie für Anwendungsgebiete entwickelt wurden, in denen andere Anforderungen[9] Vorrang haben. Präzise vorhersagbare Latenz- und Reaktionszeiten ergeben sich, wenn man alle störenden Einflüsse fernhält (hierzu gehören unter anderem Versuche, auf der Maschine beliebige weitere Anwendungen laufen zu lassen oder von den Realzeitprogrammen aus direkt aufs Internet zuzugreifen). In letzter Konsequenz führt dies zur Arbeitsteilung zwischen Industriestandard- und Realzeitmaschinen, wobei die harten Realzeitaufgaben mit Stand-Alone-Programmen oder mit selbst entwickelten Schaltungslösungen erledigt werden.

1.2 Ein Blick in die Entwicklungsgeschichte

Der Steuerungsentwurf war traditionell ein Spezialgebiet besonders befähigter Entwickler, die intuitiv und erfinderisch an die Probleme herangegangen sind. Auf diese Weise sind geniale Entwicklungsleistungen erbracht worden, ganz ohne Mikrocontroller, RISC-Prozessoren, FPGAs und aus dem Internet herunterzuladende Betriebssysteme[10].

9. Leichte Bedienbarkeit, graphische Mensch-Maschine-Schnittstellen, Kommunikation, Unterstützung großer Datenbestände, das gleichzeitige Ausführen mehrerer Anwendungen usw.
10. Die Literatur zur Technikgeschichte und die technischen Museen bieten eine Vielzahl an Beispielen. Um wenigstens zwei zu nennen, sei auf die Kommando- und Bediengeräte von Flugmotoren und auf die Kurssteuerung von Torpedos verwiesen ([1], [2]).

Die Automatisierungs- und Kommunikationstechnik (beginnend u. a. mit den Chemieanlagen, den automatisierten Werkzeugmaschinen und der Telefonvermittlung) hat zu einem solchen Bedarf an Steuerungslösungen geführt, daß systematische Verfahren gefunden werden mußten, die es auch dem Durchschnittsfachmann ermöglichen, solche Aufgaben zu bearbeiten. Für unser Anliegen sind vor allem zwei Ansätze vom Bedeutung, die Automatentheorie und die Boolesche Algebra.

Die Theorie der abstrakten Automaten (Automatentheorie) stellt Begriffe und theoretische Grundlagen bereit, mit denen man Steuerautomaten formal beschreiben kann, unabhängig von ihrem Einsatzzweck und ihrer technischen Realisierung. Viele – auch sehr komplexe – Steuerungsprobleme kann man lösen, indem man die Steuerung als Zustandsautomaten entwirft, entweder als einen einzigen oder als einen Verbund von mehreren[11].

Relaisschaltungen und andere Anordnungen aus zweiwertig arbeitenden Funktionselementen kann man mit den formalen Mitteln der Aussagenlogik bzw. Booleschen Algebra beschreiben (Schaltalgebra).

Weitere Beschreibungsmittel haben sich aus der Anwendungspraxis ergeben. Typische Beispiele sind Kontaktpläne (Ladder Diagrams), Ablaufpläne (Flußdiagramme), Funktionspläne, Zustandsdiagramme (State Charts) sowie anwendungsspezifische Programmier- und Beschreibungssprachen.

Jedes Beschreibungsmittel ist für bestimmte Arten von Aufgaben besser geeignet als für andere. So nützt ein Zustandsdiagramm nicht viel, wenn Formelausdrücke zu berechnen sind. Flußdiagramme können sequentielle Abläufe anschaulich darstellen. Sie sparen aber nur dann Arbeit, wenn man nicht jede Einzelheit graphisch darstellen muß. Deshalb erfaßt man im Flußdiagramm nur den grundsätzlichen Ablauf und beschreibt das, was die Blöcke im einzelnen leisten sollen, in einer Programmiersprache.

Die Nutzung des Universalrechners und damit die Lösung von Steuerungsaufgaben durch sozusagen gewöhnliches Programmieren kam erst später auf. Steuerungstechnik und Rechentechnik haben sich zunächst unabhängig voneinander entwickelt. Es waren unterschiedliche Fachgebiete.

Die ersten speicherprogrammierbaren Steuerungen (SPS) waren Spezialmaschinen zum Emulieren von Relaisschaltungen, die mit Kontaktplänen (Ladder Diagrams) beschrieben wurden[12]. Für die Anwender wurden Programmschnittstellen bereitgestellt, die an die traditionellen Gepflogenheiten des Steuerungsentwurfs angepaßt waren. Die herkömmliche SPS-Programmierung ist im Grunde ein klassischer Steuerungsentwurf über Kontaktpläne, Anweisungslisten, Programmablaufpläne usw., die vom Prozessor der SPS emuliert werden. Solche Programmschnittstellen gibt es auch in modernen Systemen und in einschlägigen Standards. Dabei bleibt man manchmal weit unter dem, was eigentlich problemlos möglich wäre.

11. Als kleine einführenden Literaturauswahl sei auf [3] bis [10] verwiesen.
12. Der Fachbegriff im Englischen: PLC = Programmable Logic Controller. Mit einfachen Booleschen Verknüpfungen zum Emulieren von Relaisschaltungen beginnend (als Beispiel siehe [11] bis [14]) ist der Funktionsumfang nach und nach immer komplexer geworden.

Manche Entwicklungssysteme unterstützen sogar schon die Boolesche Algebra ... Man kann aber nicht einfach den jeweils gewünschten Zusammenhang als Booleschen Funktionsausdruck hinschreiben, womöglich verschachtelt, mit Klammern usw., sondern muß am Bildschirm Gattersymbole anfordern und entsprechend beschalten oder die Booleschen Verknüpfungen Variable für Variable ausprogrammieren.
Moderne SPS beruhen auf Mikrocontrollern und Computermodulen, die so verpackt sind, daß sie die Einsatzbedingungen in Industrieumgebungen aushalten. Auch in der Anwendungsprogrammierung werden sie mehr und mehr als vollwertige Universalrechner angesehen. Man löst dann eine Steuerungsaufgabe nicht mehr dadurch, daß man eine Relaisschaltung entwirft, die auf einer zeitgemäßen Plattform emuliert wird, sondern man denkt von Anfang an in Software – die Steuerung ist im Grunde ein Programm, ausgeführt auf einer passenden Maschine (Soft-SPS). Die Programmentwicklung wird durch Programmiersprachen und Entwicklungssysteme unterstützt, die eigens für diesen Anwendungsbereich ausgelegt sind[13].

Die Steuerung soll programmierbar sein
Die freie Programmierbarkeit hat sich in der Entwicklungsgeschichte nur allmählich durchgesetzt. In der ersten Zeit war sie teuer. Man konnte gar nicht daran denken, universelle programmierbare Computer zu verwenden. Nicht selten waren gerade außergewöhnlich befähigte Entwickler stolz darauf, ohne auszukommen (Draht als beste Programmiersprache). Wieso wird eine Rakete besser, wenn eine Rechenmaschine mitfliegt? Der Höhepunkt diesen Entwicklung war wohl das vollautomatische Rendezvous mit anschließender Kopplung (Docking) zweier Raumflugkörper in der Erdumlaufbahn[14].
Irgendwann wurde diese Auffassung überwunden. Nun sollte möglichst alles programmierbar sein. Die Voraussetzungen waren die Verfügbarkeit ausreichend kleiner, kostengünstiger Computer, die höheren Programmiersprachen und die Verbreitung des grundsätzlichen Gedankens, Probleme durch Programmieren zu lösen.
Wenn man den Grundgedanken des Programmierens erst einmal verinnerlicht hat, gibt es auch nichts, das näher liegt. Im Grunde ist es das uralte Kommandoprinzip, die Dienstvorschrift. Ersten tue dies, zweitens tue jenes, wenn der Fall A vorliegt, gehe weiter mit Schritt 150, wenn der Fall B vorliegt, mit Schritt 210 usw.
Die Entwickler der Vergangenheit haben sich eine Lösung der Anwendungsaufgabe einfallen lassen und hierzu die jeweils verfügbaren (zuhandenen) Prinzipien, Bauteile, Technologien usw. ausgenutzt, und das nicht selten äußerst erfindungs- und trickreich. Die meisten der heutigen Entwickler denken kaum daran, daß man das Problem womöglich auch ganz von Grund auf

13. Beispiel eines internationalen Standards: IEC 61131. Beispiele herstellerspezifischer Systeme: Opto 22 Pac Project und ioProject ([15]). Kontaktpläne, Funktionspläne, Anweisungslisten usw. werden weiterhin unterstützt. Das dient vor allem der Abwärtskompatibilität (Einsatz in herkömmlichen Anwendungsumgebungen). Auch möchte man den vielen Fachleuten entgegenkommen, die mit diesen Beschreibungsmitteln wohlvertraut sind.
14. In Anlehnung an [16]. Es ist eine umfassende, anschauliche Darstellung der Entwicklungsgeschichte und der unterschiedlichen Auffassungen der maßgeblichen Entwickler. [17] gibt einen kurzen Überblick zur Steuerungs- und Computertechnik in Raumfahrzeugen und nennt viele weitere Quellen.

lösen könnte, sondern fangen sofort an zu programmieren, wenn sie sich nicht gar darauf beschränken, nach vorgestanzten Lösungen zu suchen, die man einfach herunterladen kann.

1.3 Speicher- oder Schaltungsprogrammierung?

Der intuitive Steuerungsentwurf und die daraus resultierenden Tricklösungen sind kaum noch von Bedeutung. Das hat sich aus den Anforderungen der Praxis ergeben. Oftmals sind Vorschriften zu befolgen, die tief in die Einzelheiten gehen. Lösungsprinzipien, Entwicklungsabläufe und Plattformen werden vorgegeben. Es wird gefordert, bestimmte Sprachumgebungen und Entwicklungswerkzeuge zu nutzen und dabei Entwurfs- und Programmierrichtlinien (Coding Styles) einzuhalten. Die Lehrinhalte in Ausbildung und Studium sind seit längerem darauf gerichtet[15].

Die Steuerungsentwicklung in der industriellen Praxis, die einschlägigen Standards usw. wollen wir aber nicht näher betrachten. Unser Ausgangspunkt ist die volle Freiheit beim Suchen nach Problemlösungen. Dabei wollen wir methodisch und systematisch vorgehen. Wir beginnen mit theoretisch wohlbegründeten Modellvorstellungen, Beschreibungsmitteln und Lösungsansätzen (Zustandsautomat, Programmablaufplan, Boolesche Algebra usw.) und machen uns erst einmal klar, wie das zu lösende Problem beschaffen ist (Tiefenstruktur). Daraus ergibt sich, was – als Gebrauchswert – eigentlich herauskommen soll. Dann überlegen wir, wie wir die Entwurfsaufgabe lösen. Die typische Aufgabe ist das Entwickeln einer vergleichsweise komplexen Steuerungslösung unter Nutzung zeitgemäßer Technologien. Dabei wollen wir ausschließlich mit programmierbarer Hardware arbeiten, so daß wir die Problemlösung immer wieder ändern, anpassen und aktualisieren können, ohne die Hardware umbauen zu müssen[16]. Es gibt zwei Alternativen, die – fertig bezogene oder selbstgebaute – Maschine mit programmierbarem Speicher und die programmierbare Schaltung im CPLD oder FPGA.

Weshalb wünscht man sich eigentlich die freie Programmierbarkeit?

Immerhin hat auch der Schaltungsentwurf etwas für sich: wir haben die größtmögliche Gestaltungsfreiheit, können alles so auslegen wie wir wollen und sind nicht an die Konventionen einer vorgegebenen Rechner- und Systemarchitektur gebunden. Draht ist manchmal tatsächlich die beste Programmiersprache. Komplizierte Spezialschaltungen aus der anwendungsseitigen Problemstellung heraus zu entwickeln ist aber eine anspruchsvolle Aufgabe, deren Lösung außergewöhnliche Fähigkeiten voraussetzt. Der Entwicklungsgang über erfinderische Intuition und Versuch und Irrtum dauert seine Zeit und ist fehleranfällig.

Nun wird man sich heutzutage die Arbeit erleichtern, indem man den Entwurf auf der Register-Transfer-Ebene (RTL) erfaßt oder gar nur das gewünschte Verhalten formal beschreibt und die

15. Wer sich heutzutage der Steuerungsentwicklung zuwendet, wird wohl lieber programmieren als Schaltungen entwerfen; nur eine Minderheit dürfte Draht als die beste Programmiersprache ansehen. Auch sind die Grundlagen der Zustandsautomaten und der Schaltalgebra, die standardisierten Modellierungs- und Entwurfssprachen (wie UML, IEC 61131, VHDL, Verilog usw.) weit verbreitete Lehrinhalte.
16. Also ohne die Leiterplatten, Verdrahtung und Verkabelung zu ändern.

Implementierung der Schaltungen dem Entwicklungssystem überläßt (Schaltungssynthese). Das führt aber auf Probleme der Booleschen Algebra. Manche Algorithmen der Schaltalgebra, die tief im Innern der Entwicklungssoftware ausgeführt werden, haben eine NP-vollständige Zeitkomplexität. Es kann sein, daß alles in kurzer Zeit erledigt ist. Der Zeit- und Speicherplatzbedarf kann aber auch in die Größenordnung kommen, die erforderlich wäre, die gesamte Lösungsmenge des Anwendungsproblems zu bilden und zu durchmustern (exponentielle Abhängigkeit mit $O(2^n)$ oder höher). Mit anderen Worten, es kann dauern. Das ist auch dann der Fall, wenn die Entwurfsabsicht auf eine Weise erfaßt wird, die wie gewöhnliches Programmieren aussieht (Verhaltensbeschreibung). Das Beschreiben ist leicht, das Synthetisieren dauert, das Fehlersuchen ist schwierig. Sehr komplexe Anwendungsprobleme ausschließlich mit Schaltungsentwurf zu lösen (Spezialhardware), kostet Zeit und Geld, nicht selten ausgesprochen viel. Es ist vergleichsweise aufwendig, Entwurfsfehler zu erkennen und zu beseitigen (Fehlersuche in der Hardware, mehrere Iterationen der Schaltungssynthese).
Demgegenüber ist es eine Erfahrungstatsache, daß man auch sehr komplexe Probleme durch Programmieren lösen kann. Das gelingt deshalb, weil alles von Anfang an auf elementare Verarbeitungsschritte zurückgeführt wird und die natürliche Intelligenz bereits bei der Problemerfassung zur Wirkung kommt[17]. Programmieren ist immer irgendwie Tricksen und Hacken ...
Grundsätzlich geht es darum, die Komplexität zu beherrschen und die Kosten gering zu halten. Das ergibt sich, indem wir die Anwendungslösung aus zwei Komponenten aufbauen, aus der gegenständlichen (physischen) Maschine (Hardware) und dem gespeicherten Programm (Software). Die Maschine ist universell nutzbar und vergleichsweise einfach. Da es sich um vergleichsweise einfache Wirkprinzipien handelt, können nicht allzu viele Entwurfsfehler[18] unterlaufen. Die eigentliche Komplexität der Anwendungslösung steckt im Programm, also nicht mehr in der Schaltung, sondern in einem Speicherinhalt. Die meisten Entwurfsfehler sind dann keine Schaltungsfehler, sondern Programmierfehler, die meisten Änderungen sind keine Schaltungsänderungen, sondern Programmänderungen. Wenn die Hardware erst einmal läuft, kann man unbekümmert hacken und schustern, bis es einigermaßen funktioniert, ohne daß dabei Synthesealgorithmen mit NP-vollständiger Komplexität durchlaufen werden müssen, wenn alles soft ist, kann man so lange beim Kunden ändern, wie der sich das gefallen läßt ...

Speicherprogrammierung
Eine Maschine mit programmierbarem Speicher kann man sich leicht vorstellen. Jeder Mikrocontroller oder jede Computerplattform kann als Beispiel dienen. Man schreibt ein Programm, läßt es übersetzen und lädt es hinein. Wenn sich Fehler zeigen, ändert man das Programm und probiert von neuem. Es ist alles wirklich "soft", man muß keine Leiterzüge auftrennen, Drähte löten oder gar neue Platinen entwickeln.

17. Es ist experimentell bewiesen worden, daß man es auf diese Weise bis zum Mond und zurück schafft, und zwar mit weit weniger als 100 kBytes Speicherkapazität ...
18. Ja, in der Praxis sind es immer noch genug, erkennbar u. a. aus den sog. Errata Sheets der Prozessorhersteller. Beim Entwickeln komplexer Einzweckschaltungen kann aber noch viel mehr schiefgehen ...

Schaltungsprogrammierung
Auch die CPLDs und FPGAs sind programmierbar. Um die einzuprogrammierenden Schaltungen zu entwerfen, ist es keineswegs mehr nötig, Gatter für Gatter und Flipflop für Flipflop am Bildschirm aufzurufen und miteinander zu verbinden (Schaltplanentwurf). Moderne Entwicklungsumgebungen unterstützen den Schaltungsentwurf auf Grundlage der Verhaltensbeschreibung. Das gewünschte Verhalten wird mit einer Hardwarebeschreibungssprache beschrieben, gelegentlich auch mit einer entsprechend erweiterten "gewöhnlichen" Programmiersprache. Die Verhaltensbeschreibung sieht fast wie ein richtiges Programm aus. Eigentlich sind Verilog und C, VHDL und Ada usw. gar nicht so weit auseinander. Ist das Steuerungsproblem nicht allzu komplex, können wir gleichsam naiv herangehen. Womöglich kommen wir ohne tiefergehende Kenntnisse der Digitaltechnik aus. Wir kümmern uns dann gar nicht um die Schaltungsstrukur und formulieren die Verhaltensbeschreibung wie einen üblichen Programmtext – tun also im Grunde so, als würden wir einen Mikrocontroller programmieren. Alles weitere überlassen wir der Schaltungssynthese. Ein solcher Entwicklungszyklus entspricht nahezu dem der Mikrocontrollerprogrammierung (Quellcode ändern – Schaltung synthetisieren lassen – Schaltkreis programmieren – zusehen, ob es funktioniert usw.)[19].
Mit den heutigen Entwicklungswerkzeugen wird somit die Einzweckschaltung (wieder[20]) beherrschbar. Es kann sein, daß sie in Hinsicht auf Ressourcenbedarf, Stromaufnahme und Störstrahlung deutliche Vorteile aufweist. Das ist vor allem dann der Fall, wenn man alles in einem gemeinsamen Schaltkreis unterbringen kann, sowohl den Steuerautomaten als auch die anwendungsspezifischen E-A- und Schnittstellenschaltungen. Solche Schaltungen kann man in genauer Anpassung an die Anwendungsumgebung so entwerfen, daß nur implementiert wird, was man tatsächlich braucht[21].

Ablaufprogrammierung und Schaltungssynthese
Es gibt einen grundsätzlichen Unterschied:

- Der Speicherinhalt der speicherprogrammierten Maschine ergibt sich aus der Übersetzung der Programmierabsicht, die als Programmablaufbeschreibung erfaßt wird. Der Programmablauf ist also von Anfang an gegeben und muß nur noch ins Maschinenprogramm umcodiert werden.
- Die Programmierdaten des Schaltkreises (CPLD, FPGA) ergeben sich aus einer Schaltungssynthese.

19. Die Schaltungen in modernen CPLDs und FPGAs kann man sogar in ihrer Anwendungsumgebung ändern, also beim Kunden (Update). Zum Aktualisieren von Schaltungsstrukturen siehe die einschlägigen Applikationsschriften der FPGA-Hersteller, als historische Beispiele siehe [18] und [19].
20. Bezogen auf die Entwicklungsgeschichte. Bevor diese Schaltkreise und Entwicklungswerkzeuge aufkamen, hat man gelegentlich speicherprogrammierbare Lösungen sozusagen aus Vorsicht gewählt (mit anderen Worten, um das Entwicklungsrisiko gering zu halten), weil die Schaltungen einfach sind und die Komplexität im Speicherinhalt steckt, der sich leicht ändern läßt.
21. Im Gegensatz zum Mikrocontroller, in dem oftmals nur ein Teil der E-A-Ports und der peripheren Funktionseinheiten ausgenutzt wird.

Grundsätzliche Probleme der Schaltungssynthese

Die Verhaltensbeschreibung ist im Grunde eine Ablaufbeschreibung. Sie muß letzten Endes in eine Schaltungsstruktur umgesetzt werden. Zunächst ergeben sich Boolesche Funktionen, die dann mit den Ressourcen des Schaltkreises implementiert werden müssen (funktionelle Dekomposition). Diese Algorithmen weisen eine NP-vollständige Zeitkomplexität auf. Aus Sicht des naiven Anwendungsprogrammierers mit seiner Verhaltensbeschreibung sind die Ergebnisse nicht wirklich vorhersagbar. Boolesche Probleme haben ihre Eigenheiten mit Auswirkungen auf die Laufzeit der Synthese[22], auf den Verbrauch an Ressourcen (Zellen, Verbindungswege usw.) sowie auf die Schaltungstiefe und damit auf die Taktfrequenz, mit der man die Schaltung betreiben kann.

Kleine Änderungen können zur Folge haben, daß die Synthese viel zu lange dauert, daß man die Schaltung nicht mehr mit der vorgesehenen Taktfrequenz betreiben kann, oder daß sie zuviel Strom aufnimmt (Abb. 1.6 und 1.7). Womöglich paßt sie gar nicht mehr in den Schaltkreis[23].

Taktfrequenzen

Die Taktfrequenzkennwerte im Datenblatt eines CPLDs oder FPGAs betreffen Taktfrequenzen, mit denen die Taktsteuer- und Verteilerschaltungen, die Taktsignalwege usw. noch zurechtkommen, nicht aber die Taktfrequenz der einprogrammierten Anwendungsschaltung. Deren maximale Taktfrequenz ergibt sich aus den Verzögerungszeiten der synthetisierten Schaltungsstrukturen und Signalwege. Eine 10fache Verlangsamung (Slowdown) ist ein brauchbarer pauschaler Richtwert. Ist das FGPA für 1 GHz spezifiziert, so kann eine typische Anwendungsschaltung mit etwa 100 MHz betrieben werden. Es können aber auch viel weniger sein ...

Beim gewöhnlichen Programmieren kommt so etwas nicht vor ...

Tief im Innern beruht die Schaltungssynthese auf Booleschen Funktionen und sehr komplexen Algorithmen. Nun ist die Boolesche Gleichung ein vielseitig nutzbares, universelles, elegantes Gebilde der Mathematik. Eigentlich ist sie viel zu schade, um sie nur zu minimiereren. Aber gerade infolge der Allgemeinheit ist es nicht möglich, die Besonderheiten des jeweiligen Anwendungsproblems auszunutzen, um die Komplexität der Programmabläufe zu verringern[24].

Die Schaltungssynthese ist deshalb ein kritischer Teil im Entwicklungsablauf. So liegt es nahe, nach Wegen zu suchen, sie zu vermeiden oder zumindest auf Schaltungen überschaubarer Komplexität zu beschränken.

Beim gewöhnlichen Programmieren wird die Entwurfsabsicht von Anfang an als Programmablauf erfaßt und vom Entwicklungssystem nur in den Maschinencode umgesetzt. Eine speicherprogrammierte Maschine ist bequemer umzuprogrammieren, weil es genügt, den Speicherinhalt zu ändern. Das Fehlersuchen entspricht dem üblichen Debugging von Programmen. Eine Schal-

22. Womöglich Stunden und mehr ...
23. Das Entwicklungssystem gibt dann beispielsweise eine Fehlermeldung "Does not fit" aus.
24. Nach [21] besteht ein Nachteil dieses Ansatzes (mit Booleschen Gleichungen Anwendungsprobleme zu lösen) darin, daß „... die Aufgaben ihre Spezifik einbüßen, mit der man die Lösungssuche beschleunigen könnte.“

tung müßte hingegen neu synthetisiert werden. Das Fehlersuchen in Schaltungen ist zumeist erheblich schwieriger. Je größer die Schaltung, desto länger dauert die Synthese, desto mehr Entwurfsfehler können unterlaufen. Ein Großvorhaben sollte deshalb sorgfältig geplant und von Hand[25] in überschaubare Funktionseinheiten aufgeteilt werden.

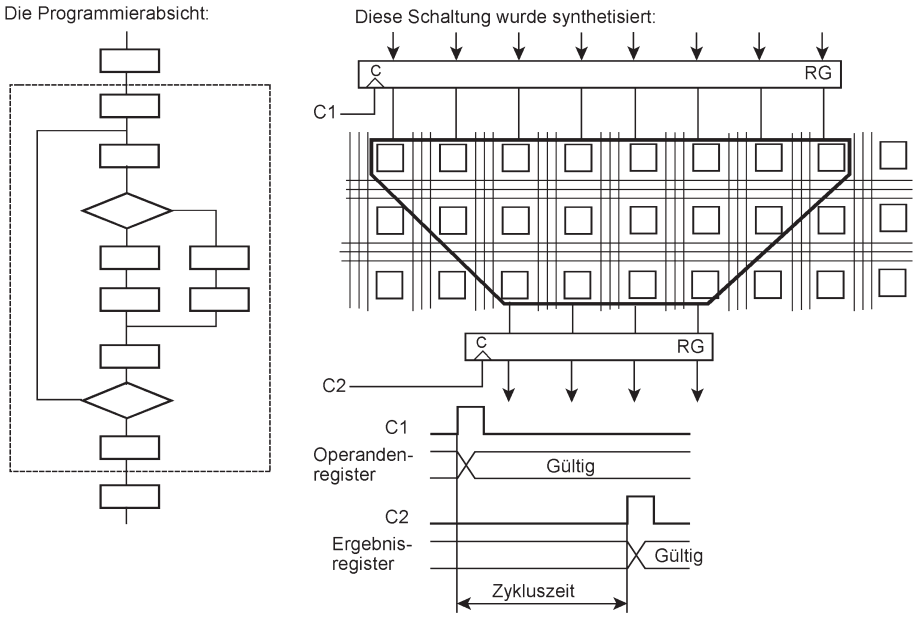

Abb. 1.6 Schaltungssynthese (1). Die als Verhaltensbeschreibung erfaßte Entwurfsabsicht (links) wird in eine Schaltung umgesetzt und mit den Logikzellen des FPGA implementiert. In diesem Beispiel klappt alles. Es werden nicht allzu viele Zellen verbraucht, und die Schaltungstiefe ist gering. Somit kann die Schaltung mit einer entsprechend hohen Taktfrequenz betrieben werden.

Ein Blick in die Fehlersuch- und Änderungspraxis

Der grundsätzliche Unterschied zwischen Programmübersetzung und Schaltungssynthese macht sich in der Praxis vor allem beim Fehlersuchen und Ändern bemerkbar. Häufig muß man Fehlerhypothesen bilden ("es könnte daran liegen, daß ...."). Wie finden wir heraus, ob eine solche Hypothese zutrifft oder nicht? Es liegt nahe, sie zunächst für wahr zu halten und das Programm oder die Schaltung probeweise zu ändern. Hat die Änderung geholfen? Wenn nein, wieder rückgängig machen und sich neue Fehlerhypothesen einfallen lassen. Wenn ja, die Änderung endgültig ausarbeiten und einbauen. Womöglich kommt man durch bloßes Nachdenken allein nicht auf brauchbare Fehlerhypothesen. Also genauer beobachten, was eigentlich abläuft. Auch dazu muß man gelegentlich ändern. Man muß beispielsweise Ausgaben auf ein

25. Soll heißen: Mitdenken und von der natürlichen Intelligenz ausgiebig Gebrauch machen. Die Architektur des Projekts nicht blindlings der Entwicklungssoftware überlassen ...

Bedienterminal einfügen oder Signale aus dem FPGA nach außen führen, um sie mit einem Logikanalysator aufzuzeichnen. Die Grundsatzfrage ist, wie lange es dauert, so etwas einzubringen. Einen Speicherinhalt kann man vergleichsweise schnell von Hand ändern. Eine geänderte Schaltung muß hingegen neu synthetisiert werden.

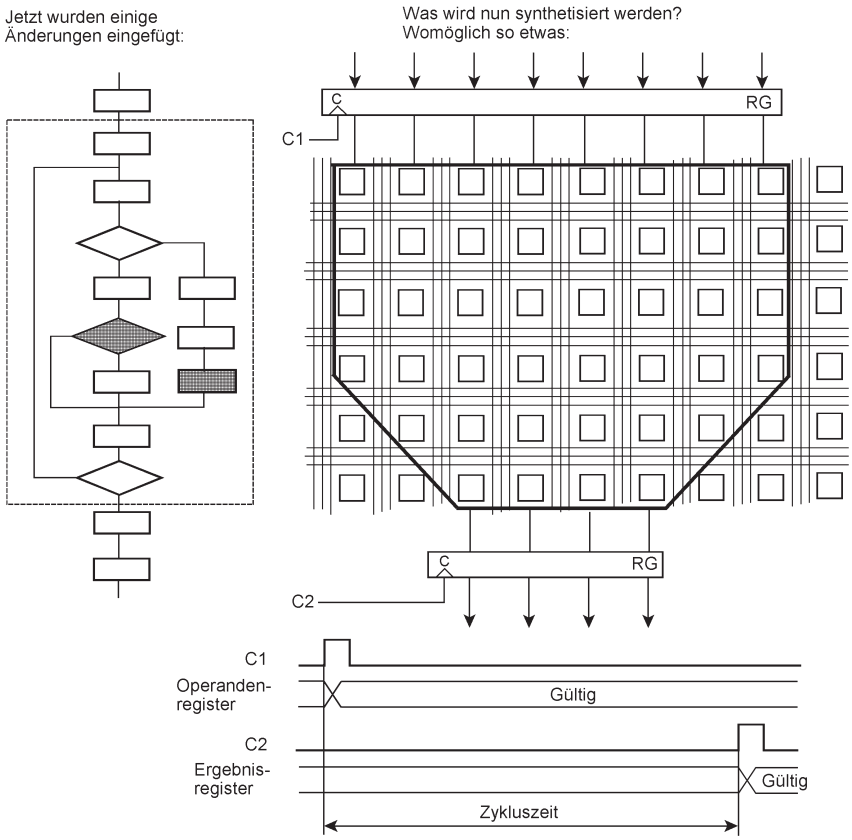

Abb. 1.7 Schaltungssynthese (2). Die Verhaltensbeschreibung von Abb. 1.6 wurde geringfügig geändert. Nur ein paar Anweisungen ... Daraus erzeugt der Synthesealgorithmus aber eine viel größere Schaltung[26]. Die Schaltungstiefe ist so groß geworden, daß man die Taktzyklen beträchtlich verlängern und die Taktfrequenz entsprechend herabsetzen muß.

Viele typische Entwurfsfehler sind im Grunde Kleinigkeiten

Wie lange dauert es, sie zu beseitigen? Abb. 1.8 veranschaulicht einfache Hardware- und Programmierfehler.

26. Das muß nicht immer so sein, aber kann vorkommen.

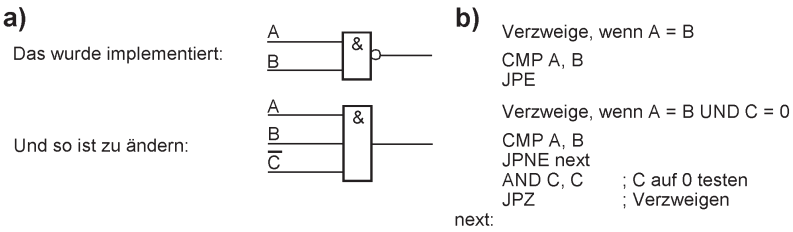

Abb. 1.8 Kleinigkeiten ändern. Zwei Beispiele. a) Hardware; b) Software.

a) Hardware. Ein Gatter ist zu ändern. Denken wir an eine Maschine mit einigen hunderttausend Gattern. Früher war es ein Schrank, jetzt ist es ein FPGA. Ganz früher wurde so geändert: Leiterplatte raus, nachsehen, in welchen Schaltkreisen noch passende Gatter frei sind, Leiterzüge auffräsen, Draht löten. Wenn man schustern darf, dauert es etwa eine halbe Stunde. Wenn es ein förmlicher Änderungsantrag sein muß, mit Genehmigung vom Chef und Ausführung in der Zentralwerkstatt, ist es nach oben offen ... Im FPGA ist es aber unmöglich, schnell mal ein Gatter von Hand auszuwechseln. Vielmehr muß die Schaltungssynthese erneut durchlaufen werden. Es fragt sich, wie lange das dauert und was dabei herauskommt[27] ...

b) Software. Ein Verzweigungsablauf ist zu ändern. Ganz früher wurde zunächst probeweise von Hand geändert. Zu Zeiten des S/360 und der PDP-11 konnte man die Befehle am Bedienpult mit Schaltern eingeben. Wort für Wort einstellen und mit LOAD oder DEPOSIT laden. Dauerte ein paar Minuten. Ehrensache, die Befehle auswendig zu wissen ... Heute ist es nur noch in Ausnahmefällen zweckmäßig, probeweise in den Maschinencode einzugreifen. Die Entwicklungsumgebungen sind schnell genug. Es würde viel länger dauern, die Bitmuster der Maschinenbefehle von Hand aufzusetzen[28], als das betreffende Programm-Modul neu zu compilieren und zu laden. Damit die Änderungszeit (Turn-Around Time) auch bei großen Projekten erträglich bleibt, muß man die Software sorgfältig strukturieren[29]. In Projekten mit sehr maschinennaher Programmierung müssen diese Probleme von Anfang an bedacht werden und in die Konzeption der Entwicklungsabläufe sowie in die Termin- und Kostenplanung einfließen. Erforderlichenfalls müssen Hilfsmittel und Werkzeuge beschafft oder selbst entwickelt werden. Das alles ist aber lösbar. Die Zeitkomplexität der Programmänderungen ist somit stets deutlich geringer als die der Schaltungssynthese. Die Schaltungssynthese ist NP-vollständig, das Schreiben und Ändern der Programme nicht.

27. Mehrverbrauch an Zellen, Verlängerung der Durchlaufzeiten (vgl. die Abb. 1.6 und 1.7).
28. Man muß nicht nur die Befehlscodierung bis aufs Bit kennen, sondern auch die Konventionen der Registernutzung, die Eigentümlichkeiten des Befehlspipelining, der Caches und des virtuellen Speichers, muß die Adreßrechnung womöglich mit Papier und Bleistift nachvollziehen usw. Im Maschinenprogramm ändern ist eine anspruchsvolle Aufgabe. Man überläßt sie besser der Maschine ...
29. Das betrifft u. a. die Zerlegung in Module, die unabhängig voneinander compiliert werden können.

2. Zustandsautomaten und Steuerautomaten

2.1 Der Steuerautomat als Zustandsautomat

2.1.1 Einführung

Der Zustandsautomat (Finite State Machine, FSM) ist ein bewährtes und theoretisch wohlbegründetes Modell des Steuerungsentwurfs. Es ist unter anderem dann zweckmäßig, von vornherein in Zuständen und Zustandsübergängen zu denken, wenn das zu implementierende Verhalten nacheinander auszuführende Schritte aufweist oder wenn es sich mit Wenn-So-Reaktionen beschreiben läßt, wobei die jeweilige Reaktion nicht nur von der aktuellen Eingabe, sondern auch von der Vorgeschichte abhängt. Die Entwurfsabsicht wird mit einschlägigen Beschreibungsmitteln dargestellt, die Schaltungssynthese vom Entwicklungssystem erledigt. Die Standardisierungsgremien, die Schaltkreishersteller und die Anbieter von Entwicklungssystemen haben sich auf diese Vorgehensweise eingestellt. Programmierbare Logikschaltkreise sind zum Implementieren vollsynchroner Schaltwerke optimiert. Die zeitgemäße Steuerschaltung ist deshalb der vollsynchrone Zustandsautomat[1]. Auch beim Programmieren ist es oftmals von Vorteil, in Zuständen und Zustandsübergängen zu denken. Das betrifft nicht nur Steuerungsalgorithmen im engeren Sinne, sondern u. a. auch Mensch-Maschine-Schnittstellen und Computerspiele (Games). Es gibt Entwicklungsumgebungen, die die Problemlösung mit Zuständen und Zustandsübergängen unterstützen[2].

Die Grenzen dieses Ansatzes ergeben sich aus der funktionellen Komplexität. Damit so ein Vorhaben beherrschbar bleibt, darf es nicht allzu viele Zustände geben, und die Übergänge zwischen ihnen dürfen nicht allzu kompliziert sein. Oftmals kann man diese Schwierigkeiten überwinden, indem man mehrere – jeweils für sich überschaubare – Zustandsautomaten im Verbund einsetzt. Wenn eine solche Zerlegung nicht gelingt, liegt es vor allem an der Kompliziertheit der Zustandsübergänge. Die Anzahl der Zustände ist praktisch bedeutungslos, da mit n Flipflops oder Bits bis zu 2^n Zustände darstellbar sind. Es bereitet also keine grundsätzlichen Schwierigkeiten, komplizierte Zustandsübergänge in mehrere aufeinanderfolgende überschaubare Übergänge aufzulösen und dafür weitere Zustände (Zwischen- oder Hilfszustände) einzuführen. Im Extremfall ist der einzelne Zustandsübergang nur ein Fortschreiten vom aktuellen Zustand zu einem einzigen Nachfolger oder eine Entscheidung zwischen zwei Folgezuständen. Es liegt dann nahe, die Überführungsfunktion nicht mit Gatternetzen, sondern mit gespeicherten Bitmustern darzustellen (speicherbasierte Steuerung). In letzter Konsequenz kann man auf programmtechnische Prinzipien übergehen (Ausprogrammieren der Zustandsübergänge), dafür aber optimierte Schaltungslösungen vorsehen, die in dieser Hinsicht mehr leisten als universelle Prozessoren (Mikroprogrammsteuerung).

1. Mit Synchronisierungsschaltungen an den Schnittstellen zur Außenwelt.
2. Als Beispiel sei auf die Programmiersprache UML verwiesen (vgl. beispielsweise [22] bis [25]).

Der Zustandsautomat ist die technische Verkörperung des deterministischen endlichen Automaten der abstrakten Automatentheorie. Abb. 2.1 zeigt ein Blockschaltbild, das sich ergibt, wenn man auf gleichsam naive Weise die Wirkprinzipien, wie sie die Automatentheorie beschreibt, in eine Schaltungslösung umsetzt. Es ist nur ein sehr pauschaler Überblick, der sowohl Mealy- als auch Moore-Automaten betrifft. Diese grundsätzlichen Automatentypen unterscheiden sich darin, wovon die Ausgabefunktion abhängt (Mealy = vom Zustand und den Eingängen, Moore = vom Zustand allein). Für die beiden Funktionen des abstrakten Automaten – die Überführungsfunktion und die Ausgabefunktion – ist jeweils ein Funktionszuordner vorgesehen. Die Überführungsfunktion ergibt den Folgezustand, die Ausgabefunktion die Signale an den Ausgängen. Die Funktionszuordnerblöcke in Abb. 2.1 sind in ganz allgemeinem Sinne zu verstehen. Abb. 2.2 veranschaulicht typische Alternativen.

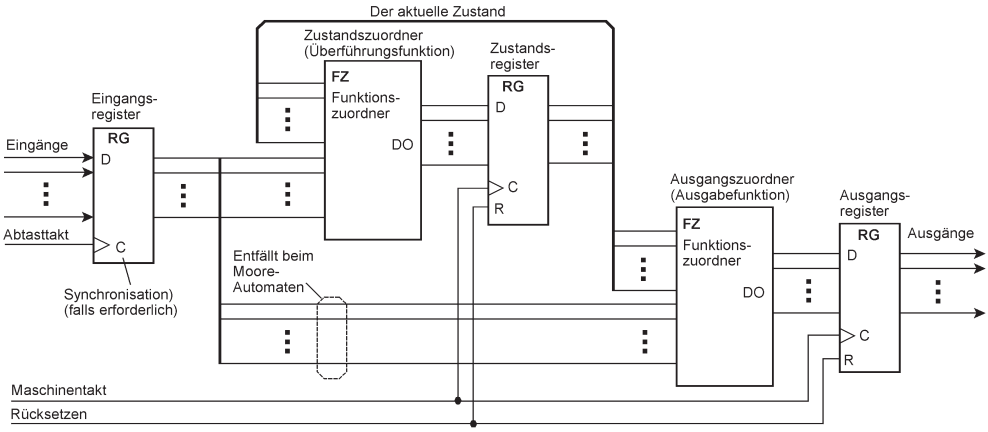

Abb. 2.1 Der Steuerautomat als Zustandsautomat.

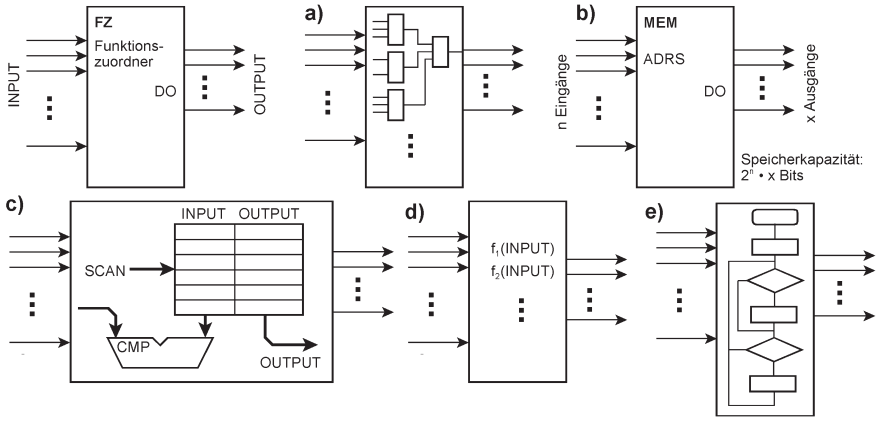

Abb. 2.2 Wie man einen Funktionszuordner implementieren kann.