41651

Schaltungsentwicklung/Entwurf

Begleitmaterial zur Vorlesung

Teil 1

Inhalts-Übersicht
Zwei Problemkreise sind im wechselseitigen Verbund anzugehen:

1. *Was* ist zu entwerfen?

1. Einführung
2. Die klassische Digitaltechnik
3. Problemlösung mit Mikrocontrollern
4. Problemlösung mit programmierbaren Schaltkreisen

2. Wie ist zu entwerfen?

1. Entwerfen über Schaltplan
2. Problemlösung durch (elementares) Programmieren
3. Entwerfen mittels Hardware-Beschreibungssprache (ABEL)
4. Die Hardware-Beschreibungssprache VHDL (Einführung)

41651
Schaltungsentwicklung/Entwurf

1. Der SERDES als Beispiel einer klassischen Digitalschaltung
2. CPLDs und FPGAs als Grundlage moderner Schaltungslösungen (Einführung)
3. Reguläre Steuerwerke
4. Mikrocontroller (Einführung)
5. Elementare Realzeitprogrammierung
6. Konfigurationen aus mehreren Mikrocontrollern

Praktisches Entwerfen:

a) Grundschaltungen in CPLDs
b) einfache Sequencer- und Controllerschaltungen
c) Mikrocontroller-Applikationen
d) Porterweiterungen für Mikrocontroller

Reguläre Steuerwerke:

- Datenflußsteuerung
- Sequencer
- State Machines
- Grundlagen der Programmsteuerung und Mikroprogrammierung

Mikrocontroller (Einführung):

- Controller und Prozessoren
- Signalprozessoren
- Speicheranschaltung
- E-A-Anschaltung
- E-A-Ausstattung
- Ausführungsbeispiele
- leistungssteigernde Zusatzbeschaltung
- Acceleratoren und Coprozessoren
- die Infrastruktur: Stromversorgung, Takt, Rücksetzen
- industrielle PC-Moduln (Einführung)

Elementare Realzeitprogrammierung:

- Abfragesteuerung
- Interruptsteuerung
- Ereignissteuerung
- Multitasking
- Boolesche Funktionen und State Machines in elementarer Software
- Vorkehrungen für Test, Debugging und Fehlerbehandlung

Einführung

Entwerfen heißt, eine bestimmte Aufgabe mit zuhandenen technischen Mitteln auf wirtschaftlich und anwendungspraktisch sinnvolle Weise zu lösen.

1. die Aufgabe

- Realzeitraster: in welcher Zeit ist es zu tun?
- Funktionalität: was ist zu tun? (die auszuführenden Informationswandlungen).

2. die Voraussetzungen und Nebenbedingungen

- Zuhandenheit,
- Infrastruktur,
- Kosten,
- Zeitbedarf (Einordnung in eine Terminplanung, Time to Market),
- Akzeptanz.

Das Realzeitraster

- in welcher Zeit ist die Aufgabe zu erledigen?
- einmalig oder zyklisch?
- Latenzzeit: von der Anforderung bis zum Beginn der Reaktion.

Ziel:
Durchführung der geforderten Informationswandlungen in der vorgegebenen Zeit t mit kostenoptimalen Mitteln (Kosten über alles = über den gesamten Lebenszyklus = Total Cost of Ownership).

Je geringer die Zeitvorgabe t und je größer die Kompliziertheit bzw. Komplexität, um so größer der Aufwand

Hinweis:
Komplexität und Kompliziertheit sind keine Wechselworte:

Komplexität beschreibt Ressourcen-Anforderungen des Algorithmus in Abhängigkeit von der Problemgröße. Beschreibung hat die Form O (n) (Order of... = Größenordnung von...). *Komplex* = mehr als linear mit der Problemgröße wachsende Anforderungen (Verarbeitungsleistung, Speicherplatz).

Kompliziertheit = Spitzfindigkeit, Verwickeltheit, Vielfalt der Funktionen. *Kompliziert* = verwickelt (sophisticated). Läßt sich näherungsweise durch Umfang der Problembeschreibung (= funktionelle Spezifikation) kennzeichnen.

Nutzung zuhandener Mittel und Entwicklungstechnologien

- t groß: das kostengünstigste zuhandene Mittel aussuchen: reicht es aus?
- t extrem klein: ist die Aufgabe überhaupt realisierbar? - Es kommen nur schaltungstechnische Lösungen in Frage. Erfinderische Bemühungen gelegentlich notwendig.

Zeitraster	Beispiele	technische Mittel
10 ns	Befehlsausführung in Prozessoren, Speicheransteuerung, Videodarstellung	Hardware, Gatter-Ebene (auch: Transistor-Ebene)
100 ns	Gerätesteuerung	Hardware auf Gatter-Ebene, State Machines, Mikroprogrammierung
1 µs	Gerätesteuerung	Hardware auf Gatter-Ebene, State Machines, Mikroprogrammierung, Maschinenprogrammierung (was sich mit 10...50 Maschinenbefehlen erledigen läßt)
1 ms	Gerätesteuerung, Regelungsaufgaben (Closed Loop)	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 (Retrieval)	komplexe Realzeit-Software, Vernetzung
1 s	Prozeßsteuerung, Informationsbeschaffung (Retrieval)	Realzeit- bzw. beliebige Software (je nach Hardware-Plattform)
kommt nicht drauf an	Fertigungssteuerung, allgemeine Verwaltung	beliebige Software (auch Windows etc.), Vernetzung (incl. Internet)

Beispiele	Größenordnung der Zeitvorgabe t	zu erbringende Funktionen[1)]
serielle Hochgeschwindigkeits-Bussysteme (Fibre Channel, FDDI, Escon, Gigabit Ethernet, IEEE 1394)	Taktfrequenzen 250 MHz... > 1 GHz; t zwischen 4 und < 1 ns	Serialisierung und Deserialisierung[2)]
serien-parallele Hochgeschwindigkeits-Bussysteme (Direct Rambus, SLIO, AGP)	Taktfrequenzen bis ca. 500 MHz; t um 5...2 ns	Serialisierung und Deserialisierung, Steuerung der Zugriffsabläufe bzw. Zustandsübergänge[2)]
Befehlsablaufsteuerung in Prozessoren	Taktfrequenzen bis > 400 MHz; t bis zu 2 ns	komplizierte Zustandsübergänge, Datenverknüpfungen, Beobachten/Erkennen von Nebenläufigkeiten und Sonderbedingungen
Videosignale	zwischen ca. 5 (Fernsehen) und 250 MHz; t bis zu 4 ns	Serialisierung und D-A-Wandlung
parallele Bussysteme (Prozessor-Bussysteme, PCI)	typische Taktfrequenzen: 33, 66, 100 MHz; t zwischen 30 und 10 ns	Steuerung der Zugriffsabläufe bzw. Zustandsübergänge
Steuern und Regeln technischer Prozesse	ergibt sich gemäß Abtasttheorem; t typischerweise im ms-Bereich[3)]	anwendungsspezifisch; typischerweise hohe algorithmische Kompliziertheit
Mensch-Maschine-Interfaces (Bedientafeln, Tastaturen, Anzeigen, Bedienoberflächen)	ergibt sich aus psychologischen Erfahrungstatsachen (“was wird als befriedigend empfunden”) und aus physiologischen Grenzen (Informationswahrnehmung, Geschwindigkeit von Bedienhandlungen); t typischerweise zwischen 20 und 200 ms[3)]	anwendungsspezifisch; typischerweise mittlere...hohe algorithmische Kompliziertheit

Zuhandenheit

- Technologien,
- Beschreibungsmittel,
- Fachwissen (State of he Art), gedankliche Ansätze (Paradigmata), Problemverständnis,
- Verifizierungsmittel:
  - Testen,
  - Simulation,
  - Prüfen/Messen,
  - Fehlersuchen (Debugging).

Was man nicht prüfen kann (Funktionsnachweis), kann man eigentlich gar nicht realisieren.

Ebenen der Zuhandenheit:

- Basistechnologien (Si, GaAs usw.),
- Fertigungstechnologien (z. B. 0,35 µm CMOS),
- Bauelemente - fertige Schaltkreise, auch programmierbare:
  - Gatter-Ebene
  - Register-Transfer-Ebene
  - Funktionsblöcke (Building Blocks)
- Hardware-Plattformen,
- Entwicklungsumgebungen,
- Systemumgebungen.

Paradigmata:

grundlegende Lösungsansätze und Beschreibungsmittel	technische Lösungsansätze	Programmier-"Philosophien"
▪ Boolesche Gleichungen, ▪ Kontaktpläne (≙ herkömmliche Steuerungstechnik), ▪ Schaltpläne, ▪ Zustandsgraphen (State Machines, abstrakte Automaten), ▪ Entscheidungstabellen, ▪ Hardware-Beschreibungssprachen, ▪ Petri-Netze, ▪ analoge Signalverarbeitung (≙ Regelungstechnik, rückgekoppelte Systeme, Laplace-Transformation), ▪ digitale Signalverarbeitung (≙ z-Transformation), ▪ neuronale Netze, ▪ Fuzzy-Logik	▪ kombinatorische Zuordnung, ▪ Table Lookup, ▪ State Machines (abstrakte Automaten), ▪ Sequencer (einfache Folge-, Zeitplan- und Ablaufsteuerungen), ▪ Mikrocontroller, ▪ universelle Prozessoren, ▪ spezielle Prozessoren, ▪ Multiprozessorsysteme, ▪ hochparallele Systeme (auch: zelluläre und systolische)	▪ herkömmliche Programmierung (Programmablauf, Flowchart- bzw. if-then-goto-Paradigma) ▪ strukturierte Programmierung, ▪ ereignisgesteuerte Programmabläufe ▪ Multitasking, ▪ abstrakte Datentypen, ▪ objektorientierte Programmierung , ▪ relationale Ansätze (Datenbasis; Memory Based Reasoning), ▪ regelbasierte Ansätze (Expertensysteme, Prolog usw.)

Ungewöhnliche Kombinationen = Erfindung. (Es gibt keinen Universalalgorithmus des Erfindens, wohl aber ein Methodenwissen einschließlich einer gut gefüllten Trickkiste.)

Vorgehen: Durchmustern der Wissensbasis, Zusammenstellen geeigneter Kombinationen.

Auf welcher Ebene der Zuhandenheit aufsetzen?
Das hängt ab von der Realzeitraster-Vorgabe und ist deshalb fließendes Ziel in Abhängigkeit vom Stand der Technik:

- < 1 ns: Basis- (Halbleiter-) Technologie,
- wenige ns: Fertigungstechnologie (zuhandene Halbleiter-Technologien + Integrationsgrad + Kombinationen (Mixed Signal, Logik + DRAM usw.),
- < 1 µs: Bauelementetechnologie (zuhandene Schaltkreise),
- vom ms-Bereich an: zuhandene Funktionseinheiten...fertige Systemumgebungen

Kostenoptimierung = nicht zuviel Overkill für die Problemlösung = gerade soviel, um Reserven für Änderungen zu haben (den gesamten Lebenszyklus bedenken!).

Software oder Hardware?
Im strengen Sinne gibt es die Alternative nicht, da jede Software eine Hardwareplattform braucht, um laufen zu können.

Beides zusammen = Kostenoptimum unter Erfüllung der vorgegebenen Realzeitanforderungen.

Stand der Technik:

- Hohe Anforderungen an die Funktionalität.
- Programmier-Paradigma hat sich soweit durchgesetzt, daß praktisch jedem Lösungsansatz für auch nur halbwegs komplexe funktionelle Anforderungen irgendeine Form der Programmierbarkeit zugrunde gelegt wird - kaum noch jemand baut hochkomplizierte Spezialschaltungen, die nicht irgendwie programmierbar bzw. durch Programmierung steuerbar (parametrisierbar) sind.
- Es geht also in der Praxis nicht um den extremen Gegensatz: fertige Systemumgebung vs. Einzweck-Spezialschaltung - sondern um kostenoptimierte Verbundlösungen.
  - Ausnutzung zuhandener Prozessoren - Auswahl, Gestaltung der Systemumgebung, Zusatzbeschaltung, Implementierung vorteilhafter Programmier-Paradigmata (Multitasking, Ereignissteuerung, State Machine usw.), Mehrprozessorkonfigurationen.
  - Gestaltung neuartiger Prozessoren.
  - programmierbare, funktionsvariable Hardware.

Das Entscheidungsproblem (Hardware vs. Software) stellt sich in der Praxis:

- zur Erfüllung extremer Realzeitanforderungen,
- zur Erfüllung gegebener Anforderungen auf kostengünstigere Weise (Funktionalität implementieren, die bisher im gegebenen Kostenrahmen nicht implementierbar war),
- zum “entwicklungsmäßig schneller sein” (Time to Market).

Weshalb ersetzen wir Hardware durch Software?

- der herkömmliche Trend seit den 60er Jahren,
- Rückgriff auf ein höheres Niveau der Zuhandenheit,
- Zuhandenheit hochleistungsfähiger Technologien (Taktfrequenzen),
- Verkürzung der Entwicklungszeit (“es ist nur zu programmieren” = Time to Market),
- Verringerung der Hardwarekosten (Controller usw.),
- Beherrschung von Kompliziertheit (durch Zerlegen in handliche Programmstücke),
- Ersparen der Hardware-Entwicklung (zuhandene funktionstüchtige Plattformen),
- kurze Änderungszyklen,
- flexible Entwicklungswerkzeuge,
- funktionelle Flexibilität allgemein (man darf programmieren - es läßt sich alles programmieren),

- Software kann nicht ausfallen (verschleißen) -- wohl aber Fehler enthalten (und auch veralten -- wenn es keine Hardware-Plattform mehr gibt, auf der sie laufen könnte).

Weshalb ersetzen wir Software durch Hardware?

- ein (wieder mal) neuerer Trend,
- ermöglicht durch hochintegrierte programmierbare Schaltkreise (FPGAs, CPLDs),
- mehr Leistung,
- Kontrolle über den gesamten Lebenszyklus (nicht mehr auf Zulieferungen, fremde Standards und fremde Schutzrechte angewiesen sein),
- Kostenoptimierung = Verbilligung der Hardware-Plattform, kein Software-Overhead (besserer Wirkungsgrad),
- Lösung der Aufgabe mit weniger Silizium, Strom, EMV (langsamst-mögliche Taktierung),
- besseres Realzeitverhalten,
- kürzere Latenzzeiten,
- echter Parallelismus (von vornherein = gemäß Struktur des Problems - der dem Problem überhaupt inhärente Parallelismus könnte voll ausgenutzt werden),
- unmittelbare (evidente) Vergegenständlichung der Informationswandlungen (kein Zwang zum Programmieren (= unangemessen, unübersichtlich, “semantische Lücke” zwischen Problembeschreibung und Programmablauf, zuviel Overhead).

Herkömmliches (intuitives) Entwerfen in der klassischen Digitaltechnik

Beispiel: Serializer/Deserializer (SERDES)

1. sich die Aufgabenstellung klar machen

   1.1. Hauptaufgabe(n) definieren

   1.2. Voraussetzungen und Bedingungen fixieren, unter denen die Funktionen im Normalfall erbracht werden (diese bilden die Grundlage der entwurfsmäßigen Umsetzung er Normalfunktionen)

   1.3. Neben- und Sonderbedingungen (auch: mögliche Gotchas) erkennen

2. hauptsächliche Datenwege und funktionelle Hardware entwerfen (Register-Transfer-Niveau)

3. aus 2. ergibt sich die “zu steuernde” Hardware. Dafür Steuerung entwerfen (als State Machine(s)).

4. Sonderbedingungen betrachten. Überlegen, was zu tun ist, wenn die Voraussetzungen und Bedingungen nicht erfüllt sind:

   - gar nicht beachten? (weil extrem unwahrscheinlich; Fehlverhalten wird ohnehin an anderer Stelle erkannt)
   - erkennen und registrieren? (Zwecks gelegentlicher Abfrage)
   - erkennen und signalisieren? (Interrupt, Programmausnahme)

Grundsatzlösungen Serialisierung:

- Abfrageprinzip
- Schiebeprinzip

Grundsatzlösungen Deserialisierung:

- Verteilerprinzip
- Schiebeprinzip

Hauptprobleme:

- Übernehmen/Abgeben der parallelen Daten. Läuft auf ein Zählen der (seriellen) Bits hinaus
- Entkopplung des Übernehmens/Abholens von der seriellen Übertragung (Pufferung)
- Erkennen, wann die Deserialisierung beginnen soll

Programmierbare Logik

***Die* zeitgemäße Grundlage, Digitalschaltungen zu realisieren (auch in geringen Stückzahlen)**

GAL: Generic Array Logic - für einfach(st)e Schaltungen

CPLD: Complex Programmable Logic Device - für kompliziertere Kombinatorik, State Machines, Zähler usw.

FPGA: Field Programmable Gate Array - für sehr umfangreiche bzw. komplizierte Schaltungen (bis hin zu Graphikcontrollern, Spezialprozessoren usw.)

GAL:

- UND-ODER-Strukturen mit nachgeschalteten Makrozellen (Eingang - UND - ODER - Makrozelle - Ausgang)
- “Industriestandards” für diverse Ausführungen (Second Source)
- nur wenige Typen “in System” programmierbar
- Entwurfserfassung über Boolesche Gleichungen usw. (Sprachen: CUPL, PALASM, ABEL usw.)

CPLD:

- UND-ODER-Strukturen mit nachgeschalteten Makrozellen
- mehr Makrozellen als GAL
- interne Verbindungsnetzwerke (um Makrozellen zusammenschalten zu können)
- Laufzeiten gut vorhersagbar
- sehr herstellerspezifisch
- moderne Typen “in System” programmierbar
- auf den ersten Blick ähnliche Architekturen, aber Unterschiede in den Einzelheiten - die bisweilen wirklich stören (Entwurf paßt in CPLD des Herstellers A, aber nicht in jene des Herstellers B)
- Entwurfserfassung über Schaltplan (Schematic Entry), Boolesche Gleichungen, State-Machine-Beschreibungen, Hardwarebeschreibungssprachen (Verilog, VHDL)

FPGA:

- mehr oder weniger elementare, universelle Zellen in einer Matrixanordnung von Verbindungswegen
- aus den Zellen läßt sich “alles” zusammenschalten (Silizium ist flexibler nutzbar als bei CPLD)
- sehr herstellerspezifisch (beachtliche Unterschiede in der Architektur)
- Laufzeiten schwierig vorhersagbar (Entwurf langwieriger als mit CPLD)
- FPGAs können umfangreiche und komplizierte Schaltungen aufnahmen (die nicht unbedingt UND-ODER-Flipflop-Strukturen sein müssen)
- Entwurfserfassung: wie CPLD. Entwicklungsumgebungen teuer. Einarbeitung langwierig

Programmierverfahren	Ausführung	Änderbarkeit	Bemerkungen
Maskenprogrammierung	beim Halbleiterhersteller	im einzelnen Schaltkreis nicht mehr änderbar	nur bei extremen Stückzahlen von praktischer Bedeutung
Durchschmelzprinzip (Fuse)	beim Anwender*)	im einzelnen Schaltkreis praktisch nicht mehr änderbar	alle Verbindungen sind vorgefertigt, die nicht benötigten werden beim Programmieren getrennt
Aufschmelzprinzip (Antifuse)	beim Anwender*)	im einzelnen Schaltkreis praktisch nicht mehr änderbar	alle Verbindungsstellen sind zunächst getrennt, benötigte Verbindungen werden beim Programmieren hergestellt
Ladungsspeicherung mit UV-Löschung	beim Anwender*)	durch Löschen und Neuprogrammieren	Löschen durch UV-Licht; erfordert Quarzglasfenster im Schaltkreis (es gibt auch preisgünstige Ausführungen ohne Fenster; diese kann man nicht mehr löschen)**)
Ladungsspeicherung mit elektrischer Löschung (EEPROM, Flash)	beim Anwender*)	durch Löschen und Neuprogrammieren (auch: in der Anwendungsschaltung (In System Programming)	Löschen durch elektrische Impulse
RAM-Zellen	beim Anwender*) bzw. während des Betriebs	durch Umladen; auch während des normalen Betriebs beliebig oft möglich	Halten der Information in Flipflops bzw. Latches; nach jedem Einschalten ist erneutes Laden erforderlich

*): Anwender = Hersteller des Gerätes bzw. der Funktionseinheit; **) Bezeichnung: OTP (One Time Programmable)

Reguläre Steuerwerke (Einführung)

Möglichkeiten zum Realisieren der geforderten Informationswandlungen:

1. durch unmittelbare Vergegenständlichung,
2. durch Folgen elementarer Informationswandlungen.

zu 1. Schaltung folgt Funktion. Entwurfsmethodik: Funktionsblöcke + Datenwege + Steuerung.

Detaillierung: Register-Transfer-Niveau (RTL) + Steuerung:

RAM- und ROM-Arrays + taktgesteuerte Speicherglieder (Register) + Kombinatorik + Datenwege + Steuerung

zu 2. Schaltung (Hardware) = Plattform der Programmierung. Muß mit eigentlicher Funktion gar nichts zu tun haben. Anforderungen an Leistungsfähigkeit (in MIPS und MBytes usw.) steigen, je weiter funktionelle Anforderungen und Möglichkeiten der Plattform auseinanderliegen (semantische Lücke).

Von der Hard- zur Softwarelösung

- kombinatorische Zuordnung
- Datenflußprinzip
- Ablaufsteuerung (Sequencing)
- State Machines
- Mikroprogrammsteuerung
- herkömmlicher Universalrechner (Mikrocontroller oder -prozessor)

Kombinatorische Zuordnung
Aus den Eingangsdaten (Argumenten) werden die Ausgangsdaten unmittelbar gebildet, und zwar mit kombinatorischen Netzwerken. Diese werden durch Boolesche Gleichungen beschreiben und sind zustandsfrei.

Alternative: Zuordnung über ROM oder RAM (gespeicherte Wertetabellen)

a) Speicher wird über Eingangsdaten adressiert,
b) Speicher wird mit Eingangsdaten nach Lösung durchsucht
   1) Assoziativprinzip
   2) sequentielle Durchmusterung eines adressierbaren Speichers

Grenzen der Realisierbarkeit sind gegeben durch:

1. die Anzahl der Argument- und Resultatbits,
2. die Kompliziertheit der Verknüpfungen.

- bei überschaubarer Kompliziertheit (z. B. Addieren, Verschieben, Durchschalten (Routing)) > 100 Bits (= Ein- und Ausgangsleitungen). Z. B. Addierwerk mit 129 Eingängen (2 · 64+ Carry Inject) und 67 Ausgängen (64-Bit-Ergebnis + ZF + CF + OF).
- bei beliebiger Kompliziertheit um 24 Eingänge bei ca. 64 Ausgängen (Zuordnung durch Adressierung eines Speichers (ROM oder RAM) von 16M · 64 Bits (128 MBytes), obere sinnvolle Grenze der Durchführbarkeit bei max. 32 Eingängen (4G Einträge).

Wenn es nicht klappt...

1. Ausweg: abschnittsweise Verknüpfung
Eine solche Anordnung arbeitet taktgesteuert, wobei in jedem Takt aus den jeweils ausgewählten Argument-Abschnitten die jeweiligen Ergebnis-Abschnitte gebildet werden. Die höchste überhaupt mögliche Verarbeitungsleistung einer solchen Anordnung wird dann erreicht, wenn in jedem Maschinentakt ein Abschnitt des Ergebnisses gebildet werden kann, dessen Bitanzahl der Verarbeitungsbreite der Anordnung entspricht bzw. wenn es ausreicht, zur Bildung des Ergebnisses auf jeden Argument-Abschnitt genau einmal zuzugreifen. Ob dies überhaupt gelingen kann, ist letzten Endes eine Eigenschaft des Algorithmus.

1. Beispiel (wo es klappt): die Bildung des Skalarprodukts zweier Vektoren.

2. Beispiel (wo es im allgemeinen Fall nicht klappt): Sortierabläufe. Hierbei ist es praktisch nicht zu vermeiden, auf die einzelnen Argument-Abschnitte mehrmals zuzugreifen.

Die Implementierungseffizienz
Diese Eigenschaft des Algorithmus läßt sich durch eine Zahlenangabe ausdrücken, die wir als *Implementierungseffizienz* e_i bezeichnen wollen.

$$e_i = \frac{\sum_{i=1}^{n} CARDB(A_i) + \sum_{j=1}^{m} CARDB(R_j)}{n \cdot (ARG\text{-}LINES + RES\text{-}LINES)}$$

Zur Bedeutung der Symbole:

- CARDB (A_i), CARDB (R_j): Anzahl der Bits, mit denen die Argumente und Resultate repräsentiert (bzw. codiert) werden,
- ARG-LINES, RES-LINES: Anzahl der Signalleitungen, die zum Transportieren der Argument- und Resultatabschnitte zur Verfügung stehen (Stichwort: Verarbeitungsbreite),
- z: Anzahl der Maschinenzustände (Taktzyklen), die benötigt werden, um einen Resultat-Abschnitt zu bilden ($z \geq 1$).

Hinweis: Ist eine technisch gegebene Leitungszahl größer als die Anzahl der Bits des betreffenden Abschnittes, ist die Abschnittslänge anstelle der Leitungszahl einzusetzen.

Die Implementierungseffizienz e_i ist somit eine dimensionslose Zahl im Intervall $0 < e_i \leq 1$. e_i ist gleich 1, wenn die beschriebene zweckmäßigste Implementierung tatsächlich möglich ist, d. h., wenn es gelingt, eine Anordnung zu bauen, die in jedem Taktzyklus entsprechend ihrer Verarbeitungsbreite zum gewünschten Endergebnis beiträgt. Dies kann nur dann erreicht werden, wenn für jeden Abschnitt des Resultats gilt, daß dieser durch kombinatorische Zuordnung aus den jeweils ausgewählten Abschnitten der Argumente gebildet werden kann, wobei in diese Zuordnung höchstens noch Zustands-Angaben einbezogen werden, die eindeutig aus den zuvor verarbeiteten Abschnitten bestimmt worden sind.

Es ist durchaus möglich, diesen Ansatz bis in die Feinheiten hinein zu formalisieren. Um zu Beginn der Bearbeitung eines Projekts Entscheidungen zu treffen, reicht aber typischerweise eine “heuristische” Anwendung aus. Hierbei ist zu untersuchen, ob die Problemstellung auf Algorithmen führt, die eine Implementierungseffizienz von 1 (oder nahezu 1) haben oder ob diese ohnehin deutlich geringer ausfällt.

Praxistip: für die leistungsbestimmenden Informationswandlungen Probeentwürfe ausführen. Ein “gröberes” Register-Transfer-Niveau reicht vollkommen aus (erforderliche Breite der Signalwege, Abschätzung der Aufwendungen für die kombinatorischen Zuordnungs-Netzwerke usw.). Es wird sich recht schnell zeigen, ob so etwas überhaupt durchführbar ist oder nicht (Aufwand zu hoch, Steuerung zu kompliziert, Ergebnisse lassen sich grundsätzlich nicht auf gewünschte Weise bilden (d. h. mit nur einmaligem Zugriff auf jede Informationsstruktur).

Je näher e_i an Eins liegt, um so eher lohnt es sich, an eine Hardwarelösung zu denken (auch in dem Sinne, eine gegebene Prozessorstruktur durch Zusatzbeschaltung zu erweitern).

Bei an sich kleinem e_i ist es typischerweise besser, Hardware-Entwicklungen zu vermeiden. Stattdessen die jeweils leistungsfähigste Software-Plattform aussuchen.

Die Implementierungseffizienz ist < 1, wenn

1. die abschnittsweise Zuordnung technisch nicht verwirklicht werden kann (Aufwand, Kompliziertheit), so daß Folgen mehrerer Taktzyklen (Maschinenzustände) nötig sind, um jeweils einen Resultatabschnitt zu bilden,
2. Resultatbits von Argumentbits abhängen, die sich in verschiedenen Abschnitten befinden können, so daß Zugriffe auf mehrere Argumentabschnitte notwendig sind, um diese Resultatbits zu bestimmen,
3. gewisse Argumentbelegungen zur Folge haben, daß Teile bereits gebildeter Resultatabschnitte geändert werden müssen (dies erfordert, erneut auf diese Abschnitte zuzugreifen) bzw. daß man das Ergebnis gar nicht in abschnittsweise liefern kann, sondern daß zunächst Zwischenergebnisse zu bilden sind.

Ein Sachverhalt nach Punkt 1 ist nicht immer ein unüberwindliches Hindernis: es ist letztlich eine Ermessensfrage, was man als “zu aufwendig” oder “zu kompliziert” ansieht. Hingegen bezeichnen die Punkte 2 und 3 objektive Grenzen. Solche Algorithmen können nie mit e_i =1 implementiert werden (auch dann nicht, wenn der Aufwand keine Rolle spielt). Ein plausibles Beispiel ist das Umordnen von Bits in Bitfeldern (z. B. von Pixeln in einem Bildspeicher) gemäß den Angaben einer Indexliste. Jede Argumentposition kann hierbei grundsätzlich in jede Resultatposition transportiert werden, so daß es notwendig sein kann, bereits gebildete Resultatabschnitte nochmals aufzurufen, um neue Werte einzufügen.

Wenn die abschnittsweise Zuordnung nicht zu verwirklichen ist...

- entweder aus Aufwandsgründen oder als Folge der inhärenten Eigenschaften des Algorithmus -

dann muß die Verarbeitungsaufgabe auf Folgen einfacherer Verknüpfungen zurückgeführt werden:

1. Stufe: zeitliche Folge der Operationen liegt fest: Sammlung von Operationswerken, die durch zeitstarre Steuerung ausgewählt werden. Sequencer als Steuerung),

2. Stufe: Zwischenergebnisse müssen in nachfolgenden Operationen einfließen: (1) gemeinsamer Speicher, (2) in seinen Operationen umschaltbares Operationswerk, (3) Operationswerksausgang auf Speicher zurückgeführt. Einfügen von Zwischenregistern notwendig (Übergang zur Registermaschine).

3. Stufe: Steuerung nicht mehr zeitstarr, sondern von Bedingungssignalen abhängig (die über einen Zuordner einfließen: einfache State Machine),

4. Stufe: Zustandsübergänge zu kompliziert oder Steuerung nicht flexibel genug: Steuerspeicher (= Mikroprogrammsteuerung) einführen (löst komplizierte Zustandsübergänge in Folgen einfacherer auf und ist leicht änderbar),

5. Stufe: Steuerspeicher wird mit Datenspeicher vereinigt. An sich Aufwandsverringerung, aber: Zugriffe auf Daten und Steuerangaben nur nacheinander möglich (weitere Serialisierung). Weitere Zwischenregister erforderlich.

6. Stufe: auch Speicheradressierung läuft über Operationswerk. Hiermit Übergang zum klassischen v.Neumann-Rechner vollzogen.

Von Stufe zu Stufe sinkt (typischerweise) der Aufwand und es steigt die Vielseitigkeit (Universalität). Es wächst aber auch der Zeitbedarf, da alle Informationswandlungen auf Folgen elementarer Abläufe zurückgeführt werden müssen.

State Machines

Selbstverständlich kann man Steuerschaltungen sozusagen intuitiv aufbauen, also Gatter, Flipflops usw. so verschalten, wie es gerade zweckmäßig erscheint. Solche Schaltungen sind aber nicht sehr übersichtlich und sie werden typischerweise, von einfachsten Formen abgesehen, kaum auf Anhieb perfekt funktionieren Reguläre, übersichtliche Schaltungsprinzipien sind deshalb sehr wünschenswert. Die direkte Anwendung automatentheoretischer Überlegungen bietet eine Lösung. Man geht von den funktionell notwendigen *Maschinenzuständen* aus, ordnet jedem Zustand eine bestimmte Belegung der Speichermittel (Flipflops) zu (Zustandscodierung) und gestaltet die kombinatorischen Verknüpfungen so, daß sie zu jedem Taktzeitpunkt aus aktuellem Zustand und aktueller Eingangsbelegung den Folgezustand und die geforderte Ausgangsbelegung bestimmen (Funktionszuordner).

Steuerschaltungen erzeugen bedingungsabhängige Ausgangssignale, im allgemeinen Fall abhängig (1) von den Eingangssignalen und (2) vom aktuellen Zustand (State). Die State Machine ist technische Verwirklichung des Konzepts vom abstrakten Automaten.

- $E = \{e_1, e_2, ...\}$ Eingangsbelegung,
- $A = \{a_1, a_2, ...\}$ Ausgangsbelegung,
- $Z = \{z_1, z_2, ...\}$ Zustände.

Mealy-Automat:

Überführungsfunktion

$z (t+1) = f (z(t), e(t))$

Ausgabefunktion (bei vollsynchroner Ausführung)

$a (t+1) = g (z(t), e(t))$

(Die Ausgangsbelegung kann aus aktuellem Zustand und Eingangsbelegung kombinatorisch gebildet werden. Sie steht im nächsten Takt zur Verfügung.)

Moore-Automat:

Überführungsfunktion

$z (t+1) = f (z(t), e(t))$

Ausgabefunktion

$a (t+1) = g (z(t))$

(Ausgangsänderung erfordert erst Zustandsänderung. Somit steht sie bei vollsynchroner Arbeitsweise erst im übernächsten Takt zur Verfügung (= 1 Takt später als beim Mealy-Automat).)

Jeder Mealy-Automat kann in einen Moore-Automaten mit gleichem Verhalten umgewandelt werden (von der Verzögerung der Ausgangserregung um 1 Takt abgesehen).

Semiautomat (Medwedjew-Automat):

Keine Ausgabefunktion vorhanden (es interessieren nur die Zustände):

Überführungsfunktion

z (t+1) = f (z(t), e(t))

Eine Tendenz im modernen Hardware-Entwurf besteht darin, Steuerschaltungen in überschaubare State Machines zu zerlegen und diese über Zustandsgraphen und die Schaltgleichungen der Funktionszuordner zu dokumentieren.

Die Funktionszuordner selbst werden oft mit programmierbaren Schaltkreisen realisiert. Wenn es nicht auf die letzte ns ankommt, kann man die Funktionszuordner mit (PROM-) Speicherschaltkreisen aufbauen. Die Vorteile: unbeschränkte Komplexität (auch komplizierteste Zustandsübergänge kosten keinen Mehraufwand), größte Einfachheit, leichte Änderbarkeit. Passende PROM-Register-Anordnungen gibt es auch als fertige Schaltkreise (State Machine PROMs).

Zustandscodierung

Wenn man n Maschinenzustände (States) vorsehen muß - wieviele Flipflops werden dazu benötigt und wie ordnet man diese den einzelnen Zuständen zu? - Dies ist das Problem der Zustandscodierung. Grundsätzlich gibt es eine unabsehbare Vielfalt von Codierungsmöglichkeiten (auch sehr trickreiche), aber nur wenige haben in der Praxis weite Verbreitung gefunden:

Binärcodierung. n Zustände werden mit ceil (ld(n)) Flipflops binär codiert. Dies ergibt die geringste Flipflopanzahl, erfordert aber oft einen komplexen Funktionszuordner (viele Gatter, hohe Schaltungstiefe).

1-aus-n-Codierung. Ganz einfach: jeder Zustand wird durch einen Flipflop repräsentiert. Man braucht n Flipflops für n Zustände, wobei jeweils nur ein Flipflop aktiviert ist (im Fachenglisch spricht man auch vom *One-Hot Encoding*, OHE). Das erfordert vergleichsweise viele Flipflops. Dafür wird die Zuordner-Kombinatorik einfach und schnell.

Gemischte Codierung. Manchmal sind bestimmte Codierungen in der Schaltungspraxis einfach nicht anwendbar. Einen 20-Bit-Zähler (über eine Million Zustände!) wird man kaum mit 1-aus-n-Codierung verwirklichen. So liegt es nahe, Zustände, die durch Zählprozesse aufeinanderfolgen, binär zu codieren und solche, zwischen denen kompliziertere Übergänge vorgesehen sind, im 1-aus-n-Code. Ein weiteres Beispiel ist die Abwandlung des 1-aus-n-Codes dahin, daß der "Ruhezustand" (Idle State) nicht durch ein gesondertes Flipflop, sondern durch die Nullbelegung aller verbleibenden Flipflops codiert wird.

Auswahl
Ein guter Schaltungsentwickler strebt minimale Kosten im Rahmen der gegebenen Technologie an. "Akademische" Minimalkriterien (etwa aus der Schaltalgebra und Automatentheorie) sind dabei nur bedingt hilfreich. So ist es ein wichtiger Gesichtspunkt, ob man die State Machine noch in einem bestimmten CPLD, FPGA oder Gate Array unterbringt. Moderne Technologien legen es nahe, die 1-aus-n-Codierung zu bevorzugen: an Flipflops herrscht kaum Mangel, aber komplizierte Verknüpfungsnetzwerke fressen Siliziumfläche bzw. sind (im einzelnen programmierbaren Schaltkreis) gar nicht realisierbar. Hingegen ist bei PROM-Realisierung die binäre (minimale) Zustandscodierung praktisch unbedingt erforderlich, um die Anforderungen an die Speicherkapazität zu reduzieren (ein Bit weniger, und wir kommen mit einem halb so großen PROM aus!). Die Kompliziertheit der kombinatorischen Verknüpfungen spielt hingegen beim PROM keine Rolle.

Faustregeln:

- diskreter Entwurf (MSI): je nach Zustandszahl, Kompliziertheit der Zustandsübergänge und geforderten Ausgangssignalen binäre oder OHE-Codierung oder Tricklösung,
- ROM- oder RAM-Zuordner: unbedingt binär. Jedes eingesparte Zustandssignal verringert die Anforderungen an die Speicherkapazität auf die Hälfte!
- CPLDs: ähnlich MSI. CPLD-Architektur (Zellen, AND-Arrays, globale Verbindungs-Ressourcen) ansehen und überlegen, was am besten passen könnte. Umfangreiche AND-OR-Strukturen sprechen eher für binäre Codierung. Es kann aber sein, daß dies die Verbindungsressourcen zustopft. Anderer Ansatz: vorhandene Flipflops ausnutzen (wenn OHE ‘reinpaßt, dann ja).
- FPGAs und Gate Arrays: OHE bevorzugen. Umfangreiche UND-ODER-Knoten fressen Zellen. 1 Zelle ist typischerweise eine einfache Kombinatorik oder ein Flipflop oder eine Kombination von beidem. Wichtig: minimale Zellenzahl in den Grenzen der ausführbaren Verbindungen (Routing) zwischen den Zellen, nicht aber minimale Flipflop- oder Gatterzahl.

Einfachste Sequencer

Sequencer sind State Machines mit besonders einfachen Zustandsübergängen. Im einfachsten Fall: taktweises Weiterschalten von Zustand zu Zustand (erzeugt zeitstarre Signalfolgen).

- Umlauf-Schieberegister
- Zähler mit Decodierung
- adressierbarer Speicher mit Adreßzähler
- FIFO unter Ausnutzung der Retransmit-Funktion

Einfache State Machines

Diese sind durch charakteristische Formen von Zustandsübergängen gekennzeichnet. Sie können somit von bewährten Grundschaltungen ausgehend in Anpassung an die konkreten Bedingungen entworfen werden.

Charakteristische Zustandsübergänge:

- Single Shot
- Warten - Weiter - Woandershin
- Akzeptorautomaten

State Machines mit Zuordnern

Diese entsprechen gleichsam bilderbuchmäßigen Realisierungen abstrakter Automaten. Zuordner mit ROM oder RAM oder CPLD. Es gibt eigens ROMs und CPLDs (auch: GALs) für solche Anwendungen.

RAM-Zuordner erfordern ein Laden nach dem Einschalten. Lösungen:

- Register vor den Adreßeingängen umschaltbar ausbilden (als Zähler oder als Schieberegister),
- Schalterbauelemente (CBT, QuickSwitch usw.) zwischenschalten.

Steuerung des Ladeablaufs: entweder von einem kleinen Mikrocontroller oder über die Plattform, in der die Schaltung eingesetzt ist (z. B. vom PC aus).

Programmierbare Steuerungen (Einführung)

Eine programmierbare Steuerung erbringt die geforderten Zustandsänderungen nicht durch direkte Zuordnung, sondern durch Folgen elementarer Informationswandlungen, Abfragen und Verzweigungen.

Die Steuerangaben sind in Form von Befehlen in einem Steuerspeicher (Control Storage) untergebracht. Eine solche Einrichtung muß kein Universalrechner sein. (Wir sprechen deshalb hier auch von Mikrobefehlen, Mikroprogrammierung, vom Mikroprogrammspeicher usw.)

Praktische Bedeutung:

- Alternative zu Mikrocontrollern in programmierbaren Schaltkreisen und ASICs (Kostensenkung durch feinstufigere Anpassung an das zu lösende Problem),
- Steuerung von Spezialprozessoren,
- reguläre Steuerungsprinzipien für alle Zwecke, wo es auf Geschwindigkeit ankommt, die Zustandsübergänge aber so kompliziert sind, daß sie sich nicht über direkte Zuordnung verwirklichen lassen.
- virtuelle Peripherie. Programmierbare statt Einzweckhardware.

Grundstruktur
Die Steueranordnung (Mikroprogrammsteuerwerk) besteht aus einem Mikroprogrammspeicher, einem Mikrobefehlsdatenregister, einem Mikrobefehlsadreßregister sowie Schaltmitteln zur Ablaufsteuerung, Operandenspeicherung und - verknüpfung und Koppelstufen.

Mikrobefehlsadressierung
Um Mikrobefehle zu adressieren, gibt es 2 Möglichkeiten:

1. Adreßzählung. Hierfür ist das Mikrobefehlsadreßregister als Zähler ausgebildet. In einfachster Ausführung ergibt sich so ein Sequencer, der den Mikroprogrammspeicher fortlaufend adressiert.

2. Folgeadresse im Mikrobefehl. Befehl um Adreßteil verlängert. Erspart ggf. Mikrobefehlsadreßregister. Nacheinander auszuführende Mikrobefehle müssen im Speicher nicht hintereinander stehen.

Verzweigung
Abläufe, die über ein einfaches “Sequencing” hinausgehen, erfordern, die Mikrobefehlsadressierung in Abhängigkeit von bestimmten Bedingungen ändern zu können. Alternativen:

Adreßzählung + Verzweigungsmikrobefehle. Letztere enthalten eine Verzweigungsadresse sowie ein Feld zur Bedingungsauswahl. Ist die ausgewählte Bedingung erfüllt, so wird die Verzweigungsadresse in das Mikrobefehlsadreßregister geladen, andernfalls wird die Adresse um 1 erhöht (Adreßzählung).

Alternative Folgeadressen im Mikrobefehl. Der Mikrobefehl enthält die Folgeadressen für beide Verzweigungsrichtungen sowie ein Feld zur Bedingungsauswahl.

Alternative der Bedingungsauswahl: es gibt Auswahlmikrobefehle, die je nach Bedingung einen Bedingungscode (bzw. Flagbit) setzen. Die Verzweigung findet dann nur statt, wenn der betreffende Mikrobefehl einen gesetzten Bedingungscode (Flagbit) vorfindet.

Überspringen (SKIP) + unbedingte Verzweigung. Die Bedingungsauswahl wirkt lediglich auf die Adreßzählung. Ist die Bedingung z. B. nicht erfüllt, wird der nächste Mikrobefehl adressiert (Adreßzählung um 1), ist sie erfüllt, der übernächste (Adreßzählung um 2). Um bei nicht erfüllter Bedingung zu verzweigen, kann man auf die Folgeadresse (+ 1) einen Mikrobefehl zur unbedingten Verzweigung setzen.

Funktionsverzweigung
Solche Mikrobefehle bewirken, daß bestimmte Signale direkt in Bits der Mikrobefehlsadresse einfließen. 1 Signal bewirkt eine Verzweigung in 2 Richtungen, 2 Signale in 4 Richtungen usw. So könnte man z. B. ganze Bytes von einem Interface in die Mikrobefehlsadresse einspeisen, um eine Verzweigung gemäß dem anliegenden Kommandocode zu bewirken.
Speist man die Bedingungen in die niedrigsten Adreßbitpositionen ein, so erhält man Blöcke von Mikrobefehlen, in denen jeder einer anderen Bedingung entspricht (dort bringt man typischerweise je eine unbedingte Verzweigung zum jeweiligen Behandlungsablauf unter).

Spätverzweigung
Mehrere Folgemikrobefehle werden parallel adressiert. Während der Ausführung des aktuellen Mikrobefehls stehen somit alle seine möglichen Nachfolger bereit. Der Mikrobefehl hat Auswahlfelder, über die bestimmt werden kann (und zwar typischerweise mittels Funktionsverzweigung) , welcher der Nachfolger tatsächlich wirksam wird. Erlaubt lückenloses Aneinanderreihen von Verarbeitungs-Mikrobefehlen. Praxisübliche Zahl an Verzweigungsrichtungen: 2, 4, 8. Programmierung ist richtiggehender Denksport (Produktivität um 100 Mikrobefehle/Mannjahr) bzw. erfordert spezielle Unterstützungssoftware.

Unterprogramme
Die Unterprogrammtechnik erfordert Vorkehrungen zur Adreßrettung. Alternativen:

- keine Unterprogramme. Stattdessen Inline-Code. Programmiererleichterung durch Makros. Werden Unterprogramme nur selten gerufen, so rechtfertigt dies den Aufwand kaum. Programmspeicher ist demgegenüber oftmals kostengünstiger.
- nicht die Adresse retten, sondern einen Rückkehrcode hinterlassen. Rückkehr durch programmierte Verzweigung.
- explizite Rettungsregister
- Hardwarestack
- Stack in Datenspeicher (mit Stackpointer usw.).

Problem einfacher Hardwarestacks: wir kommen nicht an die gerettete Adresse heran. Ausweg: explizite (und programmseitig zugängliche) Rettungsregister. Entweder dem Hardwarestack vorgeschaltet oder weiteres Retten in programmseitig verwalteten Stack.

Interrupts
Interrupts erfordern Vorkehrung zur Interrupteinleitung, zur Adreßrettung und ggf. zur Maskierung. Alternativen:

- keine Interrupts, stattdessen Rücksetzen der Hardware (= unbedingte Festadressenaufschaltung) und programmseitige Verzweigung (ggf. auf Grundlage hinterlegter Bedingungscodes)
- keine Interrupts, stattdessen Abfrageprinzip (Polling)
- Interrupts mit den diversen Alternativen der Adreßrettung (vgl. Unterprogramme)

Interrupt-Adressierung:

- nur 1 Festadresse (weiter durch programmierte Verzweigung),
- mehrere Festadressen je nach auslösender Ursache,
- Interrupttabelle

Interruptadressierung durch Adreßumschaltung (Break-In-Prinzip)
Für jede Interrupt-Ursache wird 1 Adreßregister vorgesehen (das ggf. durch Flagbits usw. ergänzt sein kann). Interrupt und Rückkehr bestehen dann einfach im Auswählen des jeweiligen Adreßregisters zur Adressierung des Mikroprogrammspeichers. Schneller geht’s nicht!

Wartemikrobefehle
Als Alternative zur Warteschleife: Wartebedingung im Mikrobefehl. Mikrobefehl bleibt so lange aktiv, bis Wartebedingung erfüllt ist.

Weiterbildung: Format für Warten - Weiter - Woandershin (ach als Funktionsverzweigung mit Bedingungssignalen)

Zeitzählung
Hochauflösender (taktgenauer) Maschinenzeitzähler und Wartemikrobefehl, um auf das Ablaufen eines Zeitintervalls zu warten.

Verarbeitungsaufgaben

1. nur Abfragen und Bits setzen
2. Vergleichen gemäß verschiedenen Bedingungen, Datenflüsse steuern, Direktwerte ausgeben
3. arithmetische und logische Operationen Direktwert gegen Register bzw. Lokalspeicher
4. arithmetische und logische Operationen mit Register- bzw. Lokalspeicherinhalten
5. Sonderverknüpfungen (= spezielle Verarbeitungshardware)

Mikrobefehlsformate

a) horizontal
für alles, was in einem Zyklus zu steuern sein könnte, die erforderlichen Steuerbits bzw. Steuerfelder, Direktwerte, Adressen usw. aneinanderreihen. Führt auf bisweilen extrem lange, aber wirkungsvolle Mikrobefehle.

b) vertikal
kurze Formate, je Format im Grenzfall nur 1 Wirkung (Abfrage, Verarbeitung. Verzweigung usw.). Führt auf Formate, die den Maschinenbefehlen einfacher Mikrocontroller ähneln.

c) Kompromißlösungen
wenige Formate mittlerer Länge. Trennung z. B. zwischen Verarbeitung, Interfacesteuerung und Verzweigung.

Praxistips:

1. Voll horizontal lohnt sich nur, wenn tatsächlich alle Wirkungen im selben Maschinenzyklus erbracht werden können. Ansonsten das Leistungsvermögen der zugrunde liegenden Hardware als Kriterium nehmen: 1 Befehlsformat = 1 (möglichst brauchbare) Sammlung von Steuerwirkungen, die die Hardware jeweils in einem Maschinenzyklus erbringen kann.

2. Klotzen statt Kleckern (H. Guderian): extreme Sparlösungen lohnen sich heutzutage kaum noch. Mikrobefehlsgestaltung sollte nicht zur übermäßigen Trickprogrammierung zwingen*), ansonsten könnte man gleich einen der marktgängigen Controller nehmen.

*): es ist u. a. sehr ungünstig, wenn man von einem Mikrobefehl aus nicht den gesamten Speicher- oder Registeradreßraum erreicht (wir zielen auf ein üblichen Controllern überlegenes Leistungsvermögen - da darf es auf ein paar Bits nicht ankommen). Die Wirtschaftlichkeit ergibt sich vielmehr aus der Anpassung an das Problem (kein Overkill, wie z. B. ungenutzte Verarbeitungsbefehle, ungenutzte Ports usw.)

Weitere Erfordernisse: Hardware-Multitasking, eingebaute Debugging-Vorkehrungen.

Problemlösung mit MIkrocontrollern

Mikrocontroller sind programmierbare Universalrechner, die zum Einsatz in Embedded Systems optimiert sind. Kennzeichnende Merkmale:

- Implementierung der jeweiligen Informationswandlungen auf Grundlage herkömmlicher Programmmiermodelle,
- Kostenoptimierung,
- Einbau in eine Anwendungsumgebung,
- Programmierung auf den jeweiligen Anwendungszweck hin (Controller wird durch Programmierung zur Einzweckmaschine),
- Programme typischerweise vom Nutzer nicht änderbar,
- wir dürfen programmieren (Narrenfreiheit, potentielle Funktionsvielfalt),
- wir müssen programmieren (Zwang zur Rückführung der anwendungsseitig geforderten Informationswandlungen auf elementare Programmschritte; Serialisierung der Informationsverarbeitung, Problemlösung entspricht nicht mehr dem anwendungsseitigen Datenflußschema).

Anwendungsfälle:

- Realzeitraster im ms-Bereich oder langsamer,
- es stehen keine anderweitigen Forderungen entgegen (extremes Stromsparen, EMV, Sicherheitsvorschriften).

Mikrocontroller sind meist deutlich kostengünstiger als CPLDs und einfacher zu programmieren.

Infrastruktur:

- Stromversorgung,
- Takt,
- Rücksetzen,
- E-A-Anschaltung,
- ggf. Speicheranschaltung.

Mikrocontroller + Hardware Schwerpunkte

Takt, Rücksetzen
jeweils gemäß Datenblatt/Applikationshandbuch vorgehen.
Achtung: es kann sein, daß andere Schaltungsteile kritischere Anforderungen (Takttoleranzen, Dauer des Rücksetzens und Rücksetz-Schwelwerte (V_{CC}) haben!

E-A-Ankopplung
- Strombelastung
- kapazitive Last
- Derating beachten
- Treibfähigkeit beachten

Richtungssteuerung
Vorsicht vor Buskonflikten beim Einschalten und im Betrieb!

Hochohmige (floating) Busleitungen
Vermeiden. (Pull-up-Widerstände, Bushalteschaltungen, Park-Hardware.)

Port-Belegung

- Im Problemfall Ports stets für die höher integrierte Funktion im µC ausnutzen, dafür die einfachen I/O-Funktionen auslagern.

ungenutzte Ports:

- als Ausgänge und mit Festwert belegen (nur, wenn garantiert unbeschaltet)
- als Eingänge mit externem Pull-up
- als Eingänge und internen Pull-up's scharf machen (oder lassen)
- NIE als wirklich offene Eingänge (falls nicht ausdrücklich im Datenblatt spezifiziert)

Bus-Erweiterung mit Pufferstufen und Registern

- Buskoppelstufen und Register
- Abfrage: Multiplexer
- Universal-Busschaltkreise ausnutzen
- Boundary Scan
- einfache Schiebe-Interfaces

Auf *asynchrone Eingänge* achten (Metastabilität). Ggf. über D-FF-Register synchronisieren.

Taktierung bzw. Output Enable von außen - nur selten vorhanden ("echte" Tristate-Ports, um µC seinerseits als Slave an einen Bus zu hängen)

Bei Anschaltung von Bustreibern *Buskonflikte* vermeiden (Richtungssteuerung).

Speicheranschaltung

dafür vorgesehene Prozessoren

- ausgebauter Speicherbus (H8/300H)
- Multiplex-Speicherbus (PIC, 8051 usw.)

sonst: Ports für Anschaltung ausnutzen. Es gibt auch spezielle kleine RAMs (z. B. von Siemens).

Adreßdecodierung
Alias-Adressen: nie für Programmiertricks ausnutzen!

Adreßerweiterung
Bankregister (oder Segment-RAM) und Bank-Organisation. Heutzutage: einfachste Hardwarelösung verwenden. (Genügt diese nicht, anderen Prozessortyp wählen).

Externe Beschleunigungsmaßnahmen

- Adresse für Ausgabezwecke ausnutzen
- Parallelisierung von Speicher- und Interfacezugriffen
- Speicher breiter gestalten, als Art Mikroprogrammspeicher betrieben (µC übernimmt Sequencing)
- Aufschalten von Daten auf Teile des Datenbus
- externe Steuer- bzw. Zwischenspeicher (adressierbar oder FIFO)

Einfache Multiprozessorkopplungen

- Direktverbindungen
- Dual-Port-RAMs
- einfache Bussysteme
- Schiebering-Strukturen
- geschaltete Verbindungen (Crossbar, Switched Fabric)