1. Grundlagen

1.1 Einführung

Die weitaus meisten Anwendungsaufgaben der Steuerungs-, Regelungs- und Automatisierungstechnik werden heutzutage mit Mikrocontrollern gelöst. Der Mikrocontroller ist ein vollständiger Computer in einem einzigen Schaltkreis. Programmspeicher, Datenspeicher und die Funktionseinheiten der Ein- und Ausgabe sind eingebaut. Somit stehen alle E-A-Anschlüsse zum Lösen der Anwendungsaufgabe zur Verfügung (Abbildung 1.1). Die einfachsten Systeme enthalten einen einzigen Mikrocontroller, dessen Anschlüsse direkt mit der jeweiligen Anwendungsumgebung verbunden sind, also mit Sensoren, Leistungsstufen, Bedienelementen, Anzeigeeinrichtungen usw. (Abbildung 1.2). Die Anwendungsfunktionen werden vom Programm erbracht, das im Mikrocontroller läuft. Es fragt alle Eingangsbelegungen ab und stellt alle Ausgangsbelegungen ein.

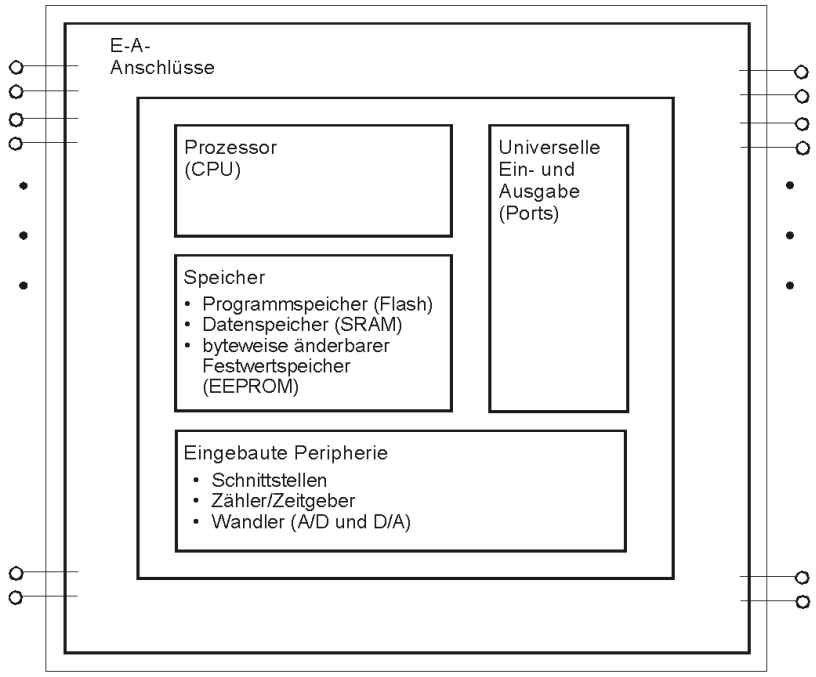


Abb. 1.1 Ein Mikrocontroller. Alle Funktionseinheiten befinden sich im selben Schaltkreis.

Zur Entwicklungsgeschichte
Als es noch gar keine Computer gab, blieb den Erfindern und Entwicklern kaum etwas anderes übrig, als nach geeigneten physikalischen Effekten und technischen Wirkprinzipien zu suchen und diese Ansätze bis hin zum brauchbaren Gerät konstruktiv durchzubilden. Hierbei sind nicht selten wirklich geniale Lösungen geschaffen worden. Ein typisches Beispiel ist der Zündverteiler im Auto. Die Aufgabe der Informationsverarbeitung besteht darin, den Zündzeitpunkt in Abhängigkeit von den aktuellen Betriebsverhältnissen des Motors zu bestimmen. Herkömmlicherweise hat man zwei Betriebskennwerte berücksichtigt: die Drehzahl (über Fliehgewichte – Fliehkraftverstellung) und die Belastung (über den Unterdruck im Saugrohr – Unterdruckverstellung). Abbildung 1.3 veranschaulicht, dass es sich um wirklich trickreiche mechanische Lösungen handelt.

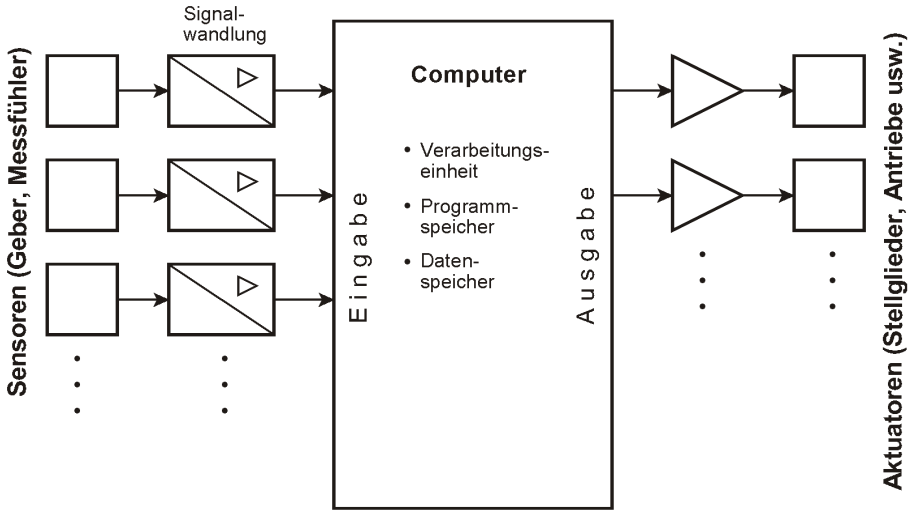


Abb. 1.2 Heuzutage steht der Mikrocontroller im Mittelpunkt.

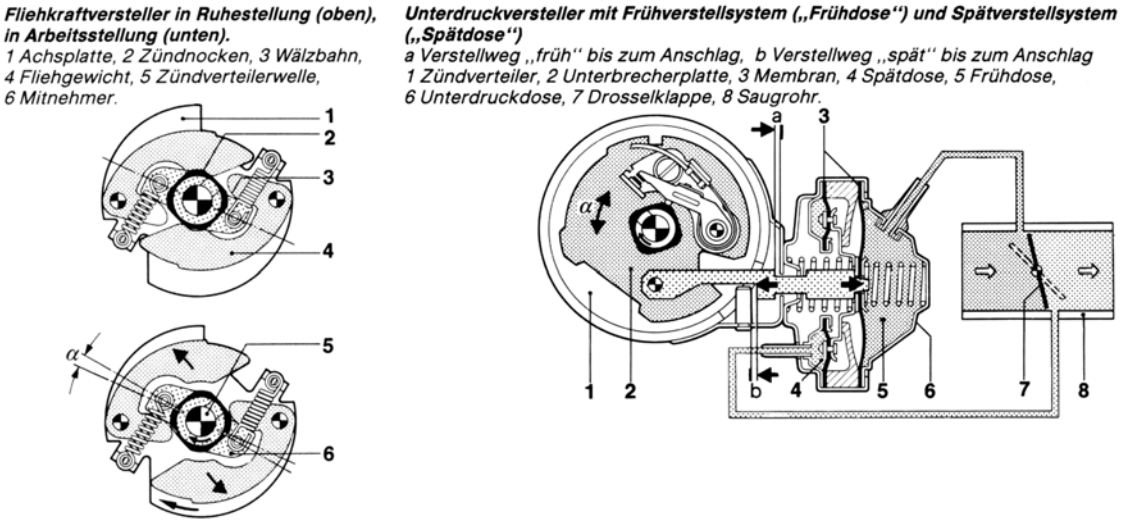


Abb. 1.3 Informationsverarbeitung „auf mechanisch" – der herkömmliche Zündverteiler (Bosch)[1].

Der Mikrocontroller ermöglicht es, das Erfassen der eingangsseitigen Kenngrößen und das Auslösen der ausgangsseitigen Wirkungen von der Informationsverarbeitung zu trennen (Abbildung 1.4). Die meisten Einsatzfälle lassen sich in einem allgemeinen Blockschaltbild ähnlich Abbildung 1.2 darstellen.

1: Welchem Windows- oder gar Linux-Programmierer würde wohl sowas einfallen?

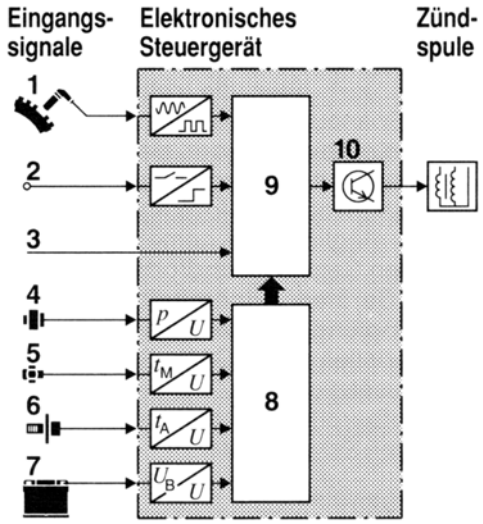


Abb. 1.4 Informationsverarbeitung mit Mikrocontroller – die elektronische Zündung (Bosch).1 - Motordrehzahl*); 2 - Schaltersignale; 3 - CAN-Bus (zur Vernetzung mit anderen Einrichtungen im Fahrzeug); 4 - Saugrohrdruck*); 5 - Motortemperatur*); 6 - Ansauglufttemperatur*); 7 - Batteriespannung; 8 - Mikrocomputer; 9 - Analog-Digital-Wandler; 10 - Leistungsstufe. *): Sensoren.

Wer heutzutage eine solche Aufgabe zu lösen hat, muss sich nach passenden Gebern und Messfühlern (Sensoren) und Stellgliedern oder Antrieben (Aktuatoren) umsehen sowie einen geeigneten Computer heraussuchen. (Offensichtlich bereitet das – zumindest in den weitaus meisten Fällen – beträchtlich weniger Mühe, als sich eine Anordnung ähnlich Abbildung 1.3 auszudenken.) Die Hauptarbeit besteht (meistens) nicht in der konstruktiven Durchbildung, sondern im Programmieren – es sind brauchbare Algorithmen zu (er)finden und in zuverlässig funktionierende Programme umzusetzen. Die Informationsverarbeitung gemäß dem Schema von Abbildung 1.2 hat sich weithin durchgesetzt – und zwar auch für Aufgaben, die man bisher schon vergleichsweise kostengünstig mit mechanischen oder elektromechanischen Mitteln lösen konnte.

In den 70er Jahren des vorigen Jahrhunderts wurde es möglich, Schaltkreise mit mehreren tausend Transistoren zu fertigen. Damals schon hätte man die meisten Anwendungsschaltungen auf einem einzigen Schaltkreis unterbringen können. Die Schaltkreisfertigung ist aber nur dann wirtschaftlich, wenn es sehr große Stückzahlen sind. Auch ist es sehr aufwendig, solche Schaltkreise zu entwickeln. In Gegensatz zur herkömmlichen Leiterplattentwicklung kann man nicht einfach probieren und Entwurfsfehler mit Draht beseitigen. Deshalb ist man darauf gekommen, universelle Schaltkreise zu schaffen, die in hohen Stückzahlen abgesetzt werden können. Nun gibt es nichts Vielseitigeres als den Universalrechner. Auf Grundlage einer überschaubaren, wohlbekannten Schaltungsstruktur kann man alles implementieren, was den Charakter eines Algorithmus hat – und das sind alle anwendungspraktisch in Betracht kommenden Abläufe und Informationswandlungen. Die Grenzen ergeben sich lediglich aus der Speicherkapazität und der Programmlaufzeit. Man baue einen kleinen Computer, versehe ihn mit universellen Anschlußschaltungen – und die Bauelementefrage ist grundsätzlich gelöst. An die Stelle der anwendungsspezifischen Schaltungsentwicklung tritt das Programmieren. Programme kann man noch viel leichter ändern als Leiterplatten. Hat man die Grundfertigkeiten des Programmierens erst einmal verinnerlicht, kann man sich der Lösung vieler Probleme durch fortlaufendes Probieren – mit anderen Worten, durch unbekümmertes Hacken – nach und nach annähern.

Die meisten Mikrocontroller werden nicht deswegen eingesetzt, weil es etwas zu rechnen gibt, sondern nur, um bestimmte Funktionen billiger zu erledigen als dies mit zweckgebundenen Schaltungen möglich wäre. In solchen Fällen kommt es oftmals auf geringste Hardwarekosten an. Der Schaltkreis ist soweit wie irgend möglich auszunutzen. Des zwingt oftmals zur maschinennahen Programmierung, zur Nutzung ungewöhnlicher Programmiertechniken – und nicht selten zum Tricksen auf Biegen und Brechen.

Bereits in den siebziger Jahren des vorigen Jahrhunderts waren in vielen Anwendungsfällen Mikrocontroller kostengünstiger als herkömmliche mechanische Lösungen[2]. Überall dort, wo Tasten und Schalter auszuwerten, Anzeigen anzusteuern und Schaltvorgänge auszulösen sind, war schon damals der Mikrocontroller das Mittel der Wahl.

Um derartige Anwendungslösungen auszuarbeiten, sind folgende grundsätzliche Aufgaben zu lösen:

- Es ist zunächst einmal zu erkennen, ob diese Einfachlösung ausreicht oder nicht.
- Ein passender Mikrocontroller ist auszuwählen.
- Dessen Ausstattung ist zweckmäßig auszunutzen.
- Er ist vernünftig zu programmieren. Hierzu sind geeignete Programmiermodelle und Programmierwerkzeuge auszuwählen.
- Es ist alles zum Laufen zu bringen.

Aus der Vielfalt der Problemstellungen – und dem üblichen Termindruck – ergeben sich typische Arbeitsbedingungen:

- Die komplette Anwendungslösung ist von Grund auf auszuarbeiten.
- Es gibt ein beträchtliches Maß an Wahlfreiheit (System- und Prozessorarchitekturen, Schaltkreise, Programmiermodelle usw.).
- Die Ressourcen sind knapp.
- Die Problemlösung steht unter Zeit- und Kostendruck – es muss alles schnell gehen und darf nichts kosten.
- Bei den Materialkosten kommt es oft auf die Stellen nach dem Komma an.
- Wir haben weder genügend Zeit noch können wir, um Schwierigkeiten aus dem Wege zu gehen, auf immer dickere Prozessoren oder Systeme zurückgreifen.
- Man kann nicht alles haben.
- Es läuft keineswegs alles ideal – man muss sich halt zu helfen wissen ...

Die typische Entwicklungsaufgabe ist eine Allround-Aufgabe (Abbildung 1.5):

- Die grundsätzliche Systemlösung ist zu finden.
- Die Bauelemente sind auszuwählen.
- Die Hardware ist zu entwerfen – und zwar unter Berücksichtigung aller Anforderungen der Praxis (EMV, ESD, Prüfbarkeit, wirtschaftliche Fertigung, Service (fertigungs-, prüf- und servicegerechter Entwurf)).
- Die Software ist zu erstellen.

2: Und zwar auch für Probleme, die gar nicht besonders kompliziert sind. Eine der ersten Anwendungen am Massenmarkt war die Steuerung einfacher Mikrowellengeräte. Hier hatte der Mikrocontroller nur die Funktion einer Zeitschaltuhr zu übernehmen.

- Falls erforderlich, ist eine Testumgebung aufzubauen.
- Es ist alles zum Laufen zu bringen.

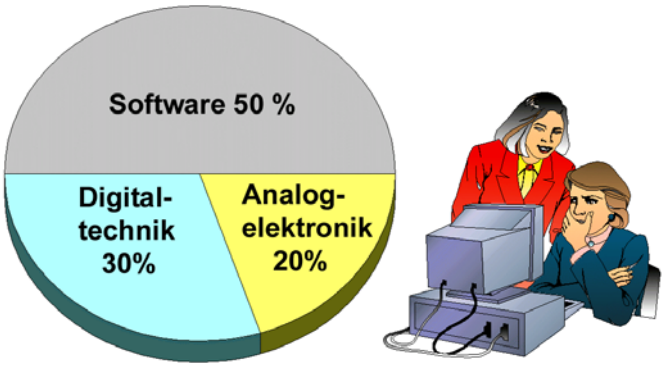


Abb. 1.5 Richtwerte. Das typische Entwicklungsvorhaben ist von ein bis zwei Ingenieuren in nicht mehr als sechs Monaten zu bewältigen. 50 % der Entwicklungsaufwendungen betreffen die Software, 30 % die Digitaltechnik und 20 % die analogen Schaltungen (nach Hewlett-Packard).

1.2 Elementare Programmabläufe und Programmbeispiele

Typische Anwendungsaufgaben führen auf grundsätzliche Programmschleifen, die folgende Schritte enthalten (Abbildung 1.6):

1. Eingabe. Die Eingänge lesen (Sensoren, Bedienfelder usw.).
2. Verarbeitung. Die eigentlichen Verarbeitungs- und Steuerabläufe ausführen. Hierbei werden der Folgezustand und die Ausgangsdaten bestimmt.
3. Ausgabe. Die Ausgänge schreiben (Anzeigen, Leistungsstufen usw.).
4. Zurück zu Schritt 1. Beim nächsten Durchlauf arbeitet die Schleife mit dem Folgezustand, der in Schritt 2 ermittelt wurde.

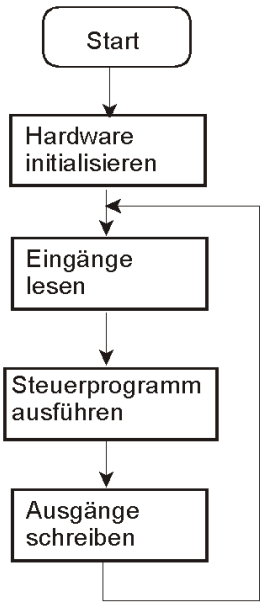


Abb. 1.6 Die typische Programmschleife einer Anwendungslösung.

Das Prinzip soll zunächst anhand überschaubarer Anwendungen veranschaulicht werden.

Zu den einfachsten Anwendungsbeispielen gehört das Steuern einer Jalousie. Im Beispiel von Abbildung 1.7 wird ein Gleichstrommotor eingesetzt. Für jede Bewegungsrichtung ist ein Relais vorgesehen, über dessen Kontakte die jeweils entsprechend gepolte Betriebsspannung angelegt wird. Die Endabschaltung erfolgt über mechanisch betätigte Ruhekontakte (Mikroschalter o. dergl.). Die Bewegung der Jalousie wird mit zwei Bedientasten gesteuert.

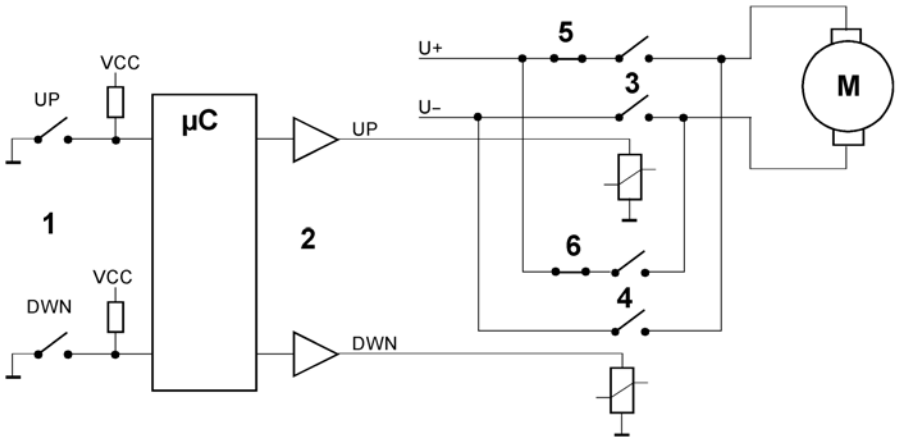


Abb. 1.7 Antriebssteuerung mittels Mikrocontroller. 1 - Tasten; 2 - Leistungsstufen; 3 - Aufwärtsrelais; 4 - Abwärtsrelais; 5 - oberer Endlagenkontakt; 6 - unterer Endlagenkontakt.

Als erstes soll das wohl einfachste Funktionsprinzip implementiert werden. Solange die Taste UP betätigt wird, läuft der Motor in Aufwärtsrichtung. Solange die Taste DWN betätigt wird, läuft der Motor in Abwärtsrichtung. Erreicht die Jalousie die jeweilige Endlage, wird der zugehörige Ruhekontakt aktiv und unterbricht den Motorstromkreis.

Hierzu braucht man eigentlich gar keinen Mikrocontroller. Simple Drahtverbindungen zwischen den Tastern und den Endlagenkontakten (Mikroschaltern) würden genügen. Trotzdem ist es lehrreich, die Funktionsbeschreibung in einen Programmablauf umzusetzen (Abbildung 1.8). Bereits diesese einfache Flußdiagramm zeigt die typische Programmstruktur aus einer Endlosschleife und den Behandlungsabläufen der einzelnen Funktionen.

Die Endlossschleife (Hauptsteuerschleife, Main Control Loop) fragt ab, was zu tun ist. Ist nichts zu tun, wird sie immer wieder durchlaufen. Ist etwas zu tun, verzweigt sie zum jeweiligen Behandler. Hat der Behandler seine Arbeit beendet, kehrt der Programmablauf zur Hauptsteuerschleife zurück.

Unschön ist, daß man den Daumen so lange auf der Taste lassen muß, bis die Jalousie die gewünschte Stelle erreicht hat. Sorgen wir also für etwas mehr Bedienkomfort. Ein bloßes Antippen soll genügen. Wenn nichts weiter geschieht, läuft die Jalousie bis zum Endanschlag durch. Um die Bewegung anzuhalten, soll es genügen,die jeweils andere Taste anzutippen. Beispiel: UP antippen – die Jalousie läuft nach oben – DWN antippen – die Jalousie bleibt stehen[3].

3: Das ist die typische Funktionsweise einfacher Jalousiensteuerungen.

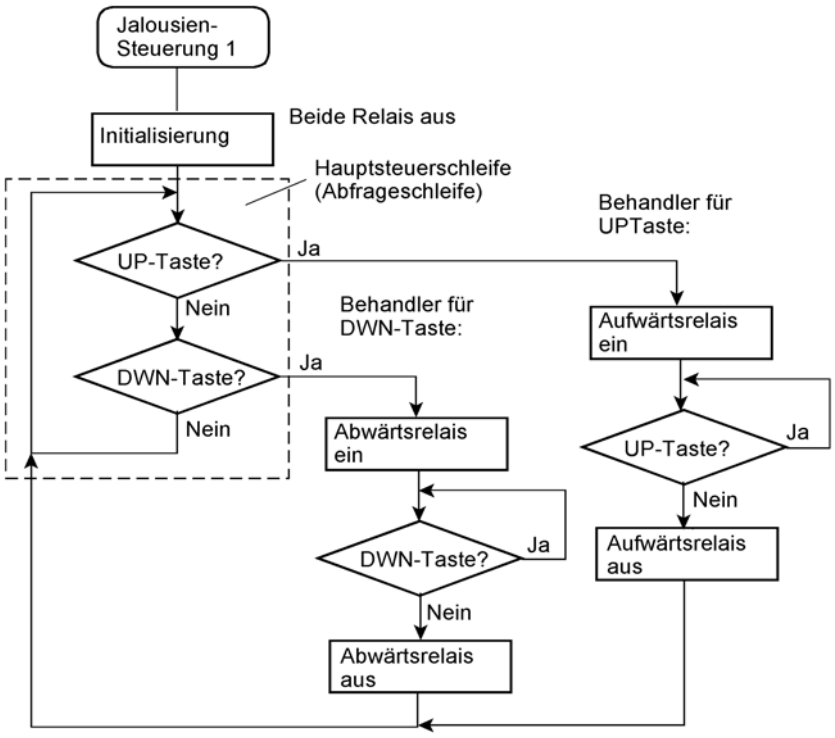


Abb. 1.8 Jalousiensteuerung ganz einfach. Spitzfindigkeiten – beispielweise das Entprellen der Kontakte – wurden hier vernachlässigt.

Diese Funktionsweise ist offensichtlich mit Draht allein nicht zu verwirklichen. Natürlich ginge es mit zusätzlichen Selbsthaltekontakten an den Relais oder mit RS-Flipflops. Solche Alternativen sind aber heutzutage teurer als ein Mikrocontroller. Der Programmablauf ist allerdings komplizierter, als es womöglich auf den ersten Blick aussieht. Das Grundproblem: wie erkennen wir, ob eine Tastenbetätigung eine Bewegung auslösen oder anhalten soll? Abbildung 1.9 veranschaulicht zwei Lösungen. Die eine erledigt das Problem mit einer Zeitkontrolle, die andere, indem sie auch die Endlagenkontakte auswertet[4].

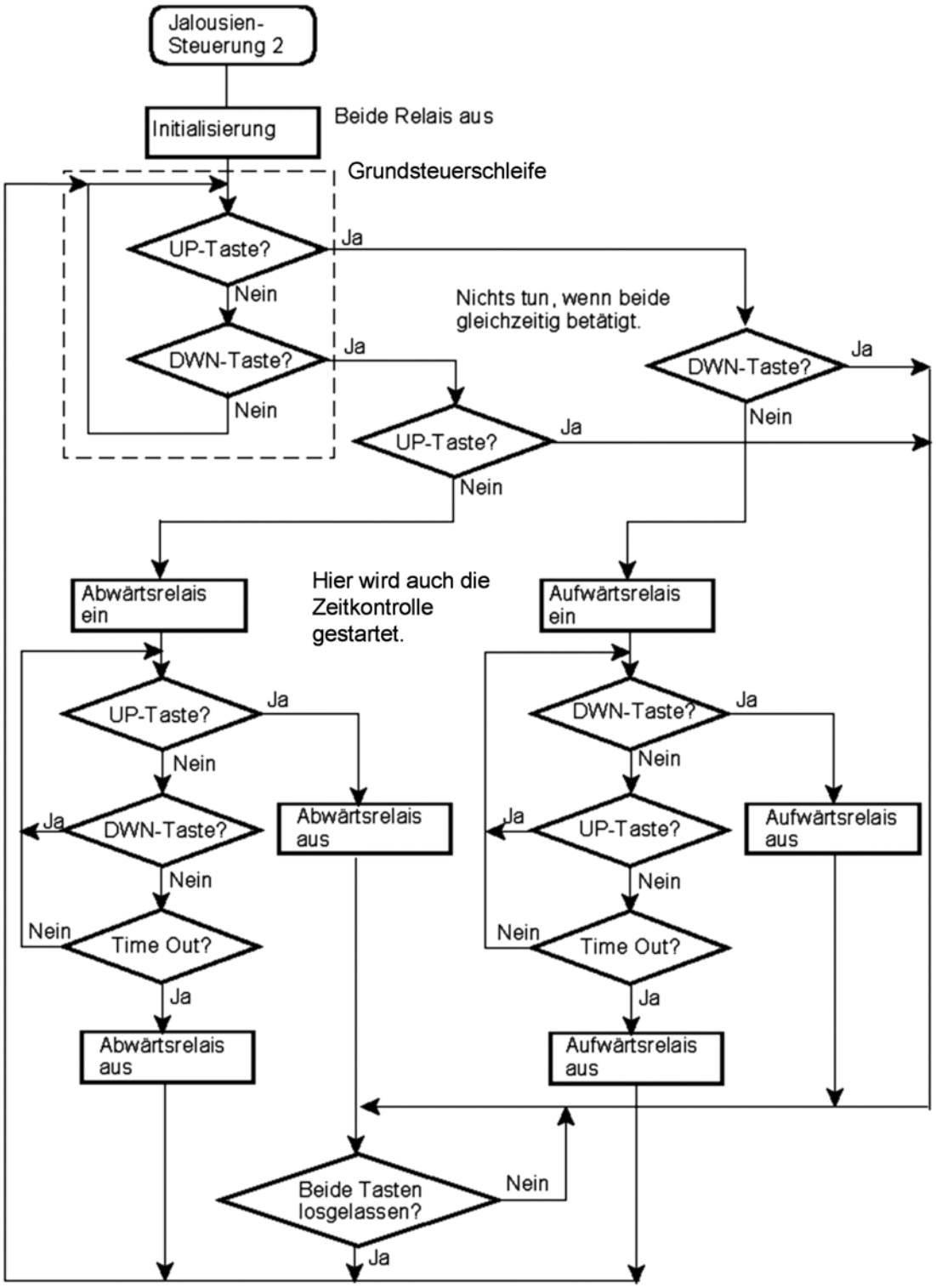
Hat man sich einmal für den Mikrocontroller entscheiden, ist die Funktionsvielfalt im Grunde unbegrenzt. Weitere Funktionen – so kompliziert sie auch sein mögen – kosten im Grunde nur noch Speicherplatz, also praktisch nichts. Der Funktionsumfang kann unter anderem bis zur ortsabhängigen (astronomischen) Zeitsteuerung[5] und bis zur Fernbedienung übers Internet erweitert werden ...

Die Abbildungen 1.10 und 1.11 veranschaulichen ein weiteres Anwendungsbeispiel. Es handelt sich darum, einen Toaster mit einem Mikrocontroller zu steuern.

4: Das erscheint als einleuchtende, bessere Lösung, hat aber den Nachteil, daß zusätzliche Leitungen von den Endlagenkontakten zur Steuerung erforderlich sind. Bei Elektroinstallateuren ist so etwas nicht gerade beliebt. Man kann jedoch die Endlage auch auf andere Weise erkennen... (Übungsaufgabe: wie?)

5: Öffnen bei Sonnenaufgang, Schließen bei Sonnenuntergang.

Eine erste Tastenbetätigung löst die Bewegung aus. Dabei wird auch eine Zeitkontrolle (Time Out) aktiv. In diesem Zeitintervall kann die jeweils andere Taste die Bewegung anhalten. Das Zeitintervall entspricht näherungsweise der längsten Laufzeit (von Anschlag zu Anschlag).



Hier kann man die Bewegung mit beiden Tasten anhalten. Wird die Jalousie nicht angehalten, endet der Bewegungszustand mit dem Erreichen der jeweiligen Endlage.

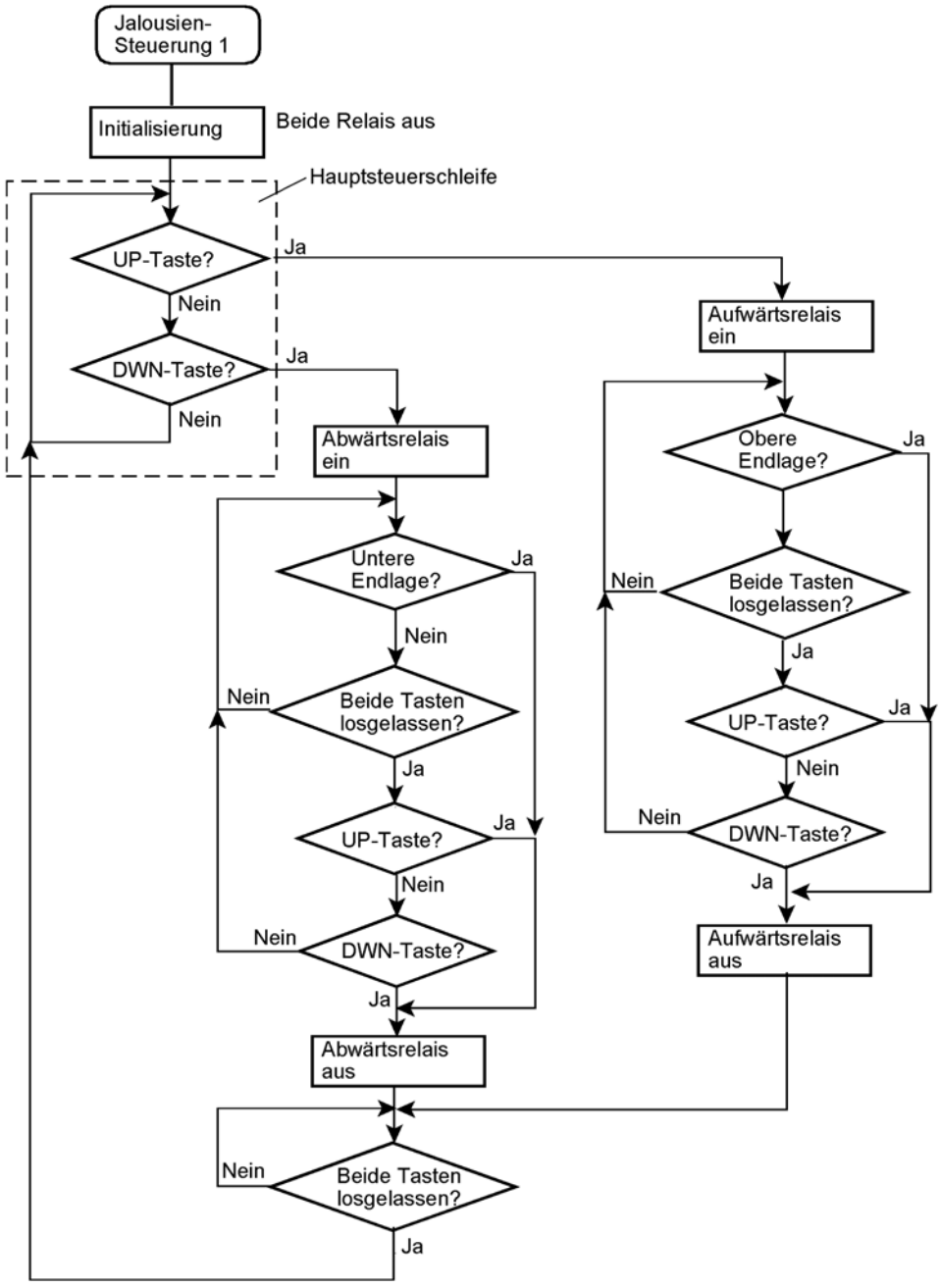


Abb. 1.9 Jalousiensteuerung mit bescheidenem, aber brauchbarem Bedienkomfort. Der Mikrocontroller muß jetzt erkennen, ob eine nachfolgende Tastenbetätigung die Jalousie anhalten oder erneut in Bewegung setzen soll.

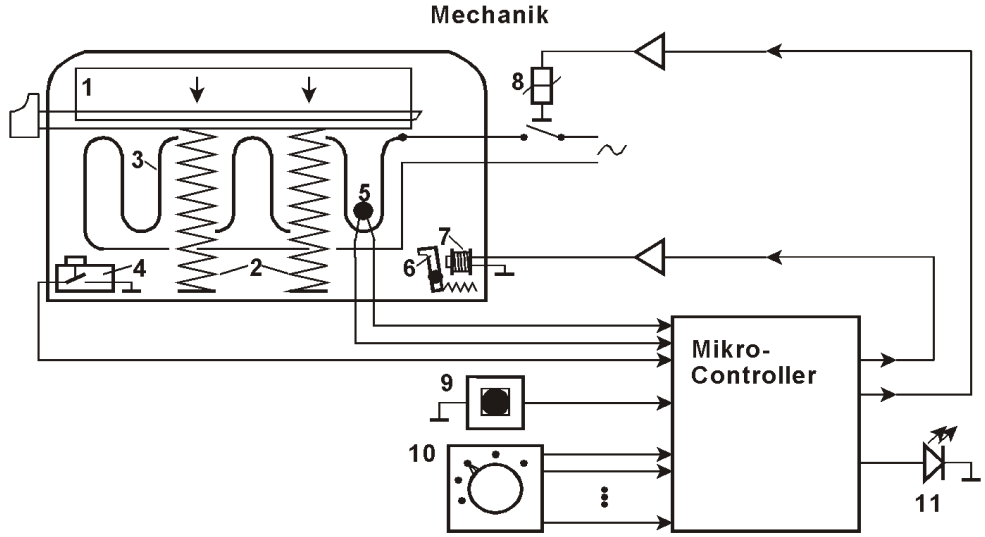


Abb. 1.10 Die zu steuernde Hardware. 1 - Korb; 2 - Druckfeder; 3 - Heizwendel; 4 - Endlagenkontakt; 5 - Temperatursensor; 6 - Klinke; 7 - Auslösemagnet; 8 - Schaltrelais (oder Triac); 9 - Stoptaste (zum Abbrechen des Toastvorgangs); 10 - Drehschalter zum Einstellen des Bräunungsgrades; 11 - Kontrollanzeige (Leuchtdiode).

Wir legen die Brotscheiben ein und drücken den Korb 1 nach unten. Dort wird er durch die Klinke 6 gehalten. Dieser Betriebszustand wird vom Endlagenkontakt 4 signalisiert. Damit beginnt der Toastvorgang. Um ihn zu beenden, wird der Auslösemagnet 7 erregt und somit die Klinke 6 ausgelöst. Daraufhin drückt die Druckfeder 2 den Korb 1 wieder nach oben. Zur Beeinflussung des Ablaufs sind eine Stoptaste 9 (vorzeitiges Beenden) und ein Drehschalter 10 (zum Einstellen des Bräunungsgrades) vorgesehen. Das Toasten selbst beruht auf einer Erregung der Heizwendel 3. Hierzu wird das Relais 8 angesteuert.

Erläuterungen zum Programmablauf:

1) Nach dem Einschalten wird alles in die Grundstellung versetzt (Initialisierung). Ist der Korb 1 unten, wird er ausgelöst.
2) Der Korb 1 darf nicht (in der unteren Lage) eingerastet sein. Ggf. warten, bis der Endlagenkontakt 4 abgeschaltet hat.
3) Warteschleife im Ruhezustand.
4) Mit dem Einrasten des Korbes 1 (Meldung über Endlagenkontakt 4) beginnt der Toastvorgang.
5) Die Stoptaste 9 wird immer wieder abgefragt, um zu erkennen, ob der Vorgang abgebrochen werden soll.
6) Der Drehschalter 10 wird immer wieder abgefragt. Somit kann man den gewünschten Bräunungsgrad auch ggf. mitten im Ablauf ändern (Bedienkomfort).
7) Mittels Temperatursensor 5.
8) Durch interne Zeitzählung im Mikrocontroller.
9) Die bisher umgesetzte Wärmemenge wird aus Temperatur und Zeit errechnet (und in jedem Schleifenumlauf aufsummiert). Es handelt sich hier um eine interne Hilfsgröße, die nicht normgerecht (als SI-Einheit J (Joule)) ermittelt werden muß.

10) Das ist ein einfaches, oft angewendetes Prinzip: man rechnet nicht mit komplizierten Formeln, sondern man hat im Mikrocontroller eine Wertetabelle fest gespeichert, die zu jedem einstellbaren Bräunungsgrad die zugehörige Wärmemenge angibt (die Werte wurden während der Entwicklung durch Versuch bestimmt).
11) Die Schleife des Toastvorgangs.
12) Zurück zur Warteschleife. Hat der Korb die untere Endlage verlassen, kann ein neuer Toastvorgang gestartet werden.

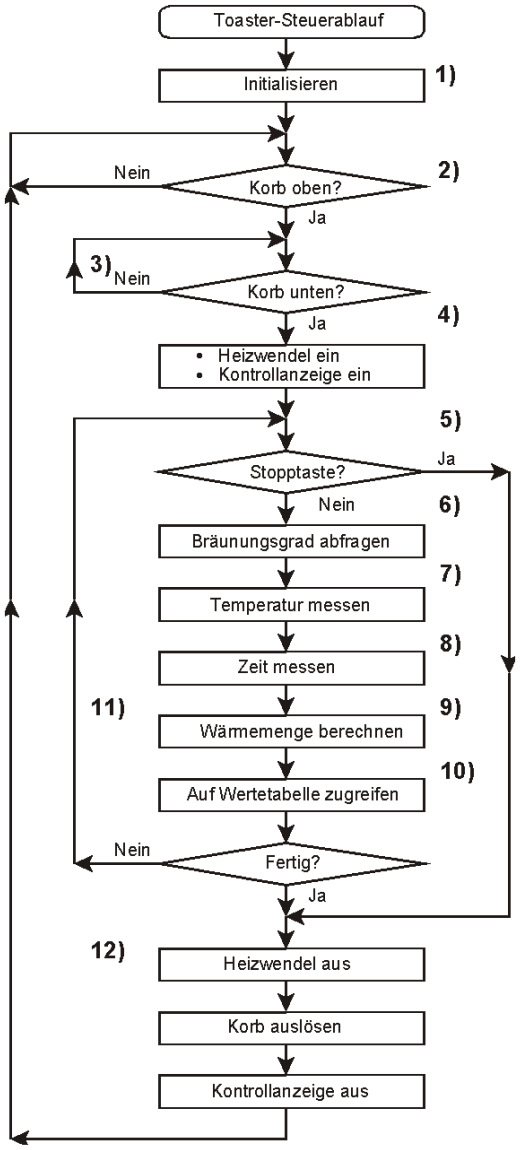


Abb. 1.11 Programmablauf. Erläuterungen vorstehend.

Auch dieses Beispiel zeigt das Organisationsprinzip eines typischen Anwendungsprogramms:

- Es wird gestartet.
- Es richtet sich ein (Initialisierung).
- Es fragt ab, ob etwas zu tun ist.
- Ist etwas zu tun, so wird es ausgeführt (Behandlung der jeweiligen Anforderung).
- Ist die Arbeit erledigt, wird wiederum abgefragt, ob etwas zu tun ist (Abfrageschleife).

1.3 Programmentwicklung

Herkömmliche (prozedurale) Programmierung
Programmieren heißt, eine funktionelle Absicht auf das Ausführen von Anweisungen zurückzuführen, die Bits transportieren (Eingabe, Ausgabe, Laden, Speichern) oder miteinander verknüpfen (Operationen) oder die den Programmablauf beeinflussen (Verzweigungen). Ein Programm ist im Grunde eine Anweisungsfolge. Eine zu steuernde Einrichtung wird kommandiert: führe A aus, dann führe B aus, wenn X, dann führe Y aus, wenn nicht X, dann führe Z aus usw. Die Anweisungen müssen aus einem jeweils vorgegebenen Anweisungsvorrat entnommen werden (Programmiersprache oder Maschinensprache). Sie beziehen sich auf gespeicherte Daten oder auf Schnittstellen der Hardware (Ein- und Ausgabe). Auch die Struktur der Daten entspricht bestimmten Vorgaben (Datentypen).

Diese Programmierweise hat sich auf sozusagen natürlichem Wege ergeben. Die ersten Computer wurden als Rechenmaschinen mit automatischer Steuerung erfunden. Der Maschine ist – wie einem menschlichen Rechner – zu sagen, was sie nacheinander zu tun hat. Ganz am Anfang wurden die Bitmuster der Programme von Hand aufgesetzt. Dann wurden symbolische Maschinensprachen (Assemblersprachen) entwickelt. Schließlich wurden höhere Programmiersprachen geschaffen, um die Programmierarbeit zu erleichtern, den Programmierkomfort zu erhöhen und das Programmieren von den Eigentümlichkeiten der Maschinen unabhängig zu machen. Das Programmieren – gleich mit welchen Sprachmitteln und Entwicklungswerkzeugen – besteht aber stets darin, Prozeduren – also Handlungsanweisungen – zu schreiben, die auf Variable – also Daten – einwirken.

Jedes Programmieren führt – zumindest beim aktuellen Stand der Technik – letzten Endes auf prozedurale Programme, denn es kommen immer Maschinenprogramme heraus, die auf gewöhnlichen Prozessoren laufen.

Wer Aufbau und Wirkungsweise der Prozessoren sowie Feinheiten und Spitzfindigkeiten der Programmabläufe kennenlernen möchte, muß sich auch mit dem prozeduralen – vor allem dem maschinennahen – Programmieren beschäftigen. Deshalb werden wir uns nachfolgend auf diese Art des Programmierens beschränken. Zunächst sollen jedoch einige Alternativen kurz vorgestellt werden.

Objektorientierte Programmierung
Der Begriff der objektorientierten Programmierung bezeichnet nichts anderes als eine bestimmte Art von Programmierwerkzeugen und -hilfsmitteln. Sie betreffen vor allem die Wiederverwendung, Abwandlung und Weiterentwicklung der Programme sowie das Vermeiden elementarer Programmierfehler. Die anwendungsbezogene Problemlösung ist jedoch nach wie vor im Kopf zu durchdenken und von Hand auszuarbeiten. Und auch beim objektorientierten Programmieren entstehen letzten Endes Maschinenprogramme, die auf gewöhnlichen Prozessoren laufen.

Programmgeneratoren und interpretative Systeme
Das herkömmliche Programmieren – in einer Sprache wie C oder gar Assembler – ist zwar von den Grundlagen her nichts weiter als eine Art Kommandieren – also etwas, das sich geradezu von selbst versteht. Es ist aber nicht jedermanns Sache. Die Einarbeitung ist zeitaufwendig; es ist viel zu lernen, und ohne Übung wird es nichts rechtes. Deshalb hat man Programmentwicklungssysteme geschaffen, die es ermöglichen, die Programmierabsicht mit

Beschreibungsmitteln zu erfassen, die im jeweiligen Anwendungsbereich üblich sind. Es sind letzten Endes Struktur- oder Funktionsbeschreibungen. Hiermit kann der Anwender seine Problemlösung selbst formulieren.

Typische Strukturbeschreibungen beruhen auf Kontaktplänen, Funktionsblöcken oder Schaltsymbolen (Abbildung 1.12 bis 1.15). Typische Funktionsbeschreibungen beruhen auf Flußdiagrammen, Zustandsdiagrammen und Modellierungssprachen (Abbildung 1.16 bis 1.19). Die Beschreibung wird von einem Programmgenerator in einen Programmquelltext umgesetzt oder von einem interpretierenden Programm (Emulator) ausgewertet. Im Bereich der Mikrocontroller wird der Programmgenerator (Code Generator) bevorzugt. Manche Programmgeneratoren erzeugen den Maschinencode direkt. Am weitesten verbreitet ist jedoch der Weg über zwischengeschaltete Sprachebenen. Aus der Beschreibung wird zunächst ein Code in einer höheren Programmiersprache (zumeist in C) erzeugt, der mit einem üblichen Compiler übersetzt werden kann. Manche Entwicklungsumgebungen unterstützen den gesamten Weg, von der Beschreibung über den C-Code, den Assemblercode für den jeweils gewählte Maschine bis hin zum binären Maschinencode. Dieses Verfahren hat zwei Vorteile:

- Compilierte Maschinenprogramme laufen deutlich schneller als eine Emulation (Richtwert: 10 bis 50mal).
- Man kann auf dem Übersetzungsweg eingreifen und den Code (in C oder Assembler) nach eigenen Vorstellungen abwandeln.

Beschreibungsmittel im Verbund einsetzen
Jedes Beschreibungsmittel ist für bestimmte Arten von Aufgaben besser geeignet als für andere. So nützt ein Zustandsdiagramm nicht viel, wenn Formelausdrücke zu berechnen sind (wie im Beispiel von Abbildung 1.14). Flußdiagramme sind eine anschauliche Darstellung sequentieller Abläufe. Sie sparen aber nur dann Arbeit, wenn man nicht jede Einzelheit graphisch darstellen muß. Deshalb modelliert man mit dem Flußdiagramm nur den grundsätzlichen Ablauf und beschreibt das, was die Blöcke im einzelnen leisten sollen, mit einer Programmiersprache (vergleiche Abbildung 1.18).

Von Hand programmieren
Manchmal ist die Entwicklungsumgebung zu kostspielig oder für den jeweiligen Prozessortyp nicht verfügbar oder sie reicht nicht aus, um die gesamte Entwurfsaufgabe zu erledigen. Es liegt dann nahe, solche Beschreibungsmittel nur zum Erfassen der Entwurfsabsicht einzusetzen und das eigentliche Programm gleichsam zu Fuß zu erstellen. Flußdiagramme und Funktionspläne können in übliche Programmtexte umgesetzt werden. Kontaktpläne und Zustandsdiagramme führen letzten Endes auf das Ausrechnen Boolescher Ausdrücke und auf elementare Fallunterscheidungen.

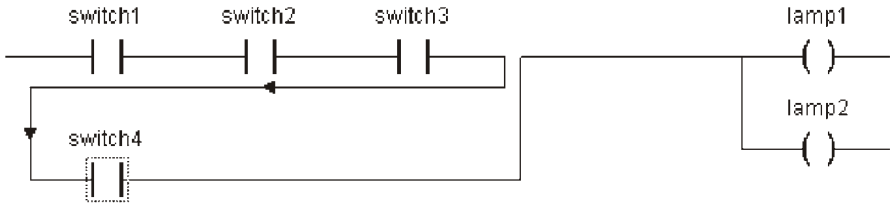


Abb. 1.12 Beispiel eines Kontaktplans (IEC61131 KP).

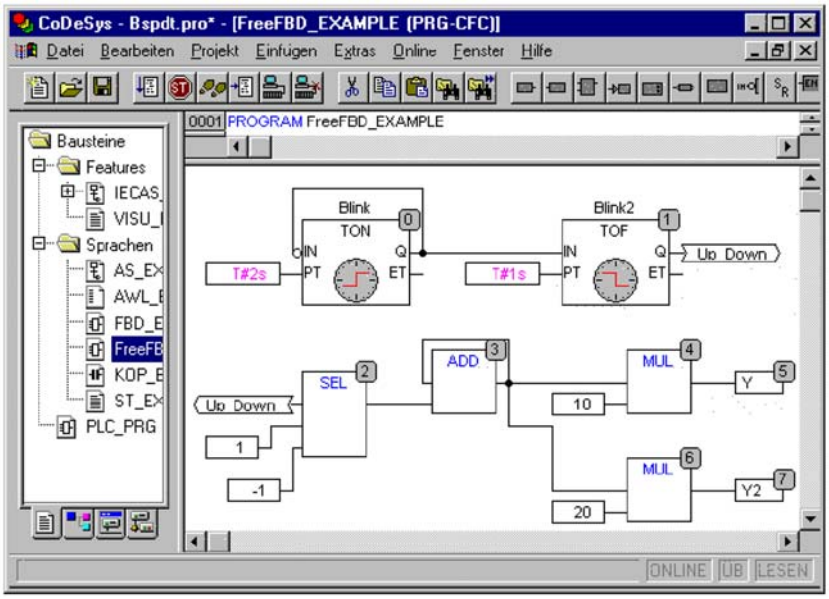


Abb. 1.13 Funktionsblöcke in einer Entwicklungsumgebung (IEC61131 FUP).

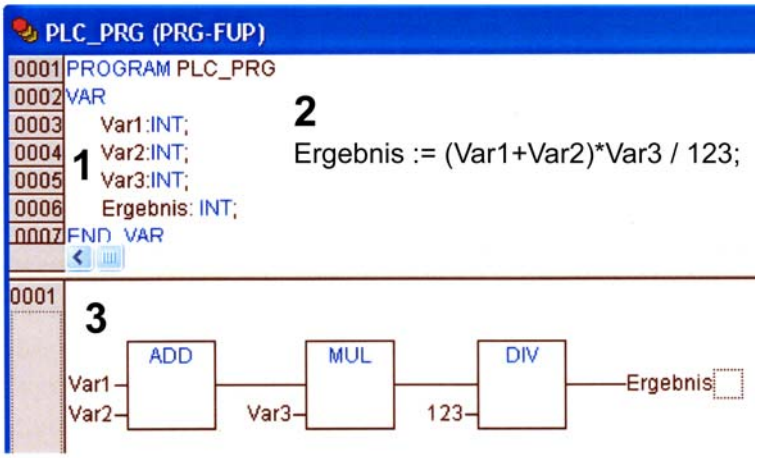


Abb. 1.14 Auf verschiedene Weise programmieren (IEC61131). 1 - Deklaration der Variablen; 2 - Formulierung in der Programmiersprache Structured Text (ST); 3 - Erfassung als Funktionsplan. Die Funktionsblöcke werden gemäß dem Datenfluß hintereinandergeschaltet.

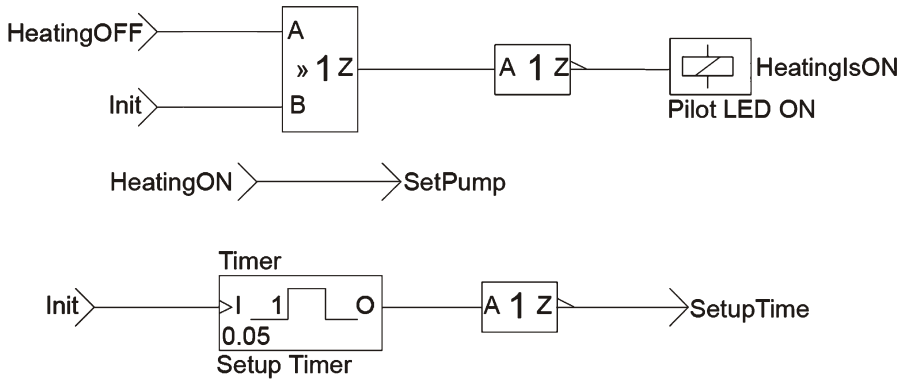


Abb. 1.15 Funktionsblöcke und Schaltsymbole in einer anderen Entwicklungsumgebung.

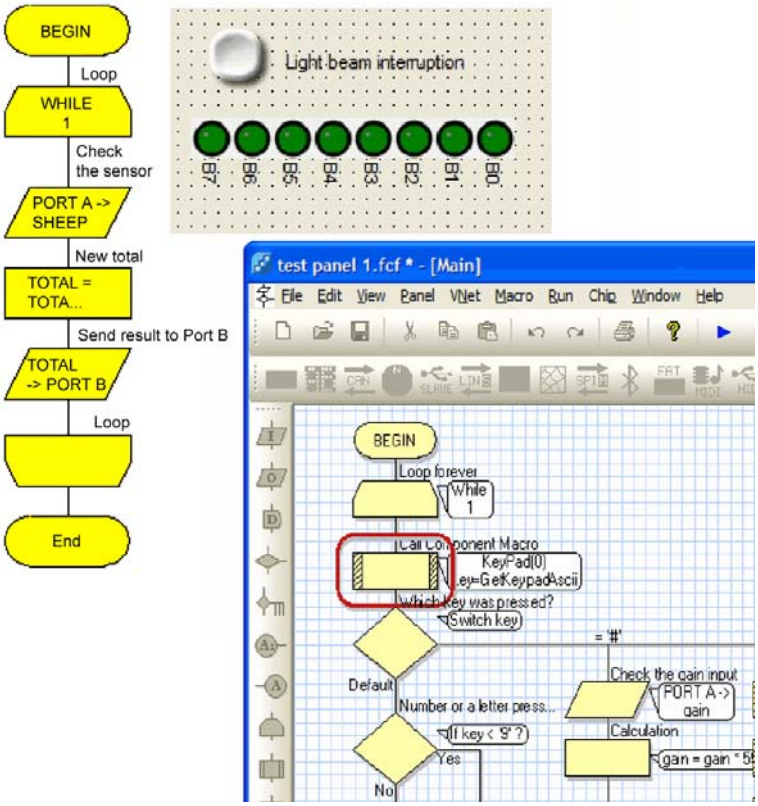


Abb. 1.16 Eine Entwicklungsumgebung, die auf Flußdiagrammen beruht. Neben den Programmabläufen können auch die Einrichtungen der Ein- und Ausgabe graphisch dargestellt werden, um das Programm am Bildschirm zu simulieren.

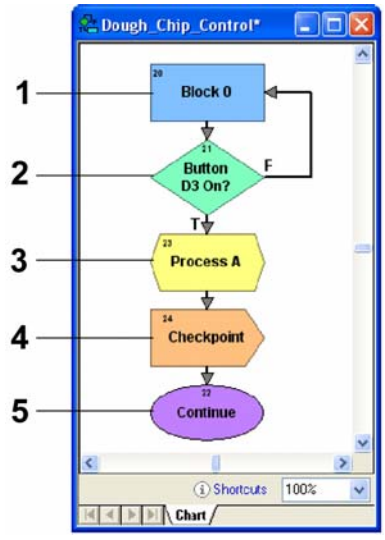


Abb. 1.17 Eine weitere Entwicklungsumgebung auf Grundlage von Flußdiagrammen. 1- Aktionsblock; 2 - Bedingungsblock; 3 - OptoScript-Block; 4 - Prüfpunktblock; 5 - Fortsetzungsblock. OptoScript ist die zum System gehörende Programmiersprache.

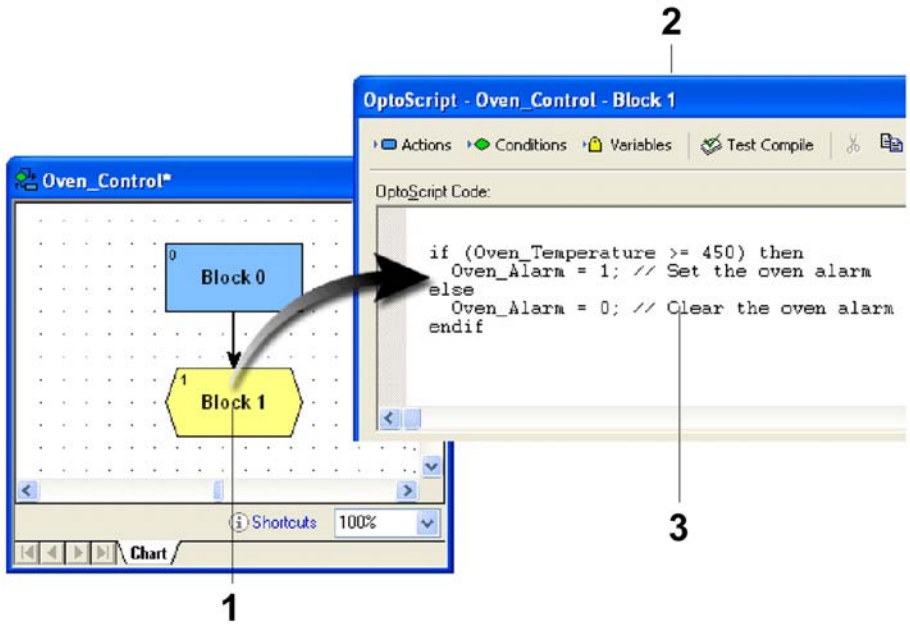


Abb. 1.18 Was ein Block leisten soll, wird in der zugehörigen Programmiersprache formuliert. 1 - OptoScript-Block; 2 - Editorfenster; 3 - Quelltext des Programmcodes.

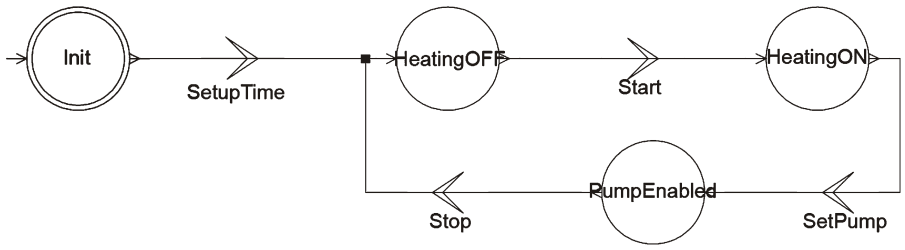


Abb. 1.19 Ein Zustandsdiagramm.

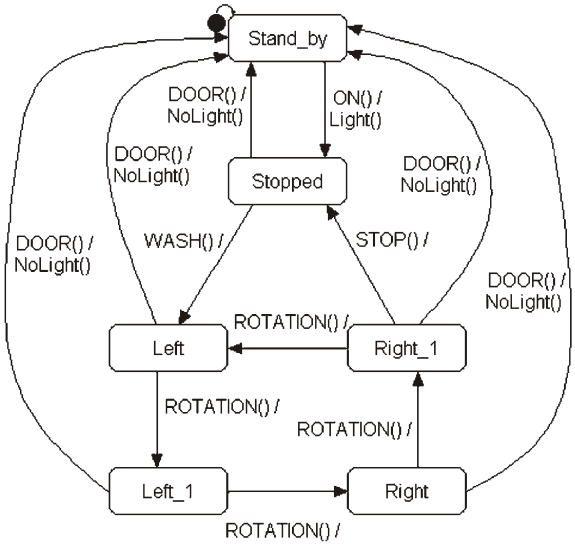


Abb. 1.20 Ein Zustandsdiagramm in einer Entwicklungsumgebung, die auf der Modellierungssprache UML beruht.

Maschinennahe Programmierung
Maschinennahe Programme werden in der jeweiligen Assemblersprache oder in einer höheren prozeduralen Programmiersprache geschrieben. Die Programmiersprache C wird am häufigsten eingesetzt. C ist im Grunde ein halbwegs maschinenunabhängiger Assembler mit vorgefertigten Kontrollstrukturen und Bibliotheksfunktionen.

Weshalb in Maschinensprache (Assembler) programmieren?

- Es ist in der Praxis nach wie vor erforderlich (Nutzung maschinenspezifischer Besonderheiten, maximale Ausnutzung der Hardware (höchstes Leistungsvermögen oder geringster Aufwand), Umgehung von Unzulänglichkeiten (Workarounds), bis auf den einzelnen Maschinentakt exaktes Zeitverhalten).
- Es vermittelt grundlegendes Erfahrungswissen zum Verstehen, Beurteilen und Auswählen von Prozessorarchitekturen.

Ein Volksmärchen:
"C ist eine maschinenunabhängige höhere Programmiersprache. Ein richtig geschriebenes C-Programm läuft auf jeder Maschine. Man muß es nur entsprechend übersetzen." Schön wär's ...

Die harten Tasachen:
C ist NICHT maschinenunabhängig – vor allem nicht im Bereich der Mikrocontroller. Zudem ist es nicht einmal unabhängig vom Compiler. Der Compiler macht, was er will, und nimmt keine Rücksicht auf subtile Programmierabsichten, wie sie für die Mikrocontrollerprogrammierung typisch sind. Das betrifft u. a.

- die Unterbringung von Variablen in verschiedenartigen Speicherbereichen (SRAM, Flash, EEPROM usw.),
- die Notwendigkeit, auf manche Variable immer wieder zuzugreifen, obwohl sie im Programmverlauf eigentlich nicht verändert wurden (weil es sich um Werte handelt, die von außen oder anderswo her kommen (Eingaben)),
- Befehlsfolgen, die eigentlich nichts berechnen, sondern nur dazu da sind, um Zeit zu verbrauchen.

Da solche Unterscheidungen nicht in den Sprachstandard eingebaut sind, hat sie jeder Compilerautor auf eigene Faust definiert. Selbst bei Beschränkung auf eine einzige Zielarchitektur (z. B. Atmel AVR) entspricht somit jeder Compiler praktisch einem eigenen C-Dialekt. Beim Übergang von Compiler A auf Compiler B muß im Quelltext geändert werden. Wer die Eigentümlichkeiten der jeweiligen C-Umgebung nicht kennt, kann entsprechend geschriebene Quelltexte nicht verstehen. Programmierempfehlungen zur Optimierung hängen sowohl von der Maschinenarchitektur als auch vom Compiler ab[6].

– C, ohnehin schon häßlich genug, wird noch häßlicher. –

6: Zu den wichtigsten Zielen der Entwicklung höherer Programmiersprachen sollte es gehören, daß man Programmabläufe verständlich, wenn nicht gar evident (also von Grund auf einleuchtend) formulieren kann. Im Idealfall ist der Programmtext seine eigene Dokumentation. C ist in dieser Hinsicht besonders schlecht. Gegenbeispiel: Ada. Die Anwendung von Optimierungsempfehlungen, akademischen Weisheiten und Tricks unfehlbarer Gurus macht C-Quelltxete vollends unlesbar. Solche Programme versteht oftmals nur der Urheber – und acht Wochen später auch der nicht mehr...

C als Mittel zur Gewährleistung der Maschinenunabhängigkeit
In der Theorie schaut alles viel großartiger aus als es wirklich ist. Wenn man wirkliche Maschinenunabhängigkeit haben will, muß man entsprechend entwickeln. Diese Entwicklungsmethodik kostet aber Zeit.

C als Mittel zur Arbeitserleichterung
Das ist das Hauptmotiv, weshalb man C einsetzt. Man darf sich nur nicht der Illusion hingeben, die Entwicklungsleistung wären maschinenunabhängig und durch einfaches Kompilieren auf x-beliebige andere Maschinen zu übertragen. Deshalb ist es gängige Praxis, bei einer Prozessorachitektur und einer Programmentwicklungsumgebung zu bleiben und deren Nachteile hinzunehmen (zumal professionelle Entwicklungsumgebungen auch richtig Geld kosten[7] – die Annahme, es sei alles umsonst, ist eine irrtümliche ...).

Weshalb nehmen wir eigentlich C? (Es zählen nur vernünftige Gründe.)
- Um schneller voranzukommen.
- Um tatsächlich – soweit es irgendwie geht – trotz allem maschinenunabhängig zu werden (Portabilität).
- Der Compiler kümmert sich um
  - den Unterprogrammaufruf,
  - das Belegen, Retten und Wiedereinstellen von Registern,
  - die Speicherbelegung,
  - das Adressieren der Variablen,
  - die Kontrollstrukturen (Entscheidungen, Schleifen usw.),
  - die elementare Arithmetik (= alle vier Grundrechenarten mit den gängigen Datentypen),
  - weitere elementare Funktionen (Bibliotheksroutinen), bis hin zu Zeichenketten, Gleitkommazahlen, Winkelfunktionen usw.

Andernfalls – wenn wir von Grund auf alles in Assembler programmieren – müßten wir all dies zu Fuß tun bzw. eigene Lösungen finden.

Gutgemeinte Empfehlungen:

1. Teile und herrsche. Das Problem aufteilen: was ist innere Programmlogik (maschinenunabhängig), was ist echte Ein- und Ausgabe (unvermeidlich maschinenspezifisch), was sind organisatorische Kleinigkeiten (z. B. Speicheraufteilung)?
2. Von oben herangehen (Top-Down) und schrittweise verfeinern. Problemlösung zunächst – ganz abstrakt – nur in C bearbeiten: Variablen – Programmlogik – Funktionen an den Schnittstellen zur Hardware ("logische" Ein- und Ausgabe).
3. Die typische C-Schnittstelle ist ein Funktionsaufruf. Alles,was sich nicht mit standardgemäßem (ANSI-) C ausdrücken läßt, zunächst mit Funktionen erledigen (und wenn diese zunächst bloße Attrappen (Dummies) sind).
4. Diese Funktionen schrittweise verfeinern. Wenn erforderlich, durch Assembler-Programmstücke ersetzen. (Notfalls durch Suchen und Ersetzen im Editor.)
5. Wenn möglich, Programmlogik auf PC ausprobieren. Funktionskörper ggf. passend abwandeln. Z. B. Eingabe über Tastatur oder Schnittstelle, Ausgabe auf Bildschirm bis hin zur Nachbildung der Peripherie mit Windows-Steuerelementen oder Kombination aus PC und angeschlossener Peripherie-Nachbildung.

7: Es geht praktisch nicht ohne Wartungsvertrag. Auch die teuersten Entwicklungssysteme enthalten Fehler...

6. Mit C assemblermäßig programmieren. Möglichkeiten der Sprache nicht bis zum Äußersten ausnutzen (Write High-Level, Think Low-Level).
7. Man hält sich entweder an akademische Weisheiten oder schreibt Programme, die in endlicher Zeit fertig werden und vernünftig laufen. Keine Angst vor GOTOs, globalen Variablen usw. Der wirkliche Könner ist nicht derjenige, der solche Programmiertechniken ängstlich vermeidet, sondern der weiß, wo man sie einsetzt und wo nicht.
8. Compilerspezifische Komfort-Funktionen nicht ausnutzen.
9. Systematisch entwickeln.
10. Alles richtig dokumentieren. C-Quelltext – auch mit Kommentaren – ist keine brauchbare Programmdokumentation!
11. Das Codieren sollte höchstens 30 % der Programmentwicklungzeit erfordern.
12. Erst denken, dann hacken.
13. Das Internet weiß nicht alles.

1.4 Atmel AVR – das Lehrbeispiel

Die Anfangsgründe der maschinennahen Programmierung und der Rechnerarchitektur gehören zusammen. Beide Wissensgebiete sind keine exakten Wissenschaften. Zwar versucht man sich immer wieder an formalisierten Theorien, aber es ist nach wie vor üblich, in der Lehre mit dem grundsätzlichen Aufbau des Universalrechners und dessen Nutzung zu beginnen. Dabei bezieht man sich meistens auf konkrete Beispiele. Rechnerarchitekturen werden (auch in Hochschullehrbüchern) zumeist ähnlich beschrieben wie Tier- oderPflanzenarten in der Biologie. Wir verpassen also nicht viel, wenn wir uns tiefgründige Theorien schenken und sofort beginnen, uns in eine bestimmteArchitektur einzuarbeiten. Es sollte eine richtige, in der Anwendungspraxis verbreitete Architektur sein; also keine, die eigens zu Lehrzwecken entwickelt wurde. Nur so kann man praxistypische Beschränkungen und Spitzfindigkeiten (Gotchas) am konkreten Beispiel kennenlernen. Die Architektur soll aber auch überschaubar sein; es muß möglich sein, sich in kurzer Zeit einzuarbeiten und nach wenigen Stunden erste Erfolge zu erzielen.

Die AVR-Mikrocontroller der Fa. Atmel haben sich als sehr zweckmäßiger Kompromiß erwiesen. Wichtig ist, wieviel Zeit der Lernende braucht, um vom Zustand der vollkommenen Kenntnislosigkeit bis zum ersten Verständnis und zu ersten Erfolgserlebnissen zu kommen, also zu Programmen, die wirklich laufen. Es gibt einfachere Architekturen. Diese weisen aber stärkere Einschränkungen auf. Man muß zwar weniger Befehle lernen, es sind aber viel mehr Besonderheiten und Spitzfindigkeiten zu beachten. Nun ist auch AVR vonEinschränkungen nicht frei. Einige sind sogar SEHR ärgerlich. Es ist aber nicht grundsätzlich schwierig, sich zu behelfen. Andere Mikrocontrollertypen haben ein beträchtlich höheres Leistungsvermögen und eine bei weitem umfangreichere Ausstattung. Sie sind aber auch viel komplizierter. Die Einarbeitung vom Stand Null an ist nicht in wenigen Stunden zu schaffen. Sie kostet mehrere Wochen am Stück (Richt- und Erfahrungswert: wenigstens vier). Um die E-A-Ports eines modernen Hochleistungsprozessors auch nur zu initalisieren, braucht man bereits mehrere Seiten Programmtext. Die E-A-Ports der AVR-Mikrocontroller sind hingegen vergleichsweise einfach. Ihre Wirkungsweise ist leicht zuverstehen, und es genügen einige Programmzeilen, um sie zu initialisieren. Vor allem aber haben sie nicht die häßlichen Nebeneffekte, die die E-A-Ports anderer Mikrocontrollerfamilien aufweisen. Näheres dazu in Abschnitt 4.1.

2. Universalrechner

2.1 Grundlagen

Alle Mikrocontroller und Prozessoren sind programmgesteuerte Universalrechenmaschinen. Betrachten wir diese Begriffe im einzelnen:

- Rechenmaschine: Hauptsache ist das numerische Rechnen, wenigstens in den Grundrechenarten.
- Programmgesteuert: es soll alles automatisch ablaufen.
- Universell: es sollen sich alle überhaupt denkbaren Rechenvorgänge ausführen lassen. Die praktischen Beschränkungen liegen nicht im Grundsätzlichen, sondern in Verarbeitungszeit und Speicherbedarf. Mit anderen Worten: durch Programmieren kann man *jeden beliebigen* Algorithmus verwirklichen – vorausgesetzt, das Programm passt in den Speicher und die Ausführungszeit spielt keine Rolle (unter diesen Voraussetzungen könnte auch der kleinste Prozessor die größten Aufgaben bewältigen).

Die ursprünglichen erfinderischen Ansätze waren zunächst vom Vorgehen eines Menschen angeregt, der komplizierte Rechnungen auszuführen hat. Hierzu ein einfaches Beispiel – eine Formel aus einer Formelsammlung der Elektrotechnik:

$$U_A = -R_G \cdot \left(\frac{U_1}{R_1} + \frac{U_2}{R_2} \right)$$

Das Ausrechnen erfordert offensichtlich die folgenden Schritte:

1. $U_1 : R_1$ berechnen. Zwischenergebnis notieren.
2. $U_2 : R_2$ berechnen. Zwischenergebnis notieren.
3. Beide Zwischenergebnisse addieren.
4. Diesen Wert mit R_G multiplizieren.
5. Das Vorzeichen wechseln.

Es sind also mehrere Rechenoperationen der Grundrechenarten nacheinander auszuführen. Die einfachste Art der Programmsteuerung besteht darin, starre Folgen von Eingaben, Rechenschritten und Ausgaben auszuführen (Abbildung 2.1). Sofern ein hinreichender Vorrat an Rechenoperationen vorgesehen ist, genügt bereits dieses einfache Schema, um viele nicht triviale Anwendungsaufgaben zu lösen.

Wirkliche Universalität ist dann gegeben, wenn:

1. die Reihenfolge der Verarbeitungsschritte in Abhängigkeit von den Verarbeitungsergebnissen abgewandelt werden kann (bedingte Verzweigung),

   und

2. die Reihenfolge der auszuführenden Verarbeitungsschritte nicht durch eine unveränderliche Ablaufsteuereinrichtung, sondern durch gespeicherte (und damit beliebig veränderbare oder auswechselbare) Steuerangaben bestimmt wird (speicherprogrammierbare Steuerung).

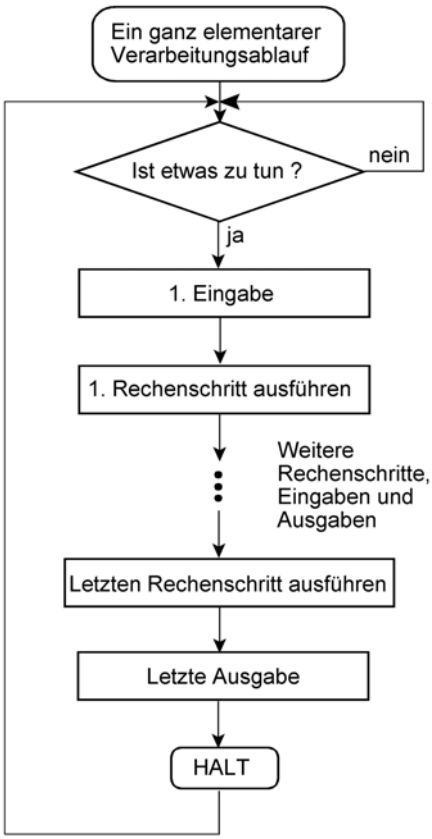


Abb. 2.1 Komplizierte Berechnungen ausführen – der grundsätzliche Ablauf.

Funktionseinheiten
Die grundsätzliche Struktur umfasst vier Funktionseinheiten (Abbildung 2.2):

1. Speicherwerk (Speichersubsystem),
2. Steuerwerk,
3. Rechen- oder Verarbeitungswerk,
4. Einrichtungen zur Ein- und Ausgabe (E-A-Einrichtungen).

Das Rechenwerk führt die Rechenoperationen aus. Das Steuerwerk bewirkt, dass die jeweils gewünschten Verarbeitungsschritte nacheinander ausgeführt werden. Das Speicherwerk speichert die zu verarbeitenden Daten (Operanden), die Ergebnisse und die Programme. Die Speicherung erfolgt in Speicherzellen, die über Speicheradressen angesprochen werden. Das Speichersubsystem kann aus verschiedenartigen Speichereinrichtungen aufgebaut sein, die unterschiedliche Speicherkapazitäten, Organisationsformen und Zugriffszeiten aufweisen (Speicherhierarchie). Die E-A-Einrichtungen stellen die Verbindung zur Außenwelt her. Der Verbund von Rechenwerk und Steuerwerk erbringt die eigentlichen Verarbeitungsleistungen. Deshalb wird er als eine einzige Funktionseinheit betrachtet und als Prozessor bezeichnet[8].

8: Andere Bezeichnungen: zentrale Verarbeitungseinheit oder Central Processing Unit (CPU). Das ist heutzutage typischerweise ein einziger Schaltkreis (Mikroprozessor).

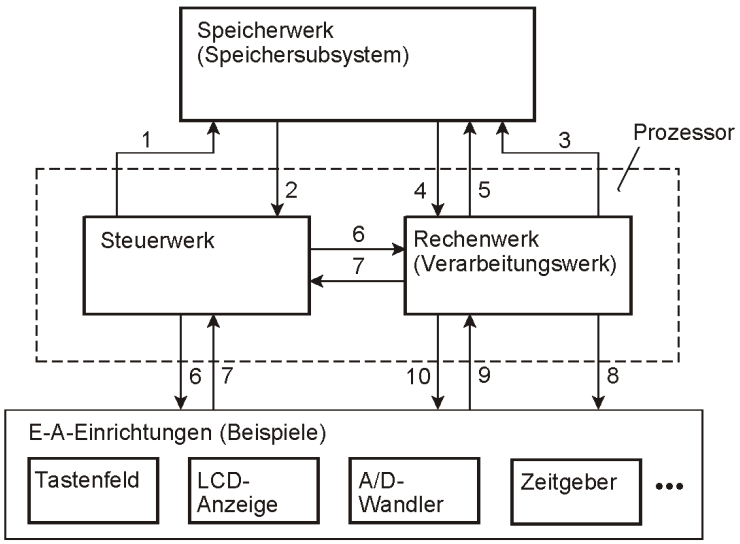


Abb. 2.2 Der Universalrechner im Blockschaltbild. 1 - Befehlsadresse; 2 - Lesen der Maschinenbefehle; 3 - Datenadresse; 4 - Lesen von Daten; 5 - Schreiben von Daten; 6 - Steuersignale; 7 - Bedingungs- und Zustandssignale; 8 - E-A-Adressierung; 9 - Eingabe von Daten; 10 - Ausgabe von Daten.

Maschinenbefehle

Die Programmsteuerung erfolgt durch gespeicherte Maschinenbefehle. Ein Maschinenbefehl (Abbildung 2.3) beschreibt, was zu tun ist (Operationscode) und womit es zu tun ist (Adressteil). Der Adressteil kann mehrere Adressangaben enthalten Zu einer Zeit wird jeweils ein Maschinenbefehl ausgeführt. Es gibt verschiedene Arten von Maschinenbefehlen:

- Operationsbefehle. Sie weisen Operation an, die vom Rechenwerk ausgeführt wird.
- Transportbefehle. Typische Transportvorgänge sind u. a. die Eingabe, die Ausgabe, das Holen des Inhalts einer Speicherzelle (Lesen) und das Ablegen von Daten in eine Speicherzelle (Schreiben).
- Verzweigungsbefehle. Sie beeinflussen die Reihenfolge der Befehlsausführung.
- Steuerbefehle. Sie üben Steuerwirkungen aus. Hierzu gehört unter anderem das Einstellen von Betriebsarten.

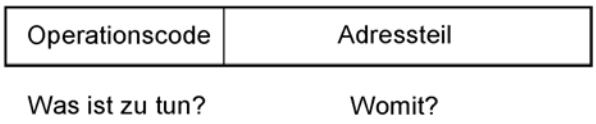


Abb. 2.3 Ein Maschinenbefehl. Er beschreibt, was zu tun ist und womit es zu tun ist. In den Einzelheiten der Formatgestaltung und Wirkungsweise gibt es erhebliche Unterschiede.

Die Befehlsliste (Befehlsvorrat, Instruction Set)

Die Befehlsliste ist das Verzeichnis aller Maschinenbefehle, die ein Prozessor ausführen kann.

Was leistet ein Maschinenbefehl?

Ein einzelner Maschinenbefehl kann nur sehr elementare Informationswandlungen oder Steuerungsabläufe veranlassen. Typische Befehlswirkungen sind z. B.:

- das Addieren zweier Binärzahlen,
- das Setzen, Löschen und Abfragen einzelner Bits,
- das Verzweigen zu einem bestimmten anderen Befehl im Programm,
- das Holen oder Abspeichern von einzelnen Binärzahlen, Maschinenworten usw.

Einfache Prozessoren haben um 30...50 verschiedene Maschinenbefehle, Hochleistungsprozessoren typischerweise einige hundert (150...300 sind üblich).

Einfache und komplexe Befehle – RISC und CISC
Ein Transportbefehl ist – seiner Wirkung nach – offensichtlich einfacher als ein Divisionsbefehl. Es ist eine entscheidende Grundsatzfrage beim Rechnerentwurf, ob man die Befehle auf einfache Wirkungen beschränkt (damit sie schneller ausgeführt werden können und die Maschine nicht zu kompliziert wird) oder ob man Befehle vorsieht, die vergleichsweise komplexe Funktionen auslösen (damit es die Programmierer leichter haben). Die pauschale Einteilung in einfache und komplexe Befehle wird durch zwei Marketingbegriffe ausgedrückt:

- CISC = Complex Instruction Set Computer. Die Befehlswirkungen sind vergleichsweise komplex. Manche Befehlswirkungen umfassen sowohl Transport- als auch Verarbeitungsfunktionen. Es gibt viele Befehle und Befehlsformate. Typische Beispiele: 8051, Z 80, 68k/Coldfire, 8086 und alle Weiterentwicklungen bis hin zu den Prozessoren der Personalcomputer.
- RISC = Reduced Instruction Set Computer. Die Befehlswirkungen sind vergleichsweise elementar. Transport- und Verarbeitungsfunktionen sind voneinander getrennt (Load/Store-Prinzip). Es gibt nur wenige Befehlsformate. Die meisten Befehle sind gleich lang. Typische Beispiele: PIC, AVR, V850, ARM, MIPS, SPARC, PowerPC.

Das Rechenwerk
Das Rechenwerk eines typischen universellen Prozessors besteht aus der Speicherzugriffseinheit und der Verarbeitungseinheit.

Die Speicherzugriffseinheit dient dazu, die Operanden aus dem Speicher zu holen und die Ergebnisse abzuspeichern. Ein einfache Speicherzugriffseinheit enthält ein Speicheradreßregister, wenigstens ein Speicherdatenregister (oder zwei, eines zum Lesen und eines zum Schreiben) sowie die zugehörigen Steuerschaltungen.

Die Verarbeitungseinheit dient dazu, die Informationswandlungen auszuführen, die von den Maschinenbefehlen angewiesen werden. Die Operanden und Ergebnisse sind Bits oder Bitketten (beispielsweise Bytes oder Maschinenwörter). Die einfachste Form der Informationswandlung ist die kombinatorische Verknüpfung. Typischerweise werden zwei Operanden miteinander verknüpft, um ein einziges Ergebnis zu bilden (vgl. die Grundrechenarten):

$$C := A \ \mathbf{op} \ B$$

Das Ergebnis kann durch zusätzliche Bedingungssignale (Flagbits) ergänzt werden. Weitere Abwandlungen:

- nur ein Operand,
- mehr als zwei Operanden,
- mehr als ein Ergebnis,
- keine Bedingungssignale, sondern nur Ergebnisse,

- keine Ergebnisse, sondern nur Bedingungssignale,
- nur eine einzige Operation (Einzweckschaltung),
- mehrere auswählbare Operationen,
- mehrere Operationen, die gleichzeitig ausgeführt werden,
- feste Formate (Wortlänge, Verarbeitungsbreite),
- wählbare oder einstellbare Formate.

Operanden und Ergebnis werden in Registern gehalten. Dazwischen liegen die konmbinatorischen Verknüpfungsschaltungen (Abbildung 2.4). Aus der Erfahrung heraus hat sich eine Menge von besonders zweckmäßigen elementaren Operationen ergeben, die in nahezu allen Prozessorarchitekturen vorgesehen sind. Je nachdem, ob die Operanden als Binärzahlen oder als Bitketten behandelt werden, unterscheidet man aritmetische und logische Operationen. Die elementaren Operationen des typischen Universalprozessors umfassen Datentransporte, bitweise Boolesche Verknüpfungen, Verschiebeoperationen und – als komplizierteste dieser Operationen – die Addition zweier Binärzahlen[9]; die Verarbeitungseinheit[10] ist gleichsam um den Binäraddierer (Adder) herumgebaut.

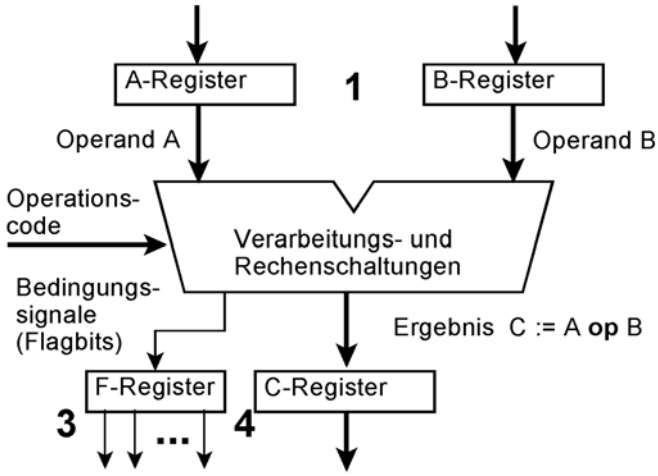


Abb. 2.4 Eine Verarbeitungseinheit. 1 - Operandenregister; 2 - Ergebnisregister; 3 - Bedingungsregister (Flagregister, Zustandsregister).

Das Steuerwerk

Das Steuerwerk hat die Aufgabe, die Befehle aus dem Speicher zu holen, die Befehlsbitmuster zu decodieren, Steuersignale an die anderen Einrichtungen zu lieferen und Zustandsmeldungen dieser Einrichtungen auszuwerten. Die üblichen Prozessoren lesen die Befehle nacheinander von aufeinander folgenden Speicheradressen. Die jeweilige Befehlsadresse wird durch einen Zählvorgang gewonnen (Befehlszähler, Instruction Counter IC). Ist eine Verzweigung auszuführen, wird der Befehlszähler mit der Verzweigungsadresse überladen. Der aus dem Speicher gelesene Befehl wird in ein Befehlsregister (Instruction Register IR) geladen, dem die Decodier- und Ablaufsteuerschaltungen nachgeordnet sind.

9: Die Maschinenoperationen können gar nicht allzu kompliziert sein, denn es muß möglich sein, eingermaßen kostengünstige Schaltungen zu bauen, um sie auszuführen.

10: Andere Bezeichnungen: Operationswerk, Arithmetik-Logik-Einheit, Arithmetic/Logic Unit (ALU).

Die Register

Register sind jene Speichermittel, die direkt mit den kombinatorischen Schaltungen verbunden sind. Sie bestehen aus Latches oder Flipflops (vgl. Digitaltechnik). In der Architektur – und damit beim Programmieren – sind nur jene Register von Bedeutung, die programmseitig zugänglich sind (Registermodell der Architektur). Man kann die Architektur auch ganz ohne Register auslegen (natürlich enthält die Hardware welche, nur merkt der Programmierer nichts davon[11]). Es hat sich aber bewährt, eine gewisse Anzahl an Registern dem Programmierer zugänglich zu machen. Sie dienen als Speicher mit sehr kurzer Zugriffszeit. Die Auslegung als sog. Universalregister (General Purpose Registers) hat sich als besonders zweckmäßig erwiesen. Jedes dieser Register kann wahlweise als Speicher für Variable und Ergebnisse oder als Adreßregister verwendet werden. Abbildung 2.5 zeigt das Blockschaltbild eines typischen Universalprozessors, der mit einem Universalregistersatz ausgerüstet ist.

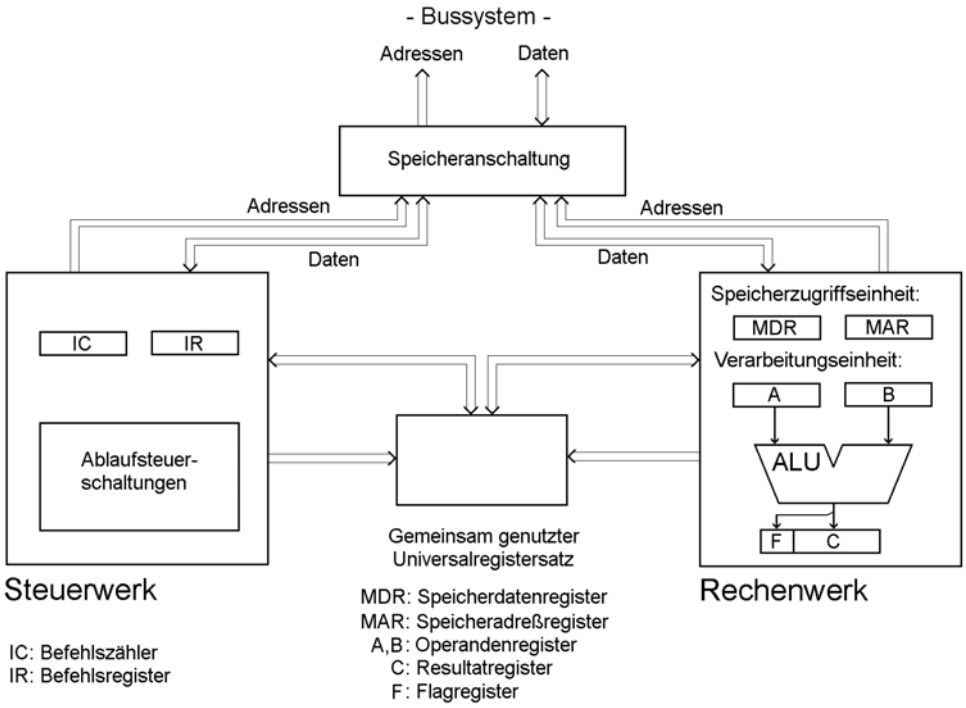


Abb. 2.5 Ein Universalprozessor.

Architektur

Dieser pauschale Allerwelts- und Allgemeinbegriff bezeichnet in der Informatik nichts anderes als die Ausgestaltung wohldefinierter Schnittstellen. Um eine Architektur zu nutzen, braucht man nur die Schnittstellendefinition zu kennen und kann alle Fragen des inneren Aufbaus vernachlässigen. So spricht man von der Architektur eines Computers, eines Prozessors, eines Netzwerks, eines Betriebssystems usw. Die Rechnerarchitektur (Computer Architecture) betrifft die Schnittstelle zwischen Hardware und Software; sie beschreibt Aufbau und Wirkungsweise des Computers aus der Sicht des Programmierers. Die wichtigsten Architekturmerkmale umfassen die Wortlänge, den Befehlsvorrat, den Registersatz und die Adressierungsvorkehrungen.

11: Man spricht dann davon, daß diese Register implizit genutzt werden.

Programmiermodelle
Das Programmiermodell[12] (Programming Model) ist das Erscheinungsbild der Architektur aus der Sicht des Programmierers. Es ergibt sich aus den Architekturmerkmalen und den grundsätzlichen Vorgaben zu deren Nutzung. Manche Architekturen gestatten es, zwischen verschiedenen Programmiermodellen zu wählen (typische Beispiele sind die ARM-Prozessoren und die Prozessoren der Personalcomputer). Viele Merkmale der Universalrechner haben sich aus den technischen Bedingungen der Anfangszeit heraus ergeben. Im Laufe der Entwicklungsgeschichte sind sie zu Industriestandards geworden, die man in allen anwendungspraktisch bedeutsamen Rechnerarchitekturen wiederfindet:

- die binäre Arbeitsweise,
- der adressierbare Speicher,
- die Binärzahl als elementare Datenstruktur,
- die Zweierkomplementarithmetik,
- bestimmte elementare Formate, z. B. Datenstrukturen von 4, 8, 16, 32 usw. Bits Länge,
- die Auslegung der elementaren (adressierbaren) Speicherzellen (Byte- oder Wortadressierung),
- die grundsätzlichen Befehlswirkungen.

Anwendungsprogrammschnittstelle (API)
Als Anwendungsprogrammschnittstelle oder API (Application Programming Interface) bezeichnet man Schnittstellen, die dazu dienen, andere Programme oder Funktionseinheiten der Hardware vom Anwendungsprogramm aus anzusprechen. Eine API ist nichts Körperliches, sondern eine Art Dienstvorschrift zum Aufrufen der jeweiligen Funktionen. Ein (stark vereinfachtes) Beispiel: Um ein Zeichen zu drucken, übertrage zunächst die laufende Nummer des gewählten Druckers, dann das Druckkommando und dann den Zeichencode. Ist das Zeichen gedruckt, kommt ein Zustandsbyte zurück. Enthält es den Wert Null, ist alles o.k. Ansonsten hat sich ein Fehler ergeben, zu dem der zurückgegebene Wert nähere Angaben enthält.

Maschinenwort und Verarbeitungsbreite
Das Maschinenwort ist eine Aneinanderreihung von Bitpositionen. Es ist im Grunde ein Behälter für typische Datenstrukturen, die von den Maschinenbefehlen angesprochen werden können. Die Wortlänge ist einer der ganz grundsätzlichen Architekturkennwerte. Das Wort ist die Datenstruktur, für die die jeweilige Architektur von Grund auf ausgelegt ist. Typische Wortlängen moderner Prozessoren betragen 8, 16, 32 und 64 Bits. In diesem Sinne spricht man von einer 8-Bit-Architektur, von einer 16-Bit-Architektur usw.

Die Verarbeitungsbreite wird in Bits angegeben. Der Kennwert bringt zum Ausdruck, wie lang eine elementare Datenstruktur ist, die im Prozessor auf einmal (mit allen Bitpositionen gleichzeitig in einem Taktzyklus) transportiert oder verarbeitet werden kann (Abbildung 2.4). Moderne universelle Prozessoren haben Verarbeitungsbreiten von 4, 8, 16, 32 oder 64 Bits. Spezialprozessoren und Beschleunigungszusätze haben bisweilen noch größere Verarbeitungsbreiten (z. B. 128 oder 256 Bits).

Die Wortlänge ist ein Kennwert der Architektur, die Verarbeitungsbreite ein Kennwert der Schaltung.

12: Ein Modell ist hier keine verkleinerte und vereinfachte Nachbildung. Das Wort ist vielmehr ein Allgemeinbegriff für Prinziplösungen, grundsätzliche Ausführungsformen usw., wobei tiefere Einzelheiten vernachlässigt werden.

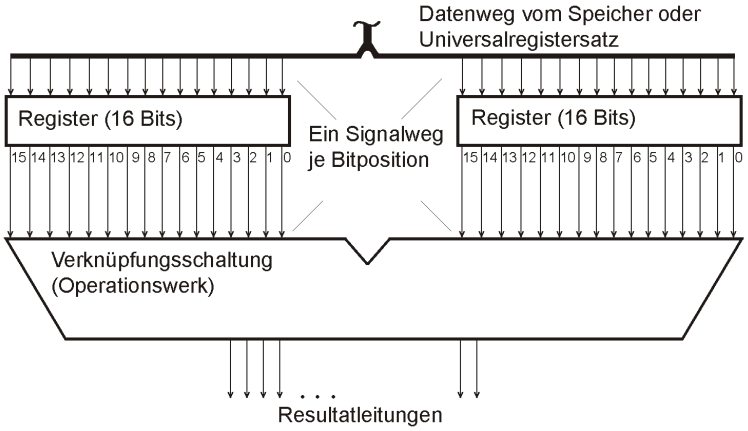


Abb. 2.6 Zur Erklärung der Verarbeitungsbreite.

Wortlänge = Verarbeitungsbreite?
Die Gleichsetzung liegt nahe. Sie ist auch der heutzutage typische Stand der Technik (eine 32-Bit-Architektur wird schaltungstechnisch mit einer Verarbeitungsbreite von 32 Bits implementiert usw.). Das ist aber nicht immer der Fall. Manchmal ist die Verarbeitungsbreite geringer als die Wortlänge. Dann sind mehrere Taktzyklen erforderlich, um ein Wort zu transportieren oder zwei Wörter miteinander zu verknüpfen. Ist die Verarbeitungsbreite größer, ist es möglich, mehrere Wörter gleichzeitig zu transportieren und zu verarbeiten (Parallelverarbeitung).

Maschinenwort und Befehlsformat
Wenn man eine Prozessorarchitektur entwirft, ist – in dieser Hinsicht – zweierlei festzulegen:

1. Der Funktionsumfang. Welche Operationen und Abläufe sollen von den Befehlen ausgelöst werden? Damit ordnet sich die Maschine in eine bestimmte Leistungsklasse ein.
2. Die Befehlsformate. Wie werden die Befehle binär codiert?

Jedes Befehlsformat ist ein Kompromiß. Nun kann man sich eine – auf den ersten Blick – kompromißlose Auslegung vorstellen: jeder Befehl ist so lang, wie es jeweils nötig ist, um alle Angaben (Adressen, Direktwerte usw.) unterzubringen, die zur Ausführung der jeweiligen Funktion benötigt werden (variable Befehlslänge). Es ist aber trotzdem ein Kompromiß, und zwar zugunsten des Programmierkomforts auf Kosten der Kompliziertheit (vor allem der Steuerung). In Mikrocontrollern kann man sich diese Kompliziertheit nicht leisten. Deshalb werden die Befehle fest formatiert. Typische Befehlslängen entsprechen einem halben Maschinenwort, einem Maschinenwort oder zwei Maschinenwörtern. Man bekommt aber nicht immer alles unter, was man zur Ausführung einer bestimmten Funktion benötigt. Solche Funktionen müssen dann mit mehreren Befehlen erledigt werden. Aus der Absicht, mit kurzen Befehlen auszukommen (Schaltungsaufwand, Speicherkapazität) ergeben sich mehr oder weniger ärgerliche Einschränkungen und der Zwang, manchmal ein simple Programmierabsicht mit mehreren Befehlen auf geradezu hanebüchen trickreiche Weise ausprogrammieren zu müssen[13].

Der Adreßraum
Dieser Begriff bezeichnet die Menge der jeweils adressierbaren Speicherzellen, E-A-Anschlüsse, Register usw. Man spricht dann sinngemäß vom Speicheradreßraum, E-A-Adreßraum usw. Eine

13: Auch der Atmel AVR ist von solchen Einschränkungen und dem Zwang zur Trickprogrammierung nicht frei. Also nicht ärgern, nur wundern...

Adreßraumangabe ist nichts im wörtlichen Sinne Räumliches, sondern eine bloße Zahl, die besagt, wieviele Elemente überhaupt einzeln adressierbar sind.

Einer Adresse von n Bits Länge entspricht ein Adreßraum von 2^n Elementen. Bei einer 20-Bit-Speicheradresse umfasst der Speicheradreßraum 2^{20} = 1 048 576 einzeln adressierbare Speicherzellen.

Adressierungsvermögen
Der Begriff entspricht im Grunde dem des Adreßraums, wird aber in einem etwas anderen Sinne verwendet: wird eine Adresse mit einer Länge von n Bits ausgelegt, so wird damit ihr Adressierungsvermögen (Addressing Capability) auf 2^n adressierbare Datenstrukturen eingeschränkt. Tabelle 2.1 gibt einen Überblick über typische Adreßlängen und deren Adressierungsvermögen.

Wozu immer mehr Adreßbits?
Grundsätzlich kann man mit jedem Prozessor – sofern er als wirklicher Universalrechner ausgelegt ist – jede Verarbeitungsbreite und jede Adressierungsweise nachbilden, und zwar mittels Software und (erforderlichenfalls) externer Zusatzbeschaltung. So könnte man z. B. eine 64-Bit-Architektur (64 Bits Verarbeitungsbreite, 64-Bit-Adressierung) mit einem 8-Bit-Prozessor realisieren – es würde funktionieren, wäre aber recht langsam. Der Übergang auf immer mehr Bits wird vor allem durch Forderungen seitens der Adressierung vorangetrieben. Dass man einzelne Bytes adressieren kann, gehört zum Stand der Technik. Lange Zeit war es weit mehr als ausreichend, jedes von rund vier Milliarden Bytes auswählen zu können (32-Bit-Adresse). So große Speicher konnte man über Jahrzehnte hinweg gar nicht bauen. Zwischenzeitlich sind aber auch die Datenmengen ins Gigantische gewachsen. Beispielsweise fallen bei der digitalen Speicherung eines Spielfilms zunächst mehrere Millionen Bytes je Sekunde an. Nun möchte man derartige Videodaten zunächst "roh" speichern und dann mit Software nachbearbeiten. Hierzu ist es aber notwendig, jedes Byte einzeln auswählen (= adressieren) zu können. Es lässt sich leicht überschlagen, dass z. B. ein Film von zwei Stunden Spieldauer mehr Bytes umfasst, als mit einer 32-Bit-Adresse auswählbar sind (beispielsweise ergeben 7200 s zu je vier Millionen Bytes rund 29 Milliarden Bytes). Aber auch die Weiterentwicklung der traditionellen Datenverarbeitung führt auf riesige Datenmengen – man möchte einfach alles speichern, was anfällt, um dann Stoff zum Auswerten zu haben (Data Warehousing, Data Mining). Ebenso muss ein Server, der eine gutsortierte Internet-Site bedient, mit derartigen Datenmassen umgehen können.

Adresslänge	Adressierbare Bytes	Anmerkungen
8 Bits	2^8 = 256	Meist zu wenig. Viele 8-Bit-Prozessoren haben 16-Bit-Adressierung
16 Bits	2^{16} = 65 536 = 64 k	Oftmals zu wenig. Viele 16-Bit-Prozessoren haben 20- oder 24-Bit-Adressierung
32 Bits	2^{32} = 4 294 967 296 4 G ≈ 4 Milliarden	Ein typischer Industriestandard
36 Bits	2^{36} = 68 719 476 736 = 64 G ≈ 68 Milliarden	Adresserweiterung verschiedener PC-Prozessoren
40 Bits	2^{40} =1 099 511 627 776 = 1 T ≈ 1 Billion = 1000 Milliarden	Technisches Adressierungsvermögen vieler 64-Bit-Typen
48 Bits	2^{48} = 281 474 976 710 656 = 256 T ≈ 280 Billionen	Eine heutzutage bei weitem ausreichende Grenze des Adressierungsvermögens

Adresslänge	Adressierbare Bytes	Anmerkungen
64 Bits	2^{64} = 18 446 7444 073 709 551 616 = 16 E ≈ 18 Trillionen = 18 Milliarden Milliarden	Der Industriestandard im obersten Leistungsbereich

(k = Kilo, M = Mega; T = Tera; E = Exa)

Tabelle 2.1 Adresslänge und Adressierungsvermögen.

Typische Verarbeitungsbreiten

1 Bit
Bitseriell arbeitende Maschinen sind in der heutigen Praxis bedeutungslos. 1-Bit-Prozessoren sind keine Universalrechner, sondern Spezialprozessoren für Steuerungsaufgaben (Bitprozessoren, Programmable Logic Controllers (PLCs)). Sie haben eigentlich nur historische Bedeutung als Vorläufer der modernen speicherprogrammierbaren Steuerungen (SPS). Die seinerzeit gefundenen Lösungen können aber als Anregungen dienen, wenn programmierbare Steuerschaltungen zu entwerfen sind.

4 Bits
Eine Verarbeitungsbreite von vier Bits kommt offensichtlich mit besonders geringen Aufwendungen aus. Diese Verarbeitungsbreite ist durchaus angemessen, wenn nur mit binär codierten Dezimalzahlen oder gar nicht zu rechnen ist (Steuerungsaufgaben usw.). 4-Bit-Maschinen sind ungeeignet, wenn umfangreichere Adressrechungen auszuführen sind. Sie eigenen sich nur für algorithmisch einfache Aufgaben. Typische Einsatzbereiche sind Zeitrelais, Schaltuhren, Haushaltgeräte, Spielzeuge und LCD-Anzeigen. Manche Hersteller haben 4-Bit-Schaltkreise für solche Einsatzfälle nach wie vor im Fertigungsprogramm[14]. Diese Verarbeitungsbreite kann auch in Eigenentwicklungen von Vorteil sein. Der Ressourcenbedarf (Datenwege usw.) ist gering, und es ist möglich, alle Verknüpfungen von zwei 4-Bit-Operanden mit Zuordnerspeichern zu erledigen, bis hin zu Additions- und Subtraktionstabellen und zum kleinen Einmaleins.

8 Bits
Die meisten der kleinen Mikrocontroller haben eine Verarbeitungsbreite von acht Bits. Für viele Anwendungen reicht das Leistungsvermögen vollauf aus – nicht selten auch bei naiver Hochsprachenprogrammierung, also ohne leistungssteigernde Tricks, maschinenspezifische Anpassungen im Quelltext und optimierende Compiler. Die Grenzen liegen vor allem im Adressierungsvermögen und in der Adressrechnung. Die meisten Architekturen sind auf 16 Adressbits beschränkt (64 kBytes). Eine Adressverlängerung ist manchmal vorgesehen. Deren Nutzung ist aber vergleichsweise mühevoll – man muss beim Programmieren schon mitdenken. Praxistipp: Wenn man einen größeren Adressraum braucht, ist es oftmals besser, eine von Grund auf entsprechend großzügig ausgelegte Architektur zu wählen. Dann muss man sich nicht mit Speicherbereichen, Bankregistern usw. abmühen.

16 Bits
Die Verarbeitungsbreite kommt vor allem dann zur Wirkung, wenn umfangreichere Berechnungen auszuführen sind. Acht Bits sind zur Darstellung von Messwerten usw. oftmals

14: Dass man nur noch die laufende Massenfertigung im Blick hat, ist u. a. daran zu erkennen, dass keine Starterkits und Entwicklungsumgebungen angeboten werden.

zu wenig, weil die Auflösung zu gering ist (rund 0,4 % vom Endwert). Viele Analog-Digital-Wandler haben 10 Bits und mehr (Auflösung bei 10 Bits rund 0,01 %, bei 14 Bits rund 0,006 % vom Endwert). Eine Verarbeitungsbreite von 16 Bits reicht aus, um solche Werte darzustellen. Sie genügt aber zumeist nicht als Speicheradresse. Die meisten 16-Bit-Architekturen haben längere Adressen. Praxistipp: Beim Auswählen von 16-Bit-Prozessoren vor allem auf die Vorkehrungen zur Adressverlängerung und Adressrechnung achten. In dieser Hinsicht kann es grundsätzliche Unterschiede geben, die sich bis in die Anwendungsprogrammierung und die Compiler hinein bemerkbar machen. Als Beispiele seien die Architekturen x86 und 68k genannt (x86: 20 Adressbits und Segmentierung; 68k: 24 Adressbits und linearer Adressraum).

32 Bits
Die Verarbeitungsbreite ist für die weitaus meisten Anwendungen vollauf ausreichend – auch für die Adressrechnung. Eine Adressverlängerung ist zumeist unnötig. Die meisten Mikrocontroller und Plattformen der oberen Leistungsbereiche beruhen auf RISC-Prozessorkernen mit 32 Bits Verarbeitungsbreite. Die 32-Bit-Schaltkreise haben im Grunde nur einen manchmal schwerwiegenden anwendungspraktischen Nachteil – nämlich extrem miniaturisierte Gehäuse mit vielen Anschlüssen, die auf herkömmliche Weise (Zweiebenenplatinen, Schwalllöten usw.) nicht zum Einsatz zu bringen sind. Deshalb ist man oftmals auf fertige Plattformen angewiesen.

64 Bits
Solche Prozessoren sind vor allem entwickelt worden, um das Adressierungsvermögen zu erweitern. Darüber hinaus ergibt sich aufgrund der Verarbeitungsbreite eine allgemeine Leistungssteigerung. Sie kommt vor allem dann zur Wirkung, wenn tatsächlich mit so langen Datenstrukturen gearbeitet wird. Sind die Datentypen kürzer (32-Bit-Binärzahlen, Bytes usw.) ergibt ein Verdoppeln der Verarbeitungsbreite aber keineswegs eine doppelt so hohe Verarbeitungsleistung.

Das Leistungsvermögen der Prozessoren
Der einzig wirklich wahre Leistungskennwert ist die tatsächliche Ausführungszeit für die jeweilige Anwendungsaufgabe. Demgemäß wäre das einzig brauchbare Bewertungsverfahren das Ausprobieren. Zumeist geht es aber darum, sich zu Beginn eines Entwicklungsvorhabens für eine bestimmte Grundsatzlösung zu entscheiden und die wichtigsten Bauelemente auszuwählen. Aufwendige Vorversuche oder gar mehrere Anläufe wird man sich kaum leisten können. Deshalb muss man sich an pauschalen Kennwerten orientieren.

Datenblattkennwerte
Datenblätter enthalten Geschwindigkeits- oder Zeitangaben für die einzelnen Abläufe. Von besonderer Bedeutung sind die Befehlsausführungszeiten und die Latenzzeiten der Unterbrechungsbehandlung. Typische pauschale Leistungskennwerte sind

- die Anzahl der Taktzyklen je Befehl (Cycles per Instruction, CPI),
- die Anzahl der Befehle, die gleichzeitig ausgeführt werden (Instruction Level Parallelism, ILP),
- die Anzahl der Befehle, die in einem Taktzyklus ausgeführt werden (Instructions per Cycle, IPC),
- die Anzahl der ausgeführten Befehle in der Sekunde (Instructions per Second, IPS). Dieser Wert ist die Verarbeitungsleistung. Mit entsprechenden Vorsätzen ergeben sich die in Marketingkreisen so beliebten MIPS und FLOPS.

Die meisten Prozessoren sind Einzelprozessoren. Sie können zu einer Zeit nur einen Befehl ausführen (ILP = 1). Leistungsfähigere Typen haben mehrere Verarbeitungswerke (Superskalarprozessoren), um mehrere Befehle gleichzeitig auszuführen (ILP > 1). CPI = 2 und ILP = 4 bedeuten, dass zur Ausführung eines Befehls zwei Taktzyklen erforderlich sind und dass vier Befehle gleichzeitig ausgeführt werden. Aus CPI und ILP kann man die Anzahl der Befehle je Taktzyklus als fiktive Größe IPC errechnen:

$$IPC = \frac{ILP}{CPI} \tag{1.1}$$

Zunächst soll angenommen werden, diese Werte seien konstant (alle Befehle erfordern gleich viele Takte, und es werden immer gleich viele Befehle parallel ausgeführt).

Die Programmausführung
Während der Ausführungszeit t_{EX} eines Programmablaufs werden in n_C Taktzyklen mit der Taktzykluszeit t_C n_I Befehle ausgeführt. Jeder Befehl hat eine Ausführungszeit t_I. Die Ausführungszeit des Programms ergibt sich dann zu:

$$t_{EX} = n_C \cdot t_C = \frac{n_I}{ILP} \cdot t_I \tag{1.2}$$

Die Ausführungszeit eines Befehls t_I ergibt sich aus der Anzahl der Taktzyklen je Befehl CPI und der Taktzykluszeit t_C:

$$t_I = CPI \cdot t_C \tag{1.3}$$

Damit wird

$$t_{EX} = n_I \cdot t_C \cdot \frac{CPI}{ILP} = \frac{n_I \cdot t_C}{IPC} \tag{1.4}$$

Hieraus ergibt sich die Anzahl der Befehle je Taktzyklus IPC als Verhältnis der Anzahl der ausgeführten Befehle n_I zur Anzahl der Taktzyklen n_C:

$$IPC = \frac{n_I \cdot t_C}{t_{EX}} = \frac{n_I \cdot t_C}{n_C \cdot t_C} = \frac{n_I}{n_C} = t_C \cdot \frac{n_I}{t_{EX}} \tag{1.5}$$

Die Ausführungszeit t_{EX} ist zumeist eine Entwicklungsvorgabe, die sich aus den Realzeitanforderungen ergibt. Aus dem Programm ist ersichtlich, wieviele Befehle auszuführen sind (Anzahl n_I). Ist die Taktfrequenz vorgegeben, kann man mit (1.5) ausrechnen, wieviele Befehle je Taktzyklus ausgeführt werden müssen. Kommt man mit einem Einzelprozessor aus oder braucht man eine Superskalarmaschine? IPC > 1 bedeutet, dass in jedem Taktzyklus mehrere Befehle auszuführen sind. Aus (1.1) ergibt sich, wieviele das sind:

$$ILP = IPC \cdot CPI \tag{1.6}$$

Ist IPC < 1, kann es sein, dass der Einzelprozessor genügt. Das ist dann der Fall, wenn ILP gemäß (1.6) kleiner als Eins ist.

Ergibt sich ein ILP > 1, ist nachzusehen, ob ein passender Superskalarprozessor verfügbar ist. Wenn der zur Wahl stehende Prozessor die Ausführungszeit t_{EX} einhalten kann, ist es in Ordnung. Ansonsten könnte ein Ausweg darin bestehen, t_C zu verkürzen, also die Taktfrequenz zu erhöhen. Ist das nicht möglich, könnte man probieren, ob eine andere Prozessorarchitektur besser geeignet ist. Eine weitere Alternative könnte darin bestehen, einen anwendungsspezifischen Prozessor mit genügend vielen Verarbeitungswerken selbst zu entwerfen. In der Praxis dürfte das zumeist auf einen Spezialprozessor hinauslaufen, der mehrere Verarbeitungswerke enthält, aber in seinen Wirkprinzipien nicht allzu kompliziert ist.

Die Verarbeitungsleistung
Dauert die Ausführung eines Befehls CPI Taktzyklen, werden ILP Befehle gleichzeitig ausgeführt und beträgt die Taktfrequenz f_c Hz, so ergibt sich die Verarbeitungsleistung P_I in Befehlen/Sekunde folgendermaßen:

$$IPS = f_c \cdot \frac{ILP}{CPI} = f_C \cdot IPC \qquad (1.7)$$

Die Verarbeitungsleistung kann auch aus der Ausführungszeit t_{EX} eines Programms und der Anzahl n_I der ausgeführten Befehle errechnet werden:

$$IPS = \frac{n_I}{t_{EX}} \qquad (1.8)$$

Der herkömmliche Einzelprozessor führt zu einer Zeit jeweils einen Befehl aus, der nur eine einzige Operation anweisen kann. Somit ist ILP = 1 und es gilt

$$IPS = \frac{f_C}{CPI} \qquad (1.9)$$

CPI und ILP in der Praxis
Diese Werte als konstant anzusehen ist nur eine grobe Näherung. Der herkömmliche Einzelprozessor benötigt für jeden Befehl mindestens einen Taktzyklus. Es können aber auch mehrere sein. Das hängt von der Befehlswirkung und von verschiedenen Nebenumständen ab (beispielsweise davon, ob sich die zu verarbeitende Daten im Cache befinden oder ob bei Speicherzugriffen Wartezustände auftreten). Das gilt sinngemäß für die Parallelarbeit in Superskalarprozessoren. Typisch sind zwei bis acht Befehle je Taktzyklus. Aber auch dieser Wert ist nicht konstant. Er hängt unter anderem davon ab, welche Befehle aufeinander folgen und auf welche Daten sie zugreifen. Für genauere Rechnungen müssen deshalb Erwartungswerte angesetzt werden.

Einfache Prozessoren
Ist der Prozessor einfach, sind die Verhältnisse überschaubar. Man ist bestrebt, in jedem Taktzyklus einen Befehl auszuführen. Die Marketingangabe, die kennzeichnet, dass man dieses Ziel näherungsweise erreicht hat, lautet "1 MIPS pro MHz". Sie drückt nichts anderes aus als den hier in Rede stehenden Sachverhalt (ein Befehl je Taktzyklus), klingt aber viel großartiger...

Diese auf die Taktfrequenz f_C bezogen Prozessorleistung P_C ergibt sich wie folgt:

$$P_C = \frac{IPS}{f_C} = IPS \cdot t_C \cdot \frac{1}{MHz} \qquad (1.10)$$

Die in einfachen Prozessoren erreichbaren Befehlsausführungszeiten hängen von der Taktzykluszeit ab, die sich letzten Endes aus der Schaltungstruktur und der Schaltkreistechnologie ergibt. Man kann – den RISC-Grundsätzen (s. weiter unten) gemäß – die Befehlswirkungen auf jene Operationen beschränken, die durch kombinatorische Zuordnung zu erledigen sind. Dann erfordert jeder Befehl nur einen einzigen Taktzyklus. Die Taktfrequenz f_C in Hz entspricht dann der Verarbeitungsleistung in Befehlen je Sekunde:

$$IPS = \frac{1}{t_C} \cdot 1\,Befehl = f_c\ Befehle / s \qquad (1.11)$$

Mehr geht nicht. Zur Leistungssteigerung verbleiben zwei Wege:

- Befehlspipelining. Das Holen der Befehle, die Befehlsdecodierung usw. wird mit der eigentlichen Operationsausführung überlappt, so dass sich mehrere Befehle in verschiedenen Phasen der Befehlsausführung befinden. Die Taktperiode t_c ergibt sich dann aus der Verzögerungszeit des Operationswerks.
- Parallelausführung. Es werden – auf welche Weise auch immer – mehrere Befehle gleichzeitig ausgeführt (Superskalarprinzip).

Diese Maßnahmen wirken aber nicht ständig, sondern nur mit einer gewissen Wahrscheinlichkeit oder Trefferrate. Es ist nicht immer möglich, eine bestimmte Anzahl an Befehlen gleichzeitig auszuführen. Beim Befehlspipelining müssen auch die Speicherschnittstellen mit der kurzen Taktperiode zurecht kommen, sonst gibt es Wartezustände. Die Abhilfe sind Caches, Befehlspuffer usw. Sprungbefehle und Datenabhängigkeiten führen dazu, dass der lückenlose Befehlsstrom zeitweise aufgehalten wird. Je komplexer der Prozessor, desto schwieriger ist es, die Dauer der Befehlsausführung zu bestimmen. Nicht alle Befehle benötigen gleich viele Taktzyklen. Bei einigen Befehle kann sie von den Werten zu verarbeitenden Daten abhängen (z. B. bei der Multiplikation und Division), bei manchen von den Adressen. Die Belegung der Caches, Sprungzielpuffer, Speicherschnittstellen usw. wirkt sich auf die Dauer der Befehlsausführung aus. Im Datenmaterial sind manchmal genauere Angaben zu finden. Da meist nicht bekannt ist, wann welche Betriebszustände vorliegen, kann man sie nur auf Grundlage von Wahrscheinlichkeitsschätzungen auswerten.

Taktzykluszeit und Schaltungstiefe

Ein Verarbeitungswerk ist ein Schaltnetz, das ein- und ausgangsseitig mit Registern (Flipflops oder Latches) verbunden ist (vgl. Abbildung 2.4). Die minimale Taktzykluszeit t_C, mit der man eine solche Funktionseinheit betreiben kann, hängt von der Schaltungstiefe s und von Zeitkennwerten ab, die sich aus der Schaltkreistechnologie ergeben:

$$t_C = s \cdot t_{p\_g} + t_{p\_d} + t_{prop} + t_{su} + t_{skew} \qquad (1.12)$$

t_{p_g}: Gatterverzögerungszeit; t_{p_d}: Datenverzögerungszeit der Flipflops; t_{prop}: Signallaufzeit über die Signalwege; t_{su}: Vorhaltezeit der Flipflops; t_{skew}: Takttoleranz.

Ist die Schaltkreistechnologie vorgegeben, kann man den Taktzyklus nur kurz halten, indem man die Schaltungstiefe auf den Wert beschränkt, der aus (1.12) berechnet werden kann. Es bleibt dann die Frage, ob man mit sehr geringen Schaltungstiefen auch Verknüpfungen implementieren kann, die anwendungsseitig brauchbar sind. Im Universalprozessor ist es üblicherweise die binäre Addition über die gesamte Verarbeitungsbreite, die die Schaltungstiefe s bestimmt; damit ergibt sich die maximale Taktfrequenz gemäß (1.12) letzten Endes aus dem Zeitbedarf der Addition.

RISC – die entscheidende Überlegung
Die elementaren Befehle (Laden, Speichern, Addieren, Vergleichen, Verzweigen usw.) werden am häufigsten genutzt. Solche Befehle sind beispielsweise mit einer Schaltungstiefe s = 10 zu implementieren. Damit lässt sich eine entsprechend kurze Zykluszeit t_C erreichen. Wird durch Hinzufügen komplexerer Befehle die Schaltungstiefe auch nur um 1 erhöht, so werden *alle* Abläufe um 10 % langsamer. Die Nutzungshäufigkeit und Leistungsfähigkeit der hinzugefügten Befehle müsste also so groß sein, dass der Leistungsverlust von 10 % mehr als aufgewogen wird.

Die Ziele der "klassischen" (sprich: akademischen) RISC-Entwicklung:

- Der Prozessor soll in jedem Taktzyklus einen Befehl ausführen können (1 MIPS/MHz). Damit werden nur Befehlswirkungen zugelassen, die mit kombinatorischen Zuordnungen implementiert werden können bestehen.
- Die Befehlswirkungen sollen so einfach sein, dass man den Schaltkreis mit möglichst vielen MHz betreiben kann (einfache Befehlswirkungen – einfache Schaltungsstrukturen – geringe Schaltungstiefe).
- Alles, was komplizierter ist, soll die Software erledigen (Compiler + Laufzeitsystem).

In der Praxis sieht man es aber nicht so verbissen (Abschnitt 3.7).

MIPS und FLOPS
Setzt man die Taktfrequenz in MHz an, so hat die Leistungsangabe IPS die Maßeinheit Millionen Befehle/s oder MIPS (Millions of Instructions per Second). Handelt es sich um Operationen mit Gleitkommazahlen (Floating Point Operations), werden die MIPS zu FLOPS (MFLOPS, wenn Millionen, GFLOPS, wenn Milliarden und TFLOPS, wenn Billionen solcher Operationen in einer Sekunde ausgeführt werden).

Es sind typische Hausnummern
.. um es vorsichtig auszudrücken[15]. Diese Angaben betreffen nämlich zumeist die lückenlos aufeinanderfolgende Abarbeitung der jeweiligen Befehle unter den jeweils günstigsten Bedingungen. Die gigantischsten Werte stammen von Multiprozessorsystemen, die hunderte, manchmal mehrere tausend Prozessoren enthalten. Die vielen TFLOPS ergeben sich nach folgender Milchmädchenrechnung:

Gesamtleistung des Systems =
Anzahl der Prozessoren • Maximalleistung des einzelnen Prozessors (1.14)

Viele Leistungsangaben beruhen auf dieser einfachen Formel. In der Praxis werden aber nicht selten nur 10 bis 15 % der theoretischen Maximalleistung (Peak Performance) wirksam.

15: Eine seit längerem in Fachkreisen verbreitete Interpretation: MIPS = Misinformation to Promote Sales ...

Will man verschiedene Architekturen miteinander vergleichen, so ist zu bedenken, dass Befehle in unterschiedlichen Architekturen Unterschiedliches leisten, selbst dann, wenn sie die gleiche Operation auf gleiche Datentypen anwenden (beispielsweise 32-Bit-Binärzahlen zueinander addieren). Sie können sich in den Adressierungsweisen, in der Anzahl der Register, die im Befehl angesprochen werden können, in der Reaktion auf Bedingungen usw. unterscheiden.

Latenz- und Reaktionszeiten
Beide Zeitangaben 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 erstenReaktion darauf, die sich in der Außenwelt bemerkbar macht (Abbildung 2.7).

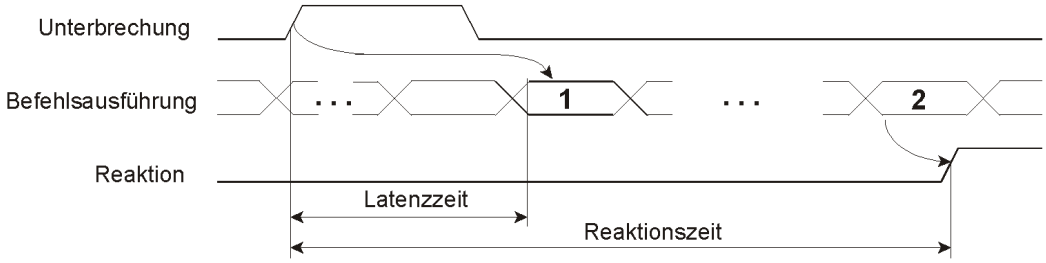


Abb. 2.7 Latenz- und Reaktionszeit am Beispiel einer Unterbrechungsbehandlung. 1 - der erste Befehl der Unterbrechungsbehandlung; 2 - dieser Befehl veranlasst die erste Reaktion in der Außenwelt, beispielsweise das Einschalten einer Leuchtanzeige.

Manche Latenzzeiten sind schaltungstechnisch bedingt und im Datenmaterial des Prozessors angegeben. Das betrifft unter anderem die Zeit zwischen der Unterbrechungsauslösung und dem Beginn der Unterbrechungsbehandlung. Diesen Werten liegen aber immer die günstigsten Betriebsverhältnisse zugrunde (Beispiel: die Unterbrechung ist zugelassen, es sind keine Unterbrechungen höherer Priorität anhängig, der vorhergehende Befehl hat die kürzeste Ausführungszeit, und es gibt keine Wartezustände). In der Praxis sind aber stets die jeweils ungünstigsten Verhältnisse anzusetzen. Und da können aus wenigen hundert Nansosekunden 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.) Das typische Kennzeichen einer guten Realzeitumgebung sind eindeutig voraussehbare Latenz- und Reaktionszeiten. Systemplattformen aus dem Bereich der Personalcomputer können diese Bedingung prinzpbedingt nicht erfüllen, da sie eigentlich für ganz andere Anwendungsgebiete vorgesehen sind, in denen naturgemäß andere Anforderungen erfüllt werden müssen. Präzise vorhersagbare Latenz- und Reaktionszeiten ergeben sich, wenn man alle störenden Einflüsse fernhält (hierzu gehören unter anderem Versuche, auf der selben Maschine weitere Anwendungen laufen zu lassen oder von den Realzeitprogrammen aus direkt aufs Internet zuzugreifen). In letzter Konsequenz führt dies zur Funktionentrennung zwischen Industriestandard- und Realzeitmaschinen, wobei die harten Realzeitaufgaben mit Stand-alone-Programmen oder gar mit selbst entwickelten Schaltungslösungen erledigt werden.

Rechnerarchitekturen auswählen
Die heutigen Prozessorarchitekturen kann man als Abmischungen aus vergleichsweise wenigen Grundsatzlösungen ansehen. Wer sich mit einer dieser Architekturen auskennt, wird kaum Schwierigkeiten haben, sich in eine andere einzuarbeiten. Die elementaren Datenstrukturen und

Befehlswirkungen sind im Grunde überall gleich. Die Unterschiede zwischen den Architekturen betreffen vor allem:

- die Wortlänge und Verarbeitungsbreite,
- das Adressierungsvermögen,
- das Registermodell,
- die Zubringerfunktionen,
- die Speicherorganisation,
- die Speicher- und E-A-Ausstattung.

Praxistips zur Architekturauswahl:

1. Die anwendungspraktisch wesentlichen Unterschiede zwischen den einzelnen Architekturen sind viel geringer, als man aufgrund der einschlägigen Werbeaussagen vermuten könnte.
2. Die Auslegung der Maschinenbefehle (ob CISC, ob RISC) ist in der Anwendungspraxis nur von geringer Bedeutung.
3. Oft kommt es gar nicht auf die Architektur an, sondern auf Kosten, E-A- und Speicherausstattung, Taktfrequenz, Gehäusebauform, Speisespannung und Kompatibilität.
4. Alle Architekturen und Befehlssätze sind "powerful" – manche mehr, manche weniger...
5. Es gibt keine ideale Architektur – ja nicht einmal ein allgemeingültiges Optimum. Jede Auslegung ist ein Kompromiss.
6. Man gewöhnt sich an alles.

Rechnerarchitektur und Hochsprachenprogrammierung
Die Programmiersprache sollte eigentlich den Anwendungsprogrammierer von der Maschinenarchitektur unabhängig machen. Wenn das Realzeitraster keine allzu hohen Anforderungen stellt – und somit auf maschinennahe Optimierungen verzichtet werden kann – ist das auch weitgehend der Fall. Beim Auswählen kommt es dann nicht auf Spitzfindigkeiten der Architektur an. Neben den Ausstattungsmerkmalen ist vor allem das pauschale Leistungsvermögen von Bedeutung (damit es schnell genug läuft). Hierzu genügt oft ein kurzer Blick auf Verarbeitungsbreite und Verarbeitungsleistung oder Taktfrequenz, manchmal auch auf den Registersatz. Die ganz grobe Faustregel: Je mehr von all dem, desto besser. Die bessere Faustregel: so viel, dass man nicht von Anfang an zum Tricksen gezwungen ist und dass genügend Reserven für Änderungen und künftige Erweiterungen da sind. Erfahrungsbericht: ausreichende Leistungsreserven ergeben sich von 4 MIPS an aufwärts, bei näherungsweise bei 1 MIPS/MHz also von 4 MHz Taktfrequenz an. Typische nichttriviale Anwendungen, die in Sprachen wie C oder IEC 61131 ST programmiert wurden, laufen dann schnell genug.

Rechenmaschinen und Realzeitmaschinen
Real- oder Echtzeitverarbeitung (Real Time Processing) heißt, bestimmte Verarbeitungsleistungen in einem vorgegebenen Zeitraster zu erbringen. In letzter Konsequenz müsste man einen Zustandsautomaten bauen, der alle Ausgaben auf den Takt genau erledigt. Das ist aber nicht die Auslegung der üblichen Prozessoren und Computersysteme. Hier geht man auf die höchste Leistung, die sich im Rahmen der jeweiligen Aufwendungen erreichen lässt. Man gibt sich damit zufrieden, dass die Maßnahmen der Leistungssteigerung nur unter jeweils günstigen Bedingung wirksam werden. Ein heutiger Hochleistungsprozessor ist nur dann wirklich schnell, wenn das jeweils auszuführende Programmstück im Cache steht, wenn sich alle zugehörigen Variablen in den Universalregistern befinden, wenn sich aufeinanderfolgende Maschinenbefehle auf Daten beziehen, die voneinander unabhängig sind usw. Treffen solche Bedingungen nicht zu, wird die Maschine zeitweise langsamer. Am PC fällt es kaum auf, wenn ein Spiel einige Millisekunden

lang nicht von der Stelle kommt, wenn ein Datenbankzugriff gelegentlich etwas länger dauert usw. – wichtig ist allein, dass sich der Leistungsabfall nicht ständig, sondern nur ab und zu bemerkbar macht. Realzeitmaschinen im strengen Sinne des Wortes kann man aber nicht so auslegen. Motorsteuerungen, Antiblockiersysteme, Autopiloten usw. müssen unter allen Umständen im jeweiligen Zeitrahmen funktionieren – es darf keineswegs sein, dass der Treibstoff später eingespritzt wird, nur weil sich ein Unterprogramm gerade mal nicht im Cache befindet.

Wichtig ist, dass der Entwickler das Problem überhaupt kennt. Es hilft keineswegs immer, nach den Prozessoren der obersten Leistungsklassen zu greifen – womöglich in der Hoffnung, durch GHz, große Caches und schnelle Speicherschnittstellen den Schwierigkeiten von vornherein aus dem Wege zu gehen. In vielen Einsatzfällen haben auch einfachere Prozessoren genügend Leistungsüberschuss[16]. Dann können die Realzeitanforderungen auch unter den ungünstigsten Bedingungen eingehalten werden. Kommt es wirklich darauf an, so erledigt man die harten Realzeitaufgaben am besten mit Prozessoren, die ein überschaubares Zeitverhalten aufweisen. Dieser Ansatz führt oftmals zu einer Arbeitsteilung, also zu einem heterogenen System. Für Aufgaben, die hohe Anforderungen an die Rechenleistung stellen, bei denen es aber zeitweilig nicht auf ein paar Millisekunden ankommt, wird beispielsweise eine PC-Plattform vorgesehen, für die harten Realzeitaufgaben hingegen ein RISC-Prozessor, ein Signalprozessor oder eine anwendungsspezifische Eigenentwicklung.

2.2 Speichersubsysteme

Das Speichersubsystem hat die Aufgabe, Programme und Daten so zu speichern, dass sie (1) für den Prozessor zugänglich sind und dass sie (2) auf Dauer – auch in ausgeschaltetem Zustand – erhalten bleiben. Beim herkömmlichenUniversalrechner obliegt die erste Aufgabe dem Arbeitsspeicher (auch System- oder Hauptspeicher), der zweite den Massenspeichern (Abbildung 2.8). In Mikrocontrollern wird die zweite Aufgabe vor allem durch den Einsatz von Festwertspeichern gelöst.

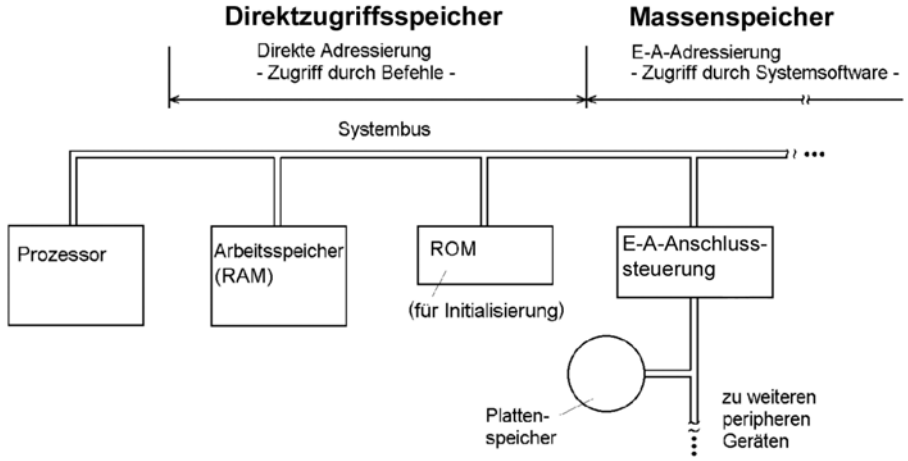


Abb. 2.8 Ein typisches Speichersubsystem mit Direktzugriffs- und Massenspeichern.

Das Speichersubsystem aus Sicht der Rechnerarchitektur
Die Schnittstelle zwischen Hard- und Software kennt keine Speicherschaltkreise, Bussysteme usw., sondern nur Speicheradressräume und Zugriffsfunktionen.

16: Das gilt nicht selten auch für die kleinen Mikrocontroller.

Direktzugriffsspeicher
Direktzugriffsspeicher werden vom Prozessor aus adressiert. Die Bits werden in Speicherzellen gehalten, die in Speichermatrizen angeordnet sind. Die jeweiligen Speicherzellen werden durch Decodieren der Adresse unmittelbar ausgewählt (wahlfreie Zugriffe).

Massenspeicher
Zu den herkömmlichen Massenspeichern gehören die Festplatten und die optischen Laufwerke (CD, DVD). Der typische Massenspeicher der Mikrocontroller ist der Flash-ROM in NAND-Technologie, beispielsweise in Form von Speicherkarten. Die Bits werden in einem Speichermedium oder in sequentiell zugänglichen Speicherzellen gehalten. Aus Sicht der Software werden die Daten jedoch ebenso adressiert wie die Bytes oder Wörter der Direktzugriffsspeicher (Linear Block Addressing LBA). Man könnte eine Festplatte oder eine Speicherkarte durchaus wie einen besonders großen RAM betreiben. Die Zugriffsabläufe sind aber viel komplizierter als die der Direktzugriffsspeicher, weil die Massenspeicher nur über komplexe standardisierte Schnittstellen zugänglich sind (beispielsweise USB, ATA oder SCSI). In der Praxis werden die gespeicherten Daten als Dateien (Files) organisiert. Hierbei bevorzugt man bewährte Dateisysteme. Die Steuerung der Massenspeicher ist somit eine Angelegenheit der Systemsoftware.

Caches und virtuelle Speicher
Ein Cache ist ein RAM, der zwischen dem eigentlichen Arbeitsspeicher und dem Prozessor angeordnet ist, um Speicherzugriffe zu beschleunigen. Er hat eine vergleichsweise geringe Speicherkapazität, aber eine sehr kurze Zugriffszeit, so dass der Prozessor Daten und Befehle aus dem Cache viel schneller holen kann als aus dem Arbeitsspeicher. Im Cache werden Teile des Arbeitsspeicherinhalts bereitgehalten.Sie sind dort unter den jeweiligen Arbeitsspeicheradressen zugänglich. Der Cache hat keinen eigenen Adressbereich. Die Anordnung aus dem Cache und dem eigentlichen Arbeitsspeicher soll sich aus der Sicht des Prozessors so verhalten wie ein einziger Speicher mit der (geringen) Zugriffszeit des Caches und mit der (großen) Speicherkapazität des Arbeitsspeichers. Manche Cache-Subsysteme bestehen aus mehreren Ebenen mit unterschiedlich langen Zugriffszeiten. Die Caches der ersten Ebene (L1 = Level 1) haben die kürzeste Zugriffszeit. Sie sind direkt an den Prozessorkern angeschlossen.

Ein virtueller Speicher soll dem Anwendungsprogramm eine große Speicherkapazität zu erträglichen Kosten bereit stellen. Virtuell = scheinbar. Es ist aber offensichtlich, dass es total scheinbar nicht geht, sondern dass die zu unterstützende Speicherkapazität irgendwo vorhanden sein muss. Die Grundsatzlösung besteht darin, Arbeitsspeicher (teuer) und Massenspeicher – vor allem Festplatten – (kostengünstig) im Verbund zu betreiben.

Beide Prinzipen haben das Ziel, der Software gegenüber ein nahezu ideales Speichersubsystem darzustellen: die Speicherkapazität entspricht dem maximalen Adressierungsvermögen, die Zugriffszeit ist nicht länger als ein Maschinentakt, und der Anwendungsprogrammierer muss sich überhaupt nicht darum kümmern. Caches werden üblicherweise von der Hardware allein verwaltet, virtuelle Speicher von einem Verbund aus Hardware und Systemsoftware. Die wichtigste Verwaltungsaufgabe besteht darin, nachzusehen, ob der jeweils adressierte Speicherinhalt im Cache oder im Arbeitsspeicher verfügbar ist, und andernfalls Zugriffe zum jeweils nachgeordneten Speicher auszuführen. Da von Zeit zu Zeit solche Zugriffe erforderlich sind, ist das Realzeitverhalten nicht vorhersagbar.

2.3 Ein- und Ausgabe

Die technischen Mittel, die dazu dienen, den Universalrechner mit der Außenwelt zu verbinden, werden als Ein- und Ausgabeeinrichtungen (E-A-Einrichtungen, I/O Devices) bezeichnet. Die einfachsten E-A-Abläufe beruhen auf Maschinenbefehlen, die Daten in E-A-Einrichtungen transportieren (Ausgabebefehle) oder aus solchen Einrichtungen abholen (Eingabebefehle). Grundsätzlich kann man folgende Arten von E-A-Einrichtungen unterscheiden:

- Universelle E-A-Anschlüsse, die zu E-A-Ports zusammengefasst sind. Die E-A-Ports der typischen Mikrocontroller sind so ausgelegt, dass jeder einzelne Anschluss wahlweise als Eingang oder als Ausgang betrieben werden kann.
- Periphere Einrichtungen für typische Aufgaben der Zeitdarstellung, Impulserzeugung, Ablaufsteuerung, Signalwandlung usw.
- Anwendungsspezifische periphere Einrichtungen.
- Besonders kostengünstige universelle Erweiterungsschnittstellen. Hierzu gehören unter anderem die serielle Schnittstelle, SPI, der I^2C-Bus und der CAN-Bus.
- Besonders leistungsfähigeuniverselle Erweiterungsschnittstellen, wie beispielsweise PCI, USB oder Ethernet.

Mikrocontroller und Hochleistungsprozessoren
Aufgrund der unterschiedlichen Einsatzgebiete ergibt sich eine geradezu entgegengesetzte Auslegung der E-A-Subsysteme von Mikrocontrollern und Hochleistungsprozessoren.

Mikrocontroller
Die Ein- und Ausgabe ist die Hauptsache. Mikrocontroller sind eigens entwickelt worden, um angeschlossene Einrichtungen auf möglichst kostengünstige Weise zu steuern. Die meisten Anwendungen sind eigentlich E-A-Programme. Es ist wichtig, die E-A-Einrichtungen von den Maschinenbefehlen aus freizügig ansprechen zu können. Die kürzeste Zeiteinheit der Ein- und Ausgabe ist der Maschinentakt. Die größtmögliche Flexibilität ist dann gegeben, wenn alle Anschlüsse programmseitig zugänglich sind (Abbildung 2.9). Die elementare Mikrocontrollerschnittstelle ist somit der bitweise frei programmierbare E-A-Port. Die weitere Ausstattung mit Zählern und Zeitgebern, Wandlern, Schnittstellen usw. richtet sich nach den Bedürfnissen jener Anwendungsgebiete, in denen die höchsten Stückzahlen nachgefragt werden.

Hochleistungsprozessoren
Die Ein- und Ausgabe ist die alleinige Angelegenheit des Systems. Die Anwendungsprogramme dürfen damit gar nichts zu tun haben. E-A-Befehle sind privilegierte Befehle, die nur vom Betriebssystem ausgeführt werden. Maßnahmen zur Steigerung der Verarbeitungsgeschwindigkeit und zum Bereitstellen sehr großer Speicherkapazitäten (Caches, virtuelle Speicher usw.) vertragen sich nicht mit den Anforderungen der Ein- und Ausgabe. Sie müssen deshalb umgangen werden. Die Vorgänge der Ein- und Ausgabe sollen möglichst wenig Prozessorleistung beanspruchen. Die typische Systemauslegung beruht deshalb auf autonomen E-A-Einrichtungen (Abbildung 2.10), die über mehrere Softwareschichten angesprochen werden (Gerätetreiber). Die E-A-Vorgänge weisen lange Latenzzeiten auf; das Realzeitverhalten ist aus Sicht der Anwendungsprogrammierung genaugenommen nicht vorhersagbar[17]. Ein den Mikrocontrollern entsprechendes Realzeitverhalten ergibt sich nur dann, wenn man die betreffenden Abläufe auf

17: Die extrem hohen Datenraten nützen in dieser Hinsicht nichts, da sie nur für Datenströme gelten, die an der Anwendung gleichsam vorbeigeleitet werden (Datenübertragung zwischen Arbeitsspeicher und E-A-Gerät).

der Ebene der Gerätetreiber implementiert oder die gesamte Anwendung als Stand-alone-Programm laufen lässt.

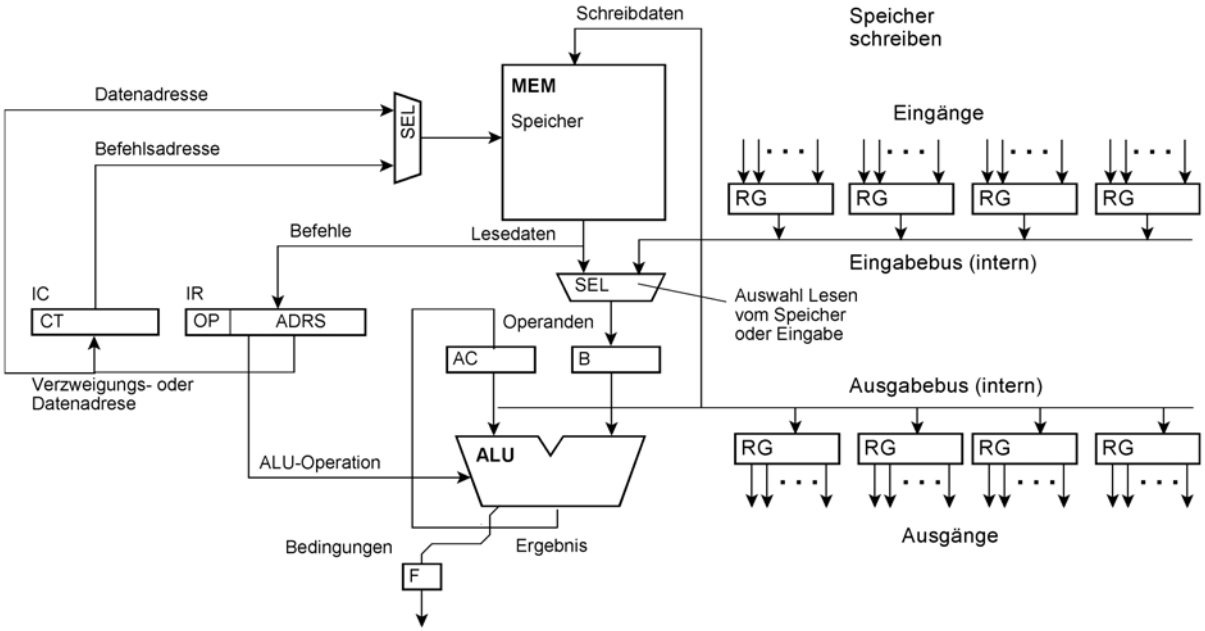


Abb. 2.9 Mikrocontroller – hier eine Akkumulatormaschine – mit Ein- und Ausgaberegistern.

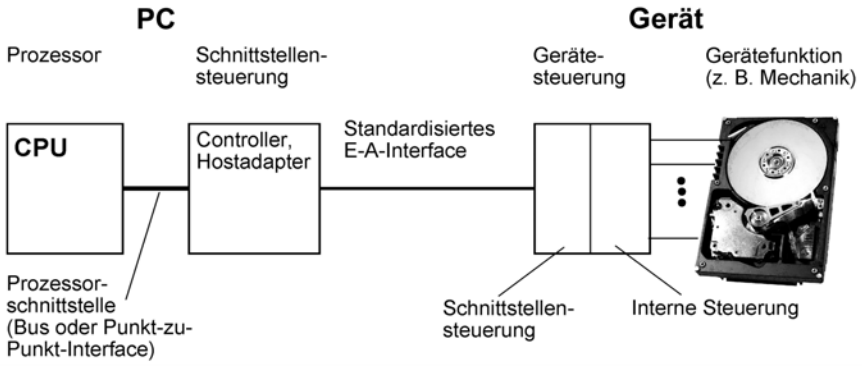


Abb. 2.10 Die Struktur eines E-A-Subsystems mit standardisierten Interfaces.

2.4 Prozessorschaltkreise

Der Prozessorkern (Processor Core)
Dieser Begriff bezeichnet den Verbund der Verarbeitungs- und Steuerschaltungen, die die eigentliche Verarbeitungsleistung erbringen. Nicht zum Prozessorkern gehören Caches höherer Ebenen sowie die Schaltmittel der Interfaces und Bussysteme (Abbildung 2.11).

Am Kern ändert sich meist nicht viel
Ein moderner Hochleistungsprozessor ist so kompliziert (und seine Entwicklung so kostspielig), dass die Entwickler froh sind, wenn sie ihn soweit zum Funktionieren gebracht haben, dass man

es wagen kann, ihn zu verkaufen[18]. Deshalb werden Prozessorkerne nur sehr behutsam geändert. Die Leistungssteigerung wird vor allem erreicht durch:

- Verbesserungen der Halbleitertechnologie (vor allem: Fertigung mit geringeren Strukturbreiten – das erlaubt den Betrieb mit höheren Taktfrequenzen).
- Größere und besser organisierte Caches.
- Verbesserungen am Interface (schneller, wirksamere Vorkehrungen zum Puffern von Zugriffen (Schreibpufferung, spekulatives Lesen)).

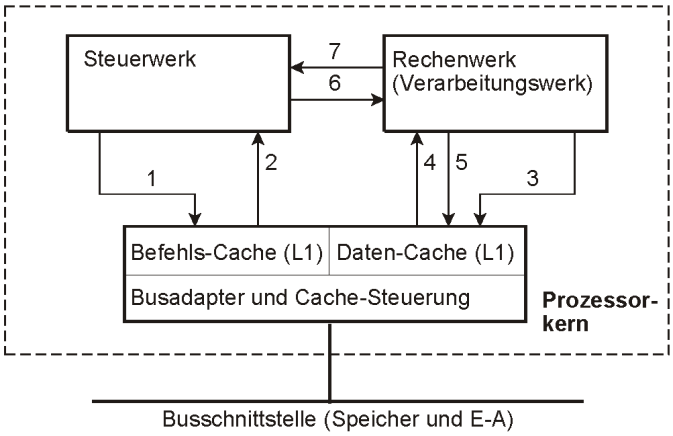


Abb. 2.11 Ein Prozessorkern im Blockschaltbild. 1 - Befehlsadresse; 2 - Lesen der Maschinenbefehle; 3 - Datenadresse; 4 - Lesen von Daten; 5 - Schreiben von Daten; 6 - Steuersignale; 7 - Bedingungssignale.

Skalierbarkeit (Scalability)

Der Begriff hat im Computerwesen mehrere Bedeutungen. Hier – wenn es um den einzelnen Prozessor geht – meint man damit die Möglichkeit, eine ganze Familie von Prozessoren mit ein und derselben Architektur anbieten zu können: besonders preisgünstige, solche mit besonders geringem Strombedarf, mit besonders hoher Verarbeitungsleistung usw. In der Vergangenheit hat man, um derartige Anforderungen zu erfüllen, jeweils eine eigene Hardware entworfen. So bestand das legendäre System /360 aus mehreren Modellen mit Verarbeitungsbreiten von 4, 8, 16, 32 und 64 Bits, die teils auf geringe Kosten, teils auf hohe Leistung hin ausgelegt waren. Heutzutage greift man für solche Abstufungen auf jeweils einen einzigen Prozessorkern zurück, der lediglich mit mehr oder weniger aufwendigen Zusatzschaltungen, Caches, Schnittstellen usw. erweitert und in verschiedenen Halbleitertechnologien gefertigt wird.

Mikrocontroller und Skalierbarkeit

Die Schaltkreistechnologien ermöglichen es, immer kleinere Halbleiterstrukturen zu fertigen. Geht es um Digitalschaltungen in CMOS-Technologie, so können diese Fortschritte auch voll und ganz ausgenutzt werden. Die Verkleinerung der Strukturen findet aber ihre Grenze bei Ausgangsstufen und Analogschaltungen. Um einen bestimmten Strom zu treiben, braucht man entsprechend niedrige Kanalwiderstände (R_{DSon}) der Treibertransistoren und somit entsprechend viel Schaltkreisfläche. Infolgedessen werden Prozessorkerne und Speicher immer kleiner,

18: Vollkommen fehlerfrei ist er dann immer noch nicht. Die Kleinigkeiten – man spricht dann auf einmal Latein und nennt sie „Errata" – werden teils stillschweigend mittels Software ausgebügelt. Praxistip: bei absolut unerklärlichen Fehlern – also dann, wenn man beim besten Willen nicht darauf kommt – in den Errata Sheets des jeweiligen Prozessortyps nachsehen. (Internet).

während der Platzbedarf für die E-A-Ausstattung praktisch gleich bleibt. Mit dem Fortschreiten der Schaltungsintegration lohnt es sich immer weniger, beim Prozessorkern zu sparen.

Daraus folgt die richtige Lösung für typische kleinere Systeme: Einfachheit der Wirkprinzipien bei Großzügigkeit der Auslegung. Die Verarbeitungsbreite sollte der Stellenanzahl der längsten Binärzahlen entsprechen, die in leistungsentscheidenden Abläufen vorkommen. Die Mindestforderung: Verarbeitungsbreite = Adresslänge, so dass alle üblichen Adressrechenvorgänge in jeweils einem einzigen Maschinentakt erledigt werden können. Ein solcher Controller kommt mit vergleichsweise niedrigen Taktfrequenzen aus.

Mehrere Kerne

Die Schaltungsintegration hat einen Stand erreicht, der es ermöglicht, auf einem Schaltkreis mehrere Prozessorkerne unterzubringen. In den oberen Leistungsbreichen werden die Kerne an gemeinsame Caches und Busschnittstellen angeschlossen (Abbildung 2.12). Das eigentliche Problem besteht darin, derartige Anordnungen auszunutzen. Mehrere Prozessoren kann man offensichtlich nur dann sinnvoll beschäftigen, wenn es mehrere Programme gibt, die gleichzeitig auszuführen sind. Manchmal ist das sozusagen von Hause aus der Fall. Gibt es mehrere Anwendungen, die gleichzeitig laufen sollen, kann man jeder einen eigenen Prozessor zuweisen[19]. Sollen Anordnungen aus mehreren Prozessoren hingegen dazu ausgenutzt werden, die einzelne Anwendung schneller auszuführen, so muss man sich etwas einfallen lassen ...

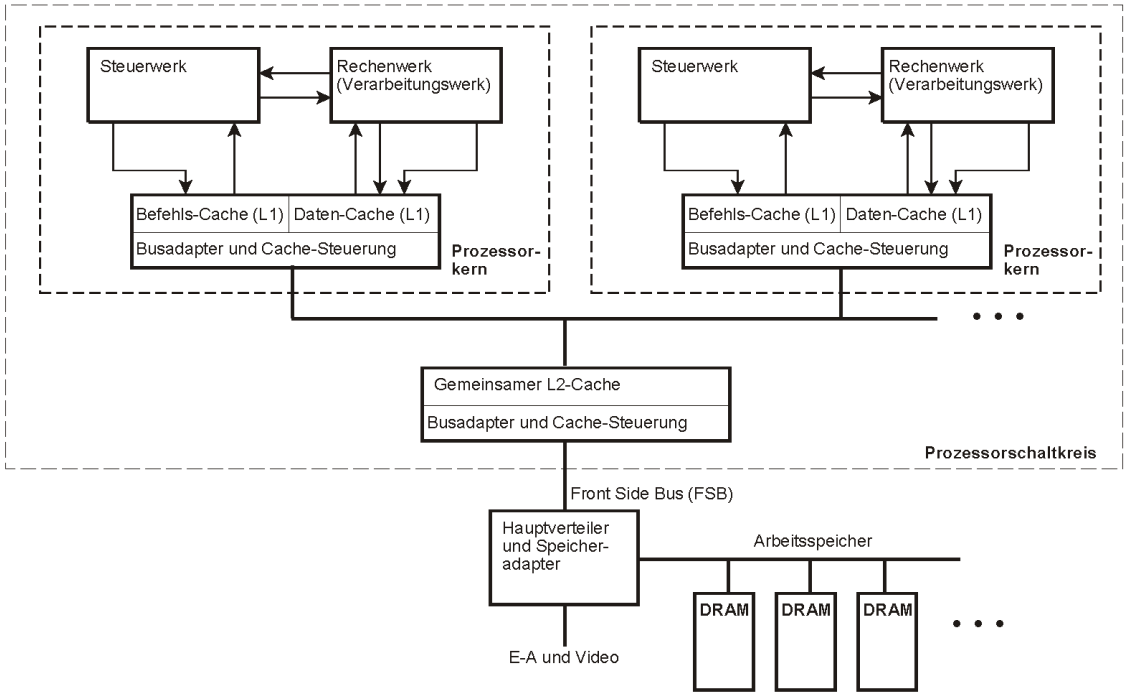


Abb. 2.12 Prozessorschaltkreis mit zwei Kernen. Die Erweiterbarkeit auf mehr Kerne versteht sich von selbst.

Keine Milchmädchenrechung

Je mehr Kerne und Prozessorfunktionen, desto leistungsfähiger der Computer. Zwei Kerne leisten mehr als einer, vier Kerne leisten das Doppelte von zweien usw. Diese Aussagen stimmen aber

19: Über viele Jahre hinweg hat es sich genau anders herum verhalten. Man konnte sich nur einen einzigen Prozessor leisten, und den mussten sich mehrere Anwendungen teilen (Multitasking).

nur sehr näherungsweise. Welcher Leistungsgewinn sich tatsächlich ergibt, hängt vor allem von der Art der Programme und vom Geschick der Programmierer ab. Wenn es nichts gibt, was parallel (also gleichzeitig) laufen kann, oder wenn man es ungeschickt anstellt, dann nützen auch noch so viele Kerne nichts.

3. Einführung in die Rechnerarchitektur

3.1 Datenstrukturen

3.1.1 Adressierbare Behälter

Die in der Architektur festgelegten elementaren Datenstrukturen (Bytes, Wörter usw.) sind Behälter, die beliebige Angaben aufnehmen können. Die jeweilige Bedeutung des Inhaltes ergibt sich aus dem Gebrauch in der Maschine. Wird eine UND-Verknüpfung mit dem Inhalt eines solchen Behälters ausgeführt, ist es ein Binärvektor, wird eine Gleitkommamultiplikation ausgeführt, ist es eine Gleitkommazahl usw.

Bytes
Herkömmlicherweise entspricht das Wort einer Binärzahl und das Byte einem Zeichen (Buchstaben, Ziffer usw.). In heutigen Architekturen ist ein Byte acht Bits lang (Abbildung 3.1). Es ist die kürzeste adressierbare Datenstruktur (Byteadressierung).

Halbbytes
Ein aus acht Bitpositionen bestehendes Byte kann in zwei Hälften unterteilt werden. Diese vier Bits langen Datenstrukturen werden als Halbbytes, Tetraden oder Nibbles bezeichnet. Sie dienen vor allem dazu, binär codierte dezimale Ziffernstellen aufzunehmen.

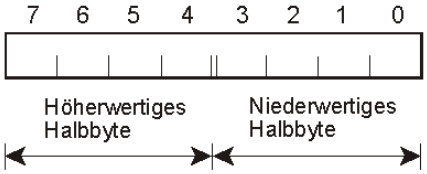


Abb. 3.1 Das Byte als elementare Datenstruktur. Es kann in zwei Halbbytes aufgeteilt werden.

Wörter
Die Länge eines Wortes entspricht typischerweise einer Zweierpotenz (2, 4, 8 usw. Bytes = 16, 32, 64 usw. Bits). Zu den ersten Grundsatzentscheidungen beim Ausarbeiten einer Prozessorarchitektur gehört die Festlegung einer Wortlänge, um die alles andere gleichsam herumgebaut wird (Maschinenwort). Demgemäß ergibt sich eine 8-Bit-Architektur, eine 16-Bit-Architektur, eine 32-Bit-Architektur usw. Eine Informationsstruktur, die nur halb so lang ist, heißt Halbwort, eine mit zweifacher Länge Doppelwort und eine mit vierfacher Länge Quadwort (Abbildung 3.2).

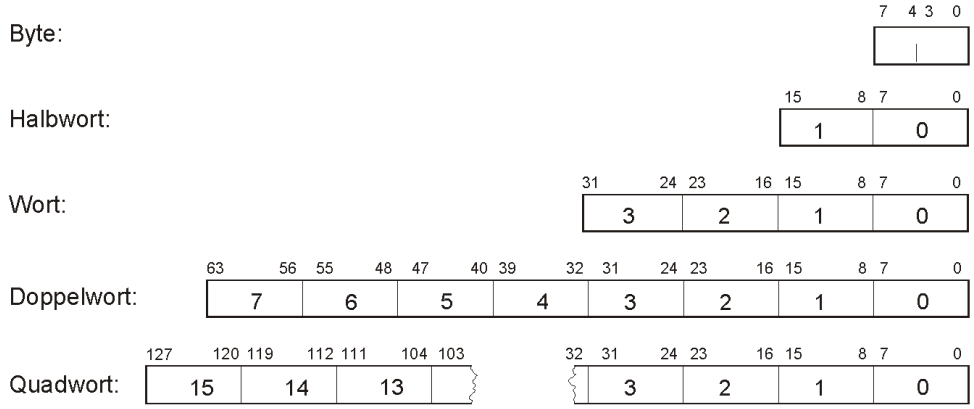


Abb. 3.2 Bytes und Wörter. Die Bezeichnungen betreffen eine architekturseitige Wortlänge (Maschinenwort) von 32 Bits.

3.1.2 Natürliche Binärzahlen

Natürliche Zahlen sind vorzeichenlos (unsigned). Der niedrigste Wert ist Null. Der Wertebereich einer natürlichen Binärzahl x aus n Bits (Abbildung 3.3) ist gegeben durch:

$$0 \le x \le 2^n - 1$$

Beispiel: eine natürliche Binärzahl mit acht Bits (n = 8):

- Kleinster Wert = 0 = 0000 0000B = 00H.
- Größter Wert = $2^8 - 1 = 255 = 1111\ 1111B = FFH$.

In Tabelle 3.1 sind typische Formate natürlicher Binärzahlen angegeben.

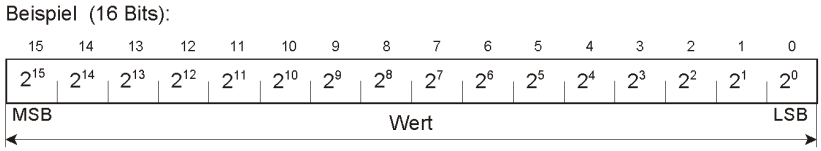


Abb. 3.3 Natürliche (vorzeichenlose) Binärzahlen.

Länge		Größter Wert	
in Bits	**in Bytes**		
8	1	$2^8 - 1 =$	255
16	2	$2^{16} - 1 =$	65 535
32	4	$2^{32} - 1 =$	4 294 967 295 (4G – 1)
64	8	$2^{64} - 1 =$	$18{,}4 \cdot 10^{18}$ (18,4 Trillionen*)

*): Engl. Quintillionen. $2^{64} - 1 = 18\ 446\ 744\ 073\ 709\ 551\ 615$.

Tabelle 3.1 Natürliche Binärzahlen als elementare Datentypen.

3.1.3 Ganze Binärzahlen

Ganze Zahlen (Signed Numbers, Integers) haben ein Vorzeichen (+ oder –). Das Vorzeichen belegt die höchstwertige Bitposition (Abbildung 3.4).

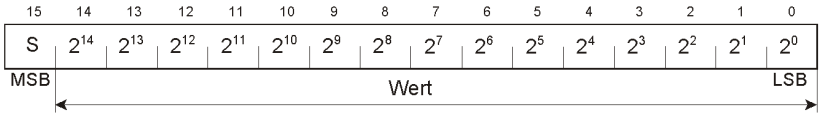


Abb. 3.4 Beispiel einer ganzen Binärzahl (16 Bits). S = Vorzeichen (Sign). Die übliche Vorzeichencodierung: 0 = positiv (+), 1 = negativ (–).

Kleiner und größer
Eine Zahl ist um so größer, je näher sie an plus Unendlich (+ ∞) liegt; sie ist um so kleiner, je näher sie an minus Unendlich (– ∞) liegt (Abbildung 3.5). Beispiele: – 8 ist kleiner als – 3; – 2 ist kleiner als + 2; – 3 ist größer als – 8; + 2 ist größer als – 2. – ∞ ist kleiner als + ∞. Jede negative Zahl ist kleiner als Null. Die größte negative Zahl ist – 1.

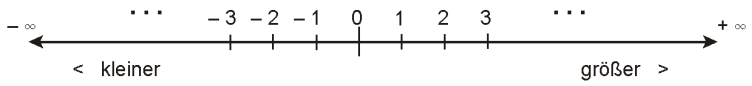


Abb. 3.5 Kleiner und größer – der Zahlenstrahl.

Positive und negative Zahlen
In der Rechnerarchitekur sind vor allem die folgenden Zahlendarstellungen von Bedeutung:

Vorzeichen und Wert (Sign/Magnitude)
Die Wertangabe ist eine natürliche Binärzahl, die durch ein Vorzeichenbit in der höchstwertigen Bitposition ergänzt wird Die übliche Codierung des Vorzeichens: 0 = positiv, 1 = negativ. Anwendungsbeispiel: Signifikanden in Gleitkommazahlen.

Zweierkomplement (Two's Complement)
Eine positive Binärzahl ist eine natürliche Binärzahl mit vorangestelltem Vorzeichenbit Null. Eine negative Binärzahl ist das Zweierkomplement der betragsgleichen positiven Zahl. Es ergibt sich durch bitweise Negation einschließlich des Vorzeichenbits und anschließendes Addieren einer Eins. Das Zweierkomplement einer n-stelligen Binärzahl x:

$$-x = 2^n - x$$

Diese Darstellung hat sich als Industriestandard durchgesetzt, da die elementaren Rechenschaltungen den geringsten Aufwand erfordern. Sie sind für vorzeichenbehaftete und vorzeichenlose Zahlen gleichermaßen nutzbar. Eine Vorzeichenbehandlung ist nicht erforderlich, weil das Addieren und Subtrahieren auf das Addieren vorzeichenloser Zahlen zurückgeführt werden kann. Zahlen, die länger sind als die Verarbeitungsbreite, können in aufeinander folgenden Abschnitten addiert oder subtrahiert werden.

Der Wertebereich einer ganzen Binärzahl x mit n Bitpositionen ist gegeben durch:

$$-\left(2^{n-1}\right) \le x \le 2^{n-1} - 1$$

Beispiel: eine natürliche Binärzahl mit acht Bits (n = 8):

- Kleinste negative Zahl = – (2^7) = – 128 = 1000 0000B = 80H.
- – 1 (größte negative Zahl) = 1111 1111B = FFH.
- 0 = 0000 0000B = 00H.
- + 1 (kleinste positive Zahl) = 0000 0001B = 01H.
- Größte positive Zahl = 2^7 – 1 = 127 = 0111 1111B = 7FH.

In Tabelle 3.2 sind typische Formate ganzer Binärzahlen angegeben.

Länge		Größte Werte			
in Bits	in Bytes	negativ		positiv	
8	1	$-2^7 =$	– 128	$2^7 - 1 =$	127
16	2	$-2^{15} =$	– 32 768	$2^{15} - 1 =$	32 767
32	4	$-2^{31} =$	– 2 147 483 648	$2^{31} - 1 =$	2 147 483 647
64	8	$-2^{63} =$	$\approx -9{,}2 \cdot 10^{18}$	$2^{63} - 1 =$	$\approx 9{,}2 \cdot 10^{18}$

Ganz genau: $-2^{63} = -9\,223\,372\,036\,854\,775\,808$; $2^{63} - 1 = 9\,223\,372\,036\,854\,775\,807$.

Tabelle 3.2 Ganze Binärzahlen (Integer-Zahlen) als elementare Datentypen.

3.1.4 Sättigungsarithmetik

Die Mathematik kennt unendlich viele positive und negative Zahlen. Die naheliegende graphische Darstellung ist der Zahlenstrahl. Im Computer können aber die Zahlen nur mit endlich vielen Bits dargestellt werden. Der Zahlenstrahl wird somit zum Kreis (Abbildung 3.6). Wenn man beispielsweise von Null aus vorwärts zählt (0 => 1 => 2 => 3 usw.), so kommt man irgendwann einmal zur größten positiven Zahl und von dort durch einfaches Weiterzählen zur kleinsten negativen (von 0111B nach 1000B bzw. von + 7 nach – 8). Dann geht es rückwärts weiter bis zur – 1 (1111B) und im nächsten Zählschritt wieder zur Null. Wenn man beim Rechnen nicht auf die Überlaufbedingung achtet, kann das Ergebnis nach der jeweils anderen Seite umschlagen: wird die kleinste negative Zahl unterschritten, so ergibt sich ein positives Ergebnis, wird die größte positive Zahl überschritten, ein negatives. Abbildung 3.6 zeigt das an einem Rechenbeispiel (5 + 3 = 0101B + 0011B = 1000B = – 8). Ein weiteres Beispiel: – 7 + (– 2) = + 7 (1001B + 1110B = 0111B). Wegen dieses Umschlagens heißt die herkömmliche (Zweierkomplement-) Arithmetik im Englischen auch Wrap-Around-Arithmetics.

Das Prinzip der Sättigungsarithmetik (Saturation Arithmetics) besteht darin, dieses Umschlagen zu vermeiden und die Zahlwerte sozusagen gegen den jeweiligen Anschlag laufen zu lassen (Tabelle 3.3). Wird der Wertebereich überschritten, so wird als Ergebnis der jeweilige Größtwert geliefert, wird der Wertebereich unterschritten, der jeweilige Kleinstwert.

Das ist vor allem beim Rechnen mit Audio- und Videodaten von Bedeutung. Eine maximale Amplitude kann nicht noch weiter wachsen, ein Farbwert “schwarz” kann nicht noch dunkler werden usw. Hingegen würde bei Nutzung der herkömmlichen Arithmetik der Versuch, ein schwarzes Pixel noch schwärzer zu machen, zu einem besonders hellen Pixel führen. Diesen Effekt könnte man auch vermeiden, indem man die Überlaufbedingung auswertet und die Ergebnisse entsprechend korrigiert. Das kostet aber Zeit. Es ist wichtig, dass Audio- und Videodaten als gleichsam fließende Datenströme verarbeitet werden können. Einzelne “Ausreißer” im Datenstrom sind akzeptabel (sie äußern sich schlimmstenfalls als Knacks oder als kurzzeitige Bildstörung), nicht aber Verzögerungen, wie sie durch die programmseitige Behandlung von Überlaufbedingungen und Bereichsüberschreitungen entstehen könnten.

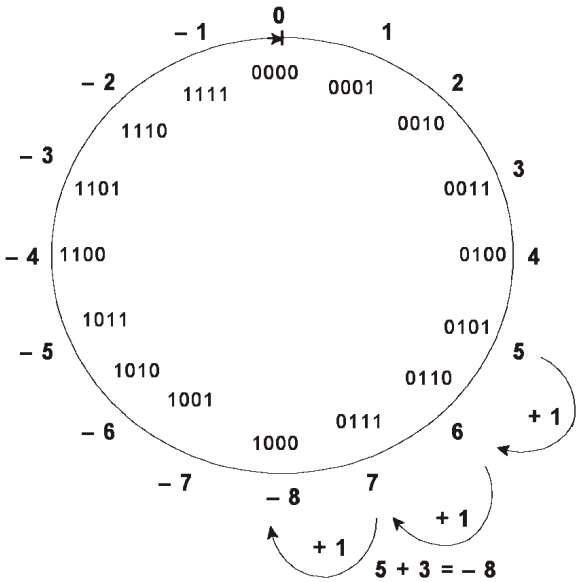


Abb. 3.6 Zweierkomplementarithmetik als Wrap-Around-Arithmetik.

Zahlenart	Herkömmliche Arithmetik*	Sättigungsarithmetik
Natürliche (vorzeichenlose) Binärzahlen (n Bits)	• Bereichsunterschreitung ergibt 2^n - Resultat (Zweierkomplement). • Bereichsüberschreitung ergibt Resultat – 2^n.	• Bereichsunterschreitung ergibt stets Null. • Bereichsüberschreitung ergibt stets den größten Wert (2^n – 1; FFF...FH).
Ganze (vorzeichenbehaftete) Binärzahlen (n Bits)	• Bereichsunterschreitung ergibt positiven Wert (2^n + Resultat). • Bereichsüberschreitung ergibt negativen Wert (– (2^n – Resultat)).	• Bereichsunterschreitung ergibt stets den kleinsten negativen Wert (– 2^{n-1}; 800...0H). • Bereichsüberschreitung ergibt stets den größten positiven Wert (2^{n-1} – 1; 7FF...FH).

*: "Resultat" ist hier das eigentliche (mathematische) Rechenergebnis.

Tabelle 3.3 Herkömmliche und Sättigungsarithmetik.

3.1.5 Gleitkommazahlen

Gleitkommazahlen (Floating Point Numbers, Reals) sind fest formatierte maschineninterne Darstellungen von Näherungswerten reeller Zahlen. Reelle Zahlen sind vorzeichenbehaftete Zahlen mit beliebig vielen Dezimalstellen nach dem Komma. Wertebereich und Stellenanzahl der reellen Zahlen sind unbegrenzt. Gleitkommazahlen sind hingegen endlich und diskret, das heißt:

- Es kann nur ein endliches Intervall aus dem Bereich der reellen Zahlen dargestellt werden.
- Es kann nur eine endliche Menge von Zahlen exakt dargestellt werden.
- Werte, die nicht exakt darstellbar sind, müssen auf den jeweils nächstliegenden darstellbaren Wert gerundet werden.

Gleitkommazahlen beruhen auf der Exponentialschreibweise, die auch im Dezimalsystem üblich ist. So lässt sich die Zahl 5 621 000 beispielsweise darstellen als

- $5\,621 \cdot 10^3$,
- $562{,}1 \cdot 10^4$,
- $56{,}21 \cdot 10^5$,
- $5{,}621 \cdot 10^6$,
- $0{,}5621 \cdot 10^7$ usw.

Diese Schreibweise hat die allgemeine Form $r \approx s \cdot b^e$. Darin ist

- r die reelle Zahl, deren Wert, wenn möglich, exakt, zumindest aber näherungsweise anzugeben ist,
- s die Wertangabe (Signifikand),
- b die Basis des Zahlensystems (im Beispiel also 10),
- e der Exponent.

Der Exponent gibt an, um wie viele Stellen der Signifikand zu verschieben ist, damit sich der tatsächliche Wert ergibt. Ein positiver Exponent entspricht einer Linksverschiebung. Damit sich aus $5621 \bullet 10^3$ 5621 000 ergibt, ist der Signifikand um drei Stellen nach links zu verschieben. Ein negativer Exponent entspricht einer Rechtsverschiebung. Damit sich aus 5261 000 000 $\bullet 10^{-3}$ 5261 000 ergibt, ist der Signifikand um drei Stellen nach rechts zu verschieben.

Zur Entwicklungsgeschichte
Die Gleitkommadarstellung im Computer (Abbildung 3.7) beruht auf Binärzahlen. Die Standards ANSI/IEEE 754 und 854 haben sich durchgesetzt. Die weiteren Erläuterungen beschränken sich deshalb auf Zahlendarstellungen, die diesen Standards entsprechen.

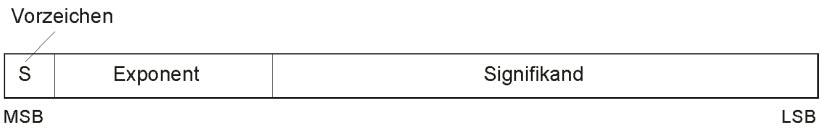


Abb. 3.7 Der grundsätzliche Aufbau einer Gleitkommazahl.

Die Basis
Für Binärzahlen kommt als Basis b nur eine Zweierpotenz in Frage. Unter anderem wurden die Werte $2^1 = 2$, $2^3 = 8$ und $2^4 = 16$ verwendet. Die standardisierten Gleitkommaformate beruhen auf der Basis b = 2.

Vorzeichen und Signifikand
Beide Angaben bilden eine vorzeichenbehaftete Binärzahl. In den standardisierten Gleitkommaformaten wird sie mit Vorzeichen und Wert (Sign/Magnitude) dargestellt. Der Signifikand ist die Wertangabe. Da sich die Zahlen lediglich im Vorzeichen unterscheiden, genügt es, in den folgenden Erläuterungen nur die positiven Zahlen zu berücksichtigen.

Der Exponent
Der Exponent e veranlasst eine Verschiebung des Signifikanden um e Stellen. Ein positiver Exponent e bedeutet eine Linksverschiebung um e Bitpositionen (Multiplikation $2^e \bullet s$), ein negativer Exponent bedeutet eine Rechtsverschiebung (Multiplikation $2^{-e} \bullet s$).

Der Wert einer Gleitkommazahl zur Basis b = 2 mit dem Exponenten e und einem n-stelligen Signifikanden s ergibt sich zu:

$$(-1)^s \cdot 2^e \cdot \left(2^{n-1} \cdot s_{n-1} + 2^{n-2} \cdot s_{n-2} + \ldots + 2^0 \cdot s_0\right)$$

Normalisierung

Man ist bestrebt, alle Signifikandenstellen auszunutzen. Führende Nullen haben keinen Wert. Der Exponent wird deshalb so eingerichtet, dass es keine führenden Nullen gibt. Solche Gleitkommazahlen heißen normalisiert (normalized). Es gibt zwei Auslegungen (Abbildung 3.8):

a) Mit einer Null vor dem binären Komma; mit anderen Worten, als echter Bruch.

b) Als Eins mit nachfolgendem Binärbruch. Diese Auslegung wird in den Standards bevorzugt. Der Signfikand ist der Zähler des Bruchs (Fraction). Der Exponent gibt an, welche Wertigkeit die Binärstelle vor dem Komma hat.

Abb. 3.8 Normalisierte Gleitkommazahlen am Beispiel eines achtstelligen Signifikanden. a) als echter Binärbruch; b) mit führender Eins.

Die Gleitkommazahl als lange Binärzahl

Die Gleitkommazahl ist im Grunde nichts anderes als die komprimierte Darstellung einer langen Binärzahl (Abbildung 3.9).

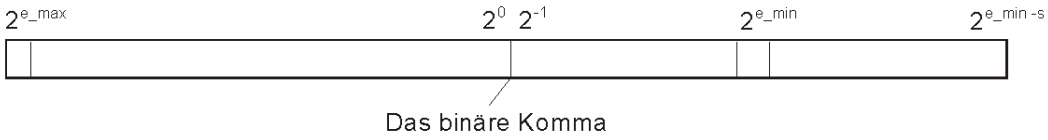


Abb. 3.9 Normalisierte Gleitkommazahlen als Darstellungen langer Binärzahlen.

In den Standards ist festgelegt, dass – wie in Abbildung 3.8b gezeigt – die normalisiere Gleitkommazahl eine Eins vor dem Komma hat. In den meisten Formaten wird die führende Eins nicht mitgespeichert (implizite Darstellung, Hidden Bit). Der Exponent e setzt diese Eins auf die Bitposition 2^e. Die Bits des Signifikanden belegen die nächst-niederwertigen Bitpositionen. Die verbleibenden Bitpositionen werden mit Nullen aufgefüllt. Abbildung 3.10 veranschaulicht, wie sich der Wertebereich einer normalisierten Gleitkommazahl ergibt. Der größte Wert des Exponenten ist e_max, der kleinste e_min. Der Signifikand besteht aus s Bitpositionen. Die lange Binärzahl setzt sich aus einer ganzen Zahl und einem Bruch zusammen. Die ganze Zahl hat e_max + 1 Bitpositionen, der Bruch |e_min| + s. Die Gesamtlänge ergibt sich somit zu

e_max + 1+ |e_min| + s Bitpositionen.

Die Wertigkeit der Bitstellen liegt zwischen 2^{e_max} und $2^{e\_min\,-\,s}$.

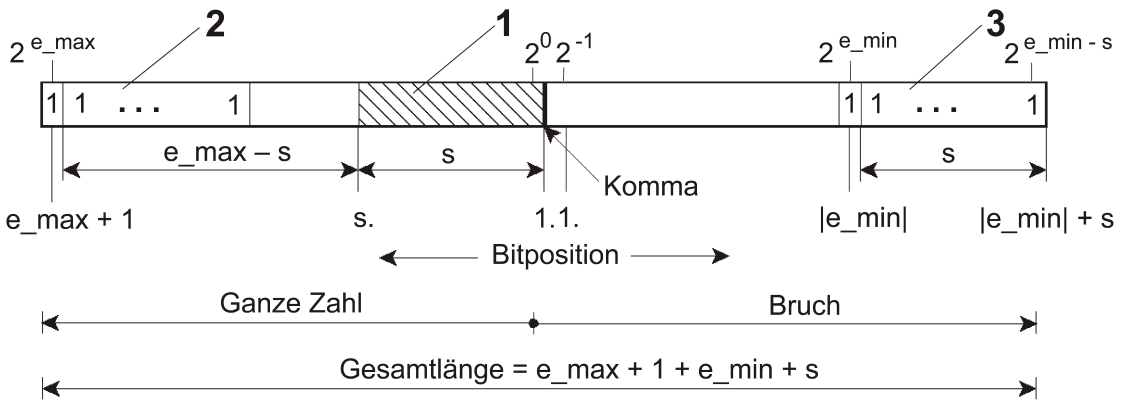


Abb. 3.10 Zum Wertebereich normalisierter Gleitkommazahlen. 1 - der Signifikand als ganze Binärzahl (ohne Verschiebung); 2 - der Signifikand der größten darstellbaren Zahl (links außen alles Einsen); 3 - der Signifikand der kleinsten darstellbaren Zahle (rechts außen alles Nullen).

Der größte Wert

Der Exponent setzt die Eins auf die Bitposition 2^{e_max} (Abbildung 3.11a). Von der ersten Bitposition (Wertigkeit 2^0) an gesehen ist dies die e_max + 1-te Bitposition. Alle Signifikandenstellen sind mit Einsen belegt (Abbildung 3.11b). s Einsen ergeben – als natürliche Binärzahlen gesehen – einen Wert $2^s - 1$. Die höchstwertige Signifikandenstelle befindet sich auf der e_max-ten Bitposition. Sie ist gegenüber der Ursprungslage um e_max – s Bitpositionen nach links verschoben. Der größte Wert ergibt sich somit aus dem Stellenwert der führenden Eins = 2^{e_max} und dem Wert des um e_max – s Bits verschobenen, mit Einsen belegten Signifikanden = $2^{e\_max-s} \bullet (2^s - 1)$. Diese Summe kann vereinfacht werden:

$$2^{e\_max} + 2^{e\_max-s} \cdot \left(2^s - 1\right) = 2^{e\_max+1} - 2^{e\_max-s}$$

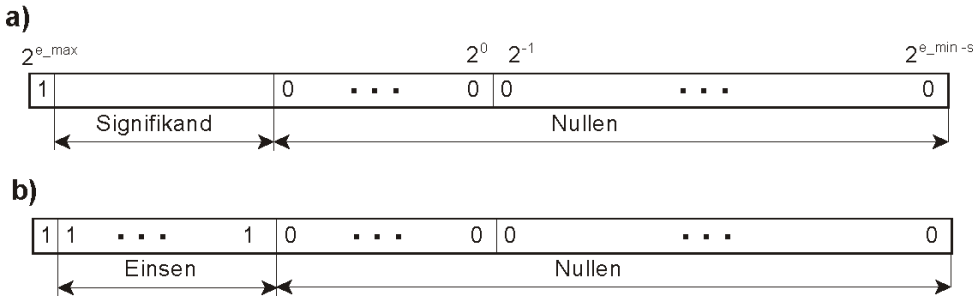


Abb. 3.11 Die größten Werte. a) Die lange Binärzahl mit dem größten Exponenten. b) Der größte Wert ergibt sich, wenn der Signifikand alles Einsen enthält.

Der kleinste Wert

Der Exponent setzt die Eins auf die Bitposition 2^{e_min}. (Abbildung 3.12a). Der Signifikand ist mit Nullen belegt (Abbildung 3.12b). Der kleinste Wert entspricht somit 2^{e_min}.

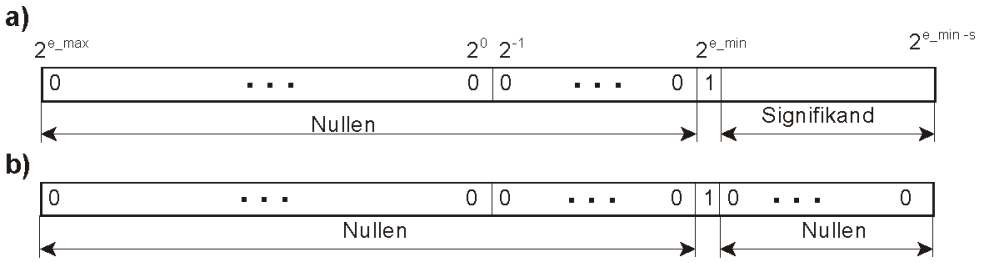


Abb. 3.12 Die kleinsten Werte. a) Die lange Binärzahl mit dem kleinsten Exponenten. b) Der kleinste Wert ergibt sich, wenn der Signifikand alles Nullen enthält.

Der gesamte Wertebereich
Der Wertebereich der Gleitkommazahl z ergibt sich zu:

$$2^{e\_max+1} - 2^{e\_max-s} \leq z \leq 2^{e\_min}$$

Denormalisierte Zahlen
Die normalisierte Darstellung führt zu Genauigkeitsproblemen, wenn die Zahlenwerte nahe bei Null liegen. Auch die Null selbst ist nicht darstellbar. Die kleinste normalisierte Zahl ist 2^{e_min} (vergleiche Abbildung 3.15b). Um die Lücke zwischen der kleinsten normalisierten Zahl und der Null mit Werten belegen zu können, sind denormalisierte Gleitkommazahlen vorgesehen. In solchen Zahlen steht anstelle der Eins eine Null vor dem Komma (Abbildung 3.13a). Somit ist auch die echte Null (True Zero) darstellbar. Die kleinste von Null verschiedene denormalisierte Zahl enthält eine einzige Eins der Wertigkeit 2^{e_min-s} (Abbildung 3.13b).

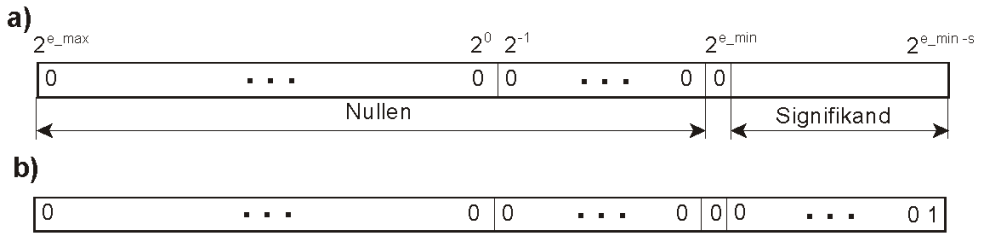


Abb. 3.13 Die denormalisierte Gleitkommazahl als ange Binärzahl. a) Prinzip; b) der kleinste von Null verschiedene Wert.

Standardisierte Gleitkommaformate
Es sind Gleitkommazahlen einfacher, doppelter, erweiterter und vierfacher Genauigkeit vorgesehen (Single Precision, Double Precision, Extended Precision, Quad Precision Reals). Sie sind 32, 64, 80 und 128 Bits lang (Abbildung 3.14, Tabellen 3.4 bis 3.7).

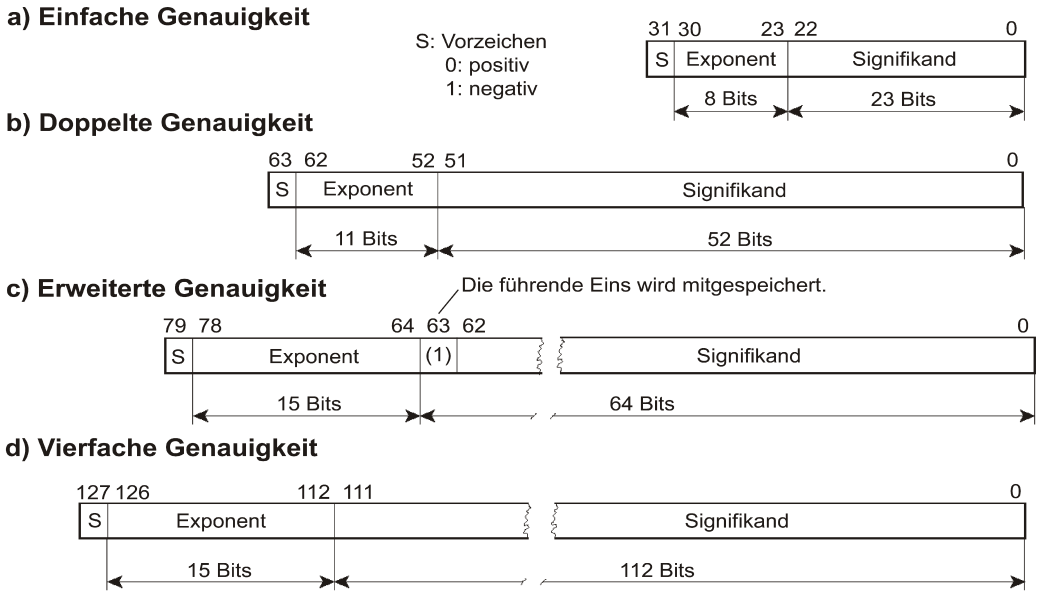


Abb. 3.14 Standardisierte Gleitkommaformate.

Gleitkommaformat	Länge (Bits)	Länge des Signifikanden	Exponent			
			Länge	e_max	e_min	Versatz (Bias)
Einfache Genauigkeit	32	23 (+1)[1)]	8	+ 127	– 126	+ 127
Doppelte Genauigkeit	64	52 (+1)[1)]	11	+ 1023	– 1022	+ 1023
Erweiterte Genauigkeit	80	64[2)]	15	+ 16383	– 16382	+ 16383
Vierfache Genauigkeit	128	112 (+1)[1)]	15	+ 16383	– 16382	+ 16383

1): die führende Eins wird *nicht* mitgespeichert; 2): die führende Eins wird mitgespeichert (beim Berechnen von Längen und Wertebereichen sind also nur 63 Signifikandenstellen anzusetzen).

Tabelle 3.4 Standardisierte Gleitkommaformate.

Format	Kleinster Wert, denormalisiert	Kleinster Wert, normalisiert	Größter Wert
Einfache Genauigkeit	2^{-149}	2^{-126}	$2^{128} - 2^{104}$
Doppelte Genauigkeit	2^{-1074}	2^{-1022}	$2^{1024} - 2^{971}$
Erweiterte Genauigkeit	2^{-16445}	2^{-16382}	$2^{16384} - 2^{16320}$
Vierfache Genauigkeit	2^{-16494}	2^{-16382}	$2^{16384} - 2^{16271}$

Tabelle 3.5 Wertebereiche (1). Binär.

Format	Kleinster Wert, denormalisiert	Kleinster Wert, normalisiert	Größter Wert
Einfache Genauigkeit	$1{,}4013 \cdot 10^{-45}$	$1{,}1755 \cdot 10^{-38}$	$3{,}4028 \cdot 10^{38}$
Doppelte Genauigkeit	$4{,}9407 \cdot 10^{-324}$	$2{,}2251 \cdot 10^{-308}$	$1{,}7977 \cdot 10^{308}$
Erweiterte Genauigkeit	$3{,}6452 \cdot 10^{-4951}$	$3{,}3621 \cdot 10^{-4932}$	$1{,}1897 \cdot 10^{4932}$
Vierfache Genauigkeit	$6{,}4752 \cdot 10^{-4966}$	$3{,}3621 \cdot 10^{-4932}$	$1{,}1897 \cdot 10^{4932}$

Tabelle 3.6 Wertebereiche (2). Dezimal (gerundet).

Format	Länge der Binärzahl	Ganze Zahl	Bruch
Einfache Genauigkeit	276	128	126 +23 = 149
Doppelte Genauigkeit	2098	1024	1022 + 52 = 1074
Erweiterte Genauigkeit	32829	16 384	16 382 + 63 = 16 445
Vierfache Genauigkeit	32878	16 384	16 382 + 112 = 16 494

Tabelle 3.7 Die Längen der darstellbaren Binärzahlen (Bits).

3.1.6 Binär codierte Dezimalzahlen

In binär codierten Dezimalzahlen (BCD-Zahlen) belegt eine Dezimalstelle vier Bits, die – als Binärzahlen – die Werte von 0 bis 9 annehmen können (Tabelle 3.8). Es gibt ungepackte und gepackte BCD-Zahlen (Abbildung 3.15). Eine ungepackte Dezimalziffer ist ein Byte mit einer einzigen Dezimalstelle im niederwertigen Halbbyte (Bits 0...3). Das höherwertige Halbbyte (Zone) wird nicht beachtet. Die typische ungepacke Dezimalzahl ist die aus Ziffernzeichen gebildete Zeichenkette. Gepackte BCD- Zahlen enthalten in jedem Byte zwei Dezimalstellen. Rechenbefehle beziehen sich zumeist auf gepackte Dezimalzahlen. Ganze Dezimalzahlen werden mit Vorzeichen und Wert dargestellt (Sign/Magnitude). Das Vorzeichen wird in einer weiteren Tetrade angegeben.

Neuner- und Zehnerkomplement

Das Neunerkomplement wird stellenweise gebildet. Das Neunerkomplement einer Zahl n ergibt sich zu 9 – n (vergleiche Tabelle 3.8). Das Zehnerkomplement entspricht dem Zweierkomplement der Binärzahlen. Das Zehnerkomplement einer n-stelligen Dezimalzahl z ergibt sich zu 10^n – z oder als Neunerkomplement + 1.

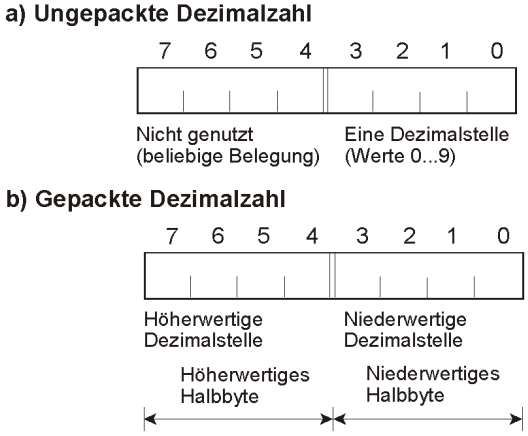


Abb. 3.15 Binär codierte Dezimalzahlen (BCD-Zahlen).

Binärstelle					Neunerkomplement				
Dez.	**23**	**22**	**21**	**20**	**Dez.**	**23**	**22**	**21**	**20**
0	0	0	0	0	**9**	1	0	0	1
1	0	0	0	1	**8**	1	0	0	0
2	0	0	1	0	**7**	0	1	1	1

Binärstelle					Neunerkomplement				
3	0	0	1	1	**6**	0	1	1	0
4	0	1	0	0	**5**	0	1	0	1
5	0	1	0	1	**4**	0	1	0	0
6	0	1	1	0	**3**	0	0	1	1
7	0	1	1	1	**2**	0	0	1	0
8	1	0	0	0	**1**	0	0	0	1
9	1	0	0	1	**0**	0	0	0	0

Tabelle 3.8 Der BCD-Code und das Neunerkomplement.

Binär codierte Dezimalzahlen

BCD-Arithmetik hat Zeit. Sie wird deshalb typischerweise programmseitig implementiert[20]. Die architekturseitige Unterstützung betrifft zumeist nur die Dezimalkorektur beim Addieren und Subtrahieren sowie das Austauschen des höher- und des niederwertigen Halbbytes in einem Byte (SWAP-Befehl).

Addieren und Subtrahieren in einer BCD-Stelle

Die 4-Bit-Angaben werden als vorzeichenlose Binärzahlen zueinander addiert oder voneinander subtrahiert. Das Ergebnis ist unter folgenden Bedingungen korrekt:

- Beim Addieren wird der Wert Neun nicht überschritten.
- Beim Subtrahieren wird der Wert Null nicht unterschritten.

Dezimalkorrektur

Ist das Ergebnis nicht korrekt, wird eine sogenannte Dezimalkorrektur ausgeführt:

- Wird beim Addieren der Wert Neun überschritten, so wird zusätzlich eine Sechs addiert.
- Wird beim Subtrahieren der Wert Null unterschritten, so wird zusätzlich eine Sechs subtrahiert.

Der Dezimalübertrag

Ein einlaufender Übertrag wird ebenso verrechnet wie im Binären. Ein Ausgangsübertrag ist abzugeben, wenn beim Addieren der Wert Neun überschritten oder beim Subtrahieren der Wert Null unterschritten wird (negatives Ergebnis = Borgen). Der Dezimalübertrag ist also immer dann zu bilden, wenn eine Dezimalkorrektur auszuführen ist.

Der Halbbyteübertrag

In vielen Architekturen wird der Übertrag von Bitposition 2^3 nach Bitpositon 2^4 als zusätzliche Bedingung erfasst (Half Carry Flag HF, Auxiliary Flag AF), um das Rechnen mit BCD-Stellen zu unterstützen.

20: Im Grunde läuft es darauf hinaus, das schulmäßige Rechnen in den vier Grundrechenarten auszuprogrammieren.

3.1.7 Zahlendarstellung, Wertebereich und Genauigkeit

Auflösung

Die Auflösung (Resolution) einer Zahlendarstellung ist der Unterschied zwischen zwei unmittelbar aufeinander folgenden Werten. Solche Werte unterscheiden sich in der niedrigstwertigen Bitposition (LSB). Die Änderung in der niedrigstwertigen Bitposition wird auf den darstellbaren Maximalwert oder den vollen Wertebereich (Full Scale FS) bezogen (Abbildung 3.16). Kann die Zahlenangabe Z verschiedene Werte darstellen, so hat die niedrigstwertigen Bitposition den Stellenwert

$$LSB = \frac{FS}{Z}$$

Handelt es sich um eine (ohne Vorzeichen) n Bits lange Binärzahl, so gilt

$$LSB = \frac{FS}{2^n}$$

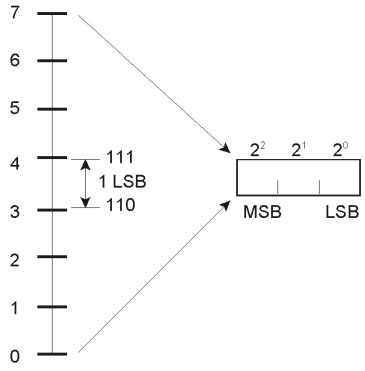


Abb. 3.16 Die Auflösung anhand eines Beispiels.

Dynamikbereich

Der Dynamikbereich (Dynamic Range DR) einer Zahlendarstellung ist das Verhältnis zwischen dem Maximalwert V_{max} und dem kleinsten positiven, von Null verschiedenen darstellbaren Wert V_{min} (Abbildung 3.17).

$$DR = \frac{V_{max}}{V_{min}}$$

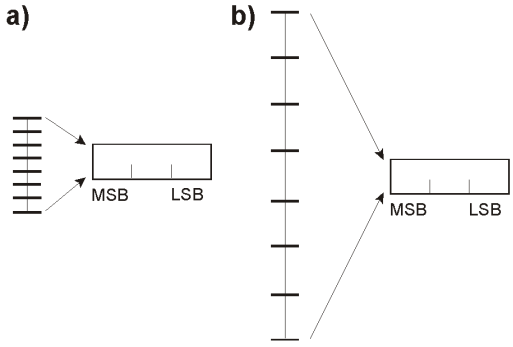


Abb. 3.17 Der Dynamikbereich. Das Bitmuster der Zahlendarstellung kann einen von 2^n Werten auswählen. Es ist aber die Frage, wie weit diese Punkte auf dem Zahlenstrahl auseinander liegen.

Genauigkeit

Eine Zahlendarstellung ist um so genauer, je weniger der codierte (maschineninterne) Wert M vom exakten Zahlenwert Z abweicht. Der auf den Endwert des Wertebereichs bezogene Darstellungsfehler einer (ohne Vorzeichen) n Bits langen Binärzahl entspricht der halben relativen Auflösung (½ LSB). Tabelle 3.9 enthält die Wertebereiche und Darstellungsfehler typischer Binärzahlenformate.

Bits	Wertebereich (Endwert)	Relativer Darstellungsfehler (±), bezogen auf den Endwert
6	63	0,8 %
8	255	0,2 %
10	1 023	0,05 % = 490 ppm
12	4 095	122 ppm
16	65 535	15,3 ppm
18	262 143	1,6 ppm
20	1 048 575	0,48 ppm
24	16 777 215	0,03 ppm
32	4 294 967 295	0,00011 ppm

Tabelle 3.9 Natürliche (vorzeichenlose) Binärzahlen, Wertebereiche und Darstellungsfehler. Wird das höchstwertige Bit als Vorzeichen genutzt, so kann nur der halbe Betrag dargestellt werden. Infogedessen halbiert sich der Wertebereich und verdoppelt sich der Darstellungsfehler.

Binärzahlen oder Gleitkommazahlen?

Sind es gleich viele Binärstellen, so kann auch nur die gleiche Anzahl an Punkten ausgewählt werden. Es ist lediglich die Frage, an welchen Orten auf dem Zahlenstrahl diese Punkte liegen. Bei ganzen Zahlen haben alle benachbarten Punkte den gleichen Abstand zueinander, nämlich Eins. Bei Gleitkommazahlen wächst der Abstand zwischen den Punkten mit der Größe des Exponenten. Der Wertebereich ist größer, aber die Auflösung und damit die Genauigkeit der Zahlendarstellung geringer.

Beispiel: Eine Temperaturgröße zwischen 35 und 45 °C (Abbildung 3.18) soll mit 32 Bits langen Zahlen angegeben werden. Eine vorzeichenlose Binärzahl kann rund 4 Milliarden unterschiedliche Werte darstellen. Der relative Darstellungsfehler beträgt 0,00011 ppm (vergleiche Tabelle 3.12). Eine Gleitkommazahl gemäß IEEE 785 kann etwa 16 Millionen Werte zwischen zwei Zweierpotenzen darstellen. Für das Intervall zwischen 35 und 45 kommt die Zweierpotenz 32 und somit der Exponent 5 in Betracht. Der Abstand zur nächsten Zweierpotenz (64) beträgt 32. Wird ein Intervall der Länge 32 mit 16 Millionen Werten aufgelöst, so kommen auf ein Intervall der Länge 45 – 35 = 10 fünf Millionen Werte. Das ergibt einen relativen Darstellungsfehler von 0,1 ppm. Die Binärzahl ist in diesem Anwendungsfall etwa tausendmal genauer als die gleich lange Gleitkommazahl.

Das Beispiel der Temperaturmessung wurde jedoch nur deshalb gewählt, weil es leicht verständlich ist. Dass es gar keine Temperatursensoren gibt, die so genau sind, bleibt dabei außer Betracht. Dieser Punkt ist aber in der Praxis von entscheidender Bedeutung. Man sollte zwar – als Vorsichtsmaßnahme gegen Rundungsfehler – mit höherer Auflösung rechnen als es die Außenwelt verlangt, es ist aber unzweckmäßig, eine extrem hohe Genauigkeit um jeden Preis anzustreben.

Die Gleitkommadarstellung überstreicht einen so großen Wertebereich, dass es zumeist nicht nötig ist, die anwendungsseitigen Größen zu skalieren. Man kann viele Schwierigkeiten von Anfang an vermeiden, wenn man alle numerischen Variablen als Gleitkommazahlen definiert. Die höhere Auflösung, die man mit Festkommazahlen und entsprechender Skalierung erreicht, kann man in vielen Anwendungsfällen gar nicht ausnutzen. Das Rechnen mit Gleitkommazahlen kostet aber deutlich mehr Aufwand (einen Prozessor mit Geitkommaverarbeitungseinheit (FPU)) oder Rechenzeit. Es bleibt also die Frage, ob man sich einen solchen Prozessor leisten kann. Oftmals ist es möglich, Prozessoren einzusetzen, die einen komfortablen Leistungsüberschuss aufweisen. Dafür kann man an anderen Stellen sparen (Speicherkapazität nur so groß, E-A-Ausstattung nur so umfangreich wie nötig). Es gibt aber auch Fälle, wo man mit vergleichsweise bescheidenen Mikrocontrollern auskommen muss. Dann ist es erforderlich, die Festkommarithmetik auszuprogrammieren. Solche Anwendungsfälle sind so häufig, dass in manchen Architekturen spezielle Maschinenbefehle eingeführt wurden.

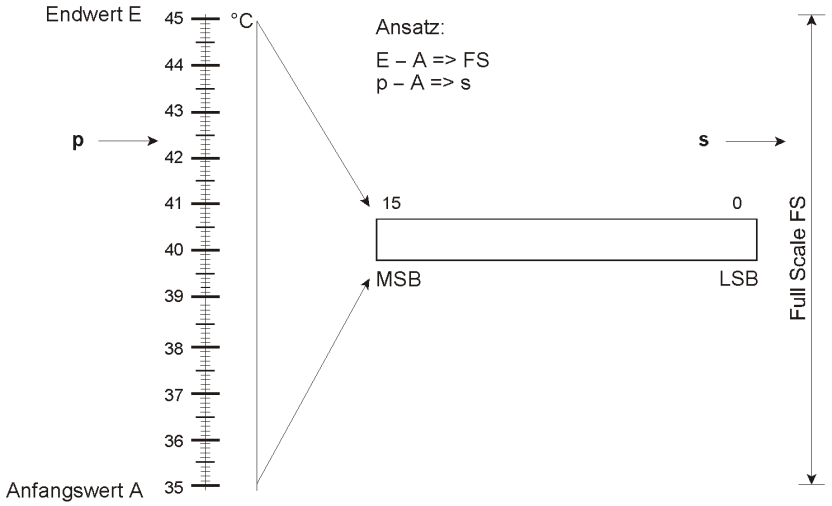


Abb. 3.18 Die anwendungsseitige Skalierung am Beispiel des Fieberthermometers.

3.1.8 Zeichen und Zeichencodes

Zeichen – Buchstaben, Ziffern usw. – werden als Bitmuster dargestellt (Zeichencode). Ein Zeichencode bezieht sich auf einen Zeichenvorrat (Zeichensatz, Character Set). Der Zeichenvorrat kann als geordnete Menge der zugehörigen Zeichen angesehen werden. Die laufenden Nummern der einzelnen Zeichen (mit anderen Worten, deren Ordinalzahlen) bilden den Zeichencode. Jedes Zeichen kann somit durch eine natürliche (vorzeichenlose) Binärzahl dargestellt werden (der Zahlenwert 0 entspricht dem ersten Zeichen, der Zah1enwert 1 dem zweiten Zeichen usw.). Umfasst der Zeichenvorrat n verschiedene Zeichen, muss der Zeichencode wenigstens ceil (ld n) Bitpositionen lang sein. Zeichencodes werden zumeist als Liste oder Matrix angegeben.

Alphanumerische Zeichen

"Alphanumerisch" ist der übliche Sammelbegriff für Buchstaben, Ziffern und Sonderzeichen. Braucht man nur die Groß- und Kleinbuchstaben des lateinischen Alphabets, die zehn Ziffern und die üblichen Satzzeichen (Punkt, Komma usw.), so sind mehr als 32, aber weniger als 64 Zeichen zu codieren. Für das einzelne Zeichen genügen also sechs Bits. Solche Codes wurden in den ersten Computern verwendet. Bald hat sich aber die Notwendigkeit ergeben, den

Zeichenvorrat zu erweitern (landesspezifische Zeichensätze, Sonderzeichen, Umlaute usw.). So ist man in den 60er Jahren des vorigen Jahrhunderts zum 8-Bit-Byte gekommen. Aber auch die 256 Zeichen, die man damit codieren kann, sind für die heutigen Bedürfnisse viel zu wenig.

Zeichencode und Zeichenform
Zeichencodes betreffen Zeichen, aber keine Zeichenformen. Zeichen (Characters) sind die kleinsten informationstragenden Einheiten in Texten (Buchstaben, Ziffern, Satzzeichen usw.). Zeichenformen (Glyphs) sind Bilddarstellungen von Zeichen, die angezeigt oder gedruckt werden können. Zusammengehörende Zeichenformen, die einem Zeichensatz zugeordnet sind, bilden eine Schrift (Font). Die Zeichencodes betreffen nur die abstrakten Zeichen, nicht aber deren bildmäßige Wiedergabe (Punktraster, Schriftarten, Schriftgrößen usw.).

Zeichencodes und Prozessorarchitektur
Die EDV-Anlagen der Vergangenheit waren für bestimmte Zeichencodes ausgelegt (beispielsweise EBCDIC oder ASCII). Neuere Prozessorarchitekturen unterstützen hingegen nur den Umgang mit elementaren Behältern, die Zeichencodes aufnehmen können, also mit einzelnen Bytes, 16-Bit-Wörtern usw. Alle Funktionen, die mit Zeichencodes und Zeichenketten zu tun haben, müssen ausprogrammiert werden.

ASCII
ASCII = American Standard Code for Information Interchange. Das ist der am weitesten verbreitete standardisierte Zeichencode im Bereich der Mikrocontroller und Mikroprozessoren. Der ursprüngliche ASCII-Code ist ein 7-Bit-Code (Wertebereich 0...127 oder 00H...7FH). Im Computer entspricht ein Zeichen einem Byte. Das höchstwertige Bit ist stets Null. Nur der Wertebereich von 20H bis 7EH ist mit darstellbaren Zeichen belegt (maximal 95 verschiedene Zeichen).

Unicode
Unicode ist ein internationaler Standard. Das grundsätzliche Ziel besteht darin, jedem auf der Erde gebrauchten Schriftzeichen ein Codewort (Fachbegriff: Code Point) zuzuordnen. Hierfür ist ein Wertebereich (Codespace) vorgesehen, der bei Null beginnt und bei 10FFFFH endet. Es stehen somit 1 114 112 Code Points zur Verfügung. DerUnicode-Standard beschreibt drei Codierungen: UTF-32, UTF-16 und UTF-8. Sie lassen sich wechselseitig ineinander umwandeln (UTF = Unicode Transformation Format).

UTF-32
Jeder Zeichencode wird durch ein 32-Bit-Wort dargestellt. Hierbei werden nur die Bitmuster im Wertebereich von 0 bis 10FFFFH belegt (Codespace). Da alle Codes gleich lang sind, ist die programmseitige Nutzung einfach, die Anforderungen an die Speicherkapazität sind aber hoch. Deshalb verwendet man gelegentlich zum Speichern UTF-16 oder UTF-8 und setzt die Codes erst dann in UTF-32 um, wenn Programmabläufe auszuführen sind, die einzelne Zeichen betreffen.

UTF-16
Die Codierung beruht auf 16-Bit-Worten. Es ist eine Weiterentwicklung der ersten Versionen des Unicode-Standards, in denen ausschließlich 16-Bit-Codes vorgesehen waren. Die 2^{16} = 65 536 Codewörter waren aber bald auch zu wenig. UTF-16-Zeichencodes bestehen aus einem einzigen 16-Bit-Wort für häufig gebrachte und aus zwei Wörten für seltener gebrauchte Zeichen. UTF-16 ist ein Kompromiss zwischen Speicherbedarf und Komplizertheit der Programmierung. Dieser Code wird deshalb in vielen Systemen bevorzugt.

UTF-8
Die Zeichen werden mit Bytes codiert. ASCII-Zeichen im Bereich von 00H bis 7FH brauchen nur ein Byte (ASCII-Transparenz). Die höchstwertige Bitposition dieser Bytes ist mit Null belegt. Die weiteren Zeichencodes belegen zwei, drei oder vier Bytes. Das erste Byte gibt an, wie viele Bytes nachfolgen. Anhand der höchstwertigen Bitpositionen kann man erkennen, ob es sich um ein erstes oder um eines der folgenden Bytes handelt. UTF-8 wird vor allem in HTML-Dateien und Internetprotokollen verwendet. UTF-8-Zeichenketten passen zu den Zeichenkettenkonventionen der Programmiersprache C. Keines der Bitmuster, die in längeren Zeichencodes auftreten, entspricht einem ACSII-Code. Somit hat auch kein anderes Byte außer dem ASCII-Zeichen NUL den Wert 00H, der in C das Ende der Zeichenkette angibt.

Zeichenketten
Zeichenketten (Character Strings) sind Aneinanderreihungen von Zeichen. Die architekturseitige Unterstützung von Zeichencodes und Zeichenketten ist ein typisches Merkmal der herkömmlichen EDV-Anlagen (Mainframes). Neuere Prozessorarchitekturen haben gar keine oder nur sehr elementare Vorkehrungen. Diese bestehen zumeist nur darin, die fortlaufende Adressierung zu unterstützen (Adresszählung). Alle anwendungsseitigen Elementarperationen mit Zeichenketten (Transportieren, Codes wandeln, Verknüpfen, Einfügen, Abschneiden, Verlängern, Verkürzen usw.[21] sind auszuprogrammieren.

Um mit einer im Speicher vorliegenden Zeichenkette arbeiten zu können, muß man wisssen, wo sie anfängt (Anfangsadresse) und wo sie aufhört bzw. wie lang sie ist. Es gibt mehrere Grundsatzlösungen, um die Länge einer Zeichenkette darzustellen (Abbildung 3.19)

a) Endekennung im Zeichen selbst. Beispiel: Der Zeichencode ist 7-Bit-ASCII. Eine Eins im 8. Bit des Bytes kennzeichnet das Ende der Zeichenkette.

b) Ein besonderes Zeichen als Endemarke. Die traditionellen EDV-Anlagen hatten verschiedene Codes für Endemarken. Heutzutage am meisten verbreitet ist wohl die Endemarke 00H (Programmiersprache C[22]).

c) Längenangabe am Anfang der Zeichekette. Typische Varianten:
   - Das erste Byte gibt die Länge an (Pascal).
   - Die ersten zwei oder vier Bytes geben die Länge an (z. B. Delphi).

d) Es gibt zwei Längenwerte, die Gesamtlänge und die aktuell belegte Länge.

e) Die Längenangabe steht in einem Deskriptor[23], der seinerseits auf die Zeichenkette verweist (die grundsätzlich beste Lösung).

21: Vgl. beispielsweise die Zeichenkettenfunktionen der Programmiersprache Basic.

22: Diese Lösung ergibt für ganz einfache Zeichenkettenoperationen den geringsten Programmieraufwand, ist aber ansonsten grundschlecht. Die Zeichenkettenunterstützung in C ist lausig (Gegenbeispiel: Basic).

23: Eine Datenstruktur, die andere Daten beschreibt. Etwas Ähnliches wie der Verzeichniseintrag eines Dateisystems, nur einfacher.

f) Die Längenangabe steht in den Maschinenbefehlen, die die Zeichenkettenoperationen ausführen, oder sie muß zuvor in ein Register geladen werden. Diese Lösungen wurden u. a. beim legendären EDV-System S/360 (IBM) gewählt[24].

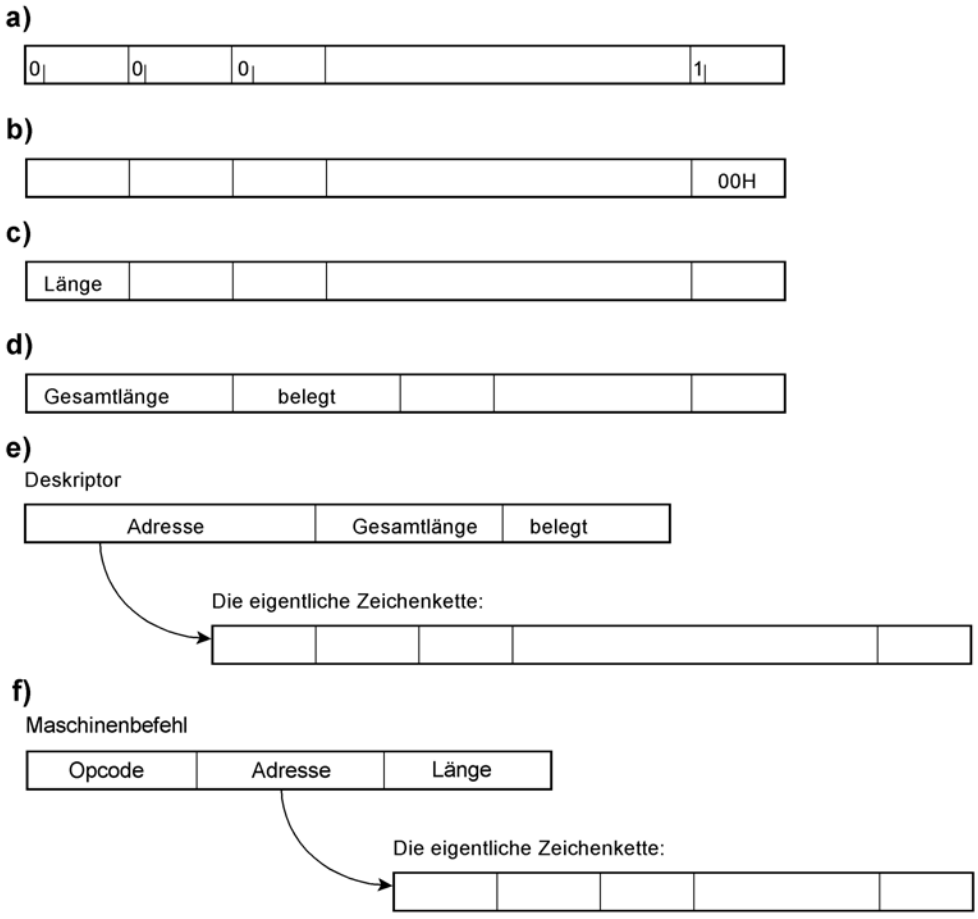


Abb. 3.19 So kann die Länge einer Zeichenkette dargestellt werden.

3.2 Elementaroperationen

3.2.1 Hardware und Software

Operationen sind Umformungen oder Verknüpfungen von Operanden. Sie können schaltungstechnisch oder programmseitig implementiert werden. Irgend etwas programmseitig – mit Software – zu erledigen hat aber zur Voraussetzung, dass schaltungstechnisch implementierte Operationen verfügbar sind, die freizügig aufgerufen werden können. Die heutigen Prozessorarchitekturen weisen einen gemeinsamen Vorrat an elementaren, universell nutzbaren Operationen auf, die eine Art Industriestandard darstellen. Sie sind im Grunde leicht verständlich. Die zugehörigen Schaltungen sind vergleichsweise einfach und kostengünstig.

24: Die Längenangabe im Befehl mag seinerzeit als revolutionär gegolten haben, ist aber SEHR häßlich (Übungsaufgabe: weshalb?). Immerhin hat IBM auch die Umgehungslösung geliefert, nämlich den sog. EXEC-Befehl. Trotzdem ist es unschön und umständlich.

Geht es um komplexere Operationen, so stehen wiederum die grundsätzlichen Alternativen zur Wahl: Schaltungslösung (Hardware) oder Programmablauf (Software). Zu beiden Alternativen gibt es eine reichhaltige Literatur. Es ist keineswegs falsch, sich in beiden Richtungen näher umzusehen. Letzten Endes geht es um Algorithmen, also um schematische Informationswandlungen. Es gibt aber typische Besonderheiten, die vor allem dann zu beachten sind, wenn man Erkenntnisse des einen Gebietes im jeweils anderen Gebiet anwenden möchte. Manche Forschungs- oder Entwurfsziele der einen Richtung sind in der anderen bedeutungslos.

Forschungsarbeiten mit dem Ziel der Schaltungsentwicklung beziehen sich darauf, dass am Ende tatsächlich eine Schaltung gebaut wird. Dabei hat man alles alles bis auf den einzelnen Taktzyklus und die einzelne Boolesche Gleichung unter Kontrolle, manchmal sogar bis auf den einzelnen Transistor. Die hier zu erreichenden Beschleunigungseffekte können in der Software offensichtlich gar nicht wirksam werden. Viele Informationswandlungen, die im Programm mehrere Anweisungen oder Maschinenbefehle benötigen, kann man direkt mit Schaltnetzen oder Funktionszuordnern erledigen. Manches erledigt sich sogar durch passendes Anschließen.

Forschungsarbeiten mit dem Ziel der Programmierung beziehen sich darauf, die Probleme mit gegebenen Befehlssätzen oder Sprachmitteln zu lösen. Für die Schaltungsentwicklung sind solche Erkenntnisse durchaus nicht wertlos. Womöglich kommt man mit vergleichsweise einfachen Verarbeitungswerken aus, die man nur durch eine optimierte Ablaufsteuerung ergänzen muss. Ein Ablauf in mehreren Schritten erfordert aber Register für die Zwischenergebnisse. Die Ablaufsteuerung stellt ein zusätzliches Entwurfsproblem dar.

In der Hardware gibt es beträchtliche Unterschiede (Aufwand, Verzögerungszeit) zwischen einer Addition und einem einfachen Transport (von einem Register zum anderen). In der Software dauern hingegen Addition und Transport zumeist gleich lang (es ist jeweils ein elementarer Befehl, der oftmals in einem einzigen Taktzyklus erledigt wird). Entscheidungen laufen in der Hardware praktisch parallel ab (Schaltnetze und Funktionszuordner können eine Vielzahl von Bedingungsignalen auf einmal auswerten, um den jeweiligen Folgezustand zu bestimmen). In der Software sind Entscheidungen hingegen mit bedingten Verzweigungen zu implementieren, die vergleichsweise viel Zeit kosten. Zu einer Zeit kann nur eine einzige Ja/Nein-Entscheidung getroffen werden. Verzweigungen dauern oftmals länger als Operationsbefehle (beispielsweise dauert der Verzweigungsbefehl zwei Taktzyklen, der Operationsbefehl aber nur einen Taktzyklus). Nicht selten dauert eine Entscheidung, mit der man eine Operation vermeiden könnte, länger als das Ausführen dieser Operation.

Problemlösung mit Software heißt im Grunde Problemlösung mit einem festen, vorgegebenen Ressourcenvorrat, der nur nach starren Regeln genutzt werden kann (Programmiermodell). Es ist die Frage, auf welchen Ressourcenvorrat man sich beschränkt. Begnügt man sich mit einem Minimum, setzt man eine gewissermaßen mittlere Ressourcenausstattung voraus oder stellt man sich eine zusammen, die man als besonders zweckmäßig ansieht? Gleich wie die Wahl ausfällt, man muss sich am Anfang festlegen und mit dem auskommen, wofür man sich entschieden hat; es nützt nichts, wenn man mitten in der Programmierung auf den Gedanken kommt, dass eigentlich schön wäre, noch ganz andere Befehlswirkungen und Sprachkonstrukte zur Verfügung zu haben. Problemlösung mit Hardware läuft hingegen darauf hinaus, (1) seine Ressourcen mit Gattern und Flipflops selbst zu bauen und sie (2) so zusammenzuschalten, wie man es für richtig hält. Tabelle 3.10 veranschaulicht diesen grundsätzlichen Unterschied anhand einiger einfacher Beispiele.

Problem	Programmseitige Lösung	Schaltungstechnische Lösung
Einen Binärvektor auf Null testen	Den Binärvektor mit sich selbst konjunktiv verknüpfen. Dabei bleiben alle Werte erhalten: $a \cdot a = a$. Enthält der Binärvektor ausschließlich Nullen, wird die Nullbedingung (Zero Flag ZF) gesetzt. Ansonsten wird sie gelöscht.	Ein NOR-Gatter anschließen.
Einen Binärvektor löschen	Den Binärvektor mit sich selbst antivalent verknüpfen oder von sich selbst subtrahieren: $a \oplus a = a - a = 0$.	Flipflops mit Rücksetzfunktion verwenden oder UND-Gatter in den Signalweg einfügen (Sperrwirkung)
Einen Binärvektor bitweise invertieren	XOR-Verknüpfung mit Einsen oder Einerkomplementbefehl (COM)	Signale an invertierte Ausgänge (z. B. von Flipflops) anschließen oder Negatoren zwischenschalten
In einem Binärvektor die niedrigstwertige Eins erkennen	Den Binärvektor mit dessen Zweierkomplement konjunktiv verknüpfen.	Einen Prioritätscodierer (Priority Encoder) anschließen
Bitanordnung wechseln, beispielsweise von Big Endian zu Little Endian	Erst alle benachbarten Bits gegeneinander tauschen, dann alle 2-Bit-Abschnitte, dann alle 4-Bit-Abschnitte usw. Getauscht wird durch Verdoppeln des Operanden, gegeneinander verschieben und ineinander einfügen.	So über Kreuz anschließen, wie es sein muss.
Herkömmliche schrittweise Division	Ist der Rest negativ, muss der ursprüngliche Wert des Dividendenabschnitts wieder hergestellt werden (Addition).	Ist der Rest negativ, wird der Dividendenabschnitt gar nicht verändert.

Tabelle 3.10 Problemlösungen im Vergleich.

3.2.2 Boolesche Operationen über Binärvektoren

Die einfachsten Booleschen ("logischen") Operationen betreffen adressierbare Behälter (Bytes, Wörter usw.), deren Inhalt als Binärvektor interpretiert wird. Das Ergebnis ist ein Binärvektor der gleichen Länge. Typischerweise wird zusätzlich erkannt, ob das Ergebnis gleich Null ist oder nicht (Nullbedingung; Zero Flag).

Invertierung (Einerkomplement)
Ein Binärvektor wird bitweise invertiert.

Verknüpfungen
Zwei Binärvektoren werden bitweise verknüpft. Üblich sind die elementaren Verknüpfungen UND, ODER und Antivalenz (XOR). Sie sind vielseitig nutzbar (Tabelle 3.11, Abbildung 3.20).

Bits setzen
Es wird eine ODER-Verknüpfung ausgeführt, wobei der zweite Operand an allen zu setzenden Bitpositionen Einsen enthält und sonst Nullen.

Bits löschen
Es wird eine UND-Verknüpfung ausgeführt, wobei der zweite Operand an allen zu löschenden Bitpositionen Nullen enthält und sonst Einsen.

Bits ändern
Um Bits in ihrem Wert zu ändern (von 0 nach 1 und umgekehrt), wird eine Antivalenzverknüpfung (XOR) ausgeführt, wobei der zweite Operand an allen zu ändernden Bitpositionen Einsen enthält und sonst Nullen.

Setzen	Löschen	Ändern
$a \vee 0 = a$ $a \vee 1 = 1$	$a \cdot 0 = 0$ $a \cdot 1 = a$	$a \oplus 0 = a$ $a \oplus 1 = \bar{a}$

Tabelle 3.11 Typische Anwendungen elementarer Boolescher Verknüpfungen.

Bits einfügen
Bits einfügen heißt, ein Bitmuster in ausgewählte Bitpositionen zu transportieren. Das erfordert zwei Verknüpfungen:

1. Löschen der Bitpositionen, in die das Bitmuster eingefügt werden soll, mittels einer UND-Verknüpfung deren zweiter Operand in allen zu löschenden Bitpositionen Nullen enthält und sonst Einsen.
2. Setzen gemäß dem einzufügenden Bitmuster mittels einer ODER-Verknüpfung, deren zweiter Operand an den betreffenden Bitpositionen die einzufügenden Werte enthält und sonst Nullen.

Abb. 3.20 Setzen, Löschen und Ändern anhand von Beispielen.

Bitfelder transportieren
Aus einem ersten Operanden (Quelloperanden) sind Bits aus bestimmten Bitpositionen zu entnehmen und in Bitpositionen eines zweiten Operanden (Zieloperanden) einzufügen. Hierzu müssen beide Operanden durch Löschen von Bitpositionen vorbereitet werden. Im Quelloperanden sind alle Bitpositionen zu löschen, deren Inhalt nicht übertragen werden soll, im Zieloperanden alle Bitpositionen, die Ziel der Übertragung sind. Das endgültige Bitmuster des Zieloperanden ergibt sich dann durch eine ODER-Verknüpfung der beiden so vorbereiteten Operanden. Befinden sich die zu übertragenden Bits nicht in gleichen Bitpositionen, so ist der Quelloperand passend zu verschieben.

Auf gelöschte oder gesetzte Bits testen
Um zu prüfen, ob bestimmte Bits alle gelöscht sind oder ob wenigstens eines dieser Bits gesetzt ist, braucht man eine UND-Verknüpfung, deren zweiter Operand in allen betreffenden Bitpositionen Einsen enthält und sonst Nullen. Sind alle so geprüften Bits gelöscht, so sind alle Ergebnisbits Null. Demgemäß wird die Nullbedingung (Zero Flag) gesetzt. Ist wenigstens eines der geprüften Bits gesetzt, wird die Nullbedingung gelöscht.

Bitmuster vergleichen
Um zu prüfen, ob zwei Bitmuster einander gleich sind oder nicht, werden sie bitweise antivalent verknüpft. Bei Gleichheit sind alle Ergebnisbits Null. Demgemäß wird die Nullbedingung (Zero Flag) gesetzt.

3.2.3 Einzelbitoperationen

Bittransporte
Zumeist beschränkt man sich auf zwei Einfachabläufe:

- Übertragen des Inhalts einer ausgewählten Bitposition in ein Bedingungsregister.
- Übertragen des Bedingungsbits (Flagbits) in eine ausgewählte Bitposition.

Es ist eine Art Industriestandard, die Übertragsbedingung (Carry Flag CF) für Bitoperationen ausnutzen. Da die gleiche Bedingung auch in das Verschieben und Rotieren einbezogen ist, können Bittransport- und Verschiebeoperationen problemlos zusammenwirken (Einschieben von ausgewählten Bits, Ausgeben von Bits, die aus einem Byte oder Wort herausgeschoben werden usw.).

Festwerte eintragen
In die ausgewählte Bitposition kann ein Festwert Null oder Eins eingetragen werden (Bit löschen, Bit setzen).

Bit testen (Bitabfrage)
Der Inhalt der ausgewählten Bitposition dient als Verzweigungs- oder Übersprungbedingung (Verzweigen/Überspringen, wenn Bit gelöscht oder gesetzt). Manchmal besteht die Testfunktion nur darin, ein Bedingungsbit zu stellen. Beispiel: IA-32. Hier kann der Transport ins Carry Flag (CF) mit dem Löschen, Setzen oder Invertieren der ausgewählten Bitposition verbunden werden.

Weitere Operationen
Manche Architekturen unterstützen das Invertieren von Bits, manche auch elementare Boolesche Verknüpfungen. Beispiel: 8051 (Tabelle 3.12).

Befehl		Wirkung
SETB	C	Carry Flag setzen
SETB	Bit	Ausgewähltes Bit setzen
CLR	C	Carry Flag löschen
CLR	Bit	Ausgewähltes Bit löschen
CPL	C	Carry Flag invertieren (0 wird 1, 1 wird 0)
CPL	Bit	Ausgewähltes Bit invertieren
MOV	C, Bit	Bitbelegung in das Carry Flag eintragen
MOV	C, /Bit	Invertierte Bitbelegung in das Carry Flag eintragen
ANL	C, Bit	UND-Verknüpfung $C := C \cdot Bit$
ANL	C, /Bit	UND-Verknüpfung $C := C \cdot \overline{Bit}$
ORL	C, Bit	ODER-Verknüpfung $C := C \vee Bit$
ORL	C, /Bit	ODER-Verknüpfung $C := C \vee \overline{Bit}$
JC	rel	Verzweigen, wenn Carry Fag gesetzt
JNC	rel	Verzweigen, wenn Carry Flag gelöscht
JB	Bit, rel	Verzweigen, wenn ausgewähltes Bit gesetzt
JNB	Bit, rel	Verzweigen, wenn ausgewähltes Bit gelöscht
JBC	Bit, rel	Verzweigen, wenn ausgewähltes Bit gesetzt; Bit löschen

Tabelle 3.12 Die Einzelbitbefehle der 8051-Mikrocontroller. Ein Beispiel für eine vergleichsweise reichhaltige Ausstattung mit Einzelbitoperationen. C - Übertragsbedingung (Carry Flag); Bit - Bitadresse; rel - relative Verzweigungsadresse (auf den Befehlszähler bezogen).

Die Bitauswahl
In den meisten Architekturen sind die Bitauswahlangaben nur Direktwerte (beispielsweise Bitposition b in Register r). Der Auswahlbereich ist oftmals auf bestimmte Register, E-A-Ports oder Speicherbereiche beschränkt. Einige Architekturen unterstützen die indirekte Bitadressierung (Bitadresse in Register). Beispiel: IA-32. Es kann ein einzelnes Bit in einer Bitkette von 4G – 1 Bits Länge ausgewählt werden, die an jeder beliebigen Byteadresse beginnen kann.

3.2.4 Bitfeldoperationen

Entnehmen und Einfügen

Ein Bitfeld ist ein Ausschnitt aus einem Binärvektor. Die einfachsten Bitfeldoperationen dienen dazu, Bitfelder zum Rechnen bereitzustellen und Verarbeitungsergebnisse in Bitfelder einzutragen (Abbildung 3.21).

Entnehmen (Extract)
Das Bitfeld wird aus dem Quelloperanden entnommen und rechtsbündig gemäß der Verarbeitungsbreite bereitgestellt. Die verbleibenden Bitpositionen werden mit Nullen gefüllt (Nullerweiterung). Ein solcher Binärvektor kann als Binärzahl aufgefasst und beliebigen Operationen unterzogen werden.

Einfügen (Deposit)
Ein rechtsbündig bereitstehendes Bitfeld wird in einen Zieloperanden eingefügt. Die verbleibenden Bitpositionen des Zieloperanden werden nicht verändert.

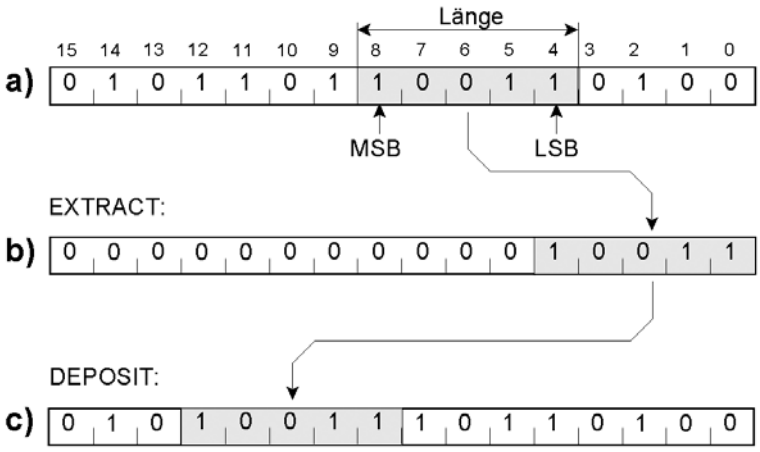


Abb. 3.21 Elementare Bitfeldtransporte. a) das ursprüngliche Bitfeld; b) Entnehmen (Extract); c) Einfügen (Deposit).

Transportieren
Das Bitfeld wird aus dem Quelloperanden entnommen und in einen Zieloperanden eingefügt (vergleiche die Bitfelder in Abbildung 3.21a und c). Die verbleibenden Bitpositionen des Zieloperanden werden nicht verändert.

Prioritätscodierung
Prioritätscodierung (Priority Encoding)[25] heißt, die Position der niedrigstwertigen oder der höchstwertigen Eins in einem Binärvektor oder Bitfeld aufzufinden. Das Ergebnis ist ein Indexvektor oder ein Bitindex (Abbildung 3.22). Wenn der Binärvektor keine einzige Eins enthält, ist auch der Indexvektor ausschließlich mit Nullen belegt. Dieser Sonderfall wird in Abbildung 3.22 durch das Gültigkeitsbit V vorgesehen.

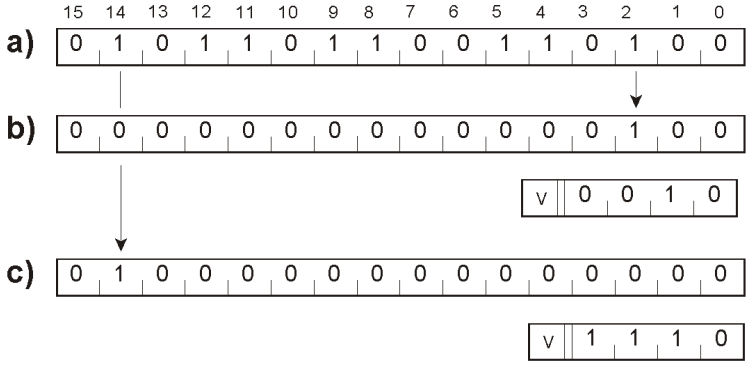


Abb. 3.22 Prioritätscodierung. a) Beispiel eines Binärvektors. b) die Position der niedrigstwertigen Eins (First Occurrence); c) die Position der höchstwertigen Eins (Last Occurrence). Das Ergebnis ist jeweils als Indexvektor und als Bitindex angegeben. V - Gültigkeitsbit. V = 0: keine einzige Eins; V = 1: Bitindex gültig (Wert 0 = Eins in Bitposition 0, Wert 1 = Eins in Bitposition 1 usw.).

25: Der Name erklärt sich aus typischen Anwendungen, wo es u. a. darum geht, die zuerst zu behandelnde Unterbrechungsbedingung oder den ersten freien Sektor auf einem Datenträger auszuwählen.

Die Anzahl der Einsen
Es wird gezählt, wie viele Einsen ein Binärvektor oder Bitfeld enthält (Quersumme, Number of Occurrences, Population Count, Sideways Addition).

3.2.5 Verschieben und Rotieren

Diese Operationen bewirken, dass die Operandenbits zu jeweils benachbarten Bitpositionen transportiert werden. Die Operationen unterscheiden sich in Hinsicht auf die Schieberichtung (nach links oder nach rechts) sowie in der Behandlung der hinausgeschobenen Bits und der freigewordenen Bitpositionen (Abbildung 3.23).

Schieberichtungen:

- Linksverschieben (Shift Left) = Verschieben in Richtung der höherwertigen Bitpositionen. Beim Linksverschieben um ein Bit gelangt jedes Bit in die jeweils höherwertige benachbarte Bitposition (Bit 6 nach Bitposition 7, Bit 5 nach Bitposition 6 usw.). Das höchstwertige Bit (MSB) wird hinausgeschoben. Die niedrigstwertige Bitposition (LSB) wird frei.
- Rechtsverschieben (Shift Right) = Verschieben in Richtung der niederwertigen Bitpositionen. Beim Rechtsverschieben um ein Bit gelangt jedes Bit in die jeweils niederwertige benachbarte Bitposition (Bit 1 nach Bitposition 0, Bit 2 nach Bitposition 1 usw.). Das niedrigstwertige Bit (LSB) wird hinausgeschoben. Die höchstwertige Bitposition (MSB) wird frei.

Verschieben
Die hinausgeschobenen Bits gehen verloren, die freigewordenen Bitpositionen werden aufgefüllt. Die typische Einfachlösung besteht im Auffüllen mit Nullen. Manche Architekturen unterstützen zudem das Auffüllen mit Bits, die von einem weiteren Operanden geliefert werden. Beispiel: IA-32.

Um mehr als ein Bit verschieben
Beim Verschieben um n Bits werden an einem Ende n Bits hinausgeschoben und am anderen Ende n Bitpositiponen frei. Beim Linksverschieben kommt das Bit von Position a in die Bitposition a + n, beim Rechtsverschieben in die Bitposition a – n. (Beispielsweise gelangt beim Linksverschieben um zwei Bits Bit 5 nach Bitposition 7, Bit 4 nach Bitposition 6 usw.).

Arithmetisches Rechtsverschieben
Beim arithmetischen Rechtsverschieben wird nicht die Null oder ein Füllwert, sondern das Vorzeichen (also der Inhalt der höchstwertigen Stelle) in alle frei werdenden Stellen eingetragen (Vorzeichenerweiterung).

Rotieren
Beim Rotieren werden die zum einen Ende hinausgeschobenen Bits am jeweils anderen Ende wieder zurückgeführt (Wrap Around). Rotieren ist also ein zyklisches Verschieben in den Grenzen der jeweiligen Verarbeitungsbreite. Wird beispielsweise ein Byte um ein Bit nach links rotiert, gelangt Bit 7 nach Bitposition 0, bei Rechtsrotation entsprechend Bit 0 nach Bitposition 7.

Ein typische Einfachlösung
Das Verschieben um mehrere Bits erfordert einen vergleichsweise hohen Schaltungsaufwand. Deshalb haben viele Prozessoren nur Befehle zum Verschieben und Rotieren um ein Bit.

Abbildung 3.24 veranschaulicht eine universelle Auslegung (Beispiel: AVR). Das herausgeschobene Bit wird in ein Bedingungflipflop übernommen, und zwar üblicherweise in die Übertragsbedingung (Carry Flag CF). Beim Verschieben wird die freigewordene Bitposition mit einer Null aufgefüllt. Beim Rotieren gehört die Übertragsbedingung zum Schiebeweg, der somit um eine Bitposition länger wird. Der bisherige Wert wird eingeschoben; das am anderen Ende herausgeschobene Bit wird in die Übertragsbedingung übernommen.

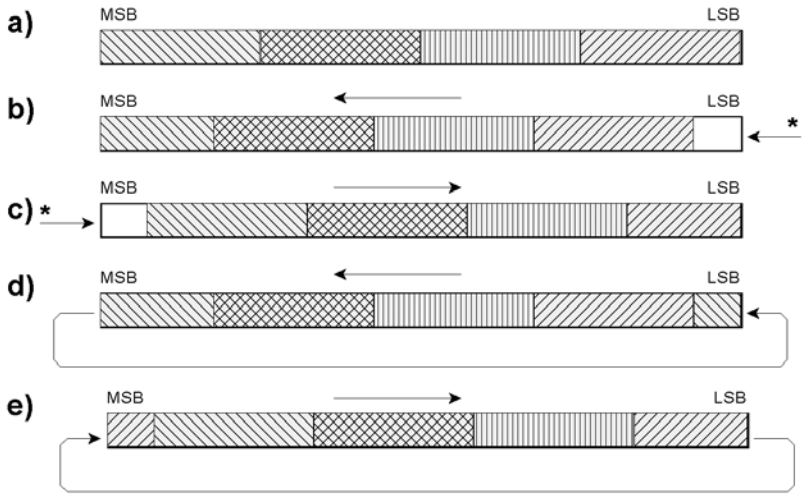


Abb. 3.23 Verschieben und Rotieren. a) Beispiel eines Binärvektors. Die vier Felder sollen beliebige Bitmuster andeuten. b) Linksverschieben; c) Rechtsverschieben; d) Linksrotieren; e) Rechtsrotieren. *: Auffüllen mit Füllwerten (beispielsweise mit Nullen oder mit Bits aus einem weiteren Operanden).

Längere Operanden verschieben und rotieren

Ein Verschieben über mehrere Maschinenwörter beginnt mit einem Verschiebebefehl, dem Rotationsbefehle für die weiteren Wörter nachfolgen. Hierdurch gelangt das aus dem vorhergehenden Wort herausgeschobene Bit über die Übertragsbedingung (CF) in die freigewordene Bitposition (Abbildung 3.25).

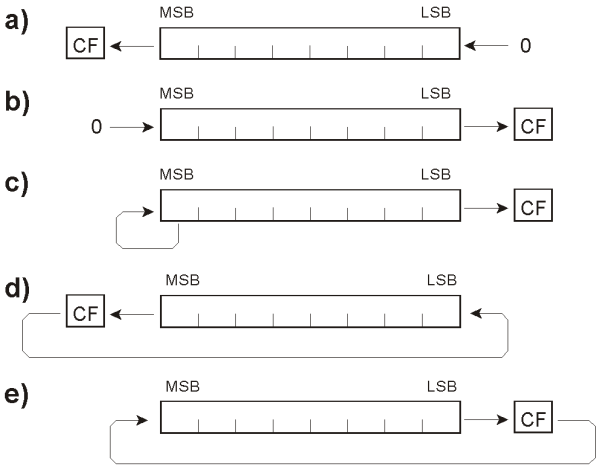


Abb. 3.24 Eine universelle Einfachauslegung der Verschiebe- und Rotationsoperationen. a) Linksverschieben; b) Rechtsverschieben; c) arithmetisches Rechtsverschieben; d) Linksrotieren; e) Rechtsrotieren; CF = Übertragsbedingung (Carry Flag).

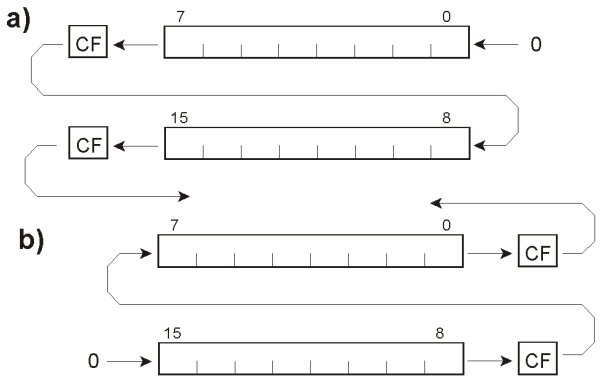


Abb. 3.25 Längere Operanden verschieben. a) Linksverschieben; b) Rechtsverschieben. Das Linksverschieben beginnt mit dem niedrigstwertigen Maschinenwort, das Rechtsverschieben mit dem höchstwertigen. Die herausgeschobenen Bits werden über das Carry Flag CF in die jeweils nachfolgenden Maschinenwörter weitergegeben.

3.2.6 Mit Binärzahlen rechnen

Die Zweierkomplementdarstellung als Industriestandard

Die Binärzahl ist eine Bitkette, deren Bitpositionen Stellenwerte zugeordnet sind. Die Stellenwerte sind Zweierpotenzen. Die höchstwertige Bitposition kann als höchster Stellenwert oder als Vorzeichen interpretiert werden. Die elementare Rechenoperation ist die Addition. Sie betrifft stets sämtliche Bitpositionen. Die Subtraktion wird als Addition des Zweierkomplements ausgeführt. Das Zweierkomplement wird durch bitweise Invertierung des Operanden gebildet; die zu addierende Eins läuft als Eingangsübertrag in das Addierwerk ein. Auch die Vorzeichenbits werden in diesen Rechenweg einbezogen. Abbildung 3.26 veranschaulicht ein typisches Rechenwerk. Alle am Markt erhältlichen Prozessoren arbeiten so, die Entwicklungssysteme synthetisieren solche Schaltungen, und die Zellen der FPGAs enthalten Vorkehrungen, die deren Implementierung unterstützen. Deshalb genügt es, sich auf diese Zahlendarstellung zu beschränken.

Ganze und natürliche Binärzahlen beim Addieren und Subtrahieren

Addition und Subtraktion laufen für natürliche und ganze Binärzahlen gleichermaßen ab. Es werden auch dann korrekte Resultate gebildet, wenn man einen Operanden als natürliche und den anderen als ganze Binärzahl interpretiert (eine typische Anwendung: die Adressrechnung). Das Rechnen mit den verschiedenen Zahlenarten unterscheidet sich nur darin, wie man das Ergebnis interpretiert und wie die Bedingungssignale ausgewertet werden. Deshalb braucht man keine besonderen Additions- und Subtraktionsbefehle für natürliche (vorzeichenlose) und ganze Binärzahlen.

Bereichsgrenzen

Ein Binärvektor von n Bits Länge kann nur Zahlen in einem begrenzten Wertebereich darstellen. Beim Rechnen können sich jedoch Werte ergeben, die außerhalb des jeweiligen Bereichs liegen. Diese Tatsache wird über Bedingungsbits (Flagbits) signalisiert.

Bedingungssignale (Flagbits)

Beim Addieren und Subtrahieren von Binärzahlen werden typischerweise die in Tabelle 3.13 angegebenen Bedingungssignale gebildet.

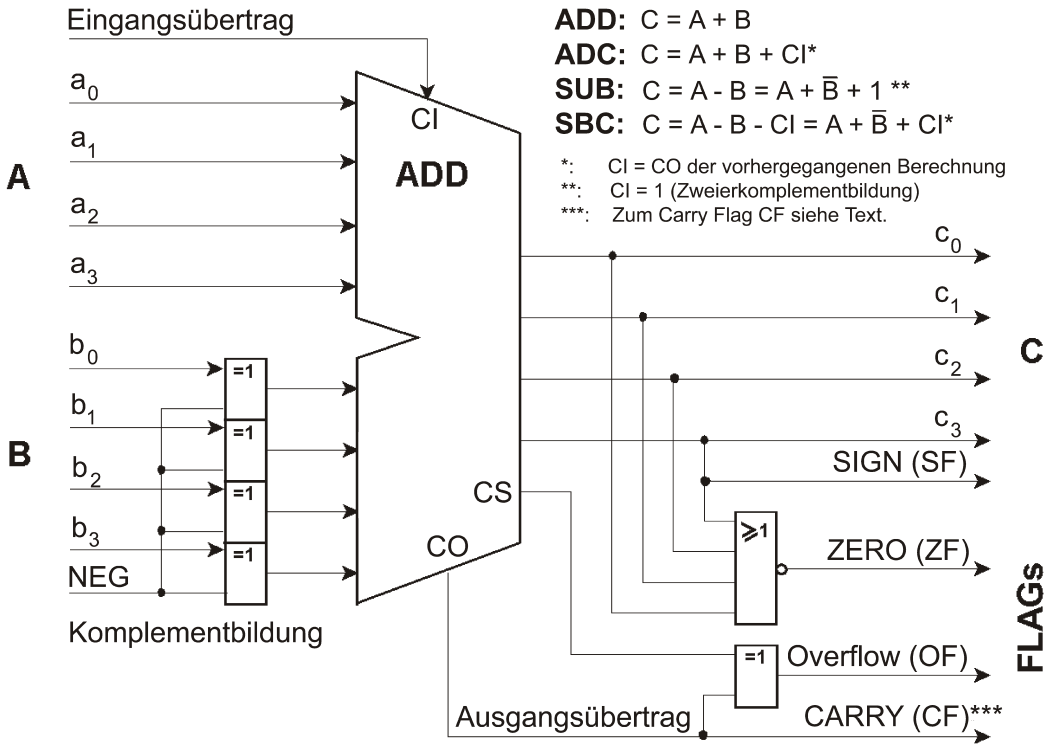


Abb. 3.26 Das elementare Rechenwerk – ein Industriestandard. Es beruht auf der binären Addition und der Zweierkomplementbildung. Die Aufgabe der Prozessorarchitektur besteht im Grunde darin, eine solche Schaltung mit Arbeit zu versorgen.

Bezeichnung		Bedeutung
Zero Flag	ZF	Ergebnis = 0
Carry Flag	CF	Ausgangsübertrag; Resultat außerhalb des Wertebereichs (siehe Text)
Overflow Flag	OF	Overflow = Ausgangsübertrag ≠ Übertrag in die Vorzeichenstelle = Ausgangsübertrag ⊕ Übertrag in die Vorzeichenstelle
Sign Flag	SF	Vorzeichen (höchstwertige Ergebnisbitposition). 0 = positiv, 1 = negativ

Tabelle 3.13 Typische Bedingungssignale (Flagbits).

Natürliche (vorzeichenlose) Binärzahlen

Der Wertebereich kann nur beim Addieren überschritten werden. Dann entsteht ein Ausgangsübertrag (Carry Out CO). Ist das eigentliche (mathematische) Rechenergebnis gleich r (ist also $r \geq 2^n$), so hat das Ergebnisbitmuster den Wert $r - 2^n$.

Der Wertebereich kann nur beim Subtrahieren unterschritten werden. Wird A – B gerechnet, so liegt das Ergebnis im Wertebereich, wenn A ≥ B ist. Dann wird auch ein Ausgangsübertrag CO abgegeben. Entsteht kein Ausgangsübertrag, so unterschreitet das Ergebnis den Wertebereich (A < B). Das eigentliche (mathematische) Ergebnis ist negativ und somit im Bereich der natürlichen Zahlen nicht zulässig. Ist das eigentliche Ergebnis eine negative Zahl – r, so entspricht das Ergebnisbitmuster dem Zweierkomplement des eigentlichen Ergebnisses.

Ausgangsübertrag und Übertragsbedingung (Carry Flag CF)
Wenn man die Operaden als vorzeichenlose Binärzahlen interpretiert, so zeigt der Ausgangsübertrag CO der Zweierkomplementrechnung an, ob sich das Ergebnis im Wertebereich befindet oder nicht. Die Interpretation des Signals richtet sich aber nach der jeweiligen Rechenoperation:

- Beim Addieren liegt das Ergebnis außerhalb des Wertebereichs, wenn CO = 1 ist (Bereichsüberschreitung).
- Beim Subtrahieren liegt das Ergebnis außerhalb des Wertebereichs, wenn CO = 0 ist (Bereichsunterschreitung).

Um dem Programmierer eine einfach auszuwertende Bedingung bereit zu stellen, wird in vielen Architekturen die Übertragsbedingung CF in Abhängigkeit von der Rechenoperation gebildet. Beim Addieren entspricht CF dem Ausgangsübertrag CO, beim Subtrahieren wird CO invertiert.

$$CF = CO \oplus SUB - Befehl$$

Dann bedeutet CF = 1 immer, dass das Ergebnis außerhalb des Wertebereichs liegt. Diese Konvention ist eine Art Industriestandard.

Ganze Binärzahlen
Ob der der Wertebereich über- oder unterschritten wird, ist anhand des Ausgangsübertrags CO und des in die Vorzeichenstelle einlaufenden Übertrags CS erkennbar:

- Das Resultat liegt im Wertebereich, wenn (1) weder ein Übertrag in die Vorzeichenstelle noch ein Ausgangsübertrag auftreten, oder wenn (2) diese beiden Überträge gleichzeitig auftreten.
- Das Resultat liegt außerhalb des Wertebereichs, wenn nur einer der beiden Überträge auftritt.
- Wird die größte positive Zahl überschritten, so entsteht nur ein Übertrag in die Vorzeichenstelle, aber kein Ausgangsübertrag.
- Wird die kleinste negative Zahl unterschritten, so entsteht nur ein Ausgangsübertrag, aber kein Übertrag in die Vorzeichenstelle.

Allgemein wird das Verlassen des Wertebereichs ganzer Zahlen als Überlauf (Overflow) bezeichnet.

Binärzahlen vergleichen
Zwei Binärzahlen können durch Subtrahieren miteinander verglichen werden. Aus den Tabellen 3.14 und 3.15 sind die jeweils auszuwertenden Bedingungen ersichtlich.

Rechnen mit längeren Binärzahlen
Sind die Zahlen länger als die Verarbeitungsbreite, so kann man sie – mit den niederwertigen Stellen beginnend – abschnittsweise addieren oder subtrahieren (Abbildung 3.27). Der Ausgangsübertrag muss dabei gespeichert und bei der nachfolgenden Addition oder Subtraktion wieder als Eingangsübertrag verwendet werden. Die meisten Prozessoren haben eigens Befehle zum Addieren und Subtrahieren mit Eingangsübertrag (ADD WITH CARRY ADC, SUBTRACT WITH CARRY SBC).

Vergleichsaussage	Bedingung	Flagbits	Typische Bezeichnung der Verzweigungsbedingung
A = B	Ergebnis = 0	ZF = 1	Equal
A ≠ B	Ergebnis ≠ 0	ZF = 0	Not Equal
A < B	Kein Ausgangs-übertrag (Ergebnis negativ)	CO = 0; CF = 1	Below
A > B	Ergebnis ≠ 0 und Ausgangsübertrag (Ergebnis positiv)	$\overline{ZF} \cdot CO$; $\overline{ZF \vee \overline{CO}}$ $\overline{ZF} \cdot \overline{CF}$; $\overline{ZF \vee CF}$;	Above
A ≤ B	Ergebnis = 0 oder kein Ausgangsübertrag (A nicht größer als B)	$ZF \vee \overline{CO}$; $ZF \vee CF$	Below or Equal
A ≥ B	Ausgangsübertrag (A nicht kleiner als B)	CO = 1; CF = 0	Above or Equal

Tabelle 3.14 Vergleichsbedingungen beim Subtrahieren natürlicher (vorzeichenloser) Binärzahlen. Rechengang A – B. CO = Ausgangsübertrag des Zweierkomplementaddierers, CF = Carry Flag (CO ⊕ SUB).

Vergleichsaussage	Bedingung	Flagbits	Typische Bezeichnung der Verzweigungsbedingung
A = B	Ergebnis = 0	ZF = 1	Equal
A ≠ B	Ergebnis ≠ 0	ZF = 0	Not Equal
A < B	Ergebnis negativ und kein Überlauf oder positiv und Überlauf	$OF \oplus SF$	Less
A > B	Ergebnis ≠ 0 und negativ und Überlauf oder positiv und kein Überlauf (A nicht kleiner als B und nicht gleich B)	$\overline{ZF} \cdot \overline{(OF \oplus SF)}$; $\overline{ZF \vee (OF \oplus SF)}$	Greater
A ≤ B	Ergebnis = 0 oder negativ und kein Überlauf oder positiv und Überlauf (A nicht größer als B)	$ZF \vee (OF \oplus SF)$	Less or Equal
A ≥ B	Ergebnis negativ und Überlauf oder Ergebnis positiv und kein Überlauf (A nicht kleiner als B)	$\overline{OF \oplus SF}$	Greater or Equal

Tabelle 3.15 Vergleichsbedingungen beim Subtrahieren ganzer (vorzeichenbehafteter) Binärzahlen durch Addieren des Zweierkomplements. Rechengang A – B. ZF - Ergebnis = 0; OF - Overflow; SF - Vorzeichen (höchstwertige Stelle des Ergebnisses).

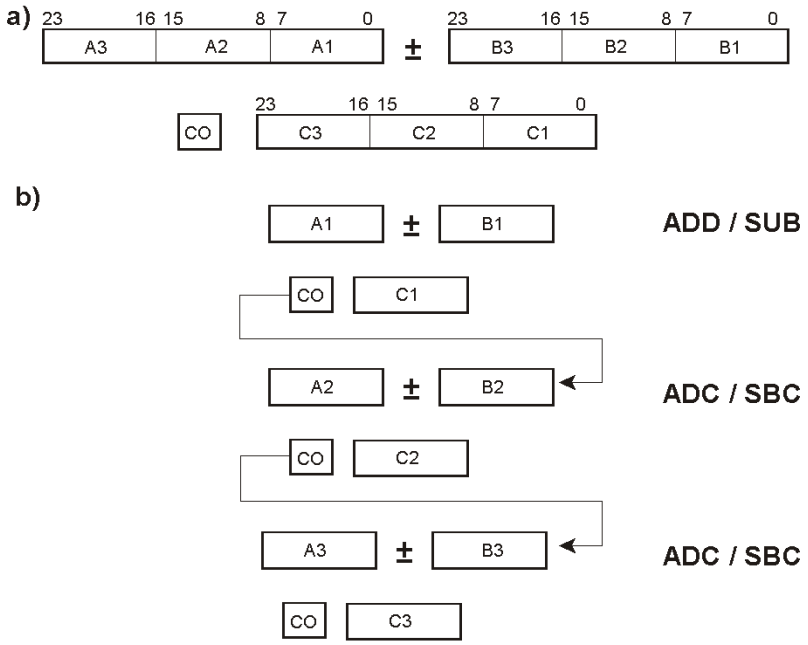


Abb. 3.27 Addieren und Subtrahieren längerer Binärzahlen. a) Rechenbeispiel C := A ± B. Zwei 24 Bits lange Zahlen ergeben ein Ergebnis von 24 Bits Länge sowie den Ausgangsübertrag CO. b) Ergebnisbildung mit einer Verarbeitungsbreite von 8 Bits. Es sind drei Schritte erforderlich. Vom zweiten Schritt an wird der Ausgangsübertrag des vorhergehenden Schrittes als Eingangsübertrag wirksam.

Multiplizieren und Dividieren

Im Gegensatz zum Addieren und Subtrahieren kommt es hier auf den Datentyp an; man braucht besondere Multiplikations- und Divisionsbefehle für vorzeichenlose (natürliche) und vorzeichenbehaftete (ganze) Zahlen.

Aus Multiplikand und Multiplikator ergibt sich ein Produkt doppelter Länge. Die Division ist üblicherweise als Umkehrung der Multiplikation ausgelegt. (A • B) : B muss wieder A ergeben und (A • B) : A wieder B. Der Dividend ist doppelt so lang wie der Divisor. Quotient und Rest haben jeweils die Länge des Divisors. Manche Divisionsbefehle hinterlassen den Quotienten und den Rest (Reminder) in zwei programmseitig zugänglichen Registern. Oftmals gibt es aber getrennte Befehle zur Quotienten- und zur Restberechnung (DIV, REM). Ist der Rest ungleich Null, so ist der Quotient ein Ergebniswert, der in Richtung Null auf die nächste ganze Zahl gerundet wurde. Beispiel 19 : 5 = 3 Rest 4. Genaues Ergebnis: 3,8.

Das herkömmliche Registermodell der Multiplikation und Division

Die Operationen beziehen sich auf zwei Register, den Akkumulator (AC) und das Multiplikator-Quotienten-Register (MQ-Register). In manchen Architekturen sind diese Register fest vorgesehen, in anderen werden zwei Universalregister gemäß diesem Schema ausgenutzt oder es wird ein Universalregister durch ein fest zugeordnetes weiteres Register ergänzt.

Beim Multiplizieren wird der Multiplikator im MQ-Register erwartet (Abbildung 3.28a).

Beim Dividieren steht die höherwertige Hälfte der doppelt langen Dividenden im Akkumulator und die niederwertige im MQ-Register (Abbildung 3.28b).

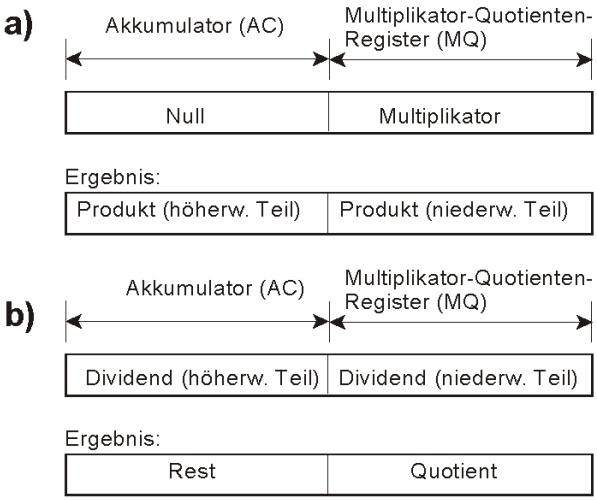


Abb. 3.28 Herkömmliche Registermodelle a) Multiplikation, b) Division.

Ergebnisbitmuster beim Multiplizieren
Die vorzeichenlose und ganzzahlige Multiplikation zweier Operandenbitmuster liefert ein gleiches Bitmuster in der niederwertigen Hälfte des Ergebnisses (Abbildung 3.29). Anwendung: beispielsweise bei der Adressrechnung. Manche Prozessoren unterstützen nur die ganzzahlige Multiplikation, weil bei der Adressrechnung in der höherwertigen Ergebnishälfte ohnehin eine Null herauskommen muß (denn sonst würde sich eine zu lange Adresse ergeben).

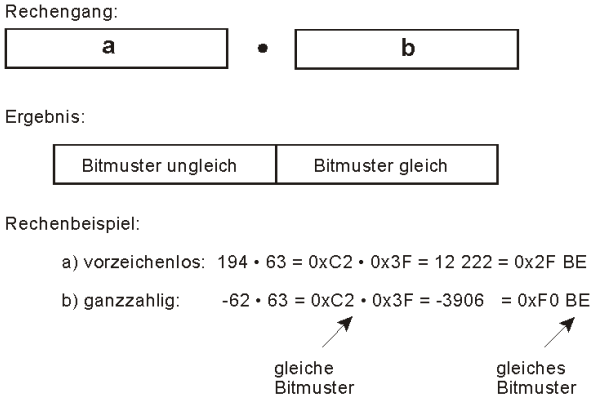


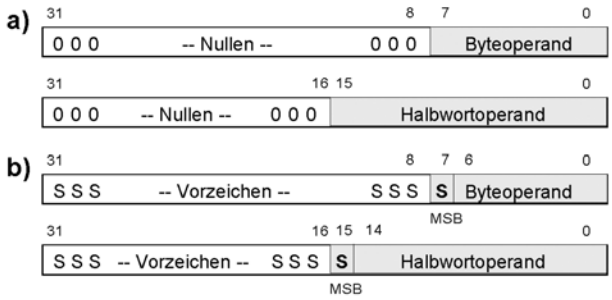
Abb. 3.29 Ergebnisbitmuster beim vorzeichenlosen und ganzzahligen Multiplizieren.

Quotient und Divisor
Bei doppelt langem Dividenden darf der Quotient nicht größer werden als der Divisor. Typische Wertebereiche des Quotienten (am Beispiel der vorzeichenlosen Division):

- 16 Bits : 8 Bits : 0...255,
- 32 Bits : 16 Bits: 0...65 535,
- 64 Bits : 32 Bits: $0...2^{32}-1$.

Es ist aber ohne weiteres möglich, größere Quotienten zu erhalten. Der Grenzfall: Dividieren durch Eins. Dann ergibt sich ein Quotient, der dem Divisor entspricht. Manche Maschinen reagieren in solchen Fällen mit einer Ausnahmebedingung. Es kommt dann vor, daß das Laufzeitsystem daraufhin eine Fehlermeldung “Divison durch Null” anzeigt...

Rechnen mit kürzeren Binärzahlen
Sind die Zahlen kürzer als die Verarbeitungsbreite, so müssen sie erweitert werden. Hierbei kommt es auf den Datentyp an. Vorzeichenlose Zahlen werden durch Nullerweiterung, vorzeichenbehaftete durch Vorzeichenerweiterung verlängert. Abbildung 3.30 veranschaulicht typische Erweiterungsaufgaben anhand einer Verarbeitungsbreite (= Wortlänge) von 32 Bits. Die zu erweitenden Operanden sind acht oder 16 Bits lang (Bytes, Halbwörter).



a) Nullerweiterung (Zero Extension). Binärvektoren und vorzeichenlose Binärzahlen werden mit Nullen erweitert.
b) Vorzeichenerweiterung (Sign Extension). Ist eine ganze Binärzahl kürzer als die Verarbeitungsbreite, so ist das Vorzeichen – also das höchstwertige Bit (MSB) – in alle höherwertigen Stellen einzufügen.

Abb. 3.30 Typische Erweiterungsaufgaben.

3.3 Maschinenbefehle

3.3.1 Der Befehlssatz

Der Befehlssatz (Instruction Set) ist die Menge aller Befehle, die in einer Prozessorarchitektur vorgesehen sind. Die von den Befehlen ausgelösten Abläufe sind vergleichsweise elementar. Die meisten Befehle leisten viel weniger als die typischen Anweisungen in höheren Programmiersprachen.

Minimallösungen
Wie umfangreich ein Befehlssatz sein soll, ist eine Streitfrage seit es Computer gibt. Die Theorie der Berechenbarkeit – ein Teilgebiet der mathematischen Grundlagenforschung – kennt fiktive Maschinen mit extrem einfachen Befehlen. Diese Erkenntnisse kann man nicht ohne weiteres auf die Praxis übertragen, weil sich bei Beschränkung auf ganz elementare Befehlswirkungen sehr große Programme und lange Laufzeiten ergeben würden. Eine praxisgerechte Minimallösung kann beispielsweise mit folgenden Befehlswirkungen auskommen:

- Addieren und Subtrahieren von Binärzahlen in Zweierkomplement-darstellung,
- Verschieben und Rotieren,
- bitweise wirkende logische Operationen (NICHT, UND, ODER, XOR),
- bedingte Verzweigungen auf die Bedingungen Ausgangsübertrag (Carry Flag CF), Überlauf (Overflow Flag OF) und Ergebnis Null (Zero Flag ZF),
- unbedingte Verzweigung,
- Unterprogrammruf und Rückkehr,
- Transporte (Laden = Operanden aus dem Speicher holen, Speichern = Ergebnisse in den Speicher schreiben).

Diese Befehlswirkungen werden durch die indirekte Adressierung als einfachste und vielseitigste Form der Adressrechung ergänzt. Manche Mikrocontroller sind nach solchen Gesichtspunkten ausgelegt. Es sind nur die nötigsten Befehlswirkungen und Zubringerfunktionen implementiert. Alles weitere ist programmseitig zu erledigen. Dafür geht es schnell, und der Schaltkreis kostet nicht viel (Beispiel: PIC16).

Typische Praxislösungen
Die Vorstellung, mit möglichst wenigen Befehlswirkungen auskommen zu müssen, ist beim heutigen Stand der Technik bedeutungslos. Auch RISC bedeutet nicht, so wenige Befehle wie möglich vorzusehen, sondern die Befehlswirkungen nicht komplexer auszulegen, als es seitens der Anwendungspraxis erforderlich ist, wobei vorausgesetzt wird, dass die Anwendungsprogramme in höheren Programmiersprachen geschrieben werden. So können auch Architekturen mit "reduziertem" Befehlssatz mehr als einhundert verschiedene Befehle haben. Die Befehlssätze der heutigen Architekturen sind auf der Grundlage umfangreicher statistischer Analysen optimiert worden (man hat tausende Programme daraufhin untersucht, wie häufig bestimmte Befehle verwendet werden, wozu sie vorgesehen sind, wie oft sie ausgeführt werden, welchen Einfluss sie auf die Programmlaufzeit haben usw.). Die Philosophie läuft darauf hinaus, jene Funktionen architekturseitig und schaltungstechnisch zu unterstützen, die besonders häufig benötigt werden oder die für das Leistungsvermögen im jeweiligen Anwendungsbereich von entscheidender Bedeutung sind. Alles, was seltener vorkommt oder etwas mehr Zeit hat, muss programmseitig erledigt werden.

Typische Befehlswirkungen im Überblick
Faustregel: Die Wirkungen sind überall gleich, nur die Verpackung (Befehlsformate) und die Zubringerfunktionen (Adress- und Registermodell, Speicherverwaltung usw.) machen den Unterschied. Die Tabellen 3.16 bis 3.18 geben einen Überblick über die Befehlsausstattung typischer Rechnerarchitekturen.

3.3.2 Befehlsabläufe

Befehle, die nacheinander auszuführen sind, werden im Speicher aufeinander folgend angeordnet. Der jeweils auszuführende Befehl wird vom Befehlszähler (Instruction Counter IC)[26] adressiert, aus dem Speicher gelesen und ins Befehlsregister (Instruction Register IR) geladen. Im Bitmuster des gelesenen Befehls ist dessen Länge codiert. Um den Folgebefehl zu adressieren, wird der Inhalt des Befehlszählers um den Wert der Befehlslänge erhöht. Sind alle Befehle gleich lang, so kann der Befehlszähler ein einfacher Zähler sein. Wird von der Befehlsfolge abgewichen (Verzweigung, Unterprogrammruf, Unterbrechung), so wird die jeweilige Adresse in den Befehlszähler geladen (Abbildung 3.31).

26: Auch als Instruction Pointer IP oder Program Counter PC bezeichnet.

Binärzahlen		**Gleitkommazahlen**	**Dezimalzahlen (BCD)**
natürliche	**ganze**		
Addieren		Addieren	nicht unterstützt
Subtrahieren		Subtrahieren	*bzw.*
Vergleichen		Multiplizieren	Dezimalkorrektur (Hilfsbefehle)
Multiplizieren	Multiplizieren	Dividieren	*bzw. volle Unterstützung:*
Dividieren	Dividieren	Vergleichen	Addieren
Verschieben	Verschieben (arithmetisch)	Betrag	Subtrahieren
	Vorzeichenwechsel	Vorzeichenwechsel	Multiplizieren
		Wandeln (Konvertieren)	Dividieren
		weitere mathematische Funktionen, wie $\sqrt{x}$, sin x usw.	Vergleichen
			Wandeln (Konvertieren)

Tabelle 3.16 Verarbeitungsbefehle für numerische Daten.

Adressierbare Behälter (Bytes, Worte usw.)	**Zeichenketten**	**Bitketten, Bitfelder**	**Einzelbits**
UND	Auffüllen	Bereitstellen (rechtsbündig)	Abfragen
ODER	Ausschneiden	Einfügen (aufs Bit adressiert)	Setzen
NICHT	Einfügen	Position der niedrigstwertigen Eins	Löschen
Exklusiv-ODER (XOR)	Vergleichen	Position der höchstwertigen Eins	Wechseln (0 => 1, 1 => 0)
Vergleichen (logisch)	Durchsuchen	Anzahl der Einsen	
Verschieben / Rotieren	über Tabelle wandeln		

Tabelle 3.17 Verarbeitungsbefehle für nichtnumerische Daten.

Transportbefehle			
Laden (Speicher => Register)		Umladen (Register => Register)	
Speichern (Register => Speicher)		Umspeichern (Speicher => Speicher)	
Programmsteuerbefehle			
Verzweigen, unbedingt		Unterprogrammruf	Systemruf, Wechsel der Privilegebene
Verzweigen, bedingt (auf Null, auf Übertrag, bei Gleichheit, bei Ungleichheit usw.)		Rückkehr aus Unterprogramm	Unterbrechung auslösen
Systembefehle			
Ein- und Ausgabe	Betriebsarten umschalten	Laden/Speichern von Systemregistern	Taskumschaltung
Unterbrechungssteuerung	Steuerung der Speicherverwaltung	Sonderzustände einleiten	Rückkehr aus Supervisorzustand
Hilfsbefehle für Test- und Fehlersuchzwecke			

Tabelle 3.18 Transport-, Programmsteuer- und Systembefehle.

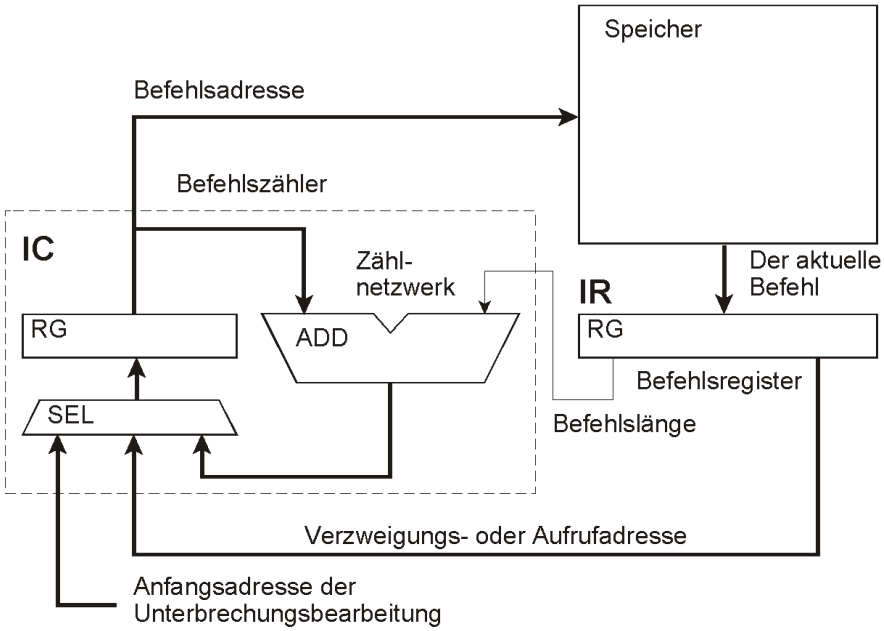


Abb. 3.31 Befehlsadressierung.

Die Befehle werden in mehreren aufeinanderfolgenden Phasen ausgeführt (Abbildung 3.32). Im klassischen Fall erfordert jede Phase wenigstens einen Maschinenzyklus. Weitere Maschinenzyklen können aus verschiedenen Ursachen hinzukommen:

- Wenn die Schaltmittel und Informationswege nicht alle beteiligten Bits parallel verknüpfen oder übertragen können (sparsame Auslegung der Hardware, schmale Datenwege usw.).
- Wenn die Abläufe so kompliziert sind, dass sie grundsätzlich nicht innerhalb eines einzigen Maschinenzyklus erledigt werden können (Beispiel: die Division).
- Wenn der Prozessor aufgrund der Systemauslegung (z. B. Bestückung mit langsameren Speicherschaltkreisen) oder aufgrund besonderer Betriebsumstände auf andere Einrichtungen warten muss (Wartezustände).
-

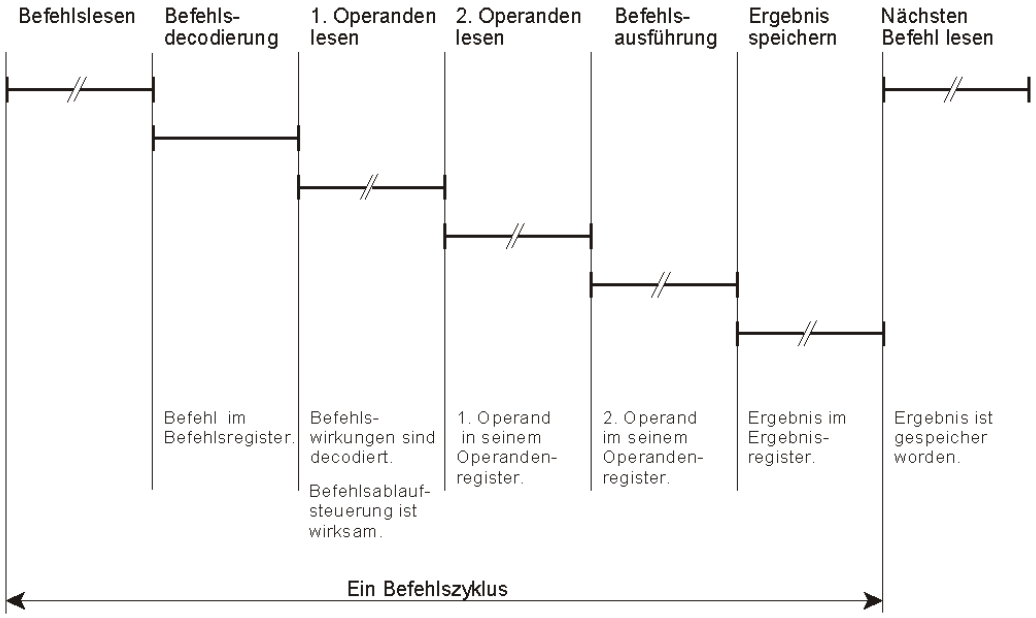


Abb. 3.32 Phasen der Befehlsausführung (1). Der klassische Operationsbefehl. Aus zwei Operanden wird ein Ergebnis gebildet.

Befehlslesen (Instruction Fetch)
Mit der Befehlsadresse (Inhalt des Befehlszählers) werden Lesezugriffe zum Speicher ausgeführt. Das Befehlslesen läuft so lange, bis der ganze Befehl in den Prozessor geholt wurde.

Befehlsdecodierung (Instruction Decode)
Die im Befehl codierten Typ- und Wirkungsangaben müssen decodiert werden. Der gesamte Befehl wird üblicherweise in einem Befehlsregister gehalten. Mit Decodierschaltungen, die an das Befehlsregister angeschlossen sind, werden alle Befehlswirkungen entschlüsselt.

Operandenlesen (Operand Fetch)
Das Operandenlesen – als besondere Phase – kann entfallen, wenn der Befehl mit impliziten oder mit Direktwert-Operanden arbeitet, wenn also vorausgesetzt wird, dass die Operanden bereits in Registern zur Verarbeitung bereitstehen (bei Direktwerten: im Befehlsregister). Ist das nicht der Fall, müssen Zugriffe zum Speicher oder zu einem Registersatz ausgeführt werden. Dem eigentlichen Zugriff gehen oft noch Abläufe der Adressrechnung voraus.

Ausführung (Operandenverknüpfung; Instruction Execute)
In dieser Phase wird die eigentliche Befehlswirkung erbracht. Die Anzahl der hier nötigen Maschinenzyklen hängt von der Länge der Operanden, der Verarbeitungsbreite der

Verknüpfungsschaltungen, der Kompliziertheit der jeweiligen Operation und in manchen Fällen auch von den aktuellen Werten der Operanden ab.

Speichern der Ergebnisse (Result Store)
Ergebnisse werden in Registern festgehalten oder in Universalregister bzw. in den Speicher geschrieben. Ähnlich dem Operandenlesen kann dem Speichern eine Adressrechnung vorausgehen.

Ablaufüberlappung
Einige der in Abbildung 3.32 angegebenen Vorgänge können gleichzeitig ablaufen. Das hängt im einzelnen von den architekturseitig festgelegten Befehlswirkungen, von der Schaltungsauslegung und davon ab, was man als Maschinenzyklus ansieht. So können beide Operanden gleichzeitig gelesen werden, wenn es zwei Speicherzugriffswege gibt (wie bei Universalregistersätzen üblich). Wenn das Ergebnis durch kombinatorische Verknüpfungen gebildet wird (wie beim Addieren von Binärzahlen) und wenn die Speicher drei Zugriffswege haben, ist es durchaus möglich, den gesamten Befehlsablauf vom Holen des Befehls bis zum Speichern des Ergebnisses in einem einzigen Maschinenzyklus zu erledigen (dessen Dauer von den Verzögerungs- und Zugriffszeiten der genutzten Schaltmittel abhängt).

Transportbefehle
Es gibt keine Operation im eigentlichen Sinne. Deshalb entfällt die Phase der Operationsausführung. Die zu transportierenden Daten müssen aber zwischengespeichert werden. Das erfordert wenigstens einen Maschinenzyklus (Abbildung 3.33).

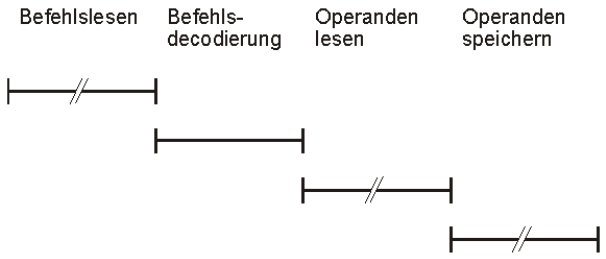


Abb. 3.33 Phasen der Befehlsausführung (2). Der Zweiadress-Transportbefehl.

Verzweigungsbefehle
Der Befehlsdecodierung folgt (bei bedingter Verzweigung) die Entscheidung darüber, ob die Verzweigung stattfindet oder nicht (Abbildung 3.34). Die Verzweigung selbst erfordert das Bereitstellen der Verzweigungsadresse (erforderlichenfalls mittels Adressrechnung) und das Laden des Befehlszählers. Ist keine Verzweigung auszuführen, wird der Befehlszähler nicht überladen. Somit wird der Folgebefehl gelesen.

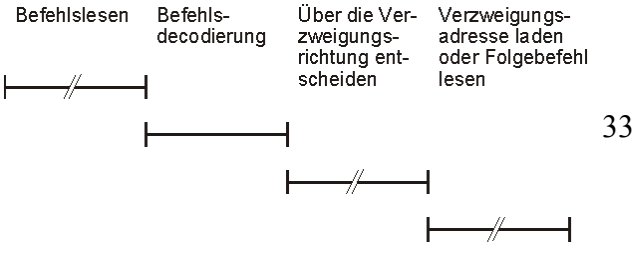


Abb. 3.34 Phasen der Befehlsausführung (3). Bedingte Verzweigung.

Unterprogrammruf
Der Ablauf entspricht dem der Verzweigungsbefehle, wobei die Adresse des Folgebefehls gerettet werden muss. Die Rettungsphase kann Adressrechnungen und Speicherzugriffe enthalten. Nach dem Retten wird die Aufrufadresse in den Befehlszähler geladen (Abbildung 3.35).

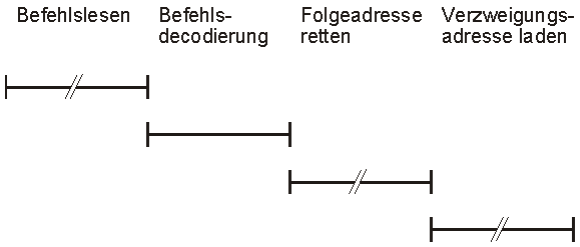


Abb. 3.35 Phasen der Befehlsausführung (4). Unterprogrammruf.

Befehlspipelining
In den Zyklen, die in den Abbildungen 3.32 bis 3.35 dargestellt sind, werden Signalwege, kombinatorische Schaltungen und Register durchlaufen oder angesprochen. In besonders kostengünstigen Maschinen werden für unterschiedliche Zyklen die gleichen Schaltmittel genutzt (beispielsweise dann, wenn es nur ein einziges Rechenwerk gibt, das sowohl für die Ergebnisberechnung als auch für die Adressrechenabläufe verwendet wird). Dann muss alles nacheinander ablaufen. Sieht man hingegen für jeden Zyklus eigene Schaltmittel vor, kann man schon dann, wenn der aktuelle Befehl decodiert wird, mit dem Holen des nächsten Befehls beginnen. Wird der einzelne Befehl in n aufeinander folgenden Phasen ausgeführt und sind für jede Phase eigene Verknüpfungsschaltungen und Register vorgesehen, so können sich insgesamt n Befehle in verschiedenen Bearbeitungsstadien befinden. Dieses Befehlspipelining ist die übliche Arbeitsweise der Hochleistungsprozessoren.

Verzweigungen und Befehlspipelining
Jede Verzweigung unterbricht den fortlaufenden Befehlsfluss. Bei einer unbedingten Verzweigung muss zumindest der Verzweigungsbefehl geholt und decodiert worden sein, um die Adresse des Folgebefehls zu kennen. Bei einer bedingten Verzweigung wird es noch schwieriger: woher soll die Hardware wissen, ob überhaupt zu verzweigen ist, bevor die Bedingung ausgewertet wurde? Verzweigungen haben einen nicht unbeträchtlichen Anteil an den auszuführenden Befehlen (10...20 % und mehr). Deshalb hat man sich verschiedene Lösungen einfallen lassen, um Lücken im Befehlsstrom zu vermeiden:

- Sprungzielvorhersage (Branch Direction Prediction). Sobald die Befehlsvorbereitungsschaltungen eine Verzweigung erkennen, wird das vorbeugende Befehlslesen auf die jeweils wahrscheinlichere Richtung umgesteuert. Die Wahrscheinlichkeit kann nach Erfahrungswerten festgelegt werden[27]. Eine Alternative bestht darin, im Befehl Hinweise zur wahrscheinlichen Verzweigungsrichtung zu codieren (Beispiel: IA-64).

27: Zwei einfache Beispiele: (1) Ein Verzweigen bei ungleich Null dient oft zum Schließen von Schleifen. Schleifen werden zumeist mehrmals durchlaufen, aber nur einmal verlassen. Deshalb ist es wahrscheinlicher, dass das Programm in Verzweigungsrichtung fortgesetzt wird. (2) Ein Verzweigen bei Überlauf dient oft zur Fehlerbehandlung. Es kommt aber öfter vor, dass ein Rechenergebnis im Wertebereich bleibt als dass es außerhalb des Wertebereiches liegt.

- Sprungzielpuffer (Branch Target Cache). Das ist ein Schnellspeicher, der jene Befehle aufnimmt, die zuletzt als Ziele von Verzweigungen erreicht wurden. Beispielsweise sind die letzten 256 Sprungziele in diesem Cache enthalten. Es ist sehr wahrscheinlich, dass immer wieder zu diesen Befehlen verzweigt wird. Die Verzweigung kostet dann keinen zusätzlichen Zyklus, und die Hardware hat genügend Zeit, die jeweils folgenden Befehle heranzuschaffen.
- Auffüllen der Lücke mit anderen Befehlen (Nacheilende Verzweigung; Delayed Branch). Dieses Prinzip ist für RISC-Maschinen charakteristisch. Die Befehlspipeline wird wegen einer Verzweigung nicht angehalten. Demzufolge wird die Verzweigung nicht sofort, sondern beispielsweise einen Zyklus später ausgeführt. Im aktuellen Zyklus kommt deshalb immer der nächste Befehl zur Wirkung. Es ist Angelegenheit des Compilers, einen passenden Befehl entsprechend einzufügen. Das kann beispielsweise ein Transportbefehl sein, der ohnehin erforderlich ist. Gibt es nichts Passendes, ist ein Befehl einzufügen, der nicht schadet (NOP).

Konflikte in der Pipeline
Konflikte entstehen dann, wenn Datenabhängigkeiten auftreten. Beispiel: ein Befehl verändert in der Ausführungsphase einen Speicherinhalt, den der nachfolgende Befehl lesen möchte. Dieser Befehl befindet sich aber bereits in einer Phase des Operandenlesens. Inhaltsänderung und Lesezugriff werden also gleichzeitig ausgelöst. Der zweite Befehl muss aber den vom ersten veränderten Speicherinhalt vorfinden. Solche Konflikte werden automatisch erkannt und vermieden. Hierzu werden unter anderem Wartezustände eingefügt, Pipelinestufen zwecks schneller Datenweitergabe mit besonderen Zugriffswegen überbrückt (Data Forwarding) und Abläufe wiederholt.

3.3.3 Programmablaufsteuerung

Programmstart
Nach dem Einschalten oder Rücksetzen beginnnt der Programmablauf üblicherweise an einer festen Adresse, beispielsweise an Adresse Null. Zu dieser Zeit müssen die anfänglich auszuführenden Befehle im Speicher bereit stehen.

Nichts tun
Eine wichtige Befehlswirkung besteht darin, nur einen Befehlszyklus auszuführen, den Folgebefehl zu adressieren und sonst gar nichts zu tun (No Operation; NOP). Typische Anwendungen:

- Darstellung von Zeit. Beispiel: Zwischen einer Ausgabe und der ersten nachfolgenden Eingabe muss ein Zeitintervall von wenigstens zwei Taktzyklen liegen. Das ist so wenig, dass es sich offensichtlich nicht lohnt, eine Zeitschleife aufzusetzen oder eine Zeitgebereinheit zu starten. Eine Befehlsfolge OUT – NOP – NOP – IN leistet das Gewünschte auf einfachste Weise.
- Programmerprobung (Debugging). Ein verdächtiger Befehl soll probeweise nicht ausgeführt werden. Hierzu wird er mit einem NOP überschrieben.
- Platzhalter, Füllen von Lücken, definierte Belegung ungenutzter Speicherbereiche.
- Bedingte Befehlsausführung. Abhängig von Bedingungssignalen werden bestimmte Befehle entweder ausgeführt oder als NOP interpretiert.

Halt
Kein Maschinenprogramm kann nur auf der Befehlsadresszählung beruhen, denn irgendwann würde einmal die letzte Speicherposition erreicht werden. Das Verhalten der Maschine wäre

dann undefiniert. Jeder Programmablauf muss entweder zu einem definierten Ende kommen oder zu einer Schleife werden. Manche Architekturen haben eigens HALT-Befehle (Beispiel: x86). Ein solcher Befehl hält die Befehlsadresszählung an. Die Ausführung des HALT-Befehls kann nur von außen beendet werden, nämlich durch Ausschalten, Rücksetzen oder Auslösen einer Unterbrechung.

Unbedingte Verzweigung
Um Programmschleifen zu bilden, muss der Befehlszähler mit der Anfangsadresse der Schleife überladen werden (unbedingte Verzweigung, Sprungbefehl). Die einfachsten brauchbaren Programme sind starre Schleifen. Ein Anhalten des Programmablaufs ergibt sich durch Verzweigen auf die eigene Befehlsadresse (Sprung auf sich selbst).

Bedingte Verzweigung
Die bedingte Verzweigung, mit der man den Programmabauf in Abhängigkeit von Verarbeitungsergebnissen und anderen Bedingungen verändern kann, ist das Kennzeichen des Universalrechners schlechthin. Eine Maschine ohne bedingte Verzweigungen wäre nur eine Art Folgesteuerung (Sequencer) mit Rechenfunktionen. Die bedingte Verzweigung beruht auf einer Entscheidung darüber, ob der Folgebefehl ausgeführt oder ob der Befehlszählers mit der Verzweigungsadresse überladen wird (Abbildung 3.36). Die Entscheidung hängt davon ab, ob eine im Befehl angegebene Bedingung erfüllt ist (Wahrheitswert 1) oder nicht (Wahrheitswert 0).

Verzweigungsbedingungen
Die typischen Verzweigungsbedingungen entstehen im Ergebnis von Rechenoperationen oder Booleschen Verknüpfungen hervor (Ergebnis gleich Null, Ausgangsübertrag usw.). Viele Mikrocontroller weisen zudem eine Einzelbitabfrage auf, die sich auf Registerinhalte und E-A-Ports bezieht. Zumeist ist es möglich, wahlweise bei Erfüllung oder Nichterfüllung der jeweiligen Bedingung zu verzweigen.

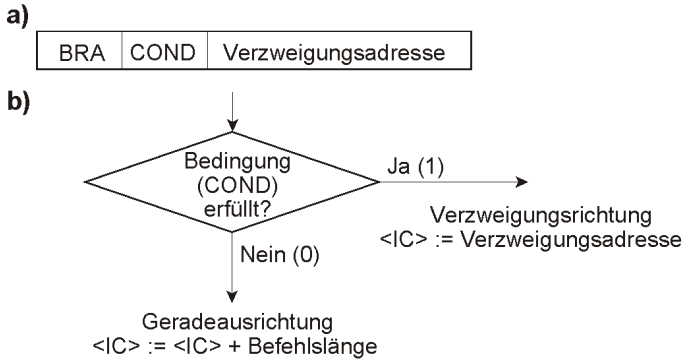


Abb. 3.36 Bedingte Verzweigung. a) Befehlsformat; b) Ablauf.

Überspringen (Skip)
Das Befehlsformat einer bedingten Verzweigung muss den Operationscode, die Bedingungsangabe und die Verzweigungsadresse aufnehmen. Sind die Befehle kurz, so gibt es Schwierigkeiten, alles unterzubringen. Eine Radikallösung besteht darin, die Verzweigungsadresse wegzulassen und die bedingte Verzweigung als Überspringen des Folgebefehls zu implementieren. Ist die Bedingung nicht erfüllt, wird der Folgebefehl ausgeführt. Ist die Bedingung erfüllt, wird der Folgebefehl übergangen und stattdessen der übernächste Befehl ausgeführt. Diese Auslegung ermöglicht es, vergleichsweise lange

Bedingungsangaben[28] in kurzen Befehlen unterzubringen. Sie wird deshalb in vielen kleineren Maschinen bevorzugt. Der Folgebefehl – der entweder übersprungen oder ausgeführt wird – kann ein beliebiger Befehl sein (Abbildung 3.37a). Eine typische Nutzung besteht aber darin, als Folgebefehl eine unbedingte Verzweigung (oder einen Unterprogrammaufruf) vorzusehen und diesen bei nicht erfüllter (invertierter) Bedingung zu überspringen (Abbildung 3.37b). Manchmal gelingt es, den Unterschied im Programmablauf auf das Ausführen oder Nichtausführen eines einzigen Befehls zurückzuführen[29]. Beispiel: ein Einzelbittransport. Die Belegung der Bitposition X (beispielsweise in einem E-A-Register) soll in die Bitposition Y (beispielsweise im Speicher) übertragen werden. Hierzu gibt es Löschbefehle (CLEAR), Setzbefehle (SET) und Übersprungbefehle (SKIP):

```
CLEAR Y
SKIP IF X = 0
SET Y
```

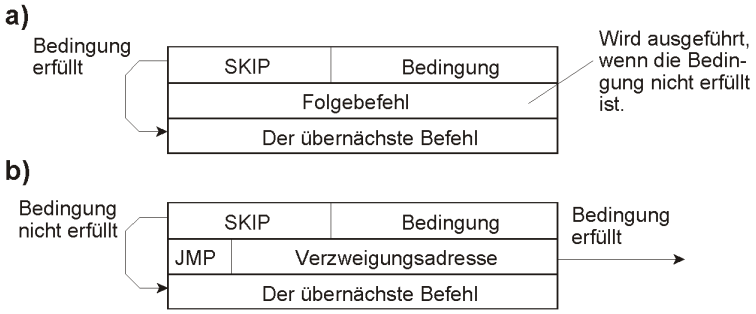


Abb. 3.37 Das Überspringen (SKIP) als Grundoperation der bedingen Verzweigung. a) allgemein: b) bedingte Verzweigung durch Überspringen eines unbedingten Verzweigungsbefehls bei nicht erfüllter Bedingung.

Bedingte Befehlsausführung

Bedingungen entscheiden darüber, ob der aktuelle Befehl ausgeführt wird oder nicht (Conditional Execution, Predication). Im Gegensatz zum Überspringen wird der Befehl immer gelesen. Soll er nicht ausgeführt werden, wird er als NOP interpretiert. Abbildung 3.38 veranschaulicht zwei Beispiele.

Die bedingte Befehlsausführung ist vor allem in Maschinen von Vorteil, in denen sich mehrere Befehle gleichzeitig in verschiedenen Phasen der Ausführung befinden (Pipelining). Ist eine Verzweigung auszuführen, so muss – da der nächste auszuführende Befehl nicht mehr von einer Folgeadresse kommt – die Pipeline gelöscht werden und neu anlaufen. Bei jeder Verzweigung gehen somit mehrere angearbeitete Befehle verloren. Sind nur wenige Befehle zu überspringen, so ist der Zeitverlust geringer, wenn sie in der Pipeline bleiben und als NOP ausgeführt werden. Dieser Fall – dass nur wenige Befehle zu übergehen sind – ergibt sich recht häufig, da typische IF-Zweige oftmals nur wenige Anweisungen (nicht selten nur eine einzige) enthalten. Beispiel (Sprache C): if (A = = 0) x++;

28: Beispielsweise zur Bitabfrage von E-A-Ports. Hierzu braucht man sowohl die Portadresse als auch den Index der jeweiligen Bitposition (Bitadresse).

29: Ein Prinzip, auf dem viele Programmiertricks beruhen.

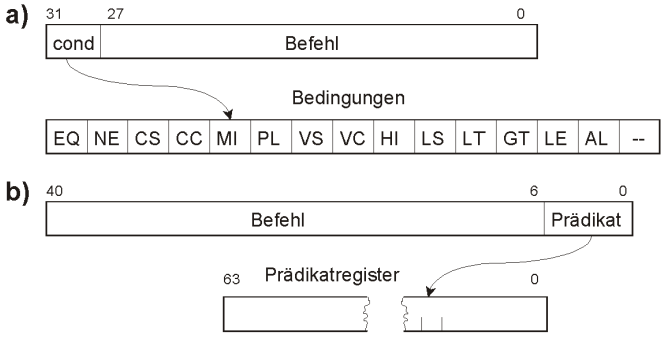


Abb. 3.38 Bedingte Befehlsausführung. a) ARM, b) IA-64.

a) ARM. Die Befehle haben ein Bedingungsfeld (cond) von vier Bits Länge. Hiermit wird eine von maximal 16 typischen Bedingungen ausgewählt (EQ = Gleichheit, NE = Ungleichheit; CS = Übertrag gesetzt; CC = Übertrag gelöscht, AL = immer ausführen usw.; zu Einzelheiten siehe die Architekturbeschreibung).

b) IA-64. Die Bedingungen heißen Prädikate. Sie werden in einem Prädikatregister gehalten. Es gibt 64 Prädikatbits. Die Befehle haben ein Auswahlfeld, das die jeweils auszuwertende Bitposition angibt. Ist das Prädikatbit gesetzt, wird der Befehl ausgeführt. Es gibt Vergleichs- und Testbefehle, mit denen die Prädikatbits gesetzt werden können. Die Prädikatbits werden auch bei bedingten Verzweigungen ausgewertet.

Unterprogramme

Ein Unterprogramm (Subroutine) ist ein Programm, das von einem anderen Programm, dem sogenannten Hauptprogramm, aufgerufen wird und das am Ende seines Ablaufs eine Rückkehr zum Hauptprogramm veranlasst (Abbildung 3.39). Damit das Prinzip anwendungspraktisch nutzbar ist, müssen folgende Funktionen implementiert werden:

- Das Aufrufen des Unterprogramms.
- Die Rückkehr zum Hauptprogramm.
- Die Übergabe der Parameter (der Daten, die das Unterprogramm verarbeiten soll).
- Die Rückgabe der Ergebnisse.
- Falls erforderlich, das Retten und Wiederherstellen von Registerinhalten und anderen Zustandsangaben des Hauptprogramms.

Es gibt Aufruf- und Rückkehrbefehle (CALL, RETURN (RET)). Der Unterprogrammaufruf ist eine Verzweigung, der ein Retten der Folgeadresse vorangeht. Die Rückkehr ist eine indirekte Verzweigung mit der geretteten Folgeadresse, die in den Befehlszähler zurückgebracht wird. Alles andere ist typischerweise programmseitig zu erledigen.

Das Retten der Folgeadresse (Rückkehradresse)

Die aus dem Befehlszähler entnommene Adresse muss irgendwo gespeichert werden. Hierzu gibt es mehrere Möglichkeiten:

- In einem speziellen Register (Linkregister). Beispiele: PowerPC, SPARC.
- In einem Register, dessen Adresse im Befehl angegeben wird. Beispiel: S/360.
- In einem Hardwarestack. Beispiel: PIC16.
- In einem Stackbereich im Arbeitsspeicher. Beispiele: AVR, x86.

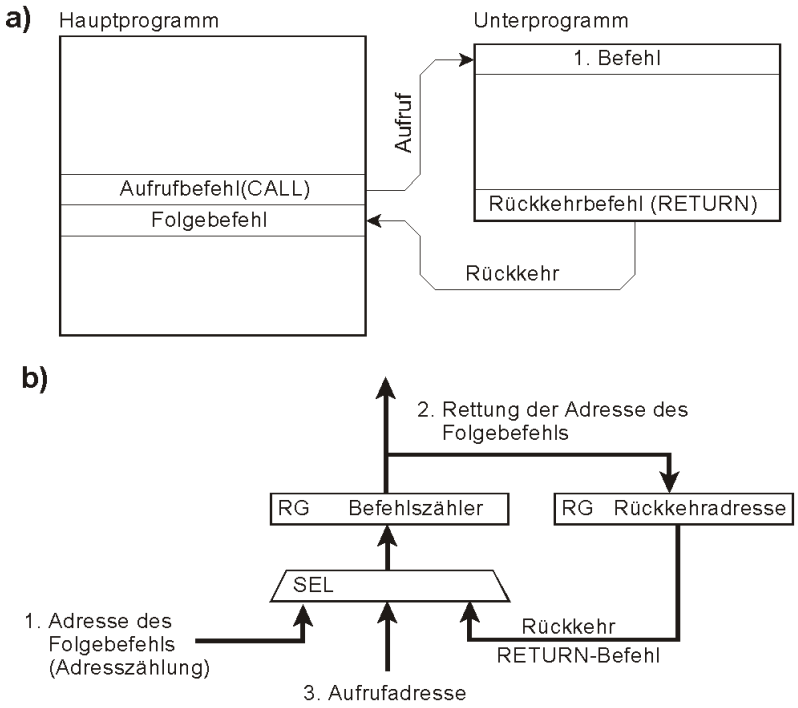


Abb. 3.39 Prinzip des Unterprogrammrufs.a) Programmschema; b) Rettung der Rückkehradresse. Der CALL-Befehl läuft in den Schritten 1 bis 3 ab. 1 - die Adresse des Folgebefehls gewinnen; 2 - die Adresse des Folgebefehls retten; 3 – Überladen des Befehlszählers (Verzweigung zum Unterprogramm).

Was ist noch zu retten?
Alle Registerinhalte und anderen Prozessorzustände, die vom Unterprogramm genutzt oder beeinflusst werden, aber für das Hauptprogramm erhalten bleiben müssen. Das betrifft unter anderem Bedingungssignale (Flagbits) und Register, die vom Unterprogramm genutzt werden. Die Rettungs- und Wiederherstellungsvorgänge sind typischerweise auszuprogrammieren. Es gibt aber auch Prozessoren mit umschaltbaren Registersätzen (Beispiele: SPARC, IA-64).

Unterprogramme schachteln
Um Unterprogramme wirkungsvoll zu nutzen, muss man sie ineinander schachteln können (Nested Subroutine Calls; Programm A ruft Unterprogramm B, dieses ruft Unterprogramm C usw.). Die allgemeine Lösung ist der Kellerspeicher (Stack). Ist kein Stackmechanismus vorhanden, muss er programmseitig nachgebildet werden (Software Stack).

Praxistipp zur Schachtelungstiefe: Eine Schachtelungstiefe von zwei (das Hauptprogramm ruft Unterprogramm A, diese ruft Unterprogramm B) ist sehr knapp (Beispiel: PIC12). Für typische Mikrocontrolleranwendungen genügt oftmals eine Schachtelungstiefe von acht, für große Systeme wird üblicherweise eine Schachtelungstiefe von 32 als ausreichend angesehen

Parameterübergabe
Zur Übergabe der zu verarbeitenden Daten (Parameter Passing) sind folgende Prinzipien üblich:

- In Registern (Beispiel: PC-BIOS).
- In eigens vereinbarten Speicherbereichen (Beispiel: einige Funktionen des PC-BIOS).
- Im Stack (Beispiel: C/Unix, Windows-API usw.).

- Im Programm selbst (an den Unterprogrammaufruf anschließend).
- Kombinationen dieser Verfahren.

Ergebnisrückgabe
Ergebnisse können in Registern, im Stack oder im Speicher zurückgegeben werden.

3.4 Befehlsformate

Abbildung 3.40 zeigt das Blockschaltbild eines einfachen Prozessors. Ganz im Innern sehen alle Prozessoren im Grunde ähnlich aus – es sind Schaltwerke, die aus Registern, und Verknüpfungsschaltungen bestehen. Die Befehle haben die Aufgabe, die Verarbeitungsschaltungen mit Operanden zu versorgen, die Ergebnisse abzutransportieren und die Reihenfolge der Verarbeitungsschritte zu steuern. Der typische elemenare Verarbeitungsablauf besteht darin, aus zwei Operanden ein Ergebnis zu bilden (C := A **op** B). De Befehlswirkungen werden entweder codiert – also mit Bitmustern in den Befehlen angegeben – oder von der Befehlsablaufsteuerung in der Schaltung automatisch ausgelöst (implizite Befehlswirkung).

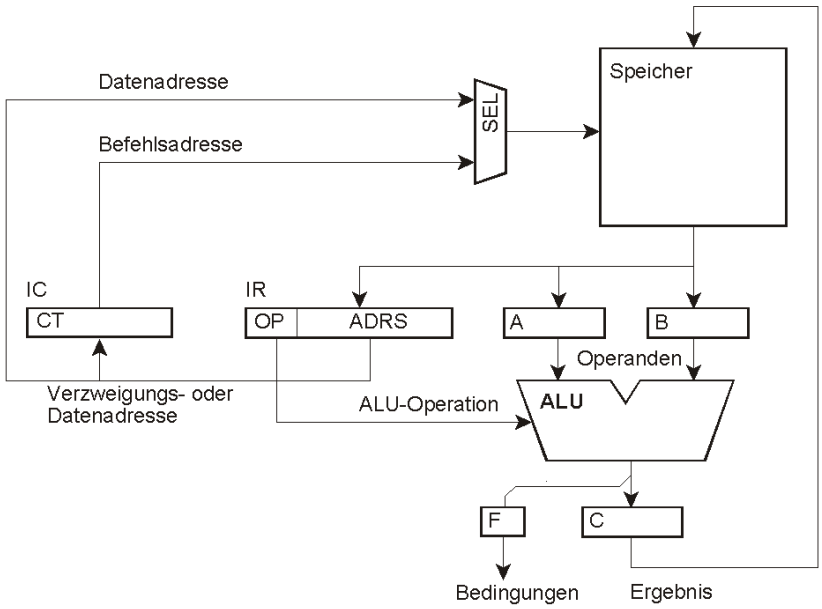


Abb. 3.40 Ein einfacher Prozessor im Blockschaltbild.

Adressen und Direktwerte
Eine Adressangabe im Befehl bewirkt, dass ein weiterer Zugriff ausgeführt wird, um die adressierte Datenstruktur heranzuschaffen oder um ein Ergebnis abzuspeichern. Direktwertangaben (Immediates) hingegen werden unmittelbar aus dem Befehl heraus den Verarbeitungsschaltungen zugeführt. Der Theorie nach braucht man keine Direktwerte, denn man könnte alle festen Angaben als Konstanten im Speicher unterbringen und diese von den Befehlen aus adressieren. Direktwertbefehle haben sich aber bewährt. Sie sind offensichtlich dann von Vorteil, wenn der Festwert kürzer ist als die Adresse, die notwendig wäre, um ihn – als Konstante – im Speicher auszuwählen. In vielen Architekturen gibt es Lade- und Verarbeitungsbefehle in zwei Ausführungen: mit Operandenadresse und mit Direktwert. In Mehradressbefehlen kann einer der Operanden ein Direktwert sein.

Dreiadressbefehle

Ein aus zwei Operanden gebildetes Ergebnis ist unter einer eigenen Adresse abzuspeichern. Die Verarbeitungsbefehle müssen somit drei Adressangaben enthalten (Abbildung 3.41).

a) Verarbeitungsbefehle

Operation	Adresse Operand A	Adresse Operand B	Adresse Ergebnis

b) Transportbefehle

Operation	Quelladresse	Zieladresse

c) Verzweigungsbefehle*

Operation	Verzweigungsadresse

*: Einschließlich Unterprogrammaufruf

Abb. 3.41 Naheliegende Befehlsformate mit Dreiadress-Verarbeitungsbefehlen.

Zweiadressbefehle

Die Ergebnisadresse im Verarbeitungsbefehl wird eingespart. Stattdessen wird einer der Operanden mit dem Ergebniswert überschrieben (Abbildung 3.40). Verknüpfungsschema: <A> := <A> **op** <B>.

Operation	Adresse Operand A	Adresse Operand B
	Ergebnisadresse	

Abb. 3.42 Verarbeitungsbefehl im Zweiadressformat. Die Adresse des ersten Operanden ist zugleich die Ergebnisadresse.

Einadressbefehle

Alle Befehlsformate enthalten nur eine einzige Adresse (Abbildung 3.42). Abläufe, die zwei und mehr Adressen benötigen, sind dann mit mehreren Befehlen zu implementieren. Damit das funktioniert, braucht man Speichermittel, die in den Befehlsabläufen implizit angesprochen werden. Die einfachste Lösung ist ein einziges kombiniertes Operanden- und Ergebnisregister, das als Arbeitsregister oder Akkumulator (Accumulator; AC) bezeichnet wird.

LD	Adresse
OP	Adresse
ST	Adresse

Abb. 3.43 Einadressbefehle. Die Befehle beziehen sich auf ein implizit genutztes Operanden- und Ergebnisregister, den Akkumulator.

Eine typische Operandenverknüpfung <C> := <A> **op** <B> läuft folgendermaßen ab:

1. Laden des Akkumulators mit dem Inhalt von Adresse A (Ladebefehl LD A).
2. Verknüpfung mit dem Inhalt von Adresse B. Das Ergebnis wird in den Akkumulator geschrieben (Verarbeitungsbefehl OP B).

3. Speichern des im Akkumulator stehenden Ergebnisses gemäß Adresse C (Speicherbefehl ST C).

Ein Transportablauf besteht darin, dass der Inhalt der Quelladresse in den Akkumulator gebracht und anschließend mit der Zieladresse gespeichert wird (Befehlsfolge: LD Quelladresse, ST Zieladresse).

Nulladressbefehle
Wenn man in den Verarbeitungsbefehle gar keine Adressen angibt, werden die Befehle noch kürzer (Abbildung 3.44). Dann müssen alle Register in der richtigen Reihenfolge implizit angesprochen werden. Bereits zu Beginn der Entwicklungsgeschichte hatte Konrad Zuse eine elegante Lösung gefunden. Die grundsätzliche Lösung ist das Kellerspeicher- oder Stackprinzip. Eine typische Operandenverknüpfung <C> := <A> **op** <B> läuft folgendermaßen ab:

1. Laden des ersten Operanden (Ladebefehl LD A).
2. Laden des zweiten Operanden (Ladebefehl LD B).
3. Verknüpfung der beiden Operanden (Verarbeitungsbefehl OP). Das Ergebnis wird beispielsweise in das Register A eingetragen.
4. Speichern des Ergebnisses gemäß Adresse C (Speicherbefehl ST C).

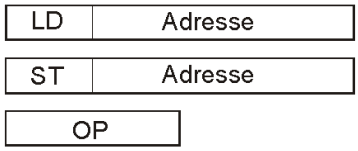


Abb. 3.44 Nulladressbefehle. Die Verarbeitungsbefehle enthalten keine Adressangaben. Sie müssen sich deshalb auf implizit genutzte Register beziehen.

Variable Befehlslänge
Jeder Befehl ist nur so lang wie unbedingt notwendig. Wichtige, häufig gebrauchte Befehle werden mit einem Byte, höchstens mit zwei Bytes codiert. Varianten, die seltener gebraucht werden, Sonderwirkungen und Ähnliches erfordern zusätzliche Bytes.

Feste Befehlslänge
Alle Befehle sind gleich lang. Der Befehl ist in Felder aufgeteilt, in denen die jeweiligen Angaben codiert sind. Üblicherweise entspricht die Befehlslänge der Verarbeitungsbreite (beispielsweise 16 oder 32 Bits). Die Feldaufteilung nimmt keine Rücksicht auf Bytegrenzen. Natürlich wird auch bei einer solchen Formatgestaltung versucht, häufig gebrauchte, leistungsbestimmende Verarbeitungsoperationen, Transportabläufe, Verzweigungen usw. in jeweils einem Befehl unterzubringen. Was auf diese Weise nicht zu codieren ist, erfordert mehrere Befehle. Beide Auslegungen haben Vor- und Nachteile (Tabelle 3.19).

Variable Befehlslänge	Feste Befehlslänge
• Die Befehle sind kürzer. Für einen bestimmten Verarbeitungsblauf braucht man weniger Bits. • Bessere Ausnutzung der Speicherdatenwege und Speicherzyklen. • Höhere Trefferraten der Caches.	• Einfachere Schaltungen zum Adressieren, Holen und Decodieren der Befehle. Somit lässt sich die Taktfrequenz und damit die Arbeitsgeschwindigkeit erhöhen. • Die weitaus meisten Befehle kann man bei beherrschbaren Aufwendungen in einem einzigen Maschinenzyklus ausführen.

Tabelle 3.19 Variable und feste Befehlslängen im Vergleich. Hier sind jeweils typische Vorteile angegeben.

3.5 Registermodelle

Aus der Sicht der Digitaltechnik ist ein Register eine Anordnung gemeinsam zugänglicher Binärspeicher (Latches oder Flipflops). Aus der Sicht der Rechnerarchitektur ist ein Register eine programmseitig zugängliche Speichereinrichtung im Prozessor. Nur diese Register gehören zur API, nur sie bekommt der Programmierer zu sehen. Solche Register wirken als Schnellspeicher. Sie sind (1) im Prozessor ohne besonderen Zeitaufwand erreichbar, und es gibt (2) nur vergleichsweise wenige davon, so dass besondere Auswahlangaben in den Befehlen (Registeradressen) entweder gar nicht benötigt werden (implizite Registernutzung) oder viel kürzer sind als die Speicheradressen. Register werden üblicherweise direkt adressiert; bei Registerzugriffen gibt es keine Adressrechnung. Im Laufe der Entwicklung des Universalrechners sind viele Registeranordnungen vorgeschlagen und ausgeführt worden. Akkumulatormaschinen, Universalregistermaschinen und Stackmaschinen haben die weiteste Verbreitung gefunden.

Gar kein Registermodell
In der Architektur geht es auch ganz ohne Register. Wenn man kein Registermodell definiert hat, gibt es auch keine Probleme mit späteren Erweiterungen. Der Konzeptionsfehler “zu wenige Register” kann gar nicht unterlaufen. Wenn es keine Register gibt, ist es auch nicht notwendig, deren Inhalte bei der Unterbrechungsbearbeitung, Taskumschaltung usw. zu retten und wieder zurückzuschreiben. Die am weitesten verbreitete Architektur ohne Register ist die Java Virtual Machine (JVM). Herkömmliche Maschinen ohne Register wurden als Zwei- oder Dreiadressmaschinen ausgelegt. Alle Adressangaben waren Speicheradressen. Sind die Speicheradressen lang, ergeben sich aber auch lange Befehle.

Wann es ungünstig ist, mehr als einen Speicheroperanden im Befehl zu haben
Das ist dann der Fall, wenn ein virtueller Speicher implementiert werden soll. Bei jedem Speicherzugriff kann es vorkommen, dass sich der adressierte Operand nicht im Arbeitsspeicher befindet. Dann ist das Betriebssystem zu rufen, um den jeweiligen Ausschnitt des Speicherinhalts (Segment oder Seite) vom Massenspeicher zu holen. Der Befehl ist dann erneut auszuführen (Befehlswiederanlauf). Es ist offensichtlich, dass der Verwaltungsaufwand stark ansteigt, wenn so etwas während der Ausführung eines Befehls zwei- oder dreimal vorkommt.

Laden, Verarbeiten und Speichern – das Load-Store-Prinzip
Herkömmlicherweise wirken die Verarbeitungsbefehle so, dass ein Registeroperand mit einem Speicheroperanden verknüpft wird. Der Registeroperand wird mit dem Ergebnis überschrieben:

<R> := <R> **op** <S>

Das ist die typische Auslegung der CISC-Maschinen. In RISC-Maschinen lassen sich hingegen nur Registeroperanden miteinander verknüpfen:

<R3> := <R2> **op** <R1>

Sind Speicheroperanden zu verarbeiten, müssen diese zunächst mit Ladebefehlen in die Register geschafft werden. Die CISC-Befehlswirkung wäre also folgendermaßen nachzubilden:

1. Laden: <R1> := <S>
2. Rechnen: <R2> := <R2> **op** <R1>

Hierfür braucht man zwar mehr Befehle. Der Ansatz hat aber beträchtliche Vorteile:

- Speicheroperanden kommen nur in den vergleichsweise einfachen Lade- und Speicherbefehlen vor. Damit vereinfachen sich auch die Vorkehrungen, die man benötigt, um einen virtuellen Speicher zu implementieren. Die einschlägigen Programmausnahmen können nur beim Laden oder Speichern auftreten. In diesen Fällen kostet der Befehlswiederanlauf (Instruction Restart) keinen übermäßigen Aufwand.
- In vergleichsweise kurzen Befehlen (beispielsweise 16 oder 32 Bits) bekommt man zwei oder drei Registeradressen unter.
- Weil die Registeradressen kurz sind, ist es schaltungstechnisch nicht allzu aufwendig, Gelegenheiten zur Parallelausführung von Befehlen zu erkennen. So sind <R3> := <R2> **op** <R1> und <R6> := <R5> **op** <R4> offensichtlich unabhängig voneinander, können also gleichzeitig ausgeführt werden, falls die Verarbeitungsschaltungen entsprechend ausgelegt sind.
- Die Operationsbefehle können vergleichsweise schnell ablaufen, da keine Adressrechnungen und Speicherzugriffe auszuführen sind.
- Da in den Lade- und Speicherbefehlen keine Operandenverknüpfungen auszuführen sind, können die Verarbeitungsschaltungen zur Adressrechnung genutzt werden, ohne dass dies die Verarbeitungsleistung beeinträchtigt.

Latenzzeiten der Kontextumschaltung
Wenn mehrere Programme gleichzeitig lauffähig sein sollen, ergibt sich das Problem der Kontextumschaltung. Ist bisher das Programm X gelaufen und soll nunmehr das Programm Y Laufzeit erhalten, so müssen die Registerinhalte des Programms X in den Arbeitsspeicher ausgelagert und jene des Programms Y in den Registersatz geladen werden. Ähnliche Auslagerungsvorgänge sind erforderlich, wenn Funktionen ineinander geschachtelt sind (Funktion A ruft Funktion B auf usw.) und wenn für die Variablen der eingeschachtelten Funktion im Registersatz kein Platz mehr ist.

Hat die Maschine vor allem Aufgaben der Ein- und Ausgabe zu erledigen, hat sie nicht viel zu rechnen, aber häufig Kontextumschaltungen auszuführen (Unterbrechungsbehandlung, Multitasking, Schachtelung von Funktionsaufrufen), so ist es besser, die Daten direkt aus dem Arbeitsspeicher heraus zu verarbeiten. Hat die Maschine hingegen vor allem verarbeitungsintensive Aufgaben zu bearbeiten und treten vergleichsweise wenige Kontextumschaltungen auf, ist das Load-Store-Prinzip oftmals überlegen.

Akkumulatormaschinen

Der Akkumulator ist als Operanden- und Ergebnisregister an allen Operationen beteiligt. Der Vorteil der Akkumulatormaschine besteht in der schaltungstechnischen Einfachheit. Es erfordert nur wenige Änderungen, um den Prozessor von Abbildung 3.40 zu einer Akkumulatormaschine umzubauen (Abbildung 3.45). Beim Programmieren ist der Akkumulator dann von Vorteil, wenn Kettenrechnungen auszuführen sind, wenn es also ein Zwischenergebnis gibt, das fortlaufenden Umformungen unterzogen wird. Typische Beispiele sind das Aufsummieren mehrerer Zahlenwerte und das Ausrechnen von Formelausdrücken – all das, was sich mit einem gewöhnlichen Taschenrechner (ohne Speicher) fortlaufend ausrechnen lässt, kann für eine Akkumulatormaschine 1:1 abprogrammiert werden.

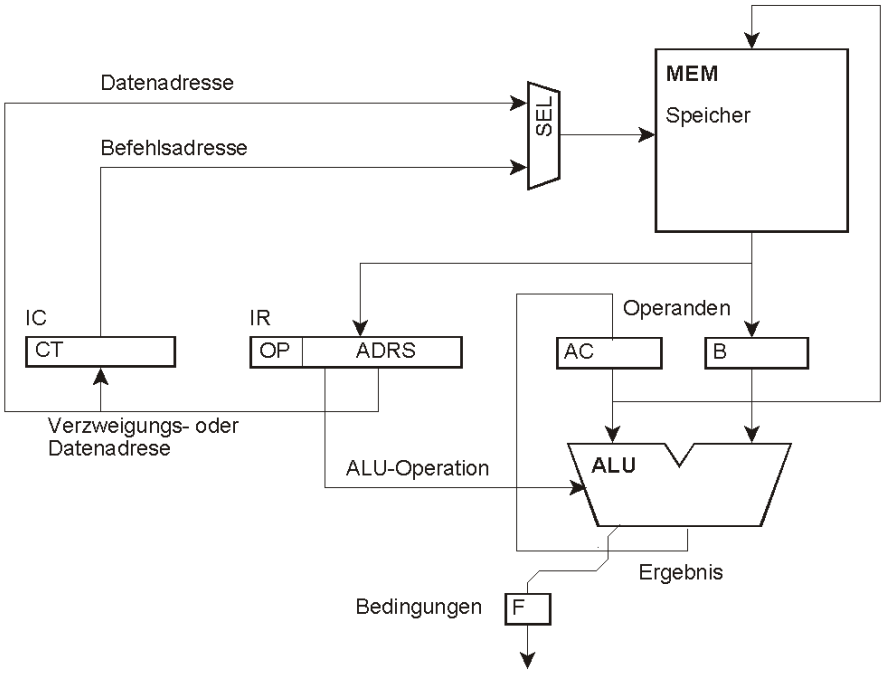


Abb. 3.45 Eine einfache Akkumulatormaschine.

Ganz einfache Einadressmaschinen

Der Akkumulator AC ist das einzige programmseitig zugängliche Register. Alle Transport- und Verarbeitungsbefehle beziehen sich auf den Akkumulator. Das Register B dient nur als Halteregister für die Speicherdaten. Alle Verknüpfungen haben die Form <AC> := <AC> **op** <B>. Das Laden des Akkumulators erfolgt über die Arithmetik-Logik-Einheit (ALU). Die Befehlsformate entsprechen Abbildung 3.43.

Mehrregistermaschinen

Die ursprünglichen Akkumulatormaschinen wurden entwickelt, um mit dem geringsten Schaltungsaufwand auszukommen. Dabei wurde auch die Tatsache ausgenutzt, dass das Programm im selben Speicher steht wie die Daten und dass es somit möglich ist, die Befehle während der Verarbeitung zu verändern. Unter anderem lassen sich die Adressrechnung und die Rückkehr aus Unterprogrammen erledigen, indem man die Adressfelder der jeweiligen Befehle vor der Ausführung überlädt. Selbstmodifizierende Programme haben sich aber als unzweckmäßig erwiesen[30]. Deshalb wurde die Akkumulatormaschine nach und nach durch weitere Register ergänzt: Indexregister (zur Adressrechnung und Datenadressierung),

30: Stehen die Programme in einem ROM, so funktioniert es offensichtlich gar nicht.

Multiplikator-Quotienten-Register, Zählregister usw. Die ursprüngliche x86-Architektur ist ein typisches Beispiel für die Erweiterung mit solchen Einzweckregistern (Abbildung 3.46).

In vielen Architekturen sind mehrere Registersätze verschiedener Zweckbestimmung vorgesehen. So kann man die Anzahl und Länge der Register sowie die zugehörigen Befehlswirkungen jeweils passend festlegen. Auch kann man die Register in der Hardware zweckgerecht anordnen (einen Gleitkomma-Registersatz unmittelbar in der Gleitkomma-Hardware, einen Segmentregistersatz in den Adressierungsschaltungen usw.).

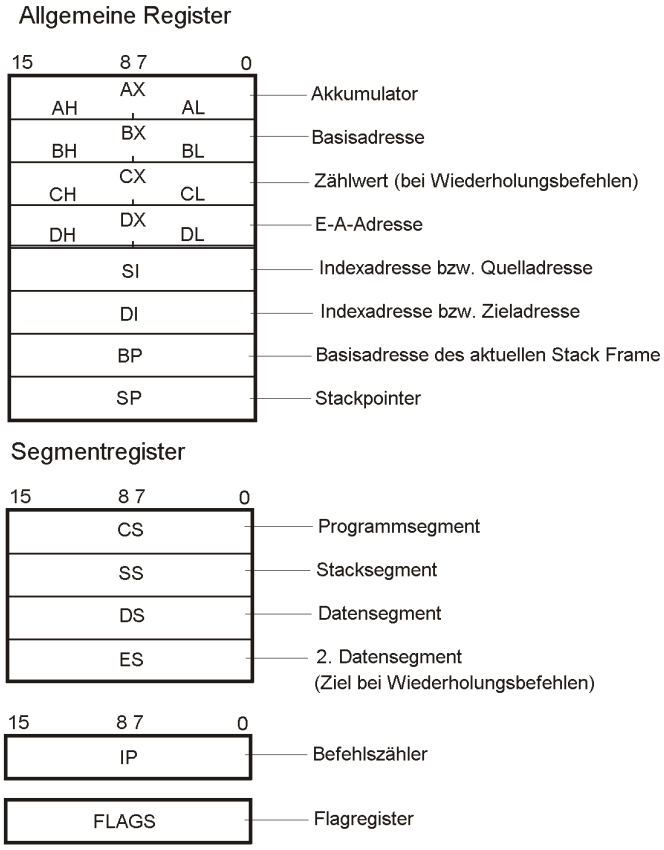


Abb. 3.46 Die Register des 8086 und deren zweckgebundene Nutzung. Die englischsprachigen Bezeichnungen der Register deuten auf ihren ursprünglichen Verwendungszweck hin: AX = Accumulator, BX = Base; CX = Count; DX = Data Address; SI = Source Index; DI = Destination Index; BP = Base Pointer; SP = Stack Pointer.

Universalregistermaschinen

Eine Anordnung gleichartiger Register, die als Adressregister, Indexregister, Operandenregister usw. nutzbar sind (Universalregistersatz, General Purpose Register File), hat auf den ersten Blick den Vorzug der Eleganz. Die grundsätzlichen Auslegungsmerkmale eines Universalregistersatzes betreffen die Anzahl der Register insgesamt, die Anzahl der Register, die gleichzeitig adressiert werden können, sowie die Anzahl und Art der Zugriffswege.

Zweiadressmaschinen

Es gibt nur zwei Zugriffswege. Deshalb muss einer der Operanden mit dem Ergebnis überschrieben werden. RISC-Maschinen können nur Registerinhalte miteinander verknüpfen:

<R1> := <R1> **op** <R2>

CISC-Maschinen können zwei Registeroperanden oder einen Register- und einen Speicheroperanden miteinander verknüpfen. Die Register dienen dann im Grunde als Akkumulatoren. Sie können aber auch als Indexregister, Adressregister, Zählregister usw. genutzt werden.

Dreiadressmaschinen
Es gibt drei Zugriffswege. Zwei Registerinhalte werden miteinander verknüpft; das Ergebnis wird in einem dritten Register gespeichert:

<R3> := <R2> **op** <R1>

Solche Maschinen sind grundsätzlich nach dem Load-Store-Prinzip ausgelegt; in den Dreiadress-Verarbeitungsbefehlen gibt es keine Speicheroperanden.

Zwei oder drei Adressen?
Man muss wissen, was man will:

- Wenn die Universalregister als Akkumulatoren oder Indexregister verwendet werden sollen, ist das Zweiadressprinzip oftmals zweckmäßiger. Die Befehle sind kürzer, die Hardware ist einfacher (nur zwei Zugriffswege).
- Wenn die Universalregister die lokalen Variablen der aktuellen Funktion halten sollen, kommen Verknüpfungen der Art C := A **op** B vergleichsweise häufig vor, so dass das Dreiadressprinzip oftmals besser ist.

Beide Prinzipien sind funktionell ineinander überführbar und somit gleichwertig. Wenn die Universalregister vor allem als Akkumulatoren oder Indexregister genutzt werden, wären Dreiadressbefehle unnötig lang (Platzverschwendung). Verknüpfungen der Art C := A **op** B müssen bei Zweiadressauslegung mit zwei Befehlen implementiert werden, beispielsweise mit C := A gefolgt von C := C **op** B. Das bedeutet sowohl Leistungsverlust (zwei Befehle) als auch Platzverschwendung (zwei Operationscodes; eine der Registeradressen ist zweimal anzugeben). In kleineren Embedded Systems ist es gelegentlich möglich, sämtliche Variable der Anwendung in Registern zu halten. Das gilt sinngemäß für die Variablen typischer Funktionen (man denke beispielsweise an die Programmiersprache C). Viele Rechengänge entsprechen dem Schema C := A **op** B. Aus zwei Operanden ist ein Ergebnis zu bilden, das einer weiteren Variablen als Wert zugewiesen wird. Die Variablen A und B gehen als Operanden in Verknüpfungen ein, werden aber nicht jedesmal im Sinne eines akkumulierenden Rechnens überschrieben. Die Werte der Operanden müssen also vergleichsweise oft erhalten bleiben. In solchen Fällen ist das Dreiadressprinzip zweckmäßiger.

Wieviele Universalregister?
Die Register werden auf zweierlei Weise genutzt:

- Als Behälter für Operanden und Resultate (mit anderen Worten: zur Parameterübergabe an die Verarbeitungs- und Adressierungsschaltungen).
- Als Schnellspeicher für Variable, auf die besonders häufig zugegriffen wird.

Beide Nutzungsweisen geraten nicht selten in Konflikt miteinander.

Acht Universalregister
Diese Anzahl genügt, um die Laufzeitmodelle höherer Programmiersprachen zu unterstützen. Die meisten Register werden als Adress- oder Indexregister verwendet. Für die Variablen sind sie nur als Akkumulatoren oder Operandenregister zu gebrauchen. Eine typische Aufteilung: Stackpointer – Frame Pointer – zwei Datenzeiger (Indexregister) – ein Adressregister für die Ein- und Ausgabe – Multiplikator-Quotienten-Register, zweiter Operand, Akkumulator.

16 Universalregister
Diese Anzahl ist oftmals ausreichend. Etwa acht Register braucht man – wie vorstehend beschrieben – für die Laufzeitumgebung. Somit stehen acht Register zur Verfügung, um Variable aufzunehmen. Manchmal reicht es, manchmal ist es etwas knapp.

32 Universalregister
Etwa 24 Register stehen als Variablenspeicher zur Verfügung. Damit kann man zumeist gut auskommen. Viele RISC-Architekturen sehen deshalb 32 Universalregister vor.

Hilft viel wirklich viel?
Es liegt nahe, die Anzahl der Register weiter zu erhöhen, sobald man es sich leisten kann (Sache der Schaltkreistechnologie). Beispiele: (1) SPARC: 196 Universalregister, (2) IA-64: 128 Universalregister, (3) AltiVec-Erweiterung der PowerPC-Prozessoren: 32 Register zu 128 Bits. Die grundsätzlichen Nachteile extrem großer Registersätze:

- Die vielen Register wollen bei Taskumschaltungen usw. auch gerettet sein.
- Lange Registeradressfelder in den Befehlen.
- Längere Zugriffszeiten. Richtwert: Verdoppelung der Speicherkapazität bedeutet Verlängerung der Zugriffszeit um 30 %. (Mit anderen Worten: ist der Registersatz kleiner, kann der Prozessor mit einer höheren Taktfrequenz betrieben werden.)

Deshalb begnügt man sich manchmal mit nur 16 Registern (ARM, AVR-32). Wenn man größere Registersätze implementiert, sieht man auch höher entwickelte Zugriffsverfahren vor. Diese laufen darauf hinaus, auf die Registernutzung das Prinzip des Stack Frame sinngemäß zu übertragen (Registerfenster). Beispiele: SPARC und IA-64.

Praxistipp: Maschinen mit großen Universalregistersätzen sind im Grunde nichts für Multitasking-Realzeitanwendungen – mag doch die Werbung behaupten, was sie will ...

Load-Store-Maschinen mit großen Registersätzen sind eigentlich entwickelt worden, um traditionelle, verarbeitungsintensive C-Programme auszuführen. Die meisten Funktionen, die in der Programmiersprache C formuliert werden, haben weniger als acht lokale Variable. In solchen Funktionen werden oftmals Schleifen durchlaufen. Damit ist die Maschine vergleichsweise lange beschäftigt. Eine übliche Verfahrensweise besteht darin, dass der Compiler pauschal 8 bis 16 Register für die lokalen Variablen und Parameter der aktuell auszuführenden Funktion reserviert. Beim Funktionsaufruf sind nur die Parameter in die Register zu laden. Nach der Rückkehr findet das rufende Programm das Ergebnis in einem dieser Register vor. Kommt die Funktion mit den Parametern und den lokalen Variablen aus, ist es gar nicht nötig, auf den Arbeitsspeicher zuzugreifen. Somit ergibt sich ein günstiges Verhältnis zwischen Rechenzeit und Transportaufwand. Wenn hingegen die Funktionen vor allem dazu dienen, Steuerungsabläufe und Entscheidungen zu implementieren, die Laufzeiten also vergleichsweise kurz sind, wenn viele Funktionen ineinander geschachtelt werden, wenn immer wieder Speicher- und E-A-Zugriffe erforderlich sind und in kurzen Abständen

Kontextumschaltungen auftreten, dann erweisen sich die CISC-Architekturen oftmals als überlegen. Jedoch bekommen viele Anwender die prinzipbedingten Nachteile der RISC-Architektur gar nicht zu spüren, weil die Schaltkreistechnologie so hohe Taktfrequenzen ermöglicht ...

Stackmaschinen

Ein Stack (Kellerspeicher) ist eine Speicheranordnung, die eine Anzahl gleich langer Informationsstrukturen (beispielsweise Maschinenwörter) aufnehmen kann. Die zuletzt in den Stack gebrachten Daten werden beim Lesen als erste zurückgeliefert (Reihenfolge: Last In, First Out; LIFO). Die Stackzugriffe haben keine Adressen, sie beziehen sich immer auf eine einzige implizit gegebene Speicherposition, die als oberstes Stackelement oder Top of Stack (TOS) bezeichnet wird. Am Anfang ist der Stack leer. Abbildung 3.47 veranschaulicht elementare Schreib- und Lesezugriffe.

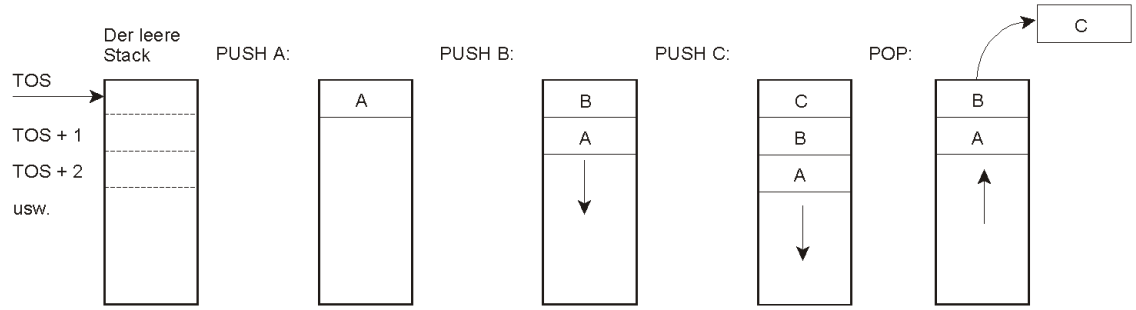


Abb. 3.47 Elementare Stackzugriffe.

Schreiben (PUSH)

Das zu schreibende Speicherwort wird in den Stack eingetragen und somit zum neuen TOS. Der bisherige Inhalt des Stack wird hierzu auf nachfolgende Stackpositionen verschoben: Der bisherige Inhalt des TOS kommt nach TOS + 1 usw.

Lesen (POP)

Der Inhalt des TOS of Stack wird gelesen und aus dem Stack entnommen. Der verbleibende Inhalt wird in Richtung TOS verschoben; der bisherige Inhalt von TOS + 1 kommt nach TOS usw.

Das Verknüpfungsschema:

<TOS> := <TOS> **op** <TOS+1>, wobei <TOS+1> aus dem Stack entfernt wird.

Die Stackmaschine bezieht sich auf den Stack wie die Akkumulatormaschine auf den Akkumulator und das zweite Operandenregister (Abbildung 3.48). Ein typischer Operationsbefehl entfernt die beiden obersten Einträge vom Stack, verknüpft sie miteinander und legt das Ergebnis auf den Stack zurück. Die Operationsbefehle haben keine Adressteile. Die Operanden verschwinden aus dem Stack, das Ergebnis wird zum neuen Inhalt des TOS.

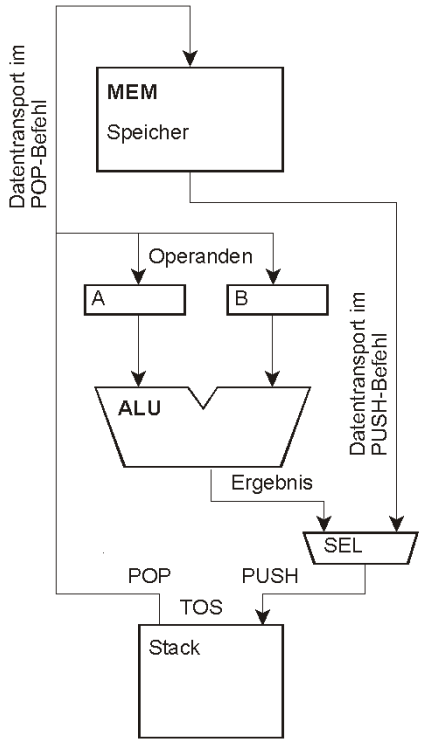


Abb. 3.48 Das Verarbeitungswerk einer einfachen Stackmaschine.

In der Schaltung von Abbildung 3.48 wird ein Verarbeitungsbefehl in folgenden Schritten ausgeführt:

1. <TOS> nach Register A (POP-Ablauf),
2. <TOS> nach nach Register B (POP-Ablauf),
3. Ausführung der Operation,
4. Ergebnis nach TOS (PUSH-Ablauf).

Die umgekehrte polnische Notation (RPN)

RPN (Reverse Polish Notation) ist eine klammerfreie Formelschreibweise, die auf den polnischen Logiker Lukasiewicz zurückgeht. Die Operanden stehen hintereinander. Darauf folgt der Operator. A + B wird so zu A B + , (A + B) • C zu A B + C •. Ein Operator bezieht sich stets auf die beiden links von ihm stehenden Symbole. Sind beide Symbole Variable, so wird das Verknüpfungsergebnis ausgerechnet und als Zwischergebnis in weitere Verknüpfungen einbezogen. A B + heißt: "wende den Additionsoperator auf A und B an". A B + C • heißt: "wende den Additionsoperator auf A und B an und wende den Multiplikationsoperator auf dieses Zwischenergebnis und auf C an".

(A + B) • (C + D) entspricht A B + C D + •; A • (B + C • (D + E)) kann dargestellt werden als A B C D E + • + •. Abbildung 3.49 veranschaulicht, wie diese Ausdrücke ausgewertet werden. Die Zeichenkette wird hierbei von links nach rechts so lange durchmustert, bis man auf einen Operator stößt, dem zwei Variablen vorangehen. Durch Anwenden des Operators wird ein Zwischenergebnis Z1 gebildet, das anstelle der Variaben und des Operators in den Ausdruck eingesetzt wird. Der so gebildete Ausdruck wird erneut durchmustert, es werden weitere Zwischenergebnisse gebildet usw. Der Vorgang ist beendet, wenn der Ausdruck auf zwei Variablen oder Zwischenergebnisse und einen Operator reduziert und daraus das Endergebnis berechnet wurde.

Zu den wichtigsten Erkenntnissen der Informatik gehört, dass sich Ausdrücke in umgekehrter polnischer Notation 1 : 1 in Befehlsfolgen für Stackmaschinen umsetzen lassen. Jede Variablenangabe entspricht einem PUSH-Befehl, jeder Operator dem zugehörigen Operationsbefehl. Die Zwischenergebnisse bleiben so lange im Stack, bis sie von weiteren Operationsbefehlen verrechnet werden. Nach Ausführung der Befehlsfolge liegt das Endergebnis auf dem Stack. Das wird in Tabelle 3.20 und Abbildung 3.50 anhand von Beispielen gezeigt.

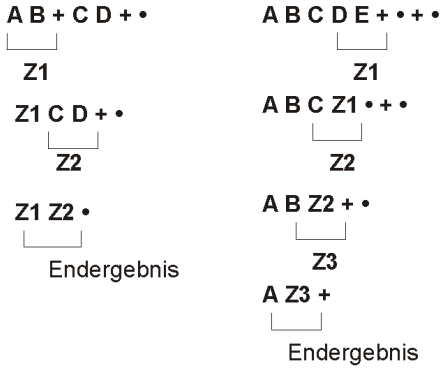
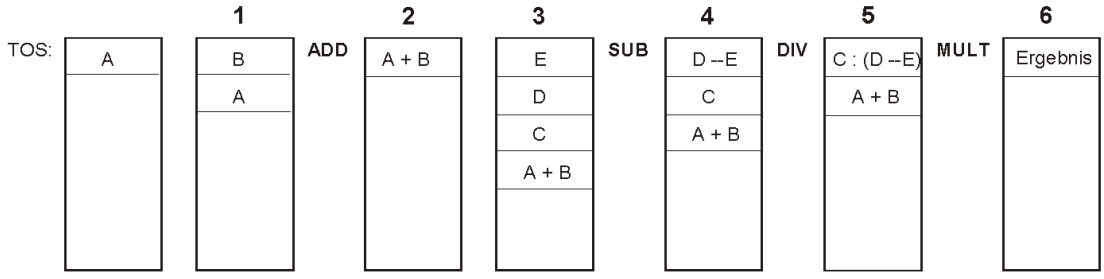


Abb. 3.49 Auswertung von Ausdrücken in umgekehrter polnischer Notation.

(A + B) • (C + D)	A • (B + C • (D + E))	(A + B) • (C : (D – E))
A B + C D + •	A B C D E + • + •	A B + C D E – : •
PUSH A PUSH B ADD PUSH C PUSH D ADD MULT	PUSH A PUSH B PUSH C PUSH D PUSH E ADD MULT ADD MULT	PUSH A PUSH B ADD PUSH C PUSH D PUSH E SUB DIV MULT

Tabelle 3.20 Programmbeispiele.



1 - Operanden A, B auf dem Stack; 2 - der Additionsbefehl bewirkt, dass A und B vom Stack genommen werden und stattdessen die Summe A + B auf den Stack gelegt wird; 3 - die Operanden C, D, E wurden auf den Stack gelegt; 4 - der Subtraktionsbefehl ersetzt die Operanden D und E auf dem Stack durch deren Differenz; 5 - der Divisionsbefehl hinterlässt das Zwischenergebnis C : (D – E); auf dem Stack; 6 - der Multiplikationsbefehl bewirkt schließlich, dass das Endergebnis als einziges auf dem Stack verbleibt.

Abb. 3.50 Operandenverknüpfungen in einer Stackmaschine. Zu berechnender Ausdruck: (A + B) • (C : (D – E)).

Der Stack ist das Mittel der Wahl, um Verschachtelungen, Klammerungen usw. (die durchaus nicht auf mathematische Formeln beschränkt sind) programmseitig aufzulösen. Die meisten Compiler nutzen dieses Prinzip. Der Quelltext wird zunächst in den Code einer fiktiven Stackmaschine umgesetzt. Weithin bekannte fiktive Stackmaschinen sind die P-Code-Maschine (Programmiersprache Pascal) und die virtuelle Maschine der Programmiersprache Java (Java Virtual Machine JVM). Die Programmiersprache Forth stellt den Stackmechanismus sogar als Anwendungsprogrammschnittstelle bereit; sie ist gleichsam um das Stackprinzip herumgebaut.

Stacks in der Rechnerarchitektur
Es gibt Stackmaschinen, Maschinen mit Stackunterstützung und Maschinen ohne Stack. Wenn die Architektur keine Stacks vorsieht oder wenn die vorgesehenen ungeeignet sind, wird die Stackorganisation softwareseitig nachgebildet.

Einige Spitzfindigkeiten:

- Nichtkommutative Operationen (also solche, bei denen es auf die Operandenreihenfolge ankommt, wie die Subtraktion und Division). Die vorstehenen Beispiele beruhen darauf, dass <TOS> – <TOS +1> und <TOS> : <TOS +1> gerechnet wird. Entsprechende Architekturen haben entweder zwei Befehlsvarianten oder einen Austauschbefehl (SWAP), der die Inhalte von TOS und TOS + 1 vertauscht, so dass bedarfsweise die jeweils erforderliche Operandenanordnung hergestellt werden kann.
- Verschiedene lange Variable und Zwischenergebnisse. Es gibt Befehle für verschiedene Operandenlängen. Was nicht passt, wird passend gemacht. So kennt die Java Virtual Machine beim ganzzahligen Multiplizieren und Dividieren keine Produkte und Dividenden doppelter Länge; alle Operanden und Ergebnisse sind jeweils gleich lang.
- Speicheradressierung. Stackmaschinen können auch Speicheradressen im Stack halten und somit den Stack zur Adressrechnung ausnutzen. Die Abbildung 3.51 veranschaulicht elementare Zugriffsbefehle.

Architekturen im Vergleich
Stackmaschinen ergeben oftmals die kürzesten Programme (Tabelle 3.21). Die meisten Stackbefehle haben keinen Adressteil. Viele können mit einzelnen Bytes codiert werden (Bytecode).

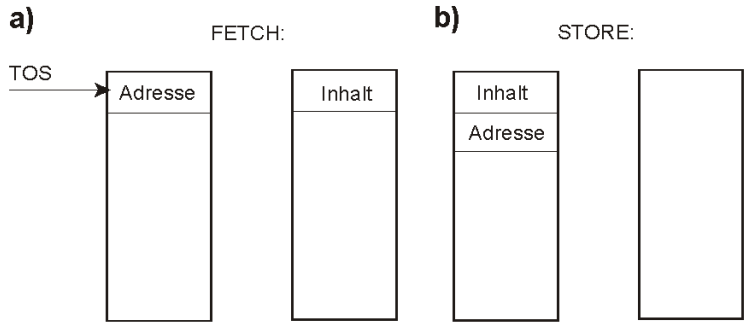


Abb. 3.51 Elementare Speicherzugriffsbefehle. a) Lesen (FETCH; Symbol in Forth: @). Ersetzt die Adresse durch den gelesenen Speicherinhalt; b) Schreiben (STORE; Symbol in Forth: !). Schreibt den Inhalt von TOS auf die Adresse, die in TOS+1 steht und entfernt beide Angaben vom Stack.

Ausdruck: (A + B) · (C : (D - E))			
	Akkumulatormaschine	**Universalregistermaschine**	**Stackmaschine**
Programmablauf	Laden D Subtrahieren E Speichern nach H1 Laden C Dividieren durch H1 Speichern nach H1 Laden A Addieren B Multiplizieren mit H1	Laden A nach R1 Laden B nach R2 Addieren R1 + R2 nach R1 Laden D nach R2 Laden E nach R3 Subtrahieren R2 - R3 nach R2 Laden C nach R3 Dividieren R3 : R2 nach R2 Multiplizieren R1 · R2 nach R1	Push A Push B Addieren Push C Push D Push E Subtrahieren Dividieren Multiplizieren
Benötigt werden	9 Befehle, 9 Speicherzugriffe, eine Hilfszelle (H1) im Speicher	9 Befehle, 3 Register, 5 Speicherzugriffe	9 Befehle, 5 Speicherzugriffe, bis zu 4 Stackpositionen belegt
Programmlänge	9 · 4 = 36 Bytes[1)]	9 · 4 = 36 Bytes[1)]	4 + (5 · 4) = 24 Bytes[2)]

1): 1 Befehl = 4 Bytes (wie 32-Bit-RISC); 2): ein Operationsbefehl = 1 Byte, ein Push oder Pop = 4 Bytes.

Tabelle 3.21 Architekturen im Vergleich. Es wird der Ausdruck von Abbildung 2.70 berechnet.

Weshalb gibt es nicht nur noch Stackmaschinen?
Abgesehen von Fragen der Abwärtskompatibilität haben Stackarchitekturen auch Probleme:

- Stacks sind hervorragend geeignet, um Einträge gleicher Länge aufzunehmen. Mit variabel langen Operanden (Zeichenketten usw.) gibt es hingegen Schwierigkeiten. Die Stackverwaltung wird dann nicht mehr so einfach.
- Zugriffe zu Stackpositionen im Arbeitsspeicher dauern wesentlich länger als zu Registern auf dem Prozessorschaltkreis. Dieser Nachteil lässt sich mit Registerstacks und Stack-Caches weitgehend beheben. Die anderen Nachteile bleiben aber bestehen.
- Die Stackzugriffe sind immer wieder auszuführen; eine in den Stack gebrachte Variable ist nach dem Rechnen verschwunden. Wenn eine Variable in einem Programmablauf fünfmal benötigt wird, ist sie fünfmal in den Stack zu schaffen, wenn dieses Programmstück hundertmal durchlaufen wird, fallen deswegen 500 Speicher- und Stackzugriffe an.
- Aus der Sicht der Theorie ist das Stack-Prinzip äußerst elegant und liefert sehr kompakte Programme. Es ist aber – da immer eines nach dem anderen ablaufen muss – grundsätzlich sequentiell, wodurch der weiteren Leistungssteigerung Grenzen gesetzt sind.

Deshalb bilden folgende Verfahrensweisen den Stand der Technik: Compiler erzeugen zunächst einen Zwischencode für eine (programmseitige implementierte) virtuelle Stackmaschine. Dieser Code wird in Optimierungsläufen weiter bearbeitet, um daraus schnell ablaufende und kompakte Maschinenprogramme für Mehrregistermaschinen zu erzeugen, wobei die zweckmäßige Nutzung der Register ein wichtiges Optimierungskriterium ist. Java-Applets werden als JVM-Bytecodes übers Internet übertragen. Der ankommende Bytestrom wird aber sofort in den Maschinencode des jeweiligen Prozessors übersetzt (Just-in-Time Compilation).

3.6 Adressrechnung

Die meisten Programme werden in höheren Programmiersprachen geschrieben. Die Adressrechnung verbirgt sich unter Formulierungen, wie a[i++, j], personal.name, *index_1 usw. Ohne Adressrechnung ist es nicht einmal möglich, auf lokale Variable zuzugreifen. Die Ausdrücke der Adressrechung fallen im Quelltext kaum auf, haben aber einen beträchtlichen Anteil am Maschinencode. Oftmals haben sie auch entscheidenden Einfluss auf die Programmlaufzeit.

Physische und effektive Adresse
Die physische Adresse ist die Adresse, die über Adresssignalwege zum Speicher geschickt wird. Sie umfasst so viele Bitpositionen, wie nötig sind, um die maximale Speicherausstattung adressieren zu können. Die effektive Adresse ist die Adresse, die von den Adressrechenvorkehrungen des Prozessors gebildet wird. Ihre Länge entspricht der Adresslänge der Prozessorarchitektur. In einfachen Prozessoren entspricht die physische Adresse der effektiven, wobei – abhängig von der vorgesehenen Speicherkapazität – ggf. höherwertige Bitpositionen weggelassen werden (Beispiel: effektive Adresse 16 Bits, installierte Speicherkapazität 16 kBytes, deshalb genügen 14 Bits als physische Adresse). Höher entwickelte Prozessoren enthalten Speicherverwaltungseinheiten (Memory Management Units; MMUs), die die effektive in die physische Adresse umsetzen (Abbildung 3.52).

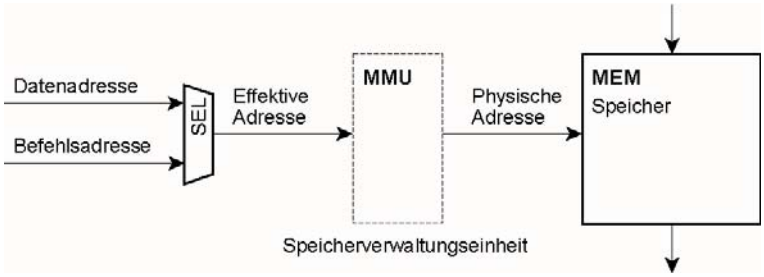


Abb. 3.52 Von der effektiven zur physischen Adresse.

Absolute und direkte Adressierung
Die effektive Adresse steht als Direktwert im Befehl (Abbildung 3.53). Diese Adressierungsweise hat den geringsten Schaltungsaufwand und die kürzeste Verzögerungszeit. Es ist keine Adressrechnung, sondern nur ein Weiterleiten. Damit können aber die folgenden Adressierungsaufgaben nicht gelöst werden:

a) Verschieblichkeit
Die Datenstrukturen und Programme müssen an festen Adressen gespeichert werden; es ist nicht möglich, sie dort unterzubringen, wo gerade Platz ist.

b) Indexrechnung
Es ist nicht möglich, Schleifen zu programmieren, die auf Elemente höher aggregierter Datenstrukturen zugreifen, beispielsweise auf Zeichenketten (Strings), Vektoren oder Matrizen (Arrays). So ist schon die einfache Programmieraufgabe nicht lösbar, eine gespeicherte Zeichenkette Byte für Byte über eine E-A-Schnittstelle auszugeben.

c) Kurze Befehle und lange Adressen
Viele Anwendungen stellen hohe Anforderungen an die Speicherkapazität. Es wäre aber offensichtlich unwirtschaftlich, in jedem Befehl Adressen von beispielsweise 32 oder 48 Bits Länge vorzusehen.

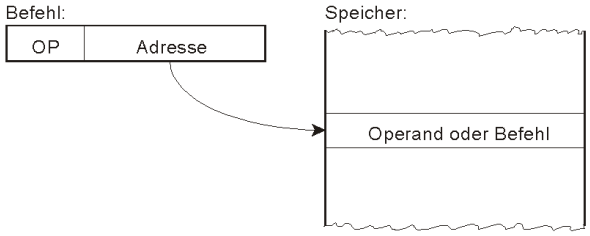


Abb. 3.53 Absolute und direkte Adressierung.

Relative Adressierung

Die Adressangabe im Befehl bezieht sich auf eine Basisadresse, die in einem Register gehalten wird. Die übliche Form der Adressrechnung besteht darin, den Direktwert im Befehl zur Basisadresse zu addieren (Abbildung 3.54). Der Registerinhalt wird dabei nicht verändert. Der Direktwert (Displacement, Offset) D ist die Differenz zwischen der effektiven Adresse EA der zu adressierenden Datenstruktur und der Basisadresse BA:

$$D = EA - BA$$

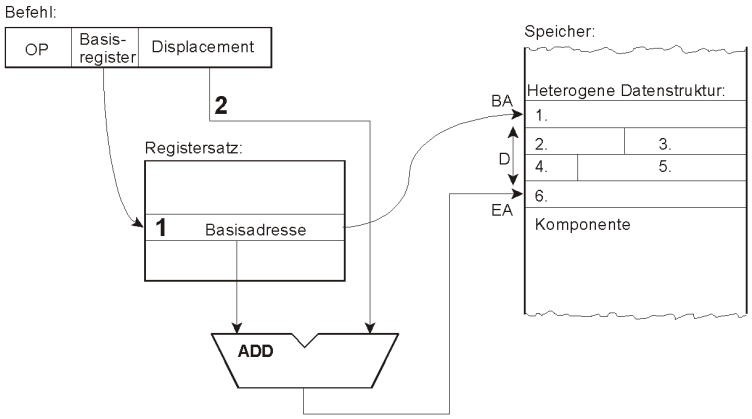


Abb. 3.54 Adressierungsprinzip Basis + Displacement.Im Beispiel wird auf eine heterogene Datenstruktur zugegriffen. 1 - die Basisadrese zeigt auf den Anfang der Datenstruktur. 2 - die einzelnen Komponenten (Zahlen, Zeichenketten usw.) können durch Addieren der jeweiligen Adressendifferenz (Displacement) adressiert werden.

Adressregister

Diese Register werden als Basiadressregister oder Indexregister bezeichnet. Es gibt folgende Auslegungen:

- Nur wenige (manchmals nur ein einziges) Indexregister.
- Ein besonderer Adressregistersatz.
- Einige Universalregister werden als Adressregister verwendet (AVR).
- Alle Universalregister dürfen als Basisadressregister verwendet werden.

Displacementangaben

Die Direktwerte werden zumeist als vorzeichenbehaftete Binärzahlen verrechnet, in manchem Maschinen aber als vorzeichenlose Binärzahlen (S/360, AVR). Die Länge des Displacements liegt zwischen sechs und 16 Bits (Richtwerte).

Diese Adressierungsweise erlaubt es, Variable, Datenstrukturen und Befehle zu adressieren, die an beliebigen Adressen gespeichert sind. Hierzu muss man nur die jeweilige Anfangs- oder Bezugsadresse ins Basiadressregister laden.

Der Befehlszähler als Basisadressregister
In vielen Architekturen sind die Adressangaben in Verzweigungsbefehlen relative Adressen in Bezug auf den Befehlszähler. Es sind vorzeichenbehaftete Binärzahlen in Zweierkomplementdarstellung. Bezieht man sich auf die Adresse des Folgebefehls, ist die Sprungweite in Richtung niederer Adressen um Eins größer als in Richtung höherer Adressen (Abbildung 3.55).

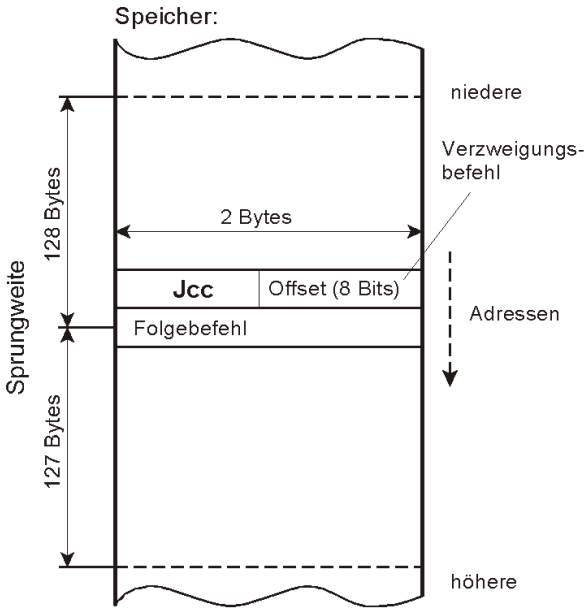


Abb. 3.55 Die Verzweigungsadresse als Relativadresse. Beispiel: die kurzen bedingten Verzweigungsbefehle der x86-Architektur (Befehlslänge 16 Bits).

Indirekte Adressierung

Der Befehl enthält eine Adressangabe. Der adressierte Speicher- oder Registerinhalt wird gelesen und seinerseits als Adresse verwendet ("Adresse von Adresse"). Mit dieser Adresse wird dann der eigentliche Zugriff ausgeführt (Abbildungen 3.56, 3.57).

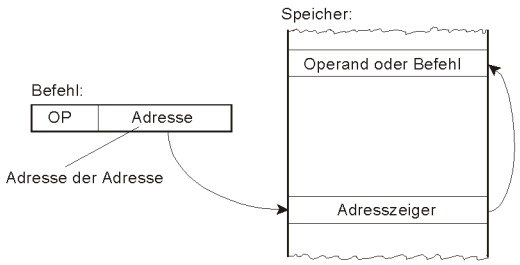


Abb. 3.56 Indirekte Adressierung (1). Adresszeiger im Speicher.

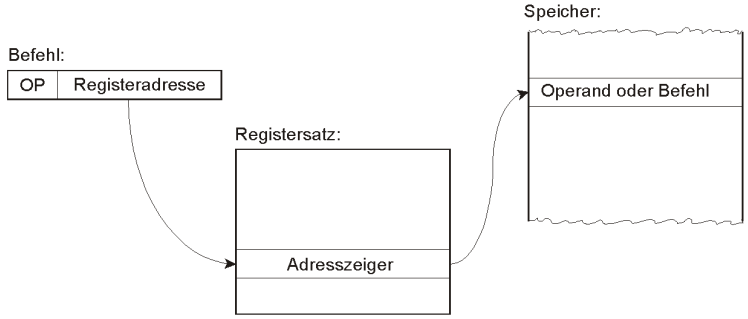


Abb. 3.57 Indirekte Adressierung (2). Adresszeiger in einem Register.

Erweiterte Registeradressierung

Nach dem Grundsatz, Funktionen, die besonders häufig genutzt werden, auch besonders zu unterstützen, hat man die Registeradressierung mit weiteren Adressierungsarten ergänzt. Eine der ersten Maschinen mit solchen universellen Registeradressierungsvorkehrungen war die PDP-11. Die seinerzeit eingeführten Lösungen wurden in verschiedene Mikroprozessorarchitekturen übernommen. Abbildung 3.58 veranschaulicht Adressierungsarten, die auch in RISC-Maschinen vorgesehen sind.

a) Register
Das vom Befehl adressierte Register enthält die Variable.

b) Register indirekt (deferred)
Das vom Befehl adressierte Register enthält die Adresse der Variablen.

c) Basis + Displacement (Index)
Das vom Befehl adressierte Register enthält eine Basisadresse. Die Speicheradresse der Variablen ergibt sich durch Addition eines Festwertes, der im Befehl enthalten ist (Displacement).

d) Automatische Adresserhöhung nach dem Zugriff (Autoincrement)
Das vom Befehl adressierte Register enthält die Adresse der Variablen. Nach dem Speicherzugriff wird der Registerinhalt um die Operandenlänge OL erhöht (<REG> := <REG> + OL).

e) Automatische Adressverminderung vor dem Zugriff (Autodecrement)
Das vom Befehl adressierte Register enthält die Adresse der Variablen. Der Registerinhalt wird zunächst um die Operandenlänge OL vermindert (<REG> := <REG> – OL). Mit dieser Adresse wird der Speicherzugriff ausgeführt.

Ein wichtiger Anwendungsfalll der automatischen Adresserhöhung und -verminderung ist die Implementierung von Stackmechanismen. Es ist – s. weiter unten – eine Art Industriestandard, dass ein Stack in Richtung der niederen Adressen wächst und dass der Stackpointer auf den obersten Eintrag (TOS) zeigt. Ein PUSH-Ablauf erfordert dann eine Adressverminderung vor dem Schreibzugriff (Predecrement), um die nächst-niedrigere freie Speicherposition zu adressieren. Ein POP-Ablauf besteht hingegen aus einem Lesezugriff zum Abholen des aktuell adressierten Speicherinhalts und einer nachfolgenden Adresserhöhung (Postincrement).

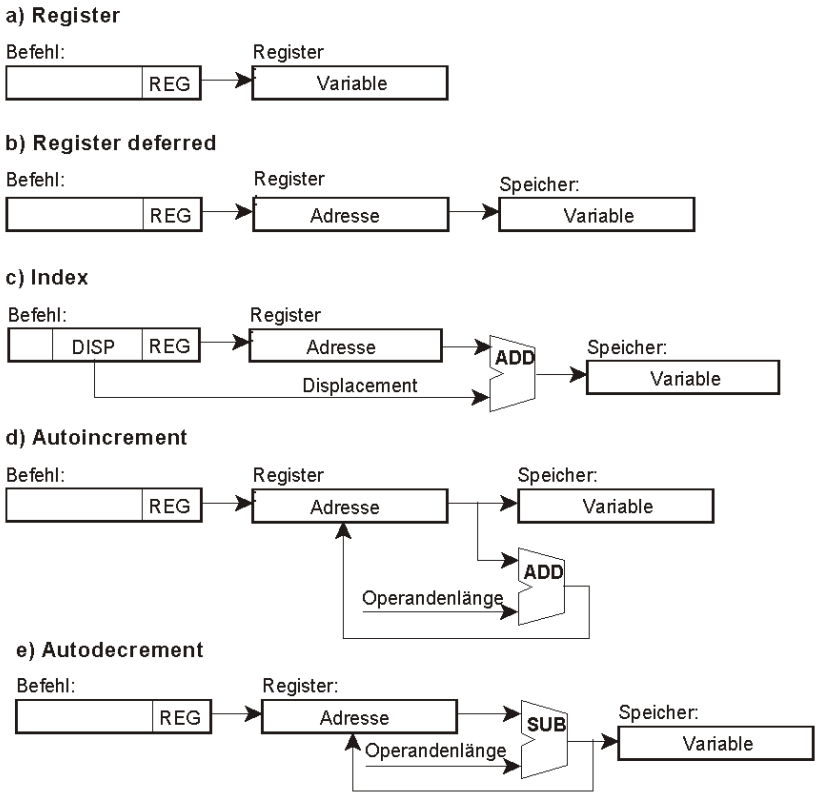


Abb. 3.58 Registeradressierung am Beispiel der PDP-11 (1).

Die PDP-11 unterstützt darüber hinaus – wie andere CISC-Maschinen auch – die indirekte Adressierung (Abbildung 3.59). Die Adresse, die sich aus der Registeradressierung ergibt, wählt hierbei nicht die Variable aus, sondern ein Speicherwort, das seinerseits als Adresse verwendet wird. Die Adressierungsarten mit automatischer Erhöhung oder Verminderung sind vorgesehen, um auf Adresslisten (Adress-Arrays) zuzugreifen. Deshalb wird der Registerinhalt um den Wert der Adresslänge erhöht oder vermindert.

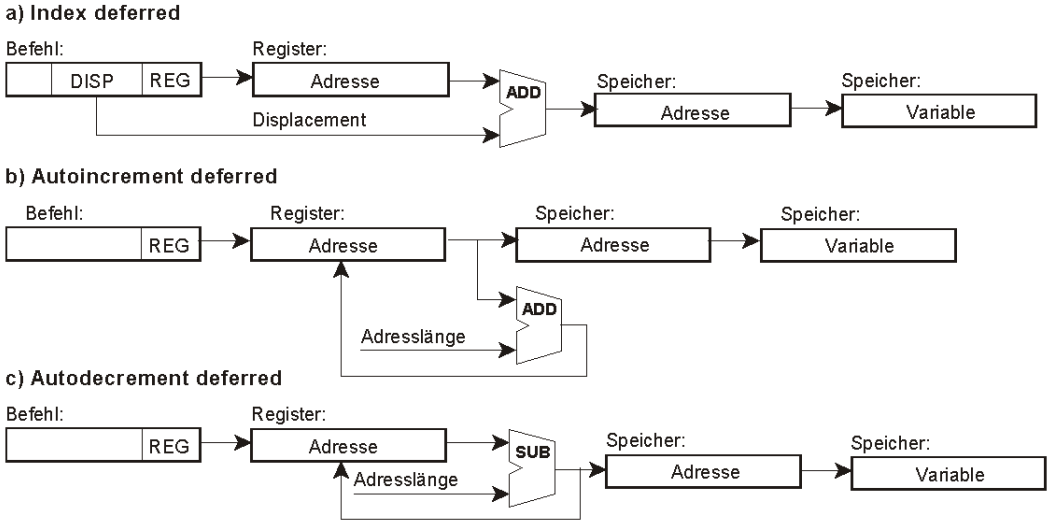


Abb. 3.59 Registeradressierung am Beispiel der PDP-11 (2). Indirekte Adressierung mit Adresszeigern im Speicher. Die Adressen sind zwei Bytes lang (Adresslänge = 2).

Wie viele Adressierungsweisen braucht man wirklich?
Es ist zweckmäßig, sich nach den Anforderungen zu richten, die sich seitens der höheren Programmiersprachen ergeben (man denke beispielsweise an die Sprache C). Ausdrücke in Programmiersprachen adressieren Variable und Komponenten von zusammengesetzten Datenstrukturen. Greift das Programm direkt auf Variable zu, so beziehen sich die Variablenadressen zumeist auf eine Anfangsadresse (Basisadresse). Zeigervariable und Indexadressen sind indirekte Adressen. So kann ein Array-Index (a[x]) ein beliebiger Ausdruck sein (Beispiel: a[i+j*k]). Der Index ist aber nur die laufende Nummer des Feldelements im Array. Wurde der Wert des Indexausdrucks bestimmt, muss die Speicheradresse berechnet werden.

Offensichtlich ist die indirekte Registeradressierung (Register Deferred gemäß Abbildung 3.58b) die einzig wirklich notwendige Adressierungsweise. Die Adresse wird in einem Register aufbereitet, sei es durch Laden eines Direktwerts, sei es durch mehr oder weniger komplizierte Rechenabläufe. Dann wird mit dieser Adresse auf den Speicher zugegriffen.

Wenn man eine Load-Store-Architektur hat und genügend viele Register und die Maschine zudem schnell genug ist, genügt das auch tatsächlich. Beispiel: IA-64. Die Verarbeitungsbefehle arbeiten ausschließlich mit Registern (Registeradressierung gemäß Abbildung 3.58a). Speicheradressen werden nur noch zum Laden und Speichern benötigt. Sie werden ebenfalls in Registern bereitgestellt.

Ist der Registersatz nicht so groß (32 Register oder weniger), sind Kompromisse zu schließen. Die Erfahrung hat gezeigt, dass die Form Basis + Displacement (Indexadressierung gemäß Abbildung 3.58c) vollauf ausreichend ist. Die Displacementangabe wird als ganze Binärzahl interpretiert und mit Vorzeichenerweiterung verrechnet. Ein Displacement von Null bewirkt eine indirekte Registeradressierung (Register deferred). Alle komplexeren Adressrechenvorgänge werden ausprogrammiert.

Wie viele Basisadressen?
Das hängt von der Laufzeitumgebung, von den Anforderungen an die Verarbeitungsleistung und von den Adressierungsvorkehrungen der Architektur ab. Eine einfache Laufzeitumgebung, die auf die Programmiersprache C abgestellt ist, erfordert wenigstens zwei Basisadressregister, eines für die lokalen Variablen (Frame Pointer) und eines für die dynamischen Variablen (Stackpointer). Für Zugriffe mit berechneten Adressen (Zeigervariable, Komponenten von zusammengesetzten Datenstrukturen) braucht man mindestens ein weiteres Adressregister (indirekte Adressierung)[31]. Globale Variable werden absolut adressiert. Ist das nicht möglich, muss ein weiteres Basisadressregister bereit gestellt werden. Eine Maschine, die im Multitasking betrieben wird, braucht je ein Basisadressregister für

- Systemvariable, gemeinsame Speicherbereiche usw. (sofern diese nicht absolut adressiert werden können),
- die globalen Variablen der aktuellen Task,
- die lokalen Variablen der aktuellen Task (Frame Pointer),
- die dynamischen Variablen der aktuellen Task (Stack Pointer).

31: Besser: so viele Register, dass man in den innersten Schleifen Operationen der Art c[k] := a[i] **op** b[j] erledigen kann, ohne vor jedem Zugriff die Adresse laden zu müssen.

Wie lang soll das Displacement im Befehl sein?
In der Anwendungspraxis ist das Displacement die Adresse einer Komponente in einer Datenstruktur oder einer Variablen in einem Stack Frame (s. unten). Wie viele Komponenten oder Variable können vorkommen? Das wurde bereits vor längerer Zeit messtechnisch untersucht. In keinem der untersuchten Programme wurden mehr als 4096 Variable adressiert; am häufigsten waren Werte im Bereich um 17 und um 64. Wenn die Displacements Byteadressen sind und wenn man im Durchschnitt vier Bytes je Variable annimmt, kommt man auf Adresslängen zwischen sechs und 14 Bits. Die Längen der Displacements in den Befehlen der typischen RISC-Maschinen liegen auch tatsächlich in dieser Größenordnung (AVR: 6 Bits, ARM: 12 Bits, SPARC: 13 Bits, MIPS und NIOS: 16 Bits).

Stacks
Das Stackprinzip ist vielseitig nutzbar. Stacks dienen zum Auswerten geschachtelter Funktionsausdrücke, zum Rettung von Rückkehradressen und Registerinhalten, zur Parameterübergabe und als Speicherplatz für lokale Variable. Es sind zumeist Speicherbereiche, die von einem Adressregister, dem Stackpointer, adressiert werden. Der Stackpointer ist zumeist programmseitig zugänglich. In vielen Architekturen wird eines der Universalregister als Stackpointer verwendet. Manche Architekturen sehen mehrere Stacks vor. Es ist aber auch möglich, alle Aufgaben mit einem einzigen Stack zu erledigen.

Wachstumsrichtung
Es ist eine reine Konventionsfrage, ob bei PUSH-Abläufen der Inhalt des Stackpointers erhöht und bei POP-Abläufen vermindert wird oder umgekehrt. In vielen Architekturen wachsen Stacks immer in Richtung der niederen Adressen. Der Inhalt des Stackpointers wird bei PUSH-Abläufen vermindert und bei POP-Abläufen erhöht.

Zähl- und Zugriffsreihenfolge
Es ist auch eine reine Konventionsfrage, ob zunächst der Inhalt des Stackpointers verändert und dann der neue Eintrag gespeichert wird oder umgekehrt. In den meisten Fällen wird der Stack folgendermaßen adressiert:

- Der Stackpointer zeigt auf das oberste Element im Stack (TOS).
- PUSH: Der Inhalt des Stackpointers wird zunächst vermindert (Predecrement), dann wird der neue Eintrag gespeichert.
- POP: Der Inhalt des TOS wird gelesen, dann wird der Inhalt des Stackpointers erhöht (Postincrement).

Stack-relative Adressierung
Ein Nachteil der reinen Stackadressierung besteht darin, dass man an Einträge, die sich weiter unten im Stack befinden , nicht ohne weiteres herankommt. Deshalb ist es zweckmäßig, auch Zugriffe nach dem Prinzip Basis + Displacement zu unterstützen (Indexadressierung), wobei der Stackpointer als Basisregister dient.

Stack Frames
Ein Stack Frame ist ein fester Bereich im Stack. Er dient vor allem dazu, die lokalen Variablen des laufenden Programms aufzunehmen.

Statische und dynamische Variable
Statische Variable werden im Programmtext deklariert (jeder Variablenname wird angegeben, und es wird ihm ein Datentyp zugewiesen). Dynamische Variable entstehen hingegen im Laufe der Verarbeitung (also ohne dass sie der Programmierer ausdrücklich deklarieren muss).

Der Programmierer kann beispielsweise hinschreiben:

Gewicht = Länge * Breite * Höhe * Spezifisches_Gewicht;

Der Compiler muss diesen Formelausdruck in eine Folge von Maschinenbefehlen umsetzen.. Da die einzelnen Befehle nur ganz elementare Operationen ausführen können, fallen im Verlauf der Rechnung Zwischenergebnisse an. Das sind die dynamischen Variablen, die üblicherweise auf dem Stack abgelegt werden.

Sowohl statische als auch dynamische Variable werden im Stack untergebracht
Das muss nicht unbedingt so sein, hat sich aber bewährt. Und zwar vor allem deshalb, weil man gern Programme in Programme schachtelt (Unterprogrammtechnik, Funktionsaufrufe). Dann liegt es nahe, die verfügbare Speicherkapazität im Sinne eines Stack zu verwalten. Zuerst kommt der Stack Frame des ersten Programms. Darüber (in Richtung zu den niederen Adressen hin) werden die gerade aktuellen dynamischen Variablen auf den Stack gelegt. Wenn nun das Programm ein Unterprogramm aufruft, kommt dessen Stack Frame auf den Stack, darüber werden dessen dynamische Variable abgelegt usw.

Das Zugriffsproblem
Gemäß dem Rechenablauf wächst oder schrumpft der Stack. Andererseits sind aber immer die gleichen statischen Variablen zu adressieren. Würde man sich aber stets auf den Stackpointer (als Basisadresse) beziehen, so würden sich bei jedem Zugriff andere Displacements zu den statischen Variablen ergeben. Deshalb sieht man typischerweise ein weiteres Adressregister vor, den sogenannten Frame Pointer oder Base Pointer (Abbildung 3.60).

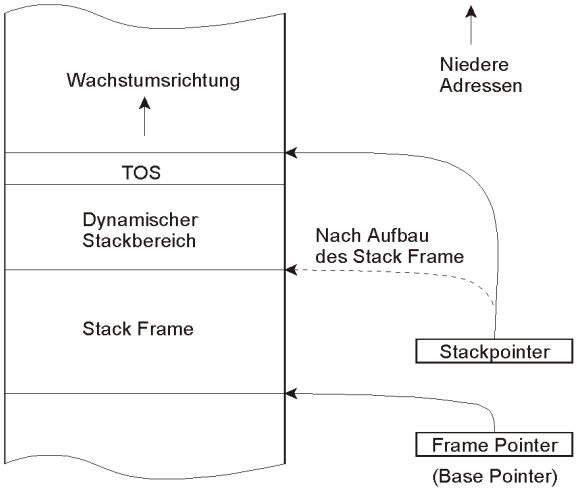


Abb. 3.60 Stack-Organisation mit Stack Frame

Der Frame Pointer zeigt stets auf den Anfang des aktuellen Stack Frame. Alle Inhalte des aktuellen Stack Frame sind somit über negative Displacements erreichbar. Ruft das aktive Programm seinerseits ein Unterprogramm, so wird der aktuelle Inhalt des Stackpointers in den Frame Pointer übernommen. Dann wird oberhalb des dynamischen Bereichs des rufenden Programms der Stack Frame des gerufenen aufgebaut. Nähere Einzelheiten sollen am Beispiel der UNIX-Stackorganisation erläutert werden, die – über die Programmiersprache C – zum Industriestandard geworden ist.

Für jeden Prozess werden zwei Stacks verwaltet (Abbildung 3.61): der User Stack zum Aufrufen von Anwendungsprogrammen und der Kernel Stack zum Aufrufen der Systemfunktionen. Beide Stacks werden auf gleiche Weise genutzt. Diese Aufteilung soll sicherstellen, dass, wenn die Anwendung abstürzt, der Systemstack überlebt und somit das System in der Lage ist, den Fehler zu behandeln.

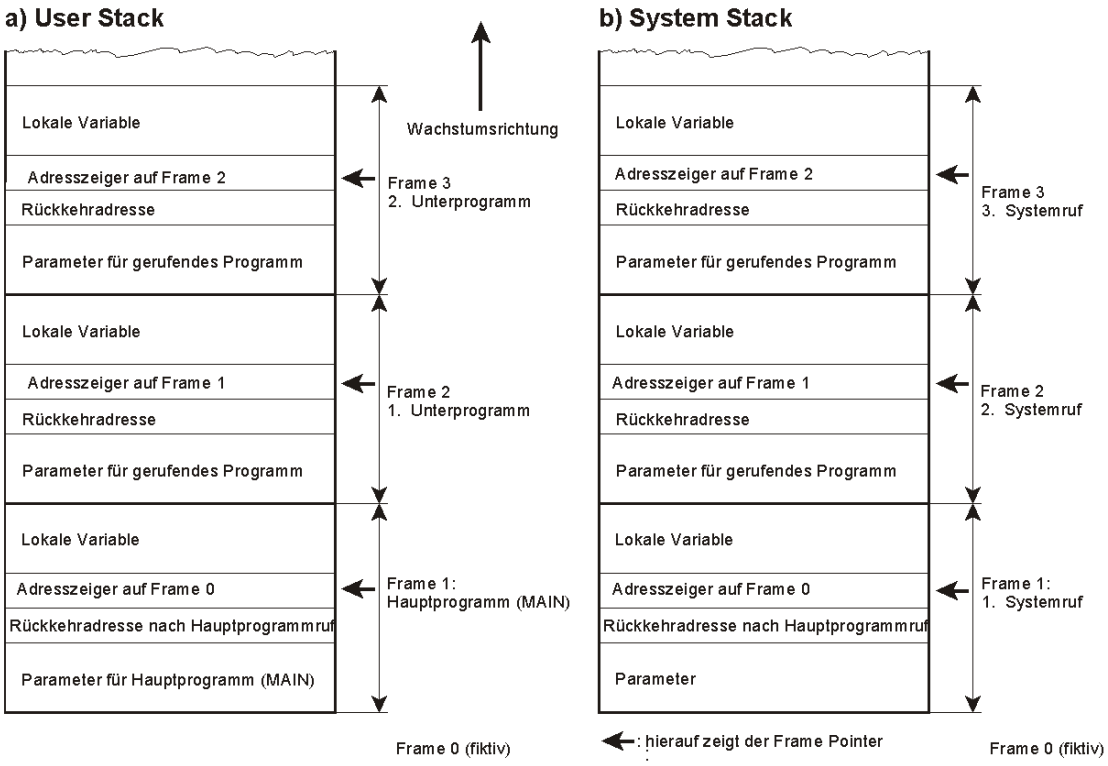


Abb. 3.61 Die UNIX-Stackorganisation.

Ein UNIX-Unterprogrammaufruf (= C-Funktionsaufruf) läuft folgendermaßen ab (Abbildung 3.62):

1. Das rufenden Programm legt die zu übergebenden Parameter auf den Stack.
2. Die Funktion wird gerufen (CALL-Befehl). Dabei gelangt die Rückkehradresse auf den Stack.
3. Die gerufenene Funktion legt den bisherigen Frame Pointer auf den Stack (Rückverweis). Danach wird der aktuelle Inhalt des Stackpointers zum neuen Frame Pointer.
4. Die gerufene Funktion schafft auf dem Stack so viel Platz, dass die lokalen Variablen hineinpassen.

Bei der Rückkehr wird:

1. der Stackpointer mit dem Inhalt des Frame Pointer überladen (hierdurch gehen die lokalen Variablen verloren),
2. der alte Frame Poiner aus dem Stack zurückgeholt,
3. die Rückkehr zum rufenden Programm ausgelöst (RETURN-Befehl).

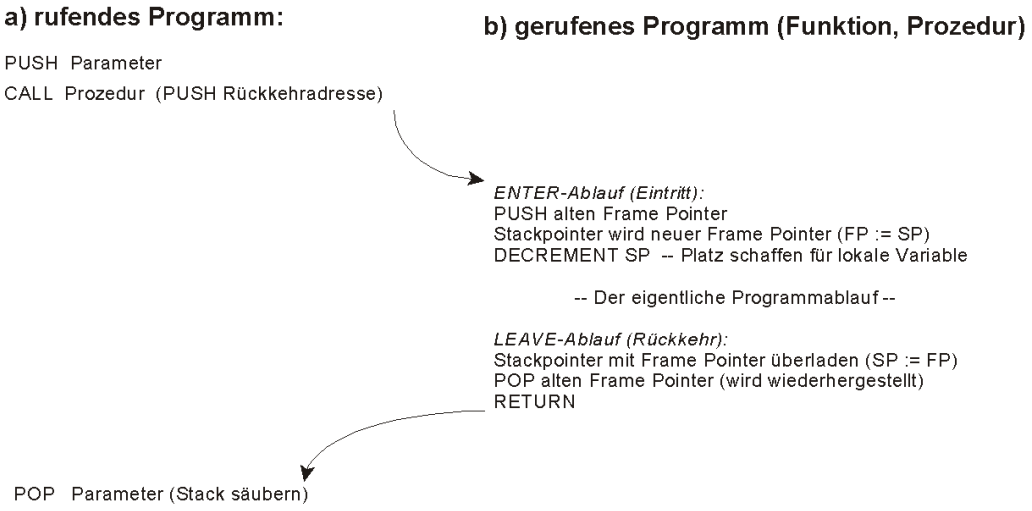


Abb. 3.62 Unterprogrammruf in einer Laufzeitumgebung, die auf Stack Frames beruht.

Die Stackbelegung anhand eines Beispiels
Deklaration einer Funktion:

```
int F_EX (int A, int B, double C);
{
int X, Y;
double Z;
float H, I;
...
....

return (Y);
}
```

Jetzt wird die Funktion aufgerufen:

```
...

OMEGA = F_EX (ALPHA, BETA, GAMMA);

...
```

Zunächst werden die Parameter auf den Stack gelegt, dann wird die Funktion aufgerufen (Abbildung 3.63):

```
PUSH_DOUBLE GAMMA -- Übergabe beginnt von hinten (C-Konvention)
PUSH BETA
PUSH ALPHA
CALL F_EX
```

Eintritt in die Funktion: Es wird zunächst der Eintrittscode ausgeführt (ENTER-Ablauf):

```
F_EX:   PUSH  FP       -- Frame Pointer auf Stack
        MOV   SP, FP   --  neuen FP einrichten
        DEC   SP, 24   -- die lokalen Variablen brauchen 24
                       -- Bytes
```

.... jetzt kommt der Funktionskörper von F_EX ....

Rückkehr (LEAVE-Ablauf):

```
MOV FP, SP      -- Zurückstellen des Stackpointers
POP FP          -- alten FP aus Stack zurückholen
RETURN
```

Das rufende Programm muss dann die übergebenen Parameter aus dem Stack entfernen, beispielsweise mit vier POP-Befehlen.

Ergebnisrückgabe
Es ist nichts standardisiert. Eine typische Praxislösung ist die Übergabe des berechneten Funktionswertes in einem bestimmten Register.

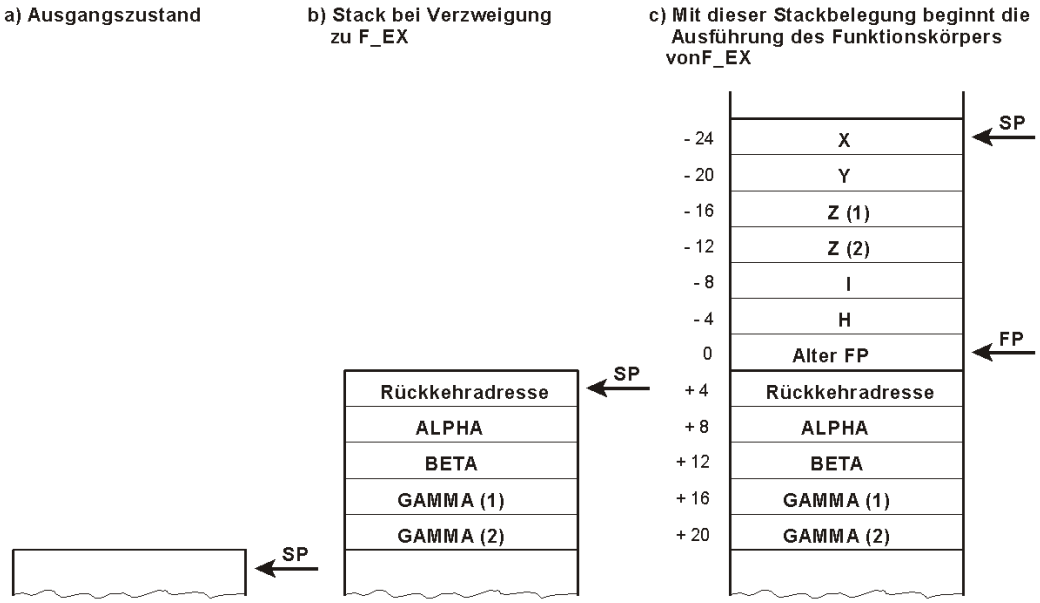


Abb. 3.63 Typische Stackbelegungen des Beispiels.

Homogene Datenstrukturen (Arrays)
Eine homogene oder Array-Struktur besteht aus gleichartigen Komponenten, auf die typischerweise in Schleifen zugegriffen wird, beispielsweise in FOR-Anweisungen. Die Komponenten werden über laufende Nummern (Ordinalzahlen) ausgewählt (Komponentenindex). Beispiele: a[x], b[x][y]. In den meisten Programmiersprachen gilt die Konvention, dass die erste Komponente den Index Null hat. Beispiele: a[0], b[0][0]. Jeder Zugriff erfordert eine Adressrechnung nach dem Prinzip Basis + Offset. Die Basisadresse ist die erste (niedrigste) Adresse der Datenstruktur. Wenn die Indexangaben im Array Variable sind, ist der Offset zur Laufzeit auszurechnen. Wird der Rechengang nicht von der Architektur unterstützt, muss er ausprogrammiert werden.

Zugriff auf eindimensionale Arrays:
b = Komponentenlänge, x = Komponentenindex (Abbildung 3.64a).

$$\text{Offset} = x \cdot b$$

Ist die Komponentenlänge b eine Zweierpotenz, kann die Multiplikation durch eine Linksverschiebung um ld b Bits ersetzt werden.

Zugriff auf zweidimensionale Arrays
z = Zeilenanzahl, s = Spaltenanzahl, b = Komponentenlänge, x = Zeilenindex, y = Spaltenindex (Abbildung 3.64b). Die Anzahlen zählen von Eins an.

Bei zeilenweiser Anordnung:

$$\text{Offset} = (x \cdot s + y) \cdot b$$

Bei spaltenweiser Anordnung:

$$\text{Offset} = (x + y \cdot z) \cdot b$$

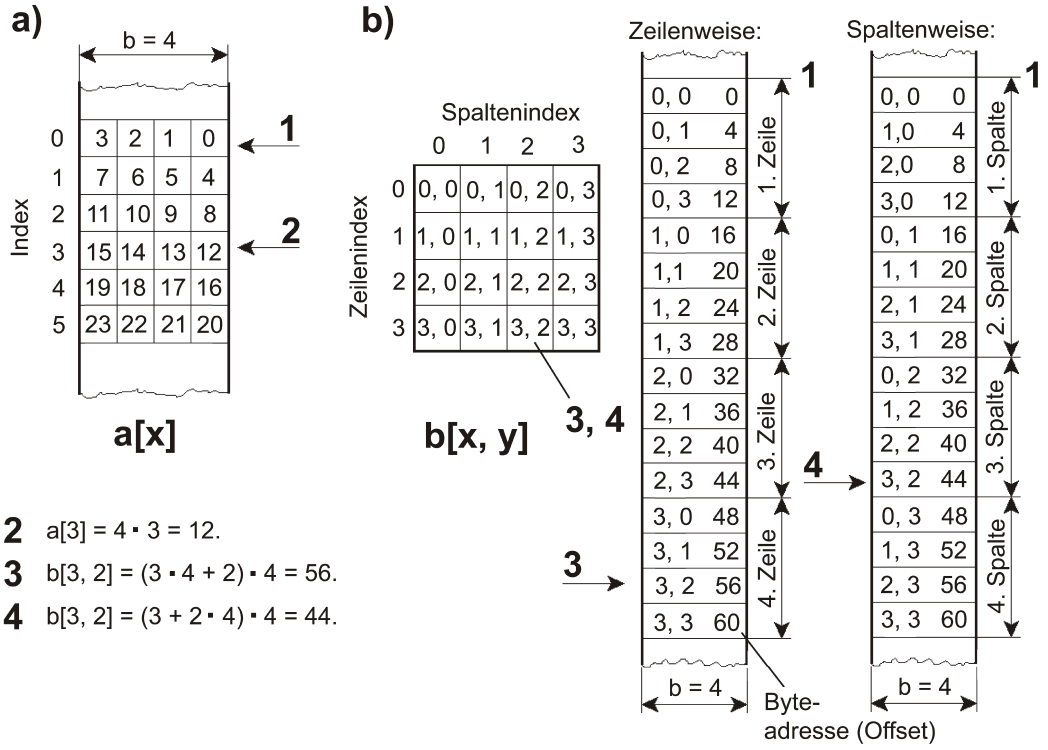


Abb. 3.64 Homogene Datenstrukturen. a) eindimensional (Vektor). Jede Komponente ist vier Bytes lang. Es sind alle Byteadressen angegeben. b) zweidimensional (Matrix). 1 - Basisadresse; 2 - Adressierungsbeispiel a[3]; 3, 4 - Adressierungsbeispiele b[3, 2][32].

3.7 v.Neumann-Architektur und Harvard-Architektur

Die Begriffe bezeichnen allgemeine Architekturkonzepte, die sich darin unterscheiden, wie viele Speicheradressräume und Speicherzugriffswege grundsätzlich vorgesehen sind (Abbildung 3.65).

32: Schreibweise der Indexangaben wie in der Mathematik üblich (nicht wie in C).

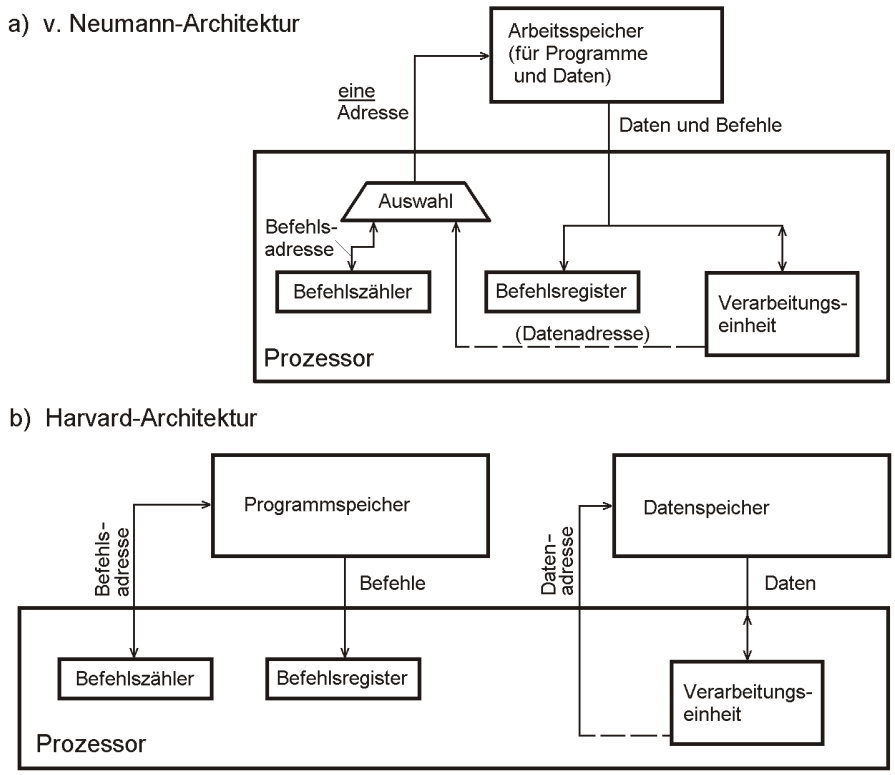


Abb. 3.65 v. Neumann- und Harvard-Architektur.

Zu den Namen:

- v. Neumann-Architektur: nach dem Mathematiker John v. Neumann.
- Harvard-Architektur: nach der Harvard-Universität (Cambridge, Massachusetts)[33].

v. Neumann-Architektur
Es gibt nur einen einzigen Speicheradressraum und einen einzigen Zugriffsweg. Mit anderen Worten: es gibt aus der Sicht des Programmierers nur einen einzigen, von Adresse Null an fortlaufend adressierbaren Speicher, der alle Programme, Daten usw. aufnimmt.

Harvard-Architektur
Es gibt zwei Speicheradressräume; einen für Daten und einen für Befehle. Die Harvard-Maschinen unterscheiden sich nach der Anzahl der Speicherzugriffswege und der Organisation des Speichersubsysteme:

Echte Harvard-Maschinen
Es gibt eine gesonderte Speicheranordnung je Adressraum (also einen Programmspeicher und einen Datenspeicher) und unabhängige Zugriffswege zu beiden Speicheranordnungen. Beispiel: AVR.

v. Neumann-Maschinen mit Harvard-Architektur
Diese haben zwar die beiden getrennten Adressräume, aber nur einen einzigen gemeinsamen Speicherzugriffsweg (Speicherbus). Beispiel: 8051.

33: Die ersten Maschinen mit dieser Architektur wurden von Konrad Zuse gebaut (in Berlin)...

Der große Vorteil der v. Neumann-Architektur: der einheitliche lineare Adressraum
Wir können mit der Speicherkapazität anstellen, was wir wollen. Vor allem können wir die gesamte installierte Speicherkapazität voll ausnutzen – ob wir vorwiegend Programme oder vorwiegend Daten speichern, ist gleichgültig. Auch lassen sich Programme ohne weiteres als Daten behandeln, also mit den üblichen Maschinenbefehlen transportieren, ändern usw.

Die Vorteile der Harvard-Architektur:

1. Höhere Leistung
Eine v. Neumann-Maschine holt Befehle und Daten nacheinander aus dem selben Speicher. Dies beeinträchtigt naturgemäß das Leistungsvermögen – man spricht bildhaft vom v. Neumann-Flaschenhals (v.Neumann Bottleneck). Eine echte Harvard-Maschine kann hingegen gleichzeitig (parallel) auf Daten und Befehle zugreifen.

2. Aufwandsoptimierung
Beide Speicher einer echten Harvard-Maschine kann man unabhängig voneinander hinsichtlich der Zugriffsbreite, der Technologie usw. optimieren. So liegt es nahe, den Programmspeicher eines Mikrocontrollers als (Flash-) ROM auszulegen und den Datenspeicher als RAM. Ist beispielsweise das Byte die elementare Datenstruktur, so müssen bei einer v. Neumann-Maschine die Befehlsformate in Bytestrukturen gezwängt werden. Eine echte Harvard-Maschine kann man hingegen so auslegen, dass der Befehlsspeicher eine jeweils genau passende Zugriffsbreite hat – auch wenn es ein "krummer" Wert ist. Unter anderem sind Prozessoren mit Befehlen von 12, 14, 24 Bits usw. gebaut worden.

3. Höheres Adressierungsvermögen
Da es zwei Adressräume gibt, wird das Adressierungsvermögen der Hardware praktisch verdoppelt. Beispiel: ein Prozessor ist für 16-Bit-Adressen ausgelegt (Byteadressierung). In einer v. Neumann-Architektur beträgt das Adressierungsvermögen insgesamt $2^{16} = 64$ kBytes. Eine Harvard-Maschine kann hingegen mit einem Befehls- und einem Datenspeicher von jeweils 64 kBytes bestückt werden (insgesamt 128 kBytes). Deshalb hat man viele Mikrocontroller, obwohl sie nur einen einzigen Speicherbus haben, als Harvard-Maschinen ausgelegt. Beispiel: 8051. Die schlechte Nachricht: wenn man beispielsweise 20 kBytes für die Programme, aber 100 kBytes für die Daten braucht, so passt es nicht. Praxistipp: Die meisten Maschinen haben Konstantenladebefehle, so dass man Festwerte ohne weiteres im Befehls-ROM unterbringen kann. Ist aber der Programmspeicheradressraum zu klein, wird es schwierig. Zumeist ist es das Beste, nicht mit Tricklösungen zu experimentieren, sondern eine andere Prozessorarchitektur auszuwählen.

Mikrocontroller und Signalprozessoren
Die meisten Signalprozessoren und auch viele der kleinen Mikrocontroller sind Harvard-Maschinen (Tabelle 3.22).

Universelle Hochleistungsprozessoren
Die modernen Hochleistungsprozessoren sind v.Neumann-Maschinen. Das heißt, sie verhalten sich aus Sicht des Programmierers als solche. Es ist aber möglich, die Vorteile beider Architekturen zu vereinigen. Wenn man getrennte Befehls- und Datencaches einsetzt, ergibt sich für den Prozessorkern eine Art Harvard-Architektur mit unabhängigen Zugriffswegen für Daten und Befehle (Abbildung 3.66).

Vorteil	Erläuterung	Beispiele
Höheres Adressierungs-vermögen	Weil es zwei unabhängige Adressräume gibt	8051
Technologische Trennung zwischen Daten- und Programmspeicher	Datenspeicher: SRAM, Programmspeicher: ROM (z. B. Flash)	PIC, AVR
Interne Optimierung	Die Befehle werden so breit ausgelegt wie es jeweils zweckmäßig ist (also nicht in Bytestrukturen gepresst)	PIC16x, 24x
Leistungssteigerung	Durch parallelen Zugrifff auf Befehle und Daten	Die meisten Signalprozessoren

Tabelle 3.22 Weshalb Mikrocontroller und Signalprozessoren oft als Harvard-Maschinen ausgelegt werden.

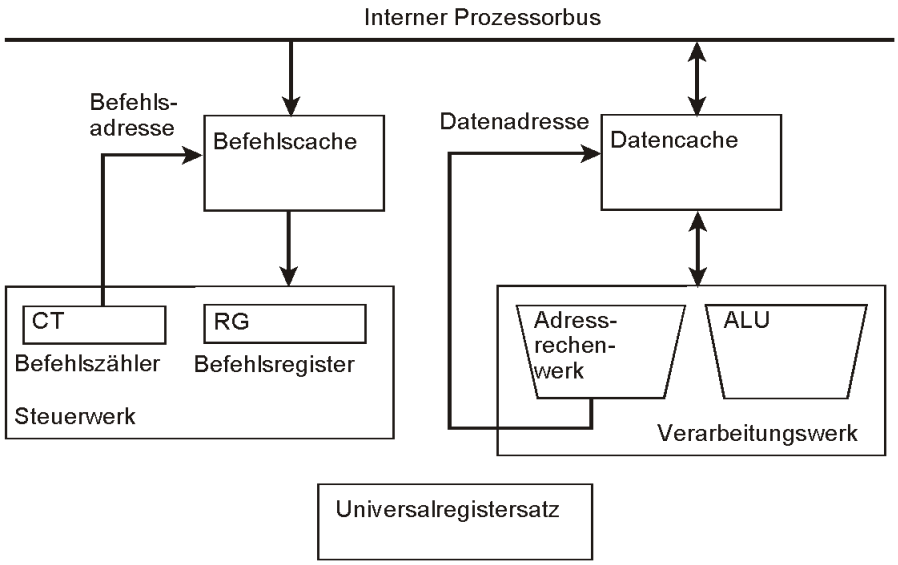


Abb. 3.66 Durch Befehls- und Datencaches mit unabhängigen Zugriffswegen wird der Prozessorkern zur Harvard-Maschine.

3.8 CISC und RISC

Die Marketingleute vieler Anbieter scheinen der Auffassung zu sein, RISC klinge – und sei womöglich auch – von Grund auf besser. Was besser klingt, ist hier nicht zu entscheiden (Geschmacks- und Gefühlssache). In der Praxis ist der Name aber Schall und Rauch ...

RISC für Praktiker

Einige extreme akademische Auffassungen sind aufgegeben worden. So werden auch Befehlswirkungen implementiert, die sich nicht in einem einzigen Taktzyklus erledigen lassen (unter anderem alle vier Grundrechenarten). Vor allem aber werden Wartezustände und Konflikte in der Befehlspipeline von der Hardware behandelt. Der ursprüngliche Ansatz hat darin bestanden, gar keine Schaltmitel vorzusehen, sondern alles vom Compiler erledigen zu lassen (der die Abhängigkeiten erkennen und ggf. Füllbefehle einschieben muss). Dann würden aber Architektur und Hardware eng zusammenhängen; die Struktur der Pipeline würde auf die Programmschnittstelle (API) durchschlagen. Das ist im Grunde eine Verletzung des Architekturgedankens, denn eigentlich sollte die Befehlsliste als Programmschnittstelle (API) die Architektur von der Hardware des Prozessors isolieren (eine Architektur, viele

verschiedene kompatible Implementierungen). Nur so ist es möglich, binärkompatible Programme anzubieten, die aus der Verpackung heraus auf jeder beliebigen programmkompatiblen Maschine installiert werden können (Shrinkwrapped Software). Für jede neue akademische RISC-Maschine müssten die Programme hingegen neu compiliert werden.

Von den akademischen RISC-Grundsätzen verbleiben:

- Große Universalregistersätze (16 oder 32 Register sind typisch). Man ist bestrebt, die lokalen Variaben der aktuellen Funktion nicht in einem Stack Frame im Arbeitsspeicher zu halten, sondern im Registersatz.
- Das Load-Store-Prinzip (Operationsbefehle mit zwei oder drei Registeradressen, Speicherzugriffe nur mit Lade- und Speicherbefehlen).
- Der einfache Unterprogrammruf, wobei die Rückkehradresse in ein Rettungsregister geladen wird. Stackmechanismen müssen ausprogrammiert werden.
- Der Verzicht auf die Unterstützung variabel langer Datentypen, wie Zeichenketten und Dezimalzahlen.

In Tabelle 3.23 sind typische Merkmale von CISC- und RISC-Architekturen gegenübergestellt. Abbildung 3.67 zeigt das Blockschaltbild eines Prozessorkerns, der – jeweils entsprechend abgewandelt – als CISC- oder RISC-Maschine gebaut werden kann. Abbildung 3.68 veranschaulicht entsprechende generische Befehlsformate.

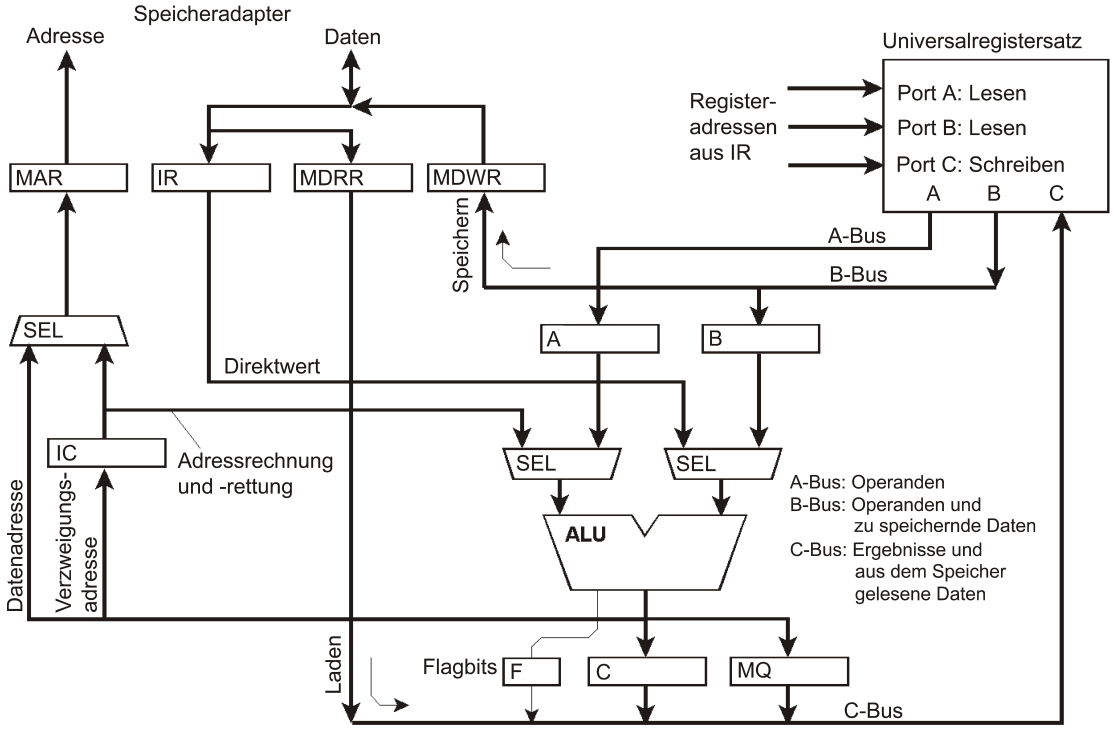


Abb. 3.67 Ein Prozessorkern im Blockschaltbild.IC = Befehlszähler; MAR = Speicheradressregister; IR = Befehlsregister; MDRR = Speicherdatenleseregister; MDRW = Speicherdatenschreibregister.

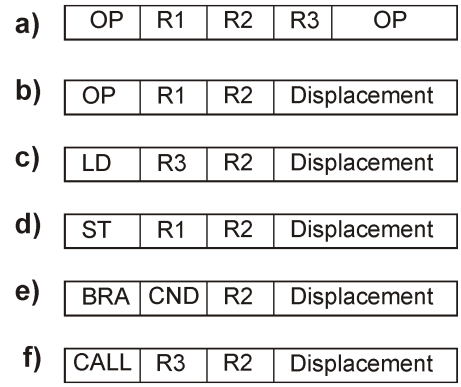


Abb. 3.68 Generische Befehlsformate. a) Verarbeitungsbefehl RISC. b) Verarbeitungsbefehl CISC. R1 - Operanden- und Ergebnisregister; c) Ladebefehl. R3 - Zielregister. d) Speicherbefehl. R1 - Quellregister; e) Verzweigungsbefehl. CND - Bedingungsauswahl. f) Unterprogrammruf. R3 - Rettungsregister. In b) bis f) ist R2 immer das Basisadressregister.

CISC	**RISC**
Komplexe Befehle	**Einfache Befehle**
Was ist hier komplex? • Operationsbefehle mit Speicherzugriff, • verschiedene Arten der Adressrechnung, • Unterstützung aller vier Grundrechenarten, • Unterstützung variabel langer Datentypen (Dezimalzahlen, Zeichenketten usw.), • Blockoperationen (Transportieren, Wandeln, Vergleichen), • Unterstützung komplizierter Organisationsabläufe (Funktionsaufruf, Taskumschaltung usw.).	Was ist hier einfach? • Operationsbefehle wirken nur auf Registerinhalte, • Trennung von Speicherzugriffs- und Operationsbefehlen (Load /Store), • nur einfache Adressrechnung in den Speicherzugriffsbefehlen (Basis + Displacement), • es werden nur Datentypen fester Länge unterstützt, • nur Operationen, die in einem Maschinenzyklus ablaufen können, • nur elementarer Unterprogrammruf, • alles, was komplizierter ist, muss ausprogrammiert werden.
So großartig komplex ist CISC aber auch nicht – die meisten Befehle haben nur einen Register- und einen Speicheroperanden: <R> := <R> **op** <MEM>	
Befehle variabler Länge. Kompakt, aber aufwendig zu decodieren (Decodierung kann mehrere Pipeline-Stufen erfordern).	Befehle fester Länge. Teils Platzverschwendung, aber einfach zu decodieren (oft genügt eine einzige Pipeline-Stufe).
Steuerung kompliziert. Mikroprogrammsteuerung.	Steuerung einfach. Sequentielle Steuerung.
Das Leistungsvermögen des einzelnen Befehls ist entscheidend (darf länger dauern, wenn er entsprechend viel leistet). Anwendungsseitige Eleganz, die die Assemblerprogrammierung erleichtert.	Die Befehlsauführungszeit entscheidet. Alle Befehlsabläufe müssen in ein Pipeline-Schema passen, sonst werden sie nicht implementiert (ohne Rücksicht auf Eleganz und Komfort). Ohne Compiler nur schwer zu programmieren.
Das Anwendungsproblem wird mit vergleichsweise wenigen, leistungsfähigen, langsam ablaufenden Befehlen gelöst. Die Speicherbandbreite ist vergleichsweise unkritisch. Wegen des kompakten Codes sind die Trefferraten der Caches hoch. Somit genügen vergleichsweise kleinen Caches.	Das Anwendungsproblem wird mit elementaren, schnell ablaufenden Befehlen gelöst. Deshalb werden mehr Befehle benötigt. Es kommt auf die Speicherbandbreite an. Für eine befriedigende Leistung braucht man große Caches.

Tabelle 3.23 CISC und RISC.

Eine CISC-Maschine kommt mit zwei Zugriffswegen zum Universalregistersatz aus, braucht aber eine komplizierte Steuerung. So muss in einem Verarbeitungsbefehl gemäß Abbildung 3.68b zunächst der Speicheroperand gelesen werden. Die Adressrechnung läuft hier über die ALU. Der Ablauf: Basiadresse nach Operandenregister B – das Displacment aus dem Befehlsregster dazu addieren – Ergebnis nach MAR – Lesezugriff auslösen – R1 nach Operandenregister A – die gelesenen Daten aus MDRR über R1 nach Operandenregister B – Operation ausführen – Ergebnis nach R1 – nächsten Befehl lesen. Ein RISC-Befehl gemäß Abbildung 3.59a könnte hingegen in schlimmstenfalls drei Taktzyklen erledigt werden: R1 und R2 gleichzeitig nach A und B – Operation ausführen – Ergebnis nach R3. Wenn man es passend einrichtet (Frage der Auslegung des Universalregistersatzes) könnte man mit einem einzigen – entsprechend längerem – Taktzyklus auskommen. Die Register A, B, C könnten dann wegfallen.

4. Universelle E-A-Ports

4.1 Einfache E-A-Ports

Ein universeller E-A-Anschluss muss wahlweise als Eingang oder Ausgang wirken können. Um einen Signalweg so anzusteuern, dass er in beiden Richtungen (bidirektional) betrieben werden kann, stehen zwei Schaltungsauslegungen zur Wahl: die Transistorstufe mit Arbeitswiderstand (Open Collector, Open Drain, Open Source) und die Tri-State-Stufe.

Am Anfang der Entwicklung ging es knapp zu. Bei der Schaltungsauslegung kam es auf jeden Transistor an. Die E-A-Ports der ersten Mikrocontroller waren typische Sparlösungen. Manche dieser Typen werden bis heute gefertigt. Dabei werden neue Technologien angewendet, die Schaltkreise werde in neuartigen Gehäusen angeboten usw. Die Schaltungsauslegung der Ports wird aber zumeist nicht modernisiert, sondern beibehalten (Kompatibilität). Deshalb sind auch heute noch Spitzfindigkeiten zu beachten, die sich seinerzeit als Folge der Sparsamkeit ergeben haben. Gespart wurde vor allem beim Rücksetzen der Portregister, beim Zurücklesen der Registerinhalte, bei Einzelbitfunktionen und bei den Portadressen.

Open Collector- oder Open-Drain-Ausgänge
Diese Ausgangsstufen brauchen keine Richtungssteuerung. Zur Ausgabe genügt ein Datenregister (Abbildung 4.1). Der Port benötigt somit nur eine einzige E-A-Adresse. Da der Treibertransistor wie ein Negator wirkt, muss er invertiert angesteuert werden (eine Null im Datenregister muss einen Low-Pegel am Anschluss bewirken, eine Eins einen High-Pegel). Soll der Anschluss als Eingang betrieben werden, wird die Bitposition im Datenregister mit einer Eins geladen. Dann wird der Treibertransistor nicht angesteuert, und am Anschluss ergibt sich ein High-Pegel. Führt das Eingangssignal einen High-Pegel, so ändert sich nichts. Hingegen zieht ein Low-Pegel den Anschluss auf Low. Dabei muss die jeweils ansteuernde Einrichtung den Strom aufbringen, der durch den Arbeitswiderstand fließt.

Elementare Anwendungsprogrammierung:

- Ausgabe: die betreffenden Bitpositionen des Datenregisters mit dem gewünschten Wert laden.
- Eingabe: die betreffenden Bitpositionen des Datenregisters mit Einsen laden. Diese müssen stehen bleiben, wenn die Anschlüsse als Eingänge genutzt werden sollen. Die Treibertransistoren werden dann nicht angesteuert.
- Der Anfangszustand (nach dem Rücksetzen): alles auf Eingabe (Datenregister gesetzt, Anschlüsse auf High-Pegel).

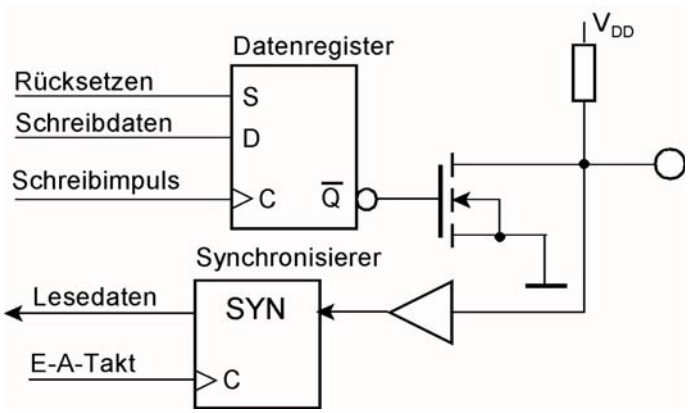


Abb. 4.1 E-A-Port mit Open-Drain-Ausgang (eine Bitposition).

Tri-State-Ausgänge

Eine Tri-State-Stufe kann beide Signalpegel aktiv treiben. Sie hat einen Erlaubniseingang, über den die Betriebsart gesteuert wird. Ist das Erlaubnissignal inaktiv, so wird der Signalanschluss nicht getrieben. Er kann somit als Eingang genutzt werden. Der elementare Tri-State-Port benötigt zwei Register (und somit zwei Adressen): ein Richtungssteuerregister und ein Datenregister (Abbildung 4.2). Die meisten Mikrocontroller haben E-A-Anschlüsse mit Tri-State-Stufen.

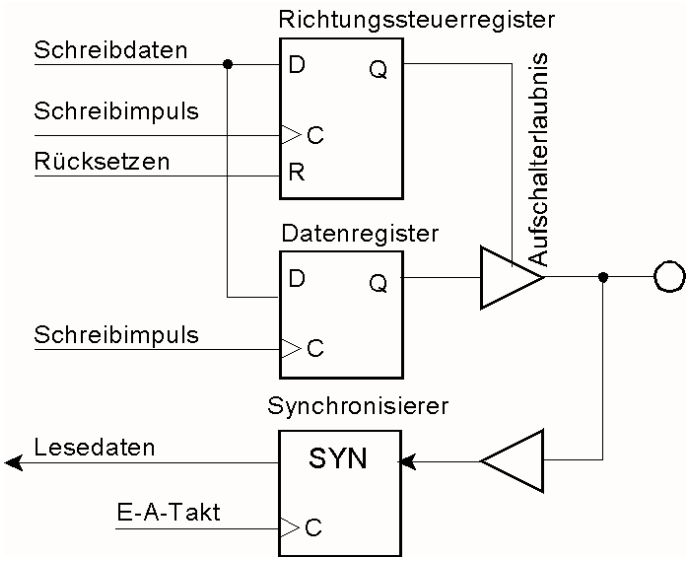


Abb. 4.2 E-A-Port mit Tri-State-Ausgang (eine Bitposition).

Elementare Anwendungsprogrammierung:

- Ausgabe: die betreffenden Bitpositionen des Richtungssteuerregister mit Einsen laden[34]. Der zugehörige Inhalt des Datenregisters erscheint an den Anschlüssen.
- Eingabe: die betreffenden Bitpositionen des Richtungssteuerregisters mit Nullen laden. Die Treiberstufen werden hochohmig. Somit dürfen die Anschlüsse von außen getrieben werden.

34: Das betrifft nur E-A-Ports, die gemäß Abbildung 4.2 ausgelegt sind (Beispiele: AVR und PIC). Es gibt auch E-A-Ports, deren Richtungssteuerregister anders herum wirkt (Null = Ausgabe; Eins = Eingabe).

- Lesen: Es werden stets der Signalpegel an den Anschlüssen gelesen. Je nach Inhalt des Richtungssteuerregisters handelt es sich um den Inhalt des Datenregisters (Ausgabe) oder um ein außen anliegendes Signal (Eingabe).
- Der Anfangszustand (nach dem Rücksetzen): alles auf Eingabe (Anschlüsse hochohmig).
- Umschalten in die Ausgaberichtung: erst die Daten, dann die Richtungssteuerung. Sonst stellen sich an den Anschlüssen kurzzeitig die anfänglichen Bitmuster des Datenregisters ein (Abbildung 4.3). Das kann in der Anwendungsschaltung sehr hässliche Nebenwirkungen haben.

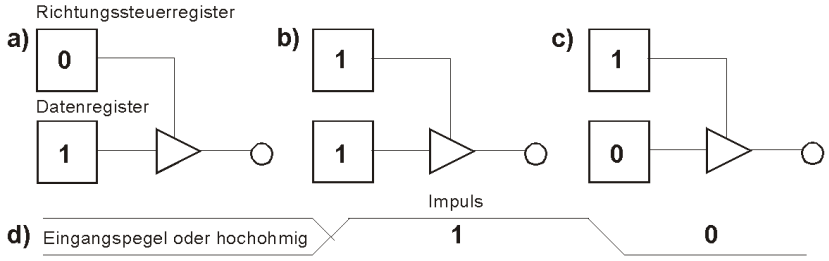


Abb. 4.3 Eine typische Fehlerquelle. a) am Anfang steht die Bitposition auf Eingabe. Der Inhalt des Datenregisters ist an sich gleichgültig. b) zuerst wird die Richtung auf Ausgabe gestellt. c) dann wird das Datenregister geladen. d) der Signalverlauf am Anschluss. Hat zuvor der jeweils andere Wert im Datenregister gestanden, erscheint ein Impuls. Es hängt von der Anwendungsschaltung ab, ob er schadet oder nicht.

Rücksetzen und Initialisierung

Der Mikrocontroller kann nicht von vornherein wissen, in welcher Umgebung er zum Einsatz kommt. Deshalb müssen seine Anschlüsse unmittelbar nach dem Einschalten als Eingänge konfiguriert sein – sie dürfen die Signalwege so lange nicht treiben, bis sie vom Initialisierungsprogramm anwendungspezifisch eingerichtet wurden. Bei Open-Drain-Stufen muss das Datenregister gesetzt werden. Bei Tri-State-Stufen ist das Richtungssteueregister so zu stellen, dass sich die Anschlüsse als Eingänge verhalten (im Beispiel von Abbildung 4.2 ist es zu löschen). Nun benötigt ein Flipflop mit Setz- oder Rücksetzeingängen einige Transistoren mehr als eines ohne. Deshalb hat man gelegentlich nur das Nötigste getan, also nur das Richtungssteuerregister zurückgesetzt, nicht aber das Datenregister. Der Grundgedanke: weil die Anfangseinstellung der Ausgänge anwendungsspezifisch ist, muss man sie sowieso programmseitig einstellen[35], also kann man auf das Rücksetzen des Datenregisters verzichten.

Ports zurücklesen

Manchmal ist es erforderlich, die Belegung von Ausgängen wieder einzulesen. Fehler treten vor allem dann auf, wenn die zurückgelesenen Bitmuster verwendet werden, um Registerinhalte zu modifizieren (Ablauffolge Read – Modify – Write).

Wie werden Einzelbits und Bitfelder beeinflusst?

Der übliche Weg führt vom E-A-Port in die Arithmetik-Logik-Einheit (ALU) des Prozessors und von dort zurück zum Port (Abbildung 4.4). Die Bitbeeinflussung in der ALU beruht auf elementaren bitweisen Verknüpfungen. Einzelbitbefehle sehen zwar komfortabel aus, erledigen ihre Aufgabe aber auch nur auf diesem Wege. So wird beispielsweise ein Befehl zum Setzen von Bitposition 3 im Datenregister eines Ports als ODER-Verknüpfung ausgeführt

35: Das wird aber in der Praxis gelegentlich vergessen...

(Datenregisterinhalt ODER Festwert 0000 1000B). Aufgrund dieser Arbeitsweise können sich Datenverfälschungen ergeben. Solche Fehler sind schwer zu finden, weil sie nur dann auftreten, wenn tatsächlich abweichende Bitmuster entstehen (datenabhängige Fehler).

Das ungestörte bitweise Modifizieren ist besonders wichtig, wenn verschiedene Tasks verschiedene Interfaces steuern, die an gemeinsame Ports angeschlossen sind. Beispiel: Task 1 steuert die Bits 2 und 5; Task 2 steuert die Bits 1, 6, und 7. Jede Task muss sich darauf verlassen können, dass sie ihre Bits immer so vorfindet, wie sie sie hinterlassen hat.

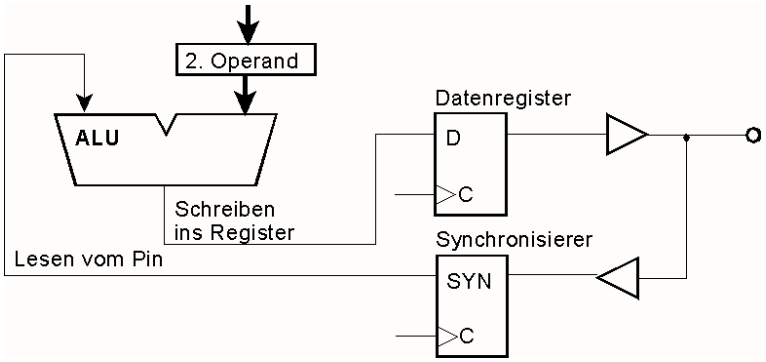


Abb. 4.4 Prinzip der Bitmodifkation über die Arithmetik-Logik-Einheit (ALU).

Rücklesen der Portbelegung (1). Die Zeitfrage
Ausgegebene Signalbelegungen können typischerweise nicht sofort zurücklesen werden. Wenn wir zu zeitig zurücklesen, so lesen wir die alten Signalpegel. Beispiel: Erst soll Bit 4 eingeschaltet werden und einen Befehl später Bit 5. Der Befehl SET Bit 5 holt aber die Ausgangsbelegung *vor* SET Bit 4 zurück, so dass Bit 4 als Null in die Verknüpfung eingeht und folglich beim Modifizieren wieder gelöscht wird (Abbildung 4.5).

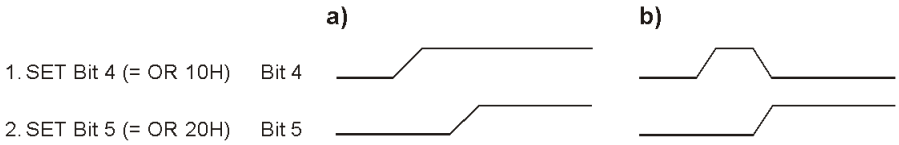


Abb. 4.5 Theorie und Praxis. a) Programmierabsicht; b) der tatsächliche Signalverlauf an den Anschlüssen

Die wichtigste Ursache ist die zum Synchronisieren nötige Eintaktierungszeit. Die Anzahl der Takte, die zwischen Ausgabe und Rücklesen mindestens durchlaufen werden müssen, steht im Datenblatt. Wie schnell sich eine Änderung des Datenregisterinhalts am Anschluss bemerkbar macht (und somit zurückgelesen werden kann), hängt auch von der kapazitiven Belastung des Anschlusses ab. Die Synchronisation kann erst dann wirksam werden, wenn der Signalpegel den jeweils gewünschten Wert erreicht hat. Bei hoher kapazitiver Belastung und höheren Taktfrequenzen (20 MHz und mehr) können sich die Anstiegs- und Abfallzeiten so verlängern, dass die Synchronisation erst mit einer Verzögerung von einer Taktperiode (oder mehr) einsetzen kann.

Weitere Problemstellen
Die Verzögerung zwischen Ausgabe und Eingabe ist auch in folgenden Fällen zu berücksichtigen:

- Beim Zurücklesen zu Testzwecken (beispielsweise um nachzusehen, ob die Treiberstufen richtig funktionieren).
- Dann, wenn Eingänge gelesen werden, deren Signalwerte von zuvor ausgegebenen Bitmustern abhängen. Das ist beispielsweise dann der Fall, wenn das typische Verhalten von Zustandsautomaten implementiert wird. Die neue Eingabe hängt – vermittelt über die Umwelt des Automaten – von der vorhergegangenen Ausgabe ab. Wenn man nach dem Ausgabebefehl sofort den Eingabebefehl folgen lässt, ergibt sich das beschriebene Problem.

Abhilfe 1
Zwischen Ausgabe und Rücklesen Wartebefehle (einfachste Lösung: NOPs) einfügen. Praxistipp: Einen Makro definieren, der nach Bedarf angepasst werden kann. Für einen langsamen Prozessor wird er beispielsweise als ein einziger NOP implementiert, für einen schnellen Prozessor hingegen als Zählschleife.

Abhilfe 2
Solche Sequenzen stets durch Schreiben ganzer Bytes programmieren:

Statt

```
SET Bit 4
SET Bit 5
```

also beispielsweise

```
OUT 10H
OUT 11H
```

Dazu muss man allerdings wissen, wie die anderen Bits belegt sind. Beim Multitasking funktioniert das nicht.

Rücklesen der Portbelegung (2). Inhaltsverfälschungen
Das Problem tritt auf, wenn der Datenregisterinhalt modifiziert werden soll, aber nur die Signale an den Anschlüssen zurücklesen werden (Abbildung 4.6). Auch dann, wenn nicht vorzeitig zurückgelesen wird, ist achtzugeben, und zwar abhängig von der eingestellten Übertragungsrichtung:

- Ist der Anschluss ein Ausgang, so ist alles in Ordnung; der Signalpegel entspricht dem Registerinhalt.
- Ist der Anschluss ein Eingang, so wird der am Anschluss anliegende Signalpegel gelesen. Dieser Wert wird beim Modifizieren zum neuen Registerinhalt. Wird jetzt die Bitposition des Ports zum Ausgang umgeschaltet, so wird nicht ein ggf. zuvor geladener Registerinhalt wirksam, sondern der zuletzt gelesene Eingangspegel.

Das ist dann kein Problem, wenn die Eingänge stets Eingänge bleiben. Es wird aber nicht funktionieren, wenn die Übertragungsrichtung umgestellt wird und wenn man sich dabei auf den Inhalt des Datenregisters verlässt (Abbildung 4.6a bis e). Ein typisches Beispiel ist die Nachbildung des Open-Collector-Verhaltens an einem Tri-State-Port (Abbildung 4.6f bis h). Das Datenregisterbit ist stets Null, und das Richtungssteuerregisterbit bestimmt die Anschlussbelegung. In Eingaberichtung ist der Port hochohmig. Der Anschluss wird über einen externen Arbeitswiderstand auf High gezogen (Abbildung 4.6f). In Ausgaberichtung wird die Null aus dem Datenregister auf den Anschluss geschaltet (Abbildung 4.6g). Der naive Programmierer setzt die Bitposition des Datenregisters bei der Initialisierung auf Null und

kümmert sich dann nie mehr darum. Nun stehe ein solcher Anschluss auf High (Eingangsrichtung). Jetzt finde – in ganz anderem Zusammenhang – ein Lesen mit nachfolgender Modifikation des Datenregisterinhalts statt. Dann wird die Null im Datenregister von außen mit einer Eins überschrieben werden. Infolgedessen wird die Open-Collector-Nachbildung nie mehr einen Low-Pegel ausgeben können (Abbildung 4.6h).

Was wird bei Lesezugriffen wirklich gelesen?
Es gibt verschiedene Auslegungen sparsamer E-A-Ports:

- Nur die Anschlüsse des Ports (Beispiel: PIC 16C5x). Der Inhalt des Richtungssteuerregisters kann nicht modifiziert werde. Deshalb gibt es besondere Ladebefehle für die Richtungssteuerung.
- Richtungsregister und Portbelegung (Beispiel: diverse PIC-Typen). Dies ermöglicht es, den Inhalt des Richtungssteuerregisters zu modifizieren.
- Das Richtungssteuerregister bestimmt, von wo das Datenbit zurückgelesen wird (Beispiel: Renesas H8/300H). Ist der Anschluss ein Eingang, wird der anliegende Signalpegel zurückgelesen, ist der Anschluss ein Ausgang, das Datenregister (Abbildung 4.7). Diese Auslegung beseitigt das Zeitproblem. Die zur Ausgabe vorgesehenen Bits werden aus dem Register geholt, sind also nicht der Zeitverzögerung am Port unterworfen. Es bleibt aber das Problem der Inhaltsverfälschung durch Überladen von Datenregisterpositionen, die auf Eingabe geschaltet sind.

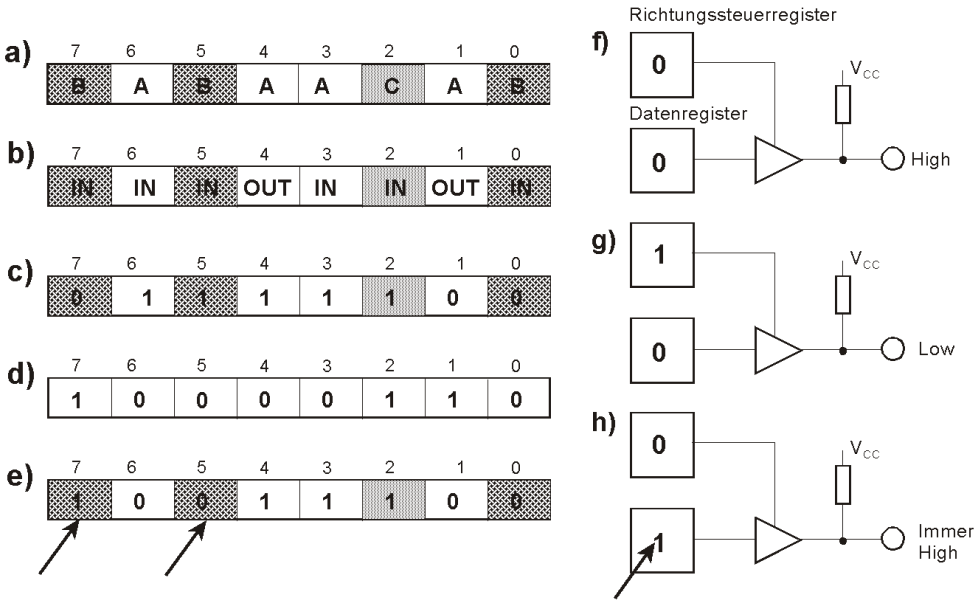


Abb. 4.6 Inhaltsverfälschungen in einem Datenregister. a) bis e) Ablaufbeispiel; f) bis h) die Auswirkung in einem typischen Anwendungsfall (s. Text). a) Zuordnung der Bitpositionen zu den Tasks A, B, C; b) die zugehörigen Richtungseinstellungen; c) anfängliche Belegung des Datenregisters; d) diese Werte liegen an den Pins und werden eingelesen; e) da auf dem Wege über die ALU das gesamte Datenregister überschrieben wird, werden manche der auf Eingabe geschalteten Bitpositionen verfälscht (Pfeile). Wenn sich die Task B darauf verlässt, dass die ursprünglich eingeschriebenen Werte (c) auch weiterhin drinstehen, so geht es schief...

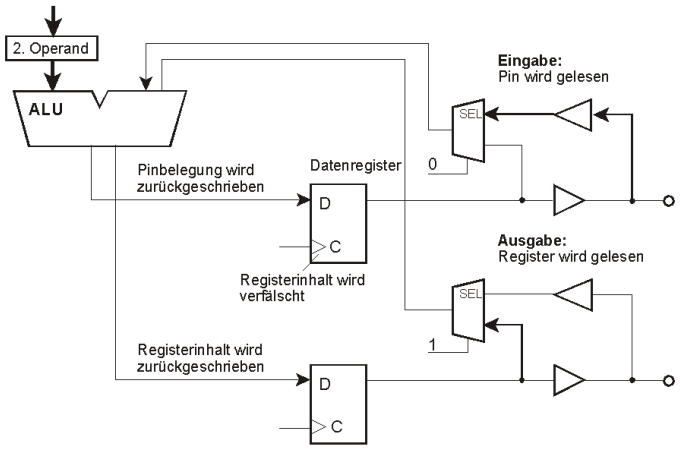


Abb. 4.7 Hier ist das Lesen von der Richtungssteuerung abhängig.

Ports ohne Rückleseprobleme

Die drei im folgenden beschriebenen Grundsatzlösungen haben den Charakter von Industriestandards.

1. Das befehlsgesteuerte Rücklesen

Was der Prozessor zurückliest, hängt vom jeweiligen Maschinenbefehl ab. Beispiel: 8051. Alle Befehle, die durch einen Read-Modify-Write-Ablauf gekennzeichnet sind, lesen das Datenregister, alle anderen die E-A-Anschlüsse. Beispiel: 8051 (Tabelle 4.1, Abbildung 4.8).

Befehl	Wirkung
ANDL; ORL; XRL; CPL	logische Verknüpfungen UND, ODER, XOR, Negation
JBC	Verzweigen, wenn Bit = 1 und Bit löschen
INC, DEC	Erhöhen oder Vermindern um 1
DJNZ	Vermindern und verzweigen, wenn nicht Null
MOV Px.y,C	Übertrags-Flagbit in Bit y des Ports x schreiben
CLR Px.y; SET Px.y	Bit y des Ports x löschen oder setzen

Tabelle 4.1 8051-Befehle, die bewirken, dass das Datenregister gelesen wird.

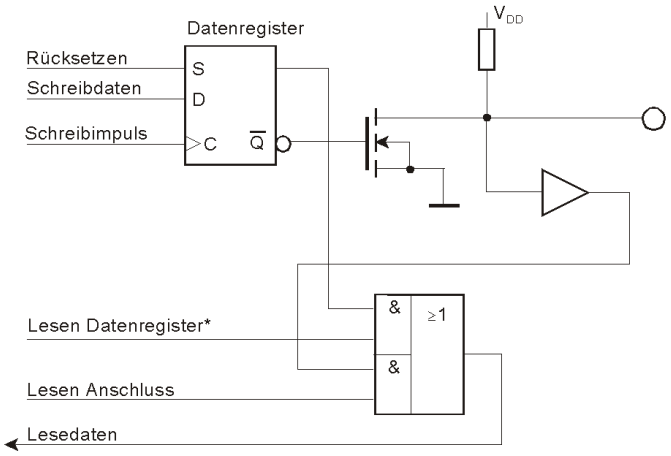


Abb. 4.8 Befehlsgesteuertes Rücklesen.

2. Drei E-A-Adressen
Für jeden Port werden drei E-A-Adressen vorgesehen, so dass neben den beiden Registern auch die Anschlüsse zum Lesen ausgewählt werden können (Abbildung 4.9). Beispiel: AVR. Die E-A-Adressen werden wie folgt verwendet:

- Richtungssteuerregister: zum Einstellen der Signalflussrichtung.
- Datenregister: zur Ausgabe.
- Anschlüsse: zur Eingabe. Die Bits gelangen direkt in den Prozessor; der Inhalt des Datenregisters wird nicht verändert.

3. Bitbeeinflussung im E-A-Port
Die Bitbeeinflussung erfolgt nicht über die Arithmetik-Logik-Einheit (ALU) des Prozessors, sondern direkt im E-A-Port. Hierzu wird der Port um zwei Register erweitert, das Setzregister und das Löschregister (Abbildung 4.10 und 4.11). Manche Ports haben ein weiteres Register zum Umschalten des Datenregisterinhalts (Toggle-Register).

Jedes Register hat eine eigene Adresse. Das Richtungssteuerregister und das Datenregister sind auch für Lesezugriffe zugänglich. Die in den Abbildungen 4.10 und 4.11 gestrichelt dargestellten Register speichern nichts; es sind keine richtigen Register, sondern Schaltmittel, die dann wirken, wenn ein Zugriff mit der jeweiligen Adresse ausgeführt wird. Die Setz-, Lösch- und Umschaltregister sind nur für Schreibzugriffe vorgesehen, das Eingangsregister kann nur gelesen werden. Diese Zugriffsregister wirken auf jene Bitpositionen des Datenregisters, die durch eine Eins in den Schreibdaten gekennzeichnet sind. Alle anderen Bitpositionen bleiben unverändert (Abbildung 4.12).

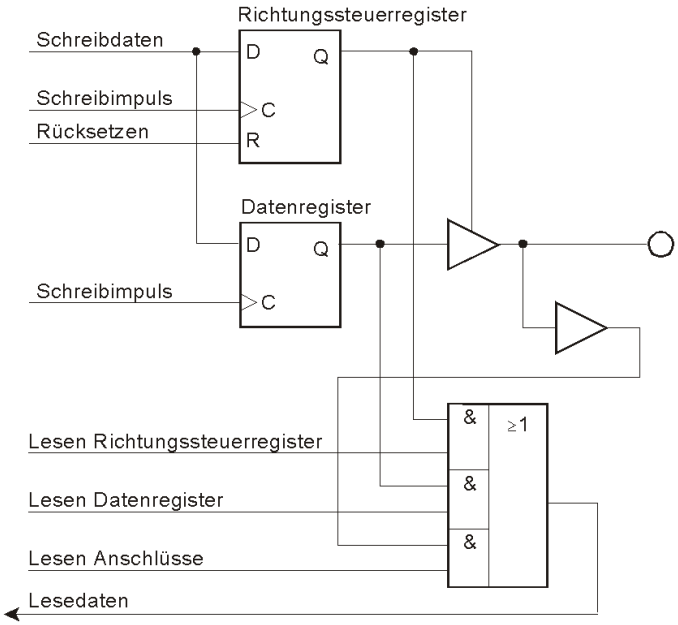


Abb. 4.9 Dieser E-A-Port unterstützt drei Lesezugriffe.

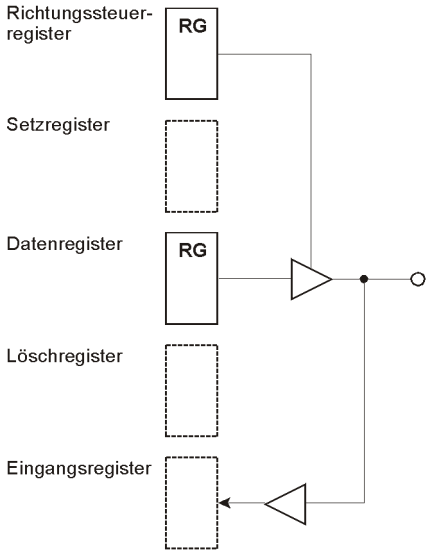


Abb. 4.10 E-A-Port mit Bitbeeinflussung. Das Registermodell aus der Sicht des Programmierers.

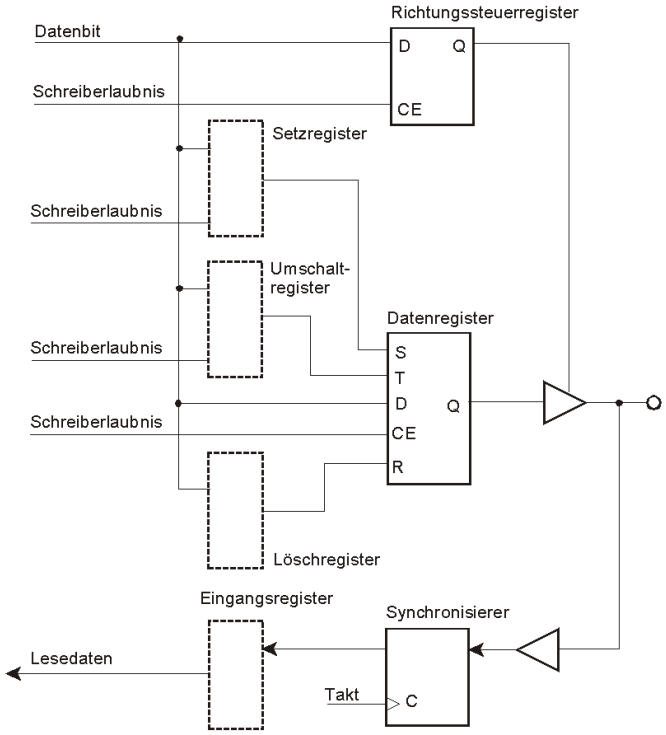


Abb. 4.11 Prinzipschaltung der Bitbeeinflussung über Registeradressn zum Setzen, Löschen und Umschalten von Bits (Zugriffsregister).

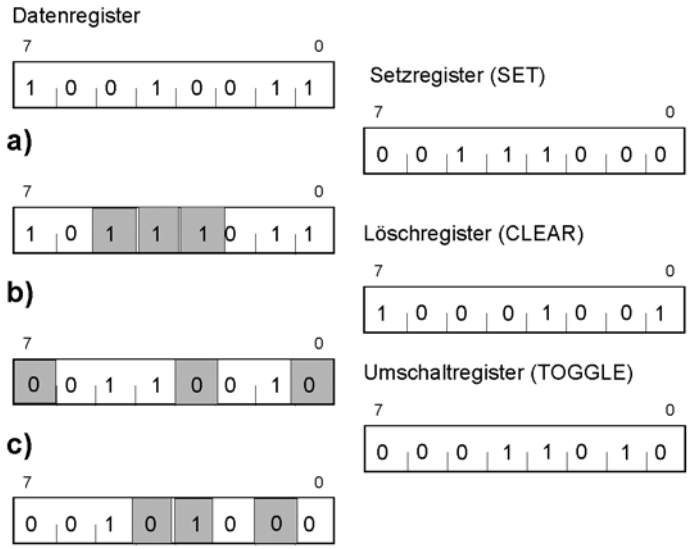


Abb. 4.12 Zugriffsbeispiele zur Wirkung der Setz-, Lösch- und Umschaltregister.

a) Setzen: Eine Eins im Setzregister bewirkt, dass die zugehörige Bitposition des Datenregisters gesetzt wird.
b) Löschen: Eine Eins im Löschregister bewirkt, dass die zugehörige Bitposition des Datenregisters gelöscht wird.
c) Umschalten: Eine Eins im Umschaltregister bewirkt, dass die zugehörige Bitposition des Datenregisters ihren Wert wechselt (von 0 nach 1, von 1 nach 0).

4.2.2 Neuzeitliche E-A-Ports

Heutzutage kommt es nicht mehr auf jedes Adressbit und Flipflop an. Die E-A-Ports werden großzügig ausgestattet. Zusätzliche Register dienen zur Steuerung der Ein- und Ausgangsstufen in Hinsicht auf Signalpegel, Flankensteilheit usw. sowie zum Konfigurieren der Unterbrechungs- und Ereignssignalisierung. Manche Register sind nicht nur für Lese- und Schreibzugriffe, sondern auch – wie vorstehend beschrieben – für Setz-, Lösch- und Umschaltzugriffe eingerichtet; ihnen sind dann bis zu vier Adressen zugeordnet. Die grundsätzlichen Merkmale stellen eine Art Industriestandard dar, die Einzelheiten sind typspezifisch.

Ausgangsstufen
Einfachausführungen sehen lediglich einen Pull-up-Widerstand vor, der programmseitig zugeschaltet werden kann. Der Widerstandswert liegt zwischen 20 kΩ und 200 kΩ. Damit kann ein offener Eingang auf High-Pegel gehalten werden. Ganz einfache Lösungen brauchen keine zusätzlichen Register. Der Widerstand wird angeschaltet, wenn der Anschluss als Eingang konfiguriert ist und wenn die betreffende Bitposition des Datenregisters eine Eins enthält (Beispiel: AVR). Höher entwickelte E-A-Ports haben zusätzliche Steuerregister, um die Widerstände anzuschalten.

Eingangsstufen
Die Vorkehrungen betreffen die Wahl zwischen verschiedene Signalpegeln und das Abschalten der Eingänge (Stromsparbetrieb).

Zwei Beispiele

Abbildung 4.13 zeigt einen vergleichsweise einfachen E-A-Port, der fünf Register enthält. Alle Register können programmseitig gelesen werden. Die erweiterten Vorkehrungen betreffen das Zuschalten eines Pull-up-Widerstandes und die Wahl der Eingangspegel (TTL, CMOS oder Schmitt-Trigger). Der E-A-Port von Abbildung 4.14 umfasst hingegen 14 Registeradressen (Tabelle 4.2). Jeder Anschluss hat ein eigenes Register zur Konfigurationssteuerung (Tabelle 4.3 und 4.4). Die Abbildungen 4.14 und 4.15 veranschaulichen die programmseitig wählbaren Konfigurationen der Ausgänge.

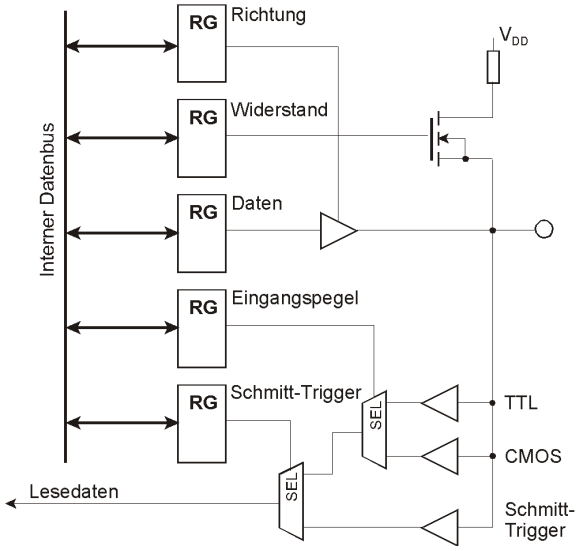


Abb. 4.13 Ein vergleichsweise einfacher E-A-Port.

Unterbrechungs- und Ereignissignalisierung

Die Einfachauslegung besteht darin, dass man einige Anschlüsse zum Auslösen von Unterbrechungen konfigurieren kann (externe Interrupts). Die Unterbrechungen werden typischerweise auch dann wirksam, wenn die Anschlüsse als Ausgänge betrieben werden. Hierdurch wird es möglich, Unterbrechungen programmseitig auszulösen (Software-Interrupts). Erweiterungen bestehen vor allem darin, die Anschlüsse auf Änderungen der Signalpegel zu überwachen, um Unterbrechungen auszulösen (Pin-Change Interrupt; Change of State CoS) oder um Ereignisse an andere Einrichtungen auf dem Schaltkreis zu melden. Typische Wahlmöglichkeiten für die Unterbrechungsauslösung bei Pegeländerungen an Anschlüssen sind:

- Gar keine Unterbrechung (Maskierung),
- Unterbrechung bei steigender Flanke,
- Unterbrechung bei fallender Flanke,
- Unterbrechung bei jeder Flanke,
- Unterbrechung bei einem bestimmten Pegel.

Die Ports enthalten zusätzliche Register zur Unterbrechungssteuerung. In den Einzelheiten gibt es teils beträchtliche Unterschiede zwischen den Prozessorfamilien.

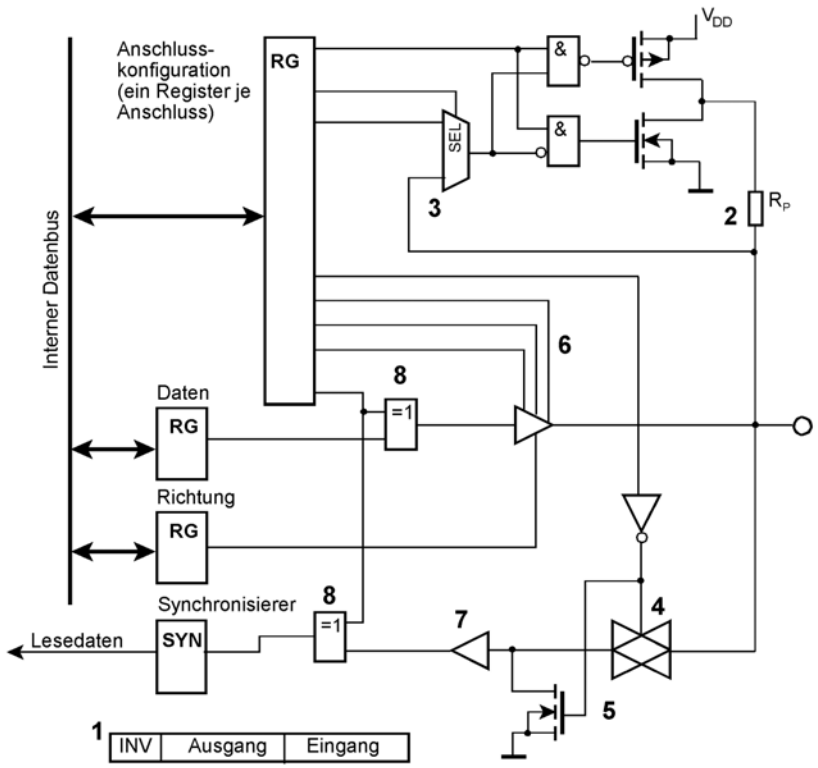


Abb. 4.14 Ein vergleichsweise komplizierter E-A-Port. 1 - ein Register zur Steuerung der Anschlusskonfiguration; 2 - eingebauter Widerstand; 3 - Rückführung zum Halten des Ausgangspegels (Bushalteschaltung); 4 - Übertragungsgatter zum Abschalten des Anschlusses; 5 - dieser Transistor hält den abgeschalteten Eingang auf einem festen Pegel (Low); 6 - Steuerung der Treibertransistoren; 7 - Schmitt-Trigger; 8 - Invertierung (siehe INV in Pos. 1).

Richtungssteuerung
Richtungssteuerung setzen
Richtungssteuerung löschen
Richtungssteuerung umschalten
Daten (Ausgabe)
Datenbits setzen
Datenbits löschen
Datenbits umschalten
Eingabe (Anschlüsse)
Unterbrechungssteuerung
Bitmaske für Unterbrechung 1
Bitmaske für Unterbrechung 2
Unterbrechungsanzeige (Flags)
Anschlusskonfiguration (ein Register je Anschluss)

Tabelle 4.2 Die Register des Ports von Abbildung 4.14.

Ausgang	Treibertransistoren	Widerstand	Abbildung
Gegentakt	Beide aktiv	Inaktiv	4.15a
Gegentakt	Beide aktiv	Bushalteschaltung	4.15b
Gegentakt	Beide aktiv	Nach V_{DD} (Pull-Up)*	4.15c
Gegentakt	Beide aktiv	Nach Masse (Pull-Down)*	4.15d
Wired AND	Nur N-Kanal aktiv (zieht nach Low)	Inaktiv	4.16a
Wired OR	Nur P-Kanal aktiv (zieht nach High)	Inaktiv	4.16b
Wired AND	Nur N-Kanal aktiv (zieht nach Low)	Nach V_{DD} (Pull-Up)	4.16c
Wired OR	Nur P-Kanal aktiv (zieht nach High)	Nach Masse (Pull-Down)	4.16d

*: Diese Betriebsarten sind nur dann von Bedeutung, wenn der Anschluss als Eingang konfiguriert ist.

Tabelle 4.3 Betriebsarten eines Ausgangs.

Unterbrechung bei beiden Flanken
Unrerbrechung bei ansteigender Flanke
Unterbrechung bei abfallender Flanke
Unterbrechung bei Low-Pegel
Eingang ausschalten

Tabelle 4.4 Betriebsarten eines Eingangs.

Wer hat die Unterbrechung ausgelöst?

Üblicherweise hat nicht jeder Anschluss eine eigene Unterbrechungsadresse. Zumeist gibt es nur eine Unterbrechungsadresse je Port. Die genaue Unterbrechungsursache muss programmseitig herausgefunden werden. Es liegt nahe, die Bitmuster an den Anschlüssen zu lesen und mit gespeicherten Werten zu vergleichen, um zu erkennen, was sich geändert hat. Es kann aber vorkommen, dass das auslösende Signal in der Zeit von der Unterbrechungsauslösung bis zur programmseitigen Auswertung (Reaktionszeit) bereits wieder auf den vorhergehenden Pegel zurückfällt. Wenn dies in der jeweiligen Anwendung ein Problem darstellt, muss der E-A-Port mit programmseitig löschbaren Fangschaltungen ausgerüstet werden..

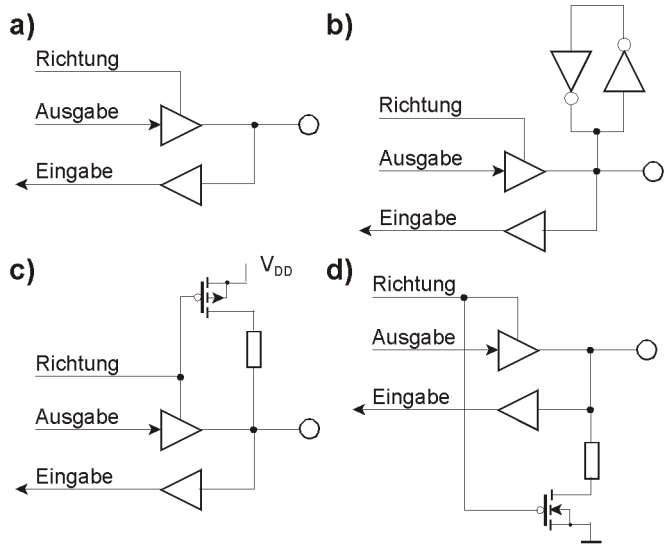


Abb. 4.15 Tri-State-Ausgänge. Erklärung in Tabelle 4.3.

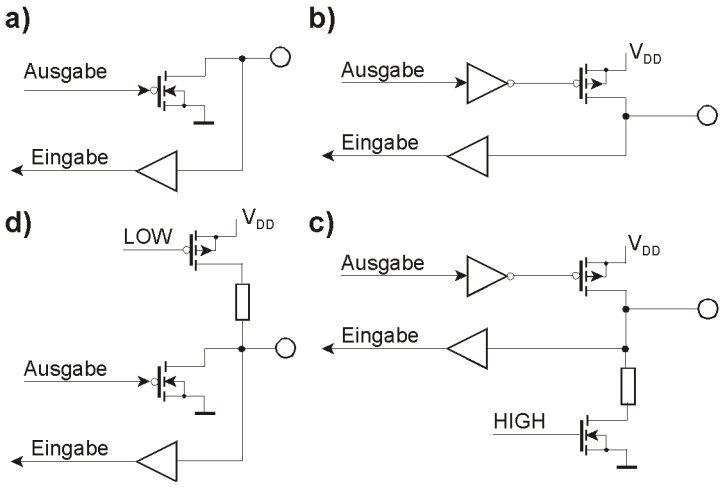


Abb. 4.16 Open-Drain- und Open-Source-Ausgänge (Wired AND und Wired OR). Erklärung in Tabelle 4.3.