Automatisierungstechnik AU 1

– Übersicht –

21. 7. 2008

Der Lehrstoff:

1. Aufgaben und Lösungsansätze in der Automatisierungstechnik
2. Problemlösung mit Mikrocontrollern
3. Mikrocontroller – Aufbau und Wirkungsweise
4. Einführung in die Rechner- und Prozessorarchitektur
5. Programmiermodelle und Programmierphilosophien
6. Problemlösung durch Anwendungsprogrammierung
7. Grundlagen der Maschinenprogrammierung
8. Einführung in die Realzeitprogrammierung

Der Fokus: Mikrocontroller im praktischen Einsatz

Typische Szenarien der Anwendung:

- die Hardware ist gegeben,
- die Entwicklungsumgebung ist gegeben,
- die Ressourcen sind knapp,
- die Problemlösung steht unter Zeitdruck,
- man kann nicht alles haben,
- es läuft keineswegs alles ideal – man muß sich halt zu helfen wissen ...

Anmerkung:
Im Gegensatz dazu betrifft das Fach Hard- und Software-Engineering die Entwickung des gesamten Komplexes aus Hard- und Software von Grund auf.

Der Mikrocontroller
- ist zum einen ein kleiner Prozeßrechner,
- ist zum anderen eine Alternative zur anwendungsspezifischen Hardware (Kostensenkung – er wird nicht deswegen eingesetzt, weil es etwas zu rechnen gibt, sondern nur um eine bestimmte Funktion billiger zu erfüllen als dies mit einer zweckgebundenen Schaltung möglich wäre).

Vermischte Stichworte

Der Mikrocontroller und die Außenwelt:

- E-A-Ports
- Eingebaute Peripherie
- Externe Erweiterung

Problemlösung durch Anwendungsprogrammierung:
Wir haben nur die blanke Hardware und einen Compiler und/oder Assembler.

Anwendungsprogrammierung in C:

- Besonderheiten (z. B. gegenüber der PC-Programmierung): C ist hier maschinen- und compilerspezifisch,
- Ansätze und Auswege.

Der C-Compiler ist ein Mittel zur Arbeitserleichterung, gewährleistet aber keine Maschinenunabhängigkeit. Bereits beim Wechsel vom Compiler des Anbieters A zum Compiler des Anbieters B geht es nicht ohne Eingriffe von Hand ab.

Die gängige Praxis:

- C-Programmierung, wenn es um algorithmische Probleme geht,
- Assemblerprogrammierung, wenn die Hardware bis aufs letzte auszunutzen ist (z. B. exakte Kontrolle über das Zeitverhalten),
- die meisten Entwickler bleiben bei einer Entwicklungsumgebung (und damit beim gleichen Compiler). Nachteile (jedes System hat welche), die sich von Zeit zu Zeit unangenehm bemerkbar machen, werden in Kauf genommen.
- die gesamte Anwendung wird an sich in C programmiert; Assemblerprogrammstücke werden bedarfsweise eingefügt (richtig: in bestimmten Funktionen konzentriert; falsch: gleichsam mit dem Salzstreuer über das gesamte Programm verteilt).

Einführung in die Rechner- und Prozessorarchitektur:
AU1 betrifft nicht die Rechnerarchitektur im allgemeinen Sinne. Vielmehr beschränken wir uns

- auf Auslegungen, die sich in der Praxis durchgesetzt haben,
- auf Leistungsklassen, wie sie für Embedded Systems typisch sind (ein Befehl zu einer Zeit, keine Parallelverarbeitung usw.).

Grundlagen der Maschinenprogrammierung:

Weshalb maschinennahe Programmierung?

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

Lernziele:

- Grundlagen des Aufbaus und der Wirkungsweise von Prozessoren und Mikrocontrollern (Einführung in die Rechnerarchitektur),
- Grundlagen der Anwendungsprogrammierung,
- Grundlagen der Maschinenprogrammierung,
- Grundlagen der Programmentwicklung (Ausdenken – Programmieren – zum Laufen bringen).

Übungen:

- die E-A-Ports der Mikrocontroller ausnutzen,
- Anwendungsprogrammierung in C,
- Anwendungsprogrammierung in Assembler (Atmel AVR).

Die E-A-Ports der Mikrocontroller

- sind universell als Ausgänge oder Eingänge nutzbar,
- jede Bitposition ist einzeln umschaltbar,
- industrieübliche Vorzugslösung: Richtungssteuerregister und Datenregister (Abb. 1),
- Ports stehen nach dem Einschalten typischerweise auf Eingang (Pins hochohmig),
- Signale werden oftmals ungeordnet angeschlossen (an jene Pins, die beim Entflechten der Leiterplatte gerade günstig liegen). Prinzip: möglichst niedrige Hardwarekosten, möglichst viel mit Software erledigen. Deshalb haben Einzelbitzugriffe (Setzen, Abfragen) eine besondere Bedeutung. Einzelbitfunktionen werden intern typischerweise auf bitweise logische Verknüpfungen über die ALU zurückgeführt (Abb. 2).

Genau nachsehen:

- was ist programmseitig adressier- und abfragbar?
- was wird hardwareseitig zurückgesetzt?
- wie lange dauert es, bis eine Ausgangsbelegung zurückgelesen wird?
- welche Sonderfunktionen / Zusätze sind vorgesehen (schaltbare Pull-up-Widerstände, Stromsparbetriebsarten, Interruptauslösung bei Pegeländerung, Slave Port usw.)?

Typische Gotchas:

1. Es werden nicht wirklich ALLE Bitpositionen hardwareseitig zurückgesetzt (z. B. nur das Richtungsregister, nicht aber das Datenregister).

   Praxistip: Die gesamte Initialisierung in Software erledigen – auch dann, wenn die Hardware an sich alles richtig einstellt. Anwendung: programmseitiger Wiederanlauf.

2. Erst die Datenbelegung stellen, dann die Bitposition auf Ausgang schalten (sonst erscheint womöglich die falsche Datenbelegung kurzzeitig am Pin – und könnte in der angeschlossenen Hardware Schaden anrichten).

3. Was tun mit ungenutzten I/-O-Pins? – Keinesfalls hängen lassen, sondern:

   - auf Ausgang schalten (Belegung mit Festwert (Parken)),
   - als Eingang lassen und hardwareseitig mit Festwert belegen,
   - als Eingang lassen und – falls vorhanden – eingebaute Pull-up-Widerstände scharfmachen.

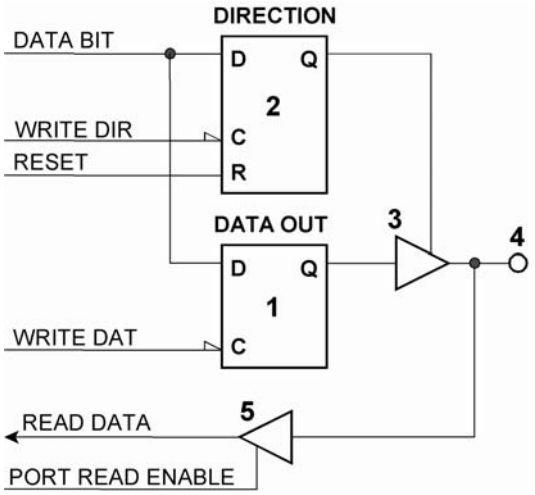
4. Jeweils die richtige Zugriffsadresse verwendet? (Z. B. Atmel AVR: Rücklesen von PORTx bringt nur die Registerbelegung., aber keine Neuigkeiten von außen.)

5. Zurücklesen von Ausgängen: die gerade ausgegebene Belegung ist nie im nächsten Takt verfügbar. Es vergeht wenigsten ein Takt, bevor sie wieder synchronisiert ist (Einzelheiten des I/O Timing stehen im jeweiligen Handbuch). Bei schnellem Takt (z. B. 16...20 MHz) und hoher kapazitiver Belastung macht sich zudem die Anstiegszeit bemerkbar.

6. Bei typischen Sparlösungen (z. B. Zurücklesen stets von den Pins) ist mit Verfälschung von Datenregisterinhalten zu rechnen. Nachsehen, ob dies stört. Wann es u. a. stören könnte:

   - wenn während des Betriebs die Richtung umgesteuert wird (z. B. Nachbildung des Open-Collector-Verhaltens mit einer Tri-State-Schnittstelle),
   - wenn verschiedene Bits unabhängigen Tasks zugeordnet sind (und jede Task sich darauf verläßt, daß "ihre" Registerinhalte erhalten bleiben).

   Abhilfe (notfalls): auf die Portregister nur mit ganzen Bytes zugreifen. Kopien der Registerinhalte im Speicher halten und ggf. dort ändern. Entsprechende Funktionen oder Macros vorsehen. Nachteil: Laufzeit.



1 - Datenregister; 2 - Richtungssteuerregister; 3 - Ausgangstreiber (Tri State); 4 - E-A-Anschluß; 5 - Lesesignaltreiber.

Abb. 1 E-A-Port mit Tri-State-Ausgang (eine Bitposition)

Abb. 1 veranschaulicht den grundsätzlichen Aufbau eines Tri-State-Ports, wie er in vielen modernen Mikrocontrollern zu finden ist. Jede Bitposition kann einzeln als Eingang oder als Ausgang konfiguriert werden. Hierzu ist das Richtungssteuerregister entsprechend zu laden.

- Ausgabe: Richtungssteuerregister 2 mit Eins laden. Inhalt des Datenregisters 2 erscheint am Anschluß 4.
- Lesen: Gelesen wird durch Aktivieren des Lesesignaltreibers 5. Es wird stets die Signalbelegung am Anschluß 4 gelesen. Je nach Inhalt des Richtungssteuerregisters 2 handelt es sich um den Inhalt des Datenregisters 1 (Ausgabe) oder um ein außen anliegendes Signal (Eingabe).
- Eingabe: Richtungssteuerregister 2 mit Null laden. Ausgangstreiber 3 wird hochohmig. Somit darf der Anschluß 4 von außen belegt werden.

- Der Anfangszustand (nach dem Rücksetzen): alles auf Eingabe (Anschlüsse hochohmig).
- Einstellen: erst Datenbelegung, dann Richtungssteuerung (nur so erscheinen von Anfang an korrekte Ausgangsbelegungen und nicht solche, die sich womöglich aus einer zufälligen Belegung des Datenregisters ergeben).

Rücksetzen und Initialisierung
Nach dem Einschalten müssen die Anschlüsse zunächst hochohmig sein, um Buskonflikte zu vermeiden. Zu den ersten Aufgaben der Software gehört die Initialisierung der E-A-Ports (Belegen mit Anfangswerten, Einstellen der Übertragungsrichtung). Die Grundfrage: was wird wirklich hardwareseitig zurückgesetzt?

- die Optimallösung: alle Register, und zwar auf einen Wert, der nicht schadet (z. B. Richtung auf Eingang, Daten auf Null),
- eine gelegentlich zu findende Sparlösung: nur das Richtungsregister (Beispiel. PIC16C5x). Wichtig: erst das Datenregister mit der Anfangsbelegung laden, dann die Übertragungsrichtung einstellen (sonst stellen sich an den Anschlüssen kurzzeitig die anfänglichen Zufallsbelegungen des Datenregisters ein – was sehr häßliche Nebenwirkungen haben kann...).

Praxistip:
Stets eine Software-Routine zum Rücksetzen vorsehen (die alle Ports, Steuerregister usw. einstellt) – auch dann, wenn in der Hardware an sich alles richtig zurückgesetzt wird. Anwendung: zum programmseitig ausgelösten Rücksetzen zwecks Wiederanlauf (z. B. als Reaktion auf Fehlermeldungen und fehlerhafte Interrupts).

Ports zurücklesen
Manchmal ist es erforderlich, die Belegung von Ausgängen wieder einzulesen. Typische Anwendungen:

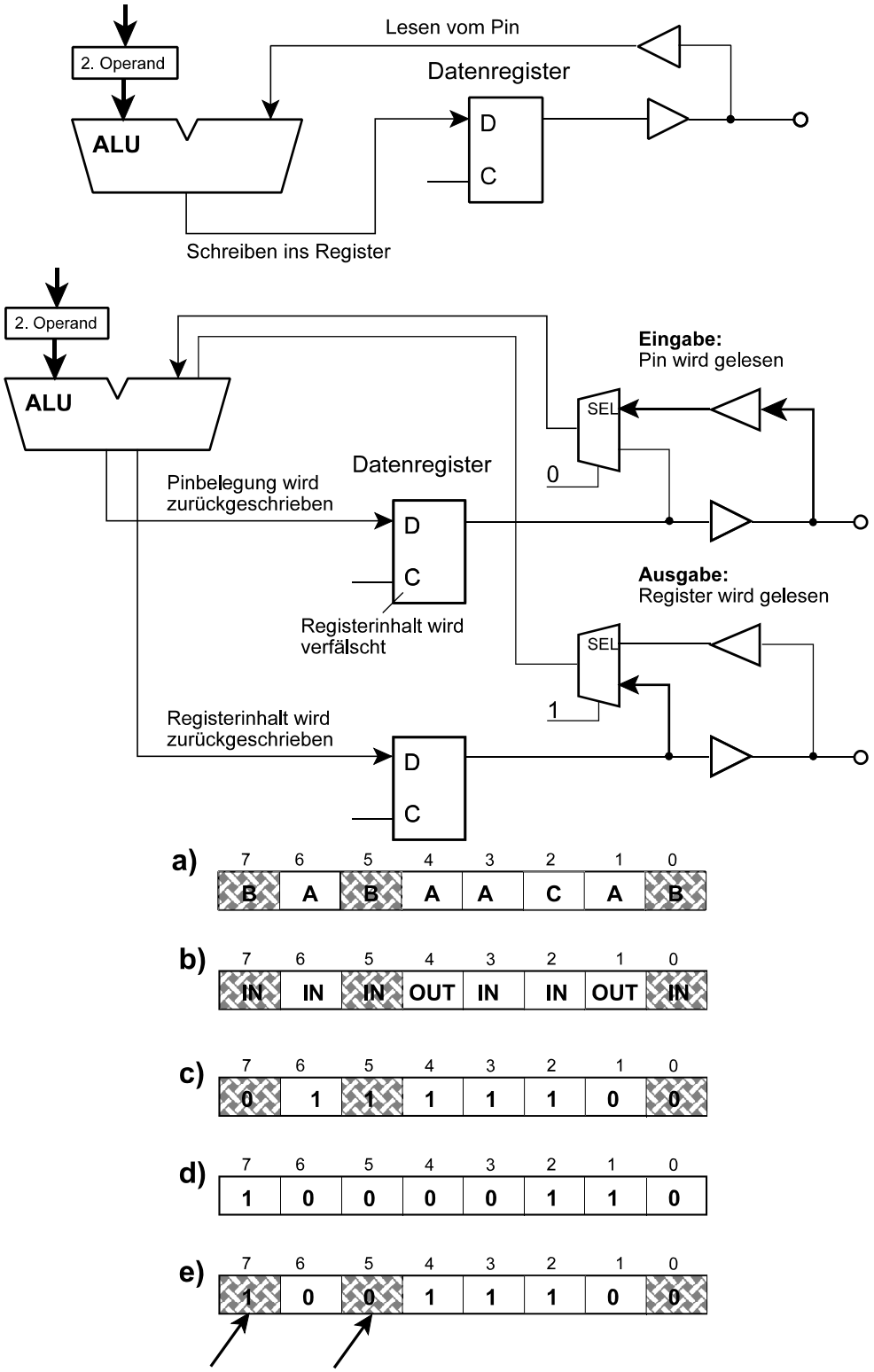
- zum Modifizieren von Bitpositionen und Feldern (Read - Modify - Write),
- zu Testzwecken (nachsehen, ob die Ausgänge tatsächlich richtig schalten).

Wie werden Einzelbits und Bitfelder beeinflußt?
Der übliche Weg führt vom Port in die Arithmetik-Logik-Einheit (ALU) des Prozessors und von dort zurück zum Port (Abb. 2). Die Bitbeeinflussung in der ALU beruht auf elementaren bitweisen Verknüpfungen:

- Setzen von Bits: ODER mit Einsen an den zu setzenden Positionen (sonst Nullen),
- Löschen von Bits: UND mit Nullen an den zu löschenden Positionen (sonst Einsen),
- wechseln (Invertieren) von Bits: XOR mit Einsen an den zu invertierenden Positionen (sonst Nullen),
- Eintragen eines Wertes in ein Bitfeld: erst Löschen des Feldes (UND mit Nullen in den Feldpositionen), dann Setzen der Einsen (ODER mit dem neuen Feldinhalt).

Auch Controller und Prozessoren mit mehr oder weniger komfortablen Bitbefehlen erledigen das so. Probleme bei Portzugriffen können deshalb zu Datenverfälschungen und somit zu Fehlern führen, die nur zeitweilig auftreten (nämlich dann, wenn tatsächlich abweichende Bitbelegungen entstehen) und nur schwer zu finden sind (datenabhängige Fehler).

Wann ist ungestörtes bitweises Modifizieren besonders wichtig? – Vor allem beim echten Multitasking, 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.



a) Zuordung der Pins 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 nach dem Einlesen ins Datenregister eingeschrieben wird, werden auf Eingabe geschaltete Bitpositionen verfälscht (Pfeile). Wenn sich nun aber Task B darauf verläßt, daß die ursprünglich eingeschriebenen Werte (c) auch weiterhin drinstehen...

Abb. 2 Bitmodifkation über die Arithmetik-Logik-Einheit (ALU)

Rücklesen der Portbelegung (1). Die Zeitfrage
Ausgegebene Signalbelegungen sind typischerweise nicht sofort zurücklesbar (lesen wir zu zeitig zurück, so lesen wir die alte Belegung). Hierfür gibt es zwei Ursachen:

- die kapazitive Belastung des Anschlusses (Schaltungsabhängig),
- die zur Synchronisation nötige Eintaktierungszeit (fest).

Beispiel:

Programmierabsicht:

SET Bit 4 (= OR 10H)
SET Bit 5 (= OR 20H)

Es soll erst Bit 4 eingeschaltet werden und einen Befehl später Bit 5. Der Befehl SET Bit 5 trifft aber auf die Ausgangsbelegung *vor* SET Bit 4, so daß Bit 4 als Null gelesen und beim Modifizieren wieder gelöscht wird (Abb. 3).

Abb. 3 Theorie und Praxis. a) Programmierabsicht; b) der tatsächliche Signalverlauf an den Anschlüssen

Die Anzahl der Takte, die zwischen Ausgabe und Rücklesen mindestens durchlaufen werden müssen, steht im Datenblatt. Wie schnell sich eine Änderung der Belegung des Daten-Latch am Anschluß bemerkbar macht (und somit zurückgelesen werden kann), hängt von der kapazitiven Belastung des Anschlusses ab. Bei hoher kapazitiver Belastung kann die zwischen Ausgabe und Rücklesen abzuwartende Zeit länger sein als eine Taktperiode (vor allem bei hohen Taktfrequenzen).

Abhilfe 1
Zwischen Ausgabe und Rücklesen Wartebefehle (einfachste Lösung: NOPs einfügen).

Praxistips:

1. Das Schaltverhalten an den Anschlüssen beobachten (Oszilloskop) – und lieber einen NOP zuviel einfügen als einen zuwenig.
2. Programmieren des Intervalls zwischen Schreiben und Lesen: nicht einfach NOPs hineinhacken, sondern einen Makro definieren (der nach Bedarf angepaßt werden kann – bei einem langsamen Prozessor wird er z. B. als ein einziger NOP implementiert, bei einem schnellen Prozessor hingegen als Warteschleife).

Abhilfe 2
Solche Sequenzen stets durch Schreiben ganzer Bytes programmieren[1)]:

statt
SET Bit 4
SET Bit 5

also z. B.
OUT 10H
OUT 11H

Rücklesen der Portbelegung (2). Inhaltsverfälschungen
Wir wollen den Datenregisterinhalt modifizieren, können aber nur die Portbelegung an den Anschlüssen zurücklesen. Wir nehmen an, daß nicht vorzeitig zurückgelesen wird. Trotzdem ist achtzugeben, und zwar abhängig von der eingestellten Übertragungsrichtung:

- wenn Ausgang, so ist es o. k. (Portbelegung = Registerinhalt),
- wenn Eingang, wird die Portbelegung gelesen. Sie wird so zum neuen Registerinhalt. Wird jetzt der Port zum Ausgang, so wird nicht ein ggf. zuvor geladener Registerinhalt wirksam, sondern die zuletzt gelesene Portbelegung.

Ist das tatsächlich ein Problem? – Es kommt drauf an:

- nein, wenn Eingänge für immer Eingänge bleiben,
- ja, wenn die Übertragungrichtung immer wieder umgestellt wird und wenn wir uns auf die im Datenregister befindliche Belegung verlassen. Beispiel: die Nachbildung des Open-Collector-Verhaltens an einem Tri-State-Port. Das Datenregisterbit ist stets Null, und das Richtungssteuerregisterbit bestimmt die Anschlußbelegung. Bit = High: Eingangsrichtung (Port hochohmig; Anschluß wird über externen Pull-up-Widerstand auf High gezogen). Bit = Low: Ausgangsrichtung; Anschluß wird mit der Null aus dem Datenregister belegt. Der naive Programmierer setzt das Datenregisterbit 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 Lesevorgang statt. Dann wird die besagte Null im Datenregister von außen mit einer Eins überschrieben werden. Infolgedessen wird unsere Open-Collector-Nachbildung nie mehr einen Low-Pegel ausgeben können...

Was wird bei Lesezugriffen wirklich gelesen?
Es gibt verschiedene Auslegungen:

- nur die Portbelegung (Beispiel: PIC 16C5x). Richtungsregister kann nicht modifiziert werde. Deshalb besondere Ladebefehle (PIC: TRIS).
- Richtungsregister und Portbelegung (Beispiel: diverse PIC-Typen). Erlaubt Modifizieren des Richtungssteuerregisters.
- Steuerung über Richtungssteuerregister: wenn Eingang, dann Portbelegung, wenn Ausgang, dann Register (Beispiel: Renesas H8/300H; vgl. Abb. 2.5). Beseitigt das Zeitproblem (unmittelbar zuvor geänderte Ausgangsbelegungen werden aus dem Register geholt, also nicht der Zeitverzögerung am Port unterworfen). Die Inhaltsverfälschungen (Überladen von Datenregisterpositionen mit Eingangsbelegungen) bleiben aber.
- Richtungssteuerregister, Datenregister und Portbelegung (Beispiel: Atmel AVR).

---

1) dazu müssen wir allerdings wissen, wie die anderen Bits belegt sind. Beim Multitasking funktioniert das nicht.

Nicht rücklesbare Register können nur insgesamt überschrieben, nicht aber bitweise in ihrem Inhalt geändert werden.

Abhilfe bei fehlender Rücklesbarkeit:
Software-Kopien der Registerinhalte verwalten. Zugriffe über Makros oder Unterprogramme.

Eine Auslegung, die diese Probleme vermeidet
Jeder Port muß drei Lesezugriffe unterstützen (vgl. Atmel AVR): vom Richtungssteuerregister, vom Datenregister und von den Anschlüssen (Abb. 4). Zugriffe:

- zum Modifizieren der Signalflußrichtung: auf das Richtungssteuerregister,
- zum Modifizieren der Ausgangsbelegung: auf das Datenregister,
- zur Eingabe: auf die Anschlüsse.

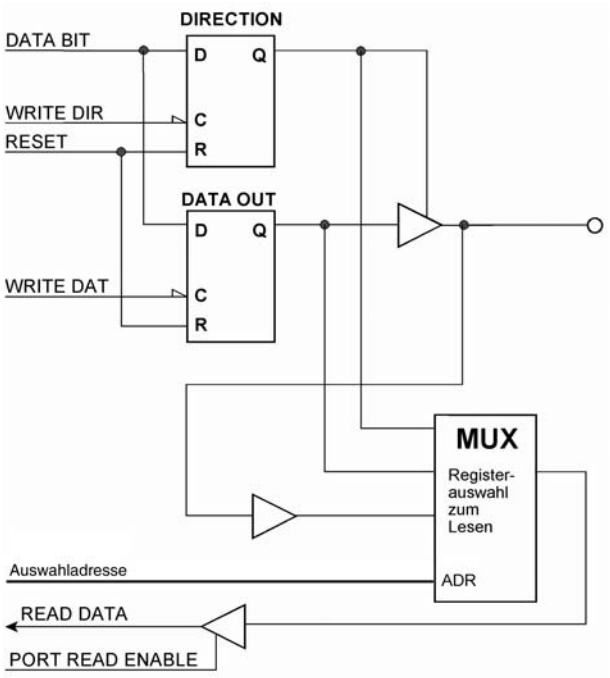


Abb. 4 Dieser E-A-Port unterstützt drei Lesezugriffe

Einführung in die Prozessorarchitektur und Maschinenprogrammierung

Lernziele:

- Grundlagen des Aufbaus und der Wirkungsweise von Prozessoren und Mikrocontrollern (Einführung in die Rechnerarchitektur),
- Grundlagen der Maschinenprogrammierung,
- Grundlagen der Programmentwicklung (Ausdenken – Programmieren – zum Laufen bringen).

Weshalb maschinennahe Programmierung?

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

Grundlagen der Rechnerarchitektur und maschinennahen Programmierung kennenlernen
Beide Lehrgebiete sind eng miteinander verbunden. Wer beim Programmieren bis auf die blanke Hardware durchgreifen will, muß deren Wirkprinzipien kennen. Die Rechnerarchitektur ist keine exakte Wissenschaft. Es ist nach wie vor üblich, in der Lehre mit dem grundsätzlichen Aufbau des Universalrechners zu beginnen und dann die einzelnen Funktionseinheiten zu diskutieren. Dabei bezieht man sich üblicherweise auf konkrete Beispiele – Rechnerarchitekturen werden zumeist ähnlich beschrieben wie Tier- oder Pflanzenarten in der Biologie. Wir verpassen also nicht viel, wenn wir uns tiefgründige Theorien schenken und sofort beginnen, uns in eine bestimmte Architektur einzuarbeiten.

AU1 betrifft nicht die Rechnerarchitektur im allgemeinen Sinne. Vielmehr beschränken wir uns

- auf Auslegungen, die sich in der Praxis durchgesetzt haben,
- auf Leistungsklassen, wie sie für Embedded Systems typisch sind (ein Befehl zu einer Zeit, keine Parallelverarbeitung usw.).

Allgemeine Merkmale einer Rechnerarchitektur:

- Programmiermodell,
- Datenstrukturen,
- Befehlswirkungen,
- Befehlsformate,
- Registermodell,
- Speicheradressierung,
- Speicherverwaltung,
- Ein- und Ausgabe,
- Unterbrechungssystem,
- Schutzvorkehrungen,
- Maschinenzeit,
- Kaltstart.

Es gibt so viele Controller und Prozessoren. Wie können wir uns da zurechtfinden?
Alles halb so schlimm – elementare Datenstrukturen und Befehlswirkungen sind im Grunde gleich. Die Unterschiede betreffen vor allem:

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

Typische Registermodelle:

- Akkumulatormaschine,
- Universalregistermaschine,
- Stackmaschine.

Typische Auslegungen der Maschinenbefehle
Üblich ist eine pauschale Einteilung in "einfache" und "komplexe" Befehle, die durch zwei Marketingbegriffe ausgedrückt wird:

- RISC = Reduced Instruction Set Computer. Vergleichsweise einfache Befehlswirkungen. Wenige Befehlsformate. Wichtig: Transport- und Verarbeitungsfunktionen sind voneinander getrennt (Load/Store-Prinzip). RISC-Maschinen sind typischerweise Universalregistermaschinen.
- CISC = Complex Instruction Set Computer. Vergleichsweise komplexe Befehlswirkungen. Wichtig: Transport- und Verarbeitungsfunktionen kommen in manchen Befehlen zusammen vor. Viele Befehlsformate, umfangreicher Befehlsvorrat.

Typische Beispiele:

1. Atmel AVR. Eine RISC-Architektur mit 32 Universalregistern und 8 Bits Verarbeitungsbreite.
2. Zilog Z80 und Intel 8086. Typische CISC-Architekturen.

Um zunächst die typischen Befehlswirkungen kennenzulernen, beginnen wir mit dem Atmel (Überschaubarkeit).

Themenübersicht Programmierung/Programmentwicklung

- Was leistet ein Assembler?
- Datenstrukturen
- Informationstransporte
- Ein- und Ausgabe
- arithmetische Verknüpfungen
- kombinatorische Verknüpfungen
- Verschieben und Rotieren
- Einzelbit- und Bitfeldoperationen
- Boolesche Funktionen
- Testen und Vergleichen
- Speicheradressierung und Adreßrechnung
- Verzweigen
- Schleifen
- Unterprogrammruf
- Kontrollstrukturen
- mit Sinn und Verstand programmieren
- Entwicklungsgänge
- Programme zum Laufen bringen (Debugging)

Datenstrukturen:
- Bits, Bytes, Worte
- Binärvektoren
- fest formatierte Zahlendarstellungen
- Bitketten
- Zeichenketten
- Tabellen
- homogene Strukturen (Arrays)
- heterogene Strukturen (Records)
- Stacks
- Konstanten und Variable
- deskriptive Angaben

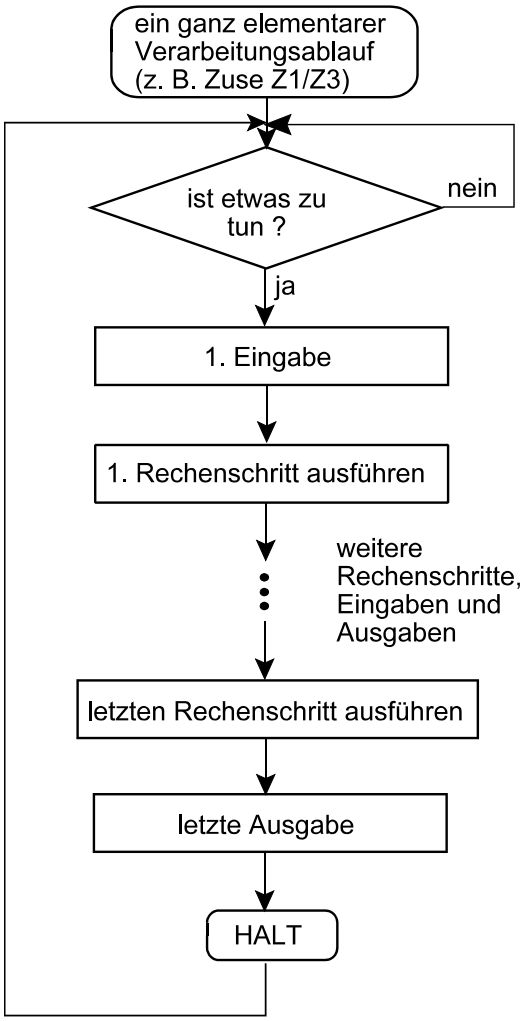
> Zum Philosophieren sind die zwei ersten Erfordernisse diese: erstlich, daß man den Mut habe, keine Frage auf dem Herzen zu behalten, und zweitens, daß man alles das, *was sich von selbst versteht*, sich zum deutlichen Bewußtsein bringe, um es als Problem aufzufassen.
>
> *Arthur Schopenhauer*

Rechnerarchitektur – eine Kurzeinführung

Der Computer ist eine programmgesteuerte Universalrechenmaschine.

- Rechenmaschine: Hauptsache ist das numerische Rechnen, wenigstens in den Grundrechenarten.
- universell: es sollen sich alle überhaupt denkbaren Rechenvorgänge ausführen lassen. Die praktischen Beschränkungen liegen nicht im Grundsätzlichen, sondern in Verarbeitungszeit und Speicherbedarf.
- programmgesteuert: es soll alles automatisch ablaufen.

Die einfachste Art der Programmsteuerung besteht darin, starre Folgen von Eingaben, Rechenschritten und Ausgaben auszuführen. Sofern ein hinreichender Vorrat an Rechenoperationen vorgesehen ist, genügt bereits dieses einfache Schema, um viele nichttriviale Anwendungsaufgaben zu lösen (klassische Lochkartentechnik, Zuse Z1/Z3).



Wirkliche Universalität ist dann gegeben, wenn:

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

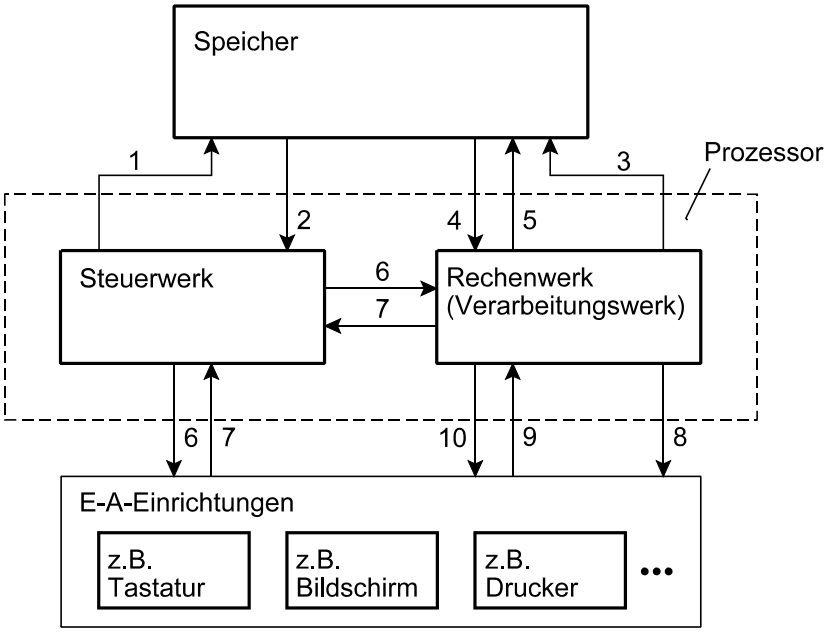
   und

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

Aus den allgemeinen Anforderungen ergibt sich unmittelbar die Struktur der Hardware:

- Rechenwerk: führt die Rechenoperationen aus,
- Steuerwerk: bewirkt, daß die jeweils gewünschten Verarbeitungschritte nqacheinander ausgeführt werden,
- Speicherwerk: speichert die zu verarbeitenden Daten (Operanden) und die Programme.

Das Speicherwerk ist eine lineare Folge von Speicherzellen. Jede Speicherzelle hat eine laufende Nummer – die Adresse. Die Speicherzellen werden über ihre laufenden Nummern (Adressen) angesprochen (Speicheradressierung).

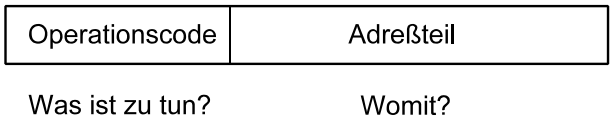


1 - Befehlsadresse; 2 - Lesen der Maschinenbefehle; 3 - Datenadresse; 4 - Lesen von Daten; 5 - Schreiben von Daten; 6 - Steuersignale; 7 - Bedingungs- und Zustandssignale; 8 - E-A-Adressierung; 9 - Eingabe von Daten; 10 - Ausgabe von Daten.

Die Programmsteuerung erfolgt durch Maschinenbefehle. Zu einer Zeit wird jeweils ein Maschinenbefehl ausgeführt. Es gibt verschiedene Arten von Maschinenbefehlen, die durch ihre jeweiligen Wirkungen bzw. Funktionen gekennzeichnet sind:

- es wird eine elementare Operation angewisen, die vom Rechenwerk ausgeführt wird (Operationsbefehl),
- es wird ein elementarer Transportvorgang (Eingabe, Ausgabe, Holen des Inhalts einer Speicherzelle (Lesen), Ablegen von Daten in in eine Speicherzelle (Schreiben)) veranlaßt (Transportbefehl),
- es wird die Reihenfolge der Befehlsausführung beeinflußt (Verzweigungsbefehl),
- es werden allgemeine Steuerwirkungen ausgeübt, z. B. das Einstellen von Betriebsarten (Steuerbefehl).

Der Maschinenbefehl selbst ist eine gespeicherte Angabe, die ein bestimmtes Format aufweist.



In der Auslegungen der Universalrechner hat sich vieles aus technischen Bedingungen heraus ergeben – viele Merkmale sind zu Indutriestandards 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 8, 16, 32 usw. Bits Länge,
- die Auslegung der elemenaren (adresierbaren) Speicherzellen (Byte- oder Wortadressierung),
- der elementare Befehlsvorrat.

Merksätze:

- Die anwendungspraktisch wesentlichen Unterschiede zwischen den einzelnen Architekturen sind viel geringer, als man aufgrund der einschlägigen Werbeaussagen vermuten könnte.

- Oft kommt es gar nicht auf die Architektur an, sondern auf Kosten, E-A- und Speicherausstattung, Taktfrequenz, Gehäusebauformen, Speisespannung und Kompatibilität.

- Alle Architekturen und Befehlssätze sind "powerful" – manche mehr, manche weniger ...

- Es gibt keine ideale Architektur – ja nicht einmal ein wirkliches allgemeingültiges Optimum.

- Man gewöhnt sich an alles.

Typische Befehlswirkungen im Überblick

Die Wirkungen sind überall gleich, nur die Verpackung (Befehlsformate usw.) macht den Unterschied.

Operationsbefehle (1). Numerische Daten			
Binärzahlen		Gleitkomma-zahlen	Dezimalzahlen (BCD)
natürliche	ganze		
Addieren Subtrahieren Vergleichen		Addieren Subtrahieren Multiplizieren	nicht unterstützt *bzw.* Dezimalkorrektur (Hilfsbefehle)
Multiplizieren Dividieren Verschieben	Multiplizieren Dividieren Verschieben (arithmetisch) Vorzeichen-wechsel	Dividieren Vergleichen Betrag Vorzeichen-wechsel Wandeln (Konvertieren) weitere mathematische Funktionen, wie x, sin x usw.	*bzw. volle Unterstützung:* Addieren Subtrahieren Multiplizieren Dividieren Vergleichen Wandeln (Konvertieren)

Operationsbefehle (2). Nichtnumerische 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	Vergleichen	Position der höchstwertigen Eins	Wechseln
Vergleichen (logisch)	Durchsuchen	Anzahl der Einsen	
Verschieben / Rotieren	über Tabelle wandeln		

Transportbefehle	
Laden (Speicher 6 Register)	Umladen (Register 6 Register)
Speichern (Register 6 Speicher)	Umspeichern (Speicher 6 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

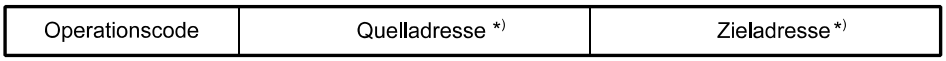
Systembefehle			
Ein- und Ausgabe	Betriebsarten umschalten	Laden/Speichern von Systemregistern	Taskumschaltung
Unterbrechungssteuerung	Steuerung der Speicherverwaltung	Sonderzustände einleiten	Rückkehr aus Supervisorzustand
Hilfsbefehle für Test- und Fehlersuchzwecke			

Der Maschinenbefehl ist eigentlich nur eine fest formatierte Aneinanderreihung von Bits, wobei je nach Befehlstyp bestimmte Bitfelder bestimmte Bedeutungen haben. In den Einzelheiten unterscheiden sich die Befehlsformate der verschiedenen Architekturen bisweilen beträchtlich voneinander.

a) Operationsbefehl: < Ergebnis >: = < 1. Operand > **op** < 2. Operand >

b) Operationsbefehl: < 1. Operand >: = < 1. Operand > **op** Direktwert

c) Transportbefehl: < Zieladresse >: = < Quelladresse >



d) Verzweigungsbefehl (bedingte Verzweigung):

*) : Register- oder Speicheradresse

Register- und Adreßmodelle
Aus der Sicht der Digitaltechnik ist ein Register eine Anordnung gemeinsam zugänglicher Binärspeicher (Flipflops). Solche Register wollen wir Hardware-Register nennen.

Aus der Sicht der Rechnerarchitektur ist ein Register eine programmseitig zugängliche Speichereinrichtung im Prozessor, die üblicherweise soviele Binärstellen umfaßt wie die Verarbeitungsbreite angibt. Solche Register wollen wir als programmseitig zugängliche oder Architektur-Register bezeichnen.

Derartige 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 ausfallen als Speicheradressen (so erfordert ein Registersatz von 32 Register Adreßangaben von nur 5 Bits Länge).

Die Auslegung des Registersatzes hängt wesentlich vom Programmiermodell ab, das der betreffenden Architektur zugrunde liegt. Anzahl und Länge der Register werden maßgeblich von Leistungszielen und Erfahrungswerten bestimmt. Im Laufe der Entwicklung des Universalrechners sind vielfältige Registeranordnungen vorgeschlagen und ausgeführt worden. Dem aktuellen Stand er technik zufolge haben sich folgende Auslegungen durchegsetzt:

Einadreßmaschine:
<Akkumulator> := <Akkumulator> **OP** <Operand>. Benötigt implizites Register (Akkumulator) und Transportbefehle (8051, PIC).

Mehrregister-Einadreßmaschine (1½Adreßmaschine):
<Register> := <Register> **OP** <Operand>. Oft verwendet (IBM 360, Intel usw.). Klassenunterschiede in der Größe und Universalität der Registersätze.

- Nur ein Speicheroperand im Befehl erleichtert Implementierung eines virtuellen Speichers. -

Mehrregister-Dreiadreßmaschine (Load-Store):
<Register 3> ;= <Register 1> **OP** <Register 2>. Universelle Lösung (RISC, z. B. ARM, MIPS, SPARC PowerPC, Intel Itanium).

Mehrregister-Zweiadreßmaschine (Load-Store):
<Register 1> ;= <Register 1> **OP** <Register 2>. Gibt gelegentlich kompakteren Code (AVR, ARM THUMB, Altera NIOS).

Zwei oder drei Adressen? – Man muß wissen, was man will. Es gibt zwei Ansätze oder Philosophien der Nutzung universeller Registersätze:

1. Die Universalregister dienen als Akkumulatoren oder Indexregister. Das Zweiadreßprinzip ist zweckmäßiger (weniger Bits in den Befehlen, einfachere Hardware (nur zwei Zugriffswege)).

2. Die Universalregister halten die lokalen Variablen der aktuellen Funktion. Dann 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. Bei überwiegender Nutzung als Akkumulator oder Indexregister bedeutet die Dreiadreßauslegung Platzverschwendung (unnötig lange Befehle). Bei überwiegender Nutzung zur Variablenspeicherung mit vielen Verknüpfungen der Art C := A **op** B bedeutet Zweiadreßauslegung sowohl Leistungsverlust (zwei Befehle) als auch Platzverschwendung (zwei Operationscodes; eine der Registeradressen ist zweimal anzugeben).

   Im Bereich der kleineren Embedded Systems ist es gelegentlich möglich, *sämtliche* Variable der Anwendung in Registern zu halten. Das entspricht im Grunde dem o. g. zweiten Ansatz.

Wieviele Universalregister?
Einige der Universalregister sind typischerweise für allgemeine Zwecke beiseite gesetzt (Stackpointer, Frame Pointer, Befehlszähler, Festwert 0 usw.).

- 8 Universalregister: sind praktisch nur als Akkumulatoren oder Indexregister zu verwenden. Alles andere ist blanke Theorie (vgl. beispielsweise die Optimierungsempfehlungen von Intel).
- 16 Universalregister: als Variablenspeicher wird es etwas knapp.
- 32 Universalregister: als Variablenspeicher nutzbar (vgl. die Registersätze der typischen RISC-Maschinen). Aber Achtung: die vielen Register wollen bei Unterbrechungen und Taskumschaltung auch gerettet sein.

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

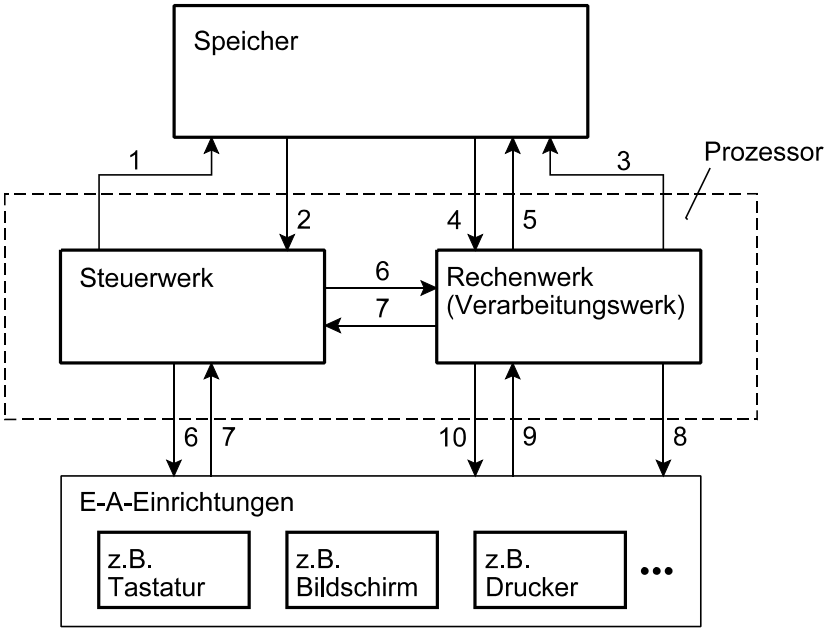
Stackmaschine:
<TOS> := <TOS> **OP** <TOS+1>. Aus den obersten Einträgen im Stack wird das Resultat gebildet. Die Operanden-Einträge werden aus dem Stack entfernt. Das Resultat ist der neue oberste Stack-Eintrag. Sehr kompakter Code. Eine (fiktive) Stackmaschine ist typischerweise das erste Umsetzungsziel eines Compilers (P-Code-Maschine (Pascal), JVM (Java)).

Register und Speicher

Register	Arbeitsspeicher
• Speicherkapazität sehr beschränkt (8 bis 32 sind üblich; extreme Auslegungen haben z. B. 128 (Itanium) oder 196 (Sparc). So große Registersätze werden aber auf andere Weise genutzt (Register-Fenster). • kurze Zugriffszeiten (1 Maschinentakt). Der ideale Universalregisterspeicher erlaubt in einem Maschinentakt gleichzeitig drei Zugriffe auf verschiedene Adressen (triple port memory). • überwiegend nur direkte Adressierung. Alle Register müssen einzen angesprochen werden. Vorkehrungen zur Adreßrechnung sind selten (z. B. PIC und AVR). • Speicherkapazität kann nicht ohne weiteres beliebig erhöht werden – viel hilft hier nicht immer viel. Erfahrungswert: Verdoppelung der Speicherkapazität verlängert Zugriffszeit um 30 %.	• große bis extreme Speicherkapazität (kBytes ... GBytes), • vergleichsweise lange Zugriffszeiten (einige hundert ns), • Adreßrechnung, • Zugriffsbeschleunigung durch hardwareseitige Maßnahmen: Caches, • Kapazitätserhöhung durch softwareseitige Maßnahmen: virtueller Speicher. • Speicherkapazität kann beliebig erhöht werden. Ausgleich der Geschwindkeitseinbußen durch Caches.

Der Einzelprozessor

Im klassischen Sinne besteht ein Einzelprozessor (Abb. 5 bis 8) aus der Steuereinheit (Steuerwerk, Control Unit), der zentralen Verarbeitungseinheit (Operationswerk, Central Processing Unit CPU) und der Speicheranschaltung (Memory Port, Storage Adapter, Storage Control Unit SCU). Die Steuereinheit enthält den Befehlszähler, das Befehlsregister sowie alle Schaltmittel zur Ablaufsteuerung. Die CPU enthält die Verknüpfungsschaltungen, die notwendig sind, um die in den Operationsbefehlen angegebenen Operationen auszuführen, die Adressierungsschaltungen für Datenzugriffe sowie die notwendigen Register. Beide Funktionseinheiten liefern Adressen, um Lese- und Schreibzugriffe auszuführen. Die Steuereinheit holt Befehle und Befehlsadressen aus dem Speicher (z. B. bei Rückkehr aus einem Unterprogramm oder beim Einleiten einer Unterbrechungsbehandlung) und schreibt Befehlsadressen zurück (Adreßrettung). Die CPU holt Operanden bzw. Operandenadressen und schreibt Ergebnisse zurück. Der Universalregistersatz wird von Steuereinheit und CPU gemeinsam genutzt. Die Speicheranschaltung verbindet Steuereinheit und CPU mit dem Arbeitsspeicher. Moderne Prozessoren sind um zusätzliche Funktionsblöcke erweitert, beispielsweise um eine Gleitkomma-Verarbeitungseinheit (Floating Point Processing Unit FPU), eine Befehlsvorbereitungseinheit, eine Segmentierungseinheit, eine Seitenverwaltungseinheit, Caches usw.



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.

Abb. 5 Der Einzelprozessor im Blockschaltbild

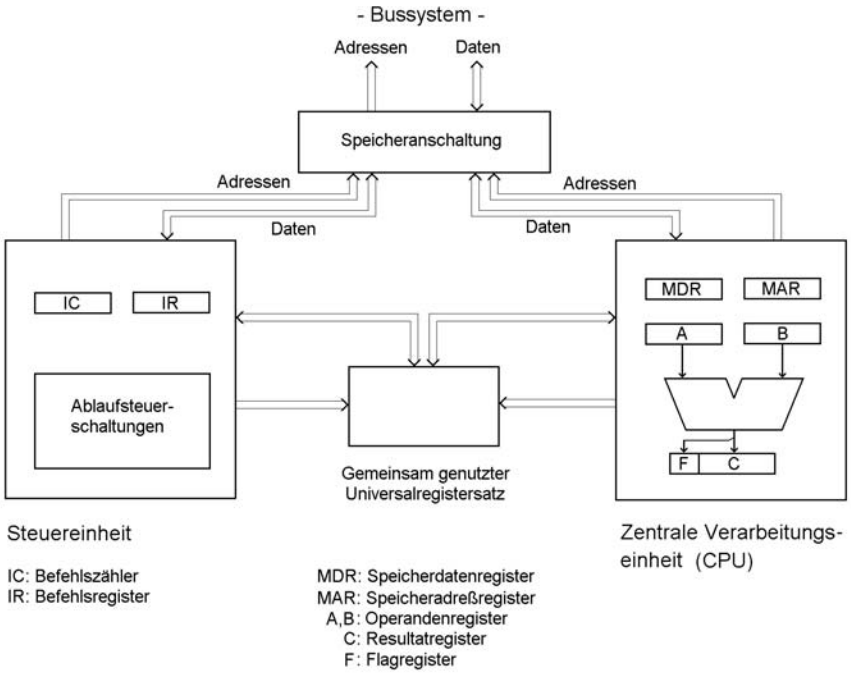
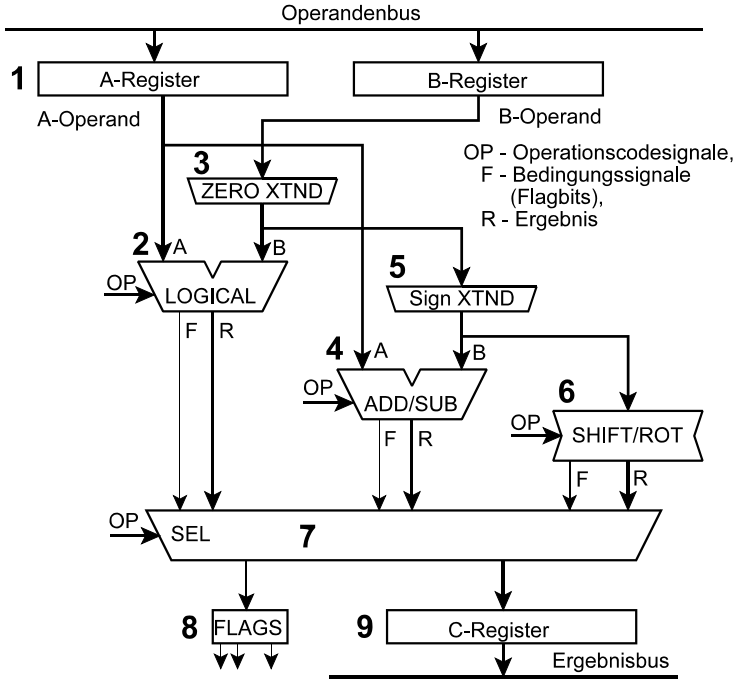


Abb. 6 Dieses Blockschaltbild zeigt weitere Einzelheiten



1 - Operandenregister; 2 - bitweise Verknüpfungen (AND, OR, XOR, CMP usw.); 3 - Nullerweiterung kürzerer Operanden; 4 - binäre Addition/Subtraktion (Zweierkomplement-Arithmetik; Näheres s. Abb. 4); 5 - Vorzeichenerweiterung kürzerer Operanden; 6 - Verschiebe- und Rotationsnetzwerk; 7- Ergebnisauswahl gemäß Ooperationscode; 8 Bedingungssignale; 9 - Ergebnisregister

Abb. 7 Eine typische Arithmetik-Logik-Einheit. Alle gängigen Prozessoren beruhen auf den hier dargestellten elementaren Operationen (bitweise Verknüpfungen, Addition/Subtraktion von Binärzahlen in Zweierkomplemetarithmetik, Verschieben/Rotieren)

v. Neumann-Architektur und Harvard-Architektur

Die Begriffe bezeichnen allgemeine Architekturkonzepte, die sich darin unterscheiden, wieviele Speicheradreßräume bzw. Speicherzugriffswege grundsätzlich vorgesehen sind (Abb. 9).

Gemeinsamkeiten:

Beide Architekturen haben einige wesentliche Prinzipen gemeinsam:

- es gibt einen einzigen Befehlsstrom, wobei Befehl für Befehl nacheinander ausgeführt wird,
- die Befehlsadressierung erfolgt vorzugsweise durch Weiterzählen der Befehlsadresse (Ausnahmen davon sind Verzweigungen, Unterprogrammrufe, Unterbrechungen usw.).

Architekturen, die diese Merkmale nicht aufweisen, sind – beim aktuellen Stand der Technik – eher in den Bereich der akademischen Forschung einzuordnen (man spricht hier von rekursiven Architekturen, funktionalen Architekturen, Datenflußarchitekturen usw.).

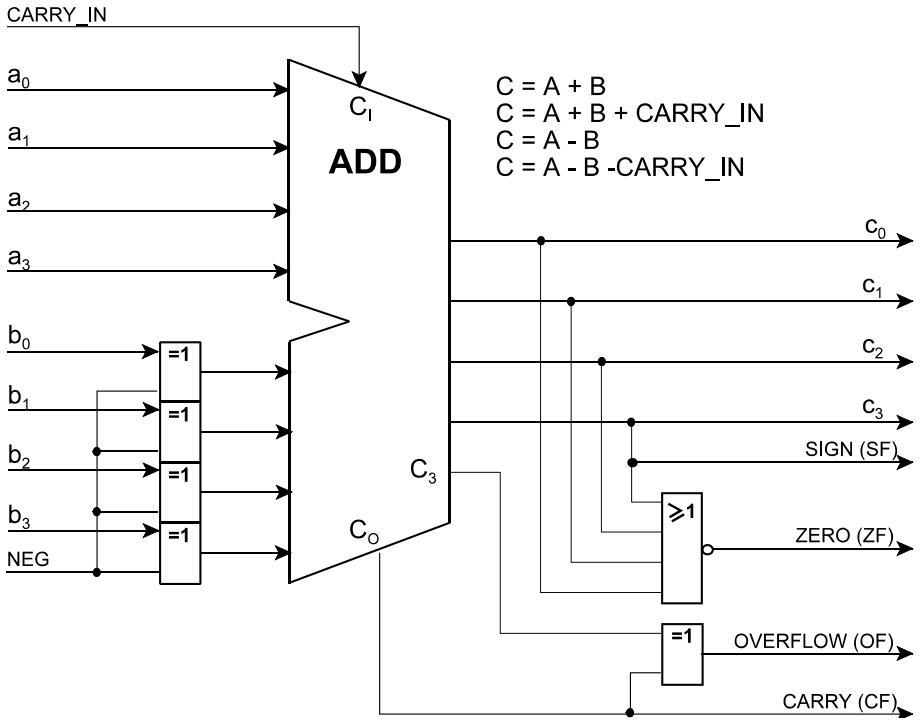


Abb. 8 Die Grundlage des eigentliche Rechnens in der Maschine – das Zweierkomplement-Addierwerk. Es ist Aufgabe der Rechnerarchtektur, diesen Apparat mit Arbeit zu versorgen ...

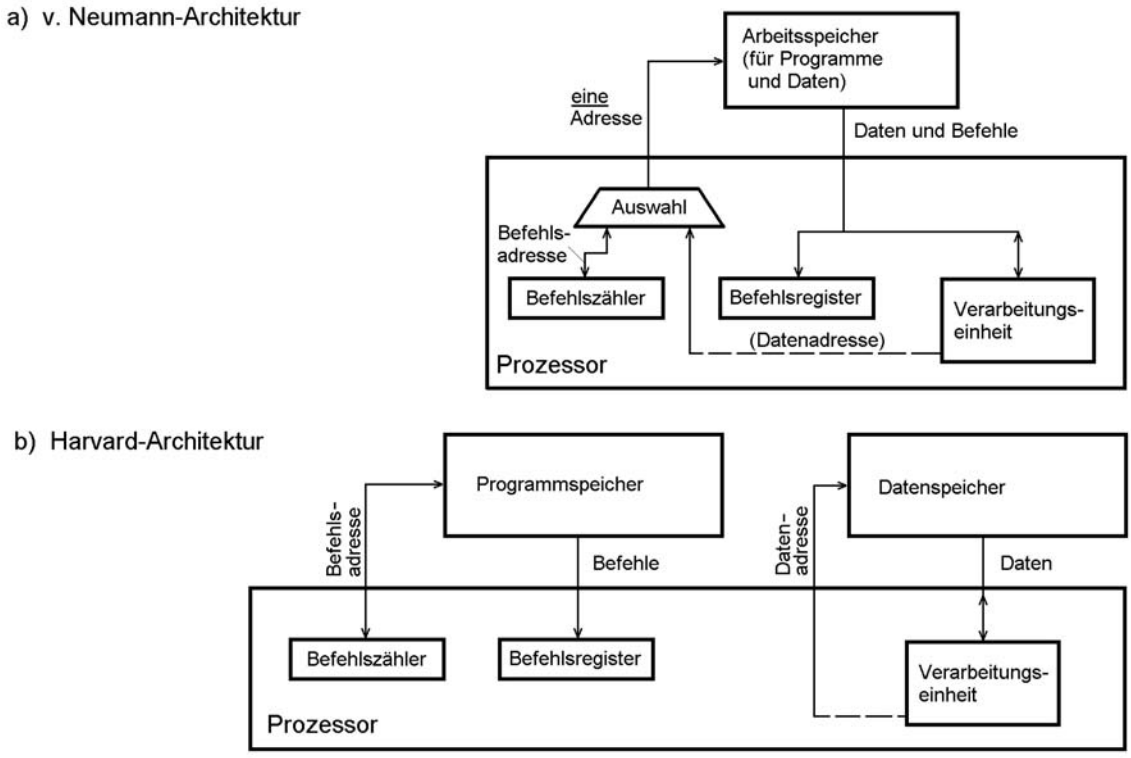


Abb. 9 v. Neumann.- und Harvard-Architektur

Zu den Namen:

- v. Neumann-Architektur: nach dem Mathematiker John v. Neumann,
- Harvard-Architektur: nach der Harvard-Universität (Cambridge, Massachusetts)[2)].

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 0 an fortlaufend adressierbaren Speicher, der alle Programme, Daten, deskriptiven Angaben usw. aufnimmt, die für die laufende Arbeit benötigt werden.

Harvard-Architektur
Es gibt zwei Speicheradreßräume – einen für Daten und einen für Befehle. Gemäß der technischen Auslegung unterscheiden wir:

1. echte Harvard-Maschinen
Es gibt eine gesonderte Speicheranordnung je Adreßraum (also einen Programmspeicher und einen Datenspeicher) und unabhängige Zugriffswege zu beiden Speicheranordnungen (Abb. 5b). Architekturbeispiel: Atmel AVR.

2. Maschinen mit Harvard-Architektur
Diese haben zwar die beiden getrennten Adreßräume, aber nur einen einzigen gemeinsamen Speicherzugriffsweg (Speicherbus). Architekturbeispiel: 8051.
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[3)]. 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[4)] usw. optimieren. Ist beispielsweise das Byte die elementare Datenstruktur, so müssen bei einer v. Neumann-Maschine auch die Befehlsformate in Bytestrukturen gezwängt werden (auch Befehle sind Aneinanderreihungen von Bytes). 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. U. a. sind Prozessoren mit Befehlen von 12, 14, 24 Bits usw. gebaut worden. Beispiel: PIC 16x und 24x.

---

2): die ersten funktionsfähigen Maschinen gemäß dieser Architektur wurden übrigens von Konrad Zuse gebaut (in Berlin) ...

3): man spricht bildhaft vom "v. Neumann-Flaschenhals" (v. Neumann Bottleneck).

4): z. B. liegt es nahe, den Programmspeicher eines Mikrocontrollers als ROM auszulegen und den Datenspeicher als RAM.

3. Höheres Adressierungsvermögen
Da es zwei Adreßräume gibt, wird das Adressierungsvermögen der Hardware praktisch verdoppelt. Beispiel: ein Prozessor sei für 16-Bit-Adressen ausgelegt (Byteadressierung). In einer v. Neumann-Architektur beträgt das Adressierungsvermögen insgesamt 2^{16} = 64 kBytes. Eine Harvard-Maschine könnte hingegen mit einem Befehls- und einem Datenspeicher von jeweils 64 kBytes bestückt werden (insgesamt 128 kBytes)[1)]. Die schlechte Nachricht: wenn wir z. B. 20 kBytes für die Programme, aber 100 kBytes für die Daten brauchen, so paßt es nicht (Abhilfe: tricksen ...).

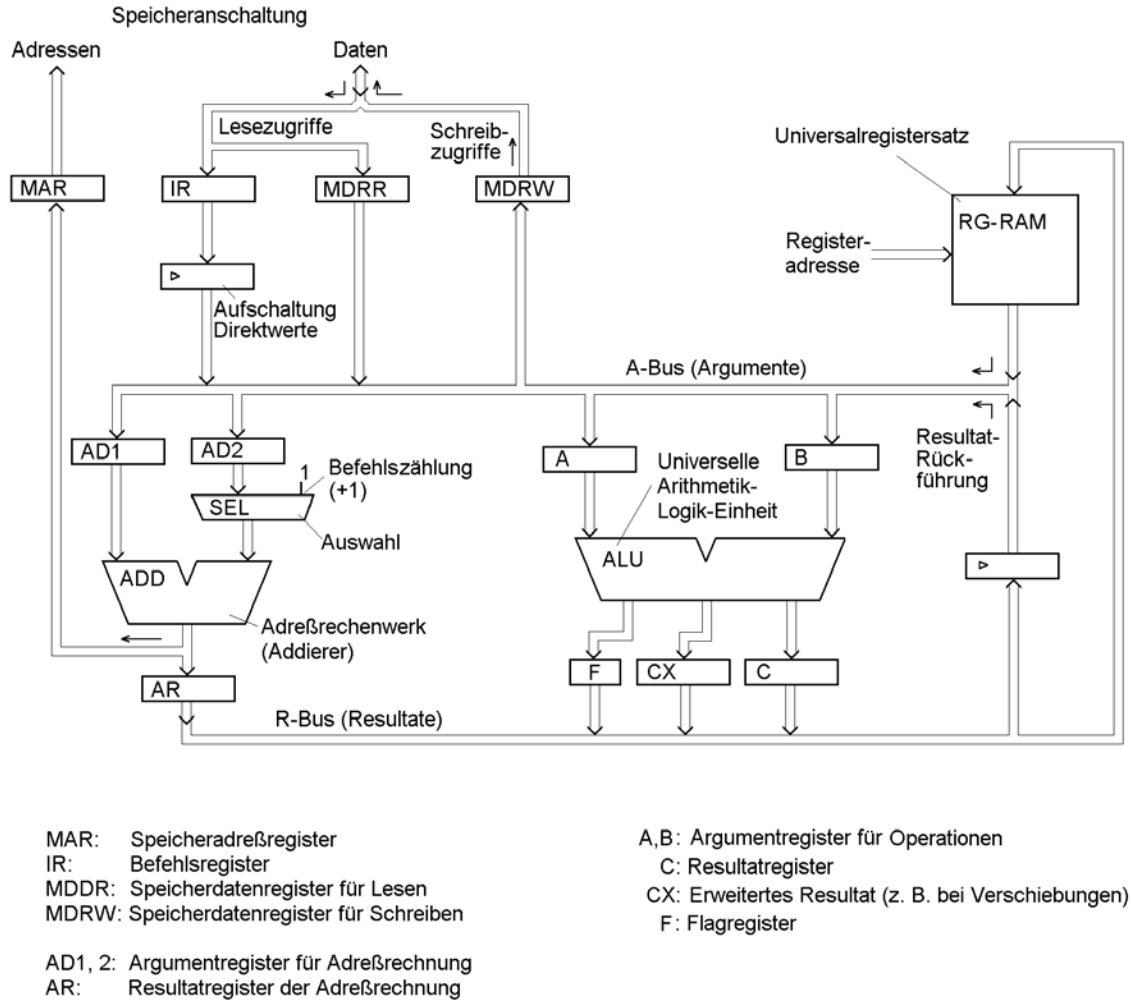
Welche dieser Architekturen bestimmt den Stand der Technik?
Alle modernen Hochleistungsprozessoren[2)] sind ausnahmslos v. Neumann-Maschinen. Das heißt, sie verhalten sich aus Sicht des Programmierers als solche. Intern gibt man sich aber Mühe, die Vorteile beider Architekturen zu vereinigen.Demgegenüber sind die meisten Signalprozessoren und auch viele der kleinen Mikrocontroller als Harvard-Maschinen ausgelegt (Tabelle 1).

Vorteil	Erläuterung	Beispiele
höheres Adressierungs-vermögen	weil es 2 unabhängige Adreßräume gibt	Intel 8051
technologische Trennung zwischen Daten- und Programmspeicher	Datenspeicher: SRAM, Programmspeicher: ROM (auch EPROM oder Flash)	Microchip PIC, Atmel AVR
interne Optimierung	Befehle werden so breit ausgelegt wie es jeweils zweckmäßig ist (also nicht in Bytestrukturen gepreßt)	Microchip PIC16x, 24x
Leistungssteigerung	durch parallelen Zugrifff auf Befehle und Daten	die meisten Signalprozessoren

Tabelle 1 Weshalb Mikrocontroller und Signalprozessoren oft als Harvard-Maschinen ausgelegt werden

1): deshalb hat man viele Mikrocontroller, obwohl sie nur einen einzigen Speicherbus haben, als Harvard-Maschinen ausgelegt. Beispiel: 8051.

2): und auch die meisten Prozessoren und Mikrocontroller der mittleren Leistungsklassen.



MAR: Speicheradreßregister
IR: Befehlsregister
MDDR: Speicherdatenregister für Lesen
MDRW: Speicherdatenregister für Schreiben

AD1, 2: Argumentregister für Adreßrechnung
AR: Resultatregister der Adreßrechnung

A,B: Argumentregister für Operationen
C: Resultatregister
CX: Erweitertes Resultat (z. B. bei Verschiebungen)
F: Flagregister

Abb. 10 Ein typischer CISC-Prozessor im Blockschaltbild

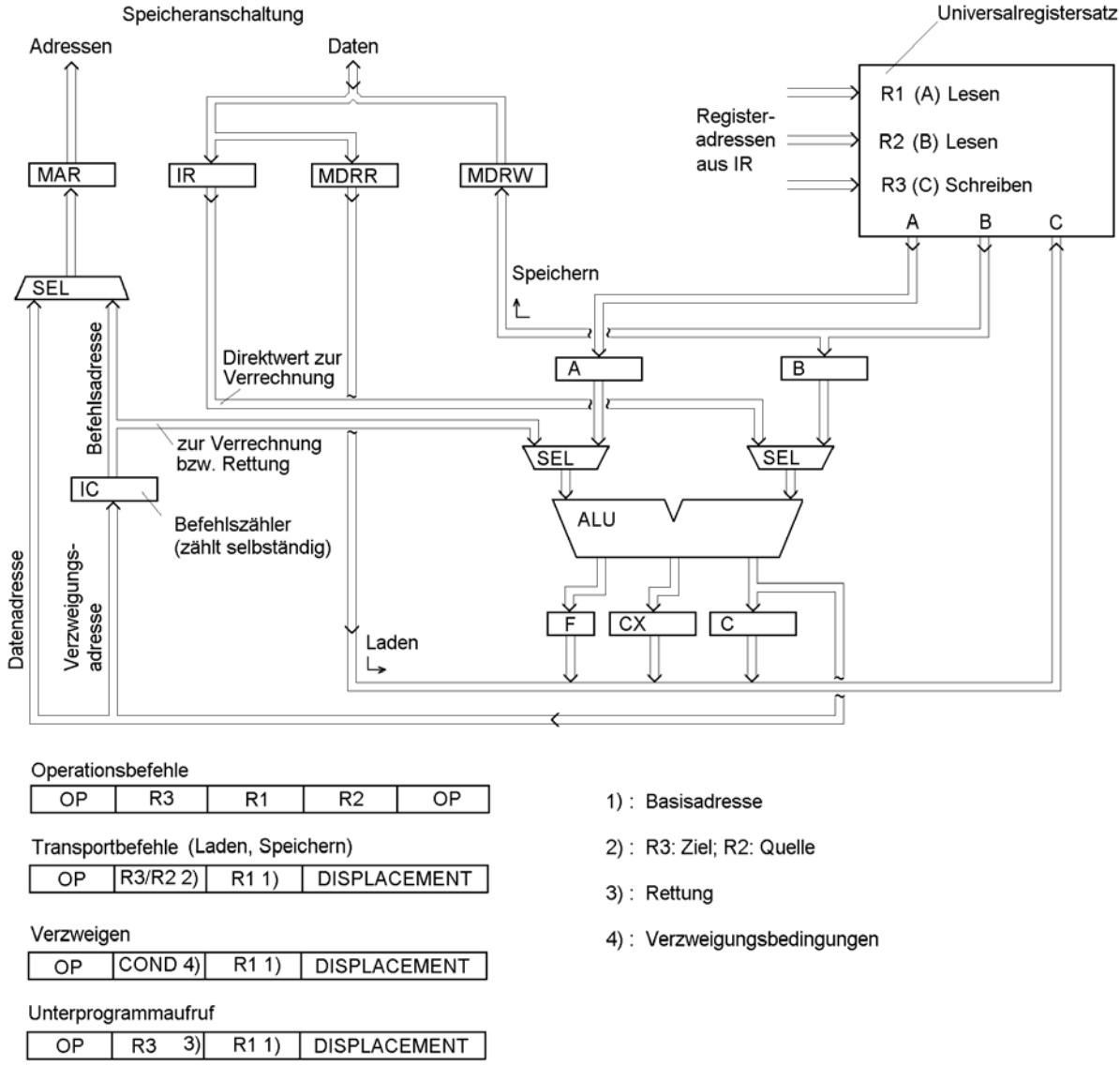


Abb. 11 Ein typischer RISC-Prozessor im Blockschaltbild

RISC für Praktiker

- Extreme akademische Auffassungen werden aufgegeben -

- komplexere Befehlswirkungen (alle 4 Grundrechenarten und mehr),
- hardwareseitige Steuerung der Pipeline (API von Pipeline-Struktur unabhängig).

Vom akademischen RISC-Ansatz verbleiben:

- Operationsbefehle der Art Register - Register - Register (3 Registeradressen),
- große Universalregistersätze (32 Register und mehr),
- Speicherzugriffe nur mit besonderen Befehlen (LOAD-STORE),
- einfacher Unterpogrammruf,
- keine Unterstützung variabel langer Datentypen.

Auch RISC-Prozessoren können mehr als 200 verschiedene Befehle haben (z. B. PowerPC)...

CISC	RISC
komplexe Befehle	**einfache Befehle**
Was ist hier komplex? • Operationsbefehle mit Speicherzugriff • verschiedene Arten der Adreßrechnung • Unterstützung aller 4 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, ist auszuprogrammieren
So großartig komplex ist CISC aber auch nicht – die meisten Befehle haben nur einen Speicher- und einen Registeroperanden: <R> := <R> **OP** <MEM>	
Befehle variabler Länge. Kompakt, aber aufwendig zu decodieren (ggf. mehrere Pipeline-Stufen).	Befehle fester Länge. Teils Platzverschwendung, aber einfach zu decodieren (oft genügt eine einzige Pipeline-Stufe).
Mikroprogrammsteuerung	sequentielle Steuerung
Leistungsvermögen des einzelnen Befehls entscheidend (darf länger dauern, wenn er entsprechend viel leistet). Anwendungsseitige Eleganz (Assemblerprogrammierung).	Ausführungsgeschwindigkeit entscheidend. Alle Befehlsabläufe müssen in ein Pipeline-Schema passen (es wird nur unterstützt, was in so ein Schema paßt - ohne Rücksicht auf Eleganz und Komfort). Ohne Compiler nur schwer zu programmieren.
Anwendungsproblem wird mit vergleichsweise wenigen, leistungsfähigen, langsam ablaufenden Befehlen gelöst. Speicherbandbreite vergleichsweise unkritisch. Wegen des kompakten Codes sind die Cache-Trefferraten hoch. Befriedigende Leistung auch auch mit vergleichsweise kleinen Caches. Steuerung kompliziert.	Anwendungsproblem wird mit elementaren, schnell ablaufenden Befehlen gelöst. Deshalb werden mehr Befehle benötigt. Speicherbandbreite leistungsentscheidend. Befriedigende Leistung erfordert bisweilen große Caches. Steuerung vergleichsweise einfach.
Befehlsliste (API) isoliert Hardware von Architektur (eine Architektur, viele verschiedenen kompatible Implementierungen). Shrinkwrapped Software praktikabel.	Architektur und Hardware hängen eng zusammen. Im (akademischen) Extremfall schlägt Struktur der Pipeline auf die API durch. Beeinträchtigt Narrenfreiheit beim Implementieren. Shrinkwrapped Software nicht immer praktikabel (statt dessen Neucompilierung für jede neue Hardware)

Tabelle 2 CISC und RISC

Elementare Programmiertechniken

Unterprogramme
Ein Unterprogramm ist ein Programmablauf, der mehrmals – an verschiedenen Stellen des eigentlichen (Haupt-) Programms – auszuführen ist. Die prinzipiellen Merkmale des Unterprogramms:

- es wird nur einmal gespeichert,
- es wird aus dem jeweiligen Hauptprogramm heraus aufgerufen,
- am Ende seines Ablaufs veranlaßt es eine Rückkehr in das jeweiligen Hauptprogramm.

Die Vorkehrungen, die erforderlich sind, um dieses Prinzip anwenden zu können, betreffen:

- den Aufruf (Unterprogramnruf) und die Rückkehr ins Hauptprogramm,
- die Übergabe der Parameter (der Eingangsdaten, die das Unterprogramm verarbeiten soll),
- die Rückgabe der Ergebnisse.

Unterprogrammruf (Subroutine Call)
Der Unterprogrammruf wirkt so, daß die (durch Weiterzählen gewonnene) Adresse des Folgebefehls vor dem Überladen gerettet wird (der Befehlszähler zählt zunächst weiter, dann wird sein Inhalt gerettet, dann wird er mit der Verzweigungsadresse überladen). "Retten" bedeutet die Überführung des Befehlszählerinhaltes in eine besondere, von der nachfolgenden Verarbeitung normalerweise nicht berührte, Register- oder Speicherposition.

Der Zweck dieser Vorkehrungen besteht darin, das Programm nach dem Unterprogrammruf wieder fortsetzen zu können (Abb. 12). Man verzweigt mit einem Unterprogrammruf aus einem Programm A zu dem (Unter-) Programm B und kann nach dessen Ausführung zum Programm A an der Stelle nach dem Unterprogrammruf zurückverzweigen. Dazu braucht man noch *Rückkehrbefehle*, die die gerettete Befehlsadresse (Rückkehradresse, Return Address) wieder in den Befehlszähler laden.

Adreßrettung beim Unterprogrammruf:

- festes Register (Linkregister)
- feste Speicherposition
- auswählbares (Universal-) Register
- auswählbare (adressierte) Speicherposition
- Hardwarestack
- Stack im Arbeitsspeicher

Parameter und deren Übergabe
Woher nimmt das Unterprogramm die Daten, die es verarbeiten soll? Diese Daten heißen *Parameter*. Sie müssen an das Unterprogramm übergeben werden (Parameter Passing). Hierfür sind folgende Formen üblich:

- in Registern (Beispiel: PC-BIOS),
- in eigens vereinbarten Speicherbereichen (Beispiel: manche Funktionen des PC-BIOS (z. B. Unterstützung großer Festplatten),
- im Stack (Beispiel: C, Windows API usw.),
- im Programm selbst (an den Unterprogrammruf anschließend),
- Kombinationen dieser Verfahren.

Übergabeprinzipien:

- Wertübergabe (by Value)
- Adreßübergabe (by Reference)

Rückgabe von Ablaufinformationen (z. B. Unterscheidung o.k./Fehler):

1. in Register (vgl. PC-BIOS,
2. in den Flagbits (komfortabel, weii so leicht verzweigt werden kann),
3. im Stack (Fehlercode = Funktionswert (eine typische C-Programmiergepflogenheit)),
4. durch Änderung der Rückkehradresse (sehr komfortabel, weil es Abfragen grundsätzlich erspart). Z. B. wenn Fehler, Rückkehr auf Folgeadresse, wenn o.k., Rückkehr auf übernächste Adresse. Ermöglicht es, so zu programmieren:

   CALL
   JMP Error_handler
   ... Fortsetzung...

Um Unterprogramme besonders wirkungsvoll zu nutzen, muß man sie "ineinanderschachteln" können (Nested Subroutine Calls; Programm A ruft Unterprogramm B, dieses ruft Unterprogramm C usw.).

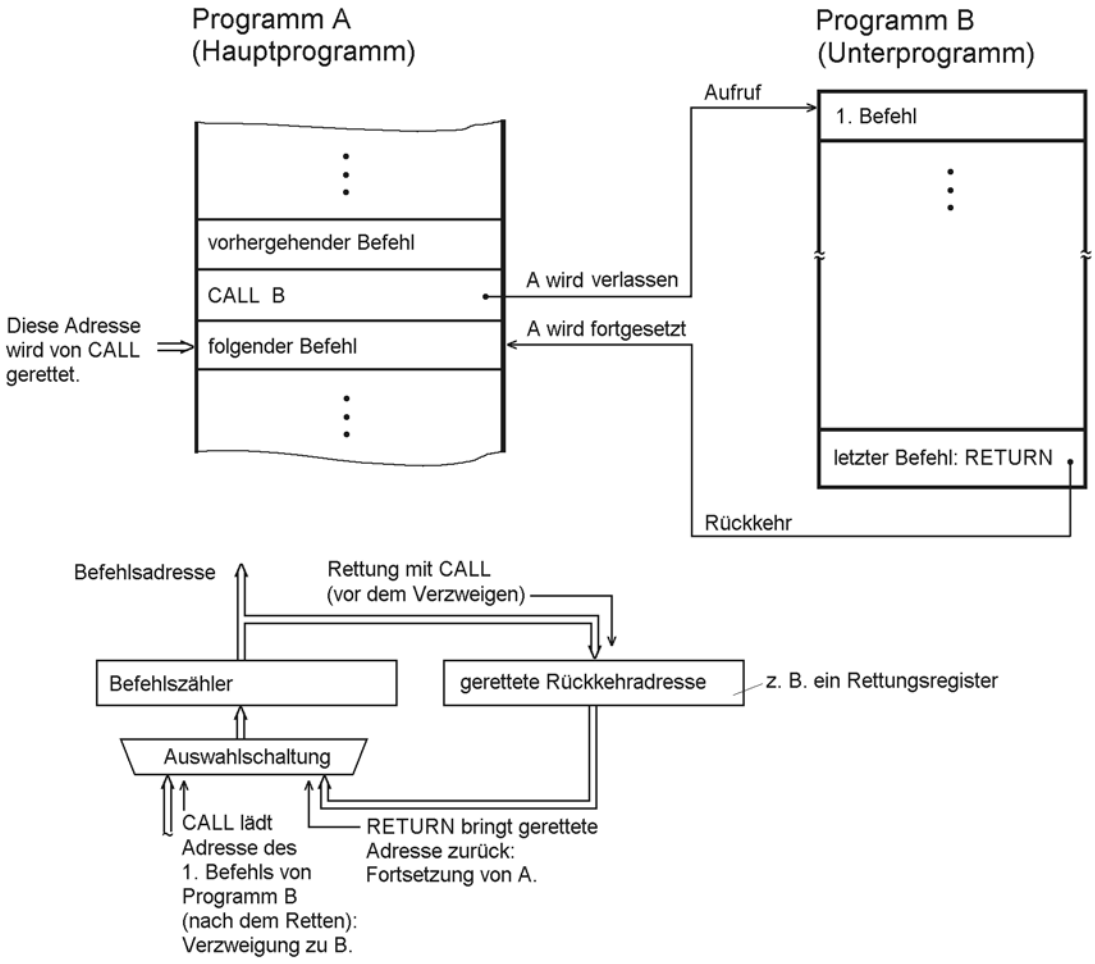
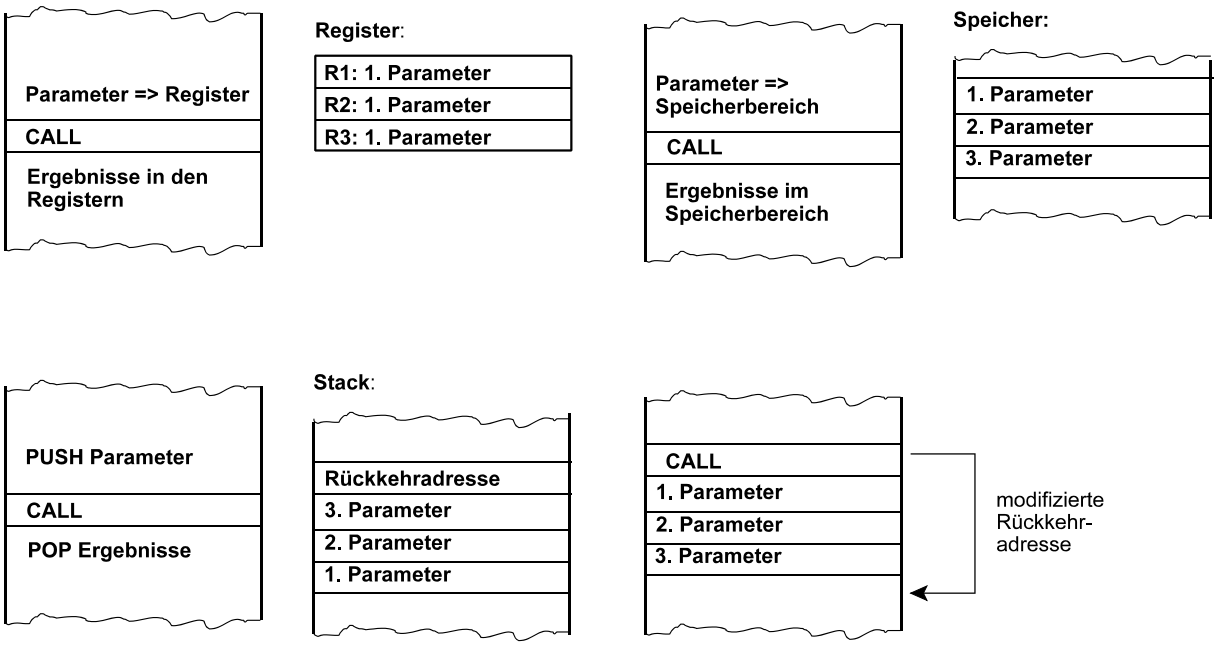


Abb. 12 Prinzip des Unterprogrammrufs



Was ist außer der Befehlsadresse zu retten?

- Arbeitsregister
- Flagbits

Verzweigungen

Sprungdistanz
Oft unzureichend. Kompromiß zwischen Bedingungsauswahl und Sprungdistanz. Viele Befehle werden zum Schließen von Schleifen genutzt, wozu meist eine geringe Sprungdistanz (mit Bezug auf den Befehlszähler) ausreicht.

Bezugsadresse:

- absolut (lineare Adresse 0),
- Basisregister (oder Segment),
- Befehlszähler

Springen und Überspringen (Skip)
Bei zu geringer Sprungdistanz in bedingten Verzweigungsbefehlen eine passende unbedingte Verzweigung überspringen (mit der komplementären Sprungbedingung). Auf diese Weise lassen sich auch Unterprogramme bedingt aufrufen, wenn es keine bedingten CALLs gibt.

Weiter weg springen durch Überspringen der komplementären Sprungbedingung:

IF carry THEN GOTO weit_weg

Ideal wäre: BRANCH_ON_CARRY, weit_weg

Ausweichlösungen (je nach Befehlsvorrat):

SKIP_ON_NO_CARRY JUMP weit_weg	BRANCH_ON_NO_CARRY, weiter JUMP weit_weg weiter: ...

Aufgabe:
Es soll ein Unterprogramm OUT_OF_RANGE gerufen werden, falls das CARRY-Flag gesetzt ist.

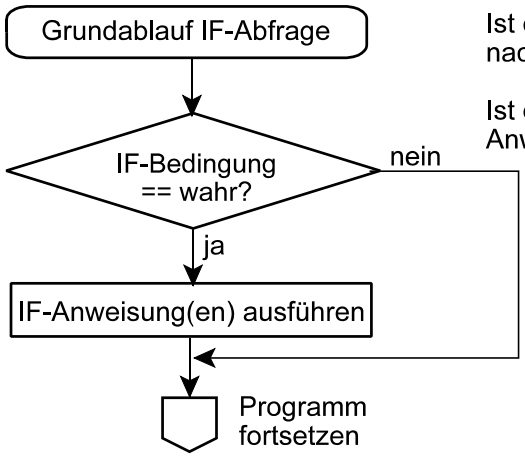
Springen mit berechneten Adressen (Funktionsverzweigung)
Indirekte Verzweigung mit Adresse aus Register.

Gewährleistung der Verschieblichkeit ohne Relativadressierung (im einzelnen Programm) oder ohne ladbare Bezugsadresse (z. B. nur PC-relative Verzweigungen):

Mit Header und Ladeprogramm. Befehlsmodifikation beim Laden. Header enthält Zeiger auf die zu modifizierenden Befehle (vgl. .EXE-Dateien, DLLs).

IF A then B

```
        Test A (Liefert eine Sprungentscheidung 1/0 =  JA/NEIN)
        BRANCH IF 0, WEITER
        B ausführen
WEITER:
```



Ist die IF-Bedingung erfüllt (wahr), werden die nachfolgenden Anweisungen ausgeführt.

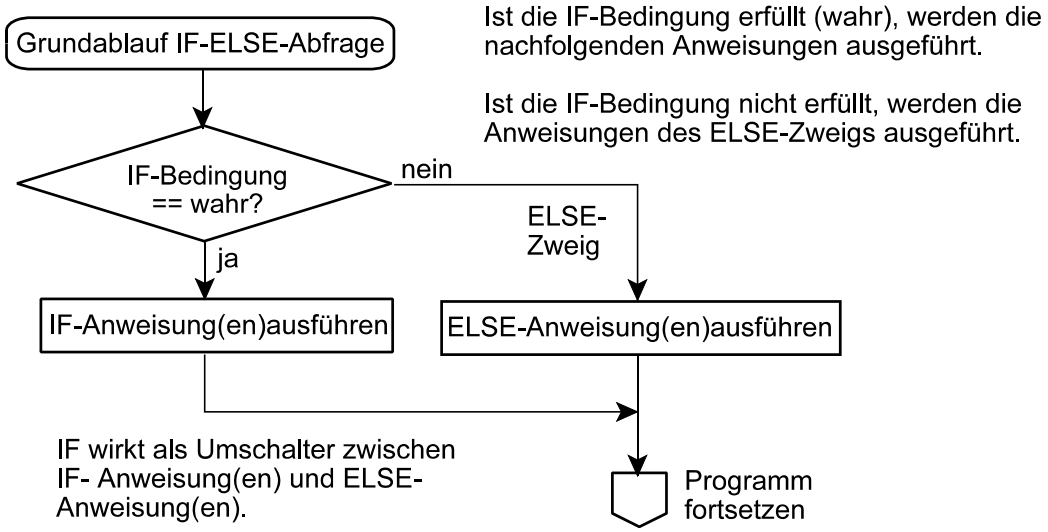
Ist die IF-Bedingung nicht erfüllt, werden diese Anweisungen übergangen.

IF wirkt als Umschalter zwischen IF- Anweisung(en) und Programm-fortsetzung (IF-Anweisung(en) ausführen oder nicht ausführen).

	IF-Bedingung A abfragen
	wenn nicht erfüllt, zu WEITER
	B ausführen (IF-Zweig)
WEITER:	Programm fortsetzen

IF A then B else C

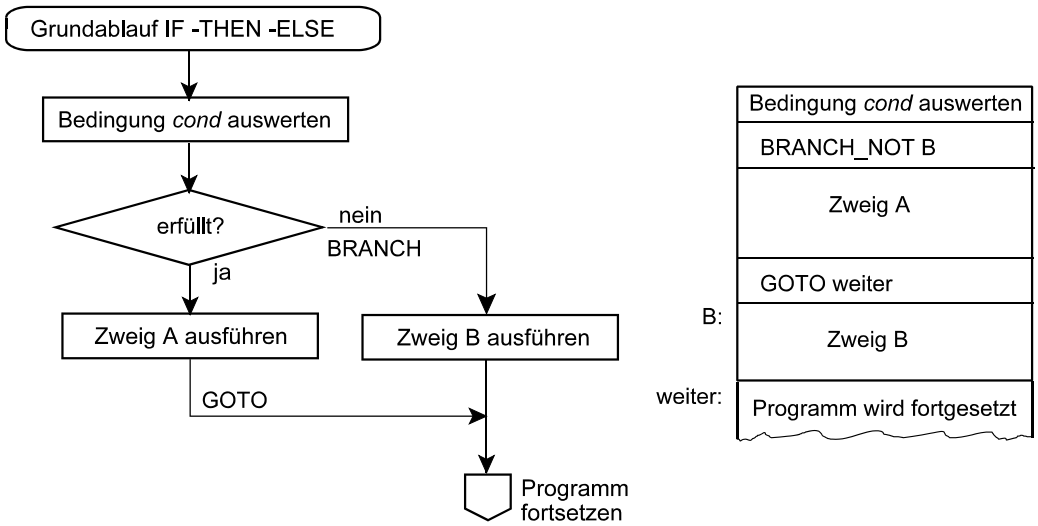
```
            Test A (Liefert eine Sprungentscheidung 1/0 =  JA/NEIN)
            BRANCH IF 1, ELSE   -- gehe zu else-Zweig (C)
            B ausführen
            JUMP WEITER -- else-Zweig übergehen
ELSE:       C ausführen
WEITER: ...
```



	IF-Bedingung A abfragen
	wenn nicht erfüllt, zu ELSE
	B ausführen (IF-Zweig)
	zu WEITER
ELSE:	C ausführen (ELSE-Zweig)
WEITER:	Programm fortsetzen

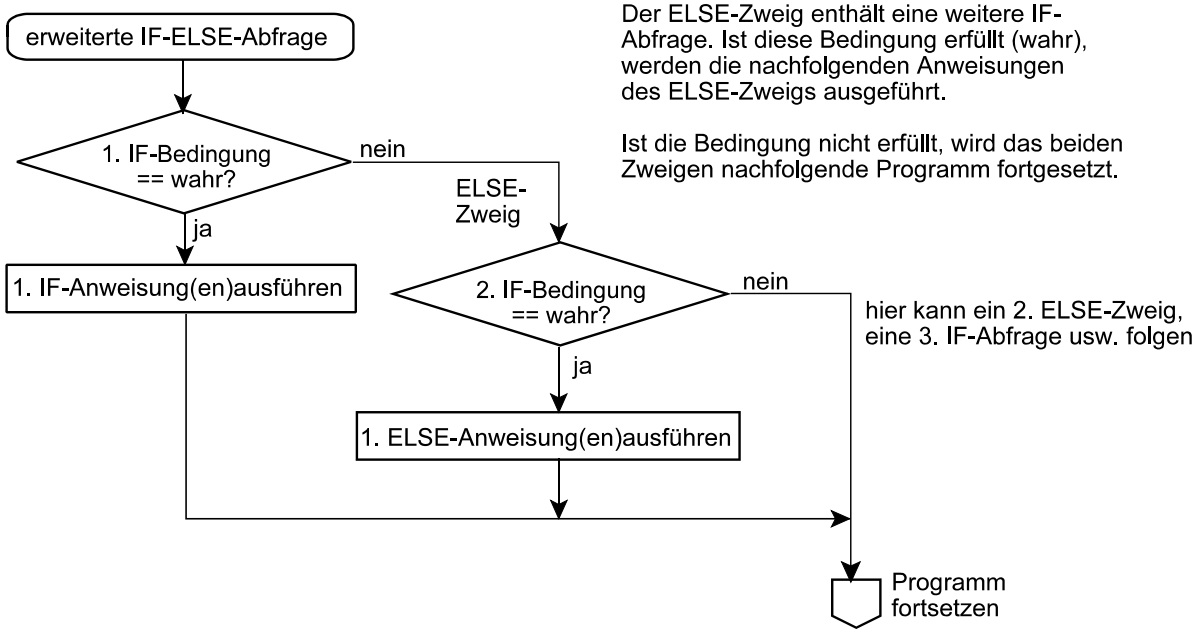
Praxistip:
Entsprechende Muster eingeben – erst nur das Skelett, dann die Abläufe im einzelnen. Alles so kommentieren, daß die Programmierabsicht erkennbar wird.

IF *cond* THEN A ELSE B



IF A then B else if X then C else D

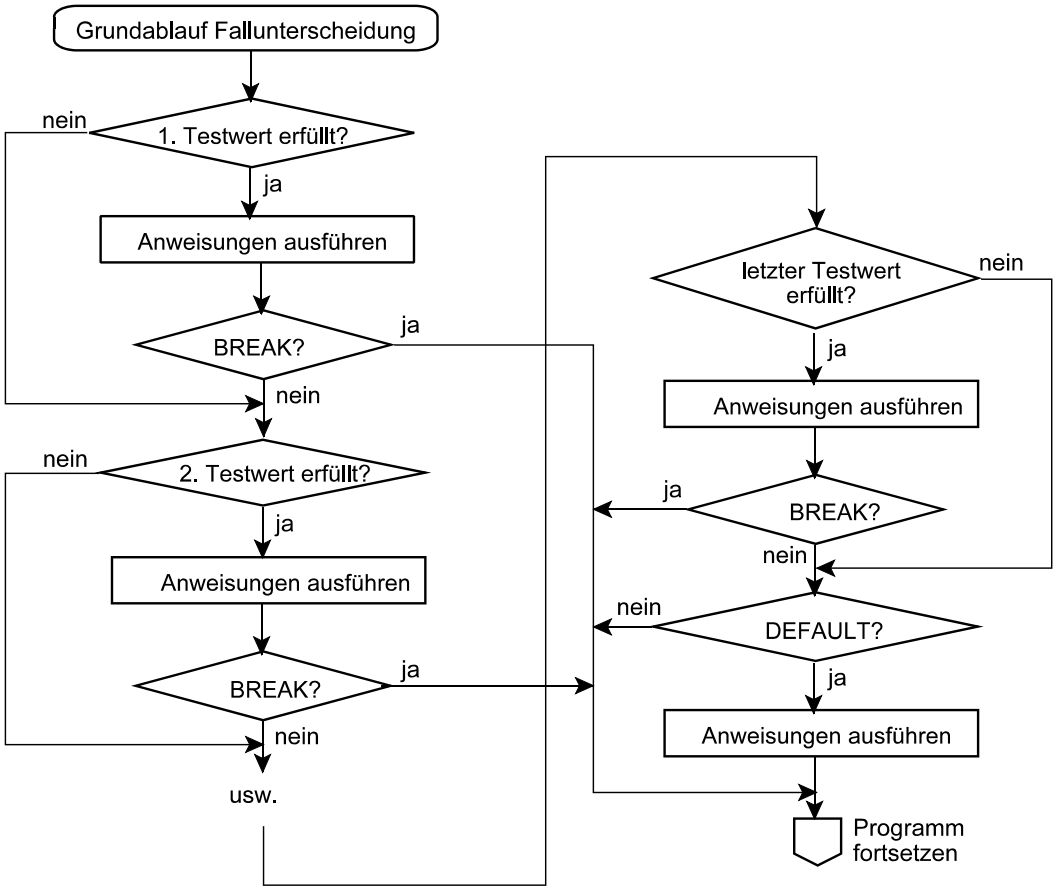
```
                Test A
                BRANCH IF 0, TO_NEXT
                B ausführen
                JUMP WEITER
TO_NEXT:        Test X
                BRANCH IF 0, TO_D
                C ausführen
                JUMP WEITER
                TO_D: D ausführen
WEITER: ...
```



Der ELSE-Zweig enthält eine weitere IF-Abfrage. Ist diese Bedingung erfüllt (wahr), werden die nachfolgenden Anweisungen des ELSE-Zweigs ausgeführt.

Ist die Bedingung nicht erfüllt, wird das beiden Zweigen nachfolgende Programm fortgesetzt.

	IF-Bedingung A abfragen
	wenn nicht erfüllt, zu NEXT
	B ausführen (1. IF-Zweig)
	zu WEITER
NEXT:	IF-Bedingung X abfragen
	wenn nicht erfüllt, zu D
C:	C ausführen (2. IF.-Zweig)
	zu WEITER
D:	D ausführen (2.ELSE-Zweig)
WEITER:	Programm fortsetzen



Alternativen:

1. Testwerte einzeln abfragen.
2. Testwerte so einrichten, daß sie ein bestimmtes Intervall bilden. Hierfür eine Verzweigungstabelle aufbauen.

Schleifen:

1. Eintritt, Anfangswerte
2. Wann wird abgefragt?
3. Mit welchen Werten wird der Schleifenkörper ausgeführt?

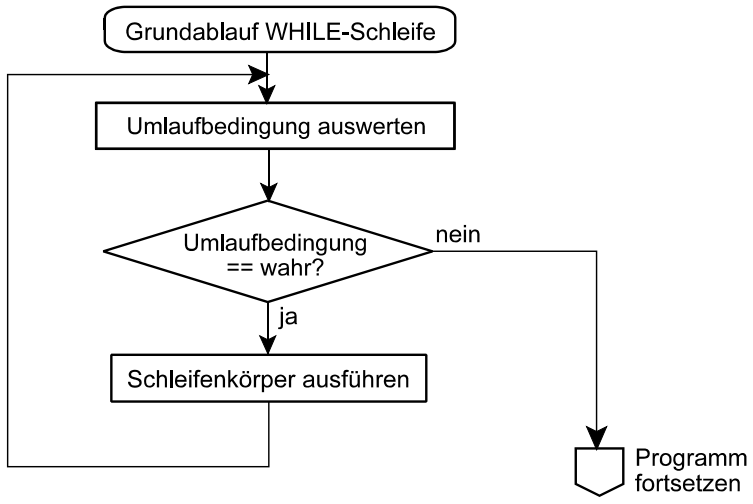
Sonderverzweigungen in Schleifen:

1. Schleifenkörper beenden und zur Auswertung der Umlaufbedingung verzweigen (continue).
2. Schleife hart beenden (break).
3. Aus der Schleife heraus irgendwohin verzweigen (goto).

1. WHILE-Schleife:

```
while (Umlaufbedingung)

{

-- Schleifenkörper --

}
```

Die Umlaufbedingung wird am Anfang geprüft. Schleife wird durchlaufen, solange Umlaufbedingung erfüllt (= logisch wahr). Schleife wird nie durchlaufen, wenn Umlaufbedingung schon zu Anfang nicht erfüllt (= logisch falsch).

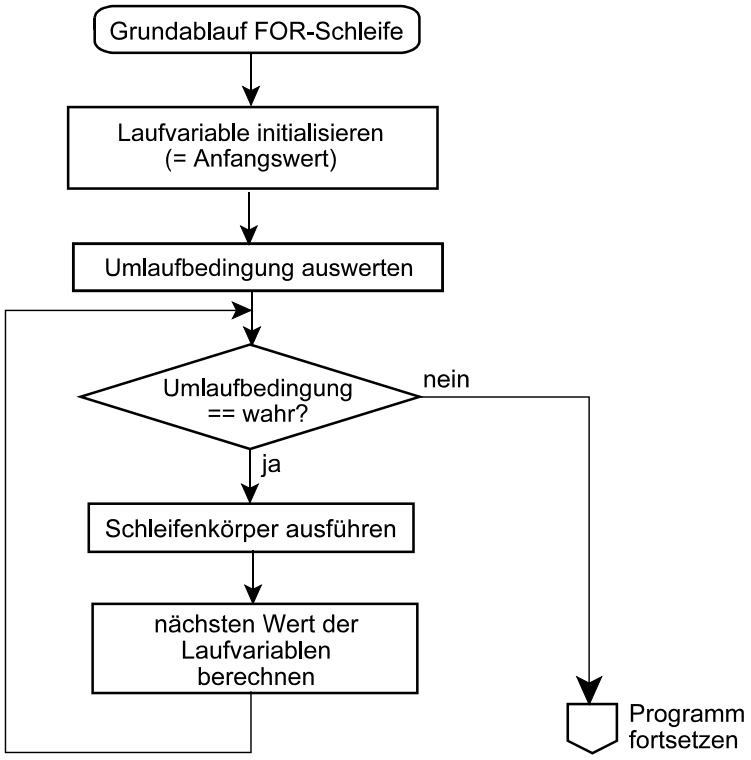


LOOP:	Umlaufbedingung auswerten
	wenn nicht erfüllt, zu WEITER
	Schleifenkörper ausführen
	zu LOOP
WEITER:	Programm fortsetzen

2. FOR-Schleife:

```
for (Laufvariable = Anfangswert; Umlaufbedingung; Wertberechnung für Laufvariable)
{

-- Schleifenkörper --

}
```

Zu Beginn der Schleife wird Laufvariable initialisiert. Dann wird Umlaufbedingung geprüft. Schleife wird durchlaufen, wenn Umlaufbedingung erfüllt (= logisch wahr). Nach dem Durchlauf wird der neue Wert der Laufvariablen berechnet. Schleife wird nie durchlaufen, wenn Umlaufbedingung schon zu Anfang nicht erfüllt (= logisch falsch).

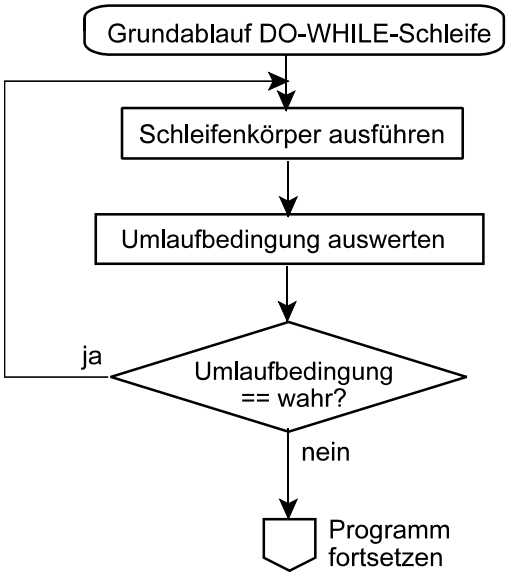


	Laufvariable initialisieren
LOOP:	Umlaufbedingung auswerten
	wenn nicht erfüllt, zu WEITER
	Schleifenkörper ausführen
	nächsten Wert der Laufvariablen berechnen
	zu LOOP
WEITER:	Programm fortsetzen

3. DO-WHILE-Schleife:

```
do
{

-- Schleifenkörper --

}

while (Umlaufbedingung);
```

Die Umlaufbedingung wird erst am Ende geprüft. Schleife wird somit wenigstens einmal durchlaufen. Schleife wird weiterhin durchlaufen, solange Umlaufbedingung erfüllt (= logisch wahr).



LOOP:	Schleifenkörper ausführen
	Umlaufbedingung auswerten
	wenn erfüllt, zu LOOP
	Programm fortsetzen

Grundlagen der Adreßrechnung

Absolute und direkte Adressierung

Alle Adreßangaben sind Direktwerte in Befehlen (Abb. 13). Das klassische Beispiel: Zuse Z3. Der Vorteil: Einfachheit. Die wesentlichen Nachteile:

- Datenstrukturen und Programme sind nicht verschieblich (relocatable); sie müssen vielmehr an festen Plätzen gespeichert werden,
- es ist nicht möglich, Programmschleifen auf Elemente höher aggregierter Datenstrukturen anzuwenden (z. B. auf Vektoren und Matrizen (Arrays).

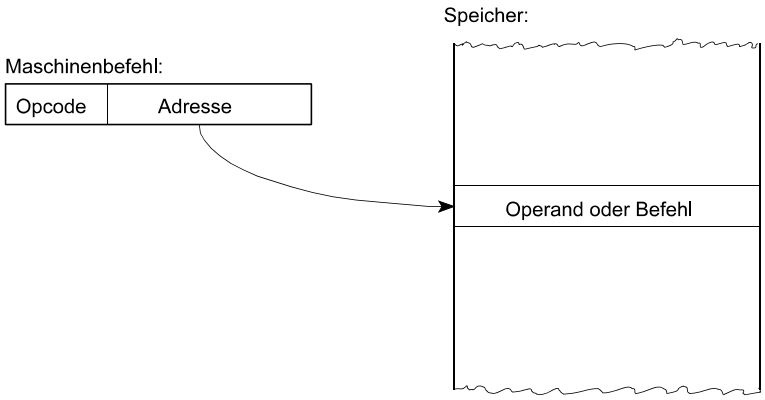


Abb. 13 Absolute und direkte Adressierung

Ein Trick, um die Nachteile zu überwinden: Selbstmodifizierende Programme. Adreßrechnung modifiziert die Adreßangaben in den Zugriffs- bzw. Verzweigungsbefehlen. Selbstmodifizierende Programme sind aber – aus guten Gründen – seit längerem nicht mehr in Mode.

Universelle Adreßregister (Registeradressierung)

Prinzip: die betreffende Adresse wird aus einem Register entnommen (Abb. 14). "Universell" bedeutet hier, daß der Registerinhalt ohne weiteres in beliebige Verarbeitungsabläufe einbezogen werden kann. Bei "echten" Universalregistern ist dies ohne weiteres gegeben; Adreßregister im eigentlichen Sinne (wie bei CDC 6600 oder Motorola 68k) müssen zumindest für Lese- und Schreibzugriffe zugänglich sein.

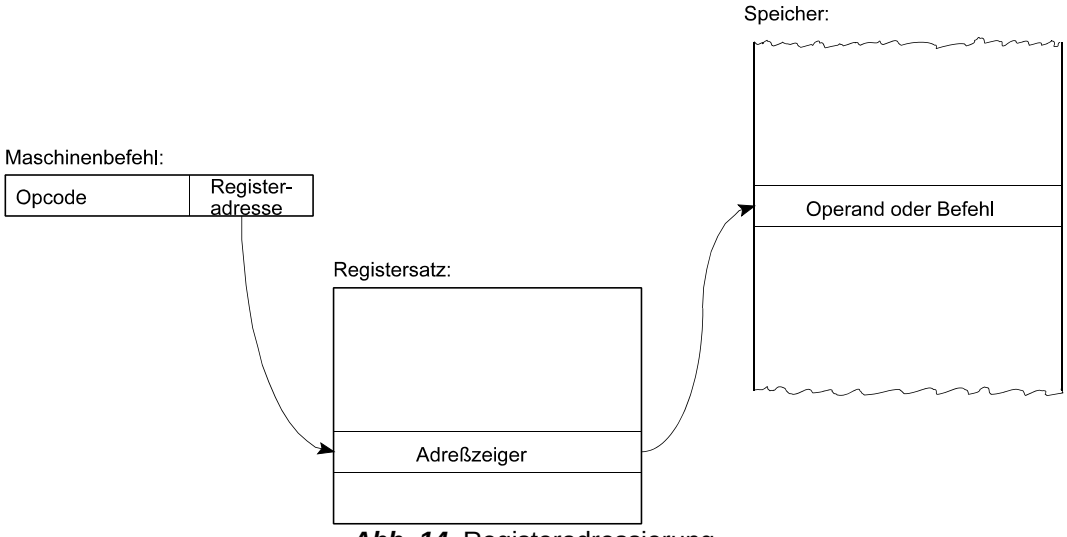


Abb. 14 Registeradressierung

Indirekte Adressierung ("Adresse von Adresse")
Eine Adreßangabe ist z. B. als Direktwert im Befehl oder als Registerinhalt gegeben. Der hiermt adressierte Register- oder Speicherinhalt wird aber nicht als Datenstruktur oder als Befehl, sondern als weitere Adresse interpretiert (Abb. 15). Mit dieser Adresse wird dann der eigentliche Zugriff ausgeführt. Das Prinzip entspricht an sich der Registeradressierung, nur ist man bei der Unterbringung von Adressen nicht auf den Registersatz beschränkt, sondern kann den Arbeitsspeicher dazu ausnutzen.

Mehrstufige indirekte Adressierung (“Adresse von Adresse von Adresse...”)
Die ursprüngliche Adreßangabe adressiert eine Speicherposition, deren Inhalt adressiert eine weitere Speicherposition usw. Der letzte dieser Adreßzeiger verweist schließlich auf den gewünschten Inhalt. Es gibt zwei Möglichkeiten, das Ende dieser indirekten Adressierung zu kennzeichnen:

- durch eine Endekennung in den Adreßzeigern (Adreßzeiger muß länger sein als die Adreßangabe (Beispiel: PDP-1),
- durch Angabe der Stufenzahl (Levels of Indirection).

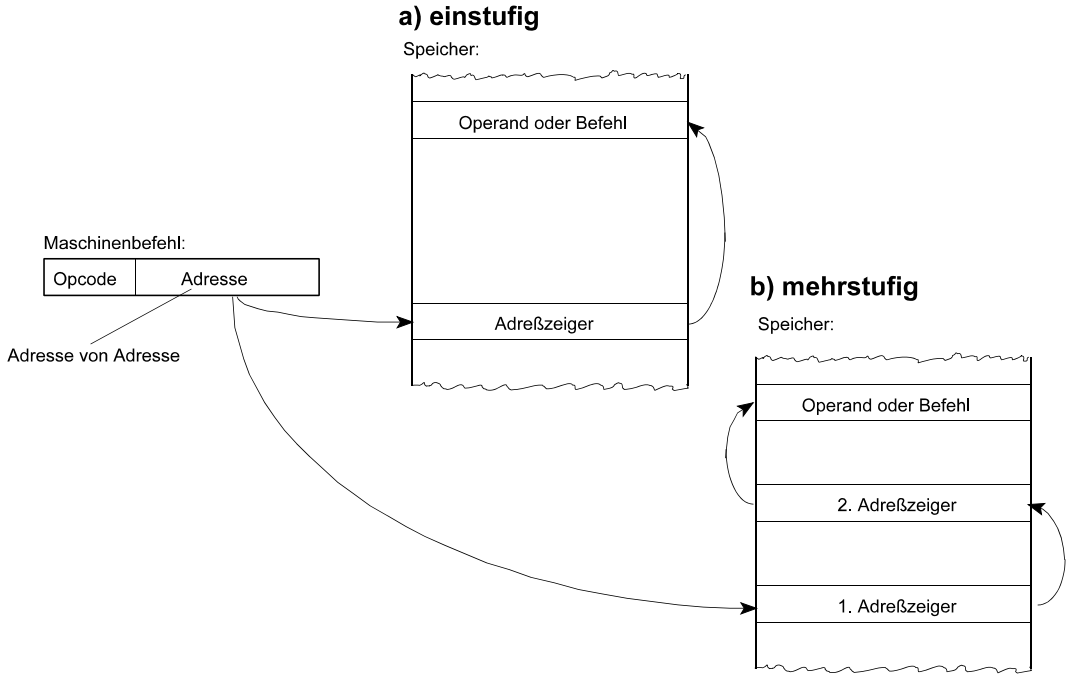


Abb. 15 Indirekte Adressierung

Was sind die einfachsten wirklich universellen Adressierungsweisen?

1. Registeradressierung + Direktwert-Ladebefehle. Jedem Zugriff geht das Laden eines Adreßregisters bzw. eine entsprechende Adreßrechung voraus. Die einfachste Form: das Laden der Adresse als Direktwert.
2. einstufige indirekter Adressierung + Absolutadressierung (um die Speicherzellen erreichen zu können, die die Adreßzeiger enthalten).

Wieviele Adressierungsweisen braucht man wirklich? – Die herkömmliche RISC-Philosophie
Die Erfahrung hat gezeigt, daß die Form *Basis + Displacement* (Abb. 16) vollauf ausreichend ist – allerdings unter der Voraussetzung, daß genügend Register zur Verfügung stehen, um Adreßangaben und Daten gleichzeitig halten zu können[3]. Die Basisadresse befindet sich in einem allgemeinen Register, und die Displacementangabe ist als Direktwert im Befehl untergebracht. Sie wird als ganze Binärzahl interpretiert und mit Vorzeichenerweiterung verrechnet. Displacement 0 bewirkt die reine Registeradressierung. Solche Displacementangaben sind üblicherweise zwischen 13 und 16 Bits lang (diese Größenordnung hat sich in der Praxis als zumeist voll ausreichend erwiesen)[4]. Mit schnell ablaufenden elementaren Befehlen lassen sich auf dieser Grundlage alle komplexeren Adreßrechenvorgänge ausprogrammieren.

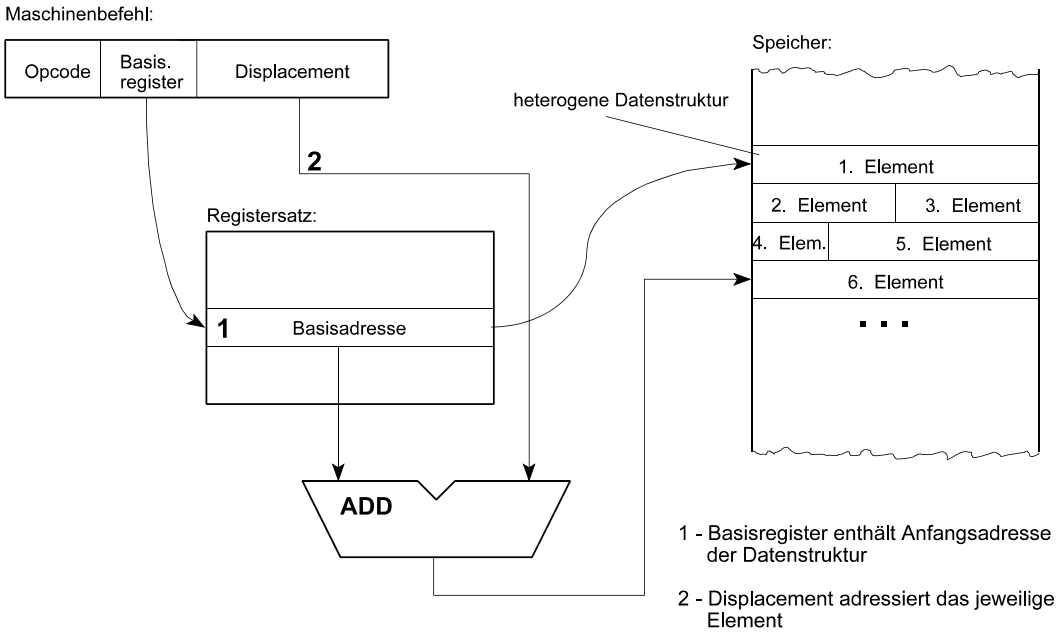


Abb. 16 Adressierungsprinzip Basis + Displacement.

Stackorganisation und -adressierung
Das Stack- (Kellerspeicher-) Prinzip ist in der Informatik von grundsätzlicher Bedeutung, namentlich was die Programmiersprachen, die Compiler und die Systemsoftware angeht. Manche Architekturen haben Vorkehrungen, um Stacks zu unterstützen, manche nicht (dann müssen Stacks als normale Datenbereiche vorgesehen werden, deren Verwaltung mit elementaren Befehlen auszuprogrammieren ist). Die Grundprinzipien bleiben stets die gleichen.

3 Mindestens 16, besser 32.

4 Diese Auslegung beschränkt die typischen Zugriffe auf das Schema von Abb. 1.6. Basis + Displacement unterstützt die typischen Zugriffe auf Activation Records (Stack Frames) und auf Einträge im Stack (mit Stackpointer als Basisregister). Bei Zugriffen auf homogene Datenstrukturen ist die Adreßrechnung auszuprogrammieren.

Grundlagen
Ein Stack ist eine Speicheranordnung, die eine gewisse Anzahl gleich langer Informationsstrukturen (*Stack-Elemente*) aufnehmen kann. Es gibt keinen wahlfreien Zugriff, sondern die Speicheranordnung wird implizit von einem Adreßzähler (*Stackpointer*) adressiert.

Stackzugriffe
Es gibt nur zwei grundlegende Zugriffsabläufe:

- ein *Push*-Ablauf legt ein Element auf den Stack,
- ein *Pop*-Ablauf entnimmt das zuletzt (vom letzten Push) auf den Stack gelegte Element (beim nächsten Pop wird dann das vom vorletzten Push abgelegte Element entnommen usw.).

Die Stack-Organisation wird deshalb gelegentlich auch als LIFO (Last In, First Out) bezeichnet.

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 (auch bei AVR) wachsen Stacks immer in Richtung niedere Adressen, d. h. der Stackpointer zeigt anfänglich immer auf die höchstwertige Adresse. Sein Inhalt wird bei Push-Abläufen vermindert und bei Pop-Abläufen erhöht.

Zähl- und Zugriffsreihenfolge
Ebenso ist es eine reine Konventionsfrage, ob bei einem Push zunächst der Stackpointer verändert und dann das neue Element gespeichert wird oder umgekehrt. Tabelle 3 nennt zwei typische Beispiele.

Architektur	Intel IA-32 (x86)	Atmel AVR
Stackponter zeigt auf...	das oberste Element im Stack (Top of Stack, TOS)	die erste freie Stackposition
Push-Ablauf	Stackpointer-Inhalt wird zunächst vermindert (Prädecrement), dann wird das Element gespeichert.	Das Byte wird auf den Stack gelegt, dann wird der Inhalt des Stackpointers um 1 vermindert (Postdecrement).
Pop-Ablauf	das Element wird entnommen, dann wird der Stackpointer-Inhalt erhöht (Postincrement).	Der der Inhalt des Stackpointers wird erhöht (Präincrement), dann wird das im Stack adressierte Byte entnommen.

Tabelle 3 Stack-Konventionen im Vergleich. In beiden Fällen Wachstum in Richtung niederer Adressen. Ein mit einem der Register X, Y oder Z implementierter Software-Stack würde sich wie ein Intel-Stack verhalten Prädecrement/Postincrement).

Verfeinerungen

Stack-relative Adressierung
Es ist oft von Vorteil, wenn man zu Elementen des Stack auch wahlfrei zugreifen kann. So kann man auch untere Elemente im Stack erreichen, ohne die oberen zuvor entfernen zu müssen. Solche Zugriffe beziehen sich zweckmäßigerweise auf den Stackpointer, so daß das erste, zweite usw. Element im Stack für Lese- und Schreibzugriffe zugänglich ist, wobei der Stackpointer nicht verändert wird (explizite Stackzugriffe nach dem Prinzip Basis + Displacement mit dem Stackpointer als Basisadreßregister).

Variabel lange Stackelemente
In den meisten Architekturen, so sie überhaupt Stacks vorsehen, sind alle Elemente in einem Stack *gleich lang*. Kürzere Angaben werden zwecks Ablage auf dem Stack entsprechend erweitert.

Stack Frames
Ein Stack Frame ist ein fester Bereich im Stack. Er dient vor allem dazu, die statischen 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). Beispiel (wir verwenden der Anschaulichkeit halber eine an Pascal und Ada orientierte Syntax):

```
Artikel_Nr: Integer;                    -- 4 Bytes (ganze 32-Bit-Binärzahl)
Bezeichnung: String(64);                -- 64 Bytes (Zeichenkette)
Preis: Unpacked_BCD(16);                -- 16 Bytes (BCD-Zahl)
Länge, Breite, Höhe: Small_Integer;     -- je 2 Bytes (ganze 16-Bit-Binärzahlen)
Gewicht, Spezifisches_Gewicht: Float;   -- je 4 Bytes (32-Bit-Gleitkommazahlen)
usw.
```

Jeder diese Variablen muß der Compiler entsprechenden Speicherplatz zuweisen.

Dynamische Variable entstehen hingegen im Laufe der Verarbeitung (also ohne daß sie der Programmierer ausdrücklich deklarieren muß). Beispiel: der Programmierer schreibt hin:

```
Gewicht := Länge * Breite * Höhe * Spezifisches_Gewicht;
```

Der Compiler muß diese Formel in eine Folge von Maschinenbefehlen umsetzen (hierbei sind u. a. verschiedene Datentypen ineinander zu wandeln). Da die einzelnen Befehle nur ganz elementare Operationen ausführen können, fallen im Verlauf der Rechnung Zwischenergebnisse an. Dies sind die dynamischen Variablen, die typischerweise 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). Dann liegt es nahe, die verfügbare Speicherkapazität im Sinne eines Stack zu verwalten und den Speicher vom oberen Ende her aufzufüllen (vgl. weiter unten Abb. 18). 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 sog. Frame Pointer oder Base Pointer (Abb. 17).

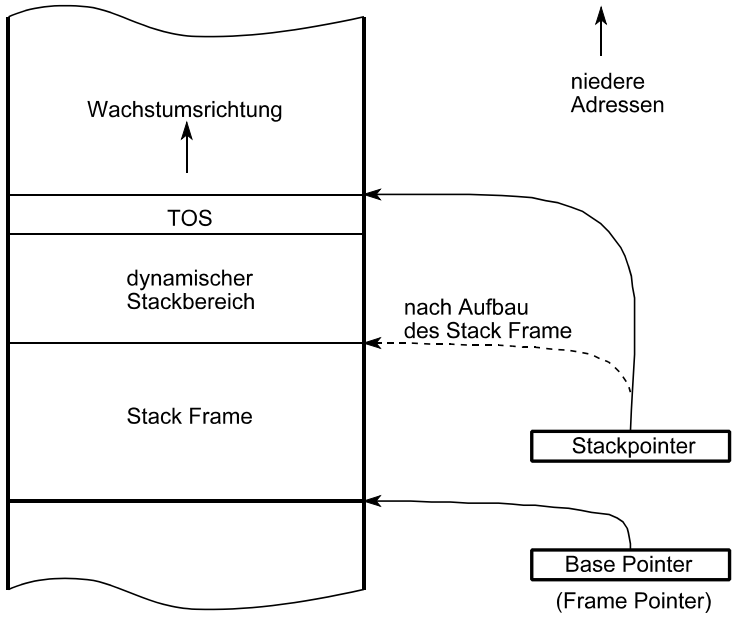


Abb. 17 Stack-Organisation mit Stack Frame

Der Base Pointer (Frame Pointer) zeigt stets auf den Anfang des aktuellen Stack Frame (d. h. auf das Wort an der jeweils höchsten Adresse). Alle Inhalte des aktuellen Stack Frame sind somit über negative Displacements (bezogen auf den Base Pointer) erreichbar.

Ruft das aktive Programm seinerseits ein Unterprogramm, so wird der aktuelle Inhalt des Stackpointers in den Base Pointer übernommen, und oberhalb des dynamischen Bereichs des rufenden Programms wird der Stack Frame des gerufenen aufgebaut. Spitzfindigkeiten erläuterm wir im folgenden anhand von UNIX und der Architektur IA-32.

Grundlagen der systemseitigen Speicherverwaltung
Die Speicherverwaltung hat die Aufgabe, den einzelnen Programmen im Arbeitsspeicher eine angemessene Speicherkapazität zur Verfügung zu stellen. Wieviel Speicher (z. B. in Bytes ausgedrückt) braucht aber ein Programm? – Es sind unterzubringen:

- das Programm selbst,
- die zugehörigen konstanten Daten,
- Arbeits- und Übergabebereiche,
- bedarfsweise Symbol- und Verweistabellen.

In einfachen Systemen kann man die verfügbare Speicherkapazität fest aufteilen (statische Speicheraufteilung). Moderne Hochleistungssysteme sind hingegen dadurch gekennzeichnet, daß sich die Speicherbelegung ständig ändert (dynamische Speicheraufteilung). Abb. 18 veranschaulicht ein Prinzip, das häufig implementiert wird.

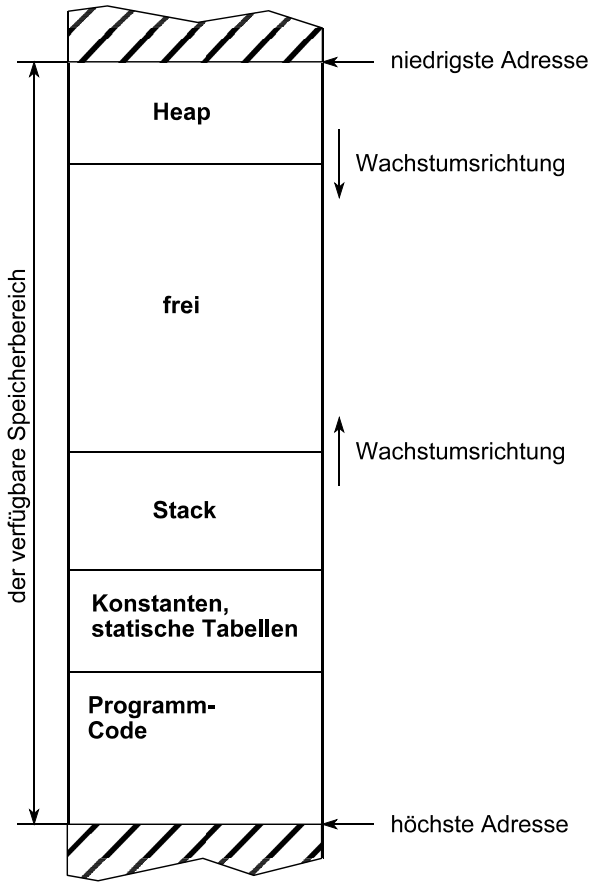


Abb. 18 Zum Prinzip der Speicheraufteilung (Beispiel)

Wir beginnen damit, daß ein “hinreichend” großer Speicherbereich zunächst bereitsteht. Dieser wird folgendermaßen belegt.

- der Programmcode an sich wird ganz hinten untergebracht,
- davor kommen die “statischen” – in ihrer Größe unveränderlichen – Datenbereiche (Konstanten, Symboltabellen usw.),
- im Anschluß daran – zu den niederen Adressen hin – wird der Stack eingerichtet. Er nimmt dynamische Daten, Parameter, statische Variable, Zwischenergebnisse und Rückkehradressen auf. Er wächst in Richtung niederer Adressen.
- ergänzend zum Stack sieht man oft eine weitere veränderliche Struktur vor, den Heap (sprich: Hiep; wörtlich = Haufen). Der Heap wird am Anfang des Speicherbereichs angeordnet. Er wächst in Richtung höherer Adressen. Zur Verwendung von Stack und Heap siehe Tabelle 4.

Sowohl Stack als auch Heap wachsen oder schrumpfen während der Ausführung des Programms. Durch die Anordnung an entgegengesetzten Enden ist stets gewährleistet, daß sich ein möglichst großer freier Bereich zwischen Stack und Heap befindet. Nur in dem – vergleichsweise unwahrscheinlichen – Fall, daß beide Strukturen wachsen und wachsen, kann es vorkommen, daß irgendwann einmal nichts mehr frei ist, daß also der Stack versucht, ein Stück des Heap zu belegen oder umgekehrt. Die Schutzvorkehrungen der Hardware bzw. das Laufzeitsystem der Software sollten dies erkennen und entsprechend reagieren (z. B. mit dem Abbruch der Programmausführung und einer entsprechenden Fehlermeldung).

	Stack	Heap
Nutzung (gespeichert werden...)	Rückkehradressen, lokale Daten (verschwinden bei Rückkehr aus der jeweiligen Funktion)	dynamische Daten (bleiben solange erhalten, bis sie explizit (vom Programm) wieder freigegeben werden
Belegung und Freigabe (Auf- und Abbau)	automatisch gemäß dem LIFO-Prinzip	typischerweise (vgl. Programmier-sprache C) vom Programmierer anzufordern und freizugeben
besondere Eignung	für kleinere und einfachere Datenstrukturen (zu beispielsweise 32 oder 64 Bits)	für größere und kompliziertere Datenstrukturen (z. B. von 256 Bytes an aufwärts)

Tabelle 4 Zur Verwendung von Stack und Heap

Die UNIX-Stackorganisation
Für jeden Prozeß werden zwei Stacks verwaltet (Abb. 19):

- der User Stack zum Aufrufen von Anwendungsprogrammen,
- der Kernel Stack zum Aufrufen der Systemfunktionen.

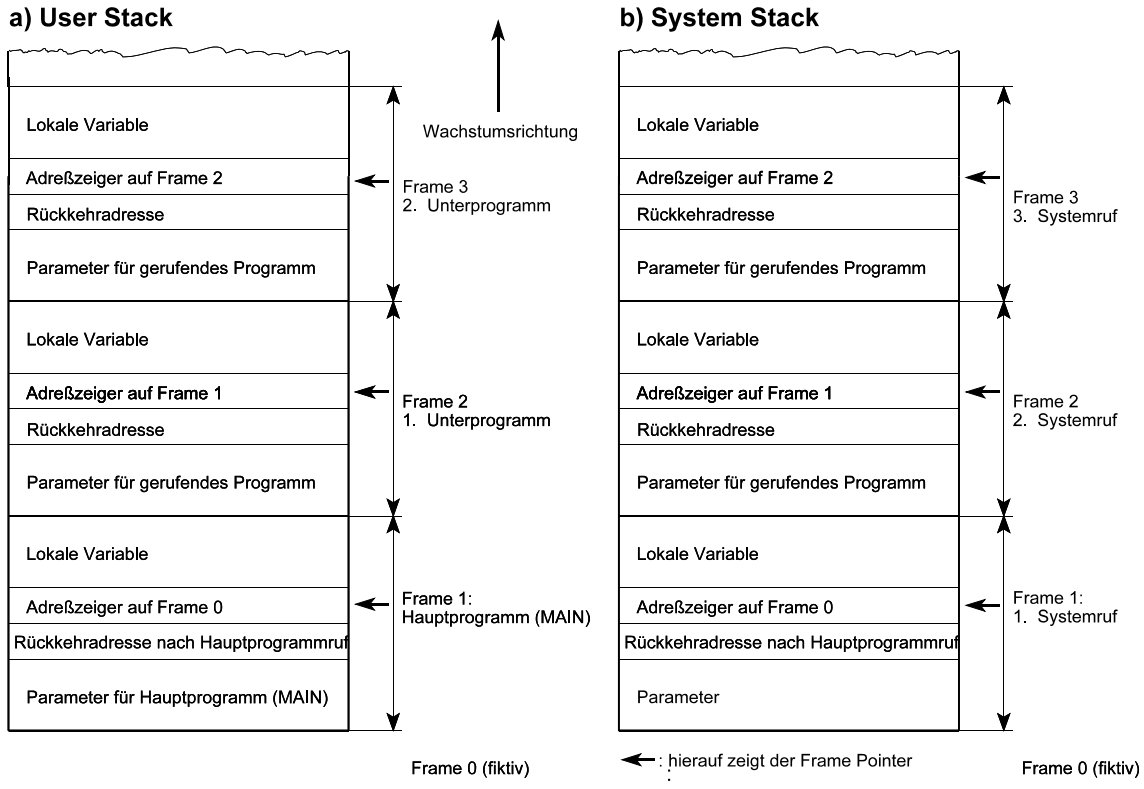


Abb. 19 Die UNIX-Stackorganisation

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

1. das rufenden Programm legt die zu übergebenden Parameter auf den Stack. Aus Tabelle 5 sind typische Konventionen des Unterprogrammrufs ersichtlich.
2. der Aufruf wird ausgeführt. Dabei gelangt die Rückkehradresse auf den Stack.
3. das gerufenene Programm kopiert den bisherigen Frame Pointer auf den Stack (Adreßzeiger als Rückverweis). Typischerweise wird der aktuelle Inhalt des Stackpointers zum neuen Frame Pointer[5].
4. das gerufene Programm kopiert seine lokalen Variablen in den Stack (bzw. schafft auf dem Stack soviel Platz, daß die lokalen Variablen hineinpassen),
5. der aktuelle Frame bzw. Base Pointer wird eingerichtet.

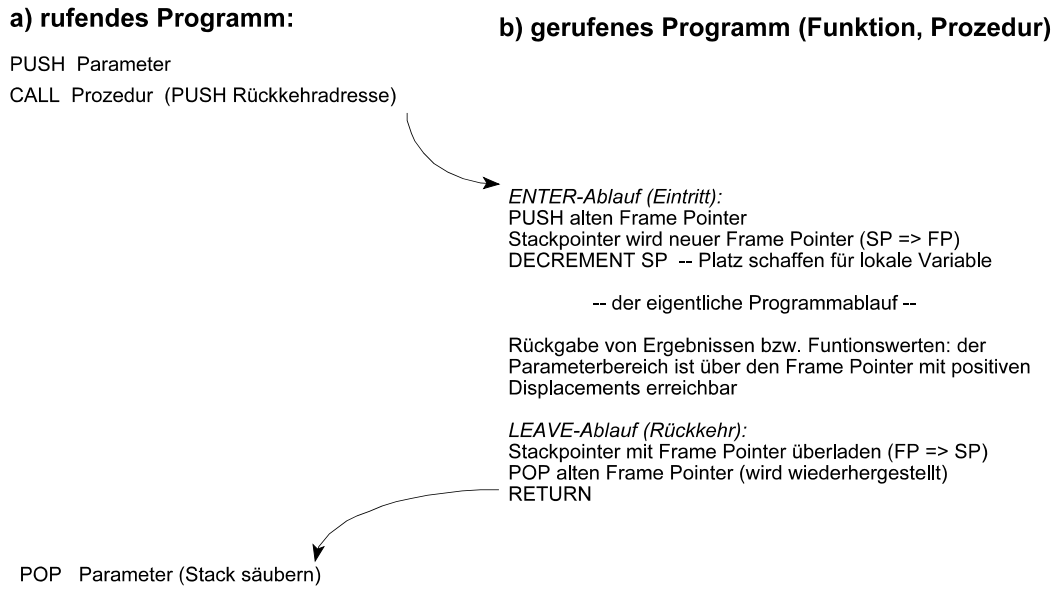


Abb. 20 Unterprogrammaufruf in einer Laufzeitumgebung, die auf Stack Frames beruht.

	Pascal	**C**
Reihenfolge der Parameterübergabe	von links nach rechts	von rechts nach links
wer stellt bei der Rückkehr die ursprüngliche Stackbelegung wieder her (Stack Cleanup)?	das gerufene Programm	das rufende Programm
Vor- und Nachteile der Stack-Cleanup-Konvention	• Cleanup-Ablauf nur einmal vorhanden (im gerufenden Programm), • Funktionsaufrufe typischerweise nur mit fester Parameteranzahl	• Cleanup-Ablauf in jedem rufenden Programm erforderlich, • erster Parameter (ganz links im Funktionsaufruf) kommt stets auf TOS zu liegen (erleichtert Implementierung von Funktionsaufrufen mit variabler Parameteranzahl)

Tabelle 5 Typische Konventionen des Unterprogrammrufs

5 Dieser Ablauf wird gelegentlich von der Hardware unterstützt (z. B. IA-32-ENTER-Befehl).

Die Stackbelegung anhand eines Beispiels

Deklaration einer Funktion:

```
int MAUSI (int A, int B, double C);

{
int X, Y;
double Z;
float H, I;
...

....

return (Y);

}
```

Jetzt wird die Funktion aufgerufen:

```
...

OMEGA = MAUSI (ALPHA, BETA, GAMMA);

...
```

Zunächst werden die Parameter auf den Stack gelegt, dann wird die Funktion aufgerufen:

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

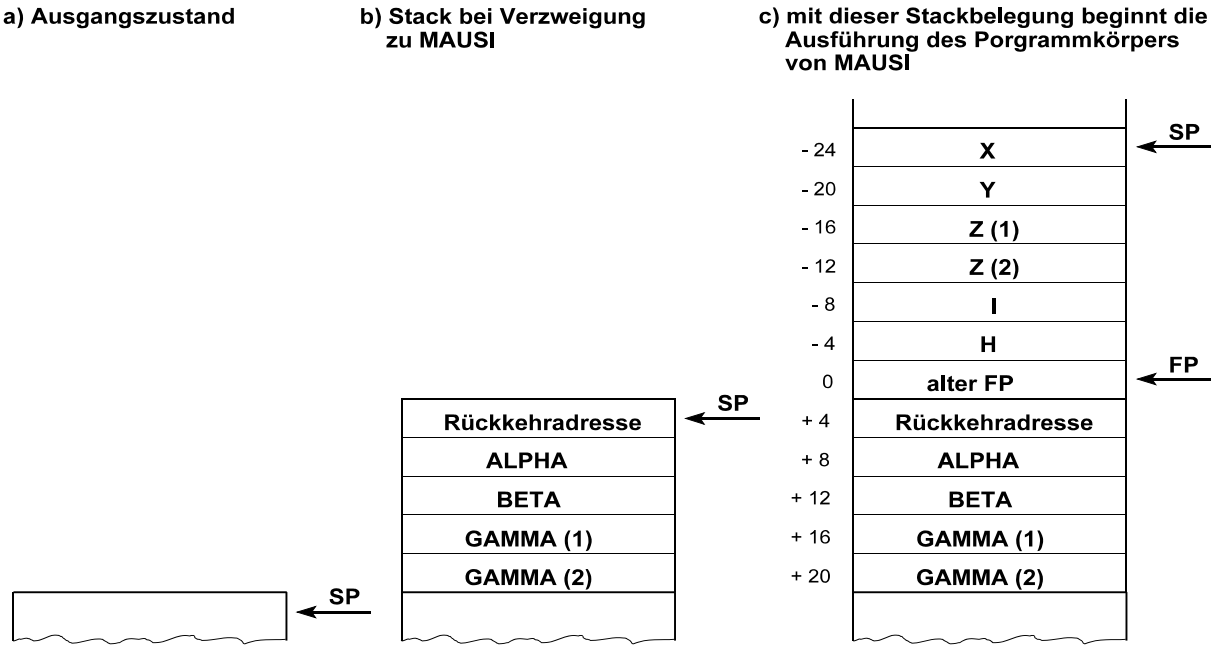
Eintritt in die Funktion. Es wird zunächst der Eintrittscode ausgeführt (ENTER-Ablauf):

```
MAUSI:  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 MAUSI ....

        Rückkehr (LEAVE-Ablauf):

        MOV Y, (FP + 20)  -- Rückgabewert in Stack schreiben
        MOV FP, SP        -- Zurückstellen des Stackpointers
        POP FP            -- alten FP aus Stack zurückholen
        RETURN
```

Weiter mit dem rufenden Programm:

```
POP_DOUBLE          -- Stack freimachen
POP
POP OMEGA           -- mit dem letzten POP wird der Rückgaberwert zugewiesen
```

Hinweis zur Ergebnisrückgabe:
Es ist nichts standardisiert. Die hier gezeigte Rückgabe über den Stack ist nur ein Beispiel. Eine typische Praxislösung ist die Übergabe in einem bestimmten Register (im PC: zmeist EAX).



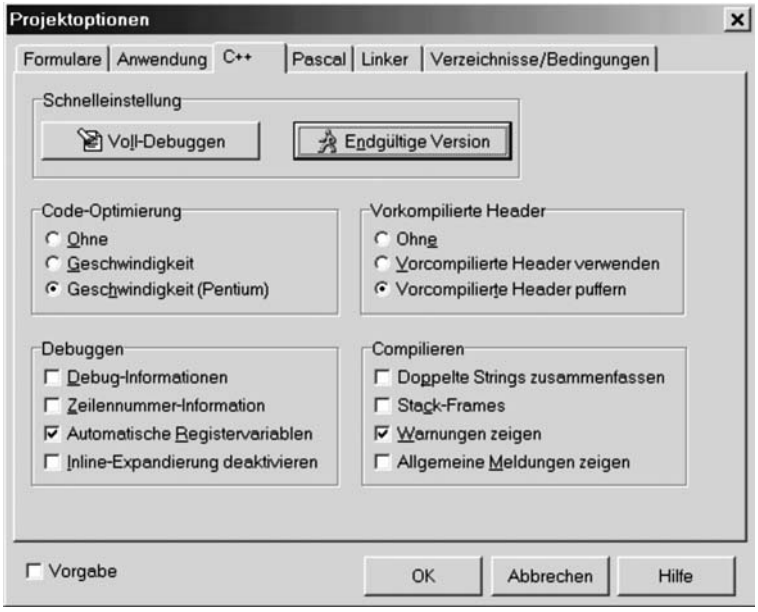
Programmoptimierung:

- kurze Funktionen nicht rufen, sondern als Inline-Code einfügen,
- Funktionen ohne Parameter und lokale Variable werden nach einem verkürzten Verfahren aufgerufen. Also: nicht nach der reinen Lehre, sondern auf Leistung programmieren...

Zum Fehlersuchen (Debugging) müssen diese Optimierungen ggf. ausgeschaltet werden (damit man im Speicherausdruck (Memory Dump) erkennen kann, was eigentlich losgewesen ist):

- kein Inline-Code,
- es wird stets ein richtiger Stack Frame erzeugt.

Beispiel der Steuerung eines C-Compilers:



Homogene Datenstrukturen (Arrays)
Eine homogene bzw. Array-Struktur besteht aus gleichartigen Elementen, auf die typischerweise in Schleifen zugegriffen wird (z. B. in FOR-Anweisungen). Jeder derartige Zugriff erfordert eine Adreßrechnung.

Wenn das einzelne Element einer eindimensionalen Feldstruktur (Datentyp *Array*) aus b Bytes besteht, so errechnet sich die Offsetadresse des n-ten Elementes (n = 1, 2,..), auf den Feldanfang bezogen, gemäß Offset = (n-1) ⊛ b. n ist die laufende Nummer (Ordinalzahl) des Elements (Elementindex).

Zwei Konventionen:

a) das erste Element hat den Index 0 (z. B. in C),

b) das erste Lement hat den Index 1 (z. B. in BASIC wählbar (OPTION BASE)).

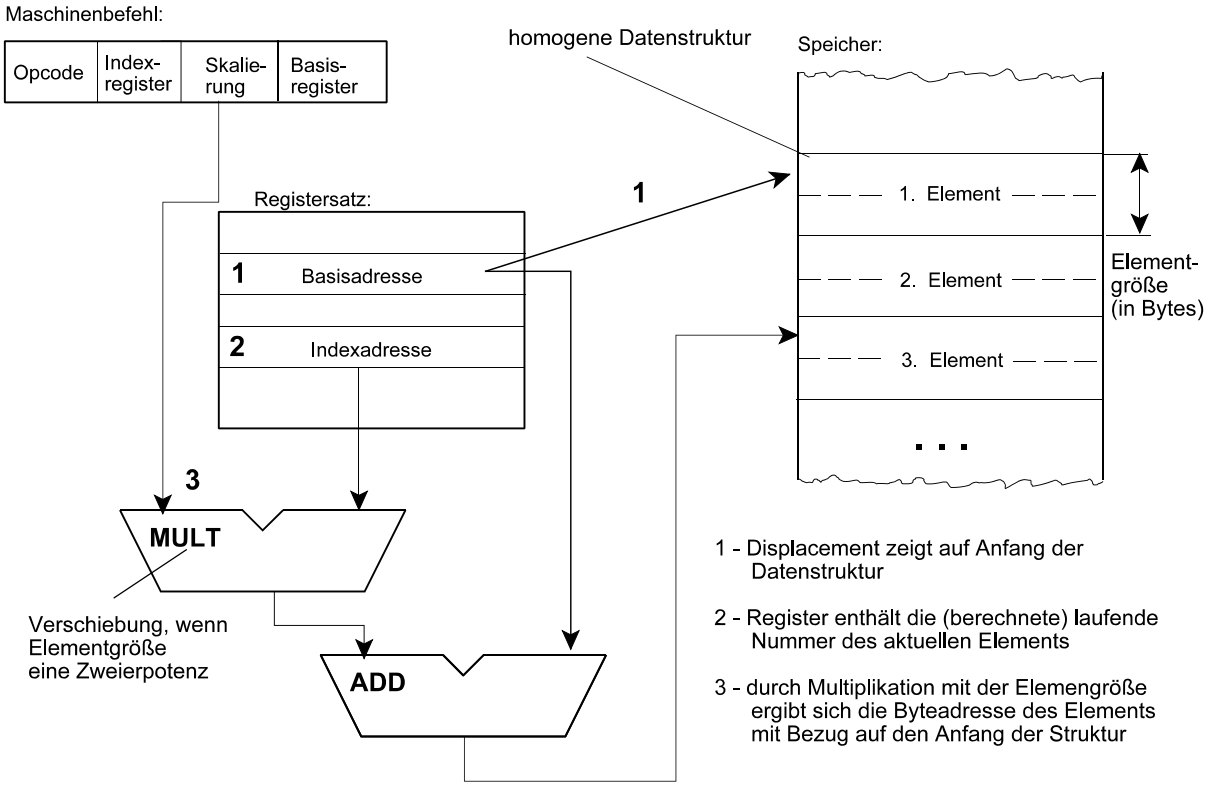
Ist die Elementgröße b eine Zweierpotenz, so kann die Multiplikation durch eine Linksverschiebung um ld b Bits ersetzt werden.

Manche Architekturen haben eigens Vorkehrungen für bestimmte häuifig gebrauchte Elementgrößen.

Beispiel IA-32: Basis + (Index ⊛ Skalierung) + Displacement.
(Einzelne Komponenten dieses Adressierungsschemas können weggelassen werden. Hierdurch ergeben sich insgesamt 9 Adressierungsarten.

Index = laufende Nummer des Feldelements (von 0 an).

Skalierung = Elementgröße. Unterstützt werden 2, 4 oder 8 Bytes.



Zugriff auf zweidimensionale Arrays (1). Zeilenweise Anordnung.
z = Zeilenzahl, s = Spaltenzahl, b = Elementgröße.

x = Zeilenindex, y = Spaltenindex.

$$Offset_0 = (x \cdot s + y) \cdot b$$

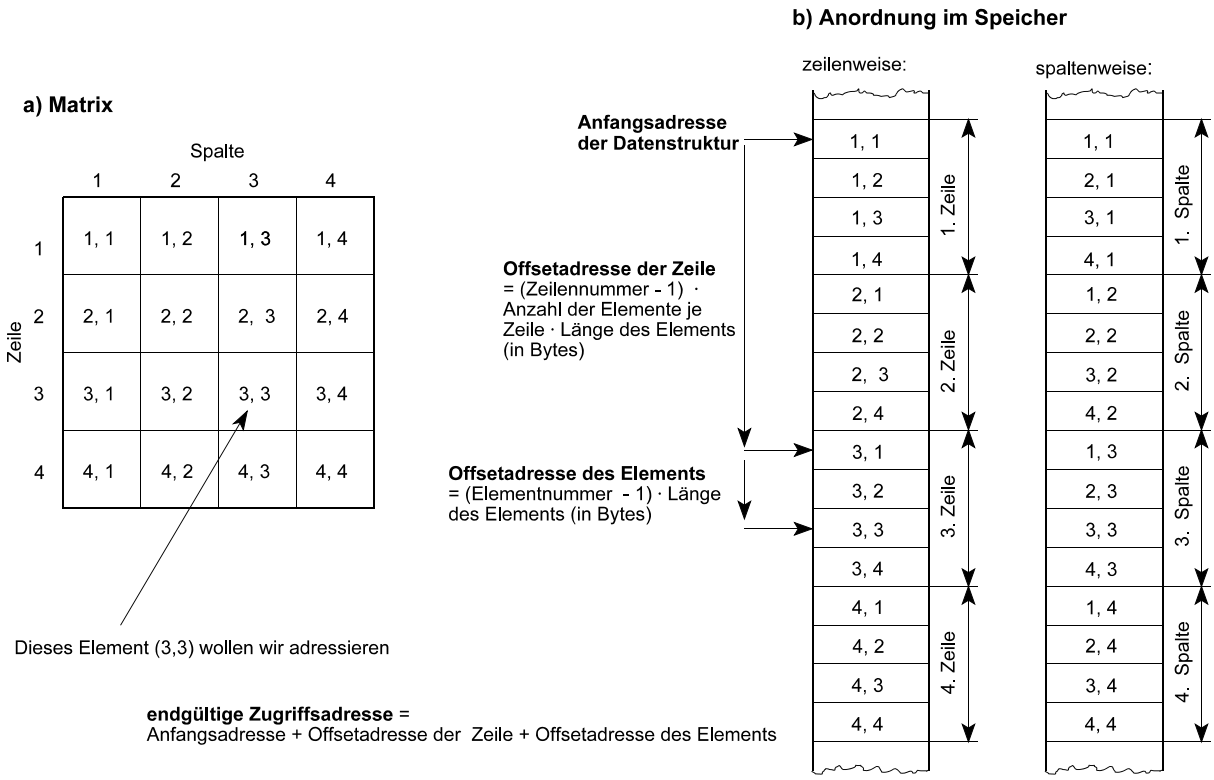
$$Offset_1 = ((x-1) \cdot s + y - 1) \cdot b$$

Zugriff auf zweidimensionale Arrays (2). Spaltenweise Anordnung.
z = Zeilenzahl, s = Spaltenzahl, b = Elementgröße.

x = Zeilenindex, y = Spaltenindex.

$$Offset_0 = (x + y \cdot z) \cdot b$$

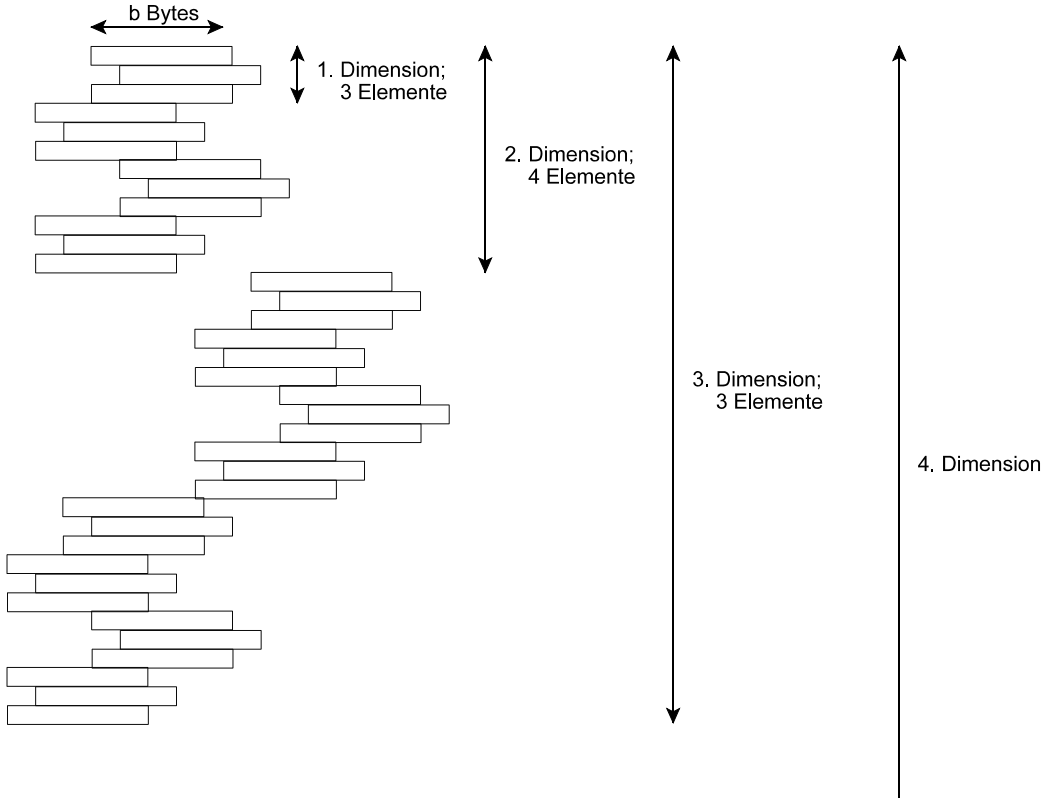
$$Offset_1 = (x - 1 + (y-1) \cdot s) \cdot b$$



Die allgemeine Form der Adreßrechnung:
Der Adressen-Offset (bezogen auf den Feldanfang) für ein Element eines mehrdimensionalen Feldes aus gleichartigen Elementen, die b Bytes lang sind, ist zu berechnen. Die gegebenen Feldkoordinaten seien k_1, k_2, ... ; die Anzahl der Elemente in jeder Felddimension sei C_1, C_2, ... Dimension 1 sei die innerste Dimension. Es werden jeweils C1 Elemente aufeinanderfolgend angeordnet usw. Um die Formeln nicht allzu unübersichtlich zu machen, wird bei den Anzahlen C_1, C_2 usw. von 1 an gezählt und bei den Indicies k_1, k_2 usw. von 0 an.

$$\mathrm{Offset}_0 = \left(k_1 + k_2 \cdot C_1 + k_3 \cdot C_1 \cdot C_2 + k_4 \cdot C_1 \cdot C_2 \cdot C_3 + \ldots + k_n \cdot C_1 \cdot C_2 \cdot \ldots \cdot C_{n-1}\right) \cdot b$$

$$\mathrm{Offset}_0 = \left(k_1 + C_1 \cdot (k_2 + C_2 \cdot (k_3 + C_3 \cdot (k_4 \ldots)))\right) \cdot b$$



Bei wahlfreien Zugriffen geht es nicht ohne Multiplikation.

Erste Abhilfe: Längen und Dimensionen als Zweierpotenzen oder als Vielfache kleiner Zweierpotenzen (z. B. 4 oder 8). Ggf. den Mehrverbrauch an Speicherplatz in Kauf nehmen.

Zweite Abhilfe: Mehrdimensionale Arrays in eindimensionale aufrollen und Adreßrechnung zu Fuß programmieren (ggf. in Assembler).

Meistens ist es nicht ganz so schlimm, denn beim Rechnen mit Arrays verwendet man typischerweise Schleifen, und es ergeben sich überschaubre Zugriffsmuster.

Wichtige Zugriffsmuster des numerischen Rechnens:

- eine Zeile einer Matrix
- eine Spalte einer Matrix
- die Hauptdiagonale einer quadratischen Matrix
- die obere Dreiecksmatrix einer quadratischen Matrix
- die untere Dreiecksmatrix einer quadratischen Matrix
- die geraden Elemente eines Vektors
- die ungeraden Elemente eines Vektors
- die transponierte Matrix
- die Oberfläche eines Würfels
- eine Matrix, die durch Extraktion der ungeraden Elemente der ungeraden Zeilen einer Matrix gebildet wird
- Untermatrizen

Fraglich ist nur, ob ein C-Compiler derartige Zugriffsmuster erkennen kann...

Bessere Sprachen nehmen (die sowas unterstützen) oder entsprechende Packages beschaffen (die intern optimiert sind). Zu Fuß: keine mehrdimensionalen Arrays und entsprechend geschachtelte Schleifen,

sondern Adressierung über Pointer (die oft mit einfachen Laufanweisungen berechnet werden können, z. B. durch Addieren von Abstandswerten).

Der grundsätzliche Vorteil des mehrdimensiuonalen Arrays: Es ist oftnals eine 1:1-Entsprechung der jeweiligen mathematischen Strukturen. Die jeweiligen Formeln können so oftmals 1:1 abprogrammiert werden (z. B. Matrizenrechnung, Computergraphik).

Zeichenketten
Die Zeichenkette ist ein eindimensionales Array, dessen Elemente Zeichen sind.

Das Hauptproblem: die Lämgenangabe.

Grundsatzlösungen:

a) Endekennung

   1. im Byte. z. B. 7-Bit-ASCII. 8. Bit = 1 kennzeichnet Ende.

   2. besondere Zeichen als Endemarke. In C: 00H.

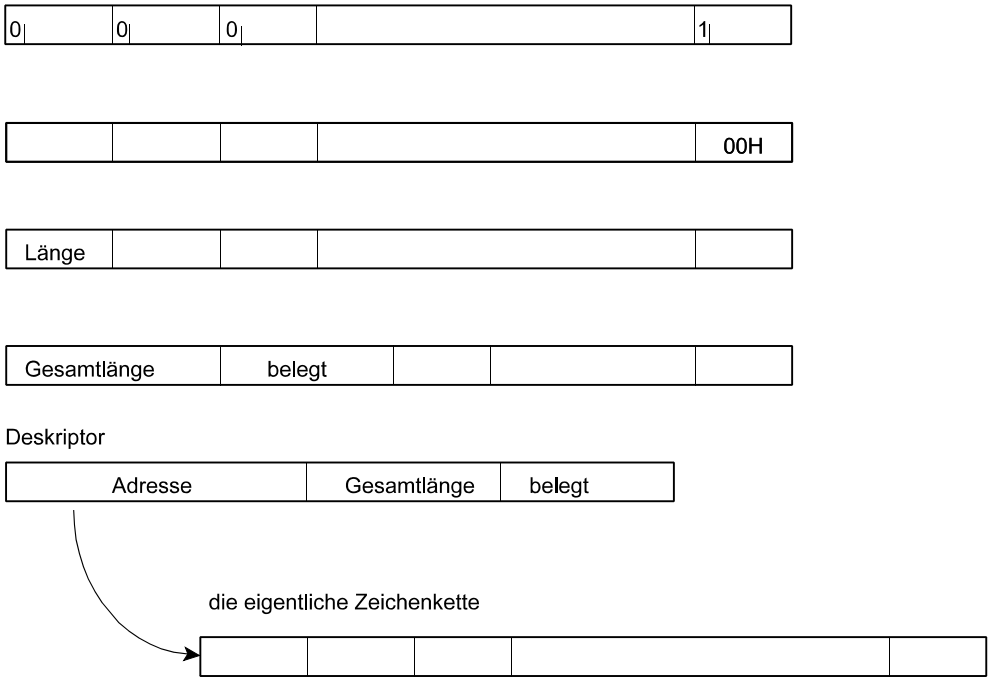
b) Längenangabe

   1. das erste Byte gibt die Länge an (Pascal)

   2. die ersten zwei oder vier Bytes geben die Länge an (z. B. Delphi)

   3. es gibt zwei Längenwerte: Die Gesamtlänge und die aktuell belegte Länge

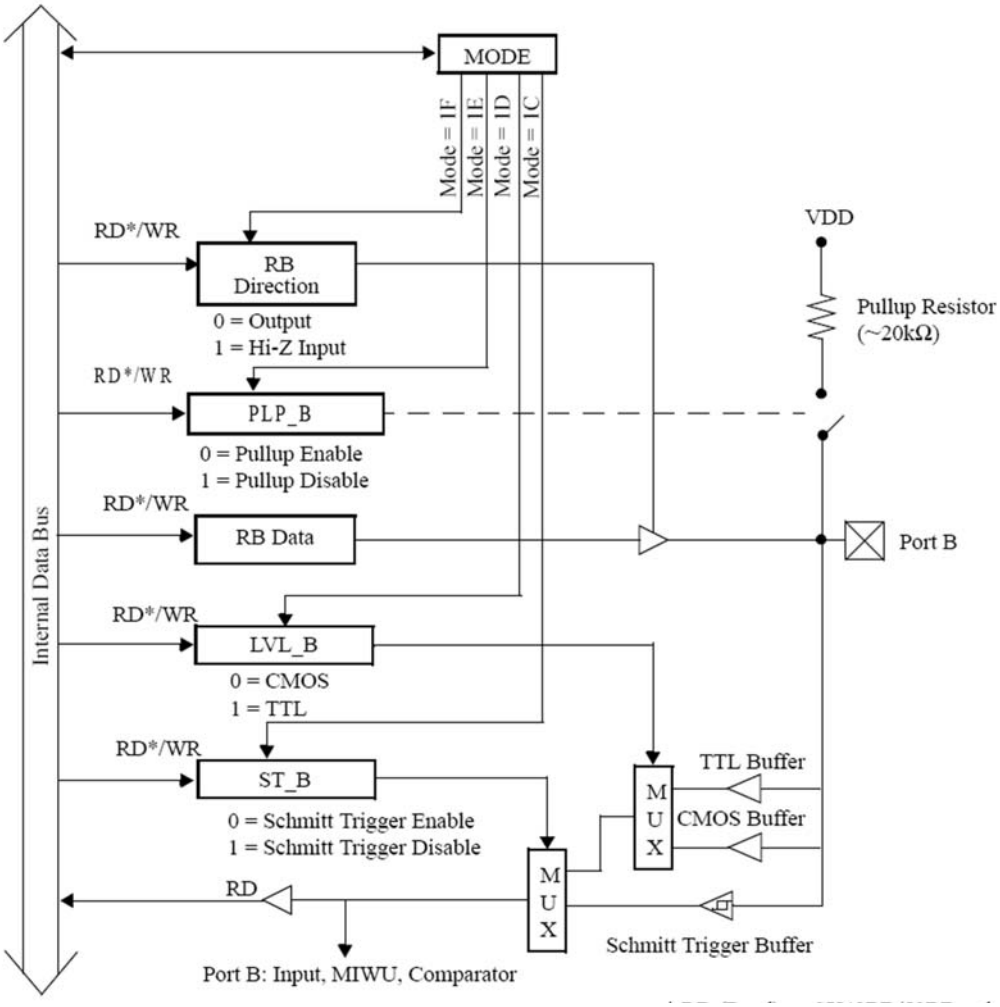
   4. Längenangabe in gesonderten Deskriptoren



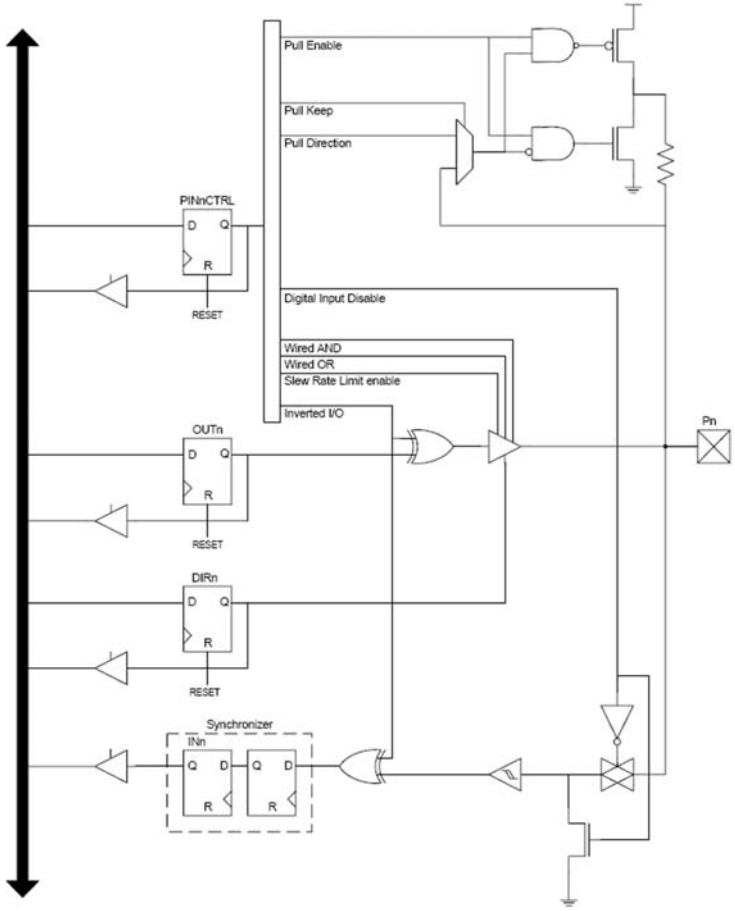
Advanced I/O Ports

– Nur zur Information –

(Kommt nicht dran.)

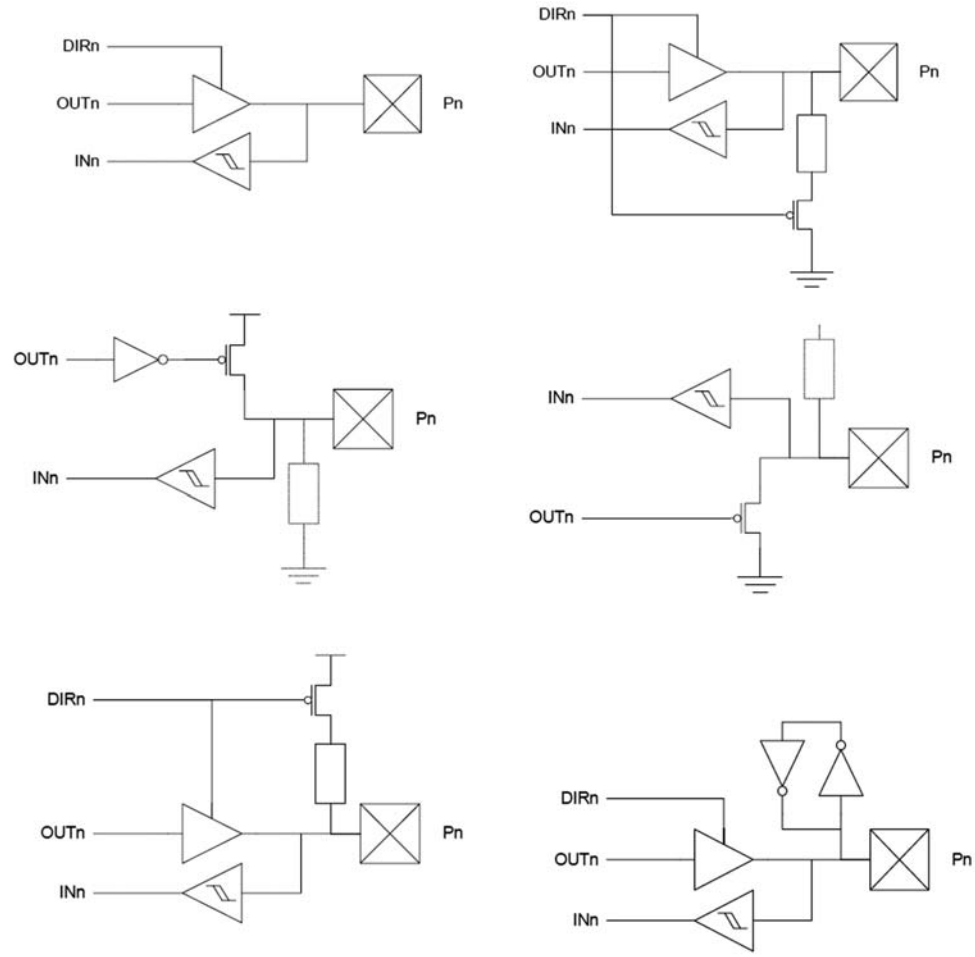


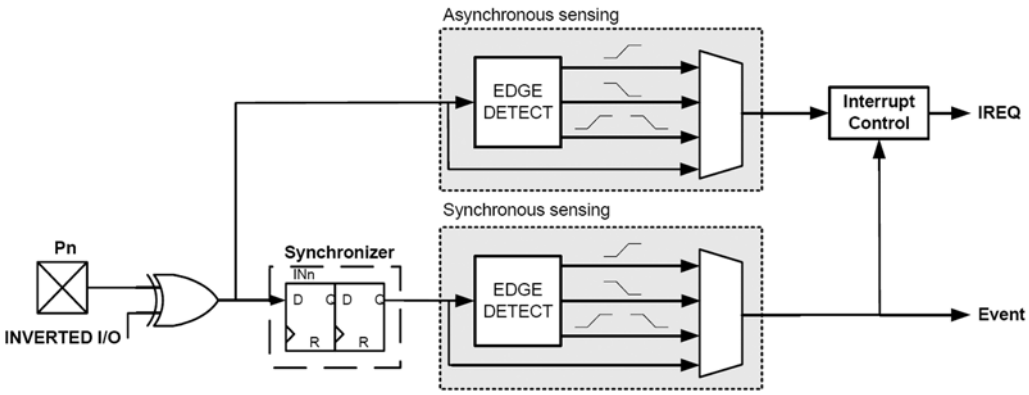
Bit	7	6	5	4	3	2	1	0	
	P7	P6	P5	P4	P3	P2	P1	P0	Port 0 Data Register (PDR00)
	PS7	PS6	PS5	PS4	PS3	PS2	PS1	PS0	Port 0 External Pin State Register (EPSR00)
	D7	D6	D5	D4	D3	D2	D1	D0	Port 0 Direction Register (DDR00)
	IE7	IE6	IE5	IE4	IE3	IE2	IE1	IE0	Port 0 Input Enable Register (PIER00)
	IL7	IL6	IL5	IL4	IL3	IL2	IL1	IL0	Port 0 Input level Register (PILR00)
	EIL7	EIL6	EIL5	EIL4	EIL3	EIL2	EIL1	EIL0	Port 0 Extended Input Level Register (EPILR00)
	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0	Port 0 Output Drive Register (PODR00)
	HD7	HD6	HD5	HD4	HD3	HD2	HD1	HD0	Port 0 High Drive Register (PHDR00)
	PU7	PU6	PU5	PU4	PU3	PU2	PU1	PU0	Port 0 Pull-Up Control Register (PUCR00)



12.17 Register Summary – Ports

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
+0x00	DIR	DIR[7:0]								123
+0x01	DIRSET	DIRSET[7:0]								123
+0x02	DIRCLR	DIRCLR[7:0]								124
+0x03	DIRTGL	DIRTGL[7:0]								124
+0x04	OUT	OUT[7:0]								124
+0x05	OUTSET	OUTSET[7:0]								125
+0x06	OUTCLR	OUTCLR[7:0]								125
+0x07	OUTTGL	OUTTGL[7:0]								125
+0x08	IN	IN[7:0]								125
+0x09	INTLVL	-	-	-	-	INT1LVL[1:0]		INT0LVL[1:0]		126
+0x0A	INT0MASK	INT0MSK[7:0]								126
+0x0B	INT1MASK	INT1MSK[7:0]								126
+0x0C	INTFLAGS	-	-	-	-	-	-	INT1IF	INT0IF	127
+0x0D	Reserved	-	-	-	-	-	-	-	-	
+0x0E	Reserved	-	-	-	-	-	-	-	-	
+0x0F	Reserved	-	-	-	-	-	-	-	-	
+0x10	PIN0CTRL	SRLEN	INVEN		OPC[2:0]			ISC[2:0]		127
+0x11	PIN1CTRL	SRLEN	INVEN		OPC[2:0]			ISC[2:0]		127
+0x12	PIN2CTRL	SRLEN	INVEN		OPC[2:0]			ISC[2:0]		127
+0x13	PIN3CTRL	SRLEN	INVEN		OPC[2:0]			ISC[2:0]		127
+0x14	PIN4CTRL	SRLEN	INVEN		OPC[2:0]			ISC[2:0]		127
+0x15	PIN5CTRL	SRLEN	INVEN		OPC[2:0]			ISC[2:0]		127
+0x16	PIN6CTRL	SRLEN	INVEN		OPC[2:0]			ISC[2:0]		127
+0x17	PIN7CTRL	SRLEN	INVEN		OPC[2:0]			ISC[2:0]		127
+0x18	Reserved	-	-	-	-	-	-	-	-	
+0x19	Reserved	-	-	-	-	-	-	-	-	
+0x1A	Reserved	-	-	-	-	-	-	-	-	
+0x1B	Reserved	-	-	-	-	-	-	-	-	
+0x1C	Reserved	-	-	-	-	-	-	-	-	
+0x1D	Reserved	-	-	-	-	-	-	-	-	
+0x1E	Reserved	-	-	-	-	-	-	-	-	
+0x1F	Reserved	-	-	-	-	-	-	-	-	


DIRn
OUTn
INn
Pn
DIRn
OUTn
INn
Pn
OUTn
INn
Pn
INn
OUTn
Pn
DIRn
OUTn
INn
Pn
DIRn
OUTn
INn
Pn


Asynchronous sensing
EDGE DETECT
Interrupt Control
IREQ
Synchronous sensing
Pn
INVERTED I/O
Synchronizer
INn
D
D
R
R
EDGE DETECT
Event