Heft 1
Datenstrukturen - Maschinenbefehle - Speicheradressierung

- Grundlagen der Rechnerarchitektur, Kapitel 1 und 2 -

Stand: 1.2

- Mit Abbildungen -

1. Datenstrukturen

1.1. Adressierbare Behälter

Die einfachsten Datenstrukturen sind lediglich Aneinanderreihungen von Bits, die gemeinsam adressierbar sind. Sie sind gleichsam Behälter, die eine feste Größe haben und in diesem Rahmen an sich beliebige Angaben aufnehmen können. Eine bestimmte Bedeutung erhält die so verpackte Information nur dann, wenn Befehle oder andere in der Architektur definierte Funktionsabläufe auf sie angewendet werden*). In nahezu allen Architekturen von irgendwelcher Bedeutung am heutigen Markt sind vier derartige Strukturen vorgesehen, die 8, 16, 32 und 64 Bits lang sind. In diesem Zusammenhang sind zwei Begriffe von Bedeutung: das *Byte* und das *Wort*. Ein Byte ist grundsätzlich (in allen modernen Architekturen) die Aneinanderreihung von 8 Bits. Als Wort bezeichnet man üblicherweise die Datenstruktur, deren Bitanzahl (bzw. Länge) der Verarbeitungsbreite entspricht.

*): diese Tatsache bereitet Anfängern gelegentlich Schwierigkeiten. Sie sollten sich deshalb eines wirklich klarmachen: eine Aneinanderreihung von beispielsweise 32 Bits kann eine natürliche (vorzeichenlose) Zahl sein, eine ganze (vorzeichenbehaftete) Zahl, eine Gleitkommazahl, eine Folge von zwei oder vier Textzeichen, eine 8-stellige Dezimalzahl, eine Folge von zwei 16-Bit-Zahlen oder auch eine wahlfrei aus Bitfeldern verschiedener Länge zusammengesetzte Struktur. Den 32 Bits an sich können wir nicht ansehen, was sie eigentlich bedeuten; deren Bedeutung ergibt sich vielmehr erst aus dem aktiven Gebrauch in der Maschine, aus der Nutzung dieser 32 Bits während der Verarbeitungsabläufe (mit anderen Worten daraus, welche Maschinenbefehle auf sie angewendet werden).

1.1.1. Verarbeitungsbreite

Die Verarbeitungsbreite gibt jene Anzahl an Bits an, die im Prozessor als *ein* Operand an Transport- oder Verknüpfungsoperationen teilnehmen kann. Um dies näher zu veranschaulichen, zeigt Abbildung 1.1 eine Verknüpfungsschaltung (z. B. ein Addierwerk) mit zugeordneten elementaren Speichermitteln (Registern). Können beide Register jeweils 16 Bits aufnehmen, so beträgt die Verarbeitungsbreite 16 Bits. Man spricht dann allgemein von einer 16-Bit-Maschine. Ist die gesamte Architektur gleichsam um diese Datenstruktur herum aufgebaut, so handelt es sich um eine 16-Bit-Architektur. Die Datenstruktur von 16 Bits Länge heißt dann Maschinenwort oder einfach Wort (engl. Word; sprich: Wöhrd).

Beispielsweise bilden in den Architekturen x86 und IA-32 16 Bits ein Wort, 32 Bits ein Doppelwort und 64 Bits ein Quadwort (Vierfachwort). Diese Bezeichnungsweise ist aus der Entwicklungsgeschichte heraus zu erklären: der 8086 war einer der ersten 16-Bit-Mikroprozessoren, und dessen kennzeichnende Informationsstruktur war demzufolge das 16-Bit-*Wort*. Andere Architekturen wurden hingegen von Anfang an für 32 Bits Verarbeitungsbreite ausgelegt. In einem solchen Fall ist natürlicherweise eine Aneinanderreihung von 32 Bits ein *Wort*; 16 Bits bilden ein *Halbwort*, 64 Bits ein *Doppelwort*.

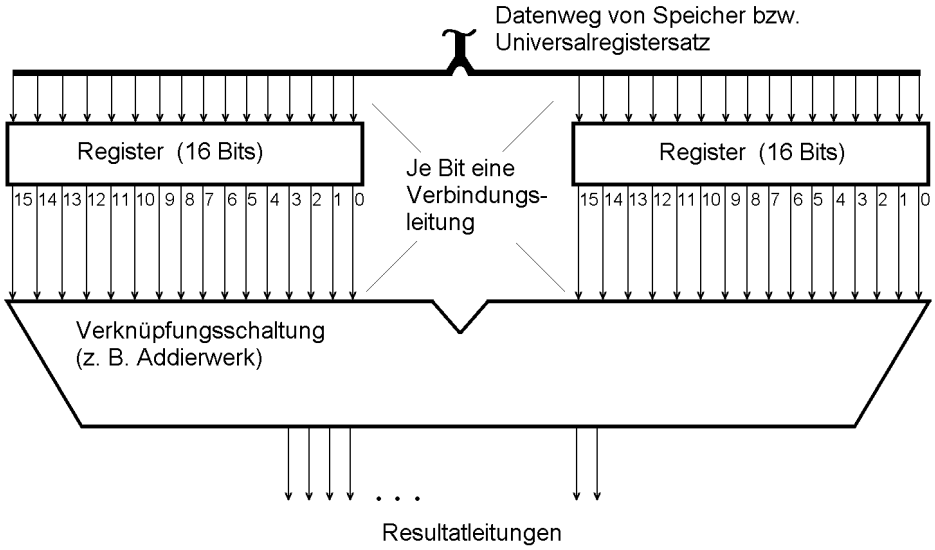


Abbildung 1.1 Zur Erklärung der Verarbeitungsbreite

Wir sprechen hier grundsätzlich nur von der Verarbeitungsbreite, wie sie der Architektur zugrunde liegt. Die Verarbeitungsbreite einer bestimmten Hardware, in der die Architektur implementiert ist (technische Verarbeitungsbreite), kann aber davon abweichen. Beispielsweise wurde die S/360-Architektur (32 Bits) in Hardware-Modellen mit Verarbeitungsbreiten von 4, 8, 16, 32 und 64 Bits implementiert.

1.1.2. Bit- und Byteanordnung (Rechts- und Linksadressierung)

Wenn wir Bytes und andere Datenstrukturen näher betrachten, ist es wichtig, zu wissen, wie die einzelnen Bits numeriert werden und worauf die Adreßangaben zeigen. Leider gibt es auch hier keine Einheitlichkeit. Vielmehr hat man es mit zwei Adressierungs- bzw. Numerierungsweisen zu tun: mit Rechts- und Linksadressierung, wobei es noch Unterschiede in der Adressierung der Bytes und der Durchnumerierung der Bits geben kann. Für letzteres ist auch der Begriff Indizierung in Gebrauch; die einzelne Bit-Nummer heißt dann Bit-Index.

Stellenwert
In Binärzahlen hat jedes Bit einen Stellenwert, genau wie jede Ziffer in einer Dezimalzahl. In diesem Sinne spricht man allgemein (auch bei nichtnumerischen Datenstrukturen) von nieder- und höherwertigen Bits. Zeichnerisch wird das niedrigstwertige Bit (Least Significant Bit, LSB) ganz rechts dargestellt, das höchstwertige Bit (Most Significant Bit, MSB) hingegen ganz links, in völliger Entsprechung zur üblichen Zahlenschreibweise.

Rechtsadressierung
Die Anfangsadresse einer Informationsstruktur zeigt immer auf deren niedrigstwertiges Byte.

Rechtsindizierung
Das niedrigstwertige Bit hat den Bit-Index 0, das höchstwertige Bit einer n Bits langen Informationsstruktur hat den Index (n-1), im Byte also den Index 7, im 32-Bit-Wort den Index 31 usw.

Linksadressierung
Die Anfangsadresse einer Informationsstruktur zeigt immer auf deren höchstwertiges Byte.

Linksindizierung
Das höchstwertige Bit hat den Bit-Index 0, das niedrigstwertige Bit einer n Bits langen Informationsstruktur hat den Index (n-1), im Byte also den Index 7, im 32-Bit-Wort den Index 31 usw.

Little Endian, Big Endian
Das sind die üblichen Fachbegriffe:

- Little Endian = Rechtsadressierung (sprich: Littl Enndiänn),
- Big Endian = Linksadressierung (sprich: Bigg Enndiänn).

Beispiele (Abbildung 1.2):

- Little Endian: x86 und IA-32 sowie viele andere Mikroprozessoren,
- Big Endian: die IBM-Mainframes (S/360.../390), PowerPC, JVM (Java Virtual Machine).

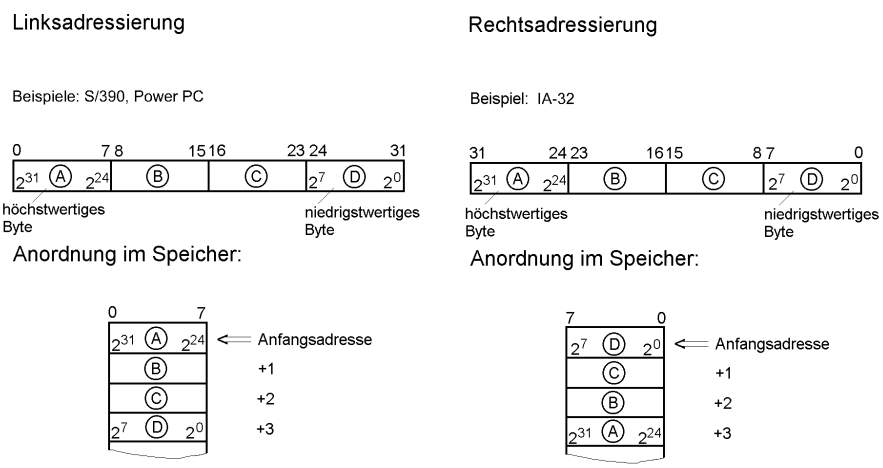


Abbildung 1.2 Links- und Rechtsadressierung am Beispiel von 32-Bit-Worten

Umschaltbare Adressierung
Die meisten neueren Architekturen können für beide Adressierungsweisen konfiguriert werden (z. B. PowerPC, Mips und IA-64).

Achtung:
Im Fall des Falles - die Dokumentation genau lesen! Die Adressierungsweise wird zwar umgeschaltet, die Numerierung der Bits entspricht aber der jeweiligen Vorzugsauslegung (z. B. PowerPC: Big Endian, Mips und IA-64: Little Endian; Abbildung 1.3).

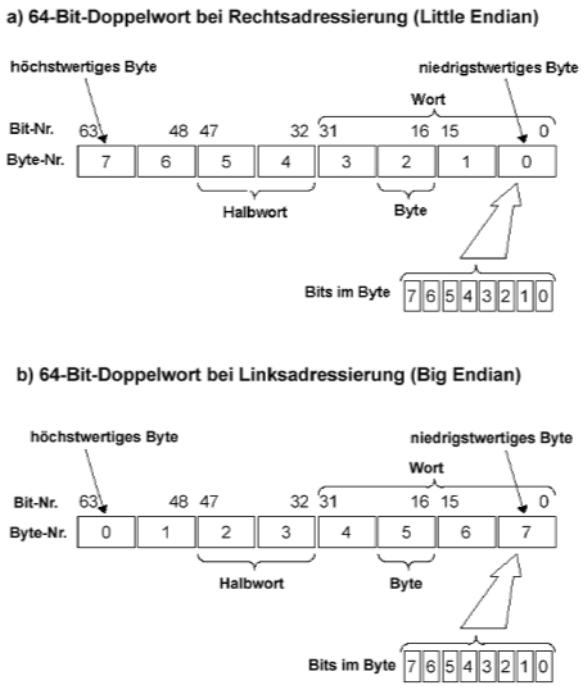


Abbildung 1.3 Umschaltbare Adressierung (Auszug aus einem Architekturhandbuch (NEC))

Vorsicht, Falle:
Beachten Sie die Spitzfindigkeiten sorgfältig, wenn Prozessoren verschiedener Typen gekoppelt werden, wenn es um den Datenaustausch zwischen verschiedenen Systemen geht oder wenn Datenbestände von einer Architektur auf eine andere übertragen werden sollen*). (Denken Sie daran, wenn Sie bei solchen Kopplungsversuchen hinzugezogen werden, weil "unerklärliche" Fehler auftreten.)

*): ein typischer Kopplungsfall: 68k- oder PowerPC- und IA-32-Prozessoren im selben System (gekoppelt über VME-Bus, CompactPCI o. dergl.).

Vor- und Nachteile
An sich ist die Wahl gleichgültig. Die Entscheidung hing aber in der Vergangenheit typischerweise davon ab, wofür die Maschinen bevorzugt eingesetzt, das heißt, welche Datenstrukturen vorzugsweise gespeichert und verarbeitet werden sollten. Rechtsadressierung und -indizierung ist vorteilhaft, wenn es sich vorwiegend um binär codierte Angaben handelt (Bit-Index bzw. Adresse entsprechen direkt der binären Wertigkeit, z. B. Bit-Index $0 \triangleq 2^0$, Bit-Index $1 \triangleq 2^1$ usw.). Die Adressierung gewissermaßen "von hinten" ist aber nicht

sehr anschaulich und entspricht nicht der gleichsam natürlichen Adressierungsweise von Zeichenketten und von Ziffernfolgen variabler Länge. In der Vergangenheit wurden deshalb Systeme, die überwiegend für technisch orientierte Anwendungen gedacht waren (wie DEC VAX und die meisten Mikroprozessoren), mit Rechtsadressierung ausgelegt, vorwiegend für kommerzielle Anwendungen bestimmte Systeme (wie S/360 usw.) hingegen mit Linksadressierung. Heutzutage ist mit beiden Adressierungsweisen zu rechnen (Stichwort: Kompatibilität).

Hinweis:
Das Problem wurde immerhin als so bedeutsam eingeschätzt, daß die Hersteller entsprechende Lösungen eingeführt haben. Beispielsweise wurde vom 486 an ein Byteaustauschbefehl (Byte Swap) vorgesehen, mit dem man die Byteanordnung im Doppelwort umdrehen kann. Viele SBCs (Single Board Computers) zum Einsatz in industriellen Bussystemen (z. B. VME-Bus) haben sogar eine Byteaustausch-Hardware in den Buskoppelschaltungen (so daß die Byteanordnung während der Buszugriffe automatisch umgestellt wird).

Wie werden Datenstrukturen im Lehrmaterial dargestellt?
Im PC-Bereich ist - ausgehend von der x86-Architektur - die Rechtsadressierung üblich. Deshalb wollen auch wir uns daran halten.

1.1.3. Beschreibung der elementaren Datenstrukturen

Abbildung 1.4 gibt einen Überblick über die elementaren Datenstrukturen vom Byte bis zum 64-Bit-Quad- bzw. Doppelwort.

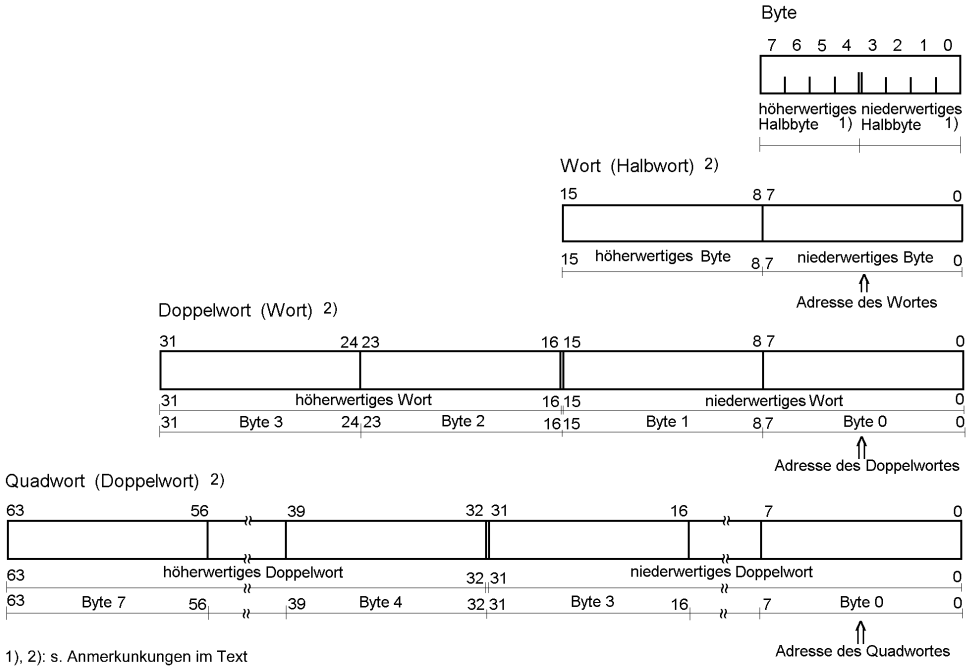


Abbildung 1.4 Adressierbare Behälter

Anmerkungen:

1) Halbbytes (auch Tetrade oder Nibble (sprich: Nibbl) genannt) dienen vor allem zur Speicherung von Ziffernstellen binär codierter Dezimalzahlen (Abschnitt 1.2.7.).
2) in Klammern: die Bezeichnung in Architekturen, die von Grund auf mit 32 Bits Verarbeitungsbreite ausgelegt wurden (z. B. Mips).

1.1.4. Zu Entwicklungsgeschichte und Zukunft

In der Frühzeit der Computer-Entwicklung war Hardware sehr teuer. Die Entwickler versuchten deshalb, mit möglichst wenigen Bits für die elementaren Datenstrukturen auszukommen. Um Zeichen (Buchstaben, Ziffern usw.) darzustellen, hatte man sich zunächst mit 6 Bits begnügt. Damit können 64 verschiedene Zeichen codiert werden. Maschinenworte waren dann Vielfache von 6 Bits. So ergaben sich Verarbeitungsbreiten von 12, 24, 36, 48 und 60 Bits.

IBM hatte seit 1963 mit dem System /360 das 8-Bit-Byte und das 32-Bit-Wort faktisch als Industriestandard eingeführt. Ein wichtiger Grund für 8 Bits war seinerzeit die Möglichkeit, im Vergleich zu 6 Bits genügend Code-Reserven für internationale Zeichensätze zu haben (denken Sie nur an deutsche Umlaute, an das griechische Alphabet, an kyrillische Zeichen usw.). Hinzu kommt die Eleganz: man kann alle Datenstrukturen systematisch vom Bit an in Schritten aufbauen, die Zweierpotenzen sind (2, 4, 8, 16 usw.). Das vereinfacht Adressierungsschaltungen in der Hardware und oft auch Adreßrechnungen in Programmen.

Architekturen mit Datentypkennzeichnung
Die Tatsache, daß Informationsstrukturen als bloße Aneinanderreihungen von Bits gespeichert und transportiert werden, hat von Anfang an wesentlich zur Universalität des digitalen Computers beigetragen. Die Nachteile sind aber offensichtlich:

- Anfälligkeit gegen Fehler. Die fälschliche Nutzung einer Datenstruktur wird nicht bemerkt; der Computer wird beispielsweise mit einem Additionsbefehl für Binärzahlen ohne weiteres eine Zeichenkette zu einer Dezimalzahl addieren, wobei naturgemäß ein unsinniges Ergebnis entsteht.
- unübersichtlicher Befehlsvorrat. Da nur der Befehl, der gerade ausgeführt wird, über die Interpretation der Datenstrukturen bestimmt, braucht man für jede zulässige Kombination von Datentypen einen besonderen Befehl. Befehle stehen also nicht für geradezu selbstverständliche Operationen, wie Addieren, Multiplizieren usw., wobei die Maschine von sich aus das jeweils Richtige tut. Statt dessen muß der Programmierer genau angeben, ob Binärzahlen mit Vorzeichen, solche ohne Vorzeichen, Gleitkommazahlen usw. zueinander addiert oder miteinander multipliziert werden sollen.

Diese Nachteile haben zu verschiedenen Ansätzen für Architekturen mit Datentypkennzeichnung geführt. Dabei ist jeder adressierbare Behälter um zusätzliche Bits erweitert, die den Typ des Behälter-Inhalts angeben. (Kennzeichnung oder Markierung heißt im Englischen Tag*), daher nennt man solche Architekturen Tagged Architectures*). Ein Vertreter dieser Tagged Architectures war die Rechnerfamilie B5000...B6700 der Fa. Burroughs, die von 1962 an bis in die 80er Jahre hinein angeboten wurde. Abbildung 1.5 veranschaulicht den Unterschied zwischen üblichen Datenstrukturen und solchen mit Datentypkennzeichnung.

*): sprich: Tägg, Täggd Arkitetschrs.

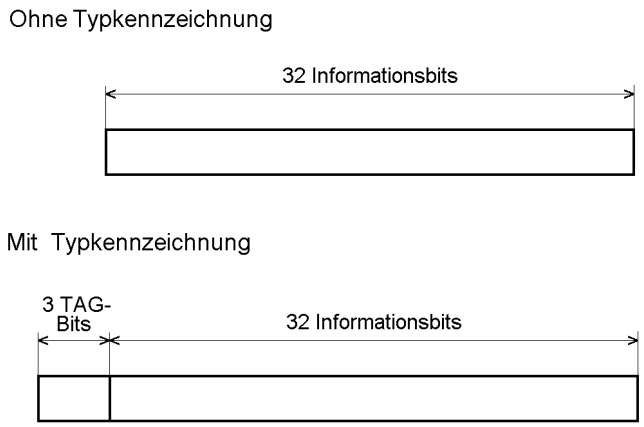


Abbildung 1.5 Datenstrukturen mit und ohne Typkennzeichnung (Beispiele)

Wir erkennen aber auch die offensichtlichen Nachteile der zusätzlichen Datentypkennzeichnung: mehr Bits, damit Mehrbedarf an Speicherplatz, geringere Flexibilität und Universalität. Die Kennzeichnung gilt schließlich nur für die Datentypen, die in der Architektur von vornherein definiert sind. Natürlich kann man beliebige neue einführen; man muß nur wenigstens einen architekturseitigen Datentyp "universeller Behälter" vorsehen. Dann hat man allerdings keinen Vorteil mehr, aber den Nachteil, die Kennzeichnungsbits immer mitspeichern zu müssen, und man ist an das naturgemäß unflexible Behälter-Format gebunden. Die bisher üblichen Datenstrukturen werden also auch in der absehbaren Zukunft Bestand haben. Hierfür sprechen folgende Tatsachen:

- alternative Ansätze haben sich bisher nicht durchsetzen können; Maschinen mit zusätzlichen Tag-Bits wurden in den letzen Jahren nur noch zu Spezialzwecken entwickelt (z. B. um die Programmiersprache LISP zu unterstützen),
- für die Anwender ist es zumeist noch schwieriger, mit neuartigen, vom Üblichen abweichenden, Datenstrukturen zurechtzukommen als mit neuartigen Maschinenbefehlen. (Ein neues Programm, das mit den vorhandenen Daten arbeiten kann, wird einfach installiert. Wirklich ungemütlich wird es hingegen dann, wenn auch die Datenbestände gewandelt werden müssen - die heutzutage nicht selten eine geradezu gigantische Größenordnung haben.*))
- bei Programmierung in höheren Programmiersprachen können Zulässigkeitsprüfungen vom Compiler übernommen werden. Die Eleganz der Maschinenbefehle spielt dabei ohnehin keine Rolle.

- ganz elementare Formen der Datentypkennzeichnung lassen sich auch im Rahmen weithin eingeführter Datenstrukturen unterbringen. So unterstützen die Architekturen IA-32 (vom 486 an) und Sparc ein 32-Bit-Wort, in dem zwei Bits als Datentypkennzeichnung (Tag) interpretiert werden. Eine derart einfache Vorkehrung reicht oft aus, um, gestützt auf die Verarbeitungsgeschwindigkeit moderner Prozessoren, weiterführende Konzepte programmseitig zu implementieren.
- Typkennzeichen können in *deskriptiven Strukturen* untergebracht werden, die ihrerseits die eigentlichen Daten beschreiben (Stichwort: Objektorientierung). Dabei entfallen die Einschränkungen der bekannten Architekturen mit Datentypkennzeichnung. Beispiel: die sog. Class Files der Java Virtual Machine (Seite 90).

*): die Erfahrung zeigt, daß selbst der bescheidene private PC-Nutzer seine Schwierigkeiten hat, wenn Datenbestände, die sich im Laufe der Zeit angesammelt haben, plötzlich nicht mehr ohne weiteres zur Software passen (z. B. bei Wechsel zu einem neuen “Office”-Programmpaket).

1.2. Numerische Datentypen

Numerische Datentypen dienen der Darstellung von Zahlenangaben. Im folgenden wollen wir die wichtigsten numerischen Datentypen überblicksmäßig beschreiben.

1.2.1. Zur Einführung: Dezimal- und Binärzahlen

Unser gewohntes Dezimalsystem ist ein Stellenwertsystem. Machen wir uns klar, wie die Zahlen eigentlich dargestellt werden! Es gibt 10 Zahlzeichen (Ziffern): 0, 1, 2 usw. bis 9. Bei einer einstelligen Zahl drückt die einzige Ziffer ihren Wert direkt aus. 0 bedeutet "nichts", 1 bedeutet Eins usw. Bei einer zweistelligen Zahl muß man sich die links stehende Ziffer mit 10 multipliziert denken. 92 bedeutet an sich (9 · 10) + 2. Allgemein: die am weitesten rechts stehende Ziffer kann man sich mit 1 multipliziert denken, die zweite (links davon stehende) Ziffer mit 10, die dritte mit 100 usw. Nun gilt $1 = 10^0$; $10 = 10^1$; $100 = 10^2$; $1000 = 10^3$ usw. Die Bedeutung der einzelnen Stellen hängt also irgendwie von der Zahl Zehn ab: Zehn ist die *Basis* unseres Dezimalsystems (Abbildung 1.6).

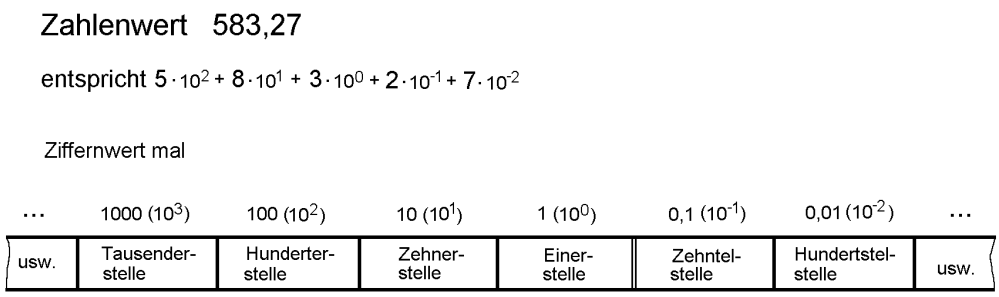


Abbildung 1.6 Zahlendarstellung im Dezimalsystem

Dieses Schema läßt sich verallgemeinern. Jede beliebige natürliche Zahl, die größer als Null ist, läßt sich als Basis eines Stellenwertsystems verwenden. Sei diese Basis-Zahl n. Wir brauchen dann n verschiedene Zahlzeichen (Ziffernsymbole) für die Werte Null (die

Null brauchen wir unbedingt!), Eins, Zwei usw. bis (n-1). Wir setzen das Symbol # als Stellvertreter für jedes der n Ziffernsymbole und erhalten so das allgemeine Schema eines Stellenwertsystems (Abbildung 1.7).

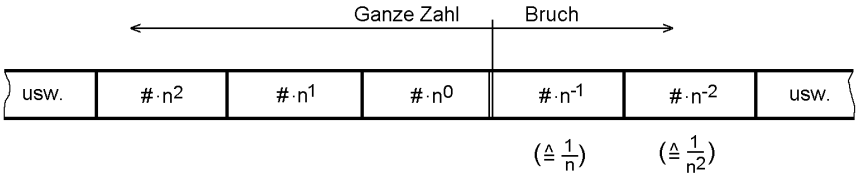


Abbildung 1.7 Zahlendarstellung in einem beliebigen Stellenwertsystem zur Basis n

Das Dezimalsystem ist aus mathematischer Sicht durch keine Besonderheit hervorgehoben. (Es ist offenbar aus dem Zählen mit den Fingern beider Hände hervorgegangen. Aus der antiken Mathematik kennen wir auch Systeme zur Basis 12 und zur Basis 60; unsere Zeit-Einteilung in Stunden, Minuten usw. geht noch darauf zurück.)

Binärzahlen
Wenn alles reine Vereinbarungssache ist, warum dann nicht die *kleinste* Basis wählen? Wir brauchen die Null und noch ein weiteres Ziffernsymbol für den Wert Eins und können somit beliebige Zahlen in einem Stellenwertsystem zur Basis 2 (Binärsystem) darstellen (Abbildung 1.8). Der Vorteil: Zahlenwerte lassen sich in einer solchen Darstellung mit technischen Mitteln besonders einfach übertragen, speichern und verarbeiten (Prinzip der Zweiwertigkeit.

usw.	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	usw.
	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	

Beispiel: 1 0 1 1 0 = $2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$

Abbildung 1.8 Zahlendarstellung im Binärsystem

1.2.2. Zur Notation: binärer - hexadezimal - dezimal

Binärzahlen
Betrachten wir den Ausdruck "10110". Wenn wir nicht genau wissen, worum es geht, würden wir ihn als Zahl "Zehntausendeinhundertzehn" ansehen. Es kann sich aber auch um eine Binärzahl handeln. Aus Abbildung 1.8 kennen wir deren Wert: 22. Wie also die Zahlenangaben unterscheiden? - Konrad Zuse hatte seinerzeit einfach die binäre Eins auf den Kopf gestellt und durch ein "L" wiedergegeben (10110 binär entspricht also LOLLO und ist deshalb nicht mit Zehntausendeinhundertzehn zu verwechseln).

Achtung:
Diese Schreibweise ("L" als binäre Eins und auch als Wahrheitswert "1") hat sich in der Literatur des deutschen Sprachraums bis weit in die 60er Jahre hinein gehalten. Verwechseln Sie diese Darstellung nicht mit dem logischen Pegel L (für LOW)! (Abgesehen davon ist älteres Schrifttum als Grundlagen-Lehrmaterial durchaus noch brauchbar, wenn nicht

gar mancher modernen Darstellung überlegen. Verständlich: als die Digitaltechnik noch weithin ungewohnt war, ist man eben an die Grundlagen gewissenhafter herangegangen.)

Heutzutage hat man sich an die Schreibweise moderner Programmiersprachen angelehnt. Man schließt binäre Angaben beispielsweise in Hochkommas ein (Beispiel: '10110') oder stellt ein "B" nach (10110B; diese Bezeichnungsweise wollen wir im Lehrmaterial beibehalten). Mehrstellige binäre Angaben werden häufig nach jeweils 3, 4 oder 8 Stellen durch Leerzeichen getrennt (10110B = 1 0110B), vergleichbar zu den Zwischenräumen nach jeweils drei Stellen im Dezimalen (9236374 = 9 236 374).

Oktal- und Hexadezimalzahlen
Das sind keine besonderen Datenstrukturen, sondern Notationsweisen für binäre Werte. Natürlich kann man diese als Folgen von Einsen und Nullen angeben, die man, wie eben gezeigt, beispielsweise mit einem nachgestellten "B" abschließt. Die Probleme:

- solche Angaben sind lang und unübersichtlich (z. B. folgendes 16-Bit-Wort: 1001000011001011B),
- man kann sie kaum aussprechen.

Der Ausweg liegt darin, mehrere Binärstellen in einem Zeichen zusammenzufassen. In einer *Oktalzahl* werden drei aufeinanderfolgende Bits zusammengefaßt, in einer *Hexadezimalzahl* vier (Abbildung 1.9). Bei Oktalzahlen wird jedes Bitmuster mit einer Ziffer zwischen 0 und 7 gekennzeichnet, bei Hexadezimalzahlen mit einer der Ziffern 0...9 bzw. einem der Buchstaben A...F. Im obigen Beispiel wird das 16-Bit-Wort*) oktal mit 110313 wiedergegeben und hexadezimal mit 90CB.

*): die binäre Darstellung in entsprechend aufgelöster Form:
1 001 000 011 001 011 bzw. 1001 0000 1100 1011.

Nutzung und Kennzeichnung
Oktalzahlen stammen aus der Zeit, als Zeichen üblicherweise noch mit 6 Bits codiert wurden. Die Maschinen hatten Verarbeitungsbreiten von beispielsweise 12, 24, 36 oder 48 Bits. Um Binärwerte bequem notieren, aussprechen und an Bedientafeln anzeigen sowie eingeben zu können, bot sich die Zerlegung in 3-Bit-Abschnitte an (der hardwareseitige Vorteil: für die Ziffern zwischen 0 und 7 konnten allgemein verfügbare dezimale Anzeigen und Eingabe-Schalter verwendet werden).

Mit der Einführung des 8-Bit-Bytes (IBM System /360) setzte sich die Hexadezimaldarstellung nach und nach durch.

Hinweis:
Manchmal bezeichnet man Hexadezimalzahlen auch als *Sedezimalzahlen*.

Oktalzahlen werden oft durch eine tiefgestellte "8" am Ende gekennzeichnet, Hexadezimalzahlen durch ein vorangestelltes "X" oder ein nach- bzw. tiefergestelltes "H".

Die Zusammenfassung der Bits in Dreier- bzw. Vierergruppen beginnt immer mit der niedrigstwertigen Position (also von ganz rechts an). So wird z. B. eine 6-Bit-Angabe 110100B zwecks Hexadezimaldarstellung in 11 0100B zerlegt; es ergibt sich also 34H. Will man kürzere Bitfolgen oktal oder hexadezimal wiedergeben, so werden die Bits rechtsbündig angeordnet. Längere Bitketten werden vom niedrigstwertigen Bit an in Abschnitte von 3 oder 4 Bits Länge zerlegt. Dabei werden links außen fehlende Bitpositionen stets durch Nullen ergänzt.

So gilt z. B. 11B = 0011B = 3H, 101B = 5H, 11 1010B = 3AH, 101 1101 = 5DH = 135_8.

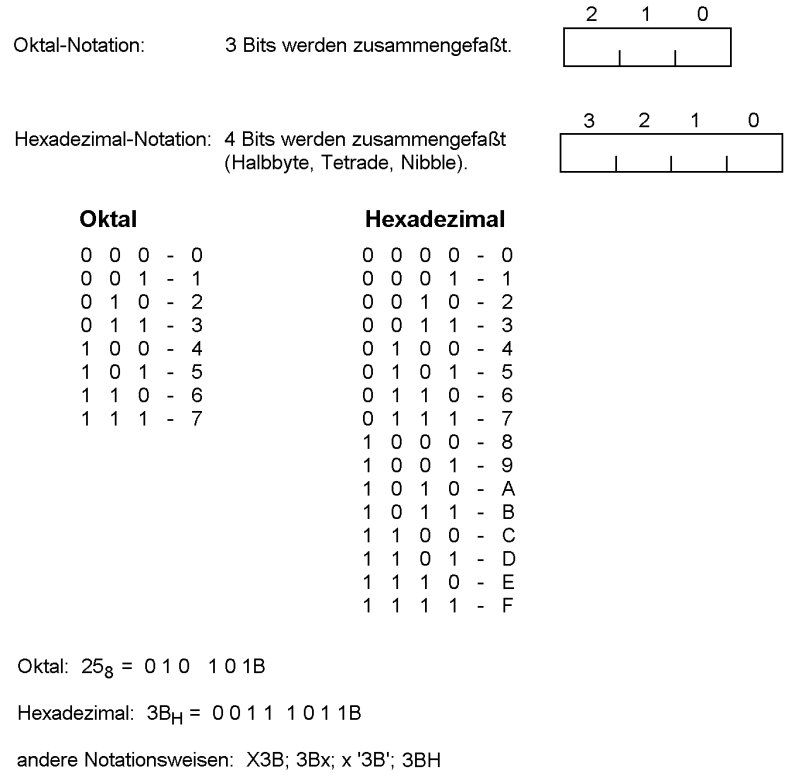


Abbildung 1.9 Oktal- und Hexadezimalnotation

Hinweise:

1. Oktalzahlen sind in der PC-Technik bedeutungslos.
2. Wir verwenden ausschließlich Hexadezimalzahlen und kennzeichnen sie durch ein nachgestelltes "H".
3. Was wirklich “sitzen” sollte: die Wandung von Hexadezimalen ins Binäre und umgekehrt (Hexadezimalzahlen sind in der professionellen Dokumentation nahezu allgegenwärtig, Konfigurationsangaben, Fehlermeldungen usw. werden oft hexadezimal dargestellt).
4. Zum Rechnen mit Binär- und Hexadezimalzahlen: es gibt besondere Taschenrechner- Modelle, die entsprechende Funktionen haben. Auch der in Windows enthaltene Rechner ist brauchbar. In Anhang 2 haben wir Anregungen zum Rechnen mittels PC sowie einige Übungsbeispiele zusammengestellt.

Die Dezimalnotation
Auch diese wird gelegentlich angewendet. Beispiel: IP-Adressen. Die herkömmliche IP-Adresse ist 32 Bits lang. Die 32-Bit-Angabe wird in 4 Abschnitte zu je 8 Bits (Octets) eingeteilt. Jedes Octet wird als natürliche (vorzeichenlose) Binärzahl aufgefaßt, deren Wert in's Dezimale gewandelt wird. Der jeweils kleinste Wert: 00H = 0, der jeweils größte Wert: FFH = 255. Die 4 Dezimalzahlen werden durch Punkte (Dots) voneinander getrennt (Dotted Decimal Notation).

Beispiel einer IP-Adresse:
169.254.61.151 = A9 FE 3D 97 = 1010 1001 1111 1110 0011 1101 1001 0111.

Auch Zeichencodes (Abschnitt 1.3.2.) werden gelegentlich dezimal angegeben. So entspricht der ASCII-Code 32 = 20H dem Leerzeichen, der Code 71 = 47H dem Zeichen "G" usw.

1.2.3. Natürliche Binärzahlen

Natürliche Zahlen sind vorzeichenlos*). Der niedrigste Wert ist Null. Der gesamte Wertebereich einer natürlichen Binärzahl x aus n Bits (Abbildung 1.10) ist gegeben durch

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

Beispiel: eine natürliche Binärzahl aus 8 Bits (n = 8):

- kleinster Wert = 0 = 0000 0000B = 0H,
- größter Wert = 2^8-1 = 255 = 1111 1111B = FFH.

*): engl. Natural bzw. Unsigned Numbers (sprich: Nättscherell/Annseind Nammbers). Das Vorzeichen - hier feht es - heißt Sign (sprich: Sein).

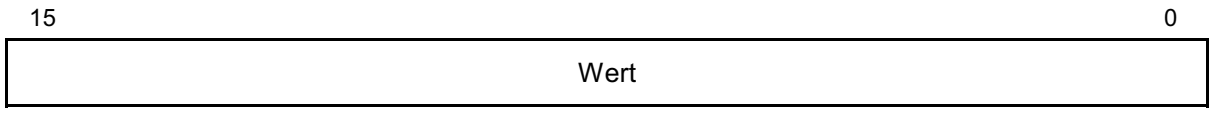


Abbildung 1.10 Beispiel einer natürlichen (vorzeichenlosen) Binärzahl (16 Bits)

Elementare Formate
In Tabelle 1.1 sind Formate natürlicher Binärzahlen angegeben, die heutzutage gebräuchlich sind (die Zahlen belegen gemäß ihrer Länge Bytes, Worte usw.).

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*)

*): 10^9 = 1 Milliarde (im Englischen: 1 Billion); 10^{18} = 1 Trillion (im Englischen: 1 Quintillion); 18,4 Trillionen = 18 Milliarden Milliarden. Ganz genau: $2^{64} - 1$ = 18 446 744 073 709 551 615

Tabelle 1.1 Natürliche Binärzahlen als elementare Datentypen

1.2.4. Ganze Binärzahlen

Ganze Zahlen*) haben ein Vorzeichen (+ oder -). Das Vorzeichen belegt ein Bit, und zwar jeweils das höchstwertige (Most Significant Bit MSB) im betreffenden Behälter (Abbildung 1.11).

*): Fachbegriff: Integer-Zahlen oder kurz Integers (sprich: Inntetschrs). Auch: Signed Numbers (sprich: Seind Nammbers).

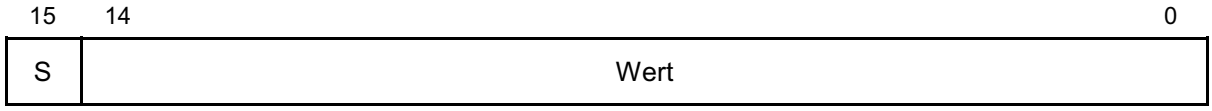


Abbildung 1.11 Beispiel einer ganzen Binärzahl (16 Bits). S = Vorzeichen (Sign)

Zahlendarstellung
In allen modernen Architekturen wird die sog. Zweierkomplementdarstellung verwendet. Das Vorzeichen wird folgendermaßen codiert:

- 0: positiv (+),
- 1: negativ (-).

Die verbleibenden Bits repräsentieren den Wert der Zahl, allerdings *nicht* unabhängig vom Vorzeichen.

Eine positive Zahl (einschließlich der Zahl 0) wird genau so dargestellt wie eine natürliche Binärzahl. Sie hat in der höchstwertigen Stelle (Vorzeichenbit) eine Null.

Eine negative Zahl hat in der höchstwertigen Stelle (Vorzeichenbit) eine Eins. Der Zahlenwert wird als Zweierkomplement angegeben ($-x \triangleq 2^n - x$).

Der gesamte Wertebereich einer ganzen Binärzahl z aus n Bits ist gegeben durch

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

Beispiel: eine natürliche Binärzahl aus 8 Bits (n = 8):

- kleinster Wert (kleinste negative Zahl) = $-(2^7) = -128 = 1000\ 0000B = 80H$,
- Wert - 1 (größte negative Zahl) = 1111 1111B = FFH,
- Wert 0 = 0000 0000B = 00H,
- Wert + 1 (kleinste positive Zahl) = 0000 0001B = 01H,
- größter Wert (größte positive Zahl) = $2^7 - 1 = 127 = 0111\ 1111B = 7FH$.

Kleiner/größer - eine Wiederholung aus der Elementarmathematik
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 (eine negative Zahl ist um so kleiner, "je negativer" sie ist). Beispiele: - 8 ist kleiner als - 3 (- 8 < - 3) und - 2 ist kleiner als + 2 (- 2 < + 2). Umkehrung: - 3 ist größer als - 8 (- 3 > - 8), + 2 ist größer als - 2 (+ 2 > - 2). - ∞ ist kleiner als + ∞. Jede negative Zahl ist kleiner als Null.

Elementare Formate
In Tabelle 1.2 sind Formate ganzer Binärzahlen angegeben, die heutzutage gebräuchlich sind (die Zahlen belegen gemäß ihrer Länge Bytes, Worte usw.).

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 1.2 Ganze Binärzahlen (Integer-Zahlen) als elementare Datentypen

1.2.5. Einführung in das Rechnen mit Binärzahlen

Addieren und Subtrahieren
Im Binären wird genauso schulmäßig gerechnet wie im Dezimalen. In einer beliebigen Stelle haben wir zwei Operandenbits und einen einlaufenden Übertrag zu verarbeiten. Wir erhalten ein Summen- und ein Übertragsbit für die nächste Stelle. Beim Subtrahieren kennzeichnen die Überträge ein "Borgen" von der jeweils höherwertigen Stelle, genau wie im Dezimalen. Im Binären sind die Rechenregeln recht einfach (Abbildungen 1.12, 1.13).

Addieren und Subtrahieren im Computer
Die Subtraktion wird typischerweise nicht unmittelbar gemäß den angegebenen Rechenregeln, sondern als Addition des Zweierkomplements (engl. Two's Complement) ausgeführt.

Bildung des Zweierkomplements
Der Zahlenwert wird zunächst bitweise negiert. Danach wird eine Eins hinzuaddiert.

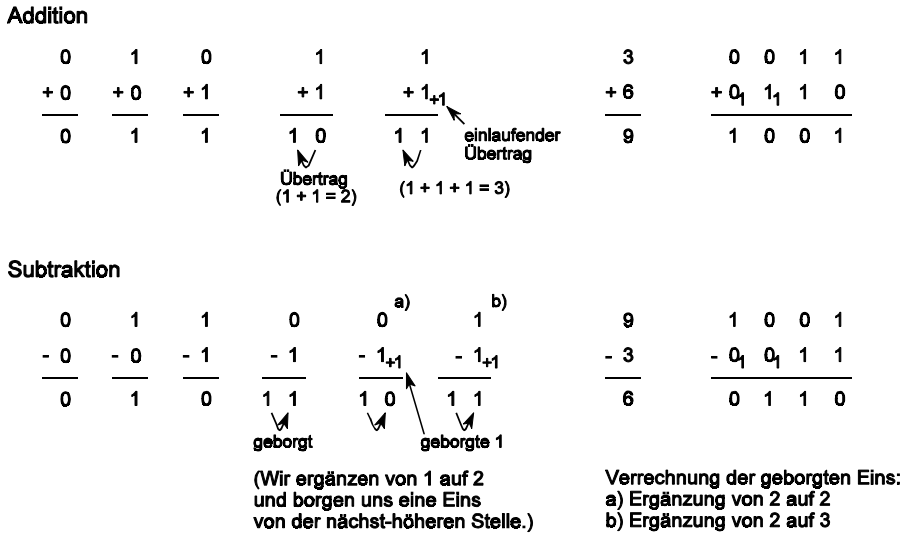


Abbildung 1.12 Addieren und Subtrahieren von Binärzahlen

Über- und Unterschreiten des Wertebereichs

1. beim Rechnen mit natürlichen Binärzahlen
Wird der Wertebereich über- oder unterschritten, so entsteht ein Übertrag in der höchstwertigen Stelle (*Ausgangsübertrag* (Carry Out; sprich: Kärrie Aut)), der das Rechenergebnis gleichsam um eine Stelle verlängert (Abbildung 1.14). Von anwendungspraktischer Bedeutung ist weiterhin, welche Ergebnisse entstehen, wenn wir den Ausgangsübertrag gar nicht berücksichtigen (wenn wir also nur das Ergebnis in den n Bits gemäß der jeweiligen Operandenlänge betrachten).

Entsteht der Ausgangsübertrag beim Addieren (A + B), so überschreitet das Ergebnis den Wertebereich (Abbildung 1.14a). Ist das eigentliche (mit dem Ausgangsübertrag um eine Stelle längere) Rechenergebnis gleich r (ist also $r \geq 2^n$), so ergibt sich das Resultat r_n bei n Bits Operandenlänge (d. h. ohne Ausgangsübertrag) zu $r_n = r - 2^n$.

Entsteht der Ausgangsübertrag beim Subtrahieren (A - B), so unterschreitet das Ergebnis den Wertebereich (ist A < B, so entsteht ein negativer Wert - der im Bereich der natürlichen Zahlen nicht zulässig ist)[*)]. Ist das eigentliche Ergebnis eine negative Zahl - r, so entspricht das Resultat r_n bei n Bits Operandenlänge dem Zweierkomplement des eigentlichen Ergebnisses (Abbildung 1.14b).

2. beim Rechnen mit ganzen Binärzahlen
Das Resultat liegt *im Wertebereich*, wenn (1) weder ein Übertrag in die Vorzeichenstelle noch ein Übertrag aus der Vorzeichenstelle (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 (Abbildung 1.13b).

Wird die kleinste negative Zahl unterschritten, so entsteht nur ein Ausgangsübertrag, aber kein Übertrag in die Vorzeichenstelle (Abbildung 1.13e).

Allgemein wird das Verlassen des Wertebereichs ganzer Zahlen als *Überlauf* (Overflow; sprich: Oferfloh) bezeichnet.

*): wird durch Addieren des Zweierkomplements subtrahiert, so verhält es sich genau umgekehrt (Bereichsunterschreitung, wenn *kein* Ausgangsübertrag gebildet wird).

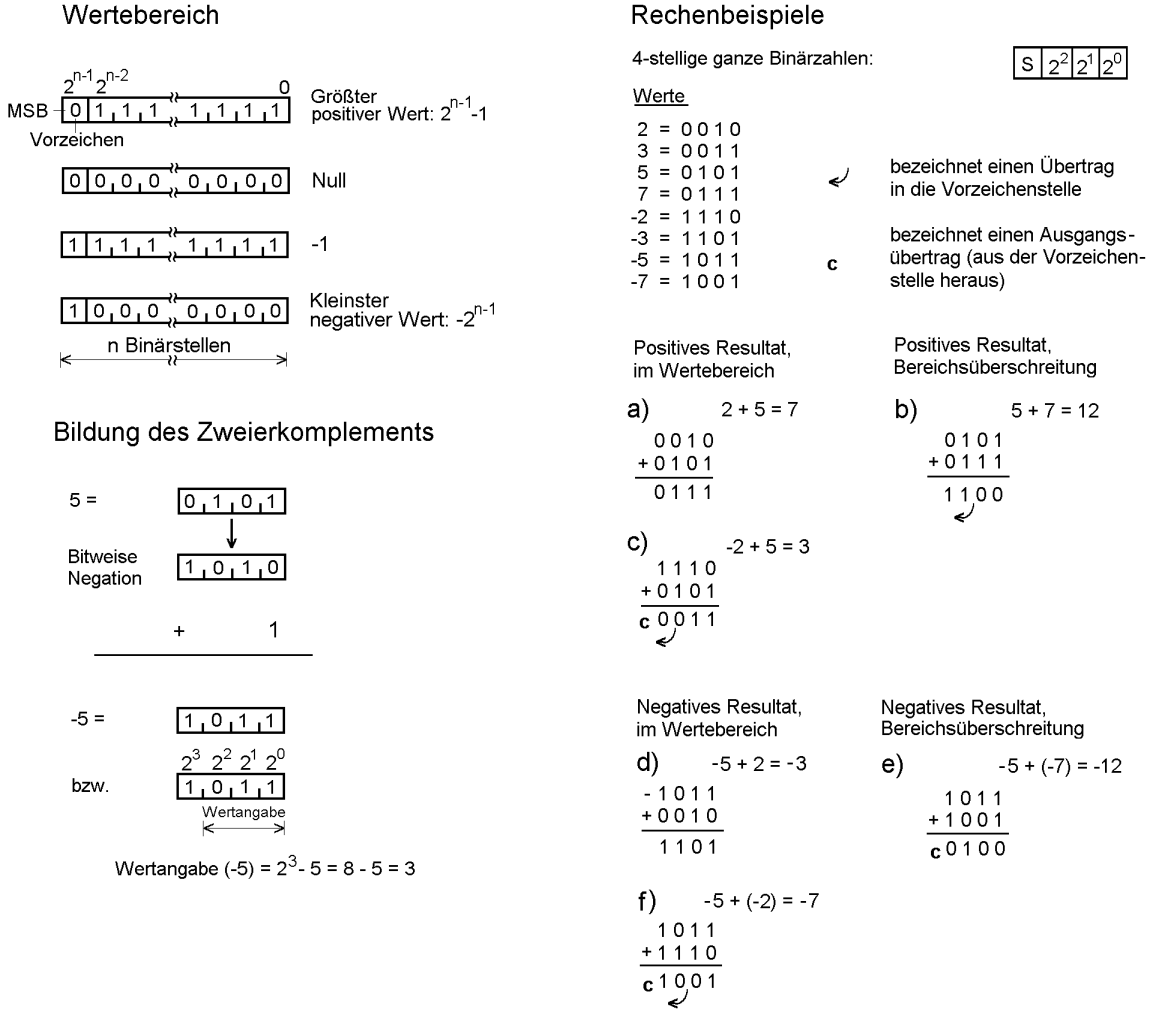


Abbildung 1.13 Elementares Rechnen mit ganzen Binärzahlen

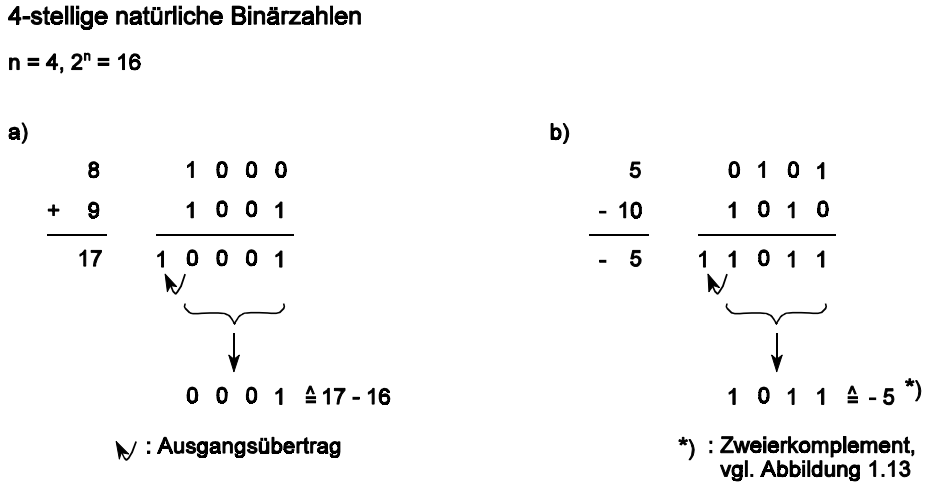


Abbildung 1.14 Über- und Unterschreiten des Wertebereichs beim Rechnen mit natürlichen Binärzahlen

Addieren und Subtrahieren beliebig langer Binärzahlen
Um Rechenoperationen mit beliebig langen natürlichen Binärzahlen programmieren zu können, muß es möglich sein, den Ausgangsübertrag (Carry Out) als Eingangsübertrag (Carry In) in nachfolgende Rechnungen einfließen zu lassen. Die meisten Architekturen haben entsprechende Befehle (Beispiel: x86/IA-32: "Addieren/Subtrahieren mit Eingangsübertrag" (ADC/SBC)).

Ganze und natürliche Binärzahlen beim Addieren und Subtrahieren
Addition und Subtraktion laufen für natürliche und ganze Binärzahlen gleichermaßen ab; die Unterscheidung kommt einzig dadurch zustande, wie Operanden und Resultat interpretiert werden (eine der vorteilhaften Eigenschaften der Zweierkomplementarithmetik)[*)]. Verschiedene Additions- und Subtraktionsbefehle für natürliche und ganze Binärzahlen sind deshalb nicht notwendig, wohl aber verschiedene Multiplikations- und Divisionsbefehle.

*): die Ergebnisse in Abbildung 1.13 sind auch korrekt, wenn man die 4-stelligen Binärzahlen als natürliche (vorzeichenlose) Zahlen interpretiert. In diesem Sinne dezimal umcodiert, stellen sich die Beispiele folgendermaßen dar:

a) 2 + 5 = 7; b) 5 + 7 = 12; c) 14 + 5 = 19;

d) 11 + 2 = 13; e) 11 + 9 = 20; f) 11 + 14 = 25

(Der Ausgangsübertrag wird als fünfte Binärstelle angesehen.)

Des weiteren werden auch dann korrekte Resultate gebildet, wenn man einen Operanden als natürliche und den anderen als ganze Binärzahl interpretiert (eine typische Anwendung: die Adreßrechnung - vgl. Abschnitt 2.6.2.). Das Ergebnis ist wiederum eine natürliche (in den Beispielen: vierstellige) Binärzahl.

Beispiele (gemäß Abbildung 1.13):

e) läßt sich auffassen als - 5 + 9 = 4 bzw. als 11 - 7 = 4,

f) läßt sich auffassen als -5 + 14 = 9 bzw. als 11 - 2 = 9.

Hinweis:
In manchen Architekturen (z. B. Mips) sind gesonderte Additions- und Subtraktionsbefehle für natürliche und für ganze Binärzahlen vorgesehen. Der Unterschied besteht aber nicht im Rechengang, sondern lediglich darin, daß beim Rechnen mit natürlichen Zahlen die Überlaufbedingung (Overflow) nicht ausgewertet wird (tritt hingegen beim Rechnen mit ganzen Zahlen ein Überlauf auf, so wird eine Ausnahmebedingung (Kapitel 4) wirksam).

Multiplizieren und Dividieren
Im Binären wird genauso multipliziert und dividiert, wie wir es aus dem Schulunterricht vom Dezimalen her kennen (Abbildung 1.15).

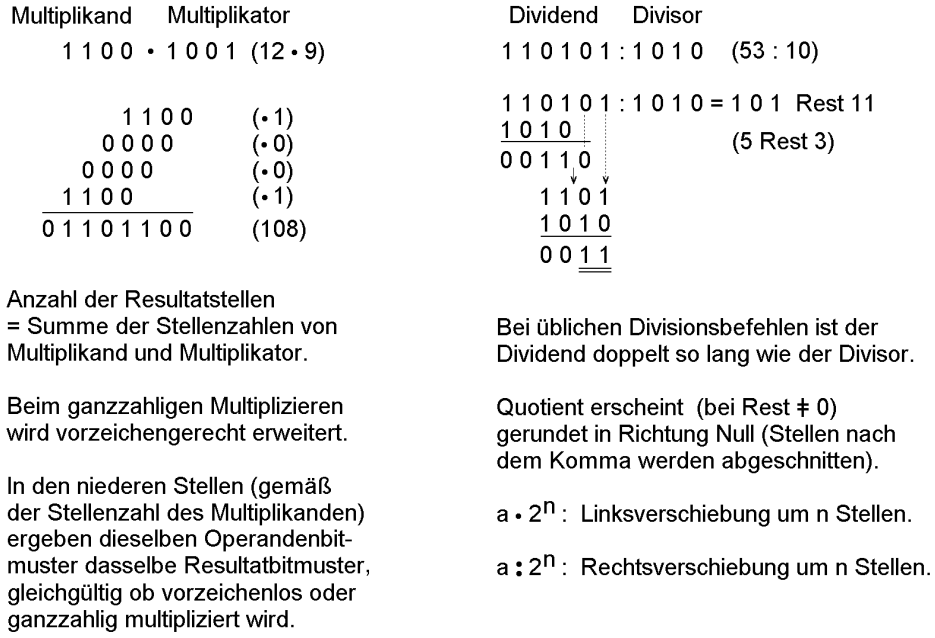


Abbildung 1.15 Multiplikation und Division von Binärzahlen anhand von Beispielen

Rechnen mit kurzen Binärzahlen
Sind Zahlen kürzer als die Verarbeitungsbreite, so werden sie *vorzeichengerecht* erweitert (Sign Extend). Dabei wird das Vorzeichen in *alle* auszufüllenden Stellen eingetragen (Abbildung 1.16).

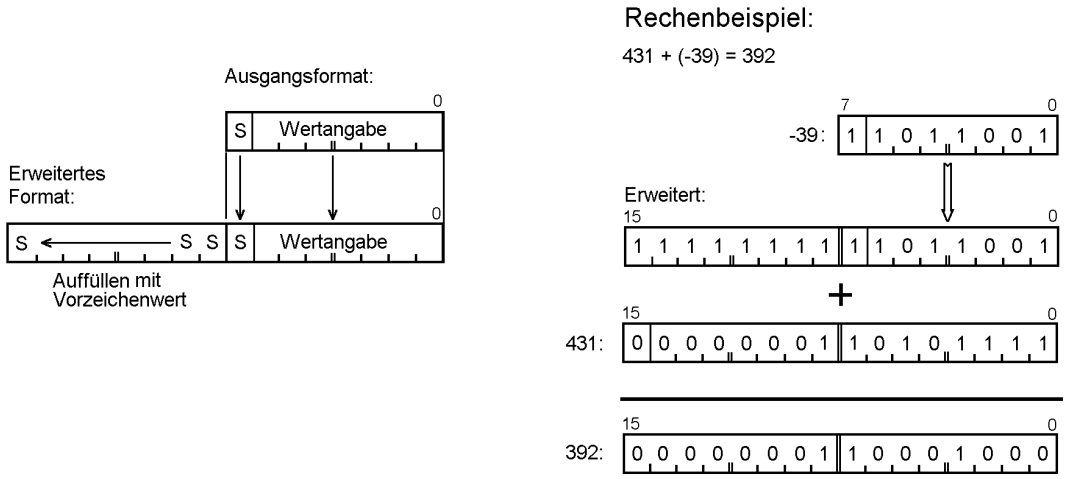


Abbildung 1.16 Prinzip der Vorzeichenerweiterung ganzer Binärzahlen

Vorsicht, Falle:
Daß gesonderte Additions- bzw. Subtraktionsbefehle für ganze und natürliche Binärzahlen überflüssig sind, wird in vielen modernen Rechnerarchitekturen ausgenutzt: Addition und Subtraktion sind zumeist nur für ganze Binärzahlen (Integers) spezifiziert (so auch bei x86/IA-32). Selbstverständlich lassen sich diese Befehle auch verwenden, um mit natürlichen (vorzeichenlosen) Binärzahlen zu rechnen. Ist aber ein Operand kürzer als der andere (wenn wir z. B. ein Byte zu einem Doppelwort addieren), so wird der kürzere Operand immer *vorzeichengerecht* erweitert, unabhängig davon, welche Bedeutung er vom Programmierer erhalten hat. Stellen Sie sich beispielsweise vor, Sie wollen Meßwerte verarbeiten und sind auf gute Ausnutzung des Speichers angewiesen. Liegen die möglichen numerischen Werte beispielsweise zwischen Null und 200, so werden Sie intuitiv für jeden Wert ein Byte vorsehen. Wenn Sie diese Bytes mit 16-Bit- oder 32-Bit-Zahlen verknüpfen wollen und einfach losrechnen, so macht die Vorzeichenerweiterung aus allen Werten über 127 negative Werte! *Abhilfe*: die Meßwerte werden *vor* dem eigentlichen Rechnen auf die jeweilige Verarbeitungsbreite gebracht (die Erweiterung wird also ausprogrammiert - was nicht besonders schwierig ist; in manchen Architekturen gibt es eigens Befehle zum Laden ohne Vorzeichenerweiterung[*)]).

*): IA-32: "Laden mit Nullerweiterung" (MOVZX).

1.2.6. Herkömmliche Arithmetik und Sättigungsarithmetik

Die herkömmliche (Zweierkomplement-) Arithmetik heißt im Englischen auch “Wrap-Around-Arithmetik” - ein wirklich bildhafter Begriff (Abbildung 1.17).

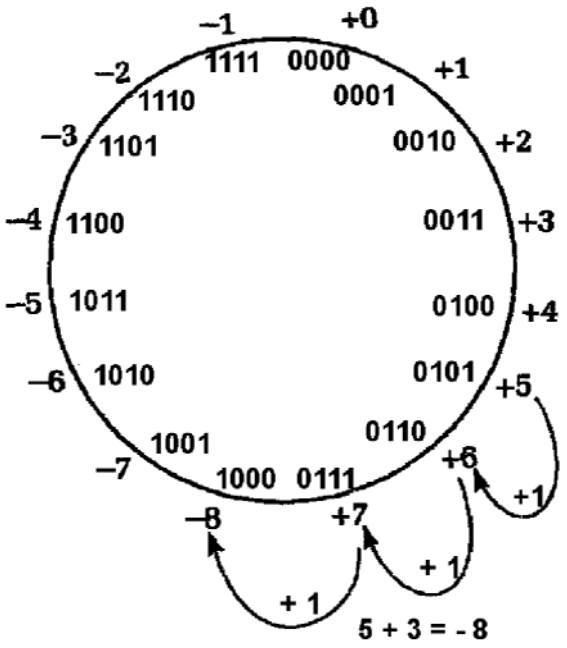


Abbildung 1.17 Zweierkomplement-Arithmetik als Wrap-Around-Arithmetik

Erklärung:
Die Mathematik kennt jeweils 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. Wenn wir beispielsweise von Null aus vorwärts zählen (0 → 1 → 2 → 3 usw.), so kommen wir irgendwann einmal zur größten positiven Zahl - und von dort durch einfaches Weiterzählen zur kleinsten negativen (0111B → 1000B ≙ 7 → - 8). Dann geht es rückwärts weiter bis zur -1 (1111B) und im nächsten Zählschritt wieder zur Null. Dies wirkt sich aus, wenn wir rechnen und dabei nicht auf die Überlaufbedingung achten. Das Ergebnis kann jeweils nach der anderen Seite umschlagen: wird die kleinste negative Zahl unterschritten, so ergibt sich ein positives Ergebnis, wird die größte positive Zahl überschritten, ein negatives. Abbildung 1.17 zeigt dies an einem Rechenbeispiel (5 + 3 = 0101B + 0011B = 1000B = - 8). Ein weiteres Beispiel: - 7 - 2 = + 7 (1001B + 1110B = 0111B).

Hinweis:
Der Ausgangsübertrag wird jeweils vernachlässigt (vgl. Abbildung 1.14).

Das Prinzip der Sättigungsarithmetik (Saturation Arithmetics*)) besteht nun darin, dieses Umschlagen zu vermeiden und die Zahlwerte sozusagen gegen den jeweiligen Anschlag fahren zu lassen (Tabelle 1.3): wird der Wertebereich überschritten, so wird als Ergebnis der jeweilige Größtwert geliefert, wird der Wertebereich unterschritten, der jeweilige Kleinstwert. Dis ist vor allem beim Rechnen mit Video- und Audio-Daten von Bedeutung (eine maximale Amplitude kann nicht noch weiter wachsen, ein Farbwert “schwarz” kann nicht noch dunkler werden usw. - bei herkömmlicher Arithmetik würde womöglich der Versuch, ein schwarzes Pixel noch schwärzer zu machen, zu einem hellen Pixel führen).

*): die herkömmliche Arithmetik heißt auch Non Saturation Arithmetics.

Anwendung: beispielsweise bei der MMX-Erweiterung (Abschnitt 9.1.7.).

Hinweis:
Die gleiche Wirkung ließe sich auch erreichen, 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 im Datenstrom (wie sie durch die programmseitige Behandlung von Überlaufbedingungen bzw. Bereichsüberschreitungen entstehen könnten).

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)

Tabelle 1.3 Herkömmliche und Sättigungsarithmetik

1.2.7. Binär codierte Dezimalzahlen

In binär codierten Dezimalzahlen (BCD-Zahlen) belegt eine Dezimalstelle 4 Bits, die (binär codiert) die Werte von 0 bis 9 annehmen können. Es gibt ungepackte und gepackte BCD-Zahlen (Abbildung 1.18).

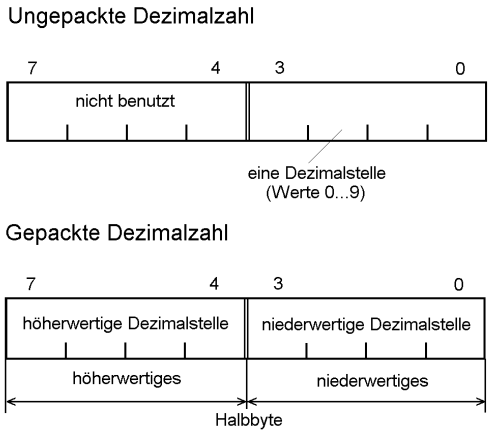
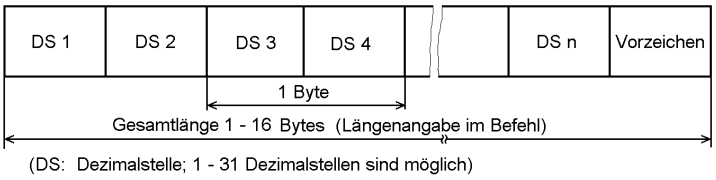


Abbildung 1.18 Binär codierte Dezimalzahlen (BCD-Zahlen)

Eine ungepackte Dezimalzahl ist ein einzelnes Byte mit einer einzigen Dezimalstelle im niederwertigen Halbbyte (Bits 0...3). Gepackte BCD- Zahlen enthalten in jedem Byte zwei Dezimalstellen, wobei die Stelle im höherwertigen Halbbyte die höherwertige ist. Diese

Datenstrukturen sind an sich vorzeichenlos. Ein Vorzeichen muß gesondert codiert werden (Vorzeichen und Betrag sind voneinander unabhängig; Sign/Magnitude-Darstellung). Abbildung 1.19 zeigt Beispiele von BCD-Zahlen.

1. S / 370



2. x86 BCD-Format für Gleitkommaverarbeitung

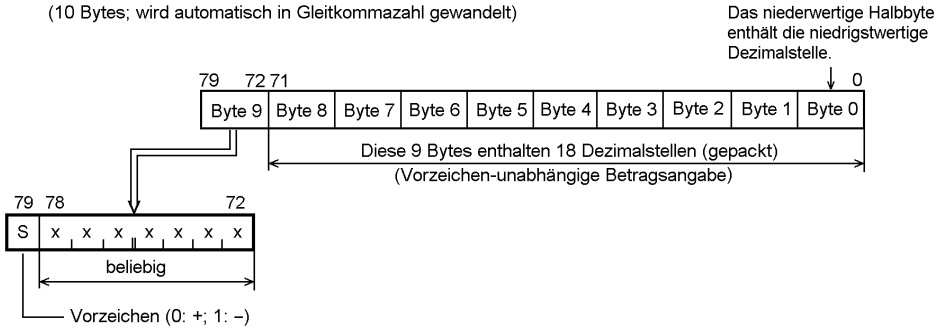


Abbildung 1.19 Formate vorzeichenbehafteter BCD-Zahlen (Beispiele)

Zur Entwicklungsgeschichte

In den klassischen Rechenzentrums-Maschinen sind Befehle für alle vier Grundrechenarten mit Dezimalzahlen vorgesehen. Zu Beginn der Entwicklung arbeiteten viele Rechner sogar ausschließlich mit Dezimalzahlen. Ein wichtiger Grund: die Anwendung im Finanzwesen (um nicht durch das Rundungsverhalten der Binärarithmetik andere Ergebnisse "nach dem Komma" zu bekommen als beim Rechnen von Hand oder mit Lochkartengeräten). Seinerzeit wurden verschiedene Codes für Dezimalzahlen entwickelt. Diese haben aber keine praktische Bedeutung mehr.

Moderne Mikroprozessoren haben keine ausgebaute BCD-Arithmetik. Die befehlsseitige Unterstützung - falls überhaupt vorgesehen (wie z. B. bei x86/IA-32) - betrifft vielmehr nur Hilfsoperationen, die dazu dienen, trotz der an sich binären Arbeitsweise des Prozessors korrekte dezimale Verarbeitungsresultate zu erzeugen (Dezimalausgleich*)). Die eigentliche BCD-Arithmetik wird softwareseitig implementiert.

*): Erklärung: die einzelnen BCD-Halbbytes dürfen lediglich die Werte zwischen 0H und 9H einnehmen. Beim binären Rechnen entstehen aber auch Werte zwischen AH und FH. Beispiel: 5 + 7 = 12 = CH (binäre Addition). Im BCD-Code müßte aber herauskommen: 2H + ein Dezimal-Übertrag in die nächste Stelle. BCD-Befehle sorgen u. a. dafür, daß das binäre Ergebnis entsprechend korrigiert wird.

1.2.8. Gleitkommazahlen

Grundlagen
Gleitkommazahlen (Floating Point Numbers, Reals)*) sind fest formatierte maschineninterne Darstellungen von Näherungswerten für reelle Zahlen. Reelle Zahlen sind vorzeichenbehaftete Zahlen mit beliebig vielen Dezimalstellen nach dem Komma.

*): sprich: Flohting Point Nammbers, Riehls.

An sich ist der Wertebereich der reellen Zahlen unbegrenzt, und eine reelle Zahl kann mit beliebiger Genauigkeit (d. h. Stellenzahl nach dem Komma) angegeben werden. Zwischen zwei reelle Zahlen passen jeweils immer noch unendlich viele andere. Demgegenüber sind Gleitkommazahlen 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.

Abbildung 1.20 zeigt den grundsätzlichen Aufbau einer Gleitkommazahl.

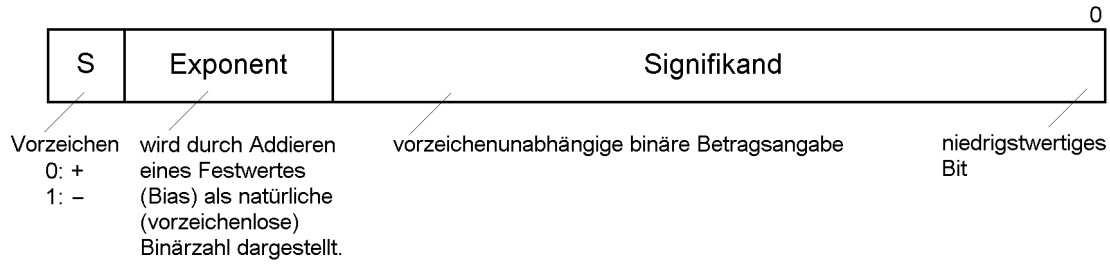


Abbildung 1.20 Grundsätzlicher Aufbau einer Gleitkommazahl

Eine Gleitkommazahl besteht aus:

- dem Signifikanden, auch als Mantisse oder Fraktion bezeichnet (engl. Mantissa, Fraction). Dieses Feld enthält die "signifikanten", d. h. gültigen, Ziffernstellen.
- dem Exponenten, auch als Charakteristik bezeichnet. Dieses Feld gibt die Größenordnung der Zahl an.
- dem Vorzeichen. Das Vorzeichen gibt an, ob die Zahl positiv oder negativ ist. Der Betrag der Zahl wird, unabhängig vom Vorzeichen, durch Signifikand und Exponent dargestellt (Sign/Magnitude-Darstellung).

Gleitkommazahlen beruhen auf der Exponentialdarstellung, die im Dezimalsystem keineswegs ungewohnt 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 $r \simeq s \cdot b^e$. Darin ist

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

Die Gleitkommadarstellung im Computer entspricht genau diesem Prinzip, beruht allerdings auf der Basis 2, d. h. auf Binärzahlen. Die Darstellungsweise kann man sich folgendermaßen veranschaulichen: Die reellen Zahlen sind auf einem Zahlenstrahl angeordnet. Mit Gleitkommazahlen kann ein endlicher Abschnitt dieses Zahlenstrahls maschinenintern abgebildet werden, wobei sich bestimmte Zahlenwerte exakt darstellen lassen und die verbleibenden näherungsweise. Der Exponent gibt jeweils eine Zweierpotenz an. Im Intervall bis zur nächstgrößeren Zweierpotenz kann ein n-stelliger Signifikand 2^n-1 diskrete Werte auswählen (unter Einschluß der durch den Exponenten bezeichneten Zweierpotenz), d. h. das Intervall wird in 2^n gleich lange Abschnitte unterteilt. Diese diskreten Werte sind exakt darstellbar; Zwischenwerte können nur näherungsweise durch Rundung auf den jeweils nächstgelegenen diskreten Wert dargestellt werden. Eine Gleitkommazahl ist also praktisch nichts anderes als eine verkürzte, komprimierte Darstellung einer extrem langen Binärzahl (Abbildung 1.21).

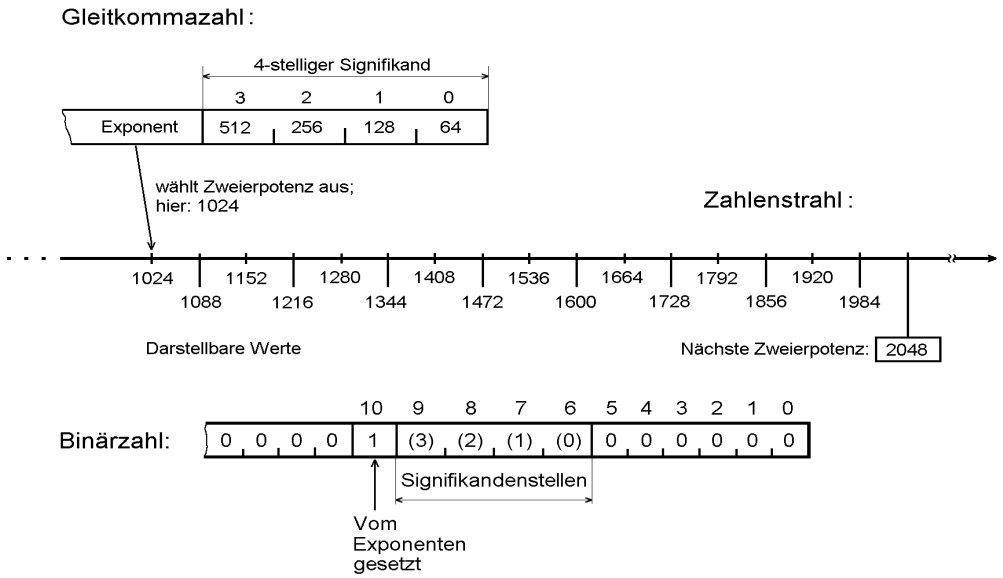


Abbildung 1.21 Näherungsweise Darstellung einer langen Binärzahl durch eine Gleitkommazahl

Erklärung zu Abbildung 1.21:
Der Signifikand umfaßt hier nur 4 Binärstellen. Der Exponent sei = 10. Er wählt somit die Bitposition 10 bzw. die Zweierpotenz $2^{10} = 1024$ aus. Diese Stelle wird = 1 gesetzt. Die 4 rechts folgenden (niederwertigen) Stellen werden mit den Bits des Signifikanden belegt. Die restlichen 6 Stellen bis zur Bitposition 0 (2^0) werden mit Nullen gefüllt. Eine so gebildete Binärzahl kann - je nach Belegung der Signifikandenbits - 16 mögliche Werte annehmen. Einige Beispiele:

- Signifikand = 0000B: 1024,
- Signifikand = 0001B: 1088 (1024 + 64),
- Signifikand = 1111B: 1984 (1024 + 512 + 256 + 128 + 64).

Negative Exponenten
Ein negativer Exponent bezeichnet einen Binärbruch, also eine Zahl < 1. Beispiel: Exponent = -4. Die erste Eins wird damit auf die Position $2^{-4} = 1/2^4 = 1/16$ des Zahlenstrahls gesetzt. Die Signifikandenbits 3, 2, 1, 0 haben dann sinngemäß die Stellenwerte 1/32, 1/64, 1/128, 1/256.

Beispiele:

- Signifikand = 0000B: 1/16 = 0,0625,
- Signifikand = 0001B: 1/16 + 1/32 = 3/32 = 0,009375,
- Signifikand = 1111B: 1/16 + 1/32 + 1/64 + 1/128 + 1/256 = 31/256 = 0,12109375.

Negative Zahlen
Sie unterscheiden sich im Prinzip nicht von den positiven. Es ist lediglich das Vorzeichen, das besagt, daß der Zahlenstrahl in die Gegenrichtung verläuft (von 0 aus nach links).

Normalisierte und denormalisierte Gleitkommazahlen
Der Exponent wird üblicherweise so gewählt, daß der Signifikand die Form $1 + b_0\ b_1\ b_2$ annimmt (eine Eins gefolgt von einem Binärbruch). Somit setzt der Exponent - wie in Abbildung 1.21 gezeigt - die höchstwertige Zweierpotenz der extrem langen Binärzahl stets auf 1. Nur wenn der Exponent den kleinstmöglichen Wert hat, wird im Interesse der Genauigkeit diese Darstellungsweise verlassen. Der Signifikand hat dann die Form $0.b_0\ b_1\ b_2$ ... (denormalisierte Darstellung).

Exponentenversatz (Bias)
Der Exponent wird durch Addieren eines konstanten Versatzwertes (Bias) stets als natürliche Binärzahl dargestellt. Obwohl an sich negative Exponenten gebraucht werden, vermeidet man so das Exponentenvorzeichen. Gleitkommazahlen in dieser Darstellung lassen sich dadurch auf einfachste Weise (wie natürliche Binärzahlen) miteinander vergleichen (Voraussetzung: gleiches Format, gleiches Vorzeichen). Den echten Exponenten erhält man durch Subtraktion des Versatzwertes.

Tabelle 1.4 zeigt ein Beispiel der Codierung einer Gleitkommazahl.

Darstellungsart	Wert
Dezimaldarstellung	239,125
Binärdarstellung	1 1 1 0 1 1 1 1.0 0 1 (0,125 = 1/8 = 0.001)
Gleitkommadarstellung (normalisiert)	1 .1 1 0 1 1 1 1 0 0 1 E 111 ↑----------↑ Komma um 7 Stellen verschoben
Darstellung mit einfacher Genauigkeit *)	
Vorzeichen	0
Exponent (mit Versatz + 127; 8 Bits)	1 0 0 0 0 1 1 0 (= 1 1 1 + 0 1 1 1 1 1 1 1)
Signifikand (23 Bits)	(1)*) 1 1 0 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
*): vgl. Abbildung 1.22 und Tabelle 1.5	*): die führende 1 wird nicht mitgespeichert (implizit)

Tabelle 1.4 Beispiel der Codierung einer Gleitkommazahl

Gleitkommaformate nach Standard IEEE 754-85

Zur Entwicklungsgeschichte
Die Gleitkommadarstellung wurde bereits von Konrad Zuse in seinem ersten Rechner Z1 eingeführt. Bis in die 70er Jahre hinein hatte praktisch jede Rechnerarchitektur ihre eigenen Gleitkommaformate (so gab es Gleitkommazahlen mit 24, 32, 36, 48, 60, 64, 96 und 128 Bits Länge). Datenbestände waren so nur schwer austauschbar. Noch problematischer war folgendes: Naturgemäß müssen gleiche Operationen mit gleichen Zahlenwerten zu gleichen Ergebnissen führen. Das ist jedoch keineswegs selbstverständlich, wenn diese Zahlen in verschiedenen Gleitkommaformaten dargestellt werden (Wertebereiche, Rundungsfehler). Deshalb wurde im Rahmen des IEEE (eines US-amerikanischen Normenausschusses) versucht, einen einheitlichen Standard für Gleitkommazahlen zu schaffen. Dieser Standard war in der Version IEEE 754-85 ausgereift. Er liegt praktisch allen modernen Architekturen zugrunde.

Gemäß IEEE 754-85 sind Gleitkommazahlen einfacher, doppelter, erweiterter und vierfacher Genauigkeit vorgesehen (Single Precision, Double Precision, Extended Precision, Quad Precision Reals). Sie sind 32, 64, 80 bzw. 128 Bits lang (Abbildung 1.22, Tabelle 1.5).

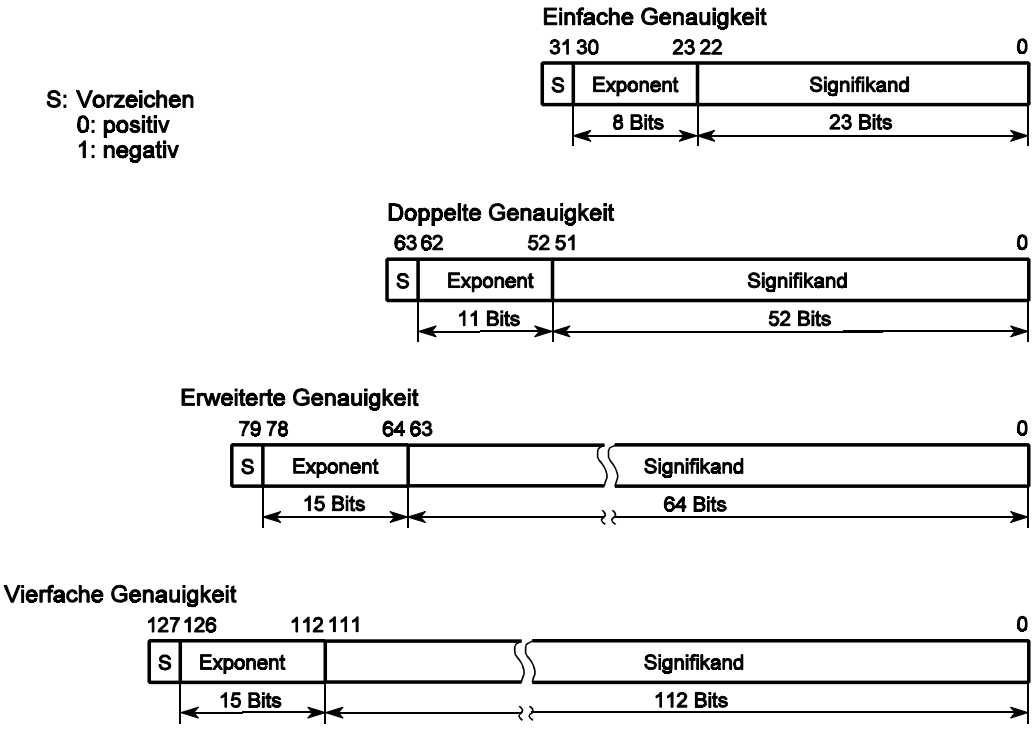


Abbildung 1.22 Gleitkommaformate nach IEEE 754-85

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): führende Eins wird *nicht* mitgespeichert; 2): führende Eins wird mitgespeichert

Tabelle 1.5 Parameter der Gleitkommaformate nach IEEE 754-85

Hinweise:

1. Nicht alle Architekturen unterstützen alle Formate.
2. IA-32 unterstützt nur 32, 64 und 80 Bits, wobei intern alle Formate in das 80-Bit-Format gewandelt werden (Ausnahme: SIMD-Befehle, z. B. 3DNow und SSE).
3. Typische RISC-Architekturen (Mips, Sparc, Alpha, PowerPC) unterstützen nur die einfache und doppelte Genauigkeit (32 und 64 Bits).
4. 80-Bit-Gleitkommazahlen werden - namentlich in 32-Bit- oder 64-Bit-Maschinen - gelegentlich in "Behältern" gespeichert, die 96 oder 128 Bits lang sind, also Vielfachen der Maschinen-Wortlänge entsprechen.

Sonderprobleme:

- Gleitkommadarstellungen sind nicht immer exakte Repräsentationen für numerische Werte,
- es ist mit Rundungsfehlern zu rechnen,
- die absolute Genauigkeit nimmt ab, je größer die darzustellenden Zahlen werden,
- die Darstellung sehr kleiner Zahlen (solcher, die nahe bei Null liegen) ist problematisch.

Um den besonderen Situationen gerecht zu werden, die bei numerischen Berechnungen auftreten können, sind Sonderdarstellungen für gewisse numerische Werte vorgesehen. Das betrifft:

- denormalisierte Zahlen (zur Darstellung sehr kleiner Zahlenwerte),
- Nullen,
- Werte für plus/minus Unendlich,
- unzulässige bzw. undefinierte Werte. Fachbegriff: NaN = Not a Number (wörtl. "keine (gültige) Zahl").

Derartige Angaben sind in besonderer Weise in den Gleitkommaformaten codiert. Sie sind teils vorgesehen, um die Rechengenauigkeit zu verbessern, teils um reagieren zu können, wenn während des Rechnens ungewöhnliche Bedingungen eintreten.

1.3. Nichtnumerische Datenstrukturen

1.3.1. Elementarstrukturen

Die elementaren Strukturen von 8, 16, 32 und 64 Bits Länge sind nicht nur als beliebig nutzbare Behälter anzusehen, sondern auch als Datentypen, die bestimmten Operationen unterzogen werden können (Verschiebeoperationen, bitweisen logischen Verknüpfungen usw.).

Kettendaten (Strings)
Eine Kette (String) ist eine zusammenhängende Folge sog. String-Elemente (das können Bytes, Worte, Doppelworte usw. sein). Solche Datenstrukturen erfordern folgende Bestimmungen:

- eine Adresse,
- eine Längenangabe für die gesamte Kette. Möglichkeiten: Anzahl der String--Elemente, Anzahl der Bytes, Adresse des letzten String-Elementes bzw. Bytes, besondere Endekennzeichnung. Üblicherweise wird die Länge in Bytes angegeben.
- eine Längenangabe für das einzelne String-Element (8, 16, 32 Bits usw.).

Strings sind in den meisten Architekturen *keine* eigenständigen Datenstrukturen, sondern sie entstehen, indem aufeinanderfolgende Bytes durch Befehle entsprechend interpretiert

werden. Bei Nutzung solcher Stringbefehle werden die beschreibenden Angaben des String (Adresse, Länge) aus allgemeinen Registern oder direkt aus dem Befehl selbst entnommen. Beispiele:

1. in den Architekturen S/370.../390 sind drei Arten von Bytestrings vorgesehen, die sich in ihrer maximalen Länge unterscheiden (1...16 Bytes, 1...256 Bytes, gesamter Speicheradreßraum). Befehle, die für die ersten beiden Arten gelten, enthalten Längen- und Adressierungsangaben als Direktwerte. Längenangaben und Adressen von Zeichenketten der dritten Art werden in Universalregistern erwartet.
2. in der x86- bzw. IA-32-Architektur werden Adresse und Gesamtlänge der Strings in fest zugeordneten Registern erwartet. Der Stringbefehl bestimmt die Länge des String-Elements (Byte, Wort, Doppelwort).

Bitketten (Bitstrings), Bitfelder, Einzelbits

Der höchste Grad an Flexibilität ist erreicht, wenn jedes einzelne Bit direkt adressiert werden kann und wenn nicht nur fest formatierte, sondern wahlfreie, beliebig lange Aneinanderreihungen von Bits (Bitstrings; Abbildung 1.23) verarbeitet werden können. Entsprechende Vorschläge (und auch tatsächlich ausgeführte Maschinen) gibt es seit den 60er Jahren. Weshalb haben sie sich nicht durchgesetzt? - Dafür gibt es zwei wesentliche Gründe: (1) Verarbeitungsleistung, (2) Kompliziertheit der Architektur.

Verarbeitungsleistung

Um eine hohe Leistung zu erreichen, müssen mehrere Bits parallel transportiert und verarbeitet werden. Eine solche Hardware kann man gar nicht anders bauen als mit einer festen Verarbeitungsbreite. Alle breiteren Datenstrukturen brauchen dann entsprechend mehrere Verarbeitungsschritte, und alle Datenstrukturen, die im Speicher die Verarbeitungsbreite überlappen (vgl. Abbildung 1.23), erfordern mehrere Zugriffe. Diese Zugriffe kosten aber Verarbeitungsleistung.

Kompliziertheit der Architektur

Wenn es beispielsweise nur vier fest formatierte elementare Datenstrukturen gibt, so braucht man zur Formatangabe im Befehl höchstens zwei Bits. Wahlfreie Aneinanderreihungen von Bits erfordern aber stets (1) eine Bitadresse (um 3 Bits länger als die entsprechende Byteadresse) sowie (2) eine Längenangabe (5...32 Bits und mehr).

Deshalb hat sich das Prinzip in allgemeiner Form nicht durchsetzen können. Moderne Architekturen enthalten aber Befehle, um elementare Operationen mit adressierten Einzelbits, mit Bitfeldern und Bitstrings (Bitketten) zu unterstützen. Ein *Bitfeld* ist eine Aneinanderreihung, die höchstens soviele Bits umfaßt wie die Verarbeitungsbreite angibt (z. B. 32). Bitketten können demgegenüber beträchtlich länger sein.

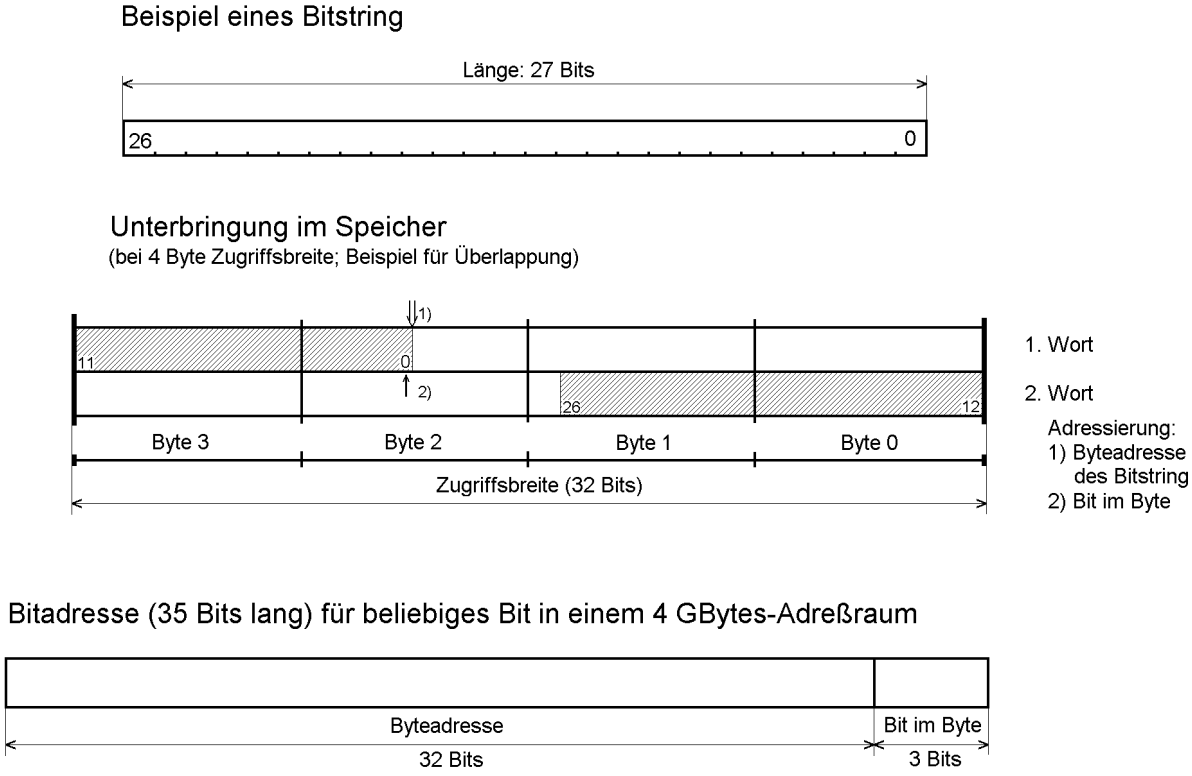


Abbildung 1.23 Bitkette (Bitstring)

Beispiel: IA-32
Es sind lediglich Einzelbitbefehle vorgesehen, die einzelne Bits wahlweise in einem allgemeinen Register bzw. im Speicher adressieren. Die Auswahlangabe für ein Bit im Speicher ist maximal 32 Bits lang. Man kann damit ein Bit in einem Bereich von 4 GBits (512 MBytes) adressieren, der seinerseits durch eine Byteadresse ausgewählt wird.

1.3.2. Alphanumerische Zeichen

"Alphanumerisch" ist der übliche Sammelbegriff für Buchstaben, Ziffern und Sonderzeichen. Es ist gar nicht schwierig, einen Zeichencode zu entwickeln: wir schreiben alle Zeichen, die wir darstellen wollen, neben- oder untereinander (wir stellen also - in der Redeweise des Mathematikers - unseren Zeichenvorrat als geordnete Menge dar). Die laufende Nummer des einzelnen Zeichens in dieser Menge (mathematisch: dessen Ordinalzahl) bildet nun - als natürliche Binärzahl dargestellt - den jeweiligen Zeichencode (0 entspricht dem 1. Zeichen, 1 dem zweiten Zeichen usw.).

Die Zeichencodes unterscheiden sich somit lediglich in folgendem:

- welche Zeichen gehören zum Zeichenvorrat?
- wieviele Bits werden zur Codierung verwendet?
- welcher Code wird welchem Zeichen zugewiesen?

Zeichencodes werden zumeist in Tabellen- oder Listenform dargestellt. Ein Zeichen wird beim heutigen Stand der Technik typischerweise in einem Byte oder in einem 16-Bit-Wort codiert. (Früher waren u. a. 6-Bit-Codes üblich. Der klassische Fernschreibcode ist ein 5-Bit-Code.)

Zeichenketten (Character Strings, z. B. Worte oder Sätze) sind Aneinanderreihungen von Zeichencodes. Viele Architekturen haben Befehle, die Operationen mit einzelnen Zeichen und mit Zeichenketten unterstützen.

Ungültige, freie, reservierte Codes
Mit n Bits können wir insgesamt 2^n verschiedene Zeichen codieren. Meist ist der Zeichenvorrat aber kleiner. Es ist dann eine Ermessensfrage, wie die ungenutzten Codes interpretiert werden (typischerweise werden sie für künftige Erweiterungen reserviert).

ASCII und ANSI
ASCII = American Standard Code for Information Interchange (sprich: Asskieh). Der ursprüngliche ASCII-Code ist an sich ein 7-Bit-Code (Wertebereich 0...127 bzw. 00H ...7FH). In Computern entspricht aber ein Zeichen einem Byte; dabei ist das höchstwertige Bit stets Null. Es ist lediglich der Wertebereich von 20H bis 7EH mit darstellbaren Zeichen belegt (maximal 95 verschiedene Zeichen).

Der erweiterte IBM-Zeichensatz
Mit dem PC hat IBM einen erweiterten Zeichensatz eingeführt, in dem die verbleibenden Belegungen ausgenutzt werden, um zusätzliche Zeichen zu codieren (Abbildung 1.24).

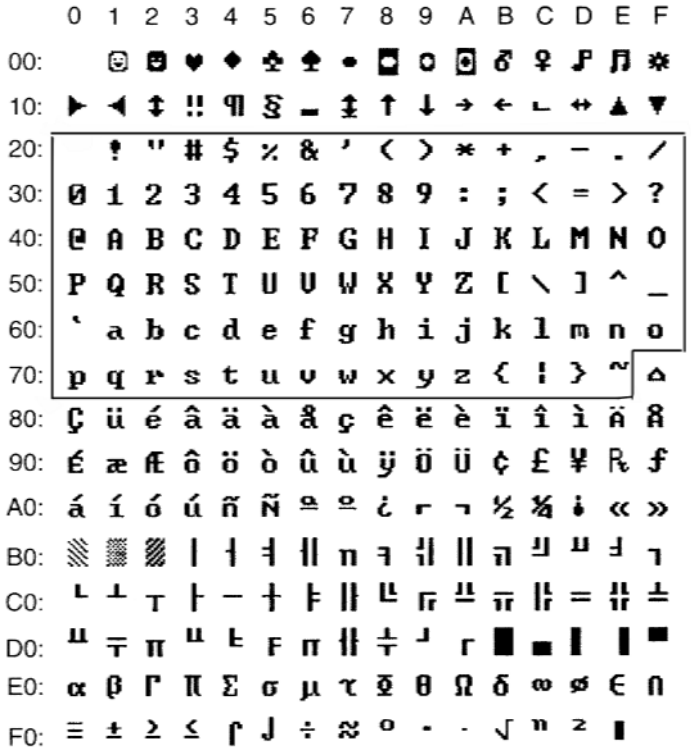


Abbildung 1.24 Der erweiterte IBM-Zeichensatz (Codetabelle). Eingerahmt: der ursprüngliche Zeichenvorrat nach ASCII

Erklärung zu Abbildung 1.24:
Die Spalte entspricht dem niederwertigen, die Zeile dem höherwertigen Halbbyte. Beispiel: Zeichen "d": Zeile = 60H + Spalte = 4H = 64H (Übungsbeispiele in Anhang 2). 20H ist der Code des Leerzeichens (engl. Space; sprich: Spehs). Die Codes 20H...7EH entsprechen dem ursprünglichen ASCII-Code (in der Abbildung durch Einrahmung gekennzeichnet).

Landesspezifische Zeichensätze - Codeseiten (Code Pages)
Es gibt verschiedene Abwandlungen des erweiterten Zeichensatzes. Diese werden als Codeseiten (Code Pages) bezeichnet.

Hinweis:
Die jeweils gewünschte Codeseite wird in der Konfigurationsdatei CONFIG.SYS angegeben (in der COUNTRY-Anweisung). In Deutschland wird typischerweise die Codeseite 850 bevorzugt.

Der ANSI-Zeichensatz
ANSI = American National Standards Institute. Auch ASCII ist an sich ein ANSI-Standard (ANSI X3.4). Microsoft bezeichnet aber den für Windows gewählten 8-Bit-Zeichensatz pauschal als ANSI-Zeichensatz (obwohl er eigentlich auf dem Standard ISO/IEC 8859-1 beruht).

ASCII und Windows-ANSI:

- die Zeichencodes 20H...7EH sind in beiden Codes gleichartig belegt,
- im ANSI-Zeichensatz fehlen (vgl. Abbildung 1.24): die griechischen Buchstaben (α, β usw.), die mathematischen Symbole (±, ≤ usw.) sowie die Sonderzeichen zum Zusammensetzen einfacher graphischer Darstellungen (nicht mehr erforderlich, weil Windows eine "richtige" Graphikschnittstelle (GDI) hat).

Weitere herkömmliche Zeichencodes
Rechenzentrums-Maschinen (Mainframes) verwenden typischerweise den EBCDIC-Code, der je Zeichen ebenfalls ein Byte vorsieht. Zur Datenkommunikation sind weitere Zeichencodes in Gebrauch (u. a. gemäß ITU/CCITT).

Unicode
Für erweiterte Zeichensätze (weitere Alphabete, verschiedene Schriftarten usw.) braucht man mehr als 8 Bits. Unicode (sprich: Juhnikohd) ist ein internationaler Standard, der je Zeichen 16 Bits vorsieht.

Wichtige Merkmale im Überblick:

- es ist ein reiner Zeichencode; jeder 16-Bit-Wert kann praktisch als Ordinalzahl angesehen werden, die aus der Menge aller Zeichen genau eines auswählt. Zeichenformen bzw. Schriftarten, Schriftgrößen usw. werden hierbei *nicht* codiert.
- das Ziel besteht darin, jedem auf der Erde gebrauchten Schriftzeichen ein Codewort zuzuordnen; es gibt kein Umschalten zwischen Zeichensätzen usw. Beispielsweise betrifft Unicode Version 2.0 genau 38 885 Zeichen.

- jedes der Unicode-Zeichen hat eine standardisierte Bezeichnung in Englisch (wichtig, um sich ohne Kenntnis der jeweiligen Sprache und ohne graphische Zeichendarstellung verständigen zu können - denken wir nur an chinesische, hebräische, arabische usw. Zeichen).
- die $2^{16} = 65\,536$ Codes sind in verschiedene Bereiche aufgeteilt (Tabelle 1.6).
- die ersten 127 Codes (0000H...007FH) entsprechen, Spitzfindigkeiten beiseite gelassen, den ASCII-Codes 00H...7FH.

Bezeichnung	Inhalt	Codebereich
allgemeine Schriften (General Scripts)	enthält 4096 Zeichen, darunter lateinische, griechische, hebräische und arabische Schriftzeichen	0000...1FFF
Symbole (Symbols Area)	enthält 4096 Zeichen, darunter Formelzeichen der Mathematik und der Chemie, Währungszeichen, Interpunktionszeichen, Symbole (Dingbats) usw.	2000...2FFF
Chinesisch, Japanisch, Koreanisch (CJK): Symbole und phonetische Zeichen (CJK Symbols Area)	enthält 1024 Zeichen, die den einzelnen Sprachen zugeordnet sind	3000...33FF
Chinesisch, Japanisch, Koreanisch (CJK): Bildzeichen (CJK Ideographics Area)	enthält 20 902 Schriftzeichen	4E00...9FFF
Koreanisch: Hangul-Silbenzeichen (Hangul Syllables Area)	enthält 11 172 Schriftzeichen	AC00...D7A3
Erweiterung (Surrogates Area)	enthält 2 048 Codes, die für künftige Erweiterungen vorgesehen sind	D800...DFFF
privat (Private User Area)	ermöglicht es, bis zu 6 400 spezifische Zeichen zu codieren	E000...F8FF
Sonderzeichen (Compatibility and Specials Area)	enthält praktisch den “Rest”, der sich woanders nicht unterbringen ließ (1792 Zeichen)	F900...FFFF

Tabelle 1.6 Unicode (Übersicht)

Zeichencodes im PC:

- DOS verwendet ASCII,
- Windows 3.x verwendet ANSI,
- die höher entwickelten Windows-Versionen verwenden Unicode, unterstützen aber auch ANSI.

1.3.3. Datenstrukturen für graphische Darstellungen

Eine leistungsfähige graphische Bildschirmanzeige gehört zu den wichtigsten Ausstattungsmerkmalen eines Personalcomputers. Die anzuzeigenden Bilder werden

üblicherweise als *Punktraster* gespeichert. Die einzelnen Bildpunkte werden *Pixel* genannt. Hochauflösende Graphikdarstellungen bestehen aus entsprechend vielen Pixels - und die müssen nicht nur gespeichert, sondern auch erzeugt bzw. verarbeitet werden. Deshalb liegt es nahe, solche Datenstrukturen in der Architektur vorzusehen. Geht es um höchstes Leistungsvermögen, so sind zwei Arten von Angaben zu unterstützen (sie sind unabhängig voneinander nutzbar): die eigentlichen Pixel- und die z-Puffer-Angaben.

Pixel-Angaben
Je Pixel ist ein Helligkeits- bzw. Farbwert zu speichern. Hierfür hat man im Laufe der Entwicklungsgeschichte 1, 2, 4, 8, 16, 24 oder 32 Bits vorgesehen. Die Belegung der einzelnen Bits in den Pixel-Angaben ist an sich dem Programmierer freigestellt. In der Praxis wird sie aber zumeist durch die jeweilige videoseitige Farbcodierung bestimmt (Abbildungen 1.25).

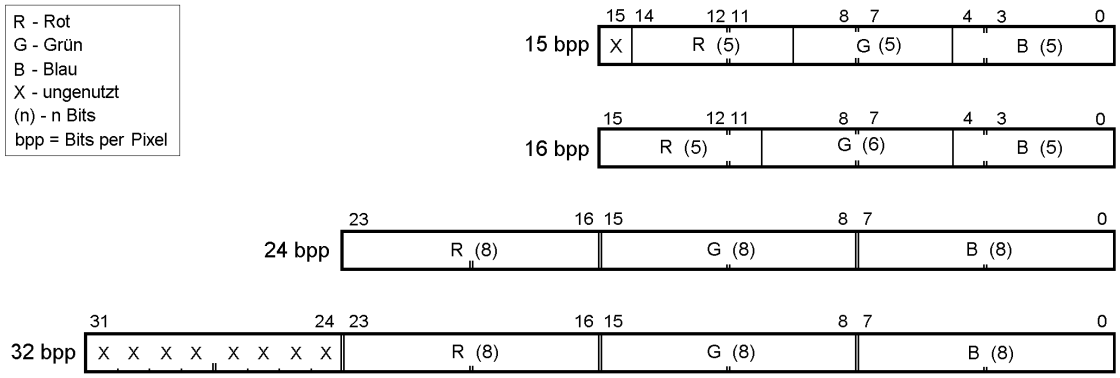


Abbildung 1.25 Pixel-Codierungen bei höheren Farbtiefen (nach Microsoft)

Zur Bildspeicherorganisation
Es gibt zwei Organisationsformen:

1. Bitebenen-Organisation (Bitmapped oder Planar- Organization)
Der Bildspeicher ist in Bitebenen (Bit Planes) aufgeteilt. Zu den n Bits des einzelnen Pixels trägt jede Ebene mit einem Bit bei.

2. Gepackte Organisation (Packed Organization)
Alle Bits, die zu einem Pixel gehören, werden zusammenhängend gespeichert.

Beide Organisationsformen wollen wir anhand des herkömmlichen VGA-Videostandards kurz erläutern.

Die Organisationsweisen des VGA-Bildspeichers
Der VGA-Bildspeicher besteht grundsätzlich aus 4 Ebenen, wobei je Ebene wenigstens 64 kBytes vorgesehen sind. Jede Ebene ist byteweise organisiert. In den graphischen Betriebsarten liefern die Ebenen entweder einzelne Bits, aus denen die Pixelangaben zusammengesetzt werden (Bitebenen-Organisation, Abbildung 1.26), oder die Ebenen enthalten aufeinanderfolgende Bytes, die jeweils 1 Pixel repräsentieren (gepackte Organisation, Abbildung 1.27).

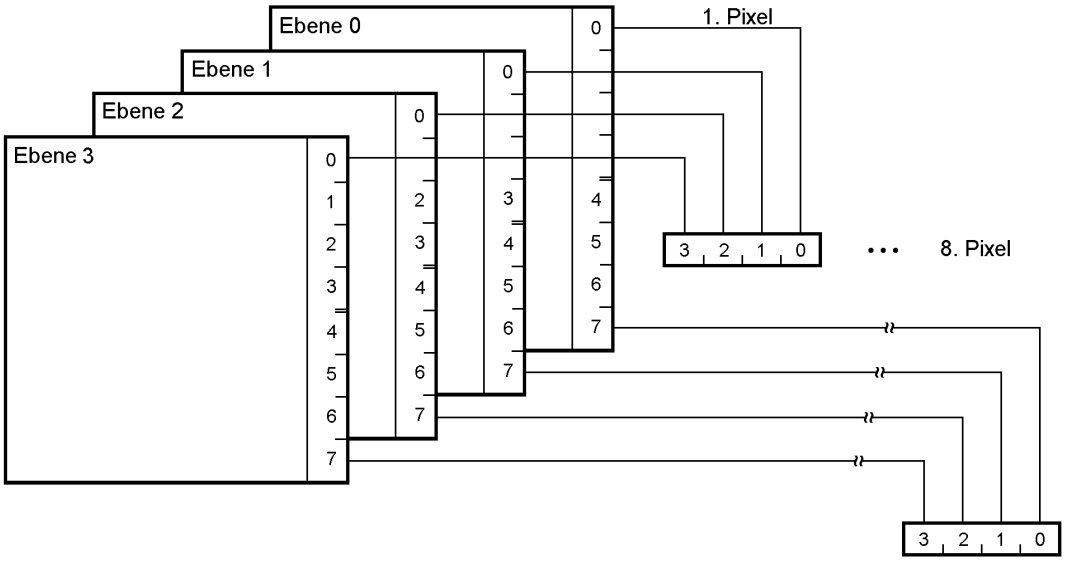


Abbildung 1.26 Bitebenen-Organisation (am Beispiel der VGA-Betriebsart 12H)

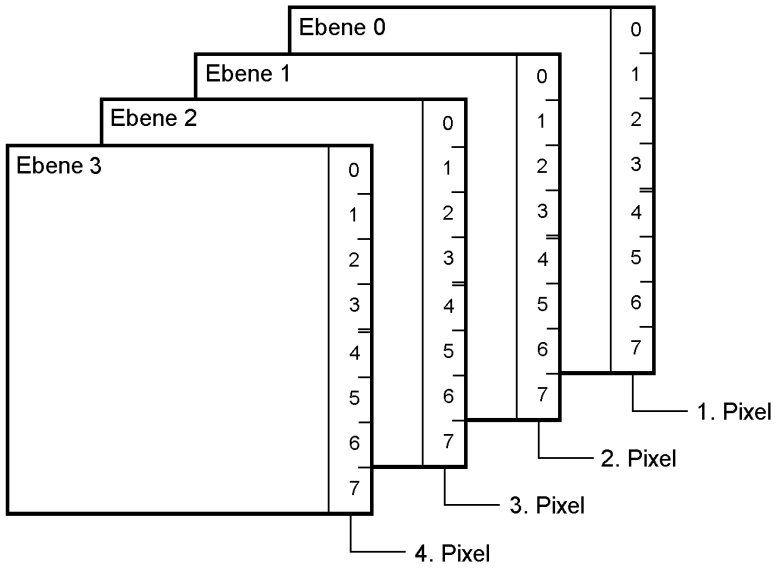


Abbildung 1.27 Gepackte Organisation (am Beispiel der VGA-Betriebsart 13H)

z-Puffer-Angaben

Der z-Puffer ist für dreidimensionale (3D) Darstellungen von Bedeutung. Bei der perspektivischen Wiedergabe dreidimensionaler Darstellungen ist jedem Pixel ein Wert zugeordnet, der dessen "Tiefe im Raum" (die z-Koordinate) angibt. Auf der (zweidimensionalen) Bildschirmdarstellung dürfen nur jene Pixel erscheinen, die im Raum "am weitesten vorn" liegen (die dahinter liegenden werden verdeckt). Wenn ein Punkt im dreidimensionalen Raum ermittelt wurde, muß deshalb geprüft werden, ob auf der entsprechenden Position der zweidimensionalen Bildebene nicht bereits ein Punkt dargestellt ist, der - aus der Position des (vom Programm angenommenen) Betrachters gesehen - weiter vorn liegt. Abbildung 1.28 veranschaulicht das Problem.

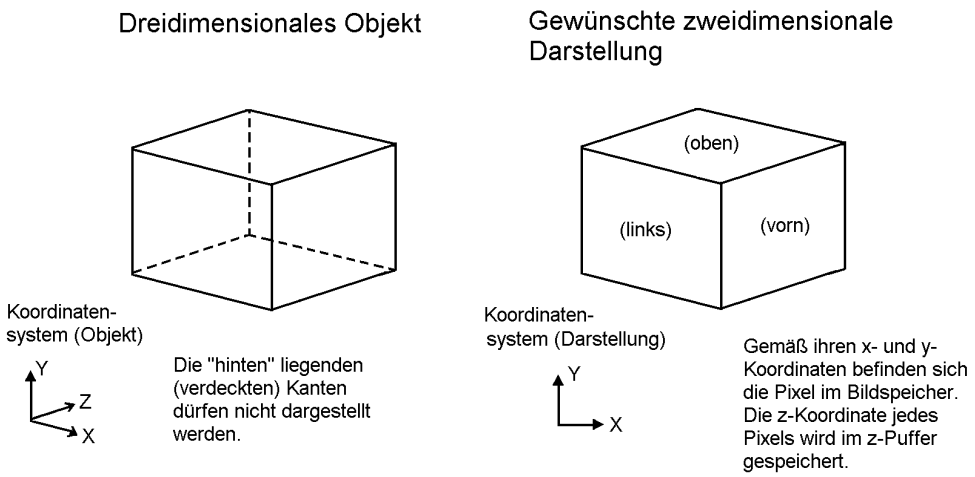


Abbildung 1.28 Der Zweck des z-Puffers - die perspektivische Darstellung dreidimensionaler Objekte zu unterstützen

Um dies zu ermöglichen, kann jedem Pixel eine z-Puffer-Angabe zugeordnet werden. Typische z-Puffer-Angaben sind 16 oder 32 Bits lang. Sie werden als natürliche (vorzeichenlose) Binärzahlen oder als Gleitkommazahlen interpretiert. Manche Graphikkarten haben einen eingebauten z-Puffer nebst den erforderlichen Verarbeitungsschaltungen. Auch liegt es nahe, die SIMD-Erweiterungen der modernen Hochleistungsprozessoren auszunutzen (Abbildung 1.29).

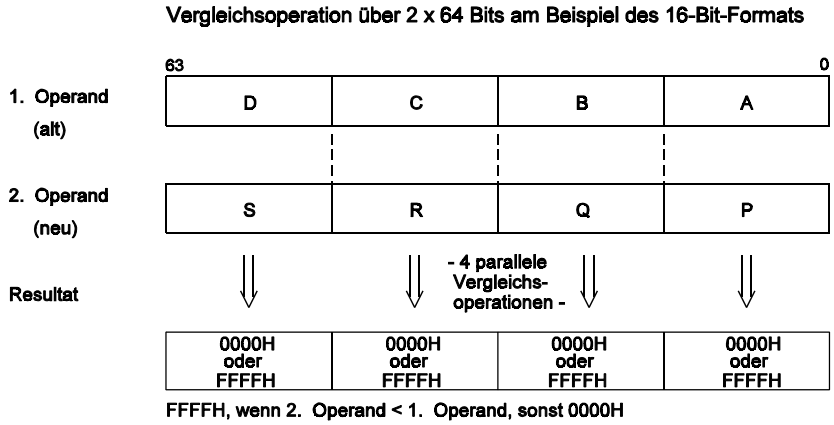


Abbildung 1.29 z-Puffer-Angaben und deren Verarbeitung am Beispiel der MMX-Erweiterung

Erklärung:
Bei einer Verarbeitungsbreite von 64 Bits werden vier z-Puffer-Angaben zu 16 Bits (wie in der Abbildung gezeigt) oder zwei zu 32 Bits verarbeitet, das heißt, 4 oder 2 alte z-Puffer-Werte werden mit 4 oder 2 neuen verglichen. Dabei entsteht folgendes Resultat:

- alle Positionen, in denen die neuen Werte kleiner sind als die alten[*)], enthalten Einsen (Belegung FF...FH),
- die verbleibenden Positionen enthalten Nullen (Belegung 00...0H).

Dieses Ergebnis kann z. B. von anschließenden Pixel-Transportabläufen ausgewertet werden, um nur die weiter vorn liegenden Pixel in den Bildspeicher zu schaffen.

*): in der Computergraphik bevorzugt man ein Koordinatensystem, dessen z-Achse nach hinten zeigt. Je kleiner also der z-Puffer-Wert, desto weiter vorn liegt das Pixel.

1.4. SIMD-Datenstrukturen

SIMD = Single Instruction, Multiple Data. Dies bedeutet, daß mit einem einzigen Maschinenbefehl mehrere Operanden gleichzeitig verarbeitet werden, wobei typischerweise mehrere Ergebnisse entstehen. Beim derzeitigen Stand der Technik werden gleichartige, in entsprechend großen "Behältern" gemeinsam verpackte Operanden gleichzeitig einer einzigen Operation unterzogen. Im folgenden beschreiben wir typische Datenstrukturen anhand einschlägiger Erweiterungen der IA-32-Architektur (MMX, 3DNow, SSE). Allgemeinbegriffe: gepackte (packed) Daten (Intel), (kurze) Vektoren (Motorola).

Die MMX-Erweiterung
Als universelle Behälter dienen 64-Bit-Worte (Quadwords), die mehrere gleichartige Datenstrukturen aufnehmen können (Abbildung 1.30).

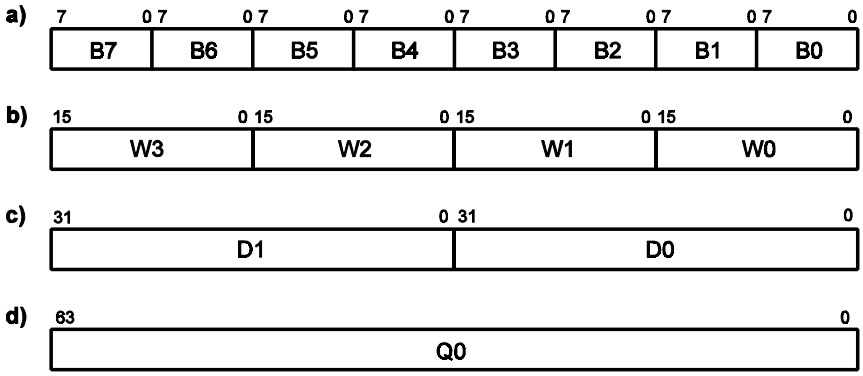


Abbildung 1.30 Datenstrukturen der MMX-Erweiterung

Erklärung:
Ein 64-Bit-Wort kann wahlweise enthalten:

a) 8 Bytes (B7...B0),
b) 4 16-Bit-Worte (W3...W0),
c) 2 32-Bit-Worte (Doppelworte; D1, D0),
d) 1 64-Bit-Quadwort (Q0).

Diese Datenstrukturen können folgendermaßen interpretiert werden:

- als einfache Aneinanderreihungen von Bits (z. B. Pixel-Daten),
- als natürliche (vorzeichenlose) Binärzahlen,

- als ganze (vorzeichenbehaftete) Binärzahlen in 2er-Komplement-Darstellung.

Die 3DNow-Erweiterung (AMD)

Ergänzend zu den Datenstrukturen der MMX-Erweiterung sind in einem 64-Bit-Wort zwei 32-Bit-Gleitkommazahlen untergebracht (Abbildung 1.31). Gleitkommaformat: einfache Genauigkeit gemäß IEEE754-85.

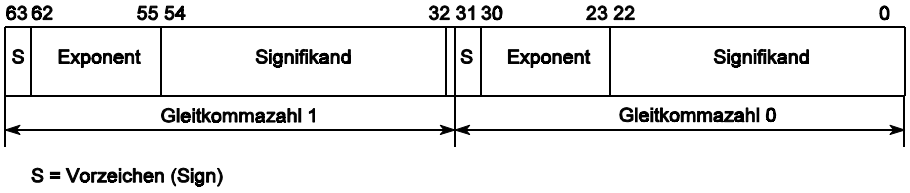


Abbildung 1.31 Datenstruktur der 3DNow-Erweiterung

Die SSE-Erweiterungen (Intel)

Die SSE-Erweiterung der P6-Prozessoren (SSE1)

SSE = SIMD Streaming Extensions. Der Behälter ist ein 128-Bit-Wort, das 4 32-Bit-Gleitkommazahlen aufnehmen kann (Abbildung 1.32). Gleitkommaformat: einfache Genauigkeit gemäß IEEE754-85.

SSE2

Die 128-Bit-Worte können weitere Datenstrukturen aufnehmen (Abbildung 1.33).

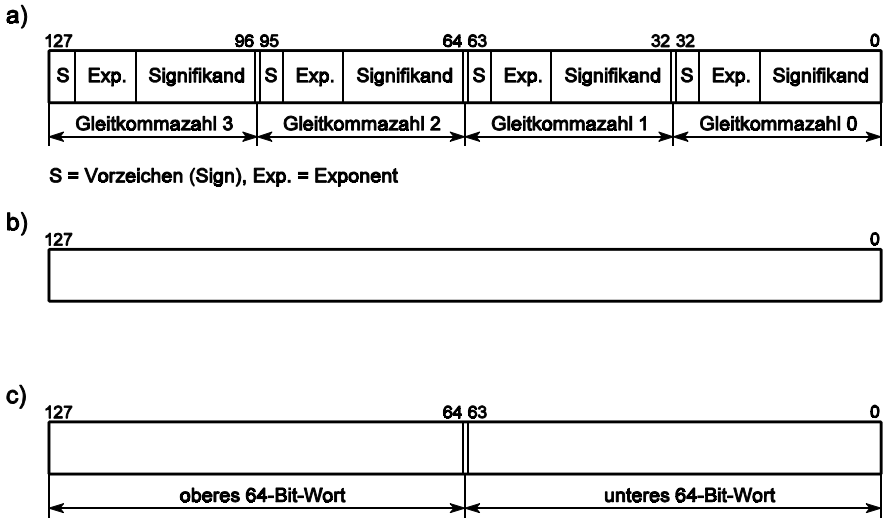


Abbildung 1.32 Datenstrukturen gemäß SSE1

Erklärung zu Abbildung 1.32:

a) - 128-Bit-Wort enthält 4 Gleitkommazahlen zu 32 Bits; b) - 128-Bit-Wort aus Sicht der bitweisen Verknüpfungen (Und, Oder, Und-Nicht, Exklusiv-Oder); c) - 128-Bit-Wort aus Sicht der Transportbefehle, die 64-Bit-Worte bewegen.

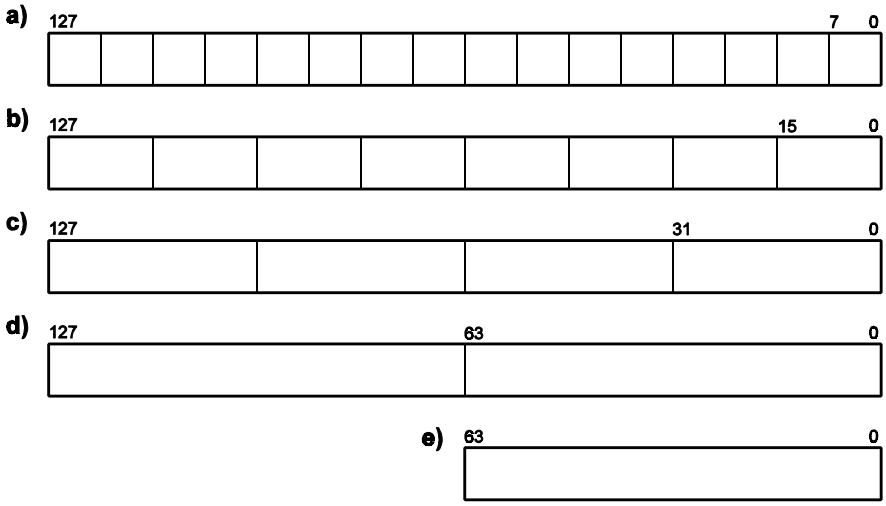


Abbildung 1.33 Zusätzliche Datenstrukturen (SSE2)

Erklärung:

a) 16 8-Bit-Binärzahlen mit/ohne Vorzeichen (Signed/Unsigned Byte Integers),
b) 8 16-Bit-Binärzahlen mit/ohne Vorzeichen (Signed/Unsigned Word Integers),
c) 4 32-Bit-Binärzahlen mit/ohne Vorzeichen (Signed/Unsigned Doubleword Integers),
d) 2 64-Bit-Binärzahlen mit/ohne Vorzeichen (Signed/Unsigned Quadword Integers) oder 2 64-Bit Gleitkommazahlen gemäß IEEE 754-85 (vgl.Abbildung 1.22),
e) eine einzelne 64-Bit-Binärzahl mit/ohne Vorzeichen (Signed/Unsigned Quadword Integer).

Die AltiVec-Erweiterung der PowerPC-Prozessoren (Motorola)
Es sind 128-Bit-Strukturen vorgesehen. Ein 128-Bit-Wort (Vektor) kann jeweils enthalten:

- 16 8-Bit-Binärzahlen mit/ohne Vorzeichen (Signed/Unsigned Byte Integers),
- 8 16-Bit-Binärzahlen mit/ohne Vorzeichen (Signed/Unsigned Word Integers),
- 4 32-Bit-Binärzahlen mit/ohne Vorzeichen (Signed/Unsigned Doubleword Integers),
- 4 32-Bit-Gleitkommazahlen gemäß IEEE754-85.

Diese Strukturen sind grundsätzlich so aufgebaut wie in den Abbildungen 1.32a und 1.33a...c dargestellt (Motorola bevorzugt aber die Linksadressierung (Big Endian); somit befindet sich Bit 0 links außen und Bit 127 rechts außen).

1.5. Strukturen beschreibender Angaben

1.5.1. Deskriptoren - eine Einführung

Ein Deskriptor ist im allgemeinen eine Informationsstruktur, die eine andere nach Lage (Adresse), Größe (Länge), Typ, Status und Nutzungsrechten beschreibt (Abbildung 1.34). Solche Strukturen werden in komplexen Programmsystemen (z. B. in den Laufzeitsystemen höherer Programmiersprachen) ausgiebig verwendet. Mit Deskriptoren kann man objektorientierte Zugriffsweisen verwirklichen. Dabei werden die einzelnen Informationsstrukturen nicht direkt adressiert, sondern über eine Objektkennung (Object Identifier; sprich: Obbdscheckt Identifeir) aufgerufen. Dieser ist gleichsam die laufende Nummer (die Ordinalzahl) des einzelnen Objekts in der Menge aller Objekte. Auf diese Weise entfällt die Vielzahl der Adreßrechnungen, die ansonsten in Anwendungsprogrammen notwendig ist; das Anwendungsprogramm muß nicht einmal die Adresse des einzelnen Objekts kennen. Abbildung 1.35 veranschaulicht diese Zugriffsweise in der Gegenüberstellung mit der üblichen Adressierung von Datenstrukturen.

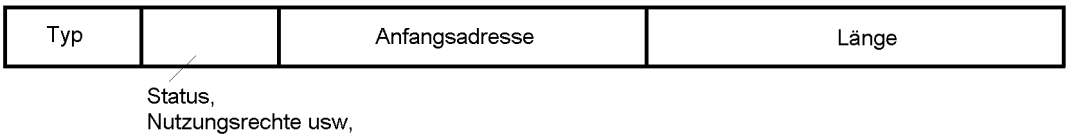


Abbildung 1.34 Inhalt eines Deskriptors

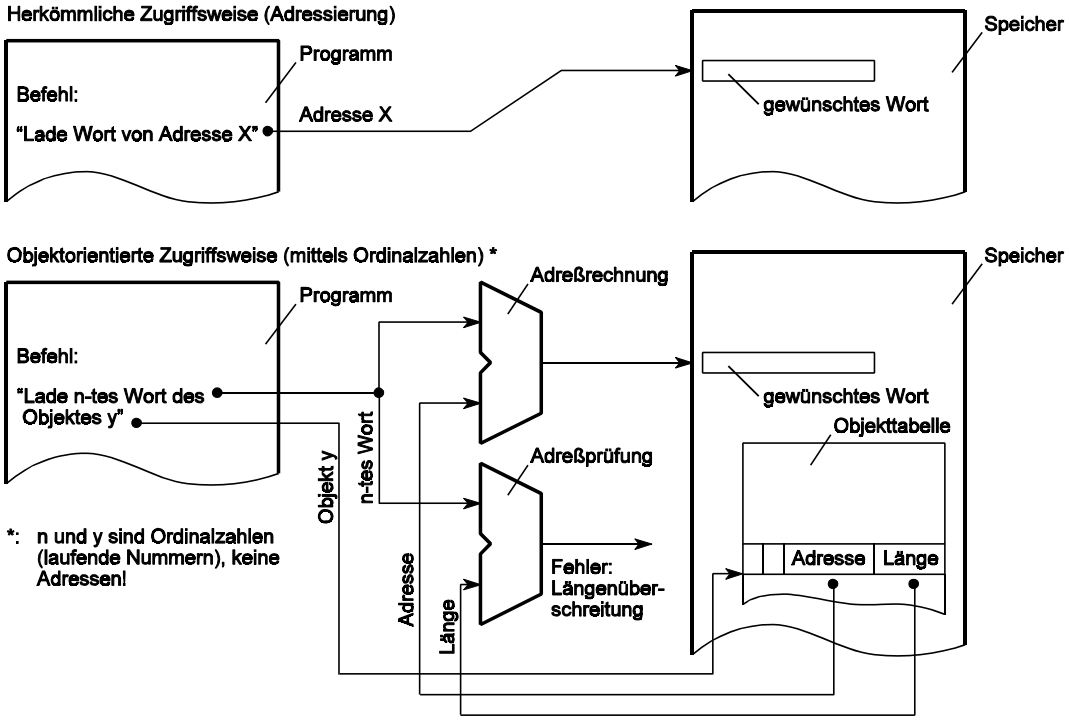


Abbildung 1.35 Herkömmliche und objektorientierte Zugriffsweise

Erklärung zu Abbildung 1.35:

Um auf das n-teWort des Objekts zuzugreifen, wird zunächst der Deskriptor des Objekts y aus der Objekttabelle geholt. Dieser gibt die Anfangsadresse des Objekts und dessen Länge an. Der Wert n wird zur Anfangsadresse addiert, um die Adresse des gewünschten n-ten Wortes zu errechnen (Adreßrechnung). Ist n zu groß (n > Längenangabe im Deskriptor), wird ein Fehler gemeldet (Adreßprüfung).

Zur Enwicklungsgeschichte
Das Deskriptorkonzept wurde bereits Anfang der 60er Jahre in den Rechenanlagen der Fa. Burroughs am Markt wirksam (B 5000...B 6700). Im Mikroprozessor iAPX 432 (Intel, um 1980) wurde es in geradezu exzessiver Weise der gesamten Architektur zugrunde gelegt. Dieser Prozessor war allerdings nicht sonderlich erfolgreich. Weshalb? Die Antwort können Sie ohne weiteres aus Abbildung 1.35 ablesen: zu geringe Leistung!

- bei der herkömmlichen Adressierung sind zwar gelegentlich programmseitige Adreßrechnungen erforderlich, ist die Adresse aber bekannt, kann man sofort zu den Daten zugreifen.
- hingegen müssen beim objektorientierten Prinzip jedem Datenzugriff Deskriptorzugriffe vorangehen. Um dabei Leistungseinbußen gering zu halten bzw. zu vermeiden, muß diese Zugriffsweise ganz massiv durch Hardware unterstützt werden.

Überblick über die deskriptiven Angaben (IA-32)
Die in der Architektur IA-32 vorgesehenen deskriptiven Angaben werden nachfolgend kurz erläutert. Sie sind zu folgenden Zwecken vorgesehen:

- Adreßzeiger und Segmentselektoren dienen zur Adressierung und Segmentauswahl,
- Segmentdeskriptoren beschreiben Segmente,
- Gate-Deskriptoren geben Eintrittspunkte an (zum Eintritt in Programme bzw. Tasks),
- Taskzustandssegmente (TSS) nehmen Task-Umgebungen auf,
- Deskriptortabellen enthalten alle Deskriptoren, die für die laufende Arbeit benötigt werden,
- Seitenverzeichnisse und Seitentabellen sind für die Seitenverwaltung notwendig.

Hinweise:

1. Wenn wir uns im Zusammenhang mit den deskriptiven Strukturen auf IA-32 konzentrieren, so ist das keine sonderliche Einschränkung. Solche Informationsstrukturen sind immer notwendig, wenn einschlägige Konzepte verwirklicht werden sollen; sie kommen in praktisch allen hochleistungsfähigen Computersystemen vor (sind aber oft softwaremäßig implementiert).
2. Wir beschreiben zunächst nur die Strukturen, wobei wir Spitzfindigkeiten, Erweiterungen usw. vernachlässigen. Wozu manche dieser Strukturen dienen, wird womöglich erst später klarwerden. Sie sollten deshalb den Rest des Kapitels 1 zunächst nur des Überblicks halber durchlesen und dann, wenn die funktionellen Zusammenhänge erläutert werden, gelegentlich wieder hier nachschlagen.

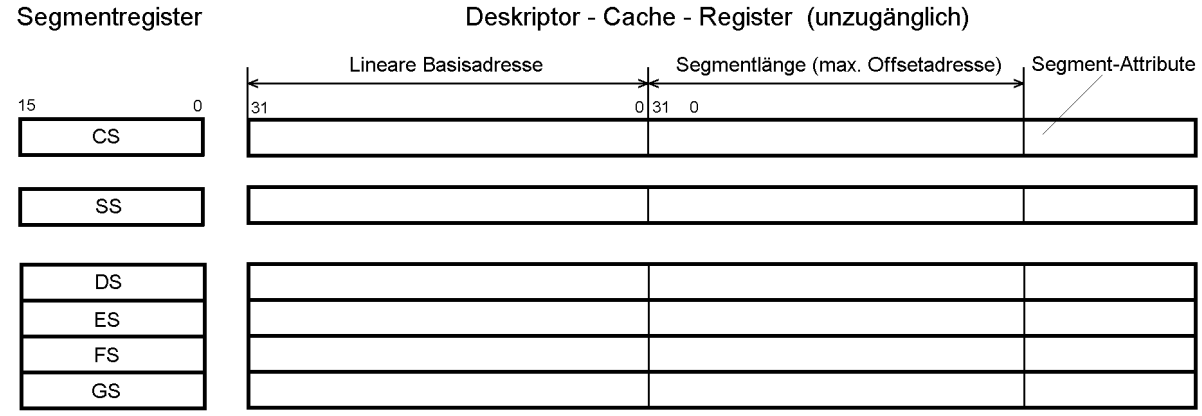
1.5.2. Deskriptoren in der Architektur IA-32

Deskriptoren wurden vom 80286 an eingeführt - allerdings in einer etwas unausgereiften Form. Da der 286 heutzutage nicht mehr unterstützt wird, genügt es, daß wir uns auf die - mit dem 386 eingeführte - voll ausgebaute IA-32-Architektur beschränken (Abbildungen 1.36...1.40).

IA-32 ist nicht für die objektorientierte Verarbeitung im eigentlichen Sinne ausgelegt. Vielmehr dienen die Deskriptoren dazu, verschiedene Modelle der Speicheradressierung verwirklichen zu können, die Abwärtskompatibilität zu gewährleisten und elementare Schutzfunktionen vorzusehen.

Wir merken uns:
In der Architektur IA-32 beschreiben die Deskriptoren Segmente (Speicherbereiche) und keine Objekte. Indem die Nutzungsweise derart beschränkt wurde, konnten Leistungseinbußen mit erträglichem Hardware-Aufwand vermieden werden.

Segmentzugriffe
Es sind 6 Segmentregister vorgesehen. Die Segmentregister sind um zusätzliche, dem Programmierer unzugängliche Deskriptor-Cache-Register erweitert (Abbildung 1.36). Wird ein Segmentregister geladen, so holt die Hardware automatisch den betreffenden Deskriptor aus dem Speicher und bringt ihn in das zugehörige Deskriptor-Cache-Register. Die beschreibenden Angaben stehen somit bei allen Datenzugriffen in der Prozessor-Hardware bereit, so daß die Nutzung des Deskriptorprinzips nicht zur Leistungsminderung führt.



CS: aktuelles Programmsegment
SS: Stacksegment
DS....GS: Datensegmente

Einzelheit der Segment - Attribute:

Anwesenheit	Privilegebene	Zugriffs-anzeige	Längen-einheit	Erweiterungs-richtung	Leseerlaubnis	Schreib-erlaubnis	Ausführungs-erlaubnis	Stacklänge	Privileg-anpassung

Abbildung 1.36 IA-32-Segmentregister

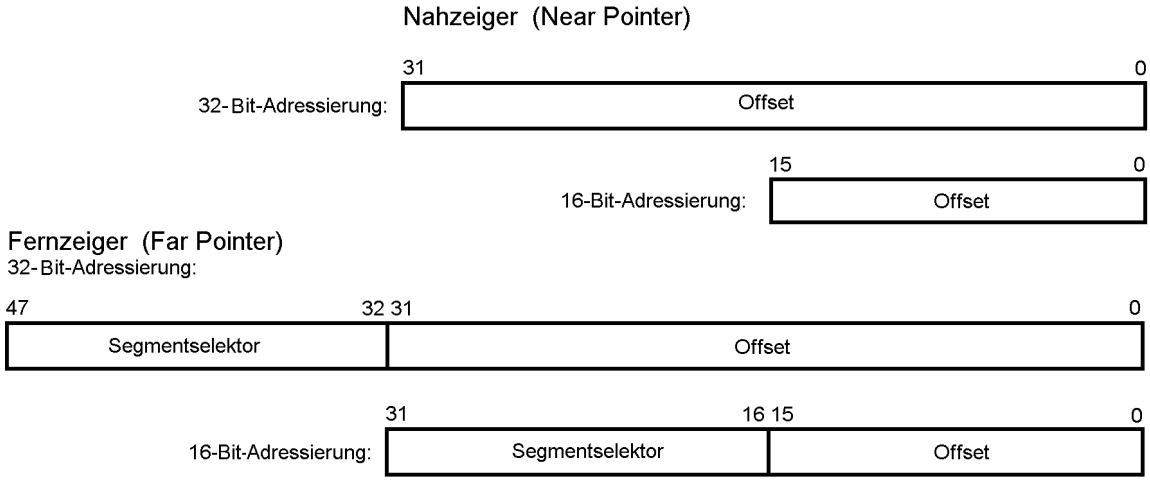


Abbildung 1.37 Adreßzeiger

Erklärung:
Es sind 2 Formate definiert. Der *Nahzeiger* (Near Pointer) gibt eine Byteadresse innerhalb eines Segmentes an. Seine Länge entspricht der Adreßlänge (32 bzw. 16 Bits). Der *Fernzeiger* (Far Pointer) hat eine Länge von 48 bzw. 32 Bits, wobei eine Byteadresse (32 bzw. 16 Bits) durch einen 16-Bit-Segmentselektor ergänzt ist, mit dem ein Segment direkt ausgewählt wird.

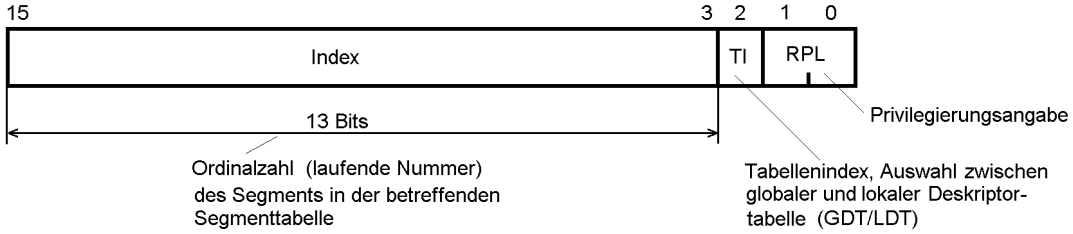


Abbildung 1.38 Segmentselektor

Erklärung:
Der Segmentselektor enthält die Ordinalzahl des gewünschten Segmentes, d. h. dessen laufende Nummer in einer Deskriptortabelle. Eine Deskriptortabelle umfaßt bis zu 8192 (8k) Deskriptoren, und der Segmentselektor gestattet es, eine von zwei Deskriptortabellen (globale bzw. lokale) auszuwählen. Ein Segmentselektor mit Belegung Null (Nullselektor) kennzeichnet, daß kein Segment ausgewählt wird.

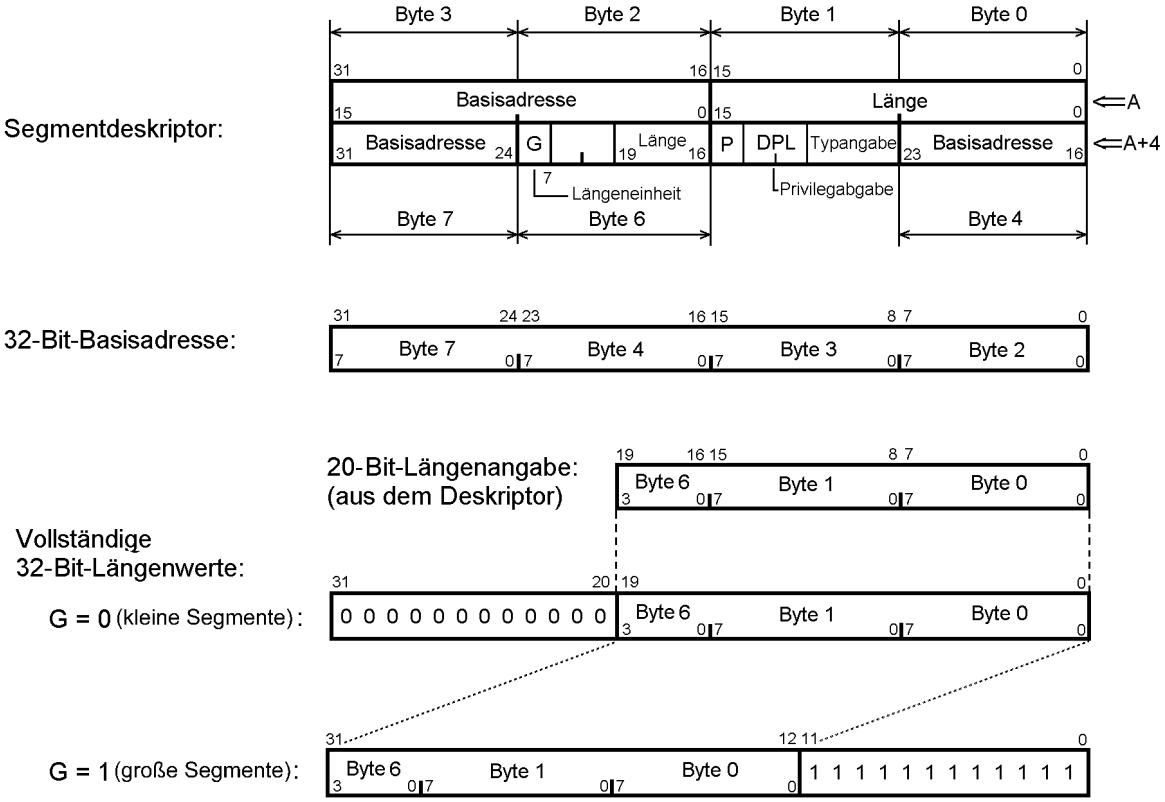


Abbildung 1.39 Segmentdeskriptor

Erklärung:
Segmentdeskriptoren enthalten die Basisadresse und die Länge des betreffenden Segments sowie weitere beschreibende Angaben (Typ, Zugriffsrechte usw.). Ein Segmentdeskriptor beschreibt entweder ein Programmsegment, ein Datensegment bzw. Stacksegment, ein Taskzustandssegment (TSS) oder eine lokale Deskriptortabelle (LDT).

Der Segmentdeskriptor ist an sich gemäß Abbildung 1.34 aufgebaut, nur sind die einzelnen Angaben in einer etwas merkwürdigen Weise auf die 8 Bytes des Deskriptorformats verteilt. In der Abbildung oben: das tatsächliche Format. Darunter: so werden Basisadresse und Länge aus den einzelnen Feldern des Deskriptorformats gleichsam zusammengestükkelt. Die Basisadresse bestimmt die Lage des Segments im linearen Speicheradreßraum. Die Länge kleiner Segmente (bis 1 MBytes) ist bis aufs Byte angebbar. Darüber hinaus können Segmente bis zu 4 GBytes groß sein. Die Länge solcher Segmente wird in Stufen zu 4 kBytes angegeben (Steuerung über Bit G (Granularity)). Befindet sich ein Segment nicht im Speicher, so wird dies im Deskriptor angezeigt (das Anwesenheitsbit P (Presence) ist gelöscht), und die weitaus meisten Bits des Deskriptorformats sind für das Betriebssystem verfügbar, um entsprechende Angaben über den Ort des Segments unterzubringen (z. B. um die Position des Segments auf einem Plattenspeicher zu beschreiben).

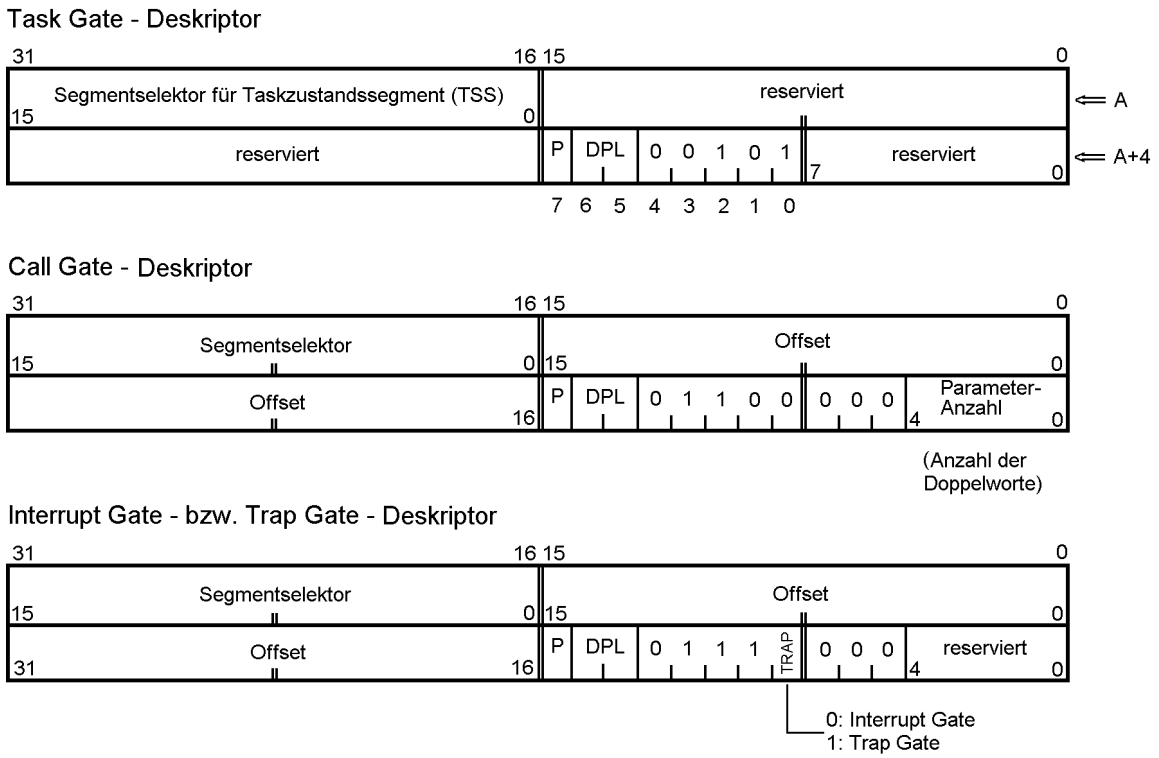


Abbildung 1.40 Gate-Deskriptoren

Erklärung:
Gates sind Eintrittspunkte in Programme. Sie stellen besondere Schutzvorkehrungen für den Programmaufruf bereit. Es sind folgende Arten von Gates vorgesehen:

- *Call Gates* dienen zum Aufruf von Unterprogrammen. Sie enthalten einen Selektor für das betreffende Programmsegment sowie die Adresse des ersten auszuführenden Befehls.
- *Trap Gates* und *Interrupt Gates* veranlassen den Aufruf von Programmen zur Unterbrechungsbehandlung. Ein Interrupt Gate verhindert automatisch nachfolgende Unterbrechungen, ein Trap Gate nicht. Die Deskriptoren sind ähnlich formatiert wie die der Call Gates.
- *Task Gates* dienen zur Steuerung des Multitasking. Zu jeder Task gehört ein *Taskzustandssegment* (Task State Segment TSS), in dem die aktuelle Information über den Zustand der Task untergebracht ist. Der Task-Gate-Deskriptor enthält einen Selektor für das betreffende TSS.

Für jede Art ist jeweils ein Deskriptorformat definiert. Die Deskriptoren von Call-, Trap- und Interrupt-Gates beschreiben an sich Sprungziele, und zwar mit zwei Angaben:

1. das jeweilige Programmsegment (Segmentselektor),
2. die Anfangsadresse im betreffenden Programmsegment (Offset).

Ein Task Gate bewirkt eine Taskumschaltung, wobei die eigentlichen Programmstartangaben im ausgewählten Taskzustandssegment (TSS; Abschnitt 1.5.4.) stehen.

1.5.3. Deskriptortabellen (IA-32)

Deskriptortabellen sind Listen von Deskriptoren. Es sind drei Arten von Deskriptortabellen vorgesehen (Abbildung 1.41):

- eine globale Deskriptortabelle GDT,
- lokale Deskriptortabellen LDT (bedarfsweise),
- eine Interruptdeskriptortabelle IDT.

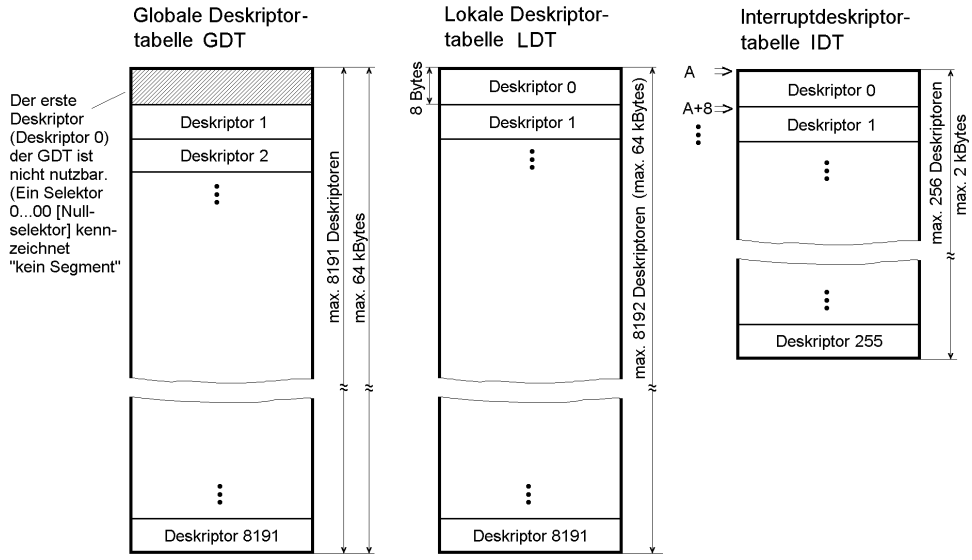


Abbildung 1.41 Deskriptortabellen

Die globalen und lokalen Deskriptortabellen GDT, LDT können bis zu 8k Deskriptoren umfassen (GDT: 8k - 1, da Segmentselektor 0 bedeutet, daß gar kein Segment ausgewählt wird - demzufolge ist dieser Eintrag nicht nutzbar). Die Tabellen können Segmentdeskriptoren, Call-Gate-Deskriptoren und Task-Gate-Deskriptoren enthalten. Die Interruptdeskriptortabelle IDT umfaßt bis zu 256 Deskriptoren gemäß den maximal 256 verschiedenen Unterbrechungs- bzw. Ausnahmebedingungen, die in der Architektur vorgesehen sind. Sie kann Task Gate-, Interrupt Gate- oder Trap Gate-Deskriptoren enthalten, aber keine Segmentdeskriptoren.

1.5.4. Taskzustandssegmente (IA-32)

IA-32 unterstützt das Multitasking. Das heißt, es können mehrere unabhängige Programme in einem Prozessor zeitversetzt abgearbeitet werden. Natürlich kann nur jeweils ein Programm den Prozessor aktiv nutzen. Um die Zustände der einzelnen Programme (Tasks) aus dem Prozessor holen, im Speicher bereithalten und zwecks Verarbeitung einstellen zu können, ist für jede Task ein Taskzustandssegment (Task State Segment TSS) vorgesehen (Abbildung 1.42). Dieses ist fest formatiert, um die grundlegenden Zustandsangaben aufzunehmen (vor allem Registerinhalte; vgl. Abbildung 1.42 mit Abbildung 9.3), kann aber um frei nutzbare Bereiche erweitert werden.

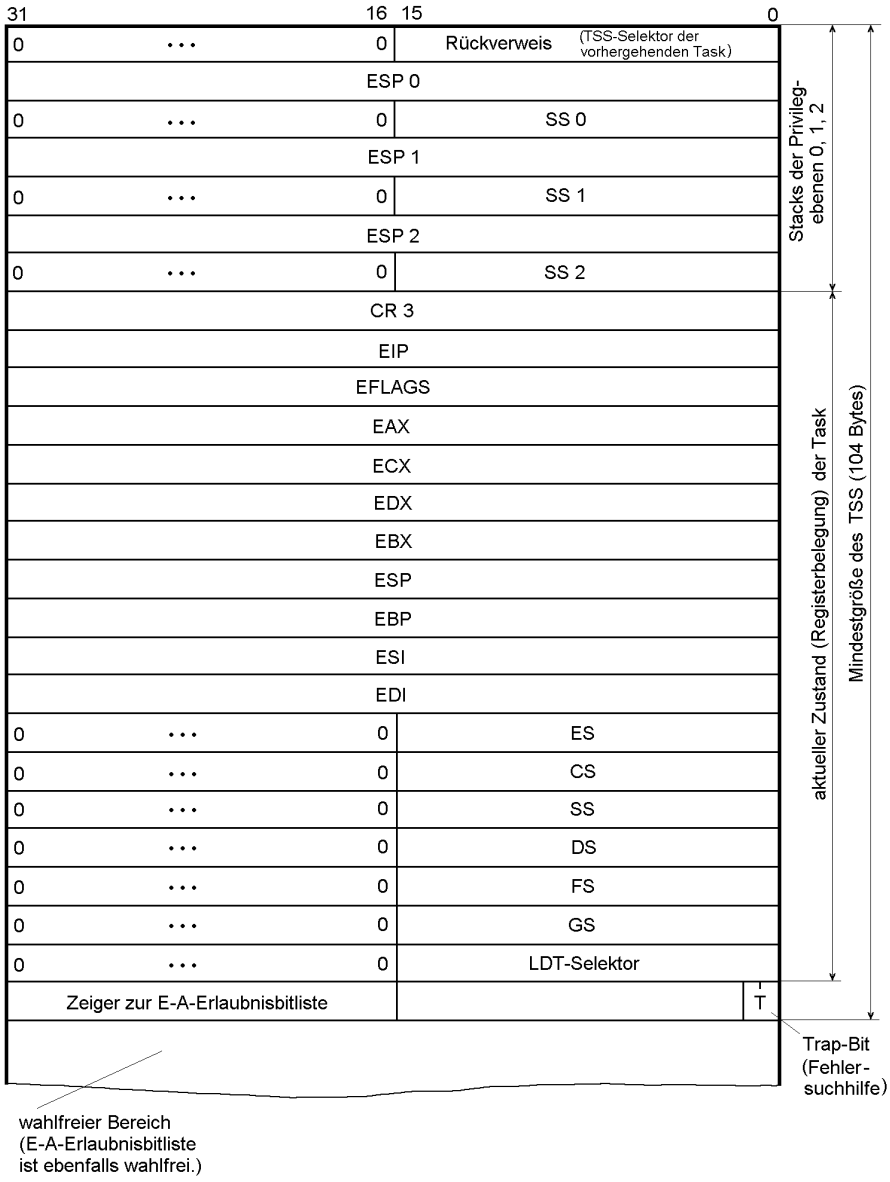


Abbildung 1.42 Taskzustandssegment (TSS)

1.5.5. Informationsstrukturen der Seitenverwaltung (IA-32)

Eine *Seite* (Page) hat normalerweise eine Größe von 4 kBytes. Zur Aufnahme von Seiten ist der physische Speicher in 4 kBytes große *Seitenrahmen* (Page Frames) unterteilt (Abbildung 1.43). Die Lage der Seite im Speicher wird im Rahmen eines zweistufigen Verzeichnissystems beschrieben, das aus einem *Seitentabellenverzeichnis* (Page Directory) und einer gewissen Zahl von *Seitentabellen* (Page Tables) besteht. Jeder Verzeichniseintrag hat eine Länge von 4 Bytes und enthält die Adresse des Seitenrahmens, in dem die betreffende Seite untergebracht ist. Befindet sich die Seite nicht im Speicher, so kann der zugehörige Verzeichniseintrag entsprechende Lokalisierungs- bzw. Steuerangaben aufnehmen (Abbildung 1.44).

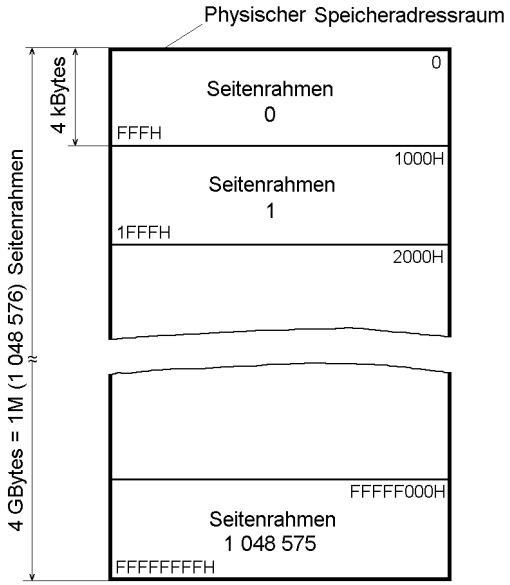


Abbildung 1.43 Seitenrahmen im physischen Speicher

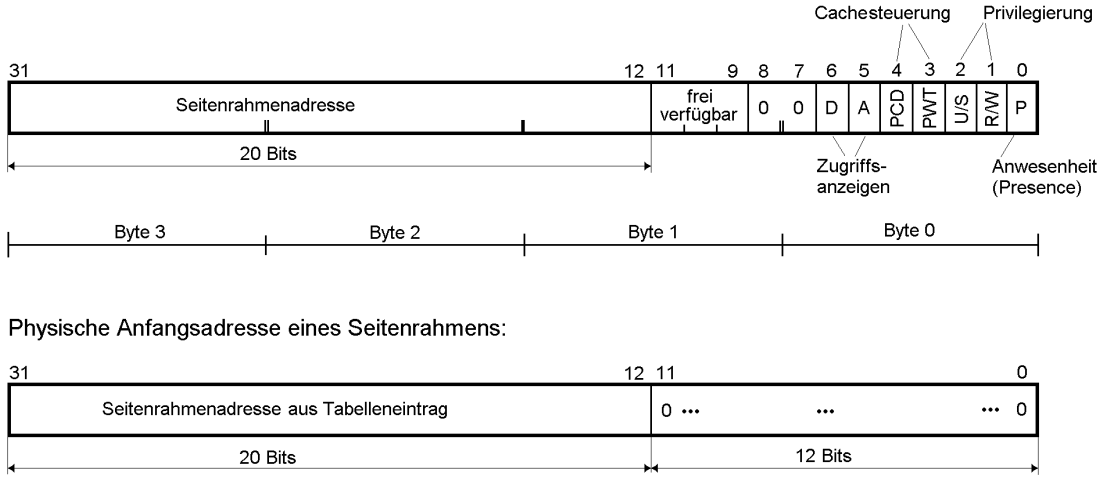


Abbildung 1.44 Tabelleneinträge der Seitenverwaltung

Hinweise:

1. Seiten und Seitenrahmen sind fest an integralen 4k-Adressen (Abschnitt 2.6.2.) angeordnet; die niederwertigen 12 Bits der Anfangsadresse sind immer gleich Null. Somit müssen in den Verzeichniseinträgen nur die höherwertigen 20 Adreßbits untergebracht werden.
2. Zu Besonderheiten (längere Seiten von 2 oder 4 MBytes, Adreßverlängerung usw.) siehe Abschnitt 3.5.5.

2. Maschinenbefehle

2.1. Die Befehlsliste

Was leistet ein Befehl?
Befehle (Instructions; sprich: Inns-tracktschns)) sind gespeicherte binär codierte Angaben, die den Verarbeitungsablauf steuern. In den klassischen Rechnerarchitekturen ist vorgesehen, daß Befehl für Befehl aus dem Speicher geholt und ausgeführt wird. Die Befehlsausführung bestimmt:

- die Auswahl der zu verarbeitenden Daten,
- die Art der Informationswandlungen,
- die Reihenfolge der Verarbeitungsschritte,
- die Zuweisung der Verarbeitungsergebnisse.

Die Befehlsliste
Die Befehlsliste ist das Verzeichnis aller Befehle einer bestimmten Rechnerarchitektur (Befehlsvorrat; Instruction Set). Eine deutliche Vorstellung davon zu haben, welche Befehle in solchen Listen enthalten sind, gehört zum allgemeinen Fachwissen - auch dann, wenn man in der Praxis nie programmiert oder nur mit höheren Programmiersprachen arbeitet.

Zum Einstieg:

- Tabelle 2.1 enthält eine Übersicht über Befehlswirkungen, die in modernen Architekturen vorgesehen sind,
- Kapitel 9 enthält Kurzdarstellungen des Befehlsvorrats verschiedener Architekturen,
- in Anhang 1 finden Sie die Befehlsliste unseres fiktiven Prozessors P/F (der eigens zu Lehrzwecken entwickelt wurde).

Wie umfangreich soll eine Befehlsliste sein?
Das ist eine Streitfrage seit es Computer gibt. Grundsätzlich soll es sich um eine *universelle* Befehlsliste handeln, mit der man wirklich *alles* programmieren kann.

Dafür reichen schon sehr wenige Befehlswirkungen aus:

- Addition von Binärzahlen,
- bitweise wirkende logische Verknüpfungen (UND, ODER),
- bitweise Negation (0 → 1, 1 → 0),
- bedingte Verzweigungen auf Ausgangsübertrag (Carry Flag), Überlauf (Overflow Flag) und “Ergebnis = Null" (Zero Flag),
- Unterprogrammruf und Rückkehr,
- Transporte (Laden = Operanden aus dem Speicher holen, Speichern = Ergebnisse in den Speicher schreiben).

Das ist alles. Und es dürfte dem Fachmann für Berechenbarkeit und Algorithmentheorie noch viel zuviel sein*). Die Vorstellung, mit möglichst wenigen Befehlswirkungen auskommen zu müssen, ist aber in der heutigen Praxis bedeutungslos. Auch "RISC" bedeutet nicht "die Befehlsliste so kurz wie möglich", sondern "die Befehlsliste nicht umfangreicher und die Befehlswirkungen nicht komplexer als seitens der Anwendungspraxis nötig". So haben auch neuentwickelte Architekturen mit "reduzierter" Befehlsliste durchaus um die hundert verschiedene Befehle (und mehr)**). Die Befehlslisten der modernen Architekturen sind in algorithmischer Hinsicht vollkommen ausreichend (damit läßt sich wirklich "alles" programmieren), und sie sind, namentlich was die neueren Entwicklungen angeht, auf der Grundlage umfangreicher statistischer Untersuchungen 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.).

*): der theoretische Grenzfall: die sog. Turing-Maschinen (nach dem englischen Mathematiker Alan Turing). Es handelt sich um eine Art fiktiver Prozessoren, die tatsächlich nur einfachste Transport- und Entscheidungsabläufe (nach dem Schema “wenn - dann”) ausführen können. Sie sind in der Praxis unbrauchbar, aber in der mathematischen Grundlagenforschung von größter Bedeutung.

**): Beispiel PowerPC: Grundbefehlsvorrat: über 180 Befehle, hinzu kommen über 160 AltiVec-Befehle (SIMD-Verarbeitung).

Wann hängt die Entscheidung (für den Einsatz einer bestimmten Architektur bzw. eines bestimmten Prozessors) von Einzelheiten der Befehlsliste ab? - Dafür gibt es heutzutage an sich nur ein Szenarium: es sind extreme Anforderungen zu erfüllen - und dazu ist auf Maschinenebene zu programmieren. Zudem kommt es darauf an, die Hardware bis zum letzten auszunutzen. So etwas dürfte eigentlich nur bei Embedded Systems zutreffen. (Hier ist keineswegs nur von absoluten Höchstleistungen die Rede. So wird man, wenn es etwa um die Steuerung eines Staubsaugers geht, es zu schätzen wissen, daß der Entwicklungsingenieur durch geschickte Auswahl des Prozessortyps und durch dessen trickreiche Ausnutzung 0,30 € je Staubsauger eingespart hat.) Bei Personalcomputern, Servern usw. sollte man sich hingegen nicht allzu sehr von den Vorzügen der Befehlsliste (und auch der Architektur allgemein) leiten lassen; auch die beste Architektur nützt nichts, wenn es keine Software gibt, die sie wirklich ausnutzt.

Operationsbefehle							
Binärzahlen		Gleitkomma-zahlen	Dezimalzahlen (BCD)	adressierbare Behälter (Bytes, Worte usw.)	Zeichenketten	Bitketten, Bitfelder	Einzelbit
natürliche	ganze						
Addieren Subtrahieren Vergleichen		Addieren Subtrahieren Multiplizieren	nicht unterstützt *bzw.* Dezimalkorrektur (Hilfsbefehle)	UND ODER NICHT	Auffüllen Ausschneiden Einfügen	Bereitstellen (rechtsbündig) Einfügen (aufs Bit adressiert)	Abfragen Setzen Löschen
Multiplizieren Dividieren Verschieben	Multiplizieren Dividieren Verschieben (arithmetisch) Vorzeichen-wechsel	Dividieren Vergleichen Betrag Vorzeichen-wechsel Wandeln (Konvertieren) weitere mathematische Funktionen, wie $\sqrt{x}$, sin x usw.	*bzw. volle Unterstützung:* Addieren Subtrahieren Multiplizieren Dividieren Vergleichen Wandeln (Konvertieren)	Exklusiv-ODER Vergleichen (logisch) Verschieben / Rotieren	Vergleichen Durchsuchen über Tabelle wandeln	Position der niedrigstwertigen Eins Position der höchstwertigen Eins Anzahl der Einsen	Wechseln

Die Tabelle wird auf der nächsten Seite fortgesetzt.

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
Unterbrechungs-steuerung	Steuerung der Speicherverwaltung	Sonderzustände ein-leiten	Rückkehr aus Supervisor-zustand
Hilfsbefehle für Test- und Fehlersuchzwecke			

Tabelle 2.1 Befehlswirkungen in modernen Prozessor-Architekturen (Übersicht)

2.2. Befehlstypen

2.2.1. Wie ist ein Befehl aufgebaut?

Der Maschinenbefehl ist eigentlich nur eine fest formatierte Aneinanderreihung von Bits, wobei je nach Befehlstyp bestimmte Bitfelder bestimmte Bedeutungen haben (Abbildung 2.1). In den Einzelheiten unterscheiden sich die Befehlsformate der verschiedenen Architekturen bisweilen beträchtlich voneinander (Beispiele: in Kapitel 9).

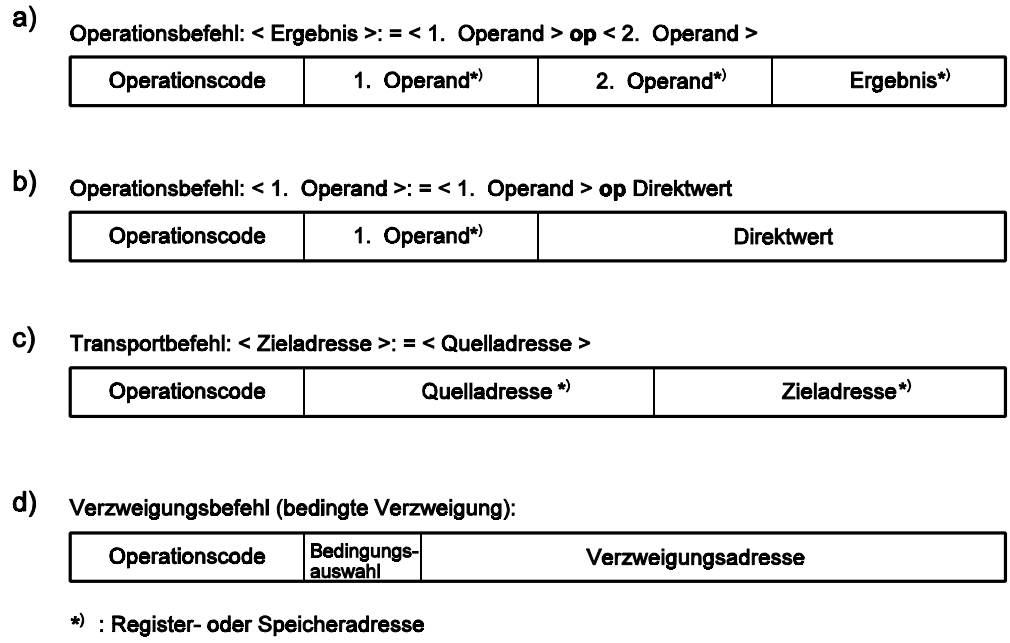


Abbildung 2.1 Befehlsformate (Beispiele in Prinzipdarstellung). Erklärung der Symbolik in Abschnitt 2.4.2.

2.2.2. Operationsbefehle

Grundlage der Verarbeitung bilden die Operandenverknüpfungen oder Operationen. Es sind nur vergleichsweise wenige und recht elementare Operationen vorgesehen. Alle vom Anwender gewünschten Funktionen, Gebrauchseigenschaften, Informationswandlungen usw. müssen auf Folgen solcher Operationen zurückgeführt werden (das ist im Grunde genommen die Aufgabe jeglicher Programmierung).

Operationen
In üblichen Rechnerarchitekturen bestimmt *ein* Befehl nur *eine* Operation*). Der Befehl muß somit eine Kennzeichnung enthalten, welche Operation auszuführen ist. Diese Angabe heißt *Operationscode* (Opcode).

*): Ausnahme: die sog. VLIW-Architekturen.

Operanden
Jede Operation wirkt auf Operanden. Die Anzahl der Operanden ist gering; sie liegt zwischen 1 und 3. Die meisten Befehle haben zwei Operanden (das ist beispielsweise bei den vier Grundrechenarten der Fall). Um eine Operation auszuführen, braucht die Hardware von jedem Operanden (1) dessen Format und (2) dessen Wert.

Operandenformate
Ist nur ein einziges Operandenformat vorgesehen, so ergibt sich dies implizit. Das bedeutet, daß es keine besondere codierte Angabe gibt, sondern daß die Hardware von sich aus dafür sorgt*), daß die als Operand gelieferte Aneinanderreihung von Bits dem betreffenden Format entspricht.

*): mit anderen Worten: sie ist genau dafür aufgebaut (z. B. als 16-Bit-Maschine, als 32-Bit-Maschine usw.).

Sind hingegen mehrere Operandenformate zulässig (z. B. Bytes oder Worte, Binärzahlen von 8, 16, 32 Bits usw.), so muß das jeweils zutreffende Format irgendwie codiert angegeben sein. Dafür gibt es folgende Möglichkeiten:

- im Befehl selbst, beispielsweise im Rahmen des Operationscodes,
- im Operanden selbst (Datentypkennzeichnung),
- in besonderen deskriptiven Angaben, die den Operanden beschreiben,
- in zusätzlichen Auswahlangaben.

In manchen Architekturen werden mehrere Prinzipien im Verbund eingesetzt. So wird in der Architektur IA-32 die Operandenlänge (16 oder 32 Bits) durch Steuerbits in Deskriptoren bestimmt. Weiterhin ist im Befehl selbst codiert, ob einer der Operanden ein Byte lang ist oder ob er der jeweiligen Operandenlänge entspricht. Zudem kann dem einzelnen Befehl ein Vorsatzbyte (Abschnitt 9.1.6.) vorangestellt werden, um die Operandenlänge zu wechseln (z. B. um mit einem 16-Bit-Operanden zu arbeiten, obwohl die Operandenlänge im Programmsegment auf 32 Bits eingestellt ist).

Operandenwerte
Der Wert eines Operanden ist nichts anderes als die jeweils zu verarbeitende Aneinanderreihung von Bits. Es gibt folgende Möglichkeiten, sie zur Verarbeitung in der Hardware bereitzustellen:

- im Befehl selbst, beispielsweise im Anschluß an den Operationscode. Solche Operanden heißen *Direktwerte* (Immediate Values bzw. Immediates (sprich: Immiedijets)).
- in Hardware-Registern. Register werden wir in Abschnitt 2.5. näher betrachten. Es kann sich um fest zugeordnete Register handeln (implizite Registernutzung) oder um Register, die aus einem Registersatz (Register File) ausgewählt werden. Das erfordert entsprechende Auswahlangaben (Registeradressen) im Befehl.
- im Speicher. Dort müssen die Operanden adressiert werden. Die Angaben zur Adressierung sind Bestandteil des Befehls. Es gibt verschiedene Adressierungsweisen (Abschnitt 2.6.4.).

Diese Möglichkeiten sind in vielen Architekturen kombiniert vorgesehen. So gibt es in der Architektur IA-32 bei Befehlen mit zwei Operanden folgende Kombinationsmöglichkeiten:

- Register-Register,
- Speicher-Register,
- Direktwert-Register,
- Register-Speicher,
- Direktwert-Speicher.

Sind Direktwerte unbedingt erforderlich?
Strenggenommen: nein. Man könnte schließlich alle derartigen Angaben als Konstanten speichern und in den Befehlen entsprechend adressieren. Direktwertangaben in den Befehlen haben sich aber seit Jahrzehnten als zweckmäßig erwiesen (Erfahrungssache). Operationsbefehle mit Direktwerten gibt es deshalb auch in ganz modernen Architekturen (z. B. IA-64).

Ergebnisse
Verarbeitungsergebnisse (Resultate) müssen für die weitere Verarbeitung irgendwo gespeichert werden. Hierfür gibt es nur zwei Alternativen: (1) in Hardware-Registern, (2) im Speicher.

Wie viele Operanden und Ergebnisse?
Typische Operationsbefehle bilden aus zwei Operanden ein Ergebnis. Verarbeitungsschema (vgl. Abschnitt 2.4.2.): Ergebnis := Operand 1 **op** Operand 2. Dabei ist höchstens einer der Operanden ein Direktwert*). Wieviele Adreß- oder Wertangaben aber in einem Operationsbefehl vorgesehen sind, ist Sache des jeweiligen Register- bzw. Adreßmodells (Abschnitt 2.5.).

*): aus der Verknüpfung zweier Direktwerte (= Konstanten) würde wiederum ein fester (konstanter) Wert entstehen. Den könnte man aber schon zur Compilierzeit vorausberechnen.

Bedingungsanzeige
Bei genauer Betrachtung liefern viele Operationen nicht nur *ein* Ergebnis. Es ist üblich, bestimmte Kennzeichen des eigentlichen Ergebnisses (z. B. einer Addition) in der Hardware zu ermitteln und gesondert (z. B. in einem Register) zu speichern. Diese Kennzeichen (z. B. für "Resultat gleich Null", "Ausgangsübertrag" "Überlauf" usw.) heißen *Flags, Flagbits oder Bedingungscodes* (Condition Codes)*) - neuerdings (IA-64) auch Prädikate (Predicate Bits). Im Ergebnis mancher Operationen werden nur derartige Bits gestellt, es wird aber kein eigentliches Verarbeitungsergebnis abgespeichert (Beispiel: Vergleichsoperationen).

*): sprich: Fläggs, Konndischn Kohds...

2.2.3. Transportbefehle

Sofern Operanden aus Registern entnommen bzw. Ergebnisse in Registern hinterlassen werden, benötigt man zusätzliche Befehle, um Werte aus dem Speicher in Register zu bringen, um Registerinhalte in den Speicher zu schaffen und um Werte aus einem Register in ein anderes zu transportieren. Dafür sind besondere Transportbefehle vorgesehen, die die transportierten Werte nicht verändern. Diese müssen entsprechende Adreßangaben enthalten. Speicheradressen werden dabei üblicherweise aus einer Direktwertangabe im Befehl und aus den Inhalten weiterer Register zusammengesetzt (Abschnitt 2.6.3.).

Fachbegriffe:

- Transport allgemein: Move (sprich: muhf),
- Transport vom Speicher in ein Register: Laden (Load; sprich: Lohd),
- Transport vom Register in den Speicher: Speichern (Store; sprich: S-tohr).

2.2.4. Programmsteuerbefehle

Ein sehr wichtiges Merkmal des Universalrechners ist die Möglichkeit, den Verarbeitungsablauf auf Grund von Verarbeitungsresultaten flexibel zu steuern (genau dies macht eben die volle Universalität aus). Wie geschieht das? Befehle werden aus dem Speicher geholt. Die Hardware muß dafür eine Befehlsadresse liefern. Ohne besondere Modifikation wird Befehl für Befehl nacheinander abgearbeitet. Das erfordert nur ein schrittweises Erhöhen (Hochzählen) der Befehlsadresse. Die entsprechende Funktionseinheit heißt deshalb Befehlszähler (Instruction bzw. Program Counter (IC, PC) oder auch, so bei x86/IA-32, Instruction Pointer (IP)). Um diesen Ablauf zu beeinflussen, sind besondere Befehle notwendig. Sie bewirken, daß die jeweils nächste Befehlsadresse nicht durch Weiterzählen, sondern durch Überladen des Befehlszählers mit einem Adreßwert bestimmt wird.

Verzweigungen
Eine solche Abweichung von der Reihenfolge der Befehlsabarbeitung heißt Sprung (Jump; sprich: Dschampp) bzw. Verzweigung (Branch; sprich: Brahntsch). Die Adreßangabe heißt demzufolge Sprung- bzw. Verzweigungsadresse. Sie ist als Direktwert im Befehl enthalten, wird aus einem Register entnommen oder durch eine Kombination von Registerinhalt und Direktwert gebildet (die Adressierung beim Verzweigen behandeln wir in Abschnitt 2.6.4.).

Bedingte Verzweigungen (Jump/Branch on Condition)
Wird ein solcher Verzweigungsbefehl ausgeführt, so ist es eine im Verlauf der Verarbeitung entstandene Bedingung, die entscheidet, ob die Verzweigung tatsächlich wirksam wird oder ob der nächste Befehl doch von der Folgeadresse geholt wird. Während der Ausführung des Verzweigungsbefehls entscheidet somit die Hardware, ob der Befehlszähler weiterzählt oder ob er mit der Verzweigungsadresse überladen wird. Die Verzweigungsbedingung ist im Befehl angegeben. Zumeist handelt es sich um eine bestimmte Belegung von Flags bzw. Bedingungscodes. In manchen Architekturen sind auch Verzweigungsbefehle vorgesehen, die die entscheidenden Operandenverknüpfungen selbst ausführen (ein typischer Fall ist das Vergleichen zweier Operanden und das Verzweigen bei Gleichheit).

Unbedingte Verzweigung (Unconditioned Jump/Branch)
Ein solcher Verzweigungsbefehl prüft keine Bedingung, sondern lädt stets die Verzweigungsadresse in den Befehlszähler.

Unterprogrammruf (Subroutine Call)
Bei der gewöhnlichen Verzweigung wird der Befehlszähler einfach überladen. Hingegen wirkt ein Unterprogrammruf 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 (Abbildung 2.2). 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.

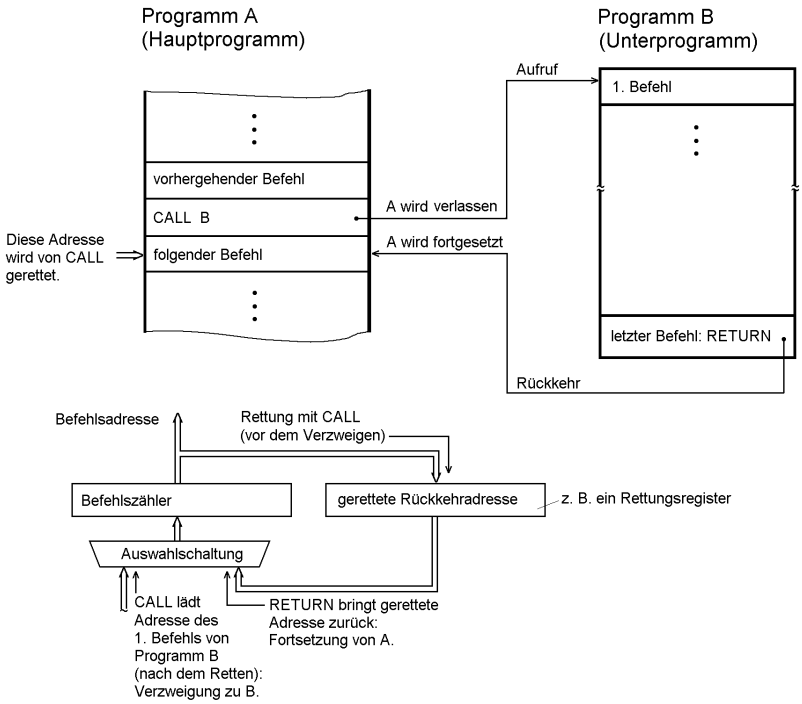


Abbildung 2.2 Prinzip des Unterprogrammrufs

Die Kombination von Unterprogrammruf und Rückkehr ist ein ganz wichtiges Prinzip in der Rechnerarchitektur. Man kann gleichartige, immer wiederkehrende Programmstücke als Unterprogramme auslegen - und gleich zweimal sparen: man muß sie (1) nur einmal schreiben und (2) auch nur einmal speichern. Programmkomplexe, die wir als Betriebssysteme, als Programmpakete für numerische Rechnungen, für Aufgaben der Dateiverwaltung usw. kennen, sind im wesentlichen Sammlungen von Unterprogrammen, die, einmal entwickelt, praktisch allen Anwendern verfügbar sind.

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: (1) in Registern, (2) in besonders vereinbarten Speicherzellen, (3) im Stack, (4) im Programm selbst (an den Unterprogrammruf anschließend). In Lehrbüchern der Grundlagen des Programmierens nimmt die Parameterübergabe einen beachtlichen Raum ein. Da wir nicht programmieren wollen, werden wir auch nicht weiter auf Einzelheiten eingehen.

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

2.2.5. E-A-Befehle

E-A = Eingabe-Ausgabe (I/O = Input/Output*)). Maßgebend ist die Auslegung des E-A-Subsystems:

1. gar keine besonderen E-A-Befehle. E-A-Einrichtungen werden über den Speicheradreßraum adressiert (Memory Mapped I/O), so daß die E-A-Vorgänge mit Speicherzugriffsbefehlen auf die betreffenden Adressen ausprogrammiert werden können. Beispiele: DEC VAX, Motorola 68k, die meisten RISC-Architekturen.
2. elementare Transportbefehle, die sich auf (wenigstens) einen besonderen E-A-Adreßraum beziehen. Übertragungsrichtungen: Input = von E-A-Einrichtung zum Prozessor, Output = vom Prozessor zur E-A-Einrichtung. Beispiel: x86/IA-32.
3. Steuerbefehle zur Kommunikation mit dem autonom arbeitenden (“intelligenten”) E-A-Subsystem (typische Wirkungen: Starten und Anhalten von E-A-Abläufen, Übergeben von Parametern, Abfragen von Zuständen). Beispiel: S/360.../390.

*): sprich: Ei-Oh, Inputt-Autputt.

2.2.6. Weitere Befehle

Wir haben bis jetzt die wichtigsten Befehlsarten eines Universalrechners kennengelernt. Viele Architekturen enthalten zudem Befehle, die sich nicht ohne weiteres in dieses Schema einordnen lassen. Dazu gehören namentlich die *Systembefehle*, die in vielen Architekturen nur in besonderen Betriebszuständen ausführbar sind (sie werden nur in Systemprogrammen verwendet; Anwenderprogramme dürfen solche Befehle nicht ausführen). Das betrifft u. a. Befehle zur Unterbrechungssteuerung, zum Übergang zwischen Systemzuständen und zur Unterstützung des Fehlersuchens.

2.3. Befehlsformatgestaltung

Wir haben soeben gesehen, was im einzelnen Befehl enthalten sein kann. Fassen wir zuammen:

- eine Angabe, die ihn selbst nach Art und genauer Wirkung beschreibt (diese heißt üblicherweise Operationscode, auch wenn es sich beispielsweise um einen Transport- oder Verzweigungsbefehl handelt),
- Angaben zu den Operanden und Ergebnissen (Adressen, Direktwerte, Formatkennzeichnung),
- Angaben zum Programmablauf (beispielsweise Verzweigungsadressen).

Extrem lange Befehle
An sich kommt man mit einem einzigen Befehlsformat aus. Man muß dazu nur alle

codierten Angaben aneinanderreihen, die notwendig sind, um all das zu steuern, was in der Prozessor-Hardware überhaupt gleichzeitig gesteuert werden kann. Damit kommt man auf Befehlslängen von hundert Bits und weit mehr. So etwas gibt es auch tatsächlich: es ist das Prinzip der Mikroprogrammierung und der Maschinen mit besonders langem Befehlswort (der VLIW-Architekturen).

Offensichtlich ist dieses Prinzip extrem aufwendig. Schließlich kann man bei weitem nicht alle Funktionen der Hardware gleichzeitig sinnvoll ausnutzen. Es geht dabei nicht nur um den Bedarf an Speicherplatz. Speicherschaltkreise haben immer größere Speicherkapazitäten und werden - auf die Speicherkapazität bezogen - immer preiswerter. Diese Entwicklung kann aber ein Problem nicht aus der Welt schaffen: die Übertragungsgeschwindigkeit. Selbst wenn man den Speicherplatz geschenkt bekäme, um beispielsweise 256 Bits lange Maschinenbefehle in beliebiger Anzahl unterzubringen: sie müssen doch zwecks Ausführung in den Prozessor transportiert werden. Die Übertragungsgeschwindigkeit hat aber physikalisch und technologisch bedingte Grenzen. Es kostet also entweder einen hohen Aufwand (256 Bits breite Informationswege) oder ein Mehrfaches der Zeit, um einen 256-Bit-Befehl zu holen als beispielsweise einen 32-Bit-Befehl. Damit sich das 256-Bit-Format rechnet, müßte ein solcher Befehl mehr leisten als 8 32-Bit-Befehle, das heißt, es müssen sich mehr als 8 mal so viele nützliche Wirkungen im Befehl codieren lassen (z. B. Adreß- oder Operationsangaben, die wirklich zur Lösung der Verarbeitungsaufgabe beitragen). Das ist beim heutigen Stand der Technik zwar für gewisse Klassen von Anwendungen gegeben, aber nicht für alle, im besonderen nicht für viele "Wald-und-Wiesen"-Anwendungen. Deshalb sind (zumindest noch heutzutage) kürzere Befehle (mit typischerweise nur jeweils einer Wirkung) flexibler bzw. vielseitiger nutzbar und daher wirtschaftlicher.

Das Bemühen um möglichst kurze Befehle, auf Grund der zunächst hohen Speicherkosten von Beginn an ein wesentliches Ziel der Architekturentwicklung, ist also auch künftig von Bedeutung*). Wir wissen, daß in der Technik jede Auslegung einen Kompromiß zwischen verschiedenen einander widersprechenden Anforderungen darstellt. In diesem Sinne steht bei neueren Entwicklungen aber eher die Leistungssteigerung als die Kostensenkung im Vordergrund. Deshalb wird die "Kompression" der Befehlsformate nicht mehr bis zum Extrem getrieben.

*): auch die modernen VLIW-Architekturen sehen nicht etwa Riesenbefehle von 128 oder 256 Bits Länge vor. Vielmehr werden mehrere kurze Einzelbefehle in größeren Behältern zusammengefaßt. Beispiele: (1) IA-64: 3 Befehle zu 41 Bits bilden ein sog. Bündel (Bundle), das in einem Behälter von 128 Bits untergebracht wird. (2) die Superskalar-Signalprozessorfamilie TMS320C6x: 8 Befehle zu 32 Bits sind in einem 256 Bits langen Wort untergebracht (Abbildungen 9.22...9.24). (3) Majc (Sun): 4 32-Bit-Befehle sind zu einem Paket von 128 Bits zusammengefaßt. Sind weniger als 4 Befehle parallel ausführbar, so wird auch das jeweilige Paket entsprechend kürzer (Abbildungen 9.25, 9.26). (4) die Crusoe-Prozessoren der Fa. Transmeta.

Variable Befehlslänge

Jeder Befehl ist nur so lang wie unbedingt notwendig. Wichtige, häufig gebrauchte Befehle werden mit einem Byte, höchstens mit zwei Bytes codiert. Weniger häufig gebrauchte

Varianten, Modifikationen, Sonderwirkungen und Ähnliches kosten dann zusätzliche Bytes. Die aktuelle Befehlslänge wird vom ersten Byte ausgehend nach und nach entschlüsselt. (Hochleistungsprozessoren holen und entschlüsseln mehrere Bytes gleichzeitig.)

Feste Befehlslänge
Alle Befehle haben dieselbe Länge. Unter Verzicht auf beliebige Kombinationsmöglichkeiten ist der einzelne Befehl in verschiedene Felder aufgeteilt, in denen die unterschiedlichen Angaben codiert sind. Üblicherweise entspricht die Befehlslänge der Verarbeitungsbreite (in modernen Architekturen sind 32 Bits üblich). Die Feldaufteilung nimmt keine Rücksicht auf Bytegrenzen. Natürlich wird auch bei einer solchen Formatgestaltung versucht, häufig gebrauchte, leistungsbestimmende Formen von Operationen, Transporten, Verzweigungen usw. in jeweils einem Befehl unterzubringen. Was auf diese Weise nicht zu codieren ist, erfordert eben mehrere Befehle. Welche Grundgedanken stehen hinter diesem Prinzip?

- die Hardware, die solche fest formatierten Befehle adressiert, holt und entschlüsselt, ist wesentlich einfacher als bei variabler Befehlslänge. Somit läßt sich die Arbeitsgeschwindigkeit erhöhen.
- die weitaus meisten der vergleichsweise einfachen, fest formatierten Befehle kann man bei beherrschbaren Aufwendungen in einem einzigen Maschinenzyklus ausführen.

Wir haben es also mit folgenden Alternativen zu tun:

1. variabel lange, kompakt codierte Befehle, die bedarfsweise auch komplexe Wirkungen hervorbringen,
2. Befehle fester Länge mit vergleichsweise einfacher Wirkung, die dafür aber schneller ablaufen.

CISC und RISC
Architekturen, die gemäß dem ersten Prinzip ausgelegt sind, heißen CISC-Architekturen (CISC = Complex Instruction Set Computer); solche gemäß dem zweiten Prinzip heißen RISC-Architekturen (RISC = Reduced Instruction Set Computer). In den letzten Jahren wurde um die Vor- und Nachteile beider Prinzipien in der Fachwelt recht heftig gestritten. Beispiele beider Architekturprinzipien: in Kapitel 9.

2.4. Befehls- und Programmablaufsteuerung

2.4.1. Grundlagen

Wenn wir über Befehls- und Programmabläufe reden, müssen wir zwei Ebenen streng voneinander unterscheiden:

1. die in der Architektur auf *logischer* Ebene vorgesehenen Abläufe. Diese gehören zum Programmiermodell der betreffenden Architektur.

2. die in einer bestimmten Maschine (einem Rechnermodell, einem Prozessorschaltkreis usw.) tatsächlich vonstatten gehenden Abläufe (die Abläufe der *physischen* Ebene).

Grundsätzlich gilt, daß jede Implementierung (jede Ausführungsform, jedes Rechnermodell, jeder Prozessortyp usw.) in seinen Wirkungen der Architekturdefinition entsprechen muß. Beabsichtigte oder von den Anwendern in Kauf genommene Abweichungen sind Inkompatibilitäten, andere funktionelle Abweichungen sind Entwurfsfehler. Abweichungen, die in der einzelnen Maschine auftreten, haben ihre Ursache zumeist in Fehlfunktionen oder Ausfällen der Hardware.

Die *Programmablaufsteuerung* betrifft die Abarbeitungsreihenfolge der Befehle, die *Befehlsablaufsteuerung* die Ausführung des einzelnen Befehls.

Maschinen- und Befehlszyklen
Alle Abläufe sind an bestimmte Zeittakte gebunden. Den der Befehlsablaufsteuerung zugrunde liegenden Zeittakt nennen wir Maschinenzyklus (Machine Cycle; sprich: Mäschien Seikl). Er hängt unmittelbar mit der Schaltungsstruktur und mit den technologischen Grundlagen der Implementierung zusammen. Der Programmablaufsteuerung liegen hingegen *Befehlszyklen* (Instruction Cycles) zugrunde, deren Dauer im allgemeinen variabel ist: sie bestehen zumeist aus mehreren Maschinenzyklen, wobei deren Anzahl oft von Nebenumständen des Verarbeitungsablaufs abhängt (so kann es besondere Wartezustände geben, wenn beispielsweise ein Speicherzugriff nicht sofort erledigt werden kann).

Die sequentielle Arbeitsweise
Ein wichtiges Kennzeichen der Architektur klassischer Universalrechner (wozu auch die PCs gehören, trotz immer wieder bombastisch verkündeter Neuheiten) ist die sequentielle Arbeitsweise. Das heißt: es wird alles nacheinander erledigt; sowohl die einzelnen Verarbeitungsschritte im Befehl als auch die befehlsweise Ausführung der Programmabläufe. Der überragende Vorteil dieses Prinzips besteht in Einfachheit, Klarheit, Überschaubarkeit, der augenfällige Nachteil in der beschränkten Arbeitsgeschwindigkeit.

Seit es geeignete Hardware-Technologien gibt, hat man versucht, etwas gegen den Geschwindigkeits-Nachteil zu tun. So befinden sich in Hochleistungsprozessoren mehrere Befehle gleichzeitig in verschiedenen Zuständen ihrer Ausführung (Stichworte: Befehlspipelining, Superskalarität, Superpipelining). *Aus der Sicht des Programmierers bleibt aber die sequentielle Arbeitsweise nach wie vor erhalten*. Dafür wird ein beachtlicher Aufwand getrieben.

2.4.2. Die Zuweisungsnotation

Diese Notationsweise brauchen wir gelegentlich, um Befehlsabläufe, Befehlswirkungen, Zugriffe usw. zu beschreiben. Beispiele:

- Y := X
- C := A + B
- C := A **op** B

- A := <R1>
- B := <<R2> + OFFSET>

Das Symbol ":=" bezeichnet eine Zuweisung. Y := X bedeutet, daß Y den Wert von X zugewiesen erhält (der Doppelpunkt kennzeichnet das Ziel der Zuweisung). C := A + B bedeutet, daß A und B zueinander addiert werden und daß C das Ergebnis zugewiesen bekommt. Setzen wir A = 3 und B = 5, so ergibt sich C := 3 + 5, d. h. C erhält den Wert 8 (an sich also ganz einfach; Sie müssen sich nur daran gewöhnen, daß hier das Resultat links steht). **op** steht für eine an sich x-beliebige Operation. Spitze Klammern bedeuten "Inhalt von". "<R1>" bedeutet also "Inhalt von Register R1". Ein Name ohne spitze Klammern steht für einen Wert, in spitzen Klammern hingegen für eine Adresse. OFFSET ist ein Wert, <OFFSET> wäre hingegen der Inhalt der Speicherzelle, die durch die Adresse OFFSET adressiert wird. <<R2> + OFFSET> bedeutet demnach: der Inhalt des Registers R2 wird zum Wert OFFSET (Direktwert im Befehl) addiert. Das Ergebnis wird als Adresse verwendet. Der gesamte Ausdruck steht dann für den Inhalt der so adressierten Speicherzelle.

2.4.3. Befehlsablaufsteuerung

Jeder Befehl wird in mehreren aufeinanderfolgenden Phasen ausgeführt (Abbildung 2.3).

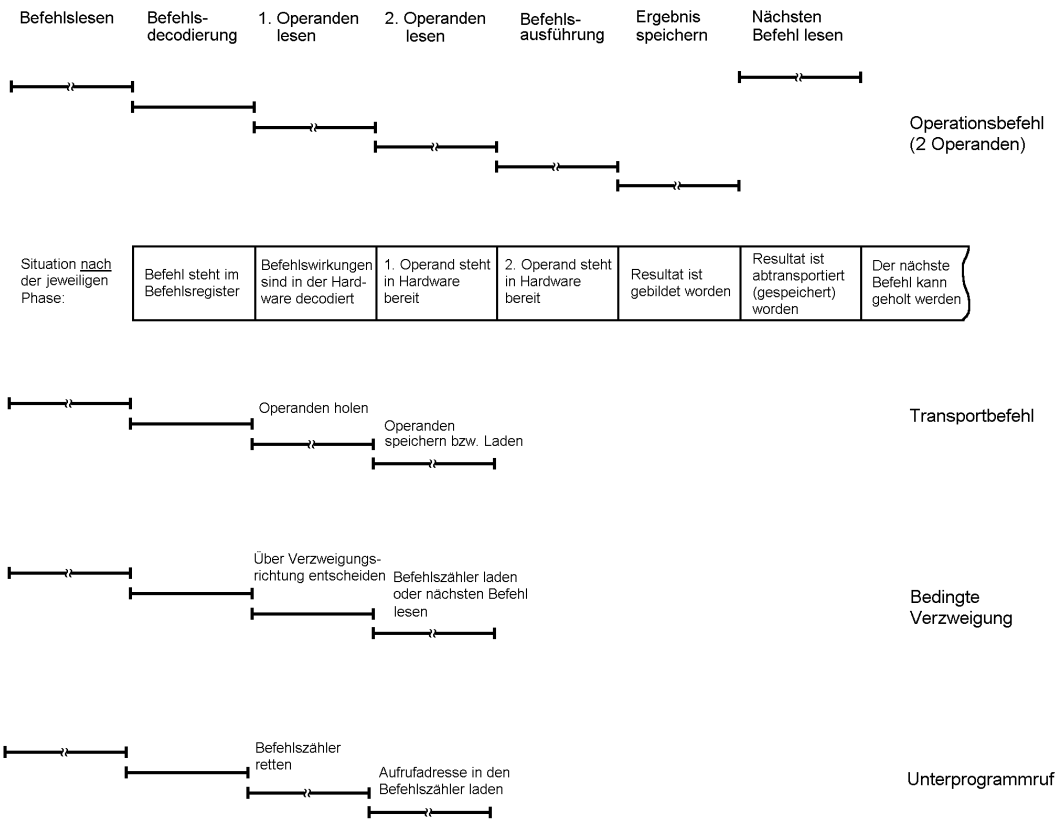


Abbildung 2.3 Phasen der Befehlsausführung

Wie lange dauern die einzelnen Phasen?

Im klassischen Fall (ohne jegliche Ablaufüberlappung oder Parallelisierung) erfordert jede Phase wenigstens einen Maschinenzyklus. Weitere Maschinenzyklen können aus verschiedenen Ursachen hinzukommen:

- wenn die Schaltmittel und Informationswege nicht alle beteiligten Bits parallel verknüpfen bzw. übertragen können (sparsame Auslegung der Hardware, schmale Datenwege usw.)*),
- wenn der Prozessor auf Grund der Systemauslegung (z. B. Bestückung mit langsameren Speicherschaltkreisen) oder auf Grund besonderer Betriebsumstände auf andere beteiligte Einrichtungen warten muß (Wartezustände).

*): manche Abläufe sind so kompliziert, daß sie grundsätzlich nicht innerhalb eines einzigen Maschinenzyklus erledigt werden können (Beispiel: die Division zweier Binärzahlen - vgl. Abbildung 1.15).

Befehlslesen (Instruction Fetch)
Mit der Befehlsadresse (Inhalt des Befehlszählers) wird ein Lesezugriff zum Speicher ausgeführt. Das Befehlslesen läuft so lange, bis der ganze Befehl in den Prozessor geholt wurde. Dazu kann es notwendig sein, die einzelnen Befehlsbytes nacheinander zu entschlüsseln, um festzustellen, wieviele Bytes noch nachfolgen. Der Befehlszähler zeigt am Ende dieser Phase auf die Anfangsadresse des nachfolgenden Befehls.

Befehlsdecodierung (Instruction Decode)
Aus dem geholten Befehl muß die Hardware die einzelnen Wirkungen erkennen bzw. entschlüsseln (decodieren). Der gesamte Befehl wird im Prozessor üblicherweise in einem Befehlsregister gehalten (dieses ist für den Programmierer völlig unzugänglich und daher auch nicht Bestandteil des Registersatzes im Programmiermodell). So ist es nicht besonders schwierig, alle Befehlswirkungen gleichzeitig durch entsprechend angeordnete Decodierschaltungen zu erkennen (Abbildung 2.4).

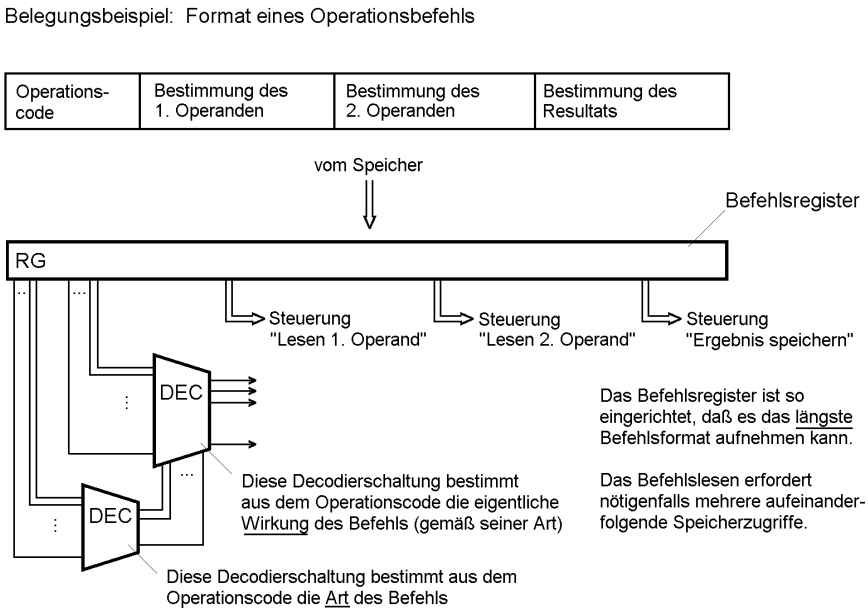


Abbildung 2.4 Befehlsregister

Operandenlesen (Operand Fetch)
Sind im Befehl Operanden explizit angegeben, so müssen sie in der Hardware zur Verarbeitung bereitgestellt werden. 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 Hardware-Registern zur Verarbeitung bereitstehen (bei Direktwerten: im Befehlsregister). Ist dies nicht der Fall, müssen entweder Zugriffe zum Speicher oder zu einem Registersatz (Register File) ausgeführt werden. Dem eigentlichen Zugriff gehen oft noch Abläufe der Adreßrechnung voraus, die aus Adreßangaben im Befehl selbst, in Registern bzw. in weiteren Speicherpositionen die jeweilige Zugriffsadresse ermitteln (Abschnitt 2.6.).

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.

Beispiel: Eine Multiplikation erfordert mehr Zyklen als eine Addition. Moderne Hochleistungsprozessoren multiplizieren nicht stur gemäß der Verarbeitungsbreite, sondern sie erkennen die höchstwertige Eins des Multiplikanden und beenden den Ablauf entsprechend. Zyklen, in denen nur führende Nullen multipliziert werden würden, werden also eingespart.

Speichern der Ergebnisse (Result Store)
Ergebnisse werden implizit in Hardware-Registern festgehalten (das betrifft im besonderen Flags bzw. Bedingungscodes) bzw. sie werden in den Speicher oder in einen Registersatz geschrieben. Ähnlich dem Operandenlesen kann dem Speichern eine Adreßrechnung vorangehen.

Besonderheiten

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

Verzweigungsbefehle
Dem Befehlslesen folgt (bei bedingter Verzweigung) die Entscheidung darüber, ob die Verzweigung stattfindet oder nicht. Die Verzweigung selbst erfordert das Bereitstellen der Verzweigungsadresse (erforderlichenfalls mittels Adreßrechnung) und das Laden des Befehlszählers.

Unterprogrammruf
Der Ablauf entspricht dem der Verzweigungsbefehle, wobei vor dem Laden des Befehlszählers dessen bisheriger Inhalt (die auf den Folgebefehl zeigende Adresse) gerettet werden muß. Diese Rettungsphase kann Adreßrechnungen und Speicherzugriffe enthalten.

2.4.4. Programmablaufsteuerung

Kaltstart
Jeder Programmablauf muß irgendwie anfangen. Normalerweise wird ein Programm gestartet, indem der Befehlszähler mit der Adresse des ersten Befehls geladen wird. Wie kommen wir aber zum allerersten Befehl, nachdem wir die Maschine eingeschaltet haben? Dafür gibt es zwei Prinzipien: Festadressierung und Anfangsprogrammladen. Näheres in Kapitel 7.

Befehlsadressierung
Befehle werden zwecks fortlaufender Ausführung nacheinander aus dem Speicher geholt. Dazu wird die Befehlsadresse entsprechend der Länge des jeweils geholten Befehls erhöht. Die aktuelle Befehlsadresse wird im architekturseitig definierten Befehlszähler gehalten. Üblicherweise ist dies die Adresse des jeweils folgenden Befehls.

Hinweis:
Das Prinzip der fortlaufenden Befehlsadressierung erlaubt es, Befehle *vorbeugend* zu holen (Instruction Prefetch bzw. Look Ahead).

Verzweigung
Wird eine Verzweigung wirksam, so muß der Befehlszähler mit der gewünschten Befehlsadresse (der Adresse des Sprungziels) überladen werden. Die Verzweigungsabläufe unterscheiden sich vor allem in zwei Einzelheiten:

1. Herkunft der Adresse. Typische Varianten: Direktwert im Befehl, Registerinhalt, Inhalt einer Speicherposition.
2. Verrechnung der Adresse. Typische Varianten: absolut, relativ zum Befehlszähler, relativ zu einer Basisadresse (Näheres in Abschnitt 2.6.).

Manche Verzweigungs- und Unterprogrammrufbefehle können weitergehende Wirkungen veranlassen (Beispiel IA-32: Übergang in ein anderes Segment, Übergang in eine andere Privilegierungsstufe, Taskumschaltung).

Die Sprungweite (Sprungdistanz)
Diese Angabe besagt, in welchem Bereich des Adreßraums man mit der Adreßangabe des jeweiligen Befehls tatsächlich verzweigen kann. Beispiel: eine 8-Bit-Adreßangabe, die vorzeichengerecht zum Inhalt des Befehlszählers addiert wird, erlaubt nur ein Verzweigen in einem Adreßbereich von -128...+127 Bytes, bezogen auf die Anfangsadresse des Folgebefehls (Abbildung 2.5).

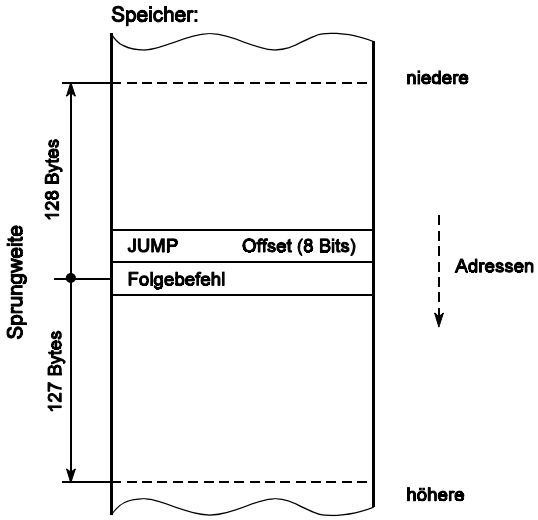


Abbildung 2.5 Zum Begriff der Sprungweite

Überspringen (Skip)
Gelegentlich ist die im (bedingten) Verzweigungsbefehl vorgesehene Sprungweite zu gering. Die Abhilfe: wir verwenden einen Verzweigungsbefehl mit umgekehrter Verzweigungsbedingung und lassen diesem eine unbedingte Verzweigung nachfolgen (solche Befehle haben typischerweise eine ausreichende Sprungweite). Das Sprungziel des ersten Verzweigungsbefehls ist dann der übernächste Befehl. Somit wird bei nicht erfüllter (ursprünglicher) Verzweigungsbedingung der zweite Verzweigungsbefehl übersprungen (Abbildung 2.6).

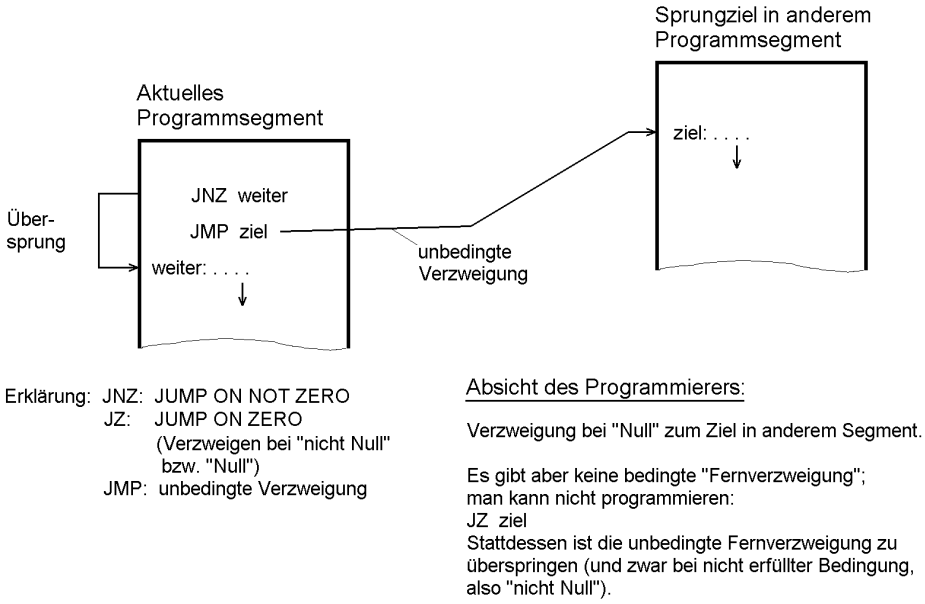


Abbildung 2.6 Überspringen (Skip)

Hinweis:
Manche Architekturen (das betrifft vor allem kleinere Mikrocontroller) haben eigens Übersprungbefehle, z. B. “Überspringen, wenn Bit Nr. x gesetzt”. Bedingte

Verzweigungen werden dann aus zwei Befehlen aufgebaut: aus einem Übersprungbefehl und aus einer unmittelbar nachfolgenden unbedingten Verzweigung. Es wird dann so programmiert, daß bei nicht erfüllter Verzweigungsbedingung die unbedingte Verzweigung übersprungen (also: nicht ausgeführt) wird.

Unterprogrammruf
Der Verzweigung zum Unterprogramm muß ein Retten der Befehlsadresse vorangehen.

Im einfachsten Fall gibt es eine einzige Rettungsmöglichkeit (ein Register oder eine fest zugeordnete Speicherzelle). Dann ist es allerdings nicht ohne weiteres möglich, Unterprogramme zu schachteln, das heißt, in einem Unterprogramm wieder ein Unterprogramm zu rufen (es gelingt zwar, aber der Programmierer muß sich eigens darum kümmern*).

In modernen Architekturen hat sich allgemein das Prinzip des *Kellerspeichers* (Stacks) durchgesetzt. Ein solcher Stack (sprich: S-täck) wird typischerweise im Arbeitsspeicher eingerichtet. Ein Register im Prozessor (der Stackpointer; SP) zeigt auf das "obere Ende" des Stack (Top of Stack TOS). Ablaufbeispiel: Ist eine Adresse zu retten, so wird zunächst der Inhalt des Stackpointers vermindert (Stacks wachsen typischerweise in Richtung niederer Adressen; Abschnitt 2.6.5.). Daraufhin wird die Adresse in den Stack transportiert. Bei der Rückkehr wird die gerettete Adresse aus dem Stack zurückgeholt. Daraufhin wird der Inhalt des Stackpointers wieder erhöht.

*): die typische Lösung: das Stack-Prinzip wird mittels Software nachgebildet.

Tasks, Multitasking, Taskumschaltung
In der Rechnerarchitektur bezeichnet der Begriff *Task* allgemein die Umgebung (Register, Speicherbereiche usw.), in der ein einzelnes Programm lauffähig ist. Solche Umgebungen (Task Environments) sind in Abbildung 2.7 veranschaulicht. Im üblichen Sprachgebrauch wird auch ein Programm, das die betreffenden Vorkehrungen nutzt, das in eine solche Umgebung gleichsam eingebaut ist, als Task bezeichnet. Näherungsweise gleiche Bedeutung wie der Begriff "Task" haben auch die Begriffe Prozeß, Partition und virtuelle Maschine, die in der Literatur gelegentlich gebraucht werden, um die hier behandelten Sachverhalte zu bezeichnen. Der Begriff "virtuelle Maschine" ist besonders einprägsam. Wenn Sie Abbildung 2.7 näher betrachten, werden Sie erkennen, daß jede einzelne Umgebung einem Prozessor entspricht, der durch die Registeranordnung und die zugeordneten Speicherbereiche bestimmt ist. Es liegt somit nahe, von virtuellen - scheinbaren - Prozessoren zu reden, wobei zu einer Zeit jeweils ein virtueller Prozessor den realen Prozessor belegt.

Multitasking bedeutet, daß mehrere Programme gleichzeitig lauffähig sein können. Selbstverständlich kann in einem einzelnen Prozessor jeweils nur ein Programm tatsächlich arbeiten. Die betreffende Task heißt die *arbeitende* (Running) Task. Andere Programme können *arbeitsfähig* sein, das heißt mit der Befehlsausführung beginnen, sobald die betreffende Task Laufzeit erhält. Solche Tasks heißen *aktive* (Busy) Tasks. Weiterhin ist es möglich, Task-Umgebungen vorzusehen (im Speicher), die nicht unmittelbar arbeitsfähig sind und sie nur unter bestimmten Bedingungen arbeitsfähig zu machen. Solche Tasks heißen *inaktive* (Not Busy bzw. Idle) Tasks. Abbildung 2.8 veranschaulicht verschiedene Taskzustände und die Umschaltmöglichkeiten zwischen ihnen.

Wie geht die Taskumschaltung vonstatten?
Nach dem Hardware-Rücksetzen und der Initialisierung gibt es eine erste arbeitende Task. Eine andere Task kann (1) durch Umschaltbefehle oder (2) durch Unterbrechungsauslösung (Interrupts) Laufzeit erhalten, also zur arbeitenden Task werden.

Was ist bei einer Taskumschaltung zu tun?
Die Umgebung der arbeitenden Task belegt den Prozessor, namentlich dessen Register. Bei der Umschaltung ist diese Registerbelegung zu retten und die Registerbelegung der neuen Task ist im Prozessor einzustellen (bzw. in den Prozessor zu laden). Zum Retten und Halten der Registerbelegungen müssen besondere Bereiche im Speicher reserviert sein (vgl. Abbildung 2.7).

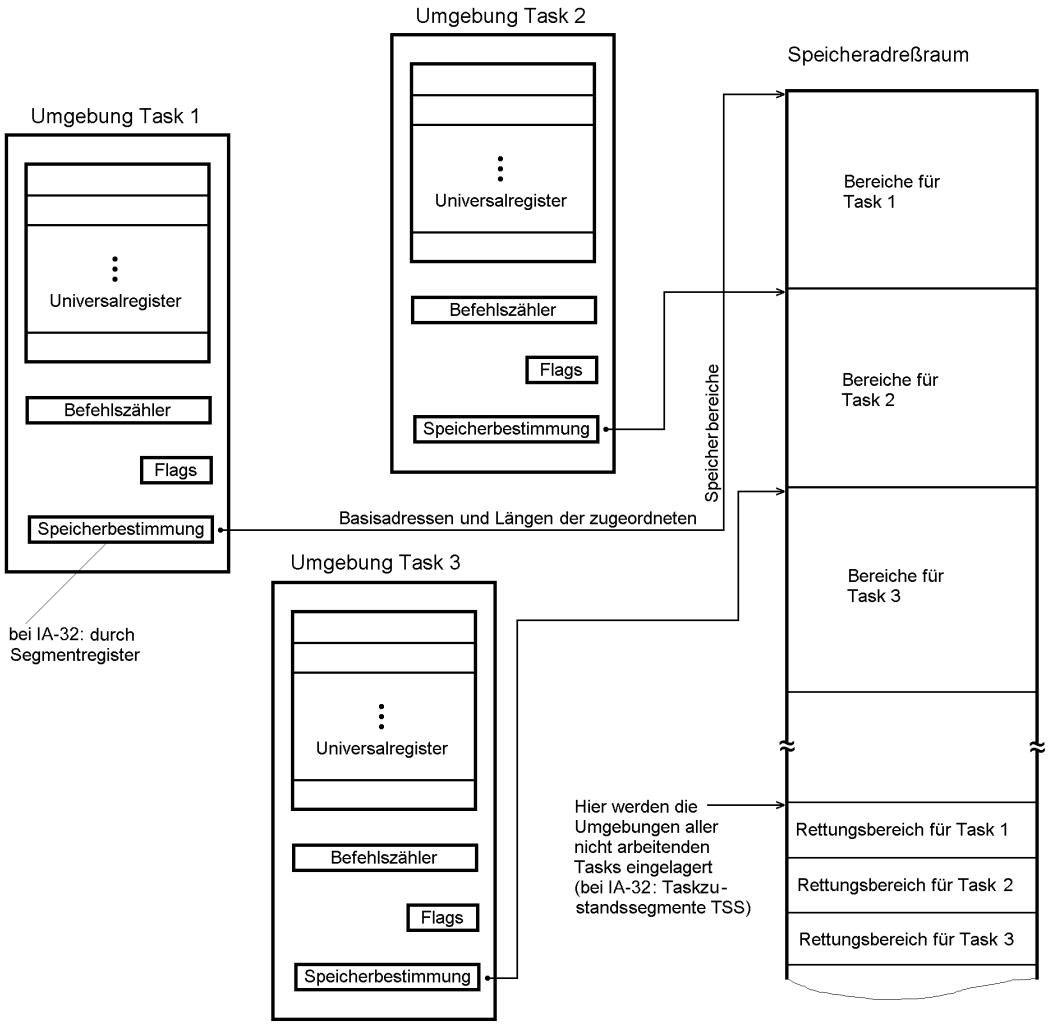


Abbildung 2.7 Task-Umgebungen

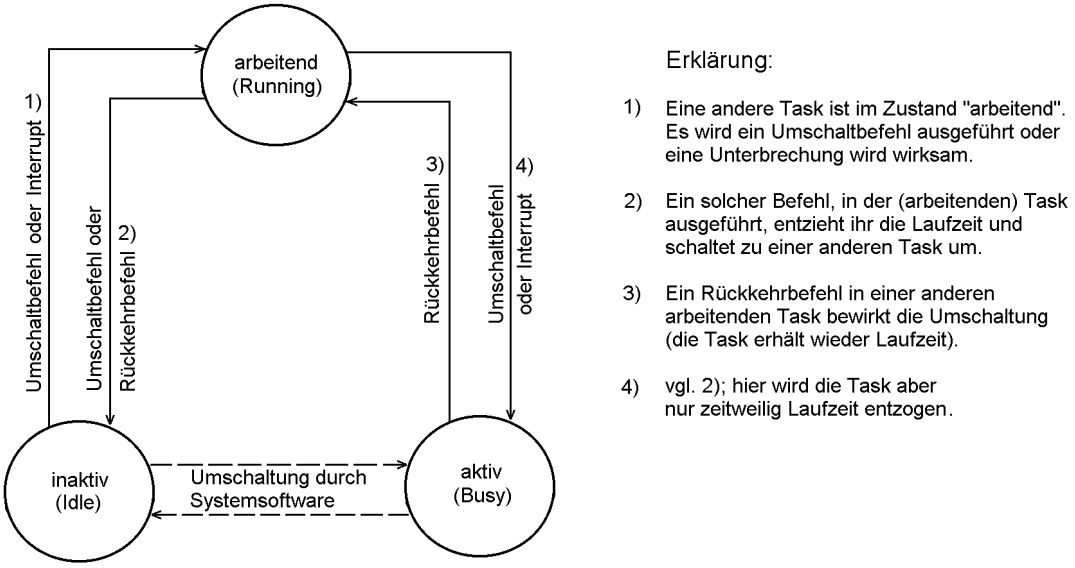


Abbildung 2.8 Taskzustände und Umschaltmöglichkeiten (am Beispiel IA-32)

Architekturseitige Unterstützung
An sich kann die Taskumschaltung voll ausprogrammiert werden. In manchen Architekturen sind aber von vornherein unterstützende Maßnahmen vorgesehen. So gibt es in der Architektur IA-32 besondere Taskzustandssegmente (Task State Segments TSS), die die Registerinhalte und andere Angaben der Taskumgebung aufnehmen. Wird bei einer Verzweigung, einem Unterprogrammruf, einer Unterbrechungsauslösung oder bei einer Rückkehr aus der Unterbrechungsbehandlung ein TSS-Deskriptor angesprochen, so läuft die Taskumschaltung automatisch ab.

Hardware-Multitasking
Das Retten und Einstellen der Registerbelegungen ist vergleichsweise zeitaufwendig. Es kann entfallen, wenn in der Prozessor-Hardware genügend Registersätze vorgesehen sind (einer für jede Task-Umgebung). Man muß dann nur noch entsprechend umschalten, so daß jeweils ein Registersatz mit den Verarbeitungsschaltungen, Datenwegen usw. aktiv verbunden ist.

Besonders elegant ist die Umschaltung in einem festen Zeitrahmen, beispielsweise in jedem Maschinenzyklus. Mit n Registersätzen kann man n virtuelle Maschinen mit jeweils 1/n-facher Verarbeitungsleistung vorsehen (Abbildung 2.9). Der wesentliche Vorteil: Wartezustände, die sich im einzelnen Befehl ergeben, können mit Zyklen anderer Tasks sinnvoll ausgefüllt werden. Dieses Prinzip wurde bereits 1964 in der Anlage CDC 6600 verwirklicht, um mit einer einzigen Hardware bis zu 10 virtuelle Maschinen als Ein-Ausgabe-Prozessoren bereitzustellen (die Hardware wurde alle 100 ns umgeschaltet, so daß jeder virtuelle Prozessor einen Maschinenzyklus von 1 µs hatte). Es ist damit zu rechnen, daß dieses Prinzip wieder an Bedeutung gewinnt (Stichworte: Vertical Multithreading (Sun), Hyper Threading (Intel)).

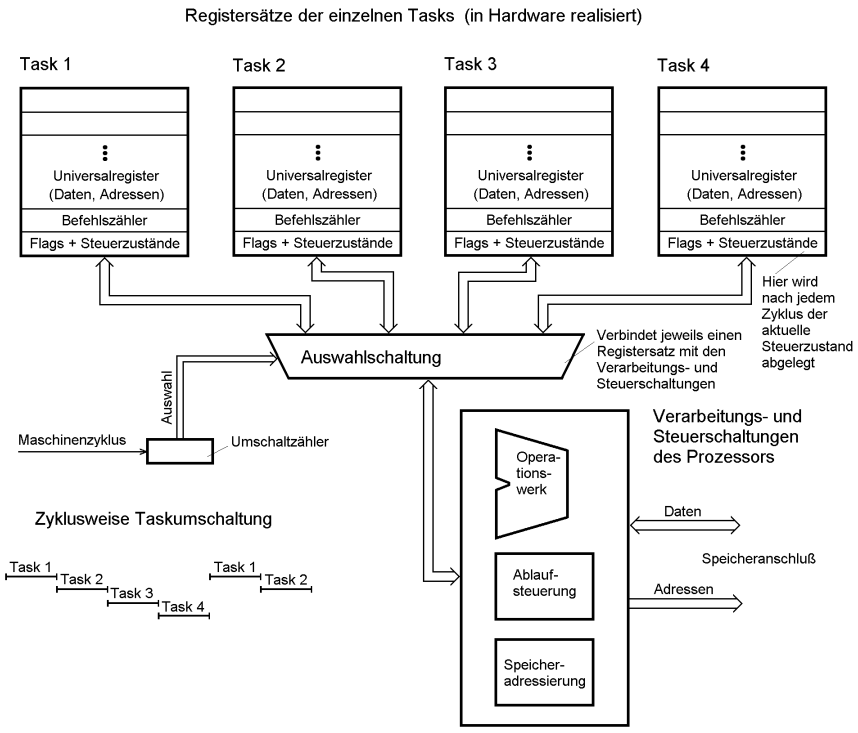


Abbildung 2.9 Hardware-Multitasking

Unterbrechungen
Unterbrechungen sind gleichsam erzwungene Unterprogrammrufe, wobei die Anfangsadresse des zu rufenden Programms auf Grundlage architekturseitiger Festlegungen geliefert wird (beispielsweise als Festwert oder über eine Tabellenstruktur). Näheres in Kapitel 4.

2.5. Register- und Adreßmodelle

2.5.1. Register

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).

Im folgenden geht es ausschließlich um die Architektur-Register. Unsere Betrachtungen sind zunächst grundsätzlicher Natur. Zu Registersätzen konkreter Architekturen sei auf Kapitel 9 verwiesen.

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. Diese wollen wir in Form von *Register- und Adreßmodellen* überblicksmäßig betrachten, wobei wir zwischen 4 Arten unterscheiden:

- Mehradreßmodelle,
- Einzweckregister- oder Akkumulatormodelle,
- Mehrregistermodelle,
- Stackmodelle.

*): engl. Register File; sprich: Retschistr Feil.

2.5.2. Mehradreßmodelle

Dreiadreßmaschinen
Brauchen wir - aus der Sicht des Programmierers - überhaupt Register? - Grundsätzlich: nein. Wir müssen unsere Architektur nur so auslegen, daß alle Angaben entweder im Speicher oder als Direktwerte im Befehl untergebracht sind. Typische elementare Operationen (z. B. die 4 Grundrechenarten über Binärzahlen) bilden aus zwei Operanden ein Ergebnis (Resultat). Verknüpfungsschema (vgl. Abschnitt 2.4.2.): <Resultat> := <1. Operand> **op** <2. Operand>. Es liegt also nahe, die entsprechenen Adreßangaben in einem Befehl zusammenzufassen (Dreiadreßbefehle). Abbildung 2.10 veranschaulicht die Befehlsformate einer solchen (hypothetischen) Architektur.

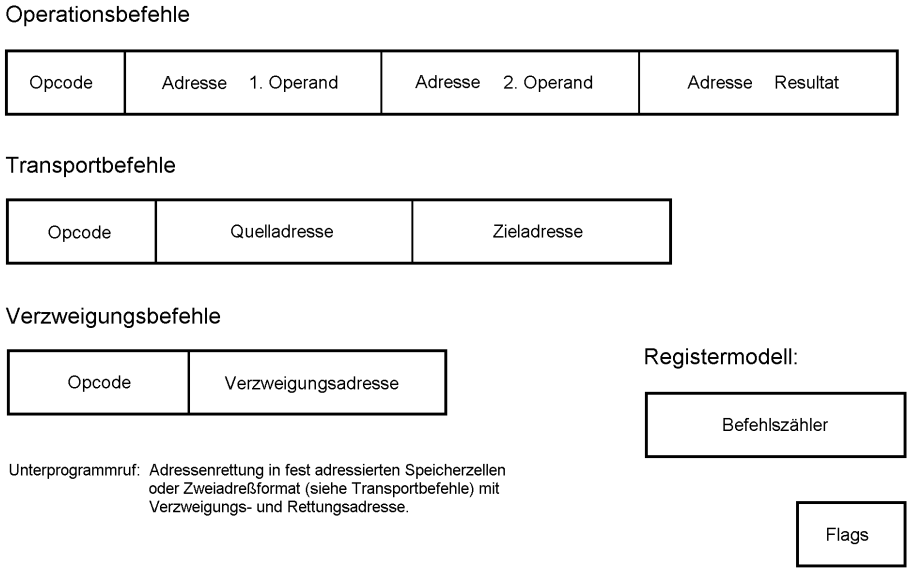


Abbildung 2.10 Befehlsformate und Registermodell einer (hypothetischen) Dreiadreßmaschine

In der Anfangszeit der Computertechnik hat man solche Maschinen tatsächlich gebaut (waren Speicher schon klein und teuer, so konnte man sich eine Vielzahl von Registern (mit wenigstens einer Elektronenröhre je Bit!) noch weniger leisten).

Dreiadreß-Registermaschinen
Das Prinzip ist aber keineswegs veraltet. In den modernen RISC-Architekturen begegnet es uns wieder, aber unter geradezu entgegengesetzten Verhältnissen: es gibt *nur noch Operationen mit Registerinhalten*, so daß es sich bei den drei Adreßangaben im Operationsbefehl sämtlich um Registeradressen handelt (Verknüpfungsschema: <R3> := <R1> **op** <R2>). Damit kommt ein Dreiadreßbefehl ohne weiteres mit 32 Bits aus (vergleichen Sie Abbildung 2.10 mit den Befehlsformaten in den Abbildungen 9.10 und 9.18).

Zweiadreßmaschinen
Viele Operationen laufen darauf hinaus, einen Operanden durch Verknüpfung mit einem anderen zu ändern (Verknüpfungsschema: <1. Operand> := <1. Operand> **op** <2. Operand>). In vielen Programmabläufen kommen derartige Operationen häufiger vor als jene, die dem Verknüpfungsschema der Dreiadreßmaschine entsprechen. Deshalb beschränkt man sich in manchen Architekturen auf zwei Adreßangaben (Alternativen: Speicher - Speicher, Speicher - Register, Register - Register). Hierzu gehören u. a. die "klassischen" Architekturen S/360 und x86/IA-32. Auch unser fiktiver Prozessor P/F ist eine Zweiadreß-RISC-Maschine.

2.5.3. Einzweckregistermodelle

Wenn jedes Register nur eine einzige bestimmte Aufgabe hat (oder ein genau umschriebenes Aufgabengebiet, für das es kein weiteres - alternativ nutzbares - Register gibt), so braucht man gar keine besonderen Auswahlangaben im Befehl: ein solches Register wird gleichsam automatisch verwendet, wenn die betreffenden Funktionen auszuführen sind (implizite Nutzung). Typische Beispiele für Einzweckregister sind der Befehlszähler, das Flagregister, der Stackpointer, Operandenregister, Zählregister usw.

In der Vergangenheit waren viele Architekturen so ausgelegt, auch die ersten Mikroprozessoren. Wir begegnen diesem Einzweckprinzip noch in der x86-Architektur (Abbildung 2.11).

Hinweis:
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. Auch etliche IA-32-Befehle nutzen diese Register noch zweckgebunden.

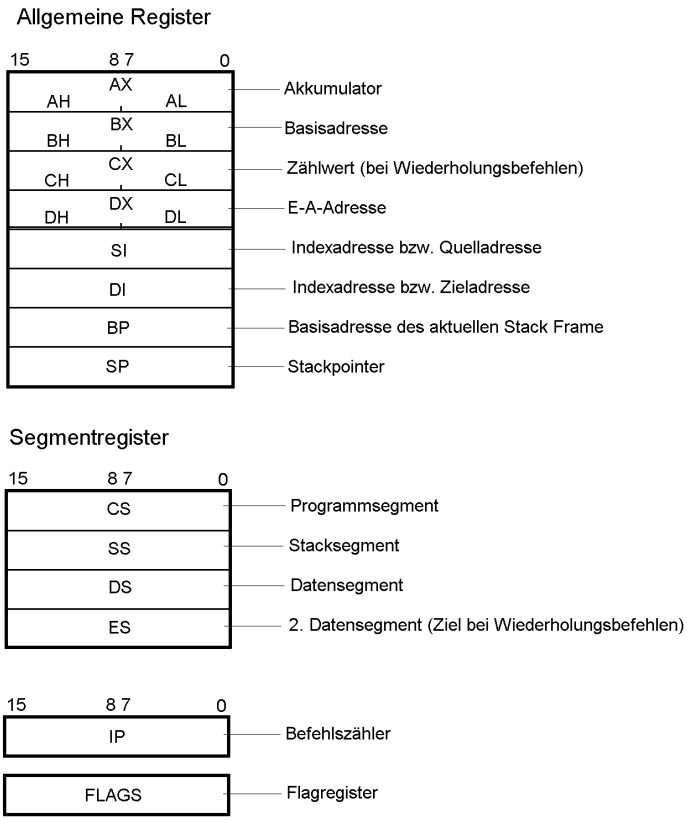


Abbildung 2.11 Die Register des 8086 und deren zweckgebundene Nutzung

2.5.4. Einadreßmaschinen

Man kommt für alle Operandenverknüpfungen mit einem einzigen Register aus, wenn man daraus einen der Operanden entnimmt (der zweite stammt aus dem Speicher) und den Registerinhalt durch das gebildete Resultat überschreibt. Ein solches Register nennt man *Akkumulator*. Auf dieser Grundlage lassen sich sehr einfache Rechnerarchitekturen angeben, deren Befehle nur eine einzige Adreßangabe enthalten (Einadreßmaschinen, Abbildung 2.12). Das Verknüpfungsschema: <Akkumulator> := <Akkumulator> **op** <Adresse>.

Die Einadreßmaschine mit Akkumulator geht auf den Mathematiker John von Neumann zurück. Des weiteren hat John von Neumann den gemeinsamen Speicher für Daten und Befehle eingeführt. Auch diese Auslegung trägt zur Einfachheit solcher Maschinen bei: wenn Daten und Befehle im Speicher nicht voneinander zu unterscheiden sind, kann das Programm seine eigenen Befehle als Daten behandeln und im Speicher ändern. Und damit läßt sich ohne Ende tricksen...

Beispiel: die Rückkehr aus einem Unterprogramm. Wie bekommt man in der Maschine von Abbildung 2.12 die Rückkehradresse in den Befehlszähler? - Beim Unterprogrammaufruf wird die Rückkehradresse in den Akkumulator gerettet. Der Befehl zum Verlassen des Unterprogramms (Rückkehrbefehl) ist eine unbedingte Verzweigung.

Der erste Befehl des Unterprogramms muß ein Speicherbefehl sein, der den Akkumulatorinhalt in das Adreßfeld dieses Rückkehrbefehls bringt. Man spart so einen besonderen Datenweg zum Befehlszähler.

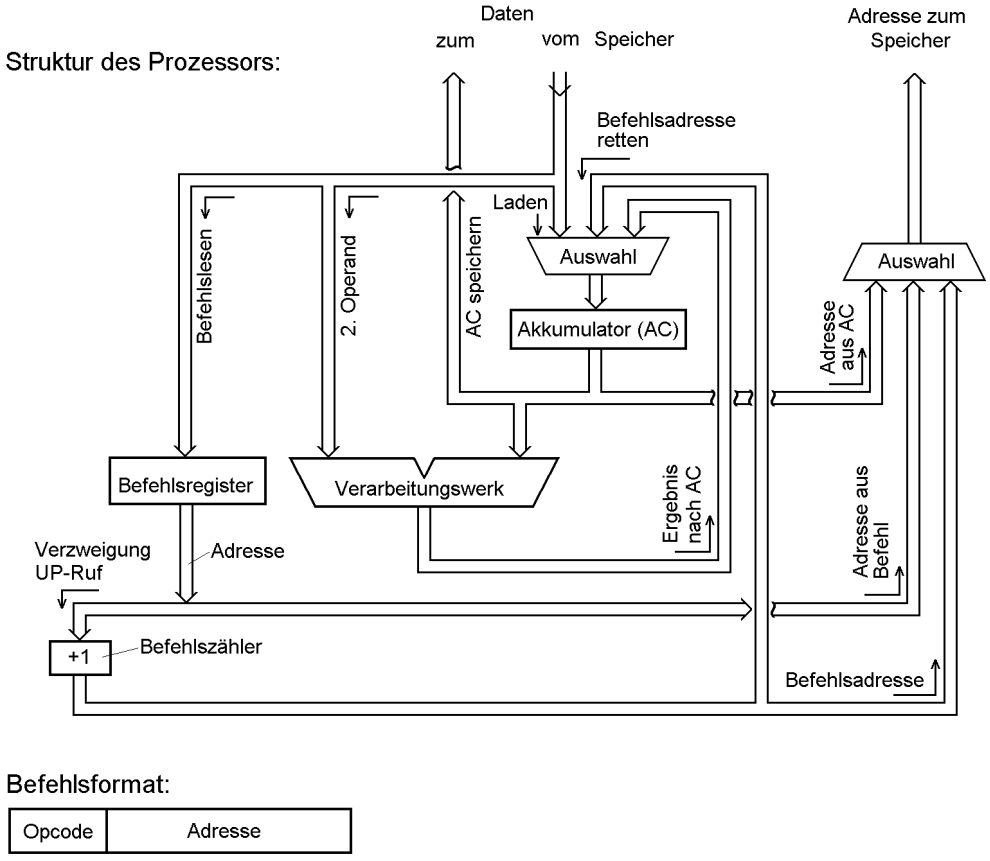


Abbildung 2.12 Einadreßmaschine mit Akkumulator (hypothetisches Beispiel)

Verständlicherweise waren solche Konzepte in der Anfangszeit geradezu ideal: sparsamste Hardware*) und uneingeschränkte Universalität. Ein offensichtlicher Nachteil: die vergleichsweise geringe Verarbeitungsleistung. Des weiteren ist man heutzutage von selbstmodifizierenden Programmen gänzlich abgekommen - man hat sogar besondere Schutzvorkehrungen eingeführt, um Schreibzugriffe auf Programme als fehlerhaft zu erkennen.

*): einfacher geht es kaum noch: die ganze Maschine besteht - neben dem Verarbeitungswerk und den (nicht dargestellten) Steuerschaltungen - nur aus 3 Registern (Akkumulator, Befehlszähler, Befehlsregister) und aus 2 Auswahlschaltungen (ggf. müßte noch ein Flagregister hinzukommen - aber auch darauf könnte man verzichten).

2.5.5. Mehrregistermodelle

Es sind mehrere gleichartige Register vorgesehen. In den Befehlen können die

gewünschten Register explizit ausgewählt werden. Ein Mehrregistermodell kann Registersätze zu verschiedenen Zwecken umfassen, beispielsweise:

- Datenregister,
- Adreßregister,
- Steuerregister,
- Testregister usw.

Universalregister
Eine Anordnung gleichartiger Register, die für alle Verwendungszwecke nutzbar sind (General Purpose Register File), hat zweifellos den Vorzug der Eleganz. Man kann beispielsweise soweit gehen, auch den Befehlszähler als ein Universalregister vorzusehen: das Retten der Befehlsadresse ist dann ein Umladen in ein anderes Register, das Verzweigen ein Schreiben in das betreffende Register usw.

In vielen modernen Architekturen sind mehrere in sich regulär gestaltete 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.). Allgemein üblich ist eine Aufteilung in Registersätze für allgemeine Operationen und Adreßrechnungen (allgemeine Register), für Adressierungszwecke, für die Systemsteuerung (Steuerregister), für Diagnose- und Fehlersuchzwecke (Testregister, Fehlersuch- oder Debugging-Register) sowie für spezielle Datentypen (Gleitkommaregister, Vektorregister usw.). Beispiele: in Kapitel 9.

*): Gegenbeispiel: die Majc-Architektur. Hier sind alle Datenregister lang genug, um jeden Datentyp (Integer, Gleitkomma usw.) aufnehmen zu können.

Wie groß soll ein Universalregistersatz sein?
Nach wie vor eine Streitfrage. Machen wir uns zunächst klar, daß die Register eigentlich auf zweierlei Weise genutzt werden:

1. als Behälter für Operanden und Resultate (mit anderen Worten: zur Parameterübergabe an die eigentliche Verarbeitungshardware),
2. als Schnellspeicher für Variable, auf die besonders häufig zugegriffen wird.

Beide Nutzungsweisen geraten nicht selten in Konflikt miteinander - vor allem dann, wenn es (1) nur wenige Register gibt bzw. wenn (2) bestimmte Befehle die Nutzung bestimmter Register erzwingen - wie dies vor allem bei IA-32 der Fall ist (manche Rechenbefehle verlangen nach den Registern EAX und EDX, Stringbefehle benötigen die Register ESI und EDI usw.). Zudem erzwingen die typischen Programmiermodelle der Laufzeitsysteme höherer Programmiersprachen eine bestimmte Nutzung der Register ESP und EBP (als Stackpointer und Base Pointer; vgl. Abschnitt 2.6.4.). Da es insgesamt nur 8 “universelle” Register gibt (Abbildung 9.3), bleiben kaum Register übrig, die freizügig als Schnellspeicher verwendet werden können.

Modernere (RISC-) Architekturen sehen deshalb typischerweise 32 Universalregister vor, wovon nur wenige eine Sondernutzung haben (Beispiele: Alpha, Mips, PowerPC (Kapitel 9)).

Sehr große Registersätze
Es liegt nahe, die Anzahl der Register weiter zu erhöhen, sobald man sich dies leisten kann (Sache der Schaltkreistechnologie). Beispiele: (1) Sparc: 136 Universalregister, (2) IA-64: 128 Universalregister, AltiVec-Erweiterung der PowerPC-Prozessoren: 32 Register zu 128 Bits. Der wesentliche Nachteil: die vielen Register wollen bei Taskumschaltungen usw. auch gerettet sein. Weitere Probleme: (1) lange Registeradreßfelder in den Befehlen, (2) wenn man die Registeranzahl sehr groß wählt, wird die Registerzugriffszeit zu lang (mit anderen Worten: bei gleicher Schaltkreistechnologie könnte man einen Prozessor mit kleinerem Registersatz schneller laufen lassen).

Deshalb sieht man im Zusammenhang mit großen Registersätzen auch höher entwickelte Zugriffsverfahren vor. Diese laufen darauf hinaus, auf die Registernutzung das Prinzip des Stack Frame (Abschnitt 2.6.5.) sinngemäß zu übertragen. Beispiele: Sparc und IA-64 (Kapitel 9).

Maschinen- bzw. modellspezifische Register (MSRs)
Ein seit langem bewährtes Prinzip, um reguläre Programmschnittstellen zu den vielen maschinenspezifischen Steuerfunktionen bereitzustellen. Alle Angaben, die einzustellen oder abzufragen sind, werden in solchen Registern zusammengefaßt, und es werden entsprechende Zugriffsbefehle (zum Lesen und Schreiben) vorgesehen. Der Rest ist dann - aus Sicht des Programmierers - Sache der Dokumentation (es bleibt nichts anderes übrig, als die Handbücher zu studieren - dort sind die Zugriffsadressen und die Registerbelegungen bis auf's Bit erklärt).

2.5.6. Stackmodelle

Die Organisation von Stacks (Kellerspeichern, LIFOs) werden wir in Abschnitt 2.6.5. behandeln. Wir kennen Stacks bereits als Einrichtungen zur Adressenrettung beim Unterprogrammruf (Abschnitt 2.4.4.). Stacks können aber auch vorteilhaft zur Datenspeicherung verwendet werden. Ein typischer Operationsbefehl entfernt die beiden obersten Elemente vom Stack, verknüpft sie miteinander und legt das Ergebnis auf den Stack zurück. Ein Ladebefehl holt einen Operanden aus dem Speicher und legt ihn auf den Stack (Push--Ablauf), ein Speicherbefehl entnimmt den Operanden vom Stack (Pop-Ablauf) und bringt ihn in den Speicher. Damit haben wir das Schema einer einfachen Stackmaschine (Abbildung 2.13).

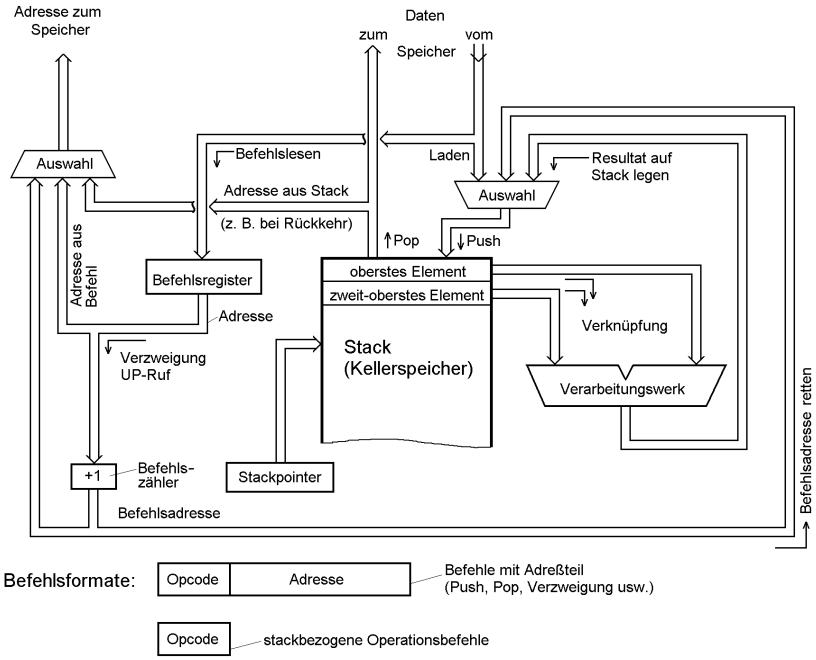


Abbildung 2.13 Einfache Stackmaschine (hypothetisch)

Ein offensichtlicher Vorteil: wir brauchen keine Registeradressen; Operationsbefehle brauchen sogar überhaupt keine Adressen. Ein weiterer Vorteil ist nicht so offensichtlich: das Stack-Prinzip eignet sich hervorragend zur Ausführung ineinandergeschachtelter Operationen, die in vielen Anwendungen auftreten. Beim Rechnen mit kompliziert aussehenden Ausdrücken (z. B. Formeln, die Klammern enthalten) wird dies besonders deutlich.

Beispiel
Der Ausdruck (A + B) · (C : (D - E)) ist zu programmieren. In Tabelle 2.2 sind Programmbeispiele für eine Akkumulatormaschine, eine Mehrregistermaschine und eine Stackmaschine gegenübergestellt.

Wie ist die Befehlsfolge für die Stackmaschine zustande gekommen? - Dazu wurde der Ausdruck folgendermaßen umgeformt: A B + C D E - : ·

Das ist die sogenannte *umgekehrte polnische Notation* (Reverse Polish Notation, RPN), die auf den polnischen Logiker Lukasiewicz zurückgeht. Es ist eine *klammerfreie* Schreibweise. Operanden werden nacheinander angeführt, darauf folgt der Operator (z. B. + oder ·). Operatoren werden auf die jeweils unmittelbar vorangehenden Operanden angewendet bzw. auf Zwischenergebnisse, die von bereits zuvor angewendeten Operatoren auf dem Stack hinterlassen wurden. So erklärt es sich, daß mehrere Operatoren aufeinander folgen können. Das schrittweise "Abbauen" eines solchen Ausdrucks wird in Abbildung 2.14 am Beispiel von Tabelle 2.2 erklärt.

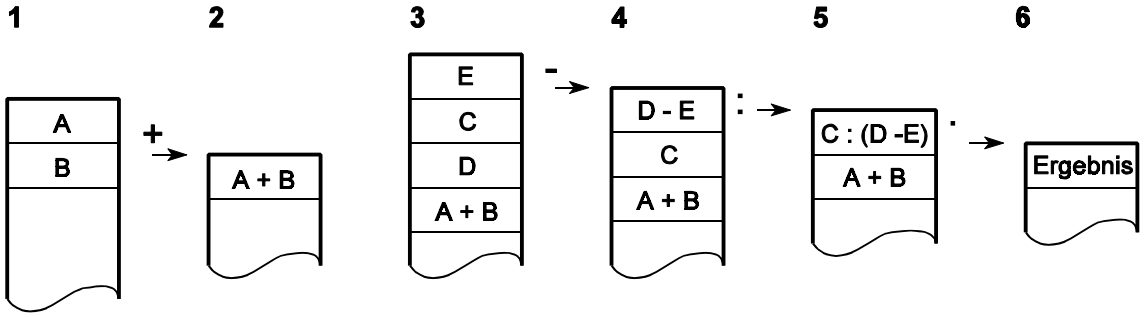


Abbildung 2.14 Operandenverknüpfungen in einer Stackmaschine. Zu berechnender Ausdruck: (A + B) · (C : (D - E))

Erklärung:
1 - Operanden A, B auf dem Stack; 2 - der Additionsbefehl (+) bewirkt, daß A und B vom Stack genommen werden und daß statt dessen 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); f) auf dem Stack; 6 - der Multiplikationsbefehl (·) bewirkt schließlich, daß das Endergebnis als einziges auf dem Stack verbleibt.

Wenn Sie die in der Abbildung gezeigten Schritte mit der dritten Spalte von Tabelle 2.2 vergleichen, so werden Sie erkennen, daß der Ablauf in der Stackmaschine und die umgekehrte polnische Notation einander voll entsprechen. Der Stack ist *das* Mittel der Wahl, um Verschachtelungen, Klammerungen usw. (die durchaus nicht auf mathematische Formeln beschränkt sind) programmseitig aufzulösen. Praktisch alle Compiler nutzen dieses Prinzip. Wenn die Architektur keine Stacks vorsieht oder wenn die vorgesehenen ungeeignet sind, wird die Stack-Organisation softwareseitig nachgebildet. Deshalb hat es seit den 60er Jahren Architekturen gegeben, denen die Stack-Organisation von vornherein zugrunde liegt.

Register-Stack
Die Operandenregister sind als Stack organisiert. Das ist beispielsweise in den Gleitkomma-Coprozessoren bzw. -Verarbeitungseinheiten der Architekturen x86/IA-32 vorgesehen.

Stack im Arbeitsspeicher
Die Systeme der Fa. Burroughs (B 5000... B 6700) beruhten seit den 60er Jahren auf einer Stack-Architektur.

In den heutzutage verbreiteten Architekturen (auch x86/IA-32) ist zwar ein Stackpointer vorgesehen, und die Nutzung von Stacks wird auch anderweitig unterstützt (durch besondere Datensegmente, Taskzustandssegmente usw.), es handelt sich aber nicht um "echte" Stack-Architekturen, da der Stack nicht die einzige Einrichtung ist, in der Operanden zu Verknüpfungszwecken bereitgestellt und Ergebnisse abgelegt werden (es stehen auch noch Register und der auf übliche Weise adressierte Arbeitsspeicher zur Wahl).

Ausdruck: (A + B) · (C : (D - E))			
	Akkumulatormaschine	Mehrregistermaschine	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
Bedarf	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)]

Annahmen zur Befehlslänge:
1): 1 Befehl = 4 Bytes (≙ 32-Bit-RISC)
2): ein Operationsbefehl = 1 Byte, ein Push oder Pop = 4 Bytes (32-Bit-Adressierung ähnlich RISC)

Tabelle 2.2 Programmabläufe in der Gegenüberstellung. Zu berechnenderAusdruck:
(A + B) · (C : (D - E))

Stack-Cache
Das Programmiermodell sieht eine reine Stack-Architektur vor, der Stack ist jedoch technisch aus einem Schnellspeicher auf dem Prozessorschaltkreis (dem Stack Cache) und einem Bereich im Arbeitsspeicher zusammengesetzt, wobei mit besonderen Schaltmitteln versucht wird, Stack-Zugriffe zum Arbeitsspeicher durch geschicktes Zwischenpuffern im Schnellspeicher zu vermeiden. Beispiele dafür sind der Hyperstone-Prozessor sowie die verschiedenen Java-Prozessoren (z. B. PicoJava von Sun).

Stacks und die Programmiersprache Java
Zu den Kennzeichen der Programmiersprache Java gehört, daß man neben der eigentlichen Sprache auch einen fiktiven Prozessor entworfen hat, der die compilierten Java-Programme ausführt: die Java Virtual Machine (JVM). Dies ist eine klassische Stackmaschine, erweitert um Vorkehrungen zum Beschreiben der Objektstrukturen. Beispielsweise erfordert eine Berechung C: = A · B, worin A, B, C Gleitkommazahlen einfacher Genauigkeit sind, folgenden Programmablauf:

- ***fload*** A -- legt die (lokale) Variable A auf den Stack,
- ***fload*** B -- legt die (lokale) Variable B auf den Stack,
- ***fmul*** -- multipliziert die beiden obersten Werte im Stack, entfernt diese -- vom Stack und ersetzt sie durch das Ergebnis der Multiplikation,
- ***fstore*** C -- entnimmt den obersten Wert vom Stack und überführt ihn in die -- (lokale) Variable C.

Die Java Virtual Machine ist objektorientiert
Die Stackzugriffsbefehle herkömmlicher Stackmaschinen beziehen sich lediglich auf Speicheradressen und damit auf Behälter für an sich x-beliebige Daten. Ein JVM-Programm schleppt jedoch gleichsam seine Objektbeschreibung bzw. Symboltabelle mit sich (im sog. Class File). Dies ist das eigentlich Komplizierte an der JVM-Spezifikation (die Befehlsliste selbst sieht vergleichsweise harmlos aus).

Erklärung:
Symboltabellen werden beim Compilieren angelegt. In ihnen werden alle Angaben zu den Variablen des Programms vermerkt (Name, Datentyp, Größe usw.). Herkömmlicherweise bezieht sich das compilierte Programm (beim PC z. B. eine .EXE-Datei) nur noch auf Adreßbereiche im Speicher - ausführbaren Dateien ist gar keine Symboltabelle beigegeben[*)]. In einem Java-Programm hingegen beziehen sich die Zugriffsangaben in bestimmten Befehlen auf Eintrittspunkte in die mitgelieferte Beschreibung. Man hat hier aber keine fest formatierten Tabellenstrukturen vorgesehen, sondern die betreffenden Angaben als fortlaufenden Bytestrom codiert - es handelt sich also praktisch um eine Art komprimierte Symbol- bzw. Objekttabelle. Mit anderen Worten: herkömmlicherweise werden die Bezüge (References) von den Befehlen auf die Daten zur Compilierzeit aufgelöst, in der JVM aber zur Laufzeit. Es ist Angelegenheit der JVM, von den Bezugsangaben zu den eigentlichen Daten zu kommen - die Bezugsangabe eines solchen JVM-Befehls wird somit nicht einfach als Adresse verwendet, sondern von Soft- oder Firmware (Mikroprogramme in “echten” Java-Prozessoren) entsprechend ausgewertet.

*): gelegentliche Ausnahme: die Eintrittspunkte in das Programm. Ein Java Class File enthält aber auch die symbolischen Bezüge *innerhalb* des Programms.

Weshalb gibt es nicht nur noch Stack-Maschinen?
Abgesehen von Fragen der Abwärtskompatibilität hat eine Stack-Architektur auch Probleme:

1. Stacks sind hervorragend geeignet, um Einträge gleicher Länge aufzunehmen. Mit variabel langen Operanden (Zeichenketten usw.) gibt es hingegen Schwierigkeiten (die Stack-Verwaltung wird dann nicht mehr so einfach). Deshalb hatten auch die klassischen Stack-Maschinen von Burroughs noch herkömmliche Befehle, um Zeichenketten, Dezimalzahlen usw. verarbeiten zu können.
2. Zugriffe zu Stackpositionen im Arbeitsspeicher dauern wesentlich länger als zu Registern auf dem Prozessorschaltkreis. Dieser Nachteil läßt sich mit Stack-Caches weitgehend mildern, Nachteil 1 bleibt aber bestehen.
3. aus logischer Sicht, aus der Sicht des Theoretikers, ist das Stack-Prinzip äußerst elegant und liefert sehr kompakte Programme. Es ist aber seiner Natur nach *grundsätzlich sequentiell* (es muß alles nacheinander ablaufen), wodurch der weiteren Leistungssteigerung von vornherein Grenzen gesetzt sind.

Deshalb bildet folgende (hier grob skizzierte) Verfahrensweise den Stand der Technik: Compiler erzeugen zunächst einen Zwischencode für eine (in Software 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. Mit dem Aufkommen entsprechender Architekturen werden weitere Optimierungen mehr und mehr Bedeutung erlangen, die versuchen, Gelegenheiten zur *gleichzeitigen* Ausführung verschiedener Befehle zu erkennen (Stichwort: innewohnender (inhärenter) Parallelismus).

Hinweis:
Auch die im Umfeld der Programmiersprache Java entwickelten Hochleistungsprozessoren sind typischerweise keine Stack-Maschinen, sondern beispielsweise VLIW-Maschinen mit Universalregistersätzen (Beispiel: die Majc-Architektur von Sun).

2.6. Speicheradressierung

2.6.1. Grundlagen

Was ist zu adressieren?
Um überhaupt arbeiten zu können, muß der Prozessor Befehle aus dem Speicher holen: *Befehlsadressierung*. Die Operanden, die verarbeitet werden sollen, sind ebenfalls aus dem Speicher heranzuschaffen, Verarbeitungsergebnisse sind in den Speicher zu transportieren: *Datenadressierung*. Zwischenergebnisse, Rückkehradressen usw. werden häufig in Stackbereichen abgelegt, die sich ebenfalls im Arbeitsspeicher befinden:

Stackadressierung.

Vom Befehl zur Speicheradresse
Will der Prozessor einen Speicherzugriff ausführen, so muß er eine Adresse bereitstellen. Die vom Prozessor gelieferten Adressen wollen wir als *effektive* Adressen bezeichnen. Die Adressen, die letzten Endes tatsächlich den Speicherschaltkreisen zugeführt werden, heißen *physische* Adressen. Nur in sehr einfachen Systemarchitekturen sind effektive und physische Adressen einander gleich (die Adressenausgänge des Prozessors sind dann direkt zu den Adresseneingängen der Speicherschaltkreise durchgeschaltet). In modernen, leistungsfähigen Systemen werden üblicherweise die effektiven Adressen in *Speicherverwaltungseinrichtungen* über Zwischenstufen (wie logische bzw. virtuelle und lineare Adressen) in die physischen Adressen umgewandelt. (Die Begriffsbildungen unterscheiden sich geringfügig von Architektur zu Architektur. Wir verwenden die Redeweise der Architekturen x86/IA-32.)

Hinweis:
Dieses Kapitel betrifft ausschließlich die Bildung *effektiver Adressen im Prozessor*. Im folgenden Kapitel wird es darum gehen, wie daraus über die Speicherverwaltung (Memory Management) die physischen Adressen gebildet werden.

Implizite Adressen
Für bestimmte Zugriffe können Adressen implizit geliefert werden (d. h. ohne besondere Angaben, die z. B. in Befehlen codiert sind). Das ist bei der üblichen fortlaufenden Befehlsadressierung (mittels Befehlszähler) der Fall. Weitere Beispiele sind viele Arten von Stackzugriffen (über einen architekturseitigen Stackpointer) oder der Kaltstart von einer Festadresse.

Adressen als Direktwerte
Alle programmseitig ausgelösten Zugriffe erfordern entsprechende Adreßangaben. Die einfachste Form einer solchen Angabe ist der Direktwert im Befehl. Das bedeutet: Im Befehlsformat ist ein Adreßfeld entsprechender Länge vorgesehen, und die Adreßleitungen des Prozessors sind, erforderlichenfalls über Auswahlschaltungen, den betreffenden Bitpositionen des Befehlsregisters nachgeschaltet.

Kommt man mit Direktwerten aus? - Grundsätzlich: ja, wenn (1) die Bequemlichkeit beim Programmieren sowie die Länge der Programme gar keine Rolle spielt bzw. wenn (2) es möglich ist, Befehle während des Programmablaufs zu modifizieren (deren Adreßfelder zu überschreiben). Heutzutage legt man aber darauf Wert, daß sich Programme *nicht* selbst ändern - moderne Architekturen haben eigens Vorkehrungen, um Programmbereiche im Speicher vor dem Überschreiben zu schützen. Somit ist es unbedingt erforderlich, Adressen programmseitig ändern bzw. beeinflussen zu können (Adreßrechnung).

Wozu Adreßrechnung?
Es gibt drei wesentliche Gründe: (1) den Wunsch nach Verschieblichkeit, (2) Programmverdichtung und (3) Zugriffe auf komplexe Datenstrukturen.

Verschieblichkeit (Relocatability)

Wenn es keine besondere Speicherverwaltung gibt, müssen wir auf alles, was im Speicher steht, mit den Adressen zugreifen, die der jeweils tatsächlichen (physischen) Position im Speicher entsprechen. Solche Adressen nennen wir *absolute* Adressen. Handelt es sich um ein einziges Programm und um eine einzige Anordnung von Datenstrukturen, so bereitet dies keine Schwierigkeiten (viele Mikrocontroller-Anwendungen können hierfür als Beispiel dienen). In komplexeren Anwendungsfällen, wo unterschiedliche Programme geladen werden, um unterschiedliche Datenstrukturen zu bearbeiten, wäre es hingegen äußerst unpraktisch, ausschließlich mit absoluten Adressen zu arbeiten. Jedes Programm, jede Datenstruktur müßte dann nämlich genau an einer bestimmten Stelle im Speicher abgelegt werden, es wäre nicht möglich, den Speicher je nach Bedarf von Anfang an aufzufüllen, Programme oder Daten dort unterzubringen, wo gerade Platz ist usw.

Die Möglichkeit, Programme oder Datenbestände an beliebigen absoluten Adressen zu speichern, nennt man *Verschieblichkeit* (Relocatability). Jede verschiebliche Struktur beginnt, für sich gesehen, an Adresse Null. Die einzelnen Bytes, Worte usw. sind mit Adressen, die von Null an beginnen, beliebig adressierbar. Solche Adressen, die sich auf das einzelne Programm, die einzelne Datenstruktur usw. beziehen, heißen *relative Adressen*. Die absolute Adresse, der die relative Adresse 0 entspricht, heißt *Basisadresse*.

Programmverdichtung (Code Compression)
Programme sollen möglichst kompakt sein, also möglichst wenig Speicherplatz belegen (wir kennen bereits Gründe dafür, weshalb diese Forderung auch angesichts gigantischer Speicherkapazitäten nach wie vor gilt). Wie bekommt man einzelne Befehle und ganze Programme kompakt? - Offensichtlich indem man es so einrichtet, daß (1) die einzelnen Angaben nur so viele Bits belegen wie unbedingt notwendig (Codeverkürzung) und daß (2) in den einzelnen Angaben möglichst wirksame Funktionen codiert sind (Codewirksamkeit).

Codeverkürzung
Es ist wenig wirtschaftlich, in den Befehlen Adreßangaben immer in voller Länge (z.B. 32 Bits) mitzuschleppen. Vollkommen wahlfreie Zugriffe gibt es kaum. Es kommt praktisch nie vor, daß z. B. der Befehl 1 auf Adresse 346 zugreift, der Befehl 2 auf Adresse 3 456 102, der Befehl 3 auf Adresse 4, der Befehl 4 auf Adresse 345 190 277 usw. Vielmehr werden Daten- und Befehlsadressen, wenn man jeweils nur einen Ausschnitt aus dem Verarbeitungsablauf betrachtet, jeweils immer vergleichsweise eng zusammenliegen. (Diese Tatsache bezeichnet man als *Lokalität* von Zugriffen. Sie ist für die Wirksamkeit der modernen Prinzipien zur Speicherorganisation entscheidend. Wir werden in Abschnitt 3.1.5. darauf zurückkommen.)

Codewirksamkeit
Wer auf Maschinenebene (in Assemblersprache) programmiert, wird es zu schätzen wissen, wenn Adressierungsverfahren, die man oft benötigt, von vornherein in den Befehlen vorgesehen sind, wenn man also, um auf eine Zeichenkette in einem Datensatz, auf eine Gleitkommazahl in einer Matrix usw. zuzugreifen, nur den entsprechenden Befehl hinschreiben muß (so daß besondere Befehlsfolgen oder gar Unterprogramme zum Berechnen von Adressen unnötig sind). Das sind die Vorteile für den Programmierer. Dazu die Vorteile für den Anwender: man braucht weniger Speicherplatz, und das Programm

läuft schneller (die zusätzlichen Befehle, die ansonsten nötig wären, müssen weder gespeichert, noch in den Prozessor gebracht, noch ausgeführt werden).

Hinweis: der CISC-RISC-Gegensatz
Wie wirksam der einzelne Befehl sein sollte, ist eine wichtige Streitfrage in der Architekturentwicklung. Hochwirksame Befehle (solche mit komplexen Wirkungen) brauchen umfangreichere Schaltmittel in der Hardware. Diese wird deshalb möglicherweise langsamer, als wenn nur einfache Befehle abzuarbeiten wären. Zudem müssen Befehlswirkung und Absicht des Programmierers geradezu "saugend" zueinander passen. Aus der Erfahrung heraus hat man sich bei neueren Architekturen vorzugsweise zugunsten einfacherer, aber schnell ablaufender Befehle entschieden.

Zugriffe auf komplexe Datenstrukturen
Daß man lediglich einzelne Bits, Bytes, Worte usw. zu verarbeiten hat, kommt an sich nur in vergleichsweise einfachen Embedded Systems vor, so in den meisten Anwendungsfällen der kleinen Mikrocontroller. Bei den PCs haben wir es hingegen zumeist mit umfangreichen und komplexen Datenstrukturen zu tun. Solche Strukturen können aus *gleichartigen Elementen* aufgebaut sein. Sie heißen Feldstrukturen oder *Arrays* (Beispiel: ein Matrix aus Gleitkommazahlen). Es gibt aber auch Datenstrukturen aus *verschiedenartigen Elementen*. Sie heißen Datensätze oder *Records*. Als Beispiel mag ein Datensatz dienen, der für die Beschreibung von Lagerbeständen vorgesehen ist. Er soll enthalten (Abbildung 2.15):

- eine Artikelnummer als 32-Bit-Binärzahl,
- eine Bezeichnung als Zeichenkette von 64 Bytes,
- eine Preisangabe als binär codierte Dezimalzahl von 16 Bytes,
- drei Abmessungsangaben (Länge, Breite, Höhe) - jeweils als 16-Bit-Binärzahl,
- eine Gewichtsangabe als 32-Bit-Gleitkommazahl,
- ein Byte zur Codierung besonderer Zustände.

Auf derartige Elemente in komplexen Datenstrukturen muß man einzeln zugreifen können, das heißt, ausgehend vom jeweils gewünschten Element (z. B. der Zahl in Zeile 3, Spalte 5 einer Matrix oder der Preisangabe in der Datenstruktur von Abbildung 2.15) ist die zugehörige Speicheradresse zu berechnen. Dabei ist es grundsätzlich gleichgültig, ob die Abläufe der Adreßrechnung in komplexen Befehlen von vornherein vorgesehen sind oder ob sie mit einfacheren Befehlen ausprogrammiert werden müssen (was heutzutage üblicherweise der Compiler erledigt).

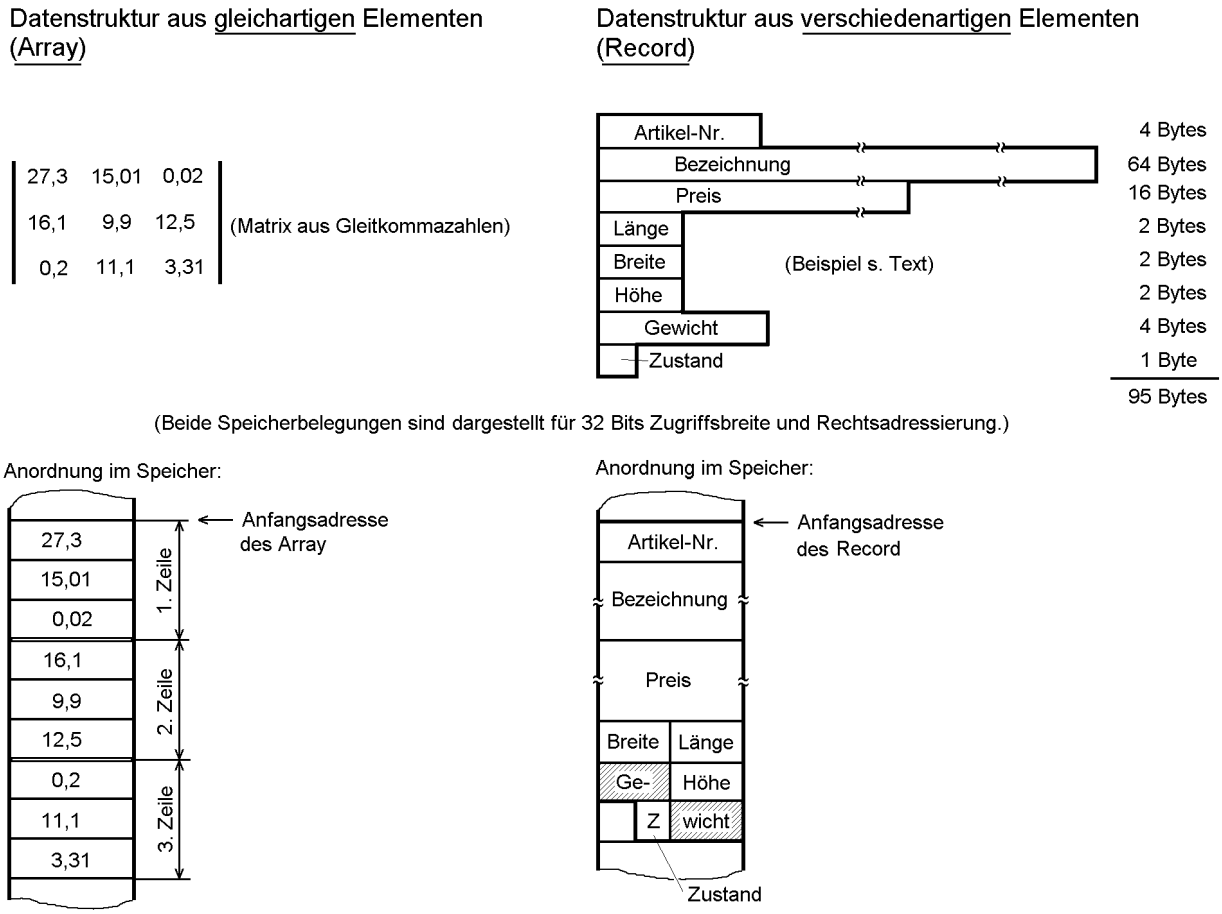


Abbildung 2.15 Beispiele für komplexe Datenstrukturen

2.6.2. Integrale Adressen (integrale Grenzen)

Dieser Begriff ist von Bedeutung, wenn es um die Unterbringung von Informationsstrukturen im Speicher geht.

Die integrale Adresse*) (integrale Grenze) eines 16-Bit-Wortes ist eine gerade Binärzahl (das niedrigstwertige Adreßbit ist gleich Null). Die integrale Adresse eines 32-Bit-Wortes ist eine durch 4 teilbare Binärzahl (die beiden niedrigstwertigen Adreßbits sind gleich Null) usw.

*): auch: Aligned Address (sprich: Äleint Address; das Gegenteil: Non-Aligned bzw. Misaligned...).

Allgemein
Der Begriff der integralen Adresse betrifft Informationsstrukturen, deren Länge (in Bytes) eine Zweierpotenz ist (Abbildung 2.16, Tabelle 2.3).

Wir merken uns:
Integrale Adressen einer Struktur von 2^n Bytes Länge sind durch 2^n teilbar, das heißt in ihren n niedrigstwertigen Bits fest mit Null belegt. Anders ausgedrückt: ist eine solche Struktur n Bits lang, so enthält deren integrale Adresse in ihren ld n niedrigstwertigen Bits Nullen (d. h. in den Bitpositionen ((ld n) - 1)...0). Beispiel: Länge der Struktur: 8 Bytes. ld 8 = 3, also sind die 3 Adreßbits 2...0 mit Nullen belegt.

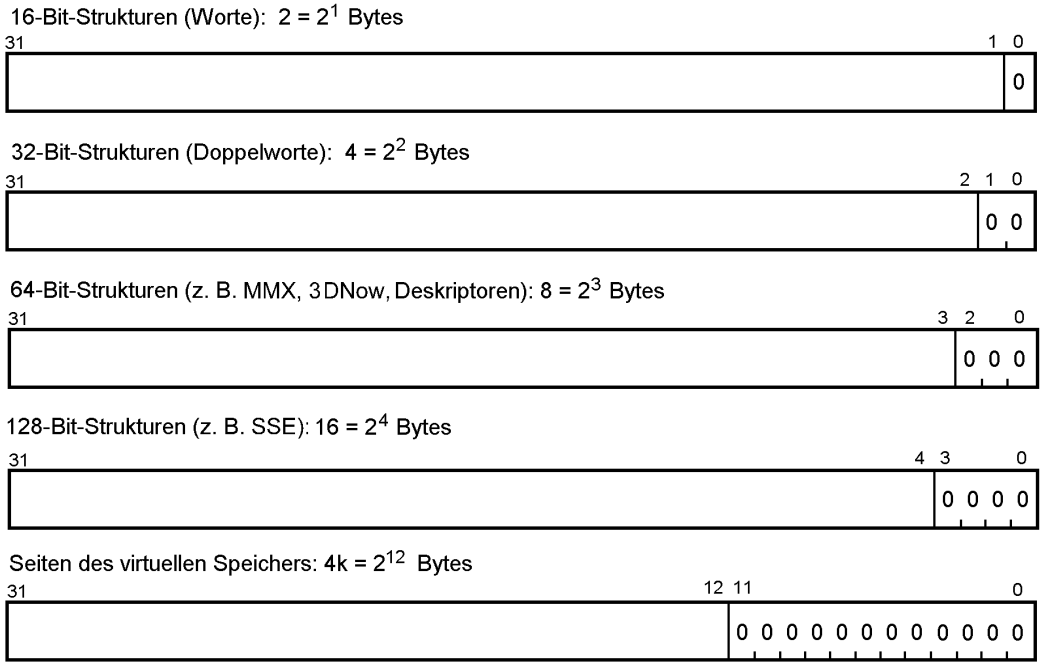


Abbildung 2.16 Beispiele integraler Adressen (IA-32)

Informationsstrukturen	Integrale Adresse teilbar durch
Wort	2
Doppelwort	4
Quadwort (z. B. MMX-Datenstruktur oder Deskriptor)	8
SSE-Datenstruktur oder Cache-Eintrag, 16 Bytes	16
Cache-Eintrag, 32 Bytes	32
Seiten bzw. Seitenrahmen (4k-Seiten)	4096

Tabelle 2.3 Integrale Adressen wichtiger Informationsstrukturen (IA-32)

Der technische Hintergrund
Wir haben Byteadressen, aber Speicher mit größerer Zugriffsbreite. Wenn wir in einem solchen Speicher Datenstrukturen unterbringen, deren Länge (1) größer als 1 Byte ist und (2) höchstens der Zugriffsbreite entspricht, so ist mit zwei Fällen zu rechnen (Abbildung 2.17).

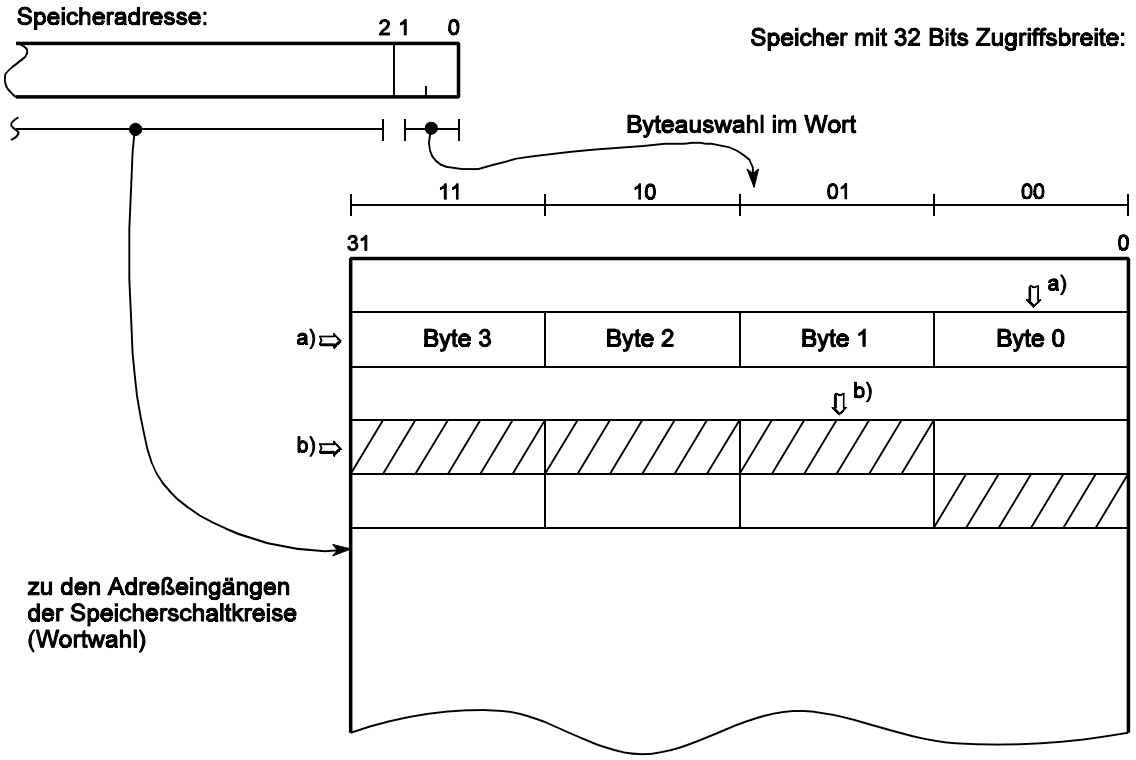


Abbildung 2.17 Zum technischen Hintergrund integraler Adressen

Erklärung:
Der Speicher hat hier eine Zugriffsbreite von 32 Bits = 4 Bytes. Die Adreßbits 1, 0 der Byteadresse wählen das jeweils erste Byte im Speicherwort aus. Die höherwertigen Adreßbits dienen zur Auswahl eines 4-Byte-Wortes (wir können uns vorstellen, daß die entsprechenden Adreßsignale direkt mit den Adreßeingängen der Speicherschaltkreise verbunden sind). Es gibt zwei Zugriffsfälle:

a) die adressierte Datenstruktur ist innerhalb der Zugriffsbreite (eines Speicherwortes) untergebracht. Beispiel: wir greifen auf ein 4-Byte-Wort mit Byteadresse 0 = 00B zu. Offensichtlich genügt dann ein einziger Speicherzugriff.
b) die Datenstruktur überlappt zwei Speicherworte. Beispiel: wir greifen auf ein 4-Byte-Wort mit Byteadresse 1 = 01B zu. Um eine gleichsam mitten im Speicherwort beginnende 4-Byte-Struktur (in der Abbildung schraffiert) zu erreichen, sind zwei Speicherzugriffe erforderlich (Leistungsverlust!).

Es ist ersichtlich, daß nur Adressen, die - gemäß der Zugriffsbreite - in ihren niedrigstwertigen Bitpositionen mit Nullen belegt sind, den Zugriffsfall a) gewährleisten.

Vorsicht, Falle:
Manche Architekturen erfordern stets die Beachtung integraler Grenzen, manche teilweise (z.B. - bei festem Befehlsformat - bei der Befehlsadressierung), manche gar nicht. Aber auch dann muß man gelegentlich aufpassen.

Beispielsweise ist es bei der Architektur IA-32 *nicht* notwendig, Worte, Doppelworte usw. an integralen Adressen zu speichern. So läßt sich der Speicher sehr gut ausnutzen, was namentlich dann von Vorteil ist, wenn aus verschiedenartigen Teilen bestehende Datenstrukturen (die oben erläuterten Records) zu verpacken sind. *Aber:* die IA-32-Prozessoren adressieren zwar aus logischer Sicht bis aufs Byte, greifen aber im angeschlossenen Speicher auf Doppel- oder Quadworte (32/64 Bits) an integralen Adressen zu. Sind Informationsstrukturen nicht an ihren integralen Grenzen gespeichert, so ergeben sich - vgl. Abbildung 2.17 - leistungsmindernde Überlappungen. Wenn IA-32-Maschinen nicht die erwarteten Leistungen bringen, kann eine mögliche Ursache also darin liegen, daß integrale Grenzen nicht berücksichtigt wurden.

Integrale Adressen können auch dann von Bedeutung sein, wenn Daten an Systeme anderer Architektur geliefert werden sollen und diese die Beachtung integraler Adressen erfordern (z.B. verschiedene RISC-Architekturen).

2.6.3. Elementare Adreßrechnung

Im folgenden wollen wir elementare Adressierungsweisen und Rechengänge betrachten, mit denen effektive Adressen berechnet werden können. Manche sind von so grundsätzlicher Bedeutung, daß sie in einem modernen Universalrechner unbedingt zur Architektur gehören, bei anderen ist es hingegen eine Frage der Philosophie (z. B. der Entscheidung zwischen CISC- und RISC-Auslegung), ob sie in den Maschinenbefehlen vorgesehen werden, oder ob erwartet wird, daß sie (in der Regel über die Compiler) mit elementaren Befehlen ausprogrammiert werden.

Universelle Adreßregister (Registeradressierung)
Prinzip: die betreffende Adresse wird direkt aus einem Register entnommen. "Universell" bedeutet hier, daß der Registerinhalt ohne weiteres in beliebige Verarbeitungsabläufe einbezogen werden kann. Schema: Operand := <<Register>> (Registerinhalt adressiert den Speicher, Speicherinhalt wird als Operand verwendet).

Indirekte Adressierung ("Adresse von Adresse")
Eine Adresse ist z. B. als Direktwert im Befehl oder als Registerinhalt gegeben. Der Inhalt der adressierten Speicherposition wird aber nicht als Befehl oder als Datenstruktur, sondern als weitere Adresse interpretiert. Mit dieser Adresse wird dann der eigentliche Zugriff ausgeführt. Das Prinzip entspricht an sich der Registeradressierung; man ist aber bei der Unterbringung von Adressen nicht auf den Registersatz beschränkt, sondern kann den Arbeitsspeicher dazu ausnutzen. Schema: Operand := <<< Register>>> (Registerinhalt adressiert den Speicher, Speicherinhalt wird als Adresse des Operanden verwendet).

Weshalb kommt man mit einem dieser Prinzipien allein nicht aus?
Zum einen verschlechtert sich die Verarbeitungsleistung bedeutend. Allein im Interesse der Verschieblichkeit von Programmen und Daten wäre für jede Verzweigung, für jeden Unterprogrammruf und für nahezu jeden Datenzugriff eine Adreßrechnung notwendig, die jeweils mehrere Befehle erfordert. Zudem wären viele Register mit Adressen belegt; man könnte sie nicht als Schnellspeicher für Daten verwenden. Deshalb hat man

herkömmlicherweise elementare Rechengänge fest vorgesehen (vgl. aber Seite 97 ganz unten). Hierbei wird die effektive Adresse aus verschiedenen Anteilen zusammengesetzt, die üblicherweise Basis, Displacement (= Versatz; sprich: Displehsment)*), Index und Skalierung genannt werden.

*): alternative Bezeichnung: Offset.

Elementare Rechengänge der Adreßrechnung

1. Relativadressierung: Basis + Displacement

2. Indexadressierung: Basis + Index + Displacement

3. Skalierte Adressierung: Basis + (Index · Skalierung) + Displacement

Unterbringung und Struktur der Adreßangaben
Basis- und Indexadressen sind in Registern untergebracht, Displacementangaben als Direktwerte in Befehlen. Effektive Adresse, Basisadresse und Indexadresse sind vorzeichenlose (natürliche) Binärzahlen. Displacements sind zumeist ganze Binärzahlen (*mit* Vorzeichen). Skalierungsangaben sind natürliche Binärzahlen und - wenn überhaupt vorgesehen - als Direktwerte in Befehlen untergebracht.

Adreßrechnung in der IA-32-Architektur
Der allgemeine Rechengang:

Effektive Adresse = Basis + (Index · Skalierung) + Displacement.

Basis- und Indexadresse werden in einem Universalregister (EAX, EBX usw.) erwartet, Skalierung und Displacement sind Direktwerte im Befehl. Durch Weglassen einzelner Anteile ergeben sich insgesamt 9 Kombinationsmöglichkeiten (darunter auch die oben unter 1 und 2 beschriebenen Rechengänge).

Wieviele Adressierungsweisen braucht man wirklich?
Die Erfahrung hat gezeigt, daß die Form *Basis + Displacement* vollauf ausreichend ist. Die Basisadresse befindet sich dabei 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 eine reine Registeradressierung. Solche Displacementangaben sind üblicherweise zwischen 13 und 16 Bits lang (diese Größenordnung hat sich in der Praxis als ausreichend erwiesen)*). Mit schnell ablaufenden elementaren Befehlen lassen sich auf dieser Grundlage alle komplizierteren Adreßrechnungen ausprogrammieren (RISC-Philosophie).

*): bei IA-32 hat man sich in dieser Hinsicht allerdings einen Schnitzer geleistet: Adreßangaben in den Befehlen sind entweder 1 Byte lang, oder deren Länge entspricht der jeweils aktuellen Adreßlänge. Es gibt zwei Adreßlängen: 16 und 32 Bits. Der Normalfall: Adreßlänge = Operandenlänge = Verarbeitungsbreite (16 Bits bei 16-Bit-Verarbeitung, 32 Bits bei 32-Bit-Verarbeitung). Kommt man also mit Byte-

Displacements nicht aus (deren "Reichweite" beträgt -128...+127 Bytes, bezogen auf die Basisadresse), so müssen Befehle des 32-Bit-Modus 4 Bytes lange Adreßangaben mitschleppen. Nun kann man die Adreßlänge auf 16 Bits herunterschalten, dies erfordert aber - da die Verarbeitungsbreite ja weiterhin bei 32 Bits bleiben soll -, jedem entsprechenden Befehl ein sog. Vorsatzbyte (Address Size Override Prefix) voranzustellen. Ein solchens 16-Bit-Displacement kostet also auch insgesamt 3 Bytes (und 2 würden vollauf genügen...). U. a. deshalb sind 32-Bit-Programme typischerweise deutlich länger als 16-Bit-Programme.

Ganz moderne Architekturen (IA-64)
Es sind genügend viele Register vorhanden. Die Speicherzugriffsbefehle beruhen auf der Registeradressierung, und die eingebaute Adreßrechnung weicht vom Gewohnten ab: (1) der Inhalt des Adreßregisters wird verändert, (2) die Veränderung erfolgt *nach* dem Speicherzugriff.

2.6.4. Befehlsadressierung

Die Befehle, die im Prozessor ausgeführt werden sollen, werden vom Befehlszähler adressiert. Dieser schaltet automatisch von Befehl zu Befehl weiter, wobei - im Falle variabel langer Befehle - die Befehlslänge automatisch verrechnet wird.

Betrachten wir zunächst den Fall, daß sich ein Programm in der Ausführung befindet. Neben der fortlaufenden Adressierung sind gelegentlich Verzweigungen auszuführen und Unterprogramme zu rufen. Wie werden die jeweiligen Befehlsadressen, zu denen verzweigt werden soll, (1) angegeben und (2) als effektive Adressen ermittelt? - Hierfür gibt es zwei Prinzipien:

1. Befehlszähler-relative Adressierung
Dieses Prinzip wird am häufigsten verwendet. Die Adreßangabe im Verzweigungs- oder Unterprogrammrufbefehl wird als ganze Binärzahl interpretiert und - erforderlichenfalls vorzeichengerecht erweitert - zum aktuellen Inhalt des Befehlszählers addiert (der Befehlszähler zeigt dabei anfänglich auf den Folgebefehl). Diese Displacement- bzw. Offset-Angaben sind in verschiedenen Längen vorgesehen (Beispiele: (1) IA-32: 8 oder 16 Bits, (2): Sparc: 22 Bits).

2. Absolutadressierung
Die effektive Adresse wird direkt geliefert und in den Befehlszähler geladen. Eine solche Adresse kann als Direktwert im Befehl angegeben sein oder aus einem Register bzw. aus einer Speicherposition entnommen werden (Beispiel: Start eines Unterbrechungsbehandlers mit einer Adresse, die aus der Interrupttabelle entnommen wird; vgl. Abschnitt 4.3.2.).

2.6.5. Stackorganisation und -adressierung

Stacks haben wir bereits mehrfach erwähnt. 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

implementiert werden, deren Verwaltung mit elementaren Befehlen auszuprogrammieren ist). Die Prinzipien bleiben aber stets die gleichen.

Stackzugriffe
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. Es gibt nur zwei grundlegende Zugriffsabläufe:

- ein *Push*-Ablauf (sprich: Pusch...) 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.

Ausdehnungsrichtung
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 niederer 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. Eine typische Auslegung (auch von x86/IA-32): der Stackpointer zeigt immer auf das oberste Element im Stack (Top of Stack TOS). Bei einem Push wird der Stackpointer-Inhalt zunächst vermindert (Ausdehnungsrichtung!); dann wird das Element gespeichert. Umgekehrt wird bei einem Pop das Element entnommen und dann der Stackpointer-Inhalt erhöht (Fachbegriff des Zugriffsprinzips: Predecrement/Postincrement).

Variabel lange Stackelemente?
Typischerweise sind alle Elemente in einem Stack *gleich lang*. Kürzere Angaben werden zwecks Ablage auf dem Stack entsprechend erweitert, längere Angaben in mehreren aufeinanderfolgenden Stack-Elementen untergebracht.

Stack-relative Adressierung
Es ist oft von Vorteil, wenn man auf den Inhalt des Stacks auch wahlfrei zugreifen kann. So kann man "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).

Stack Frames
Ein Stack Frame (sprich: S-täck Frehm) 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 beziehen uns auf Abbildung 2.15 und 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.	

*): diese Variable kommt in Abbildung 2.15 nicht vor.

Jeder dieser 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. auch 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. 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 Dis-

placements zu den statischen Variablen ergeben. Deshalb sieht man typischerweise ein weiteres Adreßregister vor, den sog. Frame Pointer oder Base Pointer (Abbildung 2.18).

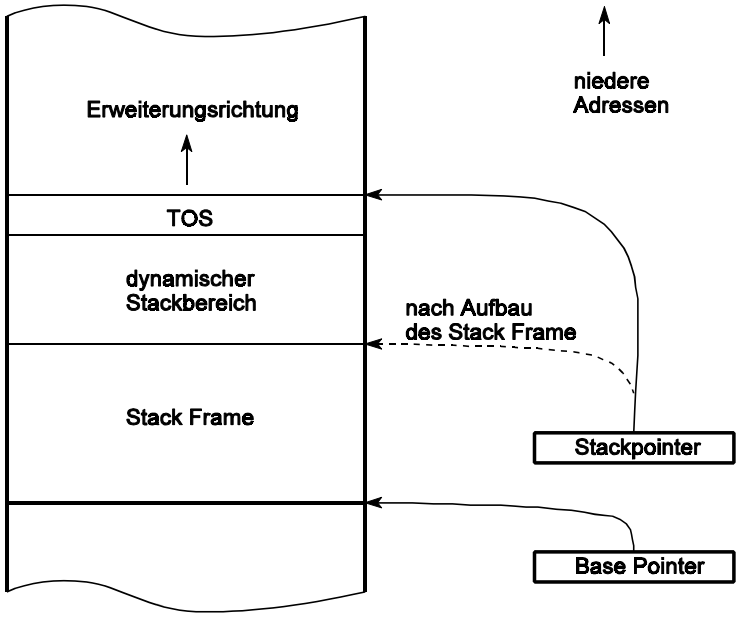


Abbildung 2.18 Stack-Organisation mit Stack Frame

Erklärung:
Der Base 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.

Die Parameterübergabe ist gleichsam umsonst
Der typische Ablauf eines Unterprogrammrufs: das rufende Programm legt zunächst die zu übergebenden Parameter auf den Stack (z. B. mit Push-Befehlen) und ruft dann das Unterprogramm (Call-Befehl). Betrachten wir nun nochmals Abbildung 2.18. Stellen wir uns vor, wir hätten ein Unterprogramm auf diese Weise gerufen. Dabei wäre der Base Pointer mit der Belegung des Stackpointers (vor dem Aufruf) überladen worden. Ersichtlicherweise können wir mit positiven Offsets (bezogen auf den Base Pointer) den dynamischen Stackbereich des rufenden Programms adressieren. Auf diese Weise kann das Unterprogramm die ihm übergebenen Parameter direkt aus dem Stack entnehmen; es sind keine weiteren Transportabläufe erforderlich. Ebenso können Ergebnisse an das rufende Programm zurückgegeben werden.