Heft 02
Elementare Informationsstrukturen

Stand: 1.03

- Vorläufiges Exemplar. Nur zur Information -

(c) Prof. Dr. Wolfgang Matthes 2001

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. Verarbeitungsbreite

Die Verarbeitungsbreite bezeichnet jene Anzahl an Bits, 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 herumgebaut, 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 waren 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*.

Wir sprechen hier grundsätzlich nur von der Verarbeitungsbreite, wie sie der Architektur konzeptionell 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.

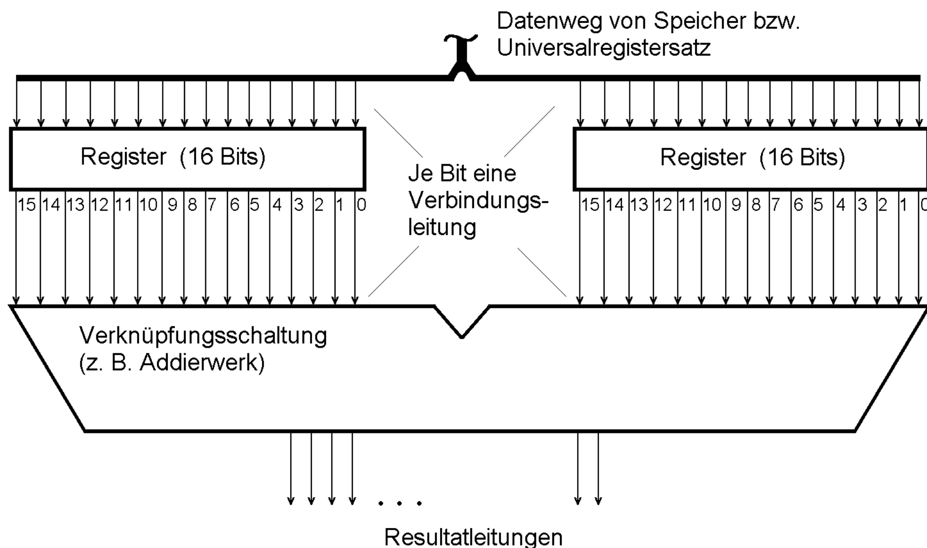


Abbildung 1.1 Zur Erklärung der Verarbeitungsbreite

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,
- Big Endian = Linksadressierung.

Beispiele (Abbildung 1.2):

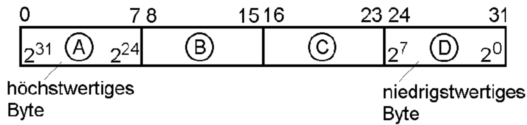
- Little Endian: x86 und IA-32 sowie viele andere Mikroprozessoren,
- Big Endian: die IBM-Mainframes (S/360.../390), PowerPC, JVM (Java Virtual Machine).

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

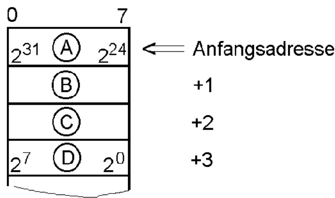
Achtung:
Im Fall des Falles - die Dokumentation genau lesen! Die Adressierungsweise wird zwar umgeschaltet, die Numerierung der Interfacesignale entspricht aber der jeweiligen Vorzugsauslegung (PowerPC: Big Endian, Mips und IA-64: Little Endian; Abbildung 1.3).

Linksadressierung

Beispiele: S/390, Power PC

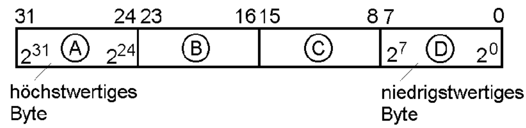


Anordnung im Speicher:



Rechtsadressierung

Beispiel: IA-32



Anordnung im Speicher:

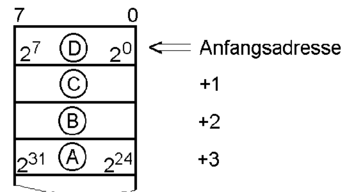
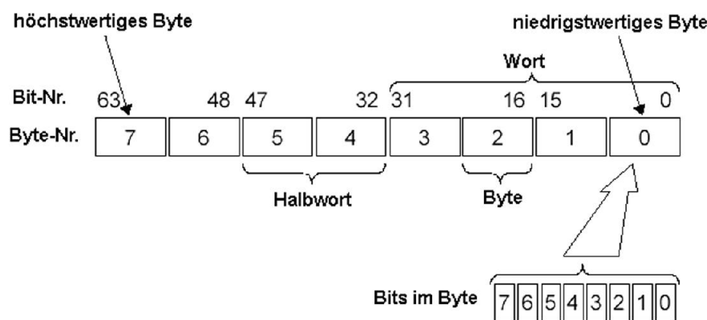


Abbildung 1.2 Links- und Rechtsadressierung am Beispiel von 32-Bit-Worten

a) 64-Bit-Doppelwort bei Rechtsadressierung (Little Endian)



b) 64-Bit-Doppelwort bei Linksadressierung (Big Endian)

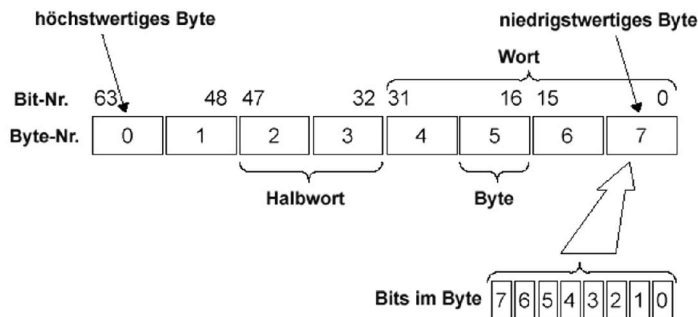


Abbildung 1.3 Umschaltbare Adressierung (Auszug aus einem Architekturhandbuch (NEC))

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*).

*): 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 ≙ 2^0, Bit-Index 1 ≙ 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. Da heutzutage beide Adressierungsweisen weit verbreitet sind, ist die Entscheidung praktisch eine reine Kompatibilitätsfrage.

Hinweis:
Das Problem wurde immerhin als so bedeutsam eingeschätzt, daß die Hersteller entsprechende Lösungen eingeführt haben. So 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.

1.3. Beschreibung der elementaren Datenstrukturen

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

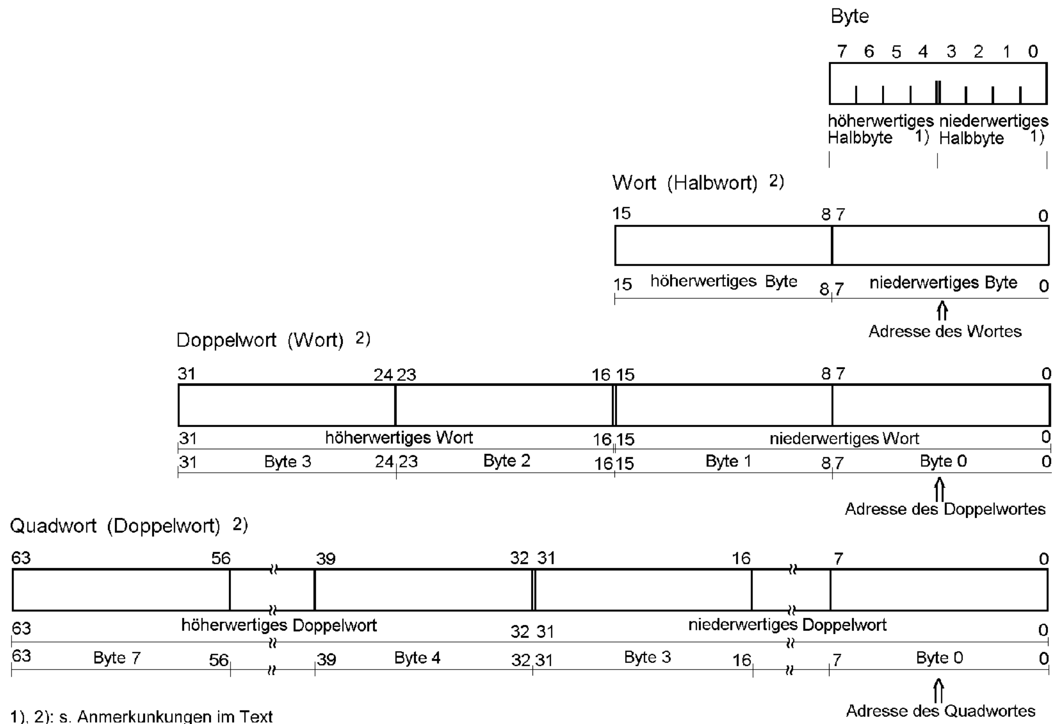


Abbildung 1.4 Adressierbare Behälter

Anmerkungen:

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

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.

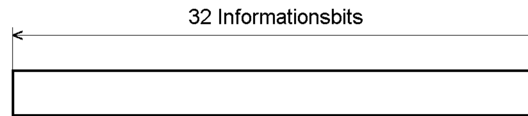
Architekturen mit Datentypkennzeichnung
Die Tatsache, daß Informationsstrukturen als bloße Aneinanderreihungen von Bits gespeichert und transportiert werden, hat von Anfang an wesentlich zur Universalität des digitalen Computers beigetragen. Die Nachteile sind aber offensichtlich:

- # 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. Stattdessen 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 Arkitetschr.

Ohne Typkennzeichnung



Mit Typkennzeichnung

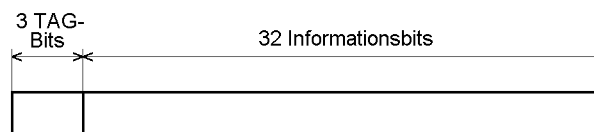


Abbildung 1.5 Datenstrukturen mit und ohne Typkennzeichnung (Beispiele)

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.)

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.

2. Numerische Datentypen

2.1. Zur Einführung: Dezimal- und Binärzahlen

Unser gewohntes Dezimalsystem ist ein Stellenwertsystem. 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 $(9 \cdot 10) + 2$. 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 2.1).

Zahlenwert 583,27

entspricht $5 \cdot 10^2 + 8 \cdot 10^1 + 3 \cdot 10^0 + 2 \cdot 10^{-1} + 7 \cdot 10^{-2}$

Ziffernwert mal

...	1000 (10^3)	100 (10^2)	10 (10^1)	1 (10^0)	0,1 (10^{-1})	0,01 (10^{-2})	...
usw.	Tausenderstelle	Hunderterstelle	Zehnerstelle	Einerstelle	Zehntelstelle	Hundertstelstelle	usw.

Abbildung 2.1 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 2.2).

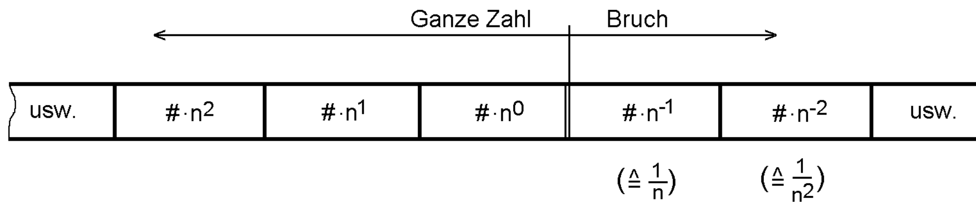


Abbildung 2.2 Zahlendarstellung in einem beliebigen Stellenwertsystem zur Basis n

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 2.3). Der Vorteil: Zahlenwerte lassen sich in einer solchen Darstellung mit technischen Mitteln besonders einfach übertragen, speichern und verarbeiten (Prinzip der Zweiwertigkeit - vgl. Lehrheft 1).

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 2.3 Zahlendarstellung im Binärsystem

Komplemente

Das Komplement eines Zahlenwerts ist dessen Differenz zu einem Festwert, der von der jeweiligen Basis b des Zahlensystems bestimmt wird.

Das (b - 1)-Komplement
Dieses Komplement wird stellenweise gebildet:

$k_{b-1s} = b-1-w_s$

(Der Index s bezeichnet die jeweilige Stelle.)

Das Neunerkomplement einer Dezimalzahl
Es ergibt sich stellenweise zu 9 - w_s (Tabelle 2.1).

Wert	Neunerkomplement	Wert	Neunerkomplement
0	9	5	4
1	8	6	3
2	7	7	2
3	6	8	1
4	5	9	0

Tabelle 2.1 Das Neunerkomplement in einer Dezimalstelle

Das Einerkomplement einer Binärzahl
Es ergibt sich stellenweise zu $1 - w_s$. Das Einerkomplemennt von 0 ist 1, das von 1 ist 0. Demzufolge kann das Einerkomplement durch *Negation* gebildet werden:

$$k_1 s = \overline{w_s}$$

Das b-Komplement
Das b-Komplement einer n-stelligen Zahl z ist die Differenz zur nächsthöheren Potenz b^n bzw. das um 1 erhöhte (b-1)-Komplement:

$$k_b = b^n - z = k_{b-1} + 1$$

Das Zehnerkomplement einer Dezimalzahl

$$k_{10} = 10^n - z = k_9 + 1$$

Das Zweierkomplement einer Binärzahl

$$k_2 = 2^n - z = k_1 + 1$$

Bildung des Zweierkomplements beim praktischen Rechnen
Der Zahlenwert wird zunächst bitweise negiert (Bildung des Einerkomplements). Danach wird eine Eins hinzuaddiert. Zu hardwareseitigen Alternativlösungen siehe Lehrheft 3.

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 2.3 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 also diese Darstellung nicht mit dem logischen Pegel L (für LOW; vgl. Lehrheft 1)! (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: 10010000 11001011B),
- man kann sie kaum aussprechen.

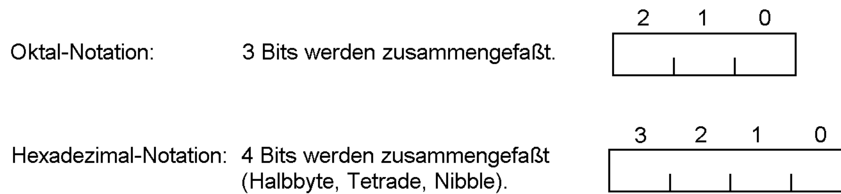
Der Ausweg liegt darin, mehrere Binärstellen in einem Zeichen zusammenzufassen. In einer *Oktalzahl* werden drei aufeinanderfolgende Bits eines Binärvektors zusammengefaßt, in einer *Hexadezimalzahl* vier (Abbildung 2.4). Bei Oktalzahlen wird jedes Bitmuster mit einer Ziffer zwischen 0 und 7 gekennzeichnet, 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 (beim IBM System /360) setzte sich die Hexadezimaldarstellung nach und nach durch.

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



Oktal

0	0	0	-	0
0	0	1	-	1
0	1	0	-	2
0	1	1	-	3
1	0	0	-	4
1	0	1	-	5
1	1	0	-	6
1	1	1	-	7

Hexadezimal

0	0	0	0	-	0
0	0	0	1	-	1
0	0	1	0	-	2
0	0	1	1	-	3
0	1	0	0	-	4
0	1	0	1	-	5
0	1	1	0	-	6
0	1	1	1	-	7
1	0	0	0	-	8
1	0	0	1	-	9
1	0	1	0	-	A
1	0	1	1	-	B
1	1	0	0	-	C
1	1	0	1	-	D
1	1	1	0	-	E
1	1	1	1	-	F

Oktal: 25_8 = 0 1 0 1 0 1B

Hexadezimal: $3B_H$ = 0 0 1 1 1 0 1 1B

andere Notationsweisen: X3B; 3Bx; x '3B'; 3BH

Abbildung 2.4 Oktal- und Hexadezimalnotation

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; freie Bitpositionen werden als Null angenommen. Längere Binärvektoren 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.

Hinweise:

1. Oktalzahlen sind in heutzutage bedeutungslos.
2. Wir verwenden ausschließlich Hexadezimalzahlen und kennzeichnen sie durch ein nachgestelltes "H".
3. Was wirklich “sitzen” sollte: die Wandung von Hexadezimalen in’s Binäre und umgekehrt (Hexadezimalzahlen sind in der professionellen Dokumentation nahzu allgegenwärtig, Konfigurationsangaben, Fehlermeldungen usw. werden oft in hexadezimaler Form ausgegeben).
4. Zum Rechnen mit Binär- und Hexadezimalzahlen: es gibt besondere Taschenrechner-Modelle, die entsprechende Funktionen haben. Auch der in Windows enthaltene Rechner ist brauchbar.
5. Im Anhang haben wir Anregungen zum Rechnen mittels PC und 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 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 werden gelegentlich dezimal angegeben. So entspricht der ASCII-Code 32 = 20H dem Leerzeichen, der Code 71 = 47H dem Zeichen “G” usw.

2.3. Natürliche Binärzahlen

Natürliche Zahlen sind vorzeichenlos*). Der niedrigste Wert ist Null. Der gesamteWertebereich einer natürlichen Binärzahl x aus n Bits (Abbildung 2.5) ist gegeben durch:

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

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

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

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

15	0
Wert	

Abbildung 2.5 Beispiel einer natürlichen (vorzeichenlosen) Binärzahl (16 Bits)

Elementare Formate
In Tabelle 2.2 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 2.2 Natürliche Binärzahlen als elementare Datentypen

Ordinal- und Kardinalzahlen
Die Begriffe stammen aus der Mengenlehre. Eine *Ordinalzahl* ist gleichsam die laufende Nummer eines Elementes in einer geordneten Menge. In diesem Sinne sind natürliche Binärzahlen als Ordinalzahlen die Grundlage der Adressierung. Des weiteren werden natürliche Binärzahlen zur Angabe von Anzahlen verwendet (in der Mengenlehre: Mächtigkeiten bzw. Kardinalitäten; deshalb werden solche Angaben als *Kardinalzahlen* bezeichnet).

Achtung: In vielen Informationsstrukturen *zählt der Wert Null mit*: eine Ordinalzahl Null wählt das jeweils *erste* Element aus (beispielsweise adressiert die Adresse Null das erste Byte im Speicher) und nicht etwa keines.

Für *Kardinalzahlen* gibt es drei Interpretationsweisen:

1. der Wert n bezeichnet n Elemente; eine Kardinalzahl Null kennzeichnet eine leere Menge (also “nichts”).
2. eine Kardinalzahl Null steht für die Anzahl Eins; eine Kardinalzahl mit Wert n bezeichnet (n+1) Elemente.

3. eine Kardinalzahl x aus n Bits wird "modulo 2^n" interpretiert, das heißt im Wertebereich $1 \leq x \leq 2^n$. 1 entspricht einem Element, 2 entspricht zwei Elementen usw. (bis 2^{n-1}). Der Wert Null steht für 2^n Elemente.

Beispiele: In der Architektur IA-32 gilt die erste Interpretation beispielsweise für Zählangaben in Wiederholungsbefehlen, die zweite für Längenangaben in Segmentdeskriptoren und die dritte für die Durchlaufzahl in Schleifenbefehlen.

In der Praxis kommen auch natürliche Binärzahlen abweichender Länge vor (13 Bits, 20 Bits usw.). Will man solche Zahlen in Programmen verarbeiten, so müssen sie *rechtsbündig* aufbereitet werden. Dazu muß das niedrigstwertige Bit in Bit 0 des niedrigstwertigen Bytes stehen. Von der höchsten Stelle an bis zum Ende des jeweiligen Verarbeitungs-Formates (z. B. 32 Bits) sind Nullen aufzufüllen (Nullerweiterung; Zero Extend).

2.4. Ganze Binärzahlen

Ganze Zahlen*) (Abbildung 2.6) haben ein Vorzeichen (+ oder -). Das Vorzeichen belegt ein Bit, und zwar jeweils das höchstwertige (Most Significant Bit MSB).

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

Abbildung 2.6 Beispiel einer ganzen Binärzahl (16 Bits). S = Vorzeichen (Sign)

Zahlendarstellung
In allen modernen Architekturen wird die 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. Da die höchstwertige Stelle vom Vorzeichenbit (hier = 0) belegt ist, hat eine n Bits lange ganze Binärzahl x einen positiven Wertebereich von $0 \leq x \leq 2^{n-1} - 1$.

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:

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

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

niedrigster 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,
höchster 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 (im besonderen ist eine negative Zahl um so kleiner, "je negativer" sie ist). Beispiele: - 8 ist kleiner als - 3 (- 8 < - 3) und - 2 ist kleiner als + 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 2.3 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 2.3 Ganze Binärzahlen (Integer-Zahlen) als elementare Datentypen

2.5.2. Einführung in das Rechnen mit Binärzahlen

Addieren und Subtrahieren
Im Binären wird genau so 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 2.7...2.9).

Addition

0	1	0	1	1	3	0 0 1 1
+ 0	+ 0	+ 1	+ 1	+ 1_{+1}	+ 6	+ 0_1 1_1 1 0
0	1	1	1 0	1 1	9	1 0 0 1

einlaufender Übertrag

Übertrag (1 + 1 = 2)

(1 + 1 + 1 = 3)

Subtraktion

0	1	1	0	0 a)	1 b)	9	1 0 0 1
- 0	- 0	- 1	- 1	- 1_{+1}	- 1_{+1}	- 3	- 0_1 0_1 1 1
0	1	0	1 1	1 0	1 1	6	0 1 1 0

geborgt

geborgte 1

(Wir ergänzen von 1 auf 2
und borgen uns eine Eins
von der nächst-höheren Stelle.)

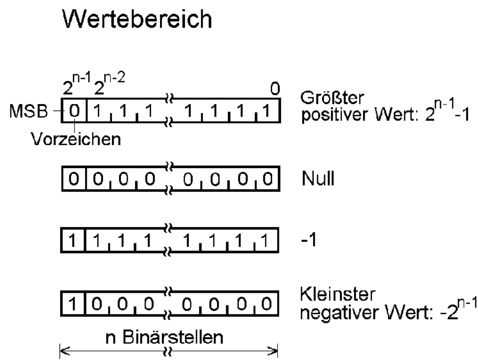
Verrechnung der geborgten Eins:
a) Ergänzung von 2 auf 2
b) Ergänzung von 2 auf 3

Abbildung 2.7 Addieren und Subtrahieren von Binärzahlen

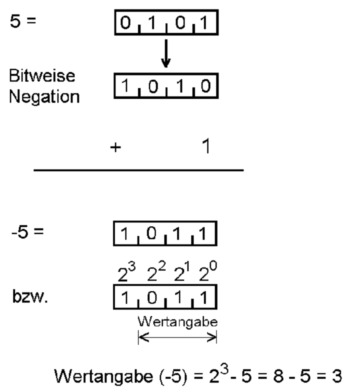
Addieren und Subtrahieren im Computer

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

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 2.9). 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 Verarbeitungsbreite bzw. Operandenlänge betrachten).



Bildung des Zweierkomplements



Rechenbeispiele

4-stellige ganze Binärzahlen: S 2^2 2^1 2^0

Werte

2 = 0010
3 = 0011
5 = 0101
7 = 0111
-2 = 1110
-3 = 1101
-5 = 1011
-7 = 1001

↙ bezeichnet einen Übertrag in die Vorzeichenstelle

c bezeichnet einen Ausgangsübertrag (aus der Vorzeichenstelle heraus)

Positives Resultat, im Wertebereich

a) 2 + 5 = 7
```
  0010
 +0101
  0111
```

c) -2 + 5 = 3
```
  1110
 +0101
 c0011
```

Positives Resultat, Bereichsüberschreitung

b) 5 + 7 = 12
```
  0101
 +0111
  1100
```

Negatives Resultat, im Wertebereich

d) -5 + 2 = -3
```
 -1011
 +0010
  1101
```

f) -5 + (-2) = -7
```
  1011
 +1110
 c1001
```

Negatives Resultat, Bereichsüberschreitung

e) -5 + (-7) = -12
```
  1011
 +1001
 c0100
```

Abbildung 2.8 Elementares Rechnen mit ganzen Binärzahlen

Über- und Unterschreiten des Wertebereichs

1. beim Rechnen mit natürlichen Binärzahlen

Entsteht der Ausgangsübertrag beim Addieren (A + B), so überschreitet das Ergebnis den Wertebereich (Abbildung 2.9a). Ist das eigentliche (mit dem Ausgangsübertrag um eine Stelle längere) Rechenergebnis gleich r ($r \geq 2^n$), 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 ensteht ein negativer Wert - der im Bereich der natürlichen Zahlen nicht zulässig ist). Ist das eigentliches Ergebnis eine negative Zahl - r, so entspricht das Resultat r_n bei n Bits Operandenlänge dem Zweierkomplement des eigentlichen Ergebnisses: $r_n = 2^n - r^{*)}$ (Abbildung 2.9b).

*): dieses Umschlagen des Ergebnisses wird auch als "Wrap Around" bezeichnet.

Hinweis: Siehe auch Abschnitt 2.3. (Sättigungsarithmetik).

4-stellige natürliche Binärzahlen

n = 4, $2^n = 16$

a)

```
   8     1 0 0 0
+  9     1 0 0 1
 ----   ---------
  17   1 0 0 0 1
       ↘
         └──┬──┘
            ↓
         0 0 0 1  ≙ 17 - 16
```

↘ : Ausgangsübertrag

b)

```
   5     0 1 0 1
- 10     1 0 1 0
 ----   ---------
 - 5   1 1 0 1 1
       ↘
         └──┬──┘
            ↓
         1 0 1 1  ≙ - 5 *)
```

*) : Zweierkomplement, vgl. Abbildung 1.13

Abbildung 2.9 Über- und Unterschreiten des Wertebereichs beim Rechnen mit natürlichen Binärzahlen

Hinweis:
In Abbildung 2.9 haben wir schulmäßig subtrahiert (vgl. Abbildung 2.7). Eine Unterschreitung des kleinsten zulässigen Wertens (Null) führt hier zu einem Ausgangsübertrag (= Borgen von der nächst-höheren Stelle). Wird hingegen durch Addieren des Zweierkomplements subtrahiert, so verhält es sich genau umgekehrt (Tabelle 2.4). Siehe weiterhin Seite 24.

Rechenart	Ausgangsübertrag, wenn...	
	Resultat im Wertebereich	Resultat außerhalb des Wertebereichs
Addieren	nein	ja
Subtrahieren	ja	nein

Tabelle 2.4 Über- und Unterschreiten des Wertebereichs beim Rechnen mit natürlichen Binärzahlen im Zweierkomplement

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.

Bereichsüberschreitung
Wird die größte positive Zahl überschritten, so entsteht nur ein Übertrag in die Vorzeichenstelle, aber kein Ausgangsübertrag (Abbildung 2.8b). Das Resultat r_n bei n Bits Operandenlänge ist ein negativer Wert: $r_n = -(2^n - r)$.

Bereichsunterschreitung
Wird die kleinste negative Zahl unterschritten, so entsteht nur ein Ausgangsübertrag, aber kein Übertrag in die Vorzeichenstelle (Abbildung 2.8e). Das Resultat r_n bei n Bits Operandenlänge ist ein positiver Wert: $r_n = 2^n + r$.

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

Bestimmung der Overflow-Bedingung:
Overflow = Ausgangsübertrag ≠ Übertrag in die Vorzeichenstelle = Ausgangsübertrag ⊕ Übertrag in die Vorzeichenstelle.

Rekonstruktion des Übertrags in die Vorzeichenstelle
Dieser ist zumeist unzugänglich. Er kann aber aus den beiden höchstwertigen Operandenbits A, B und dem höchstwertigen Summenbit S rekonstruiert werden (Tabelle 2.5).

A	B	S bei aktivem Eingangsübertrag C_n
0	0	1 (müßte ohne C_n 0 sein; 0 + 0 => 0)
0	1	0 (müßte ohne C_n 1 sein; 0 + 1 => 1)
1	0	0 (müßte ohne C_n 1sein; 0 + 1 => 1)
1	1	1 (müßte ohne C_n 0 sein; 1 + 1 => 0)

Tabelle 2.5 Belegungen der Operandenbits A, B und des Summenbits S bei aktivem Eingangsübertrag

Erklärung:
Um mit bekannten Operandenbits ein bestimmtes Ergebnis zu bilden, muß der Eingangsübertrag in die betreffende Stelle jeweils einen bestimmten Wert haben. In Tabelle 2.5 sind die Wertekombinationen aus den Operandenbits A, B und dem Summenbit S angeführt, die sich nur dann ergeben können, wenn der Eingangsübertrag $C_n = 1$ ist. Ersichtlicherweise ist dies dann der Fall, wenn die Anzahl der Einsen über A, B, S ungerade ist. Somit genügt eine Antivalenzverknüpfung, um C_n zu rekonstruieren:

$C_n = A \oplus B \oplus S$.

Die Overflow-Bedingung ergibt sich sinngemäß zu $A \oplus B \oplus S \oplus C_o$.

Vergleichen zweier Binärzahlen

Wir vergleichen 2 Binärzahlen A, B miteinander, indem wir sie voneinander subtrahieren und bestimmte Bedingungen auswerten. Diese Bedingungen werden in den üblichen Prozessoren als Flagbits bzw. Bedingungscodes gespeichert (Tabelle 2.6). Sie können zwecks Programmverzweigung abgefragt werden (in JMP- bzw. BRANCH-Befehlen). Aus den Tabellen 2.7 und 2.8 sind die typischen Verzweigungsbedingungen ersichtlich. Tabelle 2.9 zeigt anhand von Beispielen, wie die Verzweigungsbedingungen beim Rechnen mit ganzen (vorzeichenbehafteten) Binärzahlen entstehen.

Typische Bezeichnung	Benennung	Bedeutung
ZF	Zero Flag	Ergebnis = 0
CF	Carry Flag	Addition: Ausgangsübertrag = 1, Subtraktion: Auslegung 1: Ausgangsübertrag = 1*) , Auslegung 2: Ausgangsübertrag = 0*)
OF	Overflow Flag	Overflow = 1
SF	Sign Flag (auch: Negative Flag)	Vorzeichen (= höchstwertige Bitposition) = 1. Wert ist negativ

*): siehe den folgenden Text

Tabelle 2.6 Flagbits

Ausgangsübertrag und Carry Flag (CF)

Es gibt zwei Auslegungen:

1. die naive Auslegung: das Carry Flag entspricht dem Ausgangsübertrag der Zweierkomplement-Arithmetik. Es schaltet demzufolge gemäß Tabelle 2.4. Das Ergebnis liegt außerhalb des Wertebereichs, wenn beim Addieren CF = 1 gebildet wird und beim Subtrahieren CF = 0. Beispiele: die PIC-Mikrocontroller der Fa. Microchip. CF = 0 entspricht einem Borgen von der nächst-höheren Binärstelle.
2. die Vorzugsauslegung: das Carry Flag wird in Abhängigkeit von der jeweiligen Rechenoperation gestellt (viele Controller und Prozessoren sind so ausgelegt: Atmel AVR, Intel x86 usw.):
   - Addieren: Carry Flag = Ausgangsübertrag (gemäß Zweierkomplement-Arithmetik),
   - Subtrahieren: das Carry Flag soll das Borgen aus der nächst-höheren Binärstelle kennzeichnen. Zu borgen ist aber nur dann, wenn der Minuend kleiner ist als der Subtrahend. (Beim Rechengang A - B also dann, wenn $A < B$.) Beim Rechnen im Zweierkomplement entsteht aber ein Ausgangsübertrag nur dann, wenn $A \geq B$. Das Carry Flag entspricht somit beim Subtrahieren dem invertierten Ausgangsübertrag. Der Vorteil: beim Rechnen mit natürlichen (vorzeichenlosen) Binärzahlen zeigt CF = 1 stets (sowohl beim Addieren als auch beim Subtrahieren) an, daß das Resultat außerhalb des Wertebereichs liegt.

Vergleichen natürlicher Binärzahlen. Rechengang: A - B			
Vergleichsaussage	Bedingung[1)]	Flagbits[2)]	typische Bezeichnung[*3)]
A = B	Ergebnis = 0 (sowie Ausgangsübertrag)	ZF = 1	Equal
A ≠ B	Ergebnis ≠ 0	ZF = 0	Not Equal
Auslegung 1: CF = Ausgangsübertrag der Zweierkomplementrechnung[2)]			
A < B	kein Ausgangsübertrag (≙ Borgen)	CF = 0	Below
A > B	Ergebnis ≠ 0 und Ausgangsübertrag (≙ kein Borgen)	$\overline{ZF} \cdot CF = 1$ bzw. $ZF \vee \overline{CF} = 0$	Above
A ≤ B	Ergebnis = 0 oder kein Ausgangsübertrag (≙ Borgen)	$ZF \vee \overline{CF} = 1$	Below or Equal
A ≥ B	Ausgangsübertrag (≙ kein Borgen)	CF = 1	Above or Equal
Auslegung 2: CF = invertierter Ausgangsübertrag der Zweierkomplementrechnung[2)]			
A < B	kein Ausgangsübertrag (≙ Borgen ≙ CF = 1)	CF = 1	Below
A > B	Ergebnis ≠ 0 und Ausgangsübertrag (≙ kein Borgen≙ CF = 0)	$\overline{ZF} \cdot \overline{CF} = 1$ bzw. $ZF \vee CF = 0$	Above
A ≤ B	Ergebnis = 0 oder kein Ausgangsübertrag (≙ Borgen ≙ CF = 1)	$ZF \vee CF = 1$	Below or Equal
A ≥ B	Ausgangsübertrag (≙ kein Borgen≙ CF = 0)	CF = 0	Above or Equal

1): Zweierkomplement-Arithmetik; 2) siehe Seite 24; 3): in Befehlsbeschreibungen

Tabelle 2.7 Vergleichen natürlicher (vorzeichenloser) Binärzahlen

Vergleichen ganzer Binärzahlen. Rechengang: A - B			
Vergleichsaussage	Bedingung	Flagbits*)	typische Bezeichnung**)
A = B	Ergebnis = 0	ZF = 1	Equal
A ≠ B	Ergebnis ≠ 0	ZF = 0	Not Equal
A < B	Ergebnis negativ und Überlauf oder Ergebnis positiv und kein Überlauf	SF ≠ OF SF ⊕ OF = 1	Less
A > B	Ergebnis ≠ 0 und Ergebnis negativ und kein Überlauf oder Ergebnis positiv und Überlauf	ZF = 0 und SF = OF ZF ∨ (SF ⊕ OF) = 0	Greater
A ≤ B	Ergebnis = 0 oder Ergebnis negativ und Überlauf oder Ergebnis positiv und kein Überlauf	ZF oder SF ≠ OF ZF ∨ (SF ⊕ OF) = 1	Less or Equal
A ≥ B	Ergebnis negativ und kein Überlauf oder Ergebnis positiv und Überlauf	SF = OF SF ⊕ OF = 0	Greater or Equal

*) siehe Tabelle 2.6; **): in Befehlsbeschreibungen

Tabelle 2.8 Vergleichen ganzer (vorzeichenbehafteter) Binärzahlen

Addieren und Subtrahieren beliebig langer Binärzahlen

In den meisten Architekturen kann der Ausgangsübertrag der Zweierkomplementrechnung (Carry Out) als Eingangsübertrag (Carry In) in nachfolgende Rechnungen einfließen (z. B. mit Befehlen "Addieren/Subrahieren mit Eingangsübertrag" - ADC/SBC - der Architekturen x86/IA-32). Somit kann man Rechenoperationen mit beliebig langen natürlichen Binärzahlen programmieren.

Der Eingangsübertrag wird typischerweise vom Carry Flag CF abgeleitet:

Addieren: CF = Ausgangsübertrag. Demgemäß wird CF als Eingangsübertrag verwendet.

Subtrahieren: da das Zweierkomplement addiert wird, muß auch der Ausgangsübertrag der vorhergehenden Zweierkomplementaddition in der nachfolgenden verrechnet werden:

- Auslegung 1: CF wird als Eingangsübertrag verwendet,
- Auslegung 2: die invertierte Belegung von CF wird als Eingangsübertrag verwendet.

Rechenbeispiel	Rechengang	Vergleichsergebnis	OF	SF
5 - 3	0 1 0 1 1 1 0 1 1 0 0 1 0		0	0
3 - (-3)	0 0 1 1 0 0 1 1 0 1 1 0		0	0
-3 - (-5)	1 1 0 1 0 1 0 1 1 0 0 1 0	A > B	0	0
3 - (-5)	0 0 1 1 0 1 0 1 1 0 0 0		1	1
5 - (-5)	0 1 0 1 0 1 0 1 1 0 1 0		1	1
3 - 5	0 0 1 1 1 0 1 1 1 1 1 0		0	1
-5 - 3	1 0 1 1 1 1 0 1 1 1 0 0 0	A < B	0	1
-3 - 5	1 1 0 1 1 0 1 1 1 1 0 0 0		0	1
-5 - (-3)	1 0 1 1 0 0 1 1 1 1 1 0		1	1
5 - 5	0 1 0 1 1 0 1 1 1 0 0 0 0	A = B	1	0
-5 - (-5)	1 0 1 1 0 1 0 1 1 0 0 0 0		1	0

Tabelle 2.9 Vergleichen ganzer Binärzahlen: Rechenbeispiele

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 ist dabei als fünfte Binärstelle eingerechnet).

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). Das Ergebnis ist wiederum eine natürliche (in den Beispielen vierstellige) Binärzahl.

So läßt sich das Beispiel e) auffassen als - 5 + 9 = 4 bzw. als 11 - 7 = 4; Beispiel f) als -5 + 14 = 9 bzw. als 11 - 2 = 9.

Hinweis:
In manchen Architekturen (z. B. Mips) sind gesonderte Additions- und Subtraktionsbefehle für natürliche und 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 (beim Rechnen mit ganzen Zahlen wird hingegen eine Ausnahmebedingung (Overflow Exception) wirksam, falls ein Überlauf auftritt).

Multiplizieren und Dividieren
Im Binären wird genau so multipliziert und dividiert, wie wir es aus dem Schulunterricht vom Dezimalen her kennen (Abbildung 2.10).

Multiplikand Multiplikator

```
    1 1 0 0 • 1 0 0 1   (12 • 9)

       1 1 0 0          (• 1)
     0 0 0 0            (• 0)
   0 0 0 0              (• 0)
 1 1 0 0                (• 1)
 ---------------
 0 1 1 0 1 1 0 0        (108)
```

Anzahl der Resultatstellen
= Summe der Stellenzahlen von Multiplikand und Multiplikator.

Beim ganzzahligen Multiplizieren wird vorzeichengerecht erweitert.

In den niederen Stellen (gemäß der Stellenzahl des Multiplikanden) ergeben dieselben Operandenbitmuster dasselbe Resultatbitmuster, gleichgültig ob vorzeichenlos oder ganzzahlig multipliziert wird.

Dividend Divisor

```
1 1 0 1 0 1 : 1 0 1 0    (53 : 10)

1 1 0 1 0 1 : 1 0 1 0 = 1 0 1  Rest 11
1 0 1 0                        (5 Rest 3)
-------
0 0 1 1 0
    1 1 0 1
    1 0 1 0
    -------
    0 0 1 1
```

Bei üblichen Divisionsbefehlen ist der Dividend doppelt so lang wie der Divisor.

Quotient erscheint (bei Rest $\neq 0$) gerundet in Richtung Null (Stellen nach dem Komma werden abgeschnitten).

$a \cdot 2^n$: Linksverschiebung um n Stellen.

$a : 2^n$: Rechtsverschiebung um n Stellen.

Abbildung 2.10 Multiplikation und Division von Binärzahlen anhand von Beispielen

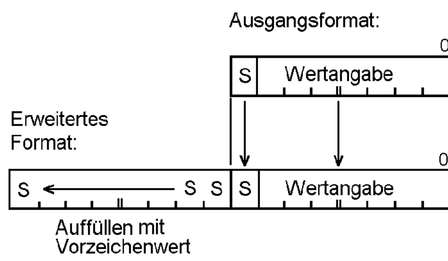
Rechnen mit kurzen Binärzahlen

Sind Zahlen kürzer als die Verarbeitungsbreite, so werden sie *vorzeichengerecht* erweitert (Sign Extend). Dabei wird das Vorzeichen in *alle* auszufüllenden Stellen eingetragen (Abbildung 2.11).

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).



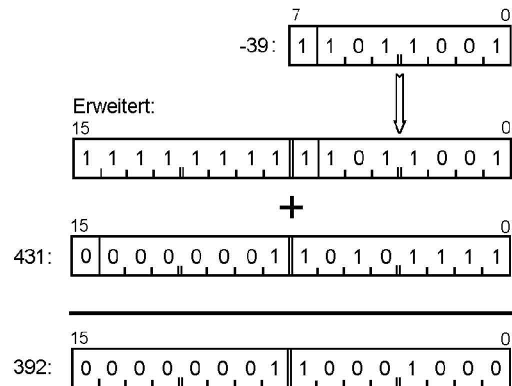


Abbildung 2.11 Prinzip der Vorzeichenerweiterung ganzer Binärzahlen

2.5.3. Herkömmliche Arithmetik und Sättigungsarithmetik

Den Begriff “Wrap-Around-Arithmetik” kennen wir bereits (Seite 21). Abbildung 2.12 veranschaulicht diesen Begriff noch deutlicher.

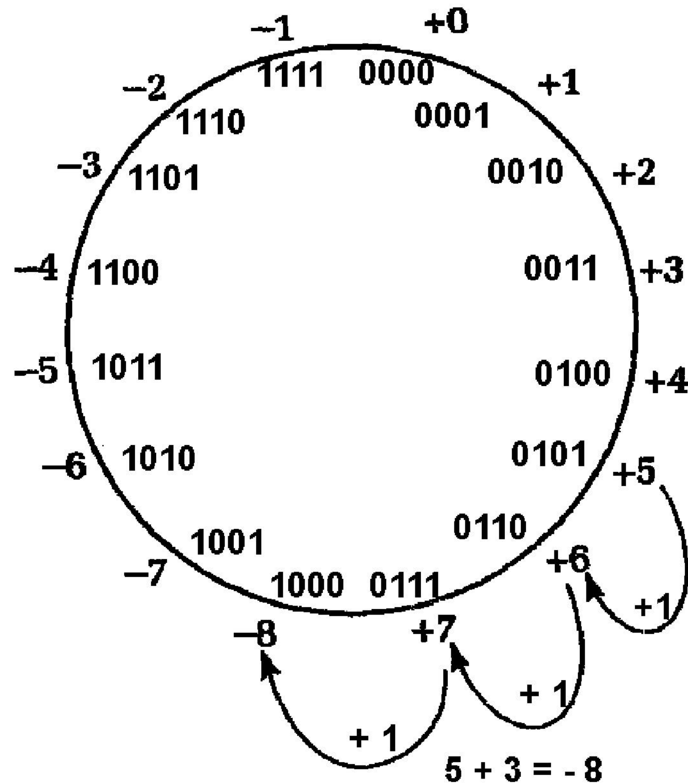


Abbildung 2.12 Zweierkomplement-Arithmetik als Wrap-Around-Arithmetik

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 2.12 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.

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 2.9): wird der Wertebereich überschritten, so wird als Ergebnis der jeweilige Größtwert geliefert, wird der Wertebereich unterschritten, der jeweilige Kleinstwert.

Dies 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.

Hinweis:
Die gleiche Wirkung ließe sich auch erreichen, indem man die Überlaufbedingung auswertet und die Ergebnisse entsprechend korrigiert. Das kostet allerdings Zeit. Es ist aber 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)	P Bereichsunterschreitung ergibt 2^n - Resultat (Zweierkomplement), P Bereichsüberschreitung ergibt Resultat - 2^n	P Bereichsunterschreitung ergibt stets Null, P Bereichsüberschreitung ergibt stets den größten Wert (2^n-1; FFF...FH)
ganze (vorzeichen-behaftete) Binärzahlen (n Bits)	P Bereichsunterschreitung ergibt positiven Wert (2^n + Resultat), P Bereichsüberschreitung ergibt negativen Wert (- (2^n -Resultat))	P Bereichsunterschreitung ergibt stets den kleinsten negativen Wert - 2^{n-1} (800..0H), P Bereichsüberschreitung ergibt stets den größten positiven Wert (2^{n-1} - 1; 7FF...FH)

Tabelle 2.10 Herkömmliche und Sättigungsarithmetik

2.5.4. 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 2.13).

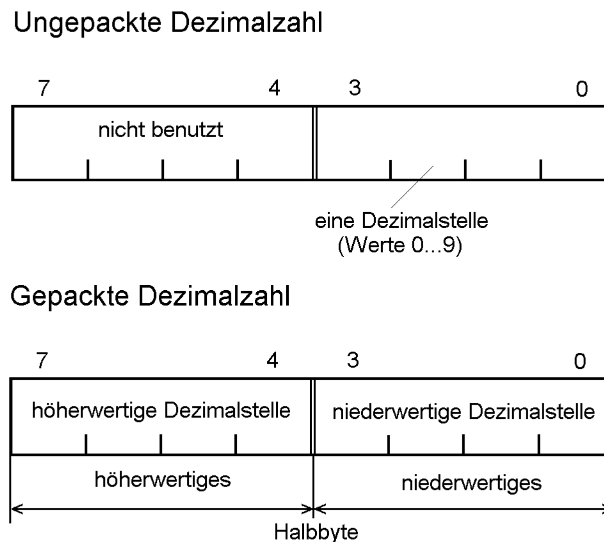


Abbildung 2.13 Binär codierte Dezimalzahlen (BCD-Zahlen)

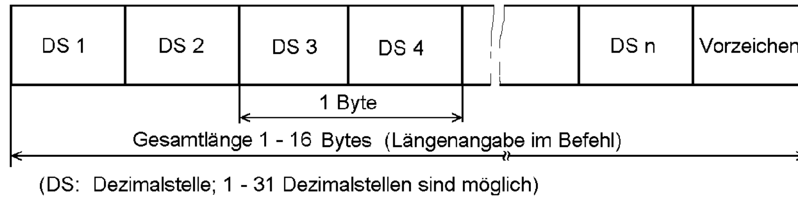
Eine ungepackte Dezimalzahl ist ein einzelnes Byte mit einer einzigen Dezimalstelle im niederwertigen Halbbyte (Bits 0...3). Gepackte BCD- Zahlen enthalten in jedem Byte zwei Dezimalstellen, wobei die Stelle im höherwertigen Halbbyte die höherwertige ist. Diese Datenstrukturen sind an sich vorzeichenlos. Ein Vorzeichen muß gesondert codiert werden (Vorzeichen und Betrag sind voneinander unabhängig; Sign/Magnitude-Darstellung). Abbildung 2.14 zeigt Beispiele von BCD-Zahlen.

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. S / 370



2. x86 BCD-Format für Gleitkommaverarbeitung
(10 Bytes; wird automatisch in Gleitkommazahl gewandelt)

Das niederwertige Halbbyte enthält die niedrigstwertige Dezimalstelle.

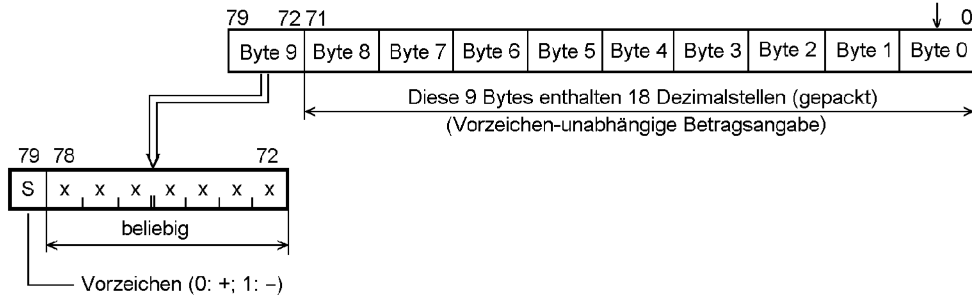


Abbildung 2.14 Formate vorzeichenbehafteter BCD-Zahlen (Beispiele)

3. Nichtnumerische Datenstrukturen

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:

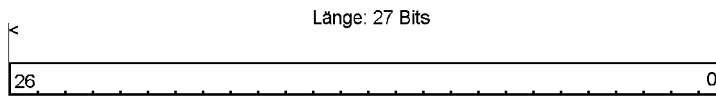
- eine Adresse,
- eine Längenangabe für die gesamte Kette. Möglichkeiten: Anzahl der String--Elemente, Anzahl der Bytes, Adresse des letzten String-Elementes bzw. Bytes, besondere Endekennzeichnung. Üblicherweise wird die Länge in Bytes angegeben.
- eine Längenangabe für das einzelne String-Element (8,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

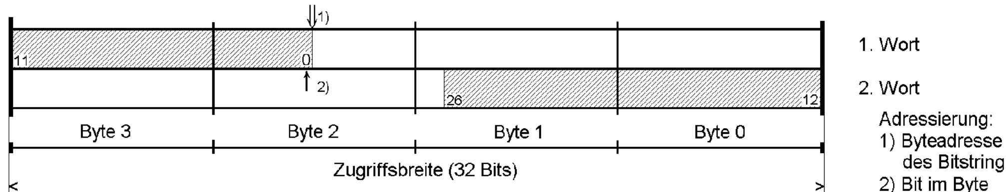
1. in den Architekturen S/370.../390 sind drei Arten von Bytestrings vorgesehen, die sich in ihrer maximalen Länge unterscheiden (1...16 Bytes, 1...256 Bytes, gesamter Speicheradreßraum). Befehle, die für die ersten beiden Arten gelten, enthalten Längen- und Adressierungsangaben als Direktwerte. Längenangaben 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 3.1) 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.

Beispiel eines Bitstring



Unterbringung im Speicher
(bei 4 Byte Zugriffsbreite; Beispiel für Überlappung)



Bitadresse (35 Bits lang) für beliebiges Bit in einem 4 GBytes-Adreßraum

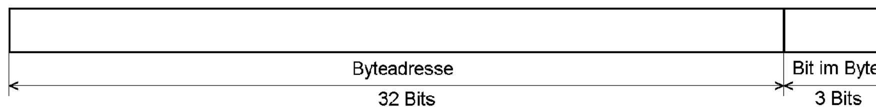


Abbildung 3.1 Bitkette (Bitstring)

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 3.1), 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.

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.

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 3.2).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00:		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	♪	♫	☼
10:	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
20:		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30:	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40:	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50:	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60:	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70:	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂
80:	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
90:	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
A0:	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
B0:	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C0:	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D0:	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E0:	α	ß	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	φ	ε	∩
F0:	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■	

Abbildung 3.2 Der erweiterte IBM-Zeichensatz (Codetabelle). Eingerahmt: der ursprüngliche Zeichenvorrat nach ASCII

Erklärung:

Die Spalte entspricht dem niederwertigen, die Zeile dem höherwertigen Halbbyte. Beispiel: Zeichen “d”: Zeile = 60H + Spalte = 4H = 64H (Übungsbeispiele im Anhang). 20H ist der Code des Leerzeichens (engl. Space; sprich: S-pehs). 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 (ein Fachbegriff aus dem Umfeld des DOS-Betriebssystems).

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 3.2): 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 3.1).
- 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 3.1 Unicode (Übersicht)

Zeichencodes im PC:

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

Anhang: Rechnen und Üben mit Hexadezimalzahlen

1. Rechenhilfsmittel

Der Rechner im Zubehör von Windows

Er genügt in den weitaus meisten Fällen. Wichtig: auf die “wissenschaftliche” Ansicht umstellen (Abbildungen A1 bis A3).

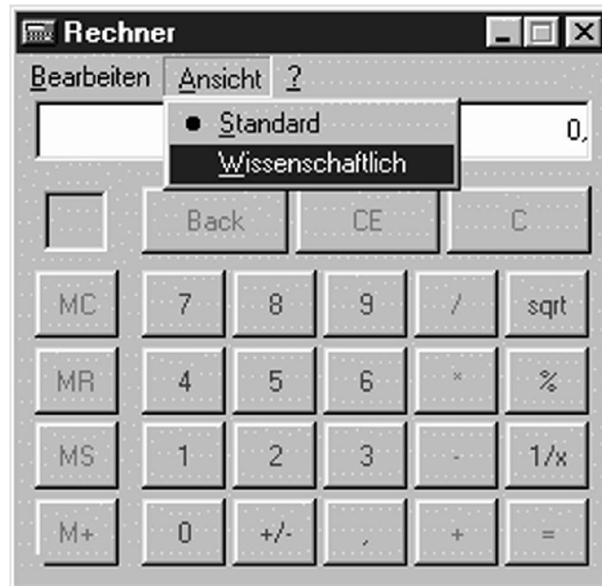


Abbildung A1 Der Rechner wird auf die “wissenschaftliche” Ansicht umgestellt

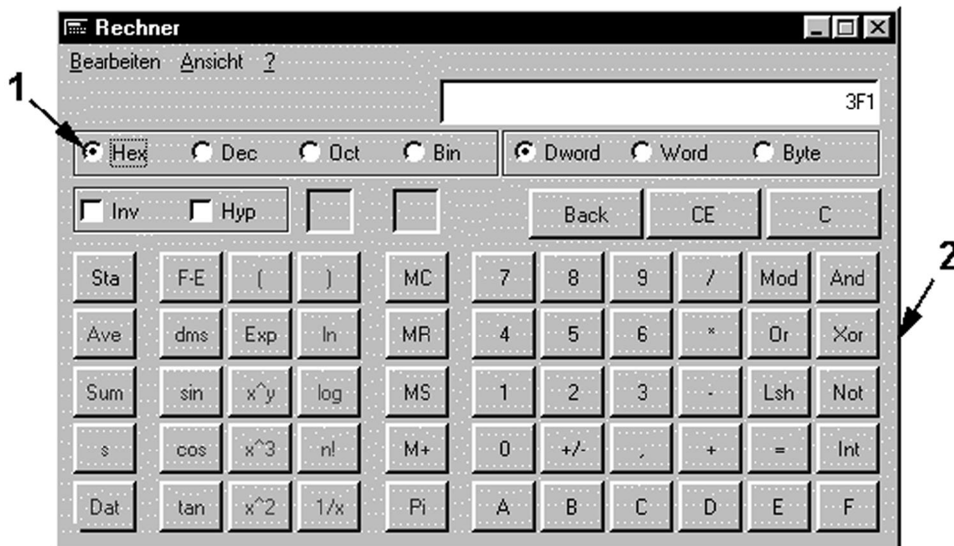


Abbildung A2 Eingabe einer Hexadezimalzahl

Erklärung zu Abbildung A2:
1 - Auswahl der Hexadezimaldarstellung (zur weiteren Wahl stehen noch die dezimale, die oktale und die binäre Darstellung); 2 - über diese Tasten können Operanden logischen Verknüpfungen unterzogen werden. Genauere Beschreibung: nicht erforderlich, da zu jeder Taste Direkthilfe geboten wird (mit Mauszeiger anfahren und rechte Maustaste betäti- gen).

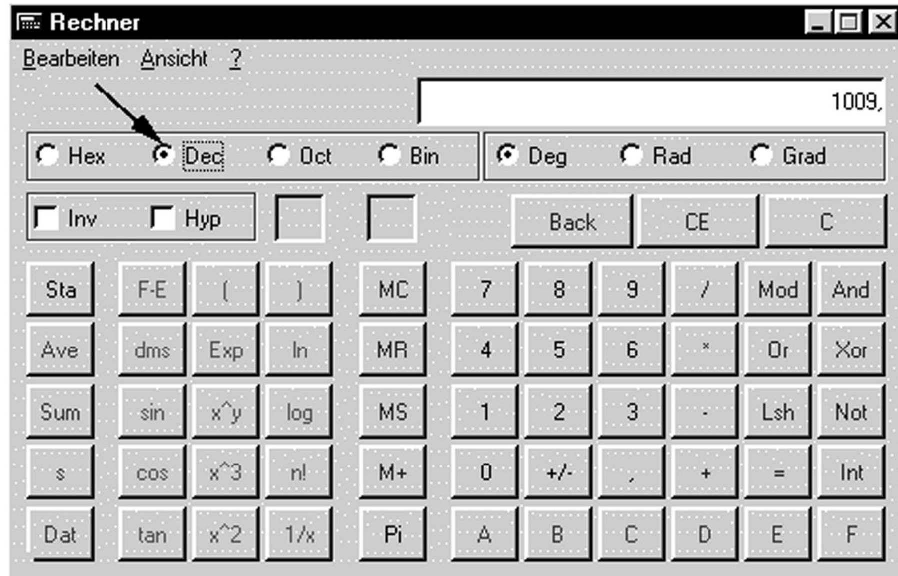


Abbildung A3 Durch Wahl der Dezimaldarstellung (Pfeil) wird die Zahl in's Dezimale gewandelt

Elementares Hexadezimalrechnen mittels BASIC
BASIC erlaubt es, einfache Programme in kurzer Zeit zusammenzuschreiben.

Eingabe hexadezimaler Konstanten: in Form **&Hxy**. Beispiel: &H2F.

Wandeln von (einzugebenden) Hex-Konstanten in die dezimale Darstellung: **PRINT &Hxy**.

> Beispiel: der Wert 14A0H ist in eine Dezimalzahl zu wandeln. Dazu ist einzugeben: PRINT &H14A0. Ergebnis: 5280.

Muster einer Rechenanweisung mit hexadezimaler Ausgabe: **PRINT HEX$** (Zahlenwert).

Der Zahlenwert kann Ergebnis einer Berechnung sein, die unmittelbar in der PRINT-Anweisung angegeben werden kann, z. B. PRINT HEX$ (a **op** b). Hierin sind a und b beliebige Zahlenwerte.

Operatoren: + - *, /, ^, NOT, AND, OR, XOR, EQV.

> NOT ist ein einstelliger Operator: PRINT HEX$ (NOT a).

Beispiel: wir wollen das Resultat der Multiplikation 1FH · 2BH wissen. Es ist einzugeben:

PRINT HEX$(&H1F * &H2B).

Ergebnis: 535H. (Das abschließende H wird nicht mit ausgegeben.)

2. Rechenübungen

2.1. Stellen Sie die folgenden Bitfolgen hexadezimal dar:

a) 1101 0110B

b) 01 1011B

c) 10110011 10001110B

2.2. Geben Sie folgende hexadezimal dargestellte Werte binär an:

a) 3F2H

b) 22CCH

c) 127H

2.3. Eine Einrichtung soll an einem Systembus unter der Adresse C028H erreichbar sein. Die Belegung der drei höchstwertigen Adreßbits ist an einem Drehschalter einzustellen, dessen 8 Stellungen mit 0...7 beschriftet sind. Welchen Wert stellen Sie ein?

2.4. Geben Sie die IP-Adresse 164.244.50.111 in hexadezimaler und in binärer Darstellung an.

2.5. Aus der meßtechnischen Analyse eines Datenstroms erkennen Sie folgendes Bitmuster, das offensichtlich als IP-Adresse dient: 1101 0110 1110 0011 0100 0011 1110 1000B. Geben Sie die IP-Adresse in Dotted Decimal Notation an.

2.6. Geben Sie die in ASCII vorliegende Zeichenfolge *Mausi* in Hexadezimaldarstellung an.

2.7. Welcher Zeichenfolge (in ASCII) entspricht die Bitkette 50 43 49 2D 42 75 73H?

Lösungen

Lösung zu 2.1.
Sie müssen von rechts außen an die Binärangabe in Vierergruppen unterteilen. Freie höherwertige Bitpositionen sind mit Nullen aufzufüllen. Entsprechend ergibt sich:

a) 1101 0110B = D6H

b) 01 1011B = 1BH

c) 10110011 10001110B = B38EH

Lösung zu 2.2.
Sie müssen die Hexadezimalstellen einzeln wandeln. Wenn nicht von vornherein bekannt ist, daß höchstwertige Bits weggelassen werden können (z. B. bei der vierstelligen Hex-Angabe eines 14-Bit-Wortes), so sollten Sie die führenden Nullen mit angeben. Es ergibt sich:

a) 3F2H = 0011 1111 0010B

b) 22CCH = 0010 0010 1100 1100B

c) 127H = 0001 0010 0111B

Lösung zu 2.3.
Es interessiert nur die höchstwertige Hexadezimalstelle CH = 1100B. Hiervon brauchen wir die drei höchstwertigen Bits 110B = 6. Also ist Schalterstellung 6 zu wählen.

Lösung zu 2.4.
164.244.50.111 = A4 F4 32 6FH = 1010 0100 1111 0100 0011 0010 0110 1111B.

Lösung zu 2.5.
1101 0110 1110 0011 0100 0011 1110 1000B = 161.227.67.232

Rechenhilfe zu 2.4 und 2.5: die Werte einzeln in den Windows-Taschenrechner eintippen und entsprechend wandeln. *Achtung:* binäre Ergebnisse enden mit der höchstwertigen Eins und sind deshalb ggf. auf die jeweilige Stellenzahl (z. B. 8 Stellen) zu ergänzen (links mit Nullen auffüllen).

Lösung zu 2.6.
Mausi ≙ 4D 61 75 73 69H.

Lösung zu 2.7.
50 43 49 2D 42 75 73H = *PCI-Bus*.

Hinweis zu 2.6 und 2.7.: Wechselseitige Wandlung z. B. mit Hilfe von Abbildung 3.2. Beispiel: das Zeichen *M* finden wir im Schnittpunkt von Zeile 40 und Spalte D. Also: M ≙ 4DH. Sinngemäß gehen wir z. B. mit der Hexadezimalzahl 73H in Zeile 70 und Spalte 3 und finden im Kreuzungspunkt das Zeichen *s*.