Binärzahlen – eine Einführung

1. 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 ⋅ 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* des Dezimalsystems (Abb. 1.1).

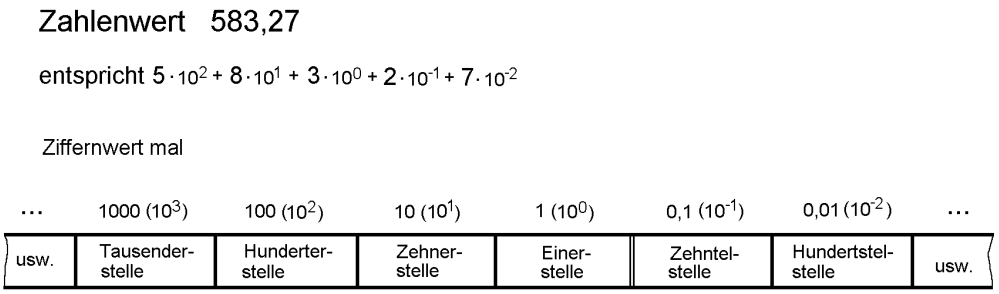


Abb. 1.1 Zahlendarstellung im Dezimalsystem

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 (Abb. 1.2).

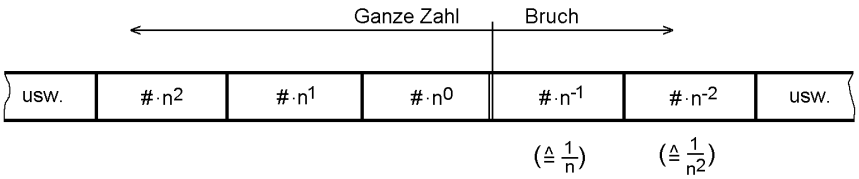


Abb. 1.2 Zahlendarstellung in einem beliebigen Stellenwertsystem zur Basis n

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; die 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 (Abb. 1.3). Der Vorteil: Zahlenwerte lassen sich in einer solchen Darstellung mit technischen Mitteln besonders einfach übertragen, speichern und verarbeiten; die Informationswandlungen lassen sich mit den Mitteln der Boleschen Algebra beschreiben (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 = $2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$

Abb. 1.3 Zahlendarstellung im Binärsystem

Werte von Zweierpotenzen
Tabelle 1.1 enthält die Werte der ersten 32 Zweierpotenzen mit positivem und negativem Exponenten.

Exponent n	**2^n**	**2^{-n}**
1	2	.5
2	4	.25
3	8	.125
4	16	.0625
5	32	.03125
6	64	.015625
7	128	.0078125
8	256	.00390625
9	512	.001953125
10	1024	.0009765625
11	2048	.00048828125
12	4096	.000244140625
13	8192	.0001220703125
14	16384	.00006103515625
15	32768	.000030517578125
16	65536	1.52587890625E-5
17	131072	7.62939453125E-6
18	262144	3.814697265625E-6
19	524288	1.9073486328125E-6
20	1048576	9.5367431640625E-7
21	2097152	4.76837158203125E-7
22	4194304	2.38418579101563E-7
23	8388608	1.19209289550781E-7
24	16777216	5.96046447753906E-8
25	33554432	2.98023223876953E-8
26	67108864	1.49011611938477E-8
27	134217728	7.45058059692383E-9
28	268435456	3.72529029846191E-9
29	536870912	1.86264514923096E-9
30	1073741824	9.31322574615479E-10
31	2147483648	4.65661287307739E-10
32	4294967296	2.3283064365387E-10

Tabelle 1.1 Zweierpotenzen

Hinweis:
Die Werte der Zweierpotenzen mit negativem Exponenten sind als Dezimalbrüche angegeben. "E-x" steht dabei für 10^{-x}. Beispiel: E-9 entspricht 10^{-9}.

Wichtige Näherungswerte:

$2^{10} = 1024 \ . \ 10^3$ (tausend)

$2^{20} = 1\ 048\ 576 \ . \ 10^6$ (eine Million)

$2^{30} = 1\ 073\ 741\ 824 \ . \ 10^9$ (eine Milliarde)

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 Abb. 1.3 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 diese Darstellung nicht mit dem logischen Pegel L (für LOW).

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 hier 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 (Abb. 1.4). 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.

Beispiel:

- 16-Bit-Wort: 1001000011001011B.
- binäre Darstellung mit Abständen: 1 001 000 011 001 011 bzw. 1001 0000 1100 1011.
- oktal: 110313.
- hexadezimal mit 90CB.

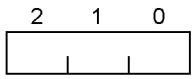
Nutzung und Kennzeichnung
Oktalzahlen stammen aus der Zeit, als Zeichen üblicherweise noch mit 6 Bits codiert wurden. Die Maschinen hatten Verarbeitungsbreiten von beispielsweise 12, 24, 36 oder 48 Bits. Um Binärwerte bequem notieren, aussprechen sowie an Bedientafeln (Abb. 1.5) anzeigen und eingeben 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).

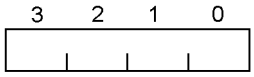
Mit der Einführung des 8-Bit-Bytes (IBM System /360) setzte sich die Hexadezimaldarstellung nach und nach durch.

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

Oktal-Notation: 3 Bits werden zusammengefaßt.



Hexadezimal-Notation: 4 Bits werden zusammengefaßt (Halbbyte, Tetrade, Nibble).



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

Abb. 1.4 Oktal- und Hexadezimalnotation

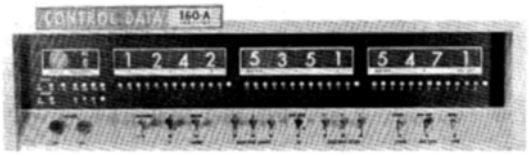


Abb. 1.5 Bedientafel eines Computers (CDC 160; ca. 1963) mit Oktal-Anzeigen (Control Data Corporation)

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.

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 ins 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 werden gelegentlich dezimal angegeben. So entspricht der ASCII-Code 32 = 20H dem Leerzeichen, der Code 71 = 47H dem Zeichen “G” usw.

Natürliche Binärzahlen
Natürliche Zahlen sind vorzeichenlos. Der niedrigste Wert ist Null. Der gesamte Wertebereich einer natürlichen Binärzahl x aus n Bits (Abb. 1.6) ist gegeben durch

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

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

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

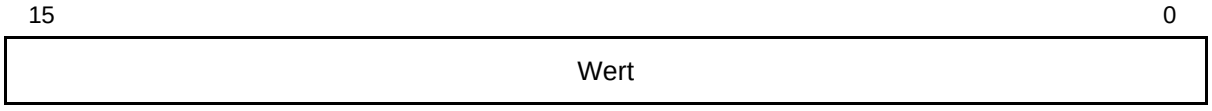


Abb. 1.6 Beispiel einer natürlichen (vorzeichenlosen) Binärzahl (16 Bits)

Elementare Formate
In Tabelle 1.2 sind Formate natürlicher Binärzahlen angegeben, die heutzutage gebräuchlich sind (die Zahlen belegen 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 ≈ 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 1.2 Natürliche Binärzahlen als elementare Datentypen

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 (Abb. 1.7).

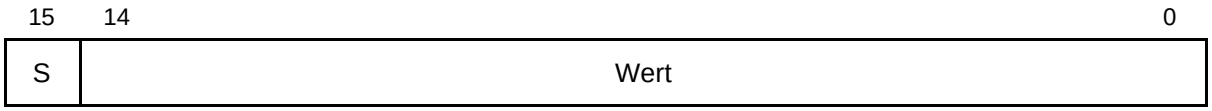


Abb. 1.7 Beispiel einer ganzen Binärzahl (16 Bits). S = Vorzeichen (Sign)

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 genauso 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 entspricht 2^n - x).

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

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

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 (+ 4) liegt; sie ist um so kleiner, je näher sie an minus Unendlich (- 4) 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). - 3 ist größer als - 8 (- 3 > - 8), + 2 ist größer als - 2 (+ 2 > - 2). - 4 ist kleiner als + 4. Jede negative Zahl ist kleiner als Null.

Elementare Formate
In Tabelle 1.3 sind Formate ganzer Binärzahlen angegeben, die heutzutage gebräuchlich sind (die Zahlen belegen 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} =	. - 9,2 @ 10^{18}	$2^{63}-1$ =	. 9,2 @ 10^{18}

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

Tabelle 1.3 Ganze Binärzahlen (Integer-Zahlen) als elementare Datentypen

2. Mit Binärzahlen rechnen

2.1 Wandeln (dezimal – binär, binär – dezimal)

Es gibt verschiedene Wandlungsverfahren. Ein wichtiger Unterschied besteht darin, womit diese Verfahren durchgeführt werden sollen:

- von Hand,
- in der Maschine (= hardware- und/oder programmseitig).

2.1.1 Wandeln ins Binäre (1) – von Hand

Sonderwerte:

- 0 => 0.
- 1 => 1.
- 2^n => 1, gefolgt von n Nullen.

Wandlung nach dem Subtraktionsverfahren
Sei n die zu wandelnde Dezimalzahl. Die Binärzahl wird von links nach rechts aufgebaut (Tabelle 2.1).

1. Die kleinste Zweierpotenz suchen, die größer als n ist.
2. Mit der nächst-kleineren Zweierpotenz beginnen. Die erste Eins hinschreiben.
3. Differenz bilden n = n - Zweierpotenz ($n = n - 2^p$).
4. Wenn p = 0, dann Ende. Sonst die nächst-kleinere Zweierpotenz nehmen: p = p - 1.
5. Wenn n >= Zweierpotenz, dann rechts eine Eins anfügen und weiter mit Schritt 3, sonst rechts eine Null anfügen weiter mit Schritt 4.

Wandlung nach dem Divisionsverfahren
Sei n die zu wandelnde Dezimalzahl. Die Binärzahl wird von rechts nach links aufgebaut (Tabelle 2.2).

1. n = n : 2. Der Rest kann nur 0 oder 1 sein (gerade oder ungerade Zahl). Rest wird links angefügt
2. Wenn Quotient = 0, dann Ende. Sonst weiter mit Schritt 1.

Schritt	n	p	2^p	Binärwert	Anmerkungen
1	1997	11	2048		2048 > 1997
2	1997	10	1024	**1**	mit der nächst-kleineren Zweierpotenz geht‘s los
3	973				1997 - 1024 = 973
4		9	512		die nächste Zweierpotenz
5				1**1**	973 ist größer als 512, also eine Eins
3	461				973 - 512 = 461
4		8	256		die nächste Zweierpotenz
5				11**1**	461 ist größer als 256, also eine Eins
3	205				461 - 256 = 205
4		7	128		die nächste Zweierpotenz
5				111**1**	205 ist größer als 128, also eine Eins
3	77				205 - 128 = 77
4		6	64		die nächste Zweierpotenz
5				1111 **1**	77 ist größer als 64, also eine Eins
3	13				77 - 64 = 13
4		5	32		die nächste Zweierpotenz
5				1111 1**0**	13 ist kleiner als 32, also eine Null
4		4	16		die nächste Zweierpotenz
5				1111 10**0**	13 ist kleiner als16, also eine Null
4		3	8		die nächste Zweierpotenz
5				1111 100**1**	13 ist größer als 8, also eine Eins
3	5				13 - 8 = 5
4		2	4		die nächste Zweierpotenz
5				1111 1001 **1**	5 ist größer als 4, also eine Eins
3	1				5 - 4 = 1
4		1	2		die nächste Zweierpotenz
5				1111 1001 1**0**	1 ist kleiner als 2, also eine Null
4		0	1		die nächste Zweierpotenz
5				1111 1001 10**1**	1 ist gleich 1, also eine Eins
3	0				0 - 0 = 0
4				Ende: 111 1100 1101	p 0 0, also fertich ...

Tabelle 2.1 Wandlung ins Binäre mittels Subtraktionsverfahren

n	n:2	Rest	Binärwert
1997	998	1	**1**
998	499	0	**0**1
499	249	1	**1**01
249	124	1	**1**101
124	62	0	**0** 1101
62	31	0	**0**0 1101
31	15	1	**1**00 1101
15	7	1	**1**100 1101
7	3	1	**1** 1100 1101
3	1	1	**1**1 1100 1101
1	0	1	**1**11 1100 1101

Tabelle 2.2 Wandlung ins Binäre mittels Divisionsverfahren

2.1.2 Wandeln ins Binäre (2) – maschinell

Die Dezimalzahlen sind in stellenweise binär codierter Form gegeben (BCD-Zahlen).

1. Verfahren (einfach, aber langsam)
Dezimalzahl stellenweise bis auf Null herunterzählen, dabei Binärzahl um den jeweiligen Stellenwert erhöhen.

Zum Anfang: Binärzahl = niedrigste Dezimalstelle.

Zehnerstelle auf Null herunterzählen, dabei Binärzahl jeweils um 10 erhöhen.

Hunderterstelle auf Null herunterzählen, dabei Binärzahl jeweils um 100 erhöhen usw.

2. Verfahren (nach Konrad Zuse (clever und schnell; nutzt Leistungsvermögen der Maschine voll aus)):
Gegeben sei z. B. eine 5stellige BCD-Zahl mit den Dezimalstellen a b c d e.

Die Binärzahl ergibt sich zu:
e + 10d + 100c + 1000d + 10000a (Rechnen im Binären).

Der Rechengang läßt sich umformen zu

e + 10 (d + 10 (c + 10 (b + 10 a)))

Start: bin = a
1. Teilrechnung: bin = 10 bin + b
2. Teilrechnung: bin = 10 bin + c
3. Teilrechnung: bin = 10 bin + b
Schlußrechnung: bin = 10 bin + a

Multiplikation mit 10 = Linksverschiebung um 3 Bits (• 8) + Linksverschiebung um 1 Bit (• 2).

2.1.3 Wandeln ins Dezimale (1) – von Hand

Die Dezimalzahl ergibt sich als Summe von Zweierpotenzen. Die Binärzahl wird von rechts nach links abgefragt (Tabelle 2.3).

1. Dezimalzahl d = 0, Stellennummer n = 1 (erste Stelle (rechts außen)).
2. Enthält die betreffende Stelle eine Eins, so ist $d = d + 2^{n-1}$ (hinzuaddieren der Zweierpotenz).
3. Wurde die höchstwertige Stelle abgefragt, dann Ende. Sonst weiter mit Schritt 2.

$$\text{Dezimalzahl} = \sum_{n=1}^{s} (w(n) \cdot 2^{n-1})$$; s = Stellenzahl, w(n) = Binärwert (0 oder 1) an Stelle n.

Binärzahl	Wertigkeit der Stelle	Dezimalzahl
111 1100 1101	1	1
111 1100 110	2	
111 1100 11	4	4
111 1100 1	8	8
111 1100	16	
111 110	32	
111 11	64	64
111 1	128	128
111	256	256
11	512	512
1	1024	1024
		Endwert: 1997

Tabelle 2.3 Wandlung ins Dezimale

2.1.4 Wandeln ins Dezimale (2) – maschinell

1. Verfahren:
Zweierpotenzen als Dezimalwerte (BCD) zueinander addieren (Dezimaladdition).

Zweierpotenzen können fest gespeichert oder durch Verdoppeln (Dezimaladdition) in jedem Schritt immer wieder neu berechnet werden.

2. Verfahren
Fortlaufende Division durch 10.

bin : 10 => bin; Rest = Einerstelle

bin : 10 => bin; Rest = Zehnerstelle

bin : 10 => bin; Rest = Hunderterstelle

usw. (bis Quotient = 0).

Abwandlung: Statt Division durch 10 Multiplikation mit 0,1 (Gleitkommarechnung):

$$0{,}1D = 0{,}000110011001100...B = \frac{1}{2^4} + \frac{1}{2^8} + \frac{1}{2^4} + \frac{1}{2^{64}} + \frac{1}{2^{128}} + \frac{1}{2^{256}} + \frac{1}{2^{2048}} + \frac{1}{2^{4096}} ...$$

Rechengang nach Konrad Zuse (Z1, Z3):

$0{,}1D = 2^{-4} \bullet 1.1001100110011001B$

Das 0,1fache einer beliebigen Binärzahl X:

$0{,}1\ X = 2^{-4} \bullet X \bullet 1.1001100110011001B$

Wir rechnen zunächst $X \bullet 1{,}100110011001100...$ B und multiplizieren dann mit 2^{-4} (= Division durch 16 = Rechtsverschiebung um 4 Bits).

$X_1 = X + .1\ X = X + {}^{X}/_{2}$ $(X + X \bullet 2^{-1})$
$X_2 = X_1 + .0001\ X_1 = X_1 + {}^{X_1}/_{16} = X \bullet 1.10011$ $(X_1 + X_1 \bullet 2^{-4})$
$X_3 = X_2 + .000\ 000\ 01\ X_2 = X_2 + {}^{X_2}/_{256} = X \bullet 1.1001100110011$ $(X_2 + X_2 \bullet 2^{-8})$
$X_4 = X_3 + .000\ 000\ 000\ 000\ 0001\ X_3 = {}^{X_3}/_{65536} = X \bullet 1.10011001100110011001100110011$ $(X_3 + X_3 \bullet 2^{-16})$

Abschließend wird X^4 mit 2^{-4} multipliziert.

3. Verfahren
Fortlaufende Subtraktion von 10 statt Divison durch 10. Praktikabel für Zahlenwerte < 100.

4. Verfahren
Division durch absteigende Zehnerpotenzen (die als Binärzahlen dargestellt werden). Der Quotient ergibt jeweils eine Dezimalstelle, der Rest wird durch die nächst-kleinere Zehnerpotenz dividiert.

Beispiel: 16-Bit-Binärzahl.

Binärzahl : 10 000 = 5. (höchstwertige) Dezimalstelle.
Rest : 1000 = 4. Dezimalstelle.
Rest : 100 = 3. Dezimalstelle.
Rest : 10 = 2. Dezimalstelle
Rest = 1. (niedrigstwertigste) Dezimalstelle.

2.1 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 (Abb. 2.1 und 2.2).

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.

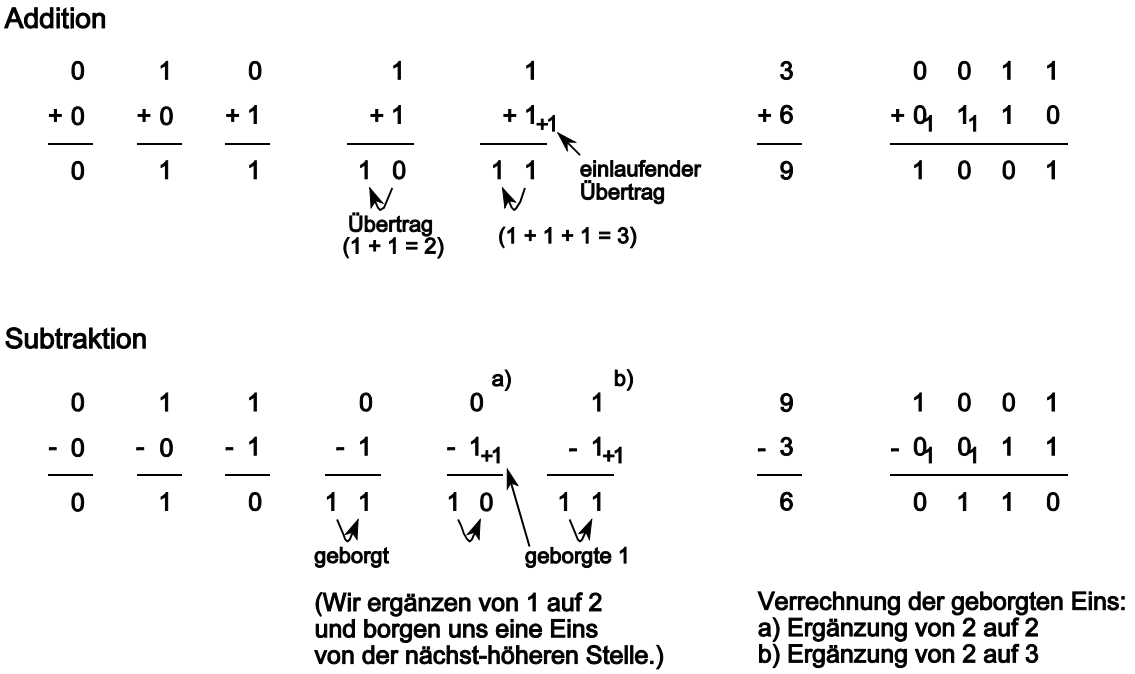


Abb. 2.1 Addieren und Subtrahieren von Binärzahlen

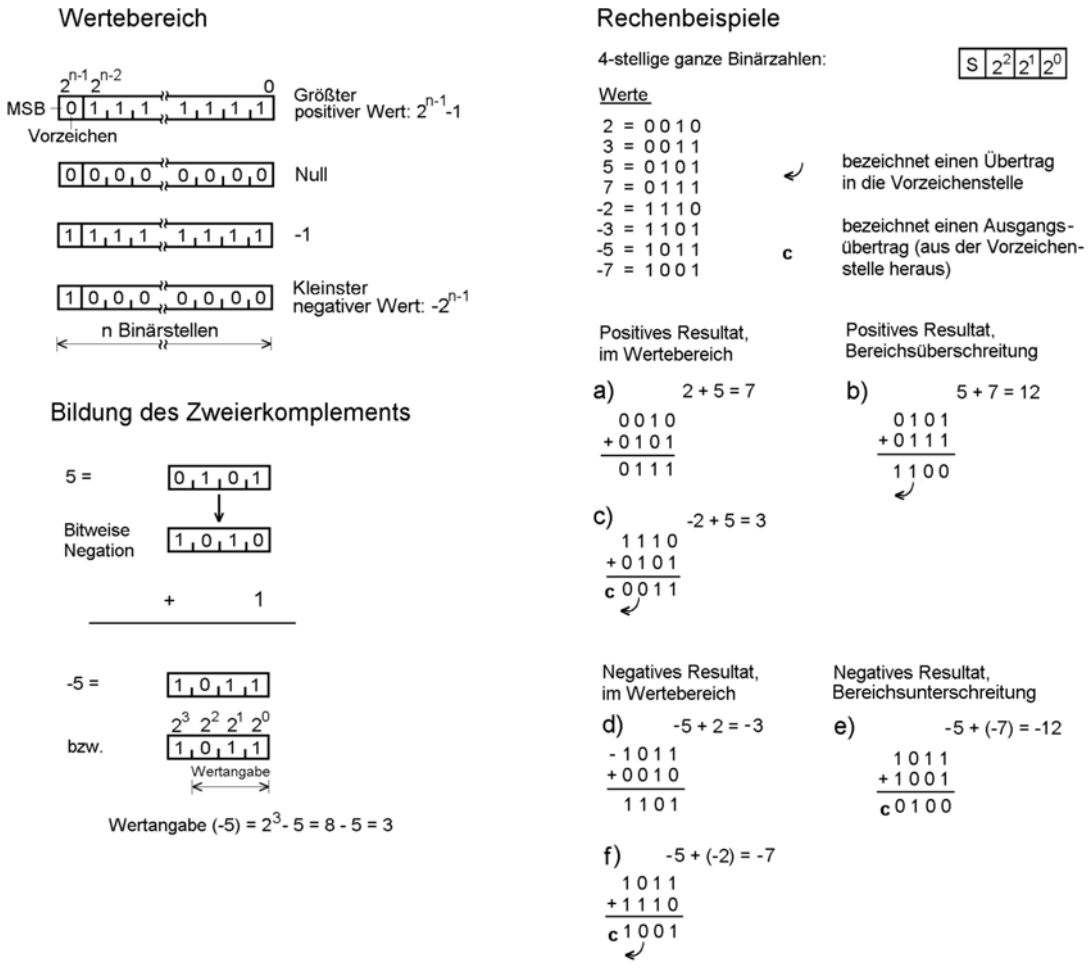


Abb. 2.2 Elementares Rechnen mit ganzen Binärzahlen

Über- und Unterschreiten des Wertebereichs

1. beim Rechnen mit natürlichen Binärzahlen
Ob der der Wertebereich über- oder unterschritten wird, ist anhand des Übertrags in der höchstwertigen Stelle erkennbar (*Ausgangsübertrag* (Carry Out)), der das Rechenergebnis gleichsam um eine Stelle verlängert (Abb. 2.3). 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 (Abb. 2.3a). Ist das eigentliche (mit dem Ausgangsübertrag um eine Stelle längere) Rechenergebnis gleich r (r # 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: $r_n = 2^n - r$ (Abb. 2.3b).

Dieses Umschlagen des Ergebnisses wird auch als "Wrap Around" bezeichnet. Siehe auch den Abschnitt zur Sättigungsarithmetik.

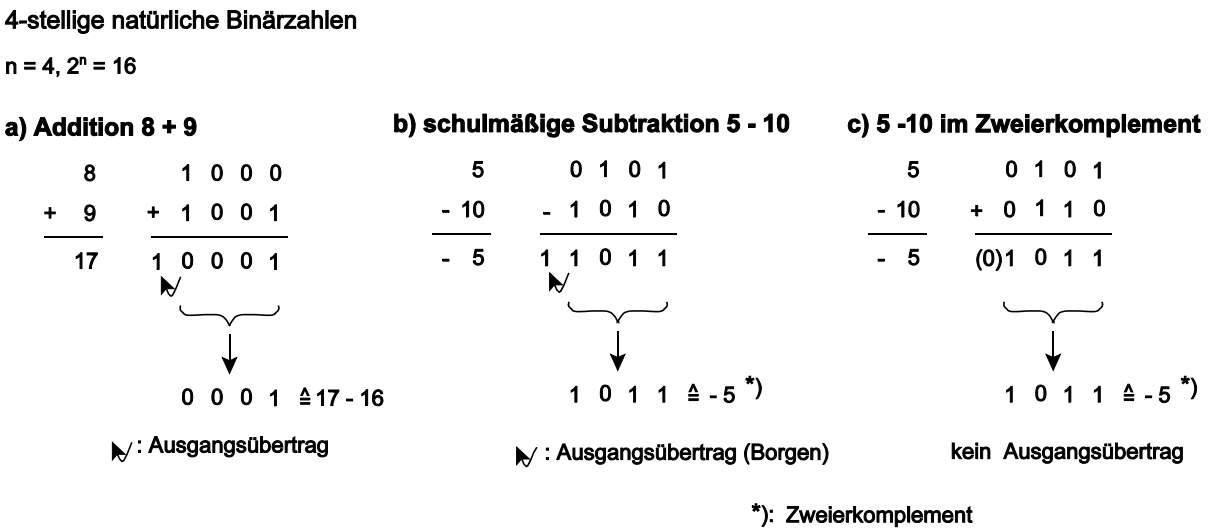


Abb. 2.3 Über- und Unterschreiten des Wertebereichs beim Rechnen mit natürlichen Binärzahlen

Hinweis:
In Abb. 2.1 haben wir schulmäßig subtrahiert. 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).

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 (vgl. Abb. 2.2b). Wird die kleinste negative Zahl unterschritten, so entsteht nur ein Ausgangsübertrag, aber kein Übertrag in die Vorzeichenstelle (vgl. Abb. 2.2e). Allgemein wird das Verlassen des Wertebereichs ganzer Zahlen als *Überlauf* (Overflow) bezeichnet.

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

Beim Addieren:
Ein Überlauf kann nur entstehen, wenn beide Operanden gleiche Vorzeichen haben. Der Überlauf ist dann gegeben, wenn das Ergebnis ein anderes Vorzeichen hat als die Operanden. Operanden mit unterschiedlichen Vorzeichen erzeugen keinen Überlauf.

Beim Subtrahieren:
Ein Überlauf kann nur entstehen, wenn beide Operanden unterschiedliche Vorzeichen haben. Operanden mit gleichen Vorzeichen erzeugen keinen Überlauf.

Bestimmung der Overflow-Bedingung:
Overflow = Ausgangsübertrag ... Übertrag in die Vorzeichenstelle = Ausgangsübertrag $\oplus$ Übertrag in die Vorzeichenstelle = $C_o \oplus C_n$.

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

Wenn bei bekannten Operandenbits A, B ein bestimmtes Ergebnisbit S entsteht, muß der Eingangsübertrag in diese Stelle einen bestimmten Wert haben. In Tabelle 2.5 sind jene Wertekombinationen angeführt, die sich nur dann ergeben, wenn der Eingangsübertrag $C_n = 1$ ist. Ersichtlicherweise ist dies dann der Fall, wenn die Anzahl der Einsen über A, B und 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 $A \oplus B \oplus S \oplus C_o$.

Bestimmung der Overflow-Bedingung aus den höchstwertigen Operandenbits A, B und dem zugehörigen Ergebnisbit S. Ein Überlauf ensteht, wenn:

- A und B beide = 1 sind und S = 0 ist, d. h., wenn in dieser Stelle ein Übertrag entsteht, aber kein Übertrag einläuft,
- A, B beide = 0 sind und S = 1, d. h., wenn in diese Stelle ein Übertrag einläuft, aber nicht weitergegeben wird.

$$\text{Overflow} = AB\overline{S} \vee \overline{AB}S$$

Rekonstruktion der Überlauf-Ursache
In Prozessoren haben wir typischerweise nur Flagbits für Overflow (OF) und Ausgangsübertrag (CF). Daraus kann die jeweilige Urache des Überlaufs wie folgt rekonstruiert werden:

- wenn CF = 1, muß CF den Überlauf verursacht haben (Bereichsunterschreitung),
- wenn CF = 0, muß der Übertrag in der Vorzeichenstelle den Überlauf verursacht haben (Bereichsüberschreitung).

Achtung: ggf. bei Subtraktion CF-Belegung invertieren (vgl. Tabelle 2.7).

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).

Beispiele:
Die Ergebnisse in Abb. 2.2 sind auch korrekt, wenn man die 4stelligen 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 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 wirksam).

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 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.

Programmablauf:
Es wird abschnittsweise gemäß Verarbeitungsbreite (8 Bits, 16 Bits usw.) addiert oder subtrahiert, und zwar die niedrigstwertigen Stellen zuerst. Der erste Befehl ist jeweils eine gewöhnliche Addition oder Subtraktion (ohne Ausgangsübertrag). Die nachfolgenden Rechenbefehle beziehen dann den Ausgangsübertrag ein. Beispiel: In einem 8-Bit-Prozessor sind zwei 32-Bit-Operanden A, B zu verarbeiten. Abb. 2.4 veranschaulicht die Datenstrukturen, Tabelle 2.6 die Befehlsfolgen.

31 24	23 16	15 8	7 0
4. Byte	**3. Byte**	**2. Byte**	**1. Byte**

Abb. 2.4 Die Struktur einer 32-Bit-Binärzahl

Addition	Subtraktion
ADD A(7...0), B(7...0) => C(7...0) ADC A(15...8), B(15...8) => C(15...8) ADC A(23...16), B(23...16) => C(23...16) ADC A(31...24), B(31...24) => C(31...24)	SUB A(7...0), B(7...0) => C(7...0) SBC A(15...8), B(15...8) => C(15...8) SBC A(23...16), B(23...16) => C(23...16) SBC A(31...24), B(31...24) => C(31...24)

Tabelle 2.6 Addieren und Subtrahieren beliebig langer Binärzahlen (Rechenbeispiele)

2.2 Vergleichen zweier Binärzahlen

Wir vergleichen zwei 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 2.7). Sie können zwecks Programmverzweigung abgefragt werden (in JMP- bzw. BRANCH-Befehlen). Aus den Tabellen 2.8 und 2.9 sind die typischen Verzweigungsbedingungen ersichtlich. Tabelle 2.10 zeigt anhand von Beispielen, wie die Verzweigungsbedingungen beim Rechnen mit ganzen (vorzeichenbehafteten) Binärzahlen entstehen.

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: manche 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 # 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.

Typische Bezeichnung	**Benennung**	**Bedeutung**
ZF	Zero Flag	Ergebnis = 0
CF	Carry Flag	Addition: Ausgangsübertrag = 1, Subtraktion (s. den nachfolgenden Text): 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

Tabelle 2.7 Flagbits

Vergleichen natürlicher Binärzahlen. Rechengang: A - B			
Vergleichsaussage	Bedingung[1)]	Flagbits[2)]	typische Bezeichnung[2)]
A = B	Ergebnis = 0 (sowie Ausgangsübertrag)	ZF = 1	Equal
A ... B	Ergebnis ... 0	ZF = 0	Not Equal
Auslegung 1: CF = Ausgangsübertrag der Zweierkomplementrechnung			
A < B	Borgen = kein Ausgangsübertrag	CF = 0	Below
A > B	Ergebnis ... 0 und kein Borgen = Ausgangsübertrag	$\overline{ZF}$ @ CF ' 1 bzw. ZF w $\overline{CF}$ ' 0	Above
A # B	Ergebnis = 0 oder Borgen = kein Ausgangsübertrag (A nicht größer als B)	ZF w $\overline{CF}$ ' 1	Below or Equal
A $ B	kein Borgen = Ausgangsübertrag (A nicht kleiner als B)	CF = 1	Above or Equal

Vergleichen natürlicher Binärzahlen. Rechengang: A - B			
Auslegung 2: CF = invertierter Ausgangsübertrag der Zweierkomplementrechnung			
A < B	Borgen = Ausgangsübertrag (CF = 1)	CF = 1	Below
A > B	Ergebnis … 0 und kein Borgen = kein Ausgangsübertrag (CF = 0)	$\overline{ZF}$ @ $\overline{CF}$ ' 1 bzw. ZF w CF = 0	Above
A # B	Ergebnis = 0 oder Borgen = Ausgangsübertrag (CF = 1). A nicht größer als B	ZF w CF = 1	Below or Equal
A $ B	kein Borgen = kein Ausgangsübertrag (CF = 0). A nicht kleiner als B	CF = 0	Above or Equal

1): Zweierkomplement-Arithmetik; 2): in Befehlsbeschreibungen

Tabelle 2.8 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 kein Überlauf oder Ergebnis positiv und Überlauf	SF … OF SF $\oplus$ OF = 1*)	Less
A > B	Ergebnis … 0 und Ergebnis negativ und Überlauf oder Ergebnis positiv und kein Überlauf	ZF = 0 und SF = OF ZF w (SF $\oplus$ OF) = 0	Greater
A # B	Ergebnis = 0 oder Ergebnis negativ und kein Überlauf oder Ergebnis positiv und Überlauf (A nicht größer als B)	ZF oder SF … OF ZF w (SF $\oplus$ OF) = 1	Less or Equal
A $ B	Ergebnis negativ und Überlauf oder Ergebnis positiv und kein Überlauf (A nicht kleiner als B)	SF = OF SF $\oplus$ OF = 0	Greater or Equal

*): Atmel AVR: zur sog. Signed-Flag zusammegefaßt

Tabelle 2.9 Vergleichen ganzer (vorzeichenbehafteter) Binärzahlen

Rechenbeispiel	Rechengang	Vergleichsergebnis	OF	SF
5 - 3	0 1 0 1 1 1 0 1 1 0 0 1 0	A > B	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		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	A < B	0	1
-5 - 3	1 0 1 1 1 1 0 1 1 1 0 0 0		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		0	1
-5 - 4	1 0 1 1 1 1 0 0 1 0 1 1 1		1	0
5 - 5	0 1 0 1 1 0 1 1 1 0 0 0 0	A = B	0	0
-5 - (-5)	1 0 1 1 0 1 0 1 1 0 0 0 0		0	0

Tabelle 2.10 Vergleichen ganzer Binärzahlen: Rechenbeispiele

2.3 Multiplizieren und Dividieren

Im Binären wird genauso multipliziert und dividiert, wie wir es aus dem Schulunterricht vom Dezimalen her kennen (Abb. 2.5).

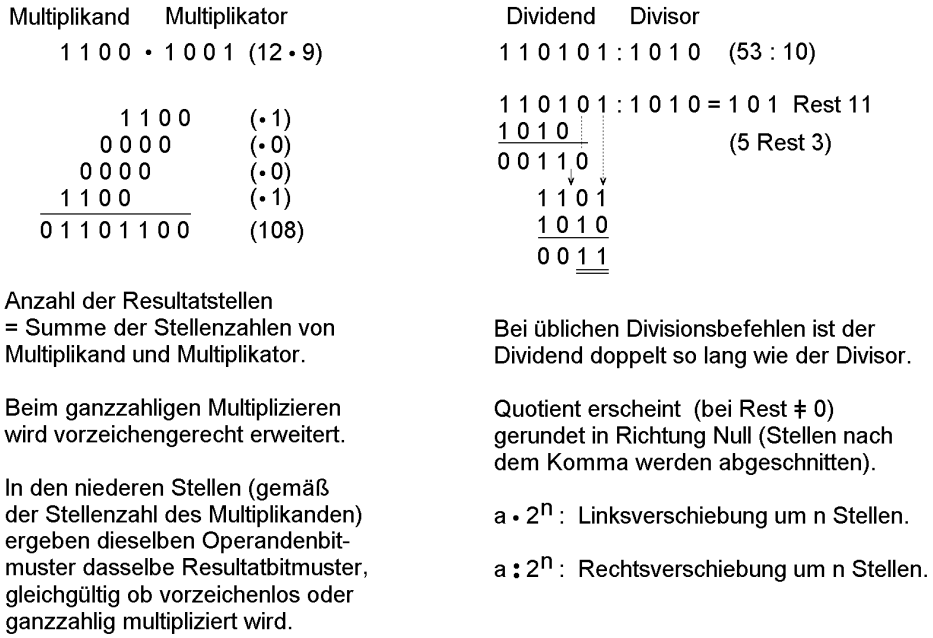


Abb. 2.5 Multiplikation und Division von Binärzahlen anhand von Beispielen

2.4 Rechnen mit kurzen Binärzahlen

Sind ganze Binärzahlen kürzer als die Verarbeitungsbreite, so werden sie *vorzeichengerecht* erweitert (Sign Extend). Dabei wird das Vorzeichen in *alle* auszufüllenden Stellen eingetragen (Abb. 2.6).

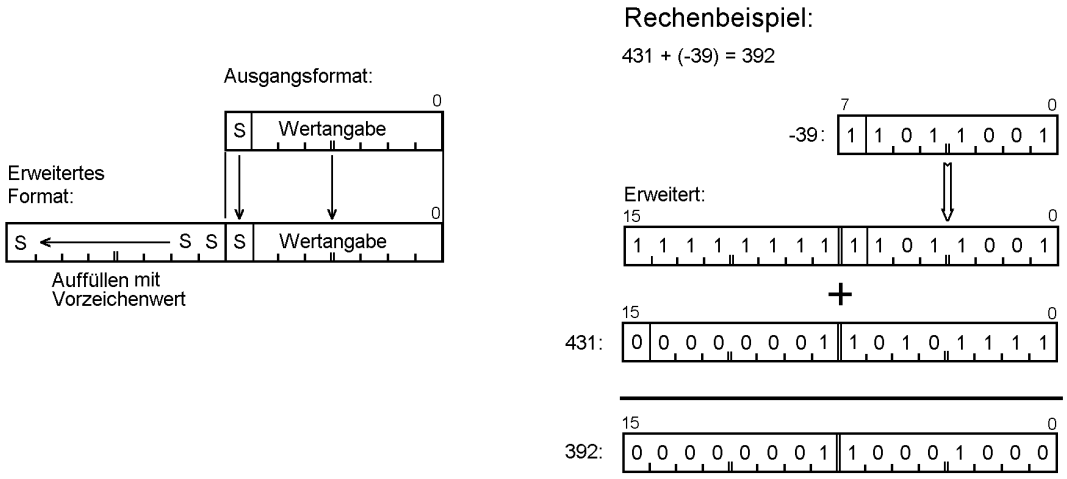


Abb. 2.6 Prinzip der Vorzeichenerweiterung ganzer Binärzahlen

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 (z. B. IA-32: “Laden mit Nullerweiterung” (MOVZX)).

3. Herkömmliche Arithmetik und Sättigungsarithmetik

Die herkömmliche (Zweierkomplement-) Arithmetik heißt im Englischen auch “Wrap-Around-Arithmetik” - ein wirklich bildhafter Begriff (Abb. 3.1).

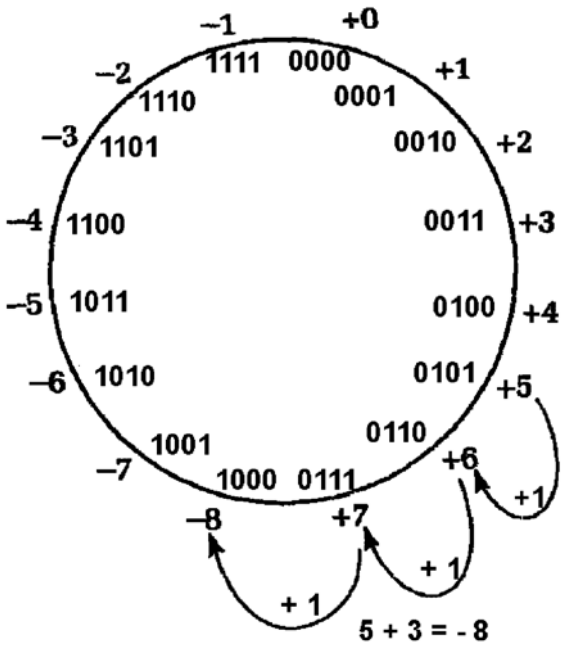


Abb. 3.1 Zweierkomplement-Arithmetik als Wrap-Around-Arithmetik

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 (von 0111B nach 1000B bzw. von +7 nach -8). Dann geht es rückwärts weiter bis zur -1 (1111B) und im nächsten Zählschritt wieder zur Null. 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. Abb. 3.1 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 3.1): wird der Wertebereich überschritten, so wird als Ergebnis der jeweilige Größtwert geliefert, wird der Wertebereich unterschritten, der jeweilige Kleinstwert.

Das ist vor allem beim Rechnen mit 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).

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 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 3.1 Herkömmliche und Sättigungsarithmetik

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 (Abb. 4.1).

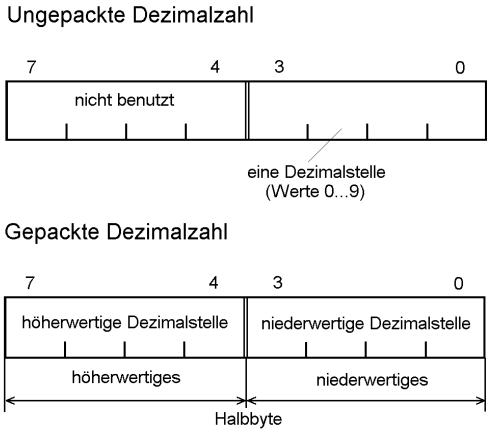


Abb. 4.1 Binär codierte Dezimalzahlen (BCD-Zahlen)

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). Abb. 4.2 zeigt Beispiele von BCD-Zahlen.

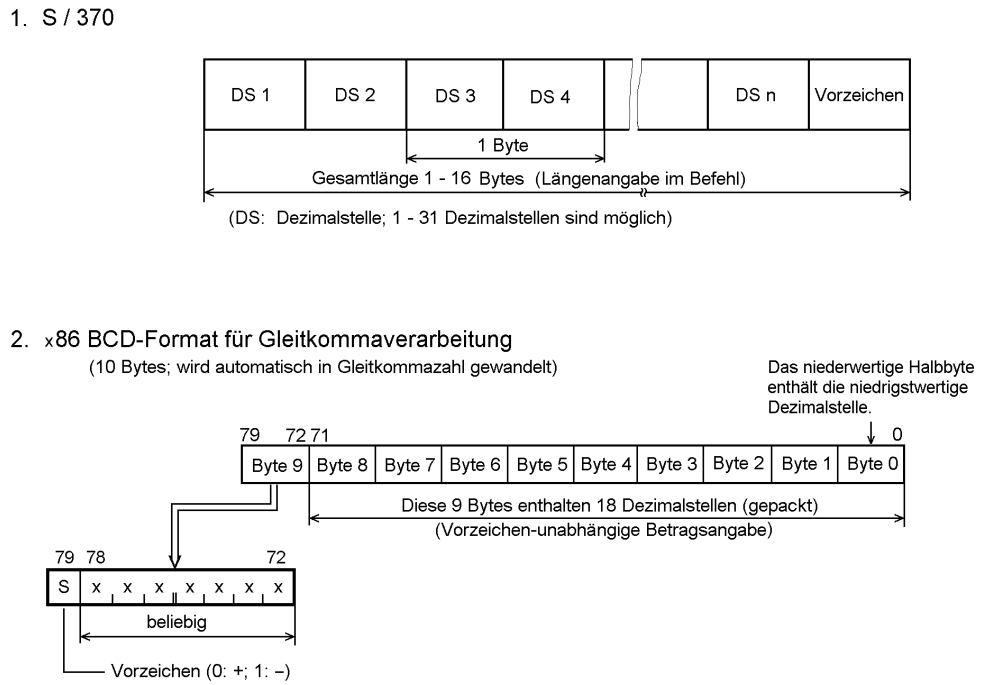


Abb. 4.2 Formate vorzeichenbehafteter BCD-Zahlen (Beispiele)

5. Elementare logische Operationen

Modifizieren, Testen,Vergleichen
In diesem Abschnitt wollen wir zeigen, wie man mit den elementaren logischen Verknüpfungen wichtige Anwendungsaufgaben lösen kann (Abb. 5.1).

Setzen von Bits
Um bestimmte Bits (in einem Register, Maschinenwort usw.) zu setzen, ist eine ODER-Verknüpfung (OR-Befehl) notwendig, wobei der zweite Operand an allen zu setzenden Stellen Einsen enthält und sonst Nullen.

Löschen von Bits
Um bestimmte Bits (in einem Register, Maschinenwort usw.) zu löschen, ist eine UND-Verknüpfung (AND-Befehl) notwendig, wobei der zweite Operand an allen zu löschenden Stellen Nullen enthält und sonst Einsen.

Wechseln von Bits
Um bestimmte Bits (in einem Register, Maschinenwort usw.) in ihrem Wert zu ändern (von 0 nach 1 und umgekehrt), ist ein Antivalenzverknüpfung (XOR-Befehl) notwendig, wobei der zweite Operand an allen zu ändernden Stellen Einsen enthält und sonst Nullen.

Entnehmen von Bits
Um zusammenhängende Bitfelder oder bestimmte einzelne Bits aus einem Register, Maschinenwort usw. zu entnehmen (andere Redeweisen: ausblenden, maskieren), ist eine UND-Verknüpfung notwendig, wobei der zweite Operand an allen ausgewählten Stellen Einsen enthält und sonst Nullen. (Dieser Operand heißt üblicherweise Maskenoperand.)

Einfügen von Bits
Um zusammenhängende Bitfelder oder bestimmte einzelne Bits in ein Register, Maschinenwort usw. einzufügen, ist auf das Register, Maschinenwort usw. zunächst eine UND-Verknüpfung anzuwenden, deren zweiter Operand an allen betreffenden Stellen Nullen enthält und sonst Einsen (Maskenoperand). Nachfolgend ist eine ODER-Verknüpfung anzuwenden, deren zweiter Operand an den betreffenden Stellen die einzufügenden Werte enthält und sonst Nullen.

Testen auf gelöschte bzw. gesetzte Bits
Um zu prüfen, ob bestimmte Bits (in einem Register, Maschinenwort usw.) alle gelöscht sind oder ob wenigstens eines dieser Bits gesetzt ist, braucht man eine UND-Verknüpfung, deren zweiter Operand an allen betreffenden Stellen Einsen enthält und sonst Nullen. Sind alle so geprüften Bits gelöscht, ist das Zero-Flagbit gesetzt. Ist wenigstens eines der geprüften Bits gesetzt, ist das Flagbit gelöscht. Diese Funktion steht beispielsweise in der x86-Architektur als TEST-Befehl zur Verfügung (das ist ein AND-Befehl, der kein Ergebnis zurückschreibt).

Alles löschen
Es gibt mehrere Möglichkeiten (z. B. Laden eines Direktwertes Null oder AND mit Null). Gelegentlich besonders effektiv: ein XOR mit sich selbst, z. B. <R1> := <R1> XOR <R1> (der Befehl hat keinen Direktwert und entspricht oft dem jeweils kürzesten Befehlsformat).

Vergleichen
Um zu prüfen, ob zwei Bitmuster einander gleich sind oder nicht, kann man eine Antivalenzverknüpfung (XOR-Befehl) verwenden. Bei Gleichheit ist das Zero-Flag gesetzt, bei Ungleichheit gelöscht. XOR-Befehle ohne Zurückschreiben des Ergebnisses gibt es in manchen Architekturen als "logische" Vergleichsbefehle (Compare Logical). Will man nur ausgewählte Bits miteinander vergleichen, so müssen die anderen Bits ausgeblendet werden, und zwar entweder durch OR mit Einsen oder durch AND mit Nullen (man muß nur beide Operanden auf gleiche Weise maskieren).

Ausgewählte Gotchas:

1. Vergleichen. Manche als “logischer Vergleich” (Logical Compare) bezeichnete Befehle führen keine XOR- oder XNOR-Verknüpfungen aus, sondern nur UND-Verknüpfungen. Es sind also keine echten Vergleichsbefehle (mit Vergleichsaussage *gleich/ungleich* als Ergebnis). Sie testen vielmehr nur gesetzte Bits (Aussage: *im 1. Operanden ist keines der im 2. Operanden gesetzten Bits gesetzt/es ist wenigstens eines der besagten Bits gesetzt*). Beispiel TEST (x86/IA-32).

2. Negation (Einerkomplement) und Zweierkomplement. Beides ist nicht das gleiche. Aufpassen – die Assembler-Mnemonics sind nicht immer so selbsterklärend, wie man es naiverweise erwartet. 1. Beispiel: x86/IA-32: Zweierkomplement = NEG, Negation = NOT. 2. Beispiel: Atmel AVR; Zweierkomplement = NEG, Negation (Einerkomplement) = COM.

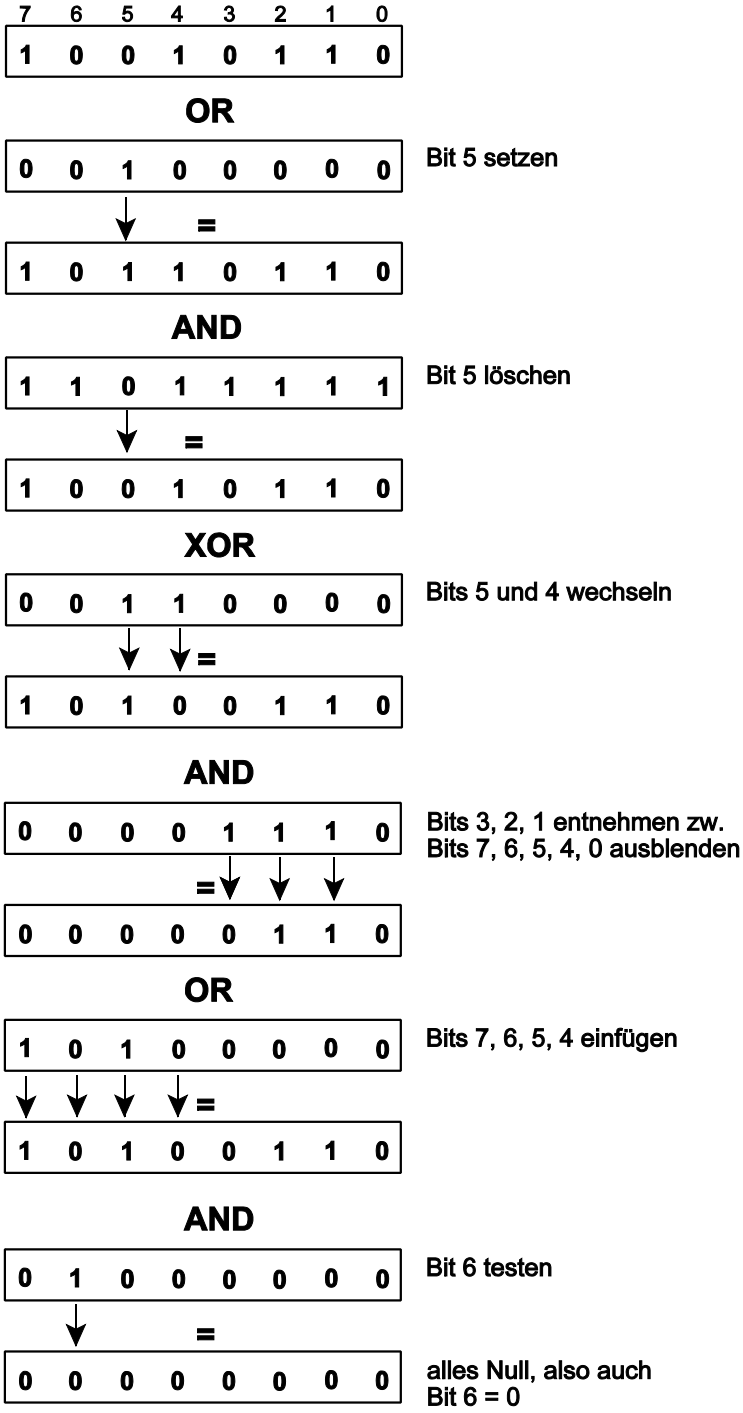


Abb. 5.1 Elementare Bitoperationen anhand von Beispielen

6. Elementare Datenstrukturen

Adressierbare Behälter
Die einfachsten Datenstrukturen sind lediglich Aneinanderreihungen einer jeweils festen Anzahl 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 praktisch 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.

Rechts- und Linksadressierung
Wenn wir Bytes, Worte usw. näher betrachten, ist es wichtig, zu wissen, wie die einzelnen Bits numeriert werden und worauf die Adressenangaben zeigen. Es gibt zwei Adressierungs- bzw. Numerierungsweisen: 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 Significand Bit, LSB) zumeist ganz rechts dargestellt, das höchstwertige Bit (Most Significand 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.

Beispiel:
In den Intel-Architekturen (x86, IA-32, IA-64) gilt die Rechtsadressierung konsequent vom Bit an (sowohl Rechtsadressierung als auch -indizierung). Bit 0 bezeichnet die Binärstelle 2^0, Bit 1 2^1 usw. Jede Auswahlangabe (Bit-Index, Adresse) betrifft somit stets die jeweils niedrigstwertige Position, praktisch das "untere Ende" der betreffenden Informationsstruktur (solche Prozessoren werden deshalb gelegentlich als "Little Endian"-Maschinen bezeichnet).

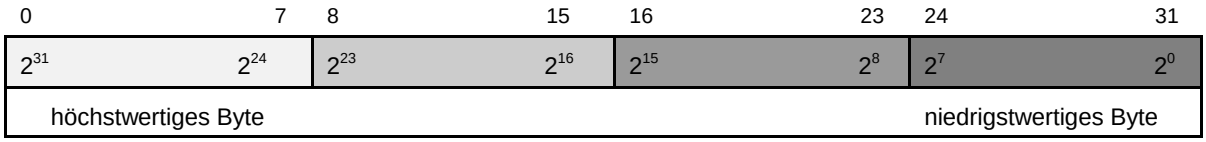
Linksadressierung
Die Anfangsadresse einer Informationsstruktur zeigt immer auf deren höchstwertiges Byte.

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.

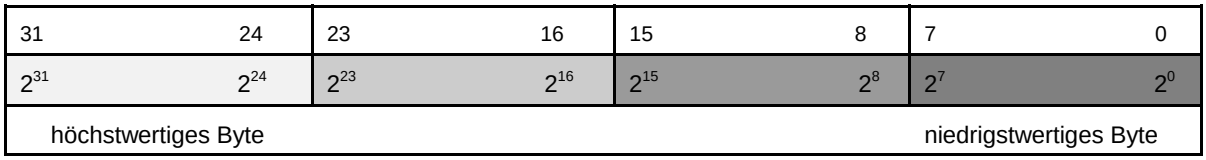
Abbildung 6.1 zeigt beide Adressierungsweisen im Vergleich.

Beispiel:
In den IBM-Mainframes gilt die Linksadressierung konsequent vom Bit an (sowohl Linksadressierung als auch -indizierung). In einem Byte hat Bit 0 die Wertigkeit 2^7 und Bit 7 die Wertigkeit 2^0. Jede Auswahlangabe (Bit-Index, Adresse) betrifft somit stets die jeweils höchstwertige Position, praktisch das "obere Ende" der betreffenden Informationsstruktur (solche Prozessoren werden deshalb gelegentlich als "Big Endian"-Maschinen bezeichnet).

Linksadressierung (Big Endian)



Rechtsadressierung (Little Endian)



Anordnung im Speicher

Rechtsadressierung

Linksadressierung

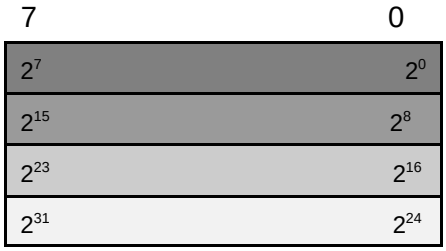


Abb. 6.1 Links- und Rechtsadressierung

Vor- und Nachteile
Ob man sich beim Architektur-Entwurf für Links- oder Rechtsadressierung entscheidet, hängt davon ab, wofür die Maschinen bevorzugt eingesetzt, das heißt, welche Datenstrukturen vorzugsweise gespeichert und verarbeitet werden sollen. Rechtsadressierung und -indizierung ist vorteilhaft, wenn es sich vorwiegend um binär codierte Angaben handelt (Bit-Index bzw. Adresse entsprechen direkt der binären Wertigkeit; die niederen Bitpositionen – die beim Rechnen zuerst verarbeitet werden – werden auch als erste adressiert (Anordung im Speicher entspricht Verarbeitungsreihenfolge)). 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 und /370) hingegen mit Linksadressierung. Da heutzutage beide Adressierungsweisen weit verbreitet sind, ist die Entscheidung praktisch eine reine Kompatibilitätsfrage.

6. Übungsaufgaben

1. Stellen Sie die folgenden Bitfolgen hexadezimal dar:

   a) 1101 0110B
   b) 01 1011B
   c) 10110011 10001110B

2. Geben Sie folgende hexadezimal dargestellte Werte binär an:

   a) 3F2H
   b) 22CCH
   c) 127H

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?

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

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.

6. Wandeln Sie folgende Dezimalzahlen in Binärzahlen um (binäre Darstellung):

   a) 55
   b) 123
   c) -123 (Zweierkomplementdarstellung)

7. Wandeln Sie folgende Binärzahlen in Dezimalzahlen um:

   a) 110 1101 (vorzeichenlos)
   b) 1110 1101 (Zweierkomplementdarstellung)
   c) 3E2H

8. Wie sieht die größte negative Zahl aus, die sich mit 23 Bits angeben läßt (Hexadezimaldarstellung)? Welchen Wert hat diese Zahl?

9. Die Bitposition 3 (Rechtsindizierung) in einem Byte soll gelöscht werden. Welche Verknüpfung wenden Sie hierfür an? Wie sieht der entsprechende Operand aus (Hexadezimaldarstellung)?

10. Wir arbeiten mit 16-Bit-Hardware und Sättigungsarithmetik für natürliche Binärzahlen. Welche Resultate (Headezimaldarstellung) ergeben sich bei folgenden Berechnungen (Operanden in dezimaler Darstellung):

    a) 12 391 - 12 381
    b) 22 345 + 51 933
    c) 12 381 - 12 391