de Gruyter Lehrbuch
Niemeyer · Programmieren in ASSEMBLER

Gerhard Niemeyer

Einführung in das

Programmieren in ASSEMBLER

Systeme IBM, Siemens, Univac Comparex, IBM-PC/370

6., bearbeitete und erweiterte Auflage

Walter de Gruyter · Berlin · New York 1989

Dr. rer. pol. *Gerhard Niemeyer*,
o. Professor der Betriebswirtschaftslehre, insbes.
Wirtschaftsinformatik, an der Universität Regensburg

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Niemeyer, Gerhard:
Einführung in das Programmieren in ASSEMBLER : Systeme IBM, Siemens, Univac, Comparex, IBM-PC/370 / Gerhard Niemeyer. - 6., bearb. u. erw. Aufl. - Berlin ; New York : de Gruyter, 1989
(De-Gruyter-Lehrbuch)
ISBN 3-11-012174-3

Satz und Druck: Mercedes-Druck, Berlin.
Bindearbeiten: Dieter Mikolai, Berlin.

Vorwort

Die Programmiersprache ASSEMBLER ist eine maschinenorientierte Sprache für Bytemaschinen der Serien IBM/360, IBM/370, Siemens 4004 und Univac 9000. Aufgrund der außerordentlich großen Verbreitung dieser Anlagen im Bereich der Wirtschaft und der öffentlichen Verwaltung gehört ASSEMBLER zu den meist verwendeten Programmiersprachen für kaufmännische und verwaltungstechnische Aufgaben.

Die von den Herstellern angebotenen problemorientierten Programmiersprachen COBOL (common business oriented language), PL/1 und RPG (report program generator), die ebenfalls auf derartige Aufgabenstellungen zugeschnitten sind und die den Programmieraufwand reduzieren sollen, konnten ASSEMBLER nicht aus seiner führenden Position verdrängen. Dies mag teils mit der absoluten Universalität und dem Testkomfort der ASSEMBLER-Sprache zusammenhängen und teils durch die spürbar längeren Übersetzungszeiten für die in problemorientierten Sprachen geschriebenen Programme begründet sein. Hinzu kommen sicherlich auch „gefühlsmäßige" Präferenzen durch die Programmierer, die an der Programmiersprache ASSEMBLER die größere Freiheit der Programmgestaltung und die größere Transparenz der eigentlichen Datenverarbeitungsvorgänge in der Maschine schätzen.

In der Tat erscheinen diese Faktoren geeignet, insbesondere den Neuling auf dem Gebiet der elektronischen Datenverarbeitung für diese Tätigkeit zu interessieren und ihn gründlich in dieses Gebiet einzuführen. Aus diesem Grunde sieht das vom Verfasser konzipierte Informatik-Studium für Studenten der Wirtschaftswissenschaften an der Universität Regensburg u. a. eine Einführung in ASSEMBLER im Grundstudium vor.

Über diese didaktischen Überlegungen hinaus muß jedoch auch auf die Bedeutung der ASSEMBLER-Sprache als universelle Basis für „höhere" Programmieraufgaben, wie Systemprogrammierung, Compilerbau und Aufbau von Datenbank- und Informationssystemen hingewiesen werden. Das Erlernen der ASSEMBLER-Sprache schafft also einen idealen Ausgangspunkt für vielerlei berufliche Spezialisierungen im Bereich der elektronischen Datenverarbeitung.

Das vorliegende Lehrbuch ist als möglichst umfassende Einführung in ASSEMBLER gedacht und kann darüber hinaus vom Programmierer auch als Nachschlagewerk benutzt werden; letzteres allerdings mit der Einschränkung, daß wegen der Knappheit des verfügbaren Raumes nicht der gesamte Sprachumfang dargestellt werden konnte. Jedoch dürfte das Gebotene für die meisten Aufgabenstellungen ausreichen.

Das Lehrbuch ist in vier Hauptabschnitte unterteilt:
Der erste Abschnitt behandelt die Datendarstellung in Bytemaschinen, der zweite die Maschineninstruktionen und der dritte die ASSEMBLER-Sprache.

Der vierte Abschnitt zeigt die Anwendung anhand mehrerer Beispiele, die von einfachen kaufmännischen Problemen bis hin zu Problemen, wie sie im Compilerbau vorkommen, reichen.

Bei der Lektüre des Buches empfiehlt es sich, mit den Beispielen in Hauptabschnitt 4 zu beginnen. Die dabei auftretenden Fragen können dann unter Benutzung des ausführlichen Schlagwortregisters in den Hauptabschnitten 1 bis 3 sowie im Anhang geklärt werden. Vor der Lektüre des Buches sollten allerdings Grundkenntnisse der Elektronischen Datenverarbeitung vorhanden sein, und zwar besonders hinsichtlich des Aufbaus und der Funktion der Zentraleinheit und der wichtigsten Peripherie-Geräte.

Regensburg, im Februar 1973 *Gerhard Niemeyer*

Vorwort zur 6. Auflage

Schneller als erwartet war die 5. Auflage vergriffen, so daß eine erneute Bearbeitung des Buches fällig wurde. Der bisher erreichte Reifegrad ließ allerdings nur noch wenige Verbesserungen und Aktualisierungen übrig.

Um jedoch den neueren Entwicklungen auf dem Gebiet der Assembler-Programmierung Rechnung zu tragen, wurde ein Kapitel zur Einführung in den 8086/Assembler angefügt. Bei dieser Sprache handelt es sich um den am weitesten verbreiteten Assembler für Mikrocomputer. Er gilt für alle Maschinen, die mit 16-Bit Prozessoren der Intel Familie ausgerüstet und nach dem sogenannten Industriestandard des Marktführers IBM gebaut sind.

Für Interessierte dürfte ein Vergleich dieser Sprache mit dem nun schon „klassischen" 360/ASSEMBLER wichtige Aufschlüsse und Anregungen für die Weiterentwicklung beider Sprachen bringen.

Regensburg, im März 1989 *Gerhard Niemeyer*

Inhalt

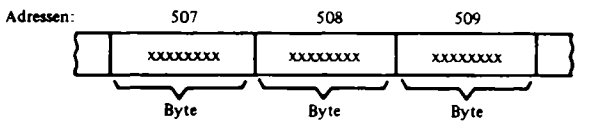
1. Die Darstellung von Daten in Byte-Maschinen

1.1 Die Organisation des Arbeitsspeichers und der Register

a) **Arbeitsspeicher**

Der Arbeitsspeicher der Byte-Maschinen ist so organisiert, daß je 8 bits[1] eine adressierbare Speichereinheit, ein sogenanntes Byte bilden. Die Speicheradresse ist eine Zahl, die die laufende Nummer eines Byte im Speicher angibt. Die Adressierung beginnt mit 0 und endet mit $2^n - 1$, wobei n von der Größe der jeweiligen Anlage abhängt.

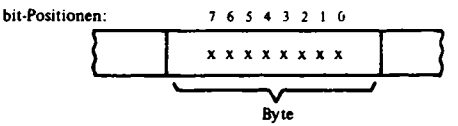
Speicherorganisation (logisch):



Die Größe des Arbeitsspeichers wird üblicherweise in Kilobytes (KB) ausgedrückt, wobei jedoch ein KB nicht 1000 Bytes, sondern $2^{10} = 1024$ Bytes sind.

Für die Zwecke der Datenverschlüsselung ordnet man den einzelnen bits eines Byte sogenannte bit-Positionen zu, und zwar ist es praktisch, dem äußerst rechten bit die Position 0 und dem äußerst linken bit die Position 7 zu geben.

Die bit-Positionen im Byte:



Jedes bit hat zwei mögliche Zustände, denen man die Werte 1, 0 oder „an", „aus" oder „ja", „nein" zuordnen kann. Für die nachfolgenden Darstellungen sollen durchgängig die Werte 1 und 0 verwendet werden.

Da der Zustand jedes bit unabhängig von den Zuständen der übrigen bits variierbar ist, lassen sich nach den Gesetzen der Kombinatorik aus den 8 bits eines Byte $2^8 = 256$ verschiedene bit-Muster bilden.

[1] In Wirklichkeit besteht ein Byte aus 9 bits. Das 9. bit dient jedoch als „parity bit" ausschließlich der maschineninternen Funktionskontrolle bei der Datenübertragung und braucht daher den Programmierer nicht zu interessieren.

Zur Veranschaulichung sind nachfolgend die $2^3 = 8$ möglichen bit-Muster aus 3 bits dargestellt.

0 0 0	1 0 0
0 0 1	1 0 1
0 1 0	1 1 0
0 1 1	1 1 1

Zur Übung bilde man die $2^4 = 16$ möglichen bit-Muster aus 4 bits.

Ordnet man jedem bit-Muster eine bestimmte Bedeutung zu, so lassen sich in einem Byte 256 verschiedene Informationen verschlüsseln.

Für bestimmte Zwecke können auch mehrere Bytes zu größeren logischen Einheiten zusammengefaßt werden, wodurch sich die Zahl der möglichen bit-Muster beträchtlich steigern läßt. Die bit-Positionierung läuft dann entsprechend über die Bytegrenzen hinweg (vgl. dazu auch S. 18). Folgende Gruppierungen sind möglich:

2 Bytes (16 bits) = ein Halbwort
4 Bytes (32 bits) = ein Vollwort
8 Bytes (64 bits) = ein Doppelwort

Der Zugriff zu diesen Gebilden erfolgt durch die Angabe der jeweils niedrigsten Byteadresse.

b) **Register**

Byte-Maschinen verfügen über 20 Operandenregister, zu denen der Programmierer direkten Zugriff hat. Von diesen sind 16 sogenannte Festpunktregister mit je 32 bits und 4 sogenannte Gleitpunktregister mit je 64 bits.

Die einzelnen bits der Register werden in der Regel durch Flip-Flop-Schaltungen realisiert mit Kippzeiten im Nanosekundenbereich. Zum Vergleich sei erwähnt, daß die Ummagnetisierungszeiten im Arbeitsspeicher im Mikrosekundenbereich liegen. Die Zustandsänderungen der einzelnen bits vollziehen sich also in den Registern etwa um den Faktor 10^3 mal schneller als im Arbeitsspeicher. Deswegen werden die Register vor allem dann benutzt, wenn hohe Rechenleistungen erzielt werden sollen.

Die Adressierung der 16 Festpunktregister geschieht durch die Nummern 0 bis 15 und die Adressierung der Gleitpunktregister durch die Nummern 0, 2, 4 und 6. Die Unterscheidung von Festpunkt- und Gleitpunktregistern mit gleichen Nummern geschieht durch den jeweiligen Befehlscode, mit dem die Register angesprochen werden; d.h. es gibt spezifische Festpunktregister- und spezifische Gleitpunktregisterbefehle.

1.2 Der EBCDI-Code

Der Extended Binary Coded Decimal Interchange Code ist typisch für Byte-Maschinen. Er basiert auf dem Prinzip:

1 Byte = 1 Zeichen (Ziffer, Buchstabe, Sonderzeichen)

Die hauptsächlichen Anwendungen dieses Code liegen bei der Zeichenkettenverarbeitung und bei der Datenein- und Datenausgabe.

Zeichenkettenverarbeitung bedeutet Umschichtung von Bytefolgen (oder einzelner Bytes) im Arbeitsspeicher und Vergleichen von Bytefolgen (oder einzelner Bytes) untereinander. Für diese Arbeiten sind die Daten in der Regel EBCDI-verschlüsselt.

Die Ein- und Ausgabe ist bei Bytemaschinen so eingerichtet, daß Eingabedaten von Lochkarten oder Lochstreifen zunächst in den EBCDI-Code umgewandelt werden. Umgekehrt können Daten über den Schnelldrucker oder den Bedienungsblattschreiber nur ausgegeben werden, wenn sie im Arbeitsspeicher in EBCDI-Codierung bereitstehen.

1.2.1 Die Darstellung von Ziffern, Zahlen und Vorzeichen

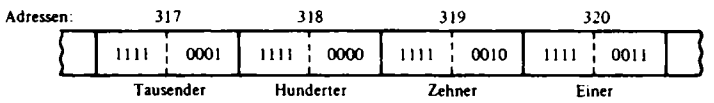
Für die Zifferndarstellung werden im Byte nur die bit-Positionen 0 bis 3 benötigt. Die bit-Positonen 4 bis 7 werden mit Einsen aufgefüllt.

Zifferndarstellung:

EBCDI	dezimal
1 1 1 1 0 0 0 0	0
1 1 1 1 0 0 0 1	1
1 1 1 1 0 0 1 0	2
1 1 1 1 0 0 1 1	3
1 1 1 1 0 1 0 0	4
1 1 1 1 0 1 0 1	5
1 1 1 1 0 1 1 0	6
1 1 1 1 0 1 1 1	7
1 1 1 1 1 0 0 0	8
1 1 1 1 1 0 0 1	9
7 6 5 4 3 2 1 0 (bit-Positionen)	

Die Zahlendarstellung im EBCDI-Code erfolgt durch die logische Zusammenfassung mehrerer Ziffernbytes, wobei jedem Byte ein bestimmter dezimaler Stellenwert zugeordnet wird.

Zahlendarstellung (Beispiel):



Legt man fest, daß das Byte Nr. 317 den Stellenwert 10^3, das Byte Nr. 318 den Stellenwert 10^2, das Byte Nr. 319 den Stellenwert 10^1 und das Byte Nr. 320 den Stellenwert 10^0 hat, dann läßt sich die oben dargestellte Bytekette als die Zahl 1023 interpretieren.

Die Vorzeichendarstellung geschieht im Byte mit dem niedrigsten Stellenwert in den bit-Positionen 7 6 5 4. Zahlen, die in diesen Positionen die bit-Muster 1 1 1 1 aufweisen, gelten als positiv.

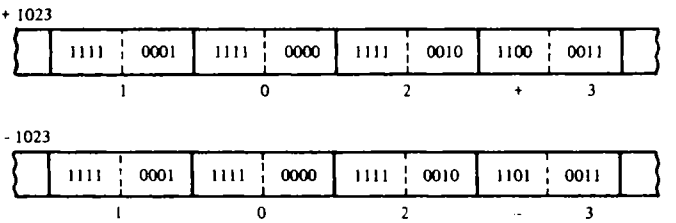
Zur besonderen Markierung positiver Zahlen werden jedoch in diesen bit-Positionen auch folgende bit-Muster verwendet:

1 1 0 0
1 0 1 0
1 1 1 0
(1 1 1 1)

Zur Markierung negativer Zahlen dienen folgende bit-Muster:

1 1 0 1
1 0 1 1

Vorzeichendarstellung (Beispiele):



1.2.2 Die Darstellung von Buchstaben und Sonderzeichen

Buchstaben werden im Byte in der Weise dargestellt, daß die bit-Muster 0 0 0 1 bis 1 0 0 1 zur Darstellung der Ziffern 1 bis 9 in den bit-Positionen 3 2 1 0 mit den bit-Mustern 1 1 0 0, 1 1 0 1 und 1 1 1 0 in den bit-Positionen 7 6 5 4 kombiniert werden. Die Darstellung geschieht in Analogie zum Lochkartencode, wobei die bit-Muster 1 1 0 0, 1 1 0 1 und 1 1 1 0 die Funktionen der Zonenlochungen 12, 11 und 0 übernehmen. In diesem Zusam-

menhang nennt man auch die bit-Positionen 7 6 5 4 den „Zonenteil" und die bit-Positionen 3 2 1 0 den „Zifferteil" des Byte.

Buchstabendarstellung

bit-Pos.	3210	3210	3210	3210	3210	3210	3210	3210	3210
7654	0001	0010	0011	0100	0101	0110	0111	1000	1001
1100	A	B	C	D	E	F	G	H	I
1101	J	K	L	M	N	O	P	Q	R
1110		S	T	U	V	W	X	Y	Z

Für die Verschlüsselung der Sonderzeichen werden ebenfalls der Zonenteil und der Zifferteil des Byte benötigt. Eine unmittelbare Analogie zum Lochkartencode läßt sich hierbei jedoch nicht herstellen.

Sonderzeichendarstellung

bit-Pos.	3210	3210	3210	3210	3210	3210	3210	3210
7654	0000	0001	1010	1011	1100	1101	1110	1111
0100	blank		¢	.	<	(	+	|
0101	&		!	$	*	)	;	¬
0110	–	/		,	%	_	>	?
0111			:	#	@	'	=	"

1.2.3 Die sedezimale Schreibweise von bit-Mustern

Bei der Angabe von Byte-Inhalten ist es recht umständlich, die bit-Muster hinzuschreiben. Auch lassen sich diese bit-Muster wegen ihrer Unübersichtlichkeit nur schwer entschlüsseln. Um nun das Schreiben und Lesen von Byte-Inhalten zu erleichtern, wurde die sedezimale Schreibweise geschaffen.

Diese Schreibweise beruht auf der Einteilung des Byte in zwei 4-er Gruppen (Tetraden) von bit-Positionen und der Kennzeichnung der bit-Muster in jeder Tetrade durch ein einfaches Zeichen. Die Aufteilung des Byte in zwei Tetraden ergibt sich aus der Natur des EBCDI-Code, bei dem für die Verschlüsselung von Ziffern, Buchstaben und Sonderzeichen die bit-Positionen 7 6 5 4 und 3 2 1 0 als selbständige logische Einheiten zusammenwirken.

Für diese logischen Einheiten sind neben den Bezeichnungen „Zonenteil" und „Zifferteil" vor allem im Zusammenhang mit der sedezimalen Schreibweise auch die Bezeichnungen „linkes Halbbyte" und „rechtes Halbbyte" gebräuchlich.

Mit den 4 bits einer Tetrade lassen sich $2^4 = 16$ verschiedene bit-Muster erzeugen, denen man je eine Ziffer des Sedezimalsystems, zu Deutsch 16-er System, zuordnet. Die 16 Ziffern des Systems lauten 0 1 2 3 4 5 6 7 8 9 A B C D E F.

Die Zuordnung der Ziffern zu den bit-Mustern geschieht nun nicht willkürlich, sondern nach folgendem sinnreichen Prinzip:

Die einzelnen bit-Positionen der Tetrade erhalten duale Stellenwerte, und zwar erhält die Position Nr. 0 den Wert 2^0, die Position Nr. 1 den Wert 2^1 usw.; allgemein gilt:

$$\text{dualer Stellenwert} = 2^{\text{bit-Position}}$$

Wichtig ist, daß die bit-Positionen in jeder Tetrade, also auch im linken Halbbyte von 0 bis 3 numeriert werden. Multipliziert man jetzt die Einsen eines bit-Musters mit den entsprechenden Stellenwerten und addiert man die Produkte, so erhält dieses bit-Muster einen bestimmten Dezimalwert. Dabei können sich Dezimalwerte zwischen 0 und 15 ergeben. Diesen Werten ordnet man schließlich die Sedezimalziffern 0 bis F entsprechend den Positionsnummern 0 bis 15 in der oben dargestellten Reihenfolge zu.

Beispiel: Das bit-Muster 1011 hat den Dezimalwert $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$. Die elfte Sedezimalziffer lautet aber B. Folglich wird dem bit-Muster 1011 die Ziffer B zugeordnet.

Zuordnung von Sedezimalziffern zu bit-Mustern

bit-Muster				Dezimalwert	Sedezimalziffer
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	8
1	0	0	1	9	9
1	0	1	0	10	A
1	0	1	1	11	B
1	1	0	0	12	C
1	1	0	1	13	D
1	1	1	0	14	E
1	1	1	1	15	F
2^3	2^2	2^1	2^0	(duale Stellenwerte)	

Anm.: Die Zeichen A bis F sind hier nicht als Buchstaben, sondern als Sedezimalziffern zu verstehen!

Der Vorzug dieses Systems besteht wie gesagt in der Möglichkeit der Kompaktdarstellung von bit-Mustern in Halbbytes. Statt 1 1 1 0 schreibt man einfach E, oder statt 1 0 0 1 einfach 9. Nutzt man dies für die in den vorangehenden Kapiteln dargestellten EBCDI-Verschlüsselungen von Ziffern, Zahlen, Buchstaben und Sonderzeichen aus, so erhält man nunmehr folgende Codiertabellen:

Ziffern

dezimal	EBCDI (sedezimal)
0	F0
1	F1
2	F2
3	F3
4	F4
5	F5
6	F6
7	F7
8	F8
9	F9

Zahlen

dezimal	EBCDI (sedezimal)
(+) 1023	F1F0F2F3
+ 1023	F1F0F2C3
	od. F1F0F2A3
	od. F1F0F2E3
– 1023	F1F0F2D3
	od. F1F0F2B3

Buchstaben

linkes Halbbyte	rechtes Halbbyte 1 2 3 4 5 6 7 8 9
C	A B C D E F G H I
D	J K L M N O P Q R
E	S T U V W X Y Z

Sonderzeichen

linkes Halbbyte	rechtes Halbbyte 0	1	A	B	C	D	E	F
4	blank		¢	.	<	(	+	|
5	&		!	$	*	)	;	¬
6	–	/		,	%	_	>	?
7			:	#	@	'	=	"

In den nachfolgenden Ausführungen wird ausgiebig von der sedezimalen Schreibweise Gebrauch gemacht. Aus diesem Grund wird empfohlen, sich mit den Grundzügen dieser Schreibweise vertraut zu machen. Vor allem sollte stets beachtet werden, daß mit jeder Sedezimalziffer ein 4-bit-Muster gemeint ist, denn nur in dieser Form befinden sich die Daten in der Maschine.

1.3 Die dezimal gepackte Zahlendarstellung

Eine Eigenheit von Bytemaschinen ist, daß arithmetische Operationen nicht im EBCDI-Code, sondern entweder im sogenannten dezimal gepackten oder im dualen Code oder in der sedezimalen Gleitpunktdarstellung vorgenommen werden.

Der ASSEMBLER-Programmierer hat dabei für die Umcodierung zu sorgen; d.h. die im EBCDI-Code eingelesenen Zahlen müssen zum Zweck der Dezimalrechnung gepackt werden. Umgekehrt ist zu beachten, daß das Ausdrucken von Zahlen nur im EBCDI-Code möglich ist; d.h. die Resultate irgendwelcher Rechnungen müssen zu diesem Zweck wieder entpackt werden.

Soll dual gerechnet werden, so ist zu beachten, daß die Umwandlung vom EBCDI-Code in den Dual-Code nur über den dezimal gepackten Code möglich ist. Das gleiche gilt auch für die Rückumwandlung.

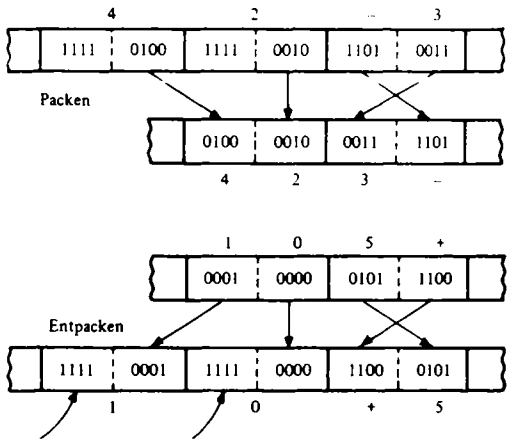
Die Code-Umwandlung in die und aus der sedezimalen Gleitpunkt-Darstellung ist nur mit Hilfe größerer Programmstücke möglich. Dabei werden in der Regel sowohl die dezimal gepackte als auch die duale Zahlendarstellung als Zwischenstufen benötigt.

Der Vorgang des Packens besteht nun darin, daß die EBCDI-verschlüsselten Zahlen in ein anderes Bytefeld umgespeichert werden, wobei jedoch nur die rechten Halbbytes, also die Ziffernteile übertragen werden. Zugleich wird so übertragen, daß in den neuen Bytes beide Halbbytes mit Ziffern belegt werden; die ohnehin redundanten bit-Muster 1 1 1 1 (sedezimal: F) aus den linken Halbbytes entfallen dabei. Ausgenommen von dieser Regel ist die niederst-

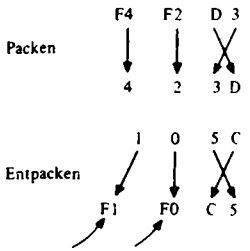
wertige (äußerst rechte) Stelle des Zahlenfeldes. Hier wird das bit-Muster 1 1 1 1 (bzw. 1 1 0 1, falls die Zahl negativ ist) zum Zweck der Vorzeichendarstellung mit übertragen, jedoch tauschen Vorzeichen und Ziffer die Halbbytes.

Der Vorgang des Entpackens verläuft genau entgegengesetzt, wobei die bit-Muster 1 1 1 1 automatisch in die linken Halbbytes eingesetzt werden. Auch das Vorzeichen wechselt wieder das Halbbyte.

Beispiele:



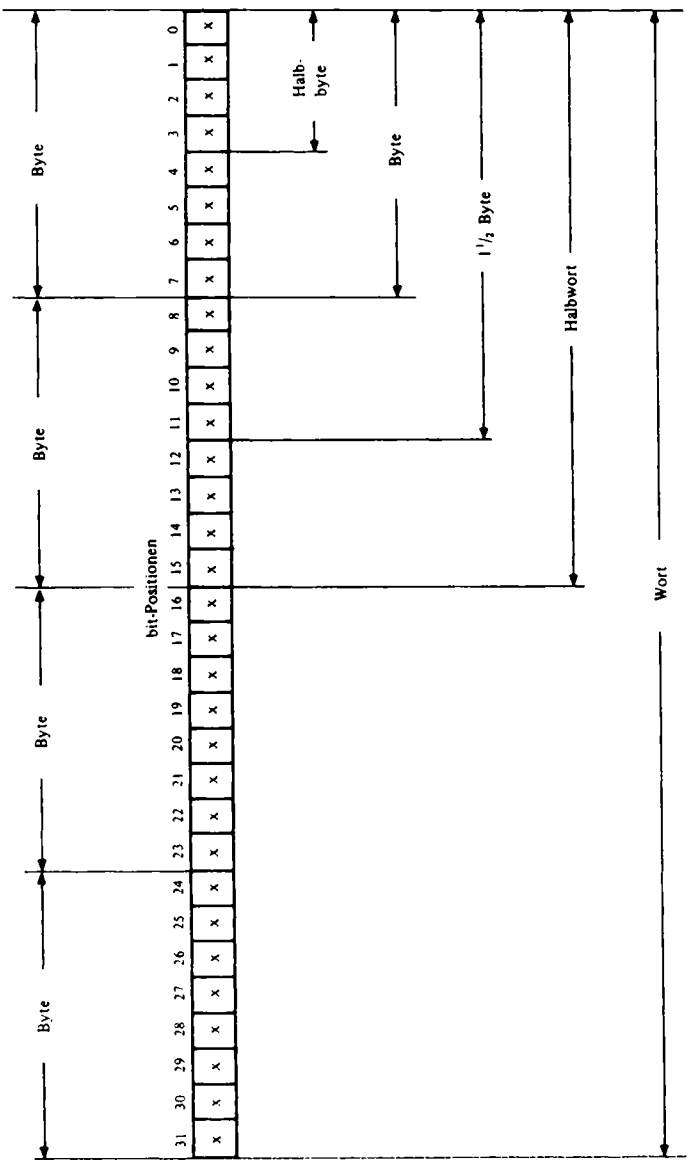
In sedezimaler Schreibweise stellen sich diese Vorgänge wie folgt dar:



1.4 Die duale Zahlendarstellung

Die duale Zahlendarstellung spielt eine wichtige Rolle bei der Berechnung von Arbeitsspeicheradressen und bei der arithmetischen Verarbeitung numerischer Daten. Für letztere sind die Zahlen von der EBCDI-Verschlüsselung über die dezimal gepackte Form in die duale Codierung zu überführen.

Die duale Zahlendarstellung tritt auf im Halbbyte, Byte, $1^1/_2$ Byte, Halbwort (2 Bytes), Wort (4 Bytes) und in den Festpunktregistern. Das allgemeine Codierungsprinzip ist, daß man den bit-Positionen dieser Bytegebilde duale Stellenwerte zuordnet, und zwar stets von rechts nach links, wie das folgende Schema zeigt.



Der duale Stellenwert ergibt sich aus dem Ausdruck

$$2^{\text{bit-Position}}$$

Die in diesen Bytegebilden dargestellten Zahlen haben dann folgenden Dezimalwert:

$$z_{n-1} \cdot 2^{n-1} + z_{n-2} \cdot 2^{n-2} + \ldots + z_2 \cdot 2^2 + z_1 \cdot 2^1 + z_0 \cdot 2^0 = \text{Zahl}_{10}$$

Dabei haben die z_i die Werte 0 oder 1 und n ist die Anzahl der Dualziffern.

Beispiel:

$$01011001_2 = 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$= 0 + 64 + 0 + 16 + 8 + 0 + 0 + 1 = 89_{10}$$

Die größte im Halbbyte darstellbare Zahl ist:

$$1111_2 = 2^3 + 2^2 + 2^1 + 2^0 = 8 + 4 + 2 + 1 = 15_{10} = 2^4 - 1$$

Die Zahl der möglichen bit-Muster ist hingegen $16 = 2^4$.

Die größte im Byte darstellbare Zahl ist:

$$11111111_2 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$$
$$= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$$
$$= 255_{10}$$
$$= 2^8 - 1$$

Die Zahl der möglichen bit-Muster ist hingegen $256 = 2^8$.

Die größte in $1^1/_2$ Bytes darstellbare Zahl ist:

$$111111111111_2 = 2^{11} + 2^{10} + 2^9 + 2^8 + (2^8 - 1)$$
$$= 2048 + 1024 + 512 + 256 + 255$$
$$= 4095_{10} = 2^{12} - 1$$

Die Zahl der möglichen bit-Muster ist hingegen $4096 = 2^{12}$.

Die Zahlen 15, 255 und 4095 treten im Zusammenhang mit der Adreßrechnung sehr häufig auf. Es wird daher empfohlen, sie im Gedächtnis zu behalten.

1.5 Die Zahlendarstellung im Halbwort, Wort und Festpunktregister

Halbwort und Wort im Arbeitsspeicher sind Operanden für die Dualarithmetik, die mit Hilfe der Festpunktregister abgewickelt wird. Die in diesen Bytegebilden verschlüsselten Zahlen müssen daher auf die Zahlendarstellung in den Festpunktregistern abgestimmt sein.

Gegenüber der rein dualen Zahlendarstellung besteht hierbei insofern ein Unterschied, als im Halbwort, Wort und Festpunktregister die jeweils höchste

(äußerst linke) bit-Position der Vorzeichendarstellung vorbehalten ist, und zwar gilt:

Positive Zahlen beginnen mit einer 0 und negative Zahlen mit einer 1 in der 15. bzw. 31. bit-Position.

Ferner gilt für negative Zahlen die Besonderheit, daß sie ausschließlich im sogenannten Zweierkomplement dargestellt werden. Diese Komplementierung geschieht in der Weise, daß von der bit-Position 0 ausgehend alle Nullen (so vorhanden) bis einschließlich der ersten Eins übernommen und daß von da ab alle Dualziffern umgekehrt werden. Also aus Einsen werden Nullen und umgekehrt. Um diese Komplementierung braucht sich jedoch der Programmierer normalerweise nicht zu kümmern, da sie von der Maschine automatisch vorgenommen wird. Allerdings sind diese Maschinenfunktionen auch durch den Programmierer ansprechbar.

Beispiele:

$+ 105_{10}$ $\qquad\qquad$ $- 105_{10}$

15 31 0 $\qquad$ 15 31 0

0 . . . . 01101001 $\Rightarrow$ 1 10010111

$- 76_{10}$ $\qquad\qquad$ $+ 76_{10}$

15 31 0 $\qquad$ 15 31 0

1 . . . . 10110100 $\Rightarrow$ 0 01001100

Das letzte Beispiel zeigt, daß die Komplementierung einer negativen Dualzahl (des Zweierkomplements) wieder die ursprüngliche positive Dualzahl ergibt.

Die größten im Halbwort und Wort darstellbaren positiven Zahlen sind:

15 14 31 30 0

0 1 1 . . . . . 1 ,

was den Dezimalzahlen $2^{15} - 1 = 32767$ bzw. $2^{31} - 1 = 2147483647$ entspricht.

Die dem Betrag nach größtmöglichen negativen Zahlen im Wort und Halbwort sind:

15 14 31 30 0

1 0 0 . . . . . 0 ,

was den Dezimalzahlen $-2^{15} = -32768$ bzw. $-2^{31} = -2147483648$ entspricht; dabei wird also die Vorzeichenstelle zugleich auch als Ziffernstelle ausgenutzt. Die Komplemente dieser größten negativen Zahlen ergeben sich nach der genannten Regel ebenfalls zu

15 31 0

1 0 0 . . . . . 0 ,

d.h. sie bleiben negativ. Daher ist die Komplementierung der größten negativen Zahlen nicht möglich.

Die Dualzahl

```
15                    0
31                    0
 1 1 1 . . . . . 1 1
```

stellt eine dezimale – 1 dar, denn ihr Komplement ist

```
15                    0
31                    0
 0 0 0 . . . . . 0 1 ,
```

also eine dezimale + 1.

Die Dualzahl

```
15                    0
31                    0
 1 1 1 . . . . . 1 1 0
```

stellt eine dezimale – 2 dar, denn ihr Komplement ist

```
15                    0
31                    0
 0 0 0 . . . . . 0 1 0 ,
```

also eine dezimale + 2.

Die Dualzahl

```
15                    0
31                    0
 1 1 1 . . . . . 1 0 1
```

stellt eine dezimale – 3 dar, denn ihr Komplement ist

```
15                    0
31                    0
 0 0 0 . . . . . 0 1 1 ,
```

also eine dezimale + 3.

Zur Übung stelle man die Zahlen – 4 bis – 10 im Dual-Code dar und überprüfe sie durch Rückkomplementierung.

1.6 Exkurs: Zahlensysteme

1.6.1 Die polynomiale Zahlendarstellung

Das allen geläufige dezimale Zahlensystem basiert auf 10 Ziffern, nämlich 0 bis 9. Diese stehen in Gruppen zusammen und bilden durch Zuordnung zu dezimalen Stellenwerten die Dezimalzahlen.

Nimmt man also beispielsweise die Zifferngruppe 9172 und ordnet man z. B. der 9 den Stellenwert 1000, der 1 den Stellenwert 100, der 7 den Stellenwert

10 und der 2 den Stellenwert 1 zu, so erhält man die Dezimalzahl

$$9 \cdot 1000 + 1 \cdot 100 + 7 \cdot 10 + 2 \cdot 1 \; ;$$

in Worten: neuntausendeinhundertzweiundsiebzig.

Wählt man andere Stellenwerte, z. B. 10, 1, $^1/_{10}$, $^1/_{100}$, so ergibt sich die Dezimalzahl

$$9 \cdot 10 + 1 \cdot 1 + 7 \cdot {}^1/_{10} + 2 \cdot {}^1/_{100} ;$$

die üblicherweise als 91,72 (in Worten: einundneunzig-Komma-siebenzwei) geschrieben wird.

Aus den Beispielen erkennt man, daß es sich bei den Stellenwerten durchweg um 10er-Potenzen handelt. Dezimalzahlen lassen sich daher allgemein als Polynome der folgenden Art beschreiben:

$$Z_{10} = z_{n-1} \cdot 10^{n-1} + \ldots + z_2 \cdot 10^2 + z_1 \cdot 10^1 + z_0 \cdot 10^0 + z_{-1} \cdot 10^{-1} + \ldots$$
$$\ldots + z_{-m} \cdot 10^{-m}$$

Darin bedeuten:

Z_{10}:	Dezimalzahl
10:	Basis des Zahlensystems
z_i:	Dezimalziffer, $0 \leqslant z_i \leqslant 9$
10^i:	Stellenwert der Ziffer z_i
n:	Zahl der Ziffern „vor dem Komma"
m:	Zahl der Ziffern „nach dem Komma"

Nun läßt sich nicht nur auf der Basis 10, sondern auf der Basis jeder natürlichen Zahl größer 1 ein Zahlensystem aufbauen. Zwei solcher Zahlensysteme, nämlich das duale und das sedezimale, wurden bereits vorgestellt.

Die allgemeine Form der Zahlensysteme lautet:

$$Z_b = z_{n-1} \cdot b^{n-1} + \ldots + z_2 \cdot b^2 + z_1 \cdot b^1 + z_0 \cdot b^0 + z_{-1} \cdot b^{-1} + \ldots$$
$$\ldots + z_{-m} \cdot b^{-m}$$

Darin bedeuten:

Z_b:	Zahl eines bestimmten Zahlensystems
b:	Basis des Zahlensystems
z_i:	Ziffer, $0 \leqslant z_i \leqslant b - 1$
b^i:	Stellenwert der Ziffer z_i
n:	Anzahl der Ziffern „vor dem Komma"
m:	Anzahl der Ziffern „nach dem Komma"

Übersicht über die im ASSEMBLER relevanten Zahlensysteme:

	Dezimalsystem	Dualsystem[2]	Sedezimalsystem
Basis	10	2	16
Ziffern	0 bis 9	0 bis 1	0 bis F
Stellenwerte	$...10^2\,10^1\,10^0\,10^{-1}...$	$...2^3\,2^2\,2^1\,2^0$	$...16^2\,16^1\,16^0\,16^{-1}...$

1.6.2 Die Umrechnung von Zahlen in andere Zahlensysteme

Der ASSEMBLER-Programmierer wird häufig vor der Aufgabe stehen, Zahlen eines Zahlensystems in Zahlen eines anderen Zahlensystems umzurechnen. Diese Arbeit wird durch die in allen Programmierhandbüchern abgedruckten Umrechnungstabellen erleichtert (vgl. dazu auch Anhang S. 317). Jedoch dekken diese Tabellen zumeist nur einen eng begrenzten Zahlenbereich ab; auch sind diese Tabellen nicht immer zur Hand. Aus diesen Gründen ist es nützlich, die diesen Tabellen zugrundeliegenden Umrechnungsverfahren zu kennen.

a) Umrechnung von Dualzahlen und Sedezimalzahlen in Dezimalzahlen

Diese Umrechnung benutzt die polynomiale Zahlendarstellung, und zwar ist

$$Z_{10} = z_{n-1} \cdot b^{n-1} + \ldots + z_2 \cdot b^2 + z_1 \cdot b^1 + z_0 \cdot b^0 + z_{-1} \cdot b^{-1} + \ldots$$
$$\ldots + z_{-m} \cdot b^{-m}$$

Darin bedeuten:

- Z_{10}: gesuchte Dezimalzahl
- z_i: Dualziffer (Sedezimalziffer) als Dezimalzahl geschrieben
- b^i: dualer (sedezimaler) Stellenwert als Dezimalzahl geschrieben

Beispiele:

(1) Die Dualzahl 011101 ist in eine Dezimalzahl umzurechnen.

$$011101_2 = 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$= 0 + 16 + 8 + 4 + 0 + 1 = 29_{10}$$

(2) Die Sedezimalzahl A2E ist in eine Dezimalzahl umzurechnen.

$$A2E_{16} = A \cdot 16^2 + 2 \cdot 16^1 + E \cdot 16^0$$
$$= 10 \cdot 256 + 2 \cdot 16 + 14 \cdot 1 = 2606_{10}$$

[2] Gebrochene Dualzahlen sind im Zusammenhang mit der elektronischen Datenverarbeitung nicht üblich.

(3) Die Sedezimalzahl E.3 ist in eine Dezimalzahl umzurechnen.

$$E.3_{16} = E \cdot 16^0 + 3 \cdot 16^{-1} = 14 \cdot 1 + 3 \cdot {}^1/_{16} = 14.1875_{10}$$

b) Umrechnung ganzzahliger Dezimalzahlen in Dualzahlen und in Sedizimalzahlen

Die Umrechnung von ganzzahligen Dezimalzahlen in andere Zahlensysteme geht ebenfalls von der bereits vorgeführten polynomialen Schreibweise aus; jedoch bricht hier wegen der Ganzzahligkeit das Polynom mit dem Stellenwert b^0 ab.

$$Z_{10} = z_{n-1} \cdot b^{n-1} + \ldots + z_3 \cdot b^3 + z_2 \cdot b^2 + z_1 \cdot b^1 + z_0 \cdot b^0$$

Bekannt sind jetzt aber die Dezimalzahlen Z_{10} und die Basis b des Zahlensystems, in das Z_{10} umgerechnet werden soll. Gesucht werden hingegen die Ziffern z_{n-1} bis z_0 der betreffenden Zahl im b-Zahlensystem.

Aus diesem Grunde wird das obige Polynom zunächst mit Hilfe des Hornerschemas dargestellt:

$$Z_{10} = z_0 + b\,(z_1 + b\,(z_2 + b\,(z_3 + \ldots + b \cdot z_{n-1}))) \ldots)$$

Sodann werden die gesuchten Ziffern z_i durch fortgesetzte Modulodivision[3] dieses Polynoms mit der Basis b ermittelt. Dies ist möglich, weil die Ziffern z_i wegen $0 \leqslant z_i \leqslant b - 1$ stets kleiner als die Basis b sind und daher als Divisionsreste herausfallen.
Betrachtet man das obige Hornerschema, so ist leicht einzusehen, daß bei der Division von Z_{10} durch b das Resultat

$$z_1 + b\,(z_2 + b\,(z_3 + \ldots + b \cdot z_{n-1})) \ldots)$$

lautet, wobei z_0 als Rest bleibt. Dividiert man dieses Resultat wieder durch b, so erhält man

$$z_2 + b\,(z_3 + \ldots + b \cdot z_{n-1}) \ldots)$$

und als Rest bleibt z_1, usw.

Geht man ohne Beschränkung der Allgemeinheit von einem Polynom vierten Grades aus und bezeichnet man alle im Laufe des Algorithmus auftretenden Dividenden mit y_i, dann läßt sich dieses Polynom wie folgt auflösen:

[3] Modulodivision ist eine Division, bei der als Resultat der jeweilige Divisionsrest genommen wird.

$$Z_{10} = \underbrace{z_0 + b(z_1 + b(z_2 + b(z_3 + b \cdot z_4)))}_{y_0}$$

$$y_1 = z_1 + b\underbrace{(z_2 + b(z_3 + b \cdot z_4))}_{}$$

$$y_2 = z_2 + b\underbrace{(z_3 + b \cdot z_4)}_{}$$

$$y_3 = z_3 + b \cdot \underbrace{z_4}_{y_4}$$

Der schematische Ablauf des Algorithmus stellt sich dann wie folgt dar:

$Z_{10} = y_0 : b = y_1$ Rest z_0

$y_1 : b = y_2$ Rest z_1

$y_2 : b = y_3$ Rest z_2

$y_3 : b = y_4$ Rest z_3

$y_4 : b = 0$ Rest z_4

Die Dezimalzahl Z_{10} wird also im b-Zahlensystem durch die Ziffernfolge $z_4\ z_3\ z_2\ z_1\ z_0$ repräsentiert.

Beispiele:

(1) Gegeben sei die Dezimalzahl 38; gesucht wird die entsprechende Dualzahl

$38 : 2 = 19$ Rest 0 (z_0)

$19 : 2 = 9$ Rest 1 (z_1)

$9 : 2 = 4$ Rest 1 (z_2)

$4 : 2 = 2$ Rest 0 (z_3)

$2 : 2 = 1$ Rest 0 (z_4)

$1 : 2 = 0$ Rest 1 (z_5)

Es ist also $38_{10} = 100110_2$

Probe:

$1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$

$= 32 + 0 + 0 + 4 + 2 + 0 = 38_{10}$

(2) Gegeben sei die Dezimalzahl 38; gesucht wird die entsprechende Sedezimalzahl

$38 : 16 = 2$ Rest 6 (z_0)

$2 : 16 = 0$ Rest 2 (z_1)

Es ist also $38_{10} = 26_{16}$

Probe:

$2 \cdot 16^1 + 6 \cdot 16^0 = 32 + 6 = 38_{10}$

c) **Umrechnung von Dezimalbrüchen in Sedezimalbrüche**

Diese Umrechnung ist vor allem für das Verständnis der sedezimalen Gleitpunktarithmetik von Bedeutung. Ausgangspunkt der Rechnung ist wieder die polynomiale Darstellung; diese sieht für Dezimalbrüche wie folgt aus:

$$.Z_{10} = z_{-1} \cdot b^{-1} + z_{-2} \cdot b^{-2} + \ldots + z_{-m} \cdot b^{-m}$$

oder als Hornerschema geschrieben:

$$.Z_{10} = b^{-1}(z_{-1} + b^{-1}(z_{-2} + b^{-1}(z_{-3} + \ldots + b^{-1} \cdot z_{-m}))) \ldots)$$

Die Ziffern z_{-1} bis z_{-m} werden nun nicht durch fortgesetzte Modulodivision, sondern durch fortgesetzte Multiplikation mit der Basis b gewonnen.

Betrachtet man das obige Hornerschema, so ist leicht einzusehen, daß sich bei der Multiplikation von $.Z_{10}$ mit b als Produkt

$$z_{-1} + b^{-1}(z_{-2} + b^{-1}(z_{-3} + \ldots + b^{-1} \cdot z_{-m})) \ldots)$$

ergibt. Spaltet man z_{-1} ab und multipliziert man den Rest wieder mit b, so ergibt sich z_{-2} usw.

Geht man ohne Beschränkung der Allgemeinheit von einem Polynom (minus) dritten Grades aus und bezeichnet man alle im Laufe des Algorithmus auftretenden Multiplikanden mit y_i, dann läßt sich dieses Polynom wie folgt auflösen:

$$.Z_{10} = \underbrace{b^{-1}(z_{-1} + b^{-1}(z_{-2} + b^{-1} \cdot z_{-3}))}_{y_{-1}}$$

$$y_{-2} = b^{-1}(z_{-2} + \underbrace{b^{-1} \cdot z_{-3}}_{y_{-3}})$$

Der schematische Ablauf des Algorithmus stellt sich dann wie folgt dar:

$$y_{-1} \cdot b = z_{-1} + y_{-2}$$

$$y_{-2} \cdot b = z_{-2} + y_{-3}$$

$$y_{-3} \cdot b = z_{-3}$$

Der Dezimalbruch $.Z_{10}$ wird also im b-Zahlensystem durch die Ziffernfolge $z_{-1}\ z_{-2}\ z_{-3}$ repräsentiert.

Beispiele:

(1) Gegeben sei der Dezimalbruch 0.3; gesucht wird der entsprechende Sedezimalbruch

$$.3 \cdot 16 = 4 + .8$$

$$.8 \cdot 16 = 12 + .8$$

$$.8 \cdot 16 = 12 + .8$$

$$\vdots$$

usw.

Schreibt man die gefundenen Werte z_{-1} als Sedezimalziffern, so erhält man

$$0.3_{10} = 0.4CC\ \ldots\ _{16}$$

(2) Gegeben sei die Dezimalzahl 213.04; gesucht wird die entsprechende Sedezimalzahl

a) Umrechnung des ganzzahligen Teiles:

$$213 : 16 = 13 \quad \text{Rest } 5$$

$$13 : 16 = 0 \quad \text{Rest } 13(D)$$

b) Umrechnung des Dezimalbruchs:

$$0.04 \cdot 16 = 0 + 0.64$$

$$0.64 \cdot 16 = 10(A) + 0.24$$

$$0.24 \cdot 16 = 3 + 0.84$$

$$0.84 \cdot 16 = 13(D) + 0.44$$

$$\vdots$$

usw.

Es ist also $213.04_{10} = D5.0A3D\ldots_{16}$

Probe:

$$13 \cdot 16^1 + 5 \cdot 16^0 + 0 \cdot 16^{-1} + 10 \cdot 16^{-2} + 3 \cdot 16^{-3} + 13 \cdot 16^{-4}$$
$$\approx 208 + 5 + 0 + 0.0391 + 0.0007 + 0.0002$$
$$= 213.04_{10}$$

d) Umrechnung von Dualzahlen in Sedezimalzahlen

Für die Umrechnung einer Dualzahl Z_2 in eine Sedezimalzahl Z_{16} benutzt man am einfachsten wieder die polynomiale Darstellung.

$$Z_{16} = z_{n-1} \cdot 16^{n-1} + \ldots + z_0 \cdot 16^0$$

Die Sedezimalziffern z_{n-1} bis z_0 erzeugt man sich aus der Dualzahl, indem man diese von rechts nach links in Tetraden einteilt (dabei ist die äußerst linke bit-Gruppe eventuell mit führenden Nullen aufzufüllen) und diese dann in die entsprechenden Sedezimalziffern umcodiert.

Beispiel:

Gegeben sei die Dualzahl 111011011. Diese wird eingeteilt und umcodiert:

0001	1101	1011
1	D	B

Dann ist

$$Z_2 = 1 \cdot 16^2 + D \cdot 16^1 + B \cdot 16^0 = 1DB_{16}$$

Zur Probe werden beide Zahlen in Dezimalzahlen umgerechnet:

$$Z_2 = 1 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$
$$= 256 + 128 + 64 + 0 + 16 + 8 + 0 + 2 + 1$$
$$= 475_{10}$$

oder

$$Z_{16} = 1 \cdot 16^2 + D \cdot 16^1 + B \cdot 16^0$$
$$= 1 \cdot 256 + 13 \cdot 16 + 11 \cdot 1$$
$$= 256 + 208 + 11 = 475_{10}$$

e) Umrechnung von Sedezimalzahlen in Dualzahlen

Für die Umrechnung einer Sedezimalzahl Z_{16} in eine Dualzahl Z_2 empfiehlt sich ein analoges Vorgehen.

$$Z_2 = z_{n-1} \cdot 2^{n-1} + \ldots + z_0 \cdot 2^0$$

Die Dualziffern z_{n-1} bis z_0 erzeugt man sich aus den Sedezimalziffern, indem man diese in die entsprechenden Tetraden auflöst.

Beispiel:

Gegeben sei die Sedezimalzahl 2FA; diese Zahl wird in folgende Tetraden aufgelöst:

$$0010\ |1111|\ 1010$$

Dann ist

$$Z_{16} = 1 \cdot 2^9 + 0 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$
$$= 1011111010_2$$

Zur Übung rechne man beide Zahlen in Dezimalzahlen um.

1.7 Die Zahlendarstellung im Gleitpunktregister

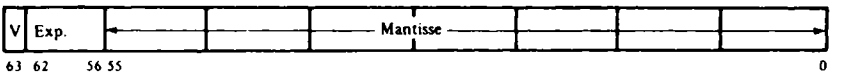
Eine Gleitpunktszahl setzt sich zusammen aus dem Vorzeichen, dem Exponenten und der Mantisse; dabei enthält die Mantisse die Ziffernfolge und der Exponent gibt den Stellenwert der äußerst linken Ziffer an.

Es besteht die Möglichkeit, Zahlen in Gleitpunktregistern in zwei verschiedenen Längen darzustellen und zu verarbeiten, wobei sich diese Längenunterschiede allein auf die Mantisse beziehen.

Kurze Gleitpunktzahlen

Es werden nur die bit-Positionen 63 bis 32 des Gleitpunktregisters benutzt.

Lange Gleitpunktzahlen



Es werden alle bit-Positionen des Gleitpunktregisters benutzt.

Vorzeichen

Positive Zahlen haben in der bit-Position 63 eine 0; negative Zahlen haben dort eine 1. Das Vorzeichen-bit ist das einzige Unterscheidungsmerkmal zwi-

schen positiven und negativen Gleitpunktzahlen. Eine Komplementierung wie bei den dualen Festpunktzahlen findet nicht statt.

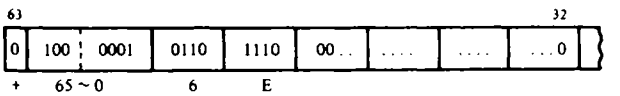
Exponent

Für die Verschlüsselung der Exponenten stehen 7 bit-Positionen zur Verfügung. Damit lassen sich Dezimalzahlen zwischen 0 und 127 dual verschlüsseln. Um nun aber negative und positive Exponenten zur Verfügung zu haben, wird der 0 der Wert -65, der 1 der Wert -64, ... , der 65 der Wert 0, der 66 der Wert $+1$, ... , der 126 der Wert $+61$ und schließlich der 127 der Wert $+62$ zugeordnet. Auf diese Weise gewinnt man Exponenten im Bereich von -65 bis $+62$.

Mantisse

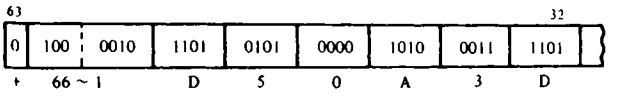
Die Mantisse bietet die Besonderheit, daß in ihr Dualzahlen verschlüsselt werden, daß aber die bit-Muster tetradenweise als Sedezimalziffern interpretiert und verarbeitet werden. Der Exponent gibt also den sedezimalen Stellenwert der äußerst linken Sedezimalziffer der Mantisse (bit-Positionen 55, 54, 53, 52) an.

Beispiele:



$$+ 6 \cdot 16^0 + E \cdot 16^{-1} = + 6.E_{16} = + 6.875_{10}$$

$$- 8 \cdot 16^{-2} = - 0.08_{16} = - 0.03125_{10}$$



$$+ D \cdot 16^1 + 5 \cdot 16^0 + 0 \cdot 16^{-1} + A \cdot 16^{-2} + 3 \cdot 16^{-3} + D \cdot 16^{-4}$$
$$= + D5.0A3D_{16} \approx + 213.04_{10}$$

2. Maschineninstruktionen und Operandenadressierung

2.1 Allgemeine Vorbemerkungen

Bytemaschinen verfügen in der Regel über einen Vorrat von mehr als 140 Maschinenfunktionen, die der Verarbeitung der Daten, also dem Datentransport, dem Datenvergleich, der Datenveränderung und der dynamischen Programm-Modifikation dienen. Diese Funktionen werden durch Maschineninstruktionen ausgelöst; dies sind Bytefolgen im Arbeitsspeicher, deren bit-Muster nach der Decodierung durch das Steuerwerk die betreffenden Funktionsmoduln der Maschine aktivieren.

Eine Folge von Maschineninstruktionen bildet das sogenannte Maschinenprogramm (Objektprogramm), mit dessen Hilfe eine entsprechende Folge von Maschinenfunktionen zur automatischen Abwicklung einer Datenverarbeitungsaufgabe gesteuert wird. Die Erzeugung der Maschineninstruktionen und der Objektprogramme im Arbeitsspeicher wird Gegenstand späterer Erörterungen sein.

Im folgenden sollen zunächst der Aufbau und die Wirkung der Maschineninstruktionen und insbesondere der in ihnen verankerte Zugriff zu den Operanden behandelt werden. Dazu sei vorweg bemerkt, daß es bei Bytemaschinen 4 verschiedene Arten von Operanden gibt, und zwar Registeroperanden, Arbeitsspeicheroperanden, Direktoperanden und Kanalworte[4].

Registeroperanden werden dadurch adressiert, daß in den Maschineninstruktionen die Nummern der Operandenregister (Festpunktregister 0 bis 15, Gleitpunktregister 0, 2, 4, 6) verschlüsselt sind.

Speicheroperanden werden durch Basisadreßregister, Distanzadressen und bei bestimmten Instruktionstypen zusätzlich durch Indexregister adressiert; dabei bestimmt sich die Speicheradresse eines Operanden aus dem Inhalt des Basisadreßregisters plus Distanzadresse bzw. aus dem Inhalt des Basisadreßregisters plus dem Inhalt des Indexregisters plus Distanzadresse. Die Registernummern und die Distanzadressen sind in den Instruktionen verschlüsselt.

Direktoperanden schließlich sind unmittelbarer Bestandteil der betreffenden Maschineninstruktion, d.h. ihr Wert ist in der Instruktion verschlüsselt; eine Adressierung ist daher nicht erforderlich.

[4] Kanalworte werden in dieser Einführung nicht behandelt.

Zur Darstellung des Aufbaus der Maschineninstruktionen werden üblicherweise Symbole verwendet, deren Bedeutung nachfolgend beschrieben wird.

Op.-Code	– Operationscode der Instruktion
R	– Operandenregister
B	– Basisadreßregister (Basisregister)
X	– Indexadreßregister (Indexregister)
D	– Distanzadresse (displacement)
I	– Direktoperand (immediate operand)
L	– Länge eines Speicheroperanden (in Bytes)
M	– Bedingungsmaske bei Sprungbefehlen

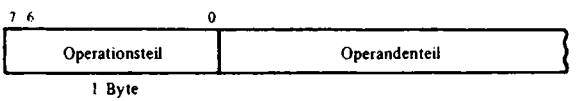
Die Buchstaben R, B, X, D, I, L und M werden mit 1, 2 oder 3 indiziert, um die Operanden zu bezeichnen.

Als Besonderheit bei den Registern gilt zu beachten, daß es sich bei den Basisregistern (B) und Indexregistern (X) stets um die Festpunktregister 0 bis 15 handelt. Die Operandenregister (R) können je nach Operation sowohl Festpunkt- als auch Gleitpunktregister sein. Die Vergabe der Symbole B, X und R richtet sich nur nach der aktuellen Verwendung: Wenn eines der Festpunktregister als Operandenregister für eine arithmetische oder logische Operation benutzt wird, nennt man es R; wenn es als Basisregister verwendet wird, heißt es B und wenn es als Indexregister fungiert, heißt es X. Die Symbole B, X und R sind somit nur funktionallogische Bezeichnungen. In einem Programm kann möglicherweise ein und dasselbe Festpunktregister alle 3 Funktionen ausüben.

2.2 Die Instruktionstypen und ihre Formate

Byte-Maschinen haben 5 Instruktionstypen, die entsprechend der mit ihnen angesprochenen Operandenarten mit den mnemotechnischen Kürzeln RR (Register und Register), RX (Register und indizierter Speicher), RS (Register und Speicher), SI (Speicher und Direktoperand), SS (Speicher und Speicher) gekennzeichnet werden. Bei den SS-Instruktionen unterscheidet man die Versionen SS (logisch) und SS (arithmetisch), so daß man insgesamt auch von 6 Instruktionstypen sprechen kann.

Alle Instruktionen haben einen Operationsteil und einen Operandenteil.



Die Verschlüsselung des Operationsteils erfolgt im äußerst linken Byte. Dabei ist von Interesse, daß die bit-Positionen 7 und 6 die Verschlüsselung des Instruktionstyps und der Instruktionslänge übernehmen:

bit-Positionen 7 6	Instruktionslänge	Instruktionstyp
0 0	2 Bytes (1 Halbwort)	RR
0 1	4 Bytes (2 Halbworte)	RX
1 0	4 Bytes (2 Halbworte)	RS, SI
1 1	6 Bytes (3 Halbworte)	SS

Durch diese Codierung wird das Steuerwerk u. a. in die Lage versetzt, das Befehlszählregister um die Bytezahl des jeweiligen Befehls zu erhöhen und damit auf die Adresse des nachfolgenden Befehls einzustellen.

An das Byte zur Verschlüsselung des Operstionsteils schließen sich je nach Instruktionstyp 1, 3 oder 5 Bytes zur Verschlüsselung des Operandenteils an.

Bei den Operanden handelt es sich einmal um Daten, die mit Hilfe der Instruktionen verarbeitet werden sollen. Diese Daten können in Operandenregistern, im Arbeitsspeicher oder in den Instruktionen selbst stehen. Der betreffende Operandenteil der Instruktionen besteht in diesen Fällen aus der Verschlüsselung der Registernummern oder Speicheradressen. Direktdaten sind in der Instruktion selbst verschlüsselt.

Operanden können ferner aber auch Speicheradressen sein, die als Sprungziele dienen oder irgendwelchen Adreßrechnungen unterworfen werden sollen. Diese Adressen können ebenfalls in Operandenregistern stehen oder aber gleichsam als Direktoperanden auftreten. In diesen Fällen besteht der betreffende Operandenteil der Instruktionen ebenfalls aus der Verschlüsselung der Registernummern oder der betreffenden Speicheradressen.

Operanden können sodann Steuergrößen für shift-Operationen sein. Sie treten in der Form von Speicheradressen als Direktoperanden auf. Es wird jedoch nicht auf eine Speicheradresse zugegriffen, sondern der Wert der „Adresse" bildet die Steuergröße.

Operanden können schließlich Bedingungsschlüssel für Verzweigungsoperationen sein. Auch sie treten als Direktoperanden auf, indem sie wie Registernummern erscheinen. Es wird jedoch nicht auf ein Register zugegriffen, sondern die „Registernummer" bildet die Bedingungsmaske.

Grundsätzlich haben alle Instruktionstypen zwei Operanden, mit Ausnahme einiger RS-Instruktionen, die drei Operanden haben. Jedoch läßt sich dieser Sonderfall so umdeuten, daß auch hier nur zwei Operanden im Spiel sind.