Fortran 90 Kurs

technisch orientiert

Einführung in die Programmierung mit Fortran 90

von
Prof. Dipl.-Ing. Günter Schmitt

R. Oldenbourg Verlag München Wien 1996

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Schmitt, Günter:
Fortran-90-Kurs technisch orientiert : Einführung in die Programmierung mit Fortran 90 / von Günter Schmitt. - München ; Wien : Oldenbourg, 1996
ISBN 3-486-23896-5

Gesamtherstellung: R. Oldenbourg Graphische Betriebe GmbH, München

ISBN 3-486-23896-5
ISBN 978-3-486-23896-9

Inhaltsverzeichnis

Vorwort

Dieses Buch behandelt die Programmiersprache Fortran 90 mit technisch und naturwissenschaftlich orientierten Beispielen und Übungsaufgaben. Es entstand aus dem bewährten „Fortran Kurs" unter Berücksichtigung der neuen Fortran 90 Norm. Dabei wurde besonderer Wert auf die grundlegenden Programm- und Datenstrukturen gelegt, um auch dem Einsteiger, für den Fortran die erste Programmiersprache darstellt, allgemeingültige Programmierkenntnisse zu vermitteln. Besonderheiten des Fortran 90, die nur Anwendungsprogrammierer interessieren, wurden zugunsten der Grundlagen in Sonderabschnitte verlegt.

Alle Beispiele und Übungsaufgaben wurden mit einem Fortran 90 System (Literaturangabe [1] auf einem PC unter dem Betriebssystem DOS getestet. Die Lösungsvorschläge für die Übungen sind im Kapitel 8 abgedruckt. Die Begleitdaten enthalten neben den Quelltexten der Programmbeispiele und Lösungen der Übungsaufgaben auch die im Text eingestreuten unvollständigen Beispiele als komplette Programme. Die Daten sind unter http://www.oldenbourg.de erhältlich.

Günter Schmitt

1. Einführung

Dieses Kapitel beschreibt die Grundlagen der digitalen Rechentechnik. Die Verfahren und Schaltungen werden im Kapitel 6 mit Beispielprogrammen und Übungsaufgaben behandelt. Eilige Leser können diese Einführung zunächst überschlagen.

1.1 Die Darstellung von Daten

Daten sind Zahlen (Gehalt in DM), Zeichen (Buchstabe X), digitalisierte Meßwerte (Raumtemperatur) oder Steuersignale (Meldeleitung eines Druckers). Sie werden im Rechner *binär* gespeichert und verarbeitet. Binär bedeutet zweiwertig:

```
falsch  oder  .FALSE.  oder  0  oder  Low-Potential
  wahr  oder  .TRUE.   oder  1  oder  High-Potential
```

Bei der Ausgabe erscheinen binäre Speicherinhalte normalerweise in einer verkürzten oktalen bzw. hexadezimalen Darstellung entsprechend *Bild 1-1*.

binäre Darstellung	oktal	hexadezimal	dezimal
0 0 0 0	0 0	0 0H $0	0
0 0 0 1	0 1	1 1H $1	1
0 0 1 0	0 2	2 2H $2	2
0 0 1 1	0 3	3 3H $3	3
0 1 0 0	0 4	4 4H $4	4
0 1 0 1	0 5	5 5H $5	5
0 1 1 0	0 6	6 6H $6	6
0 1 1 1	0 7	7 7H $7	7
1 0 0 0	1 0	8 8H $8	8
1 0 0 1	1 1	9 9H $9	9
1 0 1 0	1 2	A 0AH $A	10
1 0 1 1	1 3	B 0BH $B	11
1 1 0 0	1 4	C 0CH $C	12
1 1 0 1	1 5	D 0DH $D	13
1 1 1 0	1 6	E 0EH $E	14
1 1 1 1	1 7	F 0FH $F	15

Bild 1-1: binäre, oktale, hexadezimale und dezimale Darstellungen

In Fortran und den anderen höheren Programmiersprachen arbeitet man bei der Eingabe und Ausgabe von Zahlen im gewohnten dezimalen Zahlensystem. Für die Beurteilung von numerischen Rechenungenauigkeiten und Fehlerzuständen ist es auch für den Fortranprogrammierer unerläßlich, sich mit der binären Darstellungs- und Arbeitsweise einer Rechenanlage vertraut zu machen. Die Normen der Fortran Sprache machen über die Art der Speicherung und Verarbeitung der Daten keine Aussage.

Ein *Bit* ist eine Speicherstelle, die einen der beiden logischen Zustände 0 oder 1 enthält. Ein *Byte* (8 bit) besteht aus 8 Bits, ein *Wort* (16 bit) aus zwei Bytes und ein *Doppelwort* (32 bit) aus vier Bytes. Weitere Einheiten sind das *Kilobyte* (1024 byte) und das Megabyte (1024 kilobyte). Die klein geschriebenen Bezeichnungen bit, byte usw. sind Maßeinheiten für den Informationsgehalt wie z.B. cm für die Länge; groß geschrieben bezeichnen sie Speicherstellen.

Die Assemblersprache kennzeichnet binäre Speicherinhalte durch ein vorangestelltes Zeichen % oder durch den nachgestellten Buchstaben B. Bei hexadezimalen Inhalten wird das Zeichen $ vorangestellt oder der Kennbuchstabe H angehängt; vor den Ziffern A bis F muß eine führende Null stehen. In Fortran verwendet man die Kennbuchstaben B (binär), O (oktal) und Z (hexadezimal) für BOZ Konstanten. Beispiele:

```
%0000 = 0000B =  0H = $0 = B'0000' = O'00' = Z'0'
%1001 = 1001B =  9H = $9 = B'1001' = O'11' = Z'9'
%1010 = 1010B = 0AH = $A = B'1010' = O'12' = Z'A'
%1111 = 1111B = 0FH = $F = B'1111' = O'17' = Z'F'
```

Für die binäre Speicherung von *Zeichen* verwendet man im Betriebssystem DOS den ASCII Code, einen auf 8 bit erweiterten Fernschreibcode; Windows arbeitet mit einem ähnlich aufgebauten ANSI Code. Im Bereich der üblichen Textzeichen sind beide Codes im wesentlichen gleich. Der Anhang zeigt die ASCII Codetabelle. Man unterscheidet:

- Steuerzeichen wie z.B. `00001101 = $0D` für den Wagenrücklauf,
- Sonderzeichen wie z.B. `00101010 = $2A` für das Zeichen *,
- Ziffern wie z.B. `00110000 = $30` für die Ziffer 0,
- Buchstaben wie z.B. `01000001 = $41` für den Buchstaben A sowie
- Umlaute wie z.B. ü als `10000001 = $81` (ASCII) `11111100 = $FC` (ANSI).

Ziffer	0	1	2	3	4	5	6	7	8	9
Code	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Bild 1-2: BCD Code zur Darstellung von Dezimalzahlen

Für die binäre Speicherung von Dezimalzahlen kann der in *Bild 1-2* dargestellte BCD Code verwendet werden. Er entsteht aus der Zeichencodierung der Ziffern durch Entfernen des linken Halbbytes. Beispiel für die Dezimalzahl 13:

```
Zeichencode: 00110001 00110011 binär = 3133 hexa
   BCD Code:     0001     0011 binär = 13 hexa
```

Wegen der schnelleren parallelen Rechenwerke arbeitet man jedoch fast ausschließlich nicht im dezimalen, sondern im **dualen Zahlensystem**. Dies ist ein Stellenwertsystem mit den beiden binären Ziffern 0 und 1; die Wertigkeiten der Dualstellen sind Potenzen zur Basis 2. Negative Exponenten ergeben Stellen hinter dem Dualkomma.

$$Z_3*2^3 + Z_2*2^2 + Z_1*2^1 + Z_0*2^0 + Z_{-1}*2^{-1} + Z_{-2}*2^{-2} \ldots$$

Bei der *Umrechnung* einer Dualzahl in eine Dezimalzahl werden die Dualstellen mit ihrer Stellenwertigkeit multipliziert; die Teilprodukte sind zu addieren. Zur Kennzeichnung des Zahlensystems kann man die Basis als tiefergestellten Index hinter die Ziffernfolge setzen. Beispiel:

$1101{,}101 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3}$
$= 1*8 + 1+4 + 0*2 + 1+1 + 1*0{,}5 + 0*0{,}25 + 1*0{,}125$
$1101{,}101_2 = 13{,}625_{10}$

Bei einer **Dezimal-Dualumwandlung** wird die Dezimalzahl in die dualen Stellenwertigkeiten zerlegt. Das folgende Beispiel verwandelt die Dezimalzahl 13,625 durch Subtrahieren der dualen Stellenwertigkeiten in eine Dualzahl. Stellen vor der höchsten und nach der niedrigsten Wertigkeit sind 0 und werden nicht berücksichtigt.

```
13,625   5,625   1,625   1,625   0,625   0,125   0,125
-8,000  -4,000  -2,000  -1,000  -0,500  -0,250  -0,125
------  ------  ------  -------  -----  ------  ------
 5,625   1,625 negativ!  0,625   0,125 negativ!  0,000 fertig!

=1*8    + 1*4    + 0*2    + 1*1    + 1*½    + 0*¼    + 1*⅛
=1101,101₂
```

Ist der Teiler enthalten, so wird er subtrahiert; und die Dualstelle ist 1. Ist er nicht enthalten, so ist die Dualstelle 0. Ersetzt man die Subtraktionen durch ganzzahlige Divisionen, so ist die Dualstelle gleich dem ganzzahligen Quotienten, das Verfahren ist mit dem ganzzahligen Rest fortzusetzen. Im Rechner werden die Dualzahlen in einer festen Länge als Byte, Wort oder Doppelwort gespeichert und verarbeitet, die Umwandlungsverfahren müssen also auch führende Nullen berücksichtigen.

Das **Teilerverfahren** dividiert die Vorkommastellen einer Dezimalzahl nacheinander durch die dualen Stellenwertigkeiten; bei einer 8-bit-Darstellung im dezimalen Bereich von 0 bis 255 beginnt man mit dem Teiler $2^7 = 128$. Der ganzzahlige Quotient liefert die höchste Dualstelle; der ganzzahlige Rest wird weiter zerlegt. Das folgende Beispiel verwandelt die Dezimalzahl 26 in acht Schritten in eine achtstellige Dualzahl.

```
26 : 128 = 0 Rest 26     26 = 0*128 + 26
26 :  64 = 0 Rest 26        = 0*64  + 26
26 :  32 = 0 Rest 26        = 0*32  + 26
26 :  16 = 1 Rest 10        = 1*16  + 10
10 :   8 = 1 Rest  2        = 1*16  + 1*8 + 2
 2 :   4 = 0 Rest  2        = 1*16  + 1*8 + 0*4 + 2
 2 :   2 = 1 Rest  0        = 1*16  + 1*8 + 0*4 + 1*2 + 0
 0 :   1 = 0 Rest  0        = 1*16  + 1*8 + 0*4 + 1*2 + 0*1 + 0

26 = 0*128 + 0*64 + 0*32 + 1*16 + 1*8 + 0*4 + 1*2 + 0*1
26₁₀  = 00011010₂
```

Das **Divisionsrestverfahren** dividiert die Vorkommastellen der Dezimalzahl durch die Basis des neuen Zahlensystems. Der ganzzahlige Divisionsrest ergibt eine Stelle des neuen Zahlensystems. Das Verfahren wird mit dem ganzzahligen Quotienten fortgesetzt, bis dieser Null ist. Das folgende Beispiel zerlegt die Dezimalzahl 26 in eine Dualzahl (Basis 2) ohne führende Nullen.

```
26 : 2 = 13 Rest 0
13 : 2 =  6 Rest 1
 6 : 2 =  3 Rest 0
 3 : 2 =  1 Rest 1
 1 : 2 =  0 Rest 1
fertig!
       Dualzahl: 1 1 0 1 0   mit führenden Nullen: 00011010₂
```

Die folgende Darstellung zeigt, wie die Reste fortlaufend zerlegt werden, so daß geschachtelte Klammern mit dem Faktor 2 entstehen. Multipliziert man in der letzten Zeile die Faktoren 2 wieder in die Klammern hinein, so entstehen Potenzen zur Basis 2.

```
26 = 2*13 + 0
   = 2*(2*6 + 1) + 0
   = 2*(2*(2*3 + 0) + 1) + 0
   = 2*(2*(2*(2*1 + 1) + 0) + 1) + 0
   = 2*(2*(2*(2*(2*0 + 1) + 1) + 0) + 1) + 0
   = 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0
   = 011010₂          mit führenden Nullen: 00011010₂
```

Bei der ersten Division entsteht die niedrigste Dualstelle, bei der letzten die höchste. Ist die Speicherlänge vorgegeben, so darf das Verfahren nicht beim Quotienten Null abgebrochen werden, sondern ist entsprechend der Anzahl der Stellen mit führenden Nullen fortzusetzen.

Die Umwandlungsverfahren lassen sich auch auf andere Zahlensysteme anwenden. Das folgende Beispiel zeigt das Divisionsrestverfahren zur Umrechnung in das Hexadezimalsystem, das mit der Basis 16 und den Ziffern 0 bis 9 sowie A bis F für die Reste von 10 bis 15 arbeitet. Das **hexadezimale** Zahlensystem entsteht aus dem dualen durch Zusammenfassen bzw. Ausklammern von jeweils vier Dualstellen.

```
26 : 16 = 1  Rest  10    liefert  16*1 + 10
 1 : 16 = 0  Rest  1     liefert  16*(16*0 + 1) + 10

  Hexadezimalzahl: 1 A  = 1*16^1   + 10*16^0  = 1A₁₆
                        = 0001*2^4 + 1010*2^0 = 00011010₂
```

Das **oktale** Zahlensystem arbeitet mit der Basis 8 und den Ziffern 0 bis 7. Es faßt jeweils drei Dualstellen zusammen. Die Umwandlung der Nachkommastellen wird im Zusammenhang mit der Darstellung reeller Zahlen behandelt. Bei der Speicherung von ganzen Zahlen unterscheidet man vorzeichenlose und vorzeichenbehaftete Dualzahlen.

Vorzeichenlose ganze Zahlen werden als natürliche Dualzahlen ohne Vorzeichen gespeichert, die linkeste Stelle hat die höchste Wertigkeit. Bei der Speichereinheit Byte (8 bit) ist dies der Wert 128. *Bild 1-3* zeigt die wichtigsten ganzzahligen Datentypen Byte, Wort und Doppelwort. In der Programmiersprache Fortran sind vorzeichenlose Zahlen standardmäßig nicht vorgesehen.

Typ	Länge	dezimal	hexadezimal	Fortran
Byte	8 bit	0...255	00....FF	
Wort	16 bit	0....65535	0000....FFFF	
Doppelwort	32 bit	0....4294967295	00000000....FFFFFFFF	

Bild 1-3: Vorzeichenlose (unsigned) duale Datentypen

Bei **vorzeichenbehafteten Dualzahlen** verwendet man aus rechentechnischen Gründen eine Zahlendarstellung, bei der negative Werte durch ihr *Komplement* dargestellt werden, positive Zahlen bleiben unverändert. Zur Beseitigung des negativen Vorzeichens addiert man zur negativen Zahl zunächst einen Verschiebewert, der nur aus den größten Ziffern des Zahlensystems (z.B. 11111111 bei 8 bit) besteht. Da der Verschiebewert nur die Ziffern 1 enthält, läßt sich die duale Subtraktion auf die Fälle 1 - 0 = 1 und 1 - 1 = 0 und damit auf die bitweise Negation zurückführen. Es entsteht das *Einerkomplement*, das rechentechnisch durch einen einfachen Negierer (aus 1 mach 0 und aus 0 mach 1) realisiert wird.

Addiert man einen um 1 größeren Verschiebewert (z.B. 11111111 + 1 = 100000000 bei 8 bit), so entsteht das *Zweierkomplement*, das sich durch Weglassen der linkesten (z.B. 9. Stelle) einfacher korrigieren läßt. Das folgende Beispiel zeigt die Darstellung der Zahl $-26_{10} = -00011010_2$ im Zweierkomplement.

```
 Verschiebewert:    1 1 1 1 1 1 1 1     (für Einerkomplement)
  negative Zahl:  - 0 0 0 1 1 0 1 0
Einerkomplement:    1 1 1 0 0 1 0 1
                  +                 1     (für Zweierkomplement)
Zweierkomplement:   1 1 1 0 0 1 1 0
```

Positive Zahlen werden nicht komplementiert, sie müssen in der linkesten Bitposition eine führende Null als positives Vorzeichen enthalten. Bei *negativen* Zahlen entsteht durch die Zweierkomplementdarstellung immer eine 1 in der linkesten Bitposition. Durch *Rückkomplementieren* läßt sich das negative Vorzeichen wiederherstellen. Dabei wird wieder *erst* komplementiert und *dann* eine 1 addiert. Das folgende Beispiel verwandelt die negative Zahl 11100110 aus der Zweierkomplementdarstellung wieder in eine Dualzahl mit Vorzeichen:

```
Zweierkomplementdarstellung:    1 1 1 0 0 1 1 0
      bilde Einerkomplement:    0 0 0 1 1 0 0 1
             addiere eine 1:  +                1
          negative Dualzahl:  - 0 0 0 1 1 0 1 0 = - 26(10)
```

Vorzeichenbehaftete Dualzahlen der Länge 8 bit liegen im Bereich von -128 bis +127; die linkeste Bitposition ist keine Dualstelle, sondern enthält das Vorzeichen. Bei vorzeichenlosen Dualzahlen gleicher Länge, die den Zahlenbereich von 0 bis 255 umfassen, hat die linkeste Bitposition die Stellenwertigkeit $2^7 = 128$. *Bild 1-4* zeigt den Fortran Datentyp `INTEGER`, der in drei Speicherlängen (Arten) verfügbar ist.

Typ	Länge	dezimal	hexadezimal	Fortran
Byte	8 bit	-128....+127	80....7F	Art 1
Wort	16 bit	-32768....+32767	8000....7FFF	Art 2
Doppelwort	32 bit	±2147483647	80000000....7FFFFFFF	Art 4

Bild 1-4: Vorzeichenbehaftete (signed) INTEGER *Datentypen*

Reelle Zahlen können Stellen vor und hinter dem Dezimalkomma enthalten. Bei der Umwandlung einer reellen Dezimalzahl in eine Dualzahl werden die Vorkommastellen wie eine vorzeichenlose ganze Zahl behandelt; die Nachkommastellen müssen in die Stellenwertigkeiten $2^{-1} = 0{,}5$ bzw. $2^{-2} = 0{,}25$ bzw. $2^{-3} = 0{,}125$ usw. zerlegt werden. Das dem Divisionsrestverfahren entsprechende Umwandlungsverfahren multipliziert die Dezimalzahl kleiner 1 (nur Nachkommastellen!) fortlaufend mit der Basis des neuen Zahlensystems. Jedes der dabei entstehenden Produkte wird in eine Vorkommastelle und in die restlichen Nachkommastellen zerlegt. Die Vorkommastelle ergibt die Ziffer des neuen Zahlensystems; mit den Nachkommastellen wird das Verfahren fortgesetzt, bis das Produkt Null ist oder die gewünschte Anzahl von Nachkommastellen erreicht wurde. Die erste Multiplikation liefert die erste Stelle hinter dem Komma. Das folgende Beispiel verwandelt die Dezimalzahl 0,6875 in eine entsprechende Dualzahl durch fortlaufende Multiplikationen der Nachkommastellen mit dem Faktor 2.

```
0,6875 * 2 = 1,3750 = 1      + 0,3750
0,3750 * 2 = 0,7500 = 0      + 0,7500
0,7500 * 2 = 1,5000 = 1      + 0,5000
0,5000 * 2 = 1,0000 = 1      + 0,0000
0,0000 * 2 = 0,0000 = 0      + 0,0000
fertig!
```

Dualzahl genau: 0,10110 mit folgenden Nullen: 0,10110000

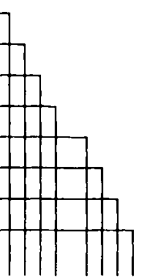
Die folgende Darstellung zeigt, wie durch das Abspalten der Nachkommastellen geschachtelte Klammern mit dem Faktor 0,5 entstehen. Multipliziert man in der letzten Zeile die Faktoren 0,5 wieder in die Klammern hinein, so entstehen Potenzen zur Basis $0{,}5 = \frac{1}{2} = 2^{-1}$.

```
0,6875 = 0,5*(1 + 0,375)
       = 0,5*(1 + 0,5*(0 + 0,7500))
       = 0,5*(1 + 0,5*(0 + 0,5*(1 + 0,5)))
       = 0,5*(1 + 0,5*(0 + 0,5*(1 + 0,5*(1 + 0))))
       = 0,5*(1 + 0,5*(0 + 0,5*(1 + 0,5*(1 + 0,5*(0 + 0)))))
```

$$= 1*2^{-1} + 0*2^{-2} + 1*2^{-3} + 1*2^{-4} + 0*2^{-5} = 0{,}10110_2$$

Das Verfahren kann bei dem Produkt Null beendet werden, da dann nur noch nachfolgende Nullen entstehen, die man in der üblichen Zahlendarstellung fortläßt; bei rechnerinternen Darstellungen wird mit einer festen Anzahl von Nachkommastellen gearbeitet. Im Gegensatz zu Vorkommastellen, die sich immer in eine ganze Dualzahl umwandeln lassen, kann bei der Umwandlung eines endlichen Dezimalbruchs ein *unendlicher* Dualbruch entstehen; das Verfahren muß dann beim Erreichen einer bestimmten Anzahl

von Dualstellen abgebrochen werden; der Rest bleibt als **Umwandlungsfehler** unberücksichtigt. Das folgende *abschreckende* Beispiel zeigt die Umwandlung der Dezimalzahl 0,4 in eine angenäherte Dualzahl mit 8 Dualstellen hinter dem Komma.



```
0,4000 * 2 = 0,8000 = 0          + 0,8000
0,8000 * 2 = 1,6000 = 1          + 0,6000
0,6000 * 2 = 1,2000 = 1          + 0,2000
0,2000 * 2 = 0,4000 = 0          + 0,4000
0,4000 * 2 = 0,8000 = 0          + 0,8000
0,8000 * 2 = 1,6000 = 1          + 0,6000
0,6000 * 2 = 1,2000 = 1          + 0,2000
0,2000 * 2 = 0,4000 = 0          + 0,4000
Abbruch!
   Dualzahl genähert: 0,0110 0110  + Restfehler!
```

Bei der Umwandlung der Dezimalzahl 0,4 ergibt sich eine Periode 0110 in den dualen Nachpunktstellen. Bricht man das Umwandlungsverfahren nach 8 Stellen ab, so entsteht ein Umwandlungsfehler; die Rückrechung in das Dezimalsystem ergibt einen kleineren Wert. Beispiel:

```
0,4 dezimal ≈ 0,01100110 ... dual ≈ 0,3984375 dezimal
```

Reelle Dezimalzahlen werden zunächst getrennt nach Vorkomma- und Nachkommastellen in Dualzahlen verwandelt. Das folgende Beispiel zeigt die Dezimalzahl 26,6875 in der dualen *Festkommadarstellung* (Fixed Point oder Festpunkt) mit jeweils 8 Stellen vor und hinter dem Komma.

```
26,6875 dezimal = 00011010,10110000 dual
```

In der üblichen *Gleitkommadarstellung* (Floating Point oder Gleitpunkt) besteht die Zahl aus einer normalisierten Mantisse und einem Faktor mit einem ganzzahligen Exponenten zur Basis des Zahlensystems. In den folgenden Beispielen enthält die Mantisse eine Vorkommastelle; andere Darstellungen arbeiten nur mit Nachkommastellen.

```
   26,6875 dezimal = 2,66875     * 10^1  dezimal
11010,1011 dual    = 1,10101011 * 2^4   dual
```

Normalisieren bedeutet, das Dezimal- bzw. Dualkomma so zu verschieben, daß es vor bzw. hinter der werthöchsten Ziffer steht; bei Zahlen kleiner als 1 wird der Exponent negativ. Die folgenden Beispiele zeigen den reellen Datentyp REAL der Programmiersprache Fortran, der 4 Bytes (32 bit) im Speicher belegt. Das Vorzeichen der Zahl steht in der linkesten Bitposition, dann folgt der *Absolutwert*, nicht das Komplement wie bei ganzen Zahlen. Die 8-bit-Charakteristik setzt sich zusammen aus dem dualen Exponenten und einem Verschiebewert (127 dezimal = 01111111 dual), der das Vorzeichen des Exponenten beseitigt. Damit ergibt sich ein Zahlenbereich von ca. $-3{,}4*10^{-38}$ bis $+3{,}4*10^{+38}$. Die 23-bit-Mantisse bedeutet eine Genauigkeit von ca. 7 Dezimalstellen. Die führende 1 der Vorpunktstelle wird bei der Speicherung unterdrückt und muß bei allen Umwandlungen und Rechnungen wieder hinzugefügt werden. Beispiel:

Zahlenumwandlung:

$+26{,}6875_{10} = +\ 11010{,}10110000000000000000_2$

$= +\ 1{,}10101100000000000000000_2 * 2^4$ normalisiert

Darstellung als Datentyp REAL:

```
    Vorzeichen:    0
Charakteristik:    |10000011  = 4 + 127 = 131
      Mantisse:    ||||||||10101011000000000000000
Speicher binär:    01000001110101011000000000000000
   hexadezimal:       4   1   D   5   8   0   0   0
```

Gemäß dem Standard IEEE 754 bestehen für den Wert Null sowohl Charakteristik als auch Mantisse aus lauter Nullerbits; wegen des Vorzeichenbits gibt es sowohl eine **+0** als auch eine **-0**. Ist die Charakteristik 0, die Mantisse aber ungleich 0, so entstehen sehr kleine nicht normalisierte Zahlen verminderter Genauigkeit, die unterhalb des normalisierten Zahlenbereiches liegen. Das Bitmuster $7F der Charakteristik dient nicht der Zahlendarstellung, sondern kennzeichnet arithmetische Fehlerzustände wie z.B. *unendlich*; dies kann zum Fehlerabbruch in Systemfunktionen führen. Die folgenden Beispiele sind abhängig vom verwendeten Fortran System. $ bedeutet hexadezimal.

`$00000000` ergibt `0`
`$80000000` ergibt `-0` (was sagt die Mathematik dazu?)
`$00000001` ergibt `1.401298e-45` ($1{,}401298 * 10^{-45}$ nicht normalisiert!)
`$7F800000` ergibt `+INF` (INFinite = unendlich)
`$FF800000` ergibt `-INF`
`$7FC00000` ergibt `+NAN` (Not A Number = keine Zahl)
`$FFC00000` ergibt `-NAN`
`$7F800001` ergibt `z.B. "Floating point error"`

1.2 Rechen- und Speicherschaltungen

Die Rechen- und Speicherschaltungen bilden die Hardware des Computers. Sie lassen sich mit den in *Bild 1-5* zusammengestellten Symbolen der Digitaltechnik beschreiben.

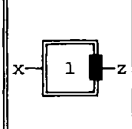
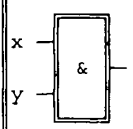
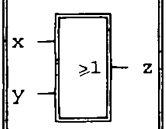
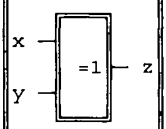
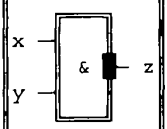
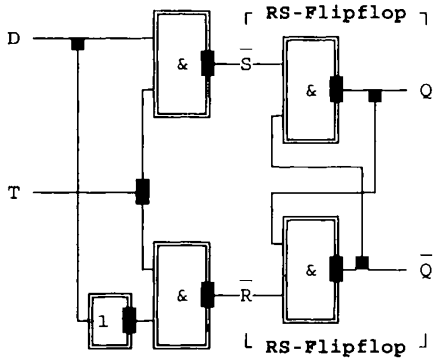
Funktion	Nicht	Und	Oder	Eoder	Nand
Symbol					

Tabelle																	
	x	z	x	y	z	x	y	z	x	y	z	x	y	z	x	y	z
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	1	0	0	1	0	0	1	1	0	1	1	0	1	1	0	1	1
			1	0	0	1	0	1	1	0	1	1	0	1	1	0	1
			1	1	1	1	1	1	1	1	0	1	1	0	1	1	0

Bild 1-5: Logische Grundfunktionen der digitalen Rechentechnik

Die *Nicht*-Schaltung liefert das Einerkomplement nach der Regel: aus 0 mach 1 und aus 1 mach 0. Die *Und*-Schaltung hat nur dann am Ausgang eine 1, wenn alle Eingänge 1 sind; sie liefert das Übertragbit der dualen Addition. Bei der *Oder*-Schaltung ist nur dann der Ausgang 0, wenn alle Eingänge 0 sind. Schließt man den Fall 1 1 an den Eingängen aus, so kann die *Oder*-Schaltung für eine einstellige duale Addition verwendet werden. Die *Eoder*-Schaltung liefert eine 1, wenn beide Eingänge ungleich sind, sie bildet das Summenbit der dualen Addition. Die in Bild 1-6 dargestellte Speicherschaltung ist aus *Nand*-Schaltungen aufgebaut. *Nand* bedeutet Not AND = *Nicht-Und*.

Speicherschaltungen dienen dazu, binäre Zustände (0 oder 1) über längere Zeiträume aufzubewahren. Nichtflüchtige Speicher sind der Magnetspeicher (Disk) und die Festwertspeicherbausteine (EPROM = Erasable Programmable Read Only Memory) des Computers. Flüchtige Speicher verlieren (vergessen) ihren Speicherinhalt nach dem Abschalten der Versorgungsspannung. Dazu zählen der Arbeitsspeicher des Computers (RAM = Random Access Memory) und die Register des Mikroprozessors. *Bild 1-6* zeigt das Prinzip der elektronischen Speicherung in einem **Flipflop**.



Tabelle

T	D	Q	Zustand
0	x	Q	speichern
1	0	0	rücksetzen
1	1	1	setzen

Schaltungssymbol

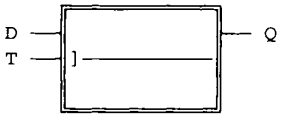


Bild 1-6: Das taktzustandsgesteuerte D-Flipflop

Das einfache *RS-Flipflop* besteht aus zwei rückgekoppelten *Nand*-Schaltungen. Es kann nur zwei stabile Zustände annehmen, also eine 0 oder eine 1 speichern. Eine logische 0 am Setzeingang S bringt die Schaltung in den Zustand Q = 1; eine logische 0 am Rücksetzeingang R bringt es in den Zustand Q = 0. Sind beide Eingänge 1, so speichert die Schaltung den zuletzt eingeschriebenen Zustand. Der Speicherinhalt kann am Ausgang Q abgegriffen werden, ohne daß sich der Zustand des Speichers ändert (zerstörungsfreies Lesen). Nach dem Einschalten der Versorgungsspannung hat das Flipflop einen zufälligen Anfangswert (0 oder 1). Dies gilt auch für aus Flipflops aufgebaute Register und Speicherbausteine (RAM); nach dem Einschalten ist ihr Inhalt nicht vorhersehbar.

Durch Vorsetzen einer **Taktschaltung** entsteht das *D-Flipflop*. Für den Takt T = 1 wird der am Eingang D anliegende Zustand in den Speicher übernommen; für T = 0 speichert die Schaltung den zuletzt eingeschriebenen Wert, Änderungen am Eingang D werden während T = 0 nicht wirksam. Bei einem taktflankengesteuerten Flipflop werden die Daten nur mit einer z.B. steigenden Flanke des Taktsignals übernommen. *Bild 1-7* zeigt ein aus vier taktflankengesteuerten Flipflops bestehendes **Register**.

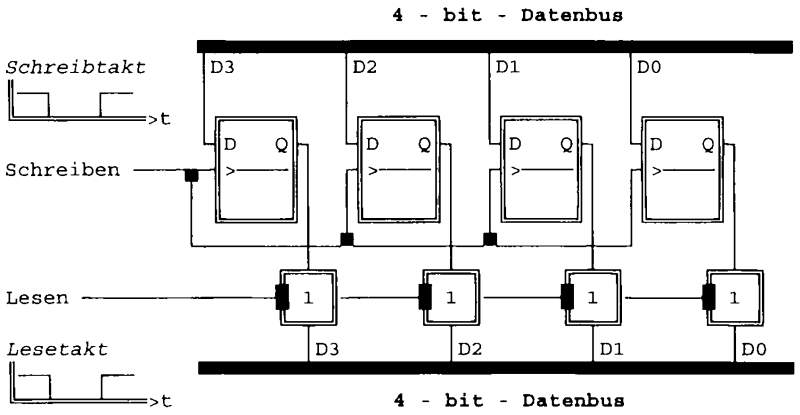


Bild 1-7: Paralleles 4-bit-Register

Mit der steigenden Flanke (0 nach 1) des *Schreibtaktes* übernehmen alle vier Flipflops gleichzeitig (parallel) die an den Eingängen anliegenden Daten. Im Zustand 0 des *Lesetaktes* wird der Inhalt der vier Flipflops zerstörungsfrei ausgelesen und auf den Datenbus geschaltet. Ein *Bus* ist ein Bündel von parallelen Leitungen, an die mehrere Sender und Empfänger angeschlossen sein können. Liegen die *Ausgänge* mehrerer Register parallel, so wird nur das ausgelesen, das einen *Lesetakt* erhält; alle anderen bleiben inaktiv. Bei parallel geschalteten *Eingängen* werden die Daten nur in dem Register gespeichert, das einen *Schreibtakt* erhält; alle anderen bleiben unverändert. Die Speicherauswahl erfolgt durch eine **Adresse**; *Bild 1-8* zeigt die Auswahlschaltung.

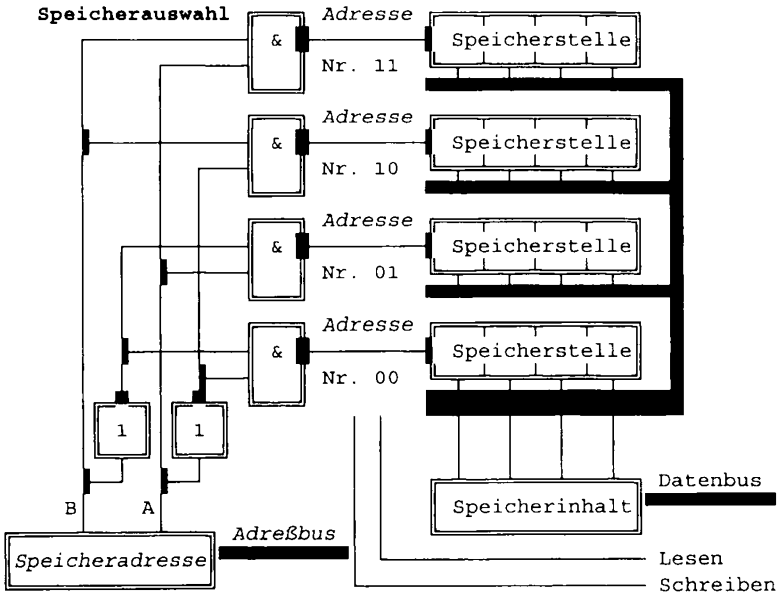


Bild 1-8: Speicherauswahl durch Adressen

Eine **Adresse**, eine vorzeichenlose Dualzahl, ist die "Hausnummer" einer Speicherstelle, über die auf den Inhalt zugegriffen wird. Die Speicherauswahl erfolgt durch logische Verknüpfung der Adreßbits. Z.B. wird für A = 1 und B = 1 das oberste *Nand* der Auswahlschaltung aktiv (Ausgang 0) und gibt die Speicherstelle mit der Adresse $11_2 = 3_{10}$ frei; alle anderen Speicherstellen sind durch ihre Auswahl-*Nands* gesperrt (Ausgänge 1). Ist gleichzeitig das Lesesignal aktiv, so wird die Speicherstelle durch einen Lesetakt ausgelesen ; bei aktivem Schreibsignal werden die am Datenbus anliegenden Daten mit einem Schreibtakt in die Speicherstelle geschrieben. Die auf dem Adreßbus anliegende Speicheradresse bestimmt, welche Speicherstelle gelesen oder beschrieben werden soll. Die Datenübertragung erfolgt über den Datenbus. Die Steuersignale *Lesen* und *Schreiben* bestimmen die Richtung.

Das **Rechenwerk** des Prozessors dient zur Verknüpfung der Daten. Bei der Addition zweier Dualstellen entsteht eine einstellige Summe und im Fall $1 + 1 = 10_2 = 2_{10}$ ein Übertrag auf die nächsthöhere Dualstelle. Die *Rechenregeln* der dualen Addition lauten:

```
0 + 0 = 0 0 = Übertrag 0 Summe 0
0 + 1 = 0 1 = Übertrag 0 Summe 1
1 + 0 = 0 1 = Übertrag 0 Summe 1
1 + 1 = 1 0 = Übertrag 1 Summe 0
```

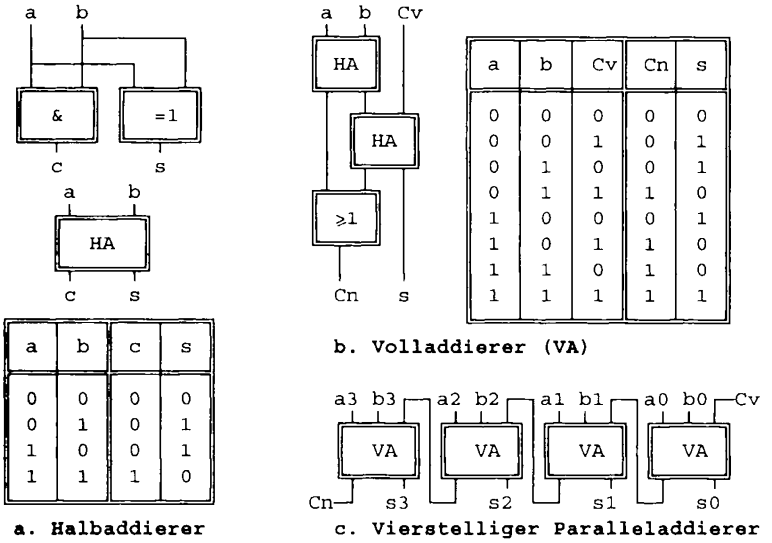


Bild 1-9: Additionsschaltungen

Für die rechentechnische Ausführung der dualen Addition lassen sich die in *Bild 1-9* dargestellten **Additionsschaltungen** verwenden. Der *Halbaddierer* nach Bild 1-9a verknüpft die beiden Dualstellen a und b zu einer Summe s (*Eoder*-Schaltung) und einem Übertrag c (*Und*-Schaltung) auf die nächsthöhere Dualstelle. Für mehrstellige Additionen ist ein *Volladdierer* (Bild 1-9b) erforderlich, der die beiden Dualstellen a und b und den Übertrag Cv der Vorgängerstelle zu einer einstelligen Summe s und einem Übertrag Cn auf die nächste Stelle addiert. Der erste Halbaddierer verknüpft die

beiden Dualstellen a und b, der zweite die Summe des ersten Halbaddierers mit dem Übertrag Cv der Vorgängerstelle. Die *Oder*-Schaltung addiert die beiden Zwischenüberträge der Halbaddierer zu einem Gesamtübertrag Cn, der an die nachfolgende Dualstelle weitergereicht wird. Dabei ist sichergestellt, daß die beiden inneren Zwischenüberträge niemals gleichzeitig 1 sind. Der in Bild 1-9c dargestellte *Paralleladdierer* verknüpft zwei vierstellige Dualzahlen zu einer vierstelligen Summe. Ist der Übertrag Cn des linkesten Volladdierers eine 1, so bedeutet dies, daß eine fünfstellige Summe entstanden ist. *Bild 1-10* zeigt die Erweiterung des Paralleladdierers durch zwei Steuereingänge zum **Addierer/Subtrahierer** mit oder ohne *Zwischenübertrag*. Bewertungsschaltungen prüfen das Ergebnis auf Null, Vorzeichen und Überlauf.

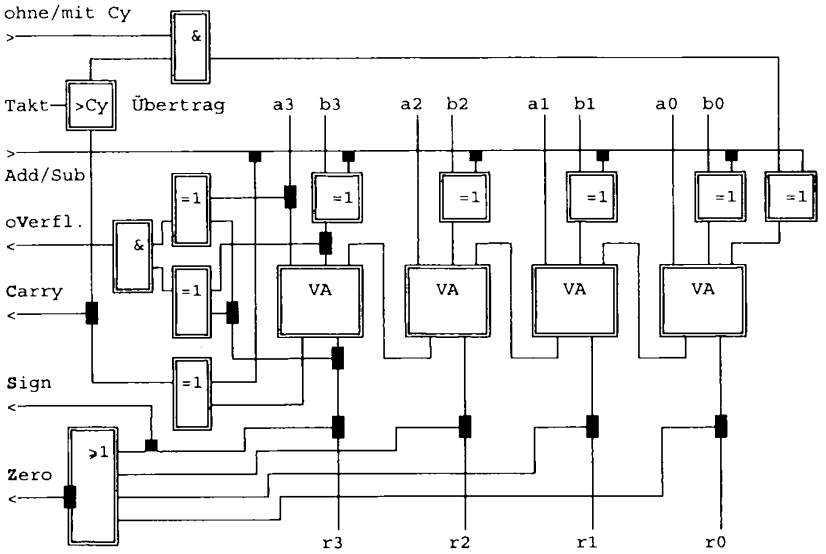


Bild 1-10: Addierer/Subtrahierer mit Bewertungsschaltungen

Bei der **Addition** *vorzeichenloser Dualzahlen* im Bereich von 0 (0000) bis 15 (1111) dient der Carryausgang als Überlaufmarke. Für C = 0 liegt die Summe im zulässigen 4-bit-Bereich; für C = 1 ist ein Ergebnis größer 15 (1111) entstanden. Durch Abschneiden der fünften Stelle entsteht ein Überlauffehler. Beispiel:

```
        1 1 1 1   = 15
      + 1 1 1 1   = 15
--------------------
C = 1   1 1 1 0   = 30   abgeschnitten  1 1 1 0 = 14
```

Der Carryausgang wird in einem besonderen Speicherbit festgehalten und kann dann als *Zwischenübertrag* verwendet werden. Bei der Addition von Dualzahlen z.B. der Länge 16 bit mit einem 4-bit-Addierer beginnt man mit der wertniedrigsten 4-bit-Gruppe und muß bei den folgenden Gruppenadditionen den Übertrag der Vorgängergruppe mit berücksichtigen. Dazu enthält das Rechenwerk einen *Steuereingang* ohne/mit Cy. Die wertniedrigste Gruppe wird ohne Zwischenübertrag Cy addiert, alle anderen mit Cy. Nach der letzten Gruppe liefert das Carrybit wieder die Überlaufmarke.

Die Schaltung addiert auch vierstellige *vorzeichenbehaftete Dualzahlen* im Bereich von -8 (1000) bis +7 (0111). Ein Überlauf entsteht, wenn die Summe zweier positiver Zahlen größer als 7 wird; der Addierer liefert in diesem Fall ein negatives Vorzeichen! Bei einem Unterlauf wird die Summe zweier negativer Zahlen kleiner als -8; durch den Addierer entsteht für die Summe ein positives Vorzeichen! Dieser Vorzeichenwechsel wird durch eine Vergleicherschaltung als Overflow-Fehler erkannt. Beispiele:

```
    0 1 1 1 = +7              1 0 0 0 = -8
+   0 1 1 1 = +7          +   1 1 1 1 = -1
---------------           ---------------
    1 0 0 0 = -8 Überlauf!    0 1 1 1 = +7 Unterlauf!
```

Die **Subtraktion** wird unabhängig von der Zahlendarstellung auf die Addition des Zweierkomplementes zurückgeführt nach der Formel "A - B = A + (-B)". Ist der *Steuereingang* Add/Sub gleich 1, so wird der Operand B durch die *Eoder*-Schaltungen negiert (Einerkomplement); durch die Addition einer 1 am negierten Eingang des wertniedrigsten Volladdierers entsteht das Zweierkomplement. Bei der Subtraktion vorzeichenloser Zahlen kann nun auch ein Unterlauf entstehen, also eine negative Differenz, die ebenfalls durch das nunmehr negierte Carrybit erkannt wird.

Das in Bild 1-10 dargestellte Rechenwerk addiert und subtrahiert sowohl vorzeichenlose als auch vorzeichenbehaftete Dualzahlen mit oder ohne Zwischenübertrag. Die Bewertungsschaltungen verknüpfen Bitmuster unabhängig von der Zahlendarstellung. Nach einer Operation mit *vorzeichenlosen* Dualzahlen ist das Carrybit auszuwerten:

C = 0: Ergebnis im zulässigen Bereich

C = 1: Überlauf (Addition) oder Unterlauf (Subtraktion)

Nach einer Operation mit *vorzeichenbehafteten* Dualzahlen sind das Overflowbit (V-Bit) und das Vorzeichenbit (S = Sign) auszuwerten:

V = 0: Ergebnis im zulässigen Bereich

V = 1: Überlauf oder Unterlauf (Addition bzw. Subtraktion)

S = 0: Ergebnis positiv

S = 1: Ergebnis negativ

Unabhängig von der Rechenoperation und der Zahlendarstellung prüft das Z-Bit (Zero = Null) das Ergebnis auf **Null**. Die *Oder*-Schaltung ist nur dann 0, wenn alle Eingänge und damit alle Bitpositionen des Ergebnisses 0 sind; mindestens eine 1 bringt den Ausgang des Oder auf 1. Durch die Negation entsteht folgende Zuordnung:

Z = 0: Ergebnis *nicht* Null (0 = nein)

Z = 1: Ergebnis *ist* Null (1 = ja)

Für die **Multiplikation** und **Division** muß der Addierer/Subtrahierer durch Schieberegister und Ablaufsteuerungen erweitert werden. Es gelten die gleichen Kontrollbedingungen wie für die Addition und Subtraktion. Die Division wird ganzzahlig durchgeführt und ergibt einen ganzzahligen Quotienten sowie einen ganzzahligen Rest; Stellen hinter dem Komma entstehen nur bei reellen Rechenwerken! Bei einer Division durch 0 bzw. bei einem Divisionsüberlauf liefert das Rechenwerk ein besonderes Signal "*Divisionsfehler*", das zum Abbruch des Programms führt. Beispiele für die Division:

```
9 : 3 = 3 Rest 0
9 : 2 = 4 Rest 1   nicht 4,5 wie bei reellen Zahlen!
9 : 0 = "Divisionsfehler!"
```

Für das Rechnen mit **reellen Zahlen**, die in der dualen Mantisse-Exponent-Darstellung vorliegen, sind besondere Verfahren erforderlich. Sie werden entweder durch Unterprogramme (Software) oder wesentlich schneller hardwaremäßig durch Arithmetikprozessoren (Numerikprozessoren oder Coprozessoren 80x87) ausgeführt, die in modernen Schaltungen bereits auf dem Prozessorbaustein integriert sind. Bei arithmetischen Operationen mit reellen Zahlen können Rundungs- und Anpassungsfehler auftreten. Für die Addition im reellen Rechenwerk wird der Exponent der kleineren Zahl dem der größeren angeglichen. Dies geschieht durch Verschieben des Kommas nach links um die Differenz der Exponenten. Das folgende *dezimale* Beispiel addiert die beiden Zahlen $1{,}000000*10^7$ und $1{,}234567*10^1$, die entsprechend dem Datentyp REAL mit sieben Dezimalstellen, einer Vorkomma- und sechs Nachkommastellen gespeichert werden sollen. Der Anteil $0{,}000000234567*10^7$ oder $0{,}234567*10^1$ oder 2,34567 der kleineren Zahl geht verloren!

```
Speicher:              Reelles Rechenwerk:
1.000000*10^7   =>     1.000000          * 10^7   bleibt
1.234567*10^1   => +   0.000001 234567   * 10^7   anpassen!
                    ---------------------------------------
                    =   1.000001          * 10^7   abgeschnitten
```

Rundungs- und Anpassungsfehler, die besonders bei der Addition und Subtraktion reeller Zahlen entstehen, werden wie die bereits behandelten Fehler bei der Umwandlung von Nachkommastellen nicht berücksichtigt und können zu schwerwiegenden Rechenungenauigkeiten führen. Überschreitungen des Zahlenbereiches und eine Division durch Null werden bei der Rechnung mit reellen Zahlen erkannt und führen in der Regel zu einem Abbruch des Programms.

Vorsicht im Umgang mit Zahlen im Rechner!

Ganze Zahlen sind immer genau; es können Überlauffehler auftreten!
Reelle Zahlen können Umwandlungs- und Rundungsfehler enthalten!

Die Programmiersprache Fortran ist rechnerunabhängig definiert und legt nicht fest, wie die Daten intern gespeichert und verarbeitet werden. Dies gilt auch für das Verhalten bei Fehlerzuständen wie Überlauf oder Division durch Null. Dieses Kapitel ist daher nur als Einführung in die Probleme der digitalen Rechentechnik anzusehen. Der Abschnitt 2.8 Numerische Sonderfragen zeigt die genormten Modelle für ganze und reelle Zahlen sowie eine Reihe von Standardfunktionen, mit denen sich die "Ungenauigkeiten" der reellen Arithmetik untersuchen lassen.

1.3 Aufbau und Arbeitsweise eines Fortran Systems

Fortran ist eine höhere problemorientierte Programmiersprache für naturwissenschaftliche und technische Anwendungen wie Pascal oder C++. Die Sprachelemente wurden zuletzt als **Fortran 90** in den DIN Normen festgelegt. In der Sprache Fortran formulierte Programme müssen auf Rechenanlagen ablaufen; *Bild 1-11* zeigt eine zusammenfassende Übersicht über die Hardware und Software eines Personal Computers (PC).

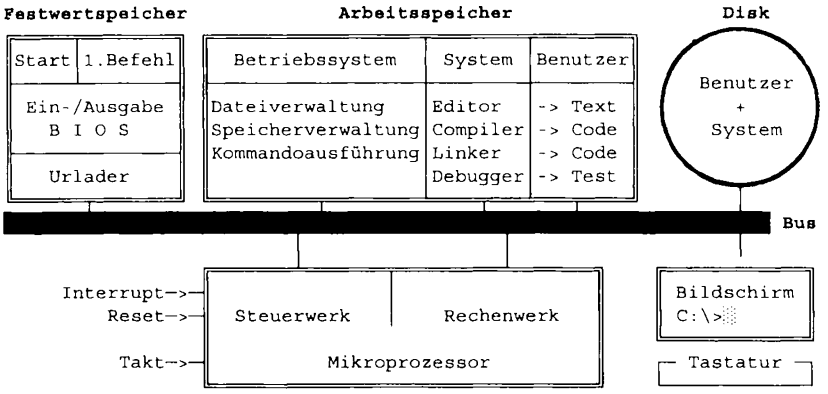


Bild 1-11: Hardware und Software einer Rechenanlage (PC)

Die ***Hardware*** des Rechners besteht aus dem Mikroprozessor, Speichereinheiten und der Peripherie, die durch ein Leitungsbündel, den Bus, miteinander verbunden sind. Der *Festwertspeicher* enthält ein fertiges Maschinenprogramm, das beim Einschalten des Rechners das Betriebssystem startet. Der *Arbeitsspeicher* hat beim Einschalten einen undefinierten Anfangszustand und kann während des Betriebes beschrieben und gelesen werden; beim Abschalten des Rechners gehen die Speicherinhalte verloren. Die *magnetischen Speichereinheiten* Festplatte (Harddisk) und Wechselplatte (Floppydisk) dienen dazu, Daten und Programme längerfristig aufzubewahren. Ein batteriegepufferter Speicher enthält Konfigurationsdaten (z.B. Größe des Arbeitsspeichers und Art der Laufwerke). Als *Peripherie* bezeichnet man die Geräte für die Eingabe und Ausgabe von Daten und Programmen wie den Bildschirm, die Tastatur und den Drucker.

Das ***Betriebssystem*** besteht aus fertigen Programmen, die beim Kauf des Rechners bereits vorinstalliert mitgeliefert werden. Man unterscheidet das Basissystem im Festwertspeicher und die Systemsoftware, die sich üblicherweise auf einem Festplattenlaufwerk befindet. Das Basissystem (BIOS) prüft in der Startphase die Hardware des Rechners, lädt die speicherresidenten Teile des Betriebssystems von der Festplatte in den Arbeitsspeicher und übernimmt dann die Eingabe und Ausgabe der Daten von bzw. zur Peripherie. Der Benutzer gibt über die Tastatur bzw. über die Maus *Kommandos* ein, die vom Betriebssystem ausgeführt werden. Die wichtigsten Aufgaben sind:

- Laden und Starten von Programmen,
- Dateiverwaltung auf den externen Speichereinheiten,
- Organisation des Arbeitsspeichers und
- Dienstleistungen bereitstellen wie z.B. Datum und Uhrzeit.

Betriebssysteme werden von verschiedenen Herstellern und in unterschiedlichen Versionen für verschiedene Rechnersysteme angeboten wie z.B.
- DOS (Disk Operating System) für die Eingabe von Kommandos über die Tastatur,
- Windows mit graphischer Bedienerführung und Mauseingabe und
- UNIX mit Abwandlungen (LINUX) und OS/2 für Workstations (Arbeitsplatzrechner).

Das Betriebssystem kann nur Programme laden und starten, die bereits im binären Maschinencode des Mikroprozessors vorliegen, die Übersetzung aus einer Programmiersprache in diesen ausführbaren Code ist die Aufgabe von Assemblern und Compilern, die zusätzlich zum Betriebssystem erworben werden müssen. Ein *Assembler* übersetzt aus einer prozessornahen symbolischen Programmiersprache in den binären Code des betreffenden Prozessors. Beispiel:

```
LADE        AKKU,X     Code: 1111 0001
ADDIERE     AKKU,Y     Code: 1100 0010
LADE        Z,AKKU     Code: 0011 0011
```

Ein ***Compiler*** übersetzt aus einer prozessorunabhängigen Programmiersprache in den binären Maschinencode eines bestimmten Prozessors. Beispiel:

```
z = x + y              Code: 1111 0001  1100 0010  0011 0011
```

Für die Eingabe und Ausgabe von Daten ruft der Compiler in der Regel betriebssystemabhängige Unterprogramme auf, die von einem *Linker* (Binder) mit dem übersetzten binären Programm verbunden werden müssen. Das Laden und Starten des ausführbaren Maschinenprogramms übernimmt des Betriebssystem.

Ein ***Fortran Entwicklungssystem*** besteht in der Regel aus
- einem *Texteditor*, mit dem der Programmierer den Quelltext eingibt und ändert,
- einem *Fortran Compiler*, der diesen Quelltext in einen Objektcode übersetzt,
- einem *Linker*, der aus dem Objektcode eine ausführbare Ladedatei erzeugt und
- einem *Debugger* (Testhilfeprogramm) zum Aufspüren von Laufzeitfehlern.

Fortran Compiler werden von mehreren Herstellern (z.B. Lahey, Microsoft und Watcom) für unterschiedliche Fortran Versionen (z.B. Fortran 77 und Fortran 90) und für verschiedene Rechner (z.B. PCs oder Workstations) und Betriebssysteme (z.B. Windows oder UNIX) angeboten. Die Fortran Norm beschreibt lediglich die Sprachelemente, aus denen der Quelltext aufzubauen ist; sie legt nicht fest, wie die Anweisungen im Rechner auszuführen sind. Für die Arbeit mit dem Betriebssystem stellen die meisten Compiler über die Norm hinausgehende Unterprogrammbibliotheken zur Verfügung, von denen man jedoch nur sparsam Gebrauch machen sollte, wenn der Quelltext mit einem anderen Compiler oder für ein anderes Rechnersystem übersetzt werden soll. Binäre Maschinenprogramme sind in der Regel nicht ohne weiteres auf andere Rechner übertragbar. In der praktischen Anwendung können sich besonders bei der Eingabe und Ausgabe von Daten sowie in der Behandlung von Fehlerzuständen erhebliche Unterschiede zwischen den Fortran Systemen und Rechnersystemen ergeben.

2. Grundlagen der Fortran Sprache

Dieses Kapitel führt in die Grundlagen ein, die nötig sind, um technische und mathematische Formeln mit einfachen Programmen auswerten zu können. Die Eingabeform der Beispielprogramme wurde so gewählt, daß diese sowohl von Fortran 77 als von Fortran 90 Compilern übersetzt werden können. Es wird vorausgesetzt, daß die Entwicklungsumgebung (Editor, Compiler, Binder und Lader) installiert ist und daß sich der Leser mit ihrer Bedienung sowie mit dem Betriebssystem vertraut gemacht hat.

Die Anweisungen der Fortran Sprache werden in diesem Buch meist in einem Rahmen dargestellt. Kennwörter erscheinen wie in den Beispielprogrammen in Großbuchstaben, vom Benutzer anzugebende Sprachelemente stehen kursiv. Eckige Klammern kennzeichnen optionale Elemente, die auch entfallen können. Elemente, die sich wiederholen lassen, werden durch `[, ... ]` beschrieben. Das folgende Beispiel zeigt die Anweisung **END**, bei der das zusätzliche Kennwort PROGRAM und der vom Benutzer vergebene *Name* entfallen können.

```
END [ PROGRAM Name ]
```

2.1 Einführendes Beispiel

Aufgabe:
Es ist ein Programm zu entwickeln, das zwei ganze Zahlen liest und die Summe berechnet und ausgibt.

```
! k2b1.for  Bild 2-1: Einfuehrendes Beispiel
      PROGRAM Beispiel                          ! Programmanfang
      INTEGER a, b, c                           ! Variablen vereinbaren
      PRINT *, 'Bitte zwei ganze Zahlen -> '    ! Meldung ausgeben
      READ *, a, b                              ! Zwei Zahlen lesen
      c = a + b                                 ! Summe berechnen
      PRINT *, 'Summe =', c                     ! Summe ausgeben
      PRINT *, 'Weiter -> '; READ *             ! Warten auf Taste
      END PROGRAM Beispiel                      ! Programmende
```

Bild 2-1: Einführendes Programmbeispiel

Die erste Zeile des in *Bild 2-1* dargestellten Programms enthält einen *Kommentar*. Er beginnt mit dem Ausrufezeichen `!` als Marke in Spalte 1 der Eingabezeile. Der Kommentartext enthält Erklärungen für den Programmierer bzw. für den Leser; er wird vom Compiler bei der Übersetzung nicht beachtet. Die Bezeichnung `k2b1.for` ist der Name der Quelltextdatei im Inhaltsverzeichnis des Betriebssystems und bedeutet *Programm zum Kapitel 2 Bild 1* mit der Erweiterung `for` für Fortran. Die folgenden Eingabezeilen enthalten ebenfalls Kommentare, die von der Marke `!` bis zum Ende der Eingabezeile reichen.

Die zweite Zeile beginnt in der gewählten Eingabeform erst ab Spalte 7. Das Kennwort PROGRAM kennzeichnet den Anfang des Hauptprogramms. Der hinter einem Leerzeichen stehende Programmname `Beispiel` wurde frei gewählt.

Die dritte Zeile enthält hinter dem Kennwort INTEGER eine Liste mit den Namen von drei Variablen, die später ganzzahlige Werte aufnehmen sollen.

Die vierte Zeile gibt hinter dem Kennwort PRINT * eine Meldung aus. Zur Laufzeit des Programms erscheint auf dem Bildschirm der zwischen den Hochkommazeichen stehende Text `Bitte zwei ganze Zahlen ->`.

Die Anweisung READ * der fünften Zeile verlangt die Eingabe von zwei Zahlen, die in den Speicherstellen `a` und `b` abgelegt werden. Zur Laufzeit des Programms muß der Bediener über die Tastatur zwei durch mindestens ein Leerzeichen getrennte Zahlenwerte eingeben. Die Eingabezeile ist durch einen *Wagenrücklauf* abzuschließen.

Die arithmetische Anweisung der sechsten Zeile berechnet die Summe der beiden Werte und legt sie in der Speicherstelle `c` ab.

In der siebenten Zeile werden der Text `Summe =` und der Inhalt der Speicherstelle `c` als Zahlenwert ausgegeben.

Die letzte Zeile beendet mit dem Kennwort END PROGRAM und der gewählten Programmbezeichnung `Beispiel` den Quelltext. Zur Laufzeit kehrt das Programm an dieser Stelle in das aufrufende System zurück, also in die Entwicklungsumgebung oder in das Betriebssystem.

Bei der Entwicklung des Programms gibt der Benutzer den Quelltext (Bild 2-1) zunächst mit einem *Editor* (Textverarbeitungsprogramm) in den Rechner ein. Dann folgen die Übersetzung durch den *Compiler* und die Verbindung mit Hilfsprogrammen durch den *Linker* (Binder). Werden dabei Fehler gemeldet, so sind diese mit dem Editor im Quelltext zu korrigieren. Nur fehlerfrei übersetzte und gebundene Programme lassen sich ausführen. *Bild 2-2* zeigt das Ergebnis zweier Testläufe, die aus dem Betriebssystem DOS gestartet wurden. Eingaben des Benutzers wurden unterstrichen.

```
D:\BSP>k2b1.exe
Bitte zwei ganze Zahlen -> 1 2
Summe =           3

D:\BSP>k2b1.exe
Bitte zwei ganze Zahlen -> 1000000000 2000000000
Summe = -1294967296

D:\BSP>
```

Bild 2-2: Ergebnisse der Testläufe

In dem ersten Testlauf wurde richtig `1 + 2 = 3` gerechnet. Der zweite Test lieferte ein unerwartetes Ergebnis: zwei positive Zahlen können keine negative Summe ergeben! Dieser Fehler entsteht durch einen Zahlenüberlauf im Rechenwerk und läßt sich nur schwer abfangen. Er zeigt, daß sich auch Computer irren können und daß man alle Ergebnisse kritisch beurteilen sollte!

```
K2B1      BAK          635  24.11.95   16:59
K2B1      FOR          635  24.11.95   17:20
K2B1      OBJ        1.363  24.11.95   17:20
K2B1      MAP       17.758  24.11.95   17:21
K2B1      EXE       60.544  24.11.95   17:21
```

Bild 2-3: Die Dateien des einführenden Beispiels (DOS)

Das einführende Beispiel wurde mit einem Fortran 90 Compiler unter den Betriebssystemen DOS und Windows getestet. *Bild 2-3* zeigt die Dateien des Beispielprogramms. Die Dateibezeichner bestehen aus einem maximal acht Zeichen langen Namen (z.B. `K2B1`) und einer aus maximal drei Zeichen bestehenden Erweiterung. Die Datei `K2B1.FOR` enthält den mit einem Editor aufgebauten Quelltext des Bildes 2-1. Die Datei `K2B1.BAK` (Backup) ist eine alte Sicherungsdatei. Bei der Übersetzung durch den Compiler entsteht die Datei `K2B1.OBJ` (Object) mit dem Maschinencode. Der Binder baut daraus das ablauffähige Programm `K2B1.EXE` (executable). Die Datei `K2B1.MAP` enthält eine Liste der gebundenen Programme. Der Benutzer gibt die Kommandos und Dateinamen des Betriebssystems wahlweise mit großen oder kleinen Buchstaben ein; die Ausgabe erfolgt wie in Bild 2-3 immer groß.

2.2 Eingabevorschriften

Der ***Zeichensatz*** für die Eingabe des Quelltextes besteht aus
- den großen Buchstaben von `A` bis `Z`,
- den kleinen Buchstaben von `a` bis `z`,
- den Ziffern von `0` bis `9`,
- Sonderzeichen wie z.B. `= + - * / ( ) , . ' ! " % & ; < > _` sowie
- Steuerzeichen wie z.B. *Wagenrücklauf* (neue Zeile).

Benutzerdefinierte Bezeichner dürfen keine deutschen Umlaute (Ä, Ö, Ü, ä, ö und ü) sowie das deutsche Zeichen ß enthalten. Von ihrer Verwendung in Kommentaren und Textkonstanten ist abzuraten, da bei der Übertragung des Textes auf ein anderes System unerwarte Zeichen auf der Ausgabe erscheinen können. Der Anhang enthält die Codetabelle des ASCII Zeichensatzes (DOS). Kleine Buchstaben haben in Bezeichnern die gleiche Bedeutung wie die entsprechenden großen Buchstaben. In den Beispielen dieses Buches werden alle vordefinierten Bezeichner (Kennwörter, Schlüsselwörter) groß geschrieben, benutzerdefinierte wie z.B. Variablennamen dagegen klein. Beispiele:

```
INTEGER  laenge
READ *, laenge
```

Anweisungsmarken bestehen aus vorzeichenlosen ganzen Zahlen im Bereich von 1 bis 99999. Sie kennzeichnen Anweisungen bzw. Formatbeschreibungen, auf die sich eine andere Anweisung bezieht. Jede Marke darf in einem Programmtext nur einmal enthalten sein.

Kommentare sind Bemerkungen und Erläuterungen zum Programm, die vom Compiler weder geprüft noch übersetzt werden.

Die ***spaltenorientierte*** Eingabeform des Programmtextes (Fortran 77) stammt noch aus der Zeit der Lochkarte. Sie gilt auch für die moderne Bildschirmeingabe:
- Kommentare erhalten ein `C` oder `*` in Spalte 1 und reichen bis zum Zeilenende.
- Anweisungsmarken stehen in den Spalten 1 bis 5 der Eingabezeile.
- Ein Zeichen außer `0` in Spalte 6 kennzeichnet Fortsetzungszeilen (maximal 19).
- Kommentare lassen sich nicht fortsetzen, sondern erfordern eine neue Zeile.
- Die Fortran Anweisungen stehen in den Spalten 7 bis 72.
- Die Spalten 73 bis 80 werden wie Kommentare vom Compiler nicht ausgewertet.

Die meisten Fortran 77 Compiler lassen folgende *Erweiterungen* zu:
- Das Ausrufezeichen `!` leitet ebenfalls Kommentare ein.
- Mehrere Anweisungen auf einer Zeile sind durch ein Semikolon `;` zu trennen.

Die ***freie Eingabeform*** (Fortran 90) kennt keine besondere Spalteneinteilung mehr.
- Es sind maximal 132 Zeichen auf einer Zeile zulässig.
- Das Ausrufezeichen `!` leitet Kommentare bis zum Zeilenende ein.
- Mehrere Anweisungen auf einer Zeile sind durch ein Semikolon `;` zu trennen.
- Das Zeichen `&` als letztes Zeichen kennzeichnet, daß der Text auf der *folgenden* Zeile fortgesetzt wird.
- Es sind maximal 39 Fortsetzungszeilen möglich.
- Die Fortsetzungszeile *kann* mit einem Zeichen `&` am Anfang gekennzeichnet werden.
- Kommentare lassen sich nicht fortsetzen.
- Leerzeichen dürfen nicht in Bezeichner eingefügt werden.
- Bei einigen vordefinierten Bezeichnern sind zwei Schreibweisen möglich, wie z.B. `END PROGRAM` oder `ENDPROGRAM`.

```
! k2b4.for  Bild 2-4: Allgemeine Eingabeform
! Kommentarzeile durch Ausrufungszeichen eingeleitet
      IMPLICIT NONE      ! Kommentar hinter Anweisung
      INTEGER a, b, c    ! Anweisung ab Spalte 7
      a = 1; b = 2       ! Anweisungen durch Semikolon getrennt
                         ! die folgende Zeile wird fortgesetzt
      c =                                                    &
     &a + b              ! & in Spalte 6 kennzeichnet Fortsetzung
      WRITE(*,100) c     ! formatierte Ausgabe statt PRINT *, c
100   FORMAT(' c =',I5)  ! markiertes Ausgabeformat in Spalte 1 bis 3
      PRINT *, 'Weiter -> '; READ *            ! 2 Anweisungen
      END                ! Endemarke des Programms ohne PROGRAM
```

Bild 2-4: Allgemeines Eingabeformat