Speicherung und Verarbeitung von Daten im Computer

Der Computer kennt nur die Ziffern (Bits) 0 und 1; traditionell wird eine Kette von 8 Bits als ein Byte bezeichnet. Alle Daten müssen zunächst in die Binärform umgewandelt werden und werden in dieser Form vom Computer verarbeitet.

Im Folgenden soll untersucht werden, wie dies für die in der Sprache Java üblichen Datentypen *int, double, boolean* und *char* (sowie *String*) geschieht.

1. Der Datentyp *int* (Ganzzahl)

1.1. Darstellung

```
   0000 0000  0000 0000  0000 0000  0000 0000 =           0
   0000 0000  0000 0000  0000 0000  0000 0001 =           1
   0000 0000  0000 0000  0000 0000  0000 0010 =           2
   0000 0000  0000 0000  0000 0000  0000 0011 =           3
    ....
   0111 1111  1111 1111  1111 1111  1111 1110 =  2 147 483 646
   0111 1111  1111 1111  1111 1111  1111 1111 =  2 147 483 647
   1000 0000  0000 0000  0000 0000  0000 0000 = -2 147 483 648
   1000 0000  0000 0000  0000 0000  0000 0001 = -2 147 483 647
   1000 0000  0000 0000  0000 0000  0000 0010 = -2 147 483 646
    ....
   1111 1111  1111 1111  1111 1111  1111 1101 =          -3
   1111 1111  1111 1111  1111 1111  1111 1110 =          -2
   1111 1111  1111 1111  1111 1111  1111 1111 =          -1
(1)0000 0000  0000 0000  0000 0000  0000 0000 =           0
```

Normalerweise werden in Java für eine Ganzzahl 4 Byte Speicherplatz reserviert. Damit sind die Zahlen 0 .. 4 294 967 295 dual darstellbar, die entweder nur als positive Zahlen oder -- wie üblich -- als "positive" Zahlen von 0 .. 2 147 483 647 und als negative Zahlen von -2 147 483 648 .. -1 (Zweierkomplement) aufgefasst werden.

Probiere
```
public void zeigeGanzzahlÜberlauf()
{
  for (int i=0; i<5; i++)
  {
    System.out.println (2147483646+i);
  }
}
```
(Die kleinste und größte Ganzzahl vom Typ *int* sind in Java übrigens in den Konstanten Integer.MIN_VALUE = -2 147 483 648 bzw. Integer.MAX_VALUE = 2 147 483 647 abgelegt. Außer dem Bereich *int* gibts's auch die 2-Byte-Ganzzahlen *short* -32 768 .. 32 767 und den 1-Byte-Typ *byte* -128 .. 128).

a) Nichtnegative Zahlen:
a1) Da jede Stelle der Dualzahl einer Zweierpotenz entspricht, lässt sich eine Dualzahl leicht in eine Dezimalzahl verwandeln: $0110\ 1001 = 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 64 + 32 + 8 + 1 = 105$.

a2) Das Verfahren lässt sich auch in der umgekehrten Richtung durchführen, um zu einer Dezimalzahl ≥ 0 die dazugehörige Dualzahl zu finden: man prüft einfach, welche Zweierpotenzen in der Dezimalzahl stecken!

Nichtnegative ganze Dezimalzahl (hier 0 .. 2^{15}) in Dualdarstellung umwandeln:

```
public String dez2dual (int dezimal)
  {
    String dual="";
    for (int exponent=15; exponent>=0; exponent--)
    {
      int stufenzahl = (int)Math.pow (2, exponent);
      if (dezimal >= stufenzahl)
        {
          dual = dual + "1";
          dezimal = dezimal - stufenzahl;
        }
      else
        {
          dual = dual + "0";
        }
     }
     return (dual);
  }
```

a3) Eine andere Möglichkeit, eine Dezimal- in eine Dualzahl zu verwandeln, bietet das Divisionsrest-Verfahren, das hier beispielhaft an der Umwandlung von 105 dargestellt werden soll:

```
    105 : 2  =  52  Rest 1, --------+
     52 : 2  =  26  Rest 0, --------+|
     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. -110 1001 dual (für 105 dezimal)
```

b) Negative Zahlen: Zunächst kann die Dualdarstellung der entsprechenden positiven Zahl wie in Abschnitt a) bestimmt werden, dann wird in dieser Dualzahl jede 0 durch eine 1 ersetzt und umgekehrt (d.h. alle Bits werden invertiert -> Einerkomplement) und schließlich wird noch die Zahl 1 hinzu addiert (-> Zweierkomplement).

Beispiel: Dualdarstellung von -5. Dualzahl zu +5 ist 0000 0101. Umkehren ergibt 1111 1010 und plus 1 macht daraus schließlich die endgültige Darstellung von -5, nämlich 1111 1011. (Nach der gleichen Methode wird aus -5 auch wieder +5: 1111 1011 --> 0000 0100 + 1 --> 0000 0101.)
An der vordersten Stelle (Bit ganz links) lässt sich leicht erkennen, ob die Dualzahl eine negative (1) oder eine nicht-negative Zahl (0) repräsentiert.

1.2. Addition und Subtraktion

Die Addition von Dualzahlen (egal, ob diese positive oder negative Dezimalwerte repräsentieren!) geschieht wie beim schriftlichen Addieren ziffernweise, beginnend mit der letzten Stelle: 0+0 gibt 0, 1+0 oder 0+1 gibt eine 1, 1+1 gibt 0 und einen Übertrag von 1, usw. Die Überträge über die vorhandene Stellenzahl hinaus gehen verloren!

Da die Subtraktion die Addition von negativen Zahlen ist, reicht das Additionsverfahren (und ein dafür zuständiges Addierwerk im Computer) für beide Operationen!

Beispiele für die Addition bzw. Subtraktion ganzer Zahlen:

```
  11        0000 1011                 11          0000 1011
 + 5      + 0000 0101               - 5       +   1111 1011
-----    -------------             -----    ----------------
 =16      = 0001 0000               = 6    = (1) 0000 0110
```

Und natürlich funktioniert das auch, wenn das Ergebnis der Addition bzw. Subtraktion negativ ist!

1.3. Multiplikation und ganzzahlige Division

Auch die Multiplikation zweier (Dual-)Zahlen lässt sich auf die Addition zurückführen. Wieder dient das schriftliche Verfahren bei Dezimalzahlen als Vorbild:

Beispiel:

```
  17 * 203
      34
+      0
+     51
-----------
=    3451
```

Genauso z.B. 10 * 5 =

```
1010 * 0101
     0
+   1010
+      0
+    1010
-------------
=   110010   = 50
```

Hier ist -- außer dem Verschieben der Zahlen nach rechts -- nur die Addition nötig. Da als Dualziffern nur 0 und 1 auftreten, ist (anders als beim normalen schriftlichen Multiplizieren) auch nicht die Kenntnis des Kleinen Einmaleins notwendig! Bei negativen Faktoren kann es hingegen Probleme geben, die aber vermeidbar sind, wenn man nur die Beträge miteinander multipliziert und dem Produkt nachher das richtige Vorzeichen gibt. Dabei kann das Ergebnisvorzeichen leicht durch einstellige, übertragslose Addition der vordersten Bits der beiden Faktoren bestimmt werden: 0+1 = 1+0 = 1 (entspricht +•– = – •+ = –) und 0+0 =1+1= 0 (entspricht +•+ = –•– = +).

Das Dividieren von zwei ganzen Zahlen (das natürlich nicht immer ohne Rest gelingt) ist genauso einfach, z.B. 100 : 8 =

```
   01100100  :  01000 = 1100  Rest 0100  (dezimal 12 Rest 4)
 + 11000
(1)001001      Hier zeigt die erste 1 im Ergebnis, dass 01000
 + 11000       (=8) einmal in 01100 steckt. Statt dann 01000
 (1)000010     von 01100 abzuziehen, wird das 2er-Komplement
     + 0       11000 (=-8) addiert. Die zweite 1 zeigt, dass
       100     01000 in 1001 steckt; die Nullen folgen, weil
      + 0      1000 weder in 10 noch in 100 steckt.
       100     Diese letzte 100 (=4) bleibt als Rest übrig.
```

In Java liefert (int)100/8 das ganzzahlige Ergebnis 12, der Rest 4 kann mit 100%8 erfragt werden. Bei der Dualoperation ist wieder nur Verschieben sowie die Addition (hier des Zweierkomplements des Divisors) nötig, so dass sämtliche Grundrechenarten mit Ganzzahlen mit Hilfe des Computer-Addierwerks erledigt werden können.

2. Der Datentyp *double* (Gleitkommazahl)

2.1. Darstellung

Die Umwandlung von Dezimalbrüchen in eine entsprechende Dualkommazahl gelingt ebenfalls mit dem in 1.1. a1) vorgestellten Algorithmus, sofern dort auch Zweierpotenzen mit negativen Exponenten zugelassen werden. Die Anzahl der erlaubten Kommastellen wird vom zur Verfügung gestellten Speicherplatz bestimmt (in Java erlaubt der übliche Typ *double* Dezimalwerte bis $\pm 1{,}79 \cdot 10^{308}$ mit etwa 16 Dezimalstellen Genauigkeit; die kleinste von Null verschiedene Zahl ist $4{,}9 \cdot 10^{-324}$. *float* erlaubt nur etwa 9 Dezimalstellen Genauigkeit, Werte bis $\pm 3{,}4 \cdot 10^{38}$ und als nächsten Wert nahe Null $1{,}4 \cdot 10^{-45}$). Ungünstig auf die Rechengenauigkeit wirkt sich aber in jedem Fall aus, dass viele endliche Dezimalbrüche keine endliche Dualbruchdarstellung haben, also selbst bei vielen Stellen nie genau dargestellt werden können.

Beispiel: Dualdarstellung des Dezimalbruchs 0.1 (1 Zehntel). Wendet man (z.B. mit `for (int exponent=1; exponent>-5; exponent--)`) den in 1.1.a2) angegebenen Algorithmus an, so erhält man folgende Wertebelegungstabelle

Zahl (=dezimal):	0.1	0.1	0.1	0.1	0.1	0.1	0.0375
Stelle(=exponent):	1	0	-1	-2	-3	-4	-5
$2^{exponent}$:	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
stufenzahl:	2	1	0.5	0.25	0.125	0.0625	0.03125
Dualzahl (=dual):	0	0 .	0	0	0	1	1

Zieht man auch jetzt wieder 2^{-5}, also 0.03125, von der verbliebenen Zahl 0.0375 ab, so erhält man 0.00625, das aber 2^{-6} (= 1/64 = 0.015625) und 2^{-7} (= 1/128 = 0.0078125) nicht enthält, aber wieder 2^{-8} sowie 2^{-9} usw. Tatsächlich ist die Dualdarstellung von 0.1 der periodische Dualbruch 0.0 0011 0011 0011 0011.., der im Computer irgendwo abgeschnitten und damit ungenau werden muss. (Die Periode sieht man leichter bei Ausführung der Dual-Division 01 : 01010)

Die verheerende Auswirkung kann man an einem kurzen Java-Programmstück studieren, das nie aufhört, obwohl man eigentlich erwartet, dass beim zehnten Durchlauf die 1.0 erreicht wird:

```
double x = 0.0;
while (x != 1.0)
{
  x = x + 0.1;
  System.out.println (x);
}
```

Es liefert als Ergebnisse 0.1 0.2 0.30000000000000004 0.4 0.5 0.6 0.7 0.7999999999999999 0.8999999999999999 0.9999999999999999 1.0999999999999999 1.2 1.3 1.4000000000000001 ... und endet nie! Statt == oder != sollte besser x <= 1.0 verwendet werden.

Exaktheit ist nur möglich, wenn die Dezimalbrüche nicht in Dualbrüche verwandelt werden, sondern auch im Rechner die Dezimalziffern einzeln abgespeichert werden (wobei natürlich jede Dezimalziffer für sich binär durch eine 4-Bit-Folge dargestellt werden muss, was zum Namen BCD-{=binär-codierte-Dezimalziffern}-Darstellung geführt hat), und durch geeignete Algorithmen das Dezimalrechnen imitiert wird. Eigentlich reicht sogar wieder die normale Dualzahl-Addition, wenn man dabei von rechts beginnend zu Ergebnisziffern, die einen Übertrag zur nächsten 4-Bitgruppe verursachen, sowie zu ungültigen Ziffern (> 1001=9) die Dualdarstellung

0110 der 6 unter Berücksichtigung des Übertrags hinzuaddiert. BCD-Codierung und -Rechnung war z.B. für das Bankwesen vorgeschrieben, damit auch 10-Cent- und Centbeträge korrekt behandelt werden!

2.2. Grundrechenarten

Addition und Subtraktion können praktisch wie bei Ganzzahlen ausgeführt werden, wenn man das Komma nur vorher geeignet verschiebt:

Beisp.: dezimal (E=10er-Exponent) dual (D=2er-Exponent)

```
  0.1250 E+0      12500. E-5          0.10 D-2     100. D-5
+ 0.9375 E-1    +  9375. E-5          0.11 D-3   +  11. D-5
                ------------                     ----------
                = 21875. E-5                     = 111. D-5
             (= 0.21875  E+0)     (= 0.111 D-2 = 0.111 • 2^-2)
```

Und die Multiplikation wird -- genau wie beim schriftlichen Rechnen mit Dezimalzahlen -- ohnehin mit Ganzzahlen durchgeführt und erst anschließend wird das Komma an die richtige Stelle gesetzt. Bei der Division schließlich wird das Komma gesetzt, wenn beim Dividenden Stellen nach dem Komma "heruntergeholt" werden müssen -- z.B. eine weitere Null wie beim Beispiel 100/8 aus 1.3. (was zum Dualergebnis 100.1 führt). Durch Erweitern von Dividend und Divisor muss natürlich immer zuvor der Divisor ganzzahlig gemacht werden!

2.3. Höhere Rechenarten

Für *double*-Zahlen stehen in Java auch viele mathematische Funktionen zur Verfügung -- wie z.B. Math.sqrt (Wurzel) oder Math.sin (Sinus).

Der nachfolgende Algorithmus benutzt nur die Grundrechenarten, um in wenigen Schleifen-Durchläufen die (Quadrat-)Wurzel aus einem Radikand zu berechnen:

```
public double heron (double radikand) // Wurzel-Berechnung nach Heron
  {
    double t1;
    double t2 = 7; // beliebiger Startwert
    do {
      t1 = t2;
      t2 = t1/2 + radikand/(2*t1);
    } while (Math.abs(t1-t2) > 0.0001); // gewünschte Genauigkeit
    return (t2); // Näherungswert für die Wurzel
  }
```

Andere Funktionen können leicht mit Hilfe ihrer Taylorschen Potenzreihe bestimmt werden, wie z.B. der Sinus einer Zahl x (x im Bogenmaß):

sin(x) := x/1 - x*x*x/(1*2*3) + x*x*x*x*x/(1*2*3*4*5)
- x*x*x*x*x*x*x/(1*2*3*4*5*6*7) + - ...,

wofür ebenfalls nur die Grundrechenarten benötigt werden.

Abschließend sei festgehalten, dass sich sämtliche Rechnungen mit Zahlen auf die Grundrechen-

arten, damit auf die entsprechenden Ganzzahloperationen und somit letzlich aufs Addieren sowie das Verschieben und -- zusätzlich für die Komplementbildung -- aufs ziffernweise Invertieren (1 wird 0, 0 wird 1) zurückführen lassen. Für alles Rechnen braucht ein Computer also lediglich ein Addierwerk, ein Schieberegister und einen Inverter!

3. Der Datentyp *boolean* (Wahrheitswert)

3.1. Darstellung

Da es nur zwei verschiedene Wahrheitswerte, false und true, gibt, wäre zur Codierung nur 1 Bit mit den beiden Möglichkeiten 0 und 1 erforderlich. Weil der Speicherplatz des Computers aber i.a. byteweise organisiert ist, wird meist trotzdem ein ganzes Byte zur Darstellung genommen. Während in Pascal false (falsch) = 0 = 0000 0000 und true (wahr) = 1 = 0000 0001 benutzt wird und dadurch auch die Reihenfolge false < true festgelegt ist, wird in Basic meist anders codiert, z.B. ist in Microsoft-Basic wahr durch -1 = 1111 1111 repäsentiert (und falsch weiterhin durch 0 = 0000 0000), woraus sich dort false > true ergibt. In Java ist die Implementation hingegen nicht so leicht festzustellen, da hier Wahrheitswerte nicht (z.B. mit (int)true) als Zahl ausgegeben und auch nicht mit < verglichen werden können. Man muss sie sich einfach als 0 (false) und 1 (true) vorstellen.

3.2. Logische Verknüpfungen

In Java stehen die beiden 2-stelligen Verknüpfungen && und **| |** sowie die 1-stellige Operation ! zur Verfügung. Für true=1 und false=0 brauchen diese Operationen nur auf ein Bit zu wirken. Bei ! (not) wird das Bit mit dem Inverter umgewandelt; auch für && (and) und | | (or) stehen im Computer hardwaremäßig spezielle Schaltungen ("Gatter") bereit. (Zweistellig: es werden zwei Wahrheitswerte a und b verknüpft wie bei x = a && b. Einstellig: der Operator bezieht sich nur auf einen Wahrheitswert wie bei x = ! a)

4. Der Datentyp *char* (Schriftzeichen)

Die Schriftzeichen sind in einer Tabelle abgelegt und durchnummeriert. Im Rechner wird an Stelle des Schriftzeichens seine Nummer abgelegt – und zwar bei ‘normalen’ Buchstaben und Zeichen in einem Byte die Nummer 0..127 des Zeichens im ASCII-bzw. ANSI-Code. Die Nummern 0..31 sind dort für nicht ausdruckbare Zeichen (z.B. für die Return- und die Löschtaste) reserviert; Ziffern, Klammern und die Schriftzeichen des ‘normalen’ Alphabets sind mit 32..127 nummeriert. Der Buchstabe 'A' ist z.B. das Zeichen Nr. 65 und wird entsprechend als 0100 0001 im Computer dargestellt, während 'a' das Zeichen Nr. 97 ist und damit als 0110 0001 dargestellt wird. Die Zeichen Nr. 128 ..255 waren schon in der ASCII-Tabelle nicht verbindlich genormt und wurden von verschiedenen Computerherstellen mit unterschiedlichen Zeichen und Funktionen belegt. Hier unterscheiden sich dann auch ASCII-, ANSI- und die Unicode-Tabelle, weswegen die Umlaute von Dos-Texten (ASCII) in Windows-Programmen (ANSI) falsch dargestellt werden und umgekehrt. Und die von Java verwendete Unicode-Codierung der Umlaute und nationalen Sonderzeichen ist nochmal anders und geschieht im Allgemeinen in zwei Bytes, wie man bei der Analyse einer class-Datei mit String-Konstanten im Hex-Editor sehen

kann! Ein kurzes Programmstück gibt Auskunft über die aktuelle Belegung. In der Konsole (Dos-Box) wird evtl. der ASCII-Zeichensatz benutzt (oder der ANSI-Zeichensatz mit der voreingestellten Codepage 850; Codepage 437 entspricht hingegen ASCII), nicht der Uni-Code, der von Java für Sonderzeichen verwendet wird – deswegen die Beschränkung auf i<128 statt i<65535 (wie sie theoretisch mit 2 Bytes zu erreichen wäre):

```
for (int i=32; i<128; i++)
  {
    char zeichen = (char)i;
    System.out.print (i+"="+zeichen+" ");
  }
```

Da es keine Rechnungen mit Schriftzeichen gibt, müssen dafür auch keine Vorkehrungen getroffen werden. Die Reihenfolge der Buchstaben wird durch ihre Nummern festgelegt (die man in Java mit der Umwandlung `nummer = (int)buchstabe` erfragen kann – so liefert beispielsweise `System.out.println ((int)'A');` die Nummer 65 und `(int)'1'` ist 49).

5. *String*-Objekte (Zeichenketten)

Ergänzend sei hier noch auf den Java-Typ *String* hingewiesen, der im Wesentlichen mehrere Schriftzeichen hintereinander speichert. Allerdings ist die Länge nicht festgelegt, sondern richtet sich nach dem Inhalt. Nach dem letzten gültigen Zeichen wird das nicht druckbare Zeichen Nr. 0 als Endmarkierung angehängt. Die Zeichenkette "Test" würde also (wegen T=84, e=101, s=115 und t=116) von Java kodiert als 0101 0100 0110 0101 0111 0011 0111 0100 0000 0000 , wobei im Speicher natürlich alle Bits direkt hintereinander stehen und nicht – wie hier – der Übersichtlichkeit halber durch Leerzeichen getrennt sind

6. Weitere Datentypen, Allgemeines, Programmsteuerung

Alle weiteren (zusammengesetzten und benutzerdefinierten) Datentypen lassen sich auf die vier oben behandelten Grundtypen zurückführen und werden auch im Computer entsprechend dargestellt und ggf. verknüpft.
Da alle unterschiedlichen Datentypen im Computer aber immer in Binärform vorliegen, muss sich die Speicherplatzverwaltung merken, ob z.B. 0100 0001 jetzt die Zahl 65 oder das Zeichen 'A' oder einen bestimmten Teil einer Komma-Zahl oder gar ein bestimmtes Java-Schlüsselwort im Programmtext darstellt. Denn auch das Programm selbst wird natürlich binär codiert, der Java-Programmtext im Editor wird zeichenweise nach der ASCII/Unicode-Tabelle abgespeichert (In Basic-Umgebungen werden die Basic-Schlüsselworte und -befehle oft intern sofort in 2-Byte-Nummern übersetzt und so entsprechend Platz sparender gesichert). Die Übersetzung von der Hochsprache in die Maschinensprache erfolgt bei Java in zwei Stufen: Zunächst wird der Quelltext (Datei-Endung .java) in einen Java-Bytecode (Endung .class) für die so genannte Virtuelle Maschine übersetzt (der natürlich auch nur als Folge von Nullen und Einsen gespeichert wird). Der Bytecode ist aber noch unabhängig vom verwendeten Computertyp. Erst die Virtuelle Maschine erzeugt daraus zur Laufzeit entsprechend dem verwendeten Prozessor die zugehörige prozessorspezifische Maschinensprache. Diese besteht dabei aus den Nummern, die den einzelnen Maschinenbefehlen zugeordnet sind („OpCode“), sowie den „Adressen“ (=Speichernummern) der Variablen – beides natürlich als Dualzahlen kodiert. Ein einzelner Java-Befehl wird

im Allgemeinen den Aufruf mehrerer Maschinenbefehle erfordern. Java-Interpreter und Virtuelle Maschine übersetzen jede Java-Anweisung in alle dafür nötigen Maschinenbefehle; die Befehlsnummern werden von der Virtuellen Maschine (die bei der Installation eines Java Development Kits / Java Software Kits oder eines javafähigen Web-Browsers auf dem Computer installiert wurde) in einer festen OpCode-Tabelle nachgesehen, während die Adress-Tabelle entsprechend der jeweiligen Variablendeklaration für jedes Programm eigens erstellt wird (Näheres über die Maschinensprache später!).

Nachdem nun grundsätzlich klar ist, wie ein Computer ausschließlich mit 0 und 1 arbeiten kann, und auch komplizierte Anwendungen nur einige Gatter (für logische Verknüpfungen) bzw. im Wesentlichen ein Addierwerk (zum Rechnen) erfordern, soll im folgenden Anhang B die dafür benötigte Hardware, d.h. der physikalisch-technische Aufbau der Gatter, des Rechenwerks und des Speichers besprochen werden, um so den Computer ganz in Funktion und Aufbau zu durchschauen und ihn dadurch zu entmystifizieren.

Aufgaben und Übungen zur Computerarithmetik

1. Dualdarstellung
   a) In Fernschreibern stehen nur die 26 Großbuchstaben 'A' bis 'Z' und die Ziffern 0 bis 9 zur Verfügung. Satz- und Sonderzeichen außer dem Leerzeichen gibt's nicht („ANKOMME MORGEN 15 UHR 10 STOPP“). Wie viele Bit sind zur Dualcodierung nötig?
   b) Gib dezimal den Ganzzahlbereich an, der sich mit Dualzahlen der Länge
      b1) 3 Bit b2) 13 Bit b3) n Bit
      darstellen läßt, wenn (1) nur positive oder (2) neg. und positive Zahlen codiert werden sollen.
   c) Ist das Morsealphabet ein binärer Code?

2. Fünfer-System: Im Land der Einhändigen hat sich das Fünfersystem durchgesetzt.
   a) Welche Ziffern gibt es (im Fünfersystem ist die Basis=5) ?
   b) Übersetze ins Dezimalsystem $(4301)_5$, $(2331042)_5$ bzw. ins Fünfersystem 378 u. 47 !
   c) Schreibe ein Java-Programm für eine der Übersetzungsrichtungen in b). Verwende dabei die fertige Methode Math.pow (z.B. double potenz = Math.pow (3.5, 4.12) liefert $3{,}5^{4{,}12} \approx 174{,}41$).
   d) Welcher positive Zahlbereich ist mit einer 4-stelligen Fünferzahl abzudecken?
   e) Welcher negative und positive Zahlbereich ist mit 3-stelligen Fünferzahlen abzudecken? Wie würden vermutlich -1 und -2 im Fünfersystem lauten? Warum?
   f) Stelle dir die Multiplikation zweier (positiver) Fünfer-Zahlen vor. Welche Operationen sind zusätzlich zur Verschiebung und Addition von Fünferzahlen nötig?

3. Rechnen mit Dualzahlen
   a) 5-Bit-Dualzahlen repräsentieren Dezimalzahlen von -16 bis +15. Führe die angegeben Rechnungen mit genau 5 Bit aus und notiere zusätzlich, ob das Ergebnis richtig ist (also nicht etwa durch einen Übertrag in die vorderste Stelle das falsche Vorzeichen erhält). Verwende ggf. das Zweierkomplement und übersetze nur die 5-Bit-Ergebnisse in Dezimalzahlen.
      a1) 01101 + 01011
      a2) 00010 - 00101
      a3) 01001 - 10110
      a4) 00101 * 01101
      a5) 00011 / 01000 (mit allen Nachkommstellen)

   b) In Java gibt es u.a. die Funktion Math.exp(x), für die laut Formelsammlung gilt:

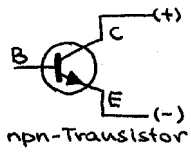
$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots . \qquad wobei\ z.B.\ 3! = 1 \cdot 2 \cdot 3\ ist\ .$$

      Berechne (dezimal) exp(0,4) auf 3 Stellen genau. Woran kann der Computer merken, ob er aufhören darf?
   c) Die Formel in b) lässt Schwierigkeiten für x > 1 befürchten. Welche/warum? Berechne z.B. das 19. Glied der Reihe für x = 1,7 sowie für x = 83.
   d) Welche Hardware ist nötig, um einem Dual-Computer sämtliche Rechenarten für alle Zahlentypen zu ermöglichen?

   später
   e) Verwandele in BCD-Zahlen und berechne 4,28 + 6,56 sowie 0,234 + 2,87
   f) Warum wird bei der BCD-Addition zur Dezimalanpassung „0110" addiert?

Aufbau der Computer-Hardware

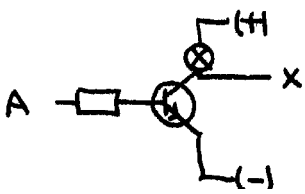
In heutigen Computern werden die beiden Dualziffern (Bits) 0 und 1 durch elektrische Spannungen dargestellt: 1 = +5 Volt (neuerdings auch oft nur ca. +3 Volt) = Verbindung mit dem Pluspol der Spannungsquelle; 0 = 0 Volt = Verbindung mit dem Minuspol der Quelle. Zwischen beiden Zuständen könnte mit einem mechanischen Schalter oder auch mit einem Relais hin- und her geschaltet werden; tatsächlich verwendet man inzwischen Transistoren (von den viele auf einem einzigen Chip eines IC [integrated circuit = zusammengefasste Schaltung] untergebracht sind). Nur, wenn in die Basis B eines npn-Transistors Strom fließt (was nur durch Anlegen einer [positiven] Spannung ermöglicht wird), ist die Collector-Emitter-Verbindung C-E leitend. Solange Spannung (d.h. eine logische Eins) an der Basis anliegt, wird praktisch der Schalter in der C-E-Strecke betätigt.



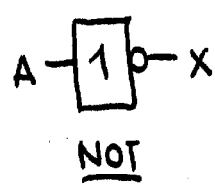
Logische Schaltungen (Gatter)

Im folgenden werden nur Prinzip-Schaltungen gezeigt; in Wirklichkeit werden aus fertigungstechnischen Gründen oder für höhere Geschwindigkeiten auch andere, aufwändigere Schaltungen benutzt.

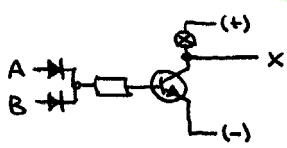
1) Inverter (NOT-Gatter): Wird am Eingang A Spannung (also eine Eins) angelegt, so wird die C-E-Strecke des Transistors leitend. Damit gibt es aber eine gute Verbindung des Ausgangs X mit dem Minuspol der Spannungsquelle, so dass die Spannung am Ausgang auf null sinkt, d.h. eine 0 aus dem Ausgang kommt. Liegt andererseits der Eingang auf 0, so bleibt die C-E-Verbindung unterbrochen und der Ausgang ist über den (kalten) Glühfaden der Lampe nur mit dem Pluspol verbunden: der Ausgang zeigt eine 1 (+5 V). Der Ausgang weist also genau das Gegenteil des Eingangs auf, womit wir bereits einen Inverter, also ein NOT-Gatter (Nicht-Glied) realisiert haben.



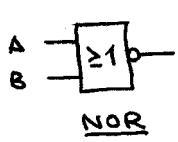
A	X	
0	1	Lampe aus
1	0	Lampe hell



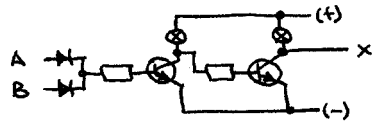
2) NOR- und OR-Gatter: Der ursprüngliche Eingang eines Inverters kann durch zusätzliche Dioden zu zwei (oder mehr) Eingängen erweitert werden. Nun reicht es, wenn an einem dieser Eingänge eine 1 ankommt (die anderen Eingänge dürfen ruhig mit 0 verbunden sein: ein Abfluss von Strom über einen solchen Eingang ist wegen der Dioden trotzdem nicht möglich!), um den Transistor leitend zu schalten und damit den Ausgang auf 0 zu setzen. Nur wenn alle Eingänge auf 0 sind, erscheint am Ausgang die 1. Damit hat die Schaltung genau die Eigenschaften eines NOR-Gatters (Nicht-Oder-Schaltung), wie der Vergleich von Schalt- und Wahrheitswerttabelle zeigen.



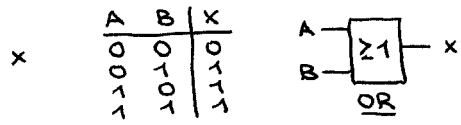
A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

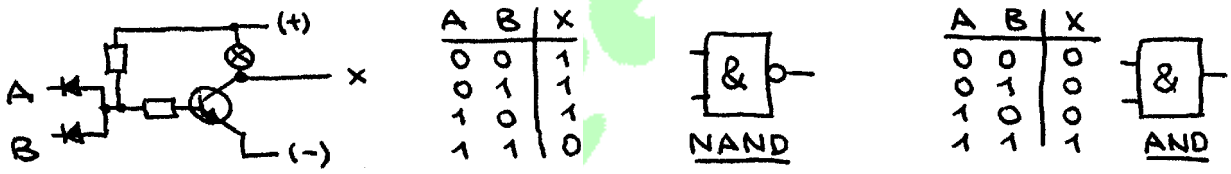


Ein "richtiges" Oder (OR-Gatter) erhält man durch Invertieren des NOR-Ausgangs, d.h. in dem man an den Ausgang des NOR-Gatters einfach einen oben beschriebenen Inverter anhängt.



3) NAND- bzw. AND-Gatter: Gegenüber dem NOR-Gatter sind hier die Anforderungen vertauscht. - Beim NAND-Gatter soll nämlich genau dann die 0 am Ausgang abgegriffen werden können, wenn alle Eingänge auf 1 sind. Nur dann darf die C-E-Strecke des Transistors leiten. Andernfalls muss also jeder 0-Eingang dafür sorgen, dass an der Basis keine Spannung anliegt und evtl. Strom zum Minuspol der Quelle abfließt. Das erzwingt einmal, dass die Eingangsdioden anders herum eingebaut werden müssen, und zum anderen, dass jetzt durch einen Widerstand die Basis B mit Strom versorgt wird, weil ein Stromfluss durch die Dioden in Sperrrichtung ja nicht mehr möglich ist. Das Bild zeigt das resultierende NAND-Gatter; wieder wird durch Nachschalten eines Inverters daraus ein AND-Gatter!

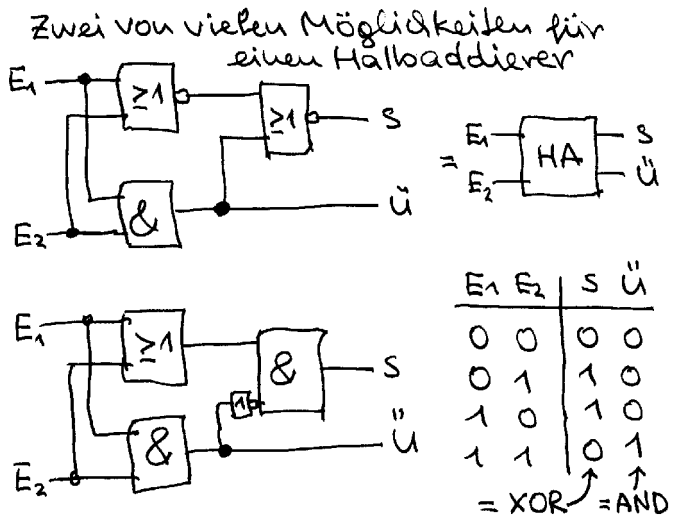




Die oben beschriebenen Schaltungen ermöglichen bereits sämtliche Operationen, die im Computer für den Typ boolean nötig sind. Für das Rechnen mit Zahlen (egal welchen Typs) wurde darüber hinaus nur noch ein Addierwerk gebraucht. Dieses Addierwerk soll im folgenden Abschnitt aus Logikgattern zusammengebaut werden.

Schaltnetze

Intuitiv können verschiedene Gatter zusammengefügt werden: so könnte z.B. ein XOR (=Entweder-Oder) bzw. ein Halbaddierer wie nebenstehend gebaut und aus zwei Halbaddierern und einem OR ein Volladdierer zusammengesetzt werden. Acht parallele Volladdierer bilden das Paralleladdierwerk zum Addieren zweier 8-Bit-Dualzahlen.



Für die Konstruktion beliebiger Schaltungen soll statt des intuitiven Verfahrens jedoch ein systematisches Vorgehen ermöglicht werden:

Disjunktive Normalform (DNF): Um kompliziertere Schaltnetze aufzubauen, werden zunächst die Anforderungen an das Bauteil in einer Wahrheitswerttabelle festgelegt. Eine Möglichkeit, eine Funktionsvorschrift für die zugrunde liegende Schaltfunktion aufzustellen, geht von folgender Überlegung aus: Findet man zu jeder 1 in der Ausgangs-(=Ergeb-

nis-)spalte der Wertetabelle einen Term, der genau für die in dieser Zeile angegebene Eingangsbelegung den Funktionswert 1 erzeugt, so braucht man diese sog. Minterme nur noch durch eine Oder-Verknüpfung (Disjunktion) zu verbinden, um die gesamte Funktionsvorschrift zu erhalten. Die einzelnen Minterme lassen sich ihrerseits einfach aus den mit Und verbundenen Werten der Eingänge (wenn dort eine 1 anliegt) bzw. der negierten Eingänge bilden (falls der Eingangswert 0 ist). Auf diese Weise findet man die Funktionsvorschrift in der disjunktiven Normalform (DNF), die mit den drei Logikoperationen Oder, Und und Nicht auskommt, für die bereits Hardware-Bausteine vorgestellt wurden.

Volladdierer: Sollen zwei mehrstellige Dualzahlen addiert werden, so braucht man für jede Stelle ein Bauteil mit drei Eingangs- und zwei Ausgangsleitungen. An zwei Eingängen liegen die beiden Ziffern der Summanden an, am dritten Eingang kommt ein eventueller Übertrag von früheren, niederwertigen Stellen hinzu. An einem Ausgang (S) erscheint die richtige Stelle des Ergebnisses, am anderen Ausgang (Ü) kommt ein Übertrag für die nächsthöhere Stelle heraus. Die abgebildeten Wahrheitswerttafeln beschreiben das geforderte Verhalten des Volladdierers für beide Ausgänge.

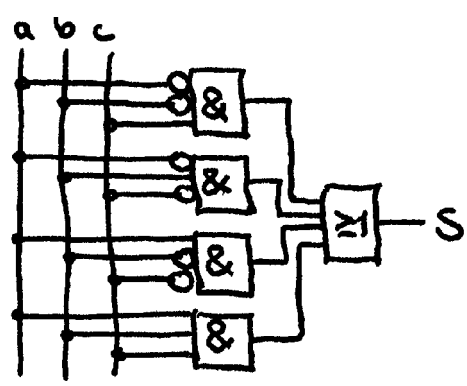
a	b	c	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

a	b	c	Ü
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Für diese beiden Ausgänge lässt sich dann nach dem oben beschriebenen Verfahren der Funktionsterm in DNF beschreiben:

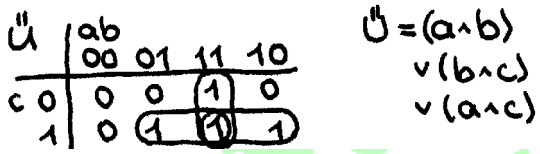
$$S(a,b,c) = (\bar{a} \wedge \bar{b} \wedge c) \vee (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge b \wedge c)$$
$$Ü(a,b,c) = (\bar{a} \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c}) \vee (a \wedge b \wedge c)$$

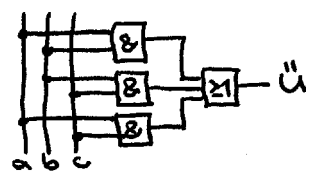
(wobei wie in der Mathematik üblich $\wedge$ = 'und' und $\vee$ = 'oder' bedeutet. In vielen Büchern wird leider das $\wedge$-Zeichen ganz weggelassen und das $\vee$-Zeichen durch ein '+' ersetzt). Für den Summenausgang S wird die DNF wie angegeben in Hardware übertragen. Bemerkenswert ist, dass die Realisierung der DNF immer nur zu höchstens dreistufigen Schaltnetzen führt (jedes Signal muss höchstens einen Inverter, ein Und-Glied und ein Oder-Gatter passieren; alle Inverter und Und-Gatter arbeiten parallel). Entsprechend sind solche Schaltungen recht schnell. Für den Übertrags-Ausgang Ü ist natürlich eine analoge Umsetzung möglich; allerdings können bei Ü noch Vereinfachungen durchgeführt werden.



Vereinfachung von Schaltnetzen nach Karnaugh und Veitch: In der DNF werden manchmal zwei Minterme aufgenommen, die sich nur in einer einzigen Eingangsbelegung unterscheiden. In solchen Fällen ist es aber gar nicht nötig, beide Minterme zu notieren (und durch zwei entsprechende Und-Gatter zu realisieren). Vielmehr hat die Änderung des Wertes an diesem Eingang offenbar keine Auswirkung auf das Ergebnis. Deshalb reicht ein Und-Term, in dem der entsprechende Eingangswert gar nicht mehr vorkommt; in der Hardware kann also ein Und-Gatter und die Beschaltung dieses Eingangs eingespart werden.
Um solche Einsparungsmöglichkeiten leichter zu finden, haben Karnaugh und Veitch eine graphische Methode vorgeschlagen. Statt in der üblichen Wahrheitswerttafel werden die Funktionswerte in einem tabellenförmigen Diagramm angeordnet, dessen Zeilen und Spalten mit den verschieden Eingangswert(kombination)en so beschriftet sind, dass sich benachbarte Spalten bzw. Zeilen immer nur um genau ein Bit in den Eingangswerten unterscheiden. Haben benachbarte Zellen den gleichen Funktionswert 1, so ist das unterschiedliche Eingangsbit für diesen Wert irrelevant. Aber nicht nur Paare benachbarter Einsen lassen sich so zusammenfassen, auch Vierer-, Achter-, Sechzehner-,.. -gruppen sind möglich, wenn mehrere Eingänge für einen Funktionswert 1 unerheblich sind. Dabei muss noch beachtet werden, dass auch gegenüberliegende Ränder des Diagramms als benachbart gelten. Außerdem kann eine Zelle mit einer 1 durchaus in mehreren Gruppen enthalten sein.

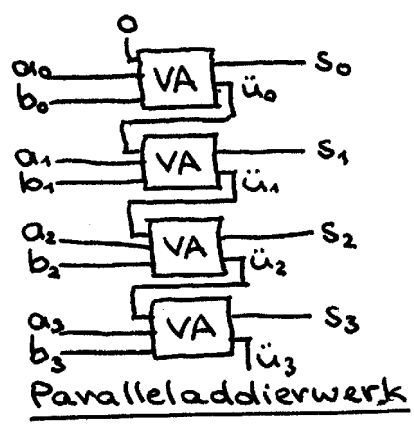
Hier wird beispielhaft das Karnaugh-Diagramm für den Übertrags-Ausgang Ü des Volladdierers angegeben; nur noch für jede Gruppe ist ein Und-Term nötig!





Der Hardware-Aufwand ist entsprechend geringer als bei der direkten Umsetzung des ursprünglichen DNF-Funktionsterms, wobei die vereinfachte Schaltung weiterhin höchstens dreistufig bleibt.
Die gesamte Hardware des Volladdieres wird zu einem Symbol "VA" (oder auch "Σ") zusammengefasst.

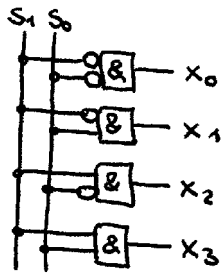
Aufgabe: *Normalerweise klingelt es bei Oma Rüstig, wenn jemand unten an der Haustür (a) oder an ihrer Wohnungstür (b) auf den Klingelknopf drückt. Um nicht im Mittagsschlaf gestört zu werden, hat sie sich einen zusätzlichen Schalter (c) einbauen lassen, mit dem sie die Klingel aus- (0) oder einschalten kann (1). Stelle die Wahrheitswerttabelle* a b c | Klingel *auf, notiere die DNF, vereinfache nach Karnaugh/Veitch (falls möglich) und zeichne die Hardware!*



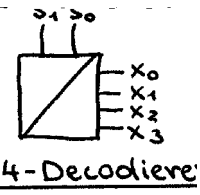
Parallel-Addierwerk: Für die Addition z.B. zweier vierstelliger Dualzahlen ist eine Parallelschaltung aus vier Volladdierern nötig (Parallel-Addierwerk), wobei das Ergebnis allerdings erst dann als endgültig betrachtet werden kann, wenn genügend Zeit verstrichen ist, damit alle Überträge bis zur höchstwertigen Stelle durchgelaufen sind. In der Praxis werden daher auch noch andere Addierschaltungen verwendet.

Mit den Logikgattern und dem Addierwerk stehen mithin alle Bauteile zur Verfügung, die bei der Verarbeitung von Daten im Computer überhaupt gebraucht werden. Auch für die Zweierkomplementbildung reichen diese Bauteile: Das Einerkomplement wird durch stellenweises Invertieren der Ursprungszahl mit je einem NOT-Gatter pro Bit bewirkt; mit Hilfe des Addierwerks kann dann die 1 zur letzten Stelle hinzugezählt werden.

Decodierschaltung



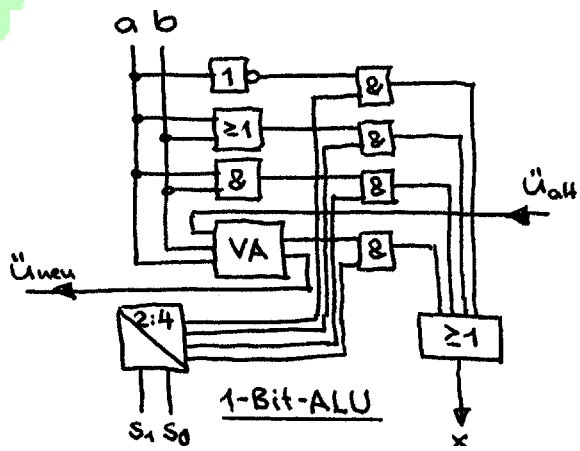
S_1	S_0	X_0	X_1	X_2	X_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



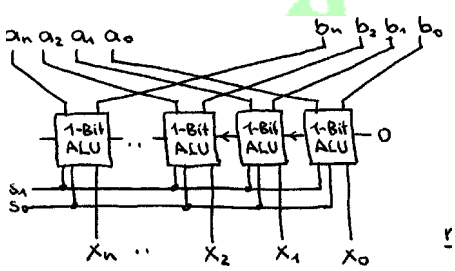
2:4-Decodierer

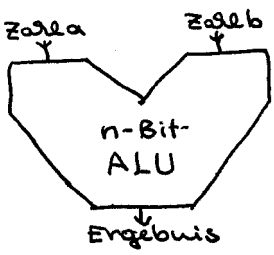
Allerdings muss auch gezielt die richtige Verarbeitungseinheit ausgewählt werden können. Willkürlich lassen sich hier die vier Einheiten mit den Nummern 0 bis 3 versehen; in der computergerechten Darstellung sind diese Nummern 2-Bit-Dualzahlen. Jetzt wird noch eine Schaltung benötigt, die bei Anlegen der Dualzahl an zwei Eingängen (wie immer ein Eingang pro Bit) die richtige der vier Ausgangsleitungen aktiviert. Eine solche Schaltung heißt Decodierer und wird nebenstehend gezeigt. Für jeden Ausgang findet sich in der Ergebnisspalte der Wahrheitswerttabelle genau eine 1; die DNF jedes Ausgangs besteht also nur aus einem einzigen Minterm, der durch das entsprechende UND-Gatter vertreten ist.

Aufbau der arithmetisch-logischen Einheit (ALU)

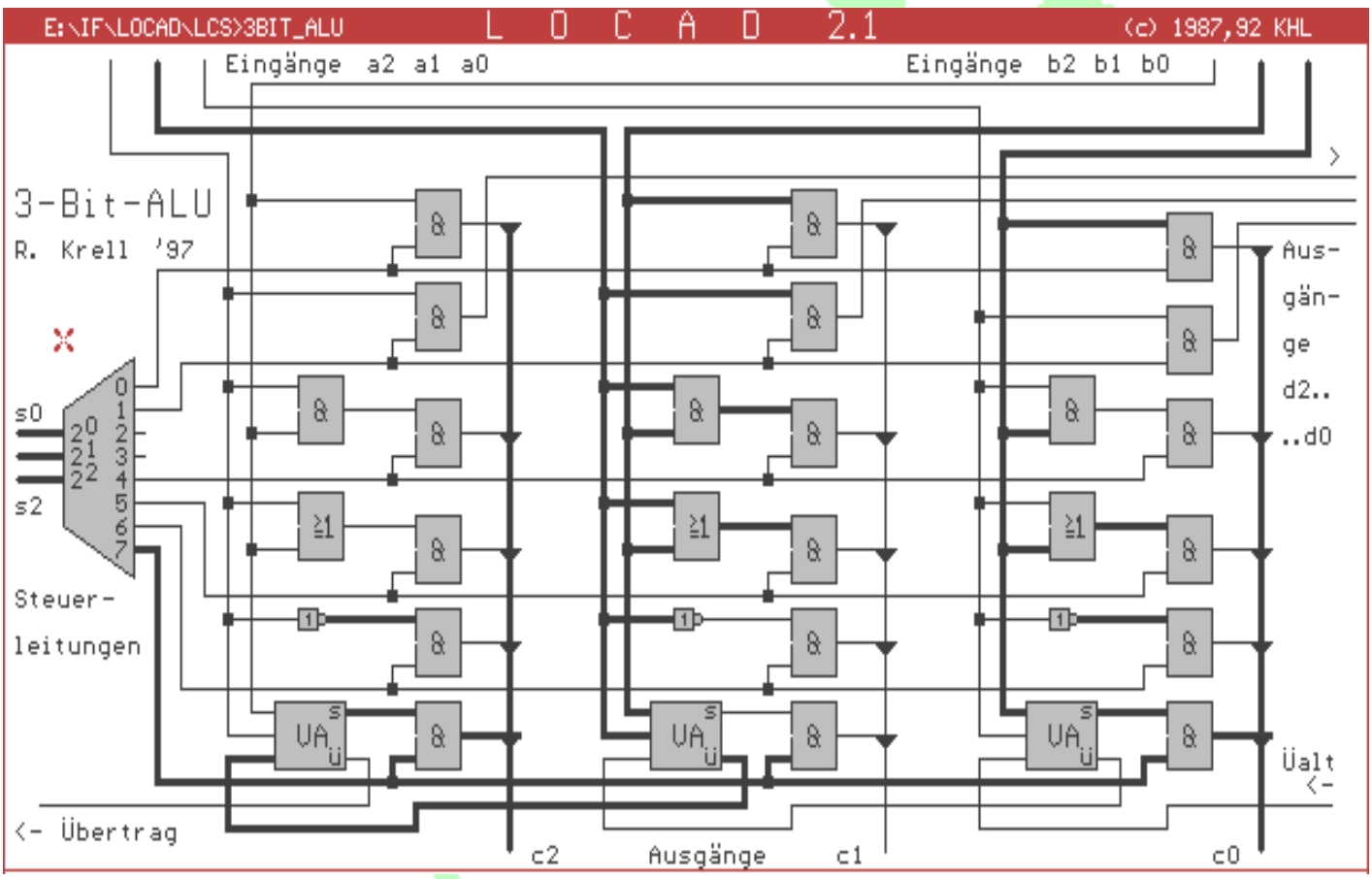


Für jede Stelle der zu verarbeitenden Zahlen werden die vier Verarbeitungseinheiten NOT, OR, AND und Addierer sowie ein Decodierer zusammengepackt. Alle vier Verarbeitungseinheiten verarbeiten die Eingangssignale gleichzeitig. Von den Und-Gattern in den Ausgangsleitungen erhält aber nur eines vom Decodierer die zusätzliche 1, die nötig ist, um das Ergebnis der so ausgewählten Verarbeitungseinheit endgültig zum Ausgang des gesamten Bauteils gelangen zu lassen. 8, 16, 32 oder eine der Bus-/Zahlenbreite des Computers entsprechende Anzahl solcher Gesamtbauteile ergeben, parallelgeschaltet, die arithmetisch-logische Einheit der Maschine, die -- zusammen mit mindestens einem Speicher-Register für Ergebnisse -- das Rechenwerk des Computers bildet, das alle Verarbeitungen durchführt.



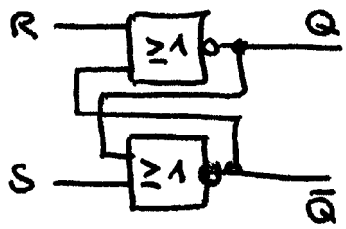


Die nachfolgende Abbildung zeigt eine mit dem Simulationsprogramm LOCAD aufgebaute arithmetisch-logische Einheit, die jeweils zwei 3-Bit-Dualzahlen verarbeiten kann. Außer den logischen Befehlen 4) AND, 5) OR und 6) NOT und dem arithmetischen Befehl 7) ADD beherrscht diese ALU noch die beiden Datentransportbefehle 0) LAD und 1) TRF. Die Befehlsnummern 0) bis 7) müssen -- als Dualzahlen 000 bis 111 -- an die Steuerleitungen links angelegt werden, um über den Decodierer (dessen Aussehen im Simulationsprogramm etwas von der üblichen Notation abweicht) den gewünschten Befehl auszuwählen, wobei allerdings den Nummern 2) und 3) keine Funktion zugeordnet wurde. Grundsätzlich ist die Nummerierung der Befehle willkürlich; hier folgt sie der Nummerierung des 1_AMOR-Modellrechners. Im dargestellten Fall werden gerade die beiden Dualzahlen a=010 und b=011 zu c=101 addiert, weil durch s=111 die Addition (ADD) ausgewählt wurde. Ein alter Übertrag, der bereits bei der niederwertigsten Stelle zu berücksichtigen wäre, wurde rechts unten nicht eingegeben.

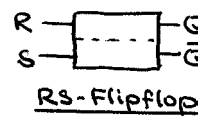


Üblicherweise arbeitet die ALU mit Registern zusammen, d.h. mit Speichern, die die erzeugten Zwischenergebnisse aufnehmen können bzw. benötigte Operanden liefern. Hier würden die Eingänge a vom Akkumulator und die Eingänge b vom Speicherregister beliefert, während die Ausgänge c zum Akkumulator und die Ausgänge d zum Speicherregister führen (vgl. Architektur des Modellrechners 1_AMOR, s.u. Seite 8).
Der Aufbau von Speicherzellen wird im nächsten Kapitel bei den “Schaltwerken” erläutert.

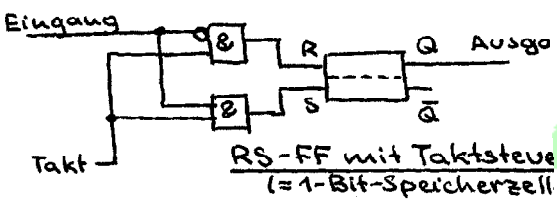
Schaltwerke



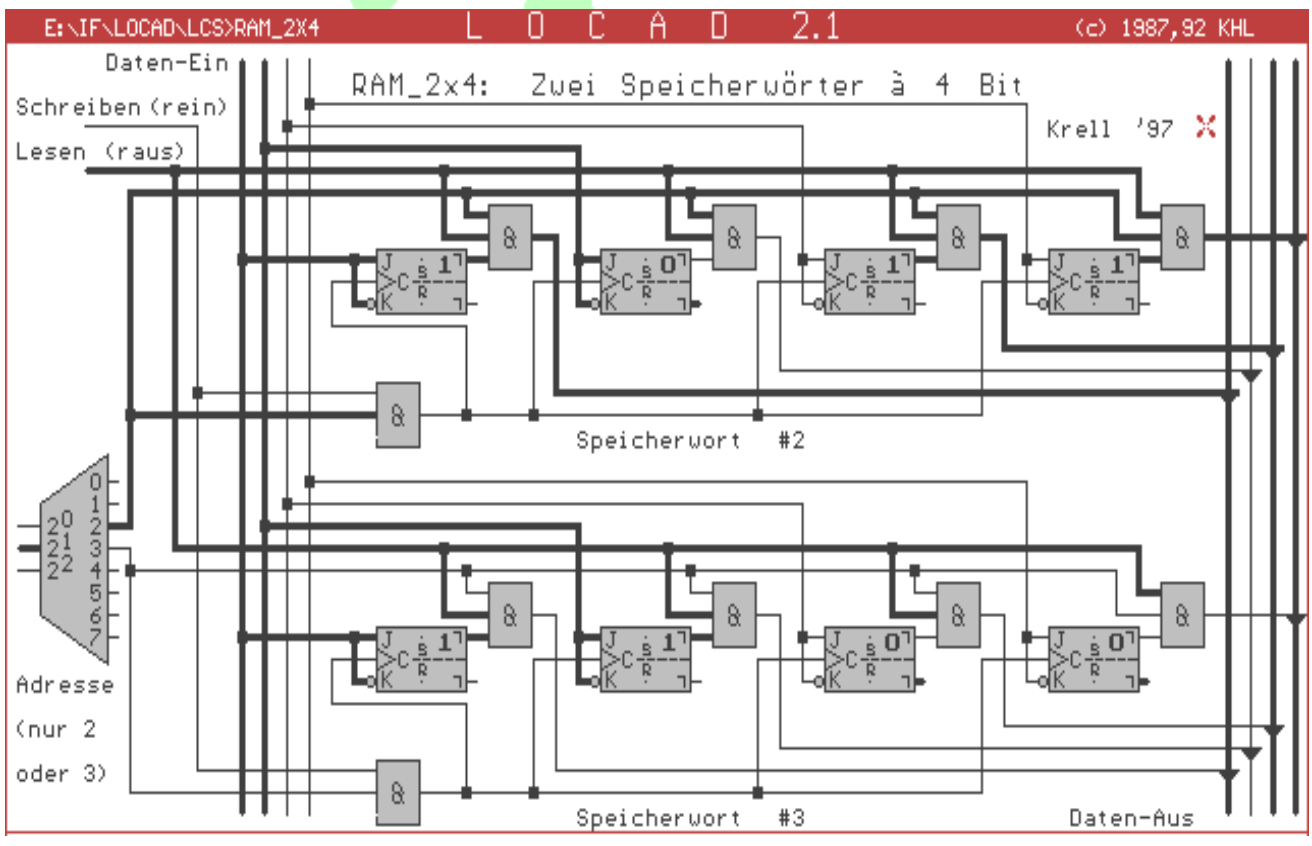
R	S	
0	0	Inhalt halten
0	1	Setzen: Q:=1
1	0	Rücksetzen: Q:=0
1	1	(unerlaubt)



Flipflop: Wurden bisher bei den Schaltnetzen die Eingänge immer unabhängig von den Ausgängen beschaltet, so werden bei Schaltwerken Ausgangssignale auf die Eingangsleitungen zurückgeführt. Diese Rückkopplung erlaubt es der nachfolgenden Schaltung aus zwei kreuzgekoppelten NOR-Gattern, auch dann jeweils einen von zwei stabilen Zuständen zu behalten, wenn das ursprünglich auslösende Eingangssignal nicht mehr anliegt: die Rückkopplung übernimmt dessen Aufgabe. Das entstandene RS-Flipflop wäre damit bereits als 1-Bit-Speicherbaustein geeignet. Eine zusätzliche Beschaltung der Eingänge erreicht, dass das Flipflop immer nur beim Eintreffen eines "Takt"-Signals (z.B. aus dem Adress-Decodierer) die Information einer Eingangsleitung übernimmt und diese beliebig lange -- bis zum nächsten Taktsignal -- behält. Am Ausgang Q kann jederzeit der Speicher-inhalt abgefragt werden, ohne diesen zu zerstören. Eine solche Speicherzelle ist auch unter dem Namen Data-Latch bzw. D-Auffang-Flipflop bekannt und links abgebildet.

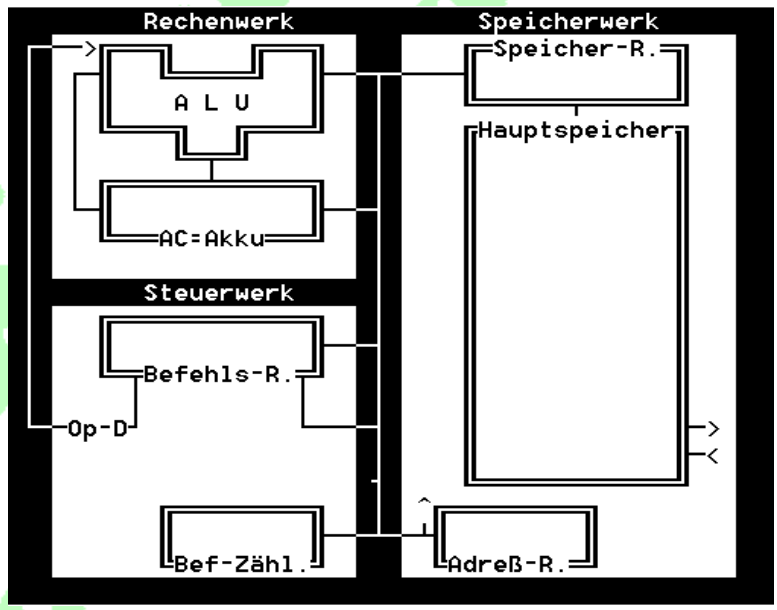


Speicher: Beim Computerspeicher sind üblicherweise acht 1-Bit-Speicherzellen parallel angeordnet (und werden durch den Adress-Decodierer gleichzeitig an- bzw. ausgewählt und durch acht parallele Leitungen gleichzeitig mit Daten versorgt bzw. abgefragt. Der Bildschirmausdruck einer mit dem Simulationsprogramm LOCAD erzeugten Modellschaltung zeigt zwei Blöcke von jeweils nur vier [statt acht] parallelen 1-Bit-Speicherzellen, die gemeinsam adressiert und zum Lesen oder Schreiben aktiviert werden können. Hat der Hauptspeicher



eines Personalcomputers eine Kapazität von 640 kByte = 640 x 1024 Byte = 640 x 1024 x 8 Bit = 5.242.880 Bit, so müssen über 5,2 Millionen solcher Speicherbausteine vorhanden sein. Dank heutiger Miniaturisierung und Fertigungstechnologie gelingt die Unterbringung zum Glück auf wenigen Chips, so dass inzwischen 256 oder mehr MByte (1 MByte = 1024x1024 Bytes ≈ 1,05 Mio. Bytes) Speicherkapazität üblich und erschwinglich sind (256 MB = 2,14 Billionen 1-Bit-Speicherzellen). Für den Hauptspeicher werden allerdings i.a. nicht unbedingt die hier beschriebenen Flipflops, sondern auch pro Bit ein Kondensator eingesetzt, der aufgeladen eine “1” und ungeladen eine “0" repräsentiert. Die geladenen Kondensatoren müssen regelmäßig und rechtzeitig nachgeladen werden (“refresh”), bevor ihre Spannung unter den für “1” festgelegten Mindestwert sinkt. Millionen von Kondensatoren in Halbleiterbauweise finden auf einem Speicherchip Platz.
Selbst für den recht bescheidenen Hauptspeicher des Modellrechners 1_AMOR wären -- noch ohne alle Register -- immerhin 640 Flipflops bzw. Speichereinheiten nötig, von denen allerdings jeweils 10 gemeinsam adressiert und an 10 parallele Datenleitungen angeschlossen würden.

Architektur (Aufbau) des Modellrechners 1_AMOR
Der Hauptspeicher besteht aus 64 Zellen à 10 Bit (10-Bit-Speicherworten), nummeriert von 0 bis 63.
Z. 62: Bildschirmspeicher
Z. 63: Tastaturpuffer




```
dual-dez--mnem.----Beschreibung des Maschinenbefehls-----------------

OOOO  0)  LAD n    Lade (aus dem Speicher in den Akku) AC := Mem[n]
OOOI  1)  TRF n    Transferiere (Speichere)  Mem[n] := AC;  AC := 0
OOIO  2)  NOP n    (no operation)
OOII  3)  NOP n    (no operation)
OIOO  4)  AND n    Verknüpfe bitweise mit "Und"  AC := AC AND Mem[n]
OIOI  5)  OR n     Verknüpfe bitweise mit "Oder"  AC := AC OR Mem[n]
OIIO  6)  NOT n    Invertiere bitweise  AC := NOT(AC)  { = -1-AC }
OIII  7)  ADD n    Addiere  AC := AC + Mem[n]
IOOO  8)  SUB n    Subtrahiere  AC := AC - Mem[n]
IOOI  9)  SHL n    Schiebe links um Mem[n] Stellen  AC:=AC*(2^Mem[n])
IOIO 10)  SPR n    Springe (unbedingt)  GOTO n
IOII 11)  SP< n    Springe bei negativem AC-Inhalt IF AC<0 THEN GOTO n
IIOO 12)  SP= n    Springe bei null  IF AC=0 THEN GOTO n
IIOI 13)  NOP n    (no operation)
IIIO 14)  NOP n    (no operation)
IIII 15)  STOP n   Beende die Ausführung des Maschinenprogramms  HALT

wobei: AC Inhalt des Akkumulators,  Mem[n] Inhalt der n-ten Speicher-
stelle,  n Adresse (0..63),  Inhalt  -512 .. 511
```