Zahlensysteme

Auf der Welt gibt es unzählige Sprachen, von denen jede ihren eigenen Wortschatz, ihre eigene Grammatik und ihre eigenen Symbole, zur bildlichen Darstellung dessen, was man ausdrücken will, mitbringt.

Die deutsche Sprache verwendet neben einigen Sonderzeichen (z.B. Ä, Ü, Ö) beispielsweise die Symbole A bis Z bzw. a bis z, um daraus Wörter zusammenzusetzen. Chinesisch verwendet wiederum völlig andere Symbole, ebenso wie Sprachen aus dem arabischen Raum.

Sprachen verwenden wir, um Worte auszudrücken, z.B. "Hallo", "Computer" oder "Country"

Zahlensysteme werden im Gegensatz dazu verwendet, um Zahlenwerte auszudrücken.

Auch hier gibt es "Sprachen", die jeweils ihre eigenen Symbole (Ziffern) mitbringen, um einen Zahlenwert auszudrücken.

Das Dezimalsystem beispielsweise, welches wir Menschen hauptsächlich verwenden, bringt 10 Ziffern mit, um einen Zahlenwert auszudrücken. Diese lauten 0, 1, 2, 3, 4, 5, 6, 7, 8 und 9

Außerdem bringt es noch das Symbol "-" zur Darstellung von negativen Zahlen mit.

Das Binärsystem wiederum bringt nur zwei Ziffern mit, nämlich 0 und 1.

Im Hexadezimalsystem gibt es sogar 16 Ziffern, nämlich neben den Ziffern 0 bis 9 noch die Buchstaben A-F.

Es gibt auch noch andere Zahlensysteme, z.B. das Oktalsystem welches die Ziffern 0 bis 7 mitbringt.

Diese Ziffern ergeben zusammengesetzt einen bestimmten Zahlenwert, so wie man Buchstaben zu einem Wort zusammensetzt.

Und ebenso wie ein bestimmtes Wort in unterschiedlichen Sprachen unterschiedlich dargestellt wird, wird ein Zahlenwert von einem Zahlensystem zum anderen ebenfalls unterschiedlich dargestellt.

Das deutsche Wort "wo" beispielsweise lautet in englischer Sprache "where".

Übertragen auf Zahlensysteme würde ein Zahlenwert, der im dezimalen Zahlensystem als 200 dargestellt wird, im binären Zahlensystem als %11001000 und im hexadezimalen Zahlensystem als $C8 dargestellt werden.

Hinweis:

In den folgenden Ausführungen werde ich klarerweise sehr oft Darstellungen von Werten in unterschiedlichen Zahlensystem verwenden.

Damit ich die Schreibweisen nicht immer extra erwähnen muss, gebe ich ab jetzt Binärzahlen mit führendem "%", Hexadezimalzahlen mit führendem "$" und Dezimalzahlen ohne führendes Symbol an.

Keine Sorge, wir kommen auf die "Übersetzungen" zwischen den einzelnen Zahlensystemen noch genau zu sprechen, der soeben genannte binäre Zahlenwert %11001000 und hexadezimale Zahlenwert $C8 sollten nur mal als Beispiel für unterschiedliche Darstellungen in unterschiedlichen Zahlensystemen dienen.

Nichtsdestotrotz muss ich ein wenig vorgreifen und die Übersetzungen der Ziffern zwischen dem Dezimalsystem, dem Binärsystem und dem Hexadezimalsystem gegenüberstellen.

Nehmen Sie diese Übersetzungen für den Moment bitte mal so hin. Wie man eine Zahl von einem Zahlensystem ins andere übersetzt, werde ich noch im Detail erklären.

dezimal	**hexadezimal**	**binär**
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Die farbig markierten Bereiche stellen die Ziffern in dem jeweiligen Zahlensystem dar.

Wie bereits erwähnt, bestehen Zahlenwerte, unabhängig vom Zahlensystem, aus einer oder mehreren Ziffern. Die Positionen an denen sich die Ziffern innerhalb des Zahlenwertes befinden, möchte ich ab jetzt als Stellen bezeichnen, welche ich für die nachfolgenden Ausführungen von rechts und mit 0 beginnend aufsteigend durchnummerieren möchte.

Beispiel 1:

Beginnen wir mit unserem vertrauten Dezimalsystem und nehmen als Beispiel die Zahl 5376.

Diese Zahl besteht aus 4 Ziffern.

An Stelle Nr. 0 haben wir hier die Ziffer 6, an Stelle Nr. 1 die Ziffer 7, an Stelle Nr. 2 die Ziffer 3 und an Stelle Nr. 3 die Ziffer 5.

Beispiel 2:

Die binäre Zahl %10110

Diese Zahl besteht aus 5 Ziffern welche von 0 bis 4 nummeriert sind.
An Stelle Nr. 0 haben wir hier die Ziffer %0, an Stelle Nr. 1 die Ziffer %1, an Stelle Nr. 2 die Ziffer %1, an Stelle Nr. 3 die Ziffer %0 und an Stelle Nr. 4 die Ziffer %1.

Beispiel 3:

Die hexadezimale Zahl $C8

An Stelle Nr. 0 haben wir die Ziffer $8 und an Stelle Nr. 1 die Ziffer $C.

Doch was unterscheidet nun die Zahlensysteme voneinander wenn man mal von der unterschiedlichen Anzahl und Art der Ziffern absieht?

Der Unterschied liegt in der Wertigkeit der einzelnen Stellen. Jedes Zahlensystem hat einen bestimmten Basiswert, auf dem die Wertigkeit der einzelnen Stellen basieren.

Im Dezimalsystem ist dies die Zahl 10, im Binärsystem die Zahl 2 und im hexadezimalen System die Zahl 16. Im angesprochenen Oktalsystem wäre das die Zahl 8, man kann also mit jedem Basiswert ein neues Zahlensystem bilden, z.B. auch mit dem Basiswert 3 wenn man will.

Die Wertigkeiten der einzelnen Stellen ergeben sich aus der Potenz:

Basiszahl des Zahlensystems hoch Nummer der Stelle

Wertigkeiten im Dezimalsystem:

10 hoch 0 = 1
10 hoch 1 = 10
10 hoch 2 = 100
10 hoch 3 = 1000
10 hoch 4 = 10000
...

Wertigkeiten im Binärsystem:

2 hoch 0 = 1
2 hoch 1 = 2
2 hoch 2 = 4
2 hoch 3 = 8
2 hoch 4 = 16
…

Wertigkeiten im Hexadezimalsystem:

16 hoch 0 = 1
16 hoch 1 = 16
16 hoch 2 = 256
16 hoch 3 = 4096
16 hoch 4 = 65536
…

Ok, was haben wir bis jetzt? Wir haben eine Reihe aus Ziffern vor uns, wir wissen aus der obigen Tabelle welchen dezimalen Wert jede Ziffer hat.

Im Falle einer Binärzahl entweder den dezimalen Wert 0 oder 1 und im Falle einer Hexadezimalzahl ein dezimalen Wert zwischen 0 und 15. Die hexadezimale Ziffer $D würde laut der Tabelle dem dezimalen Wert 13 entsprechen.

Wir wissen außerdem, welche dezimale Wertigkeit eine bestimmte Stelle im jeweiligen Zahlensystem hat (siehe oben)

Wie kommen wir nun zum dezimalen Gesamtwert einer Zahl die uns als Zahl aus einem anderen Zahlensystem vorliegt?

Um den dezimalen Wert einer Zahl zu erhalten, geht man die Zahl Stelle für Stelle durch, multipliziert die dezimale Wertigkeit an dieser Stelle mit der dezimalen Ziffer an dieser Stelle und am Ende summiert man diese Produkte auf, um den dezimalen Wert zu erhalten, den die Zahl darstellt.

Spielen wir die Sache mal anhand der oben genannten dezimalen Zahl 5376 durch. Klar, wir wissen natürlich was rauskommt, nämlich 5376, aber nehmen wir mal an, wir sehen nur die dezimalen Ziffern 5, 3, 7 und 6 vor uns und wollen den Gesamtwert ausrechnen.

Wir gehen vor wie vorhin beschrieben, d.h. wir bilden jeweils das Produkt der Ziffer und der Wertigkeit an dieser Stelle und am Ende summieren wir diese Produkte zum Gesamtwert auf.

6 * (10 hoch 0) = 6 * 1 = 6
7 * (10 hoch 1) = 7 * 10 = 70
3 * (10 hoch 2) = 3 * 100 = 300
5 * (10 hoch 3) = 5 * 1000 = 5000

Nun summieren wir diese Einzelwerte:

6 + 70 + 300 + 5000 = 5376

Nun wenden wir dasselbe Schema auf die oben genannte Binärzahl %10110 an.

%0 * (2 hoch 0) = 0 * 1 = 0
%1 * (2 hoch 1) = 1 * 2 = 2
%1 * (2 hoch 2) = 1 * 4 = 4
%0 * (2 hoch 3) = 0 * 8 = 0
%1 * (2 hoch 4) = 1 * 16 = 16

Nun summieren wir wieder die Einzelwerte:

0 + 2 + 4 + 0 + 16 = 22

Wenden wir das Schema nun auch auf die hexadezimale Zahl $C8 an.

$8 * (16 hoch 0) = 8 * 1 = 8
$C * (16 hoch 1) = 12 * 16 = 192

Summe der Einzelwerte:

8 + 192 = 200

Jetzt haben wir gelernt wie wir eine Binärzahl oder eine Hexadezimalzahl ins Dezimalsystem umrechnen können.

Spielen wir das zur Übung nochmal anhand zweier Beispiele durch:

Beispiel 1:

Binärzahl %1001111000101

%1 * (2 hoch 0) = 1 * 1 = 1
%0 * (2 hoch 1) = 0 * 2 = 0
%1 * (2 hoch 2) = 1 * 4 = 4
%0 * (2 hoch 3) = 0 * 8 = 0
%0 * (2 hoch 4) = 0 * 16 = 0
%0 * (2 hoch 5) = 0 * 32 = 0
%1 * (2 hoch 6) = 1 * 64 = 64
%1 * (2 hoch 7) = 1 * 128 = 128
%1 * (2 hoch 8) = 1 * 256 = 256
%1 * (2 hoch 9) = 1 * 512 = 512
%0 * (2 hoch 10) = 0 * 1024 = 0
%0 * (2 hoch 11) = 0 * 2048 = 0
%1 * (2 hoch 12) = 1 * 4096 = 4096

1 + 0 + 4 + 0 + 0 + 0 + 64 + 128 + 256 + 512 + 0 + 0 + 4096 = 5061

Beispiel 2:

Hexadezimalzahl $FA37

$7 * (16 hoch 0) = 7 * 1 = 7
$3 * (16 hoch 1) = 3 * 16 = 48
$A * (16 hoch 2) = 10 * 256 = 2560
$F * (16 hoch 3) = 15 * 4096 = 61440

7 + 48 + 2560 + 61440 = 64055

Doch wie sieht der umgekehrte Weg aus? Wie kann man eine Dezimalzahl in eine Binärzahl oder eine Hexadezimalzahl umrechnen?

Kurzbeschreibung die für jedes gewünschte Ziel-Zahlensystem gilt:

Man dividiert die Dezimalzahl laufend durch die Basis des gewünschten Ziel-Zahlensystems und schreibt den Rest, der bei der Division bleibt auf (in der Darstellung des Ziel-Zahlensystems).

Das Ergebnis der Division nimmt man als Ausgangsbasis für die nächste Division. Man dividiert so lange, bis das Ergebnis der Division gleich 0 ist. Dann schreibt man die Restwerte von unten beginnend nebeneinander und hat das gewünschte Ergebnis.

Beispiel 1:

Umrechnen der Dezimalzahl 2348 in das Binärsystem.

2348 : 2 = 1174	Rest %0
1174 : 2 = 587	Rest %0
587 : 2 = 293	Rest %1
293 : 2 = 146	Rest %1
146 : 2 = 73	Rest %0
73 : 2 = 36	Rest %1
36 : 2 = 18	Rest %0
18 : 2 = 9	Rest %0
9 : 2 = 4	Rest %1
4 : 2 = 2	Rest %0
2 : 2 = 1	Rest %0
1 : 2 = 0	Rest %1

Nun schreiben wir von unten beginnen die Restwerte nebeneinander und kommen auf das Ergebnis:

%100100101100

Beispiel 2:

Umrechnen der Dezimalzahl 55327 in das Hexadezimalsystem.

55327 : 16 = 3457	Rest $F (dezimal 15)
3457 : 16 = 216	Rest $1 (dezimal 1)
216 : 16 = 13	Rest $8 (dezimal 8)
13 : 16 = 0	Rest $D (dezimal 13)

Nun schreiben wir wie vorhin die Restwerte von unten beginnend nebeneinander und kommen auf das Ergebnis:

$D81F

Beispiel 3:

Umrechnen der Dezimalzahl 12 in das Binärsystem.

12 : 2 = 6	Rest %0
6 : 2 = 3	Rest %0
3 : 2 = 1	Rest %1
1 : 2 = 0	Rest %1

Das Ergebnis ist %1100 und wenn Sie einen kurzen Blick in die Tabelle werfen, in der die Ziffern der Zahlensysteme gegenübergestellt wurden, werden Sie sehen, dass dies korrekt ist:

Beispiel 4:

Umrechnen der Dezimalzahl 1024 in das Hexadezimalsystem.

1024 : 16 = 64	Rest $0 (dezimal 0)
64 : 16 = 4	Rest $0 (dezimal 0)
4 : 16 = 0	Rest $4 (dezimal 4)

Nun wieder die Restwerte von unten beginnend nebeneinander schreiben und wir kommen auf das Ergebnis:

$400

Eine Hexadezimalzahl in eine Binärzahl umzuwandeln und umgekehrt ist recht einfach wie Sie gleich sehen werden.

Das Hexadezimalsystem besitzt wie wir nun wissen 16 Ziffern, 0-9 und A-F.

Wieviele Werte kann man mit einer Ziffer im Hexadezimalsystem darstellen? Richtig, 16.

Wieviele Ziffern braucht man dafür im Binärsystem? Richtig, 4, denn $F entspricht %1111
Um den höchsten Wert, welchen man mit einer hexadezimalen Ziffer darstellen kann ($F), binär darzustellen, braucht es also eine vierstellige Binärzahl.

Um also eine Hexadezimalzahl in eine Binärzahl umzuwandeln, braucht man nur die binären Gegenstücke ihrer Ziffern aus der Tabelle nebeneinander aufschreiben.

Dabei gilt es jedoch folgendes zu beachten:

Sollte das binäre Gegenstück der jeweiligen hexadezimalen Ziffer weniger als 4 Stellen haben, dann muss diese mit 0-Stellen auf diese Länge von 4 Stellen von links aufgefüllt werden, damit die Reihenfolge der entsprechenden Stellen wieder zusammenpasst.

Hier ein Beispiel:

Wir wollen die Hexadezimalzahl $AFDE ins Binärsystem umwandeln.

$A = binär %1010
$F = binär %1111
$D = binär %1101
$E = binär %1110

Nun schreiben wir binären Gegenstücke der Ziffern nebeneinander.

$A	$F	$D	$E
1010	1111	1101	1110

Das binäre Gegenstück zu $AFDE lautet also %1010111111011110

Nun ein Beispiel bei dem die binären Gegenstücke der hexadezimalen Ziffern teilweise weniger als 4 Stellen haben.

Wir wollen die Hexadezimalzahl $3D7F ins Binärsystem umwandeln.

$3 = %11
$D = $1101
$7 = %111
$F = %1111

$3	$D	$7	$F
11	1101	111	1111

Das ergibt nebeneinander geschrieben 1111011111111

Rechnen wir diese Binärzahl ins Dezimalsystem um:

%1 * (2 hoch 0) = 1 * 1 = 1
%1 * (2 hoch 1) = 1 * 2 = 2
%1 * (2 hoch 2) = 1 * 4 = 4
%1 * (2 hoch 3) = 1 * 8 = 8
%1 * (2 hoch 4) = 1 * 16 = 16
%1 * (2 hoch 5) = 1 * 32 = 32
%1 * (2 hoch 6) = 1 * 64 = 64
%1 * (2 hoch 7) = 1 * 128 = 128
%0 * (2 hoch 8) = 0 * 256 = 0
%1 * (2 hoch 9) = 1 * 512 = 512
%1 * (2 hoch 10) = 1 * 1024 = 1024
%1 * (2 hoch 11) = 1 * 2048 = 2048
%1 * (2 hoch 12) = 1 * 4096 = 4096

1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 0 + 512 + 1024 + 2048 + 4096 = 7935 oder $1EFF

Dies entspricht jedoch nicht unserer ursprünglichen Zahl $3D7F

Daher müssen wir folgende Korrektur vornehmen:

$3	$D	$7	$F
0011	1101	0111	1111

%1 * (2 hoch 0) = 1 * 1 = 1
%1 * (2 hoch 1) = 1 * 2 = 2
%1 * (2 hoch 2) = 1 * 4 = 4
%1 * (2 hoch 3) = 1 * 8 = 8
%1 * (2 hoch 4) = 1 * 16 = 16
%1 * (2 hoch 5) = 1 * 32 = 32
%1 * (2 hoch 6) = 1 * 64 = 64
%0 * (2 hoch 7) = 0 * 128 = 0
%1 * (2 hoch 8) = 1 * 256 = 256
%0 * (2 hoch 9) = 0 * 512 = 0
%1 * (2 hoch 10) = 1 * 1024 = 1024
%1 * (2 hoch 11) = 1 * 2048 = 2048
%1 * (2 hoch 12) = 1 * 4096 = 4096
%1 * (2 hoch 13) = 1 * 8192 = 8192

%0 * (2 hoch 14) = 0 * 16384 = 0
%0 * (2 hoch 15) = 0 * 32768 = 0

1 + 2 + 4 + 8 + 16 + 32 + 64 + 0 + 256 + 0 + 1024 + 2048 + 4096 + 8192 + 0 + 0 = 15743 bzw. nun korrekt $3D7F.

Wie sieht es umgekehrt aus, also vom Binärsystem ins Hexadezimalsystem?

Angenommen wir wollen die Binärzahl %10011100001010 ins Hexadezimalsystem umwandeln.

Dann teilen wir die Binärzahl von rechts beginnend in Gruppen von 4 Ziffern auf.

10 0111 0000 1010

Am linken Ende reichen die Stellen nicht aus um eine Vierergruppe zu bilden, daher füllen wir sie mit Nullen auf.

0010 0111 0000 1010

Nun gehen wir den umgekehrten Weg wie vorhin bei der Umrechnung einer Hexadezimalzahl ins Binärsystem:

0010	0111	0000	1010
$2	$7	$0	$A

Das Ergebnis lautet: $270A oder dezimal 9994.

Rechnen wir nach:

$A * (16 hoch 0) = 10 * 1 = 10
$0 * (16 hoch 1) = 0 * 16 = 0
$7 * (16 hoch 2) = 7 * 256 = 1792
$2 * (16 hoch 3) = 2 * 4096 = 8192

10 + 0 + 1792 + 8192 = 9994

Und zur Übung und Kontrolle rechnen wir das gleich ins Binärsystem um:

9994 : 2 = 4997	Rest %0
4997 : 2 = 2498	Rest %1
2498 : 2 = 1249	Rest %0
1249 : 2 = 624	Rest %1
624 : 2 = 312	Rest %0
312 : 2 = 156	Rest %0
156 : 2 = 78	Rest %0
78 : 2 = 39	Rest %0
39 : 2 = 19	Rest %1
19 : 2 = 9	Rest %1
9 : 2 = 4	Rest %1
4 : 2 = 2	Rest %0
2 : 2 = 1	Rest %0

1 : 2 = 0 Rest %1

Wodurch wir dann wieder zu der Binärzahl %10011100001010 kommen.

Addition von Binärzahlen

Im Grunde funktioniert die Addition im Binärsystem genau gleich wie wir es vom Dezimalsystem her kennen.

0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 0 und Übertrag 1 auf die nächste Stelle

Falls wir folgende Situation haben:

1 + 1 + Übertrag 1, dann ergibt das 1 und Übertrag 1 auf die nächste Stelle

Beispiel 1

Addieren wir mal die beiden Binärzahlen %00111010 (dezimal 58) und %00010011 (dezimal 19) Das ergibt %01001101 (dezimal 77)

Folgende Tabelle stellt diese Addition dar. In der ersten Zeile stehen die Ziffern der ersten binären Zahl, in der zweiten Zeile stehen die Ziffern der zweiten binären Zahl und das Ergebnis der Addition ist gelb markiert in der vierten Zeile dargestellt.

In der dritten Zeile wird durch die hellblauen Zellen hervorgehoben, dass hier ein Übertrag stattgefunden hat.

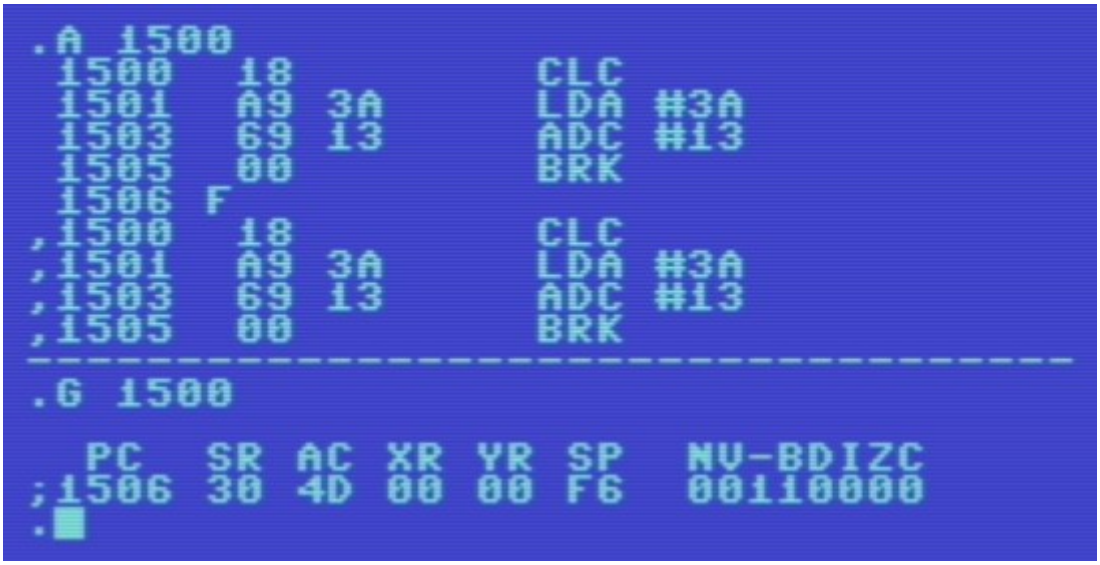
0	0	1	1	1	0	1	0
0	0	0	1	0	0	1	1
	1	1			1		
0	1	0	0	1	1	0	1

Beginnen wir von rechts mit der Addition.

0 + 1 = 1
1 + 1 = 0 + Übertrag 1
0 + 0 + Übertrag 1 = 1
1 + 0 = 1
1 + 1 = 0 + Übertrag 1
1 + 0 + Übertrag 1 = 0 + Übertrag 1
0 + 0 + Übertrag 1 = 1
0 + 0 = 0

So sieht der zugehörige Assembler-Code aus, als Ergebnis ergibt sich im Akkumulator der Wert $4D (dezimal 77 bzw. wie oben in der Tabelle gelb markiert binär %01001101)

Was es mit dem Befehl CLC auf sich hat, erkläre ich noch. Den Befehl ADC kennen sie bereits, er addiert einen Wert zum Inhalt des Akkumulators.



Beispiel 2

Addieren wir nun die beiden Binärzahlen %01110110 (dezimal 118) und %00111111 (dezimal 63) Das ergibt %10110101 (dezimal 181)

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

Wir beginnen wieder von rechts mit der Addition.

0 + 1 = 1
1 + 1 = 0 + Übertrag 1
1 + 1 + Übertrag 1 = 1 + Übertrag 1
0 + 1 + Übertrag 1 = 0 + Übertrag 1
1 + 1 + Übertrag 1 = 1 + Übertrag 1
1 + 1 + Übertrag 1 = 1 + Übertrag 1
1 + Übertrag 1 = 0 + Übertrag 1
0 + 0 + Übertrag 1 = 1

So sieht der zugehörige Assembler-Code aus, als Ergebnis ergibt sich im Akkumulator der Wert $B5 (dezimal 181 bzw. wie oben in der Tabelle gelb markiert binär %10110101)

Beispiel 3

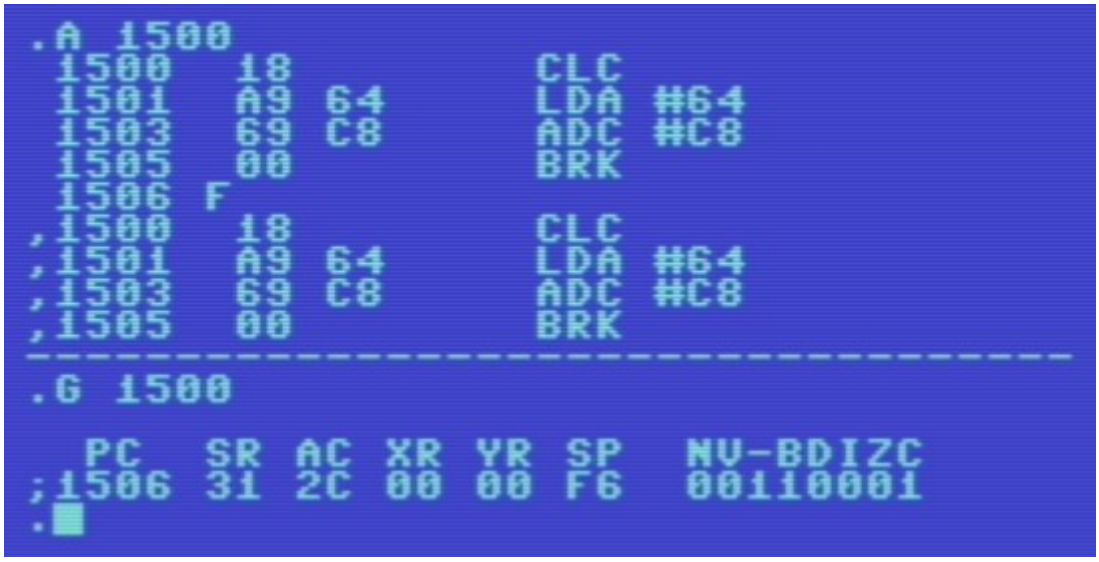
Addieren wir nun die beiden Binärzahlen %01100100 (dezimal 100) und %11001000 (dezimal 200) Das ergibt dezimal 300. Hier ergibt sich jedoch das Problem, dass dieser Wert größer als 255 ist, also nicht mehr mit 8 Bits (also einem Byte) dargestellt werden kann.

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

Wie bisher beginnen wir wieder von rechts mit der Addition.

0 + 0 = 0
0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
0 + 0 = 0
1 + 0 = 1
1 + 1 = 0 + Übertrag 1
0 + 1 + Übertrag 1 = 0 + Übertrag 1

So sieht der zugehörige Assembler-Code aus, als Ergebnis ergibt sich im Akkumulator der Wert $2C (dezimal 44 bzw. wie oben in der Tabelle gelb markiert binär %00101100), die 44 ergibt sich aus der Differenz 300 – 256, also



Hier findet in der letzten Stelle ein Übertrag statt. Das bedeutet, dass das Ergebnis größer als 255 ist, sich also nicht mehr mit 8 Bits (also einem Byte) darstellen lässt.

Dies wird dadurch gekennzeichnet, dass im Statusregister das Carry Flag auf 1 gesetzt wird, der Übertrag an der linken Stelle wird also in dieses Flag übertragen.

Beispiel 4

Addieren wir nun die beiden Binärzahlen %11111111 (dezimal 255) und %11111111 (dezimal 255)

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

Beginnen wir wieder von rechts mit der Addition.

1 + 1 = 0 + Übertrag 1
1 + 1 + Übertrag 1 = 1 + Übertrag 1
1 + 1 + Übertrag 1 = 1 + Übertrag 1
1 + 1 + Übertrag 1 = 1 + Übertrag 1
1 + 1 + Übertrag 1 = 1 + Übertrag 1
1 + 1 + Übertrag 1 = 1 + Übertrag 1
1 + 1 + Übertrag 1 = 1 + Übertrag 1
1 + 1 + Übertrag 1 = 1 + Übertrag 1

So sieht der zugehörige Assembler-Code aus, als Ergebnis ergibt sich im Akkumulator der Wert $FE (dezimal 254 bzw. wie oben in der Tabelle gelb markiert binär %11111110), die 254 ergibt sich aus der Differenz 510 – 256, also

Auch hier findet in der letzten Stelle wieder ein Übertrag statt, d.h. das Ergebnis ist größer als 255, was auch durch ein gesetztes Carry Flag gekennzeichnet wird.

Kommen wir nun zur Bedeutung des Befehls CLC (Clear Carry)

Seine Aufgabe ist sehr einfach, er setzt das Carry Flag im Statusregister zurück d.h. er setzt es auf den Wert 0.

Warum habe ich den Befehl hier immer eingebaut? Der Grund ist, dass der Befehl ADC das Carry Flag in die Rechnung miteinbezieht.

Spielen wir Beispiel 1 nochmals durch, gehen dieses mal jedoch davon aus, dass das Carry Flag gesetzt ist. Mit dem Befehl SEC (Set Carry) kann man übrigens das Carry Flag setzen.

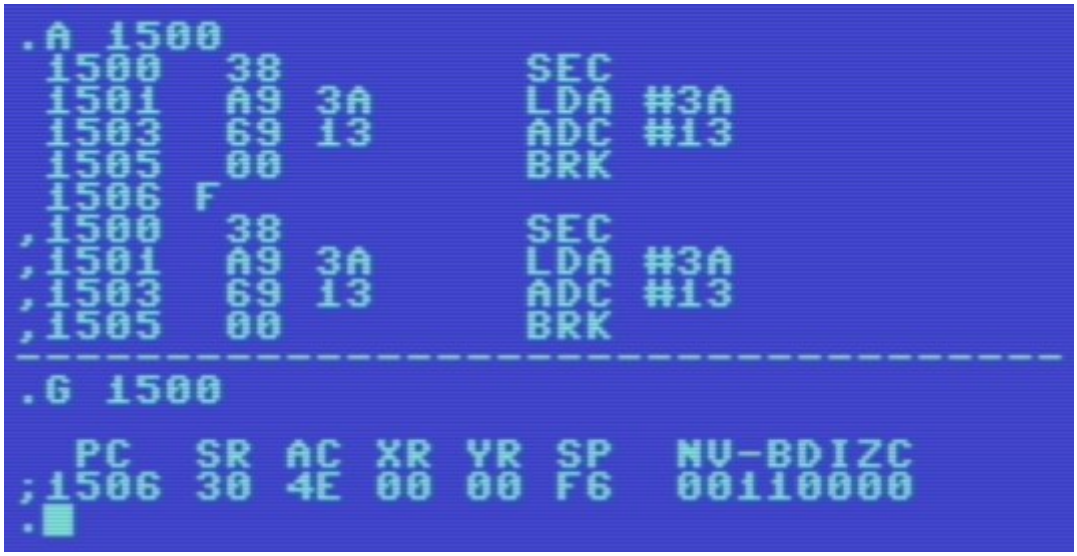
Dann passiert folgendes: Das gesetzte Carryflag wird als „Übertrag“ zu Beginn unserer Addition übernommen, was dann natürlich zu einem völlig anderem Ergebnis führt.

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

In Beispiel 1 haben wir das Ergebnis %01001101 (dezimal 77) erhalten.

Hier erhalten wir jedoch das Ergebnis %01001110 (dezimal 78) weil zu Beginn der Addition zusätzlich der Übertrag in das Ergebnis eingeflossen ist.

Hier der zugehörige Assembler-Code mit dem Ergebnis $4E (dezimal 78) im Akkumulator.



Sie fragen sich jetzt sicher, wofür das gut sein soll und wo man dieses Setzen / Löschen des Carry Flags in Zusammenhang mit der Addition anwenden kann.

Die Antwort lautet:

Addition von zwei 16 Bit Zahlen

Angenommen wir wollen die Zahlen 1000 und 2000 addieren. Das sind beides 16 Bit Werte, das Ergebnis lautet 3000 und ist ebenfalls wieder ein 16 Bit Wert.

Verwenden wir die Speicherstelle $1500 für das niederwertige Byte des Ergebnisses und die Speicherstelle $1501 für das höherwertige Byte des Ergebnisses, denn wie wir bereits wissen, werden 16 Bit Werte in dieser Reihenfolge im Speicher abgelegt.

Geben Sie ab Adresse $1508 folgendes Programm ein und starten es anschließend mit dem Befehl G 1508

Zeigen Sie nun mit dem Befehl M 1500 1510 den Inhalt dieses Speicherbereichs an.

An der Speicherstelle $1500 sieht man das Byte $B8 und an der Speicherstelle $1501 das Byte $0B.

Dies entspricht dem Wert $0BB8 oder dezimal 3000, also der Summe aus 1000 und 2000.

Wie funktioniert das Programm?

Der Wert 1000 entspricht der Hexadezimalzahl $03E8, d.h. das niederwertige Byte ist $E8 und das höherwertige Byte ist $03.

Der Wert 2000 entspricht der Hexadezimalzahl $07D0, d.h. das niederwertige Byte ist $D0 und das höherwertige Byte ist $07.

Die Summe aus 1000 und 2000 lautet 3000, was der Hexadezimalzahl $0BB8 entspricht. Das niederwertige Byte ist $B8 und das höherwertige Byte ist $0B.

Als erstes setzen wir das Carry Flag mit dem Befehl CLC zurück, da zu Beginn der Addition ja noch kein Übertrag stattgefunden hat.

Dann addieren wir die beiden niederwertigen Bytes der beiden Zahlen, wir rechnen also:

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

+ 1 übertrag

An der letzten Stelle ergibt sich also:

1 + 1 + 1 = 1 + Übertrag 1, wodurch das Carry Flag gesetzt wird

Das Ergebnis %10111000 oder $B8 schreiben wir in die Speicherstelle $1500, da dies das niederwertige Byte unserer Summe ist.

Nun addieren wir die höherwertigen Bytes der beiden Zahlen, beachten jedoch, dass es bei der Addition der beiden niederwertigen Bytes einen Übertrag gab, was durch das dunkelblaue Feld gekennzeichnet wird.

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

Das Ergebnis lautet %00001011 oder $0B.

Diesen Wert schreiben wir in die Speicherstelle $1501, da dies das höherwertige Byte unserer Summe ist.

Nun steht die Summe im Speicher wie wir bereits im obigen Speicherdump ab der Adresse $1500 gesehen haben.

Die Darstellung von negativen Zahlen

Wollen wir eine negative Zahl, z.B. die -58, binär darstellen, dann rechnen wir zuerst den positiven Wert (58) in eine Binärzahl um.

Das ergibt: %00111010

Nun bilden wir das sogenannte Einerkomplement. Um das Einerkomplement einer Binärzahl zu erhalten, drehen wir alle Bits um, d.h. aus einer 1 wird eine 0 und aus einer 0 wird eine 1.

Einerkomplement: %11000101

Negative Zahlen werden im sogenannten Zweierkomplement dargestellt. Um das Zweierkomplement zu erhalten, addieren wir 1 zum Einerkomplement.

Das ergibt:

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

Die binäre Darstellung der negativen Zahl -58 lautet also %11000110

Negative Zahlen zeichnen sich dadurch aus, dass das am weitesten links stehende Bit 7 gesetzt ist, so wie es auch hier der Fall ist.

Durch die Reservierung des Bit 7 für die Darstellung des Vorzeichens, stehen nur mehr die Bits 0-6 für den eigentlichen Wert zur Verfügung. Dadurch reduziert sich der Darstellungsbereich von vorzeichenbehafteten Zahlen auf -128 bis + 127.

Ohne Reservierung des Bit 7 für das Vorzeichen liegt der Wertebereich ja im Bereich von 0 bis 255.

Nun könnte man fragen:

Der binäre Wert %11000110 entspricht doch auch dem dezimalen Wert 198 oder?
Woran erkennt die CPU ob wir nun die -58 oder die 198 meinen?

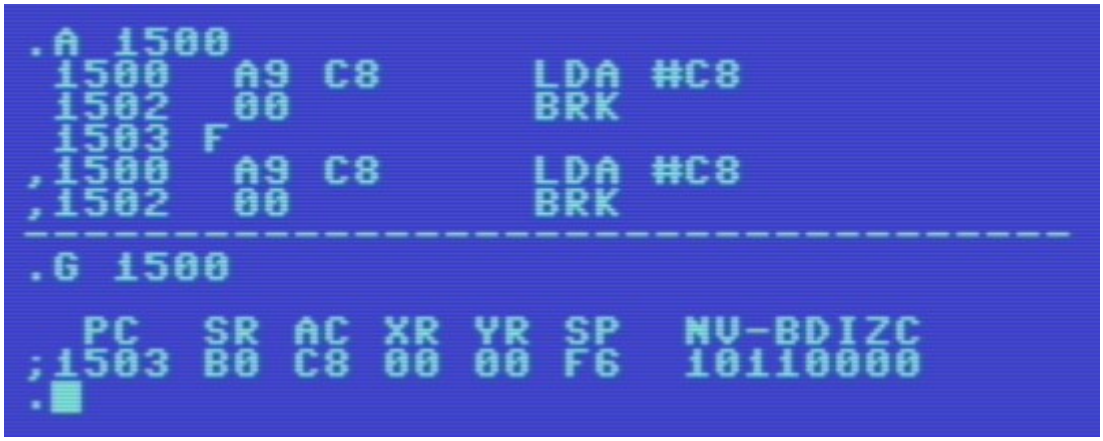
Die Antwort lautet:

Gar nicht, die CPU kennt keine negativen Zahlen in diesem Sinn, für sie gibt es nur den binären Wert %11000110.

Es liegt allein am Programmierer, wie der Wert interpretiert werden soll, ob als negative Zahl -58 oder als 198.

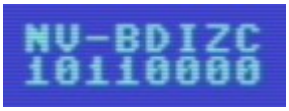
Ist das Ergebnis einer Operation negativ, das am weitesten links stehende Bit hat also den Wert 1, dann wird im Statusregister das sogenannte Negative Flag gesetzt. Es steht im Statusregister am weitesten links, also an der Bit Position 7 und stellt eine Kopie des am weitesten links stehenden Bits des Wertes dar, mit dem man das Register geladen hat.

Hier ein Beispiel:



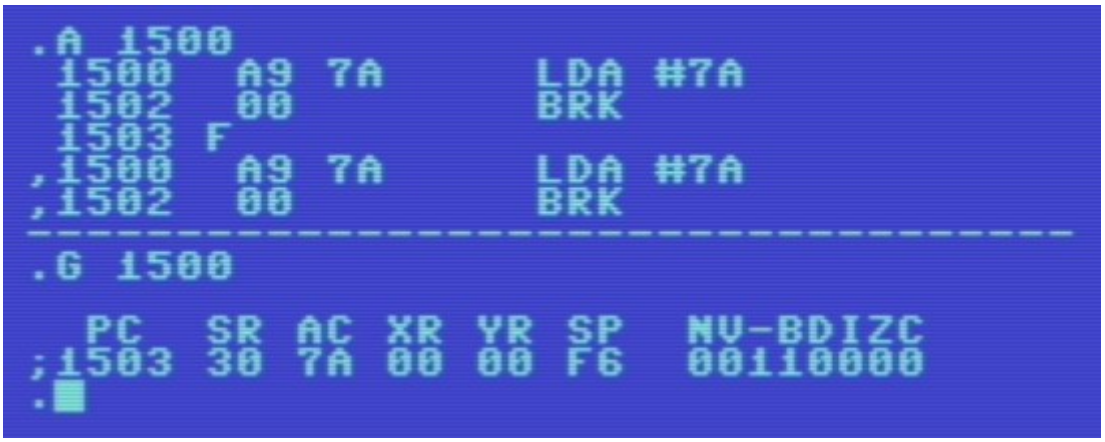
Hier wird der Wert $C8 in den Akkumulator geladen, d.h. der binäre Wert %11001000, das am weitesten links stehende Bit hat also den Wert 1.

Als Ergebnis sieht man, dass nun im Statusregister das Negative Flag gesetzt ist.



Das Negative Flag ist das am weitesten links stehende Flag (N)

Sehen Sie sich zum Vergleich dieses Beispiel an:



Hier wird der Wert $7A in den Akkumulator geladen, d.h. der binäre Wert %01111010, das am weitesten links stehende Bit hat also den Wert 0.

Als Ergebnis sieht man, dass dieses mal das Negative Flag nicht gesetzt ist, also den Wert 0 enthält.

Logische Verknüpfungen

Sehen Sie sich folgendes BASIC-Programm an:

In Zeile 20 wird hier geprüft, ob eine Aussage (X=64) wahr oder falsch ist.

Wenn sie wahr ist, dann wird der Text „WAHR“ ausgegeben, ansonsten der Text „FALSCH“
In diesem Fall ist sie wahr, denn X hat den Inhalt 64.

Würden wir in Zeile 10 für X einen anderen Wert verwenden, dann wäre die Aussage X=64 falsch, wodurch in Zeile 30 dann der Text „FALSCH“ ausgegeben wird.

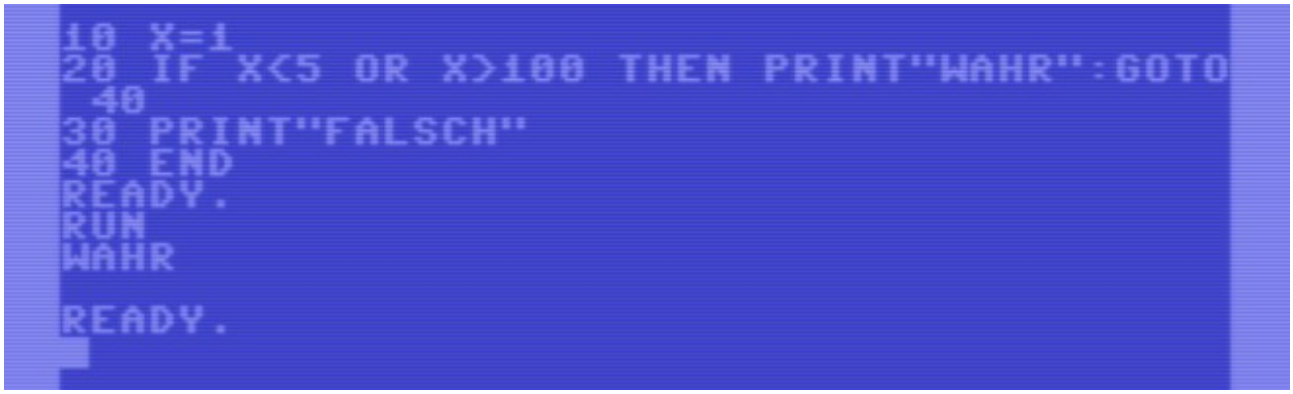
Die Aussage „X=64“ in Zeile 20 könnte man mit einem Bit (nennen wir es B) gleichsetzen, das entweder den Wert 1 (wahr) oder 0 (falsch) enthält.

Man kann durch Anwendung des Schlüsselwortes NOT die Prüfung der Aussage auch umkehren.

IF NOT(X=64) THEN PRINT“WAHR“:GOTO 40

Der Ausdruck NOT(X=64) ist umgekehrt genau dann wahr, wenn X einen Wert ungleich 64 besitzt bzw. falsch wenn X den Wert 64 beinhaltet.

Erweitern wir nun unsere Prüfung.



Hier werden nun zwei Aussagen überprüft, nennen wir sie A1 und A2.

Die Aussage A1 lautet „X < 5“ und die Aussage A2 lautet „X > 100“

Auch hier können wir die Aussage A1 mit einem Bit B1 und die Aussage A2 mit einem Bit B2 gleichsetzen.

Ist die Aussage A1 wahr, dann ist B1 = 1, andernfalls 0.
Ist die Aussage A2 wahr, dann ist B2 = 1, andernfalls 0.

Welchen Wert hat B1 in unserem Fall hier? Richtig, den Wert 1, da 1 ja kleiner als 5 ist.

Und welchen Wert hat B2? Richtig, den Wert 0, da 1 ja nicht größer als 100 ist.

Durch das Schlüsselwort OR haben wir es hier mit einer ODER-Verknüpfung zu tun, für die folgende Wahrheitstabelle definiert ist.

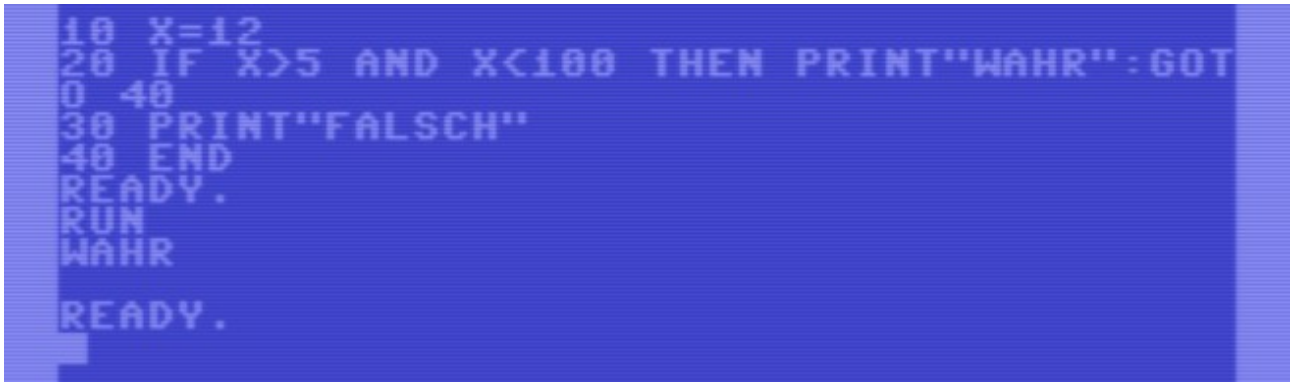
X	**B1**	**B2**	**Ergebnis der ODER-Verknüpfung**
25	0	0	0
1	1	0	1
120	0	1	1
-	1	1	1

Die gelb markierte Zeile stellt unseren Fall hier dar, die Aussage A1 ist wahr und die Aussage A2 ist falsch.

Der Ausdruck **X<5 OR X>100** stellt das Ergebnis der ODER-Verknüpfung von Aussage A1 und Aussage A2 dar. Er ist immer dann wahr, wenn eines der beiden Bits B1 und B2 den Wert 1 hat, oder beide.

Der letzte Fall kann hier jedoch nicht eintreten, da X nicht kleiner als 5 und gleichzeitig größer als 100 sein kann.

Kommen wir zur nächsten Art der Verknüpfung, der UND-Verknüpfung, welche nachfolgend durch das Schlüsselwort AND dargestellt wird.



Die Aussage A1 lautet „X>5“ und die Aussage A2 lautet „X<100“

Da X gleich 12 ist, ist die Aussage A1 wahr und auch die Aussage A2 wahr. Wenn wir nun wieder die Aussage A1 dem Bit B1 und die Aussage A2 dem Bit B2 gleichsetzen, dann hat hier sowohl B1 als auch B2 den Wert 1, da 12 ja größer als 5 und gleichzeitig aber auch kleiner als 100 ist.
Für die UND-Verknüpfung ist folgende Wahrheitstabelle definiert.

X	B1	B2	Ergebnis der UND-Verknüpfung
-	0	0	0
121	1	0	0
2	0	1	0
12	1	1	1

Der Ausdruck **X>5 AND X<100** stellt das Ergebnis der UND-Verknüpfung von Aussage A1 und Aussage A2 dar. Er ist immer nur dann wahr, wenn beide Bits B1 und B2 den Wert 1 haben.

Auch hier würde das Voranstellen des Schlüsselwortes NOT die Abfrage umkehren.

NOT (X>5 AND X<100) ist also genau dann wahr, wenn umgekehrt X nicht zwischen 5 und 100 liegt, also z.B. den Wert 2 oder den Wert 121 beinhaltet.

Neben der ODER- und der UND-Verknüpfung gibt es beispielsweise auch noch die Exklusiv-Oder Verknüpfung (XOR)

Hier ist das Ergebnis der Verknüpfung immer dann wahr, wenn die Bits B1 und B2 verschieden sind.

B1	B2	Ergebnis der XOR-Verknüpfung
0	0	0
1	0	1
0	1	1
1	1	0

TODO: Programmbeispiel für XOR

Nun möchte ich Ihnen die Assembler-Version der Abfrage X=64 aus dem ersten Programm zeigen. Ebenso wie das BASIC-Programm soll das Assembler-Programm den Text „WAHR“ oder „FALSCH“ ausgeben, je nachdem ob der jeweilige Ausdruck wahr oder falsch ist.

Doch in Assembler ist das nicht so einfach, denn wir müssen uns zunächst überlegen, wie wir diese Texte ausgeben wollen und wo die Texte überhaupt herkommen.

Ich habe in diesem Beispiel folgende Lösung umgesetzt:

Wir beginnen an der Adresse $1500, wobei wir hier jedoch nicht mit unserem Programmcode beginnen, sondern ab dieser Adresse die Texte „WAHR“ und „FALSCH“ im Speicher ablegen.

Starten wir also den SMON und geben den Befehl

M 1500 1520

ein.

```
.M 1500 1520
:1500  00 00 FF FF FF FF 00 00    ........
:1508  00 00 FF FF FF FF 00 00    ........
:1510  00 00 FF FF FF FF 00 00    ........
:1518  00 00 FF FF FF FF 00 00    ........
.
```

Nun bewegen Sie den Cursor in die erste Zeile des Dumps rechts neben der Adresse $1500 (im Screenshot ist dort der Wert $00 zu sehen)

Geben Sie nun nacheinander die Werte ein, welche auf folgendem Screenshot zu sehen sind. Wenn Sie den letzten Wert in der ersten Zeile ($41) eingegeben haben, drücken Sie die RETURN-Taste, damit die Werte übernommen werden.

Danach bewegen Sie den Cursor auf den ersten Wert in der zweiten Zeile neben der Adresse $1508 und setzen die Eingabe der Werte fort. Die Eingabe endet mit dem Wert $00 an der Adresse $150E.

Die Position des Cursors in dem Screenshot ist jene Stelle, an der Sie die RETURN-Taste drücken müssen, damit auch die Werte in dieser Zeile übernommen werden.

```
.M 1500 1520
:1500  57 41 48 52 0D 00 46 41    WAHR..FA
:1508  4C 53 43 48 0D 00 00 00    LSCH....
:1510  00 00 FF FF FF FF 00 00    ........
:1518  00 00 FF FF FF FF 00 00    ........
.
```

Nun Erstellen wir ab der Adresse $1510 folgende zwei Unterprogramme:

```
.D 1510
,1510  A2 00     LDX #00
,1512  BD 00 15  LDA 1500,X
,1515  F0 07     BEQ 151E
,1517  20 D2 FF  JSR FFD2
,151A  E8        INX
,151B  4C 12 15  JMP 1512
--------------------------------
,151E  60        RTS
--------------------------------
,151F  A2 00     LDX #00
,1521  BD 06 15  LDA 1506,X
,1524  F0 07     BEQ 152D
,1526  20 D2 FF  JSR FFD2
,1529  E8        INX
,152A  4C 21 15  JMP 1521
--------------------------------
,152D  60        RTS
--------------------------------
.
```

Das erste Unterprogramm reicht von der Adresse $1510 bis zur Adresse $151E und ist dafür zuständig, den Text „WAHR“ mit anschließendem Zeilensprung auf dem Bildschirm auszugeben.

Das zweite Unterprogramm reicht von Adresse $151F bis zur Adresse $152D und ist dafür zuständig, den Text „FALSCH“ mit anschließendem Zeilensprung auf dem Bildschirm auszugeben.

Wie funktionieren die beiden Unterprogramme? Fangen wir mit dem ersten an.

Hier wird das X Register als Schleifenzähler verwendet und mit dem Wert $00 initialisiert.

Der Text WAHR beginnt an Adresse $1500 und endet an Adresse $1505.
Ausgegeben werden jedoch nur die Zeichen bis incl. Adresse $1504, der Wert $00 an Adresse $1505 soll das Ende des Strings markieren.

Adresse $1500 = Zeichencode für W
Adresse $1501 = Zeichencode für A
Adresse $1502 = Zeichencode für H
Adresse $1503 = Zeichencode für R
Adresse $1504 = Zeichencode für Zeilensprung ($0D)
Adresse $1505 = Endemarkierung, ich habe hier den Wert $00 gewählt

Mit dem Befehl LDA 1500,X beginnt nun eine Schleife, welche nacheinander die Buchstaben des Textes WAHR in den Akkumulator lädt.

Der Aufruf JSR $FFD2 bewirkt den Aufruf einer Kernal-Routine, welche auch unter dem Namen CHROUT bekannt ist (siehe auch https://www.c64-wiki.de/wiki/CHROUT)

Sie liest den Wert im Akkumulator aus und gibt das Zeichen, welches diesem Code entspricht auf dem Bildschirm aus.

Die Schleife läuft solange bis der Wert $00 in den Akkumulator geladen wird. Dies ist nun jene 0, welche wir oben am Ende des Strings platziert haben.

Dadurch wird das Zeroflag im Statusregister auf 1 gesetzt und dies bewirkt, dass durch den Befehl BEQ 151E zum Befehl RTS an der Adresse $151E verzweigt wird, was den Rücksprung aus dem Unterprogramm zum Aufrufer bewirkt.

Das zweite Unterprogramm funktioniert genau gleich, nur dass hier der Text FALSCH mit anschließendem Zeilensprung ausgegeben wird.

Der Text FALSCH beginnt an Adresse $1506 und endet an Adresse $150D.
Ausgegeben werden jedoch nur die Zeichen bis incl. Adresse $150C, der Wert $00 an Adresse $150D markiert wieder das Ende des Strings.

Adresse $1506 = Zeichencode für F
Adresse $1507 = Zeichencode für A
Adresse $1508 = Zeichencode für L
Adresse $1509 = Zeichencode für S
Adresse $150A = Zeichencode für C
Adresse $150B = Zeichencode für H
Adresse $150C = Zeichencode für Zeilensprung ($0D)
Adresse $150D = Endemarkierung analog zum erstem Unterprogramm

Zugegeben, man wäre sicher auch mit einem Unterprogramm ausgekommen, da sich die beiden ja nur dadurch unterscheiden, welchen Text sie ausgeben. Doch das soll uns an dieser Stelle nicht weiter stören.

Erstellen wir nun ab Adresse $152E das „Hauptprogramm“

Hier wird zunächst der Inhalt der Adresse $030D (dezimal 781) in das X Register geladen.

Kommt Ihnen diese Adresse 781 bekannt vor? Genau, es ist jene Speicherstelle, in die wir einen Wert schreiben können, den der SYS Befehl vor dem Start des Maschinenprogramms in das X Register lädt.

Vor dem Start des Programms über den Befehl SYS müssen wir daher einen Wert in diese Speicherstelle schreiben.

Als nächstes wird durch den Befehl CPX #40 der Inhalt des X Registers mit dem Wert $40 (dezimal 64) verglichen.

Steht im X Register der Wert 64, dann wird durch die Gleichheit das Zeroflag im Statusregister gesetzt, wodurch durch den Befehl BEQ an die Programmadresse $153B verzweigt wird.
Dort wird das Unterprogramm aufgerufen, welches den Text WAHR ausgibt und anschließend das Programm beendet.

Steht im X Register jedoch nicht der Wert 64, dann wird das Zeroflag im Statusregister nicht gesetzt und es geht mit dem Befehl nach dem Befehl BEQ weiter. Dort wird das Unterprogramm aufgerufen, welches den Text FALSCH auf dem Bildschirm ausgibt. Dann wird durch den Befehl JMP 153E zum Befehl RTS gesprungen, wodurch das Programm beendet wird.

Nun können wir unser Programm starten, das an der Adresse $152E (dezimal 5422) beginnt.

Wechseln Sie daher mit dem Befehl „X“ zu Basic, führen zuerst den Befehl

POKE 781,64

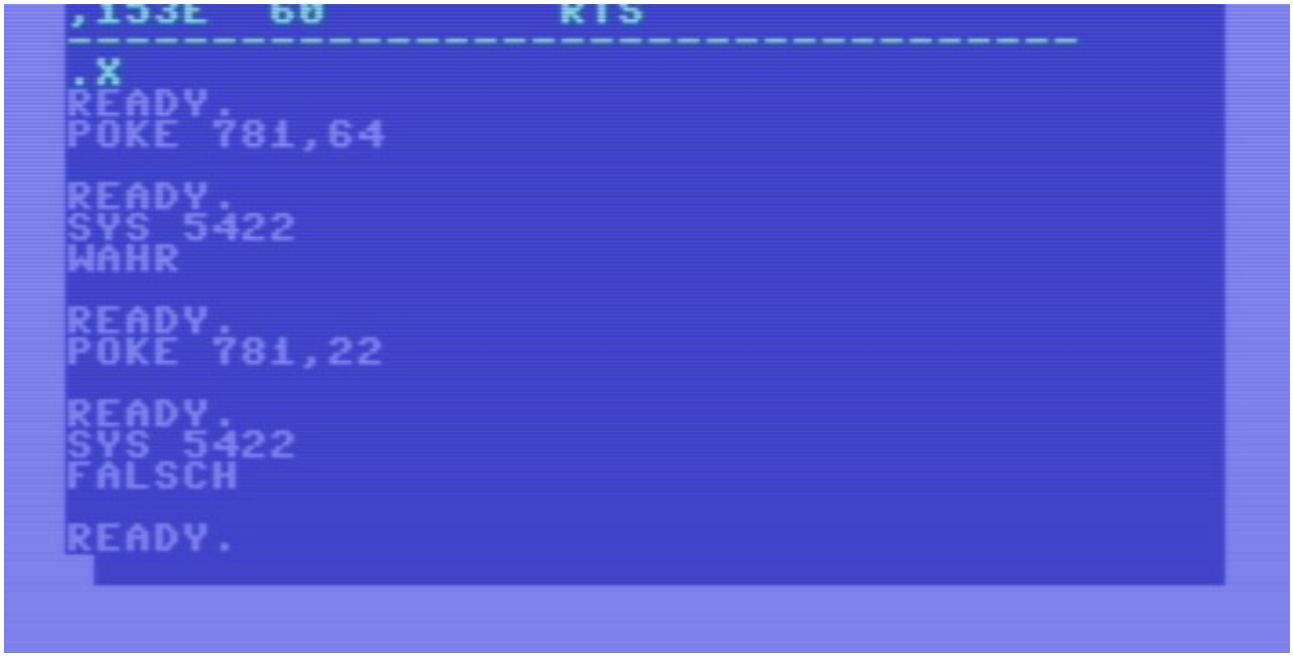
aus. Dadurch geben wir an, dass der Befehl SYS den Wert 64 in das X Register schreiben soll, bevor er unser Programm startet.

Danach starten Sie das Programm mit SYS 5422 und Sie sehen, dass der Text WAHR ausgegeben wird. Klar, denn zu Beginn des Programms steht nun der Wert 64 im X Register und durch den Befehl CPX #40 wird geprüft, ob dieses den Wert 64 enthält.

Dann führen wir den Befehl

POKE 781,22

aus. Dadurch wird vor dem Programmstart der Wert 22 in das X Register geschrieben, was dann zur Ausgabe der Meldung FALSCH führt, weil 22 ungleich 64 ist.



Nun wollen wir uns die Assembler-Umsetzung des Ausdrucks X>5 AND X<100 ansehen.

In dieser Umsetzung lernen wir auch gleich ein paar neue Assembler-Befehle und Flags im Status-Register kennen.

Glücklicherweise können wir das Anlegen der Texte und die beiden Unterprogramme unverändert übernehmen, sodass wir uns nur mehr um die Umsetzung der Prüfung X>5 AND X<100 kümmern müssen.

Beginnen wir mit der Umsetzung des Ausdrucks x>5

Hier wird zuerst der Inhalt der Speicherstelle 781 in das X Register eingelesen.

Dann wird der Inhalt des X Registers durch den Befehl CPX mit dem Wert 5 verglichen.
Ist er gleich 5, dann ist die Bedingung x>5 nicht erfüllt und es wird zur Programmadresse $1537 gesprungen, an der das Unterprogramm, das den Text FALSCH ausgibt, aufgerufen wird und dann durch den Sprung zum Befehl RTS das Programm beendet wird.

Ist der Inhalt des X Registers jedoch nicht gleich 5, wird durch den Befehl BCS an jene Stelle gesprungen, an der das Unterprogramm aufgerufen wird, das den Text WAHR ausgibt.

Anschließend wird das Programm durch den Befehl RTS beendet.

Erklärung zum Befehl BCS (Branch on Carry Set)

Dieser Befehl verzweigt an die angegebene Programmstelle, wenn das Carry Flag gesetzt ist.

Warum überprüfen wir an dieser Stelle das Carry Flag?

Dies hängt mit dem Ergebnis des Befehls CPY zusammen. Die CPU subtrahiert den übergebenen Wert vom Inhalt des X Registers und wenn die Differenz > 0 ist, dann zeigt dies der CPU an, dass der Inhalt des X Registers größer als der Wert ist, den man nach dem CPX Befehl als Vergleichswert angegeben hat. In diesem Fall wird dann das Carry Flag auf den Wert 1 gesetzt.

Angenommen, wir tragen mit dem Befehl POKE 781,12 den Wert 12 als Startwert für das X Register ein.

Nun kommt der Befehl CPX #05 an die Reihe. Die CPU subtrahiert nun 5 vom Inhalt des X Registers, rechnet also 12 – 5, was den Wert 7 ergibt. Dieser Wert ist > 0 und daher wird das Carry Flag gesetzt.

Und nun versuchen wir die Kombination mit dem Ausdruck x<100

Auch hier findet zunächst ein Vergleich des X Registers mit dem Wert 5 statt.

Ist er gleich 5, dann ist die Bedingung x > 5 nicht erfüllt und es wird zur Programmadresse $1540 gesprungen, an der das Unterprogramm, das den Text FALSCH ausgibt, aufgerufen wird und dann durch den Sprung zum Befehl RTS an der Programmadresse $1549 das Programm beendet wird.

Ist der Inhalt des X Registers jedoch > 5, dann wird durch den Befehl BCS zur Programmadresse $153A verzweigt, an der die Überprüfung beginnt, ob x < 100 ist.

Dort wird zuerst durch den Befehl BEQ geprüft, ob der Inhalt des X Registers gleich 100 ist.
Ist dies der Fall wird, wird zur Programmadresse $1540 gesprungen, an der das Unterprogramm, das den Text FALSCH ausgibt, aufgerufen wird und dann durch den Sprung zum Befehl RTS an der Programmadresse $1549 das Programm beendet wird.

Durch den Befehl CPX #64 zieht die CPU den Wert $64 (dezimal 100) vom Inhalt des X Registers (dezimal 12) ab, was zu dem Resultat -88 führt. Dieser Wert ist < 0 und daher wird das Carry Flag nicht gesetzt.

Deshalb wird durch den Befehl BCC an die Programmadresse $1546 gesprungen, an der das Unterprogramm aufgerufen wird, das den Text WAHR ausgibt, und das Programm anschließend durch den Befehl RTS beendet wird.

Erklärung zum Befehl BCC (Branch on Carry Clear)

Dieser Befehl verzweigt an die angegebene Programmstelle, wenn das Carry Flag nicht gesetzt ist, also den Wert 0 enthält.

Abschließen möchte ich dieses Kapitel mit einem wichtigen Thema, das uns in den nächsten Kapiteln oft begegnen wird.

Es geht dabei um die Frage, wie man in einem Zahlenwert ein oder mehrere Bits gezielt auf 1 oder umgekehrt auf 0 setzen kann. Hier benötigten wir nun die eingangs beschriebenen logischen Bit-Verknüpfungen wie ODER (AND) und UND (OR)

Löschen von Bits

Nehmen wir als Beispiel die binäre Zahl %10110010 (hexadezimal $B2, dezimal 178)

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

Nun wollen wir die Bits an den Positionen 1 und 4 auf 0 setzen.

Schritt 1:

Wir bilden eine Binärzahl, die genau an diesen Positionen ein gesetztes Bit enthält.

In diesem Fall lautet diese Binärzahl: %00010010

Schritt 2:

Dann drehen wir alle Bits in dieser Binärzahl um, d.h. wir bilden das Einerkomplement.

Ergebnis: %11101101

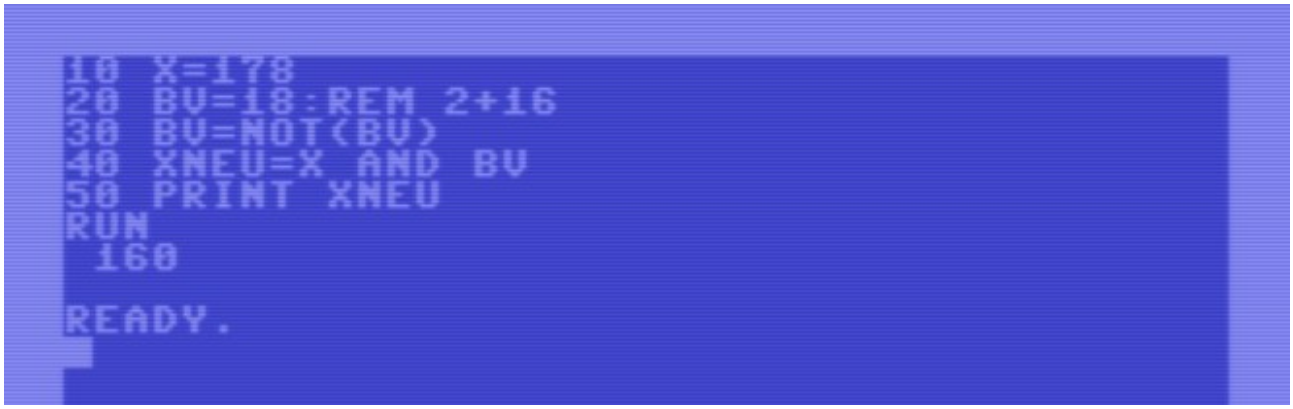
Schritt 3:

Wir verknüpfen die Zahl von der wir ausgegangen sind mit diesem Ergebnis durch eine UND-Verknüpfung.

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

In der untersten Zeile sieht man das Ergebnis der UND-Verknüpfung, wobei die beiden Bits 1 und 4 nun den Wert 0 enthalten.

Spielen wir das alles mal in BASIC nach.



Hier weisen wir X den dezimalen Wert 178 zu, denn diese Zahl haben wir hier als Beispiel verwendet.

Wir wollen die Bits an den Positionen 1 und 4 zurücksetzen, d.h. wir bilden in Zeile 20 die Binärzahl welche an diesen Bitpositionen eine 1 enthält.

Wertigkeit an Bitposition 1 = 2 hoch 1 = 2
Wertigkeit an Bitposition 4 = 2 hoch 4 = 16

Die Summe lautet 18, daher weisen wir der Variablen BV diesen Wert zu. In der nächsten Zeile drehen wir durch das Schlüsselwort NOT alle Bits in der Variablen BV um. Und schließlich weisen wir der Variablen XNEU das Ergebnis der UND-Verknüpfung zu und geben das Ergebnis aus.

Die dezimale Zahl 160 entspricht der Binärzahl %10100000 und dies deckt sich exakt mit dem Ergebnis in der letzten Zeile inder obigen Tabelle.

Und jetzt probieren wir das mal in Assembler und geben ab Adresse $1501 folgenden Code ein:

Hier lernen wir gleich zwei neue Assembler-Befehle kennen: EOR und AND

Zunächst laden wir den dezimalen Wert 18 ($12) in den Akkumulator. Das ist jener Wert der an Bitposition 1 und 4 eine 1 enthält.

Durch den Befehl EOR führen wir eine Exklusiv-Oder Verknüpfung (XOR) zwischen dem Inhalt des Akkumulators und einem Zahlenwert durch (hier $FF)

Warum machen wir das? Wir haben in Assembler leider keinen direkten Befehl, der die Bits eines Zahlenwertes umdreht, daher wählen wir den Umweg über die XOR-Verknüpfung mit dem Wert $FF (dezimal 255), welcher zum selben Ergebnis führt.

Überprüfen wir das mal:

Zu Beginn steht der Wert $12 (dezimal 18, binär %00010010) im Akkumulator.

Nun führen wir eine XOR-Verknüpfung zwischen diesem Wert und dem Wertt %11111111 (hexadezimal $FF, dezimal 255) durch.

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

Wenn wir die Bits in der orangen Zeile nun mit denen in der grünen Zeile vergleichen, sehen wir, dass die Bits aus der orangen Zeile umgedreht wurden.

Wir erinnern uns, das Ergebnis einer XOR-Verknüpfung zwischen den Bits B1 und B2 ist genau dann 1, wenn B1 und B2 verschiedene Werte haben.

Also dann, wenn B1 = 1 und B2 = 0 bzw. Umgekehrt wenn B1 = 0 und B2 = 1.

Wir benutzen die Speicherstelle $1500 um dieses Ergebnis zwischenzuspeichern, denn im nächsten Befehl wird der Akkumulator mit dem Wert $B2 (dezimal 178) geladen.

Nun führen wir durch den Befehl AND eine UND-Verknüpfung zwischen dem Inhalt des Akkumulators und dem Inhalt der Speicherstelle $1500 durch.

Das Ergebnis dieser Verknüpfung sehen wir dann im Akkumulator ($A0, dezimal 160), welches identisch zu der Ausgabe des BASIC-Programms ist.

Setzen von Bits

Nun wollen wir uns ansehen, wie man umgekehrt ein oder mehrere Bits auf den Wert 1 setzen kann.

Nehmen wir als Beispiel wieder die binäre Zahl %10110010 (hexadezimal $B2, dezimal 178)

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

Setzen wir doch mal die Bits an den Positionen 2 und 6 auf 1.

Schritt 1:

Wir bilden eine Binärzahl, die genau an diesen Positionen ein gesetztes Bit enthält.

In diesem Fall lautet diese Binärzahl: %01000100

Schritt 2:

Wir verknüpfen die Zahl von der wir ausgegangen sind mit diesem Ergebnis durch eine ODER-Verknüpfung.

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

In der untersten Zeile sieht man das Ergebnis der ODER-Verknüpfung, wobei die beiden Bits 2 und 6 nun den Wert 1 enthalten.

Sehen wir uns nun wieder wie vorhin die Umsetzung in BASIC an.

Hier weisen wir X wieder den dezimalen Wert 178 zu, denn diese Zahl haben wir hier als Beispiel verwendet.

Wir wollen die Bits an den Positionen 2 und 6 auf den Wert 1 setzen.

Wertigkeit an Bitposition 2 = 2 hoch 2 = 4
Wertigkeit an Bitposition 6 = 2 hoch 6 = 64

Die Summe lautet 68, daher weisen wir der Variablen BV diesen Wert zu.
Dann weisen wir der Variablen XNEU das Ergebnis der ODER-Verknüpfung zu und geben das Ergebnis aus.

Die dezimale Zahl 246 entspricht der Binärzahl %11110110 und dies deckt sich exakt mit dem Ergebnis in der letzten Zeile in der obigen Tabelle.

Und nun wieder die Umsetzung in Assembler.

Auch hier lernen wir wieder einen neuen Assembler-Befehl kennen, den Befehl ORA.

Hier laden wir unseren Ausgangswert $B2 (dezimal 178) in den Akkumulator und durch den Befehl ORA gefolgt vom hexadezimalen Wert $44 (dezimal 68) wird eine ODER-Verknüpfung zwischen dem Inhalt des Akkumulators und dem Wert $44 durchgeführt.

Das Ergebnis dieser Verknüpfung sehen wir dann im Akkumulator ($F6, dezimal 246), welches identisch zu der Ausgabe des BASIC-Programms ist.