SKRIPTUM

PROGRAMMIEREN VON MIKROCOMPUTERN

CPU 6502

G. Eisenack

N. Nowak

PROGRAMMIEREN VON MIKROCOMPUTERN

von

G. Eisenack

1980

Fachbereich Elektrotechnik
Fachhochschule Bielefeld

N. Nowak, Spindelstr. 856

Vorwort

Das vorliegende Skriptum "Programmieren von Mikrocomputern" besteht aus zwei Teilen: Die "Einführung in das Programmieren von Mikrocomputern" enthält den größten Teil des Vorlesungsstoffs meiner Wahlveranstaltung "Mikrorechner" und setzt gewisse Basiskenntnisse der Datenverarbeitung voraus.
Der zweite Teil - "Grundprogramme" - beschreibt eine Anzahl grundlegender Programme, die in der Mikrocomputer-Technik häufig verwendet werden. Die Beschreibung der Programme in Form von Ablaufplänen oder Strukturdiagrammen erfolgt prozessorunabhängig.
Bei der Auswahl des Mikroprozessortyps 6502 (für Teil I) spielte eine wesentliche Rolle, daß für den 6502 frühzeitig ein preiswerter Mikrocomputer (KIM) mit Bildschirmanschluß und Externspeicher-Möglichkeit vorhanden war und - was im Bereich der Datenverarbeitung selten ist - gute Handbücher für diesen Prozessor existierten. Der 6502 ist nicht der "stärkste" Mikroprozessor, aber wegen seines übersichtlichen Befehlsvorrats für didaktische Zwecke gut geeignet.
In vielen Grundkonzepten der Programmierung sind die Mikroprozessoren sehr ähnlich. Auf Unterschiede zu anderen Mikroprozessoren (6800, 8080/85, Z80) wird in den Bemerkungen hingewiesen.

Bielefeld, im September 1980 G. Eisenack

Inhaltsverzeichnis

Teil I: Einführung in das Programmieren von Mikrocomputern

Teil II: Grundprogramme

TEIL I

Einführung in das

PROGRAMMIEREN VON MIKROCOMPUTERN

1. Aufbau eines Mikrocomputers

Mikrocomputer und herkömmliche Computer arbeiten nach dem gleichen Prinzip.
Daher wollen wir uns zunächst noch einmal an die wesentlichen Elemente einer DV-Anlage erinnern und dabei einige Begriffe amerikanischen Ursprungs einführen, die zum Verständnis der Literatur über Mikrocomputer wesentlich sind.

1.1 Grundbegriffe

Die Zentraleinheit einer DV-Anlage besteht aus
dem Prozessor (= CPU = Central Processing Unit),
dem Hauptspeicher und
den Ein/Ausgabe-Werken (falls vorhanden).

Der Prozessor enthält das Leitwerk und das Rechenwerk.
Das Leitwerk sorgt für die Ausführung des Befehls (Instruction), dessen Adresse im
Befehlszähler (= PC = Program Counter) steht.
Das Leitwerk bedient sich des Rechenwerks, dessen Hauptbestandteil die
Arithmetisch-Logische Einheit (= ALU = Arithmetic Logical Unit) ist, mit der arithmetische und logische Verknüpfungen durchgeführt werden.

Der Speicher (Memory) des Hauptspeichers besteht heutzutage aus Halbleiterspeichern. Ein Halbleiterspeicher, der sowohl gelesen als auch beschrieben werden kann, heißt
RAM (Random Access Memory, d.h. Speicher mit wahlfreiem Zugriff).
Ein Festspeicher, der nur gelesen werden kann, ist zwar in der Regel auch ein Speicher mit wahlfreiem Zugriff, wird aber
ROM (Read Only Memory) genannt.
PROM (Programmable ROM) ist ein Festspeicher, der vom Anwender programmiert werden kann; ist das PROM sogar wieder löschbar - meist mit UV-Licht -, so spricht man von einem
EPROM (Erasable PROM).

Bei kleinen Systemen nicht unbedingt erforderlich, aber meist sehr nützlich, sind Externspeicher. Im einfachsten Fall wird ein handelsüblicher Kassettenrecorder verwendet.

1.2 Zentraleinheit eines Mikrocomputers

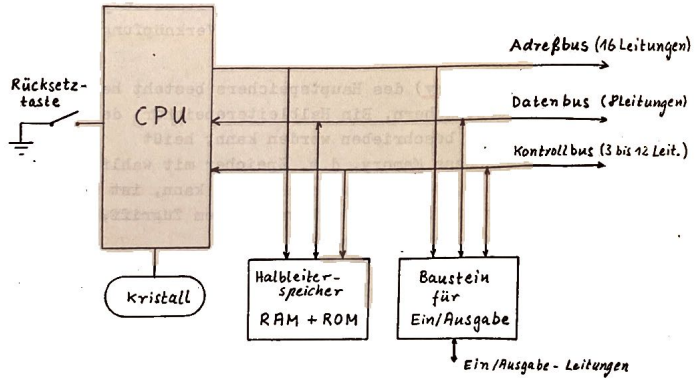
Mikroprozessoren sind Prozessoren (Prozessor = Leitwerk + Rechenwerk = CPU), die in einer - oder sehr wenigen - integrierten Schaltung(en) realisiert sind.

Mikrocomputer = Computer, dessen Prozessor ein Mikroprozessor ist.

Heute wird meistens mit Mikroprozessoren gearbeitet, die eine Wortlänge von 8 bit = 1 Byte und 16 bit-Adressen haben. Zu diesen Mikroprozessoren gehören z.B. die "großen Vier":

8080/85	von Intel
6800	von Motorola
6502	von MOS-Technology
Z80	von Zilog

Die Zentraleinheit mit einem solchen Mikroprozessor sieht nun folgendermaßen aus:



Die 8 bit-CPU hat 8 Anschlüsse (Pins) für die Datenleitungen, die zusammen den sogenannten Datenbus ergeben. An diese Leitungen werden die Speicher (ROM + RAM) und Bausteine für die Ein-/Ausgabe angeschlossen.
Die CPU hat ferner 16 Anschlüsse für den "Adreßbus", an den ebenfalls die Speicher und eventuell auch die Ein-/Ausgabe-Bausteine angeschlossen werden.
Über den Kontrollbus, der bei den verschiedenen Mikroprozessoren auch recht verschiedene Kontrollinformationen überträgt, werden von der CPU einige Kontrollsignale ausgesendet und manchmal auch empfangen.
Typisches Beispiel für eine Kontrollinformation ist die Mitteilung der CPU an den Hauptspeicher, ob die CPU lesen oder schreiben möchte (R/W - Signal).

Der eingezeichnete Kristall dient zur Definition der Taktfrequenz (z.B. 1 MHz) und die Rücksetztaste ist erforderlich, um die CPU in einen definierten Anfangszustand zu versetzen.

Ganze Zentraleinheiten mit beschränktem Speicher (typisch: 2 kB ROM + 64 Byte RAM) werden heute von vielen Herstellern als Ein-Chip-Mikrocomputer angeboten.
Sofern der ROM-Bereich nicht programmierbar ist - und das ist in der Regel der Fall - wird das Programm beim Herstellungsprozeß des Ein-Chip-Mikrocomputers in den ROM-Bereich gebracht. Dies ist bei Verwendung geringer Stückzahlen ziemlich teuer.
Es gibt aber inzwischen auch Ein-Chip-Mikrocomputer bei denen das ROM durch ein EPROM ersetzt ist (Z.B. Intel 8748, 1 kB EPROM + 64 byte RAM, Preis 1979:ca. 170 DM).

Und nun noch einige Bemerkungen zum Mikroprozessor-Markt.
Der erste Mikroprozessor "Intel 4004" wurde 1971 auf den Markt gebracht. Er arbeitete mit einer Wortlänge von 4 bit.
Es folgte der 8 bit-Mikroprozessor 8008 und 1974 der bekannte 8080 von Intel. Seit 1975 sind auch 16 bit-CPUs erhältlich.

1978 wurden bereits 14 Millionen Mikroprozessoren verkauft. 13 Millionen davon im 4 bit-Bereich und 1 Million im * 8 bit-Bereich. Die Preise für 8 bit-Mikroprozessoren liegen je nach Stückzahl zwischen 5 DM und 50 DM (1979).

1.3 Betriebssystem

Jeder Computer, in den Programme eingegeben werden sollen, benötigt bereits ein Programm, mit dem man Programme eingeben kann. Dieses Eingabeprogramm muß in einem Festspeicher stehen (heute im ROM, früher fest verdrahtet).
Nach dem Drücken der Rücksetztaste (Reset) wird die Anfangsadresse des Eingabe-Programms in den Befehlzähler geladen und der Computer beginnt zu laufen (Systemstart).
Ein laufendes Programm kann jederzeit mit der Rücksetztaste unterbrochen werden; anschließend läuft das Eingabe-Programm.

Dieses Eingabe-Programm wird meist zusammen mit anderen nützlichen Programmen in einem ROM bzw. PROM gespeichert und bildet mit diesen ein kleines Betriebssystem. Dieses kleine Betriebssystem wird häufig Monitor genannt.
Der Monitor gestattet es, Speicherzellen anzuzeigen und deren Inhalte zu verändern (Programmeingabe). Ferner kann der Befehlszähler verändert werden (Programmstart).

Ist ein Externspeicher (z.B. Kassettenrecorder oder Floppy disk) vorgesehen, so enthält der Monitor auch Programme zum Lesen und Abspeichern von Programmen.
Bei größeren Systemen wird der Monitor durch weitere Programme, die auf einem Externspeicher stehen, zu einem umfangreicheren Betriebssystem ergänzt.

Beispiel: Beim KIM-Mikrocomputer besteht das "Betriebssystem" aus 2 kB Programmen mit folgenden Fähigkeiten:

---

*Quelle: Zeitschrift Byte 7/79, p.99. Danach wurden 1978 am meisten verkauft: 4 bit TMS 1000 (Ein-Chip-Mikrocomputer)
8 bit 6502

Anzeigen und Ändern von Zellen
Starten von Programmen
Ausgeben von Speicherbereichen auf Kassette oder Lochstreifen
Einlesen von Speicherbereichen von Kassette oder Lochstreifen
Einzelschrittdurchführung von Programmen

Bemerkung 1: Soll der Mikrocomputer nur für einen Zweck eingesetzt werden, wird kein Eingabe-Programm benötigt. Das erforderliche Programm steht im ROM und wird durch Drücken der Rücksetztaste gestartet (oder durch Einschalten des Geräts).

Bemerkung 2: Der Zusammenbau eines Mikrocomputers (in den Programme eingegeben werden sollen) aus seinen Einzelkomponeneten ist jedenfalls für das erste Exemplar ein echtes Problem, da ja zum Arbeiten mit dem Mikrocomputer ein Eingabe-Programm erforderlich ist (das Eingabe-Programm kann also auf dem zusammengebauten System nicht getestet werden).
Deshalb bieten einige Hersteller ein ROM an, in dem ein bereits ausgetesteter Monitor abgespeichert ist. Der TIM-Monitor für den 6502 und Mikbug für den 6800 sind hierfür Beispiele.

2. Grundkonzepte der Programmierung

Wir werden uns bei der Programmierung mit dem Mikroprozessor 6502 beschäftigen, der sich besonders durch seinen sehr übersichtlichen Befehlsvorrat auszeichnet. In den Grundkonzepten sind die Mikroprozessoren aber sehr ähnlich. Bei größeren Abweichungen zu anderen Mikroprozessoren (es werden 6800, 8080, Z80 berücksichtigt) werden Anmerkungen gemacht.

2.1 Befehlsstruktur von Mikroprozessoren (8 bit)

Die Wortlänge der 8 bit-Mikroprozessoren ist - wie der Name sagt - 8 bit, also:
Wortlänge = 8 bit = 1 Byte
Für die Adressen werden üblicherweise 16 bit verwendet:
Adresse = 16 bit = 2 Bytes
Somit können mit diesen Mikroprozessoren 2^{16} Bytes = 64 kB adressiert werden (Siemens 305: 16 kWorte = 48 kB).

Den Inhalt eines Bytes gibt man sedezimal (=hexadezimal) an,

z.B. 8A = 10001010

Auch Adressen werden sedezimal geschrieben, z.B.

OE5F

Befehle im Mikrocomputer haben keine feste Länge. Es gibt 1 Byte-, 2 Byte- und 3 Byte-Befehle (beim Z80 auch noch 4 Byte-Befehle).

Beispiele für Befehle

1. a) Lade den Akkumulator mit dem Inhalt einer Zelle (LDA)

AD 5F OE — 3-Byte-Befehl
Op-Code Adreßteil (gibt Adresse OE5F an)

AD ist der Operationscode, also der Operationsteil des Befehls.
Die 16 bit-Adresse der Zelle wird eigenartigerweise in umgekehrter Reihenfolge im Befehl angegeben.

b) Lade den Akkumulator mit der Sedezimalzahl 6D (LDA)

A9 6D — 2 Byte-Befehl
Op-Code Sedezimalzahl

2. Verschiebe Inhalt des Akkus um eine Stelle nach links (ASL)

0A — 1 Byte-Befehl
Operationscode

Die Operationscodes sind der Befehlstabelle für den 6502 (Anhang) zu entnehmen:
In der Zeile LDA - Abkürzung für Load Accumulator - sind die Operationscodes AD (in Spalte "absolute") und A9 (in Spalte "immediate") zu finden.
In der Zeile ASL - Abkürzung für Arithmetic Shift Left - der Operationscode 0A (in Spalte "accumulator").
Die große Anzahl von Spalten ergibt sich durch die Vielzahl von Adressierungsmöglichkeiten, auf die wir später eingehen werden.

Die 8 bit-Mikroprozessoren und auch die meisten 16 bit-Mikroprozessoren sind Ein-Adreß-Maschinen, d.h. im Befehlswort wird höchstens ein Operand, der im Hauptspeicher steht, adressiert. Falls im Befehl ein zweiter Operand erforderlich ist (z.B. bei der Addition), steht dieser entweder im Akkumulator oder in einem andren Register der CPU.

Bemerkung: Die Eigenart, im Adreßteil des Befehls zunächst das niederwertige, dann das höherwertige Byte der Adresse anzugeben, teilt der 6502 mit den Mikroprozessoren 8080 und Z80. Beim 6800 wird die Adresse dageger "normal" angegeben.

2.2 Einführende Programme (Addition)

Zur Einführung soll ein Programm geschrieben werden, welches zwei Hex-Zahlen (=Hexadezimal-Zahlen = Sedezimal-Zahlen) addiert.
Erinnern wir noch einmal an eine Dualaddition

z.B.
```
  11001010
  11100101
 110101111
 ↑Übertrag = Carry
```

Das Carry-Bit wird bei der Addition in einen Spezialspeicher der CPU eingetragen (Carry-Bit = 1, falls Übertrag; Carry-Bit = 0, falls kein Übertrag) und bei der nächsten Addition noch zuaddiert. Soll also durch das Carry-Bit kein Fehler bei der Addition entstehen, so ist vor der Addition Carry-Bit = 0 zu setzen. Dies geschieht durch den Befehl CLC.

Aufgabe: 1. Summand in 0000 (0000 Adresse einer Speicherzelle)
2. Summand in 0001

Summe nach 0002

Wir wollen das Programm, beginnend bei Adresse 0200 (Angabe in Hex!) abspeichern.

Adresse	Befehl	Symbol	
0200	18	CLC	(Clear Carry)
0201	AD 00 00	LDA	(Load Accumulator)
0204	6D 01 00	ADC	(Add with Carry)
0207	8D 02 00	STA	(Store Accumulator)
020A	4C 4F 1C	JMP	(Jump)

Nach dem Löschen desCarry-Bits wird der 1. Summand in den Akkumulator geladen. Anschließend der 2. Summand zum Akkumulator (dual) addiert, wobei das Ergebnis im Akkumulator steht. Deshalb wird durch STA der Akkumulator-Inhalt in die Ergebnis-Zelle gebracht. Der letzte Befehl ist ein Sprung in das Eingabe-Programm des Monitors.

Das Programm wird mit Hilfe des Monitors eingegeben; ebenfalls die beiden Summanden. Anschließend wird das Programm mit Hilfe des Monitors bei Adresse 0200 gestartet.

Nach Durchlauf des Programms kann der Monitor wieder bedient werden, da ja in das Eingabe-Programm zurückgesprungen wurde.
Beim Fehlen des Sprungs in das Eingabe-Programm würden nach STA völlig unkontrollierbare Befehle ablaufen (das System "bricht zusammen") und meist ist nur mit der Rücksetztaste aus diesem Zustand herauszukommen.

Im zweiten Programmbeispiel wollen wir Zahlen, die sich in 2 Bytes darstellen lassen (0-65535) addieren. Wie im ersten Beispiel soll ein Überschreiten des Zahlenbereichs unberücksichtigt bleiben.

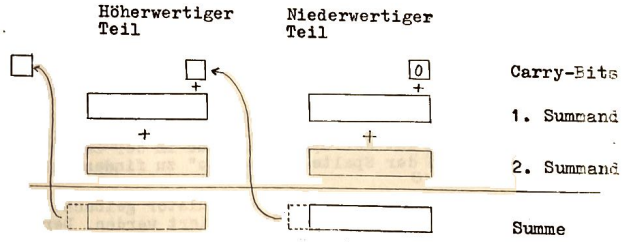
2-Byte-Addition 1. Summand in 0010,0011
2. Summand in 0012,0013

Summe nach 0014,0015

Adresse	Maschinen-Befehl	Symbolischer Befehl	
0300	18	CLC	
0301	AD 11 00	LDA $0011	
0304	6D 13 00	ADC $0013	
0307	8D 15 00	STA $0015	
030A	AD 10 00	LDA $0010	
030D	6D 12 00	ADC $0012	
0310	8D 14 00	STA $0014	
0313	4C	JMP $....	Sprung in Monitor

In den ersten vier Befehlen wird der niederwertige Teil der Summanden addiert und die Summe abgespeichert. Anschließend werden die höherwertigen Teile der Summanden addiert, wobei ein Übertrag berücksichtigt wird.

Folgendes Bild soll die 2 Byte-Addition verdeutlichen:



Übertrag, der als Carry-Bit erschein

Wir geben bereits jetzt die Befehle auch in symbolischer Form an, so wie sie später von einem Assembler bearbeitet werden können. $ deutet an, daß die Adresse als Hex-Zahl angegeben wird.

Häufig möchte man Dezimalzahlen und nicht Dualzahlen verarbeiten. Dazu dient die BCD-Darstellung (Binary Coded Decimal Code) von Dezimalzahlen. Jede Dezimalziffer wird als 4 Bit-Dualzahl verschlüsselt. In einem Byte lassen sich also 2-stellige Dezimalzahlen darstellen.

Beispiel: 93 wird durch 10010011, also durch die Hex-Zahl 93 dargestellt.

Um Dezimalzahlen in BCD-Darstellung addieren bzw. subtrahieren zu können, muß das Rechenwerk auf "Dezimalarithmetik" umschalten. Dies geschieht mit dem Befehl SED (Set Decimal Mode) und wird mit dem Befehl CLD (Clear Decimal Mode) rückgängig gemacht. Das Rechenwerk kann nur für die Addition und die Subtraktion auf Dezimalarithmetik umgeschaltet werden.

Dezimaladdition 1. Summand in 0000
2. Summand in 0001
Summe in 0002,0003

In diesem Beispiel soll das Überschreiten des 2-ziffrigen Zahlenbereichs berücksichtigt werden.

```
0200   F8          SED          (Set Decimal Mode)
0201   18          CLC
0202   AD 00 00    LDA $0000
0205   6D 01 00    ADC $0001
0208   8D 03 00    STA $0003
020B   A9 00       LDA #00      (Lösche Akku)
020D   69 00       ADC #00      (Addiere 00 zum Akku)
020F   8D 02 00    STA $0002
0212   D8          CLD          (Clear Decimal Mode)
0213   4C .. ..    JMP $....    Sprung in Monitor
```

#00 bedeutet, daß 00 nicht als Adresse des Operanden zu deuten ist, sondern der Operand (als Hex-Zahl) selbst ist. In solchen Fällen ist der Operationscode in der Befehlsliste des 6502 in der Spalte "Immediate" zu finden.

LDA #00
ADC #00 bewirkt also, daß der Akkumulator gelöscht wird und zu diesem 00 und das Carry-Bit addiert werden. Der Akkumulator enthält somit einen eventuellen Übertrag.

Bemerkung: 1. Bei den Mikroprozessoren 8080 und Z80 wird die Operandenadresse (z.B. bei der Addition) in der Regel nicht direkt im Befehl angegeben, sondern muß vorher in ein Registerpaar der CPU eingegeben werden.
2. Für die 2 Byte-Addition ist beim 8080 und Z80 ein Befehl vorhanden.
3. Die Dezimaladdition bei den Mikroprozessoren 8080, Z80 und 6800 wird nicht durch Umschalten des Rechenwerks erreicht, sondern nach einer Dualaddition wird durch einen Sonderbefehl (Decimal Adjust Accumulator) das Ergebnis korrigiert.

2.3 Logische Befehle, Schiebebefehle

Logische Befehle

Die logischen Grundoperationen Konjunktion (= UND = AND), Disjunktion (= ODER = OR) und Antivalenz (= exklusives ODER) sind im Befehlssatz fast aller Mikroprozessoren enthalten, denn durch Verknüpfung dieser Grundoperationen läßt sich jede logische Funktion herstellen.

Die Wahrheitstafeln für diese Grundfunktionen sind:

UND	0	1
0	0	0
1	0	1

ODER	0	1
0	0	1
1	1	1

Exklusives ODER	0	1
0	0	1
1	1	0

Nehmen wir an

Akkumulatorinhalt:	10101101
Inhalt der Zelle 3E4F:	11001001

, so ersieht man die bitweise Wirkung der Befehle AND, ORA (OR with Accumulator) und EOR (Exclusive OR) aus folgender Tabelle:

Maschinenbefehl	symb. Befehl	Ergebnis im Akkumulator
2D 4F 3E	AND $3E4F	10001001
0D 4F 3E	ORA $3E4F	11101101
4D 4F 3E	EOR $3E4F	01100100

Die Negation des Akkumulatorinhalts (= 1er-Komplement = Vertauschen von Nullen und Einsen) erreicht man durch:

EOR #FF (Maschinenbefehl: 49 FF)

Schiebebefehle (Shift-Befehle), Rotationen

1. Rechtsverschiebung LSR (Logical Shift Right)

Bei der logischen Rechtsverschiebung wird der Inhalt der adressierten Speicherzelle um 1 Bit nach rechts verschoben, wobei von links eine 0 nachrückt und das herausgeschobene Bit im Carry-Bit aufgefangen wird, z.B.:

Speicherplatz-Inhalt vor LSR: 10110111 □
Speicherplatz-Inhalt nach LSR: 01011011 [1] — Carry-Bit
Bit 7

Bemerkung: a) Rechtsverschiebung bewirkt bei Dualzahlen eine Division durch 2.

b) Im Gegensatz zur logischen Rechtsverschiebung wird bei der "arithmetischen Rechtsverschiebung" Bit 7 dem Bit 6 angeglichen. Dies entspricht der Division durch 2 bei Dualzahlen mit Vorzeichen. (Beim 6502 und 8080 nicht vorhanden).

2. Linksverschiebung ASL (Arithmetic Shift Left)

Die logische Linksverschiebung stimmt mit arithmetischen Linksverschiebung überein. Die Wirkung von ASL ist analog zur Wirkung von LSR; z.B.:

Speicherplatz-Inhalt vor ASL □ 10110111
Speicherplatz-Inhalt nach ASL: [1] 01101110
Carry-Bit

Bemerkung: Linksverschiebung bewirkt bei Dualzahlen die Multiplikation mit 2.

3. Links-Rotation ROL (Rotate Left) und Rechts-Rotation ROR

Die Wirkung der Rotationsbefehle auf den Inhalt einer Speicherstelle und das Carry-Bit erkennt man an folgenden Beispielen:

Speicherplatz-Inhalt vor ROL: 11011101 0 — Carry-Bit
Speicherplatz-Inhalt nach ROL: 10111010 1

Speicherplatz-Inhalt vor ROR: 0 11011101
Speicherplatz-Inhalt nach ROR: 1 01101110
Carry-Bit

Beispiel: In den Zellen 1200,1201,1202 stehe eine Dualzahl a mit $a < 2^{23}$. a soll verdoppelt werden.

```
0200   0E 02 12        ASL $1202
0203   2E 01 12        ROL $1201
0206   2E 00 12        ROL $1200
0209   4C .. ..        JMP $....   Sprung in den Monitor
```

2.4 Flags und Statusregister

Bei zahlreichen Befehlen werden außer der Durchführung der eigentlichen Operation noch "Flags" (Fahnen) gesetzt; das sind Bits, die gewisse Zustände anzeigen sollen.
Die Flags werden in dem sogenannten Status-Register der CPU geführt.

N	V		B	D	I	Z	C

C: Carry (Übertrag)
Z: Zero (Null)
I: Interrupt disable (Intern. unbrauchbar machen)
D: Decimal Mode
B: Break
unbenutzt
V: Overflow (Überlauf)
N: Negativ

Die Bedeutung der meisten Flags kann jetzt noch nicht besprochen werden.
Die Carry-Flag und die Dezimal-Flag haben wir bereits kennengelernt. Die Dezimal-Flag wurde durch SED auf 1 und durch CLD auf 0 gesetzt. Die Carry-Flag wurde automatisch nach einem Additionsbefehl - je nach Ergebnis - auf 0 oder 1 gesetzt. Welche Flags durch einen Befehl verändert werden, ist in der Befehlsliste des 6502 zu finden.

Beispiel: Bei dem Befehl LDA (Load Accumulator) wird

a) Z = 1 gesetzt, falls 0 in den Akkumulator gebracht wurde, sonst wird Z = 0 gesetzt.

b) N = 1 gesetzt, falls Zahl im Akkumulator negativ (d.h. Bit 7, das höchstwertigste Bit ist 1), sonst wird N = 0 gesetzt.

c) Alle weiteren Flags bleiben unverändert.

Es gibt auch Befehle die nur Flags verändern:

CLC	C = 0	(Clear Carry)
SEC	C = 1	(Set Carry)
CLD	D = 0	(Clear Decimal Mode)
SED	D = 1	(Set Decimal Mode)
CLI	I = 0	(Clear Interrupt Disable Flag)
SEI	I = 1	(Set Interupt Disable Flag)
CLV	V = 0	(Clear Overflow Flag)

Zu den Befehlen, die ebenfalls nur Flags verändern, gehören noch BIT (mit dem man gewisse Bits von Speicherzellen kontrollieren kann) und die wichtigen Vergleichsbefehle.

Der Vergleichsbefehl CMP (Compare) setzt die Flags auf folgende Weise:

CMP M bewirkt:
- Z=1 , falls (Akku) - (M) = 0
- Z=0 , falls (Akku) - (M) ≠ 0
- C=1 , falls (Akku) ≥ (M)
- C=0 , falls (Akku) < (M)

(Akku) bedeutet "Inhalt des Akkus" und (M) analog "Inhalt der Speicherzelle M".

Flags können nun abgefragt werden. Dies geschieht mit bedingten Sprüngen, die im nächsten Abschnitt behandelt werden.

Bemerkung: Das Flagkonzept wird wohl bei allen Mikroprozessoren verwendet. Unterschiede gibt es in der Art und in der Behandlung der Flags. C, Z, N sind meist vorhanden.

2.5 Sprünge und Unterprogramme

Unbedingter Sprung (Jump)

Bei diesem Sprung wird, ohne auf eine Bedingung zu achten, gesprungen (vgl. GO TO in FORTRAN). Die Sprungadresse kann direkt oder indirekt angegeben werden. Wir werden die Befehle an Beispielen erläutern.

a)Direkt (absolute)

Beispiel:

4C 03 04 Springe nach 0403
symbolisch: JMP $0403 (Jump)

b)Indirekt

Beispiel:

6C 03 04 Springe zur Adresse, die in 0403,0404 steht.
symbolisch: JMP ($0403)

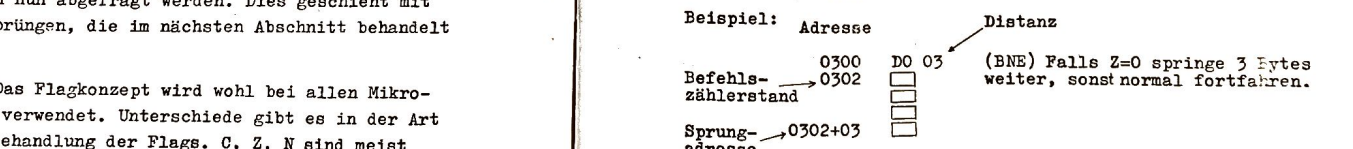
Bedingte Sprünge (Branches)

In Abhängigkeit vom Zustand gewisser Flags wird gesprungen:

BCC	Sprung, falls C = 0	(Branch on Carry Clear)
BCS	Sprung, falls C = 1	(Branch on Carry Set)
BEQ	Sprung, falls Z = 1	(Branch on Equal Zero)
BNE	Sprung, falls Z = 0	(Branch on not Equal Zero)
BMI	Sprung, falls N = 1	(Branch on Minus)
BPL	Sprung, falls N = 0	(Branch on Plus)
BVC	Sprung, falls V = 0	(Branch on Overflow Clear)
BVS	Sprung, falls V = 1	(Branch on Overflow Set)

Diese Sprünge sind relative Sprünge, d.h. die Adresse wird nicht direkt (absolute), sondern als Distanz zum Befehlszählerstand angegeben.

Beispiel:



Wird eine negative Zahl angegeben, z.B. -5 ≙ FB, so

```
Sprungadresse    0302-5  ☐
                         ☐
                         ☐
                         ☐
                 DO FB   ☐   Falls Z=0 springe 5 Bytes
Befehlsz.-stand  0302        zurück, sonst fortfahren.
```

Mit diesen relativen Sprüngen kann also 128 Bytes zurück und 127 Bytes vorwärts gesprungen werden (vgl. Zahlendarstellung in einem Byte). Durch eine Kombination eines relativen bedingten Sprungs mit JMP sind aber auch große Distanzen zu überbrücken.

Beispiel: Falls C=0, soll nach 2400 gesprungen werden.

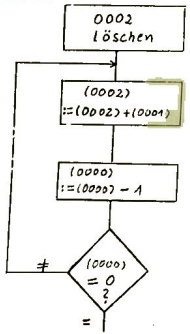
```
BO 03                BCS WEITER
4C 00 24             JMP $2400
...          WEITER  ...
```

Kommt man in einem Programm mit relativen Sprüngen aus, so hat man den großen Vorteil, daß beim Verschieben eines Programms in einen anderen Speicherbereich keine Änderungen von Sprungadressen erforderlich sind.

Im folgenden Programmbeispiel wollen wir eine Multiplikation m·n durchführen, indem wir einfach m mal n addieren.

Aufgabe: Berechne m·n, wobei m in 0000 (m·n ≤ 255, m > 0)
n in 0001
m·n in 0002

Programmablaufplan




```
      LDA #00
      STA $0002
MARKE CLC
      LDA $0002
      ADC $0001
      STA $0002

      DEC $0000   (Decrement)

      BNE MARKE
      ...
```


In dem Programm verwenden wir den neuen Befehl DEC (Decrement). Er bewirkt, daß der Inhalt einer Speicherzelle um 1 erniedrigt wird (ohne den Akkumulatorinhalt zu verändern).
Ob das Dekrementieren den Wert 0 oder einen Wert ≠ 0 ergeben hat wird in der Zero-Flag festgehalten, so daß diese unmittelbar für einen bedingten Sprung (BNE oder BEQ) benutzt werden kann.
Diese Schleifentechnik - am Ende der Schleife dekrementieren und anschließend bedingt springen - wird sehr häufig angewendet.

Das Maschinenprogramm zur Multiplikationsaufgabe sieht nun so aus:

```
0400  A9 00          LDA #00
0402  8D 02 00       STA $0002
0405  18      MARKE  CLC
0406  AD 02 00       LDA $0002
0409  6D 01 00       ADC $0001
040C  8D 02 00       STA $0002
040F  CE 00 00       DEC $0000
0412  DO F1          BNE MARKE
0414  ...            ...
```

Das Programm könnte etwa mit einem Sprung in den Monitor beendet werden.

Die Distanz F1 im bedingten Sprung BNE erhält man durch Subtraktion der Hexadezimalzahlen

```
 0405   (Sprungziel)
-0414   (Befehlszählerstand)
 --
 ..F1
```

oder durch Rückwärtszählen 00,FF,FE,FD,...,F1 ausgehend von 0414.

Bedingte Sprünge können gut mit Vergleichsbefehlen kombiniert werden. Wir wollen die wichtigsten Vergleiche dabei aufführen:

```
Springe, falls C < D        LDA C
                            CMP D
                            BCC Sprungziel (Distanz)

Springe, falls C ≤ D        LDA D
                            CMP C
                            BCS Sprungziel (Distanz)

Springe, falls C = D        LDA C
                            CMP D
                            BEQ Sprungziel (Distanz)

Springe, falls C ≠ D        LDA C
                            CMP D
                            BNE Sprungziel (Distanz)
```

```
Springe, falls C > D        LDA D
                            CMP C
                            BCC Sprungziel (Distanz)

Springe, falls C ≥ D        LDA C
                            CMP D
                            BCS Sprungziel (Distanz)
```

Bemerkung: Das Programmbeispiel benutzt eine Abfrage am Schleifenende. Deshalb wird die Schleife mindestens einmal durchlaufen. Bei m = 0 würde das Programm also nicht richtig laufen. Abhilfe bringt ein Schleifenaufbau mit einer Abfrage am Schleifenanfang.

Unterprogrammsprung (Jump to Subroutine)

```
20 3A 12        JSR $123A       (Jump to Subroutine)
```

bewirkt: Es wird zum Unterprogramm gesprungen, das bei 123A beginnt und die Rücksprungadresse wird in bestimmten Zellen (dem Stack) gespeichert. Der Stack besteht aus den Zellen 0100-01FF, das sind 256 Zellen.

Durch den Rücksprung vom Unterprogramm

```
60          RTS     (Return from Subroutine)
```

wird vom Unterprogramm in das aufrufende Programm zurückgesprungen. Die Rücksprungadresse wird im Stack gefunden.

Beispiel für ein Unterprogramm

Aufgabe: Eine 16-bit Dualzahl in 5020,5021 soll inkrementiert werden (d.h. um 1 erhöht werden).

```
0300  EE 21 05     INC $0521       (Increment)
0303  D0 03        BNE R
0305  EE 20 05     INC $0520
0308  60        R  RTS             (Return form Subroutine)
```

Zunächst wird die Zelle 0521 erhöht; falls sich bei dieser Erhöhung aus FF 00 ergeben hat, muß auch noch Zelle 0520 inkrementiert werden, andernfalls kann sofort auf RTS gesprungen werden.
Ein Arbeiten mit der Carry-Flag ist hier nicht möglich, da beim Inkrementieren nur die Zero- und die Negativ-Flag berührt werden.

Das angegebene Unterprogramm soll nun benutzt werden, um 1F3A zu inkrementieren:

```
0200  A9 1F        LDA #1F
      8D 20 05     STA $0520
      A9 3A        LDA #3A
      8D 21 05     STA $0521
      20 00 03     JSR $0300   Aufruf des Unterprogramms
      ...
```

Häufig ist es sinnvoll, Unterprogramme des Monitors zu benutzen. Beispielsweise besitzt fast jeder Monitor Unterpr., um Zahlen, Buchstaben und andere Zeichen auszugeben. Das Zeichen wird üblicherweise in der ASCII-Codierung (American Standard Code for Information Interchange; siehe Anhang) in den Akkumulator eingegeben, um es mit Hilfe des entsprechenden Unterprogramms (auf einem Bildschirm, Drucker oder einer sonstigen Anzeige) auszugeben.

Beim KIM-Monitor beginnt das Ausgabe-Unterprogramm bei 1EA0.

Beispiel: Das Zeichen B ist laufend auszugeben.

```
0400  A9 42        ANFG LDA #42        42 ASCII-Code für B
      20 A0 1E          JSR ASCII      Zeichenausgabe
      4C 00 04          JMP ANFG       Wiederhole die Ausgabe
```

Bemerkung: Die Berechnung der Distanz bei den Sprüngen mit relativer Adressierung ist eine mühselige Angelegenheit, wenn man Maschinenprogramme ohne Assemblerhilfe erstellt.
Die relativen Sprünge sparen aber Speicher (2 Byte- statt 3 Byte-Befehle). Weiter sind relative Sprünge Voraussetzung für die Verschiebbarkeit eines Programms im Hauptspeicher.
Die Distanz im relativen Sprung muß nämlich - im Gegensatz zur absoluten Adresse - nicht geändert werden, wenn das Programm in einem anderen Speicherbereich ablaufen soll.

Die Verschiebbarkeit von Programmen bedeutet in vielerlei Hinsicht eine Erleichterung bei der Benutzung von Programmen. Oft ist z.B. ein Speicherbereich belegt, in dem normalerweise das Programm X abläuft. Ohne die Möglichkeit der Verschiebung wäre Programm X nun nicht ablauffähig.

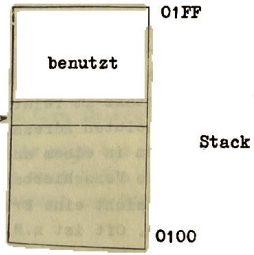
Eine Vermarktung von Programmen in ROMs wird durch ihre Verschiebbarkeit ebenfalls wesentlich begünstigt, da das ROM dann einen beliebigen Teil des adressierbaren Speichers benutzen darf.
Die relative Adressierung wird von den bekannten 8 bit-Mikroprozessoren nur unvollkommen unterstützt. So existieren keine relativen Sprünge über eine größere Distanz; der 6502 und Z80 besitzen keine relativen Unterprogrammsprünge und der 8080/85 kennt keine relative Adressierung.
Neuentwicklungen von Mikroprozessoren (z.B. Motorola 6809 im 8 bit-Bereich und Motorola 68000 im 16 bit-Bereich) beseitigen die erwähnten Mängel.

2.5 Stapelspeicher (Stack)

Ein Stack ist ein Speicher, der wie ein Stapel von Spielkarten behandelt wird:

Im Stack speichern: Karte auf Stapel legen
Vom Stack lesen: Karte vom Stapel nehmen (gelesene Information ist nach dem Lesen nicht mehr verfügbar)

Stack beim 6502: 0100-01FF (im RAM)
"Unterste Karte" bei 01FF (=oberste Zelle)



Der Stackpointer (Stapelzeiger) ist ein Register der CPU und enthält die Adresse (beim 6502 nur den niederwertigen Teil der Adresse) der ersten freien Zelle im Stack.
Durch den Stapel-Befehl

PHA (Push Accumulator on Stack)

wird der Inhalt des Akkumulators im Stack gespeichert und der Stackpointer um 1 erniedrigt.

PLA (Pull Accumulator from Stack)

holt ein Byte vom Stack in den Akkumulator und erhöht den Stackpointer um 1.

Das Statusregister kann ebenfalls - ohne den Umweg über den Akkumulator - durch

PHP (Push Processor Status on Stack)

im Stack gespeichert werden und durch

PLP (Pull Processor Status from Stack)

geladen werden.

Der Stack dient insbesondere zur Speicherung von Rücksprungadressen in der Unterprogrammtechnik.
Wie bei der Erläuterung des Unterprogramm-Sprungs bereits erwähnt wurde, wird die Rücksprungadresse (2 Bytes !) im Stack gespeichert; dabei wird natürlich der Stackpointer um 2 erniedrigt. Beim Rücksprung vom Unterprogramm wird die Rücksprungadresse vom Stack geholt und der Stackpointer wird um 2 erhöht.

Die Stackbearbeitung geschieht gerade so, wie sie für die Bearbeitung geschachtelter Unterprogramme erforderlich ist. Das zuletzt aufgerufene Unterprogramm ist das erste, welches wieder verlassen wird. Entsprechend ist die zuletzt abgelegte Rücksprungadresse auch die erste, die für den Rücksprung benutzt und vom Stack entfernt wird.

Bemerkung: In einer älteren Unterprogrammtechnik (z.B Siemens 305) wird die Rücksprungadresse in reservierten Zellen am Anfang des Unterprogramms abgespeichert. Diese Zellen müssen einem Schreib-Lese-Speicher angehören, d.h. ein Unterprogramm kann bei der älteren Unterprogrammtechnik nicht in einem ROM abgespeichert werden. Die Stacktechnik dagegen gestattet das Abspeichern von Unterprogrammen in ROMs.
Auch rekursive Aufrufe (Aufruf eines Unterprogramms durch sich selbst) sind mit der Stacktechnik möglich.

Eine weitere typische Verwendung des Stack ist das Retten von Akkumulator-, Register- und Zelleninhalten.

Beispiel: Ein Unterprogramm soll so geschrieben werden, daß nach Ablauf des Unterpogramms der Akkumulator und das Status-Register unverändert sind.

```
08      PHP     (Push Processor Status on Stack)
48      PHA     (Push Accumulator on Stack)
        }       eigentliches Unterprogramm ohne RTS
68      PLA     (Pull Accumulator from Stack)
28      PLP     (Pull Processor Status from Stack)
60      RTS     (Return from Subroutine)
```

Achtung: Bei der Benutzung des Stack in einem Unterprogramm ist darauf zu achten, daß der Stack wieder abgebaut wird (zu jedem Push gehört ein Pull), da sonst bei RTS eine falsche Rücksprungadresse verwendet wird.

Bemerkung: Die Stacktechnik wird bei fast allen Mikroprozessoren angewendet. Im Gegensatz zum 6502 ist der Stackbereich bei den Mikroprozessoren 6800, 8080, Z80 nicht festgelegt. Der RAM-Bereich, der für den Stack verwendet werden soll, kann hier beliebig durch Initialisieren des Stackpointers gewählt werden.

2.7 Adreßarten ohne Index-Register

Viele Befehle haben verschiedene Adressierungsmöglichkeiten. Diese Vielzahl von Adreßarten vereinfachen Programme oft erheblich.

Absolute → Extended → 6800

Der Befehl ist ein 3 Byte-Befehl, z.B. 6D A0 3F.
Die Adresse des Operanden steht in Byte 2 und Byte 3 des Befehls.

Zero Page → Direkt → 6800

Man teilt den Hauptspeicher in Seiten (pages) von 256 Bytes auf. Seite 0 hat die Adressen 0000 - 00FF, Seite 1: 0100 - 01FF usw. . Liegt eine Adresse in Seite 0, so genügt zur Adressierung 1 Byte (niederwertiger Teil der Adresse).

```
Beispiel: Statt 6D A1 00      ADC $00A1
          kann  65 A1         ADC $A1     geschrieben werden.
                   Adresse in Seite 0
```

Die Zero Page-Adressierung erspart Speicherplatz und Rechenzeit. Deshalb wird diese Adreßart ausgiebig verwendet.

Accumulator und Implied

Bei dieser Adressierungsart sind die Befehle 1 Byte-Befehle, bei denen die gesamte Befehlsinformation bereits im Operationscode enthalten ist (implied = mit einbegriffen).

Immediate

Bei dieser "Adreßart" wird keine Adresse angegeben, in der der Operand steht. Der Operand steht unmittelbar (immediate) im 2-ten Byte des Befehls.

Relative

Bei bedingten Sprüngen enthält das 2-te Byte eine relative Adresse, d.h. eine Distanz zum Befehlszähler stand.

Indirekt

Nur bei JMP, z.B. 6C 05 10 ; die Sprungadresse steht in den Zellen 1005, 1006.
Die indirekte Adressierung ist auch für zahlreiche andere Befehle möglich. Dies geschieht aber in Verbindung mit einem Indexregister und wird später besprochen.

Beispiel: Addiere zur Zahl in Zelle 001o die Hex-Zahl 1A, Ergebnis in Zelle 0011.

```
18      CLC         (Implied)
A5 10   LDA $10     (Zero Page)
69 1A   ADC #1A     (Immediate)
85 11   STA $11     (Zero Page)
```

In der symbolischen Darstellung der Befehle zeigt # an, daß die Adreßart "Immediate" verwendet wird.

*$ ≙ Zero-Page operation

Bemerkung: In der Bezeichnung der Adressierungsarten gibt es leider keine Einheitlichkeit. "Absolute" heißt beim 8080 "Direct", beim 6800 und beim Z80 "Extended". "Zero Page" ist beim 6800 "Direct".
Die Adreßart "Zero Page" ist beim 8080/Z80 und die relative Adressierung beim 8080 nicht vorhanden.

3. Editieren und Assemblieren

Das Arbeiten mit dem Maschinencode ist auch bei kleinen Programmen sehr mühselig. Der Befehlscode ist schwer erlernbar, Maschinenadressen sind nichtssagend und müssen häufig bei der kleinsten Ergänzung des Programms geändert werden. Diese Änderungen sind ganz besonders unangenehm und fehleranfällig.
Deshalb benutzt man ein Assemblerprogramm, bei dem Befehlscode und Adressen symbolisch angegeben werden. Als Befehlscode wird ein sogenannter Mnemocode (Mnemonik = Kunst das Gedächtnis durch Hilfsmittel zu unterstützen) verwendet, den der Hersteller des Mikroprozessors festlegt. Beim 6502 besteht der Mnemocode aus drei Buchstaben (z.B. STA für Store Accumulator).

Der Assembler - selbst ein Programm - übersetzt das Assemblerprogramm in das Maschinenprogramm (assemblieren).
Ein Cross-Assembler benutzt eine andere (nicht mit dem Mikroprozessor arbeitende, in der Regel größere) DV-Anlage zum Assemblieren. Das erstellte Maschinenprogramm kann über Lochstreifen, Kassette oder ein anderes Medium in den Mikrocomputer übertragen werden.
Ein Resident-Assembler läuft auf einem Computer, der dieglecheCPU wie der Mikrocomputer besitzt; im Idealfall läuft der Resident-Assembler also auf dem Mikrocomputer selbst. Der Resident-Assembler bietet den Vorteil, daß nach dem Assemblieren sofort - ohne Transport des Maschinenprogramms - Programmläufe durchgeführt werden können.
Für alle wesentlichen Mikroprozessoren sind Cross- und Resident-Assembler verfügbar. Resident-Assembler benötigen als Minimum ca. 2 kB Speicherplatz, sind aber meist umfangreicher.

Ein Assemblerprogramm ist ein Text, der sogenannte Quelltext. Diesen Text kann man an einem Bildschirmgerät mit Hilfe eines Programms (Editor) im Speicher erstellen (editieren). Das Erstellen des Quelltextes mit Hilfe des Editors ersetzt die Eingabe des Textes über Lochkarten.

Jeder Editor gestattet es, Zeilen einzugeben, Zeilen einzufügen, Zeilen zu löschen und Zeilen zu korrigieren.

Beispiel eines Assemblerprogramms

Aufgabe: Das Zeichen + ist 10 mal auszudrucken.

```
        .OR 3000      Beginn des Programms
ASCI    .DL 1EA0      Unterprogramm:ASCII-Zeichen Ausgabe.
STOP    .DL 1C4F      Eingabeprogramm des Monitors
ZAEL    .DL 0000      Zähler
- - - - - - - - -
        LDA #0A       0A ≙ 10
        STA ZAEL      Zähler setzen
SCHL    LDA #2B       2B ASCII-Zeichen für +
        JSR ASCI      Ausgabe von +
        DEC ZAEL      Zähler dekrementieren
        BNE SCHL      Falls Zähler ≠ 0, erneute Ausgabe
        JMP STOP      Sprung in den Monitor
- - - - - - - - -
        .EN           Ende des Quelltextes
```

Die erste Zeile legt fest, wo das Programm im Hauptspeicher beginnen soll (OR = Origin). Die nächsten drei Zeilen definieren die symbolischen Adressen ASCI, STOP, ZAEL (DL = Define Label). Das eigentliche Programm steht zwischen den gestrichelten Linien. Statt der Maschinenadressen werden hier die symbolischen Adressen verwendet.

Weitere Formalien zur Erstellung des Quelltextes sind im Anhang zu finden.

Die Programm-Entwicklung mit Hilfe eines Resident-Editors und eines Resident-Assemblers geschieht nun so:

1. Editieren des Assemblerprogramms
   a) Start des Editors
   b) Editieren des Quelltextes
   c) Beenden des Editors

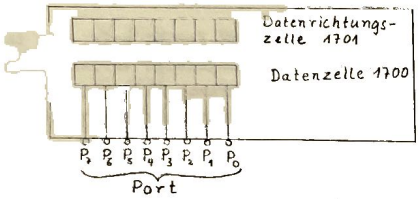
2. Assemblieren
   a) Start des Assemblers
   b) Eingabe, wo der Quelltext steht.
      (eventuell noch:Eingabe, wo das Maschinenprogramm abgespeichert werden soll)

3. Start des Maschinenprogramms

In der Regel werden im Quelltext Fehler sein, die im Assembliervorgang erkannt und angezeigt werden. Dann ist vor dem Start des Maschinenprogramms natürlich ein erneutes Editieren (Korrektur des Quellprogramms) und Assemblieren erforderlich.
Vor dem Start des Maschinenprogramms ist es zu empfehlen, den Quelltext (z.B. auf Kassette oder Floppy-Disk) zu retten, denn ein fehlerhaftes Programm kann den Hauptspeicherinhalt völlig zerstören.

4. Ein-/Ausgabe (Ports)

Der Inhalt gewisser Zellen (beim KIM: 1700 und 1702) kann mit Hilfe eines Ein-/Ausgabe-Bausteins über sogenannte Ports ausgegeben werden (0 ≙ 0 Volt, 1 ≙ 5 Volt).



Über die "Pins" $P_7, P_6, \ldots, P_0$ des Ports kann aber auch eingelesen werden. Über welchen Pin eingelesen oder ausgegeben werden soll, wird in der Datenrichtungszelle (beim KIM: 1701 bzw. 1703) festgelegt:

1 in Bit i bedeutet: Über Pin P_i soll ausgegeben werden
0 in Bit k bedeutet: Über Pin P_k soll eingelesen werden

Beispiel:

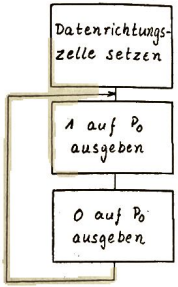
Inhalt der Datenrichtungszelle: 00000011
Dann wird über P_0, P_1 ausgegeben und über P_2, P_3, P_4, P_5, P_6, P_7 wird eingelesen.

Die Signale mit denen an den Pins gearbeitet wird, sind die sogenannten TTL-Signale (0 ≙ 0-0,8 Volt, 1 ≙ 2-5 Volt). Um bei der Eingabe eine 0 zu erzeugen, kann der Pin auf Masse gelegt werden. Ein offener Eingang erzeugt eine 1 (oder 5 V über einen Widerstand ≥ 1 kΩ).

Wird über Pin P_i ein Signal ausgegeben, so bleibt dieses bis zu einer einer erneuten Ausgabe (durch Beschreiben der Datenzelle) erhalten.

Zur Demonstration der Ausgabe soll eine Rechteckfolge auf Pin P_0 erzeugt werden. Die Rechteckfolge kann mit einem Oszillographen sichtbar gemacht werden, indem zwischen P_0 und Masse gemessen wird.

Aufgabe: Rechteckfolge auf Pin P_0



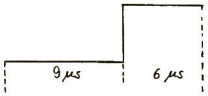

```
        .OR 3000
DRZ     .DL 1701        Datenrichtungsz.
PORT    .DL 1700
        LDA #01
        STA DRZ
MARKE   LDA #01
        STA PORT
        LDA #00
        STA PORT
        JMP MARKE
        .EN
```

Ermitteln wir einmal an Hand der beigefügten Tabelle (Anhang) die Ausführungszeit der Schleife (1 Zyklus = 1 µs):

2 µs (LDA #01) + 4 µs (STA PORT) + 2 µs (LDA #00) + 4 µs (STA PORT) + 3 µs (JMP MARKE) = 15 µs

Die Frequenz des ausgegebenen Rechtecksignals ist also 66,$\overline{6}$ kHz.

Da nach der Ausgabe der 0 auf P_0 der Sprung folgt, dauert die "0" 3 µs länger als die "1".

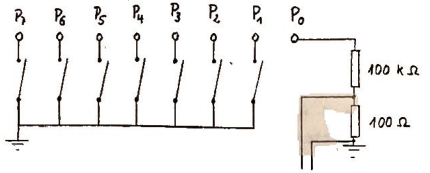


Im nächsten Programmbeispiel wollen wir einen Rechteckgenerator programmieren, dessen Frequenz durch Schalterstellungen verändert werden kann.
Dabei sollen die Schalterstellungen als Dualzahl interpretiert werden; diese Dualzahl definiert dann die Dauer einer Warteschleife.

Aufgabe: Variabler Rechteckgenerator

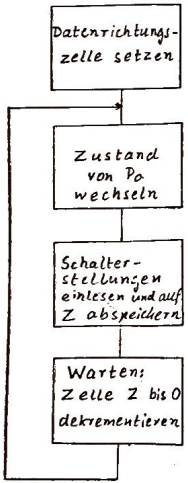
Die Frequenz eines Rechteckgenerators soll durch Schalterstellungen verändert werden.

Hardware:



zum Kassettenrecorder, der auf Aufnahme gestellt wird.

Die Rechtecksignale sollen über den Lautsprecher eines Kassettenrecorders hörbar gemacht werden.




```
          .OR 3000
DRZ       .DL 1701
PORT      .DL 1700
Z         .DL 0000

          LDA #01
          STA DRZ

MARKE     INC PORT

          LDA PORT
          LSR          log. Rechtsrech.
          STA*Z

WART      DEC Z
          BNE WART

          JMP MARKE

          .EN
```


Nach dem Setzen der Datenrichtungszelle (P_0 auf Ausgabe, $P_1,\ldots,P_7$ auf Eingabe) wird das Bit an P_0 durch Inkrementieren der Datenzelle gewechselt. Hier wird die Tatsache benutzt, daß ein Port wie eine Speicherzelle behandelt werden kann. Anschließend wird der Port gelesen und durch Rechtsverschiebung (LSR) die - den Schalterstellungen entsprechende - Dualzahl hergestellt (gleichzeitig wird ein unerwünschter Einfluß von P_0 eliminiert) und auf Zelle Z abgespeichert. Dann wird Zelle Z bis zur Null abwärts gezählt (Warteschleife).

Eine Addition der Ausführungszeiten für die Befehle ergibt eine Periodendauer von $T = 2(18 + 7(n-1) + 8)\ \mu s = 52 + 14(n-1)\ \mu s$ (n eingelesene Dualzahl, $n \geq 1$). Es lassen sich daher Frequenzen zwischen 550 Hz und 19 kHz erzeugen.

Die hier besprochene digitale Ein-/Ausgabe läßt sich selbstverständlich auch für analoge Ein-/Ausgabe verwenden, indem ein Analog/Digital- bzw. ein Digital/Analog-Wandler zwischengeschaltet wird.

Bemerkung: Bei geeigneten Ein/Ausgabe-Bausteinen lassen sich wohl bei allen Mikroprozessoren die Ports wie Speicherzellen ansprechen. Beim 8080 und Z80 werden die Ports aber meist mit speziellen Ein/Ausgabe-Befehlen angesprochen. Dies hat bei größeren Systemen den Vorteil, daß keine Speicherplätze an Ports "verloren gehen". Der Nachteil ist, daß nicht mehr alle Befehle, die Speicherplätze bearbeiten, verwendet werden können.

5. Programmierung mit Index-Registern

5.1 Index-Register

Außer dem Akkumulator besitzt der 6502 noch zwei allgemein verwendbare Register: Das X-Register und das Y-Register. Für das Arbeiten mit diesen 8 bit-Registern sind folgende Befehle vorhanden:

LDX	Load X-Register
LDY	Load Y-Register
STX	Store X-Register
STY	Store Y-Register
TAX	Transfer Accumulator to X-Register
TAY	Transfer Accumulator to Y-Register
TXA	Transfer X-Register to Accumulator
TYA	Transfer Y-Register to Accumulator
TSX	Transfer Stack-Pointer to X-Register
TXS	Transfer X-Register to Stack-Pointer
INX	Increment X-register
INY	Increment Y-Register
DEX	Decrement X-Register
DEY	Decrement Y-Register
CPX	Compare with X-Register
CPY	Compare with Y-Register

Die Bezeichnung "Index"-Register rührt von der Möglichkeit her, den Registerinhalt - wie etwa i bei a_i in der Mathematik - als Index zu benutzen. Die Verwendung von X, Y als "Index"-Register wird im nächsten Abschnitt 5.2 besprochen.

X und Y-Register eignen sich gut als Zähler zur Schleifenbildung. Der Zähler benötigt keinen Speicherplatz und das Dekrementieren (Inkrementieren) des Registers erfordert erheblich weniger Zeit als das Dekrementieren (Inkrementieren) einer Speicherzelle.

Beispiel: Ein Programmstück soll 7 mal durchlaufen werden.

```
        LDX #07
ANFG    ... ...  }
        .        }  Programmstück, welches das X-Register
        .        }  nicht verändert.
        ... ...  }
        DEX
        BNE ANFG
```

oder:

```
        LDX #00
ANFG    INX
        ... ...  }
        .        }  Programmstück, welches das X-Register
        .        }  nicht verändert.
        ... ...  }
        CPX #07     Falls (X) = 07, wird Z-Flag gesetzt.
        BNE ANFG
```

Bei einem Programmstück, welches das X-Register verändert, läßt sich eine solche Schleife natürlich nicht verwenden, da der Schleifenzäler X vernichtet wird.

Um die Verwendung der Index-Register nicht unnötig einzuschränken, sollte ein Unterprogramm stets X- und Y-Register unverändert lassen. Dies geschieht durch Retten der Register im Stack.

Das Retten und Wiederherstellen von Akkumulator, Statusregister, X- und Y-Register geschieht nun so:

```
PHP              Push Processor Status
PHA              Push Accumulator
TXA              Transfer X-Register to Accumulator
PHA              Push Accumulator
TYA              Transfer Y-Register to Accumulator
PHA              Push Accumulator
... ...  }
.        }  Unterprogramm ohne RTS
... ...  }
PLA              Pull Accumulator from Stack
TAY              Transfer Accumulator to Y-Register
PLA              Pull Accumulator from Stack
TAX              Transfer Accumulator to X-Register
PLA              Pull Accumulator from Stack
PLP              Pull Processor Status from Stack
RTS              Return from Subroutine
```

5,2 Indizierte Adressierung

Bei der absoluten Adressierung wird die Adresse des Operanden direkt im Adreßteil des Befehls angegeben. Bei der indizierten Adressierung erhält man die Adresse des Operanden, indem zum Adreßteil noch der Inahlt eines Registers addiert wird.

```
Beispiel:    A2 03          LDX #03
             BD 20 1C       LDA $1C20,X
```

In diesem Beispiel wird die Adreßart "Absolute,X" verwendet. Im zweiten Befehl wird also nicht der Inhalt der Zelle 1C20, sondern von Zelle 1C20+03 = 1C23 in den Akkumulator geladen.

Im folgenden Programmbeispiel enthält das X-Register gerade die Indizes der Werte a_0, a_1, ... , a_{20} (daher "Index"-Register).

Aufgabe: Auf den Zellen 2E10, 2E11, ... , 2E30 stehen die Werte a_0, a_1, ... , a_{20} (20 ist als Hex-Zahl zu deuten), die nach 2F00, 2F01, ... , 2F20 kopiert werden sollen.

```
        .OR 3000
ERST    .DL 2E10
ZIEL    .DL 2F00
STOP    .DL 1C4F
        LDX #00
SCHL    LDA ERST,X      Adressierungsart "Absolute,X"
        STA ZIEL,X      Adressierungsart "Absolute,X"
        INX
        CPX #21         Vergleich, ob Schleife beendet.
        BNE SCHL
        JMP STOP
        .EN
```

Auch das Y-Register kann als Index-Register vewendet werden; dann heißt die Adreßart "Absolute,Y". Bei Zero Page-Adressen existieren entsprechend die Adressierungsarten "Zero Page,X" und "Zero Page,Y" (wobei allerdings "Zero Page,Y" nur für LDX und STX vorhanden ist).

Die indizierte Adressierung ist immer dann sinnvoll, wenn gleichartige Operationen mit - in festem Abstand - aufeinanderfolgenden Speicherzellen durchgeführt werden sollen. Auch eignet sich die indizierte Adressierung gut, um Werte einer Tabelle zu entnehmen.
Dies soll das folgende Programmbeispiel zur Steuerung einer Ampel zeigen.

Aufgabe: Nach folgender Tabelle soll ein Zyklus für eine Ampel ablaufen.

Zeitintervall #	Rot	Gelb	Grün
0	0	0	1
1	0	0	1
2	0	0	1
3	0	0	1
4	0	0	1
5	0	1	0
6	1	0	0
7	1	0	0
8	1	0	0
9	1	0	0
10	1	0	0
11	1	0	0
12	1	1	0

1 ≙ Licht ein
0 ≙ Licht aus

Das rote, gelbe, grüne Licht der Ampel sollen die Pins P_2, P_1, P_0 eines Ports steuern.

```
        .OR 3000
DRZ     .DL 1701        Datenrichtungszelle
PORT    .DL 1700        Adresse des Ports
        LDA #FF
        STA DRZ         Datenrichtungszelle setzen
ZYKLUS  LDY #0
ZEIT    LDA TABELLE,Y   Tabellenwert entnehmen
        STA PORT        Ampel steuern
        JSR WARTEN      Unterprogramm zum Warten
                        (1 Zeitintervall)
        INY
        CPY #D
        BNE ZEIT        Nächstes Zeitintervall
        JMP ZYKLUS      Nächster Ampelzyklus
TABELLE .HS 0101010101  }
        .HS 02          } Beim Assemblieren wird die
        .HS 040404040404} Tabelle hinter dem Programm
        .HS 06          } abgespeichert.
        .EN
```

Zum Übersetzen und Ablauf des vorstehenden Programms ist natürlich das Unterprogramm WARTEN, welches ca. 1 Zeitintervall wartet, erforderlich.

5.3 Indirekte und indirekte, indizierte Adressierung

Bei indirekter Adressierung verweist der Adreßteil des Befehls auf zwei Zellen, in denen die Adresse des Operanden steht. Beim 6502 ist die indirekte Adressierung nur in Verbindung mit einem Index-Register möglich (außer bei JMP):

Adreßart (Indirect),Y

Der Adreßteil des Befehls verweist bei dieser Adreßart auf zwei Zero Page-Zellen. In diesen Zellen steht die Adresse a. Ist y der Inhalt von Y, so ist dann a+y die Adresse des Operanden.

```
Beispiel:  A0 15          LDY #15      | Inhalt von 5A sei:01
           B1 5A          LDA ($5A),Y  | Inhalt von 5B sei:1A
```

Im zweiten Befehl wird nicht der Inhalt von Zelle 005A, sondern der Inhalt von Zelle 1A16 (1A16 = 1A01+15) in den Akkumulator geladen.

Enthält das Y-Register die Null, so liegt normale indirekte Adressierung vor. Diesen Fall wollen wir im nächsten Programmbeispiel demonstrieren:

Aufgabe: Beginnend bei Zelle 0300 soll fortlaufend Text, der am Bildschirmterminal eingegeben wird, abgespeichert werden. Nach Eingabe des Wagenrücklaufzeichens (CR = Carriage Return) soll das Programm beendet werden.

```
        .OR 3000
ADR     .DL 0010
LIES    .DL 1E5A
STOP    .DL 1C4F

        LDA #00
        STA ADR
        LDA #03      } Adresse 0300 in die Zero Page-Zellen
        STA ADR+1    } 0010, 0011 bringen
```



```
LESEN JSR LIES      Einlesen eines ASCII-Zeichens in den Akk
       CMP #0D    }
       BEQ ENDE   } Prüfe, ob Wagenrücklauf ( ≙ 0D )
       LDY #00
       STA (ADR),Y  Abspeichern des Zeichens
       INC ADR    }
       BNE SE     } Adreßzähler erhöhen
       INC ADR+1  }
SE     JMP LESEN
ENDE   JMP STOP
       .EN
```

In diesem Programm wird ein Unterprogramm des Monitors benutzt, welches auf die Eingabe eines Zeichen vom Bildschirmterminal wartet und nach erfolgter Eingabe im Akkumulator abspeichert (Unterprogramm LIES).

Im folgenden Programmbeispiel wird eine Kombination von indirekter und indizierter Adressierung verwendet.

Aufgabe: Es ist ein Unterprogramm zu schreiben, welches 32 Werte, die forlaufend im Hauptspeicher stehen, über einen Port ausgibt.

```
        LDY #00
MARKE   LDA (ADR),Y    ADR,ADR+1 enthält Anfangsadresse der
        STA PORT                  auszugebenden Werte
        INY            Y erhöhen
        CPY #20        Prüfung, ob (Y) = 32    ( ≙ 20 Hex)
        BNE MARKE      Schleifenende
        RTS
```

Um das Unterprogramm zu benutzen, wird die Anfangsadresse der Werte in die Zero Page-Zellen ADR, ADR+1 geschrieben.

Bemerkung: Damit die Register durch das Unterprogramm nicht verändert werden, wird vor dem ersten Befehl noch PHP, PHA, TYA, PHA und vor RTS noch PLA, TAY, PLA, PLP eingefügt.

Eine weitere Adreßart, die indirekte und indizierte Adressierung kombiniert, ist:

Adreßart (Indirect,X)

Der Adreßteil enthält bei dieser Adreßart eine Zero Page-Adresse z, die um den Inhalt x von X erhöht wird (z+x < 256). Die Adresse steht dann in den Zero Page-Zellen z+x, z+x+1.

```
Beispiel:  A2 05        LDX #05
           A1 40        LDA ($40,X)
```

Die Adresse der Zelle, die in den Akkumulator geladen werden soll, steht in den Zellen 0045,0046.

Ist der Inhalt von X gleich Null, so liegt wiederum indirekte Adressierung vor; wie im ersten Programmbeispiel zu (Indirect),Y.

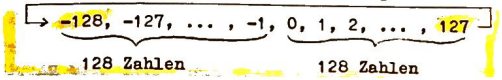
Die Adressierungsart (Indirect,X) eignet sich zur Verarbeitung von Daten, deren Adressen in einer Tabelle der Zero Page vorliegen. Eine Parameterliste zur Übergabe an ein Unterprogramm ist ein Anwendungsbeispiel.

Bemerkung: Die Mikroprozessoren 8080/85, Z80 haben eine wesentlich größere Zahl von Registern (8080/85: Akku+ 6 allg. verwendbare Reg., Z80: 2Akkus + 12 allg. Verwendbare Reg. + 2 "Index-Register") als der 6502 und der 6800. Ihr Befehlssatz ist mehr registerorientiert. D.h. es können zahlreiche Operationen mit den Registern durchgeführt werden, dafür werden gewisse Abstriche gemacht, wenn sich die Operanden im Speicher befinden.
Indirekte und indizierte Adressierung werden ebenfalls recht unterschiedlich gehandhabt. Die indirekte Adressierung geschieht beim 8080, Z80 über Registerpaare und beim 6800 über das sogenannte Index-Register. Zunächst muß ein Registerpaar (bzw. Indexregister) mit der Adresse geladen werden.
Index-Register sind beim 8080 nicht vorhanden. Beim Z80 und 6800 existieren sogenannte Index-Register. Sie enthalten aber nicht den Index. Das "Index-Register" enthält hier eine Adresse, zu der eine 8 bit-Größe (Displacement), die im Befehlscode steht, addiert wird. Bei dieser Art der Adressierung geht zwar der Index im üblichen Sinne verloren (und damit seine Verwendung als Schleifenzähler), dafür können aber Speicherbereiche, die mehr als 256 Bytes umfassen, bearbeitet werden.

6. Arithmetik mit Vorzeichen

Bisher wurden alle Werte als nicht-negativ angenommen. Sollen auch negative Zahlen bearbeitet werden, so werden sie - wie üblich in der Datenverarbeitung - durch das 2er-Komplement ihres Betrages ($2^n - |a| = 2^n + a$) dargestellt.

In einem Byte können $2^8 = 256$ Zahlen dargestellt werden:



Der Pfeil ↳ soll andeuten, daß die Addition von 1 zu 127 -128 ergibt.

```
00000000  ≙    0
00000001  ≙    1
00000010  ≙    2
   ...
01111111  ≙  127
10000000  ≙ -128
10000001  ≙ -127
   ....
11111111  ≙   -1
   ↑         ↑
 Bit 7     Bit 0
```

Bit 7 ist 1, falls die Zahl negativ ist. Daher wird die Negativ-Flag N auf 1 gesetzt, falls bei einer Operation Bit 7 = 1 wurde.

Überlauf (Overflow)

Eine Addition wird - unabhängig davon, ob das Byte als Zahl mit Vorzeichen oder als normale Dualzahl zu interpretieren ist - dual durchgeführt. Dies führt auch bei vorzeichenbehafteten Zahlen zum richtigen Ergebnis, sofern nur die Summe im Zahlenbereich noch darstellbar ist und ein Übertrag fortgelassen wird (bei 8 bit: $-128 \leq a+b \leq 127$). Es ist nämlich gerade ein wesentlicher Vorteil des 2er-Komplements in der Zahlendarstellung, daß negative Zahlen keiner Sonderbhandlung bedürfen.

Nun kann es bei der Addition a+b (a, b in einem Byte darstellbare Zahl) vorkommen, daß a+b nicht mehr im 8 bit-Bereich darstellbar ist, d.h. a+b > 127 oder a+b < -128. Dann ist ein Überlauf (nicht zu verwechseln mit Übertrag) eingetreten und es wird die Overflow-Flag V gesetzt (V=1).

Untersuchen wir nun, wann ein Überlauf eintritt:

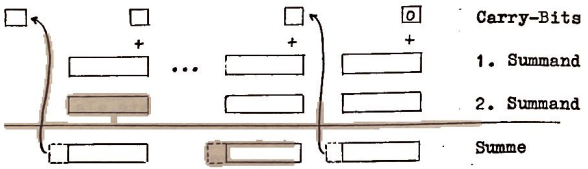
Die Zahlen a, b seien durch die Dualzahlen A, B dargestellt (d.h. $A = a$, falls $a \geq 0$ und $A = 2^8+a$, falls $a < 0$).

1. Bei Addition a+b kann ein Überlauf nicht eintreten, wenn a und b verschiedene Vorzeichen haben, d.h. $A_7 \neq B_7$. (A_7 = Bit 7 von A).
2. a, b > 0: Überlauf, falls $254 \geq a+b \geq 128$, d.h. $(A+B)_7 = 1$
3. a, b < 0: Überlauf, falls $-256 \leq a+b \leq -129$, d.h.
   $2^8+2^8-256 \leq A+B \leq 2^8+2^8-129$ ($A = 2^8+a$ 2er Kompl.)
   oder: $2^8 \leq A+B \leq 2^8+127$
   oder dual: $100000000 \leq A+B \leq 101111111$, d.h. $(A+B)_7 = 0$

Das Überlaufbit wird also gesetzt, falls $A_7 = B_7$ und $A_7 \neq (A+B)_7$

Im nächsten Programmbeispiel werden wir eine vorzeichenbehaftete Addition vornehmen und dabei einen Überlauf überwachen.

n-Byte Addition (mit Vorzeichen)



Der Überlauf braucht nur nach der letzten Addition geprüft zu werden.

Aufgabe: Eine 8 Byte-Addition ist durchzuführen.
1. Summand in den Zellen 0001, 0002, ..., 0008
2. Summand in den Zellen 0011, 0012, ..., 0018
Summe in den Zellen 0021, 0022, ..., 0028

```
SMD1   .DL 0000
SMD2   .DL 0010
SUMME  .DL 0020

       CLC              Carry-Bit löschen
       LDX #08          Zähler setzen
MARKE  LDA SMD1,X
       ADC SMD2,X       Summe bilden
       STA SUMME,X
       DEX              Zähler erniedrigen
       BNE MARKE        Schleifenende
       BVS OVFL         Prüfung, ob Überlauf
       ...
```

Subtraktion

Bekanntlich wird eine Subtraktion in DV-Anlagen über das 2er-Komplement durchgeführt. Genauer: Um a-b zu ermitteln, wird das 2er-Komplement 2^n-b von b zu a addiert und ein eventueller Übertrag wird fortgelassen. Das 2er-Komplement erhält man aus dem Komplement $\overline{b}$ (Vertauschen der Nullen und Einsen von b) durch Addition von 1. Zur Durchführung der Subtraktion a-b wird also $a+\overline{b}+1$ gerechnet.

Der Befehl SBC B (Subtract with Carry)
bewirkt:

$$(\text{Akku}) + \overline{(B)} + \text{Carry-Bit} \longrightarrow (\text{Akku})$$

$\overline{(B)}$: Komplement des Inhalts von B

Soll also eine Subtraktion mit SBC durchgeführt werden, so ist vorher C = 1 zu setzen:

```
SEC              Carry-Bit 1 setzen
SBC B            Subtraktion durchführen, Ergebnis
                 im Akkumulator.
```

Eine Mehr-Byte-Subtraktion $(a_1,a_2,\ldots,a_n)-(b_1,b_2,\ldots,b_n)$ (a_i, b_i jeweils ein Byte) ist so durchzuführen: $(a_1,a_2,\ldots,a_n) + \overline{(b_1,b_2,\ldots,b_n)} + 1 = (a_1,a_2,\ldots,a_n) + (\overline{b}_1,\overline{b}_2,\ldots,\overline{b}_n) + 1$. Genau dies wird aber durch eine n-fache Anwendung von SBC erreicht, sofern nur am Anfang C = 1 gesetzt wird.

Aufgabe: Eine 8 Byte-Subtraktion ist durchzuführen.
Minuend in den Zellen 0001, 0002, ... , 0008
Subtrahend in den Zellen 0011, 0012, ... ,0018
Differenz in den Zellen 0021, 0022, ... ,0028

```
MIN    .DL 0000
SUB    .DL 0010
DIFF   .DL 0020
       SEC              Carry-Bit setzen
       LDX #08          Zähler setzen
MARKE  LDA MIN,X  }
       SBC SUB,X  }     Differenz bilden
       STA DIFF,X }
       DEX              Zähler eniedrigen
       BNE MARKE        Schleifenende
       ...
```

Dieses Programm kann auch für Arithmetik mit Vorzeichen verwendet werden und nach der Schleife kann - wie im Programmbeispiel zur Addition - geprüft werden, ob ein Überlauf eingetreten ist.

Bemerkung: Bei der Subtraktion a-b, die ja nach der Formel $a+(2^n-b)$ (2^n-b 2er-Komplement von b) ermittelt wird, tritt ein Übertrag auf, wenn $a+2^n-b \geq 2^n$, d.h. wenn $a \geq b$. Bei der Subtraktion wird also C=1 gesetzt, wenn $a \geq b$ und C = 0 gesetzt, wenn $a < b$.
Ganz entsprechend wird mit den Vergleichsbefehlen verfahren. Bei CMP M wird - wie in 2.4 angegeben - C=1 gesetzt, falls (Akku) ≥ (M) und C=0 gesetzt, falls (Akku) < (M).

7. Programmunterbrechungen (Interrupt-Bearbeitung)

Durch Steuersignale auf die sogenannten Interrupt-Anschlüsse der CPU ist es möglich, ein laufendes Programm von außen her zu unterbrechen und ein anderes Programm zu starten.

Der Mikroprozessor 6502 besitzt (außer Reset) folgende Interruptmöglichkeiten:

1. IRQ (Interrupt Request)
Ist die I-Flag = 0, (I-Flag = Interrupt disable)
so bewirkt IRQ = 0: (0 V an IRQ-Anschluß)

a) Laufender Befehl wird noch abgearbeitet
b) Befehlszähler und Statusregister werden im Stack gerettet; I-Flag wird 1 gesetzt, damit ein weiterer Interrupt auf IRQ vorerst unmöglich ist.
c) Bei dem Befehl, dessen Adresse in FFFE,FFFF steht, wird fortgefahren (Interrupt-Bearbeitungsprogramm).

Der Befehl RTI (Return from Interrupt), der das Interruptbearbeitungsprogramm beendet, lädt das Statusregister wieder mit dem geretteten Wert. Sonst wird wie bei RTS verfahren, d.h. das unterbrochene Programm wird fortgesetzt.

Bemerkung: Ist I = 1, so wird auf IRQ = 0 nicht reagiert, d.h. es findet keine Programmunterbrechung statt.

2. BRK (Break, Software-Interrupt)
Der Befehl BRK wirkt wie IRQ = 0 und I = 0. Um eine Unterscheidung zwischen IRQ-Interrupt und BRK zu ermöglichen, wird bei BRK noch die B-Flag 1 gesetzt. Der Befehl wird vorwiegend für Testzwecke verwendet.

3. NMI (Non -Maskable Interrupt)
Bei einer Flanke von 1 nach 0 am NMI-Anschluß der CPU wird ohne Rücksicht auf die I-Flag wie bei IRQ ein Interrupt durchgeführt. Jedoch steht die Anfangsadresse des Interrupt-Bearbeitungsprogramms in FFFA, FFFB.

Bemerkung: Reset = 0 löst ebenfalls einen Interrupt aus; die Adresse des Interrupt-Bearbeitungsprogramms steht in FFFC, FFFD.

Häufig liegt der Bereich FFFA bis FFFF in einem ROM. Hier sind aber die Interuptvektoren, d.h. die Adressen, bei denen das Interuptbearbeitungsprogramm beginnt, abgespeichert. Somit beginnt das Interrupt-Bearbeitungsprogramm stets an derselben Stelle. Um trotzdem variabel zu bleiben, wird an dieser festen Stelle ein indirekter Sprung ausgeführt, der sich die Sprungadresse aus einem RAM-Bereich holt.

Beispiel KIM:

IRQ: "FFFE, FFFF" liegt im ROM und enthält 1F, 1C. Deshalb wird nach einem IRQ-Interrupt bei 1C1F fortgefahren. 1C1F liegt ebenfalls in einem ROM; dort beginnt der indirekte Sprung

6C FE 17 JMP ($17FE) ,

der sich die Sprungadresse aus dem RAM-Bereich 17FE, 17FF holt.
Somit kann die Anfangsadresse des Interrupt-Bearbeitungsprogramms in die Zellen 17FE, 17FF geschrieben werden.

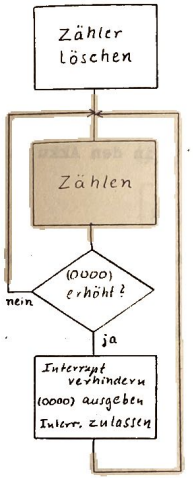
NMI: Analog zu IRQ: "FFFA, FFFB" enthält 1C, 1C und dort ist der indirekte Sprung JMP ($17FA) zu finden.
Die Anfangsadresse des Interrupt-Bearbeitungsprogramms kann also in die Zellen 17FA, 17FB geschrieben werden.

Programmbeispiel zur Interruptverarbeitung.

Aufgabe: 1) Ein Programm zähle in den Zellen 0000, 0001, 0002 und gebe nach jeder Erhöhung von 0000 den Inhalt von 0000 auf dem Bildschirm aus.

2) Bei einem IRQ-Interrupt werde ein I ausgegeben.

zu 1)




```
        .OR 3000
ZAEL    .DL 0000        Zähler
BYTE    .DL 1E3B        Ausgabe eines Bytes
                        \ Seite 80
        LDA #00
        STA ZAEL
        STA ZAEL+1
        STA ZAEL+2

MARKE   INC ZAEL+2
        BNE MARKE
        INC ZAEL+1
        BNE MARKE
        INC ZAEL

        LDA ZAEL
        SEI             Interr. Verhindern
        JSR BYTE        Ausgabe eines Bytes
        CLI             Interr. zulassen

        JMP MARKE
        .EN
```

Zur Ausgabe der Zelle 0000 auf dem Bildschirm wird das Unterprogramm BYTE benutzt, welches ein Byte als zwei Hex-Zeichen ausschreibt. Die Ausgabe eines Zeichens auf den Bildschirm geschieht durch eine zeitlich genau festgelegte Impulsfolge. Diese Folge würde durch einen Interrupt gestört werden. Um einen Interrupt in einem solchen zeitkritischen Programmabschnitt zu verhindern, wird der Abschnitt in die Befehle SEI (Set Interrupt Disable) und CLI (Clear Interrupt Disable) eingeschlossen.
Da Programm 1) - außer zwischen SEI und CLI - jederzeit unterbrochen werden kann, später aber wieder fortgesetzt werden soll, dürfen vom Programm 1) verwendete Register vom Interrupt-Bearbeitungsprogramm nicht verändert werden. Da das Status-

Register ohnehin gerettet wird, sind noch Akkumulator und eventuell X und Y-Register zu retten.

zu 2)

```
        .OR 3100
ASCI    .DL 1EA0    Unterprogramm ASCII-Zeichen Ausgabe
        PHA
        TXA
        PHA         Retten der Registerinhalte
        TYA
        PHA
        LDA 'I      ASCII-Code von I in den Akku laden
        JSR ASCI    I ausgeben
        PLA
        TAY
        PLA         Wiederherstellen der Registerinhalte
        TAX
        PLA
        RTI         Ende der Interrupt-Bearbeitung
        .EN
```

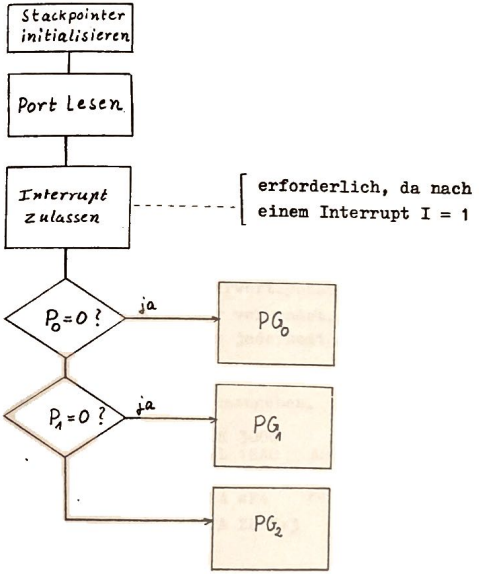
Bemerkung: Da im unterbrechbaren Teil von Programm 1) nur der Akkumulator (und nicht X- und Y-Register) benutzt wird, hätte in 2) das Retten des Akkumulators ausgereicht. Trotzdem sollte das Interrupt-Bearbeitungsprogramm möglichst alle Register retten, damit Programmerweiterungen ohne Schwierigkeiten möglich sind.

Beide Programme werden nungetrennt übersetzt. Vor dem Starten von Programm 1) wird noch die Anfangsadresse 3100 von Programm 2) in 17FE, 17FF (beim KIM, sonst FFFE, FFFF) eingetragen. Zum Auslösen des IRQ-Interrupts kann einfach die IRQ-Leitung auf Masse gelegt werden.

Soll mit Hilfe einer Tastatur eines von mehreren Programmen ausgewählt werden, so ist ein Vorgehen analog zum folgenden Beispiel sinnvoll.

Aufgabe: Im Mikrocomputer stehen 3 Programme PG_0, PG_1, PG_2 mit unendlichen Schleifen. Eines dieser Programme laufe. Ein IRQ-Interrupt bewirke, daß auf Programm PG_i umgeschaltet wird, wenn an Pin P_i eines Ports 0 anliegt (falls an P_0, P_1, P_2 eine 1 liegt, werde auf PG_2 umgeschaltet).

Interrupt-Bearbeitungsprogramm



Vor dem ersten Interrupt muß der "Interrupt-Vektor" geladen werden, d.h. beim KIM muß in 17FE, 17FF die Anfangsadresse des Interrupt-Bearbeitungsprogramms geladen werden. Ferner ist die Datenrichtungszelle so zu setzen (0 eingeben), daß über den Port eingelesen werden kann.

Bemerkung: 6502 und 6800 haben sehr ähnliche Interrupt-Techniken. Der 8080 sendet nach einem Interrupt ein Signal aus, der einen Eingabebaustein auffordert, über den Datenbus ein Byte zu senden. Dieses Byte wird als Operationscode eines Befehls gedeutet, der dann bearbeitet wird. Ein nicht-maskierbarer Interrupt (der also ohne Rücksicht auf eine Bedingung durchgeführt wird) ist beim 8080 nicht vorgesehen. Beim 8085 und Z80 ist ein nicht-maskierbarer Interrupt vorhanden. Neben der 8080-Interruptbeabeitung besteht beim 8085 und Z80 noch die Möglichkeit, nach einem Interrupt an einer festen Stelle in der Zero Page fortzufahren (ohne einen Befehl vom Datenbus zu lesen). Der Z80 bietet noch weitergehende Interruptmöglichkeiten.

8. Zeitgeber (Timer)

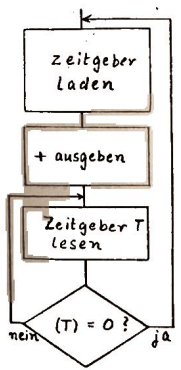
Zeitgeber-Bausteine enthalten Zähler, die vom Programm her geladen und jederzeit gelesen werden können. Ist ein Zähler abgelaufen, so sendet er ein Signal aus, das - bei entsprechender Beschaltung - einen Interrupt auslösen kann.
Der Ein/Ausgabe-Baustein 6530, den auch der KIM benutzt, enthält außer zwei Ports auch einen Zeitgeber. Der Zeitgeber ist ein 8 bit-Zähler, der in Intervallen von 1, 8, 64 und 1024 µs dekrementiert wird (unabhängig vom Programm !).
Das Laden des Zählers geschieht wie das Laden einer Speicherzelle. Die Adresse wird durch die Hardware festgelegt.

z.B. KIM: Laden des Zählers durch Laden der "Zelle"

```
1704  , falls  1 µs-Intervall erwünscht ist
1705  ,   "    8 µs-    "         "      "
1706  , falls 64 µs-    "         "      "
1707  , falls 1024 µs-Intervall erwünscht ist
```

Es handelt sich nicht um vier verschiedene Zähler, sondern um einen Zähler; die beiden niederwertigsten Adreßbits werden zur Definition der Intervalldauer verwendet.
Durch Lesen der "Zelle" 1706 kann jederzeit der Inhalt des Zählers abgefragt werden.

Aufgabe: Alle 0,25 s ist ein + auszugeben.




```
        .OR 3000
ASCI    .DL 1EA0    ASCII-Zeichen Ausgabe
ZEIT    .DL 1704    1. Zeitgeber-Adresse
ANFG    LDA #F4     F4 ≙ 244, 244×1024 µs
                        ≈ 0,25 s
        STA ZEIT+3

        LDA '+
        JSR ASCI

LIES    LDA ZEIT+2  Zelle 1706 lesen
        BNE LIES

        JMP ANFG
        .EN
```

Bemerkung: Der Zeitgeber kann nach dem Abwärtszählen auf 00 einen Interrupt erzeugen, wenn Bit 7 von Port B mit IRQ verbunden wird und statt

1704	170C
1705	170D
1706	170E
1707	170F

verwendet wird.

Zum Schluß wollen wir noch ein Beipiel bringen, in dem Zeitgeber-Benutzung und Interrupt-Bearbeitung verwendet werden.

Aufgabe: Reaktionstest (mit Interrupt)

1) Nach einigen Sekunden wird ein Zeichen ausgegeben und anschließend werden die 1/100 s dezimal auf Zelle 0000 gezählt.

2) Das Interrupt-Bearbeitungsprogramm schreibt Zelle 0000 aus.

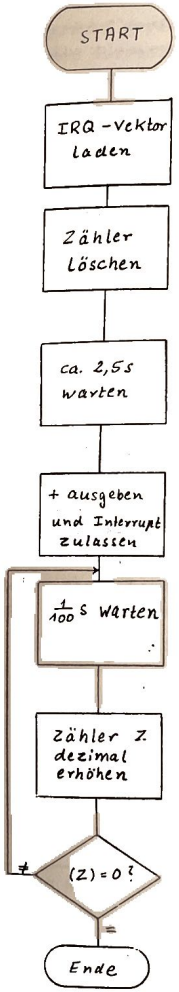
Da in diesem Beispiel nicht vorgesehen ist, daß Programm 1) nach der Interrupt-Bearbeitung fortgesetzt wird, braucht das Interrupt-Bearbeitungsprogramm keine Register zu retten.

zu 2)

```
        .OR 3100
ZAEL    .DL 0000        Zähler
BYTE    .DL 1E3B        Ausgabe eines Bytes
STOP    .DL 1C4F

        LDA ZAEL
        JSR BYTE
        JMP STOP        Sprung in den Monitor
        .EN
```

Die Anfangsadresse 3100 des Programms muß als "IRQ-Vektor" in 17FE, 17FF (beim KIM, sonst FFFE, FFFF) geladen werden.




```
        .OR 3000
ZAEL    .DL 0000        Zähler
ASCI    .DL 1EA0        Zeichenausgabe
STOP    .DL 1C4F        Monitor
IRQ     .DL 17FE        IRQ-Vektor
ZEIT    .DL 1704        Zeitgeber
        LDA #00
        STA IRQ
        LDA #31
        STA IRQ+1

        LDA #00
        STA ZAEL

        LDX #0A
WART    LDA #F4
        STA ZEIT+3      10x244x1024 µs
LIES    LDA ZEIT+2      warten
        BNE LIES
        DEX
        BNE WART

        LDA '+
        JSR ASCI
        CLI

WRT1    LDA #9C         9C ≙ 156
        STA ZEIT+2      156x64 µs ≈ 0,01 s
LS1     LDA ZEIT+2
        BNE LS1

        SED             Dezimalmodus setzen
        CLC
        LDA ZAEL
        ADC#01
        STA ZAEL
        CLD             Dezimalmodus löschen

        LDA ZAEL
        BNE WRT1

        JMP STOP
        .EN
```

TEIL II

GRUNDPROGRAMME

In diesem Teil werden Grundprogramme behandelt, die in der Mikrocomputertechnik häufig verwendet werden.
Die Programme werden in Programmablaufplänen oder Stukturdiagrammen dargestellt. Diese beschreiben die Programme so allgemein, daß danach Assemblerprogramme für Mikroprozessoren verschiedener Wortlänge (z.B. 8 bit, 16 bit) erstellt werden können.
Bei der Beschreibung der Grundprogramme stehen Symbole für Werte oder Speicherplätze, die Werte enthalten, nicht aber für Adressen (wie in den Beschreibungen zu Teil I). Ist also z.B. 1000 die Adresse des Speicherplatzes W, der 55 enthält, so wollen wir unter W+1 die Zahl 56 (und nicht 1001) verstehen.

1. Bearbeitung von Zeichen (im ASCII-Code)

Zur Darstellung von Zeichen wird in Mikrocomputern überwiegend der ASCII- Code*(American Standard Code for Information Interchange) verwendet.
Der ASCII-Code ordnet einem Zeichen ein 7 bit-Binärwort zu. Zum Abspeichern eines Zeichens wird in der Regel der ASCII-Code in die 7 niederwertigen Bits eines Bytes geschrieben, so daß Zeichen byteweise abgespeichert werden.

Die Übertragung eines Zeichens - z.B. zwischen Mikrocomputer und Peripheriegerät - kann parallel oder seriell geschehen. Bei der parallelen Übertragung werden 7+1 Leitungen zur Übertragung des ASCII-Codes benötigt (eventuell kommen noch einige Kontroll-Leitungen hinzu). Mit 2 Leitungen kommt man aus, wenn seriell übertragen wird. Die einzelnen Bits des ASCII-Codes werden zeitlich nacheinander gesendet (genaueres im folgenden Abschnitt).

---

*1. Der ASCII-Code ist - bis auf das $-Zeichen - gleich der internationalen Referenzversion des "Internationalen Alphabets Nr. 5" der CCITT.
2. Der ASCII-Code ist nicht zu verwechseln mit dem 8 bit-Code ASCII-8. Zwischen beiden Codes besteht aber ein einfacher Zusammenhang.

Die parallele Ein/Ausgabe des ASCII-Codes ist sehr einfach zu programmieren; man braucht nur die 7 Bits über einen Port einzulesen bzw. auszugeben.
Schwieriger ist die Programmierung der seriellen Übertragung. Bei der Ausgabe muß aus dem ASCII-Code eine zeitlich genau festgelegte Folge von Bits hergestellt werden (Parallel/Serien-Wandlung); bei der Eingabe muß aus einer solchen Folge der ASCII-Code ermittelt werden (Serien/Parallel-Wandlung).
Diese Parallel-Serien-Wandlung kann auch von einem speziellen Baustein übernommen werden, um den Mikroprozessor (bzw. das Programm) von der Aufgabe der Wandlung zu entlasten.

1.1 Serielle Ein/Ausgabe von ASCII-Zeichen

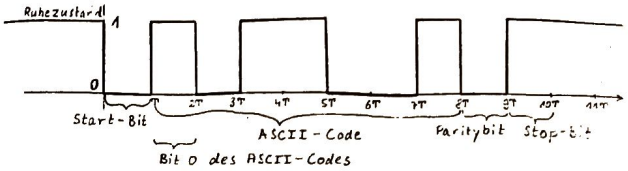
Für die serielle Übertragung von Zeichen im ASCII-Code existieren genauere Empfehlungen (vgl. [9]), nach denen sich die Hersteller von Peripheriegeräten richten. Dem codierten Zeichen wird eine zeitliche Folge von Bits zugeordnet; wobei jedes Bit diegleiche Zeit T in Anspruch nimmt.
Die Folge sieht so aus:

1. Eine 0 als Start-Bit.
2. Die 7 Bits des ASCII-Codes, beginnend mit dem niederwertigen Bit.
3. Ein Paritätsbit.
   Zur Fehlererkennung werden die 7 Bits des ASCII-Codes durch ein Paritätsbit (gerade oder ungerade ist nicht festgelegt) ergänzt. Wird auf Fehlererkennung keinen Wert gelegt, so kann dieses Bit beliebig festgelegt werden, z.B. 0. Das Paritätsbit wird aber nicht fortgelassen.
4. Eine 1 als Stop-Bit.
   Bei langsameren elektromechanischen Geräten (bis 200 bit/s) kommt noch eine 1 als zweites Stop-Bit hinzu.

Die Zeit T für die Dauer eines Bits hängt von der gewünschten Übertragungsgeschwindigkeit ab, z.B:

T ≈ 3,33 ms	bei 300 Baud	Baud = bit/s
T ≈ 417 µs	bei 2400 Baud	

Bit-Folge zur seriellen Übertragung des ASCII-Codes:



Nach der Übertragung eines Zeichens wird solange eine 1 übertragen (eventuell über längere Zeit) bis ein Start-Bit die Übertragung eines neuen Zeichens einleitet. (Start-Stop-Betrieb).

Bemerkung: Empfänger, die eine Bit-Folge mit einem Stop-Bit empfangen können, verarbeiten selbstverständlich auch die Bit-Folge mit 2 Stop-Bits. Empfängt daher der Mikrocomputer die Bit-Folgen mit einem Stop-Bit korrekt und sendet der Mikrocomputer Bit-Folgen mit 2 Stop-Bits, so treten wegen der verschiedenen Anzahl der Stop-Bits keine Probleme auf.

Die Übertragung des ASCII-Codes durch die angegebenen Bit-Folgen geschieht über serielle Schnittstellen. Bei der V 24-Schnittstelle (≙ RS 232) wird die Bitfolge durch Spannungssignale (0 ≙ 3 V bis 15 V, 1 ≙ -15 V bis -3 V), bei der TTY-Schnittstelle (TTY = Teletype) durch Stromsignale übertragen (0 ≙ 0 A, 1 ≙ 20 mA).
Diese Schnittstellen-Signale können über Port-Bits gesteuert oder empfangen werden, wenn geeignete Schaltungen die Wandlung von TTL-Signalen (0 ≙ 0-0,8 V, 1 ≙ 2-5 Volt) in Schnittstellensignale und umgekehrt vornehmen.

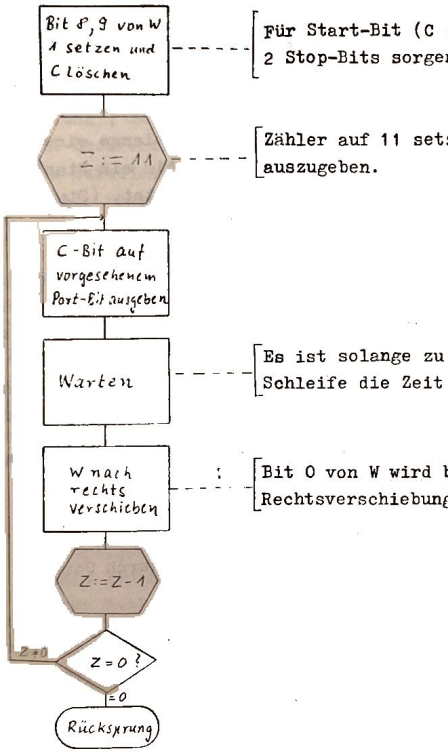
Serielle Ausgabe eine ASCII-Zeichens

Der ASCII-Code (einschließlich Paritybit) stehe rechtsbündig in einem n bit-Speicherplatz W mit n ≥ 10. Bit 0 von W werde durch logische Rechtsverschiebung von W in die Flag C transportiert.
Die Steuerung der seriellen Übertragung geschehe durch ein Bit eines Ports.

---

*Bei einer Wortlänge r mit 7 ≤ r <10 (z.B. r=8) zerlegt man W in zwei Speicherworte W1, W2. In W2 stehe dann rechtsbündig der ASCII-Code.

Programmablaufplan zur Ausgabe von ASCII-Zeichen:



Bemerkung: Im Unterprogramm wurde Wert darauf gelegt, daß die Ausgabe eines jeden Bits - bis auf das unkritische letzte Stop-Bit diegleiche Zeit beansprucht. Dies ist bei der Verwendung höherer Übertragungsgeschwindigkeiten wesentlich.

Serielles Einlesen von ASCII-Zeichen

Der ASCII-Code (einschließlich Paritybit) eines Zeichens soll seriell eingelesen werden und auf einem n bit-Speicherplatz W ($n \geq 8$) linksbündig gespeichert werden.
Die Eingabe geschehe über ein Bit eines Ports.

Das Einlesen des ASCII-Codes geschieht so: Man wartet zunächst bis zum Beginn des Start_Bits; nach der Zeit für 1 1/2 Bits liest man in 1 Bit-Abständen die 8 Bits (ASCII-Code + Paritybit) ein. Nach dem Einlesen des 8-ten Bits wartet man durch einen weiteren 1 Bit-Abstand noch das (erste) Stop-Bit ab.

Wir wollen hier ein Unterprogramm beschreiben, welches nicht auf einen Zeitgeber zurückgreift. Ferner soll Wert darauf gelegt werden, daß die Anpassung an verschiedene Baudraten nur an einer Stelle, nämlich in einem Warte-Unterprogramm erfolgen muß.

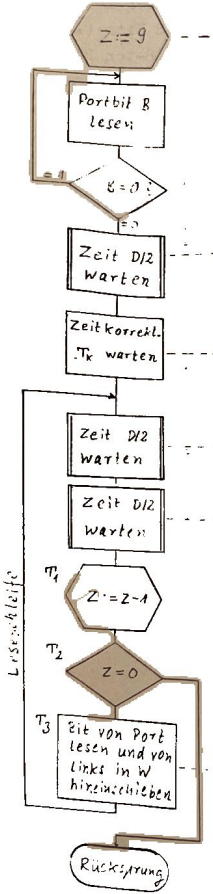
Bemerkungen zur Programmbeschreibung auf der nächsten Seite:

1. Wegen $T_k = 3/2\ T - T_1 - T_2 - 3/2\ D = 3/2\ T - T_1 - T_2 - 3/2(T - T_1 - T_2 - T_3) = 1/2\ T_1 + 1/2\ T_2 + 3/2\ T_3$ ist die Zeitkorrektur unabhängig von der Baudrate !
2. In vielen Fällen wird T_k klein gegen T sein. Dann kann die Zeitkorrektur fortgelassen werden.
3. Die Anpassung an eine andere Baudrate geschieht nur in dem Unterprogramm "D/2 warten", so daß Ausführung + Unterprogrammaufruf die Zeit $D/2 = (T - T_1 - T_2 - T_3)/2$ beanspruchen.

Programmablaufplan zum Einlesen eines ASCII-Zeichens:

T = vorgeschriebene Zeit für die Dauer eines Bits

T_1, T_2, T_3 sind Ausführungszeiten einiger Anweisungen (siehe Ablaufplan)

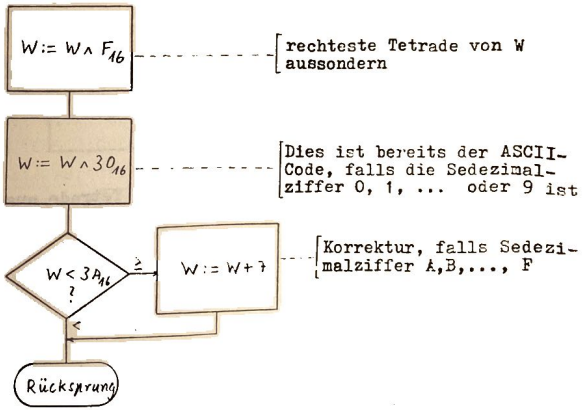


1.2 Umwandlung von Sedezimalziffern: Dual - ASCII-Code

Den Inhalt von Speicherplätzen gibt man gern als Sedezimalzahl an, indem man den Speicherplatz in Tetraden aufteilt und jede Tetrade (die ja 16 Zahlen darstellen kann) als Sedezimalziffer auffaßt.
Ein/Ausgabe der Sedezimalziffern erfordern Wandlungen in den ASCII-Code und umgekehrt. Dieselben Wandlungen sind erforderlich, wenn Dezimalziffern ein- bzw. auszugeben sind.

Umwandlung einer Sedezimalziffer: Dualzahl → ASCII-Code

Die rechteste Tetrade eines n bit-Speicherplatzes W ($n \geq 7$) ist als Sedezimalziffer aufzufassen. Der ASCII-Code dieser Sedezimalziffer ist rechtsbündig in W abzuspeichern (für W kann z.B. der Akkumulator gewählt werden).

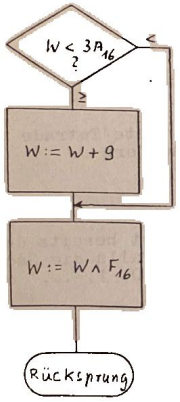


Umwandlung einer Sedezimalziffer:ASCII-Code → Dualzahl

Der ASCII-Code einer Sedezimalziffer stehe rechtsbündig in einem n bit-Speicherplatz W (n ≥ 7). Die Dualdarstellung der Sedezimalziffer ist rechtsbündig in W zu speichern (für W kann z.B. der Akkumulator gewählt werden).

Besonders einfach ist die Umwanlungsroutine, wenn nicht geprüft wird, ob der vorgelegte ASCII-Code tatsächlich eine Sedezimalziffer darstellt: Bei den Ziffern 0, 1, ..., 9 liefert die Tetrade der 4 niederwertigen Bits bereits die gewünschte Dualzahl; bei den Ziffern A, B, ..., F muß zu dieser Tetrade noch 9 addiert werden.

Programmablaufplan zur Umwandlung einer Sedezimalziffer vom ASCII-Code in eine Dualzahl (ohne Fehlerprüfung):



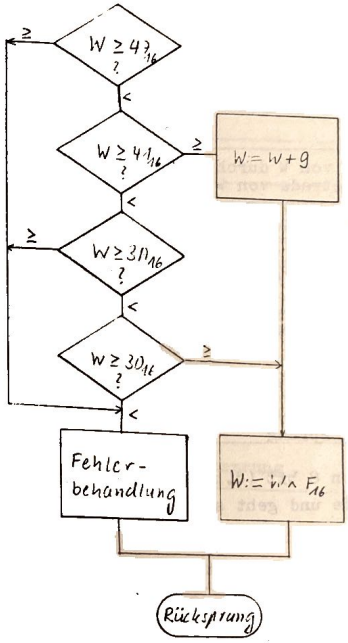
Prüfen, ob 0, 1, ... , 9

Korrektur, falls A,B, ..., F

Rechte Tetrade aussondern

Soll auch geprüft werden, ob der ASCII-Code eine Sedezimalziffer darstellt, so ist noch zu erkennen, ob der Code zwischen 30 und 39 oder zwischen 41 und 46 liegt.

Programmablaufplan zur Umwandlung einer Sedezimalziffer vom ASCII-Code in eine Dualzahl (mit Fehlerprüfung):



Falls W ≥ 47 liegt keine Sedezimalziffer vor

Korrektur, falls W < 47 und W ≥ 41, da dann A, B, ..., F vorliegt.

Falls W < 41 und W ≥ 3A, liegt keine Sedezimalziffer vor.

Falls W < 3A und W ≥ 30, liegt 1,2,...,9 vor; falls W < 30 keine Sedezimalz.

Aussondern der rechten Tetrade

1.3 Ein/Ausgabe eines Speicherworts (als Sedezimalzahl)

Unter Speicherwort wollen wir hier irgendein n bit-Register oder einen n bit-Speicherplatz des Hauptspeichers verstehen; wobei n die Wortlänge des Mikrocomputers ist.

Ausgabe eines Speicherworts

Das n bit-Speicherwort W bestehe aus m Tetraden, d.h. n = 4m. Die m Tetraden von W sind von links nach rechts als Sedezimalziffern (im ASCII-Code) auszugeben.

Strukturdiagramm zur Ausgabe eines Speicherwortes W:
(Für W kann z.B. der Akkumulator gewählt werden)

wiederhole m mal
linkeste Tetrade von W durch Linksrotationen zur rechtesten Tetrade von W machen.
S := W (W retten z.B. auf Stack)
rechteste Tetrade von W umwandeln in den ASCII-Code, der dann rechtsbündig in W steht (vgl. 1.2)
Ausgabe des ASCII-Codes, der rechtsbündig in W steht. (Vgl. 1.1)
W := S (W wiederherstellen)
Rücksprung

Bei einer Wortlänge von 8 bit (d.h. m=2) verzichtet man besser auf die Schleife und geht so vor:

S := W (W retten z.B. auf Stack)
linke Tetrade von W zur rechten Tetrade von W machen.
rechte Tetrade von W ausgeben (vgl. 1.2 und 1.1)
W := S (W wiederherstellen)
rechte Tetrade von W ausgeben
Rücksprung

Eingabe eines Speicherworts (als Sedezimalzahl)

Das n bit-Speicherwort (n ≥ 8) W bestehe aus m Tetraden, d.h. n = 4m . m Sedezimalziffern, die im ASCII-Code eingegeben werden, sind (dual) von links nach rechts in W zu speichern (für W kann z.B. der Akkumulator gewählt werden).

Strukturdiagramm zur Eingabe eines Speicherworts:

H sei ein Hilfsspeicherwort.

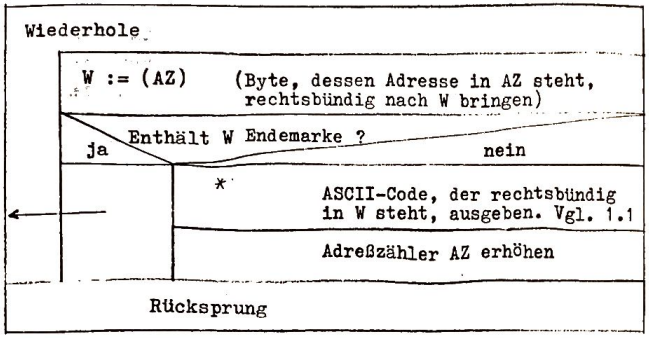
wiederhole m mal
Hilfsspeicherwort W um 4 Bits logisch nach links verschieben
Eingabe einer Sedezimalziffer im ASCII-Code nach W (rechtsbündig). Vgl. 1.1
Umwandlung der Sedezimalziffer vom ASCII-Code in die Dualdarstellung (rechtsbündig in W). Vgl. 1.2
H := H ∨ W (Zusammensetzen der Tetraden durch Oder-Funktion)
Rücksprung

1.4 Ein/Ausgabe eines Textes

Ausgabe eines Textes

Der Text stehe byteweise ASCII-codiert im Hauptspeicher
Der Text beginne in einem Byte, dessen Adresse in einem Adreßzähler AZ steht. Der Text ende mit einer Endmarke in Form eines speziellen ASCII-Codes (z.B. ETX = End of Text ≙ 03_{16})
Es wird angenommen, daß der Mikrocomputer Bytes adressieren kann.

Strukturdiagramm zur Ausgabe eines Textes:



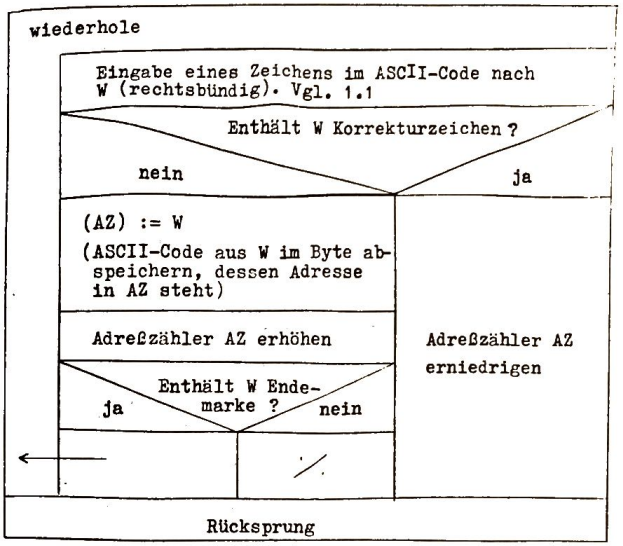
Bemerkung: Soll während der Textausgabe auf spezielle ASCII-Zeichen besonders reagiert werden (z.B auf ein Tabulator-Zeichen, so wird Block * entsprechend modifiziert.

Eingabe eines Textes

Da ein Text oft fehlerhaft und unvollständig eingegeben wird, muß er meist korrigiert und ergänzt werden. Dies besorgt ein Text-Editor; häufig ein ziemlich umfangreiches Programm. Das folgende Unterprogramm ist deshalb nur ein Grundbaustein zur Texteingabe.

Beginnend bei einem Byte, dessen Adresse in einem Adreßzähler AZ steht, soll Text bis zu einer Endmarke E (einschließlich E) in aufeinanderfolgenden Bytes abgespeichert werden. Nach Eingabe eines Korrekturzeichens K (bei Bildschirmgeräten in der Regel "Cursor 1 Position nach links") werde das zuletzt eingegebene Zeichen überschrieben.
W sei ein n bit-Speicherplatz mit n ≥ 8 (z.B. der Akkumulator) Der Mikrocomputer lasse sich byteweise adressieren. Der Adreßzähler AZ enthalte nach dem Ende der Eingabe die Adresse des Bytes hinter E.

Strukturdiagramm zur Eingabe eines Textes:



2. Ganzzahlarithmetik (Multiplikation, Division)

In diesem Kapitel sollen Multiplikation und Division ganzer Zahlen auf Additionen bzw. Subtraktionen zurückgeführt werden.
Für Prozessoren, die über Multiplikations- und Divisionsbefehle verfügen, ist dieses Kapitel unwesentlich. Auch wenn z.B. eine Multiplikation mit mehrfacher Genauigkeit gewünscht wird, verwendet man nicht die hier angegebenen Algorithmen, sondern führt die Multiplikation mit mehrfacher Genauigkeit auf Multiplikationen mit einfacher Genauigkeit zurück. (Vgl. Knuth [7], Vol 2, 4.3)

2.1 Multiplikation nicht-negativer Dualzahlen

Sind $a_{n-1}, a_{n-2}, \ldots, a_1, a_0$ die Ziffern (Bits) der n bit-Dualzahl a, so ist $a = a_{n-1}2^{n-1}+a_{n-2}2^{n-2}+\ldots+a_1 2^1+a_0 2^0$

und $a \cdot b = a_{n-1}b\cdot 2^{n-1}+a_{n-2}b\cdot 2^{n-2}+\ldots+a_1 b\cdot 2^1+a_0 b$

$$=(\ldots(a_{n-1}b)2+a_{n-2}b)2+\ldots+a_1 b)2+a_0 b$$

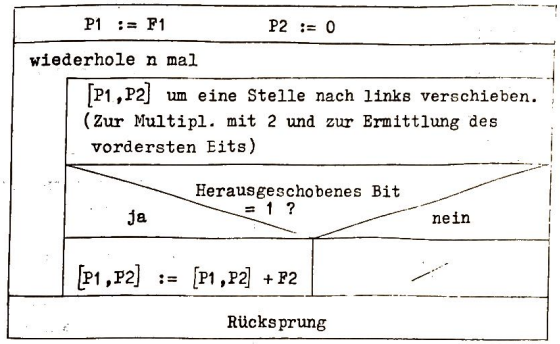
Für die Multiplikation von Dualzahlen ergibt sich also folgendes Verfahren:

1. Vorderste Dualziffer von a (a_{n-1}) mit b multiplizieren. Das Ergebnis ist 0 oder b.
2. Das Ergebnis mit 2 multiplizieren (≙ Linksverschiebung)
3. Nächste Dualziffer (a_{n-2}) mit b multiplizieren und zum verschobenen Ergebnis addieren.
4. Ergebnis mit 2 multiplizieren (≙ Linksverschiebung) u.s.w. bis $a_0 b$ addiert wurde.

Dieses Verfahren wird im folgenden Unterprogramm verwendet.

Die Faktoren a, b mögen in n bit-Speicherbereichen F1, F2 stehen. Für das Produkt sei ein 2n bit-Speicherbereich P vorgesehen, der aus zwei n bit-Speicherbereichen P1, P2 besteht. Für P wollen wir auch [P1,P2] schreiben.

Strukturdiagramm zur Multiplikation nicht-negativer Dualzahlen:



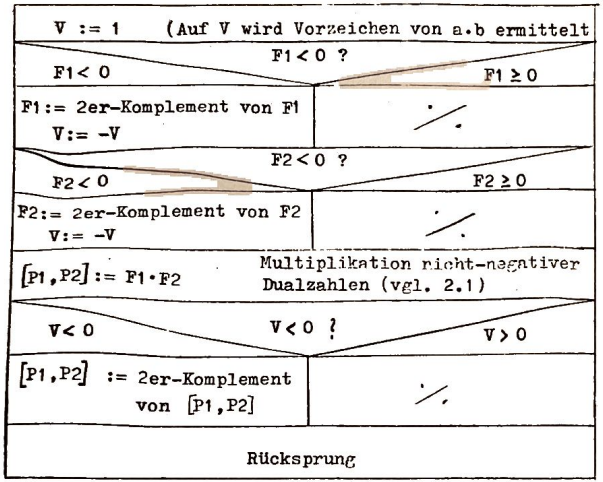
Bemerkung: 1. Das Programm kann auch benutzt werden, wenn Länge von F2 = Länge von P2 = m mit $m \neq n$.
2. Für F1 und P1 kann derselbe Speicherplatz verwendet werden; natürlich wird dann der Faktor a in F1 zerstört.
3. Werden für a, b auch negative Zahlen zugelassen, die als 2er-Komplement ihres Betrages dargestellt werden (vgl. den nächsten Abschnitt), so liegt in P2 das richtige Ergebnis vor, sofern sich a·b mit n Dualstellen darstellen läßt, d.h. $-2^{n-1} \leq a\cdot b \leq 2^{n-1}-1$.

2.2 Multiplikation ganzer Dualzahlen (mit Vorzeichen)

Im letzten Abschnitt hatten wir angenommen, daß in einem n bit-Speicherbereich die Zahlen $0,1,\ldots,2^n-1$ dargestellt werden. Wir wollen nun auch negative Zahlen a zulassen und stellen diese - wie üblich in der Datenverarbeitung - als 2er-Komplement des Betrages dar: $2^n-|a| = 2^n+a$. In einem n bit-Speicherbereich sind dann die ganzen Zahlen x mit $-2^{n-1} \leq x \leq 2^{n-1}-1$ darstellbar.

Man kann nun die Multiplikation a·b auf die Berechnung von $|a|\cdot|b|$ mit anschließender Vorzeichenkorrektur zurückführen:

Strukturdiagramm zur 1. Methode der Multiplikation ganzer Dualzahlen (mit Vorzeichen): (Bezeichnungen wie in 2.1)



Bemerkung: V kann nur zwei Werte annehmen und kann daher als Boolsche Variable und -V als Negation aufgefaßt werden.

Eine bessere Lösung als die Rückführung von $a \cdot b$ auf $|a| \cdot |b|$ erhält man, indem man zunächst - ohne Rücksicht auf das Vorzeichen - die Multiplikation der Dualzahlen (wie in 2.1) durchführt und anschließend das Ergebnis korrigiert. Die erforderliche Korrektur soll nun hergeleitet werden.

In den n bit-Speicherbereichen F1, F2 stehen die Zahlen a, b. Negative Zahlen werden im 2er-Komplement $2^n - |a| = 2^n + a$ dargestellt. Das Ergebnis soll eine 2n bit-Zahl sein, wobei ein negatives Ergebnis $a \cdot b$ wiederum im 2er-Komplement $2^{2n} - |a \cdot b| = 2^{2n} + a \cdot b$ dargestellt wird.
Werden die Zahlendarstellungen A, B von a, b als nicht-negative Dualzahlen aufgefaßt und multipliziert, so ergibt sich:

$$A \cdot B = \begin{cases} a \cdot b & a, b \geq 0 \\ (2^n + a) \cdot b = 2^n b + a \cdot b & a < 0,\ b \geq 0 \\ a \cdot (2^n + b) = 2^n a + a \cdot b & a \geq 0,\ b < 0 \\ (2^n + a)(2^n + b) = 2^{2n} + 2^n a + 2^n b + a \cdot b & a < 0,\ b < 0 \end{cases}$$

Beachtet man, daß Überträge über das 2n-te Bit fortgelassen werden (Rechnung mod 2^{2n}), so erhält man das gewünschte Ergebnis, indem man $A \cdot B$ folgendermaßen korrigiert:

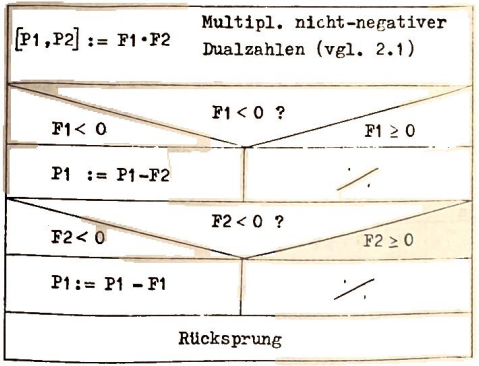
$A \cdot B$	, falls $a, b \geq 0$	
$A \cdot B - 2^n B$	, falls $a < 0, b \geq 0$	$(B = b)$
$A \cdot B - 2^n A$	, falls $a \geq 0, b < 0$	$(A = a)$
$A \cdot B - 2^n A - 2^n B$	, falls $a < 0, b < 0$	$(A = 2^n + a,\ B = 2^n + b)$

Die Richtigkeit der letzten Korrektur folgt aus der Gleichung

$$\begin{aligned} A \cdot B - 2^n A - 2^n B &= (2^n + a)(2^n + b) - 2^n(2^n + a) - 2^n(2^n + b) \\ &= 2^{2n} + 2^n a + 2^n b + a \cdot b - 2^{2n} - 2^n a - 2^{2n} - 2^n b = a \cdot b - 2^{2n} \\ &= a \cdot b \bmod 2^{2n} \end{aligned}$$

Die angegebenen Korrekturen führen zu folgendem Algorithmus.

Strukturdiagramm zur 2.Methode der Multiplikation ganzer Dualzahlen(mit Vorzeichen): (Bezeichnungen wie in 2.1)



Bemerkung: 1. Die zuletzt angegebene Methode ist nur richtig, wenn die Speicherbereiche F1, F2 gleich groß sind (n Bits).
2. Die n niederwertigen Bits des Produkts werden nicht korrigiert (vgl. auch Bemerkung 3 in 2.1).

2.3 Division nicht-negativer Dualzahlen

Zur Ermittlung n-stelligen Quotienten
$q = q_{n-1}2^{n-1} + \ldots + q_1 2^1 + q_0 2^0$ ($q_{n-1},\ldots,q_1,q_0$ Dualziffern von q) der n-stelligen Dualzahl a durch die Dualzahl b, wird nach folgenden Formeln vorgegangen:

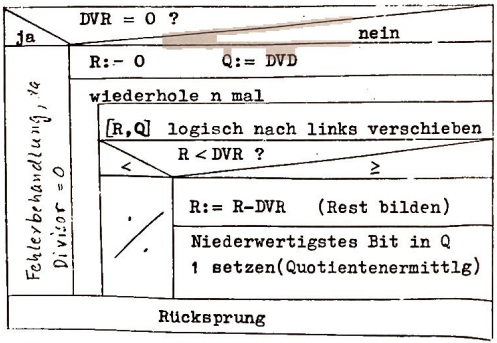
$$r_{n-1} = a - q_{n-1}(b \cdot 2^{n-1})$$
$$r_{n-2} = r_{n-1} - q_{n-2}(b \cdot 2^{n-2})$$
$$\vdots$$
$$r_0 = r_1 - q_0(b \cdot 2^0)$$

Man ermittelt zunächst, wie oft $b \cdot 2^{n-1}$ in a enthalten ist (q_{n-1}) und bildet den Rest r_{n-1}. Dann ermittelt man, wie oft $b \cdot 2^{n-2}$ im Rest r_{n-1} enthalten ist (q_{n-2}) u.s.w.
$r_0 = r_1 - q_0 b = a - q_{n-1} b \cdot 2^{n-1} - q_{n-2} b \cdot 2^{n-2} - \ldots - q_1 b \cdot 2 - q_0 b$
$= a - q \cdot b$ ist dann der Divisionsrest.

Auf dem angegebenen Divisions-Algorithmus beruht folgendes Programm:

Der Dividend a und der Divisor b mögen als Dualzahlen in den n bit-Speicherbereichen DVD und DVR stehen. Für das Ergebnis seien zwei n bit Speicherbereiche vorgesehen; Q für den Quotienten, R für den Divisionsrest. Für den 2n-bit-Speicherbereich, der aus R und Q (aufgefaßt als Einheit) besteht, wollen wir [R,Q] schreiben.

Strukturdiagramm zur Division nicht-negativer Dualzahlen:



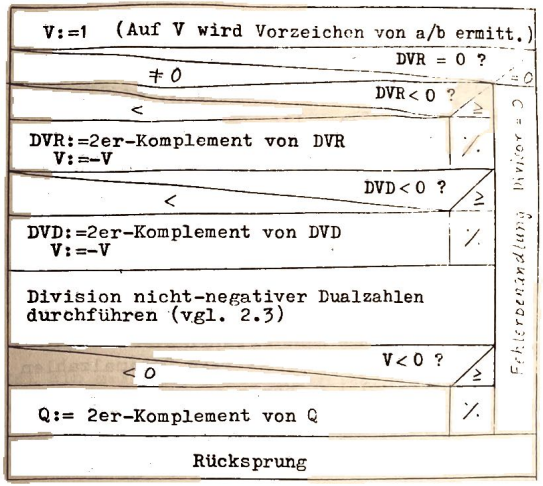
Bemerkung: 1. Das Programm kann auch benutzt werden, wenn Länge von Q = Länge von DVD = m ≠ n.
2. Für Q und DVD kann derselbe Speicherplatz verwendet werden; natürlich wird dann der Dividend in DVD zerstört und schrittweise durch den Quotienten ersetzt.

2.4 Division ganzer Dualzahlen (mit Vorzeichen)

Die Division ganzer Dualzahlen a/b führt man auf die Division der nicht-negativen Dualzahlen |a| / |b| zurück, anschließend korrigiert man das Vorzeichen.
Das folgende Programm ermittelt den Quotienten q und den Betrag |r| des Rests r, so daß a = b·q+r mit |r| < b. Negative Zahlen werden als 2er-Komplement des Betrags dargestellt.

Strukturdiagramm zur Division ganzer Dualzahlen (mit Vorzeichen):

(Bezeichnungen wie in 2.3)



Bemerkung: In R wird nicht der Rest r, sondern r berechnet. Wird auf einen vorzeichenrichtigen Rest Wert gelegt, so muß noch das 2er-Komplement von R gebildet werden, falls der Dividend negativ ist.

3. Konversion ganzer Zahlen: Dual-Dezimal

Zur Darstellung von Dezimalzahlen im Computer wird meist ein BCD-Code (Binary Coded Decimal Code) verwendet, bei dem jede einzelne Dezimalziffer binär dargestellt wird. Wir wollen hier als BCD-Code die übliche Tetradendarstellung verwenden, bei der jeder Dezimalziffer die entsprechende 4-stellige Dualzahl zugeordnet wird.
Wir wollen in diesem Kapitel annehmen, daß alle Zahlen nicht-negativ sind. Die Behandlung negativer Zahlen erfordern nur einfache Ergänzungen.

3.1 Wandlung von Dualzahlen in Dezimalzahlen (BCD-Code)

Zur Darstellung von n-stelligen Dualzahlen im Dezimalsystem werden

$$r = \mathrm{Int}(1+\log_{10} 2^n) \qquad ; \ \mathrm{Int}(x) = \text{größte ganze Zahl } k \text{ mit } k \le x.$$
$$= \mathrm{Int}(1+n\cdot\log_{10} 2)$$
$$\approx \mathrm{Int}(1+0{,}3\,n)$$

Dezimalstellen benötigt.
Die Umwandlung einer n-stelligen Dualzahl $a=a_{n-1}2^{n-1}+\ldots+a_1 2^1+a_0 2^0$ ($a_{n-1},\ldots,a_1,a_0$ Dualziffern) kann nach der Formel

$$a = ((\ldots(a_{n-1}\cdot 2+a_{n-2})2+\ldots+a_1)2+a_0$$

geschehen. Dabei sind die Rechenoperationen (Multiplikation mit 2 und Addition einer Dualziffer) dezimal durchzuführen. Da die meisten Mikroprozessoren eine dezimale Addition gestatten, ist die angegebene Formel leicht anzuwenden: *

DUAL sei ein n bit-Speicherplatz, auf dem eine nicht-negative Dualzahl gespeichert ist. DEZ sei ein m bit-Speicherbereich (m=4r), der r-ziffrige Dezimalzahlen

($r \ge \mathrm{Int}(1+n\cdot\log_{10} 2)$) im BCD-Code speichern kann.

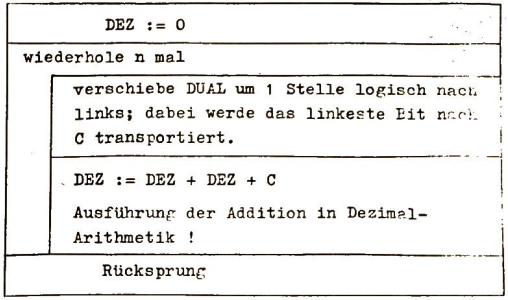
---

*Eine andere Methode, bei der nur dual gearbeitet wird, berechnet die Dezimalziffern $b_{r-1},\ldots,b_1,b_0$ nach den Formeln

$$a/10 = q_0 \text{ Rest } b_0$$
$$q_0/10 = q_1 \text{ Rest } b_1$$
$$q_1/10 = q_2 \text{ Rest } b_2$$
$$\vdots$$

Abgebrochen wird, wenn $q_i = 0$.

Strukturdiagramm zur Umwandlung von Dualzahlen in Dezimalzahlen (BCD-Code):



3.2 Wandlung von Dezimalzahlen (BCD-Code) in Dualzahlen

Zur Darstellung von r-ziffrigen Dezimalzahlen als Dualzahlen werden

$$n = \mathrm{Int}(1+\log_2 10^r) \qquad ; \ \mathrm{Int}(x) = \text{größte ganze Zahl } k \text{ mit } k \le x.$$
$$= \mathrm{Int}(1+r/\log_{10} 2^2)$$
$$\approx \mathrm{Int}(1+3{,}32\,r)$$

Dualstellen benötigt.
Die Umwandlung einer r-stelligen Dezimalzahl $b=b_{r-1}10^{r-1}+\ldots+b_1 10+b_0$ ($b_{r-1},\ldots,b_1,b_0$ Dezimalziffern von b) geschieht - analog zu 3.1 - nach der Formel:

$$b = ((\ldots(b_{r-1}\cdot 10+b_{r-2})10+\ldots+b_1)10+b_0$$

Dabei sind die Rechenoperationen dual durchzuführen.

DEZ sei ein m=4r bit-Speicherbereich, auf dem eine nicht-negative Dezimalzahl im BCD-Code (tetradenweise) gespeichert ist. DUAL sein ein n bit-Speicherbereich mit $n \ge \mathrm{Int}(1+r/\log_{10} 2^2)$.

Strukturdiagramm zur Umwandlung von Dezimalzahlen (BCD-Code) in Dualzahlen:

DUAL := 0
wiederhole r mal (r Anzahl der Dezimalziffern)
DUAL := DUAL·1010_2 * (Multiplikation mit 10 im Dualsystem)
Linkeste Tetrade von DEZ durch Linksrotationen zur rechtesten Tetrade machen.
DUAL := DUAL + DEZ $\wedge F_{16}$ (Rechteste Tetrade von DEZ aussondern und zu DUAL addieren)
Rücksprung

*Die Multiplikation von DUAL mit 10 kann folgendermaßen durchgeführt werden:

DUAL um 1 Stelle logisch nach links verschieben
HILF := DUAL (HILF Hilfsspeicherbereich)
DUAL um 2 Stellen logisch nach links verschieben
DUAL := DUAL + HILF

Literatur

Zu Teil I:

8080, 6800, 6502

1. Schmitt, G.: Maschinenorientierte Programmierung für Mikroprozessoren. Oldenbourg Verlag, München Wien 1979
2. Martin, W.: Mikrocomputer in der Prozeßdatenverarbeitung. Hanser-Verlag, München Wien 1977 (hardware-orientiert)
3. Programmier-Fibel, MOS-Technology. MCDS Microcomputer Datensysteme, Darmstadt 1977
3. Hardware-Handbuch, MOS-Technology. MCDS Microcomputer Datensysteme, Darmstadt 1977
5. KIM-1 User Manual (Deutsche Übersetzung). MCDS Microcomputer Datensysteme, Darmstadt 1977
6. KIMath Subroutines Programming Manual. MCDS Microcomputer Datensysteme, Darmstadt

Zu Teil II:

7. Knuth, D.E.: The Art of Computer Programming, Vol 2: Seminumerical Algorithms. Addison-Wesley Pblishing Company 1969
8. Siemens: Programmbibliothek SAB 8080
   Band 1:Grundrechenarten 1
   Band 2: Code-Umwandlungen 1
   Band 3: Grundrechenarten 2
9. CCITT-Empfehlungen der V-Serie und der X-Serie:Datenübertragung. R.v.Deckers Verlag, G. Schenk, Heidelberg Hamburg 1 Heidelberg Hamburg 1977

Zeitschrift: BYTE, the small systems journal. Byte Publications, Peterborough

ANHANG

R6500 microprocessor instruction set

		Execution Time (clock cycles)												
		Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect), Y	Absolute Indirect
ADC	Add Memory to Accumulator with Carry	•	2	3	4	•	4	4*	4*	•	•	6	5*	•
AND	"AND" Memory with Accumulator	•	2	3	4	•	4	4*	4*	•	•	6	5*	•
ASL	Shift Left One Bit (Memory or Accumulator)	2	•	5	6	•	6	7	•	•	•	•	•	•
BCC	Branch on Carry Clear	•	•	•	•	•	•	•	•	•	2**	•	•	•
BCS	Branch on Carry Set	•	•	•	•	•	•	•	•	•	2**	•	•	•
BEQ	Branch on Result Zero	•	•	•	•	•	•	•	•	•	2**	•	•	•
BIT	Test Bits in Memory with Accumulator	•	•	3	•	•	4	•	•	•	•	•	•	•
BMI	Branch on Result Minus	•	•	•	•	•	•	•	•	•	2**	•	•	•
BNE	Branch on Result not Zero	•	•	•	•	•	•	•	•	•	2**	•	•	•
BPL	Branch on Result Plus	•	•	•	•	•	•	•	•	•	2**	•	•	•
BRK	Force Break	•	•	•	•	•	•	•	•	•	•	•	•	•
BVC	Branch on Overflow Clear	•	•	•	•	•	•	•	•	•	2**	•	•	•
BVS	Branch on Overflow Set	•	•	•	•	•	•	•	•	•	2**	•	•	•
CLC	Clear Carry Flag	•	•	•	•	•	•	•	•	2	•	•	•	•
CLD	Clear Decimal Mode	•	•	•	•	•	•	•	•	2	•	•	•	•
CLI	Clear Interrupt Disable Bit	•	•	•	•	•	•	•	•	2	•	•	•	•
CLV	Clear Overflow Flag	•	•	•	•	•	•	•	•	2	•	•	•	•
CMP	Compare Memory and Accumulator	•	2	3	4	•	4	4*	4*	•	•	6	5*	•
CPX	Compare Memory and Index X	•	2	3	•	•	4	•	•	•	•	•	•	•
CPY	Compare Memory and Index Y	•	2	3	•	•	4	•	•	•	•	•	•	•
DEC	Decrement Memory by One	•	•	5	6	•	6	7	•	•	•	•	•	•
DEX	Decrement Index X by One	•	•	•	•	•	•	•	•	2	•	•	•	•
DEY	Decrement Index Y by One	•	•	•	•	•	•	•	•	2	•	•	•	•
EOR	"Exclusive OR" Memory with Accumulator	•	2	3	4	•	4	4*	4*	•	•	6	5	•
INC	Increment Memory by One	•	•	5	6	•	6	7	•	•	•	•	•	•
INX	Increment Index X by One	•	•	•	•	•	•	•	•	2	•	•	•	•
INY	Increment Y by One	•	•	•	•	•	•	•	•	2	•	•	•	•
JMP	Jump to New Location	•	•	•	•	•	3	•	•	•	•	•	•	5
JSR	Jump to New Location saving Return Address	•	•	•	•	•	6	•	•	•	•	•	•	•
LDA	Load Accumulator with Memory	•	2	3	4	•	4	4*	4*	•	•	6	5*	•
LDX	Load Index X with Memory	•	2	3	•	4	4	•	4*	•	•	•	•	•
LDY	Load Index Y with Memory	•	2	3	4	•	4	4*	•	•	•	•	•	•
LSR	Shift Right One Bit (Memory or Accumulator)	2	•	5	6	•	6	7	•	•	•	•	•	•
NOP	No Operation	•	•	•	•	•	•	•	•	2	•	•	•	•
ORA	"OR" Memory with Accumulator	•	2	3	4	•	4	4*	4*	•	•	6	5*	•
PHA	Push Accumulator on Stack	•	•	•	•	•	•	•	•	3	•	•	•	•
PHP	Push Processor Status on Stack	•	•	•	•	•	•	•	•	3	•	•	•	•
PLA	Pull Accumulator from Stack	•	•	•	•	•	•	•	•	4	•	•	•	•
PLP	Pull Processor Status from Stack	•	•	•	•	•	•	•	•	4	•	•	•	•
ROL	Rotate One Bit Left (Memory or Accumulator)	2	•	5	6	•	6	7	•	•	•	•	•	•
ROR	Rotate One Bit Right (Memory or Accumulator)	2	•	5	6	•	6	7	•	•	•	•	•	•
RTI	Return from Interrupt	•	•	•	•	•	•	•	•	6	•	•	•	•
RTS	Return from Subroutine	•	•	•	•	•	•	•	•	6	•	•	•	•
SBC	Subtract Memory from Accumulator with Borrow	•	2	3	4	•	4	4*	4*	•	•	6	5*	•
SEC	Set Carry Flag	•	•	•	•	•	•	•	•	2	•	•	•	•
SED	Set Decimal Mode	•	•	•	•	•	•	•	•	2	•	•	•	•
SEI	Set Interrupt Disable Status	•	•	•	•	•	•	•	•	2	•	•	•	•
STA	Store Accumulator in Memory	•	•	3	4	•	4	5	5	•	•	6	6	•
STX	Store Index X in Memory	•	•	3	•	4	4	•	•	•	•	•	•	•
STY	Store Index Y in Memory	•	•	3	4	•	4	•	•	•	•	•	•	•
TAX	Transfer Accumulator to Index X	•	•	•	•	•	•	•	•	2	•	•	•	•
TAY	Transfer Accumulator to Index Y	•	•	•	•	•	•	•	•	2	•	•	•	•
TSX	Transfer Stack Pointer to Index X	•	•	•	•	•	•	•	•	2	•	•	•	•
TXA	Transfer Index X to Accumulator	•	•	•	•	•	•	•	•	2	•	•	•	•
TXS	Transfer Index X to Stack Pointer	•	•	•	•	•	•	•	•	2	•	•	•	•
TYA	Transfer Index Y to Accumulator	•	•	•	•	•	•	•	•	2	•	•	•	•

*Add one cycle if indexing across page boundary
**Add one cycle if branch is taken. Add one additional if branching operation crosses page boundary

Beim KIM: 1 clock cycle = 1 μs

MNEMO	Operation / Adreßart	IMMED.	ABSOLUTE	Z.PAGE	ACCUMUL.	IMPLIED	(IND,X)	(IND),Y	Z.PAGE,X	ABS,X	ABS,Y	RELATIVE	INDIRECT	Z.PAGE,Y	FLAGS
ADC	A+M+C → A	69	6D	65			61	71	75	7D	79				NZCV
AND	A ∧ M → A	29	2D	25			21	31	35	3D	39				NZ
ASL	C ← [7 … 0] ← 0		0E	06	0A				16	1E					NZC
BCC	Branch C=0											90			-
BCS	" C=1											B0			-
BEQ	Branch Z=1											F0			-
BIT	A∧M M_7→N M_6→V		2C	24								30			NZV
BMI	Branch N=1											D0			-
BNE	" Z=0											10			-
BPL	" N=0														-
BRK	Interrupt					00									B=I=1
BVC	Branch V=0											50			-
BVS	" V=1											70			-
CLC	0 → C					18									C=0
CLD	0 → D					D8									D=0
CLI	0 → I					58									I=0
CLV	0 → V					B8									V=0
CMP	A-M, C=0 bei A<M	C9	CD	C5			C1	D1	D5	DD	D9				NCZ
CPX	X-M, C=0 bei X<M	E0	EC	E4											NZC
CPY	Y-M, C=0 bei Y<M	C0	CC	C4											NZC
DEC	M-1 → M		CE	C6					D6	DE					NZ
DEX	X-1 → X					CA									NZ
DEY	Y-1 → Y					88									NZ
EOR	A ⊻ M → A	49	4D	45			41	51	55	5D	59				NZ
INC	M+1 → M		EE	E6					F6	FE					NZ
INX	X+1 → X					E8									NZ
INY	Y+1 → Y					C8									NZ
JMP	Sprung		4C										6C		-
JSR	Unterpr. Sprg.		20												-
LDA	M → A	A9	AD	A5			A1	B1	B5	BD	B9				NZ
LDX	M → X	A2	AE	A6							BE			B6	NZ
LDY	M → Y	A0	AC	A4					B4	BC					NZ
LSR	0 → [7 … 0] → C		4E	46	4A				56	5E					NZC
NOP	No Oper.					EA									-
ORA	A ∨ M → A	09	0D	05			01	11	15	1D	19				NZ
PHA	A → Stack					48									-
PHP	P → Stack					08									-
PLA	Stack → A					68									NZ
PLP	Stack → P					28									alle
ROL	[7 … 0] ← C		2E	26	2A				36	3E					NZC
ROR	C → [7 … 0]		6E	66	6A				76	7E					NZC
RTI	Return Interr.					40									alle
RTS	Return					60									-
SBC	A-M-$\overline{C}$ → A	E9	ED	E5			E1	F1	F5	FD	F9				NZCV
SEC	1 → C					38									C=1
SED	1 → D					F8									D=1
SEI	1 → I					78									I=1
STA	A → M		8D	85			81	91	95	9D	99				-
STX	X → M		8E	86										96	-
STY	Y → M		8C	84					94						-
TAX	A → X					AA									NZ
TAY	A → Y					A8									NZ
TSX	S → X					BA									NZ
TXA	X → A					8A									NZ
TXS	X → S					9A									-
TYA	Y → A					98									NZ

Befehls - Liste des 6502

ASCII-Zeichencode

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Beispiel: 47 ist der ASCII-Code für den Buchstaben G

Bedeutung einiger Sonderzeichen:

20 ≙ SP (Space) Leerzeichen
0D ≙ CR (Carriage Return) Wagenrücklauf
0A ≙ LF (Line Feed) Zeilenvorschub

Bedienung des KIM-Mikrocomputers

Starten: RESET-Taste, RUBOUT-Taste
Anzeigen einer Speicherzelle: Adresse(in Hex), Leertaste
Ändern einer Speicherzelle: Adresse, Leer, neuer Inhalt, •
Inhalt der nächsten Zelle anzeigen: RETURN-Taste
Inhalt der vorangehenden Zelle anzeigen: LINE FEED-Taste

Starten eines Programms: Startadresse, Leer, G
Abbrechen eines Programms: RESET-Taste, RUBOUT-Taste

Einzelschrittausführung von Programmen: In 17FA, 17FB
00, 1C eintragen; dann den SST-Schalter auf ON.

Startadressen von Unterprogrammen des KIM-Monitors:

Ausgabe eines ASCII-Zeichens (ASCII-Zeichen im Akku): 1EA0
Einlesen " " " " " " " : 1E5A

Ausgabe eines Bytes (Akkuinhalt als zwei Hex-Zeichen): 1E3B
Einlesen " " (Byte nach dem Einlesen im Akku): 1F9D

Sprung in das Eingabeprogramm des Monitors: 1C4F

Bemerkung: Zur Vorbereitung der Ein/Ausgabe ist ein Aufruf des Monitor-Unterprogramms bei 1E8C erforderlich. Dieser Aufruf geschieht normalerweise durch den RESET-Vorgang (Starten).

Port und Zeitgeber

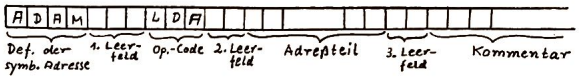
Port A: Datenzelle 1700 , Datenrichtungszelle: 1701
Port B: " 1702 , " : 1703

Timer laden: 1704, 1705, 1706, 1707 (1,8,64,1024-μs Intervall)
" lesen: 1706

Interrupt-Bearbeitung

IRQ- und BRK-Vektor in: 17FE, 17FF
NMI-Vektor in: 17FA, 17FB

Assembler-Anweisungen



Symbolische Adressen (SYMB) bestehen aus beliebig vielen alphanumerischen Zeichen; Leerfelder bestehen aus mindestens 1 Leerzeichen. Kommentarzeilen beginnen mit :

Adreßteil:

Immediate:	#HH, z.B. #0A	(HH 2 Hexzeichen) oder
	'A , z.B. 'Q	(A Zeichen, das in den ASCII-Code umgewandelt wird)
Absolute:	SYMB	(Symbolische Adresse)
Absolute,X:	SYMB,X	
Absolute,Y:	SYMB,Y	
Zero Page:	*SYMB	
Zero Page,X:	*SYMB,X	
Zero Page,Y:	*SYMB,Y	
Indirect:	(SYMB)	
(Indirect,X):	(SYMB,X)	
(Indirect),Y:	(SYMB),Y	
Relative:	SYMB	

Zusatz: 1. Als symbolische Adresse darf auch SYMB+HH (symb. Adresse + 1 oder 2 Hex-Zeichen) bzw. SYMB-HH verwendet werden.
2. Symbolische Adressen dürfen auch durch feste Adressen, die aus 1 bis 4 Hex-Ziffern bestehen, ersetzt werden: $HHHH
3. Bei Adreßart Immediate kann durch #SYMB*L der niederwertige und durch #SYMB*H der höherwertige Teil einer Adresse als Operand verwendet werden.

Außer den Assembler-Anweisungen, die Maschinenbefehle erzeugen, gibt es nachfolgende Anweisungen für den Assembliervorgang:

	.OR HHHH	definiert den Anfang des Maschinenprogr.
SYMB	.DL HHHH	ordnet SYMB die Hexadresse HHHH zu.
	.EN	Ende des Quellprogramms.
	.HS HH...H	jeweils 2 Hex-Zeichen werden in aufeinanderfolgenden Bytes abgespeichert.
	.AS 'AA..A'	Zeichen werden in aufeinanderfolgenden Bytes im ASCII-Code abgespeichert.
	.SA SYMB	Die Hex-Adresse von SYMB wird in den nächsten 2 Bytes abgespeichert.

Bedienung des 6502-Editor/Assemblers

Editor:

1. Start

   Startadresse: 0200

   a) Nach dem Start sind Anfangsadresse, Endadresse RETURN des Speicherbereiches anzugeben, der für das Quellprogramm zur Verfügung steht (Anfangsadresse ≥ 2000).

   b) Bei Neuerstellung des Quelltextes ist die Frage NEW-N ? mit N, sonst mit RETURN zu beantworten.

2. Schreiben einer Zeile

RETURN	beendet Zeile (Zeile wird bis Cursor übernommen)
LINE FEED	beendet Zeile (Gesamt-Zeile wird übernommen)
CTRL H	ein Zeichen zurück
CTRL G	ein Zeichen vorwärts
CTRL I	Tabulator

3. Editorkommandos

   Editorkommandos werden mit RETURN beendet.
   Zeilennummern und n sind maximal 4-stellige Dezimalzahlen.

%Zn	bei Zeile n fortfahren (%Z bei Zeile 0 fortfahren)
%Ln	n Zeilen listen (%L Ausschreiben des gesamten Textes)
%Rn	n Zeilen rückwärts (%R = 1 Zeile rückwärts)
%E	Einfügemodus setzen (bis zum %EE - Kommando werden Zeilen eingefügt)
%EE	Einfügen Ende
%H	Editor - Assembler beenden (halten)
%A	Assemblieren

Bemerkung: a) Löschen einer Zeile geschieht durch RETURN in der ersten Spalte.

b) Durch LINE FEED wird zur nächsten Zeile übergegangen.

Assembler:

1. Assemblieren

   Nach dem Erstellen des Quelltextes wird mit %A der Assembler gestartet.

   a) Auf die Frage: OBJECT CODE ? wird die Anfangsadresse RETURN des Speicherbereichs angegeben, in dem das übersetzte Maschinenprogramm abgespeichert werden soll.

   b) Auf die Frage: OUTPUT - Y ? ist mit Y zu antworten, falls ein Assemblerprotokoll ausgeschrieben werden soll; andernfalls ist RETURN zu geben.

   c) Auf die Frage: CROSS-REFERENCE - Y ? ist mit Y zu antworten, falls eine Symboltafel ausgedruckt werden soll; andernfalls ist mit RETURN zu antworten.

Stichwortverzeichnis