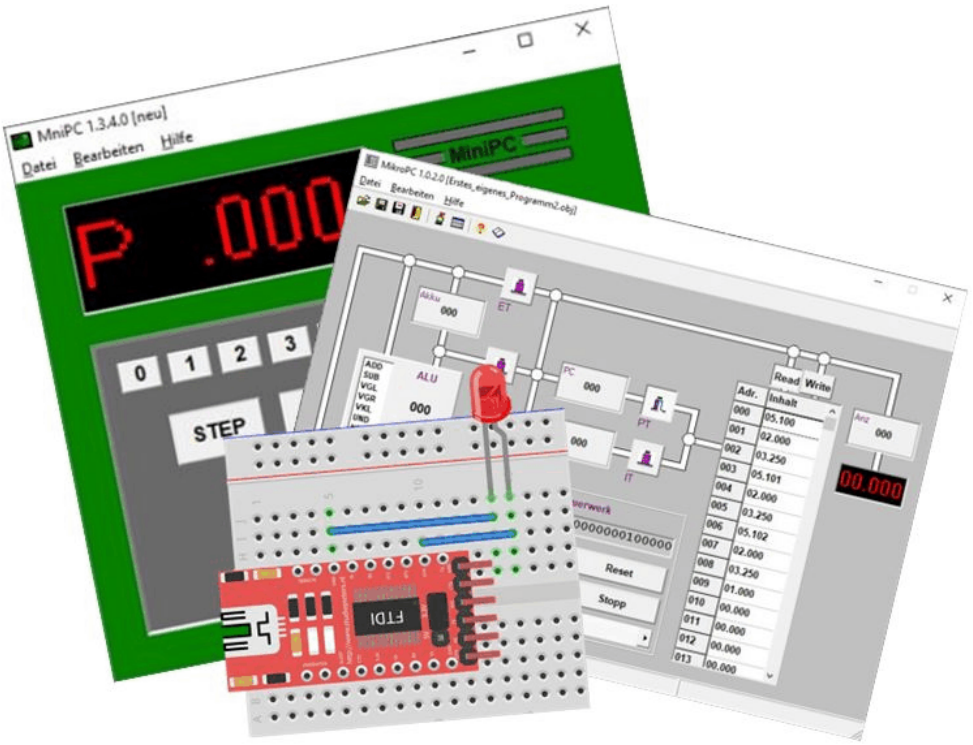


Mini-PC

Wie Computer aufgebaut sind,
wie sie funktionieren und
wie sie programmiert werden

Eine praktische Einführung
in die Grundlagen
mit zahlreichen Übungen

von G. Heinrichs

Überarbeitete und erweiterte Auflage vom 02.09.2021
Jetzt mit Anbindung zu FT232-Boards

Inhaltsverzeichnis

Einleitung

Abb. E.1

Computer - überall sind wir von ihnen umgeben. Wir nutzen sie unablässig, sei es unbewusst, wenn wir auf unsere Digitaluhr schauen oder mit dem Handy ein Gespräch führen, sei es bewusst, wenn wir auf einem Computer einen Text schreiben oder eine Tabellenkalkulation erstellen. Der Computer ist in den letzten Jahrzehnten selbstverständlicher Bestandteil unseres Lebens geworden genauso wie vorher schon das Auto, das elektrische Licht oder der Fernseher. Und wie viele andere vertraute Dinge werden heute auch Computer häufig einfach hingenommen, und nur wenige fragen sich: Wie funktioniert überhaupt ein solcher Computer?

Dass es elektrische Signale sind, welche - durch Stromkreise verarbeitet - den Computer zu seinen Leistungen befähigen, wird auch noch jeder wissen. Aber damit ist noch lange nicht geklärt, wie eine solche Maschine - und nichts anderes ist ein Computer - es schafft, derart unterschiedliche Aufgaben wie das Steuern eines Raumschiffes, die Verwaltung eines Lagerbestandes oder das Spielen einer Partie Schach höchst erfolgreich zu bewältigen.

Das machen die Programme, die geben den Computern erst die Intelligenz, und die sorgen auch dafür, dass Computer so vielseitig einsetzbar sind - das ist die rasche Antwort auf die letzte Frage. Aber auch mit dieser Antwort wird mancher sich nicht zufrieden geben wollen, wundert er sich doch:

– Wie sehen überhaupt solche Programme aus? Wie setzen sie den Computer in die Lage, bestimmte Aufgaben zu erfüllen?
– Wie versteht der Computer solche Programme überhaupt und wie setzt er sie um? Braucht er dazu nicht schon wieder eine gewisse Intelligenz?

Für all diejenigen, die umfassende Antworten auf derartige Fragen suchen, ist dieses Buch gedacht: Wir werden beschreiben und erklären, aus welchen Bestandteilen sich ein Computer zusammensetzt, wie er Signale empfängt, sie weiter verarbeitet und wieder aussendet. Wir werden darlegen, wie der Computer bei der Verarbeitung der Signale durch einzelne Befehle gesteuert wird. Und schließlich werden wir auch zeigen, wie man mit diesen Befehlen ganze Programme schreiben kann.

Dabei erwartet den Leser aber keine trockene theoretische Abhandlung; vielmehr handelt es sich hier um eine praktische Einführung. “Praktisch” ist hier wörtlich zu verstehen; denn zu diesem Buch gibt es einen Mini-PC, an dem man alles Gelernte gleich ausprobieren kann. Dieser Mini-

PC liegt allerdings nicht in Hardware-Form vor[1]. Vielmehr habe ich zwei Programme mit den Namen ***MiniPC*** und ***MikroPC*** geschrieben, welche unseren Mini-PC simulieren. Die Simulation hat gleich mehrere Vorteile: Zunächst einmal ist sie deutlich billiger; man kann sie kostenlos von meiner Homepage herunterladen (s. u.). Zum anderen gestattet sie auch Einblicke in das Innere des Mini-PCs, der bei einem realen Computer nicht so einfach möglich wäre; so können wir uns bei MiniPC z.B. die Speicherinhalte auch bei laufendem Programm anschauen. MikroPC schließlich gestattet es sogar, die einzelnen Signale beim Ablaufen eines Programmes zu verfolgen.

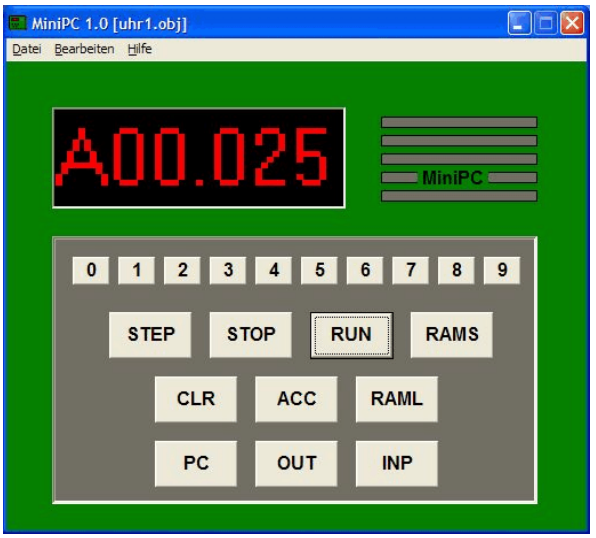


Abb. E.2: Das Simulationsprogramm MiniPC

In den ersten beiden Kapiteln machen wir uns zunächst mit dem Programm MiniPC vertraut. Wie lernen, wie man Daten in den Speicher eingibt und ausliest, wie man Programme eingibt und zum Laufen bringt. Zentraler Begriff wird hier die Speicherzelle sein.

Im Kapitel 3 werfen wir einen ersten Blick in das Innere unseres Mini-PC. Wir lernen den Akku-Register, die ALU, den Programmzähler und den Datenbus kennen und erfahren, dass ein Befehl sich aus zwei Zahlen, dem Operator und dem Operanden, zusammensetzt. Mit diesem Wissen schreiben wir unsere ersten Programme und testen sie aus.

Programmierstrukturen wie bedingte und unbedingte Sprünge, Schleifen und Verzweigungen sind schon etwas komplexer; wie man damit umgeht, davon handelt Kapitel 4. In diesem Kapitel wird auch eine wichtige Komponente unseres Mini-PCs vorgestellt: das Flag-Register.

In Kapitel 5 legen wir nun ausführlich dar, wie unser Mini-PC eigentlich funktioniert. Hier werden viele der am Anfang gestellten Fragen beantwortet. Dabei lernen wir weitere Bestandteile unseres Mini-PCs kennen: weite-

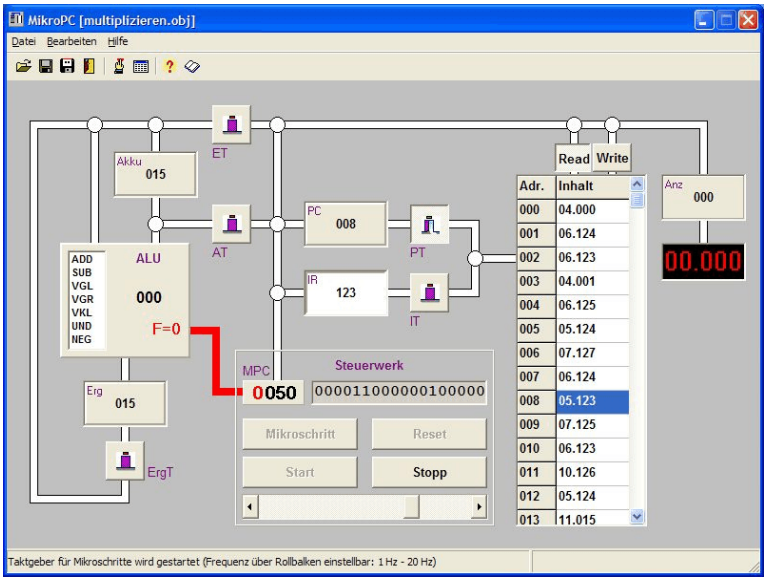


Abb. E.3: Das Simulationsprogramm MikroPC

[1]Unter dem Namen CP1 wurde um 1980 eine solche Hardware von der Firma Kosmos verkauft; an dieses Lernsystem lehnt sich unser Simulationsprogramm MiniPC an.

re Register neben dem Akku, ferner so genannte Tore zu Steuerung des Datenflusses und vor allem das Steuerwerk. Wie dieses Steuerwerk mit Hilfe des so genannten von-Neumann-Zyklus Befehl für Befehl abarbeiten lässt, schauen wir uns ganz genau an. Dazu benutzen wir das Simulationsprogramm MikroPC (vgl. Abb. E.3). MikroPC verdeutlicht, dass jeder Befehl, den unser Mini-PC kennt, sich aus einfachen, elementaren Mikroschritten zusammensetzt. Mit MikroPC kann man diese Mikroschritte nachvollziehen; ja man kann sogar selbst neue Befehle aus solchen Mikroschritten konstruieren.

Programme bestehen, wie oben bereits erwähnt, letztlich nur aus Zahlen. Das ist für den Programmierer häufig sehr unübersichtlich. Deswegen benutzen Programmierer meist sogenannte Assembler-Programme. Diese erleichtern das Programmieren wesentlich. Zu MiniPC gibt es auch einen Assembler. Wie man damit umgeht, das zeigt das Kapitel 6.

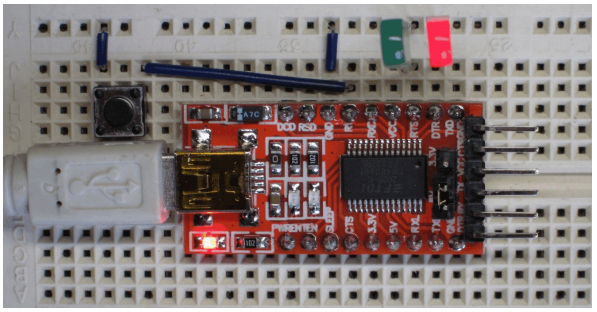


Abb. E.4: Breadboard mit FT232-Modul

Computer müssen Informationen aus der Umgebung aufnehmen und Informationen an die Umgebung zurückgeben; sonst sind sie wertlos. Zur Eingabe besitzt unser Mini-PC eine Tastatur, zur Ausgabe ein Display. MiniPC kann aber auch Kontakt mit der Außenwelt aufnehmen. Dazu verbindet man ein Experimentierboard wie in Abb. E.4 mit einem USB-Anschluss des Rechners. Wer noch einen COM-Anschluss an seinem Rechner besitzt, kann auch diesen benutzen. Mehr über die verschiedenen Anschlussmöglichkeiten findet man im Anhang. In den Kapitel 7 und 8 zeigen wir nun, wie man damit kleinere elektronische “Anlagen” steuert und kontrolliert: Blinklichter, Ampelanlagen, Reaktionstester, Dämmerungsschalter oder auch Soundgeneratoren können damit realisiert werden.

Das abschließende Kapitel 9 beschreibt fortgeschrittene Programmiertechniken. Hier geht es um indirekte Adressierung, Unterprogramme und Interrupts.

Alle Kapitel weisen eine Vielzahl von Aufgaben auf. Es ist nicht nur lehrreich, diese Aufgaben zu lösen - es macht auch Spaß! Und wenn man einmal nicht mehr weiter weiß, dann wirft man eben einen Blick auf die ausführlich kommentierte Lösungsdatei.

Jetzt bleibt mir nur noch übrig, dem Leser viel Freude beim Lesen der folgenden Seiten und Erfolg beim eigenen Programmieren zu wünschen.

Georg Heinrichs

Alle benötigten Dateien findet man zum Download auf meiner Homepage über:

http://www.g-heinrichs.de/wordpress/index.php/informatik/minipc/

Hier findet man:

– Software: Die Programme minipc.exe und mikropc.exe (mitsamt den zugehörigen Mikrocodetabellen mikrocode.dat und mikrocode0.dat) sowie eine leere Mikrocode-Tabelle im pdf-Format für eigene Kreationen
– Beispiele: Die Dateien für die im Buch behandelten Beispiele (OBJ und ASM)
– Skript: Dieses Buch im pdf-Format

1. Der Speicher

Eines der wichtigsten Bauteile eines Computers ist der **Speicher**. In diesem werden solche Informationen abgelegt, auf die man später noch einmal zurückgreifen möchte: möglicherweise Telefonnummern, Zwischenergebnisse von Rechnungen usw. Einen solchen Speicher kann man vergleichen mit einem großen Aktenschrank; dieser besitzt zahlreiche Schubladen, in welche man Daten ablegen kann (Abb. 1.1). Die Schubladen sind sorgfältig von außen mit Etiketten beschriftet, damit man ohne mühsames Suchen die Daten wiederfinden kann. Im Falle unseres Modellcomputers hat der Aktenschrank 128 Schubladen; diese werden als **Speicherzellen** bezeichnet. Die Speicherzellen sind durchnummeriert, und zwar von 0 bis 127. Diese Nummern kann man mit den Etiketten unserer Schubladen vergleichen. Sie helfen bei der Suche nach bestimmten Informationen genauso wie die Adresse bei der Suche nach einem bestimmten Haus. Deswegen bezeichnet man diese Nummern als **Zelladressen** oder auch kurz Adressen.

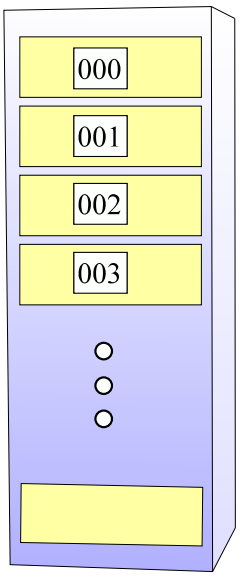


Abb. 1.1

In den einzelnen Zellen unseres Computers kann man nur ganze Zahlen ablegen. Dies gilt übrigens auch für alle Computer: Will man Texte, Dezimalzahlen oder sogar Zeichnungen und Musikstücke speichern, so muss man sie zuerst mit Hilfe von ganzen Zahlen darstellen. Da dies der Computer meist von sich aus erledigt, merken wir dies nur selten.

Der Speicher unseres Computers wird als **RAM** bezeichnet. RAM ist eine Abkürzung für *Random Access Memory*. Übersetzt heißt das: Speicher mit beliebigem Zugriff. Das bedeutet, dass wir über die Adresse direkt auf den Inhalt einer speziellen Zelle zugreifen können, ohne dass wir erst andere Zellen bemühen müssen. Bei einem langen Papierstreifen mit Morsezeichen oder einem Magnetband wäre dieser direkte Zugriff nicht möglich; da hier Markierungen wie Adressen oder Etiketten fehlen, müsste man auf der Suche nach einer bestimmten Information den Streifen oder das Band zu Beginn starten und dann die vorgefundenen Zeichen abzählen.

Wenn man Informationen in eine Speicherzelle ablegt, spricht man auch von einem **Schreibvorgang**. Die Vorstellung dabei: Wir schreiben unsere Zahl auf einen Zettel und legen diesen in eine Schublade unseres Aktenschranks. Holen wir Informationen aus einer Zelle, sprechen wir von einem **Lesevorgang**: Die entsprechende Schublade wird geöffnet, der Zettel in der Schublade wird gelesen; dabei wird der Zettel aber nicht aus der Schublade genommen. Beim Lesevorgang wird die Information in einer Zelle also nicht gelöscht. **Löschen** kann man eine Zelle – genauer: deren Inhalt – nur, indem man eine neue Zahl in diese Zelle schreibt. Man sagt dann: der alte Wert wird **überschrieben**.

Wie kann man bei unserem Modellcomputer MiniPC nun Werte aus dem RAM lesen? Dazu benutzt man die OUT-Taste. **OUT** steht hier für auslesen. Natürlich müssen wir unserem Computer auch mitteilen, aus welcher Zelle wir den Inhalt anzeigen lassen wollen; wir geben also zuerst die Zelladresse ein und betätigen dann die OUT-Taste; dabei müssen führende Nullen auch angegeben werden. Die Adresse 17 wird demnach als 017 eingegeben:

Versuchen wir es:

0 1 7 OUT

In der Anzeige erscheint:

C 0 0 . 0 0 0

Der Buchstabe C weist auf das englische Wort *cell* für Zelle hin. Auf die Bedeutung des Punktes kommen wir später noch zu sprechen (Abschnitt 3.2). Zu Beginn befindet sich in der Speicherzelle 017 offensichtlich die Zahl 0. Das gilt übrigens auch für alle anderen Speicherzellen unseres RAMs.

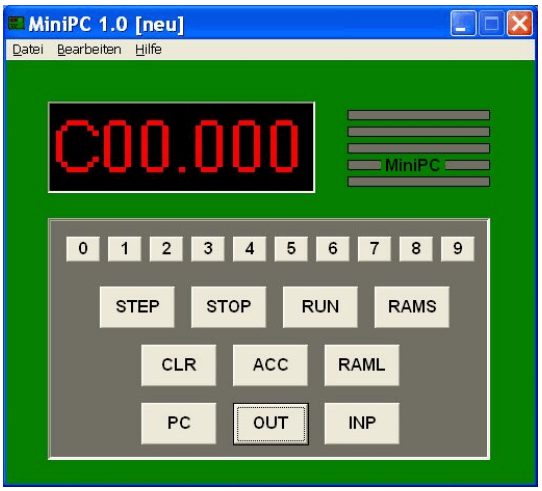


Abb. 1.2

Zelladressen müssen immer genau 3 Stellen besitzen; gibt man zu viele oder zu wenige Stellen an, erfolgt eine Fehlermeldung (F .001).

Jetzt wollen wir eine Zahl in die Speicherzelle 017 schreiben, z. B. die Zahl 11. Dazu benutzen wir die Input-Taste **INP**:

0 0 0 1 1 INP

Auch hier müssen wieder führende Nullen mit angegeben werden; jeder Zellinhalt besitzt genau 5 Stellen. Die Eingabe wird durch die Anzeige E 00.011 bestätigt.

Woher weiß jetzt aber der Computer, in welche Zelle er die Zahl 11 schreiben soll? Ganz einfach: er greift auf die Zelle zurück, welche als letzte benutzt wurde. (Es gibt eine Ausnahme von der Regel; auf diese kommen wir gleich zu sprechen.) In unserem Fall war das die Zelle mit der Adresse 017. Der Computer merkt sich diese Zelladresse in einem so genannten **Ein-Ausgabe-Zeiger**; diesen können wir uns als eine zusätzliche Schublade vorstellen, die der Computer für seine eigenen Aufgaben braucht. Dass die Zahl 11 tatsächlich unter der Adresse 017 abgespeichert worden ist, kann man leicht mit der Out-Taste nachprüfen:

0 1 7 OUT

In der Anzeige erscheint tatsächlich

C 0 0 . 0 1 1

Häufig kommt der Fall vor, dass man mehrere Zellen hintereinander beschreiben möchte; in diesem Fall kann man sich zunutze machen, dass der Ein-Ausgabe-Zeiger nach jedem Drücken der INP-Taste automatisch um 1 erhöht wird. An einem Beispiel soll das verdeutlicht werden. Wir stellen mit der OUT-Taste den Ein-Ausgabe-Zeiger auf 000.

0 0 0 OUT

Nun geben wir nacheinander ein:

0 0 0 1 1 INP

0 0 0 2 2 INP

0 0 0 3 3 INP

0 0 0 4 4 INP

Dann werden diese Zahlen in den Zellen 000, 001, 002 und 003 abgespeichert. Dies könnte man mit Hilfe der OUT-Taste nachprüfen.

Es gibt aber auch eine andere Methode, die Inhalte der Zellen zu überprüfen; diese funktioniert nur bei unserem Simulationsprogramm, nicht hingegen bei einem realen Computer: Wir können beim Simulationsprogramm nämlich direkt in das RAM hineinschauen. Dazu gehen wir in das Bearbeiten-Menü und wählen dort die Option RAM anschauen. Sofort wird im rechten Teil das RAM mit seinen Adressen und Inhalten dargestellt (Abb. 1.3). Tatsächlich kann man die Inhalte hier nicht nur anschauen, sondern auch editieren. Dazu muss man nur die gewünschte Zelle doppelt anklicken.

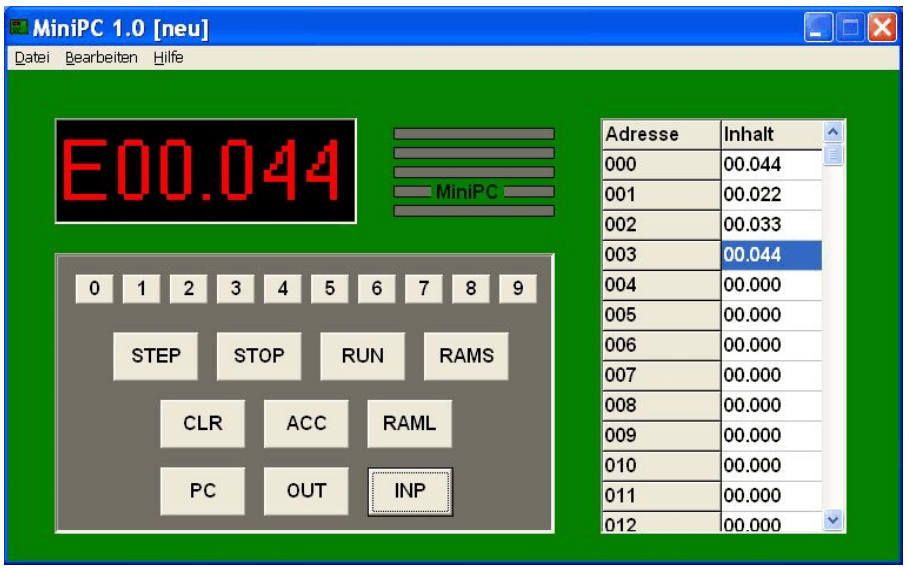


Abb. 1.3

Beim Schließen des Simulationsprogramms gehen - wie auch beim Ausschalten eines realen Computers - die im RAM gespeicherten Informationen verloren. Beim Simulationsprogramm können wir den RAM-Inhalt dauerhaft auf einem Datenträger, z. B. der Festplatte speichern. Dazu wählt man im Datei-Menü die Option Speichern unter... Mit der Option Öffnen können die Informationen dann wieder vom Datenträger in das RAM geladen werden.

Kein Mensch ist fehlerfrei, und so wird es immer wieder vorkommen, dass bei der Eingabe Fehler auftauchen. Dann betätigt man die **CLR**-Taste (*Clear* = Löschen). Die Anzeige wird dadurch gelöscht. Wurde nun die CLR-Taste betätigt, bevor die INP-Taste gedrückt wurde, kann man den Zellinhalt einfach von Neuem eingeben und die Eingabe mit der INP-Taste abschließen. Bemerkt man hingegen seinen Fehler erst nach Betätigen der INP-Taste, muss man die Adresse mit der OUT-Taste neu einstellen; dies ist nötig, weil der Ein-Ausgabe-Zeiger durch die INP-Taste ja schon automatisch um 1 erhöht worden ist.

2. Programme eingeben

Böse Zungen behaupten, Lehrer legten die Noten manchmal mit einem Würfel fest. Das kann zumindest für die Noten der Oberstufe nicht gelten. Hier gibt es nämlich 16 Noten - von 0 bis 15 Punkten; und einen Würfel mit 16 Flächen dürften nur wenige Lehrer haben. Allerdings kann man Zufallszahlen auch anders erzeugen, z. B. mit unserem Modellcomputer: Wir starten das Programm MiniPC und tätigen folgende Eingaben:

```
0 0 1       OUT

0 4 0 0 0   INP
0 6 0 0 0   INP
0 4 0 0 1   INP
0 7 0 0 0   INP
0 3 0 5 0   INP
0 6 0 0 0   INP
0 4 0 1 5   INP
1 2 0 0 0   INP
1 1 0 0 3   INP
0 9 0 0 1   INP
```

Adresse	Inhalt
000	00.001
001	04.000
002	06.000
003	04.001
004	07.000
005	03.050
006	06.000
007	04.015
008	12.000
009	11.003
010	09.001
011	00.000
012	00.000

Abb. 2.1

Über die Option *RAM anzeigen* des *Bearbeiten*-Menüs kann die Eingabe kontrolliert werden: Die Zellen 001 bis 010 müssen die Inhalte aus Abb. 2.1 haben. Am besten speichert man das RAM jetzt erst einmal unter dem Namen “noten_ wuerfeln.obj” ab.

Nun blenden wir die RAM-Anzeige wieder aus. Mit der Eingabe

```
0 0 1   PC
```

teilen wir nun dem Computer mit, dass unser Programm in der Zelle 001 beginnt; er quittiert diese Eingabe mit P .001. (Warum hier nicht die Zelle 000 als erste Zelle des Programms benutzt wurde, werden wir gleich noch sehen!) Der Computer merkt sich diesen Startwert in einer eigenen Speicherzelle; diese Zelle wird **Programmzähler** (englisch: *p*rogram *c*ounter, abgekürzt: **PC**) genannt.

Wenn wir jetzt die Taste RUN betätigen, wird das Programm gestartet und unser Modellcomputer fängt an zu arbeiten. Erst wenn wir die Stopp-Taste **STOP** betätigen, hält das Programm an und wir sehen das Ergebnis dieser Arbeit: Der Computer hat eine Zahl zwischen 0 und 15 “gewürfelt” und zeigt sie an (s. Abb 2.2). Jedesmal wenn wir die Tastenfolge

RUN STOP

betätigen, erhalten wir eine neue Note.

Wie geht nun MiniPC beim Ermitteln der Noten vor? Um dies zu verstehen, blenden wir wieder mit der Option *RAM anzeigen* im *Bearbeiten*-Menü das RAM ein. Starten wir anschließend das Programm, sehen wir, dass der Wert in der Zelle 000 sich fortwährend ändert; bei genauerer Betrachtung stellt man fest, dass die Werte rasch hochgezählt werden, bis die Zahl 15 erreicht ist, dann fängt die Zählung wieder bei 0 an. Dies wird unablässig wiederholt, bis die STOP-Taste betätigt wird. In diesem Fall hört das Zählen auf und der aktuelle Wert der Zelle 000 wird im Display angezeigt. Betätigt man erneut die RUN-Taste, wird der Zählvorgang weitergeführt.

Abb. 2.2

Wir sehen: In Wirklichkeit werden hier vom Modellcomputer gar keine Zufallszahlen gebildet. Es hat nur den Anschein, als ob die auf dem Display ausgegebenen Zahlen zufällig seien. Das liegt aber nur daran, dass die Zeiten zwischen dem Starten und dem Stoppen des Programms unterschiedlich lang sind; somit wird unterschiedlich lange gezählt und wir erhalten immer wieder andere Noten.

Übrigens: Damit man das Zählen im RAM überhaupt sehen kann, wurde der Vorgang mit einem Verzögerungsbefehl künstlich verlängert. Dadurch vergehen jeweils 50 Millisekunden, bis die in der Zelle 000 angezeigte Zahl verändert wird. Wenn man also die RUN- und die STOP-Taste sehr rasch hintereinander betätigt, wird nur wenig weitergezählt; dadurch kann man das Notenprogramm natürlich überlisten; solange man nur schnell genug die Tasten bedient, ist die nächste Note kaum anders als die vorherige. Das gelingt allerdings nicht mehr so einfach, wenn diese Verzögerung verringert oder sogar ganz beseitigt wird. Ein Blick auf den RAM-Inhalt in Abb. 2.1 lässt ahnen, welcher Zellinhalt abgeändert werden muss, damit eine Manipulation der zufälligen Noten kaum noch möglich ist.

Wenn man die STOP-Taste betätigt, hält das Programm nicht nur einfach an; es geschieht noch mehr: In diesem Fall zeigt MiniPC nämlich zusätzlich auch den Inhalt der Zelle 000 im Display an. Mit diesem Trick können auf einfache Weise Zwischenergebnisse sichtbar gemacht werden. Dies ist auch der Grund dafür, dass die Zelle 000 (und nicht irgendeine andere) für den Zählwert reserviert wurde und das eigentliche Programm erst in der Zelle 001 beginnt.

Aufgabe:

Versuche das Programm so abzuändern, dass Zufallszahlen zwischen 0 und 10 angezeigt werden.

3. Programme entwickeln

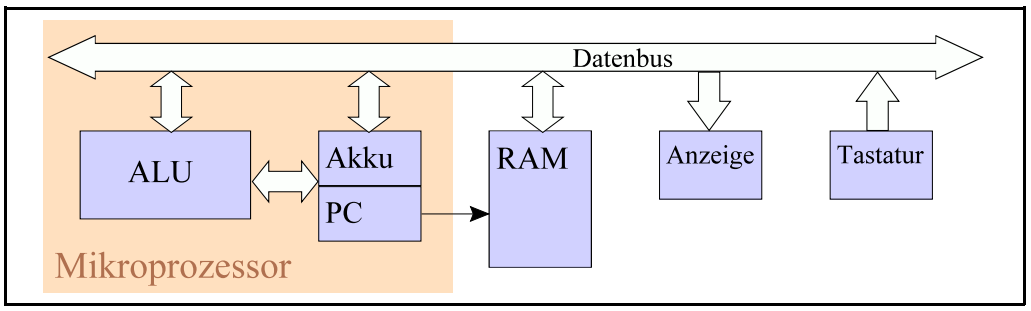


Abb. 3.1

3.1 Aufbau eines Computers

Um selbst Programme entwickeln zu können, muss man den grundsätzlichen Aufbau unseres Modellcomputers kennen. In Abb. 3.1 ist er schematisch dargestellt. Einen Teil der dort gezeigten Komponenten kennen wir bereits: das RAM, die Anzeigeeinheit, die Tastatur und den Programmzähler (PC). Nei sind hier der Akku, die ALU und der Datenbus. Der Ein-Ausgabe-Zeiger fehlt in dieser Darstellung, weil er nur für die Ein- und Ausgabe von Programmteilen, nicht aber für das Ablaufen der Programme erforderlich ist.

Beim **Akku** (Abkürzung für Akkumulator) handelt es sich um einen speziellen Speicher, in dem Zwischenergebnisse abgelegt werden können. **ALU** ist eine Abkürzung für Arithmetic Logical Unit, auf deutsch: arithmetisch-logische Einheit. Die ALU ist also der Baustein, welcher im Computer die eigentliche Rechenarbeit verrichtet. Daneben kann er aber auch logische Aufgaben ausführen, z. B. überprüfen, ob eine Zahl größer ist als eine andere.

All diese Komponenten sind durch den **Datenbus** miteinander verbunden. Über diesen Leiterstrang können Daten ausgetauscht werden. Die Pfeile in Abb. 3.1 deuten an, in welche Richtung die Daten strömen können. So können die Daten vom Datenbus in das RAM und von dort wieder auf den Datenbus gelangen (Doppelpfeil!); bei der Tastatur und der Anzeige ist dagegen nur eine einzige Datenstromrichtung sinnvoll.

Der Programmzähler ist auch direkt mit dem RAM verbunden; sein Inhalt legt fest, mit welcher Zelle des RAMs der Datenbus seine Daten austauscht.

ALU, Akku und PC werden üblicherweise in einen einzigen Baustein integriert, dem **Mikroprozessor**. In diesem Mikroprozessor befinden sich noch weitere Komponenten; da sie für ein erstes Verständnis nicht erforderlich sind, werden wir sie erst später behandeln.

3.2 Programm und Befehl, Operator und Operand

Computer sind dumm. Damit sie eine Arbeit erledigen können, muss man ihnen Schritt für Schritt erklären, was sie zu tun haben. Diese Einzelanweisungen werden auch **Befehle** genannt. Ein **Programm** besteht dann aus einer Abfolge von solchen Befehlen. Derartige Programme gibt es auch bei anderen Maschinen. Waschmaschinen besitzen z. B. verschiedene Waschprogramme. Ein Waschprogramm setzte sich aus verschiedenen Anweisungen für die Maschine zusammen (Abb. 3.2). Wichtig ist hier auch, dass die Anweisungen in einer bestimmten Reihenfolge durchgeführt werden. So wäre es nicht sinnvoll, zuerst die Wasserheizung laufen zu lassen und anschließend erst das Wasser zuzuführen.

1. Wasserzulauf
2. Wasser aufheizen
3. Waschtrommel drehen
4. Waschmittel zuführen
5. Waschtrommel drehen.
6. Wasserablauf
7. Wasserzulauf

Abb. 3.2

Bei unserem Modellcomputer stehen die Befehle jeweils in einer Zelle des RAMs. Und weil der Modell-Computer nicht mit Texten umgehen kann, gibt man die Anweisungen mit Hilfe eines Befehlscode. Ein solcher Befehlscode besteht beim MiniPC immer aus zwei Zahlen, welche durch einen Punkt getrennt sind, z. B.

0 5 . 0 1 1

Welche Bedeutung haben diese beiden Teile? Die Zahl links vom Punkt kennzeichnet, was der Mini-PC machen soll. Man bezeichnet diesen Teil als **Operator**. In unserem Beispiel ist der Wert 5; in diesem Fall soll ein Wert aus dem RAM in den Akku geladen werden. Der Operator 05 kennzeichnet den so genannten Ladebefehl.

Die Zahl rechts vom Komma gibt an, womit der Operator (genauer: der Befehl, der durch den Operator gekennzeichnet ist) arbeiten soll. Diesen Teil bezeichnet man als Operanden. In unserem Beispiel gibt die Zahl 11 die Adresse derjenigen RAM-Zelle an, aus welcher der Wert in den Akku geladen werden soll.

Hat der Operator den Wert 0, dann soll nichts getan werden. Der Zahlencode

0 0 . 1 4 7

stellt damit lediglich eine Zahl dar, nämlich 147. Man spricht bei einem solchen Code auch von einem (reinen) Datum. (Datum ist übrigens der Singular von Daten!) Man beachte: Auch hier darf man bei der Eingabe die Nullen nicht vergessen.

Es gibt auch Befehle, welche keinen Operanden benötigen. Ein solcher Befehl ist der Anzeige-Befehl. Dieser bringt den Inhalt des Akkus zur Anzeige. Der Code für diesen Operator ist 01. Der vollständige Befehlscode lautet dann

0 2 . 0 0 0

Übrigens: Ob man für den Operanden nun die Zahl 0 eingibt oder eine andere, spielt keine Rolle. Weglassen darf man den Operanden bei der Eingabe aber nicht; ansonsten kommt es zu einer Fehlermeldung.

Werfen wir noch einmal einen Blick auf den Befehl mit dem Code **0 5 . 0 1 1**. Gehen davon aus, dass in der Zelle mit der Adresse 11 der Befehlscode **0 0 . 1 4 7** (also die Zahl 147) gespeichert ist, dann wird durch **0 5 . 0 1 1** die Zahl 147 in den Akku geladen.

3.3 Mnemonics

In der Tabelle 1 sind einige wichtige Befehle mit ihren Codes wiedergegeben. In einer zusätzlichen Spalte stehen die so genannten **Mnemonics**. Dabei handelt es sich um Kürzel für die einzelnen Befehle; diese können sich Programmierer leichter merken als die Zahlencodes. Deswegen schreiben sie die Programme zunächst mit Hilfe dieser Mnemonics und setzen sie erst danach in die entsprechenden Codes um. Diese Vorgehensweise wollen wir nun auch bei unserer ersten Programmentwicklung benutzen.

Code	**Bedeutung**	**Mnemonic**
01.000	Programmablauf anhalten	HLT
02.000	Akku-Inhalt anzeigen	ANZ
03.xxx	Verzögern um xxx Millisekunden	VZG
05.xxx	Inhalt von Zelle xxx in Akku laden	LDA

Tabelle 1

Ziel ist es, ein Programm zu schreiben, welches drei Zahlen nacheinander im Display anzeigt. Die drei Zahlen sollen in den Zellen mit den Adressen 100, 101 und 102 liegen. Da es keinen Befehl gibt, der Werte direkt aus einer Zelle des RAMs zur Anzeige bringt, müssen wir den Umweg über den Akku benutzen.

Üblicherweise notiert man die Folge von Befehlen eines Programms in einer Tabelle: Neben die Zelladressen in der ersten Spalte schreibt man in die zweite Spalte die Mnemonics, in die dritte die zugehörigen Zahlencodes und in die letzte gegebenenfalls einen Kommentar.

Adresse	Mnemonic	Code	Kommentar
000	LDA 100	05.100	Lade Inhalt von Zelle 100 in den Akku
001	ANZ	02.000	Zeige Akku-Inhalt im Display an
002	LDA 101	05.101	Lade Inhalt von Zelle 101 in den Akku
003	ANZ	02.000	Zeige Akku-Inhalt im Display an
004	LDA 102	05.102	Lade Inhalt von Zelle 102 in den Akku
005	ANZ	02.000	Zeige Akku-Inhalt im Display an

Wir geben nun die Werte in unseren Modellcomputer ein (Erstes_Programm0.obj):

```
000    OUT

05100  INP
02000  INP
05101  INP
02000  INP
05102  INP
02000  INP
```

Dann schreiben wir noch die Zahlen 11, 22 und 33 in die Zellen mit den Adressen 100, 101 und 102:

```
100    OUT

00011  INP
00022  INP
00033  INP
```

Schließlich starten das Programm mit

```
000    PC
RUN
```

Das Ergebnis ist enttäuschend: Wir erhalten nur eine Fehlermeldung F.002 (ungültiger Operationscode). Und die anzuzeigenden Zahlen haben wir auch nicht gesehen.

Tatsächlich wurden die Werte im Display angezeigt; allerdings arbeitet unser Modellcomputer so rasch, dass wir dies gar nicht wahrnehmen können. Deswegen bauen wir nun nach jedem ANZ-Befehl einen Verzögerungsbefehl ein:

```
VZG 250
```

Dadurch legt der Modellcomputer eine Pause von 250 ms, also einer Viertelsekunde, ein. Das reicht aus, um die Zahl zu erkennen.

Leider kann man die neuen Befehle nicht einfach einfügen; das Programm muss erneut (fast) vollständig eingegeben werden. Alternativ kann man auf die fertige Datei “Erstes_ Programm1.obj” zurückgreifen.

Starten wir es erneut mit der RUN-Schaltfläche. Jetzt kann man kurz die Anzeigen

```
A00.011
A00.022
A00.033
```

aufflackern sehen. Allerdings erhalten wir am Ende immer noch dieselbe Fehlermeldung wie beim ersten Programm.

Offensichtlich ist unser Modellcomputer auf einen Befehlscode gestoßen, den er nicht kennt. Wenn der Computer von einem Befehl zum nächsten übergeht, stößt er schließlich auf die Zelle mit der Adresse 009. In dieser Zelle steht 00.000. Der Computer versucht den Befehl mit dem Operatorcode 00 auszuführen. Einen solchen Befehl gibt es aber nicht; deswegen erfolgt eine Fehlermeldung und das Programm wird abgebrochen.

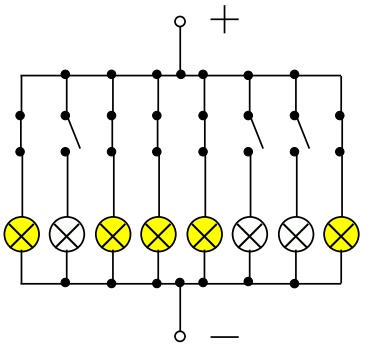
Um diese Fehlermeldung zu vermeiden, muss man dem Computer klar machen, dass er mit seiner Arbeit aufhören soll, wenn er den Befehl in der Zelle 008 bearbeitet hat. Dazu schreiben wir in die folgende Zelle (009) den Halte-Befehl (HLT). Dieser Befehl hält das Programm an, d. h. der Modellcomputer geht nicht mehr automatisch zum nächsten Befehl (in Zelle 010) weiter. Das fertige Programm findet man auch in der Datei “Erstes_Programm2.obj”.

Wenn wir nun dieses Programm laufen lassen, werden die drei Zahlen kurz hintereinander angezeigt; das Programm bleibt stehen und zeigt im Display mit P.010 die Befehlsadresse an, die auf die zuletzt ausgeführte folgt.

3.4 Bit und Byte

Unser Modellrechner kann nur Daten verarbeiten, die zwischen 0 und 255 liegen. Versuchen wir z. B. größere Operanden einzugeben, erhalten wir eine Fehlermeldung (F.004). Diese Beschränkung hat etwas damit zu tun, wie die Zahlen abgespeichert und transportiert werden.

Natürlich wissen wir, dass dies auf elektrischem Wege geschieht. Verschieden große Werte könnten etwa durch unterschiedlich starke Ströme dargestellt werden. Diese Vorgehensweise bezeichnet man als **analog**. In Computern benutzt man aber die **digitale** Methode: Hier gibt es nur die beiden Möglichkeiten “Strom ein” und “Strom aus”. Um verschiedene Zahlen zu übertragen, reicht jetzt ein Stromkreis nicht mehr aus.



Zahl	Code
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000
9	00001001
10	00001010

Abb. 3.3

In Abb. 3.3 sind 8 Lämpchen über 8 Schalter mit einer elektrischen Quelle verbunden. Je nach Schalterstellung leuchtet eine andere Kombination von Lämpchen auf. Jede dieser Kombinationen steht für eine bestimmte Zahl. In der Tabelle von Abb. 3.3 sind die Kombinationen für die Zahlen von 0 bis 10 jeweils durch Nullen und Einsen dargestellt.

Insgesamt gibt es $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$ verschiedene Kombinationen; mit 8 Lämpchen können wir also die 256 Zahlen von 0 bis 255 darstellen. Die Zuordnung des Lämpchencodes zu den Zahlen ist nicht willkürlich; vielmehr kann man aus dem Lämpchencode die zugeordnete Zahl folgendermaßen ausrechnen:

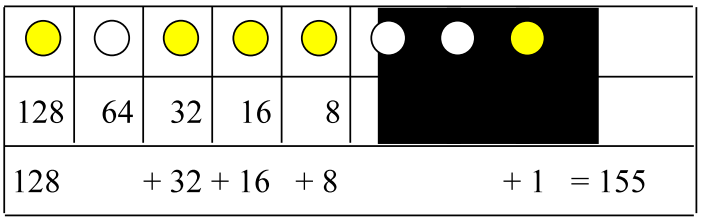


Abb. 3.4

Die einzelnen Lämpchen stehen also für bestimmte Werte, und zwar 1, 2, 4, 8, 16, 32, 64 und 128; das sind gerade die Potenzen von Zwei: $1 = 2^0$, $2 = 2^1$, $4 = 2^2$, $8 = 2^3$ usw. (Mathematiker wissen, dass es sich hier um die ersten 8 Stellenwerte im Zweiersystem handelt.) Um zu einer Lämpchenkombination die zugehörige Zahl zu bestimmen, addieren wir also einfach nur diejenigen Zweierpotenzen, deren Lämpchen leuchten.

Die Zahlen von 0 bis 255 können wir also mit 8 Informationseinheiten (Licht an oder aus) darstellen. Eine solche kleinste Informationseinheit wird als ein **Bit** (von *bi*nary digi*t*) bezeichnet. Eine Kombination von 8 Bits nennt man ein **Byte**. Unser Modellcomputer arbeitet also mit 1-Byte-Zahlen.

Auch das Speichern einer Zahl geschieht Bit für Bit. Die kleinste Speichereinheit kann man sich z. B. als einen Kondensator vorstellen. Ein geladener Kondensator entspricht einem leuchtenden Lämpchen, ein ungeladener Kondensator einem nicht leuchtenden Lämpchen. Sowohl für den Operator als auch für den Operanden werden jeweils 1 Byte benötigt. Jede RAM-Speicherzelle hat also insgesamt 2 Byte, ebenso wie der Akku. Auch der Datenbus muss 2 Byte übertragen können, damit der Inhalt einer Speicherzelle vollständig (d. h. mit Operator *und* Operand) in den Akku übertragen werden kann. Er muss also aus $2 \cdot 8 = 16$ Leitungen bestehen.

Wir sehen: Der Computer arbeitet eigentlich auf der Basis eines Zweiersystems. Damit uns der Umgang mit dem Modellcomputer leichter fällt, werden hier aber die Werte in dem uns vertrauten Zehnersystem eingegeben und auch ausgegeben.

3.5 Einzelschritte

Um Programme auf unserem Modellrechner zu testen, ist es manchmal sinnvoll, nur einzelne Befehle des Programms ausführen zu lassen. Dazu benutzt man die **STEP**-Schaltfläche. Am Beispiel des Programms "Erstes_Programm0.obj" wollen wir dies ausprobieren. Nach dem Öffnen dieser Datei stellen wir den Programmzähler auf 000. Anschließend führen wir den ersten Schritt aus, indem wir einmal die STEP-Taste betätigen. (In der Anzeige erscheint dann **P.001**; hierdurch wird die Speicherzelle des *nächsten* Befehls angezeigt.) Jetzt müsste der Inhalt der Zelle 100 bereits in den Akku geladen sein. Dies können wir nachprüfen, indem wir die **ACC**-Taste betätigen. Durch diese Taste wird der Inhalt des Akkus zur Anzeige gebracht:

A 0 0 . 0 1 1

Beim nächsten Betätigen der STEP-Taste wird der Anzeige-Befehl durchgeführt, beim nächsten der Verzögerungsbefehl. Wenn wir noch einmal die STEP-Taste drücken, wird die zweite Zahl (022) in den Akku geladen. Auch das können wir wieder mit der ACC-Taste nachprüfen.

3.6 Rechnen

In diesem Abschnitt soll unser Modellcomputer seinem Namen alle Ehre erweisen: Das Wort Computer stammt nämlich vom englischen Wort "to compute" = "ausrechnen". Deswegen soll er nun seine Rechenkünste unter Beweis stellen. In Tabelle 2 sind alle neuen Befehle zusammengestellt, die wir in diesem Abschnitt brauchen.

Code	Bedeutung	Mnemonic
06.xxx	Den Inhalt des Akkus in der Zelle xxx speichern	ABS
07.xxx	Inhalt der Zelle xxx zum Inhalt des Akkus addieren. Ergebnis im Akku.	ADD
08.xxx	Vom Akku-Inhalt den Inhalt der Zelle xxx subtrahieren. Ergebnis im Akku.	SUB
04.xxx	Konstante xxx in den Akku laden	AKO

Tabelle 2

Unser Rechner soll zunächst die Zahlen in den Zellen 100 und 101 addieren und das Ergebnis in der Zelle 102 ablegen. Ein Blick auf die Tabelle 2 zeigt, dass es keinen Befehl gibt, der direkt Zahlen aus zwei Zellen des RAMs addiert. Ein solcher Befehl müsste auch mehr als einen Operanden haben; wir könnten ihn daher gar nicht in unserem RAM abspeichern. Deswegen verfährt man folgendermaßen:

- ersten Summanden im Akku zwischenspeichern
- Akku mit dem **ADD**-Befehl zu zweiten Summanden *add*ieren; das Ergebnis wird im Akku zwischengespeichert
- Akkuinhalt mit dem **ABS**-Befehl im RAM *abs*peichern

Das Programm sieht dann so aus (addieren0.obj):

Adresse	Mnemonic	Code	Kommentar
000	LDA 100	05.100	Lade Inhalt von Zelle 100 in den Akku
001	ADD 101	07.101	Akku zu Inhalt von Zelle 101 addieren; Ergebnis im Akku
002	ABS 102	06.102	Inhalt des Akkus in Zelle 102 speichern
003	HLT	01.000	Anhalten

Um das Programm zu testen, sollte man zuerst das Programm eingeben, dann die zwei Summanden in die Zellen 100 und 101 schreiben und anschließend das Programm starten. Zuletzt lassen wir uns den Inhalt der Zelle 102 mit der OUT-Schaltfläche anzeigen:

102 OUT

Die Subtraktion geschieht auf ähnliche Weise mit dem **SUB**-Befehl. Zu bedenken ist hier allerdings, dass unser Modellrechner nicht mit negativen Zahlen umgehen kann.

Um Zahlen aus dem RAM in den Akku zu laden, kann man neben dem LDA-Befehl auch den **AKO**-Befehl benutzen. Durch

AKO xxx (Code: 04.xxx)

wird der Operand xxx in den Akku geladen. Weil diese Zahl im RAM nicht losgelöst vom Operator abgeändert werden kann, bezeichnet man sie als eine **Konstante**. (AKO: Lade *Ko*nstante in den *A*kku). Zu beachten ist: Durch den AKO-Befehl wird nur der Operand in den Akku geladen; beim LDA-Befehl werden hingegen Operator *und* Operand der adressierten Zelle in den Akku geladen.

Aufgaben:

1. Ergänze das Programm addieren0.obj so, dass das Ergebnis für genau 2 Sekunden angezeigt wird.

2. Schreibe ein Programm, welches die Zahlen in den Zellen 100 und 101 subtrahiert und das Ergebnis für eine Sekunde im Display anzeigt.

4. Schleifen und Verzweigungen

Der Einsatz eines Computers ist vor allem dann zweckmäßig, wenn es um Routinearbeiten geht, also um *relativ einfache Arbeiten*, die in gleicher oder ähnlicher Weise *immer wieder durchgeführt werden* sollen. Eine solche Routinearbeit ist das Zählen. Dazu muss zum aktuellen Akkuinhalt immer wieder eine 1 addiert werden und der Akkuinhalt angezeigt werden:

Lade 0 in den Akku
Zeige den Akkuinhalt an
Warte 250 Millisekunden
Addiere 1 zum Akku
Zeige den Akkuinhalt an
Warte 250 Millisekunden
Addiere 1 zum Akku
Zeige den Akkuinhalt an
Warte 250 Millisekunden
Addiere 1 zum Akku
...

Es ist klar: Wenn wir so vorgehen, ist das RAM schnell mit Befehlen vollgeschrieben; und unser Modellcomputer wird so noch nicht einmal bis 50 zählen können. Wir brauchen eine neue Idee!

4.1 Schleifen und unbedingte Sprünge

Schauen wir das obige Programm an, sehen wir, dass immer wieder dieselbe Befehlsfolge

1 *Zeige den Akkuinhalt an*
2 *Warte 250 Millisekunden*
3 *Addiere 1 zum Akku*

ausgeführt werden muss. Nach der Addition in Befehl 3 muss das Programm eigentlich nur wieder mit dem Befehl 1 fortfahren, anstatt den folgenden Befehl auszuführen. Dies erreicht man mit dem (*u*nbedingten) *Sp*rungbefehl

SPU xxx (Code: 09)

Damit lautet unser Zählprogramm (zaehlen1.obj):

Adresse	Mnemonic	Code	Kommentar
000	AKO 000	04.000	0 in den Akku laden
001	ANZ	02.000	Akkuinhalt anzeigen
002	VZG 250	03.250	250 Millisekunden warten
003	ADD 100	07.100	Akkuinhalt um 1 erhöhen (s. u.)
004	SPU 001	09.001	Springe zur Zelle 001
...			
100		00.001	Inkrement

Jedesmal wenn der SPU-Befehl ausgeführt wird, wird der Programmzähler PC wieder auf 001 gesetzt. Das Programm fährt dann wieder mit dem ANZ-Befehl in der RAM-Zelle 001 fort; es folgt der nächste Befehl in der Zelle 002 usw., bis wieder ein Sprung zur Zelle 001 erfolgt. Dies geht endlos so weiter; deswegen nennt man eine solche Programmstruktur auch eine **Endlosschleife** (s. Abb. 1).

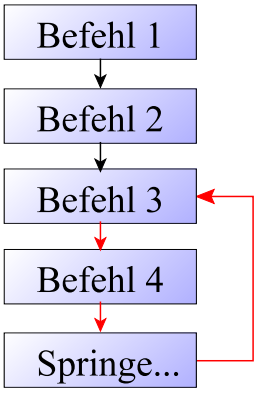


Abb. 4.1: Hier werden die Befehle 3 und 4 fortwährend wiederholt.

Lassen wir das Programm laufen, stellen wir fest, dass das Programm bis 255 zählt und danach mit einer Fehlermeldung F.006 anhält: Unser Modellcomputer kann die anstehende Rechnung 255 + 1 nicht durchführen, weil das Ergebnis 256 so groß ist, dass es nicht mehr im Akku abgelegt werden kann. (Wir erinnern uns: Die größte 1-Byte-Zahl ist 255.)

Allerdings kann man den Programmablauf auch vorzeitig mit der STOP-Schaltfläche unterbrechen. Der Modellcomputer merkt sich dabei den Wert des Programmzählers (genauer gesagt: die Adresse des nächsten Befehls); so kann das Programm nach jedem Stopp problemlos fortgesetzt werden, indem man die RUN-Schaltfläche betätigt.

Wenn die STOP-Schaltfläche betätigt wird, werden gleich mehrere Aktionen ausgelöst:

- Adresse des nächsten Befehls merken,
- Programm anhalten, bis RUN-Taste betätigt wird,
- Inhalt der Zelle 000 anzeigen, wenn hier kein Befehl, sondern eine Zahl abgespeichert ist.

Die letzte Aktion ist eine Besonderheit unseres Modellcomputers; mit ihrer Hilfe kann man auf einfache Weise Zwischenergebnisse anzeigen lassen.

Eine solche Unterbrechung, in der spezielle Befehlsfolgen ausgeführt werden, bezeichnet man als einen **Interrupt** (to interrupt = unterbrechen). Bei einem PC löst zum Beispiel eine Mausbewegung einen Interrupt aus; die zugehörigen Interrupt-Befehle teilen dem Anwenderprogramm die neuen Mauskoordinaten mit und sorgen dafür, dass der Mauszeiger auf dem Bildschirm entsprechend verschoben wird.

In unserem Fall können wir den Interrupt ausnutzen, um den aktuellen Stand des Akkus anzeigen zu lassen. Dazu fügen wir in unserem Programm vor oder hinter dem ANZ-Befehl den Befehl

ABS 000

ein (zaehlen2.obj). Dadurch wird bei jedem Schleifendurchlauf der Inhalt des Akkus in die Zelle 000 geschrieben (Der Befehl AKO 000 wird beim ersten Schleifendurchlauf überschrieben; er wird da aber auch schon nicht mehr benötigt.) Wenn nun beim 45. Schleifendurchlauf die STOP-Schaltfläche betätigt wird, wird das Programm angehalten, und es erscheint die Anzeige

C 0 0 . 0 4 5

4.2 Ein Trick: Endergebnisse dauerhaft anzeigen

Stößt unser Modellrechner auf einen HLT-Befehl, beendet er die Bearbeitung der einzelnen Befehle des Programms und gibt die Kontrolle an das Minibetriebssystem des Rechners ab; nur so ist er in der Lage, Tastaturabfragen für die Eingabe von Programmen und zur Steuerung des Rechners entgegenzunehmen. Dabei wird automatisch die letzte Anzeige gelöscht und durch den Inhalt des Programmzählers ersetzt. Wenn keine zusätzlichen Verzögerungsbefehle eingebaut werden, vergeht dazwischen so wenig Zeit, dass man die gewünschte Anzeige gar nicht mehr wahrnehmen kann.

In diesem Fall kann eine Endlosschleife am Ende des Programms Abhilfe schaffen:

Adresse	Mnemonic	Code	Kommentar
...	...	...	...
057	ANZ	02.000	Akkuinhalt anzeigen
058	SPU 057	09.057	Springe zur Zelle 058
059	HLT	01.000	

Nach der Anzeige des Akkuinhalts springt das Programm wieder zur Zelle 057, der Akkuinhalt wird wieder angezeigt; das ganze geht endlos so weiter, bis man die STOP-Schaltfläche betätigt. Der HLT-Befehl in Zelle 059 wird vom Programm also nie erreicht; er könnte deswegen auch weggelassen werden.

Code	Bedeutung	Mnemonic
10.xxx	Setze Flag auf 1, wenn Akkuinhalt gleich dem Inhalt von xxx ist, sonst setze Flag auf 0.	VGL xxx
13.xxx	Setze Flag auf 1, wenn Akkuinhalt kleiner als der Inhalt von xxx ist, sonst setze Flag auf 0.	VKL xxx
12.xxx	Setze Flag auf 1, wenn Akkuinhalt größer als der Inhalt von xxx ist, sonst setze Flag auf 0.	VGR xxx
11.xxx	Springe zur Zelle xxx, wenn das Flag auf 1 gesetzt ist; fahre ansonsten mit dem nächsten Befehl fort	SPB xxx

Tabelle 1

4.3 Bedingte Sprünge

Häufig möchte man eine Schleife nicht endlos, sondern nur 20 oder 100 mal durchlaufen lassen. Um derartige Schleifen zu programmieren, benötigt man einen so genannten **bedingten Sprungbefehl**:

> Springe zur Adresse xxx, wenn die Bedingung yyy erfüllt ist.

Da unser Modellcomputer aber nur Befehle mit einem einzigen Operanden beherrscht, muss hier schrittweise vorgegangen werden:

Zuerst wird die Bedingung mit einem so genannten Vergleichsbefehl überprüft. Ist die Bedingung erfüllt, erhält ein spezielles Register, das so genannte **Flag-Register**, den Wert 1; ist die Bedingung nicht erfüllt, bekommt es den Wert 0. In Tabelle 1 sind alle Vergleichsbefehle (VGL, VKL und VGR) unseres Modellcomputers angegeben.

Abb. 4.2: Wenn die Bedingung erfüllt ist, schwenkt der Vergleicher die 1-Flagge.

Wenn nun anschließend ein bedingter Sprung ausgeführt wird, richtet sich der Modellcomputer nach dem Wert dieses Flags: Hat es den Wert 1, d. h. ist das Flag gesetzt, wird der Sprung ausgeführt, andernfalls wird zum nächsten Befehl gewechselt.

An einem Beispiel wollen wir dies deutlich machen. Es sollen nacheinander die Zahlen von 0 bis 20 angezeigt werden. Danach soll das Programm anhalten, ohne dass die STOP-Schaltfläche betätigt wird (zaehlen3.obj).

Adresse	Mnemonic	Code	Kommentar
000	AKO 000	04.000	0 in den Akku laden
001	ANZ	02.000	Akkuinhalt anzeigen
002	VZG 250	03.250	250 Millisekunden warten
003	ADD 100	07.100	Akkuinhalt um 1 erhöhen (s. u.)
004	VKL 101	13.101	Setze Flag auf 1, wenn Akkuinhalt kleiner ist als die Zahl in der Zelle 101; sonst setze Flag auf 0
005	SPB 001	11.001	Springe zur Zelle 001, wenn Flag auf 1 gesetzt
006	HLT	01.000	Anhalten
...			
100		00.001	Inkrement
101		00.021	Vergleichszahl

Wenn der Befehl in Zelle 004 zum ersten Mal abgearbeitet wird, hat der Akku den Wert 1. Er ist also kleiner als die Vergleichszahl 21 in Zelle 101. Dementsprechend wird das Flag auf 1 gesetzt. Stößt der Modellcomputer nun auf den folgenden bedingten Sprungbefehl, schaut er zunächst im Flagregister nach; da es den Wert 1 hat, wird zum Befehl 001 gesprungen. Die folgenden Befehle werden ausgeführt; beim nächsten Vergleich ist der Akkuinhalt gleich 2, also immer noch kleiner als die Vergleichszahl. Also wird wieder zum Befehl in der Zelle 001 gesprungen. Dies geschieht so lange, bis der Inhalt des Akkus schließlich den Wert 21 hat. Nun ist der Akkuinhalt nicht mehr kleiner als die Vergleichszahl; deswegen wird vom Vergleichsbefehl das Flag nun auf 0 gesetzt und der anschließende Sprung wird nicht durchgeführt. Vielmehr geht das Programm direkt zum nächsten Befehl über. Das ist in unserem Fall der HLT-Befehl; der Modellcomputer beendet damit seine Arbeit.

Man beachte, dass die Vergleichszahl hier tatsächlich 21 betragen muss, damit die letzte angezeigte Zahl 20 ist. Das muss hier so sein, weil bei unserem Programm der Akkuinhalt erst angezeigt, dann um 1 vergrößert und erst anschließend verglichen wird. Würde der Vergleich schon vor der Vergrößerung durchgeführt, müsste die Vergleichszahl 20 lauten.

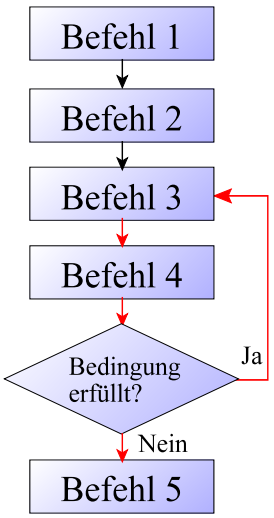


Abb. 4.3

In Abb. 4.3 ist das Schema unseres Programms noch einmal in Form eines Diagramms dargestellt. Diagramme wie in Abb. 4.1 und 4.3 bezeichnet man auch als **Flussdiagramme**. Sie zeigen an, in welcher Reihenfolge die einzelnen Befehle vom Computer abgearbeitet werden (sollen). Normale Anweisungen haben dabei

die Form von Rechtecken, bedingte Sprünge sind durch Rauten gekennzeichnet. Vergleichs- und Sprungbefehl werden hier also durch ein einziges Symbol dargestellt, obwohl sie eigentlich durch zwei getrennte Befehle realisiert werden. Diese beiden Befehle müssen auch noch nicht einmal direkt hintereinander stehen; wegen ihres engen Bezuges vermeidet man aber meist eine Trennung.

Aufgaben:

1. Schreibe ein Programm, welches die Zahlen von 1 bis 10 addiert und das Ergebnis in der Zelle 100 speichert.

2. Ergänze das Programm aus Aufgabe 1 so, dass das Ergebnis in der Anzeige endlos angezeigt wird.

3. Zeichne ein Flussdiagramm zu dem Programm aus Aufgabe 2.

5. Versuche ein Programm zu schreiben, welches zwei Zahlen (in den Zellen 100 und 101) multipliziert und das Ergebnis in Zelle 102 ausgibt. Nutze aus, dass die Multiplikation auf die Addition zurückgeführt werden kann: $3 \cdot 5 = 5 + 5 + 5$.

4.4 Verzweigungen

Bedingte Sprünge werden nicht nur zur Programmierung von Schleifen benutzt; sie werden auch eingesetzt, wenn je nach Situation unterschiedliche Befehlsfolgen durchlaufen werden sollen. Eine derartige Programmstruktur bezeichnet man als eine **Verzweigung**.

An einem Beispiel wollen wir dies deutlich machen. In den Zellen 100 und 101 liegen zwei Messwerte - wir nennen sie A und B; unser Modellcomputer soll den Unterschied zwischen diesen beiden Zahlen bestimmen und anzeigen. Die Differenz A - B liefert das richtige Ergebnis aber nur dann, wenn $A \geq B$ gilt. Wenn nämlich $A < B$ ist, dann führt die Subtraktion A - B zu einer Fehlermeldung. Je nachdem, ob $A \geq B$ oder $A < B$ gilt, muss entweder die Rechnung A - B oder die Rechnung B - A durchgeführt werden. Im Flussdiagramm von Abb. 4.4 ist dies dargestellt.

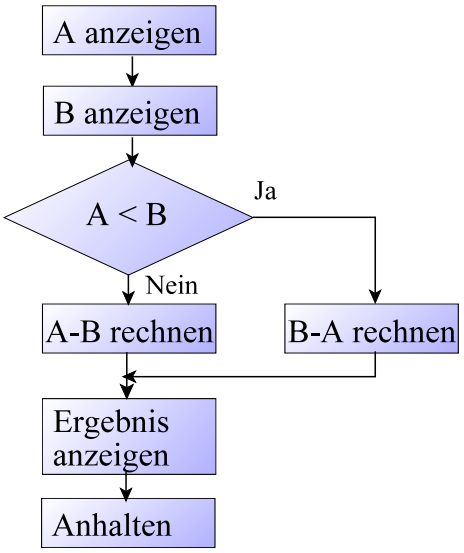


Abb 4.4

Das Programm zur Berechnung dieses Betrags ist in der folgenden Tabelle wiedergeggeben (unterschied.obj). Dabei steht im Kommentar die Abkürzung [xxx] für “Inhalt von xxx”.

Adresse	Mnemonic	Code	Kommentar
000	LDA 100	05.100	Lade [100] in den Akku
001	ANZ	02.000	Zeige Akkuinhalt
002	VZG 250	03.000	Verzögerung um 250 ms
003	LDA 101	05.101	Lade [101] in den Akku
004	ANZ	02.000	Zeige Akkuinhalt
005	VZG 250	03.000	Verzögerung um 250 ms
006	LDA 100	05.100	Lade [100] in den Akku
007	VKL 101	13.101	Ist Akkuinhalt kleiner als [101]
008	SPB 012	11.012	Wenn ja, dann springe nach 012
009	LDA 101	05.100	Lade [100] in den Akku
010	SUB 100	08.101	Rechne: [Akku] - [101]
011	SPU 014	09.014	Springe nach 014
012	LDA 100	05.101	Lade [101] in den Akku
013	SUB 101	08.100	Rechne: [Akku] - [100]
014	ANZ	02.000	Zeige Akkuinhalt
015	VZG 250	03.000	Verzögerung um 250 ms
016	ABS 102	06.102	Speichere Akkuinhalt in Zelle 102
017	HLT	01.000	Anhalten

Die bedingte Verzweigung bei einem Programm kann man vergleichen mit der Verzweigung einer Bahnstrecke (Abb. 4.5): Die Schienen beschreiben den Programmablauf. Ein Zug fährt von Schwelle zu Schwelle und arbeitet dabei nacheinander Befehle ab. Kommt er an eine Weiche, kann er auf zwei mögliche Weisen weiterfahren. Der Weicheneinsteller stellt die Weiche nun so ein, wie es die Flagge (flag) des Vergleichers vorgibt. Und der richtet sich nach dem Ergebnis des letzten Vergleichbefehls. Letztlich legt also die Bedingung die weitere Fahrt des Zuges, d. h. den weiteren Programmablauf fest.

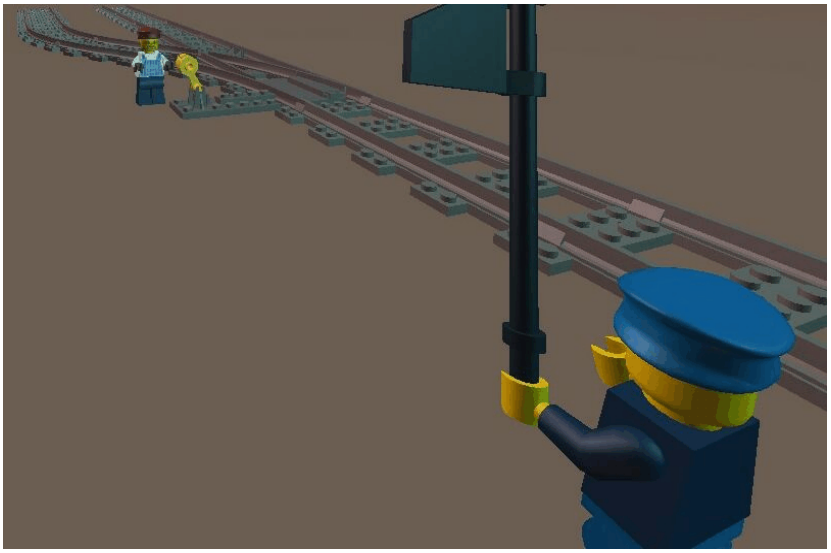


Abb. 4.5: Der Weichensteller richtet sich nach dem Flagwert.

Aufgabe

Wie muss das Programm “unterschied.obj” abgeändert werden, wenn man den VGR-Befehl benutzen möchte?

5. Aufbau und Funktionsweise des Mikroprozessors

5.1 Mikroschritte

Einige wichtige Bestandteile unseres Mikroprozessors haben wir schon in den vorangehenden Kapiteln kennen gelernt:

- den Akku, in dem u. a. Zwischenergebnisse festgehalten werden
- den Programmzähler (PC), in welchem die Adresse des aktuell zu bearbeitenden Befehls steht.
- die arithmetisch logische Einheit (ALU), welche Rechnungen und Vergleiche durchführt.

Damit der Mikroprozessor korrekt arbeiten kann, muss er die Informationen aus dem RAM, dem Akku und dem Programmzähler in der richtigen Reihenfolge verarbeiten. Dabei hängt die Art und Weise sowie auch die Reihenfolge der Verarbeitung von den Inhalten des RAMs selbst ab: Schließlich stehen in diesem ja nicht nur Werte (die Operanden), sondern auch Befehle (die Operatoren). Zu diesem Zwecke benutzt unser Mikroprozessor weitere Elemente, welche durch den Datenbus und andere Leitungen miteinander verbunden sind (Abb. 5.1):

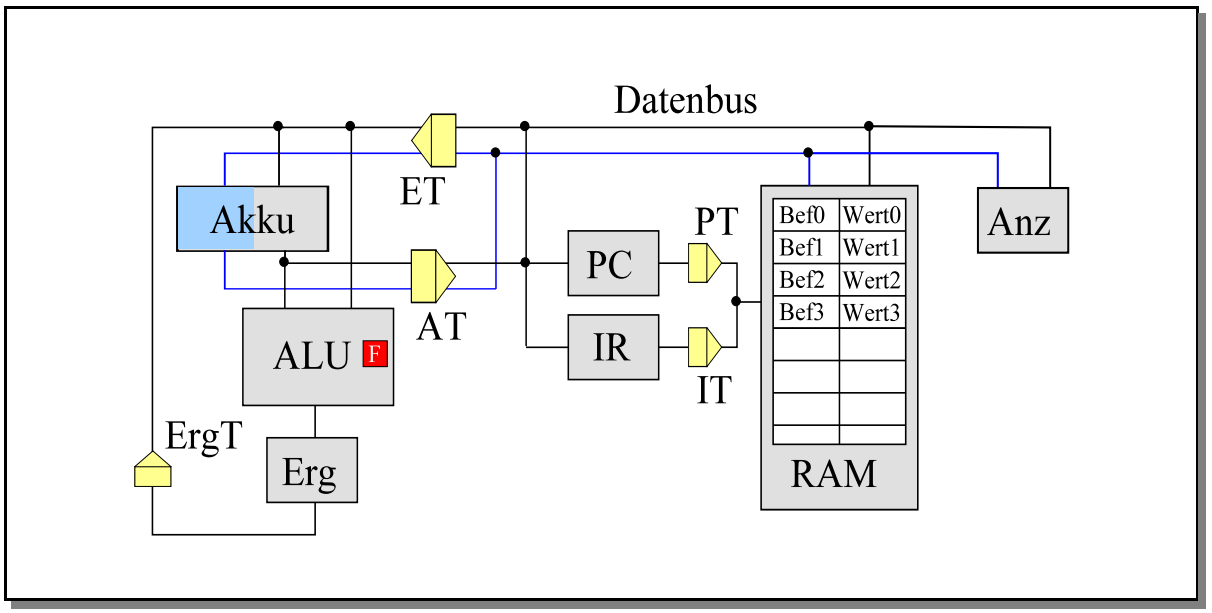


Abb. 5.1: Vereinfachte Schaltskizze unseres Modellrechners. Die Steuerleitungen (s. folgenden Text) sind nicht eingezeichnet.

- weitere Register (Ergebnisregister Erg, Anzeigeregister Anz und Indexregister IR)
- so genannte Tore (Eingangstor ET, Ausgangstor AT, Ergebnistor ErgT, Programmzählertor PT, Indexregistertor IT)

Wie diese Elemente funktionieren und welche Bedeutung sie für unseren Mikroprozessor haben, werden wir uns jetzt der Reihe nach anschauen.

Die einzelnen Bauteile sind durch Leitungen miteinander verbunden. In Wirklichkeit handelt es sich hier aber nicht um einzelne Kabel, sondern um Kabelstränge mit jeweils 8 Leitungen. Über diese Stränge können Zahlen von 1 Byte übertragen werden. Auch die Bauteile sind auf 1-Byte-Werte ausgelegt: Die ALU kann jeweils zwei Werte zwischen 0 und 255 verarbeiten, die Speicher Erg, PC und IR können jeweils 1-Byte-Zahlen speichern. Der Akku bildet hier (wie auch das RAM, s. u.) eine Ausnahme: Er ist als Speicherpaar konzipiert; auf diese Weise kann er einen vollständigen Befehl (Operator und Operanden) speichern; dementsprechend besteht der Datenbus aus 2 · 8 Leitungen (vgl. Abb. 5.1).

Alle Bauteile besitzen einen oder mehrere Eingänge und einen Ausgang. Über den Eingang werden die Daten entgegengenommen, über den Ausgang werden sie zur weiteren Verarbeitung ausgegeben. Lediglich beim RAM sind die Ein- und Ausgangsleitungen identisch; ob Daten eingegeben oder ausgegeben werden, hängt von zusätzlichen Steuersignalen ab (Write-Signal; Read-Signal); die dazu benötigten Signal- oder Steuerleitungen sind der Übersichtlichkeit halber nicht in Abb. 5.1 eingezeichnet. Das RAM besteht aus 128 Speicherpaaren; diese Speicherpaare sind gerade die uns schon bekannten Zellen. Welche der 128 Speicherzellen gerade über die Ein- bzw. Ausgangsleitungen mit dem Datenbus verbunden ist, wird durch den Wert im Speicher **PC** (**P**rogramm **C**ounter = Programmzähler) festgelegt.

Neben den Registern gibt es noch so genannte Tore. Bei einem **Tor** handelt es sich um eine Schaltereinheit aus 8 elektronischen Schaltern, welche durch ein elektrisches Signal gemeinsam ein- und ausgeschaltet werden können (Abb. 5.2). Derartige Tore können im Gegensatz zu mechanischen Schaltereinheiten Signale nur in eine Richtung hindurch lassen; in Abb. 5.1 zeigt die Spitze der Torsymbole in Durchlassrichtung.

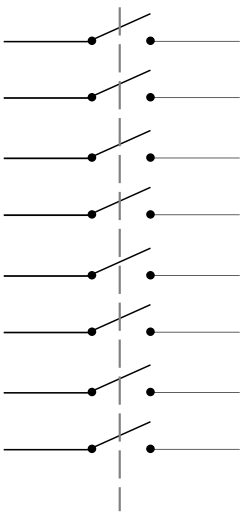


Abb. 5.2

Wie wirken nun diese Komponenten zusammen? An zwei Beispielen, den Befehlen AKO und LDA, soll dies deutlich gemacht werden. Zunächst wird der AKO-Befehl betrachtet; die Ausgangssituation sei folgende: Im Programmzähler (PC) steht die Adresse 005. In der Zelle mit der Adresse 005 befindet der Befehl AKO 37: "Transportiere die Zahl 37 in den Akku." Welche Schritte sind nun erforderlich?

Als Erstes wird das **PC-Tor** (**PT**) geöffnet; im RAM wird dadurch die Zelle 005 adressiert. Nun wird an das RAM ein Read-Signal gegeben; die Informationen der Speicherzelle 005 liegen dann auf dem Datenbus. Anschließend wird das Eingangstor geöffnet; jetzt liegen die Informationen schon am Eingang des Akkus. Wird jetzt noch ein Merke-Signal an den Akku gegeben, übernimmt er diese Informationen. Um bei den

folgenden Befehlen keine Probleme zu bekommen, werden am Schluss alle Steuersignale wieder zurückgesetzt. Insbesondere wird das Eingangstor wieder geschlossen. (Hier besteht eine kleine Abweichung vom Original-CP1: Dort wird der Operator nicht in den Akku übernommen!) Zuguterletzt wird der Inhalt des Programmzählers um 1 erhöht. Dies geschieht mit einem so genannten Inkrementier-Signal (inkrementieren = erhöhen); durch dieses Signal kann der Programmzähler direkt, d.h. ohne Unterstützung von anderen Bausteinen, seinen Wert inkrementieren.

Diese oben genannten Schritte darf man nicht mit den einzelnen Befehlen eines Programms verwechseln. Vielmehr ist es so, dass diese Schritte einen einzigen Befehl bilden. Man nennt sie **Mikroschritte**. Ein Befehl setzt sich also aus mehreren Mikroschritten zusammen.

Bei dem Befehl LDA 102 ist diese Mikroschrittfolge schon etwas komplizierter. Zuerst werden wieder das PT geöffnet und das Read-Signal für das RAM gegeben. Nun wird der Wert 102 in das Index-Register übernommen und das Read-Signal kann ausgeschaltet werden. Anschließend wird das Index-Tor geöffnet; damit es zu keiner Datenkollision (Abb. 5.3) kommt, muss das PC-Tor allerdings vorher geschlossen worden sein. Jetzt ist die Zelle 102 im RAM adressiert. Die folgenden Mikroschritte sind wie beim AKO-Befehl. Dadurch wandert der Inhalt der Zelle 102 in den Akku. Zuletzt werden das Index-Tor wieder geschlossen und der PC inkrementiert.

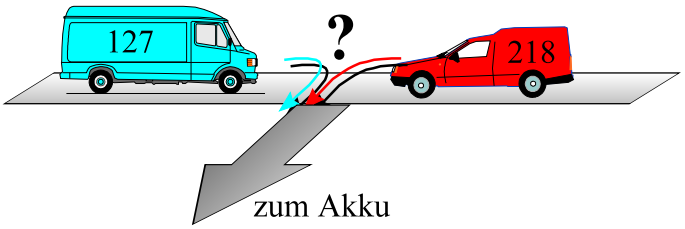


Abb. 5.3: Wenn sich zwei verschiedene Daten auf dem Datenbus befinden, kommt es zu einer Kollision; es ist dann nicht klar, welches Datum zum Akku gelangt.

5.2 Mikroschritte verfolgen mit MikroPC

Die Mikroschrittfolgen eines Befehls können mitunter recht komplex sein. Um diese Schrittfolgen zu veranschaulichen, wurde das Programm MikroPC (Abb. 5.4) entwickelt. MikroPC arbeitet ähnlich wie das schon bekannte Simulationsprogramm MiniPC. Es ist zwar nicht ganz so leistungsstark, dafür gestattet es aber einen Blick in die internen Vorgänge des Prozessors. Alle Bauteile aus der Schaltskizze in Abb. 5.1 sind hier wiederzufinden: Von den Registern über die ALU bis zum RAM. Der Datenbus ist bei MikroPC zwar graphisch etwas vereinfacht dargestellt, arbeitet aber genauso wie in der Abb. 5.1 dargestellt.

In diesem Abschnitt werden wir die Bauteile nur per Handsteuerung bedienen; später werden wir sehen, wie dies mit Hilfe des so genannten Steuerwerks automatisiert werden kann. Tore und Register werden über die linke Maustaste geöffnet und geschlossen. Dabei hat das PC-Register noch eine Spezialeigenschaft: Sein Inhalt kann über die rechte Maustaste inkrementiert werden; über die Reset-Schaltfläche kann es wieder auf 000 gesetzt werden. Bei der ALU kann man die gewünschte Operation über ein Auswahlfeld einstellen. Wie schon beim Programm MiniPC können die Inhalte des RAMs direkt editiert werden.

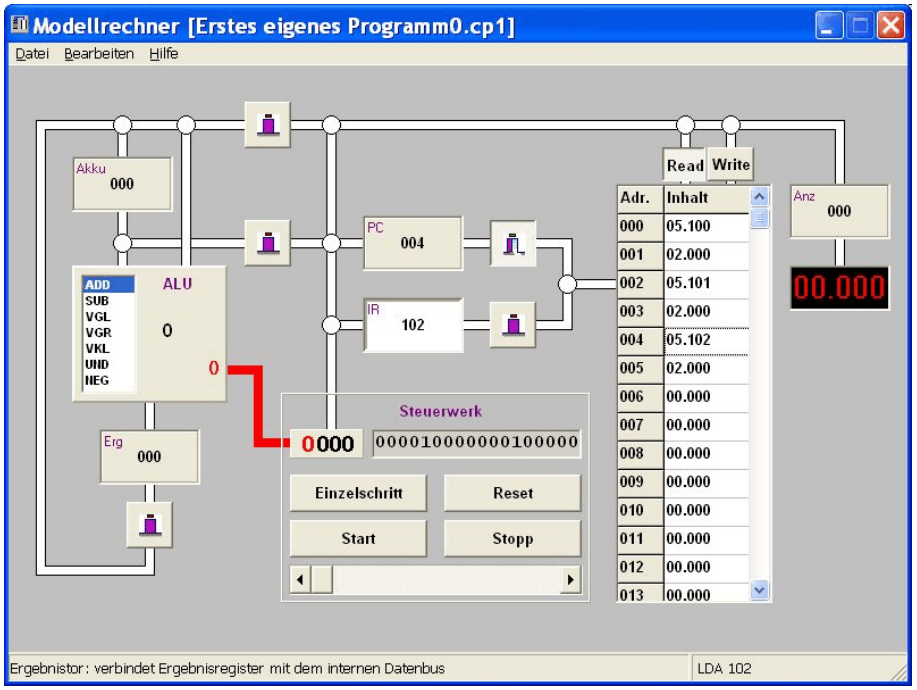


Abb. 5.4

In Abb. 5.4 wurde über Datei - Öffnen bereits eine obj-Datei (Erstes_Programm0.obj) in das RAM geladen. Auch wurden hintereinander schon das PC-Tor (PT), das RAM-Read (Read) und

das Indexregister (IR) angeklickt; dadurch wurde zunächst die Zelle mit der Adresse 004 aktiviert (man erkennt das an dem grauen Rahmen der entsprechenden Zelle), und deren Operand wurde über den externen Datenbus in das Indexregister übernommen. Das Indexregister zeigt nun den Inhalt 102 an; das ist die Adresse der Zahl, die in den Akku geladen werden soll.

Wie geht es nun weiter? Zunächst klicken wir wieder auf das Indexregister; dadurch wird sein Eingang geschlossen und es behält seinen Wert, auch wenn der Datenbus andere Werte erhält. Nun schließen wir das PC-Tor (jetzt wird automatisch die Zelle 000 adressiert) und öffnen danach das Indexregister-Tor; dadurch wird die Zelle 102 adressiert. In dieser befindet sich der Wert 033 (genauer: 00.033). Da das RAM-Read-Tor noch geöffnet ist, genügt jeweils ein Klick auf das Eingangstor und auf den Akku – und schon befindet sich der Wert 033 im Akku (Abb. 5.5).

Jetzt schließen wir zunächst den Akku und dann die restlichen Tore; der Befehl LDA 102 wurde fast vollständig ausgeführt. Was fehlt, ist das Inkrementieren des PC. Das erledigen wir, indem wir zuguterletzt noch einmal mit der rechten Maustaste auf das PC-Register klicken.

Es ist sehr lehrreich, diese Folge von Mikroschritten einmal selbst an diesem Simulationsprogramm auszutesten. Auf jeden Fall sollte man auch einmal andere Reihenfolgen austesten. Insbesondere ist es interessant zu beobachten, was geschieht, wenn man “vergisst”, ein Register rechtzeitig zu schließen.

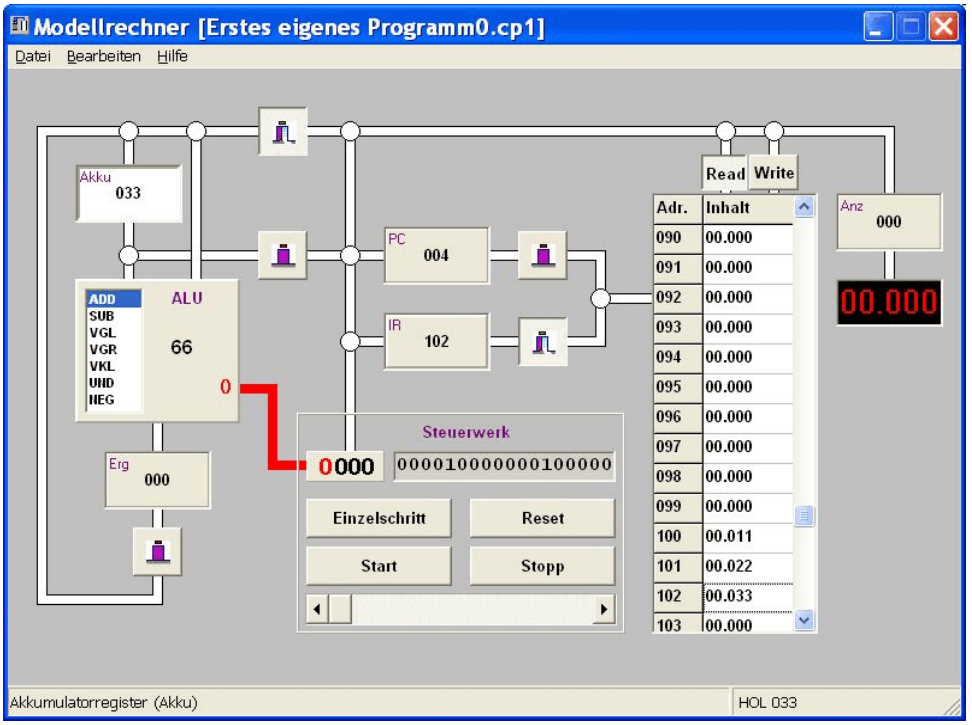


Abb. 5.5

Aufgaben

5.1 Trage den Wert 033 in die Zelle 000 ein. Lösche den PC mit der Reset-Schaltfläche und lade den Wert 033 in den Akku.

Bei den folgenden Aufgaben wird davon ausgegangen, dass im Akku der Wert 033 steht.

5.2 Überlege eine Mikroschrittfolge für ANZ und teste sie mit MikroPC aus.

5.3 Schreibe den Code des Befehls ABS 001 in die Zelle 000, stelle den PC auf 000, überlege eine Mikroschrittfolge für diesen Befehl und teste sie mit MikroPC aus.

5.4 Warum zeigt die ALU in Abb. 5.5 den Wert 66 an?

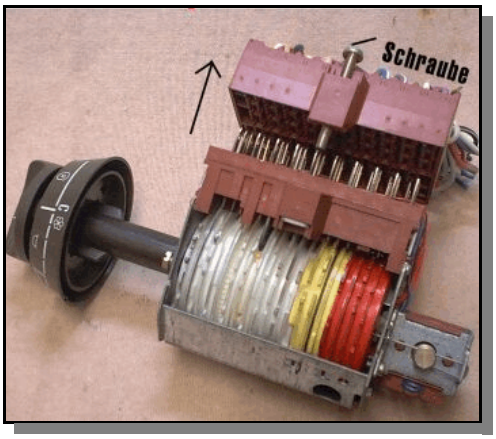


Abb. 5.6

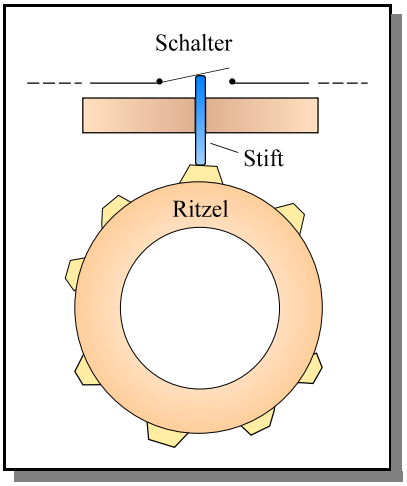


Abb. 5.7

5.3 Steuerwerk

Bislang haben wir nur beschrieben, in welcher Reihenfolge welche Bauteile aktiv werden müssen, d. h. welche Steuersignale gegeben werden müssen. Was wir noch nicht geklärt haben, ist: Auf welche Weise werden diese Steuersignale überhaupt in einem Mikroprozessor *automatisch* erzeugt? Dieser Frage widmen wir uns jetzt.

Ein ähnliches Steuerproblem taucht auch bei Waschmaschinen auf. Hier müssen – je nach "Waschprogramm" – in festgelegter Reihenfolge verschiedene elektrische Bauteile wie Ventile, Heizung oder Motoren ein- und ausgeschaltet werden. Dazu bedient man sich häufig eines Steuerwerks wie in Abb. 5.6. Es besteht aus einer Walze, die von einem Motor langsam gedreht wird. Auf dieser Walze sind unregelmäßige Zahnkränze angebracht. Oberhalb der Walze befindet sich eine Reihe von elektrischen Schaltern, diese werden von den Zähnen der Walze über Stifte einzeln betätigt.

Die einzelnen Schritte beim Schleudern können z. B. so aussehen:

Schalter für...	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pumpe Wasserablauf	X			X			X
Ventil Wasserzulauf							
Heizung							
Motor Linkslauf langsam		X					
Motor Linkslauf schnell				X		X	
Motor Rechtslauf langsam			X				
Motor Rechtslauf schnell					X		X
Ventil Waschmittelzufuhr							

Jeder Schalter wird durch einen eigenen Zahnkranz gesteuert (Abb. 5.7). Die Kreuze in der Tabelle geben an, welche Schalter in welcher Phase geschlossen werden. Im ersten Schritt wird nur die Pumpe für den Wasserablauf eingeschaltet. Im zweiten ist diese Pumpe wieder ausgeschaltet, dafür dreht sich der Motor für die Trommel langsam nach links, so dass sich die Wäsche in der Trommel verteilt und weiteres Wasser aus der Wäsche austreten kann. Im dritten Schritt läuft die Trommel rechts herum. Anschließend wird wieder Wasser abgepumpt und die erste Schleuderphase eingeleitet: Der Motor dreht sich schnell rechts herum. Die Drehrichtung ändert sich in den nächsten Schritten immer wieder und am Schluss wird noch einmal Wasser abgepumpt. Damit die Motoren nicht abrupt ihre Richtung ändern, könnte man noch zusätzliche Auslaufphasen einplanen; zur Vereinfachung haben wir hier aber darauf verzichtet.

Der Wechsel von einer Phase zur nächsten erfolgt durch die Drehung der Walze; dieser wird von einem eigenen Motor angetrieben. Wie rasch sich der Motor dreht, kann auch über einen weiteren Zahnkranz auf der Walze beeinflusst werden. Auf diese Weise können bestimmte Phasen sehr schnell durchlaufen, im Extremfall sogar übersprungen werden.

Ein Segment der Walze steht für einen bestimmten "Waschbefehl": Eines für den Vorwaschgang, eines für den Hauptwaschgang, eines für das Schleudern. Diese Befehle werden durch einzelne Schritte, wie sie in der Tabelle beispielhaft für den Schleuderbefehl dargestellt sind, ausgeführt. Diesen Einzelschritten entsprechen bei unserem Mikroprozessor gerade die Mikroschritte.

Auch unser Mikroprozessor besitzt ein **Steuerwerk** (SW). Wie sieht dieses nun aus? Sicherlich befindet sich in unserem Mikroprozessor keine Walze mit mechanischen Schaltern. In der Tat greift man hier wieder auf ein elektronisches Bauteil zurück: das **ROM**. Die Abkürzung ROM steht für *R*ead *O*nly *M*emory, das bedeutet Nur-Lese-Speicher. Das ROM kann als ein RAM aufgefasst werden, welches vom Benutzer nur gelesen, nicht aber beschrieben werden kann. Die Speicherinhalte werden vom Hersteller ein für alle Male festgelegt. Für unser Steuerwerk ist das auch sinnvoll; schließlich sollen die Abfolgen von Mikroschritten bei den einzelnen Prozessorbefehlen immer dieselben sein!

In jeder einzelnen Speicherzelle des ROMs sind die Schalterstellungen für einen einzigen Mikroschritt abgespeichert, und zwar als Folge von Nullen und Einsen: Eine 1 steht hier für eingeschaltet, eine 0 für ausgeschaltet. Da unser Prozessor insgesamt 18 Steuerleitungen besitzt, müssen in jeder dieser Zellen insgesamt 18 Einsen und Nullen stehen.

Die Befehle unseres Mikroprozessor sind so ausgelegt, dass sie sich durch maximal 10 Mikroschritte (von 0 bis 9 durchnummeriert) erledigen lassen. Ein einziger Befehl lässt sich also durch 10 Speicherzellen vollständig beschreiben. In Abb. 5.8 ist ein Teil des Steuerwerk-ROMs für einen Befehl dargestellt.

Die ersten 10 Zellen liefern die Steuersignale für den Befehl 00, die nächsten 10 Zellen für den Befehl 01, usw. Die erste Ziffer der Zelladresse (von rechts gelesen) gibt also immer die Nummer des Mikroschritts an; die nächsten beiden Ziffern kennzeichnen den Befehl. In der Zelle 023 stehen also die Steueranweisungen für den Mikroschritt 3 des Befehls mit dem Code 02.

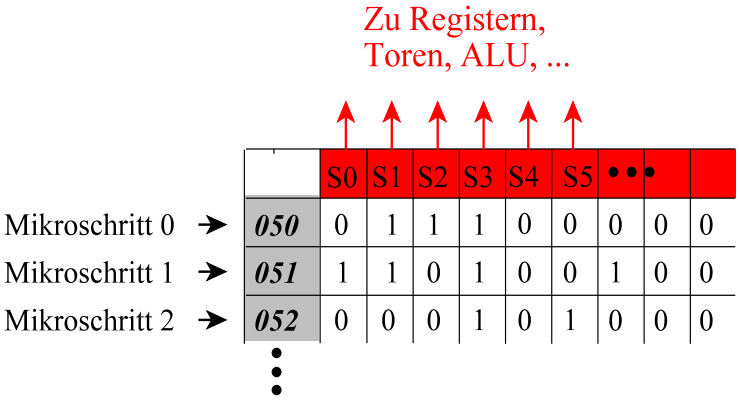


Abb. 5.8: Ein Ausschnitt aus dem Mikroschritt-ROMs: Die ersten drei Zellen des Befehls 05 (schematisch).

Was für das RAM der Programmzähler ist, das ist für unser Mikroschritt-ROM der Mikroschrittzähler (**MPC** = *M*icro *P*rogram *C*ounter): Dieses Register kann genauso wie der Programmzähler über eine Steuerleitung inkrementiert werden. Diese Steuerleitung wird in regelmäßigen Zeitabständen aktiviert. Die Signale werden von einer elektronischen Uhr (Taktgeber) ausgeschickt; deswegen bezeichnet man die Signale auch als Takt- oder Clock-Signale (clock = Uhr). Daneben besitzt der Mikroschrittzähler auch noch einen Lösch-Eingang CLEAR; wenn an diesen Eingang ein Signal gegeben wird, wird der Inhalt des MPC auf 000 gesetzt.

Der Ausgang des MPC ist derart mit dem Mikroschritt-ROM verbunden, dass der Inhalt des MPC die aktuelle Zelle des ROM bestimmt (vgl. Abb. 5.9). Bei jedem Clock-Signal wird von einem zum anderen Mikroschritt gewechselt. MPC und Taktgeber stehen also für den Motor, der unser mechanisches Steuerwerk der Waschmaschine angetrieben hat.

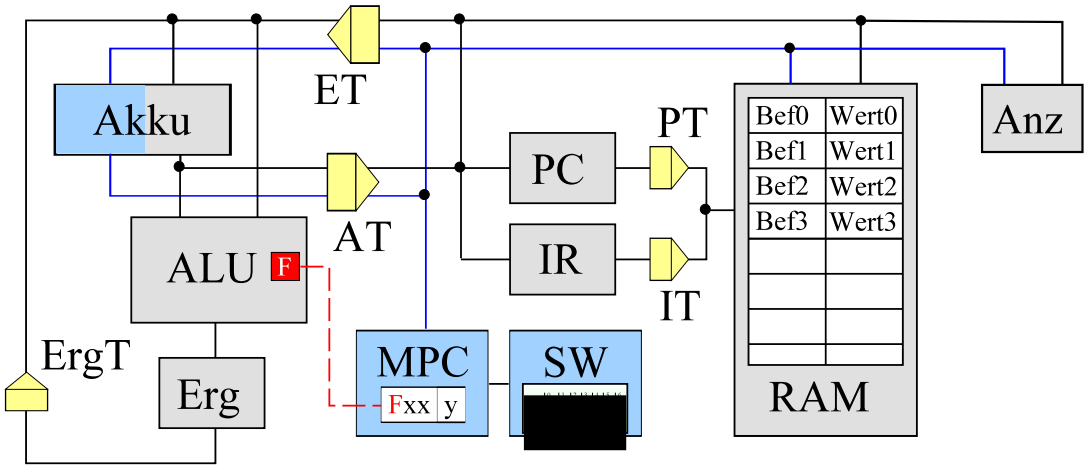


Abb. 5.9

Auch unser Programm MikroPC besitzt ein solches Steuerwerk mit einem ROM, in welchem die wichtigsten Befehle unseres Modellrechners abgespeichert sind. An dem Befehl LDA 012, der uns nun hinlänglich vertraut ist, soll dies deutlich gemacht werden.

Wir starten MikroPC und geben ein:

- 05.012 in die Zelle 000
- 00.123 in die Zelle 012

Wenn man nun einmal auf die Schaltfläche Einzelschritt klickt, steht der MPC auf 050. (Warum zunächst dieser Einzelschritte erfolgen muss, wird im nächsten Abschnitt behandelt.) Durch die Einsen in der Zelle 050 des ROMs sind bereits drei Bausteine aktiviert: PT, IR und RAMRead. Bei jedem weiteren Einzelschritt werden weitere Signale gegeben. In Abb. 5.10 ist die Situation beim vierten Schritt dargestellt: Der MPC steht auf 053; in dieser ROM-Zelle finden wir die Schalterstellung 100001100001000000. Dadurch werden hier Akku, ET, RAMRead und IT aktiviert. Die grünen Pfeile zeigen, welche Eins hier gerade welche Komponente steuert. Klickt man noch weitere zwei Mal auf die Einzelschritt-Schaltfläche, so ist der Befehl 05.012 vollständig abgearbeitet: Der Eintrag 123 aus der Zelle 012 steht im Akku und der Inhalt des Programmzählers ist um 1 erhöht worden. Warum am Ende der MPC wieder auf 000 springt, auch das werden wir noch im nächsten Abschnitt klar machen.

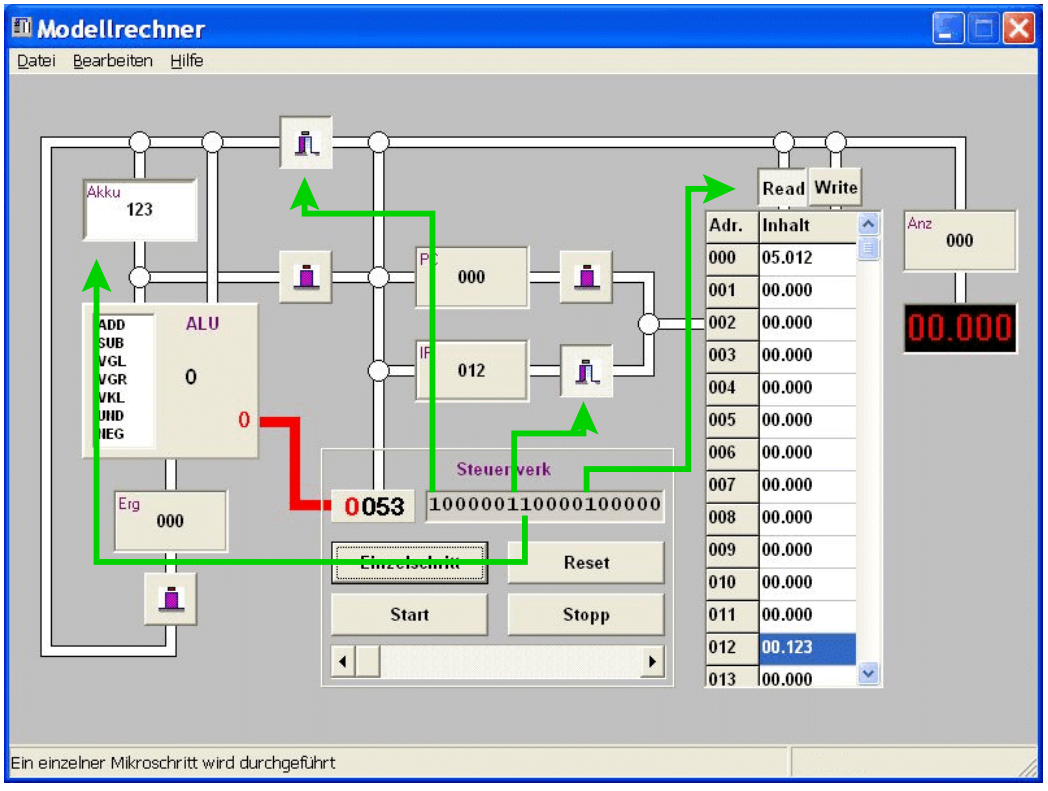


Abb. 5.10

Aufgabe

A 5.5 Studiere mit Hilfe des Programms MikroPC die Befehle AKO 123, ABS 005. Starte MikroPC neu, schreibe diese Befehle in die RAM-Zellen 000 und 001und beobachte, was geschieht, wenn die Einzelschritt-Schaltfläche betätigt wird.

5.4 Der von-Neumann-Zyklus

Einige Fragen sind aus dem letzten Abschnitt noch übrig geblieben: Was hat es mit den zusätzlichen Mikroschritten auf sich? Welche Eins ist für welche Komponente (wir sagen auch: Element) zuständig? Wie arbeiten alle diese Elemente zusammen? Wie wird erreicht, dass ein Befehl nach dem anderen vom Steuerwerk abgearbeitet wird?

An einem konkreten Beispiel wollen wir dies erläutern: Am Ausgangspunkt unserer Betrachtungen sollen wieder einmal der LDA-Befehl stehen. Der entsprechende Code 05.012 soll sich wieder in der RAM-Zelle 000 befinden und in der Zelle 012 steht wieder das Datum 123. der PC soll auch schon auf diese Zelle 002 weisen. In der nächsten RAM-Zelle 001 möge ein AKO-Befehl stehen: 04.157. Ferner gehen wir davon aus, dass MikroPC resettet wurde und nach einem ersten Einzelschritt der Code dees LDA-Befehls in den MPC übernommen worden ist; in unserem Fall steht dann also der Wert 050 im MPC. Wie dieser Code überhaupt in den MPC gelangt, werden wir gleich noch erklären.

Durch den MPC wird die Zelle 050 unseres SW-ROMs adressiert und die entsprechenden Steuerleitungen werden unverzüglich aktiviert. Wir können sie der folgenden Tabelle entnehmen. Im Gegensatz zur Abb. 5.8 sind hier allerdings die Mikroschritte waagerecht und die Steuersignale senkrecht angeordnet. Einsen sind durch ein X gekennzeichnet, Nullen werden nicht notiert; so wird die Tabelle übersichtlicher. Im Schritt 0 sind demnach das PC-Tor und das Indexregister-Tor geöffnet sowie das RAM-Read-Signal aktiviert worden.

Element	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Eingangstor (ET)				X	X					
Ausgangstor (AT)										
Programmzähler (PC)										
PC-Inkrement (PCI)					X					
PC-Tor (PT)	X	X								
Indexregister (IR)	X									
IR-Tor (IT)			X	X	X					
Akku				X						
Ergebnisregister (Erg)										

Element	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Erg-Tor (ErgT)										
MPC-Clear (pos. Flanke)						X				
MPC-Load (pos. Flanke)										
RAM-Read	X	X		X	X					
RAM-Write										
ALU0										
ALU1										
ALU2										
Anz										

Damit liegt der Operand 012 auf der schwarzen Leitung des Datenbusses und das Datum gelangt in das geöffnete IR-Register. Im nächsten Schritt wird das IR-Register wieder geschlossen. Danach wird das PC-Tor geschlossen und das Indexregister-Tor IT geöffnet; dadurch wird nun die Zelle 004, sondern die Zelle 012 im RAM adressiert. Im Mikroschritt 3 gelangt der Wert der Zelle 012 durch das RAM-Read-Signal auf den Datenbus und wird durch das geöffnete Eingangstor ET in den Akku geladen. Würde man im nächsten Schritt den Eingang des Akkus gleichzeitig mit ET und dem RAM-Read deaktivieren, könnte der Inhalt des Akkus wieder gelöscht werden. Bedingt durch die leicht unterschiedlichen Laufzeiten der Signale auf den Leitungen könnte nämlich das Eingangstor um den Bruchteil einer Sekunde früher geschlossen werden als der Akku-Eingang; in dieser kleinen Zeitspanne würde nicht mehr der Wert der Zelle 012 am Eingang des Akkus anliegen; und wenn der Akku-Eingang endlich geschlossen ist, würde auch nicht mehr die der Wert der Zelle 012 im Akku stehen. (Bei geschlossenem Tor sind die einzelnen Leitungen bei unserem Simulationsprogramm MikroPC normalerweise auf 0, so dass im Akku dann der Wert 000 gespeichert würde.) Aus diesem Grund lässt man beim Schließen des Akku-Eingangs im Mikroschritt 4 sowohl das Eingangstor offen und lässt RAM-Read aktiviert. Gleichzeitig kann man aber schon den Programmzähler, der ja immer noch auf 000 steht, um 1 erhöhen. Wäre das PT jetzt nicht geschlossen, würde schon der nächste Befehl, in unserem Fall AKO 127, adressiert. Im letzten Schritt wird das MPC-Clear-Signal gegeben. Direkt zu Beginn dieses Schrittes, also beim Einschalten des Signals, wird der MPC gelöscht. Er zeigt damit auf den Schritt 0 des Befehls mit dem Code 00.

Dieser Befehl mit dem Code 00 ist ein ganz besonderer Befehl; es handelt sich um den so genannten **HOL-Befehl**: Wie der Name andeutet, holt er den nächsten Befehl aus dem RAM und lädt ihn in den MPC. Die Mikroschritt-Tabelle des HOL-Befehls sieht so aus:

Element	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Eingangstor (ET)										
Ausgangstor (AT)										
Programmzähler (PC)										
PC-Inkrement (PCI)										
PC-Tor (PT)	X	X								
Indexregister (IR)										
IR-Tor (IT)										
Akku										
Ergebnisregister (Erg)										
Erg-Tor (ErgT)										
MPC-Clear										
MPC-Load		X								
RAM-Read	X	X								
RAM-Write										
ALU0										
ALU1										
ALU2										
Anz										

Die Tabelle ist fast leer; es passiert also nicht viel. Aber was hier geschieht, ist entscheidend für den automatischen Ablauf eines Programms: Im ersten Schritt (0) wird das Programmzählertor (PT) geöffnet und im RAM die Zelle 001 angesteuert; erinnern wir uns daran, dass in einem der letzten Mikroschritte des vorangehenden Befehls der Programmzähler um 1 erhöht worden ist und damit jetzt auf die nächste Zelle zeigt. Durch das RAM-Read-Signal liegt der Befehlscode dieses nächsten Befehls aus der Zelle 002 nun auf dem Datenbus. Wenn nun im Mikroschritt 1 das MPC-LOAD-Signal gegeben wird, wird dieser Befehlscode über die blaue Leitung des Datenbusses in den MPC übernommen (Abb. 5.9). Genauer gesagt, werden beim MPC die beiden Ziffern xx durch den Befehlscode ersetzt und der Wert y auf 0 gestellt; in unserem Fall erhält der MPC also den Wert 040. Dies geschieht ***direkt zu Beginn*** dieses zweiten Schrittes; ***unmittelbar danach*** weist der MPC schon im SW-ROM auf den Mikroschritt 0 des AKO-Befehls. Seine weiteren Mikroschritte werden dann ähnlich wie beim LDA-Befehl abgearbeitet, bis am Ende wieder der PC inkrementiert und der MPC gelöscht wird.

Bei jedem Befehl (außer dem HOL-Befehl) endet die Mikroschrittfolge also damit, dass der Programmzähler um 1 erhöht wird und anschließend der MPC gelöscht wird. Dadurch wird der HOL-Befehl eingeleitet, welcher seinerseits den nächsten Befehlscode in den MPC lädt und ausführen lässt; dieser endet wieder mit dem Starten des HOL-Befehls usw.

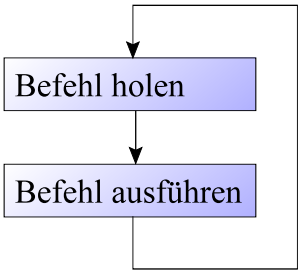


Abb. 5.11

Befehl ausführen - Befehl holen - Befehl ausführen - Befehl holen ..., das ist das Konzept, auf dem der automatische Ablauf unseres Programms basiert. Es geht auf den Mathematiker John von Neumann zurück; deswegen nennt man diese Abfolge auch **von-Neumann-Zyklus**.

Ein letztes Beispiel: der Additionsbefehl ADD. Erinnern wir uns: Durch ADD xxx wird der Wert der Zelle xxx zum Wert des Akkus addiert und in diesem gespeichert. Die Addition wird von der ALU durchgeführt. Nun kann die ALU neben der Addition auch noch weitere Aufgaben durchführen. Welche Operation sie gerade durchführen soll, wird durch Steuersignale festgelegt; in unserem Fall reichen dazu für die ALU drei Steuerleitungen aus:

	ALU2	ALU1	ALU0
CLR	0	0	0
ADD	0	0	1
SUB	0	1	0
VGL	0	1	1
VGR	1	0	0
VKL	1	0	1
UND	1	1	0
NEG	1	1	1

Die ersten Mikroschritte des ADD-Befehls sind (vom zusätzlichen ALU0-Signal abgesehen) mit denen des LDA-Befehls identisch. Das Ergebnis der Addition wird schließlich über das Ergebnisregister und das Ergebnistor in den Akku geschrieben.

Element	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Eingangstor (ET)				X	X					
Ausgangstor (AT)										
Programmzähler (PC)										

Element	0	1	2	3	4	5	6	7	8	9
PC-Inkrement (PCI)								X		
PC-Tor (PT)	X	X								
Indexregister (IR)	X									
IR-Tor (IT)			X	X	X					
Akku						X				
Ergebnisregister (Erg)				X						
Erg-Tor (ErgT)						X	X			
MPC-Clear									X	
MPC-Load										
RAM-Read	X	X		X	X					
RAM-Write										
ALU0			X	X	X					
ALU1										
ALU2										
Anz										

5.5 Steuerwerk und Mikrocodetabelle bei MikroPro

Das Programm MikroPC kann, wie wir bereits im Abschnitt 5.3 gesehen haben, die Abläufe bei der Bearbeitung einzelner Befehle veranschaulichen. Da es auch den HOL-Befehl beherrscht, ist es auch in der Lage, selbstständig einen neuen Befehlscode in den MPC zu laden. Somit können ganze Folgen von Befehlen, also auch vollständige Programme, von MikroPC bearbeitet werden. Wir wollen nun den Mikroprozessor bei seiner Arbeit studieren. Dazu benutzen wir zunächst den Einzelschrittmodus; so können wir in Ruhe die Auswirkungen sämtlicher Signale des Steuerwerks betrachten.

Beginnen wir mit einem einfachen Beispiel. Dazu laden wir das “Erstes_ Programm0.obj” aus 3.3 in das RAM. Dieses Programm zeigt lediglich nacheinander die Zahlen 11, 22 und 33 im Display an. Beim Laden der RAM-Datei werden automatisch alle Register gelöscht und sämtliche Tore geschlossen. Damit ist unser Mikroprozessor bereit, das Programm abzuarbeiten. Insbesondere haben der Programmzähler und der MPC den Inhalt 000 erhalten und der zugehörige 18-stellige Steuercode aus dem SW-ROM wird automatisch angezeigt: 000010000000100000. Hierbei werden diejenigen Steuerelemente durch eine 1 gekennzeichnet,

die über die (nicht abgebildeten) Steuerleitungen aktiviert werden sollen. Dabei entspricht die erste 1 (von links gelesen) dem PC-Tor, und die zweite 1 dem RAM-Read. Allgemein ergibt sich die Zuordnung der einzelnen Ziffern des Steuercodes zu den Steuerelementen aus der Reihenfolge in der Mikroschritttabelle. MikroPC zeigt uns dies auch prompt an: Die Elemente PC-Tor (PT) und RAM-Read sind aktiviert. Das PC-Tor ist dementsprechend geöffnet und die Zelle 000 des RAMs wird adressiert. Da zusätzlich auch das RAM-Read-Signal gegeben worden ist, liegt nun der Inhalt der RAM-Zelle 000, also 05.100 auf dem externen Datenbus; insbesondere liegt der Code 05 am Eingang des MPC an.

Nun klicken wir auf die Einzelschritt-Schaltfläche. Jetzt wird der MPC um 1 erhöht und der Mikroschritt 1 des HOL-Befehls wird ausgeführt: Das Steuerwerk gibt das MPC-Load-Signal; dadurch übernimmt der MPC sofort den Code 05 vom Datenbus und es wird die ROM-Zelle 050 adressiert. Dieser Vorgang verläuft so rasch, dass man diesen zweiten Schritt des HOL-Befehls nicht wahrnehmen kann; er geht sozusagen direkt in den 1. Schritt des neu geladenen Befehls 05 über.

Bei den nächsten Clicks werden die Mikroschritte des LDA-Befehls abgearbeitet. Dies haben wir schon im Abschnitt 5.3 eingehend studiert. Am Ende dieser Schrittfolge steht der PC auf 1 und der MPC wird wieder 000 gesetzt; damit wird der 1. Schritt des HOL-Befehls eingeleitet.

Im Folgenden sollte man jeweils auch die Inhalte der übrigen Register und die Anzeige betrachten. Hintereinander erscheinen hier tatsächlich die Zahlen 11, 22 und 33. Irgendwann weist der Programmzähler auf die Zelle 006. Wenn nun der HOL-Befehl ausgeführt wird, wird der Code 00 geladen. Da dies ist aber gerade der Code des HOL-Befehls ist, wird aus derselben Zelle wieder der Code 00 geladen. Der Rechner befindet sich damit in einer Endlosschleife.

Hier zeigen sich Unterschiede zu MiniPC: Zunächst laufen bei MikroPC die einzelnen Schritte so langsam ab, dass die Zahlen auf der Anzeige gelesen werden können, ohne dass zusätzliche Verzögerungsbefehle wie bei MiniPC erforderlich sind. Auch ist beim MikroPC am Ende des Programms nicht unbedingt ein HLT-Befehl nötig. Das liegt daran, dass beim MiniPC noch ein Betriebssystem vorhanden ist; dieses Programm kontrolliert hier die Tastatureingaben und gibt nur dann die Kontrolle an die Befehle im RAM ab, wenn die RUN-Taste betätigt worden ist. Der HLT-Befehl dient nun dazu, die Kontrolle wieder an das Betriebssystem zurückzugeben, damit weitere Eingaben über die Tastatur erfolgen können. Bei MikroPC ist der HLT mit dem HOL-Befehl identisch.

Die Taktsignale für die einzelnen Mikroschritte können auch durch einen automatisch Taktgeber erfolgen. Dazu muss man nur die Schaltfläche *Start* betätigen. Wenn der Scrollbalken unterhalb dieser Schaltfläche am linken Anschlag steht, erfolgen die Taktsignale im Abstand von 1 Sekunde. Wenn die Taktfrequenz erhöht werden soll, muss man den Knopf auf dem Scrollbalken nach rechts verschieben. Am rechten Anschlag beträgt die Taktfrequenz 20 Mikroschritte pro Sekunde. Die automatischen Taktsignale werden mit Hilfe der Stopp-Schaltfläche beendet.

Es ist schon beeindruckend zu sehen, wie viele Aktionen schon bei kleinen Programmen vom Prozessor durchgeführt werden müssen. Deswegen sollte man unbedingt einige der Programme,

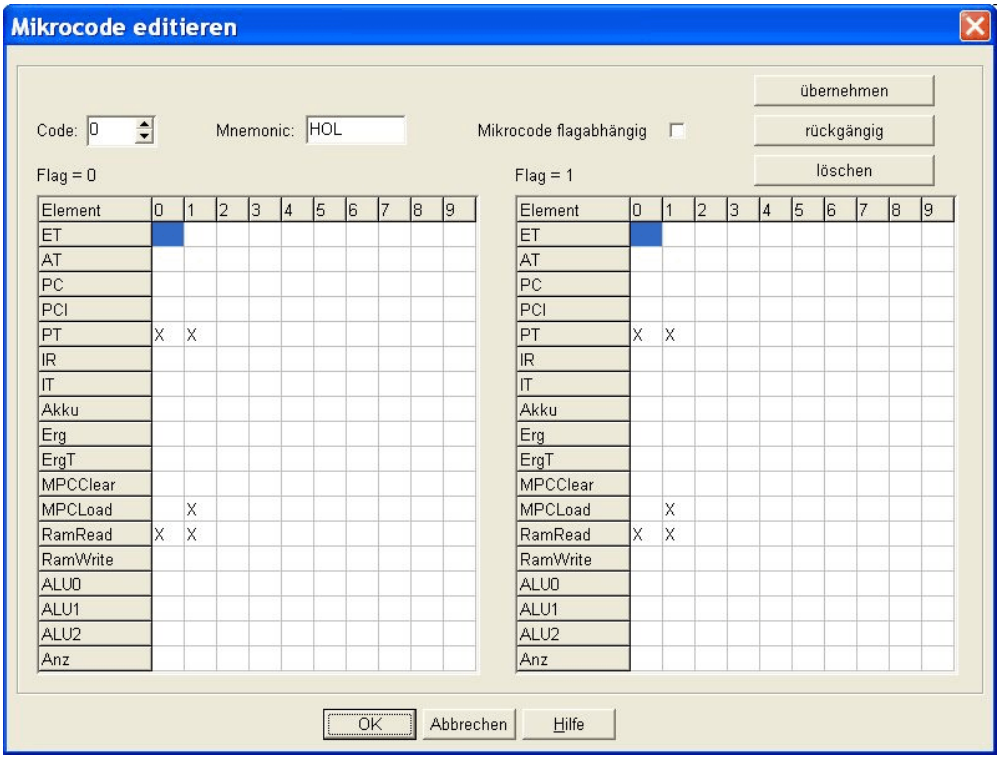


Abb. 5.12: Die Mikroschritttabelle des HOL-Befehls im Editor

welche schon mit MiniPC hergestellt wurden, mit MikroPC laufen lassen - durchaus auch mit maximaler Geschwindigkeit. Dann kann man zwar nicht mehr die einzelnen Aktionen nachvollziehen; es wird aber deutlich, dass die ganze Intelligenz des Prozessors nur in einer wohlgeordneten Abfolge von Ein- und Ausschaltvorgängen besteht.

MikroPC bietet die Möglichkeit, den Mikrocode, d. h. die Gesamtheit sämtlicher Mikroschrittfolgen, zu editieren. Die einzelnen Mikroschritte lassen sich also nicht nur anzeigen, man kann sie auch abändern oder sogar auch vollständig neue Befehle entwickeln. Den Mikrocode-Editor öffnet man mit *Bearbeiten* - *Mikrocode bearbeiten*. Es erscheint ein Fenster wie in Abb. 5.12. Wir erkennen sofort unsere Mikrocodetabelle für den HOL-Befehl. Warum sie hier gleich doppelt erscheint, darauf werden wir im nächsten Abschnitt noch ausführlich zu sprechen kommen. Für uns ist nur die linke Tabelle interessant; die rechte wird automatisch synchronisiert, solange bei "Mikrocode flagabhängig" kein Häkchen gesetzt ist.

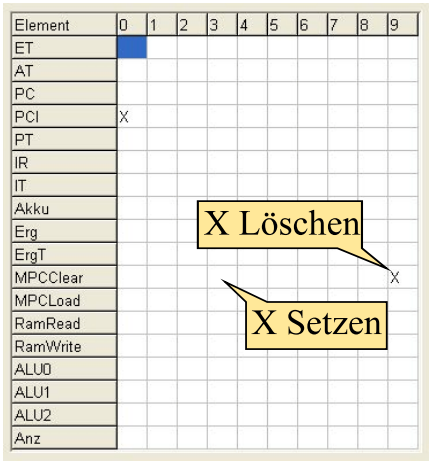


Abb. 5.13

Zunächst wollen wir einen bereits vorhandenen Befehl abändern, und zwar den Befehl VZG mit dem Code 03. Dieser Befehl besitzt – im Gegensatz zu MiniPC – keine Parameter; die zeitliche Verzögerung ist bei MikroPC immer gleich. Sie wird hier allein dadurch erreicht, dass der Prozessor alle 10 Mikroschritte durchführen muss, bis er auf MPCClear stößt. Dabei geschieht in den Mi-

kroschritten 1 bis 8 rein gar nichts (Abb. 5.13). Diese Wartezeit soll nun verkürzt werden. Dazu muss das X bei MPCClear in der Spalte 9 zunächst gelöscht und eine neues X in die Spalte 3 derselben Zeile gesetzt werden. Dies erreicht man, indem man mit der linken Maustaste in die entsprechende Felder klickt.

Damit der neue Mikrocode übernommen wird, muss zuerst die Schaltfläche *Übernehmen* betätigt werden; MikroPC merkt sich die neue Mikroschritttabelle, speichert sie aber noch nicht ab. Erst wenn man das Fenster mit der OK-Schaltfläche schließt, werden die Mikroschritttabellen sämtlicher Befehle dauerhaft im Arbeitsverzeichnis von MikroPC gespeichert. Die entsprechende Datei hat den Namen "mikrocode.dat".

An dieser Stelle muss noch einmal ganz deutlich darauf hingewiesen werden, dass durch die Manipulation am Mikrocode die Eigenschaften des Prozessors selbst (genauer seines ROMs) geändert werden. Alle Programme, in denen der VZG-Befehl auftaucht, werden jetzt ein klein wenig schneller laufen.

Weist der geänderte Mikrocode Fehler auf, so kann dies dazu führen, dass Programme nicht mehr korrekt laufen. Deswegen findet man im Verzeichnis MicroPCProg eine Kopie der Original-Mikrocode-Datei mit dem Namen Mikrocode0.dat. Sollte es einmal nicht gelingen, eine fehlerhafte Mikrocode-Datei zu reparieren, kann man diese einfach unter dem Namen Mikrocode.dat in das Arbeitsverzeichnis von MikroPC kopieren.

Nun wollen wir noch erklären, wie man selbst einen neuen Befehl herstellen kann. Es soll ein Befehl erzeugt werden, welcher den Inhalt einer RAM-Zelle direkt zur Anzeige bringt. Dieser Befehl soll

ZGR xxx (*Z*eige *R*AM-Zelle)

lauten und den Code 25 haben. Dazu stellen wir zunächst im Mikrocode-Editor (Abb. 5.14) den Code auf 25 ein. Standardmäßig wird zunächst die Mikroschritttabelle für den NOP-Befehl angezeigt. Nun geben wir neben dem Code als Mnemonic *ZGR* ein und löschen dann mit der *Löschen*-Schaltfläche die Einträge in der Mikroschritttabelle. Anschließend klicken wir die entsprechenden Zellen wie in Abb. 5.14 an. Wie schon beim Ändern des VZG-Befehls kümmern wir uns dabei nicht um die rechte Tabelle. Am Schluss werden die Schaltflächen *Übernehmen* und *OK* betätigt; dadurch wird der neue Befehl in der Datei Mikrocode.dat gespeichert und steht auch nach einem Neustart des Programms zur Verfügung. So hat der Prozessor von MikroPC einen neuen Befehl erhalten.

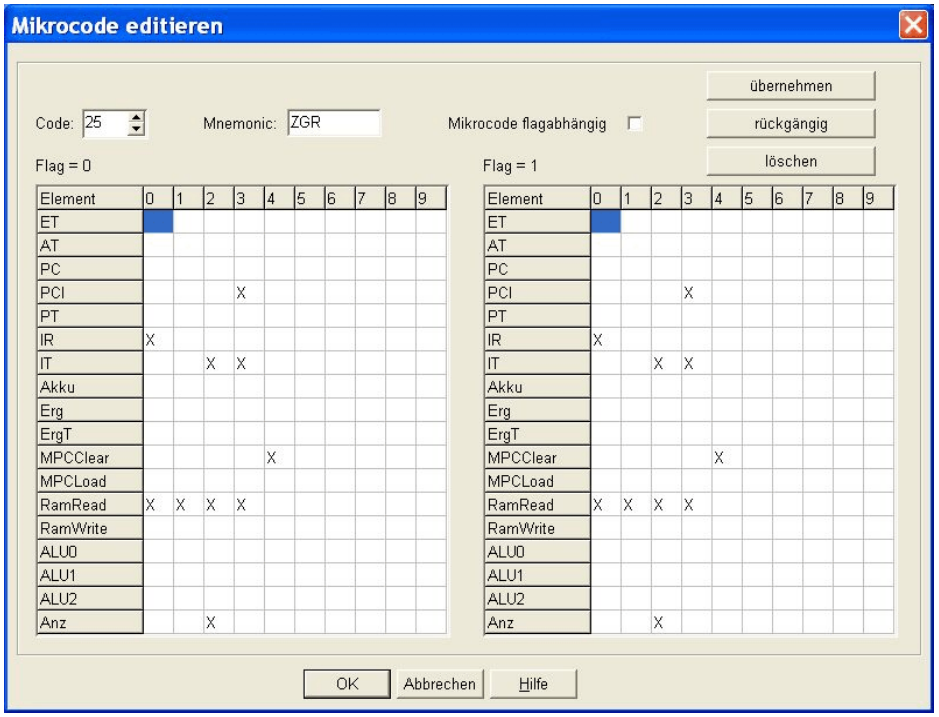


Abb. 5.14

Das sollte man gleich testen: In die Zelle 000 des RAMs schreiben wir den Befehl 25.010 und in die Zelle 010 schreiben wir das Datum 00.123. Kurz nach Betätigen der Start-Schaltfläche steht auch schon der Inhalt der Zelle 010 im Display (Abb. 5.15).

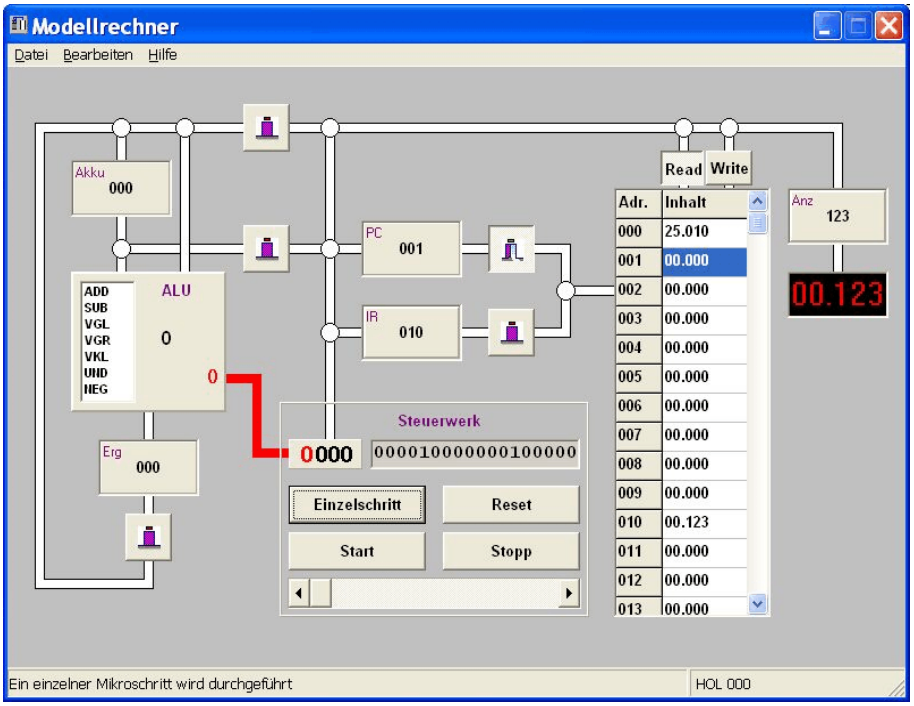


Abb. 5.15

5.6 Sprünge und Verzweigungen

Bislang wurden die einzelnen Befehle genau in der Reihenfolge ausgeführt, wie sie im RAM stehen. Durch Sprungbefehle kann diese Reihefolge durchbrochen werden. Widmen wir uns zunächst den unbedingten Sprüngen: SPU xxx, springe direkt zu dem Befehl in Zelle xxx. Die Idee zur Konstruktion der entsprechenden Mikrocodetabelle ist einfach: Der Operand xxx muss in den Programmzähler geladen werden. Dabei muss dieser Wert erst im Akku zwischengespeichert werden. Bei einem direkten Laden aus dem RAM ohne den Umweg über den Akku würde sich wegen des geöffneten PTs schon während des Ladevorgangs die aktuelle Zelladresse ändern. Am Schluss würde also nicht der Operand unseres SPU-Befehls im PC stehen, sondern der einer anderen Zelle. Damit der alte Inhalt des Akkus bei dieser Aktion nicht verloren geht, wird er zuvor im Ergebnisregister zwischengespeichert und am Schluss über das Ergebnistor wieder in den Akku geladen.

Element	0	1	2	3	4	5	6	7	8	9
ET			X	X						
AT					X	X				
PC					X					
PCI										
PT			X	X						
IR										
IT										
Akku			X				X			
Erg	X									
ErgT							X	X		
MPCClear									X	
MPCLoad										
RamRead			X	X						
RamWrite										
ALU0	X	X								
ALU1										
ALU2										
Anz										

Abb. 5.16

Bei bedingten Sprüngen (Verzweigungen) hängt der Programmablauf von dem Wert des Flags ab: Dieses Register kann nur die Werte 0 und 1 annehmen; welchen Wert es gerade besitzt, hängt von der letzten Operation der ALU ab. Hat diese zum Beispiel bei dem Befehl VGL xxx festgestellt, dass der Inhalt von xxx und dem Akku gleich sind, wird das Flag auf 1 gesetzt, ansonsten auf 0.

Wie lässt sich diese Flag-Abhängigkeit im Mikrocode realisieren? Dies schauen wir uns für den bedingten Sprung SPB genauer an: Bei dem Befehl SPB yyy (Code: 11) springt der Ablauf zur Adresse yyy, wenn das Flag auf 1 gesetzt ist, ansonsten wird wie gewöhnlich mit dem nächsten Befehl fortgefahren. Um eine solche Verzweigung im Ablauf zu realisieren, bedient man sich

folgenden Tricks: Die Mikrocodetabelle für diesen Befehl gibt es nicht nur einmal, sondern gleich in doppelter Ausführung. Je nach Wert im Flag wird auf die eine oder die andere Tabelle zurückgegriffen; dadurch kann man ein je nach Flag-Wert unterschiedliches Verhalten des Prozessors erhalten - die Verzweigung ist erreicht!

Bei den Befehlen, welche nicht vom Wert des Flags abhängen (sollen), wie z. B. LDA oder ADD, stimmen die beiden Varianten der Mikrocodetabelle überein. Dann ist es egal, welchen Wert das Flag hat; es werden die gleichen Schritte ausgeführt.

Der Einfachheit halber werden alle diese Tabellen in einem einzigen ROM hintereinander geschrieben. Die Zelladresse wird jetzt 4-stellig: die F-Stelle (vgl. Abb. 5.9) bestimmt, auf welche **Variante** der Mikroschritttabelle zugegriffen wird. Sie ist direkt mit dem Flag-Register verbunden. Wenn der Flag-Wert also 0 ist, werden die Mikroschritte 0110, 0111, 0112, ... ausgeführt; wenn der Flag-Wert 1 ist, werden die Mikroschritte 1110, 1111, 1112, ... ausgeführt.

In der ersten Variante des SPD-Befehls (F = 0) wird lediglich der PC um 1 erhöht (und am Schluss natürlich der MPC gelöscht). In der zweiten Variante (F = 1) stehen dieselben Mikroschritte wie beim unbedingten Sprung SPU. Um bei MikroPC flagabhängige Mikroschritte einzugeben, muss man zunächst das Häkchen bei “Mikrocode flagabhängig” setzen. Dadurch wird die automatische Synchronisierfunktion zwischen den beiden Varianten abgestellt.

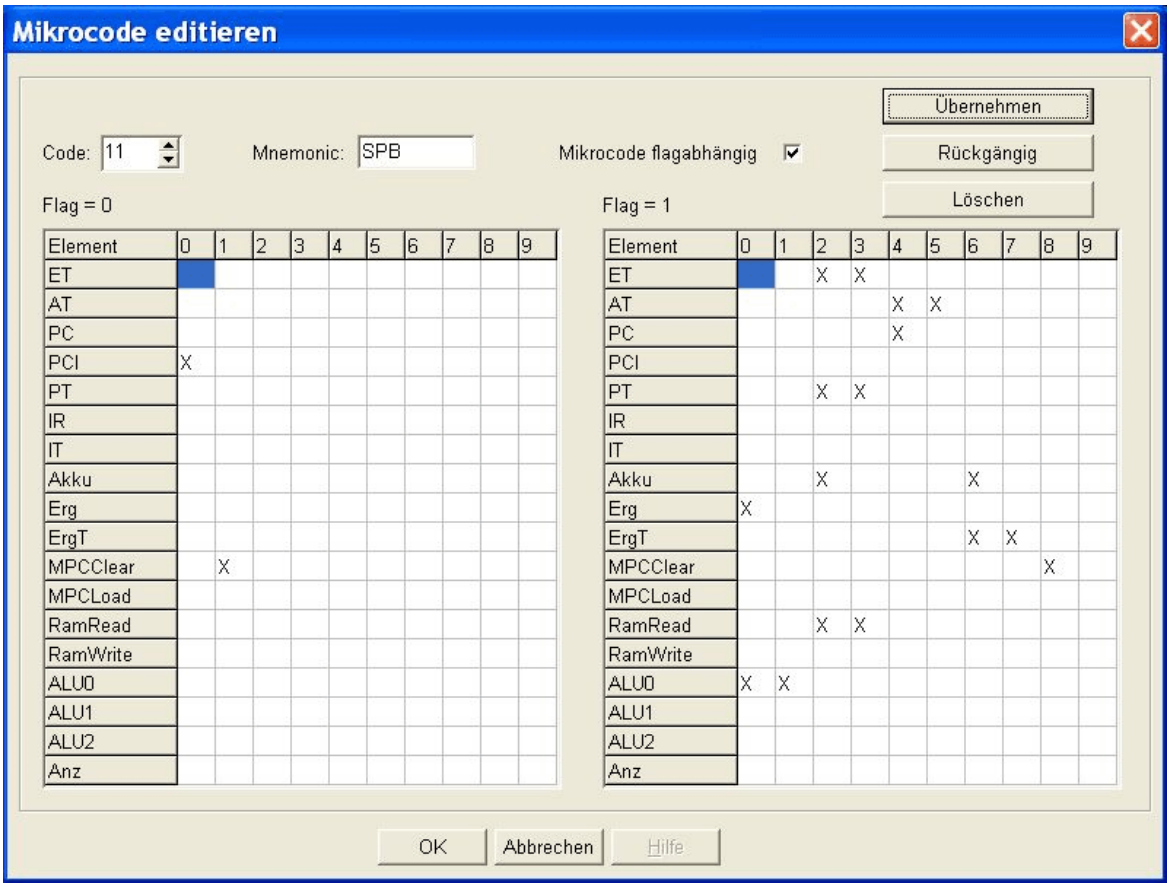


Abb. 5.17

Aufgaben

5.6 Schreibe den Mikrocode für den SKIP-Befehl: Er soll das Mnemonic SKP und den Code 26 haben. Dieser SKIP-Befehl soll dafür sorgen, dass die folgende Zelle im Programmablauf übersprungen wird; auf diese Weise kann man Zahlen auch zwischen den Befehlen speichern. Teste den neuen Befehl auch aus.

5.7 Schreibe den Mikrocode für den Akku-Mal-2-Befehl. Er soll das Mnemonic AM2 und den Code 27 haben. Dieser Befehl soll den Wert des Akkus mit 2 multiplizieren und das Ergebnis wieder im Akku abspeichern. Teste den neuen Befehl auch aus.

5.8 Schreibe den Mikrocode für einen bedingten SKIP-Befehl (SKB) und teste ihn aus.

5.9 Die Datei "Erstes_Programm0.obj" aus dem Kapitel 3 besteht nur aus wenigen Befehlen (und drei Daten in den Zellen 100, 101 und 102). Bestimme allein mit Hilfe der Mikrocodetabellen (s. Anhang), welche Zeit das Programm benötigt, bis es auf den HOL-Befehl 00.000 in Zelle 006 stößt. Geh dabei von einer Taktfrequenz von 1 Hz aus.

Gehe folgendermaßen vor:

- Begründe zunächst, dass man für die HOL-Phase zeitmäßig nur einen Takt veranschlagen darf.
- Berechne dann die gesamte benötigte Zeit.
- Kontrolliere das Ergebnis mit Hilfe von MikroPC.

5.10 Welche Funktion hat der Befehl HWF aus Abb. 5.18?

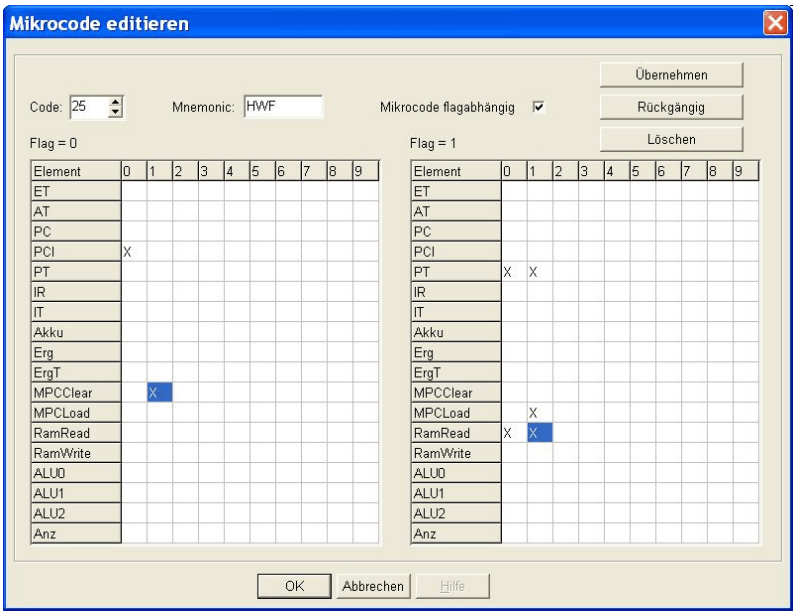


Abb. 5.18

6. Assembler

Die Entwicklung eines Programms lässt sich in verschiedene Phasen einteilen. Zunächst wird das Problem analysiert und eine Lösung umgangssprachlich oder mit einem Flussdiagramm dargestellt. In einem nächsten Schritt wird ein Programm mit Hilfe von Mnemonics hergestellt. Zuletzt muss dieses Programm noch in die zugehörigen Zahlencodes umgesetzt werden. Diesen letzten Schritt kann man auch von speziellen Übersetzungsprogrammen, den so genannten **Assemblern**, durchführen lassen.

Assembler nehmen uns nicht nur lästige Übersetzungsarbeit ab; sie besitzen einige zusätzliche Eigenschaften, die das Erstellen, das Ergänzen und Verbessern von Programmen erleichtern. Ohne Assembler ist es beispielsweise schon sehr mühselig, auch nur einen einzigen Befehl in ein Programm einzufügen: In diesem Fall müssen nämlich nicht nur alle folgenden Befehle im RAM verschoben werden, es müssen auch eine Reihe von Adressangaben (insbesondere bei den Sprüngen) angepasst werden. Wenn man einen Assembler benutzt, ist dies kein Problem mehr.

6.1 Assemblercode

Unser MiniPC-Programm besitzt einen einfachen Assembler. Damit der Assembler den Übersetzungsvorgang erfolgreich durchführen kann, muss man sich bei der Eingabe der Mnemonic-Codes an bestimmte Regeln halten. Der Quelltext, der diese Assembler-Regeln berücksichtigt, heißt **Assemblercode**.

An einem einfachen Beispiel sollen die Regeln unseres Assemblers erklärt werden. Ziel soll es sein, eine Sekunden-Uhr zu programmieren: Unser MiniPC soll im Sekundentakt von 00 bis 59 zählen, dann wieder auf 00 springen usw. Das Flussdiagramm dazu ist in Abb. 6.1 wiedergegeben. Wir sehen: In einer inneren Schleife wird ein Zähler immer wieder um 1 erhöht, bis die Zahl 59 erreicht ist; diese Schleife dauert eine Minute. Diese Minuten-Befehlsfolge wird in einer äußeren Schleife immer wieder wiederholt. Auf diese Weise wird in einer Endlosschleife Minute nach Minute durchgezählt, bis die STOP-Schaltfläche betätigt wird.

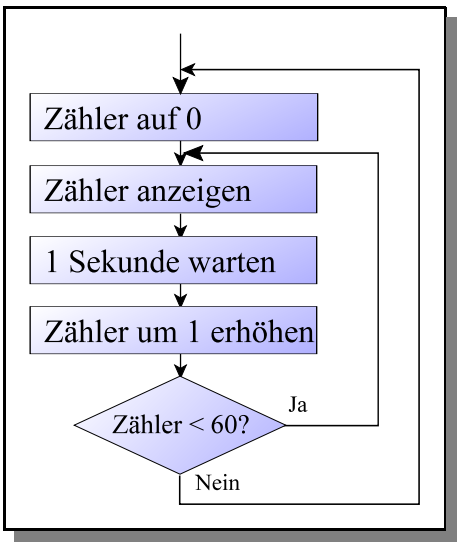


Abb. 6.1: Sekunden-Uhr

Der Assemblercode für unsere Sekunden-Uhr kann so aussehen (uhr1.asm):

```
//Sekundenuhr
var:
        max=60              //Anzahl der Sekunden in 1 Minute
        eins=1              //Inkrement
ende

konst:
        startwert=0         //Startwert für die Sekundenanzeige
ende

progr:
As:     AKO startwert       //Akku auf 0 stellen (Zähler)
Is:     ANZ                 //Anzeigen
        VZG 250             //1 Sekunde warten
        VZG 250
        VZG 250
        VZG 250
        ADD eins            //Akkuwert um 1 erhöhen
        VKL max             //Akkuwert kleiner als 60?
        SPB Is              //Wenn ja, springe zur Marke Is
        SPU As              //Springe zur Marke As
ende
```

Als erstes fällt die Dreigliederung auf: Der erste Teil wird eingeleitet durch var: und beendet durch ende. Der zweite Teil beginnt mit konst: und hört wieder auf mit ende. Der letzte Teil bildet das eigentliche Programm; er wird durch progr: und ende eingeschlossen. **Kommentare** werden durch ein Doppelslash (//) eingeleitet; sie werden beim Übersetzungsvorgang übersprungen. Die Reihenfolge der Teile muss immer var - konst - progr sein. Der erste, der zweite oder auch die ersten beiden Teile können fehlen, der Programmteil hingegen nicht.

Betrachten wir die einzelnen Teile genauer. Im ersten Teil findet die so genannte **Variablendeklaration** statt. Hier werden Speicherplätze für Zahlen reserviert und mit einem Namen verknüpft. Unser Assembler wählt für die erste auftauchende Variable immer die Zelle 127, für die nächste die Zelle 126 usw. Bei allen Befehlen, welche eine Zelladresse als Operand benötigen, kann ein solcher Variablenname stellvertretend für die Zelladresse benutzt werden.

Jedesmal, wenn der Assembler nun beim Übersetzungsvorgang z. B. auf den Bezeichner “max” stößt, ersetzt er ihn automatisch durch die Adresse 127. Wählt man die Variablennamen sinnvoll, kann die Lesbarkeit des Programms dadurch wesentlich erleichtert werden.

Der rechts vom Gleichheitszeichen stehende Wert wird vom Assembler bei der Reservierung in die Speicherzelle eingetragen. (Lässt man bei der Deklaration die Zuweisung =xxx weg, wird bei der Reservierung kein Wert in die Zelle eingetragen.) Dieser Wert bildet nur einen **Start-**

wert; er kann beim Programmablauf überschrieben werden.

Ähnliches gilt für den zweiten Teil, die **Konstantendeklaration**. Konstanten stehen allerdings nicht für Zelladressen, sondern für Daten, die von Befehlen wie AKO oder VZG direkt verarbeitet werden. Konstanten stellen also feste Zahlen dar; die Zuweisung bei der Deklaration darf deswegen nicht fehlen. Wenn der Assembler unser Programm übersetzt, wird z. B. "startwert" durch die Zahl 0 ersetzt. Der Einsatz einer Konstanten ist insbesondere dann besonders wertvoll, wenn sie im Programm mehrfach auftaucht. Will man deren Wert abändern, muss dies nur im Deklarationsteil geschehen.

Widmen wir uns nun dem Programmteil: Er besteht aus den bereits bekannten Mnemonics, Daten, Variablen, Konstanten und den so genannten **Marken**. Diese Marken enden immer mit einem Doppelpunkt; sie stehen vor einem Befehl und kennzeichnen dessen Position im Programm. Genauer gesagt stehen sie stellvertretend für die Adresse der Zelle, in welcher sich der hinter der Marke stehende Befehl befindet. Statt nun bei einem Sprungbefehl die Zieladresse selbst einzugeben, benutzt man den Namen der Marke, zu der gesprungen werden soll. Der Assembler-Code SPU As in unserem Beispiel wird vom Assembler z. B. so übersetzt, dass er einen Sprung zum Befehl AKO … veranlasst.

Entscheidend ist, dass man Sprungadressen nun nicht mehr abändern muss, wenn einzelne Zeilen in den Assemblercode eingefügt werden. Bei jedem Übersetzungsvorgang (Assemblieren) berechnet der Assembler diese Sprungadressen nämlich automatisch neu.

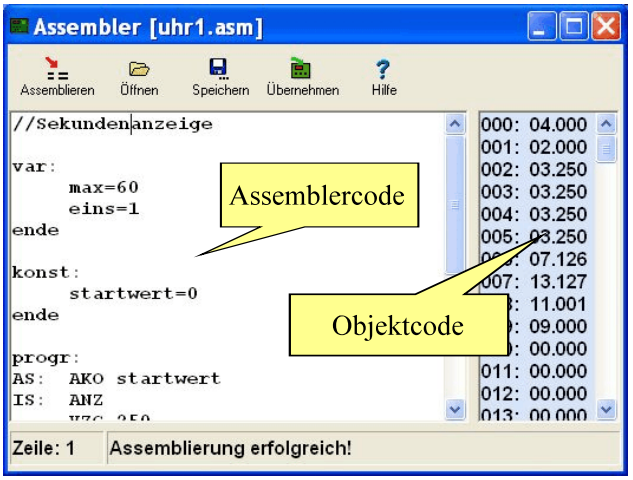


Abb. 6.2

6.2 Bedienung des MiniPC-Assemblers

Wir wissen nun, wie der Assembler arbeitet. Es bleibt zu erklären, wie man den MiniPC-Assembler bedient. Zunächst öffnen wir das Assemblerfenster mit Bearbeiten - Assembler (Abb. 6.2). In den Assemblercode-Bereich geben wir den Assemblercode ein. Dabei rückt man die Variablen, die Konstanten und die Mnemonics mit Hilfe der TAB-Taste (Tabulator) ein; die

Schlüsselwörter var:, konst:, progr: und ende sowie sämtliche Marken werden linksbündig geschrieben. Das ist nicht zwingend erforderlich; der Assembler könnte auch ohne diese Formatierung arbeiten, ja man könnte sogar auch auf die Leerzeichen zwischen Operator und Operand verzichten. Da durch das Einrücken das Programm für den Menschen aber übersichtlicher wird, sollte man auf jeden Fall die Tabulatoren benutzen.

Nach der Eingabe des Quelltextes sollte man das Programm zunächst abspeichern. Dazu benutzen wir die Schaltfläche *Speichern unter...*. Die Endung dieser Datei ist "asm". Danach wird die Schaltfläche *Assemblieren* betätigt. Ist der Assemblercode fehlerfrei, erscheinen die Zahlencodes der entsprechenden Befehle im rechten Bereich. Diese Folge von Codes wird auch **Objektcode** genannt. Ist der Assemblercode noch mit Fehlern behaftet, so erhält man eine Fehlermeldung in der Statuszeile am unteren Rande.

Der Objektcode unserer Sekundenuhr ist in Abb. 6.3 wiedergegeben. In den Zellen 000 bis 009 steht der Programmcode. In den letzten beiden Zellen erkennt man die Werte der Variablen max und eins.

Mit der *Übertragen*-Schaltfläche überträgt unser Assembler-Programm den Objektcode in den Speicher unseres MiniPCs. Allerdings kann das Programm erst dann gestartet werden, wenn das Assembler-Fenster geschlossen worden ist. Keine Sorge: Solange MiniPC nicht beendet wird, merkt sich auch der Assembler den Quelltext und zeigt ihn sofort an, wenn man ihn wieder aktiviert.

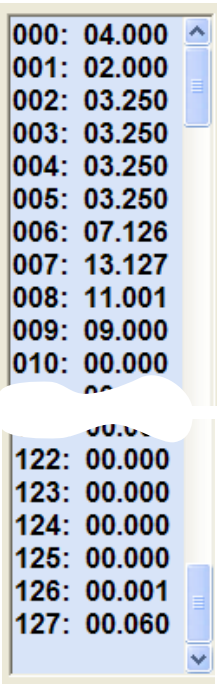


Abb. 6.3

6.3 Dividieren mit Assembler

Nun, da wir in dem Assembler ein mächtiges Werkzeug in den Händen haben, können wir auch kompliziertere Aufgaben in Angriff nehmen, ohne Angst davor haben müssen, den Überblick zu verlieren. So wollen wir nun ein Programm schreiben, welches zwei Zahlen dividieren kann - eine Aufgabe, welche in der Grundschule immerhin erst in der 3. oder 4. Klasse gelernt wird.

Die Idee ist eigentlich ganz einfach: Wir subtrahieren den Divisor vom Dividenden; von dem Rest subtrahieren wir noch einmal den Divisor usw., bis der Rest schließlich kleiner als der Divisor ist. Dabei zählen wir, wie oft wir diese Subtraktionen durchführen. Diese Anzahl gibt gerade den Quozienten, das Ergebnis der Division, an. Teilen wir z. B. auf diese Weise 42 durch 8, dann subtrahieren wir zunächst die Zahl 8 vom Dividenden 42; übrig bleibt der Rest 34. Davon subtrahieren wir wieder den Divisor 8 usw. Nach 5 solcher Subtraktionen hört die Rechnung auf, weil der Rest dann gleich 2 und somit kleiner als der Divisor 8 ist. Das Ergebnis der Rechnung ist damit 5 (Rest 2).

Die Vorgehensweise können wir folgendermaßen mit Variablen beschreiben:

Version 1:

```
setze zähler auf 0
solange dividend>divisor ist, rechne:
        ersetze dividend durch dividend-divisor
        ersetze zähler durch zähler+1
zeige zähler im Display
```

Das Ergebnis der Rechnung dividend - divisor wird hier wieder in der Zelle der Variablen dividend gespeichert werden. Da der Wert des Dividenden nach der ersten Subtraktion selbst nicht mehr gebraucht wird, kann in dieser Variablen auch der Rest abgelegt werden.

Diese Folge von Anweisungen formen wir nun schrittweise um, bis gültiger Assemblercode entstanden ist.

Version 2:

```
            setze zähler auf 0
Schleife:   wenn dividend<divisor, dann springe zur Ausgabe (zähler bleibt 0)
            ersetze dividend durch dividend - divisor
            ersetze zähler durch zähler + 1
            springe zu schleife
Ausgabe:    Zeige zähler
            springe zur Ausgabe
```

Version 3:

```
            setze zähler auf 0
Schleife:   prüfe, ob dividend<divisor
            springe bedingt zur Ausgabe (zähler bleibt 0)
            lade dividend in Akku
            subtrahiere divisor von Akku
            speichere Akku in dividend
            lade 1
            addiere zähler zu Akku
            speichere Akku in zähler
            springe unbedingt zu Schleife
Ausgabe:    lade zähler in Akku
            zeige Akku
            springe unbedingt zur Ausgabe
```

Jetzt ersetzen wir noch die zweite Zeile durch die folgenden beiden Zeilen:

```
Schleife:   Lade dividend in Akku
            ist Akku kleiner als divisor?
```

Zuletzt ersetzen wir den ausführlichen Text noch durch Mnemonics und erhalten (division0.asm):

```
var:
            dividend=100
            divisor=12
            zähler                  //liefert Quotient
ende

progr:
            AKO 0                   //Startwert für den Zähler ist 0
            ABS zähler
schleife:   LDA dividend
            VKL divisor
            SPB ausgabe
            LDA dividend
            SUB divisor
            ABS dividend
            AKO 1
            ADD zähler
            ABS zähler
            LDA dividend
            VKL divisor
            SPB ausgabe
            SPU schleife
Ausgabe:    LDA zähler
            ANZ
            SPU Ausgabe
ende
```

6.4 Aufgaben

6.4.1 Gib den Assemblercode des Divisionsprogramms ein und teste es aus.

6.4.2 Startet man das Programm aus Aufgabe 1 erneut, nachdem man den ersten Programmlauf mit der STOP-Schaltfläche abgebrochen hat, mit **000 PC - START**, wird die Rechnung nun nicht mehr korrekt durchgeführt. Woran liegt das? Korrigiere den

Assemblercode entsprechend! Teste das Programm auch mit anderen Dividenden und Divisoren.

6.4.3 Ergänze das Programm aus Aufgabe 2 so, dass es nach dem (ganzzahligen) Quotienten auch den Rest anzeigt.

6.4.4 Schreibe ein Multiplikationsprogramm (multiplikation0.asm) und teste es aus.

6.4.5 Etwas zum Staunen: Speichere den Objekt-Code von Aufgabe 6.4.1 bzw. 6.4.4 und starte das Programm mit MikroPC!

7. Steuern

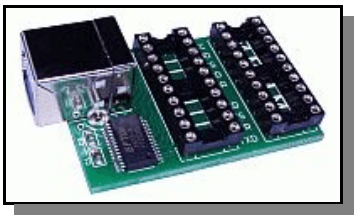


Abb. 7.1: AK-MODUL-BUS-Set mit FT232

Unser MiniPC kann auch zur Steuerung von kleinen Anlagen eingesetzt werden. Dazu können unterschiedliche Experimentier-Boards eingesetzt werden, z. B. die COM-Platine von AK-Modul-Bus (Abb. 7.1) oder ein Breadboard mit einem FT232-Modul (Abb. 7.2). In beiden Fällen wird ein FT232-Chip verwendet; dieser wandelt die USB-Signale in TTL-Signale um (mehr dazu im Abschnitt 7.3). Alternativ kann man statt eines FT232-Moduls auch den USB-TTL-Wandler CP2102 benutzen. Alle diese Module werden über ein Mini-USB-Kabel mit dem PC verbunden. Bezugsquellen und Hinweise zu weiteren Anschlussmöglichkeiten sind im Anhang zu finden.

Im Folgenden werden die Experimente am Beispiel eines FT2323-Boards (aus Abb. 7.1 oder 7.2) behandeln. Das FT232-Modul in Abb. 7.2 kostet z. B. nur wenige Euro; obendrein werden bei diesem Modul die benötigten Treiber automatisch installiert.

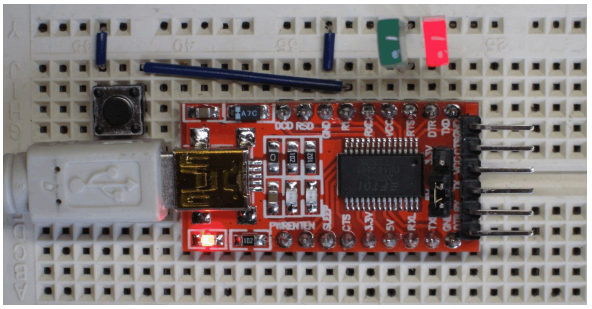


Abb. 7.2: Breadboard mit FT232-Modul

Zahlreiche interessante Versuche zum Steuern, Messen und zur Datenübertragung können damit durchgeführt werden; so kann man damit eine Fußgängerampel, einen einfachen Soundgenerator oder auch einen Lügendetektor bauen.

Manchmal kann es gefährlich sein, wenn man selbst gebastelte Geräte an den PC anschließt: Die Bauteile oder sogar der PC können Schaden nehmen. Bei den hier vorgestellten Versuchen kann dies nicht geschehen, solange man sich an diese Anweisungen hält:

Für unsere Versuche dürfen nur die oben genannten USB-TTL-Wandler benutzt werden. Es dürfen keine elektrischen Quellen wie Akkus, Batterien oder Netzgeräte, erst recht nicht die Steckdose benutzt werden!

Für usere Experimente benötigen wir noch einige Bauteile: eine grüne und eine rote LED, einen Taster, einen LDR, eine kleinen billigen Kopfhörer sowie einige jumper-Kabel. Diese Bauteile werden je nach Bedarf in die Kontakte der Platine gesteckt. Wenn das Board mit dem PC verbunden ist, kann man mit unserem MiniPC-Programm z. B. die Leuchtdiode in Abb. 7.2 ein- und ausschalten; der Rechner liefert auch den für den Betrieb der Leuchtdiode erforderlichen Strom. Geräte, die vom Computer geschaltet werden, bezeichnet man als **Aktoren**. Weitere Aktoren sind Motoren, Kopfhörer, Glühbirnen oder Elektromagnete. Allerdings ist die von unserem FT232-Modul gelieferte Stromstärke recht gering (max. 20 mA); daher können nicht alle möglichen Aktoren damit betrieben werden.

Der Computer kann aber auch in Erfahrung bringen, ob ein Schalter auf der Platine geschlossen ist oder gerade viel Licht auf einen lichtempfindlichen Widerstand trifft. Solche Geräte, welche dem Computer Informationen über Helligkeiten, Temperaturen oder Schalterzustände liefern, bezeichnet man als **Sensoren**.

7.1 Port-Befehle

Aktoren und Sensoren sind nicht Bestandteile von unserem MiniPC; sie werden deswegen als **externe Geräte** bzw. **Peripherie-Bausteine** bezeichnet. Um mit diesen Geräten Informationen auszutauschen, besitzt MiniPC Portbausteine, welche ebenfalls an den externen Datenbus angeschlossen sind. Diese **Ports** sind Speicher ähnlich dem Anzeige-Register, welches wir schon in Kapitel 5 kennen gelernt haben. Sie lassen sich über Steuerleitungen beeinflussen. Diese Ports sind aber auch mit den Signalleitungen unserer Schnittstelle, dem FT232-Modul, verbunden. Auf diese Weise können Daten vom Akku zu einem angeschlossenen Aktor und umgekehrt auch von einem Sensor zum Akku transportiert werden. Im Gegensatz zum Anzeige-Register haben unsere Ausgangs- und Eingangsports aber nicht 8, sondern jeweils nur 1 Bit.

Es stehen drei Ausgangsports (nummeriert von 0 bis 2) zur Verfügung. Um einen Aktor am Ausgangsport x ein- oder auszuschalten, benutzt man den Befehl

CPA 00x (*C*OM-*P*latine-*A*usgang)

Hierdurch wird der Ausgangspegel auf 5 V gesetzt und das Gerät eingeschaltet, wenn der Akku den Wert 1 hat; steht im Akku gerade eine 0, wird der Pegel auf 0 V gesetzt und das Gerät ausgeschaltet. Bei der Benutzung des CPA-Befehls darf im Akku also nur eine 1 oder 0 stehen; ansonsten liefert das MiniPC-System eine Fehlermeldung.

Umgekehrt kann man den Zustand eines Schalters am Eingang 00x in den Akku laden. Dazu wird der Befehl

CPE 00x (*C*OM-*P*latine-*E*ingang)

benutzt. Ist der Schalter offen, wird eine 1(!) im Akku abgelegt, ansonsten eine 0. Statt des Schalters können auch veränderliche Widerstände eingesetzt werden. Ein offener Schalter entspricht dann einem hohen Widerstand. Im Kapitel 8 findet man mehr dazu.

7.2 Blinken von Leuchtdioden

Jetzt ist es soweit: Wir wollen eine Leuchtdiode (kurz: LED) mit MiniPC blinken lassen. Dazu stecken wir zunächst die Leuchtdiode auf das Experimentierboard: Der längere Anschluss, die Anode, wird über den Anschluss DTR des FTD232-Moduls mit dem Ausgang A2 des MiniPC verbunden, der längere, die Kathode, wird mit dem GND-Anschluss des FT232-Moduls verbunden. Generell wird ein Aktor immer mit einem Anschluss an einen Ausgang und mit dem anderen an Masse angeschlossen. **Der Jumper am FT232-Modul wird auf 5 V gesteckt.**

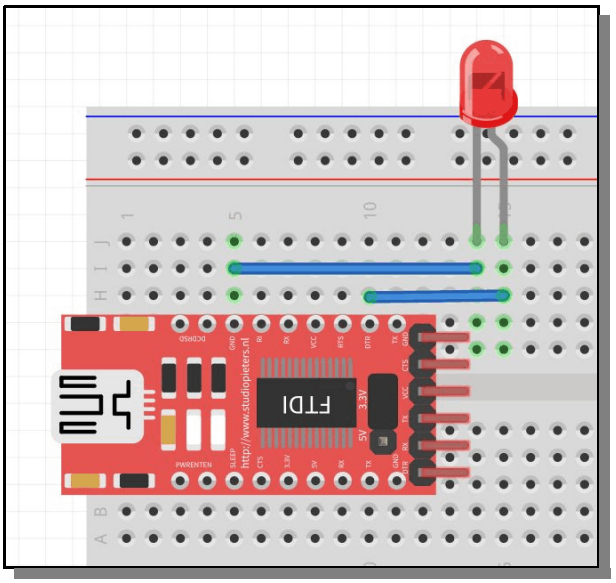


Abb. 7.3: Blinklichtschaltung

Nun wird das FT232-Modul über ein Mini-USB-Kabel an den PC angeschlossen. Nach dem Anschließen blinkt die LED ein paar mal auf und leuchtet dann dauerhaft. Grund dafür ist: Am Ende der Initialisierung besitzen alle Ausgänge des FT232-Moduls einen Pegel von 5 V. Falls erforderlich können wir sie am Anfang eines Programms mit dem PCA-Befehl auf einen Pegel von 0 V bringen.

Als nächstes aktivieren wir im MiniPC-Programm unsere Schnittstelle. Beim ersten Verbinden des FT232-Moduls wurden automatisch Treiber installiert und dem Port eine Com-Nummer zugewiesen. Dieses Com-Nummer lässt sich leicht mit Hilfe des Geräte-Managers herausfinden: Starten Sie den Geräte-Manager; es erscheint ein Fenster mit einer in Gruppen geordneten Liste der zur Verfügung stehenden Geräte. Hier klicken sie die Gruppe “Anschlüsse (Com und LPT)” an. In dieser Gruppe findet man unser FT232-Modul unter der Bezeichnung “USB Serial Port”. Dahinter ist in Klammern die COM-Nummer angegeben, z. B. COM1. In diesem Fall wählen wir im *Bearbeiten-Menü* von MiniPC die Option *Schnittstelle → COM1* (vgl. Abb. 7.4). Im Normalfall wird nun rechts vom Display ein grünes COM1-Feld angezeigt; dies ist das Zeichen dafür, dass die Verbindung zum COM1-Anschluss erfolgreich hergestellt werden konnte. Ist dieses Feld aber rot, ist ein Fehler aufgetreten: Möglicherweise gibt es den benutzten Anschluss nicht oder er wird gerade schon von einem anderen Programm benutzt. Dann sollte man einen anderen COM-Anschluss benutzen oder gegebenenfalls das andere Programm erst schließen und daraufhin beim MiniPC-Programm die COM-Schnittstelle erneut aktivieren (Weitere Informationen dazu im Anhang). Zu beachten ist noch, dass die Standardeinstellung *TTL* (rechts neben der COM-Anzeige) hier nicht verändert werden darf (vgl. Abb. 7.4).

Abb. 7.4

Nach all diesen Vorbereitungen ist das Schreiben des passenden Programms fast schon eine Kleinigkeit. Der folgende Assemblercode lässt die LED immer wieder jede viertel Sekunde ein- und ausschalten, bis das Programm mit der STOP-Schaltfläche angehalten wird (blinken1.asm).

```
//Blinklicht: Periode = 2*Blinkzeit
//LED an A2

konst:
        an=1
        aus=0
        blinkzeit=250
        ausgang=2           //A2
ende

progr:
Blink:  AKO an              //Einschalten vorbereiten
        CPA ausgang         //LED am Ausgang 2 einschalten
        VZG blinkzeit       //250 ms warten
        AKO aus             //Ausschalten vorbereiten
        CPA ausgang         //LED am Ausgang 2 ausschalten
        VZG blinkzeit       //250 ms warten
        SPU Blink           //springe zur Marke Blink
ende
```

7.3 Serielle Datenübertragung: RS232 und TTL

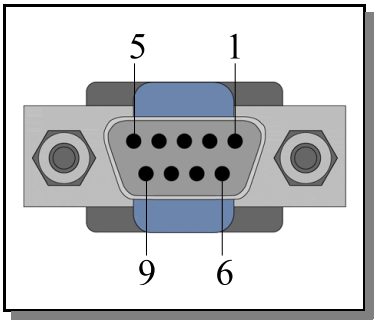


Abb. 7.5

Bei der von uns benutzten Schnittstelle handelt es sich um eine so genannte **COM-Schnittstelle**. Diese wurde in den 1960er Jahren entwickelt. Hiermit werden Daten seriell übertragen. Dabei werden über die beiden Leitungen TxD und RxD Datenbytes seriell gesendet bzw. empfangen. Seriell bedeutet: Die einzelnen Bits des Datenbytes werden nacheinander übertragen. Neben diesen beiden Datenleitungen (und der Masseleitung) gibt es noch eine Reihe von weiteren Leitungen, die zu Kontrollzwecken eingesetzt werden können. All diese Leitungen (mit Ausnahme von RxD) werden nun von MiniPC benutzt, um Aktoren zu steuern oder den Zustand von Sensoren abzufragen.
In der Abb. 7.5 sieht man die Pin-Belegung einer Buchse von der ursprünglichen COM-Schnittstelle.

In der folgenden Tabelle (auf der nächsten Seite) ist in den ersten beiden Spalten angegeben, welchem Ein- bzw. Ausgang von MiniPC welche COM-Datenleitung zugewiesen ist. In der zweiten Spalte ist neben der Bezeichnung auch die Bedeutung der einzelnen Leitungen im Rahmen der seriellen Datenübertragung angegeben. Da wir diese Leitungen aber anderweitig nutzen, gehen wir darauf nicht weiter ein. Die Namen dieser Leitungen sind allerdings für uns wichtig; denn sie finden sich auch an den Anschluss-Pins unseres FT232-Moduls.

Mini-PC Ein-/Ausgang	**COM-Bezeichnung**	**FT232**	**Bemerkung**
Aus (A1)	TxD (Transmit Data)	✔	
Ein (✖)	RxD (Receive Data)	(✔)	nicht von MiniPC unterstüzt
Aus (A0)	RTS (Request to Send)	✔	
Ein (E2)	CTS (Clear to Send)	✔	
	GND (Ground)	✔	
Ein (E0)	DCD (Data Carrier Detect)	✔	
Aus (A2)	DTR (Data Terminal Ready)	✔	
Ein (E3)	RI (Ring Indicator)	✔	
Ein (E1)	DSR (Data Set Ready)	✔	auf FT232 mit “RSD” bezeichnet

Die ursprüngliche COM-Schnittstelle und unser FT232-Modul benutzen die gleichen Daten- und Steuerleitungen, sie unterscheiden sich jedoch in ihren physikalischen Eigenschaften: Beim FT232-Modul wird mit den für den **TTL-Standard** typischen Pegeln 0 V (für den Bitwert 0) und 5 V (für den Bitwert 1) gearbeitet. Dagegen benutzt man bei der ursprünglichen COM-Schnittstelle die für den **RS232-Standard** typischen Pegel +12 V und - 12 V. Dabei unterscheiden sich die beiden Standards aber nicht nur durch die Beträge der Spannung, auch die Logik ist bei einigen RS232-Signalen umgekehrt. MiniPC kann mit beiden Standards arbeiten; dazu muss man nur neben der COM-Anzeige den benutzte Standard RS232 bzw. TTL auswählen.

Erwähnt sei noch: Auch wenn am Ausgang des FT232-Moduls ein Pegel von 5 V vorliegt, benötigen wir für die LED keinen Vorwiderstand. Bei der Belastung durch die LED sinkt nämlich durch die eingebaute Strombegrenzung beim Ausgang der Pegel auf etwa 2 V und die Stromstärke (m)einer LED liegt dann bei etwa 4 mA.

7.4 Sounds erzeugen

Mit Hilfe unseres Experimentierboards können wir MiniPC auch Töne über einen (passiven!) Buzzer erzeugen lassen. Ein Anschluss des Buzzers wird an Ausgang A2 angeschlossen, der andere wie üblich an GND. Die Verdrahtung ist praktisch genauso wie in Abb. 7.3; sollte ein Anschluss des Buzzers mit + gekennzeichnet sein, wird dieser an A2 angeschlossen.

Wie erzeugt jetzt der Buzzer einen Ton? Nun, wir setzen in schneller Folge das Ausgangsport auf 1, dann auf 0, dann wieder auf 1, danach auf 0 und so weiter. Dadurch wird – je nach Bauart – im Buzzer ein kleiner Elektromagnet oder ein Piezo immer wieder ein- und ausgeschaltet; auf diese Weise wird eine Membran zum Schwingen gebracht. Diese Schwingung nehmen wir als Ton wahr. Alternativ zum Buzzer kann man hier auch einen hochohmigen Kopfhörer einsetzen.

Als Programm lässt sich das Blinkprogramm aus dem vorletzten Abschnitt einsetzen; allerdings muss die "Blinkzeit" stark verkürzt werden, um einen Ton von 100 Hz zu erzeugen (sound100Hz.asm).

Die Soundqualität ist natürlich nicht erstklassig. Das liegt zum Teil an der Art der Tonerzeugung; diese bedingt, dass die Membran so genannte Rechteckschwingungen durchführt, deren Töne recht scharf, ja fast unangenehm klingen. Zum anderen stellt man aber auch unschöne Lücken im Ton fest. Diese kommen dadurch zustande, dass sowohl unser MiniPC-Programm als auch das Betriebssystem Windows zwischendurch Zeit benötigen, um andere Aufgaben zu erledigen. In diesen Phasen kommt es dann zu kleinen Aussetzern. Immerhin lassen sich aber durch Verändern der Schwingungsdauer unterschiedlich hohe Töne erzeugen. Auf diese Weise lässt sich z. B. ein Alarmsignal programmieren (alarm.asm).

```
//Alarm
//Ohrhörer an A2 (DTR) und Masse

var:
        zähler
        max
ende

konst:
        an=1
        aus=0
        zeit1=002
        zeit2=001
        ausgang=2
ende

progr:
Wdhlg:      AKO 50      //50 Schwingungen mit Schwingungsdauer 004 ms
            ABS max
            AKO 0
            ABS zähler
```

```
// Fortsetzung

Ton1:       AKO an
            CPA ausgang
            VZG zeit1
            AKO aus
            CPA ausgang
            VZG zeit1
            AKO 1
            ADD zähler
            ABS zähler
            VKL max
            SPB Ton1
            AKO 100      //100 Schwingungen mit Schwingungsdauer 002 ms
            ABS max
            AKO 0
            ABS zähler
Ton2:       AKO an
            CPA ausgang
            VZG zeit2
            AKO aus
            CPA ausgang
            VZG zeit2
            AKO 1
            ADD zähler
            ABS zähler
            VKL max
            SPB Ton2
            SPU Wdhlg
ende
```

7.5 Pulsweitenmodulation

Bislang haben wir unsere LED nur ein- und ausgeschaltet. Jetzt wollen wir eine LED mit unterschiedlicher Helligkeit leuchten lassen. Leider lässt sich die Spannung an den Ausgängen der seriellen Schnittstelle nicht verändern. Deswegen greifen wir auf ein Verfahren zurück, welches vor allem bei der Ansteuerung von Elektromotoren (z.B. bei Modelleisenbahnen) Anwendung findet: die so genannte **Pulsweitenmodulation**, kurz: **PWM**.

So kompliziert das Wort klingt, so einfach ist das Verfahren selbst. Die LED erhält nur kurze Stromimpulse. Wenn sie zeitlich sehr rasch erfolgen, dann kann das Auge die einzelnen Lichtblitze nicht mehr auflösen. Es nimmt vielmehr eine gemittelte Helligkeit war; diese ist um so höher, je weiter die Pulse im Zeitdiagramm (Abb. 7.6) sind.

In unserem Programm benutzen wir eine Frequenz von 100 Lichtblitzen pro Sekunde. Wenn dann ein Puls z. B. 3 ms lang dauert, ist die LED 30% der Zeit angeschaltet; in diesem Fall leuchtet sie recht schwach. Wenn der Puls hingegen 10 ms lang ist, dann leuchtet die LED permanent und entsprechend hell.

Das zugehörige Programm steht in der Datei “pwm.asm”.

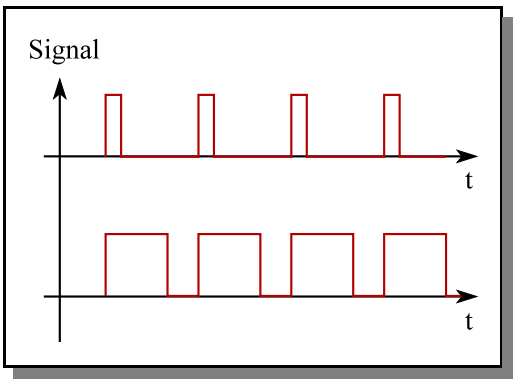


Abb. 7.6: Das untere Signal führt zu einer größeren Helligkeit.

```
//Pulsweitenmodulation
//LED an A2

konst:
        an = 1
        aus = 0
        ausgang = 2
        anzeit = 5
        auszeit = 2
ende

progr:

Blink:  AKO an
        CPA ausgang
        VZG anzeit
        AKO aus
        CPA ausgang
        VZG auszeit
        SPU Blink
ende
```

7.6 Aufgaben

7.6.1 Baue mit Hilfe einer grünen und einer roten LED eine Fußgängerampel. Schreibe ein Programm, welches beliebig viele Ampelzyklen durchläuft (fussgaengerampel.asm).

7.6.2 Erweitere das Programm aus 7.6.1 so, dass die Dauer der einzelnen Ampelphasen durch eine Variable im Assemblercode verändert werden können. Benutze dazu eine Zähl-

Schleife. Überlege zunächst: Warum kann man den VZG-Befehl nicht mit einer Variablen kombinieren? (Vgl. bedarfsampel2.asm)

7.6.3 Schreibe ein Programm, welches eine LED allmählich heller werden lässt. Vgl. auch Aufg. 7.6.2!

7.6.4 Ändere das Programm aus 7.6.1 so ab, dass die LEDs “sanft” ein- und ausgeschaltet werden.

8. Messen

Viele Fußgängerampeln schalten nur bei Bedarf um. Will ein Fußgänger die Straße überqueren, muss er zunächst einen Knopf drücken. Erst dann bekommen die Autos auf der Straße ein Rot-Signal und die Fußgängerampel springt von Rot auf Grün, um nach einer Weile wieder auf Rot zurückzukehren.

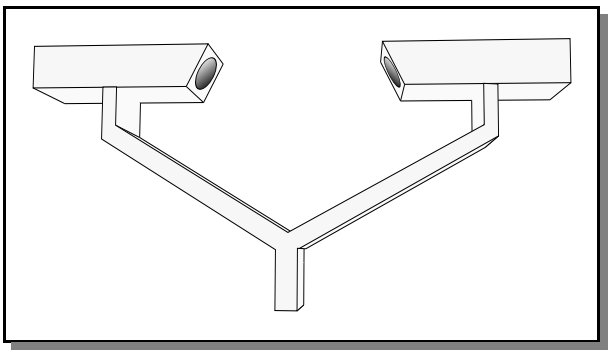


Abb. 8.1: Nebelmessgerät

An Rändern von Autobahnen sieht man zuweilen Nebelmessgeräte wie in Abb. 8.1. Je nach Stärke des Nebels werden mit ihrer Hilfe über Computer entsprechende Geschwindigkeitsbegrenzungen aktiviert.

Bei diesen beiden Anlagen ist entscheidend, dass sie auf äußere Einflüsse reagieren, indem sie Signale von Sensoren auswerten. Wenn diese Sensoren nur zwischen zwei Zuständen – wie im Fall eines Schalters oder Tasters – unterscheiden können, sprechen wir von **digitalem Messen**. Im Fall des Nebelmessgerätes liefert der Sensor viele unterschiedliche Messwerte, welche in ihrer Größe der Nebeldichte entsprechen. In solchen Fällen sprechen wir hier von **analogem Messen**. In den folgenden Abschnitten werden wir uns nur dem digitalen Messen widmen.

8.1 Tasterzustände abfragen

Unsere serielle Schnittstelle besitzt mehrere Eingänge. Werden sie über ein Kabel an eine elektrische Quelle angeschlossen, so übernehmen sie deren Spannungswert. Nach dem TTL-Standard gilt ein Spannungspegel über 2,4 V als **High-Zustand**, und ein Pegel unter 0,8 V als **Low-Zustand**. Bei Pegeln zwischen 0,8 V und 2,4 V ist im TTL-Standard der Zustand undefiniert. Bei unserem FT232-Modul ist dieser Zwischenraum allerdings verschwindend klein: Pegel oberhalb von ca. 1,04 V werden als High-Zustand interpretiert, Pegel unterhalb dieses Wertes als Low-Zustand. Allgemein gilt: Offene Eingänge haben automatisch einen Pegel von 5 V, besitzen also einen High-Zustand. Wenn dieser Eingang z. B. mit Masse verbunden wird, nimmt er den Pegel von 0 V an, d. h. es liegt dann ein Low-Zustand vor.

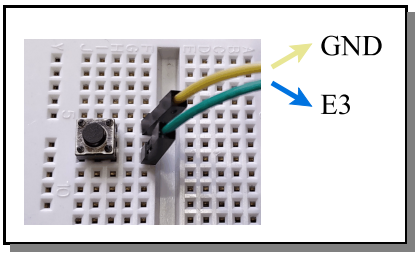


Abb. 8.2: Breadboard mit Taster

In Abb. 8.2 ist der Eingang E3 (RI) über einen Taster mit GND (Masse) verbunden. Ist der Schalter geöffnet, dann ist der Eingang E0 unbeschaltet, also im High-Zustand. Wenn er ge-

schlossen ist, dann ist er leitend mit Masse verbunden; der Eingang E3 ist nun im Low-Zustand.

Die Eingänge E0, E1, E2 und E3 können von MiniPC auf ihren Zustand abgefragt werden. Dazu steht der CPE-Befehl (*C*om-*P*ort-*E*ingang) zur Verfügung:

Code	**Bedeutung**	**Mnemonic**
16.00x	Setze Akkuinhalt auf 1 [0], wenn am Eingang 00x ein High- [Low-] Signal anliegt.	CPE 00x

Um nun zu überprüfen, ob ein Taster wie in Abb. 2 offen oder geschlossen ist, verfährt man deswegen folgendermaßen: Der Eingangszustand wird mit CPE 003 abgefragt; der Inhalt des Akkus gibt dann an, ob der Taster geschlossen oder offen ist:

Akkuinhalt	**Taster**
0	geschlossen
1	offen

Der Akkuwert wird schließlich mit dem ANZ-Befehl auf dem Display angezeigt. Die angegebene Befehlsfolge wird in eine Endlosschleife gesetzt.

Das folgende Programm (messen2.asm) lässt MiniPC den Tasterzustand anzeigen: Wenn der Schalter geschlossen ist, zeigt es im Display eine 0 an, sonst eine 1.

```
//Programm für ft232: Eingänge und Ausgänge standardmäßig auf High(1)
//E3 (RI) über Taster mit GND (Masse) verbinden
//Tasterzustand auf Display anzeigen
//Eingang hat den Wert 0, wenn Taster geschlossen, sonst den Wert 1.

konst:
        E = 3 // Eingangsnummer von RI
ende

progr:
S1:     CPE E           // E1 abfragen
        ANZ
        VZG 010         // 10 ms Verzögerung für Anzeige
        SPU S1          // Endlosschleife
ende
```

Abb. 8.3: Das Programm “messen2.asm”

8.2 Hell und Dunkel anzeigen

Statt des Tasters kann man auch einen **LDR** zwischen Eingang und Masse schalten. Dieses elektronische Bauteil verändert seine Leitfähigkeit je nach Lichteinfall. Fällt viel Licht auf den LDR, so wirkt er wie ein geschlossener Taster, fällt wenig Licht auf den LDR, verhält er sich wie ein geöffneter Taster. Mit Hilfe eines LDRs können wir also messen, ob viel oder wenig Helligkeit vorliegt. Der LDR sollte im abgedunkelten Zustand einige hundert kΩ haben. Je nach LDR reicht es dabei nicht aus, ihn mit der Hand abzuschatten; vielmehr wird man ihn mit einem lichtundurchlässigen Tuch abdecken müssen.

Mit einem kleinen Trick können wir aber erreichen, dass unser MiniPC-Programm auch bereits auf das Abschatten mit der Hand reagiert. Dazu benutzen wir eine Potentiometer-Schaltung (Abb. 8.4); dabei ist R_1 = 1 kΩ. Jetzt liegt am Eingang die Schwellenspannung vor, wenn der LDR einen Widerstand von etwa 250 Ohm hat. Bei größerem Widerstand (geringere Helligkeit) ist der Pegel an E3 größer und unser MiniPC zeigt auf dem Display eine 1 an. Ist der Widerstand kleiner (größere Helligkeit), dann wird eine 0 angezeigt. Zur Erklärung benutzen wir die Potentiometer-Formel:

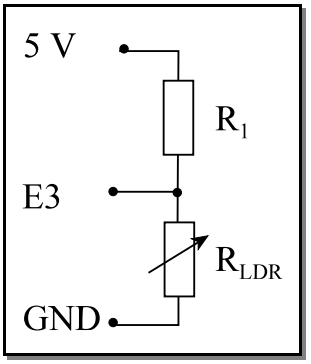


Abb. 8.4

$$\frac{R_{LDR}}{R_1 + R_{LDR}} = \frac{U_S}{U_{ges}}$$

Dabei ist U_{ges} die Summe der Spannungen an den beiden Widerständen, also 5,0 V. Für die Schwellenspannung benutzen wir als Näherungswert 1,0 V. Wir lösen nun die obige Gleichung nach R_{LDR} auf:

$$\begin{aligned}
\frac{R_{LDR}}{R_1 + R_{LDR}} &= \frac{1{,}0V}{5{,}0V} \\
R_{LDR} &= 0{,}2 \cdot (R_1 + R_{LDR}) \\
R_{LDR} &= 0{,}2\,R_1 + 0{,}2\,R_{LDR}) \\
0{,}8\,R_{LDR} &= 0{,}2\,R_1 \\
R_{LDR} &= \frac{0{,}2}{0{,}8} R_1 \\
R_{LDR} &= 250\ \Omega
\end{aligned}$$

Damit erhalten wir für den Wert des LDR-Widerstands, bei dem am Eingang E3 die Schwellenspannung vorliegt, den oben schon angegebenen Wert von 250 Ω.

8.3 Ein Dämmerungsschalter

Den digitalen Messwert für die Helligkeit können wir ausnutzen, um eine LED zum Leuchten zu bringen, wenn eine bestimmte Helligkeit unterschritten wird, d. h. wenn am Eingang von E3

unserer Schaltung aus Abb. 8.4 der Schwellenwert überschritten wird. Mit dem im Akku gespeicherten Zustandswert können wir eine LED steuern: Dazu schließen wir an den Ausgang A2 (DTR) eine LED an. Nun fügen wir im Programm aus Abb. 8.3 eine zusätzliche Konstante A = 2 für den Ausgang A2 ein; außerdem schreiben wir unter den Befehl PCE E die Programmzeile:

```
PCA A
```

Das fertige Programm steht in der Datei “daemmerung.asm”.

8.4 Auf Knopfdruck warten

Eine Fußgängerampel mit Bedarfsanforderung bleibt so lange auf Rot, wie der Taster geöffnet ist. Erst dann, wenn der Taster betätigt wird, springt sie auf Grün um. Um eine solche Ampel zu realisieren, schließen wir einen Taster und zwei LEDs auf der Platine so an:

- den Taster zwischen E3 (RI) und Masse
- die grüne LED an A0 (RTS)
- die rote LED an A2 (DTR)

Nachdem zunächst die Rot-Phase hergestellt worden ist (rote LED ein- und grüne LED ausgeschaltet), durchläuft das Programm eine Warteschleife:

```
Warte:    CPE E       //E3 abfragen
          VGL offen   //Variable offen = 1
          SPB Warte   //Warte solange wie E = offen
```

Diese Schleife wird solange durchlaufen, wie der Taster geöffnet ist, der Eingang E3 also im High-Zustand ist. Erst wenn der Eingang E3 durch den geschlossenen Taster im Low-Zustand ist, wird diese Warteschleife verlassen. Anschließend führt das Programm die Grün-Phase aus (grüne LED an und rote aus) und springt dann wieder an den Anfang des Programms (vgl. Abb. 8.5 auf der nächsten Seite).

In unserem Programm “bedarfsampel1” dauert die Grünphase lediglich 1 Sekunde. Man kann die Grünphase auf einfache Weise verlängern, indem man weitere VZG-Befehle einfügt. Man kann aber auch die Dauer der Grünphase durch eine Variable zeit vorgeben; in dem Programm bedarfsampel2.asm kann man sehen, wie die gesamte Wartezeit für die Grünphase durch eine Zählschleife realisiert wird.

8.5 Aufgaben

8.5.1 Der Wechsel der beiden Ampelphasen soll jeweils durch eine kurze Blinkphase begleitet werden. Ergänze das Programm aus Abb. 8.5 entsprechend.

8.5.2 Programmiere eine Stoppuhr, die mit einem einzigen Taster gestartet und gestoppt wird. Warum sollte man dabei mit dem Zählen erst starten, wenn der Taster wieder losgelassen wird? Hinweise: Die Stoppuhr soll Zehntel-Sekunden anzeigen. Lösung: stoppuhr.asm.

8.5.3 Programmiere einen Reaktionstest; dieser soll im Bereich von 0 bis 255 ms arbeiten.

```
//Fußgängerampel mit Bedarfsanforderung
//für FT232
//Taster zwischen E3 (RI) und Masse
//rote LED zwischen A2 (DTR) und Masse
//grüne LED zwischen A0 (RTS) und Masse

var:
            offen = 1
ende

konst:
            E = 3
            A0 = 0
            A2 = 2
ende

progr:
Anf:        AKO 1       //rote LED einschalten
            CPA A2
            AKO 0       //grüne LED ausschalten
            CPA A0
Warte:      CPE E  //E abfragen
            VGL offen
            SPB Warte   //Warte solange wie E = offen
            AKO 0       //rote LED ausschalten
            CPA A2
            AKO 1       //grüne LED einschalten
            CPA A0
            VZG 250     //... für 1000 ms
            VZG 250
            VZG 250
            VZG 250
            SPU Anf     //Springe zum Anfang (Endlosschleife)
ende
```

Abb. 8.5: Das Programm “bedarfsampel1.asm”

9. Indirekte Adressierungen

In diesem Kapitel soll es um eine neue Adressierungsart gehen, die indirekte Adressierung. Die Idee der indirekten Adressierung besteht darin, in Speicher- oder Sprungbefehlen als Operand nicht direkt die Zieladresse anzugeben, sondern die Adresse der Zelle, in welcher diese Zieladresse steht. Auf diese Weise lassen sich Programme häufig kürzer und übersichtlicher schreiben. Das Konzept der indirekten Programmierung sieht nicht nur auf den ersten Blick etwas abstrakt aus, es ist auch gewöhnungsbedürftig. Anhand verschiedener Beispiele wollen wir es deswegen genauer erläutern und die Vorteile deutlich machen.

9.1 Indirektes Speichern und Laden (AIS und LIA)

Die Ziehung der Lottozahlen verläuft eigentlich ganz ähnlich wie das Würfeln von Zahlen, so wie wir es schon in Kapitel 2 kennen gelernt haben. Hier gilt es allerdings, nicht nur eine einzige Zahl zwischen 1 und 40 zu "würfeln", sondern gleich 6 Stück. Und obendrein soll unser MiniPC sich diese 6 Zahlen merken und am Ende der Ziehung auch noch immer wieder hintereinander anzeigen. Das Zufallsmoment wird durch einen Taster (auf unserem Experimentierboard) erzeugt, welcher von einer "Lottofee" 6 mal betätigt wird.

Abb. 9.1

Dazu zählt MiniPC immer wieder den Inhalt der Variablen zufallszahl hoch bis 40. Wenn zwischendurch der Taster betätigt wird, springt das Programm zur Adresse Merken. In den folgenden Schritten wird die aktuelle Zufallszahl für 1 Sekunde angezeigt und in der Zelle 100 abgespeichert (vgl. Abb. 9.3).

Dabei wird aber nicht auf den bekannten Befehl ABS 100 zurückgegriffen. Vielmehr wird der neue Befehl

AIS xxx (*A*kku *I*ndirekt *S*peichern)

benutzt. Wie dieser Befehl funktioniert, soll anhand der Abb. 9.2 erklärt werden: Hier muss MiniPC den Befehl AIS 016 verarbeiten. MiniPC schaut dazu in der Zelle 016 nach und findet dort den Wert 020. Der Akkuinhalt wird jetzt in dieser Zelle 020 abgespeichert. Man spricht hier von **indirekter Adressierung**, weil die Adresse nicht direkt beim Speicherbefehl steht, sondern der Operand des Speicherbefehls nur angibt, wo die Speicheradresse zu finden ist.

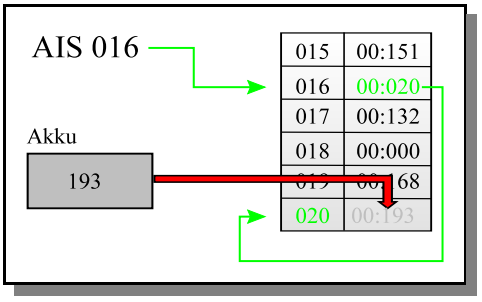


Abb. 9.2

```
//Lottozahlen 6 aus 40
//Taster zwischen E3 (RI) und Masse

var:
        zufallszahl
        lottozahlzähler
        null = 0
        eins = 1
        vierzig = 40
        hsechs = 106
ende

konst:
        zeit = 250                  //Wartezeit 250 ms
        E = 3                       //Eingang E3 (RI)
ende

progr:
                AKO 100
                ABS lottozahlzähler  //lottozahlzähler auf 100
Lotto1:         AKO 0                //neue Zufallszahl bilden
                ABS zufallszahl
Lotto2:         LDA zufallszahl
                ADD eins             //Zufallszahl um 1 erhöhen
                ABS zufallszahl
                ANZ                  //Lottozahl anzeigen...
                VZG 2                //... nur ein kurzen Augenblick
                CPE E                //Eingang E3 abfragen
                VGL null
                SPB Merken           //Aktuelle Zufallszahl merken, wenn Taster gedrückt
                LDA zufallszahl
                VKL vierzig
                SPB Lotto2           //Wenn Zufallszahl < 40 dann nach lotto2:
                SPU Lotto1           //Sonst nach lotto1: springen
Merken:         LDA zufallszahl
                AIS lottozahlzähler
                ANZ                  //Lottozahl 1 Sekunde anzeigen
                VZG zeit
                VZG zeit
                VZG zeit
                VZG zeit
warte:  CPE E
                VGL null
                SPB warte            //warte solange Taster noch gedrückt
                LDA lottozahlzähler
                ADD eins
                ABS lottozahlzähler  //Lottozahlzähler um 1 erhöht
                VKL hsechs
                SPB Lotto1           //wenn lottozahlzähler < 106, dann nach lotto1 springen
                SPB Lotto1           //wenn lottozahlzähler<106, dann nach Lotto1
```

Abb. 9.3: Erster Teil des Programms “lotto1.asm”

Welchen Vorteil bietet es unserem Lottoprogramm, diesen Umweg zu gehen? Bei der Bestimmung der ersten Lottozahl steht unser Lottozahlzähler auf 100. Durch AIS lottozahlzähler wird unsere aktuelle Lottozahl aus dem Akku demnach in die Zelle 100 gebracht. Nun wird der Lottozahlzähler um 1 erhöht und eine weitere Zufallszahl bestimmt. Diesmal wird durch AIS lottozahlzähler die aktuelle Zufallszahl in der Zelle 101 gespeichert. Auf diese Weise werden die Zufallszahlen in den Zellen 100, 101, 102, 103, ... abgelegt.

Ohne die indirekte Adressierung müsste für jede Speicherung einer Lottozahl eine eigene Befehlsfolge geschrieben werden. Für die erste Lottozahl würde der Befehl ABS 100 eingesetzt werden, für die zweite ABS 101 usw. Das wäre viel umständlicher und würde auch deutlich mehr Speicherplatz belegen.

Damit ist der Teil des Programms fertig, welcher die Zufallszahlen erzeugt und abspeichert. Nun können wir uns der übrig gebliebenen Aufgabe widmen, der Anzeige alle Lottozahlen. Nicht nur für das Speichern, sondern auch für das Laden besteht die Möglichkeit einer indirekten Adressierung. Durch den Befehl

LIA xxx (*L*ade *I*ndirekt in den *A*kku)

wird diejenige Zahl in den Akku geladen, deren Adresse in der Zelle xxx steht. Wenn man in Abb. 9.2 den roten Pfeil umkehrt, erhält man eine bildliche Darstellung dieses Befehls. Mit Hilfe dieses Befehls lässt sich die Anzeige der gezogenen Lottozahlen mit wenigen Programmzeilen realisieren:

```
...
Anzeige1:   AKO 100                 //ab hier Anzeige aller Lottozahlen
            ABS lottozahlzähler
Anzeige2:   LIA lottozahlzähler
            ANZ                     //Lottozahl 1 Sekunde anzeigen
            VZG zeit
            VZG zeit
            VZG zeit
            VZG zeit
            AKO 0                   //Zwischen zwei Lottozahlen kurz 0 zeigen
            ANZ
            VZG 250
            LDA lottozahlzähler
            ADD eins
            ABS lottozahlzähler     //Lottozahlzähler um 1 erhöht
            VKL hsechs
            SPB Anzeige2            //wenn lottozahlzähler < 106 dann nach Anzeige2 springen
            SPU Anzeige1            //Lottozahlen endlos anzeigen
ende
```

Abb. 9.4: Zweiter Teil des Programms “lotto1.asm”

Aufgaben:

A 9.1 Erstelle eine Mikrocodetabelle für AIS. Vergleiche sie ans[redacted]n MikroPC.

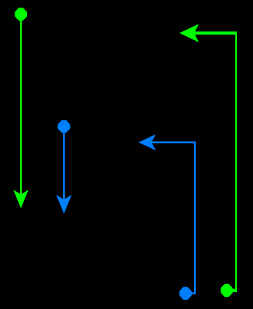
A 9.2 Schreibe ein Programm, welches den kompletten Inhalt d[redacted]m Display anzeigt.

9.2 Indirekte Sprünge und Unterprogramme (SIU)

In dem Programm des letzten Abschnitts tauchte ein und dieselbe[redacted]al auf: Die Befehle

```
UProg:   ANZ
         VZG zeit
         VZG zeit
         VZG zeit
         VZG zeit
```

dienten dazu, den Akkuinhalt für 4 mal 250 ms, also für 1 Sekunde anzuzeigen. Da stellt sich die Frage: Reicht es nicht aus, diese Befehlsfolge nur ein einziges Mal einzugeben und dann bei Bedarf zu dieser Befehlsfolge zu springen? Die Antwort lautet: Ja! Das könnte dann so aussehen wie in Abb. 9.5 dargestellt. Unser **Unterprogramm** wird hier von den Sprungbefehlen in den Zellen mit den Adressen 020 bzw. 030 angesprungen. Am Ende des Unterprogramms muss der Mikroprozessor zu der Zelle zurückkehren, die hinter der Zelle liegt, von welcher aus zu dem Unterprogramm gesprungen wurde. In unserem Fall muss es zur Zelle mit der Adresse 021 oder zur Zelle mit der Adresse 031 springen. Das Programm muss sich also merken, von wo aus es den Sprung zum Unterprogramm vorgenommen hat:



A[redacted]

```
         ...
018      AKO 21
019      ABS Rücksprungadresse        //Rüc[redacted]
020      SPU Uprog
021      ...
         ...
```

Unser Assembler kann diese Rücksprungadresse selbstständig mit Hilfe einer Marke bestimmen:

```
...
            AKO Adr1
            ABS Rücksprungadresse     //Rücksprungadresse Adr1 merken
            SPU Uprog
Adr1:       ...                       //Zeile nach dem Unterprogrammaufruf
            ...
```

Die Rücksprungadresse haben wir damit erfolgreich abgespeichert. Jetzt müssen wir nur noch dafür sorgen, dass am Ende des Unterprogramms genau zu diesen Marken zurückgesprungen wird. An dieser Stelle kommt der **indirekte Sprungbefehl** SIU ins Spiel. Durch

SIU xxx (*S*pringe *I*ndirekt *U*nbedingt)

springt das Programm zu der Zelle, deren Adresse in der Zelle xxx steht.

An das Ende unseres Unterprogramms müssen wir deswegen nur die Zeile

```
SIU Rücksprungadresse
```

anfügen; dann haben wir unser Ziel erreicht. Das entsprechende Programm findet man unter “lotto2.asm”.

Für unsere relativ kurze Befehlsfolge bietet diese neue Unterprogrammtechnik nur eine geringe Einsparung an Speicherplatz; dies ändert sich rasch, wenn das Unterprogramm länger ist und von mehr als zwei oder drei Stellen aus aufgerufen wird.

Aufgabe

A 9.3 Auf dem Experimentierboard wird eine Ampel mit 2 LEDs (rot und grün) gebaut. Die 2 Phasen GRÜN-ROT werden in einer Endlosschleife durchlaufen. Durch eine Taste soll diese Folge durch eine ROT-Blinkphase unterbrochen werden. Programmiere dies mit Hilfe eines Unterprogramms. (ampel_mit_blink.asm)

9.3 Interruptsteuerung

Eine Fußgängerampel mit ROT-Blinkphase wie in der Aufgabe 9.3 aus dem vorangehenden Abschnitt kann - grob skizziert - folgendermaßen gelöst werden:

```
Grphase:    Befehle für Grünphase
            Wenn Taster betätigt, dann Unterprogramm Blinken
Rophase:    Befehle für Rotphase
```

```
                Wenn Taster betätigt, dann Unterprogramm Blinken
                Springe zu Grphase

Blinken:        Zehn mal rot blinken
                Rückkehr zum Hauptprogramm
```

Diese Vorgehensweise hat allerdings einen Nachteil: Der Taster muss zum richtigen Zeitpunkt gedrückt sein, nämlich gerade zum Ende einer Phase. Wird der Taster z. B. lediglich mitten in der Rotphase betätigt, ist er ggf. bereits wieder geöffnet, wenn die Abfrage erfolgt; das Programm verzweigt dann nicht wie gewünscht in das Unterprogramm.

Zwei mögliche Möglichkeiten bieten sich zur Behebung dieses Problems an: Zunächst die einfachste: Der Benutzer des Programms muss den Taster solange gedrückt halten, bis die Blinkphase beginnt. Hierbei taucht aber ein Problem auf: Woher soll der Benutzer dies wissen? Schließlich wird er nicht erst eine Bedienungsanleitung für die Ampel lesen (wollen).

Eine weitere Lösung könnte so aussehen: Die Tasterabfrage wird auch schon während der einzelnen Phasen durchgeführt – etwa jede Sekunde. Aber auch hier gibt es Schwierigkeiten: Möglicherweise drückt der Benutzer ja kürzer als eine Sekunde auf den Taster – und das zur falschen Zeit! Und schon versagt wieder unser Programm. Obendrein steigen der Programmieraufwand und die Länge des Programms an.

Um hier Abhilfe zu schaffen, besitzen viele Prozessoren eine so genannte **Interruptsteuerung**. Diese ermöglicht es, ein Unterprogramm durch ein elektrisches Signal direkt auszulösen. Auch unser MiniPC (nicht jedoch MikroPC) bietet eine derartige Interruptsteuerung; diese muss allerdings erst über *Bearbeiten* - *Interrupt* mit den Optionen H->L bzw. L->H (s. u.) aktiviert werden. MiniPC besitzt dann gegenüber dem Schaltbild Abb. 5.1 einige Ergänzungen:

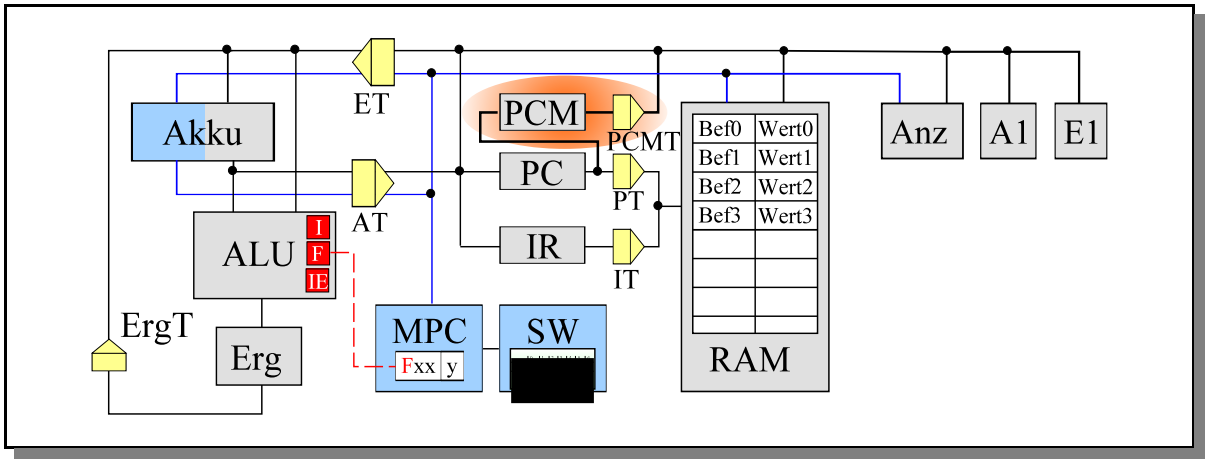


Abb. 9.6: Erweitertes Schema von MiniPC

Zunächst sehen wir neben dem Anzeige-Register ANZ stellvertretend für die verschiedenen Ein- und Ausgabeports die Register A1 und E1; von diesen haben wir bei der Kontrolle unserer

Aktoren und Sensoren schon hinlänglich Gebrauch gemacht. Sodann bemerken wir bei der ALU zwei weitere Register: Das **Interrupt-Register** I und das **Interrupt-Enable-Register** IE. Wie beim Flag-Register handelt es sich um 1-Bit-Register; d. h. diese Register können nur die Werte 0 und 1 aufnehmen. Beim Start von MiniPC hat I den Wert 0 und IE besitzt den Wert 1.

Oberhalb des Programmzählers (PC) erkennen wir noch ein weiteres 8-Bit-Register, den **Program Counter Memory (PCM)**. Der Eingang des PCM-Registers ist an den Ausgang des PC angeschlossen und sein Ausgang ist über das **PCM-Tor (PCMT)** mit dem externen Datenbus verbunden. Hinzu kommen noch weitere Steuerleitungen und elektronische Schaltungen, die wir aber in Abb. 9.5 nicht eingetragen haben.

Eine dieser Schaltungen ist mit dem Eingang E1 verbunden. Wenn dieser Eingang einen Wechsel von High nach Low (Option *Interrupt H->L*) oder von Low nach High (Option *Interrupt L->H*) aufweist, dann wird über eine Steuerleitung das Interruptregister I auf 1 gesetzt. Die grundlegende Idee ist nun: Wenn dieses Register auf 1 gesetzt ist, wird die Programmausführung sofort unterbrochen und in das Blink-Unterprogramm verzweigt. Eine solche Unterbrechung wird als **Interrupt** bezeichnet; das zugehörige Unterprogramm nennt man **Interruptprogramm**.

Eine ganze Reihe von Details müssen wir allerdings noch klären:

– Woher kennt MiniPC die Adresse des Interruptprogramms?
– Wie merkt sich MiniPC die Rücksprungadresse?
– Wie vermeidet man es, dass das Interruptprogramm seinerseits durch ein Interruptsignal unterbrochen werden kann? (Dies wird auch als Reentry-Problem bezeichnet.)

Beginnen wir mit dem Problem der Rücksprungadresse: Weil die Stelle, an welcher das Programm unterbrochen wird, im Voraus nicht bekannt ist, kann die Adresse dieser Stelle nicht direkt vom Programmierer eingegeben werden. Vielmehr muss MiniPC selbst diese Adresse zur Verfügung stellen. Dazu dient das Register PCM. Als Zwischenspeicher für den aktuellen Programmzählerstand kann er diesen dann bei Bedarf über das PCM-Tor und den Datenbus zur Verfügung stellen.

Im Normalbetrieb (wenn also I = 0 ist) wird am Ende jeder Mikroschrittfolge – genauer gesagt bei jeder Ausführung des MPCClear-Befehls – der Wert des PC in den MPC übernommen; MPC zeigt also wie PC auf den nächsten auszuführenden Befehl. Wenn jedoch ein Interruptsignal gegeben worden ist, das I-Register also auf 1 steht, dann wird bei der Ausführung des MPCClear-Befehls zunächst wie üblich der Wert des PC wieder in das PCM-Register übernommen, direkt im Anschluss daran wird aber neben dem MPC auch der PC gelöscht; zusätzlich wird das Interruptregister wieder auf 0 gesetzt. Bei der nächsten HOL-Phase wird daher nicht der Befehl aus der nächsten Zelle, sondern aus der Zelle 000 geholt. Wir brauchen in diese Zelle nur vorab einen Sprungbefehl zum Interruptprogramm zu schreiben – und schon haben wir unser Ziel erreicht: Unser Programm verzweigt automatisch zum Interruptprogramm, wenn ein Interruptsignal gegeben wurde.

Doch halt, etwas fehlt noch: Bevor zum Interruptprogramm gesprungen wird, muss die Rücksprungadresse gesichert werden. Dazu benutzen wir den RPC-Befehl:

RPC xxx *Rette Pro*

Mit diesem Befehl kann der Inhalt des … se des nächsten Befehls im Programm beinhaltet in der Zelle mit d… en.

Ein Programm, welches Interruptsigna… ner so:

```
000   RPC   Rücksprungadre…   …rungadresse
001   SPU   Interruptprogran…   …terruptprogramm
002   Start des eigentlichen P…
```

Deswegen müssen wir Programme mit … r mit der Zelle 002 starten. (Vor dem Betätigen der RUN-Taste ge…

An Hand der Abb. 9.7 wollen wir das… verdeutlichen: Wir gehen davon aus, dass das Interruptsignal (e… 0 V am Eingang E1) im Verlauf des Befehls in der Zelle 087 er… befehl (Mikroschritt PCI) wird der Inhalt des PC wie üblich um … IPCClear-Befehl noch auf 088 steht. Unmittelbar vor der Ausführ… rd dieser Wert in den PCM übernommen. Der PC selbst wird ansch… ßlich war ja das I-Register auf 1 gesetzt.

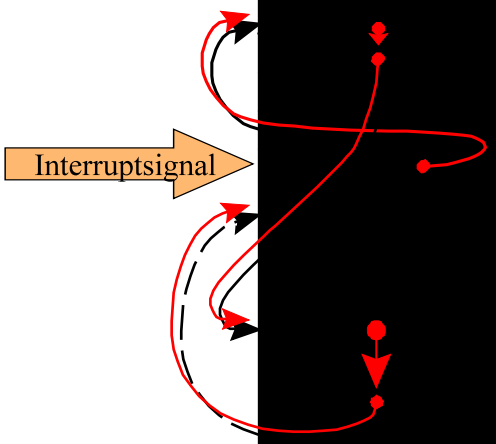


Abb. 9.7: Verarbeitung…

Dadurch springt das Programm zur Z… dieser Zelle speichert die Rücksprungadresse 088 aus dem PCM … un wird der nächste Befehl aus der Zelle 001 ausgeführt, ein Spr… welches in der Zelle 100 beginnt. Am Ende dieses Unterprogram… kter Sprungbefehl, welcher zur Zelle 088 führt. Damit wird das Ha… chtigen Stelle fortgeführt.

Sicherlich ist es schon aufgefallen: Der Rücksprung erfolgt hier mit dem neuen Befehl:

RET xxx (Return = Kehre zurück zu der Zelle, deren Adresse in xxx steht)

Warum wird hier nicht der bereits bekannte Sprung-Befehl SIU benutzt? Das hat etwas mit dem Reentry-Problem zu tun, welches wir schon oben erwähnt haben: Wie können wir vermeiden, dass das Interruptprogramm seinerseits von einem Interruptsignal unterbrochen wird? Bei der Lösung dieses Problems kommt endlich das IE-Register ins Spiel.

Ob ein Interruptsignal überhaupt vom Prozessor registriert wird, darüber entscheidet der Inhalt des Interrupt-Enable-Registers IE. Der Ausgang des IE-Registers steuert nämlich ein Tor, welches sich in der Interruptsignalleitung befindet. Steht im IE eine 1, dann wird das Interruptsignal durchgelassen und setzt das I-Register auf 1; befindet sich im IE hingegen eine 0, so wird das Interruptsignal nicht durch dieses Tor gelassen und dementsprechend kann auch das I-Register nicht gesetzt werden.

Standardmäßig steht IE auf 1, d. h. ein ankommendes Interruptsignal führt dazu, dass das I-Register gesetzt wird. Wir haben bereits gelernt, dass eine 1 im I-Register dafür sorgt, dass bei der Ausführung des Mikroschritts MPCClear automatisch eine Reihe weiterer Signale gegeben werden; noch nicht erwähnt wurde, dass zusätzlich das IE-Register auf 0 gesetzt wird. Dadurch werden alle künftigen Interruptsignale ignoriert.

Erst am Ende des Interruptprogramms darf und muss das IE-Register wieder auf 1 gesetzt werden. Dies geschieht durch den RET-Befehl: Der RET-Befehl unterscheidet sich von dem SIU-Befehl nur dadurch, dass er zusätzlich das IE-Register auf 1 setzt. Auf diese Weise versetzt er den Prozessor in die Lage, erneut auf Interruptsignale zu reagieren.

Aufgaben:

A 9.4 Warum darf der RPC-Befehl die Adresse nicht direkt aus dem PC-Register holen?

A 9.5 Beschreibe mit Worten die Mikroschritten für den Befehl RPC.

A 9.6 Ohne das IE-Register könnte auch innerhalb eines Interruptprogramms ein Interrupt ausgelöst werden. Erläutere, dass dann eine Rückkehr in das Hauptprogramm nicht mehr möglich wäre.

Beachte bei den folgenden Aufgaben, dass bei MiniPC zuerst die COM-Schnittstelle und der Interruptzugriff über das Bearbeiten-Menü aktiviert werden müssen.

A 9.7 Eine Leuchtdiode soll fortwährend mit einer Frequenz von 10 Hz blinken. Wenn ein Taster betätigt wird, soll das Blinken für 2 Sekunden unterbrochen werden. Schreibe ein entsprechendes Programm und teste es aus (blinken_ir.asm).

A 9.8 Löse die Aufgabe 9.3 jetzt mit Hilfe des Interruptkonzepts.

Das MiniPC-System verfügt über ein einfaches Betriebssytem. Dadurch können über Schaltflächen Daten mit MiniPC ausgetauscht und der Programmablauf beeinflusst werden.

Schaltfläche	Funktion
RUN	Startet den Programmablauf.
STP	Stoppt den Programmablauf und bringt den Inhalt der Zelle 000 zur Anzeige.
STEP	Führt ein einzelnen Befehl aus; der Programmzähler (PC) wird dabei um 1 erhöht.
ACC	Bringt den Inhalt des Akkus zur Anzeige
CLR	Setzt das System zurück; insbesondere wird die Anzeige gelöscht und der PC auf 000 gesetzt.
xxx PC	Stellt den PC auf die Adresse xxx ein.
xxx OUT	Gibt den Inhalt der Zelle xxx aus; bei nochmaligen Betätigen wird der Inhalt der *nächsten* Zelle ausgegeben.
xxyyy INP	Gibt den Befehl xxyyy in die aktuelle Zelle ein.
RAML	Lädt eine obj-Datei in das RAM von MiniPC.
RAMS	Speichert das RAM von MiniPC als obj-Datei.
9 OUT	Gibt den aktuellen Stand des PCs aus.

Um bei MiniPC den Zugang zur seriellen Schnittstelle zu aktivieren, wählt man über Bearbeiten → Schnittstelle eine passende COM-Nummer aus. Die auf dem Rechner existierenden COM-Schnittstellen kann man über Systemsteuerung - Hardware in Erfahrung bringen.

Die in Kapitel 9 angesprochene Interrupt-Technik benötigt relativ viele Rechnerressourcen; deswegen steht sie normalerweise nicht zur Verfügung. Sie muss über Bearbeiten - Interrupt aktiviert werden.

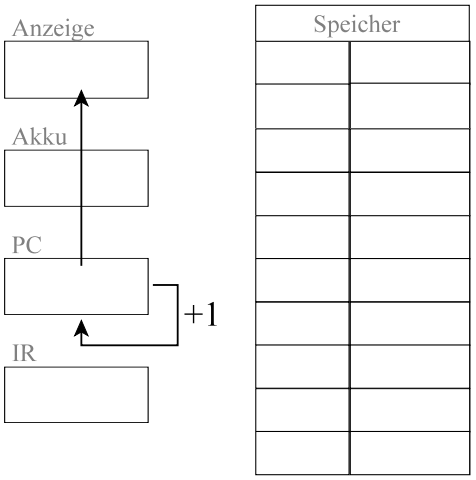
MiniPC kann auch Daten mit dem (realen) Modellcomputer CP1 von kosmos austauschen. Näheres dazu findet man in der Hilfestellung von MiniPC.

Code	Mnemonic	Bedeutung
01.000	HLT	Halt
02.000	ANZ	Akku-Inhalt anzeigen
03.xxx	VZG	Verzögern um xxx Millisekunden
04.xxx	AKO	Konstante xxx in den Akku laden
05.xxx	LDA	Inhalt von Zelle xxx in den Akku laden
06.xxx	ABS	Akku-Inhalt in Zelle xxx speichern
07.xxx	ADD	Zum Akku-Inhalt den Inhalt von Zelle xxx addieren; Ergebnis im Akku
08.xxx	SUB	Vom Akku-Inhalt den Inhalt von Zelle xxx subtrahieren; Ergebnis im Akku
09.xxx	SPU	Unbedingt auf Adresse xxx springen
10.xxx	VGL	Prüfen, ob Akku-Inhalt gleich Inhalt von Zelle xxx
11.xxx	SPB	Bedingt auf Adresse xxx springen
12.xxx	VGR	Prüfen, ob Akku-Inhalt größer als Inhalt von Zelle xxx
13.xxx	VKL	Prüfen, ob Akku-Inhalt kleiner als Inhalt von Zelle xxx
14.000	NEG	Akku-Inhalt negieren (nur 0 oder 1)
15.xxx	UND	UND-Verknüfung zwischen Akku-Inhalt und Inhalt von Zelle xxx (nur 0 oder 1)
16.00x	CPE	Zustand am Eingang x in den Akku bringen (0 oder 1); x = 0, 1, 2, 3
17.00x	CPA	Akku-Inhalt (0 oder 1) am Ausgang ausgeben; x = 0, 1, 2
18.00x	---	nicht implementiert
19.xxx	LIA	Akku indirekt laden (mit Inhalt der Zelle, deren Adresse unter xxx steht)
20.xxx	AIS	Akku-Inhalt speichern (in der Zelle, deren Adresse unter xxx steht)
21.xxx	SIU	Indirekt unbedingt springen (auf Adresse, die unter xxx steht)
22.xxx	RPC	Inhalt vom PCM in der Zelle xxx speichern
23.xxx	RET	zurück zu der Zelle springen, deren Adresse in xxx steht; zusätzlich das IE-Register auf 1 setzen

Befehl:	**HLT**	*Code*:	01
Beschreibung:	Hält das Programm an. Bei MiniPC wird danach der PC um 1 erhöht und zur Anzeige gebracht. Bei MikroPC stimmt der HLT-Befehl mit dem HOL-Befehl überein; PC bleibt hier unverändert.		

Beispiel (nur MiniPC):

HLT



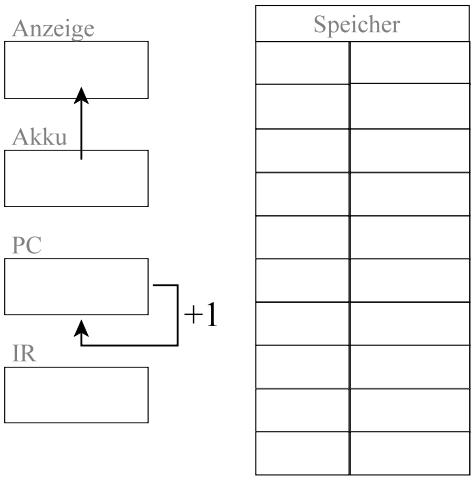
Mikroschritttabelle (nur MikroPC):

Element	0	1	2	3	4	5	6	7	8	9
ET										
AT										
PC										
PCI										
PT	X	X								
IR										
IT										
Akku										
Erg										
ErgT										
MPCClear										
MPCLoad		X								
RamRead	X	X								
RamWrite										
ALU0										
ALU1										
ALU2										
Anz										

Befehl:	**ANZ**	*Code*:	02
Beschreibung:	Kopiert den Akkuinhalt ins Anzeigeregister. Der Programmzähler wird um 1 erhöht.		

Beispiel:

ANZ

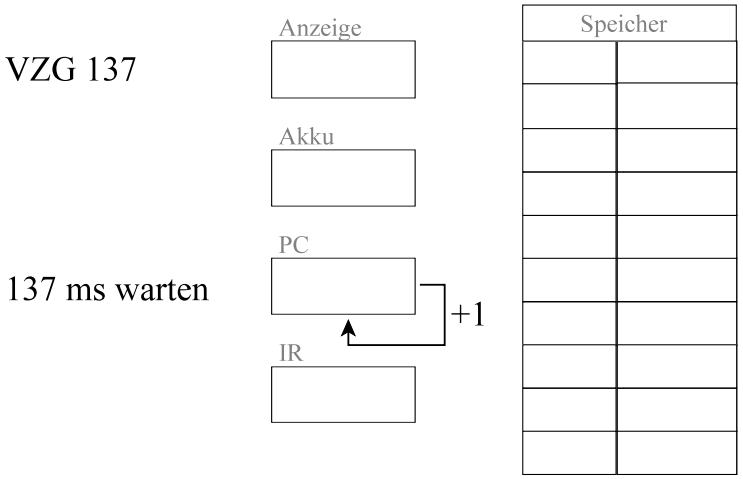


Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET										
AT	X	X								
PC										
PCI			X							
PT										
IR										
IT										
Akku										
Erg										
ErgT										
MPCClear				X						
MPCLoad										
RamRead										
RamWrite										
ALU0										
ALU1										
ALU2										
Anz	X									

Befehl:	**VZG xxx**	*Code*:	03.xxx
Beschreibung:	Verzögert den Programmablauf. Bei MiniPC wird eine Pause von xxx Millisekunden eingelegt. Bei MikroPC werden 10 Mikroschritte durchgeführt. Der Programmzähler wird um 1 erhöht.		

Beispiel (MiniPC):



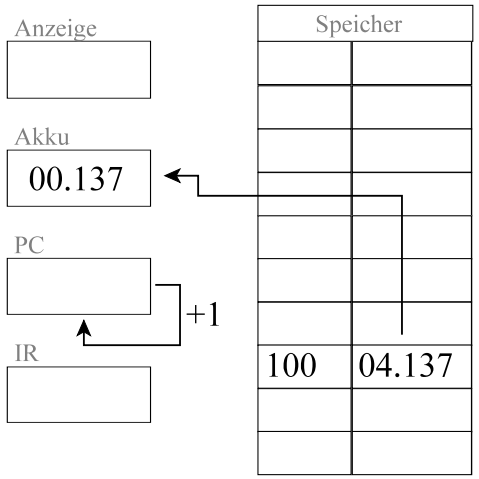
Mikroschritttabelle (MikroPC):

Element	0	1	2	3	4	5	6	7	8	9
ET										
AT										
PC										
PCI	X									
PT										
IR										
IT										
Akku										
Erg										
ErgT										
MPCClear										X
MPCLoad										
RamRead										
RamWrite										
ALU0										
ALU1										
ALU2										
Anz										

Befehl:	**AKO xxx**	*Code*:	04.xxx
Beschreibung:	Lädt xxx (bei MikroPC: 04.xxx) in den Akku; der Programmzähler wird um 1 erhöht.		

Beispiel (MiniPC):

AKO 137

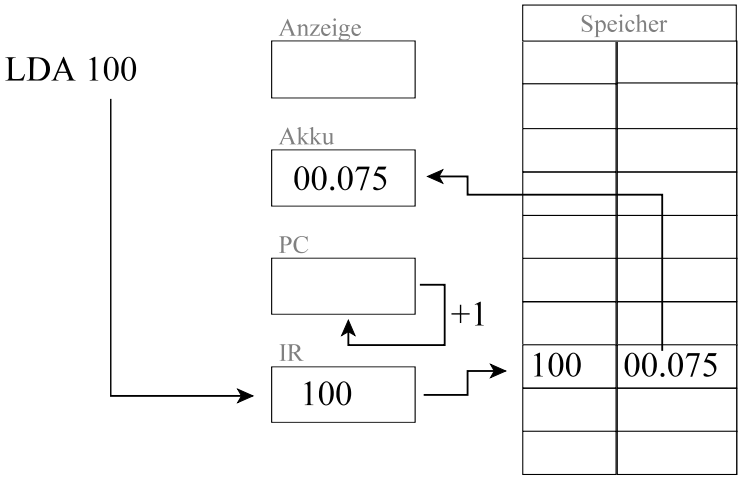


Mikroschritttabelle (MikroPC):

Element	0	1	2	3	4	5	6	7	8	9
ET	X	X								
AT										
PC										
PCI			X							
PT	X	X								
IR										
IT										
Akku	X									
Erg										
ErgT										
MPCClear				X						
MPCLoad										
RamRead	X	X								
RamWrite										
ALU0										
ALU1										
ALU2										
Anz										

Befehl:	**LDA xxx**	*Code*:	05.xxx
Beschreibung:	Lädt den Inhalt der Speicherzelle mit der Adresse xxx in den Akku; der Programmzähler wird um 1 erhöht.		

Beispiel:

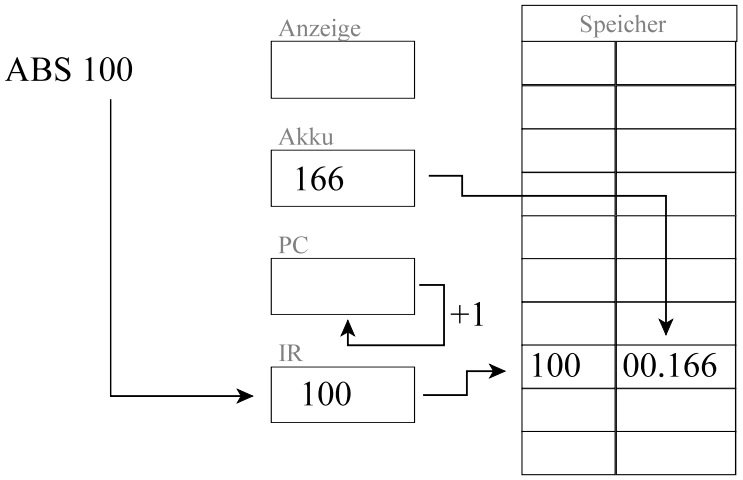


Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET				X	X					
AT										
PC										
PCI					X					
PT	X	X								
IR	X									
IT			X	X	X					
Akku				X						
Erg										
ErgT										
MPCClear						X				
MPCLoad										
RamRead	X	X		X	X					
RamWrite										
ALU0										
ALU1										
ALU2										
Anz										

Befehl:	**ABS xxx**	*Code*:	06.xxx
Beschreibung:	Speichert den Inhalt des Akkus in der Speicherzelle xxx ab; der Programmzähler wird um 1 erhöht.		

Beispiel:

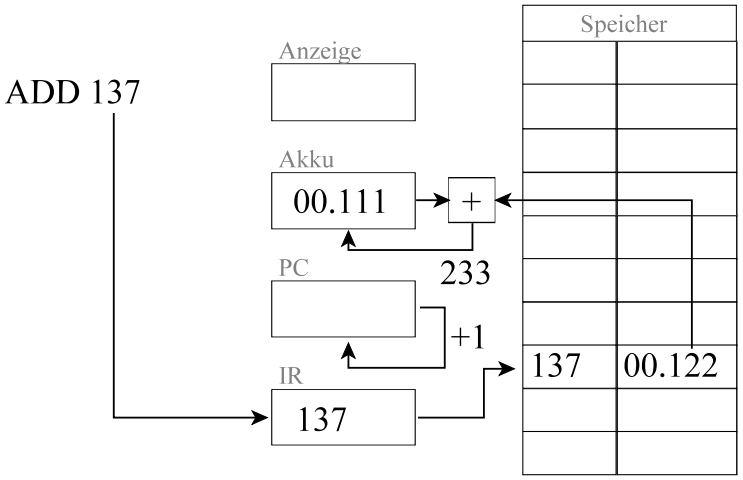


Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET										
AT				X	X					
PC										
PCI						X				
PT	X	X								
IR	X									
IT			X	X	X					
Akku										
Erg										
ErgT										
MPCClear							X			
MPCLoad										
RamRead	X	X								
RamWrite				X						
ALU0										
ALU1										
ALU2										
Anz										

Befehl:	**ADD xxx**	*Code*:	07.xxx
Beschreibung:	Addiert den Inhalt der Zelle xxx zum Inhalt des Akkus; das Ergebnis steht im Akku. Der Programmzähler wird um 1 erhöht.		

Beispiel:

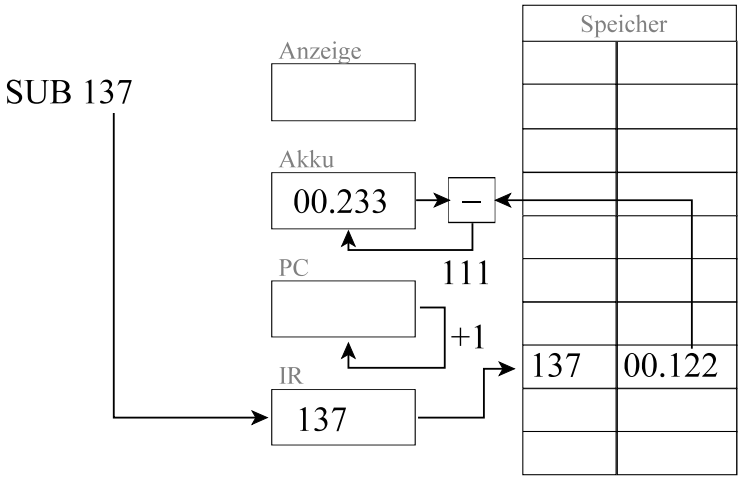


Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET				X	X					
AT										
PC										
PCI								X		
PT	X	X								
IR	X									
IT			X	X	X					
Akku						X				
Erg				X						
ErgT						X	X			
MPCClear									X	
MPCLoad										
RamRead	X	X		X	X					
RamWrite										
ALU0			X	X	X					
ALU1										
ALU2										
Anz										

Befehl:	**SUB xxx**	*Code*:	08.xxx
Beschreibung:	Subtrahiert den Inhalt der Zelle xxx vom Inhalt des Akkus; das Ergebnis steht im Akku. Der Programmzähler wird um 1 erhöht.		

Beispiel:

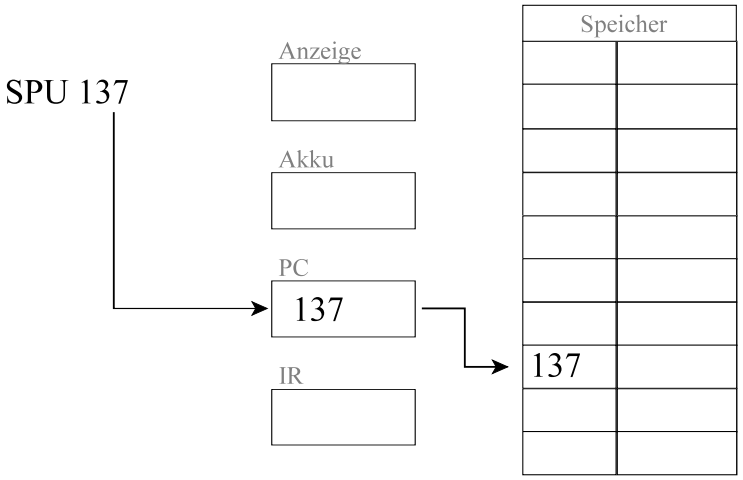


Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET				X	X					
AT										
PC										
PCI								X		
PT	X	X								
IR	X									
IT			X	X	X					
Akku						X				
Erg				X						
ErgT						X	X			
MPCClear									X	
MPCLoad										
RamRead	X	X		X	X					
RamWrite										
ALU0										
ALU1			X	X	X					
ALU2										
Anz										

Befehl:	**SPU xxx**	*Code*:	09.xxx
Beschreibung:	Springe zur Zelle mit der Adresse xxx. Der Programmzähler wird also auf xxx gesetzt.		

Beispiel:

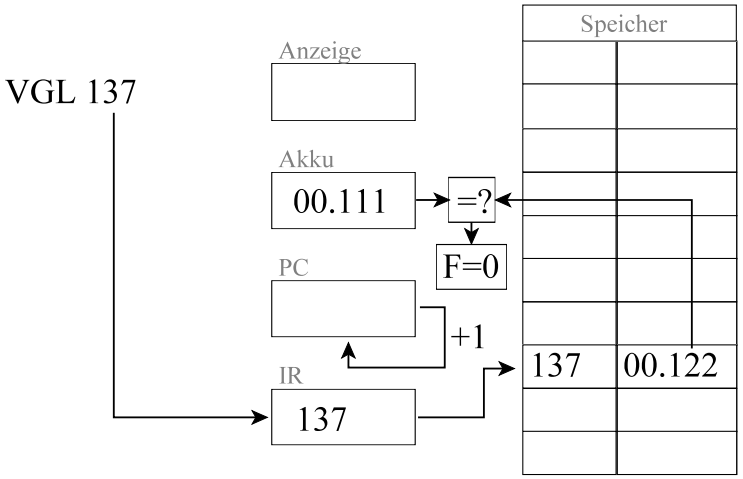


Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET			X	X						
AT					X	X				
PC					X					
PCI										
PT			X	X						
IR										
IT										
Akku			X				X			
Erg	X									
ErgT							X	X		
MPCClear									X	
MPCLoad										
RamRead			X	X						
RamWrite										
ALU0	X	X								
ALU1										
ALU2										
Anz										

Befehl:	**VGL xxx**	*Code*:	10.xxx
Beschreibung:	Vergleiche den Inhalt des Akkus mit dem Inhalt der Zelle xxx. Stimmen sie überein, wird das Flagregister auf 1 gesetzt, sonst auf 0. Der Programmzähler wird um 1 erhöht.		

Beispiel:

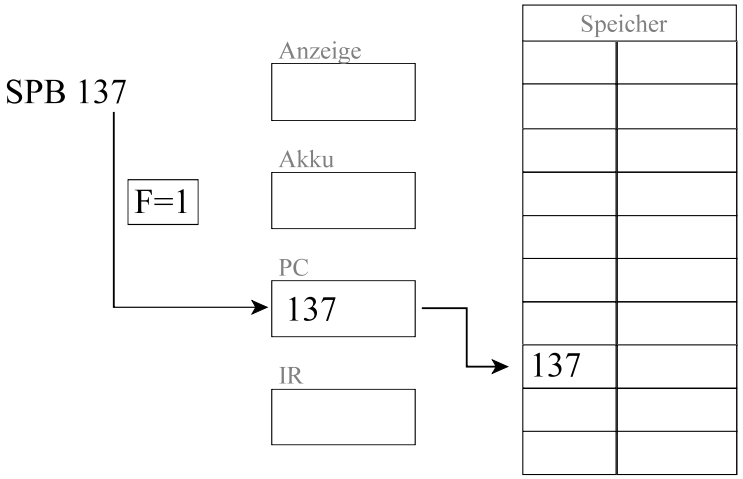


Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET				X	X					
AT										
PC										
PCI						X				
PT	X	X								
IR	X									
IT			X	X	X					
Akku										
Erg										
ErgT										
MPCClear							X			
MPCLoad										
RamRead	X	X		X	X					
RamWrite										
ALU0					X					
ALU1					X					
ALU2										
Anz										

Befehl:	SPB **xxx**	*Code*:	11.xxx
Beschreibung:	Wenn das Flagregister gesetzt ist (F=1), dann springe zur Zelle xxx, ansonsten gehe zur nächsten Zelle. Der PC wird also auf xxx gesetzt, wenn F den Wert 1 hat, ansonsten wird der PC um 1 erhöht.		

Beispiel:



Mikroschritttabelle:

Flag = 0

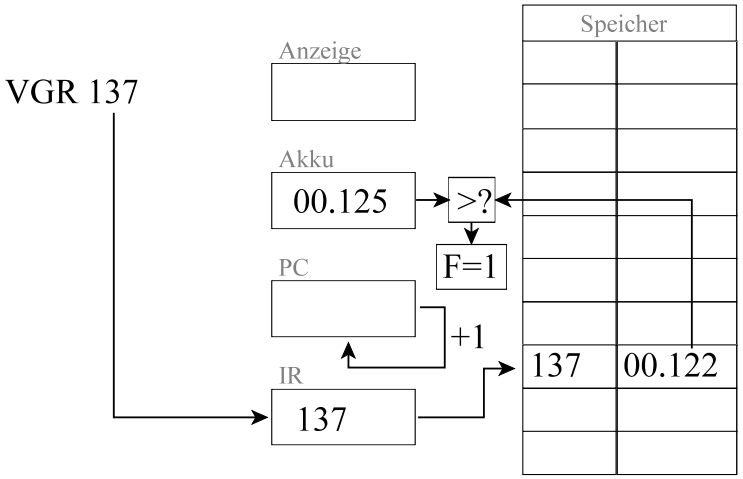
Element	0	1	2	3	4	5	6	7	8	9
ET										
AT										
PC										
PCI	X									
PT										
IR										
IT										
Akku										
Erg										
ErgT										
MPCClear		X								
MPCLoad										
RamRead										
RamWrite										
ALU0										
ALU1										
ALU2										
Anz										

Flag = 1

Element	0	1	2	3	4	5	6	7	8	9
ET			X	X						
AT					X	X				
PC					X					
PCI										
PT			X	X						
IR										
IT										
Akku			X				X			
Erg	X									
ErgT							X	X		
MPCClear									X	
MPCLoad										
RamRead			X	X						
RamWrite										
ALU0	X	X								
ALU1										
ALU2										
Anz										

Befehl:	**VGR xxx**	*Code*:	12.xxx
Beschreibung:	Vergleiche den Inhalt des Akkus mit dem Inhalt der Zelle xxx. Ist der Akkuinhalt größer, wird das Flagregister auf 1 gesetzt, sonst auf 0. Der Programmzähler wird um 1 erhöht.		

Beispiel:

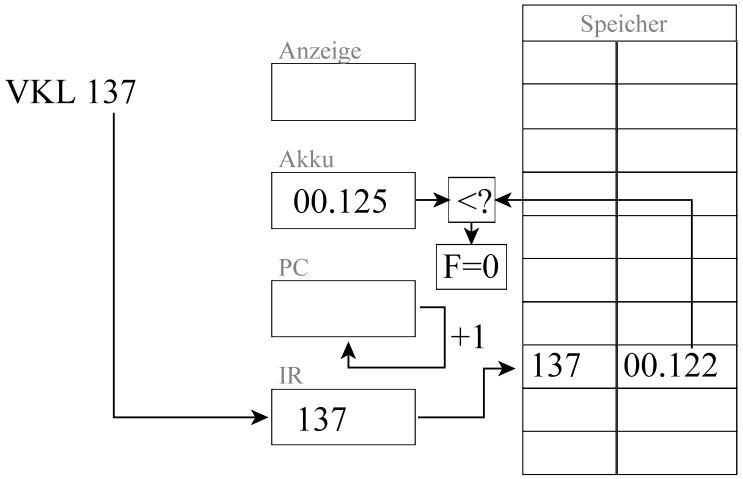


Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET				X	X					
AT										
PC										
PCI						X				
PT	X	X								
IR	X									
IT			X	X	X					
Akku										
Erg										
ErgT										
MPCClear							X			
MPCLoad										
RamRead	X	X			X					
RamWrite										
ALU0										
ALU1										
ALU2					X					
Anz										

Befehl:	**VKL xxx**	*Code*:	13.xxx
Beschreibung:	Vergleiche den Inhalt des Akkus mit dem Inhalt der Zelle xxx. Ist der Akkuinhalt kleiner, wird das Flagregister auf 1 gesetzt, sonst auf 0. Der Programmzähler wird um 1 erhöht.		

Beispiel:



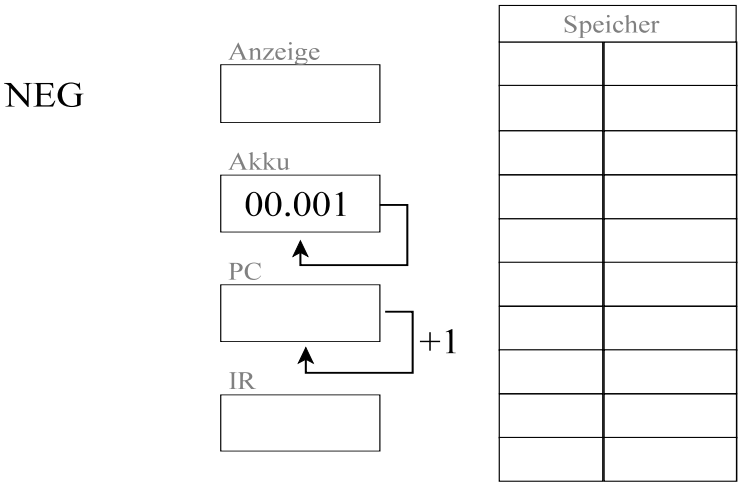
Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET				X	X					
AT										
PC										
PCI						X				
PT	X	X								
IR	X									
IT			X	X	X					
Akku										
Erg										
ErgT										
MPCClear							X			
MPCLoad										
RamRead	X	X		X	X					
RamWrite										
ALU0					X					
ALU1										
ALU2					X					
Anz										

Befehl: **NEG**	*Code*: 14.000

Beschreibung: Negiert den Wert des Akkus: Aus 0 wird 1, aus 1 wird 0. Ist der Wert des Akkus größer als 1, dann erfolgt eine Fehlermeldung (MiniPC) oder die Negation wird nur von der niederwertigsten Stelle der Zahl im Zweiersystem gebildet (MikroPC). Der Programmzähler wird um 1 erhöht.

Beispiel:

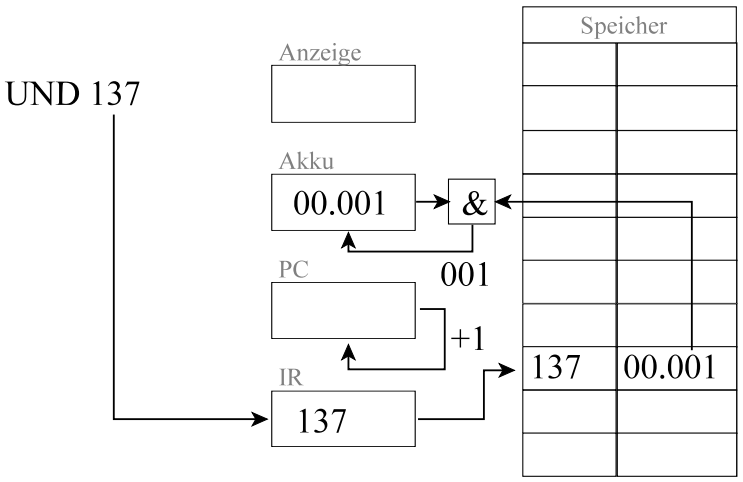


Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET				X	X					
AT										
PC										
PCI								X		
PT	X	X								
IR	X									
IT			X	X	X					
Akku						X				
Erg				X						
ErgT						X	X			
MPCClear									X	
MPCLoad										
RamRead	X	X		X	X					
RamWrite										
ALU0			X	X	X					
ALU1			X	X	X					
ALU2			X	X	X					
Anz										

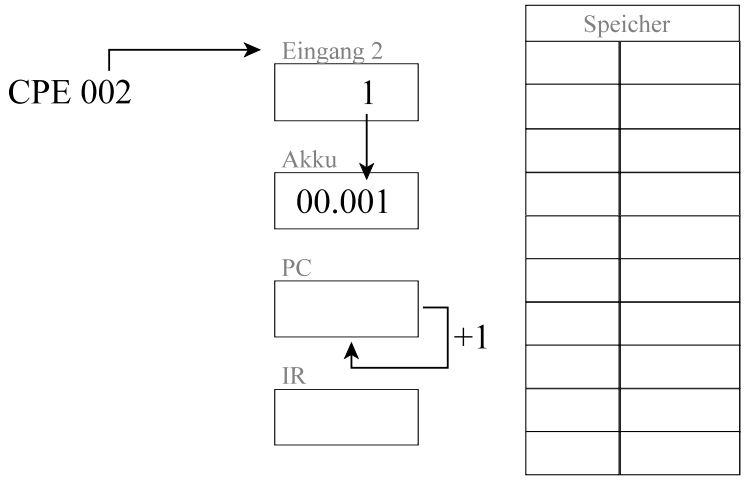
Befehl:	**UND xxx**	*Code*:	15.xxx
Beschreibung:	Bilde die UND-Verknüpfung von Akku und Zelle xxx und lege das Ergebnis wieder im Akku ab; als Werte für Akku und Zelle xxx sind nur 0 und 1 zugelassen (vgl. NEG-Befehl). Der Programmzähler wird um 1 erhöht.		

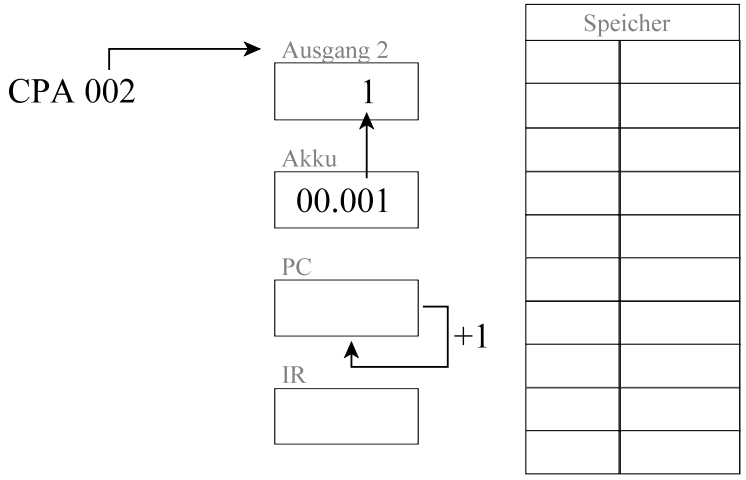
Beispiel:



Mikroschritttabelle:

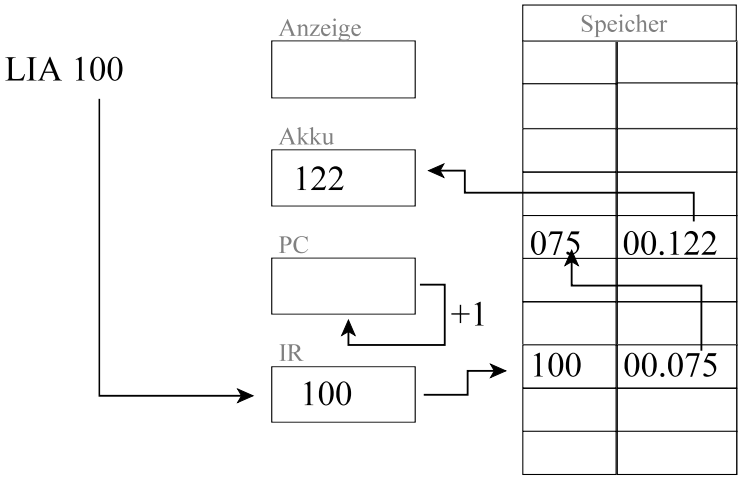
Element	0	1	2	3	4	5	6	7	8	9
ET				X	X					
AT										
PC										
PCI								X		
PT	X	X								
IR	X									
IT			X	X	X					
Akku						X				
Erg				X						
ErgT						X	X			
MPCClear									X	
MPCLoad										
RamRead	X	X		X	X					
RamWrite										
ALU0										
ALU1			X	X	X					
ALU2			X	X	X					
Anz										

Befehl:	**CPE 00x**	*Code*:	16.00x
Beschreibung:	Setze Akkuinhalt auf 1 [0], wenn am Eingang x ein High- [Low-] Signal anliegt; x kann die Werte 0..3 annehmen. Der Programmzähler wird um 1 erhöht. Den Befehl CPE gibt es nur bei MiniPC.		
Beispiel:			

Befehl:	**CPA 00x**	*Code*:	17.00x
Beschreibung:	Setze den Ausgang x auf High [Low], wenn der Akku den Wert 1 [0] besitzt; x kann die Werte 0..2 annehmen. Der Programmzähler wird um 1 erhöht. Den Befehl CPA gibt es nur bei MiniPC.		
Beispiel:			

Befehl:	**LIA xxx**	*Code*:	19.xxx
Beschreibung:	Lade den Akku mit dem Inhalt der Zelle, deren Adresse in der Zelle xxx steht (indirekte Adressierung). Der Programmzähler wird um 1 erhöht.		

Beispiel: Am Ende steht die Zahl 122 im Akku.

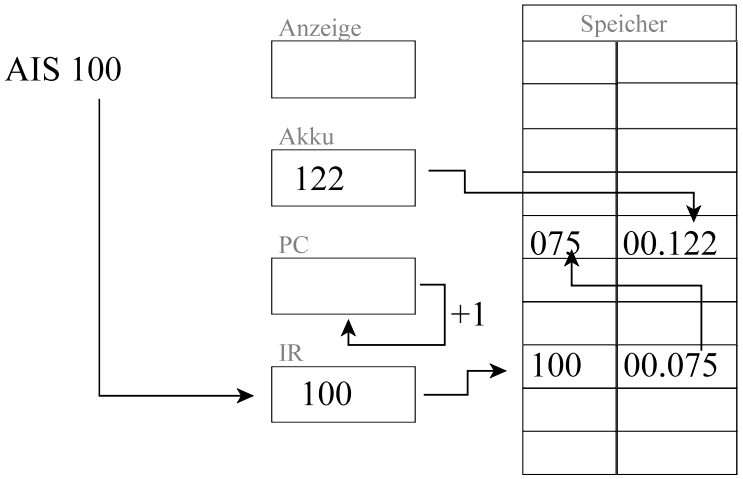


Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET			X	X			X	X		
AT					X	X				
PC										
PCI								X		
PT										
IR	X				X					
IT			X	X			X	X		
Akku			X				X			
Erg										
ErgT										
MPCClear									X	
MPCLoad										
RamRead	X	X	X	X			X	X		
RamWrite										
ALU0										
ALU1										
ALU2										
Anz										

Befehl: **AIS xxx**	*Code*: 20.xxx
Beschreibung:	Speichere den Wert des Akkus in die Zelle, deren Adresse in der Zelle xxx steht (indirekte Adressierung). Der Programmzähler wird um 1 erhöht.

Beispiel:

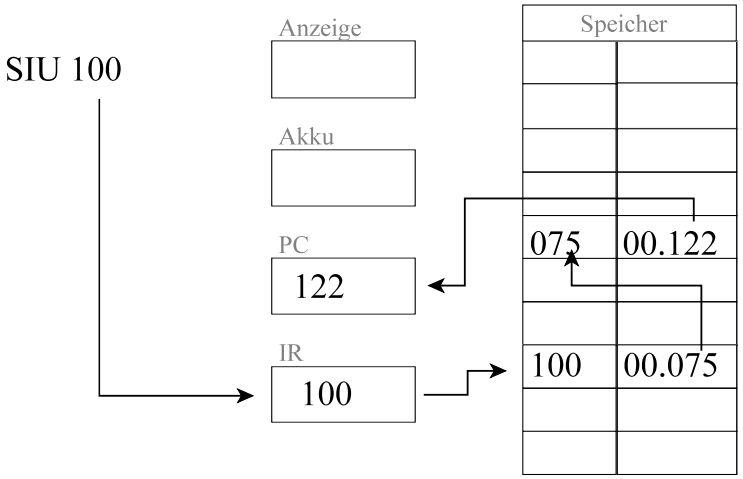


Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET			X	X						
AT					X	X	X	X		
PC										
PCI									X	
PT	X	X								
IR	X				X					
IT			X	X				X	X	
Akku			X				X			
Erg	X									
ErgT							X	X		
MPCClear										X
MPCLoad										
RamRead	X	X	X	X						
RamWrite								X		
ALU0	X	X								
ALU1										
ALU2										
Anz										

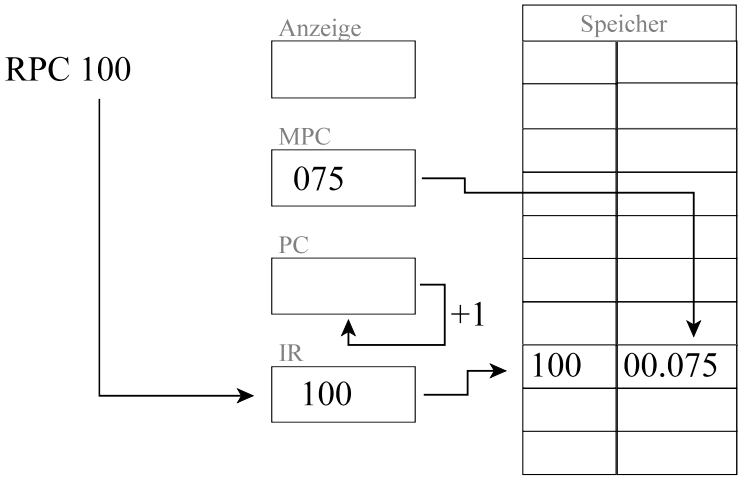
Befehl: **SIU xxx**	*Code*: 21.xxx
Beschreibung:	Springe zur Zelle mit der Adresse, die sich in der Zelle xxx befindet. Befindet sich in der Zelle xxx der Wert yyy, so wird der Programmzähler wird also auf yyy gesetzt.

Beispiel:



Mikroschritttabelle:

Element	0	1	2	3	4	5	6	7	8	9
ET			X	X						
AT					X	X				
PC							X			
PCI										
PT			X	X						
IR					X					
IT						X	X	X		
Akku			X			X				
Erg	X									
ErgT						X	X			
MPCClear									X	
MPCLoad										
RamRead			X	X			X	X		
RamWrite										
ALU0	X	X								
ALU1										
ALU2										
Anz										

Befehl:	**RPC xxx**	*Code*:	22.xxx
Beschreibung:	Speichert den Wert des PC-Puffers MPC in der Zelle xxx ab. Der Programmzähler wird um 1 erhöht. Dieser Befehl existiert nur bei MiniPC.		
Beispiel:			

Befehl:	**RET xxx**	*Code*:	23.xxx
Beschreibung:	Wie SIU xxx. Zusätzlich wird das IE-Flag auf 1 gesetzt; dadurch kann der Prozessor wieder auf Interrupts reagieren. Dieser Befehl existiert nur bei MiniPC.		
Beispiel:	vgl. SIU		

MiniPC besitzt ein rudimentäres Betriebssystem, welches eine Reihe von Fehlern abfangen und anzeigen kann. Die Bedeutung der Fehler F 001 bis F 007 ist folgende:

F 001	**Zu wenige oder zu viele Ziffern eingetippt**
a)	Es werden mehr als fünf Ziffern eingegeben.
b)	Es werden keine, eine, zwei oder vier Ziffern eingegeben und sodann die INP-Taste gedrückt.
c)	Es werden eine, zwei, vier oder fünf Ziffern eingegeben und sodann die PC-Taste gedrückt.
d)	Es werden eine, zwei, vier oder fünf Ziffern eingegeben und dann die OUT-Taste gedrückt.

F 002	**Laufendes Programm erkennt ungültigen Operationscode**
Während des Programmlaufes wurde aus einer Speicherzelle ein Wert geholt, der mit 00 ... beginnt, also keinen Befehl, sondern ein Datum darstellt. Der Programmlauf wird unterbrochen, der Programmzählerstand gibt die Adresse des fehlerhaften „Befehls" an (Taste PC drücken).	

F 003	**Laufendes Programm erkennt ungültige Adressangabe.**
a)	Beim Programmlauf wird der Programmzählerstand 127 (für die Grundversion) bzw. 255 (mit Speichererweiterung) überschritten.
b)	Beim Programmlauf wird aus dem Speicher ein Befehl mit einer Adressangabe für eine Speicherzelle geholt, die “nicht da ist" (Adressangabe größer als 127).

F 004	**Ungültigen Operationscode oder Adresse eingetippt**
a)	Es wird über die Tastatur ein Befehl eingegeben, in dem eine Adressangabe größer als 255 enthalten ist.
b)	Beim fortlaufenden Drücken der Taste OUT wird der Adressbereich überschritten (d. h. es wird versucht, den Inhalt einer Speicherzelle anzuschauen, die „gar nicht da ist").
c)	Es wird ein Befehl mit einem ungültigen Operationscode eingegeben (OP-Code größer als 24, also z. B. 25.xxx).

F 005	**Laufendes Programm erkennt ungültigen Operanden.**
Beim Programmlauf wird bei den Befehlen ADD, SUB, VGL, NEG, UND ein ungültiger Operand erkannt. (Beispiel: bei der Ausführung eines Additionsbefehls erkennt der Computer, dass in der Speicherzelle, deren Inhalt addiert werden soll, kein Datum, sondern ein Befehl steht). Der Programmlauf wird unterbrochen, der Programmzählerstand gibt an, an welcher Stelle das Programm abgebrochen wurde (Taste PC drücken!).	

F 006	**Rechenergebnis größer als 255 oder kleiner als 0**
a)	Bei der Ausführung des Additionsbefehls wird das Ergebnis größer als 255.
b)	Bei der Ausführung des Subtraktionsbefehls wird das Ergebnis kleiner als null (keine negativen Zahlen möglich).

F 007	**Fehler beim Up- bzw. Download**
Beim der Datenübertragung zwischen MiniPC und einem (realen) CP1-Computer ist ein Fehler aufgetreten.	

Experimentierboards

USB-COM-Platine von AK-Moduls-BUS

Diese Platine wird über ein USB-Drucker-Kabel mit dem PC verbunden. Mit Hilfe eines FT232R-Bausteins werden die USB-Signale vom PC in TTL-Signale gewandelt. Die Anschlüsse der Aktoren und Sensoren werden in die Kontakte auf der rechten Seite der Platine gesteckt. Die Platine kostet ca. 15 Euro (Stand Juli 2021).

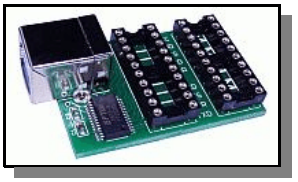


Abb. A1

Bezugsquelle: www.ak-modul-bus.de

Breadboard mit FT232-Modul

Preiswerter dürfte es sein, ein kleines Breadboard und ein FT232-Modul zu kaufen (vgl. Abb. 7.2). Beide gibt es bereits für wenige Euro.

Bezugsquellen: z. B. AZ-Delivery, Amazon, diverse Anbieter bei Ebay

Neben dem FT232 existieren weitere Bausteine mit ähnlichen Eigenschaften wie beim FT232, z. B. der CP2102-Baustein. Sie sind teilweise preisgünstiger als das FT232-Modul. Allerdings sind sie oft in der Benutzung nicht ganz so einfach: Sie lassen sich häufig nicht so leicht auf einem Breadboard montieren, bei manchen muss auch der USB-Treiber von Hand installiert werden.

COM-Platine zum Selbstbau

Wer noch eine COM-Buche an seinem PC hat oder ein USB-COM-Kabel besitzt (s. u.), der mag vielleicht auch eine COM-Platine selbst bauen. In der Abb. A2 ist eine solche Platine abgebildet; hier wurde aus Kostengründen auf eine COM-Buchse auf der Platine verzichtet und die Leitungen des Kabels direkt angeschlossen. In Abb. 2 sind alle Buchsen einer Buchsenleiste untereinander verbunden; dadurch können einfache Schaltungen (z. B. Reihenschaltungen) realisiert werden. Die Ohrhörerbuchse ist mit der darunter liegenden Buchsenleiste und dem Masseanschluss (GND) verbunden. Wie bei der AK-MODUL-BUS-Platine werden die Anschlüsse der Aktoren und Sensoren in die Kontakte der Buchsenleisten gesteckt.

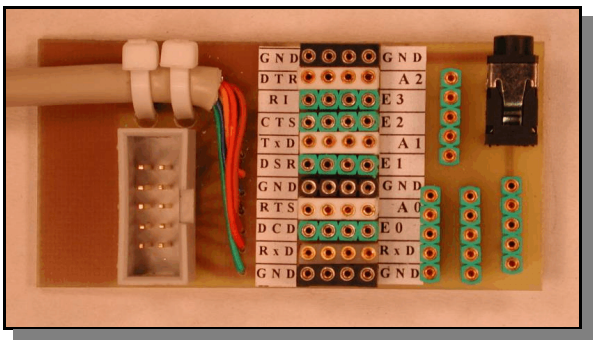


Abb. A2

Man beachte: Der COM-Anschluss eines PCs benutzt das RS232-Protokoll (vgl. S. 61 ff). Deswegen muss beim MiniPC-Programm jetzt für die Schnittstelle die Option RS232 aktiviert werden. Ggf. muss auch das eine oder andere Programm an die anderen Pegel angepasst werden. Allerdings besitzen auch hier die Ausgänge eine Strombegrenzung, so dass sich auch hier LEDs direkt anschließen lassen.

USB-COM-Konverter-Kabel (RS232)

Diese Konverter-Kabel werden von zahlreichen Händlern preiswert angeboten (z. Zt. ab etwa 3 Euro inkl. Porto; Stand Juli 2021). Leider unterstützen manche dieser Konverter nicht alle COM-Signalleitungen einwandfrei. Insbesondere werden auch USB-COM-Kabel angeboten, bei denen es Probleme mit Treibern für Windows 8-10 gibt (vgl. auch meinen Forums-Beitrag http://www.forum.g-heinrichs.de/viewtopic.php?t=85). Man beachte, dass auch diese Kabel Signale mit dem RS232-Standard liefert.

Benutzung mit MiniPC

Das Programm MiniPC muss auf den Einsatz von Experimentierboards vorbereitet werden. Dazu gibt man über das *Bearbeiten-Menü* die Nummer der benutzten COM-Schnittstelle ein (vgl. S. 52). Soll obendrein die Interruptfähigkeit von MiniPC ausgenutzt werden, muss diese Eigenschaft über *Bearbeiten - Interrupt* aktiviert werden; dabei kann gewählt werden, ob der Interrupt durch eine negative Flanke (H -> L) oder durch eine positive Flanke (L -> R) ausgelöst werden soll.

Virtuelles COM-PORT

Auch wenn die USB-Buchse für die Datenübertragung benutzt wird, so wird dieser Anschluss von Windows aus als COM-Anschluss angesehen. Dies kann man mit Hilfe des Geräte-Managers auch erkennen: In Abb. A3 sehen wir, dass unser FT232-Konverter unter der Rubrik COM & LPT aufgelistet ist. Der Grund ist: Der Baustein arbeitet mit einem speziellen Treiber zusammen, welcher dem Windows-System eine COM-Schnittstelle vorgaukelt. Tatsächlich arbeitet dann auch unser MiniPC-Programm mit Befehlen, welche sich auf eine COM-Schnittstelle beziehen. Mit anderen Worten: Sowohl MiniPC als auch die benutzten Platinen arbeiten mit COM-Signalen; nur zwischendurch werden diese Signale per USB übermittelt. Ähnliches gilt auch für die anderen Konverter.

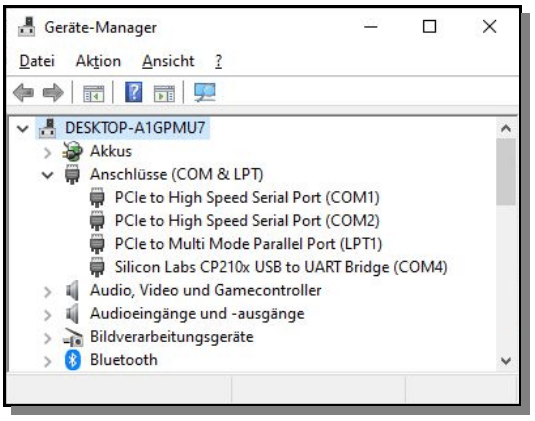


Abb. A3

Noch eine Bemerkung zur COM-Nummer. In Abb. A3 ist diese Nummer angegeben (Nr. 4). Sollte diese Nummer größer als 4 sein, muss sie geändert, weil MiniPC nur mit COM-Nummern zwischen 1 und 4 arbeiten kann. In dem Fall von Abb. A3 klickt man doppelt auf den CP210x-Eintrag; es öffnet sich ein neues Fenster. In diesem klickt man nun auf die Lasche *Anschlusseinstellungen*; wieder öffnet sich ein neues Fenster. Dort klicken wir auf die Schaltfläche *Anschlusseinstelltungen*. Abermals öffnet sich ein neues Fenster, und in diesem können wir endlich dem CP2102-Baustein eine neue COM-Nummer zuweisen.

OBJ-Code vom MiniPC auf den Kosmos CP1 übertragen

Der Kosmos CP1 war ein Lerncomputer wurde zu Beginn der 80er Jahr vom Kosmos-Verlag entwickelt und in der Reihe der Elektronikbaukästen vertrieben. Unser Simulationsprogramm MiniPC basiert auf dem CP1, liefert aber eine Reihe von zusätzlichen Funktionen. Beispielsweise wurde beim CP1 kein Assembler zur Verfügung gestellt; außerdem ließen sich Programme nur mit Hilfe eines zusätzlichen Adapters (CP2) auf einem Kassettenrekorder speichern.

Abb. A4: Kosmos-CP1

Wer stolzer Besitzer eines solchen CP1 ist, kann Programme, welche mit MiniPC erstellt wurden, mit Hilfe des Experimentierboards auf den CP1 hochladen.

Folgende Schritte sind erforderlich; dabei gehen wir davon aus, dass sich das zu übertragende Programm schon im RAM des MiniPC befindet und die Schnittstelle zum Experimentierboard schon aktiviert worden ist.

1. Experimentierboard mit dem CP1 verbinden

Experimentierboard (TTL)	**CP1**
GND	Masse
RTS	Pin 8 von Port PA1

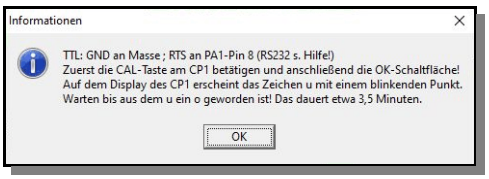


Abb. A5

2. CP1 mit Strom versorgen

3. Im Menü Bearbeiten die Option Upload wählen; es erscheint das Informationsfenster aus Abb. A5.

4. Wie im Fenster angegeben zuerst die CAL-Taste am CP1 und anschließend die OK-Schaltfläche betätigen. Auf dem Display des CP1 erscheint das Zeichen **u** mit einem blinkenden Punkt. Nun muss man warten bis aus dem **u** ein **o** geworden ist! Das dauert etwa 3,5 Minuten.

Zugegeben: Das ist eine lange Zeit, wenn man bedenkt, dass hier nur 2 · 128 Byte übertragen werden. Aber zum Speichern und Laden eines Programms musste das ursprünglich benutzte CP2-Modul die einzelnen Bits in Tonsignale umwandeln (bzw. umgekehrt), die je nach Zustand (1 oder 0) eine unterschiedliche Frequenz hatten. Immerhin kann man sich nun über die Upload-Funktion das mühselige Eintippen der Befehle auf dem CP1 ersparen.

Index