Mikroprozessor

Frank Pilhofer Kay Römer

24. Februar 1997

Zusammenfassung

In diesem Versuch werden verschiedene Programme auf einem einfachen mikroprozessorgesteuerten System geschrieben und getestet. Dabei wird ein Einblick in die Technologie gewonnen, die auch heutige Computersysteme antreibt, aber bei modernen Systemen hinter einem Wald undurchschaubarer hochspezialisierter Komplikationen versteckt ist.

1 Einleitung

Im Versuch „Digitalelektronik“ wurden aus einfachen logischen Gattern größere Schaltungen aufgebaut; unter anderem wurden Addierer und andere logische Operationen untersucht. Mit diesen Mitteln lassen sich beliebig komplexe Funktionen implementieren, doch ein Nachteil ist dabei immer die Unflexibilität. Wird die Aufgabe abgeändert, so muß die Schaltung unter Umständen von Grund auf neu aufgebaut werden.

Eine erste Erleichterung waren elektronische Rechenmaschinen, aus heutiger Sicht mit Taschenrechnern vergleichbar. Vor dem Zeitalter des Transistors zunächst aus Relais, später aus Röhren aufgebaut, konnten auf Zahlen im „Akkumulator“ Operationen ausgeführt werden. Der Operator konnte an der Konsole mit diversen Hebeln eine von mehreren Elementaroperationen auswählen und diese dann mit weiteren Hebeln ausführen lassen.

In den dreißiger Jahren ging Konrad Zuse diese monotone Arbeitsweise auf den Nerv – bei wiederholten Rechnungen, z.B. dem Ausrechnen einer Funktion mit verschiedenen Parametern, mußten die komplexen Handgriffe in der gleichen Reihenfolge wieder und wieder ausgeführt werden; eine nicht nur anstrengende, sondern auch fehlerträchtige Aufgabe. Zur Erleichterung baute er seine *programmgesteuerte* Rechenmaschine „Z1“, den ersten Computer. Anstelle jeden Rechenschritt manuell zu veranlassen, wurden Befehle von einem Lochstreifen eingelesen. War erst einmal eine Befehlsfolge, ein „Programm“, geschrieben, konnte es beliebig oft wiederholt werden.

Im Laufe der Jahre wurden zunächst die Rören zu Transistoren und später zu integrierten Schaltkreisen, aber an dem grundsätzlichen Prinzip hat sich nichts geändert. Heutige Mikroprozessoren laden Befehle eines Programms aus dem Speicher und bearbeiten ihre Registerinhalte entsprechend. Daten können aus dem Speicher in die Register geladen und Ergebnisse in den Speicher zurückgeschrieben werden. Obwohl die Recheneinheit selber nur relativ einfache Befehle zur Verfügung stellt, können mit entsprechender Programmierung beliebig komplexe Aufgaben erledigt werden.

2 Grundlagen

2.1 Der Mikroprozessor

Prozessoren werden heute als integrierte Schaltungen verkauft und sind von außen nur ein schwarzer Kasten mit vielen Beinen, denen man ihren inneren Aufbau nicht ansieht. Doch auch intern lassen sich verschiedene Bereiche unterscheiden. Abbildung 1 zeigt ganz allgemein den internen Aufbau eines einfachen Prozessors.[1] Die kleinen grauen Zellen sind Register, die einen Wert speichern können. Neben den explizt dargestellten internen Registern enthält das „Register File" `rf` die „öffentlichen" Registern, die für Rechnungen verwendet werden können. Die *Arithmetisch-Logische Einheit* `ALU` führt die eigentlichen Operationen durch.

Auch die Befehle, die ein Prozessor verarbeiten kann, lassen sich in kleinere Schritte, in „Mikrooperationen" unterteilen. In der ersten Phase, dem *Instruction Fetch*, wird der nächste Befehl aus dem Speicher in das Befehlsregister `ir` geladen. Im nächsten Schritt, dem *Instruction Decode*, wird dieser Befehl untersucht, und die gewünschten Operanden werden in die arithmetischen Register `a` und `b` geladen. Dritter Schritt ist die Ausführung, die *Execution*, in der die ALU die Operanden auf gewünschte Weise verknüpft. Schließlich wird im *Writeback* das Ergebnis an die gewünschte Adresse abgespeichert.

Nicht für alle Befehle müssen alle Phasen durchlaufen werden, und auch können die Phasen unterschiedlich lang sein – so sind Speicherzugriffe aufwendiger als Operationen, die nur Register bearbeiten. Manche Befehle können recht schnell in einem oder zwei Taktzyklen ausgeführt werden, während für andere fünf oder mehr benötigt werden.

2.2 Der Mikrocomputer

Der Mikroprozessor ist das Herz eines Computers, benötigt aber noch eine Reihe externer Schaltungen, insbesondere den Speicher, aus dem sich der Prozessor mit Befehlen und Daten versorgt. Zu diesem Zweck besitzt der Prozessor zwei „Busse"[2]; während auf dem ersten die Adresse einer Speicherzelle übertragen wird, kann auf dem zweiten das Datum in dieser Zelle gelesen oder geschrieben werden.

Insbesondere beim Einschalten des Computers ergibt sich aber ein Problem, denn der Speicher ist leer – in diesem Fall wäre unser Prozessor recht orientierungslos. Zu diesem Zweck ist üblicherweise ein Teil des Speichers als nur-lesbar ausgelegt („Read-Only Memory", ROM), in dem gewisse Befehlsfolgen fest kodiert sind, die auch beim Ausschalten nicht verloren gehen. Mit diesen Befehlen kann das System erst einmal initialisiert werden.

Dadurch ergibt sich gleich das nächste Problem. Das ROM ist physikalisch vom Rest des beschreibbaren Hauptspeichers („Random Access Memory", RAM) getrennt, der Prozessor besitzt aber nur den einen Speicherbus. Als Lösung wird zwischen Prozessor und Speicher eine „Memory Management Unit" (MMU) geschaltet, die die Adresse auf dem Speicherbus auswertet und an den richtigen Empfänger weiterleitet. Das können einerseits Speicherbausteine

[1] Die Grafik entstammt der Vorlesung von Professor Eveking.

[2] Bus: Mehrere Datenleitungen, auf denen die Bits als „Passagiere" zwischen zwei oder mehreren Bausteinen transportiert werden können

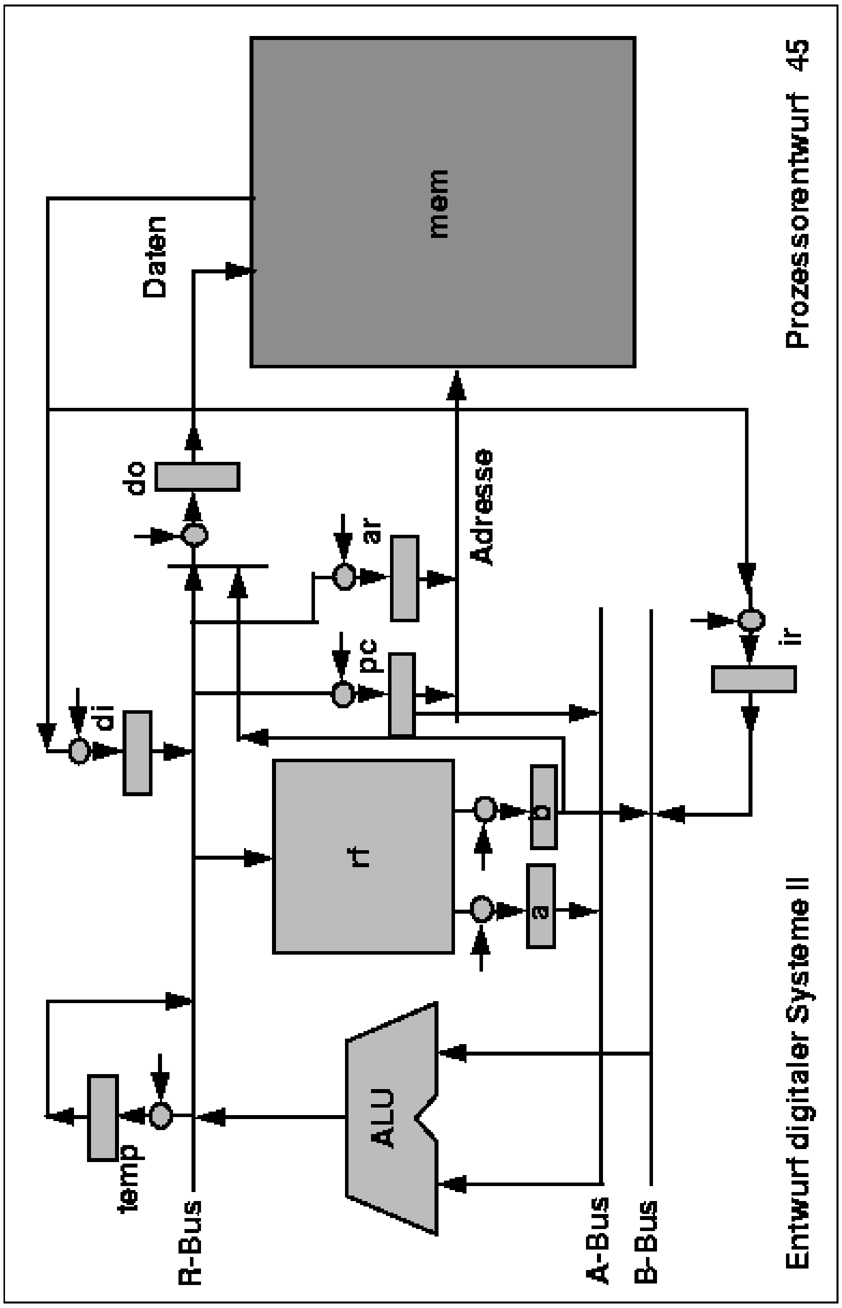

Abbildung 1: Interner Aufbau eines Prozessors

(RAM oder ROM) sein, aber auch Bausteine zur Eingabe und Ausgabe. Verschiedene externe Vorgänge können über die MMU als sogenannte „Ports“ in den Speicher „eingeblendet“ werden.

Ein Beispiel ist die angeschlossene Tastatur, deren kodierte Tastendrücke über eine bestimmte Speicheradresse gelesen werden können. Andere Speicherzellen in unserem Testsystem werden z.B. auf Siebensegmentanzeigen oder einen AD-Wandler abgebildet. Die Ausrüstung mit weiteren Elementen bleibt der Phantasie überlassen, von einem Diskettenlaufwerk bis zur Kaffeemaschine ist alles denkbar.

Aber auch die softwareseitige Ausstattung, das „Betriebssystem“ im ROM, darf nicht vernachlässigt werden. Nach der Initialisierung des Systems in einen betriebsfähigen Zustand muß es eine Möglichkeit bieten, benutzerdefinierte Programme in den Speicher zu laden und auszuführen. Eine Möglichkeit ist, sofort ein Programm von einem externen Medium wie einer Diskette zu lesen.

Das Betriebssystem unseres Testrechners beginnt nach der Initialisierung, Tastendrücke von der angeschlossenen Tastatur zu lesen und verschiedene Kommandos zu bearbeiten. Auf diese Weise können über die Tastatur weitere Programme in den Speicher eingegeben und dann ausgeführt werden.

Heutzutage werden mit zunehmender Miniaturisierung der Elemente teilweise auch ganze Mikrocomputer inklusive Prozessor, einer Speicherverwaltung und einem kleinen Speicher in einem einzigen Gehäuse untergebracht, so daß man sich einen großen Teil des externen Schaltungsaufwandes sparen kann.

3 Das Testsystem

3.1 Der 68HC11 Prozessor

Der in diesem Versuch zur Verfügung stehende Mikrocomputer wird von einem Mikroprozessor Marke „Motorola 68HC11“ angetrieben und repräsentiert ungefähr die bezahlbare Technik Anfang der achtziger Jahre. Dennoch ist die Architektur nicht direkt veraltet; heute kann solch ein Prozessor für ein paar Mark hergestellt werden, und für kleinere Aufgaben sind sie noch immer gut genug.

In der Grafik 2 ist der Prozessor mit seinen Anschlüssen abgebildet.[3] An der linken und oberen Seite werden die zwei Speicherbusse angeschlossen, der Adreßbus `MEMADDR` und der Datenbus `MEMDATA`. Über die 16 Leitungen des Adreßbusses können $2^{16} = 65536$ Speicheradressen angesprochen werden, wobei jede der Speicherzellen 8Bit speichern kann.

Der Prozessor unterstützt einen recht ausführlichen Befehlssatz mit über 300 einzelnen Befehlen und zeigt sich als Mischung zwischen einem 8bit und einem 16bit-System. Die beiden arithmetischen Register `A` und `B` sind nur 8bit groß, können aber je nach Bedarf auch gemeinsam als 16bit-Register `D` angesprochen werden. Die meisten Befehle stehen sowohl für die 8bit als auch für das 16bit-Register zur Verfügung. Weiter existieren noch die zwei 16bit Indexregister `X` und `Y`, auf denen nur einfache Operationen wie Inkrementierung durchgeführt werden können, die aber zeitweise auch zum kurzen Zwischenspeichern von Operanden praktisch sind.

[3] Grafik (c) Johannes Kepler Universität Linz, Fachbereich Mikroelektronische Systeme.

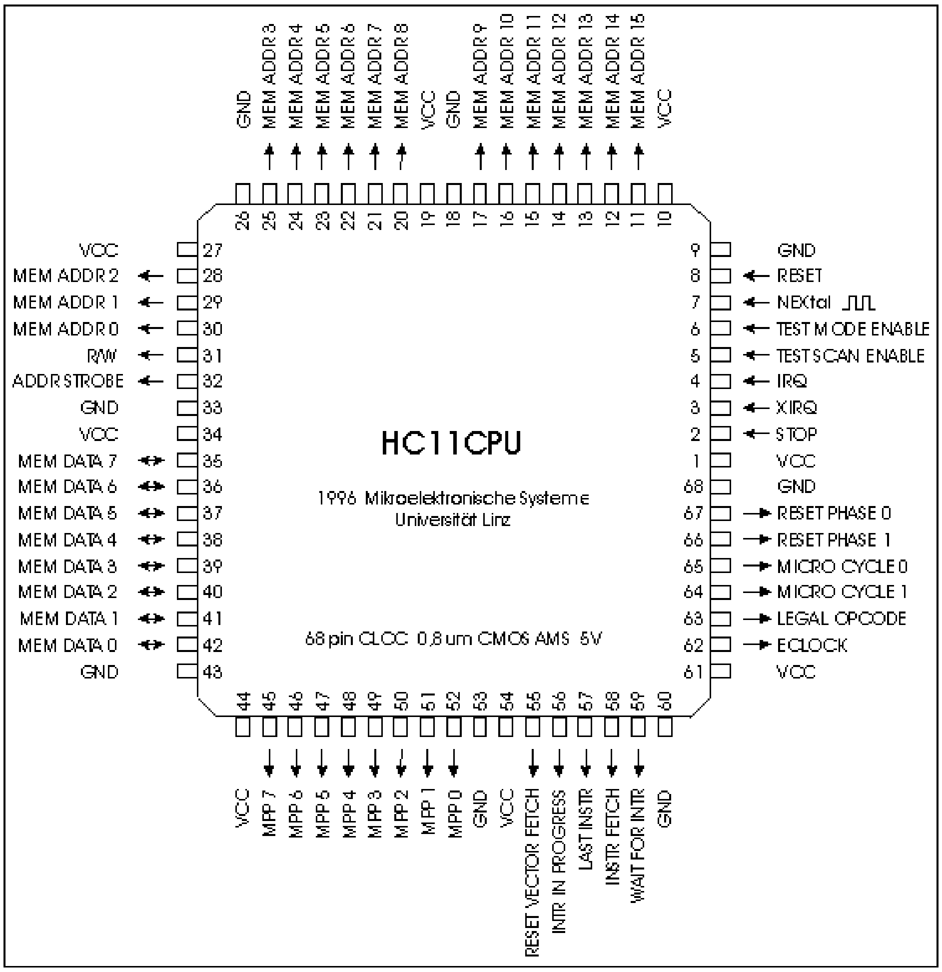

Abbildung 2: Die 68HC11 CPU

Es gibt auch noch ein Statusregister, in dem verschiedene Zustände angezeigt werden. Für uns von Interesse sind dabei Flags, die den Überlauf oder Unterlauf einer Operation anzeigen (z.B. daß eine Subtraktion einen negativen Wert zum Ergebnis hatte). Das Register wird aber dabei selten explizit angesprochen, sondern nur als Grundlage für bedingte Sprünge verwendet.

Als arithmetische Befehle stehen die Grundrechenarten und Vergleichsoperatoren zur Verfügung, wobei auch negative Zahlen im Zweierkomplement dargestellt werden können.

Betrieben wird der Prozessor mit einem Taktsignal von 2MHz, das aber intern noch durch vier geteilt wird, so daß die Taktfrequenz bei 500kHz liegt, ein Taktzyklus also 2 Mikrosekunden dauert. Da auch hier die einzelnen Befehle in verschiedene Mikrooperationen zerlegt werden, benötigt ein Befehl insgesamt zwischen zwei und sieben Taktzyklen (mit den Ausnahmen Multiplikation 10, Division 41).

3.2 Peripherie

Direkt am Speicherbus des Prozessors ist die Speicherverwaltungseinheit angeschlossen, die hier aus einem einfachen Adreßdekoder besteht, der entweder ein 32kB großes RAM oder ein 16kB großes ROM anspricht. Weiter stehen noch eine Reihe Ports zur Verfügung, die an bestimmten Speicheradressen ansprechbar sind.

Einer der Ports stellt die Verbindung zu einem intelligenten Terminal her, über das Tastendrücke gelesen werden können. „Intelligent," da es sich gleichzeitig um die Bildschirmausgabe kümmert und so ein extra Baustein zum Ansteuern einer Bildschirmeinheit eingespart werden kann. Leider enthielt die Dokumentation keinerlei Hinweise, wo oder wie das Terminal angesteuert werden konnte.

Weiterhin existieren zur Ausgabe drei Siebensegmentanzeigen. Durch das Abspeichern eines Wertes an bestimmten Speicheradressen können die einzelnen Segmente eingeschaltet werden. Ein weiterer Port steuert acht Leuchtdioden sowie acht Ausgänge an der Gehäuseseite, an die verschiedene Geräte angeschlossen werden können. Auch für eingehende Daten existiert ein Port mit acht Datenleitungen, die entweder digital, oder über einen in die CPU integrierten AD-Wandler sogar analog ausgewertet werden können.

Ein zusätzlicher „Port", der Resetschalter, wurde während der Versuchsdurchführung zum Glück nur relativ selten benötigt.

3.3 Betriebssystem

Das System ist im ROM mit einem für derartige Verhältnisse recht komfortablen Betriebssystem ausgestattet. Es erlaubt die Eingabe und Ausgabe des Maschinenprogramms in „Mnemonischer Form", so daß nicht der Befehlscode direkt eingetippt werden muß, sondern Befehle können in einigermaßen menschenverständlicher Form als Funktion mit Parameter eingegeben werden, diese Eingabe wird dann vom Betriebssystem entsprechend interpretiert. Das spart insbesondere deshalb Nerven, weil Befehle je nach verwendeter Addressierungsart eine unterschiedliche Kodierung besitzen.

Sehr bequem ist auch die Fähigkeit, Maschinenprogramme im Einzelschrittverfahren abzuarbeiten und dabei ständig die Registerinhalte auszugeben. Auf diese Weise können Fehler einfach dingfest gemacht werden.

Der einzige Nachteil war, daß keine Sprungmarken gesetzt werden konnten; bei der Programmierung von Sprüngen mußten daher zunächst die Zieladressen leer gelassen werden und dann in einem zweiten Durchlauf, wenn die Zieladresse bekannt ist, eingetragen werden.

Mit diesen Voraussetzungen können wir uns nun auf die Programmierung „unseres“ Systems stürzen.

4 Programmierung

Im folgenden werden unsere Lösungen der vorgegebenen Programmierprobleme präsentiert, zunächst die 16bit-Multiplikation, der Tongenerator und die Ansteuerung der Siebensegmentanzeige. Von den anschließenden drei Aufgaben sollten zwei bearbeitet werden. Dabei haben wir uns die Fußgängerampel geschenkt und stattdessen den Schrittmotor angesteuert und das Voltmeter programmiert.

Auf die Anfertigung von Flußdiagrammen haben wir verzichtet und stattdessen den Programmablauf in Textform erklärt und den Programmtext mit entsprechenden Anmerkungen versehen. Zudem sind die Programme übersichtlich genug um ohne aufwendige Dokumentation auszukommen.

4.1 16bit-Multiplikation

Als erstes Programm soll eine Routine geschrieben werden, die zwei 16bit-Zahlen multipliziert. Der Prozessor selber bietet nur einen Befehl zur Multiplikation zweier 8bit-Zahlen.

Zu diesem Zweck teilen wir unsere beiden Operanden X und Y in zwei Teile aus den oberen bzw. den unteren 8bit, die wir mit dem existierenden Multiplikationsbefehl verknüpfen können. Die Einzelergebnisse können dann so addiert werden, daß sich das richtige Ergebnis ergibt.

Bezeichnen wir die oberen 8bit mit dem Index h und die unteren 8bit mit dem Index l, so können wir schreiben $X = X_h \cdot 2^8 + X_l$ (entsprechend Y). Anschließend können wir wie folgt multiplizieren:

$$\begin{aligned} X \cdot Y &= (X_h \cdot 2^8 + X_l) \cdot (Y_h \cdot 2^8 + Y_l) \\ &= (X_h \cdot Y_h) \cdot 2^{16} + (X_h \cdot Y_l) \cdot 2^8 + (X_l \cdot Y_h) \cdot 2^8 + (X_l \cdot X_l) \cdot 2^0 \end{aligned}$$

Der Schritt war recht einfach. Bei der Ausführung ist nun darauf zu achten, daß ja das Ergebnis jeder einzelnen 8bit-Multiplikation eine 16bit-Zahl ist, und die Ergebnisse entsprechend „verzahnt“ werden müssen.

Unser Programm erwartet die beiden Operanden in den Indexregistern, und gibt hinterher auch das Ergebnis in diesen beiden Registern zurück (in `X` die oberen, in `Y` die unteren 16bit). Aufgrund der begrenzten Zahl von arithmetischen Registern wird temporär Zwischenspeicher an der Adresse \$1800 verwendet:

Operand #1		Operand #2		Ergebnis			
1800	1801	1802	1803	1804	1805	1806	1807
15..8	7..0	15..8	7..0	31..24	32..16	15..8	7..0
X_h	X_l	Y_h	Y_l	Z_{hh}	Z_{hl}	Z_{lh}	Z_{ll}

Gebrauch macht unser Programm von den 16bit Lade- und Speicheroperationen und der Tatsache, daß der Prozessor dabei nach dem *Big-Endian*-Prinzip arbeitet: das höherwertige Byte eines 16bit-Wertes wird in die erste Speicherzelle geschrieben, das niederwertige in die darauf folgende.

Ein weiterer Trick ist, daß der Prozessor 16bit-Operationen auch auf ungerade Speicheradressen ausführen kann – neuere Prozessoren verlangen aus technischen Gründen die Verwendung gerader Adressen (korrektes „Alignment“). Hier aber machen wir uns das Leben einfach.

Unser Programm multipliziert zunächst die beiden oberen Bytes der Argumente und speichert das Resultat vorläufig als obere 16bit des Ergebnisses ab. Anschließend werden die beiden unteren Bytes multipliziert und als untere 16bit abgespeichert. Als nächstes wird „über Kreuz“ multipliziert. Das Resultat muß auf die bereits von den vorherigen Operationen beschriebenen mittleren 16bit des Ergebnisses addiert werden. Bei dieser Addition kann ein Überlauf stattfinden, bei dem das Überlaufbit im Statusregister (das *Carry*) gesetzt wird. Ist das der Fall, inkrementieren wir das oberste Byte des Ergebnisses; ist aber das Carry nicht gesetzt, so hüpfen wir über die Inkrementierung hinweg.

Das Ergebnis der Multiplikation zweier 16bit-Zahlen hat maximal 32bit, es kann also kein Überlauf im höchstwertigen Byte auftreten.

Zum Schluß laden wir das Ergebnis aus dem Hilfsspeicher in die Indexregister und springen aus dem Unterprogramm zurück. Das fertige Programm ist in Abbildung 3 abgedruckt.

4.2 Tongenerator

Das nächste Programm soll auf einem angeschlossenen Lautsprecher einen Ton von 440Hz ausgeben und ist damit auch eine einfache Demonstration für die Ansteuerung der Ports und externer „Hardware“.

Der Lautsprecher Marke Nervensäge kann direkt an eine Leitung des „Port F“ angeschlossen werden; der angeschlossene Leistungstreiber liefert eine ausreichende Leistung um die Membran des Lautsprechers hörbar zu bewegen. Aus der Dokumentation kann entnommen werden, daß der Port an der Adresse \$1043 eingeblendet ist. Unser Programm muß also in ensprechenden Zeitabständen den Wert dieser Speicheradresse umpolen und damit die Membran des Lautsprechers vorwärts bzw. rückwärts schwingen lassen. Dabei entsteht ein hörbares Rechtecksignal der gewählten Frequenz.

Um einen Ton von 440Hz zu produzieren, müssen wir den Lautsprecher 880 mal pro Sekunde umpolen. Unser Ziel ist also eine Warteschleife, die exakt 1/880 Sekunde lang wartet. Das Herz unseres Prozessors schlägt mit 500kHz, ein Taktzyklus ist also 2μs lang. Eine Division ergibt, daß wir alle 2272.73 Taktzyklen umpolen müssen, die Rechnung also leider nicht aufgeht.

Unsere Warteschleife müssen wir auch etwas kürzer planen, da die Umpolung des Ports selber ja auch noch seine Zeit benötigt. Auch wollen wir einen weiteren Zähler mitführen, der unsere Umpolungen zählt und nach einer Sekunde das Programm beendet. Als nächstes greifen wir daher zur Prozessordokumentation, und darin nach der Taktzyklentabelle, in der exakt angegeben ist, wieviel Taktzyklen jeder einzelne Befehl benötigt.

Die einfachste Warteschleife, wie im folgenden dargestellt, benötigt drei Taktzyklen zur Vorbereitung (Laden der Wiederholungen) und dann pro Durchlauf des Schleifenkörpers weitere 6 Taktzyklen.

```
;
; Multiplikation zweier 16-bit Zahlen
;
; Operanden in X, Y, Ausgabe in XY (HI/LO)
; Verwendet Zwischenspeicher bei 1800:
; 1800  Operand #1 HI/LO       (Xh/Xl)
; 1802  Operand #2 HI/LO       (Yh/Yl)
; 1804  Ergebnis obere 16 bit  (Zhh/Zhl)
; 1806  Ergebnis untere 16 bit (Zlh/Zll)
;

2000            stx  1800
2003            sty  1802

2007            ldaa 1800
200a            ldab 1802
200d            mul              ; Xh*Yh
200e            std  1804

2011            ldaa 1801
2014            ldab 1803
2017            mul              ; Xl*Yl
2018            std  1806

201b            ldaa 1800
201c            ldab 1803
2021            mul              ; Xh*Yl
2022            addd 1805        ; addiere Zhl Zlh
2025            bcc  202a        ; Bei Uebertrag:
2027            inc  1804        ; Inkrementiere Zhh
202a    skip1:  std  1805

202d            ldaa 1801
2030            ldab 1802
2033            mul              ; Xl*Xh
2034            addd 1805        ; addiere Zhl Zlh
2037            bcc  203c        ; Bei Uebertrag:
2039            inc  1804        ; Inkrementiere Zhh
203c    skip2:  std  1805

203f            ldx  1804
2042            ldy  1806
2046            rts
```

Abbildung 3: Multiplikation zweier 16bit-Zahlen

```
        ldx  #count     ; 3 Taktzyklen
iloop:  dex             ; 3*count
        bne  iloop      ; 3*count
```

Das Umpolen des Ports (dessen Wert wir im Register `A` mitführen) sowie dessen Abspeichern benötigt 6 Taktzyklen. Der zweite Zähler, mit dem wir nach einer Sekunde abbrechen, schlägt mit weiteren 7 Taktzyklen zu Buche, der anschließende Rücksprung an den Anfang der Schleife noch einmal drei:

```
oloop:  dey             ; 4
        beq  ende       ; 3
        coma            ; 2  Lautsprecher
        staa 1043       ; 4  Umpolen
        ....            ; (innere Schleife)
        jmp  oloop      ; 3
```

Der Zeitverbrauch dieser Befehle und die Vorbereitung der Warteschleife, insgesamt 19 Taktzyklen, ist unveränderbar, aber unsere innere Warteschleife können wir anpassen. Von den 2272.73 Taktzyklen bleiben uns noch 2253.73. Wenn wir die 6 Taktzyklen der inneren Schleife 375 mal durchlaufen, haben wir 2250 verbraucht und haben noch 3.73 Taktzyklen übrig. Würden wir die Schleife eins weiter laufen lassen, so wäre sie wiederum um 2.27 Taktzyklen zu lang.

Zur Hilfe kommt hier der recht unbedeutende Befehl `NOP`, der ganz einfach nichts tut, dafür aber zwei Taktzyklen braucht. Mit zwei dieser Befehle kommt die gesamte Schleife auf immerhin 2273 Taktzyklen, was schon recht nah am gewünschten Ergebnis liegt.

Das fertige Programm zeigt Abbildung 4. Von den 2273 Taktzyklen können wir nun wieder auf die Zeit zurückrechnen und erhalten eine Frequenz von 439.95Hz, was doch ununterscheidbar nahe am gewünschten Ergebnis liegt.

Ein absoluter Genauigkeitsfanatiker könnte die Annäherung noch verbessern, indem er vier dieser Schleifen hintereinander anordnet, von denen drei 2273 Taktzyklen und eine 2272 Taktzyklen benötigt – damit läge man dann bei 439.996Hz.

Eine andere Erweiterung des Programms wäre, die statische Warteschleife so zu ändern, daß der Verzögerungswert aus einer Tabelle gelesen wird, deren Werte mehr oder weniger den bekannten Notenfrequenzen entsprechen, womit dann immerhin recht einfache Lieder gespielt werden könnten.

Doch ein audiophiles Ergebnis wird man mit dem Quäker auf diese Weise nicht erreichen; die Ansteuerung beschränkt uns auf die Ausgabe einstimmiger Rechteckfrequenzen. Den Autoren sind zwar Programme bekannt, die mit solch einem digital angesteuerten Lautsprecher echte „Musik“ ausgeben, aber das zugrundelegende Prinzip ist uns unbekannt.

Höherwertige Audiohardware steuert den Lautsprecher jedenfalls nicht digital an, sondern über einen DA-Wandler, mit dem die Membran des Lautsprechers genau „positioniert“ werden kann. Mit dieser Voraussetung ist dann eine Ausgabe digitalisierter Töne und aufgezeichneter Sprache ein leichtes.

4.3 Siebensegmentanzeige

Dieses Programm soll auf den in den Mikrocomputer integrierten Siebensegmentanzeigen eine beliebige hexadezimale Zahl ausgeben, die dem Programm in

```
;
; "Tongenerator"
;
; Gebe eine Sekunde lang einen Ton der Frequenz 440Hz aus.
; Wir steuern den Lautsprecher an Port F (1043)
;
; Bei einer Taktfrequenz von 500kHz bedeuten 440Hz 4545.45 Taktzyklen,
; wir muessen also alle 2272.73 Taktzyklen den Lautsprecher umpolen.
;
2100           ldaa #00
2102           ldy  #0370      ; 880 mal Umpolen macht 1sec
;
; Innere Schleife benoetigt 6 Taktzyklen. Wenn wir sie 375 mal
; durchlaufen, haben wir 2250 Taktzyklen verbraucht. Dazu
; kommen 7 zum Zaehlen der Umpolungen, 6 zum Umpolen des Ports,
; 3 zum Vorbereiten der Schleife und 3 zum Ruecksprung zum
; Anfang der Schleife, macht 2269. Weitere 4 Zyklen verbraten
; wir mit NOPs, also insgesamt 2273, was einer Frequenz von
; 439.947 Hz entspricht.
;                              ; Taktzyklen
2106    oloop: dey             ; 4
2108           beq  2119       ; 3
210a           coma            ; 2  Lautsprecher
210b           staa 1043       ; 4  Umpolen
;
; Verzoegerungsschleife 3 plus 375 mal 6 Taktzyklen
;
210e           ldx  #0177      ; 3
2111    iloop: dex             ; 3
2112           bne  2111       ; 3
;
; Feinkorrektur
;
2104           nop             ; 2
2105           nop             ; 2
2116           jmp  2106       ; 3
;
; Zuruecksetzen
;
2119           ldaa #00
211b           staa 1043
211e           rts
```

Abbildung 4: Ausgabe eines 440Hz-Tones auf einem Lautsprecher an Port F einem Register übergeben wird. Die drei verfügbaren Anzeigen können über die Speicheradressen \$1040, \$1041 und \$1042 (von links nach rechts) angesprochen werden. Wir brauchen hier nur zwei der Siebensegmentanzeigen und wählen die rechten zwei.

Die erste Vorarbeit besteht darin sich zu überlegen, bei welcher Zahl welche Segmente der Anzeige leuchten sollen. Das Ergebnis wird in einer kleinen Tabelle festgehalten, die wir im Speicher in den Adressen \$2300 bis \$230f bereit halten.

Die Grobstruktur des Programms ist, daß zunächst die oberen vier Bits des Arguments, später die unteren vier Bits isoliert werden. Für diese Bits wird der entsprechende Wert aus der vorgefertigten Tabelle geladen und in den gewünschten Port geschrieben.

Das größte Problem dabei war, daß wir die Isolierung der Bits in einem arithmetischen Register durchführen mußten, anschließend diesen Wert zum indizierten Tabellenzugriff in einem Indexregister brauchten. Um die Lage zu verkomplizieren ist unser arithmetisches Register `B` 8-bittig, das Indexregister aber 16-bittig, und es gibt keine Möglichkeit, den Wert des einen in das andere zu übertragen. Ideal wäre ein indizierter Zugriff `ldaa tabelle,x`, was aber ebenfalls nicht verfügbar war (die Basisadresse darf beim indizierten Zugriff nur 8-bittig sein).

Als kleiner Workaround wird der `xgdx`-Befehl verwendet, der den Inhalt des 16bit-Registers `D` mit `X` tauscht. Dabei ist `D` der Zusammenschluß aus `A` und `B`; genaugenommen stellt `A` die oberen 8 bit und `B` die unteren 8 Bit. Nachdem wir so die gewünschten 4 Bit in `B` isoliert haben, laden wir die Basisadresse der Tabelle in `A` bzw. für unseren Zweck in die oberen 8 bit von `D` und haben nach dem `xgdx` die gewünschte Zieladresse im Indexregister.

Ist dieses Problem erst einmal bewältigt, ist der Rest des Unterprogramms recht einfach. Das gesamte Programm ist in Abbildung 5 abgedruckt. Eine Vorbereitung für kommende Aufgaben ist, daß das Unterprogramm sämtliche Registerinhalte zunächst speichert und hinterher wieder unverändert restauriert, so daß ein aufrufendes Programm nicht um seine Register fürchten muß.

4.4 Ausgabe von Dezimalzahlen

Als weiterer Vorgriff auf kommende Aufgaben sollen noch zwei weitere Unterroutinen geschrieben werden. Die erste soll einen Registerwert nicht hexadezimal, sondern dezimal auf allen drei Siebensegmentanzeigen ausgeben. Hier stellt sich das Problem, daß sich die Bits nicht mehr reibungslos auf die verschiedenen Anzeigen aufteilen. Stattdessen müssen auf anderen Wegen die Werte der einzelnen Dezimalziffern ermittelt werden.

Am Anfang des Programms werden die „Hunderter“ behandelt. Von dem Register werden solange jeweils dezimal 100 abgezogen, bis ein Unterlauf stattfindet. Hierfür braucht aber keine Schleife geschrieben werden, denn wir wissen, daß wir höchstens zweimal 100 abziehen können. Mit dem ermittelten Wert für die Hunderter können wir wie im vorherigen Programm (Abbildung 5) auf die Tabelle zugreifen und damit die Anzeige füttern.

Für die unteren beiden Ziffern sparen wir uns einen großen Teil der Mühe, indem wir das bereits geschriebene Programm aufrufen. Doch dafür müssen wir erst den gewünschten dezimalen Wert nach BCD wandeln, so daß die oberen vier Bits des Registers die „Zehner“ und die unteren vier Bits die „Einer“ wiedergeben, ansonsten würde die Unterroutine ja einen falsch interpretierten Wert

```
;
; Siebensegmentanzeige
;
; Gebe die Zahl im Akku B hexadezimal auf Siebensegmentanzeige aus.
; Wir haben drei Anzeigen, davon benutzen wir die rechten beiden.
; Linke Anzeige ist Port 1041, rechte ist 1042.
; Die vorberechneten Konfigurationen der Anzeigen ist in 2300-230f
; abgespeichert.
;
2200            psha            ; Register retten
2201            pshx
2202            pshb

2203            lsrb            ; obere vier Bit isolieren
2204            lsrb
2205            lsrb
2206            lsrb
2207            ldaa #23        ; oberes Adressbyte
;
; In D=(A/B) ist nun die Adresse unseres Datums.
;
2209            xgdx            ; Index von (A/B) nach X
220a            ldaa 00,x       ; Datum aus Tabelle laden
220c            staa 1041       ; auf Anzeige

220f            pulb            ; Datum vom Stack restaurieren
2210            pshb
2211            andb #0f        ; untere vier Bit isolieren
2213            ldaa #23
2215            xgdx            ; wie oben
2216            ldaa 00,x
2218            staa 1042

221b            pulb            ; Register restaurieren
221c            pulx
221d            pula
221e            rts
;
; Daten
;                     _       _   _       _   _   _
;                    | |   |  _|  _| |_| |_  |_    |
;                    |_|   | |_   _|   |  _| |_|   |
;
2300            data  3f  06  5b  4f  66  6d  7d  07
;                     _   _   _                _   _
;                    |_| |_| |_|  |_   _  _|  |_  |_
;                    |_|  _| | |  |_| |_ |_|  |_  |
;
2308            data  7f  6b  77  7c  58  5e  79  71
```

Abbildung 5: Ausgabe der Zahl in B auf der Siebensegmentanzeige ausgeben.

Für die Zehner schreiben wir daher eine Schleife, die solange 10 subtrahiert, bis ein Unterlauf auftritt. Genauso oft akkumulieren wir die Zehner im anderen Register – aber diesmal in BCD-Kodierung. Auch die Einer könnten wir in solch einer Schleife bearbeiten, aber hier machen wir uns das Leben einfacher und fügen die Einer zu den Zehnern mit einem einfachen `orab` hinzu, dessen Parameter wir mit dem gewünschten Wert überschreiben.

An dieser Stelle modifiert sich das Programm also selber! Moderne Rechnerarchitekturen würden meckern, aber das Testsystem nimmt die Änderung gelassen hin. Die Alternative wäre, den Wert für die Einer irgendwo im Speicher zu platzieren und dann das `orab` mit jeder Adresse durchzuführen. Unser Vorgehen spart uns aber eine tempöräre Speicherzelle und mindestens einen Taktzyklus.

Anschließend kann direkt in das Unterprogramm 5 bei Adresse $2200 gesprungen werden um die restlichen zwei Ziffern anzuzeigen.

Abbildung 6 zeigt das Unterprogramm zur Anzeige einer Dezimalzahl, das als Nebeneffekt noch den Dezimalpunkt der ersten Siebensegmentanzeige einschaltet. Leider wurde vergessen, die Register zu retten, was dann noch in einem kleinen weiteren Hilfsprogramm (Abbildung 7) schnell nachgeholt wurde.

4.5 Schrittmotor

Hier soll als weiteres Peripheriegerät ein externer Schrittmotor angesteuert werden.

Schrittmotoren zeichnen sich dadurch aus, daß sie mit hoher Präzision vorgegebene Positionen anfahren können. Bei dem im Versuch vorhandenen Schrittmotor ist eine volle Umdrehung in 200 Schritte unterteilt, eine gewünschte Ausrichtung kann also auf zwei Grad genau angefahren werden. Hier bewegt der Motor einen Drehregler über eine Scheibe, auf der die Motorstellung abgelesen werden kann.

Aus dem Schrittmotor nach außen geführt sind vier Anschlüsse für die vier Phasen des Motors, von denen jeweils genau eine unter Strom steht. Durch geeignetes Drehen der Phase (umschalten von einer Phase auf die nächste) bewegt sich der Motor um einen Schritt vorwärts bzw. rückwärts.

Die Aufgabe ist, die Zahl im 16bit-Register `D` als vorzeichenbehaftet zu interpretieren und dementsprechend den Motor um diese Anzahl Schritte vorwärts bzw. rückwärts zu bewegen.

Wichtig ist dabei, die Anfangsphase des Motors zu kennen. Wird die aktuelle Stellung „vergessen“ und dann eine beliebige (oder statisch kodierte) Phase aktiviert, so muß dies nicht die korrekte sein – die Stellung des Motors würde mit der Zeit „verwaschen“ werden.

Daher speichern wir die Phase des Motors in einem Register ab. Vor dem Beginn eines Drehvorgangs wird die aktuelle Phase gelesen und hinterher wieder dorthin abgespeichert, so daß auch bei mehrmaligem Aufruf des Programms kein Fehler entsteht. Ebenfalls wird die derzeitige Position des Motors gespeichert. Die kann das Programm aber nicht selber herausfinden – vor dem ersten Programmaufruf muß der Motor manuell auf Nullstellung gedreht und die Register entsprechend initialisiert werden.

Nach der Überprüfung der Motorphasen und der Feststellung der richtigen Numerierung werden die vier Leitungen mit den unteren vier Bits des Ports E

```
;
; Dezimalanzeige
;
; Gebe die Zahl im Akku A dezimal auf Siebensegmentanzeige aus.
; Die obere Ziffer (Hunderter) geben wir selber aus, den Rest
; wandeln wir nach BCD und verwenden das Unterprogramm zur
; hexadezimalen Anzeige.
;
2400          ldab #00
2402          suba #64
2404          bcs  240e       ; Sprung wenn kleiner als 100
2406          incb
2407          suba #64
2409          bcs  240e
240b          incb            ; in B stehen die Hunderter
240c          suba #64
240e    u1:   adda #64        ; Korrektur nach Unterlauf
2410          psha            ; und merken
2411          pshx
2412          nop             ; aus technischen Gruenden ...
2413          ldaa #23        ; siehe Siebensegmentanzeige
2415          xgdx
2416          ldaa 00,x       ; Tabellenwert laden
2418          oraa #80        ; Dezimalpunkt anschalten
241a          staa 1040       ; Anzeige
241d          pulx
241e          pula
;
; BCD-Umwandlung. Fuer jede dez 10 in A addieren wir hex 10 in B
;
241f          ldab #00
2421          suba #0a
2423          bmi  2429
2425          addb #10
2427          bne  2421
2429    u2:   adda #10        ; Korrektur nach Unterlauf
242b          staa 242f       ; Parameter von ora modifizieren
242e          orab #00        ; Low-Nibble setzen
2430          jmp  2200       ; als hex-Zahl ausgeben.
```

Abbildung 6: Ausgabe einer Dezimalzahl auf die Siebensegmentanzeigen

```
;
; Hilfsprogramm: Dezimalanzeige von B ohne Aenderung der Register
;
2500        psha
2501        pshb
2502        pshy
2504        tba
2505        pshb
2506        jsr  2400
2509        pulb
250a        puly
250c        pulx
250d        pula
250e        rts
```

Abbildung 7: Hilfsprogramm: Ausgabe einer Dezimalzahl mit Retten der Register

verbunden. Die Phase des Motors wird dann intern durch vier Bits repräsentiert, von denen immer genau eins gesetzt sein muß. Zum Bewegen des Motors ist das Bit nach links (vorwärts) bzw. nach rechts (rückwärts) zu schieben. Fällt das Bit aus der Maske heraus, so ist es auf der anderen Seite wieder einzutragen.

Zur Vereinfachung „verdoppeln“ wir die Phase und wiederholen sie in den oberen vier Bit des Registers – auf diese Weise wird das Auffüllen der Phase mit herausgefallenen Bits durch das Carry vereinfacht.

Unsere Phase speichern wir in $1810, die derzeitige Position in $1811. Zwar führt das Unterprogramm nur Relativbewegungen aus, so daß die derzeitige Position genaugenommen irrelevant ist, der Wert wird aber in späteren Programmen verwendet.

Am Anfang des Programms testen wir zunächst, das Vorzeichen des Registers `D`. Ein Spezialfall ist die 0, dann sind wir sofort fertig. Andernfalls wird verzwiegen; wir haben unabhängige Routinen für Links- bzw. Rechtsdrehungen. Innerhalb der Routinen schieben wir den Wert zunächst in ein Indexregister, wo wir den Wert bequem herunterzählen können und laden die Register `A` und `B` mit der derzeitigen Phase und Motorstellung.

Beim Drehen nach rechts wird die Phase nach links gedreht; hier wird mit einem einfachen Addierungsbefehl ein evtl. herausgefallenes Bit rechts wieder angefügt. Anschließend wird die Motorstellung hochgezählt; ist die 200 erreicht, so wird der Zähler auf 0 zurückgesetzt. Zum Schluß wird der Zähler in `X` dekrementiert und, falls noch nicht alle gewünschten Schritte ausgeführt sind, zurück an den Anfang gesprungen. Abschließend werden Phase und Motorstellung wieder in ihre zugeordneten Speicherstellen zurückgeschrieben.

Das Drehen nach links ist ähnlich aufgebaut, nur umgekehrt. Hier werden die Bits der Phase nach rechts verschoben; fällt ein Bit heraus, so wird es mit einem `oraa` links wieder angefügt. Die Motorstellung wird dekrementiert und nach einem Unterlauf auf 199 gesetzt.

Beiden Routinen ist noch gemeinsam, daß sie mit der bereits geschriebenen Unterroutine ständig die aktuelle Motorposition dezimal auf den Siebensegmentanzeigen ausgeben (`jsr 2500`). Auch enthalten ist eine kleine Warteschleife, da

```
;
; Schrittmotoransteuerung
;
; Bewege den Schrittmotor um soviele Schritte wie in D angegeben.
; In 1810 speichern wir die Phase, in 1811 die derzeitige Position
; des Motors. Phase ist in Akku A, Position in B.
;
25fd            cpd  #0000
2601            bmi  262e       ; negativ? dann drehe links
2603            bne  262d       ; null? dann ab ans ende
;
; drehe rechtsherum
;
2605            xgdx            ; Zaehler von D nach X
2606            ldab 1811       ; derzeitige Motorstellung
2609            ldaa 1810       ; Phase
260c    rloop:  asla            ; ein Schritt nach rechts
260d            adca #00        ; Ueberlauf rechts anhaengen
260f            staa 1043       ; Motor bewegen
2612            incb
2613            cmpb #c8        ; Motor bei 200 angekommen?
2615            bcs  2619
2617            ldab #00        ; dann ist er auf Nullstellung!
2619    noov:   jsr  2500       ; Motorstellung anzeigen
261c            ldy  #0500
2620    del1:   dey             ; Verzoegerungsschleife
2622            bne  2620
2624            dex             ; Zaehler erniedrigen
2625            bne  260c       ; fertig?
2627            staa 1810       ; Phase abspeichern
262a            staa 1811       ; Motorstellung abspeichern
262d    ende:   rts
;
; drehe linksherum
;
262e    links:  xgdx            ; Zaehler von D nach X
262f            ldab 1811       ; Motorstellung
2632            ldaa 1810       ; Phase
2635    lloop:  lsra            ; ein Schritt nach links
2636            bcc  263a       ; Bit rechts rausgefallen?
2638            oraa #80        ; dann links reinschieben.
263a            staa 1043       ; Motor bewegen
263d            decb
263e            cmpb #ff        ; Unterlauf Motorstand?
2640            bne  2644
2642            ldab #c7        ; dann sind wir bei 199!
2644    noul:   jsr  2500       ; Motorstellung anzeigen
2547            ldy  #0500
254b    del2:   dey             ; Verzoegerungsschleife
254d            bne  264b
264f            inx             ; Zaehler erhoehen
2650            bne  2635       ; fertig?
2652            staa 1810       ; Phase abspeichern
2655            stab 1811       ; Motorstellung speichern
2658            rts
```

Abbildung 8: Ansteuerung des Schrittmotors der Rechner ansonsten die Phase schneller drehen würde als daß der Motor mitkommt. Der Verzögerungswert von $500 ist dabei empirisch ermittelt und sorgt für einen flotten Dreh.

Abbildung 8 zeigt das komplette Programm.

4.6 Voltmeter

Zu guter letzt darf der Mikrocomputer noch als einfaches Voltmeter mißbraucht werden. Zu diesem Zweck besitzt die CPU eine interne Schaltung, die eine unbekannte Eingabespannung mit einer Referenzspannung vergleicht; das Verhältnis zwischen beiden kann dann mit einem AD-Wandler abgefragt werden.

In diesem Fall ist als Referenzspannung 2.55V angelegt, was relativ praktisch ist. Für eine Eingangsspannung von 0 liefert der AD-Wandler eine 0, für 2.55V eine $FF bzw. dezimal 255. Der Wert des Wandlers entspricht also genau dem Spannungswert.

Der AD-Wandler kann über ein Steuerregister namens `ADCTL` an der Adresse $1030 programmiert werden, wobei die Ergebnisse aus den Adressen $1031 bis $1034 gelesen werden können.

Für eine Einzelmessung wird zunächst das `ADCTL` beschrieben. Typischerweise sind die oberen vier Bit gelöscht, die unteren vier geben die Leitung an, deren Wert gemessen werden soll. Nach Abschluß der Messung wird Bit 7 des Steuerregisters gesetzt. Zu diesem Zeitpunkt sind die vier Datenregistern mit vier „Samples", also vier unabhängigen Messungen des Spannungswertes am Eingang beschrieben.

Mit einem gesetzten 5. Bit des Steuerregisters könnte ein „kontinuierlicher" Modus aktiviert werden, in dem die vier Datenwerte kontinuierlich aktualisiert werden, doch wird von dem Betrieb abgeraten, da ansonsten die Messung verfälscht werden könnte.

Das Programm beschreibt zunächst das Steuerregister mit einer 0, da unser Meßobjekt am Eingang 0 angeschlossen ist. Eine Warteschleife liest dann kontinuierlich das Steuerregister und wartet bis das oberste Bit gesetzt ist.

Anstatt nur eines der vier Samples zu betrachten, bildet das Programm dann den Mittelwert aller vier, wobei wieder einmal geschickt der `AB`-`D`-Registerdualismus ausgenutzt wird – zunächst werden die vier Werte im Register `B` addiert und die Überläufe in `A` akkumuliert, dann wird das gesamte Register `D` durch 4 geteilt (zwei Rechtsshifts).

Wie beschrieben entspricht der Meßwert gerade der angelegten Spannung, wir können ihn also direkt mit dem bereits geschriebenen Unterprogramm auf den Siebensegmentanzeigen ausgeben.

Abbildung 9 zeigt das komplette Unterprogramm zur Durchführung einer einzelnen Messung. Abbildung 10 zeigt eine kurze Applikation, die ganz einfach 20 Sekunden lang mehrere Messungen durchführt.

Dabei darf nicht einfach die Messung in einer „engen" Schleife durchgeführt werden, denn vor dem kontinuierlichen Betrieb des AD-Wandlers wurde ja gerade gewarnt. Deshalb wartet das Programm jeweils eine zehntel Sekunde zwischen zwei Messungen.

```
;
; Voltmeter-Messung
;
; Wir machen eine Messung mit dem AD-Wandler an Port 1030. Das
; Ergebnis ist ein Spannungswert von 0.00 bis 2.55 Volt. Zur
; Anzeige brauchen wir die Zahl also nur mittels der Funktion
; zur Dezimalanzeige aufzurufen
;
2801         psha
2802         ldaa #00       ; Initialisierung von ADCTL
2804         staa 1030      ; Messung starten
2807  wait:  ldab 1030      ; Auf Ergebnis warten
280a         andb #80       ; wird im 7. Bit angezeigt
280c         beq  2807
;
; In 1031 ... 1034 stehen vier Ergebnisse. Wir verwenden den
; Mittelwert, indem wir alle Werte in D summieren und durch
; vier teilen.
;
280e         ldaa #00
2810         ldab 1031      ; erstes Ergebnis
2813         addb 1032      ; 8-bit-Addition zweites Ergebnis
2816         adca #00       ; Uebertrag
2818         addb 1033
281b         adca #00
281d         addb 1034
2820         adca #00
2822         lsrd
2823         lsrd
2824         jsr  2500      ; Ausgabe auf Anzeige
2827         pula
2828         rts
```

Abbildung 9: Messung mit dem Voltmeter

```
;
; Voltmeter
;
; Fuehre eine kontinuierliche Messung durch.
;
; Die innere Warteschleife von 7 Taktzyklen wird 7143 mal
; durchlaufen, verzoegert also um eine zehntel Sekunde.
; Die aeussere Schleife zaehlt bis 200, macht insgesamt
; 20 Sekunden.
;
3000           ldx  #c8
3003           ldy  #1be7
3007    loop1: dey              ; Verzoegerungsschleife
3009           bne  3007
300b           jsr  2801        ; Messung durchfuehren
300e           dex
300f           bne  3003        ; und weitermessen
3011           brk
```

Abbildung 10: Kontinuierliche Messung mit dem Voltmeter

4.7 Alles Zusammen

In den vorangegangenen Kapiteln wurden so viele Unterprogramme geschrieben, jetzt ist es auch an der Zeit, daß wir sie zusammenarbeiten lassen. Ganz praktisch ist das z.B. mit der Voltmetermessung und der Ansteuerung des Schrittmotors. Mit den vorhandenen Routinen ist es ein leichtes, den Meßwert des Voltmeters auch mittels des Schrittmotors anzuzeigen, dessen Drehscheibe ja auch Werte von 0 bis 200 anzeigen kann.

Das Ergebnis zeigt Abbildung 11. Alles was wir tun müssen ist, vom gemessenen Wert des AD-Wandlers die derzeitige Motorposition – die wir ja in $1811 abspeichern – zu subtrahieren. Das ergibt die nötige Anzahl von Differenzschritten, mit der wir dann sofort das Unterprogramm zur Motorpositionierung aufrufen können.

Das funktioniert aber nur für Spannungen bis 1.99V. Da der Schrittmotor nie die Position 200 erreicht, es aber laut unserer Logik ständig versucht, fängt er bei höheren Werten an, sich ununterbrochen zu drehen und kommt erst wieder zur Ruhe, wenn der kritische Punkt überschritten wird.

Und das ist noch nicht alles. Wir erinnern uns, daß wir an Port F, an dem ja der Schrittmotor angeschlossen ist, auch den Lautsprecher anschließen können. Stöpseln wir ihn ein, so wird er auch entsprechend den Motorbewegungen angesprochen, denn das Bit der Motorphase trifft ja auch die übrigen Bits des Ports. Bewegt sich der Motor, ist ein charakteristisches Knattern zu hören – mit der zugehörigen stereovisuellen Darstellung auf Siebensegmentanzeige und Schrittmotor sowie dem entsprechenden Blinken der Leuchtdioden entsteht ein echtes multimediales Ereignis, an dem sich jeder erfreuen kann – zumindest die Urheber dieses Unsinns.

```
;
; Voltmeter + Schrittmotor
;
; Wir messen die Spannung mit unserem Voltmeter und positionieren
; dann den Schrittmotor auf diesen Wert.
;
3100          jsr  2801      ; Voltmeter-Messung
3103          ldaa 1811      ; Motorstellung
3106          staa 1820      ; als 16-bit-Wert
3109          ldaa #00       ; (mit high byte gleich 0)
310b          staa 181f      ; nach 181f
310e          subd 181f      ; Differenz berechnen
3111          jmp  25fd      ; und Motor drehen

;
; Voltmeter + Schrittmotor, kontinuierliche Messung
;
3200          ldx  #c8
3203   oloop: ldy  #1be7
3207   loop1: dey            ; Verzoegerungsschleife
3209          bne  3207
320b          jsr  3100      ; Messung durchfuehren
320e          dex
320f          bne  3203      ; und weitermessen
3211          brk
```

Abbildung 11: Voltmeteranzeige mittels Schrittmotor