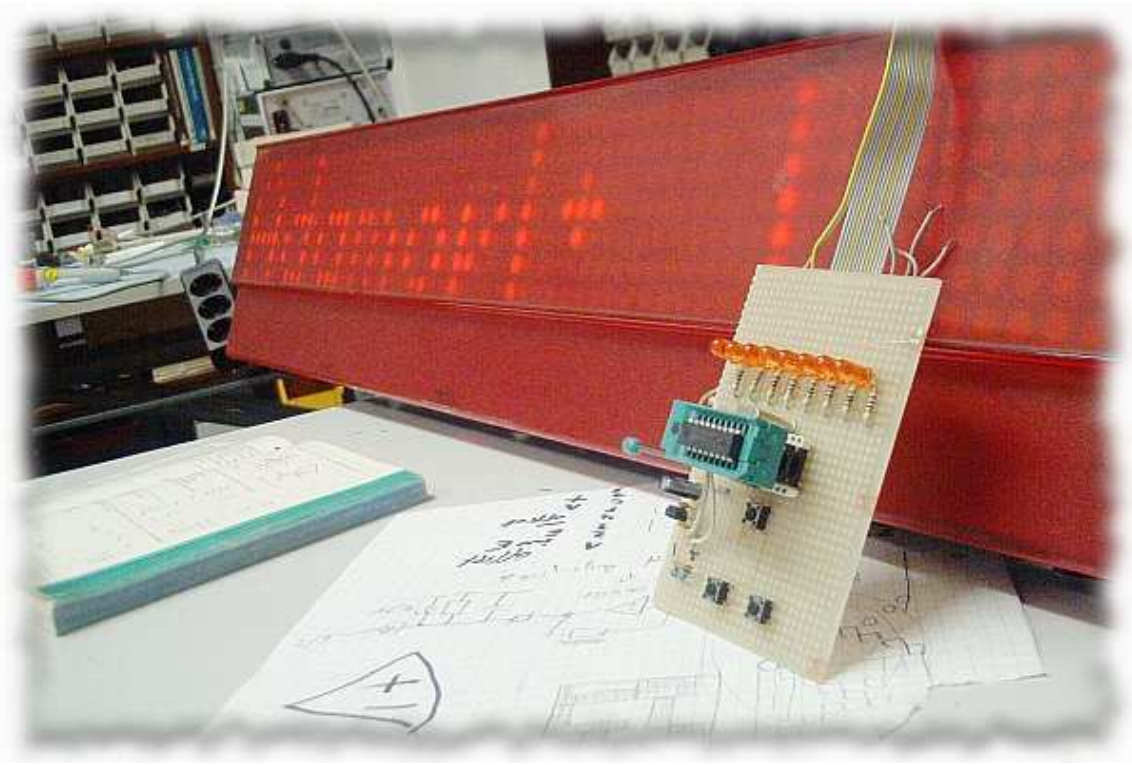
Ansteuerung einer LED-Anzeigetafel

Mario Krell Berit Grußien Volker Grabsch

5. Juli 2006

http://www.profv.de/uni/



Inhaltsverzeichnis

1 Problemstellung

1.1 Analyse

Es soll eine große Matrix von Leuchtdioden angesteuert werden. Dort werden wöchentlich wiederkehrende Ereignisse zur entsprechenden Zeit angezeigt. Hierzu verfügt die Steuerung über eine interne Uhr. Das entstandene Gerät soll einmalig mit Ereignis-Daten bestückt werden, und mehrere Monate lang autonom arbeiten können.
Zum Beispiel könnten die aktuell laufenden Vorlesungen (inkl. Raumnummern) angezeigt werden oder der Stundenplan in einer Schule.

1.2 Einschränkungen

Die LED-Anzeigetafel wird zeilenweise im Multiplex-Betrieb angesteuert, da dies gängiger Standard bei den Anzeigetafeln ist.
Dabei werden die Zeilen seriell mit Daten bestückt, um die Schaltung so flexibel wie möglich zu halten. Sollte die Anzeigetafel einen parallelen Eingang haben, so kann man immer noch ein Schieberegister dazwischen schalten. Würde unsere Schaltung hingegen parallel arbeiten und die Anzeigetafel seriell, wäre der Aufwand einer Übersetzungs-Schaltung sehr viel höher.
Auch das Umschalten der Zeilen geschieht seriell, aus dem selben Grund.
Die Tafel kann nur die Zeichen aus einem bestimmten im ROM definierten Zeichsatz anzeigen.
Die interne Uhr kann nicht nachträglich gestellt werden. Deshalb muss die Schaltung zu einem im ROM fest eingestellten Zeitpunkt gestartet werden (z.B. Montag, 8:00 Uhr).

1.3 Basisfunktionalität

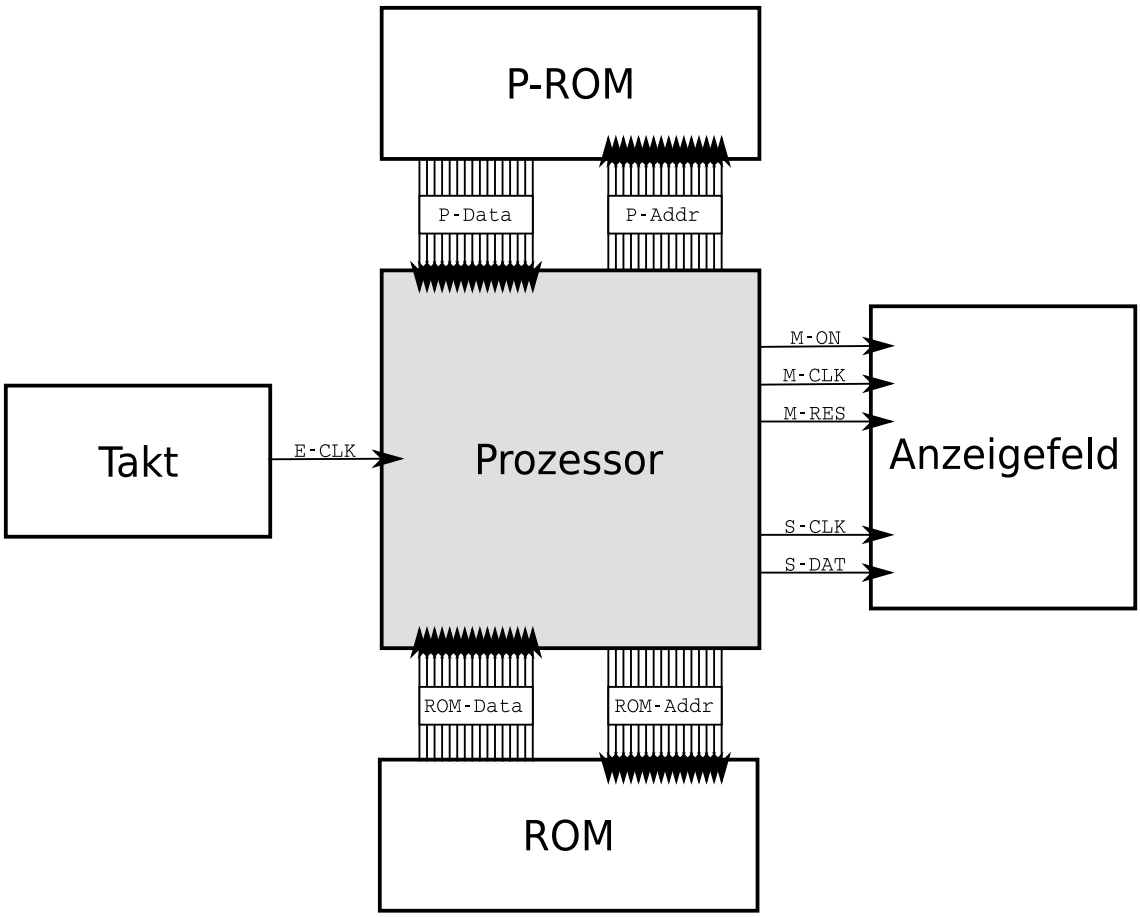
Der Prozessor erhält nur eine einzige Eingabe, nämlich einen externen, sehr zuverlässigen Takt. (z.B. Quarz-Schwingkreis)
Ausgehend von diesem Takt berechnet (zählt) er die aktuelle Uhrzeit und den Wochentag.
Anhand der Uhrzeit und den in einem ROM gespeicherten Ereignis-Daten steuert er eine LED-Anzeigetafel an.

1.4 System-Anforderungen

- Das System soll einmalig gestartet werden und danach autonom laufen können.
- Die Ereignis-Daten sollen pro Tag mindestens 128 Ereignisse enthalten können. (z.B. je 32 Veranstaltungen in 4 Zeitabschnitten)
- Jedes Ereignis soll in einer Text-Zeile bestehend aus maximal 32 Zeichen beschrieben werden.
- 6 Ereignisse (Textzeilen) werden auf dem Anzeigefeld auf einmal angezeigt. Alle 3 Sekunden werden sie um eine Textzeile nach oben gescrollt.
- Jedes Zeichen wird in einem Wort kodiert.
- Es wird ein 8-Bit-Zeichensatz verwendet, z.B. ISO-8859-1.
- Jedes Zeichen ist 6 Pixel breit und 8 Pixel hoch.
- Jede Pixel-Zeile eines Zeichens wird in einem Wort im Zeichensatz kodiert.
- Das Anzeigefeld soll sich mit einer augenfreundlichen Frequenz von 100Hz erneuern.

2 Architektur

2.1 Komponenten und Schnittstellen



Eingänge

`E-CLK`	1 Bit	Externer Takt
`P-Data`	16 Bit	Daten aus dem P-ROM
`ROM-Data`	16 Bit	Daten aus dem ROM

Ausgänge

`P-Addr`	16 Bit	Adress-Wahl an den P-ROM
`ROM-Addr`	16 Bit	Adress-Wahl an den ROM
`M-ON`	1 Bit	Ein-/Ausschalten des Anzeigefelds
`M-CLK`	1 Bit	Weiterschalten auf nächste Zeile des Anzeigefelds (reagiert auf 0-1-Flanke)
`M-RES`	1 Bit	Zurücksetzen auf erste Zeile des Anzeigefelds
`S-DAT`	1 Bit	Eingangsdaten für das Schieberegister des Anzeigefelds
`S-CLK`	1 Bit	Laden von S-DAT in das Schieberegister des Anzeigefelds (reagiert auf 0-1-Flanke)

2.2 ROM

Der ROM enthält alle für die Anwendung notwendigen Daten.
Er wird über `ROM-Addr` wortweise adressiert, und liefert genau ein Wort über `ROM-Data` zurück.
Die Daten sind im ROM wie folgt organisiert:

Zeichensatz	Zeichen 0	Pixel-Zeile 0
		...
		Pixel-Zeile 7
	Zeichen 1	
	...	
	Zeichen 65 (A)	
	...	
	Zeichen 255	

Initialisierungs-Werte	Start-Zeit	Tag
		Stunde
		Minute
	Start-Zeitabschnitt	Adresse

Ereignis-Daten	Zeitabschnitt 1	Start-Zeit	Tag
			Stunde
			Minute
		Ereignis 1	Zeichen 1
			...
			Zeichen 32
		Ereignis 2	
		...	
		Ereignis n	
		Markierung	Zeichen 0
	Zeitabschnitt 2		
	...		
	Zeitabschnitt m		
	Endmarkierung		Zeichen 0
			Zeichen 0

Anmerkungen:

- Im Zeichensatz wird jede Pixel-Zeile (6 Bit) eines Zeichen in einem Wort (16 Bit) kodiert. Dies geschieht in den höchstwertigen 6 Bit des Wortes:

  `xxxx xx00 0000 0000`

- Jedes Zeichen (8 Bit) eines Ereignisses wird in einem Wort (16 Bit) in die niederwertige Hälfte gesetzt:

  `0000 0000 xxxx xxxx`

- Die horizontalen und vertikalen Zwischenräume zwischen den Zeichen (ca. 1 Pixel Abstand) werden mit im Zeichensatz kodiert, wie es auch bei Grafikkarten im Textmodus üblich ist.

  Dies ermöglicht auch Sonderzeichen, mit denen man größere zusammenhängende Strukturen darstellen kann (z.B. Umrandungen, Logos, Pfeile, ...)

- Die Ereignisse dürfen nicht mit einem „Zeichen 0“ beginnen. Dieses dient zur Markierung der Start-Zeit des nächsten Zeitabschnittes.

  Das ist keine Einschränkung, da das „Zeichen 0“ in fast jedem Zeichensatz genauso wie ein Leerzeichen (das „Zeichen 32“) aussieht.

- Um das Ende der Ereignisdaten zu markieren, werden zwei aufeinanderfolgende „Zeichen 0“ verwendet.

2.3 Takt

Neben dem internen Takt benötigt der Prozessor einen externen Takt (`E-CLK`). Dieser ist sehr viel langsamer und zuverlässiger (z.B. ein Quarz-Schwingkreis).
Er wird für die Steuerung des Multiplex-Betriebs genutzt, d.h. seine Periodenlänge gibt genau die Umschaltzeit von einer LED-Zeile zur nächsten im Anzeigefeld an.
Außerdem wird er zur Feststellung der aktuellen Uhrzeit und zum Scrollen benutzt.

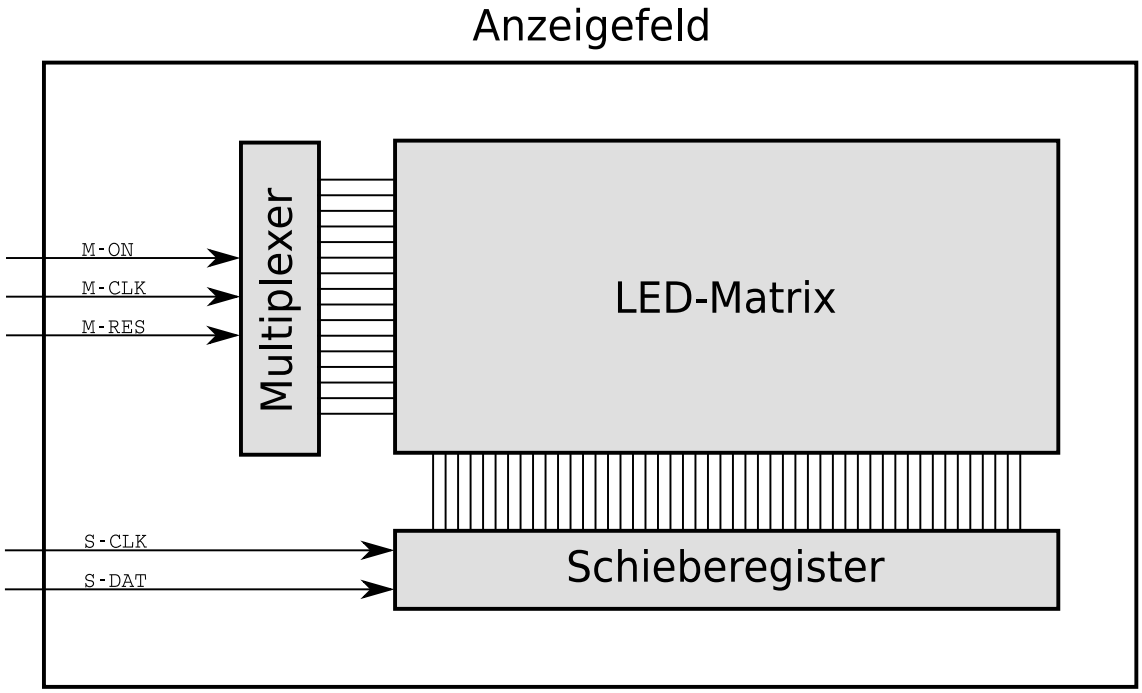
2.4 P-ROM

Dieser ROM enthält den Programm-Code.
Er wird über `P-Addr` wortweise adressiert, und liefert genau ein Wort über `P-Data` zurück.

2.5 Anzeigefeld

Wir gehen davon aus, dass das Anzeigefeld für einen zeilenweisen Multiplex-Betrieb ausgelegt ist.
Das bedeutet, es wird zu einem Zeitpunkt nur eine einzige Zeile angezeigt. Welche dies ist, wird durch einen vorgeschalteten Multiplexer bestimmt, der nach und nach jeweils genau eine Zeile aktiviert. Das Weiterschalten zur nächsten Zeile erfolgt über eine 0-1-Flanke an `M-CLK`.
Für den Fall, dass es zwischendurch zu Störungen in den Leitungen zwischen Prozessor und Anzeigefeld kommt, soll der Multiplexer regelmäßig auf die erste Zeile zurückgesetzt werden. Diese Synchronisierung geschieht, indem `M-RES` auf 1 gesetzt wird.
Der Inhalt der aktuell anzuzeigenden Zeile wird in einem Schieberegister gespeichert. Dieses übernimmt bei jeder 0-1-Flanke von `S-CLK` das zu dem Zeitpunkt in `S-DAT` gesetzte Bit.
Wir gehen davon aus, dass das Schieberegister die einzelnen Pixel von rechts nach links hinein schiebt, sodass man die Zeichen in der intuitiven Reihenfolge nacheinander laden kann.

Sollte das Schieberegister andersherum funktionieren, so braucht man das Programm nicht zu ändern. Stattdessen einfach die Zeichen spiegelverkehrt im Zeichensatz notieren, und die Zeichen jedes Ereignisses in umgekehrter Reichenfolge speichern.
Da die gesamte Ansteuerung sehr zeitkritisch verläuft, müssen die LEDs nachleuchten, auch wenn ihre jeweilige Zeile ausgeschaltet ist. Dies kann über entsprechende Kondensatoren in der LED-Matrix realisiert werden.
Wenn während des Befüllens des Schieberegisters das Anzeigefeld nicht deaktiviert wird, kommt es zu unerwünschten Nachleucht- und Flacker-Effekten. Um diese zu vermeiden, sollte das Anzeigefeld zwischendurch ausgeschaltet werden können. Dies geschieht über `M-ON`. Wird `M-ON` auf 0 gesetzt, werden alle Zeilen des Anzeigefelds ausgeschaltet. `M-ON` ist unabhängig von der im Multiplexer gespeicherten Zeile.
`M-ON` kann einfach durch dem Multiplexer nachgeschaltete Und-Gatter realisiert werden. Unser Programm funktioniert jedoch auch, wenn das Anzeigefeld kein `M-ON` unterstützt.



2.6 Anforderungs-Analyse

- Das System soll einmalig gestartet werden und danach autonom laufen können. Da es immer eine aktuelle Uhrzeit braucht, sollten der externe Takt und die Zählung der Uhrzeit auch bei Stromausfall laufen.

  Das heißt, der Prozessor und der Takt benötigen eine Batterie oder einen Akku, während die LED-Anzeigetafel den Strom direkt durch ein Netzteil bezieht.

- Das Anzeigefeld stellt 32×6 Zeichen zu je 6×8 Pixeln dar.

  $\Rightarrow$ Es zeigt eine Datenmenge von $32 \cdot 6 \cdot 6 \cdot 8$ Bit = 9kBit.

- Das Anzeigefeld muss sich mit 100Hz erneuern und besteht aus 6 Textzeilen zu je 8 Pixel-Zeilen.

  $\Rightarrow$ Jede Zeile muss mit einer Frequenz von mindestens $100\text{Hz} \cdot 6 \cdot 8 = 4{,}8\text{kHz}$ angesteuert werden.

  $\Rightarrow$ Der externe Takt (`E-CLK`) muss auf 4,8kHz eingestellt werden.

- Um eine LED-Zeile des Anzeigefeldes zu füllen, müssen 32 Zeichen dekodiert und $32 \cdot 6$ Pixel in das Schieberegister geladen werden. Dazu sind ca. 1000 Befehle notwendig, die jeweils ca. 6 Takte benötigen.

  $\Rightarrow$ Der interne Prozessor-Takt muss auf mindestens $4{,}8\text{kHz} \cdot 1000 \cdot 6 = 28{,}8\text{MHz}$ eingestellt werden.

  $\Rightarrow$ ROM und P-ROM müssen innerhalb von mindestens $\frac{1}{28{,}8MHz} \leq 35\text{ns}$ die Daten bereitstellen.

- Das Programm wird aus höchstens 1024 LOC bestehen, wobei jeder Befehl in ca. 1–2 Worten kodiert ist.

  $\Rightarrow$ Der P-ROM braucht nur

  $$1024 \cdot 2 = 2048 \text{ Worte}$$

  speichern.

- Der ROM beinhaltet den Zeichensatz, die Initialisierungs-Werte und die Ereignis-Daten.

  Der Zeichensatz kodiert 256 Zeichen, die aus jeweils 8 Pixel-Zeilen bestehen. Jede Pixel-Zeile wird in einem Wort kodiert.

  Die Initialisierungs-Werte bestehen aus 4 Worten.

  Die Ereignisdaten müssen 7 Tage zu jeweils 128 Ereignissen erfassen können. Jedes Ereignis besteht aus 32 Zeichen zu je einem Wort.

  Zusätzlich gibt in jedem der 7 Tage jeweils ca. 8 Zeitabschnitte, in die zusätzlich noch eine Start-Zeit (4 Worte) kodiert wird.

  $\Rightarrow$ Der ROM muss etwa

  $$256 \cdot 8 + 4 + 7 \cdot 128 \cdot 32 + 7 \cdot 8 \cdot 4 = 30948 \text{ Worte}$$

  speichern.

  $\Rightarrow$ Es werden mindestens 15 Bit für `ROM-Addr` benötigt. Der Einfachheit halber gehen wir von 16 Bit aus.

- Da im Prozessor alles über einen Bus laufen soll, auch die ROM-Adressen, benötigen wir mindestens eine 16-Bit-Architektur.

  Auch die `P-Addr`, Zähler, Zeichen (8 Bit) und die Pixel-Zeilen eines Zeichens (6 Bit) passen locker in 16 Bit.

  $\Rightarrow$ 1 Wort := 16 Bit

- Der P-ROM muss 2048 Worte speichern. Der ROM muss mindestens $2^{15} = 32k$ Worte groß zu sein. Das ist aber ziemlich knapp bemessen. Daher gehen wir von dem maximal adressierbaren ROM zu $2^{16} = 64k$ Worten aus. Ein Wort sind 16 Bit.

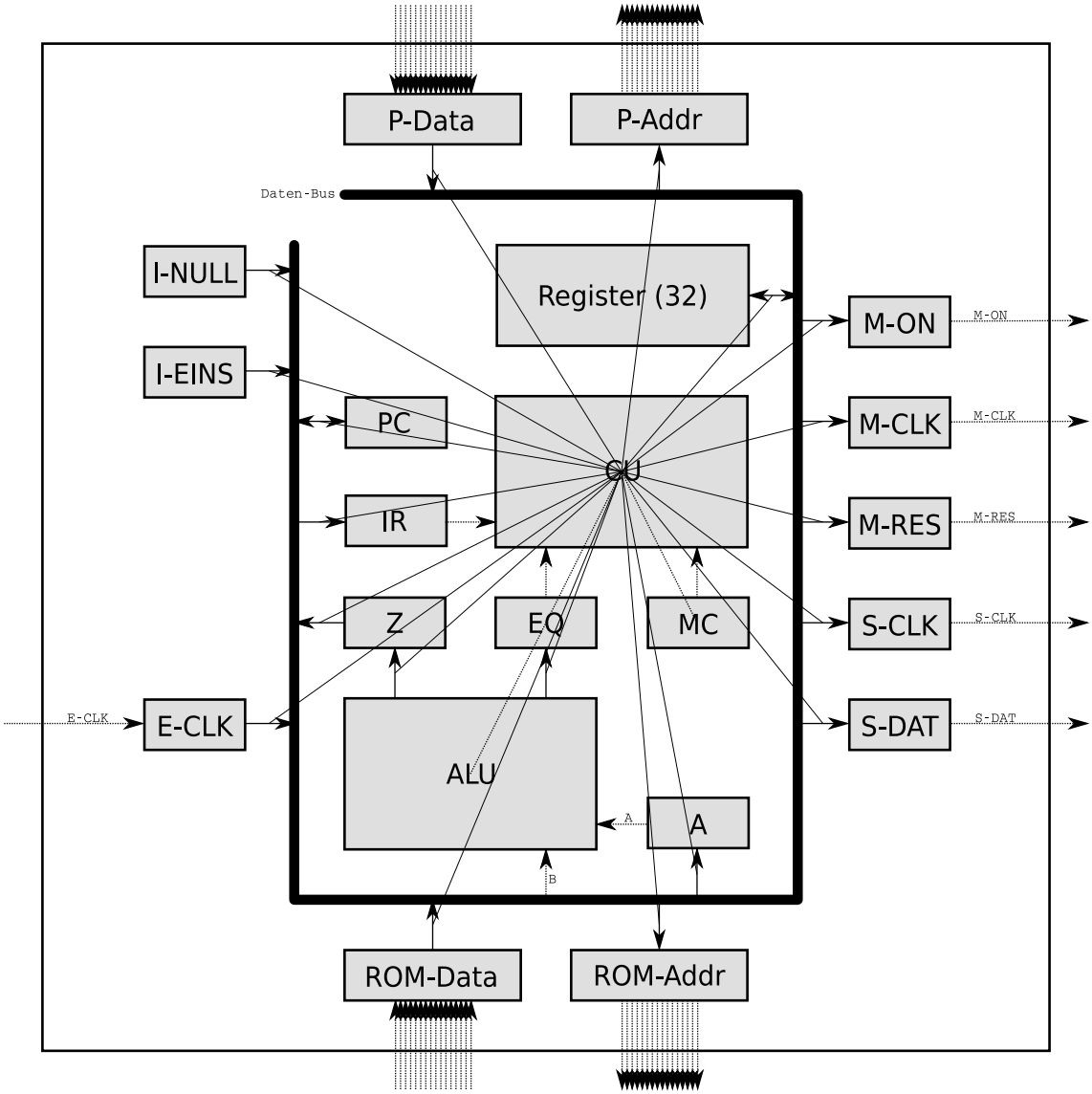
  $\Rightarrow$ Empfohlene Größe für den P-ROM:

  $$2048 \cdot 16\text{Bit} = 32\text{kBit} = 4\text{kB}$$

  $\Rightarrow$ Empfohlene Größe für den ROM:

  $$64k \cdot 16\text{Bit} = 1\text{MBit} = 128\text{kB}$$

3 Mikroprozessor



3.1 Register

Es gibt ...

7	interne Register:	`PC`, `IR`, `Z`, `EQ`, `A`, `P-Addr`, `P-Data`
32	Allzweck-Register:	`R-0`, ..., `R-31`
4	Eingabe-Register:	`I-NULL`, `I-EINS`, `E-CLK`, `ROM-Data`
6	Ausgabe-Register:	`ROM-Addr`, `M-ON`, `M-CLK`, `M-RES`, `S-CLK`, `S-DAT`

Dem Assembler-Programm stehen 42 Register zur Verfügung, nämlich alle außer den internen. Unserer Prozessor ist somit Adams-kompatibel.
Im Folgenden verwenden wir die Bezeichnungen

Reg – ein Allzweck-Register

RegI – ein Allzweck- oder Eingabe-Register

RegO – ein Allzweck- oder Ausgabe-Register

Alle RegI besitzen eine Steuerleitung zum Schreiben auf den Daten-Bus. (`R-5`$_{out}$, `E-CLK`$_{out}$, ...)
Alle RegO besitzen eine Steuerleitung zum Lesen vom Daten-Bus. (`R-7`$_{in}$, `S-DAT`$_{in}$, ...)
Auch die internen Register besitzen natürlich entsprechende Steuerleitungen.
Der Zugriff auf den ROM geschieht somit durch direkten Zugriff des Assembler-Programms auf `ROM-Addr` und `ROM-Data`.
Alle Register sind 16-Bit groß, mit Ausnahme des internen 1-Bit-Registers `EQ`.
Die Register `I-NULL` und `I-EINS` sind jeweils fest verdrahtet mit den Zahlen:

```
I-NULL = 0000 0000 0000 0000
I-EINS = 0000 0000 0000 0001
```

Von einigen 16-Bit-Registern wird jeweils nur 1 Bit mit der Außenwelt verbunden: `E-CLK`, `M-ON`, `M-CLK`, `M-RES`, `S-CLK`, `S-DAT`
Die Signal-Leitung `S-DAT` ist mit dem höchstwertigen Bit des Registers `S-DAT` verbunden:

```
x000 0000 0000 0000   (nur S-DAT)
```

Alle anderen Signal-Leitungen sind mit dem niederwertigsten Bit der entsprechenden Register verbunden.
Die jeweils übrigen 15 Bits der Register werden beim Schreiben ignoriert. Im Eingabe-Register `E-CLK` werden die übrigen 15 Bits fest auf 0 verdrahtet:

```
0000 0000 0000 000x   (Rest)
```

3.2 MC – Mikroprogramm-Zähler

Der Mikroprogramm-Zähler bestimmt die aktuelle Position im Mikroprogramm.
Er kann vom Mikrocode zurückgesetzt werden, sobald ein Befehl abgearbeitet ist, um den Befehlshole-Ausführungs-Vorgang im nächsten Takt zu starten.
Er ist ein 3-Bit-Zählwerk mit

- Reset-Leitung MC_{res}
- Takteingang CLK
- 3-Bit-Ausgang

3.3 ALU – Arithmetik-Logik-Einheit

Die Arithmetik-Logik-Einheit hat

- ein Eingangs-Register A (1 Wort)
- ein direkte Verbindung B (1 Wort) zum Daten-Bus
- ein Ausgabe-Register Z (1 Wort)
- ein Ausgabe-Register EQ (1 Bit)
- drei Eingangs-Steuerleitungen, die die Operation festlegen:

	Operation	**Z**	**EQ**
ALU_{B+1}	inkrementieren	B+1	undef.
ALU_{A+B}	addieren	A+B	undef.
$\mathrm{ALU}_{A=B}$	vergleichen	undef.	$\left\{\begin{array}{l}0, \text{ falls } A \neq B\\ 1, \text{ falls } A = B\end{array}\right\}$

3.4 Befehlssatz und -kodierung

OP-Code	**Befehl**	**Bedeutung**
00	mov RegO, RegI	RegO := RegI
01	add Reg, RegI	Reg := Reg + RegI
10	mov RegO, Num	RegO := Num
11	jne RegI1, RegI2, Addr	Springe zu Addr, falls RegI1 $\neq$ RegI2

Da die Register `ROM-Addr` und `ROM-Data` zur Verfügung stehen, sind keine Befehle zur indirekten Adressierung nötig.
Die ersten beiden Befehle werden in einem Wort kodiert:

OP-Code	Register 1	Register 2
2	7	7

Die letzten beiden Befehle werden in zwei Worten kodiert:

OP-Code	Register 1	leer	Num
2	7	7	16

OP-Code	Register 1	Register 2	Addr
2	7	7	16

Da es nur 42 Register gibt, können sie problemlos in den 7 Bits kodiert werden. Wird dort ein nicht-existentes oder ungültiges Register eingetragen (z.B. ein RegO, wo ein RegI verlangt wird), landet der Befehl entsprechend im Nirvana, d.h. es wird keine Steuerleitung eines Registers angesprochen.

3.5 Erweiterter Befehlssatz

add Reg, Num	Reg := Reg + Num
jne RegI, Num, Addr	Springe zu Addr, falls RegI $\neq$ Num
jmp Addr	Springe zu Addr

Wann immer diese Befehle im Assembler-Programm auftauchen, sind damit die folgende Befehlssequenzen gemeint. Es wird das Register `R-Num` für interne Zwecke reserviert.

- add Reg, Num
  - Falls Num = 0

    ```
    add  Reg, I-NULL
    ```

  - Falls Num = 1

    ```
    add  Reg, I-EINS
    ```

  - Sonst

    ```
    mov  R-Num, Num
    add  Reg, R-Num
    ```

- jne RegI, Num, Addr
  - Falls Num = 0

    ```
    jne  RegI, I-NULL, Addr
    ```

  - Falls Num = 1

    ```
    jne  RegI, I-EINS, Addr
    ```

  - Sonst

    ```
    mov  R-Num, Num
    jne  RegI, R-Num, Addr
    ```

- jmp Addr

  ```
  jne  I-NULL, I-EINS, Addr
  ```

3.6 Steuer-Einheit

Die Steuer-Einheit kontrolliert sämtliche Steuerleitungen:

- RegI_{out}
- RegO_{in}
- ALU_{B+1}, ALU_{A+B}, $\text{ALU}_{A=B}$
- MC_{res}

Ihr Verhalten hängt ab von:

- MC (aktuelle Position im Mikroprogramm)
- IR (aktueller Befehl)
- EQ (für Sprünge)

Die Steuereinheit ist somit einfach ein Dekodierer für den Mikrocode.
Ihre Verbindung mit dem EQ-Register ermöglicht, dass in Abhängigkeit des Ergebnisses eines Vergleichs ($\text{ALU}_{A=B}$) verschiedene Sequenzen von Mikrocode ausgeführt werden.

3.7 Mikrocode für Steuer-Einheit

IR	**MC**	**EQ**	**Steuer-Leitungen**			
*	0	*	PC_{out}	ALU_{B+1}	Z_{in}	P-Addr_{in}
	1	*	Z_{out}		PC_{in}	
	2	*	P-Data_{out}		IR_{in}	
mov RegO, Num	3	*	PC_{out}	ALU_{B+1}	Z_{in}	P-Addr_{in}
	4	*	Z_{out}		PC_{in}	
	5	*	P-Data_{out}		RegO_{in}	MC_{res}
mov RegO, RegI	3	*	RegI_{out}		RegO_{in}	MC_{res}
add Reg, RegI	3	*	RegI_{out}		A_{in}	
	4	*	Reg_{out}	ALU_{A+B}	Z_{in}	
	5	*	Z_{out}		Reg_{in}	MC_{res}
jne RegI1, RegI2, Addr	3	*	RegI1_{out}		A_{in}	
	4	*	RegI2_{out}	$\text{ALU}_{A=B}$	EQ_{in}	
	5	*	PC_{out}	ALU_{B+1}	Z_{in}	P-Addr_{in}
	6	1	Z_{out}		PC_{in}	MC_{res}
	6	0	P-Data_{out}		PC_{in}	MC_{res}

Anmerkungen:

- Ein * in der Tabelle bedeutet, dass der entsprechende Eingangs-Wert egal ist.
- Die ersten 3 Zeilen stellen die Befehlshole-Phase dar.
- Es gibt keine WFMFC-Leitung, da wir keinen RAM ansteuern und davon ausgehen, dass P-ROM und ROM schnell genug reagieren.

4 Assembler-Programm

Das Programm besteht im Wesentlichen aus verschalteten Zählern. Daher hat es einen großen Bedarf an Registern. Die Register werden der Übersichtlichkeit halber nicht mit Zahlen wie `R-0` bezeichnet, sondern erhalten stattdessen inhaltlich sinnvolle Namen wie `R-Minute`.
Das Programm ist modular aufgebaut. Steht im Quellcode eine Zeile wie:

```
[Warten auf Takt]
```

dann soll dort der Code aus dem entsprechenden Kapitel an dieser Stelle hineinkopiert werden.
Da es sich um eine zeitkritische Anwendung handelt, geht grundsätzlich Geschwindigkeit vor Code-Größe. Zum Beispiel wurden Schleifen von konstanter Länge durch Code-Wiederholung statt typischer Schleifen-Konstrukte realisiert.
Zur Übersicht noch ein paar allgemeine Informationen zum Programm:

Statistik

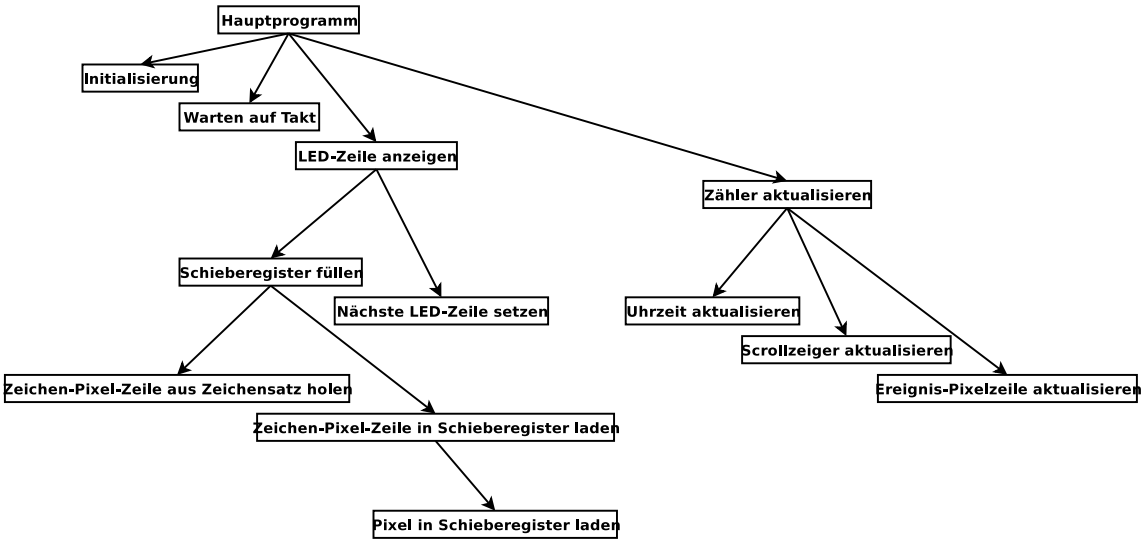
90	Code-Zeilen (ohne Wiederholungen)
109	Worte (ohne Wiederholungen)
1184	Worte insgesammt

Konstanten

`AddrZeichensatz`	Start-Adresse des „Zeichensatzes“ im ROM
`AddrInit`	Start-Adresse der „Initialisierungs-Werte“ im ROM

Register

`R-Num`	internes Register für die erweiterten Befehle
`R-Temp`	temporäres Register für verschiedene Zwischenwerte
`R-Tag`	
`R-Stunde`	
`R-Minute`	
`R-ZehntelSek`	
`R-Anzeigezeit`	die Zeit bis zum Weiterscrollen (Überlauf alle 3 Sekunden)
`R-Taktzähler`	zählt den externen Takt (Überlauf alle 0,1 Sekunden)
`R-Zeitabschnitt`	aktueller Zeitabschnittanfang, zeigt auf erstes Ereignis
`R-Ereignis`	die Start-Adresse des aktuellen Ereignisses
`R-Ereigniszähler`	zählt Ereignisse die auf der Tafel angezeigt werden
`R-EreignisPixelzeile`	Pixelzeilennr. der Ereigniszeile (0 – 7)
`R-Zeichen`	Adresse des aktuellen Zeichens
`R-ZeichenPixelzeile`	aktuelle Pixelzeile des Zeichens in R-Zeichen
`R-LEDZeile`	Nummer der LEDZeile, die gerade angezeigt wird. (0 – 47)



4.1 [Hauptprogramm]

```
    [Initialisierung]

Hauptschleife:
    [Warten auf Takt]
    [LED-Zeile anzeigen]
    [Zähler aktualisieren]
    jmp  Hauptschleife
```

4.2 [Initialisierung]

Initialisierung der Register aus dem Init-Block im ROM.

```
    mov  ROM-Addr, AddrInit+0
    mov  R-Tag, ROM-Data
    mov  ROM-Addr, AddrInit+1
    mov  R-Stunde, ROM-Data
    mov  ROM-Addr, AddrInit+2
    mov  R-Minute, ROM-Data
    mov  ROM-Addr, AddrInit+3
    mov  R-Zeitabschnitt, ROM-Data
    mov  R-ZehntelSek, 0
    mov  R-Taktzähler, 0
    mov  R-LEDZeile, 0
    mov  R-Anzeigezeit, 0
    mov  R-EreignisPixelzeile, 0
    mov  R-Ereignis, R-Zeitabschnitt
    mov  R-Ereigniszähler, 0
```

4.3 [Warten auf Takt]

Warte auf 0-1-Flanke des externen Taktes.

```
Warte0:
    jne  E-CLK, 0, Warte0
Warte1:
    jne  E-CLK, 1, Warte1
```

4.4 [LED-Zeile anzeigen]

Schaltet die aktuelle LED-Zeile aus, lädt die nächste LED-Zeile und zeigt sie an.

```
    mov  M-ON, 0
    mov  R-Zeichen, R-Ereignis
    [Schieberegister füllen (1)]
    [Schieberegister füllen (2)]
    [...]
    [Schieberegister füllen (32)]
    [Nächste LED-Zeile setzen]
    mov  M-ON, 1
```

4.5 [Schieberegister füllen]

Füllt Schieberegister mit der Pixelzeile des aktuellen Zeichens.

```
    [Zeichen-Pixel-Zeile aus Zeichensatz holen]
    [Zeichen-Pixel-Zeile in Schieberegister laden]
    add  R-Zeichen, 1
```

4.6 [Zeichen-Pixel-Zeile aus Zeichensatz holen]

Lädt die Zeichen-Pixel-Zeile des aktuellen Zeichens aus dem Zeichensatz.

R-ZeichenPixelzeile := ROM[AddrZeichensatz
+ 8 * ROM[R-Zeichen]
+ R-EreignisPixelzeile]

```
    mov  ROM-Addr,  R-Zeichen
    mov  R-Temp, ROM-Data
    add  R-Temp, R-Temp
    add  R-Temp, R-Temp
    add  R-Temp, R-Temp
    add  R-Temp, AddrZeichensatz
    add  R-Temp, R-EreignisPixelzeile
    mov  ROM-Addr, R-Temp
    mov  R-ZeichenPixelzeile, ROM-Data
```

4.7 [Zeichen-Pixel-Zeile in Schieberegister laden]

Lädt die Zeichen-Pixel-Zeile in das Schieberegister.
Der Inhalt von R-ZeichenPixelzeile wird dabei zerstört.

```
    [Pixel in Schieberegister laden (1)]
    [...]
    [Pixel in Schieberegister laden (6)]
```

4.8 [Pixel in Schieberegister laden]

Lädt ein Pixel der Zeichen-Pixel-Zeile in das Schieberegister. Schiebt die Bits von R-ZeichenPixelzeile um 1 nach links.

```
    mov  S-CLK, 0
    mov  S-DAT, R-ZeichenPixelzeile
    mov  S-CLK, 1
    add  R-ZeichenPixelzeile, R-ZeichenPixelzeile
```

4.9 [Nächste LED-Zeile setzen]

```
    add  R-LEDZeile, 1
    jne  R-LEDZeile, 48, nächste LEDZeile

    mov  R-LEDZeile 0
    mov  M-RES 1
    mov  M-RES 0
    jmp  fertig1

nächste LEDZeile:
    mov  M-CLK, 0
    mov  M-CLK, 1

fertig1:
```

4.10 [Zähler aktualisieren]

```
    [Uhrzeit aktualisieren]
    [Scrollzeiger aktualisieren]
    [Ereignis-Pixelzeile aktualisieren]
```

4.11 [Uhrzeit aktualisieren]

```
    add  R-Taktzähler, 1
    jne  R-Taktzähler, 480, fertig2

    # Überlauf R-Taktzähler
    mov  R-Taktzähler, 0
    add  R-ZehntelSek, 1
    add  R-Anzeigezeit, 1
    jne  R-ZehntelSek, 600, fertig2

    # Überlauf R-ZehntelSek
    mov  R-ZehntelSek, 0
    add  R-Minute, 1
    jne  R-Minute, 60, fertig2

    # Überlauf R-Minute
    mov  R-Minute, 0
    add  R-Stunde, 1
    jne  R-Stunde, 24, fertig2

    # Überlauf R-Stunde
    mov  R-Stunde 0
    add  R-Tag  1
    jne  R-Tag, 8, fertig2

    # Überlauf R-Tag
    mov  R-Tag 0

fertig2:
```

4.12 [Scrollzeiger aktualisieren]

Der Scrollzeiger zeigt auf das erste Ereignis, das auf der Tafel stehen soll. Wenn R-Anzeigezeit den Wert 30 (3 Sekunden) erreicht hat, wird der Scrollzeiger gewechselt. Falls er am Ende eines Zeitabschnitts angelangt ist, erfolgt ein Zeitabgleich mit dem nächsten Zeitabschnitt und gegebenenfalls folgt eine Aktualisierung.

```
    jne  R-Anzeigezeit, 30, fertig3

    # scrollen, da Überlauf R-Anzeigezeit
    mov  R-Anzeigezeit, 0
    add  R-Scrollzeiger, 32
    mov  ROM-Addr, R-Scrollzeiger
    jne  ROM-Data, 0, fertig3

    # Überlauf R-Scrollzeiger
    mov  R-Temp, R-Scrollzeiger
    mov  R-Scrollzeiger, R-Zeitabschnitt

    # nächster Zeitabschnitt schon erreicht?
    add  R-Temp, 1
    mov  ROM-Addr, R-Temp
    jne  ROM-Data, R-Tag, fertig3

    add  R-Temp, 1
    mov  ROM-Addr, R-Temp
    jne  ROM-Data, R-Stunde, fertig3

    add  R-Temp, 1
    mov  ROM-Addr, R-Temp
    jne  ROM-Data, R-Minute, fertig3

    # zum nächsten Zeitabschnitt wechseln
    add  R-Temp, 1
    mov  R-Zeitabschnitt, R-Temp
    mov  R-Scrollzeiger, R-Zeitabschnitt

fertig3:
```

4.13 [Ereignis-Pixelzeile aktualisieren]

Der Ereigniszeiger wechselt der Reihe nach durch die Pixelzeilen der 6 Ereignisse beginnend mit dem Ereignis, welches durch den Scrollzeiger gekennzeichnet ist. Falls das letzte Ereignis erreicht wurde, wird danach wieder das erste Ereignis des Zeitabschnitts angezeigt.

```
    add  R-EreignisPixelzeile, 1
    jne  R-EreignisPixelzeile, 8, fertig4

    # nächstes Ereignis
    mov  R-EreignisPixelzeile, 0
    add  R-Ereigniszähler, 1
    add  R-Ereignis, 32
    jne  R-Ereigniszähler, 6, Ende Zeitabschnitt prüfen

    # Überlauf R-Ereigniszähler
    mov  R-Ereigniszähler, 0
    mov  R-Ereignis, R-Scrollzeiger
    jmp  fertig4

    # kein Überlauf R-Ereigniszähler
Ende Zeitabschnitt prüfen:
    mov  ROM-Addr, R-Ereignis
    jne  Rom-Data, 0, fertig4
    mov  R-Ereignis, R-Zeitabschnitt

fertig4:
```

5 Schlussbemerkungen

5.1 Einschränkungen

- Da wir das Anzeigefeld sehr groß gewählt haben (9216 Pixel), brauchen wir auch eine entsprechend *große Prozessor-Geschwindigkeit*.

  Gerät man hierbei in Zeitprobleme, kann man immer noch das Anzeigefeld verkleinern, denn der benötigte interne Prozessor-Takt ist etwa proportional zur Fläche des Anzeigefeldes.

- Das Programm funktioniert nur korrekt, wenn stets *innerhalb einer Minute* alle Ereignisse eines Zeitabschnittes durchlaufen werden. Das heißt, die Scroll-Geschwindigkeit muss stets entsprechend hoch gewählt werden.

  Dies ist aber ohnehin nötig, da niemand eine ganze Minute lang vor dem Anzeigefeld warten würde, um die aktuellen Ereignisse durchzusehen.

5.2 Mögliche Erweiterungen

- Alternativ zu unserem zeilenweisen Scrollen könnte man auch seitenweises und pixelzeilenweises Scrollen anbieten.
- Eine Erweiterung wäre ein Bedienfeld, über das man die Scroll-Geschwindigkeit und die Art des Scrollens einstellen kann.
- Spaltenweises Scrollen für längere Ereignis-Texte wäre ebenffalls denkbar.

5.3 Aufgabenteilung

Problematisch am Projekt war, dass viele Entscheidungen einander bedingten. Es fiel manchmal schwer Entscheidungen zu treffen, und man fragt sich regelmäßig: „Was wäre wenn...?“
Dadurch war die Aufgabenteilung schwierig. Man musste vieles sowieso gemeinsam besprechen und konnte oftmals nicht die Denkarbeit, sondern nur die Schreibarbeit verteilen.
Jeder war an allen Bereichen des Projektes gleichermaßen beteiligt, mit 3 kleinen Ausnahmen:

- Der Mikrocode wurde überwiegend von Mario Krell entwickelt.
- Das Assembler-Programm wurde überwiegend von Berit Grußien entwickelt.
- Das LED-Anzeigefeld wurde überwiegend von Volker Grabsch beschrieben.

Auch lässt sich das Programm schwer ohne Befehlssatz schreiben. Aber wenn man diesen definiert hat, merkt man beim Programmschreiben, dass andere Befehle vielleicht besser wären oder emuliert werden durch andere Befehle.
So haben wir abwechselnd das Programm und den Befehlssatz gleichermaßen weiterentwickelt. Manchmal mussten wir sogar die Architektur nachträglich anpassen.
Da laut Aufgabenstellung gefordert war, möglichst wenige Befehle zu nutzen, haben wir von Anfang an Wert darauf gelegt, mit nur 4 Befehlen gut auszukommen.