Projektbeschreibung

Assembly Guessing Game

von Carolina Mehret, Daniel Brown und Nico Axtmann

Kurs:	TINF14B2
Vorlesung:	Systemnahe Programmierung
Dozent:	Prof. Dr. Ralph Lausen
Datum:	11.06.2016

Inhaltsverzeichnis

1 Einleitung

Es soll ein Musterratespiel entworfen werden. Bei dem Spiel wird ein zufällig generiertes Muster für eine kurze Zeit auf einer LED Matrix angezeigt. Nach dem die Zeitspanne ablaufen ist, soll der Anwender das angezeigte Bild nachbauen. Dabei stehen dem Anwender 7 Taster zur Verfügung mit welchen das Muster nachgebaut werden kann. Anschließend kann die Eingabe durch einen weiteren Taster abgegeben werden. Falls die Eingabe dem generierten Muster gleicht, erhält der Anwender einen Punkt. Andernfalls, wird bei einer falschen Eingabe dem Anwender ein Punkt abgezogen. Der Anwender startet mit 3 Punkten. Wenn er 5 Punkte erreicht hat, gewinnt. Bei dem erreichen von 0 Punkten ist das Spiel vorbei und der Anwender hat verloren.

2 Grundlagen

In diesem Kapitel werden die Werkzeuge erläutert, die zur Entwicklung des Programms benutzt werden.

2.1 Assembler

Das Spiel wird in der hardwarenahen Sprache Assembler entwickelt. Mit dieser Sprache wird der Prozessor direkt angesprochen und es gibt keine Abstraktionsschicht wie in den üblichen Hochsprachen. Häufige Operationen sind unter anderem das Setzen von Bits in Registern, Addieren zweier Ganzzahlen oder das Einlesen von simplen Eingaben über Taster oder Sensoren. Heutzutage wird Assembler-Code überwiegend aus Hochsprachen wie C mit hardwarespezifischen Compilern generiert.
Trotzdem findet die Assembler-Programmierung immer noch in einem großen Bereich der IT statt: Entwicklung von Anlagesteuerungen.
Bei überschaubarer Programm-Logik und geringen Hardw areanforderungen macht es durchaus Sinn einen kleinen, für eine spezielle Aufgabe programmierten Microcontroller zu verwenden. Mit dem Aufschwung des "Internet of Things" und somit dem Anstieg der internetfähigen Endgeräte, wird diese Branche wohl in Zukunft noch weiter ansteigen.

2.2 Befehlssatz

Im Folgenden wird ein Ausschnitt des für uns relevanten Befehlssatz aufgelistet.

Befehl	Beschreibung
ACALL <addr11>	Ruft die Subroutine an der Adresse <addr11> auf
ADD <A>,<Operand>	Addiert den <Operand> zum Inhalt des Akkumulators <A> hinzu

ADDC <A>,<Operand>	Addiert den <Operand> und das Übertragsbit zum Inhalt des Akkumulators <A> hinzu
AJMP <addr11>	Springt zu Adresse <addr11>
ANL <Zielbyte>, <Quellenbyte>	Speichert eine bitweise logische UND-Verknüpfung zwischen dem <Zielbyte> und dem <Quellenbyte> im <Zielbyte>
CJNE <Operand1>, <Operand2>,<rel>	Springe zu <rel> falls die Werte <Operand1> und <Operand2> ungleich sind
CLR <bit>/<A>	Löscht das Bit <bit> bzw. den Akkumulator <A>
CPL <bit>/<A>	Komplementiert das Bit <bit> bzw. den Akkumulator <A>
DA <A>	Korrigiere den Dezimalwert des Akkumulators <A> nach einer Addition
DEC <byte>	Dekrementiere <byte> um 1
DIV <A>,<B>	Dividiere Akkumulator <A> durch Register <B>
DJNZ <byte>,<rel>	Dekrementiere <byte> um 1 und springe zu <rel> wenn das <byte> nicht Null ist
INC <byte>/<DPTR>	Inkrementiere <byte> bzw. <DPTR> um 1
JBC <bit>,<rel>	Springe zu <rel>, wenn <byte> gesetzt ist (=1) und lösche dieses anschließend
JC <rel>	Springe zu <rel>, wenn das Übertragsbit gesetzt ist (=1)
JMP <A>+<DPTR>	Addiere den Akkumulator <A> zum Datenanzeiger <DPTR> und lade das Ergebnis in den Programmzähler
JNB <bit>,<rel>	Springe zu <rel>, wenn <byte> nicht gesetzt ist (=0)
JNC <rel>	Springe zu <rel>, wenn das Übertragsbit nicht gesetzt ist (=0)
JNZ <rel>	Springe zu <rel>, wenn der Akkumulator nicht Null ist
JZ <rel>	Springe zu <rel>, wenn der Akkumulator Null ist
LCALL <addr16>	Ruft bedingungslos Subroutine an der Adresse <addr16> auf
LJMP <addr16>	Springt bedingungslos zur Adresse <addr16>

MOV <Zielbyte>, <Quellenbyte>	Kopiere das <Quellenbit> in das <Zielbit>
MOV <DPTR>,<data16>	Lade die Konstante <data16> in den Datenzeiger <DPTR>
MUL <A>,<B>	Multipliziere den Akkumulator <A> zum Register <B>
NOP	Setze Programm mit folgendem Befehl fort
ORL <Zielbyte>, <Quellenbyte>	Speichert eine bitweise logische ODER-Verknüpfung zwischen dem <Zielbyte> und dem <Quellenbyte> im <Zielbyte>
POP <byte>	Bringe den Wert vom Stack Pointer zum Byte <byte> und dekrementiere den Stack Pointer
PUSH <byte>	Kopiere den Wert vom Byte <byte> in den Stack Pointer und inkrementiere den Stack Pointer
RET	Springt aus der Subroutine zurück
RETI	Springe aus dem Interrupt zurück
RL <A>	Schiebe die Bits des Akkumulators <A> nach links
RLC <A>	Schiebe die Bits samt Übertragsbit des Akkumulators <A> nach links
RR <A>	Schiebe die Bits des Akkumulators <A> nach rechts
RRC <A>	Schiebe die Bits samt Übertragsbit des Akkumulators <A> nach rechts
SETB <bit>	Setze das Bit <bit>
SJMP <rel>	Springe unbedingt zu <rel>
SUBB <A>,<Operand>	Subtrahiere den Operanten <Operand> vom Akkumulator <A>
SWAP <A>	Vertausche Halbbytes im Akkumulator <A>
XCH <A>,<Byte>	Speichert eine bitweise logische Exklusiv-ODER-Verknüpfung
XRL <Zielbyte>, <Quellenbyte>	Speichert eine bitweise logische Exklusiv-ODER-Verknüpfung zwischen dem <Zielbyte> und dem <Quellenbyte> im <Zielbyte>

2.3 Der 8051 Microcontroller

Die Zielplattform für das Programm ist der von Intel entwickelte Microcontroller "8051" bzw. die Prozessorarchitektur der Microcontroller-Familie "MCS-51"[1]. Diese Hardwarekomponenten vereinen Prozessor, Programmspeicher, Datenspeicher und Ein- und Ausgabeeinheiten in einem Bauteil, was sie sehr geeignet für einfache Steuerungen und andere Hardwareprojekte macht. Peripheriegeräte wie bspw. Sensoren, Taster, LED-Leuchten, etc. werden über die vier 8-bit Ports angesteuert. Der Datenspeicher hat vier Registerbänke mit jeweils 8 Registern, die je 1 Byte speichern können[2]. Diese Register können als ganzes Byte als auch über einzelne Bits adressiert werden.

2.4 MCU 8051 IDE

Die Entwicklung des Spiels erfolgt in der Entwicklungsumgebung MCU 8051 IDE von Moravia Microsystems. Sie ist für die Entwicklung unter MCS-51 Microcontrollern ausgelegt und liefert viele Komfortfunktionen, wie z.B. Code-Highlighting, Syntaxprüfung, Auto-Vervollständigung und Code-Optimierung. Ein weiterer großer Vorteil dieser Entwicklungsumgebung ist, dass sie Komponenten zur Simulation bereitstellt. Diese GUI-Komponenten werden in MCU 8051 IDE als "Virtual Hardware Simulators" bezeichnett[3]. Dies erspart Entwicklern die Beschaffung der konkreten Hardware zur Entwicklung und vereinfacht den Entwicklungsprozess. Es müssen keine Programmdaten auf einen realen Prozessorspeicher installiert werden und Benutzereingaben geschehen über bequeme Mausklicks. Im Folgenden werden die für dieses Projekt relevanten Simulations-Komponenten näher erläutert.

2.2.1 Simple Keypad

Die Eingabe des Spielers wird über die Simple Keypad Komponente simuliert. Hierbei handelt es sich um eine Reihe von acht Tastern, die an Port 1 angelegt sind. Während der Programmsimulation kann der Anwender mit der Maus auf das gewüschte Taster A-H klicken, um einen Tastendruck zu simulieren. Der Taster an Port 1.7 hat gerade den Wert 0.

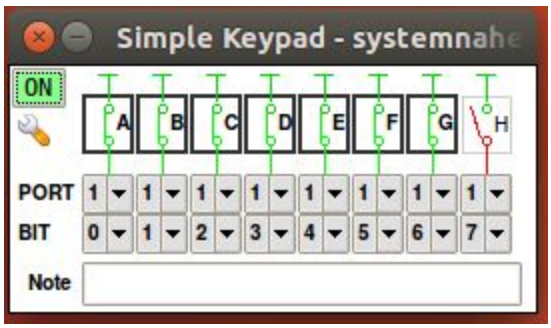

Abbildung 1: Simple Keypad

[1]Wikipedia – Intel MCS-51 – 07.06.2016 – https://de.wikipedia.org/wiki/Intel_MCS-51
[2]Microcontroller.net – 8051 – 07.06.2016 – http://www.mikrocontroller.net/articles/8051
[3]Moravia Microsystems – MCU 8051 IDE Homepage – 09.06.2016 – http://www.moravia-microsystems.com/mcu-8051-ide/

2.2.2 LED Display

Die nachzubildende Ziffer wird auf einer 7-Segment-Anzeige dargestellt. Die Bitmuster für die jeweiligen Zahlen werden im aus dem Programmcode gelesen, an die Ports weitergeleitet und diese stellen dann die gewünschte Ziffer auf der angeschlossenen Anzeige dar. Der entsprechende Virtual Hardware Simulator in der Entwicklungsumgebung heißt LED Display. Im Beispiel in Abbildung 2 ist das Display an Port 0 angelegt.

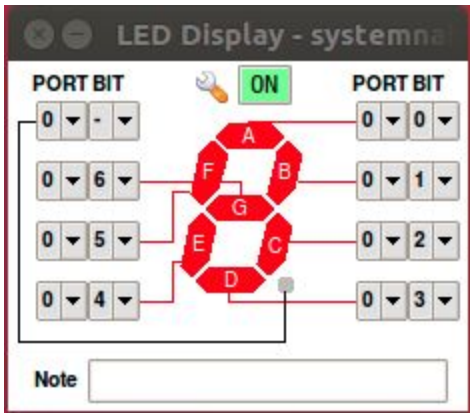

Abbildung 2: LED Display

2.2.3 LED Matrix

Die Ausgabe des Spielstandes wird über ein Raster von Lämpchen realisiert. Anders als bei einer 7-Segment-Anzeige werden hier die Segmente nicht direkt ein- und ausgeschaltet, sondern über Zeilen- und Spalten-Bits angesteuert. Sind die Port-Bits einer gewissen Spalte und Zeile gesetzt, so leuchtet die jeweilige LED auf. In der MCU 8051 IDE lässt sich dies durch die Komponente LED Matrix simulieren.

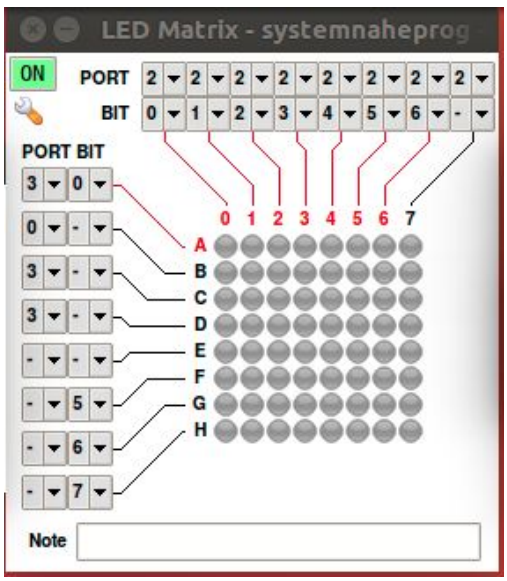

Abbildung 3: LED Matrix

3 Konzept

Abbildung 4 beschreibt den Ablaufplan des Spiels. Vor dem Start werden sollen Variablen und Ports gesetzt werden und danach kann das Spiel gestartet werden. Danach folgen Überprüfungen, ob der Spieler bereits gewonnen oder verloren hat. Ist dies der Fall ist das Spiel beendet und es wird wieder bei der Initialisierung gestartet. Danach wird ein Zufallsmuster auf der LED-Matrix erzeugt, welches nach einer gewissen Zeit wieder verschwindet. Nach diesem Schritt kann der Benutzer Eingaben machen und versuchen, das gezeigte Muster nachzukonstruieren. Sobald die Eingabe vom Benutzer bestätigt wurde wird diese Eingabe mit dem zuvor erzeugten Zufallsmuster abgeglichen. Im nachfolgenden Schritt soll der Punktestand, je nach dem, ob die Eingabe korrekt oder falsch war, erhöht oder erniedrigt werden. Danach wird wieder von vorne gestartet.

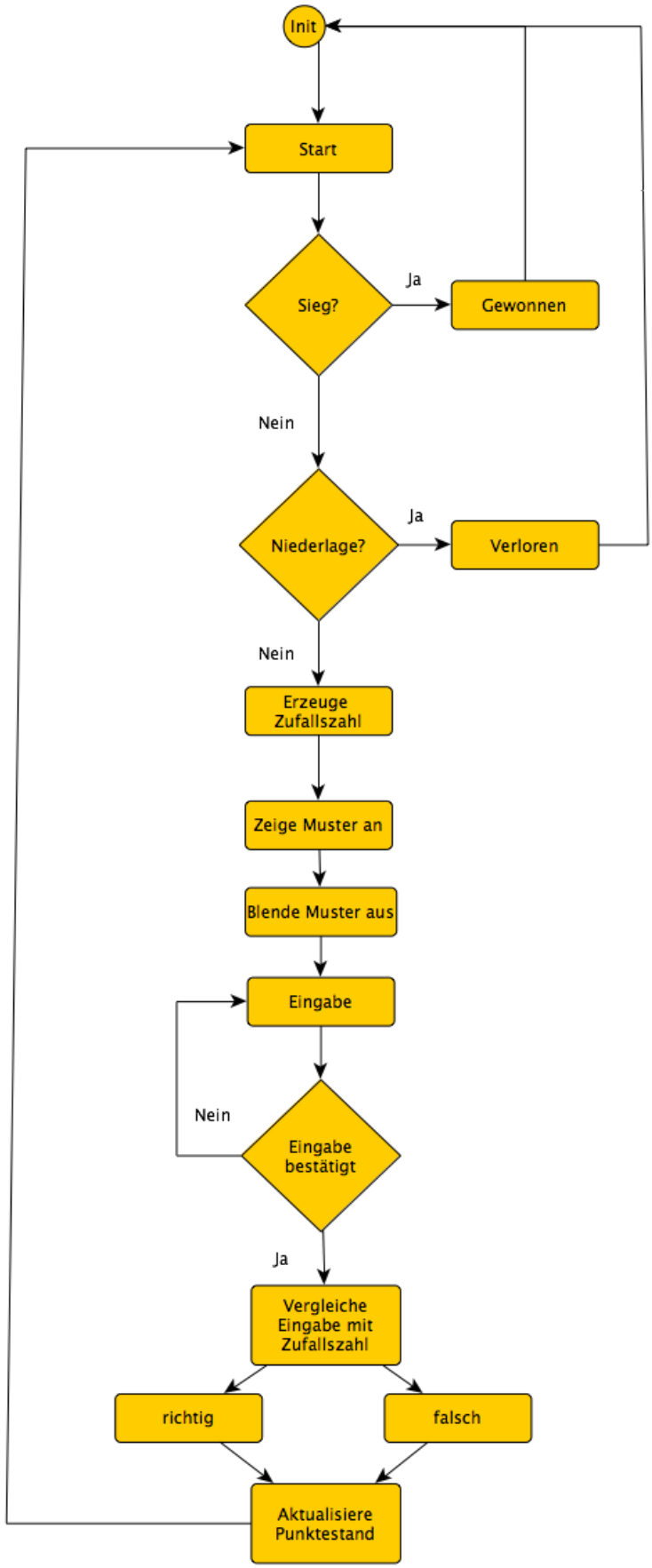

Abbildung 4: Programm-Ablaufplan

4 Implementierung

```
10 score_board DATA P0
11 input DATA P1
12 output_x DATA P2
13 output_y DATA P3
14
15 ;Constanly enable 1 to A in LED matrix
16 mov output_y, #01h
17 jmp init
18
```

Listing 1: Aliasse

Bevor die eigentliche Logik durchlaufen wird, werden zunächst Variablen auf die Ports gesetzt. Der Port P0 wird der Variable score_board zugeordnet. Der Anschluss des score_board ist eine 7 Segment LED Anzeige mit der die Punkte angezeigt werden. Die hierbei verwendete Anzeige wird als Kathode konfiguriert. Der Port P1 wird der Variable input zugeordnet. Der input wird durch ein Simple Keypad simuliert, wodurch die Benutzereingabe definiert wird. Der Port P2 sowie der Port P3 werden für die Steuerung der LED Matrix verwendet. Dabei sind die Bits 0-6 von P2 wird die Steuerung der Spalten zuständig. Beim P3 wird nur das 0. Bit verwendet um die erste Zeile zu steuern. In der Code Zeile 16 wird P3.0 dauerhaft auf 1 gesetzt, da die Zeile nicht von Benutzer angesteuert werden kann.

```
42 init:
43         ;The score of the user is saved in R7
44         ;Starting with score R7 =3
45         mov R7, #03h
46         ;Set 7seg to 3
47         lcall three_seg
48         ;Starting begin simulation
49         lcall animation_anfang
50         mov output_x, #0FFh
```

Listing 2: Label init

Im Register R7 wird der Punktestand gespeichert. Am Anfang des Spiels wird der Punktestand auf 3 Punkte gesetzt. Anschließend wird die Routine three_seg aufgerufen, wodurch die Zahl 3 auf der 7 Segment Anzeige dargstellt wird. Im nächsten Schritt erscheint eine kleine Animation, die den Anfang des Spiels simulieren soll. Anschließend werden die Spalten auf den Wert #FFh gesetzt um die Spalten auszuschalten. Die LED Matrix wurde so konfiguriert das wenn auf der Zeile eine 1 anliegt und auf der Spalte eine 0 anliegt, dass dann die LEDs angehen.

```
52 start:
53         lcall wait
54         mov A, R7
55 checkifwin:
56         jmp iswin
57 checkiflost:
58         cjne A, #00h, random
59         jmp game_over
60
61 iswin:
62         cjne A, #05h, checkiflost
63         jmp win
```

Listing 3: Label `start`

Nachdem das Spiel initialisiert wurde, kann das Spiel gestartet werden. Dazu wird noch eine kurze Warteroutine aufgerufen und anschließend der Wert von R7 in den Akkumulator geschrieben. Als nächstes erfolgt die Überprüfung, ob der Benutzer gewonnen hat. Der Benutzer hat sobald er 5 Punkte erreicht gewonnen. Falls der Wert des Akkumulators ungleich 5 ist, wird auf die Sprungadresse checkiflost verwiesen. Andernfalls wird in die win Sprungadresse gesprungen. Im checkiflost Block wird, falls der der Punktestand ungleich 0 ist, nach random gesprungen. Andernfalls wird nach game_over gesprungen.

```
22 win:
23         mov output_x, #11000000b
24         lcall wait
25         mov output_x, #00110000b
26         lcall wait
27         mov output_x, #00001100b
28         lcall wait
29         mov output_x, #00000011b
30         lcall wait
31         mov output_x, #11000000b
32         lcall wait
33         mov output_x, #00110000b
34         lcall wait
35         mov output_x, #00001100b
36         lcall wait
37         mov output_x, #00000011b
38         lcall wait
39         jmp init
40
```

Listing 4: Label `win`

Im win Block erscheint zunächst eine kleine Animation, die einem doppelten Lauflicht ähnelt. Nach dem Ablauf der Animation wird wieder nach init gesprungen.

```
64 random:
65         dec R2
66         inc R5
67         mov A, R2
68         add A, R5
69         mov R2, A
70         mov output_x, R2
71         mov A, input
72         cjne A, #00h, random
73         mov A, R2
74
75         mov output_x, R2
76         lcall wait
77         mov R0, output_x
78         mov A, #001111111b
79         anl A, R0
80         mov R0, A
81         mov output_x, #0FFh
```

Listing 5: Label `random`

Im random Block wird eine Zufallszahl erzeugt, die anschließend als generiertes Muster dienen soll. Die Register R2 und R5 werden hoch- bzw. runtergezählt. Der aktuelle Wert von R2 wird in den Akkumulator geschrieben und mit dem Wert von R5 addiert. Dieser Wert wird dann R2 zugewiesen und in output_x geschrieben. Dies soll eine Art Slot Maschine simulieren bei der zufällige Muster auf der LED angezeigt werden. Die Zufallsgenerierung durchläuft solange wie der input ungleich 0 ist. Sobald die erste Schleife durch den Benutzer abgebrochen wurde, wird in den Akkumulator der Wert von R2 reingeschrieben. Anschließend wird das Muster, welches binär im Register R2 vorliegt, eine kurze Zeitspanne angezeigt. Da bei der Eingabe nur die Bits 0-6 von P1 zur Überprüfung verwendet werden, muss der Werte von R2 durch die Maske #00111111b durch ein UND abgeschnitten werden. Im Register R0 wird das zu erratende Muster abgelegt.

```
82 eingabe: ;Eingabe des Musters
83 zeile_1:
84         mov A, input
85         mov output_x, A
86         jnb P1.7, zeile_1
```

Listing 6: Label `eingabe`

Die Eingabe des Muster erfolgt anhand den Pins 0-6. Mit dem Pin 7 von P1 wird die Eingabe bestätigt.

```
 87 abfrage:
 88         mov A, R0
 89         cjne A, output_x, falsch
 90
 91 richtig:
 92         mov A, R7
 93         inc A
 94         mov R7, A
 95         lcall select_7seg
 96         jmp start
 97
 98 falsch:
 99         mov A, R7
100         dec A
101         mov R7, A
102         lcall select_7seg
103         jmp start
104
```

Listing 7: Label `abfrage`

Nach der Bestätigung der Eingabe erfolgt die Überprüfung. Bei der Überprüfung wird verglichen ob der Wert von R0 ungleich dem vom Benutzer generierten Muster gleicht. Falls es ungleich ist, bedeutet dies dass das von Benutzer angegebene Muster falsch ist.

```
105 wait:
106         mov R5, #02h
107 w1_s1:
108         mov R6, #001h
109
110 w1_s2:
111         djnz R6, w1_s2
112         djnz R5, w1_s1
113         ret
114
```

Listing 8: Unterprogramm `wait`

Um kurze Wartezeiten wie das Anzeigen des Musters zu ermöglichen wurde eine Warte Funktion implemeniert. Dies wird durch zwei ineinander geschachtelte For Schleifen realisert. Die Register R5 und R6 wird ein Wert zugewiesen und anschließend dekrementiert.

```
115 animation_anfang:
116         mov output_x, #10101010b
117         lcall wait
118         mov output_x, #01010101b
119         lcall wait
120         mov output_x, #10101010b
121         lcall wait
122         ret
```

Listing 9: Unterprogramm `animation_anfang`

Am Anfang des Spiels wird eine einfache Animation simuliert. An der LED Matrix werden abwechselnd verschiedene Muster angezeigt.

```
124 select_7seg:
125         cjne R7, #00h, one_seg
126         mov score_board, #00111111b
127         jmp game_over
128         ret
129
130 one_seg:
131         cjne R7, #01h, two_seg
132         mov score_board, #00000110b
133         ret
134 two_seg:
135         cjne R7, #02h, three_seg
136         mov score_board, #01011011b
137         ret
138 three_seg:
139         cjne R7, #03h, four_seg
140         mov score_board, #01001111b
141         ret
142 four_seg:
143         cjne R7, #04h, five_seg
144         mov score_board, #01100110b
145         ret
146 five_seg:
147         cjne R7, #05h, game_over
148         mov score_board, #01101101b
149         ret
```

Listing 10: Unterprogramm `select_7seg`

Die Darstellung des Punktestands an der 7 Segment Anzeige erfordert eine bestimmte Reihenfolge von bits, da die 7 Segment Anzeige in einer anderen Reihenfolge als die darzustellende Zahl codiert ist. Die Zahlen 0-5 können dargestellt werden. Dabei wird über einen Bedingten Sprung überprüft ob eine der Punktestand in R7 ungleich dem anzuzeigenden Wert ist und falls dies nicht so ist, dann wird der ein bestimmter Wert angezeigt. Falls dies zutrifft, wird in dem nächsten Unterprogramm dieselbe Abfrage durchgeführt.

```
151 game_over:
152         lcall wait
153         lcall animation_anfang
154         lcall wait
155         lcall animation_anfang
156         lcall wait
157         jmp init
158
    end
```

Listing 11: Label `game_over`

Sobald das Spiel zu Ende ist, sprich der Punktestand 0 ist, hat der Spieler verloren und es erscheint eine Animation.

5 Zusammenhang

Es wurde erfolgreich ein Ratespiel in Assembler programmiert. Dadurch wurde die Programmierung in Assembler vertieft und der Umgang mit dem 8051 und der Entwicklungsumgebung erlernt. Die theoretischen Grundlagen lieferte die Vorlesung Betriebssysteme, die parallel dazu gehalten wurde. Durch die Realisierung des Projekts konnten die theoretischen Grundlagen vertieft und angewandt werden.

Quellen

- Wikipedia – Intel MCS-51 – 07.06.2016 – https://de.wikipedia.org/wiki/Intel_MCS-51
- Microcontroller.net – 8051 – 07.06.2016 – http://www.mikrocontroller.net/articles/8051
- Moravia Microsystems – MCU 8051 IDE Homepage – 09.06.2016 – http://www.moravia-microsystems.com/mcu-8051-ide/

Anhang

- Das Projekt wird in GitHub verwaltet: https://github.com/Nixoxo/assemblyguessinggame/
- Programmcode: https://github.com/Nixoxo/assemblyguessinggame/blob/master/nachbauen.asm