Pong 8051 TINF16B4

Bericht Systemnahe Programmierung

Maurice Heumann, Mirko Müller, Alexander Rengers

Inhaltsverzeichnis

Einleitung

Motivation

Im Zuge der Vorlesung „Systemnahe Programmierung“ konnten wir verschiedene Grundlagen in Assemblerprogrammierung erlernen. Dieses Projekt soll nun eine eigene Anwendung des Erlernten darstellen, indem wir unser neues Wissen in eine funktionierende Anwendung umsetzen.

Aufgabenstellung

Die Aufgabenstellung war es, eine eigene Anwendung mit der Assembler-Programmiersprache zu erstellen. Diese Anwendung soll auf der 8051-Prozessorarchitektur basieren und mit Hilfe der MCU-8051-IDE erstellt werden.

Grundlagen

Assembler

Eine Assemblersprache, kurz auch Assembler genannt, ist eine Programmiersprache, die auf den Befehlsvorrat eines bestimmten Computertyps (d. h. dessen Prozessorarchitektur, hier Intel-8051) ausgerichtet ist.

Assemblersprachen bezeichnet man deshalb als maschinenorientierte Programmiersprachen und weiterhin als Programmiersprachen der zweiten Generation: Anstelle eines Binärcodes der Maschinensprache können Befehle und deren Operanden durch leichter verständliche mnemonische Symbole in Textform (z. B. „MOVE“), Operanden z. T. als symbolische Adresse (z. B. „PLZ“), notiert und dargestellt werden.

Der Quelltext eines Assemblerprogramms wird mit Hilfe einer Übersetzungssoftware (Assembler) in Maschinencode übersetzt. Dagegen übersetzt in höheren Programmiersprachen (Hochsprachen, dritte Generation) ein sogenannter Compiler abstraktere (komplexere, nicht auf den Prozessor-Befehlssatz begrenzte) Befehle in den Maschinencode der gegebenen Zielarchitektur – oder in eine Zwischensprache.

Der 8051 Mikrocomputer

MCS-51 ist die Bezeichnung einer 1980 von Intel vorgestellten Familie von 8-Bit-Mikrocontrollern. Bei einem Mikrocontroller sind im Optimalfall alle Teile eines Computersystems (Prozessor, Programmspeicher, Datenspeicher und Ein-/Ausgabeeinheiten) in einem einzigen Baustein zusammengefasst. Zu Beginn hatte sie nur drei Mitglieder mit den Bezeichnungen 8051, 8031 und 8751. Beim 8031 befindet sich das ROM in einem externen Baustein, wohingegen es sich beim 8051 und 8751 im Baustein selbst befindet – entweder in einem maskenprogrammierten ROM (8051) oder in einem EPROM (8751). Die Familie wurde zunächst in NMOS-Technologie, nach einigen Jahren dann auch in der heute üblichen CMOS-Technologie hergestellt.

Entwicklungsumgebung MCU-8051 IDE

Die MCU-8051-IDE (Integrated Development Environment

) ist eine frei verfügbare Entwicklungsumgebung für verschiedene Mikrocontroller, die auf der 8051 Architektur basieren. Die Software bietet weiterhin einen eigenen Simulator und Assembler. Weiterhin werden zwei Programmiersprachen unterstützt: Die C- und die Assembler-Programmiersprache. Wir setzten diese IDE ein für die Entwicklung und Simulation unseres Programmes. Dabei war die Hardwaresimulation essentiell um unser Programm zu visualisieren.

Konzept

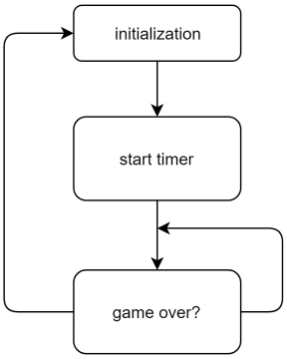
Im Folgenden stellen wir unseren Programmentwurf vor mit der graphischen Darstellung des Programmablaufes. Die Idee unseres Programmes ist es, das bekannte Spiel Pong auf der 8051-Architektur zu realisieren. Dafür verwenden wir eine 8x8-LED-Matrix, zur Darstellung des Balles und der beiden Schläger, sowie eines einfachen Keypads um Eingaben des Spielers entgegenzunehmen.

Analyse

Programmentwurf

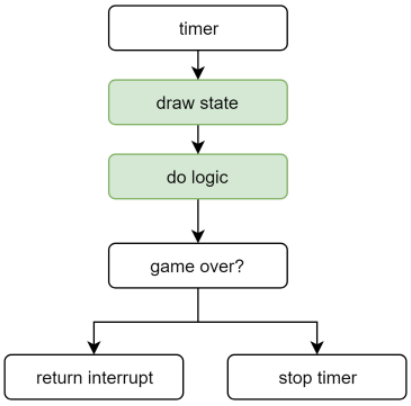
Hauptschleife

In der Hauptschleife wird ein Timer gestartet, der Parallel für die Ausführung der Logik zuständig ist. Anschließend wird in einer Schleife geprüft, ob das aktuelle Spiel vorbei ist und falls ja zurück zum Initialisierungspunkt gesprungen, um das Spiel neu zu starten.



Timer

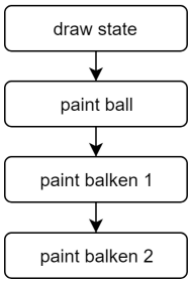
Um das Zeichnen der Objekte mit der Aktualisierungsfrequenz der LED-Matrix zu synchronisieren wird ein Timer benötigt. Zusätzlich zur Zeichenlogik soll auch die Spiellogik verarbeitet werden.



Zeichnen

Die Zeichenlogik unterteilt sich in 3 Schritte:

- Das Zeichnen des Balls als einzelnen Punkt
- Das Zeichnen des linken und
- des rechten Balkens als vertikale Folge von 3 Punkten.

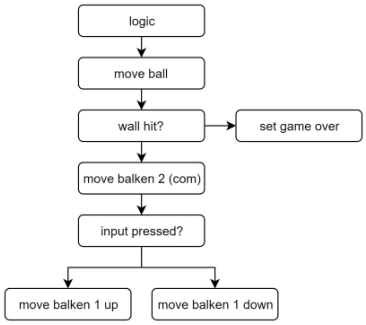


Logik

Die Spiellogik berechnet zunächst die Folgepostition des Balles. Wird dabei die linke Wand berührt, ist das Spiel vorbei und das „game over“-Flag wird gesetzt.

Ansonsten wird der rechte Balken analog zur Position des Balles verschoben.

Der linke Spielerbalken wird entweder nach Oben oder nach Unten verschoben, abhängig davon, ob der Schalter am Keypad gedrückt ist.



Zeichenlogik

Um zeichnen zu können müssen die entsprechenden Bits der Ports der LED-Matrix auf 0 gesetzt werden.

Hierfür müssen die Zielkoordinaten in eine Bitfolge übersetzt werden.

Bei den Koordinaten (3,3) muss beispielsweise das 3. Bit 0 sein und die restlichen 1.

```
bitshift_table:
        db 11111110b
        db 11111101b
        db 11111011b
        db 11110111b
        db 11101111b
        db 11011111b
        db 10111111b
        db 01111111b
```

Das entspricht dem Komplement von 2^3. Normalerweise würde man den Exponenten über Bitshifting lösen, jedoch erlaubt der 8051 nicht über mehrere Bits auf einmal zu shiften.

Der effizientere Ansatz ist daher das Auflösen der Werte über eine Tabelle deren Einträge die Ergebnisse der Exponenzierung beinhaltet.

Implementation

Hier ist unser Programmcode dargestellt, auf dem das Programm basiert.

```
ball_x EQU 30h
ball_y EQU 31h
; directions: rx(0=right/1=left) ry(0=down/1=up)
ball_rx EQU 35h
ball_ry EQU 36h

balken_player EQU 33h
balken_enemy EQU 34h

ende EQU 37h

cseg at 0h
ajmp init
cseg at 100h

org 0bh
call timer
reti

org 20h
init:
call clear_screen
mov ende, #0h
; initialize ball position to be in the middle
; of the screen
mov ball_x, #3h
mov ball_y, #3h
; intialize direction of the ball to downwards
; right
mov ball_rx, #0h
mov ball_ry, #0h
mov balken_player, #0h
mov balken_enemy, #0h
call move_balken
mov ie, #10010010b
mov tmod, #00000010b
mov r7, #00h
mov r6, #05h
mov tl0, #0c0h
mov th0, #0c0h
setb p0.0
setb tr0

run:
mov a, ende
cjne a, #0h, init
jmp run

timer:
call drawing_routine

mov a, ende
cjne a, #0h, end_timer

call do_logic
ret

end_timer:
clr tr0
ret

drawing_routine:
        call paint_ball
        call clear_screen

        mov r1, balken_player
        mov r0, #0h
        call paint_balken
        call clear_screen

        mov r1, balken_enemy
        mov r0, #7h
        call paint_balken
        call clear_screen
        ret

paint_ball:
        mov r0, ball_x
        mov r1, ball_y
        call paint_dot
        ret

paint_balken:
        mov dptr, #bitshift_table
        mov a, r0
        movc a,@a+dptr
        mov p0, a

        mov dptr, #bitshift_table_2
        mov a, r1
        movc a,@a+dptr
        mov p1, a

        ret

move_ball:
        call move_ball_y
        call move_ball_x
        ret

move_ball_y:
        mov a, ball_ry
        cjne a, #0h, change_ball_richtung

        inc ball_y
        mov a, ball_y
        cjne a, #7h, return_move_ball_y
        mov ball_ry, #1h
        jmp return_move_ball_y

        change_ball_richtung:
        dec ball_y
        mov a, ball_y
        cjne a, #0h, return_move_ball_y
        mov ball_ry, #0h
        return_move_ball_y:
        ret
move_ball_x:
        mov a, ball_rx
        cjne a, #0h, check_game_end

        inc ball_x
        mov a, ball_x
        cjne a, #6h, return_move_ball_x
        mov ball_rx, #1h
        jmp return_move_ball_x

        check_game_end:
        djnz ball_x, continue_check_x_pos
        mov ende, #1h
        jmp return_move_ball_x

        continue_check_x_pos:
        mov a, ball_x
        cjne a, #1h, return_move_ball_x
        mov a, ball_y
        mov b, balken_player

        cjne a, b, check_direction_change
        jmp switch_dir

        check_direction_change:
        inc b
        cjne a, b, check_direction_change_2
        jmp switch_dir

        check_direction_change_2:
        inc b
        cjne a, b, return_move_ball_x

        switch_dir:
        mov ball_rx, #0h
        return_move_ball_x:
        ret

move_balken:
        mov a, ball_y
        cjne a, #0h, decrement_y_variable
        jmp check_balken_range_1

        decrement_y_variable:
        dec a

        check_balken_range_1:
        cjne a, #6h, check_balken_range_2
        set_y_max:
        mov a, #5h
        jmp move_both_balken

        check_balken_range_2:
        cjne a, #7h, move_both_balken
        jmp set_y_max

        move_both_balken:
        mov balken_enemy, a
        mov a, p2
        anl a, #1h
        cjne a, #1h, inc_check
        mov a, balken_player
        cjne a, #0h, dec_balken_player
        ret
        dec_balken_player:
        dec balken_player
        ret
        inc_check:
        mov a, balken_player
        cjne a, #5h, inc_balken_player
        ret
        inc_balken_player:
        inc balken_player
        ret

;game logic
do_logic:
        call move_ball
        call move_balken
        ret

paint_dot:
        mov dptr, #bitshift_table
        mov a, r0
        movc a,@a+dptr
        mov p0, a

        mov a, r1
        movc a,@a+dptr
        mov p1, a

        call clear_screen
        ret

; reset dot matrix
clear_screen:
        mov p0, #0FFh
        mov p1, #0FFh
        ret

org 300h
bitshift_table:
        db 11111110b
        db 11111101b
        db 11111011b
        db 11110111b
        db 11101111b
        db 11011111b
        db 10111111b
        db 01111111b

bitshift_table_2:
        db 11111000b
        db 11110001b
        db 11100011b
        db 11000111b
        db 10001111b
        db 00011111b
        db 00111110b
        db 01111100b
end
```

Zusammenfassung

In diesem Projekt konnten wir lernen, wie man die Grundlagen der Assemblerprogrammierung in einer Anwendung umsetzt. Weiterhin lernten wir die differenzierenden Merkmale, die die systemnahe Programmierung ausmachen, kennen und konnten dieses Wissen in diesem Projekt anwenden. Verschiedene Schwierigkeiten wie die Eingabe konnten von uns gelöst werden und die Anwendung Pong konnte zu unserer Zufriedenheit erstellt werden. Sicherlich kann man noch weitere Funktionalitäten implementieren, dies sprengte jedoch den Rahmen dieser Vorlesung.