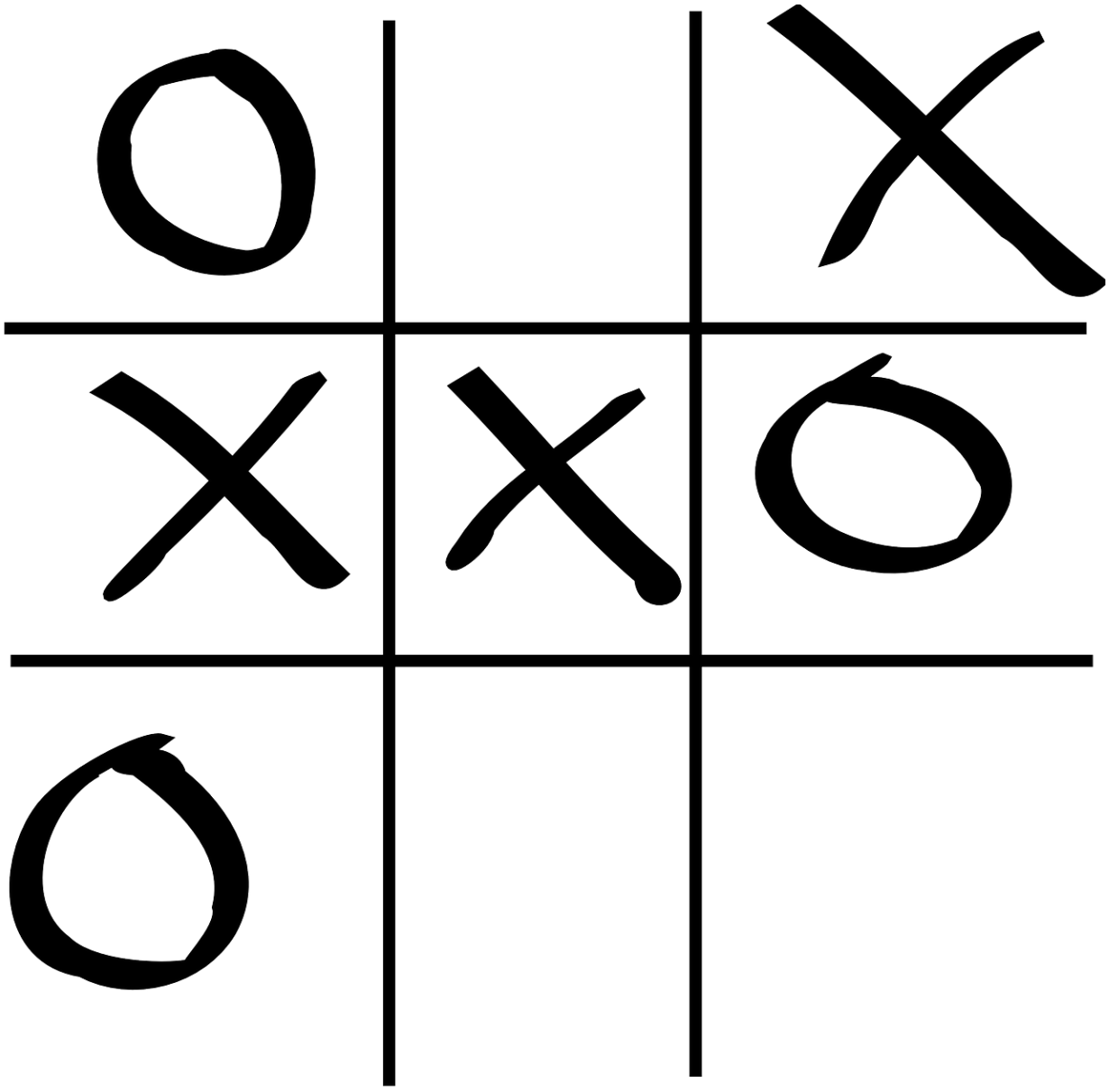
Assembler Tic-Tac-Toe



Mitglieder: Fabian Grimminger (5373238), Sebastian Brehme (7014340), Moritz Wacker (5912894), Robin Warth (6028632)

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Im Rahmen der Vorlesung „Systemnahe Programmierung I“ sollten wir ein Projekt durchführen, um die Programmiersprache Assembler besser kennenzulernen.

Das Projekt wurde in einer emulierten Umgebung für einen Mikrocontroller mit 8051 Architektur erstellt.

1.2. Aufgabenstellung

Die Aufgabenstellung lautete, ein möglichst kreatives Programm in der Entwicklungsumgebung „MCU 8051 IDE“ zu erstellen, unter Verwendung der vom Programm bereitgestellten Ressourcen, wie zum Beispiel der LED Matrix und dem Eingabefeld.

Wir haben uns dafür entschieden, das bekannte Spiel in Tic Tac Toe zu realisieren.

2. Grundlagen

2.1. Assembler

Assembler ist eine sehr hardwarenahe Programmiersprache.
Mit Assembler lassen sich kleine und große Programme für Mikrocontroller oder komplexere Prozessoren schreiben.

Für unterschiedliche Mikrocontroller und Prozessoren gibt es, speziell für diesen Typen angefertigte, Assemblersprachen. Dies erschwert es zum Beispiel, ein einmal geschriebenes Programm auf einer anderen Plattform laufen zu lassen, da es erst an den anderen Assembler angepasst werden muss.

Der Assemblercode wird durch den sogenannten Assembler direkt in ausführbaren Maschinencode umgewandelt.

Programme in Assemblersprache können alle Verarbeitungsmöglichkeiten vom Mikroprozessor nutzen und somit die angeschlossenen Hardwarekomponenten direkt ansteuern.

Assembler bietet, im Gegensatz zu höheren Programmiersprachen, keine Konstrukte wie Schleifen oder Bedingten Anweisungen. Stattdessen muss hier mit Sprüngen gearbeitet werden. Eine weitere Eigenheit ist, dass Vergleiche oft nur mit der 0 möglich sind.

Beispielhaft soll ein if-Statement in Assember geschrieben werden:

if(a == 0){

//do something

}else{

//do something else

}

In Assembler sind dafür bedingte Sprünge nötig, so kann man springen wenn ein Wert 0 oder nicht 0 ist. Damit wird dann über den Teil gesprungen, welcher nicht ausgeführt werden soll. Zur Vereinfachung bezeichnen in diesem Beispiel die Sprungziele die jeweiligen Zeilen.

```
1 jnz a 4; Springe wenn a ungleich 0
2 ; der then teil
3 jmp 5; über den else teil springen, der als nächstes kommt
4 ; der else teil
5 END ;Ende
```

2.2. Der 8051 Mikrocomputer

Der 8051 Mikrocomputer (Intel MCS-51) ist ein 1980 von Intel veröffentlichter Mikrocontroller der 8-Bit-Familie. Er besitzt bis zu 64kB externen Daten- und Programmspeicher und hat 128 Byte RAM. Außerdem sind 2 Timer, 2 Interrupts und 4 8-bit I/O Ports vorhanden.

Bei diesem Mikrocomputer sind Befehlsspeicher und Datenspeicher logisch getrennt, werden aber über einen Bus angesprochen. Dementsprechend kann nicht endgültig geklärt werden, ob es sich hierbei um einen Vertreter der Harvard-Architektur oder der Von-Neumann-Architektur handelt. Über gewisse Tricks ist es hier möglich, auch Code aus dem Datenspeicher auszuführen, was Standardmäßig nicht geplant ist

Intel hat den MCS-51-CPU-Kern schon an viele Halbleiterhersteller lizensiert, wodurch der 8051 zu einem herstellerübergreifenden Industriestandard wurde.

2.3. Entwicklungsumgebung MCU-8051 IDE

Die Entwicklungsumgebung „MCU-8051 IDE" ist eine frei Verfügbare integrierte Entwicklungsumgebung von Martin Osmera.

Die Entwicklungsumgebung erschien erstmals 2007 in Version 0.8 und ist mittlerweile in Version 1.4.9 nutzbar, wie auch wir sie in unserem Projekt verwenden.

Die IDE besitzt einen eigenen Simulator sowie auch Assembler.
Außerdem werden die Programmiersprachen Assembler und C unterstützt.

Eine der Kernfunktionen ist der Simulator, da er viele brauchbare Features mit sich bringt, wie einen eindrucksvollen Debugger mit „register status, step by step, interrupt viewer, external memory viewer, code memory viewer“[1]

Auf „Abbildung 2“ können sie eine Bildschirmaufnahme von MCU-8051 IDE sehen.
Darauf erkennt man einige der Features des Programms, wie zum Beispiel Emulatoren für 7-Segmentanzeigen, LCD Displays und LED’S.

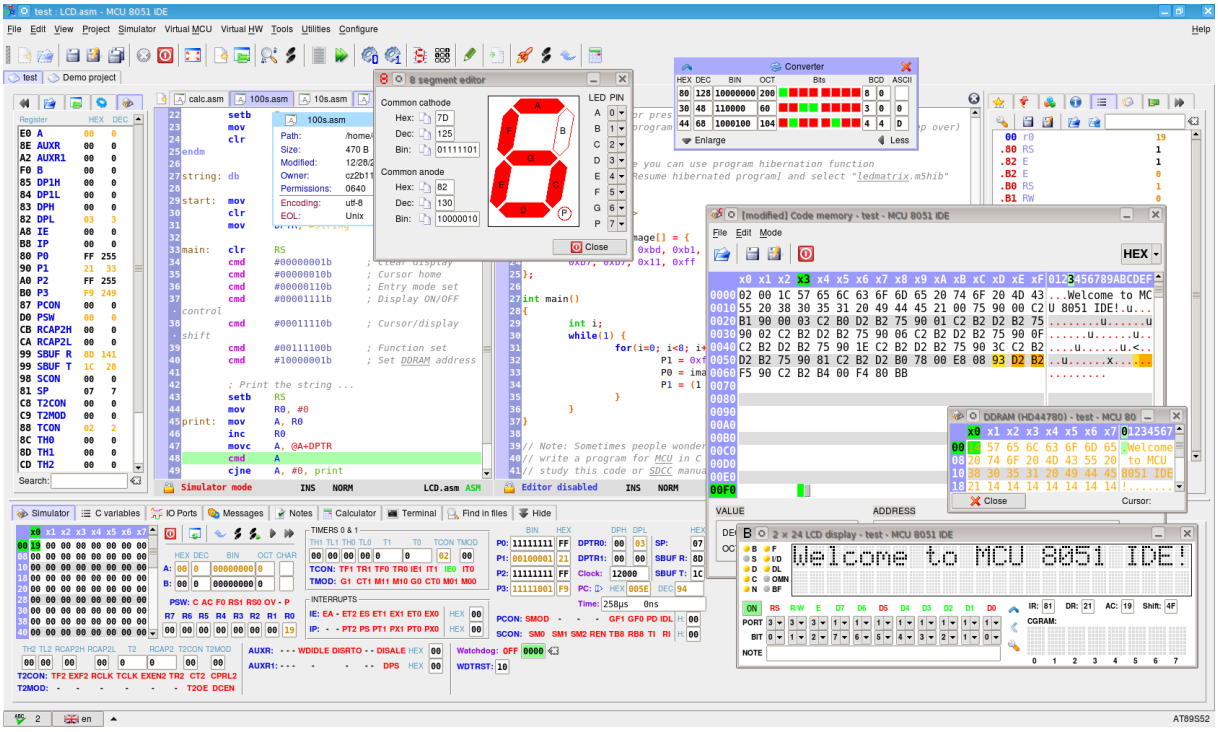


Abbildung 2

[1] https://en.wikipedia.org/wiki/MCU_8051_IDE

2.4. Spielprinzip Tic-Tac-Toe

Tic-Tac-Toe ist ein einfaches Zweipersonen Strategiespiel, welches schon vor mindestens 800 Jahren existierte.

Das Spielfeld besteht aus neun Quadratischen Feldern welche 3x3 angeordnet sind.

Beide Spieler setzen abwechselnd ihre Zeichen, in jeweils eines der Spielfelder, in einem Zug. Um die beiden Spieler unterscheiden zu können benutzt der eine Spieler Kreuze, während der andere Spieler Kreise in die Felder mahlt.

Der Spieler der es als erstes geschafft hat, drei Zeichen in eine Spalte, Zeile, oder Diagonale zu setzen, hat gewonnen.

Spielen jedoch beide Spieler „optimal“, kommt es zu einem Unentschieden, wodurch dann alle neun Felder belegt sind, ohne dass ein Spieler seine drei Zeichen in eine Spalte, Zeile, oder Diagonale gebracht hat.

3. Konzept

3.1. Simulierte Hardware

Um mit dem Benutzer interagieren zu können, benutzen wir verschiedene von MCU-8051 IDE zur Verfügung gestellte simulierte Hardware, wie die LED Matrix und das Matrix Keypad.

Die LED Matrix benutzen wir für die Anzeige des Spielfelds und zur Anzeige der gesetzten Zeichen von Spieler1 und Spieler2.

Das Matrix Keypad dient zur Eingabe der Zeichen durch den Benutzer.

3.2. Entwurf

Anders als beim klassischen Tic Tac Toe, benutzen wir keine Kreise und Kreuze um ein gesetztes Feld zu markieren, sondern einen horizontalen und einen vertikalen „Strich“.
Ein Strich setzt sich aus zwei leuchtenden nebeneinander angeordneten LEDs zusammen.

Auf einen Kreis und ein Kreuz mussten wir, aufgrund des begrenzten gebotenen Platz einer 8x8-LED-Matrix, verzichten.

Das Spielfeld wird durch ein Raster von LEDs dargestellt. (siehe Abbildung 3 unten)

Das Matrix Keypad benutzen wir für die Eingabe durch die Spieler um Felder mit ihren Zeichen zu besetzen. (siehe Abbildung 4 unten)

Die Schalter eins bis neun, dienen dem Spieler zum Setzen eines Feldes mit seinem Zeichen (Spieler1: 2 LEDs horizontal; Spieler2: 2 LEDs vertikal)

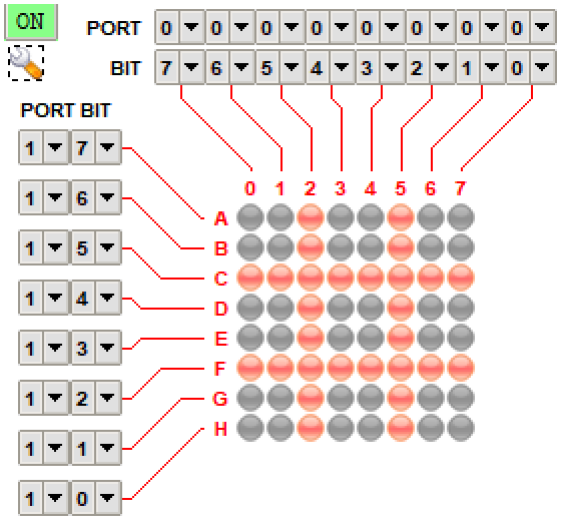


Abbildung 3

Wenn das Spiel beendet wurde, werden alle LEDs ausgeschaltet und der Benutzer kann mit dem Schalter D ein neues Spiel beginnen – Ein neues leeres Spielfeld erscheint.

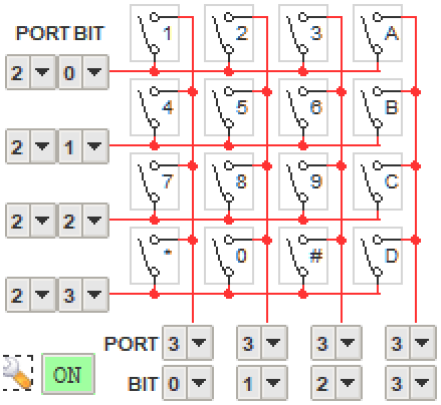


Abbildung 4

4. Implementation

Unterprogramme:

RESET	Speicherzellen, in denen die gesetzten Felder mit ihrem Wert vermerkt sind, werden zurückgesetzt – Neustart, Leeres Spielfeld
PLAYER1	Spieler1 hat gewonnen, Programm wartet bis der Benutzer den RESET-Button „D“ drückt, um das Spiel neu zu starten
PLAYER2	Spieler2 hat gewonnen, Programm wartet bis der Benutzer den RESET-Button „D“ drückt, um das Spiel neu zu starten
DISPLAY	Rufe SHOW auf um das Spielfeld aufzubauen. Prüfe nacheinander die Zeilen, Spalten und Diagonalen, um zu sehen, ob ein Spieler gewonnen hat. Wenn ein „Treffer“ gelandet wird, wird zu Spieler1 oder Spieler2 gesprungen, je nach dem wer gewonnen hat.
SHOW	„Schaltet“ die LEDs ein, indem die entsprechenden Ports aktiviert werden. Die Daten welche Felder von welchem Spieler gesetzt wurden werden aus den entsprechenden Speicherzellen (0x18, 0x19, 0x1A, 0x20, 0x21, 0x22, 0x28, 0x29, 0x2A) gelesen. Hier steht der Wert „03“ für Spieler1 und „04“ für Spieler2. Anschließend wird das Raster der LEDs für das Spielfeld gesetzt.
ONE; TWO; THREE; FOUR; FIVE; SIX; SEVEN; EIGHT; NINE	Wird aufgerufen, wenn entsprechender Schalter vom Spieler gedrückt wurde. Prüfe ob Speicherstelle für aktivierten Schalter mit jeweiliger Nummer schon gesetzt wurde. Gesetzt: Springe zu DISPLAY Nicht Gesetzt: Setze Wert von Spieler, welcher am Zug ist („03“ oder 04“) für die, dem Schalter zugeordnete, Speicherstelle. Prüfe dann welcher Spieler gerade am Zug war und wechsle zu anderem Spieler.

5. Zusammenfassung

Zusammenfassend kann man sagen, dass das Projekt gut aber nicht reibungslos abgelaufen ist.

Es gab kleinere Probleme wie den Platzbedarf eines Tic-Tac-Toe Feldes in einer 8x8 LED Matrix, die Reichweite eines Jumps, oder die Wahl der zu belegenden Speicherstellen.

Im Nachhinein kann man aber durchaus sagen das eine 8x8 LED Matrix nicht ausreicht um ein „schönes“ Tic-Tac-Toe Spiel darauf anzuzeigen und man daher davon abraten kann, diese Simulation des Spiels real nachzubauen mit echten Bausteinen.

Zu Beginn machte uns der Jump Befehl Ärger, da es damit nicht möglich ist beliebig weit zu springen, sondern die Sprungweite limitiert ist. Dieses Problem konnte jedoch schnell umgangen werden.

Durch den Aufruf des Calls wurden Werte in den Speicherzellen verändert, womit unsere Daten überschrieben wurden. Aus diesem Grund wurde ein anderer Speicherbereich ausgewählt, sodass ein überschreiben und damit eine Fehlerhafte Ausführung des Programms verhindert wird.

Insgesamt regt das Programmieren in Assembler auch zum Umdenken an, da man gewohnte Strukturen aus höheren Sprachen nicht direkt wiederverwenden kann. Es gibt weder Bedingungsstatements noch Schleifenkonstruke, sodass man entsprechend mit Jumps arbeiten muss. Auch das Vergleiche nur mit 0 gemacht werden war ungewöhnlich. Es gelang aber schnell, sich auf die neuen Gegebenheiten einzustellen und das Programm umzusetzen.

6. Literatur

6.1. Textquellen

MCU 8051 IDE	https://en.wikipedia.org/wiki/MCU_8051_IDE
Intel MCS-51	https://en.wikipedia.org/wiki/Intel_MCS-51
Systemnahe Programmierung – Prof. Dr. Ralph Lausen	https://else.dhbw-karlsruhe.de/moodle/course/view.php?id=3540

6.2. Bildquellen

Abbildung 1 - Deckblatt	https://pixabay.com/de/tic-tac-toe-spiel-quadrats-spielen-150614/
Abbildung 2	https://en.wikipedia.org/wiki/MCU_8051_IDE
Abbildung 3	Screenshot
Abbildung 4	Screenshot

7. Anhang

7.1. Assembler Quellcode

```
mov B, #03h ; id for Player 1, Player 2 would be 4
JMP DISPLAY

RESET: ; reset storage for selected fields
mov 0x18, #00h
mov 0x19, #00h
mov 0x1A, #00h
mov 0x20, #00h
mov 0x21, #00h
mov 0x22, #00h
mov 0x28, #00h
mov 0x29, #00h
mov 0x2A, #00h
mov B, #03h
JMP DISPLAY

PLAYER1: ; Player 1 wins, stay in this loop until "reset" is klicked
CLR P2.3
JNB P3.3, RESET
JMP PLAYER1

PLAYER2: ; Player 2 wins, stay in this loop until "reset" is klicked
CLR P2.3
JNB P3.3, RESET
JMP PLAYER2

DISPLAY:
CALL SHOW
; check first row
mov A,  0x18
ADD A,  0x19
ADD A,  0x1A
SUBB A, #09h
JZ PLAYER1
SUBB A, #03h
JZ PLAYER2

; check second row
mov A,  0x20
ADD A,  0x21
ADD A,  0x22
SUBB A, #09h
JZ PLAYER1
SUBB A, #03h
JZ PLAYER2

; check third row
mov A,  0x28
ADD A,  0x29
ADD A,  0x2A
SUBB A, #09h
JZ PLAYER1
SUBB A, #03h
JZ PLAYER2
```

```
; check first column
mov A,  0x18
ADD A,  0x20
ADD A,  0x28
SUBB A, #09h
JZ PLAYER1
SUBB A, #03h
JZ PLAYER2

; check second column
mov A,  0x19
ADD A,  0x21
ADD A,  0x29
SUBB A, #09h
JZ PLAYER1
SUBB A, #03h
JZ PLAYER2

; check third column
mov A,  0x1A
ADD A,  0x22
ADD A,  0x2A
SUBB A, #09h
JZ PLAYER1
SUBB A, #03h
JZ PLAYER2

; check left-to-right diagonal
mov A,  0x18
ADD A,  0x21
ADD A,  0x2A
SUBB A, #09h
JZ PLAYER1
SUBB A, #03h
JZ PLAYER2

; check right-to-left diagonal
mov A,  0x1A
ADD A,  0x21
ADD A,  0x28
SUBB A, #09h
JZ PLAYER1
SUBB A, #03h
JZ PLAYER2

mov P0, #11011011b ;set column 2 and 5 to 0
mov P1, #00h ; set all rows to 0 -> now column 2 and 5 light up
mov P1, #11011011b ; set row C and F to 0
mov P0, #00h ; set all columns to 0 -> now row C and F light up

mov P2, #0FFh ; P2 as input set all to 1
mov P3, #0FFh ; P3 as input set all to 1
;check all buttons
;buttons as radio buttons, only one can be set
clr P2.0 ;check row A
jnb P3.0,ONE ; check 0
jnb P3.1,TWO ; check 1
jnb P3.2,THREE; check 2
mov P2, #0FFh ; set all rows to 1

clr P2.1 ;check row B
jnb P3.0,FOUR ; check 0
jnb P3.1,FIVE ; check 1
```

```
jnb P3.2,SIX; check 2
mov P2, #0FFh ; set all rows to 1

clr P2.2 ;check row C
jnb P3.0,SEVEN ; check 0
jnb P3.1,EIGHT ; check 1
jnb P3.2,NINE; check 2
mov P2, #0FFh ; set all rows to 1

jmp DISPLAY

BACK: ;because a jump to display is too long...
jmp DISPLAY

ONE:
MOV A, 0x18 ;get actual stored value
JNZ BACK ; if value != 0 jump
mov 0x18,B ; store B (Player value)
;change player value with xor
mov A,#07h ; load 7 to acc
xrl A,B ; xor A B -> 7 xor 3 = 4, 7 xor 4 = 3
mov B,A ; store new player value in B
JMP BACK ; jump
TWO:
MOV A, 0x19 ;get actual stored value
JNZ BACK ; if value != 0 jump
mov 0x19,B ; store B (Player value)
;change player value with xor
mov A,#07h ; load 7 to acc
xrl A,B ; xor A B -> 7 xor 3 = 4, 7 xor 4 = 3
mov B,A ; store new player value in B
JMP BACK ; jump
THREE:
MOV A, 0x1A ;get actual stored value
JNZ BACK ; if value != 0 jump
mov 0x1A,B ; store B (Player value)
;change player value with xor
mov A,#07h ; load 7 to acc
xrl A,B ; xor A B -> 7 xor 3 = 4, 7 xor 4 = 3
mov B,A ; store new player value in B
JMP BACK ; jump
FOUR:
MOV A, 0x20 ;get actual stored value
JNZ BACK ; if value != 0 jump
mov 0x20,B ; store B (Player value)
;change player value with xor
mov A,#07h ; load 7 to acc
xrl A,B ; xor A B -> 7 xor 3 = 4, 7 xor 4 = 3
mov B,A ; store new player value in B
JMP BACK ; jump
FIVE:
MOV A, 0x21 ;get actual stored value
JNZ BACK ; if value != 0 jump
mov 0x21,B ; store B (Player value)
;change player value with xor
mov A,#07h ; load 7 to acc
xrl A,B ; xor A B -> 7 xor 3 = 4, 7 xor 4 = 3
mov B,A ; store new player value in B
JMP BACK ; jump
SIX:
MOV A, 0x22 ;get actual stored value
JNZ BACK ; if value != 0 jump
mov 0x22,B ; store B (Player value)
```

```
;change player value with xor
mov A,#07h ; load 7 to acc
xrl A,B ; xor A B -> 7 xor 3 = 4, 7 xor 4 = 3
mov B,A ; store new player value in B
JMP BACK ; jump
SEVEN:
MOV A, 0x28 ;get actual stored value
JNZ BACK ; if value != 0 jump
mov 0x28,B ; store B (Player value)
;change player value with xor
mov A,#07h ; load 7 to acc
xrl A,B ; xor A B -> 7 xor 3 = 4, 7 xor 4 = 3
mov B,A ; store new player value in B
JMP BACK ; jump
EIGHT:
MOV A, 0x29 ;get actual stored value
JNZ BACK ; if value != 0 jump
mov 0x29,B ; store B (Player value)
;change player value with xor
mov A,#07h ; load 7 to acc
xrl A,B ; xor A B -> 7 xor 3 = 4, 7 xor 4 = 3
mov B,A ; store new player value in B
JMP BACK ; jump
NINE:
MOV A, 0x2A ;get actual stored value
JNZ BACK ; if value != 0 jump
mov 0x2A,B ; store B (Player value)
;change player value with xor
mov A,#07h ; load 7 to acc
xrl A,B ; xor A B -> 7 xor 3 = 4, 7 xor 4 = 3
mov B,A ; store new player value in B
JMP BACK ; jump

SHOW:
;check if storage is selected and display player icon
mov A, 0x18
SUBB A, #03h
JNZ $+10               ;if !=0 jump over next 3 lines
mov P0, #00111111b     ;else display player1 icon
mov P1, #01111111b
jmp $+12               ; jump over next 4 lines
SUBB A, #01h
JNZ $+8                ;if != 0 jump over next 2 lines
mov P0, #01111111b     ; else display player2 icon
mov P1, #00111111b
;reset view
mov P0, #0FFh
mov P1, #0FFh

;next 8 blocks have the same logic just for the other parts of the storage
mov A, 0x19
SUBB A, #03h
JNZ $+10
mov P0, #11100111b
mov P1, #01111111b
jmp $+12
SUBB A, #01h
JNZ $+8
mov P0, #11101111b
mov P1, #00111111b
;reset view
mov P0, #0FFh
mov P1, #0FFh
```

```
mov A, 0x1A
SUBB A, #03h
JNZ $+10
mov P0, #11111100b
mov P1, #01111111b
jmp $+12
SUBB A, #01h
JNZ $+8
mov P0, #11111101b
mov P1, #00111111b
;reset view
mov P0, #0FFh
mov P1, #0FFh

mov A, 0x20
SUBB A, #03h
JNZ $+10
mov P0, #00111111b
mov P1, #11101111b
jmp $+12
SUBB A, #01h
JNZ $+8
mov P0, #01111111b
mov P1, #11100111b
;reset view
mov P0, #0FFh
mov P1, #0FFh

mov A, 0x21
SUBB A, #03h
JNZ $+10
mov P0, #11100111b
mov P1, #11101111b
jmp $+12
SUBB A, #01h
JNZ $+8
mov P0, #11101111b
mov P1, #11100111b
;reset view
mov P0, #0FFh
mov P1, #0FFh

mov A, 0x22
SUBB A, #03h
JNZ $+10
mov P0, #11111100b
mov P1, #11101111b
jmp $+12
SUBB A, #01h
JNZ $+8
mov P0, #11111101b
mov P1, #11100111b
;reset view
mov P0, #0FFh
mov P1, #0FFh

mov A, 0x28
SUBB A, #03h
JNZ $+10
mov P0, #00111111b
mov P1, #11111101b
jmp $+12
```

```
SUBB A, #01h
JNZ $+8
mov P0, #01111111b
mov P1, #11111100b
;reset view
mov P0, #0FFh
mov P1, #0FFh

mov A, 0x29
SUBB A, #03h
JNZ $+10
mov P0, #11100111b
mov P1, #11111101b
jmp $+12
SUBB A, #01h
JNZ $+8
mov P0, #11101111b
mov P1, #11111100b
;reset view
mov P0, #0FFh
mov P1, #0FFh

mov A, 0x2A
SUBB A, #03h
JNZ $+10
mov P0, #11111100b
mov P1, #11111101b
jmp $+12
SUBB A, #01h
JNZ $+8
mov P0, #11111101b
mov P1, #11111100b
;reset view
mov P0, #0FFh
mov P1, #0FFh

RET
```