BASM Befehlsüberischt und Erklärung

1. In-/Output
    a. out – Zahlen ausgeben
    b. outl – Zahlen ausgeben und neue Zeile ausgeben
    c. ptc – ASCII Zeichen von Nummern ausgeben
    d. inp – Einen Wert entgegennehmen

2. Speicherverwaltung
    a. push – Werte auf den Stack legen
    b. pop – Werte vom Stack löschen
    c. str – Werte vom Stack nehmen
    d. rts – Werte von Variablen/Registern auf den Stack legen
    e. mov – Werte verschieben
    f. ptr - Pointer
    g. hlt – Das Programm beenden

3. Mathematische Berechnungen
    a. add - Addition
    b. mul - Multiplikation
    c. div - Division
    d. sub - Subtraktion
    e. mod – Modulo

4. Variablen
    a. var – Variablen erstellen
    b. arr – Arrays erstellen
    c. aga – Wert an bestimmter Stelle eines Arrays auslesen
    d. ags – Array größe in Register oder Variable speichern
    e. arl – Bestimmte Anzahl von Elementen eines Arrays löschen
    f. asa – Array = anderer Array / x = y
    g. asv – Setzen eines Wertes an einem bestimmen Index
    h. ata – Elemente zum Array hinzufügen

5. Abfragen
    a. eqi – Prüfen auf Gleichheit
    b. nqi – Prüfen auf Ungleichheit
    c. lqi – Prüfen auf x kleiner y
    d. gqi – Prüfen auf x größer y

6. Funktionen und Sprünge
    a. Funktionen deklarieren
    b. jmp – Funktion aufrufen
    c. jnn – Funktion aufrufen, wenn hx nicht 0 ist
    d. jln – Funktion aufrufen, wenn hx kleiner 0 ist
    e. jgn – Funktion aufrufen, wenn hx größer 0 ist
    f. jen – Funktion aufrufen, wenn hx gleich 0 ist

    g. Sprungmarken deklarieren
    h. gt – Zu Sprungmarke springen
    i. gnn – Zu Sprungmarke springen, wenn hx nicht 0 ist
    j. gln – Zu Sprungmarke springen, wenn hx kleiner 0 ist
    k. ggn – Zu Sprungmarke springen, wenn hx größer 0 ist
    l. gen – Zu Sprungmarke springen, wenn hx gleich 0 ist

Vorab

Diese Aufzeichnung baut nicht aufeinander auf. Es werden auch hier oben schon Codes mit Funktionen gezeigt, die erst später kommen.

Außerdem gibt es folgende, vordefinierte und globale Register, die Integer Werte speichern können:

1. AX
2. BX
3. CX
4. DX
5. EX
6. FX
7. GX – Speicher für Input
8. HX – Meist für Sprünge, Abfragen und Funktionen verwendet

Alle Register können normal beschrieben werden, jedoch sollte man mit GX und HX vorsichtiger sein.

Mit Werten sind IMMER Zahlen gemeint. Egal ob es sich gerade um ein Register handelt, woraus die Zahl ausgelesen wird, oder einen String, der dann in einzelne Zeichen gespalten und in ASCII-Zahlen umgewandelt wird.

Die Überschriften zu Funktionen sind so gebaut, dass sie aus dem Befehlsnamen und der Bedeutung der Abkürzung bestehen. Beispiel:

ebü – Eine Beispielüberschrift
enü – Eine neue Überschrift
ddp – der dicke Peter

1 – Input und Output

out - Output

Mit OUT nehmen wir den obersten Wert vom Stack und geben ihn aus.
Beispiel:

```
main:
     push 10
     out
```

Ausgabe:

```
10C:\> _
```

outl – Output Newline

Mit OUTL nehmen wir den obersten Wert vom Stack und geben ihn aus. Dazu geben wir noch eine neue Zeile aus.
Beispiel:

```
main:
     push 10
     outl
```

Ausgabe:

```
10
C:\> _
```

ptc - Putchar

Mit PTC können wir Buchstaben ausgeben. Es wird der oberste Wert vom Stack genommen und dieser wird zu einem ASCII Zeichen umgewandelt:
Beispiel:

```
main:
     push 65
     ptc
```

Ausgabe:

```
AC:\> _
```

So können wir auch einen Zeilenumbruch ausgeben:

```
main:
      push 65
      push 10
      ptc
      ptc
```

Ausgabe:

```
A
C:\> _
```

inp - Input

Mit INP können wir Eingaben vornehmen. Es gibt zwei unterschiede: Zahlen und Strings. Diese werden bei der Eingabe automatisch konvertiert. Wenn ein String eingegeben wird, wird dieser auf Den Stack gelegt und die Länge wird in HX gespeichert. Bei Zahlen wird HX auf 0 gesetzt und die Zahl in GX gespeichert.

Ein Beispiel zu Zahlen, wo die eingegebene Zahl ausgegeben wird:

```
main:
      inp
      push gx
      outl
```

Ein Beispiel zu Strings, wo die Länge des eingegebenen Strings ausgegeben wird:

```
main:
      inp
      push hx
      outl
```

2 - Speicherverwaltung

push - Push

Mit dem Push Befehl kann man Werte auf den Stack legen:

```
main:
      push 10
      push „Hallo Welt“
```

pop - Pop

Mit dem POP Befehl löscht man den obersten Wert vom Stack.
Beispiel, wo erst etwas auf den Stack gelegt, dieses dann wieder gelöscht wird:

```
main:
     push 10
     pop
```

str – Stack to Register

Mit STR ist es Möglich, Werte vom Stack in Variablen oder Register zu laden. Hier wird eine 10 vom Stack in AX gelegt und ausgegeben:

```
main:
     push 10
     str ax
     pop
     push ax
     outl
```

rts – Register to Stack

RTS ist eine Alternative zu Push:

```
main:
     mov 10, ax
     rts ax
     outl
```

Hier könnte man auch

```
main:
     mov 10, ax
     push ax
     outl
```

schreiben.

mov - Move

Mit MOV ist es möglich, Werte überall hinzuschieben. In Register und auch in Variablen.

Beispiel, welches AX auf 10 und dann auf 7 setzt:

```
main:
     mov 10, ax
     mov 7, ax
```

ptr - Pointer

Pointer sind eine Sache, die etwas schwieriger zu verstehen sind. Jede Variable und jedes Register in BASM, wird über einen Zeiger (eng. Pointer) angesprochen. Dieser Zeiger zeigt auf eine Speicheradresse im Speicher.

Ändert man diesen Pointer und schreibt nun etwas in das, bspw., Register, dann landet der Wert woanders.

Hier ein einfaches Beispiel, wo der Pointer von AX auf eine Variable gesetzt wird und dann beschrieben wird:

```
main:
     mov 10, ax
     var _ax
     mov 7, _ax
     ptr ax, _ax
     push ax
     outl
```

Ausgabe:

```
7
C:\> _
```

Das kommt daher, da AX erst auf 10 gesetzt wird. Dann wird eine Variable _ax erstellt, welche danach auf 7 gesetzt wird. Nun wird der Zeiger von AX auf _ax gesetzt. Wenn wir also AX sagen, zeigt er von nun an auf _ax. Wollen wir das rückgängig machen, können wir das so machen:

```
ptr ax, ax
```

Dann zeigt AX wieder auf AX.

hlt - Halt

HLT beendet das Programm.
Beispiel:

```
main:
     push 10
     hlt
     outl
```

Da das Programm vor OUTL beendet wird, wird nichts ausgegeben.

3 – Mathematische Berechnungen

add - Addition

Mit ADD können wir Werte, Register und Variablen addieren.
Beispiel:

```
main:
      mov ax, 10
      add ax, 10

      push ay
      outl
```

mul - Multiplikation &
div – Division &
sub – Subtraktion &
mod - Modulo

MUL - Gleicher Syntax wie ADD, multipliziert jedoch.
DIV - Gleicher Syntax wie ADD, dividiert jedoch.
SUB - Gleicher Syntax wie ADD, subtrahiert jedoch.
MOD - Gleicher Syntax wie ADD, berechnet jedoch den Rest.

4 – Abfragen
eqi – Equals Integer

EQI überprüft zwei Werte auf Gleichheit. Wenn die Abfrage zutrifft, wird HX auf 1 gesetzt. Trifft die Abfrage jedoch nicht zu, wird HX auf 0 gesetzt.
Beispiel:

```
main:
      eqi 1, 2
      push hx
      outl
```

Da 1 nicht 2 ist, wird 0 ausgegeben.

nqi – Not Equals Integer

NQI überprüft zwei Werte auf Ungleichheit. Hier wird das gleiche mit HX angestellt, wie bei EQI.

Beispiel:

```
main:
      eqi 1, 2
      push hx
      outl
```

Da 1 nicht 2 ist, wird 1 ausgegeben.

lqi – Less Equals Integer

LQI überprüft zwei Werte darauf, dass der erste Wert kleiner als der zweite ist. Hier geschieht mit HX das gleiche wie bei EQI.

Beispiel:

```
main:
      eqi 1, 2
      push hx
      outl
```

Da 1 kleiner als 2 ist, wird 1 ausgegeben.

gqi – Great Equals Integer

GQI überprüft zwei Werte darauf, dass der erste Wert größer als der zweite ist. Hier geschieht mit HX das gleiche wie bei EQI.
Beispiel:

```
main:
      gqi 1, 2
      push hx
      outl
```

Da 1 nicht größer als 2 ist, wird 0 ausgegeben.

5 - Variablen

var - Variable

Mit VAR kann man eine neue Variable erstellen, welche Zahlen speichern kann. Diese sind dann nur in einer bestimmten Funktion gültig, sind also nicht global.
Beispiel:

```
main:
      var _ax
```

arr - Array

Mit ARR kann eine neue Liste/ein neuer Array erstellt werden. Diese Liste kann Zahlen speichern und abgeändert werden.

aga – Array Get At

Mit AGA kann man den Wert an einem bestimmten Index von einem Array in eine Variable/in ein Register speichern.
Beispiel:

```
main:
      arr test
      ata test, 10
      ata test, 12
      ata test, 13

      aga test, 1, ax
      push ax
      outl
```

Mit diesem Programm wird ein Array {10, 12, 13} erstellt. Dann wird mit AGA das Element an Index 1 (also 12) in AX geschrieben. AX wird auf den Stack gelegt und das wird dann ausgegeben. Am Ende steht also 12 auf der Konsole.

ags – Array Get Size

Mit AGS bekommen wir die größe eines Arrays.
Beispiel, dass eine leere Liste erstellt und die Länge der Liste ausgibt:

```
main:
      arr test
      ags test, ax
      push ax
      outl
```

Ausgegeben wird 0, da dem Array kein Element hinzugefügt wurde.

arl – Array Remove Element

Mit ARL ist es möglich, Elemente, von hinten, aus einem Array zu löschen. Dabei kann man eine bestimmte Anzahl angeben:

```
main:
      arr test
      ata test, 10
      ata test, 42
      ata test, 1337

      arl test, 3
      ags test, ax
      push ax
      outl
```

Es wird ein Array erstellt: {10, 42, 1337}. Danach werden 3 Elemente (von hinten an) vom Array gelöscht (also alle). Danach wird die größe des Arrays ausgegeben. Am Ende ist also nichts im Array, es steht also 0 da.

asa – Array Set Array

Mit ASA ist es möglich Arrays auf Arrays zu setzen. Das „MOV für Arrays“.
Beispiel:

```
main:
      arr test
      ata test, 10
      arr test2
      asa test, test2
      ags test2, ax
      push ax
      outl
```

Bei diesem Programm gibt es 2 Arrays, einmal test, welcher leer ist, und test2 {10}. test wird auf test2 gesetzt und danach wird die Länge von test

ausgegeben. Da test2 eine Länge von 1 hat und test darauf gesetzt wurde, wird 1 ausgegeben.

asv – Array Set Value

Mit ASA kann man den Wert eines Elementes aus einer Liste an einem bestimmten Index ändern. Beispiel:

```
main:
      arr test
      ata test, 10
      ata test, 42
      ata test, 1337

      asv test, 1, 3
      aga test, 1, ax

      push ax
      outl
```

Zuallererst wird ein Array test erstellt {10, 42, 1337}. Dann wird das Element am Index 1 (42) auf 3 gesetzt. Danach wird das Element am Index 1 (3) in AX geschrieben und ausgegeben. Am Ende steht also 3 auf der Konsole.

ata – Add To Array

Mit ATA kann man Elemente zu Arrays hinzufügen. Beispiel:

```
main:
      arr test
      ata test, 10
      ata test, 42
      ata test, 1337

      aga test, 1, ax
      push ax
      outl
```

Hier wird ein Array erstellt. Danach werden 10, 42 und 1337 hinzugefügt. Danach wird 42 ausgegeben.

6 – Funktionen und Sprünge

Funktionen deklarieren

Wie man sicherlich schon bemerkt hat, kann man Funktionen so deklarieren:

```
name:
```

Damit wurde die Funktion name erstellt. Jedoch muss es in jedem BASM-Programm eine Main-Funktion geben! (also „main“) Dort startet das Programm.

Anmerkung: Mit RETURN kannst du eine Funktion beim Ausführen frühzeitig verlassen.

Hier ein Beispiel dazu:

```
main:
      push 2
      return
      outl
      pop
      hlt
```

Es passiert nichts, außer das eine 2 auf dem Stack liegt. Durch das Return wird die Main-Funktion abgeschlossen und das Programm wird beendet.

jmp - Jump

Mit JMP kannst du eine Funktion aufrufen. Beispiel dazu:

```
sag2:
      push 2
      outl
      pop
      return

main:
      jmp sag2
      hlt
```

Dadurch, dass zu sag2 am Anfang der Main-Funktion gesprungen wird, werden die Befehle in sag2 ausgeführt und es wird eine 2 ausgegeben.

jnn – Jump Not Null

JNN ist eine erweiterte JMP-Funktion, denn JNN springt nur, wenn HX nicht 0 ist.

jln – Jump Less Null

JLN ist auch eine erweiterte JMP-Funktion, denn sie springt nur, wenn HX kleiner als 0 ist.

jgn – Jump Greater Null

JGN ist eine erweiterte JMP-Funktion, da sie nur Funktionen ausführt, wenn HX größer als 0 ist.

jen – Jump Equals Null

JEN gehört auch zu den erweiterten JMP-Funktionen, denn JEN springt nur, wenn HX gleich 0 ist.

Sprungmarken deklarieren

Sprungmarken sind dazu da, um bestimmte Bereiche in Funktionen zu markieren. Wenn man diesen Bereich (noch-) einmal aufrufen möchte, kann man zu dieser Marke Springen. In anderen Sprachen gibt es sowas unter dem Namen „goto".

Beispiel wie man eine Sprungmarke erstellt:

```
main:
      .endless:
            push 10
            outl
            pop
            gt endless
```

Es wird eine Funktion main erstellt, mit einer Sprungmarke „endless". Endless markiert also die Zeile, wohin nachher gesprungen wird. Es wird eine 10 ausgegeben und danach wird zu endless gesprungen. Dann wird wieder 10 ausgegeben und es wird wieder zu endless gesprungen. Dann wird wieder 10 ausgegeben usw.

gt - Goto

Mit GT kann man zu einer Sprungmarke springen. Diese muss jedoch in der gleichen Funktion sein, wo die GT-Anweisung steht, da die Sprungmarke sonst nicht gefunden wird.

Beispiel:

```
main:
      .endless:
            push 10
            outl
            pop
            gt endless
```

Endless-Programm von oben. Mit der Anweisung „gt endless" wird zur Zeile mit der Sprungmarke endless gesprungen und von dort an wird Code ausgeführt.

gnn – Goto Not Null

GNN ist ein erweiterter GT-Befehl. Er springt nur, wenn HX nicht 0 ist.

gln – Go Less Null

GLN gehört auch zu den erweiterten GT-Befehlen, da GLN nur springt, wenn HX kleiner als 0 ist.

ggn – Go Greater Null

GGN ist auch ein erweiterter GT-Befehl, denn er springt nur, wenn HX größer als 0 ist.

gen – Go Equals Null

GEN ist ein erweiterter GT-Befehl, denn er springt nur, wenn HX 0 ist.