

**Mit \$X werden Register bezeichnet und mit X Konstanten, wie 1234, #ABC oder 'A'.**

**Befehlsreferenz**

<b>Arithmetik</b>	
ADD \$Summe, \$A, \$B	ADDITION, \$Summe = \$A + \$B
ADD \$Summe, \$A, B	Oder: \$Summe = \$A + B
Analog SUB, MUL, DIV	SUBtraktion, MULTIplikation, DIVision*
* Es wird zusätzlich der Rest ins Spezialregister rR gespeichert. Der Wert kann mit GET in ein Allzweckregister geladen werden.	
NEG \$X	\$X = -\$X
SET \$X, \$Y	\$X = \$Y
SET \$X, Y	\$X=Y
<b>Sprungbefehle</b> Springe zur Marke,...	
BN \$X, Marke	wenn \$X negativ ist
BNN \$X, Marke	wenn \$X nicht negativ ist
BP \$X, Marke	wenn \$X positiv ist
BNP \$X, Marke	wenn \$X nicht positiv ist
BZ \$X, Marke	wenn \$X null ist
BNZ \$X, Marke	wenn \$X nicht null ist
JMP Marke	in jedem Fall!
<b>Vergleich</b>	
CMP \$X, \$Y, \$Z	\$X = -1, wenn \$Y < \$Z
CMP \$X, \$Y, Z	0, wenn \$Y = \$Z
	+1, wenn \$Y > \$Z
<b>Spezialregister</b>	
GET \$X, SpezialReg	Lädt den Inhalt des Spezialregisters nach \$X. <b>Bsp.:</b> GET \$0, rR
PUT SpezialReg, \$X	Speichert den Inhalt von \$X ins Spezialregister
<b>Adressberechnung</b>	
GETA \$X, Marke	speichert Adresse der Marke in ein Register \$X, Adressierung ist relativ und trifft also nur nahe Marken (±0x7FFF)
LDA \$X, Marke	speichert Adresse der Marke in ein Register \$X, Adressierung ist auch relativ. Trifft nur Marken in deren Nähe ein GREG @ platziert ist.
<b>Speicher</b>	
LDx \$X, \$Y, \$Z	Lädt ein Byte / Wyde / Tetra / Octa von der Adresse in \$Y bei Offset \$Z aus dem Speicher nach Register \$X
LDx \$X, \$Y, Z	
STx \$X, \$Y, \$Z	Lädt den Inhalt von Register \$X in ein Byte / Wyde / Tetra / Octa an der Adresse \$Y bei Offset \$Z im Speicher
STx \$X, \$Y, Z	
x = B für Byte, analog für Wyde, Tetra, Octa Der Z-Operand muss nicht angegeben werden.	

<b>Stack / Unterprogramme</b>	
PUSHJ \$X, Marke	Legt die X ersten Register auf den internen Stack und benennt die folgenden Register um: \$(X+1) wird zu \$0 usw. Dabei wird die Rücksprungadresse ins Spezialregister rJ gespeichert (muss bei Rekursion gemerkt werden!).
POP X, 0	Stellt den Stack wieder her (wobei in die X ersten Register ab dem PUSHJ-Register die Ergebnisse abgelegt werden) und springt zurück zu der Adresse die im Spezialregister rJ steht.
GO \$X, Marke	Springt zur Marke und speichert die Rücksprungadresse in Register \$X.
GO \$X, \$X, 0	Springt zurück zu der in \$X gespeicherten Rücksprungadresse

**Pseudobefehle**

<b>LOC</b> Adresse	
Marke <b>IS</b> abc	Definiert ein Synonym. Alle Vorkommen von <b>Marke</b> im Quellcode werden vom Assembler durch <b>abc</b> ersetzt.
Marke <b>BYTE</b> 0, 1, 2, ...	Legt eine Folge von Bytes/ Wydes/ Tetras/ Octas an aktuellen Adresse mit konkreten Werten an. Das erste Byte kann mit <b>Marke</b> identifiziert werden.
Marke <b>WYDE</b> 0, 1, 2, ...	
Marke <b>TETRA</b> 0, 1, 2, ...	
Marke <b>OCTA</b> 0, 1, 2, ...	
Marke <b>GREG</b> Wert	Legt ein globales Register mit einen bestimmten <b>Wert</b> an. Dieses Register kann anschließend über <b>Marke</b> angesprochen werden. Globale Register bleiben von Stackoperationen unberührt

**Register**

Unter MMIX gibt es 256 Allzweckregister, welche mit \$X adressiert (mit X=0 bis 255). Daneben gibt es Spezialregister, deren Inhalte mit PUT und GET manipuliert werden (z.B. Inhalt in Allzweckregister holen).

**Das @-Symbol**

... steht für die aktuelle Position im Arbeitsspeicher. Das Symbol wird vom Assembler bei Kompilierung durch die absolute Adresse ersetzt.

**Speicheraufteilung**

Der Arbeitsspeicher unter MMIX erstreckt sich über einen Adressraum von 0 bis 2^64-1.

Dieser Raum wird in Segmente gegliedert:

Interrupt-Vektortabelle
<b>Codesegment (MMIX-Befehle)</b> ab Adresse #100
<b>Data-Segment (Daten, z.B. Strings)</b> ab Adresse #20...000 oder Data_Segment
Pool-Segment
Stack-Segment
Segment für das Betriebssystem

Interessant für uns sind nur das Code-Segment und das Datensegment. Das Stacksegment wird zwar auch bei Unterprogrammen genutzt, aber wird von MMIX selbst verwaltet und ist deshalb weniger interessant während der Entwicklung eines Programms.

**Kompilierung und Ausführung**

Kompiliere deine Datei script.mms mit  
> mmixal script.mms

Dies erzeugt eine script.mmo, die von MMIX ausgeführt werden kann:  
> mmix script.mmo

Achtet darauf, dass ihr im richtigen Ordner seid!

## Kontrollstrukturen

### Bedingte Verzweigung

#### In Java:

```
if(x==1) {  
    A  
} else {  
    B  
}
```

#### In MMIX:

```
tmp    IS $0  
x      IS $1  
  
ifLbl  CMP tmp,x,1  
        BNZ tmp,elseLbl  
        //Code von A  
        JMP endLbl  
elseLbl //Code von B  
endLbl //...
```

### While-Schleife

#### In Java:

```
while(x==1) {  
    A  
}
```

#### In MMIX:

```
tmp    IS $0  
x      IS $1  
  
whileLbl  CMP tmp,x,1  
            BNZ tmp,endLbl  
            //Code von A  
            JMP whileLbl  
endLbl    //...
```

### For-Schleife

#### In Java:

```
for(i=0; i<10; i++) {  
    A  
}
```

#### In MMIX:

```
i IS $0  
tmp IS $1  
  
forLbl  SET i,0  
loop    CMP tmp,i,10  
        BNN tmp,endLbl  
        //Code von A  
        ADD i,i,1  
        JMP loop  
endLbl  //...
```

## Minimale MMIX-Programme

### Das kleinste Programm

... besteht aus

- einer LOC #100, damit die nachfolgenden Befehle ins Code-Segment geladen werden
- dem Main-Label, damit MMIX den Einstiegspunkt vom Programm kennt
- dem TRAP 0,Halt,0, damit das Programm beendet wird

```
LOC #100  
Main    TRAP 0,Halt,0
```

### Hallo Welt (nur Codesegment)

Um Nachrichten auszugeben, legt man im Speicher den Nachrichtentext an und übergibt dem Register \$255 die Adresse zu dieser Nachricht. Der neu hinzugekommene TRAP-Befehl sorgt dann für die Ausgabe.

```
LOC #100  
Buffer BYTE "Hallo Welt!",#A,0  
Main   GETA $255,Buffer  
       TRAP 0,Fputs,StdOut  
       TRAP 0,Halt,0
```

### Hallo Welt (mit Datensegment)

Daten sollte man im Datensegment ablegen. Das Code-Segment könnte z.B. aus Sicherheitsgründen schreibgeschützt sein.

Man beachte besonders die Befehle GREG und LDA!

```
LOC Data_Segment  
GREG @  
Buffer BYTE "Hallo Welt!",#A,0  
LOC #100  
Main   LDA $255,Buffer  
       TRAP 0,Fputs,StdOut  
       TRAP 0,Halt,0
```

### Speicherzugriff

Folgendes Programm addiert auf jedes Element im Array eine 1.

```
LOC Data_Segment  
GREG @  
ArraySize IS 10  
Array     BYTE 1,2,3,4,5,6,7,8,9  
LOC #100  
addr      IS $0  
index     IS $1  
tmp       IS $2  
test      IS $3  
Main      SET index,0  
          LDA addr,Array  
  
Loop      CMP test,index,ArraySize  
          BNN test,EndLbl  
  
          LDB tmp,addr,index  
          ADD tmp,tmp,1  
          STB tmp,addr,index  
          ADD index,index,1  
          JMP Loop  
  
EndLbl    TRAP 0,Halt,0
```