ADRESSIERUNG IN MMIX

Beim Programmieren mit MMIX habt ihr vielleicht schon öfter eine der folgenden Fehlermeldungen von MMIXAL bekommen:

“no base address is close enough to the address A!”

“relative address is more than #ffff tetrabytes forward!”

Diese Fehler können z.B. bei der Adressierung von Speicher oder bei Branches / Jumps auftreten. Um diese Fehler verstehen zu können, muss man sich etwas detaillierter mit der Funktionsweise von MMIX und den einzelnen Befehlen auseinandersetzen.

AUFBAU EINES PROGRAMMS

Ein MMIX-Programm sollte grundsätzlich so aufgebaut sein:

1	LOC Data_Segment	lege alle Daten im Speicher ab der Adresse #2000000000000000 ab
2	GREG @	erzeuge ein globales Register mit der Speicheradresse dieser Zeile
3	...	
4	*DATEN*	lege Daten an
5	...	
6	LOC #100	Code wird ab Speicheradresse #0000000000000100 abgelegt
7	Main BEFEHL	Hier wird das Programm gestartet
8	...	
9	*BEFEHLE*	
10	...	
11	Label BEFEHLX	Ein Label, d.h. die Speicheradresse von BEFEHLX kann über Label angesprochen werden
12	...	
13	TRAP 0,Halt,0	Ende des Programms

ERKLÄRUNG ZU WICHTIGEN BEFEHLEN

LOC: Die Pseudo-Anweisung LOC XYZ veranlässt MMIXAL (MMIX Assembly Language) die folgenden Anweisungen an der Speicheradresse XYZ abzulegen.
Data_Segment ist eine Konstante von MMIXAL und hat den Wert #20.. (noch 14 Nullen).
Durch die Verwendung von LOC in den Zeilen 1 und 6 erhält man das folgende Speicherbild:

ab Beginn der Adresse	Inhalt des Speichers
#00..000	nichts
#00..100	Code / Befehle
#20..000	Daten
#FF..FFF	Ende des Speichers

@: Das @ steht in MMIXAL immer für die aktuelle Position im Speicher.

GREG XYZ: Die Pseudo-Anweisung legt ein globales Register an, in das der WERT XYZ geschrieben wird. Die Zeile „ GREG @“ legt also ein globales Register an, in das die aktuelle Speicherposition geschrieben wird. Globale Register sind Register

	wie $255, über die z.B. mit dem Betriebssystem Parameter für die Ein- oder Ausgabe ausgetauscht werden. Eine weitere Anwendung von globalen Registern wird weiter unten erklärt.
JMP label	Der Befehl JUMP (JMP) veranlasst den Rechner die Ausführung von Befehlen an der Speicheradresse label fortzuführen. Ein Label ist wie in Zeile 11 eine einfache Markierung.
BX $X,label	Branch-Befehle beginnen immer mit einem B. Über diese kann man den Ablauf des Programms beeinflussen und so genannte konditionale Sprünge durchführen. Branches werden also nur durchgeführt, wenn eine bestimmte Bedingung erfüllt ist. Wohin das Programm springen soll, wird durch die Angabe einer Speichermarke, einem label, angegeben. Ob ein Sprung durchgeführt wird oder nicht hängt dabei von der Befehlsart und dem Inhalt des Registers X ab. Beispiele für Branch-Befehle sind u.a. BZ und BP. BZ steht für „Branch if zero“. BZ $1,label bedeutet daher: „Springe zu label, wenn der Inhalt des Registers 1 gleich 0 ist.
GETA $X,label	Lädt die Adresse der Speichermarke label in das Register X.
LDA $X,label	Lädt die Adresse der Speichermarke label in das Register X unter der Verwendung eines globalen Registers, das von MMIXAL eingesetzt wird.
LDX[U] $I,$J,$K|K	Über die LOAD-Befehle (beginnend mit LD) können Daten aus dem Speicher gelesen werden. Es gibt dabei wieder verschiedene Varianten: LDB, LDW, LDT, LDO, LDBU etc. Der Aufbau dieser ist immer gleich. Beginnend mit dem LD beschreibt das nächste Zeichen, wie groß der Speicherbereich ist, der geladen werden soll. Ein B steht für Byte, ein W für ein Wyde, etc. Das letzte optionale U gibt darüber hinaus noch an, ob das Datum an der Speicheradresse als „unsigned“ angesehen werden soll, also nicht vorzeichenbehaftet ist. Die Parameter von LDX[U] sind in der Regel drei Register. In das Register I soll der Wert geschrieben werden. Die Summe der Register J und K geben die Speicheradresse an, aus der die Daten gelesen werden sollen. Als dritter Parameter kann auch eine Konstante geschrieben werden.

BEFEHLE ZUR ADRESSIERUNG IN MMIX

Bei den Befehlen für Sprünge / Branches, das Laden von Adressen und dem Laden von Daten muss immer eine Speicheradresse angegeben werden. Eine „falsche“ Adressierung kann zu den oben angesprochenen Fehlern führen. Um ein Verständnis dieser Fehlermeldungen und deren Lösungen zu schaffen, muss der Aufbau der einzelnen Befehle verstanden werden.

Ein Befehl besteht aus 4 Bytes und ist immer wie folgt aufgebaut:

Byte			
0	1	2	3
OP-Code	Argument X	Argument Y	Argument Z
OP-Code	Argument X	Argument YZ	
OP-Code	Argument XYZ		

Der OP-Code beschreibt den Befehl, der auszuführen ist. Wie man der Tabelle entnehmen kann, können maximal 3 Argumente übergeben werden. Diese sind jeweils ein Byte groß. Werden ein oder zwei Argumente angegeben, stehen für die Argument(e) 1&2 oder 3 Bytes zur Verfügung.

Art des Speicherzugriffs

Es wird zwischen relativer und absoluter Adressierung unterschieden. Bei der relativen Adressierung wird die Adresse durch eine Verschiebung in Bezug auf die aktuelle Adresse beschrieben. Im Gegensatz dazu wird bei der absoluten Adressierung eine absolute Speichadresse zur Adressierung benutzt.

Beispiel relative Adressierung

Speicheradresse	Code
	LOC #100
#100	Main SET foo,100
	…
#200	Loop SUB foo,foo,1
	…
#300	BZ foo,loop

Wird der Branch genommen, so wird der Programcounter auf „@ - #100“ gesetzt.

Es wird also auf die aktuelle Position Bezug genommen und eine Verschiebung von -100 festgelegt.

Beispiel absolute Adressierung

Speicheradresse	Code
	LOC Data_Segment
	GREG @
#20..00	msg BYTE “Hallo!”,0
	LOC #100
#100	Main SET foo,100
	…
#200	……….LDA $1,msg

Hier wird die Speicheradresse als absoluter Wert #20..00 nach $1 geladen.

(Wie das genau funktioniert wird weiter unten beschrieben!)

WIE ADRESSIERT MAN RICHTIG?

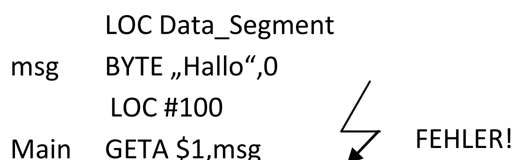
Mittels der relativen Adressierung kann nicht der gesamte Speicher adressiert werden. Dies ist auf den Aufbau der Befehle zurückzuführen. So kann den Branch-Befehlen und GETA als Verschiebung zur derzeitigen Adresse nur ein 16 Bit Wert übergeben werden. Dies entspricht einer maximalen Verschiebung um #FFFF = 65535 viele Zeilen Code à 4 Byte (jeder Befehl hat 4 Byte).

	Byte			
Befehl	0	1	2	3
GETA $X,label	OP-Code von GETA	Angabe des Registers	Angabe der Verschiebung zur aktuellen Speicherposition	
BZ $X,label	OP-Code von BZ	Angabe des Registers	Angabe der Verschiebung zur aktuellen Speicherposition	

Der Befehl JMP hingegegen hat nur einen Parameter und kann somit 3 Byte = #FFFFFF = 16777215 viele Codezeilen entfernt springen.

Erhält man die eingangs erwähnte Fehlermeldung "relative address is more than #ffff tetrabytes forward!" bedeutet dies also, dass die Adressierung mittels 2 Byte nicht dargestellt werden kann.

In dem folgenden – nicht funktionstüchtigen Beispiel wird dies gezeigt:

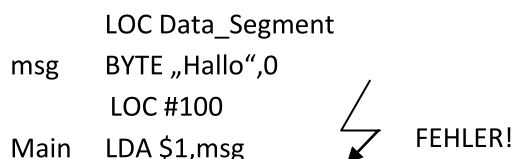


```
        LOC Data_Segment
msg     BYTE „Hallo",0
        LOC #100
Main    GETA $1,msg
```

FEHLER!

Der Fehler ist hier offensichtlich. An der Stelle #100 im Speicher wird die erste Zeile Code abgelegt. Die Daten wurden davor an der Stelle #2000000000000000 abgelegt. Um in der ersten Zeile die msg adressieren zu können, müsste also eine relative Adresse eingesetzt werden, welche größer als der mittels 2 Byte repräsentierbare Wert wäre. MMIXAL kann diesen Code nicht übersetzen.

Um weiter entfernte Speicherbereiche adressieren zu können, was besonders für das Laden und Speichern von Daten von Bedeutung ist, wird deshalb auf absolute Adressierung zurückgegriffen.



```
        LOC Data_Segment
msg     BYTE „Hallo",0
        LOC #100
Main    LDA $1,msg
```

FEHLER!

Auch dieser Code führt jedoch zu einem Fehler. Die Funktion LDA erhält nämlich eigentlich 3 Argumente:

	Byte			
Befehl	0	1	2	3
LDA $X,$Y,$Z|Z	OP-Code von LDA	Angabe des Zielregisters für das Ergebnis	Angabe eines Registers	Angabe eines Registers oder einer Konstante

Gemäß dieser eigentlichen Variante des LDA-Befehls wird die Adresse aus der Summe des Inhalts von $Y und $Z berechnet. Ein Beispiel verdeutlicht diese Vorgehensweise:

```
        LOC Data_Segment
gReg    GREG @
PI      TETRA 31415926
msg     BYTE "Hallo",0
         LOC #100
Main    LDA $1,gReg,4
```

Wie oben beschrieben wird durch GREG @ in ein globales Register der Wert der aktuellen Speicherpostion – in diesem Falle #2000000000000000 – geschrieben. Durch das vorgestellte gReg können wir auch im weiteren Programmverlauf das globale Register benennen. In der ersten Zeile des Codes laden wir nun die Adresse von msg in den Speicher: In gReg steht die sogenannte Basisadresse #2000000000000000. Da vor der msg noch ein Tetra abgelegt wurde, addieren wir zur Basisadresse 4 um auf den richtigen Speicherplatz zuzugreifen. Fertig!

Die Verwendung von Registern mit 64 Bit erlaubt daher eine vollständige Adressierung aller 2^{64} Speicherzellen.

Zugegebenermaßen: Diese Technik ist mehr als hässlich und würde die Verwendung von MMIXAL sehr unschön gestalten (siehe Bild).

Die Lösung in MMIXAL wurde oben schon kurz vorgestellt: Anstatt die Adresse „per Hand" zu errechnen, übergeben wir nur das Label:

```
        LOC Data_Segment
        GREG @
PI      TETRA 31415926
msg     BYTE "Hallo",0
         LOC #100
Main    LDA $1,msg
```

Wie ist das möglich? MMIXAL nimmt uns die Arbeit ab und schreibt unseren LDA Befehl bei dem Compilieren in die obere Variante um. MMIXAL sucht eigenständig das richtige globale Register, was wir trotzdem per Hand anlegen müssen, heraus und berechnet die richtige Verschiebung zu diesem Register. Für uns als Programmierer ist dieser Vorgang transparent, und wir merken uns nur folgenden (Pseudo-)Befehl:

	Byte			
Befehl	0	1	2	3
LDA $X,label	OP-Code von LDA	Angabe des Zielregisters für das Ergebnis	Angabe eines Labels, welches im „Umfeld" der Erzeugung eines globalen Registers liegt	

Findet MMIXAL kein globales Register, mit einer Basisadresse, welche benutzt werden kann, wird die Fehlermeldung “no base address is close enough to the address A!” ausgegeben.

Mittels der gleichen Technik können nun auch Daten aus dem Speicher geladen werden:

```
        LOC Data_Segment      Lege Segment für Daten an
        GREG @                Erzeuge globales Register mit aktueller Speicherposition
baseAdr         IS $1
char            IS $0
PI      TETRA 31415926
msg     BYTE “Hallo”,0        Erzeuge String
        LOC #100              Beginne mit Code
Main    LDA baseAdr,msg       Lade die Adresse von msg nach $0
        LDB char,baseAdr,4    Lade das fünfte Zeichen der Zeichenkette (das o) in Register $1
```

Anstatt bei LDB die Konstante 4 zu verwenden, kann natürlich auch wieder ein Register benutzt werden.