Übersicht

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

- Wiederholung: ein einfaches MMIX-Programm
- Speicherorganisation, Speicherzugriff
- Zahlen und Arithmetik
- Zeichenketten und Ein-/Ausgabe
- Kontrollstrukturen
- Unterprogramme

Quelle: Kapitel 3, 4 und 5 aus Anlauff, Böttcher, Ruckert: Das MMIX-Buch. Springer, 2002

Ein einfaches MMIX-Programm

Programm „arithmetik.mms“

```
        LOC     #100
Main    SET     $1,4        Anfangswerte setzen
        SET     $2,2
        SET     $3,40
        MUL     $4,$2,$3    Produkt von $2 und $3 nach $4
        ADD     $5,$4,$1    Summe von $4 und $1 nach $5
        SUB     $6,$1,$2    Differenz von $1 und $2 nach $6
        DIV     $7,$5,$6    Quotient von $5 und $6 nach $7
        TRAP    0,Halt,0
```

$$\frac{\$2 * \$3 + \$1}{\$1 - \$2} = \$7$$

MMIX Assembler Listing

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

Der MMIX-Assembler kann ein Assmbler Listing ausgeben:
`mmixal –l arithmetik.txt arithmetik.mms`
erzeugt die Datei `arithmetik.txt`:

```
                        LOC    #100
 ...100: e3010004  Main SET    $1,4       Anfangswerte setzen
 ...104: e3020002       SET    $2,2
 ...108: e3030028       SET    $3,40
 ...10c: 18040203       MUL    $4,$2,$3   Produkt von $2 und $3 nach $4
 ...110: 20050401       ADD    $5,$4,$1   Summe von $4 und $1 nach $5
 ...114: 24060102       SUB    $6,$1,$2   Differenz von $1 und $2 nach $6
 ...118: 1c070506       DIV    $7,$5,$6   Quotient von $5 und $6 nach $7
 ...11c: 00000000       TRAP   0,Halt,0

Symbol table:
 Main = #0000000000000100 (1)
```

MMIX Ablauf

Durch die Angabe von `-tn` kann man den MMIX-Simulator im Trace-Modus verwenden. Angabe von n bedeutet: jede n-te Verwendung eines Befehls wird ausgegeben.

Beispiel: `mmix -t1 arithmetik.mmo`

```
  1. 0000000000000100: e3010004 (SETL) $1=l[1] = #4
  1. 0000000000000104: e3020002 (SETL) rL=3, $2=l[2] = #2
  1. 0000000000000108: e3030028 (SETL) rL=4, $3=l[3] = #28
  1. 000000000000010c: 18040203 (MUL) rL=5, $4=l[4] = 2 * 40 = 80
  1. 0000000000000110: 20050401 (ADD) rL=6, $5=l[5] = 80 + 4 = 84
  1. 0000000000000114: 24060102 (SUB) rL=7, $6=l[6] = 4 - 2 = 2
  1. 0000000000000118: 1c070506 (DIV) rL=8, $7=l[7] = 84 / 2 = 42, rR=0
  1. 000000000000011c: 00000000 (TRAP) Halt(0)

8 instructions, 0 mems, 80 oops; 0 good guesses, 0 bad
 (halted at location #000000000000011c)
```

Speicherorganisation

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

Im Programm auf der vorherigen Folie wird der Speicher nur dazu genutzt, um das Programm abzulegen.

- Der MMIX-Simulator holt sich die Objektdatei in den Speicher und führt sie beginnend beim ersten Befehl aus (muss durch `Main` markiert sein)
- Daten wurden mittels SET direkt in Register geschrieben
- Alternative: Verwendung von Direktoperanden, z.B. `MUL $4,$2,40`
- Direktoperanden sind durch das Format des Befehlswortes eingeschränkt.
- Besser: Programme und Daten logisch trennen!

Trennung von Daten und Programmcode

- Programme werden leichter wartbar
- Zugriff auf Daten über ihre (symbolischen) Adressen
- Speicherorganisation wird über Pseudobefehle durch den Assembler gemacht
- MMIX unterscheidet verschiedenen Bereiche im Hauptspeicher, die Segmente genannt werden

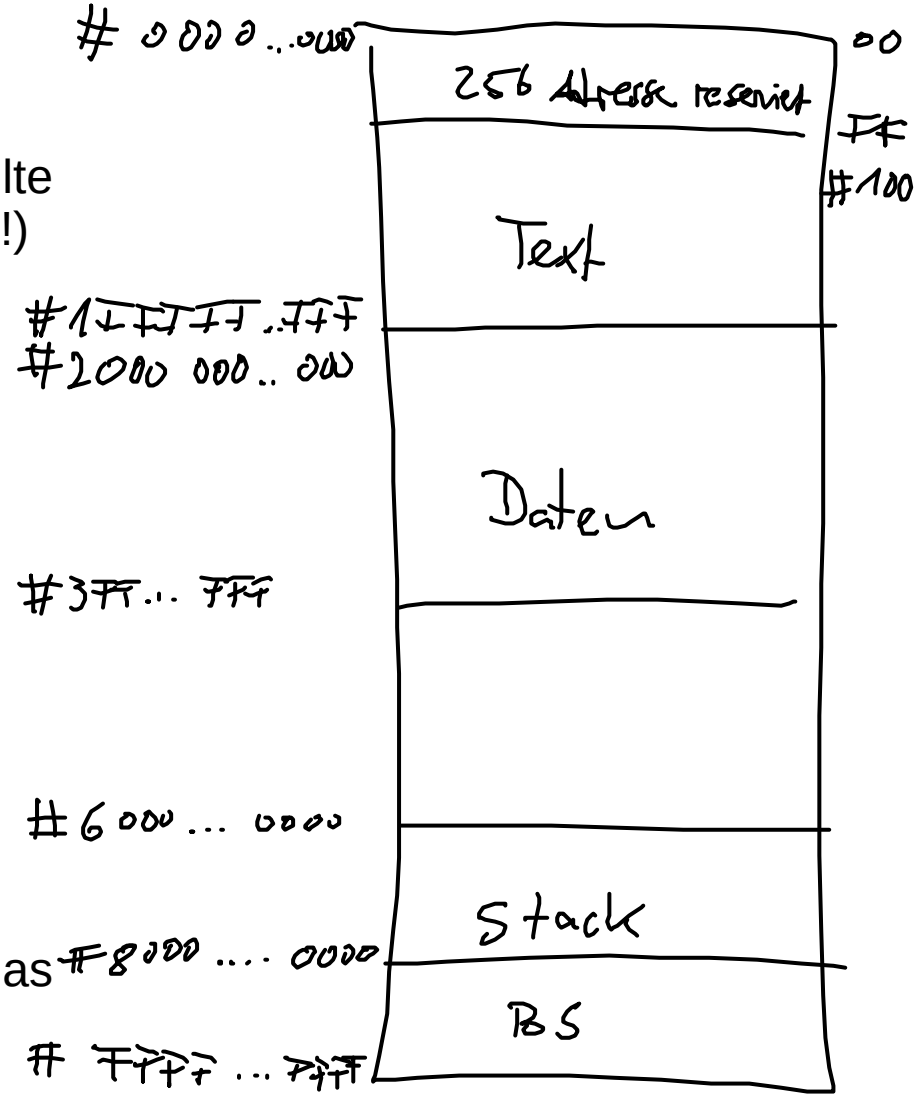
Speichersegmente

Textsegment
von #0 bis #1FFF FFFF FFFF FFFF
(#0 bis #FF reserviert! Textsegment sollte als READ-ONLY angenommen werden!)

Datensegment
von #2000 0000 0000 0000
bis #3FFF FFFF FFFF FFFF

Stacksegment
von #6000 0000 0000 0000
bis #7FFF FFFF FFFF FFFF

Für uns zunächst uninteressant:
Poolsegment (zwischen Datensegment und Stacksegment) sowie Bereich für das Betriebssystem (hinter dem Stacksegment)



Belegen von Speicher

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

Beispielprogrammfragment mit Pseudobefehlen zur Speicherorganisation:

```
        LOC   #100     Start bei Adresse #100
X1      OCTA  4        Schreibt den Wert 4 an Adresse #100
X2      OCTA  2        Danach Wert 2 an Adresse #108
X3      OCTA  40       Schliesslich Wert 40 an Adresse #116
```

Zahlenwerte werden immer entsprechend ihrem Typ ausgerichtet

- Beispiel: OCTA wird immer ab der nächsten durch 8 teilbaren Adresse geschrieben
- Holen in Register durch LDO (Load Octa) zur Trennung von Daten und Programm

Sauberere Trennung: Ablegen aller Daten im Datensegment

- durch `LOC Data_Segment`
- anschliessen Programm ab `LOC #100` ablegen

Programm mit Trennung Daten/Text (arith-ldo.mms)

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

```
        LOC     Data_Segment
        GREG    @               ???
X1      OCTA    4               Daten im Datensegment ablegen
X2      OCTA    2
X3      OCTA    40
        LOC     #100
Main    LDO     $1,X1           Anfangswerte aus dem Speicher lesen
        LDO     $2,X2
        LDO     $3,X3
        MUL     $4,$2,$3    Produkt von $2 und $3 nach $4
        ADD     $5,$4,$1    Summe von $4 und $1 nach $5
        SUB     $6,$1,$2    Differenz von $1 und $2 nach $6
        DIV     $7,$5,$6    Quotient von $5 und $6 nach $7
        TRAP    0,Halt,0
```

LDO und LDOI

Ausschnitte aus dem Trace von `arith-ldo.mms`:

```
1. 0000000000000100: 8d01fe00 (LDOI) $1=l[1] =
                                M8[#2000000000000000] = 4
1. 0000000000000104: 8d02fe08 (LDOI) rL=3, $2=l[2] =
                                M8[#2000000000000000+8] = 2
1. 0000000000000108: 8d03fe10 (LDOI) rL=4, $3=l[3] =
                                M8[#2000000000000000+16] = 40
1. 000000000000010c: 18040203 (MUL) rL=5, $4=l[4] = 2 * 40 = 80
...
```

LDO wird übersetzt in ein LDOI (Load Octa Immediate)

- Es gibt nur das Befehlsformat LDO $X,$Y,$Z bzw LDO $X,$Y,Z
- Wie soll eine 64-Bit Adresse dort hineincodiert werden?
- Lösung: Assembler übersetzt den Ladebefehl LDO $1,X1 in LDO $1,$B,Z für ein „verstecktes“ Basisregister $B und einen Offset Z relativ zu $B
- Basisregister wird durch GREG @ angelegt

Woher kommt das Basisregister?

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

Ohne das GREG @ hätte man folgende Fehlermeldung:

```
C:\...\RS>mmix arith-ldo.mms
"arith-ldo.mms", line 6: no base address is close enough to the address A!
"arith-ldo.mms", line 7: no base address is close enough to the address A!
"arith-ldo.mms", line 8: no base address is close enough to the address A!
(3 errors were found.)
```

GREG @ legt implizit ein neues Basisregister für alle Daten ab der aktuellen Adresse (@)

- Im Programm arith-ldo.mms hat @ den Wert von Data_Segment
- Assemblierter Befehl LDO $2,X2 lautet: `8d02fe08`
  - 8d = Opcode für LDOI
  - 02 = Zielregister $2
  - FE = Bezug auf neues Basisregister (hier $254)
  - 08 = Offset von X2 zur Adresse aus dem Basisregister $254
- Mehr zu globalen Registern später

Vorzeichenbehandlung beim Laden

Befehle wie LDO, LDW fassen Daten immer als vorzeichenbehaftete Zahlen auf

- Bei den Ladebefehlen wird eine geladene Einheit immer als vorzeichenbehaftete Zahl im Zweierkomplement betrachtet
- Verbleibende freie Bits werden gemäß Vorzeichen aufgefüllt
- Automatische Ausrichtung der geladenen Adresse auf den Datentyp

Beispiel: Sei die folgende Speicherbelegung gegeben…

M[100]	M[101]	M[102]	M[103]	M[104]	M[105]	M[106]	M[107]
01	23	45	67	89	AB	CD	EF

Sei $2=#100 und $3=2

Befehl:	**Wirkung auf Register $1:**
`LDB $1,$2,$3`	#0000 0000 0000 0045
`LDW $1,$2,$3`	#0000 0000 0000 4567
`LDT $1,$2,$3`	#0000 0000 0123 4567
`LDO $1,$2,$3`	#0123 4567 89AB CDEF

Vorzeichenbehandlung beim Laden

Sei wieder die folgende Speicherbelegung gegeben…

M[100]	M[101]	M[102]	M[103]	M[104]	M[105]	M[106]	M[107]
01	23	45	67	89	AB	CD	EF

Sei nun $2=#100 und $3=5, es wird also Adresse #105 angesprochen:

Befehl:	**Wirkung auf Register $1:**
`LDB $1,$2,$3`	#FFFF FFFF FFFF FF AB
`LDW $1,$2,$3`	#FFFF FFFF FFFF 89 AB
`LDT $1,$2,$3`	#FFFF FFFF 89AB CD EF
`LDO $1,$2,$3`	#0123 4567 89AB CD EF

Vorzeichen wird beachtet. Falls dies nicht gewünscht wird: „unsigned“-Varianten (z.B. LDOU) benutzen.

Vorzeichenbehandlung bei Speichern

Beim Speichern kann ein Überlauf eintreten:

- Falls die Zahl, die abgespeichert werden soll, zu groß ist für das Datenformat, in dem gespeichert wird
- z.B. STB kann nur Zahlen im Bereich -128 bis +127 ohne Überlauf abspeichern

Beispiel:

M[100]	M[101]	M[102]	M[103]	M[104]	M[105]	M[106]	M[107]
01	23	45	67	89	AB	CD	EF

Sei $1 = -65536 = #FFFF FFFF FFFF 0000

$2 = #100 $3 = 2

Adresse #102 = ... 0000 0010
bei T/b .. 0000 0000

Befehl	**Wirkung auf Speicherstelle M_8[100], Überlauf?**	
STB $1,$2,$3	#0123 00 67 89 ABCD EF	(Überlauf)
STW $1,$2,$3	#0123 0000 89ABCD EF	(Überlauf)
STT $1,$2,$3	#FFFF 0000 89ABCD EF	(kein Überlauf)
STO $1,$2,$3	#FFFF FFFF FFFF 0000	

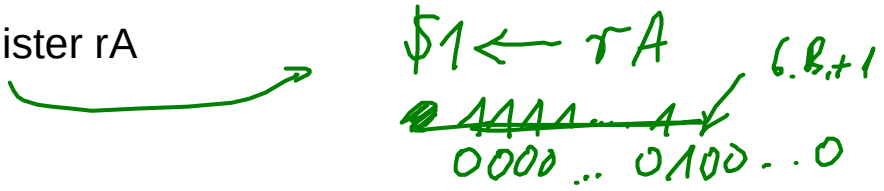
- Ein Überlauf kann auch bei positiven Zahlen auftreten!
- Auch zum Speichern gibt es „unsigned“-Varianten.

Arithmetik

Ganzzahlarithmetik wird beim MMIX mit 64-Bit-Zahlen im Zweierkomplement durchgeführt

Addition (ADD):

- Überlauf, denn die Zahl ausserhalb des darstellbaren Zahlenbereichs liegt, also $N < -2^{63}$ oder $N \geq +2^{63}$
- Überlauf angezeigt durch Bit 6 im Spezialregister rA



Subtraktion (SUB):

- Überlauf möglich, analog zur Addition

Multiplikation (MUL):

- Im Zielregister werden die 64 niederwertigsten Bit des Ergebnisses abgespeichert
- Überlauf wie bei Addition/Subtraktion angezeigt

Division (DIV):

- Ganzzahliger Anteil des Ergebnisses steht im Zielregister
- Rest der Division steht im Spezialregister rR
- Divisionsrest kann mit `GET $X,rR` in ein allgemeines Register geholt werden
- Vorsicht bei Division durch Null

Es gibt auch „unsigned“-Varianten (ganze Zahlen zwischen 0 und 2^{64}-1.

Zeichenketten

Definition Zeichenkette:

- Folge von Zeichen, die durch ein Nullzeichen abgeschlossen ist (ASCII-Code 0)
- Das Nullzeichen ist kein druckbares Zeichen, markiert nur das Ende der Zeichenkette (Konvention wie in C/C++)

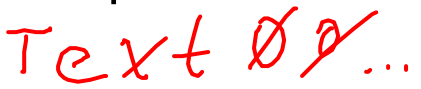
Zeichen werden im ASCII-Code im Speicher angelegt

- Festgelegte Zuordnung von darstellbaren Zeichen zu Zahlenwerten
- Ein Zeichen = Ein Byte

Beispiel:

```
OCTA #5465 7874 0000 0000
```

Im Speicher:



Alternative Formen:

- `BYTE ´T´,´e´,´x´,´t´,0`
- `BYTE „Text“,0`

Nichtdruckbare Zeichen müssen mit ihrem ASCII-Codewert angegeben werden

Beispiel: `BYTE „Text“,#A,0`

ASCII-Codes 0-63

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

0	00	NUL
1	01	SCH
2	02	STX
3	03	ETX
4	04	EOT
5	05	ENQ
6	06	ACK
7	07	BEL
8	08	BS
9	09	HT
10	0A	LF
11	0B	VT
12	0C	FF
13	0D	CR
14	0E	SO
15	0F	S1

16	10	DLE
17	11	DC1
18	12	DC2
19	13	DC3
20	14	DC4
21	15	NAK
22	16	SYN
23	17	ETB
24	18	CAN
25	19	EM
26	1A	SUB
27	1B	ESC
28	1C	FS
29	1D	GS
30	1E	RS
31	1F	US

32	20	SP
33	21	!
34	22	~
35	23	
36	44	$
37	25	%
38	26	&
39	27	`
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/

48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

ASCII-Codes 64-127

RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE RWTH

64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O

80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

96	60	´
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o

112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	DEL

Ein-/Ausgabe

RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE RWTH

Einlesen von Daten von der Tastatur und Ausgabe auf dem Bildschirm sind komplexe, Geräteabhängige Vorgänge

- Werden durch Gerätetreiber implementiert, die die Daten aus bestimmten Speicherbereichen lesen oder dorthin schreiben
- In höheren Programmiersprachen sind diese Funktionen in Bibliotheken gekapselt, die entsprechende Betriebssystemfunktionen aufrufen
- Im MMIX-Simulator ist zwar kein Betriebssystem implementiert, es werden aber ein paar nützliche Basisfunktionen simuliert

Realisierte Basisfunktionen:

- Programmhalt: `Halt`
- Einlesen einer Zeichenkette: `Fgets` („get string from file“)
- Ausgabe einer zeichenkette: `Fputs` („put string to file“)

MMIX hält diese Funktionen unter bestimmten Nummern bereit

- Nummern werden über Sprungtabellen im Betriebssystem in Adressen umgewandelt
- Nach Ausführung der Funktion: Rückkehr ins aktuelle Assemblerprogramm „als ob nichts gewesen wäre“ (Unterprogrammtechnik, siehe später)

TRAP

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

Aufruf der Basisfunktionen des Betriebssystems durch TRAP:

- erstes Argument ist immer 0
- zweites Argument: Nummer der aufzurufenden Funktion (oder ihr Name)
- drittes Argument Z und globales Register $255 kann zur Übergabe von Parametern verwendet werden

Beispiel: `TRAP 0, Halt,0`

Ausgabe von Zeichenketten mittels TRAP:

- als drittes Argument muss die Nummer des Ausgabekanals übergeben werden
- Standard-Ausgabe `StdOut` ist der Bildschirm
- Dateien können mit weiteren Kanälen verbunden werden
- Implizite Annahme: Im Register $255 steht die Anfangsadresse einer Zeichenkette

Beispiel für einfache Ausgabe

```
        LOC   Data_Segment
        GREG  @
String  BYTE  "Text",#A,0      auszugebende Zeichenkette
        LOC   #100
Main    LDA   $255,String      Adresse des Strings in $255
        TRAP  0,Fputs,StdOut   Ausgabe der Zeichenkette
        TRAP  0,Halt,0
```

Bemerkungen:

- LDA benötigt `GREG @`
- Es wird alles bis zur ersten Null ausgegeben. Vorsicht bei nicht-terminierten Zeichenketten!

Einlesen von Zeichenketten

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

Ähnlich wie bei der Ausgabe wird ein Eingabekanal gefordert:

- Normalerweise `StdIn` (Standardeingabe) = Tastatur
- Man muss zusätzlich im Programm Speicherplatz (einen Puffer) reservieren für die eingelesenen Zeichen

Beispiel: Puffer für 80 Zeichen im Speicher

```
Buffer    BYTE    0           Pufferanfang
          LOC     Buffer+80   80 Byte bleiben unbelegt
```

Zahl 0 ist eigentlich unnütz (wird ja als erstes überschrieben), wird aber benötigt, um eine Marke zu vergeben.

Es muss zusätzlich neben der Adresse es Eingabepuffers seine Größe an Fgets übergeben werden

- Wir haben aber nur ein anderes Register $255?!
- Lösung: Wir schreiben die Parameter als Octabytes hintereinander in den Speicher und setzen $255 auf die Adresse des ersten dieser beiden Werte

Beispielprogramm für die Eingabe

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

```
          LOC   Data_Segment
          GREG  @
BufSize   IS    80
Buffer    BYTE  0                 Puffer anlegen
          LOC   Buffer+BufSize
Arg       OCTA  Buffer            Anfangsadresse
          OCTA  BufSize           Größe des Puffers

          LOC   #100
Main      LDA   $255,Arg          $255 <- #2000 0000 0000 0050
          TRAP  0,Fgets,StdIn     Einlesen
          TRAP  0,Halt,0
```

Unbedingte Sprünge

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

Es gibt im MMIX zwei Befehle für bedingungslose Sprünge:

Befehl 1: `JMP Marke`

- JMP benutzt eine relative Adresse, eignet sich also gut für Sprünge in Schleifen
- Sowohl Sprung vorwärts als auch rückwärts erlaubt
- Maximale Sprunggröße durch Länge des Argumentes begrenzt (24 Bit)
- Über @ kann man auch ohne Marken arbeiten
  - @ steht für die momentane Adresse des Programms
  - Beispiel: `JMP @+2*4` Springe zum übernächsten Befehl

Befehl 2: `GO`

- Zieladresse wird in derselben Weise definiert wie bei Befehlen zum Laden und Speichern (mit Basisregister und Offset): absolute Adressierung, eignet sich also für „weite“ Sprünge
- GO merkt sich im als erstes angegebenen Register die Adresse des Befehls, der als nächstes zur Ausführung gekommen wäre (diese Eigenschaft wird später noch interessant werden)
- Beispiel:

```
        GREG @
Start   ..      Sprungziel
        ..
        GO   $0,Start
```

Bedingte Sprünge

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

Sprünge können von einer zu testenden Bedingung abhängig gemacht werden

Beispiel:

```
        BZ   $1,Fertig     Prüfe Register $1
        ..
Fertig  ..                 hier fortfahren falls Null
```

Wie beim JMP-Befehl wird die Marke hier vom Assembler eine relative Adressierung umgesetzt

- Stehen aber nur 16 Bit als Sprungweite zur Verfügung

Effizienz:

- Ein nicht gemachter Sprung ist schneller als ein gemachter Sprung (in Taktzyklen)
- Man kann dem Assembler angeben, ob in der Regel ein Sprung gemacht wird: „Probable Branch“-Varianten PBZ etc.

Beispiel: Euklid‘scher Algorithmus

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
RWTH

```
r       IS   $1       Berechnet den ggT von m und n
m       IS   $2
n       IS   $3

        LOC  #100
Main    SET  m,1228    Startwerte setzen
        SET  n,96
Start   DIV  m,m,n     m := m DIV n
        GET  r,rR      r := m MOD n (aus Spezialregister)
        BZ   r,Fertig
        SET  m,n       m := n
        SET  n,r       n := r
        JMP  Start

Fertig  TRAP 0,Halt,0  Ergebnis steht in $3
```

Man kann das auch mit nur einem Sprungbefehl machen!