10. Die Maschinenbefehle des M 68000

10.1 MOVE-Befehle

10.2 Arithmetische Befehle

10.3 Sprungbefehle

10.4 Programmbeispiele

10.1 MOVE-Befehle

MOVE

Übertrage Daten von der Quelle zum Ziel

Assembler-Syntax: MOVE.X <ea>, <ea>

Operation: <Quelle> -> <Ziel>

Welche Flags werden verändert?

X Nicht berührt

N Auf Eins gesetzt, falls der übertragene Datenwert negativ ist, sonst auf Null gesetzt

Z Auf Eins gesetzt, falls der übertragene Datenwert = 0 ist, sonst auf Null gesetzt.

V Immer auf Null gesetzt.

C immer auf Null gesetzt.

Operandengröße

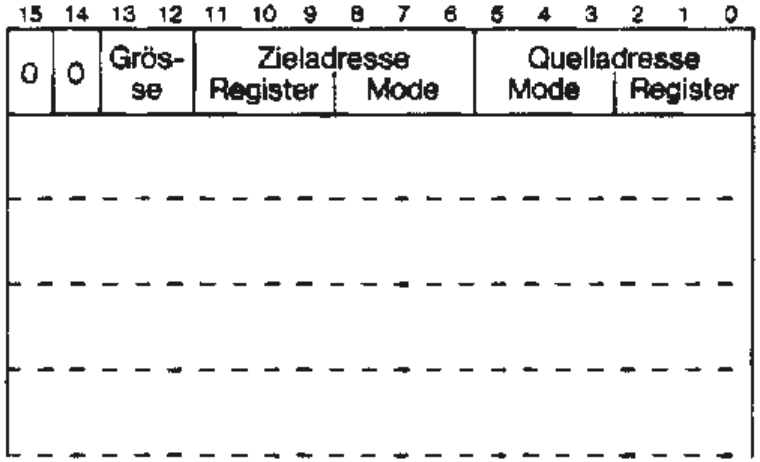
Mit dem MOVE-Befehl können Byte-, Wort- und Langwortdaten vom Quelloperanden zum Zieloperanden übertragen werden. Es kann also am Befehl .B, .W oder .L an der Stelle von .X stehen.

MOVE

Befehlsbeschreibung

Der MOVE-Befehl dient zum Übertragen von Daten vom Quell- zum Zieloperanden. Er ist der universellste und sicherlich meistverwendete Befehl im M 68000. Im Befehlssatz des M 68000 gibt es keinen Befehl LDA oder STA zum Laden oder Ablegen von Registern. Der MOVE-Befehl erfüllt auch diese Aufgaben. Er überträgt Daten von überall her und überall hin. Es können dabei Byte-, Wort- oder Langwortdaten übertragen werden. Abhängig von den übertragenen Daten werden die Flags gesetzt.

Opcode und Erklärung



MOVEM (1)

MOVE MULTIPLE

Übertrage mehrere Register

Assembler-Syntax:

MOVEM.X Registerliste, < ea >

MOVEM.X < ea >, Registerliste

Operation:

Inhalte der Register in der Registerliste -> < ea >

< ea > -> Register in der Registerliste

Welche Flags werden verändert? Keine

Operandenlänge:

Mit dem MOVEM-Befehl kann der 16- oder 32 Bit-Inhalt einer Reihe von spezifischen Registern geholt oder abgelegt werden. Es ist also Wort- oder Langwort-verarbeitung möglich, wobei an der Stelle von .X am Befehl .W oder .L stehen kann.

MOVEM (2)

Befehlsbeschreibung

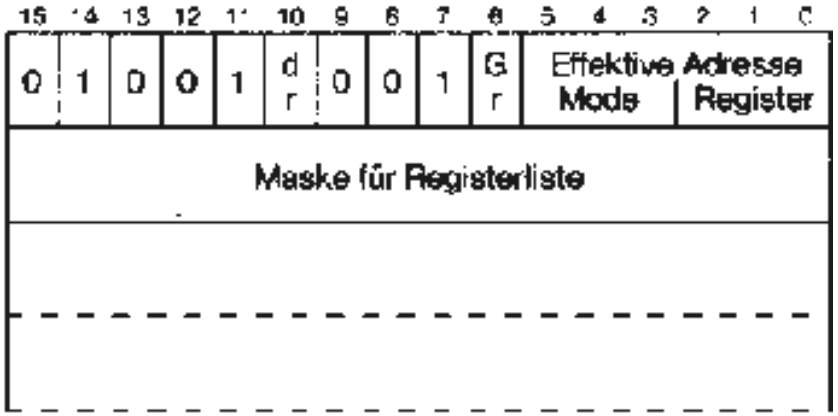
Mit dem MOVEM-Befehl werden bis zu 16 Register (Datenregister D0 – D7 und Adressregister A0 – A7) in oder aus dem Speicher übertragen. Man kann dabei einen Registerauswahl treffen. In der Assembler-Syntax wird dies mit der Registerliste ausgedrückt.

Ein Registerinhalt wird übertragen, wenn das entsprechende Bit in der Registermaske (siehe Opcode) gesetzt ist.

Außerdem kann man festlegen, ob Wort- oder Langwortverarbeitung durchgeführt wird. Bei einer Übertragung von Wortdaten in die spezifizierten Register wird der Quelloperand immer *vorzeichenrichtig* auf 32 Bit erweitert und dann in die 32 Bit des jeweiligen Registers geschrieben. Bei Übertragung von Wortdaten aus den Registern in den Speicher wird nur die untere Hälfte des jeweiligen Register übertragen.

MOVEM (3)

Opcode und Erklärung

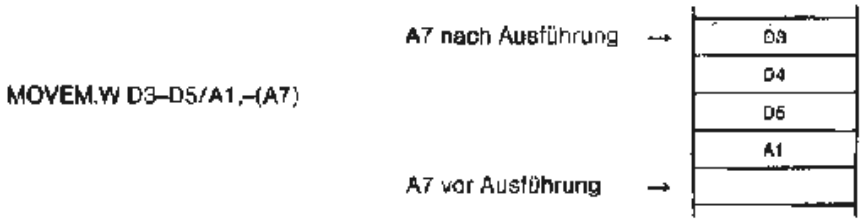


Beispiele zu MOVEM

Beispiel:

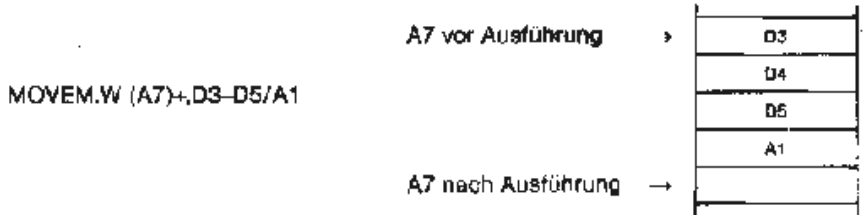
a) Ablegen der Inhalte einiger Register auf dem Stack

Ablagereihenfolge
A7 A6 A5 A4 A3 A2 A1 A0 D7 D6 D5 D4 D3 D2 D1 D0



b) Holen der Inhalte einiger Register vom Stack.

Holreihenfolge
D0 D1 D2 D3 D4 D5 D6 D7 A0 A1 A2 A3 A4 A5 A6 A7



10.2 Arithmetische Befehle

ADD

Binäre Addition

Assembler-Syntax:

ADD.X <ea>, Dn
ADD.X Dn, <ea>

Operation:

< Quelle > + < Ziel > -> < Ziel >

Welche Flags werden verändert:

X gesetzt wie das C-Bit.

N auf Eins gesetzt, falls das Ergebnis negativ ist, sonst auf 0 gesetzt.

Z auf Eins gesetzt, falls das Ergebnis = 0 ist, sonst auf 0 gesetzt.

V auf Eins gesetzt, falls ein Überlauf auftritt, sonst auf 0 gesetzt.

C auf Eins gesetzt, falls ein Überlauf auftritt, sonst auf 0 gesetzt.

Operandengröße:

Mit diesem Befehl können Byte-, Wort- und Langwort-daten binär addiert werden. .X steht also für .B, .W oder .L.

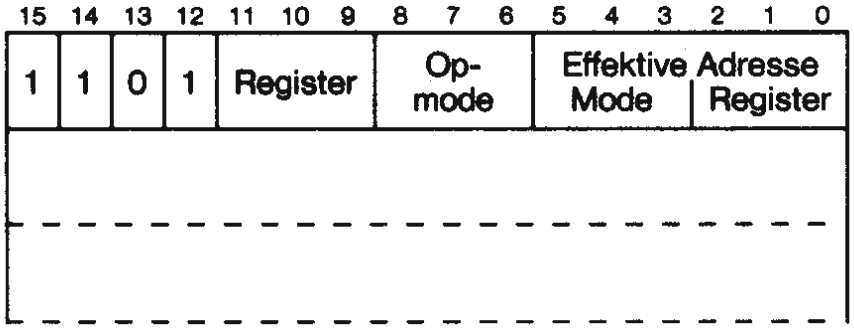
ADD

Befehlsbeschreibung

Der ADD-Befehl dient zur binären Addition des Quelloperanden zum Zieloperanden. Es wird dabei kein evtl. in einer vorher durchgeführten Addition aufgetretener Übertrag mit aufaddiert.

Will man Daten größer als 32 Bits addieren, muss man anschließend an einen 32-Bit-ADD-Befehl einen ADDX-Befehl anwenden, um einen evtl. Übertrag zu berücksichtigen. Für den ADD-Befehl muss entweder der Quell- oder der Zieloperand in einem Datenregister stehen.

Opcode und Erklärung



Beispiele zu ADD (1)

1. **Addition von Bytewerten**:
   In Register D1 steht. $xxxxxx53
   In Register D3 steht: $xxxxxx23

   ADD.B D1, D3 — Addition von D1+D3
   Ergebnis steht in D3:
   $xxxxxx76

   Flags im SR nach Ausführung der Operation:

X	N	Z	V	C
0	0	0	0	0

2. **Addition von Wortwerten**:
   In Register D1 steht: $xxxxxx1234
   In Speicherzelle $1000 steht: $5678

   ADD.W $1000, D1 — Addition der beiden Zahlenwerte. Ergebnis steht in D1:
   $68AC

   Flags im SR nach Ausführung der Operation:

X	N	Z	V	C
0	0	0	0	0

Beispiele zu ADD (2)

3. Addition von Langwortwerten:

In Register D1 steht: $82345678
In Speicherzelle $1000 ff. steht: $84510213

ADD.L $1000, D1 — Addition beider Zahlenwerte

D1: — Ergebnis steht in $0685588B

Flags im SR nach Ausführung der Operation:

```
X N Z V C
1 0 0 1 1
```


CMP (1)

Vergleiche zwei Operanden nach ihrer Größe und setze die Flags im SR

Assembler-Syntax

CMP.X <ea>, Dn

Operation

<Ziel> - <Quelle> (minus)→ **Flags** werden gemäß dem Ergebnis gesetzt:

X (nicht verändert)
N auf Eins gesetzt, falls das Ergebnis negativ ist, sonst auf Null gesetzt
Z auf Eins gesetzt, falls das Ergebnis = 0 ist, sonst auf Null gesetzt
V auf Eins gesetzt, falls ein Überlauf auftritt, sonst auf Null gesetzt.
C auf Eins gesetzt, falls ein „Borgen“ durchgeführt wird, sonst auf Null gesetzt.

Operandengröße

Mit dem CMP-Befehl können zwei Byte-, Wort- oder Langwortoperanden verglichen werden. Am Befehl CMP kann statt .X (wie üblich) .B, .W oder .L stehen.

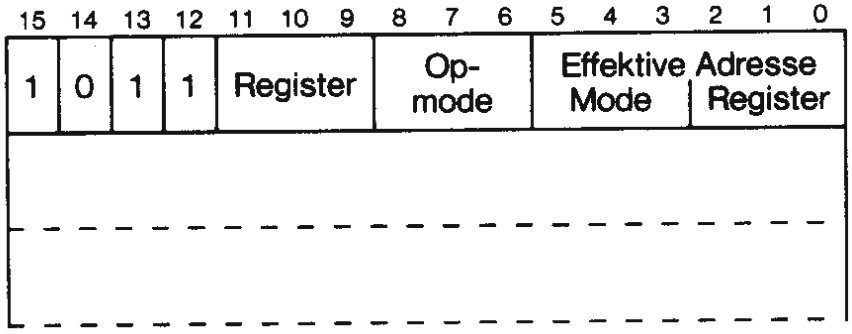
CMP (2)

Befehlsbeschreibung

Der CMP-Befehl dient zum Vergleichen der Größe zweier Operanden. Dieser Vergleich läuft folgendermaßen ab: Der Quelloperand wird vom Zieloperanden subtrahiert, ohne jedoch den Zieloperanden selbst zu verändern. In Abhängigkeit vom Ergebnis dieser Operation werden dann die Flags gesetzt.

Die Subtraktion läuft im M 68000 so ab, dass die ALU eine normale Subtraktionsoperation durchführt, das Ergebnis jedoch nicht an den Zieloperanden weitergibt.

Opcode und Erklärung



LEA (1)

Lade eine effektive Adresse in ein Adressregister

Assembler-Syntax

LEA <ea>, An

Operation:

Effektive Adresse wird in <An> geladen.

Welche Flags werden verändert? Keine

Operandenlänge:

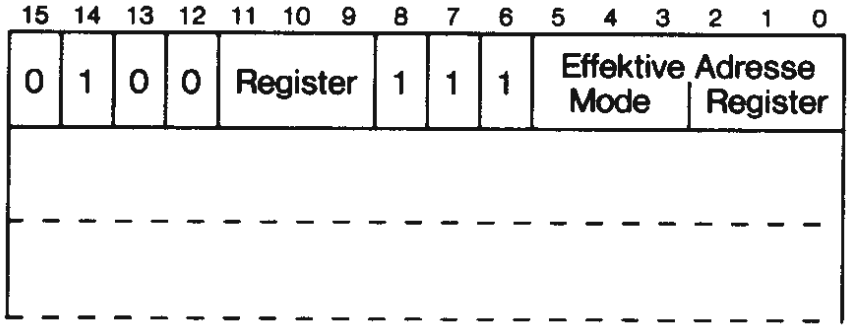
Nachdem beim LEA-Befehl eine Adresse in ein Adressregister geladen wird, arbeitet der Befehl nur mit 32 Bit langen Operanden. Es gibt kein .X am Befehl, da nur Langwortverarbeitung zugelassen ist.

LEA (2)

Befehlsbeschreibung

Der LEA-Befehl dient zum Laden einer effektiven Adresse in ein Adressregister. Bei der Durchführung des LEA-Befehls wird zunächst die effektive Adresse berechnet. Man stelle sich vor, die Adresse laute: 5(A0,S5.L). Diese eben berechnete Adresse wird dann in das 32bit-Zieladressregister geladen, wobei alle 32 Bits der Quelladresse in das Adressregister gehen. Der Unterschied zu einem MOVE-Befehl besteht darin, dass beim LEA-Befehl die **Adresse** des Quelloperanden in das Adressregister geladen wird, beim MOVE-Befehl dagegen der Inhalt des Quelloperanden.

Opcode und Erklärung



Beispiele zu LEA (1)

```
A0 =$00044A00        A0 =$12345678
D0 =$00008000
A3 = xxxxxxxx        A3 = xxxxxxxx
```

LEA 5(A0,D0), A3 LEA (A0), A3

```
  $00044A00
  $FFFF8000
+ $00000005
 $10003CA05
```

Ergebnis:

```
A0 = $00044A00       A0 = $12345678
D0 = $00008000       A3 = $12345678
A3 = $0003CA05
```

Beispiele zu LEA (2)

Berechnen einer effektiven Adresse und Ablegen in einem Adressregister vor Ausführen einer längeren Schleife:

```
        LEA $25(AO,D3), A5    Berechnen der eff. Adresse
LOOP    MOVE.W (A5), D5
        .......  (A5)         Jeweils Adressieren der
        .......  (A5)         Speicherzelle über
        .......  (A5)         Adressierungsart
                              "Adressregister indirekt"

        DBRA D5, LOOP
```

Anwendung

Will man die Adresse eines Operanden berechnen und festhalten, so ist der LEA-Befehl genau der richtige dafür. Eine sinnvolle Anwendung ist im Beispiel gezeigt. Man kann sich dadurch einiges an Programmlaufzeit und Schreibaufwand sparen. Außerdem wird das Programm übersichtlicher.

10.3 Sprungbefehle

Bcc

"branch on condition" - Bedingte Verzweigung

Assembler-Syntax
Bcc Marke

cc steht für eine Bedingung aus nachstehender Tabelle

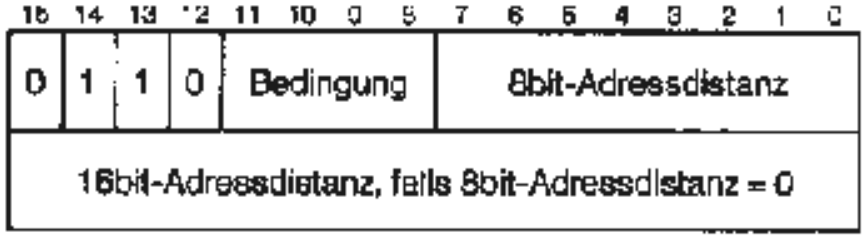
Operation
Falls Bedingung cc erfüllt, verzweige zur Marke. Falls Bedingung cc nicht erfüllt, mache mit dem nächsten Befehl weiter.

Welche Flags werden verändert? Keine.
Der Befehl nimmt lediglich Bezug auf die Flags.

Bcc (2)

Operandengröße

Hierbei handelt es sich nicht um eine Operandengröße im herkömmlichen Sinn, sondern um die Angabe, wie groß die Adressdistanz d werden kann. Die Adress-distanz kann 8 und 16 Bit groß sein (d.h. Werte von +127 bis -128 bzw. +32k bis -32k annehmen).



Bcc (3)

Befehlsbeschreibung

Der Bcc-Befehl dient zum Verzweigen innerhalb eines Programms abhängig von der jeweiligen Bedingung cc ("condition code"). Ist die im Befehl angegebene Bedingung erfüllt, so wird die Programmausführung an der Stelle PC + d weiter geführt. Die Adressdistanz ist eine Zahl im Zweierkomplement, die die relative Entfernung vom derzeitigen Programmzählerstand zur Marke, gezählt in Bytes, enthält. Der derzeitige Programmzählerstand ist die Adresse des aktuellen Verzweigungsbefehls + 2. Die Adressdistanz kann 8 oder 16 Bits groß sein. Wenn die 8-Bit-Adressdistanz im Befehlswort gleich Null ist, wird eine 16-Bit-Adressdistanz verwendet, die unmittelbar im Wort nach dem Befehlswort steht.

Ist die Bedingung im Verzweigungsbefehl nicht erfüllt, so geht der M 68000 weiter zum nächsten Befehl, der nach dem Verzweigungsbefehl Bcc steht.

Die mnemonischen Bezeichnungen, die für cc stehen können, zeigt die nachfolgende Tabelle.

Übersicht über die mnemonischen Codes (cc) der Sprungbefehle

Mnemonik	Englisch	Deutsch	Springe, wenn
Bedingte Sprünge			
BEQ	equal	gleich	Z = 1
BNE	not equal	Ungleich	Z = 0
BPL	plus	Positiv	N = 0
BMI	minus	Negativ	N = 1
*BGT	greater	Größer	Z + (N & V) = 0
*BLT	less	Kleiner	N & V = 1
*BGE	greater or equal	Größer gleich	N 6 V = 0
*BLE	less or equal	Kleiner gleich	Z + (N & V) = 1
BHI	higher	Höher	C + Z = 0
BLS	lower or same	Niedriger gleich	C + Z = 1
BGS	Carry set	Carry gesetzt	C = 1
BCC	Carry clear	Carry gelöscht	C = 0
*BVS	overflow	Überlauf	V = 1
*BVC	no overflow	Kein Überlauf	V = 0
Unbedingte Sprünge			
BRA	branch always	Springe immer	
BSR	br. to subroutine	Unterprogrammsprung	

Es bedeuten: " 2er-Komplement-Arithmetik
+ logisches ODER
& exklusiv ODER

JMP (1)

Springe

Assembler-Syntax
JMP <ea>

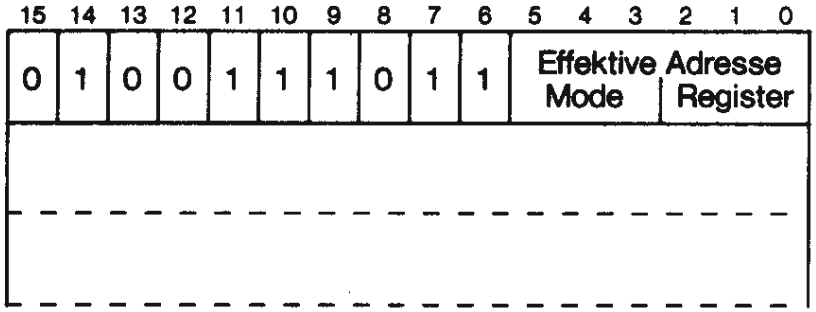
Operation:
<ea> -> <PC>

Welche Flags werden verändert? Keine

Operandengröße
Da mit dem JMP-Befehl kein Operand verarbeitet wird, gibt es auch keine Operandengröße.

Befehlsbeschreibung
Mit dem JMP-Befehl wird ein unbedingter Sprung zur angegebenen Adresse durchgeführt. Dort wird die Programmausführung fortgesetzt. Beim JMP-Befehl wird der Programmzähler des M68000 mit der im JMP-Befehl spezifizierten Adresse geladen.

JMP (2)



Anwendung

Falls man in einem Programm absolut (unbedingt) springen muss, um zum Beispiel einen Datenbereich zu überspringen, ist der JMP-Befehl dafür geeignet. Man könnte auch den BRA-Befehl verwenden, ist dann aber auf einen Verzweigungsbereich von +32k bis –32k beschränkt. Andererseits ergibt sich aus der Verwendung des BRA-Befehls der Vorteil, dass das Programm im Speicher verschiebbar (“relocatable“) bleibt, was beim JMP-Befehl nicht der Fall ist.

DBcc (1)

Assembler-Syntax

DBcc Dn, Marke

Operation

Falls Bedingung cc erfüllt:

<PC>+2 -> <PC> nächster Befehl nach DBcc

Falls Bedingung cc nicht erfüllt:

Dn-1 -> Dn;

Falls Dn ≠ -1 <PC>+d -> <PC> verzweige zur Marke

Falls Dn = -1 <PC>+2 -> <PC> nächster Befehl nach DBcc

d = Adressdistanz vom Befehlszähler (PC) zu Marke

Welche Flags werden verändert? Keine

Die Flags werden vom DBcc-Befehl nur abgefragt.

DBcc (2)

Operandengröße

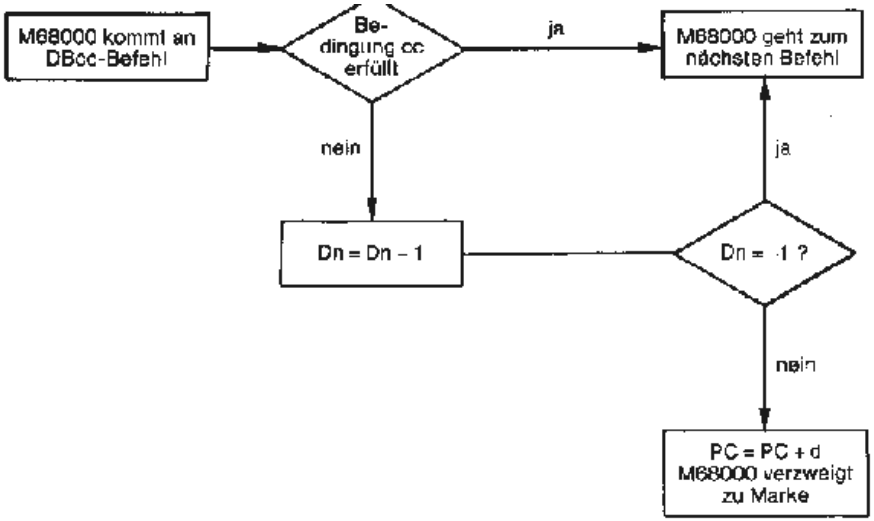
Dieser Befehl arbeitet nur mit Wortoperanden, d. h., der Schleifenzähler in Dn belegt nur das unterste Wort des Datenregisters, und die Adressdistanz d kann 16 Bit lang sein. Es sind also Sprung-Distanzen von +32 k bis -32 k möglich.

Befehlsbeschreibung

Der DBcc-Befehl dient hauptsächlich zum einfachen Aufbau einer **Zählschleife** in einem Assemblerprogramm. Der Befehl führt zum Ausführen von drei Aktionen im M68000:

1. Abfragen der Flags auf die Bedingung cc
2. evtl. Dekrementieren des Datenregisters, das den Schleifenzähler enthält
3. Sprung, wenn die Bedingung cc NICHT erfüllt ist UND das Datenregister NICHT den Wert -1 hat.

DBcc (3)



Die Bedingung für den DBcc-Befehl kann dabei mit denselben mnemonischen Codes angegeben werden wie im Bcc-Befehl. Im Gegensatz zum Bcc-Befehl wird jedoch hier nur verzweigt, wenn die Bedingung NICHT erfüllt ist. Bei erfüllter Bedingung verzweigt der DBcc-Befehl nicht!

Reservierung von Speicher, Anlegen von Konstanten (1)

DC.B	Operanden	definiere Bytes
DC.W	Operanden	definiere Wörter
DC.L	Operanden	definiere Doppelwörter
DS.B	z	reserviere z Bytes
DS.W	z	reserviere z Wörter
DS.L	z	reserviere z Doppelwörter

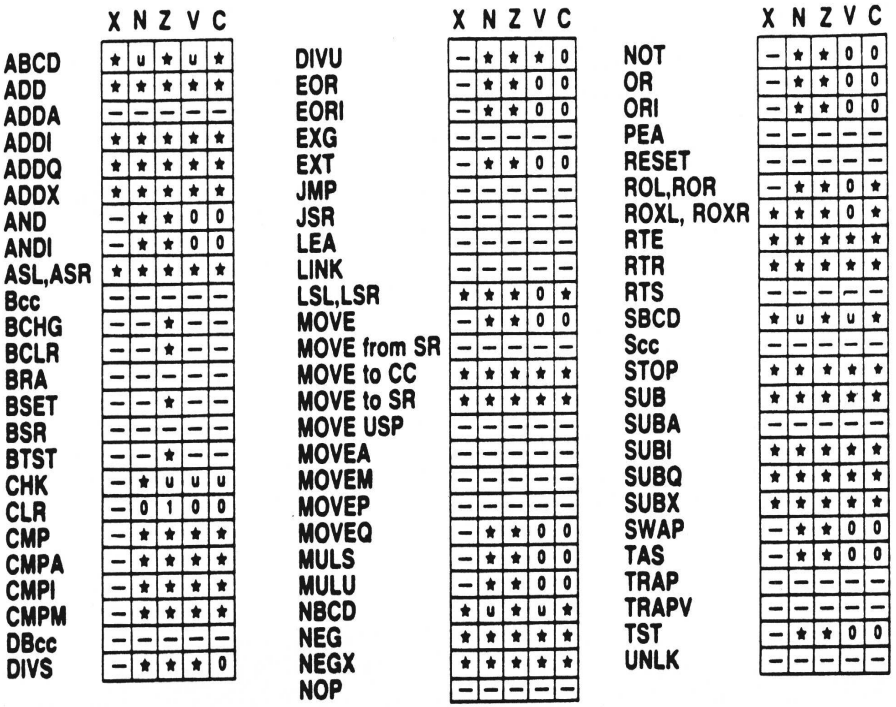
Reservierung von Speicher, Anlegen von Konstanten (2)

Beispiele

DC.B	'x'	ein Byte besetzt mit Zeichen 'x'
DC.B	'ABCD'	vier Bytes besetzt mit 'ABCD'
DC.B	10,5,7	drei Bytes besetzt mit 8-Bit-Zahlen 10,5,7
DC.W	10,5,7	drei Wörter besetzt mit 16-Bit-Zahlen 10,5,7
DC.L	$25	ein Doppelwort besetzt mit 32-Bit-Zahl 37
DC.L	Marke +(M1-M2)	ein Doppelwort besetzt mit der Adresse von “Marke“, erhöht um (M1-M2)
DS.B	10	reserviere 10 Bytes
DS.W	10	reserviere 10 Wörter
DS.L	10	reserviere 10 Doppelwörter
DS	$25	reserviere 37 Wörter
DS	0	setze Adresszählung auf nächste gerade Adresse

Überblick über den gesamten Befehlssatz des M 68000

Die Tabelle gibt zugleich an, wie die Flags im Statusregister SR von den Befehlen verändert werden.



Befehlssatz (1)

Mnemonic	Operation	Assembler-Syntax	Bedingungscodes				
			X	N	Z	V	O
ABCD	Addition dezimal mit Erweiterungs-bit	ABCD Dy, Dx ABCD -(Ay), -(Ax)	*	U	*	U	*
ADD	Addiere binär	ADD.s <ea>, Dn ADD.s Dn, <ea>	*	*	*	*	*
ADDA	Addiere Adresse	ADDA.s <ea>, An	-	-	-	-	-
ADDI	Addiere direkt	ADDI.s #<data>, <ea>	*	*	*	*	*
ADDQ	Addiere schnell	ADDQ.s #<data>, <ea>	*	*	*	*	*
ADDX	Addiere mit Er weiterungsbit	ADDX.s Dy, Dx ADDX.s -(Ay), -(Ax)	*	*	*	*	*
AND	Logisches UND	AND.s <ea>, Dn AND.s Dn, <ea>	-	*	*	0	0
ANDI	UND direkt	ANDI.s #<data>, <ea>	-	*	*	0	0
ASL,ASR	Arithmetische Verschiebung nach links, nach rechts	ASd.s Dx, D ASd # <data>, Dy ASd.s <ea>	*	*	*	*	*

Befehlssatz (2)

Mnemonic	Operation	Assembler-Syntax	Bedingungscodes				
			X	N	Z	V	0
BCC	Bedingter Sprung	Bcc <label>	-	-	-	-	-
BCHG	Prüfe ein Bit und ändere es	BCHG Dn, <ea> BCHG # <data>, <ea>	-	-	*	-	-
BCLR	Prüfe ein Bit und setze es auf 0	BCLR Dn, <ea> BCLR # <data>, <ea>	-	-	*	-	-
BRA	Unbedingter Sprung	BRA <label>	-	-	-	-	-
BSET	Prüfe ein Bit und setze es	BSET Dn, <ea> BSET # <data>,<ea>	-	-	*	-	-
BSR	Sprung zum Unterprogramm	BSR <label>	-	-	-	-	-
BTST	Prüfe ein Bit	BTST Dn, <ea>	-	-	*	-	-
CHK	Prüfe Register auf Grenzen	CHK <ea>, Dn	-	*	U	U	U
CLR	Setze Operand auf 0	CLR.x <ea>	-	0	1	0	0
CMP	Vergleiche	CMP.s <ea>, Dn	-	*	*	*	*
CMPA	Vergleiche Adresse	CMPA.s <ea>, An	-	*	*	*	*
CMPI	Vergleiche direct	CMPI.s #<data>, <ea>	-	*	*	*	*
CMPM	Vergleiche Speicher	CMPM.s (Ay)+, (Ax)+	-	*	*	*	*
DBcc -><-	Prüfe Bedingungen, vermindere und springe	DBcc Dn, <label>	-	-	-	-	-
DIVS	Division mit Vorzeichen	DIVS <ea>, Dn	-	*	*	*	0
DIVU	Division ohne Vorzeichen	DIVU <ea>, Dn	-	*	*	*	0

Befehlssatz (3)

Mnemonic	Operation	Assembler-Syntax	Bedingungscodes				
			X	N	Z	V	O
EOR	Logisches exklusiv ODER	EOR.s Dn, <ea>	-	*	*	0	0
EORI	Exklusives ODER direct	EORI.s #<data>,<ea>	-	*	*	0	0
EXG	Datentausch zwischen Register	EXG Rx, Ry	-	-	-	-	-
EXT	Vorzeichen-erweiterung	EXT.s Dn	-	*	*	0	0
JMP	Springe	JMP <ea>	-	-	-	-	-
JSR	Springe zum Unterpro-gramm	JSR <ea>	-	-	-	-	-
LEA	Lade die effektive Adresse	LEA <ea>, An	-	-	-	-	-
LINK	Verbinde und weise zu	LINK An, <distance>	-	-	-	-	-
LSL,LSR	Logische Verschiebung nach links	LSd.s Dx, Dy	*	*	*	0	*
MOVE	Transportiere Daten	MOVE.s <ea>, <ea>	-	*	*	0	0

Befehlssatz (4)

Mnemonic	Operation	Assembler-Syntax	Bedingungscodes				
			X	N	Z	V	O
MOVE zum CCR	Transportiere zum CCR	MOVE <ea>, CCR	*	*	*	*	*
MOVE zum SR	Transportiere zum Status-register	MOVE <ea>, SR	*	*	*	*	*
MOVE vom SR	Transportiere vom Status-register	MOVE SR, <ea>	-	-	-	-	-
MOVE USP	Transportiere den Anwender-Stackpointer	MOVE USP, An MOVE An, USP	-	-	-	-	-
MOVEA	Transportiere Adresse	MOVEA.s <ea>, An	-	-	-	-	-
MOVEM	Transportiere mehrere Re-gister	MOVEM.s <Register-liste>, <ea> MOVEM.s <ea>, <Registerliste>	-	-	-	-	-
MOVEP	Transportiere periphere Daten	MOVEP Dx, d(Ay) MOVEQ d(Ay), Dx	-	-	-	-	-

Befehlssatz (5)

Mnemonic	Operation	Assembler-Syntax	Bedingungscodes				
			X	N	Z	V	O
MOVEQ	Transportiere schnell	MOVEQ # <data>, Dn	-	*	*	0	0
MULS	Multiplikation mit Vorzeichen	MULS <ea>, Dn	-	*	*	0	0
MULU	Multiplikation ohne Vorzeichen	MULU <ea>, Dn	-	*	*	0	0
NBCD	Negiere dezimal mit Erweiterung	NBCD <ea>	*	U	*	U	*
NEG	Negiere	NEG.s <ea>	*	*	*	*	*
NEGX	Negiere mit Erweiterung	NEGX.s <ea>	*	*	*	*	*
NOP	Keine Operation	NOP	-	-	-	-	-
NOT	Logisches Komplement	NOT.s <ea>	-	*	*	*	*
OR	Logisches ODER	OR.s <ea> OR Dn, <ea>	-	*	*	0	0
PEA	Effektive Adr. auf den Stack	PEA <ea>	-	-	-	-	-
RESET	Normieren externer Einheiten	RESET	-	-	-	-	-

Befehlssatz (6)

Mnemonic	Operation	Assembler-Syntax	Bedingungscodes				
			X	N	Z	V	O
ROL, ROR	Ringver-schiebung nach links nach rechts	ROd.s Dx, Dy ROd.s #<data>, Dy ROd.s <ea>	-	*	*	0	*
ROXL, ROXR	Ringverschie-bung mit Erwei-terungsbit nach links, nach rechts	ROXd.s Dx, Dy ROXd.s #<data>, Dy ROXd.s <ea>	*	*	*	0	*
RTE	Springe zurück von Ausnahme	RTE	*	*	*	*	*
RTR	Springe zurück und ersetze Bedingungs-codes	RTR	*	*	*	*	*
RTS	Zurück vom Unterprogramm	RTS	-	-	-	-	-
SBCD	Subtrahiere dezimal mit Erweiterungs-bit	SBCD Dy, Dx SBCD -(Ay), -(Ax)	*	U	*	U	*
Scc	Setze in Abhängigkeit der Bedingung	Scc <ea>	-	-	-	-	-

Befehlssatz (7)

Mnemonic	Operation	Assembler-Syntax	Bedingungscodes				
			X	N	Z	V	O
STOP	Lade das Status-register und halte an	STOP #<data>	-	-	-	-	-
SUB	Subtrahiere binär	SUB.s <ea>, Dn SUB.s Dn, <ea>	*	*	*	*	*
SUBA	Subtrahiere Adresse	SUBA.s <ea>, An	-	-	-	-	-
SUBI	Subtrahiere direkt	SUBI.s #<data>,<ea>	*	*	*	*	*
SUBQ	Subtrahiere schnell	SUBQ.s #<data>,<ea>	*	*	*	*	*
SUBX	Subtrahiere mit Erweiterungsbit	SUBX.x Dy, Dx SUBX.s –(Ay), -(Ax)	*	*	*	*	*
SWAP	Vertausche Registerhälften	SWAP Dn	-	*	*	0	0
TAS	Teste und setze Operand	TAS <ea>	-	*	*	0	0
TRAP	Fangen	TRAP #<vector>	-	-	-	-	-
TRAPV	Fangen bei Überlauf	TRAPV	-	-	-	-	-
TST	Teste einen Operanden	TST.s <ea>	-	*	*	0	0
UNLK	Lösen	UNLK An	-	-	-	-	-

10.4 Programmbeispiele

Programmbeispiel 1

```
for (i=n;i>=0;i--) m[i]=0;
```

96							
97	00000014	3039	0000		MOVE.W	N, D0	D0 = N
		0034					
98	0000001A	41F9	0000		LEA	M, A0	A0=adr(m[0])
		0036					
99	00000020	6D00	000E		BLT	M2	(Fehler)
100							
101	00000024	3200		M1	MOVE.W	D0, D1	
102	00000026	E341			ASL.W	#1, D1	D1 = 2 * i
103	00000028	4270	1000		CLR.W	0(A0,D1.W)	m[i] := 0
104	0000002C	5340			SUBQ.W	#1, D0	i = i - 1
105	0000002E	6EF4			BGT	M1	if i>0 goto M1
106					.....		
107	00000030	6000	003C	M2	.....		Fehlerbeh.
108					.....		
109	00000034	001B		N	DC.W	27	
110	00000036			M	DS.W	28	m[0...n]
111							
112							

Programmbeispiel 1 (optimiert)

Dekrementieren des Indexregisters in Schritten von 2,
Wegfall des Registers D1

96							
97	00000014	3039	0000		MOVE.W	N, D0	D0 = N
		0032					
98	0000001A	41F9	0000		LEA	M-2, A0	A0 = adr(m[0])
		0032					
99	00000020	E340			ASL.W	#1, D0	Shift left
100	00000022	6D00	000A		BLT	M2	(Fehler)
101							
102	00000026	4270	0000	M1	CLR.W	0(A0,D0.W)	m[i] := 0
103	0000002A	5540			SUBQ.W	#2, D0	i = i - 1
104	0000002C	6EF8			BGT	M1	if i > 0 goto M1
105					....		
106	0000002E	6000	003C	M2	....		Fehlerbeh.
107					....		
108	00000032	001B		N	DC.W	27	
109	00000034			M	DS.W	28	m[1..n]
110							

Programmbeispiel 2

```
fac = 1;
for (i=n;i>0;i--) fac = fac*i;
```

```
106
107                 0000  0000  FAC     EQU       D0
108                 0000  0001  I       EQU       D1
109
110 0000001C        303C  0001        MOVE.W    #1, FAC
111 00000020        3239  0000        MOVE.W    N, I
                    003E
112 00000026        6F00  0012        BLE       ENDE
113
114 0000002A        C1C1        WD    MULS      I, FAC
115 0000002C        0C80  0000        CMPI.L    #$7FFF, FAC
                    7FFF
116 00000032        6E00  0006        BGT       UEBER
117 00000036        5341              SUBQ.W    #1, I
118 00000038        6EF0              BGT       WD
119                                   ....
120 ....                      ENDE    ....      Ende
    ....                      UEBER   ....      Überlauf
```