

## Mikroprozessorenkurs

# Auszug aus dem Befehlssatz der CPU32

Jann Philipp Oesch, dipl. El. Ing. HTL  
OESCH.ORG – Schönbühl und Leipzig  
Mail: jann@oesch.org

<b>1</b>	<b>Inhaltsverzeichnis</b>	
<b>1</b>	<b>INHALTSVERZEICHNIS</b>	<b>2</b>
<b>2</b>	<b>HINWEISE ZUM GEBRAUCH</b>	<b>3</b>
2.1	ALLGEMEINES	3
2.2	LESEN DER BEFEHLE	3
2.3	EINSCHRÄNKUNGEN	3
<b>3</b>	<b>AUSZUG AUS DEM BEFEHLSATZ</b>	<b>4</b>
3.1	ADD (ADD, ADDA, ADDI, ADDQ) – ADDIEREN	4
3.2	AND (AND, ANDI) – LOGISCHES AND MIT EINEM DATENREGISTER	5
3.3	BCC – BEDINGTER SPRUNG (RELATIV)	5
3.4	BCLR – LÖSCHEN EINES EINZELNEN BITS	6
3.5	BRA – UNBEDINGTER SPRUNG (RELATIV)	7
3.6	BSET – SETZEN EINES EINZELNEN BITS	7
3.7	BGND – STOPPT DIE PROGRAMMAUSFÜHRUNG (NUR MIT DEM BD32 DEBUGGINTERFACE)	8
3.8	BSR – SPRUNG AUF EINE SUBROUTINE (RELATIV)	8
3.9	BTST – PRÜFT EIN EINZELNES BIT	9
3.10	CLR – LÖSCHEN EINER EA (DATENREGISTER, SPEICHERPLATZ)	9
3.11	CMP (CMP, CMPA, CMPI) - VERGLEICHEN	10
3.12	DBCC – DEKREMENTIEREN UND BEDINGTE VERZWEIGUNG (RELATIV)	11
3.13	DIVS / DIVSL – DIVIDIEREN UNTER BERÜCKSICHTIGUNG DES VORZEICHENS	11
3.14	DIVU / DIVUL – DIVIDIEREN OHNE VORZEICHEN	12
3.15	EOR (EOR, EORI) – LOGISCHES EXKLUSIV – ODER (XOR / ANTIVALENZ) MIT EINEM DATENREGISTER	13
3.16	EXG – TAUSCHEN VON ZWEI REGISTERINHALTEN	14
3.17	JMP – UNBEDINGTER SPRUNG (ABSOLUT)	14
3.18	JSR – SPRUNG AUF EINE SUBROUTINE (ABSOLUT)	15
3.19	LEA – BERECHNEN DER EFFEKTIVEN ADRESSE UND SPEICHERN IN EIN ADRESSREGISTER	15
3.20	LSL – LOGISCHES LINKSSCHIEBEN EINES DATENREGISTERS	16
3.21	LSR – LOGISCHES RECHTSSCHIEBEN EINES DATENREGISTERS	17
3.22	MOVE (MOVE, MOVEA, MOVEQ) – SPEICHERN UND LADEN VON REGISTERINHALTEN	18
3.23	MOVEM – ABLEGEN (HOLEN) VON MEHREREN REGISTERN AUF DEN (VOM) STACK	19
3.24	MULS – MULTIPLIZIEREN UNTER BERÜCKSICHTIGUNG DES VORZEICHENS	20
3.25	MULU – MULTIPLIZIEREN OHNE VORZEICHEN	20
3.26	NEG – VORZEICHENWECHSEL EINER EA (DATENREGISTER, SPEICHERZELLE)	21
3.27	NOP – KEINE OPERATION (PLATZHALTER)	22
3.28	NOT – LOGISCHES KOMPLEMENT (NOT) EINES DATENREGISTERS	22
3.29	OR (OR, ORI) – LOGISCHES OR MIT EINEM DATENREGISTERS	22
3.30	PEA – BERECHNEN DER EFFEKTIVEN ADRESSE UND SPEICHERN AUF DEN STACK	23
3.31	ROL – ROTIEREN LINKS EINES DATENREGISTERS	24
3.32	ROR – ROTIEREN RECHTS EINES DATENREGISTERS	25
3.33	RTE – BEENDEN EINES AUSNAHMEZUSTANDES (INTERRUPT, SOFTWAREFEHLER)	25
3.34	RTS – BEENDEN EINES SUBPROGRAMMS	26
3.35	SUB (SUB, SUBA, SUBI, SUBQ) – SUBTRAHIEREN	26
3.36	SWAP – TAUSCHEN VON LOW-WORD UND HIGH-WORD EINES DATENREGISTERS	27
3.37	TST – PRÜFEN EINER EA (REGISTER, SPEICHERZELLE) UND SETZEN DER FLAGS	27
<b>4</b>	<b>VERZEICHNIS NACH FUNKTIONEN</b>	<b>29</b>
4.1	REGISTERFUNKTIONEN	29
4.2	ARITHMETISCHE FUNKTIONEN	29
4.3	SPRUNGFUNKTIONEN	30
4.4	SCHIEBE- UND ROTIERFUNKTIONEN	30
4.5	BIT – MANIPULIERFUNKTIONEN	30
4.6	VERGLEICHSFUNKTIONEN UND ANDERE BEFEHLE	30
<b>5</b>	<b>LITERATURVERZEICHNIS</b>	<b>31</b>

## 2 Hinweise zum Gebrauch

### 2.1 Allgemeines

Die hier vorliegende Befehlssammlung ist ein Auszug aus dem CPU32 Reference Manual [1] von Motorola. Es sind dabei die am häufigsten verwendeten Befehle aufgeführt. Die Sammlung soll einerseits das Suchen im Datenbuch erleichtern, andererseits aber auch sprachlich bedingte Missverständnisse aus dem Weg schaffen. Das vollständige, englische Datenbuch ist bei Motorola erhältlich. Gute Informationen kann man auch den MC68000-Büchern [2,3] vom Franzis-Verlag entnehmen.

### 2.2 Lesen der Befehle

Es gelten für die Befehlsbeschreibungen folgende Konventionen:

Dn, Dx, Dy	: Datenregister D0 – D7
An, Ax, Ay	: Adressregister A0 – A6 (A7)
Rn	: Daten- oder Adressregister D0 – D7 / A0 – A6 (A7)
d8	: 8-Bit Konstante vorzeichenbehaftet (-128 bis +127)
d16	: 16-Bit Konstante vorzeichenbehaftet (-32768 bis +32767)
d32	: 32-Bit Konstante vorzeichenbehaftet (-2147483648 bis +2147483647)
k	: Beliebige Konstante, gemäss Beschreibung
<EA>, EA	: Effektive Adresse, dies entspricht der effektiven Adresse gemäss Adressmode

Für weitere Anwendungsbeispiele und Anwendungen sei auf den Mikroprozessorkurs [5] verwiesen.

### 2.3 Einschränkungen

Die hier vorgestellten Befehle sind nur ein Auszug der vorhandenen Befehle. Exotische und selten benutzte Befehle wurden weggelassen, um die Übersicht zu verbessern. Zudem werden nicht abschliessend alle möglichen Varianten dargelegt. Dennoch kann mit den hier vorgestellten Befehlen jede beliebig komplexe Aufgabe bewältigt werden.

Die Befehle beziehen sich nur auf die CPU32, es sind keine CPU/MCU-spezifischen Instruktionen enthalten. Dazu sei das Buch MC68300 Mikrocontroller [4] empfohlen. Für den an der TEKO gehaltenen Kurs ist aber eine solch kostspielige Anschaffung unnötig.

### 3 Auszug aus dem Befehlssatz

#### 3.1 ADD (ADD, ADDA, ADDI, ADDQ) – Addieren

**Beschreibung:** Der Befehl ADD dient zum Addieren von zwei Operanden, wobei mindestens einer davon in einem Register (oder eine Konstante) sein muss. Der Befehl ADD wird benutzt, wenn Quelle oder Ziel ein Datenregister ist. Ist das Ziel ein Adressregister, muss ADDA benutzt werden, bei einer Konstantenaddition ADDI. Der Assembler erkennt normalerweise die richtige Schreibweise selbst, so genügt für alle drei Fälle ein ADD.

Der Befehl ADDQ wird für Konstanten von 1 – 8 benutzt. Er ist schneller und braucht weniger Speicherplatz als ADDI.

**Assembler Syntax:** ADD.s <EA>,Dn  
 ADD.s Dn,<EA>  
 ADDA.s <EA>,An  
 ADDI.s #kkkkkkkk,<EA>  
 ADDQ.s #k,<EA> (k=1..8)

**Operandengrösse:** s = B,W,L (Byte, Word und Long), bei Adressregistern nur s = W,L (Word, Long)

**Beeinflusste Flags:** X gleich wie C  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V '1' wenn des Resultat einen Überlauf verursacht hat, sonst '0'  
 C '1' wenn ein Übertrag (Carry) stattgefunden hat, sonst '0'

Erlaubte Adressierungsarten:

Quell - EA	Ziel – EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
Dn	ADD	ADDA	ADD	ADD	ADD	ADD	ADD	ADD
An	ADD	ADDA						
(An)	ADD	ADDA						
(An)+	ADD	ADDA						
-(An)	ADD	ADDA						
(d16,An)	ADD	ADDA						
(d8,An,Rn)	ADD	ADDA						
\$xxxxxxxx	ADD	ADDA						
#xxxxxxxx	ADDI	ADDA	ADDI	ADDI	ADDI	ADDI	ADDI	ADDI
#x (1-8)	ADDQ	ADDQ	ADDQ	ADDQ	ADDQ	ADDQ	ADDQ	ADDQ

**Beispiel:** D1 = \$12345678  
 ADDI.W #4321,D1  
 D1 = \$12349999

**Beispiel 2:** D1 = \$12345678      D2 = \$87654321  
 ADD.B D2,D1  
 D1 = \$12345699      D2 = \$87654321

### 3.2 AND (AND, ANDI) – Logisches AND mit einem Datenregister

**Beschreibung:** Mit dem Befehl AND kann eine logische AND-Verknüpfung von zwei Operanden durchgeführt werden, wobei mindestens einer davon in einem Datenregister (oder eine Konstante) sein muss. Der Befehl AND wird benutzt, wenn Quelle oder Ziel ein Datenregister ist. Ist die Quelle eine Konstante, muss ANDI verwendet werden. Der Assembler erkennt normalerweise die richtige Schreibweise selbst, so genügt ein AND.

**Assembler Syntax:** AND.s <EA>,Dn  
 AND.s Dn,<EA>  
 ANDI.s #kkkkkkkk,<EA>

**Operandengröße:** s = B,W,L (Byte, Word und Long), bei Adressregistern nur s = W,L (Word, Long)

**Beeinflusste Flags:** X keine Änderung  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V immer '0'  
 C immer '0'

**Erlaubte Adressierungsarten:**

Quell - EA	Ziel – EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxx
Dn	AND		AND	AND	AND	AND	AND	AND
An								
(An)	AND							
(An)+	AND							
-(An)	AND							
(d16,An)	AND							
(d8,An,Rn)	AND							
\$xxxxxxx	AND							
#xxxxxxx	ANDI		ANDI	ANDI	ANDI	ANDI	ANDI	ANDI

**Beispiel:** D1 = \$12345678  
 ANDI.W #4321,D1  
 D1 = \$12344220

**Beispiel 2:** D1 = \$12345678      D2 = \$87654321  
 AND.B D2,D1  
 D1 = \$12345620      D2 = \$87654321

### 3.3 Bcc – Bedingter Sprung (relativ)

**Beschreibung:** Bcc ist der Sammelbegriff aller bedingten, relativen Sprungbefehlen. Bedingt bedeutet, dass der Sprung an die angegebene Adresse nur ausgeführt wird, wenn eine spezielle Bedingung eingetroffen ist. Diese Bedingung knüpft sich an die Zustände der Flags. Der Befehl ist relativ, das bedeutet, es wird nur die Differenz von Ist-Adresse und Soll-Adresse mitgegeben. Diese Berechnung kann aber dem Assembler überlassen werden.

**Assembler Syntax:** Bcc.s kkkk

**Operandengröße:** s= S, {} (short, Bereich +/- 127 Bytes, ohne .S +/- 32767 Bytes)

Beeinflusste Flags: X keine Änderung  
 N keine Änderung  
 Z keine Änderung  
 V keine Änderung  
 C keine Änderung

Für das 'cc' im Bcc können folgende Bedingungen stehen (Auszug):

- BEQ Verzweigt, wenn das Z-Flag gesetzt ist, also wenn die letzte Operation 0 war  
 Bei CMP: Springen, wenn die verglichene Zahl genau identisch ist
- BNE Gegenstück zu BEQ, springt, wenn letzte Operation nicht 0 gab  
 Bei CMP: Springen, wenn die verglichene Zahl nicht identisch ist
- BMI Verzweigt, wenn das N-Flag gesetzt ist, also die letzte Operation eine negative Zahl ergab
- BPL Gegenstück zu BMI, springt bei einer positiven Zahl
- BLO/BCS Nach CMP: Springen, wenn die verglichene Zahl kleiner war
- BLS Nach CMP: Springen, wenn die verglichene Zahl kleiner oder gleich war
- BHS/BCC Nach CMP: Springen, wenn die verglichene Zahl grösser oder gleich war
- BHI Nach CMP: Springen, wenn die verglichene Zahl grösser war

Beispiel: BCC.S \$200800

Beispiel 2: BNE \$200800

### 3.4 BCLR – Löschen eines einzelnen Bits

Beschreibung: Der Befehl BCLR gehört zu den Bit-Manipulierbefehlen. Dadurch ist es möglich, ein bestimmtes Bit zu löschen (auf '0' zu bringen). Wird BCLR auf eine Speicheradresse angewendet, kann BCLR auf ein Byte zugreifen, es wird also Bit-Nummer 0..7 angesprochen. Ist der Befehl auf ein Datenregister angewendet, können alle 32 Bit einzeln gelöscht werden. Werden grössere Bitnummer angesprochen, wird eine Modulo-Division gemacht (Bit Nr. 32 = Bit Nr. 0).

Assembler Syntax: BCLR Dn,<EA>  
 BCLR #k,<EA> (k = 0..7 bei <EA> ≠ Dx, k = 0..31 bei <EA> = Dx)

Operandengrösse: keine (Byte oder Long, je nach <EA>)

Beeinflusste Flags: X keine Änderung  
 N keine Änderung  
 Z '1' wenn Resultat = 0, sonst '0'  
 V keine Änderung  
 C keine Änderung

Erlaubte Adressierungsarten:

Quell - EA	Ziel – EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
Dn	BCLR		BCLR	BCLR	BCLR	BCLR	BCLR	BCLR
#k	BCLR		BCLR	BCLR	BCLR	BCLR	BCLR	BCLR

Beispiel: D1 = \$12345678  
 BCLR #28,D1  
 D1 = \$02345678

Beispiel 2: D1 = \$12345678      D2 = \$87654323  
 BCLR D2,D1      \$87654323 MOD 32d = 3  
 D1 = \$12345670      D2 = \$87654323

### 3.5 BRA – Unbedingter Sprung (relativ)

Beschreibung: BRA gehört genau genommen zu den Befehlen der Bcc-Familie, ist jedoch nicht an eine Bedingung gebunden. Mit dem Befehl BRA wird ein Sprung auf die angegebene Adresse durchgeführt. Der Befehl ist relativ, das bedeutet, es wird nur die Differenz von Ist-Adresse und Soll-Adresse mitgegeben. Diese Berechnung kann aber dem Assembler überlassen werden.

Assembler Syntax: BRA.s kkkk

Operandengröße: s= S, {} (short, Bereich +/- 127 Bytes, ohne .S +/- 32767 Bytes)

Beeinflusste Flags: X keine Änderung  
 N keine Änderung  
 Z keine Änderung  
 V keine Änderung  
 C keine Änderung

Beispiel: BRA.S \$200800

Beispiel 2: BRA \$200800

### 3.6 BSET – Setzen eines einzelnen Bits

Beschreibung: Der Befehl BSET gehört zu den Bit-Manipulierbefehlen. Dadurch ist es möglich, ein bestimmtes Bit zu setzen (auf '1' zu bringen). Wird BSET auf eine Speicheradresse angewendet, kann BSET auf ein Byte zugreifen, es wird also Bit-Nummer 0..7 angesprochen. Ist der Befehl auf ein Datenregister angewendet, können alle 32 Bit einzeln gesetzt werden. Werden grössere Bitnummer angesprochen, wird eine Modulo-Division gemacht (Bit Nr. 32 = Bit Nr. 0).

Assembler Syntax: BSET Dn,<EA>  
 BSET #k,<EA> (k = 0..7 bei <EA> ≠ Dx, k = 0..31 bei <EA> = Dx)

Operandengröße: keine (Byte oder Long, je nach <EA>)

Beeinflusste Flags: X keine Änderung  
 N keine Änderung  
 Z '1' wenn Resultat = 0, sonst '0'  
 V keine Änderung  
 C keine Änderung

Erlaubte Adressierungsarten:

Quell - EA	Ziel – EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxx
Dn	BSET		BSET	BSET	BSET	BSET	BSET	BSET
#k	BSET		BSET	BSET	BSET	BSET	BSET	BSET

Beispiel: D1 = \$12345678  
 BSET #30,D1  
 D1 = \$52345678

Beispiel 2: D1 = \$12345678      D2 = \$87654321  
 BSET D2,D1      \$87654321 MOD 32d = 1  
 D1 = \$1234567A      D2 = \$87654321

### 3.7 BGND – Stoppt die Programmausführung (Nur mit dem BD32 Debugginterface)

Beschreibung: Der Befehl BGND (Background Debugging) wird zum Debuggen benutzt. Soll das Programm in Echtzeit durchlaufen bis zu einem bestimmten Punkt, so kann an diesem speziellen Punkt ein BGND im Programm eingefügt werden, und das Programm wird automatisch dort angehalten und der interne Debugger aktiviert. Ist das Programm fertig, müssen zwingend alle BGND entfernt werden, da ohne angeschlossenen Debugger das Programm abstürzen wird.

Assembler Syntax: BGND

Operandengröße: Keine, da keine Operanden folgen

Beeinflusste Flags: X keine Änderung  
 N keine Änderung  
 Z keine Änderung  
 V keine Änderung  
 C keine Änderung

### 3.8 BSR – Sprung auf eine Subroutine (relativ)

Beschreibung: Mit dem Befehl BSR wird ein Sprung auf eine Subroutine der angegebenen Adresse durchgeführt. Das bedeutet, die Adresse des nächsten Befehls wird auf den Stack gelegt, danach wird an die angegebene Adresse gesprungen und das Subprogramm bis zum RTS ausgeführt. Nach dem RTS wird das Programm nach dem BSR Befehl weiter ausgeführt. Der Befehl ist relativ, das bedeutet, es wird nur die Differenz von Ist-Adresse und Soll-Adresse mitgegeben. Diese Berechnung kann aber dem Assembler überlassen werden.

Assembler Syntax: BSR.s kkkk

Operandengröße: s= S, {} (short, Bereich +/- 127 Bytes, ohne .S +/- 32767 Bytes)

Beeinflusste Flags: X keine Änderung  
 N keine Änderung  
 Z keine Änderung  
 V keine Änderung  
 C keine Änderung

Beispiel: BSR.S \$200800

Beispiel 2: BSR \$200800

### 3.9 BTST – Prüft ein einzelnes Bit

**Beschreibung:** Der Befehl BTST gehört zu den Bit-Manipulierbefehlen. Dadurch ist es möglich, ein bestimmtes Bit zu prüfen (testen, ob das Bit '0' oder '1' ist). Wird BTST auf eine Speicheradresse angewendet, kann BTST auf ein Byte zugreifen, es wird also Bit-Nummer 0..7 angesprochen. Ist der Befehl auf ein Datenregister angewendet, können alle 32 Bit einzeln geprüft werden. Werden grössere Bitnummer angesprochen, wird eine Modulo-Division gemacht (Bit Nr. 32 = Bit Nr. 0).

**Assembler Syntax:** BTST Dn,<EA>  
BTST #k,<EA> (k = 0..7 bei <EA> ≠ Dx, k = 0..31 bei <EA> = Dx)

**Operandengrösse:** keine (Byte oder Long, je nach <EA>)

**Beeinflusste Flags:** X keine Änderung  
N keine Änderung  
Z '1' wenn das getestete Bit '0' ist, sonst '0'  
V keine Änderung  
C keine Änderung

Erlaubte Adressierungsarten:

	Ziel – EA							
Quell - EA	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
Dn	BTST		BTST	BTST	BTST	BTST	BTST	BTST
#k	BTST		BTST	BTST	BTST	BTST	BTST	BTST

**Beispiel:** D1 = \$12345678  
BTST #3,D1  
D1 = \$12345678 Z = '0'

**Beispiel 2:** D1 = \$12345678 D2 = \$87654321  
BTST D2,D1 \$87654321 MOD 32d = 1  
D1 = \$12345678 D2 = \$87654321 Z = '1'

### 3.10 CLR – Löschen einer EA (Datenregister, Speicherplatz)

**Beschreibung:** Der Befehl CLR dient zum Löschen eines Datenregisters oder eines Speicherplatzes. Das Resultat ist das gleiche wie mit MOVE #0,<EA>, der Befehl ist aber schneller und braucht weniger Speicherplatz.

**Assembler Syntax:** CLR.s <EA>

**Operandengrösse:** s = B,W,L (Byte, Word und Long)

**Beeinflusste Flags:** X keine Änderung  
N immer '0'  
Z immer '1'  
V immer '0'  
C immer '0'

Erlaubte Adressierungsarten:

	Ziel – EA							
Quell - EA	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
---	CLR		CLR	CLR	CLR	CLR	CLR	CLR

Beispiel: D1 = \$12345678  
 CLR.W #\$4321,D1  
 D1 = \$12340000

### 3.11 CMP (CMP, CMPA, CMPI) - Vergleichen

**Beschreibung:** Der Befehl CMP führt ein Vergleich von zwei Operanden durch, wobei mindestens einer davon in einem Datenregister (oder eine Konstante) sein muss. Es wird eine Subtraktion durchgeführt, und entsprechend die Flags gesetzt. Die Operanden werden bei diesem Befehl nicht geändert! Nach dem Vergleich kann im Programm mit einem bedingten Sprung (Bcc) entsprechend verzweigt werden. Der Befehl CMP wird benutzt, wenn Quelle oder Ziel ein Datenregister ist. Ist das Ziel ein Adressregister, wird CMPA benutzt, ist die Quelle eine Konstante, muss CMPI verwendet werden. Der Assembler erkennt normalerweise die richtige Schreibweise selbst, so genügt ein CMP.

**Assembler Syntax:** CMP.s <EA>,Dn  
 CMPA.s <EA>,An  
 CMPI.s #kkkkkkkk,<EA>

**Operandengröße:** s = B,W,L (Byte, Word und Long), bei Adressregistern nur s = W,L (Word, Long)

**Beeinflusste Flags:**  
 X keine Änderung  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V '1' wenn des Resultat einen Überlauf verursacht hat, sonst '0'  
 C '1' wenn ein Übertrag (Carry) stattgefunden hat, sonst '0'

**Erlaubte Adressierungsarten:**

Quell - EA	Ziel - EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxx
Dn	CMP	CMPA						
An	CMP	CMPA						
(An)	CMP	CMPA						
(An)+	CMP	CMPA						
-(An)	CMP	CMPA						
(d16,An)	CMP	CMPA						
(d8,An,Rn)	CMP	CMPA						
\$xxxxxxx	CMP	CMPA						
#xxxxxxx	CMPI	CMPA	CMPI	CMPI	CMPI	CMPI	CMPI	CMPI

Beispiel: D1 = \$12345678  
 CMPI.W #\$4321,D1  
 D1 = \$12345678 X=0 N=0 Z=0 V=0 C=0

Beispiel 2: D1 = \$12345678 D2 = \$87654321  
 AND.B D2,D1  
 D1 = \$12345678 D2 = \$87654321 X=0 N=0 Z=0 V=0 C=0

### 3.12 DBcc – Dekrementieren und bedingte Verzweigung (relativ)

**Beschreibung:** Der Befehl DBcc ist ein erweiterter Bcc Befehl. Der Befehl wird ausgeführt, wenn eine spezielle Bedingung **nicht** eingetroffen ist. Diese Bedingung knüpft sich an die Zustände der Flags. Wenn der Befehl ausgeführt wird, wird als erstes das Register Dn dekrementiert und getestet. Ist es  $\neq -1$ , wird der Sprung durchgeführt, sonst wird im Programm weitergemacht, wie auch wenn die Bedingung eingetroffen ist. Dieser Befehl eignet sich sehr gut für Schleifen. Der Sprung ist relativ, das bedeutet, es wird nur die Differenz von Ist-Adresse und Soll-Adresse mitgegeben. Diese Berechnung kann aber dem Assembler überlassen werden.

**Assembler Syntax:** DBcc Dn, kkkk

**Operandengröße:** keine, der Befehl ist auf Word wirksam und springt im Bereich +/- 32767 Bytes

**Beeinflusste Flags:**

X	keine Änderung
N	keine Änderung
Z	keine Änderung
V	keine Änderung
C	keine Änderung

Für das 'cc' im DBcc können folgende Bedingungen stehen (Auszug):

- DBEQ Schlaufe verlassen wenn das Z-Flag gesetzt ist, also wenn die letzte Operation 0 war  
Bei CMP: Schlaufe verlassen wenn die verglichene Zahl genau identisch ist
- DBNE Gegenstück zu BEQ, Schlaufe verlassen wenn letzte Operation nicht 0 gab  
Bei CMP: Schlaufe verlassen wenn die verglichene Zahl nicht identisch ist
- DBMI Schlaufe verlassen bei N-Flag gesetzt, bzw. die letzte Operation eine negative Zahl ergab.
- DBPL Gegenstück zu BMI, Schlaufe verlassen bei einer positiven Zahl
- DBCC Nach CMP: Schlaufe verlassen wenn die verglichene Zahl kleiner war
- DBLS Nach CMP: Schlaufe verlassen wenn die verglichene Zahl kleiner oder gleich war
- DBCS Nach CMP: Schlaufe verlassen wenn die verglichene Zahl grösser oder gleich war
- DBHI Nach CMP: Schlaufe verlassen wenn die verglichene Zahl grösser war
- DBF Der Sprung wird immer durchgeführt. Kriterium des Abbruchs ist nur das Register Dn. Ist Dn nach dem Dekrementieren = -1, wird der Sprung nicht mehr ausgeführt.

**Beispiel:** DBF D0, \$200800

**Beispiel 2:** DBNE D0, \$200800

### 3.13 DIVS / DIVSL – Dividieren unter Berücksichtigung des Vorzeichens

**Beschreibung:** Der Befehl DIVS/DIVSL wird zum Dividieren von vorzeichenbehafteten Zahlen eingesetzt. Das Resultat ist ganzzahlig und vorzeichenbehaftet, je nach DIVS-Befehl wird ein allfälliger Rest ausgewiesen oder nicht.  
Achtung, eine Division durch 0 gibt in jedem Fall einen Systemabsturz, da dies nicht erlaubt ist. Gute Programmierer prüfen vor jeder Division diesen Fall!

Assembler Syntax: DIVS.W <EA>,Dn (32Bit Dn) / (16Bit <EA>) = 16Bit Rest : 16Bit Quotient.  
 DIVS.L <EA>,Dq (32Bit Dn) / (32Bit <EA>) = 32Bit Quotient.  
 DIVS.L <EA>,Dr:Dq (64Bit Dr:Dq) / (32Bit <EA>) = 32Bit Rest : 32Bit Quotient.  
 DIVSL.L <EA>,Dr:Dq (32Bit Dq) / (32Bit <EA>) = 32Bit Rest : 32Bit Quotient.

Operandengrösse: nur die in der Syntax angegebenen gültig!!!

Beeinflusste Flags: X keine Änderung  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V '1' wenn Divisionsüberlauf, sonst '0'  
 C immer '0'

Erlaubte Adressierungsarten:

Divisor -	Dividend / Quotient – EA							
	Dn	Dq	Dr:Dq	Dq (Dr:Dq)				
Dn	DIVS.W	DIVS.L	DIVS.L	DIVSL.L				
An								
(An)	DIVS.W	DIVS.L	DIVS.L	DIVSL.L				
(An)+	DIVS.W	DIVS.L	DIVS.L	DIVSL.L				
-(An)	DIVS.W	DIVS.L	DIVS.L	DIVSL.L				
(d16,An)	DIVS.W	DIVS.L	DIVS.L	DIVSL.L				
(d8,An,Rn)	DIVS.W	DIVS.L	DIVS.L	DIVSL.L				
\$xxxxxxx	DIVS.W	DIVS.L	DIVS.L	DIVSL.L				
#xxxxxxx	DIVS.W	DIVS.L	DIVS.L	DIVSL.L				

Beispiel: D1 = \$000000420  
 DIVS.W #4,D1  
 D1 = \$00200100 → Resultat: \$100, Rest \$20

Beispiel 2: D1 = \$00000020 D0 = \$000001FF  
 DIVS.L #200,D1:D0  
 D1 = \$000001FF D0 = \$10000000 → Resultat \$10000000, Rest \$1FF

### 3.14 DIVU / DIVUL – Dividieren ohne Vorzeichen

Beschreibung: Der Befehl DIVU/DIVUL wird zum Dividieren von vorzeichenlosen Zahlen eingesetzt. Das Resultat ist ganzzahlig und vorzeichenlos, je nach DIVU-Befehl wird ein allfälliger Rest ausgewiesen oder nicht.  
 Achtung, eine Division durch 0 gibt in jedem Fall einen Systemabsturz, da dies nicht erlaubt ist. Gute Programmierer prüfen vor jeder Division diesen Fall!

Assembler Syntax: DIVU.W <EA>,Dn (32Bit Dn) / (16Bit <EA>) = 16Bit Rest : 16Bit Quotient.  
 DIVU.L <EA>,Dq (32Bit Dn) / (32Bit <EA>) = 32Bit Quotient.  
 DIVU.L <EA>,Dr:Dq (64Bit Dr:Dq) / (32Bit <EA>) = 32Bit Rest : 32Bit Quotient.  
 DIVUL.L <EA>,Dr:Dq (32Bit Dq) / (32Bit <EA>) = 32Bit Rest : 32Bit Quotient.

Operandengrösse: nur die in der Syntax angegebenen gültig!!!

Beeinflusste Flags: X keine Änderung  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V '1' wenn Divisionsüberlauf, sonst '0'  
 C immer '0'

Erlaubte Adressierungsarten:

	Dividend / Quotient – EA							
Divisor –	Dn	Dq	Dr:Dq	Dq (Dr:Dq)				
Dn	DIVU.W	DIVU.L	DIVU.L	DIVUL.L				
An								
(An)	DIVU.W	DIVU.L	DIVU.L	DIVUL.L				
(An)+	DIVU.W	DIVU.L	DIVU.L	DIVUL.L				
-(An)	DIVU.W	DIVU.L	DIVU.L	DIVUL.L				
(d16,An)	DIVU.W	DIVU.L	DIVU.L	DIVUL.L				
(d8,An,Rn)	DIVU.W	DIVU.L	DIVU.L	DIVUL.L				
\$xxxxxxx	DIVU.W	DIVU.L	DIVU.L	DIVUL.L				
#xxxxxxx	DIVU.W	DIVU.L	DIVU.L	DIVUL.L				

Beispiel: D1 = \$000000420  
 DIVU.W # \$40, D1  
 D1 = \$00200010 → Resultat: \$10, Rest \$20

Beispiel 2: D1 = \$00000020 D0 = \$000001FF  
 DIVU.L # \$200, D1:D0  
 D1 = \$000001FF D0 = \$10000000 → Resultat \$10000000, Rest \$1FF

### 3.15 EOR (EOR, EORI) – Logisches Exklusiv – Oder (XOR / Antivalenz) mit einem Datenregister

Beschreibung: Mit dem Befehl EOR kann eine logische EOR-Verknüpfung von zwei Operanden durchgeführt werden, wobei mindestens einer davon in einem Datenregister (oder eine Konstante) sein muss. Der Befehl EOR wird benutzt, wenn Quelle oder Ziel ein Datenregister ist. Ist die Quelle eine Konstante, muss EORI verwendet werden. Der Assembler erkennt normalerweise die richtige Schreibweise selbst, so genügt ein EOR.

Assembler Syntax: EOR.s Dn,<EA>  
 EORI.s #kkkkkkkk,<EA>

Operandengröße: s = B,W,L (Byte, Word und Long), bei Adressregistern nur s = W,L (Word, Long)

Beeinflusste Flags: X keine Änderung  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V immer '0'  
 C immer '0'

Erlaubte Adressierungsarten:

Quell - EA	Ziel – EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
Dn	EOR		EOR	EOR	EOR	EOR	EOR	EOR
An								
(An)								
(An)+								
-(An)								
(d16,An)								
(d8,An,Rn)								
\$xxxxxxxx								
#xxxxxxxx	EORI		EORI	EORI	EORI	EORI	EORI	EORI

Beispiel: D1 = \$12345678  
EORI.W # \$4321, D1  
D1 = \$12341559

Beispiel 2: D1 = \$12345678      D2 = \$87654321  
EOR.B D2, D1  
D1 = \$12345659      D2 = \$87654321

### 3.16 EXG – Tauschen von zwei Registerinhalten)

Beschreibung: Mit EXG können Adressregister und Datenregister untereinander ausgetauscht werden.

Assembler Syntax: EXG.s Dx, Dy  
EXG.s Dx, Ay  
EXG.s Ax, Dy  
EXG.s Ax, Ay

Operandengröße: s = L (Long)

Beeinflusste Flags: X keine Änderung  
N keine Änderung  
Z keine Änderung  
V keine Änderung  
C keine Änderung

Beispiel: D1 = \$12345678      A3 = \$87654321  
EXG.L D1, A3  
D1 = \$87654321      A3 = \$12345678

### 3.17 JMP – Unbedingter Sprung (absolut)

Beschreibung: Mit dem Befehl JMP wird ein Sprung auf die angegebene Adresse durchgeführt. Der Befehl ist absolut, das bedeutet, es wird an die angegebene Adresse gesprungen.

Assembler Syntax: JMP <EA>

Operandengröße: keine

Beeinflusste Flags: X keine Änderung  
 N keine Änderung  
 Z keine Änderung  
 V keine Änderung  
 C keine Änderung

Erlaubte Adressierungsarten:

	Ziel – EA							
Quell - EA	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
---			JMP			JMP	JMP	JMP

Beispiel: JMP \$200800

### 3.18 JSR – Sprung auf eine Subroutine (absolut)

Beschreibung: Mit dem Befehl JSR wird ein Sprung auf eine Subroutine der angegebenen Adresse durchgeführt. Das bedeutet, die Adresse des nächsten Befehls wird auf den Stack gelegt, danach wird an die angegebene Adresse gesprungen und das Subprogramm bis zum RTS ausgeführt. Nach dem RTS wird das Programm nach dem JSR Befehl weiter ausgeführt. Der Befehl ist absolut, das bedeutet, es wird genau an die angegebene Adresse gesprungen.

Assembler Syntax: JSR <EA>

Operandengrösse: keine

Beeinflusste Flags: X keine Änderung  
 N keine Änderung  
 Z keine Änderung  
 V keine Änderung  
 C keine Änderung

Erlaubte Adressierungsarten:

	Ziel – EA							
Quell - EA	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
---			JSR			JSR	JSR	JSR

Beispiel: JSR \$200800

### 3.19 LEA – Berechnen der effektiven Adresse und speichern in ein Adressregister

Beschreibung: Mit dem Befehl LEA wird eine effektive Adresse (EA) berechnet und in einem Adressregister abgelegt. Dadurch können beispielsweise Zeigerberechnungen schnell durchgeführt werden.

Assembler Syntax: LEA <EA>,An

Operandengrösse: keine (Immer Long)

Beeinflusste Flags: X keine Änderung  
 N keine Änderung  
 Z keine Änderung  
 V keine Änderung  
 C keine Änderung

Erlaubte Adressierungsarten:

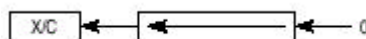
	Ziel – EA							
Quell - EA	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxx
---			LEA			LEA	LEA	LEA

Beispiel: A0 = \$87654321  
 LEA \$12345678,A0  
 A0 = \$12345678

Beispiel 2: A0 = \$11223344      A3 = \$12345678      D4 = \$88776655  
 LEA \$11,(A0,D4.W),A3  
 A0 = \$11223344      A3 = Inhalt von \$112299AA      D4 = \$88776655

### 3.20 LSL – Logisches Linksschieben eines Datenregisters

Beschreibung: Der Befehl LSL gehört zu den Schiebefunktionen. Es findet eine logische Schiebung nach links statt. Wird in einem Datenregister geschoben, kann die Anzahl der Schiebebewegungen angegeben werden. Findet das Schieben in einer Speicherzelle statt, wird immer nur um eine Stelle geschoben. Wenn die Anzahl der Schiebungen mit einer Konstanten angegeben werden, können nur 1..8 Schiebungen pro Befehl erzeugt werden, wird die Angabe mit einem Datenregister gemacht, kann 0..63 mal geschoben werden. Werden grössere Zahlen angegeben, wird eine Modulo-Division mit 64 gemacht. Nachgeschoben in das frei werdende Bit wird in jedem Fall eine '0'. Das herausfallende Bit ganz links wird in das C und das X-Flag übertragen.



Assembler Syntax: LSL.s Dx,Dy (Dx = 0..63)  
 LSL.s #k,Dy (k = 1..8)  
 LSL.s <EA>

Operandengrösse: s = B,W,L (Byte, Word und Long)

Beeinflusste Flags: X gesetzt gemäss dem hinausgeschobenen Bit ganz links  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V immer '0'  
 C gesetzt gemäss dem hinausgeschobenen Bit ganz links

Erlaubte Adressierungsarten:

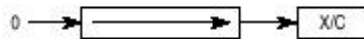
Quell - EA	Ziel – EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
Dn	LSL							
#k (1..8)	LSL							
---			LSL	LSL	LSL	LSL	LSL	LSL

Beispiel: D1 = \$12345678  
 LSL #4,D1  
 D1 = \$23456780 X,C = '1'

Beispiel 2: D1 = \$12345678 D2 = \$87654308  
 LSL D2,D1 \$87654308 MOD 64d = 8  
 D1 = \$34567800 D2 = \$87654308 X,C = '0'

### 3.21 LSR – Logisches Rechtsschieben eines Datenregisters

Beschreibung: Der Befehl LSR gehört zu den Schiebefunktionen. Es findet eine logische Schiebung nach rechts statt. Wird in einem Datenregister geschoben, kann die Anzahl der Schiebebewegungen angegeben werden. Findet das Schieben in einer Speicherzelle statt, wird immer nur um eine Stelle geschoben. Wenn die Anzahl der Schiebungen mit einer Konstanten angegeben werden, können nur 1..8 Schiebungen pro Befehl erzeugt werden, wird die Angabe mit einem Datenregister gemacht, kann 0..63 mal geschoben werden. Werden grössere Zahlen angegeben, wird eine Modulo-Division mit 64 gemacht. Nachgeschoben in das frei werdende Bit wird in jedem Fall eine '0'. Das herausfallende Bit ganz rechts wird in das C und das X-Flag übertragen.



Assembler Syntax: LSR.s Dx,Dy (Dx = 0..63)  
 LSR.s #k,Dy (k = 1..8)  
 LSR.s <EA>

Operandengrösse: s = B,W,L (Byte, Word und Long)

Beeinflusste Flags: X gesetzt gemäss dem hinausgeschobenen Bit ganz rechts  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V immer '0'  
 C gesetzt gemäss dem hinausgeschobenen Bit ganz rechts

Erlaubte Adressierungsarten:

Quell - EA	Ziel – EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
Dn	LSR							
#k (1..8)	LSR							
---			LSR	LSR	LSR	LSR	LSR	LSR

Beispiel: D1 = \$12345678  
 LSR #4,D1  
 D1 = \$012345678 X,C = '1'

Beispiel 2: D1 = \$12345678 D2 = \$87654308  
 BCLR D2,D1 \$87654308 MOD 64d = 8  
 D1 = \$00123456 D2 = \$87654308 X,C = '0'

**3.22 MOVE (MOVE, MOVEA, MOVEQ) – Speichern und laden von Registerinhalten**

**Beschreibung:** Der Befehl MOVE ist der wohl wichtigste im ganzen Befehlssatz. Dieser wird verwendet, um Daten herumschieben (engl. move). In anderen Assemblersprachen gibt es meist ein LD (Load) und ein ST (Store), bei der CPU32 wird dies alles mit MOVE abgehandelt. Der Befehl MOVE wird benutzt, wenn das Ziel kein Adressregister ist. Ist das Ziel ein Adressregister, muss MOVEA benutzt werden. Der Assembler erkennt normalerweise die richtige Schreibweise selbst, so genügt ein MOVE sowohl für Daten und Adressregister.  
 Der Befehl MOVEQ wird für das Laden von Konstanten im Bereich 0 – ±127 in ein Datenregister benutzt. Er ist schneller und braucht weniger Speicherplatz als MOVE.  
 Es gibt noch weitere spezielle MOVE-Instruktionen, die aber hier nicht erläutert werden, da sie tief in der Systemprogrammierung verwendet werden.

**Assembler Syntax:** MOVE.s <EA>,<EA> (Ziel <EA> ≠ An)  
 MOVEA.s <EA>,An  
 MOVEQ.L #d8,Dn

**Operandengröße:** s = B,W,L (Byte, Word und Long), bei Adressregistern nur s = W,L (Word, Long)

**Beeinflusste Flags:** X keine Änderung  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V immer '0'  
 C immer '0'

**Erlaubte Adressierungsarten:**

Quell – EA	Ziel – EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
Dn	MOVE	MOVEA	MOVE	MOVE	MOVE	MOVE	MOVE	MOVE
An	MOVE	MOVEA	MOVE	MOVE	MOVE	MOVE	MOVE	MOVE
(An)	MOVE	MOVEA	MOVE	MOVE	MOVE	MOVE	MOVE	MOVE
(An)+	MOVE	MOVEA	MOVE	MOVE	MOVE	MOVE	MOVE	MOVE
-(An)	MOVE	MOVEA	MOVE	MOVE	MOVE	MOVE	MOVE	MOVE
(d16,An)	MOVE	MOVEA	MOVE	MOVE	MOVE	MOVE	MOVE	MOVE
(d8,An,Rn)	MOVE	MOVEA	MOVE	MOVE	MOVE	MOVE	MOVE	MOVE
\$xxxxxxxx	MOVE	MOVEA	MOVE	MOVE	MOVE	MOVE	MOVE	MOVE
#xxxxxxxx	MOVE	MOVEA	MOVE	MOVE	MOVE	MOVE	MOVE	MOVE
#d8	MOVEQ							

Beispiel: D1 = \$12345678  
 MOVE.W #\$4321,D1  
 D1 = \$12344321

Beispiel 2: D1 = \$12345678 D2 = \$87654321  
 MOVE.B D2,D1  
 D1 = \$12345621 D2 = \$87654321

Beispiel 3: D1 = \$12345678  
 MOVEQ.L #\$21,D1  
 D1 = \$00000021

### 3.23 MOVEM – Ablegen (holen) von mehreren Registern auf den (vom) Stack

**Beschreibung:** Mit dem Befehl MOVEM ist es möglich, mehrere Register (Adress- und Datenregister) im Speicher abzulegen und wieder zu holen. Meist wird der Befehl benutzt, um Register auf dem Stack zu sichern, weil die Register für andere Zwecke eingesetzt werden. Später kann man leicht den alten Zustand wieder herstellen. Die Register werden unabhängig von der angegebenen Reihenfolge immer gleich abgelegt.

**Assembler Syntax:** MOVEM.s <register list>,<EA>  
 MOVEM.s <EA>,<register list>

**Operandengrösse:** s = W,L (Word oder Long)

**Beeinflusste Flags:** X keine Änderung  
 N keine Änderung  
 Z keine Änderung  
 V keine Änderung  
 C keine Änderung

**Register List:** Beschreibende Aufzählung aller Register, die mit diesem Befehl transferiert werden sollen. Erlaubt ist z.B.  
 D0/D4/A4/A6 Einzelnes aufzählen aller Register  
 D0-D3/A1-A3 Angabe von Bereichen (hier: D0/D1/D2/D3/A1/A2/A3)

Erlaubte Adressierungsarten:

	Ziel – EA							
Quell – EA	<reg list>		(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxx
	<reg list>		MOVEM		MOVEM	MOVEM	MOVEM	MOVEM
	(An)	MOVEM						
	(An)+	MOVEM						
	-(An)							
	(d16,An)	MOVEM						
	(d8,An,Rn)	MOVEM						
	\$xxxxxxx	MOVEM						
	#xxxxxxx							

Beispiel: MOVEM.L D1-D3/A0-A4,-(A7) → Ablage von 8 Registern auf den Stack

Beispiel 2: MOVEM.L (A7)+,D2/D6/A5 → Holen von 3 Registern vom Stack

### 3.24 MULS – Multiplizieren unter Berücksichtigung des Vorzeichens

Beschreibung: Der Befehl MULS wird zum Multiplizieren von vorzeichenbehafteten Zahlen eingesetzt. Das Resultat der Operation wird somit auch vorzeichenbehaftet.

Assembler Syntax: MULS.W <EA>,Dn (16Bit Dn) x (16Bit <EA>) = 32Bit Dn  
 MULS.L <EA>,DI (32Bit DI) x (32Bit <EA>) = 32Bit DI  
 MULS.L <EA>,Dh:DI (32Bit DI) / (32Bit <EA>) = 64Bit Dh:DI.

Operandengrösse: nur die in der Syntax angegebenen gültig!!!

Beeinflusste Flags: X keine Änderung  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V '1' wenn Divisionsüberlauf, sonst '0'  
 C immer '0'

Erlaubte Adressierungsarten:

	Multiplikand / Produkt – EA						
Multip. - EA	Dn	DI	Dh:DI				
Dn	MULS.W	MULS.L	MULS.L				
An							
(An)	MULS.W	MULS.L	MULS.L				
(An)+	MULS.W	MULS.L	MULS.L				
-(An)	MULS.W	MULS.L	MULS.L				
(d16,An)	MULS.W	MULS.L	MULS.L				
(d8,An,Rn)	MULS.W	MULS.L	MULS.L				
\$xxxxxxx	MULS.W	MULS.L	MULS.L				
#xxxxxxx	MULS.W	MULS.L	MULS.L				

Beispiel: D1 = \$000000020  
 MULS.W #\$4,D1  
 D1 = \$00000400

Beispiel 2: D1 = \$00000000      D0 = \$10000000  
 MULS.L #\$200,D1:D0  
 D1 = \$00000020      D0 = \$00000000

### 3.25 MULU – Multiplizieren ohne Vorzeichen

Beschreibung: Der Befehl MULU wird zum Multiplizieren von vorzeichenlosen Zahlen eingesetzt. Das Resultat der Operation wird somit auch vorzeichenlos.

Assembler Syntax: MULU.W <EA>,Dn (16Bit Dn) x (16Bit <EA>) = 32Bit Dn  
 MULU.L <EA>,DI (32Bit DI) x (32Bit <EA>) = 32Bit DI  
 MULU.L <EA>,Dh:DI (32Bit DI) / (32Bit <EA>) = 64Bit Dh:DI.

Operandengrösse: nur die in der Syntax angegebenen gültig!!!

Beeinflusste Flags: X keine Änderung  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V '1' wenn Divisionsüberlauf, sonst '0'  
 C immer '0'

Erlaubte Adressierungsarten:

	Multiplikand / Produkt – EA							
Multip. - EA	Dn	DI	Dh:DI					
Dn	MULU.W	MULU.L	MULU.L					
An								
(An)	MULU.W	MULU.L	MULU.L					
(An)+	MULU.W	MULU.L	MULU.L					
-(An)	MULU.W	MULU.L	MULU.L					
(d16,An)	MULU.W	MULU.L	MULU.L					
(d8,An,Rn)	MULU.W	MULU.L	MULU.L					
\$xxxxxxxx	MULU.W	MULU.L	MULU.L					
#xxxxxxxx	MULU.W	MULU.L	MULU.L					

Beispiel: D1 = \$000000020  
MULU.W #\$4,D1  
D1 = \$00000080

Beispiel 2: D1 = \$00000000      D0 = \$10000000  
MULU.L #\$200,D1:D0  
D1 = \$00000020      D0 = \$00000000

### 3.26 NEG – Vorzeichenwechsel einer EA (Datenregister, Speicherzelle)

Beschreibung: Mit dem Befehl NEG wird ein Vorzeichenwechsel von einem Operanden durchgeführt. Dies entspricht der Taste +/- auf einem Taschenrechner. Die Operation entspricht: EA = 0 – EA.

Assembler Syntax: NEG.s <EA>

Operandengröße: s = B,W,L (Byte, Word und Long)

Beeinflusste Flags: X gleich wie C  
N '1' wenn Resultat negativ, sonst '0'  
Z '1' wenn Resultat = 0, sonst '0'  
V '1' wenn des Resultat einen Überlauf verursacht hat, sonst '0'  
C '1' wenn ein Übertrag (Carry) stattgefunden hat, sonst '0'

Erlaubte Adressierungsarten:

	Ziel – EA							
Quell - EA	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
---	NEG		NEG	NEG	NEG	NEG	NEG	NEG

Beispiel: D1 = \$12345678  
NEG.W D1  
D1 = \$1234A988

### 3.27 NOP – Keine Operation (Platzhalter)

**Beschreibung:** Der Befehl NOP (No Operation) benutzt nur Speicherplatz, braucht etwas Rechenzeit, aber hat keine Funktion. Der Befehl wird meist beim Debuggen benutzt, um einen bestehenden Befehl zu überschreiben oder um eine kurze Wartezeit zu erhalten.

**Assembler Syntax:** NOP

**Operandengröße:** Keine, da keine Operanden folgen

**Beeinflusste Flags:**

X	keine Änderung
N	keine Änderung
Z	keine Änderung
V	keine Änderung
C	keine Änderung

**Beispiel:** NOP

### 3.28 NOT – Logisches Komplement (NOT) eines Datenregisters

**Beschreibung:** Mit dem Befehl NOT wird das logische Komplement eines Operanden gebildet.

**Assembler Syntax:** NOT.s <EA>

**Operandengröße:** s = B,W,L (Byte, Word und Long)

**Beeinflusste Flags:**

X	keine Änderung
N	'1' wenn Resultat negativ, sonst '0'
Z	'1' wenn Resultat = 0, sonst '0'
V	immer '0'
C	immer '0'

**Erlaubte Adressierungsarten:**

	Ziel – EA							
Quell - EA	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
---	NOT		NOT	NOT	NOT	NOT	NOT	NOT

**Beispiel:**

```
D1 = $12345678
NOT.W D1
D1 = $1234A987
```

### 3.29 OR (OR, ORI) – Logisches OR mit einem Datenregisters

**Beschreibung:** Mit dem Befehl OR kann ein logisches OR von zwei Operanden durchgeführt werden, wobei mindestens einer davon in einem Datenregister (oder eine Konstante) sein muss. Der Befehl OR wird benutzt, wenn Quelle oder Ziel ein Datenregister ist. Ist die Quelle eine Konstante, muss ORI verwendet werden. Der Assembler erkennt normalerweise die richtige Schreibweise selbst, so genügt ein OR.

**Assembler Syntax:**

```
OR.s <EA>,Dn
OR.s Dn,<EA>
ORI.s #kkkkkkkk,<EA>
```

**Operandengröße:** s = B,W,L (Byte, Word und Long), bei Adressregistern nur s = W,L (Word, Long)

Beeinflusste Flags: X keine Änderung  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V immer '0'  
 C immer '0'

Erlaubte Adressierungsarten:

	Ziel – EA							
Quell - EA	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
Dn	OR		OR	OR	OR	OR	OR	OR
An								
(An)	OR							
(An)+	OR							
-(An)	OR							
(d16,An)	OR							
(d8,An,Rn)	OR							
\$xxxxxxxx	OR							
#xxxxxxxx	ORI		ORI	ORI	ORI	ORI	ORI	ORI

Beispiel: D1 = \$12345678  
 ORI.W #\$4321,D1  
 D1 = \$12345779

Beispiel 2: D1 = \$12345678      D2 = \$87654321  
 OR.B D2,D1  
 D1 = \$12345679      D2 = \$87654321

### 3.30 PEA – Berechnen der effektiven Adresse und speichern auf den Stack

Beschreibung: Mit dem Befehl PEA wird eine effektive Adresse (EA) berechnet und auf den Stack gelegt. Dies ist besonders für hochsprachnahe Programmierungen sehr nützlich (Parameterübergabe über den Stack).

Assembler Syntax: PEA <EA>

Operandengrösse: keine (Immer Long)

Beeinflusste Flags: X keine Änderung  
 N keine Änderung  
 Z keine Änderung  
 V keine Änderung  
 C keine Änderung

Erlaubte Adressierungsarten:

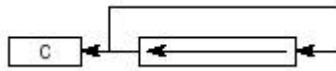
	Ziel – EA							
Quell - EA	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
---			PEA			PEA	PEA	PEA

Beispiel: -4(A7) = \$87654321  
 PEA \$12345678,A0  
 (A7) = \$12345678      A7=A7-4

Beispiel 2:            -4(A7) = \$11223344    A3 = \$12345678        D4 = \$88776655  
                          LEA \$11,(A0,D4.W),A3  
                          (A7) = \$11223344    A3 = Inhalt von \$112299AA    D4 = \$88776655    A7=A7-4

### 3.31 ROL – Rotieren links eines Datenregisters

Beschreibung:        Der Befehl ROL gehört zu den Schiebefunktionen. Es findet eine logische Rotation nach links statt. Dies entspricht dem Befehl LSL, jedoch wird nicht eine '0' nachgeschoben, sondern der hinausgeschobene Teil wird rechts eingeschoben. Wird in einem Datenregister rotiert, kann die Anzahl der Rotierbewegungen angegeben werden. Findet das Rotieren in einer Speicherzelle statt, wird immer nur um eine Stelle rotiert. Wenn die Anzahl der Rotationen mit einer Konstanten angegeben werden, können nur 1..8 Bit rotiert werden pro Befehl, wird die Angabe mit einem Datenregister gemacht, kann 0..63 mal rotiert werden. Werden grössere Zahlen angegeben, wird eine Modulo-Division mit 64 gemacht. Das rotierte, bzw. zuletzt rotierte Bit ganz links wird in das C-Flag übertragen.



Assembler Syntax:    ROL.s Dx,Dy (Dx = 0..63)  
                          ROL.s #k,Dy (k = 1..8)  
                          ROL.s <EA>

Operandengrösse:    s = B,W,L (Byte, Word und Long)

Beeinflusste Flags:    X        keine Änderung  
                          N        '1' wenn Resultat negativ, sonst '0'  
                          Z        '1' wenn Resultat = 0, sonst '0'  
                          V        immer '0'  
                          C        gesetzt gemäss dem letzten rotierten Bit ganz links

Erlaubte Adressierungsarten:

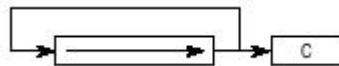
Quell - EA	Ziel – EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
Dn	ROL							
#k (1..8)	ROL							
---			ROL	ROL	ROL	ROL	ROL	ROL

Beispiel:                D1 = \$12345678  
                          ROL #4,D1  
                          D1 = \$23456781        C = '1'

Beispiel 2:             D1 = \$12345678        D2 = \$87654308  
                          ROL D2,D1                                \$87654308 MOD 64d = 8  
                          D1 = \$34567812        D2 = \$87654308        X,C = '0'

### 3.32 ROR – Rotieren rechts eines Datenregisters

Beschreibung: Der Befehl ROR gehört zu den Schiebefunktionen. Es findet eine logische Rotation nach rechts statt. Dies entspricht dem Befehl LSR, jedoch wird nicht eine '0' nachgeschoben, sondern der hinausgeschobene Teil wird links eingeschoben. Wird in einem Datenregister rotiert, kann die Anzahl der Rotierbewegungen angegeben werden. Findet das Rotieren in einer Speicherzelle statt, wird immer nur um eine Stelle rotiert. Wenn die Anzahl der Rotationen mit einer Konstanten angegeben werden, können nur 1..8 Bit rotiert werden pro Befehl, wird die Angabe mit einem Datenregister gemacht, kann 0..63 mal rotiert werden. Werden grössere Zahlen angegeben, wird eine Modulo-Division mit 64 gemacht. Das rotierte, bzw. zuletzt rotierte Bit ganz rechts wird in das C-Flag übertragen.



Assembler Syntax: ROR.s Dx,Dy (Dx = 0..63)  
 ROR.s #k,Dy (k = 1..8)  
 ROR.s <EA>

Operandengrösse: s = B,W,L (Byte, Word und Long)

Beeinflusste Flags: X keine Änderung  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V immer '0'  
 C gesetzt gemäss dem letzten rotierten Bit ganz links

Erlaubte Adressierungsarten:

Quell - EA	Ziel – EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
Dn	ROR							
#k (1..8)	ROR							
---			ROR	ROR	ROR	ROR	ROR	ROR

Beispiel: D1 = \$12345678  
 ROR #4,D1  
 D1 = \$81234567 C = '1'

Beispiel 2: D1 = \$12345678 D2 = \$87654308  
 ROR D2,D1 \$87654308 MOD 64d = 8  
 D1 = \$78123456 D2 = \$87654308 C = '0'

### 3.33 RTE – Beenden eines Ausnahmezustandes (Interrupt, Softwarefehler)

Beschreibung: Der Befehl RTE beendet einen Interrupt oder eine andere Exception. Die Rücksprungadresse wird vom Stack geholt, ebenso werden die Flags wieder auf den Zustand vor der Exception gebracht. Der Befehl sollte keinesfalls mit RTS verwechselt werden!

Assembler Syntax: RTS

Operandengrösse: Keine, da keine Operanden folgen

Beeinflusste Flags:	X	Wert vor der Exception
	N	Wert vor der Exception
	Z	Wert vor der Exception
	V	Wert vor der Exception
	C	Wert vor der Exception

### 3.34 RTS – Beenden eines Subprogramms

**Beschreibung:** Der Befehl RTS beendet ein Subprogramm. Das Programm wird die Arbeit nach dem aufrufenden JSR fortsetzen. Die Rücksprungadresse wird vom Stack genommen und in den PC geladen, die Flags werden nicht geändert.

**Assembler Syntax:** RTS

**Operandengrösse:** Keine, da keine Operanden folgen

Beeinflusste Flags:	X	keine Änderung
	N	keine Änderung
	Z	keine Änderung
	V	keine Änderung
	C	keine Änderung

### 3.35 SUB (SUB, SUBA, SUBI, SUBQ) – Subtrahieren

**Beschreibung:** Der Befehl SUB dient zum Subtrahieren von zwei Operanden, wobei mindestens einer davon in einem Register (oder eine Konstante) sein muss. Der Befehl SUB wird benutzt, wenn Quelle oder Ziel ein Datenregister ist. Ist das Ziel ein Adressregister, muss SUBA benutzt werden, bei einer Konstantensubtraktion SUBI. Der Assembler erkennt normalerweise die richtige Schreibweise selbst, so genügt für alle drei Fälle ein SUB.

Der Befehl SUBQ wird für Konstanten von 1 – 8 benutzt. Er ist schneller und braucht weniger Speicherplatz als SUBI.

**Assembler Syntax:** SUB.s <EA>,Dn  
 SUB.s Dn,<EA>  
 SUBA.s <EA>,An  
 SUBI.s #kkkkkkkk,<EA>  
 SUBQ.s #k,<EA> (k=1..8)

**Operandengrösse:** s = B,W,L (Byte, Word und Long), bei Adressregistern nur s = W,L (Word, Long)

Beeinflusste Flags:	X	gleich wie C
	N	'1' wenn Resultat negativ, sonst '0'
	Z	'1' wenn Resultat = 0, sonst '0'
	V	'1' wenn des Resultat einen Überlauf verursacht hat, sonst '0'
	C	'1' wenn ein Übertrag (Carry) stattgefunden hat, sonst '0'

Erlaubte Adressierungsarten:

Quell - EA	Ziel – EA							
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
Dn	SUB	SUBA	SUB	SUB	SUB	SUB	SUB	SUB
An	SUB	SUBA						
(An)	SUB	SUBA						
(An)+	SUB	SUBA						
-(An)	SUB	SUBA						
(d16,An)	SUB	SUBA						
(d8,An,Rn)	SUB	SUBA						
\$xxxxxxxx	SUB	SUBA						
#xxxxxxxx	SUBI	SUBA	SUBI	SUBI	SUBI	SUBI	SUBI	SUBI
#x (1-8)	SUBQ	SUBQ	SUBQ	SUBQ	SUBQ	SUBQ	SUBQ	SUBQ

Beispiel:            D1 = \$12345678  
                       SUBI.W #\$4321,D1  
                       D1 = \$12341357

Beispiel 2:         D1 = \$12345678         D2 = \$87654321  
                       SUB.B D2,D1  
                       D1 = \$12345657         D2 = \$87654321

### 3.36 SWAP – Tauschen von Low-Word und High-Word eines Datenregisters

Beschreibung:        Mit SWAP können Low-Word und High-Word eines Datenregister ausgetauscht werden. So kann beispielsweise nach einer Division der Rest geholt werden.

Assembler Syntax:    SWAP Dn

Operandengrösse:    keine, da nur W (Word) möglich

Beeinflusste Flags:    X        keine Änderung  
                               N        '1' wenn das 32-Bit Resultat negativ, sonst '0'  
                               Z        '1' wenn das 32-Bit Resultat = 0, sonst '0'  
                               V        immer '0'  
                               C        immer '0'

Beispiel:             D1 = \$12345678  
                           SWAP D1  
                           D1 = \$56781234

### 3.37 TST – Prüfen einer EA (Register, Speicherzelle) und setzen der Flags

Beschreibung:        Der Befehl TST führt ein Vergleich eines Operanden mit 0 durch, der Befehl entspricht einem CMP #0,<EA>. Es werden die entsprechenden Flags gesetzt. Der Operand wird bei diesem Befehl nicht geändert! Nach dem Testen kann im Programm mit einem bedingten Sprung (Bcc) entsprechend verzweigt werden.

Assembler Syntax:    TST.s <EA>

Operandengrösse:    s = B,W,L (Byte, Word und Long), bei Adressregistern nur s = W,L (Word, Long)

Beeinflusste Flags: X keine Änderung  
 N '1' wenn Resultat negativ, sonst '0'  
 Z '1' wenn Resultat = 0, sonst '0'  
 V '1' wenn des Resultat einen Überlauf verursacht hat, sonst '0'  
 C '1' wenn ein Übertrag (Carry) stattgefunden hat, sonst '0'

Erlaubte Adressierungsarten:

	Ziel – EA							
Quell - EA	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Rn)	\$xxxxxxxx
---	TST	TST	TST	TST	TST	TST	TST	TST

Beispiel: D1 = \$12345678  
 TST.W D1  
 D1 = \$12345678 X=0 N=0 Z=0 V=0 C=0

Beispiel 2: D1 = \$12345687  
 TST.B D1  
 D1 = \$12345687 X=0 N=1 Z=0 V=0 C=0

## 4 Verzeichnis nach Funktionen

Nachfolgend sind alle hier vorgestellten Befehle nach Funktion geordnet, um ein einfaches Auffinden zu ermöglichen.

### 4.1 Registerfunktionen

Laden von Registern	MOVE	Kapitel 3.22 Seite 18
Speichern von Registern	MOVE	Kapitel 3.22 Seite 18
Laden v. Konstanten +/-128	MOVEQ	Kapitel 3.22 Seite 18
Stackbefehle für Register	MOVEM	Kapitel 3.23 Seite 19
Löschen von Datenregistern	CLR	Kapitel 3.10 Seite 9
Löschen von Speicherzellen	CLR	Kapitel 3.10 Seite 9
Tauschen von zwei Registern	EXG	Kapitel 3.16 Seite 14
Wechseln Low/High-Word	SWAP	Kapitel 3.36 Seite 27
Berechnen der EA	LEA	Kapitel 3.19 Seite 15
Ablegen der EA auf Stack	PEA	Kapitel 3.30 Seite 23

### 4.2 Arithmetische Funktionen

Addieren

– Normale Addition	ADD	Kapitel 3.1 Seite 4
– Konstanten 1..8	ADDQ	Kapitel 3.1 Seite 4

Subtrahieren

– Normale Subtraktion	SUB	Kapitel 3.35 Seite 26
– Konstanten 1..8	SUBQ	Kapitel 3.35 Seite 26

Multiplizieren

– mit Vorzeichen	MULS	Kapitel 3.24 Seite 20
– ohne Vorzeichen	MULU	Kapitel 3.25 Seite 20

Dividieren

– mit Vorzeichen	DIVS	Kapitel 3.13 Seite 11
– ohne Vorzeichen	DIVU	Kapitel 3.14 Seite 12

Vorzeichenwechsel +/-NEG Kapitel 3.26 Seite 21

Logisches AND AND Kapitel 3.2 Seite 5

Logisches OR OR Kapitel 3.29 Seite 22

Logisches XOR (EOR)EOR Kapitel 3.15 Seite 13

Logisches NOT NOT Kapitel 3.28 Seite 22

### 4.3 Sprungfunktionen

Absoluter Sprung	JMP	Kapitel 3.17 Seite 14
Relativer Sprung	BRA	Kapitel 3.5 Seite 7
Absoluter Subroutinenaufruf	JSR	Kapitel 3.18 Seite 15
Relativer Subroutinenaufruf	BSR	Kapitel 3.8 Seite 8
Subroutinenende	RTS	Kapitel 3.34 Seite 26
Bedingter Sprung	Bcc	Kapitel 3.3 Seite 5
Schlaufenfunktion	DBcc	Kapitel 3.12 Seite 11

### 4.4 Schiebe- und Rotierfunktionen

Linksschieben	LSL	Kapitel 3.20 Seite 16
Rechtsschieben	LSR	Kapitel 3.21 Seite 17
Linksrotieren	ROL	Kapitel 3.31 Seite 24
Rechtsrotieren	ROR	Kapitel 3.32 Seite 25

### 4.5 Bit – Manipulierfunktionen

Setzen eines Bits	BSET	Kapitel 3.6 Seite 7
Löschen eines Bits	BCLR	Kapitel 3.4 Seite 6
Testen eines Bits	BTST	Kapitel 3.9 Seite 9

### 4.6 Vergleichsfunktionen und andere Befehle

Vergleichen von Registern	CMP	Kapitel 3.11 Seite 10
Vergleiche mit Speicher	CMP	Kapitel 3.11 Seite 10
Testen von Registern	TST	Kapitel 3.37 Seite 27
Testen von Speicherzellen	TST	Kapitel 3.37 Seite 27
Beenden eines Interrupts	RTE	Kapitel 3.33 Seite 25
Keine Operation	NOP	Kapitel 3.27 Seite 22
Debugging Breakpoint	BGND	Kapitel 3.7 Seite 8

## 5 Literaturverzeichnis

- [1] Motorola: CPU32RM/AD 683xx CPU32 Reference Manual
- [2] Werner Hilf: MC68000 Grundlagen. Franzis Verlag, München
- [3] Werner Hilf: MC68000 Anwendungen. Franzis Verlag, München
- [4] Josef Fuchs: MC68300 Mikrocontroller. Franzis Verlag, München
- [5] Jann Oesch: Mikroprozessorenkurs, OESCH.ORG, Schönbühl und Leipzig
- [6] Motorola Microcontroller Web: <http://mot-sps.com>