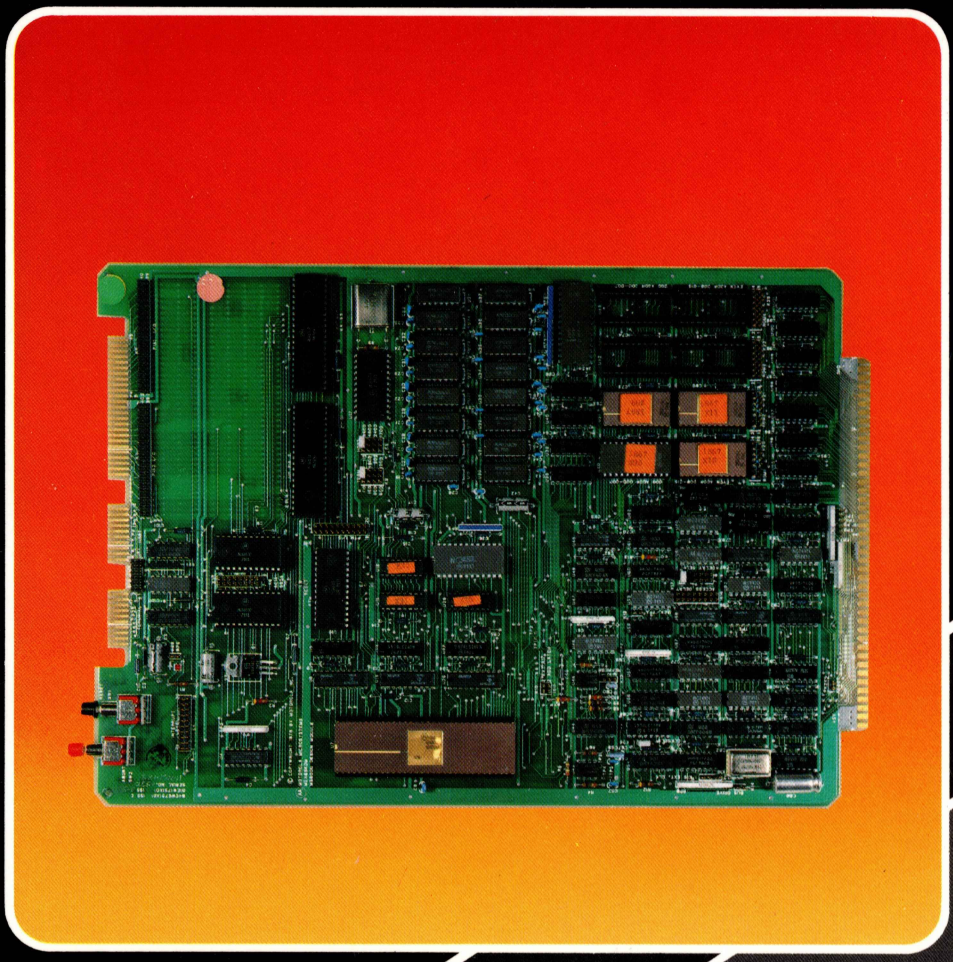
Leo J. Scanlon

Die 68'000er

Grundlagen und Programmierung



AT Verlag

Leo J. Scanion
ist Documentation Manager für den Bereich Mikroelektronik der Rockwell International in Anaheim, CA. Er ist Autor verschiedener Artikel über Mikrocomputer und Programm-Engineering in der Weltraumindustrie sowie zweier weiterer in den USA erschienener Bücher.

Leo J. Scanlon

Die 68 000er

Grundlagen und Programmierung

AT Verlag Aarau · Stuttgart

Sonderdruck aus der internationalen Fachzeitschrift
für praktische Elektronik «Elektroniker»

Die Originalausgabe erschien unter dem Titel
«The 68 000: Principals and Programming»
bei Howard W. Sams & Co., Inc. Indianapolis, Indiana 46268, USA
ISBN 0-672-21853-4

Übersetzer: Hans Iseli
Umschlag: AT-Grafik, Aarau
Herstellung: Grafische Betriebe Aargauer Tagblatt AG,
Aarau (Schweiz)
Printed in Switzerland

ISBN 3-85502-152-X

Inhaltsverzeichnis

1. Einführung in den Mikroprozessor MC 68000

1.1 Überblick

Der MC 68000 verfügt über 17 allgemein verwendbare Register, jedes 32 Bit lang, über einen 32-Bit-Programmzähler und ein 16-Bit-Statusregister. Acht der allgemeinen Register werden verwendet als Datenregister für Byte-(8 Bit), Wort-(16-Bit-) und Doppelwort-(32-Bit-)Operationen. Die andern 9 allgemeinen Register sind Adressregister, die als Stapelzeiger und Basisadressregister verwendet werden. Alle 17 allgemeinen Register können auch als Indexregister verwendet werden.
Obwohl der Programmzähler 32 Bit lang ist, werden nur die tieferwertigen 24 Bit verwendet. Diese 24 Bit geben dem MC 68000 einen Adressbereich von 16 MByte – der gleiche Adressbereich wie ein IBM System/370! Dieser Adressbereich erlaubt es, zusammen mit einem zusätzlichen Speicherverwaltungsbaustein, grosse modulare Programme zu entwickeln und auszuführen, ohne komplizierte und schwierige softwaremässige Speicherverwaltungsmassnahmen.

1.1.1 Softwaremöglichkeiten

Die Softwaremöglichkeiten des MC 68000 sind recht eindrücklich und zeigen, dass dieser Mikroprozessor *von Programmierern für Programmierer entwickelt* wurde. Wie später in Kapitel 3 gezeigt wird, bieten viele der Befehle in Kombination mit den vielseitigen Adressierungsmodi fast den Komfort und die Möglichkeit höherer Sprachen.
Der MC 68000 kann mit 5 verschiedenen Datenarten Verarbeitungen durchführen:
- 1 Bit
- 4 Bit (BCD-Werte)
- 8 Bit (Byte)
- 16 Bit (Worte)
- 32 Bit (Doppelworte)

Eine Byteadressierung ist möglich, indem das höherwertige Byte dieselbe gerade Adresse wie das entsprechende Wort hat, während das niederwertige Byte die um 1 erhöhte ungerade Adresse trägt. Das Befehlsrepertoire umfasst 56 Grundbefehle.

Dazu stehen 14 verschiedene Adressierungsmodi zur Verfügung. Die Kombination der 56 Grundbefehle mit den 14 Adressierungsmodi und den 5 Datenarten ergibt mehr als 1000 verschiedene Kombinationen, die der MC 68000 ausführen kann. Zusätzlich gibt es zwei nicht benützte Operationscodes, die nach den Wünschen des Benützers spezifiziert werden können.
Der MC 68000 wird angeboten in 4-, 6-, 8- und 10-MHz-Versionen, denen Taktperioden von 250, 167, 125 und 100 ns entsprechen.
Der schnellste Befehl, zum Beispiel für das Kopieren des Inhalts eines Registers in ein anderes Register, benötigt als Ausführungszeit 4 Taktzyklen oder 500 ns bei 8 MHz.
Der langsamste Befehl, eine Division eines 32-Bit-Doppelwortes durch ein 16-Bit-Wort mit Vorzeichen, kann bis zu 170 Taktzyklen beanspruchen oder 21,25 µs bei 8 MHz.

1.1.2 Privilegierte Zustände

Zur Unterstützung von Systemen mit mehreren Benützern verfügt der MC 68000 über zwei verschiedene Zustände: einen Benützerzustand für normale Funktionen und einen Überwachungszustand für die Systemkontrolle. Im Überwachungszustand können alle Befehle ausgeführt werden, im Benützerzustand können einige privilegierte Befehle, zum Beispiel Reset und Stop nicht benützt werden. Diese Möglichkeiten geben dem System eine gewisse Sicherheit, da der Datenzugriff kontrolliert ist und dadurch die gegenseitige Beeinflussung von Daten verschiedener Benützer verhindert wird.

1.1.3 Eingebaute Fehlersuchhilfsmittel

In Anbetracht der Tatsache, dass das Beheben von Fehlern in der SW im allgemeinen mehr Zeit in Anspruch nimmt als das Schreiben der SW selbst, haben die Entwickler des MC 68000 eine ganze Anzahl von Fehlersuchhilfsmitteln eingebaut. Als Beispiel führen

- illegale Befehle,
- die Verletzung von privilegierten Zuständen,
- fehlerhafte Adressierung,
- Division durch Null,
- illegale Speicherzugriffe usw.

den Mikroprozessor in den Überwachungszustand.

Der MC 68000 verfügt auch über einen sogenannten Tracemodus für die Fehlerbehebung in der SW. In diesem Modus verarbeitet der MC 68000 Befehle Schritt für Schritt, indem nach der Ausführung jedes einzelnen Befehls in eine Serviceroutine verzweigt wird.

1.1.4 Speicherzuweisung

Sehr wenige Speicherzellen sind für spezielle Aufgaben fest zugewiesen. Die tiefsten 8 Byte des Speichers enthalten den Rücksetzvektor und sind demzufolge als ROM ausgeführt. Zusätzliche Speicherzellen in den ersten 1024 Bit sind Unterbruchsvektoren, Fehlervektoren sowie Vektoren für verschiedene andere Arten von Ausnahmezuständen zugewiesen. Diese Speicherbereiche können entweder als ROM oder als Lese- und Schreibspeicher ausgeführt sein. Der verbleibende Rest des 16-MByte-Speichers des MC 68000 kann für jede beliebige Aufgabe verwendet werden.
Selbstverständlich werden einige Speicheradressen für zugeordnete Ein- und Ausgabebausteine im System verwendet, weil der MC 68000, wie übrigens alle Motorola-Mikroprozessoren, die Ein- und Ausgabeoperationen über den Speicher ausführt. Der MC 68000 verfügt über keine separaten Ein- und Ausgabebefehle, «sieht» aber periphere Bausteine als Speicherzellen im 16-MByte-Speicher. Bei der Programmierung von Ein- und Ausgabeoperationen für den Datentransfer von und zu peripheren Geräten werden die gleichen Befehle verwendet wie für den Datenaustausch mit dem Speicher.

1.1.5 Unterbruchstruktur (Interrupts)

Die Unterbruchstruktur des MC 68000 ist ähnlich wie die der meisten Minicomputer. Es stehen 7 verschiedene Unterbruchsebenen zu Verfügung. Mit einer Maske im Statusregister können Unterbrüche auf derselben oder auf tieferen Ebenen als der in Betrieb stehenden blockiert werden.
Wenn der MC 68000 eine Unterbruchsanforderung erhält, sendet er ein Quittierungsanforderungssignal zu allen im System vorhandenen Bausteinen. Nach dem Empfang der Quittung muss der unterbrechende Baustein eine Vektornummer auf den Datenbus einspeisen. Dieser Vektor wählt eine der 192 Unterbruchroutinen im Speicher aus. Auch Bausteine, die keine Vektornummer erzeugen können, haben die Möglichkeit, den MC 68000 zu unterbrechen. Sie veranlassen den Mikroprozessor mittels «Autovektor», zu einer Subroutine zu verzweigen, die der Unterbruchsebene des unterbrochenen Bausteins zugeordnet ist. Der MC 68000 verfügt über 7 Autovektoren.

1.1.6 Bausteinanschlüsse

Der MC 68000 wird in einem Dual-in-line-Gehäuse mit 64 Anschlüssen geliefert (entspricht ungefähr der Grösse eines normalen Feuerzeuges).
Die Adressen für Befehle und Daten werden über ein System von 25 Adressleitungen zugeführt: ein 23 Leitungen umfassender Adressbus (mit dem ein Wort im Speicher angewählt wird)

und 2 Byte-Select-Leitungen (eine zum Anwählen des tieferwertigen Byte des Wortes, das andere zum Anwählen des höherwertigen Byte des Wortes). Daten werden über einen 16-Bit-Datenbus transferiert. Wie die meisten 8-Bit-Mikroprozessoren (aber nicht wie die 16-Bit-Systeme Intel 8086 und Zilog Z8000) wird der Datenbus und der Adressbus über separate Leitungen geführt.
Die Entwickler bei Motorola stellten fest, dass eine Multiplexierung dieser Bussysteme zwar zu einem kleineren Gehäuse geführt hätte, jedoch auch eine Verkleinerung der Leistung um mehr als dreissig Prozent zur Folge gehabt hätte. Der MC 68000 kann sowohl mit asynchronen peripheren Geräten wie auch mit langsameren synchronen peripheren Geräten (wie sie auch für den MC 6800 und andere 8-Bit-Mikroprozessoren verwendet werden) verbunden werden. Er verfügt über separate Kontrolleitungen für jeden Typ von peripheren Schaltungen. Der MC 68000 benützt eine einzige Speisespannung von +5 V und verfügt über je zwei Plus-/Minusanschlüsse. Ein Anschluss ist für den TTL-kompatiblen Takteingang vorgesehen.
Eingeführt wurde der MC 68000 im Jahre 1979. Er ist erhältlich von Motorola (als MC 68000) und als Lizenzfabrikate von Rockwell International (R 68000), Hitachi (HD 68000), Mostek (MK 68000) und Signetics (SP 68000). In Europa wird der 68000 auch von Efcis hergestellt, einer Firma im Besitz von Thomson-CSF und der französischen Atomenergiekommission.

1.2 Interne Register

Da wir uns vor allem mit der Programmierung des MC 68000 befassen, sind für den Einstieg zuerst die internen Register, die zur Verfügung der Programmierer stehen, interessant. Bild 1.1 zeigt die 17 allgemein verwendbaren Register, den 32-Bit-Programmzähler und das 16-Bit-Statusregister des MC 68000.

1.2.1 Allgemein verwendbare Register

Acht der allgemein verwendbaren Register sind Datenregister, 7 sind Adressregister und 2 sind Stapelzeigerregister (eines für Benützerprogramme, das andere für Überwachungsprogramme). Die acht Datenregister (D0 ... D7) können verwendet werden für Operationen mit Byte, Wort und Doppelwort. Die verwendete Datenlänge wird spezifiziert durch einen Datenlängencode im Befehl. Byteoperationen werden immer mit den tieferwertigen 8 Bit eines Datenregisters (Bit 0 ... 7) durchgeführt; Wortoperationen werden immer mit den tieferwertigen 16 Bit eines Datenregisters (Bit 0 ... 15) durchgeführt, wie in Bild 1.1 durch die gestrichelte Linie angedeutet ist. Wenn ein Byte- oder

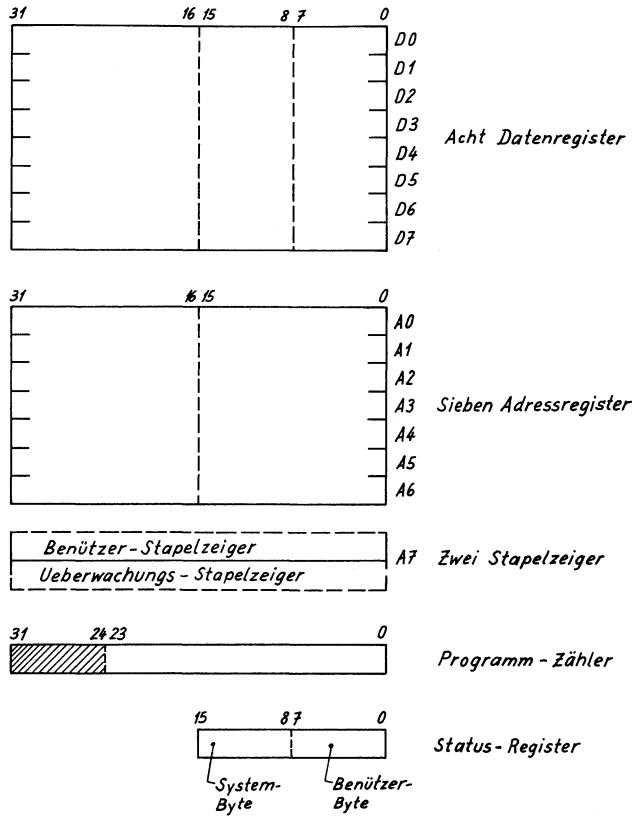
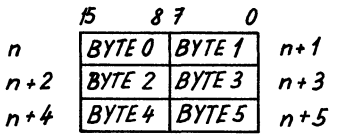
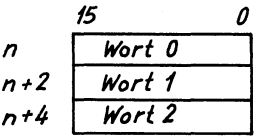


Bild 1.1
Registeranordnung im Mikroprozessor MC 68000 (Programmierungsmodell)

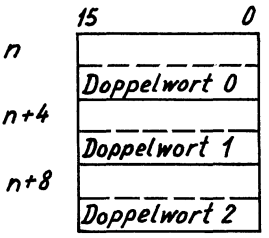
Wortoperand in einem Befehl vorkommt, wird immer das tieferwertige Byte oder Wort des Datenregisters verwendet. Die verbleibende Information im Register wird nicht berührt.
Die sieben Adressregister (A0 ... A6) dienen als Basisadressregister und als Softwarezeiger zu benützerdefinierten Speicherbereichen. Sie können auch zur temporären Aufnahme von Adresswerten verwendet werden, so dass diese Adressen irgendwo im Programm nicht wieder neu berechnet werden müssen. Die Adressregister können verwendet werden für den Zugriff zu Bytes, Worten und Doppelworten im Speicher. Wie in Bild 1.2 gezeigt, werden diese Daten in der Ordnung höher zu tiefer gespeichert, das heisst, dass Byte 0, Wort 0 und Doppelwort 0 die höchste Wertigkeit aufweisen.
Byte können gerade Adressen (Byte 0, 2 und 4 in Bild 1.2) oder ungerade Adressen (Byte 1, 3 und 5) haben, Worte und Doppelworte können nur gerade Adressen haben. Das bedeutet also, dass Worte und Doppelworte immer mit einer geraden Adresse beginnen müssen. Wenn also ein Wort sich an der Adresse N, wobei N gerade ist, befindet, ist das nächste Wort an der Adres-



a) Byteformat



b) Wortformat



c) Doppelwortformat

Bild 1.2
Datenformate im Speicher:
a) Byteformat (8 Bit)
b) Wortformat (16 Bit)
c) Doppelwortformat (32 Bit)

se N + 2. Ähnlich ist es bei Doppelworten. Wenn ein Doppelwort sich an der Adresse N, wiederum gerade, befindet, ist das nächste Doppelwort an der Adresse N + 4. Die gestrichelte Linie zwischen Bit 15 und Bit 16 im Bild 1.1 zeigt an, dass sich die Information in einem Adressregister auf ein 16-Bit-Wort (in Bit 0 ... 15) oder auf ein 32-Bit-Doppelwort beziehen kann. Viele Befehle des MC 68000 beziehen sich auf zwei Operanden, einen Quellenoperanden und einen Bestimmungsoperanden. Wenn ein Adressregister verwendet wird als Quellenoperand, wird entweder das tieferwertige Wort oder das ganze Doppelwort verwendet, je nach Abhängigkeit der Operationslänge. Wird ein Adressregister als Bestimmungsoperand verwendet, so wird das ganze Register beeinflusst, unabhängig der Operationslänge. Operationen mit Adressregistern beeinflussen das Statusregister des MC 68000 nicht. Diese Tatsache erlaubt es, innerhalb von Datenoperationen Adressen zu ändern, ohne sich Gedanken machen zu müssen, ob der Programmstatus geändert haben könnte.

Der MC 68000 verfügt über zwei Stapelzeiger, wobei zu einer bestimmten Zeit nur einer aktiv sein kann. Der Benützer-Stapelzeiger, der zur Sicherung der Rückkehradresse während Subroutinenaufrufen benützt wird, ist aktiv, wenn der Prozessor im Benützerstatus arbeitet. Der Überwachungsstatus-Stapelzeiger, der die Rückkehradresse und den Statusregisterinhalt während Trap- und Unterbruchroutinen aufnimmt, ist aktiv, wenn der MC 68000 sich im Überwachungsstatus befindet. Weil die beiden Stapelzeiger nicht gleichzeitig aktiv sein können, sind sie in Bild 1.1 als ein einziger Bestimmungsort A7 dargestellt. Jedes der 17 allgemein verwendbaren Register kann auch als Indexregi-

ster verwendet werden. Die Indexierung wird behandelt im Zusammenhang mit der Diskussion der Adressierungsmodi in Kapitel 3.

1.2.2 Programmzähler

Wie alle Mikroprozessoren, führt auch der MC 68000 Programme aus, indem er einen Befehl vom Speicher holt, ihn ausführt und dann den nächsten Befehl holt. Beim MC 68000 belegt ein Befehl 1 bis 5 Wörter im Speicher, wobei der Programmzähler bestimmt, welches Befehlswort als nächstes geholt wird. Der Programmzähler ist 32 Bit lang. In den bisher produzierten Bausteinen wurden jedoch nur die 24 tieferwertigen Bit verwendet. Weil die Befehle aus Worten bestehen, enthält der Programmzähler immer eine gerade Adresse. Mit den 24 Bit des Programmzählers können 8 M-Worte adressiert werden (8 388 608 Worte, Adressenbereich 0 ... hexadezimal FFFFFE).

1.2.3 Statusregister

Das Statusregister des MC 68000 ist aufgeteilt in ein Benützerbyte und ein Systembyte wie in Bild 1.3 dargestellt. Gelesen werden kann der gesamte Inhalt des Statusregisters jederzeit, hingegen kann das Systembyte nur im Überwachungsmodus geändert werden. Das Benützerbyte, oft auch Bedingungscoderegister genannt, enthält fünf Flagbit, die Information über ausgeführte Befehle enthalten.

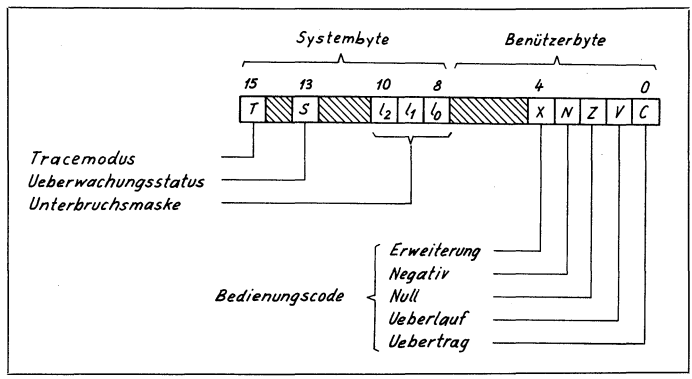


Bild 1.3
Das Statusregister (Flags) des MC 68 000

Die fünf Flagbit im Benützerbyte bedeuten im einzelnen:

- *Bit 0, Übertrag (Carry C):*

Dieses Bit wird auf 1 gesetzt, wenn bei einer Addition ein Übertrag entsteht oder bei einer Subtraktion ein Entlehnwert benötigt wird, andernfalls ist es 0.

Das Übertragsbit wird ebenfalls für die Aufnahme des Werts eines Bit verwendet, das aus einem Datenregister oder Speicherplatz geschoben oder rotiert wurde, und es enthält auch das Resultat einer Vergleichsoperation.

- *Bit 1, Überlauf (Overflow V):*

Dieses Bit hat nur eine Bedeutung während Operationen mit vorzeichenbehafteten Zahlen. Es wird auf 1 gesetzt, wenn die Addition von zwei Werten mit gleichen Vorzeichen oder die Subtraktion von zwei Werten mit ungleichen Vorzeichen ein Resultat ergeben, das den Bereich des Zweierkomplements des Operanden überschreitet, andernfalls ist es 0.
Es wird ebenfalls gesetzt, wenn das höchstwertige Bit des Operanden zu irgendeinem Zeitpunkt während einer arithmetischen Schiebeoperation ändert.

- *Bit 2, Null (Zero Z):*

Dieses Bit wird auf 1 gesetzt, wenn das Resultat einer Operation 0 ist.

- *Bit 3, Negativ (Negative N):*

Dieses Bit hat nur eine Bedeutung bei Operationen mit vorzeichenbehafteten Werten. Es wird dann auf 1 gesetzt, wenn eine arithmetische, logische, Schiebe- oder Rotieroperation zu einem negativen Resultat führt. Mit andern Worten gesagt, folgt das N-Bit dem höchstwertigen Bit des Operanden, unabhängig davon, ob er 8, 16 oder 32 Bit lang ist.

- *Bit 4, Erweiterung (X):*

Dieses Bit funktioniert als Überlaufbit für Operationen mit erhöhter Genauigkeit. Es wird durch Additions-, Subtraktions-, Negier-, Schiebe- und Rotierbefehle beeinflusst, indem es während deren Ausführung den Status des Übertragsbit (C) annimmt.

Der MC 68000 verfügt über bedingte Verzweigbefehle, die den Zustand der Bit C, V, Z, N prüfen und je nach Resultat das Programm weiterlaufen oder eine bestimmte Adresse anspringen lassen. Die Bedingungscodebit werden immer dann beeinflusst, wenn Operationen den Inhalt von Datenregistern ändern, aber nie bei Operationen mit Adressregistern.

Das Systembyte des Statusregisters besteht aus drei Feldern:

- *Bit 8 bis 10:*

Diese Bit enthalten eine Unterbruchsmaske (I0, I1 und I2), mit der die Prioritätsebene der Unterbruchsanforderungen bestimmt werden kann. Diese 3-Bit-Maske kann zur Festsetzung von einer aus sieben Prioritätsebenen verwendet werden (die achte Ebene, alles 0, bedeutet, dass jede Priorität akzeptiert

wird) und veranlasst den MC 68000, alle Unterbrechungsanforderungen auf oder unter dieser Prioritätsebene nicht zu beachten.

– *Bit 13, Überwachung (S):*

Dieses Bit zeigt an, ob der MC 68000 sich im Überwachungszustand (S = 1), oder im Benützerzustand (S = 0) befindet.

– *Bit 15, Tracemodus (T):*

Dieses Bit steuert die eingebaute Fehlersuchschaltung. Wenn das T-Bit auf 1 gesetzt ist, arbeitet der MC 68000 ein Programm Schritt für Schritt ab. Das bedeutet, dass der Prozessor nach jedem ausgeführten Befehl in den Überwachungszustand (S=1) übergeführt wird und zu einem durch den Anwender geschriebenen Trace-Unterprogramm verzweigt. Dieses Trace-Unterprogramm kann zum Beispiel verwendet werden zur Prüfung des Inhalts von ausgewählten Registern oder Speicherplätzen, zur Statusprüfung oder zur Durchführung irgendwelcher anderer Fehlersuchaufgaben.

Die nichtbenützten Bit des Statusregisters werden immer als 0 gelesen.

1.3 Entwurfsphilosophie

Mit den bisherigen Ausführungen haben wir einen allgemeinen Überblick über die Möglichkeiten des Mikroprozessors MC 68000 gewonnen. Die weiteren Kapitel werden diese Information bis ins Detail vertiefen und einen Überblick geben über die Anwendungsmöglichkeiten des MC 68000. Bevor wir jedoch auf die Details eintreten, soll noch einige Information zur Entwurfsphilosophie des MC 68000 vermittelt werden.

1.3.1 Der Stand der Mikroprozessortechnologie

Die leistungsfähigen Mikroprozessoren und die zugehörigen Bausteine, die heute zur Verfügung stehen, sind der Ausdruck einer enormen Entwicklung der Technologie der integrierten Schaltungen in der vergangenen Zeit. Seit der Entwicklung der MOS-Halbleiter in den späten fünfziger Jahren verdoppelte sich die Komplexität der Schaltungen in den siebziger Jahren jedes Jahr. Während frühere Mikroprozessoren 5000 bis 10000 Transistorfunktionen pro Baustein aufwiesen, verfügen heutige Prozessoren über 100000 Transistorfunktionen. Primäre Faktoren für diese Entwicklung sind eine höhere Dichte der Schaltungen und die Fortschritte im Schaltungsentwurf, die generell zu höheren Geschwindigkeiten und zu geringerem Leistungsverbrauch führen. Die Entwicklungsrate hat sich etwas verlangsamt durch gewisse technologische Grenzen, die Fortschritte sind jedoch immer noch enorm. Gegenwärtig wird die

Schaltungsdichte und die Schaltungsgeschwindigkeit alle zwei Jahre verdoppelt, während in der gleichen Zeit das Leistungs-Geschwindigkeits-Produkt um den Faktor 4 gesunken ist. Gleichzeitig sanken die Produktionskosten, was sich in einem reduzierten Produktpreis auswirkte, was wiederum zu erhöhtem Bedarf, neuen Anwendungen und neuen Märkten führt.

1.3.2 Begründung für die Entwicklung des MC 68000

Die eben beschriebenen Fortschritte machten einen komplexen Mikroprozessor technisch möglich, dazu kamen zusätzliche Faktoren als Motivation für Motorola, die zur Entwicklung des MC 68000 führten. Nach Edward Stritter und Tom Gunter, zwei der Hauptverantwortlichen für die Entwicklung des MC 68000, leitet sich eine der Motivationen her aus dem Bedarf für Produkte, die über die vielfältigen Möglichkeiten eingebauter Mikroprozessoren verfügen. Dieser Bedarf zeigt sich im allgemeinen Markt für Mikroprozessoren, der jährliche Zuwachsraten von 25% aufweist und zu einem jährlichen Volumen von 200 Millionen Einheiten im Jahre 1983 führen dürfte, mit einem Marktwert von ungefähr 1 Milliarde Franken. (Schätzungen lauten dahin, dass bis ins Jahr 2000 5 bis 10 Milliarden Mikroprozessoren und Mikrocomputer in Betrieb stehen werden, das heisst ungefähr 1 System pro dannzumal auf der Erde lebende Person!) Beim Entwurf des MC 68000 waren sich die Entwickler im klaren, dass ihr Produkt auf Anwendungen zugeschnitten sein musste, für die 16-Bit-Mikroprozessoren im Vordergrund stehen, wie zum Beispiel Anwendungen mit Multiprocessing und Multitasking. – Eine zweite Motivation für die Entwicklung des MC 68000 kam von der Seite der hohen Kosten für die Software-Entwicklung. Mit gegenwärtigen Kosten von 20 bis 40 Franken für jede Zeile getesteten Codes ist es nicht unüblich, für ein einfaches Programm auf Software-Entwicklungskosten von 200 000 Franken oder mehr zu kommen, was in keinem vergleichbaren Massstab mit den Hardwarekosten von einigen hundert bis allenfalls einigen tausend Franken steht. Als Gegenmassnahme zu dieser Entwicklung fördert Motorola konsequent die Unterstützung von höheren Programmiersprachen und ein klar strukturiertes Vorgehen bei der Programmierung. Zudem wird versucht, mit der 68000-Software Fehlersuche und Selbstprüfung so einfach wie möglich zu machen. – Ein dritter die Entwicklung des MC 68000 beeinflussender Faktor waren die hohen Kosten des Entwurfs und der Fabrikation von neuen Mikroprozessoren. Sowohl die Personalkosten wie auch die Kosten für Entwurf und Fabrikation von Ausrüstungen sind enorm und erreichen für die wichtigsten Hersteller Millionen von Franken pro Jahr. Die Entwickler begegnen diesem Problem auf verschiedene Weise. Erstens ist ein geradliniger Entwurf unter Verwendung von optimalen Strukturen leichter zu

realisieren, zu testen und herzustellen. Selbstverständlich führt ein geradliniger Entwurf auch zu einem verbesserten Produktionszyklus und damit zu einer Verbesserung der Wettbewerbsfähigkeit des Herstellers. Zweitens muss eine neue Architektur auf so lange Zeit hinaus wie möglich geplant werden und für die Zukunft einfach zu erweitern sein. Die Hersteller sind nicht mehr in der Lage, jedes Jahr neue Architekturen zu produzieren. Erfahrungen mit dem Versuch zur Erweiterung und Verbesserung früherer 8-Bit-Mikroprozessoren zeigten die Notwendigkeit für verbesserte Planung. Die Entwerfer müssen auf möglichst wenig Beschränkungen in ihren Entwürfen achten, so dass zukünftige Verbesserungen des Bausteins unter den bestmöglichen Voraussetzungen gemacht werden können. Zu den grundsätzlichen Mängeln in der Vergangenheit gehörten ein begrenzter Adressbereich sowie das Fehlen von zum Zeitpunkt des Entwurfs freien Operationscodes für zukünftige Befehle.

1.3.3 Realisierung der Entwurfsideen

Die Entwickler des MC 68000 hatten die nicht leichte Aufgabe, die im vorhergehenden Abschnitt beschriebenen Motive und Begründungen bei der Realisierung des Mikroprozessors zu berücksichtigen. Ihre Wahl fiel auf die schnelle n-Kanal-Silizium-Technologie HMOS (high density, short-channel MOS), die ursprünglich von der Intel Corp. entwickelt wurde. Diese Technologie bietet rund die doppelte Schaltungsdichte sowie das vierfach bessere Geschwindigkeits-Leistungsprodukt gegenüber der Standard-NMOS-Technologie. Als Resultat verfügt die gegenwärtige Version des MC 68000 über etwa 68 000 Transistorfunktionen auf dem Baustein (Bild 1.4).

Im Hinblick auf den potentionellen Anwenderkreis wählten die Entwickler für den MC 68000 eine Architektur in Richtung allgemeine Verwendung und statteten ihn mit einem 16-MByte-Adressbereich aus. Zusätzlich wurden Funktionen wie separate Überwachungs- und Anwenderstati zur Unterstützung von Multiprocessing und Multitasking vorgesehen.

Zur Eindämmung der hohen Kosten von Software-Entwicklungen unternahmen die Entwickler des MC 68000 alle Anstrengungen, die Programmierung so einfach wie möglich zu machen. Ein Weg zur Erreichung dieses Zieles ist eine sogenannte orthogonale Auslegung, das heisst, dass alle Datenregister und alle Adressregister in derselben Weise funktionieren und ebenfalls als Indexregister verwendet werden können. Im weiteren können die meisten Befehle mit Byte, Worten und Doppelworten operieren. Die Anzahl der Mnemocodes im Befehlsrepertoire wurde auf ein Minimum beschränkt, indem Gruppen von ähnlichen Funktionen gebildet wurden.

Diese Auslegung steht im Gegensatz zu einer grossen Anzahl von spezialisierten Lade-, Speicher- und Transferbefehlen, wie

sie oft bei 8-Bit-Mikroprozessoren vorkommen. Im weiteren betrachteten die Entwickler nicht nur die statisch häufigen Befehle (das sind diejenigen, die in einem Listing häufig erscheinen), sondern gingen einen Schritt weiter und hielten Ausschau nach dynamisch häufigen Befehlen, das heisst jenen, die tatsächlich häufig ausgeführt werden. Unter diesem Aspekt wurde versucht, so kurze Befehle wie möglich zu entwickeln. Zur Unterstützung von höheren Sprachen bestehen komplexe Befehle, welche Operationen ausführen, für die normalerweise eine ganze Anzahl Zeilencodes benötigt werden. Ein gutes Beispiel für diesen Fall sind die LINK- und UNLK-Befehle, die im Stapel Platz zuweisen und freigeben für den Aufruf von verschachtelten Unterprogrammen sowie der CHK-Befehl, der erlaubt, die Grenzen einer speziellen Speicheranordnung auf Überlaufbedingungen zu prüfen. Beim Festlegen des Befehlsrepertoires wurde ebenfalls darauf geachtet, dass die meisten Befehle mit allen möglichen Adressierungsmodi verwendet werden können und dadurch Compilern das effiziente Generieren von Code erlauben. Schliesslich wurde im Hinblick auf die Reduktion von Entwicklungskosten für zukünftige Entwicklungsänderungen und Verbesserungen des Mikroprozessors eine Architektur gewählt, die es erlaubt, verschiedene Versionen oder sogenannte Implementierungen zu produzieren. Die gegenwärtige Version unter dem Namen MC 68000 stellt im Prinzip nur ein Subset der kompletten 68000-Architektur dar. Das zeigt sich zum Beispiel darin, dass der MC 68000, obschon er ein 16-Bit-Mikroprozessor ist, über eine interne 32-Bit-Architektur verfügt. Das heisst, dass alle adressierbaren Register, mit Ausnahme des Statusregisters, 32 Bit lang sind. Auch der Programmzähler ist 32 Bit lang, wobei bei den bisher produzierten Bausteinen nur die tieferwertigen 24 Bit herausgeführt wurden. Bei einer allfälligen zukünftigen Version des 68000, die alle 32 Bit zur Verfügung stellt, würde das einen Adressbereich von mehr als 4 Milliarden Byte bedeuten. Selbstverständlich sind auf dem Chip auch der Datenbus und der Adressbus je 32 Bit lang.

2. Cross-Makro-Assembler

2.1 Umfang

Es gibt mittlerweile eine Vielzahl von Systemen, mit denen Software für 68000-Anwendungen entwickelt werden kann. Die einen werden ihre Programme auf Motorolacomputern wie EXORciser^{R}- oder EXORmacs^{R}-Entwicklungssystemen entwickeln, die andern verwenden dafür Mini-, Grosscomputer oder Universalentwicklungssysteme. Ungeachtet dessen, was für ein System benützt wird, nehmen wir an, dass alle Benützer in einer Assemblersprache und nicht in Maschinencode programmieren werden. Somit wird ein Übersetzungsprogramm, der sogenannte Assembler gebraucht, um den Assemblerquellencode in Maschinensprache oder Objektcode zu übersetzen, damit der Mikroprozessor ihn ausführen kann. Wir kennen zwei Assemblergrundtypen.
Ein *Cross-Assembler* läuft auf einem anderen Rechner als demjenigen, der den assemblierten Code ausführen wird. Der Rechner, mit dem assembliert wird, hat normalerweise eine umfangreiche Softwareunterstützung und schnelle Peripheriegeräte, wie zum Beispiel die Systeme IBM 360, 370 oder ein PDP 11.
Als *resident* wird ein Assembler bezeichnet, wenn er auf dem gleichen Rechner läuft wie die Anwendung. Das EXORmacs^{R}-Entwicklungssystem hat zum Beispiel einen residenten Assembler für den MC 68000.
Dieses Kapitel bezieht sich auf den Motorola «Cross Macro Assembler». Der Cross-Makro-Assembler kann auf einem EXORciser^{R}-Entwicklungssystem, auf einem IBM 370 oder auf einem DEC PDP11 laufen. Er ist zudem ein Makro-Assembler, weil er dem Programmierer die Definition von Instruktionssequenzen als Makros erlaubt. Der Begriff Makro wird in diesem Kapitel etwas später exakter behandelt.
Dieses Kapitel soll nicht eine genaue, umfassende Beschreibung des Cross-Makro-Assemblers (im weiteren nur noch Assembler genannt) geben, sondern nur eine Zusammenfassung der grundsätzlichen Eigenschaften sein. Für die speziellen Details wird auf die entsprechende Bedienungsanleitung hingewiesen.

Quellenbefehle

Ein Quellenprogramm ist eine logische Sequenz von Quellenbe-

fehlen, die dazu bestimmt ist, eine spezifische Aufgabe auszuführen. Ein Quellenbefehl kann entweder ein Assemblerbefehl, ein Kommentar oder eine Assembleranweisung sein.

2.2 Assemblerbefehle

2.2.1 Aufbau

Ein Assemblerbefehl besteht aus bis zu fünf Feldern:
Zeilennummer [Label] Mnemonik [Operand] [Kommentar].
Die Zeilennummer wird entweder vom Editor oder vom Assembler generiert, um die Quellencodezeile zu bezeichnen. Die Zeilennummern können bis zu vier Dezimalstellen aufweisen. Die andern vier Felder sind vom Anwender zu programmieren, wobei nur das mnemonische Feld für einen Befehl obligatorisch ist. Label- und Kommentarfeld sind fakultativ (dargestellt durch die eckigen Klammern). Das Operandenfeld wird nur verwendet, falls der Befehl das verlangt, andernfalls wird es weggelassen.
Der MC 68000-Assembler verwendet ein freies Format, in dem die verschiedenen Felder irgendwo auf einer Zeile stehen können. Jedes Feld muss jedoch vom vorhergehenden mindestens durch eine Leerstelle getrennt sein.

2.2.2 Das Labelfeld

Das Labelfeld ist das erste, vom Anwender geschriebene Feld einer Zeile. Jeder Befehl kann ein Label tragen. Es wird meistens nur in Verbindung mit einem Sprung- oder einem Sprung-zur-Subroutine-Befehl gebraucht. Diese Befehle laden den Programmzähler mit einem neuen Wert. Dabei wechselt die sequentielle Ausführung eines Programmes.
Das Label weist auf jenen Befehl, bei dem das Programm weiterfahren soll.
Das Label ist ein Ausdruck von 1 bis 30 alphanumerischen Zeichen, wobei das erste Zeichen ein Buchstabe (A ...Z) sein muss. Alle 30 Zeichen sind signifikant, obwohl nur die ersten 8 ausgedruckt werden. Die Zeichen A0 bis A7, D0 bis D7, CCR, SR, SP und USP werden vom Assembler als Register erkannt und dürfen somit nicht als Label gebraucht werden.
Wenn in der ersten Kolonne ein Label gesetzt wird, so muss danach mindestens eine Leerstelle folgen. Steht das Label in einer anderen Kolonne, so muss direkt danach ein Doppelpunkt (:) stehen.

2.2.3 Das Mnemonikfeld

Das Mnemonikfeld kann Assembler-Befehlsausdrücke von drei bis fünf Buchstaben enthalten. Der Assembler verwendet eine interne Tabelle, um die Befehlsausdrücke (Mnemonik) in Binärcode umzuwandeln.

Einige der Befehle des MC 68000 brauchen einen, beziehungsweise zwei Operanden, andere keinen. Die Mnemoniks geben dem Assembler Anzahl und Typ der Operanden an. Die Mnemonikliste wird in Kapitel 3 ausführlich beschrieben.
Wie schon im Kapitel 1 erwähnt wurde, kann der MC 68000 Daten als Byte, Wort und Doppelwort verarbeiten. Einige Befehle können nur mit einer Datenlänge arbeiten; andere dagegen mit zwei oder sogar mit allen drei Datenlängen. Für Befehle, die mehrere Datenlängen haben können, muss dem MC 68000 «gesagt» werden, welche Datenlänge nun verarbeitet werden soll. Das wird erreicht, indem der Mnemonik eine Nachsilbe (Datenlängen-Code) angehängt wird. Ein Befehl, der den Wert vom Datenregister D0 zum Inhalt von D1 addiert, sieht wie folgt aus:

```
ADD.X          D0, D1
```

wobei .X die zu addierende Datenlänge der beiden Register D0, D1 bedeutet. Es gibt für .X folgende Möglichkeiten:

.B	*B*yte	(8 Bit)
.W	*W*ort	(16 Bit)
.L	Doppelwort (*L*ong)	(32 Bit)

Falls der Datenlängencode weggelassen wird, nimmt der Assembler die Datenlänge eines Wortes. Damit gibt es für den ADD-Befehl folgende Möglichkeiten:

ADD.B	D0, D1	(Byte)
ADD.W	D0, D1	(Wort)
ADD	D0, D1	(Wort)
ADD.L		(Doppelwort)

2.2.4 Das Operandenfeld

Je nach Befehl wird das Operandenfeld benutzt oder nicht. Dieses Feld enthält entweder einen oder zwei Operanden und ist mindestens durch eine Leerstelle vom Mnemonikfeld getrennt. Falls zwei Operanden verlangt sind, müssen diese durch ein Komma (,) getrennt werden. Für diesen Befehlstyp bedeutet der erste Operand die *Quelle* (source) und der zweite die *Senke* (sink oder Bestimmungsort). Der Quellenoperand bestimmt den Wert, der zu etwas addiert, von etwas subtrahiert, mit etwas verglichen oder im Bestimmungsoperanden abgespeichert wird. Aus diesem Grunde kann der Quellenoperand nie durch eine Operation verändert werden. Der Bestimmungsoperand (Senke) dagegen wird praktisch immer durch die Operation verändert. In Kapitel 3 werden wir für jeden Befehl des MC 68000 die Adressiermöglichkeiten der Operanden behandeln.

2.2.5 Das Kommentarfeld

Das nicht obligatorische Kommentarfeld wird vom Programmierer verwendet, um das Programm lesbarer und verständlicher zu machen. Dieses Feld wird vom Assembler nicht beachtet, aber gleichwohl ausgedruckt. Wenn ein Kommentar geschrieben wird, muss dieser mit mindestens einer Leerstelle vom vorhergehenden Element getrennt sein.

2.2.6 Reine Kommentarzeilen

Es können auch reine Kommentarzeilen geschrieben werden, um ein Programm oder einen Codeteil, die Registerkonfiguration, die Speicherzuteilung oder sonst etwas zu dokumentieren. Diese Kommentare werden mit einem * in der ersten Kolonne markiert. Während der Assemblierung wird der Assembler den Beginn der Kommentarzeile erkennen und den Kommentar nicht beachten.

2.3 Assembleranweisungen

2.3.1 Zweck

Assembleranweisungen oder «Pseudooperationen» stellen eine spezielle Gruppe von Anweisungen an den Assembler dar. Sie ordnen dem Objektprogramm einen gewissen Speicherbereich zu, definieren Symbole, weisen bestimmte Speicheradressen für temporäre Speicherung zu, steuern das Ausdruckformat und führen eine Anzahl kleiner Verwaltungsfunktionen aus. Mit Ausnahme der Konstantendefinition werden diese Befehle nicht in Objektcode übersetzt.
Die Assembleranweisungen haben, wie die Assemblerbefehle, bis zu fünf Felder:
Zeilennummer [Label] Anweisung [Operand] [Kommentar].
Hier gilt das gleiche wie für die Assemblerbefehle. Die Zeilennummer ist eine editor- oder assemblergenerierte Quellenzeilenidentifikation, die bis zu vier Dezimalziffern lang sein kann. Die andern vier Felder werden vom Anwender definiert. Von diesen ist nur das Anweisungsfeld immer notwendig. Die Felder in eckigen Klammern sind fakultativ.
Dazu müssen noch einige Erklärungen gegeben werden. Das Kommentarfeld ist das einzige, das immer beliebig gesetzt oder weggelassen werden kann. Labels können nur in fünf Fällen verwendet werden, und Operanden können nur mit Anweisungen gebraucht werden, die diese verlangen. Tabelle 2.1 fasst die Assembleranweisungen und ihr entsprechendes Format zusammen.
Die Assembleranweisungen können wie die Befehle in einem freien Format eingegeben werden. Die Felder können also irgendwo auf einer Zeile erscheinen. Sie müssen allerdings durch mindestens eine Leerstelle getrennt sein.

Anweisung	Bedeutung			Format
Assemblierungssteuerung				
ORG	Absolute Adresszuweisung		ORG	Ausdruck
			ORG.L	Ausdruck
RORG	Relative Adresszuweisung		RORG	Ausdruck
End	Ende des Quellenprogramms		END	
Symboldefinition				
EQU	Symbol gleich dem Wert (permanent)	Label	EQU	Ausdruck
SET	Setze Symbolwert (temporär)	Label	SET	Ausdruck
Speicherzuteilung				
DC	Definiert eine Konstante	[Label]	DC.B	Operand(en)
		[Label]	DC[.W]	Operand(en)
		[Label]	DC.L	Operand(en)
DS	Definiert einen Speicherplatz	[Label]	DS.B	Operand
		[Label]	DS[.W]	Operand
		[Label]	DS.L	Operand
Ausdrucksteuerung				
PAGE	Neue Seite		PAGE	
LIST	Druckt das Assemblierte		LIST	
NO LIST	Druckt das Assemblierte nicht		NO LIST	
			NOL	
SPC n	Spring n Zeilen der Assemblierliste		SPC	n
NO PAGE	Keine Seiten numerieren		NO PAGE	
LLEN m	Setze Zeilenlänge m		LLEN	m
TTL	Drucke Titel auf jede Seite		TTL	Titelname
NOOBJ	Kein Objektcode		NOOBJ	
FAIL	Drucke Fehlermitteilung		FAIL	Ausdruck
G	Konstantencode		G	
Bedingte Assemblierung				
IFEQ	Assembliere falls = 0		IFEQ	Ausdruck
IFNE	Assembliere falls ≠ 0		IFNE	Ausdruck
ENDC	Ende des bedingten Assemblierens		ENDC	
Makrodefinition				
MACRO	Definiert ein Makro	Label	MACRO	
ENDM	Ende des Makros		ENDM	
MEXIT	Spring auf das Ende des Makros		MEXIT	

Tabelle 2.1 Übersicht der Assemblierungsanweisungen

2.3.2 Assemblierungssteueranweisungen

Der Assembler hat zwei «Origin»-Anweisungen, die absolute (ORG) und die relative (RORG). Diese erlauben dem Anwender, seine Programme, Subroutinen und Daten irgendwo im Speicher zu laden. Programme und Daten können, in Abhängigkeit der Speicherkonfiguration, in verschiedene Speicherbereiche geladen werden. Der Assembler hat einen Speicherplatzzeiger (vergleichbar mit dem internen Programmzähler des MC 68000), der auf die Speicheradresse weist, wo der Objektcode des nächsten Befehls oder Daten zu holen sind. ORG wie RORG veranlassen den Assembler, eine neue, spezifizierte Adresse in den Speicherplatzzeiger zu laden, um danach die Speicherzuweisung der folgenden Ausdrücke auszuführen. ORG weist einen absoluten Speicherplatz zu, RORG dagegen nur den relativen.
Die ORG-Anweisung wird verwendet, wenn man eine Startadresse auswählt, bei der ein Programm oder Daten abgespeichert werden sollen. Die zwei gültigen Formate ORG und ORG.L bewirken das gleiche wie die Befehle, die sich auf einen Label beziehen und im Programm assembliert werden.
Wenn ORG eingesetzt ist, werden Befehle, die sich auf einen folgenden Label beziehen, in einer kurzen, schnell durchgeführten Art assembliert. Die Labels müssen aber innerhalb der hexadezimalen Adressen 0 ... 7FFF liegen. Wird ORG.L gebraucht, so werden die gleichen Befehle in einer langen, viel Zeit kostenden Art assembliert. Dafür können sich die Labels irgendwo im Speicher befinden.
Die RORG-Zuweisung ist für verschiedene Anwendungen nützlich:

- Mischen von Assemblerprogrammen mit Programmen, die in einer höheren Sprache geschrieben werden, ohne dass man sich darum kümmern muss, wo der Objektcode hinkommt.
- Entwicklung von verschiebbaren Subroutinen, die von irgendwo im Speicher geholt und ausgeführt werden können.
- Konstruktion von Programmkomponenten, die später zu einem grossen Programm zusammengesetzt werden.

Die letzte Assemblierungskontrollanweisung, «Ende des Quellenprogrammes» (END), teilt dem Assembler mit, dass er das Ende des Quellenprogramms erreicht hat.

2.3.3 Symboldefinitionsanweisung

Die beiden Anweisungen, EQU, «Gleich dem Wert» und SET, «Setze den Wert», werden gebraucht, um Symbolen im Programm numerische Werte zuzuweisen. In beiden Fällen nimmt der Assembler den Ausdruck im Operandenfeld und weist das Resultat dem Symbol im Labelfeld zu. Symbole, die mit SET zugewiesen werden, können später im Programm neu definiert werden. Dagegen können die durch EQU definierten Symbole

nicht verändert werden. Ausdrücke und Symbole werden später in diesem Kapitel vollständig beschrieben. Kurz erklärt ist ein Ausdruck eine Kombination von Symbolen, Konstanten, algebraischen Operatoren und Klammern (vergleichbar mit der rechten Seite einer algebraischen Gleichung), während ein Symbol eine Serie von alphanumerischen Zeichen wie zum Beispiel ein Label ist. Für die EQU- und SET-Anweisung muss der Ausdruck eine ganze Zahl sein, damit er eine Adresse oder ein Datenwert sein kann.
Da die EQU-Anweisung permanent ist, wird sie angewendet, um Subroutinen- und Geräteadressen, oft gebrauchte Konstanten usw. zu definieren. Dazu einige Beispiele:

```
SUBR    EQU    $2000
CONST   EQU    5634
PIA2    EQU    $FEFF00
```

Man kann auch Symbole mit andern Symbolen definieren:

```
LAST    EQU    FINAL
STRT3   EQU    START+3
```

Das Symbol im Operandenfeld muss natürlich vorher definiert sein.
Da die SET-Anweisung temporär sein kann, wird sie für die Definition von variablen Daten wie Maskenmuster oder Konversionsfaktoren verwendet. Die folgenden SET-Anweisungen können zum Beispiel im gleichen Programm auftreten:

```
MASK1   SET    $FFFE
MASK1   SET    $FFFD
```

Wenn das Programm assembliert ist, werden alle MASK1 durch den Wert $FFFE ersetzt bis zum zweiten SET. Danach wird dem Symbol MASK1 der Wert $FFFD zugeordnet.

2.3.4 Speicherdefinitionsanweisung

Die Definitionsanweisungen für Speicherkonstanten (DC) und Speicherplatz (DS) können eine oder mehrere Adressen in einem Lese- und Schreibspeicher definieren. Zugewiesene Speicherplätze können entweder mit Werten initialisiert (DC) oder einfach für spätere Verwendung durch das Programm reserviert werden (DS). Zu beachten ist, dass die in Tabelle 1 aufgeführten Assembleranweisungen DC und DS mit Datenlängencodes zu ergänzen sind, um Byte, Wörter oder Doppelwörter zu bestimmen.
Die DC-Anweisung kann zum Aufstellen von Datentabellen wie ASCII-Mitteilungstabellen, indirekte Adressen usw. eingesetzt werden. Der Assembler wird jeden Ausdruck im Operandenfeld als Zahlenwert verstehen und diesen Wert in den ent-

sprechenden Speicherplatz schreiben. Mehrere Operanden müssen durch Kommas getrennt werden. Dazu einige Beispiele:

```
TABLE       DC.W        10, 5, 7, 2
```

Beginnend bei der Adresse TABLE werden die Dezimalzahlen 10, 5, 7 und 2 *wort*weise (.W) hintereinander in binärer Form in den Speicher geschrieben.

```
ALBL        DC          LABEL +1
```

Bei der Adresse ALBL wird die Adresse LABEL plus 1 als Wort eingetragen.

```
TABL1       DC.L        10, 5, 7, 2
```

Beginnend bei der Adresse TABL1 werden die Dezimalzahlen 10, 5, 7 und 2 *doppelwort*weise (long .L) in binärer Form rechtsbündig in den Speicher geschrieben.
ASCII-Zeichenausdrücke müssen nicht durch Kommas getrennt werden, sondern nur am Anfang und am Ende des Ausdrucks durch ein Apostroph (') gekennzeichnet sein, ausser eine weitere DC.B-Anweisung folge.

```
CONST       DC.B        43
```

Der Speicherplatz erhält den Wert 43. Das übrigbleibende Byte wird Null sein, ausser der nächste Befehl sei wieder eine DC.B-Anweisung.
Wenn man eine ungerade Anzahl von ASCII-Operanden mit DC.W- oder DC.L-Anweisungen eingibt, so wird der Assembler die restlichen Byte der rechten Seite mit Nullen auffüllen.
Zum Beispiel:

```
NUMBR       DC.L        '12345'
```

Der Speicher wird in den acht folgenden Byte die Werte «1234» und «5000» enthalten.

```
N1          DC 'X'
```

Der Speicher wird in den zwei folgenden Byte «X0» haben.
Die DS-Anweisung erlaubt, einem Speicherbereich einen Namen und die folgende Anzahl Byte zuzuordnen, ohne dabei diesen Speicherbereich in irgendeiner Weise zu initialisieren.
Zum Beispiel:

```
TEMP0       DS.B        10
```

Die 10 nächsten Byte sind von der Adresse TEMP0 an reserviert.

```
TEMP1       DS.W        10
```

Die 10 nächsten Wörter sind von TEMP1 an reserviert.

Die DS-Anweisung hat keinen eingebauten Schutz gegen Adressierungsungenauigkeit. Wenn man wortbreite Daten

erzwingen will, muss nach dem Befehl DS.B noch DS 0 beigefügt werden.
Die Steueranweisungen zum Drucken werden hier nicht beschrieben, da die meisten selbsterklärend sind und alle im MC68000-Cross-Macro-Assembler-Handbuch von Motorola vollständig erklärt sind.

2.4 Ausdrücke im Operandenfeld

Ein Ausdruck ist eine Kombination von Symbolen, Konstanten, algebraischen Operatoren und Klammern, die vom Assembler als ganzzahlige Daten- oder Adressoperanden erkannt werden.

2.4.1 Symbole

Wie die Labels bestehen auch Symbole aus 1 bis 30 alphanumerischen Zeichen, die mit einem Buchstaben (A bis Z) beginnen. Alle 30 Zeichen sind signifikant. Beim Ausdrucken werden immer nur die ersten 8 Zeichen ausgegeben. Die Symbole A0 ... A7, D0 ... D7, CCR, SR, SP und USP sind spezielle, vom Assembler erkannte Registernamen, die wohl im Operandenfeld, aber nicht im Labelfeld erscheinen dürfen.
Ein Symbol kann einen absoluten oder einen relativen Wert haben. Ein Symbol hat einen *absoluten* Wert, falls es durch EQU oder SET mit einem absoluten Wert definiert wurde oder falls ein ORG-Befehl der Symboldefinition vorangegangen ist. Ein Symbol hat einen *relativen* Wert, falls es durch EQU oder SET mit einem relativen Wert definiert wurde oder falls ein RORG-Befehl der Symboldefinition vorausgegangen ist oder falls weder ORG noch RORG der Symboldefinition vorausgegangen ist (das heisst der «Defaultwert» ist RORG 0).

2.4.2 Konstanten

Der Assembler akzeptiert sowohl numerische Konstanten als auch ASCII-Zeichen. Eine Folge von dezimalen Ziffern (zum Beispiel 12345) wird als Dezimalzahl interpretiert, eine Folge von hexadezimalen Ziffern, die mit einem Dollarzeichen beginnt (zum Beispiel $A5C7), wird als hexadezimale Zahl betrachtet. Ein ASCII-Ausdruck ist eine Folge von bis zu vier ASCII-Zeichen, die mit je einem Apostroph eingeklammert sind (zum Beispiel 'ABCD').

2.4.3 Algebraische Operatoren

Der Assembler erlaubt, Elemente eines Ausdrucks mit vier arithmetischen, vier logischen und einem speziellen Operator zu kombinieren. Die arithmetischen Operatoren sind: + (addie-

ren), - (subtrahieren), * (multiplizieren) und / (dividieren). Die EQU-Sequenz, zum Beispiel

```
START    EQU    $2000
STARTP6  EQU    START+6
STARTM1  EQU    START-1
```

weist den Symbolen STARTP6 und STARTM1 die Adressen $2006 beziehungsweise $1FFF zu.

Die logischen Operatoren haben folgende Definitionen:
- Logisches UND (AND) bewirkt, dass jedes Bit des linken Ausdruckes mit dem entsprechenden Bit des rechten logisch UND-verknüpft wird.
- Logisches ODER (OR) bewirkt, dass jedes Bit des linken Ausdruckes mit dem entsprechenden Bit des linken logisch ODER-verknüpft wird.
- Links schieben (≪) bewirkt, dass der linke Ausdruck um die Anzahl (rechter Ausdruck) Bitpositionen nach links geschoben wird.
- Rechts schieben (≫) bewirkt, dass der linke Ausdruck um die Anzahl (rechter Ausdruck) Bitpositionen nach rechts geschoben wird.

Der Spezialoperator, das Komplement-Minus, bewirkt, dass ein Teil eines Ausdruckes negiert oder von null subtrahiert wird. Dieser Operator kann nur zu Beginn eines Ausdruckes oder direkt vor einer linken Klammer auftreten.

2.4.4 Auswahl von Ausdrücken

Wie schon erwähnt, sind Ausdrücke eine Kombination von Symbolen, Konstanten, algebraischen Operatoren und Klammern. Während der Assemblierung sucht der Assembler zuerst die Ausdrücke und arbeitet Klammern von innen nach aussen ab. Danach werden die Operatoren in folgender Reihenfolge behandelt:
Komplement-Minus, Schieben, UND oder ODER, Multiplikation und Division, Addition und Subtraktion.
Operatoren der gleichen Priorität (zum Beispiel «*» und «/») werden von links nach rechts der Reihe nach verarbeitet. Alle dazwischenliegenden Werte werden zu einem ganzzahligen 32-Bit-Wert verarbeitet. Das Resultat eines Ausdruckes ist somit ein ganzzahliger, 32 Bit langer Wert.

2.5 Bedingte Assemblierung

Die Möglichkeit, bedingt zu assemblieren, erlaubt dem Anwender, je nach den zur Zeit der Assemblierung bestehenden Bedingungen, Quellenprogrammteile ein- oder auszuschliessen. Einige Anwendungen für bedingte Assemblierung:

- Ein- oder Ausschliessen bestimmter Variablen,
- Setzen von Diagnostik- oder speziellen Bedingungen für die folgenden Testläufe,
- Generieren spezieller Versionen von mehrfach verwendeten Programmen.

Für den MC68000-Assembler müssen den Quellenprogrammteilen, die entweder ein- oder ausgeschlossen werden sollen, eine der beiden Anweisungen vorangehen: IFEQ und IFNE, gefolgt von ENDC am Programmteilende. Wenn eine IFEQ-Anweisung verwendet wird, kann der Programmteil nur assembliert werden, falls der Ausdruck im Operandenfeld gleich Null ist. Wenn IFNE gebraucht wird, kann der Programmteil nur assembliert werden, falls der Ausdruck verschieden von Null ist.
Die bedingte Assemblierung wird zum Beispiel dort eingesetzt, wo es möglich sein soll, ein Programm zu schreiben, dessen Ein- und Ausgaberoutine davon abhängig gemacht wird, ob ein Disk- oder ein Lochstreifensystem verwendet wird. Dazu ist zu bemerken, dass es ein Flag DORT gibt, das anzeigt, ob Disk- oder Lochstreifen-Ein- und -Ausgabe verwendet wird. Wenn DORT Null ist, so wird das Programm für ein Disksystem, andernfalls für ein Lochstreifensystem assembliert.

2.6 Makros

Der Anwender wird oft in die Situation kommen, eine bestimmte Sequenz mehrmals in einem Programm ausführen zu müssen. Anstatt jedesmal die Befehlssequenz zu schreiben, kann diese auf zwei verschiedene Arten nur einmal geschrieben werden: entweder als *Subroutine* oder als *Makro.*
Wie die meisten Leser schon wissen, ist die Subroutine eine Befehlssequenz, die nur einmal in einem Programm erscheint. Jedesmal, wenn die Subroutine fertig durchlaufen ist, geht die Steuerung durch einen «Return»-Befehl wieder an das aufrufende Programm zurück. Subroutinen sind im Kapitel 3 im Detail beschrieben.
Wie die Subroutinen, erlauben auch die Makros dem Anwender, einer Befehlssequenz einen Namen zu geben. Jedesmal, wenn der Name in einem Operandenfeld eines Quellenprogramms auftaucht, wird der Assembler diesen Makronamen durch die entsprechenden Instruktionen ersetzen. Darin liegt der Unterschied zwischen Subroutine und Makro: Die Subroutinenbefehle werden während der *Programmausführung* eingesetzt, während die Makrobefehle beim *Assemblieren* eingesetzt werden.

Makros haben folgende *Vorteile:*
- Kürzere Quellenprogramme,
- Bessere Programmdokumentation,
- Verwendung von ausgetesteten Befehlssequenzen. Ist einmal

ein Makro fehlerfrei, so kann man sicher sein, dass bei Verwendung dieses Makros darin keine Fehler mehr auftreten.
- Einfach abzuändern. Wird ein Makro abgeändert, so ändert der Assembler jedesmal automatisch bei einem Einsatz den Makrobefehl.
- Makros können gebraucht werden, um eine Makrobibliothek aufzubauen, die ein oder mehrere Programmierer für die Programmerstellung brauchen können.
- Schnelle Ausführung. Der Mikroprozessor wird nicht wie bei Subroutinen durch Aufruf- und Rückkehrbefehle verzögert.

Nachteile der Makros:
- Wiederholung der gleichen Befehlssequenz, da das Makro jedesmal, wenn es aufgerufen wird, das Programm vergrössert.
- Ein einziges Makro kann eine Menge Befehle erzeugen.
- Fehlen von Standards.
- Mögliche unerwünschte Effekte in Registern und Statusbit, falls diese Probleme zuwenig beachtet werden.

Makrodefinition

Jede Makrodefinition besteht aus drei Teilen:
1. Makrokopf, bestehend aus MACRO-Anweisung mit dem Makronamen und dem Labelfeld.
2. Makrokörper, bestehend aus den Befehlen, die den Makrocode ausmachen.
3. Makroende, bestehend aus ENDM-Anweisung, die das Ende der Makrodefinition anzeigt.

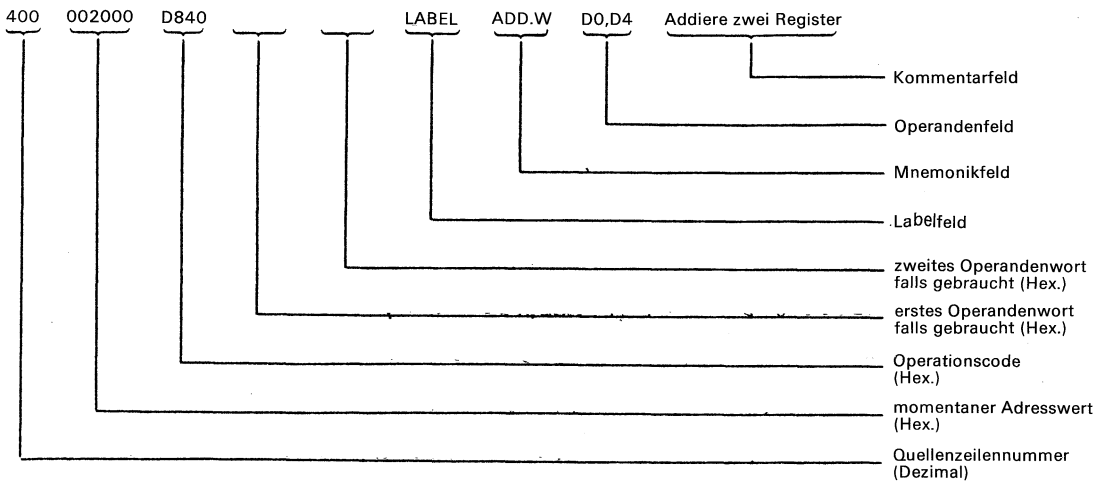


Bild 2.1 Standardformat des Objektlistings, wie es der Makroassembler liefert

Der Assembler erlaubt dem Anwender, bis zu neun Parameter in ein Makro zu überführen. Diese Parameter müssen dann im Operandenfeld des Makroaufrufes stehen. Der Assembler lässt auch variable Datenlängencode in einem Makro zu.
Es gibt noch eine Makroanweisung, die bis jetzt noch nicht erwähnt worden ist: MEXIT. Diese Anweisung wird mit der bedingten Assemblierung gebraucht, um die übrigbleibenden Befehle des Makros zu überspringen.

2.7 Zeilendruckerformat

Bild 2.1 zeigt das Zeilenformat eines Objektlistings, das vom Assembler ausgedruckt wird. Jede Seite des Listings kann einen Seitenkopf, Kommentarzeilen, Erweiterungszeilen und Fehlerzeilen haben. Die letzte Seite enthält die Zusammenstellung aller Fehlerzeilen und die Symboltabelle.

3. Der Befehlssatz des MC 68000

Das Kapitel 3 gibt eine detaillierte Beschreibung des Befehlssatzes des MC 68000 und seiner 14 Adressierungsarten. Die Befehle werden für die Behandlung durch Zusammenfassen ähnlicher Befehle nach funktionellen Kriterien geordnet. So werden Additionsbefehle zusammen mit Subtraktionsbefehlen, Schiebebefehle mit Rotierbefehlen usw. behandelt. Durch dieses Vorgehen kann die Verwandtschaft einzelner Befehle sehr einfach aufgezeigt werden.

3.1 Das Befehlsformat im Speicher

Befehle belegen, wie in Bild 3.1 gezeigt, eine bis fünf Speicherzeilen. In der ersten Zeile steht das sogenannte Operationswort, das den Befehl, die Adressierart(en) sowie die Länge des Befehls bestimmt. Die zusätzlichen Zeilen sind belegt, falls der Befehl mit unmittelbarer Adressierung, Absolutadressierung oder mit Verschiebungsangabe arbeitet. Der längste Befehl besteht aus 5 Wörtern, dem Befehlswort, gefolgt von je 2 Wörtern für das Ziel der effektiven Adresserweiterung.
Zweiwort – (oder «lange») Operanden im Falle der unmittelbaren oder absoluten Adressierung werden im Speicher in der Rangfolge höheres Wort/tieferes Wort abgelegt. Falls das höhere Wort an Adresse ADDR abgespeichert ist, befindet sich das tiefere Wort an der Adresse ADDR+2. Dieser Grundsatz ist gültig für den MC 68000 und muss bei der Programmierung eingehalten werden.

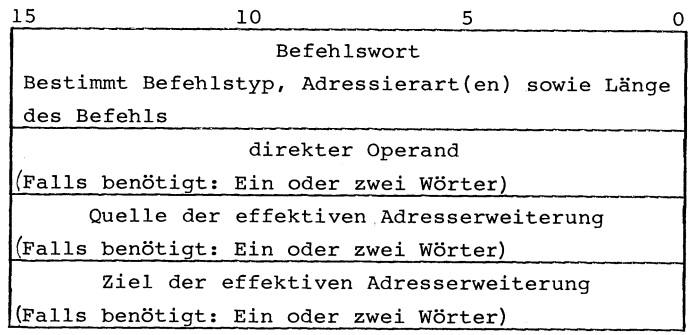


Bild 3.1
Befehlsformat im Speicher

3.2 Adressierarten

Der MC 68000 besitzt 14 Adressierarten. Sie sind in Tabelle 3.1 dargestellt und können 6 Gruppen zugeordnet werden: Register direkt, Register indirekt, absolute Adressierung, relative Adressierung, unmittelbare Adressierung, implizite Adressierung. In der genannten Tabelle ist pro Adressierungsart die Formel angegeben, mit der die effektive Adresse berechnet wird, sowie die Assemblersyntax und die Zahl allfälliger Erweiterungsworte.

Art	Adresserzeugung	Assembler-syntax	Erweiterungs-worte
Register direkt			
Datenregister direkt	EA = Dn	Dn	–
Adressregister direkt	EA = An	An	–
Adressregister indirekt			
Adressregister indirekt	EA = (An)	(An)	–
Adressregister indirekt mit Postdekrement	EA = (An) An←An+N	(An)+	–
Adressregister indirekt mit Predekrement	An←An–N,EA = (An)	–(An)	–
Adressregister indirekt mit Erweiterung	EA = (An)+d_{16}	d(An)	1
Adressregister indirekt mit Index und Erweiterung	EA = (An)+(Ri)+d_8	d(An,Ri)	1
Absolute Adressierung			
Absolut kurz	EA = (Nächstes Wort)	xxxx	1
Absolut lang	EA = (Nächstes und übernächstes Wort)	xxxxxxxx	2
Relative Adressierung			
Relativ mit Verschiebung	EA = (PC)+d_{16}	d	1
Relativ mit Index und Verschiebung	EA = (PC)+(Ri)+d_8	d(Ri)	1
Unmittelbare Adressierung			
Unmittelbar	Daten = nächstes Wort oder nächste Wörter	#xxxx	1 oder 2
Unmittelbar schnell	Daten im Befehlswort enthalten	#xx	–
Implizierte Adressierung			
Implizierte Register	EA = SR, USP, SP, PC		–

Tabelle 3.1 Die Adressierarten des MC 68000

EA = Effektive Adresse
An = Adressregister
Dn = Datenregister
Ri = Adress- oder Datenregister, verwendet als Indexregister
SR = Statusregister
PC = Programmzähler
SP = Aktiver Systemstapelzeiger
USP = Benutzerstapelzeiger
d_8 = 8-Bit-Verschiebungsangabe
d_{16} = 16-Bit-Verschiebungsangabe
N = 1 für Byte; 2 für Wort; 4 für Doppelwort
() = Inhalt von
→ = Ersetzt

Falls ein Operand adressiert wird, der im Speicher abgelegt ist (was der Fall ist bei unmittelbarer, absoluter oder Adressierung mit Verschiebung), so müssen die Adressierregeln des MC 68000 angewendet werden:

1. Auf Byteoperanden kann entweder durch *gerade oder ungerade* Adressen zugegriffen werden.
2. Auf Wort- und Doppeloperanden muss durch eine *gerade* Adresse zugegriffen werden.

Falls obige Regel nicht eingehalten wird, gibt der MC 68000 eine Fehlermeldung (siehe Kapitel 7).
Die meisten der nachfolgenden Beschreibungen der Adressierarten enthalten zur Verdeutlichung Beispiele, die den MOVE-Befehl enthalten. Der MOVE-Befehl hat das folgende allgemeine Format:

MOVE.X $(EA_{Quelle}),(EA_{Ziel})$ (EA: effektive Adresse)

X steht als Datenlängecode der zu verschiebenden Daten (B, W oder L; siehe Kapitel 2). Der MOVE-Befehl hat immer 2 Operanden; der erste adressiert den Speicherplatz oder das Register, das die zu transferierenden Daten enthält (Quelle), der zweite adressiert den Speicherplatz oder das Register, wo die verschobenen Daten abzulegen sind (Ziel/Bestimmungsort).
Der MOVE-Befehl gehört zu den wirksamsten Befehlen des MC 68000. Abhängig von der Adressierart für Quelle und Ziel können durch ihn Daten transportiert werden von Register zu Register, von Register an einen Speicherplatz, vom Speicherplatz zu einem Register oder von einem Speicherplatz zu einem anderen ohne Beeinflussung eines Registers. Es ist sogar möglich, mit ihm unmittelbar folgende Daten in ein Register oder zu einem Speicherplatz zu transportieren.

3.2.1 Adressierart Register direkt

Die Adressierungsart «Register direkt» holt Datenoperanden von irgendeinem (oder lädt ihn in irgendein) Daten- oder Adressregister. Der Befehl

```
MOVE.L A0,D1
```

lädt zum Beispiel den 32-Bit-Inhalt des Adressregisters A0 in das Datenregister D1, ohne den Inhalt von A0 zu verändern.

3.2.2 Adressierart Adressregister indirekt

In dieser Gruppe «zeigt» der Inhalt eines Adressregisters auf einen Operanden. Das bedeutet, dass das spezifizierte Adressregister eine Basisadresse enthält, die der MC 68000 zur Berechnung der effektiven Operandenadresse benützt (falls der Befehl ein Sprungbefehl ist, ist der Operand eine Adresse; sonst ist er

ein Datenwort). Der Zusammenhang zwischen Basisadresse und der effektiven Adresse hängt davon ab, welche der 5 möglichen indirekten Adressierarten verwendet wird.
Bei der einfachsten Art der Gruppe, Adressregister indirekt, enthält das Adressregister die effektive Adresse selbst. Der Befehl

```
MOVE.W  (A0,)D1
```

lädt das Wort, das an Adresse A0 abgespeichert ist, in die 16 tieferen Bit des Datenregisters D1. Bild 3.2 illustriert den MOVE-Befehl, wo A0 auf den Speicherplatz $53F00 zeigt, der den Wert $1C9A enthält.

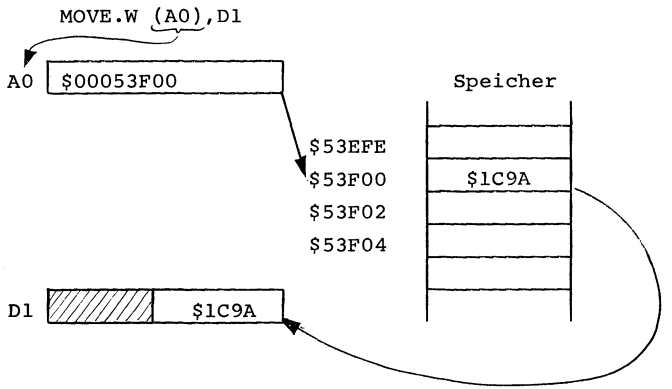


Bild 3.2
Adressierungsart «Adressregister indirekt»

3.2.3 Adressregister indirekt mit Nachinkrementierung oder Vordekrementierung

Die beiden indirekten Adressierarten mit Nachinkrementierung bzw. Vordekrementierung ermöglichen, mit einem Befehl auf Daten in einem Speicherplatz zuzugreifen und sie zu verschieben sowie die Adresszeiger entweder vor der Operation zu dekrementieren oder nach der Operation zu inkrementieren. Damit ist es sehr leicht, benachbarte Daten im Speicher, zum Beispiel bei Tabellen oder Datenfolgen, zu bearbeiten.
Die erste dieser beiden Arten, Adressregister indirekt mit Nachinkrementierung, addiert nach der Bearbeitung des Operanden 1, 2 oder 4 zum Wert des Adressregisters. Bei einer Byteoperation wird 1, bei einer Wortoperation 2 und bei einer Doppelwortoperation 4 addiert. Der Befehl

```
MOVE.W  (A0)+,(A1)+
```

lädt zum Beispiel das Wort, das an Adresse A0 abgespeichert ist, in den Speicherplatz an Adresse A1 und erhöht anschliessend beide Adresszeiger um 2. Dieser MOVE-Befehl kann

natürlich in einer Schleifenanweisung verwendet werden, um eine Anzahl von Datenworten von einem Teil des Speichers in einen andern zu verschieben. Bild 3.3 illustriert, wie der MOVE-Befehl ausgeführt wird, falls die beiden Adresszeiger zuerst auf die Adressen $53F00 bzw. $60000 zeigen und die Quellenadresse den Wert $1C9A enthält.

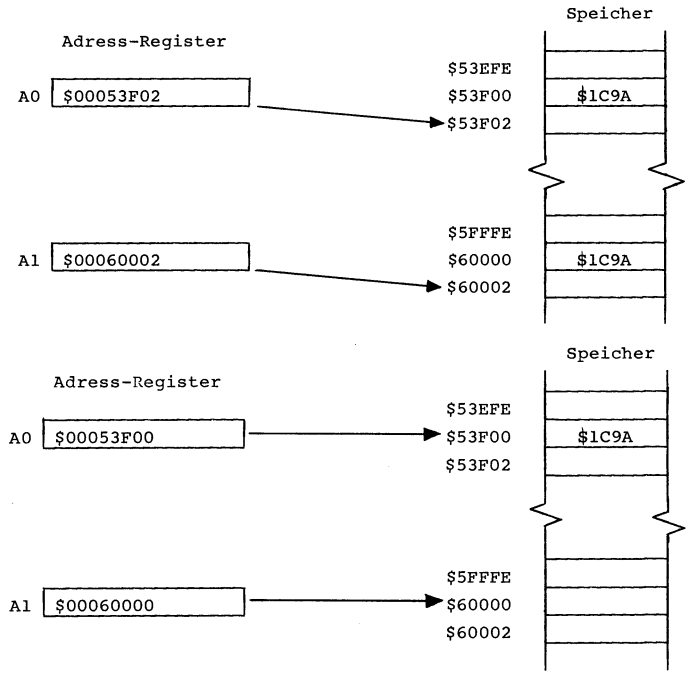


Bild 3.3
Nachinkrementieren eines Adressregisters mittels MOVE.W (A0)+, (A1)+
a) vor Ausführen, b) nach Ausführen des Befehls

In ähnlicher Weise wie bei der vorbeschriebenen Art subtrahiert die Art «Adressregister indirekt mit Vordekrementierung» 1, 2 oder 4 vom Wert des Adressregisters, bevor der Operand bearbeitet wird. Auch diese Art kann verwendet werden, um Datenblöcke von einem Speicherbereich in einen andern zu transferieren, nur werden hier die Adressregister *vor* der Befehlsausführung dekrementiert. Der Befehl

```
MOVE.W  -(A0),-(A1)
```

kopiert zum Beispiel ein Datenwort vom Quellenplatz zum Zielplatz, nachdem die beiden Adressregister um je 2 dekrementiert worden sind.
Wie in Kapitel 1 erwähnt, können alle 8 Adressregister des MC 68000 als Stapelzeiger verwendet werden. Dabei wird A7 als Systemstapelzeiger gebraucht, während A0 bis A6 als Anwenderstapelzeiger zur Verfügung stehen. Daraus folgt, dass der MC 68000 bis zu 8 Anwenderstapel im Speicher behandeln kann.

Aus den vorhergehenden Abschnitten ist ersichtlich, dass die Adressierarten mit Nachinkrementierung und Vordekrementierung sehr gut verwendet werden können, um Stapel zu bearbeiten. Falls A0 als Anwenderstapelzeiger verwendet wird, schreibt der Befehl

```
MOVE.L  D0,-(A0)
```

den 32-Bit-Inhalt von D0 in den Stapel und der Befehl

```
MOVE.L  (A0)+,D0
```

speichert den ursprünglichen Inhalt von D0 wieder zurück. Zusätzlich gibt es noch eine Variante des MOVE-Befehls, genannt «Transportiere mehrere Register» (MOVEM), um Gruppen von Registern in den Stapel zu schieben bzw. wieder herauszuholen.
Die noch zu beschreibenden zwei Adressierarten unterstützen den Tabellenzugriff, indem sie die Addition von Verschiebungen und Indizes zum Adressregister ermöglichen.

3.2.4 Adressregister indirekt mit Verschiebung

In dieser Adressierart wird vor der Befehlsausführung eine im Befehl spezifizierte, maximal 16 Bit lange ganze Zahl zum Inhalt des Adressregisters addiert und mit der resultierenden Adresse zu einem Datenwort im Speicher zugegriffen. Die Adressierart eignet sich besonders gut für die Bearbeitung von Listen oder Tabellen, wo das Adressregister die Anfangsadresse enthält und die zu addierende Zahl die relative Verschiebung zur Anfangsadresse spezifiziert.
Die Verschiebung wird in Byte angegeben. Das bedeutet, dass in Datentabellen, die als Datenelemente Byte enthalten, die Verschiebung gleich der Elementnummer ist; in Tabellen, die als Datenelemente Wörter enthalten, ist die Verschiebung gleich der Elementnummer multipliziert mit 2, in Tabellen mit Doppel-

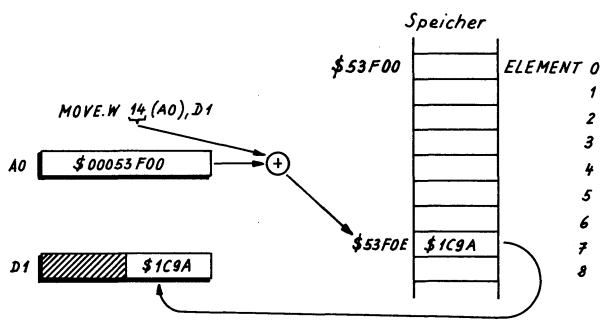


Bild 3.4
Anwenden einer Verschiebung zum Adressregister

wörtern gleich der Elementnummer multipliziert mit 4. Dadurch, dass die Verschiebung eine maximal 16 Bit lange ganze Zahl ist, ergeben sich folgende Verschiebungswerte.

Datenelement	Verschiebung	
	positiv	negativ
Byte	32 767 Byte	32 768 Byte
Wort	16 383 Wörter	16 384 Wörter
Doppelwort	8191 Doppelwörter	8192 Doppelwörter

Als Beispiel für diese Adressierart wird der Befehl

```
MOVE.W  14(A0),D1
```

diskutiert. A0 enthält die Startadresse einer wortorientierten Tabelle. Der obige Befehl lädt den Inhalt des 8. Elements (Element 7) in das tieferwertige Wort von Datenregister D1. Bild 3.4 illustriert diese Operation.

3.2.5 Adressregister indirekt mit Index

Dies ist die letzte zu beschreibende Adressierart aus der Gruppe der indirekten Adressierarten. Die effektive Operandenadresse wird berechnet, indem zum Adressregister der Inhalt eines Indexregisters (Daten- oder Adressregister) sowie eine maximal 8 Bit lange ganze Zahl addiert werden. Dies ergibt für die Berechnung der Effektivadresse folgende Formel:

$$EA = (An) + (Ri) + d_8$$

Es besteht die Möglichkeit, entweder das ganze Indexregister oder nur das tieferwertige Wort davon zu verwenden. Im ersten Fall muss zum Indexregister .L, im zweiten Fall .W gesetzt werden. Die Befehlsausführungszeit ist in beiden Fällen gleich.
Weil diese Adressierart zwei verschiedene Offsets anbietet, ist sie nützlich für die Anwendung in zweidimensionalen Arrays. In solchen Fällen enthält das Adressregister gewöhnlich die Startadresse des Arrays, während die Verschiebung und das Indexregister den Reihen- und Kolonnenoffset angeben (oder umgekehrt). Der Index ist normalerweise als Byteanzahl angegeben und in einem Datenregister enthalten; für die Verschiebung (in Byte) wird ein Symbol verwendet.
Zur Illustration dieser Adressierart wird angenommen, dass ein auf dem MC 68000 basierendes System eine chemische Produktionsanlage mit 6 Druckventilen überwacht. Das System liefert jede halbe Stunde die Werte der 6 Ventile und speichert diese ab. In der Zeit einer Woche ergibt dies einen Array mit 366 Blöcken zu je 6 Werten.

In Bild 3.5 wird das Lesen des Wertes von Ventil 4 in der zweiten Leseoperation (Ablesung 1) gezeigt. Der entsprechende Befehl lautet:

```
MOVE.W  VALVE(A0,D0.W),D1
```

Die Startadresse des Arrays ist $53F00.

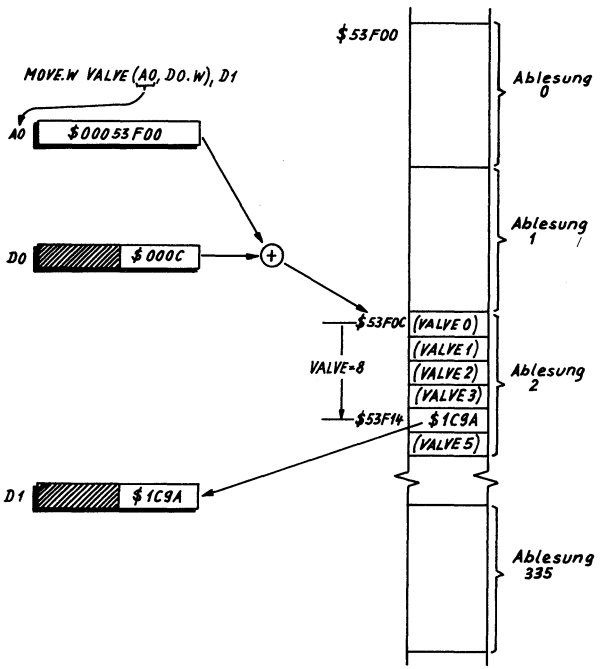


Bild 3.5
Ermittlung eines Datenwertes aus einem zweidimensionalen Array

3.2.6 Absolute Datenadressierung

Bei der absoluten Datenadressierung ist die effektive Adresse selbst als Operand spezifiziert. Es gibt zwei absolute Adressierarten: «Absolut kurz» und «absolut lang». Im ersten Fall ist der Operand eine 16-Bit-Adresse, im zweiten Fall eine 32-Bit-Adresse.
«Absolut kurz» erlaubt, entweder zu den ersten 32 KByte im Speicher (0 ... $7FFF) oder zu den höchsten 32 KByte im Speicher ($FF8000 ... $FFFFFF) zuzugreifen.
«Absolut lang» gibt die Möglichkeit, zu irgendeinem Speicherplatz des 16-MByte-Speichers des MC 68000 zuzugreifen.
Befehle mit «absolut kurz» belegen im Speicher 2 Wörter und benötigen zur Ausführung 12 Prozessorzyklen, solche mit «absolut lang» 3 Wörter respektive 16 Zyklen. Mit diesen zwei

absoluten Adressierarten unterstützt der MC 68000 Anwendungen mit einem sehr grossen Adressierraum, ohne die Effektivität von Anwendungen zu beeinträchtigen, die nur einen kleinen Adressierraum benötigen. «Absolut kurz» wird aber auch in Anwendungen mit grossem Adressierraum eingesetzt, um zu häufig gebrauchten oder zwischengespeicherten Daten, die in den höchsten 32 KByte des Speichers abgespeichert sind, zuzugreifen.
Beispiele:

```
Absolut kurz (2 Wörter, 12 Zyklen)
MOVE.W   $3F00,D1

Absolut lang (3 Wörter, 16 Zyklen)
MOVE.W   $03F00,D1
```

Beide Befehle laden das Wort an Adresse $3F00 in das tieferliegende Wort des Datenregisters D1. Der Datenlängecode .W bezieht sich auf die Grösse der verschobenen Daten.
Der Operand eines Befehls mit absoluter Adressierung ist anstelle einer hexadezimalen Zahl oft mit einem Label spezifiziert. Dies wird illustriert am Befehl

```
MOVE.L  TABLE(D0.L),D1
```

Mit diesem Befehl wird das Doppelwort, das sich an Adresse *Table* befindet, in das Adressregister A0 geladen. Bei dieser Art von Befehlsdeklarierung ist von Interesse, ob für die Ausführung «absolut kurz» oder «absolut lang» verwendet wird. Die Antwort ist abhängig davon, ob *Table* in einer höheren oder einer tieferen Adresse als der MOVE-Befehl abgelegt ist:

- Falls *Table* in einer tieferen Adresse als der MOVE-Befehl abgelegt ist (es wird rückwärts referenziert), wird der Assembler die geeignete kurze oder lange Adresse generieren.
- Falls *Table* in einer höheren Adresse als der MOVE-Befehl abgelegt ist (es wird vorwärts referenziert) und der MOVE-Befehl unter eine ORG-Anweisung fällt, wird der Assembler versuchen, eine kurze Adresse zu generieren. Bei Vorwärts-Referenz kann der Assembler durch eine ORG.L-Anweisung gezwungen werden, eine lange Adresse zu nehmen.

Hinweis: Der Assembler generiert absolute Adressen für Befehle, die unter einer ORG-Anweisung, relative Adressen für Befehle, die unter einer RORG-Anweisung stehen.

3.2.7 Programmzähler-relative Adressierung

Die zum Programmzähler relativen Adressierarten geben die Möglichkeit, positionsunabhängige, das heisst «umplazierbare» (relocatable) Programme zu entwickeln, die, wenn einmal geschrieben und assembliert, irgendwo im Speicher ausgeführt

werden können. Bei den zum Programmzähler relativen Adressierarten berechnet der MC 68000 die effektiven Adressen, indem er zu der im Programmzähler enthaltenen Adresse eine Verschiebung addiert. Der Programmzähler zeigt im Berechnungszeitpunkt der effektiven Adresse auf das Erweiterungswort des in Ausführung begriffenen Befehls. Der Wert der Verschiebung ist in diesem Erweiterungswort enthalten.
Die zwei zum Programmzähler relativen Adressierarten, die der MC 68000 anbietet – «Relativ mit Verschiebung» und «Relativ mit Index» –, werden verwendet, um zu Operanden zuzugreifen, die im Speicher einige Byte höher oder tiefer liegen als der in Ausführung begriffene Befehl. Beiden Adressierarten ist gemeinsam, dass die Verschiebung in Symbolen angegeben werden darf, da der Assembler die Verschiebung zur Zeit der Assemblierung berechnen kann (vergleiche die nachfolgend diskutierten Beispiele).
Die Umplazierbarkeit der Programme bleibt dann aber nur gewährleistet, wenn den betreffenden Befehlen eine RORG-Anweisung vorausgeht. Die RORG-Anweisung bewirkt, dass der Assembler die zum Programmzähler relative Adressierung verwendet, während die ORG-Anweisung bewirkt, dass der Assembler absolute Adressierung verwendet.

3.2.7.1 Relativ mit Verschiebung

«Relativ mit Verschiebung» ist die einfachere der beiden Adressierarten. Die effektive Adresse EA berechnet sich als Summe aus der Adresse im Programmzähler und der vorzeichenbehafteten 16-Bit-Verschiebung im Erweiterungswort des Befehls:

$$EA = (PC) + d_{16}$$

3.2.7.2 Relativ mit Index

Die effektive Adresse ist in diesem Fall die Summe der Adresse des Erweiterungswortes, das im Programmzähler enthalten ist, des Inhalts eines Indexregisters (entweder ein Daten- oder ein Adressregister) und einer maximal 8 Bit langen, vorzeichenbehafteten ganzen Zahl, die im Erweiterungswort des in Ausführung begriffenen Befehls enthalten ist. Formelmässig kann dies wie folgt geschrieben werden:

$$EA = (PC) + (Ri) + d_8$$

Diese Art ist besonders nützlich, um Werte aus einer Liste oder einer Datentabelle zu lesen. Im Falle solcher Anwendungen adressiert die Summe des Programmzählers und der 8-Bit-Verschiebung den Anfang der Tabelle, und das Indexregister gibt den Abstand des gewünschten Datenelements vom Tabellenanfang an. Dies wird illustriert in Bild 3.6.

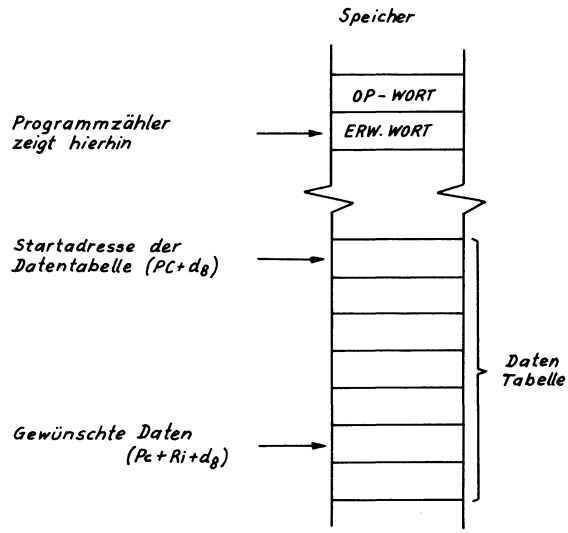


Bild 3.6
Adressierungsart «Programmzähler-relativ mit Index»

Es ist möglich, entweder nur das tieferwertige Wort des Indexregisters oder seinen ganzen, 32 Bit langen Inhalt zu verwenden, indem im Befehl zum Symbol des Registers entweder ein .W oder in .L gesetzt wird (im Falle, dass der Datenlängecode ausgelassen wird, setzt der Assembler als Defaultwert ein .W).
Als Beispiel für die Anwendung dieser Adressierart dient der folgende Befehl:

```
MOVE.W TABLE(D0.L),D1
```

Bei der Assemblierung wird der Assembler aufgrund dieses Befehls die Verschiebung in Byte zwischen dessen Erweiterungswort und der Position *Table,* der Startadresse der Datentabelle, berechnen und mit dem Ergebnis das Erweiterungswort bilden. Der Mikroprozessor wird bei der Befehlsausführung den 32-Bit-Inhalt des Datenregisters D0 zur berechneten Startadresse der Datentabelle addieren und dann den 16-Bit-Inhalt des durch die erhaltene Effektivadresse adressierten Speicherplatzes in die 16 tieferwertigen Bit des Datenregisters D1 laden. Weil die Verschiebung eine 8 Bit lange, vorzeichenbehaftete ganze Zahl ist, darf *Table* sich nicht mehr als 63 Worte höher und nicht mehr als 64 Worte tiefer im Speicher befinden als das Erweiterungswort.

3.2.8 Unmittelbare Datenadressierung

Unmittelbare Datenadressierung wird verwendet, um eine Konstante als Quellenoperanden zu spezifizieren. Diese Konstante

ist Bestandteil des Befehls. Es gibt zwei Adressierarten, die zur unmittelbaren Datenadressierung gezählt werden: «unmittelbar» und «unmittelbar schnell».

3.2.8.1 Unmittelbar

In der Art «unmittelbar» kann ein Byte, ein Wort oder ein Doppelwort als Konstante spezifiziert werden. Die Grösse der Konstante bestimmt das Befehlsformat. Im Falle eines Byte oder eines Wortes umfasst der Befehl ein Erweiterungswort, im Falle eines Doppelwortes zwei Erweiterungsworte. Dies wird gezeigt in Bild 3.7.

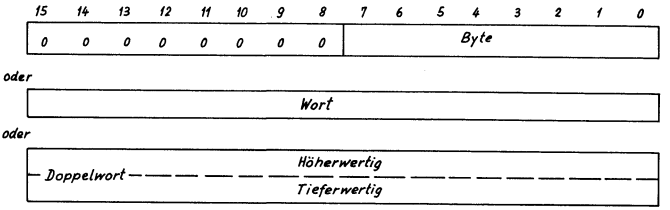


Bild 3.7
Formate der Erweiterungsworte für die Adressierart «unmittelbar»

Bei der Befehlsausführung werden zwei Fälle unterschieden: Falls das Ziel ein Adressregister ist, werden Konstante von der Länge eines Byte oder eines Wortes auf ein Doppelwort vorzeichenerweitert; im Falle, dass das Ziel ein Datenregister ist, gibt es keine Vorzeichenerweiterung. Dazu zwei Beispiele:
1. Beispiel:

```
MOVE.W #$834E,D0
```

Der Befehl lädt den Wert $834E in das tieferwertige Wort des Datenregisters D0.
2. Beispiel:

```
MOVE.W #$834E,A0
```

Der Befehl lädt den Wert $FFFF834E in das Adressregister A0 und beeinflusst damit alle 32 Bit.

3.2.8.2 Unmittelbar schnell

Es gibt nur 3 Befehlstypen, mit denen die Art «unmittelbar schnell» verwendet werden kann:
- ADDQ (addiere schnell, Q = quick)
- SUBQ (subtrahiere schnell)
- MOVEQ (transportiere schnell)

Die Befehle ADDQ und SUBQ geben die Möglichkeit, zu einem Register oder einer Speicherzelle eine ganze Zahl zwischen 1

und 8 zu addieren oder davon zu subtrahieren. Sie sind damit die Inkrementier- beziehungsweise Dekrementierbefehle des MC 68000.
Der Befehl MOVEQ ermöglicht, eine vorzeichenbehaftete Konstante in der Grösse von maximal einem Byte (−128 bis +127) in ein Datenregister zu laden. Die Konstante wird zeichenerweitert auf ein Doppelwort, so dass alle 32 Bit des Datenregisters betroffen sind. Ein Beispiel soll dies verdeutlichen:

```
MOVEQ #-2,D0
```

Der Befehl lädt den Wert $FFFFFFFE (das Zweierkomplement von −2, zeichenerweitert auf ein Doppelwort) in das Datenregister D0.
Die drei zuvor beschriebenen Adressierarten sind mit «schnell» bezeichnet, weil sie im Speicher nur ein Wort belegen (die Konstante ist in das Operationswort eingebettet). Aus diesem Grunde ist die Ausführungszeit viel kürzer als für die gewöhnlichen «unmittelbar»-Arten.

3.2.9 Implizite Adressierung

Einige Befehle verwenden bei der Befehlsausführung ein bestimmtes internes Register, ohne dass dieses im Operanden identifiziert sein muss. Die Adressierung eines solchen Registers wird implizit genannt. Der Sprungbefehl (JMP) lädt beispielsweise immer eine Adresse in den Programmzähler (PC), obwohl der Programmzähler im Befehl nicht explizit als Zielregister identifiziert wird. Als implizite Register werden neben dem Programmzähler (PC) noch folgende Register verwendet: Systemstapelzeiger (SP), Anwenderstapelzeiger (USP), Überwachungsstapelzeiger (SSP), Statusregister (SR). In Tabelle 3.2 sind die Befehle, die die implizite Adressierung verwenden, sowie die betreffenden impliziten Register aufgeführt.

Befehl	Implizite Register
Bedingter Sprung (Bcc), Unbedingter Sprung (BRA)	PC
Sprung zu Subroutine (BSR)	PC,SP
Prüfe Register auf Grenzen (CHK)	SSP,SR
Prüfe Bedingung, vermindere und springe (DBcc)	PC
Division mit Vorzeichen (DIVS)	SSP,SR
Division ohne Vorzeichen (DIVU)	SSP,SR
Sprung (JMP)	PC
Sprung zur Subroutine (JSR)	PC,SP

	Implizite Register
Zuweisung (LINK)	SP
Transportiere Bedingungscode (MOVE CCR)	SR
Transportiere Statusregister (MOVE SR)	SR
Transportiere Benützerstapelzeiger (MOVE USP)	USP
Eintragen der effektiven Adresse (PEA)	SP
Rückkehr von Ausnahme (RTE)	PC,SP,SR
Rückkehr und Rückladung Bedingungscode (RTR)	PC,SP,SR
Rückkehr aus Subroutine (RTS)	PC,SP
Falle (TRAP)	SSP,SR
Falle bei Überlauf (TRAPV)	SSP,SR
Freigabe (UNLK)	SP

Tabelle 3.2
Implizite Befehle

3.2.10 Adressierarten, die Adressen oder Daten vorzeichenerweitern

Obwohl die Daten- und Adressregister des MC 68000 grundsätzlich universell verwendbar sind, werden die Datenregister in erster Linie verwendet, um Daten abzuspeichern, und die Adressregister, um die 32-Bit-Speicheradressen abzuspeichern.

Adressierungsart	**Art der Vorzeichenerweiterung**
Adressregister direkt (als Bestimmung)	Wortadresse verlängert zu Doppelwort
Adressregister indirekt mit Verschiebung	Wortverschiebung verlängert zu Doppelwort
Adressregister indirekt mit Index	1. Byteverschiebung verlängert zu Doppelwort 2. Wortindex verlängert zu Doppelwort
Absolute Adresse kurz	Wortadresse verlängert zu Doppelwort
Programmzähler relativ mit Verschiebung	Wortverschiebung verlängert zu Doppelwort
Programmzähler relativ mit Index	1. Byteverschiebung verlängert zu Doppelwort 2. Wortindex verlängert zu Doppelwort

Tabelle 3.3
Adressierungsarten mit Vorzeichenerweiterung

Dies ist die Begründung dafür, dass durch die zur Verfügung stehenden Adressierarten Information, die in Datenregister geladen wird, nicht vorzeichenerweitert wird, dagegen Information, die in Adressregister geladen wird, immer vorzeichenerweitert wird. In Tabelle 3.3 sind die Adressierarten aufgeführt, die Ursache für Vorzeichenerweiterungen sind. In einem späteren Abschnitt des Kapitels 3 werden die Befehle diskutiert, die Ursache für Vorzeichenerweiterungen sind.

3.3 Einteilung der Adressierarten nach Verwendungszweck

Wie in den vorhergehenden Abschnitten dieses Kapitels aufgezeigt wurde, erfüllt jede der 14 Adressierarten des MC 68000 eine bestimmte Adressierfunktion. Einige davon können verwendet werden, um zu einem Operanden in einem Register, andere um zu einem Operanden an einer bestimmten Speicheradresse oder zu einem Operanden mit einer Verschiebung zu einer bestimmten Speicheradresse zuzugreifen, usw. Einige andere Arten können verwendet werden, um zu irgendeiner von verschiedenen Informationsarten zuzugreifen (beispielsweise kann mit Adressregistern indirekt zu Daten oder Adressen im Speicher zugegriffen werden), während andere in ihrer Verwendung eingeschränkt sind (Adressregister direkt kann sich beispielsweise nur auf einen Adressoperanden, jedoch nicht auf einen Datenoperanden beziehen). Aus den hier geschilderten Gründen können die einzelnen Adressierarten durch die vier folgenden Adresskategorien charakterisiert werden:

1. Daten

Falls eine Adressierart verwendet werden kann, um zu Daten zuzugreifen, wird sie als Adressierart für Daten bezeichnet.

2. Speicher

Falls mit einer Adressierart zu Speicheroperanden zugegriffen werden kann, wird sie als Speicheradressierart bezeichnet.

3. Steuerung

Falls eine Adressierart verwendet werden kann, um zu Speicheroperanden ohne Grössenangabe zuzugreifen, wird sie als Steuerungs-Adressierart bezeichnet.

4. Änderbar

Falls mit einer Adressierart zu änderbaren (schreibbaren) Adressierarten zugegriffen werden kann, wird sie als änderbar bezeichnet.

Tabelle 3.4 zeigt, welchen Adresskategorien jede der Adressierarten des MC 68000 angehört. Diese Tabelle ist für den Programmierer wichtig, weil viele der Befehle die Operanden auf bestimmte Kategorien oder Kombinationen von Kategorien beschränken.

Adressierungsarten	Adressierungskategorien Daten	Speicher	Steuerung	änderbar	Assemblersyntax
Datenregister direkt	X			X	Dn
Adressregister direkt				X	An
Register indirekt	X	X	X	X	(An)
Register indirekt nachinkrementiert	X	X		X	(An)+
Register indirekt vordekrementiert	X	X		X	–(An)
Register indirekt m. Verschiebung	X	X	X	X	d(An)
Register indirekt mit Index	X	X	X	X	d(An,Ri)
Absolut kurz	X	X	X	X	xxxx
Absolut lang	X	X	X	X	xxxxxxxx
Relativ mit Verschiebung	X	X	X		d
Relativ mit Index	X	X	X		d(Ri)
Unmittelbar	X	X			#xxxx

Tabelle 3.4 Effektive Adressierungsarten

So hat z.B. der «addiere schnell»-Befehl die allgemeine Form

```
ADDQ  #<data>,<ea>
```

Als effektive Adresse sind für diesen Befehl nur «änderbare» Adressierarten erlaubt. Dies bedeutet, dass irgendeine Adressierart mit Ausnahme von relativer und unmittelbarer Adressierung verwendet werden kann. Aus diesem Grunde ist ADDQ #2,A0 erlaubt, ADDQ #2,#2 jedoch nicht.
Ein Befehl, der eine Kombination von Kategorien im Operandenfeld benützen kann, ist der MOVE-Befehl mit der allgemeinen Form

```
MOVE <ea>, <ea>
```

Bei diesem Befehl sind für das Quellenfeld alle Adressierarten zugelassen mit Ausnahme der Adressierungsart «Register direkt» bei Byteverarbeitung. Beim Bestimmungsfeld sind nur die Adressierungsarten «Daten änderbar» erlaubt. Das bedeutet für das Bestimmungsfeld, dass sowohl die Kategorien der Datenadressierarten wie auch die der änderbaren Adressierungsarten zugelassen sind. So beinhaltet die Adressierungsart «Daten änderbar» die Datenregister direkt, Adressregister indirekt und die absoluten Adressierungsarten. Umgekehrt sind die Arten «Adressregister direkt», «Programmzähler relativ» und die «unmittelbaren» Arten ausgeschlossen.
Muss man nun annehmen, in ein Adressregister könne kein Transfer gemacht werden, weil Adressregister direkt keine

«Daten änderbar»-Adressierungsart ist? Selbstverständlich nicht, denn es muss natürlich einen Weg geben, um diese Register initialisieren zu können. Dieser Weg führt nicht über MOVE, sondern der MC 68000 verfügt für diese Aufgabe über einen anderen Befehl, MOVEA, transportiere Adresse. Obschon im MC-68000-Benutzerhandbuch MOVE und MOVEA als zwei verschiedene Befehle definiert sind, erlauben die meisten MC-68000-Assembler, eingeschlossen der von Motorola, die Spezifikation eines Adressregisters im Bestimmungsfeld eines MOVE-Befehls. Diese Assembler interpretieren den MOVE-Befehl einfach als MOVEA und erzeugen den entsprechenden Objektcode.

3.4 Befehlsarten

Wie bereits erwähnt, verfügt der MC 68000 über 56 Grundbefehle. Die mnemonische Schreibweise und die Beschreibung dieser Befehle sind in Tabelle 3.5 zusammengestellt. Im weiteren verfügen 8 dieser Befehle über Variationen zur Ausführung von speziellen Operationen; diese Variationen sind in Tabelle 3.6 zusammengestellt.

Beschreibung Mnemonic	englisch	deutsch
ABCD	Add Decimal with Extend	Addiere dezimal mit Erweiterungsbit
ADD	Add	Addiere binär
AND	Logical And	Logisches UND
ASL	Arithmetic Shift Left	Arithmetische Verschiebung links
ASR	Arithmetic Shift Right	Arithmetische Verschiebung rechts
Bcc	Branch Conditionally	Bedingter Sprung
BCHG	Bit Test and Chance	Prüfe ein Bit und ändere es
BCLR	Bit Test and Clear	Prüfe ein Bit und setze es auf 0
BRA	Branch Always	Unbedingter Sprung
BSET	Bit Test and Set	Prüfe ein Bit und setze es
BSR	Branch to Subroutine	Sprung zum Unterprogramm
BTST	Bit Test	Prüfe ein Bit
CHK	Check Register Against Bounds	Prüfe Register auf Grenzen
CLR	Clear Operand	Setze Operand auf 0
CMP	Compare	Vergleiche
DBcc	Test Cond., Decrement and Branch	Prüfe Bedingung, vermindere und springe
DIVS	Signed Divide	Divison mit Vorzeichen
DIVU	Unsigned Divide	Division ohne Vorzeichen
EOR	Exclusive OR Logical	Logisches exklusiv ODER
EXG	Exchange Registers	Vertausche Daten zwischen Registern
EXT	Sign Extend	Vorzeichenerweiterung

Beschreibung Mnemonic	englisch	deutsch
JMP	Jump	Springe
JSR	Jump to Subroutine	Springe zum Unterprogramm
LEA	Load Effective Address	Lade die effektive Adresse
LINK	Link and Allocate	Zuweisung
LSL	Logical Shift Left	Logische Verschiebung nach links
LSR	Logical Shift Right	Logische Verschiebung nach rechts
MOVE	Move Date from Source to Destination	Transportiere Daten von der Quelle zum Ziel
MOVEM	Move Multiple Registers	Transportiere mehrere Register
MOVEP	Move Peripheral Data	Transportiere periphere Daten
MULS	Signed Multiply	Multiplikation mit Vorzeichen
MULU	Unsigned Multiply	Multiplikation ohne Vorzeichen
NBCD	Negate Decimal with Extend	Negiere dezimal mit Erweiterungsbit
NEG	Negate	Negiere
NOP	No Operation	Keine Operation
NOT	One's Complement	Einerkomplement
OR	Logical Or	Logisches ODER
PEA	Push Effective Address	Eintragen der effektiven Adresse
RESET	Reset External Devices	Normieren externer Einheiten
ROL	Rotate Left without Extend	Ringverschiebung links o. Erw.-Bit
ROR	Rotate Right without Extend	Ringverschiebung rechts o. Erw.-Bit
ROXL	Rotate Left with Extend	Ringverschiebung links m. Erw.-Bit
ROXR	Rotate Right with Extend	Ringverschiebung rechts m. Erw.-Bit
RTE	Return from Exception	Rückkehr von Ausnahme
RTR	Return and Restore	Rückkehr, Rückladen Bedingungscodes
RTS	Return from Subroutine	Zurück vom Unterprogramm
SBCD	Subtract Decimal with Extend	Subtrahiere dezimal m. Erw.-Bit
Scc	Set on Condition	Setze in Abhängigkeit der Bedingung
STOP	Stop	Lade das Statusregister und halte an
SUB	Subtract	Subtrahiere binär
SWAP	Swap Data Register Halves	Vertausche Registerhälften
TAS	Test an Set Operand	Prüfe und setze Operand
TRAP	Trap	Falle
TRAPV	Trap on Overflow	Falle bei Überlauf
TST	Test	Prüfe einen Operanden
UNLK	Unlink	Freigabe

Tabelle 3.5
Die 56 Grundbefehle des MC 68 000

Der ganze Befehlssatz kann in 8 funktionelle Gruppen eingeteilt werden:

1. *Datentransportbefehle* verschieben Information zwischen Speicherzellen, Ein- und Ausgabegeräten und allgemein verwendbaren Registern in jeder Kombination.
2. Befehle für ganzzahlige *Arithmetik* führen arithmetische Operationen mit binären Zahlen in einfacher und mehrfacher Genauigkeit durch.
3. *Logische Befehle* führen logische Operationen UND, ODER und EXKLUSIV-ODER mit Speicherzellen und Registern aus.
4. *Schiebe- und Rotierbefehle* schieben und rotieren den Inhalt von Speicherzellen und Registern.
5. *Bitmanipulationsbefehle* prüfen den Zustand individueller Bit und führen je nach Resultat dieser Prüfungen Operationen aus.
6. Binärcodierte *Dezimalbefehle* (BCD) addieren und subtrahieren BCD-Werte.
7. *Programmsteuerbefehle* führen Verzweigungen, Sprünge Subroutinenaufrufe aus und steuern so den Ablauf der Programmausführung.
8. *Systemsteuerbefehle,* eingeschlossen privilegierte Befehle, Trap-Erzeugungsbefehle und Befehle, die das Statusregister benützen oder ändern.

In diesem Kapitel wird der Befehlssatz in der gerade präsentierten Ordnung beschrieben. Wir beginnen mit den Datentransportbefehlen, mit den jetzt bekannten MOVE-Befehlen.

Befehlsart	Variation	Beschreibung englisch	Beschreibung deutsch
ADD	ADD	Add	Addiere
	ADDA	Add Address	Addiere Adresse
	ADDQ	Add Quick	Addiere schnell
	ADDI	Add Immediate	Addiere unmittelbar
	ADDX	Add With Extend	Addiere mit Erweiterung
AND	AND	Logical AND	Logisch UND
	ANDI	AND Immediate	UND unmittelbar
CMP	CMP	Compare	Vergleiche
	CMPA	Compare Address	Vergleiche Adresse
	CMPM	Compare Memory	Vergleiche Speicher
	CMPI	Compare Immediate	Vergleiche unmittelbar
EOR	EOR	Exclusiv-OR	Exklusiv ODER
	EORI	Exclusiv-OR Immediate	Exklusiv ODER unmittelbar

Befehlsart	Variation	Beschreibung englisch	Beschreibung deutsch
MOVE	MOVE	Move	Transportiere
	MOVEA	Move Address	Transportiere Adresse
	MOVEQ	Move Quick	Transportiere schnell
	MOVE f. SR	Move from Status Register	Transportiere vom Statusreg.
	MOVE to SR	Move to Status Register	Transportiere zum Statusreg.
	MOVE to CCR	Move to Condition Codes	Transportiere zu den Bedingungscodes
	MOVE USP	Move User Stack Pointer	Transp. Benutzerstapelzeiger
NEG	NEG	Negate	Negiere
	NEGX	Negate With Extend	Negiere mit Erweiterung
OR	OR	Logical OR	Logisch ODER
	ORI	OR Immediate	ODER unmittelbar
SUB	SUB	Subtract	Subtrahiere
	SUBA	Subtract Address	Subtrahiere Adresse
	SUBI	Subtract Immediate	Subtrahiere unmittelbar
	SUBQ	Subtract Quick	Subtrahiere schnell
	SUBX	Subtract With Extend	Subtrahiere mit Erweiterung

Tabelle 3.6
Variationen von Befehlen

3.4.1 Datentransportbefehle

Die Datentransportbefehle gemäss Tabelle 3.7 werden verwendet, um Informationen zwischen Speicher und Daten- oder Adressregistern zu transferieren. Diese Gruppe enthält zwei zusätzliche Befehle, LINK und UNLINK; sie werden vorwiegend mit Subroutinen verwendet, so dass wir sie separat, anschliessend an die Diskussion der Programmsteuerbefehle, beschreiben.

3.4.1.1 MOVE-Befehl

Der grundsätzliche Befehl in dieser Gruppe ist der MOVE-Befehl, der verwendet werden kann, zum Transfer von Byte-, Wort- oder Doppelwortdaten zwischen zwei Speicherzellen, zwischen einer Speicherzelle und einem Datenregister oder zwischen zwei Datenregistern.
Wenn der MC 68000 im Benützerstatus ist, erlaubt der MOVE-Befehl das Nachführen des Bedingungscoderegisters (MOVE <ea> , CCR) oder das Lesen des gesamten Statusregisters (MOVE SR, <ea>). Im Überwachungsstatus erlaubt der MOVE-Befehl das Nachführen des Statusregisters (MOVE <ea>, SR), das Lesen des Benützerstapelzeigers (MOVE, USP, An), oder das Schreiben des Benützerstapelzeigers (MOVE An, USP). In den vorangegangenen Feldern darf für die effektive Adresse (mit <ea> bezeichnet) kein Adressregister als Quelle oder Bestimmungsort verwendet werden.

Mnemonik	Assemblersyntax	Operanden-grösse	Erlaubte Adressierungsarten		Bedingungs-codes
			Quelle	Ziel	XNZVC
EXG	EXG Rx,Ry	32	Dn oder An	Dn oder An	- - - - -
LEA	LEA <ea>,An	32	Kontrolle	An	- - - - -
MOVE	MOVE <ea>,<ea>	8, 16, 32	Alle (1)	Daten änderbar	- * * 0 0
	MOVE <ea>,CCR	16	Daten	CCR	* * * * *
	MOVE <ea>,SR (2)	16	Daten	SR	* * * * *
	MOVE SR,<ea>	16	SR	Daten änderbar	- - - - -
	MOVE USP,An (2)	32	USP	An	- - - - -
	MOVE An,USP (2)	32	An	USP	- - - - -
MOVEA	MOVEA <ea>,An	16, 32	Alle	An	- - - - -
MOVEM	MOVEM <list>,<ea>	16, 32		Kontrolle änderbar –(An)	- - - - -
	MOVEM <ea>,<list>	16, 32	Kontrolle oder (An)+		- - - - -
MOVEP	MOVEP Dx,d(Ay)	16, 32	Dn	d(An)	- - - - -
	MOVEP d(Ay),Dx	16, 32	d(An)	Dn	- - - - -
MOVEQ	MOVEQ #d,Dn	32	#d (3)	Dn	- * * 0 0
NOP	NOP		PV → 2-PC		- - - - -
PEA	PEA <ea>	32	Kontrolle		- - - - -
SWAP	SWAP Dn	16	Dn		- - - - -

Tabelle 3.7 Datentransportbefehle

Bemerkungen:
(1) Bei Byteverarbeitung ist die Adressierungsart «Adressregister direkt» nicht erlaubt
(2) Privilegierte Operation
(3) Acht Bit der unmittelbaren Daten, die zu einem Langwortoperand vorzeichenerweitert werden

3.4.1.2 Benützung von MOVE mit den Stapeln

Der MOVE-Befehl kann auch benützt werden, um Daten von den Stapeln in den Speicher zu transportieren und umgekehrt. Das schliesst sowohl die Systemstapel (Überwachungsstapel und Benützerstapel) als auch benützerdefinierte Stapel ein. Durch die Anordnung der Stapel im Speicher mit Adresse 0 kann die Adressierungsart «Adressregister indirekt mit Vordekrement» verwendet werden, um Daten in den Stapel zu bringen.
Zum Beispiel bringt der Befehl

```
MOVE  D0, –(SP)
```

das tiefere Wort von D0 in den aktiven Systemstapel. Umgekehrt holt die Adressierart «Adressregister indirekt mit Nachinkrementierung» Daten vom Stapel; zum Beispiel holt der Befehl

```
MOVE  (SP)+, D0
```

das nächste Wort des aktiven Systemstapels und lädt es in das tiefere Wort von D0.

3.4.1.3 MOVEM, transportiere mehrere Register

Öfters wird die Aufgabe gestellt sein, den Inhalt mehrerer Register zu transferieren. Ein grundsätzliches Beispiel dazu ist die Sicherung einer Anzahl allgemein verwendbarer Register in den Stapeln während der Ausführung einer Subroutine, um die Subroutine damit reentrant zu machen. Eine Subroutine ist *reentrant*, wenn sie unterbrochen und durch das unterbrechende Programm wieder aufgerufen werden kann. Der Befehl MOVEM, transportiere mehrere Register, kann verwendet werden zum Transfer von bis zu 16 Registern (Datenregister D0 ... D7 und Adressregister A0 ... A7) zum oder vom Speicher.
Formate:

MOVEM <list>, <ea>	Transfer Register-zu-Speicher
MOVEM <ea>, <list>	Transfer Speicher-zu-Register

In beiden Fällen bedeutet <list> die Register, die transportiert werden sollen. Der Assembler erlaubt zwei Arten, Register zu «listen». Ein Weg ist die Auflistung individueller Registernamen, getrennt durch einen Schrägstrich (/).
Zum Beispiel transferiert der Befehl

```
MOVEM  D3/D4/D5/A1,$53F00
```

das tiefere Wort von D3, D4, D5 und A1 in die vier aufeinanderfolgenden Worte, die mit derAdresse $53F00 beginnen. (In diesem Fall werden die Register in der Ordnung gespeichert, wie sie im Befehl aufgeführt sind; das ist jedoch nicht immer der Fall, wie wir noch sehen werden.)
Wenn die Registerliste aufeinanderfolgende Daten oder Adressregister enthält, erlaubt es der Assembler, nur das erste und letz-

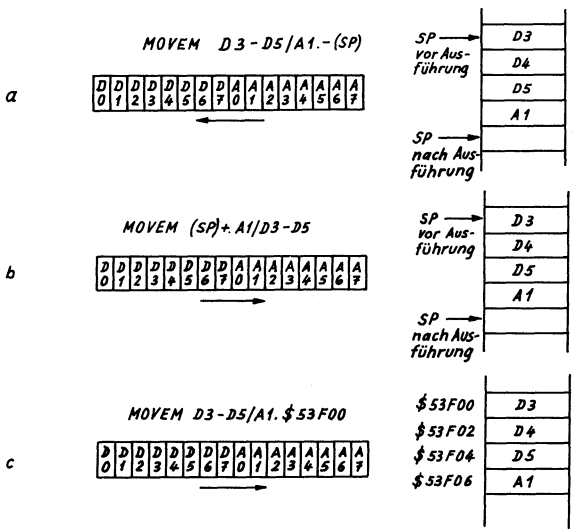


Bild 3.8
Beispiele zum Befehl MOVEM
a) Datenordnung mit Vordekrementierung
b) Datenordnung mit Nachinkrementierung
c) Datenordnung mit absoluter Adressierung

te Register aufzuführen, getrennt durch einen Bindestich (-). So könnte das vorhergehende Beispiel geschrieben werden als

```
MOVEM  D3-D5/A1,$53F00
```

Der MOVEM-Befehl transferiert immer Registerinhalte zu und vom Speicher in einer vorbestimmten Sequenz, unabhängig davon, wie sie in der Registerliste geordnet sind. Bei der Adressierungsart «Adressregister indirekt mit Vordekrementierung» werden die Register in der Ordnung A7 bis A0, dann D7 bis D0 transferiert. Demgegenüber werden für alle Steuerarten und für die Adressierungsart «Adressregister indirekt mit Nachinkrementierung» Register in der umgekehrten Ordnung, D0 bis D7, dann A0 bis A7 transferiert. Diese Unterschiede erlauben das Bilden von Stapeln und Listen in der einen Richtung und den Zugriff zu ihnen in der entgegengesetzten Richtung. Bild 3.8 zeigt einige Beispiele dazu.

3.4.1.4 Adresstransferbefehle (MOVEA, LEA, PEA)

Der MC 68000 verfügt über drei Befehle, die speziell für den Transfer von Adressen entworfen wurden. Zwei dieser Befehle, transportiere Adresse (MOVEA) und lade effektive Adresse (LEA) sind ähnlich und können leicht durch die Programmierer verwechselt werden. Beide veranlassen das Laden einer Adresse in ein Adressregister; während aber LEA die *effektive Adresse* des referenzierten Operanden lädt (ein Speicherplatz), lädt MOVEA den *Inhalt* des referenzierten Operanden (ein Speicherplatz, ein Register oder ein unmittelbarer Wert) und nimmt an, dass er eine Adresse darstellt.
LEA behandelt immer eine 32-Bit-Adresse, während MOVEA sowohl 16-Bit-Wort-Adressen als auch 32-Bit-Doppelwort-Adressen behandeln kann. Bild 3.9 zeigt zwei Beispiele des LEA-Befehls.

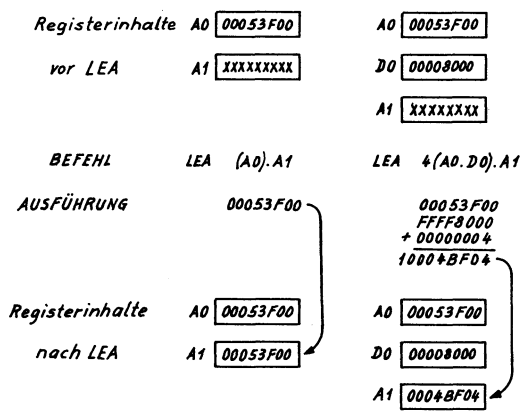


Bild 3.9
Beispiele zum Befehl LEA

Wie man sehen kann, sind LEA und MOVEA äusserst praktische Befehle. Wenn ein Programm die Berechnung verschiedener Adressen in verschiedenen Befehlen erfordert, muss die Adresse mit LEA nur einmal berechnet werden und in einem Adressregister abgelegt werden. Später kann jeder Bezug zu den adressierten Operanden mit der «Adressregister indirekt»-Adressierungsart gemacht werden. Das spart nicht nur Programmierungsaufwand, sondern benötigt auch weniger Speicherplatz und erlaubt eine schnellere Ausführung des Programms. Warum ist das so? Weil die Adressierungsart «Adressregister indirekt» keine speicherplatzbenötigenden Erweiterungsworte zu einem Befehl erfordert. Die Ausführungszeit zur Berechnung der Adresse ist mit dieser Art vier bis acht Zyklen schneller als mit den Adressierungsarten «Adressregister indirekt mit Verschiebung» und «Absolut- oder Programmzähler relativ».
Der Befehl MOVEA ist nützlich beim Zugriff zu Adressen, die im Speicher gespeichert sind. Zum Beispiel erhalten wir bei einer verbundenen Liste im Speicher, in der jeder Knoten mit einem Zeiger auf den nächsten Knoten beginnt, die Adresse des zweiten Knotens mit

```
MOVEA.L  LIST, A0
```

und die Adresse des dritten und der folgenden Knoten erhalten wir mit

```
MOVEA.L  (A0), A0
```

Zu erwähnen ist noch, dass ein MOVEA-Befehl, dessen Quellenoperand ein unmittelbares Label ist, einem LEA-Befehl entspricht. Das bedeutet, dass MOVEA.L # LABEL, A0 und LEA LABEL, A0 äquivalente Befehle sind. Dazu ist zu sagen, dass der MOVEA-Befehl 20 Zyklen für die Ausführung benötigt, währenddem der LEA-Befehl nur 12 Zyklen benötigt, so dass in diesem Fall dem LEA-Befehl der Vorzug zu geben ist.
Der letzte der drei Adresstransferbefehle, PEA (Push Effective Address) Eintragen der effektiven Adresse, ist ähnlich wie LEA, doch transferiert er berechnete effektive Adressen statt den Inhalt adressierter Speicherzeilen. Mit PEA wird *die Adresse in den aktiven Systemstapel* eingetragen (Benützerstapel oder Überwachungstapel). Der PEA-Befehl kann verwendet werden für die Übergabe von Parametern an eine Subroutine, indem die Adresse eines Parameters oder die Startadresse mehrerer hintereinander folgender Parameter in den Stapel eingetragen wird. Zum Beispiel kann die Eintrage- und Aufrufoperation mit der folgenden Befehlssequenz ausgeführt werden:

```
PEA   PARAM
JSR   SUBER
```

Weil der JSR-Befehl eine 4-Byte-Rückkehradresse in den Stapel schreibt, nachdem der PEA-Befehl seine 4-Byte-Adresse in den Stapel eingetragen hat, muss zur Parameteradresse mittels Überspringen der Rückkehradresse zugegriffen werden, wie mit dem Befehl

```
MOVEA.L  4(SP), A0
```

Mit dem Zurückholen der Parameteradresse aus dem Stapel bereinigt die Subroutine den Stapel mittels Verschiebung der Rückkehradresse um ein Doppelwort höher in den Speicher und Nachführung des Stapelzeigers. Beide Aufgaben können mit einem Befehl ausgeführt werden:

```
MOVE.L
```

3.4.1.5 Der MOVE-Befehl

Wie im Kapitel 1 erwähnt wurde, können am MC 68000 sowohl ältere synchrone 8-Bit-Peripheriebausteine als auch neuere asynchrone 16-Bit-Bausteine angeschlossen werden. Der MC 68000 verfügt über separate Steuerleitungen für jeden Typ von Peripheriebausteinen.

Leser, die 8-Bit-Systeme programmiert haben, wissen, dass die angeschlossenen Peripheriebausteine normalerweise über Register verfügen, die eine Anzahl aufeinanderfolgender Byte im Speicher belegen.

Der Befehl MOVEP, *transportiere periphere Daten,* ist bestimmt zum Transfer von Information zwischen einem MC-68000-Datenregister und einem angeschlossenen 8-Bit-Peripheriebaustein in Paketen von zwei oder vier Byte. Im MC-68000-System müssen 8-Bit-Peripheriebausteine entweder an die höheren 8 Bit des Datenbus angeschlossen werden (Linien D8 ... D15) oder an die tieferen 8 Bit des Datenbus (Linien D0 ... D7). Der MOVEP-Befehl verkehrt mit der Peripherie mit der höheren Hälfte des Bus durch die Verwendung von gerade numerierten Adressen und mit der Peripherie auf der tieferen Hälfte des Bus unter Verwendung von ungerade numerierten Adressen. In einer Speicherabbildung würden diese Peripheriebausteine abwechselnd aufeinanderfolgende gerade Byte oder aufeinanderfolgende ungerade Byte belegen.

Zwei-Byte-Transfers werden mittels Spezifikation eines Wortoperanden (MOVEP oder MOVEP.W) und 4-Byte-Transfers mittels eines Doppelwortoperanden (MOVEP.L) gemacht. Peripheriebausteine werden unter Verwendung der Adressierart «Register indirekt mit Verschiebung» adressiert. Bild 3.10 zeigt zwei Beispiele des MOVEP-Befehls – einen Doppelworttransfer mit einer geraden Adresse und einen Worttransfer mit einer ungeraden Adresse. Zu bemerken ist, dass der MOVEP-Befehl

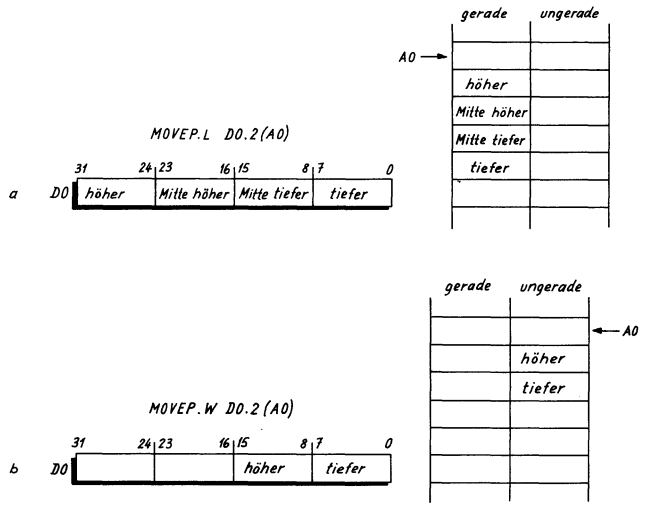


Bild 3.10
Byte-Transfers mittels MOVEP
a) Doppelworttransfer mit gerader Adresse
b) Worttransfer mit ungerader Adresse

der einzige MC-68000-Befehl ist, der die Benützung einer ungeraden Adresse mit einem Wort- oder einem Doppelwortoperanden erlaubt!
Die Ausführungszeit des MOVEP-Befehls hängt davon ab, ob Daten zu oder von asynchronen oder synchronen Peripheriebausteinen transferiert werden. Ein Register-zu-Speicher-Transfer benötigt bei asynchronen Peripheriebausteinen 18 Zyklen (Worttransfer) oder 28 Zyklen (Doppelworttransfer), während ein Speicher-zu-Register-Transfer 16 Zyklen bei Worttransfer oder 24 Zyklen bei Doppelworttransfer benötigt. Transfers zu oder von synchronen Peripherieschaltungen werden etwas mehr Zeit benötigen, weil der MC 68000 mit einem Takt synchronisieren muss, der nur $^1/_{10}$ des Systemtakts ist. Die Kapitel 6 und 8 werden dazu nähere Erläuterungen enthalten.

3.4.1.6 MOVEQ, transportiere schnell

Weil Programmierer öfters mit kleinen Konstanten operieren müssen, versahen die Entwickler des MC 68000 diesen mit drei sogenannt «schnellen» Befehlen: transportiere schnell, addiere schnell und subtrahiere schnell. Diese Befehle erlauben die Spezifikation einer kleinen Konstante im Operationswort. Der erste dieser Befehle, transportiere schnell (MOVEQ, move quick), kann einen spezifizierten, ein Byte langen Wert um das Vorzeichen erweitern auf 32 Bit und in ein Datenregister laden. Weil die Konstante 8 Bit lang ist, kann jeder ganzzahlige Wert zwischen −128 und +127 in ein Datenregister transferiert werden.
Der MOVEQ-Befehl belegt nur ein Wort im Speicher und benötigt vier Zyklen zur Ausführung. Im Gegensatz dazu benötigt der Befehl «transportiere unmittelbar» (MOVE.L #d_8, Dn)

zwei Worte im Speicher und 20 Zyklen zur Ausführung. Die meisten Assembler, eingeschlossen derjenige von Motorola, nützen diese Sparmöglichkeiten aus, indem sie einen geeigneten «transportiere unmittelbar»-Befehl als MOVEQ interpretieren und den entsprechenden Objektcode erzeugen.

3.4.1.7 SWAP und EXG, Registervertausch und Registeraustausch

Diese zwei ähnlichen Befehle haben einen ganz verschiedenen Verwendungszweck. Der Befehl SWAP, *vertausche Registerhälften,* vertauscht die höheren 16 Bit eines 32-Bit-Datenregisters mit den tieferen 16 Bit. Dieser Befehl gestattet den Zugriff zum Inhalt des oberen Wortes eines Registers und ist notwendig, weil Wortoperationen immer mit dem tieferen Wort ausgeführt werden. Ähnlich kann SWAP verwendet werden für den Zugriff zu den höheren zwei Byte eines Datenregisters. SWAP allein wird zum mittleren höheren Byte zugreifen; ein SWAP plus ein Rotierbefehl wird zum höchstwertigen Byte des Datenregisters zugreifen.

Der Registeraustauschbefehl (EXG) tauscht den *gesamten Inhalt von zwei Registern* aus. Er kann drei Formate haben:

EXG Dx,Dy – Austausch zweier Datenregister;
EXG Ax,Ay – Austausch zweier Adressregister;
EXG Dx,Ay – Austausch eines Datenregisters und eines Adressregisters.

3.4.1.8 NOP, keine Operation

Der Befehl «keine Operation» (NOP) wird normalerweise nur während der Programmentwicklung verwendet. Er führt keine Operationen durch. Er ändert weder Statusbit noch Register (mit Ausnahme des Programmzählers), noch Speicherzellen, aber er erfüllt die nützliche Funktion der Platzreservierung im Speicher.

Programmierer verwenden den NOP-Befehl häufig in einem Quellenprogramm, um Platz zu reservieren für später einzufügende Befehle. Weil jeder NOP-Befehl nur ein Wort im Speicher belegt, sind mindestens zwei NOP (besser sind drei) am Ort, wo der Platz reserviert werden soll, einzufügen.

NOP-Befehle können auch in Objektprogrammen eingefügt werden, um Befehle zu ersetzen, die entfernt wurden, so dass das Programm nicht neu assembliert werden muss. In diesem Fall sollte man für jedes Wort des entfernten Befehls $4E71 einfügen, das ist der hexadezimale Wert des NOP-Befehls.

3.4.2 Befehle für ganzzahlige Arithmetik

Der MC 68000 kann zwei binäre Operanden addieren, subtrahieren, multiplizieren, dividieren und vergleichen. Er kann auch einen einzelnen, spezifischen Operanden löschen, prüfen, das

Mnemonik	Assemblersyntax	Operanden-grösse	Erlaubte Adressierungsarten		Bedingungs-codes
			Quelle	Ziel	XNZVC
ADD	ADD <ea>,Dn	8, 16, 32	Alle (1)	Dn	* * * * *
	ADD Dn,<ea>	8, 16, 32	Dn	änderbar	* * * * *
ADDA	ADDA <ea>,An	16, 32	All	An	- - - - -
ADDI	ADDI #d,<ea>	8, 16, 32	#d	Daten änderbar	* * * * *
ADDQ	ADDQ #d,<ea>	8, 16, 32	#d (2)	änderbar (1)	* * * * *
ADDX	ADDX Dy,Dx	8, 16, 32	Dn	Dn	* * * * *
	ADDX -(Ay),-(Ax)	8, 16, 32	-(An)	-(An)	* * * * *
CLR	CLR <ea>	8, 16, 32	Daten änderbar		- 0 1 0 0
CMP	CMP <ea>,Dn	8, 16, 32	All (1)	Dn	- * * * *
CMPA	CMPA <ea>,An	16, 32	All	An	- * * * *
CMPI	CMPI #d,<ea>	8, 16, 32	#d	Daten änderbar	- * * * *
CMPM	CMPM (Ay)+,(Ax)+	8, 16, 32	(An)+	(An)+	- * * * *
DIVS	DIVS <ea>,Dn	16	Daten	Dn	- * * * 0
DIVU	DIVU <ea>,Dn	16	Daten	Dn	- * * * 0
EXT	EXT Dn	16, 32	Dn		- * * 0 0
MULS	MULS <ea>,Dn	16	Daten	Dn	- * * 0 0
MULU	MULU <ea>,Dn	16	Daten	Dn	- * * 0 0
NEG	NEG <ea>	8, 16, 32	Daten änderbar		* * * * *
NEGX	NEGX <ea>	8, 16, 32	Daten änderbar		* * * * *
SUB	SUB <ea>,Dn	8, 16, 32	Alle (1)	Dn	* * * * *
	SUB Dn,<ea>	8 16, 32	Dn	änderbar	* * * * *
SUBA	SUBA <ea>,An	16, 32	Alle	An	- - - - -
SUBI	SUBI #d,<ea>	8, 16, 32	#d	Daten änderbar	* * * * *
SUBQ	SUBQ #d,<ea>	8, 16, 32	#d (2)	änderbar (1)	* * * * *
SUBX	SUBX Dy,Dx	8, 16, 32	Dn	Dn	* * * * *
	SUBX -(Ay),-(Ax)	8, 16, 32	-(An)	-(An)	* * * * *
TAS	TAS <ea>	8	Daten änderbar		- * * 0 0
TST	TST <ea>	8, 16, 32	Daten änderbar		- * * 0 0

Bemerkungen: (1) Bei Byteoperationen ist die Adressierungsart «Adressregister direkt» nicht gestattet
(2) Unmittelbarer Operand im Wertebereich 1 bis 8

Tabelle 3.8 Befehle für ganzzahlige Arithmetik

Vorzeichen erweitern und negieren (2er-Komplement). Die Befehle für diese Operationen sind in der Tabelle 3.8 zusammengestellt.

3.4.2.1 Addierbefehle

Es bestehen fünf Befehle zur Addition von binären Zahlen. Der erste, «addiere binär» (ADD) addiert zwei Byte-, Wort- oder Doppelwortoperanden. Weil diese Operanden als Datenwerte betrachtet werden, muss einer in einem Datenregister sein, der andere kann im Speicher sein, in einem Adressregister (sofern nicht Byteoperanden addiert werden sollen) oder einem andern Datenregister. Der ADD-Befehl kann alle fünf Bedingungscodes beeinflussen, wie folgt:

1. Übertrag (C) wird gesetzt, wenn das Resultat nicht im Bestimmungsoperand Platz hat; sonst ist C gelöscht.
2. Überlauf (V) wird gesetzt, wenn zwei Zahlen mit gleichen Vorzeichen (beide positiv oder beide negativ) addiert werden und das Resultat den Bereich des 2er-Komplements der Operanden überschreitet, was zum Wechsel des Vorzeichenbit führt; sonst ist V gelöscht.
3. Null (Z) wird gesetzt, wenn das Resultat null ist; sonst ist Z gelöscht.
4. Negativ (N) wird gesetzt, wenn das Vorzeichenbit des Resultats logisch 1 ist; sonst ist N gelöscht.
5. Erweiterung (X) wird in den gleichen Zustand gesetzt wie der Übertrag (C).

Bei den ADD-Befehlen ist der Status der V- und N-Flags nur von Bedeutung, wenn Werte mit Vorzeichen addiert werden. Wenn der Bestimmungsoperand ein *Adressregister* ist, werden die Bedingungscodes *nicht* beeinflusst. Der Assembler erkennt diese Form des Additionsbefehls als Variante, «addiere Adresse» (ADDA) genannt.
Der ADD-Befehl wird eingesetzt zur Addition zweier Byte-, Wort- oder Doppelwortoperanden, sofern sich mindestens einer der Operanden in einem Datenregister befindet. Viele Anwendungen verlangen jedoch bei der Addition mehrfache Genauigkeit, oder es befinden sich beide Operanden im Speicher. Für diese Anwendungen verfügt der MC 68000 über den Befehl ADDX (erweiterte Addition). Damit kann der Inhalt zweier Datenregister oder zweier Speicherplätze addiert werden. Der ADDX-Befehl beeinflusst die C-, V-, N- und X-Flags gleich wie der ADD-Befehl. Das Z-Flag jedoch wird bei ADDX gelöscht, wenn das Resultat ungleich 0 ist, andernfalls wird Z nicht beeinflusst. Diese Charakteristik ist sehr praktisch bei Operationen mit erhöhter Genauigkeit, weil Z den Null/Nicht-null-Status einer *ganzen* Additionsoperation zeigt und nicht nur gerade den der letzten Teiloperation.
Wenn sich die Operanden im Datenregister befinden, geht dem ADDX-Befehl normalerweise ein ADD-Befehl voraus. Zum Beispiel addiert die folgende Sequenz einen ganzzahligen 64-Bit-Wert in D0 und D1 zu einem andern ganzzahligen 64-Bit-Wert in D2 und D3:

```
ADD.L   D0, D2      Addiere die 32 tieferen Bit
ADDX.L  D1, D3      Addiere die 32 höheren Bit
```

Wenn sich die Operanden im Speicher befinden, müssen vor der Additionsoperation X gelöscht und Z gesetzt werden (es sei daran erinnert, dass Z gesetzt bleibt, wenn jede nachfolgende Addition als Resultat null ergibt). Speicherplatz-zu-Speicherplatzadditionen verlangen immer vordekrementierte Adressierung, das

heisst, dass die Adressregister zuerst auf die höheren Bit, Worte oder Doppelworte im Speicher hinweisen und nachher auf die tieferen. Wenn zum Beispiel A0 und A1 auf zwei 64-Bit-Operanden im Speicher zeigen, können diese Operanden mit der folgenden Sequenz addiert werden:

```
MOVE    # 4,CCR      Setze Z = 1, alle andern Bit = 0
ADDX.L  - (A0),-(A1) Addiere die 32 tieferen Bit
ADDX.L  - (A0),-(A1) Addiere die 32 höheren Bit
```

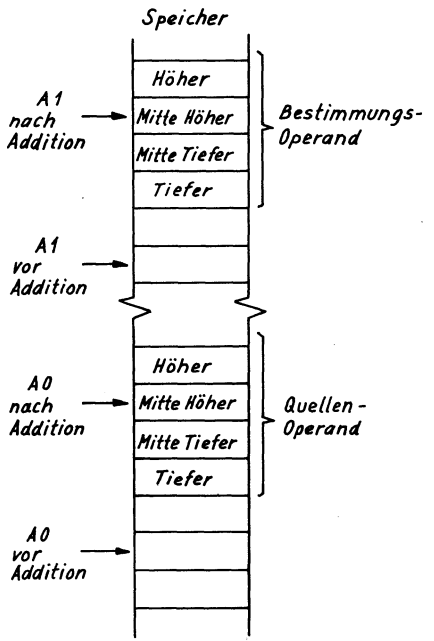


Bild 3.11
Addition zweier 64-Bit-Operanden im Speicher

Bild 3.11 zeigt die Anordnung der Operanden im Speicher und wie die Zeiger bei der Additionsoperation beeinflusst werden. Die letzten zwei Additionsbefehle, «addiere unmittelbar» (ADDI) und «addiere schnell» (ADDQ) werden verwendet, um einen konstanten Wert zu einem adressierten Operanden zu addieren. Bei ADDI kann die Konstante ein Bit, Wort oder Doppelwort sein und der Befehl belegt zwei bis fünf Worte im Speicher. Bei ADDQ kann die Konstante nur einen Wert zwischen 1 und 8 haben, der Befehl belegt aber auch nur ein bis drei Worte im Speicher. Im weiteren kann ADDQ verwendet werden zur Addition eines Werts zu einem Adressregister, während ADDI das nicht kann. ADDQ ersetzt den Inkrementbefehl in bisherigen 8-Bit-Mikroprozessoren.

3.4.2.2 Subtraktionsbefehle

Der MC 68000 verfügt über die äquivalenten Subtraktionsbefehle wie die Additionsbefehle. Drei dieser Befehle, «subtrahiere binär» (SUB), «subtrahiere unmittelbar» (SUBI) und «subtrahiere schnell» (SUBQ) beeinflussen die Bedingungscode wie folgt:

1. Der Übertrag (C) wird gesetzt, wenn die Subtraktion einen Entlehnwert (borrow) benötigt, was anzeigt, dass das Resultat nicht im Bestimmungsoperanden Platz hat; sonst ist C gelöscht.
2. Überlauf (V) ist gesetzt, wenn zwei Zahlen mit ungleichem Vorzeichen (eines positiv, das andere negativ) subtrahiert werden und das Resultat den Bereich des 2er-Komplements der Werte überschreitet; sonst ist V gelöscht.
3. Null (Z) wird gesetzt, wenn das Resultat null ist; sonst ist Z gelöscht.
4. Negativ (N) ist gesetzt, wenn das Vorzeichenbit des Resultats logisch 1 ist; sonst ist N gelöscht.
5. Erweiterung (X) wird gleich gesetzt wie der Übertrag (C).

Der Subtraktionsbefehl für mehrfache Genauigkeit «subtrahiere mit Erweiterung» (SUBX) beeinflusst C, V, N und X in der gleichen Art, löscht aber Z, wenn das Resultat nicht null ist; sonst wird Z nicht beeinflusst. Der fünfte Subtraktionsbefehl «subtrahiere Adresse» (SUBA) beeinflusst keine Flags.

3.4.2.3 Negierbefehle

Mit zwei subtraktionsähnlichen Befehlen kann das 2er-Komplement eines Byte-, Wort- oder Doppelwortoperanden im Speicher oder in einem Datenregister erzeugt werden. Diese Befehle «negiere» (NEG) und «negiere erweitert» (NEGX) erzeugen das 2er-Komplement durch Subtraktion des Operanden von null. Der NEG-Befehl beeinflusst die Bedingungscode in der gleichen Weise wie der SUB-Befehl; während aber hier ein Operand null ist, können über die Bedingungen, die die Flags setzen, eindeutigere Aussagen gemacht werden. Für NEG gilt:

1. Übertrag (C) und Negativ (N) werden gesetzt, wenn der adressierte Operand eine positive, von null verschiedene Zahl ist; sonst wird C und N gelöscht.
2. Überlauf (V) wird gesetzt, wenn der adressierte Operand einen Wert hat von $80 (Byte), $8000 (Wort) oder $80 000 000 (Doppelwort); sonst ist V gelöscht.
3. Null (Z) wird gesetzt, wenn der adressierte Operand null ist; sonst ist Z gelöscht.
4. Erweiterung (X) wird in der gleichen Weise gesetzt wie der Überlauf (C).

Der NEGX-Befehl beeinflusst in der gleichen Art C, V, N und X, löscht aber Z nur, wenn das Resultat nicht null ist. X wird nicht beeinflusst, wenn das Resultat null ist. Wie bereits beim ADDX-Befehl erklärt, zeigt damit Z den Status null/nicht-null

einer ganzen Operation mit mehrfacher Genauigkeit, und nicht nur gerade denjenigen der letzten Teiloperation.

3.4.2.4 Multiplizier- und Dividierbefehle

Der MC 68000 verfügt über zwei Multiplizierbefehle: «multipliziere mit Vorzeichen» (MULS) und «multipliziere ohne Vorzeichen» (MULU). Diese Befehle multiplizieren zwei Wortoperanden und speichern das 32-Bit-Produkt in einem Datenregister. Zahlen, die länger sind als 16 Bit können ebenfalls mit MULS und MULU multipliziert werden. Wir werden davon Beispiele im Kapitel 4 sehen, wo zwei 32-Bit-Zahlen multipliziert werden, und zwar sowohl für Werte mit wie auch ohne Vorzeichen.

Der MC 68000 verfügt auch über zwei Divisionsbefehle: «dividiere mit Vorzeichen» (DIVS) und «dividiere ohne Vorzeichen» (DIVU). Diese Befehle dividieren einen 32-Bit-Dividenden (in einem Datenregister) durch einen 16-Bit-Divisor (im Speicher oder in einem Datenregister) und legen den 16-Bit-Quotienten und den 16-Bit-Rest je in die untere und obere Hälfte eines Datenregisters. Beim Versuch, durch null zu dividieren, wird der MC 68000 einen Trap erzeugen (im Kapitel 7 beschrieben).

Eine Division, mit oder ohne Vorzeichen, wird die Bedingungscodes wie folgt beeinflussen:

1. Übertrag (C) wird immer gelöscht.
2. Überlauf (V) wird gesetzt, wenn Divisionsüberlauf angezeigt wird; sonst ist V gelöscht.
3. Null (Z) wird gesetzt, wenn der Quotient null ist; sonst ist Z gelöscht. Der Zustand von Z ist in Überlauffällen nicht definiert.
4. Negativ (N) wird gesetzt, wenn der Quotient negativ ist (für DIVS), oder wenn das höchstwertige Bit des Quotienten gesetzt wird (für DIVU); sonst ist N gelöscht. Der Zustand von N ist in Überlauffällen nicht definiert.
5. Erweiterung (X) wird nicht beeinflusst.

In Überlauffällen setzt der MC 68000 das V-Flag und beendet die Operation, ohne den Divisor oder Dividenden zu beeinflussen. Überlauf wird dann erzeugt, wenn der Dividend so viel grösser ist als der Divisor, dass der Quotient nicht in 16 Bit Platz hat. Für eine Division ohne Vorzeichen muss der Dividend mindestens 65536mal grösser sein als der Divisor, damit Überlauf erreicht wird. Für eine Division mit Vorzeichen muss der Quotient +32767 oder –32768 übersteigen, damit Überlauf erreicht wird. Es ist möglich, ein Programm zu schreiben, das immer einen gültigen Quotienten und Rest ergibt, unabhängig davon, ob ein Überlauf entsteht. Ein solches Programm wird im Kapitel 4 vorgestellt.

3.4.2.5 Vorzeichenerweiterung (EXT)

Der MC 68000 ermöglicht Operationen mit verschiedenen Datenlängen, und zwar mit einem Befehl genannt «Vorzeichenerweiterung» (EXT). Dieser Befehl erweitert das Vorzeichenbit (das höchstwertige Bit) eines Wertes in einem Datenregister von einem Byte zu einem Wort oder von einem Wort zu einem Doppelwort, wie Bild 3.12 zeigt. Damit ermöglicht der EXT-Befehl die Ausführung von Operationen wie zum Beispiel der Addition eines Byte zu einem Wort oder der Multiplikation eines Wortes mit einem Byte.

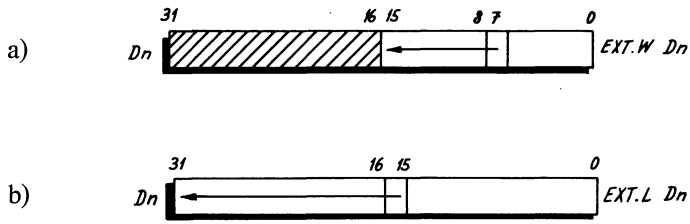


Bild 3.12
Funktion des Befehls EXT
a) Vorzeichenerweiterung von Byte zu Wort
b) Vorzeichenerweiterung von Wort zu Doppelwort

3.4.2.6 Löschbefehl (CLR)

Ein weiterer Befehl dieser Gruppe, «lösche» (CLR), setzt das adressierte Byte, Wort oder Doppelwort auf null. Er kann verwendet werden zum Löschen eines Datenregisters oder eines Speicherplatzes, aber nicht für ein Adressregister. (Der Fall, dass ein Adressregister gelöscht werden soll, ist nicht häufig, aber für diesen Fall kann mit dem Befehl SUBA.L An, An diese Operation ausgeführt werden.)
Bei zeitkritischen Anwendungen ist es nützlich zu wissen, dass CLR gegenüber dem entsprechenden MOVE #0, <ea> dann schneller ist, wenn das tiefere Byte oder tiefere Wort eines Datenregisters gelöscht werden soll. Wenn alle 32 Bit eines Datenregisters gelöscht werden sollen, ist der MOVEO #0, Dn zwei Zyklen schneller als CLR.L DN. In den meisten Fällen benötigt das Löschen eines Speicherplatzes mit MOVE.x #0, <ea> (wobei x = B, W oder DW) die gleiche Zeit wie CLR.x <ea>. Tatsächlich wird bei Benützung der indirekten Adressierungsart mit Vordekrementierung der Befehl MOVE.x #0,–(An) den Speicherplatz zwei Zyklen schneller löschen als CLR.x –(An).

3.4.2.7 Vergleichbefehle

Die meisten Programme arbeiten Befehle nicht hintereinander ab, wie sie im Speicher gespeichert sind, sondern beinhalten Sprünge, Verzweigungen, Schlaufen, Subroutinenaufrufe und andere Bedingungen, die die Programmausführung von einem Platz im Speicher zu einem andern transferieren können. Die Befehle, die diese Transfers veranlassen, werden später in die-

sem Kapitel beschrieben, wenn wir die Programmsteuerbefehle des MC 68000 behandeln. Hier werden nur die Vergleichsbefehle diskutiert, die normalerweise verwendet werden, um die Bedingungscode zu steuern, nach welchen die Programmsteuerbefehle ihre Entscheide bezüglich Transfer/kein-Transfer machen.

Die vier Vergleichsbefehle des MC 68000 arbeiten sehr ähnlich wie Subtraktionsbefehle. Das bedeutet, dass jeder dieser Befehle einen Quellenoperanden von einem Bestimmungsoperanden subtrahiert und dabei die Bedingungs-Flags je nach Resultat setzt (siehe Tabelle 3.9). Im Gegensatz zu den Subtraktionsbefehlen wird bei den Vergleichbefehlen das *Resultat der Subtraktion nicht gespeichert.* Ihr einziger Grund ist die Steuerung der Bedingungscode für Entscheide der nachfolgenden Programmsteuerbefehle.

Bedingung	X	N*	Z	V*	C
Quelle < Bestimmung	–	0	0	0/1	0
Quelle = Bestimmung	–	0	1	0	0
Quelle > Bestimmung	–	1	0	0/1	1

*Zutreffend, wenn Zahlen im Zweierkomplement verglichen werden.

Tabelle 3.9
Resultate der Vergleichsbefehle

Der Vergleichsbefehl (CMP) vergleicht einen Quellenoperanden mit einem Byte-, Wort- oder Doppelwortoperanden in einem Datenregister. Wort- oder Doppelwortadressen können unter Verwendung einer Variante von CMP, genannt «vergleiche Adresse» (CMPA), mit Adressregistern verglichen werden. Der Befehl «vergleiche unmittelbar» (CMPI) vergleicht ein Byte, Wort oder Doppelwort mit einem Bestimmungsoperanden. Der Befehl «vergleiche Speicher» (CMPM), vergleicht zwei Operanden im Speicher unter Verwendung indirekter Adressierung mit Nachinkrementierung. Dieser CMPM-Befehl wird vor allem dann eingesetzt, wenn Zeichenfolgen verglichen werden müssen, wie in einem Beispiel später in diesem Kapitel gezeigt wird (Beispiel 3.3).

3.4.2.8 Vergleich mit Null

Wie in Kap. 3.4.2.3 beschrieben, sind Negierbefehle NEG und NEGX eigentlich Subtraktionsbefehle, die spezialisierte Aufgaben übernehmen. Sie subtrahieren einen Operanden von null. Vergleichbar dazu verfügt der MC 68000 über einen spezialisierten Vergleichsbefehl, «prüfe einen Operanden (TST), der einen Operanden mit null vergleicht. Wie die Vergleichsbefehle subtrahiert auch TST den Operanden von null und setzt oder löscht die Bedingungs-Flags anhand des Resultates, speichert aber das Resultat nicht ab. Die Bedingungscodes werden durch TST wie folgt beeinflusst:

1. Übertrag (C) und Überlauf (V) werden immer gelöscht.
2. Null (Z) wird gesetzt, wenn der adressierte Operand null ist; sonst wird Z gelöscht.
3. Negativ (N) wird gelöscht, wenn der adressierte Operand eine positive Zahl ist; sonst ist N gelöscht.
4. Erweiterung (X) wird nicht beeinflusst.

3.4.2.9 Prüfe und setze einen Operanden (TAS)

Der Befehl «prüfe und setze einen Operanden» (TAS) arbeitet grundsätzlich gleich wie der TST-Befehl (er vergleicht den Operanden mit null und setzt oder löscht die Bedingungscode je nach Resultat), wobei TAS das *höchstwertige* Bit des Operanden bedingungslos immer setzt. Im weiteren arbeitet TAS nur mit Byteoperanden, so dass er also Bit 7 eines Byte setzen wird. Trotz der ähnlichen Arbeitsweisen haben TST und TAS sehr verschiedene Funktionen. Wie wir im vorderen Abschnitt gesehen haben, wird TST verwendet, um herauszufinden, ob ein Operand den Wert null hat. TAS hingegen wird vor allem gebraucht zur Statusprüfung eines Flags im Speicher und zum Setzen dieses Flags. Das ist vor allem bei Multi-tasking-Anwendungen äusserst praktisch, um den verschiedenen Aufgaben (Tasks) Speicherplatz zuzuweisen. In Multiprocessing-Anwendungen kann er verwendet werden als Zugriffschutz zu Speicherbereichen, die bestimmten Prozessoren zugeordnet sind.

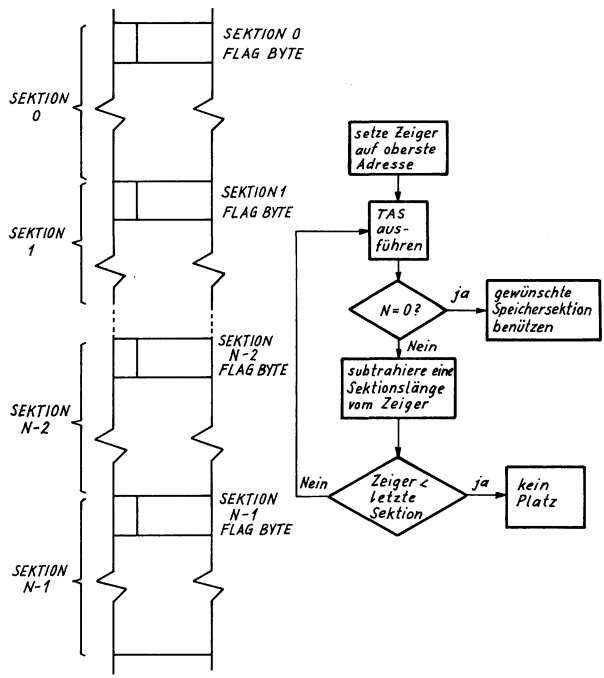


Bild 3.13
Speicherzuweisung mit dem Befehl TAS

Bild 3.13 zeigt die Verwendung von TAS in Multi-tasking-Anwendungen. Das Beispiel zeigt einen Speicherbereich, der in N Sektionen unterteilt wurde und gibt ein einfaches Flussdiagramm eines Algorithmus, der zur Lokalisierung der nächsten verfügbaren Sektion verwendet werden kann. Dieser Algorithmus benötigt zwei Adressregister, eines zur Aufnahme eines Zeigers, der auf die zu prüfende Sektion zeigt, und ein anderes zur Aufnahme eines Zeigers, der auf die letzte Sektion (Sektion 0) zeigt. Das Programm für diesen Algorithmus schliesst einige schon beschriebene Befehle ein wie MOVEA oder LEA (zur Initialisierung der Testzeiger), SUBA (zur Dekrementierung des Testzeigers) und CMPA (zum Vergleichen der zwei Zeiger). Es sind auch bedingte Verzweigungsbefehle verwendet, die unter den Programmsteuerbefehlen behandelt werden.
In einer Multiprocessing-Anwendung erlaubt TAS einem Prozessor die Interpretation eines Prüfbyte (mittels der Bedingungscodes) und das Setzen einer 1 in das höchstwertige Bit des Byte. Falls der Speicher besetzt ist, kann das Programm die Abfrage aufrechterhalten, bis er frei wird. Die folgende Routine übernimmt diese Aufgabe:

```
MFREE   TAS   TEST    Prüfe und setze das Byte TEST.
        BNE   MFREE   Wenn TEST nicht = 0, prüfe
                      weiter.
          .
          .           (Prozessor Programmbefehle)
          .
        CLR.B TEST    Lösche TAS Byte.
```

Es ist wichtig, zu wissen, dass TAS der einzige MC-68000-Befehl ist, der einen *nicht unterteilbaren* «lese-ändere-schreibe»-Zyklus ausführen kann. Das verunmöglicht jegliche Beeinflussung durch einen anderen Prozessor, sobald die TAS-Operation eingeleitet wurde.

3.4.3 Logische Befehle

Es bestehen sieben logische Befehle, dargestellt in Tabelle 3.10. Die Basisbefehle in dieser Gruppe sind «logisch und» (AND), «exklusiv oder» (EOR), «oder» (OR). Diese drei Befehle können mit Byte-, Wort- oder Doppelwortoperanden arbeiten. Einer dieser Operanden muss sich in einem Datenregister befinden. Der zweite Operand kann für den AND- und OR-Befehl im Speicher, einem Datenregister oder einem Adressregister sein, für den EOR-Befehl nur im Speicher oder in einem Datenregister. EOR kann nicht mit Adressregistern operieren.
Ein weiterer Befehl, «logisches Komplement» (NOT), kann das Einerkomplement eines Datenregisters oder eines Speicherplatzes erzeugen. Mit NOT können Operanden ohne Vorzeichen, mit NEG oder NEGX vorzeichenbehaftete Operanden komple-

Mnemonik	Assemblersyntax	Operandengrösse	Erlaubte Adressierungsarten		Bedingungscodes
			Quelle	Ziel	XNZVC
AND	AND <ea>,Dn	8, 16, 32	Daten	Dn	- * * 0 0
	AND Dn,<ea>	8, 16, 32	Dn	änderbar	- * * 0 0
ANDI	ANDI #d,<ea>	8, 16, 32	#d	Daten änderbar	- * * 0 0
	ANDI #d,SR (1)	8, 16	#d	SR	* * * * *
EOR	EOR Dn,<ea>	8, 16, 32	Dn	Daten änderbar	- * * 0 0
EORI	EORI #d,<ea>	8, 16, 32	#d		- * * 0 0
	EORI #d,SR (1)	8, 16	#d	SR	* * * * *
NOT	NOT <ea>	8, 16, 32		Daten änderbar	- * * 0 0
OR	OR <ea>,Dn	8, 16, 32	Daten	Dn	- * * 0 0
	OR Dn,<ea>	8, 16, 32	Dn	änderbar	- * * 0 0
ORI	ORI #d,<ea>	8, 16, 32	#d	Daten änderbar	- * * 0 0
	ORI #d,SR (1)	8, 16	#d	SR	* * * * *

Bemerkung: (1) Wenn die Operandengrösse Byte ist, werden nur die tieferen 8 Bit des Statusregisters beeinflusst. Wenn die Operandengrösse Wort ist, werden alle 16 Bit des Statusregisters beeinflusst und der Befehl ist privilegiert.

Tabelle 3.10 Logische Befehle

mentiert werden. Variationen der AND-, OR- und EOR-Befehle erlauben die Verwendung von Konstanten als Quellendaten. Diese Variationen, «und unmittelbar» (ANDI), «exklusiv- oder unmittelbar» (EORI) und «oder unmittelbar» (ORI) arbeiten mit Speicher- oder Datenregisteroperanden jeder Länge. Sie sind auch anwendbar für Operationen mit dem Statusregister oder mit den Bedingungscode. Operationen mit dem Statusregister (SR) sind privilegiert.

3.4.4 Schiebe- und Rotierbefehle

Der MC 68000 verfügt über vier Schiebebefehle und vier Rotierbefehle. Tabelle 3.11 zeigt diese Befehle, und Bild 3.14 zeigt ihre Funktionsweise. Wie in Tabelle 3.11 dargestellt, verfügt jeder Befehl über drei Varianten: zwei, die mit einem Datenregister operieren (Byte, Wort oder Doppelwort), und eine, die mit dem Speicher arbeitet (nur Worte).
Wenn die Operation mit einem Datenregister ausgeführt wird, kann der Schiebe- oder Rotierwert spezifiziert werden mit dem Inhalt eines andern Datenregisters (Wert = 0 bis 63, wobei 0 einen Wert von 64 erzeugt), oder als unmittelbarer Wert zwischen 1 und 8. Ein Wortoperand im Speicher kann nur um eine Bitposition geschoben oder rotiert werden.

3.4.4.1 Schiebebefehle

Zahlen mit Vorzeichen können geschoben werden unter Verwendung der Befehle «arithmetisch schieben links» (ASL) und «arithmetisch schieben rechts» (ASR). ASR schützt das Vorzeichen des Operanden durch Reproduktion des Vorzeichens während der ganzen Schiebeoperation. Bei ASL wird das Vor-

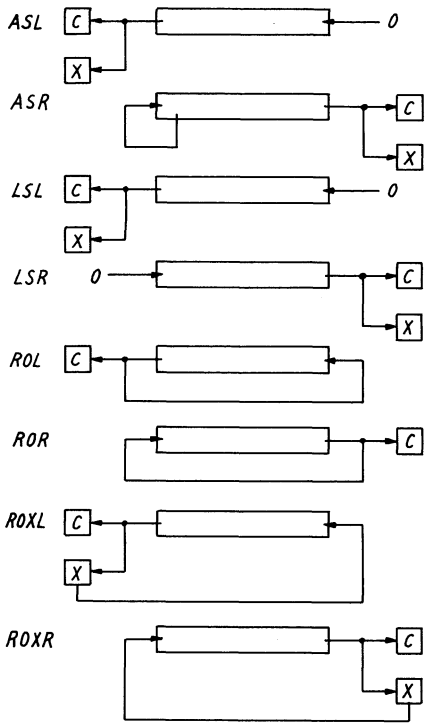


Bild 3.14
Funktionsweise der Schiebe- und Rotierbefehle

zeichenbit nicht geschützt, aber das Überlaufbit (V) wird gesetzt, wenn das Vorzeichenbit geändert wird.
Zahlen ohne Vorzeichen lassen sich unter Verwendung der Befehle «logisch schieben links» (LSL) und «logisch schieben rechts» (LSR) bearbeiten. Bei allen vier Befehlen werden Bit, die aus dem Operanden geschoben werden, in den Übertrag (C) und die Erweiterung (X) der Bedingungscode übernommen. Zusätzlich zur Anwendung dieser Befehle in allgemeinen Datenmanipulationen führen diese Schiebebefehle auch schnelle Multiplizier- und Dividieroperationen durch: Jedes Linksschieben bedeutet eine Multiplikation des Operanden mit zwei und jedes Rechtsschieben eine Division des Operanden durch zwei.

3.4.4.2 Rotierbefehle

Bei allen vier Rotierbefehlen werden Bit ausserhalb des Operanden in den Übertrag geschrieben. Für den Befehl «rotiere links» (ROL) und «rotiere rechts» (ROR) werden die Bit, die an einem Ende des Operanden herausrotiert werden, auf der entgegengesetzten Seite des Operanden wieder eingeschrieben. Bei «rotiere links mit Erweiterung» (ROXL) und «rotiere rechts mit Erwei-

terung» (ROXR) werden die an einem Ende des Operanden herausrotierten Bit in das Erweiterungs-Flag (X) und in den Übertrag (C) geschrieben, und der vorherige Wert von X wird in das entgegengesetzte Ende des Operanden eingeschrieben.
Die «rotiere mit Erweiterung»-Befehle verfügen über Möglichkeiten, die bis jetzt nicht zur Verfügung standen: die Fähigkeit des Zugriffes zu den drei höherwertigen Byte in einem Datenregister. Wir erinnern uns, dass alle Byteoperationen mit dem tiefstwertigen Byte eines Datenregisters ausgeführt werden.
Wie kann man nun mit dem zweiten Byte (das «tiefere mittlere») eines Registers operieren? Das ist möglich, indem man dieses Byte in die tiefstwertige Position bringt, unter Verwendung der Befehle ROL #8, Dn oder ROR #8, Dn. So kann auch auf das «höhere mittlere» und das höchstwertige Byte eines Datenregisters zugegriffen werden. Das höhere mittlere mit einem SWAP-Befehl, das höchstwertige mit einem ROL.L #8, DN. Aufeinanderfolgend kann zu den höheren drei Byte zugegriffen werden (wie zum Beispiel in Zeichenfolgen) unter Ausführung von drei Befehlen ROR.L #8, Dn.

Mnemonik	Assemblersyntax	Operandengrösse	Erlaubte Adressierungsarten		Bedingungscodes
			Quelle	Ziel	XNZVC
ASL	ASL Dx,Dy	8, 16, 32	Dn (1)	Dn	* * * * *
	ASL #d,Dn	8, 16, 32	#d (2)	Dn	* * * * *
	ASL <ea>	16		Speicher änderbar	* * * * *
ASR	ASR Dx,Dy	8, 16, 32	Dn (1)	Dn	* * * * *
	ASR #d,Dn	8, 16, 32	#d (2)	Dn	* * * * *
	ASR <ea>	16		Speicher änderbar	* * * * *
LSL	LSL Dx,Dy	8, 16, 32	Dn (1)	Dn	* * * 0 *
	LSL #d,Dn	8, 16, 32	#d (2)	Dn	* * * 0 *
	LSL <ea>	16		Speicher änderbar	* * * 0 *
LSR	LSR Dx,Dy	8, 16, 32	Dn (1)	Dn	* 0 * 0 *
	LSR #d,Dn	8, 16, 32	#d (2)	Dn	* 0 * 0 *
	LSR <ea>	16		Speicher änderbar	* 0 * 0 *
ROL	ROL Dx,Dy	8, 16, 32	Dn (1)	Dn	– * * 0 *
	ROL #d,Dn	8, 16, 32	#d (2)	Dn	– * * 0 *
	ROL <ea>	16		Speicher änderbar	– * * 0 *
ROR	ROR Dx,Dy	8, 16, 32	Dn (1)	Dn	– * * 0 *
	ROR #d,Dn	8, 16, 32	#d (2)	Dn	– * * 0 *
	ROR <ea>	16		Speicher änderbar	– * * 0 *
ROXL	ROXL Dx,Dy	8, 16, 32	Dn (1)	Dn	* * * 0 *
	ROXL #d,Dn	8, 16, 32	#d (2)	Dn	* * * 0 *
	ROXL <ea>	16		Speicher änderbar	* * * 0 *
ROXR	ROXR Dx,Dy	8, 16, 32	Dn (1)	Dn	* * * 0 *
	ROXR #d,Dn	8, 16, 32	#d (2)	Dn	* * * 0 *
	ROXR <ea>	16		Speicher änderbar	* * * 0 *

Bemerkungen: (1) Das Quellendatenregister enthält den Schiebewert.
Wert = 0 bis 63, wobei 0 eine Verschiebung von 64 erzeugt.
(2) Die Daten sind der Schiebewert 1 bis 8.

Tabelle 3.11 Schiebe- und Rotierbefehle

3.4.4.3 Schnellere Schiebe- und Rotieroperationen

Weil Worte im Speicher pro Operation nur um eine Bitposition geschoben oder rotiert werden können, dauert eine Schiebe- oder Rotieroperation um n Bit mindestens n-mal länger als eine 1-Bit-Schiebe- oder -Rotieroperation. Das Schieben oder Rotieren eines Worts im *Speicher* benötigt 9+ Zyklen, wobei «+» die Zeit für die Berechnung der effektiven Adresse angibt. Dadurch wird eine 2-Bit-Schiebeoperation 2 × 9+ Zyklen benötigen, usw.

Das Schieben oder Rotieren eines *Datenregisters* um n Bit benötigt 6 + 2n Zyklen, eine 1-Bit-Schiebeoperation also 8 Zyklen, eine 2-Bit-Schiebeoperation 10 Zyklen usw. Es ist klar, dass für einige Werte von n die Ausführungszeit beträchtlich gesenkt werden kann, indem man einen Speicheroperanden in ein Datenregister liest, das Register schiebt (oder rotiert) und dann das Resultat in den Speicher zurückschreibt. Das benötigt drei Befehle. Unter Verwendung von Tabellen über die Ausführungszeiten, die wir später kennenlernen, können wir die totale Ausführungszeit wie folgt berechnen:

Befehl	Ausführungszeit
MOVE <ea>, Dn	4+
ASL #n, Dn	6 + 2n
MOVE Dn, <ea>	5+
Totalzeit	(15+) + 2n

Zusammengefasst benötigt also eine n-Bit-Schiebe- oder -Rotieroperation n × 9+ Zyklen im Speicher und [(15+) + 2n] Zyklen in einem Datenregister. Von welchem Punkt an bringt es nun Vorteile, die Operation in einem Datenregister durchzuführen? Es ist klar, dass eine 1-Bit-Schiebeoperation sicher im Speicher durchgeführt werden muss (9+ Zyklen im Speicher gegenüber 17+ Zyklen in einem Register). Auch eine 2-Bit-Schiebeoperation sollte noch so durchgeführt werden (18+ Zyklen im Speicher gegenüber 19+ Zyklen in einem Register). Hingegen benötigt eine 3-Bit-Schiebeoperation 27+ Zyklen im Speicher, aber nur 21+ Zyklen in einem Datenregister! Schlussfolgerung: Wenn das Verschieben oder Rotieren im Speicher um mehr als 3 Bit-Positionen nötig ist, sollte die Operation in einem Datenregister vorgenommen werden.

3.4.5 Bitmanipulationsbefehle

Diese vier Befehle können den Zustand eines spezifizierten Bit in einem Datenregister oder einem Byte im Speicher prüfen. Diese in der Tabelle 3.12 zusammengefassten Befehle speichern den Zustand des spezifizierten Bit im Bedingungscode, Flag Null (Z): wenn das Bit 0 ist, wird Z = 1; wenn das Bit 1 ist, wird Z = 0.

Mnemonik	Assemblersyntax		Operandengrösse	Erlaubte Adressierungsarten		Bedingungscodes
				Quelle	Ziel	XNZVC
BTST	BTST	Dn,<ea>	8, 32	Dn	Daten, ausgenommen unmittelbar	– – * – –
	BTST	#d,<ea>	8, 32	#d		– – * – –
BSET	BSET	Dn,<ea>	8, 32	Dn		– – * – –
	BSET	#d,<ea>	8, 32	#d		– – * – –
BCLR	BCLR	Dn,<ea>	8, 32	Dn	Daten änderbar	– – * – –
	BCLR	#d,<ea>	8, 32	#d		– – * – –
BCHG	BCHG	Dn,<ea>	8, 32	Dn		– – * – –
	BCHG	#d,<ea>	8, 32	#d		– – * – –

Tabelle 3.12 Befehle für Bitmanipulationen

Drei der Bitmanipulationsbefehle ändern das Bit unbedingt, wie folgt:

Befehl	*Durchgeführte Operation mit dem Bit*
BTST (Bit prüfen)	Bit wird nicht beeinflusst
BSET (Bit prüfen und setzen)	Bit wird auf logisch 1 gesetzt
BCLR (Bit prüfen und löschen)	Bit wird auf logisch 0 gesetzt
BCHG (Bit prüfen und wechseln)	Zustand des Bit wird umgekehrt

3.4.6 BCD-Befehle

Im Zusammenhang mit den arithmetischen Befehlen wurde erwähnt, dass der MC 68000 über drei Befehle verfügt, mit denen Operationen mit BCD-Werten ausgeführt werden können. Alle diese Befehle (Tabelle 3.13) arbeiten mit bytelangen Daten, wobei ein Byte immer zwei BCD-Werte mit 4 Bit enthält. Im weiteren schliessen die BCD-Befehle wie auch die erweiterten Binärarithmetikbefehle das X-Bit in die Operationen ein und wechseln das Z-Bit dann, wenn ein Resultat generiert wird, das verschieden von null ist. In diesem Fall muss vor Ausführung der BCD-Befehle das X-Bit mit 0 und das Z-Bit mit 1 initialisiert werden. Am einfachsten kann dies mit dem Befehl MOVE #4, CCR erreicht werden.

Mnemonik	Assemblersyntax		Operandengrösse	Erlaubte Adressierungsarten		Bedingungscodes
				Quelle	Ziel	XNZVC
ABCD	ABCD	Dy, Dx	8	Dn	Dn	* U* U*
	ABCD	–(Ay),–Ax)	8	–(An)	–(An)	* U* U*
SBCD	SBCD	Dy,Dx	8	Dn	Dn	* U* U*
	SBCD	–(Ay),–(Ax)	8	–(An)	–(An)	* U* U*
NBCD	NBCD	<ea>	8		Daten änderbar	* U* U*

Tabelle 3.13 BCD-Befehle

3.4.6.1 BCD-Addition (ABCD) und -Subtraktion (SBCD)

Mit den Befehlen «addiere dezimal mit Erweiterung» (ABCD) und «subtrahiere dezimal mit Erweiterung» (SBCD) können dezimale Additionen und Subtraktionen mit den tieferwertigen Byte von zwei Datenregistern oder mit zwei Byte im Speicher ausgeführt werden. Die Befehle ABCD und SBCD beeinflussen die fünf Bedingungscodebit wie folgt:

1. Der Übertrag (C) wird gesetzt, wenn ABCD einen Übertrag erzeugt oder SBCD einen Entlehnwert benötigt, sonst ist C gelöscht.
2. Überlauf (V) und Negativ (N) sind für beide Befehle nicht definiert.
3. Null (Z) wird gelöscht, wenn das Resultat nicht gleich null ist, sonst bleibt Z unverändert. Bei Mehrbyteoperationen zeigt dadurch Z den Status der gesamten Operation und nicht nur gerade den der letzten Byte.
4. Die Erweiterung (X) wird wie der Übertrag C gesetzt.

Obschon die BCD-Befehle eine gewisse Ähnlichkeit mit den erweiterten Binärarithmetikbefehlen aufweisen, bedeutet die Tatsache, dass die BCD-Befehle auf Byteoperationen beschränkt sind doch, dass bei der Programmierung Unterschiede beachtet werden müssen. Zum Beispiel wird es offensichtlich mehr Befehle erfordern, um BCD-Mehrbyteadditionen oder -subtraktionen auszuführen, als für die gleichen Binäroperationen nötig wären, da bei binären Mehrbytezahlen die Wort- und Doppelwortkombinationen verwendet werden können.
Weniger augenscheinlich ist die Tatsache, dass in den meisten Fällen die Datenregister auf die Ausführung von Additionen und Subtraktionen mit Zweidigit-BCD-Werten beschränkt sind, und zwar wegen des Zugriffs auf das mittlere Byte von Datenregistern. Dieses Byte müsste zuerst in die tiefstwertige Byteposition rotiert werden, wobei wiederum zu beachten ist, dass die dafür benötigten Befehle ROR, ROL, ROXR und ROXL immer das Z-Bit beeinflussen und damit den Zwischenstatus null der Mehrbyte-BCD-Operationen zerstören! Wenn man also nicht einen speziellen Schutz der CCR-Werte vor und nach der Rotieroperation vorsieht, sollten Mehrbyte-BCD-Operationen eher mit Werten im Speicher als in Datenregistern vorgenommen werden.

Wenn Additionen und Subtraktionen von Mehrbyte-BCD-Operanden im Speicher ausgeführt werden, müssen diese Operanden, wie bei Mehrbyte-Binäroperanden, in der Ordnung höher nach tiefer gespeichert sein (siehe wieder Bild 1.1). Diese Anordnung ist verständlich, wenn die Adressierungsart im Speicher für ABCD- und SBCD-Befehle betrachtet wird.

Im folgenden Beispiel werden mit der Befehlsequenz

```
MOVE  #4, CCR
ABCD  -(A0), -(A1)
ABCD  -(A0), -(A1)
ABCD  -(A0), -(A1)
ABCD  -(A0), -(A1)
```

zwei 8-Digit-BCD-Zahlen (4 Byte) im Speicher addiert. Bild 3.15 zeigt, wie diese Zahlen gespeichert sind und die Zeiger A0 (Quelle) und A1 (Ziel) durch die Additionssequenz verändert werden.

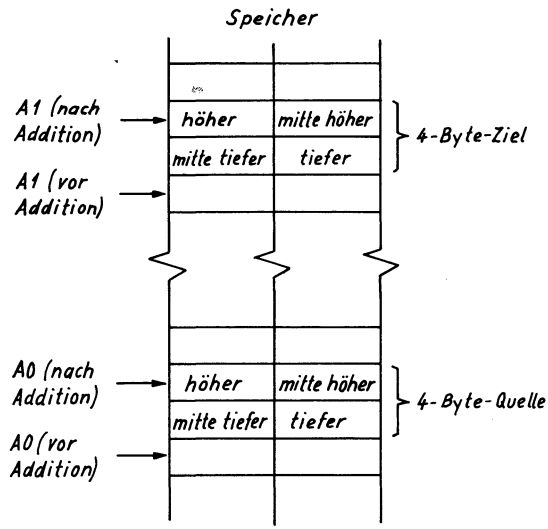


Bild 3.15
Addition von zwei BCD-Zahlen mit je 4Byte

3.4.6.2 BCD-Negierbefehl (NBCD)

Der NBCD-Befehl subtrahiert den adressierten Byteoperanden (in einem Datenregister oder im Speicher) und das Erweiterungsbit (X) von null. Wenn X = 0 ist, wird das Zehnerkomplement erzeugt; wenn X = 1 ist, das Neunerkomplement.

3.4.7 Programmsteuerbefehle

Wie bei der Behandlung der Vergleichbefehle erwähnt, werden Programmbefehle im Speicher fortlaufend abgelegt, jedoch in den seltensten Fällen in dieser Reihenfolge ausgeführt. Auch das einfachste Programm verfügt über Verzweigungen, Sprünge und Subroutinenaufrufe, die die Ablaufsequenz ändern. Die Programmsteuerbefehle (Tabelle 3.14) ermöglichen dem MC 68000 den Wechsel der Programmausführung von einem Teil

des Speichers in einen anderen. Diese Befehlsgruppe kann in drei Kategorien unterteilt werden: *bedingte, unbedingte* und *Rückkehrbefehle.*

3.4.7.1 Bedingte Befehle

Die drei ersten Eintragungen in Tabelle 3.14 sind bedingte Befehle für den MC 68000. Ihr Operationsmodus hängt ab vom Zustand eines oder mehrerer Flags im Bedingungscoderegister. Anders als in den vorhergehenden Befehlstabellen dieses Kapitels zeigt die Tabelle 3.14 nicht die Mnemonik für diesen Befehlstyp, sondern die symbolische Form Bcc, DBcc und Scc, wobei cc die geprüfte Bedingung darstellt. Die cc-Anhänge sind in Tabelle 3.15 dargestellt. Der Bcc-Befehl akzeptiert die Bedingung «immer wahr» (T) und «immer falsch» (F) nicht, mit den DBcc-und Scc-Befehlen hingegen können alle 16 Bedingungen geprüft werden. Die 14 bedingten Verzweigungsbefehle (Bcc) des MC 68000 sind die gleichen wie beim MC 6800. Mit diesen Befehlen verzweigt die Programmsteuerung bei erfüllter Bedingung zum Befehl im Speicherplatz mit der Adresse (PC)+Verschiebung. (PC = Programmzähler)
Wenn die Bedingung nicht erfüllt ist, wird der Programmablauf mit dem nächsten Befehl weitergeführt. Der Wert im PC entspricht dem Speicherplatz des Bcc-Befehls + 2. Die Verschiebung ist ein ganzzahliger Zweierkomplementwert, der der Anzahl Byte zwischen dem PC-Wert und dem Speicherplatz des Labels entspricht. Wenn der Operand ein Label ist (was normalerweise der Fall ist), wird der Assembler die Verschiebung berechnen.

Mnemonik	Assemblersyntax	Operanden-grösse	Erlaubte Adressierungsarten		Bedingungs-codes
			Quelle	Ziel	XNZVC
Bedingte Befehle					
Bcc	Bcc <label>	8, 16	If cc, then PC + d → PC		- - - - -
DBcc	DBcc Dn,<label>	16	If cc, then Dn – 1 → Dn; if Dn # –1, then PC + d → PC		- - - - -
Scc	Scc <ea>	8	If cc, then 1s → (ea); 0s → (ea)	Daten veränderbar	- - - - -
Unbedingte Befehle					
BRA	BRA <label>	8, 16	PC + d → PC		- - - - -
BSR	BSR <label>	8, 16	PC → –(SP); PC + d → PC		- - - - -
JMP	JMP <ea>		<ea> → PC	Steuerung	- - - - -
JSR	JSR <ea>		PC → –(SP); <ea> → PC	Steuerung	- - - - -
Rückkehrbefehle					
RTR	RTR		(SP) + → CCR; (SP) + → PC		* * * * *
RTS	RTS		(SP) + → PC		- - - - -

Tabelle 3.14 Befehle für die Programmsteuerung

*Zweierkomplement-Arithmetik		
Symbole:	△ =	UND
	□ =	ODER
	○ =	EXKLUSIV ODER

Anhang «cc»	Bedingung	Trifft zu wenn
EQ	Gleich	Z = 1
NE	Nicht gleich	Z = 0
MI	Minus	N = 1
PL	Plus	N = 0
*GT	Grösser als	Z △ (N ○ V) = 0
*LT	Kleiner als	N ○ V = 1
*GE	Grösser oder gleich	N ○ V = 0
*LE	Kleiner oder gleich	Z □ (N ○ V) = 1
HI	Höher als	C △ Z = 0
LS	tiefer oder gleich	C □ Z = 1
CS	Übertrag gesetzt	C = 1
CC	Übertrag gelöscht	C = 0
*VS	Überlauf	V = 1
*VC	Kein Überlauf	V = 0
T	Immer wahr	
F	Immer falsch	

Tabelle 3.15
Bedingungsprüfungen

Wenn der Befehl die Form

```
BNE *+10
```

hat, spezifiziert der Operand den Wert der Verschiebung (in diesem Fall dezimal 10) in Byte. Die Bcc-Befehle können ein Wort oder zwei Worte lang sein. Mit der Form

```
Bcc.S
```

wird der Assembler einen Einwortbefehl mit einer relativen, vorzeichenbehafteten 8-Bit-Verschiebung, die im Operationswort eingebettet ist, generieren. In dieser Form kann sich der angezielte Verzweigungsbefehl bis zu 128 Byte höher oder tiefer im Speicher befinden, als das Bcc-Operationswort plus zwei. Wird der Anhang «.S» weggelassen, erzeugt der Assembler einen Zweiwortbefehl mit einer relativen, vorzeichenbehafteten 16-Bit-Verschiebung im zweiten Wort. In dieser Form kann sich der angezielte Verzweigungsbefehl bis zu 32 KByte höher oder tiefer im Speicher befinden als das Bcc-Operationswort plus zwei (das Verschiebungswort). Wenn also der Bcc-Befehl am Speicherplatz N beginnt, gestattet die Form *Bcc.S* einen Verzweigungsbereich zwischen N+$80 und N–$7E, die Form *Bcc* einen solchen zwischen N+$8000 und N–$7FFE.
Hier einige Beispiele bedingter Verzweigbefehle:
1.) Die Sequenz

```
ADD  D0, DI
BCS  TOOBIG
```

verzweigt zum Label TOOBIG, wenn die Addition einen Übertrag ausserhalb des tiefern Wortes in D1 ergibt.
2.) Die Sequenz

```
SUB  D0,D1
BEQ  ZERO
```

verzweigt zum Label ZERO, wenn die Subtraktion im tiefern Wort von D1 Null ergibt.
3.) Zum blossen Test, ob die tiefern Wörter von D0 und D1 gleich sind, wird, ohne Register zu beeinflussen, anstelle eines Subtraktionsbefehls besser ein Vergleichbefehl verwendet.
Die Sequenz

```
CMP  D0,D1
BEQ  ZERO
```

verzweigt zum Label ZERO, wenn die tiefern Worte in D0 und D1 gleich sind.
4.) Einige Tests erfordern die Wahl zwischen zwei verschiedenen Bcc-Befehlen, je nachdem ob das Resultat einer Operation von Werten mit oder ohne Vorzeichen geprüft werden soll. Zur Erläuterung nehmen wir an, dass zum Label D1MORE verzweigt werden soll, wenn das tiefere Wort in D1 grösser ist als das tiefere Wort in D0.
Folgende Sequenzen sind zu verwenden:

```
CMP  D0,D1      für Werte von D0 und D1
BHI  D1MORE     ohne Vorzeichen

CMP  D0,D1      für vorzeichenbehaftete Werte
BGT  D1MORE     von D0 und D1
```

Die bedingten Verzweigbefehle werden häufig als letzter Befehl in einer Schlaufe eingesetzt, um diese verlassen zu können, wenn eine bestimmte «cc»-Bedingung eingetreten ist. Das Beispiel 3 zeigt diese Methode mit einem Programm, das einen ausgewählten Speicherbereich nach einem spezifizierten Wortwert absucht. Die Start- und Endadressen im Speicher befinden sich in A0 beziehungsweise A1, und der gesuchte Wert befindet sich im tiefern Wort von D0.
Dieses Programm verwendet eine Schlaufe, in der der Wert, auf den A0 zeigt, mit dem Wert in D0 verglichen wird. Wenn der gesuchte Wert gefunden ist, verzweigt BEQ.S DONE den Mikroprozessor zu DONE, wo A0 dekrementiert wird. (Das ist nötig, weil A0 immer nachher inkrementiert wurde, und dann am Schluss auf ein Wort *nach* dem Speicherplatz des verglichenen Werts zeigt.) Wenn kein gleicher Wert gefunden wird, prüft

CMPA.L A0, A1 auf die Bereichsgrenze und springt auf LOOP zurück, wenn A0 kleiner oder gleich A1 ist (wahr wenn C = 0, da BCC.S als Abschlussbefehl verwendet wird).

Programmbeispiel 3.1

```
* DIESES PROGRAMM PRUEFT, OB EIN AUS-
* GEWÄHLTER SPEICHERBEREICH EINEN
* SPEZIFIZIERTEN WORTWERT ENTHAELT. VOR
* DER AUSFUEHRUNG MUESSEN A0 UND A1 DIE
* ANFANGS- UND ENDADRESSEN DES BEREICHS
* ENTHALTEN. DAS TIEFERE WORT VON D0 MUSS
* DEN WERT, DER GESUCHT WIRD, ENTHALTEN.
* AM SCHLUSS, WENN DER WERT GEFUNDEN
* WURDE, IST Z = 1 UND A0 ENTHAELT DIE
* ADRESSE, WO DER WERT GEFUNDEN WURDE.
* FALLS DER WERT NICHT GEFUNDEN WURDE,
* IST Z = 0 UND A0 = A1

        ORG     $2000
LOOP    CMP     (A0)+,D0    WERT GEFUNDEN?
        BEQ.S   DONE        JA, AUFHOEREN
        CMPA.L  A0,A1       GRENZE DES
                            BEREICHS?
        BCC.S   LOOP        NEIN, WEITERFAHREN
DONE    SUBA.L  #2,A0       FERTIG. A0 ANPASSEN
```

Leser, die 8-Bit-Mikroprozessoren programmiert haben, wissen gut, dass Schlaufen gewöhnlich mit einer Art Zähler, normalerweise einem Register, realisiert werden. Nach jedem Schlaufendurchgang wird der Zähler um 1 dekrementiert, und die Schlaufe wird dann verlassen, wenn der Zähler null wird. Diese Prozedur benötigt immer mindestens zwei Befehle: einen Dekrementbefehl und einen bedingten Verzweigbefehl. Mit dem MC 68000 kann diese Aufgabe kombiniert werden mit einer Anzahl von Prüf-, Dekrement- und Verzweigbefehlen (DBcc: test, decrement and branch).

Bei der Ausführung eines DBcc-Befehls fragt der MC 68000 die Bedingungscode ab, um herauszufinden, ob die spezifizierte Bedingung (irgendeine der 16 Bedingungen aus Tabelle 3.15) gesetzt ist. Falls dies der Fall ist, geht das Programm zum nächsten Befehl. Wenn die Bedingung nicht erfüllt ist, dekrementiert der MC 68000 das tiefere Wort eines spezifierten Datenregisters um eins. Wenn der Wert im Datenregister –1 erreicht hat, wird der nächste Befehl ausgeführt, andernfalls verzweigt der MC 68000 zum spezifizierten Label im Speicher. Zum besseren Verständnis dieses Ablaufs sei auf Bild 3.16 hingewiesen.

Es wird ausdrücklich darauf aufmerksam gemacht, dass mit dem einfachen Befehl DBcc wie zum Beispiel

```
BNE     D0, LOOP
```

die gleichen Operationen wie mit der Befehlssequenz

```
        BNE.S   NEXT
        SUBQ    #1,D0
        BPL     LOOP
NEXT    .
        .
        .
```

ausgeführt werden. Zum Wegfall von zwei Zeilen Quellencode kommt dazu, dass ein DBcc-Befehl im Speicher zwei Worte weniger benötigt als die äquivalente Befehlssequenz (zwei Worte für DBcc gegen vier Worte für die Sequenz). Ein DBcc-Befehl wird also normalerweise doppelt so schnell ausgeführt werden wie die entsprechende 3-Befehls-Sequenz. Ein DBcc-Befehl benötigt 10 Zyklen, wenn die Verzweigung ausgeführt wird, und 12 Zyklen, wenn nicht verzweigt wird. Demgegenüber benötigt die Sequenz bei ausgeführter Verzweigung 22 Zyklen, und 10

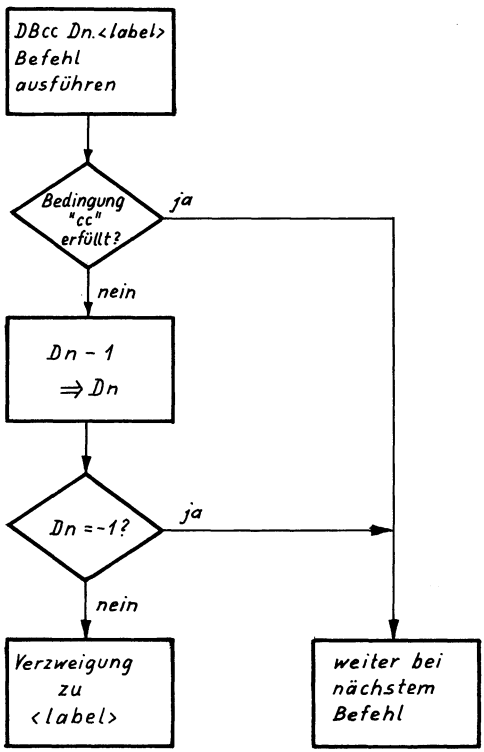


Bild 3.16
Wirkungsweise der Befehle DBcc

oder 22 Zyklen, wenn nicht verzweigt wird (je nachdem ob die «cc»-Bedingung erfüllt ist oder der Zähler auf –1 dekrementiert wurde).
Die Annahme, dass die DBcc-Befehle nur gerade als Bcc-mit-Zähler-Befehle eingesetzt werden können, wäre aber falsch. Es gibt vielmehr einige wichtige Unterschiede zwischen den DBcc- und den Bcc-Befehlen, die beachtet werden müssen:

1. Die DBcc-Befehle arbeiten in umgekehrter Art als die Bcc-Befehle, das heisst, dass die Bcc-Befehle verzweigen, wenn die Bedingung erfüllt ist, während die DBcc-Befehle *nicht verzweigen,* wenn die Bedingung *erfüllt* ist, und das Programm den nächsten Befehl ausführt.

2. Bei den DBcc-Befehlen führen zwei Wege aus der Schlaufe, da sie sowohl bei erfüllter Bedingung als auch beim Erreichen des Werts –1 im Zähler zum nächsten Befehl gehen. Daher besitzen die DBcc-Befehle auch die Charakteristik eines Befehls «mache bis gleich» (auf –1).

3. Ein Bcc-Befehl kann in einem Programm vorwärts und rückwärts verzweigen, ein DBcc-Befehl hingegen *nur rückwärts,* das heisst zu einer tieferen Speicheradresse. Die Verzweigungsmarke darf nicht mehr als 32 766 Byte ($7FFE) tiefer als der DBcc-Befehl liegen.

4. Bcc-Befehle können Ein- oder Zweiwortbefehle sein, DBcc-Befehle sind immer Zweiwortbefehle. Daher ist die Form DBcc.S illegal.

Wie bereits erwähnt, können die DBcc-Befehle mit allen 16 «cc»-Anhängen verwendet werden, also inklusive T und F. Der T-Anhang verlangt die Befehlsform

```
DBT     Dn, ‹Label›
```

die *immer* zum nächsten Befehl führt und daher nichts anderes ist als eine Zweiwort-Nulloperation! Der nützlichere F-Anhang gestattet es, die Bedingungsabfrage zu unterlassen und die Entscheidung verzweigen/nicht verzweigen allein auf den Zustand des Zählers abzustützen. Das Beispiel 3.2 zeigt, wie unter Verwendung des DBF-Befehls ein Datenblock im Speicher verschoben werden kann. Es ist zu beachten, dass der Zähler D0 mit dem Wert der Anzahl Doppelworte minus eins initialisiert wird, weil ja bis auf –1 dekrementiert wird. Wenn acht Doppelworte verschoben werden sollen, muss D0 mit dem Wort $0007 initialisiert werden. Dieses Programm dürfte vor allem Programmierer beeindrucken, die bereits Datenverschiebungen in einem 8-Bit-Mikroprozessor programmiert haben, denn das Programm besteht nur aus zwei Befehlen, besetzt lediglich drei Worte im Speicher und ist auch schnell. Der MOVE.L-Befehl

benötigt 22 Zyklen und der DBF-Befehl entweder 10 (falls verzweigt wird) oder 14 Zyklen (wenn nicht verzweigt wird).
Daher werden bei der Verschiebung von N Doppelworten 32N+4 Zyklen benötigt, für 100 Doppelworte also z.B. 3204 Zyklen oder 400,5 µs bei 8 MHz.

Programmbeispiel 3.2

```
* DIESES PROGRAMM KOPIERT EINEN DATEN-
* BLOCK VON EINEM TEIL DES SPEICHERS IN EINEN
* ANDERN. D0 ENTHAELT DIE ANZAHL DOPPEL-
* WORTE, DIE VERSCHOBEN WERDEN SOLLEN, -1.
* A0 ENTHAELT DIE URSPRUNGSADRESSE, A1 DIE
* ZIELADRESSE
        ORG     $2000
BLKMOV  MOVE.L  (A0)+,(A1)+     VERSCHIEBE EIN
                                DOPPELWORT
        DBF     D0,BLKMOVE      DURCHLAUFE
                                SCHLAUFE BIS
                                D0 + 1 BLOECKE
                                VERSCHOBEN
        END                     SIND.
```

Als einzige bedingte Befehle, die bis jetzt nicht diskutiert wurden, bleiben noch die «Setze gemäss Bedingung»-(Scc-)Befehle. Diese Befehle prüfen die spezifizierte «cc»-Bedingung (irgendeine aus Tabelle 3.15) und setzen im adressierten Byte alle Bit auf 1, falls die Bedingung erfüllt ist, beziehungsweise alle Bit auf 0, wenn die Bedingung nicht erfüllt ist. Da diese Befehle den Bedingungscode nicht beeinflussen, werden sie zur Bildung von Indikatoren eingesetzt, die nicht unmittelbar geprüft werden müssen, sondern später abgefragt werden können.

3.4.7.2 Such-Subroutine für ASCII-Zeichenfolgen

Zur Vertiefung des bisher vermittelten Stoffes betrachten wir ein Programmbeispiel, das eine gute Auswahl der bis jetzt diskutierten Befehle aufweist. Das Beispiel 3.3 ist eine Subroutine, die das erste Erscheinen einer ASCII-Zeichenfolge (die sogenannte Testzeichenfolge oder «test string») in einer anderen ASCII-Zeichenfolge (die sogenannte Hauptzeichenfolge oder «main string») im Speicher prüft. Das Beispiel hat nicht nur theoretischen Wert, sondern wird im Zusammenhang mit Textverarbeitung häufig verwendet.
Im Programm zeigt das Adressregister A0 auf die Hauptzeichenfolge (die Zeichenfolge, in der gesucht wird). In einer Textverarbeitungsanwendung ist die Testzeichenfolge wahrscheinlich ein Wort, ein Satz, ein Name, eine Telefonnummer oder etwas Ähnliches, zu dem für eine Verwendung mit einer

```
* DIESE SUBROUTINE SUCHT EINE ASCII-FOLGE IM
* SPEICHER («HAUPTFOLGE» GENANNT) NACH
* DEM VORHANDENSEIN EINER ANDERN ASCII-
* FOLGE («TESTFOLGE» GENANNT). DIE HAUPT-
* FOLGE WIRD MIT EINEM *-ZEICHEN BEENDET.
* VOR AUFRUF DER SUBROUTINE MUESSEN DIE
* STARTADRESSEN VON HAUPTFOLGE UND TEST-
* FOLGE IN DEN REGISTERN A0 UND A1 SEIN, UND
* DIE LAENGE DER TESTFOLGE IN BYTE MUSS IM
* DATENREGISTER D0 ENTHALTEN SEIN.
* DAS ERGEBNIS DER SUCHE WIRD IN A2
* GESCHRIEBEN. WENN DIE TESTFOLGE
* GEFUNDEN WIRD, ENTHAELT A2 IHRE ADRESSE
* IN DER HAUPTFOLGE. WENN DIE TESTFOLGE
* NICHT GEFUNDEN WIRD, ENTHAELT A2 NULL.
* A2 IST DAS EINZIGE BEEINFLUSSTE REGISTER.

          ORG       $1000
ASEARCH   MOVEM     D1/D3,-(SP)   SICHERE DATEN-
          MOVEM.L   A0/A3,-(SP)   REGISTER UND
                                  ADRESSREGISTER
                                  IM STAPEL
*
*         SUCHE ERSTES ZEICHEN DER TEST-
          FOLGE
*
          MOVE.B    (A1),D3       LIES ERSTES
                                  TESTZEICHEN IN
                                  D3
FIRST     SUBA.L    A2,A2         A2 = 0 FUER
                                  BEGINN
CHKEND    CMPI.B    #'*',(A0)     ENDE HAUPT-
                                  FOLGE
          BEQ.S     RETRN         JA. ZURUECK.
          CMP.B     (A0)+,D3      HAUPTZEICHEN=
                                  TESTZEICHEN?
          BNE.S     CHKEND        NEIN. WEITER-
                                  SUCHEN
*
*         ERSTES TESTZEICHEN GEFUNDEN,
          VERGLEICHE REST DER TESTFOLGE
*
          MOVE      D0,D1         BRINGE LAENGE
                                  DER TESTFOLGE
                                  IN D1.
          SUBQ      #2,D1         D1 = LAENGE -2.
          MOVEA.L   A1,A3         BRINGE ADR.
                                  'TESTFOLGE' IN
                                  A3.
```

```
        ADDQ.L   #1,A3          A3 ZEIGT AUF
                                ZWEITES TEST-
                                ZEICHEN.
        MOVEA.L  A0,A2          A2 = LAUFENDE
                                ADR. 'HAUPT-
                                FOLGE'
        SUBQ.L   #1,A2
LOOP    CMPI.B   #'*',(A0)      ENDE HAUPT-
                                FOLGE?
        BEQ.S    RETRN          WENN JA,
                                ZURUECK.
        CMPM.B   (A3)+,(A0+)    HAUPTZEICHEN =
                                TESTZEICHEN?
        BNE.S    FIRST          NEIN. WEITER-
                                FAHREN
        DBF      D1,LOOP        JA. VERGLEICH
                                WEITERFAHREN.
RETRN   MOVEM.L  (SP)+,A0/A3    REGISTER
                                ZURUECK-
                                SPEICHERN
        MOVEM    (SP)+,D1/D3
        END
```

Programmbeispiel 3.3
ASCII-Zeichen-Suchroutine

anschliessenden Operation zugegriffen werden soll. Der einzige weitere Parameter, der spezifiziert werden muss, ist die Länge der Testzeichenfolge. Dieser Wert in Byte wird im tiefern Wort des Datenregisters D0 eingeschrieben.
Das Resultat der Suche wird in das Adressregister A2 geschrieben. Wenn die Testzeichenfolge in der Hauptzeichenfolge liegt, wird A2 die Adresse dieser Hauptzeichenfolge enthalten. Falls die Testzeichenfolge nicht in der Hauptzeichenfolge ist, wird A2 null enthalten.
Die ASEARCH-Subroutine im Beispiel 3.3 beginnt mit der Verschiebung von zwei Datenregistern und zwei Adressregistern zum Systemstapel, so dass sie nach der Rückkehr aus der Subroutine unverändert sind. Der Rest der Subroutine besteht aus zwei Teilen. Im ersten Teil liest der MC 68000 das erste Zeichen der Testzeichenfolge in das Datenregister D3 und durchläuft dann eine Schlaufe (CHKEND), in der dieses Zeichen mit jedem Byte in der Hauptzeichenfolge verglichen wird. Das Zeichen in D3 wird auch mit dem Endezeichen (hier *) verglichen, um festzustellen, dass beim Durchsuchen der gesamten Hauptzeichenfolge kein gleicher Wert gefunden wurde.
Wenn das erste Zeichen der Testzeichenfolge irgendwo in der Hauptzeichenfolge gefunden wird, springt der MC 68000 in den untern Teil der Subroutine, in der die restliche Testzeichenfolge

mit der Hauptfolge verglichen wird. Für diesen Vergleich wird der Bytezählerwert der Testzeichenfolge in D1 geschrieben, dann 2 davon subtrahiert, weil der DBF-Befehl auf –1 prüft und weil das zweite Byte der Testfolge bearbeitet wird. An diesem Punkt wird die möglicherweise zutreffende Hauptzeichenfolgeadresse in A2 festgehalten. Die LOOP-Sequenz dieses Teils der Subroutine vergleicht den Rest der Testfolge und verzweigt zurück zu FIRST, wenn noch nicht die ganze Testfolge lokalisiert ist. Die Subroutine endet mit zwei MOVEM-Befehlen, um die gesicherten Register aus dem Stapel zurückzuholen, und einem RTS-Befehl, mit dem die Rückkehradresse geholt wird und damit die Kontrolle an das aufrufende Programm zurückgegeben wird.

3.4.7.3 Unbedingte Sprünge und Verzweigungen, Rückkehrbefehle

Wie beim früheren 8-Bit-Mikroprozessor MC 6800 hat Motorola auch den MC 68000 mit Sprung- und Subroutinenaufrufbefehlen je in Kurz- und Langformat ausgerüstet. Die Sprungbefehle werden «springe immer» (JMP) und «verzweige immer» (BRA) genannt. Die Subroutinenaufrufbefehle werden «springe zur Subroutine» (JSR) und «verzweige zur Subroutine» (BSR) genannt.

Das Langformat dieser Befehle, JMP und JSR, kann verwendet werden, um die Programmsteuerung *irgendwohin* in den 16-MByte-Speicherbereich zu transferieren, während das Kurzformat, BRA und BRS, beschränkt ist auf Verschiebungen *relativ* zu den Verzweigbefehlen. Wie die bedingten Verzweigbefehle (Bcc), können BRA und BSR sowohl für 8-Bit- als auch für 16-Bit-Verschiebungen verwendet werden, wobei die 8-Bit-Verschiebung mit dem Anhang .S angewählt wird (BRA.S oder BSR.S).

Alle vier Befehle veranlassen einen Transfer der Programmsteuerung durch das Laden einer neuen Adresse in den Programmzähler. Die Subroutinenaufrufbefehle JSR und BSR sichern selbstverständlich die Rückkehr des MC 68000 zu dem JSR und BSR folgenden Befehl, indem die Adresse dieses Befehls in den Stapel gerettet wird.

Im Gegensatz zu allen andern Stapeloperationen bringen die JSR- und BSR-Befehle zuerst das höhere Wort der Adresse in den Stapel und veranlassen damit die Speicherung der Rückkehradresse in der Ordnung tieferes Wort höheres Wort.
Der Befehl «Rückkehr von Subroutine» (RTS) holt die Rückkehradresse vom Stapel und lädt sie in den Programmzähler. Daher muss RTS der letzte ausgeführte Befehl jeder Subroutine sein. Zur Erläuterung von Subroutinenaufruf und -rückkehr betrachten wir ein Programm mit den zwei Befehlen:

Programm-zähler	Befehl		Kommentar
$A2000	JSR	$4EFE	Subroutinenaufruf
$A2004	MOVE	D0, D1	nächster Befehl

Bild 3.17 zeigt den Programmzähler und den Stapel zu drei Zeitpunkten: vor dem JSR-Befehl (3.17a), nach dem JSR-Befehl (3.17b) und nach der Ausführung des RTS-Befehls (3.17c).

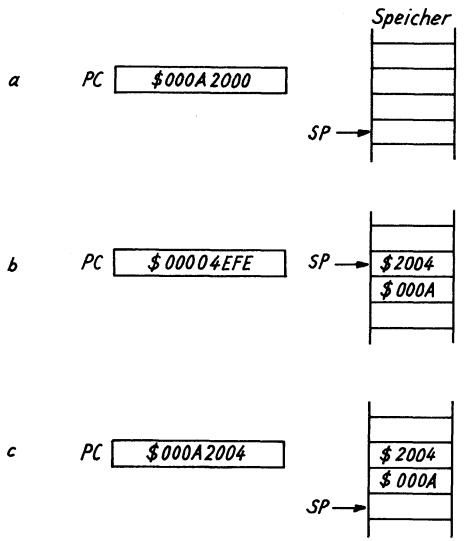


Bild 3.17
Subroutinenaufruf und -rückkehr

a) vor Ausführung von JSR$4EFE
b) nach Ausführung von JSR$4EFE
c) nach Ausführung von RTS

In früheren Diskussionen der Datentransferbefehle sahen wir, dass die Befehlsform

```
MOVEM ‹list›, –(SP)
```

zur Rettung ausgewählter Register im Stapel während der Subroutinenausführung verwendet werden kann, und zwar um diese unterbrechbar (reentrant) zu machen. In vielen Anwendungen müssen auch die Bedingungscode gesichert werden, so dass die Zusammenhänge des Programms während der Ausführung der Subroutine erhalten bleiben. Dies ist ebenfalls mit einem bereits besprochenen Befehl möglich.

```
MOVE SR,–(SP)
```

Selbstverständlich müssen vor der Rückkehr aus der Subroutine die gesicherten Werte wieder aus dem Stapel geholt werden. Dies kann mit der Sequenz

```
MOVEM  (SP)+,‹list›
MOVE   (SP)+,CCR
```

gemacht werden. Dennoch verfügt der MC 68000 über eine spezielle Version des RTS-Befehls, «Rückkehr und Rückspeicherung der Bedingungscode» (RTR) genannt, der das Bedingungscoderegister wie die Rückkehradresse aus dem Stapel holt. Beispiel 3.4 zeigt, wie die Bedingungscode und gewisse Arbeitsregister während der Subroutine erhalten werden und wie mit RTR die Rückkehr veranlasst wird.

Programmbeispiel 3.4

```
        JSR         SUBR              SUBROUTINE-
                                      AUFRUF
        MOVE        D0,D1             NAECHSTER,
                                      DIREKTER
                                      BEFEHL
        *
        *
        *
SUBR    MOVE        SR,-(SP)          SICHERE STATUS-
                                      REGISTER IM
                                      STAPEL.
        MOVEM.L     D3-D5/A1,-(SP)    SICHERE
                                      REGISTER IM
                                      STAPEL.
        *
        *                             WEITERE SUB-
        *                             ROUTINEN-
        *                             BEFEHLE
        *
        MOVEM.L     (SP)+,A1/D3-D5    REGISTER
                                      ZURUECK-
                                      SPEICHERN.
        RTR                           RUECKKEHR
                                      UND RUECK-
                                      SPEICHERN
                                      BEDINGUNGS-
                                      CODE.
```

3.4.8 LINK- und UNLK-Befehle

Die LINK- und UNLK-Befehle (Tabelle 3.16) werden zur Zuweisung und Freigabe von Datenbereichen im Systemstapel für verschachtelte Subroutinen, verbundene Listen und andere Prozeduren verwendet. Nach dem Aufruf (zum Beispiel einer verschachtelten Subroutine) setzt LINK einen Adressregisterzeiger zum Datenbereich und verschiebt den Stapelzeiger im Speicher nach unten, genau nach dem Datenbereich. Nach Ausführen der Subroutine kehrt UNLK diese Sequenz um und setzt dabei den Stapelzeiger und die Adressregister auf ihre Originalwerte, das heisst auf die Werte von LINK.

Mnemonik	Assemblersyntax	Operandengrösse	Erlaubte Adressierungsarten		Bedingungscodes
			Quelle	Ziel	XNZVC
LINK	LINK An,#d	unbestimmt	An		-----
UNLK	UNLK An	unbestimmt		An	-----

Tabelle 3.16 LINK- und UNLK-Befehle

Der LINK-Befehl hat zwei Operanden, ein Adressregister und einen 16 Bit langen, vorzeichenbehafteten Verschiebungswert. Während die verschachtelte Subroutine ausgeführt wird, enthält das Adressregister die Startadresse des Datenbereichs für diese Subroutine im Stapel.
Dieses Adressregister wird Rahmenzeiger (RZ oder FP, frame pointer) genannt. Der Verschiebungswert spezifiziert in Byte den Speicherbedarf im Stapel, der dem Datenbereich zugewiesen wird. Wenn LINK ausgeführt wird, bringt der MC 68000 den 32-Bit-Inhalt des FP in den Stapel, dekrementiert den Stapelzeiger (SZ oder SP) um vier, lädt diesen SP-Wert in den FP und addiert dann den Verschiebungswert zum SP. Erwähnenswert ist, dass der Verschiebungswert zwei Charakteristiken aufweist:

1.) Weil der Stapelzeigerwert immer gerade sein muss, muss der Verschiebungswert eine gerade Zahl sein, und
2.) weil der Verschiebungswert zum Stapelzeiger *addiert* wird, sollte er für die meisten Anwendungen *negativ* sein.

Nach der Ausführung von LINK enthält das Adressregister die Startadresse des Datenbereichs und der Stapelzeiger weist auf die dem Datenbereich folgende Speicherzeile. Ab diesem Punkt kann die Subroutine den Datenbereich sehr einfach benützen, durch indirekten Zugriff mit Adressregister und Verschiebungs- oder Indexmodus. Bild 3.18a und 3.18b zeigt den Systemstapel nach dem Subroutinenaufruf und nach LINK.

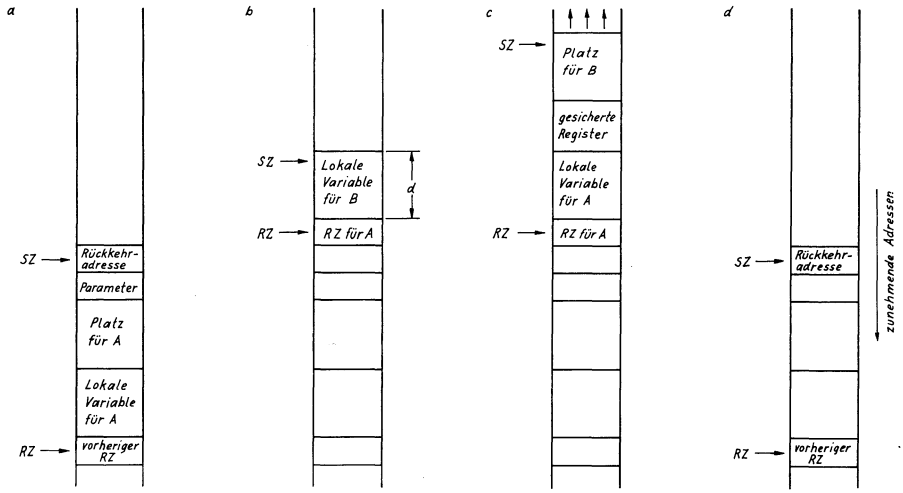


Bild 3.18 Zuweisung und Freigabe von Speicherplatz mit den Befehlen LINK und UNLK

a) nach Subroutinenaufruf, b) nach LINK, c) vor UNLK, d) nach UNLK
(SZ: Stapelzeiger, RZ: Rahmenzeiger)

Bild 3.18c zeigt die Stapelzeigeradressierung einer geraden, tieferen Speicherzeile. Diese Darstellung soll zeigen, dass der UNLK-Befehl eine normale Rückkehr einleitet (in Bild 3.18d gezeigt), ohne Rücksicht darauf, ob der Stapelzeiger inzwischen geändert haben könnte. Der UNLK-Befehl, der normalerweise unmittelbar vor der Rückkehr aus der Subroutine ausgeführt wird, lädt einfach den Stapelzeiger aus dem Rahmenzeigerregister und reinitialisiert dann den Rahmenzeiger, indem der Originalwert zuoberst aus dem Stapel geladen wird. Nach UNLK enthalten sowohl der Rahmenzeiger wie der Stapelzeiger die Werte, die sie vor LINK enthielten.

3.4.9 Systemsteuerungsbefehle

Tabelle 3.17 enthält diejenigen Befehle, die in den Herstellerunterlagen als Systemsteuerungsbefehle (system control instructions) bezeichnet werden. Es sind drei Typen zu unterscheiden: privilegierte Befehle, Trap-Erzeugungsbefehl und Statusregisterbefehle. Die Statusregisterbefehle wurden in diesem Kapitel bereits behandelt, und ihre Beschreibung soll nicht wiederholt werden.

3.4.9.1 Privilegierte Befehle

Wie bekannt ist, können privilegierte Befehle nur ausgeführt werden, wenn sich der MC 68000 im Überwachungsmodus befindet. Jeder Versuch, im Anwendermodus einen privilegier-

Mnemonik	Assemblersyntax	Operanden-grösse	Erlaubte Adressierungsarten		Bedingungs-codes
			Quelle	Ziel	XNZVC
Privilegierte Befehle					
RESET	RESET				-----
RTE	RTE		(SP)+ → SP; (SP)+ → PC		*****
STOP	STOP #d	16	#d → SR, then STOP		*****
ANDI	ANDI #d,SR (1)	16	#d	SR	-**00
EORI	EORI #d,SR (1)	16	#d	SR	-**00
ORI	ORI #d,SR (1)	16	#d	SR	-**00
MOVE	MOVE <ea>,SR (2)	16	Data	SR	*****
	MOVE USP,An (2)	32	USP	An	-----
	MOVE An,USP (2)	32	An	USP	-----
Trap-Erzeugungsbefehle					
TRAP	TRAP #<vector>		PC → -(SP); SR → -(SP); #<vector> → PC		-----
TRAPV	TRAPV		If V = 1, then TRAP		-----
CHK	CHK <ea>,Dn	16	If Dn < 0 or Dn > (ea), then TRAP	Daten	-*UUU
Status-Registerbefehle					
ANDI	ANDI.B #d,SR (1)	8	#d	CCR	-**00
EORI	EORI.B #d,SR (1)	8	#d	CCR	-**00
ORI	ORI.B #d,SR (1)	8	#d	CCR	-**00
MOVE	MOVE <ea>,CCR (2)	16	Data	CCR	*****
	MOVE SR,<ea> (2)	16	SR	Daten änderbar	-----

Bem.: (1) Beschrieben bei der Gruppe der logischen Befehle (Tabelle 3.10 und Text).
(2) Beschrieben bei der Gruppe der Datentransferbefehle (Tabelle 3.7 und Text).

Tabelle 3.17 Systemsteuerungsbefehle

ten Befehl auszuführen, wird einen Ausnahmezustand herbeiführen (in Kapitel 7 behandelt).
Der RESET-Befehl (Rücksetzen externer Bausteine) aktiviert den RESET-Anschluss des MC 68000 während 124 Taktzyklen. Dieser Anschluss ist normalerweise mit allen externen Bausteinen im System verbunden und veranlasst das Rücksetzen dieser Bausteine, ohne den Prozessor zu beeinflussen. Der RESET-Befehl kann zum Wiederanlauf nach schwerwiegenden Systemfehlern verwendet werden.
Wie im Kapitel 7 gezeigt werden wird, veranlassen Unterbrüche und andere Ausnahmezustände, dass das 16-Bit-Statusregister und der 32-Bit-Programmzähler in den Überwachungsstapel geschrieben werden, damit der Programmstatus bei Erscheinen des Ausnahmefalles gesichert wird.
Der RTE-Befehl (Rückkehr aus dem Ausnahmezustand, return from exception) bringt diese Werte aus dem Stapel zurück, nachdem die Ausnahmeroutine ausgeführt wurde. RTE entspricht also für Ausnahmezustände den Befehlen RTS und RTR für Subroutinen.

Der «Stopp Programmausführung»-Befehl (STOP) lädt einen Wert in das Statusregister und veranlasst den MC 68000, das Holen und Ausführen von Befehlen zu stoppen. Die Ausführung wird nicht wiederaufgenommen, bis der MC 68000 einen Unterbruch hinreichend hoher Priorität empfängt oder extern zurückgesetzt wird. STOP wird im praktischen Gebrauch oft benützt, um die Unterbruchsmaske zu ändern, und kann als erweiterter «warte auf Unterbruch»-(WAI-)Befehl des 8-Bit-Mikroprozessors MC 6800 betrachtet werden.

3.4.9.2 Trap-Erzeugungsbefehle

Traps («Fallen») veranlassen wie Unterbrüche, dass der Programmzähler mit einer bestimmten Adresse im Speicher geladen wird, je nach «Vektornummer», die dem Prozessor geliefert wird. Bei Unterbrüchen werden alle Vektornummern durch externe Bausteine geliefert, bei Traps werden sie *intern* erzeugt. Wie später dargestellt wird (Kapitel 7), werden Traps automatisch durch gewisse Fehlerbedingungen erzeugt; sie lassen sich aber auch durch Software mit irgendeinem der drei hier beschriebenen Befehle erzeugen.
Der TRAP-Befehl initialisiert eine unbedingte Trap-Operation und liefert eine Vektornummer (0 bis 15) im Operanden. TRAP kann also zur Erzeugung von irgendeinem von 16 Softwareunterbrüchen verwendet werden.
Der Befehl «Trap bei Überlauf» (TRAPV) prüft das Überlaufbit (V) im Bedingungscoderegister und führt bei gesetztem V zu einer spezifizierten Speicheradresse. Wenn V nicht gesetzt ist, wird der nächstfolgende Befehl ausgeführt. Auch der dritte Trap-Befehl, «Prüfe Register auf Grenzen» (CHK) operiert unbedingt. Dieser Befehl prüft den Inhalt eines Datenregisters und verzweigt zu einer spezifizierten Speicherzeile, wenn der Registerinhalt einen Wert aufweist, der kleiner als null oder grösser als ein adressierter «obere Grenze»-Operand ist. Diese Art der Prüfung hilft, Datenbereiche in den definierten Grenzen zu halten.

3.5 Zusammenfassung

In diesem Kapitel wurden die 14 Adressierungsarten und ihre Anwendung behandelt. Sie bieten alle Möglichkeiten früherer 8-Bit-Mikroprozessoren sowie eine ganze Anzahl wertvoller Ergänzungen. Die Fähigkeit, eine Adresse vor der eigentlichen Operation zu dekrementieren oder nachher zu inkrementieren, eröffnet dem Programmierer einen schnellen, wirkungsvollen Weg zur Behandlung von Zeichenfolgen und Tabellen. Weiter ermöglicht der Einbezug der Adressierungsarten mit Verschiebungswerten und Indizes einfachen Zugriff zu Datenbereichen.
Ebenfalls in diesem Kapitel wurden alle 56 mikrocodierten Befehle des MC 68000 behandelt. Wie bei den Adressierungsar-

ten dürften auch viele Befehle den Lesern mit Programmiererfahrung auf dem MC 6800 oder andern 8-Bit-Mikroprozessoren bekannt vorkommen, wobei aber auch hier verbesserte Versionen und eine einfachere Anwendung angeboten werden. Zum Beispiel wurden die Lade-, Speicher- und Registertransferbefehle in einem einzigen, MOVE genannten Befehl kombiniert. Andere häufig verwendete Operationen, die normalerweise mehrere Zeilen Code benötigen, sind zu Einzelbefehlen kombiniert worden. So finden wir im MC 68000 Befehle wie «prüfe, dekrementiere und verzweige» (DBcc), «inkrementiere mehrfach» (ADDQ) und «dekrementiere mehrfach» (SUBQ).
Im Hinblick auf die spezielle Unterstützung höherer Programmiersprachen stehen zum erstenmal Befehle wie «prüfe Register auf Grenzen» (CHK) und für die Zuweisung und Freigabe von Platz im Stapel für lokale Variable während Prozeduraufrufen (LINK und UNLK) zur Verfügung. Im weiteren erlaubt der enorme Adressbereich des MC 68000 (16 MByte) Multitasking und Multiprocessing, wobei ein «Speicherzuweisungsbefehl» (TAS) eingesetzt werden kann.
Mit dieser Übersicht über die Programmiermöglichkeiten des MC 68000 soll nun in den nächsten zwei Kapiteln deren Einsatz an praktischen Anwendungen im Zusammenhang mit mathematischen Operationen und Verarbeitung von Listen und Konversionstabellen dargestellt werden.

4. Mathematische Routinen

Leser, die ihre ersten Erfahrungen in der Mikrocomputerprogrammierung mit 4-Bit- oder 8-Bit-Mikroprozessoren gemacht haben, werden von den arithmetischen Möglichkeiten des MC 68 000 beeindruckt sein. Zum Beispiel bringt die Tatsache, dass der MC 68 000 über Multiplizier- und Dividierbefehle sowohl mit wie ohne Vorzeichen verfügt, einen Gewinn von Stunden (wenn nicht Tagen oder Wochen) bei der Entwicklung von Multiplikations- oder Divisionssubroutinen.
In diesem Kapitel werden wir auf der Basis der angebotenen Multiplikations- und Divisionsmöglichkeiten einige mathematische Aufgaben behandeln. Wir beginnen mit Multiplikationsoperationen (32 Bit × 32 Bit) mit und ohne Vorzeichen. Dann werden Überlaufsituationen bei Divisionen behandelt, und zum Schluss wird ein Programm entwickelt, mit dem die Quadratwurzel einer 32-Bit-Zahl ermittelt werden kann.

4.1 Multiplikation

Im Kapitel 3 lernten wir die Multiplikationsbefehle MULS «Multiplikation mit Vorzeichen» und MULU «Multiplikation ohne Vorzeichen» kennen und nahmen zur Kenntnis, dass sie nur mit 16-Bit-Werten (Wortlänge) operieren können. Wie werden nun Werte von 32 Bit oder länger multipliziert? Wie jedermann weiss, der ein Multiplikationsprogramm für einen 8-Bit-Mikroprozessor geschrieben hat, genügt die Existenz eines Multiplikationsbefehls irgendeiner Länge, um ihn dann für bestimmte Anforderungen zu erweitern.

4.1.1 32 Bit × 32-Bit-Multiplikation ohne Vorzeichen

Zahlen mit Mehrfachgenauigkeit ohne Vorzeichen können multipliziert werden unter Verwendung des MULU-Befehls mittels Erzeugen einer Serie von 32-Bit-*Zwischenprodukten,* die dann zum endgültigen Produkt *summiert* werden. Die gleiche Methode wird verwendet zum Multiplizieren von Dezimalzahlen von Hand, mit Papier und Bleistift. Wie der Leser sich vielleicht erinnern wird (im Zeitalter der Taschenrechner vielleicht

nicht selbstverständlich!), werden die Faktoren untereinander- oder nebeneinander geschrieben und die Multiplikation in einer Serie von einzelnen Multiplikationen für jede Stelle im Multiplikator durchgeführt. Jedes Zwischenprodukt wird direkt unter die entsprechende Stelle im Multiplikator geschrieben. Wenn alle Zwischenprodukte berechnet sind, werden sie addiert zum Resultat. Zum Beispiel wird die Multiplikation von 124 x 103 in der folgenden Art geschrieben:

```
   124    Multiplikand
 x 103    Multiplikator
 -----
   372    Zwischenprodukt #1
  000     Zwischenprodukt #2
 124      Zwischenprodukt #3
 -----
 12772    Resultat
```

Die Zwischenprodukte werden versetzt, um das *dezimale Gewicht* der einzelnen Multiplikatorstellen berechnen zu können. In diesem Beispiel ist die 3 die Einerstelle, die 0 die Zehnerstelle und die 1 die Hunderterstelle. Dadurch kann das Beispiel in der folgenden Art geschrieben werden:

$$103 \times 124 = (3 \times 124) + (0 \times 124) + (100 \times 124)$$

oder

$$103 \times 124 = (3 \times 10^0 \times 124) + (0 \times 10^1 \times 124) + (1 \times 10^2 \times 124)$$

In diesem Abschnitt werden wir eine Subroutine entwickeln, um zwei 32-Bit-Zahlen ohne Vorzeichen zu multiplizieren, was ein 64-Bit-Produkt ohne Vorzeichen ergibt. Ohne einen Multiplizierbefehl würde das bedeuten, dass 32 Multiplikationsoperationen, eine für jedes Bit im Multiplikator, durchgeführt werden müssten. Glücklicherweise verfügt der MC 68 000 über einen Befehl, der 16-Bit-Zahlen ohne Vorzeichen direkt multipliziert. Dieser Befehl MULU erlaubt uns, die 32-Bit-Faktoren als zweistellige Zahlen zu betrachten, jede Stelle mit 16 Bit. Dadurch sind nur *vier* Multiplikationen erforderlich, um das 64-Bit-Produkt zu erzeugen. Bild 4.1 zeigt die symbolische Darstellung der Faktoren und erläutert, wie die Zwischenprodukte angeordnet werden müssen, um das 64-Bit-Endprodukt zu berechnen. Die eingekreisten Zahlen in Bild 4.1 zeigen die vier 16-Bit-Additionen, die zur Berechnung des Produkts durchzuführen sind.
Unter Verwendung von Bild 4.1 ist es möglich, eine Subroutine zu entwickeln, die zwei 32-Bit-Zahlen ohne Vorzeichen multiplizieren kann. Programmbeispiel 4.1 zeigt eine solche Subroutine, MULU32 genannt, in der die Faktoren in den Datenregistern D2 und D1 eingeschrieben sind. Das 64-Bit-Produkt wird in die gleichen Register zurückgeschrieben, D1 (die 32 tieferen Bit) und D2 (die 32 höheren Bit).
Die durch die MULU32-Subroutine durchgeführten Operationen werden genau in der Reihenfolge von Bild 4.1 ausgeführt, wie aus den Befehlen und den entsprechenden Kommentaren

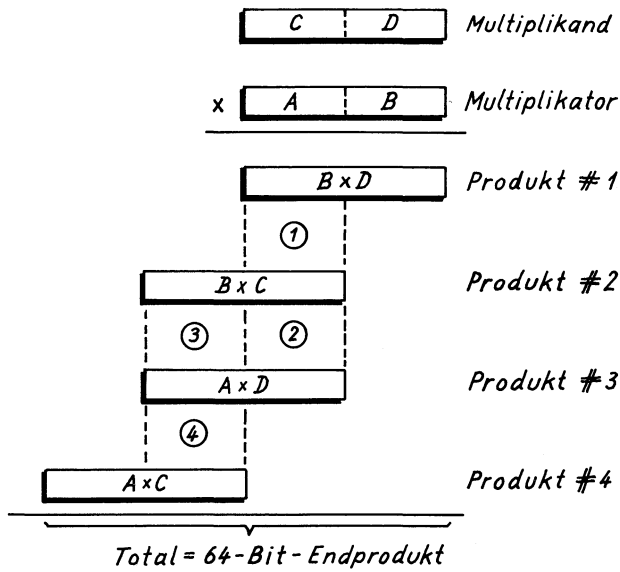


Bild 4.1
Erzeugung eines 64-Bit-Produktes mittels vier 16 Bit × 16-Bit-Multiplikationen.

ersichtlich ist. Die MULU32-Subroutine beginnt mit dem Retten des Inhalts der drei allgemein verwendbaren Register D3, D4 und D5 im Stapel und der Erstellung einer Kopie des Multiplikators in D3 und D4. Der nächste Befehl vertauscht die 16-Bit-Hälften von D4. Dieses Vertauschen ist eine notwendige Vorbereitung zur Erzeugung des zweiten und vierten Zwischenprodukts (siehe Bild 4.1), die das *höhere Wort* des ersten Multiplikanden beinhalten. Dieses Vertauschen ist notwendig, weil der MULU-Befehl nur die tiefern Worte von zwei Datenregistern miteinander multiplizieren kann. Dieser Swap-Befehl ist der erste einer ganzen Anzahl in der Subroutine. Ein Swap-D5-Befehl wird zwei Instruktionen später verwendet, um das dritte und vierte Teilprodukt zu erzeugen.

Jetzt kann, mit all den Multiplikationsoperanden am richtigen Ort, die eigentliche Multiplikation ausgeführt werden. Die Subroutine hat vier aufeinanderfolgende MULU-Befehle, welche die Zwischenprodukte eins, zwei, drei und vier in die Datenregister D1, D2, D3 und D4 bringen. Die verbleibende Aufgabe ist, die Summe dieser Zwischenprodukte zu bilden, unter Berücksichtigung ihrer Stellenwerte, um das 64-Bit-Produkt zu erhalten.

Die eingekreisten Zahlen in Bild 4.1 bestimmen die vier Paare von 16-Bit-Wörtern, die in der entsprechenden Reihenfolge addiert werden müssen. Im Programmbeispiel 4.1 folgt den vier aufeinanderfolgenden MULU-Befehlen ein SWAP-Befehl, der die Wortinhalte von D1 umtauscht (Zwischenprodukt #1). Dieser Tausch ist eine notwendige Vorbereitung für die erste Additionsoperation, weil der Additionsbefehl wie der MULU-Befehl

Programmbeispiel 4.1: Subroutine für die Multiplikation von zwei 32-Bit-Werten ohne Vorzeichen.

```
*Diese Subroutine multipliziert zwei 32-Bit-Werte ohne Vorzeichen und erzeugt ein 64-Bit-
*Produkt. Beginn mit dem Multiplikator in D2 und dem Multiplikanden in D1. Das Produkt
*erscheint in D1 (tiefere 32 Bit) und D2 (höhere 32 Bit).

          ORG       $1000
MULU32    MOVEM.L   D3-D5,-(SP)   Sichere Register
          MOVE.L    D1,D3         Schreibe Multiplikand in D3
          MOVE.L    D1,D4         und D4,
          SWAP      D4            in vertauschter Form.
          MOVE.L    D2,D5         Schreibe Multiplikator in D5,
          SWAP      D5            in vertauschter Form.
          MULU      D2,D1         Zwischenprodukt #1
          MULU      D4,D2                         #2
          MULU      D5,D3                         #3
          MULU      D5,D4                         #4
          SWAP      D1            SUM1:=PP #2 LOW +
          ADD       D2,D1         PP #1 HIGH
          CLR.L     D5
          ADDX.L    D5,D4         Übertrag in PP #4
          ADD       D3,D1         SUM2:=SUM1 + PP #3 LOW
          ADDX.L    D5,D4         Übertrag in PP #4
          SWAP      D1            Vertausche tieferes Produkt
          CLR       D2            Vorbereitung für SUM3
          SWAP      D2
          CLR       D3
          SWAP      D3
          ADD.L     D3,D2         SUM3:=PP #2 LOW + PP #3 HIGH
          ADD.L     D4,D2         SUM4:=SUM3 + PP #4
          MOVEM.L   (SP)+,D3-D5   Register zurückspeichern
          RTS
          END
```

nur die tieferen Werte von zwei Datenregistern berücksichtigt. Nach Durchführen der ersten Addition wird der Übertrag dieser Operation (in X) nach D4 gebracht (Zwischenprodukt #4) unter Benützung des Registers D5 (enthält Null) als Hilfsoperand für die «addiere-erweitert»-Operation. In der zweiten Additionsoperation wird das tiefere Wort von D3 (Zwischenprodukt #3) zum tieferen Wort von D1 addiert, welches das Resultat der ersten Additionsoperation enthält, und ein allfälliger Übertrag wird wiederum in D4 eingeschrieben. Zu diesem Zeitpunkt sind die tieferen 32 Bit des Endprodukts im Datenregister 1 jedoch nicht in der richtigen Ordnung. Ein SWAP-D1-Befehl bereinigt dieses Problem, und der 68 000 ist bereit zur Akkumulierung der 32 höheren Bit des Produkts. Das bedingt die Addition des höheren Wortinhaltes von Datenregister 2 und

3 (Teilprodukte #2 und #3) zum tieferen Wortinhalt des Datenregisters D4 (Teilprodukt #4). Die tieferen Wörter sowohl von D2 wie auch D3 enthalten unnötige Daten von den zwei ersten Additionen, so dass beide Wörter gelöscht werden und in die höhere Wortposition dieser Register getauscht werden. Zwei «addiere-lang»-Befehle setzen die tieferwertigen 32 Bit des Endprodukts in Datenregister D2. Nach der Rückspeicherung des Inhalts der Datenregister D3, D4 und D5 vom Stapel endet die Subroutine. Die MULU32-Subroutine benötigt zur Ausführung ein Maximum von 460 Zyklen oder 57,5 µs. Weil ein 32-Bit-Operand Zahlen ohne Vorzeichen bis $4{,}294 \times 10^9$ darstellen kann, verlangen viele Anwendungen keine Multiplikationssubroutinen mit grösseren Zahlen (diese würde wahrscheinlich Fliesskommaarithmetik verlangen). Es ist aber möglich, Multiplikationssubroutinen zu schreiben für 64-Bit- oder längere Zahlen mit dem in Beispiel 4.1 verwendeten Prinzip. Allerdings wird man bald über zuwenig Arbeitsregister verfügen und benötigt dann Speicherplatz für Zwischenspeicherung.

4.1.2 32 Bit × 32-Bit-Multiplikation mit Vorzeichen

Obschon das Multiplikationsbeispiel 4.1 als Subroutine zur Multiplikation von zwei nicht vorzeichenbehafteten Zahlen geschrieben wurde, wird es auch zwei Zahlen mit Vorzeichen korrekt multiplizieren, solange beide positiv sind. Das heisst, das Beispiel 4.1 ist eine 32 Bit × 32-Bit-Multipliziersubroutine für nicht negative Zahlen. Diese Subroutine kann indessen nicht zwei negative Zahlen multiplizieren, weil solche Zahlen normalerweise in *Zweierkomplementform* vorhanden sind. Wie aber können zwei Zahlen mit Vorzeichen multipliziert werden, wenn eine oder beide negativ sind? Eine Lösung wäre, den oder die negativen Operanden zu negieren, die Multiplikation durchzuführen, dann das Produkt zu berichtigen, sofern erforderlich. Wenn nur einer der beiden Operanden negativ ist, muss das Resultat in Zweierkomplement gesetzt werden. Wenn beide Operanden negativ sind, ist das (positive) Produkt korrekt. Dieses einfache Vorgehen ist im Programmbeispiel 4.2 angewendet, in dem die tieferen Bit des Datenregisters D6 zur Aufnahme eines Negativindikators dienen.

Dieser Indikator, mit Null initialisiert, wird auf alles Eins gesetzt, wenn nur einer der beiden Operanden negativ ist. Er bleibt aber Null, wenn beide Operanden entweder positiv oder negativ sind. Dann, nach Aufruf der MULU32-Subroutine zur Durchführung der 32 Bit × 32-Bit-Multiplikation, wird der Negativindikator verwendet, um festzustellen, ob das Produkt korrekt ist (Indikator Null) oder negiert werden muss (Indikator nicht Null). Die Subroutine MULS32 im Programmbeispiel 4.2 wird eine Ausführungszeit benötigen, die abhängig davon

Programmbeispiel 4.2: Eine 32 Bit × 32-Bit-Multiplikationssubroutine für vorzeichenbehaftete Zahlen.

```
*Diese Subroutine multipliziert zwei 32-Bit-Zahlen mit Vorzeichen und erzeugt ein 64-Bit-
*Produkt. Start mit dem Multiplikanden in D2 und dem Multiplikator in D1. Das Resultat wird
*in D1 (die tieferen 32 Bit) und in D2 (höhere 32 Bit plus Vorzeichen) eingeschrieben.
          ORG       $2000
MULS32    MOVE.B    D6,-(SP)      Sichere Register
          CLR.B     D6            Negativindikator = 0
          TST.L     D1            Multiplikand negativ?
          BPL.S     CHKD2         Nein. Teste Multiplikator.
          NEG.L     D1            Ja. Bilde Zweierkomplement,
          NOT.B     D6            Einerkomplement Indikator.
CHKD2     TST.L     D2            Multiplikator negativ?
          BPL.S     GOMUL         Nein. Multipliziere.
          NEG.L     D2            Ja. Bilde Zweierkomplement.
          NOZ.B     D6            Einerkomplement Indikator.
GOMUL     JSR       MULU32        Aufruf Multiplikationssubroutine.
          TST.B     D6            Vorzeichen korrekt?
          BEQ.S     DONE          Ja: Ausgang.
          NEG.L     D1            Nein: Bilde Zweierkomplement
          NEGX.L    D2
DONE      MOVE.B    (SP)+,D6      Rückspeicherung D6.
          RTS
          END
```

ist, ob die Operanden beide positiv oder beide negativ sind oder beide entgegengesetzte Vorzeichen haben. Die Ausführungszeit von MULS32 (eingeschlossen die aufgerufene Subroutine MULU32) ist wie folgt:

Operanden	Maximale Zeit (in Zyklen)	Maximale Zeit (Mikrosekunden)
Beide positiv	561	70,125
Vorzeichen verschieden	579	72,375
Beide negativ	577	72,125

Eine schnellere Lösung, die nicht die Änderung eines Operanden verlangt, kann unter Beachtung des nachfolgenden *Algorithmus* ausgeführt werden:

> Wenn einer oder beide Operanden negativ sind, führe die Multiplikation durch und modifiziere das Produkt in einer von zwei Arten:
> 1. Wenn nur ein Operand negativ ist, subtrahiere den anderen Operanden (das heisst den positiven Operanden) vom höherwertigen Teil des Produkts.
> 2. Wenn beide Operanden negativ sind, subtrahiere beide Operanden vom höherwertigen Teil des Produkts.

Sind Sie skeptisch? Zur Prüfung dieses Algorithmus wollen wir das Beispiel 103×124 noch einmal durchführen, aber mit einem negativen Multiplikator (−103). Die Papier- und Bleistiftmethode sieht wie folgt aus:

```
    01111100     Multiplikand  = +124
x   10011001     Multiplikator = -103
    01111100
   00000000
  00000000
 01111100
01111100
00000000
00000000
01111100
-----------------
0100101000011100     Produkt = +18972
```

Wenn wir das Resultat mit dem korrekten Wert (−12 772) vergleichen, sehen wir, dass unser Resultat Unsinn ist. Es ist nicht nur zu gross, es hat auch das falsche Vorzeichen. Nun wollen wir das Verfahren mit dem erwähnten Algorithmus betrachten. Der Algorithmus verlangt die Subtraktion des positiven Operanden (+124, ein einfaches Byte) vom höherwertigen Byte des Produkts. In Binärform ist es einfacher für uns, zu addieren statt zu subtrahieren, so wird das Zweierkomplement des positiven Operanden zum höherwertigen Byte des Produkts addiert:

```
 0100 1010 0001 1100     Originalprodukt = + 18972
+ 1000 0100              Zweierkomplement
                         Multiplikand = -124
-----------------
 1100 1110 0001 1100     Neues Produkt = -12772
```

Jetzt ist das Produkt korrekt. Schritt zwei des Algorithmus kann durch Anwendung der Papier- und Bleistiftmethode auf das Produkt $(-103) \times (-124)$ verifiziert werden.

Bild 4.2 zeigt die zusätzlichen Schritte, die nötig sind für die Multiplikation von Zahlen beliebiger Länge mit Vorzeichen. Wie man in Bild 4.2 sehen kann, erlaubt dieser Algorithmus das Verwenden unserer früher beschriebenen Multiplikationssubroutine für Zahlen ohne Vorzeichen (Programmbeispiel 4.1) zur Durchführung der Initialisierungsmultiplikation. Es besteht jedoch die zusätzliche Anforderung, dass der Originalmultiplikator und der Originalmultiplikand geschützt werden für die Produktkorrekturbefehle. Programmbeispiel 4.3 zeigt die neue effizientere 32 Bit × 32-Bit-Multiplikationssubroutine für vorzeichenbehaftete Zahlen.

Programmbeispiel 4.3:
Eine verbesserte 32 Bit × 32-Bit-Multiplikationssubroutine für Zahlen mit Vorzeichen.

```
*Diese Subroutine multipliziert zwei 32-Bit-Zahlen mit Vorzeichen und erzeugt ein 64-Bit-
*Produkt. Start mit dem Multiplikator in D2 und dem Multiplikanden in D1. Das Produkt wird
*zurückgeschrieben in D1 (32 tiefere Bit) und D2 (32 höhere Bit).
                ORG     $1000
MLTS32          MOVEM.L D3-D7,-(SP)     Sichere Register
                MOVE.L  D1,D6           Kopiere Multiplikanden in D6
                MOVE.L  D2,D7           und Multiplikator in D7
*
*Führe die 32 Bit × 32-Bit-Multiplikation ohne Vorzeichen aus
*
                MOVE.L  D1,D3           Kopiere Multiplikanden in D3
                MOVE.L  D1,D4           und in D4
                SWAP    D4              in vertauschter Form.
                MOVE.L  D,D5            Kopiere Multiplikator in D5
                SWAP    D5              in vertauschter Form.
                MULU    D2,D1           Teilprodukt #1
                MULU    D4,D2                   #2
                MULU    D5,D3                   #3
                MULU    D5,D4                   #4
                SWAP    D1              SUM1: = PP #2 tiefer +
                ADD     D2, D1          PP #1 höher
                CLR.L   D5
                ADDX.L  D5,D4           Bringe Übertrag in PP #4.
                ADD     D3,D1           SUM2:= SUM1 + PP #3 tiefer.
                ADDX.L  D5,D4           Bringe Übertrag in PP #4
                SWAP    D1              Korrekte Reihenfolge.
                CLR     D2              Vorbereitung für SUM3.
                SWAP    D2
                CLR     D3
                SWAP    D3
                ADD.L   D3,D2           SUM3: = PP #2 TIEF + PP #3 HOCH
                ADD.L   D4,D2           SUM4: = SUM3 + PP #4
*
*Befehle zur Modifikation des Produkts, sofern notwendig
*
                TST.L   D7              Multiplikator negativ?
                BPL.S   CHKD6           Nein. Prüfe Multiplikanden.
                SUB.L   D6,D2           Ja. Subtrahiere Multiplikanden.
CHKD6           TST.L   D6              Multiplikand negativ?
                BPL.S   DONE            Nein. Wir sind fertig.
                SUB.L   D7,D2           Ja. Subtrahiere Multiplikator.
DONE            MOVEM.L (SP)+,D3-D7     Register zurückspeichern
                RTS
                END
```

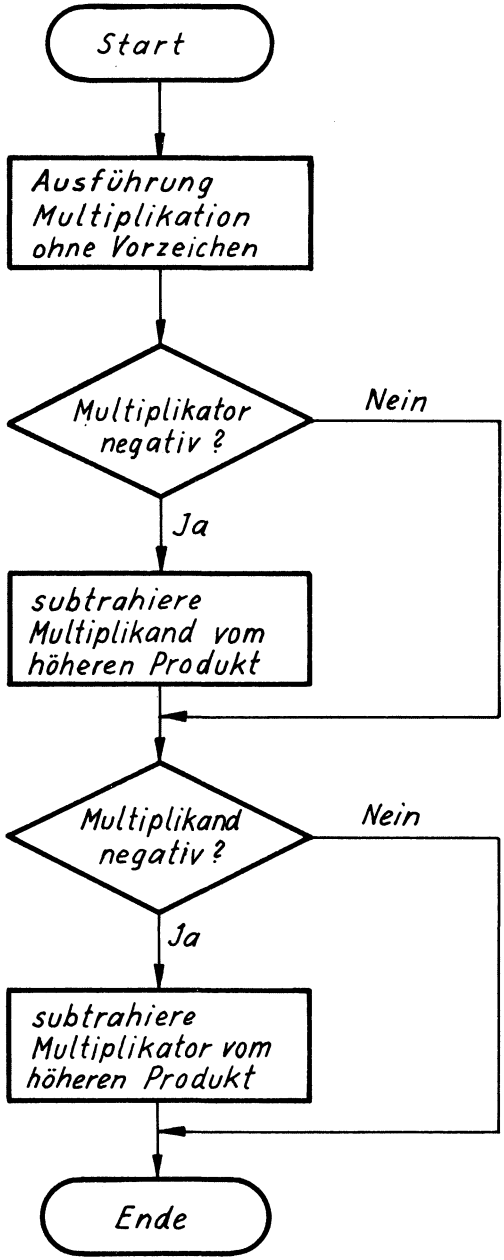


Bild 4.2
Ein Multiplikationsalgorithmus für vorzeichenbehaftete Zahlen.

Diese Subroutine MLTS32 ist nichts anderes als die MULU32-Subroutine von Beispiel 4.1 mit einigen zusätzlichen Befehlen am Anfang zum Schützen des Multiplikators und des Multiplikanden (in D7 und D6) sowie einigen zusätzlichen Befehlen am

Schluss zur Prüfung der Operandenvorzeichen und Korrektur des Produkts, sofern nötig.

Die Ausführungszeiten der MLTS32-Subroutine sind wie folgt:

Operanden	Maximale Zeit (in Zyklen)	Maximale Zeit (in Mikrosekunden)
Beide positiv	532	66,5
Vorzeichen verschieden	536	67,0
Beide negativ	540	67,5

4.2 Division

4.2.1 Division ohne Überlauf

Es gibt viele Anwendungen für die Divisionen, wobei eine der häufigsten die der Mittelwertbildung einer Anzahl Zahlen ist, zum Beispiel die Resultate einer Serie von Laborversuchen. Programmbeispiel 4.4 zeigt eine typische Routine für eine solche Aufgabe. Dieses Programm, genannt AVERAGE, mittelt eine spezifizierte Anzahl von Werten ohne Vorzeichen, auf die A0 zeigt, wobei die Anzahl der Werte im tieferen Wort von D0 enthalten ist. Der Mittelwert wird zurückgeschrieben als ganze Zahl in das tiefere Wort von D1 und ein Rest in das höhere Wort D1. Das AVERAGE-Programm verwendet zwei

Programmbeispiel 4.4: Routine für Mittelwertbildung.

```
* Diese Routine bildet Mittelwerte einer spezifizierten Anzahl von Werten ohne Vorzeichen, wel-
* che im Speicher gespeichert sind. Nach der Rückkehr wird der ganzzahlige Teil des Mittel-
* wertes in das tiefere Wort von D1 geschrieben und der Rest im höheren Wort von D1 gespei-
* chert. Die Adresse des ersten Wertes ist in A0 enthalten und die Anzahl Werte ist im tieferen
* Wort von D0 enthalten.

          ORG       $1000
AVERAGE   MOVE.L    D0/D2/D3/A0,-(SP)   Sichere Scratch-Register.
          MOVE      D0,D2               Nimm Wert in D2 und mache
          SUBQ      #1,D0               D0: = Anzahl -1.
          CLR.L     D1                  Lösche Dividenden-
          CLR.L     D3                  register und Wortregister.
LOOP      MOVE      (A0)+,D3            Nimm nächstes Wort
          ADD.L     D3,D1               und addiere es zum Total.
          DBF       D0,LOOP             Alle Werte totalisiert?
          DIVU      D2,D1               Ja. Berechne Mittelwert.
          MOVE.L    (SP)+,D0/D2/D3/A0   Register zurückspeichern.
          RTS
          END
```

Scratch-Register D2 (zur Aufnahme der Anzahl Werte) und D3 (zur Aufnahme der Werte, die aus dem Speicher gelesen werden), beeinflusst aber keine anderen Register als D1.
Es ist klar, dass die Dividieroperation in Beispiel 4.4 abbricht, wenn D0 beim Eintritt 0 enthält. Aber kann sie auch abgebrochen werden durch eine Überlaufbedingung? Nein, Überlauf kann es hier nicht geben, weil das Verhältnis des Dividenden (Wertetotal) zum Divisor (Werteanzahl) nie den Wert von 65 536 überschreiten kann. Hingegen könnte es Überlauf geben, wenn Langwortwerte verwendet werden zur Mittelwertbildung. Für diesen Fall ist es günstig, eine Prozedur zu kennen, mit der ein gültiger Quotient erhalten wird, unabhängig davon, ob ein Überlauf entsteht oder nicht.

4.2.2 Division mit Überlauf

Wie wir von Kapitel 3 wissen, setzt der MC 68 000 das Überlaufbit (V) und beendet die Operation, wenn ein Überlauf während der Ausführung der Division mit Vorzeichen (DIVS) oder ohne Vorzeichen (DIVU) entsteht, ohne Beeinflussung von Divisor oder Dividend. Überlauf entsteht dann, wenn der Dividend so viel grösser ist als der Divisor, dass der Quotient nicht in einem 16-Bit-Wort untergebracht werden kann.
In einigen Anwendungen soll Überlauf zu Fehlerbedingungen führen. In anderen Anwendungen kann ein Überlauf akzeptiert werden, bedeutet aber, dass ein Quotient mit mehr als 16 Bit resultiert. Weil die Division abgebrochen wird, wenn der MC 68 000 eine Überlaufbedingung feststellt, muss eine neue Lösung gesucht werden, falls ein solcher Quotient entsteht. Der vielleicht einfachste Weg zur Behandlung dieses Quotienten ist das Teilen des 32-Bit-Dividenden in zwei 16-Bit-Zahlen, um dann zwei 16 Bit : 16-Bit-Divisionsoperationen durchzuführen (die keinen Überlauf produzieren). Wenn der Divisor eine 16-Bit-Zahl ist (X) und der Dividend eine 32-Bit-Zahl (Y_1, Y_0), kann die Divisionsoperation betrachtet werden als

$$X \overline{\big|\ Y_1, Y_0}$$

oder, sauberer dargestellt, als

$$X \overline{\big|\ Y_1 \cdot 2^{16} + Y_0}$$

Die Division erzeugt zwei 16-Bit-Quotientenstellen (Q_1 und Q_0) und zwei 16-Bit-Reststellen (R_1 und R_0) wie folgt:

$$X \overline{\big|\ \overset{Q_1 \cdot 2^{16}}{Y_1 \cdot 2^{16}}} \text{ und } R_1 \cdot 2^{16}$$

$$X \overline{\big|\ \overset{Q_0}{(R_1 \cdot 2^{16}) + Y_0}} \text{ und } R_0$$

Wie man sehen kann, ist das Resultat dieser zwei Operationen ein 32-Bit-Quotient Q_1, Q_0 und ein 32-Bit-Rest R_0 (der Zwischenrest R_1, wenn überhaupt erzeugt, wird immer null während der zweiten Divisionsoperation). Wenn kein Überlauf entsteht, wird Q_1 Null sein und das Resultat wird zurückgeschrieben als $Q_0 = 0$ und $R_0 = 0$.
Ausgehend von den obigen Betrachtungen ist es möglich, eine Divisionssubroutine zu entwickeln, die *immer* einen gültigen Quotienten und einen gültigen Rest ergibt, unabhängig davon, ob Überlauf entsteht oder nicht. Das Programmbeispiel 4.5 zeigt die Subroutine DIVUO, welche diese Funktion ausführt.

Programmbeispiel 4.5: Eine Divisionssubroutine mit Behandlung von Überlauf.

```
* Diese Divisionssubroutine bestimmt den korrekten Quotienten und Rest ohne Rücksicht auf
* einen Überlauf. Start mit dem 16-Bit-Divisor in D0 und mit dem 32-Bit-Dividenden in D1. Der
* 32-Bit-Quotient wird in D1 und der 32-Bit-Rest in D0 gespeichert.

          ORG       $2000
DIVUO     MOVEM     D2/D3,-(SP)     Sichere Register
          CLR       D3
          DIVU      D0,D1           Ist Überlauf entstanden?
          BVC.S     FORMAT          Nein. Bilde Resultate.
          MOVE      D1,D2           Ja. Kopiere Y0 in D2.
          CLR       D1              D1 wechselt von Y1,Y0 zu Y1,0.
          SWAP      D1              D1 enthält 0,Y1.
          DIVU      D0,D1           Dividiere Werte R1,Q1.
          MOVE      D1,D3           D3 enthält Q1.
          MOVE      D2,D1           D1 wechselt von R1,Q1 zu R1,Y0.
          DIVU      D0,D1           Dividiere Werte R0,Q0.

*
* Bilde Quotient (D1) und Rest (D0)
*
FORMAT    MOVE.L    D1,D0           D0 enthält R,Q oder R0,Q0.
          SWAP      D1              D1 enthält Q,R oder Q0,R0.
          MOVE      D3,D1           D1 enthält Q,0 oder Q0,Q1.
          SWAP      D1              D1 enthält 0,Q oder Q1,Q0.
          CLR       D0              D0 enthält R,0 oder R0,0.
          SWAP      D0              D0 enthält 0,R oder 0,R0.
          MOVEM     (SP)+,D2/D3     Register zurückspeichern.
          RTS
          END
```

Sie dividiert einen 32-Bit-Dividenden in D1 durch einen 16-Bit-Divisor in D0 und prüft dann auf Überlauf. Falls ein Überlauf entsteht, verwendet die Subroutine die Datenregister D2 und D3, um die Korrektur durchzuführen. Nach diesen Divisionen (sofern sie notwendig sind) führt der MC 68 000 die Befehle bei FORMAT aus, wo der 32-Bit-Quotient in D1 und der 16-Bit-Rest in das tiefere Wort von D0 geladen wird. Wenn ein Überlauf entsteht, wird D1 Q_1, Q_0 enthalten und D0 R_0 wie in Bild 4.3 a dargestellt. Wenn kein Überlauf entsteht, enthält das tiefere Wort von D1 Q und von D0 R und das höhere Wort beider Register enthält Null wie in Bild 4.3 b dargestellt.

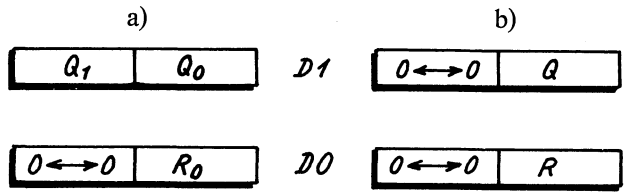


Bild 4.3
Divisionsresultate
a) mit Überlauf
b) ohne Überlauf

4.3 Quadratwurzel

Im letzten Teil dieses Kapitels wird ein Programm entwickelt, mit dem die Quadratwurzel einer 32 Bit langen, ganzen Zahl berechnet werden kann.
Die Berechnung wird mit Hilfe der klassischen Methode der sukzessiven Approximation durchgeführt. Zur Erläuterung dieser Methode nehmen wir an, dass die Zahl, deren Wurzel zu bestimmen ist, den Wert N haben soll. Die erste Approximation für die Quadratwurzel ist aus dem Wert (N/200) + 2 abgeleitet. N wird durch diesen Wert dividiert. Das Resultat wird zur ersten Approximation addiert und die Summe durch zwei dividiert. Dieses Resultat ist unsere nächste Approximation.
Zum Beispiel für die Quadratwurzel von 10 000:

N = 10 000; erste Approximation ist (10 000/200) + 2 oder 52
10 000/ 52 = 192, (192 + 52)/2 = 122
10 000/122 = 81, (122 + 81)/2 = 101
10 000/101 = 99, (101 + 99)/2 = 100
10 000/100 = 100

Wir sehen, dass die Quadratwurzel von 10 000 gleich 100 ist. Wir wissen natürlich, dass 100 die Quadratwurzel von 10 000 ist, weil ja 100 mit sich selbst multipliziert den Originalwert ergibt. Dieser spezielle Wert 10 000 hat eine ganzzahlige Quadratwurzel. Aber wir können nicht annehmen, dass die Lösung im allgemeinen eine ganzzahlige Quadratwurzel ist. Die Quadratwurzel für 9999 zum Beispiel ist keine ganze Zahl. Das bedeutet, dass bei der Berechnung der Quadratwurzel von 9999 der MC 68 000 die Berechnung laufend fortsetzt. Der Prozessor

Programmbeispiel 4.6: Subroutine für die Berechnung der Quadratwurzel aus einer 32-Bit-Zahl mittels sukzessiver Approximation.

```
*Diese Subroutine berechnet die Quadratwurzel einer ganzzahligen 32-Bit-Zahl in D0 und
*schreibt die Quadratwurzel als ganzzahlige 16-Bit-Zahl in das tiefere Wort von D1. Die ur-
*sprüngliche Zahl in D0 wird nicht beeinflusst.
        ORG     $2000
SQRT32  MOVEM.L D2/D3,-(SP)     Sichere Register.
        MOVE.L  D0,D2           Kopiere Datenwert in D2.
        DIVU    #200,D2         Dividiere durch 200,
        ADDQ    #2,D2           dann addiere 2.
NXTAPP  MOVE.L  D0,D1           Lade Datenwert in D1,
        DIVU    D2,D1           dividiere durch letzte Approximation.
        MOVE    D1,D3           Bringe neue Approximation in D3.
        SUB     D2,D3           Letzte zwei Approximationen identisch?
        BEQ.S   DONE            Ja. Ausgang.
        CMPI    #-1,D3          Nein. Differenz = 1?
        BEQ.S   DONE            Ja. Ausgang
        ADD     D1,D2           Addiere letzte zwei Approximationen
        LSR     #1,D2           und dividiere Summe durch 2
        BRA.S   NXTAPP
DONE    MOVEM.L (SP)+,D2/D3     Register zurückspeichern
        RTS
        END
```

wird fortfahren, die Approximationsbefehle zu durchlaufen, weil das Quadrat der *ganzzahligen* Approximation nie gleich 9999 sein wird. Daher muss ein Weg gefunden werden, um den Prozessor zu stoppen, wenn er den bestmöglichen Wert für die Quadratwurzel gefunden hat. Es gibt verschiedene Methoden zur Beendigung der Approximationsprozeduren. Die gewählte Methode ist abhängig von der gewünschten Genauigkeit und der zur Verfügung stehenden Ausführungszeit. Eine Lösung besteht darin, die Schlaufe zehnmal zu durchlaufen und anzunehmen, dass die Antwort genau genug ist. Diese Methode genügt für viele Anwendungen, ist ihrer Natur nach jedoch sehr willkürlich. Eine andere, genauere Lösung ist diejenige, den MC 68 000 die Schlaufe solange durchlaufen zu lassen, bis zwei aufeinanderfolgende Approximationen identisch sind oder sich nur durch den Wert Eins unterscheiden. Diese Methode wird in unserem Beispiel verwendet. Programmbeispiel 4.6 zeigt eine Subroutine (SQRT32), welche die ganzzahlige Quadratwurzel einer 32-Bit-Zahl durch sukzessive Approximation berechnet. Bei dieser Subroutine enthält das Datenregister D0 die 32-Bit-Zahl; die 16-Bit-Quadratwurzel wird in Datenregister D1 geschrieben. Die Subroutine beginnt mit der Approximation unter Verwendung der Beziehung (N/200) + 2. Der Rest der

Subroutine ist eine Schlaufe, beginnend mit NXTAPP, in der der MC 68 000 eine neue Approximation berechnet durch Division der ganzzahligen 32-Bit-Zahl durch die vorhergehende Approximation und der Bildung des Mittelwertes beider Approximationen. Vor der Mittelwertbildung prüft der MC 68 000 die Endbedingung dadurch, ob die neue Approximation gleich, um Eins grösser oder um Eins kleiner als die vorhergehende Approximation ist. Wenn eine dieser drei Bedingungen erfüllt ist, verlässt der MC 68 000 die Subroutine, wobei die 16-Bit-Quadratwurzel sich im Datenregister D1 befindet.

5. Listen und Konversionstabellen

5.1 Organisation von Daten

Es gibt verschiedene Methoden, wie Speicherinformationen für die Bearbeitung organisiert werden können. Diese Organisationstechniken sind für jede Anwendung verschieden. Sie werden unter die Begriffe *Listen, Arrays, Strings, Konversionstabellen* und *Vektoren* eingeordnet. Wir konzentrieren uns auf zwei Organisationstypen, die Listen und Konversionstabellen.
Listen sind wahrscheinlich das meistverwendete Datenspeicherformat. Sie bestehen aus Dateneinheiten (ein oder mehrere Byte), sprich Elemente, die hintereinander abgespeichert sind. Die Sequenzen der Elemente können direkt aufeinanderfolgen, indem jedes Element ein oder mehrere benachbarte Speicherplätze besetzt. Sie können auch verkettet sein, indem jedem Datenelement ein Zeiger folgt, der auf das nächste Element zeigt.
Im weiteren können die Datenelemente zufallsweise, in aufsteigender oder in absteigender Ordnung, gespeichert sein.
Die Konversionstabellen sind Datenstrukturen, die eine spezifische Eigenschaft haben. Damit will der Anwender Information (weniger Daten oder Adressen) erhalten, die eine wohldefinierte Beziehung zu einem bekannten Wort hat. Das Telefonbuch ist dafür ein gutes Beispiel. Ist der Name bekannt, so kann die entsprechende Telefonnummer herausgelesen werden.

5.2 Ungeordnete Listen

In unserer geordneten Gesellschaft, wo die Telefonbücher alphabetisch geordnet, die Hausnummern systematisch zu- oder abnehmen, kommt es uns merkwürdig an, von etwas Ungeordnetem zu sprechen.
Ungeordnete Listen sind auch das «Gift» für den Programmierer, weil sie sehr schwierig anzuwenden sind. Um einen bestimmten Wert in einer solchen Liste zu finden, muss jedesmal vom Anfang an mit der Suche danach begonnen werden. Es muss jedes Element gelesen werden, bis das richtige gefunden oder das Listenende erreicht worden ist. Ob man will oder nicht, kommen ungeordnete Listen in vielen Anwendungen des

Lebens vor. Sie bieten eine grundsätzliche Möglichkeit, zufällige, chronologisch abhängige oder dynamisch sich verändernde Daten abzuspeichern.

5.2.1 Zufügen von Daten zu einer ungeordneten Liste

Die Subroutine ADD2UL (Programmbeispiel 5.1) zeigt, wie der Anwender eine ungeordnete Liste kreieren oder ein neues Element dazugeben könnte. In diesem Beispiel beinhaltet diese Liste wortlange Werte (mit oder ohne Vorzeichen).

Programmbeispiel 5.1:
Zufügen eines Elementes zu einer ungeordneten Liste

```
* Diese Subroutine fügt das untere Wort des Datenregisters D0 einer ungeordneten
* Liste bei, wenn es nicht schon in der Liste ist.
* Die Startadresse der Liste steht im Adressregister A0.
* Die Länge der Liste, in Worten, ist im ersten Wort der Liste untergebracht.

        ORG     $2000
ADD2UL  MOVEM.L D1/A1,-(SP)   Rette Arbeitsregister.
        MOVEA.L A0,A1         Kopiere Startadresse in A1
        MOVE    (A1)+,D1      und den Wortzähler
        SUBQ    #1,D1         minus 1 in D1.
NXTEL   CMP     (A1)+,D0      Element schon vorhanden?
        BEQ.S   ITSIN         Ja, es ist in der Liste.
        DBI     D1,NXTEL      Nein! Schaue weiter.
        MOVE    D0,(A1)       Füge Element am Ende an,
        ADDQ    #1,(A0)       inkrementiere d. Wortzähler.
ITSIN   MOVEM.L (SP)+,D1/A1   Hole Arbeitsregister zurück.
        RTS
        END
```

Diese Subroutine sucht einfach die Liste Element um Element ab, um das Vorkommen des Wertes, der beigefügt werden soll, zu überprüfen. Wenn der Wert schon in der Liste ist, kehrt der 68000 von der Subroutine zurück, weil der Anwender keinen Wert duplizieren will. Ist der Wert noch nicht vorhanden, so wird er am Ende der Liste angehängt. Die Listenstartadresse ist im Adressregister A0. Das erste Listenelement (ein Wort) zeigt die Listenlänge in Worten an. So kann diese Liste maximal 64 KWorte lang sein.
Es gibt nichts Besonderes in dieser Subroutine. Sie kopiert die Listenstartadresse von A0 in A1. Danach liest sie den Wortzähler aus dem ersten Wort der Liste und deponiert diesen in D1. Dieser Zähler wird sogleich dekrementiert, weil die Suche abgebrochen wird, falls der Zähler den Wert –1 hat. Die Suche

beginnt bei NXTEL und vergleicht die Listenelemente mit D0. Wenn der Wert schon in der Liste ist, springt der 68000 auf ITSIN und kehrt zum Hauptprogramm zurück. Ist der Wert nicht vorhanden, so wird er am Listenende angehängt und der Elementzähler wird mit einer ADDQ-Instruktion um 1 grösser.
Wie lange wird dieser Subroutinen-Durchlauf dauern? Offensichtlich hängt das von der Zahl der Listenelemente ab und ob das Element schon vorhanden ist oder nicht.
Für alle, auch für die kleinste Liste, hängt die Ausführungszeit dieser Subroutine von der Anzahl Durchläufe der 3-Befehls-Schleife von NXTEL ab. Untersuchen wir einmal die Zeit für die Fälle, wenn das Element in der Liste ist und wenn es nicht dort ist. Die Liste hat N Elemente.
Wenn der gesuchte Wert *nicht in der Liste* ist, wird die NXTEL-Schleife N-mal durchlaufen. Für die N-1 ersten Ausführungen wird die Schleife 26 Zyklen brauchen; für die letzte 30 Zyklen. Die übrigbleibenden Instruktionen der Subroutine werden nur einmal ausgeführt und brauchen 110 Zyklen.

$$\begin{aligned}\text{Totale Ausführungszeit} &= 110 + 26\,(N-1)+30\\ &= 26\,N + 114 \text{ Zyklen}\end{aligned}$$

So werden also, um ein Element einer Liste mit 100 Elementen beizufügen, 2714 Zyklen oder 339,25 µs gebraucht.
Wenn der gesuchte Wert bereits *in der Liste* ist, wird der 68000 im Mittel N/2 Vergleiche machen müssen, um den betreffenden Wert zu finden. Er braucht so viel, da der gesuchte Wert zu je 50% in der ersten oder zweiten Listenhälfte sein wird. Für alle N/2 ohne den letzten Vergleich werden 26 Zyklen Ausführungszeit gebraucht. Für den letzten, bei welchem der gesuchte Wert in der Liste gefunden wird, werden 18 Zyklen gebraucht. Die restlichen Instruktionen werden noch 88 Zyklen ausmachen.

$$\begin{aligned}\text{Totale Ausführungszeit} &= 88 + 26(N/2-1)+18\\ &= 13\,N + 80 \text{ Zyklen}\end{aligned}$$

Im Fall der ungeordneten Liste mit 100 Elementen gibt das 1380 Zyklen oder 172,5 µs.

5.2.2 Löschen eines Elementes aus einer ungeordneten Liste

Um ein Element aus einer ungeordneten Liste zu entfernen, muss es zuerst gefunden werden. Danach werden alle folgenden Listenelemente um einen Platz nachrutschen. Sie überschreiben das gelöschte Element. Mit diesem Elementlöschen ist eines weniger in der Liste. Somit muss auch der Elementzähler um 1 dekrementiert werden.
Die DELEUL-Subroutine gibt ein Beispiel (5.2), wie eine solche Operation durchgeführt werden kann. Dabei wird das untere Wort vom Datenregister D0 den zu löschenden Wert enthalten,

Programmbeispiel 5.2:
Löschen eines Elementes aus einer ungeordneten Liste

```
* Diese Subroutine löscht den Wert im unteren Datenwort des Registers D0 einer
* ungeordneten Liste, falls dieser Wert in der Liste vorkommt.
* Die Listenstartadresse steht im Adressregister A0.
* Die Listenlänge, in Worten, ist im ersten Wort der Liste untergebracht.
*
          ORG       $1000
DELEUL    MOVEM.L   D1/A1,-(SP)    Retten des Arbeitsregisters.
          MOVEA.L   A0,A1          Kopiere die Startadresse in A1
          MOVE      (A1)+,D1       und den Wortzähler
          SUBQ      #1,D1          minus 1 in D1.
NEXTEL    CMP       (A1)+,D0       Kann gelöscht werden?
          NEQ.S     DELETE         Ja, lösche dieses Element.
          DBF       D1,NEXTEL      Nein, suche bis Listenende.
          BRA.S     ALLDUN         Element nicht vorhanden.
*
* Lösche ein Element, indem alle folgenden Elemente um ein Wort nachgeschoben
* werden
*
DELETE    MOVE      (A1)+,-4(A1)   Schiebe ein Wort nach.
          DBF       D1,DELETE      Alle Elemente nachgesch.?
          SUBQ      #1,(A0)        Ja, dekrementiere Elementz.
ALLDUN    MOVEM.L   (SP)+,D1/A1    Hole Arbeitsregister zurück.
          RTS
          END
```

und wie im Programmbeispiel 5.1 wird die Listenstartadresse im Adressregister A0 gespeichert sein.
Der erste Subroutineteil (DELEUL bis NEXTEL) löscht die Listenstartadresse in A1, wo der Elementzähler gespeichert ist. Danach wird der Elementzähler minus eins im D1-Register abgespeichert. Diese Instruktionen sind bis hierher die gleichen wie in 5.1. Die NEXTEL-Schleife vergleicht jedes Listenelement mit dem Wert im Register D0. Wenn das entsprechende Element gefunden wird, springt der 68000 auf die DELETE-Schleife, die jedes folgende Element um ein Wort aufschliessen lässt. Der Elementzähler wird dann um 1 dekrementiert.

5.2.3 Finden der Minimal- und Maximalwerte in einer ungeordneten Liste

Die Aufgabe, einen Minimal- oder Maximalwert in einer Liste zu finden, wird in mancher Anwendung gefordert, speziell dann, wenn Textdaten oder Statistikinformation verarbeitet werden sollen. Eine Methode geht davon aus, dass bei jedem neuen

Suchen das erste Datenelement als der entsprechende Wert betrachtet wird. Danach wird jedes weitere Element mit diesem Minimal- bzw. Maximalwert verglichen. Findet nun das Programm einen Wert, der kleiner bzw. grösser als das momentan gültige Minimum bzw. Maximum ist, dann wird dieser Wert das neue Minimum bzw. Maximum sein.
Die Subroutine MINMAX im Beispiel 5.3 wendet diese Methode auf eine ungeordnete Liste von Elementen ohne Vorzeichen an. Mit dem Aufrufen der Subroutine muss die Listenstartadresse in A0 sein. Nach der Rückkehr von der Subroutine zum Hauptprogramm sind die beiden Werte (Minimum, Maximum) in den zwei symbolischen Speicherplätzen MINVAL und MAXVAL verfügbar.

Programmbeispiel 5.3:
Finden des Minimal- und Maximalwertes in einer ungeordneten Liste

```
* Diese Subroutine findet den Minimal- und Maximalwert in einer ungeordneten Liste.
* Der Minimalwert wird im Speicherplatz MINVAL, der Maximalwert in MAXVAL
* zurückgegeben.
* Die Listenadresse steht im Adressregister A0.
* Die Listenlänge, in Worten, ist im ersten Listenelement abgespeichert.
*
            ORG       $3000
MINVAL      DS.W      1           Minimalwertspeicherplatz,
MAXVAL      DS.W      1           Maximalwertspeicherplatz.
MINMAX      MOVEM.L   A0/D0/D1,-(SP)  Rette Arbeitsregister.
            MOVE      (A0)+,D1    Schiebe Elementzähler in D1
            SUBQ      #1,D1       und dekrementiere ihn.
            MOVE      (A0),MINVAL 1. Element in MIN.
            MOVE      (A0)+,MAXVAL 1. Element in MAX.
CHKMIN      MOVE      (A0)+,D0    Lade nächstes Elem. in D0.
            CMP       MINVAL,D0   Ist das Element ein neues
                                  Minimum?
            BEQ.S     CONT
            BCC.S     CHKMAX
            MOVE      D0,MINVAL   Ja, führe MINVAL nach.
            BRA.S     CONT
CHKMAX      CMP       MAXVAL,D0   Ist das Element ein
            BLS.S     CONT        neues Maximum?
            MOVE      D0,MAXVAL   Ja, führe MAXVAL nach.
CONT        DBF       D1,CHKMIN   Listenende?
            MOVEM.L   (SP)+,A0/D0/D1  Hole Arbeitsregister zurück.
            RTS
            END
```

Im Beispiel 5.3 laden die Instruktionen zwischen MINMAX und CHKMIN den Elementzähler minus eins in das Datenregister D1 und speichern den ersten Datenelementswert in MINVAL und MAXVAL.
Bei CHKMIN wird das nächste Element in D0 geladen und danach mit MINVAL verglichen. Von diesem Punkt aus können drei Wege beschritten werden:

1. Wenn der Wert in D0 *gleich* wie MINVAL ist (Null-Flag gesetzt), springt der 68000 nach CONT, um zu prüfen, ob alle Elemente bearbeitet worden sind.
2. Wenn der Wert in D0 *grösser* als MINVAL ist (Übertrag-Flag ist gelöscht), springt der 68000 nach CHKMAX, wo D0 mit MAXVAL verglichen wird.
3. Wenn der Wert in D0 *kleiner* als MINVAL ist (Übertrag-Flag gesetzt), geht der 68000 über die Instruktion BCC.S CHMAX weg und speichert das Wort D0 als neues MINVAL ab.

Im Fall 2 oder 3 prüft die Schleifenbedingungsinstruktion bei CONT (DBF D1, CHKMIN), ob alle Elemente der Liste behandelt worden sind, und springt nach CHKMIN, falls das nicht der Fall ist.
Wie früher schon erwähnt, bearbeitet diese spezielle Subroutine Listen, die mit wortlangen Worten *ohne Vorzeichen* arbeiten. Wenn der Anwender das Minimum und Maximum in einer Liste, bestehend aus wortlangen Werten *mit Vorzeichen,* finden möchte, kann er einfach BCC.S CHKMAX mit BPL.S CHKMAX und BLS.S CONT mit BLE.S CONT ersetzen. Die anderen Instruktionen bleiben sich gleich.

5.3 Eine einfache Sortierungstechnik

5.3.1 Die Technik des «Bubble Sort»

Ungeordnete Daten sind für viele Anwendungen gerade richtig. Demgegenüber sind geordnete Daten einfacher zu analysieren und machen das Finden eines Elementes leichter. Wie kann nun eine ungeordnete Liste geordnet werden? Es existiert darüber eine beachtliche Menge an Literatur.
Eine der einfachsten Techniken wird «Seifenblasen sortieren» genannt («Bubble Sort»).
Gerade wie Seifenblasen gegen den Himmel hinaufsteigen, steigen die Listenelemente während des Sortierens im Speicher hoch. (Die Daten können in auf- oder absteigender Ordnung sortiert werden. Hier ist nur das aufsteigende Ordnen beschrieben).
Während des Sortierens werden die Listenelemente, beginnend mit dem ersten Element, sequentiell gelesen. Sie werden dann mit dem nächsten verglichen. Wenn ein Element grösser als

das folgende Listenelement ist, werden diese beiden ausgetauscht. Danach wird das nächste Paar verglichen und je nachdem ausgetauscht usw. Wenn der 68000 beim letzten Listenelement angekommen ist, wird das grösste Element in die letzte Listenposition hochgekommen sein.
Wenn dieser Seifenblasen-Sortierungsalgorithmus verwendet wird, muss der Mikroprozessor gewöhnlich mehrere Male durch die Liste sortieren. Im folgenden Beispiel kann das gut erkannt werden. Nehmen wir eine Liste von 5 Elementen an:

5 3 4 1 2

Nach einem Durchgang sieht diese Liste so aus:

3 4 1 2 5

Element 5 ist als grösstes der Liste ans Listenende gelangt. Der nächste Durchlauf ergibt diese Folge:

3 1 2 4 5

Element 4 ist auf den zweitletzten Listenplatz geschoben worden. Das Resultat nach dem nächsten Sortieren:

1 2 3 4 5

Dieses Beispiel zeigt nicht nur, wie der Seifenblasen-Sortieralgorithmus funktioniert, sondern gibt uns auch einen Hinweis, was für eine Leistung von diesem Algorithmus wir erwarten können. Man bemerke, dass drei Durchläufe von 5 Elementen gebraucht werden, um eine teilweise geordnete Liste zu sortieren. Ist die Liste schon zu Beginn geordnet, genügt ein Durchlauf. Umgekehrt braucht der Seifenblasen-Sortieralgorithmus 5 Durchläufe für eine Liste, die zu Beginn gerade in absteigender Ordnung vorliegt (schlimmster Fall), 4 zum Ordnen und einen zum Feststellen, dass keine weiteren Elemente auszutauschen sind. Aus dieser Beobachtung können wir festhalten, dass der 68000 von 1 bis N Durchläufe durch eine Liste von N Elementen machen muss, um diese Liste zu sortieren. Im Mittel ergibt das N/2 Durchläufe.
Wieviel Befehle und Ausführungszeit werden für einen Durchlauf benötigt? Das hängt vor allem vom verwendeten Algorithmus ab. Es gibt die beschriebene Art, die ganze Liste nach und nach zu überarbeiten, bis das Programm einen Durchlauf macht, in dem keine Elemente mehr ausgewechselt werden müssen. Mit dieser Methode wird das gesteckte Ziel erreicht, aber mit zu viel Zeitaufwand. Warum das? Der Grund liegt in der Tatsache, dass Elemente mit bereits «hochgesprudelten» Elementen noch verglichen werden. Diese Vergleiche sind nicht mehr notwendig. Eine schnellere und effizientere Methode ist, nur mit den noch nicht zu Ende bearbeiteten Elementen zu vergleichen.
Bemerke, dass für jede gegebene Liste beide Methoden gleich

viele Durchläufe brauchen. Die Differenz der Ausführungszeiten ist dagegen sehr beachtlich. Wenn wir das oben genannte Mittel von N/2 Durchläufen für eine Liste von N Elementen nehmen, können wir folgende Rechnung erstellen: Mit der ersten Methode vergleicht man in jedem der N/2 Durchläufe N Elemente. Mit der zweiten Methode vergleicht man in ebenfalls N/2 Durchläufen immer ein Element weniger als im ersten Durchgang. Das heisst, im ersten Durchlauf werden N Vergleiche, im zweiten N–1 usw. gemacht.
Während des letzten Durchlaufs werden nur noch zwei Elemente verglichen. Um ein Gefühl zu bekommen, wieviel mit der Methode an Zeit eingespart werden kann, nehmen wir folgendes an: Um eine Liste von 100 Elementen zu sortieren, braucht man mit der ersten Methode 4950 Vergleiche und mit der zweiten nur deren 3675, das bedeutet einen Viertel weniger.

5.3.2 Sortieren mit 16-Bit-Elementen

Mit dem vorangegangenen Wissen der Seifenblasenmethode wollen wir nun ein aktuelles Problem angehen. Es geht um eine Liste mit 16-Bit-Elementen ohne Vorzeichen. Bild 5.1 zeigt ein Flussdiagramm, das die benötigten Schritte dieser Aufgabe darstellt.
Wenn der Leser die Beschreibung des Seifenblasen-Sortieralgorithmus verstanden hat, sollte dieses Flussdiagramm ihm keine Mühe bereiten. Zu bemerken ist, dass ein Indikator dem 68000 signalisiert, wann die Liste komplett sortiert ist.
Dieser Indikator, genannt *Auswechsel-Flag,* wird nach jedem Durchgang geprüft. Es wird gesetzt (logisch 1), wenn mindestens noch eine Auswechslung während des vorangegangenen Durchlaufs gemacht wurde. Sonst ist dieses Flag gelöscht (logisch 0).
Die entsprechende Subroutine dieses Flussdiagramms ist im Programmbeispiel 5.4 aufgeschrieben. Wie der Leser daraus entnehmen kann, muss die Listenstartadresse im Adressregister A0 sein. Während der Ausführung der Subroutine behält A0 die Adresse des ersten Datenelements. Diese Adresse wird zu Beginn jedes Durchganges in A1 geschoben.
Neben A0 und A1 verwendet die Subroutine SORT noch weitere vier Datenregister. Bit 7 von D1 ist das Auswechsel-Flag. Register D3 ist der Zähler der nicht sortierten Elemente. D3 unterstützt D0 mit diesem Zähler zu Beginn jedes Durchgangs und wird nach jedem Durchlauf durch die DBF-Instruktion dekrementiert. D2 enthält während der Vergleichprozedur immer ein Element.
Die zwei Instruktionen nach DBF sollten übrigens noch näher betrachtet werden. Die Instruktion NOT.B D1 bildet das Einerkomplement des Auswechsel-Flags in D1, und BPL.S LOOP initialisiert einen neuen Sortierungsdurchgang, falls die NOT-

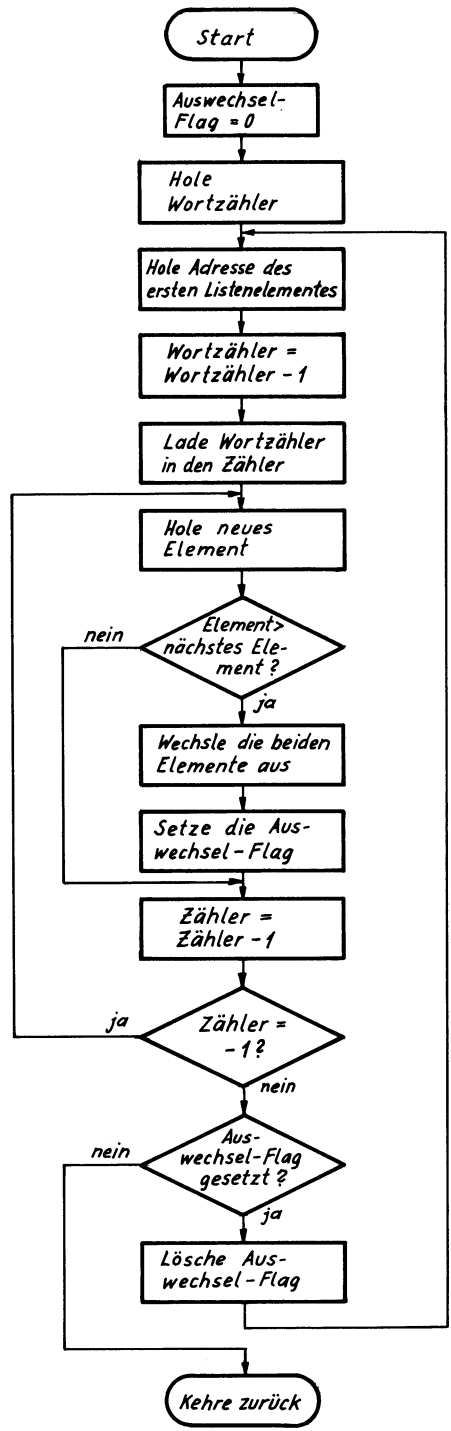


Bild 5.1
Flussdiagramm für den Sortieralgorithmus nach dem «Seifenblasenprinzip» («Bubble-sort»).

Programmbeispiel 5.4:
Eine 16-Bit-Seifenblasen-Sortierungsroutine

```
* Diese Subroutine ordnet im Speicher 16-Bit-Listenelemente in aufsteigender Reihen-
* folge. Sie verwendet den Seifenblasen-Sortieralgorithmus.
* Die Listenadresse ist in A0, die Listenlänge, in Anzahl Worten, im ersten Listenwort.
*
        ORG       $4000
SORT    MOVEM.L   D0-D3/A0/A1,-(SP)   Rette Arbeitsregister.
        CLR.B     D1                  Auswechsel-Flag = 0.
        MOVE      (A0)+,D3            Lade Wertzähler in D3.
LOOP    MOVEA.L   A0,A1               Lade Elementadresse in A1.
        SUBQ      #1,D3               Dekrementiere Wortzähler
        MOVE      D3,D0               und lade ihn in D0.
COMP    MOVE      (A1)+,D2            Bringe Element nach D2.
        CMP       (A1),D2             Nächstes Element grösser?
        BLS,S     DECCTR              Ja, fahre weiter.
        MOVE      (A1),-2(A1)         Nein, wechsle
        MOVE      D2,(A1)             diese beiden aus.
        TAS       D1                  Setze Auswechsel-Flag.
DECCTR  DBF       D0,COMP             Listenende?
        NOT,B     D1                  Ja, ist Auswechsel-Flag gesetzt?
        BPL.S     LOOP                Ja, starte neuen Durchlauf.
        MOVEM.L   (SP)+,D0-D3/A0/A1   Hole Arbeitsregister zurück.
        RTS
        END
```

Operation das Flag auf 0 gesetzt hat. Das bedeutet, dass der Sprung auf LOOP nur ausgeführt wird, falls das Auswechsel-Flag auf 1 war, bevor die NOT-Instruktion ausgeführt wurde.
In vielen Anwendungen reicht das Format 8, 16 oder 32 Bit für ein Listenelement nicht aus. Der Programmierer muss dann dafür eine spezielle Sortierungsroutine für noch längere Elemente selbst entwickeln.
Die vorangegangenen Kommentare sollten genügend Wissen gegeben haben, um ein Sortierprogramm für irgendeine Elementlänge zu entwickeln.

5.4 Geordnete Listen

Wir haben nun gelernt, eine Liste zu ordnen, und wollen jetzt betrachten, wie eine Liste nach einem bekannten Wert abzusuchen ist. Danach wollen wir zeigen, wie zwei gemeinsame Operationen, Elemente zufügen und herausnehmen, programmiert werden können.

5.4.1 Absuchen einer geordneten Liste

Wir haben gesehen, dass zum Finden eines Elementes in einer ungeordneten Liste diese sequentiell, Element für Element, abgesucht werden muss. Für eine Liste von N Elementen braucht das im Durchschnitt N/2 Vergleiche. Wenn nun eine Liste geordnet ist, kann jede Art von Suchtechnik angewendet werden. Für jede, auch für die kürzeste Liste, werden die meisten dieser Techniken schneller und effizienter sein als das sequentielle Absuchen.
Eine der bekanntesten Absuchtechniken für geordnete Listen wird «Binäres Suchen» genannt. Der Name kommt von der Tatsache, dass diese Technik die Liste in eine Serie von stetig kleineren Hälften teilt, bis das Element schliesslich gefunden wird. Das binäre Suchen beginnt in der Listenmitte und bestimmt, in welcher Listenhälfte der gesuchte Wert ist. Danach wird diese Listenhälfte genommen und halbiert usw.
Das Flussdiagramm in Bild 5.2 zeigt, wie das binäre Suchen einer geordneten Liste ausgeführt wird. Nach dem Absuchen wird als Resultat eine Adresse zurückgegeben. Wenn der gesuchte Wert in der Liste gefunden wurde, wird es die Adresse des entsprechenden Elementes sein. Falls der Wert nicht in der Liste enthalten ist, wird es die Adresse des zuletzt verglichenen Elementes sein. Man kann dann feststellen, welche der beiden Adressen herausgegeben wird, indem das Element gelesen und mit dem gesuchten verglichen wird.
Programmbeispiel 5.5 stellt eine Subroutine dar, die zum Absuchen einer geordneten Liste gebraucht werden kann. Diese Liste besteht nur aus positiven Elementwerten. Die Instruktionen von BSRCH bis CALCI führen die ersten Tests zwischen der unteren und der oberen Listengrenze durch. Diese Sequenz prüft, ob der gesuchte Wert in diesem Bereich ist oder nicht. Die restlichen Befehle (von CALCI weg) suchen die Liste mit dem nach Bild 5.2 laufenden Algorithmus ab.

Programmbeispiel 5.5:
Subroutine zum binären Suchen eines 16-Bit-Wortes

```
* Diese Subroutine sucht eine geordnete Liste nach dem wortlangen Wert in Daten-
* register D0 ab. Die Startadresse der Liste ist im Adressregister A0 und der Wort-
* zähler im ersten Listenplatz abgespeichert. Die Resultate sind in den Registern A1
* (alle 32 Bit) und D1 (untere 16 Bit) wie folgt abgespeichert:
* 1. Falls der Wert in der Liste ist, so ist D1 = 0, und A1 enthält die Adresse des
*    gesuchten Wertes in der Liste.
* 2. Falls der Wert nicht in der Liste enthalten ist, ist D1 = 0, und A1 enthält die
*    Adresse des zuletzt verglichenen Wortes.
*
```

```
        ORG     $1000
BSRCH   MOVEA.L A0,A1           Listenadresse in A1.
        CLR.L   D1              Lösche Indexregister.
* Prüfe, ob der gesuchte Wert innerhalb des Listenbereichs liegt.
        CMP     2(A1),D0        Gesuchter Wert ⩽ untere
                                Grenze?
        BHI.S   TRYHI           Nein, prüfe obere Grenze.
        BNE.S   CALCA           Ja, prüfe ob Wert > untere
        MOVEQ   #2,D1           Grenze.
CALCA   ADDQ.L  #2,A1
        RTS
TRYHI   MOVE    (A1),D1         Hole Wortzähler und
        LSL     #1,D1           mache ihn zum Byte-Index.
        CMP     0(A1,D1),D0     Gesuchter Wert > obere
        BLS.S   EQHI            Grenze?
        ADDA.L  D1,A1           Ja, berechne Adresse und
        CLR     D1              lösche D1.
        RTS
EQHI    BNE.S   CALCI           Nein, prüfe ob Wert = obere
        ADDA.L  D1,A1           Grenze.
        RTS
*
* Gesuchter Wert liegt innerhalb der Listengrenzen. Fahre mit suchen weiter.
*
CALCI   LSR     #1,D1           Teile Index durch 2.
        ANDI.B  #$FE,D1         Zwinge Index zu Wortgrenze.
        BEQ.S   RETRN           Index = 0?
        ADDA.L  D1,A1           Nein, berechne Suchadresse.
COMP    CMP     (A1),D0         Gesuchter Wert gefunden?
        BNE.S   CHKLOW
RETRN   RTS                     Ja, Ausgang m. Adresse in A1.
CHKLOW  BCC.S   CALCI           Nein, gesuchter Wert ist höher.
        LSR     #1,D1           Nein, gesuchter Wert ist tiefer.
        ANDI.B  #$FE,D1         Berechne neuen Index.
        BEQ.S   RETRN
        SUBA.L  D1,A1           Berechne neue Suchadresse
        BRA.S   COMP            und vergleiche wieder.
        END
```

Wie in früheren Beispielen in diesem Kapitel wird die Listenstartadresse über A0 der Subroutine übergeben. Die Subroutine wird diese Adresse nicht verändern. Die Resultatsadresse ist in A1 enthalten und der gefundene bzw. nicht gefundene Indikator in D1. Obwohl die BSRCH-Subroutine mit wortlangen Werten operiert, muss jedesmal der neu berechnete Index einen geraden Wert haben. Das erreicht man, indem die unteren 8 Bit des (16-

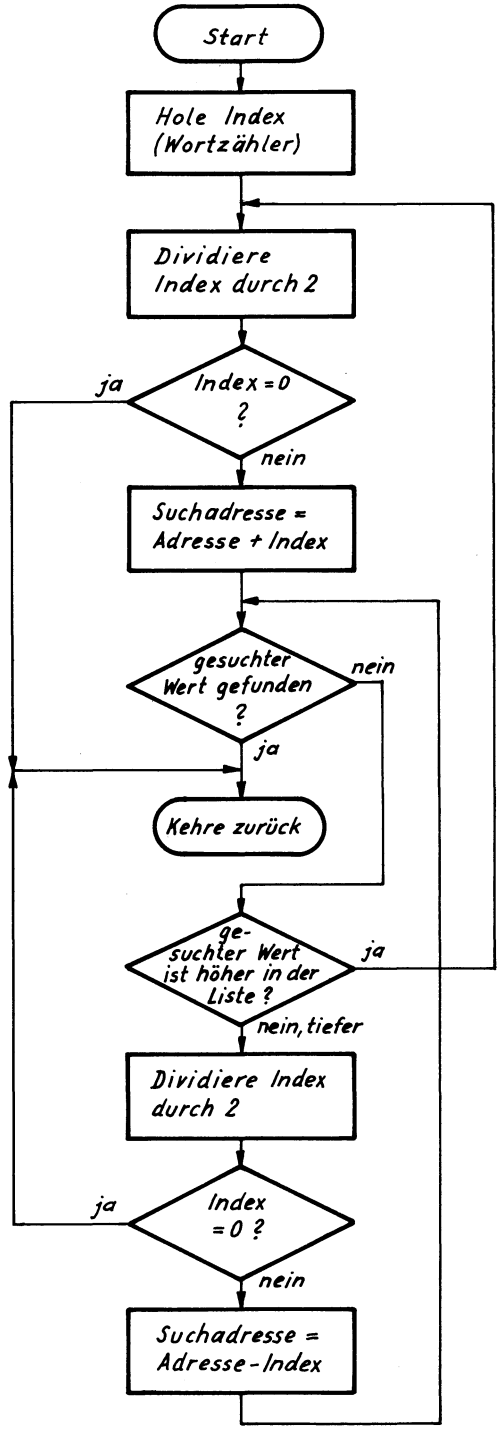


Bild 5.2
Der binäre Suchalgorithmus

Bit-)Index mit dem hexadezimalen Wert $FE UND-verknüpft wird.
Wie viel effizienter ist nun das binäre Suchen als das einfache sequentielle Vergleichen (vergleiche Beispiel 5.1)? Die mathematische Untersuchung zeigt, dass beim sequentiellen Suchen in einer Liste von N Elementen im Mittel N/2 Vergleiche gemacht werden müssen. Mit dem binären Suchen dagegen braucht es für die gleiche Liste $\log_2(N)$ Vergleiche. Für eine Liste mit 100 Elementen braucht also die sequentielle Lösung 50 und die binäre ungefähr 7 Vergleiche!

5.4.2 Zufügen eines Wertes in eine geordnete Liste

Das Zufügen eines Wertes in eine geordnete Liste kann in vier Schritte unterteilt werden:

1. Herausfinden, *wo* der Wert eingefügt werden muss.
2. Bereitstellen eines Platzes für den neuen Wert, indem alle höherwertigen Elemente um einen Platz nach oben *verschoben* werden.
3. *Einfügen* des neuen Wertes an die freigemachte Stelle.
4. *Nachführen* der Listenlänge um 1.

Die eben beschriebene Subroutine, BSRCH (Beispiel 5.5), gibt uns gerade den Ort, wo das Element eingefügt werden muss, da sie die Adresse des zuletzt verglichenen Elementes ausgibt. Wir müssen nur noch bestimmen, um Schritt 1 zu erfüllen, ob der neue Wert vor oder nach dieser Adresse eingefügt werden soll. Das kann erreicht werden, indem das letzte Element mit dem neuen verglichen wird.
Da wir nun die vier Schritte kennen, können wir eine entsprechende Subroutine entwickeln. Eine mögliche Lösung ist mit der Subroutine ADD2OL im Programmbeispiel 5.6 gegeben. Diese Subroutine beginnt mit BSRCH, um festzustellen, ob der Wert schon in der Liste ist oder nicht. BSRCH gibt eine Adresse in A1 und den gefundenen/nicht gefundenen Indikator in D1 zurück.
Nach der Rückkehr von BSRCH fragt ADD2OL das Register D1 ab und beendet die Verarbeitung, falls D1 = 0 (das neue Element ist schon in der Liste) ist. Falls D1 = 0 ist, berechnet die Subroutine die Adresse des Listenendes. Danach wird der Inhalt von A1 von dieser Adresse abgezogen und das Resultat einmal nach rechts geschoben. Der 68000 berechnet die Anzahl Worte, die im Speicher verschoben werden müssen (Element-Umladezähler), um Platz für das neue Element zu machen. Wenn das neue Element kleiner als das zuletzt verglichene ist, muss auch dieses noch verschoben werden. Damit wird der Umladezähler um 1 erhöht.
Falls das Eingabeelement grösser als das letzte Listenelement ist, dann wird die Eingabe zuhinterst angehängt. Andernfalls

Programmbeispiel 5.6:
Zufügen eines Elementes in eine geordnete Liste

```
* Diese Subroutine fügt das untere Wort von D0 zu einer geordneten Liste zu, falls der
* Wert nicht schon in der Liste vorhanden ist.
* Die Listenadresse ist in A0 und der Wortzähler im ersten Wort der Liste abgespei-
* chert. Die BSRCH-Subroutine (Beispiel 5.5) wird aufgerufen, um die Elementsuche
* auszuführen.
*
        ORG       $2000
ADD20L  MOVEM.L   D1/D2/A1/      Rette Arbeitsregister
                  A2,-(SP)
        JSR       BSRCH          Suche Liste nach Eingabe ab.
        TST       D1             Ist Eingabe schon in Liste?
        BNE.S     ITSIN          Ja, fertig.
        MOVE.L    A0,D2          Nein, berechne Listenendadresse.
        ADD.L     (A0),D2
        MOVEA.L   D2,A2          Lade Ende + 2 ins A2.
        ADDQ.L    #2,A2
        SUB.L     A1,D2          Berechne Anzahl zu verschie-
        LSR.L     #1,D2          bende Worte und
        SUBQ.L    #1,D2          subtrahiere 1 von diesem Zähler.
        CMP       (A1),D0        Verglichenes Element auch
                                 verschieben?
        BCS.L     INCCNT         Ja, erhöhe diesen Zähler.
        TST.L     D2             Nein, füge Eingabe an Listenende.
        BEQ.S     ADDIT
        BRA.S     MOVEL
INCCNT  ADDQ.L    #1,D2          Inkrementiere Umladezähler.
MOVEL   MOVE      -(A2),2(A2)    Verschiebe nächstes Wort.
        DBF       D2,MOVEL       Alle Worte verschoben?
ADDIT   MOVE      D0,(A2)        Ja, füge neues Element in die
        ADDQ      #1,(A0)        Liste ein und erhöhe Element-
                                 zähler.
ITISIN  MOVEM.L   (SP)+,D1/      Restauriere Arbeitsregister
        RTS       D2/A1/A2
        END
```

wird dieser Wert in die Liste eingefügt, was ein Verschieben aller folgenden Elemente um eine Wortposition notwendig macht. Die zwei Instruktionen lange Schlaufe bei MOVEL verschiebt, beginnend am Listenende, Element um Element. Nach der Verschiebung wird von ADDIT weg das Eingabeelement in die Liste eingefügt und der Wortzähler um 1 erhöht.

5.4.3 Löschen eines Elementes aus einer geordneten Liste

Es ist viel einfacher, ein Element aus einer geordneten Liste zu löschen, als eines neu einzufügen, weil der 68000 nur das Element *finden,* die andern Elemente *nachschieben* und den Wortzähler *dekrementieren* muss.
Das Beispiel 5.7 zeigt eine typische Lösch-Subroutine DELOL, die die BSRCH-Subroutine verwendet (Beispiel 5.5), um das zu löschende Element zu finden. Wie üblich ist die Listenstartadresse in A0 zu übergeben, der zu löschende Wert im untern Wort von D0.
Falls BSRCH den gesuchten Wert in der Liste angibt, kann die Subroutine DELOL diese Adresse und die Listenandresse verwenden, um die Anzahl Elemente zu berechnen, die verschoben werden müssen. Die zwei Instruktionen lange Schlaufe bei DELETE führt diese Operation aus. Wenn alle Elemente nachgeschoben worden sind, wird der Elementzähler im ersten Listenwort um 1 dekrementiert.

Programmbeispiel 5.7:
Löschen eines Elementes aus einer geordneten Liste

```
* Diese Subroutine löscht den Wert im unteren Wort von D0 aus einer geordneten
* Liste, falls dieser Wert in der Liste steht. Die Startadresse der Liste ist in A0 und die
* Listenlänge im ersten Wort der Liste abgespeichert. Die BSRCH-Subroutine (Bei-
* spiel 5.5) wird für die Elementsuche verwendet.
*
        ORG       $3000
DELOL   MOVEM.L   D1/D2/
                  A1,-(SP)     Rette Arbeitsregister.
        JSR       BSRCH        Suche Liste nach Element ab.
        TST       D1           Element vorhanden?
        BEQ.S     EXIT         Nein, kehre zurück.
        MOVE.L    A0,D2        Ja, berechne Listenadresse.
        ADD.L     (A0),D2
        SUB.L     A1,D2        Berechne Anzahl zu verschiebende
        LSR.L     #1,D2        Worte und
        SUBQ.L    #1,D2        subtrahiere 1 von diesem Zähler.
        BEQ.S     DECCNT
DELETE  MOVE      2(A1),(A1)+  Verschieben eines Wortes.
        DBF       D2,DELETE    Alle Worte verschoben?
DECCNT  SUBQ      #1,(A0)      Ja, dekrementiere Elementzähler.
EXIT    MOVEM.L   (SP)+,D1/    Hole Arbeitsregister zurück.
        RTS       D2/A1
        END
```

5.5 Konversionstabellen («Look-up tables»)

5.5.1 Beispiel Telefonbuch

Viele Mikroprozessorprogramme verwenden spezifische Werte, die zuerst geholt werden müssen, bevor sie verarbeitet werden können. Diese Werte können aus einem Test oder aus einer Berechnung stammen. Das könnte zum Beispiel der Sinus eines Winkels oder eine Temperatur in Grad Celsius sein. Der verlangte Wert könnte auch ein Parameter sein, der eine bestimmte Beziehung zu einem Programmeinsprung hat, die nicht berechnet werden kann. Als Beispiel kann die Telefonnummer erwähnt werden, die zu einem Namen gehört. Anwendungen wie diese lassen sich mit einer Konversionstabelle lösen. Wie der englische Name «Look-up table» schon sagt, kann für einen bekannten Wert (Argument) die entsprechende Information (Funktionswert) aus einer solchen Tabelle nachgeschaut werden.
Konversionstabellen ersetzen oft komplizierte oder zeitaufwendige Umwandlungsoperationen, wie das Berechnen der Quadrat- oder kubischen Wurzel einer Zahl oder einer trigonometrischen Funktion (Sinus, Cosinus usw.) eines Winkels. Diese Tabellen sind speziell für Funktionen anzuwenden, die nur auf einen *kleinen* Bereich des Argumentes beschränkt sind. Mit der Verwendung von Konversionstabellen muss der Mikroprozessor keine komplexen Berechnungen durchführen. Der Anwender wird bald merken, dass in den allermeisten Fällen von Beziehungen die Konversionstabellen die Ausführungszeit stark reduzieren. Es ist typisch für die Tabellen, dass sie viel Speicherplatz brauchen. Sie sind am effizientesten, falls mehr Speicherplatz zugunsten der Ausführungszeit geopfert werden kann.

5.5.2 Konversionstabellen ersetzen Gleichungen

Man kann Prozessorzeit und Programmentwicklungszeit gewinnen, indem die Resultate von komplexeren Gleichungen in Konversionstabellen abgespeichert werden. In diesem Abschnitt werden wir eine häufige Anwendung, das Berechnen des Sinus eines Winkels in Grad betrachten.
Der Sinus aller Winkel zwischen 0° und 360° kann wie in Bild 5.3b aufgezeichnet werden. Diese Kurve lässt sich mathematisch durch die in Bild 5.3a gegebene Reihe annähern.
Selbstverständlich kann ein Programm für diese Approximation entwickelt werden. Falls die Anwendung eine hohe Genauigkeit der Funktion verlangt, ist man gezwungen, dafür ein solches Programm zu schreiben, wobei dessen Ausführungszeit beachtlich lang sein wird. Für Anwendungen mit weniger hohen Anforderungen an die Genauigkeit kann jedoch eine Winkel-zu-Sinus-Konversionstabelle eingesetzt werden.

Man beachte in Bild 5.3, dass der Sinus eines jeden Winkels zwischen 0° und 180° positiv und zwischen grösser als 180° und kleiner als 360° negativ ist.

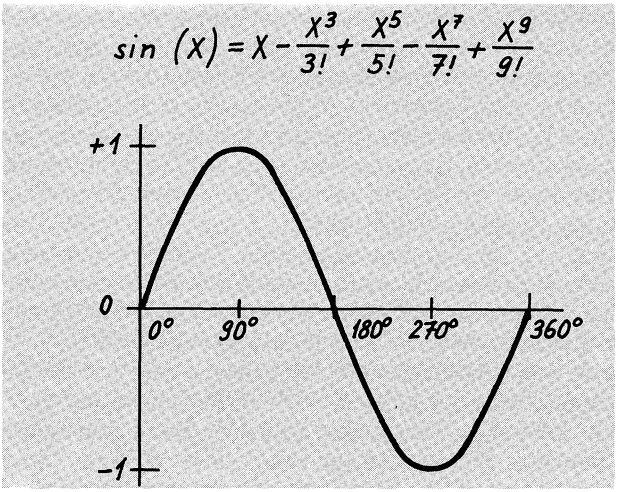


Bild 5.3
a) Mathematische Näherung für die Sinusfunktion
b) Sinusfunktion der Winkel zwischen 0° und 360°

Wie man aus Bild 5.3 weiter entnehmen kann, ist der Sinus von 91° derselbe wie der von 89°. Das Gleiche gilt auch für den Sinus von 179° und 1°. Daraus kann geschlossen werden:

für $0° \leqslant X \leqslant 90°$	→ man nehme $\sin(X)$
für $90° \leqslant X \leqslant 180°$	→ man nehme $\sin(180° - X)$
	oder $\sin(90° - [X - 90°])$

Zum Beispiel:

$$\begin{aligned}\sin(170°) &= \sin(90° - [170° - 90°]) \\ &= \sin(90° - 80°) \\ &= \sin(10°)\end{aligned}$$

Weiter gilt, dass die Winkel des 3. und 4. Quadranten einen Sinus gleicher Grösse haben, doch verschiedenes Vorzeichen gegenüber den Quadranten 1 und 2 aufweisen. Diese Beobachtung erlaubt uns, folgendes festzuhalten:

für $180° \leqslant X \leqslant 270°$	→ man nehme $-\sin(X - 180°)$
für $270° \leqslant X \leqslant 360°$	→ man nehme $-\sin(360° - X)$
	oder $-\sin(90° - [X - 270°])$

Zum Beispiel:

$$\begin{aligned}\sin(190°) &= -\sin(190°-180°)\\ &= -\sin(10°)\end{aligned}$$

$$\begin{aligned}\sin(290°) &= -\sin(90°-[290°-270°])\\ &= -\sin(90°-20°)\\ &= -\sin(70°)\end{aligned}$$

Die vorhergehenden Beziehungen zeigen uns, dass der Sinus jedes Winkels zwischen 0° und 360° mit dem Sinus zwischen 0° und 90° ausgedrückt werden kann. Für eine Konversionstabellenanwendung ist das bedeutend, weil die Tabelle nur die Sinuswerte von Winkeln zwischen 0° und 90° haben muss.
Diese Beziehungen erlauben uns also, ein Flussdiagramm für eine Winkel-zu-Sinus-Umwandlung zu konstruieren. Dieses Flussdiagramm, dargestellt in Bild 5.4, leitet den Sinus aus einem Wert, bestehend aus Grösse und Vorzeichen, ab.
Beispiel 5.8 zeigt die Winkel-zu-Sinus-Umwandlungsroutine für den 68000. Diese Subroutine nimmt Winkel zwischen 0° und 360° im Datenregister D0 an und gibt den kompletten 8-Bit-Sinuswert über D1 zurück. In dieser Subroutine SINANG wird mit der Prüfung begonnen, ob der Winkel kleiner als 181° ist. Wenn dem so ist, springt das Programm auf SINPOS. Andernfalls wird das Vorzeichenbit gesetzt. Von diesem Winkel (>180°) werden dann 180° subtrahiert.
Mit dem Vorzeichenbit in Bit 7 von D1 vergleicht die CMPI-Instruktion bei SINPOS den aktuellen Winkelwert mit 91°. Wenn der Winkel grösser oder gleich 91° ist, wird der Winkelwert von 180° subtrahiert. Die einfachste Art, diese Subtraktion auszuführen, wäre mit dem Befehl SUBI D0, #180, doch lässt der 68000 diese Subtraktion nicht zu (nur SUBI #data, Dn). Darum muss man die Subtraktion über das Zweierkomplement von D0 auf eine Addition zurückführen. Die folgenden zwei Instruktionen laden die Startadresse der Konversionstabelle (SINTAB) in A0. Danach wird der Sinus aus der Tabelle herausgelesen, indem das Adressregister indirekt mit der Indexadressierung eingesetzt wird. Das Ganze wird noch mit dem Vorzeichenbit in D1 ergänzt. Die SINTAB-Tabelle enthält 91 Byte Sinuswerte, um alle Grade zwischen 0° und 90° darstellen zu können. Tabelle 5.1 enthält die SINTAB-Werte.
Die SINANG-Subroutine braucht 19 Speicherworte. Ihre Ausführungszeit hängt davon ab, in welchem Quadaranten der Winkel liegt. In den folgenden Ausführungszeiten sind die JSR- und RTS-Instruktionen nicht inbegriffen.

Winkel zwischen	Anzahl Zyklen	Ausführungszeit (µs)
0° und 90°	62	7,75
91° und 270°	72	9,0
271° und 360°	82	10,25

Programmbeispiel 5.8:
Finden des Sinus eines Winkels

```
* Dieses Programm berechnet den binären Sinuswert eines Winkels (0...360°), der im
* unteren Wort von D0 abgespeichert ist. Es wird eine Konversionstabelle verwendet.
* Der mit Vorzeichen versehene Sinus wird im unteren Byte von D1 zurückgegeben.
* D0 bleibt unverändert.
*
          ORG       $1000
SINANG    MOVE      D0,-(SP)       Rette Arbeitsregister.
          MOVE.L    A0,-(SP)
          CLR.B     D1             Lösche das Sinus-Byte.
          CMPI      #180,D0        Ist der Winkel < 181°?
          BLS.S     SINPOS         Ja, Vorzeichen = 0.
          TAS       D1             Nein, setze Vorzeichen = 1.
          SUBI      #180,D0        Subtrahiere vom Winkel 180°.
SINPOS    CMPI      #91,D0         Ist der Winkel < 91°?
          BMI.S     GETSIN         Ja, hole den Sinus.
          NEG       D0             Nein, subtrahiere
          ADDI      #180,D0        vom Winkel 180°.
GETSIN    LEA       SINTAB,A0      Lade Tabellenadresse,
          OR.B      0(A0,D0),D1    gebe Vorzeichen dazu.
          MOVE.L    (SP)+,A0       Hole Arbeitsregister zurück.
          MOVE      (SP)+,D0

* Sinustabelle (wie Tabelle 5.1)

SINTAB    DC.B      0,2,4,8,$B,$D,$F,$11,$14,$16...
          .
          .
          .
          usw.      (Rest der Tabelle, total 91 Byte)
```

Winkel	Sinus		Winkel	Sinus	
Grad	dezimal	dual	Grad	dezimal	dual
0.00	.0000	00000000	45.00	.7071	01011010
1.00	.0175	00000010	46.00	.7193	01011100
2.00	.0349	00000100	47.00	.7313	01011101
3.00	.0523	00000110	48.00	.7431	01011111
4.00	.0698	00001000	49.00	.7547	01100000
5.00	.0872	00001011	50.00	.7660	01100010
6.00	.1045	00001101	51.00	.7771	01100011
7.00	.1219	00001111	52.00	.7880	01100100
8.00	.1392	00010001	53.00	.7986	01100110
9.00	.1564	00010100	54.00	.8090	01100111
10.00	.1736	00010110	55.00	.8191	01101000

Winkel	Sinus		Winkel	Sinus	
Grad	dezimal	dual	Grad	dezimal	dual
11.00	.1908	00011000	56.00	.8290	01101010
12.00	.2079	00011010	57.00	.8387	01101011
13.00	.2250	00011100	58.00	.8480	01101100
14.00	.2419	00011110	59.00	.8572	01101101
15.00	.2588	00100001	60.00	.8660	01101110
16.00	.2756	00100011	61.00	.8746	01101111
17.00	.2924	00100101	62.00	.8829	01110001
18.00	.3090	00100111	63.00	.8910	01110010
19.00	.3256	00101001	64.00	.8988	01110011
20.00	.3420	00101011	65.00	.9063	01110100
21.00	.3584	00101101	66.00	.9135	01110100
22.00	.3746	00101111	67.00	.9205	01110101
23.00	.3907	00110010	68.00	.9272	01110110
24.00	.4067	00110100	69.00	.9336	01110111
25.00	.4226	00110110	70.00	.9397	01111000
26.00	.4384	00111000	71.00	.9455	01111001
27.00	.4540	00111010	72.00	.9511	01111001
28.00	.4695	00111100	73.00	.9563	01111010
29.00	.4848	00111110	74.00	.9613	01111011
30.00	.5000	01000000	75.00	.9659	01111011
31.00	.5150	01000001	76.00	.9703	01111100
32.00	.5299	01000011	77.00	.9744	01111100
33.00	.5446	01000101	78.00	.9781	01111101
34.00	.5592	01000111	79.00	.9816	01111101
35.00	.5736	01001001	80.00	.9848	01111110
36.00	.5878	01001011	81.00	.9877	01111110
37.00	.6018	01001101	82.00	.9903	01111110
38.00	.6157	01001110	83.00	.9926	01111111
39.00	.6293	01010000	84.00	.9945	01111111
40.00	.6428	01010010	85.00	.9962	01111111
41.00	.6561	01010011	86.00	.9976	01111111
42.00	.6691	01010101	87.00	.9986	01111111
43.00	.6820	01010111	88.00	.9994	01111111
44.00	.6947	01011000	89.00	.9998	01111111
45.00	.7071	01011010	90.00	.1.0000	01111111

Tabelle 5.1
Sinuswerte für Winkel ganzer Grade

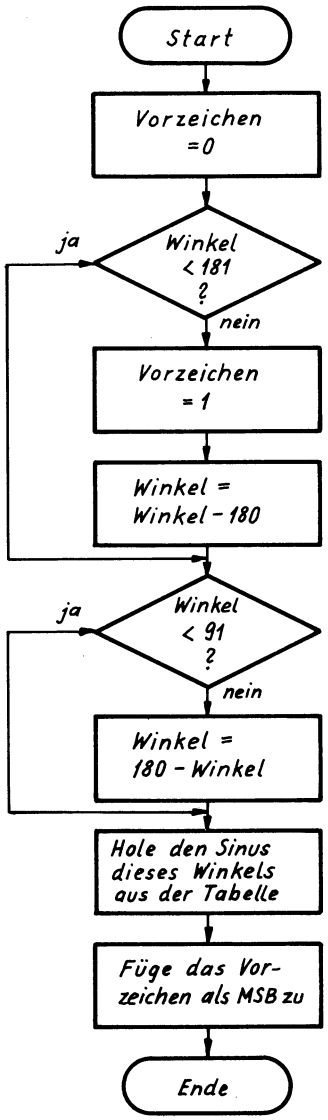


Bild 5.4
Flussdiagramm zu programmbeispiel 5.8, bei dem die Sinusfunktion mit Hilfe einer Konversionstabelle bestimmt wird.

5.5.3 Konversionstabellen führen Codewandlungen durch

Konversionstabellen können auch codierte Daten enthalten, wie zum Beispiel Anzeigecodes, Druckercodes oder Mitteilungen. Als Beispiel 5.9 nehmen wir eine Subroutine, die ein Mehrfaches an Konversionen zulässt. Sie wandelt eine hexadezimale

Programmbeispiel 5.9:
Eine Code-Umwandlungssubroutine

```
* Diese Subroutine verwendet drei Nachschlagetabellen, um eine hexadezimale Ziffer
* im unteren Byte von D0 in ASCII-, BCD- und Gray- Code umzuwandeln. Die um-
* gewandelten Werte werden in drei aufeinanderfolgenden Speicherbyte, beginnend bei
* der Adresse in A0, zurückgegeben. D0 und A0 werden von der Subroutine nicht
* verändert.
*
          ORG       $1000
LOOKUP    MOVE      D0,-(SP)          Rette Arbeitsregister.
          MOVE.L    A1,-(SP)
          EXT.W     D0
          LEA       ATABLE,A1         A1 zeigt auf die Tabelle.
          MOVE.B    0(A1D0),(A0)      Hole ASCII-Code.
          MOVE.B    $10(A1,D0),1(A0)  Hole BCD-Code.
          MOVE.B    $20(A1,D0),2(A0)  Hole Gray-Code.
          MOVE.L    (SP)+,A1          Hole Arbeitsregister
          MOVE      (SP)+,D0          zurück.
          RTS
ATABLE    DC.B      '0123456789ABCDEF'
          DC.B      0,1,2,3,4,5,6,7,8,9,$10,$11,$12,$13,$14,$15
          DC.B      0,1,3,2,6,7,5,4,$C,$D,$F,$E,$A,$B,9,8
          END
```

Ziffer im unteren Byte von D0 um in ASCII-, BCD- und Gray-Code. Die umgewandelten Werte werden in drei aufeinanderfolgenden Speicherbyte zurückgegeben. Die Startadresse dafür liegt im Adressregister A0.

5.6 Sprungtabellen

Eine Konversionstabelle kann mehr als nur Daten enthalten. In vielen Fällen sind die Tabellenelemente Adressen. Eine Fehlerroutine kann zum Beispiel eine Konversionstabelle verwenden, um die Startadresse einer Operationsfehlermeldung, basierend auf einem Code in einem Datenregister, zu finden. Auch eine Interruptroutine kann eine Konversionstabelle zum Aufrufen von verschiedenen Serviceroutinen verwenden, abhängig vom Gerät, das die Interruptanforderung generierte. Andere Routinen könnten eine Konversionstabelle zum Aufrufen von verschiedenen Steuerprogrammen gebrauchen, die mit Hilfe von Tasten ausgewählt wurden. In all diesen Anwendungen ist die Konversionstabelle, bestehend aus Adressen, als Sprungtabelle eingesetzt. Sprungtabellen werden vor allem verwendet, wenn die Programmsteuerung vom Zustand einer bestimmten Bedingung abhängig ist.

Beispiel 5.10 zeigt, wie eine Sprungtabelle die Bedürfnisse von fünf verschiedenen Anwendern in einem Multiterminal-Mikrocomputersystem befriedigen kann. Die Subroutine SELUSR interpretiert den Inhalt von D0 als Anwenderidentifikation und braucht diesen Code, um eine der Anwenderserviceroutinen aufzurufen. SELUSR prüft die Gültigkeit des Eingabecodes und geht auf die CHK-Ausnahmeroutine, falls der Code grösser als vier ist (mehr über Ausnahmen in Kapitel 7). Die Subroutine wandelt die gültige Anwenderidentifikation in einen Index um, mit dem die Adresse einer Anwenderroutine (USER0 ... USER4) ins Register A0 geholt werden kann.

Programmbeispiel 5.10:
Eine Multianwender-Auswählsubroutine

```
* Diese Subroutine ruft eine der fünf Anwendersubroutinen auf, die mit einer An-
* wenderidentifikation im unteren Byte von D0 aktiviert wird. Die Subroutine ver-
* ändert A0 und D0.
*
        RORG    $1000
SELUSR  EXT.W   D0              Benützercode in Wort.
        CHK     #4,D0           Falsche Identifikation ID?
        LSL     #2,D0           Nein, berechne Index (ID*4).
        LEA     UADDR,A0        Lade Tabellenbeginn in A0.
        MOVEA.L 0(A0,D0.W),A0   Hole Benützeradresse und
        JMP     (A0)            springe auf diese Subroutine.
UADDR   DC.L    USER0,USER1,USER2,USER 3,USER4
        END
```

Diese Aktion benützt den relativen Programmzähler mit Indexadressierung. Dieser Modus wird durch die RORG-Zuweisung zu Beginn der Subroutine aktiviert. Mit der richtigen Adresse in A0 kann ein einfacher, indirekter Sprung die Steuerung des Programms auf die Anwendersubroutine übertragen.

6. Hardware des Mikroprozessors 68 000

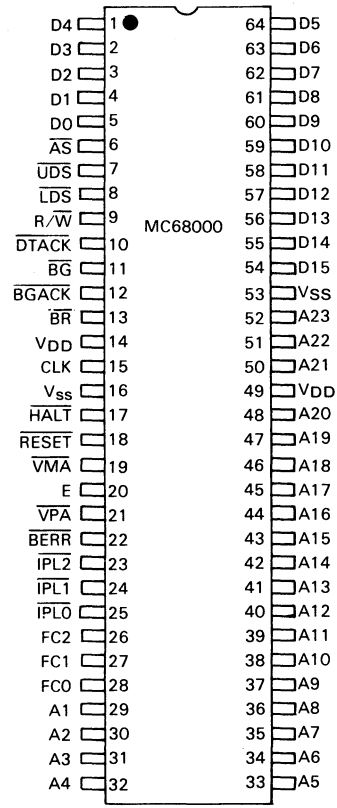


Bild 6.1
Anschlussbelegung des Mikroprozessors 68000

Der 68000-Baustein ist in einem 64poligen, zweireihigen Gehäuse untergebracht. Die Anschlussbelegung ist aus Bild 6.1 herauszulesen. Um die Verwechslung von Signalen mit «logisch 0» und «logisch 1» sowie «hoch» und «tief» auszuschalten, sprechen wir im weiteren von aktiven Signalen, wenn sie «wahr» sind, und von inaktiven, wenn sie «unwahr» sind.
Die externen Signale des 68000 werden in Funktionsgruppen beschrieben, damit sie etwas besser verstanden werden können. Diese Gruppen sind in Bild 6.2 aufgeführt.

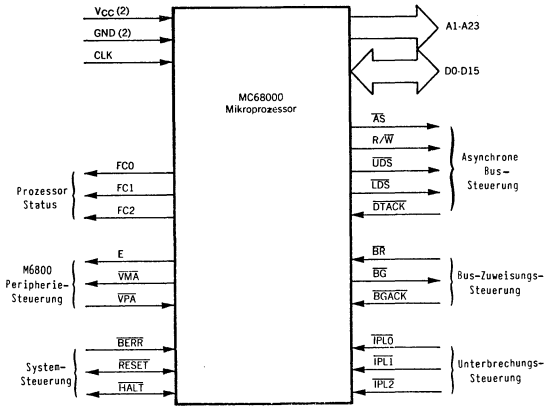


Bild 6.2
Nach Funktionsgruppen geordnete Anschlüsse des 68000

6.1 Takt-, Speisungs- und Masseleitungen

Der 68000 ist über je zwei Anschlüsse für +5 V und Masse gespeist (+5 V = V_{DD}; Masse = V_{SS}). Der Takt CLK ist ein TTL-Eingang, der Frequenzen bis zu 10 MHz verarbeitet.

6.2 Der Daten- und Adressenbus

Der 68000 ist ein 16-Bit-Mikroprozessor, da seine Informationsgrundeinheit, das Wort, 16 Bit breit ist. Er kann nur eine 16 Bit lange Information von oder zum Speicher und I/O-Baustein gleichzeitig transferieren. Um mehr als 16 Bit zu übertragen, müssen weitere Transfers ausgeführt werden. Sämtliche Informationsaustausche zwischen dem 68000 und peripheren Bausteinen werden über den bidirektionalen *16-Bit-Datenbus (D0 ... D15)* abgewickelt.
Wie wird ein Systembaustein vom 68000 adressiert, um Informationen gegenseitig auszutauschen? Der 68000 selektiert einen externen Baustein, indem er eine einzige Adresse an den *23 Bit breiten Adressenbus (A1 ... A23)* legt. Über diesen Adressenbus kann der 68000 8 388 608 Speicherworte (zu 16 Bit) anwählen. Mit den Signalen UDS und LDS können das obere bzw. das untere Byte eines Wortes noch unterschieden werden (siehe Kapitel 6.4 «Asynchrone Buskontrolle»). Der 68000 gibt mit dem (address strobe) Adressensignal AS der Peripherie bekannt, dass eine gültige Adresse auf dem Bus ist.

6.3 Funktionsstatussignale

Jedesmal, wenn der 68000 mit externen Bausteinen kommuniziert, gibt er zusätzlich zu den Adressen die drei Signale FC0, FC1 und FC2 als weitere Information aus (function code). Die-

se Funktionsstatussignale teilen der Peripherie mit, ob der 68000 den Daten- oder Programmspeicher adressiert, im Anwender- oder Überwachungsstatus ist oder gerade eine Unterbrechung bearbeitet. Die Tabelle 6.1 enthält die verschiedenen Kombinationen dieser drei Signale. Beachte das höchstwertige Bit FC2, das den Status des Überwachungsbit S im Statusregister wiedergibt.

Funktionsstatussignale

FC2	FC1	FC0	Bedeutung	Zuteilung
0	0	0	Reserviert	Anwender
0	0	1	Datenbereich	Anwender
0	1	0	Programmbereich	Anwender
0	1	1	Reserviert	Anwender
1	0	0	Reserviert	Systemüberwachung
1	0	1	Datenbereich	Systemüberwachung
1	1	0	Programmbereich	Systemüberwachung
1	1	1	Unterbrechungs-quittung	Systemüberwachung

Tabelle 6.1
Funktionsstatussignale informieren externe Bausteine über den Status des 68 000.

Die Funktionsstatussignale zeigen an, dass ein Programmabschnitt adressiert worden ist, falls der Programmzähler PC die Adressenquelle ist oder falls der Startvektor geholt worden ist. Sie können auch angeben, dass ein Datenbereich adressiert wird, falls die meisten Operanden gelesen (PC ist *nicht* Adres-

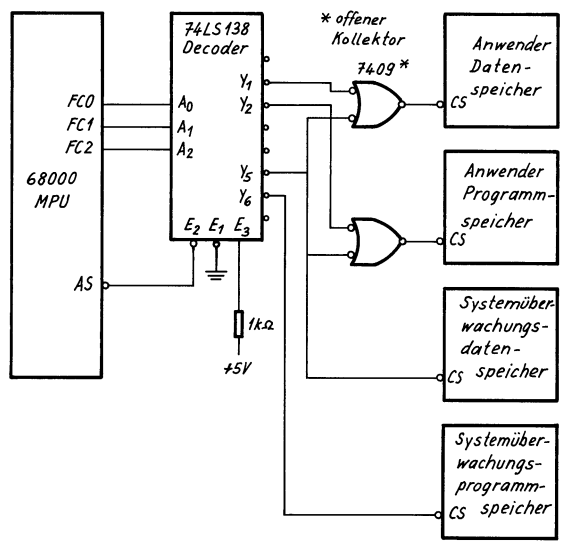


Bild 6.3
Speicherunterteilung mit den Funktionsstatussignalen

senquelle), falls alle Operanden geschrieben oder falls andere Vektoren als der Startvektor geholt wurden. Die Funktionsstatussignale können mit den Adressen zusammen für die Schreibsperre spezifischer Speicherabschnitte verwendet werden. Sie können auch mit externen Einheiten wie zum Beispiel einer Speicherverwaltungseinheit eingesetzt werden, um bestimmte Operationen im richtigen Prozessorstatus durchführen zu lassen. Im weiteren können die Funktionsstatussignale für die externe Speichererweiterung bis auf 64 MByte (4 Segmente zu 16 MByte) verwendet werden! Das Bild 6.3 zeigt eine Möglichkeit, wie diese Speichersegmentation realisiert werden kann.

6.4 Asynchrone Bussteuerung

Einige übliche 8-Bit-Mikroprozessoren, wie der 6800 und der 6502, können nur mit synchron betreibbaren Einheiten kommunizieren. Diese Mikroprozessoren sind so entwickelt worden, dass die externen Bausteine innerhalb einer gegebenen Zeit Ausgabedaten entgegennehmen oder Eingabedaten bereitstellen *müssen*. Die Kommunikation mit langsameren oder asynchronen Bausteinen erfordert jedesmal spezielle Hard- und Softwareschnittstellen. Der 68000 dagegen kann ohne weiteren Aufwand direkt mit synchronen *oder* asynchronen Bausteinen verbunden werden. Er ist mit je einem Satz Steuerleitungen für jeden Typ ausgerüstet.

6.4.1 Die asynchronen Steuerleitungen

Wie wir wissen, kann der 68000 mit den einzelnen Byte innerhalb eines Wortes arbeiten. So sprechen wir normalerweise auch von der 16-MByte- und nicht von der 8-MWort-Adressierung. Wie werden nun die einzelnen Byte adressiert? Zusätzlich zu den Adressenbit gibt es noch zwei spezielle Steuersignale:
- für das höhere Byte UDS (Upper Data Strobe)
- für das untere Byte LDS (Lower Data Strobe).

Wenn UDS vom 68000 aktiv (logisch 0) gesetzt wird, wird die Information auf den höheren 8 Bit des Datenbusses (D8 ... D15) transferiert. Wenn die Information auf den unteren 8 Datenbit (D0 ... D7) verkehren soll, setzt der 68000 das Signal LDS aktiv (logisch 0). Während eines Worttransfers sind beide Signale (UDS und LDS) aktiv, und die Information geht über den ganzen Datenbus (D0 ... D15).
Wie kann ein adressierter, externer Baustein wissen, ob der 68000 Informationen haben (lesen) oder ausgeben (schreiben) will? Durch das Steuersignal Lesen/Schreiben (R/W) kann der externe Baustein erkennen, in welcher Richtung sich der Transfer abspielen soll. Das Steuersignal R/W ist logisch 1 während Lesezyklen und logisch 0 während Schreibzyklen.

Jedesmal, wenn ein externer Baustein entweder Daten auf den Datenbus gibt (Leseoperation) oder von ihm holt (Schreiboperation), lässt der Baustein den 68000 mit dem Signal DTACK (Data Transfer Acknowledge) wissen, wann der Transfer fertig ist. Wenn der Prozessor das Signal DTACK während eines Lesezyklus erkennt, speichert er die Daten ab und beendet den Buszyklus. Da dieses Terminieren auf DTACK angewiesen ist, hängt die Geschwindigkeit, mit welcher der 68000 Daten transferieren kann, von der Zugriffszeit des adressierten Bausteins ab. Das bedeutet also, dass der 68000 mit langsamerer Peripherie auch langsamer arbeitet und entsprechend mit schnellerer Peripherie rascher. Die maximale Übertragungsrate, gegeben durch den Systemtakt, kann natürlich nicht überschritten werden.

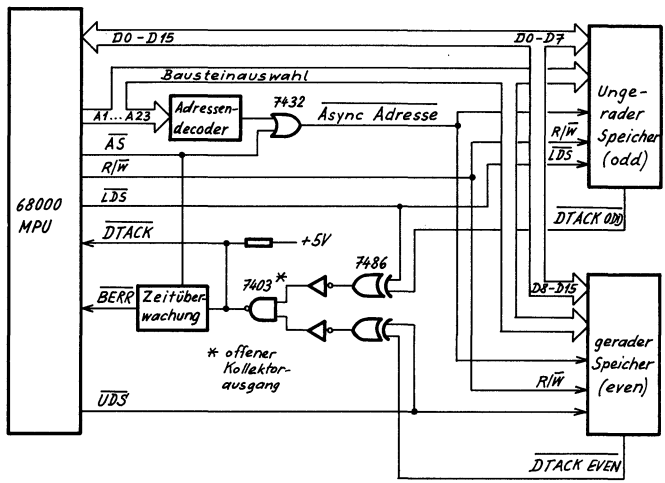


Bild 6.4
Byteadressierung auf dem asynchronen Bus

Das Bild 6.4 zeigt alle Signale, die nötig sind, um auf einen asynchronen Speicher zugreifen zu können. Das Bild enthält neben der CPU und den beiden Speichern noch eine Zeitüberwachungsschaltung («Watchdog»). Sie lässt eine bestimmte Zeit zwischen dem Anlegen von AS und dem Erhalten von DTACK zu. Wenn die beiden Speicher die richtige Kombination von DTACK ODD und DTACK EVEN innerhalb der gegebenen Zeit abgeben, wird das DTACK-Signal zum Mikroprozessor geschickt. Werden diese Bedingungen nicht eingehalten, erzeugt die Überwachungsschaltung das BERR-Signal (Bus Error), das im 68000 die Ausnahmebehandlung dafür initialisiert. Auf diese Weise kann das System vor dem «Hängenbleiben» wegen eines fehlerhaften Peripheriebausteines bewahrt werden.

6.4.2 Zeitbedingungen für asynchrone Datenübertragung

Nach den asynchronen Steuersignalen wollen wir die zeitlichen Bedingungen während einer Datentransferoperation näher betrachten. Bild 6.5 zeigt uns das Zeitverhalten der entsprechenden Signale während normaler, wortlanger Lese- und Schreibzyklen und eines langsamen (mit verzögertem DTACK) Lesezyklus. Diese Impulsdiagramme sind zeitlich auf den Systemtakt CLK (Clock, Eingangstakt des 68000) bezogen gezeichnet. Mit 8 MHz hat CLK eine Periode von 125 ns. Ein normaler (unverzögerter) Lesezyklus dauert dann 4 Perioden oder 500 ns. Wegen der internen Verzögerungszeiten und der Notwendigkeit, das Signal R/W auf logisch 0 zu bringen, braucht der Schreibzyklus eine weitere Periode, also 625 ns bei 8 MHz.

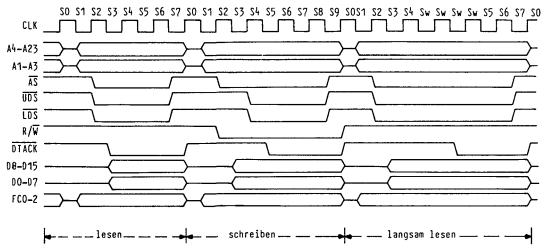


Bild 6.5
Zeitdiagramm für asynchronen Worttransfer

Der 68000 kann DTACK jederzeit nach Aktivwerden von AS entgegennehmen. Er erwartet aber, DTACK vor der 5. Periode (Lesen) oder 7. Periode (Schreiben) des Systemtaktes nach Zyklusbeginn zu empfangen. Wenn DTACK nicht vor diesen Zeitpunkten erkannt wird, fügt der 68000 Warteperioden (Wait States) in den Lese- oder Schreibzyklus ein. Auf der rechten Seite von Bild 6.5 ist das Zufügen von Warteperioden zu einem Lesezyklus dargestellt.
Die Zeitdiagramme für Byteübertragungen sind den für Wortübertragungen ähnlich, ausser dass nur eines der beiden Byteselektionssignale (UDS oder LDS) aktiv ist und dass nur eine Hälfte des Datenbusses gültige Information führt. Die andere Hälfte des Datenbusses wird im hochohmigen Zustand bleiben. Das aktive Byteselektionssignal wird vom internen Signal A0, dem untersten Bit des Programmzählers, abgeleitet. In der Benützeranleitung des 68000 (Kapitel 4.2.1) steht mehr über die zeitlichen Abläufe der Byteübertragung.

6.5 Synchrone Steuersignale

Der 68000 hat drei Steuersignale, um synchrone, periphere Bausteine wie jene aus den 6500- und 6800-Familien am Mikro-

prozessor anschliessen zu können. Es sind dies die Signale E (Enable, Freigabe), VPA (Valid Peripheral Address, gültige Peripherieadresse) und VMA (Valid Memory Address, gültige Speicheradresse).

Das *Freigabesignal E* ist ein Takt, mit dem die 8-Bit-Peripheriebausteine den Datentransfer synchronisieren. Dieser freilaufende Takt entspricht dem E- oder Φ 2-Signal der bestehenden 6500- und 6800-Systeme. Der E-Takt läuft mit einer Frequenz von einem Zehntel des 68000-Systemtaktes. In unserem Fall von 8 MHz heisst das 800 kHz für den Takt E. Im weiteren hat E ein Tastverhältnis von 60 zu 40, logisch 0 für 6 Taktperioden und logisch 1 für 4 Perioden.

Die *gültige Peripherieadresse VPA* ist ein Eingangssignal, das dem 68000 mitteilt, dass ein 6800-Peripheriebaustein adressiert wurde und dass die Datentransferoperation durch das Freigabesignal E synchronisiert werden sollte. Normalerweise wird VPA aus der decodierten Adresse und AS gewonnen. Zu bemerken ist, dass VPA für die synchrone dasselbe ist wie DTACK für die asynchrone Übertragung.

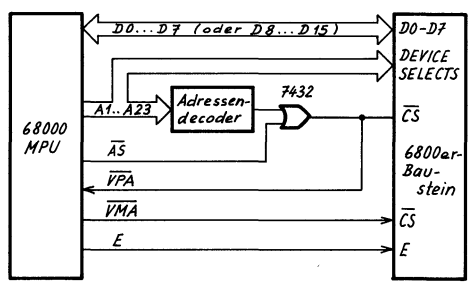


Bild 6.6
Anschluss von Peripheriebausteinen der 68 000er-Serie

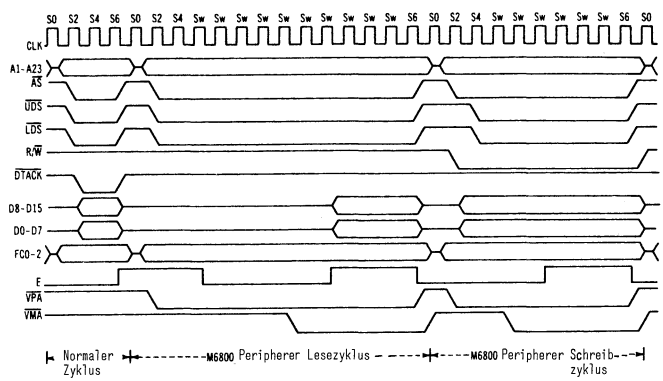


Bild 6.7
Zeitdiagramm für synchronen Datentransfer (8 Bit)

Wenn AS noch anliegt, währenddem der 68000 VPA empfängt, reagiert der Prozessor mit dem Senden von *VMA (gültige Speicheradresse),* mit der der periphere Baustein die endgültige Selektion vornehmen kann.
Bild 6.6 zeigt die Signale auf, die für das Anschliessen eines 6800-Peripheriebausteins am 68000 normalerweise verwendet werden. In Bild 6.7 sind die Zeitdiagramme für einen synchronen Lese- und Schreibzyklus dargestellt. Das Kapitel 8 enthält weitere Anregungen, wie synchrone 8-Bit-Bausteine an den 68000 angeschlossen werden können.

6.6 Busaussperrungssignale

Die Busaussperrungssignale (englisch: bus arbitration signals, d.h. «Bus-Schiedsrichter-Signale») werden für direkte Speicherzugriffe (DMA, Direct Memory Access) und Multiprozessoranwendungen verwendet, um die Steuerung des Systembusses von einem 68000-Mikroprozessor an einen externen Baustein zu übergeben. Bei all diesen Anwendungen wünscht der externe Baustein die Bussteuerung zu übernehmen, indem er dies dem 68000 mit dem Signal *BR (Bus Request,* Busanforderung) mitteilt. Der 68000 hat immer die geringere Buspriorität als ein externer Baustein und wird die Steuerung des Busses nach Beenden des momentanen Zyklus abgeben. Nach Erkennen von BR synchronisiert der 68000 intern und zeigt die Annahme der Anforderung mit dem Signal *BG (Bus Grant,* Bus freigegeben) an. Wenn mehrere Bussteuerungsanforderungen anliegen, muss eine externe Schaltung dafür sorgen, dass nur einer der Bausteine das Signal BG empfangen kann.
Der anfragende Baustein wartet nach Erhalten von BG auf die Beendung des vom Prozessor angefangenen Zyklus (zum Beispiel auf das Zurücksetzen von AS und DTACK) und gibt dem 68000 danach *BGACK (Bus Grant, Acknowledge,* Busfreigabeerkennung) zurück. Zwischen dem 68000 und dem anfragenden Baustein spielt sich also folgender Dialog ab: Mit dem Anlegen von BR sagt der Anfrager: «Ich will den Bus haben.» Mit BG antwortet der 68000: «Du kannst den Bus haben.» Am Ende des momentanen Zyklus gibt der Anfrager das Signal BGACK zum Prozessor und dem restlichen System mit der Bedeutung «Ich habe nun die Bussteuerung übernommen» aus.
Am Ende dieses Dialogs nimmt der neue Busmaster seine Anfrage mit dem Zurücksetzen von BR zurück. In gleicher Weise setzt der Prozessor BG zurück und wartet auf das Beenden der Busoperation des externen Bausteins. Zu diesem Zeitpunkt setzt der externe Baustein das Signal BGACK zurück. Damit kann der Prozessor seine normale Funktion und Arbeit wieder aufnehmen. Die Zeitdiagramme dieser ganzen Sequenz sind in Bild 6.8 dargestellt.

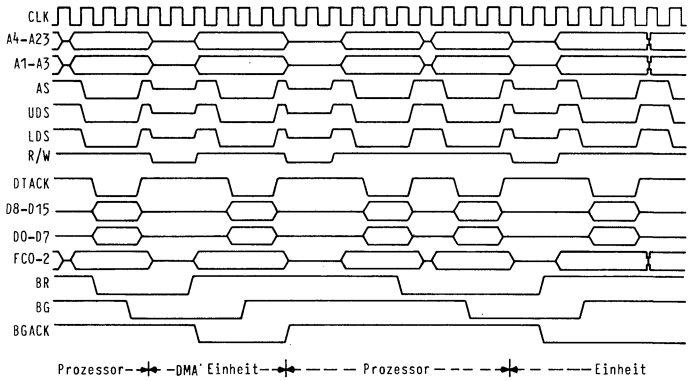


Bild 6.8
Zeitdiagramm für Bussteuerung

6.7 Systemsteuerungssignale

Der 68000 hat drei Systemsteuerungssignale. Eines ist als Eingang und die andern beiden bidirektional definiert.
RESET (Reset, Zurücksetzen) ist ein bidirektionales Signal, das dem Prozessor oder einem externen Baustein erlaubt, das ganze System zurückzustellen. Ein Zurücksetzen durch den Prozessor, mit Hilfe des Befehls RESET, wird das Signal RESET für 124 Systemtaktperioden setzen und danach wieder zurückstellen. Das gibt allen externen Bausteinen genügend Zeit, sich zurückzusetzen. Der interne Zustand des 68000 wird dabei allerdings *nicht* verändert.
Während eines katastrophalen Fehlers kann das ganze System (Prozessor und alle externen Bausteine) zurückgesetzt werden, wenn beide bidirektionalen Signale RESET und *HALT* während mindestens 100 ms am 68000 anliegen. Das veranlasst den 68000, einen «Speisungsreset» zu starten, währenddem der Prozessor in den Überwachungszustand übergeht und über den Vektor der tiefsten Speicherstufe eine Rückstellroutine startet. (Diese Sequenz und weitere Ausnahmesituationen sind in Kapitel 7 ausführlicher diskutiert.)
HALT muss aber nicht notwendigerweise von RESET begleitet werden. HALT alleine, als Prozessoreingang, kann zu Testzwecken für die Einzelschrittbetriebsart verwendet werden. Die Schaltung in Bild 6.9 zeigt uns eine mögliche Realisiserung dieser Funktion. Wenn der Ablauf/Einzelschritt-Schalter in der Position «Einzelschritt» ruht, wird der Prozessor den momentanen Befehlszyklus beenden und wieder anhalten. Das wird jedesmal, wenn der Einzelschritt/Warte-Schalter auf Einzelschritt umgeschaltet wird, so sein. Wenn der Prozessor gestoppt wird, sind der Adress-, der Daten- und der Funktionsstatusbus in hochohmigem Zustand und die Bussteuerungsleitungen inak-

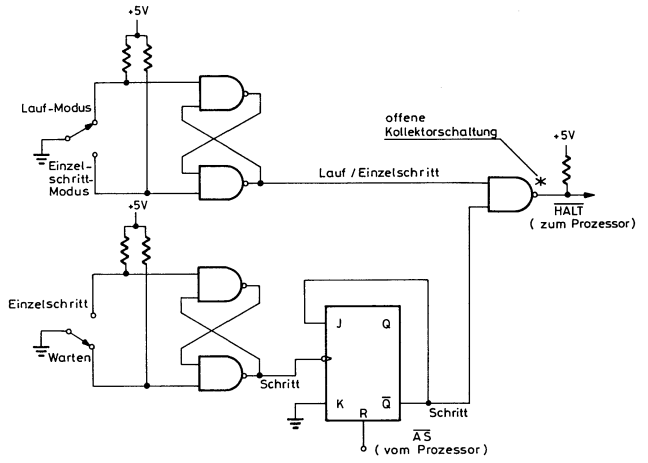


Bild 6.9
Einzelschrittsteuerung mit HALT-Signal

tiv gesetzt (die allgemeine Bussteuerung mit BR, BG und BGACK ist immer zur Verfügung).
Das HALT-Signal kann auch vom 68000 als *Ausgang* verwendet werden. Der Prozessor kann wegen eines Doppelbusfehlers (siehe Kapitel 7) gestoppt werden und setzt dann HALT als Ausgang aktiv.
Das HALT-Signal kann mit dem Systemsteuersignal BERR (Bus Error) zusammen auch als Eingang verwendet werden. Die Eigenschaft von BERR ist, den Prozessor über systeminterne Probleme zu informieren. Das heisst, dass BERR das Auftreten von unerwünschten Vorfällen (zum Beispiel unerwartete Interrupts oder illegale Speicherzugriffsanforderungen) oder das Nichtauftreten von erwarteten Signalen (zum Beispiel gibt ein externer Baustein kein DTACK oder VPA zurück) anzeigt.
Wenn der 68000 das Signal BERR erkennt, kann er entweder ein Busfehlerprogramm aufrufen (siehe Kapitel 7) oder den Buszyklus noch einmal wiederholen. Der Prozessor wird den Buszyklus repetieren, wenn HALT von aussen mit dem Auftreten von BERR gesetzt wird. Um einen Buszyklus wiederholen zu können, wird der Prozessor den Zyklus zuerst beenden, danach anhalten und den Adress- und Datenbus, die Funktionsstatus- und die Steuerungsleitungen in den hochohmigen Zustand versetzen. Wenn die externe Schaltung BERR und HALT zurücksetzt, wird der 68000 darauf den vorhergehenden Buszyklus wiederholen. Es gibt eine einzige Ausnahme: Die Instruktion TAS kann in dieser Situation nicht wiederholt werden.

6.8 Interrupt-Steuersignale

Externe Bausteine können Interrupt-Anforderungen an den 68000 senden, indem die codierte Prioritätsanforderungsstufe

an die drei Interrupt-Steuereingänge IPL0, IPL1 und IPL2 angelegt wird. Am Ende des momentanen Befehlszyklus vergleicht der 68000 die codierte Prioritätsstufe (1 bis 7, wobei 7 die höchste Priorität ist) mit der 3-Bit-Interrupt-Maske des Statusregisters. Diese Maske kann in Bild 1.3 und dem Text in Kapitel 1 («Elektroniker» 16/82, Seite EL 5) nachgesehen und -gelesen werden.
Wenn der Wert der Prioritätsstufe gleich oder kleiner als der Wert der Interrupt-Maske ist, wird der 68000 die Anforderung einfach übersehen und am Begonnenen weiterfahren. Hat aber die Interrupt-Anforderung einen höheren Wert als die Maske, setzt der 68000 die Eingabeprioritätsstufe auf den Adressbus (A1, A2 und A3), gibt mit Hilfe der Funktionsstatusbit FC0 ... FC2 die Interrupt-Anerkennung aus und initialisiert den Interrupt-Quittungsablauf. Einzelheiten dieses Ablaufs werden im Kapitel 7 folgen.

7. Verarbeitungszustände, privilegierte Zustände und Ausnahmebetrieb

Dieses Kapitel beschreibt die Verarbeitungszustände und die privilegierten Zustände des MC 68000 und erläutert dann, wie Unterbrüche, Traps und andere «Ausnahmen» durch den MC 68000 behandelt werden.

7.1 Verarbeitungszustände

Der Mikroprozessor MC 68000 befindet sich immer in einem von drei Zuständen: Normal-, Ausnahme- oder Haltezustand. Bis jetzt betrafen unsere Betrachtungen immer den *Normalzustand,* in welchem der 68000 die Befehle vom Speicher holt, sie ausführt und die Resultate im Speicher oder in einem Register ablegt. Ein Spezialfall des Normalzustandes ist der gestoppte Zustand, in den der 68000 als Antwort auf einen STOP-Befehl eintritt. Wie in Kapitel 3 erklärt wurde, ist STOP ein privilegierter Befehl, der den 68000 stoppt, bis er einen Unterbruch hoher Priorität oder eine externe Rücksetzung erhält.
Der *Ausnahmezustand* ist die Art, mit welcher der 68000 auf Abweichungen der normalen Programmausführung reagiert. Solche Abweichungen oder Ausnahmen können durch Unterbrüche, Trap-Befehle, nicht schwerwiegende Hardwarefehler oder eine Vielzahl anderer Umstände verursacht werden, und zwar sowohl von ausserhalb als auch innerhalb des Mikroprozessors. Wir werden diese Ausnahmen und ihre Behandlung in diesem Kapitel später detailliert besprechen.
Bei schwerwiegenden Hardwarefehlern, wie zum Beispiel zwei aufeinanderfolgenden Busfehlern, tritt der 68000 in den *Haltezustand.* Aus diesem Haltezustand kann der 68000 nur mit einer externen Rücksetzung neu gestartet werden. Der Haltezustand darf nicht mit dem vorher erwähnten softwareverursachten Stoppzustand verwechselt werden.

7.2 Privilegierte Zustände

7.2.1 Überwachungs- und Benützerzustand

Bisher haben wir schon öfter die zwei privilegierten Zustände, in denen der MC 68000 operieren kann, erwähnt. Diese Zustände, der sogenannte *Überwachungszustand* und der *Benützerzu-*

stand, gewährleisten ein hohes Mass an Sicherheit durch gewisse zusätzliche «Privilegien» im Überwachungszustand, die im Benützerzustand nicht verfügbar sind (siehe Tabelle 7.1).

	Benützerzustand	**Überwachungszustand**
Eintrittszustand durch:	Löschung des S-Bit im Statusregister	Trap. Rücksetzung, Unterbruch, privilegierter Befehl
Funktionscode output FC2 =	0	1
Systemstapel-zeiger:	Benützerstapel-zeiger	Überwachungsstapel-zeiger
Andere Stapel-zeiger:	Register A0...A6	Benützerstapelzeiger und Register A0...A6
Statusregister-zugriff:		
(lesen)	Gesamtes Status-register	Gesamtes Statusregister
(schreiben)	Nur Bedingungs-codes	Gesamtes Statusregister

Verfügbare Befehle:	Alle, mit Ausnahme von:
	RESET
	RTE
	STOP #d
	ANDI.W#d,SR
	EORI.W #d,SR
	ORI.W #d,SR
	MOVE <ea>,SR
	MOVE USP,An
	MOVE An,USP
		Alle, inklusive derjenigen der links aufgelisteten

Tabelle 7.1
Privilegien des Benützer- und Überwachungszustandes des MC 68 000

Programme, die im weniger privilegierten Benützerstatus arbeiten, können alle 68000-Befehle ausführen, mit Ausnahme derjenigen, welche die höheren acht Bit des Statusregisters ändern (das «Systembyte»), den Prozessor stoppen oder eine Systemrücksetzung aussenden. Im weiteren können Benützerstatus-Programme Stapeloperationen ausführen, aber sie können die Systemstapel*zeiger* weder lesen noch schreiben.
Programme, die im höher privilegierten Überwachungszustand arbeiten, haben Zugang zu den vollen Möglichkeiten des 68000. Das heisst, dass Überwachungsstatusprogramme Zugang haben auf die beiden Systemstapelzeiger und, sofern erforderlich, über die privilegierten Befehle auch das Statusregister beeinflussen können. Die Kontrolle über das Statusregister

erlaubt den Überwachungsprogrammen das Ändern von Unterbruchsmasken und den Übergang in den Trace-Modus.
In den meisten Systemen laufen Programme, die nicht für Systemsteueraufgaben zuständig sind im Benützerstatus. Grundaufgaben des Betriebssystems sollten ausgeführt werden, wenn sich der 68000 im Überwachungsstatus befindet.

7.2.2 Wechsel im privilegierten Status

Der privilegierte Zustand wird ausgewählt mit dem Überwachungsbit (S) im Statusregister. Der 68000 ist im Überwachungsstatus mit S = 1 und im Benützerstatus, wenn S = 0 ist.
Der Übergang von einem privilegierten Zustand in einen andern kann auf verschiedene Weise erfolgen. Der Prozessor geht vom *Überwachungszustand in den Benützerstatus,* wenn das S-Bit auf 0 gesetzt wird. Diese Operation kann ausgeführt werden mit den Befehlen MOVE, ANDI oder EORI, die das Statusregister (SR) als Ziel haben und eine 0 im Bit 13 des Quellenoperanden aufweisen.

Hier einige Beispiele:

	Befehl	**Ausgeführte Aktion**
MOVE	#$0400,SR	Trace ausschalten; Wechsel zum Benützerstatus; Laden der Unterbruchsmaske mit 100_2; Löschen der Bedingungsbit.
ANDI	#$DFFD,SR	Löschen Überlauf (V); Wechsel zum Benützerstatus; keine anderen Wechsel.
EORI	#$2000,SR	Wechsel zum Benützerstatus; keine anderen Wechsel.

Der Prozessor wechselt auch zum Benützerzustand zurück nach der Rückkehr aus einem Ausnahmezustand (ausgeführt mit einem RTE-Befehl), wenn die Ausnahme im Benützerzustand auftrat. Die Behandlung der Ausnahmen folgt später in diesem Kapitel.
Der Prozessor geht vom *Benützerstatus in den Überwachungsstatus,* wenn das S-Bit auf 1 gesetzt wird. Normalerweise geschieht dies unter Softwaresteuerung mit einem der Trap-Befehle, aber es kann auch bei einem Busfehler, einem Unterbruch, erzwungener Ausführung eines privilegierten Befehls oder jeder anderen Ausnahme geschehen. Bild 7.1 zeigt eine vereinfachte Zusammenfassung der Bedingungen, die den Wechsel zum privilegierten Zustand verursachen.

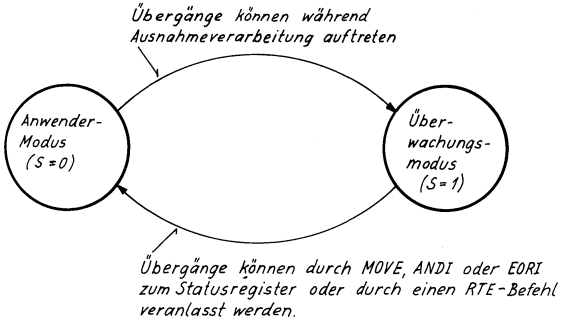


Bild 7.1
Übergänge von einem privilegierten Zustand in einen anderen

7.3 Ausnahmezustände

Wie am Anfang des Kapitels erwähnt wurde, ist eine Ausnahme eine *Abweichung von der normalen Ausführung,* abhängig von einer internen oder externen Bedingung, *die den Prozessor in den Überwachungszustand bringt.* Diese Ausnahmen (in Tabelle 7.2 zusammengestellt) werden kurz beschrieben; es ist aber sinnvoll, zunächst die Art und Weise zu erläutern, wie der 68000 Ausnahmen behandelt.

Quelle	Ausnahmeart	Veranlasst durch
Intern	Befehl	TRAP, TRAPV, CHK, DIVS, DIVU
	Privilegverletzung	Privilegierter Befehl im Benützerzustand
	Trace	Trace-Modus
	Illegale Adresse	Ungerade Adresse mit Wort oder Doppelwort
	Illegaler Befehl	Ungültiges Bitmuster
	Nichtimplementierter Befehl	Operationswort Bitmuster 1010 oder 1111
Extern	Rücksetzung	RESET aktiviert
	Unterbrüche (Interrupts)	Unterbruch genügend hoher Priorität
	Busfehler	BERR aktiviert
	Falscher Unterbruch	BERR aktiviert während Unterbruchquittung

Tabelle 7.2
Zusammenstellung interner und externer Ausnahmen

7.3.1 Verarbeitung der Ausnahmen

Bis auf die Rücksetzung wird jeder Ausnahmezustand, sei er durch ein internes Ereignis (ein Trap-Befehl zum Beispiel) oder

ein externes Ereignis (ein Unterbruch oder ein Hardwarefehler) hervorgerufen, den 68000 veranlassen, fünf eindeutige Schritte zu machen. Diese fünf Schritte sind:

1. Nach Eintritt in den Ausnahmezustand *rettet* der 68000 den 16-Bit-Inhalt des *Statusregisters* in ein nicht adressierbares, internes Register.

2. Das *Überwachungsbit (S)* im Statusregister wird auf 1 gesetzt und bringt den Mikroprozessor in den Überwachungsstatus; das *Trace-Bit (T)* wird auf 0 gesetzt und damit der Trace-Modus ausgeschaltet. Wenn der Ausnahmezustand auf einen Unterbruch zurückzuführen ist, wird die *Unterbruchsmaske* mit der entsprechenden Prioritätsebene nachgeführt, um Unterbrüche auf dieser oder tieferen Prioritätsebenen auszuschliessen, bis der behandelte Unterbruch ausgeführt ist.

3. Der 68000 bestimmt die *Vektornummer* der Ausnahme und multipliziert diese mit 4, um sie in eine *Vektoradresse* umzuwandeln. Der 68000 kann 255 verschiedene Vektornummern unterscheiden, 0 und 2 bis $FF. Bild 7.2 enthält die Vektornummer und die Vektoradresse für jede Ausnahmebedingung. Bei Unterbrüchen wird die Vektornummer durch das externe Gerät geliefert. Für alle andern Ausnahmen wird die Vektornummer intern berechnet, unter Zuhilfenahme des Mikrocodes des 68000.

4. Der aktuelle Wert des *Programmzählers* und die intern gesicherte Kopie des Statusregisters werden in den Überwachungsstapel geschrieben. In den meisten Fällen ist der Inhalt des Programmzählers die Adresse des nächsten auszuführenden Befehls.

5. Nach dem Retten dieser Informationen lädt der 68000 den Programmzähler mit dem Inhalt der berechneten Vektoradresse und beginnt, die *Serviceroutine* des Ausnahmezustandes *auszuführen.*

Eine Spezialbedingung, der *doppelte Busfehler,* soll hier noch erwähnt werden. Ein doppelter Busfehler stellt einen schwerwiegenden Fehler im System dar und erscheint, wenn ein Busfehler oder eine illegale Adressausnahme erzeugt wird, während eine Ausnahme in der oben erwähnten Gruppe 0 (Rücksetzung, Busfehler oder illegale Adresse) in Bearbeitung ist. Nach dem Erhalt zweier solcher aufeinanderfolgender Fehler bringt sich der 68000 selbst in den Haltezustand. Einmal im Haltezustand, kann der Mikroprozessor MC 68000 nur durch eine externe Rücksetzung wiedergestartet werden.

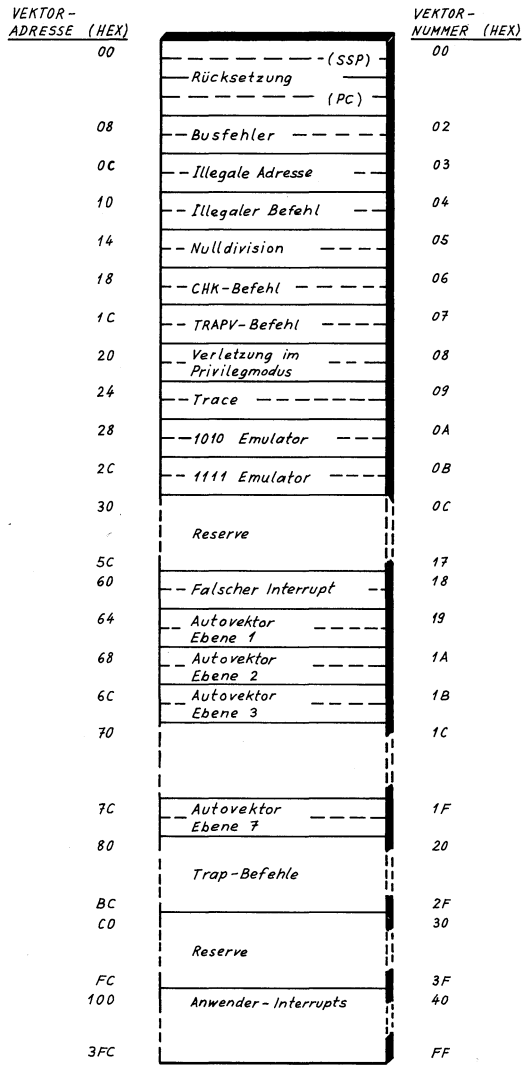


Bild 7.2
Adressenzuordnung für Ausnahmebedingungen

Bild 7.3 zeigt einen Ablaufplan der eben genannten Sequenz. Die Details dieser Ausnahme-Serviceroutine sind selbstverständlich abhängig davon, welche Ausnahme bearbeitet werden muss. Jede Serviceroutine muss grundsätzlich beendet werden mit dem Befehl *Rückkehr aus dem Ausnahmezustand (RTE)*, der den Statusregister- und Programmzählerwert aus dem Überwachungsstapel zurückschreibt und die Rückkehr zur normalen Ausführung der Befehle sicherstellt.

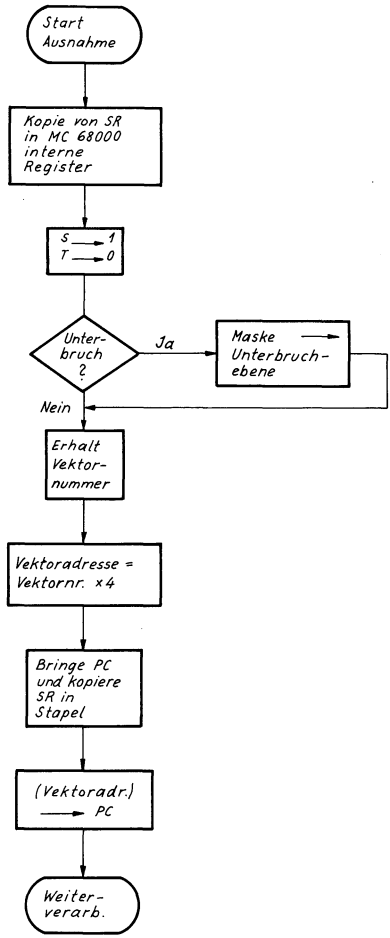


Bild 7.3
Allgemeiner Ablauf für die Verarbeitung von Ausnahmen (ausgenommen Rücksetzung)

7.3.2 Mehrfachausnahmen

Wie reagiert der 68000, wenn zwei oder mehrere Ausnahmebedingungen *gleichzeitig* auftreten? Was passiert zum Beispiel, wenn ein Unterbruch erscheint, währenddem eine Trace-Ausnahmebedingung in Ausführung begriffen ist? Die Antwort auf diese Frage wird in Tabelle 7.3 gegeben, welche die Ausnahmearten mit abnehmender Priorität darstellt. Das bedeutet, dass die Bedingungen in Gruppe 0 vor jenen in den Gruppen 1 und 2 ausgeführt werden. Dadurch wird, wenn ein Busfehler während einer Trace-Operation erscheint, die Trace-Operation zurückgestellt (an das Ende des laufenden Taktzyklus), bis die Busfehlerbearbeitung beendet ist.
Die Bedingungen innerhalb jeder Gruppe in Tabelle 7.3 sind

ebenfalls in absteigender Prioritätsordnung aufgelistet. Dadurch wird zum Beispiel bei einem Unterbruch während einer Trace-Ausnahme diese weiterverarbeitet, dann erst wird der Unterbruch verarbeitet, und schliesslich wird der 68000 mit der Ausführung der Befehle im Programm weiterfahren.

Gruppe	Ausnahme	Ausnahmeverarbeitung beginnt:
0	Rücksetzung Busfehler Illegale Adresse	Am Ende eines Taktzyklus
1	Trace, Unterbruch, Illegaler Befehl.	Am Ende eines Befehlszyklus
	Nichtvorhandener Befehl, Privilegverletzung.	Am Ende eines Buszyklus
2	TRAP, TRAPV, CHK, Division durch 0	Am Ende eines Befehlszyklus

Tabelle 7.3
Ausnahmegruppierung und Priorität

7.4 Intern erzeugte Ausnahmen

Wir beschreiben nun alle Ausnahmen, erzeugt durch *interne* Bedingungen im 68000.

7.4.1 Befehle, die Ausnahmen herbeiführen können

In der Diskussion des Befehlssatzes im Kapitel 3 begegneten uns einige Befehle, die Ausnahmezustände veranlassen können. Einer dieser Befehle (TRAP) verursacht immer einen Ausnahmezustand; die andern (TRAPV, CHK, DIVS und DIVU) verursachen je nach gewissen Bedingungen eine Ausnahme oder nicht.
«Trap» (TRAP = Falle) erzwingt eine Ausnahme zu einer von sechzehn benützerdefinierten Trap-Routinen, angewählt durch den unmittelbaren Operanden im Befehl. Im speziellen veranlassen die Befehle TRAP # 0 bis TRAP # 15 die Programmsteuerung zu einem unbedingten Sprung zu den Routinen, deren Adressen in den Doppelwortspeicherplätzen $80 bis $BC enthalten sind. Tabelle 7.4 zeigt die Zuordnung für die 16 möglichen Trapbefehle.

Befehl	Transferiert Programmsteuerung über Vektoradresse:
TRAP #0	$80
TRAP #1	$84
TRAP #2	$88
TRAP #3	$8C
TRAP #4	$90
TRAP #5	$94
TRAP #6	$98
TRAP #7	$9C
TRAP #8	$A0
TRAP #9	$A4
TRAP #10	$A8
TRAP #11	$AC
TRAP #12	$B0
TRAP #13	$B4
TRAP #14	$B8
TRAP #15	$BC

Tabelle 7.4
Vektoradresse für TRAP

Die Trapbefehle wirken als eine Anzahl von Softwareunterbrüchen und sind praktisch für den Aufruf des Betriebssystems, die Simulation von Unterbrüchen bei einer Fehlersuche, die Anzeige der Beendigung von Prozessen oder die Anzeige, dass eine Fehlerbedingung in einem Programm entstanden ist.
«Trap bei Überlauf» (TRAPV) wird einen Trap veranlassen durch Vektoradresse $1C, wenn der Überlauf (V) im Bedingungscoderegister auf 1 gesetzt ist. Eine einfache Routine auf der Betriebssystemebene kann dann jeden entstehenden Überlauf behandeln.
«Prüfe Register auf Grenzen» (CHK) bestimmt, ob sich das tiefere Wort eines spezifizierten Datenregisters innerhalb der Grenzen 0 und einer spezifizierten Zweierkomplementzahl als obere Limite (im Speicher oder in einem anderen Datenregister) befindet. Ist der Registerinhalt ausserhalb dieser Grenzen, erzeugt der 68000 einen Trap über die Vektoradresse $18. Der CHK-Befehl kann verwendet werden zur Überprüfung, dass ein Stapel nicht zu gross wird, dass eine Folge von Zeichen den ihr zugewiesenen Raum nicht überschreitet, dass ein Array sich in den dimensionierten Grössen bewegt oder dass eine bestimmte Operation nicht zu Daten ausserhalb eines zugewiesenen Speicherbereichs zugreift.
Die Befehle *«dividiere mit Vorzeichen»* (DIVS) und *«dividiere ohne Vorzeichen» (DIVU)* erzeugen nur auf eine Bedingung eine Ausnahme, dann wenn der Divisor 0 ist. Ein Divisor = 0 veranlasst einen Trap über die Vektoradresse $14.
Wie in Kapitel 3 erwähnt wurde, ist der Versuch, durch 0 zu dividieren, eine von zwei Bedingungen, die verhindern, dass die

Divisionsoperation ausgeführt wird. Die Operation wird ebenfalls gestoppt, wenn ein Überlauf während der Division auftritt (in beiden Fällen bleiben Divisor und Dividend unbeeinflusst). Wenn das passiert, setzt der 68000 einfach das V-Bit im Statusregister und fährt mit der Ausführung des nächsten Befehls weiter.
Weil Überlauf eine Fehlerbedingung darstellt, sind in einem Divisionsprogramm Massnahmen für die Weiterbehandlung zu treffen. Eine Möglichkeit besteht darin, die Divisionsroutine so zu entwerfen, dass ein gültiger Quotient entsteht, unabhängig davon, ob Überlauf entsteht oder nicht. Diese Möglichkeit ist in Kapitel 4 im Beispiel 4.4 dargestellt. Es ist auch möglich, die Überwachung bei Überlauf zu wählen, indem dem DIVS- oder DIVU-Befehl ein TRAPV-Befehl nachgestellt wird.

7.4.2 Verletzung privilegierter Befehle

Der 68000 initialisiert einen Ausnahmebetriebszustand über die Vektoradresse $20, wenn ein Benützerprogramm versucht, einen der privilegierten Befehle auszuführen. Die privilegierten Befehle sind im Kapitel 3 beschrieben (Tabelle 3.17 und begleitender Text) und sind aufgelistet in der «Benützerstatus»-Kolonne der Tabelle 7.1.

7.4.3 Tracing

Wie die Haltefunktion ist auch die Trace-Funktion vorgesehen zur Unterstützung der Programmentwicklung und Fehlerbehebung. Wenn die Trace-Funktion eingeschaltet ist (T = 1 im Statusregister), erzeugt der 68000 nach jedem ausgeführten Befehl einen Ausnahmezustand und veranlasst dadurch den Prozessor, «einzelschrittweise» durch ein Programm zu gehen. Der Trace-Ausnahmebetrieb veranlasst die Programmsteuerung, über die Vektoradresse $24 zu einer benützerdefinierten Routine im Speicher zu wechseln. Wie alle Ausnahmezustände veranlasst auch der Trace-Ausnahmezustand das Rücksetzen des Trace-Bit (T = 0) und das Speichern der gegenwärtigen Inhalte des Programmzählers und des Statusregisters in den Überwachungsstapel. Nach der Rückkehr aus diesem Ausnahmezustand besteht der Trace-Modus weiter, wenn nicht diese Trace-Routine das im Stapel gesicherte T-Bit des Statusregisters löscht. Das T-Bit kann gelöscht werden, indem dem RTE-Befehl ein ANDI #$7FFF, (SP) vorangestellt wird.
Die Trace-Routine wird normalerweise verwendet, um einen Ausdruck der Registerinhalte nach jedem Befehl zu erhalten. Abhängig von der Programmierart kann die Trace-Routine auch andere wichtige Parameter, wie zum Beispiel die Ausführungszeit jedes Befehls, ausdrucken.

Der Trace-Modus stellt auch einen einfachen Weg dar, in einem System «breakpoints» einzuführen. Dies kann gemacht werden, indem die im Stapel geretteten Adressen (durch den Trace-Ausnahmezustand) mit einer Breakpoint-Adressentabelle verglichen werden. Wenn die Adressen gleich sind, kann der Inhalt der Register angezeigt oder ausgedruckt werden. Andernfalls würde der 68000 einfach aus dem Trace-Modus zurückkehren und den nächsten Befehl des Programms ausführen.

7.4.4 Illegale Adressen

Eine illegale Adresse ist eine ungerade numerierte Adresse, die sich auf einen Wort- oder Doppelwortoperanden bezieht. Sie wird behandelt über die Vektoradresse $0C. Eine illegale Adresse kann bei jeder Art von speicherbezogenen Operationen erscheinen, tritt aber vor allem bei der Verwendung der komplexeren Adressierungsarten auf, wie zum Beispiel Adressregister indirekt mit Index, in dem verschiedene Komponenten addiert werden, um die effektive Adresse zu erhalten.

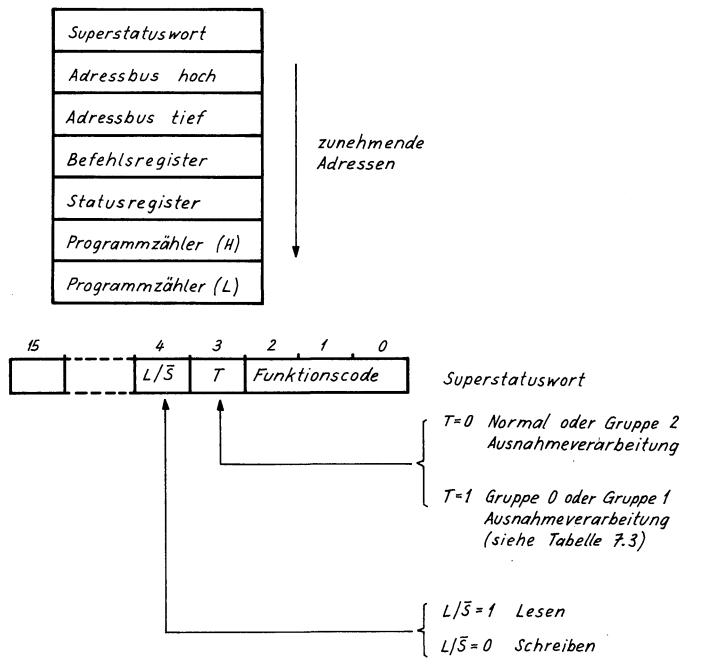


Bild 7.4
Speicherung bei illegalen Adressen und Busfehlern

Bei Ausnahmezustand «illegale Adresse» (und auch einer extern erzeugten Ausnahme, Busfehler) bringt der 68000 sieben Worte Kontext-Information in den Überwachungsstapel. Diese Worte sind in Bild 7.4 dargestellt. Wie zu sehen ist, sind die

ersten drei Worte der Programmzähler und das Statusregister, gefolgt vom Befehlsregister (dem Operationswort des Befehls, der die illegale Adresse erzeugt hat), der illegalen Adresse selbst und einem «Superstatuswort». Das Superstatuswort vermittelt spezifische Information über den versuchten Speicherzugriff, ob Schreib- oder Lesezugriff, ob der 68000 an einer Befehlsausführung war (Normalzustand oder Ausführung eines Gruppe-2-Befehls) oder in der Ausführung einer Gruppe-0-oder Gruppe-1-Ausnahme und dem Status des Funktionscodes beim Auftreten des illegalen Zugriffs.
Die Ausnahmegruppen sind in Tabelle 7.3 zusammengestellt. Wie früher erwähnt, wird der 68000 in einen Doppelbusfehlerfall eintreten, wenn einer der Befehle im Ausnahmezustand «illegale Adresse» selbst eine illegale Adresse erzeugt; der Prozessor geht damit in den Haltezustand. Ein extern erzeugter Busfehler (später in diesem Kapitel beschrieben), während der Bearbeitung des Ausnahmezustandes «illegale Adresse», wird ebenfalls einen Doppelbusfehler veranlassen.
Was passiert, wenn eine ungerade Adresse unabsichtlich in den Vektorspeicherbereichen der illegalen Adressen ($0C...$0F) gespeichert wird? Wenn diese unwahrscheinliche und unglückliche Situation auftritt und der 68000 zufällig einen Wort- oder Doppelwortspeicherzugriff an einer ungeraden Adresse versucht, werden die folgenden Ereignisse stattfinden:

1. Nach der Feststellung der illegalen Adresse wird der 68000 den Ausnahmeverarbeitungszustand «illegale Adresse» initialisieren. Nach dem Übergang in den Überwachungszustand (S = 1), Ausschalten des Trace-Modus (T = 0) und Berechnen der Vektoradresse werden sieben Worte in den Stapel geschrieben (Bild 7.4) und der Inhalt der durch die Vektoradresse bestimmten Speicherzelle in den Programmzähler geladen.

2. An diesem Punkt würde der 68000 normalerweise mit der Ausführung der Befehle der Ausnahmeroutine «illegale Adresse» beginnen. In diesem Fall hat aber der Programmzähler eine ungerade Adresse erhalten. Weil diese Befehlsadresse ungerade ist, ist sie illegal, und der 68000 versucht erneut, den Ausnahmebetriebszustand «illegale Adresse» zu initialisieren. Das bedeutet, dass der 68000 zum Schritt 1 zurückkehrt.

3. Wird diese zweite aufeinanderfolgende illegale Adresse eine Doppelbusfehlerbedingung verursachen? Nein, weil die illegale Adresse während der Initialisierungssequenz erscheint und nicht in der Ausführung der Serviceroutine. Der 68000 wird aber wiederholt Ausnahmebetriebszustände «illegale Adresse» initialisieren und jedesmal sieben Worte in den Stapel schreiben.

4. Weil die Stapel im Speicher rückwärts aufgebaut werden, kann irgendeines der folgenden Ereignisse diese wiederholte Sequenz schliesslich beenden:
- Der 68000 gerät ausserhalb des Lese/Schreib-Speichers (RAM) und versucht, Information in nicht existierende Speicher oder Nur-lese-Speicher (ROM) zu bringen. Dies sollte externe Schaltungen veranlassen, einen Busfehler-Ausnahmezustand zu initialisieren.
- Der 68000 kann versuchen, Information in den Programmspeicher zu schreiben anstatt in den Datenspeicher, was ebenfalls einen Busfehler-Ausnahmezustand herbeiführen sollte.
- Wenn der Stapel bei der Verarbeitung bis zu den tiefsten 1024 Byte im Speicher vordringt, werden neue Werte eventuell in den Vektorzeiger «illegale Adresse» (Speicherzellen $0C...$0F) gespeichert. Wenn diese neue Adresse ungerade ist, wird mit der vorhergehenden Sequenz fortgefahren. Ist sie gerade, wird der Programmzähler versuchen, an dieser neuen, zufälligen Adresse einen Befehl auszuführen, mit undefinierten Resultaten.
- Wenn der Stapel versucht, in den Rücksetzvektor im untersten Teil des Speichers zu schreiben, sollte ein Busfehler erzeugt werden, weil diese Speicherzelle einzig ein Nur-lese-Speicher (ROM) sein kann.

7.4.5 Illegaler Befehl

Ein illegaler Befehl ist ein 16-Bit-Binärmuster, das nicht eines der legalen Operationsworte des Befehlsrepertoires des 68000 darstellt. Es ist unnötig zu sagen, dass ein guter Assembler kein illegales Bitmuster erzeugen wird. Hingegen können durch Programmierer solche Muster durch Manipulationen im Objektcode erzeugt werden.

7.4.6 Nichtimplementierte Befehle

Die Entwurfsspezifikation für den MC 68000 enthält verschiedene Befehle, die in den gegenwärtigen Produktversionen nicht implementiert sind. Das sind zum Beispiel Befehle für Stringmanipulation, Feldmanipulation, Codeumsetzung, Fliesskomma-Arithmetik, Doppelwort-Multiplikation und spezielle Divisionsalgorithmen. Motorola reservierte ungefähr 20% des totalen Platzes für Mikrocode, um in zukünftigen Versionen diese (oder vielleicht auch andere) Erweiterungen einzuführen.
Der nicht benützte Platz im Microcode schliesst zwei von 16 möglichen «Operationscodes» (die vier höherwertigen Bit eines Befehls) ein. Anstatt diese zwei nicht implementierten Operationscodes, binär 1010 und 1111, intern unbenützt zu lassen, stellt Motorola eine spezielle Vektornummer im Ausnahmespei-

cherbereich für jeden dieser Operationscodes zur Verfügung. Das gibt Benützern die Möglichkeit, in ihren Programmen Emulationsbefehle einzufügen. Diese Befehle können entweder für die zukünftigen Erweiterungen zum MC 68000 vorgesehen werden (zum Beispiel String- oder Fliesskomma-Befehle) oder auch nur verschiedene praktische Funktionen für Benützeranwendungen zur Verfügung stellen.
Wie muss bei der Benützung dieser zwei nicht implementierten Operationscodes vorgegangen werden? Es ist ganz einfach: zur Benützung einer dieser Operationscodes muss einfach nur ein Wort im Programm eingefügt werden, dessen höchstwertige 4 Bit den Wert $A (1010 binär) oder $F (1111 binär) aufweisen. Am einfachsten wird die Einfügung gemacht mit einer Konstantenanweisung (define constant) wie zum Beispiel DC $A000 oder DC $F000. Wenn der 68000 einem Befehloperationswert begegnet, das mit $A oder $F beginnt, wird er es als nicht implementierten Befehl einstufen und in eine Serviceroutine über Adresse $28 (für 1010) oder $2C (für 1111) eintreten.
Als Beispiel für nicht implementierte Befehle wollen wir einen Satz von *Fliesskomma-Befehlen* emulieren, unter Benützung des Operationscodes 1010. Angenommen wir haben vier verschiedene Fliesskomma-Befehle: addieren, subtrahieren, multiplizieren und dividieren. Im weitern nehmen wir an, dass jeder dieser Befehle mit zwei Datenregistern operiert, einem Quellenregister und einem Zielregister. Bild 7.5 zeigt das Bitformat für die Fliesskomma-Operationsworte. Aus dieser Darstellung lässt sich erkennen, dass der zu emulierende Befehl eine Fliesskomma-Multiplikation von D4 mal D5 mit Speicherung des Produktes in D5 ist. Der Weg zur Einfügung dieses Befehls in einem Programm führt über die Anweisung DC $AA14.

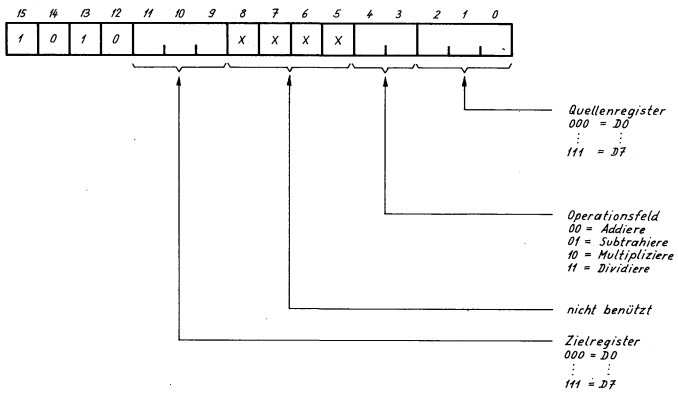


Bild 7.5
Bitformat der Fliesskomma-Befehle

Wie sieht diese Fliesskomma-Serviceroutine aus? Ein Teil dieser Routine, die Befehlsdecodierungssequenz, wird im Programmbeispiel 7.1 gezeigt. Der Routine (FLTP) sind zwei

Programmbeispiel 7.1:
Routine zur Initialisierung von Fliesskomma-Mathematik

```
*Diese Ausnahmeroutine wird ausgeführt, wenn der MC 68000 im Programm auf
*einen 1010-Befehl trifft. Sie decodiert das Operationsfeld des Befehls (Bit 3 und 4)
*und benützt diese Zahl als Index zum Sprung auf eine Fliesskomma-Additions-,
*-Subtraktions-, -Multiplikations- oder -Divisionsroutine irgendwo im Speicher. Die
*Register A1 und D1 werden benützt.
*
*Initialisiere 1010 Vektor
*
          ORG       $28
          DC.L      FLTP              1010-Vektor zeigt auf FLTP.
*
          ORG       $1000
FLTP      MOVEA.L   2(SP),A1          Programmzähleradresse nach
                                      1010.
          MOVE      -2(A1),D1         Bringe 1010-Befehl in D1;
          MOVE      D1,-(SP)          Kopie im Stapel.
          ANDI#     #0018,D1          Erhalte Operationsfeld
                                      (Bit 3, 4).
          LSR       #1,D1             Berechne Index (Operations-
                                      feld x 4).
          LEA       OPADDR,A1         Hole Adresse der Operations-
                                      tabelle.
          MOVEA.L   0(A1,D1.W),A1     Hole Adresse der gewünschten
                                      Routine
          JMP       (A1)              und springe zu
OPADDR    DC.L      FLTPADD,FLTPSUB,FLTPMUL,FLTPDIV
          END
```

Anweisungen vorangestellt, welche die 1010-Vektoradresse mit der Adresse von FLTP initialisieren. Zur Decodierung der korrekten Operation (addieren, subtrahieren, mulitplizieren oder dividieren), muss der Originalbefehl geholt und in ein Register geschrieben werden, damit die Bitnummern 3 und 4 manipuliert und abgefragt werden können. Der im Stapel gesicherte Programmzählerwert kann, um zwei vermindert, zum Wiederauffinden des 1010-Befehls durch Zugriff auf diesen Speicherplatz verwendet werden.

Wenn das Operationswort in D1 gespeichert ist, wird eine Kopie davon im Stapel gesichert für die spätere Registerdecodierung durch die Additions-, Subtraktions-, Multiplikations- oder Divisionsroutine. Wenn das getan ist, maskiert ein ANDI-Befehl das Operationsfeld heraus (Bit 3 und 4) und eine Rechtsschiebung um ein Bit wandelt es in einen OPADDR-Tabellenindex um (entspricht einer Multiplikation mit 4). Was übrigbleibt,

ist das Holen der Adresse der Operationsroutine (FLTPADD, FLTPSUB, FLTPMUL, oder FLTPDIV) in A1 und der Sprung zu dieser Routine. Die Adresse wird geholt mit einem MOVEA-Befehl unter Benützung der Adressierungsart «Programmzähler relativ mit Index». Es ist zu bemerken, dass diese FLTP-Routine ganz ähnlich ist zu der Subroutine «Auswahl von Mehrfachbenützern», SELUSR, im Programmbeispiel 5.10, da beide einen Eingangscode benützen zur Ableitung eines Index in eine Konversionstabelle. Der Hauptunterschied besteht darin, dass SELUSR herausfinden muss, ob der Identifikationscode gültig ist, während FLTP keine solche Prüfung durchführen muss, weil sie ein 2-Bit-Feld zur Auswahl von einer aus vier mathematischen Routinen decodiert. Wäre das Feld in FLTP drei Bit lang und nur fünf der acht möglichen Kombinationen wären gültig, müsste ebenfalls eine Gültigkeitsprüfung durchgeführt werden.

7.5 Extern erzeugte Ausnahmen

Nach Abschluss der Diskussion über intern erzeugte Ausnahmen werden nun Bedingungen *ausserhalb* des MC 68000 dargestellt, die einen Ausnahmezustand initialisieren. Es gibt 3 solche Bedingungen: Rücksetzung, Unterbrüche und Busfehler.

7.5.1 Rücksetzung (RESET)

Der *$\overline{\text{RESET}}$-Eingang* hat die höchste Priorität aller Ausnahmen (siehe Tabelle 7.3) und ist bestimmt für die Systeminitialisierung und den Wiederanlauf nach schwerwiegenden Fehlern wie zum Beispiel Spannungsausfall. Im wesentlichen besteht die $\overline{\text{RESET}}$-Funktion darin, dem 68000 mitzuteilen, dass alle in Ausführung stehenden Prozesse bedeutungslos sind und zu beenden sind. Nach Empfang des $\overline{\text{RESET}}$-Signals fällt der 68000 in den Überwachungsmodus (S = 1), schaltet den Trace-Modus aus (T = 0) und setzt die Unterbruchsmaske auf die höchste Ebene (Level 7), so dass kein Unterbruch diese $\overline{\text{RESET}}$-Prozedur unterbrechen kann. Anders als andere Ausnahmebedingungen schützt eine Rücksetzung weder den Programmzähler noch das Statusregister. Der Vektor der Rücksetzbedingung ist vier Worte lang und belegt die Adressen $00 ... $07; diese Adressen *müssen* im Nur-lese-Speicher (ROM) liegen. Während des Rücksetzprozesses holt der 68000 die beiden ersten Worte in den Systemstapelzeiger und die zweiten zwei Worte in den Programmzähler und beginnt dann die Ausführung der Befehle, die durch den Programmzähler adressiert werden. An dieser Stelle befindet sich die Kaltstart-/Warmstart-Routine (Stromversorgung ein/Wiederanlauf).

Bild 7.6 ist ein Flussdiagramm der Ausnahmeverarbeitung für die Rücksetzbedingung. Zu sehen ist auch, dass eine Massnahme gegen einen doppelten Busfehler getroffen wird, falls ein Bus- oder Adressfehler während der Rücksetzbedingung auftritt.

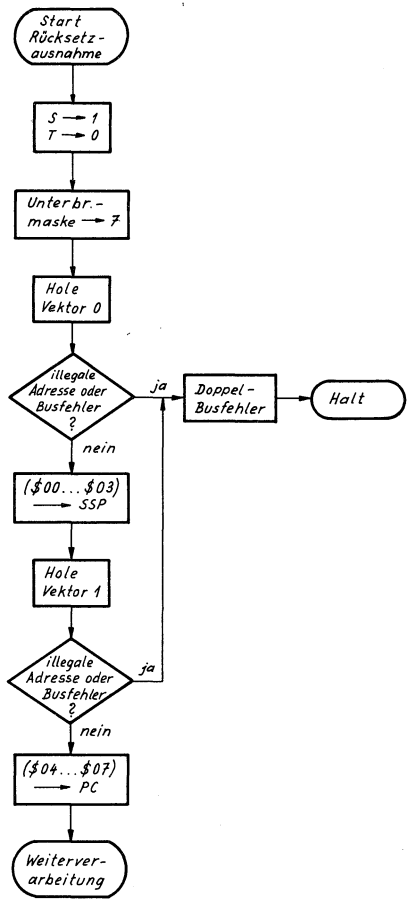


Bild 7.6
Ablauf für die Ausnahmeverarbeitung beim Rücksetzen

7.5.2 Unterbrüche (Interrupts)

Leser mit Erfahrung in der Programmierung von Unterbruchs-Abfragesequenzen für frühere 8-Bit-Mikroprozessoren werden gerne vernehmen, dass der MC 68000 über eine Minicomputer-ähnliche *Unterbruchsstruktur mit Prioritätsordnung* verfügt, die Unterbruchsanforderungen sieben verschiedener Ebenen akzeptieren kann. Im weiteren können diese Unterbrüche mit oder ohne Vektor behandelt werden.

Unterbruchsprioritäten gehen von der Ebene 1 (tiefste Priorität) bis Ebene 7 (höchste Priorität, nicht maskierbar). Wenn ein externer Baustein den 68000 zu unterbrechen wünscht, decodiert er die Unterbruchsebene der Unterbruchsanforderung auf den drei Unterbruchskontrolleitungen IPL0, IPL1 und IPL2. Vorausgesetzt, dass nicht eine Trace-, eine illegale Adress-, Busfehler- oder Rücksetz-Ausnahmebedingung in Ausführung begriffen ist, wird der 68000 den in Ausführung begriffenen Befehl fertig bearbeiten und dann die decodierte Prioritätsebene mit einer 3-Bit-Unterbruchsmaske im Statusregister vergleichen (siehe Bild 1.3 im Kapitel 1).
Wenn der decodierte Wert auf den Unterbruchs-Kontrolleitungen gleich oder kleiner ist als der Wert der Unterbruchsmaske, wird der 68000 die Anforderung einfach ignorieren und mit der Abarbeitung der Befehle normal weiterfahren. (Die einzige Ausnahme hier ist die Ebene 7, die eine andere Unterbruchsanforderung der Ebene 7 quittieren wird.) Wenn aber die Unterbruchsanforderung einen höheren Wert als die Unterbruchsmaske aufweist, wird der 68000 eine Ausnahmeverarbeitung einleiten.
In den meisten Fällen wird die Unterbruchsbearbeitung unserer allgemeinen Ausnahme-Verarbeitungssequenz (Bild 7.3) folgen, hat aber genug zusätzliche Schritte, um ihre eigene Schritt-für-Schritt-Beschreibung zu rechtfertigen. Im folgenden sind die Schritte in der Unterbruchs-Behandlungssequenz aufgeführt und in Bild 7.7 dargestellt:

1. Nach Erhalt einer Unterbruchsanforderung genügend hoher Priorität rettet der 68000 den 16-Bit-Inhalt des Statusregisters in einem nichtadressierbaren, internen Register.
2. Der 68000 geht in den Überwachungsmodus (S = 1) und verlässt den Trace-Modus (T = 0).
3. Die Prioritätsebene des quittierten Unterbruchs (1 ... 7) wird in die Unterbruchsmaske des Statusregisters geschrieben und zu allen Bausteinen im System auf den Adressleitungen A1, A2 und A3 ausgesendet. Zur Kennzeichnung der Adressbusinformation als Unterbruchsquittung beansprucht der 68000 alle drei Funktionscodeleitungen (FC0, FC1 und FC2).
4. An diesem Punkt erwartet der 68000 die Systemantwort, entweder ein Fehlersignal ($\overline{\text{BERR}}$) oder eines von zwei Nichtfehlersignalen ($\overline{\text{VPA}}$ oder $\overline{\text{DTACK}}$). Wenn weder $\overline{\text{VPA}}$ (valid peripheral address) noch $\overline{\text{DTACK}}$ (data transfer acknowledge) in einer vorbestimmten Zeit erscheint, sollte ein externes Überwachungszeitglied einen Busfehler ($\overline{\text{BERR}}$) senden, damit der 68000 weiss, dass die Unterbruchsanforderung unecht war. Ein solcher falscher Unterbruch veranlasst den 68000 zur Erzeugung der Vektornummer $18.
5. Wenn die Unterbruchsanforderung nicht falsch war, sind die gültigen Unterbruchsquittungen $\overline{\text{VPA}}$ und $\overline{\text{DTACK}}$. Die Bedeutung dieser Antworten sind:

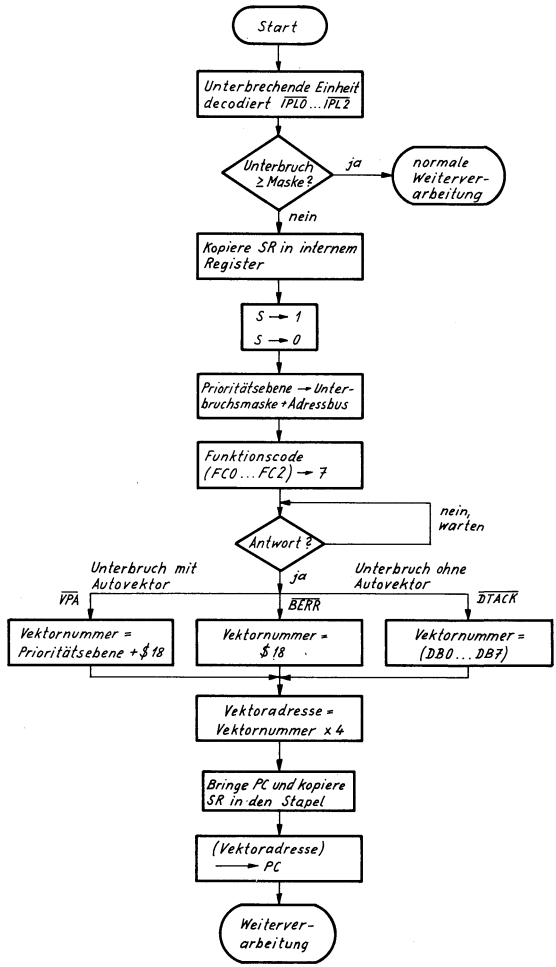


Bild 7.7
Ablauf der Unterbruchsverarbeitung (Interrupt)

- Bausteine, die speziell zur Unterstützung des 68000 entworfen wurden, antworten auf die Unterbruchsquittung durch das Plazieren einer von 192 *Benützerunterbruchs*-Vektornummern ($40 ... $FF) auf dem niedrigstwertigen Byte des Datenbus (DB0 ... DB7) und Erzeugung von $\overline{\text{DTACK}}$.
- Frühere Bausteine wie jene, welche die 6800- und 6500-Familien unterstützen, können keine Vektornummer aussenden. Diese Bausteine antworten auf die Unterbruchsquittung durch Aktivieren von $\overline{\text{VPA}}$, was den 68000 veranlasst, die Prioritätsebene zu prüfen und eine Basisadresse von $18 zu dieser Ebene zu addieren, um eine *Autovektornummer* zu erzeugen. Weil die Prioritätsebenen von 1...7 gehen, liegen die Autovektornummern im Bereich von $19 bis $1F.

6. Der 68000 multipliziert nun die Vektornummern mit 4 zur Erzeugung einer Vektoradresse. Für einen falschen Unterbruch wird die Vektoradresse $60 sein. Für Benützerunterbrüche liegen die Vektoradressen im Bereich von $100 bis $3FC. Für die Autovektoren liegen die Vektoradressen im Bereich von $64 (Ebene 1) bis $7C (Ebene 7).
7. Der laufende Programmzählerwert und die intern gesicherte Kopie des Statusregisters werden in den Überwachungsstapel geschrieben.
8. Der 68000 lädt den Programmzähler mit dem Inhalt der berechneten Vektoradresse und beginnt die Ausführung der Unterbruchserviceroutine.

7.5.3 Busfehler

Aus früheren Erklärungen wissen wir, dass das Busfehlersignal ($\overline{\text{BERR}}$) ein extern erzeugtes Eingangssignal ist, das dem 68000 einen Fehler irgendwo im System anzeigt. Wir haben die folgenden Anwendungen von $\overline{\text{BERR}}$ diskutiert:

1. Allein auftretend dient $\overline{\text{BERR}}$ zur Anzeige, dass einer von verschiedenen Fehlern im System aufgetreten ist. Zum Beispiel kann ein Überwachungszeitglied («Watchdog») $\overline{\text{BERR}}$ verwenden zur Anzeige, dass ein adressierter Speicherplatz oder eine periphere Schaltung es unterlassen hat, ein $\overline{\text{VPA}}$ oder $\overline{\text{DTACK}}$ als Antwort zum 68000 zu senden. Weiter kann ein Speicherverwaltungsbaustein $\overline{\text{BERR}}$ verwenden zur Anzeige, dass das ausführende Programm einen illegalen Speicherzugriff versucht hat (zum Beispiel den Versuch, in einen Nur-lese-Speicher zu schreiben).
2. Zusammen mit $\overline{\text{HALT}}$ verursacht $\overline{\text{BERR}}$ ein nochmaliges Ablaufen des Bus-Zyklus, mit anschliessendem Halt.
3. Das Erscheinen von $\overline{\text{BERR}}$ während einer Gruppe-0-Ausnahmeverarbeitung (Rücksetzung, illegale Adresse oder Busfehler) verursacht einen Doppelbusfehler, der den Prozessor in den Haltezustand bringt.
4. $\overline{\text{BERR}}$ während einer Unterbruchsbehandlung initialisiert den Ausnahmefall «falscher Unterbruch» über die Vektoradresse $60.

Die Bedingung 4 (Ausführung falscher Unterbrüche) veranlasst den 68000 zur Speicherung des momentanen Inhalts des Programmzählers und des Statusregisters, total drei Worte. Bedingungen 1, 2 und 3 veranlassen den 68000, sieben Worte zu sichern: Programmzähler, Statusregister, Befehlsregister, Adressbus (hoch und tief) und ein sogenanntes Superstatuswort. Diese Worte sind unter dem Thema illegale Adressverarbeitung beschrieben (Bild 7.4 und begleitender Text im Abschnitt 7.4.4).

Wie man sehen kann, wird tatsächlich nur die Bedingung 1 dazu führen, dass eine Befehlsverarbeitung stattfinden wird. Das bedeutet, dass die Busfehler-Ausnahmebedingung mit $\overline{\text{BERR}}$ eingeleitet wird, wenn der 68000 Befehle im *Normalmodus* verarbeitet oder *Ausnahmebedingungen* der *Gruppe 1* oder *Gruppe 2* verarbeitet, ausgenommen Unterbrüche. Busfehler-Ausnahmeverarbeitung veranlasst den 68000 zur internen Erzeugung einer Vektornummer von $02, und er initialisiert die Ausführung über die Vektoradresse $08.

8. Anschluss von Peripheriebausteinen

Im Kapitel 6 sind alle Signale beschrieben, die man benötigt, um periphere Bausteine richtig anzusteuern. Wir haben auch die zeitlichen Beziehungen zwischen den einzelnen Signalen untersucht und gezeigt, wie der 68 000 mit einem asynchronen 16-Bit- oder einem synchronen 8-Bit-Baustein kommuniziert. Hier folgt eine kurze Übersicht über die Elemente, die an den 68 000 anschliessbar sind. Dazu ein einfaches Anwendungsbeispiel.

8.1 Peripheriebausteine der 68 000er-Familie

Die Tabelle 8.1 zeigt die 11 ersten Peripheriebausteine für den 68 000. Sie werden alle an die asynchronen Steuerleitungen des Mikroprozessors angeschlossen.

Typennummer	Beschreibung	Hersteller	Zweitlieferant
68120/ 68121	Intelligenter Peripherie-Controller (IPC)	Motorola	Rockwell
68122	Terminal-Controller (CTC)	Motorola	–
68230	Parallel-Schnittstelle/Timer (PIT)	Motorola	–
68340	Doppelschnittstellen mit RAM (DPR)	Motorola	–
68341	Gleitkomma-ROM	Motorola	–
68450	DMA-Controller (DMAC)	Hitachi	Motorola, Rockwell
68451	Speicherverwaltungseinheit (MMU)	Motorola	Rockwell
68540	Fehlererkennungs- und Korrekturbaustein (EDCC)	Motorola	–
68560	Serieller DMA-Prozessor (SDMA)	Motorola	–
68561	Multiprotokoll-Kommunikationskontroller (MPCC)	Rockwell	Motorola

Tabelle 8.1
Peripheriebausteine für den 68 000

In Kürze können bis zu 30 solche Bausteine erwartet werden. Dabei sind auch einige schon bestehende Schaltungen von Signetics, wie zum Beispiel
- 2661 programmierbare Kommunikationsschnittstelle (EPCI)
- 2652 Multi-Protokoll-Kommunikations-Controller (MPCC)
- 2653 Polynom-Generator Checker (PGC).

Von Motorola sind ein Hard-Disk-Controller, ein leistungsfähiger CRT-Controller, eine Multiprozessor-Schnittstelle und ein I/O-Prozessor geplant. Konzentrieren wir uns nun auf die in Tabelle 8.1 aufgeführten Bausteine:

Der intelligente *Peripherie-Controller (IPC) 68 120* ist ein vielseitig einsetzbarer, anwenderprogrammierbarer I/O-Controller. Da er mit einem 8-Bit-Mikroprozessor 6801 aufgebaut ist, kann er als I/O-Prozessor oder als Hilfsprozessoreinheit in einem verteilten Prozesssystem eingesetzt werden. Dieser IPC enthält neben der 6801-CPU auch noch

- eine Systemschnittstelle,
- eine serielle Kommunikationsschnittstelle,
- 21 parallele I/O-Leitungen,
- einen 16-Bit-Timer,
- einen 128-KByte-Lese/Schreib-Speicher (RAM),
- ein ROM von 2 KByte,
- 6 Hilfsregister.

Der *Terminal-Controller (CTC) 68 122* ist ein IPC, der als serielles I/O-Subsystem programmiert werden kann. Mit diesem können bis zu 32 Terminale an einem 68 000 angeschlossen werden.
Mit dem *Parallel-Schnittstellen-Timer (PIT) 68 230* können praktisch alle Anwendungen mit parallelen Schnittstellen und zeitabhängigen Anforderungen realisiert werden. Er enthält

- zwei doppelt gepufferte I/O-Schnittstellen für mehrere Betriebsarten,
- eine dritte 8-Bit-Schnittstelle,
- einen 24-Bit-Timer,
- Logik für priorisierte Interruptvektoren.

Das *Gleitkomma-ROM 68 341* besteht aus zwei Bausteinen, die dem Anwender erlauben, positionsunabhängige, «reentrante» Gleitkommaprogramme ablaufen zu lassen. Die Firmware des ROM unterstützt folgende Funktionen:

- Addition, Subtraktion, Multiplikation, Division;
- Quadratwurzel, Vergleiche, Absolutwert usw.

Der *DMA-Controller (DMAC) 68 450* kann vier unabhängige DMA-Kanäle bedienen, mit welchen bis zu 4 MByte pro s wort- oder byteweise übertragen werden kann.

Die *Speicherverwaltungseinheit (MMU) 68451* berechnet alle Adressentranslationen und Speicherschutzfunktionen für den ganzen Speicherbereich von 16 MByte. Mit einer MMU kann man innerhalb des gesamten Speicherbereiches Segmente bis zu 256 Byte hinunter definieren. Mit Hilfe der Funktionscodezeile des Prozessors definiert die MMU für jedes Segment einen logischen Adressbereich (zum Beispiel Programm- und Datenbereich für den Überwachungs- oder Benützerstatus). Sie kann auch einen Offset zu einer physikalischen Adresse und Speicherschutz für Segmente verarbeiten. Die MMU generiert einen speziellen Busfehler, falls ein nicht erlaubter Zugriff auf ein Segment gemacht wird.
Der *Fehlererkennungs- und Korrekturbaustein (EDCC) 68540* korrigiert Einzelbitfehler und erkennt Doppelbitfehler sowohl in 8-Bit- und 16-Bit-Datenbussystemen. Der EDCC kann für künftige Systeme direkt auf 32 Bit erweitert werden.
Der *Multi-Protokoll-Kommunikations-Controller (MPCC) 68561* ist ein leistungsfähiger Datenkommunikationsbaustein, der asynchrone, bitorientierte synchrone (X.25, SDLC, HDLC) und byteorientierte synchrone (BISYNC und DDCMP) Übertragungsprotokolle verarbeiten kann.

8.2 Peripheriebausteine der 6800er-Familie

Viele Anwendungen brauchen die hohe Leistungsfähigkeit der 68000er-Peripheriebausteine nicht. Hier können zum Teil die günstigeren Bausteine der 6800er-und 6500er-Serie verwendet werden. Die Tabelle 8.2 zeigt einige gebräuchliche 6800er-Bausteine. Ein jeder kann über die synchronen Steuerleitungen des 68000 betrieben werden (siehe Kapitel 6 der Benützeranleitung des MC 68000).

Typennummer	Beschreibung
MC 6821	Peripherer Schnittstellenadapter (PIA)
MC 6840	Programmierbarer Zeitgeber
MC 6843	Floppy-Disk-Controller (FDC)
MC 6845	Bildschirm-Controller (CRTC)
MC 6847	Videoanzeige-Generator (VDG)
MC 6850	Asynchrone Kommunikationsschnittstelle (ACIA)
MC 6852	Synchrone Kommunikationsschnittstelle (SSDA)
MC 6854	Erweiterter Datenlink-Controller (ADLC)
MC 6859	Datensicherheitsbaustein
MC 6860	Digital-Modem (0 bis 600 Bit/s)
MC 6862	Modulator (2400 Bit/s)
MC 68488	Schnittstelle für den IEEE-488-Bus (GPIA)

Tabelle 8.2
Verfügbare 6800er-Peripheriebausteine

Wir wollen nun untersuchen, wie der populäre Baustein MC 6821 (PIA) an den 68 000 angeschlossen werden kann.

8.3 Anschluss eines PIA an den 68 000

8.3.1 Der PIA 6821

Der PIA 6821 (**P**eripheral **I**nterface **A**dapter) hat alles Nötige, um einen Printer, Bildschirm, eine Tastatur, Schalttafel oder ähnliches an einen 6800 oder 68 000 anzuschliessen. Der PIA kommuniziert mit dem Mikroprozessor über die Systembusse (Daten, Adressen, Steuerung). Er kann mit den angeschlossenen Peripheriegeräten über zwei 8-Bit-Schnittstellen, Port A und Port B, verkehren. Jede der 8 Leitungen der beiden Ports kann unabhängig von etwas anderem als Ein- oder Ausgang programmiert werden.
Im PIA wird jede bidirektionale Schnittstelle (Port A und Port B) unterstützt durch:

- ein *Datenrichtungsregister.* Jedes Bit des Datenrichtungsregisters bestimmt, ob die entsprechende Portleitung als Eingang (0) oder als Ausgang (1) programmiert wurde.
- ein *Kontrollregister,* das die Interruptstatusbit eines Ports speichert und die internen logischen Verbindungen innerhalb des PIA auswählt.
- ein *Peripheriedatenregister,* das Daten zwischen dem Mikroprozessor und der angeschlossenen Peripherie zwischenspeichert.
- zwei *Interruptsteuerleitungen,* die ihre Wirkung je nach Inhalt des Kontrollregisters erhalten.

Es sind sechs Register im PIA adressierbar:
- zwei Peripheriedatenregister;
- zwei Datenrichtungsregister;
- zwei Kontrollregister.

Jedes periphere Register teilt ein Speicherbyte mit einem Datenrichtungsregister. Somit brauchen wir nur vier (anstatt sechs) Adressen für den PIA. Leser, die die Eigenschaften des 6821-PIA nicht kennen, können diese im Datenblatt des Bausteins nachschlagen.
Wie alle 8-Bit-Bausteine, kann auch der 6821-PIA Informationen von 8 Bit parallel transferieren. Um mehr als 8 Bit zu übertragen, braucht es zusätzliche Transfers, falls nur ein PIA dafür vorhanden ist. Da der 68 000 über einen 16-Bit-Datenbus verfügt, kann er 16 Bit gleichzeitig übertragen, was mit zwei der oben beschriebenen PIAs möglich ist (einen für die höheren 8 Bit und einen für die tieferen 8 Bit).

8.3.2 Schnittstelle für 16-Bit-Datentransfer

Bild 8.1 zeigt ein Beispiel, wie zwei PIAs am Synchronbus des 68 000 angeschlossen werden können, um 16-Bit-Informationen gleichzeitig zu übertragen. In diesem Beispiel wird angenommen, dass die 6800-Peripheriebausteine alle im Adressenbereich von $FEF800 bis $FEFF00 verdrahtet sind, weil die gültige periphere Adresse (VPA) nur mit dem Adressenstrobe (AS) aktiv und dem Ausgang des 13-Eingangs-NAND (74LS133) logisch 0 aktiv werden kann. Im weiteren ist aus Bild 8.1 zu lesen, dass die beiden PIAs nur selektiert werden, falls die Adressen A3 ... A5 logisch 1 sind. Damit sind die beiden Bausteine nur im Adressenbereich $FEF838 bis $FEFFFF ansprechbar. Die anderen beiden Adressenbit A1 und A2 der PIA werden für das Selektieren der internen Register wie folgt verwendet:

A2	A1	Selektiertes Register
0	0	Peripherieregister A / Datenrichtungsregister A (PRA/DDRA)
0	1	Kontrollregister A (CRA)
1	0	Peripherieregister B / Datenrichtungsregister B (PRB/DDRB)
1	1	Kontrollregister B (CRB)

Weil jeder PIA vier Speicherbyte besetzt, brauchen die zwei PIAs in Bild 8.1 acht Byte (4 Worte), die vier geraden Byte für den PIA mit D8 ... D15 und die ungeraden Byte für den PIA mit D0 ... D7. Wir wollen annehmen, dass unsere PIAs die Adressen $FEFF00 bis $FEFF07 (siehe Bild 8.2) besetzen.

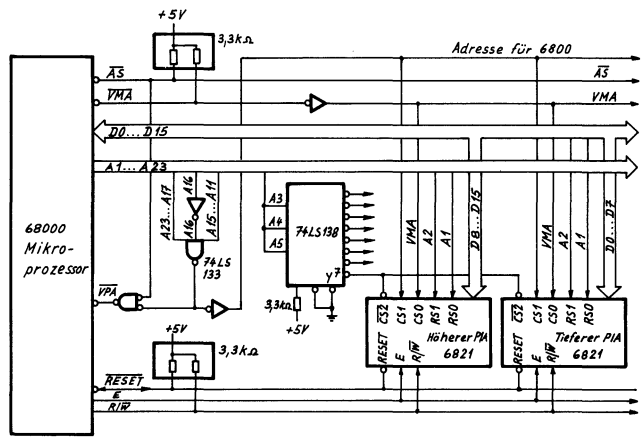


Bild 8.1
Schnittstelle zwischen einem 68 000 und zwei PIAs (6821)

Gerade Adresse	Oberer PIA	Unterer PIA	Ungerade Adresse
$FEFF00	PRA/DDRA	PRA/DDRA	$FEFF01
$FEFF02	CRA	CRA	$FEFF03
$FEFF04	PRB/DDRB	PRB/DDRB	$FEFF05
$FEFF06	CRB	CRB	$FEFF07

Bild 8.2
Anordnung der PIA-Register im Speicher.
Es bedeuten: PRA/DDRA = Peripherieregister/Datenrichtungsregister A-Seite
CRA = Kontrollregister A-Seite (analog für B-Seite).

8.3.3 Einfache 16-Bit-Transfers mit PIA

Aus Illustrationsgründen nehmen wir an, dass die PIAs in Bild 8.1 an zwei 16-Bit-Peripheriegeräte angeschlossen sind. Das Gerät, das an Port A der beiden PIAs hängt, sei nur ein Eingangsgerät (z. B. Schalterreihe). Wenn dieses Gerät ein Datenwort an Port A beider PIAs angelegt hat, teilt es dies dem 68 000 mittels eines *Daten-bereit*-Signales über Anschluss CA1 des oberen PIA mit. Nachdem der 68 000 das Wort in den Speicher gelesen hat, gibt er mit dem Signal *Daten angenommen* über den Anschluss CA2 dem Peripheriegerät bekannt, dass die Daten gelesen wurden.
Das an Port B beider PIAs angeschlossene Peripheriegerät sei nur ein Ausgabegerät (z. B. Gruppe von LED-Anzeigen). Wenn das Gerät bereit ist, ein Datenwort zu empfangen, sendet es dem 68 000 ein Signal *Peripherie bereit* auf Anschluss CB2 des oberen PIA. Der 68 000 gibt darauf ein Ausgabedatenwort auf Port B beider PIAs und teilt das mit dem Signal *Ausgabe bereit* auf dem Anschluss CB2 des oberen PIA dem Peripheriegerät mit. Bild 8.3 zeigt die eben beschriebenen Datenpfade.

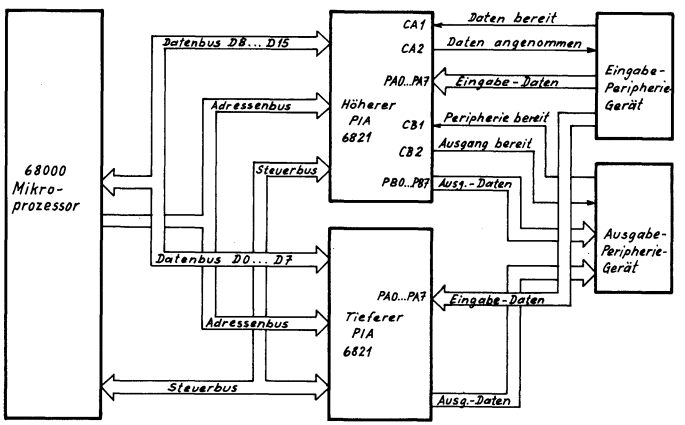


Bild 8.3
Anschluss zweier Peripheriegeräte an den Mikroprozessor 68 000 mit zwei PIAs 6821.

Wenn ein PIA mit dem angeschlossenen Peripheriegerät kommunizieren will, muss er zuerst dafür programmiert werden. PIAs werden während der Systeminitialisierung als Teil einer

Kaltstartroutine programmiert. Das Programmbeispiel 8.1 ist eine Initialisierungsroutine für zwei PIAs, wie eben beschrieben.

Der obere PIA ist so programmiert:
- DDRA alles 0 → Port A wirkt als Eingang,
- CRA → %00100110 ($26), Aktivierung des Handshakings
- DDRB alles 1 ($FF) → Port B wirkt als Ausgang,
- CRB → %00100110 ($26), Aktivierung des Handshakings.

Danach wird der untere PIA initialisiert:
- DDRA alles 0 → Port A wirkt als Eingang,
- CRA → %00000100 ($04), PRA angewählt,
- DDRB alles 1 → Port B wirkt als Ausgang,
- CRB → %00000100 ($04), PRB angewählt.

Programmbeispiel 8.1 Initialisierung von zwei PIAs

```
PIAD    EQU     $FEFF00     Adresse von PRA/DDRA
PIAC    EQU     PIAD+2      Adresse von CRA
PIBD    EQU     PIAD+4      Adresse von PRB/DDRB
PIBC    EQU     PIAD+6      Adresse von CRB
*
        MOVEA.L PIAD.A0     Zeigt auf den oberen PIA
* Programmiert               den oberen PIA
        MOVE.L  #$26FF26,D0 Setzt die Parameter auf und
        MOVEP.L D0,0(A0)    sendet sie zum PIA(H)
* Programmiert               den unteren PIA
        MOVE.L  #$04FF04,D0 Setzt die Parameter auf und
        MOVEP.L D0,0(A0)    sendet sie zum PIA(L)
```

Wenn der PIA einmal initialisiert ist, läuft der Informationstransfer zu und von der angeschlossenen Peripherie relativ einfach ab. Um ein einfaches 16-Bit-Wort an die Peripherie zu schicken, muss zuerst auf das Bereitschaftssignal der Einheit gewartet werden. Darauf kann das Datenwort zum peripheren Register B des PIA geschickt werden. Diese Sequenz ist im Pro-

Programmbeispiel 8.2
Senden eines 16-Bit-Wortes an eine periphere Einheit

```
* Gibt das in D0 enthaltene Wort aus.
OUTW    TST.B   PIBC        Periphere Einheit bereit?
        BPL.S   OUTW        Warte bis es soweit ist,
        MOVE    D0,PIBD     dann gebe das Wort aus.
        MOVE    PIBD,PIBD   Lösche das Bereitschaftsflag.
```

grammbeispiel 8.2 dargestellt. Darin werden als Ausgabewort die unteren 16 Bit des Datenregisters D0 verwendet. Die scheinbar unwirksame Instruktion MOVE PIBD,PIBD am Ende des Programms bewirkt mit der Lese-Operation nur das Löschen des peripheren Bereitschaftsflags in Bit 7 des Kontrollregisters.
Im Beispiel 8.3 ist gezeigt, dass das Übertragen von mehreren Datenwörtern beinahe so einfach ist wie für eines. Dieses Programm schreibt den Inhalt des Registers D0 regelmässig in den Ausgang, indem das Wort in D0 nach jedem Transfer inkrementiert wird.

Programmbeispiel 8.3
Wiederholtes Inkrementieren und Ausgabe eines 16-Bit-Wortes

```
* Das Ausgabewort ist ständig im D0. Es wird nach jedem Transfer inkrementiert.

OUTD0   TST.B   PIBC        Periphere Einheit bereit?
        BPL.S   OUTD0       Warte bis sie soweit ist,
        MOVE    D0,PIBD     dann gib das Wort aus.
        ADDQ    #1,D0       Erhöhe D0 um 1.
        MOVE    PIBD,PIBD   Lösche das Bereitschaftsflag
        BRA.S   OUTD0       und wiederhole.
```

Das Programm im Beispiel 8.4 zeigt eine typische Eingaberoutine, in der 35 Worte eingelesen und in sich folgenden Speicherplätzen abgespeichert werden sollen. Der Transfer ist mit indirekter Adressierung und nachträglichem Inkrementieren realisiert, damit die Adresse automatisch auf den nächsten Speicherplatz zeigt. Das Register D0 ist absichtlich mit 34 anstatt 35 initialisiert worden, weil die Abschlussinstruktion (DBF D0, IN35) das Programm erst terminiert, wenn D0 den Inhalt –1 und nicht 0 hat.

Programmbeispiel 8.4
Daten von einer peripheren Einheit lesen und abspeichern

```
* Dieses Programm liest 35 Datenworte in den Speicher, wobei die jeweilige Speicher-
* adresse in A0 steht.

        MOVE.L  #34,D0      Zähler in D0 setzen.
IN35    TST.B   PIAC        Daten bereit?
        BPL.S   IN35        Warte bis es soweit ist,
        MOVE    PIAD,(A0)+  dann lese ein Wort.
        DBF     D0,IN35     Wiederhole für 35 Wörter.
```

9. Unterstützung für den MC 68000

Die Leistungsmerkmale eines Mikroprozessors, sein Preis und seine Verfügbarkeit sind wichtige Faktoren für den Erfolg eines derartigen Produktes. Nicht weniger wichtig ist jedoch die ganze Unterstützung, die ein Anwender vom Hersteller und Lieferanten in Form von Bauelementen, fertigen Karten, leistungsfähigen Entwicklungssystemen und geeigneten Softwarepaketen erwarten kann. Sie allein macht einen Mikroprozessor in der Anwendung erst «lebensfähig». Es ist Ziel dieses Kapitels, eine Übersicht der von Motorola erhältlichen Bausteine für den MC 68000 zu geben. Darüber hinaus gibt es eine ganze Reihe von Firmen, die den MC 68000 unterstützen, sei es durch «second source» der Bauteile (z.B. Thomson-Efcis, Mostek, Signetics/Philips), eigene Peripheriebausteine, Entwicklungssysteme (GenRad), Hewlett-Packard, Philips, Tektronix usw.) oder durch Softwarepakete.

9.1 M 68000 – eine Prozessorfamilie

Mit der Einführung ihres Mikroprozessors MC 68000 im Jahr 1979 folgte Motorola ihrem Konzept der Mikroprozessor-Familie, wie sie es bereits beim MC 6800 im Jahr 1974 für 8-Bit-Maschinen getan hatte. Im Bild 9.1 ist die Familie der M68000er-Mikroprozessoren dargestellt (siehe dazu auch die Tabelle 9.1).

9.1.1 MC 68000

Die Mikroprozessoren MC 68000L4 bis MC 68000L12 unterscheiden sich lediglich durch die Frequenz des Taktes. Das Modell L4 arbeitet mit 4 MHz, das Modell L12 mit 12 MHz. Im übrigen entspricht diese Gruppe von 16-Bit-Maschinen genau der Beschreibung in dieser Artikelfolge.

9.1.2 MC 68008

Das Modell MC 68008 stellt eine Version des MC 68000 mit einem auf 8 Bit reduzierten Datenbus dar. Der MC 68008 ist vollständig softwarekompatibel mit dem MC 68000, sowohl im

Baustein-nummer MC.....	Kurzbe-zeichnung	Beschreibung	Eigenschaften, Verwendung
68120/ 68121	IPC	*Intelligent Peripheral Controller* Universeller, intelligenter Peripherie-Steuerbaustein	21 parallele I/O-Leitungen; Seriekanal; 128 Byte RAM; 2 KByte ROM (nur 68120); 16-Bit-Zeitgeber; Semaphore-Register; externe und interne Interrupts.
68230	PIT	*Parallel Interface and Timer* Parallel-Interface und Zeitgeber	24 I/O-Leitungen; 24-Bit-Zeitgeber; Logik für Interruptvektor-Erzeugung.
68440	DDMA	*Dual Direct Memory Access Controller* Steuerbaustein für direkten Speicherzugriff	Adressierbereich 16 MByte; Datenübertragung von Speicher zu Speicher, von Speicher zu Peripherie, von Peripherie zu Speicher; zwei unabhängige Kanäle; Subset des MC68450.
68450	DMAC	*Direct Memory Access Controller* Steuerbaustein für direkten Speicherzugriff	4 unabhängige Kanäle; Array-Operationen; zwei vektorisierte Interrupts pro Kanal.
68451	MMU	*Memory Management Unit* Speicherverwaltungseinheit	32 Speichersegmente mit variabler Grösse; berechnet physikalische Adressen aus logischen Adressen mittels Funktionscode (FC0...FC2); schützt Speicherbereiche vor unerlaubten Zugriffen; unterstützt Multitasking-Betriebs-systeme.
68452	BAM	*Bus Arbitration Module* Baustein für die Bussteuerung	Teilt den Bus nach einem Prioritätsschema einem von bis zu 8 Bus-Mastern zu.
68652	MPCC	*Multi Protocol Communications Controller* Steuerbaustein für synchronen Datenverkehr	Unterstützt bitorientierte Protokolle wie SDLC, ADCCP, HDLC, X.25; Byte-Steuerungsprotokolle wie DDCMP, BISYNC; Datenübertragungsgeschwindigkeit bis 2 MBaud.
68653	PGC	*Polynominal Generator Checker* Polynomerzeuger und -prüfer	Erzeugt und prüft Paritätsbit; erzeugt und prüft Prüfzeichen; für Datenübertragung zwischen Prozessor und synchronen und asynchronen Sendern und Empfängern; kompatibel mit MC68652 und MC68661.
68661	EPCI	*Enhanced Programmable Communications Interface* Universeller Interfacebaustein für synchrone und asynchrone Datenübertragung	Sendet parallele Daten in serieller Form und empfängt unabhängig davon serielle Daten für parallele Weiterverarbeitung; Baudrate bis 1 MBit/s; 3 verschiedene Baudratensets (Modell A, B, C).
68881	FPCP	*Floating-Point Co-Processor* Coprozessor für Fliesskomma-arithmetik	Leistungsfähiger Fliesskomma-Coprozessor mit der Komplexität des 32-Bit-Mikro-prozessors MC68020; acht 80-Bit-Register für Daten in Fliesskommadarstellung; umfasst den vorgeschlagenen IEEE-Standard und geht weit über diesen hinaus.

Tabelle 9.1
Einige neue Peripheriebausteine, speziell entwickelt für die Familie der Mikroprozessoren MC 68000

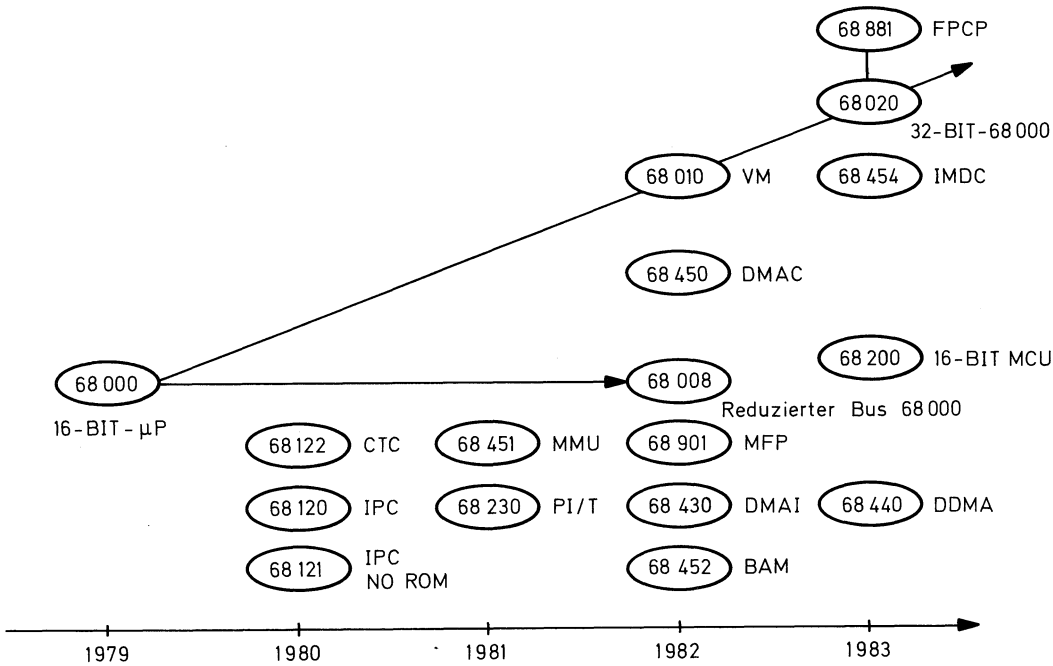


Bild 9.1
Die 68000-Familie (siehe dazu auch Tabelle 9.1)

Quellencode wie auch im Objektcode. Er wird dadurch zur kostengünstigen Alternative zu grösseren Systemen, ohne auf die Vorteile eines ausgereiften 16/32-Bit-Konzepts zu verzichten.

9.1.3 MC68010

Mit der Bezeichnung «Virtual Memory Processor» stellt der MC 68010 eine leistungsfähige Erweiterung des Grundmodells MC 68000 dar. Die Erweiterung umfasst im wesentlichen drei Punkte:

- Der Prozessor speichert bei einem auftretenden Busfehlersignal (BERR) automatisch den vollständigen Zustand ab. Nach Beheben des Fehlers kann der Zustand wieder automatisch hergestellt werden.
- Der MC 68010 enthält Mechanismen zum Einsatz als virtueller Prozessor in virtuellem Speicher und übernimmt alle Massnahmen, die diese Technik erfordert.

- Verzögerte Busfehlersignale können verarbeitet werden. Als Anwendung dieser Eigenschaft steht die Fehlersuch- und -korrekturtechnik (EDAC, *e*rror *d*etection *a*nd *c*orrection) im Vordergrund, die mit dem MC 68010 so gehandhabt wird, dass die Ausführungsgeschwindigkeit nicht beeinträchtigt ist, falls *kein* Fehler auftritt.

9.1.4 MC68020

Der MC 68020 steht kurz vor seiner Einführung und ist als logische Folge des 16/32-Bit-Konzeptes der Familie ein wahrer 32-Bit-Prozessor. Die Möglichkeiten des MC 68000 sind hier noch erweitert worden, so durch einige leistungsfähige Befehle, durch höhere Ausführungsgeschwindigkeiten, mehr Adressierungsarten, zusätzliche Betriebssystem-Unterstützungsmechanismen, Cache-Speicher und neue Bustechniken. Ein Fliesskomma-Coprozessor MC 68881 mit acht 80-Bit-Fliesskomma-Datenregistern macht mit dem MC 68020 eine aussergewöhnlich leistungsfähige Prozessorgruppe. Der Coprozessor MC 68881 erfüllt die Bedingungen der von der IEEE vorgeschlagenen Norm und geht weit darüber hinaus.

9.2 Peripheriebausteine

9.2.1 Spezielle Bausteine der 68000er-Familie

Peripheriebausteine, die speziell für die Mikroprozessoren aus der Familie 68000 entwickelt wurden, sind in der Tabelle 9.1 aufgeführt.

9.2.2 Weitere geeignete Bausteine

Neben den in Tabelle 9.1 aufgeführten speziellen Peripheriebausteinen sind weitere Interface-ICs erhältlich, die im Zusammenhang mit dem 8-Bit-Mikrocomputer MC 6800 eingeführt wurden. Ihr Anschluss an den MC 68000 erfolgt so, wie dies grundsätzlich im Kapitel 8.2 für den Peripheriebaustein PIA (MC 6821) beschrieben wurde. Die Tabelle 9.2 führt eine Reihe solcher Komponenten auf.

9.3 Lehrsystem

Für den Einsatz in Schulen, Instituten und Entwicklungslaboratorien wird ein Lehrsystem auf einer Karte mit der Bezeichnung MEX 68KECB angeboten. Es stützt sich auf den MC 68000, beinhaltet 32 KByte RAM, ein Monitorprogramm in zwei ROMs zu je 8 KByte, zwei serielle Schnittstellen RS 232 mit Baudratensteuerung (110 bis 9600 Baud), eine Centronics-Druckerschnittstelle (Bild 9.2).

Baustein-nummer	Beschreibung	Bemerkungen
MC 6821	*Peripheral Interface Adapter (PIA)* Universeller Parallel-Interface-baustein	16 I/O-Leitungen; alle einzeln in der Richtung programmierbar; 4 Handshake-Leitungen.
MC 6822	*Industrial Interface Adapter (IIA)* Interfacebaustein für die industrielle Umgebung	Wie 6821, jedoch dank «Open drain»-Ausgängen bis 18 V belastbar und damit auch CMOS-kompatibel bei 15 V.
MC 6828	*Priority Interrupt Controller (PIC)* Steuerbaustein für Interrupt-prioritäten	8 Interrupts; erzeugt Vektoradressen; berücksichtigt Prioritäten.
MC 6835	*CRT-Controller (CRTC)* Steuerbaustein für Bildschirme	Alphanumerische, halbgrafische und grafische Betriebsarten; unterstützt zwei verschiedene Bildschirmformate.
MC 6840	*Programmable Timer (PTM)* Zeitgeberbaustein	3 unabhängige Zeitgeber mit je 16 Bit; für Rechteckgeneratoren, Zeitverzögerungen, Einzelimpulse, Pulsbreitenmodulation, Frequenzvergleich.
MC 6847	*Video Display Generator (VDG)*	
MC 6850	*Asynchronous Communications Interface Adapter (ACIA)* Asynchroner Datenübertragungs-baustein (UART)	Überträgt bis 1 MBit/s; für 8- und 9-Bit-Übertragung; zweifach gepufferte Daten; mit Steuerfunktionen für Modems.
MC 6852	*Synchronous Serial Data Adapter (SSDA)* Synchroner Datenübertragungs-baustein	Für bidirektionalen, synchronen Datenverkehr; bis 1,5 MHz Taktfrequenz; 3 Byte FIFO-Speicher beim Sender und Empfänger; Modemfunktionen.
MC 6854	*Advanced Data Link Controller (ADLC)*	Interface zwischen Prozessor und Datenkanälen der Standards ADCCP, HDLC, SDLC.
MC 6859	*Data Security Device* Baustein für Datenschutz	Mit kryptografischem Algorithmus nach USA-Standard DES.
MC 6860	*Digital Modem*	Erzeugt Modulation, Demodulation von seriellen Signalen für FSK (frequency shift keying) bis 600 Bit/s; kompatibel mit ACIA (MC 6850).
MC 6862	*Digital Modulator*	Für 1200/2400 Bit/s.
MC 68488	*General Purpose Interface Adapter* Schnittstelle zu IEC-Bus	Ermöglicht Listener- und Talker-Betrieb mit allen Protokollvorschriften des IEC-Bus (IEEE 488, GPIB).

Tabelle 9.2
Peripheriebausteine aus der Familie des 8-Bit-Mikroprozessors MC 6800, auch geeignet für den MC 68000

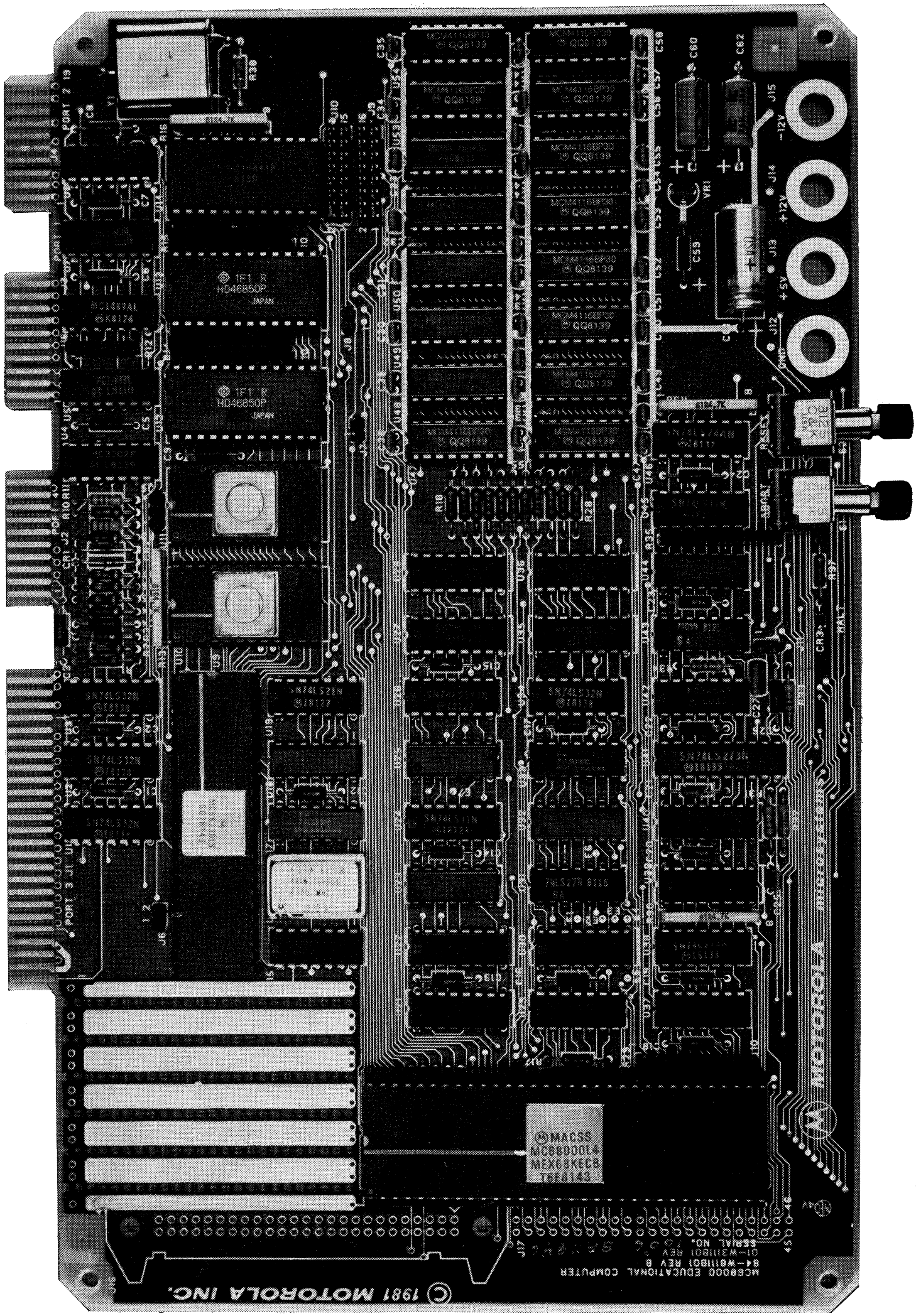

MACSS
MC68000L4
MEX68KECB
T6E8143
MC68000 EDUCATIONAL COMPUTER
84-W8111B01 REV B
01-W3111B01 REV B
SERIAL NO.
© 1981 MOTOROLA INC.
MOTOROLA
HD46850P
JAPAN

Bild 9.2
Das Lehrsystem MEX68KECB ist ein ideales Hilfsmittel für den Einstieg in das Arbeiten mit dem MC 68000

Mit Hilfe des Monitorprogramms können Programme mit einem einfachen Zeilenassembler entwickelt und ausgetestet werden. Programme, die auf einem grösseren System mittels «Crosssoftware» entwickelt wurden, lassen sich dank eingebautem Ladeprogramm bequem in das Lehrsystem umladen.
Das Monitorprogramm verwendet die gleiche Syntax wie «MACSbug», «VERSAbug» und «VMEbug», den Unterstützungsprogrammen, wie sie für die grösseren Varianten von Entwicklungssystemen erhältlich sind.

9.4 VME-Bus

Auf der Basis des VERSA-Bussystems, das eigens für grössere Anwendungen mit dem MC 68000 geschaffen wurde und für das eine ganze Familie von Karten mit der Bezeichnung

Nummer	Beschreibung
MVME 101	Einplatinencomputer (MC 68000) mit seriellen und parallelen I/O-Leitungen
MVME 110-1	Einplatinencomputer (MC 68000) mit I/O-Kanalschnittstelle
MVME 200	Speicherkarte, 64 KByte RAM mit Parität
MVME 201	Speicherkarte, 256 KByte RAM mit Parität
MVME 210	Speicherkarte, 128 KByte RAM/ROM
MVME 300	Controllerkarte für den GPIB (IEEE 488)
MVME 310	Universeller Peripheriecontroller
MVME 315	Diskcontroller zu SASI-Adapter (Shugart)
MVME 400	Zweifache Serieschnittstelle RS 232
MVME 410	Zweifache Parallelschnittstelle (je 16 Bit)
MVME 420	Peripherieadapter (SASI von Shugart)
MVME 435	Magnetbandadapter für 9 Spuren
MVME 600	8/16 Kanäle mit analogem Eingang (12-Bit-Umsetzer)
MVME 605	4 Kanäle mit analogen Ausgängen (12-Bit-Umsetzer)
MVME 610	Optokoppler-Eingänge für 120/240 V~
MVME 615	Optokoppler-Ausgänge für 120/240 V~ mit Nulldurchgangsdetektor
MVME 616	Optokoppler-Ausgänge für 120/240 V~
MVME 620	Optokoppler-Eingänge für 30 V–
MVME 625	Optokoppler-Ausgänge für 30 V–
MVME 920	VME-Busplatine für 20 Einschübe
MVME 921	VME-Busplatine für 9 Einschübe
MVME 922	I/O-Kanal-Busplatine für 5 Einschübe
MVME 941	Gehäuse für 9 VME-Bus- und I/O-Kanalkarten

Tabelle 9.3 Lieferbare VME-Bus-Karten

«VERSAmodule» existiert, entstand in Zusammenarbeit mit den Firmen Mostek, Signetics/Philips und Thomson/Efcis das Konzept des VME-Bus. Es handelt sich dabei um ein flexibles Bussystem für Europakarten, das speziell den Bedürfnissen des industriellen Bereichs (Modularität) genügt. Der VME-Bus zeichnet sich unter anderem durch folgende Eigenschaften aus:

- Geeignet für Multiprozessorsysteme;
- unterstützt Mikroprozessoren bis 32 Bit;
- Datendurchsatz bis 20 MByte/s;
- asynchrones, multiplexfreies Busprotokoll;
- Busbelegung prioritätsgesteuert (4 Ebenen);
- Interrupt-Verarbeitung zentral oder verteilt (7 Ebenen);
- Einfach-Europakarten (100 mm × 160 mm) für Peripherie-Schnittstellen;
- Doppel-Europakarten (233 mm × 160 mm) für Prozessor-, Speicher- und komplexe Funktionen.

Die oben genannten Firmen haben sich zu diesem Standardsystem verpflichtet und sind daran, eine ganze Reihe von Karten für dieses Buskonzept zu entwickeln und anzubieten. Eine Übersicht der bereits von Motorola erhältlichen Karten gibt die Tabelle 9.3.

9.5 Entwicklungssysteme

Neben dem in Kapitel 9.3 erwähnten Lehrsystem bietet Motorola ein leistungsfähiges Entwicklungssystem mit modularem Aufbau, «EXORmacs», an, das durch einen Hardware-Entwicklungszusatz HDS 400 verbunden werden kann. Der Zusatz wirkt dann in einem System als Echtzeitemulator.
Für die Beschreibung dieser Systeme wende man sich an die Lieferanten, ebenso für das umfangreiche Angebot an Entwicklungssoftware.

Anhang

Im folgenden werden einige nützliche Angaben in Tabellenform aufgeführt.

Anhang A

LSD \ MSD		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL

Tabelle A.1
Der ASCII-Zeichensatz ist ein 7-Bit-Code; ASCII steht für «American Standard Code for Information Interchange».
MSD = most significant digit, höherwertige Stelle;
LSD = least significant digit, tieferwertige Stelle.

Hexadezimal-Kolonne											
6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1.024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
7654		3210		7654		3210		7654		3210	
Byte				Byte				Byte			

Tabelle A.2
Umrechnungstabelle hexadezimal/dezimal

2^n	n
256	8
512	9
1 024	10
2 048	11
4 096	12
8 192	13
16 384	14
32 768	15
65 536	16
131 072	17
262 144	18
524 288	19
1 048 576	20
2 097 152	21
4 194 304	22
8 388 608	23
16 777 216	24

$2^0 = 16^0$
$2^4 = 16^1$
$2^8 = 16^2$
$2^{12} = 16^3$
$2^{16} = 16^4$
$2^{20} = 16^5$
$2^{24} = 16^6$
$2^{28} = 16^7$
$2^{32} = 16^8$
$2^{36} = 16^9$
$2^{40} = 16^{10}$
$2^{44} = 16^{11}$
$2^{48} = 16^{12}$
$2^{52} = 16^{13}$
$2^{56} = 16^{14}$
$2^{60} = 16^{15}$

Tabelle A.3
Potenzen von 2

16^n	n
1	0
16	1
256	2
4 096	3
65 536	4
1 048 576	5
16 777 216	6
268 435 456	7
4 294 967 296	8
68 719 476 736	9
1 099 511 627 776	10
17 592 186 044 416	11
281 474 976 710 656	12
4 503 599 627 370 496	13
72 057 594 037 927 936	14
1 152 921 504 606 846 976	15

Tabelle A.4
Potenzen von 16

Anhang B

Ausführungszeiten der Befehle des MC 68 000

Dieser Anhang enthält Tabellen, die die Befehlsausführungszeit als Anzahl externer Taktintervalle aufführen. Um die eigentliche Ausführungszeit für einen speziellen Befehl zu finden, muss der Wert aus der Tabelle mit dem Taktintervall des Mikroprozessors multipliziert werden.
Wenn Sie zum Beispiel einen 8-MHz-68000 benutzen, dann multiplizieren Sie den Wert mit 125 ns.
Die Zeitangaben in diesen Tabellen enthalten auch die Anzahl der «Bus-Lese-und-Schreib-Zyklen» für jeden Befehl. Diese Information ist in Klammern gesetzt und folgt der Anzahl Takt-

intervalle. Sie wird in der Form (L/S) dargestellt, wobei «L» die Anzahl von Lesezyklen und «S» die Anzahl von Schreibzyklen bedeutet.

	Adressierart
	Register
Dn	Datenregister direkt
An	Adressregister direkt
	Speicher
An	Adressregister indirekt
An +	Adressregister indirekt mit Nachinkrementierung
An –	Adressregister indirekt mit Vordekrementierung
An (d)	Adressregister indirekt mit Verschiebung
An (d, ix)*	Adressregister indirekt mit Index
xxx. W	Absolut kurz
xxx. L	Absolut lang
PC (d)	Programmzähler mit Verschiebung
PC (d, ix)*	Programmzähler mit Index
=xxx	unmittelbar

Tabelle B.1
Zeit für die Berechnung der effektiven Adresse

* Die Länge des Indexregisters (ix) beeinflusst die Ausführungszeit nicht.

Quelle	Ziel								
	Dn	An	An@	An@+	An@–	An@(d)	An@(d, ix)	xxx.W	xxx.L
Dn	4(1/0)	4(1/0)	9(1/1)	9(1/1)	9(1/1)	13(2/1)	15(2/1)	13(2/1)	17(3/1)
An	4(1/0)	4(1/0)	9(1/1)	9(1/1)	9(1/1)	13(2/1)	15(2/1)	13(2/1)	17(3/1)
An@	8(2/0)	8(2/0)	13(2/1)	13(2/1)	13(2/1)	17(3/1)	19(3/1)	17(3/1)	21(4/1)
An@+	8(2/0)	8(2/0)	13(2/1)	13(2/1)	13(2/1)	17(3/1)	19(3/1)	17(3/1)	21(4/1)
An@–	10(2/0)	10(2/0)	15(2/1)	15(2/1)	15(3/1)	19(3/1)	21(3/1)	19(3/1)	23(4/1)
An@(d)	12(3/0)	12(3/0)	17(3/1)	15(3/1)	17(3/1)	21(4/1)	23(4/1)	21(4/1)	15(5/1)
AN@(d, ix)*	14(3/0)	14(3/0)	19(3/1)	19(3/1)	19(3/1)	23(4/1)	25(4/1)	23(4/1)	27(5/1)
xxx.W	12(3/0)	12(3/0)	17(3/1)	17(3/1)	17(3/1)	21(4/1)	23(4/1)	21(4/1)	25(5/1)
xxx.L	16(4/0)	16(4/0)	21(4/1)	21(4/1)	21(4/1)	25(5/1)	27(5/1)	25(5/1)	29(6/1)
PC@(d)	12(3/0)	12(3/0)	17(3/1)	17(3/1)	17(3/1)	21(4/1)	23(4/1)	21(4/1)	25(5/1)
PC@(d, ix)*	14(3/0)	14(3/0)	19(3/1)	19(3/1)	19(3/1)	23(4/1)	25(4/1)	23(4/1)	27(5/1)
#xxx	8(2/0)	8(2/0)	13(2/1)	13(2/1)	13(2/1)	17(3/1)	19(3/1)	17(3/1)	21(4/1)

Tabelle B.2
Taktintervalle für MOVE (Byte und Wort)

* Die Länge des Indexregisters (ix) beeinflusst die Ausführungszeit nicht.

Quelle	Ziel								
	Dn	An	An@	An@+	An@–	An@(d)	An@(d, ix)	xxx.W	xxx.L
Dn	4(1/0)	4(1/0)	14(1/2)	14(1/2)	16(1/2)	18(2/2)	20(2/2)	18(2/2)	22(3/2)
An	4(1/0)	4(1/0)	14(1/2)	14(1/2)	16(1/2)	18(2/2)	20(2/2)	18(2/2)	22(3/2)
An@	12(3/0)	12(3/0)	22(3/2)	22(3/2)	22(3/2)	26(4/2)	28(4/2)	26(4/2)	30(5/2)
An@+	12(3/0)	12(3/0)	22(3/2)	22(3/2)	22(3/2)	26(4/2)	28(4/2)	26(4/2)	30(5/2)
An@–	14(3/0)	14(3/0)	24(3/2)	24(3/2)	24(3/2)	28(4/2)	30(4/2)	28(4/2)	32(5/2)
An@(d)	16(4/0)	16(4/0)	26(4/2)	26(4/2)	26(4/2)	30(5/2)	35(5/2)	30(5/2)	34(6/2)
AN@(d, ix)*	18(4/0)	18(4/0)	28(4/2)	28(4/2)	28(4/2)	32(5/2)	34(5/2)	32(5/2)	36(6/2)
xxx.W	16(4/0)	16(4/0)	26(4/2)	26(4/2)	26(4/2)	30(5/2)	32(5/2)	30(5/2)	34(6/2)
xxx.L	20(5/0)	20(5/0)	30(5/2)	30(5/2)	30(5/2)	34(6/2)	36(6/2)	34(6/2)	38(7/2)
PC@(d)	16(4/0)	16(4/0)	26(4/2)	26(4/2)	26(4/2)	30(5/2)	32(5/2)	30(5/2)	34(6/2)
PC@(d, ix)*	18(4/0)	18(4/0)	28(4/2)	28(4/2)	28(4/2)	32(5/2)	34(5/2)	32(5/2)	36(6/2)
#xxx	12(3/0)	12(3/0)	22(3/2)	22(3/2)	22(3/2)	26(4/2)	28(4/2)	26(4/2)	30(5/2)

Tabelle B.3
Taktintervalle MOVE (Doppelwort)
* Die Länge des Indexregisters (ix) beeinflusst die Ausführungszeit nicht.

Befehl	Grösse	op <ea>, An	op <ea>, Dn	op Dn, <M>
ADD	Byte, Wort	8(1/0)+	4(1/0)+	9(1/1)+
	Doppelwort	6(1/0)+**	6(1/0)+**	14(1/2)+
AND	Byte, Wort	–	4(1/0)+	9(1/1)+
	Doppelwort	–	6(1/0)+**	14(1/2)+
CMP	Byte, Wort	6(1/0)+	4(1/0)+	–
	Doppelwort	6(1/0)+	6(1/0)+	–
DIVS	–	–	158(1/0)+*	–
DIVU	–	–	140(1/0)+*	–
EOR	Byte, Wort	–	4(1/0)***	9(1/1)+
	Doppelwort	–	8(1/0)***	14(1/2)+
MULS	–	–	70(1/0)+*	–
MULU	–	–	70(1/0)+*	–
OR	Byte, Wort	–	4(1/0)+	9(1/1)+
	Doppelwort	–	6(1/0)+**	14(1/2)+
SUB	Byte, Wort	8(1/0)+	4(1/0)+	9(1/1)+
	Doppelwort	6(1/0)+**	6(1/0)+**	14(1/2)+

Tabelle B.4
Taktintervalle für arithmetische, logische und Vergleichsbefehle
\+ Addiere Zeit für die Berechnung der effektiven Adresse * Zeigt Maximalwert. ** Total 8 Taktintervalle für Befehle, wenn die effektive Adresse «Registerdirekt» ist. *** Die einzige verfügbare Adressierart ist «Datenregister direkt».

Befehl	Grösse	op #, Dn	op #, M	op #, SR
ADDI	Byte, Wort	8(2/0)	13(2/1)+	–
	Doppelwort	16(3/0)	22(3/2)+	–
ADDQ	Byte, Wort	4(1/0)	9(1/1)+	–
	Doppelwort	8(1/0)	14(1/2)+	–
ANDI	Byte, Wort	8(2/0)	13(2/1)+	20(3/0)
	Doppelwort	16(3/0)	22(3/2)+	–
CMPI	Byte, Wort	8(2/0)	8(2/0)+	–
	Doppelwort	14(3/0)	12(3/0)+	–
EORI	Byte, Wort	8(2/0)	13(2/1)+	20(3/0)
	Doppelwort	16(3/0)	22(3/2)+	–
MOVEQ	Doppelwort	4(1/0)	–	–
ORI	Byte, Wort	8(2/0)	13(2/1)+	20(3/0)
	Doppelwort	16(3/0)	22(3/2)+	–
SUBI	Byte, Wort	8(2/0)	13(2/1)+	–
	Doppelwort	16(3/0)	22(3/2)+	–
SUBQ	Byte, Wort	4(1/0)	9(1/1)+	–
	Doppelwort	8(1/0)	14(1/2)+	–

Tabelle B.5
Taktintervalle für «unmittelbar»-Befehle
\+ Addiere Zeit für die Berechnung der eff. Adresse.

Befehl	Grösse	Register	Speicher
CLR	Byte, Wort	4(1/0)	9(1/1)+
	Doppelwort	6(1/0)	14(1/2)+
NBCD	Byte	6(1/0)	9(1/1)+
NEG	Byte, Wort	4(1/0)	9(1/1)+
	Doppelwort	6(1/0)	14(1/2)+
NEGX	Byte, Wort	4(1/0)	9(1/1)+
	Doppelwort	6(1/0)	14(1/2)+
NOT	Byte, Wort	4(1/0)	9(1/1)+
	Doppelwort	6(1/0)	14(1/2)+
Scc	Byte, n. erf.	4(1/0)	9(1/1)+
	Byte, erfüllt	6(1/0)	9(1/1)+
TAS	Byte	4(1/0)	11(1/1)+
TST	Byte, Wort	4(1/0)	4(1/0)+
	Doppelwort	4(1/0)	4(1/0)+

Tabelle B.6
Taktintervalle für «Einfachoperanden-Befehle»
\+ Addiere Zeit für Berechnung der effektiven Adresse.

Befehl	Grösse	Register	Speicher
ASR, ASL	Byte, Wort	6 + 2n(1/0)	9(1/1)+
	Doppelwort	8 + 2n(1/0)	–
LSR, LSL	Byte, Wort	6 + 2n(1/0)	9(1/1)+
	Doppelwort	8 + 2n(1/0)	–
ROR, ROL	Byte, Wort	6 + 2n(1/0)	9(1/1)+
	Doppelwort	8 + 2n(1/0)	–
ROXR, ROXL	Byte, Wort	6 + 2n(1/0)	9(1/1)+
	Doppelwort	8 + 2n(1/0)	–

Tabelle B.7
Taktintervalle für Schiebe- und Rotierbefehle
\+ Addiere Zeit für Berechnung der effektiven Adresse.

Befehl	Grösse	Dynamisch		Statisch	
		Register	Speicher	Register	Speicher
BCHG	Byte	–	9(1/1)+	–	13(2/1)+
	Doppelwort	8(1/0)*	–	12(2/0)*	–
BCLR	Byte	–	9(1/1)+	–	13(2/1)+
	Doppelwort	10(1/0)*	–	14(2/0)*	–
BSET	Byte	–	9(1/1)+	–	13(2/1)+
	Doppelwort	8(1/0)*	–	12(2/0)*	–
BTST	Byte	–	4(1/0)+	–	8(2/0)+
	Doppelwort	6(1/0)	–	10(2/0)	–

Tabelle B.8
Taktintervalle für Bitmanipulationsbefehle
\+ Addiere Zeit für die Berechnung der effektiven Adresse * Zeigt Maximalwert.

Ausnahme	Intervalle
Adressfehler	57(4/7)
Busfehler	57(4/7)
Unterbruch	47(5/3)*
Illegaler Befehl	37(4/3)
Privilegierter Bef.	37(4/3)
Trace	37(4/3)

Tabelle B.9
Taktintervalle für Ausnahmeverarbeitung
* Für die Unterbruchquittung werden vier Taktintervalle angenommen.

Befehl	Verschiebung	Trap oder Verzweig. ausgef.	Trap oder Verzweig. n. ausgef.
Bcc	Byte	10(1/0)	8(1/0)
	Wort	10(1/0)	12(2/0)
BRA	Byte	10(1/0)	–
	Wort	10(1/0)	–
BSR	Byte	20(2/2)	–
	Wort	20(2/2)	–
DBcc	richtig	–	12(2/0)
	falsch	10(2/0)	14(3/0)
CHK	–	43(5/3)+*	8(1/0)+
TRAP	–	37(4/3)	–
TRAPV	–	37(5/3)	4(1/0)

Tabelle B.10
Taktintervalle für Verzweigung und Trap-Befehl
+ Addiere Zeit für die Berechnung der effektiven Adresse * Zeigt Maximalwert.

Befehl	Grösse	An@	An@+	An@–	An@(d)	An@(d, ix)*	xxx.W	xxx.L	PC@(d)	PC@(d, ix)*
JMP	–	8(2/0)	–	–	10(2/0)	14(3/0)	10(2/0)	12(3/0)	10(2/0)	14(3/0)
JSR	–	18(2/2)	–	–	20(2/2)	24(2/2)	20(2/2)	22(3/2)	20(2/2)	24(2/2)
LEA	–	4(1/0)	–	–	8(2/0)	12(2/0)	8(2/0)	12(3/0)	8(2/0)	12(2/0)
PEA	–	14(1/2)	–	–	18(2/2)	22(2/2)	18(2/2)	22(3/2)	18(2/2)	22(2/2)
MOVEM	Wort	12 + 4n (3 + n/0)	12 + 4n (3 + n/0)	– –	16 + 4n (4 + n/0)	18 + 4n (4 + n/0)	16 + 4n (4 + n/0)	20 + 4n (5 + n/0)	16 + 4n (4 + n/0)	18 + 4n (4 + n/0)
M → R	D'w.	12 + 8n (3 + 2n/0)	12 + 8n (3 + 2n/0)	– –	16 + 8n (4 + 2n/0)	18 + 8n (4 + 2n/0)	16 + 8n (4 + 2n/0)	20 + 8n (5 + 2n/0)	16 + 8n (4 + 2n/0)	18 + 8n (4 + 2n/0)
MOVEM	Wort	8 + 5n (2/n)	– –	8 + 5n (2/n)	12 + 5n (3/n)	14 + 5n (3/n)	12 + 5n (3/n)	16 + 5n (4/n)	– –	– –
R → M	D'w.	8 + 10n (2/2n)	– –	8 + 10n (2/2n)	12 + 10n (3/2n)	14 + 10n (3/2n)	12 + 10n (3/2n)	16 + 10n (4/2n)	– –	– –

Tabelle B.11
Taktintervalle für JMP, JSR, LEA, PEA und MOVEM-Befehle
n ist die Anzahl der verschobenen Register * Die Länge des Indexregisters (ix) beeinflusst die Befehlsausführungszeit nicht.

Befehl	Grösse	op Dn, Dn	op M, M
ADDX	Byte Wort	4(1/0)	19(3/1)
	Doppelw.	8(1/0)	32(5/2)
CMPM	Byte Wort	–	12(3/0)
	Doppelw.	–	20(5/0)
SUBX	Byte Wort	4(1/0)	19(3/1)
	Doppelw.	8(1/0)	32(5/2)
ABCD	Byte	6(1/0)	19(3/1)
SBCD	Byte	6(1/0)	19(3/1)

Taktintervalle für Befehle mit mehrfacher Genauigkeit

Befehle	Grösse	Register	Speicher	Reg. Speich.	Speicher Reg.
MOVE von SR	–	6(1/0)	9(1/1)+	–	–
MOVE zu CCR	–	12(2/0)	12(2/0)+	–	–
MOVE zu SR	–	12(2/0)	12(2/0)+	–	–
MOVEP	Wort	–	–	18(2/2)	16(4/0)
	D'wort	–	–	28(2/4)	24(6/0)
EXG	–	6(1/0)	–	–	–
EXT	Wort	4(1/0)	–	–	–
	D'wort	4(1/0)	–	–	–
LINK	–	18(2/2)	–	–	–
MOVE von USP	–	4(1/0)	–	–	–
MOVE zu USP	–	4(1/0)	–	–	–
NOP	–	4(1/0)	–	–	–
RESET	–	132(1/0)	–	–	–
RTE	–	20(5/0)	–	–	–
RTR	–	20(5/0)	–	–	–
RTS	–	16(4/0)	–	–	–
STOP	–	4(0/0)	–	–	–
SWAP	–	4(1/0)	–	–	–
UNLK	–	12(3/0)	–	–	–

Tabelle B.13
Taktintervalle weiterer Befehle
\+ addiere Zeit für die Berechnung der effektiven Adresse

Anhang C

Der Befehlssatz des MC 68000

Dieser Anhang enthält drei zusammenfassende Tabellen. Tabelle C.1 enthält die Adressierarten des MC 68000 und gruppiert sie als Daten, Speicher, Steuerung oder änderbare Adressierarten. Sie zeigt auch die Assemblersyntax für jede Art. Die gleiche Tabelle ist im Kapitel 3 als Tabelle 3.4 vorhanden und ist hier zum schnellen Nachschlagen noch einmal wiedergegeben.

Tabelle C.2 enthält eine Zusammenstellung der Bedingungen, die durch die Bcc-, DBcc- und Scc-Befehle geprüft werden können. Diese Bedingungen erschienen bereits als Tabelle 3.15 in Kapitel 3.

Adressierungsarten	Adressierungskategorien Daten	Speicher	Steuerung	veränderbar	Assemblersyntax
Datenregister direkt	X			X	Dn
Adressregister direkt				X	An
Register indirekt	X	X	X	X	(An)
Register indirekt nachinkrementiert	X	X		X	(An)+
Register indirekt vordekrementiert	X	X		X	-(An)
Register indirekt mit Verschiebung	X	X	X	X	d(An)
Register indirekt mit Index	X	X	X	X	d(An.Ri)
Absolut kurz	X	X	X	X	xxxx
Absolut lang	X	X	X	X	xxxxxxxx
Relativ mit Verschiebung	X	X	X		d
Relativ mit Index	X	X	X		d(Ri)
Unmittelbar	X	X			=xxxx

Tabelle C.1
Effektive Adressierungsarten

Anhang «cc»	Bedingung	Trifft zu wenn
EQ	Gleich	$Z = 1$
NE	Nicht gleich	$Z = 0$
MI	Minus	$N = 1$
PL	Plus	$N = 0$
*GT	Grösser als	$Z \bullet (N \oplus V) = 0$
*LT	Kleiner als	$N \oplus V = 1$
*GE	Grösser oder gleich	$N \oplus V = 0$
*LE	Kleiner oder gleich	$Z + (N \oplus V) = 1$
HI	Höher als	$C \bullet Z = 0$
LS	tiefer oder gleich	$C + Z = 1$
CS	Übertrag gesetzt	$C = 1$
CC	Übertrag gelöscht	$C = 0$
*VS	Überlauf	$V = 1$
*VC	Kein Überlauf	$V = 0$
T	Immer wahr	
F	Immer falsch	

*Zweierkomplement-Arithmetik

Tabelle C.2
Bedingte Befehle

Symbole: ● = UND
+ = ODER
⊕ = EXKLUSIV ODER

Die Tabelle C.3 enthält den Befehlssatz des MC 68000 in alphabetischer Ordnung. Sie ist eine Zusammenstellung der im Kapitel 3 vermittelten Information über die Befehle. Zum besseren Aufsuchen ist hier das Befehlsrepertoire in alphabetischer Ordnung nochmals gedruckt.

Mnemonik	Assembler Syntax	Operandengrösse	Erlaubte Adressierungsarten		Bedingungscode				
			Quelle	Ziel	X	N	Z	V	C
ABCD	ABCD Dy,Dx	8	Dn	Dn	*	U	*	U	*
	ABCD -(Ay),-(Ax)	8	-(An)	-(An)	*	U	*	U	*
ADD	ADD <ea>,Dn	8, 16, 32	Alle (1)	Dn	*	*	*	*	*
	ADD Dn, <ea>	8, 16, 32	Dn	änderbar	*	*	*	*	*
ADDA	ADD <ea>,An	16, 32	Alle	An	–	–	–	–	–
ADDI	ADDI #d,<ea>	8, 16, 32	#d	Daten änderbar	*	*	*	*	*
ADDQ	ADDQ #d,<ea>	8, 16, 32	#d (2)	änderbar (1)	*	*	*	*	*
ADDX	ADDX Dy,Dx	8, 16, 32	Dn	Dn	*	*	*	*	*
	ADDX -(Ay),-(Ax)	8, 16, 32	-(An)	-(An)	*	*	*	*	*
AND	AND <ea>,Dn	8, 16, 32	Daten	Dn	–	*	*	0	0
	AND Dn,<ea>	8, 16, 32	Dn	änderbar	–	*	*	0	0
ANDI	ANDI #d,<ea>	8, 16, 32	#d	Daten änderbar	–	*	*	0	0
	ANDI #d,SR (3)	8, 16	#d	SR	*	*	*	*	*
ASL	ASL Dx,Dy	8, 16, 32	Dn (4)	Dn	*	*	*	*	*
	ASL #d,Dn	8, 16, 32	#d (5)	Dn	*	*	*	*	*
	ASL <ea>	16		Speicher änderbar	*	*	*	*	*
ASR	ASR Dx,Dy	8, 16, 32	Dn (4)	Dn	*	*	*	*	*
	ASR #d,Dn	8, 16, 32	#d (5)	Dn	*	*	*	*	*
	ASR <ea>	16		Speicher änderbar	*	*	*	*	*
Bcc	Bcc <label>	8, 16	Wenn cc, dann PC+d→PC		–	–	–	–	–
BCHG	BCHG Dn,<ea>	8, 32	Dn	Daten änderbar	–	–	*	–	–
	BCHG #d,<ea>	8, 32	#d	Daten änderbar	–	–	*	–	–
BCLR	BCLR Dn,<ea>	8, 32	Dn	Daten änderbar	–	–	*	–	–
	BCLR #d,<ea>	8, 32	#d	Daten änderbar	–	–	*	–	–
BRA	BRA <label>	8, 16	PC+d→PC		–	–	–	–	–
BSET	BSET Dn,<ea>	8, 32	Dn	Daten änderbar	–	–	*	–	–
	BSET #d,<ea>	8, 32	#d	Daten änderbar	–	–	*	–	–
BSR	BSR <label>	8, 16	PC→-(SP); PC+d→PC	Dn	–	–	–	–	–
BTST	BTST Dn,<ea>	8, 32	Dn	Ausgenommen unmittelbare Daten	–	–	*	–	–
	BTST #d,<ea>	8, 32	#d	Ausgenommen unmittelbare Daten	–	–	*	–	–
CHK	CHK <ea>,Dn	16	Wenn Dn <0 oder Dn> (ea), dann TRAP	Daten	–	*	U	U	U
CLR	CLR <ea>	8, 16, 32	Daten änderbar		–	0	1	0	0
CMP	CMP <ea>,Dn	8, 16, 32	Alle (1)	Dn	–	*	*	*	*
CMPA	CMPA <ea>,An	16, 32	Alle	An	–	*	*	*	*
CMPI	CMPI #d,<ea>	8, 16, 32	#d	Daten änderbar	–	*	*	*	*
CMPM	CMPM (Ay)+,(Ax)+	8, 16, 32	(An)+	(An)+	–	*	*	*	*
DBcc	DBcc Dn,<label>	16	Wenn cc, dann Dn-1→Dn; wenn Dn=-1, dann PC+d→PC		–	–	–	–	–
DIVS	DIVS <ea>,Dn	16	Daten	Dn	–	*	*	*	0
DIVU	DIVU <ea>,Dn	16	Daten	Dn	–	*	*	*	0
EOR	EOR Dn,<ea>	8, 16, 32	Dn	Daten änderbar	–	*	*	0	0
EORI	EORI #d,<ea>	8, 16, 32	#d	Daten änderbar	–	*	*	0	0
	EORI #d,SR (3)	8, 16	#d	SR	*	*	*	*	*
EXG	EXG Rx,Ry	32	Dn oder An	Dn oder An	–	–	–	–	–
EXT	EXT Dn	16, 32	Dn		–	*	*	0	0
JMP	JMP <ea>		<ea>→PC	Kontrolle	–	–	–	–	–
JSR	JSR <ea>		PC→-(SP); <ea>→PC	Kontrolle	–	–	–	–	–
LEA	LEA <ea>,An	32	Kontrolle	An	–	–	–	–	–
LINK	LINK An,#d		An		–	–	–	–	–
LSL	LSL Dx,Dy	8, 16, 32	Dn (4)	Dn	*	*	*	0	*
	LSL #d, Dn	8, 16, 32	#d (5)	Dn	*	*	*	0	*
	LSL <ea>	16		Speicher änderbar	*	*	*	0	*
LSR	LSR Dx,Dy	8, 16, 32	Dn (4)	Dn	*	0	*	0	*
	LSR #d,Dn	8, 16, 32	#d (5)	Dn	*	0	*	0	*
	LSR <ea>	16		Speicher änderbar	*	0	*	0	*
MOVE	MOVE ea,ea	8, 16, 32	Alle (1)	Daten änderbar	–	*	*	0	0
	MOVE ea, CCR	16	Daten	CCR	*	*	*	*	*
	MOVE ea, SR (6)	16	Daten	SR	*	*	*	*	*
	MOVE SR, ea	16	SR	Daten änderbar	–	–	–	–	–
	MOVE USP, An (6)	32	USP	An	–	–	–	–	–
	MOVE An, USP (6)	32	An	USP	–	–	–	–	–

Tabelle C.3
Befehlssatz des MC 68 000 in alphabetischer Ordnung

Mnemonik	Assembler Syntax	Operandengrösse	Erlaubte Adressierungsarten		Bedingungscode				
			Quelle	Ziel	X	N	Z	V	C
MOVEA	MOVEA <ea>,An	16, 32	Alle	An	–	–	–	–	–
MOVEM	MOVEM <list>, <ea>	16, 32		Kontrolle änderbar oder –(An)	–	–	–	–	–
	MOVEM <ea>, <list>	16, 32	Kontrolle oder (An)+	–	–	–	–	–	–
MOVEP	MOVEP Dx,d(Ay)	16, 32	Dn	d(An)	–	–	–	–	–
	MOVEP d(Ay),Dx	16, 32	d(An)	Dn	–	–	–	–	–
MOVEQ	MOVEQ #d,Dn	32	#d (7)	Dn	–	*	*	0	0
MULS	MULS <ea>,Dn	16	Daten	Dn	–	*	*	0	0
MULU	MULU <ea>,Dn	16	Daten	Dn	–	*	*	0	0
NBCD	NBCD <ea>	8		Daten änderbar	*	U	*	U	*
NEG	NEG <ea>	8, 16, 32	Daten änderbar		*	*	*	*	*
NEGX	NEGX <ea>	8, 16, 32	Daten änderbar		*	*	*	*	*
NOP	NOP		PC+2–PC		–	–	–	–	–
NOT	NOT <ea>	8, 16, 32		Daten änderbar	–	*	*	0	0
OR	OR <ea>,Dn	8, 16, 32	Daten	Dn	–	*	*	0	0
	OR Dn,<ea>	8, 16, 32	Dn	änderbar	–	*	*	0	0
ORI	ORI #d, <ea>	8, 16, 32	#d	Daten änderbar	–	*	*	0	0
	ORI #d,SR (3)	8, 16	#d	SR	*	*	*	*	*
PEA	PEA <ea>	32	Kontrolle		–	–	–	–	–
RESET (6)	RESET				–	–	–	–	–
ROL	ROL Dx,Dy	8, 16, 32	Dn (4)	Dn	–	*	*	0	*
	ROL #d,Dn	8, 16, 32	#d (5)	Dn	–	*	*	0	*
	ROL <ea>	16		Speicher änderbar	–	*	*	0	*
ROR	ROR Dx,Dy	8, 16, 32	Dn (4)	Dn	–	*	*	0	*
	ROR #d,Dn	8, 16, 32	#d (5)	Dn	–	*	*	0	*
	ROR <ea>	16		Speicher änderbar	–	*	*	0	*
ROXL	ROXL Dx,Dy	8, 16, 32	Dn (4)	Dn	*	*	*	0	*
	ROXL #d,Dn	8, 16, 32	#(5)	Dn	*	*	*	0	*
	ROXL <ea>	16		Speicher änderbar	*	*	*	0	*
ROXR	ROXR Dx,Dy	8, 16, 32	Dn (4)	Dn	*	*	*	0	*
	ROXR #d,Dn	8, 16, 32	# (5)	Dn	*	*	*	0	*
	ROXR <ea>	16		Speicher änderbar	*	*	*	0	*
RTE (6)	RTE		(SP)+→SP;(SP)+→PC		*	*	*	*	*
RTR	RTR		(SP)+→CCR; (SP)+→PC		*	*	*	*	*
RTS	RTS		(SP)+→PC		–	–	–	–	–
SBCD	SBCD Dy,Dx	8	Dn	Dn	*	U	*	U	*
	SBCD –(Ay),–(Ax)	8	–(An)	–(An)	*	U	*	U	*
Scc	Scc <ea>	8	Wenn cc, dann ls→(ea); sonst 0s→(ea)	Daten änderbar	–	–	–	–	–
STOP (6)	STOP #d	16	#d→SR, dann STOP		*	*	*	*	*
SUB	SUB <ea>,Dn	8, 16, 32	Alle (1)	Dn	*	*	*	*	*
	SUB Dn,<ea>	8, 16, 32	Dn	änderbar	*	*	*	*	*
SUBA	SUBA <ea>,An	16, 32	Alle	An	–	–	–	–	–
SUBI	SUBI #d,<ea>	8, 16, 32	#d	Daten änderbar	*	*	*	*	*
SUBQ	SUBQ #d,<ea>	8, 16, 32	#d (2)	änderbar (1)	*	*	*	*	*
SUBX	SUBX Dy,Dx	8, 16, 32	Dn	Dn	*	*	*	*	*
	SUBX –(Ay),–(Ax)	8, 16, 32	–(An)	–(An)	*	*	*	*	*
SWAP	SWAP Dn	16	Dn		–	–	–	–	–
TAS	TAS <ea>	8	Daten änderbar		–	*	*	0	0
TRAP	TRAP # <vector>		PC→–(SP);SR→–(SP); # <vector> →PC		–	–	–	–	–
TRAPV	TRAPV		Wenn V = 1, dann TRAP		–	–	–	–	–
TST	TST <ea>	8, 16, 32	Daten änderbar		–	*	*	0	0
UNLK	UNLK An			An	–	–	–	–	–

* Bedingungsbit wird beeinflusst

Bemerkungen:

(1) Wenn die Operationslänge Byte ist, ist die Adressierungsart «Adressregister direkt» nicht erlaubt
(2) unmittelbarer Operand, mit einem Wert von 1 bis 8
(3) Bei Wortoperationen ist der Befehl privilegiert
(4) Quellendatenregister enthält den Schiebewert: 0…63, wobei der Wert 0 einen Schiebewert 64 ergibt
(5) Daten sind der Schiebewert, 1 bis 8
(6) Diese Operation ist privilegiert
(8) Acht Bit unmittelbare Daten, welche vorzeichenerweitert werden zu einem 32Bit-Operanden

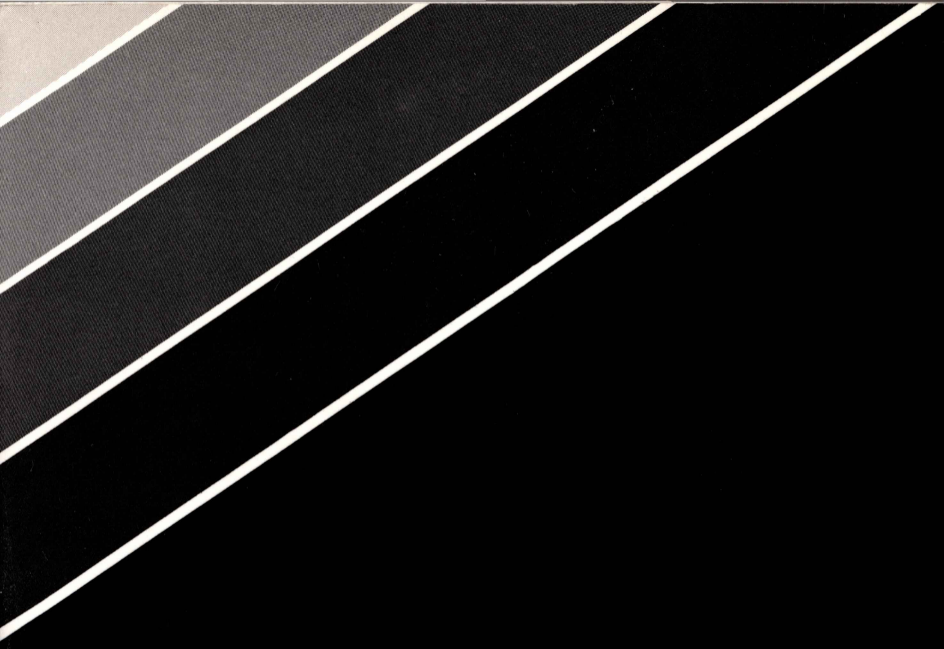
Im AT Verlag Aarau · Stuttgart erschienen u.a. folgende Fachbücher

Roland Best
Die Verarbeitung von Kleinsignalen in elektronischen Systemen
Hier wird erklärt, wie Störsignale in Elektroniksystemen entstehen und was der Elektroniker dagegen tun kann.

Albert Kloss
Stromrichter-Netzrückwirkungen in Theorie und Praxis
EMC der Leistungselektronik
Das Buch befasst sich mit der Beeinflussung der Netze durch leistungselektronische Einrichtungen. Es ist eine systematische Bearbeitung der theoretischen Grundlagen der Stromrichter-Netzrückwirkungen, besonders was die Blindleistung und die Oberschwingungen betrifft.

Andrew Stamberger
RC-Beschaltungen
Die Projektierung einer RC-Beschaltung in der Leistungselektronik
Der Band beschreibt Beschaltungssysteme von Stromrichtern und gibt Berechnungsbeispiele für die Beschaltung von Gleich- und Wechselstromstellern, Wechselrichtern und gesteuerten und ungesteuerten Gleichrichtern sowie Richtwerte für die Beschaltungselemente.

R. Dändliker
Laser-Kurzlehrgang
Eine konzentrierte Einführung in die Physik und Terminologie der Laser, Erläuterungen der wichtigsten Begriffe aus dem Bereich der Gas-, Festkörper-, Farbstoff- und Halbleiter-Laser sowie der nichtlinearen Optik und der Holographie.



Trotz der Vielseitigkeit der 4- und 8-Bit-Mikroprozessoren, die Anfang der 70er Jahre eingeführt wurden, können mit ihnen doch gewisse komplexe und schnelle Operationen nicht ausgeführt werden. Für diese anspruchsvollen Anwendungen bieten neuere 16-Bit-Mikroprozessoren, wie der Typ 68000 von Motorola, eine gültige Alternative sogar zu teureren Minicomputern. Dieses Buch beginnt mit grundlegenden Kapiteln und führt schrittweise zu komplexeren Themen, so dass der Leser den Mikroprozessor 68000 gründlich kennenlernt. Als Voraussetzung sind einige Kenntnisse binärer und hexadezimaler Zahlensysteme, boolescher Logik und der Grundlage von Assemblersprache von Vorteil.

ISBN-3-85502-152 X