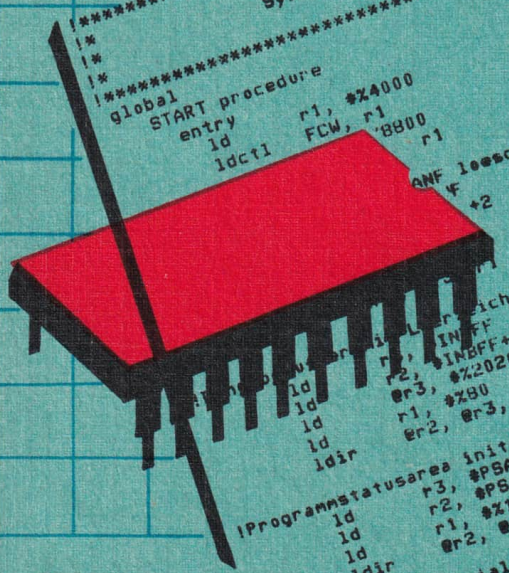
Technische Informatik

Brennenstuhl

Programmierung des 16-Bit-Mikroprozessorsystems U8000



VEB Verlag Technik Berlin

Brennenstuhl

Programmierung des 16-Bit-Mikroprozessorsystems U8000

Technische Informatik

Herausgegeben von

Dr.-Ing. Ludwig Claßen

Dr.-Ing. Dieter Nedo

Dr. rer. nat. Gerhard Paulin

Programmierung des 16-Bit-Mikroprozessorsystems U8000

Dr.-Ing. Heinz Brennenstuhl

VEB VERLAG TECHNIK BERLIN

Brennenstuhl, Heinz:
Programmierung des Mikroprozessorsystems
U8000 / Heinz Brennenstuhl. - 1. Aufl. -
Berlin : Verl. Technik, 1987. - 223 S. :
35 Bilder, 11 Taf.
ISBN 3-341-00286-3

ISBN 3-341-00286-3

1. Auflage

Lizenz 201 • 370/66/87
Printed in the German Democratic Republic
Druck und buchbinderische Verarbeitung:
(140) Druckerei Neues Deutschland, Berlin
Lektor: Jürgen Reichenbach
Einbandgestaltung: Gabriele Schwesinger
LSV 3043 • VT 3/5918-1
Bestellnummer: 553 763 7
02200

Vorwort

Die gegenwärtige Zeit ist geprägt durch eine stark anwachsende Zahl von Informationen in allen Bereichen des gesellschaftlichen Lebens. Dabei ist nicht nur ein volumenmäßiges Wachstum, sondern insbesondere bei technischen Prozessen, ein Anwachsen der Forderungen an die Genauigkeit der Information und die erforderliche Geschwindigkeit ihrer Verarbeitung festzustellen. Die Prozesse entwickeln sich unter den modernen Produktionsbedingungen in bezug auf ihre Komplexität und den Grad ihrer Verkettung ständig weiter. Ihre Beherrschung erfordert ein neues Niveau in der Informationsverarbeitung. Es ist durch leistungsfähige, dezentral organisierte und mit Intelligenz versehene Einrichtungen der Informationsverarbeitung gekennzeichnet.
Mit den 16-Bit-Mikroprozessoren ist eine Grundlage zur Realisierung dieser Zielstellung geschaffen worden. Sie ermöglichen in Verbindung mit weiteren mikroelektronischen Bauelementen den Aufbau von 16-Bit-Mikrorechnern und erschließen Anwendungsbereiche, die bisher den Kleinrechnern vorbehalten waren. Die 16-Bit-Mikrorechner verbinden die Rechenleistung der Kleinrechner mit den ökonomischen Vorteilen der Mikrorechentechnik.
Der Einsatz der Mikroprozessoren erfordert die Bereitstellung anwendungsspezifischer Software. Sie enthält den vom Mikroprozessor auszuführenden Algorithmus und gibt damit dem universellen mikroelektronischen Baustein "Mikroprozessor" eine spezielle Funktion. Durch Änderung oder Erweiterung des in einem Programm formulierten Algorithmus kann der Funktionsinhalt einer durch den Mikroprozessor mit Intelligenz versehenen Einrichtung geändert oder erweitert werden. Die Software entscheidet wesentlich über die Leistungsfähigkeit und die Einsatzgebiete der Mikroprozessoren. Sie enthält die Problemlösungen zur Rationalisierung von Anwenderaufgaben in verschiedenen Bereichen der Volkswirtschaft und bestimmt damit die Effektivität des Mikrorechnereinsatzes.
Die Mikrorechentechnik wird in alle Bereiche des gesellschaftlichen Lebens Einzug halten und viele Arbeitsplätze direkt beeinflussen. Besondere Bedeutung kommt zukünftig der Softwareerarbeitung für die in hoher Zahl als Personalcomputer eingesetzten 16-Bit-Mikrorechner zu. Nur durch neue Softwaresysteme können auf den Personalcomputern weitere Anwendungsgebiete erschlossen werden. Damit ist das Leistungsvermögen der Mikrorechner allein durch die Weiterentwicklung auf dem Gebiet der Software wesentlich erweiterbar.
Im vorliegenden Band der Reihe "Technische Informatik" wird die Programmierung des Mikroprozessorsystems U8000 erläutert. Der Aufbau, die Funktion und der Befehlssatz des Mikroprozessorsystems U8000 werden anhand von ausgewählten Beispielen der Assemblerprogrammierung vorgestellt. Die Programmierumgebung wird am Beispiel der U8000-Programmentwicklung erläutert. Zur Unterstützung der Einarbeitung in die U8000-Programmierung und zur Erleichterung der Inbetriebnahme von U8000-Mikrorechnersystemen wird ein komplettes Monitorprogramm vorgestellt.
Der Band wendet sich an alle Interessenten für die 16-Bit-Mikroprozessortechnik, an Entwicklungsingenieure, Programmierer, Lehrende und Lernende an den technischen Bildungseinrichtungen sowie an alle, die sich mit dem

Einsatz des 16-Bit-Mikroprozessorsystems U8000 befassen.
Trotz großer Sorgfalt bei der Erarbeitung des Manuskriptes ist es nicht auszuschließen, daß sich im Band Druck- oder Sachfehler befinden. Der Autor ist deshalb jederzeit für Hinweise dankbar. Diese sind an den VEB Verlag Technik oder an den Autor im

Zentrum für Forschung und Technologie
des KEAW "Friedrich Ebert"
Storkower Str. 101
Berlin
1055

zu richten.
Zu besonderem Dank bin ich dem Herausgeber Herrn Dr. L. Claßen für viele wertvolle Hinweise und Anregungen verpflichtet. Dem Lektor des Verlages Technik, Herrn J. Reichenbach, sowie vielen Kollegen des Bereichs Elektronikentwicklung des ZFT/KEAW, die durch vielfältige Diskussionen und Unterstützung einen hohen Anteil am Entstehen des vorliegenden Bandes haben, gilt ebenfalls mein Dank.

Heinz Brennenstuhl

Inhaltsverzeichnis

1. Einleitung

Die Mikroelektronik ist zu einer Schlüsseltechnologie für den wissenschaftlich-technischen Fortschritt geworden. Die Ursache dafür ist die Entwicklung der Mikroprozessortechnik zu Beginn der 70er Jahre. Sie ermöglicht es, mit "intelligenten" Mikroprozessorbausteinen leistungsfähige Mikrorechner aufzubauen und sie mit hoher ökonomischer Effektivität in breitem Umfang einzusetzen.
Das Erscheinen der Mikroprozessoren hatte einen umfassenden Einfluß auf die Entwicklung von mikroelektronischen Bauelementen und peripheren Geräten, die, zusammen mit dem Mikroprozessor, erst zum Mikrorechner werden.
Neben diesen Bauelementen und Geräten, der Hardware des Mikrorechners, wird seine Leistungsfähigkeit im wesentlichen durch seine Programme, d.h. seine Software, bestimmt. Sie enthält die vom Mikroprozessor auszuführenden Befehle, d.h., mit der Software wird die universell einsetzbare Hardware des Mikrorechners für die Lösung spezieller Anwenderprobleme angepaßt.
Moderne Mikroprozessorsysteme, wie das 16-Bit-Mikroprozessorsystem U8000 aus dem VEB Mikroelektronik "Karl Marx" Erfurt, erreichen die Rechenleistung von Kleinrechnern, in einigen Kennwerten entsprechen sie der von einigen Jahren üblichen Leistungsfähigkeit der Großrechentechnik.
Das U8000-Mikroprozessorsystem besteht gegenwärtig aus der zentralen Verarbeitungseinheit **U8000-CPU** (central processing unit) und dem Speicherverwaltungsbaustein **U8010-MMU** (memory management unit). Der Anschluß der Peripherie erfolgt über die Peripheriebausteine des 8-Bit-Mikroprozessorsystems U880. Die Leistungsfähigkeit des Mikroprozessorsystems U8000 entspricht dem international üblichen 16-Bit-Mikroprozessorniveau und erlaubt die Lösung komplizierter Aufgaben in unterschiedlichsten Anwendungsgebieten.
Das 16-Bit-Mikroprozessorsystem U8000 stellt eine neue Generation von Mikrorechnerschaltkreisen dar. Ihre Anwendungsmöglichkeiten gehen auf Grund ihrer wesentlich gestiegenen Rechenleistung über die der bisher verfügbaren 8-Bit-Mikrorechner hinaus. Die gestiegene Leistungsfähigkeit zeigt sich nicht nur in der größeren Verarbeitungsbreite des Prozessors, sondern auch in neuen Systemmerkmalen, wie z.B.

- komfortables Unterbrechungssystem
- Speicherverwaltung
- Speicherschutz
- Nutzung großer Speicherressourcen
- Multiprozessorarchitekturen
- Festkommaarithmetik.

Damit erschließen die 16-Bit-Mikroprozessoren der Automatisierungs- und der Rechentechnik neue Möglichkeiten.
Die zu automatisierenden Anlagen werden immer komplexer und müssen aus Gründen der Wirtschaftlichkeit und des Umweltschutzes mit präzis geführten Parametern betrieben werden. Die Lösung dieser Aufgaben in der Automatisierungstechnik erfordert komplexe Rechnerstrukturen, die auf der Basis von leistungsfähigen 16-Bit-Mikroprozessoren aufgebaut werden können.
In der Rechentechnik kann die Verarbeitung umfangreicher Datenmengen durch leistungsfähige Mikrorechner an ihrem Entstehungsort erfolgen. Damit ergeben sich auf diesem Gebiet neue Möglichkeiten des Aufbaus von dezentral organisierten Rechnerstrukturen.

2. Aufbau und Funktion des 16-Bit-Mikroprozessors U8000

2.1. Überblick

Die U8000-CPU wird in zwei Versionen angeboten, der 48poligen segmentiert arbeitenden U8001-CPU und der 40poligen nichtsegmentiert arbeitenden U8002-CPU. In beiden Schaltkreisen ist der gleiche Chip eingesetzt. Nach einem speziellen Testverfahren wird entschieden, ob der Siliziumchip als U8001-CPU oder U8002-CPU im Produktionsablauf konfektioniert wird. Deshalb wird der Begriff "U8000-CPU" verwandt, wenn es sich um gleiche Eigenschaften des Schaltkreises handelt.
Das Mikroprozessorsystem U8000 wird in N-Kanal-MOS-Technologie hergestellt. Es ist in seiner Standardversion in einem Temperaturbereich von 0...70 °C einsetzbar und seine Verarbeitungsgeschwindigkeit beträgt maximal 4 MHz.

U8001-CPU
Die U8001-CPU ist ein leistungsfähiger 16-Bit-Mikroprozessor in einem 48poligen DIL-Gehäuse (dual in line) /1/. Sie besitzt sechzehn allgemein verwendbare 16-Bit-Register. Ihr Befehlssatz besteht aus 110 verschiedenen Befehlen. Durch acht verschiedene Adressierungsarten ist eine Spezifizierung der einzelnen Befehle möglich. Der Mikroprozessor kann mehrere Datentypen, angefangen von Bit bis zu 32-Bit-Wörtern, bearbeiten. Insbesondere auf dem Gebiet der Zeichenkettenverarbeitung besitzt er, bezogen auf den Befehlssatz, eine hohe Leistungsfähigkeit.
Die U8001-CPU enthält intern keine Speicherverwaltung. Zum Aufbau einer Speicherverwaltung sowie eines effektiven Speicherschutzes ist der Speicherverwaltungsbaustein U8010-MMU im Mikroprozessorsystem U8000 integriert. Der Prozessor erlaubt unter Nutzung dieser Bausteine eine effektive Adressierung des verfügbaren Speicherbereichs bei der U8001-CPU von 8 MByte. Dieser Speicherbereich wird in 128 Segmente mit einem maximalen Speichervolumen von jeweils 64 KByte eingeteilt. Diese Version des U8000-Mikroprozessors wird deshalb auch als die segmentiert arbeitende CPU des U8000-Systems bezeichnet.
Zur Verbesserung der Speichereffizienz und der Abarbeitungsgeschwindigkeit des Prozessors wurden in bezug auf die Speicheradressierung zwei Betriebsarten implementiert, der segmentierte und der nichtsegmentierte Mode. Die U8001-CPU kann im segmentierten Mode den adressierbaren Speicherbereich von 8 MByte ausnutzen. Im nichtsegmentierten Mode steht ihr jeweils nur ein Speichersegment von 64 KByte zur Verfügung, der Zugriff auf ein anderes Segment ist nur nach dem Übergang in den segmentierten Mode möglich. Zur Strukturierung der Software in System- und Anwenderprogramme wurden zwei unterschiedliche Betriebsarten der U8000-CPU implementiert, der System- und Normalmode.

Neben diesen Fähigkeiten besitzt der Mikroprozessor U8000 ein leistungsfähiges Unterbrechungssystem. Programmunterbrechungen im U8000-System treten auf als:

Interrupt - Durch spezielle Hardwarebedingungen ausgelöste Unterbrechung. Ein Interrupt ist eine zeitabhängige nicht reproduzierbare Unterbrechung der Programmabarbeitung durch eine außerhalb des Mikroprozessors befindliche Interruptquelle.

Trap - Durch spezielle Softwarebedingungen hervorgerufene Unterbrechung. Ein Trap ist eine zeitunabhängige reproduzierbare Unterbrechung der Programmabarbeitung des Mikroprozessors. Traps können sowohl im Mikroprozessor als auch außerhalb des Mikroprozessors durch die U8010-MMU generiert werden.

Das U8000-System besitzt außerdem spezielle Voraussetzungen zum Aufbau von Multiprozessorsystemen sowie zur Erweiterung der Leistungsfähigkeit durch den Anschluß von EPU-Bausteinen (extended processing unit) als Co-Prozessoren. EPU-Bausteine erweitern die Leistungsfähigkeit eines Mikrorechnersystems durch besondere Fähigkeiten auf speziellen Gebieten, wie z.B. die Verarbeitung von Gleitkommaoperanden.

U8002-CPU
Für die U8002-CPU treffen mit einer Ausnahme auch die Erläuterungen zur U8001-CPU zu. Sie kann nur 64-KByte-Speicher adressieren.
Da der Speicherbereich von 64 KByte für viele Anwendungsfälle ausreicht, besitzt diese CPU ebenfalls eine hohe Bedeutung in der Mikrorechentechnik.

2.2. Hardwareorientierte Funktionen

Für die effektive Programmierung von Mikroprozessoren in der entsprechenden Assemblersprache sind einige Hardwaregrundkenntnisse erforderlich. Außerdem ist die Programmentwicklung in vielen Fällen mit der Hardwareentwicklung eng gekoppelt.
Im Bild 2.1 wird der interne Aufbau des U8000-Mikroprozessors dargestellt.

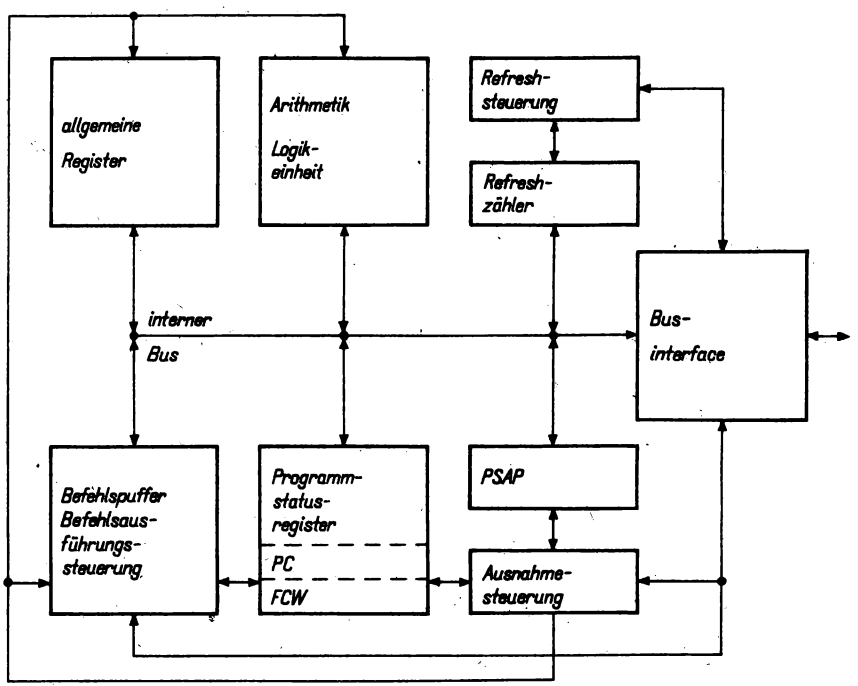


Bild 2.1. U8000-CPU Blockstruktur

Der Programmzähler dient zur Adressierung der abzuarbeitenden Befehle. Jeder Befehl wird in das Befehlsregister geladen und dekodiert. Die Dekodierung der Befehle erfolgt durch eine optimierte Logikschaltung.
Die Operanden werden den Registern oder der ALU (arithmetic logic unit) über einen internen 16-Bit-Daten- und ein 16-Bit-Adreßbus zur Verfügung gestellt.

2.2.1. Signal- und Pinbeschreibung

Das Signalspiel des U8000-Prozessors wird von einem einheitlichen Systemtakt gesteuert. Im Bild 2.2 ist das Zeitverhalten der U8000-CPU mit den drei wichtigsten Zeiteinheiten dargestellt.

Takt - Die Taktfrequenz wird durch einen externen Taktgeber vorgegeben, sie ist identisch mit dem CPU-Takt. Ein Taktzyklus (T-State) beginnt mit der steigenden Flanke des Taktes.

Bustransaktion - Sie umfaßt einen Datentransfer auf dem Bus, beginnend mit der fallenden Flanke von AS (address strobe) bis zur steigenden Flanke von DS (data strobe).

Maschinenzyklus - Ein Maschinenzyklus beginnt mit der fallenden Flanke von AS. Er beginnt stets mit einer Bustransaktion und kann sich befehlsabhängig bis über ihr Ende ausdehnen. Ein Maschinenzyklus setzt sich aus T_1 bis T_n Taktzyklen zusammen. Ein Befehl kann aus mehreren Maschinenzyklen bestehen, die Anzahl der Taktzyklen ist in Abhängigkeit von der Adressierungsart für jeden Befehl dem Befehlssatz (s. Abschnitt 3.5.) zu entnehmen.

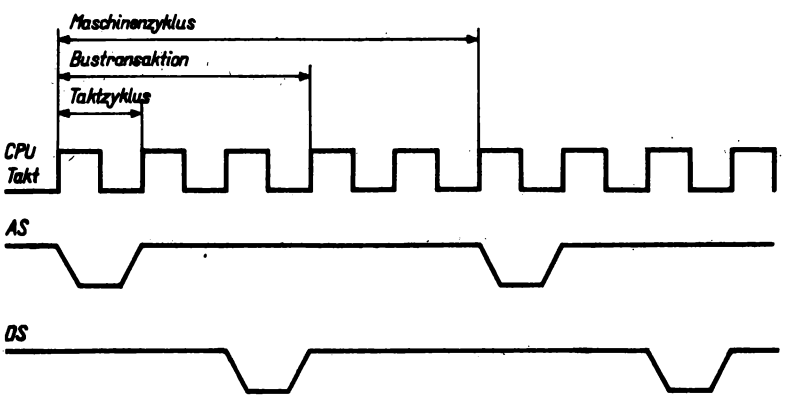


Bild 2.2. Zeitverhalten der wichtigsten U8000-Maschinenzyklen

Die 48polige U8001-CPU besitzt gegenüber der 40poligen U8002-CPU zusätzlich sieben Pins für die Segmentnummer und einen Pin für den Segmenttrap. Der Segmenttrap wird bei einem fehlerhaften Speicherzugriff von der U8010-MMU ausgelöst.
Zur Verbesserung der Übersichtlichkeit wurden im Bild 2.3 die Pins logisch gegliedert angegeben.

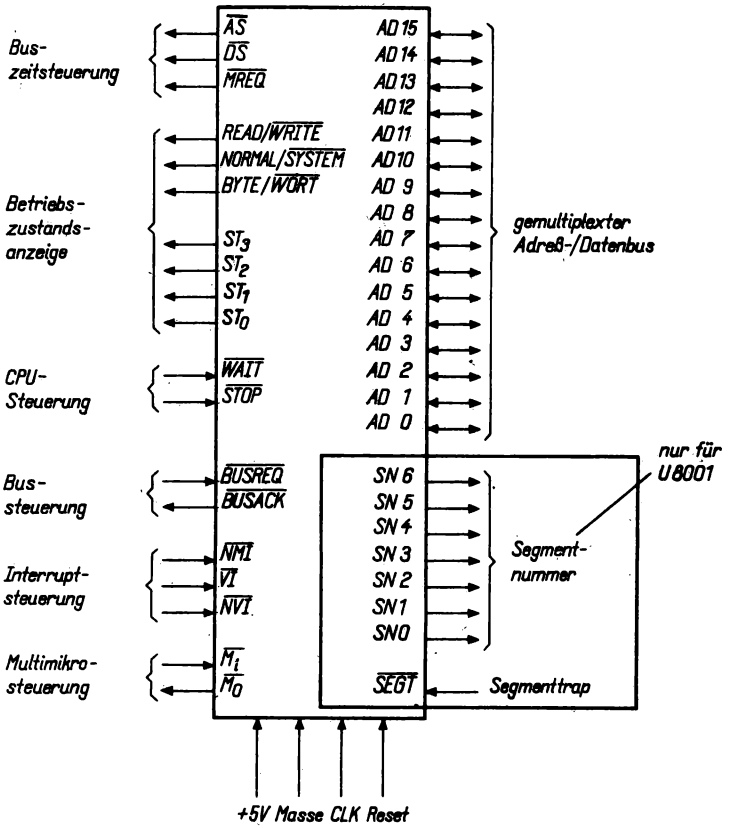


Bild 2.3. U8000-Pinbelegung

Bus-Zeitsteuerung - Die Pins AS (address strobe), DS (data strobe) und MREQ (memory request) zeigen den Beginn einer Busaktivität und die jeweilige Belegung des Busses (Adressen oder Daten) an. Das Signal AS ist stets im ersten, das Signal DS im zweiten Maschinenzyklus aktiv. Durch das Einfügen von Wait-Zyklen kann zur Anpassung an langsame Speicherschaltkreise das Signal DS verlängert werden. Das Signal MREQ wird zusätzlich zur Steuerung des Speicher- und Ein-/Ausgabezugriffs benutzt. Ist das Signal nicht aktiv (low) wird ein Speicherverkehr ausgeführt, ist es aktiv (high) zeigt es einen Ein-/Ausgabeverkehr an.

Status - Die Statussignale spiegeln den genauen Buszustand des Mikroprozessors wider. Am R/W-Pin (read/write) kann erkannt werden, ob es sich bei der laufenden Operation um eine Lese- oder Schreiboperation der CPU handelt. Ist das am R/W-Pin ausgegebene Signal aktiv (low), gibt die CPU ein Wort oder Byte aus, ist es nicht aktiv (high), liest sie ein Wort oder Byte.
Das N/S-Pin (normal mode / system mode) zeigt, ob sich die CPU in der Betriebsart System- oder Normalmode befindet. Ist das Signal aktiv (low), zeigt es die Betriebsart Systemmode an, ist es nicht aktiv (high), arbeitet der Mikroprozessor in der Betriebsart Normalmode.

Mit dem B/W-Pin (byte/word) wird die Art des Speicherzugriffs angezeigt. Ist das Signal B/W aktiv (low), erfolgt Wortzugriff, ist es nicht aktiv (high) ein Bytezugriff.
In der Tafel 2.1 ist die Belegung der Statusleitungen ST_0 - ST_3 wiedergegeben. Mit Hilfe der Statussignale kann zwischen Stack-, Daten- und Befehlszugriffen der CPU unterschieden werden. Diese Zugriffe können unter Einbeziehung des N/S-Pins nochmals unterteilt werden, damit besteht die Möglichkeit auf sechs verschiedenen Adreßräume, d.h. physisch getrennte Speicherbereiche, zuzugreifen.

Systemmode	Normalmode
Programmbereich	Programmbereich
Datenbereich	Datenbereich
Stackbereich	Stackbereich

Für die U8001-CPU ergibt sich bei Ausnutzung dieser Signale ein adressierbarer Speicherbereich von 48 MByte und für die U8002-CPU von 384 KByte. Diese Werte besitzen nur theoretischen Wert, ein 8 MByte großer Adreßraum, z.B. für den Systemstack der U8001-CPU wird sicher niemals benötigt. Eine Trennung des Daten- und Stackbereichs einerseits und des Programmbereichs andererseits ist jedoch als problemlose Speichererweiterung für die U8002-CPU nützlich.

Tafel 2.1. U8000-Statuskodierung

ST_3 - ST_0	Bedeutung
0000	interne Operation
0001	Refresh
0010	Ein-/Ausgabeoperation
0011	Spezial-Ein-/Ausgabeoperation
0100	Segmenttrap Annahmezyklus
0101	nichtmaskierbarer Interrupt-Annahmezyklus
0110	nichtvektorisierter Interrupt-Annahmezyklus
0111	vektorisierter Interrupt-Annahmezyklus
1000	Datenrequest
1001	Stackrequest
1010	Datenrequest (EPU)
1011	Stackrequest (EPU)
1100	Befehlsrequest
1101	Befehlsrequest, erstes Wort
1110	EPU-Transfer
1111	reserviert

CPU-Steuerung - Mit dem Wait-Signal kann der Mikroprozessor zeitlich an langsamere Speicher und Ein-/Ausgabegeräte angepaßt werden. Das Stop-Signal verursacht einen internen Halt der CPU-Operationen. Es dient zur Realisierung eines zyklusweisen Einzelschrittbetriebes und zur Synchronisation von EPU-Bausteinen durch kurzzeitiges Stoppen der CPU.

Bussteuerung - Mit dem Signal BUSREQ (bus request) können externe Bausteine den Bus von der CPU anfordern. Die CPU antwortet auf diese Anforderung mit dem Signal BUSACK (bus acknowlegde) und stellt damit den Bus zur Verfügung.

Interrupts - Die U8000-CPU kann auf die folgenden drei verschiedenen Interruptquellen reagieren:

NMI (nichtmaskierbarer Interrupt)
VI (maskierbarer vektorisierter Interrupt)
NVI (maskierbarer nichtvektorisierter Interrupt).

Multimikro-Steuerung - Die Signale M_1 und M_0 dienen zur Steuerung mehrerer Prozessoren durch variable Zuweisung der Bushoheit an einen Prozessor des Systems. Das M_1-Signal (M_1-Pin) dient zur Signalisierung der Anforderung eines anderen Prozessors auf die Busprioritat. Mit dem M_0-Signal (M_0-Pin) wird eine Anforderung auf die Busprioritat von der CPU ausgegeben.

Adreß-/Datenbus - Die 16 Adreß-/Datenpins werden zur Vermittlung von Adressen und Daten benutzt. Über sie wird sowohl der Speicheradreßraum als auch der Ein-/Ausgabeadreßraum angesprochen. Im segmentierten Mode des Mikroprozessors U8001 wird die 16-Bit-Adresse als Offset bezeichnet.

Segmentnummer (nur U8001) - Mit den sieben Segmentpins können 128 Segmentnummern an die MMU-Schaltkreise übertragen werden. Die Segmentnummer wird stets vor dem Offset ausgegeben, um die durch die MMU-Operationen auftretende Verzögerung auszugleichen.

Segmenttrap (nur U8001) - Über das SEGT-Pin wird von der U8010-MMU ein fehlerhafter Speicherzugriff an die CPU signalisiert. Er führt zur Abarbeitung einer Segmenttraproutine, in der die Fehlerursache ermittelt werden kann. In den Abschnitten 4 und 5 wird die Funktion und die Programmierung der U8010-MMU erläutert.

Prozessorsteuerung - Neben den Pins mit den Signalen zur Informationsverarbeitung benötigt der Mikroprozessor noch das CLK-, RESET-, GND- und +5-V-Pin zur Sicherung seiner Funktion. An das CLK-Pin wird ein synchroner externer Einphasentakt im Bereich von 500 kHz bis 4 MHz als Arbeitsfrequenz gelegt.
Mit der Aktivierung des RESET-Pins wird die CPU in einen inaktiven Zustand gebracht. Nach drei Taktzyklen beginnt sie mit dem Einlesen von Statusinformationen auf der physischen Speicheradresse Null. In diesen Statusinformationen sind alle zum Abarbeiten des Programms erforderlichen Informationen abgelegt.
Über das +5-V-Pin erhält der Mikroprozessor seine Versorgungsspannung von +5-V, über das GND-Pin das erforderliche Massepotential.

2.2.2. CPU-Betriebszustände

Die U8000-CPU kann drei Betriebszustände einnehmen, den Arbeitszustand, den Stop-/Refreshzustand und den Zustand der Bustrennung.
Das Bild 2.4 zeigt einen Zustandsgrafen, aus dem die möglichen Übergänge zwischen den drei Zuständen entnommen werden können.

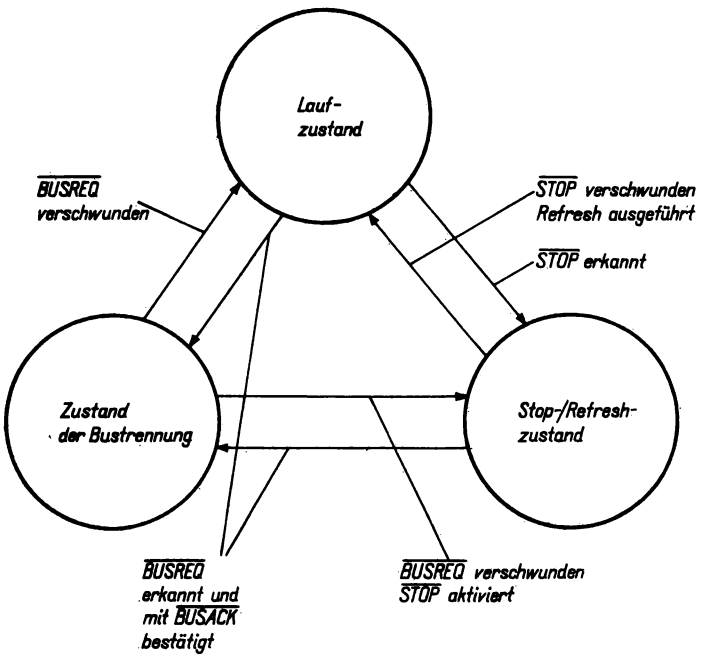


Bild 2.4. U8000-Zustandsgraf

Arbeitszustand - Im Arbeitszustand arbeitet der Mikroprozessor Programme ab oder behandelt Unterbrechungsanforderungen. Der Zustand wird nur verlassen, wenn eine der folgenden drei Bedingungen auftritt:

- Die Refreshlogik führt einen periodischen Refreshvorgang aus, die CPU geht kurzzeitig in den Stop-/Refreshzustand.
- Eine externe Stopbedingung führt den Mikroprozessor über seinen Stopeingang in den Stop-/Refreshzustand.
- Eine externe Busanforderung bringt die CPU in den Zustand der Bustrennung.

Stop-/Refreshzustand - Der Stop-/Refreshzustand wird eingenommen, wenn entweder das Stopsignal extern gesetzt wurde oder der interne Refreshzähler die Ausführung eines Refreshzyklus erzwingt.
Im Stop-/Refreshzustand generiert die CPU kontinuierlich Refreshzyklen, d.h., sie führt keine weiteren Funktionen aus. Der Stop-/Refreshzustand gibt externen Geräten die Möglichkeit, CPU-Operationen zu unterbrechen. Der Stopeingang des Mikroprozessors kann für die Ausführung von Einzelschrittoperationen und zur Synchronisation der Arbeit mit EPU-Co-Pro-

zessoren benutzt werden.
Dieser Zustand kann durch Verschwinden der internen bzw. externen Bedingung für den Stop-/Refreshzustand oder durch Übergang der CPU in den Zustand der Bustrennung verlassen werden.

Zustand der Bustrennung - Im Zustand der Bustrennung trennt sich der Mikroprozessor vom externen Systembus ab. Ein anderer Schaltkreis, z.B. ein DMA (direct memory access) übernimmt die Bushoheit. Etwa 2/3 der CPU-Signaleingänge- oder -ausgänge sind im Tri-State-Zustand, d.h., sie sind hochohmig. Die Einnahme dieses Zustandes ist sowohl aus dem Arbeitszustand als auch aus dem Stop-/Refreshzustand möglich. Der Mikroprozessor muß dazu von einem externen Bauelement, z.B. einem DMA-Schaltkreis das Signal BUSREQ empfangen. Er sendet daraufhin das Signal BUSACK und teilt dem externen Gerät damit seine Bereitschaft zur Aufgabe der Bushoheit mit. Anschließend geht der Mikroprozessor in den Tri-State-Zustand.
Der Zustand der Bustrennung wird auch zur Steuerung der Arbeit in Multiprozessorsystemen genutzt. Mehrere Mikroprozessoren sind dabei über eine Zusatzelektronik in einer Daisy-Chain-Prioritätskette miteinander verbunden. Über die Signale BUSREQ und BUSACK erfolgt, ähnlich der Interruptkaskadierung, die Vergabe der Bushoheit an den jeweiligen Prozessor.
Der Mikroprozessor verläßt den Zustand, wenn die externen BUSREQ-Bedingungen nicht mehr existieren. Der Zustand der Bustrennung besitzt die höchste Priorität im System und kann nur vom Resetsignal unterbrochen werden.

2.2.3. Befehlsausführung

Der U8000-Mikroprozessor kann zur Verkürzung der Befehlsabarbeitung die Ausführung des vorangegangenen Befehls mit dem Befehlsholezyklus des folgenden Befehls überlappen. Diese Fähigkeit wird als Instruction-Look-Ahead bezeichnet. Im Bild 2.5 sind am Beispiel von mehreren Ladebefehlen einige Befehlsfolgen dargestellt. Der nächste Befehl wird parallel zur Ausführung des vorigen Befehls eingelesen. Damit wird die Abarbeitungszeit wesentlich verkürzt und die Voraussetzung für einen hohen Datendurchsatz der U8000-CPU geschaffen.

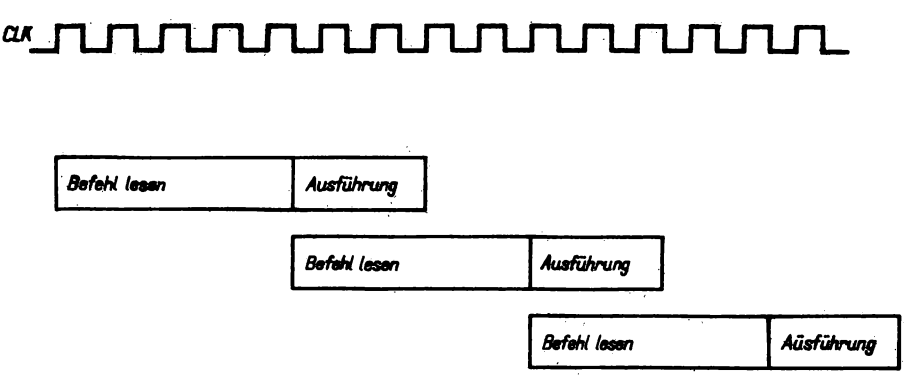


Bild 2.5. Instruction-Look-Ahead

Die Befehlsausführung erfolgt im U8000-Mikroprozessorsystem in zwei Schritten:

- Lesen des aus einem oder mehreren Wörtern bestehenden Befehls aus dem Programmspeicher; der Befehl wird durch den Inhalt des Befehlszählers adressiert.
- Ausführung des Befehls, Verändern des Befehlszählers und der Flags - dem abgearbeiteten Befehl entsprechend - im Flag- und Controlwort.

Die Ausführung der Befehle im Arbeitszustand der CPU wird durch spezielle Register des U8000-Mikroprozessors, den Befehlszähler oder PC (program counter) und die in einem Flag- und Controlwortregister (FCW) zusammengefaßten Flags (s. Abschnitt 2.3.2.) gesteuert.
Der Befehlszähler gibt die Adresse des abzuarbeitenden Befehls aus. Er wird automatisch auf den nächsten Befehl im Speicheradreßraum gestellt. Bei einem Jump-, Call-, oder Return-Befehl wird er den Erfordernissen des Programms entsprechend gestellt.
Die Flags sind einzelne Bits im Low-Byte des Flag- und Controlworts der U8000-CPU. Sie werden bei Arithmetik-, Logik-, Rotations- und Shiftbefehlen durch das Ergebnis der Operation gesetzt und können dann durch bedingte Programmsteuerbefehle zur Programmverzweigung genutzt werden. Die vom 8-Bit-Mikroprozessorsystem U880 her bekannten Flags, wie z.B. das Carry- und das Zero-Flag, sind ebenfalls im Flagbyte des FCW enthalten.
Das High-Byte im Flag- und Controlwort enthält Steuerbits, mit denen die CPU-Arbeitsweise und die Maskierung der Interrupts festgelegt wird. Für die Befehlsausführung ist die im Flag- und Controlwort festzulegende Arbeitsweise des Prozessors - der System- oder Normalmode - von Bedeutung. Für den Normalmode (s. Abschnitt 2.3.3.) ist die Ausführung einiger Befehle nicht zugelassen.

2.3. Softwareorientierte Funktionen

Die softwareorientierten Funktionen der U8000-CPU sind durch die im Befehlssatz des Mikroprozessors enthaltenen Befehle zu beeinflussen. Sie beinhalten die Verwendung der Register des Mikroprozessors, die Betriebsarten, das Unterbrechungssystem und das Ein-/Ausgabesystem.

2.3.1. Allgemeine Register

Die Leistungsfähigkeit der U8000-CPU wird im wesentlichen durch ihren flexiblen, universellen Registersatz bestimmt. Er besteht aus 16 allgemein anwendbaren 16-Bit-Registern, die als als Akkumulatoren, Adreßregister, Basisadreßregister oder Indexregister genutzt werden können.

Anwendung der allgemeinen Register
Register - Ein Register ist im U8000-System eine prozessorinterne 16 Bit breite Speichereinheit.

Akkumulator - Als Akkumulator wird ein Register bezeichnet, in dem arithmetische oder logische Operationen ausgeführt werden.

Indexregister - Ein Indexregister enthält einen Index, d.h. eine Adresse zur Unterscheidung von Positionen gleichartiger Größen einer Tabelle oder eines Felds. Die Adresse der Tabelle oder des Felds wird durch eine Basisadresse vorgegeben. Die Adressierung in der Tabelle oder im Feld erfolgt mit dem Index. Steht, z. B. bei der Behandlung von tabellarisch geordneten Daten, in einem Indexregister eine 27, so adressiert es bei einer mit 0 beginnenden Tabelle die 28. Position der Tabelle.

Adreßregister - Ein Adreßregister enthält eine Adresse aus dem Speicher- oder dem Ein-/Ausgabeadreßraum des Mikrorechnersystems. Arbeitet die CPU im segmentierten Mode sind die Speicheradreßregister 32 Bit breit, im nichtsegmentierten Mode sind sie 16 Bit breit. Die Ein-/Ausgabeadreßregister sind stets 16 Bit breit.

Basisadreßregister - Ein Basisadreßregister enthält eine Adresse aus dem Speicheradreßraum. Eine Basisadresse ist eine konstante Adresse, zu der variable Adressen addiert werden. Die Zieladresse wird im U8000-Mikroprozessor gebildet, indem zum Inhalt des Basisadreßregisters ein Offset oder der Inhalt eines Indexregisters addiert wird. Ein Offset ist eine Adresse, die zu einer Basisadresse addiert wird.

Datentypen in den allgemeinen Registern
Die Register der U8000-CPU können als Byte-, Wort-, Langwort- oder Vierfachwortregister benutzt werden. In den Registern des U8000-Mikroprozessors können die nachfolgend aufgeführten Datentypen bearbeitet werden. Die Datentypen Byte und Wort sind mit oder ohne Vorzeichen einsetzbar (s. Abschnitt 3.1.2.).

Bit (1 Bit) - Die Adresse eines einzelnen Bits wird - in der noch zu behandelnden - Assemblersprachnotation des U8000-Mikroprozessors - nach der Registerbezeichnung als Zahl angegeben. Bitadressierung kann bei Byte- und Wortregistern vorgenommen werden.

In dem Befehl

```
BIT  R6, #9
```

wird das Bit 9 des Registers R6 auf seinen Inhalt getestet, ist es 0, wird das Zero-Flag 1 gesetzt.

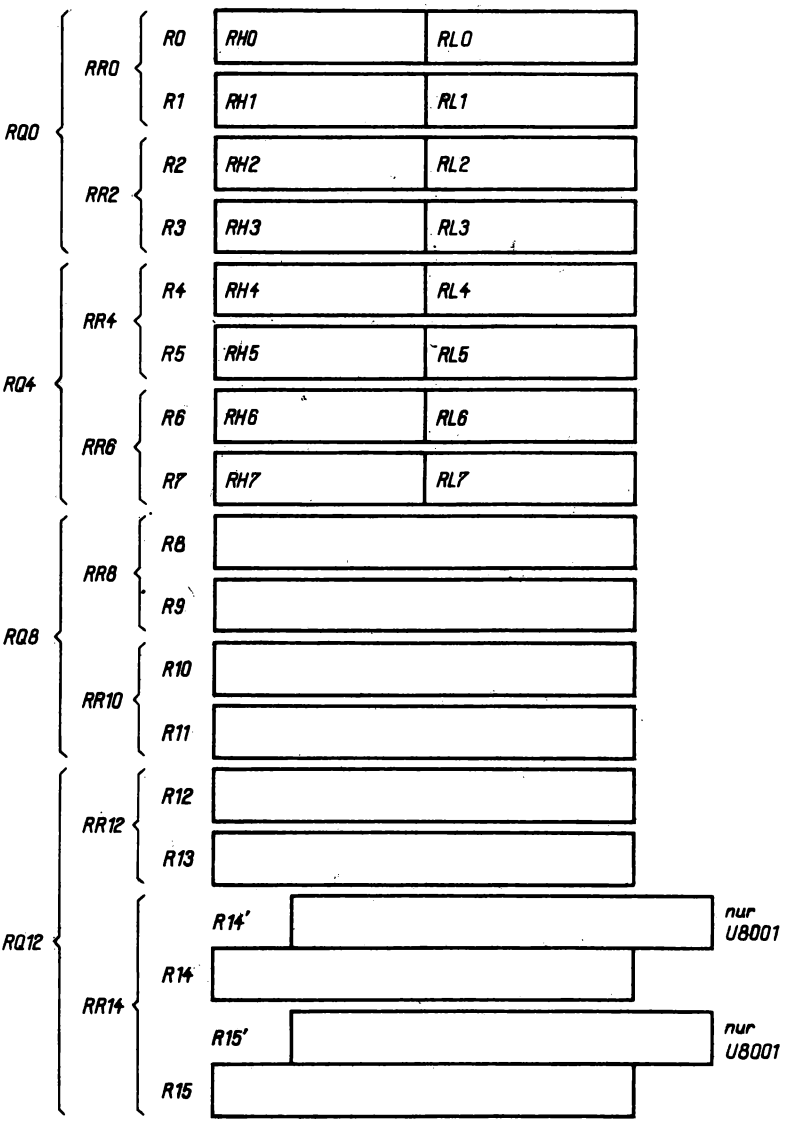


Bild 2.6. Allgemeine Register der U8001- und U8002-CPU

Byte (8 Bit) - Die ersten acht Wortregister (R0 - R7) können als Byteregister genutzt werden. Sie werden dann, z.B. im Wortregister R0, als Byteregister RL0 für den Low-Teil und als Byteregister RH0 für den High-Teil des im Register gespeicherten Werts bezeichnet. Die Byteregister sind wie alle

Register als Akkumulatoren nutzbar.
Der Befehl

```
ANDB RL2, RH5
```

bildet das Ergebnis der logischen UND-Operation im Byteregister RL2 ab und setzt in Abhängigkeit vom Ergebnis die entsprechenden Flags.

Wort (16 Bit) - Die sechzehn 16-Bit-Wortregister (R0 - R15) dienen der Speicherung und Manipulation von 16-Bit-Daten und Adressen und stellen den eigentlichen Basisregistersatz des U8000-Mikroprozessors dar. Sie sind als Akkumulatoren, Adreßregister und mit Ausnahme des Registers R0 als Indexregister nutzbar.
Der Befehl

```
LD   R6, R7
```

lädt den Inhalt des Registers R7 in R6.

Langwort (32 Bit) - Für die Verarbeitung von Langwörtern ist es möglich, acht 32-Bit-Register (RR0 - RR14) zu bilden. Diese Register sind, neben der Daten- und Adreßspeicherung, als Adreßregister und, ausgenommen das Register RR0, als Indexregister nutzbar.
Der Befehl

```
LDL  RR0, RR4
```

lädt den Inhalt des Langwortregisters RR4 in das Langwortregister RR0.

Vierfachwort (64 Bit) - Vierfachwörter können durch die Gruppierung der 16 Register in vier 64-Bit-Register (RQ0 - RQ12) abgelegt werden. Diese Register werden, insbesondere als Ergebnisregister, bei der Ausführung von arithmetischen Operationen genutzt.

Stackpointer
Bei der Abarbeitung von Programmen benötigt der U8000-Mikroprozessor einen besonderen Speicherbereich, indem er bei Unterprogrammaufrufen die Rückkehradressen und bei Programmunterbrechungen durch Interrupts oder Traps alle zur Fortsetzung der Programmabarbeitung erforderlichen Informationen zwischenspeichert.
Dieser Speicherbereich wird als Stackspeicherbereich bezeichnet. Der Stackbereich ist nach dem LIFO-Prinzip (last in first out) organisiert, die zuletzt abgespeicherte Information wird zuerst wieder ausgelesen. Der Stackspeicherbereich wird über das Stackregister der U8000-CPU adressiert. Es enthält eine als Stackpointer bezeichnete Adresse, d.h. einen Zeiger auf den Stackspeicherbereich. Der Stackpointer zeigt immer auf des zuletzt im Stackspeicherbereich eingetragene Wort bzw. nach der Initialisierung auf die höchste Adresse des Stackspeicherbereichs. Bei einem Eintrag wird er dekrementiert, beim Auslesen der gespeicherten Information wird er inkrementiert.
Die U8000-CPU besitzt zwei Stackregister, ein Stackregister für den Systemstackspeicherbereich und ein Stackregister für den Normalstackspeicherbereich. Damit ist gesichert, daß zwei völlig getrennte Speicherräume zur Verfügung stehen und eine gegenseitige Beeinflussung kaum auftreten

kann (s. Abschnitt 2.3.3.).
Im Bild 2.7 ist der Stackspeicherbereich nach einem Unterprogrammaufruf dargestellt. Beim Aufruf eines Unterprogramms, d.h. bei einem CALL- oder CALR-Befehl wird er zur Abspeicherung der Rückkehradresse dekrementiert. Nach Beendigung des Unterpogramms wird er durch den RET-Befehl wieder inkrementiert.

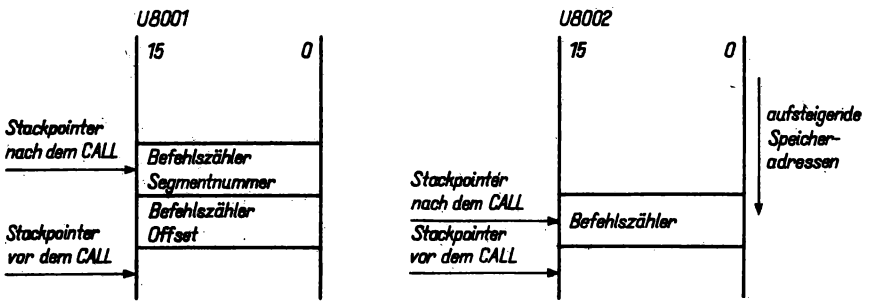


Bild 2.7. Stackpointeroperation bei einem Unterprogrammaufruf

Der **Normalstackpointer** steht im Normalmode in den Registern R14/R15 (segmentierte Version) oder R15 (nichtsegmentierte Version). Er wird bei der Abarbeitung von Unterprogrammaufrufen sowie zur kurzzeitigen Speicherung von Registerwerten genutzt. Beim Auftreten von Interrupts oder Traps geht die U8000-CPU in den Systemmode über und tauscht den Normalstackpointer gegen den Systemstackpointer aus. Nach Beendigung der Unterbrechung erfolgt wieder der Übergang in den Normalmode und damit die Aktivierung des Normalstackpointer. Im Normalmode kann auf den Systemstackpointer nicht zugegriffen werden.

Der **Systemstackpointer** dient im Systemmode sowohl für die Abarbeitung von Interrupts und Traps als auch für die Organisation der Unterprogrammarbeit und der Zwischenspeicherung von Parametern durch PUSH- und POP-Befehle.
Der Systemstackpointer steht im Systemmode in den Registern R14/R15 (segmentierte Version) oder R15 (nichtsegmentierte Version). Auf den Normalstack kann im Systemmode zugegriffen werden, er ist in einem besonderen Register abgelegt, mit den LDCTL-Befehlen kann er in ein allgemeines Register geladen und ggf. modifiziert werden.

Neben diesen Basisfunktionen besteht zusätzlich die Möglichkeit, den Stackspeicherbereich zur Zwischenspeicherung von Parametern aus den allgemeinen Registern des U8000-Mikroprozessors zu nutzen. Das Einschreiben der Informationen in den Stackbereich erfolgt mit den PUSH-Befehlen. Die abgespeicherten Informationen können unter Beachtung des LIFO-Prinzips mit den POP-Befehlen wieder aus dem Stackspeicherbereich gelesen werden. Die Wortregister können, mit dem Register R2 beginnend, auch als Stackpointer für PUSH- und POP-Befehle eingesetzt werden.

Einschränkungen der Registernutzung
Bei der Nutzung der Register zur Adressierung von Operanden sind die nachfolgenden Einschränkungen zu beachten.

R0, RR0
Die Register R0 und RR0 können nicht als Index-, Basis- oder Adreßregister verwandt werden.
Das Register R0 wird intern bei der Abarbeitung von EPU-Befehlen verändert und enthält nach der Abarbeitung dieser Befehle einen undefinierten Wert.

R15, RR14, R15', RR14'
Diese Register enthalten den System- und Normalstackpointer und sind ausschließlich als Stackpointer nutzbar.

RH1, R1, RR0
Das Register RH1 wird intern bei der Nutzung von Blocksuchbefehlen (s. Abschnitt 3.4.7.) durch die CPU verändert. Es enthält nach Abarbeitung des Befehls einen aus der Befehlsabarbeitung resultierenden Wert. Die Register R1 und RR0 sind in ihrer Nutzung beim Gebrauch von Blocksuchbefehen dementsprechend eingeschränkt.

2.3.2. Steuerregister

Neben den allgemeinen Registern enthält die U8000-CPU noch einige Register mit speziellen Funktionen zur Steuerung des Mikroprozessors. Sie können im Systemmode durch die LDCTL-Befehle (s. Abschnitt 3.5.9.) initialisiert bzw. modifiziert werden. Im Bild 2.8 sind die Steuerregister der U8001- und der U8002-CPU dargestellt.

Programmstatusregister
Die Programmstatusregister bestehen aus dem Befehlszähler bzw. PC (program counter) und dem Flag- und Controlwort (FCW).
Im U8001-Prozessor bestehen die Programmstatusregister aus vier Wortregistern, ein Wort ist für zukünftige Anwendungen reserviert, der Befehlszähler benötigt zwei Wörter, das Flag- und Controlwortregister ein Wort.
Die Programmstatusregister im U8002-Prozessor bestehen aus zwei Wortregistern: in einem Wort ist der Befehlszähler, im zweiten das Flag- und Controlwortregister des Prozessors gespeichert.

Befehlszähler (PC) - Der Befehlszähler adressiert den abzuarbeitenden Befehl im Speicherbereich des Mikrorechners. Er wird bei der Befehlsabarbeitung automatisch auf den nächsten abzuarbeitenden Befehl gestellt. Im U8001-Prozessor besteht der Befehlszähler aus zwei 16-Bit-Registern, in ihnen steht die Segmentnummer und der Offset. Die Segmentnummer adressiert eines der maximal 128 adressierbaren 64-KByte-Segmente des U8001-Mikroprozessors, der Offset enthält die Adresse im 64-KByte-Segment. In der U8002-Version wird für die Adresse nur ein Wort benötigt.
Der Befehlszähler kann im Systemmode mit dem LDPS-Befehl direkt geladen werden.

Flag- und Controlwort (FCW) - Im Flag- und Controlwort ist der Prozessorstatus gespeichert. Im Low-Teil des Worts sind es sechs Statusflags, die von den bedingten Befehlen zur Programmverzweigung genutzt werden. Der

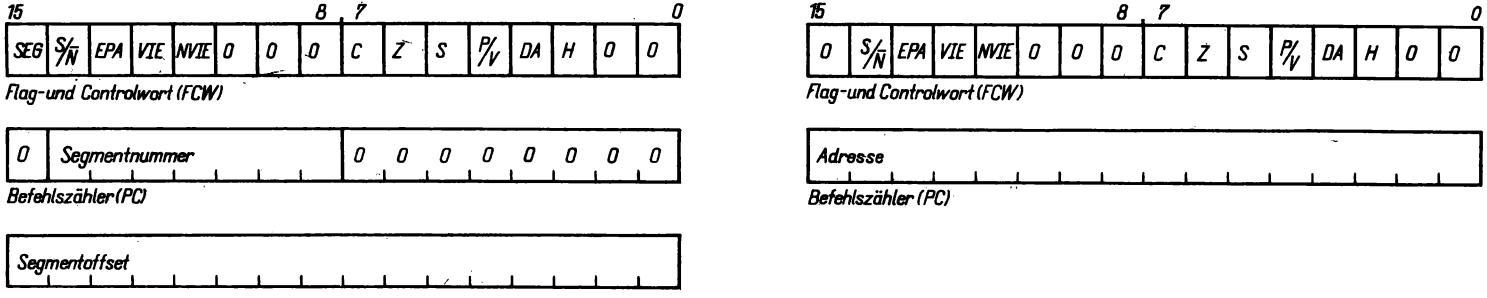
High-Teil des FCW enhält fünf Bits, die zur Steuerung des Prozessors dienen.
Im Bild 2.8 ist u. a. der Aufbau der Programmstatusregister wiedergegeben. Im Flag- und Controlwort sind die nachfolgend aufgeführten Flag- oder Statusbits enthalten:

- **Carry (C).** Das Bit zeigt einen Übertrag der höchsten Bitposition in einem als Akkumulator verwendeten Register bei der Nutzung von Arithmetik-, Rotations- oder Shiftbefehlen an.
- **Zero (Z).** Das Zero-Bit zeigt an, daß das Resultat einer Operation Null ist.
- **Sign (S).** Das Resultat einer Operation wird durch das Setzen des Sign-Bit als negative Zahl angezeigt.
- **Parity/Overflow (P/V).** Dieses Bit zeigt nach der Ausführung logischer Byteoperationen gerade oder ungerade Parität an.
  Nach arithmetischen Operationen signalisiert es einen Überlauf.
- **Decimal-Adjust (D).** Dieses Bit dient nach BCD-Arithmetik-Operationen zur Anzeige des Typs der ausgeführten Operation, Addition oder Subtraktion. Es wird bei einer nachfolgenden Korrektur der U8000-Binärarithmetik intern durch den DAB-Befehl zur Wiederherstellung des BCD-Werts genutzt.
- **Half-Carry (H).** Das Half-Carry Bit wird in der BCD-Arithmetik genutzt, es zeigt den Übertrag der Bitposition 3 auf die Bitposition 4 eines Byteregisters an.

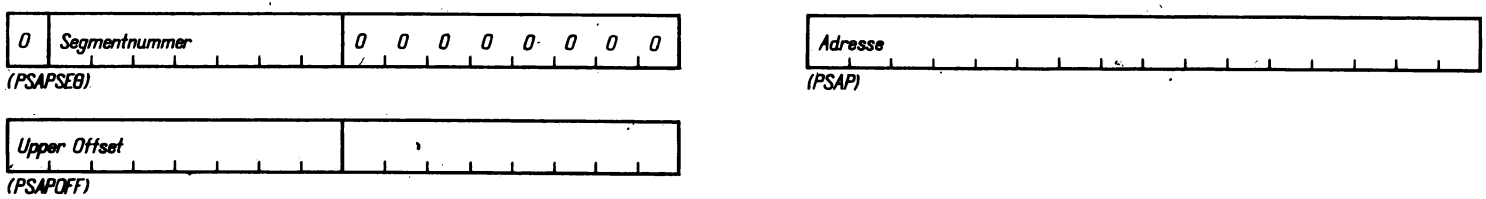
Neben diesen Statusbits werden einige Bits im Flag- und Controlwort zur Steuerung der U8000-CPU-Arbeitsweise genutzt. Mit ihrer Hilfe kann sie der Programmierer der Aufgabenstellung entsprechend beeinflussen:

- **Non Vectored Interrupt Enable (NVIE).** Ist dieses Bit gesetzt, akzeptiert die CPU nichtvektorisierte Interrupts.
- **Vectored Interrupt Enable (VIE).** Das Setzen des VIE-Bits veranlaßt die CPU zur Verarbeitung vektorisierter Interrupts.
- **Extended Processor Architecture (EPA).** Mit diesem Bit wird der CPU das Vorhandensein von EPU-Co-Prozessoren (extended processing unit) angezeigt, damit werden sogenannte "Extended Instructions", d.h. Befehle für diese Co-Prozessoren, von der CPU erkannt und führen nicht zu einem Trap. Ist dieses Bit 0, verursachen "Extended Instructions" einen Fehler, die CPU zeigt den Befehlskode als einen nicht ausführbaren Befehl an.
- **System-/Normalmode (S/N).** Ist dieses Bit gesetzt, arbeitet die U8000-CPU im Systemmode; ist es Null, arbeitet sie im Normalmode.
- **Segmentationmode (SEG).** Dieses Bit ist nur im U8001-Prozessor enthalten. Ist es gesetzt, arbeitet die U8001-CPU im segmentierten Mode, andernfalls arbeitet sie im nichtsegmentierten Mode.

Programmstatusregister



Programmstatusareapointer



Refreshregister

Bild 2.8. Steuerregister der U8000-CPU

Programmstatusareapointer (PSAPSEG, PSAPOFF / PSAP) - Der Programmstatusareapointer zeigt im Systemspeicher auf das Feld der Programmstatuswerte der Unterbrechungsroutinen des U8000-Mikrorechners, die Programmstatusarea. Beim Auftreten von Interrupts oder Traps werden die aktuellen Programmstatuswerte des unterbrochenen Programms in den Systemstackspeicherbereich des Mikroprozessors gespeichert. Anschließend werden die Programmstatuswerte der Unterbrechungsroutine aus der Programmstatusarea in die U8000-CPU geladen und zur Abarbeitung der entsprechenden Routinen genutzt.
Im U8001-Mikroprozessor benötigt der Programmstatusareapointer zwei Wortregister, die Segmentnummer (PSAPSEG) und den Offset (PSAPOFF).
Im U8002-Mikroprozessor belegt der Programmstatusareapointer ein Wortregister für die Adresse (PSAP) der Programmstatusarea.
Das Low-Byte des Programmstatusareapointer muß bei beiden Prozessorvarianten immer Null sein. Die Programmstatusarea beginnt demzufolge immer an einer 256-Byte-Grenze des Systemspeichers.
Das Register PSAPOFF ist mit dem Register PSAP identisch.

PSAPSEG	Programmstatusareapointer-Segmentadresse beim U8001
PSAPOFF	Programmstatusareapointer-Offset beim U8001
PSAP	Programmstatusareapointer beim U8002

Refreshregister (REFRESH) - Die U8000-CPU enthält einen programmierbaren Zähler, der zur automatischen Refresherzeugung für den Anschluß dynamischer RAM-Speicher genutzt werden kann. Das Refreshregister besteht aus einem 9-Bit-Zeilenzähler und einen 6-Bit-Ratenzähler. Das höchstwertige Bit des Refreshregisters ist ein "Enable Bit" für den Zähler, zum Ein- bzw. Ausschalten. Im Bild 2.8 ist der Aufbau des Refreshregisters dargestellt.
Ist das "Enable Bit" zu Eins gesetzt, wird durch Dekrementieren des 6-Bit-Ratenzählers die Refreshperiode festgelegt. Ist der Ratenzähler Null, beträgt die Refreshperiode 256 Taktzyklen. Die Refreshperiode T_R ergibt sich aus:

$$T_R = 4 * n * \frac{1}{f}$$

n Wert des Ratenzählers
f Taktfrequenz des Mikrorechnersystems.

In einem Mikrorechnersystem mit einer Taktfrequenz von 4 MHz liegt somit die Refreshperiode zwischen 1 und 64 Mikrosekunden.

Normalstackpointer (NSPSEG, NSPOFF / NSP) - In diesen Registern ist der Wert des Normalstackpointers während der Arbeit im Systemmode gespeichert. Im Systemmode kann der Normalstackpointer in allgemeine Register geladen, modifiziert und zurückgeladen werden.
Im Normalmode befindet sich der Normalstackpointer beim U8001-Prozessor in der segmentierten Arbeitsweise in den Registern R14 und R15, d.h., er benötigt zwei Wortregister. Beim U8002 bzw. beim U8001 in der nichtsegmentierten Arbeitsweise befindet er sich im Normalmode im Register R15 und benötigt ein Wortregister.
Das Register NSPOFF ist mit dem Register NSP identisch.

NSPSEG	Normalstackpointer-Segmentadresse beim U8001
NSPOFF	Normalstackpointer-Offset beim U8001
NSP	Normalstackpointer beim U8002

2.3.3. System- und Normalmode

Programme können in jedem Mikrorechnersystem in Systemprogramme, wie z.B. das Betriebssystem, und in Normalprogramme unterschieden werden. Normalprogramme sind alle Programme die nicht zum Betriebssystem gehören, insbesondere Applikationsanwendersoftware.
Das U8000-System bietet sehr gute Voraussetzungen zur Softwarestrukturierung und damit zur Erhöhung der Sicherheit bei der Programmabarbeitung durch die Möglichkeit, Programme in zwei Betriebsarten, dem System- und dem Normalmode, abzuarbeiten. Systemprogramme, wie z.B. das Betriebssystem, sollten in U8000-Mikrorechnersystemen im Systemmode laufen, allen anderen Programmen steht der Normalmode mit eingeschränktem Befehlssatz zur Verfügung. Die Trennung in beide Betriebsarten wird durch die Aufteilung der Befehle des U8000-Prozessors in

privilegierte Befehle und
nichtprivilegierte Befehle

unterstützt.

Privilegierte Befehle sind Ein-/Ausgabebefehle und Befehle mit deren Hilfe der Prozessorstatus verändert werden kann. Sie können nur im Systemmode benutzt werden. In diesem Mode gibt es keine Einschränkungen für den Programmierer.
Nichtprivilegierte Befehle sind sowohl im Systemmode als auch im Normalmode abarbeitbar. Mit ihnen ist der Anwender in der Lage alle zur Lösung seines Problems erforderlichen Programmabläufe zu programmieren. Der Prozessorstatus muß in Anwenderprogrammen nicht modifiziert werden. Ein-/Ausgabeoperationen sollten zur Verbesserung der Programmstruktur in jedem Fall über Betriebssystemfunktionen realisiert werden. Mit der Aufteilung des Befehlssatzes in privilegierte und nichtprivilegierte Befehle können Systemfunktionen und der Prozessorstatus vor dem Zugriff nicht autorisierter Anwender geschützt werden. Die erforderliche Betriebsart wird über das Bit 14 (S/N-Bit) des Flag- und Controlworts eingestellt. Ist das Bit 14 gesetzt, arbeitet die U8000-CPU im Systemmode.

Systemmode
Der Systemmode ist insbesondere für die Abarbeitung von Systemprogrammen zur Verwaltung der Ressourcen des Mikrorechners geeignet. Diesen Programmen muß der vollständige Befehlssatz des Mikroprozessors, d.h. sowohl die privilegierten als auch die nichtprivilegierten Befehle, zur Verfügung stehen. Die Trennung der Systemprogramme von allen sonstigen Programmen ist zur Erhöhung der Systemsicherheit und zur Verbesserung der Softwarestruktur zweckmäßig. Als Stackpointer wird der Systemstackpointer RR14' bzw R15' genutzt.

Normalmode
Der Normalmode dient zur Abarbeitung von Programmen, die keine Systemfunktionen beinhalten. Die Trennung von System- und sonstigen Programmen wird durch einen speziellen Stackpointer für den Normalmode unterstützt. Der Normalstackpointer steht in den Registern RR14 (segmentierte Arbeitsweise) bzw. R15 (nichtsegmentierte Arbeitsweise). In dieser Betriebsart kann nur mit nichtprivilegierten Befehlen gearbeitet werden. Der Gebrauch privilegierter Befehle führt zu einer Unterbrechung durch einen Trap

(privileged instruction trap) und zum Übergang in den Systemmode. Interrupts und Traps werden stets im Systemmode abgearbeitet. Sie benutzen, unabhängig davon in welchem Mode die Unterbrechung erfolgte, dazu stets den Systemstackpointer. Im Normalmode besteht keine Möglichkeit zur Modifikation des Systemstackpointers, der Normalstackpointer kann dagegen im Systemmode modifiziert werden. Er kann in diesem Mode mit den LDCTL-Befehlen in ein allgemeines Register geladen, modifiziert und zurückgeladen werden (s. Abschnitt 3.5.).
Es besteht bis auf den Aufruf eines System-Call keine Möglichkeit, aus dem Normalmode in den Systemmode umzuschalten. Systemfunktionen können nur über die System-Calls genutzt werden, die Rückkehr in das Normalprogramm nach dem System-Call erfolgt wieder im Normalmode.

2.3.4. Speicherorganisation

Zur Adressierung des Speicheradreßraumes verfügt der U8001-Mikroprozessor über zwei Betriebsarten, den segmentierten und den nichtsegmentierten Mode. Die Betriebsart wird durch Setzen bzw. Rücksetzen des Bits 15 (SEG-Bit) im Flag- und Controlwort gewählt.

Segmentierter Mode
Im segmentierten Mode, Bit 15 = 1 (SEG-Bit), teilt die CPU den Adreßraum in 128 Segmente auf. Jedes Segment kann einen Speicherraum von 64 KByte umfassen. Die logische 23-Bit-Adresse besteht aus einer 7-Bit-Segmentnummer und einem 16-Bit-Offset. Mit dieser Adresse kann in einem 8 MByte Adreßraum gearbeitet werden.
Die Adressierung des Speichers kann über die Register der CPU erfolgen, in diesem Fall werden 32-Bit-Register genutzt. In Befehlen enthaltene Adressen bestehen aus der im High-Teil eines Worts stehenden 7-Bit-Segmentnummer und einem 8- oder 16-Bit-Offset.
Die Segmente können im segmentierten Mode in den zwei Formaten angegeben werden.
Long-Segmente besitzen ein Speichervolumen von mehr als 256 Byte bis zu 64 KByte. Sie bestehen aus 256-Byte-Blöcken und werden insbesondere als Programm- oder Datenspeicher genutzt.
Short-Segmente umfassen ein Speichervolumen von 256 Byte, d.h., ihre Adressierung erfolgt mit einem 8-Bit-Offset. Sie bieten sich zur Nutzung als Stack- oder Datensegment an. Ein weiterer Vorteil dieser kurzen Segmente ist die Verkürzung der Adresse von 32 Bit zu 16 Bit und damit verbunden, die Verringerung des Programmspeicherbedarfs. Im Abschnitt 3.1. wird auf beide Formate ausführlich eingegangen.

Nichtsegmentierter Mode
Im nichtsegmentierten Mode, Bit 15 = 0 (SEG-Bit), kann die U8000-CPU nur mit 16-Bit-Adressen in einem Speicherbereich von 64 KByte arbeiten.
In dieser Betriebsart sind alle zur Adressierung benutzten Register 16-Bit-Register. In Befehlen enthaltene Adressen sind stets 16-Bit-Adressen.
Im U8002-Mikroprozessor ist das Bit 15 des Flag- und Controlworts stets 0, somit sind auf dieser CPU nur Programme abarbeitbar, die im nichtsegmentierten Mode geschrieben wurden. Diese Programme können auch durch die U8001-CPU nach Rücksetzen des SEG-Bits abgearbeitet werden. Segmentiert geschriebene Programme sind nur auf der U8001-CPU nach Setzen des Bits 15 lauffähig.

2.3.5. Ein-/Ausgabeadressierung

Das U8001-Mikrorechnersystem besitzt zwei im Systemode adressierbare Ein-/Ausgabeadreßräume:

Standard-Ein-/Ausgabeadreßraum
Spezial-Ein-/Ausgabeadreßraum.

Der Standard Ein-/Ausgabeadreßraum dient beim U8001- und U8002-Mikroprozessor dem Anschluß aller Peripheriegeräte, wie z. B. eines Massenspeichers oder eines Druckers an den Mikrorechner.
Der U8001-Mikroprozessor besitzt zur Programmierung der U8010-MMU-Schaltkreise einen Spezial-Ein-/Ausgabeadreßraum (s. Abschnitt 5.).
Die Art des Zugriffs auf den Standard- oder Spezial-Ein-/Ausgabeadreßraum ist aus den Statusbits des Mikroprozessors (s. Tafel 2.1) hardwaremäßig dekodierbar.
Beide Ein-/Ausgabeadreßräume können nur im Systemmode der U8001-CPU adressiert werden. Der Zugriff auf die Ein-/Ausgabeadreßräume erfolgt über Ein-/Ausgabebefehle unter Nutzung von 16-Bit-Adressen.
Durch die Aufteilung in zwei Ein-/Ausgabeadreßräume verfügt das U8001-System somit über 2 * 65536 Ein-/Ausgabeadressen.
Das U8002-System verfügt nur über den Standard Ein-/Ausgabeadreßraum. Die Breite der Ein-/Ausgabedaten kann wahlweise 8 oder 16 Bit betragen.
Der Befehl

```
OUT  %12F0, R4
```

gibt den Inhalt des Wortregisters R4 an die Standard-Ein-/Ausgabeadresse %12F0 aus.
Mit dem Befehl

```
INB RL1, %0002
```

wird ein Byte von der Standard-Ein-/Ausgabeadresse %0002 in das Byteregister RL1 eingelesen.

2.3.6. Unterbrechungssystem

Das Unterbrechungssystem des U8000 zeichnet sich durch seine Vielfalt aus. Es bearbeitet externe und interne Unterbrechungen des Programmablaufs durch ein hierarchisch organisiertes System unterschiedlicher Unterbrechungsquellen. Interne Unterbrechungen entstehen in der U8000-CPU, sie sind stets synchroner Art. Externe Unterbrechungen im System können sowohl synchron als auch asynchron sein.

Asynchrone Unterbrechungen werden als Interrupts bezeichnet. Im U8000-System sind alle Interrupts externe Unterbrechungen. Sie werden durch unabhängig von der Programmabarbeitung ablaufende Vorgänge ausgelöst.
Ein typisches Beispiel für eine externe Unterbrechung ist das interruptgesteuerte Laden eines Eingabepuffers von einem Terminal. Betätigt der Bediener eine Taste, unterbricht der Rechner das laufende Programm. Er holt das Zeichen von dem Ein-/Ausgabeport, an dem die Tastatur angeschlossen ist, ab und trägt es in den Eingabepuffer ein. Anschließend arbeitet

er im unterbrochenen Programm weiter.
Aus diesem Beispiel ist ersichtlich, daß es sich um nicht reproduzierbare Vorgänge handelt, der Tastendruck erfolgt willkürlich. Das laufende Programm wird an einer nicht vorhersehbaren Stelle unterbrochen.

Synchrone Unterbrechungen werden als Traps bezeichnet. Es gibt im U8000-System drei interne, in der CPU gebildete Traps und den externen in der U8010-MMU gebildeten Segmenttrap. Der Segmenttrap wird in der CPU über den SEGT Anschluß von der MMU ausgelöst.
Traps können jederzeit reproduziert werden. Ein Trap tritt, bis auf den EPU-Trap, auf Grund fehlerhafter Softwarebedingungen auf, z.B. bei Verletzung von Schutzmechanismen im System. Wird ein Programm unter den gleichen Bedingungen abgearbeitet, ist auch jeder Fehler dieser Art reproduzierbar. Der EPU-Trap kann in ein Softwarekonzept einbezogen werden und zum Aufruf von Softwareroutinen dienen, die einen nicht vorhandenen EPU-Co-Prozessor im System simulieren.
Mit den Traps verfügt das U8000-Mikrorechnersystem über Unterbrechungsmöglichkeiten, die bisher im 8-Bit-U880-System nicht vorhanden waren. Sie sind eine ausgezeichnete Möglichkeit zur Erkennung und Behandlung von Fehlern im Softwaresystem. Dies kann eine Unterstützung sowohl in der Testphase als auch in der Nutzungsphase eines Programms sein. Damit besteht im U8000-System die Möglichkeit, Zustände, die in anderen Systemen zum Systemabsturz führen, zu vermeiden. Das Programm kann ohne Auswirkungen auf das Betriebssystem oder andere Anwender mit einer Fehlermeldung durch das Betriebssystem abgebrochen werden.

Interrupts
Der U8000-Prozessor besitzt drei voneinander unabhängige Interruptmöglichkeiten. Für jeden dieser Interrupts steht ein CPU-Anschlußpin zur Verfügung. An den jeweiligen Interrupteingang des U8000-Mikroprozessors werden dann die im System vorhandenen Interruptquellen in einer bereits vom U880-System her bekannten Prioritätskaskadierung angeschlossen.

NMI (nichtmaskierbarer Interrupt). Dieser Interrupt ist nicht durch Software maskierbar. Er wird nur für Ereignisse der höchsten Priorität im System benutzt.
Ein typischer NMI-Anwendungsfall ist die Reaktion auf einen Netzausfall. Ist im Mikrorechnersystem Hardware zur Erkennung Netzausfällen vorhanden, löst sie bei einem Netzausfall einen nichtmaskierbaren Interrupt aus. Der Mikroprozessor kann bis zum völligen Zusammenbrechen der Spannung nur noch wenige Befehle, die für die Sicherung eines gefahrlosen Systemzustandes unbedingt erforderlich sind, ausführen.

VI (vektorisierter Interrupt). Der vektorisierte Interrupteingang ist softwaremäßig maskierbar. Durch Setzen bzw. Rücksetzen des VIE-Bits (vectored interrupt enable) kann der Programmierer im Systemmode die Abarbeitung von vektorisierten Interrupts zulassen oder sperren.
Der vektorisierte Interrupt wird von peripheren Bausteinen ausgelöst. Die Bausteine geben nach Annahme der Interruptanforderung durch die U8000-CPU ein 16-Bit-Kennwort (identifier) auf den Mikrorechnersystembus. Der Low-Teil des Kennworts wird vom Mikroprozessor als Zeiger zur Auswahl der geforderten Startadresse der Interruptserviceroutine aus der Programmstatusarea (Bild 2.10) benutzt. Die Identifikation des Schaltkreises, das

Kennbyte, entspricht dem Low-Byte des Kennworts, es ist der verwertbare Teil des Kennworts. In seinem High-Teil wird ein undefinierter Momentanwert vom Systembus gespeichert. Ein Flag- und Controlwort steht für alle Startadressen der vektorisierten Interruptserviceroutinen in der Programmstatusarea zur Verfügung. Die Verzweigung zu den Startadressen der VI-Serviceroutinen erfolgt unter Auswertung des Kennbyte durch die U8000-CPU automatisch.
Das Kennwort wird neben dem Befehlszähler und dem Flag- und Controlwort des unterbrochenen Programms im Systemstackspeicherbereich abgelegt.

NVI (nichtvektorisierter Interrupt). Der nichtvektorisierte Interrupt kann durch Setzen bzw. Rücksetzen des NVIE-Bits (non vectored interrupt enable) im Flag- und Controlwort durch den Programmierer zugelassen bzw. gesperrt werden. Damit kann der nichtvektorisierte Interrupt wie ein vektorisierter Interrupt softwaremäßig maskiert werden.
Nichtvektorisierte Interrupts werden von peripheren Bausteinen ausgelöst. Sie werden in einer Interruptserviceroutine, deren Adresse mit dem dazugehörigen Flag- und Controlwort in der Programmstatusarea steht, bearbeitet.
Das Kennwort kann bei mehreren Interruptquellen in der Interruptserviceroutine aus dem Stack ausgelesen und zur Identifikation der Interruptquelle benutzt werden. Anschließend wird innerhalb der NVI-Serviceroutine zu der entsprechenden Befehlsfolge verzweigt.
Die nichtvektorisierten Interrupts benötigen auf Grund der notwendigen Softwareaktionen zur Erkennung der Interruptquelle mehr Zeit als vektorisierte Interrupts zur Abarbeitung der erforderlichen Befehlsfolge.

Traps
Im U8000-Mikrorechnersystem können vier verschiedene Traps auftreten. Drei werden intern in der U8000-CPU gebildet, der vierte Trap ist bei Einbindung der U8010-MMU in ein U8001-Mikrorechnersystem nutzbar. Er wird bei einem fehlerhaften Speicherzugriff gebildet und extern über das SEGT-Anschlußpin an die CPU gelegt. Traps treten stets synchron auf, d.h. beim Auftreten definierter Zustände. Sie sind reproduzierbar und können nicht maskiert werden.

Extended Instruction Trap (U8000-Trap) - Der Extended Instruction Trap wird generiert, wenn die U8000-CPU eine "Extended Instruction", d.h. einen Befehl für einen EPU-Co-Prozessor (extended processing unit), ausführt und das EPA-Bit (extended processing architecture) im Flag- und Controlwort nicht gesetzt ist. Mit Hilfe des Extended Instruction Trap können auch nicht vorhandene EPU im System softwaremäßig simuliert werden. Ist die Integration eines Gleitkommaprozessors vorgesehen, so dienen die entsprechenden EPA-Befehle zur Ansteuerung des Gleitkommaprozessors der Auslösung des Traps. Er führt dann über die Abarbeitung der Gleitkommaroutine zur Simulation des Gleitkommaprozessors.

Privileged Instruction Trap (U8000-Trap) - Der Privileged Instruction Trap zeigt den Versuch der Nutzung eines privilegierten Befehls in Normalmode an. Er führt über die Programmstatusarea zu einer Traproutine, in der das den Trap verursachende Programm ermittelt wird.
Diese Routine besitzt insbesondere in Mikrorechnersystemen, in denen Anwender mit Hilfe eines Betriebssystems arbeiten, große Bedeutung. Der

Versuch eines Anwenders, privilegierte Befehle zu nutzen, führt sofort zur Unterbrechung der Abarbeitung seines Programms, das Betriebssystem ist damit wirkungsvoll vor unerlaubten Zugriffen geschützt.

System Call Trap (U8000-Trap) - Der System Call Trap tritt bei Ausführung eines System-Call auf, er führt ebenfalls über das Programmstatusarea zur Ausführung der System-Call Routine. In dieser Routine wird die im Befehlskode des Systemspeichers angegebene Nummer des System-Call ausgewertet. Bei diesem Unterprogrammaufruf wird der Befehlskode (1 Wort) als Kennwort mit dem Programmstatus im Systemstack abgelegt. Die Nummer des System-Call ist im Low-Byte des Befehlskodes enthalten und kann in einer Sprungtabelle zur Verzweigung des Programmlaufes genutzt werden. Es sind 256 unterschiedliche Routinen nutzbar.

Segmenttrap (U8001) - Der Segmenttrap wird durch die U8010-MMU in einem U8001-System ausgelöst, wenn sie eine Verletzung der Zugriffsbedingungen feststellt. Werden mehrere MMU-Speicherverwaltungsbausteine im System genutzt, kann in der Segmenttraproutine die verursachende MMU aus dem im Systemstack abgelegten Kennwort ermittelt werden. Aus den Fehlerregistern der MMU sind dann das Segment und die Fehlerursache zu ermitteln, die zur Auslösung des Traps führten. Wird im U8001-System keine U8010-MMU genutzt, ist es in speziellen Anwendungen prinzipiell denkbar, den SEGT als zusätzlichen Interrupteingang zu verwenden.

Aufbau der Programmstatusarea

Das Verständnis der Behandlung von Unterbrechungen im U8000-System setzt Kenntnisse über den Aufbau der Programmstatusarea voraus. Die Programmstatusarea ist ein softwaremäßig in der Initialisierungsphase des Systems im Systemspeicher definiertes Feld von Programmstatuswerten. Diese Werte können durch den Systemprogrammierer auch während der Programmabarbeitung modifiziert werden.
Im Bild 2.9 ist der Aufbau eines Programmstatusfeldes im Speicher abgebildet. Die U8001-CPU benötigt vier Wörter, die U8002-CPU zwei Wörter Speicherplatz für den Programmstatus.

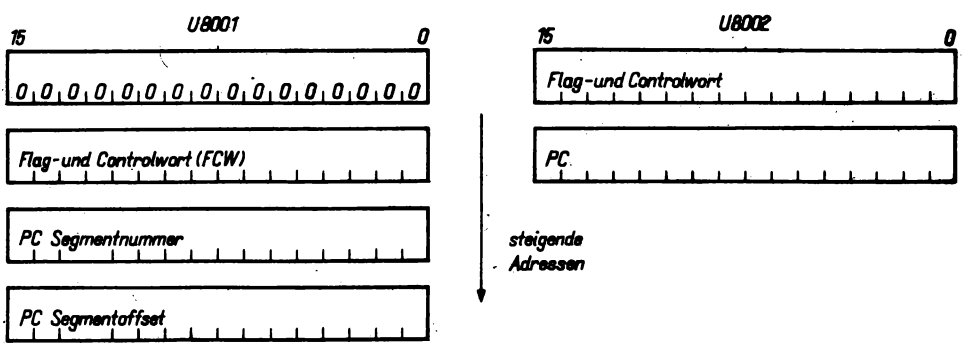


Bild 2.9. Programmstatusfeld im Speicher

Für zukünftige Systemerweiterungen ist zu Beginn der Programmstatusarea ein dem Umfang eines Trap entsprechender Speicherbereich freigehalten. Er wird gegenwärtig nicht genutzt und sollte auch vom Programmierer freigehalten werden, um die Kompatibilität der Software nach einer Weiterent-

wicklung des Prozessors zu sichern.
Die Programmstatusarea wird durch die CPU mit Hilfe des in einem speziellen Register der CPU gespeicherten Programmstatusareapointers adressiert. Dieser Pointer ist bei der segmentiert arbeitenden U8001-CPU ein aus zwei Worten bestehender Wert. Er enthält die Segmentnummer und den Offset des Programmstatusareapointers. Die U8001-CPU im nichtsegmentierten Mode und die U8002-CPU benötigen als Programmstatusareapointer eine 16-Bit-Adresse.
Das Low-Byte des Programmstatusareapointers, des Offsets oder der 16-Bit-Adresse ist stets 0, d.h., die Programmstatusarea beginnt an einer 256-Byte-Grenze im Systemspeicher. In der Programmstatusarea sind für alle Interrupts und Traps des U8000-Mikrorechnersystems die Programmstatuswerte gespeichert.
Das Bild 2.10 zeigt den detaillierten Aufbau der Programmstatusarea, in der folgenden Übersicht ist der für die Programmstatusarea bei maximaler Nutzung der VI-Startadressen benötigte Speicher angegeben:

Tafel 2.2. Speicherbedarf der Programmstatusarea

Speicherbedarf	U8001	U8002
für zukünftige Erweiterungen	1*4 Worte = 8 Byte	1*2 Worte = 4 Byte
für Programmstatuswerte der Traps	4*4 Worte = 32 Byte	4*2 Worte = 16 Byte
für den Programmstatus der NMI-Routine	2*2 Worte = 8 Byte	1*2 Worte = 4 Byte
für den Programmstatus der NVI-Routine	2*2 Worte = 8 Byte	1*2 Worte = 4 Byte
für das Flag- und Controlwort der VI-Routinen	1*2 Worte = 2 Byte	1*1 Wort 2 Byte
für die Startadressen der VI-Routine	128*2 Worte = 512 Byte	256*1 Wort = 256 Byte
	$\sum$ 570 Byte	$\sum$ 286 Byte

Der für die VI-Routinen benötigte Speicherplatz kann den Erfordernissen des Systems angepaßt werden, d.h., der Speicherbereich muß nicht vollständig freigehalten werden.
Der Platz für den Segmenttrap im U8002-Prozessor wird nicht genutzt, er muß freigehalten werden. Für alle vektorisierten Interrupts wird ein gemeinsames Flag- und Controlwort genutzt. Bei Auftreten eines vektorisierten Interrupts liest der Prozessor das Flag- und Controlwort und ermittelt über das von der Interruptquelle ausgesendete Kennbyte (Low-Byte des Kennworts) die zum Interrupt gehörende Startadresse der Interruptserviceroutine. Die Startadresse der Interruptserviceroutine wird in den Befehlszähler geladen. Anschließend beginnt die Abarbeitung der Routine mit ihrem ersten Befehl.
Die Startadresse der ersten vektorisierten Interruptserviceroutine steht beim U8001-Mikroprozessor auf der Adresse %3C der Programmstatusarea. Beim U8002-Mikroprozessor steht die Startadresse der ersten vektorisierten Interruptserviceroutine auf der Adresse %1E der Programmstatusarea. Das "%" Zeichen kennzeichnet eine hexadezimale Zahl.

Das von der Interruptquelle ausgegebene Kennbyte wird mit zwei multipliziert und zu der o.g. Adresse addiert. Damit ergeben sich in der Programmstatusarea für die vektorisierten Interrupts die folgenden Beziehungen:

Tafel 2.3. VI-Adressen in der Programmstatusarea

Vektor	Adresse in der U8001-Programmstatusarea	Kennwort Low-Byte	Adresse in der U8002-Programmstatusarea	Kennwort Low-Byte
0	%3C	%00	%1E	%00
1	%40	%02	%20	%01
2	%44	%04	%22	%02
3	%48	%06	%24	%03
.	.	.	.	.
127	%0238	%FE	%011C	%7F
128	-	-	%011E	%80
.			.	.
255	-	-	%021E	%FF

Da die Adresse einer vektorisierten Interruptserviceroutine vier Byte umfaßt, können im U8001-Mikrorechnersystem 128 vektorisierte Interrupts zur Anwendung kommen. Im U8002-Mikrorechnersystem umfaßt die Adresse einer Interruptserviceroutine nur zwei Byte, deshalb sind es 256 vektorisierte Interrupts, in diesem System werden auch ungerade Kennbyte von den Interruptquellen ausgegeben.

Behandlung von Interrupts und Traps

Die Behandlung von Interrupts und Traps erfolgt im U8000-System in fünf Schritten. Der Annahme der Unterbrechungsanforderung, dem Retten des Systemstatus vor dem Interrupt, dem Laden des neuen Programmstatus aus dem Programmstatusarea, der Ausführung der Serviceroutine und der Rückkehr in das unterbrochene Programm.
Diese fünf Schritte werden bei jeder Unterbrechung des Programmlaufs durch Interrupts oder Traps von der U8000-CPU abgearbeitet.
Dabei gibt es einen Unterschied zwischen den internen Unterbrechungen (Extendend Instruction Trap, Privileged Instruction Trap, System Call Trap) und den externen Unterbrechungen (Interrupt, Segmenttrap).
Bei den externen Unterbrechungen wird von der jeweiligen Ereignisquelle ein spezielles Kennbyte zu ihrer Identifikation ausgegeben.
Die internen Traps geben das erste Wort des zum Trap führenden Befehls anstelle des Kennworts aus. Beim System Call erfolgt damit die Verzweigung zu der gewünschten System Call Routine.

Annahme der Unterbrechungsanforderung - Beide U8000-Prozessoren gehen bei der Annahme von Unterbrechungsanforderungen in den Systemmode über, um den Systemstackbereich zu aktivieren. Sie übernehmen das Kennbyte bei einer externen Unterbrechung bzw. das erste Wort des zum Trap führenden Befehls bei einer internen Unterbrechung, zusammen mit dem Prozessorstatus des unterbrochenen Programms, in den Systemstackbereich.
Die Annahme der Unterbrechungsanforderung erfolgt beim U8001-Mikroprozessor grundsätzlich im segmentierten Mode.

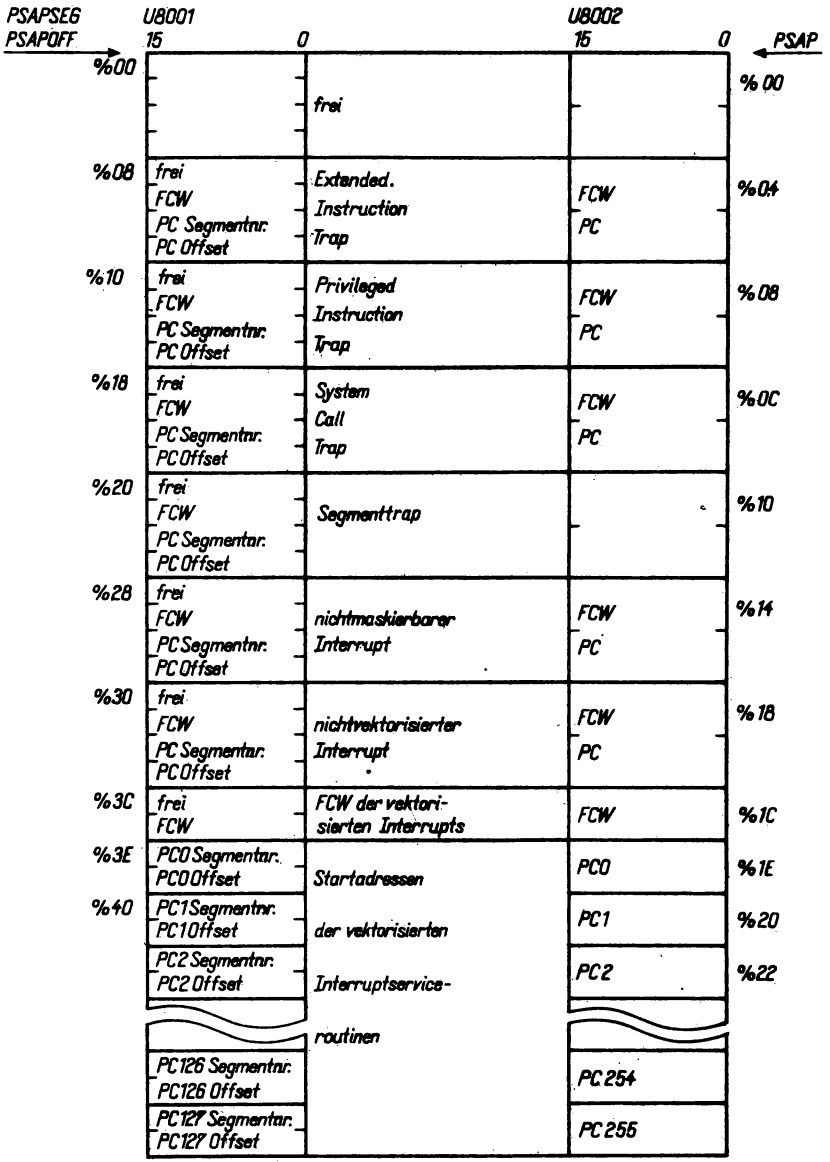


Bild 2.10. Programmstatusarea

Retten des aktuellen Systemstatus - Der aktuelle Systemstatus, d.h. der Status des unterbrochenen Programms und die Adresse des nächsten Befehls, wird im Systemstack abgespeichert.
Dabei wird zuerst der Inhalt des Befehlszählers abgespeichert, anschließend das Flag- und Controlwort sowie das Kennwort.
Aus der Tafel 2.4 ist der Inhalt des Befehlszählers für jeden Interrupt oder Trap im Detail zu entnehmen. Dabei ist zu beachten, daß bei der Ausführung von mehrfach auszuführenden Befehlen, den sogenannten Repeat-

befehlen, vor Ausführung des letzten Befehls der Inhalt des Befehlszählers wieder auf diesen Befehl gestellt wird.

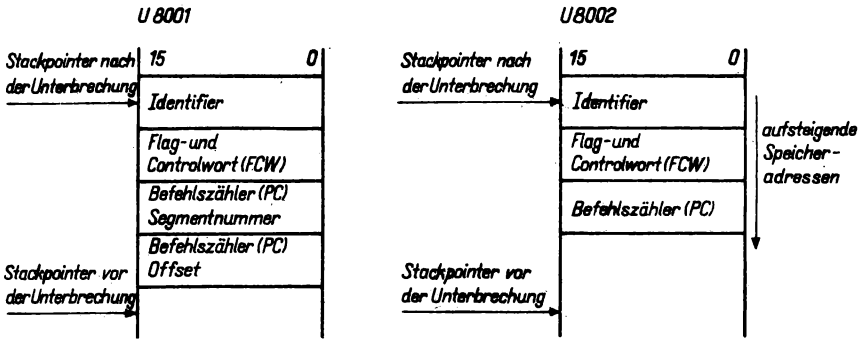


Bild 2.11. Systemstack nach einer Unterbrechung

Bei der Abarbeitung der Repeatbefehle wird ein Zähler nach jeder Befehlsausführung dekrementiert. Erst nach seinem Nulldurchgang wird die Programmabarbeitung mit dem nächsten Befehl fortgesetzt.

Tafel 2.4. Inhalt des Befehlszählers nach einer Unterbrechung

Unterbrechung	Inhalt des Befehlszählers
Interrupt	Adresse des nächsten Befehls
System Call Trap	Adresse des nächsten Befehls
Segmenttrap	Adresse des nächsten Befehls
Extended Instruction Trap	Adresse des nächsten Befehls
Privileged Instruction Trap	Adresse des zweiten Worts im Befehl bei Mehrwortbefehlen - Adresse des nächsten Befehls bei Einwortbefehlen

Laden des neuen Programmstatus - Nach der Rettung des aktuellen Systemstatus erfolgt das automatische Laden des neuen Systemstatus aus der Programmstatusarea. Die U8000-CPU lädt die zum Interrupt oder Trap gehörenden Programmstatuswerte in die Programmstatusregister. Damit ist der Anerkennungszyklus für die Unterbrechung beendet.
Wenn im eingelesenen Programmstatus die maskierbaren Interrupts im FCW freigegeben sind, kann unmittelbar nach Abschluß dieser Phase ein höher priorisierter Interrupt wiederum zu einer Unterbrechung führen.

Ausführung der Serviceroutine - Das Laden des neuen Programmstatus der Serviceroutine aus der Programmstatusarea führt zur Abarbeitung des ersten Befehls der Serviceroutine. Die in der Serviceroutine benötigten Registerinhalte sind zur Gewährleistung eines störungsfreien Weiterlaufs im unterbrochenen Programm nach Abschluß der Serviceroutine zu retten.
Der U8000-Prozessor besitzt leistungsfähige Befehle zur Registerrettung, wie z.B. den PUSHL-Befehl. Er ermöglicht die Abspeicherung von zwei 16-Bit-Registern mit einem Befehl.
Der LDM-Befehl ist auch zur Rettung des Registersatzes nutzbar, es muß jedoch beachtet werden, daß er nicht im Stackspeicherbereich angewandt wird, da er die Registerwerte ohne Veränderung des Stackpointers einträgt.

In Abhängigkeit von den Systembedingungen können über die Bits VIE und NVIE im Flag- und Controlwort in der Serviceroutine die maskierbaren Interrupts freigegeben oder gesperrt werden.

Rückkehr in das unterbrochene Programm - Der IRET-Befehl des U8000-Systems besitzt im Gegensatz zum RETI-Befehl des U880-Systems keine Bedeutung für die Hardware. Er unterscheidet sich vom RET-Befehl nur durch zusätzliche Stackoperationen.
Nach Abschluß der Aktivitäten in der Serviceroutine ist der Interruptstatus in der Interruptquelle softwaremäßig zurückzusetzen.
Von den Peripheriebausteinen des U880-Systems besitzt dazu nur der SIO-Schaltkreis (seriel input/output) die erforderlichen Voraussetzungen. Bei Einbindung weiterer Bausteine des U880-Systems, PIO (parallel input/output), CTC (counter timer circuit) und des DMA (direct memory access) ist spezielle Hardware zum Rücksetzen des Interruptstatus der Schaltkreise vorzusehen. Über einen Port im Ein-/Ausgabebereich kann sie aktiviert werden (s. Abschnitt 3.6.3.).
Die zu Beginn der Serviceroutine gespeicherten Registerinhalte werden mit den der Registerrettung entgegengesetzt wirkenden Befehlen, z.B. mit POPL-Befehlen, zurückgeladen.
Die Interrupt- oder Trapserviceroutinen werden mit einem IRET-Befehl beendet. Dieser Befehl veranlaßt die CPU, den Programmstatus aus dem Systemstack zurückzuladen und den Stackpointer auf den Wert vor der Unterbrechung zu stellen.
Der IRET-Befehl kann beim Mikroprozessor U8001 nur im segmentierten Mode ausgeführt werden; deshalb ist in jedem Fall das SEG-Bit im Flag- und Controlwort zu setzen. Da der Prozessor die Informationen nach Auftreten der Unterbrechungsanforderung im segmentierten Mode speicherte, müssen sie auch wieder im segmentierten Mode zurückgelesen werden, sonst würde der Prozessor eine falsche Rückkehradresse lesen.

Priorität der Unterbrechungen
Da Unterbrechungen eine unterschiedliche Bedeutung für das Mikrorechnersystem besitzen, gibt es im U8000-System eine definierte Priorität der Unterbrechungen. Sie ermöglichen es dem Programmierer, eine eindeutige Rang- und Reihenfolge in der Abarbeitung der Unterbrechungsroutinen festzulegen.

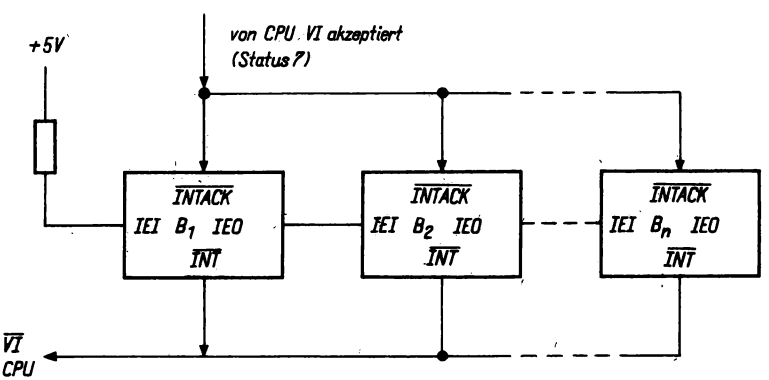


Bild 2.12. Prioritätskette am Beispiel des vektorisierten Interrupts

Nach fallender Priorität geordnet, ergibt sich die folgende Reihenfolge:

Reset
Interne Traps
Nicht maskierbarer Interrupt (NMI)
Segmenttrap (nur U8001)
Vektorisierter Interrupt (VI)
Nichtvektorisierter Interrupt (NVI)

Durch dieses Unterbrechungssystem ist eine mehrfache Schachtelung von unterschiedlichen Unterbrechungen möglich. Die Priorität in der jeweiligen Interruptebene wird beim Hardwareentwurf festgelegt. Es gibt, ähnlich dem 8-Bit-Mikroprozessorsystem U880, eine Daisy-Chain-Interruptkette.
Die Interruptquellen besitzen die Signale

IEI (interrupt enable input)
IEO (interrupt enable output)

und steuern damit, wie im Bild 2.12 dargestellt, ihre Priorität. Aus dem Bild ist ersichtlich, daß stets der zur CPU am nächsten liegende Baustein die höchste Priorität besitzt. Löst er einen Interrupt aus, schaltet er sein Signal IEO auf High und nimmt damit allen nachfolgenden Bausteinen die Möglichkeit, eine Interruptforderung an die CPU abzugeben.

2.3.7. Multiprozessorfähigkeit

Mit der Erweiterung der Anwendungsbereiche für Mikroprozessoren entstehen neue Forderungen an die Rechenleistung der Prozessoren. Ist die Rechenleistung eines Mikroprozessors nicht ausreichend, muß ein Multiprozessorsystem aufgebaut werden, indem die Informationen durch mehrere Prozessoren bearbeitet werden. Die Rechenleistung kann sowohl durch die parallele Verarbeitung von Informationen durch Co-Prozessoren als auch durch den Einsatz spezieller Co-Prozessoren, die auf einem Gebiet eine besonders hohe Leistung besitzen, erhöht werden.
Für die parallele Verarbeitung der Informationen gibt es in jedem mit relativ langsamen Peripheriegeräten ausgerüsteten Mikrorechnersystem vielfältige Einsatzmöglichkeiten.
Der Zugriff auf einen Floppy-Disk-Speicher kann z.B. für den Mikroprozessor mit dem Laden eines Puffers beendet sein. Der Co-Prozessor in der Floppy-Disk-Ansteuerung übernimmt anschließend das Schreiben der Daten auf die Diskette, während der Mikroprozessor den nächsten Pufferinhalt zusammenstellt.
Spezielle Prozessoren, z.B. Arithmetikprozessoren, zeichnen sich durch eine hohe Leistungsfähigkeit auf einem Spezialgebiet aus. Der Zentralprozessor übergibt ihnen die erforderlichen Parameter und kann das Ergebnis ohne langwierige Softwaremanipulationen übernehmen.
In Abhängigkeit vom Einsatzfall ist eine Vielzahl von Multiprozessorsystemen denkbar, angefangen von einem zentralen Mikroprozessor mit untergeordneten Co-Prozessoren und einheitlichen Ressourcen bis zu Multiprozessorsystemen mit mehreren in Verbindung stehenden Prozessoren, die in echter Parallelarbeit mit geteilten Ressourcen arbeiten.
In allen Fällen müssen die Prozessoren ihre Arbeit synchronisieren und koordinieren.

Die Möglichkeiten des Prozessors zum Aufbau von Multiprozessorsystemen werden als seine Multiprozessorfähigkeit bezeichnet.

Der U8000-Prozessor bietet sowohl hardwaremäßig als auch softwaremäßig Voraussetzungen zum Aufbau von Multiprozessorsystemen.

Hardware
Der Aufbau von Multiprozessorstrukturen wird insbesondere durch zwei Pins an der U8000-CPU unterstützt:

M_i (Multimikro-In)
M_o (Multimikro-Out)

Diese Pins dienen in Verbindung mit den gleichnamigen Signalen zum Aufbau von Multiprozessorsystemen.
Mit dem Signal M_i kann die Anforderung eines anderen Prozessors übernommen werden. Das Signal M_o dient zur Übergabe einer Anforderung an einen anderen Prozessor. In Verbindung mit der Betriebssystemsoftware wird über diese CPU-Pins die Arbeit in Multiprozessorsystemen organisiert.

Software
Zur Synchronisation und Koordination des Multiprozessorsystems stehen vier spezielle Befehle aus dem Befehlsvorrat des Prozessors zur Verfügung. Diese Befehle dienen der Verarbeitung der Signale M_i und M_o im Multiprozessorsystem.

MBIT - Mit diesem Befehl wird das Signal M_i abgefragt, ob eine Anforderung eines anderen Prozessors vorliegt.

MREQ R - Der MREQ-Befehl dient der Synchronisation mehrerer Prozessoren beim Zugriff auf gemeinsame Hardwareressourcen.
Vor der Abarbeitung des MREQ-Befehls muß in einem Register der U8000-CPU zur Realisierung einer Verzögerungszeit eine Konstante geladen werden. Die Konstante wird dekrementiert, sie ist systemabhängig und sichert den Signaldurchlauf bis zum letzten Prozessor. Die Verzögerungszeit soll Wettlauferscheinungen im System vermeiden.

MRES - Der Befehl setzt das Signal M_o zurück und gibt damit anderen Prozessoren die Möglichkeit zur Übernahme der Masterfunktion.

MSET - Der Befehl aktiviert M_o und teilt damit den anderen Prozessoren eine Anforderung auf eine gemeinsame Systemressource mit.

2.3.8. Systeminitialisierung und Reset

Die Systeminitialisierung erfolgt unmittelbar nach dem Einschalten der Betriebsspannung des Mikrorechnersystems. Durch spezielle Hardwaremaßnahmen wird dabei der Reseteingang des Mikroprozessors aktiviert.
Der Ablauf der Systeminitialisierung durch die U8000-CPU nach dem Einschalten der Betriebsspannung bzw. nach Auslösung eines Resetsignals durch den Bediener während der Programmabarbeitung ist für das Mikrorechnersystem identisch.
Nach der Aktivierung des Resetsignals werden die Programmstatusregister

(Befehlszähler und Flag- und Controlwort) gelöscht. Die CPU geht in den Systemmode und initialisiert sich mit dem Laden der Programmstatusregister.
Die dazu notwendigen Informationen liest die CPU stets am Anfang des physischen Systemspeichers. Auf der Wortadresse %0000 steht dabei sowohl beim U8001- als auch beim U8002-Prozessor keine Information, dieses Wort ist für zukünftige Anwendungen reserviert.

Die U8001-CPU liest von der Adresse %0002 das Flag- und Controlwort, es muß die segmentierte Arbeitsweise und den Systemmode festlegen. Anschließend wird von der Adresse %0004 die Segmentnummer und von der Adresse %0006 die 16-Bit-Startadresse des Programms gelesen. Der Befehlszählerinhalt ergibt sich somit aus den auf den Adressen %0004 und %0006 stehenden Werten. Der erste ausführbare Befehl des Programms kann auf der Adresse %0008 stehen.
Wird die nichtsegmentierte Arbeitsweise gewünscht, so ist sie über die Modifikation des Flag- und Controlworts durch Befehle nach dem Programmanfang vorzunehmen.

Die U8002-CPU liest nach dem Resetsignal ebenfalls das Flag- und Controlwort auf der Adresse %0002, es muß den Systemmode festlegen. Von der Adresse %0004 liest sie die Startadresse des Programms in den Befehlszähler und kann auf der Adresse %0006 den ersten Befehl des Programms abarbeiten.
In der Tafel 2.5 sind die Möglichkeiten für die U8001-CPU und die U8002-CPU tabellarisch zusammengefaßt.
Einer der ersten U8000-Befehle sollte das Laden des Refreshregisters sein. Erst nach dem Laden des Refreshregisters kann bei Nutzung dynamischer RAM-Schaltkreise sicher auf den Speicher zugegriffen werden.

Tafel 2.5. Initialbelegung des U8000-Systemspeichers

Adresse	U8001	U8002
%0000	frei	frei
%0002	FCW	FCW
%0004	Segmentnummer	Befehlszähler
%0006	Offset	1. Befehl
%0008	1.Befehl	
%000A		

Das Resetsignal löst neben den Interrupts und dem Segmenttrap auch eine externe Unterbrechung im U8000-Mikrorechnersystem aus.
Dieser Zustand wird durch Aktivieren des RESET-Pins der U8000-CPU eingenommen.
Das RESET-Pin wird durch Hardware beschaltet und kann sowohl nach dem Einschalten der Betriebsspannung als auch durch Betätigen eines am Mikrorechner angebrachten Tasters aktiviert werden. Nach Aktivierung des Resetsignals werden sofort, unabhängig vom momentanen Zustand der CPU, die o.g. Aktionen ausgeführt. Im Unterbrechungssystem des U8000-Mikrorechners besitzt das Resetsignal die höchste Priorität.

3. Programmierung des 16-Bit-Mikroprozessors U8000

Die umfassenden Möglichkeiten bei der Programmierung des 16-Bit-Mikroprozessors U8000 verleihen diesem System seine Leistungsfähigkeit und Anwendungsbreite. Durch die Software erhält das Mikroprozessorsystem erst seine spezifischen Merkmale zur Lösung von Anwenderaufgaben. Die Programmierung, d. h. die Entwicklung von Software, bildet den Schwerpunkt bei der breiten Einführung der Mikrorechentechnik.
Viele Probleme verlangen Lösungen, die nur durch die Entwicklung neuer oder die Anpassung bereits vorhandener Software erfolgen kann. Dies wird verstärkt auf dem Gebiet der höheren Programmiersprachen geschehen. Auf vielen Gebieten sind jedoch auch zukünftig Kenntnisse der Assemblersprache des Mikroprozessors erforderlich. Dies gilt insbesondere für Anwendungsgebiete, die besondere Anforderungen an die Abarbeitungszeit und die Effizienz des Maschinenkode stellen, und für die Programmierung der Treiberprogramme für die peripheren Geräte des Mikrorechners.

3.1. Adreß-, Daten- und Befehlsformate

3.1.1. Adreßformate

Das U8000-Mikrorechnersystem nutzt zwei Adreßbereiche, den

- Speicheradreßbereich und den
- Ein-/Ausgabeadreßbereich.

Im **Speicheradreßbereich** der segmentiert arbeitenden U8001-CPU kommen zwei Adreßformate zur Anwendung, der Long-Offset (segmented long offset) und der Short-Offset (segmented short offset).

Mit dem **Long-Offset** werden in der segmentierten Arbeitsweise Operanden in Long-Segmenten (s. Abschnitt 2.3.4.) adressiert.
Der Long-Offset nimmt 32 Bit bzw. zwei Wörter ein, er enthält im ersten Wort die Segmentnummer und im zweiten Wort den Offset.
Die Segmentnummer steht im High-Teil des Worts. Das Bit 15 der Segmentnummer ist bei Verwendung des Long-Offsetformats, wie im Bild 3.1 abgebildet, stets 1. Der Low-Teil des Worts, das die Segmentnummer enthält, ist immer 0.
Der Offset im Long-Offsetformat ist eine 16-Bit-Adresse, sie ermöglicht die Adressierung von Operanden in einem 64-KByte-Speicheradreßraum.
Mit dem Long-Offsetformat können Operanden im gesamten Speicheradreßraum der U8001-CPU, d. h. in 128 Segmenten mit je 64 KByte Speicheradreßraum, adressiert werden.
Mit dem **Short-Offset** werden in der segmentierten Arbeitsweise Operanden in Short-Segmenten, die ein Speicherraum von 256 Byte besitzen, adressiert.
Der Short-Offset nimmt 16 Bit, d.h. ein Wort ein.
Die Segmentnummer steht im High-Teil des Worts. Das Bit 15 des Adreßworts ist zur Kennzeichnung des Short-Offsetformats stets 0.
Der Offset im Short-Offsetformat ist eine 8-Bit-Adresse, er steht im Low-Teil des Worts und ermöglicht die Adressierung von Operanden in einem 256-Byte-Speicheradreßraum.
Im nichtsegmentierten Mode (nonsegmented) kommt zur Adressierung des Speicheradreßbereichs eine 16-Bit-Adresse zur Anwendung.

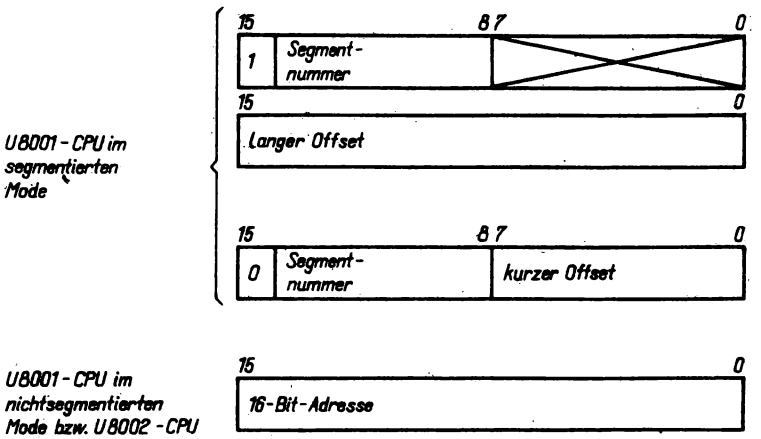


Bild 3.1. Adreßformate im segmentierten Mode (Speicheradreßbereich)

Im **Ein-/Ausgabeadreßbereich**, d.h. im Ein-/Ausgabeadreßraum und im Spezial-Ein-/Ausgabeadreßraum, werden 16-Bit-Adressen angewandt.

3.1.2. Datenformate

Die U8000-CPU unterstützt die Arbeit mit den folgenden Datenformaten:

Bit - Das Bit ist die kleinste Informationseinheit des Systems. Die U8000-CPU zeichnet sich durch eine Vielzahl statischer und dynamischer Befehle zur Verarbeitung von Bits aus.

Byte - Ein Byte besteht aus 8 Bit, es kann vorzeichenbehaftet oder ohne Vorzeichen sein. Ein vorzeichenloses Byte ist eine 8-Bit-Zahl. Sie ermöglicht die Darstellung von Zahlen im Bereich von 0 - 255. Wird durch ein Byte eine vorzeichenbehaftete Zahl dargestellt, dann handelt es sich um eine 7-Bit-Zahl. Sie ermöglicht die Darstellung von Zahlen im Bereich von -127 bis +127. Das Vorzeichen wird durch das höchstwertige Bit des Byte, das "msb"-Bit (most significant bit), ausgedrückt. Ist das "msb"-Bit gesetzt, so ist die Zahl negativ. Ist es nicht gesetzt ist die Zahl positiv.

Wort - Ein Wort besteht aus 16 Bit oder zwei Byte, es kann wie ein Byte vorzeichenbehaftet oder ohne Vorzeichen sein.
Ein vorzeichenloses Wort ist eine 16-Bit-Zahl. Sie ermöglicht die Darstellung von Zahlen im Bereich von 0 - 65535. Das vorzeichenbehaftete Wort ist eine 15-Bit-Zahl. Es ermöglicht die Darstellung von Zahlen im Bereich von -32767 bis +32767. Das Vorzeichen wird durch das höchstwertige Bit des Worts, das "msb"-Bit ausgedrückt. Ist das "msb"-Bit gesetzt, so ist die Zahl negativ. Ist es nicht gesetzt, ist die Zahl positiv.
Das höherwertige Byte (High-Byte) eines Worts steht im Speicher stets auf einer geraden Adresse ($AD_0 = 0$). Das niederwertige Byte (Low-Byte) steht auf der darauffolgenden ungeraden Adresse ($AD_0 = 1$).
Die Adresse eines Worts zeigt auf das höherwertige Byte.

Langwort - Ein Langwort besteht aus 32 Bit oder zwei Wörtern. In ihm kann eine vorzeichenlose 32-Bit-Zahl im Bereich von 0 bis 134.217.727 angegeben werden. Die Adresse eines Langworts ist, wie einen Wortadresse, eine geradzahlige Adresse (AD_0 = 0), sie zeigt auf das höherwertige Wort (High-Wort) des Langworts. Das niederwertige Wort des Langworts (Low-Wort) steht im Speicher auf der der Adresse des Langworts folgenden nächsten geraden Adresse.

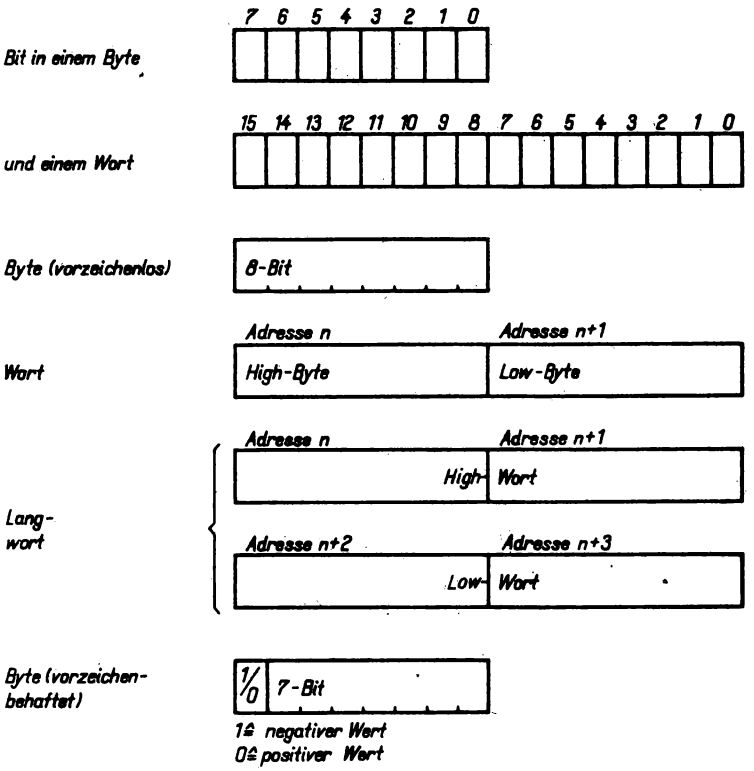


Bild 3.2. Datenformate

3.1.3. Befehlsformate

Der Mikroprozessor U8000 besitzt neun Befehlsformate, die sich in insgesamt 33 Unterformate gliedern. Die Befehlsformate sind vom jeweiligen Befehl und seiner Notation abhängig. Im Befehlssatz stehen die für den Befehl anwendbaren Befehlsformate in Verbindung mit dem entsprechenden Operationskode.
Für vier häufig benutzte Befehle wurde ein kompaktes Befehlsformat implementiert. Alle anderen Befehle werden dem allgemeinen Befehlsformat entsprechend kodiert. Im Bild 3.3 ist der Aufbau der beiden U8000-Befehlsformate dargestellt.
Im allgemeinen Befehlsformat wird im Kennzeichenfeld W/B dem Prozessor mitgeteilt, ob er eine Wort- oder eine Byteoperation ausführen soll. Ist W/B=1, führt er eine Wortoperation aus, wenn W/B=0, ist handelt es sich um eine Byteoperation. Die Adressierungsart wird im Kennzeichenfeld (mo) angegeben. Der Operationskode, Kennzeichenfeld (opco), enthält den be-

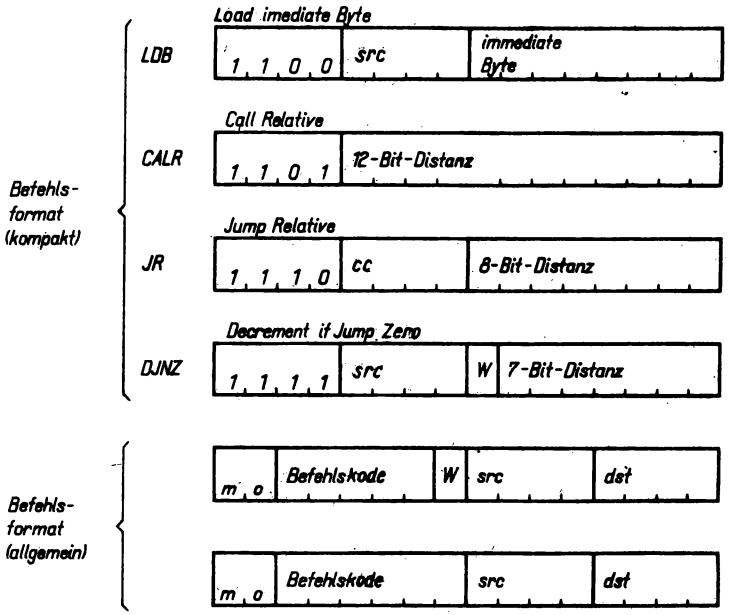


Bild 3.3. Befehlsformate der U8000-CPU

fehlsspezifischen Kode, aus ihm leitet der Prozessor seine Operationen zur Ausführung des Befehls ab. Das "src"- und "dst"-Kennzeichenfeld enthält die Registeradressen der Quell- und Zieloperanden.

3.2. Kodierung der Register und Flags

Die Register werden in den Befehlen der folgenden Tafel entsprechend kodiert.

Tafel 3.1. Kodierung der Register des Mikroprozessors U8000

Register				binäre Kodierung
RQ0	RR0	R0	RH0	0000
		R1	RH1	0001
	RR2	R2	RH2	0010
		R3	RH3	0011
RQ4	RR4	R4	RH4	0100
		R5	RH5	0101
	RR6	R6	RH6	0110
		R7	RH7	0111
RQ8	RR8	R8	RL0	1000
		R9	RL1	1001
	RR10	R10	RL2	1010
		R11	RL3	1011
RQ12	RR12	R12	RL4	1100
		R13	RL5	1101
	RR14	R14	RL6	1110
		R15	RL7	1111

Dieser Kode wird für die "src"- und "dst"-Register im Befehlsformat, entsprechend Bild 3.3, in der Übersetzungsphase durch den Assembler eingesetzt.
Neben der Kodierung der Register ist bei bedingten Befehlen auch die Kodierung der Bedingungskodes von Bedeutung. In der Tafel 3.2 ist die binäre Kodierung der Bedingungskodes angegeben. In der linken Spalte ist der im Assemblerquellprogramm in Groß- oder Kleinbuchstaben anzugebende Mnemokode eingetragen, neben ihm steht der im kodierten Befehl enthaltene Bedingungskode.
Der Bedingungskode spiegelt das Ergebnis einer Verschiebeoperation, einer logischen oder arithmetischen Operation wider.

Tafel 3.2. Kodierung der Bedingungskodes des Mikroprozessors U8000

Mnemokode	Binärkode	Bedeutung	Flagbitstellung
Z	0110	Null (zero)	Z=1
NZ	1110	Nicht Null (not zero)	Z=0
C	0111	Übertrag (carry)	C=1
NC	1111	Kein Übertrag (not carry)	C=0
PL	1101	Plus (plus)	S=0
MI	0101	Minus (minus)	S=1
EQ	0100	Gleich (equal)	Z=1
NE	0110	Ungleich (not equal)	Z=0
OV	0100	Überlauf (overflow)	V=1
NOV	1100	Kein Überlauf (no overflow)	V=0
PE	0100	Gerade Parität (parity even)	P=1
PO	1100	Ungerade Parität (parity odd)	P=0
GE	1001	Größer als oder gleich (greather than or equal)	(S XOR V)=0
LT	0001	Kleiner als (less than)	(S XOR V)=1
GT	1010	Größer als (greather than)	(Z OR (S XOR V)=0
LE	0010	Kleiner oder gleich (less or than)	(Z OR (S XOR V)=1
UGE	1111	Vorzeichenlos größer als oder gleich (unsigned greather than or equal)	C=0
ULT	0111	Vorzeichenlos kleiner als (unsigned less than)	C=1
UGT	1011	Vorzeichenlos größer als (unsigned greather than)	((C=0) AND (Z=0))=1
ULE	0011	Vorzeichenlos kleiner als oder gleich (unsigned less than or equal)	(C OR Z)=1
F	0000	falsch (always false)	-
	1000	richtig (always true)	-

Trotz unterschiedlicher Mnemoniks sind mitunter die Kodierungen gleich, da gleiche Flags abgefragt werden. Bei der Programmierung sollte stets darauf geachtet werden, daß die benutzten Mnemoniks dem Sinn des Programms entsprechen. Die Gleichheit von zwei Operanden kann sowohl mit dem Bedingungskode für Z (Null) als auch für EQ (Gleich) getestet werden. Die Bedingung EQ ist jedoch sinnentsprechend und sollte in diesem Fall zur Verbesserung der Lesbarkeit in das Quellprogramm eingetragen werden. Der Test, ob das Ergebnis einer Operation Null ist, muß dementsprechend mit dem Bedingungskode für Z erfolgen.

3.3. Hinweise zur Notation

Die Befehle werden zur besseren Darstellung in den Befehlslisten nur mit Großbuchstaben geschrieben, obwohl alle Mnemoniks auch in Kleinbuchstaben durch den Assembler (s. Abschnitt 6.) verarbeitbar sind. Der in Großbuchstaben angegebene Teil des Befehls (s. Abschnitt 3.5.) ist unveränderlich, d.h., er muß immer in dieser Form angegeben werden. Der kleingeschriebene Teil des Befehls ist der veränderliche Teil in der Notation. Er stellt Adressen, Daten oder Registerbezeichnungen dar.
Die Beispiele werden unter Einbeziehung von hexadezimalen, dezimalen und binären Zahlen erläutert.
Die möglichen Adressierungsarten werden je Befehl in einer Tafel angegeben.
In der linken Hälfte enthält die Tafel die für den Befehl anwendbaren Adressierungsarten des "dst"- und des "src"-Operanden. In der Spalte (mo) wird die in den Befehlskode einzusetzende binäre Kodierung für jede Adressierungsart angegeben.
In der rechten Hälfte der Tafel stehen die Zykluszeiten für jede Zugriffsart. Dabei wird nach Befehlen, die Byte bzw. Wortoperanden (Byte/Wort) oder Langwortoperanden (Long) verarbeiten, unterschieden.
In diesen Befehlsgruppen kann befehlsspezifisch nach den drei Notationsarten NS (nonsegmented), SS (segmented short offset) und SL (segmented long offset) unterschieden werden (s. Abschnitt 3.1.1.).
Auf der rechten Seite der Befehlsdarstellung sind spezifische Hinweise zu den jeweiligen Notationsarten angegeben.
Die Befehlsnotation gilt prinzipiell sowohl für den segmentierten als auch für den nichtsegmentierten Mode.
Die Anzahl der T-State ist je Befehl für jede Adressierungs- und Notationsart angegeben. Daraus läßt sich die Abarbeitungszeit für jeden Befehl nach folgender Beziehung ermitteln.

$$T = \frac{1}{f} * n$$

n Anzahl der T-State
f Taktfrequenz des Mikrorechnersystems

Im Befehlskode ist für Befehle, die sowohl Wort- als auch Bytebefehle in der Kodierung zulassen, im Kennzeichenfeld W angegeben. Mit W=1 wird angegeben, daß es sich um einen Wortbefehl, und mit W=0, daß es sich um einen Bytebefehl handelt. Das Kennzeichenfeld W ist identisch mit dem Kennzeichenfeld W/B im Bild 3.3. Die binäre Kodierung kann den Befehlsformaten entnommen werden.
Zur Verdeutlichung wird am Beispiel des EX-Befehls (s. Abschnitt 3.5.1.) der Tafelaufbau erklärt.
Der Befehl

```
EXB RH4, %F000
```

tauscht den Inhalt des Byteregisters RH4 mit dem Inhalt des Speicherplatzes %F000 aus. Der Befehl entspricht der dritten Zeile in der Tafel des EX-Befehls. Der "dst"-Operand ist RH4, der "src"-Operand ist %F000. Die Adressierungsart ist die Direktwertadressierung (DA). Im Befehlskode sind mo=01, W=0 und dst=0 einzusetzen.

Damit ergibt sich die Kodierung

```
0110110001000000
1111000000000000
```

für diesen Befehl. Er benötigt in der angegebenen Notationsart (nonsegmented) 15 T-State, d.h. in einem Mikrorechnersystem mit 4 MHz Taktfrequenz 3,75 Mikrosekunden Abarbeitungszeit.
Zur Arbeit mit den Befehlslisten ist die Beachtung der folgenden Vereinbarungen notwendig:

Sonderzeichen

%	Kennzeichen für eine Hexadezimalzahl Dezimalzahlen werden ohne ein vorangestelltes Sonderzeichen notiert. Binärzahlen werden im Binärkode angegeben.
#	Kennzeichen für einen Datenwert
@	Kennzeichen für eine indirekt angegebene Adresse
$	Kennzeichen für den aktuellen Wert des Befehlszählers (PC)
!	Kommentar wird in Ausrufezeichen eingeschlossen.
<< >>	Die Segmentnummern (Long-Offset) sind in zwei Winkel eingeschlossen.
|<< >> adr|	Die Kennzeichnung eines Short-Offset erfolgt durch die Einschließung der Segmentnummer und des Offsets in senkrechte Strichzeichen.

Abkürzungen

PB	Privilegierter Befehl
src.	Source- oder Quelloperand
dst.	Destination- oder Zieloperand
r	Register
r_bx	Indexregister für basisindizierte Adressierung
cc	Bedingungskode (condition code)
s	Zähler bei den Rotations- und Verschiebebefehlen
e	Kode für EPU-Co-Prozessoren in den EPU-Befehlen
W	W/B (word/byte)
SS	(segmented short offset)
SL	(segmented long offset)
NS	(nonsegmented)
msb	Höchstwertiges Bit (most significant bit)
lsb	Niederwertiges Bit (last significant bit)

Adressen

Das Adreßformat unterscheidet sich auch im Quellprogramm in seiner Notationsform. Die Segmentnummer, symbolisch oder direkt, wird in jedem Fall zur Unterscheidung vom Offset in zwei öffnende und zwei schließende Winkel eingeschlossen. Der Offset kann unmittelbar nach der Segmentnummer angegeben werden.
Der Befehl

```
LDB RH4, <<07>> %A000
```

lädt den Inhalt des Speicherplatzes %A000 aus dem Segment <<07>> im Long-Offsetformat in das Byteregister RH4.

Bei Anwendung des Short-Offsets erfolgt die Kennzeichnung durch die zusätzlich Einschließung der aus Segmentnummer und Offset bestehenden Adresse in senkrechte Striche. Adressen werden als Zahlenwert, vorzugsweise im hexadezimalen Format, oder symbolisch angegeben.
Die Adressierungsarten werden mit den im Abschnitt 3.4. angegebenen Abkürzungen notiert.
Der Befehl

```
LDB RH4, |<<34>>%F0|
```

lädt den Inhalt des Speicherplatzes %F0 aus dem Segment 34 im Short-Offsetformat in das Byteregister RH4.

Daten
Daten werden in den Beispielen als hexadezimaler, dezimaler und binärer Zahlenwert oder als symbolischer Wert, jedoch mit einem vorangestellten "#" Zeichen, als Kennzeichen für einen Datenwert, notiert.

```
LDB RH4, #CR          !Symbolischer Wert!
LD R2, #%0000         !Hexadezimaler Wert!
LD R7, #5634          !Dezimaler Wert!
LDB RL0, 01100110     !Binaerer Wert!
```

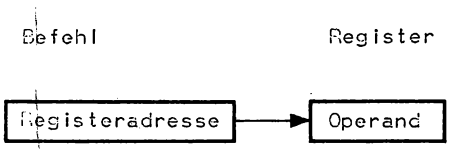
Flags
Die Angabe der Flags erfolgt in der Befehlsliste nur, wenn durch die Abarbeitung des Befehls eine Modifikation der Flags eintritt.

3.4. Adressierungsarten

Die möglichen Adressierungsarten bestimmen wesentlich die Leistungsfähigkeit eines Mikroprozessors. Die U8000-CPU besitzt ein komfortables Adressierungssystem, acht verschiedene Adressierungsarten geben dem Programmierer die Möglichkeit effektive Assemblerprogramme zu schreiben. In Verbindung mit der variablen Registerstruktur des Prozessors (Akkumulatorregister, Indexregister, Adreßregister und Basisregister) ist auch die Voraussetzung zur Erstellung effizienter Maschinenprogramme durch Compiler oder Interpreter gegeben.
Für die Entwicklung von Assemblerprogrammen ist die Kenntnis der Adressierungsarten eine grundlegende Voraussetzung zur Ausnutzung der vollen Leistungsfähigkeit des Prozessors. Die bei der Darstellung der allgemeinen Register (s. Abschnitt 2.1.) getroffenen Aussagen zur Einschränkung der Registernutzung sind unbedingt zu beachten.
Die Adressierung dient letztlich der Verarbeitung von Operanden, sie können sich im Speicheradreßraum, im Ein-/Ausgabeadreßraum oder im Registersatz der CPU befinden. Bei der Erläuterung der acht Adressierungsarten des U8000-Mikroprozessors wird bei einem Unterschied zwischen der segmentierten und der nichtsegmentierten Arbeitsweise immer zuerst ein Beispiel für die nichtsegmentierte Arbeitsweise gegeben.

3.4.1. Registeradressierung (R)

Der Operand ist Inhalt eines Registers



Diese Adressierungsart ist die schnellste im U8000-Mikrorechnersystem; sie erstreckt sich auf die allgemeinen Register der CPU. Es können in beiden Versionen der U8000-CPU Byte-, Wort-, Langwort- und Vierfachwortregister der CPU adressiert werden.
Zwischen der segmentierten und der nichtsegmentierten Arbeitsweise besteht kein Unterschied.

Beispiele:

```
LD   R1, R0                 !Lade den Inhalt des Registers R0 in das
                            Register R1!

     vorher                   nachher
  R0 0000                  R0 0000
  R1 1111                  R1 0000
  R2 2FDC                  R2 2FDC
```

```
LDL  RR2, RR4               !Lade den Inhalt des Langwortregisters RR4
                            in das Langwortregister RR2!

        vorher                   nachher
RR0  R0 10AD             RR0  R0 10AD
     R1 FFAC                  R1 FFAC
RR2  R2 A347             RR2  R2 1256
     R3 6D45                  R3 DD99
RR4  R4 1256             RR4  R4 1256
     R5 DD99                  R5 DD99
```

3.4.2. Direktwertadressierung (IM)

Der Operand ist Bestandteil des Befehles

Befehl | Register | Speicher

Operand

Bei der Direktwert- oder Immediate-Adressierung ist der Operand Teil des Programmkodes. Zur Verminderung des Speicheraufwandes besitzt der U8000-Prozessor ein Reihe von kurzen Befehlen zur Direktwertadressierung.
Zwischen der segmentierten und der nichtsegmentierten Arbeitsweise besteht kein Unterschied.

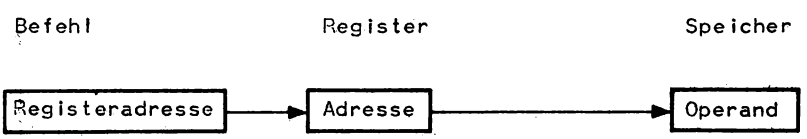
Beispiele:

```
LD R4, #%2468              !Lade die Hexadezimalzahl %2468 in das Regi-
                            ster R2!

     vorher                    nachher
  R3 EFDD                   R3 EFDD
  R4 0000                   R4 2468
  R5 2DFF                   R5 2DFF
```

```
LDL RR4, %#24689ABC        !Lade die Hexadezimalzahl %24689ABC in  das
                            Langwortregister RR4!

        vorher                    nachher
  RR2 R2 FCDE               RR2 R2 FCDE
      R3 11AB                   R3 11AD
  RR4 R4 0000               RR4 R4 2468
      R5 0000                   R5 9ABC
  RR6 R6 2256               RR6 R6 2256
      R7 FFFF                   R7 FFFF
```

3.4.3. Indirekte Adressierung (IR)

Die Adresse des Operanden steht in einem Register.



Die im Register gespeicherte Adresse dient als Zeiger zum Operanden; der Zugriff erfolgt indirekt über die Adresse in einem Register. Der Zeiger kann auf den Speicheradreßraum oder auf den Ein-/Ausgabeadreßraum gerichtet sein.
Das den Zeiger enthaltende Register, wird im Assemblerquellprogramm durch ein vorrangestelltes Zeichen "@" gekennzeichnet. Diese Adressierungsart ist sehr wichtig, weil der Zeiger in den Registern einfach manipuliert werden kann. Mit der indirekten Adressierung kann leicht auf zum adressierten Operanden benachbarte Operanden zugegriffen werden.

Beispiele:

Nichtsegmentierter Mode

LD R4, @R6 !Lade das Register R4 mit dem Inhalt des durch das Register R6 adressierten Speicherplatzes!

vorher	nachher	Speicher Adresse	Daten
R4 0000	R4 AD22	3FFC	0F11
R6 3FFE	R6 3FFE	3FFE	AD22
		4000	33FE

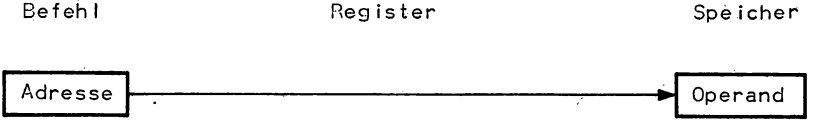
Segmentierter Mode

LDB RH4, @RR6 !Lade das Byteregister RH4 mit dem Inhalt des Speicherplatzes, der durch das Langwortregister RR6 adressiert wird!

vorher	nachher	Segment	Speicher Adresse	Daten
RR4 R4 RH4 AA	RR4 R4 RH4 55	07	73FE	3344
RL4 BB	RL4 BB	07	7400	5566
RR6 R6 8700	RR6 R6 8700	07	7402	7788
R7 7400	R7 7400			

3.4.4. Direkte Adressierung (DA)

Der Operand ist der Inhalt der im Befehlskode enthaltenen Adresse.



Die im Programmkode enthaltene Adresse dient dem Zugriff auf Operanden im Speicher und Ein-/Ausgabeadreßraum.
Die Angabe der Adresse des nächsten Befehls bei einer Programmverzweigung mit Jump- bzw. Call-Befehlen erfolgt ebenfalls in direkter Adressierung.

Beispiele:

Nichtsegmentierter Mode

LD R4, %4000 !Lade Register R4 mit dem Inhalt des Speicherplatzes %4000!

vorher	nachher	Speicher Adresse	Daten
R4 0000	R4 4444	3FFE	33BB
		4000	4444
		4002	FC55

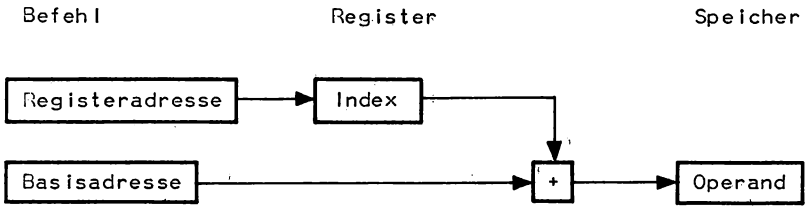
Segmentierter Mode

LDL RR4, <<06>>%534E !Lade das Langwortregister RR4 mit dem Inhalt der Speicherplätze %534E und %5350 aus dem Segment 06!

vorher	nachher	Segment	Speicher Adresse	Daten
RR4 R4 0000	RR4 R4 11AA	06	534C	0000
R5 0000	R5 2FD2	06	534E	11AA
		06	5350	2FD2

3.4.5. Indizierte Adressierung (X)

Der Operand ist Inhalt einer Adresse, die aus der Addition der im Befehl enthaltenen Basisadresse und dem in einem Register stehenden Index gebildet wird.



Der Programmkode beinhaltet die Basisadresse und die Adresse des Indexregisters. Über den in einem Wortregister (Ausnahme: R0) gespeicherten Index kann bequem auf verschiedene Positionen der durch die Basisadresse definierten Tabellen oder Felder zugegriffen werden.

Beispiele:

Nichtsegmentierter Mode

```
LD R4, %4000(R5)        !Lade Register R4 mit dem Inhalt des durch
                        die Addition der Basisadresse %4000 und dem
                        im Register R5 stehenden Index adressierten
                        Speicherplatzes!
```

vorher		nachher		Speicher Adresse	Daten
R4	0000	R4	478E	4000	33A5
R5	0004	R5	0004	4002	678F
				4004	478E

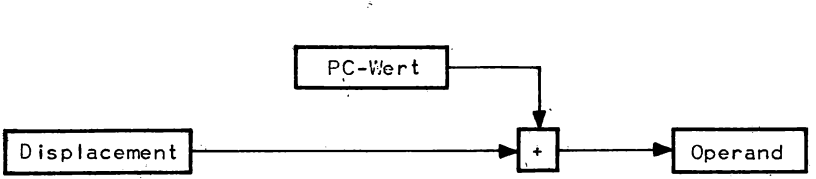
Segmentierter Mode

```
LDB RL6, <<03>>%2200(R7)    !Lade das Byteregister RL6 mit dem Inhalt
                            des durch die Addition der Basisadresse
                            %2200 und dem im Register R7 stehenden Index
                            im Segment 03 adressierten Speicherplatzes!
```

vorher		nachher		Speicher Segment	Adresse	Daten
R6	RH6 00	R6	RH6 00	03	2200	0000
	RL6 00		RL6 33	03	2202	AF22
R7	0004	R7	0004	03	2204	33BC
				03	2206	458F

3.4.6. Relative Adressierung (RA)

Der Operand ist Inhalt einer aus dem Befehlszählerinhalt und der Addition bzw. Subtraktion eines Distanzwerts (Displacement) gebildeten Adresse.



Diese Adressierungsart kann nur innerhalb eines Segments angewandt werden, da die Segmentnummer nicht verändert wird. Sie dient im wesentlichen der Schaffung von verschieblichen Programmen, die an keine physischen Speichergrenzen gebunden sind.
Der im Befehl angegebene Distanzwert ist eine 16-Bit-Zahl, demzufolge können, ausgehend vom Inhalt des Befehlszählers, Operanden im Bereich von -32768 bis +32768 adressiert werden.

Beispiele:

Nichtsegmentierter Mode

JR Z, $ + 4 !Springe, ausgehend von der Adresse des JR-Befehls, um 4 Byte vorwärts, wenn das Zero-Flag gesetzt ist!

vorher	nachher	Speicher Adresse	Programm	Mnemonik
PC 0100	PC 0104	0100	E804	JR $ + 4
		0102	A9B0	INC R11
		0104	AB60	DEC R6

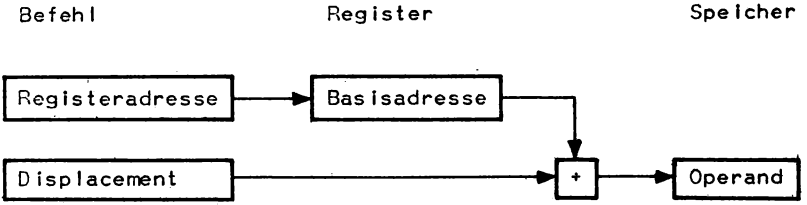
Segmentierter Mode

LD R6, $ + 2 !Lade Register R6 mit dem Inhalt des durch die Addition des Befehlszählerinhaltes und des Distanzwerts adressierten Speicherplatzes!

vorher	nachher	Speicher Adresse	Programm	Mnemonik
R6 0000	R6 1234	0200	3106	LD R6, $ + 2
PC 0200	PC 0204	0202	0002	
		0204	9E08	RET
		0206	1234	

3.4.7. Basisadressierung (BA)

Der Operand ist Inhalt der durch die Addition einer Basisadresse in einem Register und einem im Befehl enthaltenen positiven oder negativen Distanzwert (Displacement) gebildeten Adresse.



Zur Basisadressierung kann, mit Ausnahme der Register R0 und RR0, jedes Register des Mikroprozessors U8000 eingesetzt werden; sie wird ausschließlich durch Ladebefehle im Speicheradreßraum genutzt. Der Distanzwert wird zur Verbesserung der Übersichtlichkeit des Assemblerquellprogramms mit einem Doppelkreuz versehen und in Klammern angegeben.

Beispiele:

Nichtsegmentierter Mode

```
LD R2,  R3(%#24)        !Lade Register R2 mit dem Inhalt des durch
                         die Addition der Basisadresse in R3 und dem
                         Distanzwert %24 adressierten Speicher-
                         platzes!
```

vorher	nachher	Speicher Adresse	Daten
R2 0000	R2 6666	6000	3333
R3 6000	R3 6000	.	.
		6024	6666

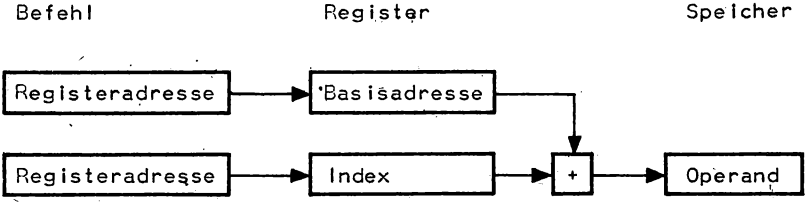
Segmentierter Mode

```
LD R2, RR4(%#12)        !Lade Register R3 mit dem Inhalt des durch
                         die Addition der Basisadresse in RR4 und
                         dem Distanzwert %12 adressierten Spei-
                         cherplatzes!
```

vorher	nachher	Segment	Speicher Adresse	Daten
RR2 R2 0000	RR2 R2 5566	03	6400	0000
R3 0000	R3 0000			
RR4 R4 0300	RR4 R4 0300	.	.	.
R5 6400	R5 6400	03	6412	5566
		03	6414	7788

3.4.8. Basisindizierte Adressierung (BX)

Der Operand ist Inhalt der durch die Addition einer Basisadresse in einem Register und dem ebenfalls in einem Register stehenden positiven oder negativen Distanzwert (Displacement) gebildeten Adresse.



Die basisindizierte Adressierung ist eine Erweiterung der Basisadressierung, sie kann nur für Ladebefehle genutzt werden. Im Unterschied zur Basisadressierung steht der Distanzwert auch in einem Register. Damit besteht die Möglichkeit über, erst in der Abarbeitungsphase des Programms berechenbare Distanzwerte bequem auf Daten in Tabellen oder Feldern zuzugreifen.

Beispiele:

Nichtsegmentierter Mode

LD R4, R5(R6) !Lade Register R4 mit dem Inhalt der durch die Addition der Basisadresse in R5 mit dem Distanzwert in R6 ermittelten Adresse!

vorher	nachher	Speicher Adresse	Daten
R4 0000	R4 6666	4200	3333
R5 4200	R5 4200	.	.
R6 0008	R6 0008	4208	6666

Segmentierter Mode

LD R3, RR4(R2) !Lade Register R3 mit dem Inhalt des durch die Addition der Basisadresse in RR4 (Segmentnummer und Offset) mit dem Distanzwert in R2 adressierten Speicherplatzes!

vorher	nachher	Segment	Speicher Adresse	Daten
RR2 R2 0004	RR2 R2 0004	03	8800	0000
R3 0000	R3 3333	03	8802	F45A
RR4 R4 0300	RR4 R4 0300	03	8804	3333
R5 8800	R5 8800	03	8806	5E66

3.5. Befehlssatz des 16-Bit-Mikroprozessors U8000

Der Befehlssatz der U8000-CPU besteht aus 110 verschiedenen Befehlen /1/. Diese Befehle sind in Verbindung mit den acht Adressierungsarten der U8000-CPU die Basis für die Softwareentwicklung. Sie werden in diesem Abschnitt, funktionell in 10 Befehlsgruppen geordnet, in ihrer Assemblernotation erläutert. Der jeweilige Maschinenkode ist je Befehl angegeben.

Lade- und Exchangebefehle - Diese Befehlsgruppe dient dem Transfer von Daten zwischen einem Quell- und einem Zieloperanden. Als Quell- oder Zieloperanden kommen die allgemeinen Register der U8000-CPU und der Speicheradreßraum des Mikrorechners in Frage.

Arithmetikbefehle - Die Befehle dieser Gruppe dienen der Ausführung von arithmetischen Festkomma- und BCD-Operationen. Im Ergebnis dieser Operationen werden die CPU-Flags gesetzt.

Logikbefehle - Mit dieser Befehlsgruppe kann die logische Verknüpfung von Byte- oder Wortoperanden ausgeführt werden. Im Ergebnis dieser Operationen werden die CPU-Flags gesetzt.

Programmsteuerbefehle - Sie dienen zur Realisierung von Programmverzweigungen durch bedingte bzw. unbedingte Sprungbefehle in der Programmabarbeitung sowie zur Ausführung unbedingter Unterprogrammrufe.

Bitmanipulationsbefehle - Diese Befehlsgruppe unterstützt die statischen und dynamischen Operationen mit Bits in Byte- und Wortoperanden, sowohl in den allgemeinen Registern der CPU als auch im Speicheradreßraum des Mikrorechners.

Rotations- und Verschiebebefehle - Mit dieser Befehlsgruppe können alle erforderlichen statischen und dynamischen Rotations- und Verschiebebefehle in den allgemeinen Registern der CPU ausgeführt werden.

Blocktransfer- und Blocksuchbefehle - Sie ermöglichen eine effiziente Arbeit mit Datenblöcken und Zeichenketten. Die Datenblöcke und Zeichenketten können maximal 64 KByte umfassen.

Ein-/Ausgabebefehle - Mit dieser Befehlsgruppe erfolgt der Informationsaustausch mit der Peripherie des Mikrorechners sowohl im Ein-/Ausgabeadreßraum als auch im Spezial-Ein-/Ausgabeadreßraum.

CPU-Steuerbefehle - Diese Befehlsgruppe dient im wesentlichen der Veränderung des Programmstatus der CPU, der Arbeit mit ihren Steuerregistern sowie der Manipulation mit den Flags der CPU.

EPU-Befehle - Sie dienen der Zusammenarbeit der U8000-CPU mit EPU-Co-Prozessoren (extended processor unit).

Der U8000-Befehlssatz ist zur Verbesserung der Übersichtlichkeit für den Programmierer in den Funktionsgruppen alphabetisch geordnet.
Im Anhang A ist eine alphabetische Übersicht der U8000-Befehle mit Verweisen auf die funktionell geordnete Befehlsbeschreibung angegeben.

3.5.1. Lade- und Exchangebefehle

CLR **Clear** CLR

Der Inhalt des adressierten Operanden wird 0 gesetzt.

Mnemonik: **CLR dst / CLRB dst**
Operation: dst := 0

Befehlsformat: F1.1 DA: dst=0
X dst<>0
W/B: mo00110wdst.src.

Adressierungsart		Zyklen Byte/Wort		
dst	mo	NS	SS	SL
R	10	7	7	7
IR	00	8	8	8
DA	01	11	12	14
X	01	12	12	15

Beispiel: CLRB %4440
Lade das durch %4440 adressierte Byte mit 0.

EX **Exchange** EX

Die Inhalte der "src"- und "dst"-Operanden werden unter Einbeziehung eines temporären Registers der CPU vertauscht.
Der EX-Befehl ermöglicht das Vertauschen von Operanden ohne die Einbeziehung eines zusätzlichen Speicherplatzes oder Registers.

Mnemonik: **EX dst,src / EXB dst,src**
Operation: tmp := src
src := dst
dst := tmp

Befehlsformat: F2.1 DA: dst=0
X : dst<>0
W/B: mo10110wsrc.dst.

Adressierungsart			Zyklen Byte/Wort		
dst	src	mo	NS	SS	SL
R	R	10	6	6	6
R	IR	00	12	12	12
R	DA	01	15	16	18
R	X	01	16	16	19

Beispiel: EXB RH3, <<43>>%FFE0(R1)
Tausche den Inhalt des Byteregisters RH3 mit dem Inhalt der sich aus der Addition von %FFE0 und dem Inhalt des Registers R12 ergebenden Adresse aus.

LD Load LD

Der LD-Befehl dient dem Datentransfer zwischen den Registern und dem Speicher sowie dem Laden von Festwerten.
Der Inhalt des durch "src" adressierten Operanden wird in den durch "dst" adressierten Operanden geladen. Der Ladebefehl kann in drei Versionen angewandt werden.

Mnemonik: **LD dst,src / LDB dst,src / LDL dst,src**
Operation: dst := src

Load Register

Adressierungsart			Zyklen Wort/Byte			Long		
dst	src	mo	NS	SS	SL	NS	SS	SL
R	R	10	3	3	3	5	5	5
R	IR	00	7	7	7	11	11	11
R	DA	01	9	10	12	12	13	15
R	X	01	10	10	13	13	13	16
R	BA		14	14	14	17	17	17
R	BX		14	14	14	17	17	17

Befehlsformat: F2.1
W/B: mo10000wsrc.dst.
L: mo010100src.dst.

R:
IR: src<>0
DA: src=0
X : src<>0

Befehlsformat: F4.4
W/B: 0011000wsrc.dst.
..Displacement..
L: 00110101src.dst.
..Displacement..

BA: src<>0

Befehlsformat: F4.3
W/B: 0111000wsrc.dst.
0000r_bx00000000
L: 01110101src.dst.
0000r_bx00000000

BX: src<>0

Beispiel: LDB RH3, <<43>>%FFE0(R12)
Lade das Byteregister RH3 mit dem Inhalt der sich aus der Addition von %FFE0 und dem Inhalt des Registers R12 ergebenden Adresse.

Load Memory

Adressierungsart			Zyklen Wort/Byte			Long		
dst	src	mo	NS	SS	SL	NS	SS	SL
IR	R	00	8	8	8	11	11	11
DA	R	01	11	12	14	14	15	17
X	R	01	12	12	15	15	15	18
BA	R	00	14	14	14	17	17	17
BX	R	01	14	14	14	17	17	17

Befehlsformat: F2.2
W/B: mo10111wdst.src.
L: mo011101dst.src.

IR:
DA: dst=0
X : dst<>0

Befehlsformat: F4.2
W/B: 0011001wdst.src.
..Displacement..
L: 00110111dst.src.
..Displacement..

BA: dst<>0

Befehlsformat: F4.1
W/B: 0111001wdst.src.
0000r_bx00000000
L: 01110111dst.src.
0000r_bx00000000

BX: dst<>0

Beispiel: LDB %FFE0, RH3
Lade den Speicherplatz %FFE0 mit dem Inhalt des Registers RH3.

Load Immediate

Adressierungsart			Zyklen Wort/Byte			Zyklen long		
dst	src	mo	NS	SS	SL	NS	SS	SL
R	IM	00	7	7	7	11	11	11
R	IM		5	*)				
IR	IM		11	11	11			
DA	IM	01	14	15	17			
X	IM	01	15	15	18			

Befehlsformat: F2.1 R: src=0
W/B: mo10000wsrc.dst.
L: mo010100src.dst.

Befehlsformat: F3.2 R:
B: 1100dst...src...

*) Bei Bytebefehlen wird vorrangig das Format 3.2 genutzt.

Befehlsformat: F5.1 IR:
W/B: mo00110wdst.0101 DA: dst=0
X dst<>0

Beispiel: LDB %FFE0(R5), #%34
Lade den Inhalt der Adresse, die sich aus der Addition von %FFE0 und dem Inhalt des Registers R5 ergibt, mit %34.

LDA **Load Address** **LDA**

Der LDA-Befehl dient dem Laden von Adressen in die allgemeinen Register. Die Adresse des "src"-Operanden wird in den "dst"-Operanden geladen. Der Quelloperand wird dabei nicht verändert.
Im segmentierten Mode ist der Inhalt der Bits 16 bis 23 und des Bit 31 der Segmentnummer nach dem LDA-Befehl undefiniert. Das Zielregister ist im nichtsegmentierten Mode ein Wortregister, im segmentierten Mode ein Langwortregister.

Mnemonik: **LDA dst,src**
Operation: dst := address (src)

Adressierungsart			Zyklen		
dst	src	mo	NS	SS	SL
R	DA	01	12	13	15
R	X	01	13	13	16
R	BA		15	15	15
R	BX		15	15	15

Befehlsformat: F2.1 DA: src=0
mo110110src.dst. X : src=0

Befehlsformat: F4.4 BA: src<>0
00110100src.dst.
..Displacement..

Befehlsformat: F4.3 BX: src<>0
01110100src.dst.
..Displacement..

Beispiel: LDA R3, R5(R7)
Lade in Register R3 die Adresse, die sich aus der Addition der Basisadresse in R5 und des Index in R7 ergibt.

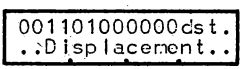
LDAR Load Address Relative LDAR

Der LDAR-Befehl ermöglicht die Anwendung von relativen Adressen.
Die Adresse des Quelloperanden wird berechnet und in das Zielregister geladen, sie ergibt sich aus der Addition der im Befehl angegebenen und der dem LDAR-Befehl folgenden Adresse.
Der Quelloperand wird nicht verändert. Das Zielregister ist ein Wortregister im nichtsegmentierten Mode. Im segmentierten Mode ist es ein Langwortregister. Die Bits 16 bis 23 und das Bit 31 der Segmentnummer besitzen nach Ausführung des LDAR-Befehls (Adressierung nur im gleichen Segment) den Wert 0. Die im Befehl angegebene Distanz ist ein 16-Bit-Wert im Bereich von -32768 bis +32767.

Mnemonik: **LDAR dst,src**
Operation: dst := adr (src)

Adressierungsart	Zyklen
dst src	
R RA	15

Befehlsformat: F4.4



Beispiel: LDAR RR4, POINT
Lade in Register RR4, die Adresse die sich aus der Addition der Adresse von POINT und dem Wert des PC + 2 ergibt. Die Bits 16 bis 23 und das Bit 31 der Segmentnummer im Register R3 sind zu 0 gesetzt.

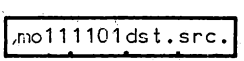
LDK Load Constant LDK

Der LDK-Befehl dient der Programmkodeoptimierung bei Verwendung von Konstanten im Bereich von 0 bis 15.
Die Konstante ist in den Bits 0 bis 3 im Befehl enthalten und wird in die gleichwertigen Bits des Zielregisters geladen. Die Bits 4 bis 15 haben nach Ausführung des Befehls den Wert 0.

Mnemonik: **LDK dst,src**
Operation: dst := src (0 - 15)

Adressierungsart	Zyklen Wort
dst src mo	
R IM 10	5

Befehlsformat: F1.2



IM: src (0 - 15)

Beispiel: LDK R4, #12
Lade in Register R4 die Konstante 12.

LDM Load Multiple LDM

Der LDM-Befehl ermöglicht das effektive Umladen von Registerinhalten zwischen CPU und Speicher. Damit besteht die Möglichkeit mit einem einzigen Befehl die allgemeinen Register der CPU in den Speicher zu laden bzw. sie aus dem Speicher in den Registersatz zurückzuladen.
In einen Zielbereich werden "n" Wörter aus einem Quellbereich geladen. Die Werte des Quellbereichs werden nicht verändert. Im Befehl wird das Register angegeben, von dem der Transfer ausgehen bzw. zu dem er führen soll, die Anzahl "n" der zu transferierenden Worte und die Speicheradresse. Die Register und der Speicherbereich werden in aufsteigender Richtung beschrieben bzw. gelesen.
Der LDM-Befehl kann in den folgenden zwei Varianten genutzt werden.

Mnemonik: **LDM dst,src,n**
Operation: dst := src (n Worte, 1<=n<=16)

Load Multiple - Register := Memory

Adressierungsart			Zyklen Wort		
dst	src	mo	NS	SS	SL
R	IR	00	11+3n	11+3n	11+3n
R	DA	01	14+3n	15+3n	17+3n
R	X	01	15+3n	15+3n	18+3n

Befehlsformat: F6.1

W: mo011100src.0001 0000dst.0000num.

DA: src=0
X : src<>0
n = num.

Beispiel: LDM R0, REGFELD(R7), #16
Alle allgemeinen Register werden, beginnend von der Adresse, die sich aus der Addition der Adresse von REGFELD und dem Index in R7 ergibt, geladen.

Load Multiple - Memory := Register

Adressierungsart			Zyklen Wort		
dst	src	mo	NS	SS	SL
IR	R	00	11+3n	11+3n	11+3n
DA	R	01	14+3n	15+3n	17+3n
X	R	01	15+3n	15+3n	18+3n

Befehlsformat: F6.1

W: mo011100dst.1001 0000src.0000num.

DA: src=0
X : src<>0
n = num.

Beispiel: LDM REGFELD(R7), R0, #16
Alle allgemeinen Register werden auf die Adresse, die sich aus der Addition von REGFELD und dem Index in R7 ergibt, geladen.

LDR Load Relative LDR

Der LDR-Befehl dient der Entwicklung von verschieblichen Programmen. Er ermöglicht den Transfer von Daten zwischen Registern und dem Speicher in verschieblichen Programmen.
Der Inhalt des Quelloperanden "src" wird in den Inhalt des Zieloperanden "dst" geladen. Die relative Adresse wird berechnet durch die Addition des Befehlszählerwerts (PC) und der im Programmkode angegebenen Adresse bzw. der Distanz oder dem Displacement. Der PC-Wert ergibt sich aus der Adresse des dem LDR-Befehl folgenden Befehls. Die Distanz ist ein 16-Bit-Wert im Bereich von -32788 bis +32767. Der Operand muß im segmentierten Mode im gleichen Segment wie der LDR-Befehl stehen.

Mnemonik: **LDR dst,src / LDRB dst,src / LDL dst,src**
Operation: dst := src

Load Relative - Register := Memory

Adressierungsart	Zyklen Wort/Byte	Long
dst src		
R RA	14	17

Befehlsformat: F4.4

W/B: 0011000w0000dst. ..Displacement..

L: 001101010000dst. ..Displacement..

Beispiel: LDRB RH2, FELD
Das Byteregister RH2 wird mit dem Inhalt des Speicherplatzes FELD geladen. Die Adresse von FELD ergibt sich aus der Addition der durch den Assembler errechneten Adresse von FELD und der Adresse des nach dem LDRB-Befehl stehenden Befehls.

Load Relative - Memory := Register

Adressierungsart	Zyklen Wort/Byte	Long
dst src		
RA R	14	17

Befehlsformat: F4.2

W/B: 0011001w0000src. ..Displacement..

L: 001101010000 src ..Displacement..

Beispiel: LDRB FELD, RH2
Das Byteregister RH2 wird auf den Speicherplatz FELD geladen. Die Adresse von FELD ergibt sich aus der Addition der durch den Assembler errechneten Adresse und der Adresse des nach dem LDRB-Befehl stehenden Befehls.

POP Pop POP

Der POP-Befehl dient der Rückspeicherung von Operanden, die mit einem PUSH-Befehl aus den allgemeinen Registern in den Speicher geladen wurden. Der Inhalt des durch das Quellregister "src" adressierten Speicherplatzes wird in das Zielregister geladen. Das Quellregister fungiert dabei als Stackpointer, es wird bei einem POP-Befehl um zwei und bei einem POPL-Befehl um vier inkrementiert.
Als Quellregister kann mit Ausnahme der Register R0 bzw. RR0 jedes Wort- oder Langwortregister genutzt werden. Bei Anwendung des POP-Befehls kann das "dst"- mit dem "src"-Register identisch sein. Der POPL-Befehl setzt unterschiedliche "src" und "dst"-Register voraus.

Mnemonik: **POP dst,src / POPL dst,src**
Operation: dst := src

Adressierungsart			Zyklen Wort			Zyklen Long		
dst	src	mo	NS	SS	SL	NS	SS	SL
R	IR	10	8	8	8	12	12	12
IR	IR	00	12	12	12	19	19	19
DA	IR	01	15	16	18	22	23	25
X	IR	01	16	16	19	23	23	26

Befehlsformat: F2.1

W: mo010111src.dst.

L: mo010101src.dst.

Beispiel: POP OR4, OR7
Der durch R7 als Quellregister adressierte Operand wird auf den durch R4 adressierten Speicherplatz geladen, R7 wird um zwei inkrementiert.

PUSH Push PUSH

Der PUSH-Befehl dient der Zwischenspeicherung von Operanden im Speicher, die im weiteren Programmablauf mit einem POP-Befehl zurückgespeichert werden können.
Der Inhalt des durch das Quellregister "src" adressierten Registers oder Speicherplatzes wird in den Speicher geladen. Das Zielregister "dst" fungiert dabei als Stackpointer. Es wird bei einem PUSH-Befehl um zwei und bei einem PUSHL-Befehl um vier dekrementiert. Als Stackpointer kann mit Ausnahme der Register R0 bzw. RR0 jedes Wort- oder Langwortregister genutzt werden. Nur bei Anwendung des PUSH-Befehls kann das "dst"- mit dem "src"-Register identisch sein, der PUSHL-Befehl setzt unterschiedliche "src"- und "dst"-Register voraus.

Mnemonik: **PUSH dst,src / PUSHL dst,src**
Operation: dst := src

Adressierungsart			Zyklen Wort			Zyklen Long		
dst	src	mo	NS	SS	SL	NS	SS	SL
IR	R	10	9	9	9	12	12	12
IR	IR	00	13	13	13	20	20	20
IR	DA	01	13	14	16	21	21	24
IR	X	00	14	14	17	21	21	24

Befehlsformat: F2.2 DA: src=0
X : src<>0

W: `mo010011dst.src.`

L: `mo010001dst.src.`

Beispiel: PUSH @R4, @R7
Der durch das Register R4 adressierte Operand wird auf den durch das Register R7 adressierten Speicherplatz geladen. Das Register R4 wird dabei um zwei dekrementiert.

PUSH Immediate

Adressierungsart			Zyklen Wort
dst	src	mo	
IR	IM	00	12

Befehlsformat: F5.1

W: `mo001101dst.1001`
`.....src.....`

Beispiel: PUSH @R4, %8000
Der durch R4 adressierte Operand wird mit dem Festwert %8000 geladen.

3.5.2. Arithmetikbefehle

ADC **Addition with Carry** **ADC**

Der ADC-Befehl realisiert die Addition mit Übertrag.
Die Inhalte der durch "src" und "dst" adressierten Operanden und des Carry-Flags werden addiert und auf dem "dst"-Operanden abgespeichert.

Mnemonik: **ADC dst,src / ADCB dst,src**
Operation: dst := dst + src + C
Flags:
C: Ist nur gesetzt, wenn ein Übertrag des "msb"-Bits erfolgt.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das Ergebnis negativ ist.
V: Ist nur gesetzt, wenn ein Überlauf auftritt.
D: Beim ADCB-Befehl gelöscht, sonst keine Beeinflussung.
H: Ist nur beim ADCB-Befehl gesetzt, wenn ein Überlauf des Bit 3 zum Bit 4 im Ergebnis erfolgt, sonst keine Beeinflussung.

Adressierungsart			Zyklen Wort/Byte
dst	src	mo	
R	R	10	5

Befehlsformat: F2.1
W/B: mo11010wsrc.dst.

Beispiel: ADD R7, R9
ADC R6, R8
32-Bit Addition - doppelt genau / Low-Woerter plus High-Woerter mit Übertrag der Low-Woerter.

ADD **Addition** **ADD**

Der ADD-Befehl realisiert die Addition von Operanden.
Die Inhalte der durch "src" und "dst" adressierten Operanden werden addiert und auf dem "dst"-Operanden abgespeichert.

Mnemonik: **ADD dst,src / ADDB dst,src / ADDL dst,src**
Operation: dst := dst + src
Flags:
C: Ist nur gesetzt, wenn ein Übertrag des "msb"-Bits erfolgt.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das Ergebnis negativ ist.
V: Ist nur gesetzt, wenn ein Überlauf auftritt.
D: Nur beim ADDB-Befehl gelöscht, sonst keine Beeinflussung.
H: Ist nur beim ADBB-Befehl gesetzt, wenn ein Überlauf des Bit 3 im Ergebnis erfolgt, sonst keine Beeinflussung.

Adressierungsart			Zyklen Wort/Byte			Long		
dst	src	mo	NS	SS	SL	NS	SS	SL
R	R	10	4	4	4	8	8	8
R	IM	00	7	7	7	14	14	14
R	IR	00	7	7	7	14	14	14
R	DA	01	9	10	12	15	16	18
R	X	01	10	10	13	16	16	19

Befehlsformat: F2.1
W/B: mo00000wsrc.dst.
L: mo010110src.dst.

IM: src=0
IR: src<>0
DA: src=0
X : src<>0

Beispiel: ADD R4, @R6
Addiere zum Inhalt des Registers R4 den Inhalt des durch R6 adressierten Speicherplatzes.

CP — Compare — CP

Der CP-Befehl dient dem Vergleich von Operanden.
Der Inhalt des durch "src" adressierten Operanden wird mit dem Inhalt des durch "dst" adressierten Operanden verglichen. Dabei wird der "src"-Operand vom "dst"-Operand subtrahiert. Es wird kein Operand verändert, im Ergebnis der Operation werden die entsprechenden Flags gesetzt.

Mnemonik: **CP dst,src / CPB dst,src / CPL dst,src**
Operation: dst - src
Flags: C: Ist nur gesetzt, wenn src > dst ist, d.h., es zeigt Borrow an.
S: Ist nur gesetzt, wenn das Ergebnis negativ ist.
V: Ist nur gesetzt, wenn ein Überlauf auftritt.

Adressierungsart			Zyklen Wort/Byte			Long		
dst	src	mo	NS	SS	SL	NS	SS	SL
R	R	10	4	4	4	8	8	8
R	IM	00	7	7	7	14	14	14
R	IR	00	7	7	7	14	14	14
R	DA	01	9	10	12	15	16	18
R	X	01	10	10	13	16	16	19
IR	IM	00	11					
DA	IM	01	14	15	17			
X	IM	01	15	15	18			

Befehlsformat: F2.1

W/B: `mo00101wsrc.dst.`

L: `mo010000src.dst.`

R:
IM: src=0
IR: src<>0
DA: src=0
X : src<>0

Befehlsformat: F5.1

W/B: `mo00110wdst.0001`
`.....src.....`

IR: dst
DA: dst=0
X : dst<>0

Beispiel:
```
CPB RH0, #%0D !Springe zur Marke CR (carriage return), wenn ein!
JR Z, CR_     !Wagenruecklaufzeichen im Byteregister RH0 steht!
CPB RH0, #%0A !Springe zur Marke LF (line feed) wenn ein!
JR Z, LF_     !Zeilenendezeichen im Byteregister RH0 steht!

CR_:

LF_:
```

DAB — Decimal Adjust — DAB

Der DAB-Befehl dient zur Korrektur von 4-Bit BCD-Ziffern.
Im Anschluß an eine Byteaddition (ADCB,ADDB) oder Bytesubtraktion (SBCB,SUBB) werden die Operationen unter Nutzung des Half-Carry-Flagbits entsprechend der folgenden Tabelle ausgeführt.

Mnemonik: **DAB dst**
Operation: dst := DA dst
Flags: C: Entsprechend nachstehender Tabelle.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit des Ergebnisses gesetzt ist.

Adressierungsart	Zyklen Byte
dst src mo	
R 10	5

Befehlsformat: F1.1

B: mo110000dst.0000

Befehl	Carry vor DAB	Carry nach DAB	H-Flag vor DAB	Bits (hex) 4-7	Bits (hex) 0-3	Addition hex
ADCB	0	0	0	0-9	0-9	00
ADDB	0	0	0	0-8	A-F	06
	0	0	1	0-9	0-3	06
	0	1	0	A-F	0-9	60
	0	1	0	9-F	A-F	66
	0	1	1	A-F	0-3	66
	1	1	0	0-2	0-9	60
	1	1	0	0-2	A-F	66
SBCB	0	0	0	0-9	0-9	00
SUBB	0	0	1	0-8	6-F	FA
	1	1	0	7-F	0-9	A0
	1	1	1	6-F	6-F	9A

Beispiel:
```
ADDB RH1, RH2
DAB RH1
```
Enthalten die Register RH1=%15 und RH2=%27, ist nach dem Befehl ADDB RH1, RH2 das Ergebnis der Addition in RH1=%3C. Nach dem DAB-Befehl steht im Register RH1 der BCD-Wert 42.

DEC Decrement DEC

Der DEC-Befehl dient dem einfachen Dekrementieren von Zählern.
Der "src"-Operand ist ein Wert im Bereich von 1-16, er wird vom "dst"-Operanden subtrahiert. Ist im Assemblerquellprogramm kein "src"-Operand angegeben, wird der Wert 1 als "src"-Operand eingesetzt. Im Maschinenkode ist der "src"-Operand ein Wert im Bereich von 0-15.

Mnemonik: **DEC dst,src / DECB dst,src**
Operation: dst := dst - src
Flags: Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das Ergebnis negativ ist.
V: Ist nur gesetzt, wenn ein Überlauf auftritt.

Befehlsformat: F1.2 DA: dst=0

W/B: mo10101wdst.src. X : dst<>0

Adressierungsart			Zyklen Wort/Byte		
dst	src	mo	NS	SS	SL
R	IM	10	4	4	4
@R	IM	00	11	11	11
DA	IM	01	13	14	16
X	IM	01	14	14	17

Beispiel: DEC R4, #12 !Dekrementiere den Inhalt des Registers R4 um 12!

DIV Division DIV

Der DIV-Befehl ermöglicht die Division von Integer-Zahlen.
Der "dst"-Operand (Dividend) wird durch den "src"-Operanden (Divisor) dividiert. Das Ergebnis (Quotient) wird im Low-Teil des "dst"-Registers gespeichert, der bei der Division ggf. entstehende Rest wird im High-Teil des "dst"-Registers gespeichert. Die Operanden können vorzeichenbehaftet sein, der Rest besitzt das Vorzeichen des "dst"-Operanden. Der DIV-Befehl benötigt für den "dst"-Operanden ein Langwortregister und für den "src"-Operanden ein Wortregister, der DIVL-Befehl benötigt für den "dst"-Operanden ein Vierfachwortregister und für den "src"-Operanden ein Langwortregister.

Mnemonik: **DIV dst,src / DIVL dst,src**
Operation: dst := dst : src
Flags:
- C: Ist gesetzt, wenn das V-Flag gesetzt ist und der Quotient beim DIV-Befehl eine Zahl von -2^{16} bis $2^{16}-1$ oder beim DIVL-Befehl von -2^{32} bis $2^{32}-1$ ist.
- Z: Ist nur gesetzt, wenn der Quotient oder Divisor Null ist.
- S: Ist nur gesetzt, wenn der Quotient negativ ist, es ist undefiniert, wenn V=1 und C=0 ist.
- V: Ist nur gesetzt, wenn der Divisor Null ist oder wenn der Quotient beim DIV-Befehl außerhalb des Bereichs -2^{15} bis 2^{15} bzw. beim DIVL-Befehl von -2^{31} bis 2^{31} liegt.

Adressierungsart			Zyklen Wort			Zyklen Long		
dst	src	mo	NS	SS	SL	NS	SS	SL
R	R	10	107	107	107	744	744	744
R	IM	00	107	107	107	744	744	744
R	IR	00	107	107	107	744	744	744
R	DA	01	108	108	111	714	725	727
R	X	01	109	109	112	725	725	728
R	R	10	13	13	13	30	30	30
R	IM	00	13	13	13	30	30	30
R	IR	00	13	13	13	30	30	30
R	DA	01	14	15	17	31	32	34
R	X	01	15	15	18	32	32	35
R	R	10	25	25	25	51	51	51
R	IM	00	25	25	25	51	51	51
R	IR	00	25	25	25	51	51	51
R	DA	01	26	27	29	52	53	55
R	X	01	27	27	30	53	53	56

Befehlsformat: F2.1

W: mo011011src.dst.

L: mo011010src.dst.

IM: src=0
IR: src<>0
DA: src=0
X : src<>0

Dieser Bereich gilt, wenn der Divi- Null ist.

Dieser Bereich gilt, wenn der absolute Wert des High-Teils des Dividenden größer als der absolute Wert des Divisors ist.

Beispiel: DIV RR4, R3
Wenn RR4 %000017 und R3 %0016 enthält, steht nach der Division in RR4 %0003 und in R3 %0001.

EXTS Extend Sign EXTS

Der EXTS-Befehl dient der Verarbeitung von mehrfach genauen Zahlen und der Umwandlung von kleinen vorzeichenbehafteten Operanden zu großen vorzeichenbehafteten Operanden.
Das Sign-Flag des Low-Teils des "dst"-Operanden wird in alle Bitpositionen des High-Teils des "dst"-Operanden geladen. Der EXTSB-Befehl benötigt ein Wortregister, der EXTS-Befehl ein Doppelwortregister und der EXTSL-Befehl ein Vierfachwortregister.

Mnemonik: **EXTSB dst / EXTS dst / EXTSL dst**
Operation: Byte : Bit8-15 := Bit7
Wort : Bit16-31 := Bit15
Langwort: Bit32-63 := Bit31

Adressierungsart		Zyklen Wort/Byte Long
dst	mo	
R	10	11

Befehlsformat: F1.1

B: mo110001dst.0000
W: mo110001dst.1010
L: mo110001dst.0111

Beispiel: EXTSB R4
Wenn Register R4 den Wert %00FF enthielt, so steht nach dem EXTSB-Befehl der Wert %FFFF in R4.

INC Increment INC

Der INC-Befehl dient dem einfachen Inkrementieren von Zählern.
Der "src"-Operand ist ein Wert im Bereich von 1-16, er wird zum "dst"-Operanden addiert. Ist im Assemblerquellprogramm kein "src"-Operand angegeben, wird der Wert 1 als "src"-Operand eingesetzt. Im Maschinenkode ist der "src"-Operand ein Wert im Bereich von 0-15.

Mnemonik: **INC dst,src / INCB dst,src**
Operation: dst := dst + src
Flags: Z: ist nur gesetzt, wenn das Ergebnis Null ist.
S: ist nur gesetzt, wenn das Ergebnis negativ ist.
V: ist nur gesetzt, wenn ein Überlauf auftritt.

Befehlsformat: F1.2 DA: dst=0
X : dst<>0

W/B: mo10100wdst.src.

Adressierungsart			Zyklen Wort/Byte		
dst	src	mo	NS	SS	SL
R	IM	10	4	4	4
IR	IM	00	11	11	11
DA	IM	01	13	14	16
X	IM	01	14	14	17

Beispiel:
```
INC R4, #12 !Inkrementiere den Inhalt!
            !des Registers R4 um 12!
```

MULT Multiplikation MULT

Der MULT-Befehl ermöglicht die Multiplikation von Integer-Zahlen.
Der Low-Teil des "dst"-Operanden (Multiplikand) wird mit dem "src"-Operanden (Multiplikator) multipliziert. Das Ergebnis (Produkt) wird im "dst"-Register gespeichert. Die Operanden können vorzeichenbehaftet sein. Der MULT-Befehl benötigt für den "dst"-Operanden ein Langwortregister und für den "src"-Operanden ein Wortregister, der MULTL-Befehl benötigt für den "dst"-Operanden ein Vierfachwortregister und für den "src"-Operanden ein Langwortregister.

Mnemonik: **MULT dst,src / MULTL dst,src**
Operation: dst := dst * src

Flags: C: Ist gesetzt, wenn das Produkt beim MULT-Befehl eine Zahl kleiner als -2^{15} bzw. größer als oder gleich $2^{15}-1$ oder beim MULTL-Befehl kleiner als -2^{31} bzw. größer als oder gleich $2^{31}-1$ ist.
Z: Ist nur gesetzt, wenn das Produkt Null ist.
S: Ist nur gesetzt, wenn das Produkt negativ ist.
V: Ist gelöscht.

Adressierungsart			Zyklen Wort			Zyklen Long		
dst	src	mo	NS	SS	SL	NS	SS	SL
R	R	10	70	70	70	282+7n	282+7n	282+7n
R	IM	00	70	70	70	282+7n	282+7n	282+7n
R	IR	00	70	70	70	282+7n	282+7n	282+7n
R	DA	01	71	72	74	283+7n	284+7n	286+7n
R	X	01	72	72	75	284+7n	284+7n	287+7n

Befehlsformat: F2.1

W: mo011001src.dst.

L: mo011000src.dst.

IM: src=0
IR: src<>0
DA: src=0
X src<>0

Beispiel: MULT RR4, R3
Wenn RR4 %000017 und R3 %0016 enthält, steht nach der Multiplikation in RR4 %00FD.

NEG — Negation — NEG

Der NEG-Befehl ermöglicht Zweierkomplementdarstellung von Zahlen.
Der "dst"-Operand wird in sein Zweierkomplement umgeformt.

Mnemonik: **NEG dst / NEGB dst**
Operation: dst := -src + 1
Flags: C: Ist nur gesetzt, wenn das Ergebnis nicht Null ist.
Z: Ist gesetzt, wenn das Ergebnis Null ist.
V: Ist gesetzt, wenn das Ergebnis für den NEG-Befehl %8000 und für den NEGB-Befehl %80 ist.

Befehlsformat: F1.1 DA: dst=0
X : dst<>0

W/B: mo00110wdst.0010

Adressierungsart		Zyklen Wort/Byte		
dst	mo	NS	SS	SL
R	10	7	7	7
IR	00	12	12	12
DA	01	15	16	18
X	01	16	16	19

Beispiel: NEG R4
Wenn Register R4 den Wert %1111 enthält, steht nach Abarbeitung des Befehls sein Zweierkomplement %EEEF im Register R4.

SBC — Subtraction with Carry — SBC

Der SBC-Befehl realisiert die Subtraktion mit Übertrag.
Der Inhalt des "src"-Operanden und des Carry-Flags werden vom "dst"-Operanden subtrahiert und auf den "dst"-Operanden abgespeichert.

Mnemonik: **SBC dst,src / SBCB dst,src**
Operation: dst := dst - src - C
Flags: C: Ist nur gesetzt, wenn ein Borrow-Übertrag erfolgt.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das Ergebnis negativ ist.
V: Ist nur gesetzt, wenn ein Überlauf auftritt.
D: Beim SBCB-Befehl gesetzt, sonst keine Beeinflussung.
H: Beim SBCB-Befehl gesetzt, sonst keine Beeinflussung.

Befehlsformat: F2.1

W/B: mo11011wsrc.dst.

Adressierungsart			Zyklen Wort/Byte
dst	src	mo	
R	R	10	5

Beispiel: SUB R7, R9
SBC R6, R8
!32-Bit Subtraktion - doppelt genau / Low-Woerter und High-Woerter mit Borrow-Uebertrag der Low-Woerter!

SUB Subtraction SUB

Der SUB-Befehl realisiert die Subtraktion von Operanden.
Der Inhalt des "src"-Operanden wird vom "dst"-Operanden subtrahiert und auf dem "dst"-Operanden abgespeichert.

Mnemonik: **SUB dst,src / SUBB dst,src / SUBL dst,src**
Operation: dst := dst - src
Flags: C: Ist nur gesetzt, wenn ein Borrow-Übertrag erfolgt.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das Ergebnis negativ ist.
V: Ist nur gesetzt, wenn ein Überlauf auftritt.
D: Beim SUBB-Befehl gesetzt, sonst keine Beeinflussung.
H: Beim SUBB-Befehl gesetzt, wenn ein Borrow-Übertrag des Bit 3 erfolgt.

Adressierungsart			Zyklen Wort/Byte			Long		
dst	src	mo	NS	SS	SL	NS	SS	SL
R	R	10	4	4	4	8	8	8
R	IM	00	7	7	7	14	14	14
R	IR	00	7	7	7	14	14	14
R	DA	01	9	10	12	15	16	18
R	X	01	10	10	13	16	16	19

Befehlsformat: F2.1

W/B: mo00001wsrc.dst.

L: mo010010src.dst.

IM: src=0
IR: src<>0
DA: src=0
X : src<>0

Beispiel: SUB R4, @R6
Subtrahiere vom Inhalt des Registers R4 den Inhalt des durch R6 adressierten Speicherplatzes.

3.5.3. Logikbefehle

AND **And** **AND**

Die Inhalte der durch "src" und "dst" adressierten Operanden werden durch eine logische UND-Operation bitweise miteinander verküpft und als "dst"-Operand abgespeichert.
Das Ergebnis einer logischen UND-Operation ist nur 1, wenn beide miteinander verknüpften Bits den Wert 1 haben.

Mnemonik: **AND dst,src / ANDB dst,src**
Operation: dst := dst AND src
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Ergebnis gesetzt ist.
P: Ist nur beim ANDB-Befehl gesetzt, wenn im Ergebnis Parität auftritt.

Adressierungsart			Zyklen Wort/Byte		
dst	src	mo	NS	SS	SL
R	R	10	4	4	4
R	IM	00	7	7	7
R	IR	00	7	7	7
R	DA	01	9	10	12
R	X	01	10	10	13

Befehlsformat: F2.1 IM: src=0

W/B: mo00011wsrc.dst. DA: src=0
X : src<>0

Beispiel: ANDB RH4, RL6
RH4 enthalte %39 (00111001) und RL6 %55 (01010101), nach der ANDB Operation ergibt sich in RH4 das Ergebnis mit %11 (00010001).

COM **Complement** **COM**

Aus dem "dst"-Operanden wird sein Einerkomplement gebildet, d. h., er wird negiert. Jede Eins wird zu Null und jede Null wird zu Eins.

Mnemonik: **COM dst / COMB dst**
Operation: dst := NOT dst
Flags: Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Ergebnis gesetzt ist.
P: Ist nur beim COMB-Befehl gesetzt, wenn im Ergebnis Parität auftritt.

Adressierungsart		Zyklen Wort/Byte		
dst	mo	NS	SS	SL
R	10	7	7	7
IR	00	12	12	12
DA	01	15	16	18
X	01	16	16	19

Befehlsformat: F1.1 DA: src=0
X : src<>0

W/B: mo00110wdst.0000

Beispiel: COMB RH4
Wenn Register RH4 %55 (01010101) enthält, ergibt sich nach dem COMB-Befehl im Register RH4 der Wert %AA (10101010).

OR — Or — OR

Die Inhalte der durch "src" und "dst" adressierten Operanden werden durch eine logische ODER-Operation bitweise miteinander verknüpft.
Das Ergebnis einer logischen ODER-Operation ist 1, wenn mindestens eines der beiden miteinander verknüpften Bits den Wert 1 hat.

Mnemonik: **OR dst,src / ORB dst,src**
Operation: dst := dst OR src
Flags: Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Ergebnis gesetzt ist.
P: Ist nur beim ORB-Befehl gesetzt, wenn im Ergebnis Parität auftritt.

Adressierungsart			Zyklen Wort/Byte		
dst	src	mo	NS	SS	SL
R	R	10	4	4	4
R	IM	00	7	7	7
R	IR	00	7	7	7
R	DA	01	9	10	12
R	X	01	10	10	13

Befehlsformat: F2.1
W/B: mo00010wsrc.dst.

IM: src=0
IR: src<>0
DA: src=0
X : src<>0

Beispiel: ORB RH4, RL6
RH4 enthalte %39 (00111001) und RL6 %55 (01010101), nach der ORB Operation ergibt sich in RH4 das Ergebnis mit %7D (01111101).

TEST — Test — TEST

Der Inhalt des durch "dst" adressierten Operanden wird auf Null getestet. Die Operation entspricht einer OR-Operation des "dst"-Operanden mit Null, der Operand wird nicht verändert.

Mnemonik: **TEST dst / TESTB dst / TESTL dst**
Operation: dst OR 0
Flags: Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Operanden gesetzt ist.
P: Ist nur beim TESTB-Befehl gesetzt, wenn im Ergebnis Parität auftritt.

Adressierungsart		Zyklen Wort/Byte			Zyklen Long		
dst	mo	NS	SS	SL	NS	SS	SL
R	10	7	7	7	13	13	13
IR	00	8	8	8	13	13	13
DA	01	11	12	14	16	17	19
X	01	12	12	15	17	17	20

Befehlsformat: F1.1
W/B: mo00110wdst.0100
L: mo011100dst.1000

DA: dst=0
X : dst<>0

Beispiel: TESTB RH1
Wenn RH1 %F0 enthält, sind nach dem TESTB-Befehl das S- und P-Flag gesetzt. Das Z-Flag ist gelöscht.

XOR Exclusive Or XOR

Die Inhalte der durch "src" und "dst" adressierten Operanden werden durch eine logische EXCLUSIV-ODER-Operation bitweise miteinander verknüpft. Das Ergebnis einer logischen XOR-Operation ist 1, wenn die beiden miteinander verknüpften Bits nicht den gleichen Wert haben. Es ist Null, wenn sie den gleichen Wert besitzen.

Mnemonik: **XOR dst,src / XORB dst,src**
Operation: dst := dst XOR src
Flags: Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Ergebnis gesetzt ist.
P: Ist nur beim XORB-Befehl gesetzt, wenn im Ergebnis Parität auftritt.

Adressierungsart			Zyklen Wort/Byte		
dst	src	mo	NS	SS	SL
R	R	10	4	4	4
R	IM	00	7	7	7
R	IR	00	7	7	7
R	DA	01	9	10	12
R	X	01	10	10	13

Befehlsformat: F2.1

W/B: mo00100wsrc.dst.

IM: src=0
IR: src<>0
DA: src=0
X src<>0

Beispiel: XORB RH4, RL6
Das Byteregister RH4 enthalte %39 (00111001) und RL6 %55 (01010101), nach der XORB-Operation ergibt sich im Byteregister RH4 das Ergebnis mit %6C (01101100).

3.5.4. Programmsteuerbefehle

CALL **Call Procedure** **CALL**

Der Inhalt des PC (Befehlszähler) enthält die Adresse des nach dem CALL-Befehl stehenden Befehls und wird in den Stackbereich gespeichert. Anschließend wird die im Befehl angegebene Adresse zur Abarbeitung des ersten Befehls der Prozedur in den PC geladen.
Die Rückkehradresse kann mit einem RET-Befehl am Ende der Prozedur wieder in den PC geladen werden.

Mnemonik: **CALL dst**

Operation:

	nichtsegmentiert	segmentiert
	PC := SP - 2	PC := SP - 4
	@SP := PC	@SP := PC
	PC := dst	PC := dst
	(nichtsegmentiert)	(segmentiert)

Befehlsformat: F1.1

W/B: mo011111dst.0000

DA: dst=0
X : dst<>0
DA: src=0

Adressierungsart		Zyklen
dst	mo	NS SS SL
IR	00	10 15 15
DA	01	12 18 20
X	01	13 18 21

Beispiel: CALL @R4
Register R4 enthalte den Wert %5000, R15 zeige auf %1000, und der CALL-Befehl stehe auf %2000, dann steht nach dem CALL-Befehl auf der Adresse %0FFE die Rückkehradresse %2004, und der PC enthält den Wert %5000!

CALR **Call Procedure Relative** **CALR**

Der Inhalt des PC (Befehlszähler) enthält die Adresse des nach dem CALR-Befehl stehenden Befehls und wird in den Stackbereich gespeichert. Anschließend wird die Adresse des ersten Befehls der Prozedur aus der im Befehl angegebenen Distanz (Displacement) und dem Inhalt des PC berechnet und in den PC geladen. Die Distanz ist ein 12-Bit Wert. Die Adresse muß dementsprechend, ausgehend von der CALR-Adresse, im Bereich von -4092 bis + 4098 Byte liegen.
Die Rückkehradresse kann mit einem RET-Befehl in der Prozedur wieder in den PC geladen werden.

Mnemonik: **CALR dst**

Operation:

	nichtsegmentiert	segmentiert
	PC := SP - 2	PC := SP - 4
	@SP := PC	@SP := PC
	PC := PC - (2 * Displacement)	PC := PC - (2 * Displacement)

Befehlsformat: F3.4

1101Displacement

Adressierungsart	Zyklen
dst	NS SS SL
RA	10 15 15

Beispiel:

```
LOC  OBJ CODE     SOURCE STATEMENT
1000 DFFF         CALR BTOH
1002 BE08         RLDB RL0, RH0
1004 93F1 BTOH:   PUSH @R15, R1
```

DJNZ — Decrement and Jump if NOT Zero — DJNZ

Der Befehl ist besonders zum Aufbau von Schleifen geeignet.
Das im Befehl angegebene Zählregister wird bei jeder Befehlsabarbeitung dekrementiert. Solange sein Inhalt <> 0 ist, wird die Programmabarbeitung an der Zieladresse fortgesetzt. Ist der Inhalt des Zählregisters = 0 wird die Befehlsabarbeitung mit dem auf den DJNZ- bzw. DBJNZ-Befehl folgenden Befehl fortgesetzt. Beim DJNZ-Befehl ist das Zählregister ein Wortregister, beim DBJNZ-Befehl ist es ein Byteregister. Die Distanz ist ein positiver 7-Bit Wert, damit kann die Zieladresse im Bereich von -252 bis +2 Byte vom DJNZ- bzw. DBJNZ-Befehl liegen. Mit beiden Befehlen können nur rückwärts gerichtete Adressen erreicht werden.

Mnemonik: **DJNZ r,dst / DBJNZ r,dst**
Operation: r := r - 1
r<>0 PC := PC - (2* disp)
r=0 PC := PC + 2

Adressierungsart	Zyklen Wort/Byte
dst	
RA	11

Befehlsformat: F3.1

W/B: 1111dst.Displac.

Beispiel:
```
        LD    R4, #1000
WAIT:   DJNZ  R4, WAIT
        !2,75 Millisekunden warten!
```

IRET — Interrupt Return — (PB) — IRET

Mit dem IRET-Befehl werden Interrupt- und Trap-Serviceroutinen beendet. Das im Stack gespeicherte Kennwort des Interrupts oder Traps wird aus dem Stack rückgelesen und verworfen. Anschließend wird das Flag- und Controlwort und der vor der Unterbrechung gerettete Inhalt des Befehlszählers in die Programmstatusregister der CPU geladen. Die U8001-CPU kann den IRET-Befehl nur im segmentierten Mode ausführen.

Mnemonik: IRET
Operation: SP := SP + 2 (nichtsegmentiert) SP := SP + 2 (segmentiert)
PS := @SP (nichtsegmentiert) PS := @SP (segmentiert)
SP := SP + 4 (nichtsegmentiert) SP := SP + 6 (segmentiert)
Flags: Das Flag- und Controlwort wird mit dem im Stack gespeicherten Programmstatus vor der Unterbrechung geladen.

Zyklen		
NS	SS	SL
13		16

Befehlsformat: F9.3

0111101100000000

Beispiel:
```
LDCTL R4,  FCW
SET   R4, #15
LDCTL FCW, R4
IRET
```

Der IRET-Befehl darf bei der U8001-CPU nur im segmentierten Mode ausgeführt werden, das Beispiel zeigt den Abschluß einer Interrupt-Serviceroutine.

JP Jump JP

Der Inhalt des PC (Befehlszähler) ergibt sich bei einem unbedingten JP-Befehl (cc=1000) aus der im Befehl als "dst"-Operand angegebenen Adresse. Bei einem bedingten JP-Befehl ergibt sich der Inhalt des PC nur aus der mit "dst" angegebenen Adresse, wenn die Bedingung erfüllt ist (s. Abschnitt 3.2.0.), andernfalls wird die Programmabarbeitung mit dem nächsten Befehl fortgesetzt.

Mnemonik: **JP cc,dst**
Operation: cc = 1 : PC := dst
cc = 0 : PC := nächstem Befehl

Adressierungsart		Zyklen					
		JP cc=1			JP cc=0		
dst	mo	NS	SS	SL	NS	SS	SL
IR	00	10	15	15	7	7	7
DA	01	7	8	10	7	8	10
X	01	8	8	11	8	8	11

Befehlsformat: F1.3

mo011110dst..cc.

DA: dst=0
X dst<>0
DA: src=0

Beispiel: TEST R1
JP Z, %2000
Enthält Register R1 Null, wird durch den TEST-Befehl das Zero-Flag gesetzt und der Sprung zur Adresse %2000 ausgeführt.

JR Jump Relative JR

Der PC (Befehlszähler) wird beim JR-Befehl berechnet indem das im Befehl angegebene Displacement (Distanz) mit zwei multipliziert und mit dem auf den nächsten Befehl weisenden aktuellen Wert des PC addiert wird.
Der PC wird bei einem unbedingten JP-Befehl (cc=1000) mit der berechneten Adresse geladen. Ein bedingter JR-Befehl führt nur zu der mit "dst" angegebenen Adresse wenn die Bedingung erfüllt ist (s. Abschnitt 3.2.0.), andernfalls wird die Programmabarbeitung mit dem nächsten Befehl fortgesetzt. Das Displacement ist ein vorzeichenbehafteter 8-Bit-Wert im Bereich von -128 bis +127. Somit kann die Zieladresse -254 oder +256 Byte vom JR-Befehl entfernt sein. Der JR-Befehl kann im segmentierten Mode nur innerhalb eines Segments angewandt werden.

Mnemonik: **JR cc,dst**
Operation: cc = 1 : PC := PC + (2 * disp)
cc = 0 PC := nächstem Befehl

Adressierungsart	Zyklen	
	JR cc=1	JR cc=0
dst		
RA	6	6

Befehlsformat: F3.3

1110.cc.Displace

Beispiel:
```
LOC  OBJ CODE  SOURCE STATEMENT
1000 E605      JR Z, $ + 10
1002 8D34      TEST R3
```
Wenn das Zero-Flag gesetzt ist, wird die Programmabarbeitung auf der Adresse %100C fortgesetzt.

RET — Return from Procedure — RET

Der RET-Befehl dient nach der Abarbeitung von Prozeduren, die mit einem CALL- oder CALR-Befehl aktiviert wurden zur Rückkehr in das rufende Programm. Wenn die mit "cc" angegebene Bedingung (s. Abschnitt 3.2.) wahr ist, wird die Rückkehradresse aus dem Stackbereich in den PC (Befehlszähler) gelesen. Ist die Bedingung nicht erfüllt wird die Programmabarbeitung mit dem nächsten Befehl fortgesetzt.

Mnemonik: **RET cc**

Operation:

	nichtsegmentiert	segmentiert
cc = 1	PC := @SP SP := SP+2	PC := @SP SP := SP+4
cc = 0	PC := nächstem Befehl	PC := nächstem Befehl

Befehlsformat: F9.4

100111100000.cc.

Zyklen	
RET cc=1	RET cc=0
NS SS SL	NS SS SL
10 13 13	7 7 7

Beispiel: RET Z
Wenn im nichtsegmentierten Mode R15 %1000 enthält und im Stackbereich auf %1000 die Rückkehradresse %3020 steht, wird unter der Bedingung Zero=1 die Programmabarbeitung auf der Adresse %3020 fortgesetzt. R15 wird zweimal inkrementiert und zeigt auf %1002.

SC — System Call — SC

Der SC-Befehl dient der Nutzung von Systemfunktionen des Betriebssystems durch gezielte Auslösung des System-Call-Trap. Der PC (Befehlszähler), das Flag- und Controlwort (FCW) und als Kennwort der SC-Befehl werden in den Stack geladen (Bild 2.11). Die Nummer des System Call steht auf dem niederwertigen Byte des Kennworts, die im Stack als PC gespeicherte Adresse zeigt auf den nach dem SC-Befehl stehenden Befehl. Der "src"-Operand ist eine 8-Bit-Wert, es können 256 System-Call-Routinen implementiert werden. Der Programmstatus der System-Call-Routine wird aus der Programstatusarea gelesen. Die System-Call-Routine muß mit einem IRET-Befehl beendet werden.

Mnemonik: **SC src**

Operation:

	nichtsegmentiert	segmentiert
	SP := SP - 4	SP := SP - 6
	@SP := PS	@SP := PS
	SP := SP - 2	SP := SP - 2
	@SP := SC-Bef.	@SP := SC-Bef.
	PS := Programmstatusarea	PS := Programmstatusarea

Befehlsformat: F9.5

01111111..src...

Zyklen
NS SS SL
33 39 39

Beispiel: SC #3
Der System-Call-Trap wird ausgelöst. In der Trap-Routine kann, z.B. im nichtsegmentierten Mode, die Nummer des System-Call mit dem Befehl LDB RL2, @R15(1) gelesen werden.

3.5.5. Bitmanipulationsbefehle

BIT **Bit Test** **BIT**

Der BIT-Befehl testet den Inhalt des adressierten Bits im "dst"-Operanden und setzt das Zero-Flag, wenn das Bit Null ist, andernfalls wird das Flag rückgesetzt. Der "dst"-Operand ist eine Wort- bzw. Byteadresse, der "src"-Operand ist die Bitadresse. Sie ist für den BITB-Befehl eine Zahl von 0 bis 7 und für den BIT-Befehl eine Zahl von 0 bis 15.
Steht der "src"-Operand in einem Register, so muß er in den vier (BIT) bzw. drei (BITB) niederwertigen Bits kodiert werden.

Mnemonik: **.BIT dst,src / BITB dst,src**
Operation: Z := NOT dst(src)
Flags: Z := ist nur gesetzt, wenn das adressierte Bit Null ist.

Adressierungsart			Zyklen Wort/Byte		
dst	src	mo	NS	SS	SL
R	IM	10	4	4	4
IR	IM	00	8	8	8
DA	IM	01	10	11	13
X	IM	01	11	11	14
R	R		10	10	10

Befehlsformat: F1.2 IR: dst<>0
DA: dst=0
W/B: mo10011wdst.src. X : dst<>0

Befehlsformat: F6.3 R : src

W/B: 0010011w0000src.
0000dst.00000000

Beispiel: BIT R2, #14
Wenn Register R2 den Wert 1000000111110000 enthält, wird das Zero-Flag gesetzt.

RES **Reset Bit** **RES**

Der RES-Befehl setzt den Inhalt des adressierten Bits im "dst"-Operanden zu Null. Der "dst"-Operand ist eine Wort- bzw. Byteadresse, der "src"-Operand ist die Bitadresse. Sie ist für den RESB-Befehl eine Zahl von 0 bis 7 und für den RES-Befehl eine Zahl von 0 bis 15.
Steht der "src"-Operand in einem Register, so muß er in den vier (RES) bzw. drei (RESB) niederwertigen Bits kodiert werden.

Mnemonik: **RES dst,src / RESB dst,src**
Operation: dst(src) := 0

Adressierungsart			Zyklen Wort/Byte		
dst	src	mo	NS	SS	SL
R	IM	10	4	4	4
IR	IM	00	11	11	11
DA	IM	01	13	14	16
X	IM	01	14	14	17
R	R		10	10	10

Befehlsformat: F1.2 IR: dst<>0
DA: dst=0
W/B: mo10001wdst.src. X : dst<>0

Befehlsformat: F6.3 R : src

W/B: 0010001w0000src.
0000dst.00000000

Beispiel: RES R2, #14
Wenn Register R2 den Wert 1100000111110000 enthält, steht nach dem RES-Befehl 1000000111110000 in R2.

SET — Set Bit — SET

Der SET-Befehl setzt den Inhalt des adressierten Bits im "dst"-Operanden zu Eins. Der "dst"-Operand ist eine Wort- bzw. Byteadresse, der "src"-Operand ist die Bitadresse. Sie ist für den SETB-Befehl eine Zahl von 0 bis 7 und für den SET Befehl eine Zahl von 0 bis 15.
Steht der "src"-Operand in einem Register, so muß er in den vier (SET) bzw. drei (SETB) niederwertigen Bits kodiert werden.

Mnemonik: **SET dst,src / SETB dst,src**
Operation: dst(src) := 1

Adressierungsart			Zyklen Wort/Byte		
dst	src	mo	NS	SS	SL
R	IM	10	4	4	4
IR	IM	00	11	11	11
DA	IM	01	13	14	16
X	IM	01	14	14	17
R	R		10	10	10

Befehlsformat: F1.2

W/B: mo10010wdst.src.

Befehlsformat: F6.3

W/B: 0010010w0000src.
0000dst.00000000

IR: dst<>0
DA: dst=0
X : dst<>0
R : src

Beispiel: SET R2, #14
Wenn Register R2 den Wert 100000011110000 enthält, steht nach dem SET-Befehl 110000011110000 in R2.

TCC — Test Condition Code — TCC

Mit diesem Befehl kann der Inhalt eines Flag-Bits in einen binären Wert kopiert werden.
Ist das mit "cc" spezifizierte Flag-Bit gesetzt, dann wird es (s. Abschnitt 3.2.) in das "lsb"-Bit des "dst"-Operanden kopiert. Die restlichen Bits des "dst"-Operanden werden nicht beeinflußt.
Ist das mit "cc" spezifizierte Flag-Bit nicht gesetzt, dann bleibt der Wert des "lsb"-Bit im "dst"-Operanden bestehen.

Mnemonik: **TCC cc,dst / TCCB cc,dst**
Operation: cc = 1 : dst (0) := 1
cc = 0 : dst (0) := 0

Adressierungsart		Zyklen Wort/Byte
dst	mo	
R	10	5

Befehlsformat: F1.4

W/B: mo10111wdst..cc.

Beispiel: TCC MI, R4
JP MI, MARK1
Steht im Register R4 der Wert 1111000011110000 und ist das Sign-Flag gesetzt, so ergibt sich R4 zu 1111000011110001.

TSET Test and Set TSET

Der TSET-Befehl testet den Inhalt des "msb"-Bits im "dst"-Operanden und kopiert seinen Inhalt in das Sign-Flag. Ist das "msb"-Bit Null, werden alle Bits des "dst"-Operanden zu Eins gesetzt.
Der Befehl hat besondere Bedeutung in Echtzeitsystemen, er dient zur Vereinfachung der Arbeit mit Semaphoren.
Zwischen dem Laden des Befehls und dem Abspeichern des "dst"-Operanden wird keine BUSREQ Anforderung angenommen. Damit wird gesichert, daß dieser Vorgang nicht unterbrochen werden kann.

Mnemonik: **TSET dst / TSETB dst**
Operation: S := dst(msb)
S = 0 : dst := 11111111

Adressierungsart		Zyklen Wort/Byte		
dst	mo	NS	SS	SL
R	10	7	7	7
IR	00	11	11	11
DA	01	14	15	17
X	01	15	15	18

Befehlsformat: F1.1
W/B: mo00110wdst.0110

DA: dst=0
X : dst<>0

Beispiel:
```
ENTRY: TSET BOX       !Ressource frei ? !
       JR MI, ENTRY   !Sprung: nicht frei!
                      !Zugriff auf die Ressource!
       CLR BOX        !Ressource freigeben  !
```

Erst wenn der Semaphor BOX Null ist kann auf die gemeinsame Ressource zugegriffen werden. Nach Abarbeitung der Routine wird BOX wieder zu Null gesetzt und gibt damit die Ressource wieder frei.

3.5.6. Rotations- und Verschiebebefehl

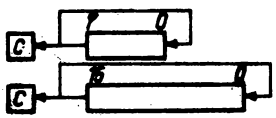
RL **Rotate Left** RL

Der RL-Befehl rotiert den Inhalt des "dst"-Operanden um eine Bitposition nach links. Die Rotation kann ein- oder zweimal erfolgen, ist kein "src"-Operand angegeben bzw. ist er Eins, wird der "dst"-Operand einmal rotiert, ist der "src"-Operand Zwei, wird er zweimal rotiert. Die Anzahl der Rotationen wird im Bit 1 des Befehlskodes angegeben, einmal s=0, zweimal s=1. Bei der Rotation wird das jeweilige "msb"-Bit in das Carry-Flag geschoben.

Mnemonik: **RL dst,src / RLB dst,src**
Operation: C := dst(msb) dst(0) := dst(msb) ; dst(n+1) := dst(n)
Flags: C: Ist gesetzt, wenn das zuletzt rotierte "msb"-Bit = 1 ist.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Ergebnis gesetzt ist.
V: Ist nur gesetzt, wenn das Vorzeichen des "dst"-Operanden während der Rotation wechselt.

Adressierungsart	Zyklen Wort/Byte	
dst	src	mo	s=0	s=1
R	IM	10	6	7

Befehlsformat: F1.1 Wort:



W/B: mo11001wdst.10s0 Byte:

Beispiel: RLB RH7, #2
Wenn das Byteregister RH7 den Wert 11110000 enthält, so steht nach der Rotation in RH7 der Wert 11000011 Das Carry-Flag und das Sign-Flag ist 1.

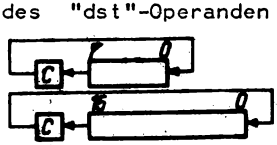
RLC **Rotate Left through Carry** RLC

Der RLC-Befehl rotiert den Inhalt des "dst"-Operanden und des Carry-Flags um eine Bitposition nach links. Die Rotation kann ein- oder zweimal erfolgen, ist kein "src"-Operand angegeben bzw. ist er Eins wird der "dst"-Operand einmal rotiert, ist der "src"-Operand Zwei wird er zweimal rotiert. Die Anzahl der Rotationen wird im Bit 1 des Befehlskodes angegeben, einmal s=0, zweimal s=1. Bei der Rotation wird das jeweilige "msb"-Bit in das Carry-Flag geschoben und das Carry-Flag in die Position des Bit 0.

Mnemonik: **RLC dst,src / RLCB dst,src**
Operation: dst(0) := C ; C := dst(msb) ; dst(n+1) := dst(n)
Flags: C: Ist gesetzt, wenn das zuletzt rotierte "msb"-Bit = 1 ist.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Ergebnis gesetzt ist.
V: Ist nur gesetzt, wenn das Vorzeichen des "dst"-Operanden während der Rotation wechselt.

Adressierungsart	Zyklen Wort/Byte	
dst	src	mo	s=0	s=1
R	IM	10	6	7

Befehlsformat: F1.1 Wort:



W/B: mo11001wdst.00s0 Byte:

Beispiel: RLC RL7, #2
Wenn das Byteregister RL7 den Wert 11110000 enthält und C=0 ist, so steht nach der Rotation in RL7 11000001. Das Carry- und das Sign-Flag ist 1.

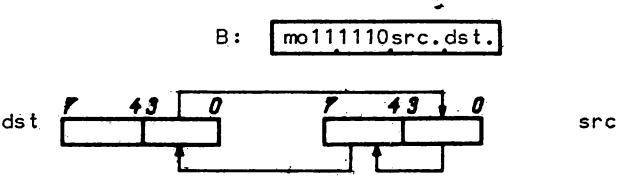
RLDB — Rotate Left Digit — RLDB

Der RLDB-Befehl rotiert BCD-Zahlen zwischen dem "src"- und "dst"-Register nach links. Die Rotation erfolgt entsprechend der Abbildung. Dieser Befehl wird in der BCD-Arithmetik zur Verschiebung von BCD-Zahlenketten nach links benutzt. Diese Operation entspricht einer Multiplikation der BCD-Zahlenreihe mit 10.

Mnemonik: **RLDB dst,src**
Operation: src(0,3) := dst(0,3)
src(4,7) := src(0,3)
dst(0,3) := src(4,7)
Flags: Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Ergebnis gesetzt ist.

Adressierungsart			Zyklen Byte
dst	src	mo	
R	R	10	9

Befehlsformat: F2.1



Beispiel:

```
MUL10:  LD    R5, #2        !Zaehler, 4stellige BCD-Zahl!
        LD    R6, #%1001    !Zeiger zur niederwertigen BCD-Zahl!
        CLRB  RH4           !Neue Einerposition der BCD-Zahl bilden!
MULT:
        LDB   RL4, @R6      !Zwei BCD-Zahlen laden!
        RLDB  RH4, RL4      !BCD-Zahlen nach links rotieren!
        LDB   @R6, RL4      !BCD-Zahl := RH4(0,3) und RL4(0,3)!
        DEC   R6            !Zeiger dekrementieren!
        DJNZ  R5, MULT      !Ruecksprung zur Bearbeitung der!
                            !hoeherwertigen BCD-Zahlen!
```

Adresse	Speicher vorher Daten binär	BCD-Zahl	Speicher nachher Daten binär	BCD-Zahl
1000	00000000	00	00000100	04
1001	01000011	43	00110000	30

Das Beispiel zeigt die Multiplikation einer BCD-Zahl mit 10. Die Zahl besitzt, auf der Adresse 1000 beginnend, 4 Positionen: Tausender, Hunderter, Zehner und Einer. Die ursprüngliche BCD-Zahl 0043 wurde mit 10 multipliziert und steht als Zahl 0430 auf dem ursprünglichen Speicherplatz. Das Beispiel läßt sich beliebig auf größere BCD-Zahlenreihen oder die Multiplikation mit größeren Zehnerpotenzen erweitern.

RR Rotate Right RR

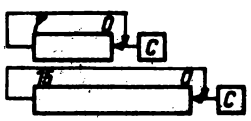
Der RR-Befehl rotiert den Inhalt des "dst"-Operanden um eine Bitposition nach rechts. Die Rotation kann ein- oder zweimal erfolgen, ist kein "src"-Operand angegeben bzw. ist er Eins, wird der "dst"-Operand einmal rotiert, ist der "src"-Operand Zwei, wird er zweimal rotiert. Die Anzahl der Rotationen wird im Bit 1 des Befehlskodes angegeben, einmal s=0, zweimal s=1. Bei der Rotation wird der Inhalt des Bit 0 in das Carry-Flag geschoben.

Mnemonik: **RR dst,src / RRB dst,src**
Operation: C := dst(0) ; dst(msb) := dst(0) ; dst(n) := dst(n+1)
Flags: C: Ist gesetzt, wenn das zuletzt rotierte "lsb"-Bit = 1 ist.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Ergebnis gesetzt ist.
V: Ist nur gesetzt, wenn das Vorzeichen des "dst"-Operanden während der Rotation wechselt.

Adressierungsart	Zyklen Wort/Byte	
dst src mo	s=0	s=1
R IM 10	6	7

Befehlsformat: F1.1 Wort:

W/B: mo11001wdst.11s0 Byte:



Beispiel: RRB RH4, #2
Wenn das Byteregister RH4 den Wert 11110000 enthält, so steht nach der Rotation in RH4 der Wert 11000011. Das Carry-Flag und das Sign-Flag ist 1.

RRC Rotate Right through Carry RRC

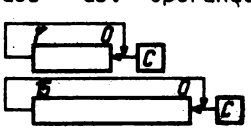
Der RRC-Befehl rotiert den Inhalt des "dst"-Operanden und des Carry-Flags um eine Bitposition nach rechts. Die Rotation kann ein- oder zweimal erfolgen, ist kein "src"-Operand angegeben bzw. ist er Eins, wird der "dst"-Operand einmal rotiert, ist der "src"-Operand Zwei, wird er zweimal rotiert. Die Anzahl der Rotationen wird im Bit 1 des Befehlskodes angegeben, einmal s=0, zweimal s=1. Bei der Rotation wird das jeweilige "lsb"-Bit in das Carry-Flag und das Carry-Flag in das "msb"-Bit geschoben.

Mnemonik: **RRC dst,src / RRCB dst,src**
Operation: dst(msb) := C C := dst(0) ; dst(n) := dst(n+1)
Flags: C: Ist gesetzt, wenn das zuletzt rotierte "msb"-Bit = 1 ist.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Ergebnis gesetzt ist.
V: Ist nur gesetzt, wenn das Vorzeichen des "dst"-Operanden während der Rotation wechselt.

Adressierungsart	Zyklen Wort/Byte	
dst src mo	s=0	s=1
R IM 10	6	7

Befehlsformat: F1.1 Wort:

W/B: mo11001wdst.01s0 Byte:



Beispiel: RRC R7, #2
Wenn Register R7 den Wert 1111000011110000 enthält und C=0 ist, so steht nach der Rotation in R7 der Wert 0011110000111100. Das Carry- und das Sign-Flag sind rückgesetzt.

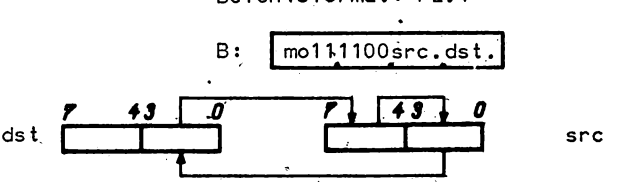
RRDB Rotate Right Digit RRDB

Der RRDB-Befehl rotiert BCD-Zahlen zwischen dem "src"- und "dst"-Register nach rechts. Die Rotation erfolgt entsprechend der Abbildung. Dieser Befehl wird in der BCD-Arithmetik zur Verschiebung von BCD-Zahlenketten nach rechts benutzt. Die Operation entspricht einer Division der BCD-Zahl mit 10.

Mnemonik: **RRDB dst,src**
Operation: src(4,7) := dst(0,3)
src(0,3) := src(4,7)
dst(0,3) := src(0,3)
Flags: Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Ergebnis gesetzt ist.

Adressierungsart			Zyklen Byte
dst	src	mo	
R	R	10	9

Befehlsformat: F2.1



Beispiel:

```
DIV10:  LD    R5, #2        !Zaehler, 4stellige BCD-Zahl!
        LD    R6, #%1000    !Zeiger zur hoeherwertigen BCD-Zahl!
        CLRB  RH4           !Neue Tausender Position der BCD-Zahl bilden!
DIV:
        LDB   RR4, @R6      !Zwei BCD-Zahlen laden!
        RRDB  RH4, RR4      !BCD-Zahlen nach rechts rotieren!
        LDB   @R6, RR4      !BCD-Zahl := RH4(0,3) und RL4(0,3)!
        INC   R6            !Zeiger inkrementieren!
        DJNZ  R5, DIV       !Ruecksprung zur Bearbeitung der!
                            !hoeherwertigen BCD-Zahlen!
```

Adresse	Speicher vorher Daten binär	BCD-Zahl	Speicher nachher Daten binär	BCD-Zahl
1000	01010100	54	00000101	05
1001	00110000	30	01000011	43

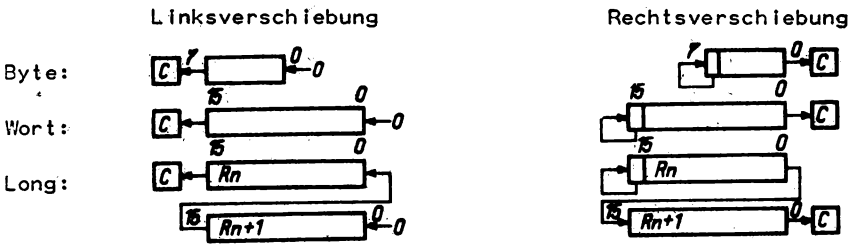
Das Beispiel zeigt die Division einer BCD-Zahl durch 10. Die Zahl besitzt, auf der Adresse 1000 beginnend, 4 Positionen: Tausender, Hunderter, Zehner und Einer. Die ursprüngliche BCD-Zahl 5430 wurde durch 10 dividiert und steht als Zahl 0543 auf dem ursprünglichen Speicherplatz. Das Beispiel ist beliebig auf größere BCD-Zahlenreihen oder die Division mit größeren Zehnerpotenzen erweiterbar.

SDA Shift Dynamic Arithmetic SDA

Der "dst"-Operand wird arithmetisch nach rechts oder links um die Anzahl der im "src"-Operanden angegebenen Bitpositionen verschoben. Der "src"-Operand muß in einem Wortregister stehen. Ist der "src"-Operand eine positive Zahl ("msb"-Bit = 0), erfolgt eine Linksverschiebung, ist er eine negative Zahl ("msb"-Bit = 1), erfolgt eine Rechtsverschiebung. Die Anzahl der Verschiebeoperationen ist befehlsabhängig:

SDAB : -8 bis +8
SDA : -16 bis +16
SDAL : -32 bis +32

Bei der Rechtsverschiebung wird das Sign-Flag rückgesetzt, das Carry-Flag ergibt sich bei der letzten Verschiebung aus dem "lsb"-Bit des "dst"-Operanden. Bei der Linksverschiebung wird in das "lsb"-Bit eine 0 geschoben, das Carry-Flag wird aus dem "msb"-Bit des "dst"-Operanden geladen. Eine Verschiebung mit dem "src"-Operanden Null führt zum Setzen der Flags.



Mnemonik: **SDA dst,src / SDAB dst,src / SDAL dst,src**

Operation: Rechts: dst(msb) := dst(msb)
dst(n) := dst(n+1)
C := dst(0)
Links: dst(0) := 0
dst(n+1) := dst(n)
C := dst(msb)

Flags:
C: Ist nur gesetzt, wenn das zuletzt aus dem "dst"-Operanden geschobene Bit = 1 ist.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das Ergebnis negativ ist.
V: Ist nur gesetzt, wenn das Vorzeichen des "dst"-Operanden während der Rotation wechselt.

Adressierungsart	Zyklen Wort/Byte Long
dst src mo	
R. R 10	13+3n

Befehlsformat: F6.6

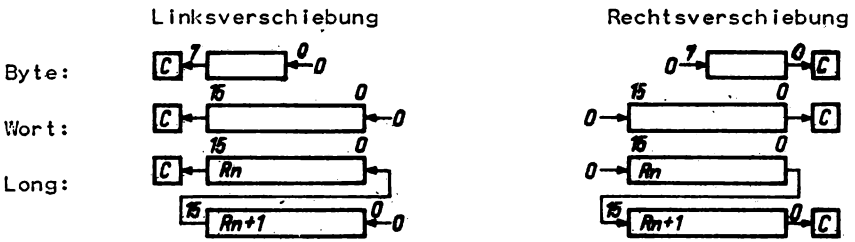
W/B: mo11001wdst.1011 0000src.00000000

L: mo11001ldst.1111 0000src.00000000

Beispiel: SDA R4, R7
Wenn Register R4 als "dst"-Operand den Wert 1111101010111100 und Register R7 als "src"-Operand den Wert 0000000000000010 enthält, erfolgt eine Linksverschiebung um 2 Bit. Nach der Verschiebung ist das Carry- und das Sign-Flag gesetzt, und in R4 steht 11101011110000.

SDL Shift Dynamic Logical SDL

Der "dst"-Operand wird logisch nach rechts oder links um die Anzahl der im "src"-Operanden angegebenen Bitpositionen verschoben. Der "src"-Operand muß in einem Wortregister stehen. Ist der "src"-Operand eine positive Zahl ("msb"-Bit = 0), erfolgt eine Linksverschiebung, ist er eine negative Zahl ("msb"-Bit = 1), erfolgt eine Rechtsverschiebung.
Die Anzahl der Verschiebeoperationen ist befehlsabhängig:
SDLB : -8 bis +8
SDL : -16 bis +16
SDLL : -32 bis +32
Bei der Rechtsverschiebung wird in das "msb"-Bit eine Null geschoben, das Carry-Flag ergibt sich bei der letzten Verschiebung aus dem "lsb"-Bit des "dst"-Operanden. Bei der Linksverschiebung wird in das "lsb"-Bit eine 0 geschoben, das Carry-Flag wird aus dem "msb"-Bit des "dst"-Operanden geladen. Eine Verschiebung mit dem "src"-Operanden Null führt nur zum Setzen der Flags.



Mnemonik: **SDL dst,src / SDLB dst,src / SDLL dst,src**
Operation: Rechts: dst(msb) := 0
dst(n) := dst(n+1)
C := dst(0)
Links: dst(0) := 0
dst(n+1) := dst(n)
C := dst(msb)
Flags: C: Ist nur gesetzt, wenn das zuletzt aus dem "dst"-Operanden geschobene Bit = 1 ist.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das "msb"-Bit im Ergebnis gesetzt ist.
V: Undefiniert.

Adressierungsart			Zyklen Wort/Byte Long
dst	src	mo	
R	R	10	15+3n

Befehlsformat: F6.6

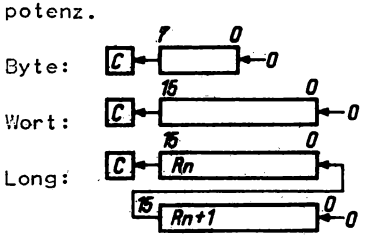
W/B: mo11001wdst.0011
0000src.00000000

L: mo110011dst.0111
0000src.00000000

Beispiel: SDLB RH4, R7
Wenn Register RH4 als "dst"-Operand den Wert 11011100 und Register R7 als "src"-Operand den Wert 0000000000000010 enthält, erfolgt eine Linksverschiebung um 2 Bit. Nach der Verschiebung ist das Carry-Flag gesetzt und das Sign-Flag rückgesetzt, in RH4 steht 01110000.

SLA **Shift Left Arithmetic** SLA

Der "dst"-Operand wird arithmetisch nach links um die Anzahl der im "src"-Operanden angegebenen Bitpositionen verschoben. Der "src"-Operand ist ein Festwert; wird er nicht angegeben, ist als "src"-Operand Eins vorgegeben. Die Anzahl der Verschiebeoperationen ist befehlsabhängig:
SLAB : 0 bis 8
SLA : 0 bis 16
SLAL : 0 bis 32
Das Carry-Flag ergibt sich bei der letzten Verschiebung aus dem "msb"-Bit des "dst"-Operanden. In das "lsb"-Bit wird eine 0 geschoben.
Eine Verschiebung mit dem "src"-Operanden Null führt nur zum Setzen der Flags. Diese Operation entspricht einer vorzeichenbehafteten Multiplikation des "dst"-Operanden mit einer im "src"-Operanden angegebenen Zweierpotenz.



Mnemonik: **SLA dst,src / SLAB dst,src / SLAL dst,src**
Operation: dst(0) := 0
dst(n+1) := dst(n)
C := dst(msb)
Flags: C: Ist nur gesetzt, wenn das zuletzt aus dem "dst"-Operanden geschobene Bit = 1 ist.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das Ergebnis negativ ist.
V: Ist nur gesetzt, wenn das Vorzeichen des "dst"-Operanden während der Rotation wechselt.

Adressierungsart	Zyklen Wort/Byte Long
dst src mo	
R R 10	13+3n

Befehlsformat: F5.1

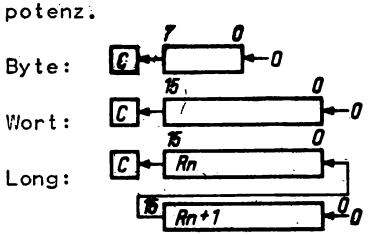
W/B: mo11001wdst.1001
.....src.....

L: mo11001ldst.1101
.....src.....

Beispiel: SLA RH4, #2
Wenn Register R4 als "dst"-Operand den Wert 10111100 und der "src"-Operand den Wert 00000010 enthält, erfolgt eine Linksverschiebung um 2 Bit. Nach der Verschiebung enthält RH4 den Wert 11110000. Das Sign-Flag ist gesetzt, das Carry-Flag rückgesetzt.

SLL Shift Left Logical SLL

Der "dst"-Operand wird logisch nach links um die Anzahl der im "src"-Operanden angegebenen Bitpositionen verschoben. Der "src"-Operand ist ein Festwert, wird er nicht angegeben, ist als "src"-Operand Eins vorgegeben. Die Anzahl der Verschiebeoperationen ist befehlsabhängig:
SLLB : 0 bis 8
SLL : 0 bis 16
SLLL : 0 bis 32
Das Carry-Flag ergibt sich bei der letzten Verschiebung aus dem "msb"-Bit des "dst"-Operanden. In das "lsb"-Bit wird eine 0 geschoben.
Eine Verschiebung mit dem "src"-Operanden Null führt nur zum Setzen der Flags. Diese Operation entspricht einer vorzeichenbehafteten Multiplikation des "dst"-Operanden mit einer im "src"-Operanden angegebenen Zweierpotenz.



Mnemonik: SLL dst,src / SLLB dst,src / SLLL dst,src
Operation: dst(0) := 0
dst(n+1) := dst(n)
C := dst(msb)
Flags: C: Ist nur gesetzt, wenn das zuletzt aus dem "dst"-Operanden geschobene Bit = 1 ist.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das Ergebnis negativ ist.
V: Ist nur gesetzt, wenn das Vorzeichen des "dst"-Operanden während der Rotation wechselt.

Adressierungsart	Zyklen Wort/Byte Long
dst src mo	
R IM 10	13+3n

Befehlsformat: F5.1

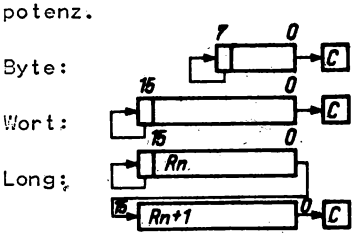
W/B: mo11001wdst.0001 .....src.....

L: mo110011dst.0101 .....src.....

Beispiel: SLL RH4, #2
Wenn Register R4 als "dst"-Operand den Wert 10111100 und der "src"-Operand den Wert 00000010 enthält, erfolgt eine Linksverschiebung um 2 Bit. Nach der Verschiebung enthält RH4 den Wert 11110000. Das Sign-Flag ist gesetzt, das Carry-Flag rückgesetzt.

SRA Shift Right Arithmetic SRA

Der "dst"-Operand wird arithmetisch nach rechts um die Anzahl der im "src"-Operanden angegebenen Bitpositionen verschoben. Der "src"-Operand ist ein Festwert, wird er nicht angegeben, ist als "src"-Operand Eins vorgegeben.
Die Anzahl der Verschiebeoperationen ist befehlsabhängig:
SRAB : 0 bis 8
SRA : 0 bis 16
SRAL : 0 bis 32
Das Carry-Flag ergibt sich bei der letzten Verschiebung aus dem "msb"-Bit des "dst"-Operanden. In das "lsb"-Bit wird eine 0 geschoben.
Eine Verschiebung mit dem "src"-Operanden Null führt nur zum Setzen der Flags. Diese Operation entspricht einer vorzeichenbehafteten Multiplikation des "dst"-Operanden mit einer im "src"-Operanden angegebenen Zweierpotenz.



Mnemonik: **SLA dst,src / SLAB dst,src / SLAL dst,src**
Operation: dst(0) := 0
dst(n+1) := dst(n)
C := dst(msb)
Flags: C: Ist nur gesetzt, wenn das zuletzt aus dem "dst"-Operanden geschobene Bit = 1 ist.
Z: Ist nur gesetzt, wenn das Ergebnis Null ist.
S: Ist nur gesetzt, wenn das Ergebnis negativ ist.
V: Ist nur gesetzt, wenn das Vorzeichen des "dst"-Operanden während der Rotation wechselt.

Adressierungsart	Zyklen Wort/Byte Long
dst src mo	
R IM 10	13+3n

Befehlsformat: F5.1

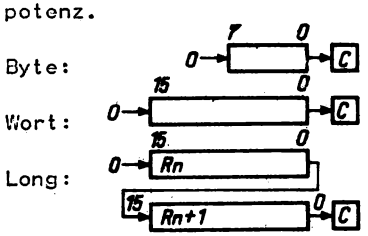
W/B: mo11001wdst.1001
.....src.....

L: mo11001ldst.1101
.....src.....

Beispiel: SRA RH4, #2
Wenn Register R4 als "dst"-Operand den Wert 10111100 und der "src"-Operand den Wert 00000010 enthält, erfolgt eine Rechtsverschiebung um 2 Bit. Nach der Verschiebung enthält RH4 den Wert 11101111. Das Sign-Flag ist gesetzt, das Carry-Flag rückgesetzt.

SRL Shift Right Logical SRL

Der "dst"-Operand wird logisch nach rechts um die Anzahl der im "src"-Operanden angegebenen Bitpositionen verschoben. Der "src"-Operand ist ein Festwert, wird er nicht angegeben, ist als "src"-Operand Eins vorgegeben. Die Anzahl der Verschiebeoperationen ist befehlsabhängig:
SRLB : 0 bis 8
SRL : 0 bis 16
SRLL : 0 bis 32
Das Carry-Flag ergibt sich bei der letzten Verschiebung aus dem "msb"-Bit des "dst"-Operanden. In das "lsb"-Bit wird eine 0 geschoben.
Eine Verschiebung mit dem "src"-Operanden Null führt nur zum Setzen der Flags. Diese Operation entspricht einer vorzeichenbehafteten Multiplikation des "dst"-Operanden mit einer im "src"-Operanden angegebenen Zweierpotenz.



Mnemonik: SLA dst,src / SLAB dst,src / SLAL dst,src
Operation: dst(0) := 0
dst(n+1) := dst(n)
C := dst(msb)
Flags: C: ist nur gesetzt, wenn das zuletzt aus dem "dst"-Operanden geschobene Bit = 1 ist.
Z: ist nur gesetzt, wenn das Ergebnis Null ist.
S: ist nur gesetzt, wenn das Ergebnis negativ ist.
V: ist nur gesetzt, wenn das Vorzeichen des "dst"-Operanden während der Rotation wechselt.

Adressierungsart dst src mo	Zyklen Wort/Byte Long
R IM 10	13+3n

Befehlsformat: F5.1

W/B: mo11001wdst.0001
.....src.....

L: mo110011dst.0101
.....src.....

Beispiel: SRL RH4, #2
Wenn Register R4 als "dst"-Operand den Wert 10111100 und der "src"-Operand den Wert 00000010 enthält, erfolgt eine Rechtsverschiebung um 2 Bit. Nach der Verschiebung enthält RH4 den Wert 00101111. Das Sign-Flag und das Carry-Flag sind rückgesetzt.

3.5.7. Blocktransfer- und Blocksuchbefehle

CPD — **Compare and Decrement** — **CPD**

Der CPD-Befehl dient zum Suchen von Daten in Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem "dst"-Operanden verglichen. Das Zero-Flag wird nur gesetzt, wenn die mit "cc" angegebene Bedingung erfüllt ist. Das "src"- und das "r"-Register werden bei jedem Befehl automatisch dekrementiert.

Mnemonik: **CPD dst,src,r,cc / CPDB dst,src,r,cc**
Operation: dst - src
Autodekrement src (-1 bei Byte -2 bei Wort) r := r - 1
Flags: C,S: Undefiniert.
Z: Ist nur gesetzt, wenn im Vergleich "cc" erfüllt ist.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
R IR	20

Befehlsformat: F6.5

W/B: 1011101wsrc.1000
0000.r..dst.src.

Beispiel: CPDB RL6, @R7, R8, EQ
Wenn in RL6 %41 steht und R7 auf ein Byte einer Zeichenkette zeigt, wird dieses Byte mit %41 verglichen, R7 und R8 dekrementiert und in Abhängigkeit vom Ergebnis das Zero-Flag gesetzt.

CPDR — **Compare, Decrement and Repeat** — **CPDR**

Der CPDR-Befehl dient zum Suchen von Daten in Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem "dst"-Operanden verglichen. Das Zero-Flag wird nur gesetzt, wenn die mit "cc" angegebene Bedingung erfüllt ist. Das "src"- und das "r"-Register werden bei jedem Befehl automatisch dekrementiert, der Befehl wird ausgeführt, bis "cc" erfüllt oder r - 1 = 0 ergibt.

Mnemonik: **CPDR dst,src,r,cc / CPDRB dst,src,r,cc**
Operation: dst - src
Autodekrement src (-1 bei Byte -2 bei Wort) r := r - 1
Flags: C,S: Undefiniert.
Z: Ist nur gesetzt, wenn im Vergleich "cc" erfüllt ist.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
R IR	11+9n

Befehlsformat: F6.5

W/B: 1011101wsrc.1100
0000.r..dst..cc.

Beispiel: CPDRB RL6, @R7, R8, EQ
Wenn in RL6 %41 steht und R7 auf ein Byte einer Zeichenkette zeigt, wird diese Zeichenkette rückwärts nach %41 durchsucht. Der Abbruch erfolgt, wenn %41 gefunden oder R8=0 wird.

CPI Compare and Increment CPI

Der CPI-Befehl dient zum Suchen von Daten in Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem "dst"-Operanden verglichen. Das Zero-Flag wird nur gesetzt, wenn die mit "cc" angegebene Bedingung erfüllt ist. Das "src"-Register wird bei jedem Befehl automatisch inkrementiert, das "r"-Register wird dekrementiert.

Mnemonik: **CPI dst,src,r,cc / CPIB dst,src,r,cc**
Operation: dst - src
Autoinkrement src (+1 bei Byte +2 bei Wort) r := r - 1
Flags: C,S: Undefiniert.
Z: Ist nur gesetzt, wenn im Vergleich "cc" erfüllt ist.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Befehlsformat: F6.5

W/B: 1011101wsrc.0000
0000.r..dst..cc.

Adressierungsart	Zyklen Wort/byte
dst src	
R IR	20

Beispiel: CPIB RL6, @R7, R8, EQ
Wenn in RL6 %41 steht und R7 auf ein Byte einer Zeichenkette zeigt, wird dieses Byte mit %41 verglichen, R7 in- und R8 dekrementiert und in Abhängigkeit vom Ergebnis das Zero-Flag gesetzt.

CPIR Compare, Increment and Repeat CPIR

Der CPIR-Befehl dient zum Suchen von Daten in Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem "dst"-Operanden verglichen. Das Zero-Flag wird nur gesetzt, wenn die mit "cc" angegebene Bedingung erfüllt ist. Das "src"-Register wird bei jedem Befehl automatisch inkrementiert, das "r"-Register wird dekrementiert, der Befehl wird ausgeführt, bis "cc" erfüllt oder r - 1 = 0 ergibt.

Mnemonik: **CPIR dst,src,r,cc / CPIRB dst,src,r,cc**
Operation: dst - src
Autoinkrement src (+1 bei Byte +2 bei Wort) r := r + 1
Flags: C,S: Undefiniert.
Z: Ist nur gesetzt, wenn im Vergleich "cc" erfüllt ist.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Befehlsformat: F6.5

W/B: 1011101wsrc.0100
0000.r..dst..cc.

Adressierungsart	Zyklen Wort/Byte
dst src	
R IR	11+9n

Beispiel: CPIRB RL6, @R7, R8, EQ
Wenn in RL6 %41 steht und R7 auf ein Byte einer Zeichenkette zeigt, wird diese Zeichenkette vorwärts nach %41 durchsucht. Der Abbruch erfolgt, wenn %41 gefunden oder R8=0 wird.

CPSD — Compare String and Decrement — CPSD

Der CPSD-Befehl dient zum Vergleich von Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem Inhalt des durch das "dst"-Register adressierten verglichen. Das Zero-Flag wird gesetzt, wenn die mit "cc" angegebene Bedingung erfüllt ist. Das "src"- und das "r"-Register werden bei jedem Befehl automatisch dekrementiert.

Mnemonik: **CPSD dst,src,r,cc / CPSDB dst,src,r,cc**
Operation: dst - src
Autodekrement src,dst (-1 bei Byte -2 bei Wort) r := r - 1
Flags: C,S: Undefiniert.
Z: Ist nur gesetzt, wenn im Vergleich "cc" erfüllt ist.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
R IR	25

Befehlsformat: F6.5

W/B: 1011101wsrc.1010
0000.r..dst..cc.

Beispiel: CPSDB @R6, @R7, R8, EQ
Wenn R6 und R7 jeweils auf ein Byte einer Zeichenkette zeigen, werden diese Bytes miteinander verglichen, R6, R7 und R8 dekrementiert und in Abhängigkeit vom Ergebnis das Zero-Flag gesetzt.

CPSDR — Compare String, Decrement and Repeat — CPSDR

Der CPSDR-Befehl dient zum Vergleich von Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem Inhalt des durch das "dst"-Register adressierten verglichen. Das Zero-Flag wird nur gesetzt, wenn die mit "cc" angegebene Bedingung erfüllt ist. Das "src"-, "dst"- und "r"-Register werden bei jedem Befehl automatisch dekrementiert, der Befehl wird ausgeführt, bis "cc" erfüllt oder r - 1 = 0 ergibt.

Mnemonik: **CPSDR dst,src,r,cc / CPSDRB dst,src,r,cc**
Operation: dst - src
Autodekrement src (-1 bei Byte -2 bei Wort) r := r - 1
Flags: C,S: Undefiniert.
Z: Ist nur gesetzt, wenn im Vergleich "cc" erfüllt ist.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
R IR	11+14n

Befehlsformat: F6.5

W/B: 1011101wsrc.1110
1011101wsrc.1110

Beispiel: CPSDRB @R6, @R7, R8, EQ
R6 und R7 zeigen jeweils auf ein Byte einer Zeichenkette, diese Bytes werden miteinander verglichen, das Zero-Flag wird gesetzt. R6, R7, und R8 werden dekrementiert. Der Abbruch erfolgt, wenn "cc" wahr ist oder r - 1 = 0 ist.

CPSI — Compare String and Increment — CPSI

Der CPSI-Befehl dient zum Vergleich von Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem Inhalt des durch das "dst"-Register adressierten verglichen. Das Zero-Flag wird nur gesetzt wenn die mit "cc" angegebene Bedingung erfüllt ist. Das "src"- und das "dst"-Register werden bei jedem Befehl automatisch inkrementiert, das "r"-Register wird dekrementiert.

Mnemonik: **CPSI dst,src,r,cc / CPSIB dst,src,r,cc**
Operation: dst - src
Autoinkrement src (+1 bei Byte +2 bei Wort) r := r - 1
Flags: C,S: Undefiniert.
Z: Ist nur gesetzt, wenn im Vergleich "cc" erfüllt ist.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
R IR	25

Befehlsformat: F6.5

W/B: 1011101wsrc.0010 0000.r..dst..cc.

Beispiel: CPSIB @R6, @R7, R8, EQ
Wenn R6 und R7 jeweils auf ein Byte einer Zeichenkette zeigen werden diese Byte miteinander verglichen, das Zero-Flag wird gesetzt. R6 und R7 werden in-, und R8 wird dekrementiert.

CPSIR — Compare String, Increment and Repeat — CPSIR

Der CPSIR-Befehl dient zum Vergleich von Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem Inhalt des durch das "dst"-Register adressierten verglichen. Das Zero-Flag wird nur gesetzt, wenn die mit "cc" angegebene Bedingung erfüllt ist. Das "src"- und "dst"-Register werden bei jedem Befehl automatisch inkrementiert, das "r"-Register wird dekrementiert, der Befehl wird ausgeführt, bis "cc" erfüllt ist oder r - 1 = 0 ergibt.

Mnemonik: **CPSIR dst,src,r,cc / CPSIRB dst,src,r,cc**
Operation: dst - src
Autodekrement src (+1 bei Byte +2 bei Wort) r := r - 1
Flags: C,S: Undefiniert.
Z: Ist nur gesetzt, wenn im Vergleich "cc" erfüllt ist.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	11+14n

Befehlsformat: F6.5

W/B: 1011101wsrc.0110 0000.r..dst..cc.

Beispiel: CPSIRB @R6, @R7, R8, EQ
R6 und R7 zeigen jeweils auf ein Byte einer Zeichenkette. Diese Bytes werden miteinander verglichen, das Zero-Flag wird gesetzt, R6, R7 in- und R8 dekrementiert, abgebrochen wird, wenn "cc" wahr ist oder r - 1 = 0 ist.

LDD — Load and Decrement — LDD

Der LDD-Befehl dient zum Blocktransfer von Daten- bzw. Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem Inhalt des durch das "dst"-Register adressierten geladen. Das "src"-, "dst"- und "r"-Register werden bei jedem Befehl automatisch dekrementiert. Das "src"- und "dst"-Register adressieren nach der Befehlsabarbeitung das vorhergehende Element der Kette.

Mnemonik: **LDD dst,src,r / LDDB dst,src,r**
Operation: dst := src
Autodekrement src,dst (-1 bei Byte -2 bei Wort) r := r - 1
Flags: Z: Undefiniert.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	20

Befehlsformat: F6.4

W/B: 1011101wsrc.1001
0000.r..dst.1000

Beispiel: LDDB @R6, @R7, R8
Steht in R6 %1001, in R7 %2001, in R8 5 und auf %2000 das Wort %AA55, so wird mit dem LDDB-Befehl %55 auf die Adresse %1001 geladen. R6 zeigt auf %1000, R7 auf %2000, in R8 steht 4.

LDDR — Load, Decrement and Repeat — LDDR

Der LDDR-Befehl dient zum Blocktransfer von Daten- bzw. Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem Inhalt des durch das "dst"-Register adressierten geladen. Das "src"-, "dst"- und "r"-Register werden bei jedem Befehl automatisch dekrementiert, der Befehl wird ausgeführt, bis r - 1 = 0 ergibt.
Der PC (Befehlszähler) wird nach jedem Transfer auf die Anfangsadresse des Befehls zurückgestellt, tritt während der Befehlsabarbeitung ein Interrupt auf, so wird die Anfangsadresse des Befehls als Rückkehradresse gespeichert.

Mnemonik: **LDDR dst,src,r / LDDRB dst,src,r**
Operation: dst := src
Autodekrement src,dst (-1 bei Byte -2 bei Wort) r := r - 1
Flags: V: Ist gesetzt.

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	11+9n

Befehlsformat: F6.4

W/B: 1011101wsrc.1001
0000.r..dst.0000

Beispiel: LDDRB @R6, @R7, R8
Steht in R6 %1001, in R7 %2001, in R8 2 und auf %2000 das Wort %AA55, so wird mit dem LDDRB-Befehl %55 auf die Adresse %1001 und %AA auf die Adresse %1000 geladen. R6 zeigt auf %0FFF, R7 auf %1FFF, in R8 steht 0, das Overflow-Flag ist Eins.

LDI — Load and Increment — LDI

Der LDI-Befehl dient zum Blocktransfer von Daten- bzw. Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem Inhalt des durch das "dst"-Register adressierten geladen. Das "src"- und "dst"-Register werden bei jedem Befehl automatisch inkrementiert, das "r"-Register (Zähler) wird dekrementiert. Das "src"- und "dst"-Register adressieren nach der Befehlsabarbeitung das nachfolgende Element der Kette.

Mnemonik: **LDI dst,src,r / LDIB dst,src,r**
Operation: dst := src
Autoinkrement src,dst (+1 bei Byte +2 bei Wort) r := r - 1
Flags: Z: Undefiniert.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	20

Befehlsformat: F6.4

W/B: 1011101wsrc.0001 0000.r..dst.1000

Beispiel: LDIB @R6, @R7, R8
Steht in R6 %1000, in R7 %2000, in R8 5 und auf %2000 das Wort %AA55, so wird mit dem LDIB-Befehl %AA auf die Adresse %1000 geladen. R6 zeigt auf %1001, R7 auf %2001, in R8 steht 4.

LDIR — Load, Increment and Repeat — LDIR

Der LDIR-Befehl dient zum Blocktransfer von Daten- bzw. Zeichenketten. Der Inhalt des durch das "src"-Register adressierten Speicherplatzes wird mit dem Inhalt des durch das "dst"-Register adressierten geladen. Das "src"- und "dst"-Register werden bei jedem Befehl automatisch inkrementiert, das "r"-Register wird dekrementiert. Der Befehl wird ausgeführt, bis r - 1 = 0 ergibt.
Der PC (Befehlszähler) wird nach jedem Transfer auf die Adresse des Befehls zurückgestellt; tritt während der Befehlsabarbeitung ein Interrupt auf, so wird die Adresse des Befehls als Rückkehradresse gespeichert.

Mnemonik: **LDIR dst,src,r / LDIRB dst,src,r**
Operation: dst := src
Autoinkrement src,dst (+1 bei Byte +2 bei Wort) r := r - 1
Flags: V: Ist gesetzt.

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	11+9n

Befehlsformat: F6.4

W/B: 1011101wsrc.0001 0000.r..dst.0000

Beispiel: LDIRB @R6, @R7, R8
Steht in R6 %1000, in R7 %2000, in R8 2 und auf %2000 das Wort %AA55, so werden mit dem LDIRB-Befehl %AA auf die Adresse %1000 und %AA auf die Adresse %1001 geladen. R6 zeigt auf %1002, R7 auf %2002, in R8 steht 0, das Overflow-Flag ist Eins.

TRDB — Translate and Decrement — TRDB

Der TRDB-Befehl dient zum Übersetzen von Daten- bzw. Zeichenketten eines Kodes in einen anderen Kode. Der Inhalt des durch das "dst"-Register adressierten Speicherplatzes wird als 8-Bit-Index zum "src"-Register addiert, das damit adressierte Byte auf den durch das "dst"-Register adressierten Speicherplatz geladen. Damit wird der ursprünglich geladene Index im "dst"-Bereich überschrieben.
Das "dst"- und "r"-Register werden automatisch dekrementiert.

Mnemonik: **TRDB dst,src,r**
Operation: dst := src(dst)
Autodekrement dst (-1) r := r - 1
Flags: Z: Undefiniert.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Befehlsformat: F6.4

B: 10111000dst.1000 0000.r..src.0000

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	25

Beispiel: TRDBB @R6, @R7, R8
Steht in R6 %1080, in R7 %2000, in R8 5 und auf %1080 das Byte %55, so wird mit dem TRDBB-Befehl von der Adresse %1055 ein Byte gelesen, und auf die Adresse %1080 geladen. R6 zeigt auf %107F, in R8 steht 4.

TRDRB — Translate, Decrement and Repeat — TRDRB

Der TRDRB-Befehl dient zum Übersetzen von Daten- bzw. Zeichenketten eines Kodes in einen anderen Kode. Der Inhalt des durch das "dst"-Register adressierten Speicherplatzes wird als 8-Bit-Index zum "src"-Register addiert. Das damit adressierte Byte wird auf den durch das "dst"-Register adressierten Speicherplatz geladen. Damit wird der ursprünglich geladene Index im "dst"-Bereich überschrieben. Das "dst"- und "r"-Register werden automatisch dekrementiert, bis r - 1 = 0 ist.
Der Inhalt des Registers RH1 wird zerstört.

Mnemonik: **TRDRB dst,src,r**
Operation: dst := src(dst)
Autodekrement dst (-1) r := r - 1 bis r = 0
Flags: Z: Undefiniert.
V: Ist gesetzt.

Befehlsformat: F6.4

B: 10111000dst.1100 0000.r..src.0000

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	11+14n

Beispiel: TRDRBB @R6, @R7, R8
Steht in R6 %1080, in R7 %2000, in R8 2 und auf %107F das Byte %55 und auf %1080 %04, so wird mit dem TRDRBB-Befehl von der Adresse %2004 ein Byte gelesen und auf die Adresse %1080 geladen. Anschließend wird von der Adresse %2055 ein Byte gelesen und auf %107F geladen. R6 zeigt auf %107E, in R8 steht 0.

TRIB Translate and Increment TRIB

Der TRIB-Befehl dient zum Übersetzen von Daten- bzw. Zeichenketten eines Kodes in einen anderen Kode. Der Inhalt des durch das "dst"-Register adressierten Speicherplatzes wird als 8-Bit-Index zum "src"-Register addiert, das damit adressierte Byte auf den durch das "dst"-Register adressierten Speicherplatz geladen. Damit wird der ursprünglich geladene Index im "dst"-Bereich überschrieben. Das "dst"-Register wird in- und das "r"-Register automatisch dekrementiert.

Mnemonik: **TRIB dst,src,r**
Operation: dst := src(dst)
Autoinkrement dst (+1) r := r - 1
Flags: Z: Undefiniert.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Befehlsformat: F6.4

B: 10111000dst.0000
0000.r..src.0000

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	25

Beispiel: TRIBB @R6, @R7, R8
Steht in R6 %1080, in R7 %2000, in R8 5 und auf %1080 das Byte %55, so wird mit dem TRIBB-Befehl von der Adresse %1055 ein Byte gelesen und auf die Adresse %1080 geladen. R6 zeigt auf %1081, in R8 steht 4.

TRIRB Translate, Increment and Repeat TRIRB

Der TRIRB-Befehl dient zum Übersetzen von Daten- bzw. Zeichenketten eines Kodes in einen anderen Kode. Der Inhalt des durch das "dst"-Register adressierten Speicherplatzes wird als 8-Bit-Index zum "src"-Register addiert. Das damit adressierte Byte wird auf den durch das "dst"-Register adressierten Speicherplatz geladen. Damit wird der ursprünglich geladene Index im "dst"-Bereich überschrieben. Das "dst"-Register wird in- und das "r"-Register automatisch dekrementiert, bis r - 1 = 0 ist.
Der Inhalt des Registers RH1 wird zerstört.

Mnemonik: **TRIRB dst,src,r**
Operation: dst := src(dst)
Autoinkrement dst (+1) r := r - 1 bis r = 0
Flags: Z: Undefiniert.
V: Ist gesetzt.

Befehlsformat: F6.4

B: 10111000dst.0100
0000.r..src.0000

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	11+14n

Beispiel: TRIRB @R6, @R7, R8
Steht in R6 %107F, in R7 %2000, in R8 2 und auf %107F das Byte %55 und auf %1080 das Byte %04, so wird mit dem TRIRB-Befehl von der Adresse %2055 ein Byte gelesen und auf die Adresse %107F geladen. Anschließend wird von der Adresse %2004 ein Byte gelesen und auf %1080 geladen. R6 zeigt auf %1081, in R8 steht 0

TRTDB Translate, Test and Decrement TRTDB

Der TRTDB-Befehl dient zum Übersetzen von Daten- bzw. Zeichenketten eines Kodes in einen anderen Kode. Der Inhalt des durch das "src1"-Register adressierten Speicherplatzes wird als 8-Bit-Index zum "src2"-Register addiert. Das damit adressierte Byte wird in das Register RH1 geladen, ist der Wert Null, wird das Zero-Flag gesetzt. Die durch die "src"-Register adressierten Speicherplätze werden nicht verändert.
Das "src1"- und "r"-Register werden automatisch dekrementiert.

Mnemonik: **TRTDB src1,src2,r**
Operation: RH1 := src2(src1)
Autodekrement src1 (-1) r := r - 1
Flags: Z: ist gesetzt, wenn der in RH1 geladene Wert Null ist.
V: ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	25

Befehlsformat: F6.4

B: 10111000src11010 0000.r..src20000

Beispiel: TRTDB @R6, @R7, R8
Steht in R6 %1080, in R7 %2000, in R8 5 und auf %1080 das Byte %55, so wird mit dem TRTDB-Befehl von der Adresse %1055 ein Byte gelesen und in das Register RH1 geladen. R6 zeigt auf %107F, in R8 steht 4, und das Zero-Flag ist gesetzt, wenn der in das Register RH1 geladene Wert Null war.

TRTDRB Translate, Test, Decrement and Repeat TRTDRB

Der TRTDRB-Befehl dient zum Übersetzen von Daten- bzw. Zeichenketten eines Kodes in einen anderen Kode. Der Inhalt des durch das "src1"-Register adressierten Speicherplatzes wird als 8-Bit-Index zum "src2"-Register addiert. Das damit adressierte Byte wird in das Register RH1 geladen, ist der Wert Null, wird das Zero-Flag gesetzt. Die durch die "src"-Register adressierten Speicherplätze werden nicht verändert.
Das "src1"- und "r"-Register werden automatisch dekrementiert.
Der Befehl wird wiederholt, bis r - 1 = 0 oder RH1 <> 0 ist.

Mnemonik: **TRTDRB src1,src2,r**
Operation: RH1 := src2(src1)
Autodekrement src1 (-1) r := r - 1 bis r - 1 = 0 oder RH1 <> 0
Flags: Z: ist gesetzt, wenn der in RH1 geladene Wert Null ist.
V: ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	11+14n

Befehlsformat: F6.4

W/B: 10111000src11110 0000.r..src21110

Beispiel: TRTDRB @R6, @R7, R8
Steht in R6 %1080, in R7 %2000, in R8 2 und auf %1080 das Byte %55, so wird mit dem TRTDRB-Befehl von der Adresse %1055 ein Byte gelesen und in das Register RH1 geladen. Ist es nicht Null, wird die Befehlsabarbeitung abgebrochen. R6 zeigt auf %107F, in R8 steht 1.

TRTIB Translate, Test and Increment TRTIB

Der TRTIB-Befehl dient zum Übersetzen von Daten- bzw. Zeichenketten eines Kodes in einen anderen Kode. Der Inhalt des durch das "src1"-Register adressierten Speicherplatzes wird als 8-Bit-Index zum "src2"-Register addiert. Das damit adressierte Byte wird in das Register RH1 geladen, ist der Wert Null, wird das Zero-Flag gesetzt. Die durch die "src"-Register adressierten Speicherplätze werden nicht verändert. Das "src1"-Register wird in- und das "r"-Register automatisch dekrementiert.

Mnemonik: **TRTIB src1,src2,r**
Operation: RH1 := src2(src1)
Autoinkrement src1 (+1) r := r - 1
Flags: Z: Ist gesetzt, wenn der in RH1 geladene Wert Null ist.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	25

Befehlsformat: F6.4

W/B: 10111000src10010 0000.r..src20000

Beispiel: TRTIB @R6, @R7, R8
Steht in R6 %1080, in R7 %2000, in R8 5 und auf %1080 das Byte %55, so wird mit dem TRTIB-Befehl von der Adresse %1055 ein Byte gelesen und in das Register RH1 geladen. R6 zeigt auf %1081, in R8 steht 4, und das Zero-Flag ist gesetzt, wenn der in das Register RH1 geladene Wert Null war.

TRTIRB Translate, Test, Increment and Repeat TRTIRB

Der TRTIRB-Befehl dient zum Übersetzen von Daten- bzw. Zeichenketten eines Kodes in einen anderen Kode. Der Inhalt des durch das "src1"-Register adressierten Speicherplatzes wird als 8-Bit-Index zum "src2"-Register addiert. Das damit adressierte Byte wird in das Register RH1 geladen, ist der Wert Null, wird das Zero-Flag gesetzt. Die durch die "src"-Register adressierten Speicherplätze werden nicht verändert. Das "src1"-Register wird in- und das "r"-Register automatisch dekrementiert.
Der Befehl wird wiederholt, bis r - 1 = 0 oder RH1 <> 0 ist.

Mnemonik: **TRTIRB src1,src2,r**
Operation: RH1 := src2(src1)
Autoinkrement src1 (+1) r := r - 1 bis r - 1 = 0 oder RH1 <> 0
Flags: Z: Ist gesetzt, wenn der in RH1 geladene Wert Null ist.
V: Ist nur gesetzt, wenn r - 1 = 0 ist.

Adressierungsart	Zyklen Wort/Byte
dst src	
IR IR	11+14n

Befehlsformat: F6.4

B: 10111000src10110 0000.r..src21110

Beispiel: TRTIRB @R6, @R7, R8
Steht in R6 %107F, in R7 %2000, in R8 2 und auf %107F das Byte %55, so wird mit dem TRTIRB-Befehl von der Adresse %1055 ein Byte gelesen und in das Register RH1 geladen. Ist es nicht Null, wird die Befehlsabarbeitung abgebrochen. R6 zeigt auf %107F, in R8 steht 1.

3.5.8. Ein-/Ausgabebefehle

IN **Input** **(PB)** **IN**

Der IN-Befehl dient zur Eingabe von Daten aus dem Ein-/Ausgabeadreßraum. Der Inhalt der durch den "src"-Operanden im Ein-/Ausgabeadreßraum angegebenen 16-Bit-Adresse wird in das "dst"-Register geladen.

Mnemonik: **IN dst,src / INB dst,src**
Operation: dst := src

Adressierungsart	Zyklen Wort/Byte
dst src	
R IR	10
R DA	12

Befehlsformat: F7.3 IR: src

W/B: 0011110wsrc.dst.

Befehlsformat: F7.1 DA: src

W/B: 0011101wdst.0100

Beispiel: IN R2, @R5
Wenn in R5 die E/A-Adresse %4500 steht, wird von dieser Adresse ein Wort in das Register R2 eingelesen.

IND **Input and Decrement** **(PB)** **IND**

Der IND-Befehl dient zur Eingabe von Datenblöcken aus dem Ein-/Ausgabeadreßraum in den Speicheradreßraum.
Der Inhalt der durch das "src"-Register im Ein-/Ausgabeadreßraum angegebenen 16-Bit-Adresse wird auf die durch das "dst"-Register angegebene Speicheradresse geladen.
Das "dst"- und das "r"-Register werden automatisch dekrementiert.

Mnemonik: **IND dst,src,r / INDB dst,src,r**
Operation: dst := src
Autodekrement dst (-1 Byte -2 Wort) r := r - 1

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	21

Befehlsformat: F6.4

W/B: 0011101wsrc.1000
0000.r..dst.1000

Beispiel: IND @RR4, @R7, R8
Wenn Register RR4 im segmentierten Mode den Wert %83003000, R7 den Wert %1006 und R8 den Wert 4 enthält, so steht nach dem IND-Befehl im Segment 3 auf der Adresse %3000 der von der E/A-Adresse %1006 gelesene 16-Bit-Wert. Das Register RR4 enthält den Wert %83002FFE, R7 verändert sich nicht, und in R8 steht 3.

INDR — Input, Decrement and Repeat (PB) — INDR

Der INDR-Befehl dient zur Eingabe von Datenblöcken aus dem Ein-/Ausgabeadreßraum in den Speicheradreßraum.
Der Inhalt der durch das "src"-Register im Ein-/Ausgabeadreßraum angegebenen 16-Bit-Adresse wird auf die durch das "dst"-Register angegebene Speicheradresse geladen.
Das "dst"- und das "r"-Register werden automatisch dekrementiert.
Der Befehl wird wiederholt, bis r - 1 = 0 ist.

Mnemonik: **INDR dst,src,r / INDRB dst,src,r**
Operation: dst := src
Autodekrement dst (-1 Byte -2 Wort) r := r - 1 bis r = 0
Flags: Z: Undefiniert
V: Ist gesetzt.

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	11+10n

Befehlsformat: F6.4

W/B: 0011101wsrc.1000
0000.r..dst.0000

Beispiel: INDR @RR4, @R7, R8
Wenn Register RR4 im segmentierten Mode den Wert %83003000, R7 den Wert %1006 und R8 den Wert 2 enthält, so steht nach dem INDR-Befehl im Segment 3 auf der Adresse %3000 der erste der von der E/A-Adresse %1006 gelesenen 16-Bit-Werte. Der zweite Wert steht auf der Adresse %2FFE. Das Register RR4 enthält den Wert %83002FFC, R7 verändert sich nicht, und in R8 steht 0.

INI — Input and Increment (PB) — INI

Der INI-Befehl dient zur Eingabe von Datenblöcken aus dem Ein-/Ausgabeadreßraum in den Speicheradreßraum.
Der Inhalt der durch das "src"-Register im Ein-/Ausgabeadreßraum angegebenen 16-Bit-Adresse wird auf die durch das "dst"-Register angegebene Speicheradresse geladen. Das "dst"-Register wird in- und das "r"-Register wird automatisch dekrementiert.

Mnemonik: **INI dst,src,r / INIB dst,src,r**
Operation: dst := src
Autoinkrement dst (+1 Byte +2 Wort) r := r - 1

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	21

Befehlsformat: F6.4

W/B: 0011101wsrc.0000
0000.r..dst.1000

Beispiel: INI @RR4, @R7, R8
Wenn Register RR4 im segmentierten Mode den Wert %83003000, R7 den Wert %1006 und R8 den Wert 4 enthält, so steht nach dem INI-Befehl im Segment 3 auf der Adresse %3000 der von der E/A-Adresse %1006 gelesene 16-Bit-Wert. Das Register RR4 enthält den Wert %83003002, R7 verändert sich nicht, und in R8 steht 3.

INIR — Input, Increment and Repeat — (PB) — INIR

Der INIR-Befehl dient zur Eingabe von Datenblöcken aus dem Ein-/Ausgabeadreßraum in den Speicheradreßraum.
Der Inhalt der durch das "src"-Register im Ein-/Ausgabeadreßraum angegebenen 16-Bit-Adresse wird auf die durch das "dst"-Register angegebene Speicheradresse geladen. Das "dst"-Register wird in- und das "r"-Register automatisch dekrementiert. Der Befehl wird wiederholt, bis r - 1 = 0 ist.

Mnemonik: **INIR dst,src,r / INIRB dst,src,r**
Operation: dst := src
Autoinkrement dst (+1 Byte +2 Wort) r := r - 1 bis r = 0
Flags: Z: Undefiniert
V: Ist gesetzt.

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	11+10n

Befehlsformat: F6.4

W/B: 0011101wsrc.0000
0000.r..dst.1000

Beispiel: INIR @RR4, @R7, R8
Wenn Register RR4 im segmentierten Mode den Wert %83003000, R7 den Wert %1006 und R8 den Wert 2 enthält, so steht nach dem INIR-Befehl im Segment 3 auf der Adresse %3000 der erste der von der E/A-Adresse %1006 gelesenen 16-Bit-Werte. Der zweite Wert steht auf der Adresse %3002. Das Register RR4 enthält den Wert %83003004, R7 verändert sich nicht, und in R8 steht 0.

OTDR — Output, Decrement and Repeat — (PB) — OTDR

Der OTDR-Befehl dient zur Ausgabe von Datenblöcken aus dem Speicheradreßraum in den Ein-/Ausgabeadreßraum.
Der Inhalt der durch das "src"-Register im Speicheradreßraum angegebenen Adresse wird auf die durch das "dst"-Register angegebene 16-Bit-Adresse im Ein-/Ausgabeadreßraum geladen. Das "src"- und das "r"-Register werden automatisch dekrementiert. Der Befehl wird wiederholt, bis r - 1 = 0 ist.

Mnemonik: **OTDR dst,src,r / OTDRB dst,src,r**
Operation: dst := src
Autodekrement src (-1 Byte -2 Wort) r := r - 1 bis r = 0
Flags: Z: Undefiniert
V: Ist gesetzt.

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	11+10n

Befehlsformat: F6.4

W/B: 0011101wsrc.1010
0000.r..dst.0000

Beispiel: OTDR @R7, @R4, R8
Wenn Register R4 die Speicheradresse %3000, R7 die E/A-Adresse %1006 und R8 den Wert 2 enthält, so wird mit dem OTDR-Befehl der Inhalt der Adresse %3000 in die E/A-Adresse %1006 geladen. Der zweite 16-Bit-Wert steht auf der Adresse %2FFE. Das Register R4 enthält %2FFC, R7 verändert sich nicht, und in R8 steht 0.

OTIR — Output, Increment and Repeat — (PB) — OTIR

Der OTIR-Befehl dient zur Ausgabe von Datenblöcken aus dem Speicheradreßraum in den Ein-/Ausgabeadreßraum.
Der Inhalt der durch das "src"-Register im Speicheradreßraum angegebenen Adresse wird auf die durch das "dst"-Register angegebene 16-Bit-Adresse im Ein-/Ausgabeadreßraum geladen.
Das "src"-Register wird in- und das "r"-Register wird automatisch dekrementiert.
Der Befehl wird wiederholt, bis r - 1 = 0 ist.

Mnemonik: **OTIR dst,src,r / OTIRB dst,src,r**
Operation: dst := src
Autoinkrement src (+1 Byte +2 Wort) r := r - 1 bis r = 0
Flags: Z: Undefiniert
V: Ist gesetzt.

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	11+10n

Befehlsformat: F6.4

W/B: 0011101wsrc.0010 0000.r...dst.0000

Beispiel: OTIR @R7, @R4, R8
Wenn Register R4 die Speicheradresse %3000, R7 die E/A-Adresse %1006 und R8 den Wert 2 enthält, so wird mit dem OTIR-Befehl der Inhalt der Adresse %3000 in die E/A-Adresse %1006 geladen. Der zweite 16-Bit-Wert steht auf der Adresse %3002. Das Register R4 enthält die Speicheradresse %3004, R7 verändert sich nicht, und in R8 steht 0.

OUT — Output — (PB) — OUT

Der OUT-Befehl dient zur Ausgabe von Daten in den Ein-/Ausgabeadreßraum. Der Inhalt des "src"-Registers wird in den "dst"-Operanden im Ein-/Ausgabeadreßraum geladen. Adressen im Ein-/Ausgabeadreßraum sind 16-Bit-Adressen.

Mnemonik: **OUT dst,src / OUTB dst,src**
Operation: dst := src

Adressierungsart	Zyklen Wort/Byte
dst src	
IR R	10
DA R	12

Befehlsformat: F7.4 IR: src

W/B: 0011111wdst.src.

Befehlsformat: F7.2 DA: src

W/B: 0011101wsrc.0110

Beispiel: OUT @R5, R2
Wenn in R5 die E/A-Adresse %4500 steht, wird in diese Adresse ein Wort aus dem Register R2 geladen.

OUTD Output and Decrement (PB) OUTD

Der OUTD-Befehl dient zur Ausgabe von Datenblöcken aus dem Speicheradreßraum in den Ein-/Ausgabeadreßraum.
Der Inhalt der durch das "src"-Register im Speicheradreßraum angegebenen Adresse wird auf die durch das "dst"-Register angegebene 16-Bit-Adresse im Ein-/Ausgabeadreßraum geladen.
Das "src"- und das "r"-Register werden automatisch dekrementiert.

Mnemonik: **OUTD dst,src,r / OUTDB dst,src,r**
Operation: dst := src
Autodekrement src (-1 Byte -2 Wort) r := r - 1

Befehlsformat: F6.4

W/B: 0011101wsrc.1010
0000.r..dst.1000

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	21

Beispiel: OUTD @R7, @RR4, R8
Wenn Register RR4 im segmentierten Mode die Speicheradresse %83003000, R7 die E/A-Adresse %1006 und R8 den Wert 4 enthält, so wird mit dem OUTD-Befehl der Inhalt der Adresse %3000 im Segment 3 in die E/A-Adresse %1006 geladen. Das Register RR4 enthält die Speicheradresse %83002FFE, R7 verändert sich nicht, und in R8 steht 3.

OUTI Output and Increment (PB) OUTI

Der OUTI-Befehl dient zur Ausgabe von Datenblöcken aus dem Speicheradreßraum in den Ein-/Ausgabeadreßraum.
Der Inhalt der durch das "src"-Register im Speicheradreßraum angegebenen Adresse wird auf die durch das "dst"-Register angegebene 16-Bit-Adresse im Ein-/Ausgabeadreßraum geladen.
Das "src"-Register wird in- und das "r"-Register wird automatisch dekrementiert.

Mnemonik: **OUTI dst,src,r / OUTIB dst,src,r**
Operation: dst := src
Autoinkrement src (+1 Byte +2 Wort) r := r - 1

Befehlsformat: F6.4

W/B: 0011101wsrc.0010
0000.r..dst.1000

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	21

Beispiel: OUTI @R7, @RR4, R8
Wenn Register RR4 im segmentierten Mode die Speicheradresse %83003000, R7 die E/A-Adresse %1006 und R8 den Wert 4 enthält, so wird mit dem OUTI-Befehl der Inhalt der Adresse %3000 im Segment 3 in die E/A-Adresse %1006 geladen. Das Register RR4 enthält die Speicheradresse %83003002, R7 verändert sich nicht, und in R8 steht 3.

SIN — Special Input — (PB) — SIN

Der SIN-Befehl dient zur Eingabe von Daten aus dem Spezial-Ein-/Ausgabeadreßraum. Der Inhalt der durch den "src"-Operanden im Spezial-Ein-/Ausgabeadreßraum angegebenen 16-Bit-Adresse wird in das "dst"-Register geladen.

Mnemonik: **SIN dst,src / SINB dst,src**
Operation: dst := src

Adressierungsart	Zyklen Wort/Byte
dst src	
R DA	12

Befehlsformat: F7.1

W/B: 0011101wdst.0101

Beispiel: SIN R2, @R5
Wenn in R5 die E/A-Adresse %4500 steht, wird von dieser Adresse ein Wort in das Register R2 eingelesen.

SIND — Special Input and Decrement — (PB) — SIND

Der SIND-Befehl dient zur Eingabe von Datenblöcken aus dem Spezial-Ein-/Ausgabeadreßraum in den Speicheradreßraum.
Der Inhalt der durch das "src"-Register im Spezial-Ein-/Ausgabeadreßraum angegebenen 16-Bit-Adresse wird auf die durch das "dst"-Register angegebene Speicheradresse geladen.
Das "dst"- und das "r"-Register werden automatisch dekrementiert.

Mnemonik: **SIND dst,src,r / SINDB dst,src,r**
Operation: dst := src
Autodekrement dst (-1 Byte -2 Wort) r := r - 1

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	21

Befehlsformat: F6.4

W/B: 0011101wsrc.1001
0000.r..dst.1000

Beispiel: SIND @RR4, @R7, R8
Wenn Register RR4 im segmentierten Mode den Wert %83003000, R7 den Wert %1006 und R8 den Wert 4 enthält, so steht nach dem SIND-Befehl im Segment 3 auf der Adresse %3000 der von der E/A-Adresse %1006 gelesene 16-Bit-Wert. Das Register RR4 enthält den Wert %83002FFE, R7 verändert sich nicht, und in R8 steht 3.

SINDR Special Input, Decrement and Repeat (PB) SINDR

Der SINDR-Befehl dient zur Eingabe von Datenblöcken aus dem Spezial-Ein-/Ausgabeadreßraum in den Speicheradreßraum.
Der Inhalt der durch das "src"-Register im Spezial-Ein-/Ausgabeadreßraum angegebenen 16-Bit-Adresse wird auf die durch das "dst"-Register angegebene Speicheradresse geladen.
Das "dst"- und das "r"-Register werden automatisch dekrementiert.
Der Befehl wird wiederholt, bis r - 1 = 0 ist.

Mnemonik: **SINDR dst,src,r / SINDRB dst,src,r**
Operation: dst := src
Autodekrement dst (-1 Byte -2 Wort) r := r - 1 bis r = 0
Flags: Z: Undefiniert
V: Ist gesetzt.

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	11+10n

Befehlsformat: F6.4

W/B: 0011101wsrc.1001
0000.r..dst.0000

Beispiel: SINDR @RR4, @R7, R8
Wenn Register RR4 im segmentierten Mode den Wert %83003000, R7 den Wert %1006 und R8 den Wert 2 enthält, so steht nach dem SINDR-Befehl im Segment 3 auf der Adresse %3000 der erste der von der E/A-Adresse %1006 gelesenen 16-Bit-Werte. Der zweite Wert steht auf der Adresse %2FFE. Das Register RR4 enthält den Wert %83002FFC, R7 verändert sich nicht, und in R8 steht 0.

SINI Special Input and Increment (PB) SINI

Der SINI-Befehl dient zur Eingabe von Datenblöcken aus dem Spezial-Ein-/Ausgabeadreßraum in den Speicheradreßraum.
Der Inhalt der durch das "src"-Register im Spezial-Ein-/Ausgabeadreßraum angegebenen 16-Bit-Adresse wird auf die durch das "dst"-Register angegebene Speicheradresse geladen. Das "dst"-Register wird in- und das "r"-Register wird automatisch dekrementiert.

Mnemonik: **SINI dst,src,r / SINIB dst,src,r**
Operation: dst := src
Autoinkrement dst (+1 Byte +2 Wort) r := r - 1

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	21

Befehlsformat: F6.4

W/B: 0011101wsrc.0001
0000.r..dst.1000

Beispiel: SINI @RR4, @R7, R8
Wenn Register RR4 im segmentierten Mode den Wert %83003000, R7 den Wert %1006 und R8 den Wert 4 enthält, so steht nach dem SINI-Befehl im Segment 3 auf der Adresse %3000 der von der E/A-Adresse %1006 gelesene 16-Bit-Wert. Das Register RR4 enthält den Wert %83003002, R7 verändert sich nicht, und in R8 steht 3.

SINIR — Special Input, Increment and Repeat (PB) SINIR

Der SINIR-Befehl dient zur Eingabe von Datenblöcken aus dem Spezial-Ein-/Ausgabeadreßraum in den Speicheradreßraum.
Der Inhalt der durch das "src"-Register im Spezial-Ein-/Ausgabeadreßraum angegebenen 16-Bit-Adresse wird auf die durch das "dst"-Register angegebene Speicheradresse geladen. Das "dst"- wird in- und das "r"-Register automatisch dekrementiert. Der Befehl wird wiederholt, bis r - 1 = 0 ist.

Mnemonik: **SINIR dst,src,r / SINIRB dst,src,r**
Operation: dst := src
Autoinkrement dst (+1 Byte +2 Wort) r := r - 1 bis r = 0
Flags: Z: Undefiniert
V: Ist gesetzt.

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	11+10n

Befehlsformat: F6.4

W/B: 0011101wsrc.0001
0000.r..dst.0000

Beispiel: SINIR @R4, @R7, R8
Wenn Register R4 den Wert %3000, R7 den Wert %1006 und R8 den Wert 2 enthält, so steht nach dem SINIR-Befehl auf der Adresse %3000 der erste der von der E/A-Adresse %1006 gelesenen 16-Bit-Werte. Der zweite Wert steht auf der Adresse %3002. Das Register R4 enthält %3004, R7 verändert sich nicht, und in R8 steht 0.

SOTDR — Special Output, Decrement and Repeat (PB) SOTDR

Der SOTDR-Befehl dient zur Ausgabe von Datenblöcken aus dem Speicheradreßraum in den Spezial-Ein-/Ausgabeadreßraum.
Der Inhalt der durch das "src"-Register im Speicheradreßraum angegebenen Adresse wird auf die durch das "dst"-Register angegebene 16-Bit-Adresse im Spezial-Ein-/Ausgabeadreßraum geladen. Das "src"- und "r"-Register werden automatisch dekrementiert. Der Befehl wird wiederholt, bis r - 1 = 0 ist.

Mnemonik: **SOTDR dst,src,r / SOTDRB dst,src,r**
Operation: dst := src
Autodekrement src (-1 Byte -2 Wort) r := r - 1 bis r = 0
Flags: Z: Undefiniert
V: Ist gesetzt.

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	11+10n

Befehlsformat: F6.4

W/B: 0011101wsrc.1011
0000.r..dst.0000

Beispiel: SOTDR @R7, @RR4, R8
Wenn Register RR4 im segmentierten Mode die Speicheradresse %83003000, R7 die E/A-Adresse %1006 und R8 den Wert 2 enthält, so wird mit dem SOTDR-Befehl der Inhalt der Adresse %3000 im Segment 3 in die E/A-Adresse %1006 geladen. Der zweite 16-Bit-Wert steht auf der Adresse %2FFE. Das Register RR4 enthält den Wert %83002FFC, R7 verändert sich nicht, und in R8 steht 0.

SOTIR Special Output, Increment and Repeat (PB) SOTIR

Der SOTIR-Befehl dient zur Ausgabe von Datenblöcken aus dem Speicheradreßraum in den Spezial-Ein-/Ausgabeadreßraum.
Der Inhalt der durch das "src"-Register im Speicheradreßraum angegebenen Adresse wird auf die durch das "dst"-Register angegebene 16-Bit-Adresse im Spezial-Ein-/Ausgabeadreßraum geladen.
Das "src"-Register wird in- und das "r"-Register wird automatisch dekrementiert.
Der Befehl wird wiederholt bis r - 1 = 0 ist.

Mnemonik: **SOTIR dst,src,r / SOTIRB dst,src,r**
Operation: dst := src
Autoinkrement src (+1 Byte +2 Wort) r := r - 1 bis r = 0
Flags: Z: Undefiniert
V: Ist gesetzt.

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	11+10n

Befehlsformat: F6.4

W/B: 0011101wsrc.0011
0000.r..dst.0000

Beispiel: SOTIR @R7, @RR4, R8
Wenn Register RR4 im segmentierten Mode die Speicheradresse %83003000, R7 die E/A-Adresse %1006 und R8 den Wert 2 enthält, so wird mit dem SOTIR-Befehl der Inhalt der Adresse %3000 im Segment 3 in die E/A-Adresse %1006 geladen. Der zweite 16-Bit-Wert steht auf der Adresse %3002. Das Register RR4 enthält die Speicheradresse %83003004, R7 verändert sich nicht, und in R8 steht 0.

SOUT Special Output (PB) SOUT

Der SOUT-Befehl dient zur Ausgabe von Daten in den Spezial-Ein-/Ausgabeadreßraum. Der Inhalt des "src"-Registers wird in den "dst"-Operanden im Spezial-Ein-/Ausgabeadreßraum geladen. Adressen im Spezial-Ein-/Ausgabeadreßraum sind 16-Bit-Adressen.

Mnemonik: **SOUT dst,src / SOUTB dst,src**
Operation: dst := src

Adressierungsart	Zyklen Wort/Byte
dst src	
DA R	12

Befehlsformat: F7.2

W/B: 0011101wsrc.0111

Beispiel: SOUT @R5, R2
Wenn in R5 die E/A-Adresse %4500 steht, wird in diese Adresse ein Wort aus dem Register R2 geladen.

SOUTD — Special Output and Decrement — (PB) — SOUTD

Der SOUTD-Befehl dient zur Ausgabe von Datenblöcken aus dem Speicheradreßraum in den Spezial-Ein-/Ausgabeadreßraum.
Der Inhalt der durch das "src"-Register im Speicheradreßraum angegebenen Adresse wird auf die durch das "dst"-Register angegebene 16-Bit-Adresse im Spezial-Ein-/Ausgabeadreßraum geladen.
Das "src"- und das "r"-Register werden automatisch dekrementiert.

Mnemonik: **SOUTD dst,src,r / SOUTDB dst,src,r**
Operation: dst := src
Autodekrement src (-1 Byte -2 Wort) r := r - 1

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	21

Befehlsformat: F6.4

W/B: 0011101wsrc.1011
0000.r..dst.1000

Beispiel: SOUTD @R4, @RR4, R8
Wenn Register RR4 im segmentierten Mode die Speicheradresse %83003000, R7 die E/A-Adresse %1006 und R8 den Wert 4 enthält, so wird mit dem SOUTD-Befehl der Inhalt der Adresse %3000 im Segment 3 in die E/A-Adresse %1006 geladen. Das Register RR4 enthält die Speicheradresse %83002FFE, R7 verändert sich nicht, und in R8 steht 3.

SOUTI — Special Output and Increment — (PB) — SOUTI

Der SOUTI-Befehl dient zur Ausgabe von Datenblöcken aus dem Speicheradreßraum in den Spezial-Ein-/Ausgabeadreßraum.
Der Inhalt der durch das "src"-Register im Speicheradreßraum angegebenen Adresse wird auf die durch das "dst"-Register angegebene 16-Bit-Adresse im Spezial-Ein-/Ausgabeadreßraum geladen.
Das "src"-Register wird in- und das "r"-Register wird automatisch dekrementiert.

Mnemonik: **SOUTI dst,src,r / SOUTIB dst,src,r**
Operation: dst := src
Autoinkrement src (+1 Byte +2 Wort) r := r - 1

Adressierungsart	Zyklen Wort/Byte
dst src r	
IR IR R	21

Befehlsformat: F6.4

W/B: 0011101wsrc.0011
0000.r..dst.1000

Beispiel: SOUTI @R7, @RR4, R8
Wenn Register RR4 im segmentierten Mode die Speicheradresse %83003000, R7 die E/A-Adresse %1006 und R8 den Wert 4 enthält, so wird mit dem SOUTI-Befehl der Inhalt der Adresse %3000 im Segment 3 in die E/A-Adresse %1006 geladen. Das Register RR4 enthält die Speicheradresse %83003002, R7 verändert sich nicht, und in R8 steht 3.

3.5.9. CPU-Steuerbefehle

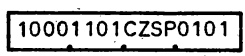
COMFLG Complement Flag (PB) **COMFLG**

Mit dem COMFLG-Befehl besteht die Möglichkeit Flags zu negieren. Dabei können mehrere oder auch ein einziges Flag im Befehl angegeben werden. Die angegebenen Flags werden negiert, nicht angegebene Flags bleiben unverändert. Bei der Angabe der Flags im Befehl ist zu beachten, daß P- und V-Flag durch ein Bit im FCW repräsentiert werden.

Mnemonik: **COMFLAG flag** flags: C, Z, S, P, V
Operation: FLAGS (4,7) := FLAGS (4,7) XOR Befehl (4,7)
Flags: C: Wenn im Befehl notiert, wird es negiert.
Z: Wenn im Befehl notiert, wird es negiert.
S: Wenn im Befehl notiert, wird es negiert.
P/V: Wenn im Befehl notiert, wird es negiert.

Befehlsformat: F9.1 P: P/V

Zyklen
7



Beispiel: COMFLG Z, C
Wenn das Zero-Flag = 1 und das Carry-Flag = 0 ist, so werden mit dem COMFLG-Befehl die Inhalte ausgetauscht. Das Zero-Flag ist danach = 0 und das Carry-Flag ist = 1.

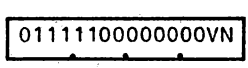
DI Disable Interrupt (PB) **DI**

Der DI-Befehl dient zum Sperren der Abarbeitung von vektorisierten Interrupts (VI) oder nichtvektorisierten Interrupts (NVI). Mit diesem Befehl wird das VI- oder NVI-Bit im Flag- und Controlwort rückgesetzt; war es bereits 0, wird es nicht verändert. Im Befehl können ein oder beide Operanden in beliebiger Reihenfolge angegeben werden.

Mnemonik: **DI int**
Operation: VI := 0 wenn int = VI
NVI:= 0 wenn int = NVI

Befehlsformat: F9.2 V: VI
N: NVI

Zyklen
7



Beispiel:
```
NVI.SERVICE: DI.    VI              !VI sperren!
             PUSHL  @R15, RR0       !R0 BIS R5 retten!
             PUSHL  @R15, RR2
             PUSHL  @R15, RR4
             LD     R2, 12(R15)     !Kennwort laden!
             CLRB   RH2             !R2 := Kennwort!
             SLA    R2              !Multiplikation mit 2!
             LD     R3, NVI.ADR(R3) !Adresse der NVI-!
                                    !Rout. aus der Tabelle!
      NVI.RUF: CALL  @R3            !Ruf der geford. Rout.!
```

EI — Enable Interrupt — (PB) — EI

Der EI-Befehl dient zur Freigabe der Abarbeitung von vektorisierten Interrupts (VI) oder nichtvektorisierten Interrupts (NVI). Mit diesem Befehl wird das VI- oder NVI-Bit im Flag- und Controlwort gesetzt; war es bereits 1, wird es nicht verändert. Im Befehl können ein oder beide Operanden in beliebiger Reihenfolge angegeben werden.

Mnemonik: **DI int** int: VI, NVI
Operation: VI := 1 wenn int = VI
NVI:= 1 wenn int = NVI

Zyklen
7

Befehlsformat: F9.2 V: VI
N: NVI

01111100000001VN

Beispiel:
```
NVI.RUF: CALL @R3          !Ruf der erford. Rout.!
         POPL RR4, @R15    !R0 bis R5 rueckspeichern!
         POPL RR2, @R15
         POPL RR0, @R15
         EI VI             !VI freigeben!
         IRET              !Rueckkehr in das unterbrochene Progr.!
```

HALT — Halt — (PB) — HALT

Der HALT-Befehl wird benutzt zur Synchronisation der Befehlsabarbeitung durch CPU mit externen Ereignissen. Die CPU führt während des HALT-Befehls Refresh-Operationen aus und reagiert auf BUSREQ-Anforderungen, ansonsten führt sie bis zum Auftreten eines Interrupt- oder Reset-Signals keine weiteren Operationen aus.
Nach einem Interrupt wird die Befehlsabarbeitung mit dem nach dem HALT-Befehl stehenden Befehl fortgesetzt.

Mnemonik: **HALT**
Operation: CPU := Stop-/Refreshzustand

Zyklen
8+3n

Befehlsformat: F9.3

0111101000000000

n: Anzahl der Perioden ohne Interrupt

Beispiel:
```
LDA R2, BUFFER    !Adresse des Ausgabepuffers laden!
LDA R3, PRINTER   !Adresse des E/A Geraetes laden!
LD  R4, #BFFLEN   !Pufferlaenge laden!
OTIR R3, R2, R4   !Puffer leeren!
HALT              !Fertigmeldung durch Interrupt!
                  !abwarten!
```

LDCTL **Load Control Register** **(PB)** **LDCTL**

Der LDCTL-Befehl dient dem Laden oder Rückladen der Steuerregister (Controlregister) der CPU. Der Transfer erfolgt zwischen den Steuerregistern und den allgemeinen Registern der CPU. Die Steuerregister werden mit den folgenden Bezeichnungen adressiert:

FCW	Flag- und Controlwort	
FLAGS	Flagbyte des Flag- und Controlworts	
NSPSEG	Normalstackpointer-Segmentnummer	(segmentierter Mode)
NSPOFF	Normalstackpointer-Offset	(segmentierter Mode)
NSP	Normalstackpointer	(nichtsegmentierter Mode)
PSAPSEG	Programmstatusareapointer-Segmentnummer	(segmentierter Mode)
PSAPOFF	Programmstatusareapointer-Offset	(segmentierter Mode)
PSAP	Programmstatusareapointer	(nichtsegmentierter Mode)

Das NSP-Register ist mit dem NSPOFF-Register und das PSAP-Register mit dem PSAPOFF-Register identisch.
Die in den Steuerregistern nicht genutzten Bits sollten beim Laden der Steuerregister im Interesse der Lauffähigkeit der Programme auf zukünftigen U8000-Mikroprozessoren gelöscht sein. Diese Bits werden ggf. bei der Weiterentwicklung der CPU eine Bedeutung erlangen.

Mnemonik: **LDCTL dst, src**
Operation: dst := src

LDCTL Load Control Register (PB) LDCTL

Load FCW

Das Flag- und Controlwort wird aus einem Wortregister geladen. Die Bits 0, 1, 8, 9, 10 beim U8001 und Bit 15 beim U8002 werden nicht beeinflußt.

Mnemonik: **LDCTL FCW,r**
Operation: FCW(2,7) := src(2,7)
FCW(11,15) := src(11,15)
Flags: Die Flags werden aus dem "src"-Register geladen.

Zyklen
7

Befehlsformat: F8.1

W: 01111101src.1010

Beispiel:
```
LDCTL R4, FCW   !FCW in R4 laden!
RES R4, #15     !Reset SEG-Bit!
LDCTL FCW, R4   !CPU arbeitet nichtsegmentiert!
```

Read FCW

Das Flag- und Controlwort wird in ein Wortregister geladen.

Mnemonik: **LDCTL r, FCW**
Operation: dst(2,7) := FCW(2,7)
dst(11,15) := FCW(11,15)
dst(0,1,8,9,10) := 0

Zyklen
7

Befehlsformat: F8.2

W: 01111101dst.0010

Beispiel:
```
LDCTL R4, FCW   !FCW in R4 laden!
RES R4, #12     !Reset VIE-BIT, CPU akzeptiert!
LDCTL FCW, R4   !keine vektorisierten Interrupts!
```

LDCTL **Load Control Register** **(PB)** **LDCTL**

Load REFRESH

Das Refreshregister wird aus einem Wortregister geladen. Das Bit 0 des Refreshregisters wird nicht beleinflußt.

Mnemonik: **LDCTL REFRESH,r**
Operation: REFRESH(1,15) := src(1,15)

Zyklen
7

Befehlsformat: F8.1

W: 01111101src.1011

Beispiel: LD R4, #%9E00
LDCTL REFRESH, R4 !Set RE-Bit, Rate=15, CTR=0 !
!Refreshzyklus bei einem 4-MHz-Takt!
!ergibt sich zu 4 * 15 *'256ns = 15 Mikrosek.!

Read REFRESH

Das Refreshregister wird in ein Wortregister geladen. Der Inhalt des Refreshregisters kann als Zufallszahl genutzt werden.

Mnemonik: **LDCTL r, REFRESH**
Operation: dst(0) := 0
dst(1,8) := REFRESH (1,8)
dst(9,15) := zufällig

Zyklen
7

Befehlsformat: F8.2

W: 01111101dst.0011

Beispiel: LDCTL R4, REFRESH R4:=ZUFALLSZAHL

LDCTL **Load Control Register** **(PB)** **LDCTL**

Load PSAPSEG

Das Programmstatusareapointer-Segmentregister wird im segmentierten Mode aus einem Wortregister der U8001-CPU geladen. Die Bits 0 bis 7 und 15 des Registers werden nicht beeinflußt.

Mnemonik: **LDCTL PSAPSEG,r**
Operation: PSAPSEG(8,14) := src(8,14)

Zyklen
7

Befehlsformat: F8.1

W: 01111101src.1100

Beispiel: LD R4, #%0300
LDCTL PSAPSEG, R4 SEGMENTNUMMER = 3

Read PSAPSEG

Der Inhalt des Programmstatusareapointer-Segmentregisters wird im segmentierten Mode in ein Wortregister der U8001-CPU geladen.

Mnemonik: **LDCTL r, PSAPSEG**
Operation: dst(0,7) := 0
dst(8,14) := PSAPSEG (8,14)
dst(15) := 0

Zyklen
7

Befehlsformat: F8.2

W: 01111101dst.0011

Beispiel: LDCTL R4, PSAPSEG !R4:= aktuelle Segmentnummer!
!der Programmstatusarea!

LDCTL Load Control Register (PB) LDCTL

Load PSAPOFF,PSAP

Das Programmstatusareapointer-Offsetregister (PSAPOFF) wird aus einem Wortregister der U8001-CPU geladen. Das Programmstatusareapointer-Register (PSAP) ist mit ihm identisch, es wird im nichtsegmentierten Mode benutzt. Die Bits 0 bis 8 des Registers werden nicht beeinflußt.

Mnemonik: **LDCTL PSAPOFF,r / LDCTL PSAP,r**
Operation: PSAPOFF,PSAP(8,15) := src(8,15)

Zyklen
7

Befehlsformat: F8.1

W: 01111101src.1101

Beispiel: LD R4, #%2400
LDCTL PSAPOFF, R4 !Offset = %2400!

Read PSAPOFF,PSAP

Der Inhalt des Programmstatusareapointer-Offsetregisters wird in ein Wortregister der U8001-CPU geladen. Das Programmstatusareapointer-Register (PSAP) ist mit ihm identisch, es wird im nichtsegmentierten Mode benutzt. Die Bits 0 bis 8 des "dst"-Registers sind = 0.

Mnemonik: **LDCTL r, PSAPOFF / LDCTL r, PSAP**
Operation: dst(0,7) := 0
dst(8,15) := PSAPOFF (8,15)

Zyklen
7

Befehlsformat: F8.2

W: 01111101dst.0101

Beispiel: LDCTL R4, PSAPOFF !R4:= aktueller Offset!
!der Programmstatusarea!

LDCTL **Load Control Register** **(PB)** **LDCTL**

Load NSPSEG

Das Normalstackpointer-Segmentregister (NSPSEG) wird im segmentierten Mode aus einem Wortregister der U8001-CPU geladen, es ist in einem zusätzlichen Register der U8001-CPU gespeichert.
Im Nomalmode ist Register R14 das Normalstackpointer-Segmentregister.

Mnemonik: **LDCTL NSPSEG,r**
Operation: NSPSEG := src

Zyklen
7

Befehlsformat: F8.1

W: 01111101src.1110

Beispiel: LD R14, #%8300
LDCTL NSPSEG, R14 !Segmentnummer = 3!

Read NSPSEG

Der Inhalt des Normalstackpointer-Segmentregisters wird im segmentierten Mode in ein Wortregister der U8001-CPU geladen, er ist in einem Steuerregister der U8001-CPU gespeichert.
Im Nomalmode ist Register R14 das Normalstackpointer-Segmentregister.

Mnemonik: **LDCTL r, NSPSEG**
Operation: dst := NSPSEG

Zyklen
7

Befehlsformat: F8.2

W: 01111101dst.0110

Beispiel: LDCTL R4, NSPSEG !R4:= aktuelle Segmentnummer!
!des Normalstackpointers!

LDCTL Load Control Register (PB) LDCTL

Load NSPOFF,NSP

Das Normalstackpointer-Offsetregister (NSPOFF) wird im Systemmode aus einem Wortregister der U8001-CPU geladen. Das Normalstackpointer-Register (NSP) ist mit ihm identisch, es wird im nichtsegmentierten Mode benutzt. Im Normalmode ist Register R15 das Normalstackpointer-Segmentregister.

Mnemonik: **LDCTL NSPOFF,r / LDCTL NSP,r**
Operation: NSPOFF,NSP(8,15) := src(8,15)

Zyklen
7

Befehlsformat: F8.1

W: 01111101src.1111

Beispiel: LD R4, #%2400
LDCTL NSPOFF, R4 !Offset = %2400!

Read NSPOFF,NSP

Der Inhalt des Normalstackpointer-Offsetregisters wird im Systemmode in ein Wortregister der U8001-CPU geladen. Das Normalstackpointerregister (NSP) ist mit ihm identisch, es wird im nichtsegmentierten Mode benutzt. Im segmentierten Normalmode ist Register R15 das Normalstackpointer-Offsetregister, im nichtsegmentierten Normalmode enthält es den Normalstackpointer.

Mnemonik: **LDCTL r, NSPOFF / LDCTL r, NSP**
Operation: dst := NSPOFF, NSP

Zyklen
7

Befehlsformat: F8.2

W: 01111101dst.0111

Beispiel: LDCTL R4, NSP !R4:= aktueller Wert!
!des Normalstackpointers!

LDCTLB Load Control Byte LDCTLB

Load FLAGS

Das Flagbyte des Flag- und Controlworts wird aus einem Byteregister geladen.

Mnemonik: **LDCTLB FLAGS,r**
Operation: FLAGS(2,7) := src(2,7)
Flags: Die Flags werden aus dem "src"-Register geladen.

Zyklen
7

Befehlsformat: F8.1

B: 10001100src.1001

Beispiel: LDB RH4, #%80
LDCTLB FLAGS, RH4 Carry := 1

Read FLAGS

Die sechs höherwertigen Bits des Flagbyte aus dem Flag- und Controlwort werden in ein Byteregister geladen. Die Bits 0 und 1 des Byteregisters sind = 0.

Mnemonik: **LDCTL r, FLAGS**
Operation: dst(0,1) := 0
dst(2,7) := FLAGS(2,7)

Zyklen
7

Befehlsformat: F8.2

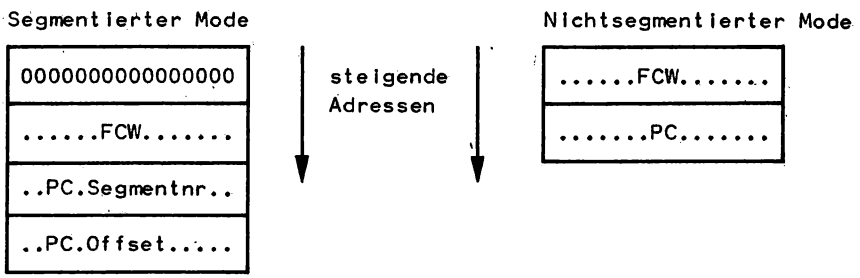
B: 10001100dst.0001

Beispiel: LDCTLB RH4, FLAGS !RH4:= aktueller Wert!
!des Flagbyte im FCW!

LDPS Load Program Status (PB) LDPS

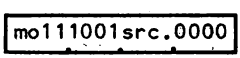
Mit dem LDPS-Befehl wird ein neuer Programmstatus (PS), bestehend aus dem Flag- und Controlwort (FCW) und dem Befehlszähler (PC), aus dem Speicheradreßraum in die CPU geladen. Der neue Programmstatus wird erst mit dem nächsten Befehl, der auf der im PC stehenden Adresse abgearbeitet wird, aktiv. Dieser Befehl bietet eine einfache und schnelle Möglichkeit, Arbeitsmodi der CPU zu wechseln.

Mnemonik: **LDPS src**
Operation: PS := src
Flags: Die Flags werden aus dem "src"-Operanden geladen.



Adressierungsart		Zyklen		
src	mo	NS	SS	SL
IR	00	12	16	16
DA	01	16	20	22
X	01	17	20	23

Befehlsformat: F2.3 DA: src=0
X : src<>0



Beispiel: LDPS @RR4
Steht im segmentierten Mode in RR4 die Adresse %87002400 und stehen im Segment 7 ab der Adresse %2400 nachfolgend die Werte %0000, %C000, %8500 und %4800, so arbeitet der Prozessor anschließend im segmentierten und Systemmode den ersten Befehl im Segment 5 auf der Adresse %4800 ab.

MBIT — Multi-Micro Bit Test — (PB) — MBIT

Der MBIT-Befehl dient der Synchronisation von mehreren Prozessoren bei der Nutzung gemeinsamer Hardwareressourcen.
Er testet den Status des M_I-Pins (aktiv = Low), ist es inaktiv, wird das S-Flag gesetzt, ist es aktiv, wird das S-Flag gelöscht.

Mnemonik: **MBIT**
Operation: S := NOT M_I
Flags: Z: Undefiniert
S: Ist nur gesetzt wenn M_I = 0 ist.

Zyklen
7

Befehlsformat: F9.3

0111101100001010

Beispiel:
```
WAIT: MBIT          !Test Multimikro-Eingang!
      JR PL, WAIT   !Warten, bis Ressource frei ist!
WORK:
```


SETFLG — Set Flag — SETFLG

Mit dem SETFLG-Befehl besteht die Möglichkeit, Flags zu setzen. Dabei können mehrere oder auch ein einziges Flag im Befehl angegeben werden. Die angegebenen Flags werden gesetzt, nicht angegebene Flags bleiben unverändert. Bei der Angabe der Flags im Befehl ist zu beachten, daß das P- und V-Flag durch ein Bit im FCW repräsentiert werden.

Mnemonik: **SETFLG flag** flags: C, Z, S, P, V
Operation: FLAGS (4,7) := FLAGS (4,7) OR Befehl (4,7)
Flags: C: Wenn im Befehl notiert, wird es gesetzt.
Z: Wenn im Befehl notiert, wird es gesetzt.
S: Wenn im Befehl notiert, wird es gesetzt.
P/V: Wenn im Befehl notiert, wird es gesetzt.

Zyklen
7

Befehlsformat: F9.1 P: P/V

10001101CZSP0001

Beispiel: SETFLG Z, C
Wenn das Zero-Flag = 1 und das Carry-Flag = 0 ist, so werden mit dem SETFLG-Befehl die Inhalte beider Flags zu Eins gesetzt. Das Zero-Flag ist danach = 1, und das Carry-Flag ist = 1.

MREQ Multi-Micro Request (PB) MREQ

Der MREQ-Befehl dient der Synchronisation von mehreren Prozessoren bei der Nutzung gemeinsamer Hardwareressourcen. Eine Anforderung auf eine gemeinsame Hardwareressource wird über das Multimikro-In-(M_i) und das Multimikro-Out-Pin (M_o) (aktiv = Low) getestet und angezeigt. Das Zero- und Sign-Flag zeigen den Zustand der Ressource nach dem MREQ-Befehl an.
Wenn das M_i-Signal inaktiv ist, wird das Sign-Flag gesetzt, das M_o-Signal wird aktiv, und die im "dst"-Register stehende Konstante wird dekrementiert. Mit der damit verbundenen Wartezeit sollen Signallaufzeiten im System in die Synchronisation einbezogen werden, da in der Zwischenzeit die Anforderung eines anderen Prozessors schon akzeptiert werden konnte. Nachdem die Konstante den Wert Null besitzt, wird das Zero-Flag gesetzt und das M_i-Pin nochmals getestet. Ist es noch inaktiv, wird das Sign-Flag gesetzt, und die Ressource kann benutzt werden, andernfalls ist die Ressource belegt, und der Zugriff muß wiederholt werden.

S-Flag	Z-Flag	M_i	Zustand
0	0	High	Anforderung liegt nicht vor, Ressource ist belegt.
0	1	High	Anforderung nicht angenommen, Ressource ist belegt.
1	1	Low	Anforderung angenommen

Mnemonik: MREQ dst
Operation: Wenn Z := 0
Flags: Z: Ist nur gesetzt, wenn eine Anforderung vorliegt.
S: Ist nur gesetzt, wenn eine Anforderung vorliegt und akzeptiert wurde.

Zyklen
12+7n

Befehlsformat: F8.2 R: dst

W: 01111011dst.1101

n: Größe der bei Annahme der Anforderung im "dst"-Register zu dekrementierenden Konstante. Wird die Anforderung nicht angenommen, ist n = 0.

Beispiel:

```
RESSOURCE:
        LD R4, #10          !Verzoegerungszeit!
        MREQ R4

        JR MI, FREE
        JR Z, NOT.ACCEPT

NOT.FREE:                   !Anforderung nicht angenommen!

        .
NOT.ACCEPT                  !Anforderung nicht akzeptiert!

        .
        JR RESSOURCE        !Versuch wiederholen!

FREE:                       !Ressource nutzen!
```

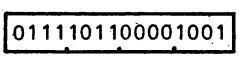
MRES — Multi-Micro Reset — (PB) — MRES

Der MRES-Befehl dient der Synchronisation von mehreren Prozessoren bei der Nutzung gemeinsamer Hardwareressourcen. Er setzt den Status des M_0-Pins auf High (aktiv = Low) und gibt damit eine belegte Ressource für andere Prozessoren frei.

Mnemonik: MRES
Operation: M_0 := High

Zyklen
5

Befehlsformat: F9.3




```
Beispiel: FREE:      !Arbeit mit der gem. Ressource!
          .
          MRES       !Freigabe der gemeins. Ressour.!
```

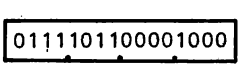
MSET — Multi-Micro Set — (PB) — MSET

Der MSET-Befehl dient der Synchronisation von mehreren Prozessoren bei der Nutzung gemeinsamer Hardwareressourcen. Er setzt den Status des M_0-Pins auf Low (aktiv = Low), und zeigt damit an, daß eine gemeinsame Ressource durch diese CPU belegt ist oder daß diese CPU eine Ressource anfordert.

Mnemonik: MSET
Operation: M_0 := Low

Zyklen
5

Befehlsformat: F9.3




```
Beispiel: REQUEST:
          MSET          !Gemeinsame Ressource anfordern!

          WAIT:  MBIT           !Test Multimikro-Eingang!
                 JR PL, WAIT    !Warten, bis Ressource frei ist!
          WORK:
```

NOP — No Operation — NOP

Es wird keine Operation der CPU ausgeführt. Der Befehl kann zur programmtechnischen Realisierung kurzer Verzögerungszeiten genutzt werden.

Mnemonik: **NOP**
Operation:

Zyklen
7

Befehlsformat: F9.3

1000110100000111

Beispiel:
```
OUTB @R5, %#88
NOP
INB  RL0, @R6
```
Nach der Ausgabe einer Steuerinformation, z. B. durch einen OUTB-Befehl, kann hardwareseitig Zeit bis zur Bereitstellung der gewünschten Daten erforderlich sein.

RESFLG — Reset Flag — RESFLG

Mit dem RESFLG-Befehl besteht die Möglichkeit, Flags zu löschen. Dabei können mehrere oder auch ein einziges Flag im Befehl angegeben werden. Die angegebenen Flags werden gelöscht, nicht angegebene Flags bleiben unverändert. Bei der Angabe der Flags im Befehl ist zu beachten, daß das P- und V-Flag durch ein Bit im FCW repräsentiert werden.

Mnemonik: **RESFLG flag** flags: C, Z, S, P, V
Operation: FLAGS (4,7) := FLAGS (4,7) AND NOT Befehl (4,7)
Flags:
C: Wenn im Befehl notiert, wird es gelöscht.
Z: Wenn im Befehl notiert, wird es gelöscht.
S: Wenn im Befehl notiert, wird es gelöscht.
P/V: Wenn im Befehl notiert, wird es gelöscht.

Befehlsformat: F9.1 P: P/V

Zyklen
7

10001101CZSP0011

Beispiel: RESFLG Z, C
Wenn das Zero-Flag = 1 und das Carry-Flag = 0 ist, so werden mit dem RESFLG-Befehl die Inhalte beider Flags zu Null gesetzt. Das Zero-Flag ist danach = 0, und das Carry-Flag ist = 0.

SETFLG Set Flag SETFLG

Mit dem SETFLG-Befehl besteht die Möglichkeit, Flags zu setzen. Dabei können mehrere oder auch ein einziges Flag im Befehl angegeben werden. Die angegebenen Flags werden gesetzt, nicht angegebene Flags bleiben unverändert. Bei der Angabe der Flags im Befehl ist zu beachten, daß das P- und V-Flag durch ein Bit im FCW repräsentiert werden.

Mnemonik: **SETFLG flag** flags: C, Z, S, P, V
Operation: FLAGS (4,7) := FLAGS (4,7) OR Befehl (4,7)
Flags: C: Wenn im Befehl notiert, wird es gesetzt.
Z: Wenn im Befehl notiert, wird es gesetzt.
S: Wenn im Befehl notiert, wird es gesetzt.
P/V: Wenn im Befehl notiert, wird es gesetzt.

Befehlsformat: F9.1 P: P/V

Zyklen
7

10001101CZSP0001

Beispiel: SETFLG Z, C
Wenn das Zero-Flag = 1 und das Carry-Flag = 0 ist, so werden mit dem SETFLG-Befehl die Inhalte beider Flags zu Eins gesetzt. Das Zero-Flag ist danach = 1, und das Carry-Flag ist = 1.

3.5.10. EPU-Befehle

Load Memory from EPU **Load Memory from EPU**

Der "Load Memory from EPU"-Befehl dient dem Laden des Speichers von einer EPU (extended processing unit).
Die CPU führt die erforderlichen Adreßberechnungen aus und generiert "n" EPU-Zyklen. Die EPU liefert "n" Wörter, die in "n" aufeinanderfolgende Speicherplätze geschrieben werden.

Mnemonik: LVAL kode
Operation: Memory := EPU

Adressierungsart		Zyklen		
dst	mo	NS	SS	SL
IR	00	11+3n	11+3n	11+3n
X	01	15+3n	15+3n	18+3n
DA	01	14+3n	15+3n	17+3n

Befehlsformat:

L: mo001111dst.11ee eeeeeeeeeeeeen-1.

IR: dst<>0
X : dst<>0
DA: dst=0

Load EPU from Memory **Load EPU from Memory**

Der "Load EPU from Memory"-Befehl dient dem Laden der EPU-Register aus dem Speicher.
Die CPU führt die erforderlichen Adreßberechnungen aus und generiert "n" EPU-Zyklen. Die EPU liest "n" Wörter, die in "n" aufeinanderfolgenden Speicherplätzen stehen, ein.

Mnemonik: LVAL kode
Operation: EPU := Memory

Adressierungsart		Zyklen		
dst	mo	NS	SS	SL
IR	00	11+3n	11+3n	11+3n
X	01	15+3n	15+3n	18+3n
DA	01	14+3n	15+3n	17+3n

Befehlsformat:

L: mo001111src.01ee eeeeeeeeeeeeen-1.

IR: dst<>0
X : dst<>0
DA: dst=0

Load CPU from EPU

Load CPU from EPU

Der "Load CPU from EPU"-Befehl dient dem Laden der CPU-Register aus der EPU (extended processing unit).
Die CPU liest "n" Wörter aus der EPU in aufeinanderfolgende CPU-Register ein. Mit dem durch "dst" kodierten Register wird begonnen. Nach dem Register R15 folgt bei fortlaufendem Transfer wieder Register R0.

Mnemonik: LVAL kode
Operation: CPU := EPU (register)

Zyklen
11+3n

Befehlsformat:

L: 100011110eee00ee
..epu_dst...n-1.

Load EPU from CPU

Load EPU from CPU

Der "Load EPU from CPU"-Befehl dient dem Laden der EPU-Register aus der CPU.
Die EPU liest "n" Wörter aus aufeinanderfolgenden Registern der CPU ein. Mit dem durch "src" kodierten Register wird begonnen. Nach dem Register R15 wird bei fortlaufendem Transfer wieder das Register R0 der CPU gelesen.

Mnemonik: LVAL kode
Operation: EPU := CPU (register)

Zyklen
11+3n

Befehlsformat:

L: 100011110eee10ee
..epu_src...n-1.

Load FCW from EPU

Load FCW from EPU

Der "Load FCW from EPU" Befehl dient dem Laden des Flagbyte im Flag- und Controlwort der CPU.
Das Flagbyte im Flag- und Controlwort der CPU wird mit einem Datenbyte (AD_0 - AD_7) von der EPU geladen.
Der Inhalt des Registers R0 der CPU ist nach Ausführung des Befehls undefiniert.

Mnemonik: LVAL kode
Operation: Flags := EPU

Zyklen
14

Befehlsformat:

L: 10001110eeee00ee
eeee0000eeee0000

Load EPU from FCW

Load EPU from FCW

Der "Load EPU from FCW" Befehl dient dem Transfer des Flagbyte im Flag- und Controlwort der CPU in die EPU.
Das Flagbyte im Flag- und Controlwort der CPU wird mit einem Datenbyte (AD_0 - AD_7) von der CPU in die EPU geladen.

Mnemonik: LVAL kode
Operation: EPU := Flags

Zyklen
14

Befehlsformat:

L: 10001110eeee10ee
eeee0000eeee0000

Internal EPU Operation

Internal EPU Operation

Der Befehl "Internal EPU Operation" dient zum Einleiten interner EPU-Operationen. Die CPU behandelt diesen Befehl als NOP-Befehl.

Mnemonik: LVAL kode

Zyklen
14

Befehlsformat:

L: 10001110eeee01ee
eeeeeeeeeeeeeeee

3.6. Programmierbeispiele

3.6.1. Initialisierung eines U8001-Mikrorechnersystems

Das Beispiel zeigt die Organisation eines auf der physischen Speicheradresse 0 beginnenden U8001-Programms. Die U8001-CPU muß nach dem Start auf der Adresse %0002 ein Flag- und Controlwort für den segmentierten Systemmode lesen. Die nichtsegmentierte Arbeitsweise kann bei Bedarf nach dem Programmstart über eine Modifikation des FCW erfolgen.

```
!*********************************!
!Start des Initialisierungsprogramms!
!*********************************!

INIT module

!VEREINBARUNGEN!
global
ENTRY_          label

external
PSAREA          label    !RAM Bereich fuer die Programmstatusarea!
SYS_STACK       label    !High-Adresse des Systemstackspeicherbereichs!
NORM_STACK      label    !High-Adresse des Normalstackspeicherbereichs!
RAMBOT          label    !Beginn des RAM-Bereichs!
TTY             label    !Prozedur zur Displayansteuerung!
TTYINIT         label    !Prozedur zur Displayinitialisierung!
INBFF           array    [%80 BYTE]
CHRDEL          array    [3 BYTE]

$SECTION PROM

internal
        word := %0000
FLG_CW  word := %C000
SEG_NUM word := %8000
PCOFFS  word := ENTRY_

C_RIGHT array   [* BYTE]        := '(C) XYZ'

global ENTRY_ procedure
entry                           !CPU arbeitet segmentiert!
!Programmstatusareapointer-Segment laden!
        clr     r1
        ldctl   PSAPSEG, r1

        ld      r1, #%4000      !CPU in den nichtsegmentierten Mode!
        ldctl   FCW, r1         !bringen!
        ld      r1, #%8800      !REFRESH-Anpassung!
        ldctl   REFRESH, r1

!%200 W|rter im RAM ab RAMBOT loeschen!
        ld      r4, #RAMBOT
        ld      r2, #RAMBOT+2
        ld      @r4, #0
        ld      r1, #%200
        ldir    @r2, @r4, r1
```

```
!Programmstatusarea initialisieren!
        ld      r4, PS_AREAPROM
        ld      r2, PSAREA
        ld      r1, PS_END - PS_AREAPROM
        ldir    @r2, @r4, r1

!Puffer mit Space initialisieren!
        ld      INBFF, #%2020
        ld      r4, #INBFF
        ld      r2, #INBFF+2
        ld      r1, #%80
        ldir    @r2, @r4, r1

!Variable in den RAM laden!
        ld      r4, #VAR_LISTE_PROM
        ld      r2, #CHRDEL
        ldk     r1, #%03
        ldir    @r2, @r4, r1

!Stackpointer laden!
        ld      r14, #%8000     !NORMALSTACKPOINTER_SEGMENT!
        ld      r15, #NSP_OFF   !NORMALSTACKPOINTER_OFFSET!

!Programmstatusareapointer-Offset laden!
        ld      r1, #PSAREA
        ldctl   PSAP, r1
        jp      MODUL1
end ENTRY_

  internal
!VARIABLENLISTE: CHRDEL/LINDEL, ^Q/^S,  N_CNT!
VAR_LISTE_PROM
        array [3 word] := [%087f %1113 %0000]


!***********************************************!
!Initialisierungsfeld fuer die Programmstatusarea!
!***********************************************!

!SEGMENTIERTE VERSION!
PS_AREAPROM:
!                           ID     FCW    PCSEG PCOFF!
        array [4 word] := [%0000 %0000 %0000 %0000]  !reserviert!
        array [4 word] := [%0000 %4000 %8000 ERROR]  !nicht impl. Befehl!
        array [4 word] := [%0000 %4000 %8000 ERROR]  !privileg. Befehl!
        array [4 word] := [%0000 %C000 %8000 S_CALL] !System Call!
        array [4 word] := [%0000 %4000 %8000 VI_ERR] !Segmenttrap!
        array [4 word] := [%0000 %4000 %8000 NMIINT] !nichtmaskierb. Int.!
        array [4 word] := [%0000 %5000 %8000 ERROR]  !nichtvekt. Int.!
VEC_FCW array [2 word] := [%0000 %4000]              !Flag- und Controlw.!

!VECTORED INTERRUPT JUMP TABLE!
!                           SEG    OFFSET!
        array [2 word] := [%8000 TTYINT]
        array [2 word] := [%8000 TTY_EMPTY]
        array [2 word] := [%8000 ERROR]
        array [2 word] := [%8000 ERROR]
PS_END:
end INIT
```

3.6.2. System Call Routine

Die im Beispiel aufgeführte System Call Routine beinhaltet zwei schnelle Routinen (SC_0 und SC_1) und eine um 254 Routinen erweiterbare Anzahl von weiteren System Call Routinen. Die schnellen System-Call-Routinen dienen der Umschaltung zwischen dem segmentierten und dem nichtsegmentierten Mode. Alle anderen System-Call-Routinen werden über den indirekten Aufruf eines Unterprogramms aus einer Adreßliste von mehreren Unterprogrammen. Der PSAREA-Vereinbarung entsprechend befindet sich die CPU beim Eingang in die System-Call-Routine im segmentierten Mode.

```
!**********************!
global SC_ENTRY.procedure
!**********************!
entry                                   !CPU arbeitet im segmentierten Mode!
        push    @r14, r0                !RR0 retten!
        push    @r14, r2                !RR2 retten!
        push    @r14, r4                !RR4 retten!
        ldctl   r0, FCW                 !CPU in den nichtseg. Mode umschalten!
        res     r0, #%0F
        ldctl   FCW, r0
        cpb     3(r15), #0              !SC_0 = SEG_Arbeitsweise!
        jr      nz, SC_01

        set     4(r15), #15             !Segment_Bit im FCW ruecksetzen!
        jr      SC_END
SC_01:
        cpb     3(r15), #1              !SC_1 = NSEG_Arbeitsweise!.
        jr      nz, SC_X
        res     %04(r15), #15           !Segment_Bit im FCW ruecksetzen!
        jr      SC_END

SC_X:
        ldb     rh0, 3(r15)             !Nr. des System-Call laden!
        ldb     rl3, rh0
        clrb    rh3
        sub     r3, #3                  !SC_2 (TYWR) = 1.Adresse in SC_ADR!
        ld      r3, SC_ADR(r3)          !Adresse des Unterprogramms zur!
                                        !Ausfuehrung des System Call laden!
        ei      vi
        call    @r3                     !Aufruf des Unterprog. als System Call!
        ld      r3, r0
SC_END:
        ldctl   r0, FCW                 !CPU in den segmentierten Mode bringen!
        set     r0, #15
        ldctl   FCW, r0
        pop     r4, @r14                !RR4, RR2 und RR0 zurueckladen!
        pop     r2, @r14
        pop     r0, @r14
        iret

    end SC_ENTRY

SC_ADR: array [*word] := [GETA PUTA TTY BTOH16 CRLF]
```

3.6.3. Programmierung von U880-Peripheriebausteinen

In das 16-Bit-Mikrorechnersystem U8000 können die Peripherieschaltkreise des 8-Bit-Mikroprozessorsystems U880 einbezogen werden. Die Schaltkreise

U855-PIO (parallel input/output)
U856-SIO (seriell input/output)
U857-CTC (counter timer circuit)
U858-DMA (direct memory access)

sind unter Verwendung zusätzlicher Hardware zur Simulation des U880-RETI-Befehls (return from interrupt) in das U8000-System integrierbar. Nach Abarbeitung einer Interrupt-Serviceroutine muß der Interruptstatus des interruptenden Schaltkreises softwaremäßig zurückgesetzt werden. Der SIO besitzt diese Möglichkeit in seinem standardmäßigen Befehlssatz. An den Port A des SIO muß das Byte %38 ausgegeben werden. Damit kann der U880-RETI-Befehl für den SIO ersetzt werden.
Für die anderen Peripherieschaltkreisen des U880-Systems muß der U880-RETI-Befehl durch die Programmierung der Zusatzhardware simuliert werden. Diese Hardware wird programmmäßig über einen Ein-/Ausgabeport aktiviert, sie erzeugt während der Ausgabe des U880-RETI-Befehls den U880-M1-Zyklus für den PIO /4/. An diesen Port wird der Maschinenkode des U880-RETI-Befehls ausgegeben. Der Peripheriebaustein empfängt die Werte %ED und %4D nacheinander und setzt damit seinen Interruptstatus zurück.
Die U880-Peripheriebausteine sind in /4/ bereits ausführlich dargestellt, deshalb wird die Kenntnis ihres Aufbaus und der Arbeitsweise vorausgesetzt.
In folgendem Beispiel wird die Programmierung eines U880-SIO und eines U880-CTC in einem U8001-Mikrorechnersystem dargestellt.
Der Kanal B des SIO wird für den Anschluß eines Displays mit einem seriellen Interface programmiert. Die Baudrate wird mit 9600 KBaud festgelegt.
Der Kanal 0 des CTC dient der Takterzeugung für den SIO. In diesem Beispiel entspricht die Zeitkonstante BAUD9600 für den CTC einer Baudrate von 9600 KBaud bei einem SIO-Vorteiler 32.
Der Kanal 1 des CTC wird als Zähler programmiert und löst nach fünf Impulsen einen Interrupt aus.

```
TTY_INIT procedure
entry
!CTC Initialisierung fuer die serielle Schnittstelle!
     LDB        RL0, BAUD9600        !Baudrate SIO Kanal B!
     LDB        RL1, #%57            !kein Int., Zaehler, Vorteiler 16,
                                     Zeitkonstante: BAUD 9600!
     OUTB       CTC0, RL1
     OUTB       CTC0, RL0

!SIO Initialisierung!
     LD         R2, #INIT_SIO_B
     LD         R0, #11              !Laenge der Initialisierungstabelle!
     LD         R1, SIOCON_B         !Controladresse des SIO Kanal B laden!
     OTIRB-     @R1, @R2, R0         !SIO ist nach der Ausgabe bereit!
     RET
     end TTY_INIT

global CLK_INIT procedure
!CTC Initialisierung fuer 5 Impulse!
entry
     LDB        RL0, #%08            !VI-Kennbyte!
     OUTB       CTC0, RL0            !VI-Kennbyte an Kanal 0 ausgeben!
```

```
     LDB       RL0, #11010111      !Int.enable/Zaehler!
     OUTB      CTC1, RL0           !Kanalsteuerwort an CTC-Kanal 1!
     LDB       RL0, #5             !Zaehler!
     OUTB      CTC1, RL0           !Zaehler an CTC-Kanal 1!
     RET
end CLK_INIT

global RETI_ procedure
entry
     LD   R1, #%ED4D               !U880 RETI-Befehlskode laden!
     OUTB RETI_PORT, RH1           !%ED ausgeben!
     OUTB RETI_PORT, RL1           !%4D ausgeben!
     RET
end RETI_

INTERNAL
INIT_SIO_B:
     ARRAY [* BYTE] := [COMM3
                        SIO_R4 + COMM2
                        S2 + C32 + NOPRTY
                        SIO_R3
                        RENABLE + B8
                        SIO_R5
                        XENABLE + T8 + DTR RTS
                        SIO_R1 + COMM2
                        PDAVCT + SAVECT
                        SIO_R2                    !Kennbyte (%10) zur!
                        %10]                      !Adressierung der ISR!
                                                  !in der Programmstatusarea!


!********************!
!SIO Programmierwerte!
!********************!
CONSTANT
BAUD9600   := %02
SIOCON_B   := %B023
SIO_R1     := %01
SIO_R2     := %02
SIO_R3     := %03
SIO_R4     := %04
SIO_R5     := %05
COMM2      := %10     !Reset ext/stat interrupt!
COMM3      := %18     !Channel reset!
C32        := %80     !x32 clock mode!
S2         := %0C     !2 Stop bits/character!
NOPRTY     := %00     !Disable parity!
RENABLE    := %01     !Receiver enable!
B8         := %C0     !Receive 8 bits/character!
XENABLE    := %08     !Transmitter enable!
T8         := %60     !Transmit 8 bits/character!
DTR        := %80     !Data terminal ready!
RTS        := %02     !Request to send!
PDAVCT     := %18     !Rx interrupt on all characters!
SAVECT     := %04     !Status affects vector!

!************!
!CTC Adressen!
!************!
CTC0       := %B011
CTC1       := %B013
RETI_PORT  := %B023
```

4. Aufbau und Funktion der Speicherverwaltungseinheit U8010-MMU

4.1. Überblick

Die steigende Leistungsfähigkeit der Mikrorechnersysteme führt zur Implementation immer komplizierterer Softwaresysteme, wie z.B. von Multi-User-Betriebssystemen auf 16-Bit-Mikrorechnern. Diese komplizierten Softwaresysteme wurden bisher nur auf Mini- oder Großrechnern implementiert. Sie erfordern zur Gewährleistung einer hohen Systemsicherheit und Zuverlässigkeit die Bereitstellung von hardwaregestützten Speicherschutzmechanismen. Mit ihrer Hilfe ist die gegenseitige Beeinflussung der Nutzer durch fehlerhafte Programme zu vermeiden. Programmfehler könnten sonst zu Fehlern in anderen Programmen bzw. zum Systemabsturz führen.
Die U8010-MMU (memory management unit) /2/, nachfolgend auch als MMU bezeichnet, bietet effektive Möglichkeiten zur Speicherverwaltung und zum Speicherschutz in U8001-Mikrorechnersystemen.

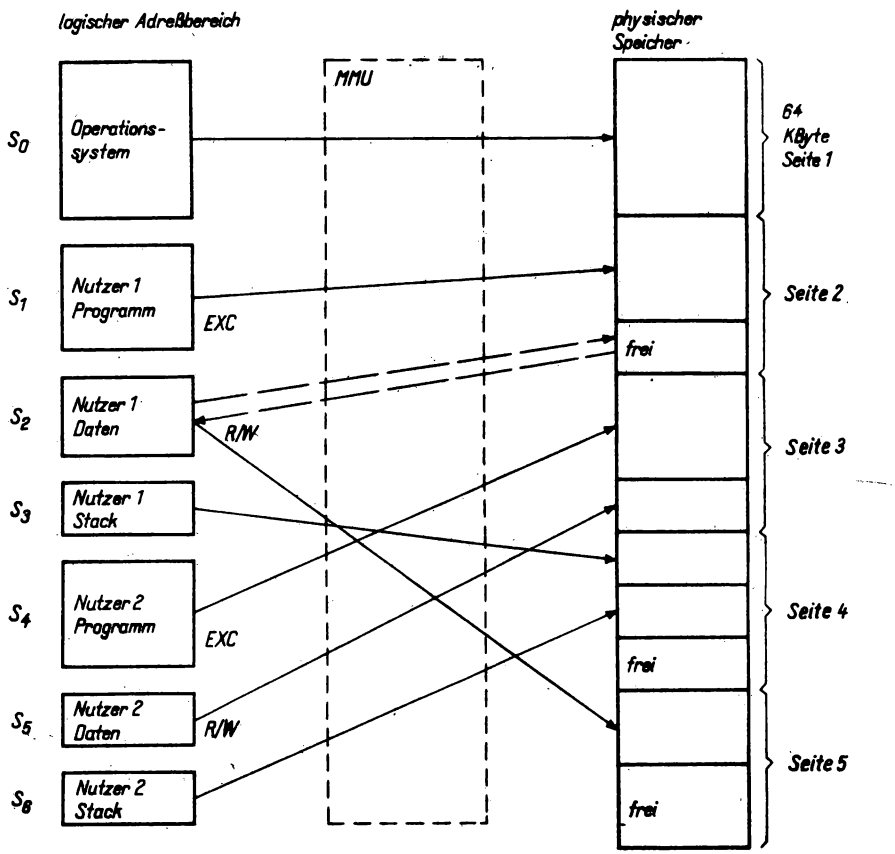


Bild 4.1. Dynamische Adreßverschiebung

Die U8010-MMU erlaubt die dynamische Verschiebung logischer Speichersegmente im gesamten physischen Speicherbereich und die variable Größendefinition von physischen Segmenten im Speicher. Im Bild 4.1 sind die damit verbundenen programmiertechnischen Möglichkeiten am Beispiel einer Multi-User-Anwendung dargestellt. Die physischen 64-KByte-Segmente werden als

Speicherseiten S1 bis S5 bezeichnet.
Im Beispiel liegen die Programme des Nutzers 1 auf der Seite 2, die des Nutzers 2 auf der Seite 3. Reicht der für die Daten des Nutzers 2 vorgesehene Speicherbereich nicht mehr, belegt das Operationssystem durch Umprogrammieren der MMU einen größeren Bereich auf einer anderen Speicherseite. Der Programmierer wird damit in die Lage versetzt mit logischen Adressen zu arbeiten und unabhängig von physischen Adressen zu bleiben.
Eine Überlappung der Segmente durch die mehrmalige Vergabe eines physischen Speicherbereichs für mehrere logische Segmente ist möglich. Das erlaubt es, Programmkode, der von mehreren Nutzern benötigt wird, nur einmal in den Speicher zu laden. Jeder Nutzer verfügt dann nur über einen eigenen Daten- und Stackbereich im physischen Speicher.
Die Programmierung der U8010-MMU ist eine Systemfunktion, sie erfolgt durch das Betriebssystem im Systemmode, d.h., der Nutzer eines Multi-User-Systems muß nicht wissen, in welchem Segment sein Programm abgearbeitet wird und wie groß dieses Segment ist.

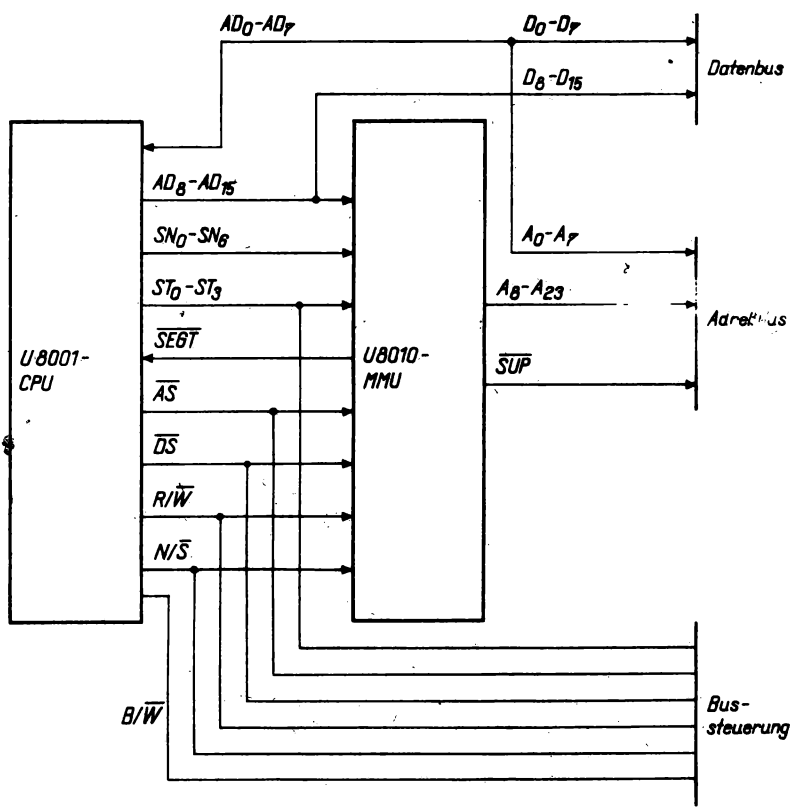


Bild 4.2. Einbindung der U8010-MMU in ein U8001-Mikrorechnersystem

Bei der Einbindung der MMU in ein U8001-Mikrorechnersystem werden die unteren 8 Bit der Adresse (AD_0 - AD_7) direkt von der CPU an den Speicher durchgeschaltet. Die oberen 8 Bit der Adresse und die Segmentnummer werden an die MMU übergeben und dort zur physischen Adresse verarbeitet.
Bei einer Verletzung der Zugriffsbedingungen wird die CPU in ihrer Arbeit

durch einen Segmenttrap unterbrochen. In der Segmenttraproutine kann der Status der MMU rückgelesen werden, aus diesen Informationen kann die Fehlerursache ermittelt werden.

4.2. Hardwareorientierte Funktionen

Die U8010-MMU ist ein 48poliger Schaltkreis im DIL-Gehäuse (dual in line). Sie benötigt neben den Bussignalen, wie die U8000-CPU, eine +5 V Versorgungsspannung und einen synchronen Einphasentakt im Bereich von 500 kHz bis 4 MHz. Die Standardversion der U8010-MMU ist in einem Temperaturbereich von 0...70 °C einsetzbar.

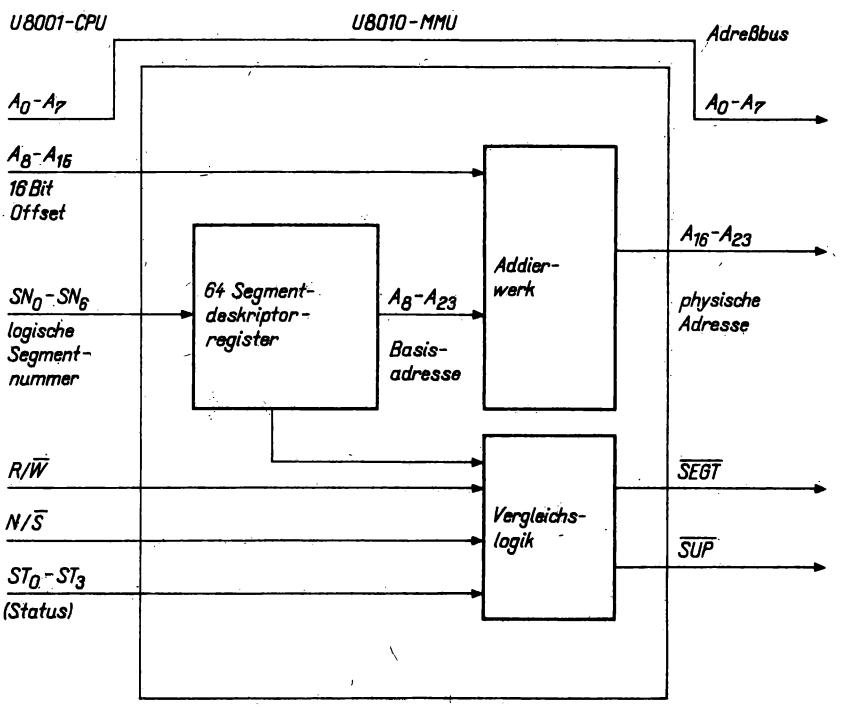


Bild 4.3. U8010-MMU, Blockstruktur

Die MMU enthält 64 Segmentdeskriptorregister, in denen alle für die Realisierung der Funktionen erforderlichen Informationen stehen. Mit einem Schaltkreis können demzufolge 64 Speichersegmente mit einem Speichervolumen von minimal 256 Byte bis maximal 64 KByte verwaltet werden. Dabei muß die Segmentgrenze stets mit einem 256-Byte-Block abschließen.
Mit zwei MMU-Schaltkreisen ist somit der im U8000-Mikrorechnersystem adressierbare Speicher von 128 * 64 KByte zu verwalten.

4.2.1. Signal- und Pinbeschreibung

Das Signalspiel ist mit dem der U8001-CPU identisch, sie arbeitet ebenfalls mit dem Systemtakt. Die grundsätzlichen CPU-Zyklen gelten auch bei der U8010-MMU. Die Segmentnummer wird stets vor dem ersten Taktzyklus ausgegeben, da die MMU intern Zeit benötigt, um das Segmentdeskriptorregister auszuwählen und die erforderlichen arithmetischen Operationen auszuführen.

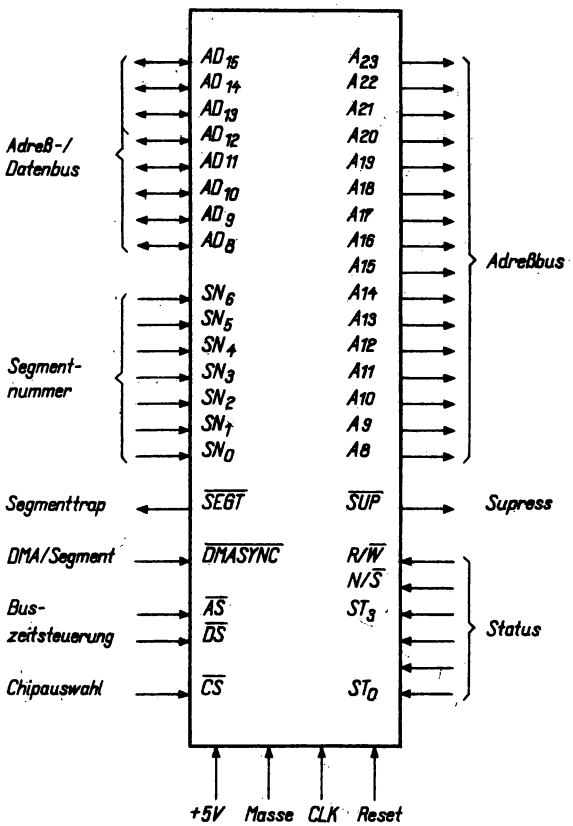


Bild 4.4. U8010-MMU, Pinbeschreibung

Die 48polige MMU ist im Bild 4.4 mit funktionell geordneten Pins dargestellt. Für die bereits bei der Erläuterung der U8000-CPU im Abschnitt 2.2.1. erklärten Signale ist hier nur eine Kurzfassung der Signalbeschreibung angegeben.

Adreß-/Datenbus - Die Pins AD_8 - AD_{15} entsprechen den gleichnamigen Pins des Adreß-/Datenbusses der CPU. Sie dienen der Übergabe der oberen 8 Bit des Offset und der Datenübergabe bei der Programmierung der MMU.
Segmentnummer. Auf den Pins SN_0 - SN_6 wird die Segmentnummer von der U8001-CPU an die U8010-MMU übergeben.

Segmenttrap - Das Pin SEGT dient der Auslösung des Segmenttraps in der CPU bei Feststellung einer Verletzung der Zugriffsbedingungen durch die MMU.
DMASYNC. Über das Pin DMASYNC wird der MMU der Zugriff eines DMA-Bausteines (DMA: direct memory access) auf den Speicher angezeigt.
Buszeitsteuerung - Die Pins AS (address strobe) und DS (data strobe) zeigen den Beginn einer Busaktivität und, wegen der wechselnden Belegung des Adreß-/Datenbusses, die jeweilige Belegung des Busses an.
Chipauswahl - Das Pin CS (chip select) dient der Schaltkreisauswahl während der Programmierung bzw. beim Rücklesen von Registerinhalten.
Adreßbus - Über die Pins A_8 - A_{23} werden die 16 höherwertigen Adreßbits der physischen 24-Bit-Adresse ausgegeben.
Supress - Bei aufgetretenen Fehlern dient das Pin SUP der Ausgabe eines Signals zur Unterdrückung von Schreiboperationen während eines Buszyklus.
Status - Neben den Statussignalen ST_0 - ST_3 wird das Signal R/W und das Signal N/S ausgewertet. Das Signal R/W gibt an, ob es sich um eine Lese- oder Schreiboperation der CPU handelt. Das Signal N/S zeigt an, ob sich die CPU im Normal- oder Systemmode befindet.
MMU-Steuerung - Auf das CLK-Pin wird der CPU-Systemtakt gelegt. Mit der Aktivierung des RESET-Pins wird die MMU in einen inaktiven Zustand gebracht, die internen Register werden gelöscht.

4.2.2. Schreib-/Lesezyklen

Zum Verständnis der Arbeitsweise des Schaltkreises ist die Darstellung des Zeitablaufs bei Speicheroperationen erforderlich. Die MMU benötigt für die internen arithmetischen Operationen eine gewisse Zeit. Aus diesem

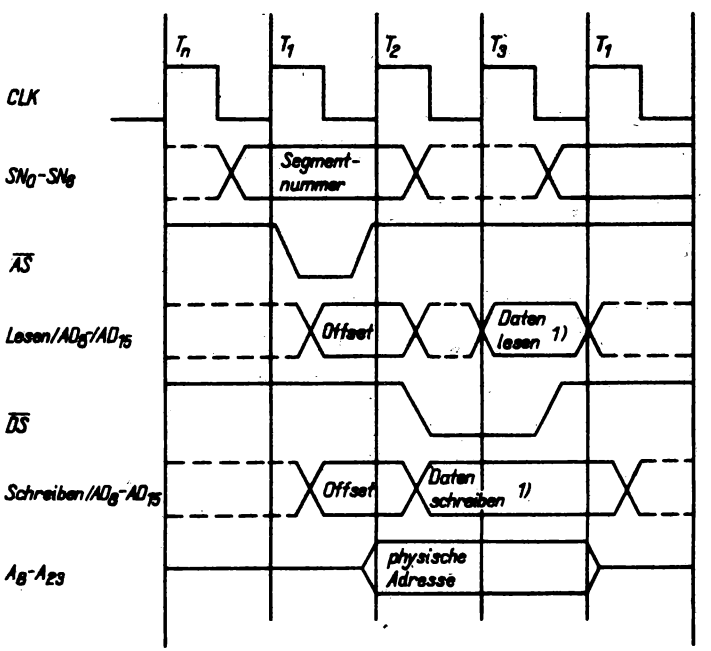


Bild 4.5. Lese-/Schreibzyklen bei Nutzung der U8010-MMU
1) Die CPU-Operationen sind zur besseren Darstellung der zeitlichen Abläufe mit abgebildet.

Grund wird die Segmentnummer von der U8001-CPU zum frühestmöglichen Zeitpunkt ausgegeben. Sie liegt bereits vor T_1 an, damit kann vor der Ausgabe des Offset schon die Auswahl des Deskriptorregisters in der MMU erfolgen. Der Offset wird zu T_1 von der U8001-CPU ausgegeben. Mit AS (address strobe) aktiv (low) wird der Offset (AD_8 - AD_{15} von der MMU übernommen. Die physische Speicheradresse wird aus dem Offset und der durch die Segmentnummer im Deskriptorregister ausgewählten Basisadresse errechnet und von T_2 bis T_3 an den Speicher ausgegeben. Während T_3 erfolgt dann mit DS (data strobe) aktiv (low) der Lese- oder Schreibzugriff auf den Speicher.

4.3. Softwareorientierte Funktionen

4.3.1. U8010-MMU Register

Die MMU benötigt zur Ausführung der Vergleichsoperationen interne Register. In ihnen werden die zur Programmierung der MMU erforderlichen Parameter eingetragen. Bei der Verletzung von Zugriffsbedingungen enthalten sie Informationen über Fehler bzw. Warnungen. Sie sind der jeweiligen Funktion entsprechend als 8- oder 32-Bit-Register aufgebaut. Nach ihrer Funktion lassen sie sich in drei Gruppen einteilen:

Segmentdeskriptorregister
Steuerregister
Statusregister.

Zur Programmierung der MMU dienen die Spezial-Ein-/Ausgabebefehle, sie können nur im Systemmode der U8001-CPU ausgeführt werden.

4.3.1.1. Segmentdeskriptorregister

Die Segmentdeskriptorregister werden durch die Bits SN_0 - SN_5 adressiert. In jedem Schaltkreis sind 64 Segmentdeskriptorregister enthalten. Die U8001-CPU kann 128 logische Segmente adressieren. Deshalb muß mit der Programmierung der MMU der Bereich, in dem die Segmentnummern (0 - 63 oder 64 - 127) übersetzt werden sollen, angegeben werden. Mit dem Bit SN_6 der Segmentnummer erfolgt dann die Aktivierung der entsprechenden MMU im

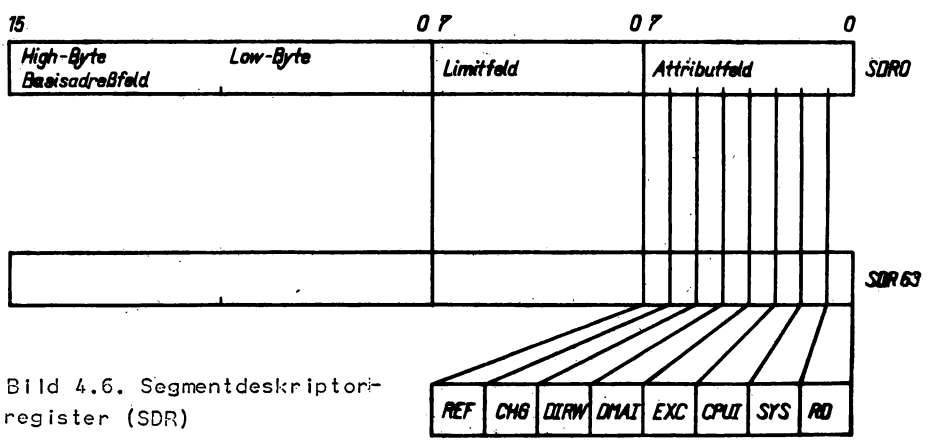


Bild 4.6. Segmentdeskriptorregister (SDR)

System. Das Bit SN_6 muß mit einem programmierbaren Flag der MMU, dem URS-Flag (s. Abschnitt 4.3.1.2.) übereinstimmen, es legt fest, ob sie im oberen oder im unteren Bereich der Segmentnummern übersetzen soll.
Ist $SN_6 = 0$, werden die logischen Segmentadressen 0 - 63 übersetzt; ist $SN_6=1$, werden die logischen Segmentadressen 64 - 127 übersetzt.
Die Segmentdeskriptorregister sind 32 Bit breit.

Basisadreßfeld
Im 16-Bit-Basisadreßfeld des Segmentdeskriptorregisters ist die Basisadresse des Segments gespeichert.

Limitfeld
Das zweite Feld im Segmentdeskriptorregister ist das 8-Bit-Limitfeld, es enthält die Länge des Segments in 256-Byte-Blöcken (siehe DIRW).

Attributfeld
Das dritte Feld des Segmentdeskriptorregisters ist das 8-Bit-Attributfeld, es enthält acht Attributflags für das jeweilige Segment. Von den acht Flags des Attributfeldes dienen sechs (RD, SYS, CPUI, EXC, DMAI, DIRW) zur Programmierung der Attribute, zwei (CHG, REF) zur Ausgabe von Informationen der MMU an die CPU. Bei Bedarf kann damit ein Zugriff auf das Segment nachgewiesen werden.

RD-Flag (read-only) - Das Setzen des RD-Flags unterbindet Schreibzugriffe, d.h., in diesem Segment darf nur gelesen werden.
SYS (system-only) - Ist das SYS-Flag gesetzt werden im Segment nur Zugriffe im Systemmode zugelassen, d.h., nicht nur einzelne Befehle, sondern auch komplette Programme können nur im Systemmode abgearbeitet werden.
CPUI (CPU inhibit) - Nach dem Setzen des CPUI-Flags kann die CPU nicht mehr auf das Segment zugreifen. Nur DMA Zugriffe sind möglich.
EXC (execute-only) - Das Setzen des EXC-Flags bewirkt, daß nur Programme abgearbeitet werden können. Es ist kein Lese- oder Schreibzugriff auf Daten möglich.
DMAI (DMA inhibit) - Es ist kein Zugriff durch einen DMA-Schaltkreis bei gesetztem DMAI-Flag möglich. Nur die U8001-CPU kann auf den Speicher zugreifen.
DIRW (direction and warning) - Ist das DIRW-Flag gesetzt, darf das Segment nur in Richtung fallender Adressen beschrieben werden, es wird als Stacksegment genutzt.
Sein Zustand hat Auswirkungen auf die Programmierung des Limitfeldes. Die Anzahl der belegten Blöcke ergibt sich nach folgender Regel:

DIRW = 0	N + 1	Blöcke
DIRW = 1	256 - N	Blöcke.

Mit N ist die in das Limitfeld eingetragene Größe gegeben.
Neben der Direction-Funktion erfolgt noch eine Warnung. Wird in den untersten 256-Byte-Block des Segments geschrieben, erfolgt eine Warnung, um Stacküberläufe zu vermeiden.

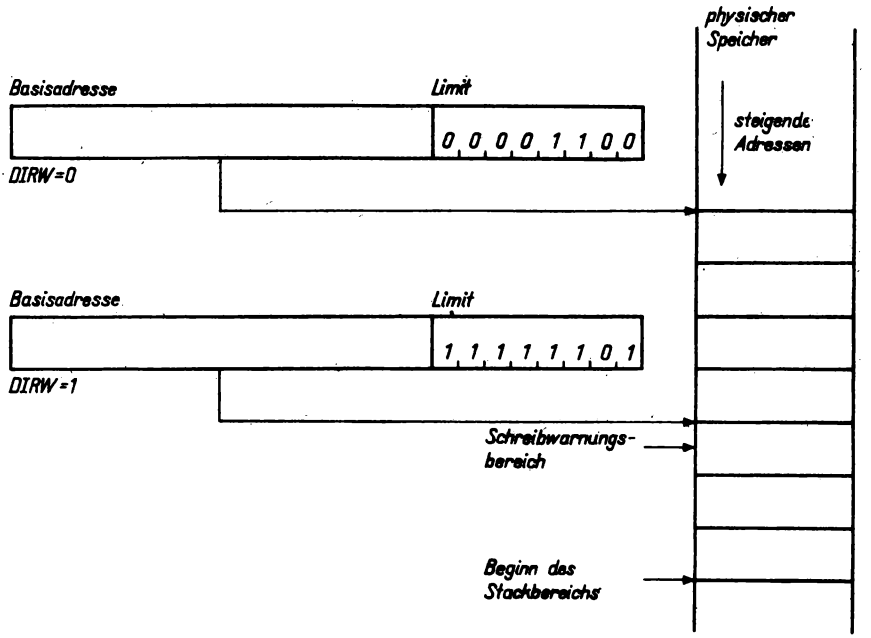


Bild 4.7. Wirkung des DIRW-Flags

CHG (changed) - Das CHG-Flag wird von der MMU gesetzt, wenn ein Schreibzugriff auf dieses Segment erfolgte. Es kann zu Kontrollzwecken genutzt werden.
REF (referenced) - Das REF-Flag wird von der MMU gesetzt, wenn ein Zugriff auf das Segment erfolgte. Es kann zu Kontrollzwecken genutzt werden.

4.3.1.2. Steuerregister

Zur Programmierung der 64 Deskriptorregister werden drei 8-Bit-Steuerregister benötigt.

Segmentadreßregister (SAR)
Das Segmentadreßregister dient zur Adressierung der 64 Segmentdeskriptorregister beim Programmieren oder Rücklesen der MMU. Das SAR besitzt Autoinkrement Fähigkeiten und ermöglicht damit die Anwendung von blockweise wirkenden Spezial-Ein-/Ausgabebefehlen.

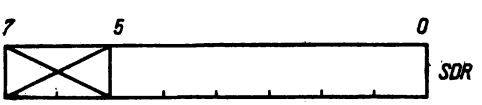


Bild 4.8. Segmentadreßregister (SAR)

Mit den blockweise wirkenden Spezial-Ein-/Ausgabebefehlen ist die kom-

plette Programmierung der 64 Segmentdeskriptorregister durch einen Datentransfer möglich.

Deskriptorselektionszähler (DSC)
Die vier Byte im SDR werden durch den Deskriptorselektionszähler adressiert. Er belegt zwei Bit eines 8-Bit-Registers. Der DSC adressiert die vier Byte des SDR nach folgendem Schema:

00 High-Byte der Basisadresse
01 Low-Byte der Basisadresse
10 Limitbyte
11 Attributbyte.

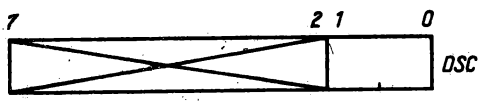


Bild 4.9. Deskriptorselektionszähler

Der Zähler wird automatisch inkrementiert; er steht nach der Programmierung eines SDR wieder auf 0.

Moderegister
In diesem Register wird mit fünf Flags die Betriebsart und mit drei weiteren Flags die Adresse der U8010-MMU in einem U8001-Mikrorechnersystem festgelegt.

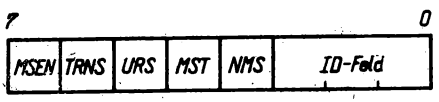


Bild 4.10. Moderegister

ID (identification) - Das ID-Feld wird bei einem Segmenttrap zur Ausgabe des Kennwortes (identifier) benutzt. Der Inhalt des ID-Felds wird dabei im 1-aus-8- Kode in eine Bitposition im High-Byte des Kennwortes (AD_8 - AD_{15}) kodiert. Das Low-Byte des Kennwortes enthält einen undefinierten Wert.

ID-Feld	Kennwort
000	00000001XXXXXXXX
001	00000010XXXXXXXX
010	00000100XXXXXXXX
011	00001000XXXXXXXX
100	00010000XXXXXXXX
101	00100000XXXXXXXX
110	01000000XXXXXXXX
111	10000000XXXXXXXX

NMS (normal mode select) - Das NMS-Flag ist in Verbindung mit dem MST-Flag zu programmieren. Das MST-Flag wird gesetzt, wenn mehrere MMU-Schaltkreise im System enthalten sind. Ist es gesetzt, wird über das NMS-Flag entschieden, ob die jeweilige MMU im System- oder Nomalmode übersetzen soll.

NMS=1 Übersetzung im Normalmode
NMS=0 Übersetzung im Systemmode

Wenn das MST-Flag nicht gesetzt ist, wird das NMS-Flag nicht abgefragt.

MST (multi segment table) - Das MST-Flag muß gesetzt werden, wenn mehr als eine U8010-MMU im Mikrorechnersystem enthalten ist und eine Aufteilung der Schaltkreise für den System- bzw. Normalmode vorgenommen wurde. Das NMS-Flag entscheidet über die Auswahl der MMU, entweder für den System- oder den Normalmode.

URS (upper range select) - Mit dem URS-Flag wird festgelegt, in welchem Bereich der logischen Adressen die MMU übersetzen soll.

URS=0 Segmentnummern 0 - 63
URS=1 Segmentnummern 64 - 127

TRNS (translate) - Ist das TRNS-Flag gesetzt, übersetzt die MMU logische in physische Adressen, ansonsten werden von der MMU die logischen als physische Adressen durchgeschaltet. Dabei erfolgt weder eine Übersetzung noch eine Prüfung der Adressen.

TRNS=1 Übersetzung erfolgt.
TRNS=0 Ungeprüftes Passieren der Adresse.

MSEN (master enable) - Mit dem Setzen des MSEN-Flags wird die MMU aktiviert. Das MSEN-Flag muß zur Sicherung der ordnungsgemäßen Funktion der MMU zusammen mit dem TRNS-Flag gesetzt werden. Wird das MSEN-Flag rückgesetzt, kann die CPU über die MMU nicht auf den Speicher zugreifen, sie ist hochohmig. Die CPU kann in diesem Fall nur arbeiten, wenn sie einen speziellen Speicherbereich besitzt, auf den sie ohne Einbeziehung der MMU zugreifen kann.

4.3.1.3. Statusregister

Die sechs 8-Bit-Statusregister dienen zur Auswertung der Fehlerinformation nach einem Segmenttrap. Sie ermöglichen eine genaue Feststellung der Fehlerursache. Die Statusregister werden durch den ersten auftretenden Fehler gesetzt und bleiben bis auf das SWW- und/oder das FATL-Flag des Violation-Type-Register durch nachfolgende Fehler unbeeinflußt.

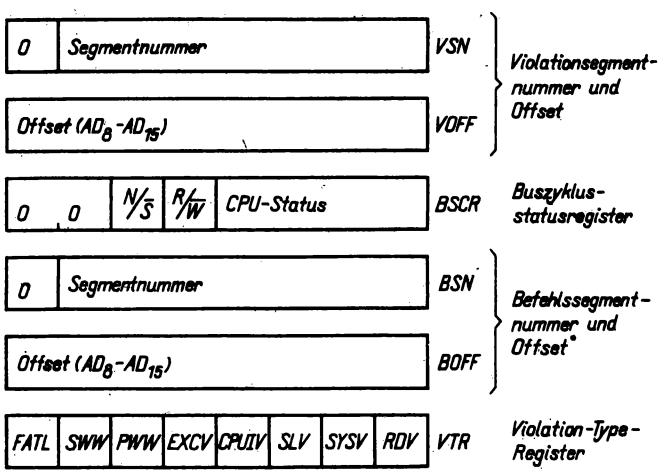


Bild 4.11. U8010-MMU Statusregister

Violationsegmentnummer (VSN) und -offset (VOFF)
In diesen Registern wird die logische Adresse, die zum Segmenttrap führte, gespeichert. Das Low-Byte des Offset wird in der MMU nicht gespeichert.

Buszyklusstatusregister
Der CPU-Status, das Signal N/S (normal/system) und das Signal R/W (read-/write) werden beim Auftreten eines Segmenttrap in dieses Register gespeichert.

Befehlssegmentnummer (BSN) und -offset (BOFF)
In diesen Registern wird die logische Adresse des Befehls vor dem Auftreten des Segmenttraps gespeichert. Dies ist, insbesondere bei Befehlen zur Programmverzweigung, erforderlich. Eine Information zu einem Fehler ist besser auswertbar, wenn der vorher abgearbeitete Befehl bekannt ist.

Violation-Type-Register (VTR)
In den acht Flags des VTR wird die Ursache des Segmenttraps angegeben. Fünf Flags (RDV, SYSV, SLV, CPUI, EXC) werden bei Zugriffsfehlern gesetzt, zwei (PWW, SWW) bei Warnungen bzw. Folgewarnungen, und ein Flag (FATL) zeigt Folgefehler an.

RDV (read-only violation). Das RDV-Flag wird bei einem Schreibversuch auf ein Read-Only-Segment gesetzt.
SYSV (system violation). Das SYSV-Flag zeigt den Versuch an, auf ein Systemsegment im Normalmode zu schreiben.
SLV (segment length violation). Bei Überschreitung der vorgegebenen Länge des Segmentes wird das SLV-Flag gesetzt.
CPUIV (CPU inhibit violation). Wenn die CPU auf ein Segment zugreifen will, das für den Transfer mit einem DMA-Baustein programmiert wurde, wird das CPUIV-Flag gesetzt.
EXCV (execute-only violation). Das EXCV-Flag zeigt an, daß die CPU auf ein Executesegment zugreifen wollte, ohne einen Befehlslesezyklus zu generieren.
PWW (primary write warning). Das PWW-Flag wird als Warnung gesetzt, wenn das DIRW-Flag gesetzt ist und die CPU bei einer Stackoperation in den letzten 256-Byte-Block des Stackbereichs schreibt.
SWW (second write warning). Mit dem SWW-Flag zeigt die MMU an, daß die CPU im Systemmode Daten in die Schreibwarnzone des Systemstackbereiches geschrieben hat und das EXCV-, CPUIV-, SLV-, SYSV-, RDV- oder PWW-Flag gesetzt war.
Nach dem Setzen des SWW-Flags verursachen nachfolgende Warnungen aufgrund der möglichen Überschreitung des Stackbereichs im Systemmode keinen neuen Segmenttrap mehr.
FATL (fatal condition). Das FATL-Flag wird gesetzt, wenn ein Fehler auftritt und irgendein anderes Flag bereits gesetzt war. Das FATL-Flag zeigt einen kritischen Systemzustand an, im allgemeinen weist es auf eine Zugriffsverletzung in der Segmenttraproutine hin. Die zum Setzen des FATL-Flags führende Ursache wird nicht angezeigt. Die ursprünglichen Fehlerinformationen bleiben in den Statusregistern erhalten.
Nach dem Setzen des FATL-Flags wird durch ggf. weitere Fehler kein Segmenttrap mehr ausgelöst, das Supress-Signal wird durch die MMU aktiviert. Es verhindert bis zum Rücksetzen des FATL-Flags Schreibvorgänge auf dem Speicher.

4.3.2. Adreßberechnung

Die U8010-MMU übersetzt eine logische 23-Bit-Adresse von der U8001-CPU in eine physische 24-Bit-Speicheradresse.

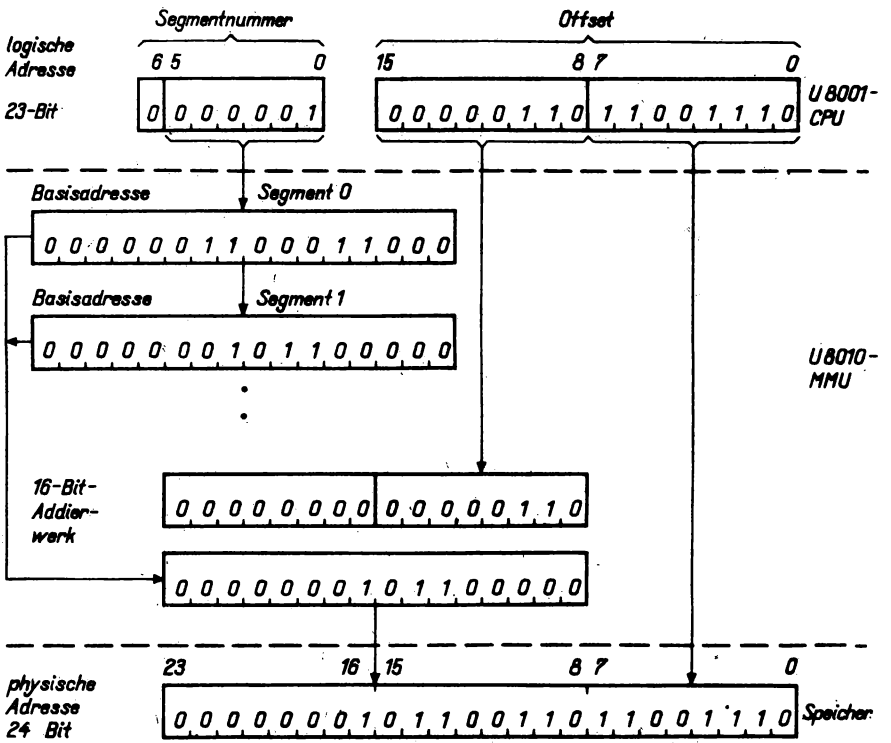


Bild 4.12. Übersetzung der logischen Adresse in eine physische Adresse

Die Umwandlung der logischen Adresse in eine physische Adresse erfolgt im wesentlichen in den nachfolgend aufgeführten Schritten:

- Die Segmentnummer SN_0 - SN_5 adressiert eines der 64 Segmentdeskriptorregister. Das Bit SN_6 der Segmentnummer dient in Verbindung mit dem URS-Flag zur Festlegung des Arbeitsbereichs der MMU. Im Segmentdeskriptorregister steht die Basisadresse des Segmentes (AD_8 - AD_{23}).
- Der Offset (AD_8 - AD_{15}) wird zur Basisadresse (AD_8 - AD_{23}) addiert. Zusammen mit den unverändert übernommenen Adreßbits (AD_0 - AD_7 wird damit eine physische 24-Bit-Adresse gebildet. Diese Adresse wird stets, d.h. auch bei der Feststellung von Fehlern, ausgegeben.
- Parallel zur Berechnung der physischen 24-Bit-Adresse erfolgt der Vergleich des Attributfelds im Segmentdeskriptorregister mit dem Status, dem R/W- und N/S Pin der CPU. Eine Verletzung der Zugriffsbedingungen führt zur Generierung eines Segmenttraps und ggf. zur Aktivierung des Supress-Signals.
- Abschließend wird der Offset mit dem Inhalt des Limitfelds aus dem Segment Deskriptorregister verglichen, bei Überschreitung der zulässigen Segmentlänge wird ein Segmenttrap und das Supress-Signal generiert.

Erfolgt der Zugriff in die unteren 256 Byte eines Segmentes dessen DIRW-Flag gesetzt ist, wird nur ein Segmenttrap generiert.

4.3.3. U8010-MMU Betriebszustände

Die MMU kann während ihres Betriebes fünf Zustände einnehmen. Den Ausgangszustand (Homezustand) und vier Zustände, die beim Auftreten von Fehlern oder Warnungen eingenommen werden. Die Einnahme dieser internen MMU-Betriebszustände, im weiteren als MMU-Zustände bezeichnet, hängt von den jeweiligen Fehlern oder Warnungen, der Reihenfolge ihres Auftretens und den Maßnahmen zu ihrer Beseitigung ab. Der Wechsel von einem Zustand in den anderen erfolgt während der Abarbeitung eines CPU-Befehls. Für den Übergang zwischen den Betriebszuständen lassen sich die Flags im Violation-Type-Register (VTR) der MMU (Abschnitt 4.3.1.) in drei Gruppen einteilen.

- RDR, SYSV, SLV, CPUIV, EXCV, PWW
  (nachfolgend als Primärflags "PRMFLG" bezeichnet)
- SWW
- FATL

Die erste Gruppe enthält die primär bei Fehlern oder Warnungen auftretenden Flags. Diese Flags sind in ihrer Bedeutung untereinander gleichwertig.

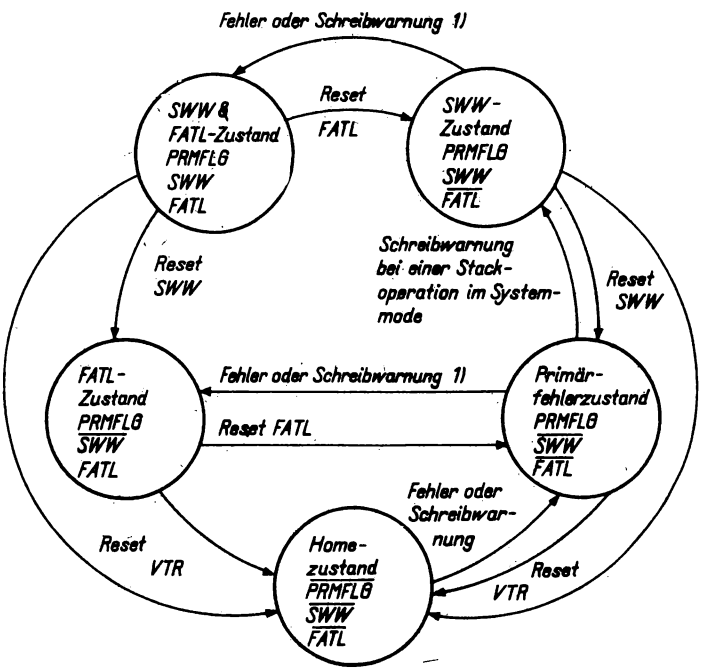


Bild 4.13. U8010-MMU, Zustandsgraf

Homezustand
Der Homezustand ist der Ausgangszustand der MMU, er wird nach jedem Hardware-Resetsignal, d.h. auch nach einem Neustart eingenommen.
Er bleibt bis zum Auftreten eines Fehlers oder einer Warnung bestehen. Wird ein Flag im Violation-Type-Register (VTR) gesetzt, verläßt die MMU den Homezustand und geht in den Primärfehlerzustand über. Der Homezustand kann durch Rücksetzen des VTR - d.h. einem Software-Reset - aus jedem der vier durch Fehler oder Warnungen bedingten Zustände eingenommen werden.

Primärfehlerzustand (single violation state)
Der Primärfehlerzustand zeigt einen Fehler oder eine Schreibwarnung an. Ein Flag der PRMFLG-Gruppe ist gesetzt und verursacht einen Segmenttrap. Dieser Zustand kann auf drei Wegen eingenommen werden:

- aus dem Homezustand, wenn ein Fehler oder eine Schreibwarnung auftritt,
- aus dem SWW-Zustand, wenn das SWW-Flag rückgesetzt wurde,
- aus dem FATL-Zustand, wenn das FATL-Flag rückgesetzt wurde.

Treten weitere Fehler oder Schreibwarnungen auf bzw. wird das Violation-Type-Register durch Softwareoperationen beeinflußt, kann der Primärfehlerzustand in drei andere Zustände übergeführt werden:

- in den Homezustand, wenn das VTR rückgesetzt wurde,
- in den SWW-Zustand, wenn eine weitere Schreibwarnung durch eine Stackoperation im Systemmode ausgelöst wurde,
- in den FATL-Zustand, wenn weitere Fehler auftreten und Warnungen, die nicht auf eine Stackoperation im Systemmode zurückgehen.

FATL-Zustand
Der FATL-Zustand zeigt einen kritischen Systemzustand an. Die Ursache kann in einer Verletzung der Zugriffsbedingungen in der Traproutine liegen. Das FATL-Flag wird auch gesetzt, wenn die Rückkehr nach der Traproutine in den Normalmode erfolgt, ohne daß ein Rücksetzen des VTR erfolgte. Das Setzen des FATL-Flags ruft einen neuen Segmenttrap hervor, der zum wiederholten Aufruf der Segmenttraproutinen führt. Nachfolgend auftretende Fehler oder Warnungen werden bis zum Rücksetzen des FATL-Flags unterdrückt.
Die Flags der PRMFLG-Gruppe werden beim Übergang in den FATL-Zustand nicht geändert. In den Statusregistern bleibt die erste Fehlerursache gespeichert, zusätzlich wird das FATL-Flag gesetzt.
Der FATL-Zustand kann auf zwei Wegen eingenommen werden:

- aus dem Primärfehlerzustand, wenn mindestens ein zweites Flag aus der PRMFLG-Gruppe gesetzt wird,
- aus dem SWW&FATL-Zustand, wenn das SWW-Flag rückgesetzt wird.

Aus dem FATL-Zustand kann der Übergang in die folgenden zwei Zustände nur durch Softwareoperationen, die das VTR beeinflussen, erfolgen:

- in den Homezustand, wenn das VTR rückgesetzt wurde,
- in den Primärfehlerzustand, wenn das FATL-Flag rückgesetzt wurde.

SWW-Zustand (second write warning state)
Der Übergang aus dem Primärfehlerzustand in den SWW-Zustand wird durch eine zweite Zugriffsverletzung, die wiederum zu einem Segmenttrap führt,

verursacht, er führt damit auch zu einem zweimaligen Aufruf der Segmenttrapserviceroutine. Stackoperationen im Systemmode können im SWW-Zustand keinen Segmenttrap mehr erzeugen. Zugriffsverletzungen durch Fehler in nachfolgenden Speicheroperationen können einen dritten Segmenttrap auslösen.
Der SWW-Zustand kann auf zwei Wegen eingenommen werden:

- aus dem Primärfehlerzustand, wenn eine Schreibwarnung durch eine Systemstackoperation auftritt. Ein derartiger Fehler tritt im wesentlichen in der Traproutine selbst auf.
- aus dem FATL&SWW-Zustand, wenn das SWW-Flag zurückgesetzt wird.

Treten weitere Fehler oder Schreibwarnungen auf bzw. erfolgen Softwareoperationen, die das VTR beeinflussen, kann der SWW-Zustand in die drei folgenden Zustände übergeführt werden:

- in den Homezustand, wenn das VTR rückgesetzt wurde,
- in den Primärfehlerzustand wenn das SWW-Flag rückgesetzt wurde,
- in den SWW&FATL-Zustand, wenn die MMU eine weitere Zugriffsverletzung durch einen Fehler oder eine Warnung, die nicht auf einer Stackoperation im Systemmode beruht, signalisiert.

FATL&SWW-Zustand
Dieser Zustand weist darauf hin, daß die MMU eine Zugriffsverletzung in der Segmenttrapserviceroutine festgestellt hat. Sie generiert einen dritten Segmenttrap, der wiederum zum Aufruf der Segmenttraproutine führt. Weitere Zugriffsverletzungen lösen keinen Segmenttrap mehr aus, das Supress-Signal wird aktiviert.
Der FATL&SWW-Zustand kann nur aus dem SWW-Zustand eingenommen werden. Dazu ist eine weitere Zugriffsverletzung durch einen Fehler oder eine Warnung, die nicht auf einer Stackoperation im Systemmode beruht, erforderlich.
Aus diesem Zustand kann durch Softwareoperationen der Übergang in drei Zustände erfolgen:

- in den FATL-Zustand, wenn das FATL-Flag rückgesetzt wurde,
- in den SWW-Zustand, wenn das SWW-Flag rückgesetzt wurde,
- in den Homezustand, wenn das VTR rückgesetzt wurde.

4.3.4. Fehler und Warnungen

Die MMU generiert einen Segmenttrap beim Auftreten von Fehlern (violation condition) und Schreibwarnungen (write warning condition).

Fehler
Die Fehler zeigen eine Verletzung der Zugriffsbedingungen an, sie werden gemäß Abschnitt 4.3.1. durch die folgenden Flags im Violation-Type-Register (VTR) angezeigt:

RDV (read-only violation)
SYSV (system-only violation)
CPUIV (CPU inhibit violation)
EXCV (execute-only violation)
SLV (segment length violation).

Diese Flags zeigen primäre Fehler an, sie werden unter der Bezeichnung PRMFLG-Gruppe zusamengefaßt. Sekundäre Fehler führen zum Setzen des FATL-Flags. Das FATL-Flag wird erst gesetzt, wenn nach dem ersten Fehler in einer folgenden Speicheroperation ein neuer Fehler auftritt. Die Ursache des neuen Fehlers wird, bis auf den durch das Setzen des FATL-Flags ausgelösten Segmenttrap, nicht angezeigt.
Treten bei einem Speicherzugriff mehrere Fehler gleichzeitig auf, werden auch die entsprechenden Flags der PRMFLG-Gruppe gleichzeitig gesetzt, und ein Segmenttrap wird ausgelöst.
Tritt in einer U8001-Konfiguration mit MMU und DMA ein Fehler auf, so erkennt ihn der DMA-Schaltkreis. Die CPU erhält keine Fehlermeldung durch die MMU, sondern durch den DMA-Schaltkreis, indem das Statusbyte des DMA-Schaltkreises ausgewertet wird.

Warnungen
Eine Warnung signalisiert eine mögliche Stacküberschreitung. Ein Segmenttrap wird durch eine Warnung generiert, wenn in die unteren 256 Byte eines Segmentes geschrieben wird, dessen DIRW-Flag gesetzt ist. Primäre Warnungen führen zum Setzen des PWW-Flags, bei sekundären Warnungen setzt die MMU das SWW-Flag.
Warnungen beeinflussen die folgenden drei Flags im VTR:

PWW (primary write warning)
SWW (secondary write warning)
FATL (fatal condition).

Das FATL- und SWW-Flag werden nicht gleichzeitig gesetzt, sie können nur durch nachfolgend auftretende Verletzung der Zugriffsbedingungen gesetzt werden. Im Bild 4.13 sind die möglichen Varianten dargestellt. Das FATL-Flag kann nur durch eine Schreibwarnung im Normalmode gesetzt werden.

Supress
Das Supress-Signal dient nach der Verletzung von Zugriffsbedingungen der Verhinderung von Schreiboperationen im Speicheradreßraum. Es wird nach der Feststellung von Fehlern zusammen mit dem Segmenttrap generiert. Bei Schreibwarnungen wird nur ein Segmenttrap ausgelöst. Das Supress-Signal würde die Eintragung der Rückkehradresse in den Stackbereich unterbinden und damit zum Systemabsturz führen.
Tritt ein Fehler bei einem Datentransfer durch einen DMA-Schaltkreis auf, gibt die MMU nur das Supress-Signal aus, sie generiert keinen Segmenttrap. Der Fehler kann dem Statusbyte des DMA-Schaltkreises entnommen werden.

5. Programmierung der Speicherverwaltungseinheit U8010-MMU

Die Programmierung der U8010-MMU ist eine wesentliche Systemfunktion. Dies bezieht sich vor allem auf die Ermittlung von Fehlerursachen durch Auswertung des Violation-Type-Registers (VTR) und die Einleitung von Maßnahmen zu ihrer Beseitigung. Diese Maßnahmen hängen von der Fehlerursache und den Systemressourcen ab. Sie können sowohl eine Fehlermeldung und den Abbruch des Programms als auch die dynamische Korrektur der Speicherbelegung beinhalten.
Die Programmierung der U8010-MMU erfolgt mit den Spezial-Ein-/Ausgabebefehlen im Systemmode der U8001-CPU. Sie übermitteln der U8010-MMU Kommandos. Sie dienen der Modifizierung bzw. Ausgabe von Registerinhalten sowie zur Steuerung der MMU. Die Spezial-Eingabebefehle dienen zum Lesen, die Spezial-Ausgabebefehle zum Schreiben der MMU-Register.
Die Adressierung der MMU-Schaltkreise kann nur durch gerade Adressen erfolgen, die Adresse wird im 1 aus 8 Kode angegeben. Als erste Adresse für eine MMU in einem Mikrorechnersystem kann %02 angegeben werden. Die MMU-Adresse wird im Low-Byte der Adresse von der CPU ausgegeben, im High-Byte steht der Befehlskode für die MMU.

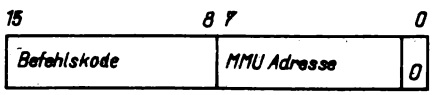


Bild 5.1. Befehlsformat der U8010-MMU

Die MMU-Befehle lassen sich in zwei Gruppen, Schreib-/Lesebefehle und Set-/Resetbefehle, einteilen. Die Lese- oder Schreibdaten werden auf den höherwertigen acht Datenbits (AD_8 - AD_{15}) übertragen.. Bei Wort-Lesebefehlen wird mit den niederwertigen acht Datenbits der momentane Buszustand, d.h. ein undefinierter Wert in das Low-Byte eines Wortregisters geladen. Bei einem Wort-Schreibbefehl werden die niederwertigen acht Datenbits von der MMU ignoriert, d.h., diese Datenbits werden nur an den Bus ausgegeben.

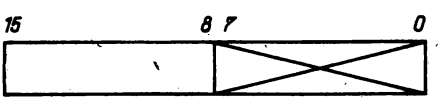


Bild 5.2. Datenformat der U8010-MMU

5.1. Schreib-/Lesebefehle

Die Schreib-/Lesebefehle dienen dem Datenaustausch zwischen der U8010-MMU und der U8001-CPU. Ein Byteregisterinhalt wird an die MMU ausgegeben bzw. von ihr gelesen. Soll ein Byte aus einem Segmentdeskriptorregister (SDR) gelesen werden, muß zuvor das Segmentadreßregister (SAR) zur Auswahl des SDR initialisiert werden. Die Adressierung der vier Byte des Segmentdeskriptorregisters erfolgt durch den Deskriptorselektionszähler (DSC).

Lesen/Schreiben Basisadreßfeld

Der Befehl dient dem Lesen oder Schreiben des 2-Byte-Basisadreßfelds im Segmentdeskriptorregister (SDR). Das SDR wird durch das Segmentadressenregister (SAR) ausgewählt. Die Adressierung des Basisadreßfelds im Segmentdeskriptorregister erfolgt durch den Deskriptorselektionszähler (DSC). Das High-Byte des Basisadreßfelds wird mit DSC = 0, das Low-Byte mit DSC = 1 adressiert. Beim Lesen oder Schreiben des High-Byte im Basisadressenfeld wird der DSC automatisch inkrementiert. Nach dem Lesen oder Schreiben des Low-Byte im Basisadreßfeld wird der Deskriptorselektionszähler wieder zu Null gesetzt.

Befehlskode: %08

```
Beispiel: LDB  RH4, #22        !SAR := 22!
          SOUTB %0102, RH4     !SAR der MMU 1 auf SDR 22 stellen!
          LDB RH4, #0          !DSC := 0!
          SOUTB %2002, RH4     !DSC Null setzen!
          SINB RH3, %0802      !High-Byte des Basisadressfelds laden!
          SINB RL3, %0802      !Low-Byte des Basisadressfelds laden!
```

Lesen/Schreiben Limitfeld

Mit diesem Befehl kann das Lesen oder Schreiben des 8-Bit-Limitfelds im Segmentdeskriptorregister (SDR) erfolgen. Das SDR wird durch das Segmentadressenregister (SAR) ausgewählt. Der Deskriptorselektionszähler (DSC) zeigt bei der Ausführung des Befehls automatisch auf das Limitfeld. Nach der Ausführung des Befehls wird der DSC wieder zu Null gesetzt.

Befehlskode: %09

```
Beispiel: LDB RH4, #63         !SAR := 63!
          SOUTB %0104, RH4     !SAR der MMU 2 auf SDR 63 stellen!
          SINB RH3, %0904      !Limit einlesen!
          INCB RH3, #12        !Limit um 3 KByte vergroessern!
          SOUTB %0904, RH3     !Neues Limit eintragen!
```

Lesen/Schreiben Attributfeld

Der Befehl dient zum Lesen oder Schreiben des 8-Bit-Attributfelds im Segmentdeskriptorregister (SDR). Das SDR wird durch das Segmentadreßregister (SAR) ausgewählt. Der Deskriptorselektionszähler (DSC) zeigt bei der Ausführung des Befehls automatisch auf das Attributfeld. Nach der Ausführung des Befehls wird der DSC wieder zu Null gesetzt.

Befehlskode: %0A

```
Beispiel: LDB RH4, #63          !SAR := 63!
          SOUTB %0110, RH4      !SAR der MMU 4 auf SDR 63 stellen!
          SINB RH3, %0A10       !Attributbyte einlesen!
```

Lesen/Schreiben Segmentdeskriptorregister

Der Befehl dient dem Lesen oder Schreiben des 4-Byte-Segmentdeskriptorregisters (SDR). Das SDR wird durch das Segmentadreßregister (SAR) ausgewählt. Die Adressierung der vier Byte im Segmentdeskriptorregister erfolgt durch den Deskriptorselektionszähler (DSC). Er muß vor Ausführung des Befehls zu Null gesetzt werden. Beim Lesen oder Schreiben der vier Byte des SDR wird er, beginnend mit dem High-Byte im Basisadreßfeld, automatisch inkrementiert. Nach dem Lesen oder Schreiben des SDR ist sein Inhalt wieder Null. Bei diesem Befehl werden vier Byte transferiert, deshalb müssen die Spezial-Ein-/Ausgaberepeatbefehle benutzt werden.

Befehlskode: %0B

```
Beispiel: LDB  RH4, #22         !SAR := 22!
          SOUTB %0120, RH4      !SAR der MMU 5 auf SDR 22 stellen!
          LDB RH4, #0           !DSC := 0!
          SOUTB %2020, RH4      !DSC Null setzen!
          LDA  RR4, DESKFLD22   !Adresse des Deskriptorfelds 22 laden!
          LD R3, #4             !Anzahl der zu transferierenden Byte!
          LD R2, #%0B20         !Befehl und MMU-Adresse laden!
          SOTIRB @R2, @RR4, R3  !Deskriptor in das SDR 22 laden!
```

Lesen/Schreiben Basisadreßfeld - Inkrement SAR

Der Befehl dient dem Lesen oder Schreiben des 2-Byte-Basisadreßfelds im Segmentdeskriptorregister (SDR) und dem Inkrementieren des Segmentadreßregisters (SAR). Das SDR wird durch das Segmentadreßregister (SAR) adressiert. Die Adressierung des Basisadreßfelds im Segmentdeskriptorregister erfolgt durch den Deskriptorselektionszähler (DSC). Er muß vor Ausführung des Befehls zu Null gesetzt sein. Beim Lesen oder Schreiben des Basisadreßfelds wird der DSC automatisch inkrementiert. Nach der Ausführung des Befehls wird das SAR inkrementiert und der DSC zu Null gesetzt. Mit den Spezial-Ein-/Ausgaberepeatbefehlen können alle Basisadreßfelder der MMU mit einem Befehl gelesen oder geschrieben werden.

Befehlskode: %0C

```
Beispiel: LDB  RH4, #00          !SAR := 00!
          SOUTB %0102, RH4       !SAR der MMU 1 auf SDR 0 stellen!
          LDB RH4, #0            !DSC := 0!
          SOUTB %2002, RH4       !DSC Null setzen!
          LDA  RR4, BASFLD       !Adresse der Basisadressfelder laden!
          LD R3, #21*2           !Anzahl der zu transferierenden Basisadr.!
          LD R2, #%0C02          !Befehl und MMU-Adresse laden!
          SOTIRB @R2, @RR4, R3 !Basisadressen in die SDR laden!
```

Lesen/Schreiben Limitfeld - Inkrement SAR

Mit diesem Befehl kann das Lesen oder Schreiben des 8-Bit-Limitfelds im Segmentdeskriptorregister (SDR) und das Inkrementieren des Segmentadreßregisters (SAR) erfolgen. Das SDR wird durch das Segmentadreßregister (SAR) ausgewählt. Der Deskriptorselektionszähler (DSC) zeigt bei der Ausführung des Befehls automatisch auf das Limitfeld. Nach der Ausführung des Kommandos wird der DSC zu Null gesetzt. Mit den Spezial-Ein-/Ausgaberepeatbefehlen können alle Limitfelder der MMU mit einem Befehl gelesen oder geschrieben werden.

Befehlskode: %0D

```
Beispiel: LDB RH4, #10           !SAR := 10!
          SOUTB %0104, RH4       !SAR der MMU 2 auf SDR 10 stellen!
          LDA  RR4, LIMFLD       !Adresse der Limitfelder im Speicher laden!
          LD R3, #4              !Anzahl der zu transferierenden Limitbytes!
          LD R2, #%0D04          !Befehl und MMU-Adresse laden!
          SOTIRB @R2, @RR4, R3 !Limitbytes in die SDR laden!
```

Lesen/Schreiben Attributfeld - Inkrement SAR

Der Befehl dient zum Lesen oder Schreiben des 8-Bit-Attributfelds im Segmentdeskriptorregister (SDR) sowie zum Inkrementieren des Segmentadreßregisters (SAR). Das SDR wird durch das Segmentadreßregister (SAR) ausgewählt. Der Deskriptorselektionszähler (DSC) zeigt bei der Ausführung des Befehls automatisch auf das Attributfeld. Nach der Ausführung des Kommandos wird der DSC zu Null gesetzt. Mit den Spezial-Ein-/Ausgaberepeatbefehlen können alle Attributfelder der MMU mit einem Befehl gelesen oder geschrieben werden.

Befehlskode: %0E

```
Beispiel: LDB RH4, #10          !SAR := 10!
          SOUTB %0104, RH4      !SAR der MMU 2 auf SDR 10 stellen!
          LDA  RR4, ATRFLD      !Adresse der Attributfld. im Speicher lad.!
          LD R3, #4             !Anzahl der zu transf. Attributbytes!
          LD R2, #%0E04         !Befehl und MMU-Adresse laden!
          SOTIRB @R2, @RR4, R3  !Attributbytes in die SDR laden!
```

Lesen/Schreiben Segmentdeskriptorregister - Inkrement SAR

Mit diesem Befehl kann das Lesen oder Schreiben des 4-Byte-Segmentdeskriptorregisters (SDR) und das Inkrementieren des Segmentadreßregisters (SAR) erfolgen. Das SDR wird durch das Segmentadreßregister (SAR) ausgewählt. Die Adressierung der vier Byte des Segmentdeskriptorregister erfolgt durch den Deskriptorselektionszähler (DSC). Er muß vor Ausführung des Befehls zu Null gesetzt werden. Beim Lesen oder Schreiben der vier Byte des SDR wird er automatisch inkrementiert. Nach den Lesen oder Schreiben des SDR ist sein Inhalt wieder Null. Mit den Spezial-Ein-/Ausgaberepeatbefehlen können alle Segmentdeskriptorregister der MMU mit einem Befehl gelesen oder geschrieben werden.

Befehlskode: %0F

```
Beispiel: LDB  RH4, #22         !SAR := 22!
          SOUTB %0120, RH4      !SAR der MMU 5 auf SDR 22 stellen!
          LDB RH4, #0           !DSC := 0!
          SOUTB %2020, RH4      !DSC Null setzen!
          LDA  RR4, DESKFLD22   !Adresse des Deskriptorfelds 22 laden!
          LD R3, #10*4          !Anzahl der zu transferierenden Bytes!
          LD R2, #%0F20         !Befehl und MMU-Adresse laden!
          SOTIRB @R2, @RR4, R3  !Deskriptor in SDR laden!
```

Lesen/Schreiben Moderegister

Der Befehl dient zum Lesen oder Schreiben des 8-Bit-Moderegisters. Mit ihm kann die MMU nach Programmierung der Segmentdeskriptorregister aktiviert werden.

Befehlskode: %00

```
Beispiel: LDB  RH4, #0          !Programmierung der MMU 1 beenden!
          SOUTB %01202 RH4      !SAR := 0!
          SOUTB %2002, RH4      !DSC := 0!
          LD RH4, #%C1          !MSEN und TRANS setzen, ID-Feld := 1!
                                !URS, MST und NMS rueckgesetzt!
          SOUTB %0002, RH4      !Im System arbeitet eine MMU im Bereich!
                                !der Segmentnummern 0 - 63!
```

Lesen/Schreiben Segmentadreßregister (SAR)

Mit diesem Befehl kann das 8-Bit-Segmentadreßregister (SAR) gelesen oder geschrieben werden. Es muß vor dem Zugriff auf ein oder mehrere Segmentdeskriptorregister mit der jeweiligen Segmentnummer initialisiert werden.

Befehlskode: %01

```
Beispiel: LDB RH4, #0           !DSC := 0!
          SOUTB %2020, RH4      !DSC Null setzen!
          LDB  RH4, #22         !SAR := 22!
          SOUTB %0120, RH4      !SAR der MMU 5 auf SDR 22 stellen!
```

Lesen/Schreiben Deskriptorselektionszähler (DSC)

Der Befehl dient dem Lesen oder Schreiben des Deskriptorselektionszählers (DSC). Der Deskriptorselektionszähler muß bei einigen Befehlen vor ihrer Anwendung mit Null initialisiert werden.

Befehlskode: %20

```
Beispiel: LDB  RH4, #00         !SAR := 00!
          SOUTB %0120, RH4      !SAR der MMU 5 auf SDR 0 stellen!
          LDB RH4, #0           !DSC := 0!
          SOUTB %2020, RH4      !DSC Null setzen!
          LDA  RR4, DESKFLD     !Adresse der Deskriptorfelder laden!
          LD R3, #64*4          !Anzahl der zu transferierenden Bytes!
          LD R2, #%0F20         !Befehl und MMU-Adresse laden!
          SOTIRB @R2, @RR4, R3  !Deskriptorregister SDR laden!
```

Lesen Violation-Type-Register (VTR)

Der Befehl dient dem Lesen des 8-Bit-Violation-Type-Registers. Er ermöglicht die Auswertung der Ursache der Zugriffsverletzung. Das VTR kann mit einem speziellen Befehl rückgesetzt werden.

Befehlskode: %02

```
Beispiel: SINB RH4, %0202          !RH4 := VTR der MMU 1!
          BITB RH4, #7
          JR   NZ, FATL_           !Sprung zur FATL-Behandlung!
          BITB, RH4, #6
          JR NZ, SWW_              !Sprung zur SWW-Behandlung!

          !Primärfehler Auswertung!
```

Lesen Violation-Segment-Nummer (VSN)

Der Befehl dient dem Lesen der 8-Bit-Violationsegmentnummer aus dem Violationsegmentregister, d.h. der Nummer des Segments, in dem es zur Verletzung der Zugriffsbedingungen kam.

Befehlskode: %03

```
Beispiel: SINB RH6, %0302          !RH6 := VSN der MMU 1!
```

Lesen Violation-Offset (VOFF)

Mit diesem Befehl wird das High-Byte des Offset (AD_8 - AD_{15}) aus dem 8-Bit-Violationoffsetregister gelesen, d.h. der Adresse, die Ursache für die Verletzung der Zugriffsbedingungen war.

Befehlskode: %04

```
Beispiel: SINB RH7, %0402          !RH7 := VOFF der MMU 1!
```

Lesen Buszyklusstatus (BCSR)

Der Befehl dient dem Lesen des 8-Bit-Buszyklusstatusregisters zur Feststellung des Systemstatus während der Verletzung der Zugriffsbedingungen.

Befehlskode: %05

```
Beispiel: SINB RH4, %0502          !RH4 := BCSR der MMU 1!
          BITB RH5, #5
          JR   Z, SYSTEM           !Sprung wenn der Fehler im Systemmode !
                                   !auftrat!
          BITB RH4, #4
          JR   Z, WRITE            !Sprung wenn der Fehler bei einem!
                                   !Schreibzyklus auftrat!

          !Auswertung des CPU-Status!
```

Lesen Befehlssegmentnummer (BSN)

Der Befehl dient dem Lesen der 8-Bit-Befehlssegmentnummer aus dem Befehlssegmentregister, d.h. der Nummer des Segmentes, in dem der Befehl stand, der zur Verletzung der Zugriffsbedingungen führte.

Befehlskode: %06

```
Beispiel: SINB RH6, %0602          !RH6 := BSN der MMU 1!
```

Lesen Befehlsoffset (BOFF)

Der Befehl dient dem Lesen des High-Byte (AD_8 - AD_{15}) des Befehlsoffset aus dem Befehlsoffsetregister, d.h. der Adresse des Befehls, der zur Verletzung der Zugriffsbedingungen führte.

Befehlskode: %07

```
Beispiel: SINB RH6, %0702          !RH6 := BSN der MMU 1!
```

5.2. Set-/Resetbefehle

Die Set-/Resetbefehle dienen dem Setzen aller CPUI- und DMAI-Flags einer U8010-MMU, dem Rücksetzen des Violation-Type-Registers (VTR), des SWW- und das FATL-Flags.
Diese Befehlsgruppe nutzt nur die Spezial-Ausgabebefehle. Der ausgegebene Registerinhalt wird von der MMU ignoriert.

Setzen aller CPUI-Flags

Mit diesem Befehl werden die CPUI-Flags in allen Attributfeldern der 64 Segmentdeskriptorregister einer MMU gesetzt.
Der Befehl dient in Systemen mit mehreren MMU-Schaltkreisen zur schnellen Umschaltung der Attributfelder auf den DMA-Transfer.

Befehlskode: %15

```
Beispiel: SOUTB %1510, RH4          !Alle 64 CPUI-Flags der MMU 4 := 1 !
```

Setzen aller DMAI-Flags

Mit diesem Befehl werden die DMAI-Flags in allen Attributfeldern der 64 Segmentdeskriptorregister einer MMU gesetzt.
Der Befehl dient in Systemen mit mehreren MMU-Schaltkreisen zur schnellen Umschaltung der Attributfelder nach einem DMA-Transfer. Nach diesem Befehl ist die CPU wieder in der Lage, über diese MMU auf den Speicher zuzugreifen.

Befehlskode: %16

```
Beispiel: SOUTB %1610, RH4          !Alle 64 DMAI-Flags der MMU 4 := 1 !
```

Rücksetzen des Violation-Type-Registers

Der Befehl dient nach der Auswertung der Ursachen der Zugriffsverletzung dem Rücksetzen der MMU, d.h. der Rückkehr der U8010-MMU in den Homezustand. Es entspricht einem Software-Reset der MMU nach dem Segmenttrap. Der Befehl wirkt nur auf die Statusregister der MMU.

Befehlskode: %11

```
Beispiel: SOUTB %1102, RH4     !MMU 1   VTR := 00 !
```

Rücksetzen des SWW-Flags im Violation-Type-Register

Der Befehl dient nach dem Auftreten von Schreibwarnungen durch Stackoperationen im Systemmode dem Rücksetzen des SWW-Flags im Violation-Type-Register und damit der Rückkehr in den Primärfehlerzustand.
Nach dem Rücksetzen des SWW-Flags ist die MMU wieder in der Lage, bei erneuten Schreibwarnungen im Systemmode einen Segmenttrap zu generieren.

Befehlskode: %13

```
Beispiel: SOUTB %1304, RH4          !MMU 2   SWW-Flag := 0 !
```

Rücksetzen des FATL-Flags im Violation-Type-Register

Der Befehl dient nach Auftreten von Folgefehlern dem Rücksetzen des FATL-Flags im Violation-Type-Register.
Er bewirkt den Übergang der MMU aus dem FATL-Zustand in den Primärfehlerzustand oder aus dem FATL&SWW-Zustand in den SWW-Zustand. Nach dem Rücksetzen des FATL-Flags ist die MMU wieder in der Lage, bei erneut auftretenden Fehlern einen Segmenttrap zu generieren.

Befehlskode: %14

```
Beispiel: SOUTB %1404, RH4          !MMU 2   FATL-Flag := 0 !
```

5.3. Abarbeitung von Segmenttraps

Segmenttraps werden im Unterbrechungssystem der U8001-CPU wie Interrupts behandelt. Ihre Priorität liegt gemäß Abschnitt 2.3.6. unter dem NMI, jedoch vor dem nichtvektorisierten und den vektorisierten Interrupts.
Der Segmenttrap wird im Segmenttrapannahmezyklus (segment trap acknowledge cycle) von der U8001-CPU angenommen. Sie geht in den Systemmode über und liest aus der Programmstatusarea die Adresse der Segmenttraproutine.
Nach Annahme des Segmenttrap legt die MMU das Kennwort, das Flag- und Controlwort, die Segmentnummer und den Offset im Systemstackbereich ab.

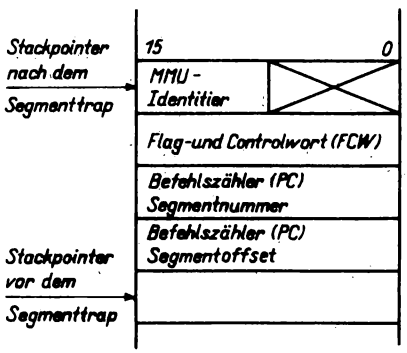


Bild 5.3. Systemstack nach einem Segmenttrap

Das MMU-Kennwort enthält im High-Byte die Adresse der MMU. Die Adresse wird im-1 aus-8-Kode ausgegeben. Sie wird bei der Programmierung in das 3-Bit-ID-Feld des Moderegisters eingetragen. Das Low-Byte des MMU-Kennwortes enthält einen undefinierten Wert. Im Bild 5.3 ist der Systemstack nach einem Segmenttrap abgebildet.
Nach Abschluß des Segmenttrapannahmezyklus ist die U8001-CPU sofort wieder in der Lage, einen weiteren Segmenttrap zu akzeptieren.
Wird, z.B. durch die o.g. Stackpointeroperation im Systemmode eine Schreibwarnung generiert, dann wird das SWW-Flag gesetzt und ein nachfolgender Segmenttrap durch die MMU generiert. Anschließend kann bis zum Rücksetzen des SWW-Flags kein weiterer Segmenttrap durch eine Stackoperation mehr ausgelöst werden.
Die Segmenttraproutine muß zu Beginn die in ihr benötigten Register in den Speicher auslagern. In Mikrorechnersystemen mit mehreren U8010-MMU-Schaltkreisen muß die Adresse der MMU aus dem Stack gelesen werden. Anschließend wird aus dem Inhalt des Violation-Type-Registers ermittelt, in welchem Zustand sich die MMU befindet. Dazu wird gemäß Abschnitt 4.3.3. als erstes das FATL-Flag und anschließend das SWW-Flag geprüft. Die Flags dürfen erst nach Beseitigung der Fehlerursache zurückgesetzt werden. Damit kann die erneute Generierung eines Segmenttraps auf Grund bereits bestehender Fehlerbedingungen vermieden werden.
Die Segmenttraproutine wird im Systemmode, bei gesetztem SEG-Bit im Flag- und Controlwort, durch einen IRET-Befehl abgeschlossen.

5.4. Programmierbeispiele

5.4.1. Programmierung einer U8010-MMU in einem U8001-Mikrorechnersystem

Im Beispiel wird davon ausgegangen, daß auf der physischen Speicheradresse 0 ein Speicherbereich von 64 KByte Festwertspeicher beginnt. Nachfolgend sind 256 KByte RAM-Speicher angeordnet. Der Speicher wird in der angegebenen Reihenfolge verwaltet. Das Segment 0 ist ein Read-Only-Segment, die Segmente 1, 2 und 3 werden für die Abarbeitung von Programmen reserviert. Das Segment 1 steht nur Systemprogrammen zur Verfügung.
Im Segmnt 4 (48 KByte) werden Daten gespeichert, das Segment 5 dient als Stackspeicherbereich.

```
MMU.INIT procedure
ENTRY
!Deskriptorselektionszaehler- und Segmentadressregister ruecksetzen!
     LDB  RH4, #0
     SOUTB %2002, RH4          !DSC := 0!
     SOUTB %0102, RH4          !SAR := 0!

!Deskriptorregister schreiben!
     LDA R4, DSKFLD            !Adresse des Deskriptorfelds laden!
     LD R3, 6*4                !Anzahl der zu transferierenden Bytes!
     LD R2, #0F02              !Befehl und MMU-Adresse laden!
     SOTIRB @R2, @R4, R3       !MMU programmieren!
     LDB RH4, #0               !Programmierung der MMU 1 beenden!

!MMU aktivieren; Moderregister schreiben!
     LD RH4, #%C1              !MSEN und TRANS setzen, ID-Feld := 1!
     SOUTB %0002, RH4          !URS, MST und NMS rueckgesetzt!
     RET
END  MMU.INIT


!******************!
!MMU-Programierwerte!
!******************!
DSKFLD:
DSKFLD0:  ARRAY [4 BYTE] := [%00 %00 %FF %09] !Attr. EXC, RD!
DSKFLD1:  ARRAY [4 BYTE] := [%01 %00 %FF %0B] !Attr. EXC, SYS, RD!
DSKFLD2:  ARRAY [4 BYTE] := [%02 %00 %FF %09] !Attr. EXC, RD!
DSKFLD3:  ARRAY [4 BYTE] := [%03 %00 %FF %09]
DSKFLD4:  ARRAY [4 BYTE] := [%04 %00 %C0 %00] !48 KByte Datenbereich!
DSKFLD5:  ARRAY [4 BYTE] := [%04 %C0 %C0 %20] !Attr. DMAI!
```

5.4.2. Programmierung von zwei U8010-MMU-Schaltkreisen in einem U8001-Mikrorechnersystem

Das Beispiel sieht die Verwaltung des physisch zur Verfügung stehenden Speicherbereichs durch zwei U8010-MMU-Schaltkreise vor. Eine MMU verwaltet den Speicherbereich, der im Systemmode genutzt wird, die zweite MMU verwaltet den im Normalmode genutzten Speicherbereich.
Es wird davon ausgegangen, daß auf der physischen Speicheradresse 0 ein Speicherbereich von 64 KByte Festwertspeicher beginnt. Nachfolgend sind sechs 256 KByte RAM-Speicher angeordnet. Die Segmente 0 bis 3 werden für die Arbeit im Systemmode genutzt.
Die obere Hälfte des physischen Speichers wird in acht logische Segmente aufgeteilt. Die Segmente 0 - 1 im Normalspeicherbereich dienen der Programmabarbeitung, die Segmente 2 - 8 dienen der Datenspeicherung, Segment 7 ist der Normalstackbereich.

```
MMU.INIT procedure
     ENTRY
!Deskriptorselektionszaehler- und Segmentadre~register ruecksetzen!
     LDB  RH4, #0
     SOUTB %2002, RH4          !SYSMMU  DSC := 0!
     SOUTB %0102, RH4          !SYSMMU  SAR := 0!
     SOUTB %2004, RH4          !NRMMMU  DSC := 0!
     SOUTB %0104, RH4          !NRMMMU  SAR := 0!
!SYSMMU  Deskriptorregister schreiben!
     LDA R4, SYSDSC0           !Adresse des Deskriptorfelds laden!
     LD R3, 4*4                !Anzahl der zu transferierenden Bytes!
     LD R2, #0F02              !Befehl und MMU-Adresse laden!
     SOTIRB @R2, @R4, R3       !MMU programmieren!
!NRMMMU  Deskriptorregister schreiben!
     LDA R4, NRMDSC0           !Adresse des Deskriptorfelds laden!
     LD R3,  8*4               !Anzahl der zu transferierenden Bytes!
     LD R2, #0F04              !Befehl und MMU-Adresse laden!
     SOTIRB @R2, @R4, R3       !MMU programmieren!
!SYSMMU aktivieren, Moderegister schreiben!
     LD RH4, #%D1              !MSEN, MST und TRANS setzen, ID-Feld := 1!
     SOUTB %0002, RH4          !URS und NMS rueckgesetzt!
!NRMMMU aktivieren, Moderegister schreiben!
     LD RH4, #%DA              !MSEN, MST, NMS und TRANS setzen,
                               ID-Feld := 2!
     SOUTB %0004, RH4          !URS rueckgesetzt!
     RET
END  MMU.INIT

!******************!
!MMU-Programierwerte!
!******************!

SYSDSC0:  ARRAY [4 BYTE] := [%00 %00 %FF %0B] !Attr. EXC, SYS, RD!
SYSDSC1:  ARRAY [4 BYTE] := [%01 %00 %FF %0B]
SYSDSC2:  ARRAY [4 BYTE] := [%02 %00 %C0 %02] !Attr. SYS!
SYSDSC3:  ARRAY [4 BYTE] := [%02 %00 %C0 %22] !Attr. DIRW, SYS!
NRMDSC0:  ARRAY [4 BYTE] := [%03 %00 %FF %09] !Attr. EXC, RD!
NRMDSC1:  ARRAY [4 BYTE] := [%04 %00 %FF %09] !Attr. EXC, RD!
NRMDSC2:  ARRAY [4 BYTE] := [%05 %00 %80 %00] !32 KByte Datenbereich!
NRMDSC3:  ARRAY [4 BYTE] := [%05 %80 %80 %00] !32 KByte Datenbereich!
NRMDSC4:  ARRAY [4 BYTE] := [%06 %00 %40 %00] !16 KByte Datenbereich!
NRMDSC5:  ARRAY [4 BYTE] := [%06 %40 %40 %00] !16 KByte Datenbereich!
NRMDSC6:  ARRAY [4 BYTE] := [%06 %80 %40 %00] !16 KByte Datenbereich!
NRMDSC7:  ARRAY [4 BYTE] := [%06 %C0 %C0 %20] !Attr. DIRW!
```

5.4.3. Aufbau einer Segmenttraproutine

Das Beispiel stellt eine einfache Segmenttraproutine dar. An das System ist ein Display angeschlossen, alle Fehlermeldungen werden über dieses Display an den Bediener ausgegeben. Die Fehlermeldungen enthalten die Fehlerkennzeichnung und die in den Statusregistern abgelegten Informationen über die Fehlerursache. Es wird angenommen, daß der Ansprung der Traproutine im segmentierten Mode erfolgt. Wird der FATL- und /oder der SWW-Zustand festgestellt, werden ebenfalls die Fehlerinformationen ausgegeben, anschließend erfolgt der Rücksprung in eine spezielle Fehlerroutine des Betriebssystems.

```
!**************************!
!Segementtrapserviceroutine !
!**************************!

TRAP procedure
     ENTRY
     PUSHL      @RR14, RR0       !Register R0 bis R5 retten!
     PUSHL      @RR14, RR2
     PUSHL      @RR14, RR4

!CPU in den nichtsegmentierten Mode bringen!
     LDCTL      R0, FCW
     RES        R0, #15          !SEG-Bit ruecksetzen!
     LDCTL      FCW, R0

!Statusregister der MMU in einen Speicherbereich transferieren!
     CALL RDSTATE

!VTR lesen!
     SINB       RL4, %0202

!Zustand der MMU bestimmen!
     BITB       RL4, #7          !FATL gesetzt ? !
     JR         NZ, FATL_STATE
     BITB       RL4, #6          !SWW gesetzt ? !
     JR         Z, PRIMFLG       !Sprung, wenn primaerer Fehler!

SWW_STATE:
     LDA        R2, SWW          !Adresse der Fehlermeldung laden!
     LD         R1, #0           !Param. fuer Ausgabeprogramm!
     CALL       PUTMSG           !Fehlermeldung und Status ausgeben!
     JR         SYSRET

FATL_STATE:
     LDA        R2, FATL
     CALL       PUTMSG           !Fehlermeldung und Status ausgeben!
     BITB       RL4, #6
     JR         NZ, SYSRET

     LD         R1, #1           !Param. fuer Ausgabeprogramm!

     LDA        R2, SWW
     CALL       PUTMSG           !Fehlermeldung ausgeben!

SYSRET:
     LDPS       SYS_STATE        !Ruecksprung in das Betriebssystem!
```

```
!Ausgabe des Primaerfehlers!
PRIMFLAG:
     DECB      RL4              !Vorbereitung der Adressrechnung!
     SLAB      RL4
     CLRB      RH4
     LDA       R2, RDV
     ADD       R2, R4           !R2 zeigt auf den Anfang der Textkette!
     LD        R1, #0           !Param. fuer Ausgabeprogramm!
     CALL      PUTMSG           !Fehlermeldung und Status ausgeben!
     LDCTL     R0, FCW
     SET       R0, #15          !SEG-Bit setzen!
     LDCTL     FCW, R0

!VTR Ruecksetzen!
     SOUTB     %1102, RH0

     POPL      RR4, @RR14       !Register R0 bis R5 zurueckladen!
     POPL      RR2, @RR14
     POPL      RR0, @RR14
     IRET
end TRAP


SYS_STATE:
          array     [4 word] := [%0000 %C000 %8000 FAT_COND]

!***********!
!Fehlertexte!
!***********!
MSG:      array [* word]  := [RDV SYSV SLV EXCV CPUIV PWW SWW FATL]
RDV:      array [19 byte] := 'read-only violation'
SYSV:     array [16 byte] := 'system violation'
SLV:      array [24 byte] := 'segment length violation'
EXCV:     array [22 byte] := 'execute-only violation'
CPUIV:    array [21 byte] := 'CPU inhibit violation'
PWW:      array [21 byte] := 'primary write warning'
SWW:      array [20 byte] := 'second write warning'
FATL:     array [21 byte] := 'fatal error condition'
```

6. Programmierumgebung

Die Programmierumgebung für das 16-Bit-Mikroprozessorsystem U8000 wird durch die zur Anwendung gelangende Rechentechnik und die Systemsoftware bestimmt. Der Bürocomputer BC A5120 auf der Basis des 8-Bit-Mikroprozessorsystems U880 kann unter dem Betriebssystem UDOS als Cross-Entwicklungssystem eingesetzt werden. Neben dem BC A5120 können U8000-Programme auch auf dem Programmier- und Entwicklungssystem P8000 entwickelt werden. Das P8000 ist ein Mikrorechnersystem auf Basis des 16-Bit-Mikroprozessorsystems U8000, zur Abarbeitung bereits vorhandener 8-Bit-Software ist auch das 8-Bit-Mikroprozessorsystem U880 im P8000 integriert. Die Arbeit kann dabei sowohl unter dem 8-Bit-Betriebssystem UDOS als auch unter dem 16-Bit-Betriebssystem WEGA durchgeführt werden. Das Betriebssystem WEGA /9/ läuft auf dem P8000 auf dem 16-Bit-Mikroprozessorsystem U8000. Die vorgestellten Systemprogramme basieren in beiden Betriebssystemen auf einem PLZ/ASM-Übersetzer. Dieses Übersetzungsprogramm wird als Assembler bezeichnet, es erfordert in beiden Betriebssystemen gleiche Programmstrukturen für die U8000-Quellprogramme. Nachfolgend werden einige Hinweise zur Nutzung der Systemprogramme für die U8000-Programmentwicklung auf beiden Mikrorechnersystemen gegeben. Eine detaillierte Beschreibung der Systemprogramme sowie der Programmstruktur ist den jeweiligen Handbüchern /13/ zu entnehmen.

6.1. Aufbau von PLZ/ASM-Programmen

Die PLZ/ASM-Programme müssen, im Gegensatz zu Programmen für das Mikroprozessorsystem U880, nach bestimmten Merkmalen aufgebaut sein. Nachfolgend werden einige grundlegende Merkmale des PLZ/ASM-Übersetzers anhand von Beispielen für häufig benutzte Notationsformen vorgestellt.
Die Struktur wird im wesentlichen durch Module, Prozeduren und Datenvereinbarungen bestimmt.
Module sind getrennt übersetzbare Teile eines Programms, ein Programm kann mehrere Module enthalten. Die Module bestehen aus Prozeduren und Datenvereinbarungen.
Prozeduren bestehen aus Assemblerbefehlen, zur Verbesserung der Übersichtlichkeit können Elemente, die sonst nur in höheren Sprachen zur Verfügung stehen, eingefügt werden. Die höheren Sprachelemente werden in dieser kurzen Übersicht nicht behandelt.
Module und Prozeduren werden mit Namen bezeichnet, über diese Namen sind sie adressierbar. In den Prozeduren können zur Organisation von Programmverzweigungen Marken namen definiert werden.
Datenvereinbarungen dienen zur Definition von konstanten oder variablen Daten, sie können innerhalb und außerhalb von Prozeduren stehen.
Der Gültigkeitsbereich von Datenvereinbarungen, Marken und Prozeduren kann festgelegt werden. Es gibt die Gültigkeitsbereiche:

GLOBAL Auf Größen, die als GLOBAL vereinbart wurden, kann aus anderen Modulen zugegriffen werden. In diesen Modulen müssen sie als EXTERNAL vereinbart werden.

EXTERNAL Alle Größen, die aus anderen Modulen benötigt werden, sind als EXTERNAL anzugeben. In dem Modul, in dem sie definiert sind, müssen sie GLOBAL sein.

INTERNAL Der Gültigkeitsbereich dieser Größen ist der Modul, in dem sie vereinbart wurden.

LOCAL Der Gültigkeitsbereich dieser Größen ist die Prozedur, in der sie vereinbart wurden.

6.1.1. Datenvereinbarungen

Daten sind konstante oder variable Größen, sie müssen vor ihrer Benutzung definiert werden. Mit den Datenvereinbarungen wird ein Name, ein Datentyp und ein Gültigkeitsbereich für den jeweiligen Datenwert festgelegt.
Für Datenvereinbarungen existieren die nachfolgend erläuterten drei Arten von Anweisungen.

Konstantendefinition
Mit dieser Definition wird einem Namen ein Wert zugewiesen. Damit können konstante Größen unter einem Namen benutzt werden.
Beispiel:

```
CONSTANT .          !Definition!
    CR := 24        !CR = Name der Konstanten!
    LF := 80
```

Typdefinition
Variable Daten müssen zur Festlegung des erforderlichen Speicherbereichs genau definiert werden, in der Typdefinition wird auch der Name der Variablen festgelegt. Datentypen können in einer Variablendeklaration angegeben werden. Es gibt einfache und strukturierte Datentypen. Einfache Datentypen sind:

```
BYTE oder SHORT_INTEGER      ein Byte (8 Bit)
WORD oder INTEGER            ein Wort (16 Bit)
LONG oder LONG_INTEGER       zwei Wörter (32 Bit)
```

Beispiele für einfache Datentypdefinitionen:

```
TYPE                    !Definition!
    ASCCHR  BYTE        !ASCCHR = Name  BYTE = Typ!
    zeiger  WORD
```

Die strukturierten Datentypen werden in die folgenden Arten unterschieden:

ARRAY Die Vereinbarung dient zur Defininition von Daten eines Typs.

RECORD Die Vereinbarung dient zur Definition von Daten unterschiedlichen Typs.

Beispiele für strukturierte Datentypvereinbarungen:

```
TYPE                                    !Definition!
    TEXT ARRAY [64 BYTE]                !TEXT = Name  ARRAY = Typ!
    FELD ARRAY [24*50 BYTE]
TTL_IC  RECORD [stueckzahl,preis INTEGER
            typ RECORD [kennung,nummer BYTE]
            ]
```

Variablenvereinbarungen
Mit Variablenvereinbarungen kann den Variablen neben der Typzuweisung ein Anfangswert zugewiesen werden.
Externe Variablenvereinbarungen dürfen keine Anfangswertzuweisung enthalten.
Beispiele:

```
cnt    WORD := %0000
SYSMSG ARRAY [9 BYTE] := 'Guten Tag'
```

6.1.2. Markenvereinbarungen

Neben den bisher beschriebenen Vereinbarungen existiert noch eine Markenvereinbarung, mit ihr werden Namen von Marken vereinbart.
Bei der Definition wird dem Namen kein Doppelpunkt nachgestellt.
Eine Marke kann folgende Gültigkeitsbereiche haben: GLOBAL, EXTERNAL, INTERNAL oder LOCAL.
Beispiel:

```
GLOBAL
     m2  LABEL

     bsp PROCEDURE        !Prozedur ist global!
         LOCAL
             .
         m1 LABEL
         ENTRY
            m1:           !m1 ist lokal zu bsp!
             .
            m2:           !m2 ist global!
             .
            m3:           !m3 ist intern zum Modul!

            m4:           !m4 ist lokal zu bsp!

     END bsp
```

6.1.3. Struktur von PLZ/ASM-Programmen

PLZ/ASM-Programme werden in /8/ ausführlich erläutert, sie müssen nach den Regeln der Sprache PLZ aufgebaut werden. Sie werden, basierend auf den Grundstrukturen Modul und Prozedur, entwickelt. Ein Programm wird durch Module strukturiert, ein Modul durch Prozeduren und Vereinbarungen. Die Prozeduren sind die ausführbaren Programmteile des Moduls.
Die Modulvereinbarung hat folgenden Aufbau:

```
modulname  MODULE
             .
     Vereinbarungen
             .
     Prozeduren

END  modulname
```

Die Bezeichnung "modulname" entspricht einem Namen. Prozeduren können auch Vereinbarungen enthalten.
Eine Prozedurvereinbarung definiert einen ausführbaren Programmteil mit einem Namen, so daß er adressiert werden kann. Im Prozedurkopf wird der Prozedurname spezifiziert. Er kennzeichnet die erste Anweisung der Prozedur und kann wie jede andere Programmarke verwendet werden. Eine Prozedurvereinbarung hat folgenden Aufbau:

```
prozedurname  PROCEDURE
        .
        Vereinbarung von Daten
        .
    ENTRY
        .
        Programmteil
        .
    END prozedurname
```

Mit den Strukturierungsanweisungen MODULE und PROCEDURE und den Möglichkeiten zur Datenvereinbarung lassen sich arbeitsfähige Programme formulieren.
Beispiel:

```
konvertierung module

    global
        KONV   byte
        OUTPTR wort
        OUTBFF array  128 byte

BTOH procedure  !BIN to HEX Konvertierung!
entry
    push    @r15, r1          !4 Bit Binaer in Hex-Konvertierung!
    andb    rl0, #%0F         !untere 4 Bit maskieren!
    ldb     rl1, KONV         !Hilfbyte laden!
    addb    rl1, rl0          !untere 4 Bit einblenden!
    ldb     KONV, rl1
    cpb     rl0, #%0A         !wenn rl0 < %0A ist, dann ist!
                              !eine Ziffer zu konvertieren!
    jr      c, ZIFFER
    addb    rl0, #%07         !Buchstabe konvertieren
ZIFFER:
    addb    rl0, #%30
    push    @r15, r6
    ld      r6, OUTPTR        !Ausgabezeiger fuer den Ausgabepuffer!
    ldb     OUTBFF(r6), rl0   !ASCII-Zahl in den Ausgabepuffer eintr.!
    inc     OUTPTR, #1        !Ausgabezeiger weiterstellen!
    pop     r6, @r15
    pop     r1, @r15
    ret
  end BTOH

end konvertierung
```

6.1.4. Assemblerdirektiven und Pseudobefehle

Die Direktiven sind Assemblersteueranweisungen und werden benutzt, um die Arbeitsweise des Assemblers zu steuern, sie sind Bestandteil des Quellprogramms und müssen allein auf einer Zeile stehen. Eine Assemblersteueranweisung muß als erstes Zeichen ein "$" haben und in Spalte 1 beginnen, unmittelbar gefolgt von der speziellen Direktive. Danach können noch Operanden folgen. Die Direktiven erscheinen im Listing.
Zusätzlich können noch erweiterte Befehle innerhalb einer Prozedur zur Erzeugung von Byte-, Wort- oder Langwortgrößen benutzt werden. Mit ihnen können für spezielle Anwendungen Pseudobefehle definiert werden.

$ABS Adresse — Der erzeugte Objektkode wird absolut adressiert. Falls eine Adresse spezifiziert ist, so beginnt der Befehlskodezähler mit dieser Adresse. Implizit wird er auf den Wert 0 gesetzt. Die Anweisung ist bis zum Auftreten einer $REL-Anweisung gültig.

$REL Adresse — Der erzeugte Objektkode wird verschieblich erstellt. Falls eine Adresse angegeben ist, so kennzeichnet sie die relative Distanz vom Anfang des aktuellen Bereichs (SECTION). Die Wirkung kann durch eine $ABS-Anweisung aufgehoben werden. Implizit ist $REL 0 gesetzt.

$DEFAULT — Beendet die Wirkung einer vorherigen $SECTION-Anweisung und stellt die Standard-Speicherzuweisung her. Muß angegeben werden, wenn nach $SECTION die Standardzuweisung wiederhergestellt werden soll.
Modulname_D Datenvereinbarungen
Modulname_P Prozedurvereinbarungen

$SECTION Name — Ein Programm kann in Abschnitte (Sektionen) eingeteilt werden, die beim Binden des Programms in zusammenhängende Speicherbereiche gebracht werden. Prozeduren und Daten können je nach Anwendungsfall in den gleichen Speicherbereich oder getrennt geladen werden. Implizit wird ein Programm in die Bereiche

"Modulname_D" für den Datenteil und
"Modulname_P" für den Programmteil getrennt.

Mit der Direktive $SECTION kann dieser Standard für den Assembler geändert werden.
Der "Name" verbindet den nachfolgend erzeugten Objektkode mit dem symbolischen Namen zur späteren Zuordnung in bestimmte Speicherbereiche.

$PAGE — Die Direktive bewirkt, daß im Listing eine neue Seite begonnen wird, sie erscheint selbst nicht im Listing.

$LISTON Option
$LISTOFF Option

Diese Direktive steuert das Listingformat.

Ohne "option" schaltet sie das Listing aus ($LISTOFF) oder ein ($LISTON). Standard ist $LISTON. Für "option" gibt es:

$TTY: Erzeugt ein 80spaltiges Listingformat. Standard ist $LISTOFF $TTY, d.h. volle Zeichenlänge.

$BEX: Steuert das Format des Objektkodes. $LISTOFF $BEX listet nur eine Objektkodezeile für eine Eingabezeile. Standard ist $LISTON $BEX, d.h., der gesamte Objektkode wird gelistet.

BVAL Ausdruck
WVAL Ausdruck
LVAL Ausdruck

Durch die Pseudoanweisung wird ein Byte definiert.
Es wird ein Wort definiert.
Es wird ein Langwort definiert

Bedingte Assemblierung

Für die bedingte Assemblierung wird die folgende Anweisung benutzt.

```
$IF  wert   $THEN

     Assembleranweisungen

$ELSE                        !optional!

     Assembleranweisungen

$FI
```

Ist die Größe "wert" ungleich 0, dann werden die Assembleranweisungen zwischen $THEN und $ELSE übersetzt. Ist sie 0, so werden nur die Assembleranweisungen zwischen $ELSE und $FI übersetzt. Schachtelungen sind erlaubt.

6.2. Programmentwicklung unter dem Betriebssystem UDOS

Das Betriebssystem UDOS unterstützt die U8000-Programmentwicklung durch die Programmierspache PLZ/ASM aus der PLZ-Sprachfamilie. Die U8000-Programmentwicklung wird durch die Systemprogramme U8000-Assembler, Linker und Imager unterstützt. Diese Systemprogramme dienen der Übersetzung von U8000-Quellprogrammen in U8000-Objektprogramme, dem Verbinden von separat übersetzten Objektprogrammen und dem Erstellen von abarbeitungsfähigen U8000-Maschinenprogrammen.

6.2.1. U8000-Assembler

Der PLZ/ASM-Assembler (U8000ASM) übersetzt eine U8000-Quelldatei in eine Objektdatei. Die Objektdatei enthält damit das Objektprogramm des U8000-Programms, sie kann absolut oder auch relativ ladbar erzeugt werden. Beim Arbeiten mit verschieblichen Programmen oder Modulen ist eine Neufestlegung der Adressen ohne nochmalige Übersetzung möglich. Weiterhin können die Programme in unabhängige Module aufgegliedert werden, die getrennt voneinander programmiert und übersetzt werden können.

Der U8000-Assembler wird mit folgender Kommandozeile aufgerufen:

```
U8000ASM Dateiname Optionen
```

Der "Dateiname" muß die Endung ".S" haben (sie braucht beim Aufruf nicht mit angegeben zu werden) und enthält die Quelle für einen einzigen U8000-Modul. Das Assemblerprogramm erzeugt implizit eine Objektkodedatei (Endung ".OBJ") und eine Listingdatei (Endung ".L").
Folgende Optionen können nach dem Dateinamen in beliebiger Reihenfolge angegeben werden:

D=string — Mit "string" wird eine Zeichenkette angegeben, sie kann aus max. 18 Zeichen bestehen und wird in der Kopfzeile des Listings angegeben.

I=Dateiname — Der bei der Übersetzung erzeugten Zwischendatei wird der Name "Dateiname" zugeordnet, sie wird nicht gelöscht. Ist "I" nicht spezifiziert, wird die Zwischendatei gelöscht.

L=Dateiname — Anstelle des aus dem Quelldateinamen mit der Erweiterung ".L" gebildeten Namens der Listingdatei, bekommt sie den Namen 'Dateiname'.

NOL — Es wird keine Listingdatei erstellt.

O=Dateiname — Anstelle des aus dem Quelldateinamen mit der Erweiterung ".OBJ" gebildeten Namen der Objektdatei bekommt die Objektdatei den Namen 'Dateiname'. Falls keine Objektdatei gewünscht wird, so ist O=$NULL anzugeben.

P — Eine Kopie der Listingdatei wird auf dem Bedienerterminal ausgegeben. Auf dem Bildschirm erscheinen keine Übersetzungsfehler. Ohne Angabe der Option werden Fehlernummer

und fehlerhafte Zeile auf dem Bildschirm ausgegeben.

Beispiel: U8000ASM MON.ROUT1 D='1. September 1986' L=ROUTINE1.L
Das Datum wird in den Listingkopf eingetragen. Der Name der Listingdatei wird mit ROUTINE1.L festgelegt.

6.2.2. Linker

Der Linker (ZLINK) verbindet mehrere separat durch den U8000-Assembler erzeugte Objektprogramme (Endung: ".OBJ") zu einem einzigen Objektprogramm. Dieses Objektprogramm kann wiederum mit weiteren Objektprogrammen zu einem neuen Objektprogramm zusammengebunden werden. Der Linker liefert somit wieder eine Datei mit der Endung ".OBJ". Die beim Linklauf entstehenden Informationen über Adressen, die Programmlänge und die bearbeiteten Marken können optional in einer Mapdatei ausgegeben werden.

Der Linker wird mit folgender Kommandozeile aufgerufen:

ZLINK Modulliste Optionen

Die in der Modulliste angegebenen Objektkodedateien müssen die Endung ".OBJ" besitzen, sie braucht beim Aufruf nicht angegeben zu werden.
Zur Verbesserung der Übersichtlichkeit der in der Modulliste angegebenen Objektkodedateien können sie unter Verwendung von runden Klammern zu Bereichen zusammengefaßt werden. Damit besteht die Möglichkeit, Datenbereiche mehrerer Objektmodule in dem neu entstehenden Objektmodul zu einem Bereich zusammenzufassen. Der neue Bereichsname wird, dem angegebenen Beispiel entsprechend, vor der Klammer angegeben.

Beispiel: ALL.TEMP=(PROC1.TEMP PROC2.TEMP PROC3.TEMP)

In der entstehenden Objektkodedatei und in der ggf. ebenfalls entstehenden Mapdatei erscheint nur noch der Bereich "ALL.TEMP". Zur Vereinfachung, den UDOS Namenskonventionen entsprechend, kann das Zeichen "*" als Abkürzung eingesetzt werden.

Beispiel: ALL.TEMP=(*TEMP)

Mit diesem Beispiel wird das gleiche Ergebnis wie im obigen Beispiel erreicht. Alle in der Modulliste zu einem Bereich zusammengefaßten Dateien sind Objektkodedateien, die zusätzlich die Endung ".OBJ" besitzen.

Der Linker kann mit den folgenden Optionen aufgerufen werden:

L=Dateiname Eine Mapdatei mit Informationen über die gelinkten Module wird erstellt und unter dem angegebenen Dateinamen abgelegt. Diese Datei wird sonst nicht erzeugt.

E=Name Der angegebene Name ist der Name einer Prozedur oder einer Marke. Wird die Option nicht angegeben, wird der Name der ersten Prozedur vom Linker zur Berechnung der Startadresse genutzt.

N=Dateiname Der Dateiname wird als Name für den durch den Linklauf entstehenden neuen Objektmodul eingesetzt. Wird die Option nicht angegeben, so wird der Name des ersten Objektkodemoduls aus der Modulliste genommen.

T=Dateiname Eine temporäre Zwischendatei wird unter dem Dateinamen auf dem Massenspeicher abgelegt. Diese Datei wird sonst automatisch gelöscht.

W=ON/OFF Mit dieser Option können Warnmeldungen während der Arbeit des Linkers die Ausgabe von Warnungen gesteuert werden. Ist diese Option nicht angegeben werden Warnungen unterdrückt.

K= Namensliste
K=ALL/ONE
K=ALLBUT Namensliste
Mit dieser Option wird die Eintragung von globalen Namen in den entstehenden Objektmodul gesteuert. Damit besteht die Möglichkeit sie in folgenden Linkläufen zu nutzen. Das Schlüsselwort ALL verursacht die Einbeziehung aller globalen Größen. Implizit ist das Schlüsselwort NONE gesetzt, d.h., es werden keine globalen Größen in die Objektkodedatei eingetragen. Wird das Schlüsselwort ALLBUT angegeben, werden alle globalen Größen, mit Ausnahme der in der Namensliste angegebenen, in die Objektkodedatei eingetragen.

Die Optionen werden durch das angegebene Zeichen gekennzeichnet und durch Leer- oder Tabulatorzeichen getrennt.

Beispiel:
```
ZLINK MON.INIT MON.ROUT0 MON.ROUT1 MON.RAM L=MON.MAP
N=MON E=ENTRY_POINT
```
Das Programm MON, bestehend aus den Modulen "MON.INIT", "MON.ROUT0", "MON.ROUT1" und "MON.RAM" wird gelinkt. Eine Mapdatei wird unter dem Namen "MONITOR.MAP" angelegt, der Name des gelinkten Moduls wird mit MON festgelegt. Die Startadresse ist die Adresse der Prozedur ENTRY_POINT.

6.2.3. Imager

Mit Hilfe des Imagerprogramms (IMAGER) wird aus einem durch den Assembler oder Linker erzeugten Objektprogramm ein U8000-Maschinenprogramm mit absoluten Adressen gebildet. Dieses Maschinenprogramm kann in den Arbeitsspeicher des Mikrorechners geladen und abgearbeitet werden.

Der Imager wird mit folgender Kommandozeile aufgerufen:

IMAGER Modulliste (Bereichsadressenliste) Fenster Optionen

Modulliste
Die in der Modulliste angegebenen Objektkodedateien müssen die vollständig anzugebende Endung ".OBJ" besitzen. Der Imager erweitert die in der Modulliste angegebenen Namen

nicht automatisch.

Bereichsadressenliste
Die Bereichsadressenliste besitzt den folgenden Aufbau:

- Segmentnummer ($=nnnn Bereichsliste)

Wird der Bereichsadressenliste das Minuszeichen vorangestellt, dann wird das folgende Segment eines segmentiert arbeitenden U8001-Programms nicht in das Maschinenprogramm einbezogen, es wird lediglich zur Adreßberechnung genutzt. Die Adressen der Bereiche werden mit $=nnnn hexadezimal angegeben, soll ein Bereich nicht mit in das Maschinenprogramm aufgenommen werden, so kann ihm ebenfalls ein Minuszeichen vorangestellt werden.

Fenster Im Fenster wird der physische Speicherbereich des Maschinenprogramms mit folgendem Aufbau vorgegeben:

'Beginnadresse Endadresse

Der Imager kann mit den folgenden Optionen aufgerufen werden:

E=nnnn Die angegebene hexadezimale Adresse ist die Startadresse des Maschinenprogramms. Wird keine Startadresse angegeben und ist in den Objektkodemodulen keine angegeben, setzt der Imager die im Fenster angegebene Anfangsadresse als Startadresse ein.

O=Dateiname Der Dateiname wird als Name für das durch den Imagerlauf entstehende Maschinenprogramm eingesetzt. Wird die Option nicht angegeben, so wird der Name des ersten Objektkodemoduls unter Weglassung der Endung ".OBJ" aus der Modulliste genommen.

Die Optionen werden durch das angegebene Zeichen gekennzeichnet und durch Leerzeichen getrennt. Der Imager unter UDOS kann Maschinenprogramme mit einem maximalen Speichervolumen von %8000 Byte erzeugen. Nicht belegte Speicherplätze werden mit %FF geladen.

Beispiel: IMAGER MON ($=0 PROM $=2000 RAM) {0 FFF} O=MON16
Aus dem Objektkodemodul "MON" wird ein ladbares Maschinenkodeprogramm, beginnend mit der SECTION PROM ab der Adresse 0, hergestellt. Ab der Adresse %2000 wird die SECTION RAM vereinbart. Der Speicherbereich bis %FFF steht zur Verfügung. Die Datei wird unter der Bezeichnung "MON16" erstellt.

6.3. Programmentwicklung unter dem Betriebssystem WEGA

Die Programmentwicklung für das Mikroprozessorsystem U8000 wird unter dem UNIX-kompatiblen Betriebssystem WEGA auf dem Programmier- und Entwicklungssystem P8000 durch folgende Softwarekomponenten unterstützt:

```
cas'    U8000-Assembler      (für segmentierte Programme)
as      U8000-PLZ-Assembler  (für nichtsegmentierte Programme)
scc     U8000-C-Compiler     (für segmentierte Programme)
cc      U8000-C-Compiler     (für nichtsegmentierte Programme)
sld     Lader                (für segmentierte Programme)
ld      Lader                (für nichtsegmentierte Programme)

plz     U8000-PLZ-Compilerdriver
plzsys  U8000-PLZ/SYS-Compiler
plzcd   U8000-Kodegenerator
```

Unter segmentierten Programmen ist hier die Entwicklung von Programmen zu verstehen, die mehr als 64 KByte Speicherplatz benötigen, nichtsegmentierte Programme können in einem Segment von 64 KByte Speichervolumen arbeiten. Die bei der Übersetzung entstehenden Module lassen sich nicht mit jedem Systemprogramm weiterverarbeiten; es ist auch nicht möglich, sie beliebig mit anderen bei der Übersetzung entstehenden Modulen zu verbinden. Im einzelnen bestehen die folgenden Kombinationsmöglichkeiten:

```
cas     mit ld und sld
cc      mit cas und ld
scc     mit cas und sld
as      mit ld
plz     mit plzsys und plzcg
plzsys mit plzcg
```

Die Erläuterung dieser Softwarewerkzeuge würde den Rahmen des Bandes überschreiten. Die folgenden Ausführungen beziehen sich deshalb auf zwei ausgewählte Softwaresysteme, den U8000-Assembler "as" und den Lader "ld". Diese Systemprogramme dienen der Übersetzung von U8000-Quellprogrammen in U8000-Objektprogramme, dem Verbinden separat übersetzter Objektprogramme sowie dem Generieren abarbeitungsfähiger U8000-Maschinenprogramme. Der Dialog zwischen dem Bediener und dem Betriebssystem WEGA erfolgt vorrangig mit Kleinbuchstaben; deshalb werden in den Kommandozeilen Kleinbuchstaben angegeben. Die Programmentwicklung kann mit dem PLZ-Assembler sowohl für Programme unter dem Betriebssystem WEGA als auch für Programme, die auf anderen U8000-Mikrorechnern eingesetzt werden, erfolgen. Die Programmierung für das Betriebssystem WEGA wird in /9/ ausführlich behandelt. In den folgenden Ausführungen wird der Schwerpunkt auf die Programmentwicklung von U8000-Programmen, die nicht WEGA-spezifisch sind, gelegt.

6.3.1. U8000-Assembler

Der WEGA PLZ/ASM-Assembler (as) übersetzt eine U8000-Quelldatei in eine Objektdatei. Die Objektdatei kann absolut oder auch verschieblich ladbar erzeugt werden. Der Aufbau der U8000-Programme erfolgt ebenfalls nach den im Abschnitt 6.1.1. behandelten Grundsätzen.

Der Assembler wird mit folgender Kommandozeile aufgerufen:

```
as optionen dateiname
```

Der Dateiname der PLZ/ASM-Quelldatei muß die Endung ".s" besitzen. Die Endung muß mit dem dateinamen angegeben werden. Das Assemblerprogramm erzeugt implizit eine Objektkodedatei mit der Bezeichnung "a.out". Diese Objektkodedatei wird im WEGA-Format angelegt.
Das Erzeugen einer PLZ/ASM-Datei muß durch die Option -z vorgeschrieben werden, sie erhällt die Bezeichnung "t.out".
Die Erstellung einer Listingdatei muß als Option eingegeben werden. Sie wird mit dem Dateinamen und der Endung ".l", anstelle der Endung ".s", erstellt.
Folgende Optionen können, durch Leer- oder Tabulatorzeichen getrennt, in beliebiger Reihenfolge angegeben werden.

-f	Die Übersetzung von Gleitkommaanweisungen wird ermöglicht.
-l	Bei der Übersetzung wird eine Listingdatei erstellt und auf dem Massenspeicher abgelegt. Sie erhält den Dateinamen, anstelle der Endung ".s" wird die Endung ".l" eingesetzt.
-o dateiname	Bei der Übersetzung entsteht eine Objektkodedatei, ihr kann mit "dateiname" ein Name zugewiesen werden. Wird diese Option nicht angegeben, erstellt der Assembler die Datei "a.out"
-p	Bei der Übersetzung wird eine Listingdatei erstellt und auf die Bedienerkonsole ausgegeben.
-u	Nicht definierte Symbole werden als externe Größen behandelt.
-z	Anstelle des WEGA-Objektkodeformates unter dem Dateinamen "a.out" wird das PLZ/ASM-Objektkodeformat unter dem Dateinamen "t.out" erzeugt.

Beispiel: as -lz prog1.s
Die Quellprogrammdatei prog1.s wird übersetzt. Die angegebenen Optionen bewirken die Erstellung einer Listingdatei und die Erstellung einer PLZ/ASM-Objektkodedatei.

6.3.2. Lader

Mit dem Lader (ld) können Maschinenkodeprogramme für die Abarbeitung unter dem Betriebssystem WEGA und für die Abarbeitung der Programme auf anderen Mikrorechnersystemen auf Basis des U8000-Mikroprozessorsystems erstellt werden.

Der Lader wird mit folgender Kommandozeile aufgerufen:

```
ld optionen dateiname
```

Er kann ein oder mehrere Objektkodemodule zu einem ladbaren Maschinenkodeprogramm verarbeiten oder aus mehreren Objektkodemodulen ein Objektkodemodul binden, der in einem folgenden Laderlauf weiterverarbeitet wird. Implizit wird eine Maschinenkodedatei mit der Bezeichnung "a.out" erstellt. Diese Datei ist ausführbar, wenn während des Bindens kein Fehler auftrat.

Die Arbeitsweise des Laders kann durch Optionen spezifiziert werden. Folgende Optionen können in beliebiger Reihenfolge angegeben werden.

-b adr
-bsection adr Mit "adr" kann die Anfangsadresse des Maschinenkodeprogramms bzw. einer mit "section" spezifizierten Sektion vorgegeben werden. Es können die drei Sektionen, Programmkode (t), Daten (d) und Stackbereich (b) festgelegt werden. Die Adresse kann unter Nutzung der in /10/ für die Sprache C vorgegebenen Kennzeichen für Zahlenformate dezimal, hexadezimal (0x) oder oktal (0) angegeben werden. Die angegebene Adresse muß ein Mehrfaches von 256 sein. Ist keine Sektion angegeben, die einen kombinierten Kode- und Datenbereich hat, wird als untere Adresse die Startadresse des kombinierten Kode- und Datenbereichs benutzt. Wenn separate Kode- und Datenbereiche existieren, wird als untere Adresse die Startadresse des Textsegmentes benutzt. Es kann nur eine -t- oder -b-Option je Sektion angegeben werden, Fehler entstehen, wenn die Sektionen sich überlappen oder ein Überlauf entsteht.

-e label Das unter "label" angegebene Symbol wird aus den globalen Namen des Objektkodemoduls als Startadresse definiert. Ist diese Option nicht angegeben, wird die Adresse des Programmkodesegmentes als Startadresse eingesetzt.

-i Bei Angabe dieser Option werden bei der Ausführung des Programms die Programm- und Datenbereiche getrennt. Das Programm wird in das Segment Null, die Daten und der nachfolgende Stackbereich in das Segment Eins eingetragen.

-Nmany Wenn für "many" das Zeichen "l" eingegeben wird, werden in der Symboltabelle durch ld ungekürzte Namen genutzt. Wird für "many" das Zeichen "s" eingegeben, werden nur die ersten 8 Zeichen eines Namens in die Symboltabelle durch ld genutzt.

-o name — Die entstehende Maschinenkodedatei wird unter der Bezeichnung "name" auf dem Massenspeicher abgelegt.

-r — Die Maschinenkodedatei wird mit "relocation bits" erstellt, damit kann sie in nachfolgenden Laderläufen weiterverarbeitet werden. Diese Option schaltet die -t- und -b-Option aus.

-s — Die Maschinenkodedatei wird zur Einsparung von Speicherplatz ohne Symboltabelle und "relocation bits" erstellt.

-t adr
-tsection adr — Mit "adr" kann die Endadresse des Maschinenkodeprogramms oder einer Sektion vorgegeben werden. Mit "section" kann die Endadresse für eine der drei Sektionen Programmkode (t), Daten (d) und Stackbereiche (b) festgelegt werden. Die Adresse kann unter Nutzung der in /10/ für die Sprache C vorgegebenen Kennzeichen für Zahlenformate dezimal, hexadezimal (0x) oder oktal (0) angegeben werden. Die angegebene Adresse muß ein Mehrfaches von 256 sein.

-usymbol — Das "symbol" wird als undefiniertes Symbol in die Symboltabelle eingetragen. Das wird beim vollständigen Binden aus einer Bibliothek genutzt. Zu Beginn des Bindens ist die Symboltabelle leer, mit dem nicht auflösbaren Symbol wird der Lader zum Binden der ersten Routine veranlaßt.

-w — Symbol-Redefinitionswarnungen werden unterdrückt. Sie können beim Suchen in Archiven auftreten, wenn in einem Archiv ein Symbol steht, das schon definiert wurde.

-x — Diese Option bewirkt, daß zur Einsparung von Speicherplatz nur GLOBAL- und EXTERNAL- und keine LOCAL-Größen in die Symboltabelle aufgenommen werden.

Beispiel: ld -s modul1
Aus der Objektkodedatei "modul1" wird die ladbare Maschinenkodedatei "a.out" erzeugt. Zur Einsparung von Speicherplatz werden keine weiteren Informationen in der Datei abgelegt.

7. U8001-Monitor UMON

Der U8001-Monitor "UMON" ermöglicht es, mit relativ geringem Aufwand ein Programm zur Inbetriebnahme von U8000-Mikrorechnersystemen zu erstellen. Mit einem Bildschirm als Ein-/Ausgabegerät können grundsätzliche Monitorkommandos, wie Lesen und Schreiben des Speicherinhaltes und das Setzen eines Unterbrechungspunktes ausgeführt werden.
"UMON" besteht aus einem hardwareunabhängigen Hauptteil, der ohne Veränderung implementiert werden kann. Alle speziellen Hardwarebesonderheiten des U8000-Mikrorechnersystems werden in einem Modul unter der Bezeichnung "UMON.HW" zusammengefaßt.
Die Umstellung des Monitorprogramms für den U8002-Prozessor ist problemlos möglich. Im wesentlichen muß die Programmstatusarea im Modul "UMON.INIT" für diesen Prozessor neu geschrieben werden. Alle Befehle zum Laden der Segmentregister (PSAPSEG, NSPSEG) müssen eliminiert werden.

7.1. Beschreibung des Monitorprogramms

7.1.1. Aufbau des Monitors

Der U8000-Monitor UMON ist in der U8000-Assemblernotation entwickelt worden. Er benötigt einen Programmspeicherbereich von 0 bis %1FFF und einen Arbeitsspeicherbereich von %2000 bis %27FF. Dem Anwender steht der Speicher ab %2800 für seine Programme zur Verfügung.
Der U8000 Monitor besteht aus den folgenden Modulen:

- UMON.INIT
  Im Initialisierungsmodul wird die Initialisierung von Arbeitsbereichen des Monitors und der Ein-/ Ausgabekanäle für das Display vorgenommen.
- UMON.ROUT0
  In diesem Modul stehen im wesentlichen die Routinen:
  FILL (Füllen von Speicherbereichen)
  MOVE (Verschieben von Speicherbereichen)
  DISPLAY (Anzeigen und Verändern von Speicherbereichen).
- UMON.ROUT1
  Die Routinen:
  REGISTER (Setzen von Registerwerten)
  PORT (Lesen und Schreiben des Ein-/Ausgabeadreßraumes)
  SPORT (Lesen und Schreiben des Spezial-Ein-/Ausgabeadreßraumes)
  sind Bestandteil dieses Moduls.
- UMON.ROUT2
  In diesem Modul sind die dynamischen Routinen des Monitors implementiert.
  GO (Ansprung der durch den Befehlszähler vorgegebenen Programmadresse oder der eingegebenen Startadresse)
  BREAK (Setzen eines Unterbrechungspunktes)
- UMON.RAM
  Dieser Modul enthält die Vereinbarung der Arbeitsbereiche im Systemspeicher des U8000-Monitors.

- UMON.HW
  In diesem Modul sind alle hardwareabhängigen Prozeduren enthalten.
  TTY.INIT (Initialisierung der E/A-Prozeduren)
  GETA (Einzelzeicheneingabe)
  PUTA (Einzelzeichenausgabe)
  TTYINT (Interrupt-Serviceroutine)

Initialisierung
Nach Initialisierung der CPU, des RAM-Speichers, des E/A Systems sowie des Programmstatusareas erfolgt die Ausgabe:

```
U8000 Monitor Vrs. 1.0
#
```

Die Ausgabe des '#'-Zeichens zeigt die Arbeitsbereitschaft des Monitors an.

E/A-Organisation
Die E/A-Adressen für den Bedienerdialog werden im hardwareabhängigen Modul des Monitors vereinbart.
Der Anwender muß diese Programme, seiner Hardware entsprechend, modifizieren und kann ohne Veränderung der hardwareunabhängigen Module des Monitorhauptprogramms den U8000-Monitor "UMON" implementieren.
Die Einzelzeicheneingabe erfolgt im Beispielprogramm durch ein V24-Terminal über einen SIO. Der Tastendruck löst einen vektorisierten Interrupt aus. In einer Interrupt-Serviceroutine des U8000 Monitors wird das Zeichen gelesen und in den Eingabepuffer geschrieben.
Die Einzelzeichenausgabe der eingegebenen Zeichen erfolgt vom U8000-Monitor unmittelbar nach Eingang eines Zeichens im Eingabepuffer. Das Zeichen wird nicht interruptgesteuert über den Ausgabekanal des SIO an das Terminal ausgegeben. Die Monitorroutinen werden nach der Identifikation des eingegebenen Kommandos wie ein Unterprogramm aufgerufen. Die Auswertung der Parameter erfolgt in den Monitorroutinen.

7.1.2. Monitorkommandos

Im U8000-Monitor sind die nachfolgend aufgeführten statischen und dynamischen Monitorroutinen implementiert. Die Kommandonamen werden durch Eingabe des 1.Zeichens identifiziert. Die Parameter werden durch Leerzeichen getrennt und, mit einem CR (carriage return) abgeschlossen, eingegeben. Der Monitor verarbeitet nur Großbuchstaben.

DISPLAY (adress)(cnt)((longwords words bytes))(L W B)
Anzeige und Veränderung des Speicherinhaltes.
Die Adreßeingabe ohne "cnt" ermöglicht das Lesen und Schreiben einzelner Adressen. Die Eingabe des Zeichens "-" bewirkt ein Dekrementieren der Adresse.

REGISTER (registername)
Anzeige und Veränderung von Registerinhalten.

BREAK (adress)
Setzen und Löschen von Unterbrechungspunkten. Das Kommando "B" ohne Parameter bewirkt das Löschen des Unterbrechungspunktes.

GO (adress)
Sprung zum letzten PC-Inhalt wenn keine Adresse angegeben ist. Wird eine Adresse angegeben, werden alle Register bis auf den Inhalt des PC zurückgeladen und das Programm auf der angegebenen Adresse gestartet.
Sprung zur angegebenen Adresse.

FILL (adress1)(adress2)(data)
Füllen von Speicherbereichen.

PORT (portadress)(W B)
Lesen und Schreiben von Ein-/Ausgabeports

SPORT (special portadress (W B)
Lesen und Schreiben von Spezial-Ein-/Ausgabeports

MOVE (adress1)(adress2)(n)
Verschieben von Speicherblöcken.

7.1.3. Binden des Monitorprogramms unter dem Betriebssystem UDOS

Besteht das Ziel, den U8000-Monitor "UMON" auf einer U8001-Hardware zu implementieren, ist jeder Modul einzeln mit dem U8000-Assembler zu übersetzen. Der Modul "UMON.HW" ist der jeweiligen Hardware entsprechend zu modifizieren. Wenn die Assemblierung aller Module fehlerfreie Objektmodule ergab, sind die Module zu einem abarbeitungsfähigen Maschinenkodemodul zu binden.
Im UDOS-System /11/ besteht die Möglichkeit, Kommandodateien abzuarbeiten. Mit dem "DO"-Kommando werden UDOS-Kommandos enthaltende ASCII-Dateien abgearbeitet. Damit kann die Eingabe von Kommandos mit umfangreichen Parametern vereinfacht werden. Der U8000-Monitor "UMON" kann mit der nachstehend angegebenen Kommandodatei gebunden werden. In dieser Datei ist der Aufruf der Programme ZLINK und IMAGER mit ihren Parametern angegeben:

```
I
V
ZLINK UMON.INIT UMON.R0 UMON.R1 UMON.R2 UMON.HW UMON.RAM L=UMON.MAP
      N=UMON.OBJ E=ENTRY
IMAGER UMON.OBJ ($=PROM $=1000 RAM) {0 0FFF} O=UMON
B
```

Mit dieser Kommandodatei werden während der Abarbeitung des ZLINK-Programms die angegebenen Objektmodule zu dem Objektmodul "UMON.OBJ" zusammengefaßt. Dabei entsteht die Mapdatei "UMON.MAP" mit Informationen zu Anfangsadressen, Dateilängen und Adressen von Variablen. Die Objektkodedatei "UMON.OBJ" wird anschließend mit dem IMAGER-Progamm zu einer ausführbaren Maschinenkodedatei weiterverarbeitet. Das Segment PROM beginnt auf der Adresse %0000, das Segment RAM auf der Adresse %1000.
Der Maschinenkode des Segmentes PROM wird für den Bereich von %0000 bis %0FFF generiert.
Die Kommandos "I", "V" und "B" sind UDOS-Kommandos /11/, die zur Ausgabe der in der Kommandodatei enthaltenen Kommandos auf das Display und zur Aktualisierung der Diskettenbelegungspläne dienen.

7.2. Listing des Monitorprogramms

```
UMON_INIT MODULE
$SECTION PROM
!*************************************************************!
!*                                                           *!
!*              U 8 0 0 1 - M O N I T O R                    *!
!*                                                           *!
!*************************************************************!

!VEREINBARUNGEN!
GLOBAL
CMDLOP          LABEL
VI_ERR          LABEL

EXTERNAL
TTY_INIT        LABEL
BREAK           LABEL
CMP             LABEL
DISPLAY         LABEL
FILL            LABEL
GO_             LABEL
PORT            LABEL
S_PORT          LABEL
MOVE            LABEL
REGISTER        LABEL
PUTA            LABEL
TMSG            LABEL
CRLF            LABEL
G_LINE          LABEL
STRO_CR_CON     LABEL
TTYINT          LABEL
SC_ENTRY        LABEL
STACK           LABEL
RAMBOT          LABEL
NSP_OFF         WORD
RF_CTR          WORD
SVSTK           LONG
OUTPTR          WORD
SEG_            WORD
PC_             WORD
FCW_            WORD
PS_             WORD
PO_             WORD
PROMT           BYTE
INBFF           ARRAY [%80 BYTE]
CHRDEL          ARRAY [4 BYTE]
PSAREA          ARRAY [%100 BYTE]


INTERNAL
        WORD := %0000
FLG_CW  WORD := %C000
SEG_NUM WORD := %8000
PCOFFS  WORD := ENTRY_

SYSMSG  WORD := 15
        ARRAY [* BYTE] := 'UMON 8001 1.0%0D%00'

GLOBAL ENTRY_ PROCEDURE
ENTRY_
        CLR     R14
        LDCTL   PSAPSEG, R14
        LD      R1, #%4000      !NICHTSEGMENTIERTE ARBEITSWEISE!
        LDCTL   FCW, R1
        LD      R1, #%9E00      !REFRESH ANPASSUNG!
        LDCTL   REFRESH, R1
```

```
        LD      RF_CTR, R1

!%200 WORD IM RAM AB RAMBOT LOESCHEN!
        LD      R4, #RAMBOT
        LD      R2, #RAMBOT+2
        LD      @R4, #0
        LD      R1, #%200
        LDIR    @R2, @R4, R1

!PUFFER MIT SPACE INITIALISIEREN!
        LD      INBFF, #%2020
        LD      R4, #INBFF
        LD      R2, #INBFF+2
        LD      R1, #%80
        LDIR    @R2, @R4, R1
        LD      R4, #VAR_LISTE_PROM
        LD      R2, #CHRDEL
        LDK     R1, #%04
        LDIR    @R2, @R4, R1

!PROGRAMMSTATUSAREA INITIALISIEREN!
        LD      R4, #PS_AREAPROM
        LD      R2, #PSAREA
        LD      R1, #%100
        LDIR    @R2, @R4, R1

!STACK_BEREICH INITIALISIEREN!
        LD      R14, #%8000     !NORMALSTACKPOINTER_SEGMENT!
        LD      R15, #NSP_OFF   !NORMALSTACKPOINTER_OFFSET!
        LDL     SVSTK, RR14
        LD      R15, #STACK
        LD      R1, #PSAREA
        LDCTL   PSAP, R1
        LD      PS_, R14
        LD      SEG_, R14
        LD      PO_, R1
        LD      PC_, #%3000     !PC IM REGISTERBEREICH INITIALISIEREN!
        LD      FCW_, #%C000    !FCW IM REGISTERBEREICH INITIALISIEREN!
        CALL    TTY_INIT        !DISPLAYKANAL INITIALISIEREN!

        CALL    CRLF
        LD      R2, #SYSMSG
ERR_LP: CALL    TMSG            !AUSGABE DER SYSTEMMELD. "UMON 8001 1.0"!

        LD      R1, #'# '
        LD      PROMT, R1

CMDLOP: LD      R15, #STACK     !KOMMANDOEINGABESCHLEIFE!
        PUSH    @R15, #CMDLOP   !RUECKKEHRADRESSE F. MONITORROUTINEN LAD.!
        EI      VI
        LDB     RL0, PROMT
        CALL    PUTA            !PROMT AUSGEBEN!
        CALL    G_LINE          !LESEN DER 1.EINGEGEBENEN ZEILE!

!ERSTES ZEICHEN DER KOMMANDOZEILE AUSWERTEN.!
        LD      R1, #%08
        LDA     R2, CMD_LISTE - 1 (R1)
        CPDRB   RL0, @R2, R1, Z !KOMMANDO SUCHEN!
        JR      NZ, ERROR       !SPRUNG WENN KOMMANDO UNBEKANNT!
        SLL     R1, #%01
        LD      R2, CMD_ADR(R1)
        JP      @R2             !ABSPRUNG IN DIE MONITORROUTINE!
    END ENTRY_

  INTERNAL
CMD_LISTE ARRAY [* BYTE] := 'BDFGPSMR%00'

CMD_ADR ARRAY [* WORD] := [BREAK DISPLAY FILL GO_ PORT S_PORT MOVE
```

```
                          REGISTER]

GLOBAL ERROR PROCEDURE
ENTRY
        LDB     RL0, #'?'
        CALL    PUTA            !'?' AUSGABE!
        LD      OUTPTR, #0      !AUSGABEPOINTER INITIALISIEREN!
        JR      CMDLOP
    END ERROR

INTERNAL
!VARIABLENLISTE: CHRDEL/LINDEL/N_CNT/%7F00/!
VAR_LISTE_PROM
        ARRAY [3 WORD] := [%087F %0000 %7F00 ]


INTERNAL VI_ERR PROCEDURE       !EINTRITTSPUNKT FUER VI-FEHLER!
      ENTRY
        LD      R2, #VE_ERR
        JR      ERR_LP
        END VI_ERR

VE_ERR: WORD := 35
        ARRAY [* BYTE] := 'UNINITIALISIERTER NVI, VI ODER TRAP'

!INITIALISIERUNGSFELD FUER PROGRAMMSTATUSAREA!
!SEGMENTIERTE VERSION!
PS_AREAPROM:
!                           ID     FCW    PCSEG PCOFF!
        ARRAY [4 WORD] := [%0000 %0000 %0000 %0000]    !RESERVIERT!
        ARRAY [4 WORD] := [%0000 %4000 %8000 VI_ERR]   !UNIMPLEM. INSTR.!
        ARRAY [4 WORD] := [%0000 %4000 %8000 VI_ERR]   !PRIVILEGED INSTR.!
        ARRAY [4 WORD] := [%0000 %C000 %8000 SC_ENTRY] !SYST. CALL INSTR.!
        ARRAY [4 WORD] := [%0000 %4000 %8000 VI_ERR]   !SEGMENT TRAP!
        ARRAY [4 WORD] := [%0000 %4000 %8000 VI_ERR]   !NONMASKABLE INT.!
        ARRAY [4 WORD] := [%0000 %5000 %8000 VI_ERR]   !NONVECTORED INT.!
VEC_FCW ARRAY [2 WORD] := [%0000 %4000]
!VECTORED INTERRUPT JUMP TABLE!
!                           SEG    OFFSET!
        ARRAY [2 WORD] := [%8000 VI_ERR]
        ARRAY [2 WORD] := [%8000 VI_ERR]
        ARRAY [2 WORD] := [%8000 TTYINT]
        ARRAY [2 WORD] := [%8000 VI_ERR]
END UMON_INIT
```

```
UMON_ROUTO MODULE

$SECTION PROM
GLOBAL
OUT_1ASC        LABEL
NEW_R           LABEL
NSIGNW          LABEL
BTOH8           LABEL

EXTERNAL
ERROR           LABEL
CMDLOP          LABEL
PUTA            LABEL
RD_DATA         LABEL
N_CT            LABEL
KONV            BYTE
OUTBFF          WORD
PTY_BF          BYTE
MAXLEN          WORD
INPTR           WORD
OUTPTR          WORD
PTYFLG          WORD
SV_R            BYTE
SVSTK           WORD
SEG_            WORD
PC_             WORD
FCW_            WORD
FLAG            BYTE
NULLCT          WORD
INBFF           ARRAY   [%80 BYTE]

CONSTANT
LF      :=%0A
CR      :=%0D
!SYSTEM_CALL'S!
SEG_V := %01
NSEG_V := %02

!*******************!
GLOBAL FILL PROCEDURE
!*******************!
ENTRY
        CALR    RD_PAR          !WEITER PARAMETER LESEN!
        JP      C, ERROR        !KEINE PARAMETER VORHANDEN!
        RES     R7, #%0         !GERADE ADRESSE EINSTELLEN!
        SUB     R5, R7          !LAENGE DES ZU BEREICHES BERECHNEN!
        JP      C, ERROR        !SPRUNG WENN ADRESSE1 > ADRESSE2!
        SRL     R5, #1          !WORTANZAHL BERECHNEN!
        INC     R5, #1
        SC      #SEG_V          !SEGMENTIERTEN MODE EINSTELLEN!
F_LOOP:
        LD      @R6, R3         !DATENWERT EINTRAGEN!
        INC     R7, #%02
        DJNZ    R5, F_LOOP
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!
        RET
    END FILL

!*******************!
GLOBAL MOVE PROCEDURE
!*******************!
ENTRY
        CALR    RD_PAR          !WEITER PARAMETER LESEN!
        JP      C, ERROR        !KEINE PARAMETER VORHANDEN!
        CP      R4, R6          !VERGLEICH DER SEGMENTNUMMERN!
        JR      NZ, SRC_GROE
        CP      R7, R5          !VERGLEICH DER OFFSET-PARAMETER!
        JR      C, SRC_KLEI
```

```
SRC_GROE:'
        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        LDIRB   @R4, @R6, R3
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!
        RET
SRC_KLEI:
        ADD     R7, R3          !BLOCKTRANSFER MIT ADRESSENDEKREMENT!
        DEC     R7, #1
        ADD     R5, R3
        DEC     R5, #1
        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        LDDRB   @R4, @R6, R3
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!
        RET
    END MOVE

LD_3C PROCEDURE                 !OEFFNENDEN WINKEL '<' IN OUTBF SCHREIBEN!
ENTRY
        LD      R1, #'< '
LD1_CHR:
        CALR    OUT_2ASC        !ZWEI ZEICHEN EINTRAGEN!
        DEC     OUTPTR, #1      !AUSGABEPOINTER DEKREMENTIEREN!
        RET
    END LD_3C

 GLOBAL
WRT_SNR PROCEDURE               !'<SEGMENTNUMMER>' IN OUTBF SCHREIBEN!
ENTRY
        CALR    LD_3C
WRT_SEG:
        RESB    RL5, #%07
        CALR    BTOH8           !SEGMENTNUMMER KONVERTIEREN!
        LD      R1, #'> '
        JR      LD1_CHR
    END WRT_SNR

  INTERNAL
    OUT_1ASC PROCEDURE          !EIN ZEICHEN IN OUTBF SCHREIBEN!
      ENTRY
        LD      R2, OUTPTR      !AUSGABEPOINTER LESEN!
        LDB     OUTBFF(R2), RL1 !ZEICHEN EINTRAGEN!
        INC     OUTPTR, #1      !AUSGABEPOINTER AKTUALISIEREN!
        RET
    END OUT_1ASC

!*********************!
GLOBAL DISPLAY PROCEDURE
!*********************!
ENTRY
        RESB    FLAG, #1        !MERKER INITIALISIEREN!
        CALR    NSIGN           !INPTR AUF DAS NAECHSTE ZEICHEN STELLEN!
        LD      R7, #%0202      !MERKER: STANDARD IST WORTAUSGABE!
        LD      R10, SEG_
        JR      NZ, YES_PARAM
        LD      R1, R3
        JR      NO_PARAM        !KEIN PARAMETER EINGEGEBEN!
YES_PARAM:
        CALR    RD_ADR          !ADRESSE IN RR2 KONVERTIEREN!
        JR      NZ, NO_CR       !SPRUNG WENN KEIN ENDEZEICHEN ERKANNT!

        SETB    FLAG, #1        !MERKER 'DATEN SCHREIBEN' SETZEN!
NO_CR:  LDL     RR10, RR2       !STARTADRESSE IN RR10 LADEN!
        BITB    FLAG, #1        !MERKER 'DATEN SCHREIBEN' ABFRAGEN!
        JR      NZ, D_WRITE     !DATEN SCHREIBEN!

        CALR    TST_WBL         !TEST: 'DATEN SCHREIBEN MIT ANGABE 'B,W,L!
        JR      Z, D_WRITE      !DATEN SCHREIBEN!
```

```
        CALR    RD_HEXZ         !LAENGE DES ANZEIGEBEREICHES IN R3 LADEN!
        LD      R13, R3         !R13 = AUSGABEZAEHLER!
        JR      Z, D_OUT        !SPRUNG WENN ALLE PARAMETER GELESEN SIND!

        CALR    TST_WBL         !AUSGEBEFORM 'B,W,L' LESEN!
        JP      NZ, ERROR

D_OUT:  TEST    R13             !LAENGE DES ANZEIGEBEREICHES = NULL ? !
        JP      Z, ERROR

D_OUT_LOOP:
        CALR    LD_3C           !'<' IN DEN AUSGABEPUFFER EINTRAGEN!
        LD      R5, R10         !SEGMENTNUMMER DER ANFANGSADRESSE LADEN!
        LDB     RL5, RH5
        CALR    WRT_SEG         !SEGMENTNUMMER AUSGEBEN!
        LD      R5, R11         !OFFSET DER ANFANGSADRESSE LADEN!
        CALR    BTOH16          !ANFANGSADRESSE IN OUTBF LADEN!
        INC     OUTPTR, #1      !AUSGABEPOINTER ERHOEHEN!
        LD      R6, #16         !ANZAHL DER AUSZUGEBENDEN BYTEADRESSEN!
        LD      R8, #58         !AUF POSITION 58 '*' FUER DEN ANFANG DER!
                                !ASCII-DARSTELLUNG EINTRAGEN!
        LDB     OUTBFF(R8), #'*'
        INC     R8, #1
NXT_BYTE:
        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        LDB     RL5, @R10       !BYTE AUS DEM SPEICHER LESEN!
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!
        LDB     OUTBFF(R8), RL5 !BYTE IN ASCII-SPALTE LADEN!
        CPB     RL5, #' '       !IST DAS BYTE DARSTELLBAR ? !
        JR      C, NO_ASCII
        CPB     RL5, #%7F
        JR      C, YES_ASCII
NO_ASCII:
        LDB     OUTBFF(R8), #'.' !KEIN ASCII-ZEICHEN, MIT '.' UEBERSCHR.!
YES_ASCII:
        INC     R8, #1          !POINTER IN DER ASCII-SPALTE ERHOEHEN!
        LDB     OUTBFF(R8), #'*' !'*' AUF DIE NAECHSTE POSITION SCHREIBEN!
        CALR    BTOH8           !BYTE KONVERTIEREN UND IN OUTBFF SCHREIB.!
        INC     R11, #1         !NAECHSTES BYTE ADRESSIEREN!
        DEC     R6, #1          !BYTEZAEHLER DEKREMENTIEREN!
        JR      NZ, ROW_BUSY    !WEITER AUSGEBEN!

!ZEILE AN DAS DISPLAY AUSGEBEN!
        INC     R8, #1          !AUSGABEPOINTER ERHOEHEN!
        LD      OUTPTR, R8
        CALR    STRO_CR_CON     !AUSGABE!
        LDB     RL7, RH7        !MERKER W,B,L NEUSETZEN!
        DEC     R13, #1         !AUSGABEBEREICH DARGESTELLT!
        JR      NZ, D_OUT_LOOP
        RET
ROW_BUSY:
        DBJNZ   RL7, NXT_BYTE   !NAECHSTES BYTE AUS DEM SPEICHER LESEN!
        INC     OUTPTR, #1
        LDB     RL7, RH7        !MERKER W,B,L NEUSETZEN!
        DEC     R13, #1         !LAENGE DEKREMENTIEREN!
        JR      NZ, NXT_BYTE    !NAECHSTES BYTE AUS DEM SPEICHER LESEN!

!ENDE DES ANZEIGEBEREICHES ERREICHT: ZEILE AN DAS DISPLAY AUSGEBEN!
        INC     R8, #1          !AUSGABEPOINTER ERHOEHEN!
        LD      OUTPTR, R8
        CALR    STRO_CR_CON
        RET

D_WRITE:                        !DATEN SCHREIBEN!
        CPB     RL7, #1         !MERKER W,B,L ABFRAGEN!
        JR      Z, NO_PARAM
        RES     R11, #%0        !GERADE ADRESSE FUER WORD ODER LONG AUSG.!
NO_PARAM:
```

```
        LDL     RR8, RR10
        CALR    LD_3C           !'<' IN DEN AUSGABEPUFFER EINTRAGEN!
        LD      R5, R10         !SEGMENTNUMMER DER ANFANGSADRESSE LADEN!
        LDB     RL5, RH5
        CALR    WRT_SEG         !SEGMENTNUMMER AUSGEBEN!
        LD      R5, R11         !OFFSET DER ANFANGSADRESSE LADEN!
        CALR    BTOH16          !ANFANGSADRESSE IN OUTBFF LADEN!
        INC     OUTPTR, #1
D_LOOP: SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        LDB     RL5, @R10       !BYTE AUS DEM SPEICHER LESEN!
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!
        CALR    BTOH8           !BYTE KONVERTIEREN UND IN OUTBFF LADEN!
        INC     R11, #1         !ADRESSE ERHOEHEN!
        DBJNZ   RL7, D_LOOP     !IN ABHAENGIGKEIT VON DER DARSTELLUNGSART!
                                !B,W,L NAECHSTES BYTE LESEN!
        INC     OUTPTR, #1
        LDB     RL7, RH7        !MERKER B,W,L NEUSETZEN!
        CALR    NEW_R           !ZEILE AUSGEBEN: NEUEN DATENWERT EINLESEN!
        JR      NC, N_END       !SPRUNG WENN KEIN ENDEZEICHEN ERKANNT!

        CPB     RL0, #'Q'       !QUIT EINGEGEBEN ? !
        RET     Z               !RUECKSPRUNG!
        PUSH    @R15, R7        !'-' EINGEGEBEN, ADRESSE DEKREMENTIEREN!
        CLRB    RH7
        SUB     R11, R7
        SUB     R11, R7
        POP     R7, @R15
        JR      NO_PARAM
N_END:  JR      Z, NO_PARAM     !'CARRIAGE RETURN' EINGEGEBEN ? !
        DECB    RL7, #1
        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        JR      NZ,WRT_W_L      !SPRUNG WENN WORD ODER LONG AUSGABE!

        LDB     @R8, RL3        !NEUES BYTE IN DEN SPEICHER EINTRAGEN!
        JR      WRT_END
WRT_W_L:
        DECB    RL7, #1
        JR      NZ, WRT_L
        LD      @R8, R3         !NEUES WORT IN DEN SPEICHER EINTRAGEN!
        JR      WRT_END

WRT_L:  LDL     @R8, RR2        !NEUES LONGWORT IN DEN SPEICHER EINTRAGEN!
WRT_END:
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!
        LDB     RL7, RH7        !MERKER NEUSETZEN!
        JR      NO_PARAM
    END DISPLAY

INTERNAL TST_WBL PROCEDURE      !TEST: WORD / BYTE / LONG !
ENTRY
        LD      R7, #%0202      !WORTMERKER ALS STANDARD SETZEN!
        CPB     RL0, #CR
        RET     Z
        CPB     RL0, #'W'
        RET     Z
        LD      R7, #%0101      !BYTEMERKER!
        CPB     RL0, #'B'
        RET     Z
        LD      R7, #%0404      !LONGWORTMERKER!
        CPB     RL0, #'L'
        RET     Z
        LD      R7, #%0202
        RET
    END TST_WBL

GLOBAL NEW_ROW PROCEDURE        !DATEN KONVERTIEREN UND EINLESEN!
ENTRY
        CALR    BTOH16          !DATEN KONVERTIERN UND IN OUTBFF LADEN!
```

```
        INC     OUTPTR, #1
NEW_R:  CALR    STRO_CON        !DATEN AUSGEBEN!
        CALL    RD_DATA         !NEUEN DATENWERT EINLESEN!
        RET
    END NEW_ROW

GLOBAL OUT_2ASC PROCEDURE       !ZWEI ASCII-ZEICHEN IN OUTBFF LADEN!
      ENTRY
        PUSH    @R15, R4
        LD      R4, OUTPTR
        LDB     OUTBFF(R4), RH1
        INC     R4, #1
        LDB     OUTBFF(R4), RL1
        INC     OUTPTR, #%02
        POP     R4, @R15
        RET
    END OUT_2ASC

GLOBAL TMSG PROCEDURE           !MESSAGE_TRANSFER ZUM OUTBFF!
ENTRY
        LD      R0, @R2         !R2 ZEIGT AUF DIE LAENGE DES TEXTES!
        CP      R0, #%0         !LAENGE = 0 ? !
        RET     Z
        LD      OUTPTR, R0
        INC     R2, #%02        !ADRESSE DES 1. ZEICHENS!
        LD      R4, #OUTBFF     !OUTBFF ADRESSE LADEN!
        LDIRB   @R4, @R2, R0    !MESSAGE TRANSFERIEREN!
        CALL    STRO_CON        !SPRUNG ZUM AUSGABEPROGRAMM!
        RET
        END TMSG

RD_PAR PROCEDURE                !PARAMETER ALS HEXADEZIMALZAHL!
ENTRY
        CALR    NSIGN           !1. ASCII-ZEICHEN IN RL0!
        CALR    RD_ADR          !1.PARAMETER (ADRESSE) IN RR2 LADEN!
        JR      Z, PAR_ERR
        LDL     RR6, RR2        !1.PARAMETER IN RR6 LADEN!
        CALR    RD_ADR          !2.PARAMETER (ADRESSE/LAENGE)IN RR2 LADEN!
        JR      Z, PAR_ERR
        LDL     RR4, RR2        !2.PARAMETER IN RR4 LADEN!
        CALR    RD_HEXZ         !3.PARAMETER IN R3 LADEN!
        RET     NC
PAR_ERR:
        SETFLG  C               !FEHLERKENNZEICHEN SETZEN!
        RET
    END RD_PAR

BTOH16 PROCEDURE                !16 BIT BINAER IN HEX-KONVERTIERUNG!
ENTRY
        LDB     RH0, RH5
        CALR    BTH
BTOH8:  LDB     RH0, RL5        !8 BIT BINAER IN HEX-KONVERTIERUNG!
BTH:    RLDB    RL0, RH0
        CALR    BTOH            !4 BIT BINAER IN HEX-KONVERTIERUNG!
        RLDB    RL0, RH0
BTOH:   PUSH    @R15, R1        !4 BIT BINAER IN HEX-KONVERTIERUNG!
        ANDB    RL0, #%0F       !UNTERE 4 BIT MASKIEREN!
        LDB     RL1, KONV
        ADDB    RL1, RL0
        LDB     KONV, RL1
        CPB     RL0, #%0A
        JR      C, BTOH1
        ADDB    RL0, #%07
BTOH1:  ADDB    RL0, #%30
        PUSH    @R15, R6
        LD      R6, OUTPTR
        LDB     OUTBFF(R6), RL0 !ASCII-ZAHL IN OUTBFF EINTRAGEN!
        INC     OUTPTR, #1
```

```
        POP     R6, @R15
        POP     R1, @R15
        RET
    END BTOH16

STRO_CR_CON PROCEDURE           !STRING-OUT MIT (CR) ZUR CONSOLE!
ENTRY
        LD      R3, OUTPTR
        LDB     OUTBFF(R3), #CR !CARRIAGE RETURN IN OUTBFF EINTRAGEN!
        INC     OUTPTR, #1
    END STRO_CR_CON

STRO_CON PROCEDURE              !STRING-OUT OHNE (CR) ZUR CONSOLE!
ENTRY
        CLR     R3              !ZEICHENZAEHLER LOESCHEN!
TX_1:   LDB     RL0, OUTBFF(R3)
        INC     R3, #1
        CALL    PUTA
        JR      Z, TX_2         !SPRUNG WENN CR AUSGEGEBEN WURDE!
        CP      R3, OUTPTR      !ALLE ZEICHEN AUSGEGEBEN ? !
        JR      C, TX_1         !SPRUNG WENN PUFFER NICHT LEER!
        JR      BF_EMPTY
TX_2:   LDB     RL0, #LF        !LINE FEED NACH TEXT SENDEN!
        CALL    PUTA
        CALL    N_CT
BF_EMPTY:
        LD      OUTPTR, #0      !OUTPTR RUECKSETZEN!
        LD      R0, #%0040
        LD      OUTBFF, #'  '
        LD      R2, #OUTBFF     !OUTBFF LOESCHEN!
        LD      R4, #OUTBFF + 2
        LDIR    @R4, @R2, R0
        RET
END STRO_CON

HTOB PROCEDURE                  !KONVERTIERUNG HEX IN BINAER!
ENTRY
        CPB     RL0, #'0'
        RET     C               !FEHLER := KEINE ZAHL!
        CPB     RL0, #'9' + 1
        JR      C, HTOB1        !SPRUNG WENN ZAHL!
        CPB     RL0, #'A'
        RET     C               !FEHLER := KEIN BUCHSTABE!
        CPB     RL0, #'F' + 1
        JR      NC, HTOB_ERR    !FEHLER := KEIN BUCHSTABE A...F!
        SUBB    RL0, #%07
HTOB1:  ANDB    RL0, #%0F       !FEHLERFREI!
        RESFLG  C
        RET
HTOB_ERR:
        SETFLG  C               !FEHLER := KEINE HEXADEZIMALZAHL!
        RET
    END HTOB

RD_ADR PROCEDURE                !ADRESSE AUS DEM EINGABEPUFFER LESEN!
ENTRY
        CALR    EKSGNR          !SEGMENTNUMMER EINLESEN!
        CALR    RD_HEXZ         !OFFSET IN R3 LESEN!
        RET
    END RD_ADR

EKSGNR PROCEDURE                !EINGABEKONVERTIERUNG DER SEGMENTNR.!
ENTRY
        LD      R2, SEG
        SET     R2, #%0F
        RESB    FLAG, #%0
        CPB     RL0, #CR
        RET     Z
```

```
        CPB     RL0, #'<'       !SEGMENTKENNZEICHEN ANFANG!
        RET     NZ              !RUECKSPRUNG: KEINE SEGMENTNUMMER!
        CLR     R3
EKSN_N:
        CALR    RDCHR           !SEGMENTNUMMER KONVERTIEREN!
        JP      Z, ERROR
        CALR    HTOB
        JR      C, EKSN_NHZ
        RLDB    RL0, RL3
        JR      EKSN_N
EKSN_NHZ:
        CPB     RL0, #'>'       !SEGMENTKENNZEICHEN ENDE!
        JP      NZ, ERROR
        CALR    NSIGNW
        LDB     RH2, RL3
        CLRB    RL2
        SET     R2, #%0F
        SETB    FLAG, #%0
        RET
    END EKSGNR

RD_HEXZ PROCEDURE               !HEXADEZIMALZAHL AUS DEM EINGABEPUFFER!
      ENTRY                     !LESEN!
        CLR     R3
        CPB     RL0, #CR        !ENDEKENNUNG ERREICHT ?!
        SETFLG  C               !FEHLERKENNZEICHEN LADEN!
        RET     Z
RD_HX1: CALR    HTOB            !HEX IN BINAER-KONVERTIERUNG!
        JP      C, ERROR
        RLDB    RL0, RL3
        RLDB    RL0, RH3
        CALR    RDCHR           !NAECHSTES ZEICHEN LESEN!
        RET     Z               !RUECKSPRUNG WENN ENDEKENNUNG ERREICHT!
        CPB     RL0, #' '
        JR      NZ, RD_HX1
        CALR    NSIGNW
        RESFLG  C
        RET
    END RD_HEXZ

GLOBAL NSIGN PROCEDURE          !NAECHSTES ZEICHEN NACH SPACE EINLESEN!
ENTRY
        CALR    RDCHR
        RET     Z               !RUECKSPRUNG WENN ENDEKENNUNG (CR)!
        CPB     RL0, #' '
        JR      NZ, NSIGN       !1. SPACE NACH DEM KOMMANDO SUCHEN!

NSIGNW: CALR    RDCHR           !1. ZEICHEN DER PARAMETERFOLGE LESEN!
        RET     Z
        CPB     RL0, #' '
        JR      Z, NSIGNW
        RET
    END NSIGN

RDCHR PROCEDURE                 !EIN ASCII-ZEICHEN AUS INBFF LESEN!
      ENTRY
        PUSH    @R15, R2
        LD      R2, INPTR
        LDB     RL0, INBFF(R2)
        INC     INPTR, #1       !INPTR := INPTR + 1!
        CPB     RL0, #CR
        POP     R2, @R15
        RET
    END RDCHR

END UMON_ROUTO
```

```
UMON_ROUT1 MODULE
$SECTION PROM

GLOBAL
S_PORT          LABEL

EXTERNAL
PUTA            LABEL
GETA            LABEL
ERROR           LABEL
NEW_ROW         LABEL
NEW_R           LABEL
OUT_2ASC        LABEL
BTOH16          LABEL
BTOH8           LABEL
STRO_CR_CON     LABEL
HTOB            LABEL
NSIGN           LABEL
NSIGNW          LABEL
RDCHR           LABEL
RD_HEXZ         LABEL
STRO_CON        LABEL
PORTBF          WORD
NULLCT          WORD
OUTBFF          WORD
MAXLEN          WORD
INPTR           WORD
OUTPTR          WORD
SV_R            BYTE
SVSTK           WORD
RF_CTR          WORD
CHRDEL          BYTE
LINDEL          BYTE
INBFF           ARRAY [%80 BYTE]

CONSTANT
BS      := %08
LF      := %0A
CR      := %0D

INTERNAL
R_BEZ     ARRAY [* BYTE] := '0 1 2 3 4 5 6 7 8 9 101112131415'
R_BEZ_B   ARRAY [* BYTE] := 'SGPCFCRFNSNOPSPO'
R_LH ARRAY [* BYTE] := 'RLRH'


GLOBAL
!*********************************!
PORT PROCEDURE  !NORMAL EIN-/AUSGABE!
!*********************************!
ENTRY
        LD      R10, #0         !INDEX FUER NORMAL I/O!
        JR      S_PORT + 4
!*********************************!
S_PORT:         !SPEZIAL EIN-/AUSGABE!
!*********************************!

        LD      R10, #24        !INDEX FUER SPEZIAL I/O!
        CALL    NSIGN           !EINGABEZEIGER AUF NAECHSTES ZEICHEN!
                                !STELLEN!
        JP      Z, ERROR        !PARAMETERFEHLER!
        CALL    RD_HEXZ         !E/A-ADRESSE EINLESEN!
        LD      R7, R3          !ADRESSE FUER E/A-BEFEHLE!
        LD      R5, R3          !E/A-ADRESSE ZUR KONVERTIERUNG UMLADEN!
        CPB     RL0, #'B'       !BYTEWEISE E/A ? !
        JR      Z, ADR
        CPB     RL0, #'W'       !WORTWEISE E/A ? !
        JP      NZ, ERROR       !PARAMETERFEHLER!
```

```
        ADD     R10, #16        !INDEX KORRIGIEREN!
ADR:    CALL    BTOH16          !E/A-ADRESSE KONVERTIEREN UND IN !
                                !E/A-PUFFER EINTRAGEN!
        INC     OUTPTR          !AUSGABEZEIGER WEITERSTELLEN!
        LD      R9, OUTPTR      !AUSGABEZEIGER FUER NAECHSTE ZEILE!
                                !SPEICHERN!

INPUT:  LD      R6, #PORTBF     !PUFFERADR.  (SRC ODER DST) IM RAM LADEN!
        CALL    EA_ (R10)       !EA-PORT EINLESEN!
        BIT     R10, #3         !1=WORT / 0= BYTE!
        JR      Z, BYTE_
        LD      R5, @R6         !WORT VOM PUFFER LESEN!
        CALL    BTOH16          !WORT KONVERTIEREN UND IN E/A-PUFFER LAD.!
        JR      INCP

BYTE_:  LDB     RL5, @R6        !BYTE VOM PUFFER LESEN!
        CALL    BTOH8           !BYTE KONVERTIEREN UND IN E/A-PUFFER LAD.!
INCP:   INC     OUTPTR          !AUSGABEZEIGER WEITERSTELLEN!
        CALL    STRO_CON        !E/A-ADRESSE UND INHALT AUSGEBEN!
        CALL    RD_DATA         !NEUEN DATENWERT ZUR AUSGABE EINLESEN!
        RET     Z               !RUECKSPRUNG WENN NUR 'CR' EINGEGEBEN!
                                !WURDE = PORT LESEN!
        RET     C               !RUECKSPRUNG WENN 'Q' EINGEGEBEN!

        LD      @R6, R3         !NEUEN WERT AUF PUFFER LADEN!
        LD      R5, R3          !EINGEGEBENEN DATENWERT UMLADEN!
        INC     R10, #8         !INDEX AUF AUSGABE STELLEN!
        CALL    EA_ (R10)       !DATENWERT AN DEN PORT SCHREIBEN!
        DEC     R10, #8         !INDEX ZURUECKSTELLEN!
        LD      OUTPTR, R9      !AUSGABEZEIGER INITIALISIEREN!
        JR      INPUT           !SCHLEIFE!

EA_:    INC     R6              !PUFFERADRESSE ZEIGT AUF LOW-BYTE!
        INIB    @R6, @R7, R8    !NORMAL-EIN-/AUSGABEBEFEHLE!
        RET
        INC     R6              !PUFFERADRESSE ZEIGT AUF LOW-BYTE!
        OUTIB   @R7, @R6, R8
        RET
        NOP
        INI     @R6, @R7, R8    !NORMAL-EIN-/AUSGABEBEFEHLE!
        RET
        NOP
        OUTI    @R7, @R6, R8
        RET
SEA_:   INC     R6              !PUFFERADRESSE ZEIGT AUF LOW-BYTE!
        SINIB   @R6, @R7, R8    !NORMAL-EIN-/AUSGABEBEFEHLE!
        RET
        INC     R6              !PUFFERADRESSE ZEIGT AUF LOW-BYTE!
        SOUTIB  @R7, @R6, R8
        RET
        NOP
        SINI    @R6, @R7, R8    !NORMAL-EIN-/AUSGABEBEFEHLE!
        RET
        NOP
        SOUTI   @R7, @R6, R8
        RET
END PORT
!**********************!
GLOBAL REGISTER PROCEDURE       !REGISTER_ROUTINE!
!**********************!
ENTRY
        CALL    NSIGN           !NAECHSTES ZEICHEN AUS INBFF LESEN!
        JP      Z, ALL_R        !"CR" ERKANNT: ALLE REGISTER AUSGEBEN!

        CPB     RL0, #'R'       !REGISTERBEZEICHNUNG PRUEFEN!
        JR      NZ, NO_RRF      !REGISTER: SG, PC, FC, NS, NO, PS ODER PO!
        CALL    RDCHR           !REGISTER "RR" ODER "RF" ?!
        JP      Z, ERROR
```

```
        CPB     RL0, #'F'       !REGISTER: "RF" ?!
        JR      NZ, NO_RRF
        LDB     RH1, #'R'       !1.ZEICHEN DER BEZEICHNUNG EINTRAGEN!
        JR      RG6
NO_RRF: ACLR    R6              !MERKER R6 = 0 = LOW-BYTE!
        CPB     RL0, #'L'       !BYTEREGISTER LOW-BYTE ?!
        JR      Z, R_L
        INC     R6, #%02        !MERKER R6 = 2 = HIGH-BYTE!
        CPB     RL0, #'H'       !BYTEREGISTER HIGH-BYTE ?!
        JR      Z, R_L
        INC     R6, #%04        !MERKER R6 = 6 = NORMALREGISTER, SG..PO!
        CPB     RL0, #'R'
        JR      NZ, NO_R
        DEC     R6, #%02        !MERKER R6 = 4 = DOPPELREGISTER!
R_L:    CALL    RDCHR           !1. ZEICHEN DER REGISTERB. IN RH0  LADEN!
        JP      Z, ERROR

NO_R:   LDB     RH1, RL0        !ZEICHEN RETTEN!
        CALL    RDCHR           !2. ZEICHEN DER REGISTERBEZ. IN RH0 LAD.!
        JR      NZ, NO_R_CR     !SPRUNG WENN KEIN "CR"!
        LDB     RL0, #' '       !SPACE EINTRAGEN WENN EINSTELLIG!
NO_R_CR:
        LDB     RL1, RL0
        LD      R7, #%0018
        LD      R8, #R_BEZ + 46
        CPDR    R1, @R8, R7, Z  !REGISTERNAMEN SUCHEN!
        JP      NZ, ERROR       !REGISTERNAMEN NICHT GEFUNDEN!

        INC     R8, #%02        !R8 ZEIGT AUF DIE REGISTERKENNUNG!
        ADD     R7, R7          !RELATIVADRESSE FUER REGISTERBEZ. IM RAM!
        CPB     RL6, #%04       !MERKER: DOPPELREGISTER PRUEFEN!
        JR      C, R_BYTE
        JR      Z, R_DOPP

!EINFACHREGISTER BZW. REGISTER: SG, PC, FC, NS, NO, PS ODER PO!
R_EINF: LD      R1, #'R '       !"R " IN DEN AUSGABEPUFFER EINTRAGEN!
        CALL    OUT_2ASC
        DEC     OUTPTR, #1
        LD      R1, @R8         !REGISTERKENNUNG AUS R_BEZ IN OUTBFF LAD.!
        CALL    OUT_2ASC
        INC     OUTPTR, #1
        LD      R5, SV_R(R7)    !REGISTERINH. AUS DEM RAM IN OUTBFF LAD.!
        CALL    NEW_ROW         !ZEILE AUSG. UND NEUEN WERT IN R3 EINLES.!
        RET     C               !"Q" ERKANNT: ENDE DER ROUTINE!
        JR      Z, R_EINF_CR    !"CR" ERKANNT: REGISTERWERT BLEIBT BEST.!
        LD      R2, @R8         !REGISTERKENNUNG LADEN!
        CP      R2, #'PC'       !"R PC" AUSGEGEBEN ?!
        JR      NZ, NO_PC
        RES     R3, #%0         !GERADE ADRESSE SICHERN!
        JR      SG16

NO_PC:  CP      R2, # 'RF'      !"R RF" AUSGEGEBEN ?!
        JR      NZ, NO_RF
        LDCTL   REFRESH, R3     !REFRESHREGISTER DER CPU LADEN!

NO_RF:  CP      R2, #'SG'       !"R SG" AUSGEGEBEN ?!
        JR      NZ, NO_SG
        CPB     RH3, #%0        !16-BIT-SEGMENTNUMMER EINGEGEBEN ?!
        JR      NZ, SG16
        LDB     RH3, RL3        !8-BIT-SEGMENTNUMMER UMLADEN!
        CLRB    RL3
NO_SG:
SG16:   LD      SV_R(R7), R3    !SG, PC, FC, NS, NO, PS ODER PO LADEN!

R_EINF_CR:
        INC     R7, #%02        !NAECHSTES REGISTER ADRESSIEREN!
        INC     R8, #%02
```

```
        CP      R8, #R_LH - 1   !ALLE REGISTER AUSGEGEBEN ?!
        JR      C, R_EINF       !NEIN!
        RET                     !ROUTINE IST BEENDET!

!DOPPELREGISTER: RR0, RR2, RR4, RR6, RR8, RR10, RR12, RR14!
R_DPPL: CP      R8, #R_BEZ + 30 !DOPPELREGISTERKENNUNG MUSS < 15 SEIN!
        JP      NC, ERROR

        CPB     RL1, #' '       !DOPPELR.: RR0, RR2, RR4, RR6 ODER RR8 ?!
        JR      Z, R_D_R

        BIT     R1, #%0         !GERADE DOPPELREG. RR10, RR12 ODER RR14 ?!
        JR      Z, R_DOPP
        JP      NZ, ERROR

R_D_R:  BIT     R1, #%08        !GER. D.REG.: RR0, RR2, RR4, RR6, RR8 ?!
        JP      NZ, ERROR

R_DOPP: LD      R1, #'RR'       !RR IN DEN AUSGABEPUFFER EINTRAGEN!
        CALL    OUT_2ASC
        LD      R1, @R8
        CALL    OUT_2ASC
        INC     OUTPTR, #1      !REGISTERKENNUNG IN OUTBFF LADEN!
        LD      R5, SV_R(R7)    !1. REGISTERWERT AUS DEM RAM LADEN!
        CALL    BTOH16
        LD      R5, SV_R+ 2(R7) !2. REGISTERWERT AUS DEM RAM LADEN!
        CALL    NEW_ROW         !ZEILE AUSG. UND NEUEN WERT IN RR2 EINL.!
        RET     C               !"Q" ERKANNT: ENDE DER ROUTINE!

        JR      Z, DOPP_CR      !"CR" ERKANNT: REGISTERWERT BLEIBT BEST.!
        LD      SV_R(R7), R2    !DOPPELREGISTERWERT IN DEN RAM LADEN!
        LD      SV_R + 2(R7), R3
DOPP_CR:
        INC     R7, #%04        !NAECHSTES DOPPELREGISTER ADRESSIEREN!
        INC     R8, #%04
        CP      R8, #R_BEZ + 31 !ALLE DOPPELREGISTER AUSGEGEBEN ?!
        JR      C, R_DOPP
        RET                     !ENDE DER ROUTINE!

!BYTEREGISTER: RL0 .... RH7!
R_BYTE: CP      R8, #R_BEZ + 16 !REGISTER MUSS AUF EINE ZAHL < 8 ZEIGEN !
        JP      NC, ERROR

        TEST    R6              !MERKER: LOW- ODER HIGH-REGISTER!
        JR      NZ, RB_OUT
        INC     R7, #1          !UNGERADES BYTE FUER LOW REGISTER ADRESS.!
RB_OUT:
        LD      R1, R_LH(R6)    !(R6) "RL" ODER "RH" IN OUTBFF LADEN!
        CALL    OUT_2ASC
        LD      R1, @R8         !REGISTERKENNUNG AUS R_BEZ LADEN!
        CALL    OUT_2ASC
        LDB     RL5, SV_R(R7)   !REGISTERWERT AUS DEM RAM LADEN!
        CALL    BTOH8           !REGISTERWERT KONVERTIEREN!
        INC     OUTPTR, #1
        CALL    NEW_R           !ZEILE AUSG. UND NEUEN WERT IN RL3 LES.!
        RET     C               !"Q" ERKANNT: ENDE DER ROUTINE!

        JR      Z, RB_CR        !"CR" ERKANNT: WERT BLEIBT BESTEHEN!
        LDB     SV_R(R7), RL3   !NEUEN REGISTERWERT IN DEN RAM EINTRAGEN!
RB_CR:
        INC     R7, #1          !NAECHSTES BYTEREGISTER ADRESSIEREN!
        DEC     R6, #%02        !ZEIGER FUER R_LH KORRIGIEREN!
        JR      Z, RB_OUT       !LOW-REGISTER AUSGEBEN!
        LD      R6, #2          !NAECHSTES HIGH-REGISTER ADRESSIEREN!
        INC     R8, #%02
        CP      R8, #R_BEZ + 16 !ALLE BYTEREGISTER AUSGEGEBEN ?!
        JR      C, RB_OUT
        RET
```

```
ALL_R:                                  !ALLE REGISTER AUSGEBEN!
!REGISTERBEZEICHNUNGEN 1. ZEILE AUSGEBEN!
        LDK     R0, #%0C                !ZAEHLER: R0 - R7, SG,PC,FC UND RF!
        LD      R4, #R_BEZ
        LDK     R6, #%08
        LD      R8, #R_BEZ + 30
        CALR    LD_B1                   !REGISTERBEZEICHNUNGEN 1. ZEILE!

        LDK     R2, #%0C                !REGISTERWERTE 1. ZEILE AUSGEBEN!
        LD      R6, #SV_R
        LDK     R8, #%08
        LD      R10, #SVSTK + 2
        CALR    LD_WERTE

!REG_BEZEICHNUNGEN 2. ZEILE AUSGEBEN!
        LDK     R0, #%0C
        LD      R4, #R_BEZ + 16
        LD      R8, #R_BEZ + 38
        LDK     R6, #%08
        CALR    LD_B1

        LDK     R2, #%0C                !REGISTERWERTE 2. ZEILE AUSGEBEN!
        LD      R6, #SV_R + %10
        LDK     R8, #%08
        LD      R10, #RF_CTR

!REGISTERWERTE DER 1. ODER 2. ZEILE AUSGEBEN!
LD_WERTE:
        LD      R5, @R6
        CALL    BTOH16
        DEC     R2, #1
        JR      Z, OUT_WERTE
        DEC     R8, #1
        JR      NZ, LD_W2
        LD      R6, R10
LD_W2:
        INC     R6, #%02
        CP      R8, #0
        JR      GT, LD_W3
        INC     OUTPTR, #%03
        JR      LD_WERTE
LD_W3:
        INC     OUTPTR, #1
        JR      LD_WERTE
OUT_WERTE:
        CALL STRO_CR_CON                !AUSGABE DER ZEILE!
        RET

    END REGISTER

LD_B1 PROCEDURE
      ENTRY
        LD      R3, OUTPTR              !"R" IN OUTBFF EINTRAGEN!
        LDB     OUTBFF(R3), #'R'
LD_B2:  INC     OUTPTR, #1
        LD      R1, @R4                 !REGISTERKENNUNG IN OUTBFF EINTRAGEN!
        CALL    OUT_2ASC
        DEC     R0, #1                  !ZAEHLER DEKREMENTIEREN!
        JR      Z, OUT_WERTE            !AUSGABE DER WERTE!
        DEC     R6, #1
        JR      NZ, LD_B3               !8 REGISTER AUSGEGEBEN ?!
        LD      R4, R8                  !SG, PC, FC, UND RF AUSGEBEN!
LD_B3:
        INC     R4, #%02                !ZEIGER IN R_BEZ WEITERSTELLEN!
        CP      R6, #0
        JR      GT, LD_B4               !8 REGISTER AUSGEGEBEN ?!
        INC     OUTPTR, #%04
        JR      LD_B2
```

```
LD_B4:
        INC     OUTPTR, #%02
        JR      LD_B1
END LD_B1

GLOBAL RD_DATA PROCEDURE        !NEUEN DATENWERT EINLESEN!
ENTRY
        LDB     RL0, #'?'       !'?' AUSGEBEN!
        CALL    PUTA
        CALR    G_LINE          !NEUEN DATENWERT IN RR2 EINLESEN!
KONV_DATA:
        LDL     RR2, #00000000  !RR2 LOESCHEN!
        CALL    NSIGNW
        RET     Z               !RUECKSPRUNG WENN ENDEKENNUNG ERREICHT!
        CPB     RL0, #'Q'       !QUIT ? !
        JR      Z, K_Q
        CPB     RL0, #'-'       !ADRESSE DEKREMENTIEREN ?!
        JR      NZ, K_LOOP
K_Q:    SETFLG  C
        RET

K_LOOP: CALL    HTOB
        JP      C, ERROR
        RLDB    RL0, RL3
        RLDB    RL0, RH3
        RLDB    RL0, RL2
        RLDB    RL0, RH2
        CALL    RDCHR
        JR      NZ, K_1
        SETFLG  P, V
        JR      K_END
K_1:    CPB     RL0, #' '
        JR      NZ, K_LOOP
        RESFLG  C
K_END:
        RESFLG  Z
        RET
    END RD_DATA

TTY PROCEDURE
ENTRY
        LD      MAXLEN, R1
TTY_L1: CLR     R1
TTY_LOOP:
        CALL    GETA            !ZEICHEN  LESEN !
        CALL    PUTA            !ZEICHEN AUSGEBEN!

        LDB     R2(R1), RL0
        CPB     RL0, CHRDEL     !CHARACTER DELETE ? !
        JR      Z, TTY_CL
        CPB     RL0, LINDEL     !LINE DELETE ?!
        JR      Z, TTY_DEL
        CPB     RL0, #CR        !CARRIAGE RETURN ?!
        JR      Z, TTY_END
        INC     R1, #%01
        CP      R1, MAXLEN      !MAXIMALE ZEILENLAENGE ERREICHT ?!
        JR      C,TTY_LOOP      !SPRUNG WENN NICHT!
        SETFLG  Z
        RET
TTY_CL:                         !EINGEGEBENES ZEICHEN LOESCHEN!
        CALR    CLCHR
        JR      PL, TTY_LOOP
        LDB     RL0, #'#'
        CALL    PUTA
        JR      TTY_L1
TTY_DEL:
        TEST    R1              !EINGEGEBENE ZEILE LOESCHEN!
        JR      Z, TTY_LOOP
```

```
        LDB     RL0, #BS        !BACKSPACE AUSGEBEN!
        CALL    PUTA
        CALR    CLCHR
        JR      TTY_DEL
TTY_END:
        LDB     RL0, #LF
        CALL    PUTA
        LDA     R2, R2(R1)      !R2 AUF PUFFERENDE STELLEN!
        RESFLG  Z
        RET
    END TTY

CLCHR PROCEDURE                 !LOESCHEN EINES ZEICHENS!
ENTRY
        LDB     RL0, #' '
        CALL    PUTA
        LDB     RL0, #BS
        CALL    PUTA
        DEC     R1, #%01
        RET
    END CLCHR

CRLF PROCEDURE                  !AUSGABE CRLF AN CONSOLE!
ENTRY
        LDB     RL0, #CR
        CALL    PUTA
        LDB     RL0, #LF
        CALL    PUTA
    END CRLF

GLOBAL N_CT PROCEDURE           !ANPASSUNG AN LANGSAME TERMINALS!
ENTRY                           !ZUR UEBERBRUECKUNG DER ZEIT FUER!
        TEST    NULLCT          !CARRIAGE RETURN!
        RET     Z
        LD      R1, NULLCT
        CLR     R0              !NULL AUSGEBEN!
N_CT1:
        CALL    PUTA
        DEC     R1, #1
        JR      NZ, N_CT1
        RET
    END N_CT

G_LINE PROCEDURE                !EINLESEN EINER KOMMANDOZEILE BIS %0D!
ENTRY
        LD      R2, #INBFF
        LD      R1, #%80        !MAXLEN LADEN!,
        LD      INPTR, #0       !EINGABEZEIGER LOESCHEN!
        CALR    TTY
        JP      Z, ERROR
        CALL    NSIGNW
        DEC     INPTR, #%01
        CPB     RL0, #CR
        RET
    END G_LINE

INCPTR PROCEDURE                !ZEICHENKETTEN PTR INKREMENTIEREN UND!
ENTRY                           !AUF UEBERLAUF TESTEN!
        LD      R2, R0
        INC     R0, #%01
        CP      R0, #%0100      !PUFFERENDE ERREICHT!
        RET     C
        CLR     R0
        RET
    END INCPTR

END UMON_ROUT1
```

```
MON_ROUT2 MODULE
$SECTION PROM

EXTERNAL
ERROR           LABEL
CMDLOP          LABEL
TMSG            LABEL
BTOH16          LABEL
STRO_CR_CON     LABEL
RD_ADR          LABEL
NSIGN           LABEL
STACK           BYTE
B_WORD          WORD
B_ADR_S         WORD
B_ADR_O         WORD
RR14_V          LONG
IDENT_          WORD
FCW_V           WORD
SV_R            BYTE
SVSTK           WORD
SEG_            WORD
PC_             WORD
FCW_            WORD
RF_CTR          WORD
N_SEG_          WORD
N_OFF_          WORD
PS_             WORD
PO_             WORD
B_CODE          WORD

  CONSTANT
BS      := %08
LF      := %0A
CR      := %0D

!SYSTEM_CALL'S!
SEG_V  := %01
NSEG_V := %02

!******************!
GLOBAL GO_ PROCEDURE            !GO_ROUTINE!
!******************!
ENTRY
        CALL NSIGN              !NAECHSTES ZEICHEN EINLESEN!
        JR      Z, GO_PC
        CALL    RD_ADR          !STARTADRESSE LADEN!
        RES     R3, #0          !GERADE ADRESSE SICHERN!
        LDL     SEG_, RR2

GO_PC:  LDL     RR14, SVSTK     !REGISTER AUS DEM RAM ZURUECKLADEN!
        DEC     R15, #%08       !STACK KORRIGIEREN!
        LD      R1, IDENT_      !STACKBEREICH AUSLESEN UND IN DEN!
        LD      R15(#%0000), R1 !MONITOR-RAM RETTEN!
        LD      R1, FCW_
        LD      R15(#%0002), R1
        LD      R1, SEG_
        LD      R15(#%0004), R1
        LD      R1, PC_
        LD      R15(#%0006), R1

        LD      R1, N_SEG_
        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        LDCTL   NSPSEG, R1      !NORMALSTACKPOINTER-SEGMENTNR. LADEN!
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!

        LD      R1, N_OFF_
        LDCTL   NSP, R1         !NORMALSTACKPOINTER-OFFSET LADEN!
```

```
        LD      R1, PS_
        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        LDCTL   PSAPSEG, R1     !PROGRAMMSTATUSAREAPOINT.-SEGMENTNR. LAD.!
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!

        LD      R1, PO_
        LDCTL   PSAPOFF, R1     !PROGRAMMSTATUSAREAPOINTER-OFFSET LADEN!

        LD      R15, #SV_R
        LDM     R0, @R15, #14   !14 ALLGEMEINE REGISTER , AB R0, LADEN!

        LDL     RR14, SVSTK
        DEC     R15, #8
        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        IRET

END GO_

!********************!
GLOBAL BREAK PROCEDURE          !BREAKPOINT_ROUTINE!
!********************!
ENTRY
        LDL     RR2, B_ADR_S    !VORHERIGEN UNTERBRECHUNGSPUNKT LOESCHEN!
        LD      R1, B_WORD
        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        LD      @R2, R1
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!
        CLR     B_ADR_S         !ARBEITSBEREICH IM RAM LOESCHEN!
        CLR     B_ADR_0
        CLR     B_WORD
        CALL    NSIGN           !NAECHSTES ZEICHEN EINLESEN!
        RET     Z               !RUECKSPRUNG WENN B (CR): NUR LOESCHEN !
        CALL    RD_ADR          !BREAKADRESSE LESEN!

        RES     R3, #%0         !GERADEN OFFSET SICHERN!
        LDL     B_ADR_S, RR2
        LD      R1, B_CODE
        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        EX      R1, @R2         !BREAK_KODE (7F00) IN DEN SPEICHER LADEN!
        LD      R4, R1          !SPEICHERINHALT IN R4 LADEN!
        LD      R1, @R2         !BREAK_KODE RUECKLADEN, TEST: PROM/RAM!
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!
        LD      B_WORD, R4      !SPEICHERINHALT IN DEN ARBEITSSPEICHER!
        CP      R1, B_CODE
        RET     Z               !RUECKSPRUNG IN DIE KOMMANDOSCHLEIFE!
        CLR     B_ADR_S
        CLR     B_ADR_0
        CLR     B_WORD
        JP      ERROR           !BREAK AUF PROMSPEICHER GELEGT!
    END BREAK


GLOBAL SC_ENTRY PROCEDURE       !SYSTEM_CALL_ENTRY!
ENTRY
        PUSH    @R14, R0
        LDCTL   R0, FCW         !IN DEN NICHTSEGMENTIERTEN MODE UEBERG.!
        RES     R0, #15
        LDCTL   FCW, R0
        CPB     3(R15), #1      !SC_01 = SEG_ARBEITSWEISE!
        JR      NZ, SC_02

        SET     4(R15), #15     !SEGMENT_BIT IM FCW SETZEN!
        JR      SC_END

SC_02:
        CPB     3(R15), #2      !NSEG_ARBEITSWEISE!
        JR      NZ, SC_BRK
        RES     %04(R15), #15   !SEGMENT_BIT IM FCW RUECKSETZEN!
```

```
SC_END: LDCTL   R0, FCW
        SET     R0, #15         !SEGMENTIERTEN MODE EINSTELLEN!
        LDCTL   FCW, R0

        POP     R0, @R14
        IRET

SC_BRK: POP     R0, @R15        !SC_00: BREAK ERKANNT!
        LD      SV_R, R0        !REGISTER R0 RETTEN!
        LDL     SVSTK, RR14     !DOPPELREGISTER RR14 (STACK) RETTEN!
        LD      R15, #STACK

        LDL     RR14_V, RR14    !REGISTERRETTUNG IN EINEN RAM-BEREICH!
        LDL     RR14, SVSTK

        LD      R0, R15(#0)     !IDENTIFIER LADEN!
        LD      IDENT_, R0
        LD      R0, R15(#%0002) !FLAG- UND CONTROLWORT LADEN!
        LD      FCW_, R0
        LD      R0, R15(#%0004) !SEGMENTNUMMER LADEN!
        LD      SEG_, R0
        LD      R0, R15(#%0006) !OFFSET LADEN!
        LD      PC_, R0

        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        LDCTL   R0, NSPSEG      !NORMALSTACKPOINTER-SEGMENTNR. LADEN!
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!
        LD      N_SEG_, R0

        LDCTL   R0, NSPOFF      !NORMALSTACKPOINTER-OFFSET LADEN!
        LD      N_OFF_, R0

        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        LDCTL   R0, PSAPSEG     !PROGRAMMSTATUSAREA-SEGMENTNR. LADEN!
        SC      #NSEG_V         !NICHTSEGMENTIERTEN MODE  EINSTELLEN!
        LD      PS_, R0

        LDCTL   R15, PSAP       !PROGRAMMSTATUSAREA-OFFSET LADEN!
        LD      PO_, R15

        LDCTL   R15, REFRESH
        LD      RF_CTR, R15

        LD      R15, #SV_R + 2
        LDM     @R15, R1, #13   !RESTLICHE NORMALREGISTER, AB R1, LADEN!
        LDL     RR14, RR14_V

        INC     SVSTK + 2, #8
        DEC     PC_, #%02
        EI      VI

        LD      R2, #T_BRK      !TEXT: BREAK AT ....!
        CALL    TMSG
        LD      R5, PC_         !PC-INHALT AUSGEBEN!
        CALL    BTOH16
        CALL    STRO_CR_CON
        JP      CMDLOP

    END SC_ENTRY


T_BRK:  WORD    := %09
        ARRAY   [* BYTE] := 'BREAK AT  '

END MON_ROUT2
```

```
UMON_HW MODULE
$SECTION PROM

EXTERNAL
TTYSRC          LABEL
INCPTR          LABEL
TTYDST          WORD
TTYFLG          BYTE
TTY_BF          WORD

CONSTANT
CR              := %0D          !CARRIAGE RETURN!
SIOCON_A        :=%FF85
SIOCON_B        := %FF87
SIODAT_B        := %FF83
CTC0            := %FFAB

!SYSTEM_CALL'S!
SEG_V := 01

GLOBAL TTYINT PROCEDURE         !INT_SERVICE_ROUTINE_EINZELZEICHENEINGABE!
ENTRY
        PUSH    @R15, R0
        PUSH    @R15, R1
        PUSH    @R15, R2

TTY_RD: INB     RL1, SIODAT_B   !ZEICHEN EINLESEN!
        RESB    RL1, #%07       !RES BIT 7 DES ASCII-ZEICHENS!
        LD      R0, TTYDST
        CALL    INCPTR
        LD      TTYDST, R0
        LDB     TTY_BF(R2), RL1
        INCB    TTYFLG, #%01

        LDB     RL1, #SIO_RETI
        OUTB    SIOCON_A, RL1   !RETI FUER SIO AN DEN KANAL A AUSGEBEN!

        POP     R2, @R15
        POP     R1, @R15
        POP     R0, @R15
        SC      #SEG_V          !SEGMENTIERTEN MODE  EINSTELLEN!
        NOP
        IRET
    END TTYINT

GETA PROCEDURE                  !EINLESEN EINES ZEICH. AUS DEM INT_PUFFER!
ENTRY
        TESTB   TTYFLG          !ZEICHEN EINGEGANGEN ?!
        JR      Z, GETA
        PUSH    @R15, R2        !PUFFERADRESSE RETTEN!
        LD      R0, TTYSRC
        CALL    INCPTR          !REL. ZEIGER IM TTY_PUFFER ERHOEHEN!
        LD      TTYSRC, R0
        LDB     RL0, TTY_BF(R2) !ZEICHEN AUS DEM PTY_PUFFER LADEN!
        DECB    TTYFLG, #%01
        POP     R2, @R15
        RET
    END GETA

GLOBAL PUTA PROCEDURE           !AUSGABE EINES ASCII-ZEICHENS!
ENTRY
        INB     RH0, SIOCON_B
        BITB    RH0, #%02       !TRANSMIT BUFFER EMTTY ? !
        JR      Z, PUTA
        OUTB    SIODAT_B, RL0

        CPB     RL0, #CR        !LETZTES ZEICHEN AUSGEGEBEN ? !
        RET
```

```
    END PUTA


TTY_INIT PROCEDURE                  !CTC INITIALISIERUNG FUER DIE !
                                    !SERIELLE SCHNITTSTELLE!
ENTRY
        LDB     RL0, #BAUD9600      !BAUDRATE SIO KANAL B!
        LDB     RL1, #%57           !KEIN INT., ZAEHLER, VORTEILER 16!
                                    !ZEITKONSTANTE: BAUD9600!
        OUTB    CTC0, RL1
        OUTB    CTC0, RL0

!SIO INITIALISIERUNG!
        LD      R2, #INIT_SIO_B
        LD      R0, #11             !LAENGE DER INITIALISIERUNGSTABELLE!
        LD      R1, #SIOCON_B       !CONTROLADRESSE DES SIO KANAL B LADEN!
        OTIRB   @R1, @R2, R0        !SIO IST NACH DER AUSGABE BEREIT!
        RET
    END TTY_INIT

INIT_SIO_B:
        ARRAY [* BYTE] := [COMM3
                           SIO_R4 + COMM2
                           S2 + C32 + NOPRTY
                           SIO_R3
                           RENABLE + B8
                           SIO_R5
                           XENABLE + T8 + DTR + RTS
                           SIO_R1 + COMM2
                           PDAVCT + SAVECT
                           SIO_R2           !KENNBYTE (%00) ZUR!
                            %00]            !ADRESSIERUNG DER SIO-INT-ROUT.!
                                            !IN DER PROGRAMMSTATUSAREA!

CONSTANT
BAUD9600  := %02
SIO_R1    := %01
SIO_R2    := %02
SIO_R3    := %03
SIO_R4    := %04
SIO_R5    := %05
COMM2     := %10          !RESET EXT/STAT INTERRUPT!
COMM3     := %18          !CHANNEL RESET!
C32       := %80          !X32 CLOCK MODE!
S2        := %0C          !2 STOP/BITS PRO CHARACTER!
NOPRTY    := %00          !DISABLE PARITY!
RENABLE   := %01          !RECEIVER ENABLE!
B8        := %C0          !RECEIVE 8 BITS/CHARACTER!
XENABLE   := %08          !TRANSMITTER ENABLE!
T8        := %60          !TRANSMIT 8 BITS/CHARACTER!
DTR       := %80          !DATA TERMINAL READY!
RTS       := %02          !REQUEST TO SEND!
PDAVCT    := %18          !RX INTERRUPT ON ALL CHARACTERS!
SAVECT    := %04          !STATUS AFFECTS VECTOR!
SIO_RETI
          := %38          !RETI-KOMMANDO FUER SIO!
END UMON_HW
```

```
MON_RAM MODULE
$SECTION RAM


GLOBAL

RAMBOT   ARRAY   [%20 BYTE]
NSP_OFF  ARRAY   [%80 BYTE]
STACK    ARRAY   [16 BYTE]
INBFF    ARRAY   [%80 BYTE]       !INPUT_BUFFERO!
OUTBFF   ARRAY   [%80 BYTE]       !AUSGABEPUFFER!
TTY_BF   ARRAY   [%100 BYTE]      !PTY_PUFFER ZUR INT. GESTEUERTEN!
                                  !ZEICHENEINGABE!
MAXLEN   WORD                     !MAXIMALE ZEILENLAENGE INPUT!
INPTR    WORD                     !INPUT_POINTERO!
OUTPTR   WORD                     !LAENGE DES AUSGABEPUFFERS!
TTYFLG   WORD                     !FLG ZEIGT AN OB EIN ZEICHEN!
                                  !EINGEGANGEN IST!
TTYDST   WORD                     !ZEIGER ZUM NAECHSTEN FREIEN PLATZ!
                                  !IM PTY_BF!
TTYSRC   WORD                     !ZEIGER ZUM NAECHSTEN AUS PTY_BF!
                                  !AUSZULESENDEN!
                                  !ZEICHEN!
B_WORD   WORD                     !SPEICHERINHALT DES BREAK!
B_ADR_S  WORD                     !SEG_NR. DES BREAK!
B_ADR_O  WORD                     !OFFSET DES BREAK!
RR14_V   LONG                     !VARIABLER STACK, FUER BREAK!
IDENT_   WORD
FCW_V    WORD                     !VARIABLES FCW FUER NEXT UND BREAK!
PORT_BF  WORD                     !I/O-PORT PUFFER!
SV_R     ARRAY [14 WORD]          !REGISTERRETTUNGSBEREICH!
SVSTK    LONG
SEG_     WORD
PC_      WORD                     !PROGRAMMCOUNTER!
FCW_     WORD                     !FCW IM REGISTERRETTUNGSBEREICH!
RF_CTR   WORD
N_SEG_   WORD                     !NORMALSTACK R14!
N_OFF_   WORD                     !NORMALSTACK R15!
PS_      WORD                     !PROGRAMMSTATUSAREA_SEGMENT!
PO_      WORD                     !PROGRAMMSTATUSAREA_OFFSET!
         WORD

FLAG     BYTE
KONV     BYTE                     !HILFSBYTE ZUR KONVERTIERUNG!

PSAREA   ARRAY [%100 WORD]

CHRDEL   BYTE
LINDEL   BYTE
NULLCT   WORD
B_CODE   WORD                     !BREAK_CODE (7F00)!
PROMT    WORD
PORTBF   WORD
END MON_RAM
```

Anhang A CPU-Befehlsliste

Standardbefehle

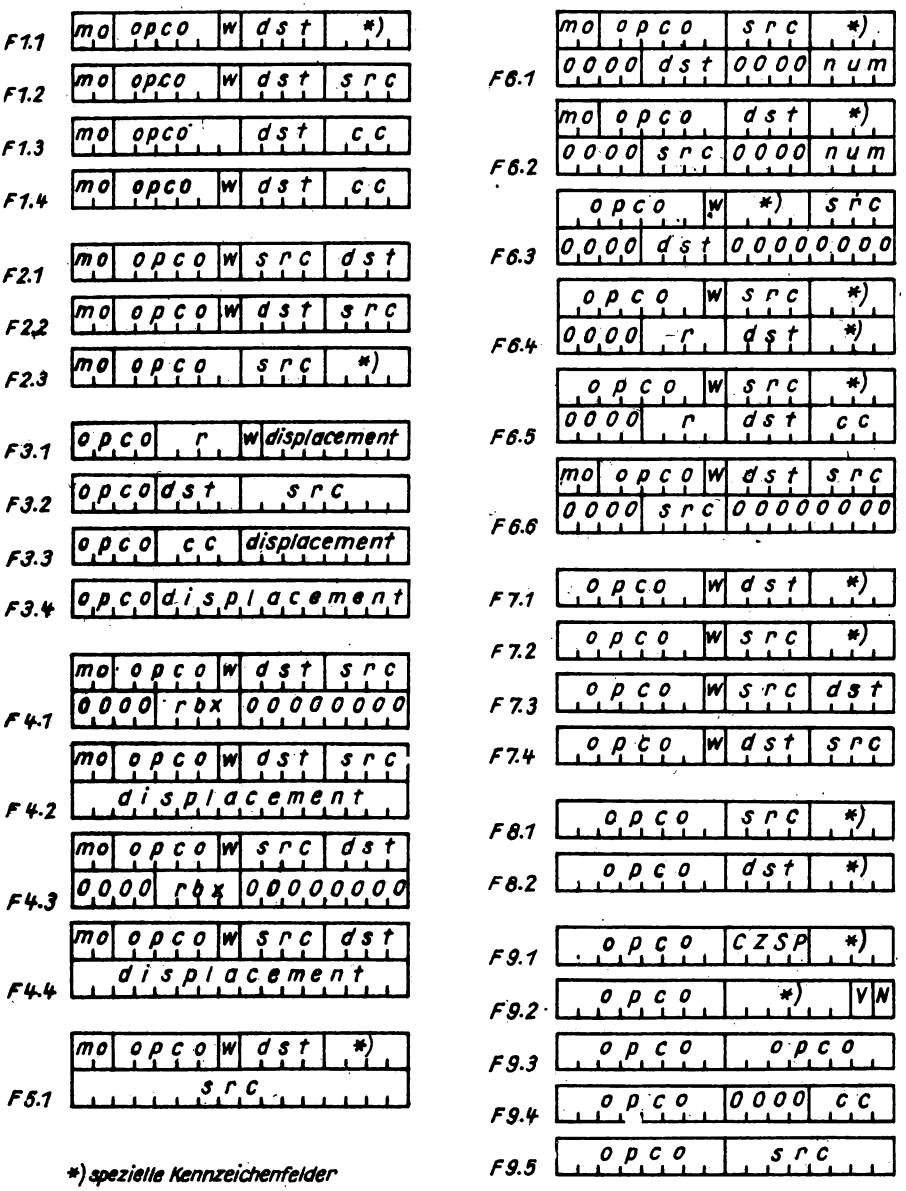
EPU-Befehle

Operation

Anhang B MMU-Befehlsliste

Anhang C Tabellen und Übersichten

Befehlsformate



*) spezielle Kennzeichenfelder

ASCII-Zeichensatz

Dezimale Konvertierung

000 NUL	032 SPA	064 @	096
001 SOH	033 !	065 A	097 a
002 STX	034 "	066 B	098 b
003 ETX	035 #	067 C	099 c
004 EOT	036 $	068 D	100 d
005 ENQ	037 %	068 E	101 e
006 ACK	038 &	070 F	102 f
007 BEL	039 '	071 G	103 g
008 BS	040 (	072 H	104 h
009 HT	041)	073 I	105 i
010 LF	042 *	074 J	106 j
011 VT	043 +	075 K	107 k
012 FF	044 ,	076 L	108 l
013 CR	045 -	077 M	109 m
014 SO	046 .	078 N	110 n
015 SI	047 /	079 O	111 o
016 DLE	048 0	080 P	112 p
017 DC1	049 1	081 Q	113 q
018 DC2	050 2	082 R	114 r
019 DC3	051 3	083 S	115 s
020 DC4	052 4	084 T	116 t
021 NAK	053 5	085 U	117 u
022 SYN	054 6	086 V	118 v
023 ETB	055 7	087 W	119 w
024 ETB	056 8	088 X	120 x
025 EM	057 9	089 Y	121 y
026 SUB	058 :	090 Z	122 z
027 ESC	059 ;	091 [	123 {
028 FS	060 <	092 \	124 |
029 GS	061 =	093]	125 }
030 RS	062 >	094 ^	126 ~
031 US	063 ?	095 _	127 DEL

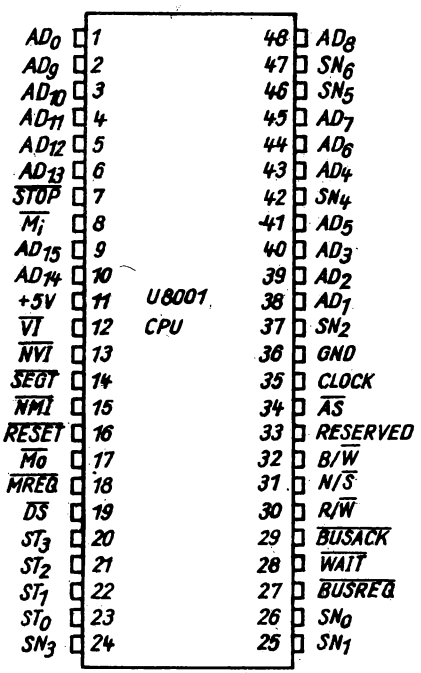
Hexadezimale Konvertierung

00 NUL	20 SPA	40 @	60 `
01 SOH	21 !	41 A	61 a
02 STX	22 "	42 B	62 b
03 ETX	23 #	43 C	63 c
04 EOT	24 $	44 D	64 d
05 ENQ	25 %	45 E	65 e
06 ACK	26 &	46 F	66 f
07 BEL	27 '	47 G	67 g
08 BS	28 (	48 H	68 h
09 HT	29)	49 I	69 i
0A LF	2A *	4A J	6A j
0B VT	2B +	4B K	6B k
0C FF	2C ,	4C L	6C l
0D CR	2D -	4D M	6D m
0E SO	2E .	4E N	6E n
0F SI	2F /	4F O	6F o
10 DLE	30 0	50 P	70 p
11 DC1	31 1	51 Q	71 q
12 DC2	32 2	52 R	72 r
13 DC3	33 3	53 S	73 s
14 DC4	34 4	54 T	74 t
15 NAK	35 5	55 U	75 u
16 SYN	36 6	56 V	76 v
17 ETB	37 7	57 W	77 w
18 ETB	38 8	58 X	78 x
19 EM	39 9	59 Y	79 y
1A SUB	3A :	5A Z	7A z
1B ESC	3B ;	5B [	7B {
1C FS	3C <	5C \	7C |
1D GS	3D =	5D]	7D }
1E RS	3E >	5E ^	7E ~
1F US	3F ?	5F _	7F DEL

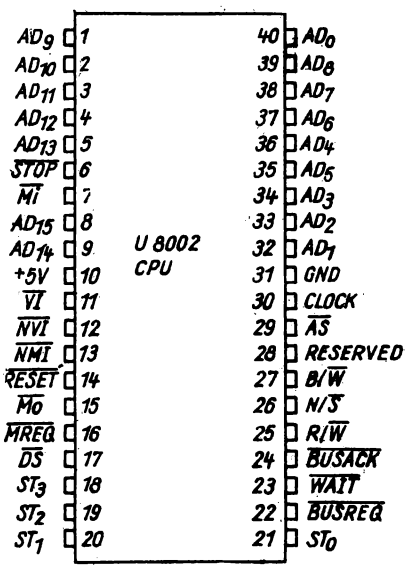
Oktale Konvertierung

000	NUL	040	SPA	100	@	140	`
001	SOH	041	!	101	A	141	a
002	STX	042	"	102	B	142	b
003	ETX	043	#	103	C	143	c
004	EOT	044	$	104	D	144	d
005	ENQ	045	%	105	E	145	e
006	ACK	046	&	106	F	146	f
007	BEL	047	'	107	G	147	g
010	BS	050	(	110	H	150	h
011	HT	051	)	111	I	151	i
012	LF	052	*	112	J	152	j
013	VT	053	+	113	K	153	k
014	FF	054	,	114	L	154	l
015	CR	055	-	115	M	155	m
016	SO	056	.	116	N	156	n
017	SI	057	/	117	O	157	o
020	DLE	060	0	120	P	160	p
021	DC1	061	1	121	Q	161	q
022	DC2	062	2	122	R	162	r
023	DC3	063	3	123	S	163	s
024	DC4	064	4	124	T	164	t
025	NAK	065	5	125	U	165	u
026	SYN	066	6	126	V	166	v
027	ETB	067	7	127	W	167	w
030	ETB	070	8	130	X	170	x
031	EM	071	9	131	Y	171	y
032	SUB	072		132	Z	172	z
033	ESC	073	;	133	[	173	{
034	FS	074	<	134	\	174	|
035	GS	075	=	135	]	175	}
036	RS	076	>	136	^	176	~
037	US	077	?	137	_	177	DEL

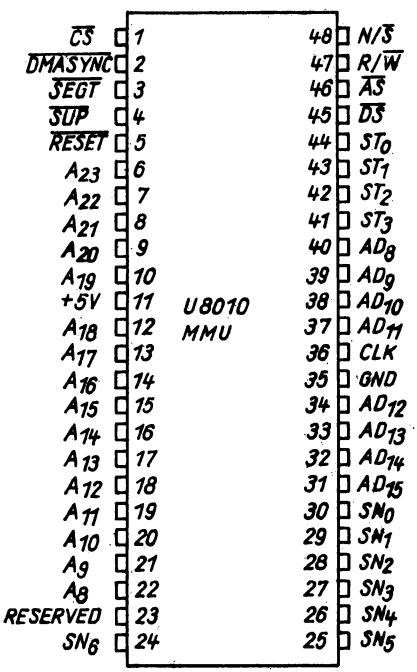
U8001-Pinbelegung, physisch



U8002-Pinbelegung, physisch



U8010-Pinbelegung, physisch



Literaturverzeichnis

/1/ 16-Bit Mikroprozessoren U8001/U8002. Kundeninformation. Erfurt: VEB Mikroelektronik "Karl Marx" 1986

/2/ Speicherverwaltungsbaustein MMU U8010. Kundeninformation. Erfurt: VEB Mikroelektronik "Karl Marx" 1986

/3/ Handbuch UDOS-1526. Teil: Sprachbeschreibung/Manual Cross-Software U8001/U8002 ASM. VEB Robotron-Buchungsmaschinenwerk Karl-Marx-Stadt 1983

/4/ Claßen, L.; Oefler, U.: Wissensspeicher Mikrorechnerprogrammierung. Berlin: VEB Verlag Technik 1986

/5/ Mateosian, Richard: Programming the Z8000. SYBEX Inc. 1980

/6/ Stuhlmüller, P.: 16-bit-Generation Z8000, Aufbau und Anwendung. München: te-wi Verlag 1980

/7/ Moore, Martin L.: Z8000 Using and Troubleshooting the Z8000. Hemel Hempstead: Prentice Hall International 1982

/8/ Report on the Programming Language PLZ. Zilog Co. 1978

/9/ Claßen, L.; Oefler, U.: UNIX und C - Ein Anwenderhandbuch. Berlin: Verlag Technik 1987

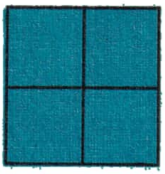
/10/ Kernighan, B.W.; Ritchie, D.M.: The C Programming Language. Hemel Hempstead: Prentice Hall 1978

/11/ Betriebssystem UDOS 1526 Anwenderdokumentation. VEB Robotron Buchungsmaschinenwerk Karl-Marx-Stadt 1983

/12/ Z80-RIO Relocating Assembler and Linker User's Manual. Zilog Co. 1978

/13/ Z8000 PLZ/ASM Assembly Language Programming Manual. Zilog Co. 1979

Sachwörterverzeichnis



Der Band vermittelt einen umfassenden Systemüberblick zum 16-Bit-Mikroprozessorsystem U 8000 mit detaillierten Angaben zur Programmierung der CPU und der MMU. Die zur Programmentwicklung erforderliche Software unter dem 8-Bit-Betriebsystem UDOS und dem UNIX-kompatiblen 16-Bit-Betriebssystem WEGA wird vorgestellt. Zur Unterstützung der Einarbeitung sind ausführliche Beispiele enthalten.

ISBN 3-341-00286-3