Praktikum zu Technische Grundlagen der Informatik

Ausgabe Winter-Semester 1999/2000, Stand: 24. März 2000

Arndt Bode
mit
Andreas Bauer, Birgit Eßbaumer, Wolfgang Karl, Markus Leberecht,
Peter Luksch, Bruno Piochacz, Max Walter, Roland Wismüller

Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik der Technischen Universität München
D-80290 München

Tel: (089) 289-28240, Fax: (089) 289-28232
email: bode@informatik.tu-muenchen.de

Id: DasBuch.tex,v 3.9 1999/12/07 14:39:56 wismuell Exp

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Teil I

Allgemeine Hinweise zum TGI-Praktikum

1 Einführung in die Aufgabestellungen

Wolfgang Karl

Ein digitales Rechensystem nach dem von-Neumann'schen Konzept besteht aus den Komponenten Prozessor, Speicher und Ein-/Ausgabeeinheiten. Der prozessor hat die Aufgabe, Rechengrößen durch ein Programm, bestehend aus einer Folge von Maschinenbefehlen, zu verarbeiten. Die zu verarbeitenden Daten und das Programm sind im Hauptspeicher abgelegt. Über die Ein-/Ausgabeeinheiten werden die Daten von den Peripheriegeräten gelesen bzw. über diese wieder ausgegeben. Die Datentransporte und die Signalübertragungen zwischen den Komponenten finden über Verbindungswege (Adreß-, Daten- und Steuerbusse) statt.

Der Prozessor besteht aus einem Leitwerk (Steuerwerk) und einem ausführenden Werk (Rechenwerk, Operationswerk). Die Abarbeitung eines Programms wird durch das Leitwerk gesteuert. Es veranlaßt das Lesen der Maschinenbefehle aus dem Hauptspeicher, interpretiert sie und steuert ihre Ausführung. Das Leitwerk übernimmt auch die Ansteuerung des Hauptspeichers und der Ein-/Ausgabeeinheiten. Das Rechenwerk speichert die zu verarbeitenden Operanden und die Resultate und führt die arithmetischen und logischen Operationen aus.

Im Praktikum Technische Grundlagen der Informatik sollen die in der Vorlesung vermittelten Kenntnisse über die Organisation, die Arbeitsweise und den Entwurf von digitalen Rechensystemen mit Hilfe praktischer Aufgaben vertieft werden. Die in drei Breiche aufgeteilten Aufgaben decken somit wesentliche Aspekte der Rechnerarchitektur ab.

Die Aufgaben aus dem Bereich I befassen sich mit der *Organisation eines Rechners*. Am Beispiel von PCs werden durch die *maschinennahe Programmierung* der Aufgaben detaillierte Kenntnisse über deren Aufbau, der Funktionsweise der einzelnen Komponenten und den Systemschnittstellen vermittelt.

Der Prozessor arbeitet die Maschinenbefehle schrittweise ab. Für jeden Maschinenbefehl legt der Maschinenbefehlszyklus die Phase für den Befehlszugriff und die Ausführung fest. Mit den Aufgaben aus dem Bereich II *Mikroprogrammierte Implementierung von Maschinenbefehlen* sollen den Praktikumsteilnehmern die internen Abläufe in einem digitalen Rechensystem bei der Abarbeitung von Befehlen anschaulich vermittelt werden.

Der interne Aufbau und der Entwurf der Funktionseinheiten und der Rechnerkomponenten wird an ausgewählten Beispielen im Bereich III *Rechnergestützter Schaltungsentwurf* behandelt.

2 Richtlinien zur projektorientierten Durchführung des TGI-Praktikums

Wolfgang Karl

2.1 Allgemeine Anmerkungen

In diesem Merkblatt sind einige Richtlinien zur projektorientierten Durchführung des TGI-Praktikums zusammengefaßt. Sie sollen den Teilnehmerinnen und Teilnehmern helfen, die Durchführung und Lösung ihrer Aufgabe systematisch zu planen und vor allem projektorientiert durchzuführen. Während den einzelnen Phasen des „Projekts“ sind von den Projektgruppen bereits verschiedene Dokumente zu erstellen, die vom Betreuer zu kontrollieren sind. Sie helfen dem Betreuer, sich über den aktuellen Stand der Arbeit genau und schnell zu informieren.

2.2 Phasen des Praktikumsprojektes

Wie jedes Projekt ist auch ein Praktikums-Projekt in Phasen aufzuteilen, also in zeitlich und funktionell abgrenzbare Teile. Für die Bearbeitung einer Aufgabe im TGI-Praktikum sind folgende Phasen sinnvoll:

- Vorbereitung des Projekts
- Analysephase (Projektumriß)
- Konzept und Spezifikation
- Realisierung und Test
- Erstellen der Dokumentation

2.3 Vorbereitung des Projekts

2.3.1 Kurzfassung

Aufgabe	Vorbereitung des Projekts, Wiederholung des in der Vorlesung behandelten Stoffes zu dem jeweiligen Themenbereich, Einarbeitung in die Themenstellung
Grundlagen	Vorlesung und Übungen zu TGI
Ziel	Mit der Einarbeitung und dem Literaturstudium zum Praktikumsversuch werden die Voraussetzungen zur Bearbeitung der eigentlichen Aufgabe geschaffen. Auch bei einem in der Arbeitswelt durchzuführenden Projekt sind oftmals Vorstudien nötig, bevor einem Auftrag zugestimmt werden kann.
Zu erstellende Dokumente	noch keine

2.3.2 Vorbereitende Schritte

Für jede Aufgabe existiert bereits eine Aufgabenbeschreibung. Die Aufgabenbeschreibung enthält die jeweils von einer Gruppe zu bearbeitenden Aufgaben.

Die im TGI-Praktikum gestellten Aufgaben können jeweils von einer Projektgruppe (3 - 4 Studenten) in etwa 3 -4 Semesterwochen bearbeitet werden, wobei pro Student und Woche etwa eine Arbeitszeit von 3-4 Stunden angenommen wird.

Damit die Aufgaben bearbeitet werden können, ist eine Wiederholung des in der Vorlesung und in den dazugehörigen Übungen behandelten Stoffes unerläßlich.

In einer ersten Besprechnung mit dem Betreuer müssen sich dann die Projektmitarbeiter genau über die Aufgabenstellung und die Zielsetzungen informieren und sich über das Projekt ein Bild machen. Bei diesem Gespräch weist der Betreuer auch hin, welche Kenntnisse zur Bearbeitung der Aufgabe notwendig sind und welche Hilfsmittel und Arbeitsgeräte zur Verfügung stehen. Damit können sich die Studenten ein Bild vom Umfang der zu bearbeitenden Aufgabe machen.

2.4 Analyse (Projektumriß)

2.4.1 Kurzfassung

Aufgabe	Umschreibung und Abgrenzung des Problembereichs. Problemanalyse, Skizzierung erster Lösungsideen. Formulierung des Plichtenheftes und darauf aufbauend Abschätzung von Umfang und Ablauf des Projekts.
Grundlagen	Aufgabenstellung, Handreichungen
Ziel	Klarheit über den Rahmen der zu bearbeitenden Aufgabe, wobei Anregungen und Wünsche des Betreuers sowie die Rahmenbedingungen festzuhalten sind.
Zu erstellende Dokumente	Pflichtenheft, Arbeitsplan, Zeitplan

2.4.2 Vorgehen

Ausgehend von gegebenen Voraussetzungen (*Analyse des Ist-Zustandes*) und der Aufgabenstellung (*Analyse des Soll-Zustandes*) erstellt die Projektgruppe ein *Pflichtenheft*. Dieses Plichtenheft umfaßt die für eine Problemlösung maßgebenden Zielvorstellungen, Randbedingungen und Bewertungskriterien.

2.4.3 Anmerkungen zum Plichtenheft

Pflichtenhefte sind immer wichtig, wenn es darum geht, verschiedene Lösungsvarianten aufzustellen, zu bewerten (zu evaluieren) und durch Vergleich eine Wahl zu treffen. Es ist wichtig, im voraus festzustellen, welches die Ausgangslage ist (Ist-Zustand) und welche Randbedingungen gegeben sind.

Ein Plichtenheft enthält folgende Punkte:

- Darstellung des Ist-Zustandes,
- Darstellung der Anforderung und Ziele der neuen Lösung.

Dabei soll sich das Pflichtenheft auf die Beschreibung jener Aspekte beschränken, die für die Lösung bzw. für den Vergleich verschiedener Lösungsmöglichkeiten wesentlich sind.

Die Zielvorstellungen soll im wesentlichen beschreiben, was die neue Lösung bringen soll. Die Liste der Anforderungen erhält man aus einer Soll-Zustandsanalyse.

2.4.4 Arbeits- und Zeitplan

Möglichst frühzeitig soll mit dem Betreuer ein erster Arbeits- und Terminplan erstellt werden.

Im Arbeitsplan werden die Arbeitsschritte festgelegt. Es werden die Tätigkeiten (Projektphasen) aufgelistet, die vermutlich anfallen und wofür Aufwand zu leisten ist. Aus dem Arbeitsplan muß auch ersichtlich sein, wie die Arbeiten unter den Projektteilnehmern aufgeteilt wird.

Im Terminplan (z. B. ein Balkendiagramm) wird die zeitliche Koordination der vorgesehenen Arbeiten festgehalten.

Arbeits- und Zeitplan müssen im Laufe der Arbeit regelmäßig überprüft und wenn notwendig angepaßt werden.

2.5 Konzept und Spezifikation

2.5.1 Kurzfassung

Aufgabe	Erarbeitung eines Lösungskonzeptes und strukturierte Entwicklung der Problemlösung;
Grundlagen	Pflichtenheft, Handreichungen
Ziel	Ausgehend von dem Grobkonzept soll die genaue Spezifikation der Lösung entwickelt werden.
Zu erstellende Dokumente	Spezifikation

2.5.2 Vorgehen bei der Entwicklung der Spezifikation

Während dieser Phase entsteht aus einem Grobentwurf die detaillierte Spezifikation der Lösung. Es sind die Schnittstellen anzugeben sowie die Struktur der Lösung systematisch zu erarbeiten. Die Spezifikation ist mit dem Betreuer abzusprechen. Erst nach Abnahme der Spezifikation kann mit der Realisierung der Lösung begonnen werden.

2.6 Realisierung

2.6.1 Kurzfassung

Aufgabe	Implementierung der Spezifikation;
Grundlagen	Spezifikation, Handreichungen
Ziel	Funktionsfähige Programme
Zu erstellende Dokumente	Programme und Programmbeschreibungen

Diese Phase umfaßt die strukturierte Implementierung der Problemlösung. Die erstellten Programme sind zu testen.

2.7 Dokumentation und Vortrag

Zum Abschluß der Arbeit ist eine ausführliche Dokumentation zu erstellen. Die während der Arbeit bereits angefertigten Dokumente (Spezifikation etc.) können dabei verwendet werden. Die Lösung ist von den Teilnehmern auch in einem Kurzvortrag zu präsentieren.

Teil II

Dokumentationen

3 Bereich I: Rechnerorganisation und maschinennahe Programmierung von Systemen

3.1 Kurzbeschreibung der Intel 80x86-Architektur

Wolfgang Karl, Andreas Bauer

Der nachfolgende Abschnitt soll eine kurze Beschreibung der Intel 80x86-Mikroprozessorfamilie liefern und als Nachschlagewerk für deren Maschinenbefehle dienen. Damit soll Ihnen eine Hilfe bei der Assembler- bzw. Maschinenprogrammierung an die Hand gegeben werden.

Es sei darauf hingewiesen, daß die nachfolgende Beschreibung den in der Vorlesung *Technische Grundlagen der Informatik* behandelten Stoff voraussetzt. Die Dokumentationen ergänzen höchstens den in der Vorlesung behandelten Stoff, ersetzen aber auch nicht das Studium der bereitstehenden Lehr- und Handbücher zur Assembler-Programmierung der Intel-Architektur.

3.1.1 Einführung

Die wichtigsten Vertreter der 16-, 32- und 64/32-Bit Mikroprozessorfamilie von Intel sind der 8086, 80286, i386, i486, Pentium und Pentium Pro.

Den Ausgangspunkt dieser Mikroprozessorreihe bildet der 16-Bit Mikroprozessor *Intel 8086*, der einen 1 MByte großen Adreßraum aufweist. Der 8086 sieht eine logische Aufteilung des adressierbaren Speichers in Segmente vor. Durch seine interne Segmentverwaltung umfassen die Adressen eine Breite von 20 Bits. Die Basisadresse eines Segments wird unmittelbar aus einem 16-Bit Selektor in einem Segmentregister gebildet, indem dieser durch vier niedrigstwertie Null-Bits auf 20 Bit erweitert wird. Die eigentlichen Programmadressen sind 16-Bit breit und wirken als Offsets innerhalb der Segmente, die damit 64 KBytes groß sein können.

Der 8086 enthält eine Verarbeitungseinheit für 16-Bit Ganzzahlarithmetik, wobei der umfangreiche Befehlssatz auch Stringbefehle umfaßt. Über eine Coprozessor-Schnittstelle wird der Anschluß des Arithmetik-Coprozessors 8087 unterstützt.

Der *Intel 80286* ist ebenfalls ein Prozessor mit einer 16-Bit Struktur, jedoch mit einem gegenüber dem 8086 erweiterten Befehlssatz. Der 80286 unterstützt einen virtuellen, segmentierten Speicher und Systeme mit Multi-Tasking. Die Speicherverwaltung arbeitet auf der Basis von Segmentdeskriptoren mit 24-Bit-Adressen und Zugriffsschutzattributen. Der Prozessor sieht vier Ebenen unterschiedlicher Priorität vor. Dieses Schutzkonzept unterstützt zusammen mit der Verwendung der Deskriptortechnik Multi-Tasking. Neben diesem als *Protected Mode* bezeichneten Betriebsmodus, kann der 80286 auch weiterhin im Betriebsmodus des 8086, dem sogenannten *Real Mode* arbeiten.

Der *Intel i386*, der erste der 32-Bit Prozessoren dieser Mikroprozessorreihe, sieht einen 32-Bit breiten Adreß- und einen 32-Bit breiten Datenbus vor, was einen Zugriff auf einen 4GBytes großen linearen Adreßraum erlaubt. Er enthält eine Verarbeitungseinheit für 32-Bit Ganzzahlarithmetik (Integer Unit), Speicherverwaltungseinheiten für die Verwaltung von Segmenten und für Seiten sowie und eine leistungsfähige Bus-Steuereinheit. Über eine Coprozessor-Schnittstelle ist der Aritmetik-Coprozessor 80387 für die Verarbeitung von 80-Bit-Gleitkommazahlen anschließbar.

Der Intel i486 integriert auf einem Chip neben den bereits auf dem i386 vorhanden Einheiten eine Verarbeitungseinheit für 80-Bit-Gleitkommaarithmetik und einen 8KByte großen Cache-Speicher für Befehle und Daten.

Die Arbeitsmodi, in denen die Prozessoren i386 und i486 arbeiten können, sind der Protected Mode und der Real-Mode. Im Real Mode verhalten sich die beiden Prozessoren wie ein schneller 8086 und sehen dabei eine 8086 Segmentverwaltung mit 20-Bit Segmentadressen und Segmentgrößen bis zu 64 KBytes vor.

Im eigentlichen Arbeitsmodus der beiden Prozessoren, dem Protected Mode, werden die leistungsfähigen Speicherverwaltungseinheiten benutzt. Die Segmentverwaltung verwendet Deskriptoren zur Beschreibung von Segmenten mit einer Größe von bis zu 4 GBytes. Die Beschreibung eines Segments bezieht auch Zugriffschutzattribute mit ein. Die Seitenverwaltung erlaubt eine bessere Nutzung des Speichers durch eine Aufteilung in Seiten.

Damit mehrere Tasks in einer Multi-Tasking-Umgebung verwaltet werden können, werden die Segmentdeskriptoren in unterschiedlichen Deskriptortabellen im Speicher bereitgestellt. Neben der globalen Deskriptortabelle (GDT) für das Betriebssystem gibt es für jede Task eine eigene lokale Deskriptortabelle (LDT). In einer lokalen Deskriptortabelle sind die Deskriptoren der Code-, Daten- und Stacksegmente einer Task enthalten, wobei spezielle Deskriptoren vorhanden sein können, die den Aufruf von Prozeduren und anderer Tasks in einer kontrollierten Weise unterstützen. Die globale Deskriptortabelle unterstützt die Zugriffe auf die Segmente des Betriebssystems, der lokalen Deskriptortabellen und auf die sogenannten Task-State-Segmente. Diese nehmen den Zustand einer Task bei einem Task-Wechsel auf. Für die Anwahl von Interrupt- und Trap-Routinen bei der Ausnahmeverarbeitung gibt es die Interrupt-Deskriptor-Tabelle. In den Segmentdeskriptoren wird auch die Prilegebene für die Prozeduren und Tasks festgelegt. Insgesamt gibt es vier Privilegebenen. Aufrufe von Prozeduren und Tasks, die mit einem Wechsel in eine höher privilegierte Ebene verbunden sind, erfolgen über spezielle Deskriptoren, wodurch die Segmentverwaltung in der Lage ist, die Zulässigkeit solcher Aufrufe zu überprüfen. Das Zusammenwirken von der vierstufigen Schutzhierarchie und der Deskriptortechnik ermöglichen die Isolierung und den Zugriffschutz zwischen Benutzer-Tasks und Betriebssystemkomponenten.

Mit dem Intel Pentium Prozessor sind eine Reihe von Maßnahmen zur Leistungssteigerung getroffen worden, ohne die Kompatibilität zu den Vorgängern aufzugeben. Der Pentium verarbeitet 32-Bit Integerdaten, enthält aber 64-Bit breite interne und nach außen geführte Datenpfade. Die Adreßleitungen umfassen weiterhin 32 Bits (64/32-Bit-Struktur).

Der Prozessor enthält mehrere Funktionseinheiten, die in zwei parallelen Befehlspipelines organisiert sind. Eine geeignete Bildung von Befehlspaaren ermöglicht es, daß mehr als ein Befehl pro Taktzyklus die Ausführung beenden können. Weitere Komponenten wie der Branch-Target-Buffer zur dynamischen Vorhersage des Verhaltens bei Verzweigungen und die Prefetch-Buffer, in denen im voraus geholte Befehle zwischengespeichert werden können, sorgen dafür, daß möglichst viele Befehle rechtzeitig zur Ausführung bereitstehen. Im Gegensatz zum i486 enthält der Pentium zwei 8 KBytes große Cache-Speicher zur getrennten Speicherung von Befehlen und Daten. Die Speicherverwaltungseinheiten arbeiten wie beim i486.

Der leistungsfähigste Prozessor der Intel 80x86-Reihe ist der Pentium Pro Mikroprozessor, der durch seine Organisation eine hohe parallele Abarbeitung der Maschinenbefehle erreicht.

3.1.2 Programmiermodell

Die nachfolgende Beschreibung geht vom Programmiermodell des 8086 Prozessors aus. Die Erweiterungen der 32-Bit Mikroprozessoren[1] werden gesondert aufgeführt.

3.1.2.1 Der Registersatz des 8086

Die dem Maschinenprogrammierer zugänglichen Register können in folgende Gruppen aufgeteilt werden:

- Allzweckregister (AX, BX, CX, DX),

[1] Diese Register des 8086 sind diejenigen, welche auch die Prozessoren 80286, i386 etc. im Real Mode verwenden.

- Indexregister,
- Zeigerregister,
- Segmentregister und
- Statusregister.

Die vier 16-Bit **Allzweckregister AX, BX, CX und DX** dienen zur Speicherung von Operanden der arithmetischen und logischen Operationen sowie der Schiebe-Operationen. Die Register haben darüber hinaus für viele Befehle eine implizite Bedeutung.

Das AX-Register dient beispielsweise als Akkumulator bei der Multiplikation oder Division sowie bei String-Operationen und als Ein-/Ausgaberegister für I/O-Ports.

Das BX-Register wird als Basisregister für Sepicherzugriffe in einem Datensegment verwendet.

Eine Reihe von Schleifenbefehlen und String-Operationen verwenden implizit das CX-Register als Zählerregister.

Das DX-Register dient als Datenregister bei der Multiplikation oder Division und als Adreßregister für I/O-Ports mit Adressen, die größer als 255 sind.

Die höher- und niederwertigen Bytes dieser Allzweckregister können explizit verwendet werden, wobei anstelle des X ein H für das höherwertige Byte und L für das niederwertige Byte geschrieben werden muß. Beipielsweise läßt sich der höherwertige Teil des AX-Registers mit AH und der dessen niederwertige Teil mit AL ansprechen.

Die **Indexregister SI und DI** dienen vornehmlich zur Adressierung von Quell- und Zieloperanden bei Stringbefehlen, können aber auch bei Transportbefehlen verwendet werden, wobei sich der Zugriff implizit auf das Datensegment bezieht.

Die **Zeigerregister IP, SP und BP** dienen zur Adressierung innerhalb eines Segments und sind daher dem Anwendungs-Programmierer ebenso zugänglich, wobei auch hier eine besondere Aufgabenteilung zu beachten ist. So wird das IP-Register (=instruction pointer) ausschließlich zum Lesen von Maschinencodes aus dem Code-Segment verwendet, wohingegen die Register SP und BP zum Adressieren des Stacks dienen.

Das **Flag-Register** oder auch Status-Register enthält verschiedene Bits, die den Zustand der CPU anzeigen. Dabei ist zu unterscheiden zwischen den Kontrollflags, welche bestimmte Betriebsmodi der CPU aktivieren und den Bedingungsflags, die bestimmte Eigenschaften von Bearbeitungsergebnissen anzeigen. Das Flag-Register ist 16 Bit breit und hat folgenden Aufbau:

Tabelle 3.1: Aufbau des Flag-Registers

Bit15															Bit0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

Die Bedingungsflags können als Bedingungen für Sprungbefehle verwendet werden. Zu ihnen gehören:

- CF

  Die CPU setet das *Carry-Flag* wird gesetzt, wenn bei einer Addition ein Überlauf vom höchstwertigen Bit in das nächsthöhere, nicht vorhandene Bit stattfindet, während es sonst gelöscht wird. Bei einer Subtraktion zeigt ein gesetztes CF einen Unterlauf an. Der Programmierer benötigt das CF zum Erkennen von Wertebereichsüberschreitungen bei Rechnungen mit vorzeichenlosen ganzen Zahlen.

  (**ACHTUNG!** Zählbefehle beeinflussen das Carry-Flag NICHT !)

  Außerdem nimmt das CF bei Schiebe- und Rotations-Befehlen das herausgeschobene Bit auf und kann je nach Befehl in die freie Stelle übernommen werden.

Obwohl das CF ein Bedingungs-Flag ist, kann es der Programmierer direkt setzen oder löschen.

- PF
  Das *Parity-Flag* wird gesetzt, wenn das Ergebnis einer arithmetischen oder logischen Operation eine gerade Zahl von Eins-Bits enthält.

- AF
  Das *Auxilliary-Carry-Flag* zeigt den Übertrag von Bit3 nach Bit4 bei einer arithmetischen 8-Bit-Operation an. Es dient zur BCD-Korrektur des AL-Registers nach einer Addition, so daß es der Programmierer in der Regel nicht direkt abfragt oder verändert.

- ZF
  Wenn das *Zero-Flag* gesetzt ist, dann war das Ergebnis der letzen Operation Null. Es gibt jedoch einige Befehle (z.B. loop), die das ZF nicht beeinflussen, wenn das CX-Register den Wert Null erreicht hat.

- SF
  Das *Sign-Flag* oder Vorzeichen-Flag wird gesetzt, wenn das höchstwertige Bit des letzten Operationsergebnises gesetzt ist (Bit7 bei 8-Bit- und Bit15 bei 16-Bit-Operationen). Bei Rechnung mit Komplementdarstellung zeigt es das Vorzeichen des Ergebnisses an (0 für positiv; 1 für negativ).

- OF
  Das *Overflow-Flag* zeigt eine Wertebereichsüber- bzw. Unterschreitung bei Rechnungen mit vorzeichenbehafteten ganzen Zahlen an. Die CPU bildet es aus der Exklusiv-Oder-Verknüpfung des Carry-Flags mit dem Übertrag von der zweithöchsten in die höchste Stelle und darf daher nicht mit dem Carry-Flag verwechselt werden!

Die Kontroll-Flags dienen zur Auswahl bestimmter Betriebsmodi. Zu ihnen gehören:

- TF
  Wenn das *Trap-Flag* gesetzt ist, dann löst der 80x86 nach jedem Maschinenbefehl einen Interrupt 01 aus, um Programme im Einzelschritt-Modus testen zu können. Es gibt keine Befehle zum Setzen oder löschen des T-Flags.

- DF
  Das *Direction-Flag* bestimmt bei wiederholten Stringbefehlen (Wiederholungs-Präfix: REP) ob die Index-Register SI und DI erhöht oder erniedrigt werden.

DF=0:	SI und DI erhöhen
DF=1:	SI und DI erniedrigen

Ab dem 80386 stehen dem Programmierer 32 Bit breite Register zur Verfügung. Diese Register können auch im Real-Mode verwendet werden, wenn im Assembler-Programm der Befehl ".386" eingesetzt ist. Die 32-Bit-Register haben die gleichen Aufgaben wie die 16-Bit-Register, wobei dem Namen lediglich ein "E" voranzustellen ist. Die 16-Bit-Register können nach wie vor mit den gewohnten Namen verwendet werden, was den Zugriff auf die niederwertigen 16-Bit eines 32-Bit-Registers erlaubt. Die 32-Bit-Register lassen sich ähnlich unterteilen wie die 16-Bit-Register:

1. Die 32-Bit-Allzweck-Register
   - EAX (AX = Low-Word von EAX)
   - EBX (BX = Low-Word von EBX)
   - ECX (CX = Low-Word von ECX)
   - EDX (DX = Low-Word von EDX)
2. Die 32-Bit-Index-Register

- ESI (SI = Low-Word von ESI)
- EDI (DI = Low-Word von EDI)

3. Die 32-Bit-Zeiger-Register
   - EBP (BP = Low-Word von EBP)
   - EIP (IP = Low-Word von EIP)
   - ESP (SP = Low-Word von ESP)
4. Das 32-Bit-Flag-Register
   - EFLAG (FLAG = Low-Word von EFLAG)
     Dieses Register enthält einige neue Flags, die hier jedoch nicht erwähnt sind.

Es gibt noch weitere 32-Bit-Register, wie zum Beispiel die Steuer-Register, die Test-Register und die Debug-Register. Diese Register werden hier jedoch nicht weiter behandelt, da der Anwendungs-Programmierer nicht darauf zugreiffen kann.

3.1.2.2 Die Betriebs-Modi der 80x86-Prozessoren

Die Speichersegmentierung im Real-Mode Der 80x86 kann im Real Mode 2^{20} Bytes (=1 MByte) adressieren, hat jedoch 16 Bit breite Adress-Register, die nur für einen 64 KByte großen Speicher ausreichen würden. Um dennoch den ganzen Adreßraum von 1MByte ansprechen zu können, muß dieser in mehrere 64 KByte große Segmente zerlegt werden, wobei zur Auswahl des Segments eines der vier Segmentregister erforderlich ist. Das Zeigerregister gibt dann die Offset-Adresse innerhalb des aktuellen Speichersegments an. Die Berechnung der physikalischen Speicheradresse ergibt sich aus folgendem Zusammenhang:

$$\text{phys. Adresse} = 16 \times \text{Segment-Adresse} + \text{Offset-Adresse} \tag{3.1}$$

Da die Segment-Adresse in einem 16 Bit großen Segment-Register liegt und diese mit 16 multipliziert wird, was 4 Linksschiebeschritten entspricht, ergibt sich:

1. Die physikalische Adresse ist 20 Bit breit (wie Adreßbus)
2. Die Segmente beginnen auf durch 16 teilbaren Adressen und überlappen sich daher.

Der 80x86 verwendet für verschiedene Arten von Speicherzugriffen eigene Segment-Register:

- Für Code-Zugriffe verwendet er das CS-Rgister. Die phys. Adresse ergibt sich aus:

$$\text{phys. Adresse} = 16 \times \text{CS} + \text{IP} \tag{3.2}$$

- Für Daten-Zugriffe verwendet er das DS-Register. Es ergibt sich:

$$\text{phys. Adresse} = 16 \times \text{DS} + \text{Offset-Adresse} \tag{3.3}$$

Die Offset-Adresse kann hierbei auf vielfältige Weise gewonnen werden. Für Datenzugriffe steht als Alternative noch das ES-Register zur Verfügung. Der 80386 und seine Nachfolger bieten zusätzlich noch die Register FS und GS.

- Für Zugriffe auf den Stack verwendet der 80x86 das SS-Register. Hier gibt es zwei Möglichkeiten:

$$\text{phys. Adresse} = 16 \times \text{SS} + \text{SP} \tag{3.4}$$

oder:

$$\text{phys. Adresse} = 16 \times \text{SS} + \text{BP} + \text{Displacement} \tag{3.5}$$

Die letzte Möglichkeit findet vor allem in Hochsprachen wie C oder PASCAL Verwendung um die Parameterübergabe an Unterprogramme zu bewerkstelligen und lokale Variablen zu adressieren. Auch die Nachfolgeprozessoren 80286, 80386, 80486 und Pentium adressieren den Arbeitsspeicher in der oben genannten Weise, sofern sie im Real- oder im Virtual-Mode betrieben werden.

Die Adressierung im Protected Mode Auch im *Protected Mode* setzt sich eine Adresse aus zwei Teilen zusammen, einem Offset und einem *Selektor*. Den Offset ermittelt der Prozessor mit den selben Adressierungsarten wie im Real Mode. Im Protected Mode verwendet er jedoch statt einer Segment-Adresse einen 16 Bit breiten *Selektor*, der wie eine Segment-Adresse in einem der Segmentregister gespeichert ist. Ein jeder Selektor besteht aus drei Teilen, dem 13 Bit breiten Index-Feld, dem 2 Bit breiten RPL-Feld und dem TI-Flag. Das Index-Feld gibt die Nummer eines Informations-Blocks (= Deskriptor) an, der sich in einer Tabelle im Arbeitsspeicher befindet und Informationen über Größe und Lage des Segments innerhalb des linearen Adressraumes enthält.

Der Anwendungs-Programmierer braucht sich jedoch nicht um die Verwaltung und den Aufbau von Selektoren, Deskriptoren und Deskriptortabellen zu kümmern, da hierfür die Betriebssysteme zuständig sind. Näheres zu diesem Themenbereich entnehmen Sie bitte der entsprechenden Literatur.

3.1.2.3 Die Adressierungsarten

Die Register-Adressierung Der Operand befindet sich in einem Register.

Beispiel (nicht für Turbo-Assembler):

```
INC CX              ;inkrementiere den Inhalt des CX-Registers
```

Die unmittelbare Adressierung Bei der unmittelbaren Adressierung ist das Datum im Maschinenbefehl enthalten.

Beispiel (nicht für Turbo-Assembler):

```
ADD AX,100          ;addiere 100 zum Inhalt des AX-Registers
```

Die direkte Adressierung Hier ist die Offset-Adresse derjenigen Speicherstelle anzugeben, die das geforderte Datum enthält. Die Segment-Adresse wird normalerweise dem DS-Register entnommen (Ausnahmen siehe SEG-Befehl).

Beispiel (für Turbo Assembler):

```
.model small
.data               ;Beginn des Daten-Segments
zaehler db 100      ;Ein Speicher-Byte reservieren,
                    ; mit 100 initialisieren und den
                    ; Symbolnamen "'zaehler"' zuordnen.
                    ;
.stack 100h         ;Stack-Segment anlegen
                    ;
.code               ;Beginn des Code-Segments
DEC zaehler         ;Speicher-Byte "Zaehler" dekrementieren
                    ; "zaehler" enthält nachher 99
end
```

Die indirekte Adressierung Im SI-, DI-, BX- oder BP-Register steht die Offset-Adresse des gewünschten Datums. Die Segment-Adresse wird normalerweise dem DS-Register entnommen. Falls jedoch BP angegeben ist, dann verwendet der 80x86 das SS-Register. (siehe auch SEG-Befehl zur Auswahl anderer Segment-Register)

Beispiel (für Turbo Assembler):

```
.model small
.data               ;Beginn des Daten-Segments
zahl db 10          ;Ein Speicher-Byte reservieren,
                    ; mit 10 initialisieren und
                    ; Symbolnamen "zahl" zuordnen
                    ;
.code               ;Beginn des Code-Segments
                    ;
.stack 100h         ;Stack-Segment anlegen
                    ;
MOV SI,offset zahl  ;SI wird mit der Offset-Adresse des
                    ; Speicher-Bytes "zahl" geladen.
MOV AL,[SI]         ;AL wird mit dem Inhalt desjenigen
                    ; Speicher-Bytes geladen, dessen Offset in SI
                    ; steht.
                    ; Das AL-Register enthält nach Ausführung
                    ; der beiden Befehle den Wert 10.
end
```

Die basisrelative Adressierung Die Offset-Adresse wird gewonnen aus dem Inhalt des BX- oder BP-Registers, addiert zu einem Displacement, welches im Befehl als 8-Bit- oder als 16-Bit-Konstante enthalten ist. Falls das BP-Register angegeben ist, dann bezieht sich die Offset-Adresse auf das Stack-Segment ansonsten auf das Daten-Segment.

Beispiel (für Turbo Assembler):

```
.model small
.data                ;Beginn des Daten-Segments
liste db 120,10,93,18,7,60,5,42,33,0,12
                     ;Tabelle mit mehreren Speicher-Bytes reservieren,
                     ; initialisieren und den Namen "'liste"' zuordnen.
                     ;
.stack 100h          ;Stack-Segment anlegen
                     ;
.code                ;Beginn des Code-Segments
MOV BX,offset liste  ;Offset-Adresse des Anfangs der Tabelle "liste"
                     ; nach BX laden
MOV AL,[BX+4]        ;Offset ergibt sich aus dem Inhalt von BX plus 4,
                     ; Inhalt dieses Speicherbytes nach AL laden
                     ;Das AL-Register enthält nach Ausführung
                     ; der beiden Befehle das 5. Byte (=Byte 4)
                     ; der Tabelle, also den Wert 7.
end
```

Die indizierte Adressierung Hier wird die Offset-Adresse durch Addition eines konstanten 8-Bit- oder 16-Bit-Displacements zum Inhalt des DI- oder SI-Registers ermittelt. Die Segment-Adresse steht im DS-Register.

Beispiel (für Turbo Assembler):

```
.model small
.data                ;Anfang des Daten-Segments
liste db 120,10,93,18,7,60,5,42,33,0,12
                     ;Tabelle mit mehreren Speicher-Bytes reservieren,
                     ; initialisieren und den Namen "liste" zuordnen.
                     ;
.stack 100h          ;Stack-Segment anlegen
                     ;
.code                ;Anfang des Code-Segments
MOV SI,3             ;3 in das SI-Register laden.
MOV AL,liste[SI]     ;Offset ergibt sich aus dem Inhalt von SI plus
                     ; Offset von "liste" (erstes Byte)
                     ;Es wird das 4. Byte (=Byte 3) der
                     ; Tabelle "liste" geladen.
                     ;Das AL-Register enthält nach Ausführung der
                     ; beiden Befehle den Wert 18
end
```

Die basisindizierte Adressierung Die Offset-Adresse des Datums entsteht aus der Summe eines konstanten Displacements, dem Inhalt eines Basisregisters (BP oder BX) und dem Inhalt eines Indexregisters (SI oder DI). Ist das BP-Register angegeben, dann wird auf das Stack-Segment zugegriffen, beim BX-Register auf das Daten-Segment.

Beispiel (für Turbo Assembler):

```
.model small
.data               ;Anfang des Daten-Segments
liste db 120,10,93,18,7,60,5,42,33,0,12
                    ;Tabelle mit mehreren Speicher-Bytes reservieren,
                    ; initialisieren und den Namen "'liste"' zuordnen.
                    ;
.stack              ;Stack-Segment anlegen
                    ;
.code               ;Anfang des Code-Segments
MOV SI,5            ;5 in das SI-Register laden
MOV BX,3            ;3 in das BX-Register laden
MOV AL,liste[BP][SI] ;Der Offset ergibt sich aus dem Inhalt von BP plus
                    ; Inhalt von SI plus Offset des ersten Bytes der
                    ; Tabelle "liste"
                    ;Es wird das 9. Byte (=Byte 8) der
                    ;Tabelle "liste" geladen.
                    ;Das AL-Register enthält nach Ausführung der
                    ; beiden Befehle den Wert 33.
end
```

Die skalierte Adressierung Die skalierte Adressierung ist beim 80386 und seinen Nachfolgern implementiert. Die Offset-Adresse ergibt sich aus dem Inhalt eines Basis-Registers plus dem eines Index-Registers mal Skalierungsfaktor plus einem konstanten Displacement, wobei als der Skalierungsfaktor 2, 4 oder 8 möglich ist. Es ist egal welches Register als Index- und welches als Basis-Register verwendet wird, jedoch greift die CPU auf das Stack-Segment zu, wenn EBP als Basis-Register angegeben ist.

Beispiel (für Turbo Assembler):

```
.MODEL small
                    ;
.DATA               ;Datensegment anlegen
tabelle
DB 11h,12h,13h,14h  ;erste Zeile der Tabelle
DB 21h,22h,23h,24h  ;zweite Zeile der Tabelle
DB 31h,32h,33h,34h  ;dritte Zeile der Tabelle
.STACK 100h         ;Stack-Segment anlegen
.CODE               ;Codesegment anlegen
.386                ;80386-Befehle ermöglichen
start:
mov ax,@data        ;Segment-Adresse des Datensegments
mov ds,ax
                    ;
mov ebx,offset tabelle ;Anfang der Tabelle ins Basis-Register
                    ;
mov esi,0           ;erste Zeile der Tabelle auswählen
mov al,[ebx+esi*4+1] ;Zahl 12h wird geladen
mov al,[ebx+esi*4+3] ;Zahl 14h wird geladen
                    ;
mov esi,2           ;dritte Zeile der Tabelle auswählen
mov al,[ebx+esi*4+0] ;Zahl 31h wird geladen
mov al,[ebx+esi*4+1] ;Zahl 32h wird geladen
                    ;
                    ;Ende des Programms
end
```

3.1.3 Die Befehle

3.1.3.1 Nach Verwendung Geordnet (nur Kurzbeschreibung)

Arithmetische Befehle

AAA	Korrektur einer ungepackten BCD-Zahl nach Addition
AAD	Korrektur einer ungepackten BCD-Zahl vor Division
AAM	Korrektur einer ungepackten BCD-Zahl nach Multiplikation
AAS	Korrektur einer ungepackten BCD-Zahl nach Subtraktion
ADC	addiert zwei Operanden und das Carry-Flag
ADD	addiert zwei Operanden
CBW	wandelt ein Byte vorzeichenrichtig in ein Wort um
CDQ	wandelt ein Doppelwort in ein Vierfachwort um (ab 80386)
CWD	wandelt ein Wort in ein Doppelwort um
DAA	Korrektur einer gepackten BCD-Zahl nach Addition
DAS	Korrektur einer gepackten BCD-Zahl nach Subtraktion
DEC	vermindert den Operanden um 1
DIV	dividiert vorzeichenlose ganze Zahlen
IDIV	dividiert vorzeichenbehaftete ganze Zahlen
IMUL	multipliziert vorzeichenbehaftete ganze Zahlen
INC	erhöht den Operanden um 1
MUL	multipliziert vorzeichenlose ganze Zahlen
NEG	bildet das Zweierkomplement des Operanden
NOT	bildet das Einerkomplement des Operanden
SBB	subtrahiert den Quell-Operand und das Carry-Flag vom Ziel-Operand
SUB	subtrahiert den Quell-Operand vom Ziel-Operand

Ein-/Ausgabe-Befehle

IN	liest einen Peripherie-Port
INS	Eingabe eines Strings über einen Eingabeport (ab 80186)
INSB	String byteweise über Eingabeport einlesen (ab 80186)
INSW	String wortweise über Eingabeport einlesen (ab 80186)
INSD	String doppelwortweise über Eingabeport einlesen (ab 80386)
OUT	schreibt einen Peripherie-Port
OUTS	Ausgabe eines Strings über einen Ausgabeport (ab 80186)
OUTSB	String byteweise auf Ausgabeport schreiben (ab 80186)
OUTSW	String wortweise auf Ausgabeport schreiben (ab 80186)
OUTSD	String doppelwortweise auf Ausgabeport schreiben (ab 80386)

Logische Befehle

AND	bitweise UND-Verknüpfung zweier Operanden
OR	bitweise ODER-Verknüpfung zweier Operanden
TEST	bitweise UND-Verknüpfung ohne die Operanden zu verändern
XOR	bitweise EXKLUSIV-ODER-Verknüpfung zweier Operanden

Präfixe

REP	wiederhole den folgenden Stringbefehl solange CX≠0
REPE	wiederhole den folgenden Stringbefehl solange CX≠0 und ZF=1
REPZ	wiederhole den folgenden Stringbefehl solange CX≠0 und ZF=1
REPNE	wiederhole den folgenden Stringbefehl solange CX≠0 und ZF=0
REPNZ	wiederhole den folgenden Stringbefehl solange CX≠0 und ZF=0
SEG	anderes Segmentregister verwenden

Protected-Mode-Befehle

ARPL	vergleicht die Selektor-Parameter von Unterprogramm und Benutzer
CLTS	lösche das Task-Schalter-Bit (ab 80286)
LAR	lade Zugriffsberechtigungsbyte (ab 80286)
LGDT	lade das Globale-Deskriptor-Tabellen-Register (ab 80286)
LIDT	lade das Interrupt-Deskriptor-Tabellen-Register (ab 80286)
LLDT	lade das Lokale-Deskriptor-Tabellen-Register (ab 80286)
LMSW	lade das Maschinen-Status-Wort (ab 80286)
LSL	lade Segmentgrenze (ab 80286)
LTR	lade Task-Register (ab 80286)
SIDT	speichere das Interrupt-Deskriptor-Tabellen-Register (ab 80286)
SLDT	speichere das Lokale-Deskriptor-Tabellen-Register (ab 80286)
SMSW	speichere das Maschinen-Status-Wort (ab 80286)
STR	speichere das Task-Register (ab 80286)
VERR	untersuche ein Segment auf Lesbarkeit (ab 80286)
VERW	untersuche ein Segment auf Schreibbarkeit (ab 80286)

Transportbefehle

LAHF lade Bedingungs-Flags nach AH	
LDS	lade Far-Pointer nach Register und DS
LEA	lade effektive Adresse
LES	lade Far-Pointer nach Register und ES
LODS	lade String
LODSB	lade Byte eines Strings
LODSW	lade Wort eines Strings
LODSD	lade Doppelwort eines Strings
MOV	kopiere Daten
MOVS	kopiere String
MOVSB	kopiere String byteweise
MOVSW	kopiere String wortweise
MOVSD	kopiere String doppelwortweise (ab 80386)
SAHF	kopiere AH in Bedingungsflags
STOS	speichert Stringdaten
STOSB	speichert Stringdaten byteweise
STOSW	speichert Stringdaten wortweise
STOSD	speichert Stringdaten doppelwortweise (ab 80386)
XCHG	vertauscht die Operanden
XLAT	Adressierung eines Tabellenelements

Schiebe- und Rotations-Befehle

RCL	rotiert den Operanden nach links durch das Carry-Flag
RCR	rotiert den Operanden nach rechts durch das Carry-Flag
ROL	rotiert den Operanden nach links
ROR	rotiert den Operanden nach rechts
SAL	schiebt den Operanden arithmetisch nach links
SHL	schiebt den Operanden logisch nach links
SAR	schiebt den Operanden arithmetisch nach rechts
SHR	schiebt den Operanden logisch nach rechts

Sprungbefehle

CALL	Aufruf eines Unterprogramms
CALLN	Aufruf eines Near-Unterprogramms (in gleichem Segment)
CALLF	Aufruf eines Far-Unterprogramms (in anderem Segment)
INT	Aufruf eines Interrupt-Programms
INTO	Aufruf eines Interrupt-Programms
IRET	Rückkehr von einem Interrupt-Programm
Jxxx	springe, wenn Bedingung xxx erfüllt
JCXZ	springe, wenn CX=0
JMP	springe immer
LOOP	dekrementiert CX, springt wenn CX nicht 0
LOOPE	dekrementiert CX, springt wenn CX nicht 0 und altes ZF=1
LOOPNE	dekrementiert CX, springt wenn CX nicht 0 und altes ZF=0
LOOPZ	dekrementiert CX, springt wenn CX nicht 0 und altes ZF=1
LOOPNZ	dekrementiert CX, springt wenn CX nicht 0 und altes ZF=0
RET	Rücksprung von Unterprogramm
RETN	Rücksprung von Near-Unterprogramm (in gleichem Segment)
RETF	Rücksprung von Far-Unterprogramm (in anderem Segment)

Stackbefehle

ENTER	reserviert Speicher auf dem Stack für lokale Variablen (ab 186/286)
LEAVE	gibt den mit ENTER reservierten Speicher wieder frei (ab 80186/286)
POP	holt ein Wort vom Stack
POPA	holt DI, SI, BP, SP, BX, DX, CX und AX vom Stack (ab 80186)
POPF	holt das Flag-Register vom Stack
PUSH	legt ein Wort auf den Stack
PUSHA	legt AX, CX, DX, BX, SP, BP, SI und DI auf den Stack (ab 80186)
PUSHF	legt das Flag-Register auf den Stack

Steuerbefehle

CLC	Carry-Flag löschen
CLD	Direction-Flag löschen

CLI	Interrupt-Flag löschen (Interrupts sperren)
CMC	Carry-Flag invertieren
ESC	Code und Speicheradressierung für Co-Prozessor
HLT	Anhalten bis Interrupt
LOCK	Zugriff auf bestimmte Speicherstellen sperren
NOP	keine Aktion
STC	Carry-Flag setzen
STI	Interrupt-Flag setzen (Interrupts zulassen)
WAIT	wartet bis Busy-Pin=High

Vergleichsbefehle

BOUND	vergleicht Near-Pointer mit Anfangs- und Endadresse einer Tabelle (ab 80186 und 80286)
CMP	vergleicht zwei Operanden miteinander
CMPS	vergleicht String-Operanden
CMPSB	vergleicht String-Operanden byteweise
CMPSW	vergleicht String-Operanden wortweise
CMPSD	vergleicht String-Operanden doppelwortweise (ab 80386)
SCAS	vergleicht den mit ES:DI adressierten String mit einem Register
SCASB	vergleicht String byteweise mit einem Register
SCASW	vergleicht String wortweise mit einem Register
SCASD	vergleicht String doppelwortweise mit einem Register (ab 80386)

Wiederholungsbefehle

LOOP	dekrementiert CX, springt wenn CX nicht 0
LOOPE	dekrementiert CX, springt wenn CX nicht 0 und altes ZF=1
LOOPNE	dekrementiert CX, springt wenn CX nicht 0 und altes ZF=0
LOOPZ	dekrementiert CX, springt wenn CX nicht 0 und altes ZF=1
LOOPNZ	dekrementiert CX, springt wenn CX nicht 0 und altes ZF=0
REP	wiederhole den folgenden Stringbefehl solange CX≠0
REPE	wiederhole den folgenden Stringbefehl solange CX≠0 und ZF=1
REPZ	wiederhole den folgenden Stringbefehl solange CX≠0 und ZF=1
REPNE	wiederhole den folgenden Stringbefehl solange CX≠0 und ZF=0
REPNZ	wiederhole den folgenden Stringbefehl solange CX≠0 und ZF=0

Stringbefehle

CMPS	vergleicht String-Operanden
CMPSB	vergleicht String-Operanden byteweise
CMPSW	vergleicht String-Operanden wortweise
CMPSD	vergleicht String-Operanden doppelwortweise (ab 80386)
INS	liest ein String-Element von einem E/A-Port
INSB	liest ein Byte eines Strings von einem E/A-Port
INSW	liest ein Wort eines Strings von einem E/A-Port
INSD	liest ein Doppel-Wort eines Strings von einem E/A-Port
LODS	lade String

LODSB	lade Byte eines Strings
LODSW	lade Wort eines Strings
LODSD	lade Doppel-Wort eines Strings
MOVS	kopiere String
MOVSB	kopiere String byteweise
MOVSW	kopiere String wortweise
MOVSD	kopiere String doppelwortweise (ab 80386)
OUTS	schreibt ein String-Element auf einen E/A-Port
OUTSB	schreibt ein Byte eines Strings auf einen E/A-Port
OUTSW	schreibt ein Wort eines Strings auf einen E/A-Port
OUTSD	schreibt ein Doppel-Wort eines Strings auf einen E/A-Port
SCAS	vergleicht den mit ES:DI adressierten String mit einem Register
SCASB	vergleicht String byteweise mit einem Register
SCASW	vergleicht String wortweise mit einem Register
SCASD	vergleicht String doppelwortweise mit einem Register (ab 80386)
STOS	speichert Stringdaten
STOSB	speichert Stringdaten byteweise
STOSW	speichert Stringdaten wortweise
STOSD	speichert Stringdaten doppelwortweise (ab 80386)

3.1.3.2 Alphabetisch Geordnet (ausführliche Beschreibung)

In dieser Befehlsbeschreibung werden folgende Abkürzungen verwendet:

Tabelle 3.2: Abkürzungen für Änderungen der Flags

?	Flag-Zustand undefiniert
X	Flag wird gesetzt oder gelöscht je nach Ergebnis
0	Flag wird gelöscht
1	Flag wird gesetzt
-	Flag wird nicht verändert

Befehle

AAA (adjust AL after addition)

Nach der Addition zweier ungepackter BCD-Ziffern (einstellige Dezimalzahl!) mit 8-Bit-Additionsbefehlen muß das Ergebnis im AL-Register mit dem AAA-Befehl wieder in ungepackten BCD-Code umgewandelt werden. Der AAA-Befehl addiert 6 zum AL-Register, falls AL eine Pseudotetrade (Zahlen von 10 bis 15) enthält, oder das A-Flag gesetzt ist. Zusätzlich werden dann die vier höherwertigen Bits von AL gelöscht und AH um 1 erhöht. AX enthält nachher eine ungepackte BCD-Zahl, die aus zwei Ziffern besteht (die niederwertige in AL, die höherwertige in AH) und der Summe der vorangehenden Addition entspricht.

```
Flags:    O  D  I  T  S  Z  A  P  C
          ?  -  -  -  ?  ?  X  ?  X
Syntax:   AAA
Dauer:    4 Zyklen
```

AAD (adjust AX before division)

Wandelt die ungepackte BCD-Zahl im AX-Register (Einer-Stelle in AL, Zehner-Stelle in AH) in eine Dualzahl um. Dabei wird AH mit 10 multipliziert und zu AL addiert. Dieser Befehl ist vor der Division von BCD-kodierten Zahlen anzuwenden.

```
Flags:    O D I T S Z A P C
          ? - - - X X ? X ?
Syntax:   AAD
Dauer:    19 Zyklen
```

AAM (adjust AX after multiply)

Wandelt die Dualzahl im AL-Register in eine ungepackte BCD-Zahl um. Hierfür dividiert der 80x86 den Inhalt von AL durch 10 und speichert den Quotienten in AH und den Rest in AL. AAM bewirkt das Gegenteil von AAD und wird nach der Multiplikation von BCD-Ziffern angewendet.

```
Flags:    O D I T S Z A P C
          ? - - - X X ? X ?
Syntax:   AAM
Dauer:    17 Zyklen
```

AAS (adjust AL after subtraction)

AAS wird verwendet, um nach der Subtraktion zweier BCD-Ziffern das Ergebnis im AL-Register zu korrigieren. Dabei wird 6 von AL subtrahiert, falls das Ergebnis eine Pseudotetrade (10 bis 15) war, oder das A-Flag gesetzt war. Die höherwertigen vier Bits von AL werden dann gelöscht und das AH-Register dekrementiert.

```
Flags:    O D I T S Z A P C
          ? - - - ? ? X ? X
Syntax:   AAS
```

ADC (add with carry)

Addiert zum Zieloperanden den Quelloperanden und das Carry-Flag.

```
Flags:    O D I T S Z A P C
          X - - - X X X X X
Syntax:   ADC Ziel,Quelle
Dauer:    2 bis 7 Zyklen
```

ADD (add)

Addiert den Quelloperanden zum Zieloperanden.

```
Flags:    O D I T S Z A P C
          X - - - X X X X X
Syntax:   ADD Ziel,Quelle
Dauer:    2 bis 7 Zyklen
```

AND (logical and)

Führt eine bitweise Und-Verknüpfung von Quell-Operand und Zieloperand durch, das Ergebnis wird im Zieloperand gespeichert.

```
Flags:    O D I T S Z A P C
          0 - - - X X ? X 0
Syntax:   AND Ziel,Quelle
```

ARPL (adjust required privilege level field of selector) (ab 80286)

Vergleicht User-CS-Selektor und Call-CS-Selektor miteinander. Ist das RPL-Feld des User-CS-Selektors kleiner oder gleich dem Call-CS-Selektor (höhere Proirität des Benutzers), dann wird das RPL-Feld des Call-CS-Selektors in das des Benutzer-CS-Selektors kopiert und das Zero-Flag gesetzt. Dieser Befehl wirkt nur im Protected Mode und steht erst ab 80286 zur Verfügung.

```
Flags:    O D I T S Z A P C
          - - - - - X - - -
Syntax:   ARPL User-CS-Selektor,Call-CS-Selektor
Dauer:
```

BOUND (check array index against bounds) (ab 80186)

Vergleicht den Inhalt eines Registers (Offset-Adresse eines Tabellen-Elements) mit der Anfangs-Offset-Adresse und der End-Offset-Adresse der Tabelle. Ist der Register-Inhalt kleiner als die Anfangsadresse, oder größer als die Endadresse, so wird ein Interrupt 5 ausgelöst. Anfangs- und Endadresse liegen dabei im Speicher, sie werden meist an den Tabellenanfang gesetzt.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   BOUND Register,Tabellengrenzen
```

CALL (call procedure)

Ruft ein Unterprogramm auf. Dabei wird zuerst die Rücksprungadresse (Adresse des folgenden Befehls) auf den Stack gelegt, wobei der Stack-Poiner um 2 oder 4 erniedrigt wird. Anschließend wird das IP-Register (instruction pointer) mit der Offset-Adresse des Unterprogramms geladen und die Bearbeitung an der neuen Adresse fortgesetzt. Es gibt zwei verschiedene CALLs, den CALL-Befehl für Near-Prodeduren und den CALL-Befehl für Far-Prozeduren.

Der Near-CALL kann nur Unterprogramme aufrufen, die im selben Segment liegen wie die aufrufende Prozedur, so daß für das Sprungziel nur die Offset-Adresse (2 Bytes) anzugeben ist. Da auch der Rücksprung innerhalb desselben Segments stattfindet, legt die CPU nur der Offset der Rücksprungadresse auf den Stack.

Mit dem Far-CALL sind Unterprogramme aus allen Segmenten des Speichers erreichbar, so daß es nötig ist, auch die Segment-Adresse des Sprungziels anzugeben. Da auch der Rücksprung über Segmentgrenzen hinweg erfolgt, wird beim Far-CALL zunächst die Segment-Adresse und anschließend die Offset-Adresse des auf den CALL folgenden Befehls auf den Stack gelegt. Hierbei werden vier Bytes auf dem Stack gespeichert und der Stack-Pointer um 4 erniedrigt.

Da je nach Art des Unterprogrammaufrufs (Near oder Far) die Rücksprungadresse in verschiedenen Größen (2 Bytes oder 4 Bytes) auf dem Stack liegt, gibt es auch zwei verschiedene Rücksprungbefehle (RETN und RETF). Der Assemblerprogrammierer muß sich daher entscheiden, ob ein Unterprogramm nur mit Near- oder nur mit Far-CALLs aufzurufen ist und dementsprechend den richtigen Rücksprungbefehl an das Ende der Prozedur setzen.

Diese Überlegungen werden dem Programmierer von den meisten Assemblern wie MASM oder TASM abgenommen. Hier kann man an den Prozedurkopf (PROC-Befehl) die Bezeichnung NEAR oder FAR anhängen. Der Assembler setzt dann statt RET entweder RETN oder RETF ein und verwendet bei allen Aufrufen dieses Unterprogramms den dazu passenden CALL-Befehl. Die Angabe NEAR oder FAR kann auch weggelassen werden. In diesem Fall verwendet der Assembler die zum gewählten Speichermodell passenden CALLs. Bei den Modellen TINY, SMALL und COMPACT erzeugt der Assembler Near-CALL's, wenn nichts anderes angegeben ist. Bei den Modellen MEDIUM, LARGE und HUGE werden Far-Calls angewendet.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   CALL Sprungziel
Dauer:    3 bis 275 Zyklen
```

CBW (convert byte into word)

Erweitert das in AL befindliche Byte vorzeichenrichtig auf Wortlänge. Dabei schreibt der 80x86 0ffh in das AH-Register falls Bit 7 (MSB) gesetzt war, ansonsten löscht er das AH-Register.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   CBW
Dauer:    3 Zyklen
```

CLC (clear carry flag)

Löscht das Carry-Flag.

```
Flags:    O D I T S Z A P C
          - - - - - - - - 0
Syntax:   CLC
Dauer:    2 Zyklen
```

CLD (clear direction flag)

Löscht das Direction-Flag. Die folgenden Stringbefehle, die SI und/oder DI verwenden erhöhen diese Register.

```
Flags:    O D I T S Z A P C
          - 0 - - - - - - -
Syntax:   CLD
Dauer:    2 Zyklen
```

CLI (clear interrupt flag)

Löscht das Interrupt-Flag. Die CPU nimmt danach keine maskierbaren Interrupts mehr an.

```
Flags:    O D I T S Z A P C
          - - 0 - - - - - -
Syntax:   CLI
Dauer:    2 Zyklen
```

CLTS (clear task switched flag) (ab 80286)

Löscht das Task-Schalter-Bit.

```
Flags:    NT IOPL O D I T S Z A P C
          0  -    - - - - - - - - -
Syntax:   CLTS
Dauer:
```

CMC (complement carry flag)

Invertiert das Carry-Flag.

```
Flags:    O D I T S Z A P C
          - - - - - - - - X
Syntax:   CMC
Dauer:    2 Zyklen
```

CMP (compare two operands)

Subtrahiert den Quelloperanden vom Zieloperanden und setzt die Flags entsprechend dem Ergebnis. Das Ergebnis selbst wird nirgends gespeichert. Es bleiben also alle Register außer dem Flag-Register unverändert.

```
Flags:    O D I T S Z A P C
          X - - - X X X X X
Syntax:   CMP Ziel,Quelle
```

CMPS, CMPSB, CMPSW (compare string operands)

CMPS subtrahiert von dem mit (DS:SI) adressierten Byte das mit (ES:DI) adressierte und setzt die Flags entsprechend dem Ergebnis, wobei das Ergebnis selbst wieder verworfen wird. Nach der Testsubtraktion werden DI und SI um 1 erhöht falls das D-Flag 0 war, oder um 1 erniedrigt falls D=1. Der Befehl CMPW arbeitet wie CMPS, jedoch werden hier zwei Wörter verglichen und die Index-Register (DI, SI) um 2 erhöht oder erniedrigt.

```
Flags:    O D I T S Z A P C
          X - - - X X X X X
Syntax:   CMPSB
          CMPSW
Dauer:    10 Zyklen
```

CMPSD (ab 80386)

CMPSD verhält sich wie CMPS, jedoch wird hier doppelwortweise verglichen. SI und DI werden anschließend um 4 erhöht oder erniedrigt.

```
Flags:    O D I T S Z A P C
Syntax:   CMPSD
Code:     01100110 10100111
```

Dauer:

CWD (convert word to doubleword)

Erweitert das Wort im AX-Register vorzeichenrichtig auf Doppel-Wort-Größe (32 Bit). Dabei wird das DX-Register mit 0FFFFh geladen, falls Bit 15 von AX (MSB) gesetzt war, sonst wird das DX-Register gelöscht.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   CWD
Dauer:    2 Zyklen
```

DAA (decimal adjust al after addition)

Nach der Addition zweier gepackter BCD-Zahlen mit einem normalen Additionsbefehl, kann es vorkommen, daß das Ergebnis nicht als BCD-Code vorliegt. Der DAA-Befehl korrigiert das Ergebnis in diesen Fällen, so daß wieder ein gepackter BCD-Code entsteht. Dabei wird 06h addiert, falls die niederwertigen vier Bit des AL-Registers eine Pseudotetrade (Hex-Ziffern A bis F) enthalten, oder das A-Flag nach der Addition gesetzt war. Mit den oberen Bits wird genauso verfahren, jedoch wird hier anstatt des A-Flags das Carry-Flag abgefragt und gegebenenfalls 60h addiert.

```
Flags:    O D I T S Z A P C
          ? - - - X X X X X
Syntax:   DAA
Dauer:    4 Zyklen
```

DAS (decimal adjust after subtraction)

Nach der Subtraktion zweier gepackter BCD-Zahlen mit einem normalen Subtraktionsbefehl, kann es vorkommen, daß das Ergebnis nicht als BCD-Code vorliegt. Der DAS-Befehl erkennt dies und führt eine Korrektur des Ergebnisses im AL-Register durch, so daß wieder ein BCD-Code entsteht. Dabei wird 06h subtrahiert, falls die niederwertigen vier Bit von AL eine Pseudotetrade (Hex-Ziffern A bis F) enthalten, oder das A-Flag nach der Subtraktion gesetzt ist. Die oberen Bit von AL werden genauso behandelt, mit dem Unterschied, daß statt des A-Flags das Carry-Flag verwendet wird und gegebenenfalls 60h statt 06h subtrahiert wird.

```
Flags:    O  D  I  T  S  Z  A  P  C
          ?  -  -  -  X  X  X  X
Syntax:   DAS
Dauer:    4 Zyklen
```

DEC (decrement by 1)

Vermindert den Operanden um 1. Man beachte, daß der DEC-Befehl das Carry-Flag nicht beeinflußt!

```
Flags:    O  D  I  T  S  Z  A  P  C
          X  -  -  -  X  X  X  X
Syntax:   DEC operand
Dauer:    2 bis 6 Zyklen
```

DIV (unsigned divide)

Führt eine Division vorzeichenloser ganzer Zahlen durch. Wenn der angegebene Operand 8 Bit breit ist, wird der Inhalt des AX-Registers durch den Operanden geteilt. Der Quotient liegt danach im AL-Register, der Rest im AH-Register. Falls der Quotient größer als 0FFh=255d ist, oder versucht wurde durch 0 zu teilen, dann wird Interrupt 0 ausgelöst. Bei einem 16-Bit-Operanden dividiert die CPU die Inhalte von DX (High-Word) und AX (Low-Word), legt den Quotienten in AX und den Rest in DX ab. Ist der Quotient größer als 0FFFFh=56535d, dann wird ein Interrupt 0 ausgelöst.

```
Flags:    O  D  I  T  S  Z  A  P  C
          ?  -  -  -  ?  ?  ?  ?  ?
Syntax:   DIV operand
Dauer:    14 bis 41 Zyklen
```

ENTER (make stack frame for procedure parameters) (ab 80186)

Reserviert eine bestimmte Anzahl von Bytes auf dem Stack, um lokale Variablen von Hochsprachepro-grammen ablegen zu können. Dabei wird zuerst der Stack-Pointer auf den Stack gelegt, anschließend wird zahl+(2×grad) vom Stack-Pointer abgezogen.

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  -  -  -  -  -
Syntax:   ENTER zahl,grad
Dauer:
```

ESC (escape)

Der ESC-Befehl sendet Funktionscodes und Daten an Coprozessoren. Für den 80x86 wirkt er wie ein NOP-Befehl. Mit den xxx-Bits wird der Funktionscode verschlüsselt, mit den restlichen Bits des zweiten Bytes kann eine Speicherstelle adressiert werden.

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  -  -  -  -  -
Syntax:
Dauer:
```

HLT (halt)

Der Prozessor wird angehalten und wartet auf einen Interrupt.

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  -  -  -  -  -
Syntax:   HALT
```

IDIV (integer divide)

Führt eine Division vorzeichenbehafteter ganzer Zahlen durch. Der Rest erhält dabei das Vorzeichen des Quotienten. Wenn der angegebene Operand 8 Bit breit ist, wird der Inhalt des AX-Registers durch den Operanden geteilt. Der Quotient liegt danach im AL-Register, der Rest im AH-Register. Falls der Quotient außerhalb von 128 bis 127 liegt, oder versucht wurde durch 0 zu teilen, dann wird Interrupt 0 ausgelöst. Bei einem 16-Bit-Operanden dividiert die CPU die Inhalte von DX (High-Word) und AX (Low-Word), legt den Quotienten in AX und den Rest in DX ab. Ist der Quotient außerhalb des Bereichs -32768 bis 32767, dann wird ein Interrupt 0 ausgelöst.

```
Flags:    O D I T S Z A P C
          ? - - - ? ? ? ? ?
Syntax:   DIV operand
Dauer:    19 bis 43 Zyklen
```

IMUL (integer multiply)

Dieser Befehl führt eine vorzeichenrichtige Multiplikation ganzer Zahlen aus. Wenn der Operand ein Byte ist, dann multipliziert die CPU den Operanden mit dem AL-Register und legt das Ergebnis in AX ab. Haben nachher alle Bits des AH-Registers denselben Zustand wie das höchste Bit des AL-Registers (Wert von AX zwischen -128 und 127), dann werden die Flags CF und OF gelöscht, ansonsten gesetzt. Liegt ein 16-Bit-Operand vor, dann wird der Operand mit dem Inhalt des AX-Registers multipliziert und das Ergebnis in DX (High-Word) und AX (Low-Word) abgelegt. Haben nachher alle Bits des DX-Registers denselben Zustand wie das höchste Bit des AX-Registers (Wert von DX-AX zwischen -32768 und 32767), dann werden die Flags CF und OF gelöscht, ansonsten gesetzt. Sind drei Operanden angegeben, dann führt der 80x86 eine 16-Bit-Multiplikation des zweiten Wortoperanden mit dem dritten aus, und legt das Ergebnis im ersten ab. Ist das Ergebnis breiter als 16 Bit, dann werden die Flags CF und OF gesetzt, ansonsten gelöscht.

```
Flags:    O D I T S Z A P C
          X - - - ? ? ? ? X
Syntax:   IMUL byteoperand
          IMUL wortoperand
          IMUL wortregister,bytekonstante             (q=1)
          IMUL wortregister,wortadresse,bytekonstante (q=1)
          IMUL wortregister,wortadresse,wortkonstante (q=0)
Dauer:    9 bis 41 Zyklen
```

IN (input from port)

Liest ein Byte oder ein Wort von einem Eingabeport in das AL- oder in das AX-Register. Entweder ist die 8-Bit-Portadresse im zweiten Operanden angegeben, oder das DX-Register enthält eine 16-Bit-Portadresse.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   IN AL,8-Bit-Portadresse
          IN AX,DX
          IN AX,8-Bit-Portadresse
          IN AX,DX
Code:     1110010w kkkkkkkk
          1110110w
```

INC (increment by 1)

Erhöht den Operanden um 1. Man beachte, daß der INC-Befehl das Carry-Flag nicht beeinflußt!

```
Flags:    O D I T S Z A P C
          X - - - X X X X -
Syntax:   INC operand
Dauer:    2 bis 6 Zyklen
```

INS, INSB, INSW (input from port to string)

Liest einen String von einem Eingabeport. Bei INSB liest die CPU ein Byte von dem Port, der mit DX adressiert ist, legt es auf Adresse ES:DI und erhöht oder erniedrigt DI anschließend um 1. Der Befehl INSW liest ein Wort von dem mit DX adressierten Port, legt es auf Adresse ES:DI ab und erhöht oder erniedrigt DI um 2. Ob DI erhöht oder erniedrigt wird hängt vom D-Flag ab (DF=0: inkrementieren; DF=1: dekrementieren). Mit dem Befehlsvorsatz REP kann eine Schleife erzeugt werden, deren Anzahl der Durchläufe im CX-Register steht. !! Achtung !! Der Eingabeport muß bei Schleifen mit der CPU-Geschwindigkeit schritt halten können!

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   INSB
          INSW
Dauer:    9 bis 29 Zyklen
```

INSD (input from port to string for double word) (ab 80386)

Verhält sich wie INS, nur wird hier ein Doppelwort (32 Bit) gelesen und an die mit ES:EDI adressierte Speicherstelle geschrieben. Anschließend wird EDI je nach D-Flag (DF=0: inkremtieren; DF=1: dekrementieren) um 4 inkrementiert oder dekrementiert.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   INSD
Dauer:
```

INT, INTO (call to interrupt procedure)

Dieser Befehl führt einen Sprung in ein Interrupt-Programm aus, wenn das Interrupt-Enable-Flag gesetzt ist. Zunächst legt die CPU das Flag-Register, dann das Code-Segment-Register und schließlich das Befehls-Zeiger-Register auf den Stack. Anschließend löscht sie die Flags TF und IF, multipliziert den 8-Bit-Operanden mit 4 und holt mit der so gewonnenen Adresse einen Far-Pointer aus dem Speicher, der die Adresse der Interrupt-Service-Routine angibt. Der Befehl INTO ruft die Interrupt-Routine Nr. 4 auf, wenn das Overflow-Flag gesetzt ist und verhält sich ansonsten wie INT. Software-Interrupts werden im PC verwendet, um BIOS-Funktionen (Interrupts 10h bis 1Fh) oder DOS-Funktionen (Interrupts 20h bis 2Fh, insbesondere der DOS-Funktions-Verteiler Interrupt Nr. 21h) aufzurufen. **Hinweis:** Jede Interrupt-Service-Routine muß mit dem IRET-Befehl abgeschslossen sein!

```
Flags:    O D I T S Z A P C
          - - 0 0 - - - - -
Syntax:   INT operand
          INTO
Dauer:    33 bis 287 Zyklen
```

IRET (interrupt return)

Führt einen Rücksprung von einer Interrupt-Service-Routine zu dem unterbrochenen Programm aus. Die CPU lädt die Register IP, CS und Flags in dieser Reihenfolge wortweise (16 Bit) vom Stack und setzt damit die Bearbeitung im unterbrochenen Programm fort. Jede Interrupt-Service-Routine (egal ob Hard- oder Software-Interrupt) muß mit diesem Befehl abgeschlossen sein um einen korrekten Rücksprung zu gewährleisten.

```
Flags:    O D I T S Z A P C
          X X X X X X X X X
Syntax:   IRET
Dauer:    22 Zyklen
```

Jxx (jump short if condition met)

Springt relativ kurz (8-Bit-Offset), wenn die angegebene Sprungbedingung erfüllt ist. Ab dem 80386 sind auch 16-Bit-Offsets zugelassen. Es gibt mehrere Sprungbedingungen, wobei bei meistens zwischen Vergleich von Zahlen mit Vorzeichen und Vergleich von Zahlen ohne Vorzeichen zu unterscheiden ist.

Tabelle 3.3: Bedingungen für Vergleiche vorzeichenloser Zahlen

Befehl	Flags	Bedingung	Code
JA	CF=0 und ZF=0	springe, wenn größer	77h
JAE	CF=0	springe, wenn größer oder gleich	73h
JNA	CF=1 oder ZF=1	springe, wenn nicht größer	76h
JNAE	CF=1	springe, wenn nicht größer/gleich	72h
JNB	CF=0	springe, wenn nicht kleiner	73h
JNBE	CF=0 und ZF=0	springe, wenn nicht kleiner/gleich	77h

Tabelle 3.4: Bedingungen für Vergleiche vorzeichenbehafteter Zahlen

Befehl	Flags	Bedingung	Code
JG	ZF=0 und SF=OF	springe, wenn größer	7Fh
JGE	SF=OF	springe, wenn größer/gleich	7Dh
JL	SF≠OF	springe, wenn kleiner	7Ch
JLE	ZF=1 und SF≠OF	springe, wenn kleiner oder gleich	7Eh
JNG	ZF=1 und SF≠OF	springe, wenn nicht größer	7Eh
JNGE	SF≠OF	springe, wenn nicht größer/gleich	7Ch
JNL	SF=OF	springe, wenn nicht kleiner	7Dh
JNLE	ZF=0 und SF=OF	Springe, wenn nicht kleiner/gleich	7Fh
JNO	OF=0	springe, wenn kein Overflow	71h
JNS	SF=0	springe, wenn Ergebnis positiv	79h
JO	OF=1	springe, wenn Overflow	70h
JS	SF=1	springe, wenn Ergebnis negativ	78h

Tabelle 3.5: Sonstige Bedingungen

Befehl	Flags	Bedingung	Code
JC	CF=1	springe, wenn CF=1	72h
JNC	CF=0	springe, wenn CF=0	73h
JE	ZF=1	springe, wenn gleich	74h
JZ	ZF=1	springe, wenn ZF=1	74h
JNE	ZF=0	springe, wenn ungleich	75h
JNZ	ZF=0	springe, wenn ZF=0	75h
JP	PF=1	springe, wenn P=1	7Ah
JPE	PF=1	springe, wenn gerade Parität	7Ah
JNP	PF=0	springe, wenn P=0	7Bh
JPO	PF=0	springe, wenn ungerade Parität	7Bh

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   Jxxx offset
```

JCXZ (jump if CX-register zero)

Springt relativ kurz (8-Bit-Offset), wenn das CX-Register 0 enthält. Hierbei wird das Zero-Flag weder verändert, noch berücksichtigt.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   JCXZ offset
Dauer:
```

JMP (jump)

Springt immer an die angegebene Adresse. Es kann innerhalb des selben Segments relativ mit 8-Bit-Operand oder relativ mit 16-Bit-Operand oder absolut oder indirekt gesprungen werden. Für Sprünge in ein anderes Segment kann die Zieladresse direkt angegeben werden, oder es wird ein Near-Pointer angegeben, der auf den Sprung-Vektor zeigt.

```
Flags:     O  D  I  T  S  Z  A  P  C
           (Änderung nur bei Taskwechsel ab 80286)
Syntax:    JMP sprungziel
Dauer:     7 bis 268 Zyklen
```

LAHF (load flags into AH)

Lädt die Bedingungs-Flags in das AH-Register, wobei die einzelnen Flags folgende Positionen erhalten: Bit7 Bit0 SF ZF - AF - PF - CF

Bitbelegung im AH-Register nach LAHF:

```
Flags:     O  D  I  T  S  Z  A  P  C
           -  -  -  -  -  -  -  -  -
Syntax:    LAHF
```

LAR (load access rights byte) nur ab 80286)

Lädt das High-Byte des angegebenen Registers mit dem Zugriffs-Berechtigungs-Byte des adressierten Deskriptors und löscht das Low-Byte. Wenn der Befehl ausgeführt werden konnte, dann wird das Zero-Flag gesetzt, ansonsten gelöscht.

```
Flags:     O  D  I  T  S  Z  A  P  C
           -  -  -  -  -  X  -  -  -
Syntax:    LAR register,deskriptoradresse
```

LDS (load double word pointer to register and DS register)

Lädt einen Far-Pointer in die Register DS (Segment-Adresse) und das angegebene 16-Bit-Register (Offset-Adresse). Der zu ladende Far-Pointer liegt im Speicher und wird mit einem Near-Pointer, der im Befehl steht adressiert.

```
Flags:     O  D  I  T  S  Z  A  P  C
           -  -  -  -  -  -  -  -  -
Syntax:    LDS register,adresse
```

LEA (load effective adresse)

Dieser Befehl lädt die effektive Adresse des zweiten Operanden in das angegebene Register. Für den Operanden kann jede beliebige Adressierungsart angegeben werden, wobei die effektive Adresse gegebenenfalls abhängig von den aktuellen Registerinhalten berechnet wird. !! Achtung !! Nicht verwechseln mit "MOV register,operand" !

```
Flags:     O  D  I  T  S  Z  A  P  C
           -  -  -  -  -  -  -  -  -
Syntax:    LEA register,operand
```

LEAVE (high level procedure exit) (ab 80186)

LEAVE lädt den Stack-Pointer mit dem Inhalt des Base-Pointers, holt ein Wort vom Stack und legt es in das Base-Pointer-Register. Dieser Befehl hebt also die Wirkung des ENTER-Befehls auf.

```
Flags:     O  D  I  T  S  Z  A  P  C
           -  -  -  -  -  -  -  -  -
Syntax:    LEAVE
```

LES (load doubleword pointer to register and es register)

Lädt einen Far-Pointer in die Register ES (Segment-Adresse) und das angegebene 16-Bit-Register (Offset-Adresse). Der zu ladende Far-Pointer liegt im Speicher und wird mit einem Near-Pointer, der im Befehl steht adressiert.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   LES register,adresse
```

LGDT (load global descriptor table) (ab 80286)

Lädt einen 6 Byte großen Wert in das Globale-Deskriptor-Tabellen-Register. Dieser Befehl wird nur in Betriebssystemen angewandt.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   LGDT adresse
```

LIDT (load interrupt descriptor table register) (ab 80286)

Lädt das Interrupt-Deskriptor-Tabellen-Register. Dieser Befehl wird nur in Betriebssystemen angewandt.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   LIDT adresse
```

LLDT (load local descriptor table register) (ab 80286)

Lädt das Lokale-Deskriptor-Tabellen-Register. Dieser Befehl wird nur in Betriebssystemen angewandt.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   LLDT adresse
```

LMSW (load machine status word) (ab 80286)

Lädt das Maschinen-Status-Wort. Dieser Befehl wird nur in Betriebssystemen angewandt.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   LMSW adresse
```

LOCK (assert bus lock signal)

Sperrt den Bus solange, bis der nachfolgende Befehl abgearbeitet ist.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   LOCK
```

LODS, LODSB, LODSW (load string operand) Der LODSB-Befehl lädt das AL-Register mit dem durch DS:SI adressierten Byte und erhöht oder erniedrigt anschließend das SI-Register um 1. SI wird erhöht, wenn das D-Flag 0 ist, sonst wird erniedrigt. Der LODSW-Befehl verhält sich genauso, nur wird hier ein ganzes Wort geladen und das SI-Register anschließend um 2 erhöht oder erniedrigt.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   LODSB
          LODSW
Dauer:    5 Zyklen
```

LODSD (load string operand) (ab 80386)

LODSD verhält sich wie LODS, jedoch wird hier doppelwortweise geladen. SI und DI werden anschließend um 4 erhöht oder erniedrigt.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   LODSD
```

LOOP, LOOPE, LOOPNE, LOOPZ, LOOPNZ (loop control with CX)

Diese Befehle dekrementieren das CX-Register um 1 und springen relativ kurz, wenn CX nicht Null ist, ohne dabei das Zero-Flag zu verändern. Die Befehle LOOPE und LOOPZ springen zusätzlich nur dann, wenn das Zero-Flag gesetzt ist. Bei LOOPNE und LOOPZ wird nur gesprungen, wenn zusätzlich das Zero-Flag gelöscht ist.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   LOOP offset
          LOOPE offset
          LOOPNE offset
          LOOPZ offset
          LOOPNZ offset
Dauer:    11 Zyklen
```

LSL (load segment limit) (ab 80286)

Lädt die Segmentgrenze in ein Wortregister.

```
Flags:    O D I T S Z A P C
          - - - - - X - - -
Syntax:   LSL register,adresse
```

LTR (load task register) (ab 80286)

Lädt das Task-Register aus dem Speicher oder einem Wortregister.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   LTR adresse
```

MOV (move data)

Kopiert Daten.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   MOV ziel,quelle
Dauer:    2 bis 22 Zyklen
```

MOVS, MOVSB, MOVSW (mov data from string to string) Diese Befehle kopieren das mit DS:SI adressierte Datum nach ES:DI und erhöhen oder erniedrigen anschließend die Register SI und DI um die Anzahl der kopierten Bytes. Wenn das D-Flag gesetzt ist, wird erniedrigt, sonst erhöht. Der Befehl MOVSB kopiert ein Byte, der Befehl MOVSW ein Wort. Mit Hilfe des REP-Vorsatzes Kannen auch Schleifen programmiert werden deren Wiederholungszahl im CX-Register steht.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   MOVSB
          MOVSW
Dauer:    7 Zyklen
```

MOVSD (mov double word from string to string) (ab 80386)

Dieser Befehl verhält sich wie MOVS, nur wird hier ein Doppelwort (32-Bit) übertragen, und die Index-Register SI und DI anschließend um 4 erhöht oder erniedrigt.

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  -  -  -  -  -
Syntax:   MOVSD
Dauer:    7 Zyklen
```

MUL (unsigned multiplication of AL or AX)

Führt eine Multiplikation vorzeichenloser ganzer Zahlen durch. Ist der angegebene Operand ein Byte, dann multipliziert die CPU das AL-Register mit dem Operanden und legt das Ergebnis im AX-Register ab. Ist nach der Multiplikation das AH-Register Null, dann werden die Flags Overflow und Carry gelöscht, sonst gesetzt. War ein 16-Bit-Operand angegeben, dann wird dieser mit dem AX-Register multipliziert und das Ergebnis in den Registern DX (High-Word) und AX (Low-Word) abgelegt. Wenn DX nach der Multiplikation 0 enthält, dann werden die Flags OF und CF gelöscht, sonst gesetzt.

```
Flags:    O  D  I  T  S  Z  A  P  C
          X  -  -  -  ?  ?  ?  ?  X
Syntax:   MUL operand
Dauer:    9 bis 41 Zyklen
```

NEG (two's complement negation)

Bildet das Zweierkomplement (echtes Komplement) des Operanden.

```
Flags:    O  D  I  T  S  Z  A  P  C
          X  -  -  -  X  X  X  X  X
Syntax:   NEG operand
Code:     1111011w mm011MMM llllllll hhhhhhhh
Dauer:    2 bis 6 Zyklen
```

NOP (no operation)

Keine Aktion. Dieser Befehl wird oft als Füllbyte zwischen anderen Maschinenbefehlen eingesetzt um nicht mehr benötigte Befehle zu entfernen oder später neue hinzufügen zu können.

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  -  -  -  -  -
Syntax:   NOP
Dauer:    3 Zyklen
```

NOT (one's complement negation)

Bildet das Einerkomplement (unechtes Komplement) aus dem angegebenen Operanden.

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  -  -  -  -  -
Syntax:   NOT operand
Code:     1111011w mm010MMM llllllll hhhhhhhh
Dauer:    2 bis 6 Zyklen
```

OR (logical inclusive or)

Verknüpft den Zielopeanden bitweise oder mit dem Quelloperanden und legt das Ergebnis im Zieloperanden ab.

```
Flags:     O  D  I  T  S  Z  A  P  C
           0  -  -  -  X  X  ?  X  0
Syntax:    OR ziel,quelle
Code:      0000110w kkkkkkkk jjjjjjjj
           1000000w mm001MMM llllllll hhhhhhhh kkkkkkkk jjjjjjjj
           0000101w mmrrrMMM llllllll hhhhhhhh
           0000100w mmrrrMMM llllllll hhhhhhhh
Dauer:     2 bis 7 Zyklen
```

OUT (output to port)

Gibt ein Byte oder ein Wort auf dem Port aus. Der Port wird entweder mit einer 8-Bit-Port-Adresse die im Befehl enthalten ist ausgewählt, oder eine 16-Bit-Port-Adresse steht im DX-Register.

```
Flags:     O  D  I  T  S  Z  A  P  C
           -  -  -  -  -  -  -  -  -
Syntax:    OUT adresse,AL
           OUT DX,AL
           OUT adresse,AX
           OUT DX,AX
Dauer:     3 bis 4 Zyklen
```

OUTS, OUTSB, OUTSW (output string to port) (ab 80186)

Diese Befehle geben das mit DS:SI adressierte String-Element auf dem Ausgabeport aus, dessen Adresse im DX-Register steht. Anschließend wird SI um die Anzahl der übertragenen Bytes erhöht, wenn DF=0. Ist DF=1, dann wird SI entsprechend erniedrigt. Mit Hilfe des REP-Vorsatzes kann eine Schleife erzeugt werden, deren Durchlaufzahl im CX-Register steht. !! Achtung !! Der Ausgabeport muß hierbei mit der Geschwindigkeit der CPU schritt halten können!

```
Flags:     O  D  I  T  S  Z  A  P  C
           -  -  -  -  -  -  -  -  -
Syntax:    OUTSB DX
           OUTSW DX
Dauer:     7 Zyklen
```

OUTSD (output double word to port) (ab 80386)

Verhält sich wie OUTS, jedoch wird hier ein Doppelwort (32 Bit) übertragen und SI anschließend um 4 erniedrigt oder erhöht.

```
Flags:     O  D  I  T  S  Z  A  P  C
           -  -  -  -  -  -  -  -  -
Syntax:    OUTSD
Dauer:     7 Zyklen
```

POP (pop a word from the stack)

Holt ein Wort (16 Bit) vom Stack und legt es im angegebenen Wort-Register ab. Liest zuerst das mit SS:SP adressierte Wort und erhöht anschließend das SP-Register (Stack-Pointer) um 2.

```
Flags:     O  D  I  T  S  Z  A  P  C
           -  -  -  -  -  -  -  -  -
Syntax:    POP register
Code:      000ss111
           10001111 mm000MMM llllllll hhhhhhhh
Dauer:     5 bis 7 Zyklen
```

POPA (pop all general registers) (ab 80186)

Lädt folgende Register in der angegebenen Reihenfolge wortweise vom Stack: DI, SI, BP, SP, BX, DX, CX, AX Der Stack-Pointer wird dabei jedesmal nachdem ein Wort vom Stack gelesen wurde um 2 erhöht. Das vierte Wort wird zwar vom Stack gelesen, jedoch nicht in den Stack-Pointer kopiert.

```
Flags:   O D I T S Z A P C
         - - - - - - - - -
Syntax:  POPA
```

POPF (pop from stack into flags register)

Liest ein Wort vom Stack, kopiert es in das Flag-Register und erhöht anschließend den Stack-Pointer um 2.

```
Flags:   O D I T S Z A P C
         X X X X X X X X X
Syntax:  POPF
Dauer:   5 Zyklen
```

PUSH (push a word onto the stack)

Erniedrigt zuerst den Stack-Poiner (=SP) um 2 und kopiert dann den Inhalt des angegebenen Wortregisters oder der angegebenen Speicherstelle auf die Adresse SS:SP. Es können auch 16-Bit-Konstanten auf den Stack gelegt werden.

```
Flags:   O D I T S Z A P C
         - - - - - - - - -
Syntax:  PUSH register
Dauer:   2 bis 5 Zyklen
```

PUSHA (push all general registers) (ab 80186)

Die CPU legt folgende Register in der angegebenen Reihenfolge auf den Stack: AX, CX, DX, BX, SP, BP, SI, DI Dabei wird der Stack-Pointer vor dem Ablegen eines Register-Inhalts um 2 erniedrigt. Es ist jedoch zu beachten, daß hierbei trotzdem der Stack-Pointer-Inhalt vor dem PUSHA-Befehl abgelegt wird.

```
Flags:   O D I T S Z A P C
         - - - - - - - - -
Syntax:  PUSHA
Dauer:
```

PUSHF (push flags register onto the stack)

Legt den Inhalt des Flag-Register auf den Stack. Dabei wird zuerst der Stack-Pointer um 2 erniedrigt und dann das Flag-Register auf die mit SS:SP adressierte Speicherstelle kopiert.

```
Flags:   O D I T S Z A P C
         - - - - - - - - -
Syntax:  PUSHF
Dauer:   4 Zyklen
```

RCL (rotate through carry left)

Schiebt den Inhalt des Operanden einmal zyklisch nach links, wobei das höchstwertige Bit im Carry-Flag gespeichert und das alte Carry in die niedrigstwertige Stelle des Operanden eingetragen wird. Wenn nach dem Schieben das Carry-Flag mit dem höchstwertigen Bit des Operanden übereinstimmt, dann wird das Overflow-Flag gelöscht, sonst gesetzt. Es können auch mehrere Schiebeschritte mit einem Befehl ausgeführt werden, wenn im CL-Register die Zahl der Schiebeschritte eingetragen wird. Ab 80186 kann die Zahl der Schiebeschritte auch als Konstante im Befehl angegeben werden. Bei mehrfachem Schieben ist jedoch der Inhalt des Overflow-Flags undefiniert.

```
Flags:    O  D  I  T  S  Z  A  P  C
          X  -  -  -  -  -  -  -  X
Syntax:   RCL operand,1
          RCL operand,CL
          RCL operand,konstante
Dauer:    3 bis 10 Zyklen
```

RCR (rotate through carry right)

Schiebt den Inhalt des Operanden einmal zyklisch nach rechts, wobei das niedrigstwertige Bit im Carry-Flag gespeichert und das alte Carry in die höchstwertigte Stelle des Operanden eingetragen wird. Wenn vor dem Schieben das Carry-Flag mit dem höchstwertigen Bit des Operanden übereinstimmt, dann wird das Overflow-Flag gelöscht, sonst gesetzt. Es können auch mehrere Schiebeschritte mit einem Befehl ausgeführt werden, wenn im CL-Register die Zahl der Schiebeschritte eingetragen wird. Ab 80186 kann die Zahl der Schiebeschritte auch als Konstante im Befehl angegeben werden. Bei mehrfachem Schieben ist jedoch der Inhalt des Overflow-Flags undefiniert.

```
Flags:    O  D  I  T  S  Z  A  P  C
          X  -  -  -  -  -  -  -  X
Syntax:   RCR operand,1
          RCR operand,CL
          RCR operand,konstante
Dauer:    3 bis 10 Zyklen
```

REP, REPE, REPZ, REPNE, REPNZ (repeat following string operation)

Diese Befehle bewirken, daß das CX-Register ohne Änderung des Z-Flags dekrementiert und der nachfolgende String-Befehl wiederholt wird, wenn das CX-Register nicht Null ist. REPE und REPZ prüfen zusätzlich das Z-Flag und wiederholen, wenn es gesetzt ist. REPNE und REPNZ wiederholen, wenn das Z-Flag gelöscht ist.

```
Flags:     O  D  I  T  S  Z  A  P  C
           -  -  -  -  -  -  -  -  -
Syntax:    REP
           REPE
           REPZ
           REPNE
           REPZ
Dauer:
```

RET (return from procedure)

Kehrt von einem Unterprogramm zurück. Die CPU holt vom Stack die Rücksprung- Adresse, erhöht den Stack-Pointer und springt dann an die von Stack gelesene Adresse. Es ist hierbei zwischen RETN für Near-Prozeduren und RETF für Far-Prozeduren zu unterscheiden (näheres hierzu siehe CALL-Befehl). Es ist möglich mit dem RET-Befehl eine zusätzliche Korrektur des Stack-Pointers vorzunehmen. Dabei addiert die CPU nach dem Erhöhen des Stack-Pointers zusätzlich noch die angegebene 16-Bit-Konstante.

```
Flags:     O  D  I  T  S  Z  A  P  C
           -  -  -  -  -  -  -  -  -
Syntax:    RET      (bei den meisten Assemblern verwendet)
           RETN
           RETF
Dauer:     10 bis 68 Zyklen
```

ROL (rotate left)

Schiebt den Operanden zyklisch nach links. Das freie niedrigstwertige Bit wird mit dem oben heraus geschobenen Bit belegt. Außerdem wird das herausgeschobene Bit noch in das Carry-Flag kopiert. Wenn nach dem Schieben das Carry-Flag mit dem höchstwertigen Bit des Operanden übereinstimmt, dann wird das Overflow-Flag gelöscht, sonst gesetzt. Es können auch mehrere Schiebeschritte durchgeführt werden, wenn im CL-Register die Zahl der Schiebeschritte steht. Ab dem 80186 kann die Zahl der Schiebeschritte auch als Konstante im Befehl angegeben werden. Bei mehrfachem Schieben ist das Overflow-Flag jedoch undefiniert.

```
Flags:   O  D  I  T  S  Z  A  P  C
         X  -  -  -  -  -  -  -  X
Syntax:  ROL operand,1
         ROL operand,CL
         ROL operand,konstante  (ab 80186)
Dauer:   3 bis 10 Zyklen
```

ROR (rotate right)

Schiebt den Operanden zyklisch nach rechts. Das freie höchstwertige Bit wird mit dem unten heraus geschobenen Bit belegt. Außerdem wird das herausgeschobene Bit noch in das Carry-Flag kopiert. Wenn nach dem Schieben das Carry-Flag mit dem höchstwertigen Bit des Operanden übereinstimmt, dann wird das Overflow-Flag gelöscht, sonst gesetzt. Es können auch mehrere Schiebeschritte durchgeführt werden, wenn im CL-Register die Zahl der Schiebeschritte steht. Ab dem 80186 kann die Zahl der Schiebeschritte auch als Konstante im Befehl angegeben werden. Bei mehrfachem Schieben ist das Overflow-Flag jedoch undefiniert.

```
Flags:   O  D  I  T  S  Z  A  P  C
         X  -  -  -  -  -  -  -  X
Syntax:  ROR operand,1
         ROR operand,CL
         ROR operand,konstante  (ab 80186)
```

SAHF (store AH into flags)

Speichert den Inhalt des AH-Registers als Bedingungsflags im Flag-Register. (siehe auch LAHF)

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  X  X  X  X  X
Syntax:   SAHF
Dauer:    3 bis 10 Zyklen
```

SAL (shift arithmetic left)

Schiebt den Operanden einmal arithmetisch nach links. Das freie niedrigstwertige Bit wird mit 0 belegt. Außerdem wird das oben herausgeschobene Bit in das Carry-Flag kopiert. Wenn nach dem Schieben das Carry-Flag mit dem höchstwertigen Bit des Operanden übereinstimmt, dann wird das Overflow-Flag gelöscht, sonst gesetzt. Es können auch mehrere Schiebeschritte durchgeführt werden, wenn im CL-Register die Zahl der Schiebeschritte steht. Ab dem 80186 kann die Zahl der Schiebeschritte auch als Konstante im Befehl angegeben werden. Bei mehrfachem Schieben ist das Overflow-Flag jedoch undefiniert.

```
Flags:   O  D  I  T  S  Z  A  P  C
         X  -  -  -  X  X  ?  X  C
Syntax:  SAL operand,1
         SAL operand,CL
         SAL operand,konstante  (ab 80186)
Dauer:   3 bis 7 Zyklen
```

SAR (shift arithmetic right)

Schiebt den Operanden einmal arithmetisch nach rechts. Das höchstwertige Bit wird nicht verändert, um das Vorzeichen beizubehalten. Außerdem wird das unten herausgeschobene Bit in das Carry-Flag kopiert. Das Overflow-Flag wird gelöscht, wenn einmal geschoben wurde, sonst bleibt es erhalten. Es können auch mehrere Schiebeschritte durchgeführt werden, wenn im CL-Register die Zahl der Schiebeschritte steht. Ab dem 80186 kann die Zahl der Schiebeschritte auch als Konstante im Befehl angegeben werden. Bei mehrfachem Schieben ist das Overflow-Flag jedoch undefiniert.

```
Flags:   O  D  I  T  S  Z  A  P  C
         X  -  -  -  X  X  ?  X  C
Syntax:  SAR operand,1
         SAR operand,CL
         SAR operand,konstante
Dauer:   3 bis 7 Zyklen
```

SBB (integer subtraction with borrow)

Subtrahiert vom Zieloperanden den Quelloperanden und das Carry-Flag und legt das Ergebnis im Zieloperanden ab.

```
Flags:    O  D  I  T  S  Z  A  P  C
          X  -  -  -  X  X  X  X  X
Syntax:   SBB ziel,quelle
Dauer:    2-7 Zyklen
```

SCAS, SCASB, SCASW (scan string)

SCAS subtrahiert vom AL- oder AX-Register den Inhalt der mit ES:DI adressierten Speicherstelle und verändert die Flags entsprechend, während das Subtraktionsergebnis verworfen wird. Nach der Subtraktion erhöht (DF=0) oder erniedrigt (DF=1) die CPU das DI-Register um 1 oder 2, je nachdem ob byteweise (SCASB) oder wortweise (SCASW) verglichen wurde.

```
Flags:    O  D  I  T  S  Z  A  P  C
          X  -  -  -  X  X  X  X  X
Syntax:   SCASB         (8-Bit-Vergleich mit AL)
          SCASW         (16-Bit-Vergleich mit AX)
Dauer:    7 Zyklen
```

SCASD (scan double word) (ab 80286)

Verhält sich wie SCAS, nur wird hier doppelwortweise (32 Bit) verglichen und DI anschließend um 4 erhöht (DF=0) oder erniedrigt (DF=1).

```
Flags:    O  D  I  T  S  Z  A  P  C
          X  -  -  -  X  X  X  X  X
Syntax:   SCASD
Dauer:    7 Zyklen
```

SEG (segment override)

Ermöglicht die Auswahl eines anderen Segment-Registers für die Adressierung. Dieser Befehl bezieht sich nur auf den nachfolgenden Maschinenbefehl. Beim MASM und TASM wird CS:, DS:, SS: oder ES: vor den betroffenen Operanden geschrieben.

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  -  -  -  -  -
Syntax:   Befehl CS:Operand
          Befehl SS:Operand
          Befehl DS:Operand
          Befehl ES:Operand
```

SGDT (store global descriptor table register) (ab 80286)

Speichert das Register der globalen Deskriptor-Tabelle mit 6 Bytes auf den adressierten Operanden ab.

Dieser Befehl wird nur vom Betriebssystem benutzt.

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  -  -  -  -  -
Syntax:   SGDT operand
```

SHL (shift logical left)

Identisch mit SAL.

SHR (shift logical right)

Schiebt den Operanden einmal logisch nach rechts. Das freie höchstwertige Bit wird mit 0 belegt. Außerdem wird das unten herausgeschobene Bit in das Carry-Flag kopiert. Das Overflow-Flag enthält nach dem Schieben den Wert des ursprünglichen höchstwertigen Bits. Es können auch mehrere Schiebeschritte durchgeführt werden, wenn im CL-Register die Zahl der Schiebeschritte steht. Ab dem 80186 kann die Zahl der Schiebeschritte auch als Konstante im Befehl angegeben werden. Bei mehrfachem Schieben ist das Overflow-Flag jedoch undefiniert.

```
Flags:   O  D  I  T  S  Z  A  P  C
 x       X  -  -  -  X  X  ?  X  C
Syntax:  SAR operand,1
         SAR operand,CL
         SAR operand,konstante  (ab 80186)
Dauer:   3 bis 7 Zyklen
```

SIDT (store interrupt descriptor table register) (ab 80286)

Speichert das Register der Interrupt-Deskriptor-Tabelle mit 6 Bytes auf den adressierten Operanden ab. Dieser Befehl wird nur vom Betriebssystem benutzt.

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  -  -  -  -  -
Syntax:   SIDT operand
```

SLDT (store local descriptor table register) (ab 80286)

Speichert das Register der lokalen Deskriptor-Tabelle mit 6 Bytes auf den adressierten Operanden ab. Dieser Befehl wird nur vom Betriebssystem benutzt.

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  -  -  -  -  -
Syntax:   SLDT operand
```

SMSW (store machine status word) (ab 80286)

Speichert das Maschinen-Status-Wort im angegebenen Operanden. Dieser Befehl wird nur vom Betriebssystem benutzt.

```
Flags:    O  D  I  T  S  Z  A  P  C
          -  -  -  -  -  -  -  -  -
Syntax:   SMSW operand
```

STC (set carry flag)

Setzt das Carry-Flag.

```
Flags:    O D I T S Z A P C
          - - - - - - - - 1
Syntax:   STC
Dauer:    2 Zyklen
```

STD (set direction flag)

Setzt das Direction-Flag.

```
Flags:    O D I T S Z A P C
          - 1 - - - - - - -
Syntax:   STD
Dauer:    2 Zyklen
```

STI (set interrupt enable flag)

Setzt das Interrupt-Flag und ermöglicht damit Interrupts.

```
Flags:    O D I T S Z A P C
          - - 1 - - - - - -
Syntax:   STI
Dauer:    2 Zyklen
```

STOS, STOSB, STOSW (store string data)

STOS speichert den Inhalt von AL (STOSB) oder AX (STOSW) auf die mit ES:DI adressierte Speicherstelle und erhöht (DF=0) oder erniedrigt (DF=1) anschließend das DI-Register um die Zahl der gespeicherten Bytes. Es ist nicht möglich mit dem SEG-Vorsatz ein anderes als das Extra-Segment-Register (ES) zu wählen. Mit dem REP-Vorsatz lassen sich Schleifen erzeugen, deren Durchlaufzahl vom Inhalt des CX-Registers abhängt.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   STOSB
          STOSW
Dauer:    4 Zyklen
```

STOSD (store double word) (ab 80386)

Verhält sich wie STOS, nur wird hier der Inhalt des EAX-Register (32 Bit) auf die Adresse ES:EDI kopiert und das EDI-Register anschließend um 4 erhöht (DF=0) oder erniedrigt (DF=1).

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   STOSD
Dauer:    4 Zyklen
```

STR (store task register) (ab 80286)

Speichert den Inhalt des Task-Registers im angegebenen Operanden.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   STR operand
```

SUB (integer subtraction)

Subtrahiert den Quell-Operanden vom Ziel-Operanden und speichert das Ergebnis im Zieloperanden.

```
Flags:    O D I T S Z A P C
          X - - - X X X X X
Syntax:   SUB ziel,quelle
Dauer:    2 bis 7 Zyklen
```

TEST (logical compare)

Führt eine bitweise Und-Verknüpfung der beiden Operanden durch und setzt die Flags entsprechend, während das Ergebnis verworfen wird.

```
Flags:    O D I T S Z A P C
          0 - - - X X ? X 0
Syntax:   TEST operand1,operand2
```

VERR (verify a segment for reading) (ab 80286)

Interpretiert das adressierte Wort als Selektor und überprüft die Privilegierungsstufe und die Lesbarkeit des zugehörigen Segments.

```
Flags:    O D I T S Z A P C
          - - - - - X - - -
Syntax:   VERR operand
```

VERW (verify a segment for writing) (ab 80286)

Interpretiert das adressierte Wort als Selektor und überprüft die Privilegierungsstufe und die Schreibbarkeit des zugehörigen Segments.

```
Flags:    O D I T S Z A P C
          - - - - - X - - -
Syntax:   VERW operand
```

WAIT (wait until busy pin is inaktive)

Wartet bis der Busy-Eingang inaktiv (high) ist.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   WAIT
```

XCHG (exchange memory/register with register)

Vertauscht die beiden adressierten Bytes oder Worte. Während des Vertauschens ist unabhängig vom LOCK-Vorsatz immer BUS LOCK aktiv.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   XCHG operand1,operand2
Dauer:    3 bis 5 Zyklen
```

XLAT (table look-up translation)

Die Summe aus BX und AL ergibt die Offset-Adresse eines Bytes, welches in das AL-Register geladen wird.

```
Flags:    O D I T S Z A P C
          - - - - - - - - -
Syntax:   XLAT
Dauer:    5 Zyklen
```

XOR (logical exclusive or)

Führt eine bitweise Exclusiv-Oder-Verknüpfung des Ziel-Operanden mit dem Quelloperanden und legt das Ergebnis im Ziel-Operanden ab.

```
Flags:     O  D  I  T  S  Z  A  P  C
           0  -  -  -  X  X  ?  X  X
Syntax:    XOR ziel,quelle
```

3.2 Das Programmieren in Assembler

Im Praktikum verwenden wir **NASM**, den *Netwide Assembler*. Die *public domain* Software ist fur DOS/Windows und Linux verfügbar. Nähere Information zum Programmieren in Assembler allgemein sowie zu NASM finden Sie unter

http://wwwbode.informatik.tu-muenchen.de/ luksch/tgi/

3.3 Die serielle Schnittstelle

Die serielle Ein-/Ausgabe übernimmt im PC ein Baustein namens *Universal Asnyncronous Receiver/Transmitter*, kurz UART. Die gängigen Chips haben die Bauteilbezeichnungen 8250, 82450, 16450 oder 16550. Alle UARTS haben im Prinzip dieselbe Schnittstelle. Zwar gibt es kleinere Unterschiede; diese sind aber im Rahmen des Praktikums nicht relevant.

Für jede serielle Schnittstelle gibt es einen UART. Die vorhandenen UARTs werden während des Boot-Vorgangs erkannt. Die Basisadressen der UAARTS stehen an folgenden Adressen im RAM:

Adresse	serieller Port
0040:0000H	COM1
0040:0002H	COM2
0040:0004H	COM3
0040:0006H	COM4

Der Zugriff auf den UART erfolgt über acht aufeinanderfolgende I/O Ports, die relativ zur Basisadresse über ein *Offset* adressiert werden. Bei *Offset* 3 liegt das *Line Control Register*. Sein höchstwertiges Byte (*most significant byte*, *MSB*) ist das *divisor latch access bit*, kurz DLAB. Dieses Bit legt fest, auf welchen Registersatz bei den *Offsets* 0 und 1 zugegriffen wird. Aus Tab. 3.6 ist zu entnehmen, auf welches Register mit unterschiedlichen Kombinationen von *Offset*, DLAB, und Read/Write zugegriffen wird.

Im **Empfangspuffer (*Receive Buffer*)** steht das zu lesende Byte. Werden weniger als 8 bit gelesen, sind die höherwertigen Bits undefiniert und müssen von der verarbeitenden Software maskiert werden.

Der **Sendepuffer (*Transmitter Holding Register, THR*)** enthält das nächste zu sendende Byte. Wenn das vorherige Byte noch nicht gesendet wurde, wird es überschrieben. Diesen *transmitter overrun* erkennt der UART nicht.

Das ***Divisor Latch*** legt die Übertragungsrate fest. Ein Divisor d entspricht einer Übertragungsrate von $\frac{115\,200}{d}$ bits pro Sekunde. Für die Übertragungsrate b ist ein Divisor von $\frac{115\,200}{b}$ einzustellen.

Das ***Interrupt Enable Register*** legt fest welche Ereignisse einen Interrupt seitens des UART auslösen. Das Registerformat ist in Tab. 3.7 beschrieben.

Eine Zustandsänderung bei „*Modem Input*“ liegt vor, wenn eines der folgenden Signale seinen Zustand geändert hat: CD (*Carrier Detect*), RI (*Ring Indicator*), DSR (*Data Set Ready*), CTS (*Clear to Send*).

Offset	DLAB	Read/Write	Register
0	0	Read	Empfangspuffer (*Receive Buffer*)
0	0	Write	Senderegister (*Transmitter Holding Register*, THR)
0	1	Read/Write	Divisor Latch (niedrigstwertiges Byte, *least significand byte, LSB*
1	0	Read/Write	Interrupt Enable Register (IER)
1	1	Read/Write	Divisor Latch (höchstwertiges Byte, *most significand byte, MSB*)
2	X	Read	Interrupt Identification Regiser (IIR)
2	X	Write	FIFO channel (nicht bei allen Chips)
3	X	Read/Write	Line Control Register (LCR)
4	X	Read/Write	Modem Control Register (MCR)
5	X	Read	Line Status Register (LSR)
6	X	Read/Write	Modem Status Regiser (MSR)
7	X	Read/Write	Scratch (nicht bei allen Chips)

Tabelle 3.6: Adressierung der Register des UART

Bits	Bedeutung
7–4	immer gelöscht
3	Zustandsänderung an „*Modem Input*“ löst Interrupt aus
2	BREAK, *parity error*, *overrun error*, *framing error* lösen Interrupt aus
1	*Transmitter Buffer Empty (TBE)* löst Interrupt aus
0	*Receive Data Ready (RxRDY)* löst Interrupt aus

Tabelle 3.7: Das *Interrupt Enable Register*

Der TBE Interrupt zeigt an, daß das nächste zu sendende Byte aus dem *Transmitter Holding Register* in das interne *Transmitter Shift Register* übernommen wurde, so daß nun der nächste Wert in das THR geschrieben werden kann.

Ein RxRDY Interrupt bedeutet, daß Daten im Empfangspuffer zum Lesen bereitliegen.

Das ***Interrupt Identification Register*** ist nur lesbar (*read only* Register). Es zeigt an ob ein Interrupt anhängig (*pending*) ist, und gibt von den anhängigen Interrupts den mit der höchsten Priorität an (die höchste Priorität ist 0. Tab. 3.8.

Bit	Bedeutung	
7	FIFO enable	
6	FIFO enable	
5–4	Null (unbenutzt)	
3–1	Interrupt source	
	000	Modem Input Change (Priorität 3)
	001	TBE (Priorität 2)
	010	RXRDY (Priorität 1)
	011	BREAK on error (Priorität 0)
	100	*illegal*
	101	*illegal*
	110	RxRDY (FIFO) (Priorität 1)
	111	*illegal*
0	Interrupt pending	
	0	Interrupt pending
	1	No interrupt pending

Tabelle 3.8: Das *Interrupt Identification Register*

Das **FIFO Control Register** ist nur schreibar (*write only* Register). Nicht alle UARTs haben FIFOs. Deshalb gehen wir hier nicht näher auf das Register ein.

4 Bereich II: Mikroprogrammierung

Wolfgang Karl, Roland Wismüller

4.1 Aufbau des mikroprogrammierbaren TGI-Rechners

In der Vorlesung Technische Grundlagen der Informatik, in den Übungen hierzu und im Praktikum wird das Konzept der Mikroprogrammierung am Beispiel eines mikroprogrammierbaren Rechners demonstriert. Im folgenden wird dieser mikroprogrammierbare Beispielrechner eingeführt, wobei die Beschreibung nur auf dessen wesentliche Komponenten und deren Funktionsweise eingeht und auf eine detaillierte Darstellung der Einfachheit wegen verzichtet wird.

Der mikroprogrammierte Rechner besteht entsprechend dem von Neumann'schen Konzept aus einem Leitwerk zur Generierung der Steuersignale für die einzelnen Komponenten des Rechners, einem Rechenwerk zur Verarbeitung von 16-Bit Integer-Daten, einem Speicherwerk mit einem $64K \times 16$ Bit großen Hauptspeicher und einem Ein-/Ausgabewerk, über das die Kommunikation mit dem PC erfolgt. Die einzelnen Werke sind über einen 16 Bit breiten Datenbus und einen 16 Bit breiten Adreßbus miteinander verbunden.

Das **Leitwerk** enthält einen $4K \times 80$ Bit großen Mikroprogrammspeicher (MPS), in dem die Mikroprogramme abgelegt sind. Ein Mikroprogramm besteht aus einer Folge von Mikroinstruktionen. In jedem Taktzyklus wird eine Mikroinstruktion ausgewählt und alle in ihr zusammengefaßten Mikrooperationen werden gleichzeitig zur Ausführung angestoßen. Die Auswahl der im nächsten Taktzyklus auszuführenden Mikroinstruktion erfolgt durch das Mikroleitwerk (MLW), das in jedem Taktzyklus eine 80-Bit breite Speicherzelle des Mikroprogrammspeichers adressiert.

Jedes Mikroprogramm kann einen Maschinenbefehl einer virtuellen Zielmaschine implementieren. Diese virtuelle Zielmaschine ist durch die Menge der durch Mikroprogramme implementierten Maschinenbefehle und das vereinbarte Programmiermodell definiert. Die Programme, die mit den Maschinenbefehlen der Zielmaschine geschrieben werden können, stehen zusammen mit den zu verarbeitenden Daten im Hauptspeicher.

Das **Rechenwerk** besteht aus einer Verarbeitungseinheit (der arithmetischen und logischen Einheit, ALU), die arithmetische und logische Operationen auf 16-Bit breiten Operanden ausführt, die von einer Registerdatei kommen können oder vom Datenbus übernommen werden. Darüberhinaus können Schiebeoperationen ausgeführt werden.

Mit Hilfe der mikroprogrammierbaren Maschine soll zum einen das Konzept bzw. die Technik der Mikroprogrammierung demonstriert werden, zum anderen können die in einem Rechner ablaufenden Vorgänge bei der Abarbeitung eines Maschinenprogramms vermittelt werden.

In Abbildung 4.1 ist die mikroprogrammierbare Maschine dargestellt, wobei zusätzlich angezeigt ist, aus welchen Feldern des Mikroinstruktionsregisters die einzelnen Komponenten des mikroprogrammierbaren Rechners gesteuert werden. (Der Übersichtlichkeit wegen ist das Unterbrechungswerk und das Ein-/Ausgabewerk nicht dargestellt.)

Die Bausteine, mit denen die einzelnen Werke aufgebaut sind, werden im folgenden vereinfacht beschrieben. Für eine genaue Beschreibung der Funktionsweise der Bausteine der Familie Am2900 sei auf die Datenblätter verwiesen.

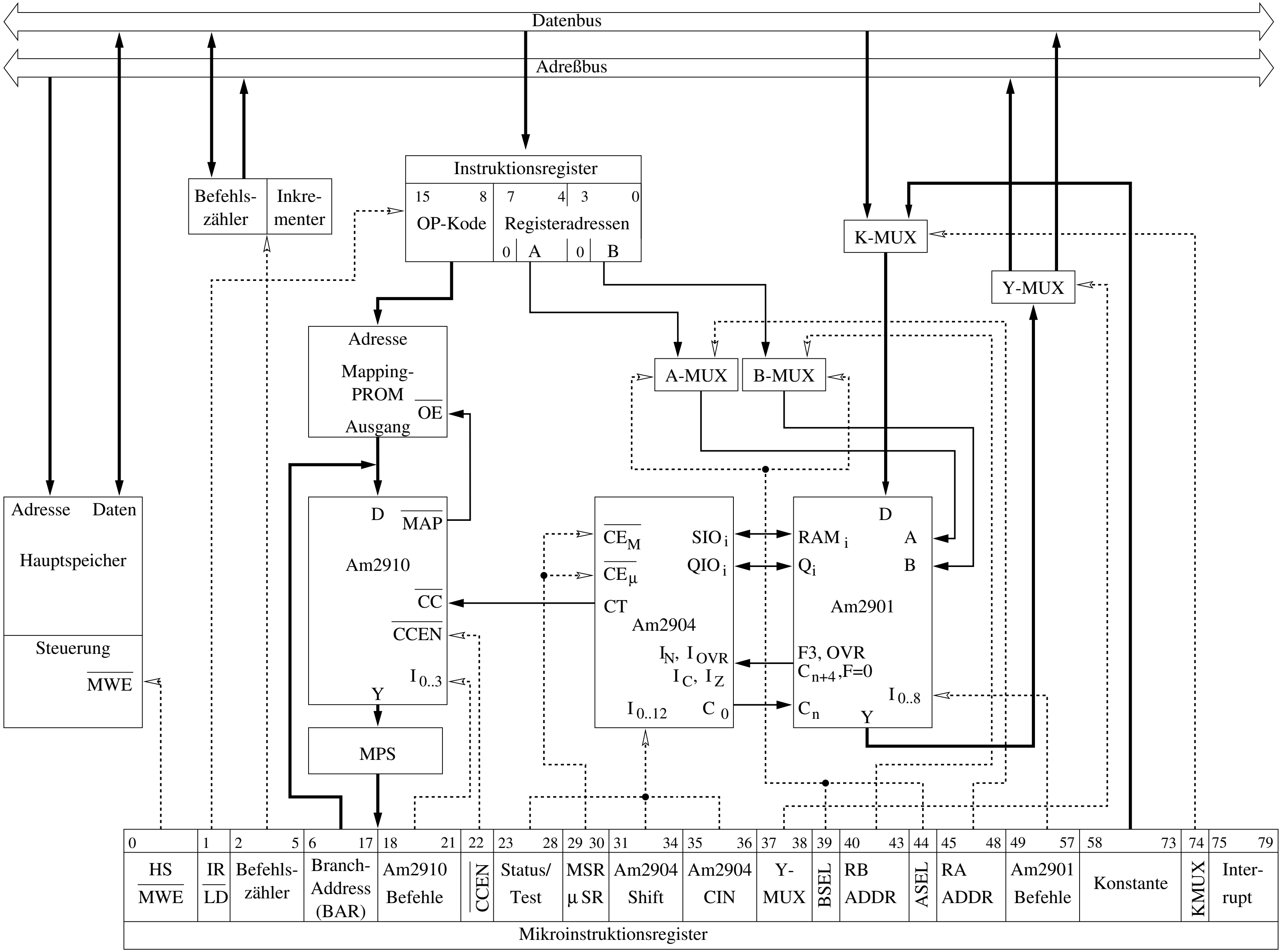


Abbildung 4.1: Blockschaltbild des mikroprogrammierbaren Beispielrechners

4.1.1 Das Leitwerk

Das Leitwerk ist aus dem Sequencer–Baustein Am2910 (Mikroleitwerk) und dem Mikroprogrammspeicher mit dem Mikroinstruktionsregister aufgebaut. Weiterhin enthält das Leitwerk einen 16 Bit breiten Befehlszähler (BZ) mit einem Inkrementierer und ein 16 Bit breites Instruktionsregister mit dem in Abbildung 4.1 dargestellten Format. Der Inhalt des Befehlszählers kann um 1 inkrementiert werden und auf den Adreß– und/oder Datenbus ausgegeben werden. Darüberhinaus können Daten vom Datenbus in den Befehlszähler übernommen werden.

4.1.1.1 Beschreibung des Sequencer–Bausteins Am2910

Der Sequencer-Baustein Am2910 hat die Aufgabe, in einem Taktzyklus die Adresse für die nächste auszuführende Mikroinstruktion zu generieren.

Damit auch mit Mikroprogrammen die von Maschinenprogrammen her bekannten Kontrollflußstrukturen (sequentieller Programmfluß, bedingte und unbedingte Sprünge sowie Schleifen) programmiert werden können enthält der Sequencer–Baustein Am2910 alle für eine komplexe Adreßbildung notwendigen Komponenten: den Adreßinkrementierer für die lineare Adreßfortschaltung, den Adreßkeller für Rücksprungadressen von Mikro-Unterprogrammen oder für Schleifenanfangsadressen, einen Schleifenzähler, den Folgeadreßmultiplexer für die Auswahl der verschiedenen Adreßquellen und eine Bedingungslogik für bedingte Aktionen. Darüber hinaus ist der Mikrobefehlszähler auf dem Baustein integriert.

Die nachfolgende Beschreibung der Befehle beschränkt sich auf die zu Verständnis der Funktionsweise des Bausteins notwendigen Informationen. Abbildung 4.2 zeigt das Blockschaltbild des Bausteins und in Tabelle 4.1 sind die möglichen Mikrooperationen zur Steuerung des Bausteins aufgeführt.

Als zentrales Element für die Auswahl der Folgeadresse arbeitet ein **Folgeadreßmultiplexer**, der genau eine von fünf möglichen Adreßquellen auswählen kann: den Mikrobefehlszähler, den Rücksprungadreßkeller, das multifunktionale Zählerregister, den am D–Eingang anliegenden Adreßwert oder die Konstante 0. Der Folgeadreßmultiplexer wird durch die im Mikroinstruktionswort (Feld 18 – 21) stehende Mikrooperation sowie bei bedingten Befehlen durch die Bedingungslogik auf dem Baustein gesteuert.

Der **Mikrobefehlszähler** dient zur linearen Adreßfortschaltung. Die vom Adreßmultiplexer ausgewählte Adresse wird zum einen über den Y–Ausgang an den Mikroprogrammspeicher gegeben und zum anderen in einem Inkrementierer um 1 erhöht, womit eine mögliche Folgeadresse im Mikrobefehlszähler für den nächsten Zyklus bereitsteht. Der Inhalt des Mikrobefehlszählers wird dann ausgewählt, wenn im Mikroinstruktionswort (Feld 18 – 21) die Mikrooperation CONT[1] (für Continue) steht.

Der fünf Stufen tiefe **Rücksprungadreßkeller** wird für das Zwischenspeichern von Rücksprungadressen bei (geschachtelten) Mikro-Unterprogrammen und von Anfangsadressen bei Mikroprogrammschleifen verwendet. Der Rücksprungadreßkeller wird über einen Kellerzeiger verwaltet, welcher immer auf das letzte in den Keller geschriebene Wort verweist. Die zulässigen Operationen auf dem Keller sind PUSH und POP. Bei PUSH wird die im Mikrobefehlszähler bereitstehende Adresse als neues Kellerelement oben auf den Keller geschrieben. Bei POP wird das oberste Kellerelement entfernt und sein Inhalt erscheint am F–Eingang des Folgeadreßmultiplexers.

Das multifunktionale **Zählerregister** dient sowohl zur Zwischenspeicherung von Adressen als auch als Schleifenzähler. Dieses Register wird über den 12 Bit breiten D–Eingang des Bausteins geladen.

Der am **D–Eingang** anliegende Bus ist ein Tri–State–Bus, so daß prinzipiell mehrere Quellen auf ihn geschaltet werden können. Die Standardquellen sind das Direktdatenfeld (Branch–Address–Feld, BAR) des Mikroinstruktionsregisters oder ein Abbildungsspeicher (Mapping–PROM). Eine weitere Adreßquelle, mit welcher der D–Eingang beschaltet werden kann, ist typischerweise ein „Vector–Mapping–PROM“, das aus einem Unterbrechungseingangsregister eine Startadresse für ein Mikroprogramm produziert, das die Unterbrechungsbehandlung dann durchführt. (Die Komponenten zur Unterbrechungsbehandlung sind der Einfachheit wegen nicht im Blockschaltbild unseres Beispielrechners aufgeführt). Die Aktivierung einer

[1] Es werden im nachfolgenden Text anstelle der weniger lesbaren Bitmuster die mnemotechnischen Namen verwendet.

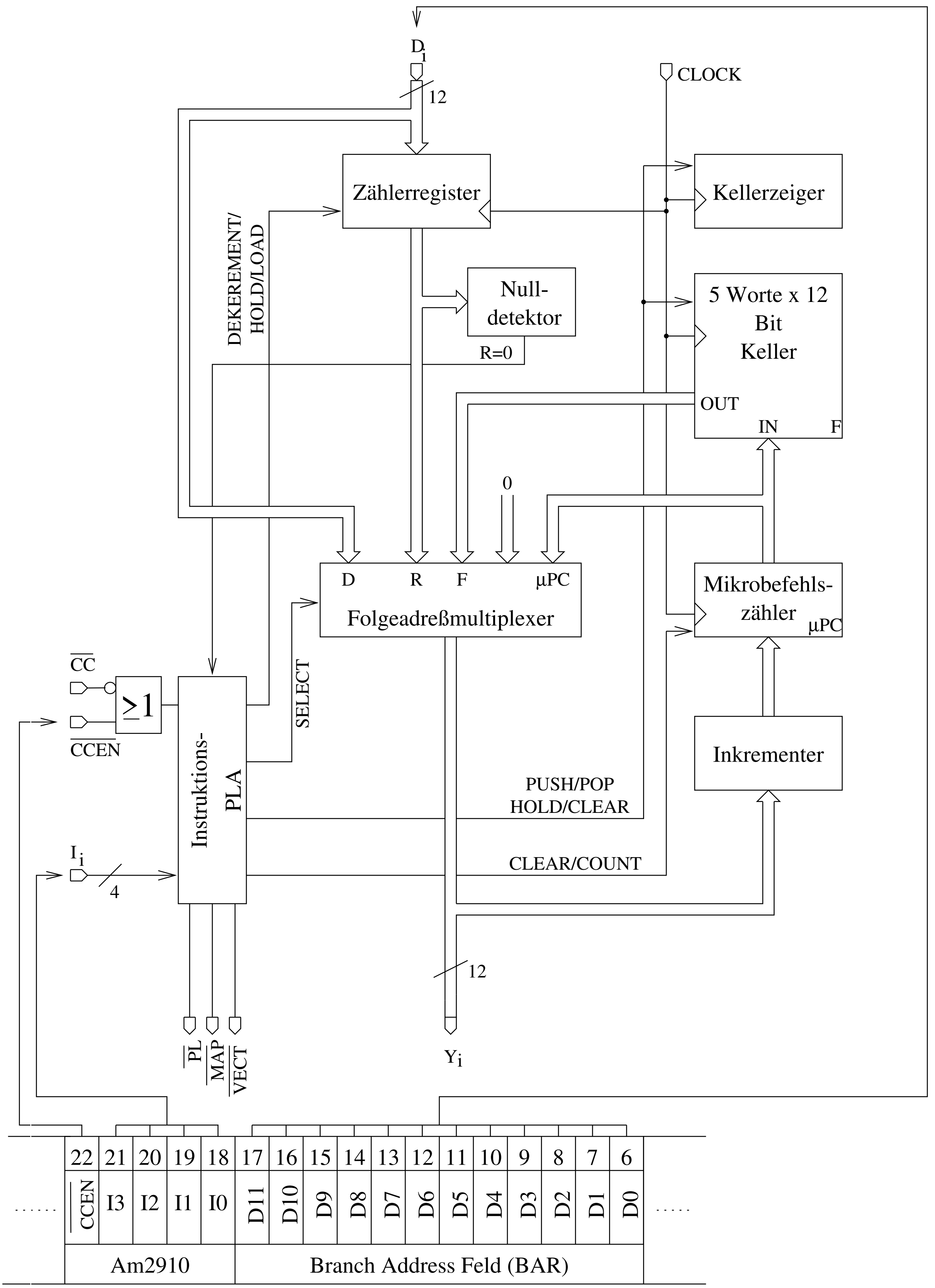


Abbildung 4.2: Funktionsschaltbild des Mikroleitwerks (Sequencer) Am2910

dieser Quellen übernimmt die bausteininterne Befehlsdekodierung. Wird die Folgeadresse aus dem Direktdatenfeld des Mikroinstruktionsregisters übernommen, so handelt es sich um einen expliziten Sprung (gegebenenfalls in Abhängigkeit einer Bedingung), da das Sprungziel als Absolutadresse in der laufenden Mikroinstruktion angegeben wird. Wenn die Folgeadresse vom „Mapping–PROM“ geliefert wird, ist es die Einsprungadresse in ein neues Mikroprogramm, dessen Startadresse über die Abbildung im Mapping–PROM aus dem Operationskodeteil des Instruktionsregisters gebildet wird (Dekodierphase des Makroinstruktionszyklus).

Soll beispielsweise eine Mikroprogrammspeicheradresse, die als Sprungziel im BAR-Feld des Mikroinstruktionsregisters steht, ausgewählt werden, muß im Mikroinstruktionswort (Feld 18 – 21) die Operation CJP (für Conditional Jump Pipeline) stehen.

Die für die **bedingte Adreßfortschaltung** notwendigen Statussignale werden vom Bedingungsmultiplexer des Bausteins Am2904 (siehe Abschnitt 4.1.2.2) ausgewählt und liegen am $\overline{CC}$–Eingang des Bausteins Am2910 an. Die Bedingungslogik prüft das am $\overline{CC}$–Eingang anliegende Signal nur, wenn der Mikroprogrammierer im Mikroinstruktionswort (Feld 22) $\overline{CCEN}$ aktiv (d. h. =0) setzt. Ansonsten werden die bedingten Operationen unbedingt ausgeführt.

Die 16 möglichen **Adreßfortschaltbefehle** des Bausteins, die durch die Instruktionsbit I_0 bis I_3 kodiert werden, sind in der Tabelle 4.1 beschrieben. Die in der Spalte „Mnemo“ angegebenen Namen sind für die Operationen sinnvollerweise vereinbart worden.

I_3–I_0	MNEMO	Name	Z/ Reg. Inh.	FAIL 1) Y	FAIL 1) Keller	PASS 2) Y	PASS 2) Keller	Z/ Reg.	OE En.
0	JZ	Jump Zero	X	0	clear	0	clear	hold	PL
1	CJS	Cond JSB PL	X	μPC	hold	D	push	hold	PL
2	JMAP	Jump Map	X	D	hold	D	hold	hold	MAP
3	CJP	Cond Jump PL	X	μPC	hold	D	hold	hold	PL
4	PUSH	Push/Cond Ld Cntr	X	μPC	push	μPC	push	3)	PL
5	JSRP	Cond JSB R/PL	X	R	push	D	push	hold	PL
6	CJV	Cond Jump Vector	X	μPC	hold	D	hold	hold	VEC
7	JRP	Cond Jump R/PL	X	R	hold	D	hold	hold	PL
8	RFCT	Repeat Loop CNTR $\neq 0$	$\neq 0$	F	hold	F	hold	dec	PL
			$= 0$	μPC	pop	μPC	pop	hold	PL
9	RPCT	Repeat PL CNTR $\neq 0$	$\neq 0$	D	hold	D	hold	dec	PL
			$= 0$	μPC	hold	μPC	hold	hold	PL
10	CRTN	Cond Rtn	X	μPC	hold	F	pop	hold	PL
11	CJPP	CJP & Pop	X	μPC	hold	D	pop	hold	PL
12	LDCT	Ld Cntr & Cont	X	μPC	hold	μPC	hold	load	PL
13	LOOP	Test End Loop	X	F	hold	μPC	pop	hold	PL
14	CONT	Continue	X	μPC	hold	μPC	hold	hold	PL
15	TWB	3–Way–Branch	$\neq 0$	F	hold	μPC	pop	dec	PL
			$= 0$	D	pop	μPC	pop	hold	PL

1): $\overline{CCEN} = L$ and $\overline{CC} = H$ X: unerheblich, “Don't Care”
2): $\overline{CCEN} = H$ or $\overline{CC} = L$
3): if $\overline{CCEN} = L$ and $\overline{CC} = H$, hold; else load.

Tabelle 4.1: Adreßfortschaltbefehle des Mikroleitwerks AM2910

Drei der hier aufgeführten Fortschaltbefehle hängen vom Inhalt des Zählerregisters ab, so daß für diese Befehle entsprechend den zwei Alternativen „$= 0$“ und „$\neq 0$“ jeweils die unterschiedlichen Aktionen beschrieben sind.

Für zehn Sequencer–Befehle bestimmt das am $\overline{CC}$–Eingang anliegende Signal die auszuführenden Aktionen. Wenn die Bedingung erfüllt ist (PASS–Bedingung), werden die in der Doppelspalte „PASS“ angegebenen Aktionen ausgeführt. Ist die Bedingung nicht erfüllt (FAIL–Bedingung), gelten die in der Doppelspalte „FAIL“ angeführten Aktionen. In der Y–Spalte wird jeweils die Quelle der vom Folgeadreßmultiplexer auszuwählenden und am Y–Ausgang anliegenden Adresse angegeben. In der Spalte „Keller“ sind die jeweiligen Operationen auf dem Keller angegeben (CLEAR für zurücksetzen, PUSH, POP und HOLD für „keine Aktion“).

Die beiden letzten Spalten erklären die Aktionen auf dem Zählerregister und das vom internen Steuerblock (Instruktions–PLA) generierte Enable–Signal für eine der möglichen Quellen für den D–Eingang (MAP für das Mapping–PROM, PL für das Direktdatenfeld des Mikroinstruktionsregisters und VEC für das Vector–Mapping–PROM).

Der Befehl JZ produziert die **Mikroprogrammspeicheradresse 0**. An dieser Adresse kann sinnvollerweise ein Mikroprogramm zum Initialisieren des Systems stehen.

Die Befehle JMAP und CJV erlauben den **Einsprung in Mikroprogramme**, deren Startadressen von externen Adreßquellen (Mapping–PROM, Vector–Mapping–PROM) über den D–Eingang geliefert und an den Mikroprogrammspeicher weitergegeben werden.

Die **sequentielle Adreßfortschaltung** im laufenden Mikroprogramm geschieht mit dem Befehl CONT. Sprungbefehle sind CJP (Sprungziel steht im Direktdatenfeld des Mikroinstruktionsregister) und JRP (Sprungziele stehen im Direktdatenfeld des Mikroinstruktionsregister oder im Zählerregister, das vorher allerdings mit der Zieladresse geladen werden muß).

Mikro-Unterprogramme werden mit den Befehlen CJS (Anfangsadresse des Unterprogramms aus dem Mikroinstruktionsregister) und JSRP (Anfangsadresse alternativ aus dem Mikroinstruktionsregister oder aus dem Zählerregister) aufgerufen, wobei automatisch die Rücksprungadresse auf das oberste Kellerelement geschrieben wird. Bedingte Rücksprünge sind dann mit dem Befehl CRTN möglich, der die Rücksprungadresse vom Keller holt.

Eine Reihe von Befehlen dient der **Schleifenprogrammierung**. Der Befehl PUSH lädt in Abhängigkeit einer externen Bedingung das Zählerregister und legt die Schleifenanfangsadresse im Keller ab. Der Befehl LDCT lädt nur den Zähler. Die Befehle RPCT, RFCT und TWB sind Schleifenendebefehle, die zunächst prüfen, ob der Inhalt des Zählerregisters 0 ist. Wenn der Inhalt des Zählerregisters ungleich 0 ist, wird an die im Direktdatenfeld des Mikroinstruktionsregisters angegebene bzw. an die auf dem Keller hinterlegte Schleifenanfangsadresse verzweigt und der Inhalt des Zählerregisters um 1 erniedrigt. Bei dem Befehl TWB ist das Abfragen des Zählerregisters kombiniert mit dem Test einer externen Bedingung, so daß eine Dreiwegeverzweigung vorliegt. Der Befehl LOOP springt in Abhängigkeit einer externen Bedingung an die auf dem obersten Kellerelement liegende Schleifenanfangsadresse. Das Herunterzählen des Zählerregisters und die Verwaltung des Kellers erfolgt automatisch.

4.1.2 Das mikroprogrammierbare Rechenwerk

Zur Verarbeitung der Daten ist ein 16 Bit breites Rechenwerk vorgesehen. Es ist auf der Basis von Rechenwerkbausteinen, den kaskadierbaren Bitslice–Komponenten Am2901, aufgebaut. Vier dieser Module, ergänzt durch den Wortrandlogikbaustein Am2904, bilden das 16 Bit breite Rechenwerk. Seine Daten erhält das Rechenwerk vom Hauptspeicher, mit dem es über den Datenbus verbunden ist. Ergebnisse können auf den Datenbus und auf den Adreßbus, falls es sich um Adressen handelt, ausgegeben werden.

4.1.2.1 Beschreibung des Rechenwerkbausteins Am2901

Abbildung 4.3 zeigt die Grundstruktur des Rechenwerkbausteins Am2901. Da es sich um einen kaskadierbaren 4–Bit–Baustein handelt, seine Datenpfade und internen Komponenten also 4 Bit breit sind, können durch Aneinanderreihung solcher Module Rechenwerke mit einem um ein Vielfaches von vier Bit breiten Datenwort aufgebaut werden.

Als zentrale Komponenten enthält der Baustein eine Zweitor–Registerdatei (RAM) mit 16 Registern und

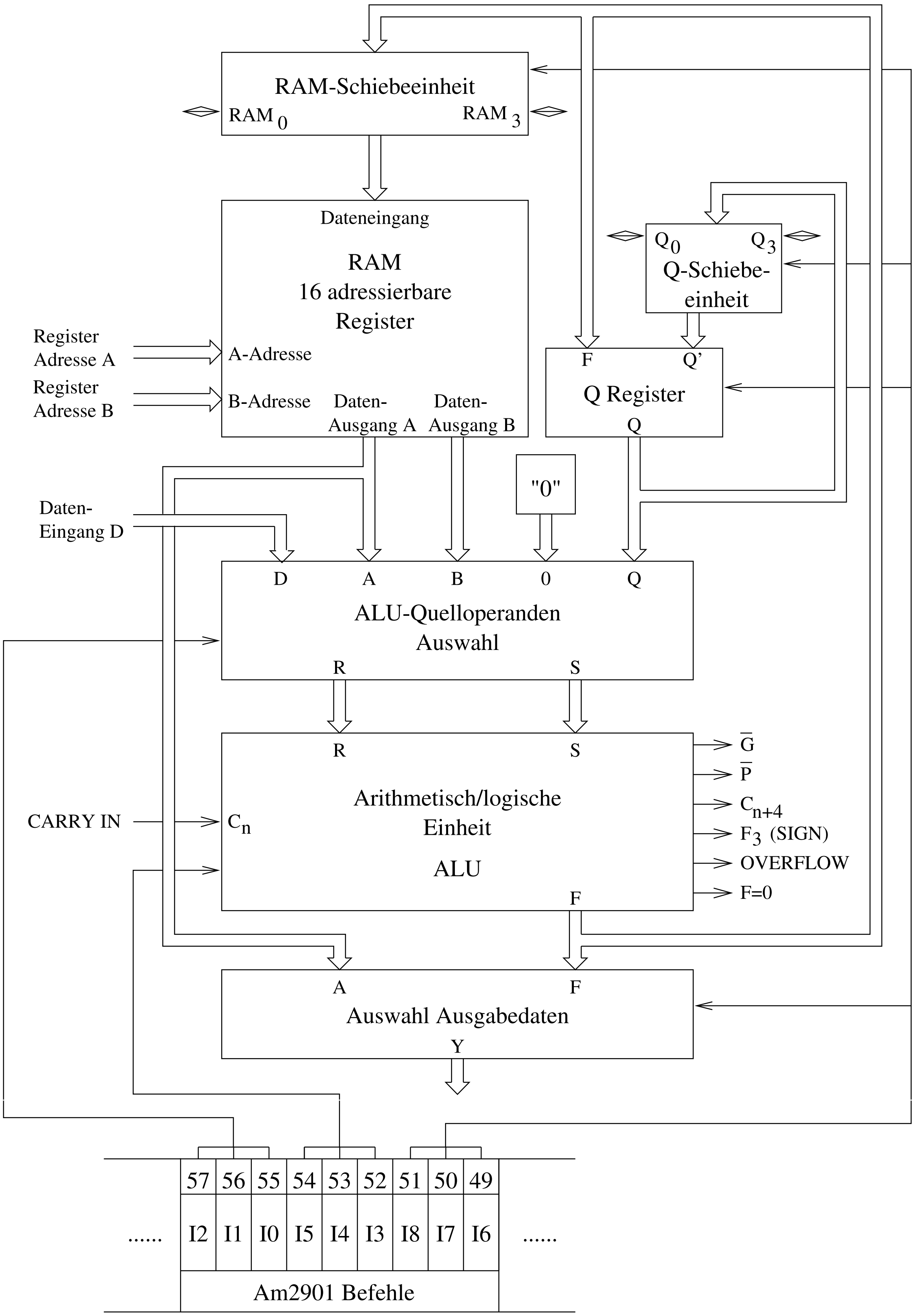


Abbildung 4.3: Funktionsschaltbild des Rechenwerkbausteins Am2901

Mnemo	Mikrokode				ALU–Quell–operanden	
	I_2	I_1	I_0	Oktal–wert	R	S
AQ	0	0	0	0	A	Q
AB	0	0	1	1	A	B
ZQ	0	1	0	2	0	Q
ZB	0	1	1	3	0	B
ZA	1	0	0	4	0	A
DA	1	0	1	5	D	A
DQ	1	1	0	6	D	Q
DZ	1	1	1	7	D	0

Tabelle 4.2: ALU Quelloperandensteuerung

Mnemo	Mikrokode				ALU Funktion	Symbol
	I_5	I_4	I_3	Oktal–wert		
ADD	0	0	0	0	R Plus S	R+S
SUBR	0	0	1	1	S Minus R	S−R
SUBS	0	1	0	2	R Minus S	R−S
OR	0	1	1	3	R OR S	$R \vee S$
AND	1	0	0	4	R AND S	$R \wedge S$
NOTRS	1	0	1	5	$\overline{R}$ AND S	$\overline{R} \wedge S$
EXOR	1	1	0	6	R XOR S	$R \oplus S$
EXNOR	1	1	1	7	R XNOR S	$\overline{R \oplus S}$

Tabelle 4.3: ALU Funktionssteuerung

Mnemo	Mikrokode				RAM Funktion		Q-Reg. Funktion		Y Ausgang
	I_8	I_7	I_6	Oktal–wert	Schieben	Laden	Schieben	Laden	
QREG	0	0	0	0	X	NONE	NONE	F→Q	F
NOP	0	0	1	1	X	NONE	X	NONE	F
RAMA	0	1	0	2	NONE	F→B	X	NONE	A
RAMF	0	1	1	3	NONE	F→B	X	NONE	F
RAMQD	1	0	0	4	DOWN	F/2 →B	DOWN	Q/2 →Q	F
RAMD	1	0	1	5	DOWN	F/2 →B	X	NONE	F
RAMQU	1	1	0	6	UP	2F→B	UP	2Q→Q	F
RAMU	1	1	1	7	UP	2F→B	X	NONE	F

Tabelle 4.4: ALU Zielsteuerung (DOWN = Rechtsschieben, UP = Linksschieben)

			I_{210}							
			AQ	AB	ZQ	ZB	ZA	DA	DQ	DZ
			0	1	2	3	4	5	6	7
Mnemo	I_{543}	ALU Funktion	ALU Quelloperanden							
			A,Q	A,B	0,Q	0,B	0,A	D,A	D,Q	D,0
ADD	0	$C_n = 0$	A+Q	A+B	Q	B	A	D+A	D+Q	D
		R+S								
		$C_n = 1$	A+Q+1	A+B+1	Q+1	B+1	A+1	D+A+1	D+Q+1	D+1
SUBR	1	$C_n = 0$	Q−A−1	B−A−1	Q−1	B−1	A−1	A−D−1	Q−D−1	−D−1
		S−R								
		$C_n = 1$	Q−A	B−A	Q	B	A	A−D	Q−D	−D
SUBS	2	$C_n = 0$	A−Q−1	A−B−1	−Q−1	−B−1	−A−1	D−A−1	D−Q−1	D−1
		R−S								
		$C_n = 1$	A−Q	A−B	−Q	−B	−A	D−A	D−Q	D
OR	3	$R \vee S$	$A \vee Q$	$A \vee B$	Q	B	A	$D \vee A$	$D \vee Q$	D
AND	4	$R \wedge S$	$A \wedge Q$	$A \wedge B$	0	0	0	$D \wedge A$	$D \wedge Q$	0
NOTRS	5	$\overline{R} \wedge S$	$\overline{A} \wedge Q$	$\overline{A} \wedge B$	Q	B	A	$\overline{D} \wedge A$	$\overline{D} \wedge Q$	0
EXOR	6	$R \oplus S$	$A \oplus Q$	$A \oplus B$	Q	B	A	$D \oplus A$	$D \oplus Q$	D
EXNOR	7	$\overline{R \oplus S}$	$\overline{A \oplus Q}$	$\overline{A \oplus B}$	$\overline{Q}$	$\overline{B}$	$\overline{A}$	$\overline{D \oplus A}$	$\overline{D \oplus Q}$	$\overline{D}$

Die Instruktionsbit I_{543} und I_{210} sind in Oktaldarstellung.

Tabelle 4.5: Quelloperanden– und ALU–Funktionsmatrix eine arithmetisch/logische Einheit (ALU), die acht Mikrooperationen ausführen kann. Die Operanden liegen an den ALU–Eingängen R und S an und können fünf verschiedenen Quellen entstammen. Weiterhin verfügt der Baustein Am2901 über ein multifunktionales Q–Register sowie zwei Schiebeeinheiten (RAM– und Q–Schiebeeinheit).

Durch Anlegen von A– und B–Registeradressen an die **Registerdatei** können jeweils zwei beliebige – auch identische – Register über die beiden Ausgabewege A und B gelesen werden. Das Ergebnis einer ALU–Operation wird in das Register geschrieben, welches im Register–B–Adreßfeld des Mikroinstruktions– bzw. des Instruktionsregisters spezifiziert ist. Der Weg vom ALU–F–Ausgang zur Registerdatei führt über die **RAM–Schiebeeinheit**, wo die Daten um ein Bit nach links bzw. nach rechts geschoben oder unverändert weitergegeben werden. Das Lesen der Register, die Verknüpfung der Operanden in der ALU und das Zurückschreiben des Ergebnisses erfolgt innerhalb eines Taktzyklus.

Die arithmetisch/logische Einheit kann, gesteuert über die Instruktionsbit I_3 bis I_5, auf den am ALU–R– und am ALU–S–Eingang anliegenden Daten drei arithmetische und fünf logische Operationen ausführen, wobei für die arithmetischen Operationen zu beachten ist, daß diese noch vom Übertrag, d. h. vom Wert des am C_n–Eingang anliegenden Signals, abhängen (siehe Tabelle 4.5). Die Operationen der ALU können der internen Registerdatei (A–Ausgang, B–Ausgang), dem Q–Register und einem unidirektionalen D–Eingang entstammen. Weiterhin kann die Null als Quelloperand auftreten.

Das **Q–Register** kann zur Speicherung von Zwischenergebnissen, als Quotientenregister bei Multiplikations– und Divisionsroutinen oder als Akkumulatorregister dienen.

Über den **D–Eingang** können Daten aus einer externen Quelle eingelesen werden. In unserem Beispielrechner sind dies Daten, welche aus dem Hauptspeicher über den Datenbus geliefert werden oder Konstanten, die im Konstantenfeld (Bit 58 - 73) des Mikroinstruktionsregisters stehen. Über das Bit 74 des Mikroinstruktionswortes wird eine der beiden Möglichkeiten ausgewählt.In Tabelle 4.2 ist die Definition der Instruktionsbit I_0 bis I_2 zur Steuerung der Auswahl der Quelloperanden für die acht möglichen Kombinationen zu entnehmen. Die Tabelle 4.3 zeigt die Wirkung der ALU auf die Quelloperanden (ALU Funktionssteuerung).

Das Ergebnis einer ALU–Operation kann direkt über den **Y–Ausgang** ausgegeben oder verschiedenen internen Komponenten zugeführt werden. Die am ALU–F–Ausgang anliegenden Ergebnisse können, wie oben bereits beschrieben, in die Registerdatei oder in das Q–Register geschrieben werden. Die in das Q–Register zu schreibenden Daten können zuvor im **Q–Schiebewerk** um ein Bit nach rechts oder nach links geschoben werden. Die am Y–Ausgang des Bausteins anliegenden Daten stammen entweder vom ALU–F–Ausgang (Ergebnis einer ALU–Operation) oder vom A–Ausgang der Registerdatei (Inhalt einer Registerzelle). Die Zielsteuerung, programmiert über die Instruktionsbit I_6 bis I_8, ist in Tabelle 4.4 definiert. Mit der Codierung RAMF der ALU-Zielsteuerung wird beispielsweise bestimmt, daß das Ergebnis am ALU-F-Ausgang in das Register geschrieben wird, das im Register-B-Adreßfeld spezifiziert wird und gleichzeitig über den Y-Bus ausgegeben wird. Im Unterschied dazu wird bei der Codierung RAMA der Inhalt des im Register–A–Adreßfeld spezifizierten Registers über den Y-Bus ausgegeben.

Der Baustein liefert eine Reihe von Statussignalen, die in einem der Statusregister des Bausteins Am2904 abgelegt werden können (siehe Abschnitt 4.1.2.2).

4.1.2.2 Der Wortrandlogikbaustein Am2904

Der Wortrandlogikbaustein Am2904 wird als Ergänzung zu dem Rechenwerkbaustein Am2901 verwendet, da er die im Zusammenhang mit der ALU notwendigen Funktionen wie die Steuerung des Übertrags, der Schiebeverbindungen, des Ablegens der Statussignale und des Abfragens von Bedingungen auf einem Chip integriert. Abbildung 4.4 zeigt vereinfacht den Aufbau des Wortrandlogikbausteins Am2904.

Der Baustein enthält zwei **Statusregister**, das Maschinenstatusregister (MSR) und das Mikrostatusregister (μSR), die unabhängig voneinander gesteuert und bitweise mit den entsprechenden Statussignalen wie Überlauf (OVR), Übertrag (C), Vorzeichen (N) und Null–Anzeige (Z) aus dem Rechenwerk (Ausgänge OVR, C_{n+4}, F_3, $F = 0$ des höchstwertigen Rechenwerkbausteins) über die vier Statuseingänge I_C, I_N, I_Z

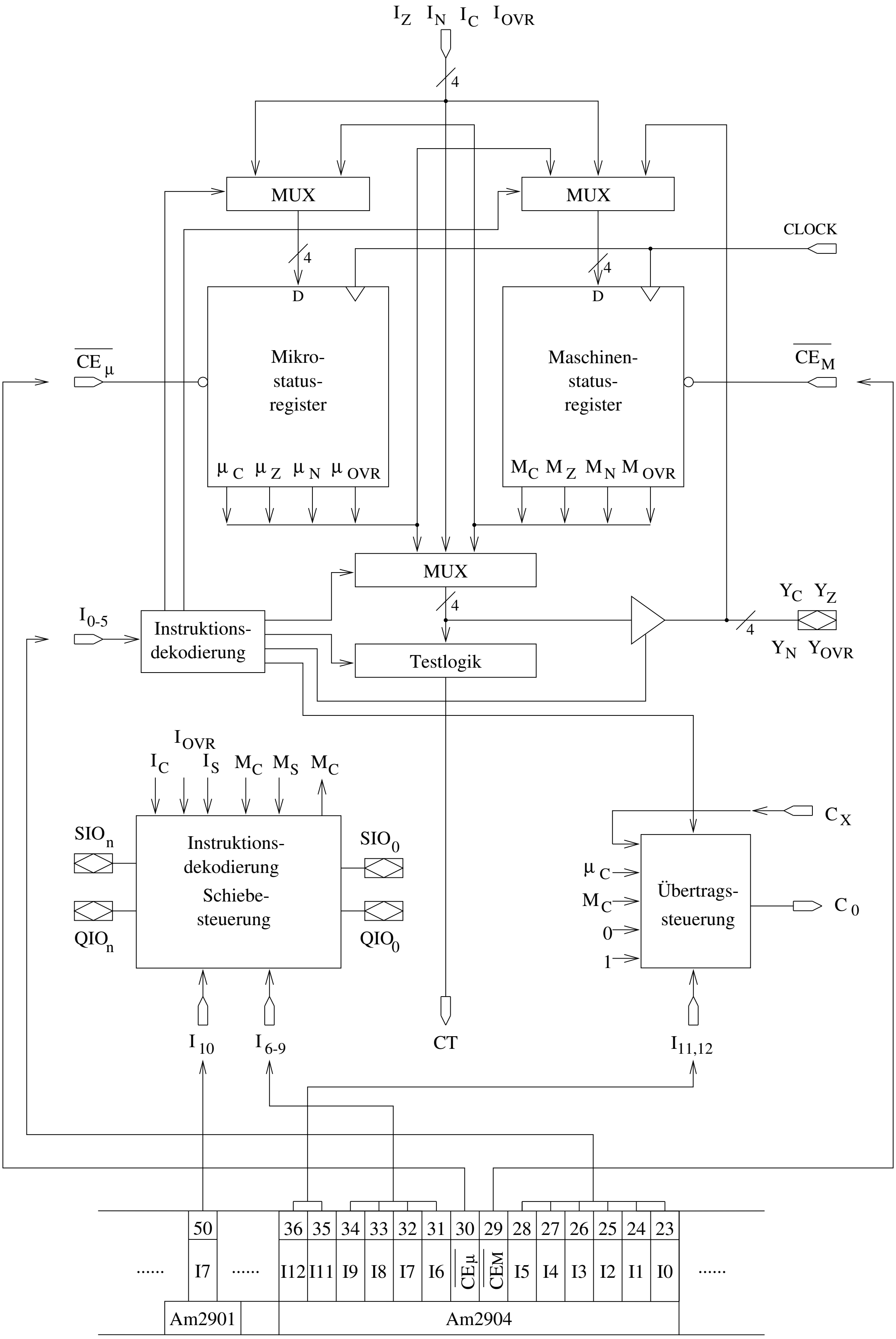


Abbildung 4.4: Funktionsschaltbild des Wortrandlogikbausteins Am2904

und I_{OVR} geladen werden können. Einzelne Bit können gesetzt bzw. zurückgesetzt und die Inhalte der beiden Register ausgetauscht werden, wobei Enablebit das Überschreiben der Register verhindern oder ermöglichen.

Die beiden Statusregister werden über die Instruktionsbit I_0 bis I_5 gesteuert. Die **Operationen auf den Statusregistern** lassen sich in die Gruppen Bit– (nur Mikrostatusregister), Register– und Lade–Operationen einteilen. Tabelle 4.6 zeigt die vollständige Decodierung dieser Instruktionsbit; ausgelassen ist lediglich ihr Einfluß auf die Übertragssteuerung (s. Tabelle 4.9). Die mit μSR, MSR und CT beschrifteten Zeilen der Tabelle geben jeweils an, wie sich die Operation auf die Statusregister und die Testlogik (CT-Ausgang) auswirken. Man erkennt aus der Tabelle, daß die Codierung der Operationen relativ komplex und insbesondere nicht orthogonal ist (Aus diesem Grund gibt es auch keine mnemotechnischen Bezeichner für die Operationen). Zur Vereinfachung sind jedoch die wichtigsten Operationen auf den Statusregistern in Tabelle 4.7 zusammengestellt. Zu beachten ist, daß die dort angegebenen möglichen Operationscodes jeweils auch Nebeneffekte verursachen, die aus Tabelle 4.6 entnommen werden können. So löscht Z.B. der Code $I_{5..0} = 001000$ nicht nur das Z-Bit im Mikrostatusregister, sondern lädt auch das Maschinenstatusregister mit invertiertem C-Bit vom I-Eingang und setzt den CT-Ausgang entsprechend dem Ergebnis der Bedingung $\mu_C \vee \mu_Z = 0$. Ein unerwünschtes Verändern der Statusregister kann mit Hilfe der Enablebit $\overline{CE_\mu}$ und $\overline{CE_M}$ (Bit 30 bzw. 29 des Mikroinstruktionsregisters) verhindert werden. Ein Statusregister wird nur dann verändert, wenn das entsprechende Enablebit auf LOW gesetzt ist.

Die Testlogik auf dem Wortrandlogikbaustein Am2904 gestattet die **Abfrage der Statusregister** μSR und MSR. Der ausgewählte Bedingungskode wird am Ausgang CT ausgegeben. Der CT–Ausgang ist mit dem Testeingang $\overline{CC}$ des Sequencerbausteins Am2910 verbunden.

Über die Instruktionsbit I_4 und I_5 kann bestimmt werden, in welchem Statusregister die Bedingungen abgefragt werden sollen. Mit $I_{5,4} = 01$ (Mnemo: MI) werden die Statusbit des Mikrostatusregisters abgefragt und mit $I_{5,4} = 10$ (Mnemo: MA) die des Maschinenstatusregisters. Mit den Instruktionsbit I_0 bis I_3 wird eine von 16 Bedingungen ausgewählt (siehe die CT-Zeilen in Tabelle 4.6). Acht dieser 16 Bitkombinationen liefern den Wert eines der Statusbit oder dessen Komplement. Die acht restlichen Bitkombinationen liefern als Ergebnis logische Verknüpfungen von Statusbit an den Ausgang CT.

Diese Bitkombinationen gestatten beispielsweise den Vergleich zweier nicht vorzeichenbehafteter oder Zweierkomplement–Zahlen nach den Kriterien „gleich“, „größer gleich“, „kleiner“ oder „kleiner gleich“. In Tabelle 4.8 sind als Beispiel verschiedene Kriterien aufgeführt, die nach der Subtraktion zweier Zahlen A und B geprüft werden können, wobei zwischen nicht vorzeichenbehafteten und Zweierkomplement–Zahlen unterschieden wird. Für jede Relation (Spalte 1) wird der Wert des bzw. der betreffenden Statusbit (Spalte 2 und 6) angegeben. In den Spalten 4,5 und 8,9 stehen die Bedingungscodes (jeweils oktal und hexadezimal), die zu programmieren sind, wenn die entsprechenden Bedingungen abzufragen sind. Wenn die Bedingung erfüllt ist, dann erscheint am CT-Ausgang und damit auch am Testeingang $\overline{CC}$ des Sequencerbausteins Am2910 eine Null. In den Spalten 3 und 7 sind die wichtigsten Mnemos für die Bedingungscodes aufgeführt.

Bezüglich des Zustands des **C-Statusbits nach der Subtraktion** vorzeichenloser Zahlen ist eine Besonderheit der ALU im Am2901 zu berücksichtigen. Die ALU realisiert intern eine Subtraktion $A - B$ duch Addition von A mit dem Komplement von B. Dies bedingt, daß bei der Subtraktion vorzeichenloser Zahlen das C-Statusbit entgegen der üblichen Erwartung dann gesetzt wird, wenn A größer oder gleich B ist (siehe Tabelle 4.8).

Falls vom Rechenwerk eine **Schiebeoperation** durchgeführt werden muß, so ist neben der entsprechenden Operation der ALU-Zielsteuerung (siehe Tabelle 4.4) auch der Wortrandlogikbaustein Am2904 zu programmieren.

Der Baustein erlaubt bei entsprechender Verbindung mit den Rechenwerkbausteinen 32 verschiedene Schiebe– und Ringschiebeaktionen zu mikroprogrammieren. Die Schiebeein– bzw. ausgänge SIO_0, SIO_n, QIO_0 und QIO_n sind mit den entsprechenden Eingängen der Rechenwerkbausteinen verbunden (SIO_0 mit RAM_0 des niedrigstwertigen Am2901–Bausteins, SIO_n mit RAM_3 des höchstwertigen Am2901–Bausteins, QIO_0 mit QIO_0 des niedrigstwertigen Am2901–Bausteins und QIO_n mit QIO_n des höchstwertigen Am2901–Bausteins, siehe Abbildung 4.5).

		I_{210}							
I_{543}		000	001	010	011	100	101	110	111
000	μSR	$M_X \to \mu_X$	$1 \to \mu_X$	$M_X \leftrightarrow \mu_X$	$0 \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_{Z,C,N} \to \mu_{Z,C,N}$ $I_V \vee \mu_V \to \mu_V$	$I_{Z,C,N} \to \mu_{Z,C,N}$ $I_V \vee \mu_V \to \mu_V$
	MSR	$Y_X \to M_X$	$1 \to M_X$	$M_X \leftrightarrow \mu_X$	$0 \to M_X$	$I_{Z,N} \to M_{Z,N}$ $M_V \leftrightarrow M_C$	$\overline{M}_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$
	CT	$(\mu_N \oplus \mu_V) \vee \mu_Z = 0$	$(\mu_N \oplus \mu_V) \vee \mu_Z = 1$	$\mu_N \oplus \mu_V = 0$	$\mu_N \oplus \mu_V = 1$	$\mu_Z = 0$	$\mu_Z = 1$	$\mu_V = 0$	$\mu_V = 1$
001	μSR	$0 \to \mu_Z$	$1 \to \mu_Z$	$0 \to \mu_C$	$1 \to \mu_C$	$0 \to \mu_N$	$1 \to \mu_N$	$0 \to \mu_V$	$1 \to \mu_V$
	MSR	$I_{Z,N,V} \to M_{Z,N,V}$ $\overline{I}_C \to M_C$	$I_{Z,N,V} \to M_{Z,N,V}$ $\overline{I}_C \to M_C$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$
	CT	$\mu_C \vee \mu_Z = 0$	$\mu_C \vee \mu_Z = 1$	$\mu_C = 0$	$\mu_C = 1$	$\overline{\mu_C} \vee \mu_Z = 0$	$\overline{\mu_C} \vee \mu_Z = 1$	$I_N \oplus M_N = 0$	$I_N \oplus M_N = 1$
010	μSR	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$
	MSR	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$
	CT	$(\mu_N \oplus \mu_V) \vee \mu_Z = 0$	$(\mu_N \oplus \mu_V) \vee \mu_Z = 1$	$\mu_N \oplus \mu_V = 0$	$\mu_N \oplus \mu_V = 1$	$\mu_Z = 0$	$\mu_Z = 1$	$\mu_V = 0$	$\mu_V = 1$
011	μSR	$I_{Z,N,V} \to \mu_{Z,N,V}$ $\overline{I}_C \to \mu_C$	$I_{Z,N,V} \to \mu_{Z,N,V}$ $\overline{I}_C \to \mu_C$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$
	MSR	$I_{Z,N,V} \to M_{Z,N,V}$ $\overline{I}_C \to M_C$	$I_{Z,N,V} \to M_{Z,N,V}$ $\overline{I}_C \to M_C$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$
	CT	$\mu_C \vee \mu_Z = 0$	$\mu_C \vee \mu_Z = 1$	$\mu_C = 0$	$\mu_C = 1$	$\overline{\mu_C} \vee \mu_Z = 0$	$\overline{\mu_C} \vee \mu_Z = 1$	$\mu_N = 0$	$\mu_N = 1$
100	μSR	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$
	MSR	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$
	CT	$(M_N \oplus M_V) \vee M_Z = 0$	$(M_N \oplus M_V) \vee M_Z = 1$	$M_N \oplus M_V = 0$	$M_N \oplus M_V = 1$	$M_Z = 0$	$M_Z = 1$	$M_V = 0$	$M_V = 1$
101	μSR	$I_{Z,N,V} \to \mu_{Z,N,V}$ $\overline{I}_C \to \mu_C$	$I_{Z,N,V} \to \mu_{Z,N,V}$ $\overline{I}_C \to \mu_C$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$
	MSR	$I_{Z,N,V} \to M_{Z,N,V}$ $\overline{I}_C \to M_C$	$I_{Z,N,V} \to M_{Z,N,V}$ $\overline{I}_C \to M_C$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$
	CT	$M_C \vee M_Z = 0$	$M_C \vee M_Z = 1$	$M_C = 0$	$M_C = 1$	$\overline{M_C} \vee M_Z = 0$	$\overline{M_C} \vee M_Z = 1$	$M_N = 0$	$M_N = 1$
110	μSR	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$
	MSR	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$
	CT	$(I_N \oplus I_V) \vee I_Z = 0$	$(I_N \oplus I_V) \vee I_Z = 1$	$I_N \oplus I_V = 0$	$I_N \oplus I_V = 1$	$I_Z = 0$	$I_Z = 1$	$I_V = 0$	$I_V = 1$
111	μSR	$I_{Z,N,V} \to \mu_{Z,N,V}$ $\overline{I}_C \to \mu_C$	$I_{Z,N,V} \to \mu_{Z,N,V}$ $\overline{I}_C \to \mu_C$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$	$I_X \to \mu_X$
	MSR	$I_{Z,N,V} \to M_{Z,N,V}$ $\overline{I}_C \to M_C$	$I_{Z,N,V} \to M_{Z,N,V}$ $\overline{I}_C \to M_C$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$	$I_X \to M_X$
	CT	$\overline{I}_C \vee I_Z = 0$	$\overline{I}_C \vee I_Z = 1$	$I_C = 0$	$I_C = 1$	$\overline{I_C} \vee I_Z = 0$	$\overline{I_C} \vee I_Z = 1$	$I_N = 0$	$I_N = 1$

V steht abkürzend für OVR, X steht stellvertretend für $\{Z, C, N, OVR\}$

Tabelle 4.6: (Fast) vollständige Decodierung der Am2904 Instruktionsbits $I_{5..0}$

Operation	Beschreibung	I_{543210} oktal	I_{543210} hex
$I_X \to \mu_X$	Lade die am Statuseingang I anliegenden Signale ins μSR	04, 05 20 – 27 32 – 47 52 – 67 72 – 77	04, 05 10 – 17 1A – 27 2A – 37 3A – 3F
$I_Z \to \mu_Z$ $I_N \to \mu_N$ $I_{OVR} \to \mu_{OVR}$ $\overline{I_C} \to \mu_C$	Lade die am Statuseingang I anliegenden Signale ins μSR wobei das C-Bit invertiert wird	30, 31 50, 51 70, 71	18, 19 28, 29 38, 39
$I_X \to M_X$	Lade die am Statuseingang I anliegenden Signale ins MSR	06, 07 12 – 27 32 – 47 52 – 67 72 – 77	06, 07 0A – 17 1A – 27 2A – 37 3A – 3F
$I_Z \to M_Z$ $I_N \to M_N$ $I_{OVR} \to M_{OVR}$ $\overline{I_C} \to M_C$	Lade die am Statuseingang I anliegenden Signale ins MSR wobei das C-Bit invertiert wird	10, 11 30, 31 50, 51 70, 71	08, 09 18, 19 28, 29 38, 39
$1 \to \mu_X$	Setze alle Bit des μSR	01	01
$0 \to \mu_X$	Lösche alle Bit des μSR	03	03
$1 \to M_X$	Setze alle Bit des MSR	01	01
$0 \to M_X$	Lösche alle Bit des MSR	03	03
$\mu_X \leftrightarrow M_X$	Tausche die Inhalte von μSR und MSR	02	02
$1 \to \mu_Z$	Setze das Z-Bit im μSR	11	09
$0 \to \mu_Z$	Lösche das Z-Bit im μSR	10	08
$1 \to \mu_C$	Setze das C-Bit im μSR	13	0B
$0 \to \mu_C$	Lösche das C-Bit im μSR	12	0A
$1 \to \mu_N$	Setze das N-Bit im μSR	15	0D
$0 \to \mu_N$	Lösche das N-Bit im μSR	14	0C
$1 \to \mu_{OVR}$	Setze das OVR-Bit im μSR	17	0F
$0 \to \mu_{OVR}$	Lösche das OVR-Bit im μSR	16	0E

X steht stellvertretend für $\{Z, C, N, OVR\}$

Tabelle 4.7: Codierungen für wichtige Operationen der Statusregister

	Nicht vorzeichenbehaftete Zahlen				Zweierkomplementzahlen			
			I_{3210}				I_{3210}	
Relation	Status	Mnemo	oktal	hex	Status	Mnemo	oktal	hex
$A = B$	$Z = 1$	Zero	05	5	$Z = 1$	Zero	05	5
$A \neq B$	$Z = 0$	NotZero	04	4	$Z = 0$	NotZero	04	4
$A \geq B$	$C = 1$	UGTEQ	13	B	$\overline{N \oplus OVR} = 1$	SGTEQ	02	2
$A < B$	$C = 0$	ULT	12	A	$N \oplus OVR = 1$	SLT	03	3
$A > B$	$C \wedge \overline{Z} = 1$	UGT	14	C	$\overline{N \oplus OVR} \wedge \overline{Z} = 1$	SGT	00	0
$A \leq B$	$\overline{C} \vee Z = 1$	ULTEQ	15	D	$(N \oplus OVR) \vee Z = 1$	SLTEQ	01	1

$I_{5,4} = 01$ (Mnemo: MI) $\longrightarrow$ Abfrage des Mikrostatusregisters
$I_{5,4} = 10$ (Mnemo: MA) $\longrightarrow$ Abfrage des Maschinenstatusregisters

Tabelle 4.8: Bedingungskodes für den Vergleich zweier Zahlen A und B nach der Operation $A - B$.

Die Codierungen der fünf Instruktionsbit I_6 bis I_{10} bestimmen die Schiebeaktion, wobei I_{10} entscheidet, in welche Richtung geschoben wird. Da die Schieberichtung für beide Bausteine (Am2901 und Am2904) gleich sein muß, ist der Pin I_{10} des Am2904 fest mit dem Pin I_7 des Bausteins Am2901 verbunden (Bit 50 im Mikroinstruktionswort). Die 16 Rechts–Schiebeaktionen sind in Tabelle 4.10 und die 16 Links–Schiebeaktionen sind in Tabelle 4.11 aufgeführt. Die dritte Spalte der beiden Tabellen zeigt jeweils die Wirkung der Schiebeaktion auf die Schiebeeinheiten der ALU und auf das Statusbit M_C des Maschinenstatusregisters. Dieses Statusbit wird dabei unabhängig vom Zustand des Freigabesignals $\overline{CE_M}$ (Bit 29 im Mikroinstruktionsformat) gesetzt. In den weiteren vier Spalten dieser Tabellen ist jeweils aufgeführt, welches Signal an den jeweiligen Schiebeein– und –ausgängen anliegt.

Die zur Ausführung der arithmetischen Operationen benötigten **Übertragssignale** werden dem Rechenwerkbaustein Am2901 vom Wortrandlogikbaustein Am2904 über den Übertragsausgang C_0 bereitgestellt. Dieser Ausgang ist mit dem Übertragseingang C_n des niedrigstwertigen Rechenwerkbausteins Am2901 verbunden (siehe Abbildung 4.5). Die Übertragssteuerungslogik des Bausteins Am2904 erlaubt das Belegen des Übertragsausgangs C_0 aus maximal sieben Quellen.

Dabei bestimmen die Instruktionsbit I_{11} und I_{12}, welche Quelle den Wert für den C_0–Ausgang liefert. Wenn I_{12} mit 0 belegt ist, dann erscheint am C_0–Ausgang der Wert von I_{11}. Falls I_{12} mit 1 und I_{11} mit 0 belegt ist, dann liegt am C_0–Ausgang der Wert des C_X–Eingangs der Übertragssteuerung an, der in unserem Beispielrechner allerdings nicht beschaltet ist. Ist I_{12} und I_{11} jeweils mit 1 belegt, dann liegt der Wert des CARRY–Bits des Maschinen– oder Mikrostatusregisters (μ_C, M_C) oder dessen Komplement $(\overline{\mu_C}, \overline{M_C})$ an, was durch die Belegung der Instruktionsbit I_5, I_3, I_2 und I_1 bestimmt wird. In Tabelle 4.9 ist der Wert des Übertragsausgang C_0 in Abhängigkeit der Instruktionsbit I_{12}, I_{11} und I_5, I_3, I_2, I_1 dargestellt.

Zur Steuerung des Bausteins Am2904 werden insgesamt bis zu 13 Instruktionsbit (I_0 bis I_{12}) sowie neun Enablebit benötigt. Von den 9 Enablebit wurden nur 2, nämlich $\overline{CE_\mu}$ und $\overline{CE_M}$, beschrieben, da alle anderen Bits bei unserem Beispielrechner auf einen festen Pegel gelegt und somit dem Mikroprogrammierer nicht zugänglich sind.

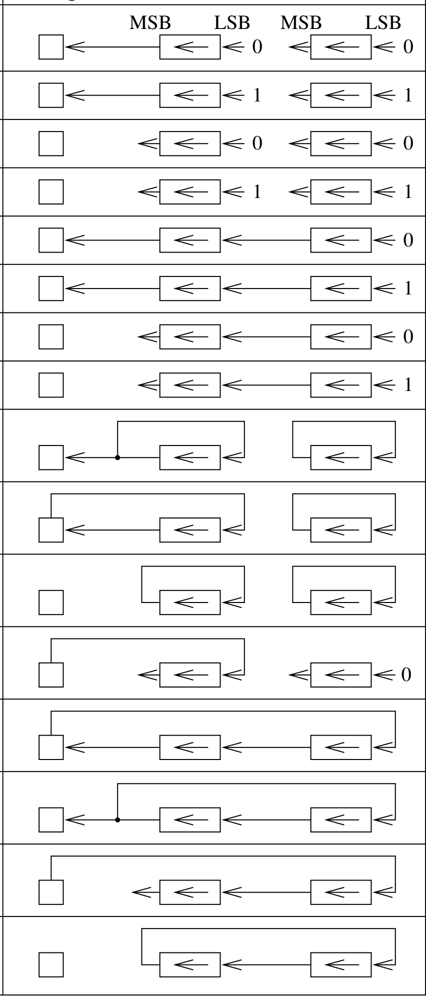
Mnemo	I_{12}	I_{11}	I_5	I_3	I_2	I_1	C_0
CI0	0	0	X	X	X	X	0
CI1	0	1	X	X	X	X	1
CIX	1	0	X	X	X	X	C_X
CIC	1	1	0	0	X	X	μ_C
CIC	1	1	0	1	1	X	μ_C
CIC	1	1	0	1	X	1	μ_C
CIC	1	1	0	1	0	0	$\overline{\mu_C}$
CIC	1	1	1	0	X	X	M_C
CIC	1	1	1	1	1	X	M_C
CIC	1	1	1	1	X	1	M_C
CIC	1	1	1	1	0	0	$\overline{M_C}$

X: Don't Care

Tabelle 4.9: Übertragssteuerung des Bausteins Am2904

I_{9876}	Mnemo	M_C RAM Q	SIO_0	SIO_n	QIO_0	QIO_n	M_C
		MSB LSB MSB LSB					
0	RSL	0 0	Z	0	Z	0	
1	RSH	1 1	Z	1	Z	1	
2	RSCONI	0 M_N	Z	0	Z	M_N	SIO_0
3	RSDH	1	Z	1	Z	SIO_0	
4	RSDC		Z	M_C	Z	SIO_0	
5	RSDN	M_N	Z	M_N	Z	SIO_0	
6	RSDL	0	Z	0	Z	SIO_0	
7	RSDCO	0	Z	0	Z	SIO_0	QIO_0
8	RSRCO		Z	SIO_0	Z	QIO_0	SIO_0
9	RSRCIO		Z	M_C	Z	QIO_0	SIO_0
A	RSR		Z	SIO_0	Z	QIO_0	
B	RSDIC	I_C	Z	I_C	Z	SIO_0	
C	RSDRCI		Z	M_C	Z	SIO_0	QIO_0
D	RSDRCO		Z	QIO_0	Z	SIO_0	QIO_0
E	RSDXOR	$I_N \oplus I_{OVR}$	Z	$I_N \oplus I_{OVR}$	Z	SIO_0	
F	RSDR		Z	QIO_0	Z	SIO_0	

Tabelle 4.10: Rechts–Schiebeaktionen des Bausteins Am2904 ($I_{10} = 0$)



I_{9876}	Mnemo	M_C RAM Q	SIO_0	SIO_n	QIO_0	QIO_n	M_C
0	LSLCO		0	Z	0	Z	SIO_n
1	LSHCO		1	Z	1	Z	SIO_n
2	LSL		0	Z	0	Z	
3	LSH		1	Z	1	Z	
4	LSDLCO		QIO_n	Z	0	Z	SIO_n
5	LSDHCO		QIO_n	Z	1	Z	SIO_n
6	LSDL		QIO_n	Z	0	Z	
7	LSDH		QIO_n	Z	1	Z	
8	LSCRO		SIO_n	Z	QIO_n	Z	SIO_n
9	LSCRIO		M_C	Z	QIO_n	Z	SIO_n
A	LSR		SIO_n	Z	QIO_n	Z	
B	LSLICI		M_C	Z	0	Z	
C	LSDCIO		QIO_n	Z	M_C	Z	SIO_n
D	LSDRCO		QIO_n	Z	SIO_n	Z	SIO_n
E	LSDCI		QIO_n	Z	M_C	Z	
F	LDSR		QIO_n	Z	SIO_n	Z	

Tabelle 4.11: Links–Schiebeaktionen des Bausteins Am2904 ($I_{10} = 1$)

4.1.2.3 Aufbau des Rechenwerks des Beispielrechners

In Abbildung 4.5 ist das Rechenwerk des Beispielrechners mit den Bausteinen Am2901, Am2902 und Am2904 dargestellt, wobei der Übersichtlichkeit wegen eine Reihe von Verbindungen nur angedeutet oder nicht eingezeichnet sind. Insbesondere fehlen alle Enablesignale.

Die vier Rechenwerksbausteine Am2901 bilden ein 16 Bit breites Rechenwerk. Die Y–Ausgänge der Rechenwerkbausteine sind über einen Multiplexer mit dem Adreß– und dem Datenbus verbunden, so daß die über die Y–Ausgänge des Am2901 ausgegebenen Daten auf den Adreß– oder Datenbus gelegt werden können. An den D–Eingängen des Am2901 liegen die über den Datenbus kommenden Daten an.

Die Übertragssteuerung zwischen den vier kaskadenartig aufgebauten Rechenwerkbausteinen übernimmt der Baustein Am2902 (Carry–Look–Ahead–Generator), der mithilfe eines Übertragsermittlungsschaltnetzes die für die Generierung des vorab ermittelten Übertrags (Carry–Look–Ahead) notwendigen Signale „carry propagate“ und „carry generate“ liefert.

Vom Baustein Am2904 führt eine Leitung vom Übertragsausgang C_0 zum Übertragseingang C_n des niedrigstwertigen Rechenwerkbausteins Am2901 und zum C_n–Eingang des Bausteins Am2902. Der Statusausgang OVR des höchstwertigen Am2901–Bausteins ist mit dem Statuseingang I_{OVR} verbunden. Von den Statusausgängen $F3$ und C_{n+4} des höchstwertigen Am2901–Bausteins führen Leitungen direkt zu den entsprechenden Statuseingängen I_N und I_C. Die Statusausgänge $F = 0$ aller vier Rechenwerkbausteine sind durch ein verdrahtetes ODER miteinander verknüpft. Die Nullanzeige liegt am Statuseingang I_Z des Am2904 an.

Der Ausgang CT des Am2904 ist direkt mit dem Testeingang $\overline{CC}$ des Sequencerbausteins Am2910 verbunden. Da dieser Testeingang invertiert ist („active LOW“), sind die zu überprüfenden Bedingungen so zu programmieren, daß am CT–Ausgang eine Null erscheint, wenn die Bedingung erfüllt ist. Dies ist in den Tabellen 4.6 und 4.8 bereits berücksichtigt.

Die Schiebeein– bzw. –ausgänge der Rechenwerkbausteine Am2901 sind untereinander und mit denen des Wortrandlogikbausteins Am2904 verbunden, so daß insgesamt 32 Schiebeaktionen durchgeführt werden können. Die Schieberichtung wird vom Instruktionsbit I_7 des Am2901 (Bit 50 des Mikroinstruktionsworts) bestimmt, da das Instruktionsbit I_{10} des Am2904 fest mit diesem verknüpft ist.

4.1.3 Der Hauptspeicher

Der Hauptspeicher des mikroprogrammierbaren Beispielrechners enthält 64K Worte zu je 16 Bit und ist nur wortweise adressierbar. Ein Zugriff auf den Speicher (lesend oder schreibend) benötigt immer zwei Takte (d.h. zwei Mikroinstruktionen). Im ersten Takt wird die Adresse auf den Adressbus ausgegeben. Die Adresse kann dabei aus dem Befehlszähler oder über den Y-MUX aus der ALU stammen. In diesem Takt ist über das $\overline{MWE}$-Bit im Mikroinstruktionswort (Bit 0) zu programmieren, ob ein Lese- oder ein Schreibzyklus angestoßen werden soll. Bei einem **Lesezyklus** liegen im zweiten Takt die Daten auf dem Datenbus an und können in das Instruktionsregister, den Befehlszähler oder über den K-MUX in die ALU geladen werden. Bei einem **Schreibzyklus** müssen im zweiten Takt die zu schreibenden Daten auf den Datenbus gelegt werden, wobei in diesem Takt auch $\overline{MWE}$ wieder auf 1 zurückzusetzen ist. Ein Pipelining, d.h. die teilweise zeitliche Überlappung von Speicherzugriffen ist in unserem Beispielrechners nicht möglich.

4.1.4 Beschreibung des Mikroinstruktionsformats

Abbildung 4.6 zeigt das Mikroinstruktionsformat des mikroprogrammierbaren Rechners. Nachfolgend sind die einzelnen Felder des Mikroinstruktionsformates beschrieben. Ebenso sind die mnemotechnischen Namen für die jeweils wichtigsten Codierungen (wenn sie nicht bereits in den Tabellen im Text genannt worden sind) angegeben.

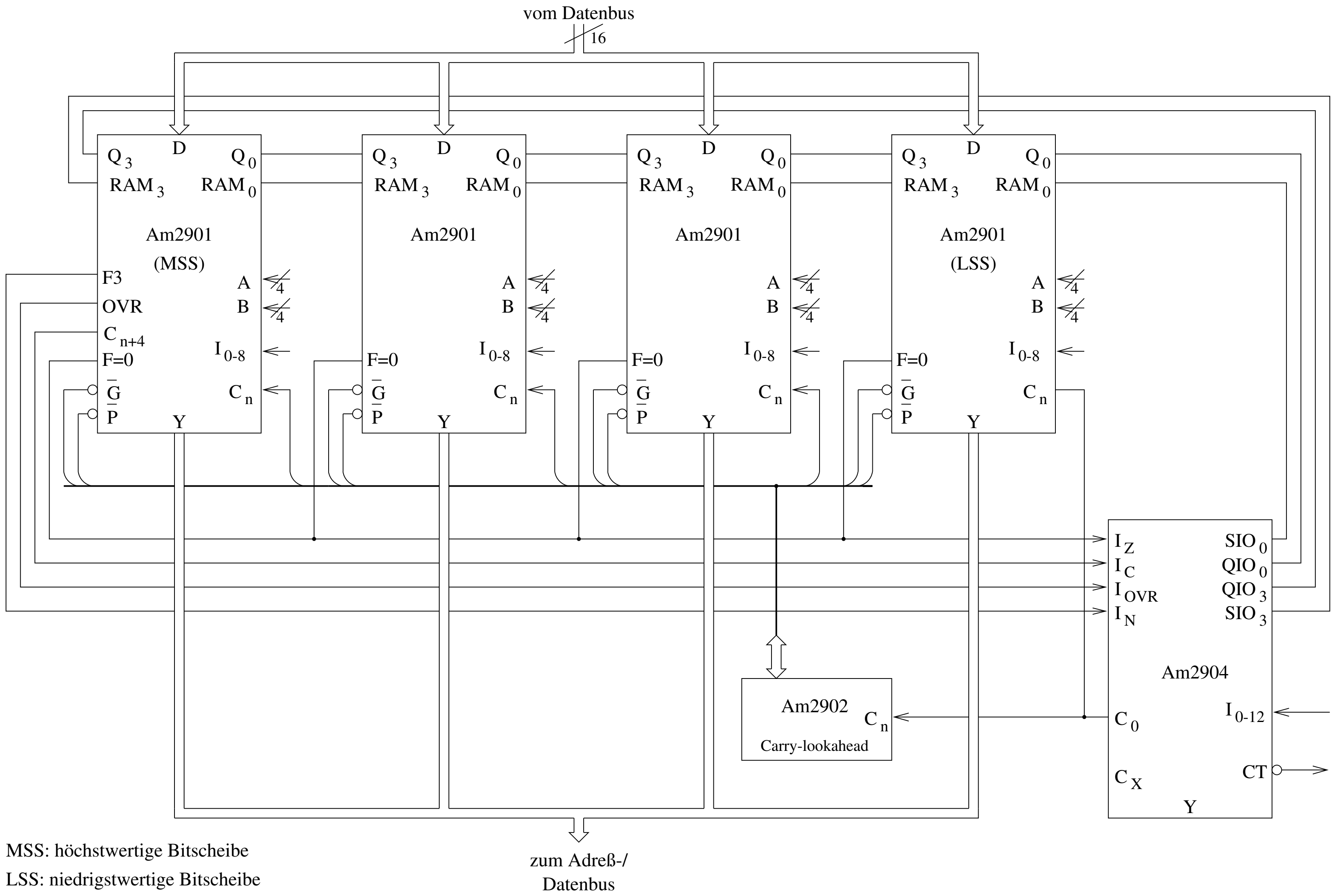


Abbildung 4.5: Aufbau des Rechenwerks des Beispielrechners

79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64	63	62	61	60	59	58
IE	I3	I2	I1	I0	KMUX	K15	K14	K13	K12	K11	K10	K9	K8	K7	K6	K5	K4	K3	K2	K1	K0
	Interrupt					Konstante															

57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37
I2	I1	I0	I5	I4	I3	I8	I7	I6	A3	A2	A1	A0	ASEL	B3	B2	B1	B0	BSEL	ABUS*	DBUS*
Src			Func			Dest			RA Addr					RB Addr					Y-MUX	
AM2901																				

36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18
I12	I11	I9	I8	I7	I6	CEMUE*	CEM*	I5	I4	I3	I2	I1	I0	CCEN*	I3	I2	I1	I0
CIN MUX		Schiebe-steuerung						Statusregister Test							AM2910-Befehle			
AM2904														AM2910				

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	BZ_LD*	BZ_ED*	BZ_INC*	BZ_EA*	IR_LD*	MEW*
Direktdaten																	
												BZ				IR	HS

Abbildung 4.6: Mikroinstruktionsformat

MI–Feld	Erläuterung	Vereinbarung mnemotechnischer Namen
MI_0	$\overline{MWE}$ *(Memory–Write–Enable)* In diesem Feld gibt der Mikroprogrammierer an, ob Daten aus dem Hauptspeicher gelesen ($\overline{MWE} = 1$) oder Daten in den Hauptspeicher geschrieben ($\overline{MWE} = 0$) werden sollen. Bei einem Hauptspeicherzugriff ist der gewünschte Wert im ersten Taktzyklus zu programmieren. Im zweiten Taktzyklus ist immer der Wert 1 (R) zu programmieren. Findet kein Hauptspeicherzugriff statt, dann muß eine 1 (R) programmiert werden.	0: W 1: R
MI_1	$\overline{IR\_LD}$ *(Instruktionsregister Laden)* Mit $\overline{IR\_LD} = 0$ wird das Instruktionsregister mit dem gerade auf dem Datenbus liegenden Datum (Befehl) geladen. $\overline{IR\_LD} = 1$ verhindert ein Ändern des Inhalts des Instruktionsregisters.	0: L 1: H
MI_2	$\overline{BZ\_EA}$ *(Befehlszähler Enable Adreßbus)* Mit $\overline{BZ\_EA} = 0$ wird der Inhalt des Befehlszählers auf den Adreßbus gegeben. $\overline{BZ\_EA} = 1$ verhindert, daß der Inhalt auf den Adreßbus ausgegeben wird.	0: E 1: H
MI_3	$\overline{BZ\_INC}$ *(Befehlszähler Inkrement)* Mit $\overline{BZ\_INC} = 0$ wird der Inhalt des Befehlszählers um 1 erhöht. $\overline{BZ\_INC} = 1$ verhindert ein Inkrementieren des Befehlszählers.	0: I 1: H
MI_4	$\overline{BZ\_ED}$ *(Befehlszähler Enable Datenbus)* Mit $\overline{BZ\_ED} = 0$ wird der Inhalt des Befehlszählers auf den Datenbus gegeben. $\overline{BZ\_ED} = 1$ verhindert, daß der Inhalt auf den Datenbus ausgegeben wird.	0: E 1: H
MI_5	$\overline{BZ\_LD}$ *Befehlszähler Laden)* Der Befehlszähler wird mit dem aktuellen Datum auf dem Datenbus geladen, wenn $\overline{BZ\_LD} = 0$ ist. $\overline{BZ\_LD} = 1$ verhindert, daß der Befehlszähler mit einem auf dem Datenbus liegenden Datum geladen wird.	0: L 1: H
MI_6..17	*BAR (Direktdatenfeld)* Über das Direktdatenfeld (BAR) kann der Mikroprogrammierer eine Absolutadresse für den Mikroprogrammspeicher in der aktuellen Mikroinstruktion angeben, um so einen Sprung, eventuell in Abhängigkeit einer Bedingung, zu programmieren. Ebenso kann im Direktdatenfeld ein Wert für das Zählerregister des Am2910 stehen. Ein im Direktdatenfeld stehendes Datum liegt am D–Eingang des Am2910 an.	
MI_18..21	$I_{0..3}$ *(Fortschaltbefehle des Am2910)* In diesem Feld werden die Fortschaltbefehle für den Sequencerbaustein Am2910 kodiert.	siehe Tabelle 4.1
MI_22	$\overline{CCEN}$ *(Condition Code Enable Am2910)* Wenn $\overline{CCEN} = 1$ ist, dann wird der am $\overline{CC}$–Eingang anliegende Wert des Am2910 ignoriert und der Am2910 arbeitet als ob am $\overline{CC}$–Eingang ein Signal mit Wert 0 anliegt (PASS–Bedingung). Falls eine Aktion in Abhängigkeit einer am $\overline{CC}$–Eingang anliegenden Testbedingung durchgeführt werden soll, dann muß $\overline{CCEN} = 0$ sein.	0: C 1: PS

Tabelle 4.12: Beschreibung des Mikroinstruktionsformats

MI–Feld	Erläuterung	Vereinbarung mnemotechnischer Namen
MI_23..28	$I_{0..5}$ *(Instruktionen des Am2904 – Statusregister / Test)* Aus diesem Feld werden die Statusregisteroperationen für das Mikrostatusregister und das Maschinenstatusregister kodiert, die in den Tabellen 4.6 und 4.7 aufgelistet sind. Die Statusregister werden jedoch nur verändert, wenn das entsprechende Enablesignal $\overline{CE_\mu}$ (MI_30) für das Mikrostatusregister bzw. $\overline{CE_M}$ (MI_29) für das Maschinenstatusregister den Wert 0 hat. Über dieses Feld wird auch der Bedingungsmultiplexer des Bausteins gesteuert. Im Feld MI_23..26 (Instruktionsbit $I_{0..3}$) wird der gewünschte Test programmiert und im Feld MI_27..28 (Instruktionsbit $I_{4..5}$) wird angegeben, in welchem Teilwerk des Bausteins die Bedingung geprüft werden soll. Die entsprechenden Instruktionskodes mit den menmotechnischen Namen sind der Tabelle 4.8 zu entnehmen, wobei den Namen für die Bedingungskodes MI oder MA (für Mikro- bzw. Maschinenstatusregister) voranzustellen ist.	Bedingungscodes: siehe Tabelle 4.8
MI_29	$\overline{CE_M}$ *(Enablebit für das MSR)* Der Inhalt des Maschinenstatusregisters des Am2904 kann nur dann verändert werden, wenn das Enablebit $\overline{CE_M} = 0$ ist.	0: L 1: H
MI_30	$\overline{CE_\mu}$ *(Enablebit für das μSR)* Der Inhalt des Mikrostatusregisters des Am2904 kann nur dann verändert werden, wenn das Enablebit $\overline{CE_\mu} = 0$ ist.	0: L 1: H
MI_31..34	$I_{6..9}$ *(Instruktionsbit des Am2904 – Schiebesteuerung)* Die Schiebesteuerung des Am2904 erlaubt die in Tabelle 4.10 und Tabelle 4.11 aufgeführten Schiebeaktionen. In Verbindung mit der ALU–Zielsteuerung des Am2901 im Feld MI_49..51 wird eine der 32 möglichen Schiebeoperationen programmiert, damit die gewünschte Schiebeaktion im Rechenwerk durchgeführt werden kann.	siehe Tabelle 4.10 und Tabelle 4.11
MI_35..36	$I_{11..12}$ *(Instruktionsbit des Am2904 – Übertragssteuerung)* Die Übertragssteuerung erlaubt das Belegen des Übertragsausgangs C_0 aus maximal sieben Quellen (siehe Abschnitt 4.1.2.2).	siehe Tabelle 4.9
MI_37	$\overline{DBUS}$ *(Datenbus Select)* Die Daten, die über den Y–Ausgang des Am2901 ausgegeben werden, kommen auf den Datenbus.	0: DB 1: H
MI_38	$\overline{ABUS}$ *(Adreßbus Select)* Die Daten, die über den Y–Ausgang des Am2901 ausgegeben werden, kommen auf den Adreßbus.	0: AB 1: H

Tabelle 4.13: Beschreibung des Mikroinstruktionsformats

MI–Feld	Erläuterung	Vereinbarung mnemotechnischer Namen
MI_39	*BSEL (RB ADDR Select)* Bei $BSEL = 1$ wird die Adresse für das Register im Register–B–Adreßfeld des Mikroinstruktionsregisters angegeben. Bei $BSEL = 0$ wird die Adresse für das Register im Register–B–Adreßfeld des Instruktionsregisters spezifiziert.	0: IR 1:MR
MI_40..43	*RB ADDR (Register B Adresse)* Der Inhalt des in diesem Feld adressierten Registers der internen Registerdatei des Am2901 wird über den B–Ausgang ausgegeben. Das Zielregister für zu schreibende Daten wird ebenfalls in diesem Feld adressiert.	0_{hex}: r0 1_{hex}: r1 $\vdots$ F_{hex}: r15
MI_44	*ASEL (RA ADDR Select)* Bei $ASEL = 1$ wird die Adresse für das Register im Register–A–Adreßfeld des Mikroinstruktionsregisters angegegben. Bei $BSEL = 0$ wird die Adresse für das Register im Register–A–Adreßfeld des Instruktionsregisters spezifiziert.	0: IR 1:MR
MI_45..48	*RA ADDR (Register A Adresse)* Der Inhalt des in diesem Feld adressierten Registers der internen Registerdatei des Am2901 wird über den A–Ausgang ausgegeben.	0_{hex}: r0 1_{hex}: r1 $\vdots$ F_{hex}: r15
MI_49..51	$I_{6..8}$ *(Instruktionsbit des Am2901 – ALU Zielsteuerung)* Die Steuerung, wohin die Ergebnisse der ALU des Am2901 geschrieben und welche Schiebeaktionen ausgeführt werden, erfolgt über dieses Feld. Die acht möglichen Kombinationen sind in Tabelle 4.4 dargestellt. Bei Schiebeaktionen ist zusätzlich im Feld MI_31..34 die gewünschte Schiebeaktion anzugeben.	siehe Tabelle 4.4
MI_52..54	$I_{3..5}$ *(Instruktionsbit des Am2901 – ALU Funktionen)* Die ALU kann die in Tabelle 4.3 angegebenen drei arithmetischen und fünf logischen Operationen ausführen, wobei eine der acht möglichen ALU–Funktionen in diesem Feld zu kodieren ist.	siehe Tabelle 4.3
MI_55..57	$I_{0..2}$ *(Instruktionsbit des Am2901 – ALU Quelloperanden)* Die Operanden für die ALU des Am2901 können seiner internen Registerdatei, seinem Q–Register und dem unidirektionalen D–Eingang entstammen. In diesem Feld werden die Operanden für den ALU–R– und den ALU–S–Eingang bestimmt. Die acht möglichen Kombinationen sind in Tabelle 4.2 angegeben.	siehe Tabelle 4.2
MI_58..73	$K_{0..15}$ Konstantenfeld	
MI_74	*KMUX* Auswahl der Quelle für den D-Eingang des Rechenwerks. Die Daten können entweder vom Konstantenfeld des Mikroinstruktionsregisters kommen oder vom Datenbus.	0: K 1: D
MI_75..78	$I_{0..3}$ *(Interruptsteuerung)* Die Interruptsteuerung ist der Einfachheit wegen nicht beschrieben.	
MI_79	*IE (Interrupt Enable)* Zulassen von Unterbrechungsanforderungen	0: IE 1: Dis

Tabelle 4.14: Beschreibung des Mikroinstruktionsformats

4.2 Dokumentation zum Simulator der mikroprogrammierten Maschine

Für die in Kapitel 4.1 beschriebene mikroprogrammierte Maschine ist ein Simulator verfügbar, der unter den Betriebssystemen Windows 95, Windows 98 und Windows NT auf PCs ablauffähig ist. Er erlaubt die Eingabe von Mikroinstruktionen über ein Dialogfenster, die Anzeige und das Verändern des Hauptspeichers und der ALU-Register, sowie die simulierte Ausführung von Mikroprogrammen. Der Simulator gestattet zudem den „Ausdruck“ der Mikroprogramme im HTML-Format, sowie die Erstellung von Ablaufprotokollen, ebenfalls im HTML-Format.

4.2.1 Bedienung des Simulators

4.2.1.1 Start

Nach dem Start erscheint das Hauptfenster des Simulators wie in Abb. 4.7 gezeigt ist. Falls das Fenster nicht sofort die volle Bildschirmgröße haben sollte, muß es maximiert werden, um alle Bedienelemente sichtbar zu machen.

In der oberen Hälfte werden die Mikroinstruktionen angezeigt, links unten der Hauptspeicherinhalt, rechts unten die Register und Statusbits. Über die Scrollbars kann der angezeigte Bereich verändert werden. Zusätzlich sind zwei Schaltflächen vorhanden, mit denen im Mikroprogramm in Schritten von 256 Instruktionen vor- und zurückgeblättert werden kann.

4.2.1.2 Eingabe von Mikroprogrammen

Durch Anklicken einer Mikroprogrammzeile mit der Maus wird ein Dialogfenster geöffnet, mit dessen Hilfe die einzelnen Felder der Mikroinstruktion besetzt werden können (siehe Abb. 4.8). Durch Drücken auf den „Send“-Knopf werden die neuen Werte in den Mikroprogrammspeicher übernommen.

Der „Copy“-Knopf im Hauptfenster erlaubt das Kopieren von Mikroinstruktionen an andere Adressen. In einem Dialogfenster werden dazu der Quell- und Zieladreßbereich abgefragt.

Vor der Eingabe eines eignenen Mikroprogramms sollte jedoch zunächst das Mikroprogramm „IFETCH“ geladen werden, das den nächsten Maschinenbefehl in das Instruktionsregister lädt, den Befehlszähler inkrementiert, und das zu dem Maschinenbefehl gehörige Mikroprogramm anspringt (siehe auch „Speichern und Laden“).

4.2.1.3 Programmieren des Mapping-PROMs

Ein Programmieren des Mapping-PROMs, das die Zuordnung zwischen den Op-Codes der Maschinenbefehle und den Anfangsadressen der sie realisierenden Mikroprogramme enthält, ist im Simulator nicht möglich. Stattdessen ist das Mapping-PROM nach der folgenden Regel fest programmiert:

> Der Op-Code i wird auf die Mikroprogrammadresse $16 * i$ abgebildet.

Das Mikroprogramm, das z.B. den Maschinenbefehl mit dem Op-Code 47 (hex.) implementiert, muß demnach also an Adresse 470 (hex.) im Mikroprogrammspeicher beginnen.

4.2.1.4 Anzeige und Ändern von Hauptspeicher- und Registerinhalten

Der Inhalt des Hauptspeichers wird links unten im Simulatorfenster als Tabelle angezeigt. In der ersten Spalte steht die Adresse der Speicherzelle, gefolgt von ihrem aktuellen Inhalt (beides in hexadezimaler

Id: MI-Simulator.tex,v 3.4 1999/05/17 14:23:28 wismuell Exp

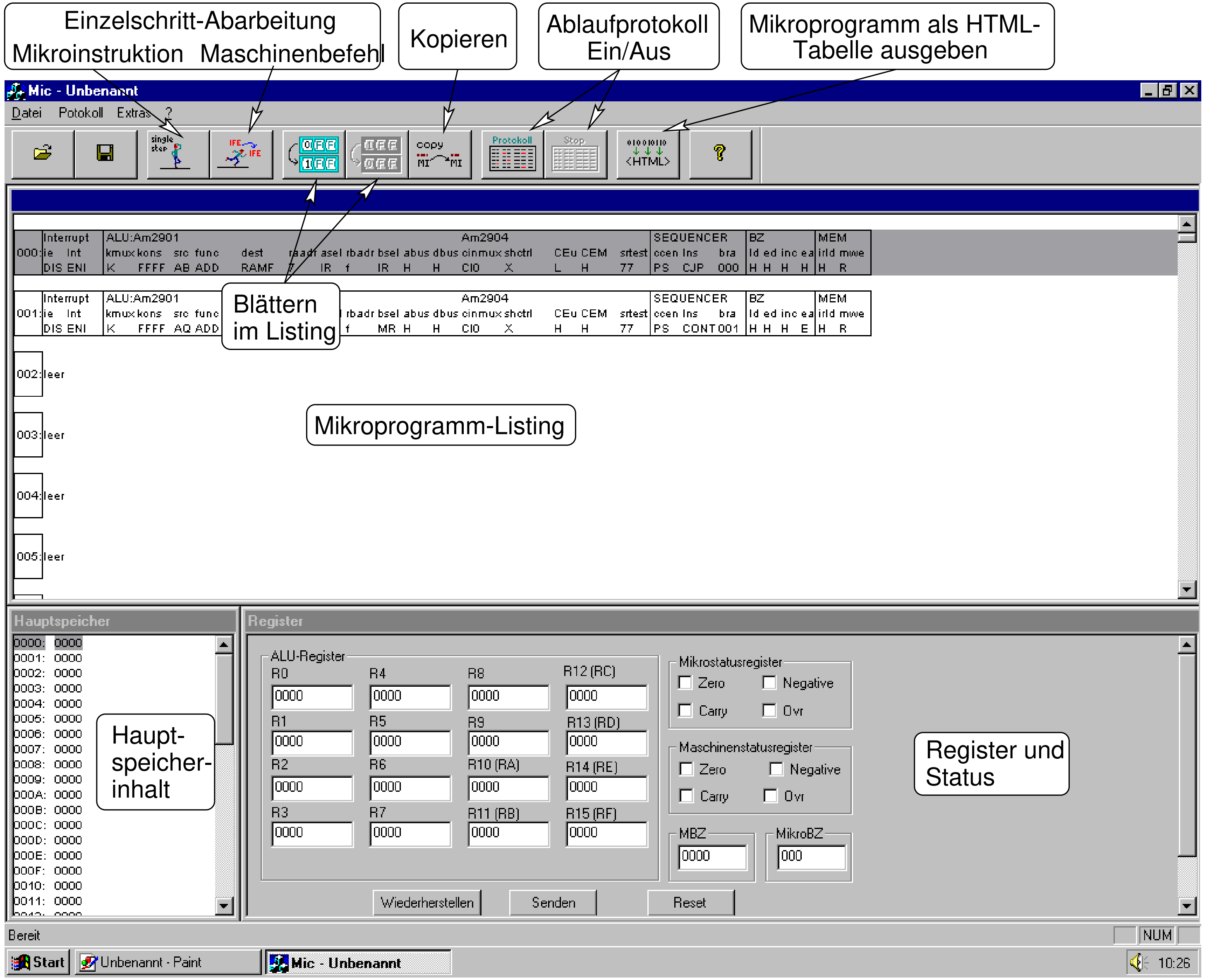


Abbildung 4.7: Hauptfenster des Mikroprogrammier-Simulators

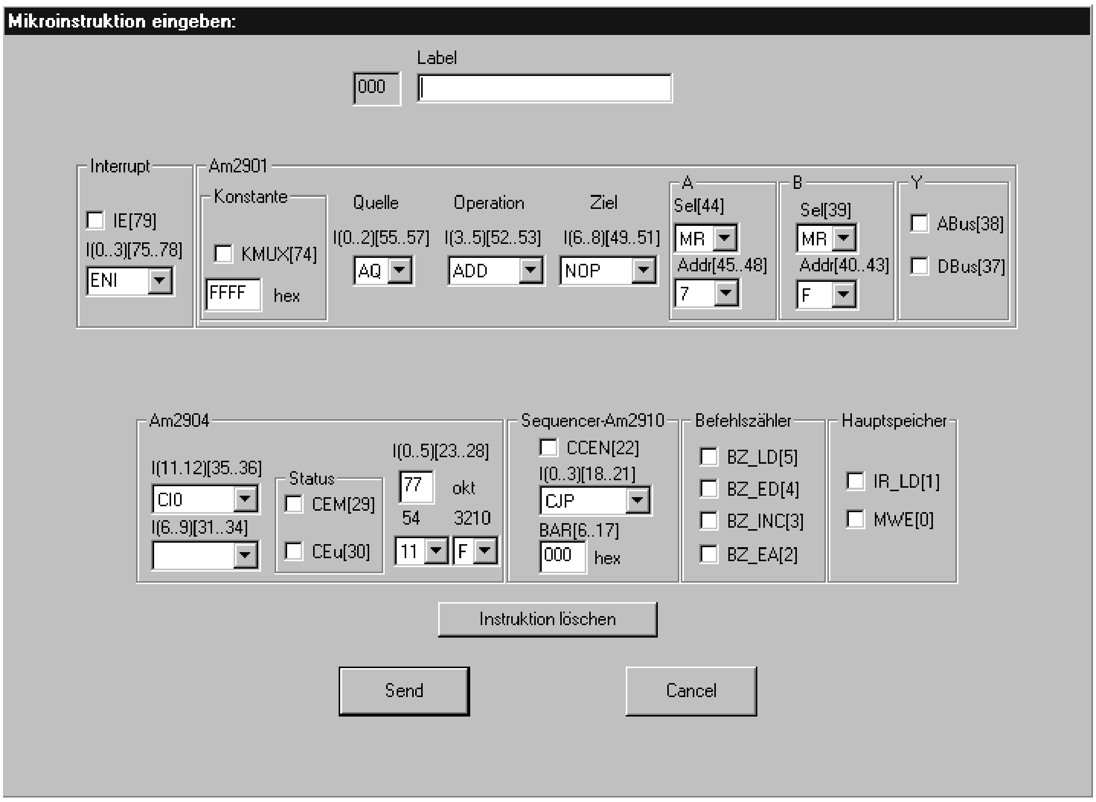


Abbildung 4.8: Dialog zur Definition von Mikroinstruktionen

Schreibweise). Die nachsten zwei Spalten, die auch leer sein können, dekodieren den Inhalt der Speicherzelle auf verschiedene Weisen. Für Spalte 3 wird der Inhalt als zwei ASCII-codierte Zeichen aufgefaßt und die beiden entsprechenden Zeichen dargestellt. In Spalte 4 wird der Inhalt als erstes Wort eines Maschinenbefehls, das den Op-Code enthält, aufgefaßt. Angezeigt wird in dieser Spalte dann das Label-Feld der ersten Mikroinstruktion, die zu diesem Op-Code gehört. Enthält das Speicherwort jedoch z.B. einen direkten Operanden oder ein Datum, so ist der Inhalt der Spalte 4 natürlich nicht sinnvoll.

Zum Ändern einer Hauptspeicherzelle kann diese mit der Maus angeklickt werden. Es erscheint ein Dialogfenster, in dem der neue Wert eingegeben werden kann. Nach dem Quittieren wird dieser in den Hauptspeicher übernommen.

Die Inhalte aller 16 Register der mikroprogrammierten Maschine, sowie der Mikro- und Maschinenstatusregister und der Mikro- und Maschinenbefehlszähler werden in dem rechten unteren Feld des Simulatorfensters angezeigt.

Register- und Befehlszählerinhalte können nach dem Anklicken direkt an Ort und Stelle verändert werden. Nach einer Änderung sollte der Knopf „Senden“ gedrückt werden, um die Änderungen in die Simulation zu übernehmen. Ein Verändern der Statusbits ist derzeit nicht möglich.

4.2.1.5 Speichern und Laden

Zum Speichern eines Mikroprogramms und des Hauptspeicherinhalts sind die Menüpunkte „Speichern“ bzw. „Speichern unter“ im Datei-Menü auszuwählen. Der Simulator speichert dabei nur die relevanten Zellen im Mikroprogrammspeicher ab, also diejenigen, die mit gültigen Mikroinstruktionen besetzt sind. Alle anderen Speicherplätze werden auch im Hauptfenster als leer angezeigt. Vom Hauptspeicher wird nur der Bereich gesichert, der auch im Hauptspeicherfenster (links unten in Abb. 4.7) angeschaut und verändert werden kann. Die Größe dieses Bereichs kann über einen Menüpunkt im „Extras“-Menü eingestellt werden.

Zum Laden von Mikroprogrammen und Hauptspeicherinhalten dient der Punkte „Laden“ im Datei-Menü. Beim Laden werden die in der Datei gespeicherten Instruktionen oder Werte in den Mikroprogrammspeicher bzw. Hauptspeicher geladen. Die übrigen Speicherzellen bleiben unverändert. Es ist also z.B. möglich, Mikroprogramme aus verschiedenen Dateien gleichzeitig zu laden, vorausgesetzt, daß sie unterschiedliche Adreßbereiche belegen.

4.2.1.6 Ausführen von Programmen

Zum Ausführen bzw. Testen der Mikroprogramme sollte zweckmäßigerweise das Programm „IFETCH“ geladen sein, und im Hauptspeicher sollte sich ein kurzes Maschinenprogramm befinden. Evtl. sollten vor der Ausführung auch noch die Maschinenregister passend initialisiert werden.

Mit dem Knopf „Single Step“ (siehe Abb. 4.7) kann dann die Abarbeitung des Programms auf Mikroinstruktionsebe Instruktion für Instruktion verfolgt werden. Mit jedem Drücken wird dabei die nächste Mikroinstruktion ausgeführt. Diese ist durch graue Hinterlegung im oberen Teil des Hauptfensters kenntlich gemacht. Gleichzeitig werden im unteren Teil die jeweils aktuellen Register- und Speicherinhalte angezeigt.

Der Knopf „IFE $\rightarrow$ IFE“ führt dagegen das Mikroprogramm solange ohne Unterbrechung aus, bis das Mikroprogramm „IFETCH“ an Mikroprogrammadresse 0 wieder erreicht wird. Es wird also eine Einzelschrittverarbeitung auf Maschinenprogramm-Ebene durchgeführt.

Der Simulator bietet über zwei Schaltflächen bzw. Menüpunkte die Möglichkeit, ein Ablaufprotokoll in HTML zu erstellen. Ist die Protokollierung eingeschaltet, wird nach jedem Ausführungsschritt eine Zeile an das Protokoll angefügt, die die Registerinhalte und Statusbits nach diesem Schritt enthält. Werte, die sich geändert haben, werden dabei rot dargestellt.

Mit Hilfe des „Reset“-Knopfes können der Mikroprogramm- und der Maschinenprogrammzähler wieder auf Null gesetzt werden.

4.2.1.7 Drucken von Programmen

Die im Simulator geladenen Mikroprogramme können durch Anklicken des entsprechenden Knopfes (siehe Abb. 4.7) in Form von HTML-Tabellen „ausgedruckt“ werden. Die Mikroprogrammtabelle wird i.a. in mehrere HTML-Dateien aufgeteilt; an den spezifizierten Dateinamen werden dabei fortlaufende Nummern angehängt. In die Tabelle werden dabei nur die besetzten Einträge im Mikroprogrammspeicher aufgenommen, „leere“ Einträge erscheinen nicht.

4.2.2 Einschränkungen des Simulators

Ab der Version 5.1 besitzt der Simulator gegenüber der Beschreibung in Kapitel 4.1 keine Einschränkungen mehr, d.h. er kann die Maschine vollständig simulieren.

Eine Einschränkung ist jedoch anzumerken, die im Prinzip für jede Art von Simulatoren gilt: Viele Aspekte des Verhaltens der mikroprogrammierten Maschine sind in der Spezifikation in Kapitel 4.1 nicht definiert. Insbesondere trifft das auf das Verhalten bei fehlerhafter Programmierung zu (z.B. wenn bei einem Speicherzugriff während des Zugriffs die Adresse auf dem Adreßbus geändert wird, etwa indem der Befehlszähler auf den Adreßbus ausgegeben und gleichzeitig inkrementiert wird). Wenn das Verhalten in einer bestimmten Situation nicht festgelegt ist, kann (und darf!) der Simulator hier anders reagieren als eine reale Implementierung der Maschine (und diese wiederum kann und darf sich anders verhalten als eine andere Implementierung derselben Maschine). Das bedeutet, das man eine Reaktion der Maschine, die nicht „offiziell“ dokumentiert ist, nicht einfach am Simulator austesten und dann davon ausgehen kann, daß die reale Maschine dieselbe Reaktion zeigen wird.

5 Bereich III: Rechnergestützter Schaltungsentwurf

Markus Leberecht

5.1 VHDL-Kurzanleitung

5.1.1 Vorbemerkung

VHDL ist eine Abkürzung von Abkürzungen und steht für *VHSIC Hardware Description Language*, wobei VHSIC die Kurzform von *Very High Speed Integrated Circuit* ist. Man erkennt hieran schon, daß diese Sprache in der Hauptsache zur Beschreibung von Schaltungen dient, die im Bereich des Entwurfes integrierter Schaltkreise verwendet werden. Ihre vornehmliche Nutzung erfolgt in der

- Dokumentation von Schaltungen, in der
- Simulation von Schaltungen und als Grundlage zur
- automatischen Synthese von Schaltungen.

Unter automatischer Synthese versteht man dabei in der Regel den Vorgang der maschinellen Realisation einer digitalen elektrischen Schaltung aus einer Beschreibung, die z. B. in VHDL vorliegt.

Diese Kurzbeschreibung präsentiert eine Untermenge von VHDL, der auf die Bedürfnisse der Veranstaltung *Technische Grundlagen der Informatik* in Übung, Klausur und Praktikum zugeschnitten ist. Dennoch wird annähernd der volle Sprachumfang umrissen, Zugeständnisse wurden lediglich bei der Vielfältigkeit verschiedener Konstrukte gemacht, so daß VHDL in kurzer Zeit leichter beherrschbar wird.

In Abschnitt 5.2 sind jeweils Beipiele aufgeführt, die die Benutzung der vorgestellten Sprachmittel verdeutlichen sollen.

5.1.1.1 Nomenklatur

`GROSSGESCHRIEBENE` Wörter Kennzeichnen Schlüsselwörter der Sprache VHDL.

`<bezeichner>` in spitzen Klammern deuten auf benutzerdefinierte Namen im VHDL-Quelltext hin.

`bezeichner` ohne spitze Klammern stehen für einzusetzende VHDL-Ausdrücke.

VHDL-Ausdrücke wie `[irgend_was]` in eckigen Klammern sind optional.

VHDL-Ausdrücke wie `{irgend_was}` stellen beliebige Wiederholungen des gleichen Ausdrucks dar.

VHDL-Ausdrücke wie `irgend_was | irgend_was_anderes` stellen eine Auswahlmöglichkeit zwischen den beiden Alternativen dar.

Ein Kommentar in VHDL ist stets einzeilig und beginnt mit zwei Bindestrichen `--`.

5.1.2 Bausteine, Module und Bibliotheken

Ein „Baustein“ in einer Schaltung kann als Black Box mit einem Interface zu seiner Umwelt verstanden werden. VHDL bietet zur Beschreibung dieses Sachverhalts die beiden Sprachmittel `entity` und `architecture` an.

Deklaration einer Entity

```
ENTITY <entity_name> IS
      PORT (
          [signal] <identifier> {,<identifier>}:[mode] <type_mark>
          {;[signal] <identifier> {,<identifier>}:[mode] <type_mark>}
      );
END [<entity_name>];
```

Die Entity stellt das Interface eines Bausteins oder einer Schaltung nach außen dar.

Deklaration einer Architecture

```
ARCHITECTURE <architecture_name> OF <entity_name> IS
      type_declaration
      | signal_declaration
      | constant_declaration
      | component_declaration
BEGIN
      {process_statement
      | concurrent_signal_assignment_statement
      | component_instatiation_statement}
END [<architecture_name>];
```

Eine Architecture stellt das zu einer Entity gehörige „Innenleben“ einer Schaltung dar. Aus diesem Grunde existiert zu einer Entity immer mindestens eine Architecture.

Innerhalb des Architecture-Kopfes können Datentypen, Signale, Konstanten und Komponenten, also Kopien anderer Bausteine, deklariert werden.

Eine Architecture selber besteht aus Prozessen, nebenläufigen Signalzuweisungen und instantiierten („eingebauten“) Komponenten.

Deklaration einer Komponente

```
COMPONENT <component_name> IS
      PORT (
          [signal] <identifier> {,<identifier>}:[mode] <type_mark>
          {;[signal] <identifier> {,<identifier>}:[mode] <type_mark>}
      );
END COMPONENT;
```

Soll ein Baustein als Komponente innerhalb einer übergeordneten Architecture verwendet werden, so muß er zusätzlich als Component in der angegebenen Form deklariert werden.

5.1.3 Datenobjekte, -typen und Modi

5.1.3.1 Datenobjekte

VHDL beinhaltet drei unterschiedliche Arten von Datenobjekten, die wiederum unterschiedlichen Typs sein können und einen bestimmten Modus besitzen.

Signals (Signale) sind die in der Praxis am häufigsten vorkommenden Datenobjekte in VHDL und verwirklichen den Datentransport und die Datenspeicherung innerhalb von Architectures und über deren Grenzen hinaus. Eine Zuweisung an ein Signal sowie das Benutzen des Wertes eines Signales sind in das Zeitmodell von VHDL eingebunden: Alle nebenläufigen Zuweisungen (siehe dort) finden gleichzeitig statt, alle Signalzuweisungen innerhalb der sequentiellen Anweisungen eines Prozesses erfolgen erst durch das `END PROCESS`-Statement.

Constants (Konstanten) stellen statische Werte dar, ganz entsprechend herkömmlichen Programmiersprachen, und werden zur Verbesserung der Lesbarkeit des Quellcodes wegen verwendet.

VHDL-Beschreibungen können Anteile enthalten, die ihr Verhalten auf eine sequentielle Weise beschreiben (Prozesse und - nicht in dieser Kurzanleitung - Unterprogramme). Sollen innerhalb solcher Blöcke lokale Informationen zwischenzeitlich gespeichert werden, so kann man sich dazu der **Variables**(Variablen) bedienen. Variablen können aber im Gegensatz zu Signalen keine Information außerhalb ihrer Blöcke transportieren. Variablen werden auch nicht wie Signale erst zu gewissen Zeitpunkten zugewiesen, sondern erhalten ihren Wert innerhalb der sequentiellen Reihenfolge der Abarbeitung von Prozessen. Ihr Wert erlischt bei der `END PROCESS`-Anweisung.

5.1.3.2 Datentypen

VHDL bietet zahlreiche Möglichkeiten zur flexiblen Benutzung und zur Definition neuer Datentypen. Der Inhalt dieser Kurzbeschreibung soll aber lediglich eine Reihe von gebräuchlichen Typen enthalten.

Typ Standard-Logik

$$\texttt{STD\_LOGIC} \in \{'\texttt{X}', '\texttt{0}', '\texttt{1}', '\texttt{Z}'\}$$

Im einzelnen stehen die Werte[1] für folgende Bedeutungen:

`'X'` steht für einen uninitialisierten bzw. unbekannten Datenwert.

`'0'` stellt die logische '0' dar.

`'1'` stellt die logische '1' dar.

`'Z'` heißt *hochohmig* (oder auch *Tristate*) und bezeichnet einen Wert, der von den anderen drei stets überschrieben werden kann.

Typ Standard-Logik-Vektor

`STD_LOGIC_VECTOR (<n1> TO | DOWNTO <n2>)`

[1] Der eigentliche Standard-Logik-Typ in VHDL hat sogar neun logische Werte und ist ein kein eingebauter, sondern nur einer im Rahmen einer Standardbibliothek eingebauter Datentyp. Für die Zwecke von TGI jedoch soll die obige Form ausreichen

Der `std_logic_vector` mit den Indizes von $n1$ nach $n2$ (Schlüsselwort `TO` nur bei $n1 < n2$, sonst `DOWNTO`!) ist eine Zusammenfassung von einzelnen `std_logic`-Elementen und kann so beispielsweise für Busse verwendet werden.

Typ Integer wird in der Regel im Rahmen von Konstanten verwendet und stellt den üblichen Ganzzahltyp dar.

Typ Boolean beinhaltet die beiden Werte `TRUE` und `FALSE` und wird zur Darstellung von Wahrheitswerten genutzt.

Der Aufzählungstyp ist benutzerdefiniert und kann beispielsweise zur einfachen Definition von Zuständen eines Automaten verwendet werden.

```
TYPE <enumeration_type_name> IS
    ( <value_name> {, <value_name>} );
```

5.1.3.3 Modi

Signale, die zum Datenaustausch zwischen einzelnen Blöcken einer VDHL-Beschreibung dienen, müssen mit einer Datenrichtung (dem Modus) ausgestattet werden, welche in der Signaldefinition vor den Typen geschrieben wird. Modi sind:

- `IN` wird ausschließlich als Eingangssignal verwendet. An diese Signale ist innerhalb des Blockes, für den sie definiert wurden, keine Zuweisung möglich.
- `OUT` wird ausschließlich als Ausgangssignal verwendet. Von diesen Signalen kann innerhalb des Blockes, für den sie definiert wurden, nicht gelesen werden.
- `BUFFER` wird als Ausgangssignal verwendet, dessen Wert auch innerhalb des Blockes, für den es definiert wurde, benötigt wird.
- `INOUT` wird verwendet, wenn das Signal bidirektional benutzt wird: Zuweisungen sowohl als aus Auslesen seiner Werte können sowohl innerhalb als auch außerhalb des definierenden Blockes erfolgen.

5.1.4 Nebenläufige Beschreibung des Aufbaus oder des Verhaltens von Architekturbeschreibungen (Architectures)

In Hardware finden in der Regel viele Vorgänge in Abhängigkeit voneinander, dennoch aber zur gleichen Zeit statt. Dieses Verhalten, welches man als *nebenläufig* bezeichnet, kann auch in VHDL beschrieben werden.

5.1.4.1 Strukturelle Beschreibung

Eine gängige Variante der Beschreibung einer Schaltung ist die Darstellung ihrer Struktur. Dabei wird ein übergeordneter Block (z. B. eine Architecture) aus untergeordneten kleineren Modulen zusammengesetzt.

Eine Form dieser Submodule sind die Komponenten, die durch ihre Instantiierung in einer Architecture vervielfältigt werden (**Instantiierung von Komponenten**):

```
<instantiation_label>:
      <component_name> PORT MAP (
            <signal_name> | OPEN {, <signal_name> | OPEN}
            );
```

Hierbei dient `<instantiation_label>` zur Unterscheidung mehrfach instantiierter Komponenten, während die Port-Map zur Zuordnung von übergeordneten Signalen zu den Ein- und Ausgängen der Komponente benutzt wird.

5.1.4.2 Verhaltensbeschreibung

Neben der Beschreibung der Struktur ihres Aufbaus kann eine Schaltungsfunktion auch durch ihr Verhalten beschrieben werden. VHDL eröffnet hierfür mehrere Möglichkeiten.

Einfache Zuweisungen ermöglichen es, Signalen Werte zuzuordnen:

```
<signal_name> <= value { operator value };
```

wobei `value` aus Signal, Variable oder Konstante besteht und die üblichen Operatoren verwendet werden können.

Bedingte Zuweisungen:

```
<signal_name> <= value_true WHEN condition ELSE value_false;
```

An das Signal wird nur dann `value_true` zugewiesen, when die Bedingung `condition` (siehe Vergleichsoperatoren) `TRUE` oder `'1'` ergibt. Ansonsten erhält `<signal_value>` den Wert `value_false`.

Der **Prozeß**in VHDL ist ein Block, dessen Verhalten über einen sequentiellen Algorithmus beschrieben wird, bei dessen Ausführung allerdings stets bedacht werden muß, daß die simulierte Zeit der ganzen VHDL-Beschreibung nicht fortschreitet.

```
[<process_label>:]
PROCESS (sensitivity_list)
      {type_declaration
      | constant_declaration
      | variable_declaration}
```

```
BEGIN
      {wait_statement
      | signal_assigment_statement
      | variable_assignment_statement
      | if_statement
      | case_statement
      | loop_statement}
END PROCESS [<process_label>];
```

Jeder Prozeß besitzt eine sogenannte Sensitivity List, eine Aufzählung von Signalen, bei deren Änderung der zwischen `BEGIN` und `END` eingeklammerte Block ausgeführt wird. Im Prozeßkopf können wieder wie üblich Typen, Konstanten und Variablen vereinbart werden.

Innerhalb des Prozesses stehen dem VHDL-Autoren eine Reihe von Sprachmitteln zur Verfügung, die z. T. auch in anderen herkömmlichen Programmiersprachen existieren.

Signalwerte, die in einem Prozeß zugewiesen werden, werden anderen Teilen einer VHDL-Beschreibung erst nach Beedingung dieses Prozesses bekanntgegeben.

Wartekommando

```
WAIT UNTIL condition;
```

Hiermit wird der Ablauf der Kommandos im Prozeß angehalten, bis die angegebene Bedingung wahr ist.

If-Kommando

```
IF condition THEN
      sequence_of_statements
      {ELSIF condition THEN sequence_of_statements}
      [ELSE sequence_of_statements]
END IF;
```

Mit dem If-Kommando werden wie üblich Entscheidungen innerhalb eines Prozesses getroffen. Vorsicht ist geboten bei der Benutzung von If ohne einen else-Zweig, sofern die Beschreibung zur Synthese eingesetzt wird: Eine fortgelassene Alternative impliziert die Benutzung von Speicherelementen bei einer Zuweisung, da nur so der vorherige Wert erhalten werden kann.

Case-When-Kommando

```
CASE expression IS
      {WHEN constant_value | OTHERS => sequence_of_statements}
END CASE;
```

Das Case-When-Kommando dient zur Auswahl aus mehreren Möglichkeiten. Zur Abdeckung aller Alternativen dient das Schlüsselwort `OTHERS`, welches die Aktion bei sämtlichen nicht explizit spezifizierten Möglichkeiten ausführt.

For-Loop-Kommando

```
[<loop_label>:]
FOR <variable_name> IN <n1> TO | DOWNTO <n2> LOOP
      sequence_of_statements
END LOOP [<loop_label>];
```

Das For-Loop-Kommando implementiert eine Schleife, deren Laufvariable nacheinander die Werte von $n1$ bis $n2$ annimmt (Benutzung von `TO` und `DOWNTO` wieder wie bei den Vektorgrenzen) und für jeden Wert die nachfolgenden Kommandos ausführt. $n1$ und $n2$ können dabei Ausdrücke aus Variablen und Konstanten sein, nicht aber aus Signalen, so daß FOR-Schleife auch keine zeitliche Reihenfolge impliziert.

While-Loop-Kommando

```
[<loop_label>:]
WHILE condition LOOP
      sequence_of_statements
END LOOP [<loop_label>];
```

Das While-Loop-Kommando implementiert eine Schleife, die so lange ausgeführt wird, wie die Bedingung `condition` wahr ist. `condition` selber kann wiederum nur ein Ausdruck aus Variablen und Konstanten sein, so daß die WHILE-Schleife analog zur FOR-Schleife keine zeitliche Reihenfolge der Abarbeitung des Schleifenkörpers mit sich bringt.

5.1.5 Operatoren

Operatoren in VHDL werden in Bedingungen, bei Werten und bei Ausdrücken verwendet. Wie jede übliche Programmiersprache bietet VHDL hiervon eine ganze Reihe an.

Logische:

`AND` Logisches UND

`OR` Logisches ODER

`NAND` Logisches NAND

`NOR` Logisches NOR

`XOR` Logisches Exklusiv-ODER

`XNOR` Logisches Exklusiv-NOR

Vergleich:

`=` Gleichheit

`/=` Ungleichheit

`<` Kleiner (für Integer)

`<=` Kleiner oder gleich (für Integer)

`>` Größer (für Integer)

`>=` Größer oder gleich (für Integer)

Schieben:

`SLL` Linksseitiges logisches Schieben

`SRL` Rechtsseitiges logisches Schieben

`SLA` Linksseitiges arithmetisches Schieben

`SRA` Rechtsseitiges arithmetisches Schieben

`ROL` Linksseitiges Rotieren

`ROR` Rechtsseitiges Rotieren

Addition:

`+` Addition

`-` Subtraktion

Vorzeichen:

`+` Positives Vorzeichen

`-` Negatives Vorzeichen

Multiplikation:

`*` Multiplikation (für Integer)

`/` Division (für Integer)

`MOD` Modulo (für Integer)

Verschiedene:

`**` Potenzierung (für Integer)

`ABS` Absolutwert (für Integer)

`NOT` logische Negierung (boolean oder Standard-Logik)

Jede Gruppe von Operatoren hat die gleiche Präzedenz. Die Gruppen selber sind nach aufsteigender Präzedenz geordnet.

5.1.6 Attribute

Attribute machen im „großen" VHDL zusätzliche Angaben über Entity- oder Architecture-Blöcke, Signale und Typen. In unserer abgespeckten Variante beschränken wir uns jedoch lediglich auf eine hauptsächliche Nutzung von Attributen.

```
<signal_name>'EVENT
```

nimmt genau dann den Wert `TRUE` an, wenn sich der Wert des Signals `<signal_name>` ändert.

5.2 Beispiele zur Benutzung der VHDL-Sprachmittel

5.2.1 Eine exemplarische Entity-Deklaration

```
ENTITY register8 IS
    PORT (
        clk, rst, en:   IN std_logic;
        data:           IN std_logic_vector(7 DOWNTO 0);
        q:              OUT std_logic_vector(7 DOWNTO 0)
    );
END register8;
```

Zu beachten sind hier zwei Dinge:

- Das fehlende Semikolon hinter dem Vektor q: In VHDL ist die PORT-Liste eine mit Semikolon getrennte Aufzählung von Signalen, so daß am Ende das Semikolon auf jeden Fall fehlt.
- Die Benutzung von DOWNTO anstelle von TO führt zu einer natürlicheren Benutzung von Bitvektoren. Ein konstanter Vektor kann somit von links nach rechts mit der herkömmlichen Bedeutung interpretiert werden: MSB kommt als erstes. `"10011011"` wird daher als hexadezimal 9B ($=155_{10}$) anstatt als D9 ($=217_{10}$) aufgefaßt.

5.2.2 Eine dazu passende Architekturdeklaration

Benutzung von Prozessen

```
ARCHITECTURE archregister8 OF register8 IS
BEGIN
    PROCESS (rst, clk)
    BEGIN
        IF (rst='1') THEN
            q <= 0;
        ELSEIF (clk'EVENT AND clk='1') THEN
            IF (en='1') THEN
                q <= data;
            ELSE
                q <= q;
            END IF;
        END IF;
    END PROCESS;
END archregister8;
```

Zu beachten sind hierbei folgende Dinge:

- Die Architecture kann eine von mehreren sein, die zu einer Entity gehören. Man kann somit unterschiedliche Implementationen eines Moduls oder unterschiedliche Abstraktionen desselben (z. B. eine Struktur- und eine Verhaltensbeschreibung) im Vergleich leicht austesten.
- Die Sensitivity List des Prozesses gibt an, welche Signaländerungen eine Aktivierung des Prozesses auslösen. Zusammen mit der geschachtelten IF-Anweisung wird hiermit folgendes Verhalten implementiert:
  - Ein asynchroner Reset: Sobald das Signal `rst` logisch 1 wird, egal zu welchem Zeitpunkt, so wird `q` auf den Wert 0 gesetz. (An `q`, welches ja ein `std_logic_vector` der Länge 8 Bit ist, kann der Integerwert 0 nur deshalb direkt zugewiesen werden, weil `std_logic_vector` als sogenannter *Resolved Data Type* eine automatische Typumwandlung beherrscht.)
  - Ein synchroner Wertewechsel mit steigender Taktflanke: Wenn `clk` seinen Wert ändert und kein Reset gewünscht wird, so wird die IF-Abfrage `ELSEIF (clk'EVENT AND clk='1') THEN`

ausgeführt, die nur bei einer Wertveränderung *und* nachfolgendem logischem 1-Wert (also einer steigenden Flanke) wahr wird.

- Innerhalb des durch `clk`-Aktivität aktivierten ELSEIF-Blocks sind alle weiteren Abfragen (hier ja nur eine, die nach dem Signal `en`) vollständig ausdekodiert. Hier bewirkt die explizite Nennung von `q <= q`, daß in der Synthese ein Speicherelement, ein Flipflop beispielsweise, eingefügt wird. Eine andere recht häufige aber nicht so gut lesbare Methode mit dem gleichen Effekt hätte darin bestanden, den ELSE-Zweig einfach vollständig wegzulassen. Hierbei hätte die Semantik von VHDL zur Folge gehabt, daß ein Speicherelement eingefügt werden müßte, um den Wert von `q` über eine einzige Aktivierung des Prozesses hinaus zu erhalten.

5.2.3 Nebenläufige Zuweisungen

Beispiele für einfache nebenläufige Zuweisungen, das heißt also Zuweisungen von Signalen, die gleichzeitig stattfinden, wären:

```
v <= a AND b AND c;
w <= a OR b AND c;
y <= a NAND b XOR c;
```

Zu beachten ist hierbei, daß alle booleschen Operatoren die *gleiche* Präzedenz besitzen! Um beispielsweise den Ausdruck für `w` eindeutiger zu beschreiben, wäre also eine explizite Klammerung notwendig.

Die bedingte Zuweisung sieht dann entsprechend aus:

```
a <= '1' WHEN b = c ELSE '0';
```

Nur im Falle von Gleichheit von `b` und `c` wird hierbei `a` auf logisch 1 gesetzt. Diese Form einer Zuweisung ist besonders dann bequem und effizient, wenn nur ein ganz bestimmer Fall aus einer großen Menge von Möglichkeiten für eine bestimmte Zuweisung sorgt[2].

5.2.4 Komponenten und ihre Instantiierung

Als Beispiel für eine Deklaration einer Komponente soll mal wieder das 8-Bit-breite Registermodul dienen:

```
COMPONENT register8 IS
    PORT (
        clk, rst, en:   IN std_logic;
        data:           IN std_logic_vector(7 DOWNTO 0);
        q:              OUT std_logic_vector(7 DOWNTO 0)
    );
END COMPONENT;
```

Steht diese Deklaration zusätzlich zu einem Entity-Architecture-Paar im Quelltext, so heißt dies, das von nun an die Komponente `register8` wie ein Bauteil verwendet werden kann, und zwar beispielsweise so:

```
ARCHITECTURE use_of_register8 OF example_reg8 IS
    SIGNAL clock, reset, enable:    STD_LOGIC;
    SIGNAL data_in, data_out:       STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
  -- irgendwelcher anderer Code (Prozesse, Zuweisungen, etc.) hier

  First_reg8: register8 PORT MAP (
```

[2]Als Alternative kann man ja mal vergleichen, wieviele Zeilen VHDL man braucht, um das gleiche Verhalten mithilfe eines Prozesses zu beschreiben.

```
      clock, reset, enable, data_in, data_out );

  -- weiterer Code hier

END use_of_register8
```

Zu beachten ist hier:

- Der Name `First_reg8` ist der Name dieser einen Instantiierung. Eine weitere Instanz dieser Komponente kann selbstverständlich noch hinzugefügt werden.
- Die Architecture-internen Signale (z. B. `clock` oder `reset`) stehen in der Port-Liste der instantiierten Komponente an der gleichen Stelle wie ihre entsprechenden Ein- und Ausgänge in der Komponentendeklaration. Man nennt dies *Positional Association*, welche im Rahmen von TGI ausschließlich genutzt wird[3].

[3]VHDL kann mehr, z. B. die sogenannte *Named Association*, die aber lediglich eine Lesehilfe ist.

5.2.5 Sequentielle Sprachkonstrukte und ihre Anwendung

Die sequentiellen Sprachkonstrukte werden ausschließlich innerhalb von Prozessen angewandt.

5.2.5.1 Die einfache Abfrage

Mithilfe von IF-THEN-ELSEIF-ELSE kann eine komplexe Abfrage von Bedingungen in VHDL realisiert werden:

```
IF (count = "00") THEN
    a <= b;
ELSIF (count = "10") THEN
    a <= c;
ELSE
    a<= d;
END IF;
```

5.2.5.2 Die komplexe Abfrage

Das CASE-Statement dient zur effizienten Auswahl aus einer ganzen Reihe von Möglichkeiten. Ein Ausdruck wird dazu mit einem Angebot von Konstanten verglichen[4].

```
CASE count IS
    WHEN "00" =>
        a <= b;
    WHEN "10" =>
        a <= c;
    WHEN OTHERS =>
        a <= d;
END CASE;
```

Hierbei ist zu beachten, daß das Schlüsselwort OTHERS alle Möglichkeiten für nichtabgedeckte Fälle der Auswahl selektiert.

5.2.5.3 Die For-Schleife

Für einen diskreten Bereich von Werten kann ein Schleifenblock ausgeführt werden:

```
meine_for_schleife:
FOR i IN 3 DOWNTO 0 LOOP
    IF reset(i) = '1' THEN
        data_out(i) <= '0';
    END IF;
END LOOP meine_for_schleife;
```

Der Bezeichner `i` repräsentiert hierbei eine Variable und kein Signal.

5.2.5.4 Die While-Schleife

```
meine_while_schleife:
WHILE (count > 0) LOOP
    count := count - 1;
    result <= result + data_in;
END LOOP meine_while_schleife;
```

[4] Im vollständigen VHDL können auch Ausdrücke anstelle von Konstanten verwendet werden, was die Auswahl noch flexibler macht.

Zu beachten ist hier der Unterschied zwischen `count` (einer Variablen, erkennbar an der Zuweisung `:=`) und `result` (einem Signal, auch erkennbar an der Zuweisung `<=`).

5.2.6 Verbindung von synchroner Logik und asynchronem Verhalten durch Prozesse

Unter synchroner Logik versteht man Zustandsautomaten, deren Zustandsänderung jeweils nur *synchron* mit einem Taktsignal stattfindet. Von Zeit zu Zeit ist es jedoch auch nötig, unabhängig vom zeitlichen Verlauf des Taktsignals, also *asynchron* zu ihm, eine Zustandsänderung des Automaten hervorzurufen. In VHDL kann man dies auf bestimmte Weise ausdrücken.

5.2.6.1 Grundlage: Ein 8-Bit-Register

Zugrundegelegt sei hier ein 8 Bit breites Register, welches zum Zeitpunkt der steigenden Flanke des Taktsignales `clk` geladen wird.

```
reg8_no_reset:
PROCESS (clk)
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        q <= data;
    END IF;
END PROCESS reg8_no_reset;
```

Dabei seien `clk` vom Typ `STD_LOGIC` sowie `q` und `data` vom Typ `STD_LOGIC_VECTOR (7 DOWNTO 0)`. Der Prozeß wird innerhalb einer Architecture verwendet.

5.2.6.2 8-Bit-Register mit synchronem Reset

Benötigt man eine Rücksetzfunktion lediglich synchron mit dem Taktsignal, so kann folgender Ansatz verwendet werden.

```
reg8_sync_reset:
PROCESS (clk)
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        IF sync_reset = '1' THEN
            q <= "00000000";
        ELSE
            q <= data;
        END IF;
    END IF;
END PROCESS;
```

Das zusätzliche Signal `sync_reset` sei erneut vom Typ `STD_LOGIC`.

5.2.6.3 8-Bit-Register mit asynchronem Reset

Bei einer asynchronen Funktion muß das auslösende Signal mit in die Sensitivity List des Prozesses aufgenommen werden.

```
reg8_async_reset:
PROCESS (clk, async_reset)
BEGIN
    IF async_reset = '1' THEN
        q <= "00000000";
    ELSIF clk'EVENT AND clk = '1' THEN
        q <= data;
    END IF;
END PROCESS;
```

Der Prozeß wird nun in VHDL auch „ausgeführt“, sobald sich eine Signaländerung am Signal `async_reset` (Typ `STD_LOGIC`) ergibt. Die Präzedenz der durch die Signale `clk` und `async_reset` ausgelösten Aktionen, also ihre Vorrangigkeit, kann bestimmt werden durch die Position des entsprechenden Signales in einer Reihe von `IF ... ELSIF ... END IF` Abfragen bestimmt. Der bemerkenswerte Unterschied zwischen beiden Signalen im Beispiel ist das Fehlen der Abfrage des `EVENT`-Attributes beim Signal `async_reset`. Dies bedeutet, daß die Resetbedingung auch dann als erstes gilt, wenn sich `async_reset` nicht selber geändert hat[5]. Selbst wenn also der Takt ordnungsgemäß weiterläuft, so kann bei gesetztem Resetsignal kein Wert aus `data` in das Register geladen werden.

5.2.6.4 Asynchron rücksetzbares Register mit synchroner Ladeberechtigung (Enable)

Synchrone und asynchrone Funktionen lassen sich auch kombinieren.

```
reg8_sync_assign_async_reset:
PROCESS (reset, clk)
BEGIN
    IF reset = '1' THEN
        q <= "00000000";
    ELSIF clk'EVENT AND clk = '1' THEN
        IF enable = '1' THEN
            q <= data;
        ELSE
            q <= q;
        END IF;
    END IF;
END PROCESS;
```

Abhängig von `clk`, also synchron zum Takt, wird hier die Abfrage von `enable` (Typ `STD_LOGIC`) eingesetzt, um so ein Laden des Zählers lediglich bei aktivierter Ladeberechtigung zuzulassen. Wie schon in 5.2.2 beschrieben, sorgt die explizite Angabe von `q <= q;` dafür, daß ein Synthesewerkzeug hierbei notwendigerweise Speicherelemente (Flipflops) einbaut, um diese Bedingung zu erfüllen. Alternativ dazu ist es auch möglich, den `ELSE`-Zweig dieser Abfrage fortzulassen. Die Semantik von VHDL, die besagt, daß jede Signalinformation eines Prozesses, die nicht innerhalb des Prozesses verändert wird, über mehrere Aufrufe hinweg erhalten bleibt, sorgt implizit dafür, daß auch in einem solchen Fall Speicherelemente benutzt würden. Man spricht im letzteren Fall von *Inferred Logic*.

[5] Aus digitaltechnischer Sicht bedeutet dies, das der Reset ein pegelgesteuerter Eingang ist, während der Takt ein flankengesteuertes Signal ist.

5.2.7 Ein vollständiges Beispiel: Ladbarer Zähler mit Nulldurchlauferkennung

Der folgende ladbare 8-Bit Zähler, der zurückgesetzt und geladen werden kann sowie aufwärts und abwärts zählen kann ist ein gutes Beispiel zum Üben von VHDL.

```
ENTITY up_down_count IS
    PORT (
        in_data:    IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        value:      BUFFER STD_LOGIC_VECTOR(7 DOWNTO 0);
        zero:       OUT STD_LOGIC;
        clk:        IN STD_LOGIC;
        reset:      IN STD_LOGIC;
        load:       IN STD_LOGIC;     -- '0' = count, '1' = load
        up_down:    IN STD_LOGIC;     -- '0' = up, '1' = down);
END up_down_count;

ARCHITECTURE arch_ud_count OF up_down_count IS
BEGIN

    counter_fsm:
    PROCESS (clk,reset)
    BEGIN
        IF reset = '1' THEN
            value <= "00000000";
        ELSIF clk'EVENT AND clk = '1' THEN
            IF load = '1' THEN
                value <= in_data;
            ELSIF up_down = '0' THEN
                value <= value + 1;
            ELSE
                value <= value - 1;
            END IF;
        END IF;
    END PROCESS;

    zero <= '1' WHEN value = "00000000" ELSE '0';

END arch_ud_count;


COMPONENT up_down_count IS
    PORT (
        in_data:    IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        value:      BUFFER STD_LOGIC_VECTOR(7 DOWNTO 0);
        zero:       OUT STD_LOGIC;
        clk:        IN STD_LOGIC;
        reset:      IN STD_LOGIC;
        load:       IN STD_LOGIC;     -- '0' = count, '1' = load
        up_down:    IN STD_LOGIC;     -- '0' = up, '1' = down);
END COMPONENT;
```