MI – Eine Maschine für die Informatikausbildung

Uwe Borghoff
Thomas Gasteiger
Anna Schmalz, Peter Weigele

o. Prof. Dr. H.J. Siegert

Inhaltsverzeichnis

Die Maschine für die Informatikausbildung, kurz MI, wird seit 1985 an der TU München und seit 1987 an anderen deutschen Universitäten eingesetzt.

Nach dem Entwurf von Prof. H.-J. Siegert wurde die Maschine von Anna Schmalz und Peter Weigele realisiert. Klaus-Dieter Schmatz und Hans-Michael Windisch implementierten die Mehr-Prozessor-MI, Franz Huber entwickelte die Graphikoberfläche.

Besonderer Dank gebührt Christine Bauer, Michael Breu, Elfriede Bunke, Thomas Gasteiger und Ludwig Zagler. Die Koordination aller Arbeiten liegt bei Uwe Borghoff.

Kapitel 1

Modell der Maschine MI

Die Maschine für die Informatikausbildung gibt es in zwei Varianten:

1. als *Ein-Rechnerkern* Maschine, kurz 1-RK-MI und

2. als *Vier-Rechnerkern* Maschine, kurz 4-RK-MI.

Im folgenden beschreiben wir das Modell der 4-RK-MI, erwähnen aber die Unterschiede zur 1-RK-MI an den entsprechenden Stellen.

Die Modellmaschine besteht aus folgenden Komponenten:

RKi	4 Rechnerkerne (Instruktionsprozessoren), $i \in \{0 \ldots 3\}$ (in der 1-RK-MI gilt: $i = 0$)
EAP1	EA-Prozessor 1, betreibt 2 Plattenspeicher-Laufwerke
EAP2	EA-Prozessor 2, betreibt 4 Duplexkanäle (darunter sind auch die Terminal- und Druckerkanäle)
ASP	Arbeitsspeicher

Diese Komponenten sind über einen gemeinsamen Bus miteinander verbunden. Der bzw. die Rechnerkern(e) sowie die EA-Prozessoren können parallel arbeiten.

1.1 Datentypen

Die kleinste adressierbare Einheit im ASP ist ein Byte. Die Objekte im ASP können jedoch unabhängig von der Adressierung folgende *Kennungen* haben:

B	Byte, Zeichen; 8 Bit breit
H	Halbwort; 16 Bit breit
W	Wort; 32 Bit breit
F	Gleitpunktzahl (floating point); 32 Bit breit
D	Double-float; 64 Bit breit

Die Interpretation des Bitmusters eines Objektes ist operationsabhängig. Nachfolgend werden die Zahldarstellungen angegeben.

1.1.1 Natürliche Zahl $\mathcal{N}^{(n)}$

b_0		b_{n-1}	←	Bit-Nummer
2^{n-1}		2^0	←	Wertigkeit des Bits

Wert $W_{\mathcal{N}} = \sum_{i=0}^{n-1} b_i \times 2^{n-1-i}$, wobei $b_i \in$ 0,1 der Wert des i-ten Bit ist.

Wertebereich: $0 \leq W_{\mathcal{N}} \leq 2^n$ -1

Zulässige Werte von n:

$n = 8$: Byte / Zeichen (B)

$n = 16$: Halbwort (H)

$n = 32$: Wort (W)

1.1.2 Ganze Zahl $\mathbf{Z}^{(n)}$

Darstellung im 2-Komplement

b_0	b_1		b_{n-1}	←	Bit-Nummer
	2^{n-2}		2^0	←	Wertigkeit des Bits

Vorzeichen $(-1)^s$, wobei $s = b_0 \in \{0,1\}$

Wert positiver Zahlen (s=0): $W_{\mathcal{Z}} = \sum_{i=1}^{n-1} b_i \times 2^{n-1-i}$, wobei $b_i \in \{0,1\}$

Wert negativer Zahlen (s=1): $W_{\mathcal{Z}} = -\{ 1 + \sum_{i=1}^{n-1}(1 - b_i) \times 2^{n-1-i}\}$

Wertebereich: $-2^{n-1} \leq W_{\mathcal{Z}} \leq 2^{n-1} - 1$

Es gilt: $W_{\mathcal{Z}} = W_{\mathcal{N}} - b_0 \times 2^n$

Zulässige Werte von n:

$n = 8$: Byte (B)

$n = 16$: Halbwort (H)

$n = 32$: Wort (W)

1.1.3 Gleitpunktzahl

b_0	b_1		b_m	b_{m+1}		b_{m+n}	←	Bit-Nummer
	2^{m-1}		2^0	2^{-1}		2^{-n}	←	Wertigkeit des Bits
	Exponent			Mantisse				

Vorzeichen der Zahl: $(-1)^s$, wobei $s = b_0 \in \{0,1\}$

Exponent: $e = \sum_{i=1}^{m} b_i \times 2^{m-i}$, wobei $b_i \in \{0,1\}$

Wertebereich: $0 \leq e \leq 2^m - 1$

Abkürzung: $e_{max} := 2^m - 1$

$e^* := 2^{m-1} - 1$

Mantisse: $f = \sum_{i=1}^{n} b_{m+i} \times 2^{-i} = 2^{-n} \times \sum_{j=0}^{n-1} b_{m+n-j} \times 2^j$

Wertebereich: $0 \leq f < 1$

Wert der Gleitpunktzahl: Man unterscheidet folgende 5 Fälle:

a) $e = e_{max} \wedge f \neq 0$: w = not a number (NaN)
b) $e = e_{max} \wedge f = 0$: w = $(-1)^s \times \infty$ (Überlauf)
c) $e = 0 \wedge f \neq 0$: w = $(-1)^s \times 2^{-(e^*-1)} \times f$ (nicht mehr normalisierte Zahl)
d) $e = 0 \wedge f = 0$: w = $(-1)^s \times 0$
e) $0 < e < e_{max}$: w = $(-1)^s \times 2^{e-e^*} \times (1+f)$ (normalisierte Darstellung)

Anschauliche Erklärung:
In der überwiegenden Mehrzahl aller Fälle liegt eine normalisierte Darstellung vor, also Fall e. Bei dieser Darstellung befindet sich in Position m stets eine 1, auf deren Abspeicherung aus Effizienzgründen durchwegs verzichtet wird (siehe Tabelle 1.1).

	kurze GP-Zahl (F)	lange GP-Zahl (D)
$m+n+1$	32	64
m	8	11
n	23	52
e_{max}	2^8 - 1 = 255	2^{11} - 1 = 2047
e^*	2^7 - 1 = 127	2^{10} - 1 = 1023

Tabelle 1.1: Zulässige Werte von n und m bei Gleitpunktzahlen

1.2 Speicherung der Daten

Das höchstwertige Bit (Bitnummer 0) einer Größe sei im Byte mit der Adresse A enthalten. Dann gilt allgemein: Bit b_i ist in der Speicherzelle mit Adresse A + (i div 8). Beispiel für ein Wort:

A:	b_0	b_7
A+1:	b_8	b_{15}
A+2:	b_{16}	b_{23}
A+3:	b_{24}	b_{31}

Die Adresse dieses Worts ist A.

Eine lange Gleitpunktzahl (Kennung D) befindet sich bei Adressierung des Registers R_i in den Registern R_i und $R_{(i+1) mod 16}$.

1.3 Register

Jeder Rechnerkern besitzt 16 frei programmierbare Register, die von 0 bis 15 numeriert sind. Die Wortbreite eines Registers beträgt 32 Bit. Jedoch kann durch geeignete Kennung

auch auf Teile eines Registers zugegriffen werden (bei B auf das rechte Byte, bei H auf das rechte Halbwort und bei W auf das gesamte Register).

Es gibt zwei ausgezeichnete Register: Der Befehlszähler **PC** (Program Counter) enthält die Adresse des nächsten auszuführenden Befehls bzw. des noch zu interpretierenden Teils des aktuellen Befehls. Das Register **R15** ist gleichzeitig der Befehlszähler **PC**.

Der Kellerpegel **SP** (Stack Pointer) enthält die Adresse des aktuellen Kellers. Das Register **R14** ist gleichzeitig der Kellerpegel **SP**.

Es gibt noch eine ganze Reihe von Sonderregistern, die im folgenden noch erklärt werden.

Alle Adressen sind vorzeichenlose ganze Zahlen im Wortformat (32 Bit Adressen).

1.4 Arbeitsmodus und Adressierungsarten

Der Arbeitsmodus ist bei der Ausführung privilegierter Befehle und beim Zugriff auf Seiten maßgebend. Es gibt:

- den System(kern)modus (kernel mode)
- den Benutzermodus (user mode)

Daneben gibt es 2 Adressierungsarten:

- die direkte Speicheradressierung, d.h. alle Programmadressen sind reale Arbeitsspeicheradressen (Maschinenadressen).
- die Seitenadressierung, d.h. alle Programmadressen werden durch Speicherabbildungstabellen auf reale Arbeitsspeicheradressen abgebildet.

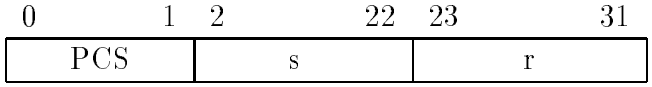
Der eingestellte Arbeitsmodus steht im Feld "aktueller Arbeitsmodus" (current mode) des Programmstatusworts. Die eingestellte Adressierungsart ist in dem Register **RKMAP** vermerkt. Seitenadressierung ist eingestellt, falls im rechten Halbwort von **RKMAP** eine 1 steht.

Anmerkung: Das linke Halbwort von **RKMAP** enthält die Rechnerkernnummer. Folgende Arbeitsweise des Betriebssystems wird vorausgesetzt: Jedem Arbeitsmodus ist ein spezifischer Keller zugeordnet. Der Keller im Benutzermodus (prozeß-spezifisch) ist normalerweise im P1-Bereich (siehe Speichermanagement). Er unterliegt dem Seitenwechsel. Der Keller für den Systemkernmodus bei einem aktiven Prozeß muß im ASP geladen sein.

Doppelnatur des Kellerpgels: Bei Wechsel des Arbeitsmodus wird der Kellerpegel **SP** aktualisiert, d.h. er zeigt immer auf den gerade benötigten Keller.

1.5 Speichermanagement

Die kleinste Adressierungsart ist das Byte. Eine virtuelle Adresse (Programmadresse, Seitenadresse) hat folgenden Aufbau:



wobei

- r die Relativadresse eines Bytes in einer Seite ist
- s die Seitennummer ist (Seitengröße = 2^9 = 512 Bytes)
- die Segmentnummer PCS angibt, in welchem Bereich die Adresse liegt.

Die Maschinenarchitektur ist geeignet, folgende Arbeitsweise des Betriebssystems zu unterstützen. Der Speicher ist dabei aufgeteilt in 4 Bereiche:

- den P0-Bereich (P0-region, program region) bei PCS = 0. Dieser Bereich enthält in der Regel das Benutzerprogramm. Er beginnt bei Adresse H'00000000'; bei dynamischer Veränderung wächst er in die Richtung ansteigender Adressen.
- den P1-Bereich (P1-region, control region) bei PCS = 1. Dieser Bereich enthält den Benutzerkeller. Er beginnt bei Adresse H'7FFFFFFF'; bei dynamischer Veränderung wächst er in die Richtung fallender Adressen.
- den Systembereich (system region) bei PCS = 2. Dieser Bereich ist in der Regel in allen Prozessen einheitlich durch die BS-Komponenten belegt, d.h. das BS ist in jedem Programmadreßraum.
- einen nicht benutzten (unzulässigen) Bereich, gekennzeichnet durch PCS = 3.

Zur Aufteilung des Programmadreßraums siehe Tabelle 1.2.

Die 3 zulässigen Bereiche des Programmadreßraums werden durch je eine Seitentafel auf reale Maschinenadressen abgebildet. Die Anfangsadressen dieser Seitentafeln stehen in Rechnerkernregistern:

POB Anfangsadresse der Seitentafel für den P0-Bereich (P0-page table base). Die Seitentafel liegt im Systembereich.

P1B Anfangsadresse der Seitentafel für den P1-Bereich (P1-page table base). Eingetragen wird dabei die Anfangsadresse des fiktiven Seitentafelelements für Seite 0. Die Seitentafel liegt im Systembereich.

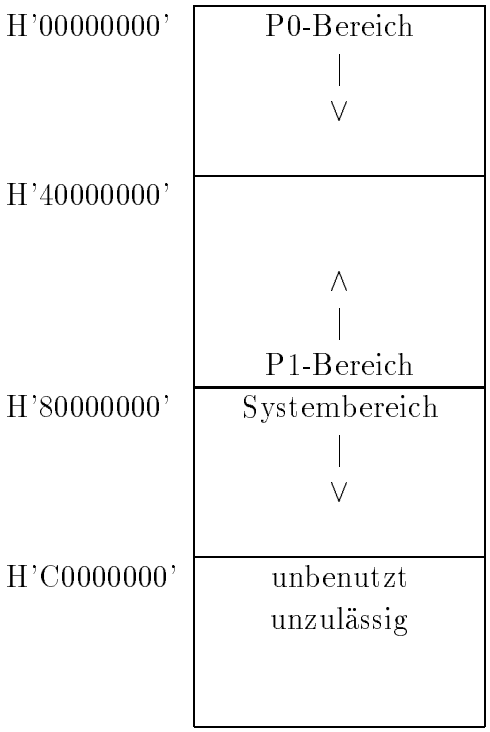


Tabelle 1.2: Aufteilung des Programmadreßraums

SB Anfangsadresse der Seitentafel für den Systembereich (system-page table base). Die Seitentafel liegt im Arbeitsspeicher.

Die aktuelle Ausdehnung der Bereiche (Länge) steht in folgenden Registern:

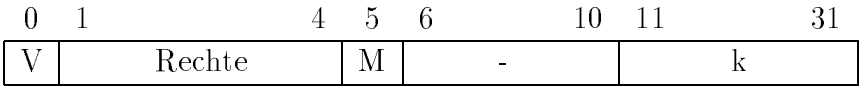
POL Nummer der ersten nicht mehr in der Seitentafel befindlichen Seite des P0-Bereichs (P0-page table length).

P1L Nummer der ersten in der Seitentafel für den P1-Bereich befindlichen Seite (P1-page table length).

SL Nummer der ersten nicht mehr in der Seitentafel befindlichen Seite des Systembereichs (system-page table length).

In üblichen Betriebssystemrealisierungen werden Seitentafeln für die Bereiche P0 und P1 je Prozeß benötigt, die Seitentafel für den Systembereich dagegen nur einmal im System.

Ein Seitentafelelement (PTE, page table entry) hat folgenden Aufbau:



Hierbei bedeuten:

V = 1: Die Abbildung ist gültig (valid), d.h. der betreffenden Seite ist die Kachel k (Rahmen, page frame) im realen Maschinenadreßraum zugeordnet.

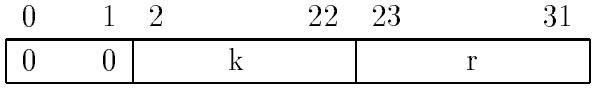
M = 1: Auf die betreffende Seite wurde schreibend zugegriffen (modify-bit). Dieses Bit wird vom jeweiligen Rechnerkern gesetzt.

Zugriffsrechte für den Zugriff auf die Seite sind unabhängig vom V-Bit gültig. Folgende Werte sind zulässig, wobei **R** für Leserecht (*read*) und **W** für Schreibrecht (*write*) steht. Die Rechte des System(kern)s sind dabei stets mindestens diejenigen des Benutzers. Damit sind nicht alle Kombinationen zulässig (siehe Tabelle 1.3).

Wert	Abkürzung	Systemmodus	Benutzermodus	Erläuterung
0	NA	-	-	no access
2	KW	RW	-	kernel write
3	KR	R	-	kernel read
4	KWUW	RW	RW	kernel write/user write
14	KWUR	RW	R	kernel write/user read
15	KRUR	R	R	kernel read/user read

Tabelle 1.3: Zugriffsrechte

Eine reale Maschinenadresse hat folgenden Aufbau (siehe auch Kapitel 3).



r Relativadresse eines Bytes in der Kachel
k Kachelnummer

Zur Beschreibung der Abbildung werden folgende Größen eingeführt:

```
type  seitentafelelement = record
      V: boolean;
      zugriffsrechte: (NA, KW, KR, KWUW, KWUR, KRUR);
      M: boolean;
      k: integer;
      endrecord;
type  programmadresse = record
      PCS: (P0-bereich, P1-bereich, systembereich);
      s: integer;
      r: integer;
      endrecord;
```

```
type  maschinenadresse = record
      k: integer;
      r: integer;
      endrecord;
type  ergebnis = (erfolgreich, speicherschutzalarm, seitefehltalarm);
```

Es folgt der Algorithmus für die Speicherabbildung im Rechnerkern. Es wird unterstellt, daß Variablen und Typen gleich benannt sein dürfen.

```
procedure speicherabbildung (
     programmadresse: programmadresse;
     var maschinenadresse: maschinenadresse;
     zugriffsart: (lesen, schreiben);
     arbeitsmodus: (systemkernmodus, benutzermodus);
     var ergebnis: ergebnis );
var  PTB: programmadresse; /* der Seitentafel */
     PTEPADR: programmadresse; /* eines Seitentafelelements */
     PTESPADR: maschinenadresse; /* eines Seitentafelelements */
     PTE: seitentafelelement;
     /* P0L, P0B, P1L, P1B, SL, SB bezeichnen Register */

begin
if       ((arbeitsmodus = systemkernmodus and RKMAP[b16...b31 ] = 0)
         or (arbeitsmodus = benutzermodus and PSW[b5] = 0))
then     maschinenadresse.k := programmadresse.s;
         maschinenadresse.r := programmadresse.r;
         ergebnis := erfolgreich return; fi;
```

```
with  programmadresse  do
      case PCS of
      P0_Bereich:      begin
                       if s ≥ P0L then
                       ergebnis := speicherschutzalarm; return; fi;
                       PTB := P0B;
                       end;
      P1_Bereich:      begin
                       if s < P1L then
                       ergebnis := speicherschutzalarm; return; fi;
                       PTB := P1B;
                       end;
      systembereich:   begin
                       if s ≥ SL then
                       ergebnis := speicherschutzalarm; return; fi;
                       PTB := SB;
                       end;
      endcase;

      /* nun Adresse des Seitentafelelements berechnen */
      PTEPADR := PTB + 4 × s;
```

```
        case PCS of
        systembereich:  begin
                        /* PTEPADR enthält reale Speicheradresse */
                        PTESPADR := PTEPADR;
                        end;
        P0_bereich,
        P1_bereich:     begin
                        /* PTEPADR enthält Adresse des Systembereichs */
                        if PTEPADR.PCS <> 2 then
                        ergebnis := speicherschutzalarm; return; fi;
                        speicherabbildung ( PTEPADR, PTESPADR,
                        schreiben, systemkernmodus, ergebnis );
                        if ergebnis <> erfolgreich then
                        return; fi;
                        end;
        endcase;

        /* nun Seitentafelelement aus Arbeitsspeicher holen */
        PTE := Arbeitsspeicherwort ab Adresse PTESPADR;

        /* nun Zugriffsrechte abprüefen, zulässige Kombinationen:
```

arbeitsmodus	zugriffsart	notwendige Zugriffsrechte in PTE
systemkernmodus	schreiben	KW $\vee$ KWUW $\vee$ KWUR
systemkernmodus	lesen	KW $\vee$ KWUW $\vee$ KWUR $\vee$ KR $\vee$ KRUR
benutzermodus	schreiben	KWUW
benutzermodus	lesen	KWUW $\vee$ KWUR $\vee$ KRUR

```
        */
        if not ausreichende zugriffsrechte then
           ergebnis := speicherschutzalarm; return; fi;
        /* Seite im ASP? */
        if not PTE.V then ergebnis := seitefehltalarm; return; fi;

        /* nun Abbildung durchführen */
        maschinenadresse.k := PTE.k;
        maschinenadresse.r := programmadresse.r;
        ergebnis := erfolgreich;
        if zugriffsart = schreiben then
           PTE.M := true;
           PTE ab PTESPADR zurückspeichern in ASP; fi;
        return;
endwith; endproc;
```

1.6 Unterbrechungen

1.6.1 Unterbrechungsursachen und Unterbrechungsprioritäten

Wir unterscheiden interne Unterbrechungen (exceptions) und externe Unterbrechungen (interrupts).

- Interne Unterbrechungen treten bei der Ausführung von Befehlen im Rechnerkern selbst auf.
- Externe Unterbrechungen werden von anderen Komponenten des Systems, z.B. von Geräten oder anderen Rechnerkernen[1], ausgelöst.

Jeder Unterbrechungsart ist eine Unterbrechungsbehandlungsroutine zugeordnet, die im Systembereich liegt. Die Adresse des Anfangs jeder Unterbrechungsbehandlungsroutine steht an definierter Stelle des Systemkontrollblocks (SCB). Die Arbeitsspeicheradresse des Anfangs des Systemkontrollblocks steht in dem Register **SCBADR**.

Jeder Unterbrechungsart ist eine Unterbrechungspriorität (0 bis 31) zugeordnet, weiterhin hat jeder Ablauf im System (d.h. die Prozesse und die Unterbrechungsroutinen) eine Ablaufpriorität. Die aktuelle Ablaufpriorität steht im Register **IPL**.

Eine Unterbrechung eines Ablaufs durch einen externen Unterbrechungswunsch ist unmittelbar nach Ausführung eines Befehls möglich, falls die Unterbrechungspriorität größer als die Ablaufpriorität ist. Die Ablaufpriorität der Unterbrechungsbehandlung wird gleich der Unterbrechungspriorität. Externe Unterbrechungswünsche werden verzögert, bis die Prioritätsverhältnisse eine Unterbrechung erlauben.

Interne Unterbrechungen erfolgen unabhängig von der Ablaufpriorität sofort. Die Ablaufpriorität der aufgerufenen Unterbrechungsbehandlungsroutine ist gleich derjenigen des unterbrochenen Ablaufs. Prozesse haben stets Ablaufpriorität 0.

1.6.2 Systemkontrollblock

Der Systemkontrollblock enthält die Startadressen der Unterbrechungsbehandlungsroutinen. Die Anfangsadresse des Systemkontrollblocks steht im Register **SCBADR**.

Als Tabelle folgen nun die Relativadressen (innerhalb des SCB) der jeweiligen Unterbrechungsbehandlungsroutine, die zugehörige Priorität und eine Bezeichnung der Unterbrechungsart (siehe Tabelle 1.4).

[1] in der 4-RK-MI

	Priorität	Unterbrechungsart
SCBADR:	31	katastrophaler Fehler
+4	*)	Befehlsalarm
+8	*)	Speicherschutzalarm
+12	*)	Seite-fehlt-Alarm
+16	*)	Trace
+20	*)	arithmetischer Alarm
+24	*)	CHMK (Systemaufruf)
+28	*)	— (für CHMU reserviert)
+32	25	Rechnerkernalarm (nur in 4-RK-MI!)
+36	24	Weckeralarm (nur in 4-RK-MI!)
+40	23	Plattenspeicher
+44	22	Kanal 0 Empfänger
+48	22	Kanal 0 Sender
+52	21	Kanal 1 Empfänger
+56	21	Kanal 1 Sender
+60	20	Kanal 2 Empfänger
+64	20	Kanal 2 Sender
+68	19	Kanal 3 Empfänger
+72	19	Kanal 3 Sender

*) Priorität der Unterbrechungsbehandlungsroutine ist gleich derjenigen des unterbrochenen Ablaufs. Die Startadresse der Unterbrechungsbehandlungsroutine steht in **SCBADR** + Relativadresse.

Tabelle 1.4: Aufbau des Systemkontrollblocks

1.6.3 Prozeßkontrollblock

Im Prozeßkontrollblock (PCB) steht während der Unterbrechung eines Prozesses dessen aktueller Rechnerkernzustand. Die Adresse des aktuellen Prozeßkontrollblocks steht im Register **PCBADR**. Es handelt sich um eine Arbeitsspeicheradresse. Zum Aufbau des Prozeßkontrollblocks siehe Tabelle 1.5.

PCBADR:	KMSP	— **SP** im Systemmodus
+4	UMSP	— **SP** im Benutzermodus
+8	**R0**	
+12	**R1**	
	...	
+60	**R13**	
+64	**R14 = SP**	— ohne Bedeutung
+68	**R15 = PC**	
+72	**PSW**	
+76	**P0B**	— Diese 4 Register
+80	**P0L**	— beschreiben die Lage
+84	**P1B**	— der Seitentafeln für
+88	**P1L**	— den jeweiligen Prozeß.

Tabelle 1.5: Aufbau eines Prozeßkontrollblocks

Das Prozessorstatuswort (**PSW**) hat folgende Bestandteile:

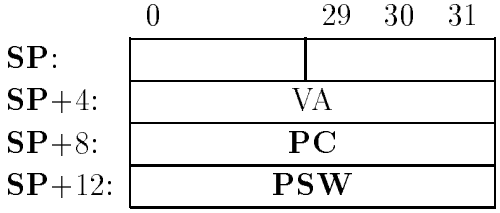
TP in b_1:	trace pending bit
BEN_MAPEN in b_5:	Adressierungsart (Benutzerbit)
	0: direkte Adressierung
	1: Seitenadressierung
CM in $b_6 b_7$:	gegenwärtiger Arbeitsmodus (current mode)
	0: Systemkernmodus
	3: Benutzermodus
PM in $b_8 b_9$:	vorheriger Arbeitsmodus (previous mode)
	0: Systemkernmodus
	3: Benutzermodus
IPL in $b_{11} \ldots b_{15}$:	gegenwärtige Ablaufpriorität (interrupt priority level, zugänglich auch unter der Registerbezeichnung **IPL**)
IV in b_{26}:	integer overflow trap disable (falls gesetzt, löst bei integer-Arithmetik ein Überlauf keine Unterbrechung aus; es wird lediglich V gesetzt).
T in b_{27}:	trace bit
N in b_{28}:	negative condition code
Z in b_{29}:	zero condition code
V in b_{30}:	Overflow-Condition (bei arithmetischen Operationen)
C in b_{31}:	Carry-Bit (bei arithmetischen Operationen)

1.6.4 Wirkungsweise einer Unterbrechung

Bei Unterbrechungen erfolgt zunächst die Ablage des Zustands (**PSW** und **PC**) des unterbrochenen Ablaufs im Systemkeller. Bei internen Unterbrechungen werden zusätzlich Unterbrechungsinformationen im Systemkeller abgelegt. Anschließend wird als neuer Arbeitsmodus der Systemkernmodus eingestellt, die Ablaufpriorität neu gesetzt und auf die Unterbrechungsbehandlungsroutine verzweigt.
Im einzelnen gilt:

1. Es wird das Prozessorstatuswort (**PSW**) im Systemkeller abgelegt mit Fortschaltung des Systemkellerpegels.
2. Es wird **PSW**.PM := **PSW**.CM.
3. Es wird Systemkernmodus eingestellt (**PSW**.CM := 0).
4. Es wird Trace-Bit und Trace-pending-Bit in **PSW** gelöscht.
5. Es wird der Befehlszähler (**PC**, **R15**) im Systemkeller abgelegt mit Fortschaltung des Systemkellerpegels. Der Befehlszähler zeigt auf den nächsten auszuführenden Befehl, außer in folgenden Fällen:
   - Befehlsalarm
   - Speicherschutzalarm
   - Seite-fehlt-Alarm

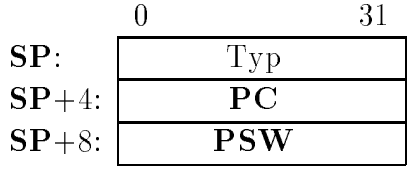
   In diesen Fällen zeigt der Befehlszähler auf den Anfang des gerade ausgeführten Befehls. Dieser kann dadurch ggf. wiederholt werden (Seite-fehlt-Alarm!).
6. Es wird bei externen Unterbrechungen die Ablaufpriorität mit der Unterbrechungspriorität besetzt.
7. Der Befehlszähler wird gemäß der Angabe im Systemkontrollblock neu besetzt, so daß er auf den Anfang der Unterbrechungsbehandlung zeigt.
8. Bei bestimmten Unterbrechungen wird zusätzliche Information im Systemkeller abgelegt mit Fortschaltung des Systemkellerpegels (die angegebenen Adressierungshinweise beziehen sich auf den **SP**-Stand bei Aufruf der Unterbrechungsbehandlungsroutine). Diese zusätzliche Unterbrechungsinformation ist:
   - bei Speicherschutzalarm, Seite-fehlt-Alarm



	b_{29}	b_{30}	b_{31}
Seite-fehlt-Alarm	0	0	0
Speicherschutzalarm			
Seite nicht in Seitentafel	0	0	1
unzureichende Rechte:			
schreibender Zugriff, kein Recht oder Leserecht	1	1	0
lesender Zugriff, kein Recht	0	1	0
Arbeitsspeicheradresse ungültig	1	1	1
virtuelle Adresse ungültig, d.h. PCS = 3	0	1	1

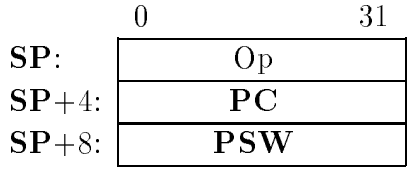
VA: virtuelle Adresse in der Seite; Zugriffswunsch darauf führte zum Alarm

- bei arithmetischem Alarm



Typ = 1	Überlauf bei ganzzahliger Arithmetik
Typ = 2	Division mit Null bei ganzzahliger Arithmetik
Typ = 3	Überlauf bei GP-Operationen
Typ = 4	Division mit Null bei GP-Operationen
Typ = 5	Unterlauf bei GP-Operationen
Typ = 6	GP-Operationen: not a number
Typ = 7	GP-Operationen: nicht mehr normalisierte Zahl

- bei **CHMK**



Op: vorzeichenrichtiger Operand von **CHMK**

Kapitel 2

Assembler-Schnittstelle

Im folgenden soll die Assemblersprache für die Maschine MI schrittweise vorgestellt werden. Ein vollständiger Abdruck der Grammatik findet sich im Anhang.

2.1 Metasprache (BNF)

Zur Beschreibung der Assemblergrammatik wird eine Metasprache verwendet, die an die Schreibweise in Backus-Naur-Form angelehnt ist. Syntaktische Variable werden wie üblich in spitze Klammern gesetzt (Beispiel: <Name>). Erläuterungen in Umgangssprache werden durch ">>" und "<<" geklammert. Zur Vereinfachung benutzen wir eine flexible Schreibweise für Wiederholungen:

Wiederholungsklammer: $\{< v >\}_t^{a,b}$

Dies bedeutet, daß die syntaktische Variable v mindestens a-mal und höchstens b-mal aufeinanderfolgt. Das Trennzeichen t steht genau einmal zwischen je 2 aufeinanderfolgenden, aus der syntaktischen Variablen <v> ableitbaren Worten. Fehlt t, so stehen die Worte dicht hintereinander. Ist $b = \infty$ so ist die obere Grenze der Zahl der Wiederholungen implementierungsabhängig festgelegt.

Gilt v ::= $< v1 > | < v2 > | \ldots | < vn >$, so darf die obige Wiederholungsklammer auch als $\{< v1 > | < v2 > | \ldots | < vn >\}_t^{a,b}$ geschrieben werden.

Anmerkung zu Zwischenräumen:
Innerhalb von Namen und Zahlen darf — wie allgemein üblich — kein Zwischenraum stehen. Namen (auch privilegierte Namen) und Zahlen müssen jedoch voneinander durch Zwischenräume getrennt werden. Um die Stellen hervorzuheben, an denen mindestens ein Zwischenraum stehen muß, wird das Zeichen "␣" verwendet.

Beispiel: **ADD** ␣ W ␣ I ␣ 3, marke

Ist zwischen zwei syntaktischen Einheiten kein Zwischenraum erlaubt, so werden sie in der Grammatik dicht hintereinander geschrieben.

Ist zwischen zwei syntaktischen Einheiten ein Zwischenraum erlaubt, aber nicht zwingend vorgeschrieben, werden sie in der Grammatik durch einen Zwischenraum getrennt geschrieben.

Statt eines Zwischenraums sind stets auch mehrere Zwischenräume zulässig.

2.2 Aufbau eines Assemblerprogramms

Zur Verdeutlichung der Beschreibung werden im folgenden jeweils die entsprechenden Stellen aus der Grammatik angeführt.

<**Assemblerprogramm**> ::= { }$^{1,\infty}$ <tanw> **END**

<**tanw**> ::= { <etanw> | <Kommentar> }$^{1,\infty}$

<**etanw**> ::= **;** | >>neue Zeile<<

<**Kommentar**> ::= – – >> beliebige Zeichenfolge ohne <etanw>-Zeichen << <etanw>

Das Assemblerprogramm besteht aus einer Folge von Segmenten, die mit **END** abgeschlossen wird. Vor dem **END** muß mindestens ein Trennzeichen für Anweisungen <tanw> stehen. <etanw> bedeutet "Elementares Trennzeichen für Anweisungen."

<**Segment**> ::= $\left\{\text{<Segmentname>}\ \textbf{:}\ \{\text{<tanw>}\}^{0,1}\right\}^{0,1}$ **SEG** { ⊔ <Ablageadresse> }^{0,1}
<tanw> $\left\{\{\text{<Marke>}\}^{0,\infty}\ \text{<Anweisung>}\ \text{<tanw>}\right\}^{0,\infty}$

<**Segmentname**> ::= <Name>

<**Ablageadresse**> ::= <vorzeichenlose ganze Zahl>

<**Marke**> ::= <Name> **:** { <tanw> }^{0,1}

Ein Segment besteht aus einer Folge von Anweisungen, die durch mindestens ein <tanw> voneinander getrennt sind. Das Ende eines Segments wird durch den Beginn eines neuen Segments oder durch das Programmende **END** erreicht.

<**Anweisung**> ::= <Maschinenbefehl>
| <Datendefinition>
| <Assemblersteuerung>

2.3 Maschinenbefehle

Maschinenbefehle haben folgenden Aufbau:

<**Maschinenbefehl**> ::= <Operationsteil>
| <Operationsteil> ⊔ { <Operandenspezifikation> }$_{,}^{1,4}$

Der Operationsteil spezifiziert die auszuführende Operation, die Kennung und die Anzahl der nachfolgenden Operanden.

Die Operandenspezifikation enthält entweder direkt den benötigten Operanden oder eine Vorschrift zur Berechnung der Adresse des Operanden im Programmadreßraum oder die Angabe eines Registers, in dem sich der Operand befindet. Die Reihenfolge der Operandenspezifikationen ist stets so gewählt, daß als letztes die Zieladresse angegeben wird.

Zur Erklärung der Wirkungsweise der Maschinenbefehle werden folgende Bezeichnungen eingeführt:

a_i:	Adresse des i-ten Operanden (ganze Zahl ohne Vorzeichen) bzw. i-te Operandenspezifikation (je nach Kontext).
$S[a_i]$:	Wert des i-ten Operanden.
$a_2 \mid a_3$	wahlweise a_2 oder a_3 möglich.
ζ:	ganze Zahl mit Vorzeichen.
δ:	beliebige Zahl.
LOP:	Anzahl der Bytes gemäß Kennung.
$\vdash$:	Ausweitung Byte zum Wort (mit Anpassung an das höchstwertigste Bit).
$\neg$:	stellenweise Negation.
$\vee$:	stellenweises Oder.
$\wedge$:	stellenweises Und.
$\otimes$:	stellenweises ausschließliches Oder.

Adressen haben Wortformat und stellen die Nummer eines Bytes dar.

2.4 Operandenspezifikation

Die Operandenspezifikation dient der Lokalisierung eines Operanden für die Befehlsausführung. Als Ergebnis liegt die Operandenadresse vor.

<**Operandenspezifikation**> ::= <absolute Adresse>
| <immediater Operand>
| <Registeradressierung>
| <relative Adressierung>
| <indirekte Adressierung>
| <indizierte relative Adressierung>
| <indizierte indirekte Adressierung>
| <Kelleradressierung>

2.4.1 Absolute Adresse

<**absolute Adresse**> ::= <Ausdruck für ganze Zahl> | { <Name> | <Importname> } { <vz> <Summand> }$^{0,\infty}$

<**Ausdruck für ganze Zahl**> ::= { <vz> }^{0,1} { <Summand> }$^{1,\infty}_{<vz>}$

Der Ort des Operanden wird direkt angegeben, entweder durch seine Adresse oder einen symbolischen Namen. Die Adressen a_i können dabei dezimal, hexadekadisch, als Bitmuster oder sogar als Zeichen mit Wortformat geschrieben werden.

Beispiele:

`MOVE H 23,25`	S[25] := S[23]
`ADD W 101,73,a3`	S[a_3] := S[73] + S[101]

2.4.2 Direkter (immediater) Operand

<**immediater Operand**> ::= **I** ⊔ <Operand>

Der Operand ist im Befehl selbst enthalten und muß der Kennung in der Befehlsbezeichnung entsprechen.

Diese Operandenspezifikation ist für Zieloperanden nicht möglich.

Schreibweise:	für a_i:	I δ
Wirkung:	statt S[a_i]:	δ
	statt a_i:	ohne Bedeutung.

Anmerkung:
Die Ersetzungen, die immer unter "Wirkung" angegeben werden, beziehen sich auf die Wirkungen der Maschinenbefehle und sind dort entsprechend zu interpretieren.

Beispiele:

`MOVE B I 2,a2`	S[a_2] := 2
`ADD W I 3,a2`	S[a_2] := S[a_2] + 3
`OR H I H'FF00',a2`	S[a_2] := S[a_2] ∨ $FF00_{16}$

2.4.3 Registeradressierung

<**Registeradressierung**> ::= <**R**x>

<**Rx**> ::= **R0** | **R1** | **R2** | **R3** | **R4** | **R5** | **R6** | **R7** | **R8** | **R9** | **R10** | **R11** | **R12** | **R13** | **R14** | **R15** | **SP** | **PC**

Schreibweise:	für a_i:	**R**x
Wirkung:	statt S[a_i]:	**R**x
	statt a_i:	unzulässig

2.4.4 Relative Adressierung

<**relative Adressierung**> ::= { <Ausdruck für ganze Zahl> + }^{0,1} ! <**R**x>

<**indizierte relative Adressierung**> ::= <relative Adressierung> <Indexangabe>

<**Indexangabe**> ::= / <**R**x> /

Der Speicherplatz des Operanden wird relativ zum Inhalt des Registers **R**x angegeben. Die ganze Zahl, nennen wir sie im folgenden ζ, wird als Displacement und das Register **R**x als vorgeschaltetes Indexregister bezeichnet. **R**y sei das bei <Indexangabe> bezeichnete Register. **R**y wird als nachgeschaltetes Indexregister bezeichnet.

Schreibweise:	für a_i:	ζ + !**R**x/**R**y/
Wirkung:	statt S[a_i]:	S[ζ + **R**x + **R**y×LOP]
	statt a_i:	ζ + **R**x + **R**y×LOP

Spezialfälle, falls $\zeta = 0$ oder /**R**y/ fehlt.
Wenn /**R**y/ fehlt, dann ist in Wirkung **R**y = 0 zu setzen.

2.4.5 Indirekte Adressierung

<**indirekte Adressierung**> ::= ! (<relative Adressierung>) | ! ! <**R**x>

<**indizierte indirekte Adressierung**> ::= <indirekte Adressierung> <Indexangabe>

Hier gelten dieselben Bemerkungen wie zur relativen Adressierung, mit dem Unterschied, daß zusätzlich indirekt adressiert wird.

Schreibweise:	für a_i:	!(ζ + !**R**x)/**R**y/
	oder wenn $\zeta = 0$:	!!**R**x/**R**y/
Wirkung:	statt S[a_i]:	S[S[ζ + **R**x] + **R**y×LOP]
	statt a_i:	S[ζ + **R**x] + **R**y×LOP

Spezialfälle, falls $\zeta = 0$ oder /**R**y/ fehlt.
Wenn /**R**y/ fehlt, dann ist in Wirkung **R**y = 0 zu setzen.

2.4.6 Kelleradressierung

<**Kelleradressierung**> ::= –!<**R**x> | !<**R**x>+

Dieser Adressierungsmodus ist vorwiegend für die Adressierung von Kellerelementen gedacht. Hierbei ist der Kellerpegel in dem angegebenen Register. Dieses Register wird entweder vor der Adreßberechnung dekrementiert oder nach Adreßberechnung inkrementiert. Die Veränderung des Kellerpegels ergibt sich aufgrund der Operandenlänge LOP gemäß Operationsteil.

Schreibweise:	für a_i:	a) -!**R**x
		b) !**R**x+
Wirkung:		
a)	statt S[a_i]:	**R**x := **R**x - LOP; S[**R**x]
	statt a_i:	**R**x := **R**x - LOP; **R**x
b)	statt S[a_i]:	S[**R**x]; **R**x := **R**x + LOP
	statt a_i:	**R**x; **R**x := **R**x + LOP

Anmerkung:
Der Pegel des grundsätzlich vom Betriebssystem eingerichteten Kellers für den Benutzer ist unter Register **R14** resp. **SP** zugänglich. Dieser Keller wächst in Richtung kleinerer Adressen. Der Pegel zeigt auf dasjenige Byte mit der niedrigsten Adresse (Anfangsadresse des letzten Elements).

2.5 Befehle der MI

2.5.1 Transportbefehle

MOVE B|H|W|F|D a_1,a_2	S[a_2] := S[a_1]	
CLEAR B|H|W|F|D a_1	S[a_1]:= 0	
MOVEN B|H|W|F|D a_1,a_2	S[a_2] := - S[a_1]	(move negated)
MOVEC B|H|W a_1,a_2	S[a_2] := ¬ S[a_1]	(move complemented)
MOVEA a_1,a_2	S[a_2] := a_1	(move address)
CONV a_1,a_2	S[a_2] := ⊢ S[a_1]	(convert (B) to (W))
PUSHR	**for** i := 14 **downto** 0 **do**	(push registers)
	SP := **SP** - 4	
	S[**SP**] := **R**i	
	endfor	
POPR	**for** i := 0 **to** 14 **do**	(pop registers)
	Ri := S[**SP**]	
	SP := **SP** + 4	
	endfor	

2.5.2 Logische Verknüpfungen

OR B|H|W a_1,a_2{,a_3}^{0,1}	S[a_2|a_3] := S[a_2] ∨ S[a_1]
ANDNOT B|H|W a_1,a_2{,a_3}^{0,1}	S[a_2|a_3] := S[a_2] ∧¬ S[a_1]
XOR B|H|W a_1,a_2{,a_3}^{0,1}	S[a_2|a_3] := S[a_2] ⊗ S[a_1]

2.5.3 Arithmetische Verknüpfungen

ADD B|H|W|F|D a_1,a_2{,a_3}^{0,1}	S[a_2|a_3] := S[a_2] + S[a_1]
SUB B|H|W|F|D a_1,a_2{,a_3}^{0,1}	S[a_2|a_3] := S[a_2] - S[a_1]
MULT B|H|W|F|D a_1,a_2{,a_3}^{0,1}	S[a_2|a_3] := S[a_2] $\times$ S[a_1]
DIV B|H|W|F|D a_1,a_2{,a_3}^{0,1}	S[a_2|a_3] := S[a_2] div S[a_1]

- Alle Operanden sind Zahlen mit Vorzeichen.
- Das Ergebnis hat dieselbe Länge wie die Operanden (nicht etwa doppelt lang bei **MULT**).
- Es gibt keine Operation, die mehrfache Genauigkeit direkt unterstützt. Mit den vorhandenen Befehlen können Prozeduren für mehrfache Genauigkeit geschrieben werden.

2.5.4 Schiebebefehle

(immer Datenformat Wort)

SH a_1,a_2,a_3	falls S[a_1] ≥ 0:	S[a_3] := Bitmuster in S[a_2] S[a_1]-mal um 1 Bit nach links geschoben Dabei wird von rechts je ein Null-bit nachgezogen Beachte: Overflow möglich
	falls S[a_1] < 0:	S[a_3] := Bitmuster in S[a_2] -S[a_1]-mal um 1 Bit nach rechts geschoben Dabei wird von links je b_0 nachgezogen
ROT a_1,a_2,a_3	falls S[a_1] ≥ 0:	S[a_3] := Bitmuster in S[a_2] S[a_1]-mal um 1 Bit zyklisch nach links geschoben Dabei wird rechts jeweils b_{31} := b_0
	falls S[a_1] < 0:	S[a_3] := Bitmuster in S[a_2] -S[a_1]-mal um 1 Bit zyklisch nach rechts geschoben Dabei wird links jeweils b_0 := b_{31}

2.5.5 Vergleichsbefehle

CMP B|H|W|F|D a_1,a_2	zero condition code := 1, falls S[a_1] = S[a_2] negative condition code := 1, falls S[a_1] $<$ S[a_2] keine Wirkung, sonst (siehe Tabelle 2.1)

2.5.6 Sprungbefehle

Bedingte Sprungbefehle

J<**Bedingung**> a_1 if <**Bedingung**> = true **then goto** a_1 **else** skip **fi**
Die Sprungbedingung bezieht sich auf das Ergebnis
RES des letzten vorangegangenen Maschinenbefehls aus
den Kapiteln 2.5.1 bis 2.5.5 (vgl. Condition Codes, Tabelle 2.1).
RES ist eine MI-interne Resultatsvariable.

Die Bezeichnungen für die Sprungbedingung <**Bedingung**> werden nachfolgend erläutert.

EQ **if** RES = 0 **then** true **fi**
NE **if** RES $\neq$ 0 **then** true **fi**
GT **if** RES $>$ 0 **then** true **fi**
GE **if** RES $\geq$ 0 **then** true **fi**
LT **if** RES $<$ 0 **then** true **fi**
LE **if** RES $\leq$ 0 **then** true **fi**
C **if** <Übertrag eines L-Bits in die fiktive Position b_{-1} > **then** true **fi**
NC **if** not <Übertrag eines L-Bits in die fiktive Position b_{-1} > **then** true **fi**
V **if** <Überlauf nach arithmetischer Operation> **then** true **fi**
NV **if** not <Überlauf nach arithmetischer Operation> **then** true **fi**

Anmerkung:
Überlauf, falls gilt: (n = Länge der Kennung)
bei Kennung B|H|W: RES $> 2^{n-1}$ -1 $\vee$ RES $<$ - 2^{n-1}
bei Kennung F|D : $e = e_{max} \wedge f = 0$ (siehe Abschnitt 1.1.3)

Unbedingter Sprungbefehl

JUMP a_1 **goto** a_1

Unterprogrammsprung

CALL a_1 co **PC** zeigt auf den Folgebefehl;
co **SP** ist der Kellerpegel (für den Benutzerkeller), **SP** = **R14**;
co Keller wächst von größeren Adressen zu kleineren;
SP := **SP** - 4
S[**SP**] := **PC**
PC := a_1
co d.h. **PC** wird auf dem Keller abgelegt und
co **SP** zeigt auf das letzte belegte Element

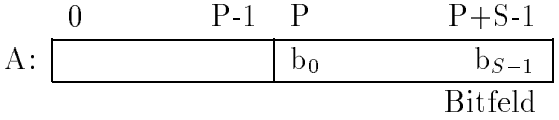
Rückkehr aus Unterprogramm

RET **PC** := S[**SP**]
SP := **SP** + 4

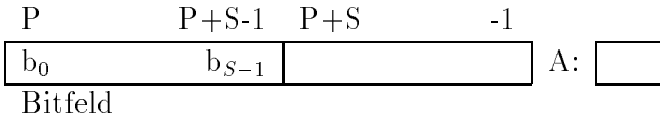
2.5.7 Bitfeldbefehle

Das Bitfeld hat eine Größe S (in Bits). Die Lage des Bitfelds ist durch eine Basisadresse A und ein Displacement P definiert. P kann positiv oder negativ sein. Es gilt $0 \leq S \leq 32$.

Falls $P \geq 0$:



Falls $P < 0$:



Anmerkung:
Im Fall $P \geq 0$ beginnt das Bitfeld bei einer Speicheradresse, die größer gleich A ist. Im Fall $P < 0$ beginnt das Bitfeld bei einer Speicheradresse, die kleiner als A ist. Das höchstwertige Bit des Bitfeldes liegt bei der ausgewählten Anfangsposition, definiert durch A+P.

Bei Registeradressierung muß zusätzlich gelten: $0 \leq P \leq 32$ und $P + S \leq 64$.
D.h. das Bitfeld muß in zwei aufeinanderfolgenden Registern enthalten sein.

EXTS a_1,a_2,a_3,a_4 P := S[a_1] (extract bitfield signed)
S = S[a_2]
A := a_3
S[a_4] := auf 32 Bit vorzeichenerweitertes Wort (integer)
gemäß dem bezeichneten Bitfeld

EXT a_1,a_2,a_3,a_4 P := S[a_1] (extract bitfield)
S = S[a_2]
A := a_3
S[a_4] := mit Nullen auf 32 Bit erweitertes Wort (integer)
gemäß dem bezeichneten Bitfeld

INS a_1,a_2,a_3,a_4 P := S[a_2] (insert bitfield)
S := S[a_3]
A := a_4
bezeichnetes Bitfeld := S[a_1]

FINDC a_1,a_2,a_3,a_4 $P := S[a_1]$ (find first bit cleared/set)
FINDS a_1,a_2,a_3,a_4 $S := S[a_2]$
$A := a_3$

$$S[a_4] := \begin{cases} \text{Nummer des ersten gefundenen} \\ \text{Bits, falls ein solches vorhanden} \\ (P \leq \text{Nummer} \leq P+S-1) \\ \\ P+S \text{ sonst} \end{cases}$$

2.5.8 Synchronisationsbefehle

JBSSI a_1,a_2,a_3 (jump on bit set and set, interlocked)
JBCCI a_1,a_2,a_3 (jump on bit cleared and clear, interlocked)

Analog wie bei Bitfeldbefehlen wird ein Bit bezeichnet durch
$P := S[a_1]$
$S := 1$ (konstant, wird in Befehl nicht angegeben)
$A := a_2$

Als unteilbare Aktion wird ausgeführt:[1]

if Befehl = **JBSSI then** tmp1 := 1 **else** tmp1 := 0 **fi**
tmp2 := bezeichnetes Bit;
bezeichnetes Bit := tmp1;
if tmp2 = tmp1 **then PC** := a_3 **fi**

2.5.9 Systemaufruf

CHMK a_1 (change mode to kernel)

Der Operand $S[a_1]$ gibt die Nummer des aufgerufenen Systemdienstes an. Der Kellerpegel zeigt auf die Position im Keller unmittelbar nach dem Vollzug von **CHMK**. Im Keller des Systemkerns wurde abgelegt:

	0 … 31
SP:	$S[a_1]$
SP+4:	**PC**
SP+8:	**PSW**

Der Aufruf des Systemkerns erfolgt über den Systemkontrollblock (Relativadresse 24). Als neuer Modus wird der Systemkernmodus eingestellt.

[1] **JBSSI** bzw. **JBCCI** können als Grundlage für die Implementierung von Semaphoren verwendet werden.

2.5.10 Rechnerkernalarm

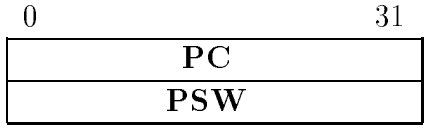
RKALARM a_1

Der Operand $S[a_1]$ gibt die Nummer desjenigen Rechnerkerns an, der unterbrochen werden soll.

Man muß die Wirkung des Befehls sowohl beim auslösenden als auch beim zu unterbrechenden Rechnerkern betrachten.

(a) Wirkung beim auslösenden Rechnerkern: keine

(b) Wirkung beim zu unterbrechenden Rechnerkern: Sobald die Unterbrechungspriorität dies zuläßt, werden im Systemkeller des Rechnerkerns (mit der Nummer $S[a_1]$) die typischen Unterbrechungsinformationen abgelegt:



Der Aufruf des Systemkerns erfolgt über den Systemkontrollblock (Relativadresse 32).

Anmerkung:
Der Befehl **RKALARM** ist nur in der 4-Rechnerkern-MI relevant und auch nur dort implementiert. Es muß $S[a_1] \in \{0, \ldots, 3\}$ gelten, da nur 4 Rechnerkerne vorhanden sind; andere Werte sind ohne Bedeutung.

Besonderheit: Jeder Rechnerkern kann sich auch selbst einen Rechnerkernalarm zustellen.

2.5.11 Privilegierte Befehle

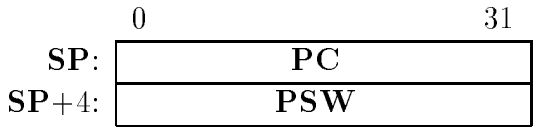
Speichern und Laden eines Prozeßkontrollblocks

SPPCB (Speichern Prozeßkontrollblock)
LPCB (Laden Prozeßkontrollblock)

Die beiden Befehle haben keinen Operanden. Die Adresse des Prozeßkontrollblocks steht im Register **PCBADR**. Beim Abspeichern des Prozeßkontrollblocks wird **PC** und **PSW** nicht aus den aktuellen Registern entnommen, sondern vom Keller des Systemkerns. Der Keller wird entsprechend abgebaut. Beim Laden des Prozeßkontrollblocks wird **PC** und **PSW** nicht in den Rechnerkern übernommen, sondern in den Keller für den Systemkernmodus abgelegt. Der Prozeß, dessen Prozeßkontrollblock geladen wurde, wird damit erst fortgesetzt, wenn der Befehl **REI** gegeben wird.

Rückkehr aus Unterbrechung

REI (return from interrupt)
Beim Aufruf zeigt **SP** auf folgende Größen:



Es wird ausgeführt:

tmp1 := S[**SP**]; **SP** := **SP** + 4 co pop **PC** in temporäre Variable;
tmp2 := S[**SP**]; **SP** := **SP** + 4 co pop **PSW** in temporäre Variable;

Befehlsalarm, falls

tmp2.CM < akt **PSW**.CM co Erhöhung der Privilegien;

oder tmp2.IPL > 0 ∧ tmp2.CM ≠ Systemkernmodus co Ablaufpriorität > 0 nur im Systemkernmodus;

oder tmp2.PM < tmp2.CM co früherer Modus kann nicht privilegierter als gegenwärtiger Modus sein;

oder tmp2.IPL > akt **PSW**.IPL co Erhöhung Ablaufpriorität;

sonst gilt: akt **PSW** := tmp2 co Neubesetzen aktuelles **PSW**;

PC := tmp1

Maschinenhalt

HALT

Besonderheit: Der **HALT**-Befehl führt bei der 4-RK-MI zum Halt aller vier Rechnerkerne.

Speichern und Laden von Sonderregistern

SP<SR> a_1 (Speichern Sonderregister)
L<SR> a_1 (Laden Sonderregister)

Die beiden Befehle haben je einen Operanden, der bei **SP**<SR> das Ziel und bei **L**<SR> die Quelle bezeichnet.

Es sind folgende Sonderregister (<SR>) möglich:

IPL	Ablaufmodus (kann als Unterbrechungssperre dienen)
MAPEN	1, falls Seitenadressierung im Systemmodus (wird in 1-Rechnerkern-MI benutzt)
RKMAP	RKMAP[b_0...b_{15}] enthält die Rechnerkernnummer RKMAP[b_{16}...b_{31}] = 1, falls Seitenadressierung im Systemmodus (wird in 4-Rechnerkern-MI benutzt)
SB	Adresse Seitentafel für Systembereich (System Base)
SL	Länge Seitentafel für Systembereich
SCBADR	Adresse des Systemkontrollblocks
PCBADR	Adresse des Prozeßkontrollblocks

2.5.12 Prozessorstatusbefehle

Der Operand a_1 benötigt die Kennung Byte (B).

SBPSW a_1 (Setzen Bits in **PSW**)	**PSW** [b_{24}...b_{31}] := **PSW** [b_{24}...b_{31}] $\vee$ S[a_1]
LBPSW a_1 (Löschen Bits in **PSW**)	**PSW** [b_{24}...b_{31}] := **PSW** [b_{24}...b_{31}] $\wedge\neg$ S[a_1]

Anmerkung:
Zeichenkettenbefehle und Befehle zur Führung von Listen sind nicht vorgesehen, da diese durch wenige Befehle realisiert werden können. Befehle zum Zugriff auf den Keller sind durch die Adressierungsart "Kelleradressierung" vorhanden. Spezielle Ein-/Ausgabebefehle sind nicht nötig, da eine "memory mapped IO" vorausgesetzt wird (siehe Kapitel 3).

2.6 Condition Codes

Folgende Condition Codes werden im **PSW** gespeichert. **C**,**Z** und **N** können durch Sprungbefehle abgefragt werden (siehe Abschnitt 2.5.6). **V** löst einen arithmetischen Alarm aus, soweit dies ein gesetztes Bit b_{26} im **PSW** (IV-Bit) bei Integer-Arithmetik nicht unterdrückt.[2]

[2]Ein gesetztes Bit b_{26} im **PSW** (IV-Bit) kann zur Programmierung mehrfacher Genauigkeit bei der Integer-Arithmetik benutzt werden. Gesetzt wird das Bit entweder durch Initialisierung im PCB oder durch den speziellen Befehl **SBPSW**, siehe Abschnitt 2.5.12.

C (carry, borrow) : wird bei Addition und Subtraktion ganzer Zahlen gesetzt, wenn ein L-Bit in die fiktive Position b_{-1} überläuft.
carry <==> (a) Addition von zwei negativen ganzen Zahlen **oder**
(b) Addition von zwei ganzen Zahlen mit unterschiedlichem Vorzeichen
und RES ≥ 0
Für Subtraktion zurückgeführt auf Addition obige Aussage analog.

V (overflow) : wird bei Überlauf gesetzt, d.h. falls
RES $> 2^{n-1}$ -1 $\vee$ RES $<$ - 2^{n-1}.
Überlauf löst eine Unterbrechung (arithmetischer Alarm) aus,
falls IV-Bit im **PSW** dies nicht unterdrückt.

Z (zero) : wird gesetzt, falls das Ergebnis einer Operation Null ist (RES = 0).

N (negative) : wird gesetzt, falls das Ergebnis einer Operation negativ ist (RES $<$ 0).

Zur Darstellung, welche Condition Codes unter welchen Bedingungen bei den einzelnen Befehlen gesetzt werden, werden folgende Bezeichnungen eingeführt:

*	wird abhängig vom Ergebnis gesetzt
-	bleibt unverändert
0	wird auf Null gesetzt
1	wird auf Eins gesetzt
BNF	bit not found
IOVL	integer overflow

Den obigen Symbolen wird in folgender Tabelle ein I bzw. F vorangesetzt, falls die entsprechene Aussage nur für Integer-Zahlen bzw. nur für Gleitpunktzahlen gilt.

Befehl	C	V	Z	N
MOVE	-	0	*	*
CLEAR	-	0	1	0
MOVEN	0	I*,F0	*	*
MOVEC / MOVEA	-	0	*	*
CONV	0	0	*	*
OR / ANDNOT/ XOR	-	0	*	*
ADD / SUB	I*,F0	*	*	*
MULT / DIV	0	*	*	*
CMP	-	-	*	*
SH	0	IOVL	*	*
ROT	-	0	*	*
JUMP	-	-	-	-
J<Bedingung>	-	-	-	-
CALL / RET	-	-	-	-
PUSHR / POPR	-	-	-	-
EXTS / EXT	0	0	*	*
INS	-	-	-	-
FINDC / FINDS	0	0	BNF	0
JBSSI / JBCCI	-	-	-	-
CHMK / RKALARM	0	0	0	0
SPPCB / LPCB	-	-	-	-
SP<SR>	-	0	*	*
L<SR>	-	0	*	*
SBPSW / LBPSW	-	-	-	-
HALT	-	-	-	-
REI	gemäß restauriertem **PSW**			

Tabelle 2.1: Condition Codes.

2.7 Datendefinition

<**Datendefinition**> ::= { **DD** ⊔ }^{0,1} <Datengruppe>

<**Datengruppe**> ::= $\left\{\{\text{<bhwfd>}\ \sqcup\}^{0,1}\ \text{<Datenelement>}\right\}_{,}^{1,\infty}$

<**Datenelement**> ::= <absolute Adresse>
| <Gleitpunktzahl>
| <String>
| (<Datengruppe>) * <vorzeichenlose ganze Zahl>

Gleitpunktzahlen können nur mit Kennung (F, D) definiert werden. Bei ganzen Zahlen darf die Kennung fehlen; es wird dann das kleinstmögliche Format (für die Zahl mit Vorzeichen) verwendet. Bei Marken wird die entsprechende Adresse als Wort abgelegt, die Angabe eines Datentyps ist also irrelevant. Das Gleiche gilt für Strings: hier wird für jedes Zeichen ein Byte (ASCII-Code, siehe Anhang) abgelegt. Strings können eine maximale Länge von 64 Zeichen haben. Der angegebene Datentyp gilt jeweils für alle folgenden Angaben innerhalb einer Datengruppe bis zur erneuten Einstellung eines Datentyps.

Beispiel 1:
```
DD    (12, H 17)*2, 35
        |     |      |
      Byte Halbwort Byte
erzeugter Code:
      0C 0011 0C 0011 23 (hexadekadisch)
```

Beispiel 2:
```
DD    'Hans',marke
        |     |
      4 Byte Adresse (Wort)
```

Beispiel 3:
```
DD H  ( B 7,13 ) * 2,100
         |    |        |
       Byte Byte Halbwort
erzeugter Code:
      07 0D 07 0D 0064 (hexadekadisch)
```

2.8 Befehle zur Assemblersteuerung

<**Assemblersteuerung**> ::= <Importanweisung>
| <Exportanweisung>
| <Gleichsetzung>
| <Reservierung>
| <Ausrichtung>

<**Importanweisung**> ::= { **IMP** | **IMPORT** } ⊔ $\left\{\textbf{ALL}\ |\ \{\text{<Importname>}\}_{,}^{1,\infty}\right\}$

<**Importname**> ::= <Segmentname>.<Name> | <Name>

<**Segmentname**> ::= <Name>

Über die Importanweisung werden Markennamen eines anderen Segmentes zugänglich. Bei Eindeutigkeit genügt der Name allein, sonst muß der Segmentname zugefügt werden. Sollen alle von anderen Segmenten exportierten Markennamen zugänglich sein, so genügt die Anweisung **IMP ALL** oder **IMPORT ALL**.

<**Exportanweisung**> ::= { **EXP** | **EXPORT** } ⊔ $\left\{\textbf{ALL} \mid \{\text{<Name>}\}_{,}^{1,\infty}\right\}$

Über die Exportanweisung werden im Segment definierte Markennamen nach außen zugänglich gemacht. Sollen alle Namen zugänglich sein, so genügt **EXP ALL** oder **EXPORT ALL**.

<**Gleichsetzung**> ::= { **EQU** | **EQUAL** } ⊔ { <Name> = <Ersetzungstext> }$_{,}^{1,\infty}$

<**Ersetzungstext**> ::= >> beliebige Zeichenfolge ohne <tanw>-Zeichen und Komma <<

Der bezeichnete Name (Makroname) wird dem Ersetzungstext zugeordnet. Bei jedem Auftreten des Makronamens wird sein Ersetzungtext eingesetzt. Es muß darauf geachtet werden, daß die Ersetzung korrekt in den Kontext paßt.

Der Ersetzungstext kann an folgenden Stellen stehen:

- <Name> Name einer Marke
- <Importname>
- <ganze Zahl>
- <Gleitpunktzahl>
- <String>
- <Marke>
- <Registerangabe>
- privilegierte Namen (in der Sprachbeschreibung in **BLOCKSCHRIFT**), z.B. **ADD**, **EXPORT**, **I**, **D**
- die Sonderzeichen * / ! ( ) + - (aber nicht das Kommentarzeichen "– –")

<**Reservierung**> ::= { **RES** | **RESERVE** } ⊔ <vorzeichenlose ganze Zahl>

Es werden <vorzeichenlose ganze Zahl> Bytes eingefügt, deren Inhalt 0 ist.

<**Ausrichtung**> ::= { **ALI** | **ALIGN** } ␣ <vorzeichenlose ganze Zahl>

Es werden n ≥ 0 undefinierte Bytes eingefügt, so daß das nächste Byte einer durch $2^{\text{<vorzeichenlose ganze Zahl>}}$ teilbaren Speicheradresse zugeordnet wird.

Formal:
Sei a eine interne Variable des Übersetzers, die angibt, bei welcher Speicheradresse das nächste vom Assembler erzeugte Byte abzulegen ist. Sei a_v bzw. a_n der Inhalt dieser Variablen vor bzw. nach der Ausrichtung. Dann gilt: $a_n := a_v + n$

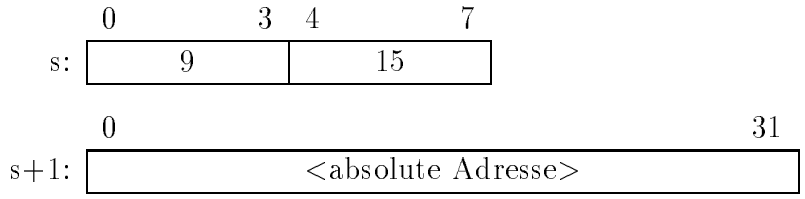
Hierbei ist $0 < n \leq 2^{\text{<vorzeichenlose ganze Zahl>}}$
sowie $a_n \bmod 2^{\text{<vorzeichenlose ganze Zahl>}} = 0$.

Dies bedeutet, daß die letzten <vorzeichenlose ganze Zahl> Binärstellen von a_n mit 0 besetzt sind.

2.9 Codierung der Operandenspezifikationen

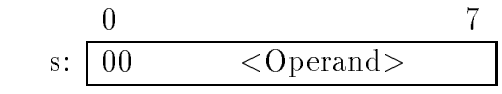
Die Darstellung ist kompatibel mit der VAX-Architektur. s ist der Beginn einer Operandenspezifikation.

2.9.1 Absolute Adresse

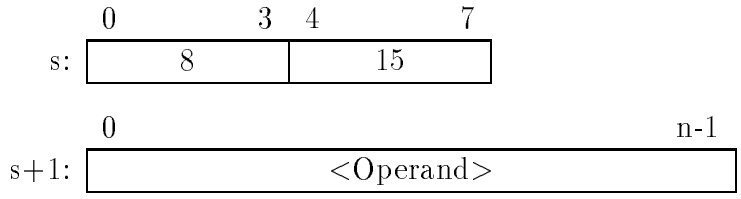


2.9.2 Darstellung direkter Operanden

Falls <Operand> die Kennung B, H oder W hat und 0 ≤ <Operand> ≤ 63:



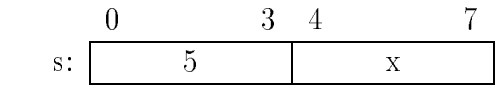
sonst:



Die Länge n des Operanden ergibt sich aus dem Befehlscode.

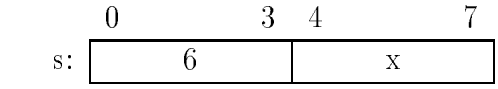
2.9.3 Registeradressierung

bei Adressierung mit Register Rx:

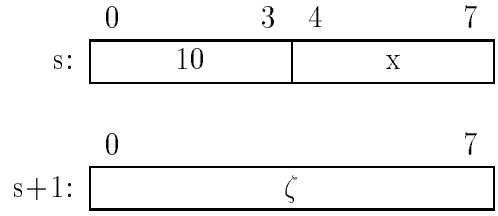


2.9.4 Relative Adressierung

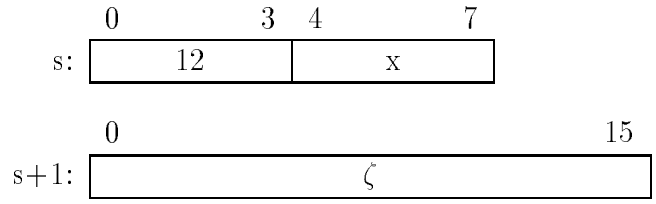
Falls $\zeta = 0$:



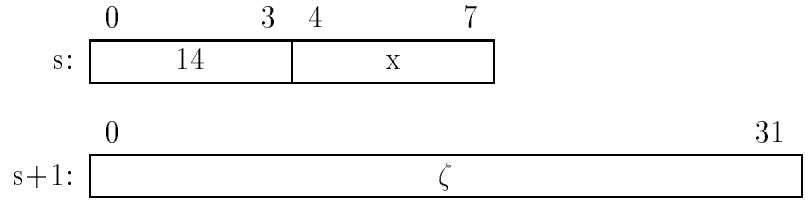
Falls ζ als ganze Zahl mit Vorzeichen im Byte darstellbar:



Falls ζ als ganze Zahl mit Vorzeichen im Halbwort darstellbar:



Falls ζ als ganze Zahl mit Vorzeichen ein Wort benötigt:



Bemerkung:
Enthält die Definition der *absoluten Adresse* auf Assemblerebene eine *Marke*, so wird relativ zum **PC** codiert; dabei steht der **PC** bei der Adreßberechnung unmittelbar auf dem Anfang des Displacements ζ und zeigt dabei auf s+1 (in folgendem Beispiel mit "|" gekennzeichnet). Die Adressen werden alle hexadekadisch angegeben.

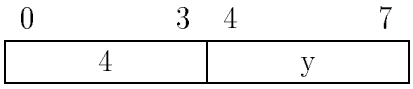
Beispiel:

```
Assemblertext :                    Ablageadresse:  erzeugter Code:
                .                  .               .
                .                  .               .
                ADD H I 3, marke + 7  00000100     C0 03 AF0E
                                                          |
                .                  .               .
                .                  .               .
marke :         DD W 0             0000010A        00000000
                .                  .               .
                .                  .               .
```

Bei Auswertung von "marke + 7" steht der **PC** bereits auf Adresse 00000103. Die Differenz zur Zieladresse (0000010A+00000007 = 00000111) beträgt damit nur noch 0000000E

2.9.5 Indizierte relative Adressierung

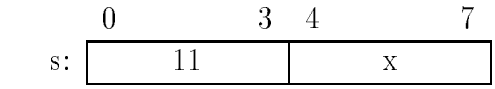
Sei **R**y das Register, das zur Indizierung verwendet wird. Die Operandenspezifikation ist wie unter Abschnitt 2.9.4 beschrieben, allerdings geführt von einem Byte mit folgendem Aufbau:

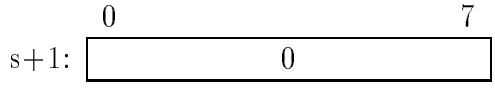


2.9.6 Indirekte Adressierung

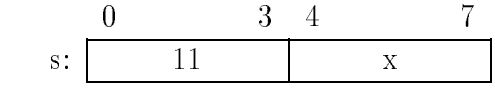
Analog wie bei Abschnitt 2.9.4 werden wieder 4 Fälle je nach Größe des Speicherbereichs zur Darstellung der Relativadresse ζ unterschieden.

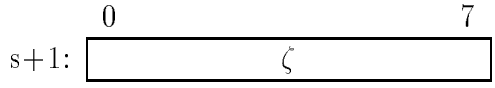
Falls $\zeta = 0$:



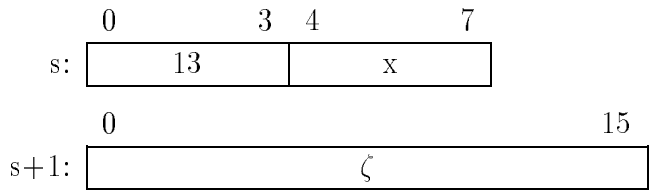


Falls ζ als ganze Zahl mit Vorzeichen im Byte darstellbar:

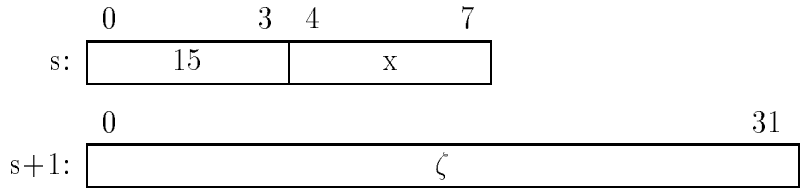




Falls ζ als ganze Zahl im Halbwort darstellbar:



Falls ζ als ganze Zahl mit Vorzeichen ein Wort benötigt:

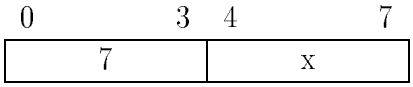


2.9.7 Indizierte indirekte Adressierung

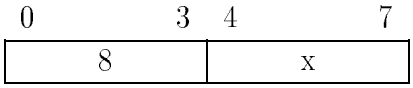
Der erste Teil der Operandenspezifikation ist ein Byte wie unter Abschnitt 2.9.5 beschrieben. Der zweite Teil der Operandenspezifikation ist wie unter Abschnitt 2.9.6 beschrieben.

2.9.8 Kelleradressierung

Falls -!Rx:



Falls !Rx+:

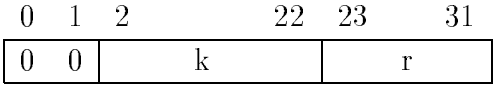


Kapitel 3

Memory mapped I/O

3.1 Maschinenadreßraum

Gemäß Abschnitt 1.5 haben reale Maschinenadressen folgenden Aufbau:



Dieser Bereich der Maschinenadressen wird nun in zwei Teile unterteilt:

- $b_2 = 0$: Bereich der Arbeitsspeicheradressen.
- $b_2 = 1$: Bereich der Adressen von Registern der EA-Prozessoren.

Die normalen Befehle des Rechnerkerns (bzw. der Rechnerkerne bei der 4-Rechnerkern-MI) können damit sowohl zur Adressierung des Arbeitsspeichers als auch zur Adressierung der EAP-Register verwendet werden.

Die nachfolgende Abbildung zeigt den Aufbau des Maschinenadreßraums.

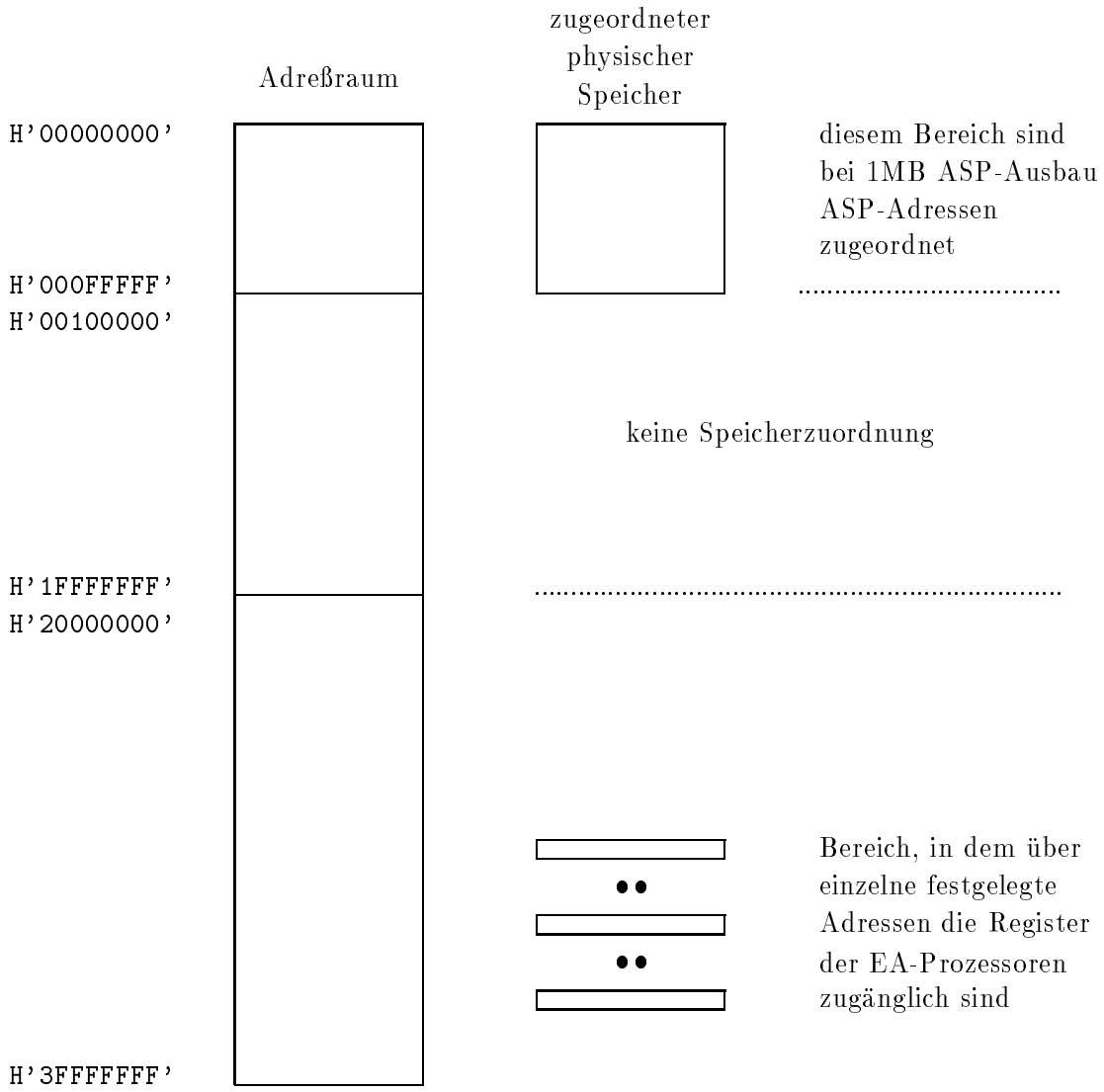


Tabelle 3.1: Aufbau des Maschinenadreßraums.

3.2 Plattenspeicher

3.2.1 Überblick

An die Anlage sind zwei Plattenspeicherlaufwerke mit den Nummern 0 und 1 angeschlossen. Diese Plattenspeicher werden von einem EA-Prozessor (EAP1, Plattenspeichersteuerung) gesteuert. Ein Plattenspeicherlaufwerk hat folgende technische Eigenschaften:

- 512 Zylinder je Laufwerk
- 2 Spuren je Zylinder
- 40 Sektoren je Spur
- 256 Bytes je Sektor

Die Arbeitsweise der Plattenspeichersteuerung wird durch geeignete Besetzung der folgenden 5 Registern gesteuert:

Name	Länge	Adresse
STR (Steuerregister)	16 Bit	H'203FF900'
SARL (Speicheradreßregister low)	16 Bit	H'203FF902'
SARH (Speicheradreßregister high)	16 Bit	H'203FF904'
PAR (Plattenspeicheradreßregister)	16 Bit	H'203FF906'
MZR (Mehrzweckregister)	16 Bit	H'203FF908'

Die relative Adresse bzgl. des Systemkontrollblocks SCB für Unterbrechungen ist 40. Einzelne Binärzeichen der Register sind eventuell nur lesbar (R) oder nur beschreibbar (W) oder sowohl zu lesen wie auch zu schreiben (RW). Wird in ein Register geschrieben, so bleiben die nur lesbaren Binärzeichen des Registers unverändert. Wird ein Register gelesen, so wird anstelle der nur schreibbaren Binärzeichen eine 0 zurückgeliefert.

3.2.2 Funktionen

Die Plattenspeichersteuerung führt 5 Funktionen (Kommandos) aus:

1. Arm positionieren
2. Daten schreiben
3. Daten lesen
4. Prüflesen
5. Sektorkopf lesen

Arm positionieren: Die Einstellung der Lese-/Schreibköpfe auf eine neue Spur erfolgt durch das Kommando "Arm positionieren." Durch Lese-/Schreibaufträge erfolgt keine neue Positionierung.

Daten schreiben/lesen: Die Daten werden zwischen einem Puffer im Arbeitsspeicher und einem bzw. mehreren aufeinanderfolgenden Sektoren transportiert. Wird ein Sektor nicht voll beschrieben, so wird der Rest mit 0 aufgefüllt.

Prüflesen: Die Daten in einem Puffer des Arbeitsspeichers werden mit den Daten auf dem Plattenspeicher verglichen.

Sektorkopf lesen: Vor jedem Sektor auf dem Plattenspeicher steht ein Sektorkopf. Dieser enthält die Spurnummer und die Sektornummer.

Im MI-Simulator wird folgendes realisiert: Es wird der Inhalt des nächsten bei den Leseköpfen ankommenden Sektors in das Mehrzweckregister der Plattenspeichersteuerung übertragen und die eingestellte Spur übergeben.

3.2.3 Steuerregister STR

Das Steuerregister enthält u.a. das Startsignal für die Plattenspeichersteuerung und muß daher als letztes der 5 Register verändert werden. Die Bedeutung der einzelnen Binärzeichen ist:

$b_1b_2b_3b_4$ (R): Fehlercode der Plattenspeichersteuerung

0: kein Fehler bei Ausführung des Kommandos

1: Lesefehler in Daten

2: Lesefehler in Sektorkopf

3: Sektorkopf nicht gefunden

4: Speicherzelle im Arbeitsspeicher nicht vorhanden

5: unbekanntes Kommando

6: falsche Laufwerkangabe

7: Arm beim Lesen oder Schreiben falsch positioniert

8: Daten haben auf der angewählten Spur keinen Platz / Sektornummer zu groß

9: falsche Distanzangabe |neue Spurnr - aktuelle Spurnr| (beim Positionieren des Arms)

Anmerkung: Vor Ausführung eines neuen Kommandos werden alle Fehlerbits gelöscht.

b_5b_6 (RW): Nummer des RK, der unterbrochen werden soll, falls b_{11} gesetzt. Diese beiden Bits sind nur in der 4-Rechnerkern-MI relevant; in der 1-Rechnerkern-MI sind sie mit 0 zu besetzen.

b_8b_9 (RW): Nummer des auszuwählenden Laufwerks

b_{10} (RW): Falls gesetzt: Plattenspeichersteuerung hat Kommando beendet.
Falls gelöscht: Plattenspeichersteuerung soll Kommando ausführen.

b_{11} (RW): Die Plattenspeichersteuerung soll nach Ausführung eines Kommandos eine Unterbrechung auslösen.

$b_{12}b_{13}b_{14}$ (RW): Wahl der Funktion mit folgender Bedeutung:

1: Arm positionieren

2: Daten schreiben

3: Daten lesen

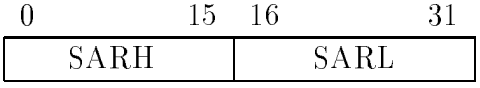
4: Prüflesen

5: Sektorkopf lesen

b_{15} (R): Das ausgewählte Laufwerk ist bereit, ein Kommando zu empfangen oder Daten zu übermitteln.

3.2.4 Speicheradreßregister SAR

Das Speicheradreßregister besteht aus 2 Teilen, die getrennt besetzt werden müssen:



Das Speicheradreßregister enthält eine Adresse des realen Arbeitsspeichers (max. 24 signifikante Bits), ab der sich der Pufferbereich für die zu übertragenden Daten befindet.

3.2.5 Plattenspeicheradreßregister PAR

Der Inhalt des PAR ist abhängig von der auszuführenden Funktion.

bei Prüflesen, Daten schreiben, Daten lesen:

$b_0 \ldots b_8$ (RW): Spurnummer $\in \{0, \ldots, 511\}$

b_9 (RW): 0 obere Oberfläche der Platte; 1 untere Oberfläche der Platte

$b_{10} \ldots b_{15}$ (RW): Sektornummer $\in \{0, \ldots, 39\}$

Nach Übertragung eines Sektors wird die Sektornummer um 1 erhöht.

bei Sektorkopf lesen: PAR wird nicht verwendet

bei Arm positionieren:

$b_0 \ldots b_8$ (RW): Anzahl der Spuren, die der Arm bewegt werden soll, d.h.
| neue Spurnr - aktuelle Spurnr |

b_{15} (RW): Bewegungsrichtung:

0 Richtung größerer Spurnummern

1 Richtung kleinerer Spurnummern

3.2.6 Mehrzweckregister MZR

Die Bedeutung des MZR ist ebenfalls abhängig von der ausgewählten Funktion. MZR ist lesbar und schreibbar.

bei Prüflesen, Daten schreiben, Daten lesen: Sollen n Zeichen übertragen werden, so wird in MZR der Wert $-n$ geschrieben. Nach Transport eines Zeichens wird MZR um 1 erhöht. Die Anzahl der Zeichen ist so zu wählen, daß die zu übertragenden Daten innerhalb der angewählten Spur Platz haben (bzw. auf der Spur stehen).

bei Sektorkopf lesen: Aufbau wie bei PAR; Werte bzgl. des gelesenen Sektorkopfes.

bei Arm positionieren: ohne Bedeutung.

3.3 Asynchrone Datenübertragung

3.3.1 Übersicht über die Register

Der EA-Prozessor 2 realisiert die EA-Steuerung für 4 asynchrone Datenübertragungskanäle. Jeder Datenübertragungskanal besteht aus einem Eingabekanal und einem Ausgabekanal. Beide sind, falls das angeschlossene Gerät dies erlaubt, gleichzeitig betreibbar. Der Ablauf des EA-Vorgangs wird wieder durch Register, die geeignet besetzt werden müssen, gesteuert. Diese sind:

- ESTR_i: das Steuerregister für den Empfang von Zeichen über den Datenkanal i.
- EDR_i: das Datenregister für den Empfang von Zeichen über den Datenkanal i.
- SSTR_i: das Steuerregister für das Senden von Zeichen über den Datenkanal i.
- SDR_i: das Datenregister für das Senden von Zeichen über den Datenkanal i.

Die Adressen dieser 16-Bit-Register sind:
H'203FFA00' $+8 \times i$ für ESTR_i
H'203FFA02' $+8 \times i$ für EDR_i
H'203FFA04' $+8 \times i$ für SSTR_i
H'203FFA06' $+8 \times i$ für SDR_i
Der Aufbau der Register wird nachfolgend diskutiert.

3.3.2 Empfangssteuerregister ESTR

b_3 **(RW):** Parität ist ungerade, falls gesetzt.

b_4 **(RW):** Paritätsbit ist vorhanden, falls gesetzt.

1. Bit wird als Paritätsbit interpretiert

$b_5 b_6$ **(RW):** Nummer des RK, der unterbrochen werden soll, falls b_9 gesetzt. Diese beiden Bits sind nur in der 4-Rechnerkern-MI relevant; in der 1-Rechnerkern-MI sind sie mit 0 zu besetzen.

b_7 **(RW):** Es werden 8-Bit-Zeichen übertragen, falls $b_7 = 1$

Es werden 7-Bit-Zeichen übertragen, falls $b_7 = 0$

Falls $b_4 = b_7 = 1$, so überwiegt die Wirkung von b_4.

b_8 **(R):** In EDR ist ein Zeichen abholbereit (b_8 wird gelöscht, falls EDR gelesen wird)

b_9 **(RW):** Unterbrechung gewünscht, falls Zeichen in EDR bereit.

b_{10} **(RW):** Zwei Stoppschritte (falls $b_{10} = 0$: 1 Stoppschritt).

$b_{11} \ldots b_{14}$ **(RW):** Schrittgeschwindigkeit:

Hat $b_{11} \ldots b_{14}$ den Wert n, so ist die eingestellte Schrittgeschwindigkeit: 150×2^n baud. Es gilt $0 \leq n \leq 7$.

b_{15} **(RW):** Der Kanal ist empfangsbereit gesetzt.

3.3.3 Empfangsdatenregister EDR

Das Register kann nur gelesen werden. Es enthält rechtsbündig das gelesene Zeichen. Falls weniger als 8 Binärziffern gelesen werden, wird das Zeichen links mit Null zum Byte aufgefüllt. Falls die Zeichen schneller übertragen als gelesen werden, werden sie überschrieben.

Das linke Byte enthält Fehlermeldungen: $b_0 = b_1 \vee b_2 \vee b_3$

b_1**:** Mindestens ein Zeichen ging verloren, da EDR nicht genügend schnell gelesen wurde.

b_2**:** Beim Lesen des Zeichens folgte nach dem Zeichen kein Stoppschritt.

b_3**:** Beim Lesen des Zeichens in EDR trat ein Paritätsfehler auf.

Die Fehlermeldungen werden unmittelbar nach dem Lesen von EDR gelöscht.

3.3.4 Sendesteuerregister SSTR

b_3 **(RW):** Parität ist ungerade, falls gesetzt.

b_4 **(RW):** Paritätsbit ist vorhanden, falls gesetzt.

1. Bit wird als Paritätsbit interpretiert.

$b_5 b_6$ **(RW):** Nummer des RK, der unterbrochen werden soll, falls b_9 gesetzt. Diese beiden Bits sind nur in der 4-Rechnerkern-MI relevant; in der 1-Rechnerkern-MI sind sie mit 0 zu besetzen.

b_7 **(RW):** Es werden 8-Bit-Zeichen übertragen, falls $b_7 = 1$

Es werden 7-Bit-Zeichen übertragen, falls $b_7 = 0$

Falls $b_4 = b_7 = 1$, so überwiegt die Wirkung von b_4.

b_8 **(R):** Zeichen in SDR wurde übertragen (b_8 wird gelöscht, falls SDR beschrieben wird).

b_9 **(RW):** Unterbrechung gewünscht, falls Zeichen aus SDR übertragen ist, d.h. ein neues Zeichen in SDR zu schreiben ist.

b_{10} **(RW):** Zwei Stoppschritte (falls $b_{10} = 0$: 1 Stoppschritt).

$b_{11} \ldots b_{14}$ **(RW):** Schrittgeschwindigkeit (wie bei ESTR).

b_{15} **(RW):** Der Kanal ist sendebereit gesetzt.

3.3.5 Sendedatenregister SDR

Ist SSTR[b_7] = 1, so sind $b_0 \ldots b_7$ ohne Bedeutung, und $b_8 \ldots b_{15}$ enthalten das zu übertragende Zeichen.
Ist SSTR[b_7] = 0, so sind $b_0 \ldots b_8$ ohne Bedeutung, und $b_9 \ldots b_{15}$ enthalten das zu übertragende Zeichen.

3.4 Anmerkungen zu den Geräten

3.4.1 Terminal

Das Terminal ist an den Kanal 3 des EA-Prozessors 2 angeschlossen und arbeitet mit einer Schrittgeschwindigkeit von 9600 baud. Das Terminal der hypothetischen Maschine wird dabei durch das Terminal des Host-Rechners nachgebildet. Es können ASCII-Zeichen von der Tastatur eingelesen werden und am Bildschirm ausgegeben werden (siehe ASCII-Tabelle im Anhang).

3.4.2 Drucker

Der Drucker ist an den Kanal 2 des EA-Prozessors 2 angeschlossen und arbeitet mit einer Schrittgeschwindigkeit von 1200 baud. Der Drucker überträgt ASCII-Zeichen, die in die Datei "druckaus" ausgegeben werden. Diese Datei kann nach Beendigung des Simulationslaufs ausgedruckt werden.

Eine Eingabe über Kanal 2 ist nicht vorgesehen.

3.4.3 Zeitliches Verhalten

Da der Rechnerkern (bzw. die Rechnerkerne bei der 4-Rechnerkern-MI) und die EA-Prozessoren EAP1 und EAP2 parallel arbeiten, muß das zeitliche Verhalten aufeinander abgestimmt sein; es wurden folgende realistische Annahmen gemacht:

- Die mittlere Dauer eines Maschinenbefehls beträgt 3 Mikrosekunden.
- Die Arm-Positionierungs-Zeit des Plattenspeichers beträgt 100 ms.

  Die Umdrehungszeit beträgt 30 ms.
- Der Drucker arbeitet mit einer Schrittgeschwindigkeit von 1200 baud, das Terminal mit 9600 baud.

Durch diese Festlegungen ist es möglich, die Dauer eines EA-Auftrags zu bestimmen und dem gewählten Rechnerkern eine ggf. gewünschte Unterbrechung zum richtigen Zeitpunkt zuzustellen. Da die Durchführung eines EA-Auftrags wesentlich mehr Zeit als die Ausführung eines Maschinenbefehls beansprucht, wird man im Übungsbetrieb auf die Beendigung eines EA-Auftrags häufig durch Aktivieren eines "Null-Prozesses" warten. Der Simulator wurde so implementiert, daß der "Null-Prozeß" keine Rechenzeit des Host-Rechners verbraucht, falls dieser nur aus einer leeren unendlichen Schleife (Sprung auf sich selbst) besteht, sogenanntes *busy waiting*.

Kapitel 4

Assembler

4.1 Bedienung des Assemblers

Der Assembler erzeugt ein fixiertes, von der Maschine MI ausführbares Ladeobjekt. Er vereinigt also die Aufgaben "assemblieren," "binden" und "fixieren." Der Grund für diese Festlegung ist, daß die zu assemblierenden Studentenprogramme relativ klein sind, in allen Teilen häufig verändert und nach einer Assemblierung auch zum Ablauf gebracht werden. Die Integration von Assembler, Binder und Fixierer ist unter diesen Randbedingungen besonders effizient.

Um mehrere Programmstücke an unterschiedliche Stellen des Arbeitsspeichers bequem laden zu können, besteht das entstehende Ladeobjekt ggf. aus mehreren Teilobjekten. Jedem Teilobjekt ist eine Arbeitsspeicheranfangsadresse (Ablageadresse) zugeordnet, ab der das Teilobjekt zusammenhängend geladen wird.

Ein Ablageblock besteht aus genau einem Segment. Einem Segment kann eine Ablageadresse zugeordnet werden, dann definiert dieses Segment den Anfang eines Ladeteilobjekts. Alle nachfolgenden Segmente, die keine Ablageadressen zugeordnet haben, werden als Ablageblöcke dieses Ladeteilobjekts betrachtet.

Anmerkung: Namenlose Segmente werden intern durchnumeriert. Diese Nummer erscheint auch in den Ausgabedateien statt eines Segmentnamens (z.B. "– – 1 – –" für das erste namenlose Segment des Programms).

Der Aufruf des Assemblers erfolgt mit dem Kommando

assembliere-MI <Eingabe> <Listing> <Code>.

Dabei ist <Eingabe> die Eingabedatei mit dem zu übersetzenden Programm, <Listing> die Ausgabedatei für das Protokoll und <Code> die Ausgabedatei für den erzeugten Code. Es werden zwei weitere Ausgabedateien im aktuellen Directory angelegt: Eine Cross-Referenz-Liste für Makros und Marken und ein Adreßbuch für die Zuordnung von Marken zu Speicheradressen

4.2 Dateien des Assemblers

4.2.1 Eingabedatei

Die Eingabedatei enthält das zu übersetzende Programm, das evtl. aus mehreren Segmenten besteht. Die erste Anweisung muß eine Segmentanweisung (**SEG**) sein (eventuell mit Segmentname und/oder Ablageadresse). Vor dieser Anweisung dürfen beliebig viele Leerzeilen, Strichpunkte und Kommentare stehen. Die letzte Anweisung des Programms ist **END**. Die vollständige Grammatik findet sich im Anhang.

Beispiel: Das nachfolgende MI-Programm bestimmt das Minimum von a0, a1, a2, a3, a4 und a5 und legt es in Register R4 ab.

```
TEST:   SEG
        MOVE   W I H'10000',SP  -- nuetzliche Vorbesetzung
        JUMP   start
a0:     DD     W 3
a1:     DD     W -1
a2:     DD     W 9
a3:     DD     W 5
a4:     DD     W 0
a5:     DD     W 2
start:  MOVEA  a0,R0           -- Anfangsadresse
        MOVE   W I 0,R2        -- i in R2
        MOVE   W !R0/R2/,R1    -- erster Wert a[0]
init:   MOVE   W I 1,R2        -- Startwert
        MOVE   W I 5,R3        -- Endwert
        JUMP   test
loop:
if:     SUB    W R1,!R0/R2/,R4 -- Bedingung akt-min < 0
        JGE    fi
        MOVE   W !R0/R2/,R1    -- entspricht min:=akt
fi:
step:   ADD    W I 1,R2        -- erhoehe Laufindex
test:   SUB    W R3,R2,R4
        JLE    loop
        MOVE   W R1,R4         -- Ergebnis ist in R4
        HALT
        END
```

4.2.2 Protokoll (Listing)

Das Listing ist spaltenweise aufgebaut. Im fehlerfreien Fall (lauffähiges Programm eventuell mit Warnungen) sieht das Listing folgendermaßen aus:

1. Spalte: Zeilennummer der Eingabedatei.

2. Spalte: definierte Marken.

3. Spalte: Befehlszählerstand (hexadekadisch, zählt ab H'00000000' durch).

4. Spalte: für Maschinenbefehle den entsprechenden Code.

- für Reservierungen (**RES**): V(00).
- für Datendefinition mit Multiplikator: V(<einfacher Code>) (V steht für Vorbelegung).

5. Spalte: Quelltext (maximal 60 Zeichen, der Rest wird unterdrückt).

6. Spalte: Kommentar (maximal 20 Zeichen, der Rest wird unterdrückt).

Ist das übersetzte Programm fehlerhaft, so fehlen die Spalten 3 und 4.

Fehler werden mit "|" an entsprechender Stelle im Quelltext markiert. Bei Fehlern im Zusammenhang mit Namen (Marken/Makros) wird dieser Name (bzw. die ersten 10 Zeichen) in die Fehlermeldung mit aufgenommen. Die Fehlermeldungen werden nach vollständiger Ausgabe einer Anweisung ausgegeben bzw. am Ende eines Segmentes oder am Programmende.

Beispiel (ohne Kommentarzeilen):

```
Zeile  Marke   Adresse   Code              Text
    1  TEST:                              SEG
    2          00000000  A0 8F00010000 5E  MOVE W I H'10000',SP
    3          00000007  F1 AF19           JUMP start
    4  a0:     0000000A  00000003          DD W 3
    5  a1:     0000000E  FFFFFFFF          DD W -1
    6  a2:     00000012  00000009          DD W 9
    7  a3:     00000016  00000005          DD W 5
    8  a4:     0000001A  00000000          DD W 0
    9  a5:     0000001E  00000002          DD W 2
   10  start:  00000022  AB AFE6 50        MOVEA a0,R0
   11          00000026  A0 00 52          MOVE W I 0,R2
   12          00000029  A0 4260 51        MOVE W !R0/R2/,R1
```

```
13  init:  0000002D  A0 01 52       MOVE W I 1,R2
14         00000030  A0 05 53       MOVE W I 5,R3
15         00000033  F1 AF10        JUMP test
16  loop:
17  if:    00000036  D0 51 4260 54  SUB W R1,!R0/R2/,R4
18         0000003B  EC AF05        JGE fi
19         0000003E  A0 4260 51     MOVE W !R0/R2/,R1
20  fi:
21  step:  00000042  C1 01 52       ADD W I 1,R2
22  test:  00000045  D0 53 52 54    SUB W R3,R2,R4
23         00000049  EE AFEB        JLE loop
24         0000004C  A0 51 54       MOVE W R1,R4
25         0000004F  00             HALT
26                                  END
```

4.2.3 Code

Der Code wird in hexadekadischen Ziffern ausgegeben, so wie ihn die Maschine verarbeiten kann. Zur besseren Lesbarkeit sind Leerzeichen und Zeilenwechsel eingestreut (für die Maschine ohne Bedeutung), wie folgt:

Operationsteil und Operandenspezifikationen sind voneinander durch Leerzeichen getrennt, verschiedene Befehle durch Zeilenwechsel. Die Kopfzeile für ein Ladeteilobjekt (= Segment mit Namen) sieht folgendermaßen aus:

$$a,\ n,\ \text{<Segmentname> <NL>}$$

wobei a die Ablageadresse (8 Hexaden) bezeichnet und n die Zahl der Zeichen im Segmentnamen (2 Hexaden). Jedes Ladeobjekt wird mit der Zeile abgeschlossen, die nur einen einzelnen Punkt in der ersten Spalte enthält.

Beispiel:

```
00000000,04,TEST
A0 8F00010000 5E
F1 AF19
00000003
FFFFFFFF
00000009
00000005
00000000
00000002
AB AFE6 50
A0 00 52
A0 4260 51
```

```
A0 01 52
A0 05 53
F1 AF10
D0 51 4260 54
EC AF05
A0 4260 51
C1 01 52
D0 53 52 54
EE AFEB
A0 51 54
00
.
```

4.2.4 Cross-Referenz-Liste

Es wird für Makros und Marken jeweils eine alphabetisch sortierte Liste mit den Zeilennummern der Definitions- und der Verwendungsstellen ausgegeben.

Beispiel:

```
CROSS - REFERENCES : MAKROS (alphabetisch sortiert)

CROSS - REFERENCES : MARKEN (alphabetisch sortiert)
    Marke  Segment  Zeilennummern
    TEST                     1
    a0     TEST             4 10
    a1     TEST                5
    a2     TEST                6
    a3     TEST                7
    a4     TEST                8
    a5     TEST                9
    fi     TEST            20 18
    if     TEST               17
    init   TEST               13
    loop   TEST            16 23
    start  TEST             10 3
    step   TEST               21
    test   TEST            22 15
```

4.2.5 Adreßbuch

Diese Ausgabedatei enthält zu jeder Marke die entsprechende Adresse in hexadekadischen Ziffern. Sie wird nur dann erzeugt, wenn das Programm fehlerfrei übersetzt wurde.

Beispiel:

```
Marke  Segment  Adresse (in Hexaziffern)
TEST            00000000
a0     TEST     0000000A
a1     TEST     0000000E
a2     TEST     00000012
a3     TEST     00000016
a4     TEST     0000001A
a5     TEST     0000001E
fi     TEST     00000042
if     TEST     00000036
init   TEST     0000002D
loop   TEST     00000036
start  TEST     00000022
step   TEST     00000042
test   TEST     00000045
```

4.3 Fehlerbehandlung

Bei Auftreten von Fehlern, die vom Assembler nicht selbst korrigiert werden können, wird an der nächsten sinnvollen Stelle im Programm aufgesetzt:

- Fehler im Operationsteil einer Anweisung: Überlesen der gesamten Anweisung.
- Fehler in einem Operanden: Überlesen dieses Operanden.
- Fehler in einer Markendefinition: Nach Möglichkeit Aufsetzen vor der nächsten Marke bzw. Anweisung.

Kapitel 5

MI-Simulator und Testhilfe

Der Benutzer kann den eigentlichen Simulator durch das Kommando

starte-MI ⊔ <Code>

starten. Die Angabe eines Dateinamens nach starte-MI ist optional. Voreinstellung ist "ladobj." Beim Ladevorgang wird der in dieser Datei abgelegte Zwischencode durch einen trivialen Übersetzungsschritt in den Ladecode überführt und im Arbeitsspeicher der MI abgelegt. Um es dem Benutzer zu ermöglichen, die in der hypothetischen Maschine ablaufenden Vorgänge zu verfolgen und ihm dadurch das korrekte Arbeiten seiner Programme anzuzeigen oder ihm bei der Fehlersuche Hilfen bereitzustellen, tritt der Benutzer in Interaktion mit der Testhilfe.

Der Benutzer wird durch das Kommando "==>" aufgefordert, eine Folge von Testhilfe-Kommandos einzugeben. Trennzeichen zwischen den einzelnen Kommandos ist dabei das Semikolon ";". Die eingegebene Kommandofolge wird durch "return" abgeschlossen. Dies löst die Ausführung der Testhilfe-Kommandos aus.

Die Leistungen der Testhilfe-Kommandos umfassen:

- Starten, Fortsetzen und Beenden eines Simulationslaufes.
- Anhalten des Simulationslaufs beim Eintreten von bestimmten Ereignissen:
  - Erreichen einer bestimmten Adresse.
  - Ausführen eines bestimmten Opcode.
  - Lesender oder schreibender Zugriff auf bestimmte Speicherbereiche und Register.
  - Auftreten einer Unterbrechung.
- Anzeigen und Verändern von Speicherinhalten und Registern in verschiedenen Formaten.
- Ausgeben der zuletzt durchlaufenen Befehle.

- Einzelbefehlsvariante.
- Sichern und Einlesen von Testhilfe-Ereignissen.
- Ausführen von Kommandos auf Betriebssystemebene durch die "shell" (Kommandoeingabeebene von UNIX).
- Ändern der Häufigkeit von Hardware-Fehlern bei externen Geräten.

Ein ausgezeichnetes Kommando ist "go," das den Simulationslauf startet und fortsetzt. Bei Eingabe der Kommandofolge soll "go" immer als letztes Kommando gegeben werden; alle auf "go" folgenden Kommandos werden ignoriert. Tritt während der Simulation ein vom Benutzer definiertes Ereignis ein oder wird der **HALT**-Befehl erreicht, wird dem Benutzer die Art des eingetretenen Ereignisses gemeldet und mittels "==>" ein neuer Eingabestring angefordert.

Das Auftreten eines Fehlers während der Abarbeitung eines Kommandos wird durch eine fehlerspezifische Meldung oder durch Ausgabe von

"@@@ syntax error"

signalisiert.

Da beim Eintreten eines bestimmten Ereignisses oft dieselben Testhilfe-Kommandos ausgeführt werden sollen, ist dem Benutzer die Möglichkeit gegeben, beim Setzen eines Ereignisses einen sogenannten "body," eine beim Eintreten eines Ereignisses automatisch abzuarbeitende Kommandofolge, anzugeben. Eine "body"-Definition wird durch eine öffnende geschweifte Klammer "{" eingeleitet und durch "}" abgeschlossen. Der Rumpf eines "body" besteht aus einer Kommandofolge. Trennzeichen zwischen den Kommandos ist das Semikolon ";" oder auch "newline." Im letzteren Fall fordert die Testhilfe mit dem Zeichen "?" die weitere Eingabe von Kommandos zur Komplettierung des Rumpfes an. Das folgende kleine Beispiel einer Einzelbefehlsvariante soll den Dialog zwischen Testhilfe und Benutzer veranschaulichen:

```
==> set single step {
? backtrace
? display SP; psw
? go
? }
==>
```

Zu den "bodies" sind folgende Anmerkungen zu machen:

1. Beim Einlesen eines "body" erfolgt keine Syntaxprüfung; diese werden erst bei der Ausführung des Kommandos erkannt.
2. Treten während der Abarbeitung eines Maschinenbefehls mehrere Testhilfe-Ereignisse ein, werden dem Benutzer zuerst alle Ereignisse gemeldet; erst dann werden die zugehörigen "bodies" ausgeführt.

3. Bei der Abarbeitung eines "body" wird jedes Kommando vor seiner Ausführung erst am Terminal ausgegeben; die Ausgabe wird durch das Symbol "– –>" eingeleitet.

4. Der Benutzer ist für den Inhalt der "bodies" selbst verantwortlich. Grundsätzlich dürfen in einem "body" alle Kommandos verwendet werden, insbesondere auch das "go"-Kommando. Es ist jedoch unzulässig, innerhalb einer "body"-Definition eine weitere "body"-Definition zu beginnen. Darüberhinaus werden nach einem "go"-Kommando alle weiteren Kommandos des "body"-Rumpfes ignoriert.

   Das "go"-Kommando selbst tritt nur in Kraft, wenn alle zur Bearbeitung anstehenden "bodies" abgearbeitet sind und alle auftretenden Ereignisse mit einem "body" versehen sind, die das "go"-Kommando enthalten. Ansonsten wird mit dem Symbol "==>" ein neuer Kommandostring angefordert.

Vor der Beschreibung der einzelnen Kommandos und deren Wirkung seien einige allgemeine Definitionen und Bemerkungen aufgeführt.

<**number**>: Dezimalzahl.

<**address**>, <**opcode**>: Hexadekadische Zahl (bestehend aus den Ziffern 0 bis 9 und den Großbuchstaben A bis F).

<**mnemonic**>: Befehlsname in Großbuchstaben.

<**register**>: R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 | R10 | R11 | R12 | R13 | R14 | SP | R15 | PC | PSW | P0B | P0L | P1B | P1L | SB | SL | IPL | RKMAP[1] (MAPEN) | RKNR

<**type**>: store | fetch | watch

Anmerkungen:

1. Ist im folgenden von "virtueller Adresse" die Rede, bedeutet das bei *current mode = user mode* virtuelle Adresse, und bei *current mode = kernel mode* Arbeitsspeicheradresse bzw. Adresse eines EAP-Registers.

2. Die Register besitzen die oben aufgeführte Ordnung. Das ist bei der Angabe von einer Registergruppe in einem Kommando von Bedeutung.

3. Wird in einem Kommando durch Angabe von zwei Adressen ein virtueller Speicherbereich definiert, so müssen beide Adressen entweder der "system region" oder aber der "program region" angehören.

4. Da jedes Ereignis mit einem "body" versehen sein kann, wird dies bei der nun folgenden Besprechung der einzelnen Kommandos nicht jedes Mal eigens erwähnt.

[1] Nur in der 4-Rechnerkern-MI relevant und implementiert. Dort gilt: MAPEN ist Synomym für RKMAP.

5.1 Starten, Fortsetzen und Beenden eines Simulationslaufes

Kommando: go $\{\sqcup$ <address>$\}^{0,1}$

Starten des Simulationslaufes ab der Adresse <address>. Wird <address> weggelassen, wird der Simulationslauf an der unterbrochenen Stelle fortgesetzt bzw. ab Startadresse 0 begonnen.

Kommando: quit

Beenden des Simulationslaufs.

5.2 Anhalten des Simulationslaufes beim Eintreten von bestimmten Ereignissen

5.2.1 Adreß-Haltepunkte

Adreß-Haltepunkte erlauben eine Unterbrechung des Simulationslaufes, wenn eine bestimmte Adresse im Assemblerprogramm erreicht wird. Dazu muß der Adreß-Haltepunkt auf die Anfangsadresse des betreffenden Befehls, d.h. auf die Adresse des Operationscodes, gesetzt werden.

Adreß-Haltepunkte erhalten zur besseren Identifikation Nummern, die fortlaufend vergeben werden. Adreß-Haltepunkte behalten ihre Nummer auch bei, wenn früher gesetzte Adreß-Haltepunkte mit einer niedrigeren Nummer gelöscht werden; ebenso erhalten neu gesetzte Adreß-Haltepunkte nie die Nummer eines zuvor gelöschten Adreß-Haltepunktes, um Verwechslungen auszuschließen. Dieser Sachverhalt gilt weiter unten auch für Befehls-Haltepunkte sowie für fetch-, store- und watch-Haltepunkte.

Kommando: set $\sqcup$ breakpoint $\sqcup$ <address> $\{$<body>$\}^{0,1}$

Setzen eines Adreß-Haltepunktes auf die virtuelle Adresse <address>.

Beispiel: `set breakpoint 80001000`

Setzen eines Adreß-Haltepunktes auf Adresse 80001000.

Kommando: clear $\sqcup$ breakpoint $\sqcup$ #<number>

Kommando: clear $\sqcup$ breakpoint $\left\{\sqcup \text{ <address1>} \ \{\sqcup \text{ <address2>}\}^{0,1}\right\}^{0,1}$

Löschen eines Adreß-Haltepunktes mit Nummer <number> bzw. Löschen aller Adreß-Haltepunkte im Speicherbereich <address1> bis <address2> (optional). Wird kein Bereich angegeben, werden nach vorheriger Abfrage alle Adreß-Haltepunkte gelöscht.

Beispiel: `clear breakpoint 00000000 3FFFFFFF`

Löschen aller Adreß-Haltepunkte im P0-Bereich.

Kommando: list $\sqcup$ breakpoint $\sqcup$ #<number>

Kommando: list $\sqcup$ breakpoint $\left\{\sqcup \text{ <address1>} \ \{\sqcup \text{ <address2>}\}^{0,1}\right\}^{0,1}$

Auflisten eines Adreß-Haltepunktes mit Nummer <number> bzw. aller Adreß-Haltepunkte im Adreßbereich <address1> bis <address2> (optional). Wird kein Bereich angegeben, werden alle Adreß-Haltepunkte gelistet.

Beispiel: `list breakpoint #3`

Auflisten des Adreß-Haltepunktes mit Nummer 3.

5.2.2 Befehls-Haltepunkte

Befehls-Haltepunkte unterbrechen den Simulationslauf, wenn ein bestimmter Befehl ausgeführt wird. Sie erhalten Nummern, die die Befehlsgruppe, auf die der Befehls-Haltepunkt gesetzt ist, zu einer Einheit zusammenfassen.

Kommando: set ⊔ trap ⊔ instruction ⊔<mnemonic1>{⊔<mnemonic2>}^{0,1} {<body>}^{0,1}
Kommando: set ⊔ trap ⊔ instruction ⊔ <opcode1>{⊔ <opcode2>}^{0,1} {<body>}^{0,1}

Setzen eines Befehls-Haltepunktes auf die angegebene Befehlsgruppe <mnemonic1> bis <mnemonic2> bzw. <opcode1> bis <opcode2>. Es kann wahlweise der Operationcode in hexadekadischer Schreibweise oder der Name des Befehls angegeben werden. Bei letzterem müssen Mehrdeutigkeiten bei den arithmetischen und logischen Befehlen, die 2 oder 3 Operanden haben können, durch Anhängen einer "2" oder "3" an den Befehlsnamen beseitigt werden.

Um ein ordnungsgemäßes Beenden des Simulationslaufes beim Erreichen des **HALT**-Befehls zu gewährleisten, wird zu Beginn einer jeden Simulation für diesen Befehl ein Befehls-Haltepunkt gesetzt; er erhält die Nummer 1 und kann vom Benutzer nicht gelöscht werden.

Werden mehrere Befehls-Haltepunkte gesetzt, dürfen sich die angegebenen Befehlsgruppen nicht überschneiden, ansonsten wird diese Eingabe ignoriert.

Beispiel: `set trap instruction ADDB2`

Setzen eines Befehls-Haltepunktes auf den 2-Adreß-Befehl ADDB.

Kommando: clear ⊔ trap ⊔ instruction ⊔ #<number>
Kommando: clear ⊔ trap ⊔ instruction $\left\{\sqcup \text{<mnemonic1>}\ \{\sqcup \text{<mnemonic2>}\}^{0,1}\right\}^{0,1}$
Kommando: clear ⊔ trap ⊔ instruction $\left\{\sqcup \text{<opcode1>}\ \{\sqcup \text{<opcode2>}\}^{0,1}\right\}^{0,1}$

Löschen eines Befehls-Haltepunktes mit Nummer <number> bzw. aller Befehls-Haltepunkte, die vollständig in der angegebenen Befehlsgruppe <mnemonic1> bis <mnemonic2> bzw. <opcode1> bis <opcode2> liegen. Ohne Angabe eines Arguments werden nach vorheriger Abfrage alle Befehls-Haltepunkte außer demjenigen für den **HALT**-Befehl gelöscht.

Beispiel: `clear instruction 01 0F`

Löschen aller Befehls-Haltepunkte auf privilegierten Befehlen.

Kommando: list ⊔ trap ⊔ instruction ⊔ #<number>

Kommando: list ⊔ trap ⊔ instruction $\left\{\sqcup \text{<mnemonic1>}\ \{\sqcup \text{<mnemonic2>}\}^{0,1}\right\}^{0,1}$

Kommando: list ⊔ trap ⊔ instruction $\left\{\sqcup \text{<opcode1>}\ \{\sqcup \text{<opcode2>}\}^{0,1}\right\}^{0,1}$

Auflisten des Befehls-Haltepunktes mit Nummer <number> bzw. aller Befehls-Haltepunkte, die vollständig oder auch nur teilweise in der angegebenen Befehlsgruppe <mnemonic1> bis <mnemonic2> bzw. <opcode1> bis <opcode2> liegen.

Beispiel: `list trap instruction ADDB2 DIVD3`

Auflisten aller Befehls-Haltepunkte auf arithmetischen Befehlen.

5.2.3 Store-, fetch- und watch-Haltepunkte

Store-, fetch- und watch-Haltepunkte erlauben eine Überwachung von virtuellen Speicherbereichen und Registern auf Zugriffe eines bestimmten Typs <type>. Es wird unterschieden zwischen schreibendem (store), lesendem (fetch) und allgemeinem (watch) Zugriff. Im Benutzermodus wird nur der Zugriff auf virtuelle Adressen überwacht, im Systemmodus der Zugriff auf Maschinenadressen.

Es werden nicht alle Zugriffe während der Ausführung eines Maschinenbefehls überwacht; vielmehr wird unterschieden zwischen impliziten und expliziten Zugriffen. Nur die expliziten, aber nicht die impliziten Zugriffe unterliegen der Überwachung. Im folgenden soll erklärt werden, was unter expliziten und impliziten Zugriffen zu verstehen ist:

- Überwachung unter dem Aspekt "Adressierungsarten."

  Bei der Adreßrechung erfolgt eine ganze Reihe von impliziten Zugriffen:

  1. relative Adresse: <Ausdruck für ganze Zahl> + !<**R**x>

     Der lesende Zugriff auf das Register <**R**x> ist implizit.

  2. indirekte Adresse: !(<Ausdruck für ganze Zahl> + !<**R**x>)

     Der lesende Zugriff auf das Register <**R**x> ist implizit; der Zugriff auf die Adresse <Ausdruck für ganze Zahl> + !<**R**x> hingegen explizit.

  3. Der lesende Zugriff auf das Indexregister /<**R**y>/ ist implizit.

  4. Kelleradressierung: !<**R**x>+ bzw. –!<**R**x>

     Der lesende Zugriff auf das Register <**R**x> zum Ermitteln des Kellerpegels und der schreibende Zugriff auf das Register <**R**x> zum Inkrementieren und Dekrementieren ist implizit.

- Überwachung unter dem Aspekt "Speicherabbildung."

  Jeglicher Zugriff auf Seitentafelelemente, ob im virtuellen Adreßraum oder im Arbeitsspeicher liegend, ist implizit.

- Überwachung unter dem Aspekt "Befehlszähler und Sonderregister."

Der während des Befehlszyklus laufend erfolgende lesende und schreibende Zugriff auf den Program Counter PC (Fortschalten des Befehlszählers) ist implizit. Explizite Zugriffe auf den Befehlszähler durch Befehle wie **JUMP !R10** oder gar **CLEAR W PC** werden selbstverständlich überwacht.

Eine Überwachung der Basis- und Längenregister der Seitentafeln des P0- und P1-Bereichs (P0B, P0L, P1B, P1L) erfolgt nur beim Laden (**LPCB**) und Speichern (**SPCB**) des Prozeßkontrollblocks. Alle Zugriffe bei der Verwendung innerhalb der Speicherabbildungsfunktion sind implizit. Analog wird auf das Basis- und Längenregister der Seitentafel des Systembereichs (SB, SL) nur durch die Befehle **LSB**, **SPSB**, **LSL** und **SPSL** explizit zugegriffen.

Der Zugriff auf die restlichen Sonderregister IPL und RKMAP (MAPEN) erfolgt explizit nur über die Befehle **LIPL**, **SPIPL**, **LRKMAP** (**LMAPEN**) und **SPRKMAP** (**SPMAPEN**). Schließlich sind alle Zugriffe auf das Prozessorstatuswort PSW implizit, soweit sie nicht über die Befehle **SPPCB** oder **LPCB** erfolgen.

Kommando: set ⊔ trap ⊔ <type> ⊔ <address1>{⊔ <address2>}^{0,1} {<body>}^{0,1}
Kommando: set ⊔ trap ⊔ <type> ⊔ <register1>{⊔ <register2>}^{0,1}

Setzen eines <type> Haltepunktes auf den virtuellen Speicherbereich <address1> bis <address2> bzw. auf eine Registergruppe <register1> bis <register2>. Auch hier wird eine Nummer vergeben, über die dieser Haltepunkt angesprochen werden kann. Werden mehrere Haltepunkte des selben Typs gesetzt, dürfen sich die Speicherbereiche bzw. Registergruppen nicht überlappen, ansonsten wird die letzte Eingabe ignoriert.

Beispiel: `set trap fetch 203FF900 203FF909`

Überwachung aller lesenden Zugriffe auf die Register des EAP1.

Kommando: clear ⊔ trap ⊔ <type> ⊔ #<number>
Kommando: clear ⊔ trap ⊔ <type> $\left\{\sqcup \text{<address1>}\ \{\sqcup \text{<address2>}\}^{0,1}\right\}^{0,1}$
Kommando: clear ⊔ trap ⊔ <type> $\left\{\sqcup \text{<register1>}\ \{\sqcup \text{<register2>}\}^{0,1}\right\}^{0,1}$

Löschen des <type> Haltepunktes mit Nummer <number> bzw. aller <type> Haltepunkte, die sich vollständig im Speicherbereich <address1> bis <address2> bzw. in der Registergruppe <register1> bis <register2> befinden. <type> Haltepunkte, die sich nur teilweise in diesem Bereich befinden, bleiben unberührt. Wird weder ein Speicherbereich noch eine Registergruppe bezeichnet, werden nach vorheriger Abfrage alle Haltepunkte des angegebenen Typs gelöscht.

Beispiel: `clear trap watch 40000000 7FFFFFFF`

Löschen aller watch Haltepunkte im P1-Bereich.

Kommando: list ⊔ trap ⊔ <type> ⊔ #<number>
Kommando: list ⊔ trap ⊔ <type> $\left\{\sqcup \text{<address1>}\ \{\sqcup \text{<address2>}\}^{0,1}\right\}^{0,1}$
Kommando: list ⊔ trap ⊔ <type> $\left\{\sqcup \text{<register1>}\ \{\sqcup \text{<register2>}\}^{0,1}\right\}^{0,1}$

Auflisten des <type> Haltepunktes mit Nummer <number> bzw. aller <type> Haltepunkte, die vollständig oder auch nur teilweise im Speicherbereich <address1> bis <address2> oder in der Registergruppe <register1> bis <register2> liegen.

Beispiel: `list trap fetch R0 R15`
Auflisten aller fetch-Haltepunkte auf den Registern R0 bis R15.

5.2.4 Unterbrechungs-Haltepunkt

Ist der Unterbrechungs-Haltepunkt gesetzt, werden dem Benutzer auftretende Unterbrechungen beim Sprung auf die Unterbrechungsbehandlung gemeldet. Der Begriff Unterbrechung wird hier im weiteren Sinn gebraucht, schließt also interne Unterbrechungen ("exceptions") sowie externe Unterbrechungen ("interrupts" im engeren Sinn, Eingriffe) mit ein.

Kommando: set ⊔ trap ⊔ interrupt {<body>}^{0,1}
Setzen des Unterbrechungs-Haltepunktes.

Kommando: clear ⊔ trap ⊔ interrupt
Löschen des Unterbrechungs-Haltepunktes.

Kommando: list ⊔ trap ⊔ interrupt
Gibt Auskunft, ob der Unterbrechungs-Haltepunkt gesetzt oder gelöscht ist.

5.2.5 Abbruch durch den Benutzer

Neben dem quit-Kommando besteht für den Benutzer auch noch folgende Möglichkeit den Simulationslauf zu beenden:

Kommando: Senden des "interrupt"-Signals mit der DEL-Taste
Die Simulation kann jederzeit durch den Benutzer durch Senden des "interrupt"-Signals unterbrochen werden. Der zur Bearbeitung anstehende Befehl jedes Rechnerkerns wird beendet; erst dann wird die Kontrolle an den Benutzer zurückgegeben.

5.3 Anzeigen und Verändern von Speicherinhalten und Registern

Mit dem "display"-Kommando können Speicherinhalte ausgegeben und Register inspiziert werden.

Kommando: display ⊔ <address1>{⊔ <address2>}^{0,1} {/<format>}^{0,1}
Kommando: display ⊔ <register1>{⊔ <register2>}^{0,1} {/<format>}^{0,1}
Ausgeben des Inhalts von Speicherbereichen <address1> bis <address2> oder Registergruppen <register1> bis <register2> in verschiedenen Formaten. Als Formate <format> sind zugelassen:

bd: Byte dezimal (Ganzzahl mit Vorzeichen)

bo: Byte oktal

bx: Byte hexadekadisch

hd: Halbwort dezimal (Ganzzahl mit Vorzeichen)

ho: Halbwort oktal

hx: Halbwort hexadekadisch

wd: Wort dezimal (Ganzzahl mit Vorzeichen)

wo: Wort oktal

wx: Wort hexadekadisch

f: Gleitpunktzahl in E-Schreibweise

d: doppelt lange Gleitpunktzahl in E-Schreibweise

a: Ausgabe von ASCII-Zeichen

Das voreingestellte Format ist "wx"; jedes Format bleibt bis zur nächsten Formatangabe in Kraft. Das Format "a" gibt ASCII-Zeichen aus, die in Hochkommata gesetzt sind. Ist das Zeichen nicht abdruckbar, wird der hexadekadische Wert ausgegeben.

Im IEEE-Standard läßt sich nicht jedes beliebige Bitmuster als Gleitpunktzahl darstellen; in diesen Fällen erscheint eine der folgenden Erläuterungen:

<**den**> Gleitpunktzahl ist nicht normalisiert

<**+oo**> Gleitpunktzahl symbolisiert + unendlich

<**-oo**> Gleitpunktzahl symbolisiert - unendlich

<**NaN**> Gleitpunktzahl ist *Not a Number*

Beispiel: `display P0B RKMAP/wx`

Ausgeben aller Sonderregister im Format "Wort hexadekadisch".

Kommando: psw

Das Prozessorstatuswort wird in seine Bestandteile zerlegt in übersichtlicher Form ausgegeben. Der Benutzer kann auf einen Blick Ablaufpriorität, Arbeitsmodus, Condition Codes usw. erfassen. Daneben besteht natürlich auch die Möglichkeit, durch "display PSW" das Prozessorstatuswort in einem beliebigen Format (außer "d") auszugeben.

Mit dem "modify"-Kommando können Speicherinhalte und Register beliebig verändert werden.

Kommando: modify ⊔ <address1>{⊔ <address2>}^{0,1} =<value>{/<format>}^{0,1}
Kommando: modify ⊔ <register1>{⊔ <register2>}^{0,1} =<value>{/<format>}^{0,1}

Der angegebene Speicherbereich <address1> bis <address2> bzw. die Registergruppe <register1> bis <register2> wird mit Datenelementen mit Wert <value> gefüllt. Ist der Wert <value> nicht vom aktuell eingestellten Format, muß das Format <format> mit angegeben werden, auch wenn es sich aus dem Zusammenhang ergeben würde. Als Formatangaben <format> sind die unter "display" angegebenen zugelassen. Im Format "a" ist das einzugebende Zeichen in Hochkommata zu setzen.

Beispiel: `modify R10 = 1.4142135/f`

In das Register R10 wird ein Näherungswert für "Wurzel 2" geschrieben.

5.4 Ausgeben der zuletzt durchlaufenen Befehle

Kommando: backtrace {⊔ <number>}^{0,1}

Es werden die letzten <number> Befehle ausgegeben, höchstens jedoch 16. Wird der Parameter <number> weggelassen, wird der als letztes ausgeführte Befehl ausgegeben. Bei der Rückverfolgung werden folgende Angaben gemacht:

- Befehlszählerstand zu Beginn des Befehls.
- Name des Befehls.
- Operationscode.

Danach folgen Informationen über die einzelnen Operanden, aufgegliedert in Adresse und Wert des Operanden. Als Adressenangabe kann dabei auftreten:

- <address>: virtuelle Adresse des Operanden.
- <register >: Operand war das Register <register>.
- i: direkter Operand.
- @@@: Bei der Bestimmung des Operanden trat ein Speicherschutz- oder Seite-fehlt-Alarm auf.

Die Angabe des Wertes des Operanden erfolgt grundsätzlich in hexadekadischer Schreibweise. Bei den arithmetischen Befehlen, die 2 oder 3 Operanden haben können, erfolgt die Operandeninformation auch bei 2-Adreßbefehlen in 3 Zeilen, um den berechneten Wert in jedem Fall angeben zu können.

Beispiel: `backtrace`

Am Terminal könnte folgende Ausgabe erscheinen:

```
00000763 : ADDW2  (1) 00001002 : 0000000F
                  (2) R10 : FFFFFFFF
                  (2) R10 : 0000000E
```

Dies würde besagen:

- Die Anfangsadresse des Befehls war 00000763.
- Es wurde der 2-Adreßbefehl ADDW ausgeführt.
- Das "Wort" ab Adresse 00001002 mit Wert 15 wurde auf das Register R10 (Inhalt -1) addiert; das Ergebnis 14 wurde im Register R10 abgelegt. Die Zahlen in Klammern vor der Operandenadresse geben die Nummer des Operanden an.

5.5 Einzelbefehlsvariante

Kommando: set ⊔ single ⊔ step {<body>}^{0,1}

Nach jedem Maschinenbefehl wird dem Benutzer das Ereignis "single step" gemeldet; dies erleichtert durch eine ständige Kontrolle der in der Maschine ablaufenden Vorgänge das Auffinden besonders hartnäckiger Fehler.

Kommando: clear ⊔ single ⊔ step

Rückkehr in Normalmodus.

Kommando: list ⊔ single ⊔ step

Es wird der eingestellte Modus ("single step set" oder "single step cleared") ausgegeben.

5.6 Sichern und Einlesen von Testhilfe-Ereignissen

Zum Testen eines Programms kann ein umfangreicher Satz von Testhilfe-Ereignisse notwendig sein. Da während eines Simulationslaufes nur eine bedingte Fehlerkorrektur durch das "modify"-Kommando möglich ist, wäre es zeitaufwendig, alle Definitionen nach dem Verbessern des Assemblerprogramms und Neustart des Simulators neu eingeben zu müssen. Mit Hilfe der Kommandos "save" und "read" können alle definierten Testhilfe-Ereignisse auf einer Datei abgespeichert werden und von dort wieder eingelesen werden.

Da die Ereignisse in Form von "set"-Kommandos abgelegt werden, kann diese Datei editiert und geändert werden. Darüberhinaus bietet sich die Möglichkeit, im Übungsbetrieb den Studenten vorgefertigte Testhilfe-Ereignissätze anzubieten.

Kommando: save {⊔ "<filename>"}^{0,1}

Alle vom Benutzer festgelegten Testhilfe-Ereignisse werden auf der Datei <filename> abgelegt. Wird kein Dateiname angegeben, wird die Datei "th_ereignisse" benutzt.

Kommando: read {⊔ "<filename>"}^{0,1}

Die in der Datei <filename> (Voreinstellung: "th_ereignisse") enthaltenen "set"-Kommandos werden ausgeführt; somit werden die durch die "set"-Kommandos definierten Ereignisse gesetzt.

5.7 Ändern der Häufigkeit von Hardware-Fehlern bei externen Geräten

Auch bei völlig korrekter Definition eines EA-Auftrags arbeiten die externen Geräte i.a. nicht völlig fehlerfrei; vielmehr treten mit einer bestimmten Häufigkeit Fehler auf. Dies Fehlerrate kann im Simulationsbetrieb in weiten Grenzen verändert werden; so können von der Hardware hervorgerufene Fehler ganz eliminiert werden oder die Fehlerrate extrem hoch festgelegt werden.

Die Fehlerhäufigkeit wird durch eine Größe "error level" repräsentiert, die die Werte 0 bis 5 annehmen kann. Die Zuordnung zwischen Fehlerhäufigkeit und Wert des "error levels" gibt nachfolgende Tabelle wieder:

error level	Fehlerhäufigkeit
0	0
1	1/4096
2	1/1024
3	1/256
4	1/64
5	1/4

Tabelle 5.1: Häufigkeitsverteilung simulierter Hardware-Fehler.

Der "error level" gilt für alle angeschlossenen Geräte und bestimmt die Wahrscheinlichkeit, mit der bei der Durchführung eines EA-Auftrags ein Fehler auftritt. Ist der "error level" z.B. mit 4 belegt, ist im Durchschnitt jeder 64. EA-Auftrag gestört.

Kommando: set ⊔ error ⊔ level ⊔ <number>

Der "error level" erhält den Wert <number>; der voreingestellte Wert ist 1.

Kommando: list ⊔ error ⊔ level

Der derzeitig gültige "error level" wird ausgegeben.

Kommando: clear ⊔ error ⊔ level

Der "error level" wird auf seinen voreingestellten Wert zurückgesetzt.

5.8 Löschen aller Testhilfe-Ereignisse

Kommando: clear

Nach vorheriger Abfrage werden alle Testhilfe-Ereignisse gelöscht; der "error level" wird auf den Wert 1 gesetzt.

5.9 Weiterleiten von Kommandos an die "shell"

Kommando: ! <command>

Der String <command> wird an die "shell" zur Ausführung weitergeleitet.

Eine typische Anwendung wäre das Auflisten des Adreßbuches (siehe Abschnitt 4.2.5) während der Simulation, um sich die Adresse einer Programmvariablen anzeigen zu lassen. Dies geschieht mit:

==> !cat "<Datei mit Adreßbuch>"

5.10 Ausgeben von Meldungen

Kommando: message ␣ "<text>"

Die Meldung <text> wird am Terminal ausgegeben. Dieses Kommando ist für die Verwendung innerhalb von "bodies" gedacht. Mit seiner Hilfe kann der Benutzer seinen abstrakten Testhilfe-Ereignissen einen konkreten, anschaulichen Hintergrund geben, da bei jeder Ausführung des "body" der Text <text> ausgegeben wird.

Beispiel: Der Benutzer setzt folgenden mit einem "body" versehenen Adreß-Haltepunkt, um sich ein Zwischenergebnis, das bei der Ausführung des Befehls ab Adresse 2029 im Register R11 abgelegt wird, ausgeben zu lassen:

```
set breakpoint 2029 {
message 'Zwischenergebnis:'
display R11/wd
go
}
```

Die Meldung "message 'Zwischenergebnis:' " erscheint bei jeder Abarbeitung des oben definierten "body."

5.11 Setzen eines Filters für Testhilfeausgaben

Um die Ausgaben verschiedener Testhilfekommandos auf die jeweils interessierenden Rechnerkerne[2] zu beschränken, wird ein "Filter" benutzt, der angibt, für welche Rechnerkerne Ausgaben auf dem Bildschirm erscheinen. Die Voreinstellung dieses Filters ist

[2]Selbstverständlich ist das *filter*-Kommando nur in der 4-Rechnerkern-MI relevant und implementiert.

Rechnerkern 0, d.h. nach dem Starten des Simulators erscheinen nur Ausgaben für diesen Rechnerkern.

Kommando: filter $\{\sqcup$ <0>$\}^{0,1}$ $\{\sqcup$ <1>$\}^{0,1}$ $\{\sqcup$ <2>$\}^{0,1}$ $\{\sqcup$ <3>$\}^{0,1}$

Wird das Kommando Filter ohne Parameter aufgerufgen, so werden die Rechnerkerne angezeigt, für die momentan Ausgaben auf dem Bildschirm erscheinen. Durch Angabe von Rechnerkernnummern (zwischen 0 und 3) werden die Ausgaben der angegebenen Rechnerkerne auf dem Bildschirm angezeigt.

Beispiel: `filter 2 3`

Setzen des Filters auf die Rechnerkerne 2 und 3.

5.12 Meldungen beim Eintreten von Ereignissen

Meldungen von Testhilfe-Ereignissen werden durch "+++" eingeleitet; danach folgt eine nähere Spezifikation des Testhilfe-Ereignisses. Diese Meldungen sind größtenteils selbsterklärend und sollen nur anhand einiger Beispiele erläutert werden. Der Zusatz *"... for RK_i"* ist nur in der 4-Rechnerkern-MI relevant und auch nur dort implementiert.

+++ breakpoint #5 address 0000A033 for RK2

Der Befehlszähler des Rechnerkerns 2 stand zu Beginn des aktuellen Befehls auf der Adresse 0000A033; dieser Adreß-Haltepunkt mit Nummer 5 wird dem Benutzer gemeldet.

+++ instruction trap #1 HALT (00) for RK1

Der letzte ausgeführte Befehl ist der **HALT**-Befehl mit Operationscode "00"; dieser Befehls-Haltepunkt besitzt immer die Nummer 1. In diesem Fall führte Rechnerkern 1 den **HALT**-Befehl aus.

+++ fetch trap #3 R8 for RK0

Im letzten Befehl wurde auf das Register R8 von Rechnerkern 0 lesend zugegriffen.

+++ store trap #9 address 80003891 - 80003894 for RK0

Bei der Ausführung des letzten Befehls wurden von Rechnerkern 0 auf die Adressen 80003891 bis 80003894 vier Bytes (ein Wort) abgelegt.

+++ watch trap #2 PC (store) for RK0

Auf den PC des Rechnerkerns 0 wurde explizit zugegriffen. Bei einem watch-Haltepunkt wird in Klammern die Zugriffsart (hier schreibender Zugriff) angegeben.

+++ interrupt trap (Plattenspeicher) for RK3

Der Prozessor EAP1 hat seinen EA-Auftrag durchgeführt und eine Unterbrechung ausgelöst, die jetzt zur Bearbeitung durch Rechnerkern 3 ansteht. Der Befehlszähler steht auf der Anfangsadresse der zugehörigen Unterbrechungsbehandlung.

+++ single step for RK0

Einzelbefehlsvariante ist gesetzt; nach der Ausführung eines jeden Befehls erscheint diese Meldung. Die Angabe der Rechnerkernnummer zeigt an, welcher Rechnerkern den Befehl beendete.

+++ user break

Der Benutzer hat eben die "DEL"-Taste gedrückt und damit den Simulationslauf unterbrochen.

5.13 Ein-/Ausschalten des Weckeralarms

Kommando:[3] clock $\{\sqcup <0 \mid 1>\}^{0,1}$

Durch Aufruf des Kommandos ohne den Parameter wird die aktuelle Einstellung — 0/1 für Weckeralarm aus/ein — angezeigt. Wird als Parameter "0" angegeben, so wird der Weckeralarm abgeschaltet, nach der Eingabe von "1" werden dagegen Weckeralarme generiert, sobald der Simulatorlauf gestartet wurde. Die Voreinstellung ist "0."

Der Weckeralarm unterbricht alle vier Rechnerkerne. Auslöser für einen Wecheralarm ist das Erreichen einer Rechnerkern-spezifischen Befehlszyklenzahl x. Hat ein Rechnerkern x Befehlszyklen durchlaufen, so wird ihm ein Weckeralarm zugestellt.

5.14 Weitere Benutzerhinweise: Externe Geräte

Der EA-Prozessor EAP2 betreibt einen Drucker sowie ein Terminal. Tritt während der Ausführung eines EA-Auftrags ein vom Benutzer hervorgerufener Fehler (z.B. falsche baud-Rate) oder ein simulierter Hardware-Fehler auf, wird das Zeichen verfälscht übertragen. Um in diesem Fall trotzdem eine lesbare Ausgabe zu erhalten, wird als verfälschtes Zeichen ein abdruckbares Zeichen gewählt.

Die Druckausgabe erfolgt auf eine Datei "druckaus". Diese Datei kann nach Beendigung des Simulationslaufs am Terminal oder dem Drucker des "Host"- Rechners ausgegeben werden.

Das simulierte Terminal wird durch das Terminal des "Host"-Rechners nachgebildet. Dabei wird folgendermaßen vorgegangen: Beim Lesen des Terminals wird durch die Aufforderung

Kanal 3 Empfaenger (Tastatur): zu uebertragender String ?

ein Eingabestring angefordert, der um das Zeichen "Linefeed" (hexadekadisch 0A) ergänzt wird. Der aktuelle und alle folgenden EA-Aufträge bedienen sich aus diesem Eingabestring, bis der Vorrat erschöpft ist und als letztes Zeichen "Linefeed" übertragen wurde. Erst

[3] Nur die 4-Rechnerkern-MI besitzt eine Uhr. Somit ist das *clock*-Kommando auch nur dort realisiert.

der nächste Lesevorgang am Terminal ruft wieder obige Meldung hervor und fordert zur Eingabe eines weiteren Strings auf.

Beispiel: Wird auf obige Anforderung der String "ja" eingebenen und liest der (vom Benutzer der MI erstellte) Treiber 3 Zeichen vom Terminal, werden nacheinander die Zeichen "j", "a" und "Linefeed" in das Empfangsdatenregister übertragen. Dadurch kann der Treiber das Ende des Eingabestrings erkennen.

Die Ausgabe am Bildschirm erfolgt analog: Am simulierten Terminal auszugebende Zeichen werden solange zwischengepuffert, bis der aufgebaute String eine Länge von 78 Zeichen erreicht hat, ein "Linefeed" übertragen wurde oder ein Testhilfe-Ereignis eintritt; erst dann wird die Zeichenkette mit der Meldung

Kanal 3 Sender (Bildschirm):

am Terminal des "Host"-Rechners ausgegeben.

Kapitel 6

Graphikoberfläche für 1-RK-MI

6.1 Einleitung

Die bisher in Gebrauch befindliche Version der 1-RK-MI war auf rein textuelle Kommunikation mit dem Benutzer ausgerichtet. Mit der Verfügbarkeit entsprechend leistungsfähiger Hardware, die es gestattet, auch verhältnismäßig rechenzeitintensive graphische Benutzerschnittstellen mit befriedigender Laufzeit zu realisieren, bot es sich an, gerade für den MI-Simulator, bei dem ja der Benutzer häufig während des Simulationslaufs gewisse Speicherstellen, Register, Statusbits usw. überprüfen soll, eine Benutzeroberfläche zu implementieren, die solche häufig benötigten Statusinformationen von sich aus ständig anzeigt und diese nicht erst nach Abfrage über eine Kommandozeile verfügbar macht. Insbesondere sollten mit der graphischen MI-Oberfläche auch in einem eingeschränkten Umfang die Eigenschaften eines Source-Level-Debuggers realisiert werden, d.h. die Möglichkeit, den Ablauf des MI-Programms innerhalb der Benutzeroberfläche auf Quellcode-Ebene mitzuverfolgen.

6.2 Der Assembler

Der MI-Assembler wird durch Eingabe des Befehls `XAsm` aus der UNIX-Shell gestartet. Die Benutzeroberfläche des MI-Assemblers besteht im wesentlichen aus einer Dateiauswahlbox ähnlich wie bei anderen graphischen Benutzerschnittstellen. Sie dient dazu, das zu assemblierende MI-Quellprogramm auszuwählen, wobei nur Dateinamen angezeigt werden, die die Endung **.mi** oder **.MI** haben. Kataloge werden durch einen "/" am Ende des Namens gekennzeichnet; der Eintrag "./" verweist auf den aktiven Katalog selbst und "../" auf den nächsthöheren Katalog. Die Auswahl einer Datei oder eines Katalogs geschieht auf zwei mögliche Arten: (a) Selektieren des Namens mit der linken Maustaste und anschließendes Betätigen des *OK*-Buttons oder (b) Doppelklick mit der linken Maustaste auf den auszuwählenden Namen.

Wurde ein Katalog ausgewählt, so erscheint nun der Inhalt dieses gewählten Katalogs; wurde eine Datei selektiert, so wird diese Datei assembliert. Der Assembler erzeugt folgende

Ausgabedateien (im selben Katalog, in dem sich auch die Quelldatei befindet; Quelldatei = Name.mi): Name.addresses, Name.code, Name.crossref und Name.listing. Nach erfolgtem Assemblerdurchlauf erscheint eine Dialogbox, in der dem Benutzer entweder die erfolgreiche Assemblierung des Quellprogramms oder das Vorkommen von Fehlern im Quelltext mitgeteilt wird. War der Assemblerlauf fehlerfrei, so können nun weitere MI-Programme assembliert werden; waren Fehler im MI-Programm, so besteht die Möglichkeit, sich durch Betätigen des *Show Errors*-Buttons in einem separaten Fenster das Listing des Assemblers anzeigen zu lassen und dort mittels der Buttons *Show next Error* und *Show previous Error* vorwärts bzw. rückwärts von Fehler zu Fehler zu springen; ist der letzte Fehler erreicht, so bewirkt Betätigen des *Show next Error*-Buttons wieder den Rücksprung zum ersten aufgetretenen Fehler; ebenso bewirkt *Show previous Error* beim ersten Fehler den Sprung zum letzten Fehler im Programm.

Nach Verlassen dieses Fensters mit Back to Assembler können weitere Dateien assembliert werden; verlassen wird der Assembler durch Betätigen des mit *Quit* bezeichneten Buttons.

6.3 Der MI-Simulator

Der MI-Simulator wird aus der UNIX-Shell durch das Kommando `Xmi` gestartet.

Zunächst erscheint wie beim Assembler eine Dateiauswahlbox mit der Aufforderung, eine MI-Codedatei auszuwählen, in der in gleicher Weise wie beim Assembler Dateien und/oder Kataloge selektiert werden können. In dieser Dateiauswahlbox werden jedoch nur die Code-Ausgabedateien des MI-Assemblers, also Dateien mit Namen Name.code, angezeigt. Nach erfolgter Auswahl einer MI-Codedatei erscheint die eigentliche Simulator-Benutzerschnittstelle, die im wesentlichen aus acht Bedienungselementen besteht (siehe Abb. 6.1):

1. Listing-Fenster (Listing Window)
2. Simulatorkontrollelemente (Simulation Controls)
3. Rechnerkernregister-Anzeige (CPU Display)
4. Ausgabefenster (Output Window)
5. Kommandozeile (Command Line) für manuelle Kommandoeingabe
6. Kellerfenster (Stack Display)
7. Sonderregisterfenster (Special Registers)
8. Speicherauszug (Memory Map)

Die folgende Abbildung zeigt eine schematische Darstellung der MI-Benutzerschnittstelle. Die Bedienungselemente (7) und (8) sind standardmäßig nicht sichtbar.

6.3.1 Das Listing-Fenster

Im Listing-Fenster wird die zur geladenen Codedatei gehörige Listingdatei, die vom MI-Assembler erzeugt wurde, angezeigt. Mit Hilfe der Scrollbars kann horizontal und vertikal durch das Listing geblättert werden. Das Listing-Fenster bietet die Möglichkeit, durch Doppelklicken auf eine Zeile im Programmlisting, die eine gültige Adresse enthält (also keine Kommentar- oder Leerzeile) Haltepunkte (Breakpoints) für den Simulator an der entsprechenden Adresse, die in dieser Zeile angezeigt wird, zu setzen. Solche Breakpoints werden durch ein "B" in der ersten Spalte des Listing-Fensters gekennzeichnet. Analog kann durch nochmaliges Doppelklicken in eine Zeile, die einen Haltepunkt enthält, dieser wieder entfernt werden. Im Listing-Fenster wird darüberhinaus während des Ablaufs des MI-Programms ständig bzw. jeweils nach Erreichen von Haltepunkten etc. der nächste auszuführende Befehl im MI-Programm durch Invertieren der entsprechenden Zeile im Listing angezeigt (siehe Abb. 6.2).

Wurde eine Codedatei geladen, zu der keine Listingdatei im aktuellen Katalog existiert, so wird dies im Listing-Fenster durch eine entsprechende Nachricht vermerkt. In diesem Fall fehlen natürlich die oben angeführten Möglichkeiten, Breakpoints zu setzen, den Programmablauf mitzuverfolgen etc.

6.3.2 Die Simulatorkontrollelemente

Das *Simulation Controls*-Fenster enthält eine Reihe von Bedienungselementen, durch die häufig gebrauchte Funktionen des Simulators über einfaches Aktivieren durch Mausklicks ausgelöst werden können:

- Das *Settings*-Pullright-Menü bietet folgende Optionen:

  read: Einlesen von früher abgespeicherten Testhilfeeinstellungen; Auswählen dieses Menüpunktes bringt einen Dateiauswahldialog zur Anzeige, in dem der Dateiname einer Einstellungsdatei angegeben werden kann.

  save: Abspeichern der aktuellen Testhilfeeinstellungen in einer Datei; analog wie oben über einen Dateiauswahldialog.

  display: Anzeigen aller aktuellen Testhilfeeinstellungen durch nacheinander folgenden Aufruf der entsprechenden list-Kommandos.

  clear: Löschen aller aktuellen Testhilfeeinstellungen; dieser Befehl wird jedoch erst nach einer zusätzlichen Rückfrage an den Benutzer durchgeführt.

- Das *Set*-Pullright-Menü enthält:

  breakpoint: Setzen eines Adreßhaltepunktes im MI-Programm; die Adresse wird vom Benutzer in einer Dialogbox (in hexadekadischer Notation) eingegeben.

  trap instruction: Setzen eines Befehlshaltepunktes; auch dieser Menüpunkt bringt einen Dialog zur Anzeige, in dem der entsprechende Befehl eingegeben werden kann.

trap interrupt: Setzen des Unterbrechungshaltepunktes. trap ... zeigt ein Untermenü, in dem aus fetch-, store- und watch-traps ausgewählt werden kann; in der folgenden Dialogbox kann dann der Adreßbereich, für den der entsprechende Haltepunkt gelten soll, eingegeben werden.

- Das *Clear*-Pullright-Menü enthält:

  breakpoint: Löschen eines Adreßhaltepunktes bzw. mehrerer Adreßhaltepunkte innerhalb eines bestimmten Speicherbereiches; die Adresse bzw. der Adreßbereich wird in der dazu angezeigten Dialogbox eingegeben.

  trap instruction: Löschen eines Befehlshaltepunktes; der betreffende MI-Befehl wird im hierzu angezeigten Dialog eingegeben.

  trap interrupt: Löschen des Unterbrechungshaltepunktes. Löschen von fetch-, store- und watch-traps; Eingabe des Adreßbereiches über eine Dialogbox.

- Das *List*-Menü besitzt denselben Aufbau wie die beiden eben beschriebenen Menüs und erlaubt selektiv das Anzeigen der momentan gesetzten Adreß-, Befehlshaltepunkte usw.

- Der Checkbox *single step mode* erlaubt das Setzen des Einzelschrittmodus des MI-Simulators; bei jedem *Go*-Befehl wird dann nur der jeweils nächste Befehl abgearbeitet.

- Die Checkbox *show memory map* bringt — falls aktiviert — ein zusätzliches Fenster zur Anzeige, in dem ein vom Benutzer wählbarer Auszug aus dem Speicherbereich der MI ausgegeben wird.

- Die Checkbox *special registers* zeigt ein weiteres Fenster, in dem die Sonderregister der MI sichtbar (und editierbar) sind.

- Die Checkbox *continuous refresh* bewirkt — sofern aktiviert —, daß auch bei nicht gesetztem Einzelschrittmodus nach jedem ausgeführten MI-Befehl alle Statusanzeigen (Listing, Register, Keller usw.) aktualisiert werden. Ist weder der Einzelschrittmodus noch continuous refresh gesetzt, so werden alle Statusanzeigen erst nach Erreichen des nächsten Haltepunktes (spätestens nach Erreichen des Programmendes) aktualisiert.

- Das Pullright-Menü *Backtrace* gestattet die Ausführung der Backtrace-Funktion des MI-Simulators in Schritten von 1, 3, 5, 10 bzw. 15, d.h. die Ausgabe des letzten bzw. der 3, 5, 10 oder 15 zuletzt ausgeführten Opcodes sowie deren Operandeninformation.

- Der *Refresh*-Button gibt die Möglichkeit, manuell alle Statusanzeigen zu aktualisieren. Dies ist beispielsweise dann sinnvoll, wenn nach einer Veränderung der Größe des Fensters des MI-Simulators im Listing der Befehlszähleranzeiger aus dem Blickfeld verschwunden ist, wenn man verschiedene Register im Rechnerkernregister-Fenster

überschrieben, aber nicht durch Betätigen der Return-Taste tatsächlich verändert hat, usw.

- Der *Statistics*-Button gibt im Ausgabefenster die Anzahl der bisher ausgeführten MI-Befehle aus.

- Der *Stop*-Button dient dazu, ein laufendes MI-Programm an beliebiger Stelle anzuhalten.

- Der *Go*-Button startet den Befehlszyklus der MI und arbeitet — abhängig davon, ob der Einzelschrittmodus gesetzt ist — alle folgenden Befehle bis zum nächsten Haltepunkt oder nur den nächsten Befehl ab.

- Der *Restart*-Button dient dazu, nach vorheriger Rückfrage eine neue Codedatei in den MI-Simulator zu laden. Zu Auswahl der neu zu ladenden Codedatei wird wie beim Start der XMI eine Dateiauswahlbox angezeigt. Beim Restart werden alle Testhilfeeinstellungen gelöscht.

- Der *Quit*-Button dient — nach vorheriger Rückfrage — zum Verlassen des MI-Simulators.

6.3.3 Das Rechnerkernregister-Fenster

Im Rechnerkernregister-Fenster werden ständig die Inhalte der 16 Register der MI angezeigt. Dabei besteht die Wahl zwischen Anzeige als hexadekadisches Wort (achtstellig mit führenden Nullen) und Anzeige als Dezimalzahl. Die Anzeigeart kann durch entsprechende Einstellung der Mode-Radio-Buttons im Rechnerkernregister-Fenster eingestellt werden (wx = word hex; wd = word decimal).

In diesem Fenster können die Registerinhalte auch direkt geändert werden: dazu wird mit der linken Maustaste der Cursor in dem entsprechenden Register plaziert (an der entsprechenden Stelle erscheint dann ein senkrechter Strich als Eingabemarkierung), wobei durch Bewegen der Maus innerhalb der Registeranzeige bei gedrückter Maustaste auch ein ganzer Bereich des Inhalts selektiert werden kann (in diesem Fall erscheint der selektierte Bereich invertiert). Es können nun neue Zeichen eingegeben werden; mit der Delete-Taste wird das jeweils links von Textcursor liegende Zeichen bzw. der selektierte Textbereich gelöscht.

Anmerkung: Generell kann der Cursor in Textfeldern immer mit *Control-B* (*Control-F*) nach links (nach rechts) bewegt werden.

Achtung: Bei Änderungen von Registerinhalten ist das jeweils durch die Mode-Radio-Buttons gewählte Zahlenformat zu berücksichtigen; d.h. wenn in Hex-Mode eine Zahl in ein Register geschrieben wird, so wird diese Zahl als Hex-Zahl interpretiert, in Dezimal-Mode entsprechend als Dezimalzahl.

Zusätzlich befindet sich im Rechnerkernregister-Fenster eine nach Statusbits aufgeschlüsselte Anzeige des Prozessorstatuswortes (**PSW**). Es ist hier auch möglich, einzelne Statusbits in gleicher Weise wie oben für die Register beschrieben zu verändern. Der Inhalt der **IPL**-Bits wird immer — unabhängig von dem für die Register **R0** bis **R15** eingestellten Zahlenformat — als hexadekadische Zahl angezeigt (und wird auch beim Verändern immer als hexadekadische Zahl interpretiert).

6.3.4 Das Ausgabefenster

Im Ausgabefenster erhält der Benutzer sämtliche Reaktionen (Erfolgs- und Fehlermeldungen) des MI-Simulators auf seine Eingaben und Kommandos; die hier ausgegebenen Meldungen erscheinen auch in der Ablauf-Protokolldatei Name.protocol.

Im Ausgabefenster bleiben alle seit Beginn des Simulationslaufs ausgegebenen Meldungen gespeichert; es kann also jederzeit mit Hilfe des Scrollbars bis an den Anfang zurückgeblättert werden, um frühere Ausgaben anzuzeigen. Soll ein früher gegebener Befehl, der im Ausgabefenster angezeigt wird (alle Befehlseingaben im Ausgabefenster beginnen mit "=>"), noch einmal ausgeführt oder editiert werden, so besteht die Möglichkeit, durch einen Doppelklick mit der linken Maustaste auf die entsprechende Zeile im Ausgabefenster den gewünschten Befehl in die Kommandozeile zurückzuholen.

6.3.5 Die Kommandozeile

In der Kommandozeile können — wie vom bisherigen MI-Simulator gewohnt — alle Kommandos über die Tastatur eingegeben werden. Die Kommandozeileneingabe wird aktiviert durch einen Klick mit der linken Maustaste innerhalb der Kommandozeile; daran anschließend ist dort ein Textcursor in Form eines vertikalen Striches sichtbar. Abgeschlossen wird die Eingabe eines Kommandos durch Betätigen der Return-Taste.

Nach der Ausführung eines solchen Kommandozeilenbefehls bleibt der gegebene Befehl in der Kommandozeile erhalten, wird aber komplett selektiert (invertiert). Dies bedeutet, daß er einerseits durch erneutes Betätigen der Return-Taste sofort wiederholt werden kann (falls nötig), andererseits durch Eingabe eines neuen Befehls ohne Betätigen der Delete-Taste sofort gelöscht wird.

Bemerkung: Wie bei Windows-Systemen üblich, muß sich zur Eingabe von Kommandos über die Kommandozeile der Mauszeiger innerhalb ihres Rahmens befinden; ansonsten werden Tastatureingaben nicht akzeptiert.

Achtung: Die Eingabe von mehrzeiligen Kommandos (*Bodies*), die in der "früheren" MI möglich war, wird von der graphischen Oberfläche nicht mehr unterstützt; dieses Feature wird aber jetzt nicht mehr benötigt, da die Kommandozeile im Prinzip "beliebig" lange sein darf; bei Überschreiten der Länge des Eingabebereiches wird die Eingabe nach links weitergeblättert.

6.3.6 Das Kellerfenster

Im Kellerfenster wird der aktuelle Inhalt des Kellers (Stack) in der Form

ADRESSE: INHALT

angezeigt. Das oberste dargestellte Element ist dabei dasjenige Hex-Wort, auf das der Kellerpegel (**R14** resp. **SP**) gerade[1] zeigt; das zweite Element ist entsprechend das Hex-Wort, das sich an der Adresse 4 + Inhalt des Kellerpegels befindet usw.

Auch der Inhalt dieses Fensters kann durch Betätigen der zugehörigen Scrollbars verschoben werden. Durch einen Doppelklick auf eines der angezeigten Kellerelemente wird dieses als Zeiger aufgefaßt und im Ausgabefenster der Inhalt des referenzierten Speicherbereiches im aktuellen Ausgabeformat ausgegeben.

Der Button *Options* bringt eine Dialogbox auf den Bildschirm, in der verschiedene Anzeigeoptionen für das Kellerfenster eingestellt werden können:

Im Feld *Number of Items displayed* kann die gewünschte Anzahl der darzustellenden Kellerelemente eingegeben werden; die Maximalzahl beträgt 100 Hex-Wörter

Mit den beiden Radio-Buttons *show absolute addresses* und *show offset relative to SP* kann gewählt werden, ob die Kelleradressen als absolute Hex-Adressen oder als Displacement relativ zum Kellerpegel in der Form

n + !**SP**

dargestellt werden sollen.

Die Radio-Buttons *decimal words* und *hex words* erlauben die Wahl der Darstellung des Kellers zwischen dezimaler und hexadekadischer Form (achtstellig mit führenden Nullen).

Durch Betätigen des *Cancel*-Buttons wird dieser Dialog beendet, ohne eventuell gemachte Änderungen zu übernehmen; Selektieren des *OK*-Buttons beendet den Dialog und übernimmt die Änderungen als aktuelle Einstellung.

6.3.7 Das Sonderregisterfenster

Dieses Fenster wird nur bei Bedarf durch Aktivieren der *special registers*-Checkbox angezeigt. In ihm befinden sich Displays für die Inhalte der Sonderregister der MI (**PCBADR**, **SCBADR** usw.). Auch diese Register können wie für die anderen Register der MI beschrieben verändert werden. Die Spezialregister werden jedoch nur im hexadekadischen Modus angezeigt. Soll das Sonderregisterfenster nicht mehr angezeigt werden, so muß nur die *special registers*-Checkbox wieder deaktiviert werden.

6.3.8 Das Speicherauszug-Fenster

Dieses Fenster wird nur angezeigt, wenn die *show memory map*-Checkbox aktiviert wurde. In diesem Fenster werden zeilenweise 16 Bytes des Arbeitsspeichers der MI in der Form

[1] siehe "Doppelnatur" des Kellerpegels: Moduswechsel führen zu einem aktualisierten **SP**; vgl. Seite 9.

Hex-Adresse: 32-Bit-Wort1 ... 32-Bit-Wort4

dargestellt. Die Adresse wird immer als hexadekadische Zahl angezeigt; bei den pro Zeile vier 32-Bit-Wörtern kann durch die beiden Mode-Radio-Buttons zwischen hexadekadischer und dezimaler Darstellung gewählt werden. In den beiden Feldern *Start Address* und *End Address* können Anfangs- und Endadresse des Speicherauszugs als hexadekadische Zahlen eingeben werden (abgeschlossen durch Return). Dabei sind folgende Punkte zu berücksichtigen:

Ist die eingegebene Startadresse größer als die größte gültige Arbeitsspeicheradresse, so wird sie durch diese ersetzt; Analoges gilt für die Endadresse.

Ist die Endadresse mehr als 500 Hex-Worte (500 * 4 Bytes = 2000 Bytes) größer als die Startadresse, so wird für die Endadresse der Wert Startadresse + 2000 verwendet.

Wird eine Endadresse eingegeben, die kleiner als die aktuelle Startadresse ist, so wird die Startadresse so mitgeführt, daß die Länge des vorher angezeigten Bereiches, wenn möglich, erhalten bleibt; d.h. die neue Startadresse erhält den Wert

max (0, neue_Endadresse - Länge_des_alten_Bereichs).

Gleiches gilt, wenn eine Startadresse eingegeben wird, die größer ist als die bisherige Endadresse: hier wird die Endadresse so angepaßt, daß die Länge des alten angezeigten Speicherbereiches erhalten bleibt; hier erhält die neue Endadresse also den Wert

min (max. Arbeitsspeicheradr., neue_Startadresse + Länge_des_alten_Bereichs).

6.4 Graphikoberfläche für 4-RK-MI

Abbi. 6.3 zeigt die Bildschirm-Darstellung der Benutzerschnittstelle für die graphische Oberfläche der 4-RK-MI.

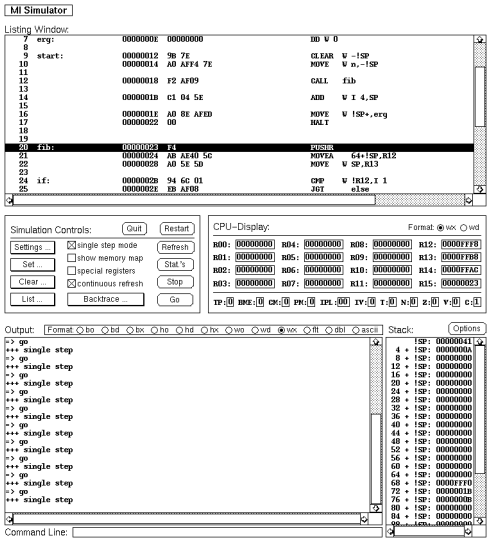


Abbildung 6.2: Bildschirm-Darstellung der Benutzerschnittstelle für die graphische Oberfläche der 1-RK-MI.

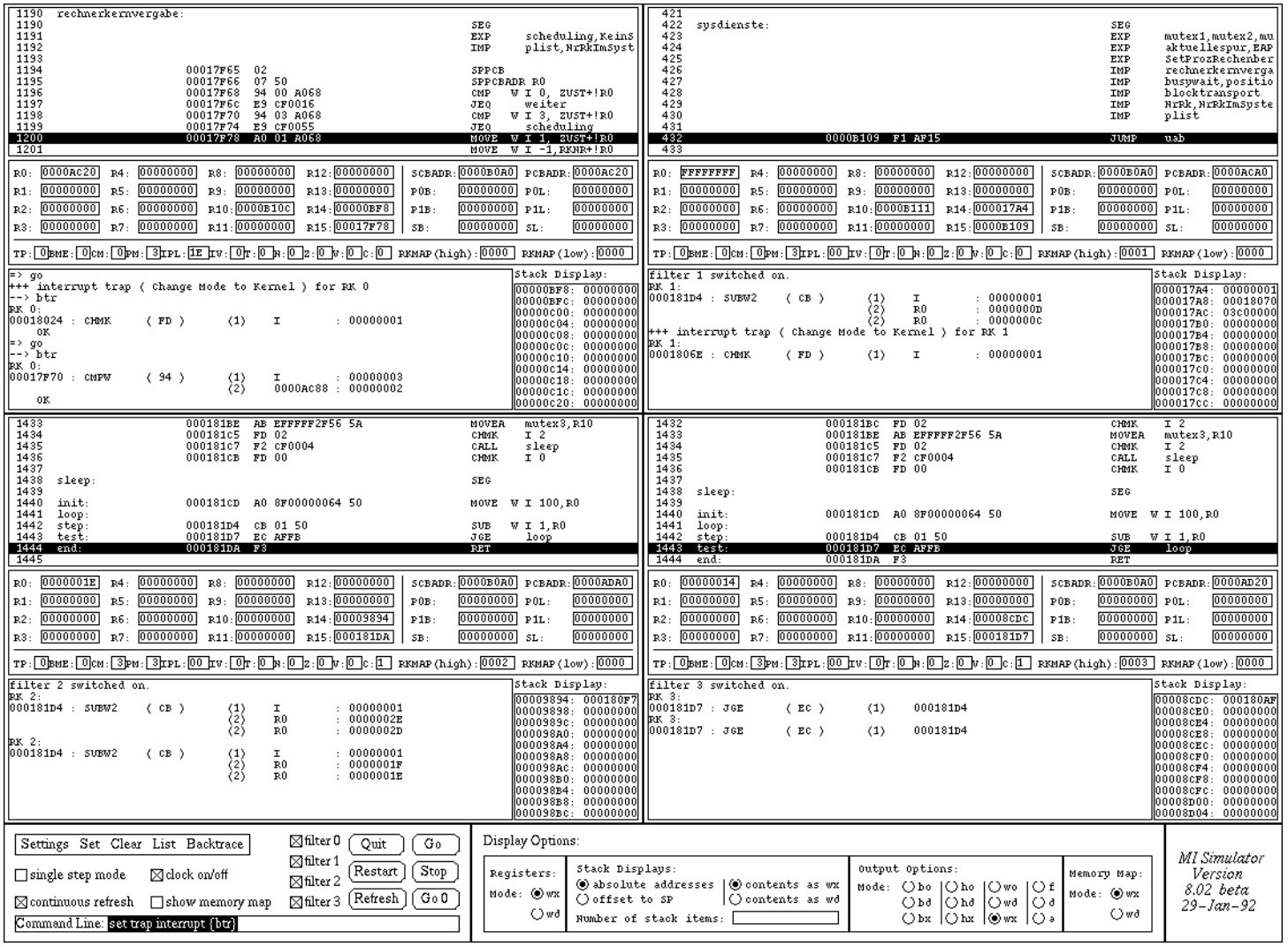


Abbildung 6.3: Bildschirm-Darstellung der Benutzerschnittstelle für die graphische Oberfläche der 4-RK-MI.

Anhang A

Die Grammatik der Assemblersprache für die Modellmaschine

<**Assemblerprogramm**> ::= { }$^{1,\infty}$ <tanw> **END**

<**tanw**> ::= { <etanw> | <Kommentar> }$^{1,\infty}$

<**etanw**> ::= **;** | >>neue Zeile<<

<**Kommentar**> ::= **- -** >> beliebige Zeichenfolge ohne <etanw>-Zeichen << <etanw>

<**Segment**> ::= $\Big\{$<Segmentname> **:** $\{$<tanw>$\}^{0,1}\Big\}^{0,1}$ **SEG** { ⊔ <Ablageadresse> }^{0,1}
<tanw> $\Big\{\{$<Marke>$\}^{0,\infty}$ <Anweisung> <tanw>$\Big\}^{0,\infty}$

<**Segmentname**> ::= <Name>

<**Ablageadresse**> ::= <vorzeichenlose ganze Zahl>

<**Marke**> ::= <Name> **:** { <tanw> }^{0,1}

<**Anweisung**> ::= <Assemblersteuerung>
| <Maschinenbefehl>
| <Datendefinition>

<**Assemblersteuerung**> ::= <Importanweisung>
| <Exportanweisung>
| <Gleichsetzung>
| <Reservierung>
| <Ausrichtung>

<**Importanweisung**> ::= { **IMP** | **IMPORT** } ⊔ $\Big\{$**ALL** | $\{$<Importname>$\}_{,}^{1,\infty}\Big\}$

<**Importname**> ::= <Segmentname>.<Name> | <Name>

<**Exportanweisung**> ::= { **EXP** | **EXPORT** } ⊔ $\left\{\textbf{ALL} \mid \{\text{<Name>}\}_{,}^{1,\infty}\right\}$

<**Gleichsetzung**> ::= { **EQU** | **EQUAL** } ⊔ { <Name> **=** <Ersetzungstext> }$_{,}^{1,\infty}$

<**Ersetzungstext**> ::= >> beliebige Zeichenfolge ohne <tanw>-Zeichen und
Komma <<

<**Reservierung**> ::= { **RES** | **RESERVE** } ⊔ <vorzeichenlose ganze Zahl>

<**Ausrichtung**> ::= { **ALI** | **ALIGN** } ⊔ <vorzeichenlose ganze Zahl>

<**Maschinenbefehl**> ::= <Operationsteil>
| <Operationsteil> ⊔ { <Operandenspezifikation> }$_{,}^{1,4}$

<**Operationsteil**> ::= <Transportbefehl>
| <bitweise boolesche Operation>
| <arithmetische Operation>
| <Vergleichsbefehl>
| <Schiebebefehl>
| <Sprungbefehl>
| <Bitfeldbefehl>
| <Synchronisationsbefehl>
| <Systemaufrufbefehl>
| <Rechnerkernalarm>
| <privilegierter Befehl>
| <Prozessorstatusbefehl>

<**Transportbefehl**> ::= **MOVE** ⊔ <bhwfd>
| **CLEAR** ⊔ <bhwfd>
| **MOVEN** ⊔ <bhwfd>
| **MOVEC** ⊔ <bhw>
| **MOVEA**
| **CONV**
| **PUSHR**
| **POPR**

<**bhwfd**> ::= **B** | **H** | **W** | **F** | **D**

<**bhw**> ::= **B** | **H** | **W**

<**bitweise boolesche Operation**> ::= **OR** ⊔ <bhw>
| **ANDNOT** ⊔ <bhw>
| **XOR** ⊔ <bhw>

<**arithmetische Operation**> ::= **ADD** ⊔ <bhwfd>
| **SUB** ⊔ <bhwfd>
| **MULT** ⊔ <bhwfd>
| **DIV** ⊔ <bhwfd>

<**Vergleichsbefehl**> ::= **CMP** ⊔ <bhwfd>

<**Schiebebefehl**> ::= **SH** | **ROT**

<**Sprungbefehl**> ::= **JUMP** | **J**<Bedingung> | **CALL** | **RET**

<**Bedingung**> ::= **EQ** | **NE** | **GT** | **GE** | **LT** | **LE** | **C** | **NC** | **V** | **NV**

<**Bitfeldbefehl**> ::= **EXTS** | **EXT** | **INS** | **FINDC** | **FINDS**

<**Synchronisationsbefehl**> ::= **JBSSI** | **JBCCI**

<**Systemaufrufbefehl**> ::= **CHMK**

<**Rechnerkernalarm**> ::= **RKALARM**

<**privilegierter Befehl**> ::= **SPPCB** | **LPCB** | **REI** | **HALT**
| **SP**<Sonderregister> | **L**<Sonderregister>

<**Sonderregister**> ::= **IPL** | **RKMAP (MAPEN)** | **SB** | **SL** | **SCBADR** | **PCBADR**

<**Prozessorstatusbefehl**> ::= **SBPSW** | **LBPSW**

<**Operandenspezifikation**> ::= <absolute Adresse>
| <immediater Operand>
| <Registeradressierung>
| <relative Adressierung>
| <indirekte Adressierung>
| <indizierte relative Adressierung>
| <indizierte indirekte Adressierung>
| <Kelleradressierung>

<**absolute Adresse**> ::= <Ausdruck für ganze Zahl> | { <Name> | <Importname> }
{ <vz> <Summand> }$^{0,\infty}$

<**Ausdruck für ganze Zahl**> ::= { <vz> }^{0,1} { <Summand> }$^{1,\infty}_{<vz>}$

<**vz**> ::= + | -

<**Summand**> ::= <vorzeichenlose ganze Zahl> | '<Zeichen>'

<**immediater Operand**> ::= **I** ⊔ <Operand>

<**Operand**> ::= <Gleitpunktzahl> | <Ausdruck für ganze Zahl>

<**Registeradressierung**> ::= <**R**x>

<**Rx**> ::= **R0** | **R1** | **R2** | **R3** | **R4** | **R5** | **R6** | **R7** | **R8** | **R9** | **R10** | **R11** | **R12** | **R13**
| **R14** | **R15** | **SP** | **PC**

<**relative Adressierung**> ::= { <Ausdruck für ganze Zahl> **+** }^{0,1} **!** <**R**x>

<**indizierte relative Adressierung**> ::= <relative Adressierung> <Indexangabe>

<**Indexangabe**> ::= **/** <**R**x> **/**

<**indirekte Adressierung**> ::= **!** (<relative Adressierung>) | **! !** <**R**x>

<**indizierte indirekte Adressierung**> ::= <indirekte Adressierung> <Indexangabe>

<**Kelleradressierung**> ::= **–!**<**R**x> | **!**<**R**x>**+**

<**Datendefinition**> ::= { **DD** ⊔ }^{0,1} <Datengruppe>

<**Datengruppe**> ::= $\left\{\{\text{<bhwfd>}\ \sqcup\}^{0,1}\ \text{<Datenelement>}\right\}_{,}^{1,\infty}$

<**Datenelement**> ::= <absolute Adresse>
| <Gleitpunktzahl>
| <String>
| (<Datengruppe>) * <vorzeichenlose ganze Zahl>

<**Name**> ::= <Buchstabe> { <Buchstabe> | <Dezimalziffer> | **_** }$^{0,\infty}$

<**vorzeichenlose ganze Zahl**> ::= <Binaerzahl>
| <Dezimalzahl>
| <Hexadekadische Zahl>

<**Gleitpunktzahl**> ::= { <vz> }^{0,1} <Dezimalzahl> { **.**<Dezimalzahl> }^{0,1}
$\left\{\mathbf{E}\ \{\text{<vz>}\}^{0,1}\ \text{<Dezimalzahl>}\right\}^{0,1}$

<**Binaerzahl**> ::= **B** '<Binaerziffernfolge>'

<**Binaerziffernfolge**> ::= { <Binaerziffer> }$^{1,\infty}$

<**Dezimalzahl**> ::= <Dezimalziffernfolge>

<**Dezimalziffernfolge**> ::= { <Dezimalziffer> }$^{1,\infty}$

<**Hexadekadische Zahl**> ::= **H** '<Hexadekadische Ziffernfolge>'

<**Hexadekadische Ziffernfolge**> ::= { <Hexadekadische Ziffer> }$^{1,\infty}$

<**String**> ::= '<Zeichenfolge>'

<**Zeichenfolge**> ::= { <Zeichen> }$^{1,\infty}$

Anmerkung: Tritt in einem String das Zeichen " ' " auf, so muß es doppelt angegeben werden.

<**Zeichen**> ::= <Buchstabe> | <Dezimalziffer> | <Sonderzeichen>

<**Buchstabe**> ::= **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z** | **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**

<**Binaerziffer**> ::= **0** | **1**

<**Dezimalziffer**> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

<**Hexadekadische Ziffer**> ::= <Dezimalziffer> | **A** | **B** | **C** | **D** | **E** | **F**

<**Sonderzeichen**> ::= >> beliebiges Zeichen der Tastatur außer <Buchstabe> und <Dezimalziffer> <<

Anhang B

Die Liste der Maschinenbefehle

Befehlsname	Datentyp	Operandenzahl	opcode
HALT	-	0	00
REI	-	0	01
SPPCB	-	0	02
LPCB	-	0	03
SPSB	Wort	1	04
SPSL	Wort	1	05
SPSCBADR	Wort	1	06
SPPCBADR	Wort	1	07
SPIPL	Wort	1	08
SPRKMAP	Wort	1	09
SPMAPEN	Wort	1	09
LSB	Wort	1	0A
LSL	Wort	1	0B
LSCBADR	Wort	1	0C
LPCBADR	Wort	1	0D
LIPL	Wort	1	0E
LRKMAP	Wort	1	0F
LMAPEN	Wort	1	0F

Tabelle B.1: Liste der privilegierten Maschinenbefehle.

Befehlsname	Datentyp	Operandenzahl	opcode
RKALARM	Wort	1	91
CMP B	Byte	2	92
CMP H	Halbwort	2	93
CMP W	Wort	2	94
CMP F	Float	2	95
CMP D	Double	2	96
JV	Wort	1	97
JNV	Wort	1	98
CLEAR B	Byte	1	99
CLEAR H	Halbwort	1	9A
CLEAR W	Wort	1	9B
CLEAR F	Float	1	9C
CLEAR D	Double	1	9D
MOVE B	Byte	2	9E
MOVE H	Halbwort	2	9F
MOVE W	Wort	2	A0
MOVE F	Float	2	A1
MOVE D	Double	2	A2
MOVEN B	Byte	2	A3
MOVEN H	Halbwort	2	A4
MOVEN W	Wort	2	A5
MOVEN F	Float	2	A6
MOVEN D	Double	2	A7
MOVEC B	Byte	2	A8
MOVEC H	Halbwort	2	A9
MOVEC W	Wort	2	AA
MOVEA	Wort	2	AB
CONV	Byte/Wort	2	AC
OR B	Byte	2	AD
OR H	Halbwort	2	AE
OR W	Wort	2	AF
OR B	Byte	3	B0
OR H	Halbwort	3	B1
OR W	Wort	3	B2

Tabelle B.2: Liste der nicht-privilegierten Maschinenbefehle (Teil 1)

Befehlsname	Datentyp	Operandenzahl	opcode
ANDNOT B	Byte	2	B3
ANDNOT H	Halbwort	2	B4
ANDNOT W	Wort	2	B5
ANDNOT B	Byte	3	B6
ANDNOT H	Halbwort	3	B7
ANDNOT W	Wort	3	B8
XOR B	Byte	2	B9
XOR H	Halbwort	2	BA
XOR W	Wort	2	BB
XOR B	Byte	3	BC
XOR H	Halbwort	3	BD
XOR W	Wort	3	BE
ADD B	Byte	2	BF
ADD H	Halbwort	2	C0
ADD W	Wort	2	C1
ADD F	Float	2	C2
ADD D	Double	2	C3
ADD B	Byte	3	C4
ADD H	Halbwort	3	C5
ADD W	Wort	3	C6
ADD F	Float	3	C7
ADD D	Double	3	C8
SUB B	Byte	2	C9
SUB H	Halbwort	2	CA
SUB W	Wort	2	CB
SUB F	Float	2	CC
SUB D	Double	2	CD
SUB B	Byte	3	CE
SUB H	Halbwort	3	CF
SUB W	Wort	3	D0
SUB F	Float	3	D1
SUB D	Double	3	D2

Tabelle B.3: Liste der nicht-privilegierten Maschinenbefehle (Teil 2)

Befehlsname	Datentyp	Operandenzahl	opcode
MULT B	Byte	2	D3
MULT H	Halbwort	2	D4
MULT W	Wort	2	D5
MULT F	Float	2	D6
MULT D	Double	2	D7
MULT B	Byte	3	D8
MULT H	Halbwort	3	D9
MULT W	Wort	3	DA
MULT F	Float	3	DB
MULT D	Double	3	DC
DIV B	Byte	2	DD
DIV H	Halbwort	2	DE
DIV W	Wort	2	DF
DIV F	Float	2	E0
DIV D	Double	2	E1
DIV B	Byte	3	E2
DIV H	Halbwort	3	E3
DIV W	Wort	3	E4
DIV F	Float	3	E5
DIV D	Double	3	E6
SH	Wort	3	E7
ROT	Wort	3	E8
JEQ	Wort	1	E9
JNE	Wort	1	EA
JGT	Wort	1	EB
JGE	Wort	1	EC
JLT	Wort	1	ED
JLE	Wort	1	EE
JC	Wort	1	EF
JNC	Wort	1	F0
JUMP	Wort	1	F1
CALL	Wort	1	F2
RET	-	0	F3
PUSHR	-	0	F4
POPR	-	0	F5

Tabelle B.4: Liste der nicht-privilegierten Maschinenbefehle (Teil 3)

Befehlsname	Datentyp	Operandenzahl	opcode
EXTS	Wort	4	F6
EXT	Wort	4	F7
INS	Wort	4	F8
FINDS	Wort	4	F9
FINDC	Wort	4	FA
JBSSI	Wort	3	FB
JBCCI	Wort	3	FC
CHMK	Wort	1	FD
SBPSW	Byte	1	FE
LBPSW	Byte	1	FF

Tabelle B.5: Liste der nicht-privilegierten Maschinenbefehle (Teil 4)

Anmerkung: Um weitgehende Aufwärtskompatibilität (von der 1-Rechnerkern-MI zur 4-Rechnerkern-MI) zu ermöglichen, sind die Befehle LMAPEN und SPMAPEN Synonyme für LRKMAP und SPRKMAP.

Anhang C

Abkürzungen der Kommandos für die Testhilfe und ASCII-Tabelle

Zum Erlernen und Einprägen der Testhilfe-Kommandos ist es sicher günstig, die Befehle in voller Länge auszuschreiben. Bei längerem Arbeiten mit dem MI-Simulator kann dies aber äußerst lästig werden. Deshalb sind für die einzelnen Kommandos folgende Abkürzungen zulässig:

sbp: set breakpoint

cbp: clear breakpoint

lbp: list breakpoint

sti: set trap instruction

cti: clear trap instruction

lti: list trap instruction

stf, sts, stw: set trap <type>

ctf, cts, ctw: clear trap <type>

ltf, lts, ltw: list trap <type>

stint: set trap interrupt

ctint: clear trap interrupt

ltint: list trap interrupt

btr: backtrace

sss: set single step

css: clear single step

lss: list single step

serr: set error level

lerr: list error level

cerr: clear error level

msg: message

Darüberhinaus kann "display" und "modify" ganz weggelassen werden; es genügt also Adressen- oder Registerangabe.

Beispiel: `R10; SP; PC`
Ausgabe der Register R10, R14 und R15 im aktuell eingestellten Format.

Beispiel: `1000 = FFFFFFFF/wx`
Ab Adresse 1000 wird ein "Wort" mit Wert -1 abgelegt.

ASCII-Tabelle

Die nachfolgende Tabelle enthält alle ASCII-Zeichen bzw. ASCII-Steuerzeichen.
Die korrespondierenden ASCII-Werte sind dabei hexadekadisch angegeben.

Zeichen	ASCII	Zeichen	ASCII	Zeichen	ASCII	Zeichen	ASCII
NUL	00	SOH	01	STX	02	ETX	03
EOT	04	ENQ	05	ACK	06	BEL	07
BS	08	HT	09	LF	0A	VT	0B
FF	0C	CR	0D	SO	0E	SI	0F
DLE	10	DC1	11	DC2	12	DC3	13
DC4	14	NAK	15	SYN	16	ETB	17
CAN	18	EM	19	SUB	1A	ESC	1B
FS	1C	GS	1D	RS	1E	US	1F
SP	20	!	21	”	22	#	23
S	24	%	25	&	26	/	27
(	28	)	29	*	2A	+	2B
,	2C	-	2D	.	2E	/	2F
0	30	1	31	2	32	3	33
4	34	5	35	6	36	7	37
8	38	9	39	:	3A	;	3B
<	3C	=	3D	>	3E	?	3F
@	40	A	41	B	42	C	43
D	44	E	45	F	46	G	47
H	48	I	49	J	4A	K	4B
L	4C	M	4D	N	4E	O	4F
P	50	Q	51	R	52	S	53
T	54	U	55	V	56	W	57
X	58	Y	59	Z	5A	[	5B
\	5C	]	5D	∧	5E	-	5F
\	60	a	61	b	62	c	63
d	64	e	65	f	66	g	67
h	68	i	69	j	6A	k	6B
l	6C	m	6D	n	6E	o	6F
p	70	q	71	r	72	s	73
t	74	u	75	v	76	w	77
x	78	y	79	z	7A	{	7B
|	7C	}	7D	∼	7E	DEL	7F

Tabelle C.1: ASCII-Tabelle.