PROJEKTGRUPPE: SIMULATOR DOKUMENTATION

Jörg Lilienbecker
Frank Stratmann
Michael Frericks

23. Juni 1995

Inhaltsverzeichnis

1 Der Simulator

Dieses Programm simuliert einen Von-Neumann-Rechner mit einem Befehlssatz, der in der Vorlesung COMPILERBAU I/II im WS93/94 bzw. SS94 (Prof. W.M. Lippe) festgelegt wurde. Eine genaue Übersicht über die Befehle geben wir unter 5.
Es wird eine CPU mit folgenden Komponenten simuliert:

- Akkumulator
  Der Akkumulator kann sowohl Real- wie auch Integerwerte enthalten. Der Akkumulator ist das Register mit dem fast alle Datenmanipulationen ausgeführt werden. Das Ergebnis solcher Manipulationen steht dann wieder im Akkumulator.
  Z.B. ADA 3 : Zum Inhalt des Akkus wird der Inhalt der Datenspeicherzelle 3 addiert.

- Indexregister
  Diese werden für die indizierte Adressierung benötigt.
  Z.B. ADA 3 1 (bzw. ADA 3, 1): Zum Inhalt des Akkus wird der Inhalt der Datenspeicherzelle mit der Nummer (3 + Inhalt vom Indexregister 1) addiert. Die Indexregister können daher nur Integer- Zahlen enthalten.

- Befehlszählregister
  Dieses Register enthält die Programmspeicheradresse des gerade bearbeiteten Befehls. Es wird bei Sprungbefehlen direkt beschrieben, bei anderen Befehlen wird es nach deren Ausführung um 1 erhöht.

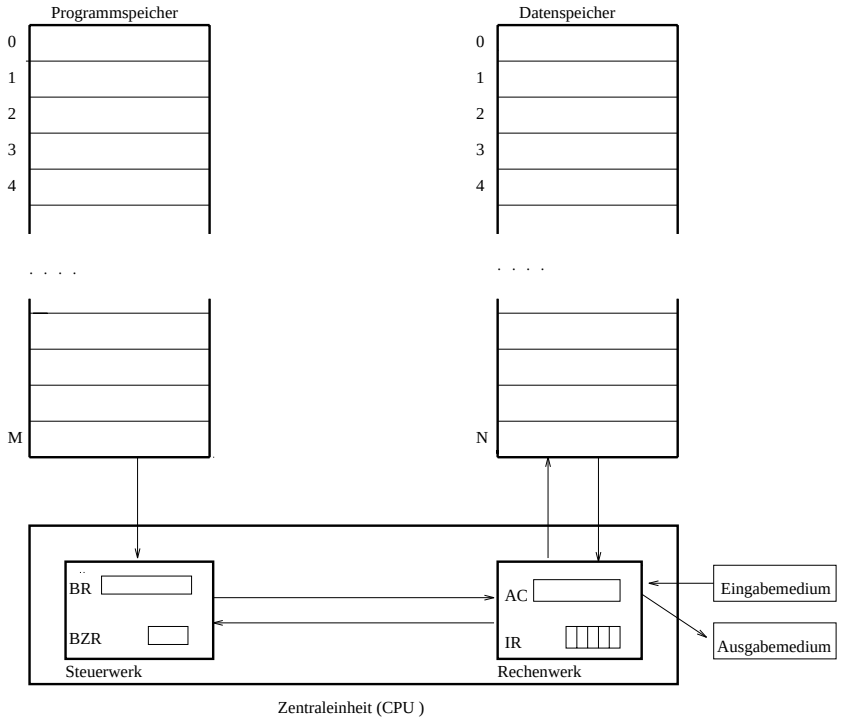
- Befehlsregister
  Dieses Register enthält den aktuellen Befehl

Weitere Bestandteile des Simulators sind der Programmspeicher und der Datenspeicher. Der Programmspeicher beginnt mit Adresse 0. Das eingelesene Programm

wird ab Zelle 0 aufsteigend ohne Lücken im Programmspeicher abgelegt (siehe 3). Die Zellen des Datenspeichers sind ähnlich wie der Akkumulator aufgebaut. Sie können ebenfalls Real- bzw. Integer-Zahlen enthalten. Zusätzlich gibt es noch ein sogenanntes Substitutionsbit, welches bei der indirekten Adressierung eine Rolle spielt.

Die Größe des Daten/Programmspeichers und die Anzahl der Indexregister kann durch Eingabe von "sim -h" ermittelt werden.

Prinzipieller Aufbau:



2 Aufruf des Simulators

Der Simulator wird unter Unix wie folgt aufgerufen:

sim [-b] [-d] [-h] [-m] [-bdhm] file

Dabei ist *sim* der Name des Simulator-Programms. Die Angaben in eckigen Klammer sind optional und haben folgende Bedeutung:

OPTIONEN:

-b : Breakpoint setzen. Hier können mehrere Programmspeicheradressen angegeben werden, an denen die Programmausführung angehalten wird und der Inhalt der Register bzw. eines anzugebenen Datenspeicherbereichs ausgegeben wird.

-d : Ausgabe der Registerinhalte nach jedem Programmschritt.

-h : Ausgabe eines Hilfetextes.

-m : *Muß* gesetzt werden, wenn Programme, die der MMS–Compiler übersetzt hat, simuliert werden. Sollte *nicht* gesetzt werden, wenn es sich um kein compiliertes Programm handelt.

3 Format der Programmdatei

Das Von-Neumann-Programm sollte ohne Leerzeilen und ohne Zeilennummern in der Datei stehen. Die einzelnen Zeilen dieser Datei sollten das folgende Format haben:

1. Das Format einer Zeile in der Programmdatei:
   i.) Operator
   ii.) Leerzeichen
   iii.) Operand
   iv.) Leerzeichen oder Komma
   v.) Indexregister, optional
   vi.) ",I" oder ",i" : direkt hinter dem letzten Operanden; nur bei den STA-Befehlen. Dann wird beim Speichern das Substitutionsbit gesetzt.
2. Kommentare in der Programmdatei werden durch # oder ; eingeleitet. Sie gelten bis ans Ende der jeweiligen Zeile.
3. Groß/Klein-Schreibung des Operators spielt keine Rolle.

Das Programm wird im Programmspeicher ab Zelle 0 abgelegt! Dies muß der Programmierer bei Sprungbefehlen beachten!!

4 MMS–Programme

Der Simulator enthält ein Laufzeitsystem für die Münsteraner–Mini–Sprache (MMS). Es ist ein Compiler verfügbar, der diese Sprache in Von–Neumann–Code übersetzt. Das Laufzeitsystem wird aber nicht in diesen Von–Neumann–Code eingebaut, sondern ist im Simulator implementiert.
Informationen über die MMS finden sich im Skript der Vorlesung Compilerbau II im SS94 von Prof. W.M. Lippe.

5 Befehlsübersicht

Wir geben nun eine Übersicht über die Befehle, die der Simulator verarbeiten kann. Zur Tabelle sei noch bemerkt:

- Mit AC, IR_{indx} bzw. $\mathrm{IR}_{\mathrm{IR}_{indx}}$ ist der Inhalt des Akkumulators, des Indexregisters mit der Nummer *indx* bzw. des Indexregisters mit der Nummer Inhalt des Indexregisters *indx* gemeint.

- Werden Boolsche Werte eingegeben oder verarbeitet, so steht 0 für FALSE und Integer–Zahlen $\neq 0$ für TRUE.

- Die Sterne in der Spalte Operandentypen geben an, daß der beteiligte Operand diesen Typ annehmen darf. Steht dort kein Sternchen, so darf der Operand diesen Typ *nicht* annehmen.
  Z.B ADA adr hat nur in der Spalte "Integer" ein '*'. Also muß der Inhalt der Datenspeicherzelle 10 *und* der Inhalt des Akkumulators eine Integer–Zahl sein.

- Ist ein '*' in der Spalte "indirekt" gesetzt, so kann durch Angabe von ",I" hinter dem Befehlswort indirekte Adressierung erreicht werden.
  Z.B STA,I 10 bewirkt, falls das Substitutionsbit nicht gesetzt ist, daß der Inhalt von Datenspeicherzelle 10 als Adresse interpretiert wird, in die der Inhalt des Akkumulators abgespeichert wird. Ist nun aber in der Zelle 10 das Substitutionsbit gesetzt, so übernimmt der Inhalt der Zelle 10 die Rolle der Zahl 10, d.h. in diesem Fall ist LDA,I 10 als LDA,I Inhalt(10) zu lesen. Dies wird solange wiederholt, bis das Substitutionsbit nicht mehr gesetzt ist. Die indirekte Adressierung mit Berücksichtigung eines Substitutionsbits wird mehrfach indirekte Adressierung genannt.

- Ein Sternchen in der Spalte "indiziert" erlaubt Indexregistermodifikationen durch Angabe eines Indexregisters hinter dem Operanden durch ein Leerzeichen oder ein Komma getrennt.
  Z.B. STA 10 3 bewirkt, daß der Inhalt des Akkumulators in die Zelle mit der Nummer 10+Inhalt(Indexregister 3) abgespeichert wird.

- Ist sowohl indizierte als auch indirekte Adressierung zugelassen, dann wird bei der Kombination von beiden zuerst das Indexregister berücksichtigt und dann die indirekte Adressierung.

Befehl	Wirkung	Adressierungsarten		Operandentypen	
		indirekt	indiziert	Real	Integer
LDA adr	AC:=Inh(adr)	*	*	*	*
STA adr	Inh(adr):=AC	*	*	*	*
ENA zahl	AC:=zahl				*
FNA zahl	AC:=zahl			*	
ADA adr	AC:=AC+Inh(adr)	*	*		*
SBA adr	AC:=AC–Inh(adr)	*	*		*
MUA adr	AC:=AC*Inh(adr)	*	*		*
DVA adr	AC:=AC/Inh(adr)	*	*		*
MDA adr	AC:=AC mod Inh(adr)	*	*		*
NGA	AC:=–AC				*
FAD adr	AC:=AC+Inh(adr)	*	*	*	
FSB adr	AC:=AC–Inh(adr)	*	*	*	
FMU adr	AC:=AC*Inh(adr)	*	*	*	
FDV adr	AC:=AC/Inh(adr)	*	*	*	
FNG	AC:=–AC			*	
AND adr	AC:=AC AND Inh(adr)	*	*		*
OR adr	AC:=AC OR Inh(adr)	*	*		*
AUT adr	AC:=AC EOR Inh(adr)	*	*		*
NOT	AC:=NOT(AC)				*
INR	AC:=Eingabe; real			*	
INI	AC:=Eingabe; integer				*
INB	AC:=Eingabe; boolean				*
OUR	Ausgabe AC; real			*	
OUI	Ausgabe AC; integer				*
OUB	Ausgabe AC; boolean				*
IR	AC:=float(AC)				*
RI	AC:=trunc(AC)			*	
ENTIER	AC:=float(trunc(AC))			*	
UJP adr	BZR:=adr	*			*
SRJ adr	Aufr. einer Laufzeitr.				*
AZJ,GR adr	BZR:=adr, falls AC> 0			*	*
AZJ,LS adr	BZR:=adr, falls AC< 0			*	*
AZJ,EQ adr	BZR:=adr, falls AC=0			*	*

Befehl	Wirkung	Adress.art.		Operandent.	
		indir.	indiz.	Real	Int
AZJ,NE adr	BZR:=adr, falls AC$\neq 0$			*	*
AZJ,GE adr	BZR:=adr, falls AC≥ 0			*	*
AZJ,LE adr	BZR:=adr, falls AC≤ 0			*	*
STP	Programmende				
HLT	wie STP mit Meldung				
NOP	No Operation				
LDI indx adr	IR_{indx}:=Inh(adr)				*
STI indx adr	Inh(adr):=IR_{indx}				*
ENI indx zahl	IR_{indx}:=zahl				*
ADI indx adr	IR_{indx}:=IR_{indx}+Inh(adr)				*
SBI indx adr	IR_{indx}:=IR_{indx}–Inh(adr)				*
MUI indx adr	IR_{indx}:=IR_{indx}*Inh(adr)				*
DVI indx adr	IR_{indx}:=IR_{indx}/Inh(adr)				*
MDI indx adr	IR_{indx}:=IR_{indx} mod Inh(adr)				*
NGI	IR_{indx}:=–IR_{indx}				*
ENAI indx	AC:=IR_{indx}				*
ENIA indx	IR_{indx}:=AC				*
ENAI,I indx	AC:=$IR_{IR_{indx}}$				*
ENIA,I indx	$IR_{IR_{indx}}$:=AC				*
UJPI indx	BZR:=IR_{indx}				*
ADAI indx	AC:=AC+IR_{indx}				*
SBAI indx	AC:=AC–IR_{indx}				*
MUAI indx	AC:=AC*IR_{indx}				*
DVAI indx	AC:=AC/IR_{indx}				*
MDAI indx	AC:=AC mod IR_{indx}				*

6 Schnittstellen

Die Schnittstelle zum Compiler ist die Programmdatei, die der Compiler erzeugt. Das Format dieser Datei ist unter 3 beschrieben.
Die Schnittstelle zum Laufzeitszstem ist die Funktion 'Laufzeitszstem':

```
int Laufzeitsystem(  char                 programmspeicher[][],
                     DATENSPEICHERZELLE   datenspeicher[],
                     long int             ir[],
                     AKKUMULATOR          *akku,
                     long int             rk,
                     int                  aufruf_nummer,
                     long int             *Sprung,
                     int                  *fehler);
```

Diese Funktion wird zur Initialisierung des Laufzeitsystems und bei SRJ- bzw. UJP-Befehlen, die negative Adressen als Operand haben, aufgerufen. Dabei wird bei der Initialisierung -1 und bei den SRJ- bzw. UJP-Befehlen die negative Adresse als `aufruf_nummer` übergeben. Die genaue Bedeutung der Aufrufparameter der Funktion 'Laufzeitsystem' ist in der Dokumentation zum Laufzeitsystem nachzulesen. Die Datentypen, die diese Funktion benutzt, finden sich in der Header-Datei 'sim.h'.

```
typedef enum
{
     INTEGER, REAL, DOUBLEINT
} ZELLENTYP;

typedef struct
{
     int integer1;
     int integer2;
} DOUBLEINTTYP;

typedef union
{
 double        real;
 int           integer;
 DOUBLEINTTYP  doubleint;
}ZELLE;

typedef struct
{
 int        substitutionsbit;  /* 1 = gesetzt; 0 = nicht gesetzt */
 ZELLENTYP  typ;               /* REAL oder INTEGER */
 ZELLE      inhalt;
} DATENSPEICHERZELLE;

typedef struct
{
 ZELLENTYP  typ;     /* REAL oder INTEGER */
 ZELLE      inhalt;
} AKKUMULATOR;
```