RETII-Assembler

Andreas Hofmeister & Patrick Stiegeler

8. August 1998

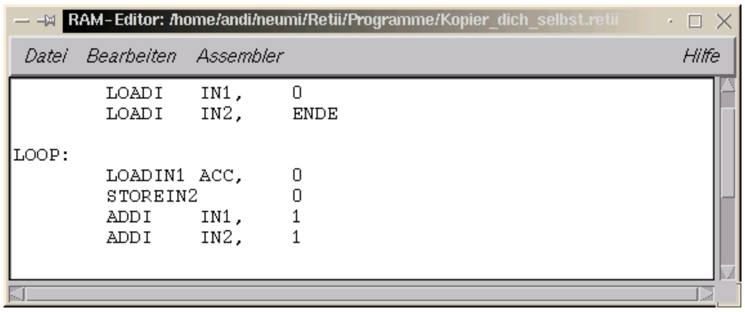
Zusammenfassung

Eine Beschreibung des Assemblers für die Simulation/Animation der RETII-CPU

Inhaltsverzeichnis

1 Ein Programm Assemblieren

Das Fenster des Editors für den Assembler erscheint beim Klicken auf das RAM der Animation. In diesem Fenster kann man nun entweder ein Assembler-Programm eintippen oder man kann über das Datei-Menü eine Textdatei laden.



[1]

Mit dem Menü “RAM->Assembler” wird der Text im Editor assembliert und, falls kein Fehler aufgetreten ist, in das RAM der Animation geladen. Das Programm kann jederzeit in das RAM geladen werden, sogar wenn die Animation läuft.

Tritt während dem Assemblieren ein Fehler auf, wird der Inhalt des RAM nicht verändert und es erscheint ein Fenster in dem die einzelnen Fehlermeldungen aufgeführt sind. Wählt man eine Fehlermeldung in der Liste aus so wird der Eingabe-Cursor des Editors auf die entsprechende Stelle des Quellcodes gesetzt.

Mit dem Menüpunkt “speichern” oder “speichern als” im Datei-Menü kann man den aktuellen Text im Editor in einer Datei speichern.

2 Aufbau der Assembler-Befehle

Jede Anweisung eines Assembler-Programms besteht aus einer Sprungmarke, einem Assembler-Befehl mit seinen Argumenten und aus einem Kommentar, wobei jeder dieser Teile auch weggelassen werden kann.. Ein Anweisung endet immer am Ende der Zeile.

2.1 Sprungmarken

Eine Sprungmarke, auch “Label” genannt, ist so etwas ähnliches wie eine Variable, man kann sie als Argument für einen Befehl verwenden, insbesondere als Ziel bei einem Sprungbefehl.

Eine Sprungmarke besteht aus einem Bezeichner gefolgt von einem Doppelpunkt, wobei zwischen dem Bezeichner und dem Doppelpunkt beliebig viele Leerzeichen stehen dürfen. Dem Bezeichner wird als Wert die Adresse, an der der folgende Befehl beginnt, zugewiesen.

Für manche Befehle (etwa der JUMP-Befehl der RETII-CPU) haben Sprungmarken eine besondere Bedeutung, eine Sprungmarke wird dazu vom Assembler besonders gekennzeichnet. Wird eine Sprungmarke in einem Ausdruck verwendet, dann wird das Ergebnis ebenfalls wie eine Sprungmarke behandelt.

[1] Das Bild zeigt das Editor-Fenster bei Verwendung des KDE-Windowmanager unter Linux

2.2 Befehle

Ein Befehl besteht aus einem Bezeichner (dem Namen des Befehls) und den Argumenten des Befehls. Welche Befehle vorhanden sind, wird im wesentlichen von der CPU bestimmt die in der Animation verwendet wird, es gibt jedoch auch Befehle die vom Assembler definiert werden und damit für alle CPUs vorhanden sind. Einem Assembler-Befehl kann eine Liste von Argumenten folgen, deren einzelne Elemente durch Kommata getrennt sind. Anzahl und Typ der Argumente ist vom Befehl abhängig. Die Allgemeine Form eines Befehls ist also

Befehl [Argument1 [, Argument2] ...]

2.2.1 Datentypen

Der Assembler unterscheidet zwischen folgenden Datentypen

Typ	Schreibweise	Bemerkung
Integer (Ganzzahl)	[1-9]*	Angabe als Dezimalzahl
	0x[1-9,A-F,a-f]*	Angabe als Hexadezimalzahl, C-Notation
	[1-9,A-F,a-f]*h	Angabe als Hexadezimalzahl, Postfix-Notation
	[1-9,A-F,a-f]*o	Angabe als Oktalzahl (zur Basis 8), Postfix-Notation
	[0,1]*b	Angabe als Binär-Zahl, Postfix-Notation
Zeichenkette	"[.]*"	
Bezeichner	[A-Z,a-z][0-9,A-Z,a-z _]*	
Register		Sind Bezeichner und werden von der CPU bestimmt.
Bedingungen		Sind ebenfalls Bezeichner, die von der CPU bestimmt werden

2.2.2 Arithmetische Ausdrücke

Anstelle einer Integer-Zahl kann auch ein arithmetischer Ausdruck stehen. Der Ausdruck wird beim Übersetzen des Programms ausgewertet. Allen Bezeichnern (mit Ausnahme der Sprungmarken) in einem Ausdruck muß vorher eine Integer-Zahl als Wert zugewiesen werden.

In einem Ausdruck stehen folgende Operatoren zur Verfügung :

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Integer-Division
%	Modulo
&	Bitweises Und
|	Bitweises Oder
ˆ	Bitweises Exklusiv-Oder

Unärer Operator	Bedeutung
+	0 + Ausdruck
-	0 - Ausdruck
!	Bitweise Negation

Ein Ausdruck kann folgende Form haben

<Bezeichner>
<Integer-Zahl>
(<Ausdruck>)
<Unärer Operator> <Ausdruck>
<Ausdruck> <Operator> <Ausdruck>

2.2.3 Bezeichner

Im allgemeinen muß einem Bezeichner ein Wert zugewiesen werden, bevor er verwendet wird. Eine Ausnahme bilden hier Sprungmarken, diese kann man auch verwenden bevor sie im Programm auftauchen.

Einige Bezeichner sind vordefiniert, etwa die Assembler Befehle oder die Bedingungen für Sprünge. Diese Bezeichner können nicht neu definiert werden oder in Sprungmarken verwendet werden.

2.3 Kommentare

Ein Kommentar beginnt mit einem Semikolon und endet am Ende der Zeile. Innerhalb des Kommentars sind beliebige Zeichen erlaubt, man kann also Programmzeilen auskommentieren.

3 Befehle des Assemblers

Der Assembler selbst definiert drei Befehle, die unabhängig von der CPU zur Verfügung stehen.

3.1 der DW-Befehl

Mit dem DW-Befehl können beliebige Integer-Zahlen in den erzeugten Code eingefügt werden. Als Argumente muß mindestens eine Integer Zahl angegeben werden. Wird mehr als eine Zahl angegeben, so erscheinen die Zahlen in der Reihenfolge im erzeugten Code, in der sie im Quelltext angegeben werden. Die Syntax für den DW-Befehl ist

DW <Wert> [,<Wert> ...]

3.2 der "define"-Befehl

Der "define"-Befehl erzeugt keinen Code, man kann ihn verwenden um einem Bezeichner einen Wert zuzuweisen. Dem Bezeichner darf vorher aber kein Wert zugewiesen sein. Die Syntax für den "define"-Befehl ist

define <Bezeichner> <Wert>

Der Wert darf jeden Datentyp annehmen, man kann zum Beispiel einem Register einen weiteren Namen zuweisen. Der "define"-Befehl kann nicht dazu benutzt einen weiteren Namen für einen Befehl zu vereinbaren.

3.3 der “undefine”-Befehl

Auch der “undefine”-Befehl erzeugt keinen Code, man verwendet ihn um eine mit “define” gemachte Zuordnung von einem Bezeichner zu einem Wert wieder aufzuheben. Die Syntax für den “undefine”-Befehl ist

undefine <Bezeichner>

Möchte man mit “undefine” eine Sprungmarke zu löschen, sollte man Beachten daß sich in diesem Fall das Löschen nur auf die Befehle hinter dem “undefine” auswirkt, ein Befehl vor dem “undefine” kann eine Sprungmarke nach dem “undefine” Verwenden. Ein Beispiel :

```
Anfang:    JUMP Ende       ; funktioniert, Ende ist noch definiert \
           undefine Anfang \
           undefine Ende \
Ende:      JUMP Anfang     ; Fehler, Anfang ist nicht mehr definiert
```

4 Befehle der RETII-CPU

4.1 Register

Als Register definiert die RETII-CPU die Bezeichner ACC, PC, IN1 und IN2 . Die Definition der Registernamen kann nicht geändert werden, man kann allerdings einen zusätzlichen Namen für ein Register vereinbaren.

4.2 Bedingungen

Für die Bedingung bei den Sprungbefehlen definiert die RETII-CPU folgende Bezeichner

Bezeichner	Bedeutung
lt	kleiner (< 0)
le	kleiner oder gleich (≤ 0)
eq	gleich ($= 0$)
ne	ungleich ($\neq 0$)
ge	größer oder gleich (≥ 0)
gt	größer (>0)

4.3 Befehle der RETII-CPU

In der folgenden Tabelle bedeutet eine Zahl in spitzen Klammern (z.B. <i>) , daß der Befehl die Zahl als vorzeichenlose Zahl verwendet, steht die Zahl in eckigen Klammern (z.B. [i]) wird die Zahl als vorzeichenbehaftete Zahl im 2’er Komplement interpretiert.

Alle Integer-Zahlen die bei RETII-Befehlen verwendet werden sind 24Bit-Zahlen.

Load-Befehle	
Befehl	Wirkung
LOAD <Register reg>, <Integer i>	reg := RAM(<i>)
LOADI <Register reg>, <Integer i>	reg := [i]
LOADIN1 <Register reg>, <Integer i>	reg := RAM(<IN1> + [i])
LOADIN2 <Register reg>, <Integer i>	reg := RAM(<IN2> + [i])
Store-Befehle	
Befehl	Wirkung
STORE <Integer>	RAM(<i>) := ACC
STOREIN1 <Integer>	RAM(<IN1>+[i]) := ACC
STOREIN2 <Integer>	RAM(<IN2>+[i]) := ACC
MOVE <Register src>, <Register dest>	dest := src
Compute-Befehle	
Befehl	Wirkung
SUBI <Register reg>, <Integer i>	[reg] := [reg] - [i]
ADDI <Register reg>, <Integer i>	[reg] := [reg]+ [i]
OPLUSI <Register reg>, <Integer i>	reg := reg⊕ <i>
ORI <Register reg>, <Integer i>	reg := reg∨ <i>
ANDI <Register reg>, <Integer i>	reg := reg∧ <i>
SUB <Register reg>, <Integer i>	[reg] := [reg] - [RAM(<i>)]
ADD <Register reg>, <Integer i>	[reg] := [reg]+ [RAM(<i>)]
OPLUS <Register reg>, <Integer i>	[reg] := [reg]⊕RAM(<i>)
OR <Register reg>, <Integer i>	[reg] := [reg]∨RAM(<i>)
AND <Register reg>, <Integer i>	[reg] := [reg]∧ RAM(<i>)
Jump-Befehle	
Befehl	Wirkung
NOP	-
JUMP gt , <Integer i> JUMP eq , <Integer i> JUMP ge , <Integer i> JUMP lt , <Integer i> JUMP ne, <Integer i> JUMP le , <Integer i>	<PC> := <PC>+[i] falls [ACC] <Bedingung> 0 erfüllt <PC> := <PC>+1 sonst

4.3.1 JUMP-Befehle und Sprungmarken

Wie aus der Tabelle hervorgeht, sind alle JUMP-Befehle relative Sprünge. Damit man nun aber eine Sprungmarke als Ziel des JUMP-Befehls verwenden kann, wird in bei der Verwendung von Sprungmarken als Argument nicht deren Wert eingesetzt. Stattdessen wird das Argument berechnet <Adresse des JUMP-Befehls> - <Inhalt der Sprungmarke>.

Beispiel :

```
marke1:     NOP                 ; der 1. Jump Befehl springt hierhin \
            define blub, 0      ; blub und marke1 haben den Wert 0, \
                                ; aber ... \
            JUMP marke1         ; ... springt zum NOP-Befehl \
            JUMP blub           ; ... ist eine Endlosschleife
```

Falls eine Sprungmarke in einem arithmetischen Ausdruck verwendet wird, wird das Ergebnis des Ausdrucks ebenfalls wie eine Sprungmarke behandelt , so daß folgendes funktioniert:

```
            NOP \
            JUMP marke2 + 1 \
marke2:     NOP                 ; marke2 hat den Wert 2 \
            NOP                 ; Ziel des Sprungs
```

Bei allen anderen Befehlen wird der Wert der Sprungmarke direkt eingesetzt.

4.3.2 Arithmetische Ausdrücke und 24-Bit Argumente

Bei allen RETII-Befehlen ist das Integer Argument nur 24 Bit lang, andererseits verwendet der Assembler intern 32-Bit Zahlen. Der Assembler prüft darum für die Integer Zahl i die Bedingung $2^{24} > i > -2^{24}$ und liefert eine Fehlermeldung falls diese Bedingung verletzt wird.

Anhang A : Ein Beispielprogramm

Das folgende Beispiel-Programm für die RETII-CPU kopiert sich selbst an eine neue Stelle im RAM und führt dann das Programm an der neuen Position aus.

```
; \
; Ein Programm, das sich selbst kopiert, und dann diese \
; Kopie ausführt. \
; \
START:  LOADI IN1,      START           ; Anfangsadress in IN1 laden \
LOOP:   LOADIN1 ACC,    0               ; Wort auf das IN1 zeigt \
                                        ; in ACC laden \
        STOREIN1        ENDE - START    ; ... und an das Ziel kopieren \
        ADDI IN1,       1               ; IN1 weiterzälen \
        JUMP eq , ENDE + 1              ; falls das letzte kopierte \
                                        ; Wort eine 0 war, die Kopie \
                                        ; ausführen ... \
        JUMP LOOP                       ; ... ansonsten weiterkopieren \
        DW 0                            ; Marker für Programmende \
ENDE:                                   ; Ziel der Kopieraktion
```

Anhang B: Assembler Syntax

Der Assembler ist so gestaltet, daß sich ein anderer/neuer Befehlssatz relativ einfach einbauen läßt. Die Auswertung der Argumente eines Befehls geschieht daher außerhalb der eigentlichen Syntax.

```
programm        ::=    befehls_zeile \
                     | programm NEW_LINE befehls_zeile \
\
befehls_zeile   ::=\
                     | befehl \
                     | label befehl \
                     | label \
\
label           ::=    IDENT DPUNKT \
\
befehl          ::=    BEFEHL   argumente \
                     | DEFINE   IDENT ausdruck \
                     | UNDEFINE IDENT \
\
argumente       ::=  \
                     | ausdruck \
                     | argumente KOMMA ausdruck \
\
ausdruck        ::=    IDENT \
                     | INTEGER \
                     | EKLAMMER_AUF indirekt EKLAMMER_ZU \
                     | ausdruck PLUS ausdruck \
                     | ausdruck MINUS ausdruck \
                     | ausdruck MAL ausdruck \
                     | ausdruck GETEILT ausdruck \
                     | ausdruck MODULO ausdruck \
                     | KLAMMER_AUF ausdruck KLAMMER_ZU \
                     | MINUS ausdruck \
                     | PLUS  ausdruck \
\
indirekt        ::=    IDENT PLUS  ausdruck \
                     | IDENT MINUS ausdruck \
                     | ausdruck
```