Verwendung der virtuellen Maschine UMach

19. Januar 2013

Inhaltsverzeichnis

1 Einführung

1.1 Ein Beispiel

Gleich am Anfang soll ein Beispiel für die Verwendung der virtuellen Maschine UMach und des Assemblers gegeben werden.

Wir haben ein UMach-Programm in eine normale Textdatei geschrieben. Das Programm kann sich über mehrere Dateien erstrecken, hier verwenden wir nur eine Datei. Das Programm sieht wie folgt aus:

Listing 1: Ein einfaches Beispiel

```
    set r1 3
loop:
    dec r1
    cmp r1 zero
    be  finish
    jmp loop
finish:
    EOP
```

Dieses Programm wurde in der Datei `prog1.uasm` gespeichert (die Endung der Datei ist eigentlich egal). Wir gehen davon aus, dass der Assembler `uasm`, die Programmdatei `prog1.uasm` und die virtuelle Maschine `umach` sich in dem selben Verzeichnis befinden. Sonst muss man die Befehle entsprechend anpassen.

Das Programm kann wie folgt assembliert werden:

```
./uasm -o prog.ux prog1.uasm
```

Die Option `-o` gibt die Ausgabedatei an. Ohne diese Option wird das assemblierte Programm in die Datei `u.out` geschrieben. Ergebnis des assemblierens ist also die Datei `prog.ux`. Jetzt kann man diese Datei „ausführen“:

```
./umach prog.ux
```

Das Programm beendet sich ohne Ausgabe. Starten wir also das Programm im Debug-Modus (Option `-d`):

```
./umach -d prog.ux
```

Es wird die erste Anweisung angezeigt, der aktuelle Program Counter (Inhalt des Registers `PC`) und ein Prompt, der auf eine Eingabe von uns wartet. So könnte es weiter gehen:

```
[256]   SET   R1    3
umdb > show reg r1
R1 = 0x00000000 = 0
umdb > s
[260]   DEC   R1
umdb > show reg r1
R1 = 0x00000003 = 3
umdb > s
[264]   CMP   R1    ZERO
umdb > s
[268]   BE    2
umdb > s
[272]   JMP   -3
umdb > s
[260]   DEC   R1
umdb > s
[264]   CMP   R1    ZERO
umdb > show reg r1
R1 = 0x00000001 = 1
umdb > s
[268]   BE    2
umdb > s
[272]   JMP   -3
umdb > s
[260]   DEC   R1
umdb > s
[264]   CMP   R1    ZERO
umdb > s
[268]   BE    2
umdb > s
[276]   EOP
umdb > s
umdb > s
The maschine is not running.
umdb > show reg r1 cmpr
R1 = 0x00000000 = 0
CMPR = 0x00000000 = 0
umdb > q
```

2 Ausführen

Das Programm `umach` ist die Implementierung der virtuellen Maschine. Es kann sein, dass man es zuerst kompilieren muss. Für die Kompilierung wird im Quellenverzeichnis ein Makefile zur Verfügung gestellt. Wechselt man in das Quellenverzeichnis, so kann man das Programm mit dem folgenden Befehl kompilieren:

```
make
```

Das Programm wurde auf Linux getestet.

Das `umach` Programm kann wie folgt ausgeführt werden:

```
./umach [optionen] programdatei.umx
```

Dabei wir die Datei `programdatei.umx` ausgeführt. Die Optionen sind unten erläutert.

2.1 Optionen

Bei der Ausführung eines UMach-Programms ist es möglich, Optionen anzugeben, die die Ausführung in bestimmter Weise beeinflussen. Alle Optionen sind optional und werden mit Standardwerten initialisiert. Es stehen die folgenden Optionen zur Verfügung:

-m Mit „-m *z*“, wobei „*z*“ eine positive ganze Zahl ist, kann der Arbeitsspeicher der virtuellen Maschine auf *z* Bytes festgelegt werden. „./umach myProg -m 4096“ startet das Programm „myProg“ mit einem Arbeitsspeicher von 4096 Bytes. Ist die „-m“ Option bei der Ausführung nicht angegeben, so wird das Programm mit 2048 Bytes Arbeitsspeicher gestartet.

-d Diese Option veranlasst die Maschine im „debug“-Modus zu starten. Hierbei wird der eingebaute Debugger gestartet, nicht ein externer. Siehe den Abschnitt 4 ab der Seite 19.

-s Disassembliert das angegebene Programm und zeigt das Ergebnis in der Konsole an. Unter Verwendung der Option -x werden alle nummerischen Werte im Hexadezimalsystem angezeigt.

-x Die Darstellung von Zahlenwerten in der Ausgabe des „debug“-Modus, oder beim Disassemblieren, wird nach Angabe dieser Option hexadezimal sein.

-v Setzt den „verbosity level“ auf „warnings“ (Warnungen und Fehler werden ausgegeben). Unter zweifacher Anwendung der „-v“-Option, entweder „-v -v“, oder „-vv“, wird der „verbosity level“ auf „info“ gestellt (allgemeine Informationen, Warnungen und Fehler werden ausgegeben). Ist die Option nicht angegeben, so ist der „verbosity level“ standardmäßig auf „error“ gesetzt (nur Fehler werden ausgegeben).

3 Assembler

3.1 Eingabe Dateien

Es können beliebig viele Assemblercode-Dateien angegeben werden. Sie werden der Reihe nach assembliert. Man beachte, dass die Instruktion `EOP` das Ende des Programms signalisiert und die Virtuelle Maschine anhält, falls dieser Befehl erreicht wird.

Im Assemblercode wird nicht zwischen Groß- und Kleinschreibung unterschieden, wobei Namen von Sprungmarken und Variablen eine Ausnahme darstellen.

Siehe den Abschnitt 3.4.2 ab der Seite 13 für ein Beispiel, in dem mehrere Assemblercode-Dateien für ein Programm verwendet werden.

3.2 Sprungmarken

Der UMach-Assembler unterstützt die Verwendung von sogenannten Sprungmarken. Eine Sprungmarke markiert das Sprungziel für die verschiedenen Sprungbefehle.

Um eine Sprungmarke im Programmcode zu definieren, schreibt man den Namen der Sprungmarke, gefolgt von einem Doppelpunkt. Namen von Sprungmarken dürfen keine Whitespaces (Leerzeichen oder Tabulatoren) beinhalten.

Folgen von Sprungmarken, die auf den selben Befehl zeigen, werden unterstützt. Dabei muss jede Sprungmarke in einer eigenen Zeile definiert sein. Sprungmarken zeigen immer auf den ersten Befehl der nach der Definition der Sprungmarke auftaucht. Das Programm 2 zeigt ein Beispiel für die Verwendung von (mehrfachen) Sprungmarken.

Listing 2: Beispiel für Sprungmarken

```
set r1 5
loop:
do:
    cmp r1 zero
    be  finish
    dec r1
    jmp loop
    jmp do
finish:
end:
    EOP
```

In diesem Beispiel zeigen die Sprungmarken `loop` und `do` auf den gleichen Befehl: `cmp r1 zero`. Analog zeigen die Sprungmarken `finish` und `end` auf den Befehl `EOP`.

3.3 Variablen

Variablen werden nach der Anweisung `.data` angegeben. Diese Anweisung muss alleine in einer Zeile stehen.

Die Variablen können auf verschiedenen Assemblercode-Dateien verteilt werden, sie werden vom Assembler zusammengefügt und ans Ende der assemblierten Datei geschrieben.

Alle Variablen haben jeweils eine Länge, die ein Vielfaches von 4 Byte darstellt. Bedarf ein Datenelement weniger als $4 \times n$ Bytes an Speicherplatz, so wird es trotzdem auf eine durch 4 teilbare Länge mit Nullbytes gefüllt.

3.3.1 Strings

String Variablen werden mit der Anweisung `.string` definiert. Es folgt der Name der String Variable und abschließend der Initialisierungswert in doppelten Anführungszeichen. Siehe das Programm 3 für ein Beispiel.

String Variablen werden von dem Assembler automatisch mit einem Nullbyte terminiert.

3.3.2 Ganze Zahlen

Integer Variablen werden mit der Anweisung `.int` definiert. Es folgt der Name der Integer Variable und abschließend der Initialisierungswert. Dieser kann im Hexa-, Oktal- oder Dezimalsystem angegeben werden, analog zu der C oder Java Syntax (`0x` bzw. `0` Präfix).

Listing 3: Verwendung von Variablen

```
set   r1 dezimal # r1 = Adresse von 'dezimal'
lw    r1 r1      # r1 = Wert an der Adresse 'dezimal'

set   r2 hexa    # r2 = Adresse von 'hexa'
lw    r2 r2      # r2 = Wert an der Adresse 'hexa'
```

```
set  r3 oktal   # r3 = Adresse von 'oktal'
lw   r3 r3      # r3 = Wert an der Adresse 'oktal'

# hier haben r1, r2 und r3 den selben Wert

set  r1 str     # r1 = Adresse der Daten 'str'
set  r2 strsize # r2 = Adresse der Daten 'strsize'
sub  r2 r2 r1   # r2 = laenge der Daten 'str'

out  r1 r2 zero # Ausgabe "Hallo"

set  r1 nl      # r1 = Adresse von nl
addi r1 r1 3    # r1 zeigt auf das 4. Byte von nl
set  r2 1
out  r1 r2 zero


.data
### Datenbereich ###

.int dezimal 171   # dezimalsystem
.int hexa    0xAB  # hexadezimalsystem
.int oktal   0253  # oktalsystem

# String daten

.string str "Hallo Welt"
.int    strsize 0   # dummy Wert
.int    nl  0x0A    # new line
```

Angenommen, der Assembler `uasm`, die virtuelle Maschine `umach` und das Programm `example_data.uasm` befinden sich im aktuellen Arbeitsverzeichniss, kann das Programm 3 wie folgt assembliert und ausgeführt werden:

```
./uasm example_data.uasm
./umach u.out
```

Es wird lediglich „Hallo Welt“ ausgegeben.

3.4 Funktionen

Der UMach Assembler unterstützt indirekt die Verwendung von Funktionen.

3.4.1 Aufbau einer Funktion

Eine Funktion auf der Assembler-Ebene ist nichts anderes als eine Codesequenz, die mit einer Sprungmarke versehen ist und deren letzte Instruktion ein `RET` ist (RET wie "Return"). Zu diesem Code wird mit der Anweisung `CALL` gesprungen. Der Name der Sprungmarke ist auch Name der Funktion.

Die virtuelle Maschine UMach kennt keine Funktionen, sondern nur einzelne Maschinenbefehle. Sie kennt also auch keine Argumente, Gültigkeitsbereiche oder Rückgabewerte. All diese Elemente, welche in höheren Programmiersprachen eine Funktion (oder Methode) ausmachen, müssen also vom Programmierer selbst implementiert werden. Jeder Programmierer ist frei in der Entscheidung, wie er diese Elemente implementiert.

Um diese Entscheidungen zu erleichtern, kann man einige Konventionen befolgen. Eine solche Konvention wird in den folgenden Punkten gegeben. Allerdings steht es dem Benutzer frei, sich andere Konventionen und Regeln auszudenken.

Name der Funktion Der Name der Funktion wird als Sprungmarke angeben. Eine Funktion beginnt also wie folgt:

```
funktionsname:
    anweisung
    anweisung
    ...
```

Ende der Funktion Das Ende einer Funktion wird durch den Befehl `RET` markiert. Für mehr Informationen betreffend dieses Befehls siehe die Spezifikation der UMach Maschine (Abschnitt 3.8.3).

Argumente Die Argumente einer Funktion werden in umgekehrter Reihenfolge vor dem Aufruf auf den Stack abgelegt[1]. Innerhalb der Funktion werden die Argumente mithilfe des Registers `SP` (Stack Pointer) und eines Offsets abgelesen.

Möchte man z.B. eine Funktion namens `max` mit zwei Argumenten aufrufen und befindet sich das erste Argument in `R1` und das zweite in `R2`, kopiert man vor dem Aufruf diese zwei Argumente in umgekehrter Reihenfolge auf den Stack:

[1] Dies entspricht übrigens gängiger Praxis.

```
PUSH R2
PUSH R1
CALL max
```

In der Funktion werden die Adressen der Argumente anhand eines Offsets relativ zum Register SP berechnet und dann mit dem Befehl LW (Load Word) geladen. Dabei muss man folgendes beachten:

1. Durch den Befehl CALL wird die Rücksprungadresse auf den Stack gelegt. Das ergibt einen zusätzlichen Eintrag auf den Stack.

2. Jeder Stack Eintrag ist 4 Byte groß, unabhängig von dessen Inhalt.

Daraus ergibt sich beim Eintritt in die Funktion das folgende Stack-Layout:

Adresse	Inhalt
$\$SP + 0$	Rücksprungadresse
$\$SP + 4$	Erstes Argument
$\$SP + 8$	Zweites Argument
...	...
$\$SP + 4n$	n-tes Argument

Um an die zwei Argumente der max Funktion zu gelangen, kann man also folgendes schreiben:

```
ADDI R17 SP 4
LW   R17 R17

ADDI R18 SP 8
LW   R18 R18
```

Durch diesen Code stehen also die Argumente (aus R1/R2) nach Aufruf der Funktion max in R17/R18.

Rückgabewerte Es gibt viele Möglichkeiten, einen Wert aus einer Funktion zurückzugeben. Einige davon wären:

- Rückgabewerte in einem vorgeschriebenen oder vereinbarten Register speichern.

- Rückgabewerte auf dem Stack speichern, wobei der aufrufende Code vor dem Aufruf dafür sorgt, dass entsprechende Stack-Einträge reserviert werden (SP dekrementieren).

- Rückgabewerte auf dem Stack speichern, wobei die Funktion selber den Stack manipuliert.
- Rückgabewerte an vorgeschriebener oder vereinbarter Heap-Adresse speichern.
- etc.

Eine der gängigsten und effizientesten Methoden ist allerdings, den Wert in einem bestimmten Register abzulegen. Auch wir nutzen diese Methode und verwenden dafür das Register `R32`. Wird ein Wert zurückzugeben, kopiert der `RET` Befehl diesen in das Register `R32`.

Beispiel Folgend ein Beispiel für den Aufruf und die Implementierung einer Funktion. Es wird eine einfache `max` Funktion implementiert, welche die größere aus zwei Zahlen ermittelt. Die Rückgabe erfolgt im Register `R32`.

Diese Funktion arbeitet mit den Registern `R17` und `R18`, deshalb werden diese zuerst auf den Stack kopiert. Das ist für dieses Beispiel nicht zwingend notwendig, da der restliche Code diese Register nicht verwendet, aber die Technik soll vorgestellt werden. Danach erfolgt eine Berechnung der Stack-Adressen beider Argumente. Dazu ist es hilfreich, sich das Stack-Layout nach dem Kopieren der zwei Arbeitsregister zu veranschaulichen:

Adresse	Inhalt
$\$SP + 16$	Zweites Argument (66)
$\$SP + 12$	Erstes Argument (55)
$\$SP + 8$	Rücksprungadresse
$\$SP + 4$	R17
$\$SP + 0$	R18

```
# Example of implementing and calling
# a function.

SET  R1 55  # first argument
SET  R2 66  # second argument
            # push args in reverse order
PUSH R2     # push second arg
PUSH R1     # push first arg
CALL max    # call function
CP   R3 R32 # copy result from R32

EOP         # stop; don't go further
```

```
############# function max #############
# Arguments: two numbers                #
# Returns:   maximum number in R32      #
########################################
max:
    PUSH R17        # save work registers
    PUSH R18

    ADDI R17 SP 12  # address of first arg
    ADDI R18 SP 16  # address of second arg
    LW   R17 R17    # load first arg
    LW   R18 R18    # load second arg

    CMP  R17 R18    # compare args
    BG   max_first  # if (r17 > r18) r32 = 17
    CP   R32 R18    # else r32 = r18
    JMP  max_finish # ready

max_first:
    CP   R32 R17

max_finish:

    POP  R18        # restore work registers
    POP  R17        # (pop in reverse order)
    RET             # return
```

3.4.2 Ein größeres Beispiel

Das folgende Beispiel verdeutlicht die Verwendung von Funktionen anhand eines Programms, das die Länge eines Strings berechnet und die Länge ausgibt. Der String ist im Programm selbst eingebettet (Datensegment). Das Programm besteht aus mehreren Dateien, die jeweils eine Funktion implementieren:

1. prog2.uasm, enthält das Hauptprogramm (Programm 4).

2. strlen.uasm, enthält die Funktion `strlen`, welche die Länge eines Strings berechnet (Programm 5).

3. printint.uasm, enthält die Funktion `printint`, welche eine beliebige ganze Zahl ausgibt (Programm 6).

4. putchar.uasm, enthält die Funktion putchar, welche einen Buchstaben ausgibt (Programm 7).

Listing 4: Verwendung der Funktionen

```
## This program computes the string length
## of the embedded string and prints the
## length to port zero (terminal)

SET  R1 str
PUSH R1
CALL strlen
# string length in R32
PUSH R32
CALL printint

SET  R1 0X0A # new line
PUSH R1
CALL putchar
EOP

.DATA

.string str "Hello World!"
```

Listing 5: Funktion strlen

```
# strlen, function to compute the string length
# Arguments:    string adress (on stack)
#               the address is supposed to be found
#               on the stack just after the return
#               address
# Return value: R32 contains the string length
# Used registers: R17 to R20

strlen:
# we use register r17 to r20, so we save them first
    PUSH R17
    PUSH R18
    PUSH R19
    PUSH R20

# The first argument is on the stack,
# just after the return address,
# that is 4 bytes after SP. To reach that
# address, we must jump over what we have
# pushed and over the return address
    ADDI R17 SP 20
```

```
# argument goes into r18,
# this is the address of the string
    LW   R18 R17

# we store the character counter in r19
# and the current character in r20
    SET  R19 0
strlen_check:
    LW   R20 R18
    CMP  R20 ZERO       # end of string?
    BE   strlen_finish
    INC  R19            # counter++
    INC  R18            # next char
    JMP  strlen_check

strlen_finish:
    CP   R32 R19        # R32 = string length
    POP  R20            # restore saved registers
    POP  R19
    POP  R18
    POP  R17
    RET
```

Listing 6: Funktion printint

```
# printint, function to print an integer
# Arguments: integer to print (on stack)
# Returns:   nothing
# Registers used: R17, R18, R19

printint:
# save the registers we will work with
    PUSH R17
    PUSH R18
    PUSH R19
# set R17 to the stack address of the integer argument
# we have pushed 3 registers and after them there is
# the return address, so we increase SP with 3*4 + 4
# (stack entries are always 4 byte)
    ADDI R17 SP 16
    LW   R17 R17

# Registers we use:
# R17 current integer value, which we divide by 10
# R18 characters counter
# R19 division remainder
    SET R18 0
```

```
printint_convert:
    CMP  R17 ZERO
    BE   printint_printchars
    DIVI R17 10
    CP   R17 HI     # R17 = R17/10
    CP   R19 LO     # R19 = R17 mod 10
    ADDI R19 R19 48 # R19 = ascii value of R19
    PUSH R19        # push the ascii value of remainder
    INC  R18        # counter++
    JMP  printint_convert

# after having pushed the string representation
# of the integer argument, we call putchar to print
# all those characters. This will print them in the
# right order because now we go the stack backwards.
printint_printchars:
    CMP R18 ZERO       # more chars to print?
    BE printint_finish
    CALL putchar       # char already on stack, print it
    ADDI SP SP 4       # move stack pointer to next char
    DEC  R18           # counter--
    JMP printint_printchars

printint_finish:
# restore register values which we previously
# saved on the stack
    POP R19
    POP R18
    POP R17
    RET
```

Listing 7: Funktion putchar

```
# putchar, function to print one character
# Arguments: character to print (on stack)
# Returns: nothing
# Work registers: R17 and R18, which will be saved
# and restored to their previous values.

putchar:

# save registers R17 and R18 before use
    PUSH R17
    PUSH R18
# jump over the pushed registers and over the
# return address stored by the call command
# that is R17 = SP + (2 * 4 + 4)
```

```
    ADDI R17 SP 12
# jump over 3 bytes to the least significant
# byte of the argument (one character)
    ADDI R17 R17 3
# we print one single byte
    SET  R18 1
# output 1 byte from mem[R17] to port zero
    OUT  R17 R18 ZERO
# restore registers R17 und R18
    POP  R18
    POP  R17
    RET
```

Dieses Programm wird wie folgt assembliert:

```
./uasm prog2.uasm strlen.uasm printint.uasm putchar.uasm
```

Bemerkung: Es werden alle benötigten Dateien angegeben. Die Reihenfolge der Dateien, die eine Funktion definieren ist beliebig. Lediglich die „main“-Datei, welche den Startpunkt des Programms beinhaltet, muss an der ersten Stelle stehen.

Nach dem Assemblieren wird eine Datei namens `u.out` erzeugt, die den „Bytecode“ für die virtuelle Maschine beinhaltet. Man könnte auch die Option `-o` verwenden, um einen alternativen Dateinamen zu spezifizieren. Um diese Datei auszuführen, gibt man folgenden Befehl ein:

```
./umach u.out
```

Als Ergebnis erscheint die Fehlermeldung

```
ERROR: Cannot load 268 bytes of program into 512 bytes of
       memory
ERROR: Cannot load program file u.out.
Aborted
```

Das bedeutet, die virtuelle Maschine hat nicht genug Speicher um dieses Programm überhaupt laden zu können. 256 Bytes werden für die Interrupttabelle reserviert und die Maschine verwendet standardmäßig 512 Bytes für den Speicher. Man kann in diesem Fall mit der Option `-m` den Speicher größer spezifizieren. Als Beispiel Nutzen wir `-m 600`:

```
./umach -m 600 u.out
```

Das erhöht den Speicher auf 600 Bytes. Es wird dann „12“ ausgegeben.

Bemerkung: Es ist zu empfehlen, beim Start der Maschine die Option `-v` anzugeben. Zu viele PUSH-Befehle können bei unzureichenden Speicher einen Stack Overflow verursachen, was derzeit die Maschine zum Stillstand bringt – es sei denn, man programmiert einen Interrupt Handler für den Stack Overflow Interrupt. Der Stack Overflow wird als Warnung ausgegeben, welche nur mit der Option `-v` erscheinen.

4 Debuggen

Eingebaut in der UMach Maschine ist ein einfacher Debugger, der mit der Option „-d“ gestartet werden kann.

4.1 Debug-Befehle

Der Debugger unterstützt die folgenden Befehle:

1. step, oder s oder leere Eingabe. Die aktuelle Instruktion ausführen und die nächste laden.

2. run. Das Programm bis zum Ende ausführen. Nach dem Programmende wartet der Debugger auf weitere Befehle.

3. quit, oder q. Debugger beenden.

4. help, h oder ?. Hilfe anzeigen.

5. show reg <Registerliste>. Zeigt den Inhalt der spezifizierten Register.

6. show mem <Adresse>. Zeigt Speicherinhalt an der angegebenen Addresse.

7. show mem <start> <wieviel>. Zeigt <wieviel> Bytes Speicherinhalt ab der Adresse <start>.

8. show ins. Zeigt die aktuelle Instruktion.

5 Qt Debugger

5.1 Kompilieren und Installation

Zum Kompilieren und Verwenden der Software wird Linux empfohlen. Unter anderen Betriebssystemen wurde die Software nicht getestet.

Komponenten Um die Software unter Linux zu kompilieren sind folgende Komponenten erforderlich:

- Qt SDK in der Version 4.7 oder neuer.
- GNU Compiler Collection.
- Alle Abhängigkeiten der UMachVM

UMachCore *UMachCore* ist eine Abwandlung der *UMachVM* für die *UMachGUI*. Um den *UMachCore* zu kompilieren sind folgende Befehle im Quellverzeichnis aufzurufen:

```
qmake
make
```

UMachGUI Um die Oberfläche zu kompilieren sind folgende Befehle im Quellverzeichnis dieser aufzurufen:

```
qmake
make
```

Installation Um die Oberfläche im kompletten Funktionsumfang verwenden zu können müssen sich folgende weitere Applikationen im Anwendungs- bzw. Arbeitsverzeichnis befinden:

- UMachCore
- uasm (Version 2)

5.2 Übersicht über die Oberfläche

Die Oberfläche besteht aus einem Hauptfenster, in dem unter anderem der *Code-Editor* eingebettet ist, und weiteren Einstellungs- und Anzeigefenstern, die über das Menü angewählt werden können.

Hauptfenster Das Hauptfenster besteht aus einer Liste *Project Files*, in der alle zum aktuellen Projekt zugehörigen Dateien aufgelistet sind. Daneben befindet sich der *Code-Editor,* in dem in mehreren Tabs *Code*-Dateien geöffnet werden können. Dies geschieht mit einem Doppel-Klick auf die entsprechende Datei in der Liste. Die Datei-Liste enthält ein *Context*-Menü, mit dessen Hilfe Dateien zu einem Projekt hinzugefügt oder entfernt werden können. Weiterhin befindet sich im Hauptfenster eine Tabelle in der zur Programmlaufzeit alle Symbole, deren Typ und ihr Wert hinterlegt sind. Der Wert kann mit einem Doppelklick direkt in der Tabellenzelle manipuliert werden. Weiterhin befindet sich im Hauptfenster eine *Toolbar*, welche folgende Buttons zum steuern des Debuggens enthält (von Rechts nach Links):

- *Build & Run* - zur Ausführungszeit *Stop*
- *Goto next Breakpoint*
- *Goto next Instruction*

Menü Über das Menü können folgende Funktionalitäten erreicht werden:

- Anlegen, Öffnen, Speichern und Schließen Projektdateien.
- Hinzufügen von Assemblerdateien zum Projekt
- Beenden des Programms
- *Build & Run*

Unter dem Menü *Windows* findet man folgende Fenster:

- *Registers* - Zum Zugriff auf die Register der Maschine
- *Break Points* - Zum setzen von Haltepunkten
- *Options* - Einstellungen

Registers In diesem Fenster findet sich eine Tabelle zum überwachen der Register. Mit dem *Plus*-Butten kann das aktuell aus der *Drop-Down-Box* ausgewählte Register zur Überwachung hinzugefügt werden. Mit dem *Minus*-Button in Tabellenzeile des entsprechenden Registers, können diese wieder entfernt werden. Der aktuelle Registerinhalt wird interpretiert als Binärzahl, Hexadezimalzahl, Vorzeichenlose- und Vorzeichenbehaftete Zahl angezeigt.

Break Points In diesem Fenster können Haltepunkte gesetzt werden. In der letzten Zeile der Tabelle befindet sich eine Eingabemaske. Hier kann im Texteingabefeld Zeilennummer oder das Label angegeben werden. Daneben befindet sich eine *Drop-Down-Box* zur Auswahl der entsprechenden Quelldatei. Mit dem *Plus*-Button wird der Haltepunkt der Liste hinzugefügt. Mit dem *Minus*-Button in einer Tabellenzeile kann der entsprechende Haltepunkt wieder entfernt werden.

Options In diesem Fenster kann der Arbeitsspeicher der Maschine eingestellt werden. Mit *Apply & Close* werden Änderungen übernommen und das Fenster geschlossen. Mit *Cancel* werden diese verworfen.

5.3 Debuggen

Trifft die Maschine im Debug-Modus auf einen Haltepunkt, so wird die entsprechende Code-Zeile im *Code-Editor* angezeigt und rot hinterlegt. Nun können unter den Symbolen hinterlegte Daten modifiziert, Register angezeigt und manipuliert werden. Mit einem Klick auf den *Goto next Breakpoint*-Button wird die Ausführung bis zum Nächsten Haltepunkt fortgesetzt. Mit dem *Goto next Instruction*-Button wird nur die nächste Instruktion ausgeführt. Danach wird die Maschine wieder angehalten.

Listings