Verwendung der virtuellen UMach Maschine

2. Oktober 2012

Inhaltsverzeichnis

1 Einführung

1.1 Ein Beispiel

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

Wir haben ein UMach-Programm in eine normale Textdatei geschrieben. Das Programm kann sich über mehrere Dateien erstrecken, aber 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 Verzeichniss 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. Wird diese Option nicht angegeben, so wird das assemblierte Programm in die Datei `u.out` geschrieben. Ergebnis des Assemblers is also die Datei `prog.ux`. Jetzt kann man diese Datei „ausführen“:

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

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

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

Es wird die erste Anweisung angezeigt, der aktuelle Programm-Counter (Inhalt des Registers `PC`) und ein Prompt, der auf eine Eingabe von uns wartet. So könntes 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

2.1 Optionen

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Das hier ist der zweite Absatz. Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Und nun folgt – ob man es glaubt oder nicht – der dritte Absatz. Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Nach diesem vierten Absatz beginnen wir eine neue Zählung. Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben

enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

3 Assembler

3.1 Eingabe Dateien

Es können beliebig viele Programmdateien angegeben werden. Sie werden der Reiche nach abgearbeitet. Man beachte, dass die Instruktion `EOP` das Ende des Programms signalisiert. Falls nachher noch weitere Befehle, eventuell in anderen Datein, angegeben werden, werden diese zwar assembliert, aber nicht ausgeführt.

Alles außer Labels is case insensitive. Man kann beliebig Leerzeichen (whitespace) verwenden.

Siehe den Abschnitt 3.4.2 ab der Seite 15 für ein Beispiel, wo mehrere Dateien für ein Programm verwendet werden.

3.2 Labels

Der UMach-Assembler unterstüzt die Verwendung von sogenannten „Labels“, oder Sprungmarken. Ein Label markiert das Sprungziel für die verschiedenen Sprungbefehle.

Um ein Label im Programmcode zu definieren, schreibt man den Labelnamen, gefolgt von einem Doppelpunkt. Zwischen dem Labelnamen und dem Doppelpunkt können, außer das „newline“-Zeichen, beliebig viele Leerzeichen eingefügt werden.

Labelname ist höchstens 128 Buchstaben lang.

Labelnamen müssen selbst keine Leerzeichen (whitespace) beinhalten.

Ein Label muss immer von einem Befehl gefolgt werden, auch wenn dieser Befehl ein `NOP` ist. Labels, die von keinem Befehl gefolgt werden gelten als undefiniert und ein Sprung zu diesem Label ist ein Fehler. Beispiel:

```
jmp label
label:
```

Hier wird das Label `label` nicht definiert, da kein Befehl nach diesem Label folgt. Die Markierung des Datenbereichs gilt nicht als Definition eines Labels:

```
jmp label
label:
.data
```

Dieses Programm wird ebenfalls nicht assembliert.

Mehrfache Labels, die auf den selben Befehl zeigen, werden unterstüzt. Diese können beliebing im Code geschrieben werden, d.h. sie können auf verschiedenen Zeilen oder auf einer einzigen Zeile geschrieben werden. Mehrfache Labels zeigen jeweils auf den nächsten Befehl. Das Programm 2 zeigt ein Beispiel für die Verwendung der (mehrfachen) Labels.

Listing 2: Beispiel für Labels

```
set r1 5

loop  :
  do  :

 it:now:cmp r1 zero
    be  finish
    dec r1
    jmp loop
    jmp do

finish: end: EOP
```

In diesem Beispiel zeigen die Labels `loop`, `do`, `it` und `now` alle auf den selben Befehl: `cmp r1 zero`. Zu diesem Befehl wird gesprungen, indem man einen von diesen Labels verwendet. Hier bewirken beide Sprungbefehle `jmp loop` und `jmp do` einen Sprung zum selben `cmp`-Befehl. Die Labels `finish` und `end` zeigen auf den Befehl `EOP`. Man bemerkt auch die freie Verwendung der Leerzeichen (über die Lesbarkeit dieses Beispiels lässt sich diskutieren).

3.3 Programmdaten

Daten werden nach der Anweisung `.data` angegeben. Diese Anweisung muss alleine auf einer eigenen Zeile stehen.

Die Programmdaten können auch auf verschiedenen Programmdateien verteilt werden, sie werden vom Assembler zusammengefügt und ans Ende der assemblierten Datei eingefügt.

Alle Daten haben jeweils eine Länge, die ein Vielfaches von 4 Byte darstellt. Bedarft ein Datenelement weniger als $4k$ Bytes, so wird es trotzdem auf eine Länge von $4k$ mit Nullbytes gefüllt.

3.3.1 Strings

String Daten werden mit der Anweisung `.string` angegeben. Nach `.string` kommt der Name des Strings und dann der eigentlich String zwischen Anführungszeichen. Siehe das Programm 3 für ein Beispiel.

Strings werden so assembliert, dass sie immer ein Vielfaches von 4 Bytes belegen. Braucht der textuelle Inhalt des Strings weniger als 4 Byte, so wird der String mit Nullbytes gefüllt.

3.3.2 Ganze Zahlen

Ganze Zahlen werden mit der Anweisung `.int` angegeben. Nach `.int` folgt der Name (Label) der Zahl, dann die eigentliche Zahl. Diese kann in Hexa-, Oktal- oder Dezimalsystem angegeben werden, analog wie in der C/Java Sprache.

Listing 3: Verwendung der Daten

```
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 Verzeichniss, kann das Programm 3 wie folgt assembliert und ausgeführt werden:

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

Es wird lediglig „Hallo Welt“ ausgegeben.

3.4 Funktionen

Der UMach Assembler unterstützt die Verwendung von Funktionen.

3.4.1 Aufbau einer Funktion

Eine Funktion auf der Assembler-Ebene ist nichts anderes als ein Stück Code, das mit einem Label versehen ist und dessen letzte Instrution ein `RET` ist (von Return). Zu diesem Code wird mit der Anweisung `CALL` gesprungen. Der Label ist der Name der Funktion.

Die UMach virtuelle Maschine kennt keine Funktionen, sondern nur einzelne Maschinenbefehle. Sie kennt also auch keine Argumente, Parameter, Blöcke und Rückgabewerte. Alle diese Elementen, die in höheren Programmiersprachen eine Funktion (Methode) ausmachen, müssen also vom Programmierer selbst implementiert werden. Jeder Programmierer ist also frei in der Entscheidung, wie er diese Elementen implementiert, wie er überhaupt einem Codeabschnitt Argumente übergibt, wie er Rückgabewerte darstellt usw.

Um diese Entscheidungen zu erleichtern kann man einige Konventionen vorstellen, wie diese Bausteine einer Funktion zu implementieren sind. Eine solche Konvention wird in den folgenden Punkten gegeben. Man muss aber verstehen, dass es sich um eine

Konvention, bzw. um den Vorschlag einer Konvention handelt. Jeder ist frei sich andere Konventionen und Regeln auszudenken.

Name der Funktion Der Name der Funktion wird als Label angeben. Eine Funktion fängt also wie folgt an:

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

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

Argumente Die Argumente einer Funktion werden vor dem Aufruf auf den Stack abgelegt und zwar in umgekehrter Reihenfolge[1]. Innerhalb der Funktion werden die Argumente mit Hilfe 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`, so pusht man vor dem Aufruf diese zwei Argumente auf den Stack in umgekehrter Reihenfolge (damit sie die Funktion in richtiger Reihenfolge liest):

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

In der Funktion, werden die Adressen der Argumente anhand eines Offsets zum Register `SP` berechnet und dann mit dem Befehl `LW` (load word) geladen. Dazu muss man auf das folgende aufpassen:

1. Der Befehl `CALL`, mit dem man die Funktion aufruft, pusht die Rücksprungadresse aus der Funktion auf den Stack bevor er den Program Counter `PC` verändert. Das ergibt einen zusätzlichen Eintrag auf den Stack.

2. Jeder Stack Eintrag is 4 Byte groß, egal was da gespeichert wird.

Das ergibt beim Eintritt in die Funktion den folgenden Stack-Layout:

[1] Dies entspricht übrigens gängiger Praxis.

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

Um an die zwei Argumente von vorhin zu gelangen, wird also in der Funktion geschrieben:

```
ADDI  R17  SP  4
LW    R17  R17

ADDI  R18  SP  8
LW    R18  R18
```

Das speichert in das Register `R17` die Stack-Adresse, wo `R1` gepusht wurde und dann gleich aus dieser Adresse wieder in `R17` den Wert von `R1` geladen. Analog für `R2` und `R18`.

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

- Rückgabewert in einem vorgeschriebenen oder vereinbarten Register.
- Rückgabewert auf dem Stack, wobei der aufrufende Code vor dem Aufruf dafür sorgt, dass entsprechende Stack-Einträge reserviert werden (`SP` dekrementieren).
- Rückgabewert auf dem Stack, wobei die Funktion selber den Stack manipuliert.
- Rückgabe an vorgeschriebener oder vereinbarter Heap-Adresse.
- etc.

Eine der gängigsten und effizientesten Methoden ist allerdings, den Wert in einem bestimmten Register abzulegen. Für diese Methode haben wir uns auch entschieden, und zwar benutzen wir das Register `R32`, das letze Register[2]. Wird also einen Wert zurückzugeben, so wird er vor dem `RET` Befehl in das Register `R32` kopiert.

[2]Für x86 verwendet man das Register `eax` für Rückgabewerte.

Überschreiben der Register Es kann sehr schnell passieren, dass eine Funktion Register überschreibt, die von einer anderen Funktion verwendet werden. Verwendet z.B. eine Funktion die Register `R1` bis `R8` für ihre Berechnungen, und ruft sie irgendwann eine zweite Funktion auf, die die selben Register verwendet, so werden in der ersten Funktion die Register verändert, ohne dass sie davon überhaupt etwas merkt. Ein Beispiel dazu:

```
main:
   SET R1 5
   SET R2 6
   INC R1
   DEC R2
   CALL function
   RET

function:
   CP R3 R1
   CP R1 R2
   CP R2 R3
   RET
```

Hier verändert die Funktion `function` die Werte der Register `R1` und `R2`, die auch von der Funktion `main` verwendet werden. Diese Art von unfreiwilliger Überschreibung der Register führt sehr schnell zu schwer entdeckbaren Fehler. Daher verwenden wir eine Programmiertechnik, die dieses Problem aufheben sollte: wir überlegen uns vor (oder während) der Implementierung, welche Register wollen wir benutzen und gleich am Anfang der Funktion, pushen wir diese Register auf den Stack. Vor dem `RET` popen wir die Register wieder zurück (in umgekehrter Reihenfolge). Im nächsten Beispiel kann man diese Technik sehen.

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

Diese Funktion arbeitet mit den Registern `R17` und `R18`, also werden diese zuerst auf Stack gepusht. Das ist für dieses Beispiel nicht zwingend notwendig, da der restliche Code die Register nicht verwendet, aber die Technik soll vorgestellt werden. Danach erfolt eine Berechnung der Stack-Adressen, wo die zwei Argumente liegen. Dazu ist es hilfreich, sich das Stack-Layout nach dem Pushen der zwei Arbeitsregister klarzumachen:

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 dieses Länge ausgibt. Der String ist im Programm selbst einbebettet (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`, die die Länge eines Strings berechnet (Programm 5).

3. printint.uasm, enhält die Funktion `printint`, die eine ganze Zahle ausgibt (Programm 6).

4. putchar.uasm, enthält die Funktion `putchar`, die 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
```

Bemerkungen: es werden alle benötigten Dateien angegeben. Die Reihenfolge des Dateien, die eine Funktion definieren ist egal, die „main“-Datei aber, die den Startpunkt des Programms beinhaltet muss an der ersten Stelle sein, denn der Code in dieser Datei muss zuerst assembliert werden.

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. Möchte man diese Datei ausführen, so könnte man folgendes eingeben:

```
/umach u.out
```

Als Ergebniss bekommt man 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, z.B. so:

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

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

Noch eine Bemerkung: es ist zu empfehlen, beim Start der Maschine die Option `-v` anzugeben, denn die viele `PUSH`-Befehle können beim 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 und Warnungen werden erst mit der Option `-v` ausgegeben.

4 Debuggen

4.1 Debug-Befehle

Der Debugger unterstützt die folgenden Befehle:

1. step, oder s. Die aktuelle Instruktion ausführen und die nächste laden.
2. quit, oder q. Debugger beenden.
3. help, h oder ?. Hilfe anzeigen.
4. show reg <Registerliste>. Zeigt den Inhalt der spezifizierten Register.
5. show mem <adresse>. Zeigt Speicherinhalt an der angegebenen Addresse.
6. show mem <start> <wieviel>. Zeigt <wieviel> Bytes Speicherinhalt ab der Adresse <start>.
7. show ins. Zeigt die aktuelle Instruktion.

Listings

Index