Linux-Assembler Tutorial

Moritz Höppner

m-hoeppner@gmx.net

Inhalt

1. Grundlagen

1.1 Einführung

Maschinen verstehen keine menschliche Sprache, sie können überhaupt nichts mit einer für (die meisten) Menschen verständlichen Sprache anfangen. Deshalb gibt es sog. Compiler, die die Befehle von Menschen an die Maschine in eine für sie verständliche Form umwandeln (gennant kompilieren) sollen. Ein solcher Compiler kann natürlich nichts mit Deutsch, Englisch etc. anfangen, da die menschliche Sprache nie völlig eindeutig ist. Aus diesem Grund muss der Mensch eine bestimmte (100%ig eindeutige) Sprache lernen, die vom Compiler dann in Maschinensprache übersetzt werden kann. Beispiele für solche Sprachen sind C/C++, Pascal und Basic.

Die eben genannten Sprachen unterscheiden sich grundsätzlich von Assembler. Sie sind sogenannte Hochsprachen und können von den meisten Menschen mit einigermaßen guten Englischkenntnissen verstanden werden:

```
Integer zahl = 2
If zahl = 3 Then
     print "'zahl' ist gleich 3."
Endif
```

Was macht dieser Code wohl? ;)

Assemblerprogramme dagegen arbeiten auf einer anderen Ebene, nämlich direkt mit der Hardware. Das ist schon nicht mehr so leicht nachzuvollziehen. Dafür haben Assemblerprogramme einen entscheidenden Vorteil: Sie sind extrem klein und schnell und benötigen, wenn sie entsprechend programmiert sind, keine Software (nicht einmal ein Betriebssystem) um ausgeführt zu werden. Assembler wird also meist bei Programmen eingesetzt, die extrem schnell arbeiten sollen (Simulationen, Passwortcracker) oder sehr klein sein müssen (Viren).
Um Assembler-Programme in Maschinencode umzuwandeln, wird kein Compiler sondern ein *Assembler* benötigt. Der Vorgang heißt auch nicht kompilieren sondern *assemblieren*. Um evtl. benötigte andere Quellen oder Libarys in der endgültigen ausführbaren Datei zu haben, müssen die assemblierten Dateien noch *gelinkt* werden (vom **Überraschung** *Linker*).

1.2 Zahlensysteme

Normalerweise arbeiten wir mit dem Dezimalsystem - mit 10 verschiedenen Ziffern. Computer dagegen arbeiten mit einzelnen Bits. Ein Bit kann entweder 0 oder 1 sein, andere Ziffern gibt es also nicht. Dieses Zahlensystem wird Binärsystem, Dualsystem oder Basis 2 genannt.

Außerdem wird oft das Hexadezimalsystem oder Basis 16 verwendet. Es kennt außer 0-9 noch die Ziffern A-F. Dies hat den Vorteil, dass große Zahlen mit weniger Stellen dargestellt werden können.

Wichtige Begriffe

Ziffer

Die Ziffer ist das "Bild". Welche Bedeutung eine Ziffer hat, hängt vom Zahlensystem ab. Beispielsweise kann die Ziffer 1 im Dezimalsystem 1, 10, 100, ... bedeuten - je nach Stelle.

Ziffernwert

Der Ziffernwert ist die Bedeutung einer Ziffer unabhängig von ihrer Stelle. 0 bedeutet 0, 1 bedeutet 1, 2 bedeutet 2 usw. Der Ziffernwert ist unabhängig vom Zahlensystem immer gleich. Lediglich im Hexadezimalsystem gibt es noch weitere Ziffernwerte (A=10, B=11, ...).

Stellenwert

Der Stellenwert ist die eigentliche Bedeutung einer einzelnen Ziffer.
Beispiel (Dezimalsystem): 12 - hier hat die erste Ziffer den Stellenwert 10 und die zweite den Stellenwert 2.
Der Stellenwert einer Zahl kann mit folgender Formel ausgerechnet werden:

```
Stellenwert = Basis ^ (die wievielste Stelle von rechts minus 1) * Ziffernwert
```

Zahlenwert

Der Zahlenwert ist der Wert einer ganzan Zahl, also die Stellernwerte jeder Zahl mit einander addiert
.

Hexadezimalsystem

Das Hexadezimalsystem (Basis 16) ist für das Programmieren das wichtigste und gerade Assemblerprogrammierer sollten sich damit auskennen.
Es kennt neben den Ziffern 0-9 auch A-F. Die Ziffernwerte sind folgende:

Ziffer	Ziffernwert
0	0
1	1
...	...
A	10
B	11

C	12
D	13
E	14
F	15

Beispiel: 87A
`16 ^ 0 * 10 =` **`10`** (x^0 ergibt immer 1)
`16 ^ 1 * 7  =` **`112`**
`16 ^ 2 * 8  =` **`2048`**
87A bedeutet dezimal also 2170 (2048+112+10).

Die gleichen Regeln gelten für jedes andere Zahlensystem, ich werde jetzt nicht weiter darauf eingehen.

Für größere Zahlen sollte man sich so einen schönen Rechner wie kcalc runterladen, damit geht es wesentlich schneller, als das so im Kopf auszurechnen ;)

1.3 Register

Register sind das A und O von Assembler. Im Gegensatz zu den Hochsprachen werden in Assembler meist die Register und nicht der Hauptspeicher für das vorübergehende Speichern von Daten benutzt.

Was sind Register?

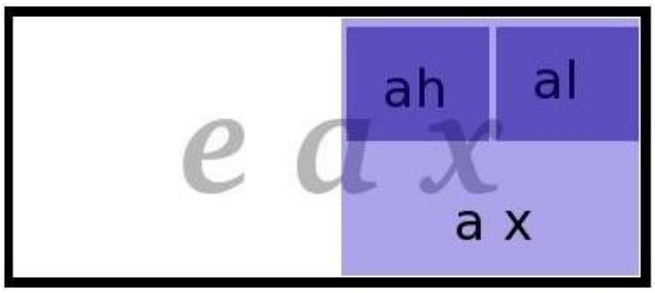
Register sind kleine Datenspeicher innerhalb des Prozessors. Bei 32 Bit Prozessoren sind die Register je 32 Bit (ein Word) groß, also wesentlich kleiner als der RAM aber dafür auch sehr viel schneller.
Anders als Variablen im Hauptspeicher haben Register feste Namen. Sie werden zu einem Teil für feste Zwecke verwendet, zum anderen kann man auch einfach Daten darin ablegen.

Die wichtigsten Register

eax	-	Allgemein verwendbar
ebx	-	Allgemein verwendbar
ecx	-	Allgemein verwendbar
edx	-	Allgemein verwendbar
ebp	-	Base Pointer
esp	-	Stack Pointer

Es kann nicht schaden, sich Namen und Bedeutungen dieser Register schon mal zu merken.

Die Datenregister eax, ebx, ecx und edx sind in weitere kleinere Register unterteilt.
Das Register eax beispielsweise beinhaltet ein weiteres namens ax (welches nur noch 2 Byte groß ist). ax wiederum ist unterteilt in ah und al, die jeweils ein Byte groß sind.



ax, ah und al sind keine zusätzlichen Register, wenn du einen Wert in eax schreibst werden die Werte in ax, ah und al überschrieben!
Das gleiche gilt für ebx, ecx und edx hier lauten die Register bx, bh, bl, cx, usw.

1.4 Die benötigten Tools

Ich werde für diesen Workshop den GNU Assembler *gas* verwenden. Wenn du einen anderen Assembler, wie z. B. *nasm* (nasm.sourceforge.net), benutzen möchtest, musst du kleine Anpassungen am Code vornehmen, da sich die Syntax etwas unterscheidet.

Bei den meisten Distributionen wird der Assembler mitgeliefert, durch eingabe von `as` kannst du überprüfen, ob du ihn installiert hast.

Falls nicht, findest du den GNU Assembler auf der Seite www.gnu.org (direkter Link: ftp.gnu.org/gnu/binutils/binutils-2.9.tar.gz).

Nach dem Herunterladen muss das Paket zuerst mit `tar xzf binutils-2.9.tar.gz` entpackt werden. Nun wecheln wir ins entpackte Verzeichnis (`cd binutils-2.9`), kompilieren und installieren die Software:

```
$ ./configure
$ make
# make install
```

Der letzte Befehl muss als Superuser ausgeführt werden!

Alternativ kannst du natürlich auch die Software über deinen Paketmanager installieren.

Der Linker *ld* wird ebenfalls im binutils-Paket mitgebracht.

Schließlich brauchst du noch einen Editor. Mein absoluter Favorit ist GNU

Emacs, du kannst natürlich auch jeden anderen benutzen (Features wie autom. Einrückung und Syntax Highlighting vereinfachen das Programmieren allerdings sehr).

Das Betriebssystem

Getestet habe ich die Beispiele unter Linux 2.6.11. Zwar sollten sie auch mit älteren Linuxversionen funktionieren (es werden nur grundlegende Systemfunktionen verwendet, die auch in Uraltversionen enthalten sein sollten), allerdings werden die Programme, anders als bei Hochsprachen, nicht unter anderen Unices (BSD, Mac OS X o. ä.) geschweige denn Windows/Dos funktionieren.
Wichtig ist außerdem, dass das System auf x86-kompatibler Hardware läuft (Intel Pentium/Celeron, AMD Athlon/Athlon 64/Sempron, ...), ein Linuxkernel für PPC nutzt wenig, da der Prozessor die x86-Befehle nicht kennt.

2. Das erste Programm

Tippe folgenden Code in einen Texteditor und speichere ihn als `exit.asm`.

```
.section .data


.section .text

.globl _start
_start:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

Als nächstes muss die Datei assembliert und gelinkt werden:

```
$ as exit.asm -o exit.o
$ ld exit.o -o exit
```

Wenn du das Programm jetzt mit `./exit` ausführst passiert nichts, außer dass es sich beendet – Gratulation, das ist dein erstes Assemblerprogramm ;)

Assemblerprogramme sind in Codeteile - Sektionen - unterteilt. In der ersten Sektion, hier `.data`, werden normalerweise Variablen definiert, die später im Code benutzt werden. Da unser Programm keine Variablen benutzt, bleibt die Sektion leer und das `.section .data` könnte man genauso gut weglassen.
In der Sektion `.text` steht der eigentliche Programmcode.
Normalerweise müssen Sprungadressen, also Speicheradressen mit Anweisungen die gerade ausgeführt werden, nur dem Programm bekannt sein. Anders ist es bei `_start`. Da das Programm vom Betriebssystem gestartet wird, muss auch die Startadresse bekannt sein. Aus diesem Grund wird in Zeile 6 die Adresse _start als global markiert - ähnlich der in Java als `public` deklarierten main() Methode.

`int $0x80` ist ein sog. *Syscall*. Syscalls sind Aufrufe an den Kernel, bei denen folgendes passiert: Linux wird mittels des (immer gleichen) Syscalls 0x80 aufgerufen und schaut im Register `%eax` nach, was zu tun ist. 1 bedeutet: „Programm beenden!“. Je nach Syscall erwartet der Kernel in den anderern Registern noch weitere Angaben. Beim exit-Syscall beispielsweise steht in `%ebx` der Rückgabewert des Programms an das System (0 bedeutet „Erfolg“). Dieser Wert darf maximal bei 255 (1 Byte) liegen. Der Rückgabewert des zuletzt ausgeführten Programms kann mit

```
$ echo $?
```

ermittelt werden.

Und schon haben wir den ersten (wirklich einfachen) Assembler-Befehl kennengelernt: `mov`. Dieser schiebt einen Wert in ein Register oder eine Stelle im Hauptspeicher.

Syntax von mov
mov <quelle>, <ziel>

2.1 Suffixe beim GNU Assembler

Der GNU Assembler hat im Gegensatz zu anderen Assemblern eine Eigenheit: Anstatt einfach nur den Befehl (z. B. `mov`) zu schreiben, muss man noch ein Suffix anhängen. Dies sagt dem Assembler mit Werten welcher Größe gearbeitet wird.
Im obigen Beispiel haben wir das Suffix 'l' verwendet, welches für 'long' – also 32 Bit – steht.
Der erste Operator von mov (die Quelle) ist zwar meistens wesentlich kleiner, 1 beispielsweise wird aber vom Assembler in 0001 (dezimal) bzw. 00000000000000000000000000000001 (binär, wer nicht nachzählen möchte: es sind 32 Bit) umgewandelt.
Außer l gibt es noch die Suffixe w (2 Byte, `movw $300, %ax`) und b (1 Byte, `movb $80, %al`).
Die Suffixe müssen nicht verwendet werden, wenn für den Assembler eindeutig ist, um wie große Werte es sich handelt (also wenn ein Operand ein Register ist). Es ist allerdings sinnvoll, sie immer zu benutzen, da so Fehler vermieden werden können, wie statt `%al` versehentlich `%ax` zu schreiben:

```
movw $1, %al
```

Das würde eine Warnung vom Assembler verursachen.

3. Datenzugriffsmethoden

Man kann auf verschiedene Arten auf gespeicherte Daten zugreifen. Der Assembler muss beispielsweise wissen, ob mit der Zahl 0x08048074 eine Adresse im Hauptspeicher, oder der Wert 134512711 gemeint ist. Deshalb legt der er eine bestimmte Syntax für eine bestimmte Art des Datenzugriffs fest. Diese Syntax ist beim GNU-Assembler recht kompliziert und trägt nicht gerade zur Schönheit des Codes bei.

Die einfachste Methode ist der sog. *Immediate Mode*, indem die Daten in der Anweisung selbst eingebettet sind. D. h. man sagt nicht "lege den Wert an Adresse blabla ins Register eax", sondern "lege den Wert 34 ins Register eax". In diesem Fall erwartet der Assembler, dass wir ein Dollarzeichen ('$') vor den Wert schreiben.

Beispiel: `movl $1, %eax` (schreibt den Wert 1 in eax)

Im *Register Addressing Mode* wird kein Wert, sondern - wie der Name schon sagt - ein Register angegeben.

Beispiel: `movl %eax, %ebx` (kopiert den Wert in eax nach ebx)

Im *Direct Addressing Mode* wird die Adresse des Wertes im Hauptspeicher angegeben.

Beispiel: `movl 48074, %eax` (kopiert den Wert an Adresse 48074 im Hauptspeicher ins Register eax)

Im *Indexed Addressing Mode* wird sowohl die Adresse, als auch ein *Offset* (in Bytes) der zur Adresse hinzuaddiert wird.

Beispiel: `movl variable(, %ebx, 1), %eax` (kopiert den ebx-ten Wert von [variable] nach eax, wobei ein Element 1 Byte groß ist)

Beim *Indirect Addressing Mode* beeinhaltet die Anweisung ein Register mit einer Adresse. Wenn das Register beispielsweise den Wert 87 enthält, wird nicht auf den Wert 87 zugegriffen, sondern auf den Wert im Hauptspeicher an Adresse 87.

Beispiel: `movl $3, (%edx)` (schreibt 3 dorthin, wo die Adresse in edx hinweist)

Der *Basepointer Addressing Mode* ist eine Mischung aus dem Indirect- und dem Indexed Addressing Mode - zu dem Register mit der Adresse wird ein Wert angegeben, der zur Adresse addiert wird.

Beispiel: `movl 4(%eax), %ebx` (kopiert den Wert auf den die Adresse in eax + 4 Bytes zeigt nach ebx)

Diese Möglichkeiten auf Daten zuzugreifen sind extrem wichtig bei der Arbeit mit dem GNU-Assembler. Wenn du beispielsweise ausversehen ein Dollarzeichen weglässt, kann das – im besten Fall – zu Fehlern beim assemblieren führen, wenn du Pech hast aber auch zu unerklärlichen Fehlfunktionen des Programms zur Laufzeit.
Bevor du weiterliest, solltest du dir also die Datenzugriffsmethoden gut einprägen.

4. Variablen

Die Möglichkeiten von Registern sind beschränkt. Sie eignen sich zwar, um für kurze Zeit kleine Datenmengen zu speichern, wenn man aber beispielsweise längere Zeichenfolgen speichern möchte, sind Register unbrauchbar, da sie nur 32 Bit groß sind.

Variablen sind im Grunde nur Aliase für Speicheradressen im RAM. Statt `mov 0x45234, %eax` kann man also `mov myVar, %eax` schreiben. Praktisch hierbei ist, dass du dich bei der Definition einer Variablen nicht selbst um deren Speicheradresse kümmern musst. Du sagst dem Assembler einfach den Namen und was du darin speichern möchtest.
Wie bereits erwähnt werden Variablen in der .data-Sektion definiert. Die Syntax dabei ist folgende:

```
<Name>: <Typ> <Wert>
```

Name

Der Name ist der Alias für die Speicheradresse. Wenn du also die Speicheradresse der Variablen in ein Register kopieren möchtest (was oft vorkommen wird), musst du `mov <Name>, <Register>` schreiben.
Wenn du allerdings den Wert der Variablen in ein Register kopieren möchtest (wobei du darauf achten musst, dass dieser nicht größer ist, als ein Register zu fassen vermag), kommt der **Indirect Addressing Mode** zum Einsatz: `mov (<Name>), <Register>`

Typ

Der Typ gibt an, was für Daten gespeichert werden sollen. Folgende Typen stehen zur Verfügung:

.ascii	-	Strings (beliebige Zeichenfolgen)
.byte	-	Eine maximal 1 Byte große Ganzzahl
.short	-	Eine maximal 2 Byte große Ganzzahl
.int	-	Eine maximal 4 Byte große Ganzzahl
.long	-	Das gleiche wie .int
.float	-	Eine maximal 4 Byte große Gleitkommazahl
.double	-	Eine maximal 8 Byte große Gleitkommazahl

Es gibt noch mehr Typen, das sind allerdings die wichtigsten. Mehr dazu in der gas-Dokumentation[1].

Wert

Welcher Wert zu welchem Typ passt, habe ich ja schon erklärt, allerdings

1 http://sourceware.org/binutils/docs-2.16/as/index.html

möchte ich noch kurz auf den Typ .ascii eingehen.

Strings werden in doppelte Anführungszeichen gesetzt. Sonderzeichen und Umlaute sind erlaubt, ebenso wie alle anderen Zeichen.
Dennoch gibt es ein paar Besonderheiten. Was zum Beispiel, wenn man Anführungszeichen oder Newline-Zeichen als Wert angeben möchte? Dann hilft folgende Tabelle:

\n	-	Neue Zeile
\t	-	Tabulator
\\	-	Backslash
\"	-	Anführungszeichen

Das sind die wichtigsten, es gibt noch mehr (s. h. gas-Manual). Der Grund, dass ich selten vollständige Tabellen schreibe ist übrigens dass vieles einfach so gut wie nie gebraucht wird und nur verwirrt. Für vollständige Infos gibt's schließlich die öffentlichen FAQs und Manuals - das nur mal so am Rande.

Soviel zur Theorie, in der Praxis sieht das folgendermaßen aus:

```
.section .data
returnValue: .byte 23

.section .text
.globl _start

_start:
    mov $1, %eax
    mov (returnValue), %ebx
    int $0x80
```

Nach kompilieren, linken und ausführen solltest du `echo $?` eingeben, um zu überprüfen, ob der Rückgabewert wie erwartet ist - es wird 23 ausgegeben, das Programm arbeitet also korrekt.

In der Praxis ist das obige Programm natürlich sinnlos, da für einen Wert dieser Größe, der auch nicht länger gespeichert werden soll, niemals der langsame Hauptspeicher benutzt werden würde. Der Code sollte bloß die Arbeit mit Variablen veranschaulichen.

4.1 Übungen

Aus welchen zwei Gründen funktioniert folgendes Programm nicht?

```
.section .data
returnVal .byte 287

.section .text
.globl _start

_start:
    mov $1, %eax
    mov (returnVal), %ebx
    int $0x80
```

5. Hello World

Endlich - das "Hello World"[2]-Programm! :)

```
.section .data
hello: .ascii "Hello World!\n"

.section .text
.globl _start
_start:
    mov $4, %eax      # 4 fuer den Syscall 'write'
    mov $1, %ebx      # File Descriptor
    mov $hello, %ecx  # Speicheradresse des Textes
    mov $13, %edx     # Laenge des Textes
    int $0x80         # und los

    mov $1, %eax      # das
    mov $0, %ebx      # uebliche
    int $0x80         # beenden
```

Assembliert und gelinkt wird wie üblich:

```
$ as hello.asm -o hello.o
$ ld hello.o -o hello
$ ./hello
Hello World!
```

Eigentlich enthält der Code nichts neues, bis auf die Kommentare. Kommentare sind Anmerkungen, die Programmierer in ihre Programme schreiben, um sie für andere (oder auch sich selbst) besser verständlich zu machen. Kommentare werden vom Assembler völlig ignoriert und machen das endgültige Programm nicht größer.
Zwar stehen Assemblerkommentare meistens hinter einem Semikolon (nasm, tasm etc.), im AT&T Syntax vom GNU Assembler allerdings hinter einem Doppelkreuz. Auch wenn die Kommentare im obigen Code völlig übertrieben sind, sollte gerade Assemblercode ausreichend kommentiert werden, da hier der Zusammenhang nicht so leicht ersichtlich ist wie in Hochsprachen.

Obwohl der Code oben keine neuen Assembleranweisungen o. ä. enthält, werde ich kurz auf Zeile 7-11 eingehen.

Zuerst wird der Wert 4 in eax geschoben. Das sagt dem Kernel später, dass ein Text ausgegeben werden soll.
In der nächsten Zeile wird 1 in ebx geschoben, das ist der sog. File Descriptor. Der File Descriptor ist eine Nummer, die beschreibt, was mit dem Text

2 Für alle Nichtprogrammierer: Traditionell sollte das „Hello World“-Programm das erste in jedem Tutorial zu einer Programmiersprache sein. Bei Assembler muss es aufgrund der komplexen Materie etwas später kommen ;)

gemacht werden soll. 1 steht für "stdout", also den Text auf dem Bildschirm über die Standardausgabe auszugeben. Würden wir 2 in ebx schieben ("stderr"), würde der Text auch auf dem Bildschirm ausgegeben werden, allerdings als Fehlermeldung. Das sähe zwar nicht anders aus, aber man sollte normale Meldungen auf keinen Fall über die Fehlerausgabe ausgeben. Der dritte File Descriptor (0 - "stdin") liest einen Text von der Standardeingabe - normalerweise die Tastatur - ein. Dies funktioniert allerdings nicht, wenn in eax 4 steht.
In der nächsten Zeile wird die Adresse des auszugebenden Textes in ecx geschoben.
Schließlich muss die Länge des Textes in edx angegeben (mov $13, %edx) und dem Kernel gesagt werden, dass er was zu tun bekommt (int $0x80).

Viele Syscalls benötigen Parameter in mehreren Datenregistern, exit ist eher eine Ausnahme. Eine Übersicht über alle Syscalls findest du auf der Seite http://www.lxhp.in-berlin.de/lhpsysc1.html.

5.1 Übungen

- Schreibe ein Programm, dass den Benutzer begrüßt und ihn anschließend auffordert, etwas einzugeben. Der eingegebene Text soll danach auf dem Bildschirm ausgegeben werden. Tipp: Zum Einlesen des Textes, guck auf der oben genannte Seite nach dem passenden Syscall.
- In folgendem Programm zum Einlesen eines Textes von der Tastatur sind 3 Fehler versteckt:

```
.section .data
buffer: .ascii ""

.section .text
.globl _start
_start:
    mov $3, %eax
    mov $1, %ebx
    mov (buffer), %ecx
    mov $10, %edx
    int $0x80
```

6. Flow Control

An alle „Denglisch-Fanatiker“: Der korrekte deutsche Titel für dieses Kapitel wäre „Programmflusskontrolle“. Da ich dieses Wort etwa so gut finde wie „MS-Dos Eingabeaufforderung“, werde ich einfach Flow Control schreiben ;)

Wir können bisher Texte ausgeben, Benutzereingaben empfangen und speichern und mit anderen Syscalls einigermaßen umgehen. Was jedoch fehlt, ist die Möglichkeit, ein Programm abhängig vom Benutzer ablaufen zu lassen (mal abgesehen davon, dass wir ein Programm mit der Aus-Taste des Computers abbrechen können) - dies wird hauptsächlich Thema dieses Kapitels sein.

6.1 Sprungadressen

Sprungadressen - auch Labels genannt - sind Markierungen im Code, die bei Bedarf ausgeführt werden können. Ein Label kennen wir bereits: _start. Das muss in jedem Programm vorhaden sein, Assembler bietet allerdings die Möglichkeit, eigene Labels zu definieren und aufzurufen:

```
.section .data
text: .ascii "hello\n"

.section .text
.globl _start
_start:
    jmp exit

    movl $4, %eax
    movl $1, %ebx
    movl $text, %ecx
    movl $6, %edx
    int $0x80

exit:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

Wie du siehst, wird die Ausführung mit `jmp <Adresse>` umgeleitet. "exit" ist nur ein Alias für die Speicheradresse, das sieht man, wenn man das Programm disassembliert (statt "jmp exit" steht dort so etwas wie "jmp 0x80879887"). Im obigen Programm wird also nicht der Inhalt der Variablen "text" ausgegeben - vorher springt das Programm mit "jmp exit" zum Ende.

Wenn die Zieladresse nur -128/+127 Bytes vom Sprungbefehl entfernt ist, spricht man von einem Near-, andernfalls von einem Far-Jump.

Desweiteren gibt es die Möglichkeit, anstelle eines Direct-Jumps (die Adresse/das Label wird direkt angegeben) einen indirekten Sprung auszuführen. Dabei steht die Adresse einfach in einem Register oder einer Variable anstatt sie direkt beim Jump-Befehl anzugeben:

```
movl $label, %eax
jmp *%eax
```

oder mit einer Variable:

```
tmp: .ascii label
[...]
jmp *tmp
```

Der Stern vor Register bzw. Variable sagt jmp, dass z. B. nicht zur Adresse "tmp", sondern zur Adresse die an tmp steht gesprungen werden soll.

Labels an sich nutzen dem Programmierer zwar noch nichts, aber in Verbindung mit Abfragen ergeben sie einen kompletten Ersatz für if, for, while, do-while, ... :)

6.2 Abfragen

Es gibt in Assembler keine Anweisung, die der der if-Klausel in Hochsprachen ähnelt. Stattdessen werden zwei Werte verglichen, das Ergebnis in einem speziellen Register namens eflags (das "Statusregister") zwischengespeichert und dann kann, abhängig vom Inhalt dieses Registers, die Programmausführung zu einem bestimmten Label springen.
In der Praxis sieht das folgendermaßen aus:

```
# Fordert den Benutzer auf, eine Zahl zwischen 1 und 5 einzugeben und
# prüft, ob der eingegebene Wert korrekt ist.


.section .data
msg: .ascii "Bitte gib eine Zahl zwischen 1 und 5 ein!\n"
wv:  .ascii "Ungültiger Wert!\n"
buf: .long

.section .text
.globl _start
_start:
    movl $4, %eax
    movl $1, %ebx
    movl $msg, %ecx
    movl $42, %edx
    int $0x80

    movl $3, %eax
    movl $0, %ebx
    movl $buf, %ecx
    movl $2, %edx       # 2 Bytes, damit das Newline-Zeichen nicht im
```

```
                        # Eingabebuffer "haengenbleibt".
    int $0x80


    mov $0, %eax
    cmpb $0x31, buf(, %eax, 1)  # Wir wollen nur das erste Byte (die
                                # eingegebene Zahl) von buf vergleichen.
    jl wrongValue               # kleiner als 1 (ASCII (Hex) 31)?
    cmpb $0x35, buf(, %eax, 1)  # groesser als 5 (ASCII (Hex) 35)?
    jg wrongValue


    movl $1, %eax
    movl $0, %ebx
    int $0x80

wrongValue:                     # gibt eine Fehlermeldung aus
    movl $4, %eax
    movl $1, %ebx
    movl $wv, %ecx
    movl $17, %edx
    int $0x80

    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

Wie du siehst, wird der Vergleich mir dem Befehl `cmp` vorgenommen. Da wir aber nur ein einziges Byte von buf lesen wollen, lautet der Befehl `cmpb`.

Syntax von cmp

```
cmp Wert1, Wert2
```

Das Verhältnis der beiden Werte wird im eflags-Register zwischengespeichert und kann mit folgenden Befehlen benutzt werden:

jmp <Adresse>	-	Springt zu <Adresse>, egal was in eflags steht.
je <Adresse>	-	Springt zu <Adresse>, wenn die beiden Werte gleich sind.
jne <Adresse>	-	Springt zu <Adresse>, wenn die bieden Werte ungleich sind.
jg <Adresse>	-	Springt zu <Adresse>, wenn der zweite Wert größer als der erste ist.
jge <Adresse>	-	Springt zu <Adresse>, wenn der zweite Wert größer oder gleich dem ersten ist.
jl <Adresse>	-	Springt zu <Adresse>, wenn der zweite Wert kleiner als der erste ist.
jle <Adresse>	-	Springt zu <Adresse>, wenn der zweite Wert kleiner oder gleich dem ersten ist.

Diese Befehle sind nicht schwer zu merken - e (equal) für gleich, g (greater) für größer und l (less) für kleiner.

Damit können auch recht einfach Schleifen programmiert werden. Herauszufinden wie genau, überlasse ich dir (s. h. Übungen). Allerdings gebe ich noch einen Tipp: Der Befehl `inc` ist für Schleifen sehr wichtig.

Syntax von inc

```
inc Register
```

`inc` erhöht den Wert des Registers um 1.

6.3 Übungen

Schreibe ein Programm, das vom Benutzer die Eingabe eines Textes verlangt und diesen sooft ausgibt, wie der Benutzer zuvor angegeben hat.

7. Mathematische Befehle

7.1 Grundrechenarten

add

Syntax: `add Quelle, Ziel`

Beschreibung: Addiert Quelle zu Ziel und speichert das Ergebnis in Ziel.

Beispiel:

```
movl $4, %eax
addl $2, %ebx
# In ebx steht jetzt 6.
```

sub

Syntax: `sub Quelle, Ziel`

Beschreibung: Wie add, nur wird Quelle von Ziel subtrahiert.

Beispiel:

```
movl $4, %eax
subl $3, %ebx # In ebx steht jetzt 1.
```

mul

Syntax: `mul Faktor`

Beschreibung: Multipliziert Faktor mit al (wenn Faktor 1 Byte groß ist), mit ax (wenn Faktor 1 Word groß ist) oder mit eax (wenn Faktor 1 Doubleword groß ist).
In jedem Fall wird das ergebnis im Register mit dem zweiten Faktor gespeichert.

Beispiel:

```
movb $4, %al
movb $2, %bl
```

```
mulb %bl
# al enthält jetzt 8.
```

imul

Syntax: `imul Operand1[, Operand2[, Operand3]]`

Beschreibung: Wenn nur ein Operand angegeben wird, macht imul das gleiche wie mul.
Wenn zwei Operanden angegen sind, werden sie miteinander multipliziert und das Ergebnis am Ort des zweiten Operanden gespeichert.
Wenn drei Operanden angegeben sind (der erste Faktor muss eine Konstante sein!), wird Operand1 mit Operand2 multipliziert und das Ergebnis in Operand3 gespeichert.

Beispiel:

```
movl $5, %eax
imul $4, %eax, %ebx
# in ebx steht jetzt 20.
```

div

Syntax: `div Operand`

Beschreibung: Dividiert al/ax/eax (s. h. mul) durch Operand und speichert das Ergebnis im Register des Dividenden. Beide Werte dürfen nicht negativ sein.

Beispiel:

```
movb $8, %al
movb $4, %bl
mulb %bl
# al enthält jetzt 2.
```

idiv

Syntax: `idev Operand`

Beschreibung: Wie div, nur dürfen beide Werte negativ sein.

```
movb $-8, %ah
movb $4, %bh
idivb %bh
# ah enthält jetzt -2
```

Bei Multiplikationen/Divisionen mit/durch 2, 4, 8 etc. sollte aus Gründen der Effizienz statt (i)mul/(i)div shl bzw. shr verwendet werden (s. u.).

7.2 Bitoperationen

and

Syntax: `and Operand1, Operand2`

Beschreibung: and verknüpft die beiden Operanden bitweise mit der AND-Operation und schreibt das Ergebnis in den zweiten Operanden.

Beispiel:

```
movl $4, %eax
andl $9, %eax
# eax enthält nun 0:
#   4 -> (dual) 0100
#   7 -> (dual) 1001
#   Ergebnis    0000 -> (dezimal) 0
```

or

Syntax: `or Operand1, Operand2`

Beschreibung: or verknüpft die beiden Operanden bitweise mit der or-Operation und schreibt das Ergebnis in den zweiten Operanden.

Beispiel:

```
movl $4, %eax
orl $9, %eax
# eax enthält nun 13:
#   4 -> (dual) 0100
#   7 -> (dual) 1001
#   Ergebnis    1101 -> (dezimal) 13
```

xor

Syntax: `xor Operand1, Operand2`

Beschreibung: xor verknüpft die beiden Operanden bitweise mit der xor-Operation und schreibt das Ergebnis in den zweiten Operanden.

Beispiel:

```
movl $4, %eax
xorl $9, %eax
# eax enthält nun 13:
#   4 -> (dual) 0100
#   7 -> (dual) 1001
#   Ergebnis    1101 -> (dezimal 13)
```

not

Syntax: `not Operand`

Beschreibung: not verändert alle Bits ins Gegenteil. Wichtig ist hierbei: Wenn man z. B. $4 in eax schiebt sind alle Bits rechts von "0100" ebenfalls 0, werden durch die not-Anweisung in eine 1 umgewandelt und das Ergebnis ist unerwartet hoch.

Beispiel:

```
movb $4, %ah
notl %ah
# ah enthält nun 251:
#   4 -> (dual) 0000 0100
#   Ergebnis    1111 1011 -> (dezimal 251)
```

shl

Syntax: `shl [Anzahl, ]Operand`

Beschreibung: shl verschiebt das Bitmuster <Anzahl> mal nach links. Der Standardwert für Anzahl ist 1. 1 shift-left entspricht einer Multiplikation mit 2, 2 einer Multiplikation mit 4 usw.

Beispiel:

```
movb $4, %ah
shlb $2, %ah
```

```
# ah enthält nun 16:
#   4 -> (dual) 0000 0100
#   Ergebnis    0001 0000 -> (dezimal 16)
```

---

shr

Syntax: `shr [Anzahl, ]Operand`

Beschreibung: Wie shl, nur wird das Bitmuster nach rechts verschoben. 1 shift-right entspricht einer Division mit 2, 2 einer Division mit 4 usw. (natürlich nur, wenn das Ergebnis eine ganze Zahl ist).

Beispiel:

```
movb $4, %ah
shrb $1, %ah
# ah enthält nun 2:
#   4 -> (dual) 0000 0100
#   Ergebnis    0000 0010 -> (dezimal 2)
```

---

8. Die glibc

Wer schon etwas länger mit GNU arbeitet, kennt die glibc. Das ist eine Library, die benutzt werden kann um z. B. Texte auf dem Bildschirm auszugeben, Dateien einzulesen, über Netzwerke mit anderen Programmen zu kommunizieren, neue Prozesse zu starten, kurz: Funktionen für praktisch alle Aufgaben. Die komplette Dokumentation findest du entweder auf der GNU-Webseite (www.gnu.org) oder auf deinem Rechner unter /usr/doc/glibc-<Version>.

Zuerst einmal stellt sich die Frage: Warum nicht mit Syscalls, sondern mit der glibc programmieren? Jetzt lernt man schon eine Low-Level Programmiersprache und dann das.
Die Antwort ist, dass die Programme dadurch plattformunabhängig werden. Solange das Zielsystem auf einem kompatiblen Prozessor läuft, der richtige Assembler und die glibc vorhanden sind, kann das Programm kompiliert werden, egal auf welchem Betriebssystem. Außerdem macht das Verwenden der glibc den Programmcode wesentlich verständlicher - wer kennt schon alle Linux-Syscalls auswendig?

8.1 Funktionen

Hochsprachler kennen Funktionen. Das sind Codeteile - oft in unterschiedlichen Dateien - die unabhängig vom eigentlichen Programm arbeiten und bei Bedarf aufgerufen werden können. Bevor du die glibc benutzen möchtest, musst du lernen, Funktionen aufzurufen. Leider ist das Programmieren von Funktionen in Assembler lange nicht so einfach wie in C oder Pascal. Aus diesem Grund unterteile ich diese Lektion in zwei Kapitel. In diesem Kapitel wirst du nur die Grundlagen über den Stack und das Aufrufen von fertigen Funktionen lernen.

Eine Funktion wird folgendermaßen aufgerufen:

```
call <Funktionsname>
```

Das allein nutzt noch nicht viel, wie soll zum Beispiel die Funktion `write` wissen, welchen Text sie wo ausgeben soll? Zu diesem Zweck gibt es *Funktionsparameter*. Das sind Werte, die vor dem Funktionsaufruf auf den Stack gelegt werden und dann später von der Funktion genutzt werden können:

```
push <Speicheradresse/Register/Wert>
```

Es können mit push immer nur 4 Bytes (ein double word) auf den Stack gelegt werden, nicht mehr und nicht weniger.
Das besondere bei Funktionsaufrufen in Low-Level Sprachen ist, dass die Parameter "falsch herum" auf den Stack gelegt werden. Das hat den einfach Grund, dass die Funktion die Werte in umgekehrter Reihenfolge vom Speicher

holt (deshalb heißt der Stack wohl wie er heißt). So muss der erste Parameter also zuletzt auf den Stack gelegt werden.

Eine Funktion löscht ihre Parameter auf dem Stack nicht. Der Ordnung halber sollte es der Programmierer also selbst tun. Die beste Lösung dazu ist, das Register esp (den Stackpointer) zu verändern. Dieses Register enthält immer die Speicheradresse des obersten Stackelements. Um genau zu sein, enthält sie die Speicheradresse des untersten Stackelements. Denn eigentlich werden neue Werte nicht auf den Stack, sondern unter ihn gelegt. Je größer die Speicheradresse also ist, desto höher im Stack befindet sich ihr Ziel. Wenn nun also die Parameter unter den Stack "geschoben" werden, müssen soviele Bytes zum Stackpointer addiert werden, wie sich Parameter auf dem Stack befinden.
Alternativ dazu kann man auch den Befehl `pop` verwenden, um Werte vom Stack zu holen. Dieser Befehl erwartet ein Register/eine Variable als Operanden, in den der Wert auf den der Stackpointer verweist geschoben wird. Statt `addl $4, %esp` kann man also `popl %eax` schreiben.

Schreiben wir ein einfaches "Hello World"-Programm mit Hilfe der glibc:

```
.section .data
msg: .ascii "Hello World!\n\0"

.section .text
.globl _start
_start:
        # Textausgabe mit der Funktion "printf" - kennt jeder ;).
        pushl $msg
        call printf
        # Nun muss der Stackpointer zurückgesetzt werden, indem
        # ich 4 Bytes (die Größe der Adresse msg) addiere
        addl $4, %esp

        # Programm beenden mit der Funktion "exit".
        # 1. und einziger Parameter: Rueckgabewert
        pushl $0
        call exit
```

Bitte beachte das `\0` am Ende von `msg`. Da `printf` keine Längenangabe des auszugebenden Strings erwartet, muss dieser ein Zeichen enthalten, das printf sagt, dass der String zu Ende ist. Dies ist das ASCII-Zeichen 0. Dem Assembler wird mittels dem Backslash mitgeteilt, dass die Null nicht als Zeichen, das zum eigentlichen String gehört, interpretiert werden soll (andernfalls würde die Null in den ASCII-Wert für Null umgewandelt werden, was 48 entspräche[3]).

3 http://www.asciitable.com

8.2 Linken mit der glibc

Das Linken dieses Programmes ist nicht mehr ganz so einfach. Da die Funktionen aus der glibc verwendet werden, muss selbige und ein weiteres Programm, mit dem Libraries zur Laufzeit geladen werden können, mit ins Binary eingebunden werden.
Assembliert wird wie immer mit

```
$ as <Quelldatei> -o <Zieldatei>
```

Gelinkt wird folgendermaßen:

```
$ ld --dynamic-linker <Pfad zum Link zu /lib/ld-x.x.x.so> \
`pkg-config --libs glib-x.x` <Quelldatei> -o <Zieldatei>
```

(Wenn auf einer Eingabezeile, ohne Backslash)

Der Wert von `--dynamic-linker` ist bei mir `/lib/ld-linux.so.2`. Ohne diese Angabe können Librarys nicht dynamisch geladen werden.
pkg-config ist ein Tool, mit dem Optionen für den Linker, welche zum einbinden einer bestimmten Library benötigt werden, herausgefunden werden können. Wichtig ist hierbei, dass die Umgebungsvariable `PKG_CONFIG_PATH` richtig gesetzt ist (bei mir lautet ihr Wert *`/usr/lib/pkgconfig`*). Falls du mit pkg-config nicht klar kommst, kannst du auch den direkten Pfad zur glibc-Library (bei mir */lib/libc.so.6*) statt dem `` `pkg-config blablub` `` angeben.
Das Linken kann evtl. Probleme verursachen, wenn du etwas nicht verstanden hast oder Probleme auftreten -> gerne E-Mail an mich.

8.3 Rückgabewerte von Funktionen

Viele Funktionen geben einen Wert zurück. Das kann entweder ein benötigter Wert sein (eine Funktion bespielsweise, die von allen Parametern den größten ermittelt und diesen zurückgibt), oder einfach eine Null für "Erfolg" bzw. ein Wert ungleich Null für "Misserfolg".
Rückgabewerte von Funktionen werden im Register eax gespeichert. Wenn sie also noch gebraucht werden, sollte eax direkt nach dem Funktionsaufruf auf den Stack/in eine Variable geschoben werden.

Ein weiteres Beispiel:

```
# Liest einen Text von der Standardeingabe ein und gibt die
# Anzahl der eingegebenen Zeichen aus.

.section .data
msg:    .ascii "Bitte gib einen Text ein und bestaetige mit <Return>!\n\0"
result: .ascii "Du hast %d Zeichen eingegeben.\n\0"
buffer: .ascii ""
```

```
.section .text
.globl _start
_start:
        # Zuerst wird die Aufforderung zur Texteingabe ausgegeben
        pushl $msg
        call printf
        addl $4, %esp

        # Dann wird der Text eingelesen:
        pushl $100      # max. 100 Zeichen einlesen
        pushl $buffer
        pushl $1        # Filedeskriptor: stdin
        call read
        addl $12, %esp # 3 Parameter mit je 4 Bytes

        # Nun wird mit der Funktion strlen die Länge des Strings ermittelt.
        # Sie erwartet als Parameter nur den String selbst und gibt die Anzahl
        # der Zeichen zurück.
        pushl $buffer
        call strlen
        addl $4, %esp

        # Jetzt noch das Ergebnis ausgeben...
        pushl %eax
        pushl $result
        call printf
        addl $8, %esp

        # ... und das Programm beenden.
        pushl $0
        call exit
```

Das war's eigentlich schon. Informationen zu bestimmten Funktionen findest du auf http://www.gnu.org/software/libc/manual/html_mono/libc.html. Dort sind Codebeispiele zwar immer in C-Syntax, was aber kein Problem ist - einfach Parameterreihenfolge umdrehen, nacheinander mittels `push` auf den Stack legen und die Funktion mit `call` aufrufen.

9. Eigene Funktionen schreiben

9.1 Warum Funktionen schreiben?

- Die gleiche Aufgabe muss oft ausgeführt werden, allerdings mit unterschiedlichen zu bearbeitenden Werten.
  Durch Einsatz von Funktionen muss nicht der gleiche Code x mal abgetippt werden, sondern er steht einmal in einer Funktion, die die zu verwendenden Werte als Parameter erwartet. Dadurch wird die Fehlersuche erleichtert, das Programm kleiner und meistens auch schneller.
- Der selbe Code wird in unterschiedlichen Programmen benötigt (s.h. glibc).
- In einem Team von Programmieren soll jeder Code schreiben, der bestimmte Aufgaben erfüllt. Solche Codestücke sollten möglichst unabhängig vom Programm testbar sein - genau das ist mit Funktionen möglich.

9.2 Einfache Funktionen

Folgender Code zeigt eine einfache Funktion, die weder Parameter erwartet, noch einen Wert zurückgibt:

```
.section .data
msg: .ascii "Funktion myfunc aufgerufen!\n\0"

.section .text
.globl _start
_start:
        call myfunc

        pushl $0
        call exit

.type myfunc, @function
myfunc:
        pushl $msg    # msg per printf
        call printf   # ausgeben...
        addl $4, %esp # ...und aufraeumen

        ret
```

Wie du siehst, werden Funktionen mittels `.type <Funktionsname>, @funktion` als solche gekennzeichnet, worauf folgt der Funktionsname und ein Doppelpunkt folgen.
Beim Aufruf von `myfunc` mit `call` wird die Adresse dieser Anweisung auf den

Stack gelegt. Das ist wichtig, damit die Funktion später zu der Anweisung zurückkehren kann, an der sie aufgerufen wurde.
In der Funktion selbst wird die Adresse `msg` auf den Stack gelegt und `printf` aufgerufen. Die darauf folgende Anweisung ist extrem wichtig, wenn sich der Code in einer Funktion befindet. `ret` lässt die Programmausführung nämlich zu der Adresse springen, die als oberstes auf dem Stack liegt. Würde der Stack nicht aufgeräumt werden, würde `ret` versuchen zur Adresse `msg` springen, was wiederum zu einem "Segmentation fault" (Speicherzugriffsverletzung) führen würde.

Der Stack

```
//////Daten///////
Rücksprungadresse    <- nach call myfunc
0x896blablub <msg>   <- nach pushl $msg
```

Nach
```
  addl $4, %esp
```
zeigt `esp` wieder auf die Rücksprungadresse
```
  ret
```
wieder auf Die ursprüngliche Adresse vor dem Funktionsaufruf.

9.3 Funktionsparameter

Wir wissen, dass Funktionsparameter vor dem Funktionsaufruf auf den Stack gelegt werden. Nach der call-Anweisung sieht der Stack also folgendermaßen aus:

```
//////Daten///////
...
Parameter2
Parameter1
Rücksprungadresse
```

`esp` verweist auf die Rücksprungadresse, auf den ersten Parameter kann man also mit `4(%esp)` zugreifen.
Dabei ergibt sich folgendes Problem: Wenn innerhalb der Funktion Werte auf den Stack gepusht werden, bevor die Parameter vom Stack geholt werden, liefert die obige Anweisung einen falschen Wert, nämlich den vorletzten auf den Stack gepushten.
Aus diesem Grund wird normalerweise das Register `ebp` - der Basepointer -

verwendet, um auf Funktionsparameter zuzugreifen:

```
.section .data
msg: .ascii "Funktion myfunc aufgerufen!\n\0"

.section .text
.globl _start
_start:
        pushl $msg
        call myfunc

        pushl $0
        call exit

.type myfunc, @function
myfunc:
        # Da der Basepointer gleich veraendert wird, muss
        # er zunaechst gesichert werden.
        pushl %ebp

        movl %esp, %ebp

        # Hier koennen jetzt beliebig viele pushs stattfinden, da nicht ueber
        # den orginalen Stackpointer auf die Parameter zugegriffen wird.

        pushl 8(%ebp) # msg per printf
        call printf   # ausgeben...
        addl $4, %esp # ...und aufraeumen

        # Jetzt noch den Basepointer wiederherstellen...
        popl %ebp

        # ... und zur Ruecksprungadresse zurueckkehren
        ret
```

Wie du siehst, wird der Wert von `esp` nach `ebp` geschoben und von dort an `ebp` für die Zugriffe auf die Parameter verwendet. Da die Adresse in `ebp` immer gleich bleibt, egal welche Werte auf den Stack gepusht werden, ist sichergestellt, dass `8(%ebp)` auf den ersten, `12(%ebp)` auf den zweiten Parameter usw. verweisen. Für den Fall, dass `ebp` relevante Daten enthält, wird der Inhalt vor dem Verändern des Registers auf den Stack gelegt und am Ende der Funktion wieder nach `ebp` geschoben.

```
//////Daten///////
...
Parameter2
Parameter1
Rücksprungadresse
Alter Wert von ebp    <- ebp
gepushter Wert #1
```

```
gepushter Wert #2     <- esp
```

9.4 Lokale Variablen und Rückgabewerte

In Verbindung mit Funktionen gibt es drei verschiedene Typen von Variablen:

- **Globale Variablen**
  Dies sind Variablen, auf die von allen Funktionen gleichermaßen zugegriffen werden kann. Im obigen Beispiel war `msg` zum Beispiel eine solche Variable.

- **Statische Variablen**
  Statische Variablen unterscheiden sich in Assembler nicht von globalen Variablen. Die Syntax ist die selbe, nur werden solche Variablen nur von einer Funktion genutzt.

- **Lokale Variablen**
  Lokale Variablen werden immer innerhalb einer Funktion definiert. Auf sie kann ausschließlich von eben dieser Funktion zugegriffen werden.

```
.section .text
.globl _start
_start:
        pushl $1
        call add
        addl $4, %esp

        pushl %eax
        call exit


# Die Funktion add addiert 6 zu einem uebergebenen Wert und gibt
# das Ergebnis zurueck.
.type add, @function
add:
        pushl %ebp      # Wie ueblich wird der
        movl %esp, %ebp # Basepointer gesichert

        # Wir schaffen Platz fuer eine lokale Variable (4 Bytes).
        subl $4, %esp

        # Den ersten Parameter nach eax...
        movl 8(%ebp), %eax
        # ... und dann in die lokale Variable schieben
        movl %eax, -4(%ebp)

        # Nun wird 6 zum Inhalt der lokalen Variable addiert
        addl $6, -4(%ebp)
        # und das ganze nach eax geschoben
```

```
        movl -4(%ebp), %eax

        # Die lokale Variable loeschen, indem der Stackpointer
        # zurueckgesetzt wird
        addl $4, %esp
        # den Basepointer zuruecksetzen
        popl %ebp
        # und zur Ruecksprungadresse springen
        ret
```

Das bedarf wohl einiger Erklärung:

- Mit `subl $4, %esp` wird der Stackpointer um 4 Bytes verschoben. Dieser Speicherplatz ist bis zu diesem Zeitpunkt ungenutzt, wir können irgendwelche Werte hineinschreiben.

  ```
  //////Daten///////
  ...
  Parameter1
  Rücksprungadresse
  Alter Wert von ebp   <- ebp
  Lokale Variable      <- esp
  ```

  Auf diesen Speicherbereich kann wieder über den Basepointer zugegriffen werden: `-4(%ebp)`.
- In den darauf folgenden zwei Anweisungen wird der einzige Funktionsparameter in den davor geschaffenen Speicherbereich geschrieben.
- Dann wird 6 dazu addiert und alles nach eax geschoben. Funktionen geben ihre Ergebnisse normalerweise in diesm Register zurück.
- Schließlich muss der vorher reservierte Speicher wieder freigegeben werden, sprich der Stackpointer um die selbe Anzahl Bytes nach oben verschoben werden (`addl $4, %esp`).
- Der Rest ist bekannt: Der Basepointer wird wiederhergestellt und die Programmausführung springt zur Ruecksprungadresse.

9.5 ENTER und LEAVE

Die Ausführung der ersten drei Befehle am Anfang einer Funktion wird "Aufbau eines Stackframes" genannt, das Aufräumen "Verlassen des Stackframes". Da diese Befehlsfolgen in Assembler sehr oft ausgeführt werden müssen, gibt es dafür einen besonderen Befehl: `enter` für den Aufbau eines Stackframes und `leave` zum Verlassen.

Syntax von enter:

```
enter Operand1, Operand2
```

Der erste Operand gibt die Größe des zu reservierenden Stackspeichers in Bytes an, der zweite die Verschachtelungstiefe (er kann normalerweise auf 0 gesetzt werden).

`leave` wird ohne Parameter aufgerufen.

Schreiben wir also obiges Programm entsprechend um:

```
.section .text
.globl _start
_start:
        pushl $1
        call add
        addl $4, %esp

        pushl %eax
        call exit


# Die Funktion add addiert 6 zu einem uebergebenen Wert und gibt
# das Ergebnis zurueck.
.type add, @function
add:
        # pushl %ebp
        # movl %esp, %ebp
        # subl $4, %esp
        enter $4, $0

        # Den ersten Parameter nach eax...
        movl 8(%ebp), %eax
        # ... und dann in die lokale Variable schieben
        movl %eax, -4(%ebp)

        # Nun wird 6 zum Inhalt der lokalen Variable addiert
        addl $6, -4(%ebp)
        # und das ganze nach eax geschoben
        movl -4(%ebp), %eax

        # addl $4, %esp
        # popl %ebp
        leave

        # und zur Ruecksprungadresse springen
        ret
```

9.6 Flüchtige und nicht flüchtige Register

Wie du an den obigen Beispielprogrammen gesehen hast, werden innerhalb der Funktionen bestimmte Register einfach überschrieben (eax) und andere

vorher gesichert und am Ende wiederhergestellt (ebp).
Damit nach dem Aufruf einer Funktion keine unerwarteten Werte in den Registern stehen, wurde folgende Regel festgelegt:

Die Register eax, ebx, ecx und edx können von einer Funktion beliebig verändert werden. Sie werden deshalb als "flüchtige Register" bezeichnet. Alle anderen Register sollten gesichert werden, bevor ihr Inhalt geändert wird, weshalb sie "nicht flüchtige Register" heißen.

9.7 Übungen

Schreibe eine Funktion, die den ersten Parameter hoch den zweiten rechnet und das Ergebnis zurückgibt. Für funktionsinterne Daten solltest du lokale Variablen benutzen.

Anhang A: Lösungen zu den Übungen

Kapitel 4:

1. In eine Byte-Variable passt nur eine ein Byte große Zahl, also 0-255.
2. Nach "returnVal" fehlt ein Doppelpunkt.
3. Ein Programm kann max. den Wert 255 zurückgeben.

Kapitel 5:

1. Das Program könnte so aussehen:

```
.section .data
msg: .ascii "Bitte gib einen Text ein!\n";
buffer: .ascii ""

.section .text
.globl _start
_start:
        movl $4, %eax
        movl $1, %ebx
        movl $msg, %ecx
        movl $26, %edx
        int $0x80

        movl $3, %eax
        movl $0, %ebx
        movl $buffer, %ecx
        movl $80, %ecx
        int $0x80

        movl $4, %eax
        movl $1, %ebx
        movl $buffer, %ecx
        movl $26, %edx
        int $0x80

        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

2. a) Der Filedescriptor für die Standardeingabe ist 0 und nicht 1.
   b) Der read-Syscall erwartet in ecx die Adresse des Buffers und nicht seinen Inhalt.
   c) Das Programm wird nicht beendet, läuft einfach weiter, führt folglich Anweisungen im Speicher aus die nicht von ihm sind, was schließlich zu der Meldung "Segmentation fault" und einem Programmabbruch führt.

Kapitel 6:

Das Programm könnte so aussehen:

```
.section .data
anzahl: .long 0x0
text: .ascii "Hello World!"
counter: .byte 0x0

.section .text
.globl _start
_start:
        # Wie oft soll <text> ausgegeben werden?
        movl $3, %eax
        movl $0, %ebx
        movl $anzahl, %ecx
        movl $2, %edx
        int $0x80

        # <counter> wird später mit der Benutzereingabe verglichen.
        # Aus diesem Grund wird er mit dem ASCII-Wert für 1
        # (Hexadezimal 0x31) initialisiert.
        movb $0x31, (counter)

loop_begin:
        # Nun vergleich wir den Zählerstand (<counter>) mit der vom
        # Benutzer eingegebenen Zahl.
        movl $0, %eax
        movb (counter), %bh
        cmpb anzahl(, %eax, 1), %bh
        # Größer? Dann beenden, <text> wurde oft genug ausgegeben.
        jg exit

        # Andernfalls geben wir <text> aus...
        movl $4, %eax
        movl $1, %ebx
        movl $text, %ecx
        movl $13, %ecx
        int $0x80

        # ...inkremieren den counter...
        movb (counter), %ah
        inc %ah
        movb %ah, (counter)

        # ... und springen wieder zum Schleifenbeginn.
        jmp loop_begin

exit:
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

Kapitel 9:

```
.section .text
.globl _start
_start:
        pushl $3
        pushl $3
        call pow
```

```
        addl $8, %esp

        pushl %eax
        call exit

.type pow, @function
pow:
        enter $4, $0

        movl 8(%ebp), %eax
        movl %eax, -4(%ebp)
        movl 12(%ebp), %ebx
        movl $0, %ecx
        subl $1, %ebx
start_loop:
        movl -4(%ebp), %edx
        imull %eax, %edx
        movl %edx, -4(%ebp)

        incl %ecx
        cmp %ebx, %ecx
        jne start_loop

        movl -4(%ebp), %eax
        leave
        ret
```