title db "64-Bit Assembler Programmierung unter Linux", 0xA

Einfach erklärt

Mark B.

DANKSAGUNG UND VORWORT

Zunächst möchte ich mich an dieser Stelle bei all denjenigen bedanken, die mich während der Anfertigung dieses Buchs unterstützt und motiviert haben.

Ganz besonders gilt der Dank meiner Freundin, die mich während der gesamten Arbeit motiviert hat und es mir nicht übel nahm, dass ich so viel von unserer gemeinsamen Freizeit in dieses Projekt steckte - danke Schatz!

Des weiteren Danke ich Elias D. der mir eine Hilfe war die Formulierungen möglichst allgemeinverständlich und klar zu halten. Auch er ein- oder andere Rechtschreibfehler fiel Ihm zum Opfer.

WAS SIE ERWARTET

Was sie in diesem Buch erwartet ist vor allem eine Einführung in die Programmierung mit Assembler sowie ein Einblick in die Herangehensweise an diverse Problemlösungen anhand einiger praktischer Beispiele.

Wer eine genaue Betrachtung diverser CPU-Architekturen und seitenlange Abhandlungen über interne Vorgänge der CPU erwartet dem lege ich an dieser Stelle das über 4.000 Seiten dicke Entwicklerhandbuch von Intel ans Herz. Wir wollen hier einen praktischeren Ansatz wählen und uns primär auf das grundlegende Verständnis von Assembler und die Herausforderungen bei dieser speziellen Art der Programmierung konzentrieren.

Natürlich werde ich die wichtigsten Grundlagen kurz darlegen und in weiterer Folge bei praktischen Aufgaben weiter darauf eingehen um das Grundlagenwissen weiter zu vertiefen.

INHALT

ASSEMBLER UND OPCODES

Assembler wird sowohl als Begriff für Programmiersprachen als auch für diejenigen Programme die aus dem Assembler-Code ausführbare Dateien bauen benutzt.

Hierbei kann man anhand der Syntax folgende zwei Schreibweisen unterscheiden:

AT&T - Syntax:
```
mov $0x1, %rax
```

und die

Intel-Syntax:
```
mov rax, 0x1
```

Wie Sie sehen sind bei AT&T und Intel die Argumente vertauscht und die Intel-Syntax ist etwas reduzierter. Wir werden in diesem Buch ausschließlich die Intel-Syntax verwenden.

Assembler als Sprache greift auf den Befehlsvorrat der CPU zurück. Dies erlaubt es beispielsweise Programme perfekt auf die entsprechende Hardware zu optimieren auf der anderen Seite werden Programme auf diese Weise nicht wirklich portabel da man zB auf `Syscalls` des Betriebssystems und eventuell spezielle Befehlssätze einer bestimmten CPU-Familie oder -Generation zurückgreift.

So kann man die maximale Leistung der Hardware abrufen aber dies würde im Extremfall auch dazu führen, dass ein

Programm auf einem anderen PC mit dem gleichen Betriebssystem aber einen anderen CPU nicht lauffähig wäre. Gleiches gilt natürlich für einen PC mit identer Hardware aber einem anderen Betriebssystem.

Eine CPU versteht nur binäre Eingaben - also eine Folge von Einsen und nullen. Der oben gezeigte Befehl `mov rax, 0x1` würde in binär wie folgt aussehen:

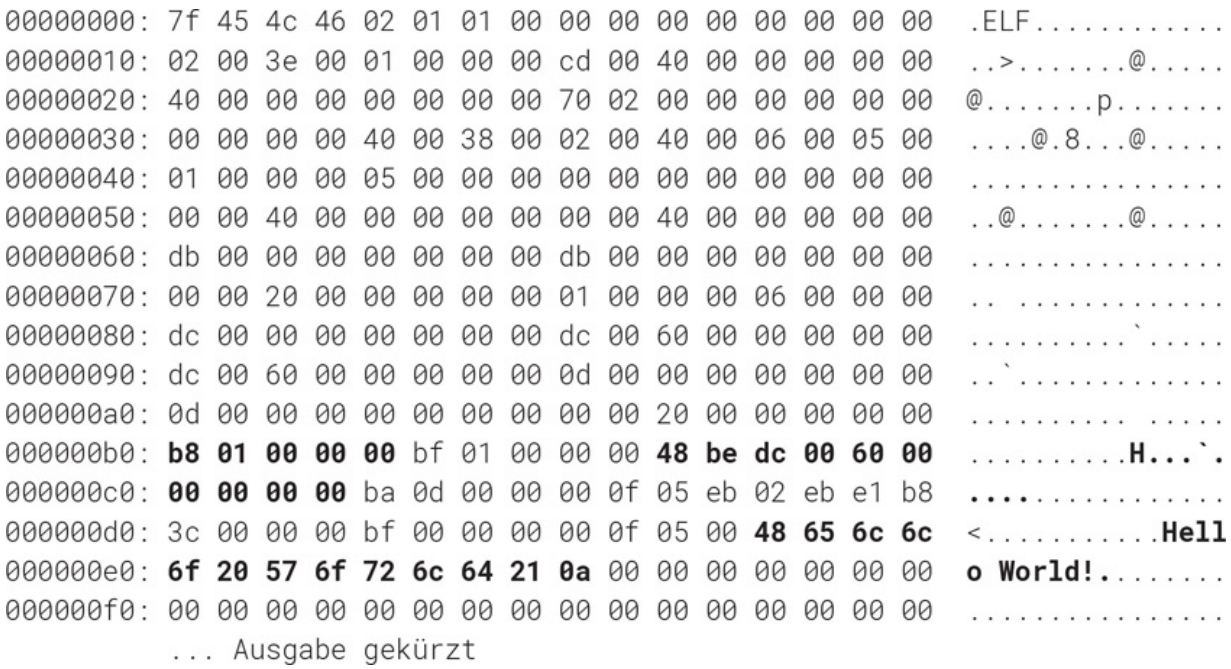
```
10111000000000010000000000000000000000000
```

Da diese lange Zahlenkolonne etwas unhandlich ist, nutzt man in der Regel die hexadezimale Schreibweise zur Darstellung von Opcodes (Operationcodes) - zB `b801000000`. Meist wird diese nochmals zur besseren Lesbarkeit in einzelne Byte aufgespalten: `b8 01 00 00 00`

Opcodes sind also nichts anderes als die Maschinenbefehle. Da die meisten Menschen weder mit `b801000000` noch mit der Binärdarstellung davon wirklich gut arbeiten könnten, hat man Assembler entwickelt. Man nennt diese auch Programmiersprachen der 2. Generation. Anstatt Binärcode nutzt man deutlich besser lesbare Abkürzungen aus dem englischen - die sogenannten Mnemonics wie zB `mov` für move, `sub` für subtract, `mul` für multiply, usw.

Der Assembler als Programm zur Erstellung von ausführbaren Dateien macht aber noch einiges mehr als nur Mnemonics in Opcodes umzuwandeln. Das Programm nimmt uns auch etwas Arbeit bei der Entwicklung ab, indem es zB Macros zur Verfügung stellt und/oder umwandelt und diverse Berechnungen für uns übernimmt.

Sehen wir uns dazu die ausführbare Datei des "Hello World" Beispiels in einem Hexeditor an:



Wir erkennen hier zB unseren Opcode `b8 01 00 00 00` für `mov rax, 0x1` und wir sehen auch den Text "`Hello World! `". Greifen wir an dieser Stelle etwas vor und ich zeige Ihnen die Zeile, in der der Text definiert wird:

```
text db "Hello World!", 0xA
```

Hierbei ist `text` ein sogenanntes Label das wir verwenden können um die Speicheradresse von dem Text anzusprechen. `db` steht für define byte und das abschließende `0xA` ist nichts weiter als die hexadezimale Schreibweise für das Newline-Zeichen.

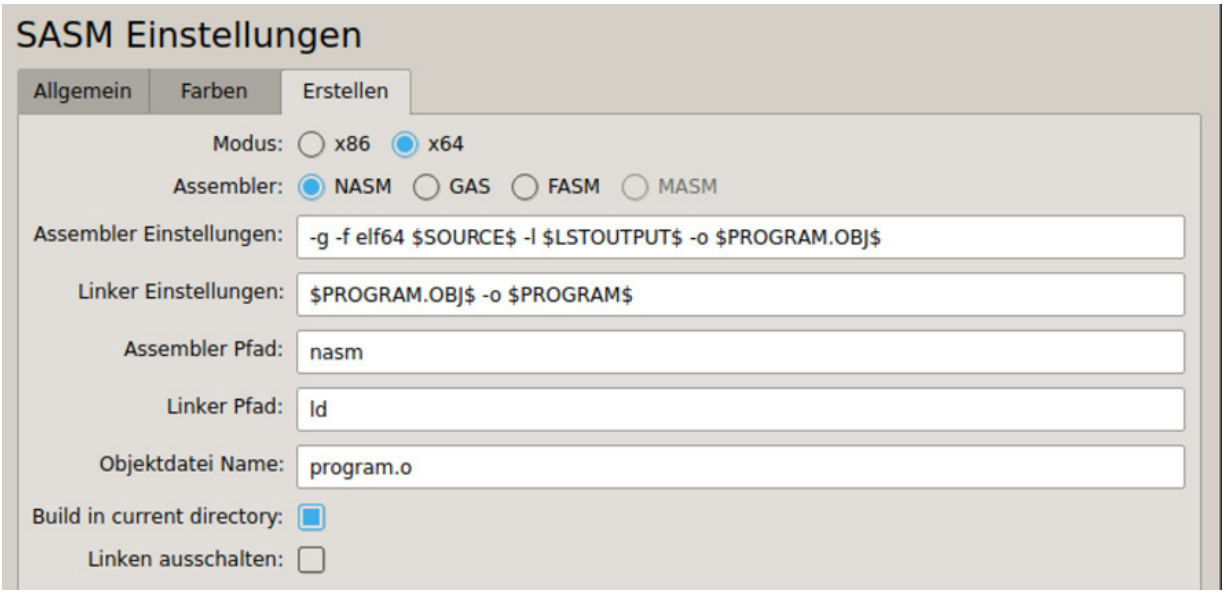
Der Opcode `48 be dc 00 60 00 00 00 00 00` steht für `movabs rsi, 0x6000DC` womit wir dem Programm mitteilen, dass der Text bei der Adresse `0x6000DC` beginnt. Diese berechnet der Assembler für uns und ersetzt jedes Vorkommen von `text` im Programm mit der Speicheradresse.

Wie wir später sehen werden haben wir in dem Programm `mov rsi, text` geschrieben also wurde nicht nur die Speicheradresse berechnet und eingesetzt sonder auch

gleich der `mov`-Befehl gegen `movabs` getauscht welcher verwendet wird um eine 64-Bit Adresse in ein Register zu laden. Der Programmcode wird also auch bis zu einem gewissen Grad bei der Umwandlung optimiert.

Als IDE für Assembler wollen wir SASM verwenden. SASM ist für Linux und Windows unter https://dman95.github.io/SASM/english.html zu beziehen und kann sowohl mit MASM-, FASM-, NASM- und GAS-Code umgehen. Außerdem bietet die IDE einen Debugger, mit dem wir später noch öfter arbeiten werden.

Laden Sie bitte die entsprechende Installationsdatei herunter, installieren und starten Sie das Programm. Wenn Sie auf `Einstellungen` -> `Einstellungen` klicken und den Tab `Erstellen` wählen füllen Sie bitte den entsprechenden Zeilen und Felder wie folgt aus:



ZAHLENSYSTEME

Wenn wir mit Assembler arbeiten sollten wir uns auch kurz die wichtigsten Zahlensysteme genauer ansehen.

Im Falle einer derart langen Kolonne von Nullen und Einsen oder eine Zeichenfolge wie `b801000000` mag es relativ eindeutig sein um welches Zahlensystem es sich handelt aber bei Zahlen wie `10` oder `100` ist die binäre, dezimale und hexadezimale Schreibweise denkbar. Daher wird binären Zahlen ein `0b` und hexadezimalen Zahlen ein `0x` vorangestellt, um das verwendete Zahlensystem explizit anzugeben. Alternativ dazu kann man auch `b` oder `h` hinten anfügen, um einen Wert als binär oder hexadezimal zu kennzeichnen.

So sind die folgenden fünf Befehle gleichbedeutend:

```
mov rax, 0b1010
mov rax, 1010b
mov rax, 0xA
mov rax, 0Ah
mov rax, 10
```

In jedem der Fälle wird dem Register `rax` die Dezimalzahl 10 zugewiesen. Was genau Register sind, sehen wir uns in einem folgenden Kapitel an. In diesem Beispiel müsste man darauf achten, dass man `0Ah` schreibt, nicht nur `Ah` denn sonst würde der Assembler den Wert aus dem `ah`-Register zuweisen. Bei eindeutigen Zahlen wie zB `10h` für den Dezimalwert 16 ist eine führende `0` nicht nötig.

Ich bevorzuge die Schreibweise mit `0b10` bzw. `0x10` da in diesen Fall ein solcher Fehler erst gar nicht passieren könnte!

Das hexadezimale Zahlensystem
... basiert auf 16. Hierbei stehen die Ziffern `0-9` für die jeweiligen Werte, `A` entspricht 10, `B` entspricht 11, usw. bis zum Buchstaben `F` welcher für 15 steht.

Das binäre Zahlensystem
... basiert auf 2. Es werden also nur die Ziffern `0` und `1` verwendet um eine Zahl darzustellen.

Sehen wir uns das anhand von ein paar Beispielen an:

Zahl	binär	hexadezimal	dezimal
10	0x1 + 1x2 = 2	0x1 + 1x16 = 16	0x1 + 1x10 = 10
11	1x1 + 1x2 = 3	1x1 + 1x16 = 17	1x1 + 1x10 = 11
100	0x1 + 0x2 + 1x(2x2) = 4	0x1 + 0x16 + 1x(16x16) = 256	0x1 + 0x10 + 1x(10x10) = 100
110	0x1 + 1x2 + 1x(2x2) = 6	0x1 + 1x16 + 1x(16x16) = 272	0x1 + 1x10 + 1x(10x10) = 110
111	1x1 + 1x2 + 1x(2x2) = 7	1x1 + 1x16 + 1x(16x16) = 273	1x1 + 1x10 + 1x(10x10) = 111

Wenn Ihnen das nun zu schnell ging oder Sie keine Lust haben, kompliziert herumzurechnen, dann kann ich Ihnen nur empfehlen, ein Taschenrechner-Programm zu nutzen oder die Python Shell:

```
>>> bin(0xb801000000)
'0b1011100000000001000000000000000000000000'

>>> hex(0b1011100000000001000000000000000000000000)
'0xb801000000'

>>> int('0xb801000000', 16)
790290759680
```