

# **Assemblerprogrammierung auf der Py Register Machine**

*basics01*

Daniel Knüttel

07.12.2015

# Contents

<b>I. Arbeiten mit der Py Register Machine</b>	<b>3</b>
<b>1. Arbeitsumgebung</b>	<b>4</b>
1.1. Start . . . . .	4
1.2. In der Befehlszeile . . . . .	4
1.3. Mit der GUI . . . . .	5
<b>2. Erste Schritte</b>	<b>7</b>
2.1. Ein erstes Programm . . . . .	7
2.2. Und noch ein Programm . . . . .	7
2.3. Direkte Werte . . . . .	8
2.4. Zum Basteln . . . . .	8
<b>3. Kontrollstrukturen</b>	<b>9</b>
3.1. Sprungadressen . . . . .	9
3.1.1. Zum Basteln . . . . .	10
3.2. Bedingtes Springen . . . . .	10
<b>II. Anhang</b>	<b>12</b>
<b>4. Lösungen</b>	<b>13</b>
4.1. Erste Schritte . . . . .	13
4.2. Sprungadressen . . . . .	13
<b>III. Copyright</b>	<b>15</b>

## **Part I.**

# **Arbeiten mit der Py Register Machine**

# 1. Arbeitsumgebung

## 1.1. Start

Zunächst müssen die Notwendigen Vorraussetzungen geschaffen werden:

**Download** Das ganze Programm von github.com herunter laden, entweder via git

```
1 git clone https://github.com/daknuett/py_register_machine
```

oder die Zip-Datei herunterladen und entpacken

```
1 unzip py_register_machine-master.zip
```

**Kompilieren** Ein Teil des Programms ist in c geschrieben. Es muss deshalb Kompiliert werden. Dies erfolgt durch die Eingabe

```
1 make
```

**Generieren einer Prozessordefinition** Da der emulierte Prozessor generisch (d.h. erweiterbar ) ist, muss bevor ein Assemblieren von Programmen möglich ist eine Prozessordefinition erstellt werden. Standartmäßig nennt man sie prc.def. Sie wird durch die folgende Eingabe erstellt:

```
1 python3 main.py procdef prc.def
```

## 1.2. In der Befehlszeile

Wer sich wirklich mit seinem Rechner auseinandersetzen will, benutzt natürlich eine Shell. In meinem Fall ist das die GNU Bash. Um mit der Bash und der Registermaschine arbeiten zu können, benötigt man eigentlich nur einen Editor. Dabei bieten sich vor allem zwei an:

**Vim** oder auch vi ist ein sehr mächtiger Editor (mein Favourite), aber für Anfänger nicht immer ganz einfach zu benutzen.

**GNU Nano** ist viel einfacher und für einen blutigen Anfänger wohl am besten geeignet.

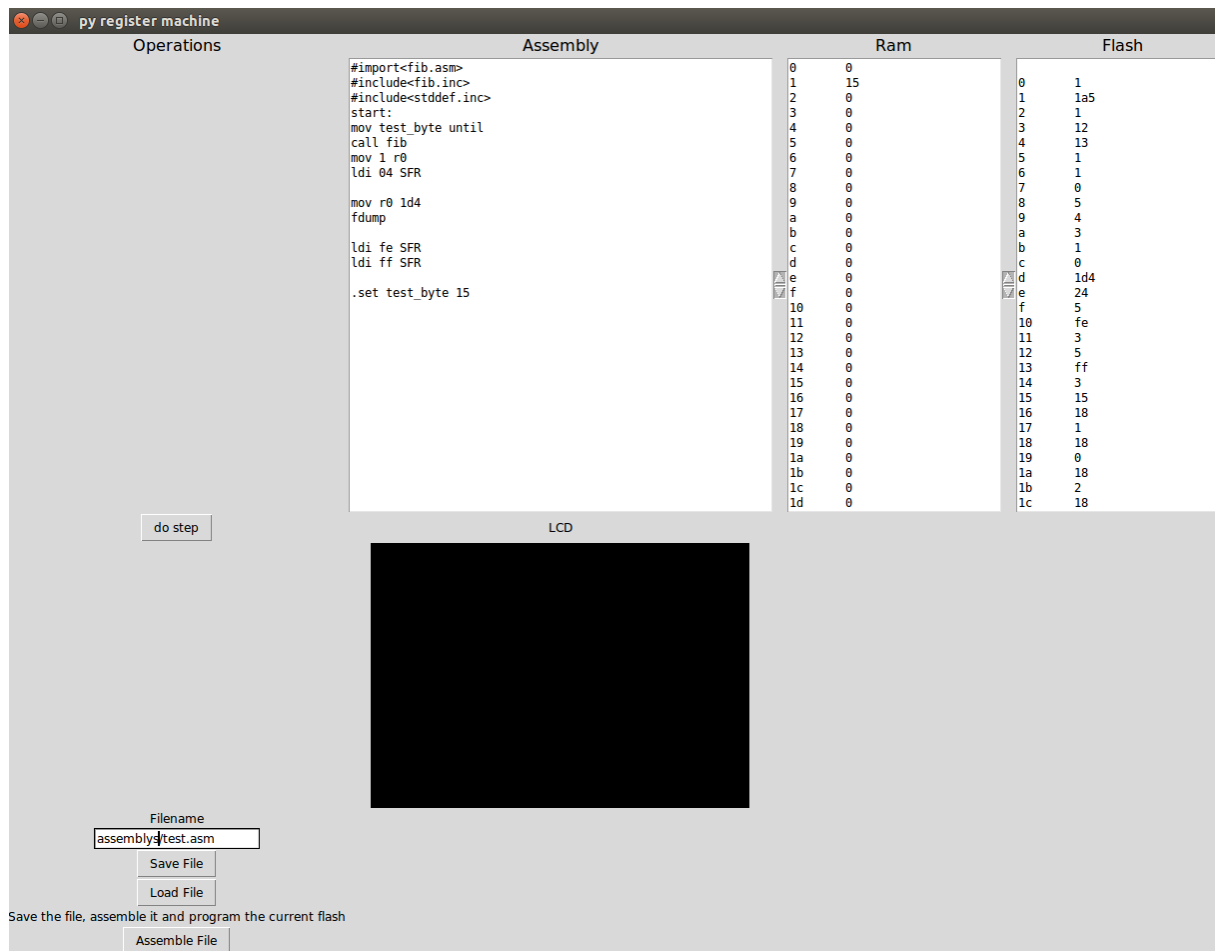


Figure 1.1.: Die einfache Graphische Oberfläche: sGUI

Es gibt auch noch *diesen anderen Editor* aber der ist nicht so super.

Dann muss man sich eigentlich nur noch diese Befehle merken:

```
1 vim name.asm # editieren
2 python3 assemble.py -f name.asm -p prc.def -o name.flash # assemblieren
3 python3 main.py execute name.flash # ausfuehren
```

## 1.3. Mit der GUI

Es wird immer eine **einfache** GUI mitgeliefert. Sie so gestartet:

```
1 python3 sGUI.py
```

Das Bild 1.1 zeigt die Ansicht in sGUI. Die Benutzung ist sehr einfach:

**Laden von Dateien** Den absoluten oder relativen Pfad in das Eingabefeld *Filename* eingeben und den Button *Load File* drücken

**Speichern von Dateien** Den absoluten oder relativen Pfad in das Eingabefeld *Filename* eingeben und den Button *Save File* drücken

**Assemblieren einer Datei** Der Name der Datei sollte schon im Eingabefeld *Filename* stehen, ansonsten muss er dort eingegeben werden. **Wichtig:** Die Datei wird immer vor dem Assemblieren gespeichert. Dann den Button *Assemble File* drücken.

**Editieren einer Datei** Man kann im Textfeld *Assembly* den Quellcode eingeben.

**Ablaufen lassen** Momentan kann man (leider) nur einzelne Schritte machen. Dafür muss man den Button *do step* drücken.

## 2. Erste Schritte

### 2.1. Ein erstes Programm

Eigentlich alles was ein Programm für die Registermaschine braucht ist ein sauberes Ende. Wird das vergessen wird ein *SIGSEGV* gemeldet, weil ein nicht korrekter Befehl vorliegt. Bei einem sauberen Ende muss man unterscheiden, ob man in der GUI ist, oder nicht. Für gewöhnlich fährt man die Registermaschine mit

```
1 ldi ff SFR
```

herunter, allerdings beendet das die Virtuellemaschine komplett, auch die GUI würde beendet. Deshalb sollte man in der GUI den Programmablauf mit

```
1 ldi fe SFR
```

stoppen.

Ein erstes Programm sieht deshalb so aus:

```
1 #include <stddef.inc>
2 ldi fe SFR
```

Die Zeile *#include<stddef.inc>* bindet eine Datei mit Definitionen ein. Sie ist nötig, damit Namen wie *SFR* benutzt werden können.

### 2.2. Und noch ein Programm

Damit ist auch schon der erste (und wohl einer der wichtigsten) Befehl genannt: *ldi*. Dieser Befehl lädt einen hexadecimalen Wert (hier *ff* oder *fe*) in eine Speicherstelle (hier *SFR*)

Der Register *SFR* ist der *Special Function Register*, d.h. wenn man einen Wert in ihn schreibt, wird eine Funktion ausgeführt. Ein weiterer wichtiger Wert für den *SFR* ist *04*. Wenn dieser Wert geschrieben wird, gibt die Registermaschine den Wert von *r0* aus (**Wichtig:** GUI-Benutzer: Das wird ausgegeben, d.h. in dem Terminal mit dem ihr die GUI gestartet habt kann man den Wert sehen ).

Und um gleich noch einen wichtigen Befehl zu nennen, ist da *add*. Er addiert zwei Werte.

( eine Liste aller Befehl findet man hier : [https://github.com/daknuett/py\\_register\\_machine/wiki/Assembly-Directives-and-Mnemonics](https://github.com/daknuett/py_register_machine/wiki/Assembly-Directives-and-Mnemonics) )

Damit kann man schon ein Programm schreiben, das zwei Werte addiert:

```
1 #include <stdint.h>
2 ; ein Strichpunkt ( = Semikolon ) markiert einen Kommentar
3 ldi 5 r0
4 ldi 6 r1
5 ; r0 und r1 sind Register
6 add r1 r0
7 ; ausgeben
8 ldi 04 SFR
9 ; ende
10 ldi fe SFR
```

## 2.3. Direkte Werte

Wer das Programm von oben abgewandelt hat und beispielsweise *ldi 10 r1* genutzt hat, wird sich vielleicht wundern, wieso das Ergebnis der Rechnung nicht *15*, sondern *21* ist. Das liegt daran, dass alle normal eingegebenen direkten Werte als hexadecimal interpretiert werden.

Wer nicht immer im Kopf von decimal nach hexadecimal umrechnen will, kann das mit einer interaktiven Pythonshell machen:

```
1 >>> hex(20)
2 '0x14'
3 >>> int("20",16)
4 32
```

Wer Characterwerte nutzen will, z.B. um Buchstaben auszugeben, kann den Buchstaben in Hochkomma setzen( *'F'* ).

## 2.4. Zum Basteln

Schreibe ein Assemblerprogramm, das drei Werte multipliziert und dann ausgibt. (Hinweis: mit *mov* kann man Werte copieren und verschieben)



## 3. Kontrollstrukturen

### 3.1. Sprungadressen

Um ein Programm zu strukturieren und um Kontrollstrukturen zu verwirklichen( dazu später noch mehr) benötigt man Sprungadressen. Sprungadressen sind eigentlich nur ein Platzhalter, die später vom Assembler ausgefüllt werden. Eine Sprungadresse kann von Sprungbefehlen genutzt werden. Zunächst die beiden einfachsten Sprungbefehle: *jmp* und *call*. Mit *jmp* springt man einfach zu einer Adresse, die darauf folgenden Befehle werden ausgeführt:

```
1 #include<stdint.h>
2 ldi 5 r0
3 ldi 6 r1
4 jmp addieren
5 subtrahieren:
6 ; nicht ausgeführt
7 sub r1 r0
8 ldi 04 SFR
9 jmp ende
10 addieren:
11 add r1 r0
12 ldi 04 SFR
13 jmp ende
14 ende:
15 ldi fe SFR
```

Es wird  $6 + 5 = 11$  ausgegeben und nicht  $6 - 5 = 1$ .

Mit *call* kann man das noch vereinfachen:

```
1 #include<stdint.h>
2 ldi 5 r0
3 ldi 6 r1
4 call addieren
5 ldi fe SFR
6
7 addieren:
8 add r1 r0
9 ldi 04 SFR
10 ret
```

```

11 ; ret ist wichtig!
12
13 subtrahieren:
14 sub r1 r0
15 ldi 04 SFR
16 ret

```

**Wichtig** ist, dass man bei *call* **immer** ( und nur dann!! ) ein *ret* anhängt.

Wenn man bei *jmp addieren* (oder bei *call*) die andere Sprungmarke einsetzt, wird entsprechend der andere Block ausgeführt.

### 3.1.1. Zum Basteln

Erstelle ein Programm, das drei Werte nimmt und diese zuerst addiert und ausgibt, danach subtrahiert und ausgibt und schließlich multipliziert und ausgibt.

(Hinweis: vorrausdenkende Registerbelegung ist wichtig und *mov* ist mit Sicherheit nützlich)

## 3.2. Bedingtes Springen

Wenn man nur unbedingt Springen kann, ist das nicht sehr hilfreich. Deshalb gibt es einige bedingte Sprungbefehle ( kein bedingtes *call* ).

Die vollständige Liste ist wieder auf [https://github.com/daknuett/py\\_register\\_machine/wiki/Assembly-Directives-and-Mnemonics](https://github.com/daknuett/py_register_machine/wiki/Assembly-Directives-and-Mnemonics).

Die Anwendung ist ganz einfach: *<bedingter sprungbefehl> <bedingung> <sprungadresse>*, dabei ist die Bedingung eine Speicherstelle, die mit 0 verglichen wird.

```

1 #include <stddef.inc>
2 ldi 1 r0
3
4 jne r0 nicht_null
5 jeq r0 schon_null
6
7 nicht_null:
8 ldi 5 r0
9 ldi 04 SFR
10 jmp ende
11
12 schon_null:
13 ldi 3 r0
14 ldi 04 SFR
15 jmp ende
16
17 ende:

```

```
18 ldi fe SFR
```

Wenn also  $r0 == 0$  wahr ist, wird nach *schon\_null* gesprungen und eben anders herum.  
Dadurch kann man leicht Schleifen implementieren:

```
1 #include <stdint.h>
2
3 ldi 5 r0
4 ; 5 iterationen
5
6 schleife:
7 ldi 04 SFR
8 ; dec erniedrigt eine Speicherzelle
9 dec r0
10 ; groesser als 0
11 jgt r0 schleife
12 ; jetzt schleife zu ende
13 ldi fe SFR
```

**Part II.**

**Anhang**

## 4. Lösungen

### 4.1. Erste Schritte

Um drei Werte zu multiplizieren muss man sie zuerst laden, und dann nacheinander multiplizieren:

```
1 #include <stdint.h>
2 ldi 5 r0
3 ldi 5 r1
4 ldi 5 r2
5
6 mul r2 r1
7 ; ergebnis r2 * r1 in r1
8 mul r1 r0
9
10 ldi 04 SFR
11 ldi fe SFR
```

Mit *mov*:

```
1 #include <stdint.h>
2 ldi 5 r0
3 ldi 5 r1
4 ldi 5 r2
5
6 mul r0 r1
7 mul r1 r2
8 ; ergebnis jetzt in r2
9 mov r2 r0
10
11 ldi 04 SFR
12 ldi fe SFR
```

### 4.2. Sprungadressen

Am besten ist die Implementation mit *call*. Wenn man in *r0* und *r1* rechnet, kann man die Werte beispielsweise in *r2* und *r3* speichern:

```
1 #include <stdint.h>
```

```
2 ldi 5 r2
3 ldi 6 r3
4 call addieren
5 call subtrahieren
6 call multiplizieren
7 ldi fe SFR
8
9 addieren:
10 mov r2 r0
11 mov r3 r1
12 add r1 r0
13 ldi 04 SFR
14 ret
15
16 subtrahieren:
17 mov r2 r0
18 mov r3 r1
19 sub r1 r0
20 ldi 04 SFR
21 ret
22
23 multiplizieren:
24 mov r2 r0
25 mov r3 r1
26 mul r1 r0
27 ldi 04 SFR
28 ret
```

**Part III.**

**Copyright**

©Copyright 2015 Daniel Knüttel

Dieses Dokument ist Freie Dokumentation, frei, wie FREIheit.

Sie dürfen Kopien davon machen, es verbreiten oder verändern, solange Sie mich als Autor mit angeben und die Änderungen kennzeichnen.

Für den Fall, dass dieses Dokument Fehler enthält, freue ich mich über einen Bericht, schließe eine Haftung für eventuelle Fehler aber aus.

Wenn Sie mit diesen Bedingungen nicht einverstanden sind, dürfen Sie dieses Dokument nicht nutzen.

Um meine und Ihre Rechte zu gewährleisten, steht es unter der GNU Free Documentation License. Sie ist hier einsehbar: <http://www.gnu.org/licenses/fdl-1.3.de.html>