Assemblerprogrammierung auf der Py Register Machine

*basics*01

Daniel Knüttel

07.12.2015

Contents

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

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

oder die Zip-Datei herunterladen und entpacken

```
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

```
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:

```
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 Favourit), aber für Anfänger nicht immer ganz einfach zu benutzen.

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

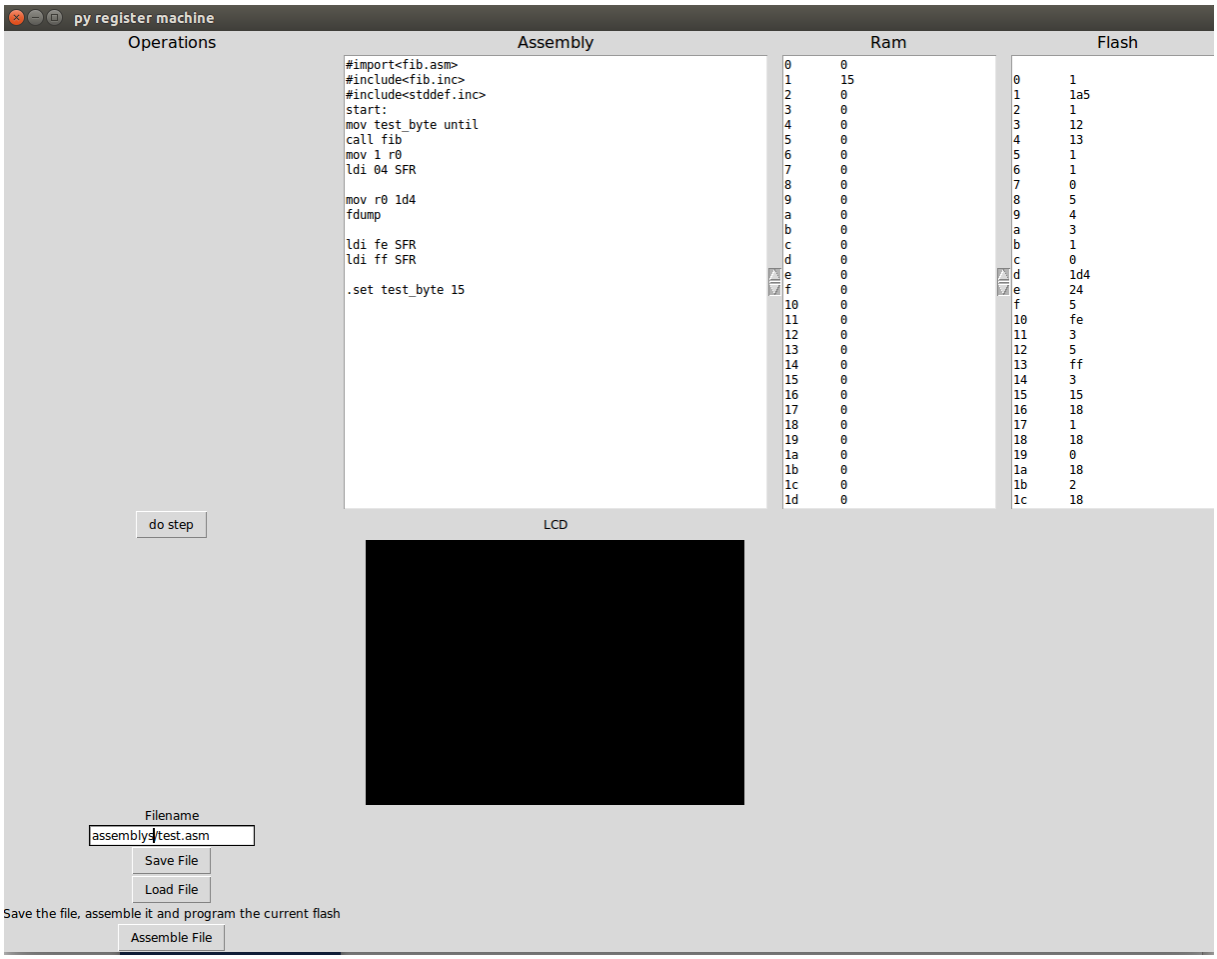

Figure 1.1.: Die einfache Graphische Oberfläche: sGUI

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

Dann muss man sich eigentlich nur noch diese Befehle merken:

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

1.3. Mit der GUI

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

```
python3 sGUI.py
```

Das Bild 1.1 zeit 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

```
ldi ff SFR
```

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

```
ldi fe SFR
```

stoppen.

Ein erstes Programm sieht deshalb so aus:

```
#include<stddef.inc>
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`)

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

```
#include<stddef.inc>
; ein Strichpunkt ( = Semikolon ) markiert einen Kommentar
ldi 5 r0
ldi 6 r1
; r0 und r1 sind Register
add r1 r0
; ausgeben
ldi 04 SFR
; ende
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:

```
>>> hex(20)
'0x14'
>>> int("20",16)
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. Sprungaddressen

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

```
#include<stddef.inc>
ldi 5 r0
ldi 6 r1
jmp addieren
subtrahieren:
; nicht ausgefuehrt
sub r1 r0
ldi 04 SFR
jmp ende
addieren:
add r1 r0
ldi 04 SFR
jmp ende
ende:
ldi fe SFR
```

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

Mit *call* kann man das noch vereinfachen:

```
#include<stddef.inc>
ldi 5 r0
ldi 6 r1
call addieren
ldi fe SFR
```

```
addieren:
add r1 r0
ldi 04 SFR
ret
; ret ist wichtig!

subtrahieren:
sub r1 r0
ldi 04 SFR
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`.

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

```
#include<stddef.inc>
ldi 1 r0

jne r0 nicht_null
jeq r0 schon_null

nicht_null:
ldi 5 r0
ldi 04 SFR
```

```
jmp ende

schon_null:
ldi 3 r0
ldi 04 SFR
jmp ende

ende:
ldi fe SFR
```

Wenn also $r0 == 0$ wahr ist, wird nach *schon_null* gesprungen und eben anders herum.

Dadurch kann man leicht Schleifen implementieren:

```
#include<stddef.inc>

ldi 5 r0
; 5 iterationen

schleife:
ldi 04 SFR
; dec erniedrigt eine Speicherzelle
dec r0
; groesser als 0
jgt r0 schleife
; jetzt schleife zu ende
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 multplizieren:

```
#include<stddef.inc>
ldi 5 r0
ldi 5 r1
ldi 5 r2

mul r2 r1
; ergebnis r2 * r1 in r1
mul r1 r0

ldi 04 SFR
ldi fe SFR
```

Mit *mov*:

```
#include<stddef.inc>
ldi 5 r0
ldi 5 r1
ldi 5 r2

mul r0 r1
mul r1 r2
; ergebnis jetzt in r2
mov r2 r0

ldi 04 SFR
ldi fe SFR
```

4.2. Sprungaddressen

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

```
#include<stddef.inc>
ldi 5 r2
ldi 6 r3
call addieren
call subtrahieren
call multiplizieren
ldi fe SFR

addieren:
mov r2 r0
mov r3 r1
add r1 r0
ldi 04 SFR
ret

subtrahieren:
mov r2 r0
mov r3 r1
sub r1 r0
ldi 04 SFR
ret

multiplizieren:
mov r2 r0
mov r3 r1
mul r1 r0
ldi 04 SFR
ret
```