Assemblerprogrammierung auf der Py Register Machine

*basics*02

Daniel Knüttel

07.12.2015

Contents

1. Abstract

Nachdem im Dokument Assemblerprogrammierung auf der *Py Register Machine - basics01* der Einstieg in die Assemblerprogrammierung auf der Py Register Machine gezeigt wurde, sollen nun die Besonderheiten und Features vermittelt werden.

Zudem wird eine große Menge an Aufgaben mitgebracht, die weitere Grundkentnisse vermitteln sollen.

Part I.

Arbeiten mit der Py Register Machine

2. Anpassen der Engine

2.1. Befehlssatz anpassen

Die Anpassung des Befehlssatzes ist bei der Py Register Machine besonders vereinfacht worden.

2.1.1. Methodik

Die Anpassung des Befehlssatzes erfolgt bei der Py Register Machine bei der Erzeugung der Prozessors:

Man kann dem Prozessor mehrere Maps mit den entsprechenden Befehlen übergeben, dabei wird immer ein *Opcode* (*int*) einem *Namen* (*str*) zugewiesen. Dabei ist zu unterscheiden, wie viele Argumente ein Befehl benötigt[1].

```
# aus main.py

r=Ram(400)
f=Flash(1000)
p=Processor(ram=r,flash=f)
```

```

# angepasst
r=Ram(400)
f=Flash(1000)
# Brozessor kennt nur Befehle mov,ldi,add,sub,mul,inc
p=Processor(ram=r,flash=f,tb_commands=
    {0x1:"mov",0x2:"ldi",
    0x3:"add",0x4:"sub",
    0x5:"mul"},
    sg_commands=
    {0x6:"inc"},
    db_commands={}
    )
```

Deshalb muss für jeden virtuellen Prozessor eine Prozessordefinition generiert werden. Diese Prozessordefinition besteht aus der Größe von Ram und Flash, sowie dem Befehlssatz. Erst dadurch kann ein Programm für den Prozessor assembliert werden.

[1]Das liegt an der Speichernutzung, siehe dazu *design02*

2.1.2. Anwendung

Es gibt eine breite Palette an Anwendungen für solche Anpassungen solche Anpassungen:

Variable Aufgabenstellung Bei unterschiedlichen Aufgabenstellungen kann es nötig sein einen unterschiedlichen Befehlssatz zu nutzen. Dadurch kann ein breites Spektrum an Aufgabentypen abgedeckt werden.

Variable Beweisführung Da eine Registermaschine oft zur Beweisführung dient, um Algorithmen zu testen, ist es oft sinnvoll einen dem Einsatz entsprechened Befehlssatz zu nutzen.

Kompabilität Wenn neue Befehle und Funktionen zur Engine hinzugefügt werden, kann sich der Befehlssatz ändern. Indem man den Befehlssatz manuell ändert kann man Probleme verhindern.

2.2. Speichergröße

Da unterschiedliche Anwendungsbereiche unterschiedlich große Befehls- und Hauptspeichergrößen erfordern, kann auch diese angepasst werden.

Als Anwendungsgebiete sind unter anderem optimiertes Programmieren (wenig Speicher) und Hochsprachen (viel Speicher) zu nennen. Da auch Ein- und Ausgabe über Register (= Speicherstellen) geregelt wird, muss bei neuen Anwendungen und Erweiterungen das Hauptspeicher- und Registerlayout verändert werden.

2.2.1. Befehlsspeichergröße

Die Größe wird dem Befehlsspeicher im Konstruktor übergeben. Das erlaubt eine sehr einfache Anpassung.

```
1 # Befehlsspeichergroesse 300 Bloecke
2 f=Flash(300)
3 # oder 5*10**3 Bloecke
4 f=Flash(5000)
```

2.2.2. Hauptspeichergröße

Beim Hauptspeicher wird ebenfalls die Größe im Konstruktor angegeben.

Wichtig ist allerdings, dass die Register bei dieser Größe mit enthalten sind: bei einer Hauptspeichergröße von 10 Blöcken und 10 Registern muss deshalb im Konstruktor **20** angegeben werden.

Table 2.1.: Registertypen

Typ	Beschreibung
3	Standardregister
1	Specialfunctionregister
2	Outputregister
4	Input-/Outputregister

```
# Ram mit 10 Bloecken und 10 Registern
# Register werden hier noch nicht uebergeben

r=Ram(20,register_count=10)

# nur 10 Register, kein "echter" Ram
r=Ram(10,register_count=10)
```

2.2.3. Registerlayout

Das Registerlayout, also die Anzahl, sowie die Art der Register werden bei der Konstruktion der Hauptspeichers mit angegeben. Dabei werden die Register als eine Stringrepräsentation[2] angegeben.

Die Stringrepräsentation ist so aufgebaut:

$$< stringrepr >= " < registerzahl > / < registerbeschreibung >; \{< registerbeschreibung >\}"$$

$$< registerbeschreibung >=< registerindex >, < registertyp >, < extrainfo >$$

Dabei ist $0 <= registerindex < registerzahl$. Registertyp ist eine Ganzzahl, die den Typ beschreibt(2.1). Der Teil *extra info* wird abhängig vom Registertyp genuntzt, bei Standardregistern beispielsweise wird er ignoriert, bei Outputregistern ist er der Name der Datei, in die geschrieben wird.

Dadurch kann man relativ einfach das Registerlayout ändern:

```
# Standardlayout
r=Ram(400,
   registers=
   "12/0,3,n;1,3,n;2,2,/dev/stdout;3,1,n; \\
   4,3,n;5,3,n;6,3,n;7,3,n;8,3,n;9,3,n;10,4,n;11,4,n;",
   register_count=12)

# weniger Register
r=Ram(400,
   registers=
```

[2] eine Art CSV-Dialekt

```
    "9/0,3,n;1,3,n;2,2,/dev/stdout;3,1,n;4,3,n;5,3,n;6,3,n;7,3,n;8,3,n;",
    register_count=9)
```

3. Aufgaben

Einige dieser Aufgaben wurden an das Schulbuch für Informatik in Bayern[1] angelehnt.

3.1. Einfache Aufgaben

3.1.1. Arithmetic

Schreiben Sie ein Programm, das die Werte $0x04, 0xf4, 0x26$ in Register lädt und miteinander zuerst Addiert und dann Multipliziert. Lösung: 4.1.1.

3.1.2. Umweltverschmutzung

Man kann aus dem CO_2 Ausstoß pro Liter Treibstoff, dem Kraftstoffverbrauch und der zurückgelegten Strecke den Gesamtausstoß an CO_2 berechen.

Gehen Sie von einem CO_2 Ausstoß von $2650\frac{g}{l}$ (Diesel), einem Verbrauch von $5\frac{l}{km}$ und einer Strecke von $300km$ aus.

Schreiben Sie ein Programm, in dem der Gesamtausstoß berechnet wird. Lösung: 4.1.2.

3.1.3. Disassembly

Man kann Maschinencode mithilfe eines Disassemblers in Assermbly zurückführen. Dieses Assembly ist natürlich nicht gut lesbar. Anwenudung findet dies unter anderem, bei Patches für nicht quelloffene Programme.

Interpretieren Sie das folgende Stück "Disassembly" und versuchen Sie herauszufinden, was es tut. Suchen sie eine Anwendung.

```
1 #include<stddef.inc>
2 ldi 4 r0
3 mov r0 20
4 ldi 5 r0
5 mov r0 21
6 ldi 7 r0
7 mov r0 22
8 call LFB0
9 call LFB1
```

[1] Informatik Lehrwerk für Gymnasien 5 (Klett Verlag 2010)

```
ldi ff SFR
LFB0:
mov 20 r0
mov 21 r1
mul r1 r0
mov 22 r1
mul r1 r0
ldi 04 SFR
ret
LFB1:
mov 20 r0
mov 21 r1
mul r1 r0
ldi 2 r1
mul r1 r0
mov r1 r4
mov 20 r0
mov 22 r1
mul r1 r0
ldi 2 r1
mul r1 r0
add r0 r4
mov 21 r0
mov 22 r1
mul r1 r0
ldi 2 r1
mul r1 r0
add r4 r0
ldi 04 SFR
ret
```

Lösung: 4.1.3.

3.2. Kontrollstrukturen

3.2.1. Fibonacci

Die Fibonacci-Reihe ist eine der beliebtesten Zahlenreihen im Universum der Informatiker. Sie lässt sich so definieren:

$$fib_n := fib_{n-1} + fib_{n-2}$$

$$fib_0 := 0$$

$$fib_1 := 1$$

Da eine rekursive Implementation (die durch diese Schreibweise nahegelegt wird) sehr ineffizent ist, kann man sie vorteilhaft iterativ implementieren. Der folgende Python3 code zeigt die Struktur:

```
1
2 def fib (n):
3   """ bis  naechst groesseren
4     Fibonacci-Zahl berechnen
5     und ausgeben """
6   fib=0
7   last=1
8   while(fib<n):
9     print( fib)
10    swp=fib
11    fib+=last
12    last=swp
13  return fib
```

Implementieren Sie einen vergleichbaren Algorithmus.

Lösung: 4.2.1

Part II.

Anhang

4. Lösungen

4.1. Einfache Aufgaben

4.1.1. Arithmetic

```
#include<stddef.inc>
; werte laden

ldi 04 r2
ldi f4 r3
ldi 26 r4

addieren:
mov r2 r0
add r3 r0
add r4 r0
ldi 04 SFR

multiplizieren:
mov r2 r0
mul r3 r0
mul r4 r0
ldi 04 SFR

; fertig
ldi ff SFR
```

4.1.2. Umweltverschmutzung

```
#include<stddef.inc>

; werte laden und in RAM schieben

ldi a5a r0
mov r0 20
; 0x20 = 32

ldi 5 r0
```

```
10 mov r0 21
11
12 ldi 12c r0
13 mov r0 22
14
15 ; laden fertig
16 ; gesamt volumen diesel berechnen
17
18 mov 21 r0
19 mov 22 r1
20
21 mul r0 r1
22
23 ; g pro l laden und einrechnen
24
25 mov 20 r0
26 mul r0 r1
27
28 ; ausgeben
29 mov r1 r0
30 ldi 04 SFR
31 ; fertig
32 ldi ff SFR
```

4.1.3. Disassembly

Das Programm speichert zuerts die Werte 4,5 und 7 im RAM. Dann ruft es zwei Routinen auf und beendet den Ablauf.

Die erste Routine multipliziert die 3 Werte und gibt sie aus.

Die Zweite Routine multipliziert immer zwei der Werte, multipliziert das Teilergebnis mit 2 und Addiert die Ergebnisse.

Eine mögliche Anwendung ist das Berechnen von Volumen (*LFB0*) und Oberfläche (*LFB1*) eines Quaders.

4.2. Kontrollstrukturen

4.2.1. Fibonacci

```
1 ; endwert 32 laden und speichern
2 ldi 20 r0
3 mov r0 20
4
5 ; werte fib und last
6 ldi 0 r0
```

```
7  ldi 1 r1
8  fib_loop:
9  ; ausgeben
10 ldi 04 SFR
11 mov r0 r3
12 add r1 r0
13 mov r3 r1
14 ; testen ob fertig
15 mov r0 r2
16 mov 20 r3
17 sub r2 r3
18 jle r3 fib_loop
19
20 ; fertig
21
22 ldi ff SFR
```

Part III.

Copyright