Registermaschinenentwurf Py Register Machine

*design*02

Daniel Knüttel

11.12.2015

Contents

1. Abstract **3**

I. Registermaschinenentwurf Py Register Machine 4

2. Sprachauswahl **5**

2.1. Python3 5

2.2. C 5

3. Implementationsreihenfolge **6**

4. Speicherorganisation **7**

4.1. Befehlsspeicher 7

4.2. Addressräume 7

II. Anhang 9

5. Ergänzungen **10**

5.1. Initialisierung des Prozessors 10

5.2. Befehlsspeichernutzung 10

1. Abstract

Die konkrete Fortführung von *Registermaschinenentwurf Py Register Machine - design01*, in diesem Dokument stehen Konkrete Entwürfe im Vordergrund.

Part I.

Registermaschinenentwurf Py Register Machine

2. Sprachauswahl

Die Wahl bei der Sprache fiel nicht zufällig[1] auf *python3* und *c*. Es sprechen einige Gründe für diese Sprachen.

2.1. Python3

Wieso es nicht *pyhton2* geworden ist, ist klar: *python3* ist einfach das aktuellere.

Ich wollte beim Entwurf allerdings auf einige wichtige Dinge Wert legen:

Generische Module Alle Module, ins besondere der Prozessor sollten generisch sein. Ein weit verbreiteter Makel ist, dass eine Registermaschine nicht generisch ist, sondern fix. Diesen Makel wollte ich von Beginn an ausmerzen. Das ermöglicht ein breites Anwendungsfeld, von der einfachen Algorithmenprüfung bis zur Hochsprachenentwicklung.

Lesbarkeit Aller Code, besonders die wirklich interessanten Teile, sollten gut lesbar sein[2]. Python bietet einfach die Möglichkeit lesbaren Code mit dem Schwerpunkt auf den interessanten Abschnitten zu produzieren.

Gute Erweiterbarkeit Möglichst jeder soll in der Lage sein die Maschine zu modifizieren und auf seine Bedürfnisse zu zu schneiden, bei Python ist dieses Ziel leicht erreichbar.

Diese Punkte sind natürlich für jeden Entwurf wichtig, auch weitere Zusatzprogramme solten dies erfüllen.

2.2. C

Ein Problem bei Python ist die in designs01 erwähnte variable Größe des Ganzzahltypen. Um dieses Problem zu umgehen, bindet das Programm eine geteilte Bibliothek ein[3], die in c geschrieben ist. Auch diese Bibliothek ist schon Objektorientiert und erlaubt Vererbungsartige Strukturen.

Der Vorteil von c besteht darin, dass es hervorragend von Python eingebunden werden kann und auch noch sehr lesbar ist.

[1] naja, nicht nur...
[2] hat auch nicht überall hingehauen
[3] Der verhindernde Punkt für DOSse

3. Implementationsreihenfolge

Die Reihenfolge bei der Implementation ist essenziell, denn anders als z.T. dargestellt ist nicht die Verarbeitung der Daten essenziell, sondern deren Speicherung. Wie schon in design01 erwähnt, ist eine Registermaschine ein sich selbstverändernder Speicher, deshalb steht zuerst die Implementation der Speicher im Vordergrund. Hierbei liegen im Fall der Py Register Machine alle Speicher im selben Addressbereich, Lesen und Schreiben ist auf alle Speicherformen möglich.

Erst danach können Prozessor und Rechenwerk implementiert werden, dabei kommen, um einen generischen Prozessor zu ermöglichen *map*s zum Einsatz. Im Konstruktor können dem Prozessor die Befehle übergeben werden.

Dabei wird unterschieden, wie viele Argumente ein Befehl benötigt, da für jedes Argument eine eigene Speicherzelle genutzt wird.[1]

[1]Beispiel: 5.1

4. Speicherorganisation

4.1. Befehlsspeicher

Die Py Register Machine nutzt eine etwas ausgefallenere Befehlsspeicherungsmethode:

Argumenttrennung Jedes Argument wird in einer eigenen Speicherzelle abgelegt. Das ermöglicht einen theoretischen Umfang an $2^{63} = 9.22 \cdot 10^{18}$ Befehlen, sowie eine Direktaddressierung von ebenfalls $9.22 \cdot 10^{18}$ Zellen. In der Praxis wird dies allerdings kaum genutzt.

variable Blockzahl Jeder Befehl mit den Argumenten kann eine eigene Zahl an Blöcken nutzen, für den Befehl ein Block, sowie für jedes Argument ein Block. Dadurch wird eine effizientere Speichernutzung möglich, allerdings verkompliziert es das Befehlszählerverhalten und den Aufbau des Assemblers.

Da Debugging durch dieses Verhalten verkompliziert wird, verfügt der Prozessor über ein eigenes Disassembly System.

Ein Beispiel für das Verhalten ist hier: 5.2.

4.2. Addressräume

Um ein möglichst breites Spektrum an Anwendungsmöglichkeiten abzudecken wurden alle Speicherformen in einen Adressraum gelegt, dabei nehmen die Register die untersten Addressen ein ($0 \rightarrow |M_{register}| - 1$, $M_{register}$ ist die Menge der Register), darüber liegt der RAM ($|M_{register}| \rightarrow |M_{register}| + |M_{ram}| - 1$) und wieder darüber liegt der Befehlsspeicher ($|M_{register}| + |M_{ram}| \rightarrow |M_{register}| + |M_{ram}| + |M_{befehle}| - 1$).

Dabei ergibt sich bei 10 Registern, 20 Blöcken Speicher und einem Befehlsspeicher von 200 Blöcken:

$$A_{start}(register) = 0$$

wobei A die Addresse ist

$$A_{end}(register) = |M_{register}| - 1 = 9$$

$$A_{start}(ram) = |M_{register}| = 10$$

$$A_{end}(ram) = |M_{register}| + |M_{ram}| - 1 = 29$$

$$A_{start}(befehle) = |M_{register}| + |M_{ram}| = 30$$

$$A_{end}(befehle) = |M_{register}| + |M_{ram}| + |M_{befehle}| - 1 = 229$$

Part II.

Anhang

5. Ergänzungen

5.1. Initialisierung des Prozessors

Bei der Initialisierung können dem Prozessor die Befehle übergeben werden:

```
p=Processor(tb_commands={0x1:"mov",0x2:"add",0x3:"sub",
    0x4:"ldi",0x05:"jgt"},
    db_commands={0x06:"jmp"},
    sg_commands={})
# sehr geringer Befehlssatz, z.B. fuer
# Implementation von Arithmethischen Ausdruecken
```

5.2. Befehlsspeichernutzung

Ein Beispiel füd die variable Befehlsspeichernutzung.

```

0 5
1 5
2 0         ldi 5 0 ; 3 Speicherzellen
3 5
4 4
5 3         ldi 4 3 ; 3 Speicherzellen
6 7
7 0         dec 0 ; 2 Speicherzellen
8 a
9 0
a fffffffb      jgt 0 fffffffb ; 3 Speicherzellen; fffffffb = -7
b 5
c ff
d 3         ldi ff 3 ; 3 Speicherzellen
```