Registermaschinenentwurf Py Register Machine

$design01$

Daniel Knüttel

11.12.2015

Contents

1. Abstract

Registermaschinen sind ein interressantes informatisches Rechenmodell, sowie eine hervorragende Möglichkeit, um hardwarenahes Programmieren zu lernen. In diesem Dokument sollen die grundliegenden Theorien sowie Entwürfe nähergebracht werden.

Part I.

Registermaschinenentwurf Py Register Machine

2. Begriffe

2.1. Registermaschine

Eine Registermaschine beschreibt ein informatisches Rechenmodell: Sie hat mehrere Zustände (mindestens 2) und kann diese ändern, den Zustand wechseln. Dabei spricht man von einer Transition.

Diese Zustände werden in der Registermaschine durch Speicherzellen repräsentiert, je nach Anwendungszweck bzw. Entwurf werden unterschiedliche Speicherzellen genutzt.

Dies lässt sich auf die Quintessenz der Registermaschine zusammenfassen: *Die Registermaschine ist ein sich selbst modifizierender Speicher.*

2.2. Transition

Ändert eine Registermaschine ihren Zustand, so spricht man von einer Transition, **jeder Befehl**, den eine Registermaschine ausführt ist eine Transition.

2.3. Endlichkeit

Eine Registermaschine ist ein Endlicher Automat. Diese Endlichkeit bezieht sich allerdings nicht, darauf, dass die Registermaschine endlich lange ihren Zustand wechselt, sondern darauf, dass sie eine *endliche Anzahl an Zuständen* besitzt.

Die wohl einfachste Registermaschine ist ein Zähler (2.1): er kann nur einen Befehl ausführen, dieser erhöht die einzige Speicherstelle. Diese Registermaschine hat eine endliche Zahl an Zuständen (die Breite der Speicherstelle), aber eine (theoretisch) unendliche Laufzeit.

Gehen wir von einem Zähler mit der Breite 4 bit aus, so ergeben sich die folgenden Zustände:

```
1 binaer decimal
2 0000 = 0
3 0001 = 1
4 0010 = 2
5 0011 = 3
6 0100 = 4
```

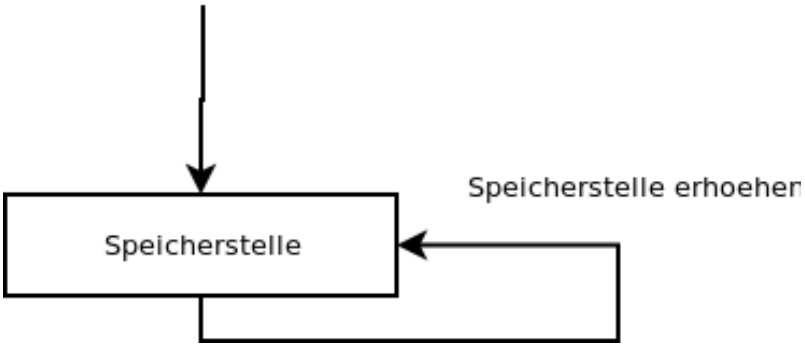

Figure 2.1.: Flussdiagramm eines Zählers: ein endlicher Automat

```
0101 = 5
0110 = 6
0111 = 7
1000 = 8
1001 = 9
1010 = 10
1011 = 11
1100 = 12
1101 = 13
1110 = 14
1111 = 15
```

[1]

Die Anzahl der Zustände lässt sich also so berechen:

$$|Z| = |S| * (2 * *|s|)$$

[2]

Wobei Z die Menge der Zustände ist, S die Menge der Speicherstellen und s eine Speicherstelle. Im fall des 4 bit Zählers:

$$|Z| = 1 * (2 * *4) = 16$$

Vorsicht bei selbstexpandierended Sprachen, wie *python3*: hier können mithilfe von Ganzzahlen keine Endlichen Automaten realisiert werden, da die Anzahl der Zustände nicht endlich ist:

```
zaehler = 0

while(1):
  zaehler += 1
```

[1] siehe auch: 5.1
[2] $x * *y$ ist die Potenz zur Basis x mit dem Exponent y

ist **kein** endlicher Automat, da *zaehler* keine feste Breite kennt. Deshalb ist die einzige in der Py Register Machine in *python3* implementierte Speicherzelle der Befehlszähler. Die Breite des Befehlszählers ergibt sich durch den gesamten Addressbereich:

$$|\mathrm{Z}_{\mathrm{Befehlszaehler}}| = 1 * (|Ram| + |Reg| + |Flash|)$$

Wenn der Befehlsspeicher einen getrennten Addressbereich hat, ist die Breite des Befehlszählers die Größe des Befehlsspeichers:

$$|\mathrm{Z}_{\mathrm{Befehlszaehler}}| = |Flash|$$

2.4. Automaten

Da die Registermaschine ein *Deterministischer Endlicher Automat* (*DEA*)[3] ist, kann ein Registermaschinenentwurf theoretisch durch die Definition eines Automaten beschrieben werden:

$$A = (Z, \Sigma, \delta, S, E)$$

Dabei ist A der Automat, Z ist die Menge der Zustände, Σ sein Eingabealphabet (respektive sein Befehlssatz), δ die Menge aller Transitionen, S der Startzustand und E die Menge der Endzustände, die in Z enthalten sind.

δ ergibt sich aus Σ und Z, da immer $(zx\sigma) \to z$. Wichtig ist hierbei, dass $|Z| \neq \infty$, da sonst kein endlicher Automat entstehen kann.

Bei einer Registermaschine muss davon ausgegangen werden, dass $E = Z$[4], daraus wird erkenntlich, dass eine derartige Notation schnell ihren Sinn verliert, da $|\delta| = |\Sigma| * |Z|$. Bei nur 10 Registern mit einer Breite von 64 bit und 5 Befehlen mit zwei Argumenten (die nur Register sein dürfen)[5] ergibt sich

$$|\delta| = |\Sigma| * |Z| = |\Sigma| * (10 * 2^{63}) = (5 * 10 * 10) * (10 * 2^{63}) \approx 4,612 * 10^{22}$$

[6]

Besonders interessant ist der Umstand, dass man **alle** Automatentypen ineinander umwandeln kann: eine *Registermaschine* kann einen *Kellerautomaten* emulieren, ein *Kellerautomat* kann einen *Turingautomaten* simulieren und ein *Turingautomat* kann eine *Registermaschine* emulieren. Das ist ein wichtiger Grundsatz: ein *DEA* muss von einem

[3] im Englischen *DFA Deterministic Finite Automaton*
[4] außer man muss einen Registerwert setzen, um die Maschine zu stoppen
[5] eigentlich besteht jedes Element von Σ aus dem Befehl und den Argumenten
[6] Wichtig ist hier die Berechnung mit ausreichend breiten Prozessoren, auf 32 bit könnte es knapp werden.

Turingautomaten umgesetzt werden können[7].

Wichtig ist immer das **Prinzip**: Registermaschinen können groß sein[8], eine Umsetzung in einen Turingautomaten oder die Notation als 5er Tupel muss nur *möglich* sein, nicht durchgeführt werden.

[7] dadurch kann er von jedem anderen Automaten umgesetzt werden und diesen Umsetzen
[8] Beleg erforderlich[9]

3. Anwendungen

3.1. Theoretische Informatik

Eine Registermaschine ist ein beliebes Mittel, um zu zeigen, dass eine Prozedur[1] deterministisch ist: lässt sie sich in einer Registermaschine umsetzen, so ist sie deterministisch[2].

Ein einfaches Beispiel ist die Fibonacci-Reihe, sie ist so definiert:

$$fib(n) := fib(n-1) + fib(n-2)$$

$$fib(1) := 1$$

$$fib(0) := 0$$

$$n \in \mathbb{N}_0$$

sie ist sogar beides: deterministisch und terminiert (für $n \neq \infty$) und kann in einer Registermaschine leicht umgesetzt werden.

Damit ist bewiesen, dass die Fibonacci-Reihe deterministisch ist.

3.2. Lernhilfsmittel

Eine Registermaschine kann (wenn sie entsprechend nah an realen Prozessoren ist) zum Erlernen von maschinennahem Programmieren genutzt werden. Zudem kann sie genutzt werden, um anderes maschinennahes Programmieren zu lernen, wie Hochsprachenentwickung und Ähnliches.

[1] damit ist nicht eine Pascal-procedure gemeint.
[2] aber nicht umbedingt terminiert.

4. Entwürfe

4.1. Grundstrukturen

Grundsätzlich geht man bei einer Registermaschine von mindestens diesen Bestandteilen aus:

Register Ein oder mehr Register repräsentieren den Zustand des Automaten, dabei kann der Befehlszähler ein Register sein oder im Prozessormodul integriert sein, er kann allerdings als Register gewertet werden.

Befehlsspeicher Der Befehlsspeicher enthält die Befehle, dabei zeigt der Befehlszähler immer auf die Stelle an der der aktuelle Befehl liegt.

(optional) Speicher Der Speicher, oft auch Hauptspeicher oder RAM wird genutzt um die Register zu ergänzen. Oft ist er allerdings überflüssig.

Dabei werden in der Praxis diverse Addressierungsformen genutzt:

Getrennte Addressbereiche Der Befehlsspeicher nutzt einen komplett anderen Addressbereich als der RAM oder die Register, er ist meist nicht im Programm ansteuerbar, alternativ meist nur lesbar. Dieser Entwurf liegt sehr nah an realen Mikroprozessoren.

Geteilter Addressbereich Befehlsspeicher und auf alle Fälle der RAM liegen im selben Addressbereich. Das ermöglicht die Verarbeitung von Zeichenketten, sowie die Ausführung von Befehlen aus dem RAM.

Registertrennung Die Register sind vom RAM getrennt, also beispielsweise direkt im Prozessor implementiert. Dieses Modell entspricht ebenfalls eher realen Prozessoren

Registerintegration Die Register sind im Addressbereich des RAM enthalten, der RAM ist also wie eine große Anzahl von Registern. Die Register werden allerdings oft trotzdem getrennt implementiert, da sie spezielle Funktionen haben können.

Bei der Py Register Maschine wurden die Konzepte des Geteilten Addressbereichs und der Registerintegration genutzt.

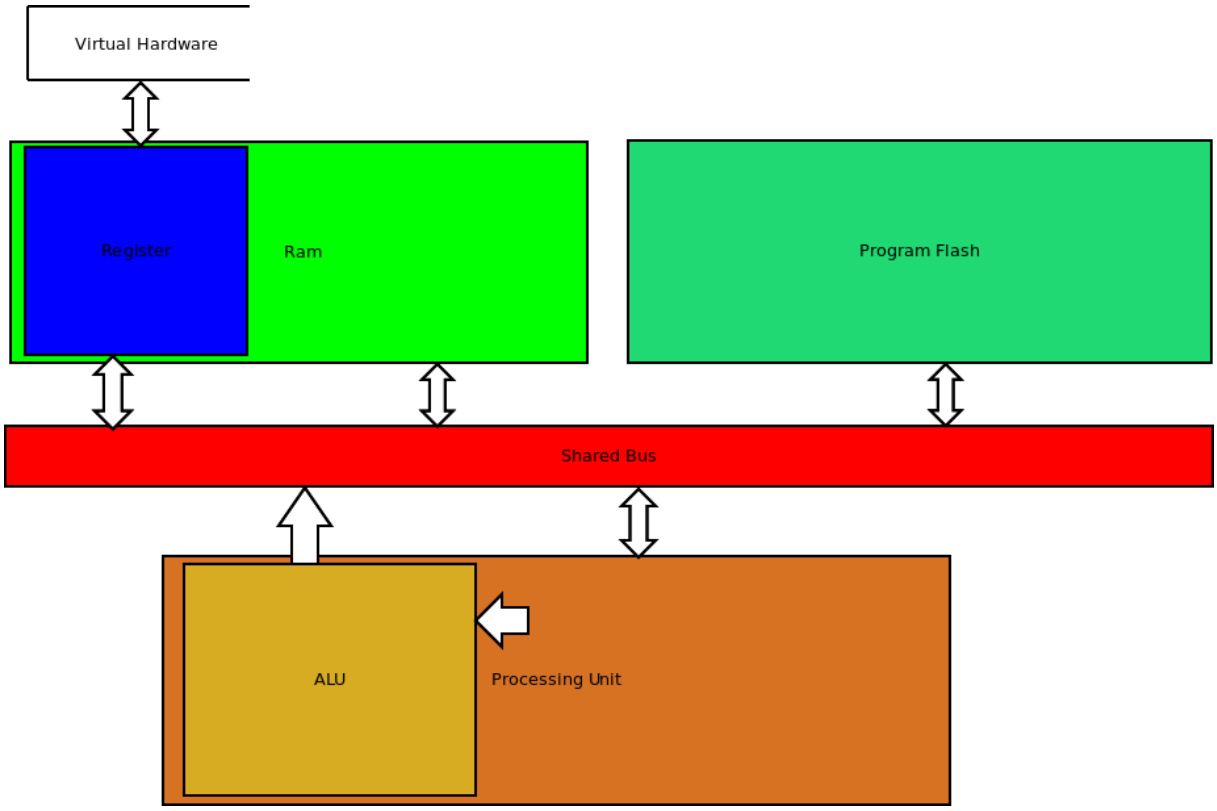

Figure 4.1.: Blockdiagramm der Py Register Machine

4.2. Diagramme

Um den Aufbau eines Entwurfs darstellen zu können, sind Diagramme natürlich von Vorteil, besonders sind hier die Folgenden hervorzuheben:

Blockdiagramm Das Blockdiagramm kommt aus dem Bereich der Prozessorentwickung und ist deshalb hervorragend geeignet um innere Zusammenhänge, sowie die Addressbereiche darzustellen. Dabei werden Module als (evtl. farbige,) beschriftete Rechtecke dargestellt, Busse werden ebenfalls als Rechtecke dargestellt, alle Module an einem Bus haben den selben Addressbereich. Blockpfeile zeigen mögliche Datenströme an. (4.1)

Flussdiagramm Das Flussdiagramm erlaubt Abläufe, wie einzelne Befehle, usw. detailliert und übersichtlich dar zu stellen. Jeder Prozess wird dabei als Rechteck, eine Entscheidung als Raute dargestellt. Der Fluss wird von beschrifteten Pfeilen angezeigt. (2.1)

4.3. Module

Sinnvoll ist die Unterteilung in mehrere Speichermodule, sowie den Prozessor:

Speicher Der Speicher oder RAM sollte durch ein Objekt repräsentiert werden, am besten von einer Klasse *Memory* abgeleitet. Bei einer Registerintegration enthält der RAM auch die Register.

Register Jeder Register sollte als eigenes Objekt repräsentiert werden, das erlaubt die Zuweisung von Spezialfunktionen.

Befelsspeicher Auch er sollte ein eigenes Objekt sein. Eine Vererbung von einer Klasse *Memory* erlaubt eine kompatible Nutzung zum RAM.

Prozessor Der Prozessor enthält die anderen Module (eventuell über Umwege). Er hat einen Befehlssatz, mit dem er die Befehle interpretiert.

Rechenwerk Das Rechenwerk (oder ALU [1]) wird fast immer in das Prozessormodul, bzw in dessen Befehlssatz integriert, da eine Aufteilung nicht sinnvoll ist.

Allerdings werden die Register manchmal auch als ein Objekt implementiert, der einzige Unterschied zum RAM besteht dann in den Addressen.

4.4. Speicher

Ein iteressantes Thema über das man allerdings vermutlich Kriege führen kann[2], ist der Speicherentwurf. Dabei gibt es zum Einen die allgemeine Speicherzellenbreite: Je nach Anwendungstyp ist eine andere Breite angebracht, im Allgemeinen wird allerdings niemand bestreiten, dass 64 bit breite Speicherzellen am sinnvollsten sind[3]. Ausnahmen bilden selbstverständliche besondere Rechenmodelle, wie 4 bit Zähler. Zudem muss daran gedacht werden, dass die Registermaschine dann minimal 64 bit addressieren kann, daraus folgt, dass bei ASCII[4] codierten Zeichen ganze 56 bit brachliegen[5], andere Codierungen sind dann allerdings sehr einfach um zu setzen.

Zudem ist der Befehlsspeicher ein wichtiger und überlegenswerter Teil des Entwurfs: Man kann für jeden Teil des Befehls eine eigene Speicherzelle belegen:

```
1 mov r0 r1
2 ...| 01 | 01 | 02 |...
```

eine Alternative ist die bei realen Prozessoren oft genutzte Technik, bei der der Opcode auch noch (zumindest eine) Speicherstellen enthält:

```
1 mov r0 r1
2 ...| 0101 | 02 |...
```

[1] Arithmetic Logical Unit
[2] Man kann über fast Alles in der Informatik Kriege führen.
[3] und es geht los...
[4] Und noch ein Kriegsgrund
[5] 56 bit sind nicht viel, finde ich.

Part II.

Anhang

5. Zusätze

5.1. Binäres Rechenmodell

Das binäre Rechenmodell ist elementar für heutige Computer, es bietet die Möglichkeit nur mit logischen Operatoren Ganzzahle zu verrechen. Der einfachste Fall ist die Addition:

Komplementäre Bits Ist eine Zahl negativ, so sind alle Bits komplementär zur positiven Zahl ($-1+1=0$), bei einer 5 bit Zahl ist also das 5.Bit das Vorzeichen ($0=+$):

$$0b00001 = 1$$

$$0b11111 = -1$$

Addition Die Addition kann wie folgt definiert werden: x, y und z seien Zahlen, man kann diese Zahlen indizieren, indem man jede binäre Stelle als Element nimmt: $0\text{b}01001_3 = 1; 0\text{b}01001_2 = 0$. Dann ist die Addition zweier Stellen so definiert:

$$\mathrm{x_n} = (\mathrm{y_n \hat{z}_n})\hat{}((\mathrm{x_{n-1}}\&\mathrm{y_{n-1}})|(\mathrm{x_{n-1}}\&\mathrm{z_{n-1}})|(\mathrm{y_{n-1}}\&\mathrm{z_{n-1}}))$$

Dabei is $\mathrm{x_{(-1)}} = 0$.

Man kann dabei vorteilhaft auch iterativ statt rekursiv rechnen:

$$0b010011 + 0b100101 = ?$$

erste Stelle:

$$0b1 + 0b1 = 0b0(0b1merken)$$

zweite Stelle:

$$0b1 + 0b0 + 0b1 = 0b0(0b1merken)$$

usw:

$$0b0 + 0b1 + 0b1 = 0b0(0b1merken)$$

$$0b0 + 0b0 + 0b1 = 0b1$$

$$0b1 + 0b0 + 0b0 = 0b1$$

$$0b0 + 0b1 + 0b0 = 0b1$$

gesamt:

$$0b010011 + 0b100101 = 0b111000$$

Subtraktion Da negative Zahlen komplementär sind, wird das Prinzip der Addition angewendet.

Overflow Das rekursive Prinip zeigt deutlich das Problem des Überlaufs: rechnet man mit 8 bit breiten Zahlen, wird x_9 nicht berücksichtigt. Ergebnisse die größer als 8 bit sind, sind deshalb inkorrekt. Dafür wurde auf realen Maschinen das Überlaufbit (bei 8 bit ist das x_9) eingeführt, es ermöglicht Prüfung auf arithmetische Fehler, sowie Addition über zwei Register.

1 + 1 = 0 Das Prinzip des Überlaufs ist bei Zählern oft nützlich: Der 4 bit Zähler zählt immer von 0 bis 15 und beginnt dann wieder bei 0:

$$0b1111 + 0b1 = 0b10000; bei 4bit 0b0000$$

Part III.

Copyright