Maschinensprache

Hans Jürgen Ohlbach

7. November 2017

Keywords: Maschinenanweisungen, Adressierung, ein einfaches Prozessorbeispiel

Empfohlene Vorkenntnisse: Digitale Arithmetik, Prozessorarchitektur

Inhaltsverzeichnis

1 Einführung

Jeder reale oder virtuelle Prozessor wird mit einer Sprache programmiert, die auf seine Architektur abgestimmt ist. In diesem Miniskript wird die Struktur dieser Sprachen eingeführt und die verschiedenen Typen der Maschinenanweisungen besprochen. Schließlich wird ein extrem vereinfachtes Übungsbeispiel für eine konkrete Prozessorarchitektur zusammen mit seiner Sprache vorgestellt. An diesem Beispiel kann man aber die wesentlichen Aspekte der Maschinenprogrammierung sehen.

Ganz wichtige Maschinenanweisungen, insbesondere die Booleschen Operationen zusammen mit den Shift-Operationen, sind auch in den meisten höhere Programmiersprachen zu finden. Wenn man diese richtig ausnutzt, dann kann man sehr effizient programmieren. Einige Beispiel dafür werden auch vorgestellt.

2 Arten von Maschinenanweisungen

2.1 Arithmetische Anweisungen

Jeder moderne Prozessor kann binär addieren, subtrahieren, multiplizieren und dividieren.

Dabei gibt es grundsätzlich die Unterscheidung zwischen

- Integer Zahlentypen, mit einer festen Anzahl von Bits, meist 32 oder 64 Bits, und
- Floating Point Zahlentypen mit Mantisse und Exponent. Dabei gibt es die `single Precision` Typen mit 32 Bits und die `Double Precision` Typen mit 64 Bits. Einige Spezialprozessoren können auch noch längere Floating Point Typen verarbeiten.

Allerdings werden die arithmetischen Operationen für Floating Point Zahlen oft in separaten Coprozessoren durchgeführt. Das bekommt man als Benutzer jedoch nicht mit.

Folgende Phänomene sind aber zu beachten:

- Prozessoren und auch einige Programmiersprachen (z.B. C und C++) unterscheiden zwischen `unsigned integer` und `signed integer` (nicht Java). Bei unsigned Integer werden alle 32 bzw. 64 Bits als *positive* Integerzahlen genutzt. Bei signed Integer wird ein Bit als Vorzeichenbit genutzt. Negative Zahlen werden im *Zweierkomplement* dargestellt.
- Arithmetische Operationen können Überläufe erzeugen. Es gibt zwei Möglichkeiten, damit umzugehen: entweder die Maschine, und damit das Programm erzeugt eine Fehlermeldung, oder eben nicht. Wenn kein Überlauf gemeldet wird, werden unerwartete Zahlen produziert. Z.B. führt ein Überlauf von positiven Integerzahlen in Zweierkomplementdarstellung zu einer negativen Zahl. In Java ergibt z.B. `Integer.MAX_VALUE + 1 = -2147483648`, ohne Fehlermeldung. Integer Division durch 0 ergibt allerdings eine Fehlermeldung.

- Integerdivisionen können zu, aus mathematischer Sicht, falschen Ergebnissen führen. Z.B. ist in Java $5/2 = 2$ und $-5/2 = -2$. Da Brüche als Integer nicht darstellbar sind, werden die Bruchanteile einfach weggelassen.
- Floating Point Zahlen haben spezielle Bitkombinationen, die z.B. Unendlichkeit darstellen können. Division durch 0 muss dann nicht zu einem Fehler führen, sondern zu der Zahl `Infinity`.

2.2 Shifts

In Programmen tauchen sie seltener auf, aber auf Maschinenebene sind sie sehr wichtig: Shift-Operationen, mit denen man Maschinenwörter nach links oder rechts verschieben kann.

Linksshift ohne Rotation:
Die Shift-Operation verschiebt ein Wort um eine oder mehrere Stellen nach links. Am rechten Rand werden entsprechend viele Nullen eingeführt, und am linken Rand fallen die Bits einfach heraus.

Beispiele mit Wortgröße 8 Bits:

vorher	Linksshifts	nachher
00001111	1	00011110
00001111	3	01111000
10101010	1	01010100

Beobachtungen:
Ein Linksshift um 1 Bit entspricht der Multiplikation mit 2.
Ein Linksshift um n Bits entspricht der Multiplikation mit 2^n.
Ein Linksshift von signed Integern kann das führende Bit, und damit das Vorzeichen, ändern.

In Programmiersprachen wie Java gibt es den Operator $<<$ für einen Linksshift. Z.B. ist $1 << 2 = 4$ (die 1 am Ende wird um 2 Stellen nach links geschoben. Das rechte Ende ist dann 100.)

Dies kann man nutzen, um bestimmte Bitmuster zu erzeugen. $(1 << 8) - 1 = 255$ bei 32 Bit Wortlänge erzeugt das Bitmuster $\underbrace{00000000000000000000000}_{24}\underbrace{11111111}_{8}$.

Mit dem Komplementoperator $\sim$ kann man das Bitmuster herumdrehen. $\sim ((1 << 8) - 1) = -256$ erzeugt das komplementäre Bitmuster $\underbrace{11111111111111111111111}_{24}\underbrace{00000000}_{8}$.

Linksshift mit Rotation:
Die Shift-Operation verschiebt ein Wort ebenfalls um eine oder mehrere Stellen nach links. Allerdings werden die Bits, die am linken Rand herausfallen rechts wieder eingefügt.

Beispiele für Linksshift mit Rotation bei Wortgröße 8 Bits:

vorher	Linksshifts	nachher
00001111	1	00011110
11001111	3	01111110
10101010	1	01010101

In Java muss man für Linksshift mit Rotation eine Methode aufrufen:
Integer.rotateLeft($3 << 30$,2) = 3.

$3 << 30$ ergibt das Bitmuster mit 2 führenden Einsen. Schiebt man diese um zwei Stellen nach links, mit Rotation, dann kommen die beiden Einsen nach ganz rechts, daher die 3 als Ergebnis.

Logischer Rechtsshift ohne Rotation:
Dieser entspricht genau dem Linksshift ohne Rotation. Statt nach links wird nach rechts geschoben. Links wird mit Nullen aufgefüllt.

Beispiele für logischer Rechtsshift ohne Rotation mit Wortgröße 8 Bits:

vorher	Rechtsshifts	nachher
00001111	1	00000111
11001111	3	00011001

Logischer Rechtsshift ohne Rotation entspricht der positiven Integerdivision durch 2.
Wird um n Bits verschoben, dann entspricht es der positiven Integerdivision durch 2^n.

In Java ist es der Operator $>>>$. Bsp.: $4 >>> 2 = 1$, und $5 >>> 2 = 1$.
Vorsicht: die Vorzeichen werden nicht erhalten. $-8 >>> 2 = 1073741822$.

Arithmetischer Rechtsshift ohne Rotation:
Diese Operation verschiebt ebenfalls nach rechts, erhält aber das Vorzeichen. D.h. wenn das Vorzeichenbit 0 war, dann wird links mit Nullen aufgefüllt. Wenn das Vorzeichenbit 1 war, dann wird links mit Einsen aufgefüllt.

vorher	Rechtsshifts	nachher
00001111	1	00000111
11001111	3	11111001

Arithmetischer Rechtsshift ohne Rotation entspricht der Integerdivision durch 2.
Wird um n Bits verschoben, dann entspricht es der Integerdivision durch 2^n.
Dabei werden die Vorzeichen erhalten.

In Java ist es der Operator $>>$. Bsp.: $8 >> 2 = 2$ und $-8 >> 2 = -2$.

Rechtsshift mit Rotation:
Dies entspricht dem Linksshift mit Rotation. Nur werden die Bits, die rechts herausfallen wieder links eingefügt.

vorher	Rechtshifts	nachher
00001111	1	10000111
11000001	3	00111000

In Java kann man hat man z.B.: Integer.rotateRight(1,1) = -2147483648. Wenn die 1 rechts herausfällt und links wieder eingefügt wird, erhält man die negative Zahl bestehend aus 1, gefolgt von 31 Nullen.

2.3 Boolesche Anweisungen

Man kennt die Booleschen Operationen `nicht, und, oder, exklusives oder (xor)` usw. auf Wahrheitswerten oder einzelnen Bits. In Maschinensprache und Programmiersprachen werden diese Operationen auf ganze Bitfolgen *bitweise* angewendet.

Beispiele (für 8-Bit Wörter):

	0111 1010
und	1001 1000
	0001 1000

	0111 1010
oder	1001 1000
	1111 1010

	0111 1010
xor	1001 1000
	1110 0010

nicht	0111 1010
	1000 0101

In Java werden dafür folgende Symbole benutzt:

und	&
oder	|
xor	^
nicht	$\sim$

Es ist z.B. $1|2 = 3$, aber auch $1|3 = 3$.

Maskierung:
Eine wichtige Anwendung der Booleschen Operationen ist die *Maskierungstechnik*. Man möchte z.B. aus einer Integerzahl i die letzten 8 Bits extrahieren.
Das geht ganz einfach mit dem Ausdruck `i & ((1 << 8)-1)`.
`(1 << 8)-1` ist die *Maske* bestehend aus 24 Nullen gefolgt von 8 Einsen. Mit dem &-Operator werden die erste 24 Bits von i gleich 0, und die letzten 8 Bits bleiben wie sie sind.

Möchte man stattdessen die ersten 24 Bits extrahieren, dann geht das mit `i & ~((1 << 8)-1)`. Der Negationsoperator $\sim$ dreht die Bits der ersten Maske herum, so dass man 24 Einsen gefolgt von 8 Nullen als neue Maske erhält.

Möchte man diese extrahierten Bits noch nach rechts schieben, dann geht das mit
`(i & ~((1 << 8)-1))>>>8`.

2.4 Sprunganweisungen

Jedes nicht triviale Programm braucht *Sprunganweisungen*, z.B. um vom Ende einer Schleife wieder an den Anfang zu springen. Auf Maschinenebene unterscheidet man zwischen *unbedingten Sprunganweisungen* und *bedingten Sprunganweisungen*.

Unbedingte Sprunganweisungen sind einfach Befehle, um an eine bestimmte Adresse eines Programms zu springen und von dort das Programm weiterlaufen zu lassen.

Bedingte Sprunganweisungen testen bestimmte Bedingungen, z.B. ob der Wert in einem bestimmten Register gleich 0 ist, oder nicht. Abhängig davon führen sie einen Sprung aus, oder nicht.

2.5 Weitere Anweisungstypen

In vielen Prozessoren gibt es noch weitere Anweisungstypen:

Speicheranweisungen:
um Daten aus dem Arbeitsspeicher zu laden oder in den Arbeitsspeicher zu schreiben.

IO-Anweisungen:
steuern den Datenaustausch mit peripheren Geräten, z.B. Tastatur oder Graphikkarte.

Speicherreferenzanweisungen:
kombinieren Speicheranweisungen mit Operationen,
z.B. addiere Inhalt von Adresse X zu Register Y.

Interne Prozessoranweisungen:
z.B. die Halt-Operation, um den Prozessor zu stoppen.

3 Adressierung

Jede Speicherzelle im Arbeitsspeicher hat eine Adresse, angefangen von 0 bis zur Nummer der letzten verbauten Speicherzelle. In den allerersten Prozessoren wurden Programme so geschrieben, dass genau diese *direkten Adressen* benutzt wurden, z.B. `springe an Adresse 5`. Das hat allerdings zwei gravierende Probleme:

- Wenn das Programm geändert werden musste, und weitere Programmschritte dazwischen geschoben werden mussten, dann war es u.U. nötig, den Sprungbefehl an eine neue Adresse springen zu lassen. Dann mussten alle Adressen im ganzen Programm angepasst werden. Das ist extrem aufwendig und fehleranfällig.
- Das konnte nur funktionieren, wenn das Programm an exakt die vorgesehene Stelle geladen wird. Sobald man aber mehrere Programme gleichzeitig laden will, wie es in heutigen Computern üblich ist, kann man nicht mehr garantieren, dass ein bestimmtes Programm exakt an eine bestimmte Stelle geladen wird.

Aus diesen beiden Gründen benutzt man die Adressen nicht mehr direkt, sondern indirekt. Dafür gibt es verschiedene Möglichkeiten:

Indizierte Adressierung:
Man lädt eine Basisadresse in ein bestimmtes Register, das *Indexregister*, und berechnet die tatsächliche Adresse, indem man die Basisadresse zu der angegebenen Adresse hinzu addiert.

Jetzt kann man z.B. ein Programm an eine beliebige Adresse im Arbeitsspeicher laden. Die erste Adresse des Programms wird in das Indexregister geladen, angenommen es ist die Adresse 100. Wenn im Programm steht, `springe an Adresse 5`, dann ist die tatsächliche Adresse für die Sprung $100 + 5 = 105$. Wird das Programm an die Stelle 200 geladen, springt der Befehl an die Stelle 205.

Man kann die Technik auch für Arrayoperationen benutzen. Man speichert die Adresse des ersten Arrayelements in das Indexregister, und kann dann die Arraylemente einfach von 0 an durchzählen. Egal wo das Array tatsächlich im Speicher liegt, man kommt immer an das richtige Arrayelement.

Indirekte Adressierung:
Hierbei steht in einer Speicherzelle nicht Daten, sondern wiederum eine Adressen, und diese soll verwendet werden. Angenommen, an Speicherzelle 5 steht die 7, an Speicherzelle 7 steht die 100. Ein Befehl `lade Inhalte von Zelle 5, indirekt` würde zunächst auf die Zelle 5 zugreifen, dort die 7 finden, und dann aus der Zelle 7 die 100 laden.

Das kann man sogar noch eine oder mehrere Stufen weiter treiben, indirekt indirekt. Wenn im obigen Beispiel an der Stelle 100 die 200 steht, dann würde der Befehl `lade Inhalte von Zelle 5, indirekt,indirekt` die 200 laden.

Indiziert indirekt: kombiniert die beiden Möglichkeiten.
Ein Befehl `lade Inhalte von Zelle 5, indiziert, indirekt` würde zunächst die 5 zum Inhalt des Indexregisters hinzuzählen, und dann dort die eigentliche Adresse auslesen.

Immediate: braucht man manchmal wenn dort, wo normalerweise Adressen stehen, jetzt doch Daten stehen. Ein Beispielbefehl wäre: `erhöhe den Inhalt des Akkumulators um 1`. Jetzt ist die 1 nicht die Adresse der Speicherzelle 1, sondern einfach die Zahl 1.

Konkrete Prozessoren haben u.U. noch weitere Adressierungsmöglichkeiten, die die Möglichkeiten der Memory Management Unit mit einbeziehen, z.B. *Seiteandressierung*. Diese wird bei der virtuellen Speicherverwaltung behandelt.

4 Struktur der Maschinenanweisungen

Jeder Prozessor hat eine ganz genau festgelegte Menge von Maschinenanweisungen. Diese werden durchnummeriert, von Anweisung 0 bis zur maximalen Anweisungsnummer. Die Nummern heißen *Funktionscode* oder kürzer *Opcode*. Die maximale Anweisungsnummer ergibt sich zum einen aus der Architektur der Hardware, und zum anderen aus der Anzahl von Bits, die man für sie vorsieht. Sieht

man z.B. 8 Bits für den Funktionscode vor, dann kann man maximal 256 Maschinenanweisungen haben, von Nummer 0 bis Nummer 255. So ist das z.B. bei der Java Virtual Machine,

Jede Maschinenanweisung braucht neben dem Funktionscode natürlich noch Argumente, sog. *Operanden*, z.B. addiere Inhalt von X zu Register Y. Deshalb ist die Struktur einer Maschinenanweisung immer

Funktionscode	Argument 1	...	Argument n

Bei einer Wortlänge von 32 Bits, und 8 Bits für den Funktionscode hat man noch 24 Bits übrig, um die Argumente unterzubringen. Falls das nicht reicht, hat man auch schon mal ein oder zwei Worte dazu genommen, so dass sich eine Maschinenanweisung über 2 oder 3 Wörter erstreckt. Dies ist bei neueren Architekturen mit 64 Bit Wortlänge aber kaum noch nötig.

5 Eine einfache Maschine

Die Maschinensprache von konkreten Prozessoren sind sehr umfangreich, oft mit hunderten von Anweisungen. Um die Beschreibung einfacher zu machen, und doch die Prinzipien deutlich herauszuarbeiten, betrachten wir eine sehr vereinfachte Maschine mit nur 16 Anweisungen.

Die Wortlänge ist 16 Bits. Davon werden 4 Bits für den Funktionscode gebraucht. Ein Bit steuert die indirekte Adressierung und ein Bit steuert die indizierte Adressierung. Es bleiben dann noch 10 Bits für die Operanden:

Opcode						Operanden									
				*	+										

* = Indirekt Bit
+ = Index Bit

Der Instruktionsvorrat, d.h. die Maschinensprache, ist:

Opcode	Mnemonic	Operation
0000	LDA	LoaD Akkumulator mit dem Inhalt der angegebenen Adresse
0001	STA	STore Inhalt des Akkumulators in die angegebene Addresse
0010	LDN	LoaD Number in den Akkumulator
0011	ADD	ADD Inhalt der angegebenen Adresse zum Inhalt des Akkumulators
0100	SUB	SUBtract Inhalt der angegebenen Adresse vom Inhalt des Akkumulators
0101	ADN	ADd angegebene Number zum Inhalt des Akkumulators
0110	SUN	SUbtract angegebene Number vom Inhalt des Akkumulators
0111	AND	AND mit Inhalt der angegebenen Adresse und dem Akkumulatorinhalt
1000	OR	OR mit Inhalt der angegebenen Adresse und dem Akkumulatorinhalt
1001	JAZ	Springe an angegbene Adresse falls der Akkumulator 0 ist (Jump on Zero)
1010	JPU	Springe an angegbene Adresse (JumP Unconditionally)
1011	JAG	Springe an angegebene Adresse falls Akkumulator > 0 ist (Jump Greater)
1100	JAL	Springe an angegebene Adresse falls Akkumulator < 0 ist (Jump Less)
1101	JAN	Springe an angegebene Adresse falls Akkumulator nicht 0 ist
1110	IO	Gebe den Inhalt des Akkumulators aus
1111	STOP	STOP den Prozessor

Die Mnemonics helfen, sich an die Befehle zu erinnern. Sie werden auch in Assemblersprachen benutzt. Das sind im Prinzip Maschinensprachen in für den Menschen lesbarer Form.

Um diese Maschinensprache in Aktion zu sehen, betrachten wir eine einfache Maschine. Sie hat folgende Komponenten:

- Es gibt den Programmzähler (PC, oder Sequence Control Register, SCR)
- Es gibt das Current Instruction Register (CIR)
- Es gibt einen Akkumulator (ACC)
- Es gibt 18 weitere Register mit den Nummern $0 \ldots 17$.
- Register 0 kann als Indexregister benutzt werden.

Die Maschine startet immer bei Adresse 1, d.h. der erste Maschinenbefehl muss in Adresse 1 stehen.

Programm 1: Als erstes betrachten wir ein ganz einfaches Programm:

ADD 3 Addiere den Inhalt von Zelle 3 zum Akkumulator (direkte nicht indizierte Adressierung)
STOP

Angenommen im Akkumulator steht die 10.

Die Speicherbelegung sieht zu Beginn so aus:

Zelle	Daten (binär)	Daten (dez.)	Mnemonic
SCR	0000 00 0000000001	1	
CIR	0000 00 0000000000	0	
ACC	0000 00 0000001010	10	
0	0000 00 0000000000	0	
1	0110 00 0000000011		ADD 3
2	1111 00 0000000000		STOP
3	0000 00 0000000100	4	

Als erstes wird nun der Befehl in der Speicherzelle, die das SCR anzeigt geladen (fetch).

Zelle	Daten (binär)	Daten (dez.)	Mnemonic
SCR	0000 00 0000000001	1	
CIR	0110 00 0000000101		ADD 5
ACC	0000 00 0000001010	10	
0	0000 00 0000000000	0	
1	0110 00 0000000011		ADD 3
2	1111 00 0000000000		STOP
3	0000 00 0000000100	4	

Jetzt muss der Befehl ausgeführt werden (execute). Der Programmzähler wird 1 weitergezählt.

Zelle	Daten (binär)	Daten (dez.)	Mnemonic
SCR	0000 00 00000000010	2	
CIR	0110 00 0000000101		ADD 5
ACC	0000 00 0000001110	14	
0	0000 00 0000000000	0	
1	0110 00 0000000011		ADD 3
2	1111 00 0000000000		STOP
3	0000 00 0000000100	4	

Das SCR enthält jetzt die 2, also wird der Inhalt von 2 in das CIR geladen.

Zelle	Daten (binär)	Daten (dez.)	Mnemonic
SCR	0000 00 0000000010	2	
CIR	1111 00 0000000000		STOP
ACC	0000 00 0000001110	14	
0	0000 00 0000000000	0	
1	0110 00 0000000011		ADD 3
2	1111 00 0000000000		STOP
3	0000 00 0000000100	4	

Im Akkumulator steht jetzt das Ergebnis.
Die Ausführung des STOP-Befehls hält dann die Maschine an.

Programm Multiplikation: Das nächste Programm multipliziert den Inhalt von Zelle 2 mit dem Inhalt von Zelle 3. Wenn es keinen expliziten Multiplikationsbefehl gibt, muss man multiplizieren durch wiederholtes addieren. Als Beispiel multiplizieren wir $12 * 3$. Die 12 steht in Zelle 2, die 3 in Zelle 3. Da $12 * 3 = 12 + 12 + 12$, müssen wir zweimal addieren.

Das Maschinenprogramm sieht so aus:

Zelle	Daten (binär)	Daten (dez.)	Mnemonic
SCR	0000 00 0000000001	1	
CIR	0000 00 0000000000		
ACC	0000 00 0000000000	0	
0	0000 00 0000000000	0	
1	1010 00 0000000100		JPU 4
2	0000 00 0000001100	12	
3	0000 00 0000000011	3	
4	0000 00 0000000010		LDA 2
5	0001 00 0000000000		STA 0
6	0000 00 0000000011		LDA 3
7	0110 00 0000000001		SUN 1
8	1001 00 0000001110		JAZ 14
9	0001 00 0000000011		STA 3
10	0000 00 0000000000		LDA 0
11	0011 00 0000000010		ADD 2
12	0001 00 0000000000		STA 0
13	1010 00 0000000110		JPU 6
14	0000 00 0000000000		LDA 0
15	1110 00 0000000000		IO
16	1111 00 0000000000		STOP

Das Programm funktioniert folgendermaßen:

- Da die Maschine immer bei Zelle 1 anfängt, die Daten aber in Zelle 2 und 3 stehen, müssen diese Zellen zunächst übersprungen werden (JPU 4). In Zelle 4 fängt das eigentliche Programm an.
- Die Zelle 0, die sonst als Indexregister benutzt wird, wird in diesem Programm als Zwischenspeicher benutzt.
- Zunächst wir die Zelle 2 in den Akkumulator geladen (LDA 2) und von da aus in Zelle 0 geschrieben (STA 0). Jetzt steht die erste 12 im Zwischenspeicher.
- Jetzt wird die Zelle 3 (die Zahl 3) in den Akkumulator geladen (LDA 3), und um 1 verringert (SUN 1). Dies zählt die Anzahl der Additionen.
- Falls der Akkumulator jetzt 0 geworden ist, wird zur Ausgabe gesprungen (JAZ 14). Falls nicht, wird der Zähler wieder in Zelle 3 zurück gespeichert. Im ersten Durchgang steht dann dort die 2.

- Jetzt wird der Zwischenspeicher wieder in den Akkumulator geladen (LDA 0), der Inhalt von 2 (die 12) dazu addiert (ADD 2) und wieder in den Zwischenspeicher zurück kopiert (STA 0). Nach dem ersten Durchgang steht dann dort die 24.
- Nun wird wieder an den Anfang der Schleife zurück gesprungen (JPU 6), wo der Zähler geladen, dekrementiert und getestet werden soll.
- Ist der Zähler auf 0, dann wird zu Zelle 14 gesprungen, dort der Zwischenspeicher geladen (LDA 0) und ausgegeben (IO). Dann ist Schluss (STOP)

Der Leser mag das mal durchspielen.

Das Beispiel verdeutlicht auch, dass in der von Neumann-Architektur Programm und Daten völlig gleichberechtigt im Speicher stehen können. Durch entsprechende STA-Befehle könnte das Programm sich sogar selbst modifizieren, indem es in die Zellen schreibt, in der der Maschinencode steht.

In modernen Betriebssystemen trennt man allerdings gerade aus diesem Grund wieder die Bereiche, wo Programm und Daten stehen. Es soll verhindert werden, dass Programme sich selbst überschreiben, und damit meist zerstören. Fast immer passiert das unbeabsichtigt wegen Programmierfehlern. Durch die Trennung und entsprechende Sicherheitsüberprüfungen kann man das verhindern.

Stackarchitektur: Die oben beschriebene Maschine ist ein extrem vereinfachtes Beispiel für einen weit verbreiteten Typ von Maschinensprachen, wie sie in vielen Prozessoren realisiert ist. Es gibt aber noch eine ganz andere Architektur, die z.B. in der Java Virtual Maschine realisiert ist, die *Stackarchitektur*.

Die Datenhaltung geschieht dabei nicht in Registern, sondern auf einem Stack. Um z.B. zwei Zahlen zu addieren, werden diese zunächst auf das Stack „gepushed“. Dann wird der Addierbefehl aufgerufen. Der Addierbefehl weiß, wo er die Zahlen findet, die er addieren soll, nämlich auf dem Stack. Er weiß auch, wo er das Ergebnis hinschreiben soll, nämlich wieder auf das Stack. Alle Kommunikation zwischen den Befehlen geht dann über das Stack. Jeder Befehl nimmt seine Argumente vom Stack, und schreibt sein Ergebnis wieder auf das Stack. Der eigentliche Maschinenbefehl braucht daher gar keine Argumente mehr. Es reicht der Opcode.

Im Maschinencode von Java ist 1 Byte für den Opcode vorgesehen. Das reicht für 256 verschiedene Maschinenbefehle. Die allermeisten von ihnen haben keine Argumente, sondern nur ein Byte Opcode. Daher heißt die Sprache auch *Bytecode*.

Mikroprogammierung: Ganz fortgeschrittene Prozessoren gehen noch eine Ebene tiefer und erlauben, die Maschinensprache direkt neu zu programmieren. Man ist dann nicht auf eine feste Maschinensprache angewiesen, sondern kann je nach Anwendungsgebiet geeignete Maschinenbefehle neu definieren. Mikroprogrammierung wird auch eingesetzt, um den Befehlssatz zu erweitern, wenn z.B. für bestimmte Anwendungen komplexe Befehle häufig gebraucht werden. Mikroprogrammierte Befehle können dann effizienter abgearbeitet werden als eine Folge von normalen Maschinenbefehlen. Dies geht aber über den Rahmen dieses Miniskripts hinaus.

6 Assemblerprogrammierung

Niemand wird wirklich direkt in Maschinensprache programmieren. Hardwarenahe Programmierung macht man stattdessen in einer etwas komfortableren Sprache, der *Assemblersprache*. Programme in dieser Sprache, sog. Assemblerprogramme, werden vom sog. Assembler dann in Maschinenprogramme übersetzt. In der Assemblersprache benutzt man für die Befehle statt der Bitkombinationen die oben schon angesprochenen Mnemonics. Anstelle von konkreten Daten und Adressen kann man Variablen benutzen, die dann vom Assembler in konkrete Zahlen übersetzt werden. Compiler von Hochsprachen wie C oder C++ übersetzen zunächst in Assemblersprache, und von da in Maschinensprache.

7 RISC vs. CISC

Im Lauf der Entwicklung von Prozessoren wurden die Maschinensprachen immer umfangreicher. Komplexe Maschinenbefehle erfordern natürlich auch eine komplexe und teure Hardware. Irgendwann kam die Frage auf, lohnt sich der Aufwand überhaupt, d.h. werden diese komplexen Befehle wirklich oft benutzt? Eine Analyse gängiger Programme ergab, dass viele komplexe Befehle nur relativ selten benutzt wurden. Es lohnt sich vielleicht, die komplexen Befehle herauszuwerfen, und damit die Hardware einfacher, billiger und schneller zu machen. Die komplexen Befehle kann man dann als kleine Programmstücke realisieren.

Die Firma Sun brachte daraufhin in den 1980ern den ersten Rechner mit sog. RISC-Prozessoren (Reduced Instruction Set Computer) heraus, und der war tatsächlich bedeutend schneller als die bisherigen CISC-Rechner (Complex Instruction Set Computer).

Inzwischen hat die Unterscheidung RISC/CISC aber an Bedeutung verloren. Die RISC-Prozessoren haben wieder mehr komplexe Befehle aufgenommen, und die CISC-Prozessorn haben auch teilweise RISC-Ideen realisiert.

Stichwortverzeichnis