Visuelle Maschine

Robert Stöhr

8. Februar 2021

Die Visuelle Maschine ist eine einfache Ausführung einer (abstrakten) *Registermaschine* mit erweitertem Befehlssatz bzw. einer *Random Access Machine* ohne indirekte Adressierung. Das Ziel ist es, die Arbeitsweise einer solchen Maschine möglichst visuell darzustellen. Gleichzeitig soll eine Brücke zur Computer-Architektur geschlagen werden, indem die Darstellung und Arbeitsweise einem 8-Bit-Computer ähnlich ist. Die Maschine läuft in jedem modernen Browser und bedarf daher keiner Installation: `https://lehrer.online/informatik/maschine`

1 Aufbau

Zunächst soll hier auf die Bestandteile einer Random Access Machine und der Umsetzung eingegangen werden. Die theoretischen Bestandteile sind:

Programm Eine Folge aus endlich vielen Befehlen (z. B. `LOAD 2`), die mit Nummer 1 beginnend durchnummeriert werden.

Befehlszähler Ein Speicher (Register) für eine Zahl mit Initialwert 1, der die Nummer des auszuführenden Befehls enthält.

Speicher Eine Menge aus unendlich vielen Speicherzellen (Registern), von denen jede eine beliebig große natürliche Zahl speichern kann. Die Speicher-Register werden mit Nummer 1 beginnend durchnummeriert (kurz R[1], R[2] usw. geschrieben).

Akkumulator Ein spezielles Register (R[0]), das einerseits als Operand dient und andererseits auch das Ergebnis der Berechnungen speichert.

Die Komponente Programm ist als (altmodische) Lochkarte umgesetzt, da dies sehr anschaulich ist und gleichzeitig die Beschränktheit der Befehlssyntax unterstreicht. Jeder Befehl wird der Reihe nach aufgeführt und nummeriert. Der Befehlszähler zeigt zu Beginn auf den ersten Befehl

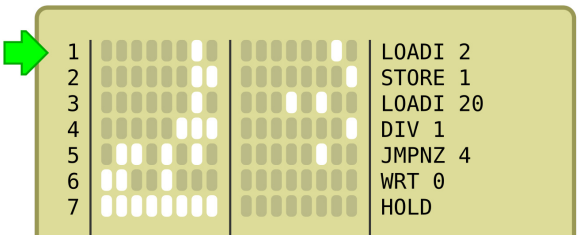


Abbildung 1: Programmkarte

und ist als grüner Pfeil dargestellt. Dies verdeutlicht auch, warum man statt „Befehlszähler“ gelegentlich auch den Begriff „Befehlszeiger“ (Instruction Pointer) verwendet. In der ersten Spalte ist die fortlaufende Nummerierung des Programmcodes. In der zweiten Spalte ist der Befehlscode als binäre Zahl eingestanzt. Die Maschine kann also maximal 256 Befehle unterscheiden. Die dritte Spalte enthält den Befehlsoperanden als binäre Stanzung. In der letzten Spalte steht der Programmcode in Assemblersprache. Der imaginäre Lochkartenleser liest nur die beiden Spalten mit den Stanzungen aus. Damit ist klar, dass jeder Befehl für die Maschine folgende Syntax hat:

```
OPERATION [byte]
```

Je nach Befehl wird die nachfolgende Zahl entweder als Adresse oder Wert interpretiert. Wird im Assembler keine Zahl angegeben, so bleibt die dritte Spalte zwar ohne Stanzung, stellt aber zwangsläufig für den Kartenleser den Wert Null dar, da es für den Kartenleser nicht ersichtlich ist, ob hier keine Stanzung erfolgte, oder bewusst der Wert 0 „gestanzt“ wurde. Die beiden Assemblerzeilen

führen also zum selben Ergebnis bei der Stanzung und damit auch bei der Ausführung. Analog gilt dies für den `NOOP` Befehl und einer Leerzeile bzw. einem unbekannten Befehl. Spezielle Befehle wie `HOLD` ignorieren den angegebenen Wert. Auf die

Karte passen insgesamt 32 Befehle. Längere Programme sind zwar möglich, werden aber graphisch nicht richtig dargestellt.

Die Maschine hat einen Speicher mit 16 Registern, wobei jedes Register maximal ein Byte (also Zahlen von 0 bis 255) speichern kann. Die einzelnen Bits des Speichers werden als kleine Lämpchen dargestellt. Jeweils acht Lämpchen ergeben ein Register. Die Register sind in Leserichtung angeordnet. Ein adressiertes Register leuchtet heller als die übrigen. In Abb. 2 ist R[5] adressiert, in dem aktuell der Wert $00010101_2 = 21_{10}$ gespeichert ist. Das spezielle Register 0 besteht aus roten Lämpchen und dient als Akkumulator. Entgegen dem theoretisch unendlich

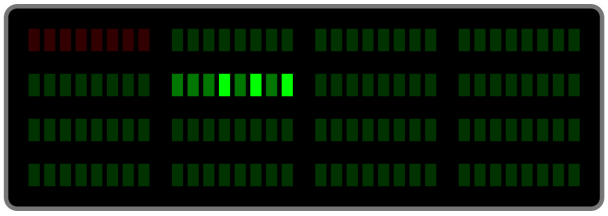


Abbildung 2: Register

großen Speicher und den Registern mit beliebig großen natürlichen Zahlen, ist die Visuelle Maschine mit 16 Registern zu je 1 Byte zwar sehr beschränkt, reicht aber aus, um die grundsätzlichen Mechanismen zu veranschaulichen. Größere „Software-Projekte“ sind bei dieser Maschine von ihrem Grundgedanken her sowieso fehl am Platz.

2 Ein- und Ausgabe

Die Grundidee hinter der Registermaschine ist ja, dass aus einer Eingabe durch die Abarbeitung des Programms am Ende eine gewünschte Ausgabe erfolgt. Die Random Access Machine erwartet die endliche Eingabe zu Beginn der Programmausführung in den Registern ab Nummer 1. In welchen Registern am Ende die erwartete

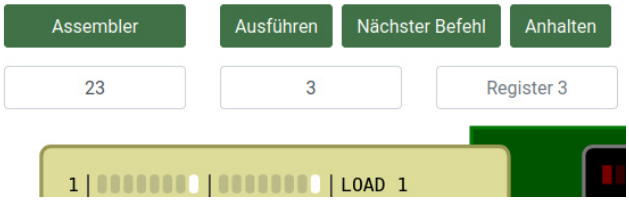


Abbildung 3: Eingabewerte

Ausgabe steht, ist dem Anwender überlassen. In der Visuellen Maschine kann die Eingabe in den vier Felder oberhalb der Maschine erfolgen. Es können nur die Register 1 bis 4 mit Eingabewerten belegt werden. Dies sollte für die meisten Übungsprogramme genügen. Natürlich ist es genauso möglich, die benötigten Werte am Anfang des Programms zu laden. Um die selbe Belegung zum Programmstart wie in Abb. 3 zu erhalten, kann folgendes geschrieben werden:

```
LOAD 23
STORE 1
LOAD 3
STORE 2
```

Die Eingabedaten vom Programm zu trennen, ist jedoch grundsätzlich sinnvoll, da ein Programm damit Eingangsdaten in bestimmten Registern erwartet und diese unabhängig vom Quellcode geändert werden können. Um ein Programm mit unterschiedlichen Eingabedaten laufen zu lassen (ohne das Programm neu assemblieren zu müssen), können die Eingabedaten geändert und anschließend der Reset-Knopf betätigt werden. Die Maschine wird dann entsprechend zurückgesetzt.

Am Ende sollte die Ausgabe in vorher festgelegten Registers stehen. Um das Ergebnis dann komfortabler zu sehen, wurde die Maschine mit einem Ausgabe-Display ausgestattet. Dieses kann

Abbildung 4: Ausgabe

15 Stellen anzeigen. Um auf dem Display den Wert aus R[n] auszugeben, kann z. B. der Befehl `WRT n` verwendet werden. Die Werte werden der Reihe nach ausgegeben.

3 Befehle

Es folgt die Liste der Befehle, die von der Maschine akzeptiert werden. Ist bei einem Befehl nichts anderes ausgesagt, so erhöht dieser nach Beendigung der Arbeit den Befehlszähler um eins, damit anschließend der nachfolgende Befehl auf der Karte ausgeführt wird.

3.1 Programm

`NOOP` Macht nichts (no operation).

`HOLD` Beendet die Ausführung des Programms.

3.2 Laden und Speichern

`LOAD n` Lädt den Wert aus dem Register n in den Akkumulator.

`LOADI n` Lädt den Wert n in den Akkumulator.

`STORE n` Speichert den Wert des Akkumulators im Register n.

3.3 Berechnen

`ADD n` Addiert den Wert in Register n zum Akkumulator.

`SUB n` Subtrahiert den Wert im Register n vom Akkumulator.

`MUL n` Multipliziert den Wert im Akkumulator mit dem Wert von Register n.

`DIV n` Dividiert den Wert des Akkumulator durch den Wert von Register n.

`MOD n` Speichert im Akkumulator den Rest, der bei der Division von Akkumulator durch Register n entsteht.

`CMP n` Vergleicht den Wert im Akkumulator mit dem Wert in Register n. Der Akkumulator wird nicht verändert. Es werden der Rechnung R[0] – R[n] entsprechend das Negativ- und Nullbit gesetzt.

Die folgenden Befehle sind die Pendants zu den vorherigen Befehlen, bei denen statt dem Register n direkt der angegebene Wert n verwendet wird: `ADDI SUBI MULI DIVI MODI CMPI`.

3.4 Bitmanipulation

`AND n` Bitweise Und-Verknüpfung des Akkumulators mit dem Register n.

`OR n` Bitweise Oder-Verknüpfung des Akkumulators mit dem Register n.

`XOR n` Bitweise Exklusiv-Oder-Verknüpfung des Akkumulators mit dem Register n.

`SHL n` Schibt die Bitfolge im Akkumulator um die im Register n angegebene Anzahl an Stellen nach links (shift left).

`SHR n` Schibt die Bitfolge im Akkumulator um die im Register n angegebene Anzahl an Stellen nach rechts (shift right).

`NOT` Bitweise Invertierung des Akkumulators.

Die folgenden Befehle sind die Pendants zu den vorherigen Befehlen, bei denen statt dem Register n direkt der angegebene Wert n verwendet wird: `ANDI ORI XORI SHLI SHRI`.

3.5 Springen

`JMP n` Setzt den Befehlszähler auf den Wert n.

`JMPP n` Setzt den Befehlszähler auf den Wert n, wenn weder das Nullbit noch das Negativbit gesetzt ist (jump positiv).

`JMPNN n` Setzt den Befehlszähler auf den Wert n, wenn das Negativbit nicht gesetzt ist (jump not negativ).

`JMPN n` Setzt den Befehlszähler auf den Wert n, wenn das Negativbit gesetzt ist.

`JMPNP n` Setzt den Befehlszähler auf den Wert n, wenn das Negativbit oder das Nullbit gesetzt ist (jump not positiv).

`JMPZ n` Setzt den Befehlszähler auf den Wert n, wenn das Nullbit gesetzt ist (jump zero).

`JMPNZ n` Setzt den Befehlszähler auf den Wert n, wenn das Nullbit nicht gesetzt ist (not zero).

`JMPC n` Setzt den Befehlszähler auf den Wert n, wenn das Übertragungsbit gesetzt ist (jump carry).

`JMPO n` Setzt den Befehlszähler auf den Wert n, wenn das Überlaufbit gesetzt ist (jump overflow).

3.6 Ausgeben

`WRT n` Schreibt den Wert aus Register n in die Ausgabe (write).

`WRTN n` Schreibt den Wert aus Register n im Zweierkomplement in die Ausgabe (write number).

`WRTB n` Schreibt den Wert aus Register n als Wahrheitswert in die Ausgabe (write boolean). Ist das Nullbit gesetzt, ist das Ergebnis `FALSCH`, ansonsten `WAHR`.

`WRTC n` Schreibt das Zeichen mit dem Code aus Register n in die Ausgabe (write char). Zum Beispiel hat der Buchstabe „A“ hat den Code 65 (siehe ASCII).

`CLR` Löscht den Inhalt der Ausgabe (clear).

4 Rechnerarchitektur

Die abstrakte Random Access Machine gibt ja nur vor, welche Wirkung die Befehle auf die Register haben (z. B. `LOADI` bewirkt eine Veränderung im Akkumulator und im Befehlszähler). Eine konkrete Maschine muss dies jedoch irgendwie technisch umsetzen. Die Visuelle Maschine stellt dabei eine Art primitiven 8-Bit-Computer dar, da die wesentliche Verarbeitungsbreite acht Bit beträgt.

Um den Prozessor der Visuellen Maschine ein wenig mit der realen Chip-Technologie zu vergleichen, kommt vielleicht der historisch interessante 4004-Chip von Intel in Frage. Der 4004 war Intels erster Mikroprozessor. Auf der Größe eines Fingernagels lieferte er soviel Rechenleistung wie der erste elektronische Computer, der noch einen ganzen Raum füllte. Das Design bestand

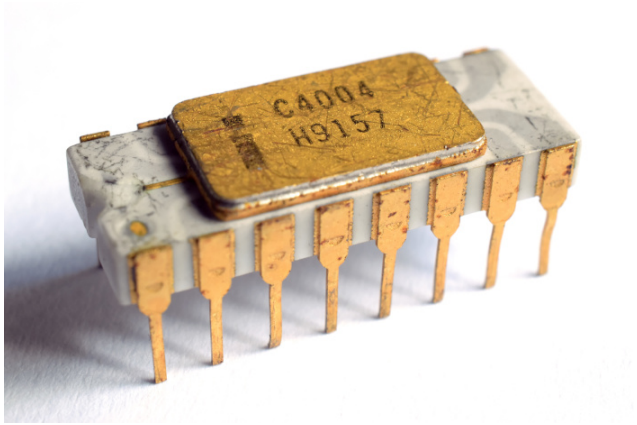


Abbildung 5: 4-Bit-Mikroprozessor Intel C4004
Thomas Nguyen, CC BY-SA 4.0
`https://creativecommons.org/licenses/by-sa/4.0`
via Wikimedia Commons

aus vier Chips: ein ROM-Chip (4001) für das Programm, ein RAM-Chip (4002) für die Daten, ein Schieberegister-Chip (4003) für die Ein- und Ausgabe sowie den 4004 als *central processing unit* (CPU). Es handelt sich hierbei um eine *Harvard-Architektur*, da Programm- und Datenspeicher getrennt sind.[1]

[1] Vgl. Intel Corporation, Die Geschichte des 4004, `https://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html` (03.02.2021).

Die Lochkarte der Visuellen Maschine entspricht dem ROM-Prinzip, da die Karte nur gelesen werden kann. Ein „richtiger“ Arbeitsspeicher (Speicher-Chip) fehlt jedoch. Den vorhandenen Speicher (16 Register als Akkumulator und Datenregister) muss man sich eher auf der CPU verortet vorstellen. Aus Gründen der Anschaulichkeit wurde das Register jedoch ausgelagert. Es gibt auch keinen *Stack* für Unterprogrammaufrufe. Da es nur 16 Register zum Ansteuern gibt, genügen dafür die vier Adress-Leitungen rechts oben am Prozessor. Die Verwendung der übrigen Pins und

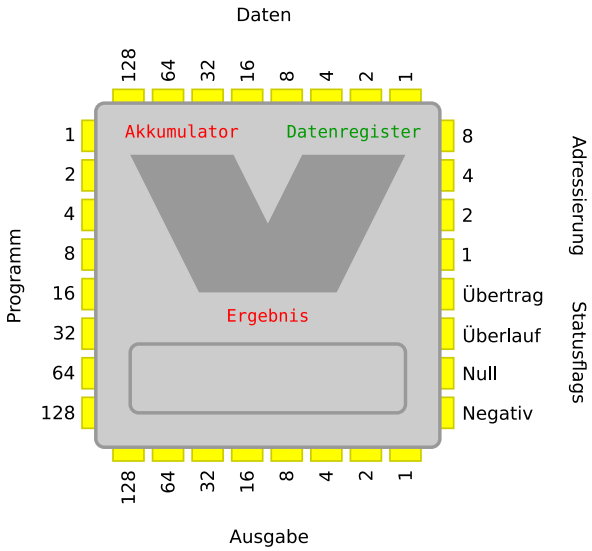


Abbildung 6: Virtueller Mikroprozessor

deren Wertigkeit können der Abb. 6 entnommen werden. Wird ein Byte zwischen zwei Komponenten ausgetauscht, so leuchten die entsprechenden Leitungen auf. Zur Vereinfachung werden Steuerbefehle nicht graphisch umgesetzt. Wird zum Beispiel dem Display das Byte $01000001_2 = 65_{10}$ übertragen, so fehlt dem Display noch die Information (Steuerbefehl), wie der Wert zu interpretieren ist, also ob die Zahl 65 oder der Buchstabe A dargestellt werden soll. In der Animation wird nur die Übertragung der Zahl 65 angezeigt. Man beachte, dass beim Wert 0 keine Leitungen aktiviert werden, diese also einen Takt lang stromlos bleiben.

Die sichtbaren Elemente auf der CPU sind die *arithmetisch-logische Einheit* (ALU) und der Befehlsdecoder. Zu Beginn des Befehlszyklus wird der Code, der an den linken Programm-Pins anliegt, ausgelesen und decodiert. Der gelesene Befehl steht anschließend im unteren Kasten auf der CPU. Die Register der ALU kann man mit dem Speicher gleichsetzen. Bei den Befehlen können

vier Meldungen (Flags) auftreten:

Übertrag Wird gesetzt, wenn durch eine Berechnung ein Übertrag in ein Bit über acht nötig wäre, also das Ergebnis größer als 255 ist:

		0	0	0	0	1	1	1	1
	×	0	0	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1	1

15
41
615

Überlauf Der Prozessor unterstützt im Zweierkomplement nur die Ganzzahl-Addition und -Subtraktion. Da die Subtraktion auf die Addition mit der Gegenzahl zurückführbar ist, genügt es, die Addition zu betrachten. Kommt es bei einer Rechnung zu einem Überschreiten des Wertebereiches von –128 bis 127 wird dieses Flag gesetzt.[2]

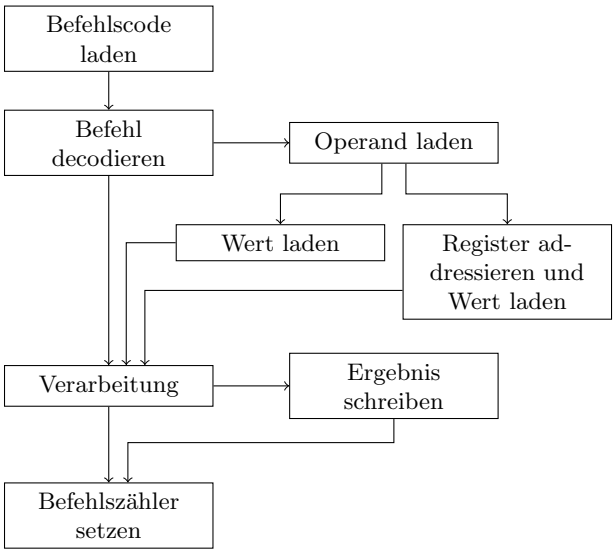
		0	0	1	1	1	0	0	1
	+	0	1	0	1	1	0	0	1
0		1	1						
		1	0	0	1	0	0	1	0

57
89
$146 \equiv -110$

Null Dieses Flag wird gesetzt, wenn durch einen Befehl der Wert Null geschrieben oder gelesen wird.

Negativ Ist das höchstwertige Bit gesetzt, so zeigt dieses gesetzte Flag an, dass der Wert als negative Zahl im Zweierkomplement interpretiert werden kann. Aber wird zum Beispiel $120 + 80 = 200$ gerechnet, wird das Negativbit gesetzt, kann aber in diesem Fall ignoriert werden.

Damit der Ablauf visuell besser erfasst werden kann, lässt sich in dem Eingabefeld rechts oben ein Wert eintragen, der die Länge der Pause in Millisekunden zwischen den Einzelschritten festlegt. Als Standard sind 500 ms vorgegeben. Die Pause wird jeweils am Ende eines Schrittes des Befehlszyklus' gesetzt:



5 Assembler

Um für die Visuelle Maschine Programme zu schreiben, gibt es einen sehr einfachen Assembler. Die Syntax ist

```
[LABEL:] BEFEHL [WERT|LABEL [KOMMENTAR]]
```

Ein Beispiel für ein sehr einfaches Programm in der Assemblersprache ist in Abb. 7 gegeben. Die

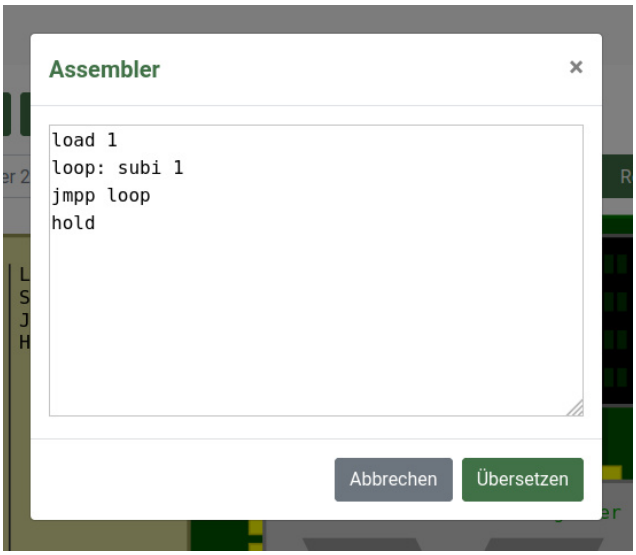


Abbildung 7: Assembler

Aufgabe des Assemblers ist es, den für den Menschen besser zu lesenden Programm-Quelltext in Assemblersprache in die Maschinensprache umzusetzen (assemblieren). Dabei ignoriert der Assembler die Groß- und Kleinschreibung. Erkennt der

[2] Entscheidend dafür sind der Übertrag in das Negativbit und der Übertrag aus dem Negativbit. Ein Überlauf liegt vor, wenn genau eines von beiden zutrifft. Das Negativbit ist das *most significant bit* (MSB).

Zwei positive Zahlen können maximal einen Übertrag in das MSB erzeugen (siehe Beispiel). In diesem Fall ist es ein Überlauf. Addiert man eine positive und negative Zahl, so bleibt das Ergebnis immer im Wertebereich. Kommt es dabei zu einem Übertrag in das MSB, so kommt es automatisch auch zu einem Übertrag aus dem MSB, da eine Zahl ja das Negativbit hat. Addiert man zwei negative Zahlen, so kommt es immer zu einem Übertrag aus dem MSB. Das Ergebnis liegt oberhalb $-129_{10} = 1\,0111\,1111_2$, wenn es auch zu einem Übertrag in das MSB kommt. Bleibt es bei dem Übertrag aus dem MSB, ist es ein Überlauf.

Assembler einen Befehl nicht, so wird kein Befehlscode in der Befehlsspalte der Lochkarte „gestanzt“. Der Prozessor wird diese Zeile als `noop` decodieren. Ist nach dem Befehl noch ein Token

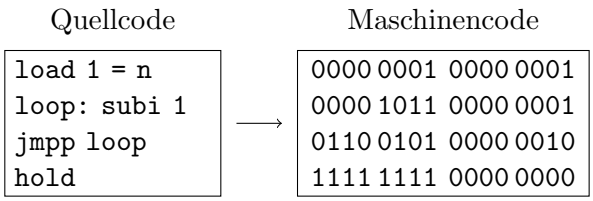


Abbildung 8: Assemblieren

gegeben, so wird versucht, dieses in eine Zahl umzuwandeln. Alles was danach kommt, wird vom Assembler ignoriert und kann daher für (idealerweise kurze) Kommentare verwendet werden. In Abb. 8 ist die Eingabe und Ausgabe beim Assemblieren exemplarisch dargestellt.

Dieser Assembler ist sehr nahe an der Maschinensprache. Ein sprachlich höherer Assembler könnte z. B. Befehle mit mehreren Operanden ermöglichen (siehe Abb. 9). Eine kurze An-

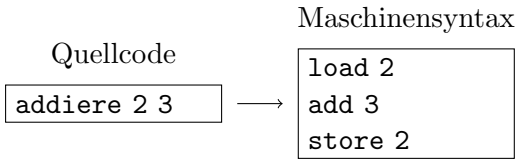


Abbildung 9: Höhere Assemblersprache

weisung wird dann in mehrere Maschinenbefehle umgewandelt. Eine andere Möglichkeit für komplexere Befehle wäre, auf der CPU mittels Interpreter (Hardwarelösung natürlich auch möglich) zusätzliche Befehle zu implementieren (vgl. RISC vs. CISC Konzept). Dann müsste allerdings die Beschränktheit der Lochkarte auf einen Operanden entsprechend geändert werden.

6 Übung

1. Einige stellen sich vielleicht schon länger die Frage, wofür so ein Akkumulator überhaupt gut sein soll? Neben den schaltungstechnischen Vorteilen ermöglicht die Verwendung eines Akkumulators die knappe Befehlssyntax der Maschine mit maximal einem Operanden. Man betrachte folgendes Programm in Pseudocode und dessen Umsetzung auf einer Maschine, die zwei Operanden ermöglicht:

```
                              Maschine mit
Pseudocode:                   zwei Operanden:
R[1] = 10                     loadi 1 10
R[2] = R[3]                   load  2 3
R[1] = R[1] + R[2]            add   1 2
```

Durch die Verwendung des Akkumulators (Zwischenspeicher) können die Maschinenbefehle auf einen Operanden reduziert werden, da der Akkumulator ggf. der notwendige zweite Operand ist. In der Sprache der Visuellen Maschine würde dies also lauten:

```
loadi 10
store 1
load  3
store 2
load  1
add   2
store 1
```

Man vergleiche die letzten beiden Code-Schnipsel und überlege, wie der Akkumulator im letzten Code zum Einsatz kommt!

2. Im Akkumulator soll das Ergebnis der Rechnung

$$\mathrm{R}[0] = 2 \times (3 + 2)$$

gespeichert werden. Man versuche dies zunächst im Taschenrechner zu rechnen. Der Taschenrechner hat mit dem [Ans]-Speicher einen vergleichbaren Akkumulator. Das vorherige Ergebnis muss nicht erneut eingetippt werden, wenn mit diesem weiter gerechnet werden soll. Ein Eintippen von links nach rechts

[2] [=] [×] [3] [=] [+] [2] [=]

führt aber natürlich zum falschen Ergebnis, da die Klammer nicht beachtet wird. Eine Umformung der Rechnung zu

$$\mathrm{R}[0] = (3 + 2) \times 2$$

führt aber auch ohne Beachtung der Klammer

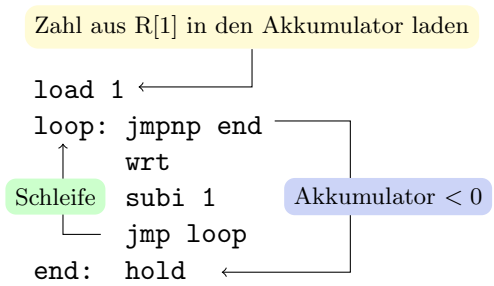
[3] [=] [+] [2] [=] [×] [2] [=]

zum richtigen Ergebnis. Führen Sie die Rechnungen selbst mit dem Taschenrechner aus! In der Visuellen Maschine lautet der entsprechende Code:

```
loadi 3
addi 2
muli 2
hold
```

Führen Sie den Code auf der Maschine aus und erweitern Sie anschließend den Code, indem Sie an geeigneter Stelle den Befehl `WRT` schreiben, um das Ergebnis im Display ausgeben! Warum ist es nicht nötig `WRT 0` zu schreiben?

3. Die in höheren Programmiersprachen zur Verfügung stehenden Kontrollstrukturen wie `for`, `while`, `if` etc. fehlen beim Assembler. Diese müssen von Hand durch entsprechende Jump-Befehle umgesetzt werden. Der folgende Code erwartet in `R[1]` eine positive Zahl, von der heruntergezählt werden soll:



Die Verwendung von `jmpnp` gegenüber `jmpz` hat den Vorteil, dass das Programm bei einer negativen Zahl in R[1] nicht in eine „Endlos-Schleife“ gerät. Warum kommt es dabei nicht wirklich zu einer endlosen Schleife?

Der obige Code gibt z. B. für R[1] = 5 die Folge `5 4 3 2 1` aus. Schreiben Sie ein Programm, dass die Zahlen in aufsteigender Reihenfolge ausgibt!

Ändern Sie das Programm dann so ab, dass statt Zahlen die Buchstaben des Alphabets ausgegeben werden! Zum Beispiel soll bei R[1] = 5 die Ausgabe `A B C D E` erfolgen!