
Bachelor-Arbeit am Institut für Informatik der Freien Universität Berlin

# Simulation Konrad Zuses Logistischer Maschine

Max Dieckmann

Matrikelnummer: 4518257

dieckmann@zedat.fu-berlin.de

Betreuer & Gutachter: Prof. Dr. Raúl Rojas

Berlin, 27. Juli 2016

**Zusammenfassung**

Zum 75. Jubiläum der „Z3“ von Konrad Zuse habe ich mich mit einer seiner weniger bekannten Erfindungen befasst und eine schematische Darstellung der „logistischen Maschine“ animiert.
Meine Simulation wurde in Python mit Hilfe der GUI-Bibliothek[1] „TkInter“ erstellt und imitiert das Original in so weit, dass man das Verhalten der wichtigen Bauteile nachvollziehen kann.
Das Programm wird im September 2016 im Rahmen des ***Heidelberg Laureate Forum***[1] ausgestellt und soll dem Zuschauer[2] die Funktionsweise der Maschine verständlich machen[3].

## Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

| | |
|---|---|
| Unterschrift | Datum |

# Inhaltsverzeichnis

# 1 Zusammenfassung

Konrad Zuse (*22. Juni 1910; †18. Dezember 1995)[3] war einer der Pioniere der Informatik.
Seine wohl größte Errungenschaft ist die Erfindung der „Z3“ (1941)[4], welches der erste funktionsfähige Digitalrechner war.

Die „Z3“ und ihr kommerzielles Nachfolgemodell „Z4“ (1945)[5] war auch in anderen Punkten ihrer Zeit voraus: Sie verfügte über Algorithmen, um von dezimal nach binär zu konvertieren (und umgekehrt), Quadratwurzeln zu berechnen und hatte IEEE-ähnliche Gleitkommazahl-Arithmetik[4]

In dieser Arbeit wollen wir uns aber mit einer anderen Maschine befassen: der „logistischen Maschine“. Dieses um 1944 erdachte Gerät war nie für die praktische Nutzung konzipiert, sondern vielmehr ein theoretisches Konstrukt, das alle Operationen der arithmetischen Maschinen in einer minimalen Architektur realisiert.
Die Idee ist vermutlich darauf zurück zu führen, dass die arithmetischen Maschinen komplexere Operationen (Division, Wurzel, Konvertierung etc.) durch Kombinationen der elementaren Operationen (Addition, Subtraktion, Shift[5]) simulieren.

In der „logistischen Maschine“ ist Zuse noch einen Schritt weiter gegangen und hat einen minimalen Computer erdacht. In diesem gibt es nur zwei 1-Bit große Register[6] und die beiden Operationen der Disjunktion und der Konjunktion.
Die Hoffnung war, dass man auf dieser Architektur bereits die vier arithmetischen Operationen implementieren kann.

[1] Ein *graphical user interface* (GUI) gibt dem Benutzer eines Programms die Möglichkeit, mit ihm zu interagieren. Im Gegensatz zum *command line interface* findet die Interaktion nicht durch Eingabe von Text, sondern durch die Manipulation graphischer Elemente statt. Eine Bibliothek bietet dem Programmierenden eine Reihe von Bausteinen, die auf die Lösung eines bestimmtes Problems zugeschnitten sind.

[2] Im Folgenden wird das generische Maskulinum verwendet. Alle gender seien dabei eingeschlossen.

[3] In diesem Sinne werde ich mich im schriftlichen Teil des Projekts darum bemühen, es auch für Laien so weit es geht verständlich zu gestalten. Daher habe ich ausgewählte oder eventuell missverständliche (Fach-) Begriffe in Fußnoten zusätzlich erläutert.

[4] Um auch rationale Zahlen im Computer behandeln zu können, teilt man die Darstellung der Zahl im Speicher in Mantisse und Exponent. Die Mantisse beschreibt die Genauigkeit, während der Exponent die Größenordnung repräsentiert. Seit 1985 gibt es den IEEE-Standard.

[5] Der Shift ist eine Operation, bei der alle Bits einer Zahl nach links bzw. rechts verschoben werden. Man fügt eine 0 oder 1 an ein Ende der Zahl hinzu, und alle anderen Bits „rutschen auf“. Auf der anderen Seite „fällt“ ein Bit aus der Zahl.

[6] Register sind Speicherbereiche direkt im Prozessor, in denen alle Berechnungen stattfinden

Die Vermutung liegt nahe, dass die hier angestellten Überlegungen Teile des Gedankengebäudes von Zuse sind, das schließlich unter dem Namen „Plankalkül“ die erste Programmiersprache der Welt werden sollte.

Am 11.05.2016 war es 75 Jahre her, dass Konrad Zuse den arithmetischen Computer „Z3“ gebaut hatte. Im Rahmen einer Präsentation zu diesem Thema wird auch die „logistische Maschine“ vorgestellt. Um deren Funktionsweise zu erklären, habe ich in dieser Arbeit eine schematische Darstellung der „logistischen Maschine“ animiert.

# 2 Die logistische Maschine

Die „logistische Maschine“ besteht aus einem Befehlsband, einem Speicher und einem Prozessor.

Da es sich um ein theoretisches Konstrukt handelt, kann man von einem beliebig langen Band und beliebig großen Speicher ausgehen. In meiner Simulation ist daher immer nur ein Teil des Bands / Speichers zu einem gegebenen Zeitpunkt sichtbar (siehe Abb. 6/7).

Der Prozessor hat zwei 1-Bit Register A und B und eine Flagge[7] Pr.

Der Befehlssatz der „logistischen Maschine“ lautet:

| | |
|---|---|
| $A \vee B$ | Disjunktion |
| $A \wedge B$ | Konjunktion |
| LOAD x | Lade den Inhalt der Speicherzelle x in ein Register. Ist $Pr = 0$, landet die Information im Register A und Pr wird auf 1 gesetzt. Andernfalls wird die Information nach Register B weiter geleitet (Pr bleibt gleich). |
| STORE x | Speichere den Inhalt von A in der Speicherzelle x und setze $Pr = 0$. Setze anschließend $A = 0$ und $B = 0$. |

Die Register können für den Zweck der logischen Operation auch negiert werden, sodass auch die Befehle $\neg A \wedge B$, $A \wedge \neg B$, $\neg A \wedge \neg B$ (analog für $\vee$) Teil des Befehlssatzes sind.
Das Ergebnis einer logischen Operation wird immer im Register A abgelegt.

Die Schaltung der „logistischen Maschine“ ist sehr simpel.
In Zuses Patent von 1944 ist der *opcode*[8] 8 Bit lang, von denen für die logischen Operationen nur 5 Bit benötigt werden. In diesem Zustand können bis zu 64 Speicher-Adressen angesprochen werden. Wenn mehr benötigt werden, muss der *opcode* entsprechend verlängert werden.

Mit einem 8-Bit großen Wort und den Bits $t_1 ... t_8$ ergibt sich folgende Codierung:

[7] Eine Flagge kann nur zwei Werte annehmen und ist stets Bestandteil einer Kontrollstruktur.
[8] Damit ein Befehl ausgeführt wird, muss die Schaltung der Maschine in den korrespondierenden Zustand gebracht werden. Der *opcode* beschreibt die Werte, die die entsprechenden Schalter annehmen müssen, damit die Maschine in besagten Zustand gelangt.

| $t_1$ | $t_2$ | |
|---|---|---|
| 0 | 0 | NOP (no operation) |
| 0 | 1 | logische Operation |
| 1 | 0 | LOAD |
| 1 | 1 | STORE |

Wenn $t_1 = 1$ handelt es sich um eine Operation auf dem Speicher und die restlichen Bits $t_3 ... t_8$ spezifizieren die Speicher-Adresse.

Wenn es sich um eine logische Operation handelt, bestimmt $t_3$ die Art der Operation. Bits $t_4$ und $t_5$ legen fest, ob die Register negiert werden.

| $t_3$ | |
|---|---|
| 0 | Konjunktion ($\vee$) |
| 1 | Disjunktion ($\wedge$) |

| $t_4/t_5$ | |
|---|---|
| 0 | $\neg A$ / $\neg B$ |
| 1 | A / B |

## 2.1 Der Speicher

Der Speicher besteht aus zahlreichen Relais[9], von denen jedes eine Speicherzelle repräsentiert. Um diese anzusteuern, wird eine „Tannebaum“-Schaltung verwendet (Abb. 1).

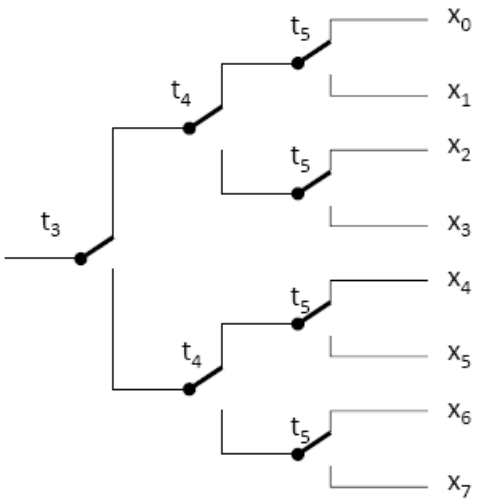


Abbildung 1: 'Tannebaum'-Schaltung mit 3 Adress-Bits[2].

[9]Ein Relais ist ein elektrischer Schalter, der zwei Zustände annehmen kann.

Hieraus ergibt sich, dass $2^n$ Adressen vergeben werden können, mit:

$$n = \text{Anzahl der Adress-Bits des } \textit{opcode}$$

Da wir für unsere Simulation von einer beim Start des Programms festgelegten, aber beliebigen Größe des Speichers ausgehen, ist die „Tannebaum“-Schaltung in unserer Animation nicht dargestellt. Stattdessen zeigen wir auf Seite des Speichers nur die Schalter $t_1$ und $t_2$.

Um den Inhalt eines Speicher-Relais in ein Register zu laden, wird eine andere Schaltung verwendet (Abb. 2).
Vor jedem Speicher-Relais gibt es einen Kontroll-Schalter ($x_0 ... x_m$).
Soll nun eine LOAD-Operation durchgeführt werden, wird der entsprechende Kontroll-Schalter gesetzt und je nach Status des Speicher-Relais ($c_1 ... c_m$) verändert sich das Register, wobei die Flagge Pr bestimmt, welches Register gesetzt wird.
Eine STORE-Operation würde in diesem Bild eines der Relais $c_1 ... c_m$ verändern (die entsprechende „Tannebaum“-Schaltung ist nicht abgebildet und würde an $t_2 = 1$ beginnen).

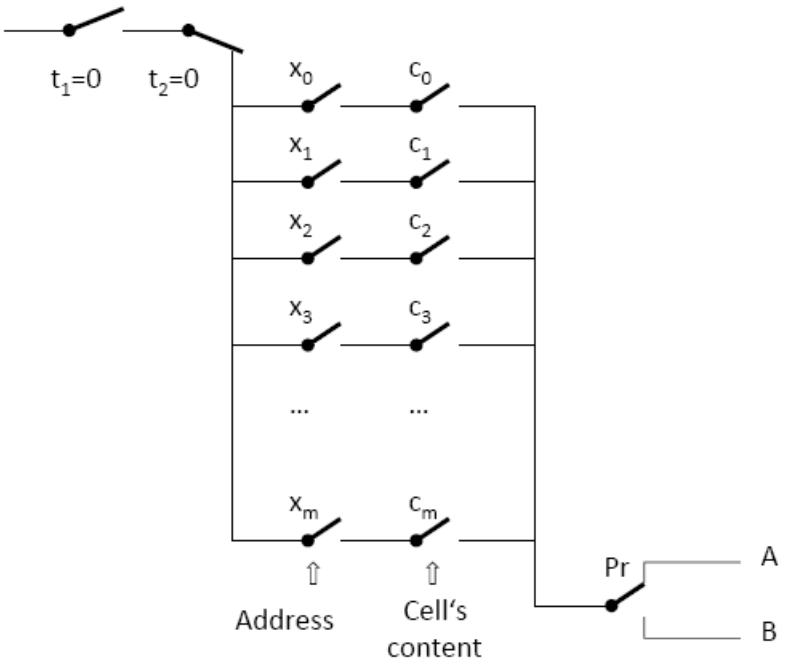


Abbildung 2: Schaltung zum Laden von Werten aus dem Speicher[2].

## 2.2 Der Prozessor

Die Schaltung des Prozessors ist vergleichsweise übersichtlich (Abb. 3).

Damit der Prozessor — und nicht der Speicher — ausgewählt ist, muss $t_1 = 0$ sein.
Wenn $t_2 = 0$, wird keine Spannung an die Logik-Schaltung angelegt, es handelt sich also um die Nulloperation.
Ist die Logik aktiviert, kann man erkennen, dass die beiden $t_3$-Schalter $\wedge$ bzw. $\vee$ einstellen und $t_4$ / $t_5$ jeweils ein Register negieren können.

Das Resultat der Berechnung wird wieder in Register A abgelegt.

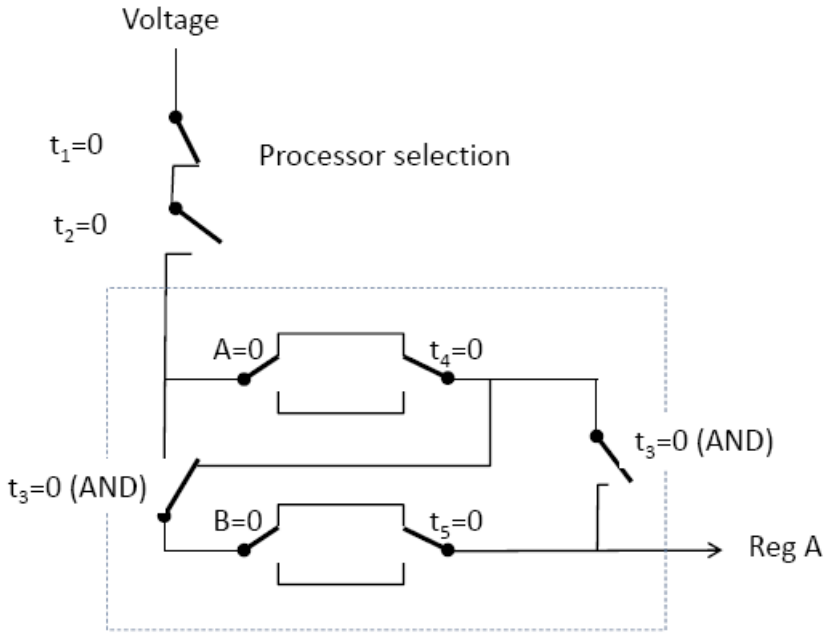


Abbildung 3: Der Prozessor der 'logistischen Maschine'[2].

## 2.3 Das Befehlsband

Da wir in unserer Simulation von einem beliebig großen Befehlsband ausgegangen sind, zeigen wir immer nur einen Ausschnitt (siehe Abb. 6).

Der aktuelle Befehl wird immer komplett ausgeführt, bevor das Band weiterläuft. Überlegungen hinsichtlich einer besseren Performance (z.B. *pipelining*[10]) sind für die „logistische Maschine“ nicht relevant, da es Zuse lediglich darum ging, einen

[10]In modernen Prozessoren werden Befehle in Teilschritte unterteilt, so dass z.B. bereits der nächste Befehl geladen wird, während der aktuelle noch in Bearbeitung ist.

möglichst reduzierten Computer zu entwerfen. Schon der Prozessor der „Z1“ ist um einiges ausgefeilter als der der „logistischen Maschine’.

## 2.4 Anmerkungen

Zuse wollte eine möglichst einfache, arithmetisch vollständige Maschine erschaffen. Mit dem Befehlssatz der „logistischen Maschine“ lassen sich tatsächlich alle arithmetischen Operationen ($+$, $-$, $*$, $/$) simulieren (siehe Anhang).
Man hätte aber die logischen Operationen noch auf $A \uparrow B$ reduzieren können, da:

$$\neg A = A \uparrow A$$
$$A \wedge B = (A \uparrow B) \uparrow (A \uparrow B)$$
$$A \vee B = (A \uparrow A) \uparrow (B \uparrow B)$$

Des weiteren ist anzumerken, dass man eine Berechnung des Prozessors nicht unterbrechen kann (im Gegensatz zur „Z3“ und „Z4“, die eine *Exception*[11] für das Teilen durch 0 haben).
Ohne diese Möglichkeit lassen sich *WHILE*-Schleifen nicht implementieren, daher ist die Maschine nicht universell.

[11] Damit ein Programm nicht einfach abstürzt, wenn Komplikationen auftreten, kann man *Exceptions* definieren. Sie werden aktiviert, wenn ein Fehler auftritt und ermöglichen eine auf das genaue Problem angepasste Reaktion.

# 3 Lösungsansatz

Nachdem ich mich mit Hilfe eines Papers[2], in dem die „logistische Maschine“ beschrieben wird und nach Rücksprache mit meinem Betreuer mit dem Problem vertraut gemacht hatte, war der erste Schritt die Wahl einer für das Projekt geeigneten Programmiersprache.

Als sinnvolle Option erschien Java, da die umfangreiche GUI-Bibliothek „Java Swing“ Bestandteil der JRE (*Java Runtime Environment*) ist.
Dadurch ließe sich das Programm ohne größere Umstände auf zahlreichen Maschinen ausführen.
Hinzu kommt, dass Java Swing seit vielen Jahren etabliert ist, daher über eine umfassende Dokumentation verfügt und detaillierte Literatur zu finden ist.

Es war für mich sehr wichtig, im Rahmen dieser Arbeit möglichst viel Neues zu lernen und da ich während meines bisherigen Studiums schon mehrfach mit Java programmiert habe, entschied ich mich dagegen.
Auch wenn mir graphische Applikationen im Allgemeinen — und Java Swing im Besonderen — fremd waren, wollte ich mich lieber in für mich gänzlich unerforschtes Gebiet wagen und eine mir noch unbekannte Sprache nutzen.

Nun gibt es natürlich zahlreiche Alternativen.
Glücklicherweise fanden sich im weiten Meer der Unkenntnis doch einige Inseln - Sprachen, von denen ich bereits genug gehört hatte, um mein Interesse zu wecken.
Die simple und elegante Syntax von Python hatte mich schon zuvor angezogen — nun hatte ich einen guten Grund mich darin einzuarbeiten.

Als nächstes galt es, eine geeignete GUI-Bibliothek zu wählen. Hierbei gab es ebenfalls vielfältige Optionen, nach einiger Überlegung habe ich mich jedoch für die am meisten genutzte Bibliothek „TkInter“ entschieden.
Nachdem ich mich etwas eingearbeitet hatte, habe ich eine kleine Demo erstellt und fand auf diesem Wege heraus, dass dieses Werkzeug für meine Zwecke geeignet war.

Da es sich um ein sehr klar definiertes, übersichtliches Problem handelt, konnte ich mit meinem Lösungsansatz etwas experimentieren, ohne den Erfolg des Projekts zu gefährden.
Sowohl die Wahl der Sprache als auch die der Bibliothek waren nicht perfekt für das Programm. Im Großen und Ganzen bin ich jedoch mit dem Ergebnis meiner Arbeit zufrieden.

Wie bereits erwähnt, habe ich zunächst eine kleines Testprogramm entwickelt, um mich mit Python und TkInter vertraut zu machen. Da ich mich gleich zu Anfang entschieden hatte, die einzelnen Teile der „logistischen Maschine“ in meiner Simulation voneinander zu trennen, habe ich mich in meinem Test nur auf das Befehlsband beschränkt.

Es hat sich relativ schnell heraus gestellt, dass das TkInter *canvas*-Objekt am besten für meine Zwecke geeignet ist.
Mit diesem kann man einen Bereich des Bildschirms zur Leinwand erklären, auf dem mit verschiedenen Methoden Polygone gezeichnet werden können.
Um das Bild in Bewegung zu setzen, habe ich eine Funktion erstellt („*animate*“), die sich periodisch selbst aufruft.

Nachdem ich ein rudimentäres Befehlsband erstellt und in Bewegung gesetzt hatte, habe ich mir Gedanken über die Struktur des Programms gemacht. An dieser Stelle kam mir die Idee, das Bild in mehrere Teile zu trennen.
Um dies zu erreichen, habe ich für jeden Teil ein eigenes *canvas*-Objekt erstellt, das ich mit dem Layout-Manager von TkInter entsprechend angeordnet habe. Somit waren die Einzelteile logisch und physisch voneinander abgegrenzt.

Damit der Benutzer das Programm bedienen kann, habe ich ein simples Interface entworfen.
In einem Menü kann man ein vorgefertigtes Programm für die „logistische Maschine“ auswählen. Dadurch öffnet sich ein neues Fenster mit der schematischen Darstellung der Maschine. Hier kann der initiale Zustand des Speichers eingestellt werden, bevor man die Animation per Knopfdruck startet. Mit einem anderen Knopf kann man jederzeit ins Menü zurückkehren.

Das in dieser Form konzipierte Layout wurde — mit einigen leichten Änderungen — von meinem Betreuer akzeptiert, sodass ich mich nun daran setzen konnte, nach und nach die Einzelteile zu erstellen.

Da sich das Befehlsband und der Speicher in meiner Animation sehr ähnlich sind, habe ich mit dem Erstellen und Animieren des Speichers begonnen.

# 4 Durchführung / Artefakte

## 4.1 Generelles

Nach dem Start des Programms erstellt sich zunächst ein TkInter-Objekt, mit dessen Hilfe (fast) alle weiteren Objekte hervor gebracht werden.

Um die gewünschte Modularisierung zu erreichen, machte ich mir die Objektorientiertheit[12] von Python zu nutze.
Jede Komponente wird von einer Klasse repräsentiert, die durch das Erzeugen einer Instanz der Klasse konfiguriert und initialisiert wird. Diese Klassen sind nur für die graphische Darstellung und Animation zuständig.
Um sie aufzurufen, die Logik der „logistischen Maschine“ zu implementieren und den Input / Output zu verwalten, gibt es eine Meta-Klasse. Für das Benutzermenü gibt es ebenfalls eine eigene Klasse.

Eine Instanz einer Komponentenklasse erstellt zunächst ein *canvas*-Objekt passender Größe an einer bestimmten Stelle des Bilds, das je nach Anforderung noch mit weiteren Objekten gefüllt wird (Linien für Kabel, Rechtecke für Speicherzellen, Texte etc.).
Jede Komponentenklasse hat zudem eine oder mehrere „*animate_**“ genannte Funktionen, mit denen der von ihr dargestellte Bildausschnitt bewegt werden kann.

Ein wiederkehrendes Problem war, dass es häufig nötig war, auf das Ende der Animation eines Ausschnitts zu warten, bevor die Animation des nächsten beginnen konnte. Glücklicherweise gibt es in TkInter die Möglichkeit, die Exekution des *mainloop*[13] mit der Methode „*wait_variable(var)*“ so lange zu stoppen, bis die Variable („*var*“) ihren Zustand ändert.

[12] In der Objektorientierten Programmierung teilt man sein Programm in verschiedene Objekte, die miteinander kommunizieren können. Sie dient vor allem dazu, die Architektur des Programms zu strukturieren und so verständlicher zu gestalten. Ähnliche Objekte können zu Klassen zusammengefasst werden und ihnen sind Attribute und Methoden zugeordnet.

[13] Diese Schleife wird anfangs gestartet und erst unterbrochen, wenn das Programm explizit beendet wird. Dadurch läuft das Programm selbst dann weiter, wenn es gerade im Ruhezustand ist.

Nicht zuletzt gibt es noch die Datei, in der die *main*-Funktion steht. Hier wird das TkInter-*mainloop* gestartet, sowie die Meta-Klasse und das Benutzer-Interface instantiiert.

Der gesamte Code wurde mit Hilfe der „PyCharm“-IDE[14] erstellt.

## 4.2 Die Klassen im Detail

### 4.2.1 zuse_sim

Im Grunde handelt es sich nicht um eine Klasse, sondern (nur) um eine Reihe von Instruktionen.
In der *main*-funktion wird ein TkInter-Objekt erstellt und konfiguriert, mit besonderem Schwerpunkt auf die Dimension des Bildes und seine Hintergrund-Farbe. Anschließend wird das Menü erstellt und schließlich der Tk-*mainloop* gestartet.

### 4.2.2 Hauptmenü

Das Menü besteht aus einer Reihe von Knöpfen, mit denen das gewünschte Programm für die „logistische Maschine“ gewählt werden kann.
So öffnet sich immer ein neues Fenster mit entsprechender Voreinstellung bezüglich Speicher-Größe und Befehlsband (Abb. 4).

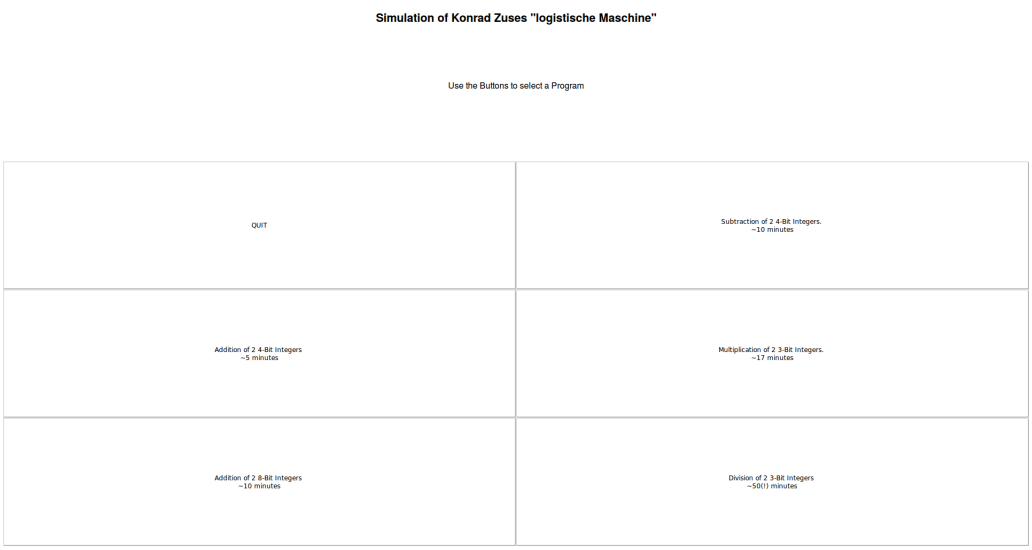


Abbildung 4: Das Hauptmenü.

[14] Ein *integrated development environment* (IDE) ist ein Programm, welches das Erstellen, Strukturieren und Warten von Code erleichtert.

### 4.2.3 App (Meta-Klasse)

Hier werden zunächst alle anderen Klassen in einer bestimmten Reihenfolge instantiiert und Variablen gesetzt.
Im folgenden wird die dem Knopf zugehörige Textdatei eingelesen. In ihr findet sich die initiale Konfiguration des Speichers und alle auszuführenden Befehle.
Nachdem das Bild entsprechend der Textdatei initialisiert wurde, werden nacheinander alle Befehle abgearbeitet:
Eine Funktion identifiziert den Befehl und ruft eine andere Funktion mit entsprechenden Variablen auf. Diese verändert den Zustand der „logistischen Maschine“ und setzt die Animation in Gang. Nach Ende der Animation wird so lange der als nächstes folgende Befehl ausgewählt, bis es keinen weiteren mehr gibt, und das Bild anhält.
Je nach ausgeführtem „logistischem Maschine“-Programm hat sich der Zustand der Maschine verändert und eventuell steht nun etwas Sinnhaftes im Speicher.

Die Meta-Klasse hat selbst keine *animate*-Funktion, da ihre Aufgabe das Steuern der anderen Klassen ist.

### 4.2.4 Kabel

Da das Bild die Schaltung der „logistischen Maschine“ nachempfinden soll, musste ich reichlich Kabel verlegen. Mit Ausnahme eines kleinen Bereichs sind sie stets nur Bestandteil einer Klasse.
Um den Stromfluss darzustellen, habe ich die Linien in viele Fragmente aufgeteilt, die unabhängig voneinander angesteuert werden können. Wenn diese nun in schneller Folge und Reihenfolge die Farbe wechseln, entsteht die Illusion einer sich füllenden Linie — ähnlich eines Lade-Balkens (Abb. 5).



Abbildung 5: Das Kabel färbt sich rot um Stromfluss darzustellen.

### 4.2.5 Schalter

Die Schalter sind durch einen Kreis an einem Ende gekennzeichnet und können ebenfalls eingefärbt werden, um den Stromfluss darzustellen.
Durch Rotation der Linie wird der Schalter verstellt.

Wie die Kabel sind sie Bestandteil der meisten Klassen. Färben und Rotieren sind in zwei verschiedenen Funktionen implementiert, um die korrekte Sequenzierung zu vereinfachen (Abb. 6).

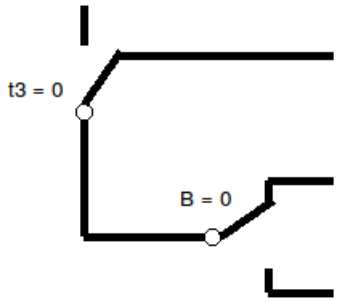


Abbildung 6: Schalter

### 4.2.6 Befehlsband

Wie bereits in 2.3 erläutert, habe ich mich dazu entschieden, immer nur einen Ausschnitt des Befehlsbandes sichtbar zu machen.

In der Mitte des Ausschnitts befindet sich immer genau der Befehl, der gerade ausgeführt wird.
Um die aktuelle Selektion zu verändern, wird der gesamte sichtbare Bereich verschoben, bis sich das gewünschte Element an der richtigen Stelle befindet. Damit die besondere Bedeutung dieses Elements sichtbar wird, habe ich es vergrößert (Abb. 7).

Wenn der aktuelle Befehl beendet ist, verkleinert sich das aktive Element, während sich das nächste in der Reihe entsprechend vergrößert. Dadurch wird die Hervorhebung verdeutlicht und es entsteht eine flüssige Bewegung.

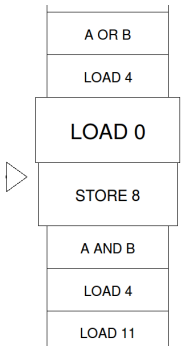


Abbildung 7: Befehlsband

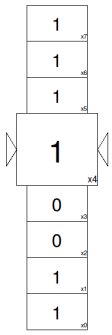


Abbildung 8: Speicher

### 4.2.7 Speicher

Der Speicher ist dem Befehlsband sehr ähnlich (Abb. 8).

Die Speicherzellen haben einen Index, damit sie identifiziert werden können, aber auch hier ist das aktuelle Element in der Mitte des Ausschnitts angesiedelt. Auf besagtes Element wird zugegriffen, wenn eine *LOAD*- oder *STORE*-Operation durchgeführt wird.
Verändert sich der Inhalt des Speichers durch eine Operation, leuchtet die entsprechende Zelle kurz auf, um den Vorgang zu visualisieren.

Wie schon eingangs erwähnt, ist die eigentliche Schaltung des Speichers nicht in meiner Simulation dargestellt. Die notwendige Menge an Verkabelung wäre von der initialen Speichergröße abhängig, die wir jedoch als beliebig definiert hatten.
Vor allem würde das Bild aber schon bei einer sehr kleinen Speichergröße unnötig unübersichtlich werden. Außerdem ist die „Tannebaum“-Schaltung nicht der spannende Teil der „logistischen Maschine“.

### 4.2.8 Memop

Um das Problem der fehlenden Schaltung zu umgehen, habe ich einen Bereich eingeführt, der in der originalen „logistischen Maschine“ von Zuse nicht existiert.

Anstatt die komplexe Verkabelung darzustellen, gibt es nur zwei *black boxes*: *LOAD* und *STORE*.
Soll nun eine Operation auf dem Speicher erfolgen, endet der Stromfluss bei einer dieser Boxen, im Anschluss findet die entsprechende Änderung im Speicher statt (Abb. 9).

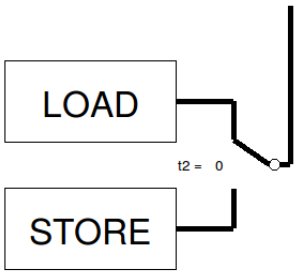


Abbildung 9: Hier entscheidet sich, wie auf den Speicher zugegriffen wird.

### 4.2.9 Prozessor

Das Herzstück der Maschine und der interessanteste Teil für die Präsentation ist natürlich der Prozessor.

Die hier dargestellte Schaltung entspricht der von Zuse erdachten und implementiert die beiden logischen Operationen (Abb. 10).

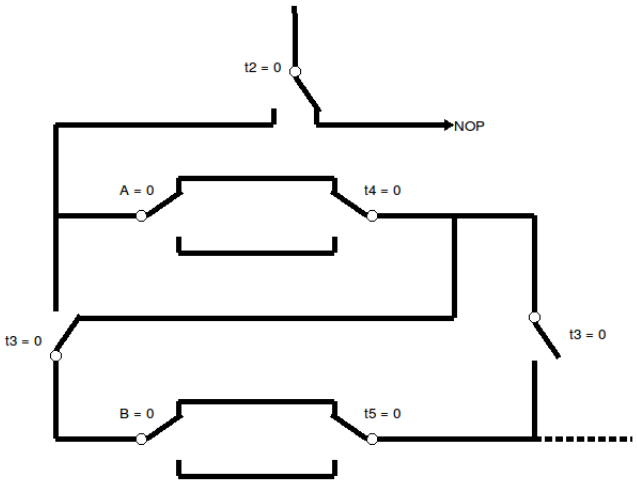


Abbildung 10: Der Prozessor

An dieser Stelle lässt sich sehr schön die Funktionsweise des eingangs erwähnten *opcode* ablesen, da dessen Bits die Position der Schalter bestimmen.
Insbesondere die beiden $t_3$-Schalter sind interessant, da sie die Art der Operation ($\wedge$ / $\vee$) einstellen.
Die Register der Maschine sind de facto auch nur Schalter im Prozessor und $t_4$ / $t_5$ bestimmen, ob auf ihrer Negation operiert werden soll.
Das Ergebnis der Berechnung wird wieder in Register A gespeichert. Damit die Schaltung übersichtlich bleibt, ist dieser Kreislauf nur indirekt dargestellt.

### 4.2.10 Register

Die soeben erwähnte indirekte Darstellung dieses Kreislaufs bezieht sich auf einen von mir zugefügten Bereich, in dem der aktuelle Status der Register noch einmal hervorgehoben ist (Abb. 11).

Dies dient der Übersichtlichkeit und verdeutlicht die Sonderstellung der Register im Gegensatz zu den anderen Schaltern im Prozessor.

Zudem konnte ich auf diese Weise eine weitere, der Speicher-Schaltung zugehörige Mechanik vom Prozessor trennen: die „Pr-Flagge“.
Letztere bestimmt, in welches der Register die Information aus dem Speicher bei einer *LOAD*-Operation geladen wird.

Da ich auf die Darstellung der Speicher-Schaltung verzichtet habe, hat sich dieser Extra-Bereich also aus verschiedenen Gründen angeboten.

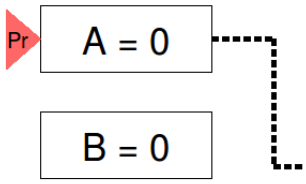


Abbildung 11: Der Zustand der Register ist jederzeit ersichtlich.

### 4.2.11 Rahmen

Mein Betreuer hat gegen Ende der Entwicklung noch vorgeschlagen, aus ästhetischen Gründen einen Rahmen um das Bild zu ziehen.
Die dazu notwendigen Klassen sind — wenig überraschend — simpel ausgefallen.

Im Zuge dessen habe ich auch noch einmal mein Layout aufgeräumt und die Anordnung der Einzelteile überarbeitet (Abb. 12).

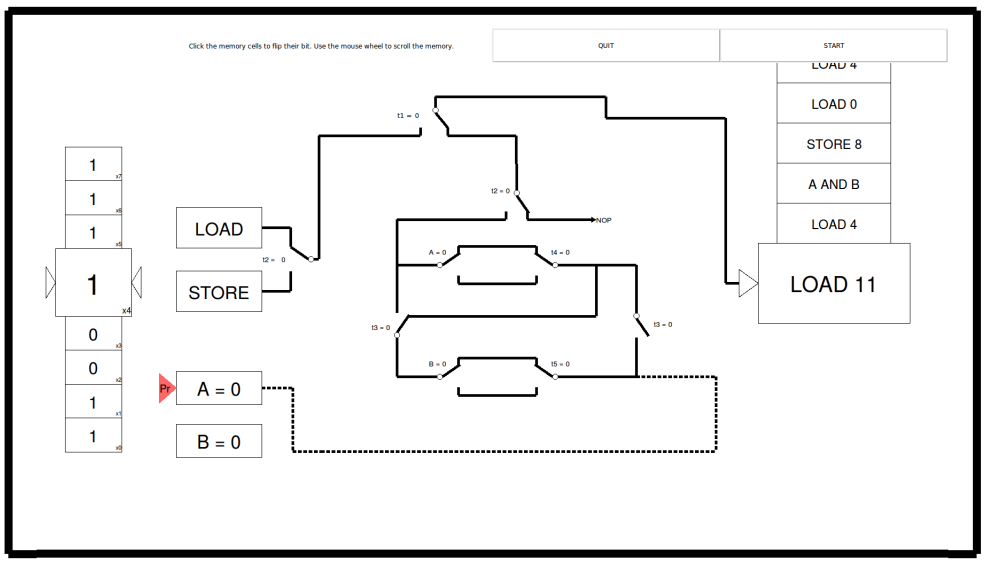


Abbildung 12: Das Endergebnis

# 5 Schwierigkeiten und Probleme

## 5.1 Wahl der Bibliothek

Nicht zuletzt aufgrund meiner Unerfahrenheit in der GUI-Programmierung habe ich mit TkInter eine Bibliothek gewählt, die mir nur wenig Arbeit abgenommen hat.

Die Animationen liefen in der Regel auf Veränderungen von Figuren in einem Koordinatensystem hinaus, wobei ich die Berechnung der neuen Koordinaten stets selbst implementieren musste (z.B. Rotation eines Schalters, Verschieben und Skalieren von Rechtecken).

Der Vorteil dieser rudimentären Bibliothek war eine geringe Einarbeitungszeit, da es von vornherein nur geringen Erklärungsbedarf bezüglich ihrer Funktionsweise gab.

## 5.2 Wahl der Sprache

Wie schon eingangs erwähnt, habe ich mich gerade deshalb für Python entschieden, weil mir die Sprache unbekannt war. Daher ist es wenig verwunderlich, dass ich vor allem anfangs recht viel Zeit investieren musste, um mich mit ihr vertraut zu machen.

Im Laufe der Entwicklung hat sich herausgestellt, dass Python als Skript-Sprache[15] mit dynamischer Typisierung[16] nicht besonders gut für größere Projekte geeignet ist.

Da mein Programm aber einen übersichtlichen Umfang hat, sind die Probleme bezüglich der Lesbarkeit in erster Linie auf meine limitierte Erfahrung und die partiell nicht ganz ausgereifte Planung zurückzuführen.

Ein besonders auffälliger Punkt ist, dass ich oft Variablen erstellt habe, die nur innerhalb der Klasse sichtbar sein sollten, aber von außen manipuliert werden konnten. Da dieser Aspekt für das Gelingen des Projekts nicht entscheidend war, habe ich die Suche nach einer Lösung nach kurzer Zeit aufgegeben.
Dies ist eines der Dinge, die mir mit Java nicht passiert wären.

[15] Bei Skript-Sprachen handelt es sich um Programmiersprachen, deren Code von einem *Interpreter* übersetzt wird. Im Gegensatz zum *Compiler* übersetzt der *Interpreter* das Programm zur Laufzeit. Der Vorteil zum *Compiler*, der das Programm einmal komplett übersetzt und dadurch eine ausführbare Datei erzeugt, ist, dass man keine Anpassungen für verschiedene Rechner-Architekturen machen muss, sofern der *Interpreter* selbst lauffähig ist. Skript-Sprachen verzichten oft auf Sprachelemente, deren Nutzen erst bei der Bearbeitung komplexerer Aufgaben zum Tragen kommt. Dies hat oft Vorteile für kleine Programme, schadet aber der Übersichtlichkeit größerer.

[16] In vielen Sprachen muss man den Typ einer Variablen beim Erstellen angeben. Eine Sprache mit dynamischer Typisierung wählt den Typen zur Laufzeit aus dem Kontext. Bei einer großen Menge von Variablen kann das ein Nachteil sein.

## 5.3 Programmstrukur / Planung

Aus Erfahrung weiß ich, dass eine sorgfältige Planung extrem wichtig für den Erfolg eines Projektes ist. Vor allem die Lesbarkeit, Wiederverwendbarkeit und Effizienz des Codes ist stark abhängig von der Vorbereitung.
Aufgrund dieser Einsicht habe ich mich in der Anfangsphase der Entwicklung hauptsächlich darauf konzentriert, die nächsten Schritte abzuwägen und eine Struktur für das Programm zu entwerfen.
Obwohl ich in diesem Sinne vorbereitet war, sind im Laufe der Implementierung Probleme aufgetreten, für deren Lösung mein Entwurf nicht ideal war.

Die Teilung des Bildes in unabhängige Einzelteile war an sich keine schlechte Idee, allerdings habe ich so das volle Potential des OO-Prinzips der Klasse nicht genutzt. Statt Gemeinsamkeiten in den kleineren Bauteilen der Maschine zu suchen und diese als Klassen zu beschreiben, von denen dann entsprechend viele Instanzen erstellt werden können, ist meine Teilung zu grob geraten:
Von jeder Klasse wird immer nur eine Instanz erstellt - das gesamte Bauteil an sich (Befehlsband, Speicher, Prozessor etc.).

Des weiteren hätte es sich angeboten, Animation und Genesis eines Objekts zu trennen.
Da jedes Bauteil bei mir ein in sich geschlossenes System darstellt, findet sich beides immer in derselben Klasse. Das Resultat einer solchen Struktur ist eine hohe Redundanz im Code, da sich beispielsweise die Animation von Speicher und Befehlsband nur in Details unterscheiden, aber in Gänze in beiden Klassen stehen.
Auch die Konstrukte um Kabel zu erstellen und zu animieren finden sich an mehreren Stellen.
Um den Rahmen darzustellen, musste ich sogar noch zusätzliche, praktisch identische Klassen erstellen, da der diskriminierende Faktor der Klassen die Position der von ihnen erschaffenen Objekte auf dem Bildschirm ist.

Die Fokussierung beeinflusst auch die Wiederverwendbarkeit des Codes.

Eine Klasse, die auf Wunsch Objekte wie Kabel, Behälter oder Schalter erstellen und / oder animieren kann, wäre auch für künftige Projekte nützlich — eine Klasse, die ein Befehlsband erstellt und animiert ist wesentlich schwerer in andere Programme zu integrieren.

Meine Struktur erlaubt es außerdem nicht, die Position eines Bauteils auf dem Bild-

schirm dynamisch zu verändern. Für unsere Zwecke ist dies nicht relevant, aber in einer besseren Architektur könnte die Positionierung der Einzelteile frei wählbar sein.

Das Benutzermenü war in meinem ersten Entwurf nicht vorgesehen und wurde erst am Ende der Entwicklung eingefügt. Das konfliktfreie Integrieren dieses Programmteils hat daher auch mehr Zeit gekostet.

## 5.4 Details in der Implementierung

Um TkInter-Objekte auf einem Bereich des Bildschirms zu positionieren, gibt es drei verschiedene *geometry manager* (*„grid“*, *„pack“* und *„place“*).
Beim Erstellen des Benutzer-Menüs musste ich etwas tricksen, damit die Knöpfe die von mir gewünschte Größe haben:
Ich habe *grid* benutzt, um Fenster anzuordnen, in denen dann die Buttons mit *pack* positioniert werden. In der Regel ist das Vermischen dieser Werkzeuge zu vermeiden, allerdings habe ich keine andere Möglichkeit gefunden, mein Layout umzusetzen.

Das TkInter-Objekt *„canvas“* entspricht einem Koordinatensystem, auf dem mit verschiedenen Funktionen Objekte gezeichnet werden können. Die erstellten Objekte haben eine ID, mit der man sie ansprechen kann.
Es gibt eine Funktion, um ihnen neue Koordinaten zu geben und sie somit zu bewegen (*„coords(ID,x1,y1,...,xn,yn)“*). Vordefinierte Funktionen für Rotationen oder Skalierungen sind nicht vorhanden.

Die Animation eines Objekts wird erreicht, indem man für jedes *Frame*[17] die Koordinaten des Objekts berechnet. Diese Berechnungen mussten alle von Hand implementiert werden, was einen nicht unerheblichen Teil der Gesamtarbeitszeit in Anspruch genommen hat.

Da ich wollte, dass das Endergebnis auf einem beliebig großen Bildschirm im Vollbild angezeigt werden kann, musste ich bei allen Berechnungen für Koordinaten mit relativen Werten arbeiten. Beim Start des Programms wird die Bildschirmgröße ermittelt und in einer Variablen abgelegt.

Die Dimension eines Bereiches ist also nicht in natürlichen Zahlen angegeben, sondern als:

$$\frac{a}{b} \cdot \text{Bildschirm-Breite} \times \frac{c}{d} \cdot \text{Bildschirm-Länge}$$

[17] Ein *Frame* ist ein einzelnes Bild. Um Bewegung zu simulieren, werden leicht veränderte Bilder in schneller Folge berechnet und dargestellt.

Diese Art die Dimension eines Objekts/Bereichs zu beschreiben, zieht sich durch das gesamte Programm und hat die nötigen Berechnungen zeitweise verkompliziert.

Die Art und Weise, wie man mit TkInter eine Animation implementiert, hat mich auch etwas Eingewöhnungszeit gekostet: Man erstellt eine sich selbst aufrufende Funktion (*„animate“*), die jedes *Frame* berechnet.
Diese rekursive Natur der *animate*-Funktionen war etwas gewöhnungsbedürftig, da es mitunter nicht ganz trivial war, die richtige Abbruchbedingung zu finden und / oder einzelne Berechnungen nicht unnötig zu wiederholen.
Da die Funktions asynchron[18] ausgeführt wird, muss man eine zusätzliche Variable einführen, auf deren Veränderung der Haupt-*Thread*[19] warten muss, damit die Animationen in der richtigen Reihenfolge ausgeführt werden.

Nicht zuletzt musste ich die von mir auf dem *canvas* erstellten Objekte verwalten:
Die ID's werden in Listen (Python kennt keine *Arrays*[20]) gespeichert und wenn ein Objekt aus dem Bild verschwindet, muss es manuell gelöscht werden.
Damit die Kabel den Stromfluss darstellen können, bestehen sie aus vielen kleinen Linien, weswegen sie als Liste von Linien implementiert wurden.
Da manche Informationen nicht auf dem Bild zu sehen sind, musste ich stets darauf achten, dass die Darstellung den internen Hintergründen entspricht.

[18] Eine asynchron aufgerufenes Stück Code wird parallel zum Rest des Programms berechnet.
[19] Ein Programm kann seinen Code auf mehrere *Threads* aufteilen. Diese Sub-Programme laufen unabhängig voneinander. Jedes Programm hat einen Haupt-*Thread* - wenn dieser beendet ist, endet auch das Programm.
[20] Ein *Array* ist ein Container, in dem andere Objekte / Variablen abgelegt werden können (auch weitere *Arrays*). Eine Liste löst die gleichen Probleme, wobei sie in der Regel langsamer aber flexibler ist.

# 6 Evaluation

Das Ziel meiner Arbeit war von Anfang an klar beschrieben:
Die „logistische Maschine“ simulieren und animieren, sodass sie dem Betrachter begreiflich wird.

Bedauerlicherweise war nicht ausreichend Zeit vorhanden, um die Simulation ausführlich zu testen. Ob in der Praxis noch *Bugs* oder Unzulänglichkeiten Auftreten, wird sich wohl noch zeigen.

Da das Benutzermenü erst kurz vor Ende des Projekts hinzugefügt wurde, implementiert es nur die nötigsten Funktionen und ist auch sonst eher rudimentär.

Es ist sicherlich deutlich geworden, dass ich mit der von mir gewählten Programmstruktur und Implementierung an vielen Punkten unzufrieden bin. Diese zu verändern hätte bedeutet, das Programm komplett neu zu schreiben — ein für den zeitlichen Rahmen einer Bachelorarbeit unrealistisches Unterfangen.

Trotz allem empfinde ich diese Arbeit im Großen und Ganzen als ein erfolgreiches Unterfangen:
Es war nicht mein Ziel, ein optimales Programm zu entwickeln, sondern an der mir gestellten Aufgabe möglichst viel zu lernen.
Ich konnte meine Kenntnisse einer Programmiersprache erweitern, habe gelernt, die enorme Wichtigkeit der Projektplanung nicht zu unterschätzen und mein Verständnis für objektorientiertes Programmieren vertieft.

Die persönlichen Bereicherungen allein wären jedoch nicht ausreichend für eine erfolgreiche Arbeit, weshalb ich zufrieden bin, dass mein Programm trotz aller Mängel die gestellten Anforderungen erfüllt.

# 7 Ausblick

Da die Simulation für eine bestimmte Präsentation geschrieben wurde, liegt die Vermutung nahe, dass sie nur in diesem Rahmen von besonderem Nutzen sein wird.

Nichtsdestotrotz bin ich gespannt auf die Reaktion des Publikums, da ich dessen Kritik für zukünftige Projekte sicherlich gut gebrauchen kann.
Sollte in Zukunft ein ähnliches Programm benötigt werden, müsste es vermutlich neu geschrieben werden, da meine Implementierung an zu vielen Punkten unflexibel ist.
Durch die in dieser Arbeit erworbenen Fähigkeiten würde mir die Entwicklung solch einer verbesserten Version leichter fallen.

Sollte es sich in der Praxis jedoch als *Bug*-frei erweisen, steht einer wiederholten Nutzung des Programms nichts im Wege.

# 8 Anhang - Quellcode

Der Quellcode des Programms mit Installationsanweisung kann auf GitHub gefunden werden:

https://github.com/mdieckmann/zuse_sim

# 9 Anhang - arithmetische Vollständigkeit[21]

## 9.1 Addition

Um zwei Bits A und B mit dem Übertrag C zu addieren, kann folgende Formel verwendet werden:

$$A \oplus B \oplus C$$

Der Übertrag C lässt sich durch

$$(A \wedge C) \vee (A \wedge B) \vee (B \wedge C)$$

bestimmen.

Zudem gilt:

$$A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$$

Um zwei Zahlen zu addieren, berechnen wir also jedes Bit-Paar nacheinander und legen die Ergebnisse an entsprechender Stelle im Speicher ab.

## 9.2 Subtraktion

Das Zweierkomplement erlaubt es uns, negative Zahlen darzustellen. Statt zu subtrahieren, addieren wir einfach negative Zahlen:

$$x - y = x + (-y)$$

Um das Zweierkomplement einer Zahl zu bilden, erzeugt man zunächst das Einerkomplement. Dafür muss schlicht jedes Bit negiert werden, was z.B.: durch den Befehl

$$\neg A \vee 0$$

erreicht werden kann. Anschließend muss nur noch 1 addiert werden.

[21] Da die folgenden Erläuterungen etwas mehr in die Tiefe gehen, bin ich davon ausgegangen, dass der interessierte Leser über Vorkenntnisse verfügt. Aus diesem Grund habe ich darauf verzichtet, die auftretenden Fachbegriffe — wie im Rest der Arbeit — zusätzlich zu erklären.

## 9.3 Multiplikation

Der übliche Algorithmus um zwei Zahlen miteinander zu multiplizieren, ist das Bilden von partiellen Produkten, die anschließend addiert werden.
Die Partialprodukte der binären Multiplikation sind erfreulich simpel, da einer der Faktoren immer entweder 0 oder 1 ist.
Somit ist das Ergebnis entweder 0 oder die Zahl selbst (mit entsprechendem Shift).

Nachdem auf diese Weise die Partialprodukte errechnet sind, müssen sie nur noch addiert werden.
Beispiel (Basis 10):

$$\begin{array}{r} 43 \\ \times\, 54 \\ \hline 162 \\ +\, 216 \\ \hline 2322 \end{array} \tag{1}$$

Beispiel (Basis 2):

$$\begin{array}{r} 1011 \\ \times\, 1110 \\ \hline 0000 \\ 1011 \\ 1011 \\ +\, 1011 \\ \hline 10011010 \end{array} \tag{2}$$

## 9.4 Division

Wie auch zuvor, errechnen wir das Ergebnis Bit für Bit. Wir betrachten beispielhaft die beiden 4-Bit Zahlen n und m mit Rest r. Dann ist:

$$\tfrac{n}{m} = (2^3 b_3 + 2^2 b_2 + 2^1 b_1 + 2^0 b_0)m + r$$

Wir berechnen $b_k$, in dem wir $n - (2^k * m)$ betrachten.
Ist das Ergebnis negativ, ist $b_k = 0$.
Ist das Ergebnis positiv, ist $b_k = 1$.

Wir testen gewissermaßen, ob wir m in dieser Größenordnung abziehen können, ohne n gänzlich zu „verbrauchen“.
Ob die Zahl negativ ist, können wir am *most significant bit* ablesen. Wenn wir etwas

abziehen können, muss sich anschließend natürlich n noch verändern bevor wir mit $b_{k-1}$ fortfahren.

Sind alle $b_k$ berechnet, ist der Rest r gleich dem verbleibenden n.
Ist der Quotient 0 (wenn Divisor > Divident), ist $r = m$.

Insgesamt ergibt sich folgender Algorithmus:

begin
  for $k := i$ to 0 step $-1$ do
    comment: Versuch von n abzuziehen / Berechnung des (Partial-)Rests
    $r := n - (2^k * m)$
    comment: Bestimmen des Vorzeichens, $c = 1 \rightarrow$ negativ, $c = 0 \rightarrow$ positiv
    $c := sign(r)$
    comment: $b_k$ ist die Negation von c
    $b_k := 1 - c$;
    comment: wenn r negativ war bleibt n erhalten, sonst wird r das neue n
    $n := (1 - c) \cdot r + c \cdot n$
  od
  comment: Prüfen, ob Quotient = 0
  $h = b_k \vee b_{k-1} ... \vee b_0$
  comment: r entsprechend anpassen
  $r = (1 - h) \cdot m + h \cdot r$
end

# Literatur

[1] Heidelberg Laureate Forum Foundation (HLFF). *Heidelberg Laureate Forum*. [Online], accessed 24-July-2016. Available: http://www.heidelberg-laureate-forum.org/.

[2] Prof. Dr. Raúl Rojas. *Konrad Zuse's Computer for Binary Logic*, 2014.

[3] Wikipedia. *Konrad Zuse*. [Online], accessed 24-July-2016. Available: https://de.wikipedia.org/wiki/Konrad_Zuse.

[4] Wikipedia. *Zuse Z3*. [Online], accessed 24-July-2016. Available: https://de.wikipedia.org/wiki/Zuse_Z3.

[5] Wikipedia. *Zuse Z4*. [Online], accessed 24-July-2016. Available: https://de.wikipedia.org/wiki/Zuse_Z4.