Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Simulation Konrad Zuses logistischer Maschine

Matthias Kupferschmidt
Matrikelnummer: 4459671

Berlin, 27. Juni 2016

Betreuer: Prof. Dr. Raúl Rojas, Julian Röder

1. Gutachter: Prof. Dr. Raúl Rojas
2. Gutachter: Prof. Dr. Tim Landgraf

Zusammenfassung

Konrad Zuses logistische Maschine ist ein sehr einfacher, arithmetisch vollständiger Relaiscomputer, dessen Prozessor beispielsweise aus nur fünf Relais besteht. Im Rahmen dieser Arbeit wurde eine interaktive Simulation entwickelt, die diese Maschine in Aktion zeigt. Die Visualisierung ist geeignet, um Laien zu demonstrieren, wie aus dem Zusammenspiel weniger, leicht zu verstehender Bauteile ein programmierbarer Computer wird.

Abstract

Konrad Zuse's "logistische Maschine" (logical machine) is a very simple but arithmetically complete electromechanical computer, for instance having a processor consisting of only five relays. In this paper, an interactive simulation was developed showing this machine in action. The visualisation is particularly suited for demonstrating to laymen how the interaction of only a few simple-to-understand components bears the power of a programmable computer.

Eigenständigkeitserklärung

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.

Berlin, 27. Juni 2016

Matthias Kupferschmidt

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Kernidee von Software ist, einer allgemein konstruierten Maschine durch einen notierten Code ein Verhalten zu verleihen, wobei das Notieren des Codes deutlich einfacher als die Konstruktion der Maschine selbst funktioniert. So betrachtet ist jeder Schlüssel eine Form von Software für sein Schloss, das eine Maschine ist, die einen Akzeptanztest durchführt und jede Spieluhr beinhaltet eine Walze, auf der Noppen eine Melodie notieren.
Vor diesem Hintergrund ist es interessant sich damit zu beschäftigen, wie man eine solche Maschine möglichst einfach konstruieren kann, aber sie dennoch möglichst mächtig ist, also man bei der Entwicklung der Software einen möglichst großen Spielraum hat, was für ein Verhalten man der Maschine verleihen kann.
Konrad Zuses logistische Maschine ist ein gutes Beispiel eines Rechners, der sehr einfach zu konstruieren ist und dennoch durch Software auf jede beliebige arithmetische Operation programmierbar ist.
Eine Simulation dessen ist gut geeignet, um auch Laien diese Kernidee von Software nahe zu bringen, da alle Elemente der logistischen Maschine einfach verständlich sind und ihr Verhalten ohne tiefere Kenntnisse nachvollziehbar ist.

1.2 Zielsetzung

Im Rahmen dieser Arbeit soll eine Simulation Konrad Zuses logistischer Maschine entwickelt werden, die visuell für Laien verständlich ist und in sehr reduzierter Form die Funktionsweise eines Computers zeigt.

2 Grundlagen

In diesem Kapitel sollen ein paar Grundlagen gelegt werden. Wir kommen von der Funktionsweise von Relais, über ein paar geschichtliche Aspekte, zur symbolischen Notation von Relais und einigen einfachen Relais-Schaltungen, die für die logistische Maschine von Bedeutung sein werden.

2.1 Funktionsweise

Ein Relais ist nichts weiter als ein durch einen Elektromagneten mechanisch gesteuerter Schalter, der in der Regel zwei Stellungen hat. Auf diesem Wege kann ein Steuerstromkreis, der den Elektromagneten mit Strom versorgt, einen zweiten, ansonsten getrennten Stromkreis schalten.
So wie es unterschiedliche Typen von Schaltern gibt, wie etwa schließende und öffnende Schalter, gibt es auch die zugehörigen Relaistypen. Zudem gehören zu einem Relais oft auch mehrere Schalter, die durch den gleichen Elektromagneten gesteuert werden. Ein öffnender Schalter etwa würde einen zu steuernden Stromkreis bei aktivem Elektromagneten unterbrechen, während ein schließender Schalter diesen verbinden würde.
Eine andere Besonderheit sind bistabile Relais. Während monostabile Relais nämlich nach Deaktivierung des Steuerstromkreises in ihren Ausgangszustand zurückfallen, weisen bistabile Relais zwei Elektromagneten auf und werden also von zwei Steuerstromkreisen gesteuert, wobei der eine Elektromagnet das Relais in den einen Zustand bringt und es der andere Elektromagnet in den anderen Zustand bringt. Werden die Steuerstromkreise deaktiviert, so ändert das den Zustand des Relais nicht.
[Wik16a]

2.2 Vom Relais in die Computerwelt

Erfunden wurde das Relais um 1835 im Zusammenhang mit der Telegrafentechnik. Verschiedene Quellen sehen hier den amerikanischen Wissenschaftler Joseph Henry als Erfinder, wobei der Zeitpunkt aber nicht klar dokumentiert ist und daher andere Quellen den englischen Erfinder Edward Davy als Erfinder sehen. [Sau85, Wik16b] André-Marie Ampère, Pierre-Simon Laplace und viele weitere entwarfen frühe Modelle und auch Samuel Morse verbesserte Relais zum Beispiel dahingehend, auf schwächere Impulse zu reagieren. [Wik16a]
Das Problem, um das zu lösen man Relais ursprünglich erfand, ist das der für die Informationsübertragung wichtige Problem der Signalverstärkung. Das einfache Prinzip Nachrichten codiert in Einsen und Nullen, in Stromkreis geschlossen oder geöffnet, zu übertragen stößt ohne Verstärkung ab gewisser Distanzen an seine Grenzen. So kam die Idee auf, an die entfernte Seite eines solchen Stromkreises einen Elektromagneten anzuschließen, der einen

Schalter steuert, welcher einen getrennten, zweiten Stromkreis schließen und öffnen kann. Das Relais war geboren.
Der Name entstand in Anlehnung an die sogenannten Relaisstationen der Post, an denen Postreiter ihre Pferde wechseln konnten und gegen frische austauschten. [Sau85, S. 13]
Zur Verbreitung der Relaistechnik trug anfangs der Aufbau von Telegrafienetzen bei. Später, ab dem Ende des 19. Jahrhunderts wurden Relais massiv in der Fernsprechvermittlungstechnik und vor allem für die Teilnehmerselbstwahl verwendet, die ohne Relais nicht möglich gewesen wäre. [Wik16a]
Relais führten 1941 schließlich auch zur Entwicklung des Computers durch Konrad Zuse. Bereits nach einigen Jahren wichen sie allerdings den Elektronenröhren, die ebenfalls durch weitere Technologien verdrängt werden sollten. [Wik16a]

2.3 Darstellung

Die folgende Grafik zeigt Schaltsymbole von Relais verschiedener Typen. Der untere Teil symbolisiert jeweils die Spule und der obere Teil die verschiedenen Schalter.

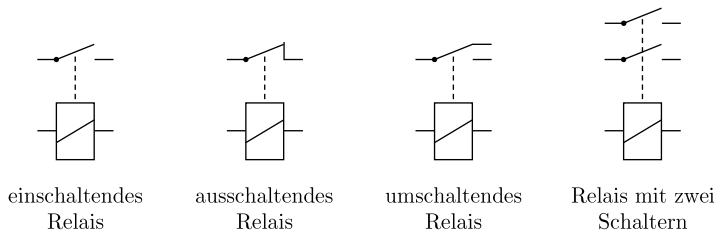


Abbildung 1: Schaltzeichen für Relais

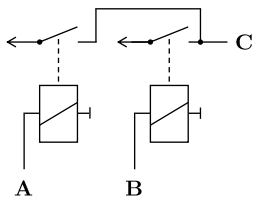
Wie man sieht werden die Schalter immer in der Position gezeichnet, in der das Relais nicht eingeschaltet ist.

2.4 Einfache Schaltungen

Wir wollen nun ein paar einfache Relais-Schaltungen betrachten. Sofern nicht anders angegeben, werden die Eingänge mit Plus für 1 und potentialfrei für 0 betrieben, bzw. Ausgänge liefern kompatibel dazu Plus für 1 und kein Potential für 0.
In den folgenden Abbildungen wird eine an Plus angeschlossene Leitung mit einer Pfeilspitze markiert und eine mit Masse verbundene Leitung mit einem orthogonalen Balken.

Zunächst sehen wir einige Beispiele logischer Schaltungen, die sich wie mathematische Funktionen verhalten. Nach einer gewissen Schaltzeit, die die Relais benötigen, um ggf. ihren Zustand zu wechseln, nehmen hier die Ausgänge Zustände ein, die direkt funktional abhängig von den Eingängen sind. Diese Schaltungen können also keine Zustände speichern. Sie verhalten sich immer unabhängig von ihrer Vorgeschichte.
Anschließend sehen wir Beispiele für sogenannte Flip-Flops, die Schaltungen sind, die Zustände speichern können. Die Möglichkeit, mit Relais Flip-Flops aufbauen zu können und so Zustände zu speichern, ist von essentieller Bedeutung für die Konstruktion komplexerer Rechenmaschinen.

2.4.1 Logisches Oder



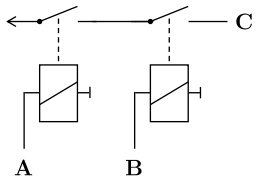
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Abbildung 2: Relaisschaltung für logisches Oder

In dieser Schaltung fließt der Strom bei $A = 1$ über den linken Schalter zum Ausgang *oder* bei $B = 1$ über den rechten und bei $A = 1$ und $B = 1$ über beide. So realisiert die Schaltung die daneben tabellierte Oder-Funktion.

2.4.2 Logisches Und

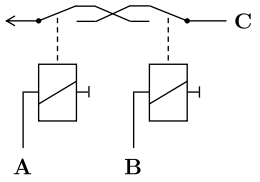
Hier fließt der Strom, wie klar erkennbar ist, nur zum Ausgang, wenn er den einen *und* den anderen Schalter passieren kann:



A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Abbildung 3: Relaisschaltung für logisches Und

2.4.3 Logisches Exklusiv-Oder

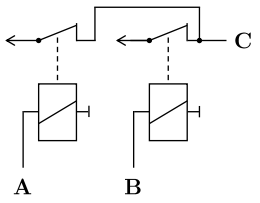


A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Abbildung 4: Relaisschaltung für logisches Exklusiv-Oder

Durch die gekreuzten Leitungen in der Mitte fließt der Strom hier genau dann zum Ausgang, wenn *entweder* der eine Eingang *oder* der andere Eingang einen Zustand von 1 hat, also beide Eingänge einen unterschiedlichen Zustand haben. Auf ähnliche Weise kann auch eine Gleichheitsfunktion geschaltet werden.

2.4.4 Logisches Nicht-Und



A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Abbildung 5: Relaisschaltung für logisches Nicht-Und

Hier fließt der Strom immer dann zum Ausgang, wenn nicht beide Eingänge 1 sind und somit beide Schalter geöffnet sind.

2.4.5 RS Flip-Flop

Die folgende Schaltung bietet die Möglichkeit mit nur einem einzigen Relais einen Zustand zu speichern, der über zwei Steuerleitungen geändert werden kann. Da dies eine sehr elementare Aufgabe ist, macht es Sinn sich dafür auf nur ein Relais zu beschränken.

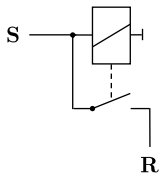


Abbildung 6: Relaisschaltung für RS Flip-Flop

Die Steuerleitung S (für Set) bewirkt das Setzen eines gespeicherten Wertes von 1 und die Steuerleitung R (für Reset) bewirkt das Setzen eines gespeicherten Wertes von 0, der auch der initial gespeicherte Wert ist. Ausgelesen werden kann der gespeicherte Wert ebenfalls über S oder indem ein Relais mit mehreren Schaltkontakten verwendet wird.
Durch das Reduzieren auf nur ein Relais, haben die beiden Steuerleitungen allerdings ein paar Besonderheiten:

1. Für $R = 0$ muss R mit Plus beschaltet werden und für $R = 1$ muss R potentialfrei sein.

2. S ist zugleich Ein- und Ausgang. An S liegt immer Plus, wenn ein positiver Zustand gespeichert ist und kein Potential, falls ein negativer Zustand gespeichert ist.

Beginnen wir mit einem nicht aktivierten Relais, so ist es möglich durch Anlegen einer positiver Spannung an S das Relais direkt zu aktivieren. Das führt dazu, dass das Relais mit seinem Schalter einen Stromkreis schließt, der es dauerhaft mit Strom versorgt, sodass das Relais in aktiviertem Zustand bleibt, auch wenn die externe Spannung von S weggenommen wird.
Um das Relais mit der Leitung R später zu deaktivieren, wird diese potentialfrei geschaltet, sodass am Relais keine Spannung mehr abfällt und es nicht mehr Spannungsversorgt ist und abschaltet. Nun wieder in der Ausgangssituation bleibt das Relais stabil im nicht aktivierten Zustand.

2.4.6 D Flip-Flop

Nun verwenden wir das eben definierte RS Flip-Flop in einer Schaltung, die schon deutlich mehr an eine Speicherzelle erinnert. Die Ein- und Ausgänge arbeiten hier wieder elegant nach ursprünglicher Definition mit Plus für 1 und potentialfreien Leitungen für 0, und die Verwendung ist deutlich komfortabler.

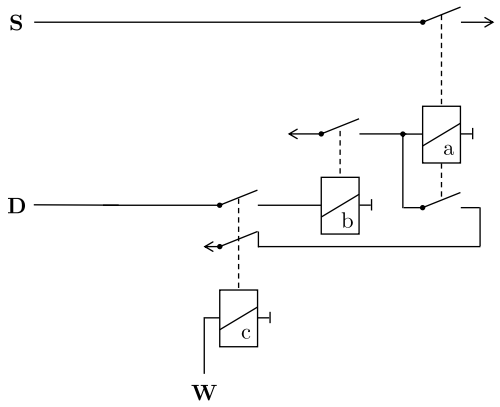


Abbildung 7: Relaisschaltung für D Flip-Flop

Die Schaltung verfügt über einen Ausgang S (hier für State), der permanent das gespeicherte Bit, also 0 oder 1 ausgibt. Außerdem gibt es noch den Eingang D (für Data), der ein zu speicherndes Bit erwartet, also 0, falls eine 0 gespeichert werden soll und 1 falls eine 1 gespeichert werden soll.
Schließlich gibt es noch den Eingang W (für Write). Wann immer W eingeschaltet wird, speichert das Flip-Flop den aktuellen Zustand von D. Es ist darauf zu achten, dass W abgeschaltet wird, bevor eine etwaige 1 von D weggenommen wird, da dies sonst zum Speichern einer 0 führen würde.
Wie schon angekündigt beinhaltet diese Schaltung im Kern das vorgenannte RS Flip-Flop, wie auf der rechten Seite (Relais a) erkennbar ist. $W = 1$ bewirkt hier, dass die Reset Leitung gesetzt ist. Dadurch, dass Relais c bei $W = 1$ aktiviert ist und D den Weg zu b freigibt, nimmt während $W = 1$ das Relais a stets den Wert von D an.
Wird wieder $W = 0$ gesetzt, so öffnet sich der Schalter von c und der Öffner von c schließt sich. Das führt dazu, dass einerseits Relais b beginnt seinen Schalter zu öffnen, falls es bisher durch $D = 1$ aktiviert war und somit bald die Set Leitung des RS Flip-Flops deaktivieren wird und andererseits die Reset Leitung des RS Flip-Flops deaktiviert wird. Wichtig ist, dass Zweiteres vor dem zu erst genannten passiert, um ein ggf. zu speicherndes Bit nicht durch die Reset-Leitung zurückzusetzen.
Da das Öffnen des Schalters von b erst durch ein zweites Relais geschieht, geht dieser Vorgang etwas langsamer von statten als das Freigeben der Reset Leitung des RS Flip-Flops. Da dieser Zeitunterschied aber von essentieller Bedeutung ist, kann es unter Umständen erforderlich sein, das Relais b mit einer Ausschaltverzögerung zu versehen, damit die Schaltung sicher arbeitet. Das geht mechanisch oder durch einen parallel geschalteten Kondensator.

2.4.7 D Flip-Flop mit universeller Datenleitung

Schließlich betrachten wir noch eine Erweiterung der eben beschriebenen Schaltung, bei der die Datenleitung D sowohl als Ein- als auch als Ausgang genutzt werden kann. Neu ist nun ebenfalls der Eingang R (für Read), der bei $R = 1$ den aktuellen Zustand des Flip-Flops auf die Datenleitung ausgibt.

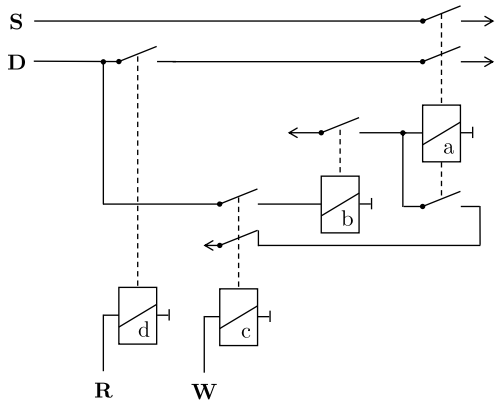


Abbildung 8: Relaisschaltung für D Flip-Flop mit universeller Datenleitung

Bei $R = 1$ ist D ein Ausgang, bei $W = 1$ ist D ein Eingang. Erreicht wird dies durch eine vorgeschaltete Umschaltungsschaltung, die, wie man sich leicht überlegen kann, dazu führt, dass für $R = 1$ die Leitung D wie S geschaltet ist und für $W = 1$ die Leitung D wie in der vorgenannten Version des D Flip-Flops geschaltet ist. Für $R = 0$ und $W = 0$ ist D überhaupt nicht geschaltet.
Das Besondere an dieser Flip-Flop-Schaltung ist, dass sie bereits geeignet ist, über einen Datenbus zu kommunizieren. So ist es denkbar, die D Leitungen beliebig vieler solcher Flip-Flops miteinander zu verbinden und so die Möglichkeit zu erlangen Bits zwischen den Flip-Flops hin und her zu kopieren.
Dazu wird bei dem Flip-Flop, dessen Zustand kopiert werden soll, die R Leitung geschaltet, sodass es seinen Zustand auf das gemeinsame D (den Datenbus) legt. Anschließend wird bei dem Flip-Flop, in das der Zustand kopiert werden soll, die W Leitung geschaltet, sodass es den am gemeinsamen D liegenden Zustand abspeichert.
Für die Simulation der logistischen Maschine werden wir auf genau diese Möglichkeit zurückgreifen.

3 Die logistische Maschine

In den 1930er Jahren kam bei dem deutschen Erfinder Konrad Zuse bei der Konstruktion von Relaiscomputern die Idee auf, wesentliche Teile eines Computers selbst in Software zu realisieren. Seine „logistische Maschine“, ist in diesem Zusammenhang ein Konzept zur Realisierung einer arithmetisch vollständigen arithmetisch logischen Einheit (ALU), mit nur vergleichsweise extrem wenigen Relais. [Roj16b]
Die logistische Maschine ist für sich genommen ein stark spezialisierter Computer, der nur für die Ausführung spezieller Programme gedacht ist, welche die arithmetischen Operationen definieren. Dieser arbeitet Bit-sequentiell, d.h. die zu verarbeitenden Daten werden Bit für Bit, nacheinander verarbeitet.
Da sich Software deutlich günstiger als Hardware reproduzieren und ändern lässt, bot dies ein großes Potential verschiedene Ressourcen zu sparen, wie vor allem Kosten, aber auch Platz, Strom, Abwärme und Gewicht. Wie wir noch sehen werden war das vorliegende Konzept allerdings nicht praktikabel, da die starke Vereinfachung der Hardware sehr lange Programme zur Folge hat, was zu extrem langen Bearbeitungszeiten der Operationen führt.
Als Quelle für die Definition der logistischen Maschine in diesem Kapitel dient in wesentlichem Maße der Artikel „Konrad Zuse's Computer for Binary Logic“[Roj16b].

3.1 Maschinenmodell und Befehlssatz

Das Modell der logistischen Maschine definiert

1. einen in beliebiger, fester Größe gewählten Bit-Speicher mit 1-Bit-Breite, der initial die zu verarbeitenden Daten enthält
2. zwei 1-Bit Register A und B, wobei initial $A = 0$ und $B = 0$
3. ein Flag namens Pr, das durch seinen Zustand stets eines der beiden Register selektiert, wobei $Pr = 0$ Register A selektiert und $Pr = 1$ Register B. Initial ist $Pr = 0$.

Die Logistische Maschine arbeitet ihr Programm sequentiell und kontinuierlich in Dauerschleife ab, beginnt also nach dem Ende immer wieder von vorn. An Befehlen gibt es

1. einen Lade-Befehl (`LOAD`), der ein Bit aus dem Speicher in das selektierte Register lädt und das Pr-Flag anschließend setzt
2. einen Speichern-Befehl (`STORE`), der das Bit aus dem selektierten Register in den Speicher speichert

3. ein paar logische Operationen (`CALC`), die die beiden Register als Parameter verwenden, das Ergebnis in Register A speichern und das Pr-Flag zurücksetzen

4. einen Befehl, der nichts tut (`NOP`).

Die Maschine versteht die Befehle in Form von sogenannten Op-Codes, die aus mehreren Bits bestehen, die t_1 bis t_n heißen. Dabei ist $n \geq 5$ und je nach Speichergröße groß genug, sodass jede Speicherzelle durch eine Adresse der Bit-Länge $n-2$ eindeutig identifiziert werden kann.
Die vier Befehle werden durch t_1 und t_2 codiert, gefolgt von näher beschreibenden weiteren Bits, die im Falle der Lade- und Speichern-Befehle die zu verwendende Adresse im Speicher angeben und im Falle einer logischen Operation diese auswählen, wofür die Bits t_3, t_4 und t_5 benötigt werden.

t_1	t_2	t_3	t_4	t_5	...	t_n	**Befehl**	**Erklärung**
0	0	frei					`NOP`	keine Operation
0	1	0	0	0	frei		`CALC` $\overline{A} \wedge \overline{B}$	logische Operation
0	1	0	0	1	frei		`CALC` $\overline{A} \wedge B$	
0	1	0	1	0	frei		`CALC` $A \wedge \overline{B}$	
0	1	0	1	1	frei		`CALC` $A \wedge B$	
0	1	1	0	0	frei		`CALC` $\overline{A} \vee \overline{B}$	
0	1	1	0	1	frei		`CALC` $\overline{A} \vee B$	
0	1	1	1	0	frei		`CALC` $A \vee \overline{B}$	
0	1	1	1	1	frei		`CALC` $A \vee B$	
1	0	Adresse					`LOAD`	laden
1	1	Adresse					`STORE`	speichern

Tabelle 1: Befehlssatz der logistischen Maschine

3.2 Konkreter Aufbau

3.2.1 Programmnotation und Befehlsdecodierung

Das Programm liegt vor in Form eines Lochbandes, dessen Anfang und Ende miteinander verklebt sind. Untereinander sind dort die Op-Codes der Befehle notiert, wobei die Werte der Bits durch Loch und nicht Loch notiert sind. Dies ermöglicht ein einfaches Lesen des Bandes mittels schleifenden Kontakten, die durch das Band an Stellen ohne Loch voneinander isoliert werden und an Stellen mit Loch Kontakt liefern.
Das Band wird stets nur in eine Richtung bewegt. Durch die Verklebung von Anfang und Ende gelangt die Maschine nach dem Lesen des Endes direkt wieder an den Anfang.

Zur Decodierung sind mit den gelesenen Bits des Op-Codes t_1 bis t_n verschiedene Relais verbunden, die durch ihre Schaltkontakte den Stromfluss an verschiedenen Stellen der ganzen Maschine steuern.

3.2.2 Prozessor

Den Kern der logistischen Maschine bildet die Prozessorschaltung, die in Hardware implementiert die logischen Funktionen berechnen kann, die im Befehlssatz zur Verfügung stehen. Das sind, wie bereits beschrieben, das logische Und und das logische Oder jeweils optional mit in allen Kombinationen invertierbaren Parametern. Die folgende Abbildung zeigt diese Prozessorschaltung.

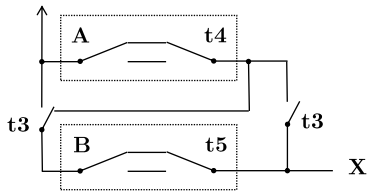


Abbildung 9: Prozessorschaltung der logistischen Maschine

In der Abbildung wurden die Relaisspulen weggelassen und nur deren Schalter dargestellt. Die Spulen sind jeweils so verdrahtet, dass sie den Zustand der Bits, mit denen ihre Schalter beschriftet sind, annehmen.
Dem aufmerksamen Leser ist im Befehlssatz sicher bereits aufgefallen, dass das Bit t_3 angibt, ob es sich um ein logisches Und (bei $t_3 = 0$) oder um ein logisches Oder (bei $t_3 = 1$) handelt. Darauf basiert die vorliegende Schaltung. Die beiden mit t_3 beschrifteten Schalter schalten um, ob die beiden gepunktet markierten Bereiche in Reihe (bei $t_3 = 0$ für Und) oder parallel (bei $t_3 = 1$ für Oder) geschaltet sind.
Die beiden gepunktet markierten Bereiche sind dann wie die Parameter einer so gewählten Und- oder Oder-Funktion, wobei eine durchgehende Leitung 1 und eine unterbrochene Leitung 0 bedeutet.
Wie wir sehen invertiert ein $t_4 = 0$ bzw. ein $t_5 = 0$ so betrachtet den ersten bzw. den zweiten Parameter wie es der Befehlssatz vorsieht. In der dargestellten Situation mit $t_3, t_4, t_5 = 0$ wird so die Funktion $X = \overline{A} \wedge \overline{B}$ mit $A, B = 0$, also $\overline{0} \wedge \overline{0} = 1 \wedge 1 = 1$ realisiert.

3.2.3 Speicher und Adressdecodierung

Die einzelnen Bits des Speichers werden durch Relais vergleichbar mit der bereits erwähnten RS-Flip-Flop-Schaltung gespeichert. Der wesentliche Aufwand in der Speichereinheit fließt in die Adressdecodierung. Diese wird nach einem Prinzip vollzogen, das Konrad Zuse „Tannenbaum“-Schaltung nennt. [Roj16b] Dabei weisen die Bits durch baumförmig angeordnete Umschalter den Weg zur adressierten Speicherzelle, wie wir in der folgenden Abbildung sehen:

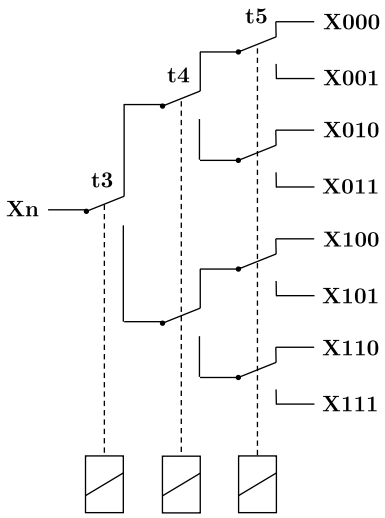


Abbildung 10: Prinzip der Adressdecodierung der logistischen Maschine

Hervorzuheben ist an dieser Stelle, dass die Adressdecodierungsschaltung, sowie die Relais der Speicherzellen das sind, was in Abhängigkeit davon, wie groß die logistische Maschine dimensioniert ist, unterschiedlich aufwändig ist. Ist die Größe des Speichers, also die Anzahl der speicherbaren Bits, n, so wird entsprechend eine Adresslänge von $\lceil \log n \rceil$ Bits benötigt, die zur Decodierung je ein Relais benötigen. In der Summe werden $n - 1$ Schalter an diesen Relais benötigt und die Schaltung der einzelnen Speicherzellen wird natürlich n mal benötigt. Somit liegen die Wachstumsfunktionen der Zahl benötigter Relais und der Zahl benötigter Schalter einer logistischen Maschine in Abhängigkeit von der Speichergröße in $\mathcal{O}(n)$. Das ist auch dann der Fall, wenn für jeden Schalter ein eigenes Relais verwendet wird, weil keine Relais mit mehreren Schaltern zur Verfügung stehen.

3.3 Beispielprogramm: Addieren von zwei zwei-Bit Zahlen

Es soll nun ein Programm definiert werden, sodass die logistische Maschine zwei 2-Bit Zahlen addieren kann. Das Ergebnis ist eine 3-Bit Zahl.
Es soll also gelten $A + B = C$, wobei $A = 2 \cdot A_2 + A_1$, $B = 2 \cdot B_2 + B_1$ und $C = 4 \cdot C_3 + 2 \cdot C_2 + C_1$.

3.3.1 Vorüberlegungen

Wir überlegen uns zunächst ein logisches Flussdiagramm. Mit einem Halb- und einem Volladdierer sieht es so aus:

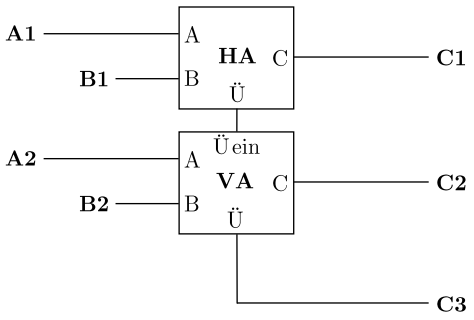


Abbildung 11: Logische Struktur einer 2-Bit-Addition

Dabei kann ein Halbaddierer wie folgt realisiert werden:

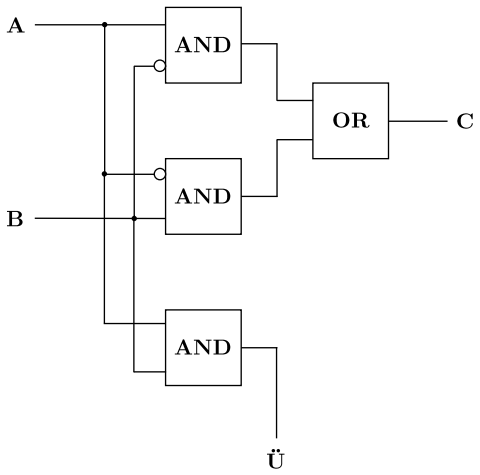


Abbildung 12: Aufbau eines Halbaddierers

Der Volladdierer lässt sich wie hier dargestellt aufbauen:

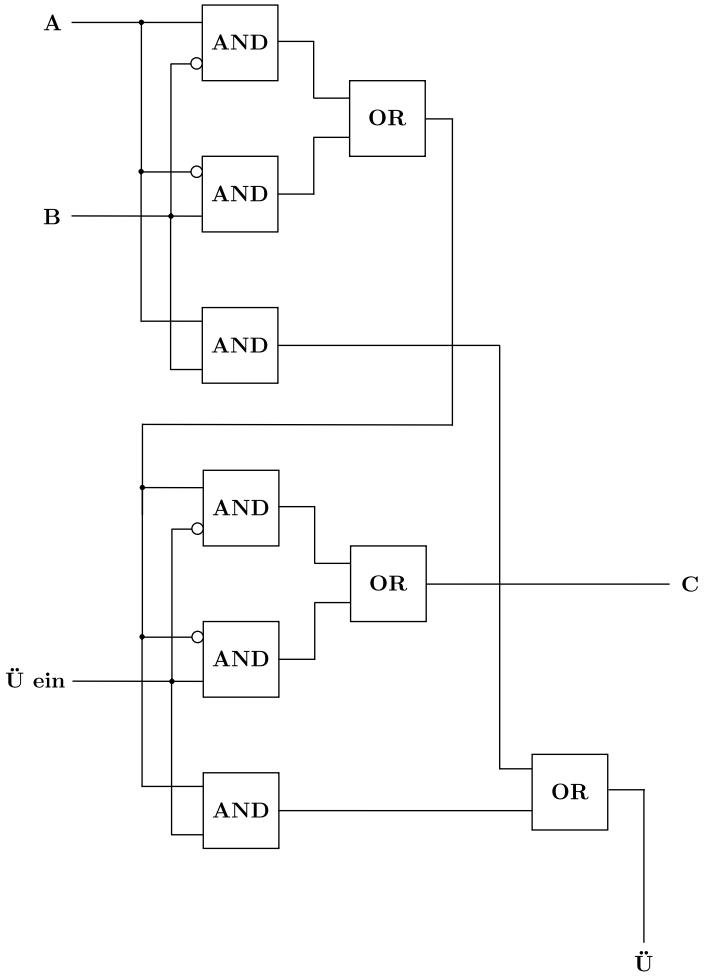


Abbildung 13: Aufbau eines Volladdierers

Das Vorgehen ist nun Schritt für Schritt die Ausgänge der verwendeten Gatter zu berechnen. Dazu bauen wir noch einmal alles zusammen und geben den Gatterausgängen Namen:

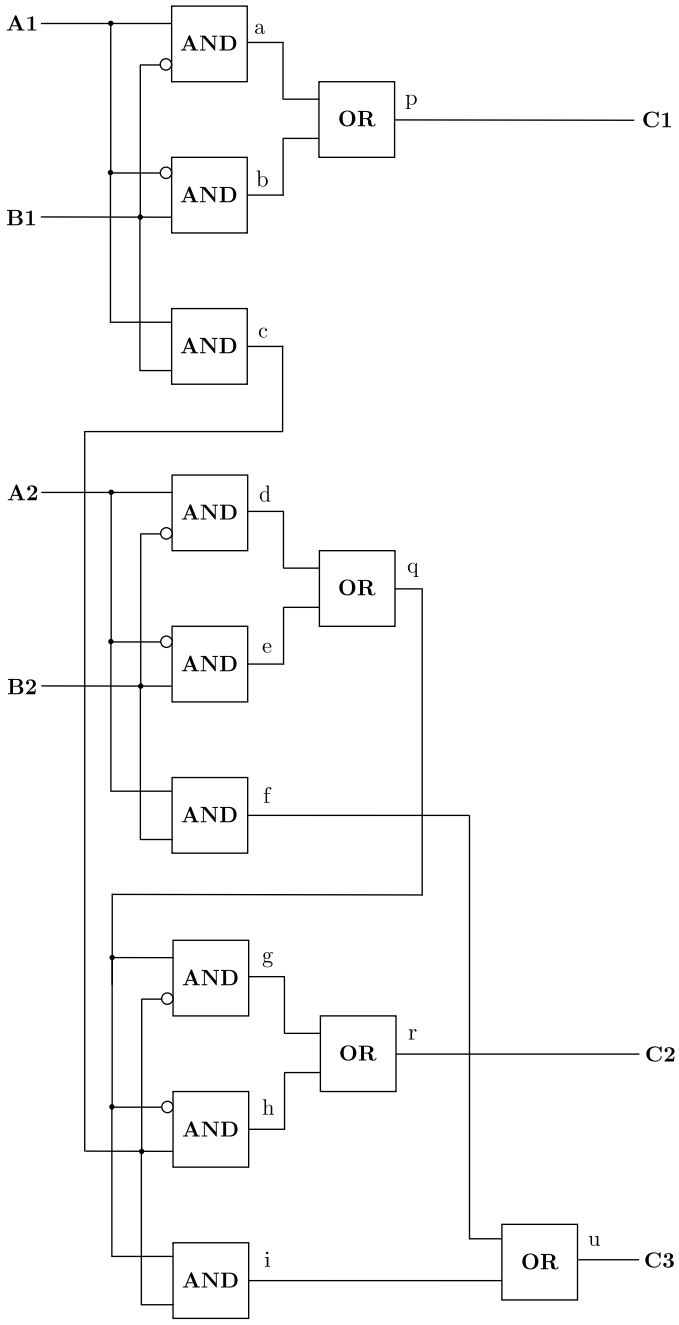


Abbildung 14: Logische Schaltung einer 2-Bit-Addition

3.3.2 Speicherbelegung

Wir wollen das Problem mit einer logistischen Maschine mit einem möglichst kleinen Speicher lösen und wählen daher einen Speicher der Größe 8 Bit. Den Speicher belegen wir wie folgt:

Adresse	**Name**	**Erklärung**
`0`	A_1	A
`1`	A_2	
`2`	B_1	B
`3`	B_2	
`4`	C_1	temporäre Werte, final C
`5`	C_2	
`6`	C_3	
`7`	X	temporäre Werte

Tabelle 2: Speicherbelegung für Beispielprogramm

Wir gehen davon aus, dass die zu addierenden Zahlen initial wie definiert gemäß $A = 2 \cdot A_2 + A_1$ und $B = 2 \cdot B_2 + B_1$ in den ersten vier Bit im Speicher liegen. Nach komplettem Durchlauf des Programms wird das Ergebnis $A + B = C = 4 \cdot C_1 + 2 \cdot C_2 + C_1$ in den folgenden drei Bits im Speicher liegen, die wir nachfolgend auch Ergebnisbits nennen. Das letzte Bit mit der Adresse `7` verwenden wir als temporäre Variable. Um mit einem Speicher von 8 Bit auszukommen müssen wir zwischenzeitlich auch die Ergebnisbits als Zwischenspeicher nutzen. Mit einem größeren Speicher wäre das natürlich nicht erforderlich. Mit diesem Beispielprogramm wollen wir aber zeigen, was bereits eine so kleine logistische Maschine kann und für Demonstrationszwecke ist es schön, wenn nur ein kleiner Speicher benötigt wird.

3.3.3 Implementierung

Um noch möglichst viele Ergebnisbits als Zwischenspeicher zur Verfügung zu haben, ist es sinnvoll mit der Berechnung der Ausgänge der längsten Wege im Diagramm zu beginnen, da diese den meisten Zwischenspeicher benötigen.
Beim Schreiben eines Maschinenprogramms ist es hilfreich eine Tabelle anzulegen, in der das Programm in abstraktere Operationen geordnet werden kann.
Ich verwende hier eine Tabelle, die das fertige Programm in zusammengesetzte Operationen zerlegt, die jeweils wie eine 1-Bit-Variablen-Deklaration mit Zuweisung eines Wertes sind. Dabei werden nur Werte in Formen ver-

wendet, wie sie Ergebnisse von logischen Operationen der logistischen Maschine sind.
Die Tabelle ist Zeile für Zeile von oben nach unten zu lesen. In der Spalte Adresse steht die Adresse, an der künftig der Wert der gedachten Variablen gespeichert werden soll. In der Spalte Name steht ihr Name und unter neuer Wert der Wert, den diese Operation der gedachten Variable, bzw. der Speicherzelle zuweist. Der Wert ist in zwei verschiedenen Notationen angegeben. Links ist der Wert jeweils symbolisch unter Verwendung der bis dorthin definierten Variablennamen notiert, rechts jeweils unter Verwendung der konkreten Speicheradressen, an denen die Variablen liegen.
In der Spalte Befehle schließlich stehen dann die Befehle der logistischen Maschine, die die Operation ausführen. Sie sind immer nach dem gleichen Muster aufgebaut. Erst wird der erste Parameter geladen, dann der zweite, dann wird die logische Operation ausgeführt und schließlich wird das Ergebnis in die entsprechende Speicherzelle zurückgeschrieben.
Als Variablennamen werden die Ausgänge der Gatter des Diagramms aus den Vorüberlegungen verwendet. Ihre Werte entsprechen stets denen der Ausgänge des Diagramms.

Adresse	**Name**	**neuer Wert**		**Befehle**
0	A_1	Ausgangssituation		
1	A_2			
2	B_1			
3	B_2			

4	d	$A_2 \wedge \overline{B_2}$	$*1 \wedge \overline{*3}$	`LOAD 001`
				`LOAD 011`
				`CALC` $A \wedge \overline{B}$
				`STORE 100`

5	e	$\overline{A_2} \wedge B_2$	$\overline{*1} \wedge *3$	`LOAD 001`
				`LOAD 011`
				`CALC` $\overline{A} \wedge B$
				`STORE 101`

7	q	$d \vee e$	$*4 \vee *5$	`LOAD 100`
				`LOAD 101`
				`CALC` $A \vee B$
				`STORE 111`

4	c	$A_1 \wedge B_1$	$*0 \wedge *2$	`LOAD 000`
				`LOAD 010`
				`CALC` $A \wedge B$
				`STORE 100`

5	g	$q \wedge \overline{c}$	$*7 \wedge \overline{*4}$	LOAD 111 LOAD 100 CALC $A \wedge \overline{B}$ STORE 101
6	h	$\overline{q} \wedge c$	$\overline{*7} \wedge 4$	LOAD 111 LOAD 100 CALC $\overline{A} \wedge B$ STORE 110
5	C_2	$g \vee h$	$**5 \vee *6$	LOAD 101 LOAD 110 CALC $A \vee B$ STORE 101
7	i	$q \wedge c$	$*7 \wedge *4$	LOAD 111 LOAD 100 CALC $A \wedge B$ STORE 111
4	f	$A_2 \wedge B_2$	$*1 \wedge *3$	LOAD 001 LOAD 011 CALC $A \wedge B$ STORE 100
6	C_3	$f \vee i$	$*4 \vee *7$	LOAD 100 LOAD 111 CALC $A \vee B$ STORE 110
4	a	$A_1 \wedge \overline{B_1}$	$*0 \wedge \overline{*2}$	LOAD 000 LOAD 010 CALC $A \wedge \overline{B}$ STORE 100
7	b	$\overline{A_1} \wedge B_1$	$\overline{*0} \wedge *2$	LOAD 000 LOAD 010 CALC $\overline{A} \wedge B$ STORE 111
4	C_1	$a \vee b$	$*4 \vee *7$	LOAD 100 LOAD 111 CALC $A \vee B$ STORE 100

Wie wir sehen haben final die Speicherzellen 4 bis 6 ihre Werte C_1 bis C_3 erreicht und die Speicherzellen 0 bis 3 blieben unberührt, sodass die Bits A_1, A_2, B_1 und B_2 wie gefordert ihre Werte behalten haben.

3.4 Bewertung und Einordnung

3.4.1 Praktische Bedeutung

Konrad Zuses Motivation für die Entwicklung seiner logistischen Maschine war vermutlich in erster Linie die Idee, mit ihr eine flexibel programmierbare Universalkernkomponente für seine Relaiscomputer zu schaffen. Zum Beispiel wäre es damit möglich gewesen für jede Grundrechenart ein definierendes Lochband einzubauen, das automatisch gewechselt worden wäre, je nachdem, welche arithmetischen Operationen ein Computer gerade ausführen wollte, in dem die logistische Maschine verbaut gewesen wäre. [Roj16b, S. 8]
Das hätte für die Weiterentwicklung zum Beispiel den Vorteil gehabt, leicht auch andere arithmetische Operationen definieren zu können.
Wie wir bereits gesehen haben, ist allerdings schon die Programmierung einfacher arithmetischer Operationen recht kompliziert und resultiert in langen Programmen, was vor allem lange Ausführungszeiten zur Folge hat und eine Verwendung der logistischen Maschine in einem Relaiscomputer sehr unattraktiv macht.
Ein anderer Punkt an dem man ansetzen kann ist die Frage, ob man durch die Verwendung einer logistischen Maschine gegenüber einer direkten Verdrahtung der arithmetischen Operationen wirklich Relais spart.
Das ist abhängig von der gewünschten Bit-Breite und der Anzahl der geforderten Operationen. Ohne Zweifel punktet hier die logistische Maschine, wenn viele verschiedene arithmetische Operationen zur Verfügung gestellt werden sollen oder große Bit-Breiten gefordert sind, wenn man davon absieht, dass dies die Szenarien sind, in denen sie besonders lange für die Berechnung braucht.
Nehmen wir an, wir möchten zwei 5-Bit-Zahlen addieren. Im Speicher werden dann für die Parameter und das Ergebnis inklusive eines Überlaufbits 16 Bit benötigt. Zuzüglich einiger temporärer Bits würde man etwa 20 Bit Speicher benötigen und je nach konkreter Schaltung für die Adressdecodierung mindestens 20 Relais für die Adressdecodierung zum Schreiben und 20 Relais für die Adressdecodierung zum Lesen, sowie mindestens 20 Relais für die Speicherzellen selbst. Wenn wir noch etwa 20 weitere Relais für alles drum herum ansetzen, benötigen wir also bereits 100 Relais um dies zu realisieren. Das sind etwa doppelt so viele Relais, wie eine direkte Umsetzung dieser Addition als logische Schaltung benötigen würde. Dafür bräuchte man für weitere Operationen allerdings auch keine weiteren Relais.
Für die Praxis war sicher das Geschwindigkeitsargument ausschlaggebend. Da wie wir gesehen haben, bereits die Addition zweier 2-Bit-Zahlen 52 Schritte benötigt, ist die logistische Maschine um Größenordnungen langsamer als eine direkte Verdrahtung.

3.4.2 Theoretische Bedeutung

Die Idee eine Computerhardware einfach zu gestalten ist eng verbunden damit, das zugehörige Maschinenmodell einfach zu gestalten und einfache Maschinenmodelle wecken grundsätzlich theoretisches Interesse. Es ist dem Modell der logistischen Maschine allerdings anzusehen, dass es sehr speziell für die praktische Umsetzung mit Relais gedacht ist.
Im Sinne eines rein theoretischen Modells würde es beispielsweise ausreichen, anstatt der acht Logikoperationen, die das Modell definiert eine einzige Nicht-Und-Operation zu definieren wie in [Roj16a] dargestellt. Insgesamt ist das Modell so betrachtet logischen Flussdiagrammen ähnlich, bietet gegenüber diesen aber keinen theoretischen Mehrwert, da hier der wesentliche Unterschied in der Notation des Programms bzw. des Diagramms liegt und ein Programm einer logistischen Maschine deutlich schlechter lesbar ist, als ein logisches Flussdiagramm, wie im vorgenannten Beispielprogramm nachvollzogen werden kann. Das hängt damit zusammen, dass ein solches Programm nur eindimensional geordnet werden kann, während das logische Flussdiagramm durch einen Graphen dargestellt wird.

4 Visualisierung

Nachdem wir nun das Konzept der logistischen Maschine kennen und einordnen können soll im folgenden Kapitel beschrieben werden, wie ich es für die angestrebte Visualisierung konkretisiert und aufbereitet habe.

4.1 Zielgruppe

Die Simulation soll im Rahmen von Ausstellungen, im Internet und in Präsentationen gezeigt werden können und richtet sich daher auch an ein populärwissenschaftliches Publikum. Vor allem soll sie im wesentlichen innerhalb weniger Minuten erfasst werden können und Interaktion bieten, die aber zum Verständnis nicht erforderlich sein soll.
Ich konzipiere die Darstellung für einen Full-HD-Bildschirm, also für eine Bildschirmauflösung von 1920x1080 Pixeln, da Geräte dieser Auflösung leicht verfügbar sind und so, dass die Simulation ohne Maus und ausschließlich durch Tasten bedient werden kann.
Als Speichergröße wähle ich 8 Bit, da dies zu einer Adresslänge von 3 Bit führt und das zum Beispiel eine gute Tiefe für die Schaltung der Adressdekodierung ist, bei der das Prinzip deutlich wird, aber nicht zu viel Platz benötigt wird. Außerdem lassen sich, wie das Beispielprogramm im vorigen Kapitel zeigt, dafür bereits kleine sinnvolle Programme schreiben.
Als Sprache für Beschriftungen wähle ich Englisch, um das Verständnis möglichst vielen Leuten zu ermöglichen.

4.2 Wahl des Abstraktionsniveaus

Die genaue Funktionsweise eines Computers ist in Darstellungen für Nicht-Experten oft nicht greifbar, weil solche Darstellungen oft sehr abstrakt sind und der Betrachter dann das Gefühl hat, dass sich in den Blackboxen die „wahre Magie“ abspielt, die den Computer zum Rechnen bringt und somit nicht sichtbar ist. Die Einfachheit des Konzepts der logistischen Maschine ermöglicht es aber weitgehend auf starke Abstraktion zu verzichten.
Ich gestalte die Darstellung daher so, dass nur Dinge abstrahiert werden, die sich auch ein Laie gut vorstellen kann, sodass bei keinem Betrachter der Gedanke versteckter, wegabstrahierter Magie aufkommt und die wesentlichen Schritte verstanden werden, wie elektromechanische Bauelemente im Zusammenspiel zum Rechnen erweckt werden können.

4.3 Grobstruktur

Das was der Betrachter zuerst sieht, ist die grobe Struktur der Darstellung. Hier bietet es sich an, die Maschine in Sektionen zu unterteilen, die an heutige Computerarchitekturen erinnern und somit bei dem Betrachter Assoziationen mit bekannten Teilen von Computern hervorrufen, der somit

gleich erkennt, dass es sich um die Simulation eines Computers handelt. Teilweise ist die Teilung in verschiedene Abschnitte sowieso bereits der Fall. In der folgenden Abbildung habe ich nun sechs solche Sektionen angeordnet, für die ich mich entschieden habe.

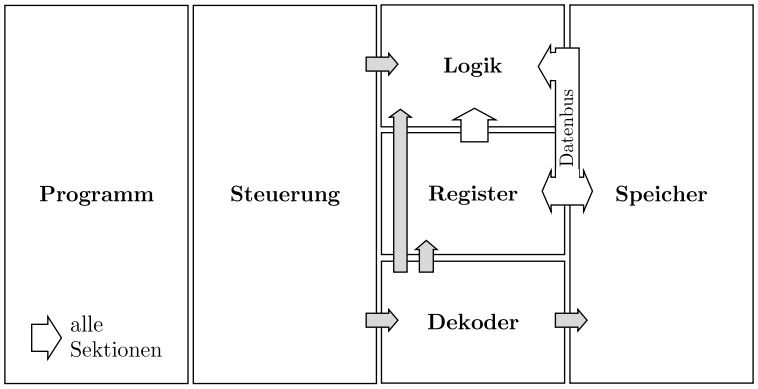


Abbildung 15: Aufteilung der logistischen Maschine in sechs Sektionen

Die weißen Pfeile in der Abbildung geben den direkten Informationsfluss zwischen den Sektionen an und die grauen Pfeile zeigen, welche Einheiten welche steuern.
Die Sektionen haben folgende Aufgaben:

- Die **Programm**-Sektion enthält das Lochband, auf dem das Programm notiert ist und die fünf schleifenden Kontakte zum Lesen der Op-Codes.
- Die **Steuerung** steuert die anderen Sektionen zu den richtigen Zeiten an und koordiniert so alle Abläufe.
- Die **Logik**-Sektion entspricht dem, was wir im letzten Kapitel Prozessor genannt haben.
- In der Sektion **Register** sind die Register, sowie das Pr-Flag untergebracht.
- Der **Dekoder** dekodiert die Bits t_1 und t_2 des Op-Codes, sodass ein gemeinsamer Eingang auf einen von vier Ausgängen für den jeweiligen Befehl durchgeschaltet werden kann.
- Die **Speicher**-Sektion enthält den Speicher sowie zwei Adressdekodierungsschaltungen, je eine für lesen und eine für schreiben.

Die Sektionen sind so angeordnet, dass das spätere Schaltungsnetz möglichst planar gezeichnet werden kann und Sektionen möglichst nur mit den zu ihnen benachbarten Sektionen kommunizieren müssen. In der Darstellung später kann dadurch die Beschriftung und Abgrenzung der Sektionen untereinander im Hintergrund erfolgen und die Gesamtschaltung darüber gelegt werden. Das unterstützt das Gefühl des Betrachters alles greifen zu können und nicht nicht-lokale Verknüpfungen im Auge behalten zu müssen.
Der einzige Bruch mit eben genannter Idee ist die Verknüpfung des in der Programmsektion gelesenen Op-Codes mit dem Rest. Hier lässt sich die Struktur nicht sinnvoll so gestalten, dass nur mit den Nachbarsektionen kommuniziert wird, da die Bits des Op-Codes verstreut in allen Sektionen verwendet werden müssen.
Außerdem habe ich bei der Anordnung der Sektionen darauf geachtet, dass das Gesamtbild sinnvoll zu verstehen ist, wenn man, wie man es vom Lesen von Texten gewöhnt ist, mit der Betrachtung oben links beginnt und sich dann von links nach rechts und von oben nach unten bewegt.

4.4 Darstellung der Leitungen

Über die Sektionen darübergelegt soll also die Schaltung dargestellt werden. Hier verwende ich die übliche Notation, die den meisten Betrachtern auch aus dem Physikunterricht bekannt sein wird. Erweitert wird sie durch die Darstellung von High-Pegeln.
Zunächst wird dazu die Schaltung normal in schwarzer Farbe dargestellt. Leitungen, die einen High-Pegel haben, werden rot dargestellt. Leitungen, die einen Low-Pegel haben, könnten in blau dargestellt werden. Das ist allerdings für die hier vorgestellte Schaltung nicht erforderlich, könnte aber interessant sein, falls andere Schaltungen im gleichen Zusammenhang gezeigt werden sollen.

4.5 Das D Flip-Flop Element

Die logistische Maschine benötigt für die Register und den Speicher die Möglichkeit Bits zu speichern. Das ist die einzige Stelle, an der ich in der Visualisierung eine Blackbox verwenden möchte. Das ist unproblematisch, weil das Speichern eines einzelnen Bits eine Aufgabe ist, die sich auch ein Laie sehr einfach vorstellen kann bzw. dahinter keine großen Kunststücke vermuten wird.
Zu konzipieren ist also ein Element bzw. eine Schaltung zur Speicherung eines Bits, das als Element in der Simulation verwendet und dargestellt werden kann. Es muss folgende Kriterien erfüllen:

1. Das Element muss sich einfach in die restliche Schaltung integrieren lassen und darf keine komplizierte Ansteuerung erforderlich machen, die die restliche Darstellung verkomplizieren würde.

2. Die Definition der Ein- und Ausgänge muss mit der grafischen Darstellung insofern korrespondieren, dass bei der Beobachtung in Aktion schnell deutlich wird wie deren Funktion und Bedeutung ist.

3. Das Element muss durch eine einfache Schaltung realisierbar sein, damit es realistisch ist eine logistische Maschine damit zu bauen.

Ich entscheide mich für eine Form eines D Flip-Flops, mit der universellen Datenleitung D für Data, den Steuerleitungen R und W für Read und Write, die zur Speicherung des Zustandes der Datenleitung führen bzw. den aktuell gespeicherten Wert lesen und auf die Datenleitung ausgeben und einer Leitung S, die ständig den gespeicherten Zustand ausgibt. Eine Schaltung für dieses D Flip-Flop habe ich bereits im Grundlagenkapitel erläutert.
Wie dort auch bereits dargestellt ermöglicht es diese Definition bereits über eine Art Datenbus zu kommunizieren, da die D-Leitungen mehrerer solcher Flip-Flops einfach verbunden werden können und eine Kommunikation komplett durch die Steuerleitungen gesteuert werden kann. Gleichzeitig passt eine D-Leitung, die Zustände senden und empfangen kann, gut in die restliche verwendete Relaislogik, da an den Ein- und Ausgängen 1 und 0 ebenfalls auf die gleiche Weise repräsentiert wird.
Zur grafischen Darstellung habe ich folgende Symbolik entwickelt, die folgend in der Situation, wenn nichts spezielles passiert, und in den Varianten eines Lese- und eines Schreibvorganges dargestellt ist.

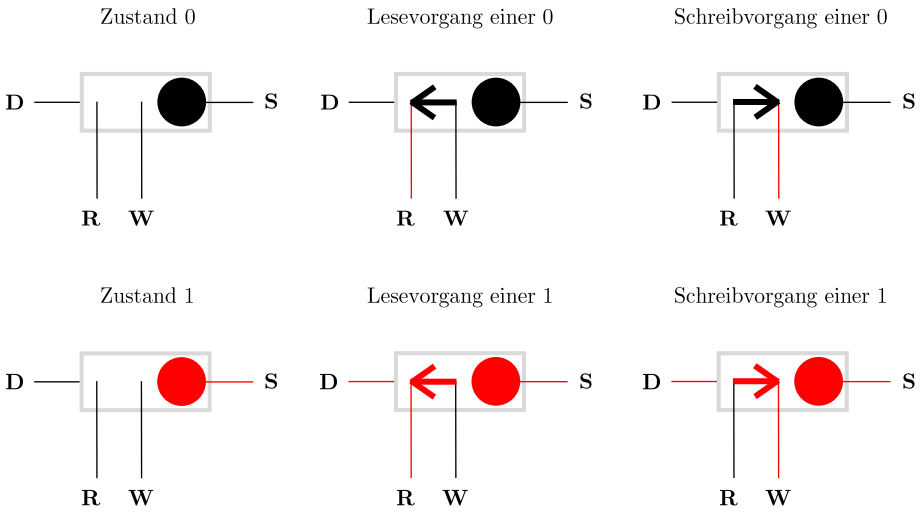


Abbildung 16: Darstellung des D Flip-Flop Elements

Als Symbolik für das gerade gespeicherte Bit dient der entsprechend für 0 schwarz oder für 1 rot eingefärbte Kreis, der an eine Lampe erinnern soll. Die Leitung S erklärt sich dadurch relativ gut, da hier quasi die Farbe der Lampe barrierelos herausfließen kann.
Auf der linken Seite ist das anders. Dort geht die Leitung D nicht direkt bis zur Lampe durch, sondern ist durch eine Lücke von ihr getrennt. Strom an den Steuerleitungen bewirkt, dass diese Lücke durch einen Pfeil gefüllt wird, der in Richtung der gesetzten Steuerleitung zeigt. Die Steuerleitung für lesen R ist daher auf der linken Seite, sodass in diesem Fall der Pfeil in Flussrichtung von der Lampe zum in diesem Fall Ausgang D zeigt. Die Steuerleitung für schreiben W ist entsprechend auf der rechten Seite, sodass hier der Pfeil in Flussrichtung vom dann Eingang D zur Lampe zeigt.

4.6 Datenbus

Die Befehle der logistischen Maschine führen jeweils dazu, dass eine der drei Sektionen Prozessor, Register und Speicher ein Bit zu einer anderen der drei schickt. Das legt die Idee nahe, diese drei Sektionen über eine gemeinsame Datenleitung, also einen Datenbus, miteinander zu verbinden, an die die sendende Sektion sendet und von der die empfangende Sektion empfängt. Folgende Tabelle zeigt die Befehle und welche Sektion jeweils auf den Datenbus senden und welche Sektion vom Datenbus empfangen muss.

t_1	t_2	**Befehl**	**Sender**	**Empfänger**
0	0	`NOP`		
0	1	`CALC`	Logik	Register
1	0	`LOAD`	Speicher	Register
1	1	`STORE`	Register	Speicher

Tabelle 4: Befehle und Kommunikation auf dem Datenbus

Später müssen die Sektionen durch Steuerleitungen zum Senden und Empfangen gebracht werden. Für die Logik-Sektion beispielsweise lässt sich das mit Relais einfach realisieren, da sie nur in einem Befehl vorkommt. Beim Speicher ist es auch kein Problem, da hier t_1 darüber entscheiden kann, ob der Speicher aktiv ist und t_2 darüber, ob er senden oder empfangen soll.
Betrachtet man die Register-Sektion, so ist es schwieriger und nicht durch eine extra Schaltung möglich. Die Entscheidung über die Aktivität könnte hier durch eine Schaltung $t_1 \vee t_2$, die sehr einfach ist getroffen werden. Die Unterscheidung zwischen senden und empfangen erfordert aber etwas mehr Logik, zum Beispiel $t_1 \wedge t_2$, die die Darstellung stören würde.
Ich habe mich daher dafür entschieden für die Simulation die Codierung der Befehle `LOAD` und `STORE` zu vertauschen, was zu folgender neuer Codierung führt:

t_1	t_2	**Befehl**	**Sender**	**Empfänger**
0	0	`NOP`		
0	1	`CALC`	Logik	Register
1	0	`STORE`	Register	Speicher
1	1	`LOAD`	Speicher	Register

Tabelle 5: Geänderte Befehlscodierung

Die Funktion der Entscheidung über die Aktivität von Registern und Speichern ändert sich dadurch nicht. Die Entscheidung über senden und empfangen des Speichers invertiert sich dadurch bloß und die Entscheidung über senden und empfangen der Register lässt sich nun direkt anhand von t_2 treffen.

4.7 Phasen

Da zur Abarbeitung eines Befehls mehrere Schritte erforderlich sind durchschreitet die Maschine immer wieder die Phasen:

- **Lesen eines Befehls**
  Hier wird das Lochband einen Befehl weiter geschoben und die an die Bits des Op-Codes angeschlossenen Relais nehmen ihre neuen Zustände an.
- **Laden der Register**
  In dieser Phase werden die Zustände der Register in die Logik-Sektion geladen. Das ist nur für den Befehl `CALC` notwendig.
- **Ausführung**
  In der Ausführungsphase kommunizieren die beiden am Befehl beteiligten Sektionen, wie im letzten Abschnitt beschrieben über den Datenbus. Es werden dabei die Zustände der durch den Befehl zu ändernden Flip-Flops geändert.
- **Anpassung des *Pr*-Flags**
  Nach Spezifikation wird in dieser Phase das Pr-Flag entsprechend angepasst.

5 Implementierung

Schließlich kommen wir zur Implementierung des nun entwickelten Visualisierungskonzeptes.

5.1 Wahl der Programmiersprache

Da die Simulation aus vielen gleichartigen Komponenten zusammengesetzt ist und einige Berechnungen durchgeführt werden müssen ist die Wahl einer objektorientierten Programmiersprache von Vorteil. Obwohl die Simulation auch im Internet gezeigt werden können soll ist die Programmierung mit Javascript und einem reinen Webinterface also nicht sinnvoll. Ich entscheide mich daher für die Implementierung in Java, strukturiere das Programm nach dem Model-View-Controller Ansatz und gestalte die Oberfläche so, dass sie mit einem einzigen Fenster auskommt, sodass es später auch gut als Java-Applet präsentiert werden kann.

5.2 Grafik

Die Grafik zeichne ich ohne weitere Bibliothek direkt mit Methoden der Klasse `Graphics`. Da sich die Darstellung in der Regel oft ändert verzichte ich auf Mechanismen, die den View bei Änderung des Models benachrichtigen und zeichne mit fester Framerate doppelt gepuffert kontinuierlich alles neu.

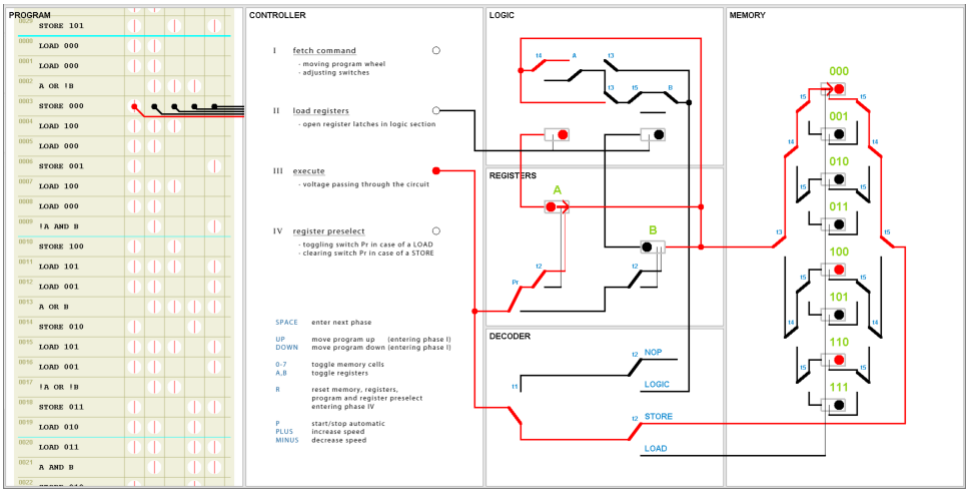


Abbildung 17: Die Simulation in Aktion

5.3 Simulation der Schaltung

Ziel ist es den Code zur Berechnung der Relaisschaltung wiederverwendbar zu gestalten, sodass er unabhängig von der konkreten Schaltung funktioniert. Dazu möchte ich trennen zwischen der Modellierung der einzelnen Elemente der Schaltung und ihrer Simulation.
Als Schaltungselemente werden grundlegend lediglich benötigt:

1. **Spannungsquelle** (Klasse `esim.parts.Source`)
2. **Leitung** (Klasse `esim.parts.Wire`)
3. **Verbindungsstück zwischen Elementen** (Klasse `esim.Connection`)

Alle anderen Elemente lassen sich daraus zusammensetzen:

- Die **Schalter** (Klasse `esim.parts.Switch`) werden durch je nach dem mehrere Leitungen implementiert, deren Verbindung umgeschaltet wird.
- Das **D-Flip-Flop-Element** (Klasse `esim.parts.Register`) ist eine Kombination aus Leitungen, Verbindungen, die umgeschaltet werden und einer Spannungsquelle.
- Die **schleifenden Kontakte** der Programm-Sektion sind Leitungen, die mit einer Spannungsquelle verbunden oder getrennt werden.

Benötigt wird nun ein Berechnungsmodell, das folgende Kriterien erfüllt:

1. Jeder Leitung muss ein Potential zugeordnet werden oder die Information, dass sie potentialfrei ist, damit die Leitungen korrekt eingefärbt werden können.
2. Die Berechnung muss unabhängig von Informationen über Änderungen von Modelldaten laufen und stets aus einem Zustand den nächsten Zustand berechnen können.
3. Gut wäre, wenn es Parallelisierung grundsätzlich offen lässt.

Ich entscheide mich dafür, für Spannungsquellen und Leitungen eine gemeinsame Oberklasse für solche Kapazitäten (`esim.Capacitor`) zu definieren, die das Potential des Elements speichert und ein sogenanntes Gewicht. Werden zwei Elemente mit einander verbunden, wie wenn man sie zusammen löten würde, so muss dafür ein Verbindungsstück erzeugt werden (Klasse `esim.Connection`), das eine Liste der Elemente enthält, die es verbindet, der die Elemente hinzugefügt werden müssen.
Die Idee zur Berechnung der Simulation ist nun, dass die Verbindungsstücke kontinuierlich gewichtete Mittelwerte der Potentiale ihrer „Mitglieder“unter

Verwendung der oben genannten Gewichte bilden und diesen allen Elementen als neues Potential zuweisen. Außerdem wird das maximal beteiligte Gewicht ermittelt und allen als neues Gewicht zugewiesen.
Die Spannungsquelle hat die Besonderheit, das hier die Methoden zur Änderung des Potentials und des Gewichts überschrieben sind, sodass sie ihr Potential und Gewicht dauerhaft behält.
Alle Elemente beginnen mit Gewicht 0, außer Spannungsquellen, die mit maximalem Gewicht und dem für sie gewählten Potential beginnen. Wir sehen, dass sich Potential und Gewichte Schritt für Schritt auf alle verbundenen Leitungen ausbreiten.
Damit sich auch eine Potentialfreiheit ausbreiten kann wird noch eine zweite Simulationsphase definiert, in der die Gewichte aller Elemente außer der Spannungsquellen leicht verkleinert werden. Dadurch sinkt das Gewicht der Leitungen, die mit keinem gewichteten Potential mehr verbunden sind Stück für Stück.
Das beschriebene Verfahren hat den Vorteil, dass die einzelnen Operationen sich jeweils nur ganz lokal auf ein Element und seine direkten Nachbarn beziehen und es nicht auf die Reihenfolge ankommt, in der sie ausgeführt werden. Es kommt lediglich darauf an, dass die Operation des Verringerns der Gewichte weniger oft ausgeführt wird, als die zu erst genannte Operation, damit sich die Potentiale überhaupt ausbreiten können.

6 Diskussion und Ausblick

In dieser Arbeit wurde eine Simulation Konrad Zuses logistischer Maschine entwickelt, die visuell aufbereitet ist, um durch Laien verstanden zu werden. Die dazu entwickelten Visualisierungskonzepte und der Rahmen der Implementierung sind darüber hinaus geeignet, auch andere einfache Relaisschaltungen zu simulieren.
Zur Demonstration wurde ein Programm für die logistische Maschine entwickelt, das zwei 2-Bit-Zahlen addiert. Der Aufbau dieses Programms wurde ebenfalls umfangreich erklärt und visualisiert. Es wäre möglich, aus diesen Diagrammen eine zusätzliche Ansicht für die Simulation zu erstellen, um auch die Funktionsweise des Beispielprogramms zu visualisieren. Das könnte parallel zur logistischen Maschine angezeigt werden und für den Betrachter den Mehrwert bringen, nicht nur die Funktionsweise einer einfachsten Computerhardware bis zur Programmierbarkeit zu verstehen, sondern darüber hinaus noch einen Schritt weiter bis zum Verständnis der Software zu gelangen. Er hätte dann komplett verstanden, wie eine Software ein Problem löst.
Eine andere Idee wäre eine Möglichkeit für den Betrachter auch selbst Programme eingeben zu können. Das für sich genommen wäre nicht weiter schwierig, aber da es wie beschrieben nicht ganz einfach ist Programme für die logistische Maschine zu schreiben, müsste das umfangreich erklärt werden. Eine Alternative wäre ein einfacher Editor für logische Flussdiagramme, in denen man Und- und Oder-Gatter zusammenklicken kann, die dann automatisch kompiliert werden. Das mag für eine Präsentation im Internet geeignet sein, die sich an Personen richtet, die explizit nach einer Simulation der logistischen Maschine suchen. Für eine Ausstellung oder eine Präsentation im Rahmen eines Vortrages würde das aber natürlich den Rahmen sprengen.
Gut geeignet ist die Simulation aber auch für den Schulunterricht, da zum Verständnis wie beschrieben keine besonderen Kenntnisse erforderlich sind.

Tabellenverzeichnis

Abbildungsverzeichnis

Literatur

[Roj16a] Raúl Rojas. Babbage meets zuse: A minimal mechanical computer. In *Unconventional Computation and Natural Computation*, Lecture Notes in Computer Science, pages 25 – 27. Springer, 2016.

[Roj16b] Raúl Rojas. Konrad zuse's computer for binary logic. Freie Universität Berlin, Dept. of Mathematics and CS, Mai 2016.

[Roj16c] Raúl Rojas. Zuse's "logistische maschine". In *Unconventional Computation and Natural Computation*, Lecture Notes in Computer Science, pages 27 – 34. Springer, 2016.

[Sau85] Hans Sauer. *Relais-Lexikon*. Hüthing, 1985.

[Wik16a] Wikipedia. Relais. In *Wikipedia, die freie Enzyklopädie (deutsche Version)*, https://de.wikipedia.org/wiki/Relais, Abgerufen am 11. Juni 2016.

[Wik16b] Wikipedia. Relais. In *Wikipedia, The Free Encyclopedia (englische Version)*, https://en.wikipedia.org/wiki/Relay, Abgerufen am 11. Juni 2016.