Technische Informatik
SS 19

Clemens Mittermaier

Sommersemester 2019

Inhaltsverzeichnis

Teil I.
Theorie

1. Grundlagen

1.1. Grundlagen der Mathematik

Fangen wir bei 0 an. Um die Boolsche Algebra zu beschreiben brauchen wir zu erst Mengen.

Mengen sind mehrere Einzelne Elemente zu einer Gruppe zusammengefasst. Dabei gibt es für Mengen kaum Begrenzungen. Es gibt leere Mengen und auch Mengen, die andere Mengen als Element enthalten. Der Fantasie sind hier wirklich keine Grenzen gesetzt.
Ok Jetzt können wir damit anfangen auf diese Mengen Relationen anzuwenden.

Relationen sind die Schnittmengen zweier Mengen mit gewissen Eigenschaften.

Funktionen sind injektive Relationen zwischen zwei Menge $f : X \to Y$.

1.2. Boolsche Algebra

Nach dem wir nun eine Funktion definiert haben widmen wir uns nun dem Hauptbestandteil der Mathematik in diesem Fach: Der **Boolschen Algebra**. Diese Art hat ein paar besonderheiten. Zum einen gibt es nur zwei Elemente in diesem Alphabet: {0&1}. Auf diese Menge gibt es drei Operatoren: die **Konjunktion**, die **Disjunktion** und die **Negation**

Konjuktion Eine Konjunktion ist eine UND-Verknüpfung. ist das Ergebnis der Funktion nur dann 1, wenn beide Inputs auch 1 sind. Das Symbol ist $\wedge$, kann auch mit einem + geschrieben werden.

x	y	f(x,y)
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 1: Wertetabelle einer UND-Funktion

Disjunktion ist eine ODER-Funktion. Das Zeichen dafür ist ein $\vee$ oder ein $*$.

x	y	f(x,y)
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 2: Wertetabelle einer ODER-Funktion

Negation ist eine Negierung. Hier werden die Werte einfach Umgedreht. Wird auch als NOR bezeichnet. Das Symbol ist $\neg$.

x	f(x,y)
0	1
1	0

Tabelle 3: Wertetabelle einer NOT-Funktion

1.3. Axiome der boolschen Algebra

In der Boolschen Algebra gibt es ein paar Axiome, die immer gelten. Die Beweise gibt es irgendwo im Internet und in meinen Aufzeichnungen.

- Komutativität $x + y = y + x$ und $x * y = y * x$
- Assoziativität $x + (y + z) = (x + y) + z$ und $x * (y * z) = (x * y) * z$
- Absobtion $x + (x * y) = x$ und $x * (x + y) = x$
- Distributivität $x + (y * z) = (x + y) * (x + z)$ und $x * (y + z) = (x * y) + (x * z)$
- Komplement $x + (y * \neg y) = x$ und $x * (y + \neg y) = x$
- Doppeltes Komplement $\neg(\neg x) = x$
- Idempotenz $x + x = x * x = x$
- De-Morgan-Regel $\neg(x + y) = (\neg x) * (\neg y)$ und $\neg(x * y) = (\neg x) + (\neg y)$

1.4. Graphen

Ein gerichteter Graph $G = (V, E)$ (nicht mit f geschrieben) ist eine Funktion mit zwei Komponenten:

1. V ist eine endliche, nichtleere Menge. (Knoten)
2. E endliche Menge (Kanten)

Dabei sind folgende Abbildungen wichtig: $Q : E \to V$ also stammt aus genau einem Knoten und $Z : E \to V$ jede Kante zeigt auf genau einen Knoten. Mehrere Knoten können, wenn sie keinen Kreisschluss haben auch einen Baum bilden. Ein Baum ist dann besonders, wenn er ausschließlich aus Knoten besteht, die genau eine Eingangskante und höchstens zwei Ausgangskanten haben. Das nennt man einen **Binärbaum**.

2. Datenkodierung

2.1. Kodierung von Zeichen

Ok, jetzt geht es richtig los. Ein Computer muss zuverlässig und störungsfrei in der Lage sein Algorithmen richtig und konstant richtig auszuführen. Das bedeutet, dass sowohl der Speicher als auch die Verarbeitung so wenig störungsanfällig wie möglich sein muss. Ein Computer kann natürlich viele verschiedene Dinge verarbeiten, aber um sie zu verstehen müssen alle diese Dinge in 1 und 0 herunter gebrochen werden. Die Dinge müssen in einer Folge von Bits repräsentiert werden. Und damit diese Bitreihe eindeutig ist müssen wir uns mit Kodierung beschäftigen.

Alphabete und Wörter Eine nicht leere Menge $A = \{a_1, \ldots, a_m\}$ wird als Alphabet der Größe m bezeichnet. Dabei sind a_1 bis a_m alles Buchstaben. Ein Wort wird mit der Funktion $A = (wörter = b_1...b_n mit n \in N, für alle möglichen Buchstaben in A)$. Die Anzahl der Buchstaben ist die Länge des Wortes. Ich denke das muss ich nicht sagen, aber ich schreibe es halt mal auf.
Eine Abbildung $c : A \to \{0, 1\}$ heißt **Code** falls c inhektiv ist.

Präfixcode Der Huffman Code versucht eine Nachricht so effektiv wie möglich zu verschlüsseln. Dabei bekommen häufig genutzte Buchstaben einen kurzen Code und selten genutzte einen längernen. Er lässt sich dabei am besten mit einer Häufigkeitsanalyse der benutzen Sprache knacken. Ein Präfixcode verwendet dabei einen Binärbaum, der verhindert, dass ein kurzes Präfix zu einem Blatt und gleichzeitig das Präfix für mindestens ein anderes Blatt ist.

2.2. Kodierung von Zahlen

Ein Zahlensystem ist ein Tripel $S = (b, Z, \delta)$ mit folgenden Eigenschaften:

- $b >= 2$ ist die Basis des Stellenwertsystems
- Z ist eine b-elementige Menge von Symbolen, den Ziffern
- $\delta : Z \to \{0, 1, \ldots, b - 1\}$ ordnet jeder Ziffer eine eindeutige natürliche Zahl zu.

Festkommazahl ist eine endliche Folge von Ziffern asu einem Zahlensastem zur Basis b mit Ziffernmenge Z. Sie bestehen aus n Vorkommastellen und k Nachkommastellen.

Komplemente werden erzeugt, indem bei einer Zahl a alle Bits invertiert werden a'. Addiert man nun $a + a'$ so muss $a + a' = [111 \ldots 111] = 0$ sein. Das Problem, dass sich bei dieser Art von Komplement (Wird auch als Einer-Komplement bezeichnet) ergibt, ist, dass die 0 nun zwei werte hat, denn das Komplement von 00000 = 11111 und damit kann es zu Problemen im Programm kommen. Die Lösung für dieses Problem ist das sogenannte **Zweier-Komplement**. Dabei wird nach der Inversion zu den nun negativen Ergebnissen noch ein weiterer Wert abgezogen.dadurch ist nun 111 der niedrigste Wert des Zahlensystems mit (indiesem Fall 3) n Stellen.

3. Kombinatorische Logik

Kombinatorische Logik ist ein Modell für Hardware, die eine boolsche Funktion $f : B^n \to b^m$ implementiert. Das bedeutet, dass es eine Schaltung gibt, die bei einer bestimmten vorraussetzung ein bestimmtes Ergebnis in der bbolschen Algebra liefern kann. Die kombinatorische Logiksynthese ist das Problem zu einer gegebenen boolschen Funktion eine möglichst effiziente

Schaltung zu finden. Die Kosten hängen von der verwendeten Technologie ab und können sich auf Größe, Verzögerung, Energieverbrauch, ect beziehen. Dabei verwenden wir hier zwei Arten von Technologien:

- PLAs
- Mehrstufige Realisierung mit allgemeinen Bibloithekszellen.

3.1. Logikgatter

Logikgatter sind kleine kombinatorische Schaltblöcke, die bis zu vier Eingänge und einen Ausgang haben. Diese Gatter werden mit Transistoren realisiert. Diese Gatter werden dann wieder zu größeren und ausgefalleneren Schaltungen zusammengeschaltet werden. Es gibt 6 Standart-Logikgatter.

Inverter Der Inverter ist die einfachste logische Schaltung zu erklären. Er hat nur einen Anschluss und nur einen Ausgang und er dreht das Signal um. Also aus einer 1 wird eine 0 und aus einer 0 eine 1.

AND Das UND-Gatter schaltet nur dann, wenn an den beiden Eingängen jeweils eine logische 1 anliegt.

OR Das ODER-Gatter schaltet eine 1, wenn an einem der beiden Eingänge eine 1 anliegt.

NAND & NOR NAND- und NOR-Gatter sind Gatter mit einem Inverter an den Ausgang geschalten.

XOR Diese Gatter ist wie ein ODER-Gatter, nur mit dem Unterschied, dass wenn an beiden Eingängen eine 1 angliegt, wieder eine 0 Ausgegeben wird.

A	**B**	**AND**	**OR**	**NOT**	**NAND**	**NOR**	**XOR**
0	0	0	0	1	1	1	0
0	1	0	1	1	1	0	1
1	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0

Tabelle 4: Die Wertetabellen der 6 Standartlogikgatter

3.2. Transistoren

Ein Transistor ist ein winziger elektrischer Schalter, der in der Lage ist einen Strom an und wieder auszuschalten. es gibt p-Kanal Transistoren und es gibt n-Kanal Transistoren. Ein p-Kanal leitet, wenn an g eine 0 anliegt sperrt bei einer logischen 1. der n-Kanal Transistor macht das Gegenteil. Indem diese beiden Transistoren richtig verschaltet, kann man die verschiedenen Logik-Gatter erstellen. Darüber hinaus möchte ich hier festhalten, dass es keine direkte Implementation von AND- und OR-Gattern mit CMOS-Transistoren gibt. Es gibt nur NAND und NOR, die jeweils in einen Inverter laufen.

Schaltkreise lassen sich mit einer Bibloithek SK darstellen. Dabei gilt die Definition: $SK = (X_n, G, typ, IN, Y_m)$. Diese Formel definiert einen Schlatkreis mit n Eingängen und mit m

Ausgängen. Dabei sind sowohl X als auch Y als Vektoren zu betrachten. G ist der azyklische, Gerichtete Graph der zwischen dem Ein- und dem Ausgang liegt, $typ : I \to BIB$ ordnet jedem Gatter einen Zellentyp zu. $IN : I \to E*$ legt für jedes Gatter eine Reihenfolge der eingehenden Kanten fest. Und zu guter Letzt $Y_m = (y_1, \ldots, y_m)$ sind alle Ausgangsknoten.

Standardzellen-Bibliothek enthält eine Menge an Gattern und kleinen kombinatorischen Schlatkreisen, sowie alle wichtigen Informationen zu den einzelnen Schlatungen, wie Fläche, Schaltgeschwindigkeit, etc.

3.3. zweistufige Synthese

Literale sind eine kurzschreibweise für Eingangsvariablen einer booleschen Funktion. Es gibt ein positives Literal x_i und das negative Literal x_i'. Sie dienen als kompakte Beschreibung, dass eine boolesche Funktion f eine wahre Aussage hat.

Monome ist die Bezeichnung für die oben beschriebene Funktion. Bei einem Monom darf jedes Literal nur einmal vorkommen, und dann auch nicht als Variable, sondern als positives oder negatives Literal. Monome heißen vollständig oder Minterm, wenn jede Eingangsvariable als positives oder negatives Literal vorkommt.

Polynome sind Disjunktionen von verschiedenen Monomen. (Also ODER-Funktionen, entweder ist das eine Monom eine logische 1 oder das nächste, aber wenn kein einziges der Monome eine logische 1 ausgibt, dann wird auch das Polynom keine 1 ausgeben) Sind alle Monome des Polynoms vollständig, dann heißt auch das Polynom vollständig.

Disjunktive Normalform von f ist ein Polynom, das beschreibt, wann f eine wahre Aussage ist. Eine Kanonisch disjunkte Normalform von f ist ein vollständiges Polynom von f.

Programmierbare logische Felder realisieren mehrere boolsche Funktionen auf einmal.

Veranschaulichung von Monomen durch Würfel funktioniert bis maximal 4 Eingängen. Dabei werden die Ecken eines Hypercubes mit allen möglichen Kombinationen bezeichnet und dann alle Monome markiert. Wenn zwei Monome mit einer Linie verbunden sind, dann kann man das eine Literal, das sie gemeinsam haben weg lassen, ist es eine Fläche, dann sind es zwei und ist es ein Volumen, dann lassen sich drei Literale weglassen. Wenn man alle Monome eingetragen und dann die Polynome ninimiert hat, erhält man die Primimplikanten der Funktion f.

3.4. Berechnung eines Minimalpolynoms

Minimalpolynom p ist die das günstigste Polynom einer Funktion f. Jedes Minimalpolynom besteht ausschließlich aus den Primimplikanten von f.

Quine-McCluskey ist ein Verfahren, welches nach den gemeinsamen Implikanten von Hypercube-Ecken sucht. McCluskey verbesserte Quine´s Algorithmus, in dem er das Überdeckungsproblem löste. Also, wenn mehrere Ecken die selben Werte an einem Literal haben, kann man dieses Implikanten in diesem Monom einfach weglassen.
Die Kosten des Verfahrens werden bei n Variablen mit 3^n berechnet.

Matrix-Überdeckungsproblem Nach der Anwendung des Quine-McCluskey-Algorithmus zur findung aller Primimplikanten müssen wir nur aus dem Ploynom noch das Minimalpolynom machen.
Dazu benötigen wir eine Matrix (Steht auch schon im Namen). Die Zeilen entsprechen den Primimplikanten von f und die Spalten den mintermen. Nun suchen wir eine Möglichkeit wie wir kostengünstig alle Hypercube-Ecken abzudecken ohne eine Ecke doppelt zu erwischen. Dabei gibt es drei wichtige Regeln zu befolgen:

1. Wesentliche Implikanten
2. Spaltendominanz
3. Zeilendominanz

Wesentliche Implikanten sind wichtig, weil diese Primimplikanten nur von einem (durch McCluskey gefundenen) Minterm abgedeckt werden können. Das bedeutet, dass diese Minterme bei dem Minimalpolynom gesetzt sind. Wichtig ist nun, dass man sich die Minterme aufschreibt und gut für später merkt! Dadurch werden andere Ecken vielleicht auch abgedeckt, gut für die. Damit sind dann schon alle diese Primimplikanten im Minimalpolynom enthalten und so können wir zu den nächsten zwei Reduktionsregeln voranschreiten.

Zweite Reduktionsregel: Spaltendominanz Spaltendominanz beruht, darauf, dass wenn ein zwei Primimplkanten von den selben Mintermen abgedeckt werden, der Primimplikant, der von mehr Mintermen abgedeckt wird, verliert und wird ignoriert. Denn sobald der, mit weniger Primimplikanten abgedeckt wird, dann wird auch automatisch der andere mitabgedeckt. In der Spalte ist es wichtig sich selten zu machen.

Zeilendominanz ist das Gegenteil von Spaltendominanz. Hier werden Min-terme und ihre Kosten verglichen, und wenn ein Minterm mehr abdeckt, dann lohnt es sich den schwächeren Minterm zu streichen.
Die letzteb beiden Regeln werden so lange angewandt bis sich wieder wesentliche Implikanten in der Matrix auftauchen, deren Minterme dann wieder "Gerettet"werden.

3.5. Arithmetische Schaltungen

Wow, der Titel klingt ect hochgestochen. Aber eigentlich sind das nur Schaltungen, die einen ganz bestimmen Zweck erfüllen können. Zwar nur einen einzigen, den dafür aber ganz hervorragend.

Kosten Die Kosten werden druch die Anzahl der Gatter eines Schaltkreises gegeben.

Tiefe ist die Maximale Anzahl an Schaltungen auf einem Pfad von einem Beliebeibigen Eingang zu einem beliebigen Ausgang. Also wie lange Strom in diesem Bauelement maximal unterwegs sein kann.
Im Folgenden wird nur noch mit logischen Gattern gearbeitet. Wenn man keinen Bock hat in ein Bauteil, was häufig in einer Schaltung vorkommt, aber nicht ein einzelnes Logikgatter ist, einzuzeichnen, dann kann man ganz entspannt auch ne Kiste malen. Darüber hinaus kommen einige Schaltbilder, und ich weiß nicht wei man die Einfügt, außerdem bin ich dafür zu faul es herauszufinden.

4. speichernde Elemente

Hier gibt es nicht viel zu erzählen. Das coolste hier sind RS-FLipflops.

RS-Flipflop sind Schaltungen, die über zwei Eingänge und zwei NAND-Gatter einen Bit speichern kann. Super cool. Das Problem dabei ist, dass man wissen muss, welchen Wert man speichert.

D-Latch Um die Flexibilität zu ermöglichen gibt es den D-Latch, welcher durch zwei weitere NAND-Gatter und einen Inverter in der Lage ist den RS-Flipflop unabhängig seiner bisherigen polung zu schalten. Um über einen D-Latch zu schreiben braucht man einen Schreibimpuls. Dafür braucht man eine gewisse Setup-Zeit t_{SDW}, die Pulsweite y und um sicher zu sein, dass die Schaltung auch wirklich geschalten hat die Hold-Zeit t_{HDW}. Dabei gibt es auch Taktgesteuerte D-Flipflops. Diese hängen an einer Clock und können dann mit nur einem Signal angesprochen werden. Es gibt ein paar einfache Bausteine mit Flipflops: Register, Schieberegister und Zähler.

- Register: Mehrere D-Flipflops hinterheinander, alle auf der selben Clock, aber alle mit unterschiedlichen Schreibsignalen.
- Schieberegister: Dabei gibt es nur einen Eingang und das Signal wird mit jedem Clock-Signal an den nächsten D-Flipflop weiter gegeben.
- Zähler: Ein Zähler, der nach und nach hochzählt.

4.1. Sequentielle Schaltkreise

Wir beschäftigen uns nun mit Schaltwerken. Das sind endliche Zustandsautomaten. Der Zustand eines Schaltwerkes ist durch die im Register gespeicherten Werte gegeben.

Endlicher Zustandautomaten sind ein Formalismus, um sequenzielles (zeitabhängiges) Verhalten zu spezifizieren.

Halbautomat $H = (I, S, S_0.\delta)$ sidn deterministische, endliche Halbautomaten. Dabei ist I das Eingabe Alphabet, S eine endliche Menge an Zuständen, $S_0 \in S$ ist eine endliche Menge an erlaubten Anfangszuständen und $\delta : S * I \to S$ eine Übertragungsfunktion.
Ein Automat lässt sich als gerichtetet Graph darstellen, wobei die Knoten die Zustände S und die Kanten die Übertragungsfunktionen δ sind.

Mealy-Automat/Moore-Automat $M = (I, O, S, S_0, \delta, \lambda)$ erweitert den Halbautomaten H um O, ein Ausgabe-Alphabet und um $\lambda : S \to O$ die Ausgabefunktion. Der Unterschied zwischen Moore und Mealy ist, dass bei Moore die Zustände gleich der Ausgabe sind. Bei Mealy wird die Ausgabe seperat auf die Übertragungsfunktion angewendet. Bei Mealy ist die Ausgabe abhängig vom Akutellen Zustand und der Eingabe. Bei einem Moore-Automaten (ein spezieller Mealy-Automat) ist die Ausgabe nur vom akutellen Zustand abhängig.
Ein Automat besteht dabei aus einem Kombinatorischen Kern und dem Register. Jeder Kern hat vier Arten von Ein- und Ausgängen:

- Primäre Eingänge: direkte Eingaben von Außen
- Primäge Ausgänge: driekte Ausgabe wieder nach draußen

- Sekundäre Eingänge: Kommen aus dem Register und ändern damit den Zustand des Automaten.
- Sekundäre Ausgänge: Speichern ins Register

4.2. Entwurf sequenzieller Schaltkreise

Zuerst braucht man ein Zustandsduragramm, eine Zustands- und Ausgangstafel, und einen Sequenziellen Schaltkreis. Danach kommt man zu einer Optimierung des Zustandsdieagrams, der Wahl der Zustandskodierung (da bekommt man die Anzahl der Flipflops und sowas raus) und zuletzt eine Implementierung über die Kombinatorische Logiksynthese.

Zustandsminimierung Die Idee ist, dass man gleiche Zustände zusammenfasst und so das Diagram verkleinert.

Zustandkodierung Ziel, wähle eine Zustandskodierung, die die nachfolgede kombinatorische Synthese erleichtert.

kombinatorische Synthese Dadurch wird das Zustanddiagramm reduziert.
letzter Schirtt ist die Implementierung. Das ist nicht mehr mein Problem.

4.3. SRAM

Static Random-Access Memory ist eine große Anzahl N and Speicherzellen, die schnell auf eine bestimmte Zelle zugreifen kann. Um das zu erreichen gibt es N D-Latches und einen gewaltigen Multiplexer um die einzelnen Speicherzellen anzusteuern. Ein N*s-Bit-SRAM besteht aus drei Hilfschaltkreisen:

- mehrfaches Oder Treiberbäumen Dekodierer

Treiberbäume sind Graphen, mit maximal 10 Kindern pro Knoten. Sobald ein 11. Kind an den Knoten hinzugefügt werden soll, dann wird in den nächst höheren Knoten ein neuer Knoten als Überknoten hinzugefügt. Das führt dazu, dass alle Blätter gleich weit von der Wurzel entfernt sind.

5. Timing

5.1. Physikalische Eigenschaften

Jede Technologie hat eine Versorgungsspannung VCC. Eine Spannung $U \in [0, VCC]$ wird als logischer Wert $I(U)$ angegeben. Wichtig ist hier nur, dass die Ausgangsspannungen für logische 1 höher ist als die Eingangsspannung für logische 1 am nächsten Gatter. Und die Ausgangsspannung für die logische 0 muss niedriger sein als die Eingangsspannung für eine logische 0 am anderen Gatter, denn Sonst bekommt das zweite Gatter gar nicht mit, dass es jetzt schalten soll. Wichtig sind hier noch die Rise- und Fall-Zeiten zu erwähnen. Da Logische Gatter immer verschieden reagieren, werden von den Herstellern verschiedene Daten über das Verhalten ihrer Bausteine gesammelt. Dazi gehören auch die minimalen und maximalen Schaltzeiten. Das ergibt

dann ein Zeitintervall, in dem der Schaltvorgang sicher passiert ist. Mit diesen Zeitintervallen kann dann wieder weitergerechnet werden.

spikefreies Umschlaten ist eine Lösung für ein Problem, dass auftritt, wenn zwei Eingänge eines Gatters beide gestauscht werden sollen $(0,1) \rightarrow (1,0)$. Dabei kann es passieren, dass ein der Eingang, der von 0 auf 1 wechselt vor dem anderen der schaltet und es so zu einer änderung der Ausgabe (zum Beispiel bei einem UND-Gatter) kommt. Das nennt man einen Spike. Um diesen Spike zu verhindern wird zu erst der eine Eingang abgeschaltet, gewartet, bis er auch sicher abgeschaltet ist, und dann erst der andere aufgeschaltet.

6. Binäre Entscheidungsdiagramme

Wie kann man feststellen, das zwei logische Schaltungen das selbe Ergebnis haben?
Diese Frage wird mit *binary decision diagrams* beantwortet.

BDDs repräsentieren die boolsche Funktion eindeutig und kompakt. Dabei ist ein BDD ein azyklischer gerichteter Graph mit genau einer Wurzel und zwei Ausgehenden Kanten für jeden inneren Knoten. Dabei sind 1 und 0 die beiden eingaben der Ausgangskanten. Es handelt sich bei einem BDD um einen Moore-Automaten.
Ein BDD ist ähnlich strukturiert wie ein PLA, das bedeutet, mit und und Oder verknüfungen werden die geforterten Zustände dargestellt. Dabei lassen sich die logischen Gatter durch die Verknüpfung von Zuständen darstellen.

reduzierte BDDs werden BDDs genannt, die bereits optimiert wurden und deswegen weniger Knoten und Kanten besitzten.

Konstuktion von BDDS geht natürlich auch.

Teil II.
Rechner Technische Informatik

1. ReTI

ReTI steht für Rechner Technische Informatik und ist ein abstraktes Konstrukt, an dass sich nach und nach angenähert werden soll. Die hier benutzte ReTI hat folgende Abstraktionen: zwei unendlich große Speicher für das Speichern von Daten S und dem Speichern von Maschienenbefehlen P. Darüberhinaus verfügt der PC über eine Zentrale Recheneinheit (CPU), welche aus vier Registern besteht: Einem PC Programm counter, einem ACC Akkumolator und zwei Indexregistern *IN1* und *IN2*. Die Programme und benötigtes Daten stehen bei Start der Maschiene in den jeweiligen Speichern. $PC = 0$ und die Maschiene arbeitet in Schritten von $t \in N$.

2. Reale ReTI

In der Realität sieht unsere ReTI ein bisschen anders aus. der Speicher hat eine größe von 2^{32} Speicherzellen mit jeweils Speicherpaltz für 32 Bit. Die CPU Register *PC,ACC, IN1* und *IN2* können auch nur Wörter von 32 Bit länge aufnehmen. Im folgenden werden wir diese 32 Bit einwenig in ihre Bestandteile zerlegen.
Es gibt 32 Bits. die ersten beiden, also bit 31 und 30 geben den Typ des Befehles an.

I[31,30]	**Typ**
0 0	Compute
0 1	Load
1 0	Store, Move
1 1	Jump

Tabelle 1: Befehlstypen einer ALU und ihre Codierung

Die zweiten zwei Bits also 29 und 28 sind für eine Unterauswahl des Befehles zuständig.

Typ	**Modus**	**Befehl**	**Wirkung**	
0 1	0 0	LOAD i	ACC:= M(<i>)	<PC>:= <PC>+1
0 1	0 1	LOADIN1 i	ACC:= M(<IN1>+[i])	<PC>:= <PC>+1
0 1	1 0	LOADIN2 i	ACC:= M(<IN2>+[i])	<PC>:= <PC>+1
0 1	1 1	LOADI i	ACC:= 0^8 i	<PC>:= <PC>+1

Tabelle 2: Alle Loadbefehle mit entsprechender Kodierung

Als nächstes betrachten wir die Move, und Store Befehle. Dabei werden mit Bits 29 und 28 wieder der Modus, mit Bits 27 und 26 die Quelle (Source) angegeben und mit 25 und 24 das Ziel (Destination) angegeben. Die Kodierung ist für Source und Destination gleich.

Ein Überblick über alle STORE- und MOVE-Befehle einer ALU

Kodierung der Compute-Befehle

Zuletzt gibt es noch Sprünge im Programmcode. Diese bieten die möglichkeit der Programmierung von while-Schleifen, denn der Sprung wird nur ausgeführt, wenn der Akkumulator die

S, D	Register
0 0	PC
0 1	IN1
1 0	IN2
1 1	ACC

Tabelle 3: Kodierung S, D

Typ	Modus	Befehl	Wirkung	
1 0	0 0	STORE i	M(<i>):= ACC	<PC>:= <PC>+1
1 0	0 1	STOREIN1 i	M(<IN1>+ [i]):=ACC	<PC>:= <PC>+1
1 0	1 0	STOREIN2 i	M(<IN2>+ [i]):=ACC	<PC>:= <PC>+1
1 0	1 1	MOVE S D	D:=S	<PC>:= <PC>+1

Tabelle 4: Die kodierten STORE- und MOVE-Befehele

Bedingung C im Bezug zur 0 erfüllt. Also wenn der Akkoumulator bei $c = 101$ ungleich 0 ist. Dann wird der Programm counter mit dem Wert i des Befehles verrechnet und das Programm setzt an der neuberechneten Stelle wieder an.

3. Aufbau der ALU

In der ALU werden alle Daten verarbeitet. Sie hat bei 32 Bit 68 Eingänge und 33 Ausgänge. Darüberhinaus hat die ALU 8 verschiedene Rechenoperationen, weshalb sie auch einen 3 bit großen select Eingang hat. Diese Funktionen werden mit einem verallgemeinerten Multiplexer gelöst. In der Realität wird die ALU so gebaut dass ähnliche Funktionen gemeinsam gereichnet werden können. Das sieht so aus, dass alle Addtitonen und Subtraktionen in einem Teil gemacht werden, und das die Vergeleichsoperationen in eigenen Schaltungen ausgeführt werden. Am Ende geht alles in 32 Bit wieder aus der ALU nach draußen.

4. Datenpfade in der ReTI

Es gibt eine Fetch und eine Execute Phase. Der Rest sind Schaubilder

Typ	Modus	F	Befehl	Wirkung	
0 0	0	0 1 0	SUBI i	[ACC] := [ACC] - [i]	<PC>:= <PC>+1
		0 1 1	ADDI i	[ACC] := [ACC] + [i]	<PC>:= <PC>+1
		1 0 0	OPLUSI i	ACC := ACC $\oplus 0^8 i$	<PC>:= <PC>+1
		1 0 1	ORI i	ACC := ACC $\vee 0^8 i$	<PC>:= <PC>+1
		1 1 0	ANDI i	ACC := ACC $\wedge 0^8 i$	<PC>:= <PC>+1
	1	0 1 0	SUB i	[ACC] := [ACC]-[M(<i>)]	<PC>:= <PC>+1
		0 1 1	ADD i	[ACC] := [ACC]+[M(<i>)]	<PC>:= <PC>+1
		1 0 0	OPLUS i	ACC := ACC $\oplus M(<i>)$	<PC>:= <PC>+1
		1 0 1	OR i	ACC := ACC $\vee M(<i>)$	<PC>:= <PC>+1
		1 1 0	AND i	ACC:= ACC $\wedge M(<i>)$	<PC>:= <PC>+1

Tabelle 5: Kodierung der Compute-Befehle

C	Bedingung c
0 0 0	nie
0 0 1	$<$
0 1 0	$=$
0 1 1	$\geq$
1 0 0	$<$
1 0 1	$\neq$
1 1 0	$\leq$
1 1 1	immer

Tabelle 6: Bedingungen für Sprünge