

Städtisches Gymnasium Gütersloh

Facharbeit

# ALU durch logische Operatoren erstellen und simulieren

verfasst von  
Philipp Kathöfer

Leistungskurs Mathematik  
Betreuer: Herr Imkamp  
Abgabetermin: 02.02.2021

# Begriffserklärung

## An

Synonyme: 1, true, wahr

An ist einer der beiden möglichen Status für ein Signal.

## Aus

Synonyme: 0, false, falsch

Aus ist der andere möglichen Status für ein Signal.

## Signal

Synonyme: Eingabe, Ausgabe, Bit, Operand

Ein Signal ist ein Knotenpunkt, der mit anderen Signalen verbunden werden kann und als Ein- oder Ausgabesignal dient. Ein Signal kann entweder an oder aus sein. Verbundene Signale haben immer den gleichen Status.

## Chip

Synonyme: Operator

Ein Chip hat einen oder mehrere Ein- und Ausgabesignale. Wenn ein Eingabesignal seinen Status ändert, werden die Ausgabesignale entsprechend angepasst. Ein Chip hat den Nutzen bestimmte Eingaben in bestimmte Ausgaben umzuwandeln.

# Inhaltsverzeichnis

1. Begriffserklärung
2. Einleitung (1)
3. Funktionen des Programmes (2 - 3)
4. Erläuterung des Quellcodes(4 - 8)
  - Programm Klasse
  - Signal Klasse
  - Chip Klasse
  - ChipLabel Klasse
  - Not Klasse
  - Table Klasse
  - Form1 Klasse
5. Mit logischen Operatoren rechnen (9 - 13)
  - Welche Operatoren werden benötigt?
  - And
  - Not
  - Nand
  - Or
  - Xor
  - Arithmetische Operatoren
  - Adder
  - 4Bit Adder
  - ALU
6. Literaturverzeichnis (14)
7. Selbstständigkeitserklärung (15)
8. Anhang

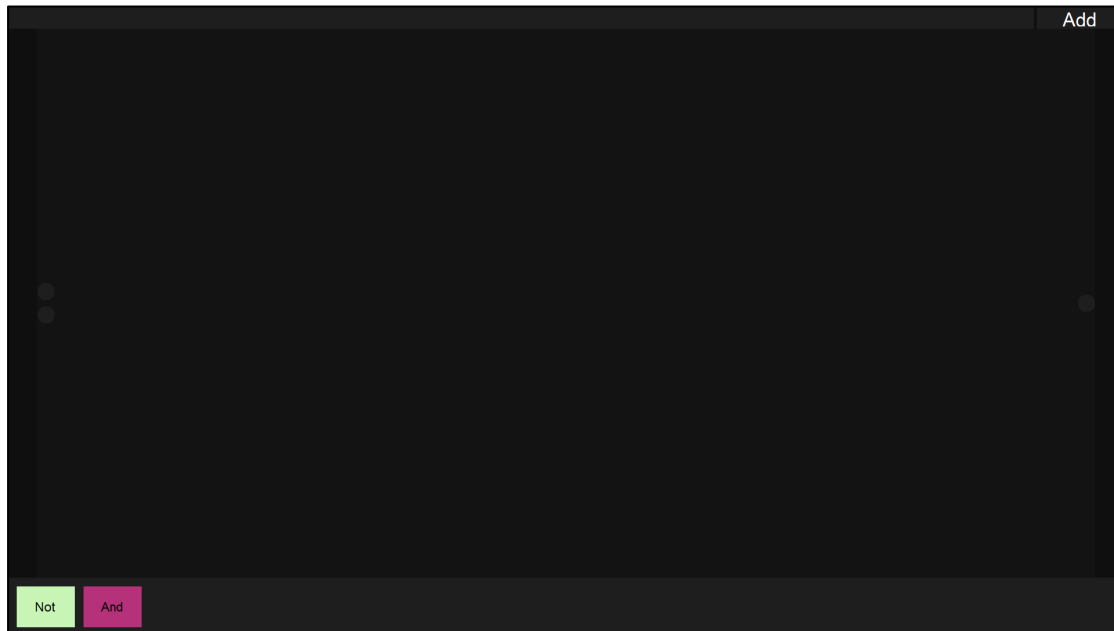
# Einleitung

Heute hat fast jeder Mensch einen Computer. In Deutschland haben ungefähr 90% der Menschen einen. [1] Sie sind aus der heutigen Zeit nicht mehr wegzudenken. Ohne Computer würde die Welt, wie wir sie heute kennen, nicht funktionieren. Computer haben einen großen Mehrwert für unsere Gesellschaft, Bereiche wie das Gesundheitswesen, das Verkehrswesen oder auch die wirtschaftliche Verwaltung wären ohne Computer heute nicht mehr denkbar.

Der Hauptbestandteil eines Computers ist die CPU (Central Processing Unit). Die CPU ist für Berechnungen und die Steuerung des Computers zuständig. Ohne die CPU würde ein Computer nicht funktionieren. Die wichtigste Komponente der CPU ist die ALU (Arithmetic & Logic Unit). Die ALU ist für arithmetische Prozesse, also das Rechnen mit Zahlen, sowie logische Prozesse, also die Untersuchung von Gültigkeit von Aussagen, zuständig. Eine ALU ist ein elektrischer Schaltkreis, welcher logische Operatoren darstellt, die aus bestimmten Anordnungen von elektrischen Verbindungen bestehen und immer komplexere Aufgaben lösen können. Logische Operatoren arbeiten mit Wahrheitswerten, welche durch „Strom fließt“ oder „Strom fließt nicht“ repräsentiert werden. Diese können auch als 1 bzw. 0 interpretiert werden, wodurch das allgemein bekannte Binärsystem entsteht. Im Anhang befindet sich ein lauffähiges Programm, welches solche logischen Operatoren simuliert und zum Konstruieren weiterer Operatoren geeignet ist. Der Fokus dieser Facharbeit liegt auf der Erklärung der Funktionen und des Quellcodes dieses Programmes. Darüber hinaus wird erklärt, wie man mit Hilfe des Programmes aus den grundlegenden logischen Operatoren „And“ und „Not“ eine ALU erstellen kann.

# Funktionen des Programmes

Das von mir entwickelte Programm hat den Nutzen, logische Operatoren zu verknüpfen und diese in sogenannte Chips zu komprimieren, um immer komplexere logische und arithmetische Operatoren bzw. Chips zu erstellen. Hier ein Screenshot aus dem Programm, wenn man es zum ersten mal startet:



In der unteren Leiste findet man alle vorhandenen Chips, welche man in das mittlere Feld ziehen kann, das wir Arbeitsfläche nennen. Zu Beginn gibt es nur die "And" und "Not" Chips. Wenn man einen Chip mit der linken Maustaste anklickt, kann man ihn auf der Arbeitsfläche platzieren, indem man die linke Maustaste erneut klickt. Jeden Chip kann man mit dem gleichen Prinzip auch verschieben.

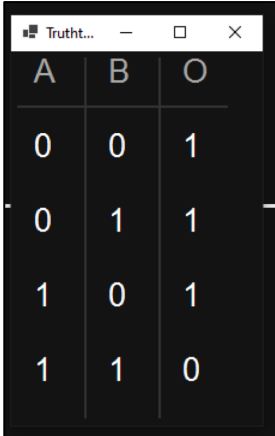
An der linken und rechten Seite des Bildschirms sieht man die Ein- bzw. Ausgaben, welche wir Signale nennen. Diese Signale kann man verbinden, indem man mit der linken Maustaste die jeweiligen Signale anklickt. Die Verbindung kann man wieder trennen, indem man sie erneut anklickt. Man kann manche Signale auch mit mehreren Signalen verbinden, wenn dies Sinn macht. Es macht z.B. keinen Sinn, zwei Eingaben auf der linken Seite des Bildschirms miteinander zu verbinden.

Man kann weitere Signale hinzufügen, indem man die Maus am linken, bzw. rechten Rand des Bildschirms platziert und die Pfeil-hoch-Taste auf der Tastatur drückt. Signale kann man entfernen, indem man die Shift-Taste drückt und ein Signal mit der linken Maustaste anklickt. Es muss jedoch mindestens ein Signal auf jeder Seite vorhanden sein.

Die Eingabe-Signale auf der linken Seite des Bildschirms kann man mit der rechten Maustaste anklicken, um sie ein- bzw. auszuschalten.

Wenn man nun einen eigenen Chip erstellt hat, kann man diesem einen Namen geben, indem man den Namen in das Textfeld oben eingibt. Wenn man auf den Add-Knopf klickt, wird dieser Chip komprimiert und in die untere Leiste hinzugefügt. Dieser Prozess kann bei vielen Eingabe-Signalen viel Zeit in Anspruch nehmen. Man sollte daher nicht mehr als 9 oder 10 Eingaben pro Chip verwenden. Nachdem der Chip komprimiert wurde, wird die Arbeitsfläche zurückgesetzt. Der Chip wird als .txt Datei in dem "ComputerSimulationSafe"-Ordner auf dem Desktop gespeichert. Wenn man einen Chip löschen möchte, muss man diesen aus dem Ordner entfernen.

Wenn man die Alt-Taste drückt und mit der linken Maustaste ein Ausgabe-Signal auf der rechten Seite des Bildschirms anklickt, öffnet sich ein Fenster, in dem man die Wahrheitstabelle für diese Ausgabe sehen kann. Dieser Prozess könnte für viele Eingabe-Signale ebenfalls länger dauern. Ein Screenshot von diesem Fenster kann man hier rechts sehen:

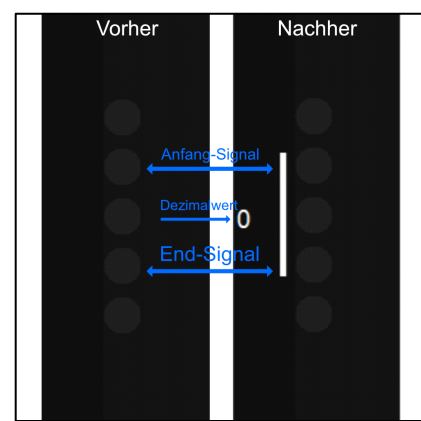


A	B	O
0	0	1
0	1	1
1	0	1
1	1	0

A, B und so weiter sind die jeweiligen Eingaben und O ist die Ausgabe. Die Wahrheitstabelle kann man sich jedoch nur für die Ausgaben auf der rechten Seite des Bildschirms anzeigen lassen.

Wenn man arithmetische Operatoren erstellen möchte und mit mehreren Eingaben arbeitet, welche Zahlen repräsentieren, kann dies schnell zu Verwirrung führen. Daher kann man sich den Dezimalwert von mehreren Signalen anzeigen lassen, indem man die Strg-Taste drückt und das Anfang-Signal und End-Signal mit der linken Maustaste anklickt.

An dem weißen Strich kann man erkennen, welche Signale zu der Dezimalanzeige gehören. Die Dezimalanzeige kann man wieder entfernen, indem man das Gleiche nochmal macht. Signale, die zu einer Dezimalanzeige gehören oder eine oder mehrere Verbindungen haben, können nicht entfernt werden. Um diese zu entfernen, müssen erst die Verbindungen bzw. Dezimalanzeigen entfernt werden.



# Erläuterung des Quellcodes

Das Programm ist aufgeteilt in acht Klassen: And, Chip, ChipLabel, Form1, Not, Program, Signal und Table. Diese sind in der Programmiersprache C# implementiert und nutzen das .NET Core Framework mit WindowsForms. Im Folgenden werde ich auf die einzelnen Klassen und ihre Methoden eingehen.

## Program Klasse

Die Program Klasse enthält die Main Methode. In dieser werden einige grafische Einstellungen vorgenommen und das Hauptfenster erzeugt.

## Signal Klasse

Die Signal Klasse ist die interne sowie grafische Darstellung eines Signals. Folgende Eigenschaften benötigt ein Signal:

- bool On;
- bool IsInput;
- List<Signal> Others;
- delegate void ChangedDelegate(bool on);
- ChangedDelegate Changed;

Im Konstruktor werden ein bool isInput und ein Chip chip verlangt, um die entsprechenden Eigenschaften zu initialisieren.

Die Signal Klasse erbt von der Label Klasse, um zu einem Fenster hinzugefügt werden zu können und grafische Eigenschaften zu erhalten. Zeile 41 bis 45 definieren die grafische Darstellung.

Der setter für den On Wert ruft von seinem Chip (falls vorhanden) die Calculate Methode auf, um die Werte der Ausgaben neu zu berechnen, wenn die Eingabe geändert wird.

## Chip Klasse

Die Chip Klasse ist die interne Darstellung eines Chips. Sie benötigt folgende Eigenschaften:

- Color Color;
- string Name;
- List<Signal> Inputs;
- List<Signal> Outputs;
- List<Chip> Chips;
- bool[,] table;

Ein Chip hat die folgenden Aufgaben:

1. Seine Eingaben zu verarbeiten und in seine Ausgaben umzuwandeln
2. Alle verbundenen Chips über Änderungen zu informieren
3. Eine neue Chip Instanz zu erstellen, die die gleiche Wahrheitstabelle hat
4. Seine Wahrheitstabelle in einer Datei zu speichern

Im Konstruktor wird ein string name verlangt, um die entsprechende Eigenschaft zu initialisieren. Die Farbe des Chips wird durch diesen berechnet. Die table Eigenschaft eines Chips speichert jede mögliche Konfiguration der Eingaben. Da jede Eingabe 2 Status haben kann, gibt es  $2^n$  mögliche Konfigurationen. Man kann sich eine Konfiguration als Binärzahl vorstellen, welche m Ziffern hat (für  $m$  = Anzahl Ausgaben). Das bedeutet, die table Eigenschaft speichert  $2^n \cdot m$  bool Werte, wodurch der Speicherbedarf eines Chips ein exponentielles Wachstum hat.

## Calculate Methode

Zuerst überprüft sie, ob die table Eigenschaft null ist und wenn ja, wird die Methode beendet. Falls das nicht der Fall ist, wandelt sie die On Eigenschaften der Eingaben in eine Dezimalzahl um, indem sie die On Eigenschaft eines Signals als Ziffer für eine Binärzahl verwendet. Das bedeutet, sie addiert  $2^i$  ( $i$  = index der Eingabe von hinten) auf die Zahl index. Diese Zahl index ist nun der Index in der table Eigenschaft, an der die entsprechende Ausgabe gespeichert ist. Dann wird die On Eigenschaft jeder Ausgabe auf den entsprechenden bool Wert gesetzt, falls der Wert noch nicht übereinstimmt. Diese Abfrage erspart viel Zeit, da das Ändern des On Wertes rekursiv ist. Zum Schluss wird die Send Methode aufgerufen, um die On Eigenschaften der Verbindungen der Ausgaben zu ändern. Die Calculate Methode ist virtual, da die „And“ und „Not“ Klassen diese überschreiben.

## Send Methode

Für jedes Ausgabesignal nimmt sie jede Verbindung und setzt dessen On Eigenschaft auf die des Ausgabesignals.

## CreateNew Methode

Zuerst erstellt sie eine neue Chip Instanz mit dem gleichen Namen. Dann erstellt sie genau so viele Ein- und Ausgaben wie in der aktuellen Instanz. Als nächstes überprüft sie, ob die table Eigenschaft nicht null ist und wenn ja, dann klonet sie diese auf den geklonten Chip. Zum Schluss wird dieser zurückgegeben.

## Safe Methode

Zuerst wird eine StreamWriter Instanz erstellt, welche eine .txt Datei in dem „ComputerSimulationSafe“ Ordner auf dem Desktop mit dem Namen des Chips erstellt. Falls dies fehlschlägt, gibt die Methode -1 als Fehlercode zurück. Falls dies nicht der Fall ist, schreibt sie die table Eigenschaft als Nullen und Einsen in diese Datei. Zum Schluss wird 1 zurückgegeben, um zu zeigen, dass keine Fehler aufgetreten sind.

## ChipLabel Klasse

Die ChipLabel Klasse ist die grafische Darstellung eines Chips. Folgende Eigenschaften werden benötigt:

- List<Signal> inputs;
- List<Signal> outputs;
- ClickedDelegate Connect;
- Chip chip;

Der Konstruktor erhält als Argument einen Chip chip. Dieser wird von dem ChipLabel grafisch dargestellt. Es werden grafische Einstellungen vorgenommen, z.B. die Größe der Länge des Namens anzupassen. Zum Schluss werden die Ein- und Ausgaben des chip über die InOutPut Methode zu dem ChipLabel hinzugefügt.

## InOutPut Methode

Die InOutPut Methode hat die Aufgabe, Ein- und Ausgaben hinzuzufügen und zu entfernen. Folgende Argumente bekommt die InOutPut Methode: List<Signal> list, bool add, Signal s. Der bool add gibt an, ob man das Element hinzufügen oder entfernen möchte. Die List<Signal> list gibt die Liste an, zu der das Element hinzugefügt wird oder aus der es entfernt wird. Und das Signal s ist das Signal, das hinzugefügt wird.

Zuerst wird überprüft, ob der bool add false ist, wenn ja, wird überprüft, ob weniger als zwei Elemente in der Liste sind, falls ja, wird die Methode abgebrochen, falls nicht, wird das erste Element aus der Liste und aus der Controls Liste entfernt. Falls der bool add true ist, wird zu dem Click Event des Signal s eine anonyme Methode hinzugefügt, welche das Connect Delegate aufruft. Danach wird das Signal s zu der List<Signal> list hinzugefügt. Am Ende der InOutPut Methode wird die Position aller Signale angepasst.



## Not Klasse

Die Not Klasse erbt von der Chip Klasse und benötigt keine weiteren Eigenschaften, da sie eine Konfiguration der Chip Klasse darstellen soll und keine Erweiterung.

Im Konstruktor wird der base, also der Chip Klasse, der name „Not“ übergeben und jeweils eine Ein- und Ausgabe hinzugefügt. Dann wird die Methode aufgerufen, um bei der Initialisierung die entsprechende Ausgabekonfiguration zu haben. Die Not Klasse überschreibt die Calculate und CreateNew Methoden der Chip Klasse. Die Not und And Klasse sind sehr ähnlich, daher wird hier nur die Not Klasse erklärt.

## CreateNew Methode

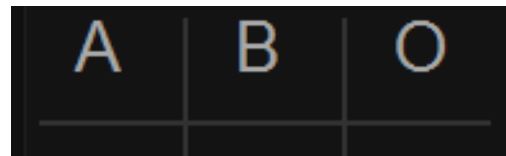
Es wird lediglich eine neue Not Instanz mit der entsprechenden Farbe zurückgegeben. Diese Methode ist trotz ihrer Simplizität notwendig, da sonst eine Instanz der Klasse Chip zurückgegeben werden würde, die nicht die Ein- und Ausgaben und die überschriebene Calculate Methode besitzt.

## Calculate Methode

Zuerst wird überprüft, ob genau eine Ein- und Ausgabe vorhanden ist. Falls ja, wird der On Wert der Ausgabe auf das Gegenteil der Eingabe gesetzt. Dann wird die Send Methode aufgerufen, um die Ausgabe weiterzugeben.

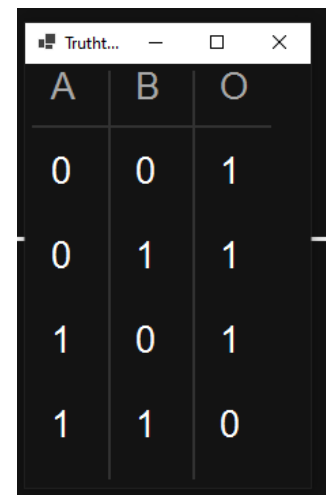
## Table Klasse

Die Table Klasse ist die grafische Darstellung einer Wahrheitstabelle (Siehe S. 4). Der Konstruktor bekommt als Argumente einen int inputs (Anzahl der Eingaben) und einen bool[] vals (On Werte der Ausgaben). In Zeile 20 wird ein Panel erstellt, welches die Wahrheitstabelle anzeigt. Der Code von Zeile 22 bis 38 wird in Form einer anonymen Methode zum Paint Delegate des Panels hinzugefügt, um auf dem Panel zu zeichnen. Zuerst wird für jede Eingabe der entsprechende Buchstabe, welcher jeweils eine Eingabe repräsentiert, sowie ein Trennstrich in einer Reihe gezeichnet. Anschließend wird zu dieser Reihe das „O“ hinzugefügt, welches die Ausgabe repräsentiert und die Reihe wird unterstrichen. Das Ergebnis kann man im Bild rechts sehen.



A	B	O
---	---	---

Um alle möglichen Werte der Eingaben darzustellen läuft eine for Schleife von 0 bis  $2^n$  ( $n$  = Anzahl der Eingaben). Das jeweilige  $i$  der for Schleife wird in Zeile 29 in eine Binärzahl konvertiert und in einem string gespeichert. Die Ziffern der Binärzahl sollen die On Werte der Eingaben darstellen. Das Problem ist jedoch, dass der string nur so viele Stellen hat, wie er benötigt. Das bedeutet, dass in den meisten Fällen vorne eine oder mehrere Nullen fehlen. Diese werden durch eine while Schleife in Zeile 30 bis 31 hinzugefügt. Dann wird jeder Buchstabe dieses strings in eine Reihe geschrieben. Zum Schluss wird die Ausgabe aus dem übergebenen bool Array an der Stelle  $i$  als 0 bzw. 1 in die Reihe geschrieben. Rechts kann man sehen, wie so eine Wahrheitstabelle für den Nand Chip aussieht.



A	B	O
0	0	1
0	1	1
1	0	1
1	1	0

## Form1 Klasse

Die Form1 Klasse steuert alle anderen Klassen und bildet die Benutzeroberfläche, auf der alle Elemente angezeigt werden. Der Code dieser Klasse ist über 400 Zeilen lang, daher werden hier nur die wichtigsten Teile gezeigt. Die hier erklärten Methoden sind:

- GetTable
- AddClicked
- InOutPut
- DrawBinaryConnections

Die Funktionen der anderen Methoden werden zum Schluss in Kürze beschrieben. Die wichtigste Eigenschaft dieser Klasse ist der Chip *chip*. Die Form1 Klasse ist dafür da, um einen Chip zu modellieren. Dieses Modell wird intern durch *chip* dargestellt. Dieser *chip* wird im weiteren Verlauf dieser Erklärung kursiv dargestellt, damit klar ist, dass diese Eigenschaft gemeint ist.

## Konstruktor

Um Ordnung zu wahren, werden die grafischen Einstellungen durch eine Präprozessor Direktive in eine `#region` geordnet. Danach werden die in dem „ComputerSimulationSafe“ Ordner gespeicherten Dateien geladen, falls diese vorhanden sind. Dazu wird zuerst überprüft, ob dieser Ordner existiert. Falls nicht, wird er erstellt. Falls doch, werden in einem string Array alle Dateipfade aus diesem Ordner gespeichert. Für jede Datei wird ein neuer Chip erstellt, welcher den Namen der Datei übernimmt. Dann wird der Inhalt aus der jeweiligen Datei in ein 2-dimensionales Array umgewandelt und in der `table` Eigenschaft des Chips gespeichert. Danach werden zu dem Chip so viele Ausgaben hinzugefügt, wie eine Zeile aus der Datei lang ist (Anzahl der Zeichen). Die Anzahl der Eingaben beträgt  $\log_2(m)$  ( $m$  = Anzahl Zeilen in der Datei). Dies ist so, weil die Anzahl der Zeilen die Anzahl möglicher Eingabekonfigurationen darstellt. Diese beträgt  $m = 2^n$  ( $n$  = Anzahl Eingaben,  $m$  = Anzahl möglicher Eingabekonfigurationen). Somit gilt:  $n = \log_2(m)$ . Zum Schluss wird der Chip durch die `AddChip` Methode hinzugefügt.

## GetTable Methode

Die Aufgabe der `GetTable` Methode ist es, die Wahrheitstabelle für ein bestimmtes Signal, welches als Argument übergeben wird, zurückzugeben. Daher hat die Methode den Rückgabewert `bool[]`. Dieses wird benutzt, wenn man mit `alt` + linke Maustaste ein Ausgabesignal anklickt (Seite 2). Dazu wird zuerst das entsprechende Array mit der Größe  $2^n$  ( $n$  = Anzahl Eingaben) angelegt. Dann wird durch eine `for` Schleife jede Eingabe des *chip* durchlaufen und der `i` Wert der `for` Schleife in Form eines string in eine Binärzahl umgewandelt. Danach werden die fehlenden Nullen vorne hinzugefügt. Dann werden die jeweiligen Zeichen des string als `On` Werte für die Eingaben benutzt. So wird jede mögliche Eingabekonfiguration einmal ausgeführt. Dann wird der `On` Wert des übergebenen Signals in dem Array an der Stelle `i` der `for` Schleife gespeichert. Zum Schluss werden die `On` Werte aller Eingaben wieder auf `false` gesetzt und das Array wird zurückgegeben.

## InOutPut Methode

Die `InOutPut` Methode hat die Aufgabe, ein Signal zu dem *chip* und zur Benutzeroberfläche hinzuzufügen. Dies ist jedoch sehr umfangreich, deshalb werden hier nur die wichtigsten Aspekte erklärt. Anders als die `InOutPut` Methode der `ChipLabel` Klasse, hat diese Methode nur ein Argument, nämlich die Liste, zu der eine Eingabe hinzugefügt werden soll. Das liegt daran, dass man keine bestimmten Signale hinzufügen muss, wie es bei der `ChipLabel` Methode der Fall ist. Das Entfernen eines Signals wird innerhalb der Methode verwaltet, dazu später mehr. Zu Beginn der Methode wird eine `Signal` Instanz erstellt. Dabei wird überprüft, ob die übergebene Liste die Eingabe Liste des *chip* ist und dieser Wert als Argument im Konstruktor des Signals übergeben wird. Der *Chip* für das Signal wird zuerst mit `null` initialisiert und später erst festgelegt. Danach wird zu dem `Changed` delegate eine anonyme Methode hinzugefügt, welche die `DrawBinaryConnections` Methode aufruft, um die Dezimalanzeigen neu zu zeichnen, wenn das Signal seinen `On` Wert ändert.

Der größte Teil der InOutPut Methode wird durch eine weitere anonyme Methode, welche zu dem MouseDown delegate des Signals hinzugefügt wird, in Anspruch genommen. Diese ist für den Umgang mit den verschiedenen Interaktionen, die man mit einem Signal machen kann, verantwortlich. Dazu gehören bspw. das Anklicken, um den On Wert zu ändern. Zuerst wird das übergebene sender Objekt zu einem Signal gecastet und in einer Signal Instanz gespeichert. Dann wird überprüft, ob die Maustaste, mit der das Signal angeklickt wurde, nicht die linke Maustaste ist, falls ja, wird die Methode abgebrochen. Es gibt vier verschiedene Interaktionsmöglichkeiten mit einem Signal. Diese sind: Wahrheitstabelle anzeigen, entfernen, zu einer Dezimalanzeige hinzufügen oder entfernen und das Verbinden mit anderen Signalen. Diese Interaktionen kann man durch das Drücken einer bestimmten Taste wählen (Seite 2 & 3). Wenn die Alt Taste gedrückt ist und das Signal keine Eingabe ist, wird eine Table Instanz erstellt und angezeigt, welche als Argumente *chip*.Inputs.Count (Anzahl an Eingaben des *chip*) und GetTable(l) (Wahrheitstabelle für das Signal, das man angeklickt hat), bekommt. Zum Schluss werden die On Werte aller Eingaben wieder auf den Wert von vor dem GetTable Aufruf gesetzt, da diese durch die Methode geändert wurden. Falls die Shift Taste gedrückt ist, wird zuerst überprüft, ob die übergebene Liste weniger als zwei Elemente hat oder ob das Signal nicht 0 Verbindungen hat. Falls ja, wird die Methode abgebrochen. Danach wird überprüft, ob das Signal zu einer Dezimalanzeige gehört, falls ja, wird die Methode abgebrochen. Falls dies nicht der Fall ist, wird das Signal aus der übergebenen Liste und von der Benutzeroberfläche entfernt und die DrawBinaryConnections Methode wird aufgerufen, um die Dezimalanzeigen zu aktualisieren. Falls keine Taste gedrückt wird, wird das Signal mit dem davor angeklickten verbunden.

### AddClicked Methode

Die AddClicked Methode wird aufgerufen, wenn der Add Button gedrückt wird, um einen Chip zu erstellen. Dazu wird zuerst überprüft, ob es bereits einen Chip mit diesem Namen gibt, falls ja, wird eine Fehlermeldung ausgegeben und die Methode wird abgebrochen. Danach wird überprüft, ob der Name valid ist. Es werden nämlich nur Zeichen zugelassen, deren Ascii Code kleiner gleich 128 ist, weil die Farbe des Chips aufgrund der Zeichen berechnet wird und diese Zeichen auf kleiner gleich 128 beschränkt sein müssen. Falls ein Zeichen größer als 128 ist, wird ebenfalls eine Fehlermeldung ausgegeben und die Methode wird abgebrochen. Dann wird ein 2-dimensionales bool Array mit den Größen  $2^n$  ( $n$  = Anzahl Eingaben) und Anzahl Ausgaben erstellt, welches die Wahrheitstabelle für den Chip darstellt. Dann geht eine for Schleife die erste Dimension des Array durch. Der entsprechende  $i$  Wert wird in eine  $n$ -stellige Binärzahl umgewandelt. Dann wird durch eine for Schleife jede Eingabe des Chips durchlaufen und der On Wert der Eingabe wird auf den Wert der Binärzahl an der Stelle  $j$  gesetzt. So wird jede mögliche Eingabekombination einmal durchlaufen. Dann wird in dem bool Array jeder On Wert der Ausgaben gespeichert. Dann wird ein neuer Chip erstellt, dessen table Eigenschaft auf das bool Array gesetzt wird. Danach wird dieser Chip durch die Save Methode als Datei gespeichert und falls die Methode -1 zurückgibt, wird eine Fehlermeldung ausgegeben. Dann wird dieser Chip hinzugefügt und die Benutzeroberfläche wird zurückgesetzt.

### DrawBinaryConnections Methode

Die Signale, die zu einer Dezimalanzeige gehören, werden in einer Liste aus Listen aus Signalen gespeichert. Eine Liste gibt an, welche Signale zu einer Dezimalanzeige gehören. Da man mehrere Dezimalanzeigen haben kann, gibt es eine Liste aus diesen Listen. In der DrawBinaryConnections Methode werden Listen durchlaufen und aus den On Werten der Signale aus einer Liste wird eine Dezimalzahl erstellt und angezeigt. Dazu wird ein int val erstellt, welcher mit 0 initialisiert wird und die Dezimalzahl darstellt. Dann wird durch eine for Schleife, die bei 1 startet jedes Signal aus einer Liste durchlaufen. Falls der On Wert des entsprechenden Signals true ist, wird auf  $val \cdot 2^{n-i-1}$  ( $n$  = Anzahl Signale, die zu der Dezimalanzeige gehören) addiert. Zum Schluss wird von  $val \cdot 2^{n-1}$  abgezogen, falls der On Wert des ersten Signals true ist, weil die vorderste Ziffer einer Binärzahl der negative Wert ist. Diese Zahl val wird dann auf den Bildschirm gezeichnet.

# Mit logischen Operatoren rechnen

Durch geschickte Verknüpfung von logischen Operatoren können weitere logische Operatoren erstellt werden. Diese wiederum können genutzt werden um arithmetische Operatoren zu erstellen. Jedoch müssen mindestens zwei logische Operatoren vorausgesetzt sein. In dieser Facharbeit werden die „And“ und „Not“ Operatoren vorausgesetzt. Das Ziel dieses Abschnittes ist es den arithmetischen Teil einer ALU mit Hilfe meines Programmes zu entwickeln. Das bedeutet der Chip den wir erstellen wollen soll zwei natürliche Zahlen addieren und subtrahieren können. Die ALU muss außerdem in der Lage sein, Metadaten über das Ergebnis zu geben, z.B. ob eine Zahl 0, negativ oder zu groß ist, um dargestellt zu werden. Die ALU, die in dieser Facharbeit digital erstellt wird, ist sehr primitiv. Heutzutage sind ALUs wesentlich komplexer und können deutlich mehr, dies würde den Rahmen dieser Facharbeit jedoch sprengen. Die Art und Weise, wie diese spezifische ALU erstellt wird, kann auch variieren. Hier wird nur eine Möglichkeit vorgestellt.

## Welche Operatoren werden benötigt?

Wie bereits angemerkt verwenden wir „And“ und „Not“ als Basis. Folgende logische Operatoren werden benötigt, die wir aus „And“ und „Not“ erstellen können:

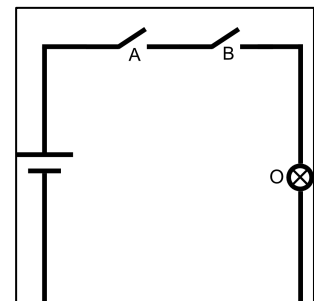
- „Nand“
- „Or“
- „Xor“

Daraus können folgende arithmetische Operatoren erstellt werden, welche für eine ALU benötigt werden:

- „Adder“
- „4Bit Adder“

## And

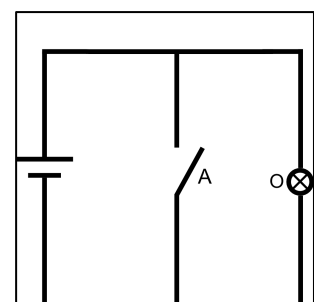
Der „And“ Operator ist ein Binäroperator. Das heißt, dass er zwei Eingaben, auch Operanden genannt, bekommt, diese nennen wir A und B. Jedoch liegt nur eine Ausgabe vor, diese nennen wir O. Da „And“ ein Operator ist, der vorausgesetzt wird, gibt es dafür keine Umsetzung im Programm. Das bedeutet, er muss als elektrischer Schaltkreis gebaut werden. Dieser sieht so aus:



In einem echten Computer wären die A und B Schalter sogenannte Transistoren, die durch elektrische Impulse geöffnet und geschlossen werden können. A und B dienen als Eingaben, welche Informationen beinhalten. Diese Informationen sind in Form ihres Zustandes, das heißt sie können „an“ oder „aus“ sein. Diese Information wird als Eingabe genutzt. Die Lampe, also die Ausgabe, geht nur dann an, wenn A **und** B an, also 1, sind. Die Anzahl an möglichen Status der Eingaben ist  $2^n$  (für  $n$  = Anzahl Eingaben, für den „And“ Operator  $n = 2$ ), da sich diese Anzahl mit jeder hinzukommenden Eingabe verdoppelt, weil jede Eingabe zwei mögliche Status haben kann, wahr oder falsch (1 oder 0, an oder aus).

## Not

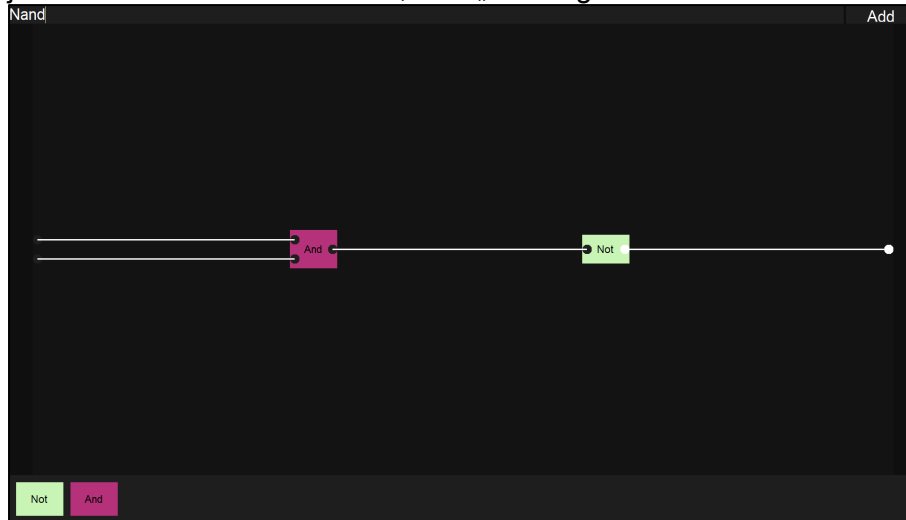
Der „Not“ Operator hat nur einen Operanden und ebenfalls eine Ausgabe. Da „Not“ ebenfalls vorausgesetzt wird, müsste hierfür in einem echten Computer ein elektrischer Schaltkreis gebaut werden. Dieser sieht so aus:



Wenn A geschlossen, also „eingeschaltet“ wird, hat der Strom einen einfacheren Weg und fließt somit nicht an der Lampe entlang.

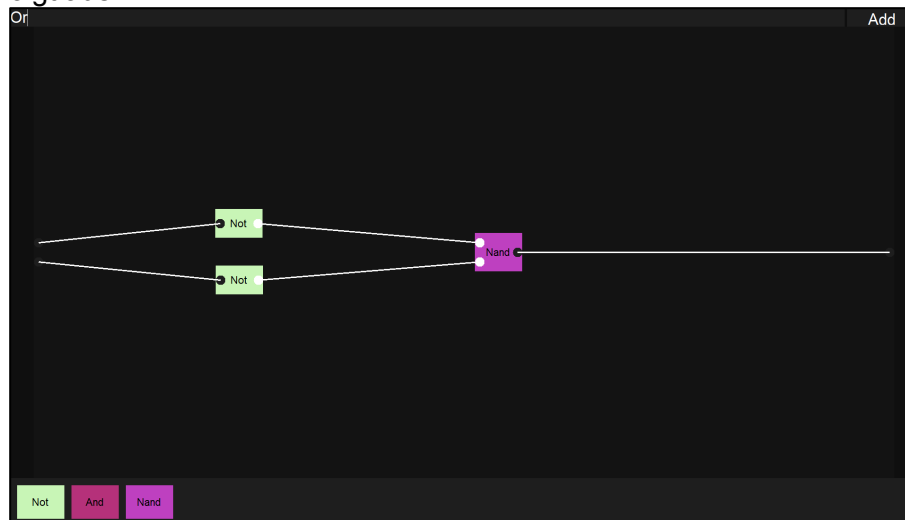
## Nand

„Nand" ist ein Binäroperator und macht fast das Gleiche wie „And", invertiert das Ergebnis jedoch. Daher wird er Not And, also „Nand" genannt. Der Aufbau sieht folgendermaßen aus:



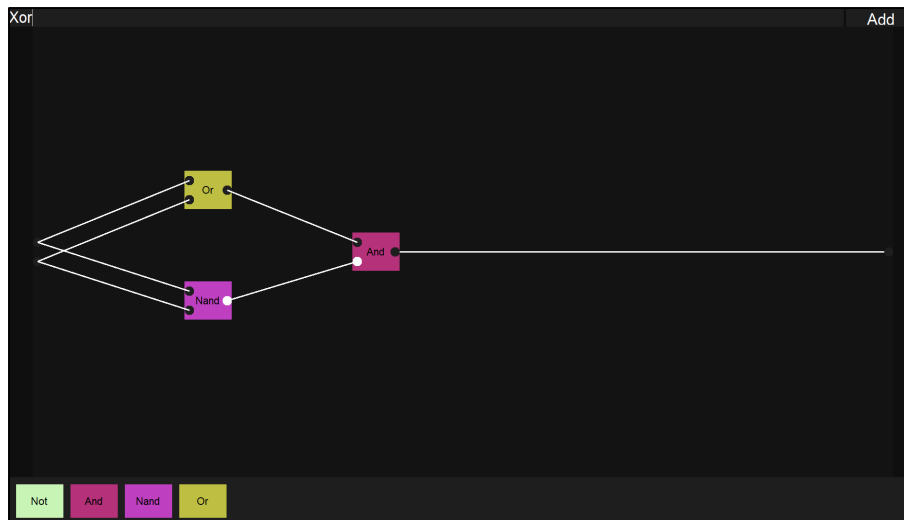
## Or

„Or" ist ebenfalls ein Binäroperator und hat auch nur eine Ausgabe. Diesen Operator erhält man, wenn man die beiden Eingaben eines „Nand" Operators invertiert. Das sieht dann wie folgt aus:



## Xor

„Xor" ist ebenfalls ein Binäroperator mit einer Ausgabe. „Xor" steht für „Exclusive or", also „ausschließendes oder". Umgangssprachlich nennt man es auch „entweder oder". Das bedeutet, es soll die gleichen Eigenschaften wie „Or" haben, aber wenn beide Operanden 1 sind, soll die Ausgabe 0 sein. Die Verknüpfung sieht wie folgt aus:



Wie man sehen kann wird durch den „Or“ Operator überprüft, ob einer der beiden Operanden 1 ist, durch den „Nand“ Operator wird überprüft ob nicht beide Operanden 1 sind. Die Ausgabe der beiden Operatoren sind die Eingabe in einen „And“ Operator. Dessen Ausgabe ist die finale Ausgabe für den „Xor“ Operator. Das bedeutet die Ausgabe des „Xor“ Operators ist dann 1, wenn eine der beiden Eingaben 1 ist, aber nicht beide. Sonst ist die Ausgabe 0.

## Arithmetische Operatoren

Um arithmetische Operatoren zu erstellen, muss man verstehen, wie Computer aus Wahrheitswerten (wahr oder falsch, 1 oder 0) Zahlen interpretieren. Wenn man sich mehrere Wahrheitswerte in einer Reihe ansieht, z.B. 101010, so erkennt man, dass dies eine Zahl ist, die im Binärsystem dargestellt ist. Wie Binärzahlen funktionieren wird in dieser Facharbeit nicht erklärt. Was jedoch angemerkt werden sollte, ist dass die vorderste Zahl das Negative ihres eigentlichen Wertes ist. Bei der Beispielszahl 101010 sieht das dann so aus:

Wert	-32	16	8	4	2	1
Binär-	1	0	1	0	1	0
Ziffer						

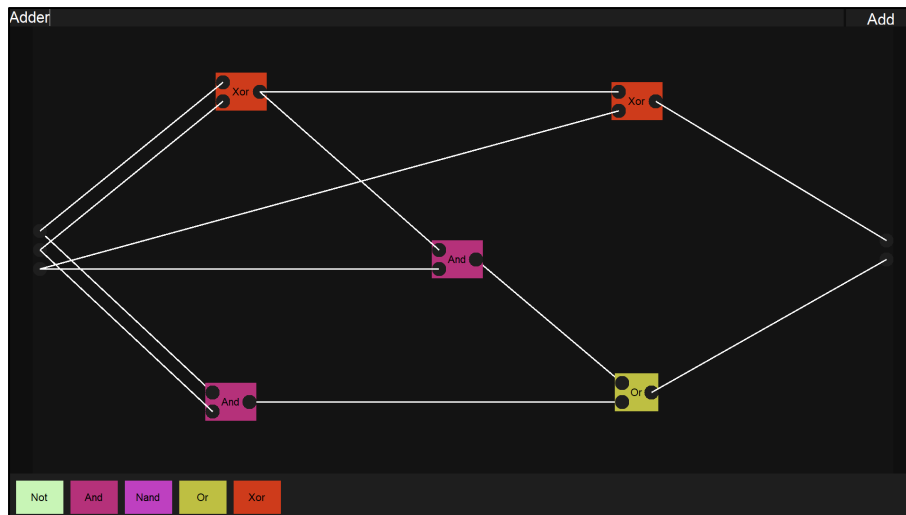
Dies halbiert die Menge an möglichen positiven Zahlen, ist aber nötig um Zahlen zu subtrahieren und negative Zahlen darzustellen. Man könnte denken, dass die vorderste Binärziffer angibt, ob die Zahl positiv oder negativ ist, die hat jedoch einige Probleme:

1. Es gäbe zwei mal die Zahl 0. 0 und -0 ist dasselbe. Man würde also eine Zahl „verschwenden“.
2. Wenn man z.B. 5 und -5 addiert, erwartet man als Ergebnis 0 zu erhalten. Wenn man dies jedoch mit der vordersten Ziffer als Indikator für positiv oder negativ probiert käme dabei 6 als Ergebnis raus. Wenn man die vorderste Ziffer jedoch als das Negative des eigentlichen Wertes interpretiert, so kommt bei diesem Beispiel 0 als Ergebnis raus. [5]

## Adder

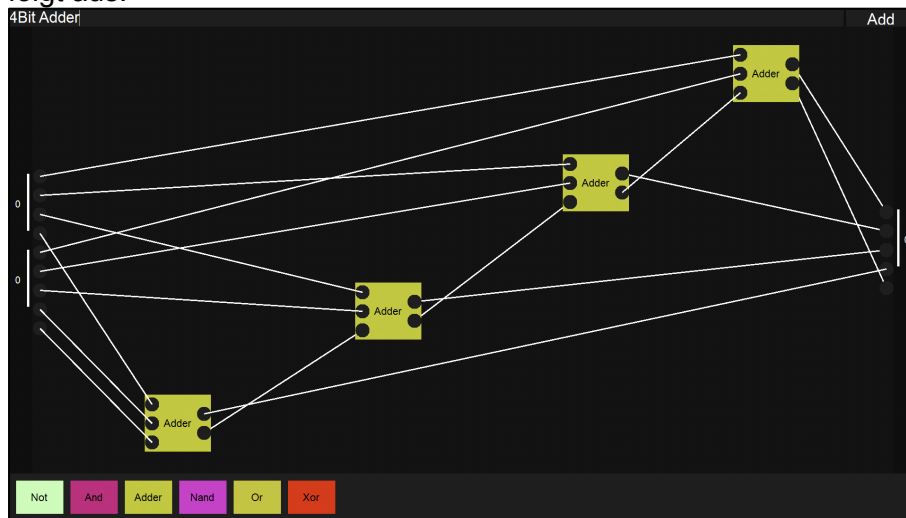
Ein „Adder“ erhält drei Eingaben und gibt zwei Ausgaben zurück. Der „Adder“ addiert zwei einstellige Binärzahlen, welche als Summanden bezeichnet werden und durch die ersten beiden Eingaben dargestellt werden. Die dritte Eingabe ist der sogenannte Carry In. Die erste Ausgabe des „Adder“ repräsentiert das Ergebnis der Addition. Wenn man jedoch die Zahlen 1 und 1 addiert, ist das Ergebnis zu groß, um in einer einstelligen Binärzahl dargestellt zu werden. Dafür ist die zweite Ausgabe des „Adder“, welche wir Carry Out nennen zuständig. Dieser kann als Carry In für weitere „Adder“ dienen.

Die Umsetzung dafür im Programm sieht so aus:



## 4Bit Adder

Der „4Bit Adder“ ist die letzte Komponente, die wir benötigen, um eine ALU zu konstruieren. Der „4Bit Adder“ erhält neun Eingaben und gibt fünf Ausgaben zurück. Acht der neun Eingaben repräsentieren zwei vierstellige Binärzahlen, die neunte Eingabe ist der Carry In. Vier der fünf Ausgaben repräsentieren eine vierstellige Binärzahl, die die Summe der beiden „Eingabe-Zahlen“ darstellt. Die fünfte Ausgabe ist der Carry Out. Den „4Bit Adder“ kann man durch eine Verkettung aus vier „Addern“ erstellen. Die Umsetzung im Programm sieht wie folgt aus:

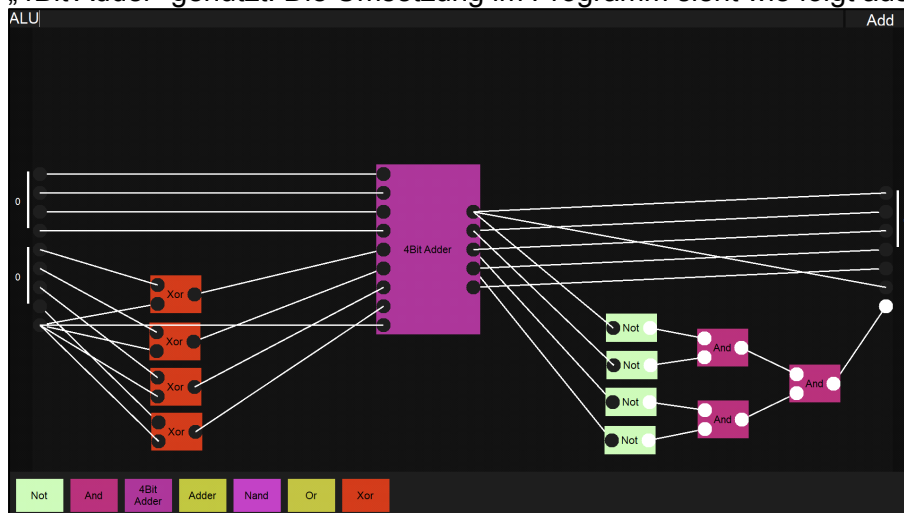


Der 4Bit „Adder“ kann  $2^4$ , also 16 Zahlen darstellen. Da auch negative Zahlen mit eingeschlossen sind, kann der „4Bit Adder“ Zahlen zwischen einschließlich -8 und 7 darstellen. Wenn eine Zahl außerhalb dieses Bereiches liegt, bekommt der Carry Out den Wert 1. Außerdem kann es passieren, dass ein „falsches“ Ergebnis berechnet wird, wenn man zwei Zahlen addiert dessen Summe größer als 7 ist, also außerhalb des Wertebereiches liegt. Dies liegt jedoch nur an der Dezimalanzeige, die das Ergebnis als negative Zahl interpretiert. Die binäre Addition ist jedoch korrekt.

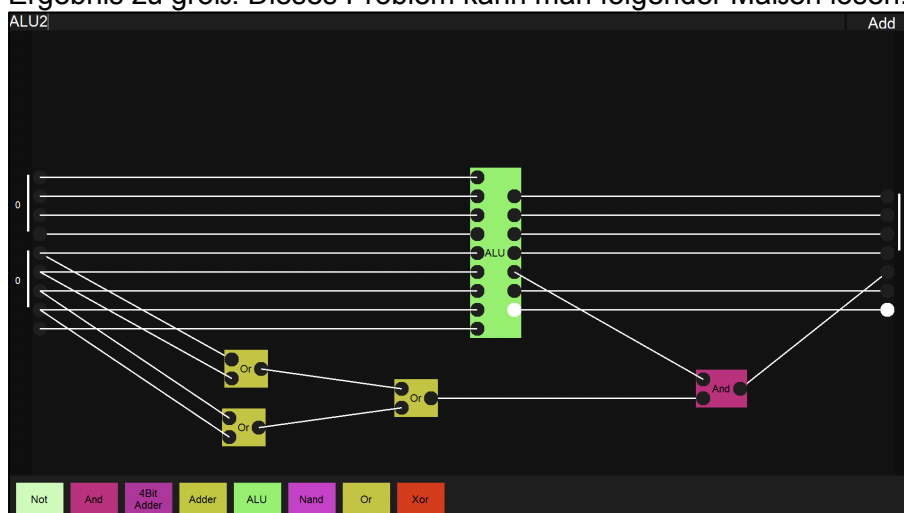
## ALU

Zuerst müssen wir klären, was die ALU können muss, da es verschiedene Definitionen gibt. Hier werden wir versuchen eine simple ALU zu erstellen. Zu den Aufgaben sollen gehören: Addition, Subtraktion für 4Bit Binärzahlen und Informationen über das Ergebnis geben, z.B ob die Zahl 0, negativ oder zu groß ist um dargestellt zu werden (overflow). Das bedeutet, wir kümmern uns nur um die arithmetischen Aufgaben und lassen dabei den Logik-Aspekt weg, da wir dafür bereits Operatoren erstellt haben.

Insgesamt benötigen wir neun Eingaben und sieben Ausgaben. Acht der neun Eingaben stellen die beiden 4Bit Zahlen dar und die neunte Eingabe, ist ein sogenanntes Controll-Bit, welches angibt, ob wir die Zahlen addieren oder subtrahieren wollen. Vier der sieben Ausgaben repräsentieren das Ergebnis der Addition, bzw. Subtraktion und die drei übrigen Ausgaben geben die eben genannten Informationen über das Ergebnis. Um einstellen zu können, ob die beiden Zahlen addiert oder subtrahiert werden sollen, brauchen wir eine Möglichkeit die Zweite Zahl in ihr Negativ umzuwandeln. Dazu muss man die Zahl auf binärer Ebene negieren, also alle Einsen in Nullen umwandeln und umgekehrt. Danach muss man 1 addieren. Dafür sind die „Xor“ Chips. Diese negieren die zweite Eingabe, falls die unterste Eingabe 1 ist. Um 1 zu addieren wird die unterste Eingabe als Carry-In für den „4Bit Adder“ genutzt. Die Umsetzung im Programm sieht wie folgt aus:



Die „Xor“ Chips wandeln die zweite Zahl in eine negative Zahl um, falls der Controll-Bit auf 1 gesetzt ist. Die unteren drei Ausgaben der ALU geben Informationen über das Ergebnis. Die obere zeigt an, ob das Ergebnis zu groß ist. Diese Ausgabe wird auf 1 gesetzt, wenn der Carry-Out des „4Bit Adder“ 1 ist. Die untere Ausgabe zeigt an, ob das Ergebnis 0 ist. Dies ist der Fall, wenn keine der oberen vier Ausgaben des „4Bit Adder“ 1 ist. Die mittlere Ausgabe der ALU gibt an, ob das Ergebnis negativ ist. Dies ist der Fall, wenn die vorderste Zahl des Ergebnisses 1 ist, da diese den negativen Wert darstellt und dessen Betrag größer als die Summe aller anderen Bits ist. Ein Problem, welches mit dieser Umsetzung entsteht ist, dass wenn man 0 subtrahiert, die Overflow-Ausgabe auf 1 gesetzt wird. Das liegt daran, dass wenn man 0 negiert in 4Bit binär 1111 bekommt. Und wenn man darauf 1 addiert, ist das Ergebnis zu groß. Dieses Problem kann man folgender Maßen lösen:



So wird die Overflow-Ausgabe nur dann auf 1 gesetzt, wenn die zweite Zahl nicht 0 ist.



# Erklärung des Verfassers

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und alle Formulierungen, die wörtlich oder dem Sinn nach aus anderen Quellen entnommen wurden, kenntlich gemacht habe.

Verwendete Informationen aus dem Internet sind dem(r) Lehrer/in vollständig im Ausdruck zur Verfügung gestellt worden, einschließlich der genauen Angabe der Internetadresse.

Sofern sich – auch zu einem späteren Zeitpunkt – herausstellt, dass die Arbeit oder Teile davon nicht selbstständig verfasst wurden, die Zitationshinweise fehlen oder Teile ohne Quellennachweis aus dem Internet entnommen wurden, so wird die Arbeit auch nachträglich mit null Punkten bzw. Note sechs gewertet.

Ich erkläre mich damit einverstanden, dass die vorliegende Arbeit für schulische Zwecke benutzt werden kann.

Gütersloh, 30.01.2021