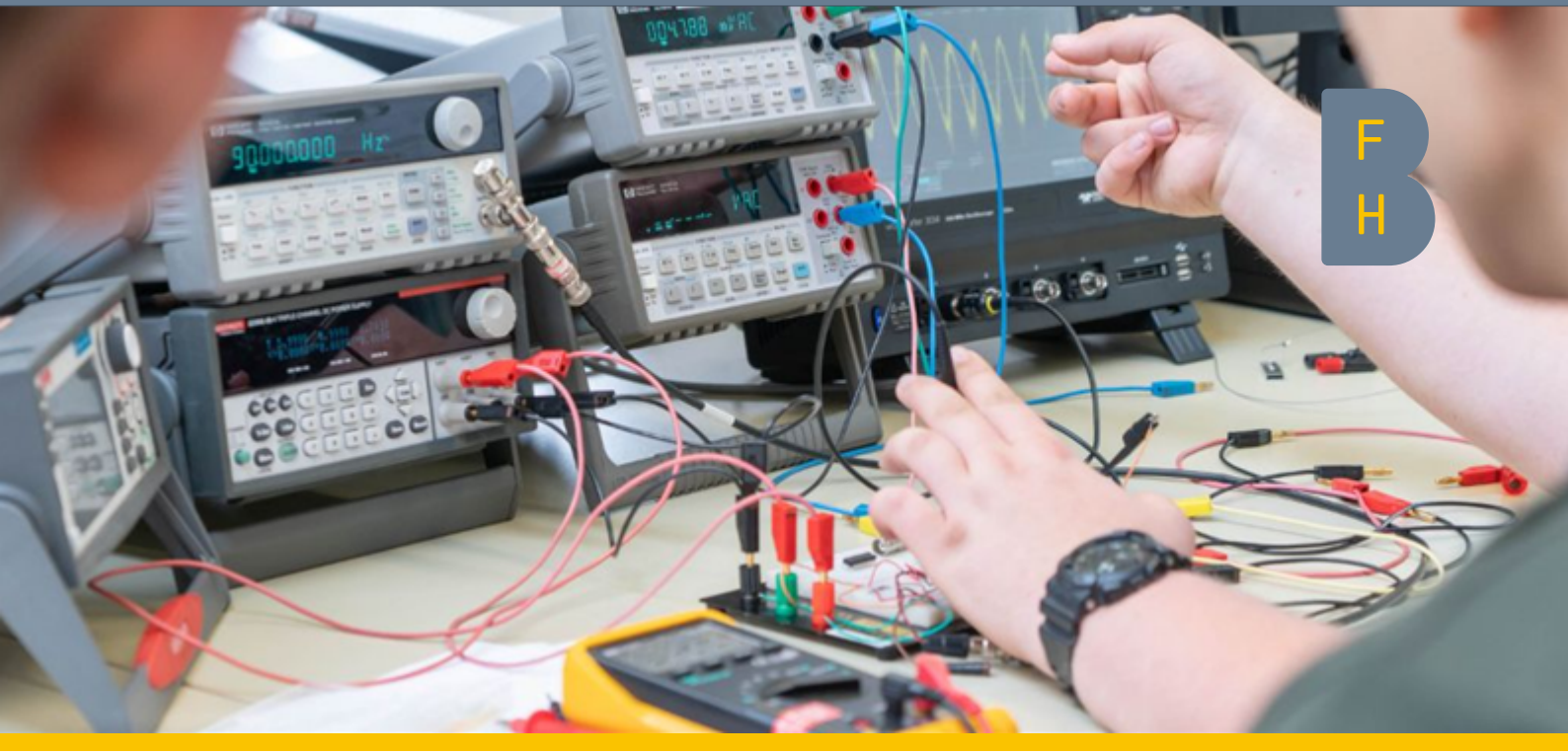


# Projekt 4: Kombinatorische Logik

BTE5213 Laborprojekte, Herbstsemester 2025/2026
Protokoll

Janis Aebischer und Simon Eisele (Gruppe D)

Version 1.0 vom 9. November 2025

- Technik und Informatik
- Elektrotechnik und Informationstechnologie

Durchführung des Versuchs am 24. Oktober 2025

Betreut durch Prof. Dr. Torsten Mähne


## Abstract

Ziel dieses Laborprojekts ist es, die im Modul Digitaltechnik erlernten Grundlagen zu vertiefen und praktisch anzuwenden. Dazu wurden mehrere kombinatorische Schaltungen entwickelt, deren logische Funktionen mithilfe der Booleschen Algebra und KV-Diagrammen vereinfacht wurden. Die entwickelten Schaltungen wurden in Logisim-evolution umgesetzt, simuliert und anschliessend auf dem Leguan-Board der BFH getestet. Durch die Verknüpfung von Theorie, Simulation und praktischer Umsetzung konnte das Verständnis für den Aufbau und die Funktionsweise digitaler Logiksysteme gezielt erweitert werden. Bei den Logikfunktionen handelt es sich um folgende Baugruppen. BCD-zu-7-Segment-Decoder, Ripple-Carry-Addierer, Borrow-Bit-Subtrahierer, Ergebnisumschalter und Binär-zu-BCD-Converter.

# Inhaltsverzeichnis

# 1. Einleitung

Ziel des Projekts 4 „Kombinatorische Logik“ war es, die Software Logisim-evolution (LE) [1][1], sowie das Leguan-Board (LB) [2][2] der Berner Fachhochschule (BFH) kennenzulernen. Dabei sollten die aus dem Modul BTE5021-Digital „Elektronik Grundlagen“ gewonnenen Kenntnisse angewendet und gefestigt werden.

Hierfür wurden verschiedene Schaltungen entwickelt, in LE aufgebaut und zum Testen auf den Field-Programmable Gate Array (FPGA) des LB geladen. Die zu entwickelnden Schaltungen wurden in der Aufgabenstellung [3] definiert:

- Binary-Coded Decimal (BCD)-zu-7-Segment-Decoder
- Ripple-Carry-Addierer
- Borrow-Bit-Subtrahierer
- Ergebnisumschalter
- Binär-zu-BCD-Konverter

Für die Entwicklung der Schaltungen wurden jeweils Wahrheitstabellen erstellt. Daraus konnten die entsprechenden KV-Diagramme abgeleitet und anschliessend die minimalen Funktionen in disjunktiver (Minterme) oder konjunktiver (Maxterme) Form bestimmt werden. Beim BCD-zu-7-Segment-Decoder mussten zusätzliche Bedingungen berücksichtigt werden. Für die Funktionen durften jeweils nur bestimmte Grundgattertypen verwendet werden. Dies bot die Möglichkeit, das Umformen mittels der Booleschen Algebra anzuwenden und zu üben.

Vor der praktischen Durchführung des Versuchs wurden die Vorbereitungsaufgaben [3, p. 3] bearbeitet. Dabei ging es darum, die Schaltungen für den BCD-zu-7-Segment-Decoder gemäss den gestellten Bedingungen zu entwickeln, sowie ein Konzept für die logische Schaltung zur Umwandlung einer 8 bit-Binärzahl in 3 BCD-Ziffern zu erstellen.

Der praktische Versuch wurde im Labor durchgeführt und dauerte vier Lektionen. Im Anschluss blieb noch eine Woche, um fehlende Schaltungen zu vervollständigen und den technischen Bericht fertigzustellen.

Die Grundlagen zur Durchführung dieses Projekts bildet das Modul BTE5021-Digital „Elektronik Grundlagen“. Die wichtigsten Inhalte sind im Skript Digitaltechnik [4] zusammengefasst.

[1] https://github.com/logisim-evolution/logisim-evolution
[2] https://leguan.ti.bfh.ch/

# 2. Methoden und Materialien

## 2.1. Versuchs- und Projektaufbau

### 2.1.1. Ziele

Die Herangehensweise wurde durch die Aufgabenstellung [3] bereits ziemlich genau definiert. Als erstes mussten die Vorbereitungsaufgaben [3, p. 3] gelöst werden, welche grundsätzlich aus drei Teilen bestehen:

- Funktionen für den BCD-zu-7-Segment-Decoder ermitteln
- Konzept für den Algorithmus zur Umwandlung einer 8 Bit-Binärzahl in 3 BCD-Ziffern
- LE kennelernen und über die Möglichkeiten dieser Software informieren.

Diese Aufgabenstellungen wurden vor der Projektdurchführung abgearbeitet und bereits zur Bewertung abgegeben. Sie bilden zusammen die Grundlage für dieses Projekt.

Während des Versuchs vor Ort sollen die anderen Aufgaben gemäss der Aufgabenstellung [3] bearbeitet werden. Diese Aufgaben sind der Aufgabenstellung zu entnehmen und lassen sich in 5 Teilaufgaben unterteilen.

- BCD-zu-7-Segment Decoder
- Ripple-Carry-Addierer
- Borrow-Bit-Subtrahierer
- Ergebnisumschalter
- Binär-zu-BCD Umwandler

Für das Abarbeiten dieser Aufgaben ist das Labor da, falls die Zeit nicht reicht muss noch nachgearbeitet werden. Anschliessend soll ein Projektbericht erstellt werden.

### 2.1.2. Vorgehen

Das allgemeine Vorgehen ist für alle Aufgabenstellungen gleich. Als erstes wird für jede Baugruppe eine Wahrheitstabelle erstellt. Mit dieser Wahrheitstabelle lässt sich das KV-Diagramm ableiten. Mit dem erstellten KV-Diagramm lässt sich die Normalform ablesen. Falls in der Aufgabenstellung gefordert, müssen die Normalformen erweitert werden, so dass sie auf den beschränkten Gattervorrat passen. Die daraus resultierenden Logikfunktionen werden anschliessend in Logisim-evolution (LE) implementiert. Für dieses Projekt gab es bereits eine Vorlage die verwendet werden musste. Mithilfe der Simulation in LE kann die jeweilige Funktion direkt auf ihre richtige Funktionsweise überprüft werden. Zum Abschluss erfolgt die Verifikation mit dem FPGA-Board der BFH, dem Leguanboard (LB). Dank der LE-Vorlage werden die erstellten Funktionen

bereits automatisch in die jeweilige Testumgebung integriert. Es bedarf nur noch dem Download der LE-Datei und dem festlegen der I/O-Elemente auf dem LB. Anschliessend kann die Logikfunktion der jeweiligen Baugruppen mit physischen Elementen getestet werden. Für alle Baugruppen wird ein Testprotokoll erstellt und ausgefüllt. Mehr dazu in Kapitel 3.

Die oben genannten Arbeitsschritte werden im Nachhinein dokumentiert. Die Ergebnisse werden in Kapitel 3 dokumentiert.

## 2.2. Projektorganisation

Das Projekt wurde ressourcentechnisch wie folgt aufgeteilt:

### Vorbereitungsaufgaben

- Funktionen für den BCD-zu-7-Segment-Decoder ermitteln –> Simon Eisele
- Konzept für den Algorithmus zur Umwandlung einer 8 Bit-Binärzahl in 3 BCD-Ziffern –> Janis Aebischer
- LE kennelernen und über die Möglichkeiten dieser Software informieren. –> Janis Aebischer, Simon Eisele

### Aufgaben

- BCD-zu-7-Segment Decoder –> Simon Eisele
- Ripple-Carry-Addierer –> Simon Eisele
- Borrow-Bit-Subtrahierer –> Simon Eisele
- Ergebnisumschalter –> Janis Aebischer
- Binär-zu-BCD Umwandler –> Janis Aebischer

### Sonstiges

- Dokumentation –> Janis Aebischer, Simon Eisele
- Erstellen KV-Diagramme –> Janis Aebischer
- Aufsetzen von LaTeX –> Simon Eisele
- Erstellen Testprotokolle –> Janis Aebischer, Simon Eisele

## 2.3. Verwendete Materialien, Werkzeuge und Software

**Entwerfen und Testen der Schaltungen** Für den Entwurf der verschiedenen Schaltungen wurde die Software Logisim-evolution (LE) [1] verwendet. Mit dieser lassen sich die Schaltungen aus Grundgattern aufbauen, sowie auch simulieren. Für die praktischen Tests wurde das Leguan-Board (LB) [2] der BFH verwendet.

**Projektbericht** Der Projektbericht wurde mithilfe von LaTeX verfasst. Um bereits eine passende Grundlage zu haben, wurde eine Vorlage der BFH verwendet und entsprechend unserer Bedürfnisse angepasst. [5]

## 2.4. Test- und Verifikationsverfahren

Jede entwickelte Schaltung wird in der Simulation und auf der Hardware überprüft. Die Simulation erfolgt in LE. Auf dem LB erfolgt die Verifikation der Funktion manuell durch betätigen der DIP-Schalter und visuell über die 7-Segement-Anzeigen. Als Anzeige für *overflow* oder *underflow* werden die verbauten LED's auf dem LB verwendet. Für den Test grösserer Schaltungen eignen sich vorwiegend Werte, mit welchen die Sonderfälle abgedeckt werden können. Diese sind Überlauf, Unterlauf und Grenzwerte der BCD-Darstellung. Die Ergebnisse sind in Kapitel 3 ersichtlich. Alle Tests wurden zuerst in Excel festgehalten und anschliessend in LaTeX übernommen.

## 2.5. Spezifische Methoden pro Baugruppe

Spezifische Methodik für bestimmte Baugruppen, falls etwas nicht dem Standardschema folgt.

### 2.5.1. BCD-zu-7-Segment Decoder

**7-Segment-Anzeige** Die 7-Segment-Anzeige besteht aus sieben einzeln ansteuerbaren Leuchtsegmenten, welche in Form einer Acht angeordnet sind. Durch unterschiedliche Ansteuerung dieser acht Segmente lassen sich die Ziffern 0 bis 9 darstellen. Die Segmente werden mit den Buchstaben A bis G bezeichnet. Zusätzlich gibt es noch ein 9. Segment, mit welchem sich ein Punkt nach der Ziffer darstellen lässt. Dieses wird aber im folgenden ignoriert, da es für die Aufgaben nicht benötigt wird und irrelevant ist.

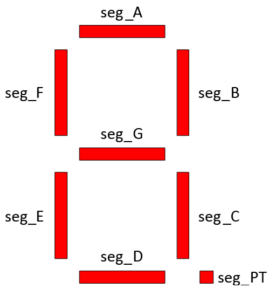



Abbildung 1: Anordnung der Segmente einer 7-Segment-Anzeige [3, Abb. 1]

**Wahrheitstabelle** Aus den Abbildungen 1 und 2 ist ersichtlich, welche Segmente A bis G zur Darstellung der einzelnen Ziffern aktiviert werden müssen. Zur Ansteuerung der 7-Segment-Anzeige werden vier Bits (ein Nibble) verwendet, womit die Dezimalzahlen 0 bis 15 codierbar sind. Da die Anzeige nur die Ziffern 0 bis 9 darstellen kann, wurden die verbleibenden Werte gemäß Aufgabenstellung [3] als „don't care" definiert.

Abbildung 2: Darstellung der Ziffern 0 bis 9 auf der 7-Segment-Anzeige [3, Abb. 2]

**KV-Diagramm** In der Aufgabenstellung sind spezifische Normalformen verlangt. Diese sind entweder konjunktiv oder disjunktiv, welche genau ist aus der Aufgabenstellung zu entnehmen. [3, pp. 4-5]

**Beschränkter Gattervorrat** Die aus den KV-Diagrammen abgeleiteten Normalformen (siehe Tabelle 2) wurden gemäß den Vorgaben der Aufgabenstellung [3, pp. 4–5] angepasst. Für die Realisierung der Schaltungen stand nur ein begrenzter Satz an Grundgattern zur Verfügung. So durfte beispielsweise das Segment A ausschließlich mit den Gattern XOR und OR umgesetzt werden [3, p. 4]. Die Umformungen der Normalformen erfolgten mithilfe der Booleschen Algebra und der De-Morgan-Theoreme.

### 2.5.2. Ripple-Carry-Addierer

Um den Ripple-Carry-Addierer zu realisieren, muss zuerst der Volladdierer gebaut werden. Anschliessend wird die erstellte Baugruppe für das Realisieren der Ripple-Carry-Addierer Baugruppe verwendet. Der Volladdierer wird nicht einzeln in einem Testprotokoll getestet, da er im kombinierten Test der übergeordneten Baugruppe automatisch getestet wird.

Volladdierer Der Ripple-Carry-Addierer besteht aus mehreren, hintereinandergeschalteten Volladdierern. Ein Volladdierer verrechnet dabei jeweils drei Bits miteinander. Je ein Bit der jeweiligen Summanden, sowie ein Übertragsbit des vorherigen Volladdierers. Das Ganze funktioniert also gleich, wie auch die schriftliche Addition. Der Volladdierer liefert dabei die Summe, welche als Ergebnis an die jeweilige Stelle kommt, sowie ein Übertragsbit, welches an den Volladdierer der nächsten Stelle übergeben wird.

Ripple-Carry-Addierer Der Ripple-Carry-Addierer wird für die Addition zweier Binärzahlen verwendet. Er besteht aus mehreren Volladdierern, und kann für eine unbegrenzte Anzahl Bits verwendet werden. Der Name „Ripple“ kommt daher, da bei der Berechnung jeweils ein Übertragsbit von einem Volladdierer zum nächsten durchgereicht wird. Dadurch steigt aber auch die Latenz mit zunehmender Anzahl Bits, da jeder Volladdierer auf den Übertrag des letzten warten muss, bevor die Berechnung durchgeführt werden kann. [6]

8-Bit-Addierer Aus dem entwickelten Volladdierer lassen sich nun beliebig grosse Addierer realisieren. Gemäss Aufgabenstellung [3] soll ein 8-Bit-Addierer gebaut werden. Dies funktioniert durch Aneinanderreihung der Volladdierer und jeweiliger Übergabe des Übertrags auf den darauffolgenden Volladdierer. Der Übertragsausgang des letzten Volladdierers wurde an den Ausgang „Overflow“ angeschlossen. Falls also die Summe der Addition zu gross ist, um noch korrekt dargestellt werden zu können, wird dies rückgemeldet.

### 2.5.3. Borrow-Bit-Subtrahierer

Um den Borrow-Bit-Subtrahierer zu realisieren, muss zuerst der Vollsubtrahierer gebaut werden. Anschliessend wird die erstellte Baugruppe für das Realisieren der Borrow-Bit-Subtrahierer Baugruppe verwendet. Der Vollsubtrahierer wird nicht einzeln in einem Testprotokoll getestet, da er im kombinierten Test der übergeordneten Baugruppe automatisch getestet wird.

Vollsubtrahierer Der Borrow-Bit-Subtrahierer besteht aus mehreren hintereinandergeschalteten Vollsubtrahierern. Jeder Vollsubtrahierer verarbeitet ein Bit des Minuenden, ein Bit des Subtrahenden, sowie das Borrow-Bit (Übertragsbit) des vorherigen Vollsubtrahierers. Der Vollasubtrahierer liefert dabei das Differenzbit, welches als Ergebnis an die jeweilige Stelle kommt, sowie ein Borrow-Bit, welches an den Vollsubtrahierer der nächsten Stelle übergeben wird, falls von dieser ein Bit geborgt werden muss. Die Funktionsweise entspricht damit der schriftlichen Subtraktion, bei der, bei Bedarf, von der nächsthöheren Stelle „ausgeliehen“ wird.

8-Bit-Subtrahierer Aus dem entwickelten Vollsubtrahierer lassen sich nun beliebig grosse Subtrahierer realisieren. Gemäss Aufgabenstellung [3] soll ein 8-Bit-Subtrahierer gebaut werden. Dies funktioniert durch Aneinanderreihung der Vollsubtrahierer und

jeweiliger Übergabe des Übertrags auf den darauffolgenden Vollsubtrahierer. Der Übertragsausgang des letzten Vollsubtrahierers wurde an den Ausgang „Underflow“ angeschlossen. Falls also die Differenz der Subtraktion kleiner als 0 ist, wird dies rückgemeldet, so dass klar ist, dass das angezeigt Ergebnis nicht die korrekte Lösung der Rechnung ist.

### 2.5.4. Ergebnisumschalter

Mit dem gleichen Prinzip wie auch bei den vorherigen Schaltungen, ist der 8-Bit-Multiplexer aus mehreren 1-Bit-Multiplexern aufgebaut. Daher muss zuerst der 1-Bit-Multiplexer realisiert werden. Dieser wird nicht einzeln in einem Testprotokoll getestet, da er im kombinierten Test der übergeordneten Baugruppe automatisch getestet wird.

### 2.5.5. Binär-zu-BCD Umwandler

Eine 7-Segment-Anzeige kann maximal 10 Ziffern anzeigen. Bei einer 8-Bit-Binärzahl kann somit nur ein Bruchteil ihres Wertes wiedergegeben werden. Damit eine 8-Bit-Binärzahl in ihrer gesamtheitlichen Grösse wiedergegeben werden kann, sind 3 Ziffern nötig. Aus der 8-Bit-Binärzahl muss also für jede Ziffer der entsprechende BCD-Wert ermittelt werden. Wie genau dies gemacht wird kann aus der Vorbereitungsaufgabe entnommen werden. (Siehe Anhang A.2) Durch geeignetes Verknüpfen der erstellten Add3-Blöcke lässt sich die gewünschte Funktion erstellen.

Add3 Sobald eine Zahl >=5 ist, wird der Wert 3 (0011 in binär) zu dieser Zahl dazugerechnet.

# 3. Resultate

## 3.1. BCD-zu-7-Segment-Decoder

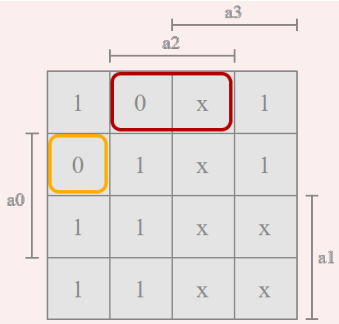
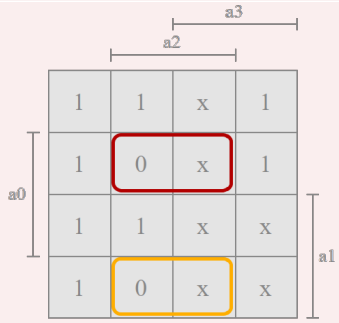
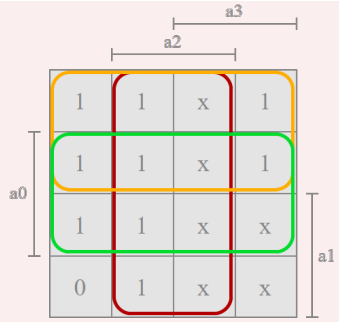
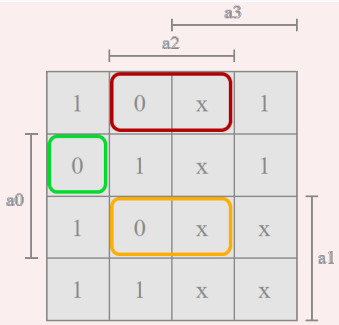
### 3.1.1. Wahrheitstabelle

Tabelle 1: Wahrheitstabelle Siebensegmentanzeige

| Input | a3 | a2 | a1 | a0 | seg_A | seg_B | seg_C | seg_D | seg_E | seg_F | seg_G |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 6 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 8 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 10 | 1 | 0 | 1 | 0 | 0 | - | - | - | - | - | - |
| 11 | 1 | 0 | 1 | 1 | 0 | - | - | - | - | - | - |
| 12 | 1 | 1 | 0 | 0 | 0 | - | - | - | - | - | - |
| 13 | 1 | 1 | 0 | 1 | 0 | - | - | - | - | - | - |
| 14 | 1 | 1 | 1 | 0 | 0 | - | - | - | - | - | - |
| 15 | 1 | 1 | 1 | 1 | 0 | - | - | - | - | - | - |

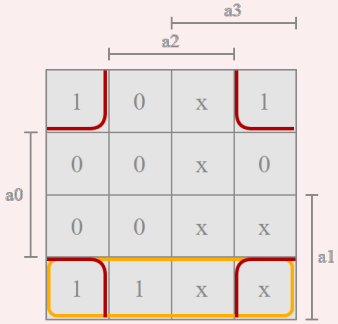
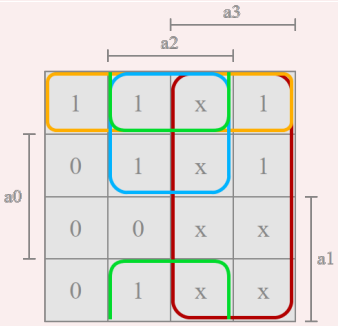
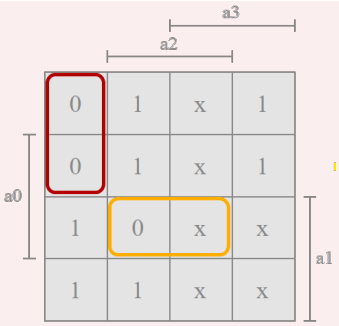
### 3.1.2. KV-Diagramme

Tabelle 2: KV-Diagramme und Normalfunktion der verschiedenen Segmente

| Segment | KV-Diagramm | Funktion |
|---|---|---|
| A |  | $Y = (a_0 + a_1 + \overline{a_2}) \cdot (\overline{a_0} + a_1 + a_2 + a_3)$ |
| B |  | $Y = (\overline{a_0} + a_1 + \overline{a_2}) \cdot (a_0 + \overline{a_1} + \overline{a_2})$ |
| C |  | $Y = a_0 + \overline{a_1} + a_2$ |
| D |  | $Y = (a_0 + a_1 + \overline{a_2}) \cdot (\overline{a_0} + \overline{a_1} + \overline{a_2}) \cdot (\overline{a_0} + a_1 + a_2 + a_3)$ |

Fortsetzung auf nächster Seite...

... Fortsetzung von Tabelle 2

| Segment | KV-Diagramm | Funktion |
|---|---|---|
| E |  | $Y = \overline{a_0} \cdot \overline{a_2} + \overline{a_0} \cdot a_1$ |
| F |  | $Y = a_3 + \overline{a_0} \cdot \overline{a_1} + \overline{a_0} \cdot a_2 + \overline{a_1} \cdot a_2$ |
| G |  | $Y = (a_1 + a_2 + a_3) \cdot (\overline{a_0} + \overline{a_1} + \overline{a_2})$ |

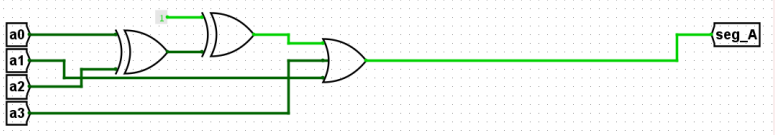
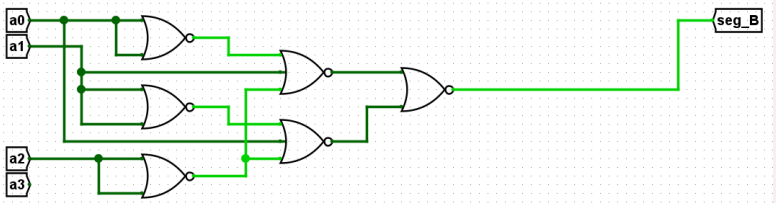
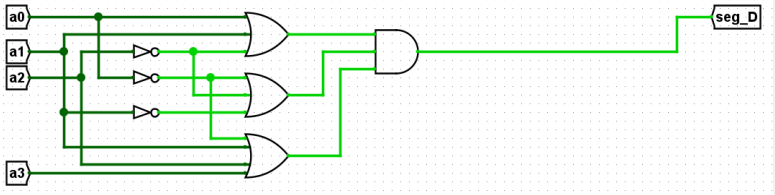
### 3.1.3. Beschränkter Gattervorrat

Tabelle 3: Funktionen mit beschränktem Gattervorrat

| Segment | Gattervorrat | Funktion |
|---|---|---|
| A | XOR und OR | $Y = a_1 + a_3 + ((a_0 \oplus a_2) \oplus 1)$ |
| B | NOR | $Y = \overline{\overline{\overline{a_0 + a_0} + a_1 + \overline{a_2 + a_2}} + \overline{a_0 + \overline{a_1 + a_1} + \overline{a_2 + a_2}}}$ |
| C | AND, OR und NOT | $Y = a_0 + \overline{a_1} + a_2$ |
| D | AND, OR und NOT | $Y = (a_0 + a_1 + \overline{a_2}) \cdot (\overline{a_0} + \overline{a_1} + \overline{a_2}) \cdot (\overline{a_0} + a_1 + a_2 + a_3)$ |
| E | AND und XOR | $Y = (a_0 \oplus 1) \cdot (((a_1 \oplus 1) \cdot a_2) \oplus 1)$ |
| F | NAND | $Y = \overline{\overline{a_3 \cdot a_3} \cdot \overline{\overline{a_0 \cdot a_0} \cdot \overline{a_1 \cdot a_1}} \cdot \overline{\overline{a_0 \cdot a_0} \cdot a_2} \cdot \overline{\overline{a_1 \cdot a_1} \cdot a_2}}$ |
| G | AND2, OR2 und NOT | $Y = ((a_1 + a_2) + a_3) \cdot ((\overline{a_0} + \overline{a_1}) + \overline{a_2})$ |

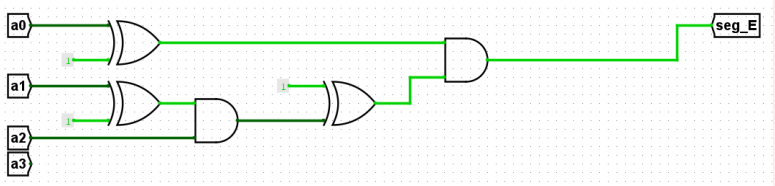
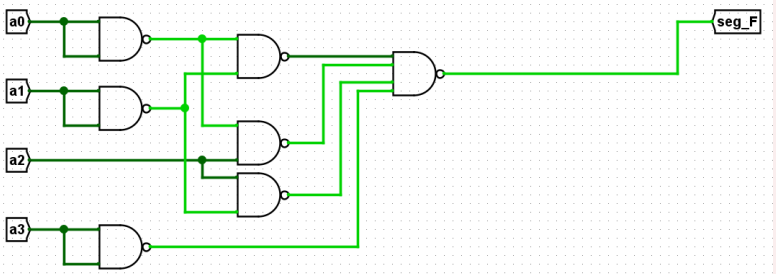
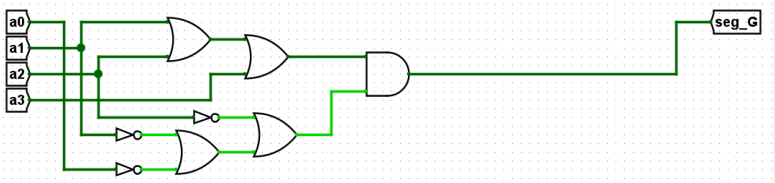
### 3.1.4. Implementierung in Logisim-Evolution

Tabelle 4: Implementierung der Funktionen in LE

| Schaltungsbezeichnung | Schaltung |
|---|---|
| segmentA |   |
| segmentB |   |
| segmentC |   |
| segmentD |   |

Fortsetzung auf nächster Seite...

... Fortsetzung von Tabelle 4

| Schaltungsbezeichnung | Schaltung |
|---|---|
| segmentE |  |
| segmentF |  |
| segmentG |  |

### 3.1.5. Test der Schaltungen

Tabelle 5: Testprotokoll Siebensegmentanzeige

| Input | | Segment A | | | Segment B | | | Segment C | | | Segment D | | | Segment E | | | Segment F | | | Segment G | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Digitalwert | Binärwert | E | LE | LB | E | LE | LB | E | LE | LB | E | LE | LB | E | LE | LB | E | LE | LB | E | LE | LB |
| 0 | 0000 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0001 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0010 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 3 | 0011 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 4 | 0100 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 0101 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6 | 0110 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 0111 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 1000 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1001 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 10 | 1010 | - | 1 | 1 | - | 1 | 1 | - | 0 | 0 | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 |
| 11 | 1011 | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 | - | 0 | 0 | - | 1 | 1 | - | 1 | 1 |
| 12 | 1100 | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 | - | 0 | 0 | - | 0 | 0 | - | 1 | 1 | - | 1 | 1 |
| 13 | 1101 | - | 1 | 1 | - | 0 | 0 | - | 1 | 1 | - | 1 | 1 | - | 0 | 0 | - | 1 | 1 | - | 1 | 1 |
| 14 | 1110 | - | 1 | 1 | - | 0 | 0 | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 |
| 15 | 1111 | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 | - | 0 | 0 | - | 0 | 0 | - | 1 | 1 | - | 0 | 0 |

*Legende:* E = Erwartet, LE = Simuliert in Logisim-evolution, LB = Getestet auf Leguan-Board

## 3.2. Ripple-Carry-Addierer

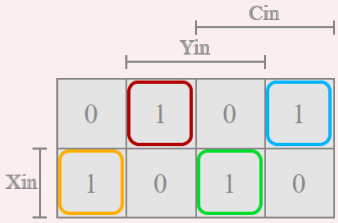
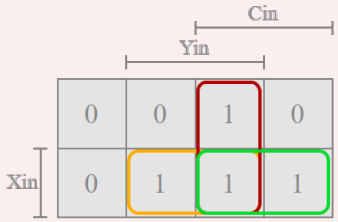
### 3.2.1. Wahrheitstabelle

Tabelle 6: Wahrheitstabelle Volladdierer

| Xin | Yin | Cin | Sum | Cout |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |

### 3.2.2. KV-Diagramme

Tabelle 7: KV-Diagramme und konjunktive Normalfunktion der Ausgänge des Volladdierers

| Output | KV-Diagramm | konjunktive Normalform |
|---|---|---|
| Sum |   | $Sum = X_{in} \oplus Y_{in} \oplus C_{in}$ |
| Cout |   | $Cout = (X_{in} \cdot Y_{in}) + (X_{in} \cdot C_{in}) + (Y_{in} \cdot C_{in})$ |

### 3.2.3. Implementierung in Logisim-evolution

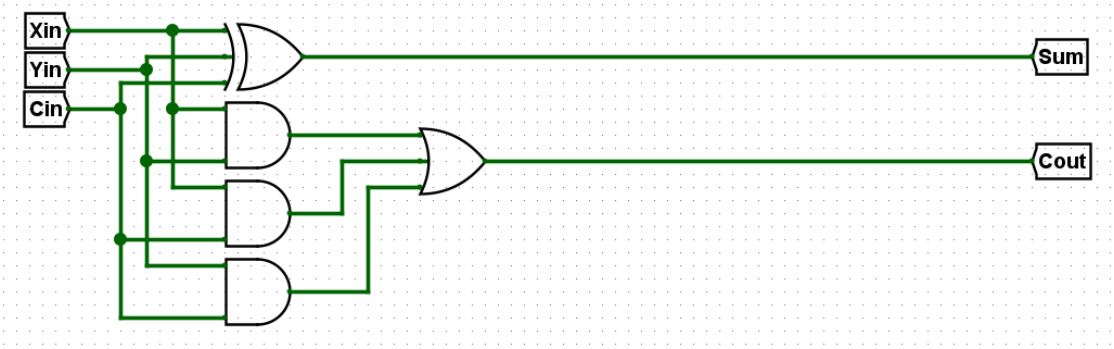



Abbildung 3: Implementierung des Volladdierers in LE

### 3.2.4. 8-Bit-Addierer

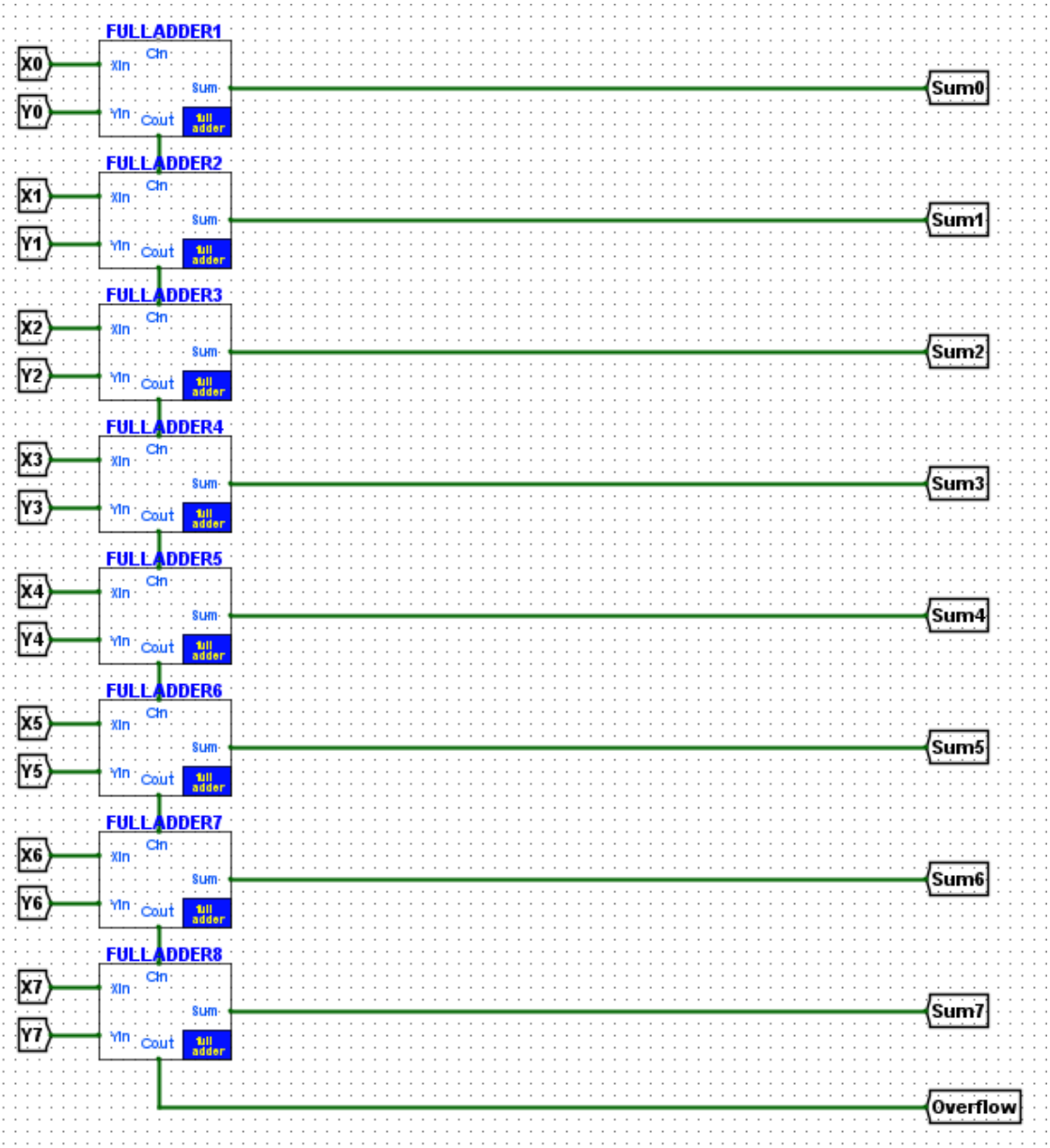



Abbildung 4: Aufbau des 8-Bit-Addierers in LE

### 3.2.5. Test der Schaltungen

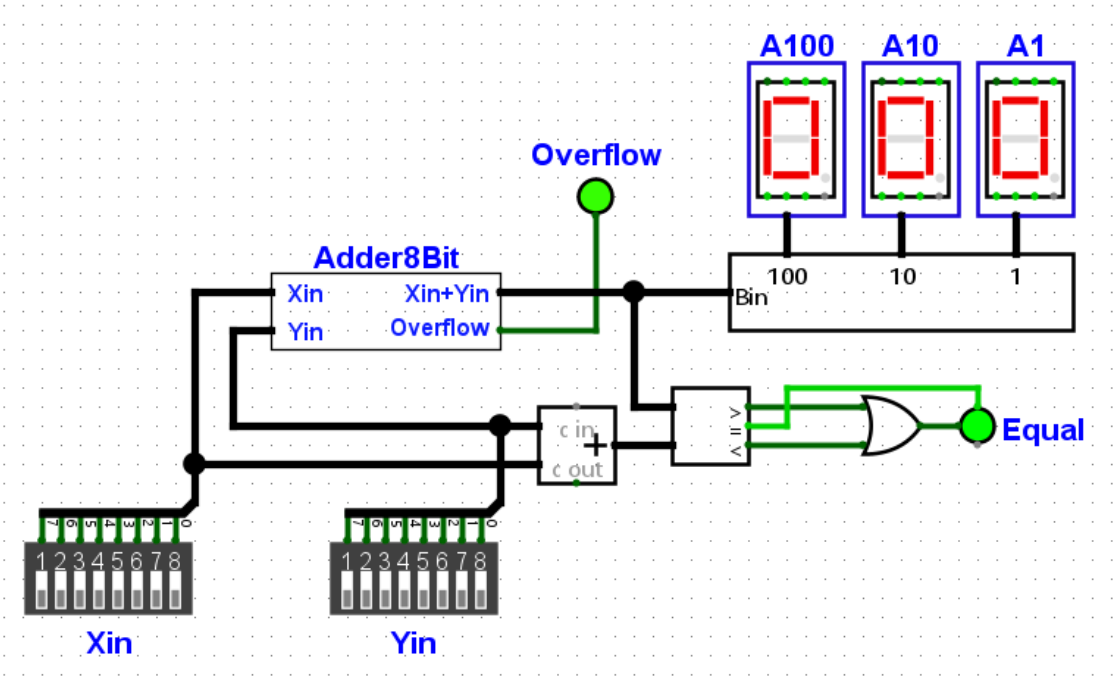



Abbildung 5: Schaltung AdderTestBench in LE

Tabelle 8: Testprotokoll 8-Bit-Addierer

| Input Summand 1 | | Input Summand 2 | | Erwarteter Wert | Resultat Simulation | Resultat Leguan-Board |
|---|---|---|---|---|---|---|
| Digitalwert | Binärwert | Digitalwert | Binärwert | | | |
| 0 | 0000 0000 | 0 | 0000 0000 | 0 | 0 | 0 |
| 255 | 1111 1111 | 0 | 0000 0000 | 255 | 255 | 255 |
| 255 | 1111 1111 | 1 | 0000 0001 | overflow+0 | overflow+0 | overflow+0 |
| 255 | 1111 1111 | 255 | 1111 1111 | overflow+254 | overflow+254 | overflow+254 |
| 240 | 1111 0000 | 16 | 0001 0000 | overflow+0 | overflow+0 | overflow+0 |
| 15 | 0000 1111 | 1 | 0000 0001 | 16 | 16 | 16 |
| 170 | 1010 1010 | 85 | 0101 0101 | 255 | 255 | 255 |
| 127 | 0111 1111 | 18 | 0000 0001 | 128 | 128 | 128 |
| 98 | 0110 0010 | 8 | 0000 1000 | 106 | 106 | 106 |
| 2543 | 1111 1110 | 1 | 0000 0001 | 255 | 255 | 255 |

## 3.3. Borrow-Bit-Subtrahierer

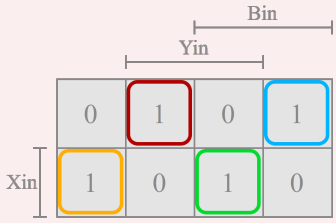
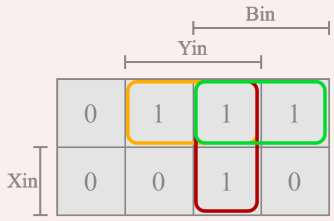
### 3.3.1. Wahrheitstabelle

Tabelle 9: Wahrheitstabelle Vollsubtrahierer

| Xin | Yin | Bin | Sum | Bout |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |

### 3.3.2. KV-Diagramme

Tabelle 10: KV-Diagramme und konjunktive Normalfunktion der Ausgänge des Vollsubtrahierers

| Output | KV-Diagramm | konjunktive Normalform |
|---|---|---|
| Sum |  | $Sum = X_{in} \oplus Y_{in} \oplus B_{in}$ |
| Bout |  | $Bout = (\overline{X_{in}} \cdot Y_{in}) + (\overline{X_{in}} \cdot B_{in}) + (Y_{in} \cdot B_{in})$ |

### 3.3.3. Implementierung in Logisim-evolution

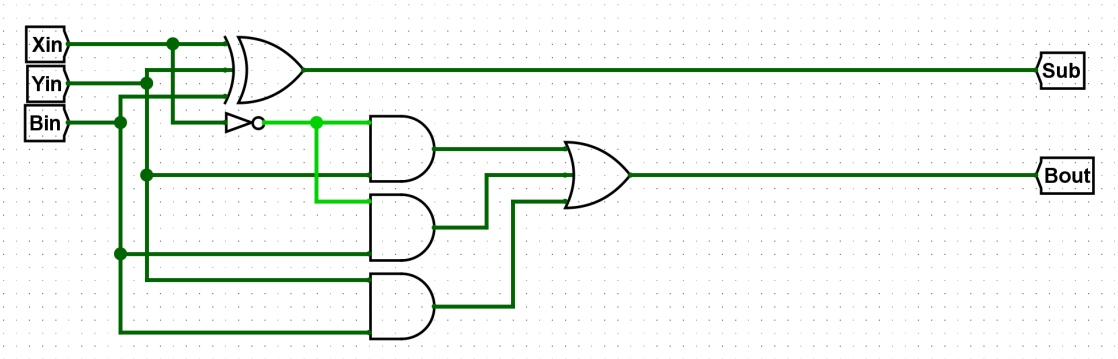



Abbildung 6: Implementierung des Vollsubtrahierers in LE

### 3.3.4. 8-Bit-Subtrahierer

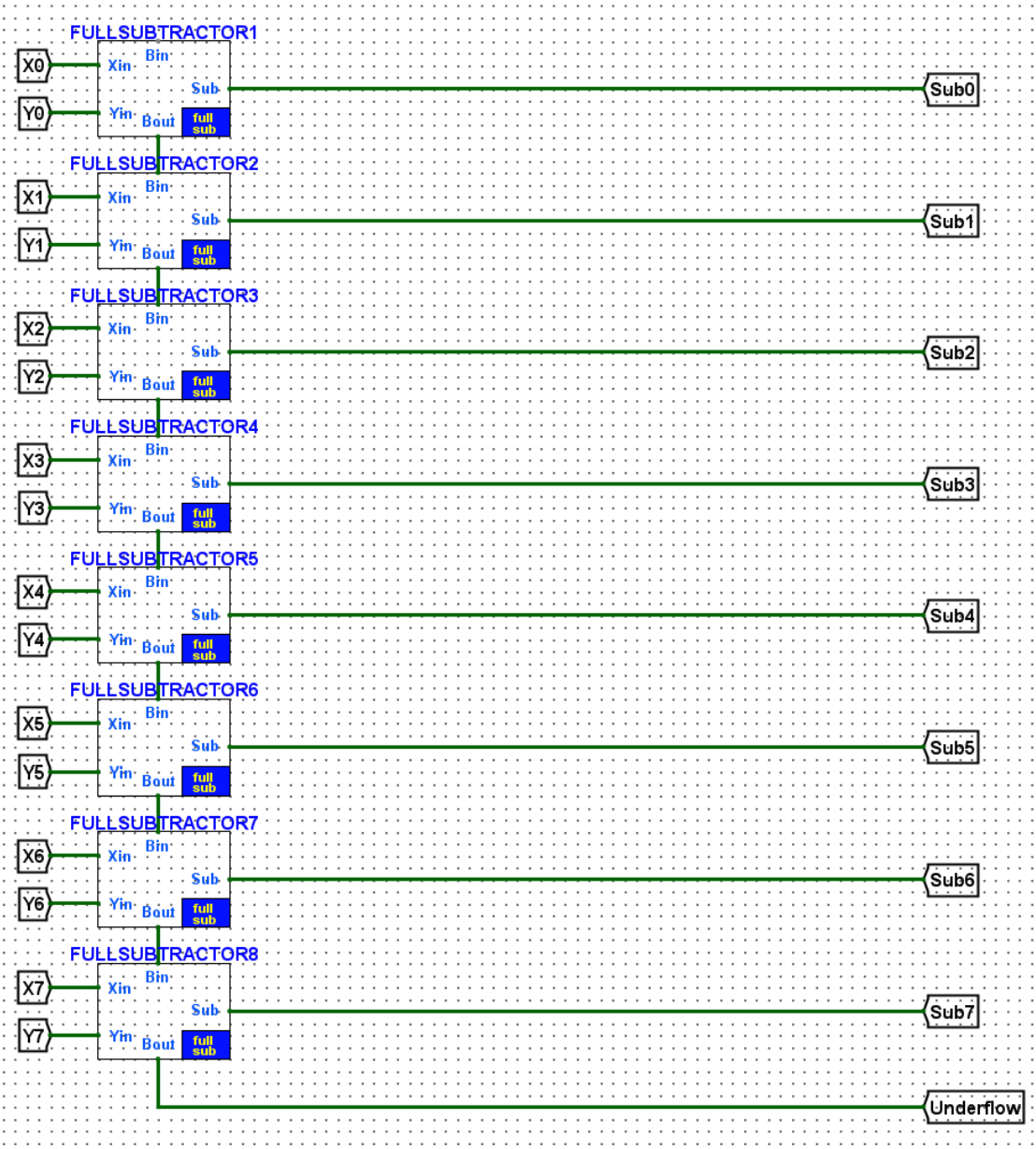



Abbildung 7: Aufbau des 8-Bit-Subtrahierers in LE

### 3.3.5. Test der Schaltungen

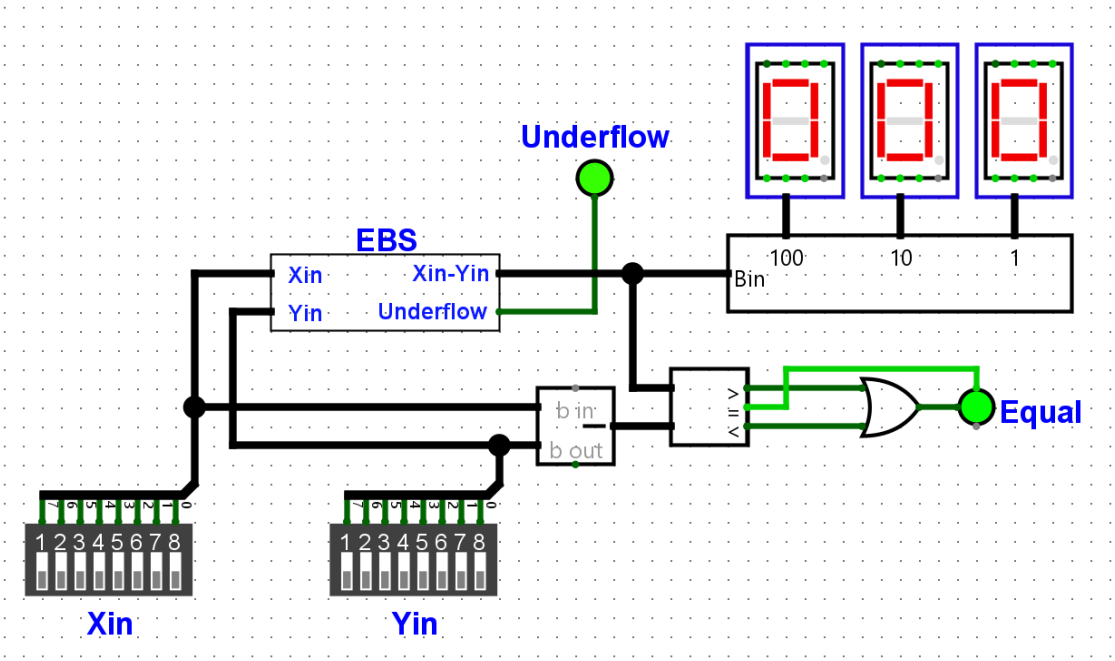



Abbildung 8: Schaltung SubtractorTestBench in LE

Tabelle 11: Testprotokoll 8-Bit-Subtrahierer

| Input Minuend | | Input Subtrahend | | Erwarteter Wert | Resultat Simulation | Resultat Leguan-Board |
|---|---|---|---|---|---|---|
| Digitalwert | Binärwert | Digitalwert | Binärwert | | | |
| 0 | 0000 0000 | 0 | 0000 0000 | 0 | 0 | 0 |
| 5 | 0000 0101 | 0 | 0000 0000 | 5 | 5 | 5 |
| 0 | 0000 0000 | 1 | 0000 0001 | underflow+255 | underflow+255 | underflow+255 |
| 255 | 1111 1111 | 255 | 1111 1111 | 0 | 0 | 0 |
| 255 | 1111 0000 | 1 | 0000 0001 | 254 | 254 | 254 |
| 1 | 0000 0001 | 255 | 1111 1111 | underflow+2 | underflow+2 | underflow+2 |
| 128 | 1000 0000 | 128 | 1000 0000 | 0 | 0 | 0 |
| 16 | 0001 0000 | 1 | 0000 0001 | 15 | 15 | 15 |
| 200 | 1100 1000 | 150 | 1001 0110 | 50 | 50 | 50 |
| 0 | 0000 0000 | 255 | 1111 1111 | underflow+1 | underflow+1 | underflow+1 |

## 3.4. Ergebnisumschalter

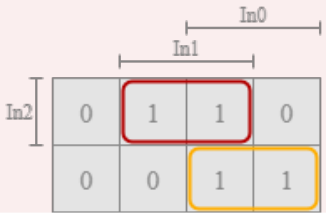
### 3.4.1. Wahrheitstabelle

Tabelle 12: Wahrheitstabelle 1-Bit Multiplexer

| In_2 SelectSub | In_1 SubIn | In_0 AddIn | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

### 3.4.2. KV-Diagramme

Tabelle 13: KV-Diagramm des 1-Bit Multiplexer

| Output | KV-Diagramm | konjunktive Normalform |
|---|---|---|
| Y |  | $Y = (In_0 \cdot \overline{In_2}) + (In_1 \cdot In_2)$ |



### 3.4.3. Implementierung in Logisim-evolution


Abbildung 9: Implementierung des 1-Bit Multiplexer in LE

### 3.4.4. 8-Bit-multiplexer

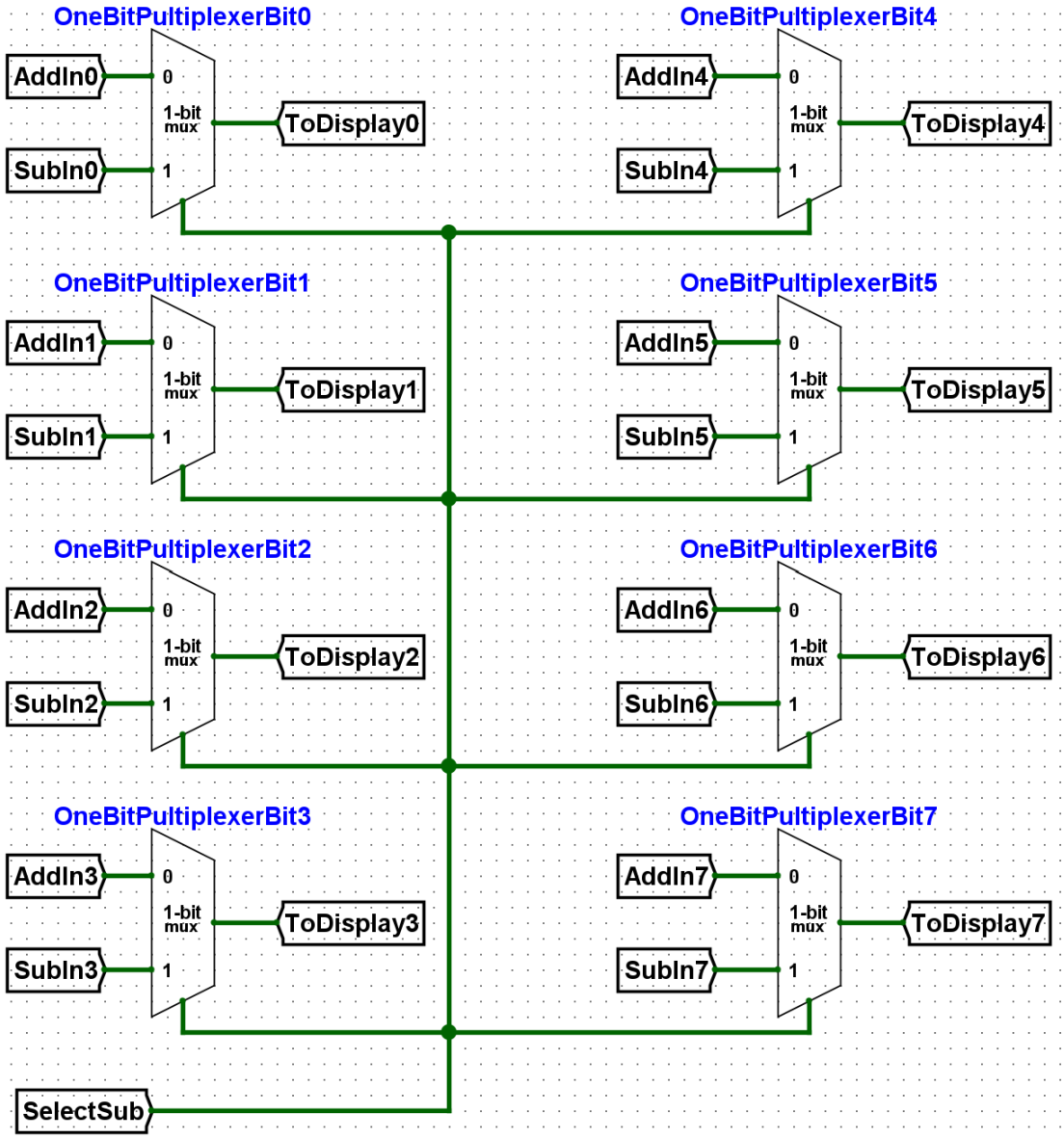



Abbildung 10: Implementierung des 8-Bit Multiplexer in LE

### 3.4.5. Test der Schaltungen

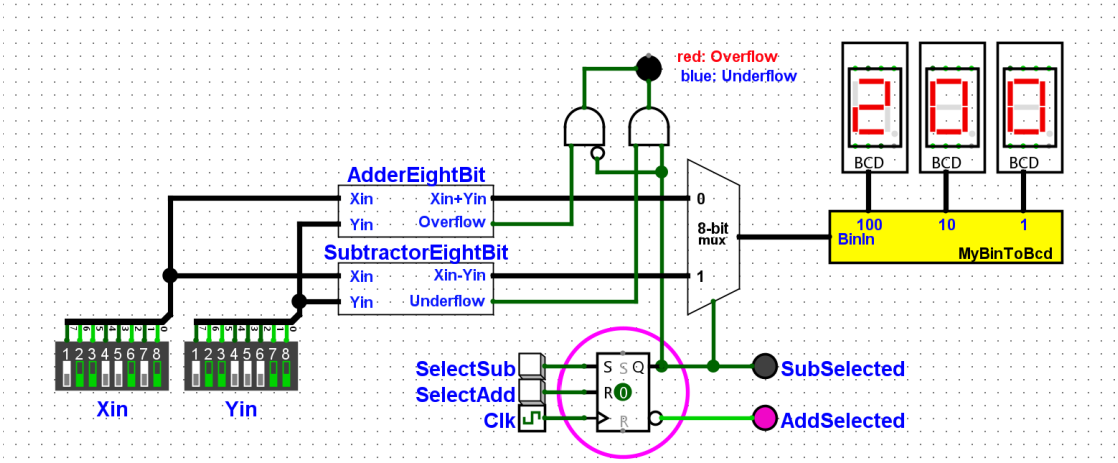



Abbildung 11: Testen des 8-Bit Multiplexer in LE

Tabelle 14: Testprotokoll 8-Bit Multiplexer

| Input Xin | | Input Yin | | SelectSub | Erwarteter Wert | Resultat Simulation | Resultat Leguan-Board |
|---|---|---|---|---|---|---|---|
| Digitalwert | Binärwert | Digitalwert | Binärwert | | | | |
| 60 | 0011 1100 | 30 | 0001 1110 | 0 | 90 | 90 | 90 |
| 15 | 0000 1111 | 16 | 0001 0000 | 0 | 31 | 31 | 31 |
| 101 | 0110 0101 | 99 | 0110 0011 | 0 | 200 | 200 | 200 |
| 255 | 1111 1111 | 255 | 1111 1111 | 0 | overflow+254 | overflow+254 | overflow+254 |
| 60 | 0011 1100 | 30 | 0001 1110 | 1 | 60 | 60 | 60 |
| 15 | 0000 1111 | 16 | 0001 0000 | 1 | underflow+255 | underflow+255 | underflow+255 |
| 101 | 0110 0101 | 99 | 0110 0011 | 1 | 2 | 2 | 2 |
| 255 | 1111 1111 | 255 | 1111 1111 | 1 | 0 | 0 | 0 |

## 3.5. Binär-zu-BCD-Konverter

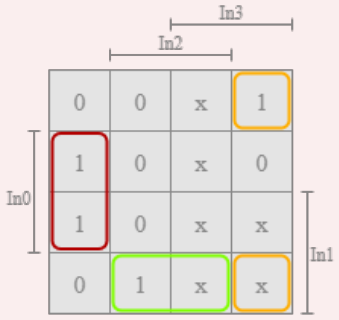
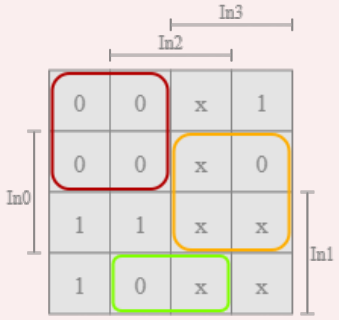
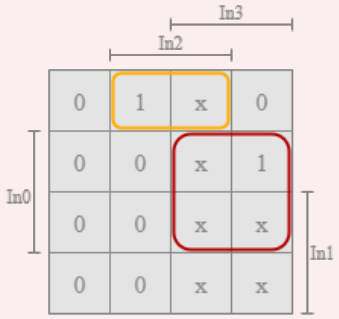
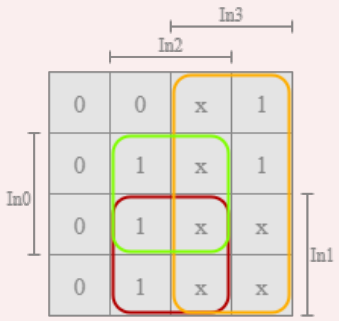
### 3.5.1. Wahrheitstabelle

Tabelle 15: Wahrheitstabelle Add3-Block

| Input | In_3 | In_2 | In_1 | In_0 | Out_3 | Out_2 | Out_1 | Out_0 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 8 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 10 | 1 | 0 | 1 | 0 | - | - | - | - |
| 11 | 1 | 0 | 1 | 1 | - | - | - | - |
| 12 | 1 | 1 | 0 | 0 | - | - | - | - |
| 13 | 1 | 1 | 0 | 1 | - | - | - | - |
| 14 | 1 | 1 | 1 | 0 | - | - | - | - |
| 15 | 1 | 1 | 1 | 1 | - | - | - | - |

### 3.5.2. KV-Diagramme

Tabelle 16: KV-Diagramme und Normalform der Ausgänge des Add3-Blocks

| Output | KV-Diagramm | Normalform |
|---|---|---|
| Out_0 LSB |   | $Out_0 = (In_0 \cdot \overline{In_2} \cdot \overline{In_3}) + (In_3 \cdot \overline{In_2} \cdot \overline{In_0}) + (In_2 \cdot \overline{In_0} \cdot In_1)$ |
| Out_1 |   | $Out_1 = (\overline{In_0} + \overline{In_3}) \cdot (In_0 + In_3) \cdot (\overline{In_2} + \overline{In_1} + In_0)$ |
| Out_2 |   | $Out_2 = (In_2 \cdot \overline{In_1} \cdot \overline{In_0}) + (In_3 \cdot In_0)$ |
| Out_3 MSB |   | $Out_3 = (In_2 \cdot In_0) + (In_2 \cdot In_1) + In_3$ |

### 3.5.3. Implementierung in Logisim-evolution

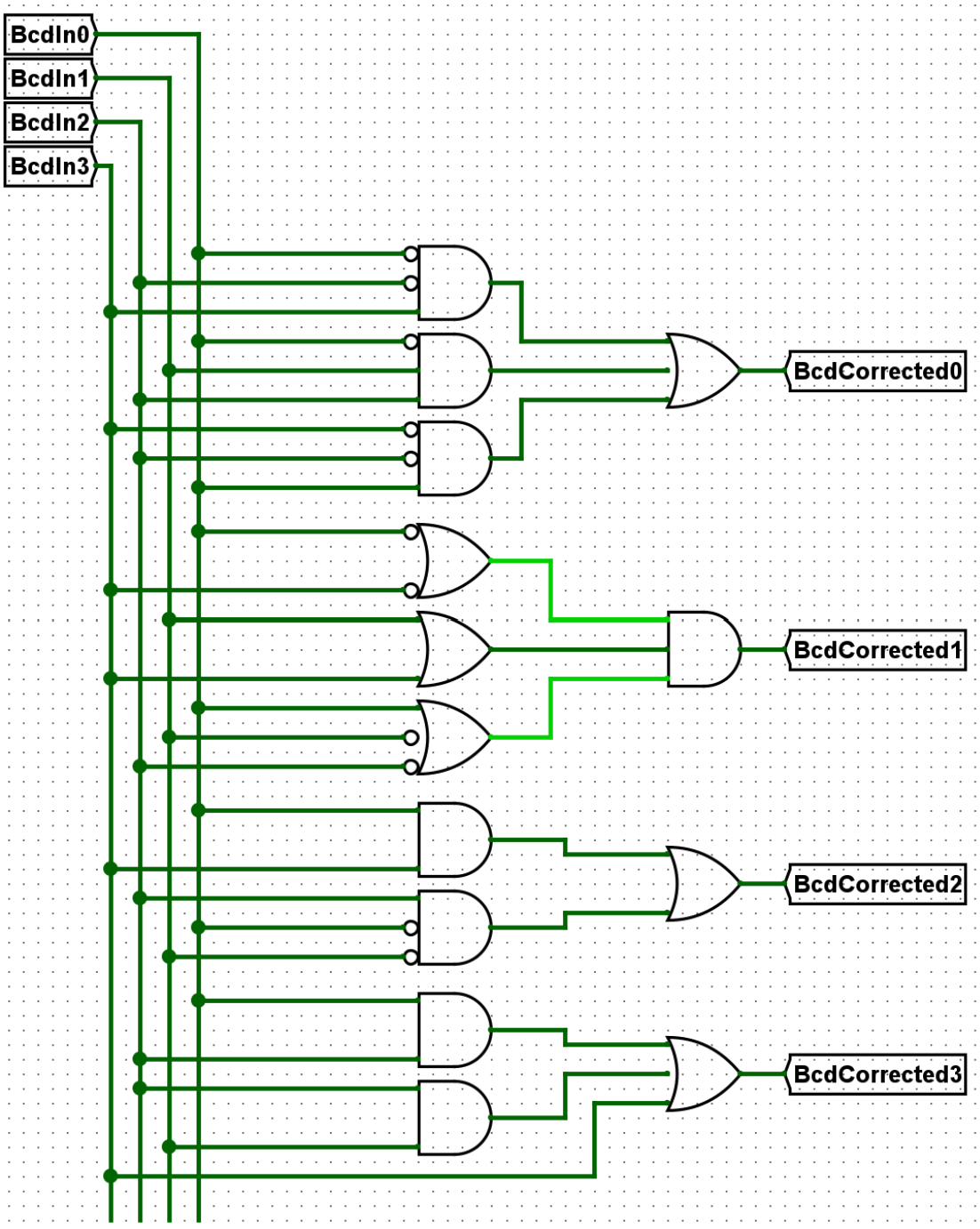



Abbildung 12: Implementierung des Add3-Blocks in LE

### 3.5.4. Gesamtschaltung

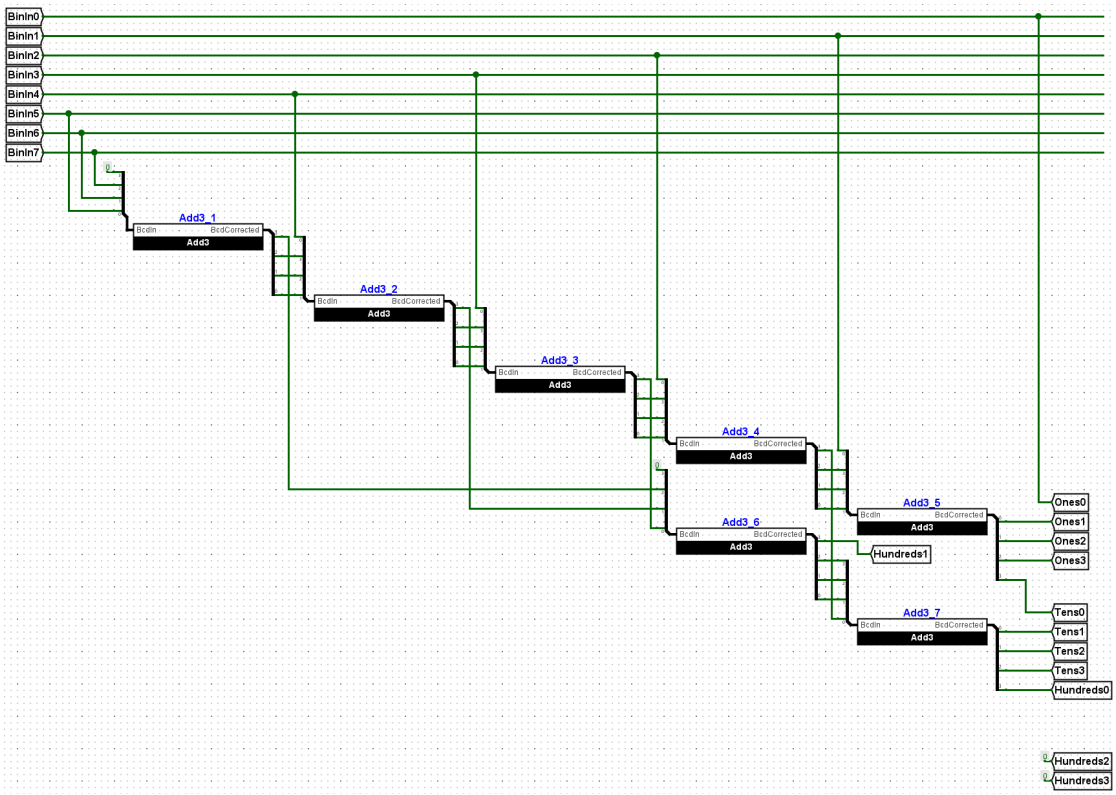



Abbildung 13: Implementierung des selbst erstellten Binär zu BCD Wandlers in LE

### 3.5.5. Test der Schaltungen

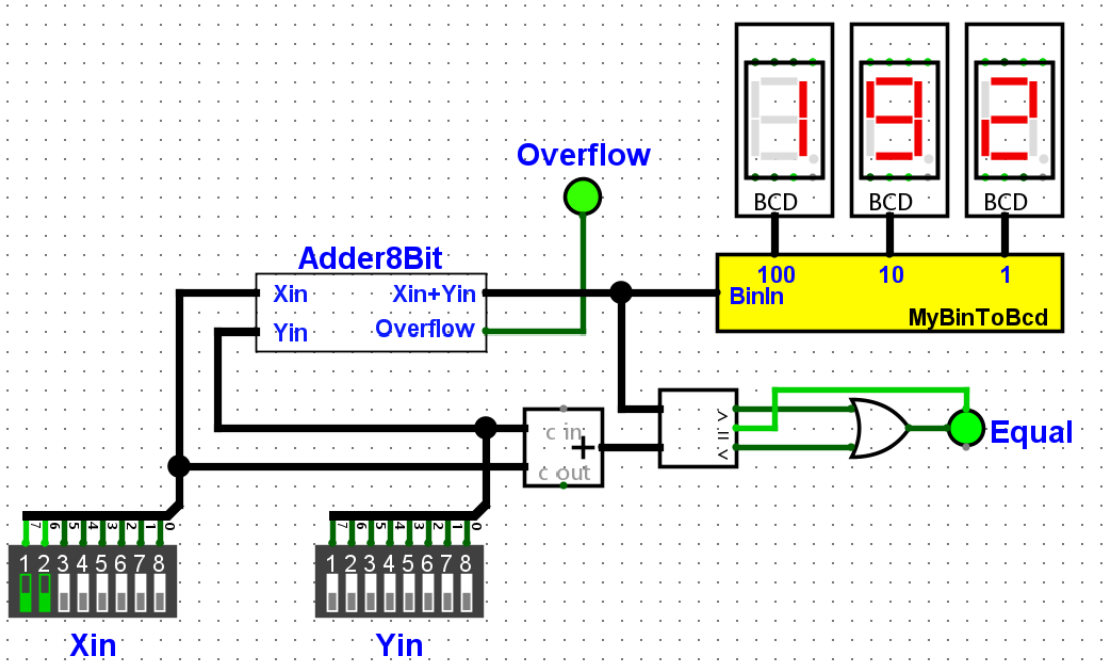



Abbildung 14: Testen des Umwandlers in LE

Tabelle 17: Testprotokoll Binär zu BCD Umwandler

| Input | | Erwarteter Wert | Resultat Simulation | Resultat Leguan-Board |
|---|---|---|---|---|
| Digitalwert | Binärwert | | | |
| 0 | 0000 0000 | 0 | 0 | 0 |
| 1 | 0000 0001 | 1 | 1 | 1 |
| 9 | 0000 1001 | 9 | 9 | 9 |
| 10 | 0000 1010 | 10 | 10 | 10 |
| 98 | 0110 0010 | 98 | 98 | 98 |
| 99 | 0110 0011 | 99 | 99 | 99 |
| 100 | 0110 0100 | 100 | 100 | 100 |
| 199 | 1100 0111 | 199 | 199 | 199 |
| 200 | 1100 1000 | 200 | 200 | 200 |
| 255 | 1111 1111 | 255 | 255 | 255 |

# 4. Diskussion

Dies ist unser erstes Laborprojekt an der BFH. Alle geforderten Funktionen wurden erreicht. Da wir das ganze Dokument in LaTeX geschrieben haben, haben wir viel darüber gelernt. Der Aufbau des Dokuments ist uns zuerst nicht gut gelungen. Es viel uns schwer die Kapitel Methoden und Resultate zu unterscheiden. Daher bedurf es im Nachhinein einer zeitintensiven Korrektur. Durch dieses Projekt haben wir auch unser Wissen im Bereich LE und LB aufbauen, sowie erweitern können. Für das nächste Mal müssen wir viel früher schon stärker darauf achten, was in welchen Teil des Dokuments kommt. Da wir jetzt aber schon erste Erfahrungen mit LaTeX gesammelt, sowie auch eine passende Struktur erarbeitet haben, wird das Erstellen des Berichts beim nächsten Mal vermutlich schon viel flüssiger laufen. Insgesamt war das Projekt sehr zeitintensiv, da auch die Erwartungshaltung für uns noch nicht ganz klar ist, und noch ein bischen herausgespürt werden muss.

## Selbstständigkeitserklärung

Ich bestätige, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel angefertigt habe. Sämtliche Textstellen, die nicht von mir stammen, sind als Zitate gekennzeichnet und mit dem genauen Hinweis auf ihre Herkunft versehen. Ich bestätige weiterhin, dass ich bei der Erstellung dieser Arbeit durchgehend steuernd gearbeitet habe und von einer KI erzeugte Texte bzw. Textfragmente nicht unreflektiert übernommen habe.

29. Oktober 2025

J. Aebischer

S. Eisele

# Literatur

[1] C. Burch u. a., *Logisim-Evolution*, Version 3.9.0, Aug. 2024. besucht am 25. Okt. 2025. Adresse: https://github.com/logisim-evolution/logisim-evolution

[2] „Home - Leguan Developer's Guide," besucht am 25. Okt. 2025. Adresse: https://leguan.ti.bfh.ch/

[3] D. Tamsel, T. Kluter und T. Mähne, *Kombinatorische Logik Projekt 4 BTE5213 - Elektrische Grundlagen: Laborprojekte und technische Berichte*, BFH-TI-Biel/Bienne, 2025.

[4] M. Jacomet und T. Kluter, *Digitaltechnik Grundlagen*, BFH-TI-Biel/Bienne, 11. Sep. 2023.

[5] „Introduction — LaTeX Manual Documentation," besucht am 1. Nov. 2025. Adresse: https://latex.ti.bfh.ch/

[6] M. Mano und M. Ciletti, *Digital Design With an Introduction to the Verilog HDL, VHDL, and SystemVerilog*, Sixth Edition. California State University.

# Abbildungsverzeichnis

# Tabellenverzeichnis

## Glossar

LaTeX Ein Textsatzsystem für wissenschaftliche Dokumente

## Akronyme

BCD Binary-Coded Decimal

BFH Berner Fachhochschule

FPGA Field-Programmable Gate Array

KV Karnaugh-Veitch

LB Leguan-Board

LE Logisim-evolution

# A. Anhang

## A.1. Logisim-Evolution Schaltpläne

Die folgenden Dateien sind als Download verfügbar:

- Projekt 4: project4.circ

## A.2. PDF

Die folgenden Dateien sind als Download verfügbar:

- Konzept Binär-zu-BCD: concept_binary_to_bcd.pdf