

# ULog - Lehrgang

Dieser Text soll zeigen, wie digitale Schaltungen aufgebaut werden und wie sie funktionieren.

## 1. Grundsaltungen

Simulationen von grundlegenden Schaltungen findet man im ULog-Programm unter den **Menüpunkten „Schaltungen“, „Gatter“**.

Das einfachste Gatter ist das **NOT-Gatter**, dessen Simulation auch gleichzeitig zeigt, wie ein Transistor funktioniert: Ein Transistor (FET) besteht aus einem Halbleiter (senkrecht breites Rechteck) und einer Kondensatorplatte (links) mit einem Anschluss, das **Gate**. Der Halbleiter besitzt überschüssige Elektronen, die den Strom transportieren können (n-leitend). Wird das Gate gegenüber dem Halbleiter negativ geladen, so stößt sie die Elektronen ab, und es entsteht im Halbleiter ein Bereich (dunkelrot), der keine überzähligen Elektronen enthält. Der Stromtransport über den Halbleiter ist damit unterbrochen. Man sagt, dass „der Transistor sperrt“. Ist hingegen das Gate positiv geladen, so ist der Bereich, der keine überzähligen Elektronen enthält, sehr klein. Die Elektronen können sich damit ungehindert im Halbleiter bewegen, „der Transistor leitet“. Da das Aufladen des Gates nur wenig Energie kostet, aber große Ströme steuern kann, kann der Transistor zur Verstärkung von Strömen genutzt werden.

Nun ist in der Schaltung ein Widerstand nachgeschaltet, der den Transistor mit der 5V-Spannung verbindet. Der Widerstand bewirkt, dass bei leitendem Halbleiter die Spannung an den Halbleiteranschlüssen (oben und unten) zusammenbricht, also fast 0V ist. Dies geschieht bei positiver Gatespannung. Ist die Gatespannung 0V oder kleiner, so sperrt der Transistor, und der obere Transistoranschluss ist über den Widerstand mit 5V verbunden. D.h., die Spannung am Transistor verhält sich entgegengesetzt zur Spannung an des Gates.

Stellt man eine 1 durch eine positive Spannung dar und eine 0 durch eine Spannung nahe 0V, so macht die Schaltung aus einer 1 an dem Gate eine 0 am Transistorausgang und umgekehrt, **negiert also den Gateeingang**.

Das **AND-Gatter** besteht aus zwei hintereinander geschaltete Transistoren, die über einen Widerstand mit 0V verbunden sind. Haben beide Transistorgates eine positive Spannung und leiten damit beide Transistoren, so liegt an dem Widerstand eine Spannung nahe 5V an. Ist hingegen eine Gatespannung 0V, so sperrt der entsprechende Transistor, und der Widerstand hat keine Verbindung mehr zu 5V. Die Spannung am Widerstand beträgt also nahe 0V.

Die Schaltung liefert damit nur bei 1 an beiden Basen eine 1 am Ausgang, sonst eine 0. Betrachtet man einen Eingang als Steuerung, so entscheidet dieser darüber, ob ein Signal vom anderen Eingang zum Ausgang durchgekoppelt wird.

Das **AND-Gatter** wirkt dann **als Schalter**.

Das **OR-Gatter** besteht aus zwei Transistoren, die beide an einen Widerstand gekoppelt sind und diesen mit 5V verbinden können. Ist einer der Transistoren leitend, so liegt an dem Widerstand fast 5V an. Nur wenn beide Transistoren sperren, ist der Widerstand nicht mehr mit 5V verbunden. Wenn also an beiden Basen 0 anliegt, liefert der Ausgang - obere Leitung am Widerstand - eine 0, sonst immer eine 1.

Das OR-Gatter koppelt beide Eingänge auf den Ausgang, es schaltet die Eingänge zusammen.

**Aus NOT-, AND- und OR-Gattern lässt sich jede digitale Schaltung z.B. mittels QMC-Verfahrens aufbauen.**

Ein NAND-Gatter ist ein And-Gatter, bei dem der Widerstand den oberen Transistor mit 5V verbindet. Durch geschickte Schaltungen lassen sich NOT-, AND- und OR-Gatter durch NAND-Gatter ersetzen.

**Multiplexer (MUX)** und **Demultiplexer (DMUX)** sind aus den Grundbausteinen zusammengesetzt: Ein Multiplexer hat mehrere Eingänge, von denen einer, abhängig von Werten auf **Adressleitungen**, auf den Ausgang durchgeschaltet wird. Ein Demultiplexer hat nur einen Eingang, der dann aufgrund von Adresswerten auf einen von mehreren Ausgängen durchgeschaltet wird. Beide Schaltungen spielen eine wichtige Rolle bei Speicherbausteinen und CPU's:

Mit Hilfe von Multiplexern können Funktionseinheiten, z.B. Addierer ausgewählt werden, die Daten bearbeiten sollen. Demultiplexer können zur Adressierung von Speichern und Ausgabekanälen genutzt werden.

Die Funktion von Speicherbausteinen und -Chips wird unter den **Menüpunkten „Schaltungen“, „Speicher“** behandelt..

Ein **RS-FlipFlop** besteht aus zwei p-leitenden Transistoren mit jeweils zwei Basen. p-leitend ist ein Transistor, wenn der Halbleiter zu wenig Elektronen hat. In diesem Fall saugt er Elektronen aus einer negativen Spannungsquelle. Die Elektronen wandern im Halbleiter von Fehlstelle zu Fehlstelle, bis sie am zweiten Anschluss von einer positiven Spannung aus dem Transistor gesaugt werden. Der Transistor verhält sich, als ob in ihm positive Ladungsträger unterwegs wären und damit umgekehrt zu einem n-leitenden Transistor. Die Schaltung ist so angelegt, dass durch Rückkopplungen

der Ausgänge auf ein Gate des anderen Transistors die Zustände „leitend“ und „sperrend“ so lang erhalten bleiben, wie die Versorgungsspannung 5V anliegt und die Eingangsspannungen erhalten bleiben oder beide 0V werden. Damit lässt sich die Schaltung zum Speichern von Daten nutzen. Man kann die Funktionsweise am Modell testen. RS-FlipFlops reagieren schnell, benötigen aber eine Versorgungsspannung. Ein Transistor mit zwei Basen verhält sich wie ein AND-Gatter. das RS-FlipFlops besteht somit aus 2 AND-Gattern.

Zum dauerhaften Speichern sind **EEProms** besser geeignet, da sie keine dauernde Versorgungsspannung benötigen. Sie werden heute z.B. in SD-Karten und USB-Speichern eingesetzt. Ein EEPROM ist ein Transistor, bei dem das Gate aus zwei Platten besteht. Die innere ist isoliert und steuert je nach Ladung den Durchfluss durch den Transistor. Die äußere Platte dient zum Auf- und Entladen der inneren Platte, indem eine Spannung angelegt wird, die die Isolierung zwischen den Platten (durch Quanteneffekte) überwindet. Der Zustand der inneren Platte bleibt auch erhalten, wenn keine Versorgungsspannung anliegt. Durch das Auf- und Entladen wird mit der Zeit die Isolierung zwischen den Platten geschädigt, so dass ein EEPROM nur eine begrenzte Lebensdauer hat. Die Funktion ähnelt der eines NOT-Gatters und kann in ULog simuliert werden.

**Chip-Aufbau** zeigt Aufbau und Funktion von Speicherchips. Ein Chip steuert nach Anlegen einer Adresse über ein DMUX eine Zeile mit Speicherelementen an, die dann über Datenleitungen beschrieben oder ausgelesen werden können. Die Schaltung ist auf die wesentlichen Teile reduziert, um die Funktion klarer zu verdeutlichen. So erfolgt die Speicherung von Daten mittels Kondensatoren.

In der Praxis müssen weitere Transistoren eingebaut werden, um z.B. zu verhindern, dass beim Lesen eines Speicherelements andere umprogrammiert werden. Zudem wird bei jedem Lesevorgang die Ladung der Kondensatoren verringert, so dass nach gewissen Zeitabständen die Ladungen über zusätzliche Schaltungen wiederaufgefrischt werden müssen (**Refreshing**). Günstiger ist es, die Kondensatoren durch EEPROMs zu ersetzen, die dann mit einem zusätzlichen Gate für die Aktivierung ausgestattet sind.

## 2. Komplexe Schaltungen

Um komplizierte Schaltungen zu entwickeln, sollte man zunächst durch eine Wertetabelle bestimmen, wie Ein- und Ausgänge voneinander abhängen. Die neue Schaltung lässt sich z.B. durch das QMC-Verfahren standardmäßig aus NOT-, AND- und OR-Gattern zusammensetzen. Bei Bedarf kann sie anschließend nach gewissen Kriterien optimiert werden. Die Schaltung wird in einem Schaltplan durch Kreise (NOT-Gatter), Rechtecke (AND-, OR-Gatter) und Datenleitungen dargestellt. Versorgungsleitungen (0V und 5V) fehlen im Schaltplan.

Ein **Rechteck mit &** im Inneren stellt ein **AND-Gatter** dar, **mit  $\geq 1$**  ein **OR-Gatter**.

ULog bietet zwei Möglichkeiten, eine Schaltung zu entwickeln: 1. manuell und 2. QMC-Verfahren.

Das **QMC-Verfahren** dient dazu, zu einer Wertetabelle eine Schaltung zu entwickeln, deren Verhalten durch die Tabelle beschrieben wird. Die Tabelle verknüpft dabei Eingangswerte der Schaltung mit Ausgangswerten.

Wählt man die **Menüpunkte „Schaltplan“, „QMC-Verfahren“**, so ist zunächst die Zahl der Ein- und Ausgangsleitungen anzugeben, danach die Wertetabelle. Nach Betätigen der ESC-Taste wird automatisch eine passende Schaltung erzeugt, und man kann wählen, ob der zugehörige Schaltplan angezeigt oder ausgedruckt werden soll.

Hinweis: Das Verfahren selbst wird in einem späteren Abschnitt an einem Beispiel erläutert.

Will man **manuell** einen **Schaltplan** eingeben und austesten, so wähle man die **Menüpunkte „Schaltungen“, „Schaltplan“**. Es erscheint ein Schaltplan mit vier Eingangsleitungen, die an 7 AND-Gattern gekoppelt werden können, darunter 2 OR- und 2 AND-Gatter, die dann an vier Ausgangsleitungen angeschlossen sind. Die Ausgänge der oberen AND-Gatter können als Eingänge an die unteren Gatter gekoppelt werden. Vor den Gattern findet man jeweils ein kleines gelbes Rechteck, das die Gatter von den zugehörigen Eingangsleitungen trennt. Klickt man ein gelbes Rechteck an, so kann man wählen, ob die Eingangsleitung mit dem Gatter direkt verbunden werden soll oder über ein NOT-Gatter („**Negieren**“). Bei Bedarf kann man mit „**Unterbrechen**“ die Verbindung einer Eingangsleitung zum Gatter wieder beseitigen. Es ist möglich, Ausgangsleitungen der Schaltung mit den Eingangsleitungen zu verbinden, um Werte zurück zu koppeln. Dies könnte bei der Simulation von Speicherbausteinen nützlich sein.

Ist die Schaltung komplett, so kann man statt „Verbinden“ den **Menüpunkt „Testen“** wählen. Der Rechner durchläuft dann mit einer einstellbaren Zeitverzögerung alle möglichen Eingangswerte und ermittelt die durch die Schaltung erzeugten Ausgangswerte. Fällt der Test nicht zur Zufriedenheit aus, so kann die Schaltung geändert und erneut getestet werden.

## 3. QMC-Verfahren

Vorgegeben sei die folgende Wertetabelle mit 3 Eingängen und 2 Ausgängen:

|           |     |           |    |
|-----------|-----|-----------|----|
| Eingänge: | ABC | Ausgänge: | DE |
|           | 000 |           | 00 |
|           | 001 |           | 01 |
|           | 010 |           | 01 |
|           | 011 |           | 10 |
|           | 100 |           | 01 |
|           | 101 |           | 10 |
|           | 110 |           | 10 |
|           | 111 |           | 11 |

Man berechne die Zahlenwerte der Eingänge im Dezimalsystem und ergänze die Tabelle:

|           |       |           |    |
|-----------|-------|-----------|----|
| Eingänge: | ABC   | Ausgänge: | DE |
|           | 000 0 |           | 00 |
|           | 001 1 |           | 01 |
|           | 010 2 |           | 01 |
|           | 011 3 |           | 10 |
|           | 100 4 |           | 01 |
|           | 101 5 |           | 10 |
|           | 110 6 |           | 10 |
|           | 111 7 |           | 11 |

Nun teile man die Tabelle und sammle zu jedem Ausgang in einer eigenen Liste die Eingänge auf, die für diesen Ausgang eine 1 liefern:

E: 001 1 ; 010 2 ; 100 4 ; 111 7                      D: 011 3 ; 101 5 ; 110 6 ; 111 7

Man unterstreiche **Koppelterme**, das sind Eingangswerte, die in mehreren Teiltabellen gleichzeitig zu finden sind, in unserem Fall also 111:

E: 001 1 ; 010 2 ; 100 4 ; 111 7                      D: 011 3 ; 101 5 ; 110 6 ; 111 7

In jeder Teiltabelle suche man die Eingangswerte, die sich nur an einer Stelle unterscheiden. Diese Werte werden zu neuen Werten zusammengefasst, wobei die Unterschiedsstelle durch ein - ersetzt wird. Zusammengefasste Eingänge werden durch Abhaken oder Kursivschrift kenntlich gemacht:

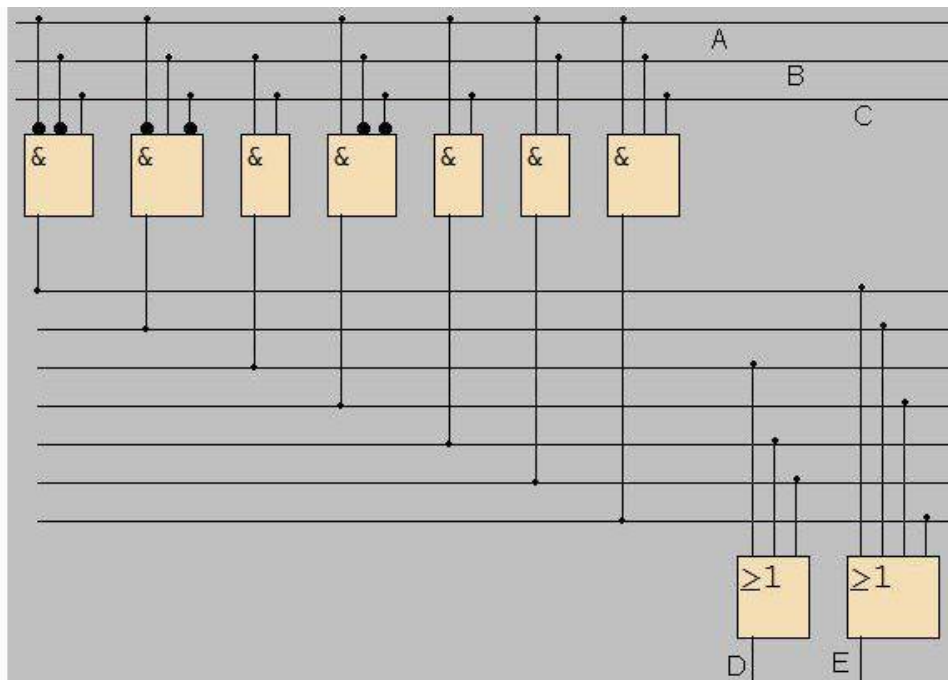
E: 001 1 ; 010 2 ; 100 4 ; 111 7                      D: *011 3* ; *101 5* ; *110 6* ; 111 7 ; -11 3,7 ; 1-1 5,7 ; 11- 6,7

Alle nicht kursiven Terme und alle Koppelterme, die am weitesten rechts stehen, werden für die Schaltung benötigt. Es ist möglich, dass Terme gleiche Eingangsbelegungen zusammenfassen. Um die Schaltung zu reduzieren, werden alle oben ausgewählten Terme in einer Tabelle als Zeilenüberschriften eingetragen, dazu alle ursprünglichen Eingangswerte für die Ausgänge D und E als Spaltenüberschriften. Durch ein x wird in der Tabelle markiert, ob ein Endterm aus einem bestimmten Eingangswert entstanden ist:

| Ausgang | E     |       |       |       | D     |       |       |       |
|---------|-------|-------|-------|-------|-------|-------|-------|-------|
| Term \  | 001 1 | 010 2 | 100 4 | 111 7 | 011 3 | 101 5 | 110 6 | 111 7 |
| 001 1   | x     |       |       |       |       |       |       |       |
| 010 2   |       | x     |       |       |       |       |       |       |
| 100 4   |       |       | x     |       |       |       |       |       |
| 111 7   |       |       |       | x     |       |       |       | x     |
| -11 3,7 |       |       |       |       | x     |       |       | x     |
| 1-1 5,7 |       |       |       |       |       | x     |       | x     |
| 11- 6,7 |       |       |       |       |       |       | x     | x     |

Es stellt sich die Frage, ob man aus der Tabelle Zeilen fortlassen kann, so dass noch in jeder Spalte ein x stehen bleibt und dass die übrigen Terme in der linken Spalte insgesamt die wenigsten Nullen und Einsen haben. In diesem Beispiel bleiben alle Zeilen erhalten. Jeder Term, der links übrig bleibt, entspricht einem AND-Gatter, bei dem Eingänge mit einer 0 negiert und mit einer 1 direkt verbunden werden. Eingänge mit einem - werden ignoriert. Pro Ausgang D oder E wird ein OR-Gatter benötigt, das diesen Ausgang erzeugt und in das als Eingänge die Ausgänge der AND-Gatter eingekoppelt werden, die jeweils für D und E noch übrig geblieben sind. Man kann mathematisch beweisen, dass das Verfahren die schnellste Schaltung liefert mit dem geringsten Aufwand an Transistoren.

In unserem Beispiel ergibt sich als Schaltung:



Die Schaltung addiert drei Eingangswerte zu einem Übertrag (D) und einem Ergebnisbit (E). Man erhält also einen

**1-Bit-Addierer**. Benutzt man mehrere dieser Schaltungen und koppelt in den dritten Eingang den Übertrag des vorausgehenden Addierers, so erhält man einen **Addierer für zwei mehrstelligen Dualzahlen** (vgl. CPU-Modell).

Hinweise:

1. Durch Verneinen von Ausgängen lassen sich evtl. Schaltungen mit gleicher Reaktionszeit, aber geringeren Aufwand erzeugen.
2. Verneinte Eingänge erfordern keine NOT-Gatter vor AND-Gattern, wenn man dort n-leitende Transistoren durch p-leitende ersetzt.

Verneinte Ausgänge gewinnt man, wenn man den Widerstand zwischen Transistor und 0V entfernt und dann zwischen Transistor und 5V schaltet.

Insgesamt ist der Aufwand durch die Gesamtzahl der Eingänge der AND- und OR-Gatter einer Schaltung gegeben.

## 5. CPU-Simulation

Dargestellt sind die wesentlichen Bestandteile einer CPU:

Das **Rechenwerk mit Registern (RALU)** im linken, weiß umrandeten Rechteck und dem Steuerwerk im rechten Rechteck. Das Rechenwerk verknüpft Zahlen zu neuen Ergebnisse, z.B. durch Addition, und kann diese in Speichereinheiten, den Registern ablegen. Alternativ können die Ergebnisse über Datenleitungen an externe Bauteile weitergereicht werden. Die Eingangszahlen können von externen Bauteil oder aus Registern stammen. Bei der Auswahl der Eingangsquellen und zu verwendenden Funktionseinheiten werden MUXe eingesetzt, bei der Bestimmung der Ausgangsziele auch DMUXe.

Als Operationen werden die AND-, die XOR-Verknüpfung und die Addition (ADD) zweier 8-Bit-Zahlen angeboten. Diese Einheiten bestehen aus jeweils 8 Schaltungen, wie sie im Abschnitt „Gatter“ vorgestellt wurden. Es werden zwei Zahlen verknüpft, deren **Quellen mit MUX2 und MUX3** bestimmt werden (vgl. „**Adressierung**“). Die drei Einheiten arbeiten gleichzeitig. Das Mux4 erlaubt, das Ergebnis einer Einheit auszuwählen, das dann weiter verarbeitet werden soll. Über MUX4 können auch Registerinhalte oder Daten aus anderen Quellen ausgewählt werden. **DMUX5** kann dann die Zahl, die MUX4 liefert, in ein Register schreiben oder auf den **externen Datenbus** (oben rechts).

Daten, die über den externen Datenbus transportiert werden, können aus Speichern oder Peripheriegeräten (z.B. Grafikkarte, USB-Port) stammen oder dahin geschickt werden. Um Speicher bzw. Peripheriegeräte auszuwählen, wird eine Adresse benötigt, die mittels **MUX1** bestimmt wird. Die Adresse wird über den externen Adressenbus übermittelt.

Zudem sind Signale nötig, die besagen, ob Daten gespeichert oder gelesen werden können: **CS (Chip-Select)** aktiviert die externe Dateneinheit, **W** schaltet die Einheit auf Einschreiben von Daten (Wert 1) oder der Lieferung von Daten (Wert 0) um.

Die Werte von CS- und W-Leitung stammen aus dem **Steuerwerk** (s.unten), ebenso wie die Adressen für MUXe und DMUX. Diese Daten laufen über die roten Leitungen. Um das Steuerwerk von außen in einen definierten Zustand zu versetzen, gibt es die **RS-Leitung (Reset)**.

Der Datenbus leitet Daten von externen Einheiten zur CPU oder exportiert Daten von der CPU. Damit dies möglich ist, wird ein **Tristate-Gatter (TRI)** benötigt: Das Gatter besteht aus zwei gegengerichteten AND-Gattern. Das erste hat einen Ausgang zum Datenbus, das zweite den Bus als Eingang und Ausgang zur CPU. Es werden zwei Steuerleitungen benötigt. Hat die erste den Wert 1, die zweite 0, so lässt nur das erste Gatter Daten durch, das zweite sperrt. Ist hingegen die erste Steuerleitung 0, die zweite 1, so sperrt das erste AND-Gatter und das zweite leitet Daten hindurch. Sind beide Steuerleitungen 0, so sperren beide AND-Gatter, und der Datenbus ist von der CPU getrennt.

Es gibt noch zwei Leitungen, die vom Rechenwerk zum Steuerwerk führen und dieses beeinflussen, die **E(qual)-** und die **C(arry)-Leitung**. Die beiden Eingänge in die Recheneinheiten werden über XOR miteinander verknüpft. Verknüpft man die einzelnen Stellen des Ergebnisses mit einem OR-Gatter und schickt das verneinte Ergebnis über die **E-Leitung**, so ist deren **Wert 1, wenn die Ausgangszahlen gleich** waren, und 0 sonst. Die Additionsschaltung liefert für jede Stelle einen Übertrag in die nächste. Der **Übertrag von der 8. in die 9.Stelle** wird als **C-Leitung** in das Steuerwerk gekoppelt.

Das **Steuerwerk (CU)** im rechten Rechteck bestimmt, welche Funktion in welcher Reihenfolge ausgeführt wird. Die Ansteuerung des Rechenwerks geschieht dabei durch die roten Leitungen, wie oben beschrieben. Umgekehrt können von außen oder dem Rechenwerk Signale an das Steuerwerk gegeben werden, die entsprechenden Leitungen sind blau gefärbt.

Das Steuerwerk ist in der einfachsten Form aufgebaut: Es besteht aus einem ROM-Speicher, der analog zum Speicher-Chip im Abschnitt „Schaltungen“, „Speicher“ aufgebaut ist. Allerdings sind dabei die geladenen Kondensatoren durch feste Verbindungen ersetzt, die nicht geladenen durch fehlende Leiterverbindungen. ROMs können nur einmal programmiert werden und behalten dann die gespeicherten Werte. Ein Teil der **ROM-Ausgänge** sind über RS-FlipFlops auf die Adressenleitungen des ROMs zurückgekoppelt (schwarze Leitungen). Die restlichen Ausgänge des ROMs sind die Steuerleitungen für das Rechenwerk (rot). Über die Steuerleitungen können auch Zahlen direkt vom Steuerwerk in das Rechenwerk übertragen werden.

Die Adressierung des ROMs setzt sich zusammen aus dem zurückgekoppelten ROM-Ausgängen und der C- und E-Leitung. Die **E-Leitung** liefert die **Einer-Stelle** der ROM-Adresse, die **C-Leitung** die **Zweier-Stelle** und die rückgekoppelten Ausgänge die restlichen Stellen.

Durch die RS-Leitung wird die Adresse mittels AND-Gattern auf 0 gesetzt.

Die Eingangsadresse wählt eine Speicherzeile des ROMs aus, das dann eine Ausgangszahl liefert. Der Ausgang bestimmt eine Aktion des Rechenwerks, das dann mittels E- und C-Leitungen und dem zurückgekoppelten Adressteil die nächste Speicherzeile des ROMs bestimmt. Auf diese Weise kommt ein Programmablauf zustande, mit dem komplexere Aufgaben gelöst werden können. Dadurch, dass alle MUXe und das DMUX gleichzeitig angesprochen werden können, können während eines Taktes (Zeit für die Abarbeitung eines Mikrobefehls) mehrere Aktionen ausgeführt werden. Dabei werden Daten zuerst eingelesen und nach Ausführung von anderen Aktionen gespeichert bzw. auf den Datenbus geschrieben-

Der **Speicherinhalt des ROMs** wird auch als **Mikroprogramm** bezeichnet und besteht in der Praxis aus Dualzahlen. Um das Programmieren des Modells zu vereinfachen sind hier für Teile eines Mikrobefehls Dezimalzahlen vorgesehen. Durch Mikroprogramme können **Assembler-Befehle** realisiert werden, die von außen die CPU steuern können. Mit diesem CPU-Modell können alle gebräuchlichen Assembler-Befehle einer Intel-CPU nachgebildet werden!

Ein **Mikrobefehl** (Inhalt einer Speicherzeile des ROMs) besteht aus (von links nach rechts):

1. einer 8-stelligen Dualzahl, die in das Rechenwerk gekoppelt werden kann
2. Steueradresse für MUX1 (externe Speicheradresse) zwischen 0 und 11
3. jeweils eine Steueradresse für MUX2 und MUX3 (Eingänge für AND, XOR-,ADD-Einheit) zwischen 0 und 11
4. Steueradresse für MUX4 (Ergebnisauswahl) zwischen 0 und 14
5. Steueradresse für DMUX5 (Ziel des Ergebnisses) zwischen 0 und 11 (12, siehe Start des CPU-Modells)
6. vordere Stellen für Adresse des nächsten Mikrobefehls (außer Einer und Zweier) zwischen 0 und 249

Die Werte von CS- und W-Leitung hängen von den Adressen der MUXe und des DMUX ab.

Die **Steueradressen** werden als Zahlen im Dezimalsystem eingegeben, die dann als Dualzahlen interpretiert werden. In der Praxis sind die Adressen reine Dualzahlen. Als Steueradressen können verwendet werden:

- 0 bis 9: Lesen bzw. Beschreiben (DMUX) von Register 0 bis 9
- 10: Lesen bzw. Beschreiben mittels externen Datenbus
- 11: Lesen einer Zahl aus dem Mikrobefehl
- 12: Lesen von Ergebnis der AND-Verknüpfung
- 13: Lesen von Ergebnis der XOR-Verknüpfung
- 14: Lesen von Ergebnis der ADD-Verknüpfung

Beim **Start des CPU-Modells** fragt der Rechner nach „**Adressumschaltung**“. Bejaht man, wird die Schaltung so ergänzt, dass das DMUX Daten als Adresse für die Mikrobefehle an das Steuerwerk sendet. Dies geschieht, wenn man als Steueradresse für DMUX5 12 eingibt. Das neue MUX5 reicht dann automatisch die Daten vom DMUX an das Steuerwerk weiter, ohne dass eine weitere Steueradresse eingegeben werden muss. Dadurch können eventuell Abfragen in Mikroprogrammen eingespart werden.

Beim **Erstellen eines Mikroprogramms** ist zu beachten, dass es **keine Befehle für Verzweigungen** gibt, stattdessen kann man ausnutzen, dass sich die C- und E-Leitung auf die Adressen der Programmzeilen auswirken.

Soll ein Befehl ohne Einfluss von C- und E-Werten ausgeführt werden, so ist er viermal hintereinander zu wiederholen, also z.B.

|      |        |
|------|--------|
| Adr. | B ...  |
| 112  | Befehl |
| 113  | Befehl |
| 114  | Befehl |
| 115  | Befehl |

Sollen zwei Befehle in Abhängigkeit vom E-Wert ausgeführt werden, so sind sie jeweils zweimal zu wiederholen, also z.B.

|      |         |       |
|------|---------|-------|
| Adr. | B ...   |       |
| 112  | Befehl1 | (E=0) |
| 113  | Befehl2 | (E=1) |
| 114  | Befehl1 | (E=0) |
| 115  | Befehl2 | (E=1) |

Befehl1 wird ausgeführt, wenn E=0 ist, Befehl2 bei E=1.

Sollen zwei Befehle in Abhängigkeit vom C-Wert ausgeführt werden, so sind sie jeweils zweimal zu wiederholen, also z.B.

|      |         |       |
|------|---------|-------|
| Adr. | B ...   |       |
| 112  | Befehl1 | (C=0) |
| 113  | Befehl1 | (C=0) |
| 114  | Befehl2 | (C=1) |
| 115  | Befehl2 | (C=1) |

Befehl1 wird ausgeführt, wenn C=0 ist, Befehl2 bei C=1.

**Sprünge** werden durch entsprechende Adressenangaben für den nachfolgenden Befehl realisiert.

Das Eingabefeld für Mikroprogramme kann mit ESC verlassen werden.

Wählt man nach Erstellen eines Mikroprogramms den Programmpunkt „**Testen**“, so kann man zu Beginn Dualzahlen in den Registern eintragen. Nach ESC kann man entscheiden, ob der Programmablauf angezeigt werden soll oder nicht. Bei der Anzeige ist noch anzugeben, wie lang jeweils ein Programmbefehl verzögert werden soll.

Als **Beispiel eines Mikroprogramms** findet man in der Datei **ULog-8088.cpu** einen Interpreter für grundlegende Befehle der Intel-Prozessoren. Die Assembler-Befehle werden wie in der Praxis durch Zahlenkodes dargestellt.

Eine **Übersicht** über die **Assemblerkodes** und der Funktion des **CPU-Modells** ist in der Datei **ULog-Mikrokode** gespeichert.

Wenn man Spaß daran hat, kann man versuchen, Mikroprogramme für die folgenden **Aufgaben** zu schreiben:

1. Man realisiere NOT, also dass die Bits in Register 1 von 0 in 1 und von 1 in 0 umgewandelt werden. Man kann dazu die Beziehung  $XOR\ 1 = NOT$  für ein einzelnes Bit nutzen.
2. Man realisiere OR für Register 1 und 2, Ergebnis in Register 3.
3. Man realisiere eine 16-Bit Addition, wobei der erste Summand aus den Zahlen in Register 1 und 2 und der zweite aus den Zahlen in Register 3 und 4 besteht. Das Ergebnis soll dann auf die Register 5 und 6 aufgeteilt werden.

## 5. Assembler

Der Menüpunkt „Assembler“ erlaubt einfache Maschinenprogramm zu erstellen und zu testen, wobei als Ausgangspunkt das CPU-Modell mit dem 8088-Maschinenkode genommen wird. Die Programme lassen sich auf kommerzielle Intel-Prozessoren übertragen, wenn man über einen entsprechenden Assembler (Ausführungsprogramm) verfügt. Zudem bietet „Assembler“, „Vergleich“ die Möglichkeit, die **Geschwindigkeit zu testen**, mit der ein Primzahlprogramm in Maschinensprache im Vergleich zu einem Programm in einer höheren Sprache ausgeführt wird.

Die Eingabe von Maschinenprogrammen kann über den Programmpunkt „**HexEdit**“ durch **Eingabe von Zahlenkodes** für die Befehle erfolgen oder über „**AssEdit**“ durch Eingabe von besser lesbaren **Textkodes** für die einzelnen Befehle wie z.B. add ax,bx für die Addition. Gibt man ein Maschinenprogramm unter „AssEdit“ ein, so kann man unter „HexEdit“ die zugehörigen Zahlenkodes finden. Die Eingabefelder kann man jeweils durch Betätigen der ESC-Taste verlassen.

**HexEdit** gibt links Adressen durch zwei vierstellige Hexadezimalzahlen mit Ziffern von 0 bis F vor, denen in der gleichen Zeile 16 zweistellige Zahlen folgen. Diese Zahlen können durch die HexKode von Maschinenbefehle überschrieben werden. Die einzelnen Positionen werden von 0 bis 15 durchgezählt. Die Adresse der HexKodes ergibt sich aus der vorgegebenen Adresse links, dazu addiert die Positionsnummer:

Beispiel: 0000 0012 : 07 08 0A 0B 0C 0D 0E 0F 17 18 1A 1B 1C 1D 1E 1F

0C steht an der 5. Stelle, also Positionsnummer 4 und hat daher die Adresse 0000 0016. In Anlehnung an das CPU-Modell dürfen als Adressen nur die Zahlen 0000 0000 bis 0000 013F verwendet werden. **Einzelheiten** zur Kodierung der Befehle findet man in der Datei **ULog-Mikrokode**. Aufbau und Funktion von HexEdit entsprechen denen eines früheren Midrosoft-Programms, mit dem Maschinenprogramme direkt eingegeben und getestet werden konnten. Ist das Programm vollständig eingegeben, so kann es nach Betätigen von ESC mit dem Befehl „**Start**“ getestet werden. Vorzugeben ist dabei die Dauer, wie lang ein Befehl angezeigt wird.

Ein mit **AssEdit** eingegebenes Programm muss vor dem Start mit „**Assemblieren**“ in den zugehörigen Zahlenkode übersetzt werden. Erst danach kann es gestartet oder mit HexEdit betrachtet werden. Der Maschinenkode enthält die wichtigsten Befehle der Prozessoren der Intel-Reihe, die Adressierung ist ebenfalls von Intel übernommen, allerdings sinnvoll erweitert.

Der simulierte Prozessor hat die Register AX (Akkumulator, Ergebnisregister), BX, CX, DX. Zum Speichern und Lesen von Daten und für Sprünge sind Adresseingaben nötig. Hierzu kann man direkt die Registerbezeichnungen verwenden oder in eckigen Klammern die konkrete Adresse im externen RAM-Speicher. Trägt man in eckigen Klammern eine Registerbezeichnung ein, so holt sich die CPU aus dem Register die Adresse, mit der das RAM angesteuert wird. Als Erweiterung kann man im Programm den Befehl **var Name** einfügen, mit dem der **Speicherplatz**, an dem dieser Befehl steht, einen Namen erhält. Dieser Name kann dann statt einer Zahl in eckige Klammern geschrieben werden, wenn man den Speicherplatz benutzen will. Durch **:Name** wird eine Speicherposition als **Sprungziel** gekennzeichnet. Man trägt dann z.B. nach jmp diesen Namen ein. Da es infolge dieser Spracherweiterungen nicht klar ist, was zum eigentlichen Programm gehört und was dann überschreibbarer Speicherplatz ist, muss die Befehlsfolge mit **end** beendet werden. **Nach end** können dann mit **var** Speicherplätze reserviert werden.

Eine **Kurzübersicht** über die verwendbaren **Maschinenbefehle** findet man auch in der Datei **ULog-Assembler**.

Beispiel: mov [ra],10  
mov ax,2  
mov bx,[ra]  
add ax,bx  
mov [rb],ax  
end  
var ra  
var rb

Hinweise zu den **Maschinenbefehlen**:

|                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| mov <i>Ziel, Quelle</i>                                                                                          | Daten von der Quelle werden zum Ziel kopiert. Ziel und Quelle können Register, RAM-Adressen und Variable sein. Alternativ kann als Quelle auch eine Zahl eingegeben werden. Ziel oder Quelle müssen ein Register sein.                                                                                                                                                                                                                               |
| push, pop                                                                                                        | Speichern, Lesen von Daten auf dem Stackspeicher. Es ist ein Register anzugeben.                                                                                                                                                                                                                                                                                                                                                                     |
| and, or <i>Ziel, Quelle</i><br>adc, sbc <i>Ziel, Quelle</i><br>shl, not <i>Quelle</i><br>cmp <i>Ziel, Quelle</i> | logisches AND, OR<br>Addition, Subtraktion von Zahl aus der Quelle zu Ziel<br>Verschieben der Zahl aus Quelle um 1 Stelle nach links, logisches NOT<br>Vergleich Zahlen aus Quelle und Ziel, Ergebnis in Z(ero)-Flag<br>Ist die Quelle selbst ein Zahl, so kann Ziel ein beliebiges Register sein,<br>ist die Quelle ein Register, so muss Ziel das AX-Register sein.<br><b>RAM-Adressen und Variablen sind als Ziel oder Quelle nicht zulässig!</b> |
| jmp <i>Ziel</i><br>jnz <i>Ziel</i>                                                                               | Sprung zum Ziel als Adresse<br>Sprung nur dann, wenn das Z-Flag 0 ist<br>Eine Zahl als Ziel ist die Zieladresse! Ist Ziel ein Register oder eine RAM-Adresse, so ist die                                                                                                                                                                                                                                                                             |

|     |                                                                                                                                                              |
|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
|     | Zahl aus dem Register oder dem RAM-Speicher die Zieladresse<br>Als Ziel kann ein Name genommen werden, der mit <b>:Name</b> eine Stelle im Programm markiert |
| nop | Befehl ohne Funktion (no operation)                                                                                                                          |

Wenn man Spaß daran hat, kann man versuchen, Mikroprogramme für die folgenden **Aufgaben** zu schreiben:

1. Man multipliziere eine Zahl im BX-Register mit 4 und speichere das Ergebnis in AX. Zwei Möglichkeiten hierzu: wiederholtes Addieren oder Verschieben der Bits nach links.
  2. Negative Zahlen werden in der Praxis auf 2 Weisen dargestellt: als Komplement, also FF - Zahl (8-stellig) oder als Zahl mit einer führenden 1 als Vorzeichen. Beispiel: Ist die Zahl -0010 1011, so ist das Komplement 1101 0100, die zweite Möglichkeit wäre 1010 1011. Positive Zahlen hätten bei der 2. Darstellung eine führende 0.
- Die Aufgabe ist nun, durch ein Programm eine Zahl in der zweiten Darstellung in das Komplement umzuwandeln.

## 6. Automat

Behandelt wird hier ein theoretischer Ansatz zur Datenverarbeitung, bei der mittels einfacher Regeln komplexe Aufgaben gelöst werden. Eine Realisierung eines Automaten ist das **Steuerwerk des CPU-Modells**. Grundsätzlich lassen sich alle Automaten wie bei dem Steuerwerk mittels ROMs realisieren. Versieht man den Automaten zusätzlich mit einem unbegrenzten Speicher, so wird aus dem Automaten eine „**Maschine**“. Es gibt einen mathematischen Nachweis, dass eine einfach strukturierte Maschine konstruierbar ist, die alle Aufgaben der Datenverarbeitung bearbeiten kann. Die Steuerung dieser Maschine geschieht über ein Programm, das in dem zusätzlichen Speicher abgelegt ist. Diese Maschine zeigt aber auch die Grenzen für die technische Datenverarbeitung, die nicht überschritten werden können.

Die Regeln, die den Automaten steuern, bestehen aus einem **Eingangswort**, dem in der konkreten Regel ein **Ausgangswort** zugeordnet ist. Beide Wörter bestehen aus einer Zeichenkette, die verarbeitet wird, und einem **Zustandssymbol**, das in eckigen Klammern steht. Es gibt einen Startzustand S und einen Endzustand E, der die Bearbeitung von Regeln beendet. Liegt eine Zeichenkette vor, so überprüft der Automat, ob ein Teil das Eingangswort einer Regel ist. Ist das der Fall, wird dieser Teil durch das Ausgangswort ersetzt.

**In ULog darf die Zeichenkette des Eingangswortes höchstens aus einem Zeichen bestehen, das Ausgangswort nur aus einem Zustand!**

Damit kann der Automat in ULog als **erkennender Automat** zur Identifizierung von Zeichenketten genutzt werden: Dies geschieht auch beim **Kompilieren** von Programmen, bei denen **Schlüsselwörter wie if oder print erkannt** werden müssen. Zudem wird so überprüft, ob die Syntax korrekt ist. Wird ein Schlüsselwort erkannt, so kann es durch den Aufruf eines passenden Maschinenprogramms ersetzt werden.

Beispiel: Regeln: 1[S] -> [A]  
0[A] -> [B]  
1[B] -> [E]

101 wird vom Automaten akzeptiert, 1101 nicht

## 7. Platine

Wird für eine Schaltung eine Platine benötigt, so sollen dort die Leiterbahnen so verlaufen, dass sich möglichst wenige überkreuzen, da dann zusätzliche Drahtbrücken nötig sind. ULog zeigt ein Verfahren, mit dem die Leiterbahnen automatisch angeordnet werden, nachdem man die Verbindungspunkte zu den Bauteilen eingegeben hat. Das Verfahren bestimmt den kürzesten Weg von einem Ausgangspunkt zu einem Zielpunkt, wobei vorhandene Leiterbahnen umgangen werden. Das Verfahren ähnelt dem Verfahren von Navigationsgeräten.

ULog gibt ein Raster vor, bei dem zunächst der Ausgangspunkt einer Verbindung anzuklicken ist, zu Bestätigen mit „**Anfang >A**“. Klickt man dann den Endpunkt an und betätigt man „**Verbinden >V**“, so entwickelt das Programm stufenweise die Verbindung zwischen beiden Punkten. Klickt man nach Wahl des Endpunktes auf „**Löschen >C**“, so wird eine Verbindung zwischen Anfangs- und Endpunkt gelöscht. Einzelne Punkte kann man nach Anklicken mit „**Löschen >X**“ löschen.

Der Menüpunkt „**Sonst >S**“ bietet Möglichkeiten zum Speichern und Laden von Platinenplänen auf externen Speichern und den Ausdruck der Platine.