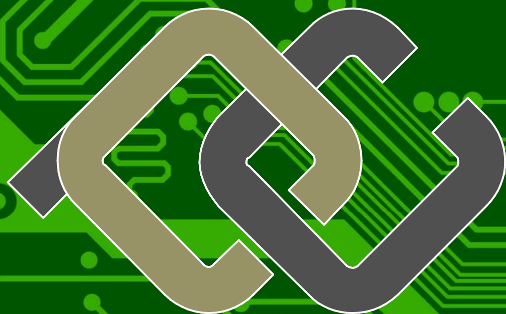
Nr. 175/Dezember 2022 € 5,00
ISSN 1022-1611



PCNEWS

ClubComputer · Digital Society

ClubDev

Backtracking mit Python

Digitaltechnik „begreifbar machen"

ClubDigitalHome

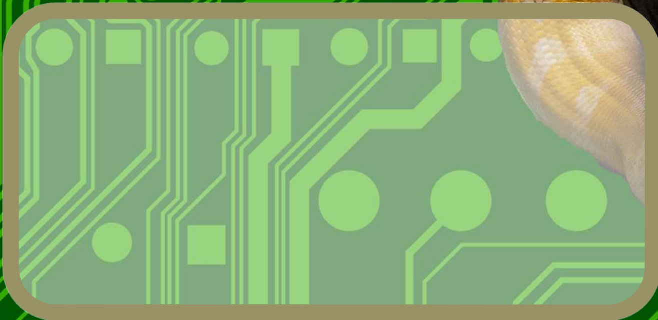
Amazon Killer

ClubTermine

30.11., 2.12., 26.1.

Seite 2,3

Österreichische Post Mz 16Z040679 M ClubComputer, Siccardsburggasse 4/1/22 1100 Wien



Inhalt

LIESMICH

1 **Cover**
Franz Fiala

Digitaltechnik und die Programmiersprache **Python** sind das Thema der aktuellen Ausgabe. Es wurde versucht, beide Aspekte auf der Titelseite zu vereinigen.

Hintergrund: Pixabay, [OpenClipart-Vectors](#) (user_id:30363), Leiterplatte, [board-158973_1280.png](#)
Motiv: Frau mit Python: Pixabay, [Victoria_rt](#) (user_id:6314823), [pose-5298179.jpg](#)

2 **Liebe Leser, Inhalt**
Franz Fiala

2 **Impressum, Autoren, Inserenten, Services**

DIGITALSOCIETY

3 **Zukunft der Demokratie**
Werner Illsinger

CLUBDEV

5 **Was ist eigentlich...Backtracking?**
Martin Weissenböck

8 **Königinnenproblem**
Martin Weissenböck

11 **Sudoku**
Martin Weissenböck

13 **Kendoku**
Martin Weissenböck

16 **Skyline**
Martin Weissenböck

18 **Zeltlager**
Martin Weissenböck

18 **Python**
Martin Weissenböck

18 **Digitaltechnik "begreifbar machen"**
Helmut Bittermann

CLUBDIGITALHOME

35 **Amazon Killer**
Günter Hartl

Liebe Leser!

Franz Fiala

PCNEWS-175

Dieses Heft hat zwei große Themen:

- *Martin Weissenböck*
  Backtracking mit Python
- *Helmut Bittermann*
  Digitaltechnik „begreifbar machen“

Vielen Dank für diese tollen Inhalte!

Wir wünschen unseren Lesern ein schönes Weihnachtsfest und ein gesundes Neues Jahr 2023.

Franz Fiala

Autoren

Bittermann Helmut — 20-34

Lehrer für Informatik und Projektmanagement im WASA-Gymnasium

bit@gb9.at

Fiala Franz Dipl.-Ing. 1948 — 1,2

Vizepräsident von ClubComputer, Leitung der Redaktion und des Verlags der PCNEWS, Lehrer für Nachrichtentechnik und Elektronik .i.R.

Werdegang Arsenal-Research, TGM Elektronik

Absolvent TU-Wien, Nachrichtentechnik

franz.fiala@clubcomputer.at
http://fiala.cc/

Hartl Günter Ing. 1963 — 35

Wirtschaftsingenieur, Systemadministrator für Windows Clients und Linux Server in Logistikcenter

Hobbies Krav Maga, Windsurfen, Lesen

ghartl3@gmail.com

Illsinger Werner Ing. 1968 — 3

Präsident der Digital Society und von ClubComputer, Inhaber Vividity Strategieberatung, Managing Partner Digital Society Institute. Lektor an der FH Kärnten für Digital Transformationmanagement.
Absolvent: TGM-Nachrichtentechnik
werneri@digisociety.ngo
https://vividity.eu

Weissenböck Martin Dir.Dr. 1950 — 11-19

Leiter der ADIM und Autor von ADIM-Skripten, Leiter des Vereins "SCHUL.InfoSMS, Univ.-Lektor an der TU Wien, Direktor der HTL Wien 3 Rennweg i.R.
martin@weissenboeck.at
http://www.weissenboeck.at/

Inserenten

techbold — 36

Dresdner Straße 89 1200 Wien
+43 1 34 34 333
office@techbold.at
http://www.techbold.at

Produkte Reparatur, Aufrüstung, Softwareinstallation, Datenrettung. Installation und Wartung von IT-Anlagen.

Impressum

Impressum, Offenlegung

Richtung Auf Anwendungen im Unterricht bezogene Informationen über Personal Computer Systeme. Berichte über Veranstaltungen des Herausgebers.

Erscheint 4 mal pro Jahr: Mär, Jun, Sep, Nov
ISSN 1022-1611

Herausgeber und Verleger **ClubComputer**
Siccardsburggasse 4/1/22 1100 Wien
01-6009933-11 FAX: -12
buero@clubcomputer.at
https://clubcomputer.at/
ZVR: 085514499
IBAN: AT74 1400 0177 1081 2896
Mitgliedsbeitrag 2019: 46,-Euro
Konto: AT74 1400 0177 1081 2896
oder
PayPal office@clubcomputer.at

Digital Society
Graben 17/10 1010 Wien
01-314 22 33
info@DigiSociety.at
https://digisociety.at/
ZVR: 547238411
IBAN: AT45 3266 7000 0001 9315

Druck **Ultra Print**
Pluhová 49, SK-82103 Bratislava
http://www.ultraprint.eu/

Versand 16Z040679 M

PDF-Version http://d.pcnews.at/_pdf/n175.pdf

Namensnennung, nicht kommerziell, keine Bearbeitungen
http://creativecommons.org/licenses/by-nc-nd/4.0/

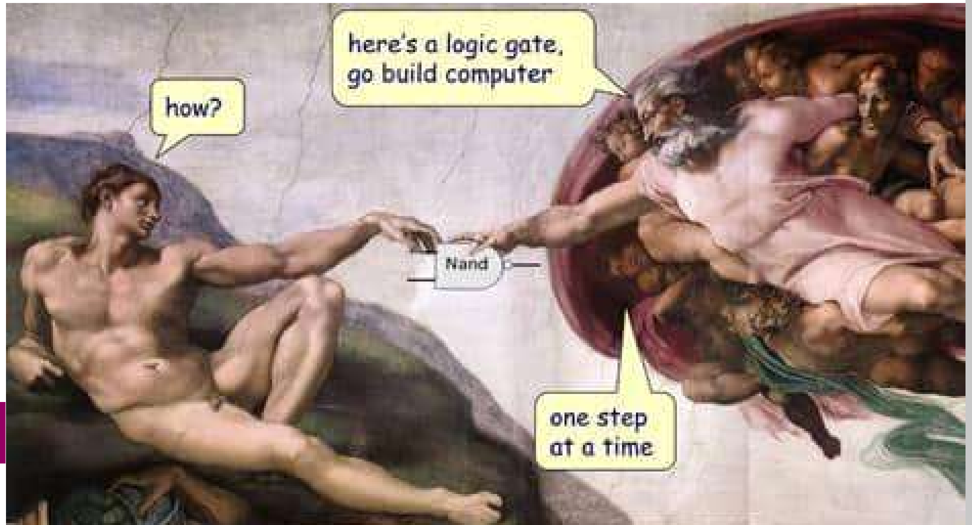


Abb.31: Aus dem TED-Talk von Shimon Schocken, einem der Begründer des „From Nand To Tetris-Projekts. (Quelle: https://www.nand2tetris.org/)

<Ende des Beitrags über Digitaltechnik von Seite 34>

Termine

Datum	Veranstaltung
Mi 30.11.2022 *siehe Seite 3*	**Information & Wissen** (Reihe „Zukunft der Demokratie“, Digital Society) **Eva Linsinger**, Profil; **Barbara Wimmer**, Futurezone; **Klaus Unterberger**, Leiter Kompetenzzentrum Public Value, ORF. Digital Society, Graben 17/10 oder online
Do 01.12.2022 *siehe Homepage*	**MFA (Multi Faktor Authentifizierung) (ClubComputer)** Referent **Roman Korecky**. Zoom-Meeting organisiert von **Georgie**,
Mi 25.01.2023 *siehe Seite 3*	**Bürger*innenbeteiligung** (Reihe „Zukunft der Demokratie“, Digital Society) **Andreas Kovar**, Geschäftsführer, Kovar & Partners; **Edward Strasser**, CEO, Innovation in Politics; **Kisten Neubauer**, Geschäftsführerin, Digital Hotpot Digital Society, Graben 17/10 oder online

Zukunft der Demokratie

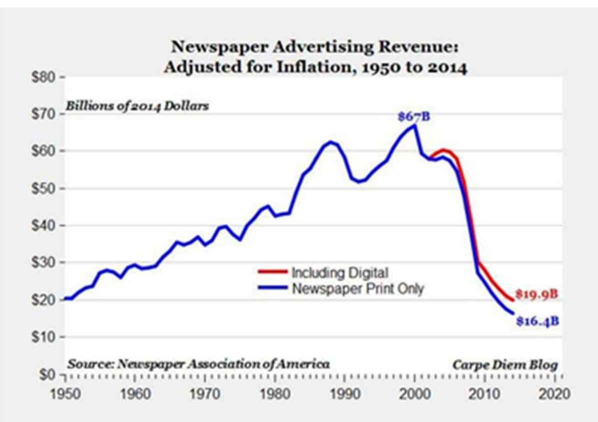
Werner Illsinger

> *„Die Demokratie ist die schlechteste aller Staatsformen, ausgenommen alle anderen."*
> (Winston Churchill)

Täglich erfahren wir neue Details von politischen Verfehlungen. Medien werden über Inserate angefüttert, Geld und Posten an politische Freunde verteilt und Korruption ist ein allgegenwärtiges Thema. Menschen wissen nicht, was man dagegen tun kann. Dieses Ohnmachtsgefühl führt zu Demokratieunzufriedenheit und Politik-Müdigkeit. Die sinkende Wahlbeteiligung spricht eine klare Sprache (Bei der letzten Bundespräsidentenwahl lag z.B. die Wahlbeteiligung nur noch bei 65%.)

Digitalisierung und Medienfinanzierung

Die Digitalisierung hat manche Probleme verursacht bzw. verstärkt. Die Geschäftsmodelle der Medien funktionieren nicht mehr. Medien haben sich früher aus vielfältigen Einnahmequellen finanziert. Der Verkauf von Zeitungen ist seit Jahren rückläufig. Inserate werden von Unternehmen daher eher online geschaltet als in klassischen Medien. Kontaktanzeigen wurden durch Tinder und andere Online Portale verdrängt, Kleinanzeigen durch Willhaben. Klassische Medien haben also viele Mitbewerber im Onlinebereich erhalten, daher sind die Einnahmen rückläufig.



Die nebenstehende Statistik zeigt die US-Zahlen. Der Zeitungs-Anzeigenmarkt ist 2020 wieder auf dem Niveau von 1950 gewesen (inklusive online) und von fast 70 Milliarden US Dollar im Jahr 1998 auf rund 20 Milliarden 2020 gefallen.

Klassische Medien finanzieren mit ihren Einnahmen Journalisten. Online Medien (wie Meta/Facebook oder Google) stellen keine Journalisten an. Sie haben aber den Großteil der Werbeeinnahmen von den klassischen Medien übernommen.

Das ist der Hauptgrund, warum das „Anfüttern" der Medien durch Inserate der öffentlichen Stellen so leicht geworden ist. Die Medien brauchen Geld, um sich zu finanzieren. Dies ist problematisch. Es ist damit keine kritische Berichterstattung der Medien mehr zu erwarten. Die Hand, die einen füttert, beißt man nicht.

Digitalisierung und Reichweite

Früher hatten Medien eine Gatekeeper Funktion. Nur jemand der es geschafft hatte, dass (reichweitenstarken) Medien über ihn berichten, konnten auch Reichweite bekommen. Heute, durch die digitalen Medien kann das jede(r). In Deutschland ist beispielsweise der YouTube Blog Härtetest mit 20 Mio. Abonnenten und über 3 Milliarden Video Aufrufen das reichweitenstärkste Blog. Im Vergleich dazu: Die ZIB1 hat eine Reichweite von 1,1 Mio. Die ARD-Tagesschau hat rund 11 Mio. Seher. Diese enormen Reichweiten von online Medien geben auch in Verbindung mit sozialen Medien die Möglichkeit „Fake News" – früher hätte man vermutlich Propaganda dazu gesagt, ungefiltert an eine große Zahl von Menschen zu verbreiten. Das Schwierige dabei ist, dass sich „Aufreger" Videos dabei besonders stark verbreiten und aufgrund von mangelnder Medienkompetenz, viele Menschen einfach alles glauben, was sie sehen oder hören und zu ihrem Weltbild passt.

Lösung der anstehenden Probleme

Der Oberösterreichische Landtag initiierte im Jahr 2020 das Demokratieforum Linz. Dort wurden Antworten auf die Frage „Wie können wir die Demokratie in Österreich reparieren?" gesucht.

Das Resultat dieser Arbeit lässt sich mit vier Themenschwerpunkten zusammenfassen, welche wir für unser Programm übernommen haben:

- **Information & Wissen (Schule, Medien)**
  Damit Bürger*innen sich informiert in den politischen Prozess einbringen können, benötigen sie sowohl Wissen über das politische System als auch verständlich aufbereitetes Wissen über die anstehenden Herausforderungen und Lösungsansätze.
- **Bürger*innenbeteiligung (Partizipation)**
  Wenn sich Bürger*innen in den Entscheidungsfindungsprozess einbringen können, erhöht das die Zufriedenheit mit gefundenen Lösungen.
- **Lösungsorientierte Politik**
  Politik soll lösungsorientiert sein, daher müssen rasch Lösungen auf die anstehenden Herausforderungen gefunden werden, die alle Beteiligten (Stakeholder) berücksichtigen.
- **Saubere Politik (Compliance)**
  Die Politik ist gefordert sich an die Gesetze, die sie selbst beschließt, auch zu halten. Tut sie das nicht, verspielt sie das Vertrauen der Menschen sehr schnell.

Lösung

Wir – als Digital Society – sind der Meinung, dass viele der großen anstehenden Probleme in diesen Bereichen mittels *digitaler Tools* leichter gelöst werden können. Wir planen über die nächsten sechs Monate eine Veranstaltungsserie (DigiTalks), zu der wir nicht nur zur Teilnahme, sondern auch zur aktiven Mitgestaltung einladen. Beispielsweise ist Wissensvermittlung an die breite Bevölkerung mittels digitaler Tools wesentlich leichter geworden. Auch die Partizipation ist mittels digitaler Tools wesentlich leichter, als für jede anstehende Entscheidung Menschen an die Wahlurnen oder ins Gemeindeamt zu Stimmenabgabe zu rufen. Digitale Tools bieten auch die Möglichkeit, Diskurs mit anderen Menschen über weite Entfernungen zu führen. So können sich Vorarlberger*innen mit Burgenländer*innen über anstehende Probleme unterhalten. Da viele Prozesse in unserem Staat bereits digital sind, bietet die zur Verfügungstellung der Informationen auch deutlich mehr Möglichkeiten für Transparenz und damit weniger Möglichkeiten für Korruption, wenn diese Informationen nur genutzt würden.

Wir veranstalten daher gemeinsam mit unseren Kooperationspartnern zu jedem der oben genannten Themen eine Abendveranstaltung (DigiTalk), um über Herausforderungen und bekannte Lösungsansätze mit Expert*innen zu diskutieren. Die DigiTalks sollen dazu dienen, die Herausforderungen konkret zu formulieren. Die Ergebnisse bilden die Arbeitsbasis für ein *Barcamp*, um dort konkrete Lösungsvorschläge zu erarbeiten.

DigiTalk-Termine

Der derzeitige Planungsstand mit den bisherigen prominenten Zusagen zu der Veranstaltungsserie, siehe Tabelle rechts.

Als Abschluss und großes Finale der Reihe planen wir ein ganztägiges *Barcamp*, in dessen Verlauf wir gemeinsam mit allen Interessierten, aber auch mit der Politik, Forderungen und Lösungsvorschläge erarbeiten möchten.

Die Ergebnisse werden dann an die Politik herangetragen und weiterverfolgt.

Nächste Schritte

Wir ersuchen die Mitglieder von Digital Society sowie ClubComputer und die Leser der PCNEWS. sich die Termine vorzumerken. Die Anmeldung zu den Veranstaltungen wird demnächst möglich sein. Für ClubComputer Mitglieder ist die Teilnahme an den Veranstaltungen kostenlos.

Wenn Sie sich für ein Thema interessieren und Sie sich auch inhaltlich oder organisatorisch in die Arbeit einbringen möchten, kontaktieren Sie die Digital Society unter info@digisociety.ngo

Kooperationspartner

Themenpartner

Medienpartner

Mi 30.11.2022

Information & Wissen

- **Eva Linsinger**, Profil
- **Barbara Wimmer**, Futurezone
- **Klaus Unterberger**,
  Leiter Kompetenzzentrum Public Value, ORF

Mi 25.01.2023

Bürger*innenbeteiligung (Partizipation)

- **Andreas Kovar**, Geschäftsführer, Kovar & Partners
- **Edward Strasser**, CEO, Innovation in Politics
- **Kisten Neubauer**, Geschäftsführerin, Digital Hotpot

Mi 22.02.2023

Lösungsorientierte Politik

Mi 29.03.2022

Saubere Politik (Compliance)

- **Irmgard Griss**
  ehem. Präsidentin des Obersten Gerichtshofes
- **Martin Kreutner**, **Dean Emeritus**
  International Anti Corruption Academy
- **Günther Ogris**, Managing Partner, SORA Institut

Sa 22.04.2023

Barcamp

Zukunft der Demokratie

Demokratie

Was ist eigentlich ... Backtracking?

Martin Weissenböck

Backtracking oder Rückwärtsverfolgung beschreibt einen Algorithmus, bei dem die Lösung einer Aufgabe durch systematisches Probieren gesucht wird. Dabei werden die Lösungsschritte für jede Teillösung nach folgendem Schema ermittelt:

1. Zuerst wird festgestellt, welche Werte in für die konkrete Teillösung möglich sind.
2. Dann wird überprüft, ob der erste Wert zu einer gültigen Lösung führen kann.
3. Wenn ja, wird Vorgang für die nächste Teillösung (Schritt (1)) fortgesetzt.
4. Wenn nein, wird der nächste Wert für die Teillösung untersucht.
5. Wenn keine Werte mehr übrig sind, ist bei der vorhergehenden Teillösung der nächste Wert zu nehmen.

Der Vorgang wird gern anhand des Königinnenproblems (auch Damenproblem genannt) auf einem Schachbrett erklärt. Eine Königin kann eine andere Figur in desselben Zeile, derselben Spalte und derselben Diagonale schlagen. Die Aufgabe lautet, acht Königinnen so auf dem Schachbrett aufzustellen, dass keine eine andere schlagen kann.

In der Wikipedia ist der Vorgang im Detail beschrieben:
https://de.wikipedia.org/wiki/Damenproblem

4 Königinnen

Probieren wir das Ganze mit vier Königinnen auf einem 4×4-Schachbrett: das ist leichter überschaubar. Da wir später ein Programm dazu schreiben werden, verwenden wir die in der Informatik übliche Zählweise beginnend mit 0.

Wenn wir die Felder mit der Zeilennummer (**`row, r`**) und der Spaltennummer (**`Column, c`**) bezeichnen, sind das die Feldnummern (**`r`**, **`c`**):

- **Bild 1**: Wir beginnen links oben, Platz **`(0,0)`**
- **Bild 2**: In der Zeile 0 kann keine weitere Königin stehen. Daher weiter in der Zeile 1. Der erste mögliche Platz ist in Spalte 2
- **Bild 3**: Aber schon gibt es keinen Platz mehr in der Zeile 2. Daher muss die Königin in Zeile 1 in Spalte 3 rücken
- **Bild 4**: Der Platz in Zeile 2 und Spalte 1 passt
- **Bild 5**: Aber nun ist kein Platz in Zeile 3. Die Königin in Zeile 2 hat in dieser Zeile keinen Platz mehr. Auch die Königin in Zeile 1 kann nicht mehr weiterbewegt werden. Daher zurück in Zeile 0 – neue Position in der Spalte 1
- **Bild 6**: Weiter in Zeile 1
- **Bild 7**: Weiter in Zeile 2
- **Bild 8**: Und mit Zeile 3 ist die Lösung fertig

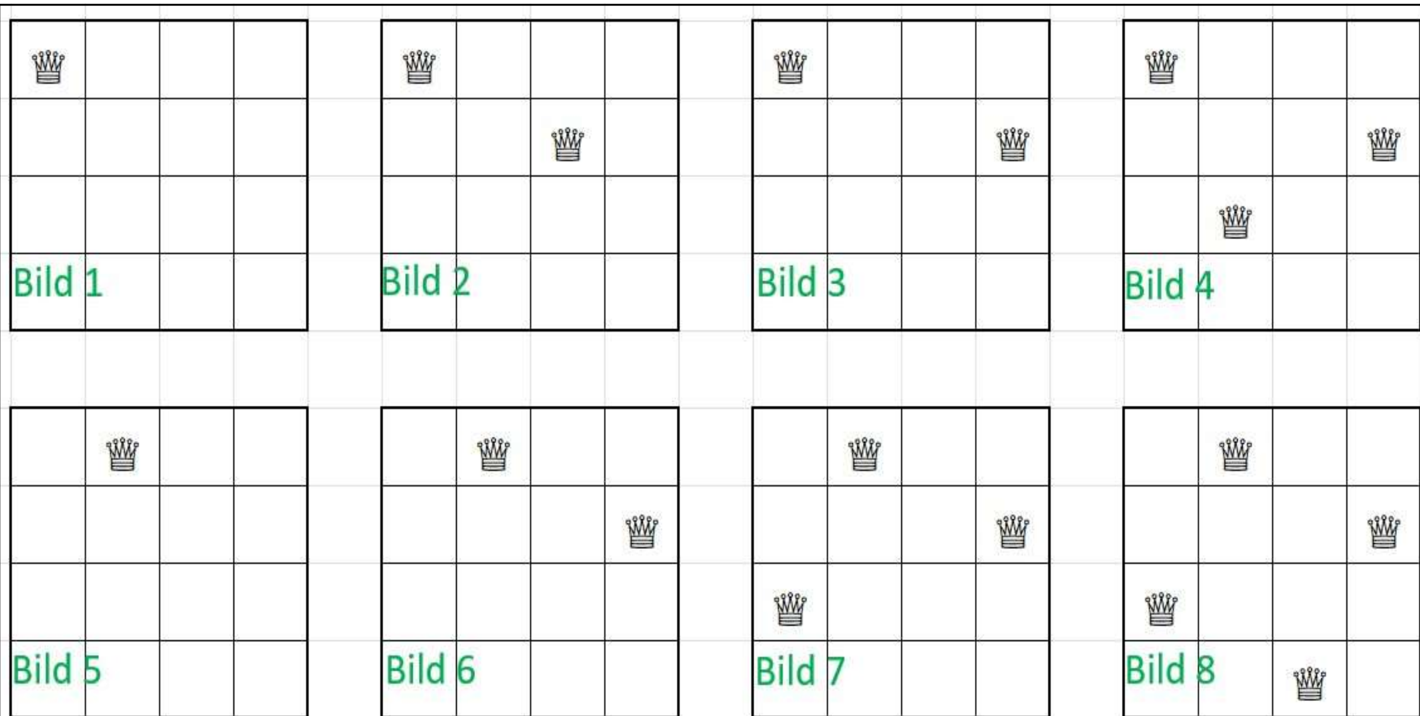
Es gibt noch eine zweite Lösung, die symmetrisch zur ersten ist – **Bild 9**:

Code zum Thema „Backtracking“

Bei jedem der folgenden Artikel wird auch das Programms abgedruckt. Der Code zu den Programmen und auch zu allen Varianten dieser Programme findet man bei der Online-Version der Artikel unter
https://clubcomputer.at/wp-content/uploads/sites/6/2022/08/backtrack.zip

Hinweise zur Handhabung von Python auf Seite 18.

```
backtrack.py
backtrack0.py
kendoku.py
kendoku_2.py
queens.py
queens_2.py
skyline.py
skyline_2.py
skyline_3.py
skyline_4.py
sudoku.py
sudoku_2.py
zeltlager.py
zeltlager_2.py
zeltlager_3.py
```



CLUBDEV.NET

Unter
https://www.mathematik.ch/spiele/N_Damenproblem/
wird der Ablauf animiert dargestellt. Größe und die Verzögerung zwischen den einzelnen Schritten können eingestellt werden.

Lösung per Programm

Alle Programme dieses Beitrags können direkt vom Webserver geladen, in ein Verzeichnis kopiert und auch gleich mit Python 3 gestartet werden. Auf weitere Programmdateien wird im Text verwiesen. Sie sind hier nicht abgedruckt sind, aber die Dateien sind dort ebenfalls zu finden. Die Namen der Programmdateien sind hier hervorgehoben.

Der Algorithmus wiederholt im Schritt (c) den Vorgang ab Schritt (a). Programmtechnisch wird das zu einer Rekursion – das ist eine Funktion, die sich selbst aufruft.

Im Internet sind viele Programmbeispiele in unterschiedlichen Sprachen zu finden. Aber wir wollen einen Schritt weiter gehen. Die Idee „Backtracking" lässt sich auf viele Probleme anwenden, auch auf typische Rätsel, wie sie beispielsweise in der Presse an jedem Sonntag veröffentlicht werden:

https://www.diepresse.com/5789519/die-erweiterte-raetselseite-der-presse-am-sonntag-zum-ausdrucken

Die objektorientierte Programmierung erlaubt die Trennung zwischen dem grundlegenden Algorithmus „Backtracking" und der konkreten Aufgabenstellung, also einem konkreten Rätsel als abgeleitete Klasse. In einem Artikel in der Zeitschrift der ACM wurde dieser Lösungsweg (erstmalig?) vorgestellt: *„An Object-Oriented View of Backtracking"* (Robert E. Noonan et.al., https://dl.acm.org/doi/pdf/10.1145/331795.331886). Der folgende Lösungsvorschlag orientiert sich daran und ist in Python 3 geschrieben.

Backtrack objektorientiert

Zuerst einmal die Klasse **`Backtrack0`**, Programmdatei: **`backtrack0.py`**, in der der grundlegende Backtrack-Algorithmus implementiert wird:

Backtrack0

Erklärungen zum Programm

Eine Reihe von Methoden sind Platzhalter, die erst bei einer konkreten Aufgabe im Detail in die abgeleitete Klasse eingefügt werden. Ein Aufruf hier führt zu einer Fehlermeldung.

- **`moves`** liefert für die Methode **`attempt`** jene Werte, die der Reihe nach auszuprobieren sind. Gegebenenfalls werden Werte, die zu keiner Lösung führen können, gar nicht in die Liste aufgenommen.

```python
# File backtrack0.py
# Inspiriert von
# https://dl.acm.org/doi/pdf/10.1145/331795.331886

class Backtrack0(object):

    def __init__(self):
        self.attempt(0)

    def moves(self, level:int):
        raise NotImplementedError()

    def valid(self, level:int, move):
        raise NotImplementedError()

    def record(self, level:int, move):
        raise NotImplementedError()

    def undo(self, level:int, move):
        raise NotImplementedError()

    def done(self, level:int):
        raise NotImplementedError()

    def attempt(self, level: int) -> bool:
        successful: bool = False
        for move in self.moves(level):
            if self.valid(level, move):
                self.record(level, move)
                if self.done(level):
                    successful = True
                    self.undo(level, move)
                else:
                    successful = self.attempt(level+1)
                    if not successful:
                        self.undo(level, move)
```

- **valid** überprüft, ob der Wert **move** im Schritt **level** ein gültiger Wert ist.
- **record** speichert den von **valid** überprüften Wert ab.
- **undo** entfernt den Wert wieder, wenn mit ihm keine Lösung möglich ist.
- **done** prüft, ob alle Schritte schon erledigt sind und das Rätsel gelöst ist. In diesem Fall wird die Lösung auch gleich ausgegeben und nach einer weiteren Lösung gesucht.
- **attempt** ist die zentrale Methode, die die vorher genannten Methoden verwendet. Sie wird mit dem Startwert **0** beim Initialisieren durch **__init__** aufgerufen.

Warum englische Bezeichnungen für die Methoden? Einerseits um den Vergleich mit dem ACM-Artikel zu ermöglichen, andererseits um eine Veröffentlichung dieses Beitrags in anderen Medien zu erleichtern.

Viele Versuche waren für die Lösung vorwendig? Wie lange hat die Suche gedauert?

Dazu erweitern wir **Backtrack0** zu **Backtrack**. Auf **Backtrack** bauen dann auch alle folgenden Beispiele auf.

Backtrack

Weitere Hinweise

Bei langen Rechnungen sollte angezeigt werden, ob das Programm noch „lebt". Daher wird nach je 100.000 Versuchen ein Punkt ausgegeben und nach jeweils 5.000.000 Versuchen ein Text.

Mit **print(..., end="")** werden die einzelnen Punkte nebeneinander und nicht untereinandergeschrieben.

Zahlen können mit einem „_"-Zeichen leichter lesbar gemacht werden. Die Erfinder der IBAN hatten leider keine vergleichbare Idee.

Das Programm verwendet Formatstrings: **formatted string literals**, siehe https://docs.python.org/3/tutorial/inputoutput.html und https://docs.python.org/3/library/string.html#formatspec. Wer sie noch nicht kennt: bitte unbedingt eine der vielen Dokumentationen dazu im Internet lesen!

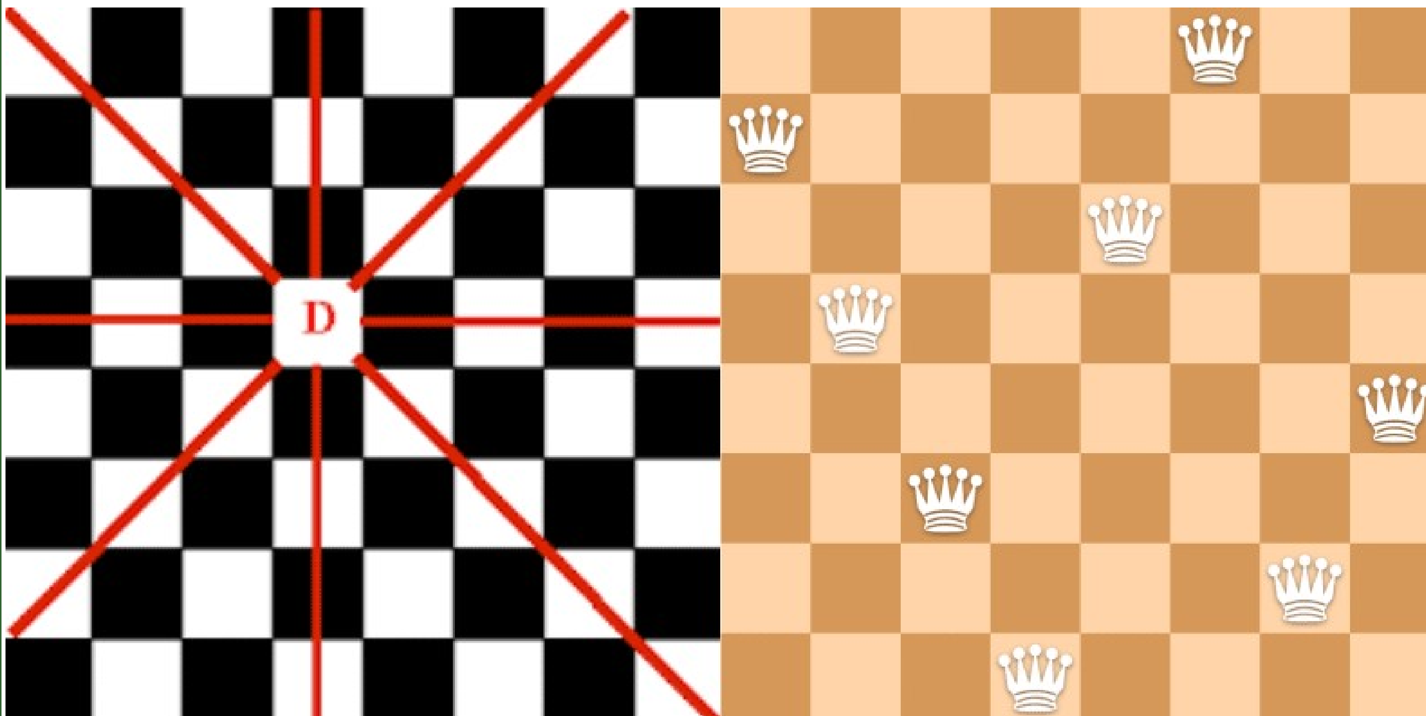
In **__init__** wird der Formatstring in der nächsten Zeile fortgesetzt. Der Fortsetzungsstring muss auch mit **f** beginnen.

```
# File backtrack.py

import time, sys
from backtrack0 import Backtrack0

class Backtrack(Backtrack0):

    attempts = 0
    showAttempts = False

    def __init__(self, test:bool = False, showAttempts:bool = False):
        self.test = test
        start = time.time()
        self.attempt(0)
        end = time.time()
        self.showAttempts = showAttempts
        if showAttempts:
            print(f"\n{self.attempts:16_d} attempts"
              f" in {end-start:8,.2f} seconds")

    def attempt(self, level: int) -> bool:
        if self.attempts>100_000:
            if self.attempts % 5_000_000 == 0:
                print(f"\n{self.attempts//1_000_000:4_d} Millionen ", end="\n")

            if self.attempts % 100_000 == 0:
                print(".", sep="", end="")

        successful: bool = False
        for move in self.moves(level):
            self.attempts += 1
            if self.test:
                print(f"attempt-test: level:{level}  move:{move}")
            if self.valid(level, move):
                self.record(level, move)
                if self.done(level):
                    successful = True
                    self.undo(level, move)

                else:
                    successful = self.attempt(level+1)
                    if not successful:
                        self.undo(level, move)
```


CLUBDEV.NET

Königinnenproblem

Martin Weissenböck



Das Königinnenproblem ist die erste Anwendung der Klasse **Backtrack**.

Queens.py

Listing siehe nächste Seite

Das Programm im Detail

Mit super.**__init__(...)** wird die Steuerung an die **__init__**-Methode der Basisklasse **Backtrack** übergeben.

Diese Klasse wird als Modul in der Datei **queens.py** gespeichert. Zum Aufruf kann ein neues Python-Programm mit zwei Zeilen eingesetzt werden: (**Code 1**)

```
from queens import Queens
Queens(test=False, showAttempts=True, beautify=True, size=4)
```

Code 1

Einfacher ist es aber, die folgenden zwei Zeilen am Ende von **queens.py** einzufügen: (**Code 2**)

```
if __name__ == "__main__":
    Queens(test=False, showAttempts=True, beautify=True, si-
```

Code 2

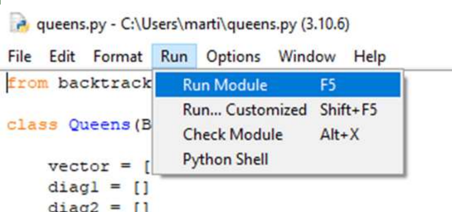
- **showAttempts** steuert die Anzeige der Anzahl der Versuche und der Dauer.
- **beautify** gleich **True** bewirkt die Ausgabe der Lösung als Quadrat.
- **size** hat den Standardwert 8 und wird hier auf 4 gesetzt.

Oder kürzer, aber weniger übersichtlich: (**Code 3**)

```
if __name__ == "__main__":
    Queens(False, True, True, 4)
```

Code 3

Die letzten zwei Zeilen erlauben es, **Queens** direkt aus der Python IDLE heraus zu starten:



Wie werden gültige Plätze für Königinnen gefunden?

Zeilen und Spalten

Vorbemerkung

- Ein Array ist in den meisten Programmiersprachen eine Zusammenfassung von gleichartigen Komponenten, zum Beispiel von ganzen Zahlen. Mehrdimensionale Arrays sind Arrays von Arrays.
- Eine Liste erweitert dieses Konzept.

Bild	vector	diag1	diag2
1	[0]	[0]	[0]
2	[0, 2]	[0, -1]	[0, 3]
3	[0, 3]	[0, -2]	[0, 4]
4	[0, 3, 1]	[0, -2, 1]	[0, 4, 3]
5	[1]	[-1]	[1]
6	[1, 3]	[-1, -2]	[1, 4]
7	[1, 3, 0]	[-1, -2, 2]	[1, 4, 2]
8	[1, 3, 0, 2]	[-1, -2, 2, 1]	[1, 4, 2, 5]
9	[2, 0, 3, 1]	[-2, 1, -1, 2]	[2, 1, 5, 4]

```python
# File queens.py
from backtrack import Backtrack
import datetime

class Queens(Backtrack):

    vector = [ ]
    diag1 =  [ ]
    diag2 =  [ ]
    solutionNr: int = 0

    def __init__(self, test:bool = False, showAttempts:bool = False,
                 beautify:bool = False, size:int = 8):
        self.beautify = beautify
        self.size = size
        super().__init__(test, showAttempts)

    def moves(self, level: int):
        return range(self.size)

    # Was ist besetzt...
    def valid(self, level: int, move:int) -> bool:
        return not(
                     # ...die Spalte?
                     move in self.vector or
                     # ...die Diagonale 1: links-oben nach rechts-unten
                     level-move in self.diag1 or
                     # ...die Diagonale 2: links-unten nach rechts-oben?
                     level+move in self.diag2
                 )

    def record(self, level: int, move):
        self.vector.append(move)
        self.diag1.append(level-move)
        self.diag2.append(level+move)

    def undo(self, level: int, move):
        self.vector.pop()
        self.diag1.pop()
        self.diag2.pop()

    def done(self, level: int):
        if level==self.size-1:
            self.solutionNr += 1
            print(f"\nLösung {self.solutionNr:4} {self.vector}")
            if self.beautify:
                self.show()
            return True
        return False

    def show(self): # "Schöne" Ausgabe
        for r in range(self.size):
            for c in range(self.size):
                print("Q  " if self.vector[r]==c else ".  ", sep="", end="")
            print()
        now = datetime.datetime.now()
        print(f"Berechnet am {now:%d. %b. %Y} um {now:%H:%M} Uhr.")

if __name__ == "__main__":
    Queens(test=False, showAttempts=True, beautify=True, size=4)
```

Listen sind in Python sehr wichtige Sprachelemente. Die Liste ähnelt dem Array, kann aber beliebige Komponenten (und natürlich auch wieder Listen) enthalten

vector ist eine Liste, in der in der Reihenfolge der Zeilen jene Spalten abgespeichert werden, die Königinnen enthalten. Zu Beginn ist die Liste leer. (siehe Tabelle rechts)

Zum besseren Verständnis findet man die **vector**-Darstellung der einzelnen Bilder in der Tabelle auf der vorigen Seite unten. Die Spalten **diag1** und **diag2** werden später erklärt.

Bild 1 (in der Tabelle) wird durch den **vector** mit dem Inhalt **[0]** beschrieben.

Diagonale 1

In den Diagonalen darf auch nur je eine Königin stehen. Zuerst einmal betrachten wir die Diagonalen von links oben nach rechts unten. Für jeden Platz des Miniatur -Schachbretts wird eine Zahl mit der Differenz Zeilennummer – Spaltennummer berechnet. Im Programm steht für die Zeilennummer **level** und für die Spaltennummer in dieser Zeile **move**. Damit wird einerseits auf den Artikel der ACM Bezug genommen und andererseits werden weitere Programmbeispiele vorbereitet. Damit ergeben sich somit für **level-move** folgende Zahlen:

c	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

diag1 ist eine Liste, die für jede Königin die so errechnete Zahl enthält. Für die erste Lösung enthält **diag1** daher die Zahlen **[-1, -2, 2, 1]**. Jede Zahl darf nur einmal vorkommen – die Königinnen kommen einander in den Diagonalen von links oben nach rechts unten nicht in die Quere

Diagonale 2

Nun zu den Diagonalen von links unten nach rechts oben. Die Zahlen pro Feld werden aus der Summe Zeilennummer + Spaltennummer bestimmt:

c	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

Daher enthält die Liste **diag2** für die erste Lösung die Zahlen **[1, 4, 2, 5]**. Auch hier kommt keine Zahl zweimal vor – die Bedingung für die Diagonalen von links unten nach rechts oben ist somit auch erfüllt.

```
def moves(self, level: int):
    return list(set(range(self.size))-set(self.vector))
```

Code 4

Die Methoden

moves liefert alle Werte, die die Variable move bei den Versuchen (**attempt**) annehmen darf.

- **valid**: Mit den Vereinbarungen wird valid recht einfach: die neue Position einer Königin in Zeile **level** und Spalte **move** ist dann zulässig, wenn weder move in vector enthalten ist noch **level-move** in **diag1** und auch nicht **level+move** in **diag2**.
- **record**: Die den Platz der Königin beschreibenden Zahlen sind an die Listen **vector**, **diag1** und **diag2** anzuhängen. **vector** wird um die Spaltennummer **move** erweitert, **diag1** um die Differenz von Zeilennummer und Spaltennummer, also **level-move** und **diag2** um die Summe der Zeilen- und Spaltennummern **level+move**.
- **undo** wird benötigt, wenn es keine weiteren Möglichkeiten für **move** gibt und daher der Schritt rückgängig zu machen ist. Dazu sind jeweils die letzten Einträge aus den Listen **vector**, **diag1** und **diag2** mit **pop()** zu entfernen.
- **done**: Eine Lösung ist gefunden, wenn in der letzten Zeile eine Königin einen Platz gefunden hat. Für das Anzeigen der Lösung reicht die Ausgabe des **vector**. Mit der Methode **show** wird die Lösung schöner dargestellt.
- Die übersichtliche Ausgabe von Datum und Uhrzeit ist immer wieder eine Herausforderung. Älteren Python-Version borgen sich Funktionen wie **strftime** von C aus. Leider sind diese Funktionen nicht sehr benutzerfreundlich. Mit der **format**-Funktion oder mit **Formatstrings** geht das viel einfacher, wird aber nur selten in Dokumentationen erwähnt. Daher gibt es hier ein Anwendungsbeispiel. Mehr dazu in https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes

Gar nicht kompliziert!

Ergebnis

Folgende Lösungen werden gezeigt:

```
Lösung   1 [1, 3, 0, 2]
.  Q  .  .
.  .  .  Q
Q  .  .  .
.  .  Q  .

Lösung   2 [2, 0, 3, 1]
.  .  Q  .
Q  .  .  .
.  .  .  Q
.  Q  .  .

60 attempts in     0.24 seconds
```

Übrigens: die Berechnung und Ausgabe aller 92 Lösungen des 8-Königinnenproblems dauert nach 15.720 Versuchen 28,9 Sekunden und ohne **beautify** 1,2 Sekunden.

Zurück zu 4 Königinnen: natürlich sind noch weitere Verbesserungen möglich. **moves** liefert als zulässige Spaltennummern alle Zahlen von 0 bis zur Anzahl der Königinnen – 1. Aber Spaltennummern, die schon im **vector** enthalten sind und somit schon vergeben sind, brauchen nicht mehr untersucht zu werden. Das wird mit (**Code 4**) erreicht:

Zuerst wird mit **range** eine Liste **[0, 1, 2, 3]** erzeugt.

Diese Liste wird in eine Menge (**set**) umgewandelt.

Ebenso wird aus der Liste der bisher gefundenen Lösungen (**vector**) in eine Menge gebildet.

Die Differenz dieser beiden Mengen gibt an, welche Spalten noch frei sind.

Die Differenzmenge wird für die weitere Verarbeitung wieder in eine Liste verwandelt.

Diese Version ist in der Datei **queens_2.py** gespeichert.

Nun sind nur mehr 32 Versuche notwendig. Da die meiste Zeit zur Ausgabe der **vector**-Werte gebraucht wird, sinkt die Gesamtzeit nicht wesentlich. Wenn wir in **done** die **print**-Zeile auskommentieren, wird die echte Rechenzeit mit weniger als 0,01 Sekunden angezeigt.

Noch ein paar Zahlen: bei 12 Königinnen werden nach 2.915.740 Versuchen die 14.200 Ergebnisse in 7,0 Sekunden errechnet oder zusammen mit den **vector**-Werten in 175.9 Sekunden angezeigt.

Es gibt viele weitere Varianten, die auf dem Königinnenproblem aufbauen.

Statt der Königinnen könnten Springer eingesetzt werden.

Oder Superköniginnen, die die Eigenschaften der Königin und des Springers vereinigen. Der Name dafür *„Maharadscha“*. Warum eigentlich nicht *„Maharani“*? Ich bin kein Gender-Fan, aber in dem Fall täte ein wenig Allgemeinbildung gut.

Wir brauchen nur die Methode **valid** zu verändern und schon können alle Varianten auch berechnet werden.

Sudoku

Martin Weissenböck

Wenden wir uns den Auslösern für die PCNEWS-Beitrag zu: zu den Rätseln in der Sonntagsausgabe der Presse. Ich vermute, dass die Sudoku-Regeln den meisten Lesern bekannt sind. Trotzdem hier noch einmal eine Kurzfassung:

Beim Sudoku sind in einem 9×9-Quadrat die Zahlen 1 bis 9 so zu verteilen, dass jede Zahl in jeder Zeile, in jeder Spalte und in jedem der neun 3×3-großen Unterquadrate genau einmal vorkommt. Das wäre aber zu leicht. Daher sind die Zahlen einzelner Felder schon vorgegeben. **(Bild oben links)**

3				1				
			5		6	9	8	
		9					1	5
	9	4	3	6	7		5	2
	1	7		9	5	3	4	8
	2	3				6		
4		6		5	2			
9			1		3			4
		5				8		

3	5	8	4	1	9	2	7	6
7	4	1	5	2	6	9	8	3
2	6	9	7	3	8	4	1	5
8	9	4	3	6	7	1	5	2
6	1	7	2	9	5	3	4	8
5	2	3	8	4	1	6	9	7
4	8	6	9	5	2	7	3	1
9	7	2	1	8	3	5	6	4
1	3	5	6	7	4	8	2	9

Dazu bedarf es einer passenden internen Darstellung und einiger Rechenmethoden: Datei **`sudoku.py`**. Die Felder des 9×9-Quadrats (oder „Plätze" im Quadrat) werden der Reihe nach gefüllt. Deshalb ist es praktisch, sie von 0 bis 80 zu nummerieren. Wir haben wieder einen **`vector`**, der zu Beginn mit 81 Nullen gefüllt wird. Jedes Mal, wenn eine neue Zahl vergeben wird, steigen wir bei der Lösung des Problems um einen **`level`** „hinauf". Daher ist der **`level`** gleich dem Index der Liste **`vector`**, die mit einem Wert (aus **`move`**) gefüllt wird.

Um den Ablauf zu beschleunigen und nur jene Zahlen in Betracht zu ziehen, die noch nicht verwendet wurden, legen wir für jede Zeile, für jede Spalte und für jedes Unterquadrat eine Menge (**`set`**) an. Wenn eine Zahl widerspruchsfrei eingefügt werden kann, wird sie im **`vector`** an der Stelle mit dem Index **`level`** gespeichert und in die drei Mengen, die zu diesem Platz gehören, eingefügt. Das macht die Methode **`record`**. Wenn sich später herausstellt, dass die Zahl doch nicht zur Lösung führt, setzt **`undo`** den Eintrag in der Liste **`vector`** auf **`0`** und entfernt die Zahl aus den drei Mengen.

`getSets`: Bei etlichen Vorgängen müssen aus dem Index (dem **`level`**) die Nummer der Zeile, der Spalte und des Unterquadrats bestimmt werden und daraus die zugehörigen Mengen ermittelt werden. Das wird von der Methode **`getSets`** erledigt. Sie errechnet aus dem **`level`** vorerst die Nummern der Zeile (**`rr`**), der Spalte (**`cc`**) und Unterquadrats (**`ss`**).

Die Zeilennummer eines Feldes ist gleich Index (**`level`**), ganzzahlig geteilt (Divisionssymbol „**`//`**") durch die Größe des Quadrats (hier gehen wir von 9 aus, aber das Programm kann auch mit anderen Größen umgehen).

- Die Spaltennummer ist gleich dem Divisionsrest des Index geteilt durch diese Größe.
- Die Nummer des Unterquadrats wird etwas trickreich bestimmt: die Zeilennummer **`rr`** wird ganzzahlig durch die Größe des Unterquadrats (hier: **3**) geteilt und gleich wieder damit multipliziert. Damit erhalten die Unterquadrate die Nummern **`0`**, **`3`** und **`6`**. Danach muss aus der Spalte **`cc`** ebenso errechnet werden, welches der drei nebeneinander liegenden Unterquadrate gemeint ist: demnach wird **`0`**, **`1`** oder **`2`** zu der Zahl aus dem ersten Schritt addiert.

Die Zwischenrechnungen sind abgeschlossen, die Methode **`getSets`** gibt eine Liste mit den drei Sets, die zu **`rr`**, **`cc`** und **`ss`** gehören, zurück.

`moves` liefert eine Liste, die jene Werte enthält, die in das Feld mit der Nummer **`level`** eingefügt werden können und der Reihe nach zu überprüfen sind.

Aber was ist mit den schon vorgegebenen Zahlen? Hier kommt das Dictionary **`sudoku`** zum Einsatz. Python unterscheidet zwischen Groß- und Kleinbuchstaben in Namen, **`Sudoku`** ist der Name der Klasse, **`sudoku`** heißt das Dictionary, das die Aufgabe beschreibt. Für jeden bereits (in der Angabe des Rätsels) belegten Platz bekommt **`sudoku`** einen Eintrag, dessen Schlüssel die Platznummer ist und dessen Wert sich aus der Angabe ergibt. Im Beispiel hat der Platz ganz links oben den Schlüssel **`0`** und den Wert **3** und daher den Eintrag **`sudoku = { 0:3, ... }`**

Ist der Wert von **`level`** als Schlüssel in **`sudoku`** enthalten, ist der zugehörige Wert die einzige Zahl, die **`Moves`** liefert: der Wert ist vorgegeben, daher darf gar nichts anderes erprobt werden. Andernfalls ermittelt **`moves`** alle jene Zahlen, die nicht in jenen drei Mengen enthalten sind, die von **`getSets`** ermittelt worden sind. Dabei wird wieder die Mengendifferenz verwendet.

`valid` bestimmt, ob die einzufügende Zahl (**`move`**) widerspruchsfrei in das Sudoku-Quadrat eingefügt werden kann. Das ist ganz einfach: **`valid`** liefert **`True`**, wenn die Zahl in keiner der drei Mengen, die zu dem Platz mit der Nummer **`level`** gehören, enthalten ist.

`done` untersucht, ob **`level`** den Höchstwert erreicht hat. Wenn ja haben wir eine Lösung, die mit **`show`** (zusammen mit einem erklärenden Text) ausgegeben wird.

Ja, das Ganze ist schon etwas komplizierter als das Platzieren von 8 Königinnen. Aber mit der objektorientierten Programmierung ist alles recht sauber getrennt.

Und hier das Ergebnis:

```
Resultat    1
[3, 5, 8, 4, 1, 9, 2, 7, 6]
[7, 4, 1, 5, 2, 6, 9, 8, 3]
[2, 6, 9, 7, 3, 8, 4, 1, 5]
[8, 9, 4, 3, 6, 7, 1, 5, 2]
[6, 1, 7, 2, 9, 5, 3, 4, 8]
[5, 2, 3, 8, 4, 1, 6, 9, 7]
[4, 8, 6, 9, 5, 2, 7, 3, 1]
[9, 7, 2, 1, 8, 3, 5, 6, 4]
[1, 3, 5, 6, 7, 4, 8, 2, 9]
              6_432_362 attempts in
41.54 seconds
```

Somit ist das die einzige Lösung, grafisch aufbereitet: (**Bild oben rechts**)

Erweiterungen

- Auf janko.at (https://www.janko.at/Raetsel/) oder puzzlephil (https://puzzlephil.com/) sind viele interessante Rätsel zu finden. Dazu zählen Sudokus, bei denen statt der Unterquadrate oder zusätzlich zu den Unterquadraten andere Formen, meist durch Farben gekennzeichnet, verwendet werden. Mit einer geänderten **`valid`**-Methode können diese Sudokus auch rasch gelöst werden.
- Auch Sudokus in der Form eines sechsstrahligen Sterns löst das Programm. An dieser Stelle zeigt sich, dass die Darstellung als vector für alle Formen geeignet ist. Wäre die Aufgabe mit einem zweidimensionalen Feld gelöst worden, wäre der Wechsel zu einem Stern nicht einfach.

Verbesserungsmöglichkeiten

- Wie beim Beispiel der Königinnen werden die Größe und die Variable **`sudoku`** als Parameter übergeben. Aufruf mit **`Sudoku(size=9, sudoku= { 0:3, 4:1, 12:5, 14:6, 15:9, 16:8, 20:9, 25:1, 26:5,…})`**. Zu finden in **`sudoku2_py`**.
- Die Ausgabe könnte grafisch schöner sein.

Ich habe auf all das verzichtet, um das Programm nicht weiter zu vergrößern.

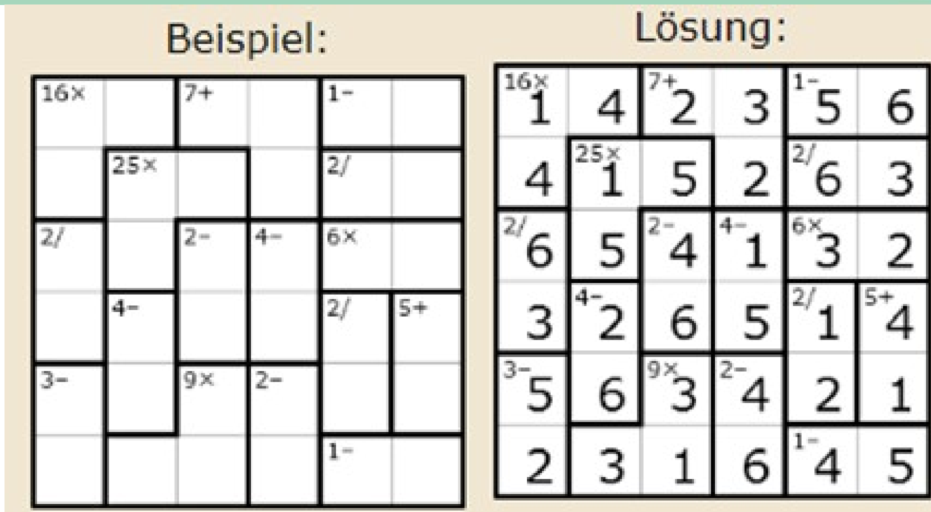
Es gibt noch genug zu tun!

```python
# File sudoku.py
from backtrack import Backtrack
import math

class Sudoku(Backtrack):

    vector = []
    size = 9
    sqrtSize = int(math.sqrt(size))
    solutionNr = 0


    rset = [set() for i in range(size)]  # row sets
    cset = [set() for i in range(size)]  # column sets
    sset = [set() for i in range(size)]  # subsquare sets

    sudoku = { # leicht
        0:3, 4:1,
        12:5, 14:6, 15:9, 16:8,
        20:9, 25:1, 26:5,
        28:9, 29:4, 30:3, 31:6, 32:7, 34:5, 35:2,
        37:1, 38:7, 40:9, 41:5, 42:3, 43:4, 44:8,
        46:2, 47:3, 51:6,
        54:4, 56:6, 58:5, 59:2,
        63:9, 66:1, 68:3, 71:4,
        74:5, 78:8
        }

    values = range(1, size+1)

    def __init__(self, test:bool= False, showAttempts:bool = False):
        self.vector = [0 for i in range(self.size*self.size)]
        super().__init__(test, showAttempts)

    def getSets(self, level:int):
        s = self.size
        sq = self.sqrtSize

        # Zeilennummer bestimmen: 0 bis size-1
        rr = level // s

        # Spaltennummer bestimmen: 0 bis size-1
        cc = level % s

        # Nummer des Unterquarats bestimmen: 0 bis size-1
        ss = rr // sq * sq + cc // sq

        return [self.rset[rr], self.cset[cc], self.sset[ss]]

    def moves(self, level: int):
        res = self.sudoku.get(level, 0)
        if res:
            return [res]
        return list(set(self.values) - set().union(*self.getSets(level)))

    def valid(self, level: int, move) -> bool:
        return move not in set().union(*self.getSets(level))

    def done(self, level: int):
        if level==self.size*self.size-1:
            self.solutionNr += 1
            self.show(f"Resultat {self.solutionNr:4d}")
            return True
        return False

    def record(self, level: int, move):
        self.vector[level]=move
        for s in self.getSets(level):
            s.add(move)

        if self.test:
            print (level, move, rr, cc, ss, self.rset[rr], self.cset[cc], self.sset[ss])

    def undo(self, level: int, move):
        self.vector[level]=0
        for s in self.getSets(level):
            s.discard(move)

    def show(self, text):
        print(f"\n{text}")
        for r in range(self.size):
            print(self.vector[r*self.size : (r+1)*self.size])

if __name__ == "__main__":
    Sudoku(test=False, showAttempts=True)
```

Kendoku

Martin Weissenböck

Kendoku gehört zu den Zahlenrätseln. Hier ein Beispiel aus der Webseite von **Angela** und **Orro Janko** https://www.janko.at/Raetsel/Kendoku.



Die Aufgabe

Siehe Bild „Beispiel".

Zwei Bedingungen sind zu erfüllen:

1. In die Felder sind natürliche Zahlen von 1 bis zur Größe des Quadrats einzutragen. Wie üblich muss in dem Quadrat jede Zahl in jeder Spalte und in jeder Zeile genau einmal vorkommen.
2. Unregelmäßig geformte Bereiche in einem Quadrat enthalten einen Wert und eine Rechenoperation (+,-,*,/). Die Zahlen in dem Bereich müssen zusammen mit der Rechenoperation den angegebenen Wert ergeben.

Die erste Herausforderung für Mathematiker besteht darin, dass die Zahlen bei „-" und „/" vertauscht werden können: wenn beispielsweise die Angabe „**2/**" lautet, wären **(6,3)**, **(3,6)**, **(4,2)**, **(2,4)**, **(2,1)** und **(1,2)** gültige Lösungsansätze.

Die Lösung der Beispielaufgabe siehe Bild „Lösung".

Die Lösung per Programm

Dateiname: **kendoku.py**. Wie üblich verwenden wir einen **vector**, der Schritt für Schritt gefüllt wird. Mit jedem Schritt wird der **level** erhöht.

Die erste Bedingung ist leicht zu erfüllen. Wie im Königinnenproblem verwenden wir Mengen (**rset** und **cset**).

Die zweite Bedingung kann erst dann geprüft werden, wenn das letzte Element eines Bereichs zu füllen ist. Dazu bestimmen wir den höchsten Index jedes Bereichs. Wenn beim Füllen eines Bereichs **level** diesen Index erreicht, startet in **valid** die Prüfung des errechneten Wertes.

Um die Aufgabe zu beschreiben, verwenden wir ein Tupel **kendoku**, das seinerseits für jeden Bereich ein Tupel enthält: Jedes dieser Tupel enthält den Wert, den die Rechnung ergeben soll, die Rechenoperation und die Nummer der Felder, die zu dem Bereich gehören. Natürlich beginnt die Nummerierung der Felder mit 0. Statt der Tupel können auch Listen verwendet werden.

In der Beispielaufgabe sieht das dann so aus:

```
kendoku = (
        (16, "x", ( 0,  1,  6)),
        ( 7, "+", ( 2,  3,  9)),
        ( 1, "-", ( 4,  5)),  …
    )
```

Diese Darstellungsform ermöglicht, das Rätsel mit geringem Schreibaufwand zu formulieren. Da wir den höchsten Wert der Feldnummer brauchen, wird **kendoku** mit der Methode **convert** in die interne Darstellung **_kendoku** umgewandelt. Die ist auch leichter lesbar. **_kendoku** enthält dann in unserem Beispiel

```
_kendoku = {
        6: {"value":16, "op":"x", "members":(0, 1, 6)},
        9: {"value":7, "op":"+", "members":(2, 3, 9)},
        5: {"value":1, "op":"-", "members":(4, 5)}, …
 }
```

Das Programm

Der Ablauf des Programms ist im Programmtext ausführlich kommentiert.

Besonderheiten von Python

Hier ein paar Hinweise auf Besonderheiten und auf die verwendeten Version 3.10.6:

In Python können die Typen von Variablen, Parametern und Funktionsergebnissen angegeben werden, sogenannte type hints (https://docs.python.org/3/library/typing.html). Es ist aber nicht verpflichtend, sondern dient der besseren Lesbarkeit und der Kontrolle durch das externe Programm. In den Programmen werden diese type hints mehrmals beispielhaft verwendet.

Beispiele

```
# Typbeschreibungen
    Places = tuple[int]
    Kendokutuple =
       tuple[int, str, Places]
    Moves = list[int]
def moves(self, level: int) -> Moves:
    …
```

Da bei „+" und „x" mehr als zwei Zahlen in einem Bereich vorkommen können, wird zur Berechnung **reduce** eingesetzt. **reduce** wendet eine Funktion (hier eine Rechenoperation) der Reihe nach auf alle Elemente der Zahlenliste an. Die Argumente von **reduce** sind:

- Die Funktion, die anzuwenden ist (hier sind es die Rechenoperationen).
- Die Liste der Werte, auf die die Funktion anzuwenden ist.

reduce kennt aber die Operatoren „+", „x" nicht. Sie werden durch die Funktionen **add**, **mul** ersetzt. Ebenso werden **sub** und **floordiv** (für die ganzzahlige Division) verwendet. Die vier Funktionen werden von **operator** importiert.

Statt dieser Funktionen könnte auch die **lambda**-Funktion eingesetzt werden. Die Anwendung lautet bei der Addition: **reduce (lambda(x,y): x+y, area)**. Die **lambda**-Funktion ist eine anonyme Funktion, die sinnvoll dort eingesetzt wird, wo eine Funktion nur einmal gebraucht wird.

Die Schreibweise **liste[0:-1]** bedeute: alle Elemente der Liste, ausgenommen das letzte.

In Python 3.10 wurde match eingeführt. match kann zwar mit **switch**-Befehlen anderer Sprachen verglichen werden, ist aber viel mächtiger. **match** wird in vielen Artikeln im Internet erklärt, zum Beispiel hier: https://hellocoding.de/blog/coding-language/python/pattern-matching. Ich empfehle, dieses neue Sprachelement genau zu studieren – es ist eine wirklich nützliche Erweiterung.

Da in Kendoku auch Bereiche ohne Rechenaufgaben zulässig sind, diese Bereiche also beliebig gefüllt werden können, endet die match-Auswahl mit

```
case _:  # Keine Operation angegeben
         return True
```

Wird bei einem Funktionsaufruf als Argument eine Liste mit vorangestelltem ***** angegeben, ist das gleichwertig zur Angabe aller Listenelemente als einzelne Argumente:

```
self.floordiv(*area)==value
```

Auch hier können **size** und **kendoku** als Argumente beim Aufruf übergeben werden. Eine Version mit diesen Argumenten und mit **lambda**-Funktionen sind in der Datei **kenduku_2.py** zu finden.

Lösung

Hier die Lösung, vom Programm errechnet:

```
Lösung    1
  1    4    2    3    5    6
  4    1    5    2    6    3
  6    5    4    1    3    2
  3    2    6    5    1    4
  5    6    3    4    2    1
  2    3    1    6    4    5
4.343 attempts in 0.20 seconds
```

Lust auf mehr? Hier kommt ein Rätsel mit Hochhäusern.

```python
# File kendoku.py
from backtrack import Backtrack

class Kendoku(Backtrack):
    # Beispiel: https://www.janko.at/Raetsel/Kendoku/009.a.htm

    import functools
    from functools import reduce
    from operator import abs, add, sub, mul, floordiv

    size = 6
    """
    Jeder Kendoku-Bereich wird durch ein Tupel codiert:
    - Der erwartete Wert
    - Die Rechenoperation (zulässig: + - x X * / )
    - Die Nummern der Felder, die zu dem Bereich gehören, als Tupel.
      Jede Feldnummer muss in genau einem der Tupel genau einmal vorkommen
    """
    kendoku = (
        (16, "x", ( 0,  1,  6)),
        ( 7, "+", ( 2,  3,  9)),
        ( 1, "-", ( 4,  5)),
        (25, "x", ( 7,  8, 13)),
        ( 2, "/", (10, 11)),
        ( 2, "/", (12, 18)),
        ( 2, "-", (14, 20)),
        ( 4, "-", (15, 21)),
        ( 6, "x", (16, 17)),
        ( 4, "-", (19, 25)),
        ( 2, "/", (22, 28)),
        ( 5, "+", (23, 29)),
        ( 3, "-", (24, 30)),
        ( 9, "x", (26, 31, 32)),
        ( 2, "-", (27, 33)),
        ( 1, "-", (34, 35))
    )


    _kendoku = {} # Interne Darstellung als dict

    # Zwischenergebnisse und Lösung
    # Obwohl eine Zahlenmatrix gesucht wird, wird zur internen Berechnung
    # ein eindimensionaler Vektor verwerdet.
    # Zeilen- und Spaltennummern werden mit der Funktion rowAndCol
    # aus der Platznummer (level) errechnet

    vector = [None for i in range(size*size)]

    # Welche Werte können in den einzelnen Feldern stehen?
    values = set(range(1, size+1))

    # In rset und cset werden jene Werte gespeichert, die in der jeweiligen Zeile
    # oder Spalte schon vorkommen und daher in moves nicht mehr zur Auswahl stehen
    rset = [set() for i in range(size)]  # row sets
    cset = [set() for i in range(size)]  # column sets

    # Zählen der Lösungen
    solutionNr = 0

    # Typbeschreibungen
    Places = tuple[int]
    Kendokutuple = tuple[int, str, Places]
    Moves = list[int]

    def __init__(self, test:bool= False, showAttempts:bool = False):
        self.convert ()
        super().__init__(test, showAttempts)

    # Für die Prüfung eines Bereichs wird die Variable kendoku in eine
    # interne Darstellung (_kendoku) als dict mit dem höchsten Index
    # als key umgewandelt
    def convert(self):
        # suche die höchste Feldnummer und verwende sie als Index
        for k in self.kendoku:
            # Höchste Platznummer als key festlegen
            key = max(k[2])

            # Durch die Verwendung eines dict lesbarer machen
            self._kendoku[key] = {"value":k[0], "op":k[1], "members":k[2]}

    # Die Hilfsfunktion roeAndCol errechnet aus der Platznummer (where)
    # die Zeilennummer (r) und die Spaltennummer (c)
    def rowAndCol(self, where: int):
```

```
        s = self.size
        r = where // s
        c = where % s
        return (r, c)

    # Bestimme die Werte, die für den Platz (level) zulässig sind
    def moves(self, level: int) -> Moves:
        r, c = self.rowAndCol(level)
        # Werte, die in derselben Spalte oder Zeile vorkommen, werden ausgeschlossen
        return list(self.values – self.rset[r] – self.cset[c])

    # Ist der Wert (move) auf dem vorgesehenen Platz (level) erlaubt?
    def valid(self, level: int, move: int) -> bool:

        # Falls der Platz nicht der letzte des Bereichs ist: weitersuchen!
        # Dazu prüft get, ob level ein gültiger Key von _kendoku ist.

        k = self._kendoku.get(level, False)

        # Wenn nein, kann die Rechenopration in dem
        # Bereich noch nicht ausgeführt werden. Weitersuchen!

        if not k:
            return True

        # Ist der key vorhanden, dann kann der Test beginnen
        # Die Nummern der Felder des Bereichs sind in members zu finden.
        # Nun müssen zur Berechnung alle Werte des Bereichs
        # (ohne das letzte) in der Liste area zwischengespeichert werden.

        area = []
        for i in k["members"][0:-1]:
            area.append(self.vector[i])

        # Das letzte Feld wird probeweise dazu genommen
        area.append(move)

        # Nun wird die Berechnung in dem Bereich ausgeführt.
        # Berechnungen, abhängig vom Operator (+ – …)
        value = k["value"]
        match(k["op"]):
            case "*" | "x" | "X":
                return self.reduce(self.mul, area)==value
            case "/" | ":":
                return self.floordiv(*area)==value or\
                    self.floordiv(*area[-1:-3:-1])==value
            case "+":
                return self.reduce(self.add, area)==value
            case "-":
                return self.abs(self.sub(*area))==value
            case _:  # Keine Operation angegeben
                return True


    # Überprüften (validierten) Wert (move) auf den Platz (level) speichern
    def record(self, level: int, move: int):
        self.vector[level]=move
        r, c = self.rowAndCol(level)
        self.rset[r].add(move)  # Wert zur Menge der Zeilenwerte hinzufügen
        self.cset[c].add(move)  # Wert zur Menge der Spaltenwerte hinzufügen

    # Der Wert (move) passt nicht, daher vom Platz (level) entfernen
    def undo(self, level: int, move: int):
        self.vector[level]=None
        r, c = self.rowAndCol(level)
        self.rset[r].remove(move)  # Wert aus der Menge der Zeilenwerte entfernen
        self.cset[c].remove(move)  # Wert aus der Menge der Spaltenwerte entfernen

    # Wurde eine Lösung gefunden?
    def done(self, level: int):
        if level==self.size*self.size-1:
            self.solutionNr += 1
            print(f„"Lösung {self.solutionNr:4}")
            for r in range(self.size):
                for c in range(self.size):
                    print(f"{self.vector[r*self.size + c]:4} ", end="")
                print()
            return True
        return False

if __name__ == "__main__":
    Kendoku(showAttempts=True)
```

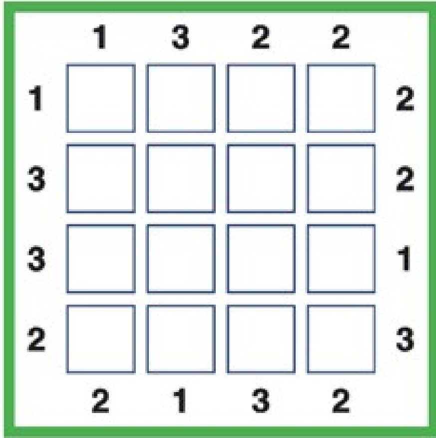
Skyline

Martin Weissenböck

Ich bin auf das Skyline-Rätsel über die Sonntagausgabe der Presse aufmerksam geworden. Auf der Seite puzzlephil https://puzzlephil.com/puzzles/skyline/ de/ finden sich ebenfalls derartige Rätsel, manchmal auch „Stadtteil“ oder „Stadtviertel“ genannt. Ich stelle hier ein leichtes und ein besonders schwieriges Rätsel vor.

„Leicht“

Das Bild zeigt mehrere Hochhäuser von oben. Die Regeln lauten:



1. Die Hochhäuser sind jeweils 10, 20, 30 … Stockwerke hoch. Jede Gebäudehöhe kommt in jeder Zeile und jeder Spalte genau einmal vor.
2. Die Zahlen am Rand geben an, wie viele Gebäude man von links (von Westen), von rechts (von Osten) usw. sieht. Höhere Gebäude verbergen niedrigere Gebäude. Die Zahl 0 oder eine fehlende Zahl bedeutet, dass die Anzahl der sichtbaren Hochhäuser nicht bekannt ist.

Die mit dem Programm **skyline.py** bestimmte Lösung für die einfache Aufgabe lautet:

```
[40, 20, 10, 30]
[10, 30, 40, 20]
[20, 10, 30, 40]
[30, 40, 20, 10]
```

Vom Westen aus sehe ich

- 1 Hochhaus (40 Stockwerke), dann
- 3 Hochhäuser (10, 30, 40 Stockwerke), dann wieder
- 3 Hochhäuser (20, 30, 40 Stockwerke) und
- 2 Hochhäuser (30, 40 Stockwerke).

Schaut nach einer interessanten Programmieraufgabe aus!

Beginnen wir mit der Codierung der Aufgabenstellung. Das ist einfach, die vier Randbeschriftungen reichen zur Beschreibung. Sie werden als Listen angelegt:

```
west  = [1, 3, 3, 2]
east  = [2, 2, 1, 3]
north = [1, 3, 2, 2]
south = [2, 1, 3, 2]
```

Mit der Länge der Listen ist auch die Größe des Stadtteils (**size**) bestimmt: in diesem Fall ist sie gleich **4**. Der Stadtteil wird als eindimensionaler **vector** gespeichert, aus dem bei Bedarf einzelne Zeilen und Spalten errechnet werden.

Nun zu den Regeln:

- Regel 1 wird wie bei vorhergehenden Rätseln umgesetzt.
- Für die Regel 2 müssen die sichtbaren Hochhäuser gezählt werden. Da das Zählen und Prüfen für alle vier Richtungen auszuführen sind, ist eine eigene Funktion (**check**) dafür sinnvoll. Aber wie werden Zeilen von links nach rechts und von rechts nach links, Spalten von oben nach unten und von unten nach oben abgearbeitet?

Beispiel

v = [0, 1, 2, 3, 4, 5, 6, 7]

v[0:4] oder **v[0:4:1]** ist der Teilbereich **[0, 10, 20, 30]**

In dem Beispiel ist **0** der Startindex, **4** der Stoppindex und **1** die Schrittweite. Zum Stoppindex (**4**) gehört das Element **v[4]**, ist gleich **40**. Es gehört nicht mehr zu dem Teilbereich.

Geht es auch in der umgekehrten Richtung?

v[4:0:-1] ergibt **[40, 30, 20, 10]**, also noch nicht den gewünschten Teilbereich. Nun ist **v[-8]** gleich **0**. Ferner ist bei einem Teilbereich der Wert an der Obergrenze nicht enthalten. Daher liefert erst

v[-5:-9:-1] oder **v[3:-9:-1]** wie gewünscht **[30, 20, 10, 0]**.

Wie kann man das ganze v umdrehen?

v[8:-1:-1] ergibt überraschend **[ ]**, aber **v[8:-9:-1]** dreht **v** um:
[70, 60, 50, 40, 30, 20, 10, 0]

Damit können wir Zeilen oder Teile von Zeilen von links nach rechts oder von rechts nach links abarbeiten. Wie kommen wir zu den Spalten?

Nehmen wir an, dass **v** einer Matrix mit 2 Zeilen und 4 Spalten darstellt. Die Spalte **0** wird mit **v[0:5:4]** aufgerufen (Resultat **[0, 40]**), die Spalte **1** mit **v[1:6:4]** (ergibt **[10, 50]**) usw.

Und in der umgekehrten Richtung?

v[4:-9:-4] ergibt **[4, 0]**, **v[5:-9:-4]** ergibt **[50, 10]**, **v[6:-9:-4]** ergibt **[60, 20]** und **v[7:-9:-4]** liefert **[70, 30]**

Die Indexausdrücke von **v** (wie etwa **[4:-9:-4]**) können jedoch nicht als Argumente an die Funktion **check** übergeben werden. Stattdessen ist ein **slice**-Objekt zu verwenden, zum Beispiel **slice(4, -9, -4)**.

v[slice(4, -9, -4)] ergibt wieder **[40, 0]**.

Hinweis: Auch die Zuweisung des **slice**-Objekts an eine Variable ist möglich:

```
sl = slice(4, -9, -4)
v[slice]
```

Damit kann die Methode **valid** recht übersichtlich geschrieben werden. Nur die Argumente von **slice** sind nicht wirklich selbsterklärend.

Programm

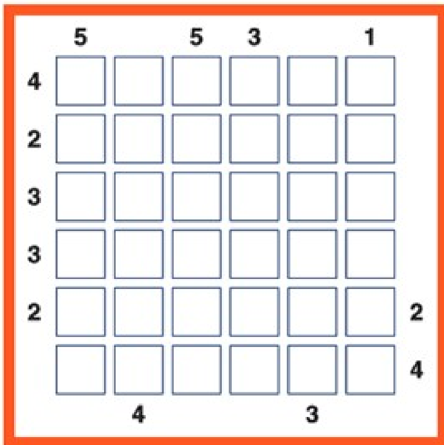
Die Datei **skyline_2.py** zeigt eine weitere Version mit der Aufgabenstellung als Argument beim Start von **Skyline**.

„Teuflisch“

Manche Skyline-Rätsel sind schon sehr aufwändig. Das „teuflische Rätsel“ und seine Lösung ist in der Datei **skyline_3.py** zu finden. Die vielen Varianten, die zu prüfen sind, werden mit je einem Punkt pro 100.000 Versuche angezeigt.

CLUBDEV.NET

TEUFLISCH



Mit

```
west  =  [4,2,3,3,2,0]
east  =  [0,0,0,0,2,4]
north =  [5,0,5,3,0,1]
south =  [0,4,0,0,3,0]
```

sind 1.316.961 Versuche notwendig, um diese Lösung zu finden:

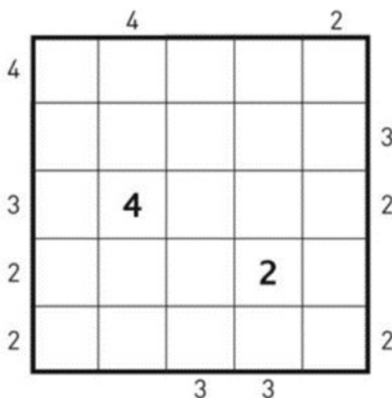
```
[20, 40, 10, 30, 50, 60]
[30, 60, 20, 10, 40, 50]
[40, 20, 30, 50, 60, 10]
[10, 50, 40, 60, 30, 20]
[50, 30, 60, 20, 10, 40]
[60, 10, 50, 40, 20, 30]
```

Gibt es noch weitere Lösungsvektoren? Es kann nur einen geben! ((C) Highlander)

Das Programm prüft trotzdem weiter, findet (wie erwartet) keine weitere Lösung und benötigt dazu insgesamt 56.388.851 Überprüfungen in 618 Sekunden.

Vorgegebene Felder

In einer Variante von **Skyline** sind einzelne Felder mit Häusern bekannter Höhe besetzt. Zur Lösung hilft dieselbe Überlegung wie beim Sudoku. Die vorgegebenen Felder werden in einem Dictionary **known** mit den Feldnummern als Schlüssel und den Höhen als Werte.



Beispiel: das Rätsel „Stadtviertel“ aus der Kurier-Beilage „Freizeit“ vom 20. August 2022:

Hier sind die Häuser nur **1, 2, 3, 4** oder **5** Stockwerke hoch. Mit dem Argument **high=1** wird das beim Aufruf berücksichtigt. Der Lösungsweg ist in **skyline_4.py** zu finden.

```python
# File skyline.py
from backtrack import Backtrack
import sys

class Skyline(Backtrack):

    vector = []
    solutionNr = 0
    high = 10

    # Siehe https://puzzlephil.com/puzzles/skyline/de/

    # leicht
    west =   [1,3,3,2]
    east =   [2,2,1,3]
    north =  [1,3,2,2]
    south =  [2,1,3,2]

    def __init__(self, test:bool = False, showAttempts:bool = False):

        if not(len(self.west) == len(self.east) == len(self.north) == len(self.south)):
                print("Ungleiche Randspalten - Programm beendet")
                sys.exit()
        self.size = len(self.west)
        self.vector = [0 for i in range(self.size*self.size)]
        self.showAttempts = showAttempts
        self.test = test
        self.rset = [set() for i in range(self.size)]
        self.cset = [set() for i in range(self.size)]
        self.values = set(range(self.high, (self.size+1)*self.high, self.high))
        super().__init__(test, showAttempts)

    def rowAndCol(self, where:int):
        s = self.size
        r = where // s
        c = where % s
        return (r, c)

    def moves(self, level:int):
        r, c = self.rowAndCol(level)
        return list(self.values - self.rset[r] - self.cset[c])

    def valid(self, level:int, move:int) -> bool:
        r, c = self.rowAndCol(level)
        s = self.size

        # Jede der vier Richtungen muss einzeln geprüft werden
        # Daher ist der Einsatz einer eigenen Funktion check sinnvoll
        # Argumente:
        # visible: wieveiel Hochhäuser sind sichtbar?
        # sl: Argumente für slice
        def check(visible:int, sl:list) -> bool:

            if visible==0:
                # Unbekannte Anzahl, daher ist die Annahme nicht falsch
                return True

            max = 0
            count = 0

            # Wenn sl[2] gleich 1 oder -1 ist, wird eine Zeile untersucht
            rowtest = abs(sl[2])==1
            rc = c if rowtest else r

            # Letzte Spalte oder letzte Zeile erreicht?
            if rc+1 == s:
                # Zählen der sichtbaren Häuser
                for v in self.vector[slice(*sl)]:
                    if v > max:
                        # Ist das Haus höher als die vorhergehenden?
                        # Dann count um 1 erhöhen
                        max = v
                        count += 1

                # Sollten Zwischenergebnisse zum Testen ausgegeben werden?
                if self.test:
                    self.show(f"level={level} r={r} c={c} move={move} attempt={self.attempts}"
                              f"\nrowtest={rowtest} max={max} count={count} visible={visible} \n"
                              f"slice={sl} ==> {self.vector[slice(*sl)]}")

                # Wurden exakt die Häuser laut Angabe gefunden?
                if count != visible:
                    return False
            return True

        self.vector[r*s + c] = move  # move probeweise eintragen
        res = all([
            check(self.west[r],  [r*s, (r+1)*s, 1]),
            check(self.east[r],  [-(s-r-1)*s-1, -(s-r)*s-1, -1]),
            check(self.north[c], [c, c+s*s, s]),
            check(self.south[c], [-(s-c), -((s+1)*s-c), -s])
        ])

        if self.test and (r+1==self.size or c+1==self.size) and res:
            self.show(f"valid={res}, level={level}, r={r}, c={c},  move={move}, attempt={self.attempts}")

        self.vector[r*s + c] = 0  # move wieder entfernen

        return res

    def record(self, level:int, move:int):
        self.vector[level]=move
        r, c = self.rowAndCol(level)
        self.rset[r].add(move)  # Wert zur Menge der Zeilenwerte hinzufügen
        self.cset[c].add(move)  # Wert zur Menge der Spaltenwerte hinzufügen


    def undo(self, level:int, move:int):
        self.vector[level]=0
        r, c = self.rowAndCol(level)
        self.rset[r].remove(move)  # Wert aus der Menge der Zeilenwerte entfernen
        self.cset[c].remove(move)  # Wert aus der Menge der Spaltenwerte entfernen

    def done(self, level: int)->bool:
        if level==self.size*self.size-1:
            self.solutionNr += 1
            self.show(f"Resultat {self.solutionNr:4d}, {self.attempts:18_d}. Versuch")
            return True
        return False

    def show(self, text):
        print(f"\n{text}")
        for r in range(self.size):
            for c in range(self.size):
                print(f"{self.vector[r*self.size + c]:4d}", end="")
            print()

if __name__ == "__main__":
    Skyline(False, True)
```

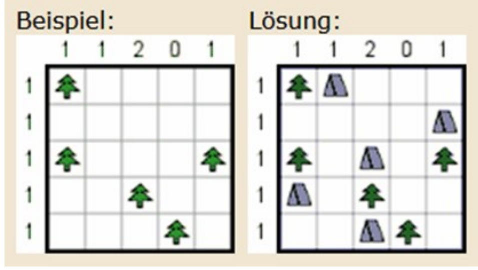
Zeltlager

Martin Weissenböck

Dieses Rätsel schaut harmlos aus, hat es aber in sich. Die Aufgabe: in einem rechteckigen oder quadratischen Feld sind Bäume eingezeichnet.

- Zu jedem Baum ist genau ein Zelt daneben (das heißt links, rechts, oberhalb oder unterhalb des Baumes) aufzustellen.
- Zwei Zelte dürfen nicht waagrecht oder senkrecht nebeneinanderstehen. Diagonal ist erlaubt.
- In jeder Zeile und jeder Spalte ist angegeben, wie viele Zelte dort zu stehen haben.

Hier wieder ein Beispiel aus der schönen Sammlung von **Angela** und **Otto Janko** (https://www.janko.at/Raetsel/Zeltlager):



Datenstruktur

In den bisherigen Programmen wurde bisher der nächste **`level`** mit dem Eintragen einer gültigen Zahl (oder einer Königin) in ein Feld erreicht. Hier ist es sinnvoller. jedem Baum eine Nummer zuzuordnen, die gleich dem **`level`** ist, und den **`level`** nach der erfolgreichen Zuordnung eines Zelts zu erhöhen. Alle Felder werden wieder Spalte für Spalte und Zeile für Zeile nummeriert. Jeder Baum erhält eine Platznummer, aus der seine Koordinaten Zeile (**`r`**) und Spalte (**`c`**) errechnet werden. Mit **`moves`** werden jene Plätze bestimmt, auf die ein Zelt gestellt werden kann. **`valid`** prüft, ob dabei die Bedingungen eingehalten werden. Die Hilfsfunktion **`neighbors`** bestimmt zu einer Feldnummer die Nummern der Nachbarn, also aller horizontal und vertikal (aber nicht diagonal) umgebenden Felder.

Das Beispiel oben enthält 25 Felder (von 0 bis 24 nummeriert). Die Bäume stehen auf den Feldern 0, 10, 14, 17 und 23. Diese Nummern werden als Liste **`trees`** gespeichert. Nachbarn von 10 wären beispielsweise 5, 11 und 15. Die geforderte Anzahl der Zelte wäre in den Zeilen (**`rows`**) 1, 1, 1, 1, 1 und in den Spalten (**`cols`**) 1, 1, 2, 0, 1. Somit wird dieses Rätsel durch die drei Listen

```
trees = [0, 10, 14, 17, 23]
rows =  [1, 1, 1, 1, 1]
cols =  [1, 1, 2, 0, 1]
```

beschrieben.

Programm zeltlager.py

Siehe nächste Seite

Ergebnisse

Das Programm (**`zeltlager.py`**) findet zwei Lösungen, hier mit den Randbeschriftungen:

```
Lösung     1

            1  1  2  0  1
 1 ->       B  Z  .  .  .
 1 ->       Z  .  .  .  .
 1 ->       B  .  Z  .  B
 1 ->       .  .  B  .  Z
 1 ->       .  .  Z  B  .
Lösung     2

            1  1  2  0  1
 1 ->       B  Z  .  .  .
 1 ->       .  .  .  .  Z
 1 ->       B  .  Z  .  B
 1 ->       Z  .  B  .  .
 1 ->       .  .  Z  B  .

44 attempts in      0.50 seconds
```

B steht Baum und **Z** für Zelt. Nette Symbole aus dem Unicode-Zeichensatz wären besser, erfordern aber mehr Aufwand, da sie keine einheitliche Zeichenbreite aufweisen.

`zeltlager_2.py` zeigt ein anderes Beispiel, wieder mit der Übergabe der Problembeschreibung als Argumente an Zeltlager.

`zeltlager_3.py` enthält eine neue Angabe für eine schwierige Aufgabe. Die Lösungen 2 und 3 sehen gleich aus. Allerdings werden Bäume und Zelte unterschiedlich zugeordnet. Um das anzuzeigen, wird hier anstelle von „**Z**" die Nummer des Baums angezeigt, der zu dem Zelt gehört. Das zeigt, dass in den Lösungen 2 und 3 die Baumnummern 11 und 14 vertauscht sind.

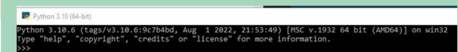
Python

Martin Weissenböck

Wir wollen Python ausprobieren. Aber wie?

Nach der Installation (siehe https://www.python.org/downloads/) kann Python sofort von der Kommandozeile aus gestartet werden. Wenn Python auf `C:\Python310` installiert ist, reicht

`C:\Python310\python.exe` und Python meldet sich mit



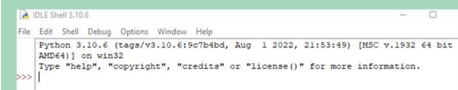
Wenn wir in ersten Experimenten Python als Rechengerät verwenden wollen, geht das ganz einfach, sogar Funktionen können damit vereinbar werden:

```
>>> 3 + 4
7
>>> 2 **20
1048576
>>>
```

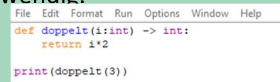
```
>>> def doppelt(i):
...     return i*2
...
>>> doppelt(3)
6
>>>
```

Aber jedes Ändern einer Funktion, allgemein jedes Editieren wird sehr mühsam.

Die Python-Installation enthält aber auch eine nette Entwicklungsumgebung *("IDLE is Python's Integrated Development and Learning Environment.")* . Sie ist mit `C:\Python310\Lib\idlelib\idle.bat` aufzurufen und führt zu:



Zum Einstieg ideal: keine zusätzliche Installation, Speichern und Laden von Programmen einfach möglich, die Syntax wird farblich hervorgehoben, eine Hilfefunktion und sogar ein Debugger sind dabei: so klein wie möglich und so groß wie notwendig.



Nach oben hin gibt es fast keine Grenze für die Funktionen und die Komplexität von Entwicklungssystemen. Um nur ein paar zu nennen:
Notepad++
https://notepad-plus-plus.org/downloads/,
Microsoft Visual Studio
https://visualstudio.microsoft.com/de/,
PyCharm
https://www.jetbrains.com/de-de/pycharm/.

Ich verzichte hier auf Screenshots - bitte die angegebenen Links aufrufen.

Wer beispielsweise mit Microsofts Visual Studio schon in anderen Programmiersprachen gearbeitet hat, wird sich rasch zurecht finden. Wer seine ersten Programmierversuche damit machen will, ist in fast allen Fällen hoffnungslos überfordert und verwirrt. Daher mein Rat: klein anfangen!

```python
# File zeltlager.py

from backtrack import Backtrack

class Zeltlager(Backtrack):

    trees = [0, 10, 14, 17, 23]
    rows =  [1, 1, 1, 1, 1]
    cols =  [1, 1, 2, 0, 1]

    # Auch rechteckige Plätze sind möglich, daher zwei Größenangaben
    rsize = len(rows)
    csize = len(cols)

    tents = []  # Zu Beginn sind keine Zelte aufgebaut
    solutionNr = 0      # Nummer der Lösung
    rowscount = [0 for i in range(rsize)]  # Aktuelle Anzahl der Zelte pro Zeile
    colscount = [0 for i in range(csize)]  # Aktuelle Anzahl der Zelte pro Spalte

    def __init__(self, test:bool= False, showAttempts:bool = False):
        super().__init__(test, showAttempts)

    def rowAndCol(self, where:int) -> tuple:
        s = self.rsize
        return (where // s, where % s)

    def neighbors(self, place:int) -> list[int]:

        r, c = self.rowAndCol(place)
        s = self.rsize
        n = [ ]

        if c+1<s:
            n.append(place+1)
        if r>0:
            n.append(place-s)
        if c>0:
            n.append(place-1)
        if r+1<s:
            n.append(place+s)
        return n

    def moves(self, level: int) -> list[int]:
        return self.neighbors(self.trees[level])

    def valid(self, level: int, move:int) -> bool:
        if move in self.trees:  # da steht ein anderer Baum
            return False

        if move in self.tents:  # da steht schon ein zelt
            return False

        for n in self.neighbors(move):
            if n in self.tents: # da steht schon ein Zelt daneben
                return False

        r, c = self.rowAndCol(move)
        if self.rowscount[r] >= self.rows[r]:   # kein weiteres Zelt möglich
            return False
        if self.colscount[c] >= self.cols[c]:   # kein weiteres Zelt möglich
            return False
        return True

    def record(self, level: int, move:int):
        self.tents.append(move)
        r, c = self.rowAndCol(move)
        self.rowscount[r] += 1
        self.colscount[c] += 1

    def undo(self, level: int, move:int):
        self.tents.pop()
        r, c = self.rowAndCol(move)
        self.rowscount[r] -= 1
        self.colscount[c] -= 1

    def done(self, level: int)-> bool:
        if len(self.trees) == len(self.tents):
            self.solutionNr += 1
            self.show()
            return True
        return False

    def show(self):

        # Einfache Ausgabe:
        # print(f"\nSolution {self.solutionNr:4} level={level} {self.tents}")

        # Schönere Ausgabe:
        tree = "B"  #  B steht für Baum
        tent = "Z"  #  Z steht für Zelt
        print(f"\nLösung {self.solutionNr:4}")
        print("  "*6, end="")
        for c in range(self.csize):
            print(f"{self.cols[c]:3}", end="")
        print()
        for r in range(self.rsize):
            print(f"{self.rows[r]:2d} -> ", end="")
            for c in range(self.csize):
                f = r*self.rsize + c # Feldnummer
                if f in self.trees:
                    print(f"  {tree}", end="")
                elif f in self.tents:
                    print(f"  {tent}", end="")
                else:
                    print("  .", end="")
            print()

if __name__ == "__main__":
    Zeltlager(test=False, showAttempts=True)
```

Digitaltechnik „begreifbar machen“

Helmut Bittermann

Die Digitaltechnik befasst sich mit digitalen Schaltungen. Diese Schaltungen bestimmen den Aufbau und die Funktionsweise nahezu aller heute verwendeten informationsverarbeitenden Systeme. Die zunehmende Nutzung dieser Systeme wird allgemein als Digitalisierung bezeichnet. Die Digitaltechnik bildet somit die Grundlage für das, was unser Leben und unsere Welt in den letzten Jahrzehnten tiefgreifend verändert hat und zukünftig wohl noch mehr bestimmen wird.

Eine zeitgemäße und zukunftsorientierte schulische Ausbildung sollte dieser Entwicklung Rechnung tragen, auch im allgemeinbildenden Bereich. Das sollte aber nicht nur über einen medienpädagogischen Ansatz erfolgen, bei dem digitale Systeme mehr oder weniger „als Blackbox“ betrachtet werden und es vor allem um deren kompetente und kritische Nutzung geht. Um das „Wesen der Digitalisierung“ wirklich begreifen zu können, ist auch eine Auseinandersetzung mit grundlegenden Konzepten und Inhalten der Informatik und Digitaltechnik erforderlich. Dabei geht es beispielsweise um Fragen wie *„Was bedeutet eigentlich digital?“*, *„Wie können digitale Systeme rechnen?“* oder *„Wie können digitale Systeme Informationen speichern?“.* Eine Allgemeinbildung einer Informationsgesellschaft im Informationszeitalter sollte das Wissen vermitteln, um diese Fragen beantworten zu können.

In diesem Beitrag wird ein methodisches und didaktisches Konzept zur Vermittlung von Digitaltechnik kursorisch vorgestellt. Es wird vom Autor schon mehrere Jahre lang an einer AHS mit Informatik-Schwerpunkt in der 10. und 11. Schulstufe eingesetzt. Das Konzept verfolgt vor allem das Ziel, den Unterricht möglichst anschaulich und aktiv zu gestalten. Einfache digitale Schaltungen werden beispielsweise auf Steckbrettern in „Maker-Manier“ aufgebaut und sind damit „angreifbar und besser begreifbar“. Für die Erstellung komplizierterer Schaltungen werden einige sehr einfach und intuitiv zu bedienende grafische Simulationstools wie Logicly und Tinkercad-Circuits eingesetzt. Es werden auch FPGA-Boards genutzt, um Schaltungen auf Microchip-Ebene zu implementieren.

Das Herzstück vieler digitaler Systeme stellen Mikroprozessoren dar. Diese gehören wohl zu den kompliziertesten Objekten, welche die Menschheit je geschaffen hat. Moderne Prozessoren können mehrere Milliarden Transistoren enthalten, welche über kilometerlange Leiterbahnen zu digitalen Schaltungen miteinander verbunden sind. Diese befinden sich in einem Silizium-Plättchen, das nicht viel größer ist, wie ein Daumennagel. Dies lässt vermuten, dass nur mit einem entsprechenden Expertenwissen der Aufbau und die Funktionsweise dieser Systeme verstanden werden kann. Bei genauerer Betrachtung stellt sich aber heraus, dass dies in Grundzügen durchaus auch ohne spezielles Knowhow möglich ist. Der Schlüssel dazu ist das Prinzip der Abstraktion.

Laut Wikipedia bezeichnet Abstraktion „den induktiven Denkprozess des erforderlichen Weglassens von Einzelheiten und des Überführens auf etwas Allgemeineres oder Einfacheres.“ Diese besondere Fähigkeit des menschlichen Denkens sorgt einerseits dafür, dass wir uns in der Komplexität der uns umgebenden Welt zurechtfinden können, indem wir nur das jeweils Wesentliche beachten. Andererseits verschafft sie uns auch die Möglichkeit, komplizierte Systeme in strukturierter Art und Weise zu erzeugen. Abstraktionen werden dabei Schicht für Schicht aufgebaut, was zu immer höheren Ebenen und Fähigkeiten führt. In der Digitaltechnik wird das besonders augenscheinlich: Funktioniert eine digitale Schaltung wie erwartet, kann sie als neue funktionale Einheit – als integrierter Schaltkreis (IC) – für die Entwicklung komplizierterer Schaltungen verwendet werden. Ihr konkreter interner Aufbau ist bei dieser Verwendung nicht mehr von Bedeutung. Die Abstraktion ermöglicht dabei eine zunehmende Kompliziertheit der Schaltungen und sorgt dafür, dass diese auch beherrschbar bleibt.

Dieses Prinzip lässt sich auch in didaktischer Hinsicht sehr gut nutzen, um schrittweise ein Verständnis über den Aufbau und die Funktionsweise digitaler Schaltungen zu vermitteln (vom Einfachen zum Komplizierten). Die Digitaltechnik als Wissenschaft gibt dabei den Weg theoretisch vor. Wenn dieser Weg mit entsprechenden Praxisbeispielen anschaulich und aktiv gestaltet wird, sind die Lernenden dabei auch entsprechend motiviert und erleben dann das Entdecken und Verstehen dieser Zusammenhänge oftmals sogar als persönlichen Erfolg.

Das vorgestellte Unterrichtskonzept eignet sich nach Meinung des Autors sehr gut für den Wahlpflichtgegenstand Informatik in der AHS-Oberstufe. Teile davon lassen sich auch in den Unterricht im Informatik-Pflichtgegenstand integrieren. Die praktische Umsetzbarkeit der vorgestellten digitalen Schaltungen ist unterschiedlich, was Schwierigkeit und Zeitaufwand betrifft. Bei einer entsprechenden Ausstattung und Vorbereitung können alle Schaltungen von den Schülerinnen und Schülern selbst aufgebaut werden. Je nach Schwerpunktsetzung und verfügbarer Unterrichtszeit kann es auch sinnvoll sein, bereits vorgefertigte Schaltungen zum Testen und Experimentieren bereit zu stellen oder deren Schaltverhalten nur zu demonstrieren und zu erörtern.

Im Folgenden wird beschrieben, wie Schaltungen mit Feldeffekttransistoren, wie Logikgatter, Arithmetik-Schaltungen, eine ALU, einfache Speicherbausteine (Latches, Flipflops und Register) und eine Zählerschaltung unter Anwendung der erwähnten didaktischen und methodischen Hilfsmittel Schritt für Schritt und aufeinander aufbauend erstellt werden können. Damit sollen grundlegende Einsichten und Kenntnisse über den Aufbau und die Funktionsweise von Computersystemen im Allgemeinen und von Mikroprozessoren im Speziellen vermittelt werden.

Ein Verständnis der binären Digitaltechnik setzt ein Verständnis der binären Informationsdarstellung voraus. Das gilt es aus didaktischer Sicht zu berücksichtigen. Theoretische Kenntnisse über das binäre Zahlensystem, die Zweierkomplement-Darstellung, das Rechnen mit binären Zahlen und Grundlagen der Booleschen Algebra sollten entweder vorweg oder begleitend vermittelt werden.

Die Basis: Feldeffekttransistoren

Die kleinsten funktionalen Einheiten von Mikroprozessoren sind Feldeffekttransistoren (FETs). Moderne CPUs enthalten mehrere Milliarden davon. Sie sind miteinander elektrisch leitend verbunden und fungieren als winzige Ein- und Ausschalter. Besonders bemerkenswert sind ihre Kleinheit und ihr Schaltverhalten. Ihre Abmessungen bewegen sich in der Größenordnung von 100 Nanometern und sie können mehrere Milliarden mal pro Sekunde aus- und eingeschaltet werden.

Der **MOSFET BS170** (vgl. https://onsemi.com/pdf/datasheet/mmbf170-d.pdf) ist ein solcher Feldeffekttransistor,

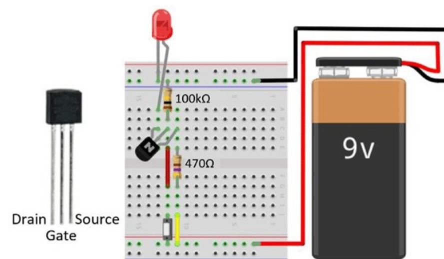


Abb.1: MOSFET BS170 durchschalten (Grafik erstellt mit Fritzing).

genauer ein Metall-Oxid-Halbleiter-Feldeffekt-Transistor (Metal-Oxide-Semiconductor Field-Effect-Transistor). Seine Abmessungen sind verglichen mit den FETs in einer CPU „gigantisch“. Dadurch kann er aber „angefasst werden“ und eignet sich zum Aufbau von Schaltungen auf Steckbrettern (vgl. **Abb.1**). Er wird über eine elektrische Spannung und ein dadurch erzeugtes elektrisches Feld am Gate-Anschluss gesteuert. Mit diesem kann der Stromfluss zwischen Source und Drain aus- und eingeschaltet werden. Dies lässt sich mit der in **Abb.1** dargestellten Schaltung genauer untersuchen.

Wird der 9V-Block angeschlossen und der Taster betätigt, leuchtet die LED auf. Sie erlischt sofort, wenn der Taster losgelassen wird. Das Drücken des Tasters bewirkt, dass am mittleren Anschluss des Transistors, dem Gate, über die rote Steckbrücke hinweg eine elektrische Spannung angelegt und dadurch der Transistor durchgeschalten wird. Es kann nun ein elektrischer Strom zwischen dem Plus-Pol und dem Minus-Pol der Batterie fließen. Der Stromfluss erfolgt dabei über die gelbe Steckbrücke, den 470Ω-Widerstand, durch den Transistor (zwischen Source und Drain) und über die LED, welche dadurch aufleuchtet. Der Widerstand von 100kΩ am Gate-Anschluss ist notwendig, damit von dort nach dem Ausschalten überschüssige Ladung sofort abfließen kann.

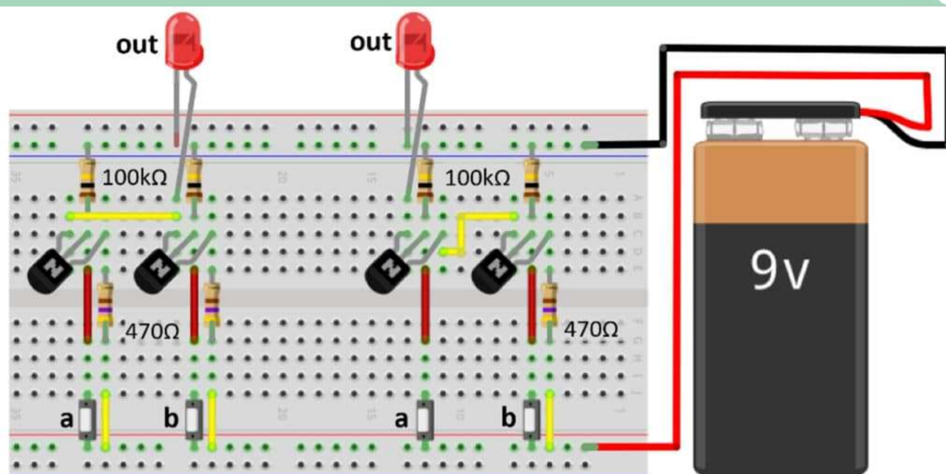
Das Schaltverhalten der Feldeffekttransistoren in Mikroprozessoren wird also durch den Spannungspegel am Gate-Anschluss bestimmt. Es werden dabei nur zwei Zustände unterschieden: Hoher Spannungspegel (High, An, 1) und niedriger Spannungspegel (Low, Aus, 0). Die damit aufgebauten elektronischen Schaltungen werden daher als binäre Schaltungen bezeichnet.

Allgemein werden in der Elektronik digitale und analoge Signale unterschieden. Charakteristisch für analoge Signale ist ihr kontinuierlicher Wertebereich. Im Gegensatz dazu besitzen digitale Signale nur eine endliche Zahl diskreter Werte. Die Digitaltechnik befasst sich mit solchen Signalen. Sind es – so wie bei den Feldeffekttransistoren in den Mikroprozessoren – nur zwei diskrete Werte, spricht man auch von der binären Digitaltechnik.

Die nächste Ebene: Logik-Gatter

Die Feldeffekttransistoren in Mikroprozessoren sind miteinander elektrisch leitend verbunden, um elektronische Schaltungen zu bilden. Auf der nächsten Abstraktionsebene sind dies zunächst Logik-Gatter wie z.B. AND, OR, XOR, NAND, NOR, XNOR und NOT.

Die beiden **Logik-Gatter OR** und **AND** lassen sich beispielsweise mit jeweils zwei MOSFETs auf einem Steckbrett, wie in



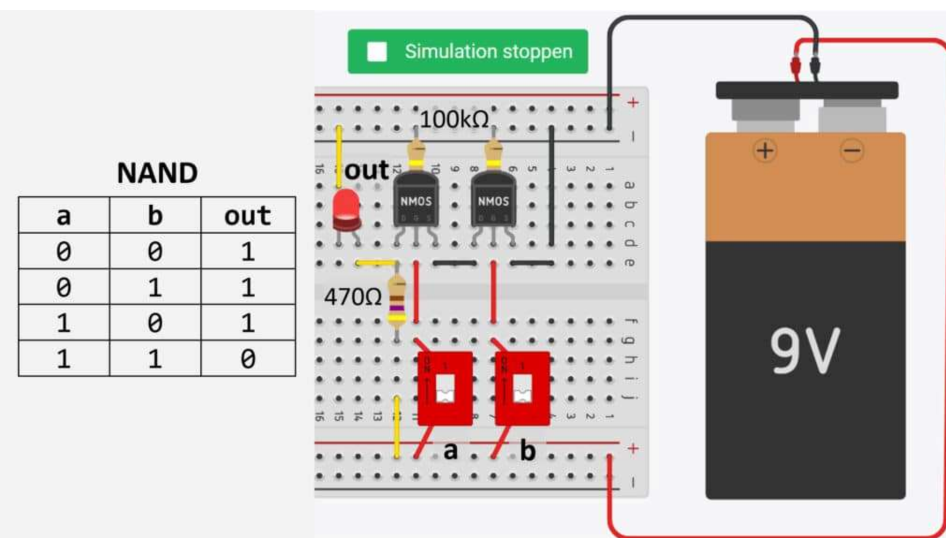
Parallelschaltung, OR

a	b	out
0	0	0
0	1	1
1	0	1
1	1	1

Reihenschaltung, AND

a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

Abb.2: OR-Gatter und AND-Gatter mit MOSFETs (Grafik erstellt mit Fritzing).



NAND

a	b	out
0	0	1
0	1	1
1	0	1
1	1	0

Abb.3: Simulation eines NAND-Gatters mit MOSFETs, erstellt mit dem Online-Tool Tinkercad-Circuits.

Abb.2 gezeigt, als einfache Parallel- und Reihenschaltung aufbauen.

Das Online-Tool Tinkercad-Circuits (https://www.tinkercad.com/circuits) ermöglicht die virtuelle Erstellung von Schaltungen in einem Browser. Es wird eine Vielzahl elektronischer Bauteile zur Verfügung gestellt, die auf virtuellen Steckbrettern sehr einfach platziert und miteinander verbunden werden können. Im Simulationsmodus kann dann das Verhalten der Schaltung beobachtet und getestet werden.

Abb.3 zeigt ein mit Tinkercad-Circuits aus zwei FETs erstelltes NAND-Gatter. Die schaltungstechnische Realisierung erfolgt dabei mit zwei in Reihe geschalteten FETs (NMOS- Bausteinen), die out auf Masse (logisch 0, schwarze Leiterbahnen) legen, wenn beide durchgeschalten sind und der Strom aufgrund des geringeren Widerstands durch sie hindurchfließt und nicht durch die LED. Ist ein FET oder sind beide FETs nicht durchgeschalten, so ist die Masseverbindung unterbrochen. Der Strom fließt dann durch die LED und bringt diese zum Aufleuchten (logisch 1, gelbe Leiterbahn). Vor der LED befindet sich ein 470Ω-Widerstand, hinter den FETs jeweils ein 100kΩ-Widerstand. Die in **Abb.3** enthaltene Wahrheitstabelle kann mit der Simulationsfunktion von Tinkercad -Circuits ermittelt werden.

NAND als Universalgatter

Die beiden Logik-Gatter NAND und NOR zeichnen sich dadurch aus, dass mit ihnen jeweils alle anderen Logikgatter (AND, OR, XOR, XNOR, NOT und NOR bzw. NAND) aufgebaut werden können. Vor allem NAND-Gatter gelten daher in der Digitaltechnik als Standardbausteine.

So enthält z.B. der CD4011BE (vgl. https://www.ti.com/lit/ds/symlink/cd4011b.pdf) als integrierter Schaltkreis vier NAND-Gatter. Er gehört zur CMOS-Familie 4000

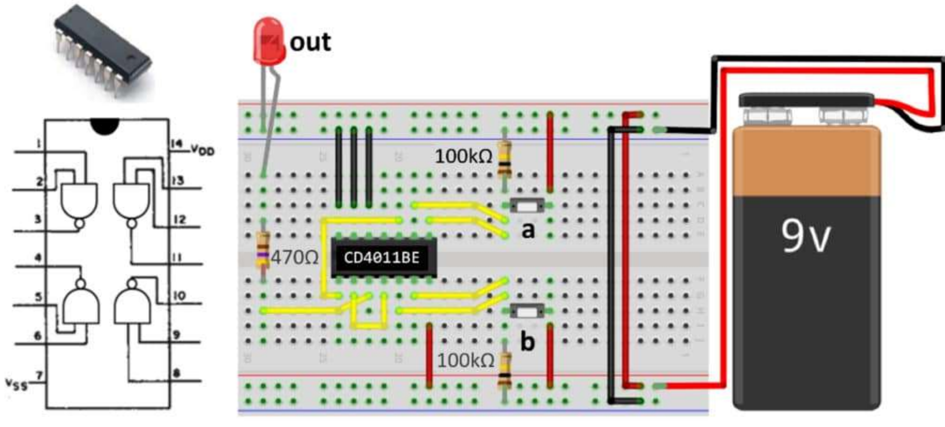


Abb.4: OR-Gatter mit drei NAND-Gattern des CD4011BE-ICs (Grafik erstellt mit Fritzing).

(CMOS: Complementary Metal-Oxide-Semiconductor). ICs dieser Familie können mit Betriebsspannungen zwischen 3 und 18V betrieben werden. Damit eignen sie sich sehr gut für einfache Experimente mit einer 9V-Batterie als Spannungs- und Stromquelle. Der CD4011BE besitzt 14 Pins (vgl. **Abb.4**). Er benötigt zum Betrieb eine Versorgungsspannung, welche zwischen Pin 14 (Vdd, +Pol) und Pin 7 (Vss, -Pol) angelegt werden muss. Die übrigen 12 Anschlüsse gehören zu seinen vier NAND-Gattern. Nicht verwendete Eingänge müssen beim Betrieb entweder auf das Potential von Vss oder Vdd gelegt werden, da sie sonst zu einer Fehlfunktion führen. Offene Ausgänge sind hingegen erlaubt.

In der Schaltung in **Abb.4** wird der CD4011BE-Baustein genutzt, um mit drei seiner NAND-Gatter ein OR-Gatter aufzubauen. Dazu wird der folgende Ausdruck verwendet:

```
a OR b = (a NAND a) NAND (b NAND b)
```

Die beiden Taster in der Schaltung stellen die Eingänge a und b dar. Im nicht-gedrückten Zustand werden ihre vom Pluspol der Stromquelle abgewandten Kontakte über einen hochohmigen Widerstand von 100kΩ auf Masse (Minuspol) gezogen. Die LED stellt den Ausgang der Schaltung dar. Sie wird durch einen Vorwiderstand von 470Ω vor einer Überlastung geschützt.

Auf die besondere Eigenschaft des NAND-Gatters als Universalgatter beziehen sich zwei aus pädagogischer und didaktischer Sicht besonders bemerkenswerte Projekte:

Das eine Projekt mit der Bezeichnung **„From Nand toTetris“** (https://www.nand2tetris.org/) besteht aus zwei Teilen. Im ersten Teil werden unter Verwendung einer Hardwarebeschreibungssprache aus NAND-Gattern sukzessive alle notwendigen Schaltungen und Hardwarekomponenten für ein virtuelles Computersystem erstellt und zu einem solchen System kombiniert. Für dieses virtuelle System wird auch eine Maschinensprache definiert und ein Assembler entwickelt. Im zweiten Teil werden dazu ein Betriebssystem und ein Compiler samt virtueller Laufzeitumgebung für eine Java-ähnliche Programmiersprache erstellt. Damit werden dann Spiele wie Tetris und Pong programmiert, welche auf diesem selbst erstellten, nur auf NAND-Gattern beruhenden, virtuellen Computersystem gespielt werden können. Das virtuelle Computersystem wird mit speziell für das Projekt programmierten Java-Anwendungen realisiert. Es existieren dazu auf Coursera auch zwei frei zugängliche und didaktisch ausgezeichnet gestaltete MOOCs (https://www.coursera.org/learn/build-a-computer und https://www.coursera.org/learn/nand2tetris2).

Das zweite Projekt mit der Bezeichnung **„Nandgame“** bezieht sich auf das Erstgenannte: „The Nand Game is inspired by the amazing course From NAND to Tetris – Building a Modern Computer From First Principles which is highly recommended.“ (https://nandgame.com/) Es kann direkt in einem Browser „gespielt werden“. Es müssen dabei eine ganze Reihe von, wiederum nur auf NAND-Gattern basierende, Schaltungen erstellt werden, welche in mehrere Levels zusammengefasst sind. Der Level „Logic Gates“ ist der Ausgangspunkt. Über die Levels „Arithmetics“, „Switching“, „Arithmetic Logic Unit“ und „Memory“ gelangt man schließlich zum Level „Processor“, in dem abschließend ein schematisierter Computer erstellt werden muss. Jede zu erstellende Schaltung wird durch eine genaue Spezifikation beschrieben. **Abb.5** zeigt ein mit Nandgame unter Verwendung des folgenden Ausdrucks erstelltes **XOR-Gatter**:

```
a XOR b = (a NAND (a NAND b))
          NAND ((b NAND (a NAND b))
```

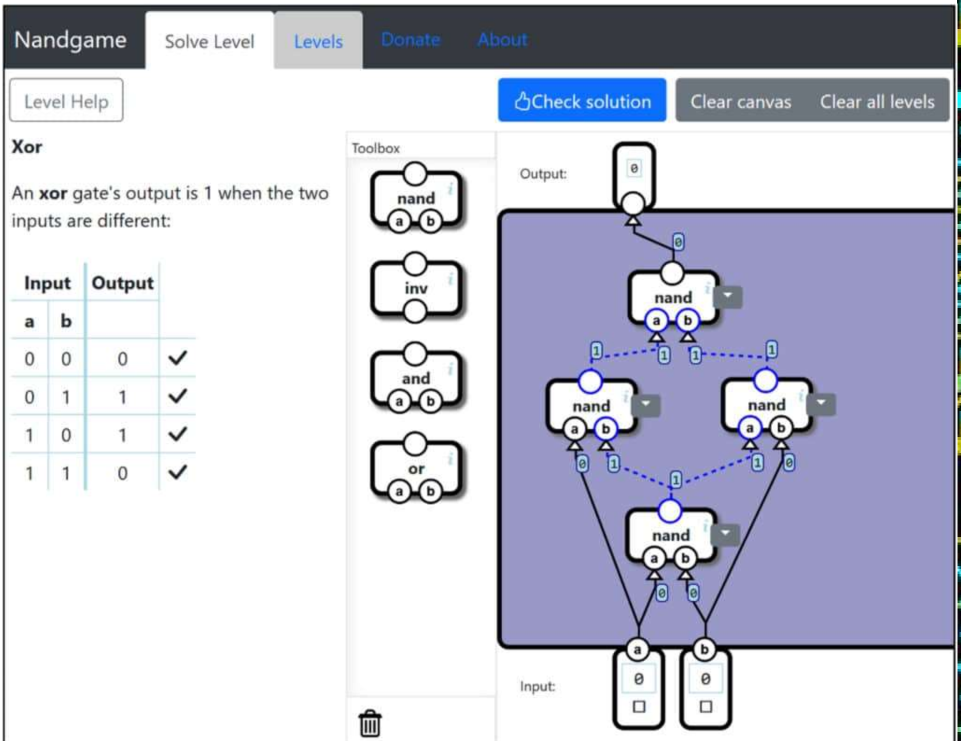
Die violette Fläche stellt das IC-Gehäuse dar. Die Anschlüsse außerhalb dieser Fläche (Input: a, b und Output) sind vorgegeben. Zum Aufbau einer Schaltung müssen aus dem Toolbox-Bereich die passenden Elemente in das IC-Gehäuse gezogen und miteinander verbunden werden. Über die Schaltfläche mit der Aufschrift „Check solution“ kann die richtige Funktionsfähigkeit der Schaltung überprüft werden.

Arithmetik-Schaltungen

Die Durchführung arithmetischer Operationen gehört zu den Kernaufgaben eines jeden Computersystems. Besonders bemerkenswert ist dabei, dass sich alle vier Grundrechnungsarten im Prinzip nur mit einer Addierer-Schaltung realisieren lassen: Eine Subtraktion kann als Addition mit einem negativen Summanden, eine Multiplikation als mehrfache Addition und eine Division als mehrfache Subtraktion ausgeführt werden.

Um Mehrbit-Werte zu addieren, werden Volladdierer-Schaltungen benötigt. Ein Volladdierer hat die Aufgabe, die Addition

Abb.5: XOR-Gatter aus NAND-Gattern, erstellt mit Nandgame.



zweier Eingangsbits (a und b) sowie eines Carry-Eingangsbits (ci), welches als Übertrag von einer Stelle weiter rechts anfallen kann, auszuführen. Am Ausgang werden ein Summationsbit (sum) sowie eine Carry -Ausgangssignal (co) bereitgestellt. Ein Volladdierer kann mit zwei XOR-Gattern, zwei AND- Gattern und einem OR-Gatter aufgebaut werden.

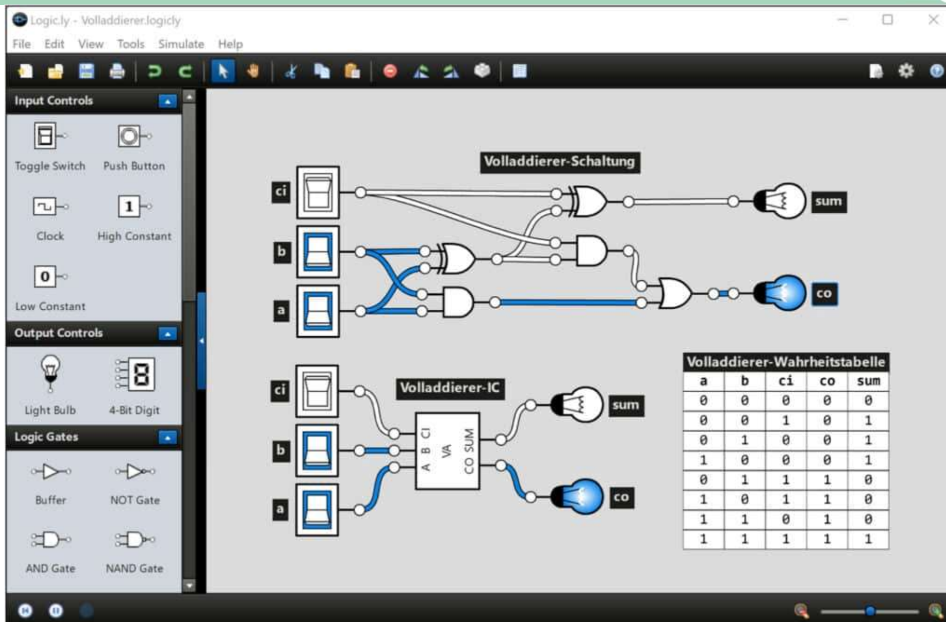
Die **Volladdierer-Schaltung** in **Abb.6** wurde mit Logicly (https://logic.ly/) erstellt. Dabei handelt es sich um eine sehr einfach und intuitiv zu bedienende Schaltungssimulationssoftware, welche als freie Online -Version zur Verfügung steht und auch als kostenpflichtige App angeboten wird.

Bei Logicly können als Eingangssignalgeber Toggle-Switches verwendet werden, welche sich ein- und ausschalten lassen. Ausgangssignale können mit Light-Bulbs dargestellt werden. Im linken Fensterbereich finden sich im Abschnitt „Logic Gates“ die üblichen Logikgatter im ANSI-Format. Diese können in den Workspace gezogen, dort passend angeordnet und mit anderen Schaltungskomponenten sehr einfach verbunden werden. Wie in **Abb.6** gezeigt, ist es mit Logicly auch möglich, eine erstellte Schaltung als integrierten Schaltkreis (IC) mit eigenem Gehäuse abzuspeichern. Die Anschlüsse können dabei selbst benannt und am IC-Gehäuse auch selbst positioniert werden. Mit einem Doppelklick auf einen solchen IC wird dessen interner Aufbau dargestellt. Sind darin weitere benutzerdefinierte ICs enthalten, so lässt sich auch deren Struktur mit weiteren Doppelklicks betrachten.

Abb.7 zeigt, wie diese **Volladdierer-Schaltung** mit Tinkercad-Circuits aufgebaut werden kann. Dabei kommen die drei ICs 74HC32 (mit vier OR-Gates), 74HC86 (mit vier XOR-Gates) und 74HC08 (mit vier AND-Gates) zum Einsatz, welche als eigene Bausteine vorhanden sind. Sie gehören zur 74HCxx- IC-Familie, deren Vertreter mit 2 bis 6 V Spannung betrieben werden können. HC steht dabei für „Highspeed CMOS“. In **Abb.7** wird auch die Pinbelegung des 74HC08 gezeigt, die beiden anderen ICs sind ebenso aufgebaut. In der Schaltung sind die nicht verwendeten Gatter der drei ICs jeweils auf Masse gezogen. Im Simulationsmodus von Tinkercad-Circuits kann mit dieser Schaltung die Wahrheitstabelle des Volladdierers verifiziert werden.

Mit Hilfe von Volladdieren ist man in der Lage, ein vollständiges Addierwerk aufzubauen. **Abb.8** zeigt einen 8-bit Addierer, erstellt mit Logicly.

Für jede Bitstelle wird dabei eine eigene Volladdierer-Komponente benötigt. Die jeweiligen Operanden-Bits von a und b werden jeweils an die a- und b-Eingänge der einzelnen Volladdierer geführt. Ein eventueller Carry-Eingang für den gesamten Addierer wird an den Carry-Eingang



a	b	ci	co	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
1	0	0	0	1
0	1	1	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Abb.6: Volladdierer-Schaltung und Volladdierer-IC, erstellt mit Logicly.

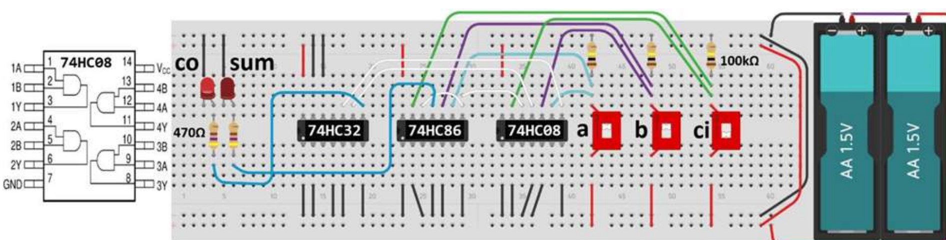


Abb.7: Volladdierer-Schaltung mit den drei ICs 74HC32, 74HC86 und 74HC08, erstellt mit Tinkercad-Circuits.

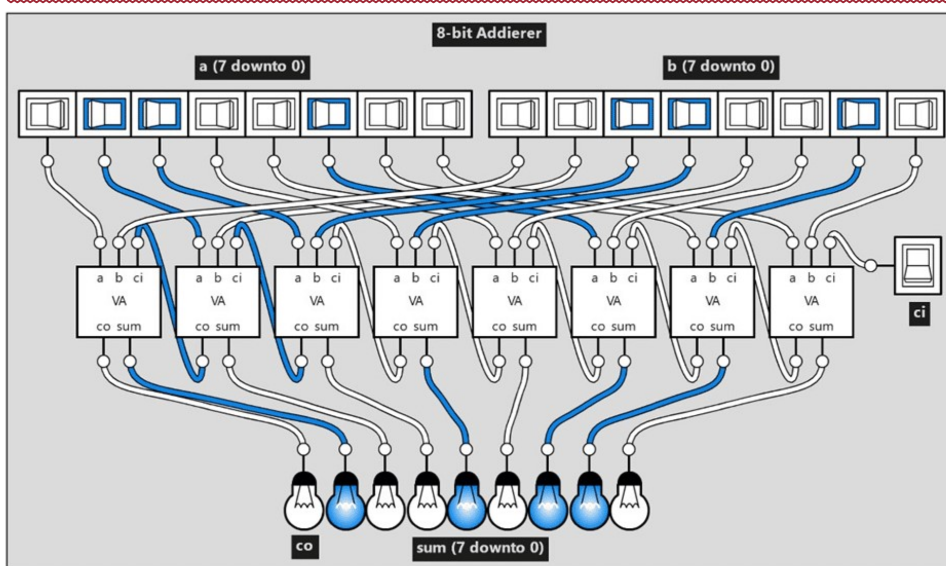


Abb.8: 8-bit Addierer, erstellt mit Logicly, es wird die Rechnung 100 + 50 = 150 dargestellt.

der niederwertigsten Stufe gelegt (ci), andernfalls wird dieser mit logisch 0 verbunden. Die Carry-Ausgänge der einzelnen Volladdierer werden in einer Kette mit den Carry-Eingängen der weiter links nachfolgenden Stufen verbunden. Der Carry-Ausgang (co) der letzten Volladdierer-Stufe kann als höchstwertiges Summationsbit interpretiert werden. Das Summationsergebnis ist damit zur Vermeidung von Überläufen um ein Bit breiter als die Operanden.

Abb.9 zeigt einen **8-bit Addierer** aufgebaut auf einem Steckbrett. Dabei werden zwei 4-bit Volladdierer MC14008B-ICs eingesetzt.

Die beiden Summanden a und b werden über DIP-Switches eingegeben, die errechnete Summe wird über einen LED-Bar-Baustein ausgegeben. Die Pin-Zuordnung des MC14008B ist in **Abb.9** links oben dargestellt. Die Versorgungsspannung muss zwischen VDD (+Pol) und VSS (-Pol) angelegt werden. Die Summanden werden über die Eingänge A1 – A4 sowie B1 –

B4 angeschlossen, die Summenbits werden an den Pins S1 – S4 ausgegeben. Cin ist der Übertrag-Bit-Eingang und Cout der Übertrag-Bit-Ausgang. Das Cin-Signal des rechten ICs ist auf Masse gezogen (logisch 0). Beim linken IC ist Cin mit Cout des rechten ICs verbunden. Cout des linken ICs ist als höchstwertiges Summationsbit mit dem LED-Bar-Baustein verbunden. Es wird die Addition 82 + 82 = 164 dargestellt.

Werden negative Zahlen im Binärsystem über das Zweierkomplement gebildet, kann die Differenz a – b direkt auf die Addition mit dem negativen Summanden (-b) zurückgeführt werden (a + (-b)). Somit lässt sich eine Addierer auch als Subtrahierer verwenden. Man erhält die Zweierkomplementdarstellung einer negativen Zahl, indem man alle Bitstellen ihrer binären Absolutbetragsdarstellung invertiert und abschließend eine 1 addiert.

Abb.10 zeigt einen mit Logicly erstellten **8-bit Addierer und Subtrahierer**.

Die Werte für a und b werden jeweils über 8 Toggle-Switches angegeben. Das add/sub-Signal bestimmt die Art der Rechenoperation (0 Addition, 1 Subtraktion). Im Falle einer Subtraktion (add/sub = 1) wird von b das Zweierkomplement gebildet. Dabei erfolgt die Inversion von b mit acht XOR-Gatter und die Addition von 1 wird durch die Einspeisung des add/sub-Signals in den ci-Eingang des Addierers bewerkstelligt. Das Ergebnis (Summe oder Differenz) wird über acht Light-Bulbs ausgegeben (sum/dif). Ein Ausgabewert von 1 bei over/neg bedeutet im Falle einer Addition einen Überlauf (Summe > 255) und bei einer Subtraktion einen negativen Ergebniswert (Differenz < 0).

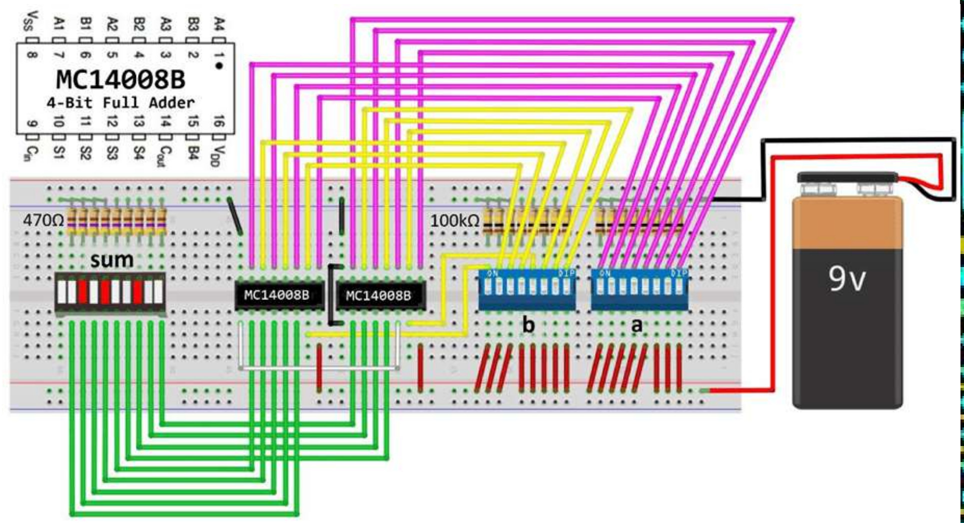


Abb. 9: 8-bit Addierer mit zwei MC14008B-ICs (Grafik erstellt mit Fritzing).

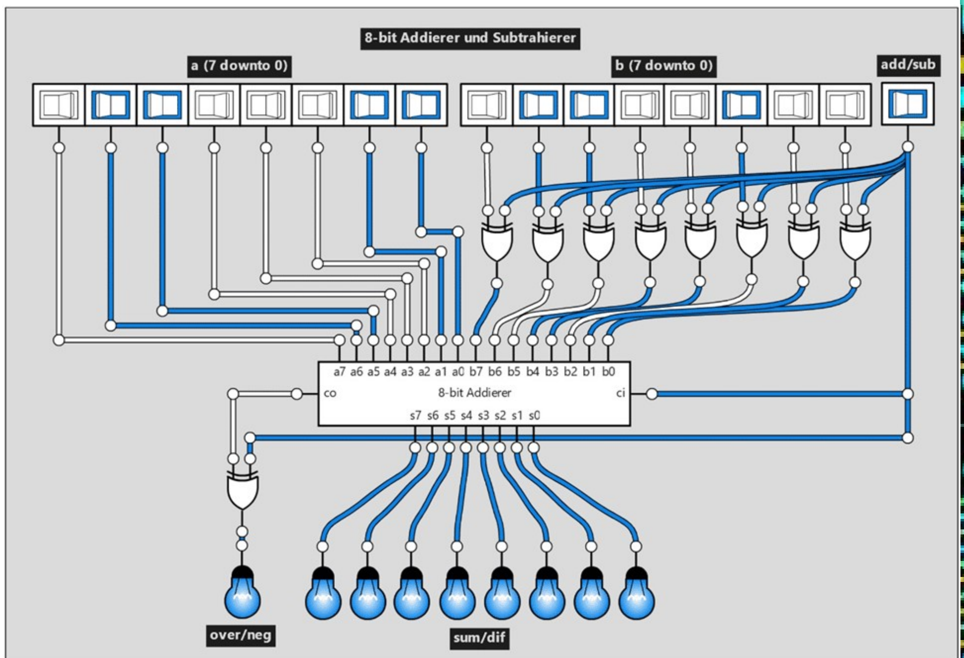


Abb.10: 8-bit Addierer und Subtrahierer, erstellt mit Logicly, es wird die Subtraktion 99 – 100 = -1 dargestellt.

Datenflusskontrolle

Bei digitalen Schaltungen ist es oft erforderlich, den Datenfluss gezielt zu steuern. Dies kann mit Multiplexern und Demultiplexern erfolgen.

Multiplexer kommen immer dann zum Einsatz, wenn eine Signalleitung mit einem Wert beschaltet werden soll, der aus mehreren Quellen stammen kann. Ein Multiplexer führt diese Datenpfade gezielt zusammen. Dabei kann über Steuerleitungen bestimmt werden, welche der verschiedenen Eingangsleitungen, auf die eine Ausgangsleitung durchgeschaltet wird. **Demultiplexer** bewirken im Gegensatz dazu eine kontrollierte Aufspaltung von Datenpfaden. Über Steuerleitungen kann festgelegt werden, mit welcher der verschiedenen Ausgangsleitungen, die eine Eingangsleitung verbunden wird.

Abb.11 zeigt mit Logicly erstellte **Multiplexer- und Demultiplexerschaltunge**n.

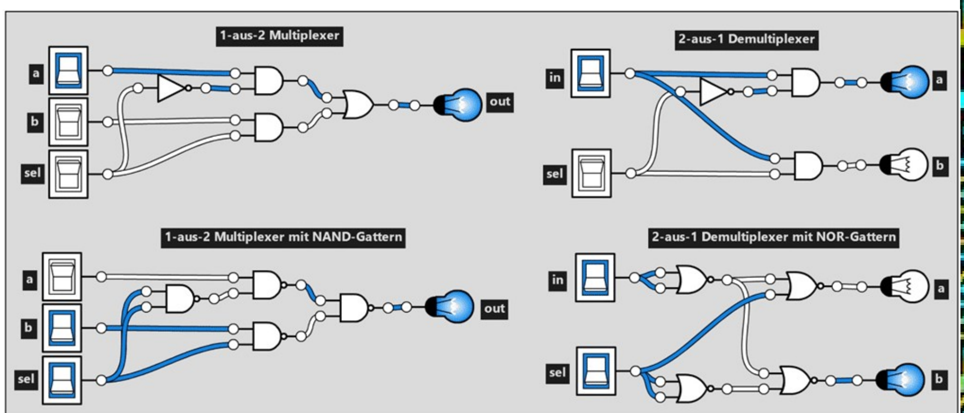


Abb.11: Multiplexer- und Demultiplexerschaltungen, erstellt mit Logicly.

Die beiden Schaltungen auf der linken Seite in **Abb.11** sind **1-aus-2 Multiplexer (2:1 MUX)**. Bei diesen kann mit dem sel-Signal bestimmt werden, ob der a-Eingang oder der b-Eingang auf den out-Ausgang durchgeschaltet wird. Ist das sel-Signal gleich 0 wird a mit out verbunden (links oben), ist sel gleich 1, wird b auf out gelegt (links unten). Beide Schaltungsvarianten sind funktionsgleich, bei der unteren Variante werden ausschließlich NAND-Gatter verwendet. **Abb.12** zeigt diese Schaltung auf einem Steckbrett (links). Dabei wird ein CD4011BE-IC mit vier NAND-Gattern verwendet.

Auf der rechten Seite in **Abb.11** befinden sich zwei **2-aus-1 Demultiplexerschaltungen (1:2 DEMUX).** Das sel-Signal bestimmt

dabei, ob das in-Signal auf den a-Ausgang oder den b-Ausgang durchgeschaltet wird. Ist das sel-Signal gleich 0, so wird das in-Signal auf den a-Ausgang gelegt (rechts oben), ist sel gleich 1, wird das in- Signal mit dem b-Ausgang verbunden (rechts unten). Wiederum sind beide Schaltungen funktionsgleich. Bei der unteren Variante werden nur NOR-Gatter eingesetzt. Wie sich diese Schaltung auf einem Steckbrett realisieren lässt, wir in **Abb.12** rechts gezeigt. Dabei wird ein CD4001BE-ICs mit vier NOR-Gattern verwendet.

Neben den hier vorgestellten einfachen Varianten werden in der Praxis auch häufig deutlich größere und kompliziertere Multiplexer- und Demultiplexer-Bausteine verwendet. Einerseits können die Ein- und Ausgangssignalleitungen (in, a, b, out) breiter sein (8-bit, 16-bit, 32-bit, …), andererseits sind mit mehreren Steuerleitungen (sel1, sel2, …) auch mehrere Ein- und Ausgangssingale möglich (n:1 MUX bzw. 1:n DEMUX).

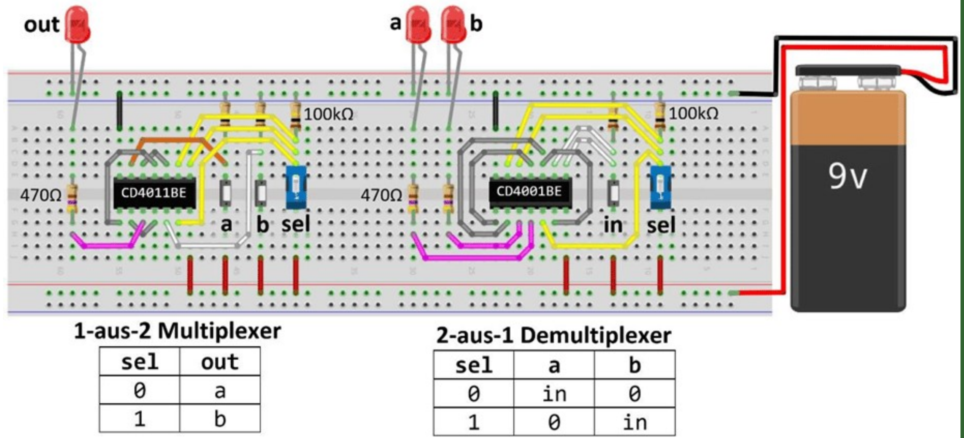
ALU

In den vorangegangenen Abschnitten wurden wichtige Grundkomponenten digitaler Schaltungen vorgestellt, die sich in modernen Mikroprozessoren auf die eine oder andere Weise wiederfinden. Einige dieser Komponenten sind als arithmetisch-logische Einheit (ALU – Arithmetic Logic Unit) zusammengefasst. Diese stellt unterschiedliche arithmetische und logische Funktionen bereit, welche über Steuerleitungen ausgewählt werden können.

Im Folgenden wird die 16-bit ALU des „From Nand To Tetris"-Projekts (https://www.nand2tetris.org/) vorgestellt. Diese ALU ist relativ einfach aufgebaut. Sie bewältigt aber alle notwendigen Berechnungen bei der Ausführung der im Rahmen dieses Projektes entwickelten Spiele Tetris und Pong einwandfrei. **Abb.13** gibt einen Überblick über ihre Anschlüsse und Funktionen.

Die ALU besitzt zwei 16-bit Eingänge (x und y), einen 16-bit Ausgang (o), sechs 1-bit-Steuereingänge (zx, nx, zy, ny, f und no) und zwei 1-bit Ausgänge zur Ausgabestatusanzeige (zr: 1 bei o=0, ng: 1 bei o<0). Die sechs 1-bit Steuereingänge (Control-Bits) bestimmen die von der ALU ausgeführte Operation. Über sie kann eine von 18 Funktionen ausgewählt werden. Diese Funktionen sind in **Abb.13** mit den dafür vorgesehenen Control-Bit-Werten aufgelistet. In den Computation-Instructions des 16-bit Maschinencodes des „From Nand To Tetris"-Systems sind für diese Steuereingänge sechs Bits als ALU-Opcode vorgesehen. Die Maschinenbefehle werden innerhalb der Control Unit der CPU ausgewertet und je nach ermitteltem Opcode wird die entsprechende Funktion der ALU aktiviert.

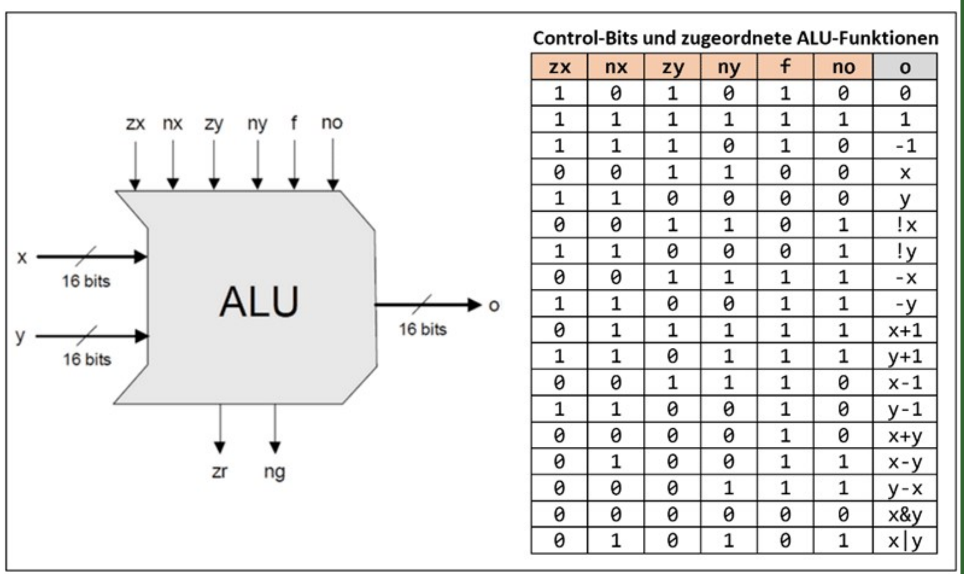
Abb.14 zeigt den internen Aufbau der „From Nand To Tetris"-ALU.



sel	out
0	a
1	b

sel	a	b
0	in	0
1	0	in

Abb.12: Multiplexerschaltung mit vier NAND-Gattern des CD4011BE-ICs und Demultiplexerschaltung mit vier NOR-Gattern des CD4001BE-ICs (Grafik erstellt mit Fritzing).



Control-Bits und zugeordnete ALU-Funktionen

zx	nx	zy	ny	f	no	o
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Abb.13: Die ALU des „From Nand ToTetris"-Projekts, ihre Ein- und Ausgänge sowie ihre 18 Funktionen (Operationen).

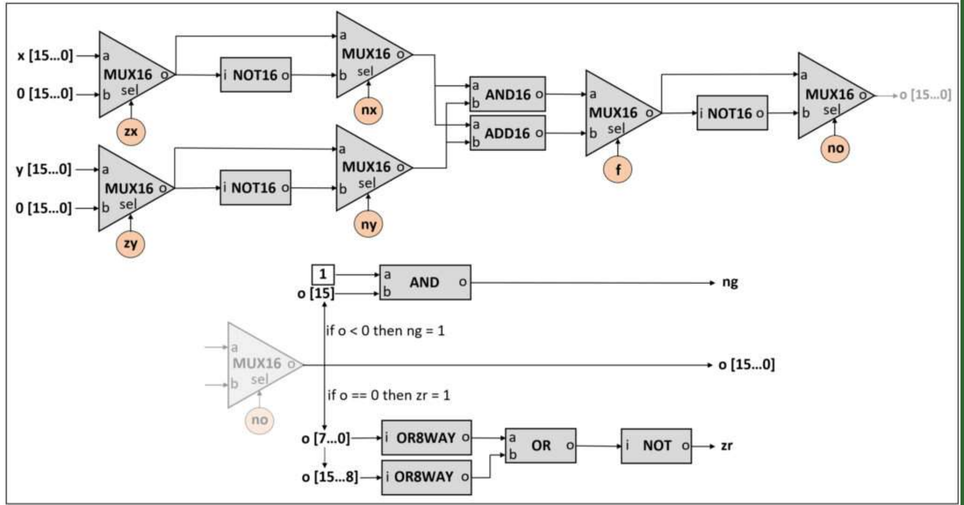


Abb.14: Der interne Aufbau der „From Nand To Tetris"-ALU.

Im oberen Teil von **Abb.14** ist der Datenpfad von den Eingangssignalen x und y zum Ausgangssignal o dargestellt, der untere Teil zeigt detailliert die schaltungstechnische Realisierung der Statusausgaben ng und zr. In der „From Nand To Tetris"-ALU sind die folgenden Komponenten enthalten:

- **6x MUX16 (16-bit 1-aus-2 Multiplexer)**

Diese Multiplexer verfügen jeweils über einen 1-bit Selektionssignaleingang (sel), mit dem eines der beiden 16-bit Eingangssignale (a oder b) ausgewählt und an den 16-bit Ausgang (o) durchgeschalten werden kann. Bei sel=0 wird a nach o durchgeschalten, bei sel=1 wird b ausgegeben. Jedes der sechs ALU-Steuersignale (zx, nx, zy, ny, f und no) dient als Selektionssignal bei einem die-

ser sechs Multiplexer.

- **3x NOT16 (16-bit NOT-Gatter)**
  Diese nehmen jeweils am Eingang i einen 16-bit Wert entgegen und geben diesen als invertierten 16-bit Wert am Ausgang o aus.
- **1x AND16 (16-bit AND-Gatter) und 1x ADD16 (16-bit Addierer)**
  AND16 ist ein Baustein, welcher die beiden 16-bit Eingänge a und b bitweise „UND-verknüpft" und das Ergebnis beim Ausgang o als 16-bit Wert ausgibt. ADD16 ist ein 16-bit Addierer mit den 16-bit Eingängen a und b für die beiden Summanden und dem 16-bit Ausgang o für die errechnete Summe.
- **2x OR8WAY, 1x OR, 1x NOT und 1x AND**
  Diese Bausteine dienen zur Ermittlung der Statusausgaben (zr und ng) der ALU. zr ist 1, wenn der von der ALU bei o ausgegebene Wert 0 ist, ansonsten 0. ng ist 1, wenn der von der ALU bei o ausgegebene Wert negativ ist, ansonsten 0.

Abb.15 zeigt die „From Nand To Tetris"-ALU als IC, erstellt mit Logicly. Bei den Steuersignalen ist nur das f-Control-Bit aktiviert. Es erfolgt daher eine Addition (vgl. **Abb.13**).

Die Erstellung des 16-bit ALU-ICs in **Abb.15** ist relativ aufwändig. Die Logicly-App unterstützt aber eine arbeitsteilige Vorgangsweise. Schaltungskomponenten können erstellt und als IC-Bibliotheksdateien exportiert und importiert werden. Schaltungsteile können auch mittels Copy & Paste zwischen App-Instanzen ausgetauscht werden.

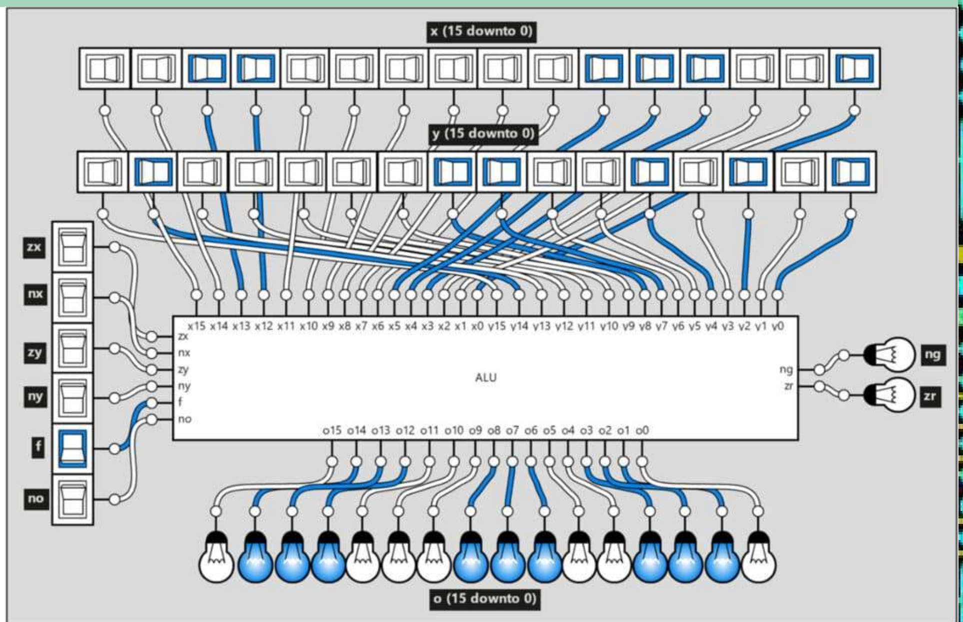


Abb.15: Die „From Nand To Tetris"-ALU als IC, erstellt mit Logicly, es wird die Addition 12345 + 16789 = 29134 dargestellt.

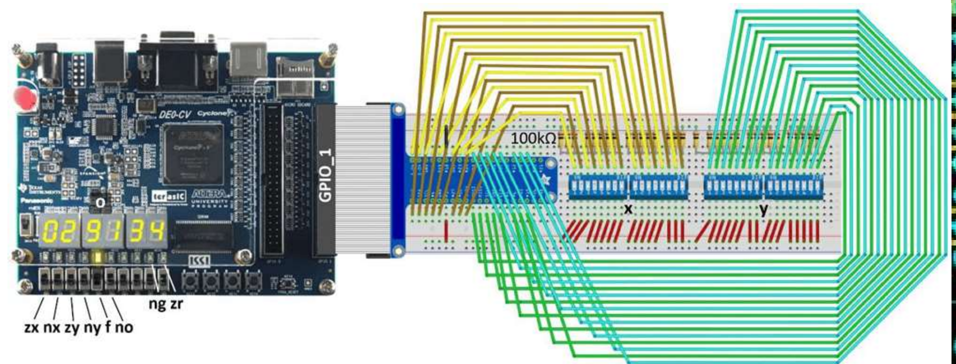


Abb.16: Aufbau zur Implementierung der „From Nand To Tetris"-ALU auf dem Terasic DE0-CV FPGA-Board (Grafik erstellt mit Fritzing).

ALU in FPGA-Implementierung

Eine Schaltung wie die ALU des „From-Nand-To-Tetris"-Projekts lässt sich auf Steckbrettern kaum mehr sinnvoll aufbauen. Eine Möglichkeit der physischen Umsetzung eröffnen in diesem Zusammenhang FPGA-Boards. FPGA steht für „Field Programmable Gate Array", was so viel wie „im Feld (vor Ort durch den Benutzer) programmierbare Logikgatter-Anordnung" bedeutet.

Ein FPGA-Chip enthält – sehr vereinfacht dargestellt – Logikblöcke, die nicht wie in herkömmlichen Mikrochips fest verdrahtet sind, sondern durch das Implementieren einer Konfigurationsdatei miteinander verbunden werden. In diesen Chips lassen sich also eigene digitale Schaltungen aufbauen und sie können immer wieder neu konfiguriert werden, auch mit unterschiedlichen Schaltungen.

Die Erstellung von FPGA-Konfigurationsdateien erfolgt mit speziellen Konfigurationssprachen. VHDL (Very High Speed Integrated Circuit Hardware Description Language) ist eine solche Sprache. VHDL-Dateien können in einem Texteditor geschrieben und müssen dann von einer Synthese-Software in eine Bitstream-Datei übersetzt werden. Diese kann dann in den FPGA-Chip geladen und damit die gewünschte Schaltung realisiert werden.

Im Folgenden wird die Implementierung der „From-Nand-To-Tetris"-ALU auf einem **Terasic DE0-CV FPGA- Board** (https://www.terasic.com.tw/) kursorisch beschrieben. Als Synthese-Software wird die kostenlos erhältliche Intel Quartus Prime Lite Edition verwendet (https://fpgasoftware.intel.com/?edition=lite). Darin ist auch Intel ModelSim, Starter Edition enthalten, eine Simulations- und Testumgebung für mit Hardwarebeschreibungssprachen wie VHDL erstellte Schaltungen.

Das Terasic DE0-CV Board ist mit einem Intel (vormals Altera) FPGA Chip Cyclone V Typ 5CEBA4F23C7 ausgestattet. Dieser verfügt u.a. über 49K programmierbare Logikelemente. Auf dem Board befinden sich viele Peripheriekomponenten, welche für eigene Schaltungen genutzt werden können.

Abb.16 zeigt den verwendeten Aufbau zur Implementierung der „From Nand To Tetris"-ALU.

Von den Board-Komponenten werden die sechs 7-Segment-Anzeigen zur Ergebnisausgabe im dezimalen Format (o), die darunter befindlichen Switches zur Eingabe der Control-Bits (zx, nx, zy, ny, f, no), die darüber befindlichen LEDs zur Anzeige der gesetzten Control-Bits und der beiden Ausgabestatusbits (ng, zr) sowie eine der beiden GPIO-Schnittstellen (GPIO_1) zur Eingabe der beiden Operanden (x, y) genutzt. Die Eingabe der Operanden erfolgt dabei über jeweils zwei 8-polige DIP-Schaltermodule auf einem Steckbrett. Die Verbindung zur GPIO-Schnittstelle des Boards kann über ein Flachbandkabel, ein 40-Pin GPIO-Breakoutboard und flexible Steckbrücken hergestellt werden. Die GPIO-Schnittstelle des Boards liefert 5V auf Pin 11, 3.3V auf Pin 29 sowie GND auf Pin 12 und Pin 30. Masseseitig befinden sich bei den DIP-Schaltern 100kΩ-Widerstände.

Mit VHDL können Schaltungen auf verschiedenen Abstraktionsebenen entworfen werden, von einer funktionalen Verhaltensbeschreibung auf höchster Ebene bis zu einer rein strukturellen Beschreibung auf Gatterebene. Für die Implementierung der „From Nand To Tetris"-ALU ist ein Strukturmodell adäquat. Der in **Abb.14**

wiedergegebenen Aufbau der ALU soll damit möglichst genau abgebildet werden.

In einem VHDL-Strukturmodell wird eine Schaltung als Entität definiert, deren Struktur durch die Instanziierung der in der Schaltung enthaltenen Komponenten und deren Verbindungen beschrieben wird. Jede verwendete Komponente muss mit ihren Schnittstellen vorweg angeführt werden und mit einem eigenen VHDL-Modell vorliegen. Zur Verbindung der einzelnen Komponenten werden Signale definiert.

Abb.17 und Abb.18 zeigen die VHDL-Beschreibung der „From Nand To Tetris"-ALU.

Der VHDL-Code gliedert sich in mehrere Abschnitte:

- **Zeile 1** in Abb.17 ist eine Kommentarzeile, Kommentare beginnen in VHDL mit der Zeichenkette „–„. Wie alle Code-Dateien, sollen auch VHDL-Dateien ausführlich mit Kommentaren versehen werden. Um den Inhalt zu verkürzen, wurden diese hier entfernt. **Zeile 2** ist eine Leerzeile. Whitespaces sowie Groß- und Kleinschreibung haben bei VHDL keine syntaktische Bedeutung.
- Mit **Zeile 3** in Abb.17 wird die Bibliothek „ieee" eingebunden und in **Zeile 4** darauf Bezug genommen. In der IEEE-Bibliothek ist u.a. der Datentyp „std_logic" definiert, der zur Beschreibung von Anschlüssen (Ports) und Signalen in der Regel verwendet wird. Er stellt eine Erweiterung des Datentyps „bit" dar und kann neben den beiden Werten „0" und „1" noch weitere Werte wie z.B. „U" für „Uninitialized" annehmen.
- Von **Zeile 6** bis **20** in Abb.17 wird die Schaltung als Entität „alu" definiert. Darin erfolgt mit dem port- Statement die Beschreibung aller ihrer Ein- und Ausgänge. Diese „Kommunikationskanäle" oder Ports werden jeweils mit Bezeichner, Signalflussrichtung und Datentyp festgelegt. Als Datentyp wird in der Regel „std_logic" verwendet. Mit „std_logic_vector" werden Ports definiert, die mehr als 1 Bit breit sind. Der Ausdruck „15 downto 0" gibt an, dass 16 Bits verwendet werden und das Bit mit dem Index 15 das Höchstwertige ist. Diese Zeilen sind im Grunde eine textuelle Beschreibung aller Ein- und Ausgänge der ALU, genauso wie in Abb.13 grafisch dargestellt.
- Mit dem architecture-Statement in **Zeile 22** in Abb.17 wird die strukturelle Beschreibung der ALU-Schaltung eingeleitet. Vorweg werden darin zunächst in den **Zeilen 23** bis **76** alle verwendeten Schaltungskomponenten mit ihren Schnittstellen als Prototypen angeführt. Im Detail sind dies ein 16-bit 1-aus-2 Multiplexer (mux2way16), ein 16-bit

```
1  -- alu.vhd
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity alu is
7  port (
8      x  : in  std_logic_vector(15 downto 0);
9      y  : in  std_logic_vector(15 downto 0);
10     zx : in  std_logic;
11     zy : in  std_logic;
12     nx : in  std_logic;
13     ny : in  std_logic;
14     f  : in  std_logic;
15     no : in  std_logic;
16     o  : out std_logic_vector(15 downto 0);
17     ng : out std_logic;
18     zr : out std_logic
19 );
20 end alu;
21
22 architecture structure of alu is
23     component mux2way16
24     port (
25         a   : in  std_logic_vector(15 downto 0);
26         b   : in  std_logic_vector(15 downto 0);
27         sel : in  std_logic;
28         o   : out std_logic_vector(15 downto 0)
29     );
30     end component;
31     component not16
32     port (
33         i : in  std_logic_vector(15 downto 0);
34         o : out std_logic_vector(15 downto 0)
35     );
36     end component;
37     component and16
38     port (
39         a : in  std_logic_vector(15 downto 0);
40         b : in  std_logic_vector(15 downto 0);
41         o : out std_logic_vector(15 downto 0)
42     );
43     end component;
44     component add16
45     port (
46         a : in  std_logic_vector(15 downto 0);
47         b : in  std_logic_vector(15 downto 0);
48         o : out std_logic_vector(15 downto 0)
49     );
50     end component;
51     component and_gate
52     port (
53         a : in  std_logic;
54         b : in  std_logic;
55         o : out std_logic
56     );
57     end component;
58     component or8way
59     port (
60         i : in std_logic_vector(7 downto 0);
61         o : out std_logic
62     );
63     end component;
64     component or_gate
65     port (
66         a : in  std_logic;
67         b : in  std_logic;
68         o : out std_logic
69     );
70     end component;
71     component not_gate
72     port (
73         i : in  std_logic;
74         o : out std_logic
75     );
76     end component;
77
```

Abb.17: VHDL-Beschreibung der „From Nand To Tetris"-ALU, Teil 1.

NOT-Gatter (not16), ein 16-bit AND-Gatter (and16), ein 16-bit Addierer (add16), ein AND-Gatter (and_gate), ein 8-Bit Mehrweg-OR-Gatter (or8way), ein OR-Gatter (or_gate) und ein NOT-Gatter (not_gate). Wie aus **Abb.14** hervorgeht, sind alle diese Bausteintypen für den Aufbau der ALU erforderlich. Ihre Schnittstellen werden wie bei der Entity-Definition mit dem port-Statement beschrieben. Für alle diese Bausteintypen muss jeweils auch ein eigenes VHDL-Modell in einer eigenen VHDL-Datei vorhanden sein, damit die Simulations- und Syntheseprogramme (ModelSim und Quartus) deren interne Strukturen auflösen können.

- In den **Zeilen 78** bis **91** in **Abb.18** werden mit dem signal-Statement vorweg 14 Signalleitungen definiert. Diese werden später zum Verbinden der einzelnen Schaltungskomponenten benötigt. Dabei werden jeweils ein Bezeichner und ein Datentyp angegeben.
- In den **Zeilen 93** bis **111** in **Abb.18** wird die Struktur der Schaltung beschrieben. Dabei werden Instanzen der vorher angegebenen Bausteinprototypen erstellt und diese mittels der vorher definierten Signalleitungen verbunden. Diese Zeilen sind im Grunde eine textuelle Beschreibung des in **Abb.14** grafisch dargestellten internen Aufbaus der ALU. In **Zeile 94** wird mit „i_mux1 : mux2way16" ein 16-bit 1-aus-2 Multiplexer instanziiert. Dieser befindet sich in **Abb.14** links oben. Mit dem „port map"- Statement werden dessen Ein- und Ausgänge konfiguriert. Eingang a wird an den Operanden x der ALU angeschlossen (a => x), Eingang b mit der Konstanten 0 (b => „0000000000000000"), der sel-Eingang mit dem Control-Bit zx der ALU (sel => zx) und o mit dem Signal mux1_s (o => mux1_s) verbunden. Das Signal mux1_s wird in den Instanzen i_not1 und i_mux_2 jeweils als Eingang verwendet. In dieser Form wird die Struktur und der Datenpfad der ALU fortlaufend beschrieben, bis schließlich in den **Zeilen 105, 106** und **110** die ALU-Ausgänge o, ng und zr beschalten werden.

In einem nächsten Schritt sollte die entworfene Schaltung mittels Simulation getestet werden. Dazu wird die Software **ModelSim** und eine sogenannte Testbench verwendet. Eine Testbench ist eine VHDL-Datei, welche das Modell der entworfenen Schaltung einbindet und der Reihe nach mit unterschiedlichen Eingangssignalen konfrontiert. Als Ergebnis erhält man einen Plot von Ein- und Ausgangssignalen mit dem die richtige Funktionsfähigkeit der Schaltung überprüft werden kann.

Abb.19 zeigt einen Ausschnitt eines solchen Plots. In diesem werden mit den Eingabe-Werten x_s=17 und y_s=3 für die beiden Operanden und den Control-Bit-Eingaben für die ALU-Funktionen x + y (zx_s=0, nx_s=0, zy=0, ny_s=0, f_s=1, no=0), x – y, y – x, x & y sowie x | y die von der Schaltung ausgegebenen Ergebnisse (o_s) nacheinander von links nach rechts angezeigt. In der Spalte „Msgs" werden alle Signalwerte in numerischer Form für die Position ausgegeben, an der sich der gelbe Balken gerade befindet (x + y). Der Balken kann mit der Maus verschoben werden.

Nach der erfolgreichen Simulation und Testung der Schaltung mit ModelSim, folgt deren Implementierung auf dem FPGA-Board. Dazu wird die Entwicklungsumgebung **Quartus Prime** verwendet. **Abb.20** zeigt das Anwendungsfenster.

Es muss dafür ein neues Quartus Prime-Projekt angelegt und diesem müssen neben der Datei mit dem VHDL-Modell der ALU auch die VHDL-Dateien aller verwen-

```
77
78      signal mux1_s : std_logic_vector(15 downto 0);
79      signal not1_s : std_logic_vector(15 downto 0);
80      signal mux2_s : std_logic_vector(15 downto 0);
81      signal mux3_s : std_logic_vector(15 downto 0);
82      signal not2_s : std_logic_vector(15 downto 0);
83      signal mux4_s : std_logic_vector(15 downto 0);
84      signal and_s  : std_logic_vector(15 downto 0);
85      signal add_s  : std_logic_vector(15 downto 0);
86      signal mux5_s : std_logic_vector(15 downto 0);
87      signal not3_s : std_logic_vector(15 downto 0);
88      signal o_s    : std_logic_vector(15 downto 0);
89      signal or1_s  : std_logic;
90      signal or2_s  : std_logic;
91      signal or3_s  : std_logic;
92
93  begin
94      i_mux1 : mux2way16 port map(a => x, b => "0000000000000000", sel => zx, o => mux1_s);
95      i_not1 : not16 port map(i => mux1_s, o => not1_s);
96      i_mux2 : mux2way16 port map(a => mux1_s, b => not1_s, sel => nx, o => mux2_s);
97      i_mux3 : mux2way16 port map(a => y, b => "0000000000000000", sel => zy, o => mux3_s);
98      i_not2 : not16 port map(i => mux3_s, o => not2_s);
99      i_mux4 : mux2way16 port map(a => mux3_s, b => not2_s, sel => ny, o => mux4_s);
100     i_and1 : and16 port map(a => mux2_s, b => mux4_s, o => and_s);
101     i_add  : add16 port map(a => mux2_s, b => mux4_s, o => add_s);
102     i_mux5 : mux2way16 port map(a => and_s, b => add_s, sel => f, o => mux5_s);
103     i_not3 : not16 port map(i => mux5_s, o => not3_s);
104     i_mux6 : mux2way16 port map(a => mux5_s, b => not3_s, sel => no, o => o_s);
105     o <= o_s;
106     i_and2 : and_gate port map(a => '1', b => o_s(15), o => ng);
107     i_or1  : or8way port map(i => o_s( 7 downto 0), o => or1_s);
108     i_or2  : or8way port map(i => o_s(15 downto 8), o => or2_s);
109     i_or3  : or_gate port map(a => or1_s, b => or2_s, o => or3_s);
110     i_not4 : not_gate port map(i => or3_s, o => zr);
111 end structure;
112
```

Abb.18: VHDL-Beschreibung der „From Nand To Tetris"-ALU, Teil 2.

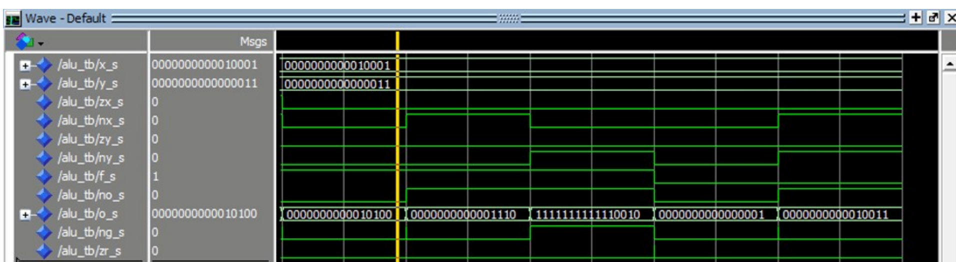


Abb.19: Ausschnitt der ModelSim-Ausgabe zur Simulation und Testung des VHDL-Modells der „From Nand To Tetris"-ALU.

Abb.20: Implementierung der „From Nand To Tetris"-ALU auf dem Terasic DE0-CV FPGA-Board mit Quartus Prime.

deten Schaltungskomponenten hinzugefügt werden. Darüber hinaus ist noch eine weitere VHDL-Datei erforderlich.

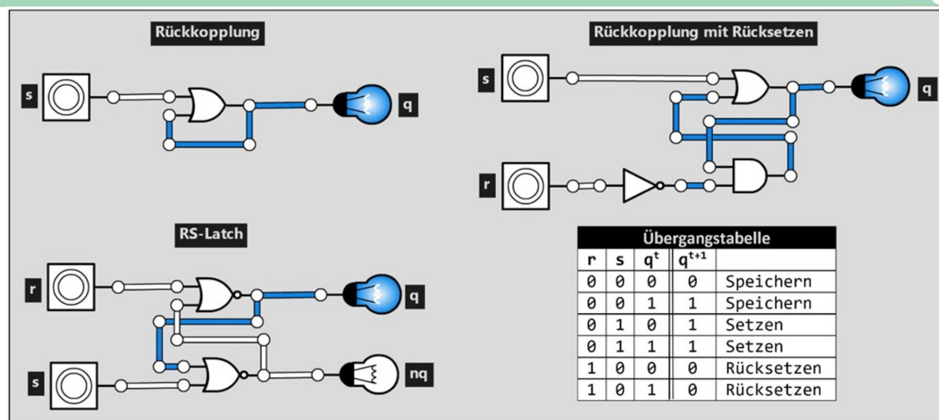
In dieser Datei werden die verwendeten Board-Komponenten (GPIO-Schnittstelle, Switches, LEDs und 7-Segment-Displays) mit ihren Ports als Top-Level-Entity definiert (alu_de0cv), die ALU-Schaltung als Komponente (alu) eingebunden, instanziiert (i_alu) und mit den Ports der Board-Komponenten verbunden. Neben der ALU sind in dieser Datei noch zwei weitere Schaltungskomponenten enthalten. Die Komponente bin16s2bcds dient zur Umwandlung der binären ALU-Ausgabe in das dezimale Format. Die Komponente bcds2sseg wird zur Darstellung der dezimalen Ziffern auf den 7-Segment-Displays benötigt. In **Abb.20** sind links oben im Bereich „Project Navigator“ alle Komponenten der Schaltung alu_de0cv mit ihren Instanzen angeführt. Das Editorfenster rechts daneben zeigt den oberen Teil der Datei alu_de0cv.vhd mit der (teilweise verdeckten) Entity-Definition.

Bevor die zur Konfiguration des FPGA-Chips erforderliche Bitstream-Datei erstellt werden kann, muss noch die Zuordnung der Ports der Board-Komponenten zu den Pins des FPGA-Chips vorgenommen werden. Dies kann mit Hilfe einer Assignment-Datei oder mit dem Quartus Pin Planner erfolgen.

Danach kann über die dreieckige Schaltfläche in der Buttonleiste des Anwendungsfensters die Kompilation des Designs durchgeführt werden. Der Fortschritt dieses Vorgangs lässt sich im „Bereich Tasks“ der Quartus- Umgebung mitverfolgen. In **Abb.20** ist dieser links, unterhalb von „Projects Navigator“ zu sehen. Es werden dabei mehre Phasen durchlaufen. Falls Fehler auftreten, müssen diese beseitigt werden. Die erzeugte Bitstream-Datei wird im Projekt-Unterverzeichnis „output files“ gespeichert und hat die Bezeichnung der Top-Level-Entität mit der Dateierweiterung „sof“ (SRAM object file).

Als letzter Schritt erfolgt der Download des Bitstreams auf das Board. Dazu muss es per USB-Kabel mit dem Entwicklungscomputer verbunden sein. Zum Download wird der Programmer über die gleichnamige Schaltfläche gestartet. In **Abb.20** ist rechts unten das Programmer-Dialogfenster zu sehen. Nachdem die Verbindung mit der auf dem Board vorhandenen USB-Blaster-Schnittstelle aufgebaut wurde, kann die Übertragung der Bitstream-Datei vorgenommen werden.

Damit ist das Terasic DE0-CV FPGA-Board als „From Nand To Tetris“-ALU konfiguriert. Über die DIP-Schalter auf dem Steckbrett können die beiden Operanden x und y eingegeben und über die Switches auf dem Board können die sechs Control-Bits zx, nx, zy, ny, f und no gesetzt werden, um eine der 18 Funktionen der ALU auszu-



r	s	q^t	q^{t+1}	
0	0	0	0	Speichern
0	0	1	1	Speichern
0	1	0	1	Setzen
0	1	1	1	Setzen
1	0	0	0	Rücksetzen
1	0	1	0	Rücksetzen

Abb.21: Grundlegende digitale Speicherschaltungen, erstellt mit Logicly.

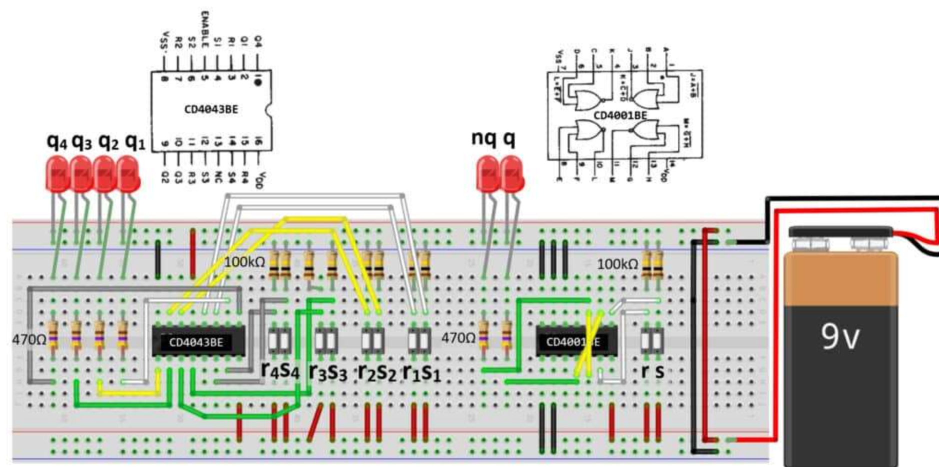


Abb.22: Ein RS-Latch mit zwei NOR-Gattern des CD4001BE (rechts), vier RS-Latches mit dem CD4043BE (links) (Grafik erstellt mit Fritzing).

wählen. Das von der ALU ermittelte Ergebnis wird auf den 7-Segment-Displays in dezimaler Form ausgegeben (vgl. **Abb.16**).

Der gesamte Implementierungsprozess ist ohne Zweifel arbeitsintensiv und zeitaufwendig. Der Unterricht muss daher je nach pädagogischer Zielsetzung entsprechend geplant und vorbereitet werden. Mit vorinstallierten und entsprechend vorkonfigurierten Systemen lässt sich der Arbeitsaufwand für die Lernenden deutlich verringern. Wenn es in erster Linie um einen „Einstieg in das Chipdesign“ geht, kann beispielsweise nur mehr das Vervollständigen der VHDL-Beschreibung der ALU sowie das Kompilieren und Übertragen auf das FPGA-Board als Aufgabenstellung verbleiben. Das mögliche Erfolgserlebnis, „seinen eigenen Mikrochip erstellt zu haben“, sollte dabei für eine entsprechende Motivation sorgen.

Digitale Speicherelemente

Alle bisher erörterten Schaltungen gehören in den Bereich der kombinatorischen Logik. Bei diesen Schaltungen sind die Signalwerte der Ausgangsleitungen unmittelbar und ausschließlich von den Signalwerten der Eingangssignale abhängig. Sie verfügen über „kein Gedächtnis“. In der Digitaltechnik werden aber auch Schaltungen benötigt, welche Informationen speichern können. Diese Schaltungen werden als Schaltwerke bezeichnet und zählen zur sequenziellen Logik. Bei diesen werden die ausgegebenen Signale nicht nur durch die aktuell anliegenden Eingabewerte bestimmt, sondern auch durch den Zustand, in dem sich die Schaltung gerade befindet.

Wie lassen sich aus Logikgattern Informationsspeicher erstellen? Die Antwort auf diese Frage lautet: durch Rückkopplung. Dabei wird ein Ausgangssignal wieder zurück in einen Eingang der Schaltung geführt. Dieser Effekt lässt sich mit Logicly und der Schaltung in **Abb.21** links oben sehr anschaulich demonstrieren. Nachdem der Button (s) einmal betätigt wurde, gibt das ODER-Gatter durch den Rückkopplungseffekt den Wert 1 aus, auch dann, wenn der Button nicht mehr gedrückt wird. Damit wird die Information „Button wurde gedrückt“ gespeichert. Was bei dieser Schaltung fehlt, ist eine Rücksetzungsmöglichkeit des Ausgangswerts (q) von 1 auf 0.

Dies kann, wie bei der Schaltung in **Abb.21** rechts oben, durch das Hinzufügen eines NOT- und eines AND- Gatters erreicht werden. Beim Drücken des unteren Buttons (r) wird dessen Signalwert 1 durch das NOT-Gatter in den Wert 0 invertiert und damit liefert das nachgeschaltete UND-Gatter nicht mehr den Wert 1 sondern den Wert 0 an das ODER-Gatter. Nachdem der obere Button (s) nicht gleichzeitig gedrückt werden kann, haben nun beide Eingänge des ODER-Gatters den Wert 0 und damit nimmt auch dessen

Ausgangssignal diesen Wert an, wird also zurückgesetzt. Beim Drücken des oberen Buttons wird wiederum die positive Rückkopplung aktiv. Eine solche Schaltung wird RS-Latch genannt. R steht dabei für Reset und S für Set.

Ein RS-Latch ist ein binärer Zustandsspeicher, d.h. es kann sich in einem von zwei Zuständen befinden, diese lassen sich auch mit 0 und 1 bezeichnen. In der Praxis wird ein RS-Latch häufig mit zwei rekursiv zusammengeschalteten NOR-Gattern implementiert. Die Schaltung in **Abb.21** links unten zeigt diese Realisierungsform. Der aktuell eingenommene Zustand wird dabei am Ausgang q angezeigt. Der zweite Ausgang nq ist stets mit der Negation von q beschaltet und damit im Grunde überflüssig. Da dieser Wert aber ohnehin intern anfällt, wird er in den meisten Fällen auch nach außen geführt. In der Übergangstabelle des RS-Latches taucht die binäre Zustandsvariable q zweimal auf – einmal als aktueller Zustand qt und einmal als Folgezustand qt+1, in den das Latch übergeht, wenn es sich im Zustand qt befindet und die Eingangssignale r und s anliegen. Die Kombination rs = 11 ist an den Eingängen eines NOR-implementierten RS-Latches zu vermeiden. Aufgrund der gatter- und leistungsbedingten Laufzeitverzögerung kann es dadurch in einen Schwingungszustand geraten.

Wie in **Abb.22** rechts dargestellt, lässt sich ein solches RS-Latch unter Verwendung eines CD4001BE-ICs auf einem Steckbrett sehr einfach aufbauen. Dabei werden zwei seiner vier NOR-Gatter genutzt. Die Eingänge der nicht verwendeten Gatter werden auf Masse gezogen. Die beiden gelben Steckbrücken sind die rückgekoppelten Signalleitungen. Ein kurzes Drücken auf den rechten Taster (s) bringt die rechte LED (q) dauerhaft zum Aufleuchten. Mit dem linken Taster (r) wird die linke LED (nq) anhaltend aktiviert.

Die Schaltung in **Abb.22** links verwendet den CD4043BE-IC. Dieser enthält vier RS-Latches auf NOR-Basis. Der Baustein verfügt nur über q- und keine nq-Ausgänge. Damit die q-Signalwerte ausgegeben werden, muss auch auf den Enable-Anschluss (Pin 5) die Versorgungsspannung (VDD) gelegt werden. Mit den s- und r- Tastern können dann die zugehörigen LEDs anhaltend ein- und ausgeschaltet werden.

Bei den in **Abb.21** und **Abb.22** vorgestellten RS-Latches kann eine Zustandsänderung zu jeder beliebigen Zeit erfolgen. Diese Schaltungen werden aus diesem Grunde als asynchrone Speicherelemente bezeichnet. Im Gegensatz dazu sind bei synchronen Schaltungen Zustandsänderungen nur innerhalb bestimmter Zeitintervalle möglich. Ein synchrones Schaltverhalten ist vor allem in Computersystemen wichtig. In diesen Systemen müssen die meisten Vorgänge zeitlich getaktet erfolgen. Nur so kann der beim Ausführen von Programmen ablaufende Von-Neumann-Zyklus funktionieren. Zur zeitlichen Synchronisation wird üblicherweise ein Taktsignal verwendet. Es entspricht einer periodischen Rechteckschwingung und wechselt zwischen den Signalwerten 0 und 1. Es wird dabei zwischen der positiven Taktphase (Signalwert 1) und der negativen Taktphase (Signalwert 0) sowie der positiven (steigenden) Taktflanke und der negativen (fallenden) Taktflanke unterschieden.

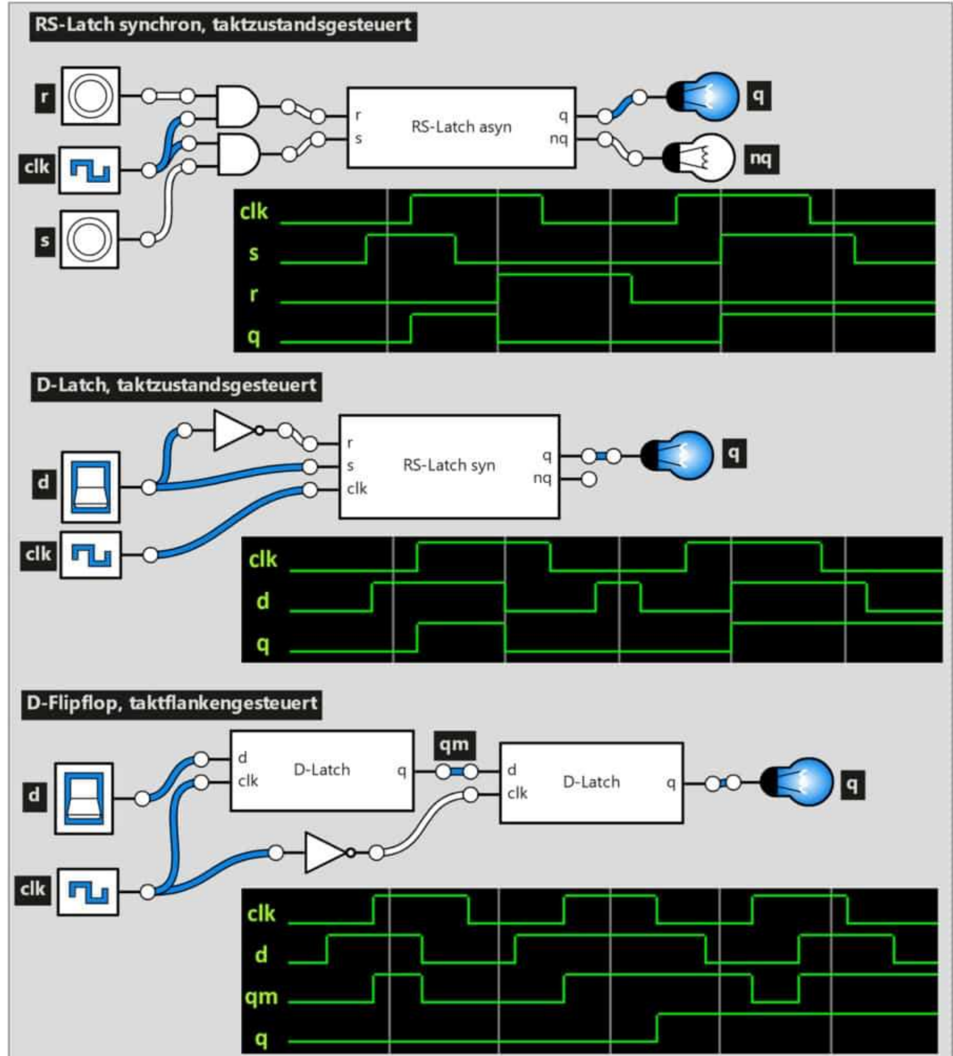


Abb.23: Vom RS-Latch zum D-Flipflop, Schaltungen erstellt mit Logicly, Zeitdiagramme erstellt mit ModelSim.

Abb.23 zeigt – das Abstraktionsprinzip mit Logicly nutzend – die Entwicklung vom asynchronen RS-Latch bis zum D-Flipflop, welches wiederum als Basis für weitere Speicherbausteine dient. Diese Entwicklung ist für das Verständnis der Funktionsweise digitaler Speicherelemente von entscheidender Bedeutung und wird deshalb hier ausführlicher behandelt. Bei jeder Schaltung ist auch ein Zeitdiagramm vorhanden, um die unterschiedlichen Schaltverhalten in Abhängigkeit vom Taktsignal zu verdeutlichen. Das Taktsignal wird bei Logicly von einem Clock-Baustein geliefert. Bei diesem lässt sich auch die Zykluszeit einstellen.

In der ersten Schaltung in **Abb.23** wird das aus zwei NOR-Gattern bestehende asynchrone RS-Latch (vgl. **Abb.22**) in Form eines IC- Bausteins (RS-Latch asyn) zum Aufbau des synchronen RS-Latches genutzt. Dabei kommt zu den Eingängen r und s der clk-Eingang für das Taktsignal hinzu, welches mit r und s jeweils über ein UND- Gatter verknüpft ist. Dies sorgt dafür, dass das Latch nur während einer positiven Taktphase gesetzt oder rückgesetzt werden kann. Es wird daher auch als taktzustands- gesteuertes synchrones RS-Latch bezeichnet.

Das **D-Latch** in **Abb.23** ist ebenfalls taktzustandsgesteuert. Im Gegensatz zum RS-Latch besitzt es neben dem Taktsignaleingang clk nur einen Eingang d. In der Schaltung wird das synchrone RS-Latch als IC (RS-Latch syn) genutzt. Dessen Eingänge r und s werden mit dem Signal d verbunden, wobei bei r ein NOT-Gatter vorgeschaltet ist. Bei d = 0 wird dadurch das Latch zurückgesetzt und bei d = 1 gesetzt. Das Signal d kann damit direkt den zu speichernden Bit-Wert repräsentieren.

Durch die Zustandssteuerung des D-Latches über die Taktphase wird zwar ein gewisser Grad an Synchronisation erreicht, bei Computersystemen ist es jedoch wichtig, potenzielle Zustandswechsel noch weiter einzuschränken. Genau dies

CLUBDEV.NET

erfolgt bei taktflankengesteuerten synchronen Speicherelementen, die einen Zustandswechsel nur noch während Taktflanken erlauben. Taktflankengesteuerte Speicherelemente werden als Flipflops bezeichnet.

Das **D-Flipflop** in **Abb.23** ist negativ taktflankengesteuert, d.h. es kann seinen Zustand nur dann ändern, wenn sich das Taktsignal von 1 nach 0 ändert, sich also in der negativen oder fallenden Flanke befindet. Dieses Schaltverhalten wird erreicht, indem zwei D-Latch-ICs (D-Latch) hintereinandergeschaltet und mit inversen Taktsignalen betrieben werden. Das erste Latch wird als Master und das zweite Latch als Slave bezeichnet, die Schaltung wird deshalb auch Master-Slave-Flipflop genannt. Wird das Master-Latch mit dem inversen Taktsignal betrieben, ist die Schaltung positiv taktflankengesteuert, wird, wie in **Abb.23**, das Taktsignal des Slave-Latches invertiert, so ergibt sich ein negativ taktflankengesteuertes Flipflop. Im Zeitdiagramm des D- Flipflops ist zur Veranschaulichung der Funktionsweise auch der Verlauf des internen Signals qm enthalten. Es verbindet den Ausgang q des Master-Latches mit dem Eingang d des Slave-Latches.

Die Funktionalität des D-Flipflops erscheint auf den ersten Blick „etwas kläglich“. Es ist nur im Stande, den am Eingang d anliegenden Signalwert über einen Taktzyklus hinweg zu speichern. Bei jeder nächsten steigenden bzw. fallenden Flanke des Taktsignals wird dieser Wert immer wieder neu eingelesen. Das D- Flipflop ist aber ein wichtiger Baustein, um weitere digitale Speicherelemente zu implementieren. Dies wird in **Abb.24** gezeigt.

Zunächst muss dafür gesorgt werden, dass das D-Flipflop den d-Signalwert nicht nur über einen Taktzyklus hinweg speichert, sondern so lange wie dies benötigt wird. Diese Schaltung wird hier als **1-bit Register** bezeichnet und ist in **Abb.24** oben dargestellt. Die dauerhafte Speicherung wird hier durch die Verwendung eines 1-aus-2 Multiplexer-Bausteins (vgl. **Abb.11**) ermöglicht. Dieser versorgt bei sel=0 den d-Eingang des Flipflops immer mit wieder mit dessen Ausgangssignal q, also mit dem aktuell gespeicherten Wert. Bei sel=1 wird hingegen ein neuer Wert vom d-Eingang der Schaltung eingelesen. Der sel-Eingang des Multiplexers ist mit dem e-Eingang (enable) der Schaltung verbunden. Das dort anliegende Signal bestimmt also, ob das Register bei der nächsten steigenden bzw. fallende Flanke speichern (e=0) oder neu einlesen (e=1) soll.

Wird diese 1-bit Registerschaltung wiederum in einen IC-Baustein (Reg1) gepackt, so lassen sich damit n-bit breite Register zur Manipulation von n-bit Datenwörtern aufbauen. **Abb.24** zeigt ein **4-bit Register** als parallele Anordnung von vier dieser ICs, die hier negativ taktflankengesteuert

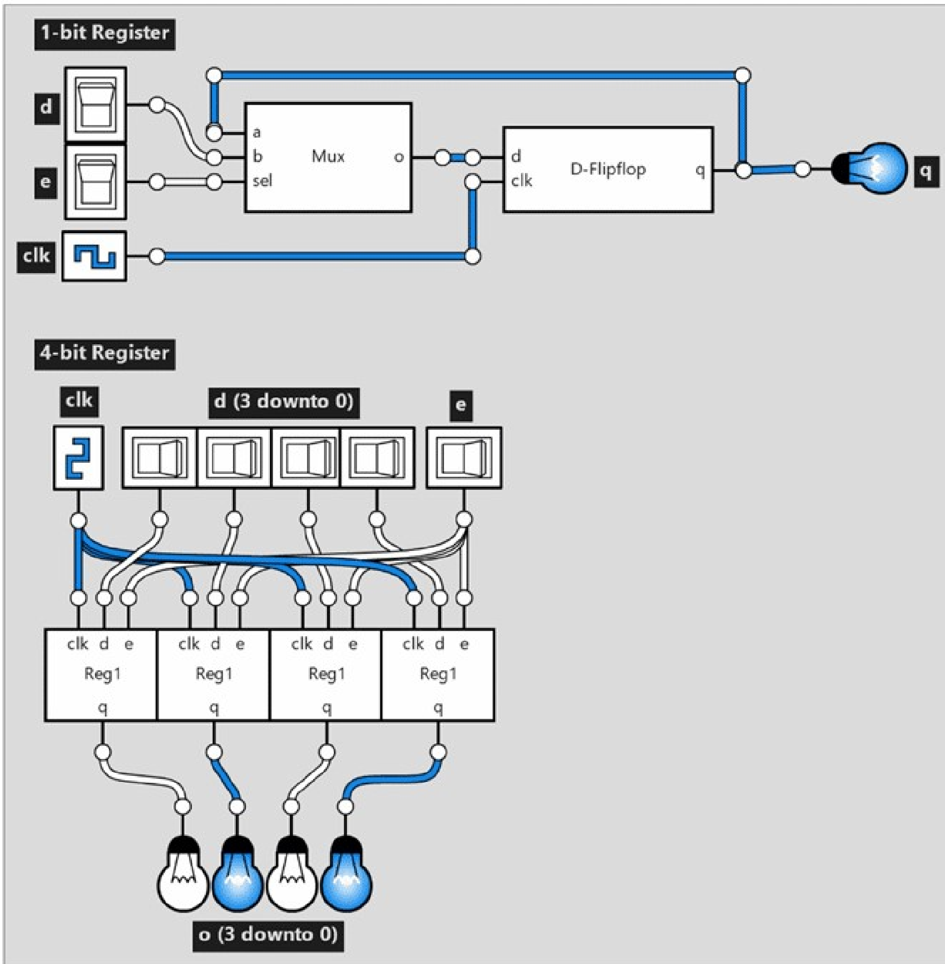


Abb.24: Vom D-Flipflop zum 4-bit Register, Schaltungen erstellt mit Logicly.

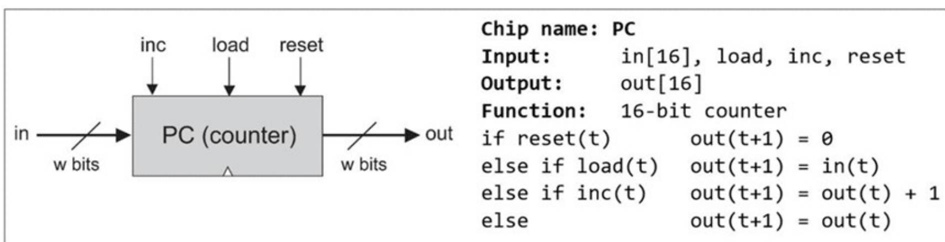


Abb.25: Spezifikation der „From-Nand-To-Tetris“-Befehlszählerschaltung (Quelle: „From Nand To Tetris“-Projekt).

synchron über eine gemeinsame Taktleitung (clk) und über eine gemeinsame Enable-Leitung (e) betrieben werden.

Wie schon erwähnt, lassen sich mit dieser parallelen Anordnung Register in beliebiger Breite erstellen. Werden mehrere Register kombiniert, kann ein Registersatz, auch Registerfile oder Registerbank genannt, aufgebaut werden. Derartige Registerstrukturen finden sich als sehr schnelle leistungsfähige Speicher im inneren Kern einer CPU.

Befehlszähler

Zählerbausteine stellen eine wichtige Klasse sequenzieller Standardkomponenten dar. Mit ihrer Hilfe lassen sich zeitlich aufeinanderfolgende Ereignisse registrieren oder auch erzeugen. Der Zählerstand muss dabei immer gespeichert werden.

In einem Computersystem ist der Befehlszähler oder Befehlszeiger (program counter oder instruction pointer) eine Schlüsselkomponente, die den Von-Neumann-Zyklus antreibt. Er dient zur Adressierung des Programmspeichers und enthält zu jedem Zeitpunkt die Adresse der Speicherzelle mit dem nächsten auszuführenden Befehl. Die Programmbefehle sind zumeist nacheinander im Hauptspeicher abgelegt. Die sequenzielle Abarbeitung des Programms wird also erreicht, indem sich der Befehlszähler bei jedem Takt schrittweise erhöht. Bei Sprungbefehlen kann nicht einfach weiter gezählt werden. Dazu benötigt die Befehlszählerschaltung eine Lademöglichkeit, mit deren Hilfe der aktuelle Zählerstand mit einem extern angelegten Datenwort überschrieben werden kann. Ein Befehlszähler sollte überdies auch noch über einen eigenen Eingang für die Rücksetzung auf den Wert 0 verfügen.

Auch im Rahmen des „From Nand To Tetris“-Projekts soll für das zu erstellende virtuelle Computersystem eine Befehls-

zählerschaltung entwickelt werden. Diese Schaltung ist Teil der CPU. **Abb.25** zeigt deren Spezifikation.

Abb.26 präsentiert eine mögliche schaltungstechnische Umsetzung dieser Spezifikation mit Logicly. Der Einfachheit halber als 4-bit-Version, im Original ist es ein 16-bit Befehlszähler.

Die Schaltung enthält ein negativ taktflankengesteuertes Register (Reg4), einen Addierer (Add4) und drei 1-aus-2-Multiplexer (Mux4). Das Taktsignal für die gesamte Schaltung wird vom Clock-Baustein oberhalb des Registers geliefert. Das Register dient zum Speichern des aktuellen Zählerstandes. Dieser wird über die Light-Bulbs (o) angezeigt. Da der Enable-Eingang (e) des Registers fix auf den Wert 1 gesetzt ist, erfolgt die Speicherung des Zählerstandes immer nur über einen Taktzyklus hinweg. Entscheidend ist daher, welcher Signalwert bei jeder fallenden Taktflanke am Eingang anliegt.

Bei inc=0, load=0 und reset=0 wird der vom Register ausgegebene Wert wieder unverändert eingelesen und der Zählerstand bleibt gleich Bei inc=1, load=0 und reset=0 wird der vom Addierer um 1 erhöhte Zählerstand übermittelt, es wird also hinaufgezählt. Mit inc=0|1, load=1 und reset=0 wird der in-Signalwert eingelesen und weitergeleitet, der Zähler also auf diesen Wert gesetzt. Bei inc=0|1, load=0|1 und reset=1 wird der Zählerstand auf den Wert 0 zurückgesetzt. Für die Umsetzung der in der Spezifikation beschriebenen Priorität der Eingangssignale inc, load und reset ist die Abfolge der Multiplexer im Datenpfad der Schaltung entscheidend.

Abb.27 zeigt den Aufbau für die 16-bit Implementierung dieser Schaltung auf dem Terasic DE0-CV FPGA- Board.

Der Zählerstand wird im dezimalen Format (0 bis 65535) ausgegeben. Dazu werden fünf der sechs 7-Segment-Anzeigen des Boards genutzt. Für die Eingabe der Steuersignale inc, load und reset werden drei Switches verwendet. Deren Einstellung wird zusätzlich durch die darüber befindlichen LEDs angezeigt. Das Setzen des Zählers auf einen bestimmten Wert erfolgt über die Schnittstelle GPIO_1. Deren Pins sind über ein Flachbandkabel mit dem 40-Pin GPIO-Breakoutboard auf dem Steckbrett verbunden. Die Pins 1 bis 10 sowie 13 bis 18 des Breakoutboards sind an die zwei 8-poligen DIP-Schaltermodule auf dem Steckbrett angeschlossen. Pin 11 liefert 5V, Pin 29 liefert 3.3V und Pin 12 sowie Pin 30 liefern GND. Bei den DIP-Switches befinden sich masseseitig 100kΩ-Widerstände. Der FPGA-Chip stellt vier Taktsignale mit 50MHz zur Verfügung. Eines davon wird über einen Frequenzteiler verwendet, um für den Zähler ein Signal mit annähernd 1Hz zu erhalten.

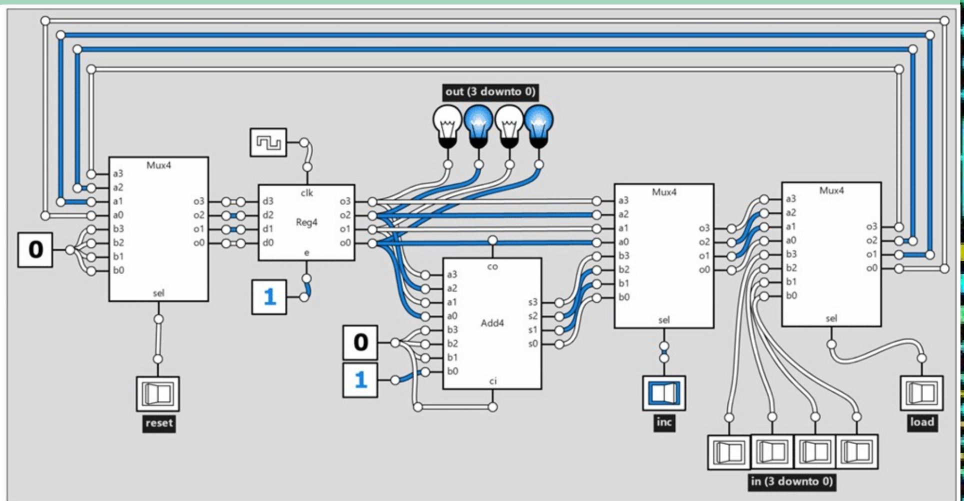


Abb.26: „From Nand To Tetris"-Befehlszählerschaltung als 4-bit Variante, erstellt mit Logicly.

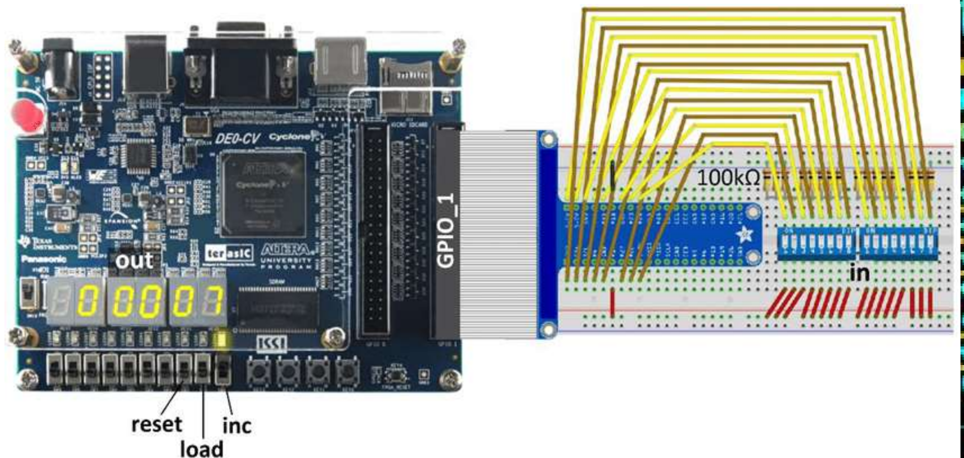


Abb.27: Aufbau zur Implementierung der „From Nand To Tetris"-Befehlszählerschaltung auf dem Terasic DE0-CV FPGA-Board (Grafik erstellt mit Fritzing).

Die weiteren Schritte entsprechen der schon bei der FPGA-Implementierung der ALU beschriebenen Vorgangsweise. Da es in erster Linie um die physische Umsetzung der entworfenen Schaltungen geht, wird aus didaktischen Gründen wiederum eine VHDL-Strukturbeschreibung verwendet. In der Praxis würde bei einem Befehlszähler ein Verhaltensmodell zur Anwendung kommen.

Abb.28 zeigt die verwendete VHDL-Beschreibung der Befehlszählerschaltung.

In der Entity-Definition werden die Ein- und Ausgänge des Befehlszählers definiert. Danach sind die Prototypen aller verwendeten Schaltungskomponenten (reg16, add16 und mux2way16) und die notwendigen Signalleitungen für deren Verbindung angeführt. Mit dem begin-Schlüsselwort wird die eigentliche Schaltungsbeschreibung eingeleitet. Hier werden die notwendigen Instanzen der Schaltungskomponenten erzeugt und mit den definierten Signalen sowie den Ein- und Ausgängen der Schaltung verbunden. Die einzelnen Schaltungskomponenten werden dabei in der gleichen Reihenfolge beschrieben, wie sie in **Abb.26**, von links nach rechts vorhanden sind.

In **Abb.29** wird die VHDL-Beschreibung der Top-Level-Entity (pc_de0cv) zur Implementierung der Befehlszählerschaltung auf dem Terasic DE0-CV FPGA-Board wiedergegeben.

In der Entity-Definition in **Abb.29** sind alle für die Befehlszählerschaltung verwendeten Board-Komponenten angeführt: die Taktsignalquelle (CLOCK_50), die GPIO-Schnittstelle (GPIO_1), die Switches (SW), die LEDs (LEDR) und die fünf 7- Segmentanzeigen (HEX0 bis HEX4). Die Zuordnung zwischen den für diese Komponenten verwendeten Kurzbezeichnungen und den betreffenden Pins auf dem FPGA-Chip wurde mit einer Assignment-Datei vorgenommen.

Bei den Schaltungskomponenten ist an erster Stelle eine Binärzählerschaltung (bincntr_g) angeführt. Diese dient zur Verringerung der von CLOCK_50 vorgegebenen Taktfrequenz von 50MHz auf etwa 1Hz. Die Variable n gibt dabei die Breite des Zählers in Bits an. Da es sich um eine sogenannte generische Komponentenbeschreibung handelt (generic), kann der Wert dieser Variablen auch noch weiter unten bei der Instanziierung angepasst werden. Die Komponente pc ist die eigentliche Befehlszählerschaltung, wie in

Abb.28 beschrieben. Die beiden Schaltungskomponenten bin16tobcd und bcs2sseg dienen zur Umwandlung des binären Zählerwerts in das dezimale Format (Binary Coded Decimal) und dessen Anzeige auf den 7-Segment-Display-Bausteinen.

Nach der Definition der benötigten Signalleitungen folgt die eigentliche Architekturbeschreibung. Dabei werden Instanzen der Schaltungskomponenten erzeugt und mit den definierten Signalleitungen sowie den Board-Komponenten verbunden.

Nach der Kompilation der VHDL-Beschreibungen mit Quartus Prime und der Übertragung des erstellten Bitstreams auf das FPGA-Board sollte der Zähler bei eingeschaltetem inc-Switch von 0 beginnend automatisch hinaufzählen. Über den load-Switch kann der Zähler auf den an den DIP-Schaltern eingestellten Wert und bei aktiviertem reset-Switch auf den Wert 0 gesetzt werden (vgl. **Abb.27**).

So wie bei der FPGA-Implementierung der ALU bereits erwähnt, kann der im Unterricht notwendige Arbeitsaufwand für diese Implementierung durch den Einsatz von entsprechend vorinstallierten und vorkonfigurierten Systemen deutlich reduziert werden.

```
1  -- pc.vhd (program counter)
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity pc is
7  port (
8      clk   : in  std_logic;
9      i     : in  std_logic_vector(15 downto 0);
10     inc   : in  std_logic;
11     load  : in  std_logic;
12     reset : in  std_logic;
13     o     : out std_logic_vector(15 downto 0)
14 );
15 end pc;
16
17 architecture structure of pc is
18
19     component reg16
20     port (
21         clk : in  std_logic;
22         i   : in  std_logic_vector(15 downto 0);
23         e   : in  std_logic;
24         o   : out std_logic_vector(15 downto 0)
25     );
26     end component;
27     component add16
28     port (
29         a : in  std_logic_vector(15 downto 0);
30         b : in  std_logic_vector(15 downto 0);
31         o : out std_logic_vector(15 downto 0)
32     );
33     end component;
34     component mux2way16
35     port (
36         a   : in  std_logic_vector(15 downto 0);
37         b   : in  std_logic_vector(15 downto 0);
38         sel : in  std_logic;
39         o   : out std_logic_vector(15 downto 0)
40
41     );
42     end component;
43
44     signal muxload_s : std_logic_vector(15 downto 0);
45     signal muxreset_s : std_logic_vector(15 downto 0);
46     signal reg_s : std_logic_vector(15 downto 0);
47     signal add_s : std_logic_vector(15 downto 0);
48     signal muxinc_s : std_logic_vector(15 downto 0);
49
50 begin
51     i_muxreset : mux2way16 port map
52         (a => muxload_s, b => "0000000000000000", sel => reset, o => muxreset_s);
53     i_reg : reg16 port map
54         (clk => clk, i => muxreset_s, e => '1',  o => reg_s);
55     o <= reg_s;
56     i_add16 : add16 port map
57         (a => reg_s, b => "0000000000000001", o => add_s);
58     i_muxinc : mux2way16 port map
59         (a => reg_s, b => add_s, sel => inc, o => muxinc_s);
60     i_muxload : mux2way16 port map
61         (a => muxinc_s, b => i, sel => load, o => muxload_s);
62 end structure;
63
```

Abb.28: VHDL-Beschreibung des „From Nand To Tetris“ Befehlszählers (pc, program counter)..

Schlusswort

Das hier vorgestellte didaktische und methodische Konzept zur Vermittlung von Grundlagen der Digitaltechnik endet mit der Vorstellung einer Befehlszählerschaltung „etwas willkürlich“. Dem Abstraktionsprinzip folgend wäre der Aufbau einer CPU-Schaltung ein möglicher nächster Schritt.

Das Design einer CPU wird in hohem Maße durch die abzubildende Computerarchitektur und die abzubildenden Maschinenbefehle, also durch den sogenannten Befehlssatz bestimmt. Die grundlegende Funktion einer CPU ist es, Maschinensprachenprogramme möglichst effizient auszuführen. Hier treffen Hard- und Software aufeinander. Schon bei der Entwicklung einer ALU, welche ein wesentlicher Bestandteil einer CPU ist, muss dies berücksichtigt werden.

Eine entsprechende Erörterung der Maschinensprachenprogrammierung im Allgemeinen und des in Betracht kommenden Befehlssatzes im Speziellen würde in diesem Beitrag, in dem es in erster Linie um die Vermittlung von Grundlagen der Digitaltechnik geht, wohl zu weit führen.

Eine mögliche Fortsetzung wäre die Vorstellung der Maschinensprache, der CPU und des damit aufgebauten Computersystems des „From Nand To Tetris“-Projektes. Wie schon erwähnt ist dieses Computersystem relativ einfach aufgebaut, aber leitungsfähig genug, um darauf Computerspiele wie Pong und Tetris mit Tastatureingabe und Bildschirmausgabe ausführen zu können.

Das „Form Nand To Tetris“-Projekt selbst sieht nur eine virtuelle Implementierung dieses Computersystems vor. Es wird in speziell dafür entwickelten Apps erstellt, getestet und genutzt. Im Zuge der Ausarbeitung des in diesem Beitrag vorgestellten Konzeptes konnte das „From Nand To Tetris“-Computersystem auch vollständig auf dem Terasic DE0-CV FPGA-Board implementiert werden.

In **Abb.30** sind zwei Bildschirmausgaben des auf dem „From Nand To Tetris“-Computersystem laufenden Computerspiels PONG gegenübergestellt. Der Screenshot links stammt von der CPU-Emulator-App des Projekts. Das Foto rechts zeigt die von der erstellten FPGA-Implementierung erzeugte Ausgabe auf einem VGA-Bildschirm.

Wie schon in der Einleitung dieses Beitrags erwähnt, ist die aktive Auseinandersetzung mit der Digitaltechnik vor allem dann interessant und spannend, wenn dabei die mitunter bestechend einfachen logischen Zusammenhänge und Konzepte entdeckt und verstanden werden können, die sich hinter den komplizierten Strukturen moderner digitaler Systeme verbergen. Das ist natürlich ganz besonders der Fall, wenn man ein selbst programmiertes Computerspiel auf einem Computersystem spielt, das man auch selbst nur aus Logikgattern aufgebaut hat!

Im Rahmen des „From Nand To Tetris“-Projektes ist dies möglich. Ein wesentlicher Aspekt dabei ist das Prinzip der Abstraktion. Shimon Schocken, einer der Begründer dieses Projektes, weist in einem TED-Talk (https://www.ted.com/) in amüsanter Weise auf die besondere Bedeutung dieses Prinzips hin, indem er einen Ausschnitt von Michelangelos

Meisterwerk an der Decke der Sixtinischen Kapelle in einer etwas modifizierten Form (vgl. **Abb.31**) mit den folgenden Worten präsentiert: *(**Abb31 siehe Seite 2**)*

„So we start this journey by telling our students that God gave us Nand and told us to build a computer, and when we asked how, God said, ‚One step at a time.'"

Literatur

- **Cord**, Elias: FPGAs für Maker. dpunkt, Heidelberg, 2016.
- **Harris**, Money David u. Sarah L. Harris: Digital Design and Computer Architecture. Elsevier, Boston. 2. Aufl. 2013.
- **Himpe**, Vincent: Digitale Logik selbst entwickeln. Elektor, Aachen, 2012.
- **Hoffmann**, W. Dirk: Grundlagen der technischen Informatik. Hanser, München, 6. Aufl. 2020.
- **Kleitz**, William: Digital Electronics with VHDL. Pearson, Essex, 2014.
- **Nisan**, Noam u. Shimon Schocken: The Elements of Computing Systems. Building a Computer from First Principles. MIT Press, Cambridge, 2. Aufl. 2021.
- **Petzold**, Charles: Code. The Hidden Language of Computer Hardware and Software. Microsoft Press, Redmond, 2013.
- **Reichardt**, Jürgen: Digitaltechnik. Eine Einführung mit VHDL. De Gruyter, Berlin, 4. Aufl. 2017.
- **Rost**, Manfred u. Sandro Wefel: Elektronik für Informatiker. De Gruyter, Berlin, 2. Aufl. 2021.
- **Schulz**, Peter u. Edwin Naroska: Digitale Systeme mit FPGAs entwickeln. Elektor, Aachen, 2016.

Links

- „Nand To Tetris"-Projektwebsite https://www.nand2tetris.org/
- „Nand To Tetris"-MOOC, Teil 1 https://www.coursera.org/learn/build-a-computer
- „Nand To Tetris"-MOOC, Teil 2 https://www.coursera.org/learn/nand2tetris2
- „Nand To Tetris"-TED-Talk https://www.ted.com/talks/shimon_schocken_the_self_organizing_computer_course
- Nandgame https://nandgame.com/
- Logicly https://logic.ly/
- Tinkercad Circuits https://www.tinkercad.com/circuits
- Terasic DE0-CV FPGA-Board https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=921#contents
- Intel Quartus Prime Lite Edition Download https://www.intel.com/content/www/us/en/collections/products/fpga/software/downloads.html?edition=lite&s=Newest&f:guidetmD240C377263B4C70A4EA0E452D0182CA=%5BIntel%C2%AE%20Quartus%C2%AE%20Prime%20Design%20Software%3BIntel%C2%AE%20Quartus%C2%AE%20Prime%20Lite%20Edition%5D
- Fritzing https://fritzing.org/

```
-- pc_de0cv.vhd

library ieee;
use ieee.std_logic_1164.all;

entity pc_de0cv is
port (
    CLOCK_50 : in std_logic;
    GPIO_1   : in  std_logic_vector(35 downto 0);
    SW       : in  std_logic_vector(9  downto 0);
    LEDR     : out std_logic_vector(9  downto 0);
    HEX0     : out std_logic_vector(6  downto 0);
    HEX1     : out std_logic_vector(6  downto 0);
    HEX2     : out std_logic_vector(6  downto 0);
    HEX3     : out std_logic_vector(6  downto 0);
    HEX4     : out std_logic_vector(6  downto 0)
);
end pc_de0cv;

architecture structure of pc_de0cv is

    component bincntr_g is
    generic (n : integer := 25);
    port (
        clk : in std_logic;
        o   : out std_logic
    );
    end component;
    component pc is
    port (
        clk   : in  std_logic;
        i     : in  std_logic_vector(15 downto 0);
        inc   : in  std_logic;
        load  : in  std_logic;
        reset : in  std_logic;
        o     : out std_logic_vector(15 downto 0)
    );
    end component;
    component bin16tobcd
    port (
        i : in  std_logic_vector(15 downto 0);
        o : out std_logic_vector(19 downto 0)
    );
    end component;
    component bcd2sseg
    port (
        i : in std_logic_vector(3 downto 0);
        o : out std_logic_vector(6 downto 0)
    );
    end component;

   signal clk_s : std_logic;
   signal i_s   : std_logic_vector(15 downto 0);
   signal pc_s  : std_logic_vector(15 downto 0);
   signal bcd_s : std_logic_vector(19 downto 0);

begin
    i_bincntr_g : bincntr_g
      generic map (n => 25) port map (clk => CLOCK_50, o => clk_s);
    i_pc : pc port map
      (clk => clk_s, i => GPIO_1(15 downto 0), inc => SW(0), load => SW(1),
      reset => SW(2), o => pc_s);
    LEDR(2 downto 0) <= SW(2 downto 0);
    i_bcd : bin16tobcd port map (i => pc_s, o => bcd_s);
    i_sseg0 : bcd2sseg port map (i => bcd_s(3  downto  0), o => HEX0);
    i_sseg1 : bcd2sseg port map (i => bcd_s(7  downto  4), o => HEX1);
    i_sseg2 : bcd2sseg port map (i => bcd_s(11 downto  8), o => HEX2);
    i_sseg3 : bcd2sseg port map (i => bcd_s(15 downto 12), o => HEX3);
    i_sseg4 : bcd2sseg port map (i => bcd_s(19 downto 16), o => HEX4);
end structure;
```

Abb.29: VHDL-Beschreibung der Top-Level-Entity zur Implementierung des „From Nand To Tetris"-Befehlszählers auf dem Terasic DE0-CV FPGA-Board.

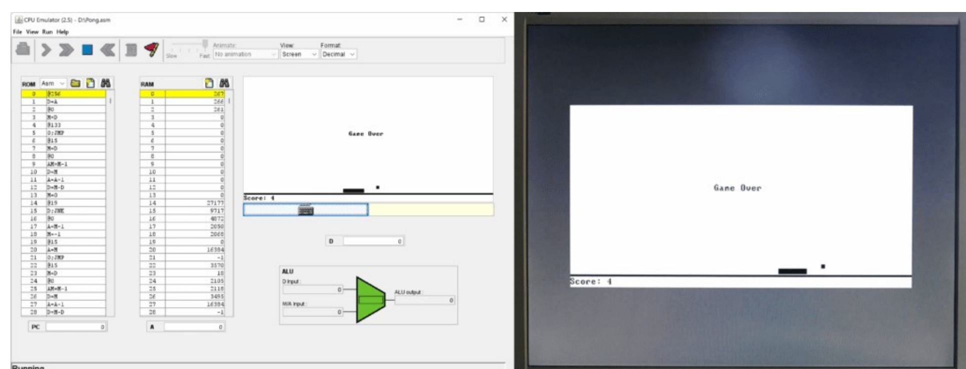


Abb.30: Das Computerspiel PONG, links in der CPU-Emulator-App des „From Nand To Tetris"-Projekts, rechts in der FPGA-Implementierung auf einem VGA-Monitor.

CLUBDEV.NET

Amazon Killer

Günter Hartl

Gefühlt bin ich sicher schon hundertmal daran vorbeigegangen. Beim Betrachten der liebevoll drapierten Auslagen fühlt man sich gleich um ein paar Jahrzehnte zurückversetzt. Da kommt nicht einmal YouTube, unsere digitale Müllhalde mit.

Der vorsichtige Blick ins Auslagenfenster bestätigte meine erfreulichsten Befürchtungen. Entweder hatte der Auslagendekorateur eine mehrjährige Haftstrafe abzusitzen oder der Laden diente als illegale China-Küche—würden jetzt Lebenserfahrene mutmaßen.

Von solch unreinen Gedanken muss ich mich jedoch auf das Schärfste distanzieren, ohne dabei den faszinierenden Blick von den angepriesenen Schuko Steckern abzuwenden. Ehrlich gesagt weiß ich auch gar nicht, ob das Geschäftslokal noch in Betrieb ist. Obwohl ich wie oben angedeutet schon öfters vorbeigegangen bin, war jedes Mal die Eingangstür während der üblichen Geschäftszeiten versperrt. Deshalb kann ich auch mit keinen Fotos vom Inneren aufwarten. Sorry.

Der naheliegende Schluss kann daher nur lauten, dass sie auch dort die Corona-Maßnahmen übererfüllen oder den Onlinehandel forcieren wollen. Keine Ahnung, ob es da überhaupt eine Webseite dazu gibt oder der Webmaster auch mit dem Auslagendekorateur eine Zelle teilt. Zumindest könnte diese Konstellation den so verheißungsvoll gepriesenen Kieselstein im Amazon-Schuh darstellen, welcher Besagtem nun die Umsätze verleidet.

Die mühsam aufrecht gehaltenen Widerstandsnester im Einzelhandel werden zwar meist vollmundig verbal verteidigt, während viele Verfechter dieser Haltung gleichzeitig ihre Amazon-Bestellungen fertig klicken. Manchmal steht auch ein digitaler Besuch ins Reich des Drachens an, wenn das Angebot passt. Und solange ein Steuermann sein Bier nicht absetzt und die 400 Meter Schüssel im Suezkanal querstellt, werden die Trümmer auch recht zeitnah eintreffen. Der Hauptgrund dafür ist meist bei einem lapidaren »ich bin ja nicht meines Geldes Feind« und irgendwas mit »bequemer, kein Anfahrtsweg und angelernte Verkäufer« zu verorten. Das ist auch nicht verwerflich, sondern zeigen nur die Umwälzungen sowohl im geschäftlichen als auch im gesellschaftlichen Kontext auf.

Viele Firmen haben ihre Arbeitsumgebungen mittlerweile in der Cloud laufen, wobei abends nach Feierabend alle Server gelöscht werden. Am nächsten Morgen werden Besagte dann per automatischer Neuinstallation wieder komplett frisch erzeugt *»Irgendwas mit Geld oder so«* war da als Hauptargument auszumachen, da somit die Kosten für den Nacht- und Wochenendbetrieb wegfallen. Und der technische Support mit dem verstümmelten Sprachgebrauch rechtfertigt so erst recht ein Stirnrunzeln beim Anfragenden, was dem Arbeitsklima nur zuträglich sein kann. Der einzige Lichtblick dabei sind die Lehrvideos auf YouTube von indisch geprägten Protagonisten. Immer wieder erheiternd, wenn deren Stimmen ein bisschen dumpfer werden, während sie vom Spickzettel das nächste Shell-Kommando ablesen. Die sparen auch schon bei den Headsets und knallen ihnen ein fixiertes Mikrofon vor das Gesicht. Und Sitzplätze haben sie auch immer zu wenige in ihren Zügen. Die haben es auch nicht leicht.

Das wird sich unser Amazon-Killer wahrscheinlich auch zu Herzen genommen haben und die Auslagen dementsprechend verwahrlosen lassen.

Wer sieht sich heutzutage auch Auslagen an? Das macht doch jemand nur so lange, bis die Frau mit dem Einkaufen fertig ist oder der Herdentrieb im Einkaufscenter einen daran unwillkürlich vorbeischleust. Somit kam ich zu dem Schluss, dass unser Amazon-Killer bisher alles richtig gemacht hat. Keine analogen Kunden bedeuten kein benötigtes Verkaufspersonal. Für die nächsten Lockdowns, sei es jetzt energiemäßig oder epidemisch, beließ man das äußere Erscheinungsbild einfach beim Alten. Und der klimaneutrale Fußabdruck wird mit der dezenten Auslagenbeleuchtung weiterhin gewahrt. Ob das Amazons Geschäftspolitik beeinflusst, wird die Zukunft zeigen.

Fest steht einmal, dass wir nun Mitte Oktober haben und zur besinnlichen Zeit die Angebote für Affären da weiters nicht ausbleiben. Wer braucht schon Affären mit kaffeeschlürfenden Gucci-Tanten? Sind die schon alle kognitiv zu früh abgebogen? Irgendwie kann ich mich nicht des Eindrucks erwehren, dass alles den Bach runtergeht. Der eine stellt seinen Pott im Suezkanal quer, der andere braucht einen Spickzettel, um seine Shell-Kommandos in Englisch runter zu lesen und nicht ausgebeutete, jungfräuliche Erdbeerpflückerinnen schulen jetzt digital auf Nächstenliebe um.

Antworten   Allen antworten   Weiterleiten

Ich werde in den nächsten Tagen wieder mal beim Amazon-Killer vorbeischauen, ob sich was an der Auslagengestaltung geändert hat. Das macht meiner Meinung nach mehr Sinn, als sich mit den angeschnittenen Themen über Gebühr zu beschäftigen.

Schöne Weihnachten und das Übliche halt.

Man liest sich! Gruß Günter

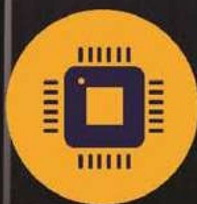
techbold

WIR BAUEN DEINEN PC

Nutze die langjährige Erfahrung der techbold Computer Experten für die perfekte Konfiguration deines PC-Systems. Egal ob Gaming Maschine, Office-PC oder Workstations für professionelle Anwendungen wie CAD, 3D Grafik und Videoschnitt – wir erstellen dir ein Angebot mit dem perfekten Preis-Leistungs-Verhältnis.

www.techbold.at/pc-zusammenstellen

BERATUNG
Umfangreicher Support von zertifizierten Experten



QUALITÄT
Ausschließlich geprüfte Markenkomponenten

TESTS
Jede Konfiguration wird umfangreich getestet

GARANTIE
3 Jahre Garantie auf alle individuellen PC-Systeme