

# Konzeptionierung eines Simulators für einen 8-Bit Prozessor

## Studienarbeit

Bachelor of Science

Studiengang Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Andreas Schmider, Nico Schrod**

Abgabedatum 16. Mai 2022

Bearbeitungszeitraum	2 Semester
Kurs	TINF19B3
Betreuer	Prof. Dr.-Ing. Kai Becher

## **Erklärung**

Wir versichern hiermit, dass wir unsere Studienarbeit mit dem Thema:

Konzeptionierung eines Simulators für einen 8-Bit Prozessor

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, 16. Mai 2022

---

Ort, Datum

Unterschrift

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>IV</b>
<b>Tabellenverzeichnis</b>	<b>V</b>
<b>Abkürzungsverzeichnis</b>	<b>VI</b>
<b>1 Einführung</b>	<b>1</b>
1.1 Ziel der Arbeit . . . . .	1
1.2 Repository . . . . .	1
1.3 Theoretische Grundlagen . . . . .	2
1.3.1 Architektur eines Prozessors . . . . .	2
1.3.2 Befehlsformate . . . . .	2
1.3.3 Harvard- und von-Neumann-Architektur . . . . .	3
1.3.4 CISC und RISC . . . . .	4
1.3.5 Parallelität nach Flynn . . . . .	5
<b>2 Projektplanung und Verlauf</b>	<b>7</b>
<b>3 Eigenschaften des Intel 8080</b>	<b>9</b>
3.1 Register . . . . .	9
3.2 Befehle . . . . .	9
3.3 OP-Code Kodierung . . . . .	10
3.4 Befehlsgruppen . . . . .	11
3.5 Typen der Maschinenzyklen . . . . .	14
3.6 Interrupts . . . . .	14
<b>4 Umsetzung</b>	<b>16</b>
4.1 Abstraktion der Architektur . . . . .	16
4.1.1 Prozessor . . . . .	16
4.1.2 ALU . . . . .	16
4.1.3 Register . . . . .	16
4.1.4 Peripherie . . . . .	17
4.2 Möglichkeiten der Simulation . . . . .	17
4.3 Aufbau der GUI . . . . .	17
4.3.1 Hauptmenü . . . . .	17
4.3.2 Intel 8080-Simulationsfenster . . . . .	18
4.4 Tutorial . . . . .	26

<b>5 Implementierung</b>	<b>27</b>
5.1 Multi-Prozessor-Simulator . . . . .	27
5.1.1 Programmablauf . . . . .	27
5.1.2 Dekodieren eines Befehls . . . . .	28
5.1.3 Programmaufbau . . . . .	29
5.1.4 Befehlsvarianten . . . . .	31
5.1.5 Binäre Addition und Subtraktion . . . . .	32
5.2 Single-Prozessor-Simulator . . . . .	33
5.2.1 Aufbau eines Befehls . . . . .	33
5.2.2 Programmablauf . . . . .	34
5.2.3 Dekodieren eines Befehls . . . . .	35
5.2.4 Programmaufbau . . . . .	35
5.2.5 Testing . . . . .	37
5.2.6 Befehlsvarianten . . . . .	38
5.2.7 Interrupts . . . . .	40
5.2.8 Besonderheiten des Simulators . . . . .	41
5.3 Laden und Anpassen der Oberfläche . . . . .	43
5.4 GUI-Methoden . . . . .	46
5.4.1 Aktualisieren der Oberfläche . . . . .	46
5.4.2 Manipulieren der Simulation . . . . .	50
5.4.3 Sonstige Interaktionen . . . . .	52
5.5 Exe-Erstellung . . . . .	53
<b>6 Fazit und Ausblick</b>	<b>54</b>
6.1 Zusammenfassung . . . . .	54
6.2 Ausblick . . . . .	54
<b>Literatur</b>	<b>55</b>
<b>Anhang</b>	<b>57</b>

# Abbildungsverzeichnis

1	Kodierung der Register [7] . . . . .	9
2	Hauptmenü . . . . .	18
3	Intel 8080 Simulationsfenster . . . . .	19
4	File Menüpunkt . . . . .	19
5	Settings Menüpunkt . . . . .	20
6	Programmeditor . . . . .	20
7	Programmtabelle . . . . .	21
8	Breakpoint . . . . .	21
9	Programmspeicher . . . . .	22
10	Range Einstellungen . . . . .	22
11	Bedien-Knöpfe zur Ausführung des Programmes . . . . .	23
12	Ausgewählte besondere Register . . . . .	23
13	Simulator Register und Darstellung . . . . .	24
14	Bereich zum Senden eines Interrupts . . . . .	24
15	Zustands-Log für aufeinanderfolgende NOP-Befehle . . . . .	25
16	Adresspuffer . . . . .	25
17	Tutorial-Fenster . . . . .	26
18	Funktion für das Lesen und Dekodieren eines Bytes . . . . .	28
19	Implementierung des Lesens eines Bytes . . . . .	29
20	Implementierung des DCX-Befehls . . . . .	30
21	Register Array des Intel 8080 . . . . .	30
22	Zugrundeliegende Implementierung von SBB und SBI . . . . .	31
23	Implementierung der Sprungbefehle . . . . .	32
24	Implementierung der binären Addition . . . . .	33
25	Implementierung der Dekodierung . . . . .	35
26	DCX-Befehl . . . . .	36
27	DCX-Maschinenzyklus . . . . .	36
28	DCX (rp_decr) Zustand . . . . .	36
29	Test für den ACI-Befehl . . . . .	37
30	einer von mehreren Tests für den ADC-Befehl . . . . .	38
31	Kodierung der Befehlstypen beim SYNC Takt [11] . . . . .	41
32	Rohes Simulator-Fenster . . . . .	44
33	Init-Functioncalls in Instanzfunktion für Intel 8080 Fenster . . . . .	44
34	Generische Funktion zum Laden der „ui“-Datei . . . . .	45
35	Initialisierung des Programmspeichers . . . . .	45
36	update_ui Funktion . . . . .	47
37	Programmschritte Instruction/Cycle/State . . . . .	47

38	Special Register Refresh . . . . .	48
39	Program Table Function . . . . .	49
40	Dictionary Ausschnitt . . . . .	49
41	Mask-Validation-Function . . . . .	50
42	Memory-Update-Function . . . . .	51
43	Knopfdruckfunktion . . . . .	51
44	Input-Validation-Function . . . . .	52
45	Range-Funktion . . . . .	52

## Tabellenverzeichnis

1	OP-Code-Typen [7] . . . . .	11
2	Befehle mit drei Befehls-Typen . . . . .	39
3	Maschinenzyklen der unterschiedlichen Typen . . . . .	40

## Abkürzungsverzeichnis

**ALU** - Arithmetic Logic Unit

**CISC** - Complex Instruction Set Computer

**GUI** - Graphical User Interface

**MISD** - Multiple-instruction stream single-data stream

**MIMD** - Multiple-instruction stream multiple-data stream

**RISC** - Reduced Instruction Set Computer

**SISD** - Single-instruction stream single-data stream

**SIMD** - Single-instruction stream multiple-data stream

# 1 Einführung

Von Nico

Dieses Kapitel befasst sich vorwiegend mit relevanten Grundlagen der Arbeit. Unter anderem wird das Ziel spezifiziert, elementare Aspekte der Arbeitsweise eines Prozessors werden erläutert und die verschiedenen Werkzeuge, mit denen das Ziel realisiert wird, werden aufgeführt.

## 1.1 Ziel der Arbeit

In dieser Arbeit soll ein Simulationsprogramm geschrieben werden, mit dem einer oder mehrere 8-Bit Prozessoren simuliert werden können. Dazu sollen die grundlegenden Eigenschaften und Funktionen in einer kurzen Einweisung erläutert werden. Mit dem Simulator soll es möglich sein die verschiedenen Eigenschaften wie das Programm oder Registerinhalte einzusehen und ändern zu können.

## 1.2 Repository

Der Quellcode kann in folgendem GitHub-Repository abgerufen werden:

<https://github.com/NicoSchrodt/8-Bit-Simulator>

## 1.3 Theoretische Grundlagen

Von Nico

Als Vorbereitung für die Implementierung der verschiedenen Prozessoren werden einige allgemeingültige Architekturprinzipien eines Prozessors erläutert.

### 1.3.1 Architektur eines Prozessors

Der fundamentale Aufbau eines Prozessors lässt sich in folgende Bausteine einteilen:

#### Rechenwerk

Das Rechenwerk ist die zentrale Einheit, mit der eingehende Befehle verarbeitet werden. Es erhält Werte aus dem Speicher und führt damit in der Arithmetisch-logische Einheit (ALU) Operationen durch. Zum Rechenwerk dazugehörig sind auch Hilfsregister, die beispielsweise als naher Zwischenspeicher fungieren.

#### Steuerwerk

Das Steuerwerk ist für die korrekte Abarbeitung von Befehlen zuständig. Es besteht aus dem Befehlsdekoder, dem Befehlszähler und einem Statusregister.

#### Programmspeicher

Der Programmspeicher eines Prozessors enthält die einzelnen Befehle, welche vom Befehlsdekoder dekodiert werden. Dabei wird der Befehl verwendet, der an der vom Befehlszähler spezifizierten Stelle im Speicher steht.

#### Ein-/Ausgabewerk

Das Ein-/Ausgabewerk ist für die Kommunikation des Prozessors mit anderen Systemkomponenten verantwortlich.

### 1.3.2 Befehlsformate

Für einen Prozessor wird zwischen vier verschiedenen Befehlsformaten unterschieden. Diese beziehen sich auf die Anzahl der Adressen, welche der ALU bei Beginn einer Operation zur Verfügung gestellt werden. [1]

#### Null-Adress-Anweisungen

Vom Akkumulator (Stack) wird der Wert der zwei obersten Adressfeldern entnommen und die Operation wird auf diese ausgeführt. Der Speicherort des Ergebnisses ist ebenfalls vorbestimmt. Daher werden zum Ausführen von Operationen keine Adres-

sen benötigt.

### **Ein-Adress-Anweisungen**

Wie bei einer Null-Adress-Anweisung wird ein Wert aus dem Akkumulator entnommen. Ein zweiter Wert wird aus dem Speicher der übergebenen Adresse entnommen. Der Speicherort des Ergebnisses ist nach wie vor fest.

### **Zwei-Adress-Anweisungen**

Für Zwei-Adress-Anweisungen ist es möglich eine Bezugsadresse für die Operanden anzugeben sowie eine Zieladresse, in der das Ergebnis der Operation gespeichert wird.

### **Drei-Adress-Anweisungen**

Drei-Adress-Anweisungen sind essentiell komplexere Zwei-Adress-Anweisungen. Eine der drei verfügbaren Adressen wird ebenfalls für die Zieladresse der Operation verwendet. Für das Beziehen der Operanden steht eine weitere Adresse zur Verfügung.

#### **1.3.3 Harvard- und von-Neumann-Architektur**

Von Andreas

Prozessoren können auf viele unterschiedliche Arten kategorisiert werden. In diesem Kapitel geht es um die Harvard- und von-Neumann-Architektur. Für beide Architekturen werden die im vorherigen Kapitel beschriebenen Bauteile benötigt. Bei der von-Neumann-Architektur gibt es nur einen Daten-, Steuer- und Adressbus, der mit allem verbunden ist. Bei einem neuen Befehl muss somit zuerst ein Befehl vom Speicher in das Steuerwerk geladen werden. Erst danach wird mit Steuersignalen den einzelnen Komponenten mitgeteilt was als nächstes geschehen soll. Häufig müssen daraufhin Daten geladen oder geschrieben werden. Damit der Speicher aber weiß woher die Daten gelesen werden müssen, muss zuerst die Adresse über den Bus geschickt werden. Dadurch, dass es bei der von-Neumann-Architektur nur einen Bus gibt, muss darüber die ganze Kommunikation ablaufen.

Die Harvard-Architektur besitzt mehrere, getrennte Busse, da es auch zwei getrennte Speicher gibt: Einen Programmspeicher und einen Datenspeicher. Bei der von-Neumann-Architektur gibt es nur einen Speicher. Deshalb müssen Programmcode und Daten zwangsweise zusammen abgelegt werden. Dadurch, dass bei der Harvard-Architektur zwei getrennte Speicher verwendet werden und jeder einen eigenen Bus besitzt, können sich Befehle und Daten nicht gegenseitig blockieren. Somit kön-

nen beide Speicher parallel angesprochen und verwendet werden. Während für den vorherigen Befehl noch Daten eingelesen werden, kann schon ein neuer Befehl aus dem Programmspeicher eingelesen werden. Daraus ergibt sich, rein aus der Theorie schon, eine höhere Ausführungsgeschwindigkeit als bei von-Neumann-Systemen. Der Nachteil davon ist allerdings, dass Bauteile öfters verwendet werden, was dazu führt, dass diese Prozessoren teurer sind. Deshalb sind heutzutage auch die meisten Prozessoren noch mit einer von-Neumann-Architektur. [2], [3]

### 1.3.4 CISC und RISC

Von Andreas

Ein Prozessor unterstützt immer nur eine gewisse Menge an Befehlen, dies wird Instruction Set genannt. Heutzutage gibt es zwei grundlegende Prozessorarchitekturen: Complex Instruction Set Computer (CISC) und Reduced Instruction Set Computer (RISC). Früher konnten Prozessoren genau einer dieser Gruppen zugeordnet werden, allerdings ist das bei den heutigen Prozessoren nicht mehr möglich, da sowohl RISC- als auch CISC-Befehle dem Prozessor zur Verfügung gestellt werden, um die Vorteile von beiden zu nutzen.

Bei CISC-Prozessoren wird versucht, soviel Arbeit wie möglich in einem Befehl ausführen zu können. So gibt es viele verschiedene Befehle, die auch unterschiedlich viel Zeit benötigen. Dadurch wird es aber auch möglich, komplexere Befehle direkt in der Hardware zu berechnen. Bei diesen Befehlen gibt es auch einige Adressierungsarten mehr als bei RISC-Prozessoren. Für vorbestimmte Aufgaben gibt es auch eigene Register, die nur dafür verwendet werden, und davon auch nur wenige. Ein Nachteil der CISC-Prozessoren ist, dass die eigenen Befehle erst noch durch ein Mikroprogramm interpretiert werden müssen und dieses die komplexen Befehle in mehrere kleine Befehle aufteilen muss, welche erst dann vom Prozessor bearbeitet werden können. Dies kostet etwas mehr Zeit und verlangsamt die Ausführung. Die Mikroprogramme, die dafür verwendet werden, werden in einem kleinen Read-only Memory (ROM) gespeichert.

Bei RISC-Prozessoren wird versucht mit nur wenigen, kleinen Befehlen auszukommen. Diese sind wiederum sehr schnell, da sie meist fest verdrahtet sind, müssen aber mit anderen kombiniert werden um die Komplexität eines einzigen CISC-Befehls zu erreichen. Im Gegensatz zu CISC-Prozessoren besitzen RISC-Prozessoren viele Register, die frei verwendbar sind und nicht für spezielle Operationen bestimmt sind. Ebenso können die meisten Befehle in nur einem einzigen Arbeitsschritt ausgeführt werden. [4]

Da bei CISC-Prozessoren mit nur einem Befehl viel berechnet werden kann, sind diese optimal für Übersetzer oder Interpreter geeignet. Bei der Entwicklung können einzelne komplexe Befehle anstatt vieler kleiner verwendet werden. Jedoch können RISC-Prozessoren schneller Befehle ausführen, da:

- es nur wenige Befehle gibt und diese schnell dekodiert werden können.
- die Befehle mithilfe von Pipelines effizienter abgearbeitet werden können.
- kein Mikroprogramm die einzelnen Befehle erst noch interpretieren muss.

### 1.3.5 Parallelität nach Flynn

Von Andreas

Von Michael Flynn wurden 1966 die folgenden vier Arten von Parallelisierung eingeführt: [5]

- Single-instruction stream, single-data stream (SISD)
- Single-instruction stream, multiple-data stream (SIMD)
- Multiple-instruction stream, single-data stream (MISD)
- Multiple-instruction stream, multiple-data stream (MIMD)

#### SISD

Diese Beschreibung trifft auf die einfachen Einprozessorsysteme zu. Dabei kann immer nur eine Operation gleichzeitig ausgeführt werden und diese werden in nur einer möglichen Reihenfolge aus einem Datenstrom abgearbeitet.

#### SIMD

Bei SIMD werden Pipelines eingesetzt, die es ermöglichen, mehrere korrekte Abfolgen von Programmbefehlen auszuführen. So können unterschiedliche Programme in sich selbst in der richtigen Reihenfolge, aber mit anderen Programmen abwechselnd ausgeführt werden.

#### MISD

Diese Variante scheint auf Anhieb keinen effizienten Nutzen zu besitzen, da mehrere Prozessoren alle die gleichen Befehle ausführen, die aus einem Daten-Strom stammen. Dies kann aber dazu verwendet werden, die Korrektheit durch Redundanzen

zu bestätigen.

### MIMD

Diese Architektur verwendet mehrere Prozessoren und mehrere Daten-Ströme. Heutzutage ist dies unter dem Begriff Mehrprozessorsysteme bekannt. So werden für jeden einzelnen Prozessor ein Datenstrom erzeugt, der unabhängig von den anderen Prozessoren arbeiten kann. Dabei ist es für die Prozessoren aber trotzdem möglich, die Daten der anderen Prozessoren zu nutzen. Nur durch MIMD oder MISD, also die Ausführung mit mehreren unabhängigen Prozessoren, ist es möglich, Programme tatsächlich parallel ablaufen zu lassen. Mit SISD oder SIMD sind nur quasi-parallele Ausführungen möglich.

## 2 Projektplanung und Verlauf

Von Andreas

Bei Projektbeginn wurde kein konkretes Ziel vorgegeben. Es wurde nur verlangt, dass ein Simulator für 8-Bit Prozessoren entwickelt werden soll. Deshalb wurde zu Beginn die Entscheidung getroffen, dass eine kleine Lernsoftware entwickelt werden soll. Der Simulator sollte ursprünglich fähig sein, mehrere unterschiedliche Prozessoren simulieren zu können. Damit sollten die Unterschiede verschiedener Prozessoren dargestellt und dem Anwender vermittelt werden. Jedoch hat sich bei der Untersuchung von mehreren Prozessoren herausgestellt, dass es nur geringe Unterschiede zwischen den Prozessoren gibt. Zwar hat jeder Prozessor seine Eigenheiten, die aber für den Anwender kaum bemerkbar sind. Der größte Unterschied sind die verschiedenen Befehlssätze. Die meisten Prozessoren verwenden für die gleichen Befehle unterschiedliche Mnemonics und Bitkombinationen. Sobald aber die Befehle dekodiert wurden, läuft im Prozessor meistens sehr Ähnliches oder sogar Gleiches ab. Dabei kann es vorkommen, dass ein Prozessor noch ein zusätzliches Register z.B. als Zwischenspeicher verwendet, welches die anderen nicht besitzen. Aber wie die Daten im Prozessor hin- und her geschoben werden und die Befehle, die die ALU ausführt, unterscheidet sich zwischen den Prozessoren kaum oder ist sogar identisch. Deshalb wurde sich später dafür entschieden, dass nicht mehrere, sondern nur ein Prozessor simuliert werden soll.

Bei dem Multi-Prozessor-Simulator war geplant, dass nur die Befehle direkt ausführbar sein sollen. Sodass eine assemblierte Quellcode-Datei eingelesen und ausgeführt werden kann. Diese Implementierung hat sichergestellt, dass die zwangsweise notwendigen Eigenschaften simuliert werden. Dazu zählt, dass die Inhalte von Registern, die vom Anwender verwendet werden können, nach der Ausführung der Befehlen die korrekten Werte enthalten. Zusätzlich dazu wurden auch die Flags simuliert, da diese für den korrekten Programmablauf notwendig sind.

Der Single-Prozessor-Simulator sollte etwas genauer darstellen, wie ein Prozessor arbeitet. Dafür wurde der Intel 8080 als Prozessor ausgewählt. Ein Intel 8080-Befehl besteht aus mindestens einem und maximal fünf bzw. in diesem Simulator sechs Maschinencyklen. Jeder dieser Maschinencyklen wiederum aus drei bis fünf Zuständen. Ein kompletter Befehl besteht, bei dem Intel 8080, insgesamt aus mindestens 4 und maximal 18 Zuständen. Mit dieser Implementierung kann jeder einzelne Zustand nachvollzogen werden. Dadurch werden auch die W- und Z-Register verwendet, auf die der Anwender normalerweise keinen Zugriff hat und deshalb beim Multi-

Prozessor-Simulator nicht beachtet wurden.

Die Aufgabengebiete bei diesem Simulator wurden wie folgt verteilt: Nico Schrödt befasste sich mit dem Frontend, also die graphische Oberfläche, und Andreas Schmidler mit dem Backend, also die Funktionsweise des Prozessors. Diese zwei Bereiche können gut voneinander getrennt werden. Somit musste sich nur auf wenige Schnittstellen geeinigt werden.

### 3 Eigenschaften des Intel 8080

Von Andreas

Als Prozessor für den Single-Prozessor-Simulator wurde der Intel 8080 ausgewählt. Der Intel 8080 besitzt eine von-Neumann-Architektur und gehört somit zur Klasse der SISD-Architektur. Er bietet ein breites Spektrum an Befehlen und ist repräsentativ für einen der ersten großen, kommerziell erfolgreichen Prozessoren. Im Folgenden werden die Befehle und Register genauer erklärt.

#### 3.1 Register

Der Intel 8080 besitzt ein SRAM-Array mit 16-Bit-Registern. Darin enthalten sind der Programmzähler mit 16 Bit und der Stack Pointer mit 16-Bit. Dazu gibt es noch acht weitere 8-Bit Register. Diese können entweder alleine oder zusammen mit einem anderen 8-Bit Register als ein 16-Bit Register verwendet werden. Dabei gibt es aber nur die fest vorgegebenen Kombinationen. Das B- und C-Register, das D- und E-Register, das H- und L- Register und die temporären W- und Z- Register. Die W- und Z-Register können nicht vom Programmierer verwendet werden und dienen nur zur internen Ausführung von Befehlen. Bei manchen Befehlen wird der Stackpointer als 16-Bit Register verwendet. Über einen Multiplexer ist es möglich, 8 Bit auf den internen Adressbus zu schreiben oder von dort zu lesen. Über das Adress-Latch des Inkrementer/Dekrementer-Chipsatzes ist es möglich 16 Bit aus den Registern an den Adress-Puffer weiterzuleiten. [6]

SSS or DDD	Value	rp	Value
A	111	B	00
B	000	D	01
C	001	H	10
D	010	SP	11
E	011		
H	100		
L	101		

Abbildung 1: Kodierung der Register [7]

#### 3.2 Befehle

Der Intel 8080 ist in der Lage, aus einem, zwei oder drei Bytes bestehen Befehle auszuführen. Dabei gibt das erste Byte immer den Operation Code an. In Byte zwei und drei werden nur Daten oder Adressen gespeichert. Dabei werden die zwei Byte

großen Adressen so gespeichert, dass das niederwertige Byte vor dem höherwertigem gespeichert wird. Die Adressen können dabei über vier verschiedene Modi verwendet werden.

- Direct
- Register
- Register Indirect
- Immediate

Bei „Direct“ wird der Wert im Speicher mit der angegebenen Adresse verwendet. Hier werden das Low-Byte im zweiten und das High-Byte im dritten Byte gespeichert. Bei „Register“ wird auf ein oder zwei Register verwiesen und der Register-Typ verhält sich wie bei „Direct“. Bei „Register Indirect“ wird der Wert aus der Adresse aus dem zweiten und dritten Byte des Befehls gelesen. Dieser Wert wird als Adresse verarbeitet und erst der Wert aus dieser Adresse ist der zu verwendende Wert. Bei „Immediate“ steht im zweiten und/oder dritten Byte ein Wert, mit dem gearbeitet wird (Low-Byte im zweiten Byte). Bei Interrupts und Verzweigungs-Befehlen gibt es nur die „Direct“- und „Register indirect“-Modi. [8]

Der Prozessor besitzt fünf Condition-Flags. Das Zero-Flag gibt an ob das Ergebnis eines Befehls den Wert 0 hatte. Das Sign-Flag gibt an, ob Bit 8, das Most Significant-Bit, des letzten Ergebnisses den Wert 1 hat. Wenn das letzte Ergebnis gerade war, ist das Paritäts-Flag gesetzt. Das Carry-Flag zeigt an ob es einen Übertrag bei einer Addition oder einen Abzug bei einer Subtraktion oder einem Vergleich gab. Das Auxiliary Carry-Flag zeigt ebenfalls einen Übertrag oder Abzug an, aber zwischen Bit 3 und Bit 4. [8]

### 3.3 OP-Code Kodierung

Jedem Befehl ist ein Byte (OP-Code) zugewiesen. Dabei gibt es mehrere Typen. In manchen Befehlen werden einzelne Bits verwendet, um Register oder Registerpaare zu kodieren, während in anderen jedes Bit verwendet wird. In Tabelle 1 sind die verschiedenen Möglichkeiten aufgelistet. Bei dem Befehl ACI ist es wichtig, dass das Byte genau 0xC7 ist. Bei ADC und INR wird ein 8-Bit Register in den Befehl kodiert. Hierbei wird unterschieden, ob Daten in das Register geschrieben (DDD für Destination) oder daraus gelesen (SSS für Source) werden. Bei Mov r1, r2 wird sowohl aus einem Register gelesen als auch in ein anderes geschrieben, weshalb

nur noch zwei Bit für den eigentlichen MOV-Befehl übrig bleiben. Bei dem INX-Befehl wird ein 16-Bit Register verwendet. Wie auf Seite 9 beschrieben, können auch zwei 8-Bit-Register zu einem 16-Bit-Register zusammen geschlossen und damit verwendet werden. Insgesamt gibt es davon drei Kombinationen. Dementsprechend werden in diesen Befehlen diese Register mit zwei Bit kodiert. Damit gibt es noch eine weitere Bitkombination, mit der ein weiteres Register verwendet werden kann. Mit der vierten noch übrig gebliebenen Kombination wird der Stackpointer als 16-Bit-Register verwendet.

Befehl	OP-Code
ACI	1100 1110
ADC r	1000 1SSS
INR r	00DD D101
MOV r1, r2	01DD DSSS
INX rp	00RP 0011

Tabelle 1: OP-Code-Typen [7]

### 3.4 Befehlsgruppen

Jeder Befehl kann genau einer der folgenden fünf Gruppen zugeordnet werden. Jede Gruppe verhält sich anders mit den Status-Flags. So gibt es eine Gruppe, bei der die meisten Befehle alle Flags ändern, aber auch Gruppen, wobei die meisten Befehle kein einziges Flag verändern.

#### Daten-Transfer-Gruppe

Für den Prozessor sind 13 Befehle aus dieser Gruppe bekannt. Bei keinem dieser Befehle werden die Condition Flags gesetzt oder zurückgesetzt. Alle Befehle, die dieser Gruppe angehören, schreiben ausschließlich Daten in Register oder in den Programmspeicher. Es werden keine Rechenoperationen oder Sprünge ausgeführt. Zu dieser Gruppe gehören die folgenden Befehle:

- Move Register
- Move from Memory
- Move to Memory
- Move immediate
- Move to Memory Immediate
- Load register pair immediate
- Load Accumulator direct
- Store Accumulator direct
- Load H and L direct
- Store H and L direct
- Load Acumulator indirect
- Store Accumulator indirect
- Exchange H and L with D and E

### Arithmetikgruppe

Dieser Gruppe gehören 20 Befehle an. Die meisten dieser Befehle beeinflussen alle Statusflags. Ausnahmen bilden dabei der INR- und DCR-Befehl, der INX- und DCX-Befehl und der DAD-Befehl. Der INR- und DCR-Befehl beeinflussen alle Flags außer das Carry-Flag. Der DAD-Befehl verändert ausschließlich das Carry-Flag. Keine Flags werden bei dem INX- und DCX-Befehl beeinflusst. Zu dieser Gruppe gehören die Befehle, die arithmetische Operationen ausführen, also z.B. Additionen oder Subtraktionen.

- Add Register
- Add Memory
- Add immediate
- Add Register with carry
- Add Memory with carry
- Add immediate with carry
- Subtract Register
- Subtract Memory
- Subtract immediate
- Subtract Register with borrow
- Subtract Memory with borrow
- Subtract immediate with borrow
- Increment Register
- Increment Memory
- Decrement Register
- Decrement Memory
- Increment register pair
- Decrement register pair
- Add register pair to H and L
- Decimal Adjust Accumulator

### Logische Gruppe

Aus dieser Gruppe sind 19 Befehle bekannt. Diese Gruppe enthält Befehle, die logische Operationen wie UND und ODER ausführen. Bis auf die Rotations-, die Komplement- und den Set-Carry-Befehl werden immer alle Flags beeinflusst.

- AND Register
- AND Memory
- AND immediate
- Exclusive OR Register
- Exclusive OR Memory
- Exclusive OR immediate
- OR Register
- OR Memory
- OR immediate
- Compare Register
- Compare Memory
- Compare immediate
- Rotate left
- Rotate right

- Rotate left through Carry
- Rotate right through Carry
- Complement Accumulator
- Complement Carry
- Set Carry

#### Verzweigungsgruppe

Zu dieser Gruppe werden die Befehle dazu gezählt, die den standardmäßigen Programmablauf verändern. Keiner dieser Befehle verändert ein Flag. Der Intel 8080 kennt eigentlich nur einen Conditional-Befehl pro Sprung/Call etc. Für den Assembler können aber eigene Mnemonics für die unterschiedlichen Flags verwendet werden.

- Jump
- Conditional Jump
- Call
- Conditional Call
- Return
- Conditional Return
- Restart
- Jump H and L indirect - move H and L to PC

#### Stapel-, Ein/Ausgangs-, und Maschinenkontrollgruppe

Die Befehle, die dieser Gruppe angehören, verändern entweder den Stack, interne Status-Flags oder sind für Ein- und Ausgaben zuständig. Zusätzlich dazu wird auch der NOP-Befehl (No Operation) dazu gezählt, obwohl er keines der Kriterien erfüllt. Letztendlich kann der NOP-Befehl keiner Gruppe genau zugewiesen werden, da dieser keine Funktion besitzt. In dieser Gruppe verändert nur der POP-Befehl die Flags.

- Push
- Input
- Push Processor status word
- Output
- Pop
- Enable Interrupts
- Pop processor status word
- Disable Interrupts
- Exchange stack top with H and L
- Halt
- Move HL to SP
- No op

### 3.5 Typen der Maschinenzyklen

Jeder Befehl oder Befehlszyklus benötigt mindestens einen und maximal fünf Maschinenzyklen zum kompletten Ausführen des Befehls. Dabei kann jeder Maschinenzyklus genau einem Typ zugeordnet werden. Die existierenden Befehlstypen sind:

- Fetch
- Memory Read
- Memory Write
- Stack Read
- Stack Write
- Input
- Output
- Interrupt
- Halt
- Halt Interrupt

Der erste Maschinenzyklus ist immer ein Fetch-Zyklus. Bei diesem wird immer ein neuer Befehl eingelesen und dekodiert. Für alle anderen Maschinenzyklenphasen kann nicht mehr genau gesagt werden zu welchem Typ diese gehören. Deshalb wird in dem ersten Zustand jedes Maschinenzyklus immer der Befehlstyp während des SYNC-Taktes auf den Datenbus kodiert. [9]

### 3.6 Interrupts

Der Intel 8080 ist in der Lage Interrupts auszuführen, die den normalen Programmfluss unterbrechen. Dies ist aber nur zwischen Befehlen möglich. Wenn ein Interrupt eintritt, während ein Befehl noch ausgeführt wird, wird dieser zuerst bis zum Schluss noch ausgeführt. Eine Besonderheit der Interrupts ist, dass jeder Interrupt ein Byte mitliefert, das angibt, was der Interrupt ausführen soll. Dabei ist jeder Befehl erlaubt, der nur ein Byte benötigt. Somit kann auch nur ein Befehl ausgeführt werden. Damit aber auch größere Interrupt Service Routines möglich sind gibt es den RST-Befehl. Dieser Befehl schreibt den aktuellen Programmzähler auf den Stack und springt an eine von acht Stellen im Programmspeicher. Welche Stelle genau wird im RST-Befehl dekodiert. Diese acht Sprungpunkte liegen alle genau acht Byte auseinander. Falls dies aber für den Interrupt nicht ausreicht, kann von dort an

eine andere Stelle im Programmspeicher gesprungen werden. Da der Intel 8080 eine von-Neumann-Architektur verwendet und der Stack an einer beliebigen Stelle im Programmspeicher (64kB) liegen kann, gibt es so gut wie keine Größenlimitierung dafür. Bei manchen anderen Prozessoren kann der Stack z.B. maximal acht Einträge besitzen. Für den Intel 8080 wurde es ermöglicht, dass Interrupts durch andere Interrupts unterbrochen werden können. Durch den RST-Befehl wird der momentane Programmzähler immer auf den Stack geschrieben. Somit kann immer der neueste Interrupt ausgeführt werden und kann dann zum zuvor ausgelösten Interrupt zurückspringen.

## 4 Umsetzung

Von Nico

In diesem Kapitel wird darauf eingegangen, wie die in Kapitel 2 erarbeitete Analyse der ausgewählten Prozessoren implementiert wird.

### 4.1 Abstraktion der Architektur

Der Prozessor wird in verschiedene Unterklassen aufgeteilt. Dabei ist vorgesehen, dass dies in einer Art Sternstruktur aufgebaut wird, das heißt die verschiedenen Bestandteile des Prozessors sollen alle mit einer zentralen Klasse interagieren, aber nicht untereinander.

#### 4.1.1 Prozessor

Die Prozessor-Klasse soll die zentrale Einheit des Simulators sein. Über diese sollen sowohl Anfragen über Informationen vorgenommen werden, zum Beispiel um diese in der Benutzeroberfläche anzuzeigen, als auch Instruktionen an den Prozessor als ganzes gesendet werden. Dabei ist zwischen zwei Arten von Anfragen zu unterscheiden, gültigen und ungültigen. Gemeint sind damit Anfragen die in einem echten Prozessor zulässig wären, beispielsweise einen externen Bus zu befüllen oder die nächste Anweisung auszuführen, und Anfragen, die unzulässig wären, wie den Programmzähler manuell zu setzen.

#### 4.1.2 ALU

Die ALU soll die Bearbeitung logischer Operationen im Prozessor simulieren, um ein möglichst genaues Bild jedes Zustandes des Prozessors darzustellen, statt beispielsweise solche Operationen manuell in der Prozessor-Klasse abzuarbeiten. Im Verlauf des Projektes wurde die Entscheidung getroffen, das Ausführen der verschiedenen Operationen eher zu emulieren als zu simulieren. Um eine entsprechende Genauigkeit zu gewährleisten, werden deshalb nicht ganze Instruktionen, sondern die einzelnen Maschinenzyklen und Zustände ausgeführt.

#### 4.1.3 Register

Die Register-Klasse soll ähnlich wie die ALU den Zugriff auf Register möglichst realitätsgerecht simulieren. Dabei sollen Methoden, die den Zugriff regeln, ähnlich aufgebaut werden wie es im echten Prozessor möglich wäre.

#### 4.1.4 Peripherie

Mit der Peripherie soll alles abgedeckt werden, was nicht Teil der zentralen Bestandteile ist, also der Register oder der ALU. Dazu zählt zum Beispiel ein externer Datenbus oder Schnittstellen für Taktsignal o.Ä.

### 4.2 Möglichkeiten der Simulation

Um einen Ablauf für die Simulation festzulegen, muss zuerst bestimmt werden, welche Parameter eingestellt, welche Optionen verändert, und welche Operationen im Simulator durchgeführt werden können. Geplant ist, dass die Simulation zu jedem Zeitpunkt unterbrechbar sein soll. Der kleinste Zeitabschnitt, für den dies sinnvoll ist, wäre ein einzelner Zustand eines Maschinenzyklus. Es ist geplant, dass alle in der Register-Klasse modellierten Register auch in der Benutzeroberfläche dargestellt und manipuliert werden können. Zu manipulierbaren Registern soll auch der Programmspeicher zählen. Dieser soll entweder manuell oder über eine externe Datei befüllbar sein. Für das externe Befüllen muss der für den Simulator vorgegebene Assembler [10] verwendet werden. Die Manipulation des Speichers soll auch während der Simulation möglich sein.

### 4.3 Aufbau der GUI

Das Simulieren des Prozessors ist die ein Aspekt des Projekts. Gleichermassen wichtig für einen funktionierenden Simulator ist aber auch die Benutzeroberfläche, mit der dieser bedient wird. In diesem Abschnitt werden die einzelnen Aspekte dieser analysiert.

#### 4.3.1 Hauptmenü

Bei Programmstart wird das in Abbildung 2 gezeigte Fenster mit dem Hauptmenü geöffnet.

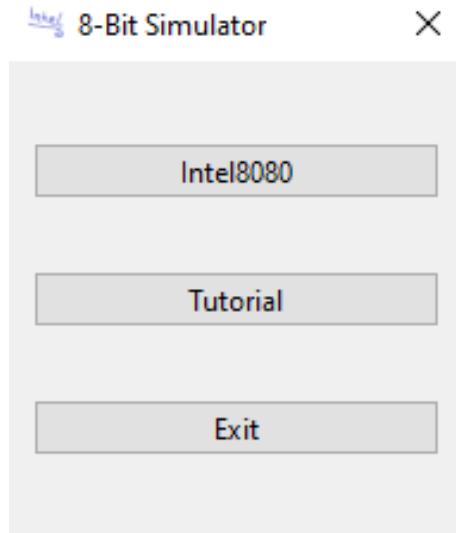


Abbildung 2: Hauptmenü

Die Aufgabe dieses Fensters ist simpel. Es gibt drei verschiedene Knöpfe, die jeweils eine Funktion erfüllen:

- Intel8080: Öffnet das Simulationsfenster des Intel 8080-Prozessors
- Tutorial: Öffnet ein Fenster mit einer kurzen Einweisung in die Handhabung des Simulators für den Intel 8080
- Exit: Schließt das Fenster und alle laufenden Hintergrundprozesse, für den Fall, dass ein Simulatorprozess in der selben Session verwendet wurde.

#### 4.3.2 Intel 8080-Simulationsfenster

In Abbildung 3 ist die aktuellste Version des Simulationsfensters für den Intel 8080 zu sehen.

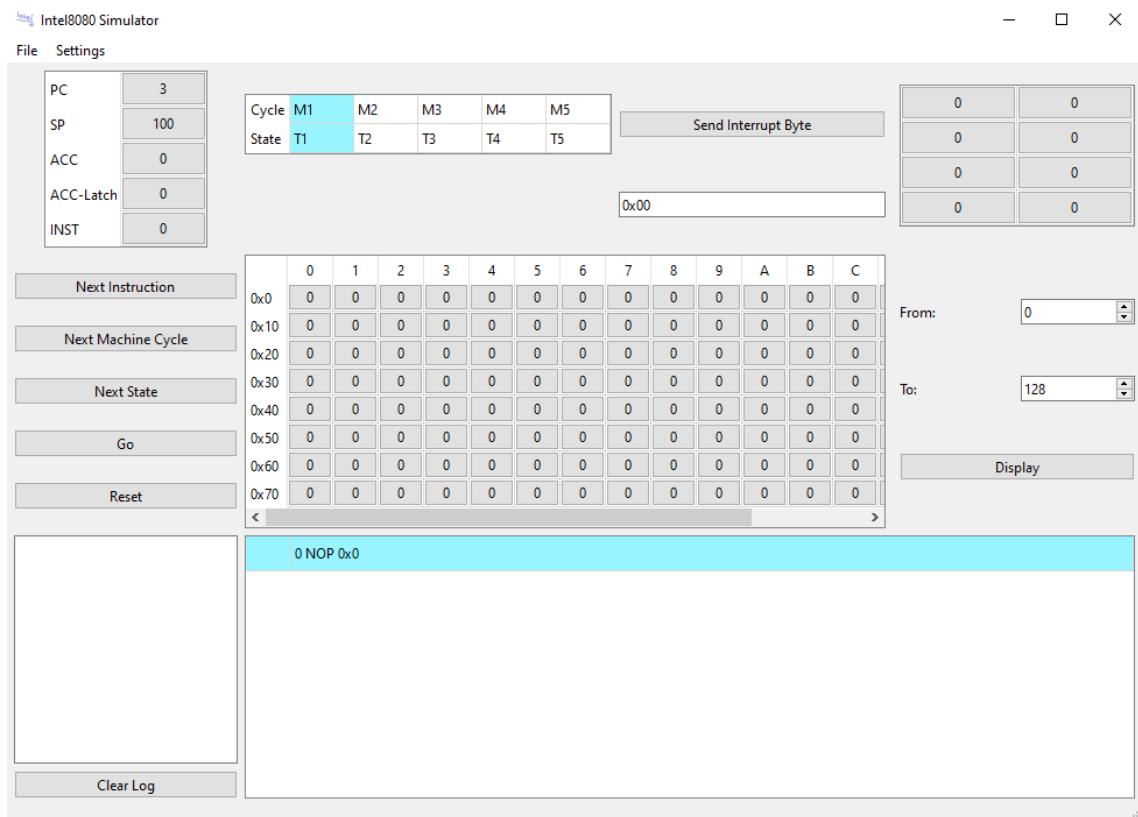


Abbildung 3: Intel 8080 Simulationsfenster

Über das Simulationsfenster für den Intel 8080 können Programme, die aus Bytebefehlen bestehen, welche ein Intel 8080 Prozessor ausführen kann, simuliert werden. Dafür sollte (muss aber nicht) zuerst ein gültiges Programm geladen werden. Dies kann über den Menüpunkt „File“ in der oberen Leiste getan werden (siehe Abbildung 4).

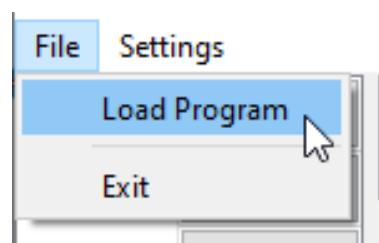


Abbildung 4: File Menüpunkt

Dadurch öffnet sich ein Dateien-Explorer, über den die gewünschte Datei ausgewählt werden kann. Dabei ist zu beachten, dass in der aktuellen Version nur Ausgabedateien des verwendeten Assemblers genutzt werden können (Dateiendung „.com“). Über „Exit“ kann der Benutzer ins Hauptmenü zurückkehren. Möchte man andere Programme als die mitgelieferten verwenden, so kann man den Editor der unter dem Settings Menüpunkt zu finden ist benutzen (siehe Abbildung 5).

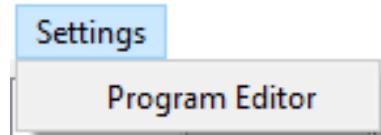


Abbildung 5: Settings Menüpunkt

Um die grobe Struktur eines Programms zu verstehen sind einige gültige Beispielbefehle bereits im Textfeld eingetragen (siehe Abbildung 6). Im Intel 8080 Programmers Manual sind noch weitere Beispiele für gültige Befehle aufgeführt. Das Programm kann über das untere Textfeld benannt werden. Wird der „Output File“-Knopf betätigt werden drei Dateien ausgegeben mit den Endungen „asm“, „sym“ und „com“. Abhängig davon ob das Programm über die .exe-Datei gestartet wurde wird entweder der „Output“-Ordner verwendet oder einfach das Verzeichnis in dem die Exe ausgeführt wurde. Liegt ein Fehler vor in der Syntax wird ein Fehler ausgegeben mit dem Problem das behoben werden muss.

A screenshot of the 'Program Editor' window. The main text area contains assembly code for a loop:

```
Loop:  
mvi b, 20  
mvi c, 30  
mov a, b  
add c
```

Below the code, there are two input fields: 'File Name:' containing 'program' and 'Output File'. The window has standard OS X-style window controls at the top right.

Abbildung 6: Programmeditor

Sobald ein Programm geladen wurde, wird dieses in der dafür vorgesehenen Tabelle angezeigt (siehe Abbildung 7).

0	MVI	0x6	0x14
2	MVI	0xe	0x1e
4	MOV	0x78	
5	ADD	0x81	
6	NOP	0x0	

Abbildung 7: Programmtabelle

Dabei ist für jeden Befehl eine Zeile vorgesehen. Angezeigt werden sowohl die Position des Bytebefehls im Speicher, die Bezeichnung des Befehls, der Bytecode selbst, als auch zusätzliche Parameter, falls der jeweilige Befehl diese besitzt. Außerdem wird die Zeile, die als nächstes ausgeführt wird farbig markiert. Abgesehen von der Anzeige des Programms und des als nächste auszuführenden Befehls, ist es auch möglich, Breakpoints zu setzen (siehe Abbildung 8).



Abbildung 8: Breakpoint

Diese werden für die automatische Ausführung des Programmes (siehe nächster Abschnitt) relevant. Wird ein Breakpoint erreicht, so stoppt das Programm, vor Ausführung des Befehls. Das Programm muss manuell fortgesetzt werden. Da vorgesehen ist, dass das gesamte Programm in dieser Tabelle angezeigt wird, ist eine Sonderbehandlung des NOP-Befehls notwendig. Dieser wird durch den Bytecode „0x00“ identifiziert. Bei Initialisierung des Speichers ist dies auch die Standardbelegung jedes Registers. Damit die Tabelle also nicht mit in den meisten Fällen funktionslosen NOP-Befehlen gefüllt wird, musste dafür eine Ausnahmeregelung getroffen werden. Sind im Speicher zwei oder mehr NOP-Befehle direkt aufeinanderfolgend, werden diese zu einem Eintrag zusammengefasst. Da diese Tabelle lediglich das Programm in komprimierter Weise darstellt aber keine Interaktion mit diesem ermöglicht, wurde zusätzlich das in Abbildung 9 gezeigte Fenster in die GUI eingefügt.

	0	1	2	3	4	5	6	7	8	9	A	B	C
0x0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	0	0	0	0	0	0	0	0	0	0	0	0	0
0x20	0	0	0	0	0	0	0	0	0	0	0	0	0
0x30	0	0	0	0	0	0	0	0	0	0	0	0	0
0x40	0	0	0	0	0	0	0	0	0	0	0	0	0
0x50	0	0	0	0	0	0	0	0	0	0	0	0	0
0x60	0	0	0	0	0	0	0	0	0	0	0	0	0
0x70	0	0	0	0	0	0	0	0	0	0	0	0	0

Abbildung 9: Programmspeicher

Dieses ermöglicht es, den Speicher, in Abschnitten von 128 Registern, auszulesen, wobei pro Zeile 16 Register zu sehen sind. Hierbei ist es im Gegensatz zur oben beschriebenen Tabelle möglich, diese zu bearbeiten. Dies geschieht durch klicken der entsprechenden Zelle, wodurch sich ein Dialog öffnet. Änderungen werden bei der nächsten Zustandsänderung („Next State/Machine Cycle/Instruction“) in die Programmtabelle übernommen. Um auszuwählen, welche Register aus dem Programmspeicher angezeigt werden, kann das in Abbildung 10 gezeigte Interface verwendet werden.

From:

To:

**Display**

Abbildung 10: Range Einstellungen

Hiermit kann der Start- oder Endwert des angezeigten Registerblocks eingestellt werden. Wird einer der beiden Werte verändert, ergänzt sich der andere automatisch, um den angezeigten Satz von 128 Registern konstant zu halten. Der Abstand muss dabei immer genau 128 Register sein. Eingetragene Werte müssen außerdem immer ein Vielfaches von sechzehn sein, da dies der Anzahl an Registern einer Zeile entspricht. Die Ausführung des Programms wird über fünf Knöpfe im Simulationsfenster gesteuert (siehe Abbildung 11).

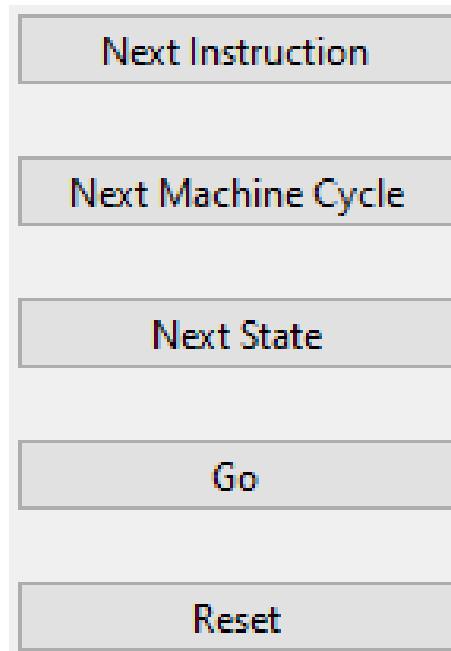


Abbildung 11: Bedien-Knöpfe zur Ausführung des Programmes

Über „Next Instruction“ wird der nächste Befehl, der in der Programmtabelle farbig hervorgehoben ist, ausgeführt und die Benutzeroberfläche wird entsprechend aktualisiert. Dasselbe gilt für „Next Machine Cycle“ und für „Next State“, wobei diese jeweils nur den nächsten Machinenzyklus oder den nächsten Zustand ausführen. Über „Go“ wird automatisch das Programm ausgeführt. Bei erneutem Drücken wird das Programm pausiert. „Reset“ setzt den Simulator auf die Ausgangssituation zurück, behält aber den aktuellen Zustand des Programmspeichers bei. Es ist auch möglich, einzelne Register des Prozessors auszulesen. Wie in Abbildung 12 zu sehen ist, werden diese mit ihren entsprechenden Bedeutungen und Werten angezeigt.

PC	0
SP	48
ACC	0
ACC-Latch	0
INST	0

Abbildung 12: Ausgewählte besondere Register

Die gezeigten Einträge stehen für:

- PC: Program Counter Register
- SP: Stack Pointer Register
- ACC: Accumulator Register
- ACC-Latch: Accumulator-Latch Register
- INST: Instruction Register

Durch Klicken auf die entsprechende Werte lassen sich diese in einem neuen Fenster bearbeiten. Das in Abbildung 13 gezeigte Modul zeigt die restlichen Register an.

0	0
0	0
0	0
0	0

(a) Simulator generische Register

W TEMP REG. (8)	Z TEMP REG. (8)
B REG. (8)	C REG. (8)
D REG. (8)	E REG. (8)
H REG. (8)	L REG. (8)

(b) Äquivalente Darstellung

Abbildung 13: Simulator Register und Darstellung

Damit gemeint sind die generischen Register ohne feste Funktion, welche jeweils eine Speicherkapazität von acht Bit haben. Über die in Abbildung 14 dargestellte Funktion ist es möglich, ein Interrupt mit einem gültigen Interrupt-Bytevektor auszulösen.

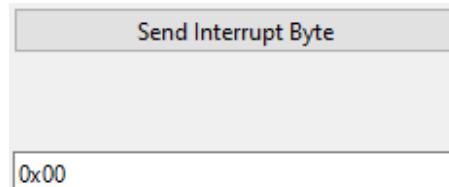


Abbildung 14: Bereich zum Senden eines Interrupts

Dabei wird geprüft, ob die Operation gültig ist. Ist die der Fall, wird sie beim nächstmöglichen Zustand berücksichtigt und ausgeführt. Ist die Operation ungültig, wird eine Fehlermeldung ausgegeben.

Die letzte Funktionalität der Benutzeroberfläche ist ein Log in dem die einzelnen Operationen aufgelistet werden, die in jedem Zustand vorgenommen werden (siehe Abbildung 15).

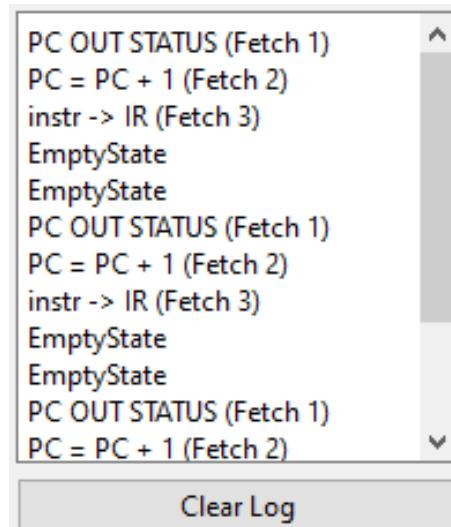


Abbildung 15: Zustands-Log für aufeinanderfolgende NOP-Befehle

Über den darunterliegenden „Clear Log“-Knopf lässt sich der Inhalt des Logs löschen. Bei Laden eines neuen Programms wird der Log ebenfalls geöschzt. Ein nicht mehr in der finalen Version der GUI vorhandenes Register, welches dem Nutzer angezeigt wird und welches bearbeitet werden kann ist in Abbildung 16 zu sehen.

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Abbildung 16: Adresspuffer

Gezeigt ist der Adresspuffer, welcher für das Auswählen von externen Geräten über das Bussystem zuständig ist. Die Darstellung wurde hier bitweise gewählt, das heißt, statt die ganzen 2 Byte des Registers zu bearbeiten, werden die einzelnen Bits angesteuert.

## 4.4 Tutorial

Das Tutorial für die Handhabung des Simulators ist über das Hauptmenü erreichbar. Es besteht aus fünf verschiedenen Bildern, welche jeweils die einzelnen Elemente des Simulators zeigen und deren Funktion näher beschreiben (siehe Abbildung 17).

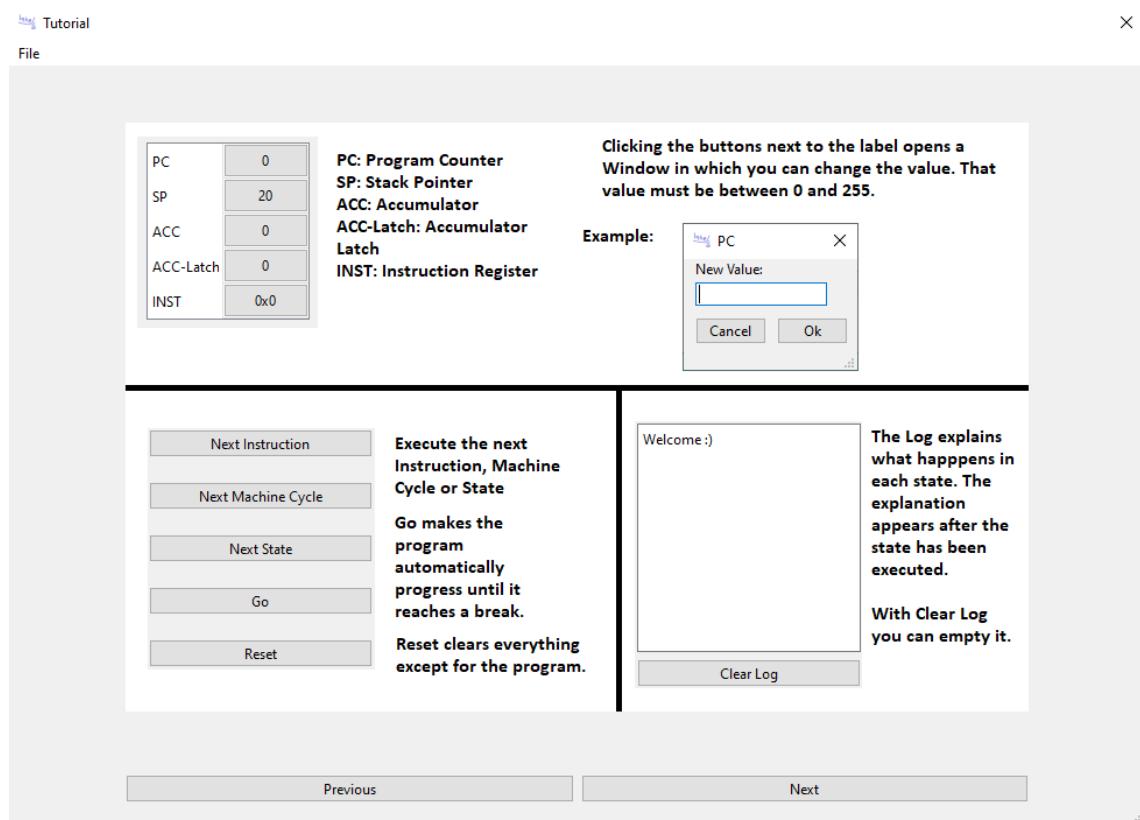


Abbildung 17: Tutorial-Fenster

Um zum nächsten bzw. vorherigen Bild zu wechseln werden die zwei unteren Knöpfe verwendet. Der Text wurde wie der Rest der Simulatoroberfläche in Englisch verfasst.

## 5 Implementierung

Von Andreas

Es wurde ein Simulator eines 8-Bit-Prozessors geschrieben. Der 8-Bit-Prozessor wurde durch eine Software mit graphischer Oberfläche realisiert. Diese zeigt dem Benutzer den aktuellen Zustand des Prozessors (Intel 8080) an. Der Simulator unterstützt alle Befehle des realen Prozessors und kann Interrupts ausführen. Lediglich der Single-Prozessor-Simulator ist in der Lage auch einen Interrupt durch die GUI zu generieren.

### 5.1 Multi-Prozessor-Simulator

Mit diesem Simulator soll es möglich sein, zuvor assembledten Code auszuführen. Dafür wird ein Assembler [10] verwendet, der aus dem Source Code (.asm) eine .com-Datei erstellt. Aus dieser kann das komplette Programm ausgelesen werden. Zusätzlich dazu wird noch eine Datei (.sym) erstellt, die die Labels/Sprungmarken abspeichert. Dieser Assembler wurde mit in den Simulator integriert. Diese Version des Simulators verfügt über keine GUI, deshalb muss dem Prozessor der Assemblercode in der Source-Datei (Intel8080.py) übergeben werden. Der Prozessor leitet den Code zum Assembler weiter und liest das Ergebnis in seinen Programmspeicher ein. Zuallererst wurde der Intel 8080 implementiert. Dieser wurde voll-funktionsfähig implementiert, bevor damit begonnen wurde, andere Prozessoren zu implementieren. Im Zuge der Recherche ergab sich, dass sich die Prozessoren nur minimal unterscheiden, deshalb wurde sich dafür entschieden einen Prozessor genauer zu simulieren. Die Beschreibung dieses Simulators kann in Kapitel 5.2 gefunden werden.

#### 5.1.1 Programmablauf

Der eigentliche Simulator verwendet nur die .com-Dateien. Zuallererst wird die .com-Datei ausgelesen und in den Programmspeicher des Prozessors geschrieben. Um den ersten Befehl ausführen zu können, wird zuerst nur ein Byte aus dem Programmspeicher, an der Stelle des Programmzählers, eingelesen. Danach wird das erste Byte dekodiert. Je nach Befehl werden noch weitere Bytes eingelesen, die für den eigentlichen Befehl benötigt werden. Sobald alle nötigen Informationen eingelesen wurden, wird der Befehl ausgeführt. Jeder Befehl muss entsprechend den Programmzähler anpassen, damit der nächste Befehl wieder korrekt eingelesen werden kann. Mit diesem Simulator ist es auch möglich, Interrupts auszulösen. Vor der Ausführung eines Befehls wird geprüft, ob ein Interrupt ausgelöst wurde. Ist das der Fall, wird das Interrupt-Byte ausgewertet und ausgeführt. Normalerweise wird dieses Byte von

dem Interrupt-Erzeuger mitgeschickt. Meistens ist dies ein Reset-Befehl, der an eine von acht Stellen springt und die dort liegenden Befehle ausführt. Mit 3 Bits kann damit entschieden werden, wo die Subroutine beginnen soll (Byte 0, 8, 16 ... 56). Es ist aber auch möglich, dass ein anderer, ein Byte langer Befehl übergeben und direkt ausgeführt wird.

### 5.1.2 Dekodieren eines Befehls

Um aus einem Byte einen Befehl analysieren zu können, wird dieser Wert mit vielen Hexadezimalwerten verglichen. Anhand der Dokumentation des Intel 8080 werden somit alle Befehle unterschieden. Wie in Abbildung 18 zu sehen ist, wird der Befehl „Add Immediate to A with carry (ACI)“ mit dem Wert 0xCE assoziiert. Bei vielen Befehlen wie ACI wird auf genau einen Wert geprüft. Es gibt aber auch andere Befehle, wobei nicht alle Bits des Bytes den Befehl beschreiben. In Abbildung 18 ist zu sehen, dass der Befehl „Add Memory to A with carry (ADC)“ nicht so einfach verglichen wird. Bei diesem Befehl werden in den letzten drei Bits des Befehlsbytes das 8-Bit-Register kodiert, welches verwendet werden soll. Deshalb wird die Instruction-Variable zuerst mit dem Wert 0xF8 bitweise verUNDet, damit die letzten drei Bit ignoriert werden. Somit werden alle Bitkombinationen erkannt, bei denen die ersten fünf Bits mit dem Wert 0x88 übereinstimmen. Wenn dann bekannt ist, dass es sich um einen ADC-Befehl handelt, wird das Register ermittelt, das verwendet werden soll. Dies geschieht über die `get_reg8d_from_inst(instruction)`-Methode. Für die unterschiedlichen Fälle, der Register-Kodierung, gibt es jeweils eine eigene Methode. Diese liefert einen Wert zwischen 0 und 8, entsprechend der Abbildung 1.

```
def identify_and_execute_instruction(self, instruction):
    if instruction == 0xCE:
        self.ALU.aci(self.get_one_byte_data())
    elif (instruction & 0xF8) == 0x88:
        self.adc(self.get_reg8d_from_inst(instruction))
    elif (instruction & 0xF8) == 0x80:
        self.add(self.get_reg8d_from_inst(instruction))
    elif instruction == 0xC6:
```

Abbildung 18: Funktion für das Lesen und Dekodieren eines Bytes

Dieses Vorgehen wird bei einigen Befehlen verwendet. In Kapitel 3.3 wird genauer gezeigt, wie die Register in den Befehlen kodiert werden.

### 5.1.3 Programmaufbau

Der Simulator besteht hauptsächlich aus drei Klassen:

- Dem Prozessor (Intel8080.py)
- Der ALU (Intel8080\_ALU.py)
- Den Register (Intel8080\_Registers.py)

Die Register-Klasse ist hauptsächlich als Datenspeicher gedacht und besitzt keine spezielle Methoden. In der Prozessor- und der ALU-Klasse werden Befehle ausgeführt. Es werden zwei Arten von Befehlen unterschieden. Befehle wie Sprungbefehle, die die ALU nicht benötigen und Befehle wie Additionen, die die ALU verwenden. Die Befehle, die die ALU nicht verwenden, wurden in der Intel8080.py implementiert, die restlichen entsprechend in der ALU. Jeder Befehl wird durch eine eigene Methode implementiert. Somit kann ein Befehl schnell gefunden und angepasst werden. In diesen Methoden läuft der komplette Befehl ab. Somit werden in diesen Methoden auch weitere Bytes eingelesen, die für die Ausführung des Befehls notwendig sind. Bei jedem Lesevorgang aus dem Programmspeicher wird der Programmzähler automatisch um eins erhöht. Deshalb ist es nicht möglich das gleiche Byte mehrfach hintereinander zu lesen. Um das gleiche Byte nochmal lesen zu können, müsste der Programmzähler manuell angepasst werden.

```
def get_one_byte_data(self):  
    self.add_pc(1)  
    return np.uint8(self.get_memory_byte(self.get_pc()))
```

Abbildung 19: Implementierung des Lesens eines Bytes

Der Befehl DCX reduziert das angegebene 16-Bit Register um eins. Abbildung 20 zeigt die Implementierung des DCX-Befehls. Dort ist zu sehen, dass es bei den 16-Bit Registern eine grundlegende Unterscheidung gibt. Es gibt einen Fall für den Stackpointer und einen für die anderen 16-Bit Register, die aus zwei 8-Bit Registern zusammengesetzt werden. Mittels des übergebenen Parameters „rp“ kann ermittelt werden ob es sich um den Stackpointer ( $rp == 3$  bedeutet Stackpointer) handelt, der angepasst werden soll. Der Stackpointer kann einmal ausgelesen, um 1 reduziert, und dann wieder in sein Register gespeichert werden. Bei den zusammengesetzten 16-Bit Registern, werden zuerst beide 8-Bit Register ausgelesen und dann zusammengebaut. Dieses Ergebnis kann dann um 1 reduziert werden. Danach wird der 16-Bit Wert wieder aufgetrennt und in die 8-Bit Register gespeichert.

```
def dcx(self, rp):
    if self.is_rp_meaning_sp(rp):
        reg_val = np.uint16(self.registers.get_register(1))
        result = np.uint16(reg_val - 1)
        self.set_sp(result)
    else:
        reg_h_value, reg_l_value = self.get_rp_values(rp)
        reg_val = build_16bit_from_8bit(reg_h_value, reg_l_value)
        result = np.uint16(reg_val - 1)
        self.registers.set_2_8bit_reg_with_offset((rp * 2), result)
```

Abbildung 20: Implementierung des DCX-Befehls

Die zusammengesetzten Register können alle gleich behandelt werden, da bei dem Programmkonzept beachtet wurde, dass dies öfter vorkommen kann. Deshalb wurden die Register so positioniert, dass auf die Register anhand ihrer Kodierung (siehe Abbildung 1) direkt zugegriffen werden kann. Ebenso liegt damit das High Byte eines 16-Bit Registers einen Platz vor dem Low Byte. Mit dieser Anordnung kann mittels einfacher mathematischer Berechnungen die entsprechende Array-Stelle ermittelt werden. So kann aus einem rp-Wert (0-2) das entsprechende Array-Feld (High Byte) mittels einer Multiplikation mit 2 und der Addition eines Offsets berechnet werden. Um auf das Low Byte zu kommen, muss dann nur noch 1 addiert werden.

```
self.registers = [np.uint16(0), # Program Counter
                  np.uint16(100), # Stack Pointer 100d
                  np.uint8(0), # B-REG 000
                  np.uint8(0), # C-REG 001
                  np.uint8(0), # D-REG 010
                  np.uint8(0), # E-REG 011
                  np.uint8(0), # H-REG 100
                  np.uint8(0), # L-REG 101
                  np.uint8(0), # Space for better REG allocation (110 -> Memory)
                  np.uint8(0), # A-REG 111 / Accumulator acc
                  np.uint8(0), # W-REG
                  np.uint8(0), # Z-REG
]
```

Abbildung 21: Register Array des Intel 8080

### 5.1.4 Befehlsvarianten

Der Intel 8080 kennt mehrere Befehle, die sowohl eine direkte als auch eine indirekte Variante besitzen. Der Unterschied liegt darin, dass die verrechneten Werte, aus unterschiedlichen Stellen geladen werden. Aus diesem Grund greifen beide Befehle im Hintergrund auf die gleichen Methoden zu. Somit ist sicher gestellt, dass bei Bedarf nur eine Stelle im Quellcode angepasst werden muss.

Ein Beispiel dafür ist die „Subtract register/immediate from A with borrow (SBB/SBI)“. Es gibt zwar für beide Mnemonics eine Methode in der Intel8080-Klasse, die Methode von SBB leitet aber später in der ALU-Klasse nur noch auf die SBI-Methode weiter.

```
def sbb(self, val_to_subtract):
    self.sbi(val_to_subtract)

def sbi(self, val_to_subtract):
    if self.get_carry_flag():
        val_to_subtract = np.uint8(val_to_subtract + 1)
    self.sui(val_to_subtract)
```

Abbildung 22: Zugrundeliegende Implementierung von SBB und SBI

Die direkten Befehle lesen dabei das zweite Byte, das zu diesem Befehl gehört und rufen die Methode SBI in der ALU-Klasse auf. Für indirekte Befehle wie SBB wird zuerst in der Intel8080-Klasse noch der Wert aus dem Speicher gelesen. Dieser wird deshalb gelesen, da die ALU keinen direkten Zugriff auf den Speicher hat. Mit dem gelesenen Wert wird dann in der ALU-Klasse die SBB-Methode aufgerufen. Diese leitet innerhalb der Klasse nur noch auf die SBI Methode weiter.

Die Sprungbefehle werden ähnlich behandelt. Für diese gibt es mehrere Varianten, die aber nur auf unterschiedliche Flags reagieren. Deshalb wurde auch dort eine zugrundeliegende Methode geschrieben, auf die alle anderen verweisen. So wird nur für die spezifische Varianten das verwendete Flag weitergegeben. Abbildung 23 zeigt wie bei dem Jump-on-Carry Befehl vorgegangen wird. So ruft die jc-Methode die jumpon-Methode mit dem Carry-Flag auf. In der jump\_on-Methode werden immer die nächsten zwei Bytes gelesen, die die Adresse beinhaltet, auf die gesprungen werden soll. Nur sofern das übergebene Flag gesetzt ist wird der Sprung ausgeführt andernfalls wird das Programm normal weitergeführt.

```
def jump_general(self, low, high):
    self.set_pc(build_16bit_from_8bit(high, low))

def jump_on(self, flag: bool):
    low = self.get_one_byte_data()
    high = self.get_one_byte_data()
    if flag:
        self.jump_general(low, high)
    else:
        pass

def jc(self):
    self.jump_on(self.ALU.get_carry_flag())
```

Abbildung 23: Implementierung der Sprungbefehle

### 5.1.5 Binäre Addition und Subtraktion

Da die Additions- und Subtraktionsbefehle das Carry- und Auxiliary-Carry-Flag verändern, musste eine manuelle Addition implementiert werden. Diese ist in Abbildung 24 zu sehen. Das Carry-Flag, das anzeigt, ob ein Überlauf beim siebten Bit stattgefunden hat, hätte noch ohne eine eigene Implementierung der Addition funktioniert. Dafür müsste vor dem Casten in einen 8-Bit Wert überprüft werden, ob der Wert größer als 255 ist. Da aber zusätzlich noch das Auxiliary-Carry-Flag beachtet werden musste, das einen Überlauf von Bit 3 anzeigt, wurde die Addition selber implementiert. Diese Implementierung liefert das Carry-, Auxiliary-Carry-Flag und das Ergebnis der Addition. Somit können die Flags korrekt gesetzt werden, sofern die Befehle diese Flags ändern. Wird eines der beiden Flags verändert, wird darauf zurückgegriffen. Der INX-Befehl, der einen 16-Bit Wert um 1 erhöht, verändert aber keine Flags, weshalb dort eine normale Addition durchgeführt wird.

```
def binary_add(self, op1, op2, carry: int):
    mask = 0x01
    result, ac, cy = 0, 0, 0
    for cycle in range(16):
        value = (op1 & mask) + (op2 & mask) + (carry * mask)

        result += value & mask
        mask <= 1
        if value & mask:
            carry = 1
        else:
            carry = 0

        if cycle == 3:
            ac = carry
        if cycle == 7:
            cy = carry

    return ac, cy, result
```

Abbildung 24: Implementierung der binären Addition

## 5.2 Single-Prozessor-Simulator

In diesem Kapitel werden die Funktionen der aktuellen Version des Simulators beschrieben. Dieser unterscheidet sich insofern von dem Multi-Prozessor-Simulator, dass jeder Befehl aus mehreren Maschinenzyklen und diese wiederum aus mehreren Zuständen bestehen. So ist es möglich, komplexe Befehle aber auch einzelne Maschinenzyklen oder sogar Zustände auszuführen. Damit kann wesentlich besser nachvollzogen werden, wie ein Prozessor tatsächlich funktioniert. In der Multi-Prozessor-Variante konnten zwar Programme ausgeführt werden, aber der Ablauf der Befehlausführung unterschied sich deutlich von der Funktionsweise eines realen Prozessors.

### 5.2.1 Aufbau eines Befehls

Jeder Befehl besteht aus einem Array von Maschinenzyklen. Zusätzlich enthält jeder Befehl auch einen Zähler, wie viele Maschinenzyklen der Befehl schon ausgeführt hat. Ein Befehl hat auch nur eine Methode, die zur Ausführung des Befehls notwendig ist. Die `next_state()`-Methode des Befehls sorgt dafür, dass der aktuelle Maschinenzyklus seine `next_state()`-Methode ausführt. Wenn die Maschinenzyklus `next_state()`-Methode True zurückliefert, bedeutet das, dass der letzte Zustand des Maschinenzyklus ausgeführt wurde. Wenn dies der Fall ist, wird der Maschinenzyklus Zähler um eins erhöht und es wird ebenfalls ein True zurückgeliefert. Dieses Signal des Maschinenzyklus wird bis in die äußerste Schicht, die GUI, weitergeleitet, damit dort spezielle Ereignisse ausgeführt werden können. Beim nächsten Aufruf von `next_state()` wird somit der erste Zustand des nächsten Maschinenzyklus aus-

geführt.

Jeder Maschinenzyklus besteht aus einem Array von Zuständen, einem Zähler, der die bisher ausgeführten Zustände zählt und einer Referenz auf den Prozessor. Diese Referenz ist notwendig, damit die Zustände, die in den Maschinenzyklen ausgeführt werden, den Prozessor verändern und dessen Methoden verwenden können. Die `next_state()`-Methode führt immer die `run()`-Methode der Zustände aus und erhöht seinen Zähler um eins. Wenn der eben ausgeführte Zustand der letzte im Maschinenzyklus war, wird der Zähler zurückgesetzt und ein True zurückgegeben.

Ein Zustand besitzt nur eine Referenz zum Prozessor und eine `run()`-Methode, die den Zustand ausführt. Da ein Zustand zu allen möglichen Stellen in einem Maschinenzyklus verwendet werden kann, besitzt der Zustand selber keine weiteren Informationen.

### 5.2.2 Programmablauf

Wie bei der Multi-Prozessor-Version dieses Simulator wird eine .com-Datei zur Ausführung des Programms benötigt. Über die GUI kann eine solche Datei eingelesen werden. Damit wird wieder der Programmspeicher gefüllt. Die GUI besitzt vier Knöpfe, die das ausführen eines Befehls/Maschinenzyklus oder Zustand auslöst. Bei dem „Next State“ -Knopf wird genau ein Zustand ausgeführt. Bei dem „Next Machine Cycle“ -Knopf wird der aktuelle Maschinenzyklus bis zum Schluss ausgeführt. Der „Next Instruction“ -Knopf verhält sich wie der Maschinenzyklus-Knopf. Der „Go“ -Knopf führt zu solange den nächsten Zustand aus bis der Knopf wieder gedrückt wird. Die drei ersten Knöpfe sind auf eine Methode in der Intel8080-Klasse gebunden, die den selben Namen tragen wie der Knopf. Jeder dieser Methoden gibt True zurück, wenn der nächste auszuführende Zustand zu einem neuen Befehl gehört. Die `next_instruction()`-Methode führt solange die `next_machine_cycle()`-Methode aus, bis diese True zurücklieft. Diese wiederum führt solange die `next_state_internal()`-Methode aus, bis diese True zurücklieft. Diese liefert True, wenn der letzte Zustand eines Maschinenzyklus ausgeführt wurde. Die `next_state()`-Methode ruft nur einmal die `next_state_internal()`-Methode auf.

Die `next_state_internal()`-Methode, steuert wann ein neuer Befehl dekodiert und eine neue Klasse geladen werden muss. Wenn der nächste Zustand der erste eines neuen Befehls ist, wird der `current_instruction`-Variable ein NOP-Befehl zugewiesen. Da bei allen Befehlen die ersten drei Zustände die gleichen sind, ist es unbedeutend, welcher Befehl hier geladen wird. Erst im dritten Zustand wird der tatsächliche Befehl geladen. Von diesem Befehl wird der zuletzt ausgeführte Zustand auf drei gesetzt. Wenn der letzte Zustand eines Maschinenzyklus ausgeführt wurde, wird True

zurückgegeben. Wenn ein Befehl frühzeitig abgebrochen werden soll, z.B. bei einem bedingten Sprung, wird das ebenfalls hier gesteuert.

### 5.2.3 Dekodieren eines Befehls

Bei jedem Fetch-Zyklus wird das aktuelle Befehlsbyte in die `cpu_instruction_register`-Variable geschrieben. Ähnlich wie bei der Multi-Prozessor-Version, wird beim Dekodieren dieses Byte mit Hexadezimalen Werten verglichen. Bei dieser Version wird beim Dekodieren aber unterschieden zwischen Befehlen, die entweder ein Register oder eine Adresse als Parameter erhalten (siehe Abbildung 25). Dies ist deshalb wichtig, da sich die Maschinencyklen davon unterscheiden. Abbildung 18 zeigt, dass in der ersten Version für alle ADD-Befehle nur eine Methode aufgerufen wird. Damit der Prozessor die richtigen Klassen zur Ausführung verwendet, wird in der `current_instruction`-Variable ein Objekt des richtigen Befehls gespeichert.

```
def decode_instruction(self):
    if self.cpu_instruction_register == 0xCE:
        self.current_instruction = Aci(self)
    elif (self.cpu_instruction_register & self.sss_inv_mask) == 0x80:
        if self.cpu_instruction_register == 0x86:
            self.current_instruction = Add_m(self)
        else:
            self.current_instruction = Add_r(self)
    elif (self.cpu_instruction_register & self.sss_inv_mask) == 0x88:
```

Abbildung 25: Implementierung der Dekodierung

### 5.2.4 Programmaufbau

Der Simulator besteht, wie der Multi-Prozessor-Simulator, immer noch hauptsächlich aus dem Prozessor, der ALU und den Registern. Durch die Verwendung eigener Klassen für Befehle, Maschinencyklen und Zustände, gibt es nun wesentlich mehr Klassen. Insgesamt werden

- 74 Befehls-Klassen,
- 85 Maschinencyklen-Klassen, und
- 101 Zustands-Klassen

verwendet. Zusätzlich dazu gibt es noch ungefähr 13 Klassen, von denen die oben genannten abgeleitet sind.

Der in Kapitel 5.1.3 gezeigte DCX-Befehl sieht nun folgendermaßen aus.

```
class Dcx(Instruction):
    def __init__(self, processor):
        super().__init__(processor)
        self.machine_cycles = [Dcx_M1(processor)]
    pass
```

Abbildung 26: DCX-Befehl

```
class Dcx_M1(Fetch):
    def __init__(self, processor):
        super().__init__(processor)
        self.states.append(EmptyState(processor))
        self.states.append(rp_decr(processor))
```

Abbildung 27: DCX-Maschinencyklus

Bei vielen Befehlen werden leere Zustände verwendet. Das kommt daher, dass es mit dem Simulator momentan nicht möglich ist, einen Zustand zu erstellen, der zwei Takte dauert. Deshalb wird mit dem ersten Takt ein leerer Zustand ausgeführt und im zweiten dann erst der eigentliche Zustand. Somit werden die Daten zur gleichen Zeit in die Register/Speicher geschrieben, wie es bei einem realen Prozessor auch der Fall ist.

```
class rp_decr(State):
    def __init__(self, processor):
        super().__init__(processor)

    def run(self):
        print("rp_decr")
        rp = self.processor.get_current_rp()

        if self.processor.rp_means_sp():
            sp = self.processor.get_sp()
            sp = np.uint16(sp - 1)
            self.processor.set_sp(sp)
        else:
            h_val, l_val = self.processor.get_rp_values(rp)
            hl_val = build_16bit_from_8bit(h_val, l_val)
            hl_val = np.uint16(hl_val - 1)
            self.processor.registers.set_2_8bit_reg_with_offset(rp * 2, hl_val)
            self.processor.StateLogger.addEntry("rp_decr")
```

Abbildung 28: DCX (rp\_decr) Zustand

Da der DCX-Befehl nur aus einem Maschinencyklus besteht und mit dem leeren Zustand nur über einen Zustand verfügt, der den Befehl ausmacht, werden im Fol-

genden anderen Befehle gezeigt, die diese Klassenstruktur besser veranschaulichen. Diese Beispiel sollen auch verdeutlichen, wie Befehle des gleichen Typs auf die selben Maschinenzyklen und Zustände zurückgreifen.

### 5.2.5 Testing

Um sicherzustellen, dass alle Befehle korrekt ausgeführt werden, wurden Tests erstellt. Speziell wurden Integrationstests geschrieben, um zu kontrollieren, dass dies der Fall ist und die Zustände sehr simpel aufgebaut sind. Abbildung 29 zeigt wie einer der Tests aufgebaut ist.

```
def test_aci(self):
    try:
        intel = Intel8080()
        intel.init_test("aci 03")

        intel.set_acc(7)

        intel.run_complete_programm(1)

        self.assertEqual(70, intel.get_acc())
    except:
        self.fail()
```

Abbildung 29: Test für den ACI-Befehl

Zuerst wird ein Intel8080-Prozessor-Objekt (ohne GUI) erzeugt. Mit `intel.init_test()` wird die Zeichenkette zuerst assembliert und dann in den Programmspeicher geschrieben. In diesem Fall wird zu Vorbereitung der Wert 7 in den Akkumulator geschrieben. Mit `intel.run_complete_programm()` führt der Simulator so viele Befehle aus wie durch den Zahlenwert übergeben. Dies ist notwendig, wenn z.B. mit Labels gearbeitet wird und diese nicht auf die Adresse 0 verweisen sollen. Nach der Ausführung des Befehls wird überprüft, ob der Wert im Akkumulator der korrekte ist. Sofern irgendwo ein Fehler auftaucht, gilt der Test als gescheitert.

Mit dem obigen Test wurde geprüft ob der Ergebniswert der zu erwartende ist. Da manche Befehle von Flags abhängig sind und diese teilweise auch verändern, gibt es noch andere Tests, um dies zu kontrollieren. Nicht jeder Befehl wird so ausgiebig getestet wie der ADC-Befehl (siehe 30). Da der ADC r, der ADC m und der ACI-Befehl aus dem ADC\_MC-Maschinenzyklus bestehen, wird nur einer dieser Befehle besonders stark getestet. Wenn ein anderer Zustand eines Befehls nicht korrekt wäre, würde sehr wahrscheinlich der einfache Test für einen, wenn nicht sogar mehrere, einen Fehler anzeigen. Somit wird mit mehreren weiteren Tests für

den Maschinenzyklus hauptsächlich die sich verändernden Flags getestet.

```
def test_adc_r_overflow_with_carry(self):
    try:
        intel = Intel8080()
        intel.init_test("adc e")

        intel.set_acc(255)
        intel.registers.set_register8_with_offset(char_to_reg("E"), 0)
        intel.ALU.set_carry_flag(True)

        intel.run_complete_programm(1)

        self.assertEqual(0, intel.get_acc())
        self.assertTrue(intel.ALU.get_carry_flag())
        self.assertTrue(intel.ALU.get_auxiliary_carry_flag())
        self.assertTrue(intel.ALU.get_zero_flag())
        self.assertFalse(intel.ALU.get_sign_flag())
        self.assertTrue(intel.ALU.get_parity_flag())
    except:
        self.fail()
```

Abbildung 30: einer von mehreren Tests für den ADC-Befehl

In diesem Test (Abbildung 30) wird überprüft, ob sich die Flags korrekt verhalten, wenn ein Überlauf stattfindet. Ebenso wird hier auch noch mal getestet, ob das Carry-Flag mit in die Berechnung einfließt. Deswegen wird vor dem Befehl das Carry-Flag gesetzt. Da dieser ADC r Befehl auf ein Register zugreift, muss dieses auch vor dem Test noch befüllt werden. Nach der Ausführung des Programms werden alle Flags ausgewertet. Sofern nur eines davon nicht den korrekten Wert enthält, wird der Test als nicht bestanden markiert.

### 5.2.6 Befehlsvarianten

Alle Befehle haben die gleichen ersten drei Zustände. Aus diesem Grund wurde ein Fetch-Maschinenzyklus erstellt, von dem jeder erste Maschinenzyklus abgeleitet wurde. Der RST-Befehl hat, als einzige Ausnahme, einen anderen dritten Zustand. Normalerweise wird im dritten Zustand nur der aktuelle Befehl in das Befehls Register geschrieben. Beim RST-Befehl wird aber zusätzlich noch das W-Register auf Null gesetzt. Dies ist deswegen wichtig, da der Befehl an eine von acht Programmstellen springen soll. Diese liegen alle am Anfang des Programmspeichers und benötigen deswegen ein leeres W-Register. Für Sprünge werden das W- und Z-Register verwendet. In den Maschinenzyklen 2-4 wird aber nur das Z-Register auf den entsprechenden Wert gesetzt.

Bei einigen Befehlen gibt es grundsätzlich drei verschiedene Typen, die unterschieden werden. Einen direkten Befehl, der ein weiteres Byte hinter dem Befehl im

Programmspeicher liest, einen Befehl, der einen Register-Wert liest, im folgenden als Register-Befehl bezeichnet, und einen Befehl, der zuerst die zwei folgenden Bytes aus dem Programmspeicher liest, daraus eine Adresse bildet und dann an diese Adresse springt und den dort liegenden Wert verwendet, im restlichen Kapitel an Speicher-Befehl bezeichnet.

Beschreibung	Register-Befehl	Speicher-Befehl	Direkter Befehl
Addition mit Acc	ADD r	ADD m	ADI
Addition mit Acc & Cy	ADC r	ADC m	ACI
Subtraktion mit Acc	SUB r	SUB m	SUI
Subtraktion mit Acc & Cy	SBB r	SBB m	SBI
Bitweise UND-Operation	ANA r	ANA m	ANI
Bitweise XOR-Operation	XRA r	XRA m	XRI
Bitweise ODER-Operation	ORA r	ORA m	ORI
Vergleich mit Acc	CMP r	CMP m	CPI

Tabelle 2: Befehle mit drei Befehls-Typen

Für die Addition ohne Carry gibt es diese drei Typen. ADD r verwendet ein Register. ADD m verwendet einen Wert, der irgendwo im Programmspeicher liegt und der ADI-Befehl verwendet das zweite Byte des Befehls als Summand. Alle drei Befehle besitzen die Aufgabe, einen Wert mit dem Akkumulator zu addieren. Dazu nutzt aber jeder Befehl Werte von unterschiedlichen Stellen und muss diese unterschiedlich laden. Als letzter Maschinenzyklus addieren diese drei Befehle das ACT- und TMP-Register ohne Carry. Das Ergebnis wird dann in den Akkumulator geschrieben. Deshalb besitzt jeder dieser Befehle als Letztes den Add\_MC Maschinenzyklus. In das ACT-Register wird immer der ursprüngliche Wert des Akkumulators geladen. Die Befehle unterscheiden sich allein darin, dass in den vorherigen Maschinenzyklen andere Werte in das TMP-Register gespeichert werden (siehe Tabelle 3).

### Unterschiede des ersten Maschinenzyklus

Der direkte Befehl und der Speicher-Befehl besitzen den gleichen ersten Zustand, sie schreiben schlicht den Inhalt des Akkumulators in das ACT-Register. Der Register-Befehl kann im ersten Maschinenzyklus sowohl den Wert des Akkumulators in das ACT-Register, als auch das im Befehl kodierte Register in das TMP-Register laden.

### Unterschiede des zweiten Maschinenzyklus

Der Register-Befehl hat im ersten Zyklus schon beide Register so vorbereitet, dass er im zweiten Zyklus diese Werte bereits addieren kann. Für diesen Befehlstyp ist das schon der letzte Zyklus. Die beiden anderen Befehle müssen zuerst noch das

TMP-Register mit dem korrekten Wert laden. Der direkte Befehl muss dafür zuerst den Programmzähler um eins erhöhen, damit das folgende Byte ausgelesen werden kann. Der Speicher-Befehl lädt den Wert an der angegebenen Adresse, direkt in das TMP-Register.

### Unterschiede des dritten Maschinenzyklus

Der Register-Befehl führt keinen dritten Zyklus aus. Der direkte Befehl und der Speicher-Befehl führen jetzt den gleichen Maschinenzyklus aus, der der Register-Befehl schon im zweiten Zyklus ausgeführt hat. Eine Zusammenfassung der Zyklen für die drei Befehlstypen ist in Tabelle 3 gegeben.

Befehlstyp	1. Zyklus	2. Zyklus	3. Zyklus
Register-Befehl	(SSS) → TMP	ADD_MC	
Speicher-Befehl	(ACC) → ACT	Data → TMP	ADD_MC
Direkter Befehl	(ACC) → ACT	Byte 2 → TMP	ADD_MC

Tabelle 3: Maschinenzyklen der unterschiedlichen Typen

#### 5.2.7 Interrupts

Ein Interrupt-Befehl wurde implementiert, um die Ausführung von Interrupts zu ermöglichen. Dieser führt nicht wirklich den Interrupt aus. Dieser dient nur dem Zweck, dass der Programmzähler hier im zweiten Zustand nicht um eins erhöht wird. Im Handbuch des Intel 8080 [6] wird der Interruptzyklus als Fetch-Zyklus beschrieben, bei dem der Programmzähler nicht erhöht wird. Viele Befehle haben in ihrem ersten Maschinenzyklus genau drei Zustände. Der erste beschreibt den Maschinenzyklus auf dem Datenbus, der zweite erhöht den Programmzähler und der dritte liest den nächsten Befehl ein und schreibt in das Instruction-Register. Somit wäre ein Fetch-Zyklus eigentlich beendet. Der Interrupt-Maschinenzyklus besitzt in der Implementierung dieses Simulators allerdings noch einen vierten, leeren Zustand. Dieser ist wichtig, da die Dekodierung des Befehls erst nach dem dritten Zustand erfolgt. Wenn der Befehl bzw. der erste Maschinenzyklus aber nur aus drei Zuständen bestehen würde, würde die Dekodierung nicht erfolgen und der übergebene Interrupt-Befehl könnte nicht ausgeführt werden.

### 5.2.8 Besonderheiten des Simulators

Bei der Entwicklung dieses Simulators wurden einige Eigenheiten eines realen Prozessors nicht repliziert. Im Folgenden soll aufgezeigt werden, welche Unterschiede es zu einem echten Prozessor gibt.

#### Adress- und Datenbus

Der Intel 8080 besitzt einen 8-Bit-Daten- und einen 16-Bit-Adressbus. In diesem Simulator wurde kein Bus implementiert. Grund dafür ist, dass die Busse anhand von verschiedenen Taktgebern beschrieben und gelesen werden. Dies könnte zwar auch simuliert werden, letztendlich wurde aber beschlossen, diese Funktion aus mangelndem Nutzen nicht zu implementieren. Um trotzdem mit dem Programmspeicher kommunizieren zu können wurde es dem Prozessor ermöglicht, ohne einen Bus, direkt auf die Daten zuzugreifen. Während die Busse die übertragene Daten direkt einsehbar machen würden, zeigen die Register/Speicherstellen erst am Ende eines Befehls Veränderungen an.

In jedem ersten Zustand aller Maschinenzyklen wird über den Datenbus ein 8-Bit-Statuswort gesendet. Damit lässt sich erkennen, welche Art von Maschinenzyklus ausgeführt wird. Die notwendigen Zustände dafür existieren, schreiben aber nichts in den Bus. In Abbildung 31 ist zu sehen, wie die verschiedenen Zustände auf dem Bus kodiert werden würden. So wird ein Memory Write Zyklus dadurch bestimmt, dass alle Datenbits  $d_{0-7}$  auf Null gesetzt werden.

STATUS WORD CHART												
	TYPE OF MACHINE CYCLE											
	DATA BUS BIT	INFORMATION	INSTRUCTION FETCH	MEMORY READ	MEMORY WRITE	STACK READ	STACK WRITE	INPUT READ	OUTPUT WRITE	INTERRUPT ACKNOWLEDGE	HALT ACKNOWLEDGE	INTERRUPT ACKNOWLEDGE WHILE HALT
D <sub>0</sub>	INTA	0	0	0	0	0	0	0	1	0	1	
D <sub>1</sub>	WO	1	1	0	1	0	1	0	1	1	1	
D <sub>2</sub>	STACK	0	0	0	1	1	0	0	0	0	0	
D <sub>3</sub>	HLTA	0	0	0	0	0	0	0	0	1	1	
D <sub>4</sub>	OUT	0	0	0	0	0	0	1	0	0	0	
D <sub>5</sub>	M <sub>1</sub>	1	0	0	0	0	0	0	1	0	1	
D <sub>6</sub>	INP	0	0	0	0	0	1	0	0	0	0	
D <sub>7</sub>	MEMR	1	1	0	1	0	0	0	0	1	0	

Abbildung 31: Kodierung der Befehlstypen beim SYNC Takt [11]

### Zustände, die zwei Takte dauern

Manche Zustände des Intel 8080 dauern zwei Takte an. Zu sehen ist das in Anhang A. Die Judge Condition, bei z.B. den bedingten Sprüngen, benötigt den vierten und fünften Zustand. Dies ist in diesem Simulator bisher nicht möglich. Deshalb wird der vierte Zustand ein leerer Zustand und der fünfte ist der eigentliche Zustand.

Ähnliches ist auch bei anderen Zuständen der Fall. Ebenfalls bei den bedingte Sprüngen gibt es im zweiten Maschinenzyklus einen Zustand, der das zweite Byte in das Z-Register schreiben soll. Dies wurde aber so eingezeichnet, dass der Text „B2“ noch im 2 Zustand und der Text „Z“ im dritten Zustand steht. Deshalb wurde das Schreiben des zweiten Bytes in das Z-Register im dritten Zustand ausgeführt. Hier wurde der vorherige Zustand aber kein leerer Zustand, da dort der Programmzähler um eins erhöht werden muss.

### Abänderungen zum Handbuch

Bei den beiden MVI-Befehlen soll im zweiten Maschinenzyklus ein weiteres Byte gelesen werden. Bei allen anderen Befehlen, die ein zweites Byte lesen wird im zweiten Zustand des zweiten Zyklus der Programmzähler um eins erhöht. Dies ist deswegen notwendig, da sonst das gleiche Byte nochmal gelesen wird, was in der Zustandsbeschreibung allerdings fehlt. Damit der MVI-Befehl aber korrekt ausgeführt werden kann, wird auch dort der Programmzähler erhöht. Ebenso wurde beim DCR r-Befehl, der einen Wert um eins reduziert soll, der Wert um eins erhöht. Dies wurde im Simulator angepasst.

### Letzter Maschinenzyklus bei Sprungbefehlen

Bei allen Sprungbefehlen, die der Intel 8080 unterstützt, wird zuletzt ein WZ OUT STATUS-Signal gesendet und der Programmzähler auf den um eins erhöhten Wert des zusammengesetzten W- und Z-Registers gesetzt. Diese zwei Zustände werden in einem realen Prozessor anstatt des normalen Fetches des neuen Befehls ausgeführt. Dies ist in diesem Simulator aber nicht so einfach möglich. Deswegen haben alle Sprungbefehle einen zusätzlichen Maschinenzyklus erhalten. Dort wird nur der Wert des 16-Bit WZ-Registers in den Programmzähler geschrieben. Während des Fetch des nächsten Befehls wird dann der Zähler um eins erhöht. Somit führt das Programm an der richtigen Stelle weitere Befehle aus.

### Kleine Eigenheiten oder Unterschiede zum Prozessor

- Es wird überall nur mit unsigned Werten gearbeitet. Zero-Flag zeigt Werte > 127 (negative Werte) an

- In der Beschreibung der Zustände werden teilweise Werte in die ALU geschrieben oder daraus gelesen (siehe Anhang A bei INR Befehl). Dies wird im Simulator nicht gemacht, da die arithmetischen Befehle nicht mehr in der ALU ausgeführt werden. Damit die Anzahl der Zustände wieder übereinstimmen, wurde der vierte Zustand in zwei aufgeteilt.
- Es gibt keinen Hold-Modus, da die Busse nicht vom Prozessor freigegeben werden müssen.

### 5.3 Laden und Anpassen der Oberfläche

Von Nico

Die Oberfläche für den Simulator wurde nicht dynamisch erzeugt. Das heißt es existiert eine statische „ui“- Datei für jede Oberfläche mit der der Nutzer interagieren kann. Insgesamt gibt es die folgenden drei:

- MainMenu.ui (Hauptmenü)
- Intel8080\_MainWindow.ui (Simulator)
- ChangeValueWindow.ui (Werte-Input)

Als Beispiel für eins dieser manuell erstellten Fenster ist in Abbildung 32 das Simulator-Fenster zu sehen.

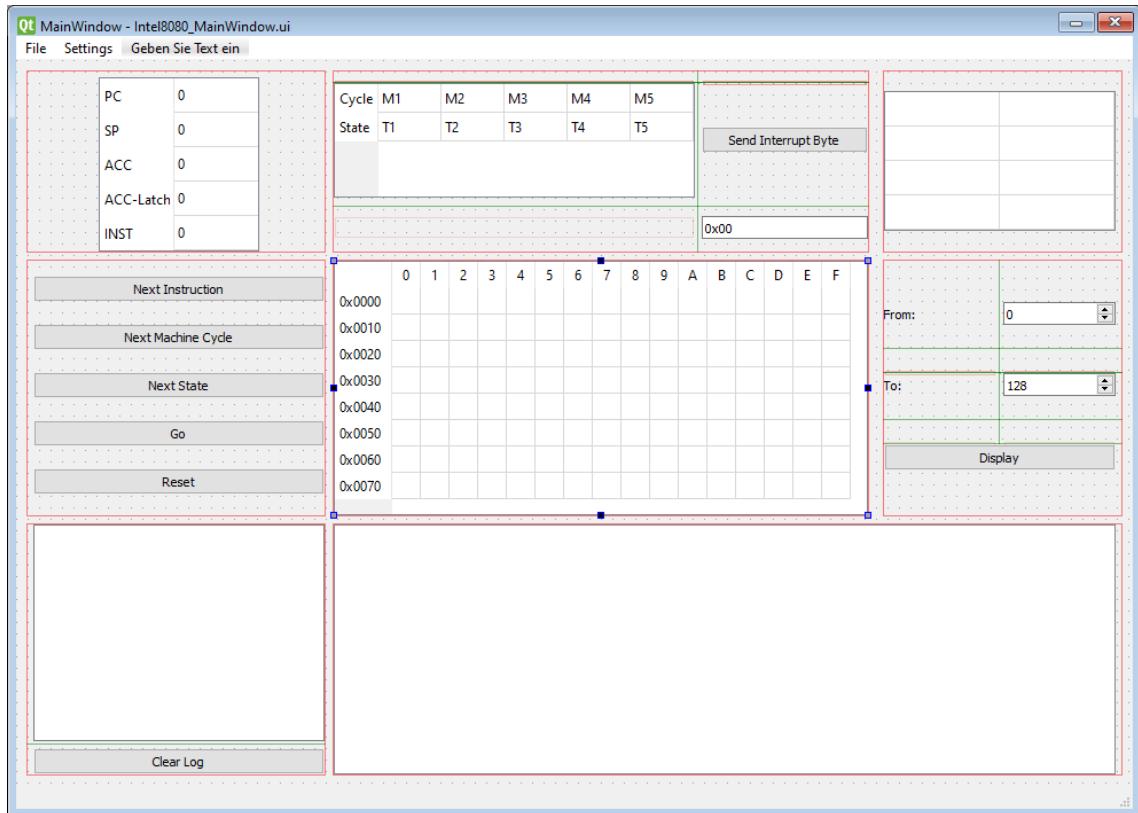


Abbildung 32: Rohes Simulator-Fenster

In diesem wird jedem Element ein Name zugewiesen, über den dann später darauf zugegriffen werden kann. Über Layouts kann außerdem die grobe Position eines einzelnen Elements oder einer Gruppe von Elementen bestimmt werden. Vorteil dabei ist, dass unveränderliche Eigenschaften wie Elementtyp (Tabelle, Textfeld, etc.), bestimmte Beschriftungen oder Elementanzahlen (Spalten, Reihen, etc.) bereits festgelegt werden können, bevor die Implementierung der Funktionalität vorgenommen werden muss. Dadurch ist es einerseits möglich, eine Änderung schnell visuell zu überprüfen, und andererseits wird an komplizierter Syntax für Fenster gespart, was den Programmcode übersichtlicher hält. Verglichen mit der letztlich dem Nutzer angezeigten Oberfläche (siehe Abbildung 3) fällt auf, dass die Größe einiger Elemente angepasst wurde und einige Knöpfe in die Zellen der Register-Elemente eingefügt wurden. Diese Anpassungen finden während der Initialisierung der „Intel8080\_MainWindow“-Klasse statt (siehe Abbildung 33).

```
self.init_ui("ui\\Intel8080_MainWindow.ui")
self.init_register_table()
self.init_register_array_table()
self.init_program_memory_table()
```

Abbildung 33: Init-Functioncalls in Instanzfunktion für Intel 8080 Fenster

```
def init_ui(self, ui_name):
    base_path = os.path.abspath("..")
    full_path = os.path.join(base_path, ui_name)
    loadUi(full_path, self)
```

Abbildung 34: Generische Funktion zum Laden der „ui“-Datei

„init \_ui“ ist hierbei eine generische Funktion die von allen Fenstern verwendet wird (siehe Abbildung 34). Aufgabe dieser ist es, den korrekten Pfad für die „ui“-Datei zu bestimmen und über den Funktionsaufruf „loadUI“ die Datei in ein leeres QMainWindow zu laden. Die Funktionen für die Initialisierung der verschiedenen Register sowie des Programmspeichers erfüllen effektiv die gleiche Aufgabe. Das Bearbeiten der Werte wird über Knöpfe geregelt. Diese werden über die Initialisierungsfunktionen erzeugt und in die einzelnen Zellen des Elements eingefügt. Weiterhin wird den Knöpfen die Funktion „pressed\_table\_cell“ zugewiesen, welche ausgeführt wird, sollte auf diese geklickt werden (um den Wert zu ändern). Ein Beispiel für eine der Funktionen, die diese Zuweisungen übernimmt, sieht man in Abbildung 35.

```
def init_program_memory_table(self):
    for row in range(self.ProgramMemory_table.rowCount()):
        for column in range(self.ProgramMemory_table.columnCount()):
            btn = QPushButton(self.ProgramMemory_table)
            btn.setText('{:x}'.format(0))
            self.ProgramMemory_table.setCellWidget(row, column, btn)
            btn.pressed.connect(self.pressed_table_cell)
```

Abbildung 35: Initialisierung des Programmspeichers

Effektiv wird die in der „ui“-Datei definierte Anzahl Zeilen und Spalten der Programmtablette abgefragt. Dann wird über jedes Element iteriert und ein Knopf erstellt mit Standardtext und einer Zuweisung. Die zugewiesene Funktion wird in Kapitel 5.4 noch näher erklärt.

## 5.4 GUI-Methoden

Von Nico

Die GUI dient als Schnittstelle zwischen Benutzer und Prozessorsimulation. Die in diesem Abschnitt erklärten Funktionen beziehen sich deshalb auf die in Kapitel 4.3 beschriebenen Funktionen für die Benutzeroberfläche.

### 5.4.1 Aktualisieren der Oberfläche

Nach der Ausführung jedes Zustandes besteht die Möglichkeit, dass sich ein in der Oberfläche angezeigter Wert ändert, beispielsweise der Programmzähler im zweiten Zustand jedes Befehls. Um die Oberfläche regelmäßig zu aktualisieren, wurden folgende Funktionen verwendet:

- `reload_registers_table`: Aktualisiert die besonderen Register wie Programmzähler, Instruction Register, etc.
- `reload_register_array_table`: Aktualisiert generische 8-Bit Register (b, c, d, e, etc.)
- `reload_memory_table`: Aktualisiert Programmspeicher
- `fill_program_table`: Aktualisiert Programmtabelle
- `color_program_table`: Färbt die derzeit ausgeführte Zeile der Programmtabelle
- `color_cycle_state`: Färbt aktuellen Maschinenzyklus und Zustand in CycleState-Tabelle

In der aktuellen Implementierung sind alle Funktionen jeweils so umgesetzt, dass alle relevanten Werte neu geladen werden, unabhängig davon, ob sie sich verändert haben oder nicht. Bei wenigen Werten, z.B. für die Register, stellt dies noch kein Problem dar, aber für den Programmspeicher (jeweils 128 8-Bit Register werden auf einmal ausgelesen) und vor allem für die Programmtabelle (komplettes Parsen und Filtern des Programmspeichers sowie Füllen der Tabelle) macht sich dies erheblich in der Performance bemerkbar. Da bisher noch kein Verfahren implementiert wurde, mit dem bei Bedarf selektiv aktualisiert werden kann, müssen bei jeder Zustandsänderung alle Funktionen für die Aktualisierung der Oberfläche ausgeführt werden. Dies geschieht über die „`update_ui`“-Funktion (siehe Abbildung 36).

```
def update_ui(self):
    self.fill_program_table()
    self.color_program_table()
    self.color_cycle_state()
    self.reload_memory_table()
    self.reload_registers_table()
    self.reload_register_array_table()
```

Abbildung 36: update\_ui Funktion

Die „update\_ui“-Funktion wird innerhalb der Klasse für die Simulatoroberfläche an mehreren Stellen verwendet:

- Zu Beginn bei Initialisierung der Oberfläche
- Bei Laden eines neuen Programms über die Benutzeroberfläche
- Bei Zurücksetzen des Simulators über den Reset-Knopf
- Bei automatischen Ausführen des Programms über den Go-Knopf (Aktualisieren nach jeder Instruction)
- Nach Ausführen einer Instruction, eines Maschinenzyklus oder eines Zustandes über die jeweiligen Knöpfe

Wie bereits erwähnt, ist der Sinn dieser Maßnahme, dass keine Veränderung der angezeigten Aspekte des Simulators übersehen wird. Schlimmstenfalls könnte es auch dazu führen, dass wenn eine Anzeige nicht aktualisiert wird, dieser fehlerhafte Wert nach Manipulieren eines anderen fälschlicherweise in die internen Register des Simulators zurückgeschrieben wird. Dieses Prinzip sieht man in Abbildung 37.

```
def perform_instruction(self):
    if self.actionCheck():
        self.processor.next_instruction()
        self.update_ui()

def perform_mc(self):
    if self.actionCheck():
        self.processor.next_machine_cycle()
        self.update_ui()

def perform_state(self):
    if self.actionCheck():
        self.processor.next_state()
        self.update_ui()
```

Abbildung 37: Programmschritte Instruction/Cycle/State

Die Implementierungen der drei Knöpfe zum Ausführen eines Befehl, eines Maschinenzyklus oder eines Zustandes sind lediglich Anweisungen an den Simulator, diese abzuarbeiten, woraufhin über „update\_ui“ die gesammelten Informationen wieder in die Benutzeroberfläche geladen werden. Ähnlich wie die „update\_ui“-Funktion alle Elemente aktualisiert, arbeiten auch die einzelnen Funktionen darin. Als Beispiel dafür ist in Abbildung 38 die Funktion zum aktualisieren der Oberfläche für die speziellen Register zu sehen.

```
def reload_registers_table(self): # This functions makes the ui match the registers
    Processor = self.processor
    Registers_table = self.Registers_table

    Registers_table.cellWidget(0, 0).setText(str(Processor.get_pc())) # PC
    Registers_table.cellWidget(1, 0).setText(str(Processor.get_sp())) # SP
    Registers_table.cellWidget(2, 0).setText(str(Processor.get_acc())) # ACC
    Registers_table.cellWidget(3, 0).setText(str(Processor.get_act())) # Temp-ACC
    Registers_table.cellWidget(4, 0).setText(str(Processor.get_instruction_reg())) # INST
```

Abbildung 38: Special Register Refresh

Da es schwer nachzuvollziehen ist wann ein Bestandteil eines Elements des Simulators seinen Zustand ändert (beispielsweise ein einzelnes Register im Speicher), liest die Funktion einfach über die im Simulator definierte Schnittstellen alle für das Element relevanten Werte aus und trägt sie im korrekten Format in die Tabelle ein. Dieses Verhalten gilt auch in ähnlicher Weise für die „reload\_register\_array\_table“- und die „reload\_program\_memory\_table“-Funktion. Etwas anders funktioniert die Funktion, welche den Programmspeicher parst („fill\_program\_table“), um das Programm übersichtlich mit der entsprechenden Bedeutung des Bytebefehls darzustellen (siehe Abbildung 39).

```

def fill_program_table(self):
    self.instruction_positions = []
    self.Program_table.setRowCount(0) # Clear Table
    i = 0
    nop_counter = 0
    operands = 0
    while i < pow(2, 16):
        if (self.processor.program[i] == 0) and nop_counter < 1:
            self.instruction_positions.append(i)
        elif self.processor.program[i] != 0:
            self.instruction_positions.append(i)
        for j in range(len(command_masks)):
            masked_command = self.processor.program[i] & command_masks[j]
            if masked_command in command_dict:
                if mask_validation(command_dict[masked_command], j):
                    if masked_command == 0x00:
                        nop_counter += 1
                    else:
                        nop_counter = 0
                    if not (nop_counter > 1):
                        operands = command_dict[masked_command][0]
                        row = self.Program_table.rowCount()
                        self.Program_table.insertRow(row)
                        self.Program_table.setItem(row, 0, QTableWidgetItem(""))
                        ItemText = str(i)
                        ItemText = ItemText + " " + command_dict[masked_command][1]
                        ItemText = ItemText + " " + hex(self.processor.program[i])
                        for k in range(i, i + operands):
                            ItemText = ItemText + " " + hex(self.processor.program[k + 1])
                        self.Program_table.setItem(row, 1, QTableWidgetItem(ItemText))
                        break
        i += operands + 1

```

Abbildung 39: Program Table Function

Zuerst werden alle Einträge der Tabelle gelöscht, woraufhin über alle Register im Programmspeicher iteriert wird. Da in jedem Eintrag der Tabelle ein Befehl sowie dessen Operanden stehen sollen, wird über eine Liste (instruction\_positions) der Index jedes Befehls gespeichert. Dadurch soll sichergestellt werden, dass Operanden in dieselbe Zeile wie der zugehörige Befehl geschrieben werden. Die Auswahl des Befehls sowie der Anzahl Operanden pro Befehl geschieht über ein Dictionary. In der Abfrage werden die Masken für jeden Befehl mit dem Bytecode im Programmspeicher verundet und der resultierende Eintrag im Dictionary gefunden (siehe Abbildung 40).

```

command_dict = {
    0xCE: [1, "ACI"], # ACI
    0x88: [0, "ADC", 0xF8], # ADC BIT 0-2 REG --> 0xF8
}

```

Abbildung 40: Dictionary Ausschnitt

Die Einträge sind hierbei die Anzahl der nachfolgenden Operanden, die Bezeichnung des Befehls und, falls angewendet, die Maske zur Bestimmung des Befehls. Um keine zufälligen Treffer zu erzielen (beispielsweise verunden mit falscher Maske führt zu anderem Befehl) wird nach einer Übereinstimmung im Dictionary die „mask\_validation“-Funktion verwendet (siehe Abbildung 41).

```
def mask_validation(dict_entry, mask_index):
    if len(dict_entry) > 2: # Does the command have a mask?
        if dict_entry[2] == command_masks[mask_index]: # Is the used masked the correct mask?
            return True
        else:
            return False
    else:
        if mask_index == 0: # If not, is the chosen mask the standard mask?
            return True
        else:
            return False
```

Abbildung 41: Mask-Validation-Function

Hierbei wird zuerst überprüft, ob der vermeintliche Befehl eine Maske verwendet und ob die angewendete Maske die korrekte Maske ist. Zusätzlich wird überprüft, ob keine Maske verwendet wurde, falls der Befehle keine erforderlich. Ist das Ergebnis negativ, wird die Übereinstimmung verworfen. Bei einem gültigen Treffer wird ein Eintrag in der Tabelle angelegt, der die Stelle im Speicher, den Namen des Befehls sowie dessen Operanden umfasst. Dabei wird berücksichtigt, ob es sich um einen NOP-Befehl handelt. Da leere Speicherstellen als solche interpretiert werden, werden diese Einträge zusammengefasst, sofern mehrere aufeinanderfolgen. Dies wird mit einem einfachen Zähler erreicht. Am Ende der Schleife werden die Anzahl Operanden auf die Zählvariable „i“ addiert, um die Einträge der Operanden zu überspringen.

#### 5.4.2 Manipulieren der Simulation

Das Zurückschreiben von der GUI in den Simulator geschieht über drei verschiedene Funktionen:

- update\_registers\_table (PC, Instr, ACC, etc.)
- update\_register\_array\_table (b, c, d, e, etc.)
- update\_memory\_table (Programmspeicher)

Diese funktionieren ähnlich wie deren Initialisierungsfunktionen (siehe Kapitel 5.3). Dabei wird über jede gültige Zelle der Tabelle iteriert, der Wert herausgeschrieben und über eine gültige Funktion in den Intel8080-Simulator übernommen. Als Beispiel

dafür ist in Abbildung 42 die konkrete Implementierung für das Zurückschreiben des Programmspeichers zu sehen.

```
def update_memory_table(self): # make program memory match the ui
    StartValue = self.From_sb.value()
    for i in range(8):
        for j in range(16):
            address = StartValue + i * 16 + j # one row = 16 bits, one column 1 bit
            value = int(self.ProgramMemory_table.cellWidget(i, j).text(), 16)
            self.processor.set_memory_byte(address, value)
```

Abbildung 42: Memory-Update-Function

Zuerst wird bestimmt, welcher Speicherbereich derzeit angezeigt wird. Da die Programmspeichertabelle immer genau acht Zeilen und sechzehn Spalten hat (also 128 Register), können diese Werte fix verwendet werden. Daraufhin wird für jede Zeile die Adresse berechnet. Der Wert wird aus den Knöpfen bezogen und zur Basis 10 umgerechnet, da diese in Hexadezimal-Darstellung als String in diesen gespeichert werden. Über die Prozessorfunktion „set\_memory\_byte“ werden die Werte dann einzeln zurückgeschrieben. Die drei „update“-Funktionen werden im Gegensatz zu den „reload“-Funktionen nur aufgerufen, wenn tatsächlich ein Wert im jeweiligen Element ändert. Dabei ist die bereits angesprochene Funktion, welche mit den Knopfelementen dieser Tabellen verknüpft ist relevant (siehe 43).

```
def pressed_table_cell(self):
    btn = self.sender()
    self.dialog = ChangeValueWindow(self, btn)
    self.dialog.show()
```

Abbildung 43: Knopfdruckfunktion

Dabei wird der Knopf, welcher die Funktion aufruft als Referenz an das neu erzeugte ChangeValueWindow übergeben. Dieser wird benötigt, um zu bestimmen welches Element eine Werteänderung vornimmt. In Abbildung 44 ist die Funktion des ChangeValueWindow zu sehen, welche die Eingabevervalidierung vornimmt und die für das Element entsprechende „update“-Funktion auruft.

```
def check_input(self):
    value = int(self.lineEdit_value.text(), 16)
    index = self.btn.parent().parent().indexAt(self.btn.pos())
    if self.btn.parent().parent() == self.Intel8080_MainWindow.Registers_table:
        # self.setWindowTitle(self.)
        if 0 <= value < 256: # May need to be different, depending on register
            self.btn.parent().parent().cellWidget(index.row(), index.column()).setText(
                self.lineEdit_value.text())
            self.Intel8080_MainWindow.update_registers_table()
            self.close()
    elif self.btn.parent().parent() == self.Intel8080_MainWindow.Register_array_table:
        if 0 <= value < 256: # May need to be different, depending on register
            self.btn.parent().parent().cellWidget(index.row(), index.column()).setText(
                self.lineEdit_value.text())
            self.Intel8080_MainWindow.update_register_array_table()
            self.close()
    elif self.btn.parent().parent() == self.Intel8080_MainWindow.ProgramMemory_table:
        if 0 <= value < 256:
            self.btn.parent().parent().cellWidget(index.row(), index.column()).setText(
                self.lineEdit_value.text())
            self.Intel8080_MainWindow.update_memory_table()
            self.close()
```

Abbildung 44: Input-Validation-Function

Als erstes wird der Wert ausgelesen und der Index des Knopfes innerhalb der Tabelle bestimmt. Danach wird geprüft, aus welchem Element der aufrufende Knopf stammt.

#### 5.4.3 Sonstige Interaktionen

Da das gesamte Auslesen und Anzeigen der 65.536 8-Bit Register zu lange dauern würde und selten notwendig ist, wurde die Anzeige der Programmspeichertabelle auf 128 aufeinanderfolgende Register limitiert. Um auswählen zu können, welche Register angezeigt werden, sind zwei Eingabefenster vorgesehen. Entweder um anzugeben bis zu oder ab welchem Register der Speicher angezeigt werden soll. Das Ändern des einen Wertes passt dabei den anderen automatisch an.

```
def adjust_to(self, value):
    if self.lock is False:
        self.lock = True
        if value % 16 != 0:
            self.From_sb.setValue(value - (value % 16))
            self.To_sb.setValue(value + 128 - (value % 16))
        else:
            self.To_sb.setValue(value + 128)
        self.lock = False
```

Abbildung 45: Range-Funktion

Dafür wird ein Signal, welches ausgelöst wird, sobald sich der Wert ändert mit der in Abbildung 45 gezeigten Funktion verknüpft. Um zu verhindern, dass die beiden

Funktionen sich in einer Endlosschleife gegenseitig aufrufen wird ein Lock verwendet. Die Oberfläche wird nicht sofort aktualisiert, dafür muss zuerst der darunterliegende Knopf gedrückt werden oder der Zustand des Simulators verändert werden.

## 5.5 Exe-Erstellung

Als letzter Schritt wurde für das Programm noch eine „exe“-Datei erstellt. Dafür wurde das „pyinstaller“ [12] Paket verwendet. Sollte keine aktuelle Version des Programms als ausführbare Datei vorliegen wird eine Python Installation (Version 3.9 oder 3.10 getestet und funktionieren) und das PyQt6 Paket [13] benötigt.

## 6 Fazit und Ausblick

Von Nico

### 6.1 Zusammenfassung

Das Ergebnis dieser Studienarbeit ist ein Simulator für den Intel 8080 welcher mit einer zugehörigen Benutzeroberfläche bedient werden kann. Die relevanten Aspekte des Prozessors werden dem Nutzer dabei angezeigt, wie beispielsweise Registerinhalte, das ausgeführte Programm oder einzelne Operationen für jeden Zustand. Es ist dem Nutzer möglich eigene Programme über die Benutzeroberfläche in den Simulator einzufügen oder diese zuvor über den verwendeten Assembler zu erstellen und dann direkt einzulesen. Die eingefügten Programme können Schrittweise ausgeführt werden, wobei es möglich ist ganze Befehle, Maschinenzyklen oder sogar einzelne Zustände auszuführen. Es ist möglich über den eingefügten Programmeditor eigene Programme mit dem verwendeten Assembler zu schreiben. Dabei wird eine Hilfestellung gegeben für die korrekte Syntax. Die Bedeutung einzelner Zustände wird in einem Befehlslog eingetragen um nachvollziehen zu können was in jedem Zustand passiert. Programme können automatisch ausgeführt werden und über Haltepunkte auch automatisch an einer beliebigen Stelle angehalten werden. Es ist zu jedem Zeitpunkt möglich die internen Aspekte des Simulators zu verändern über eine einfach zu bedienende Oberfläche. Das zurücksetzen der Oberfläche funktioniert auch problemlos. Der gesamte Befehlssatz des Intel 8080 wurde so genau wie möglich implementiert und sollte identische Ergebnisse liefern wie nativ ausgeführte Operationen.

### 6.2 Ausblick

Die Benutzeroberfläche ist derzeit vollständig in Englisch geschrieben. In Zukunft könnten noch mehr Sprachen eingebaut werden. Der Befehlslog ist aktuell noch recht kryptisch. Während es wenn man die Bedeutung der Zustände und Abkürzung kennt zwar genügend Aufschluss gibt, ist diese Variante noch Verbesserungswürdig. Das Tutorial vermittelt die notwendigen Grundlagen um den Simulator zu bedienen, aber eine interaktive Lösung mit Beispielen hätte vermutlich einen höheren Lernerfolg.

## Literatur

- [1] Instruction formats: <https://www.geeksforgeeks.org/computer-organization-instruction-formats-zero-one-two-three-address-instruction/>, zuletzt abgerufen: 25.12.2021
- [2] Von-Neumann vs. Harvard: Studyflix, <https://studyflix.de/informatik/von-neumann-vs-harvard-786>, zuletzt abgerufen: 15.05.2022
- [3] Von Neumann Architektur: <https://de.wikipedia.org/wiki/Von-Neumann-Architektur>, zuletzt abgerufen: 15.05.2022
- [4] Grundlagen der Informatik: Herold, Lurz, Wohlrab und Hopf; 3. aktualisierte Auflage (2017), Pearson
- [5] Some Computer Organizations and Their Effectiveness (S.948-960, vol. C-21): M. J. Flynn, 1972, in IEEE Transactions on Computers
- [6] Intel 8080 Microcomputer Systems User's Manual (S.16 (2-2)): Intel Corporation, 1975, [http://bitsavers.trailing-edge.com/components/intel/MCS80/98-153B\\_Intel\\_8080\\_Microcomputer\\_Systems\\_Users\\_Manual\\_197509.pdf](http://bitsavers.trailing-edge.com/components/intel/MCS80/98-153B_Intel_8080_Microcomputer_Systems_Users_Manual_197509.pdf) zuletzt abgerufen: 14.05.2022
- [7] Intel 8080 Microcomputer Systems User's Manual (S.30 ff. (2-16)): Intel Corporation, 1975, [http://bitsavers.trailing-edge.com/components/intel/MCS80/98-153B\\_Intel\\_8080\\_Microcomputer\\_Systems\\_Users\\_Manual\\_197509.pdf](http://bitsavers.trailing-edge.com/components/intel/MCS80/98-153B_Intel_8080_Microcomputer_Systems_Users_Manual_197509.pdf) zuletzt abgerufen: 14.05.2022
- [8] Intel 8080 Microcomputer Systems User's Manual (S.46 ff. (4-1)): Intel Corporation, 1975, [http://bitsavers.trailing-edge.com/components/intel/MCS80/98-153B\\_Intel\\_8080\\_Microcomputer\\_Systems\\_Users\\_Manual\\_197509.pdf](http://bitsavers.trailing-edge.com/components/intel/MCS80/98-153B_Intel_8080_Microcomputer_Systems_Users_Manual_197509.pdf) zuletzt abgerufen: 14.05.2022
- [9] Intel 8080 Microcomputer Systems User's Manual (S.18 f. (2-4)): Intel Corporation, 1975, [http://bitsavers.trailing-edge.com/components/intel/MCS80/98-153B\\_Intel\\_8080\\_Microcomputer\\_Systems\\_Users\\_Manual\\_197509.pdf](http://bitsavers.trailing-edge.com/components/intel/MCS80/98-153B_Intel_8080_Microcomputer_Systems_Users_Manual_197509.pdf) zuletzt abgerufen: 14.05.2022
- [10] Intel 8080 Assembler: Paolo Amoroso, 2021, <https://github.com/pamoroso/suite8080/tree/56c2e077dd3d178859d614932b9a1f33fa238e21>, zuletzt abgerufen: 15.05.2022
- [11] Intel 8080 Microcomputer Systems User's Manual (S.20 (2-6)): Intel Corporation, 1975, [http://bitsavers.trailing-edge.com/components/intel/MCS80/98-153B\\_Intel\\_8080\\_Microcomputer\\_Systems\\_Users\\_Manual\\_197509.pdf](http://bitsavers.trailing-edge.com/components/intel/MCS80/98-153B_Intel_8080_Microcomputer_Systems_Users_Manual_197509.pdf)

/98-153B\_Intel\_8080\_Microcomputer\_Systems\_Users\_Manual\_197509.pdf  
zuletzt abgerufen: 14.05.2022

[12] Pyinstaller Manual, <https://pyinstaller.org/en/stable/>, zuletzt abgerufen: 16.05.2022

[13] PyQt Reference Guide, <https://www.riverbankcomputing.com/static/Docs/PyQt6/index.html>, zuletzt abgerufen: 15.05.2022

## Anhang

MNEMONIC	OP CODE		M1[1]					M2			
	D <sub>7</sub>	D <sub>6</sub> D <sub>5</sub> D <sub>4</sub>	D <sub>3</sub> D <sub>2</sub> D <sub>1</sub> D <sub>0</sub>	T1	T2[2]	T3	T4	T5	T1	T2[2]	T3
MOV r1, r2	0 1	D D	D S S S	PC OUT STATUS	PC = PC +1	INST→TMP/IR	(SSS)→TMP	(TMP)→DDD			
MOV r, M	0 1	D D	D 1 1 0		↑	↑	↑	x[3]		HL OUT STATUS[6]	DATA → DDD
MOV M, r	0 1	1 1	0 S S S				(SSS)→TMP			HL OUT STATUS[7]	(TMP) → DATA BUS
SPHL	1 1	1 1	1 0 0 1				(HL)	SP			
MVI r, data	0 0	D D	D 1 1 0				X		PC OUT STATUS[6]	B2 → DDD	
MVI M, data	0 0	1 1	0 1 1 0				X			B2 → TMP	
LXI rp, data	0 0	R P	0 0 0 1				X			PC = PC +1	B2 → r1
LDA addr	0 0	1 1	1 0 1 0				X			PC = PC +1	B2 → Z
STA addr	0 0	1 1	0 0 1 0				X			PC = PC +1	B2 → Z
LHLD addr	0 0	1 0	1 0 1 0				X			PC = PC +1	B2 → Z
SHLD addr	0 0	1 0	0 0 1 0				X		PC OUT STATUS[6]	PC = PC +1	B2 → Z
LDAX rp <sup>[4]</sup>	0 0	R P	1 0 1 0				X		rp OUT STATUS[6]		DATA → A
STAX rp <sup>[4]</sup>	0 0	R P	0 0 1 0				X		rp OUT STATUS[7]		(A) → DATA BUS
XCHG	1 1	1 0	1 0 1 1				(HL)↔(DE)				
ADD r	1 0	0 0	0 S S S				(SSS)→TMP (A)→ACT		[9]		(ACT)+(TMP)→A
ADD M	1 0	0 0	0 1 1 0				(A)→ACT		HL OUT STATUS[6]		DATA → TMP
ADI data	1 1	0 0	0 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC +1	B2 → TMP
ADC r	1 0	0 0	1 S S S				(SSS)→TMP (A)→ACT		[9]		(ACT)+(TMP)+CY→A
ADC M	1 0	0 0	1 1 1 0				(A)→ACT		HL OUT STATUS[6]		DATA → TMP
ACI data	1 1	0 0	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC +1	B2 → TMP
SUB r	1 0	0 1	0 S S S				(SSS)→TMP (A)→ACT		[9]		(ACT)-(TMP)→A
SUB M	1 0	0 1	0 1 1 0				(A)→ACT		HL OUT STATUS[6]		DATA → TMP
SUI data	1 1	0 1	0 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC +1	B2 → TMP
SBB r	1 0	0 1	1 S S S				(SSS)→TMP (A)→ACT		[9]		(ACT)-(TMP)-CY→A
SBB M	1 0	0 1	1 1 1 0				(A)→ACT		HL OUT STATUS[6]		DATA → TMP
SBI data	1 1	0 1	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC +1	B2 → TMP
INR r	0 0	D D	D 1 0 0				(DDD)→TMP (TMP)+1→ALU	ALU→DDD			
INR M	0 0	1 1	0 1 0 0				X		HL OUT STATUS[6]		DATA (TMP)+1 → ALU
DCR r	0 0	D D	D 1 0 1				(DDD)→TMP (TMP)+1→ALU	ALU→DDD			
DCR M	0 0	1 1	0 1 0 1				X		HL OUT STATUS[6]		DATA (TMP)-1 → ALU
INX rp	0 0	R P	0 0 1 1				(RP)+1	RP			
DCX rp	0 0	R P	1 0 1 1				(RP)-1	RP			
DAD rp <sup>[8]</sup>	0 0	R P	1 0 0 1				X		(ri)→ACT	(L)→TMP, (ACT)+(TMP)→ALU	ALU→L, CY
DAA	0 0	1 0	0 1 1 1				DAA→A, FLAGS <sup>[10]</sup>				
ANA r	1 0	1 0	0 S S S		↓	↓	↓	(SSS)→TMP (A)→ACT	[9]		(ACT)+(TMP)→A
ANA M	1 0	1 0	0 1 1 0	PC OUT STATUS	PC = PC +1	INST→TMP/IR	(A)→ACT		HL OUT STATUS[6]		DATA → TMP

M3			M4			M5				
T1	T2[2]	T3	T1	T2[2]	T3	T1	T2[2]	T3	T4	T5
HL OUT STATUS[7]	(TMP) → DATA BUS									
PC OUT STATUS[6]	PC = PC + 1	B3 → rh								
	PC = PC + 1	B3 → W	WZ OUT STATUS[6]	DATA → A						
	PC = PC + 1	B3 → W	WZ OUT STATUS[7]	(A) → DATA BUS						
	PC = PC + 1	B3 → W	WZ OUT STATUS[6]	DATA → L	WZ OUT STATUS[6]	DATA → H				
PC OUT STATUS[6]	PC = PC + 1	B3 → W	WZ OUT STATUS[7]	(L) → DATA BUS	WZ OUT STATUS[7]	(H) → DATA BUS				
[9]	(ACT)+(TMP)→A									
[9]	(ACT)+(TMP)→A									
[9]	(ACT)+(TMP)+CY→A									
[9]	(ACT)+(TMP)+CY→A									
[9]	(ACT)−(TMP)→A									
[9]	(ACT)−(TMP)→A									
[9]	(ACT)−(TMP)−CY→A									
[9]	(ACT)−(TMP)−CY→A									
HL OUT STATUS[7]	ALU → DATA BUS									
HL OUT STATUS[7]	ALU → DATA BUS									
(rh)→ACT	(H)→TMP (ACT)+(TMP)+CY→ALU	ALU→H, CY								
[9]	(ACT)+(TMP)→A									

MNEMONIC	OP CODE		M1 <sup>[1]</sup>					M2		
	D <sub>7</sub> D <sub>6</sub> D <sub>5</sub> D <sub>4</sub>	D <sub>3</sub> D <sub>2</sub> D <sub>1</sub> D <sub>0</sub>	T1	T2 <sup>[2]</sup>	T3	T4	T5	T1	T2 <sup>[2]</sup>	T3
ANI data	1 1 1 0	0 1 1 0	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	(A)→ACT		PC OUT STATUS[6]	PC = PC + 1	B2 → TMP
XRA r	1 0 1 0	1 S S S		↑	↑	↑	(A)→ACT (SSS)→TMP		[9]	(ACT)+(TMP)→A
XRA M	1 0 1 0	1 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA	→ TMP
XRI data	1 1 1 0	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1	B2 → TMP
ORA r	1 0 1 1	0 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)+(TMP)→A	
ORA M	1 0 1 1	0 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA	→ TMP
ORI data	1 1 1 1	0 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1	B2 → TMP
CMP r	1 0 1 1	1 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)-(TMP), FLAGS	
CMP M	1 0 1 1	1 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA	→ TMP
CPI data	1 1 1 1	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1	B2 → TMP
RLC	0 0 0 0	0 1 1 1				(A)→ALU ROTATE		[9]	ALU→A, CY	
RRC	0 0 0 0	1 1 1 1				(A)→ALU ROTATE		[9]	ALU→A, CY	
RAL	0 0 0 1	0 1 1 1				(A), CY→ALU ROTATE		[9]	ALU→A, CY	
RAR	0 0 0 1	1 1 1 1				(A), CY→ALU ROTATE		[9]	ALU→A, CY	
CMA	0 0 1 0	1 1 1 1				(A)→A				
CMC	0 0 1 1	1 1 1 1				CY→CY				
STC	0 0 1 1	0 1 1 1				1→CY				
JMP addr	1 1 0 0	0 0 1 1				X		PC OUT STATUS[6]	PC = PC + 1	B2 → Z
J cond addr[17]	1 1 C C	C 0 1 0				JUDGE CONDITION		PC OUT STATUS[6]	PC = PC + 1	B2 → Z
CALL addr	1 1 0 0	1 1 0 1				SP = SP - 1		PC OUT STATUS[6]	PC = PC + 1	B2 → Z
C cond addr[17]	1 1 C C	C 1 0 0				JUDGE CONDITION IF TRUE, SP = SP - 1		PC OUT STATUS[6]	PC = PC + 1	B2 → Z
RET	1 1 0 0	1 0 0 1				X		SP OUT STATUS[15]	SP = SP + 1	DATA → Z
R cond addr[17]	1 1 C C	C 0 0 0			INST→TMP/IR	JUDGE CONDITION[14]		SP OUT STATUS[15]	SP = SP + 1	DATA → Z
RST n	1 1 N N	N 1 1 1			φ→W INST→TMP/IR	SP = SP - 1		SP OUT STATUS[16]	SP = SP - 1	(PCH) → DATA BUS
PCHL	1 1 1 0	1 0 0 1			INST→TMP/IR	(HL) → PC				
PUSH rp	1 1 R P	0 1 0 1				SP = SP - 1		SP OUT STATUS[16]	SP = SP - 1	(rh) → DATA BUS
PUSH PSW	1 1 1 1	0 1 0 1				SP = SP - 1		SP OUT STATUS[16]	SP = SP - 1	(A) → DATA BUS
POP rp	1 1 R P	0 0 0 1				X		SP OUT STATUS[15]	SP = SP + 1	DATA → r1
POP PSW	1 1 1 1	0 0 0 1				X		SP OUT STATUS[15]	SP = SP + 1	DATA → FLAGS
XTHL	1 1 1 0	0 0 1 1				X		SP OUT STATUS[15]	SP = SP + 1	DATA → Z
IN port	1 1 0 1	1 0 1 1				X		PC OUT STATUS[6]	PC = PC + 1	B2 → Z, W
OUT port	1 1 0 1	0 0 1 1				X		PC OUT STATUS[6]	PC = PC + 1	B2 → Z, W
EI	1 1 1 1	1 0 1 1				SET INTEN F/F				
DI	1 1 1 1	0 0 1 1				RESET INTEN F/F				
HLT	0 1 1 1	0 1 1 0		↓	↓	↓	X	PC OUT STATUS	HALT MODE[20]	
NOP	0 0 0 0	0 0 0 0	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	X				

M3			M4			M5					
T1	T2[2]	T3	T1	T2[2]	T3	T1	T2[2]	T3	T4	T5	
[9]	(ACT)+(TMP)→A										
[9]	(ACT)+(TMP)→A										
[9]	(ACT)+(TMP)→A										
[9]	(ACT)+(TMP)→A										
[9]	(ACT)+(TMP)→A										
[9]	(ACT)-(TMP); FLAGS										
[9]	(ACT)-(TMP); FLAGS										
PC OUT STATUS[6]	PC = PC + 1      B3 → W										WZ OUT STATUS[11]      (WZ) + 1 → PC
PC OUT STATUS[6]	PC = PC + 1      B3 → W										WZ OUT STATUS[11,12]      (WZ) + 1 → PC
PC OUT STATUS[6]	PC = PC + 1      B3 → W	SP OUT STATUS[16]	(PCH) — SP = SP - 1 → DATA BUS	SP OUT STATUS[16]	(PCL) — DATA BUS						WZ OUT STATUS[11]      (WZ) + 1 → PC
PC OUT STATUS[6]	PC = PC + 1      B3 → W[13]	SP OUT STATUS[16]	(PCH) — SP = SP - 1 → DATA BUS	SP OUT STATUS[16]	(PCL) — DATA BUS						WZ OUT STATUS[11,12]      (WZ) + 1 → PC
SP OUT STATUS[15]	SP = SP + 1      DATA → W										WZ OUT STATUS[11]      (WZ) + 1 → PC
SP OUT STATUS[15]	SP = SP + 1      DATA → W										WZ OUT STATUS[11,12]      (WZ) + 1 → PC
SP OUT STATUS[16]	(TMP = 00NNN000) → Z      (PCL) → DATA BUS										WZ OUT STATUS[11]      (WZ) + 1 → PC
SP OUT STATUS[16]	(rl) → DATA BUS										
SP OUT STATUS[16]	FLAGS → DATA BUS										
SP OUT STATUS[15]	SP = SP + 1      DATA → rh										
SP OUT STATUS[15]	SP = SP + 1      DATA → A										
SP OUT STATUS[15]	DATA → W	SP OUT STATUS[16]	(H) → DATA BUS	SP OUT STATUS[16]	(L) → DATA BUS	(WZ) → HL					
WZ OUT STATUS[18]	DATA → A										
WZ OUT STATUS[18]	(A) → DATA BUS										