DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Konzeptionierung eines Simulators für 8-bit Prozessoren

Studienarbeit

Bachelor of Science

Studiengang Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Andreas Schmider, Nico Schrodt

Abgabedatum 12. 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 8-bit Prozessoren

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, 12. Mai 2022

Ort, Datum Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

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

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 mehrere unterschiedliche 8-Bit Prozessoren simuliert werden können. Dazu sollen die Grundlegenden Eigenschaften in kurzen Lernprogrammen erläutert werden. Ebenfalls soll es eine interaktive Einweisung geben wie der Simulator verwendet werden kann.

1.2 Zeitplan

Platzhalter

1.3 Theoretische Grundlagen

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

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 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.

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 keine Adressen benötigt zum ausführen von Operationen.

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 CISC und RISC

Von Andreas komplettes Kapitel aus Buch
Ein Prozessor unterstützt immer nur eine gewisse Menge an Befehlen, diese werden 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 haben.

Bei CISC-Prozessoren wird versucht soviel wie möglich in einem Befehl ausführen zu können. So gibt es viele verschieden 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. Der Nachteil bei 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 kleinem 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 Re-

gister die frei verwendbar sind und nicht für speziellen Operationen bestimmt sind. Ebenso können die meisten Befehle in nur einem einzigen Arbeitsschritt ausgeführt werden.

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

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

1.3.4 Parallelität nach Flynn

Von Andreas
1966 wurden von Micheal Flynn die folgenden vier Arten von Parallelisierung eingeführt [Flynn 949]:

- 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 Daten-Strom abgearbeitet.
SIMD
Bei SIMD werden Pipelines eingesetzt, die es ermöglichen mehrere korrekte Abfolgen von Programmbefehlen auszuführen. So können unterschiedliche Programme in sich selber 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 um 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 Daten-Strom 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.

1.4 Instruction Sets

Von Andreas
Die meisten Prozessoren besitzen Befehle aus den folgenden fünf Gruppen:

- Daten Transfer Gruppe
- Arithmetik Gruppe
- Logische Gruppe
- Verzweigungs Gruppe
- Stapel, Ein- Ausgänge und Maschinen Kontroll- Gruppe

Die Befehle der Daten Transfer Gruppe bewegen Daten zwischen Registern und/oder Speicher wie zum Beispiel mit den MOVE Befehlen. In der Arithmetik Gruppe werden, wie der Name schon sagt, Befehle mit arithmetische Operationen wie Addition verwendet. In der Logischen Gruppen werden Operation wie das logische Oder verwendet. In der Zweig Gruppe gibt es die Befehle, welche den Standardmäßigen Programmfluss ändern und das Programm nicht zwangsläufig in der nächsten Zeile fortgesetzt wird wie bei bedingten Sprüngen. In der letzten Gruppe liegen die Befehle, die Eingänge und Ausgänge beachten oder den Stack bearbeiten. [Intel 8080 S.46 (4-1)]

2 Projektplanung

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 von den einzelnen 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 aber diese sind für den Anwender kaum spürbar. Der größte Unterschied sind die verschiedenen Befehlssätze. Die meisten Prozessoren verwenden für die gleichen Befehle, unterschiedliche Mnemonics und Bitkombinationen. Sobald aber die Befehle decodiert wurden, läuft im Prozessor meistens sehr Ähnliches oder sogar das Gleiche ab. Dabei kann es vorkommen, dass ein Prozessor noch eine 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, sind alle sehr ähnlich, wenn nicht 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 nur sicher gestellt, dass die zwangsweise notwendigen Eigenschaften simuliert werden. Dazu zählt, dass die Inhalte von Registern, die vom Anwender verwendet werden können, nach den 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 Maschinen Zyklen. Jeder dieser Maschinen Zyklen besteht wiederum aus mindestens drei und maximal fünf 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.

2.1 Multi-Prozessor-Simulator

Mit diesem Simulator soll es möglich sein mehrere unterschiedliche Prozessoren simulieren zu können. Vor der Implementierung des Simulators müssen einer oder mehrere geeignete Prozessoren ausgewählt werden. In den nachfolgenden Kapiteln werden einige potenzielle Kandidaten näher analysiert und beschrieben.

2.1.1 Z80

Fällt vermutlich weg

2.1.2 Beispielprozessor 3 ...

s.o.
Kommentar: Ein weiterer Prozessor sollte zumindest in Ansätzen analysiert werden.

2.1.3 Intel 8080

Da der Intel 8080 in dem Single-Prozessor-Simulator verwendet wird und dort auch detaillierter implementiert wird, wird dieser auch dort in Kapitel 2.2.

2.2 Single-Prozessor-Simulator

Von Andreas

Als Prozessor für den Single-Prozessor-Simulator wurde der Intel 8080 ausgewählt. Der Intel8080 bietet ein breites Spektrum an Befehlen und wäre auch repräsentativ für einen der ersten großen, kommerziell erfolgreichen Prozessoren. Aufgrund der überschaubaren Komplexität besteht auch die Möglichkeit diesen in einem größeren Unmfang zu simulieren.

2.2.1 Register

Der Intel 8080 besitzt ein SRAM Array mit 16-bit Register. Darin enthalten sind der Programm Counter 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 acht Bit auf den internen Adressbus zu schreiben oder von dort zu lesen. **Über das Adress-Latch oder den Incrementer/Decrementer-Circuit ist es möglich 16-Bit aus den Registern weiterzuleiten. Von dort aus können die Werte dann in den Adress-Puffer geschrieben werden.** [Intel 8080 S16 (2-2)]

TODO Register zuweisung b = 000 etc. einfügen

Abbildung 1: Kodierung der Register

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		

2.2.2 Befehle

Der Intel 8080 ist in der Lage Befehle, die aus einem, zwei oder drei Bytes bestehen, auszuführen. Dabei gibt das erste Byte immer den Opcode oder 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 "Direkt“ wird der Wert in dem 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 verhält sich wie bei Direkt. 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 verwendete Wert. Bei [TODO]„Immediate“steht im zweiten und/oder dritten Byte ein Wert mit dem gearbeitet wird (Lowbyte im zweiten Byte). [Intel 8080 S.47 (4-2)]
Bei Interrupts und Branch Befehlen gibt es nur den "Direct“ und "Register indirect“ Modus [Intel 8080 S.47 (4-2)].
Der Prozessor besitzt fünf Condition Flags. Das Zero flag, das angibt ob das Ergebnis eines Befehls den Wert 0 hatte. Das Sign flag, welches angibt ob Bit 8, das Most Signifikant-Bit, des letzten Ergebnisses den Wert 1 hat. Das Paritäts flag, welches gesetzt ist, wenn das letzte Ergebnis einen Modulo 2 Wert von 0 hat, also der Wert gerade ist. Das Carry flag, das einen Übertrag bei einer Addition oder einen Abzug bei einer Subtraktion oder Vergleich anzeigt und noch das Auxiliary Carry flag, welches ebenfalls einen Übertrag oder Abzug anzeigt aber zwischen dem vierten (Bit 3) und fünften Bit (Bit 4). [Intel 8080 S.47f (4-2)]

2.2.3 OP-Code Codierung

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 codieren und andere bei denen jedes Bit verwendet wird. In 1 stehen die verschiedenen Möglichkeiten. 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 codiert. 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 8 beschrieben, können auch zwei 8-Bit-Register zu einem 16-Bit-Register zusammen geschlossen werden 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 werden.

Tabelle 1: Op-Code Typen

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

2.2.4 Data Transfer Group

Für den Prozessor sind 13 Befehle aus dieser Gruppe bekannt. Bei keinem dieser Befehle werden die Condition Flags gesetzt oder zurückgesetzt.

- 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

2.2.5 Arithmetic Group

20 Befehle Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Carry, and Auxiliary Carry flags accord ing to the standard rules. All subtraction operations are performed via two's complement arithmetic and set the carry flag to one to indicate a borrow and clear it to indicate no borrow

- Add Register
- Add Memory
- Add immediate
- Add Register with carry
- Add Memory with carry
- Add immediate with carry
- Sustract 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

2.2.6 Logical Group

19 Befehle Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Auxiliary Carry, and Carry flags according to the standard rules

- 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

2.2.7 Branch Group

8 Befehle Flags are not affected

- Jump

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

2.2.8 Stack, I/O and Machine Control Group

12 Befehle Unless otherwise specified, condition flags are not affected by any instructions in this group

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

2.2.9 Maschinenzyklen und Typen

Jeder Befehl oder Befehlszyklus benötigt mindestens einen und maximal fünf Maschinenzyklen zum kompletten Ausführen des Befehls. Dabei kann jeder Befehl aus mehreren Typen bestehen, wobei der erste immer ein Fetch-Befehlstyp ist. Die existierenden Befehlstypen sind:

- Fetch
- Memory Read
- Memory Write
- Stack Read
- Stack Write

- Input
- Output
- Interrupt
- Halt
- Halt Interrupt

Jeder Maschinenzyklus besteht nochmal aus drei bis fünf Zuständen. So kann ein Befehl insgesamt zwischen vier und achtzehn Zuständen andauern. In dem ersten Zustand jedes Maschinenzykluses wird immer der Befehlstyp während des SYNC-Taktes kodiert. Die folgende Abbildung zeigt, wie die Befehlstypen über die **Data Lines** kodiert werden. [Intel 8080 S17ff. 2-3]

Abbildung 2: Kodierung der Befehlstypen beim SYNC Takt

STATUS WORD CHART

TYPE OF MACHINE CYCLE

DATA BUS BIT	STATUS INFORMATION	① INSTRUCTION FETCH	② MEMORY READ	③ MEMORY WRITE	④ STACK READ	⑤ STACK WRITE	⑥ INPUT READ	⑦ OUTPUT WRITE	⑧ INTERRUPT ACKNOWLEDGE	⑨ HALT ACKNOWLEDGE	⑩ INTERRUPT ACKNOWLEDGE WHILE HALT
		①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩ (N) STATUS WORD
D_0	INTA	0	0	0	0	0	0	0	1	0	1
D_1	$\overline{WO}$	1	1	0	1	0	1	0	1	1	1
D_2	STACK	0	0	0	1	1	0	0	0	0	0
D_3	HLTA	0	0	0	0	0	0	0	0	1	1
D_4	OUT	0	0	0	0	0	0	1	0	0	0
D_5	M_1	1	0	0	0	0	0	0	1	0	1
D_6	INP	0	0	0	0	0	1	0	0	0	0
D_7	MEMR	1	1	0	1	0	0	0	0	1	0

3 Umsetzung

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

3.1 Abstraktion der Architektur

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

3.1.1 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 zu unterscheiden zwischen zwei Arten von Anfragen, gültige und ungültige. Gemeint ist sind damit Anfragen 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.

3.1.2 ALU

Die ALU soll die Bearbeitung logischer Operationen im Prozessor simulieren. Sinn ist dabei ein möglichst genaues Bild jedes Zustandes des Prozessors darzustellen, statt beispielsweise solche Operationen einfach manuell in der Prozessor-Klasse abzuarbeiten.

3.1.3 Register

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

3.1.4 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 bspw. Taktsignal o.ä.

3.2 Möglichkeiten der Simulation

Um einen Ablauf für die Simulation festzulegen muss erst bestimmt welche Parameter eingestellt, welche Optionen verändert und welche Operationen im Simulator durchgeführt werden können.

3.3 Ablauf der Simulation

Platzhalter

3.4 Aufbau der GUI

Das Simulieren des Prozessors ist die eine Seite dieses Projekts. Gleichermaßen wichtig für einen funktionierenden Simulator ist aber auch die Benutzeroberfläche mit der dieser bedient wird. In diesem Abschnitts werden die einzelnen Aspekte dieser analysiert.

3.4.1 Hauptmenü

Bei Programmstart wird das in Abbildung 3 gezeigte Fenster geöffnet.

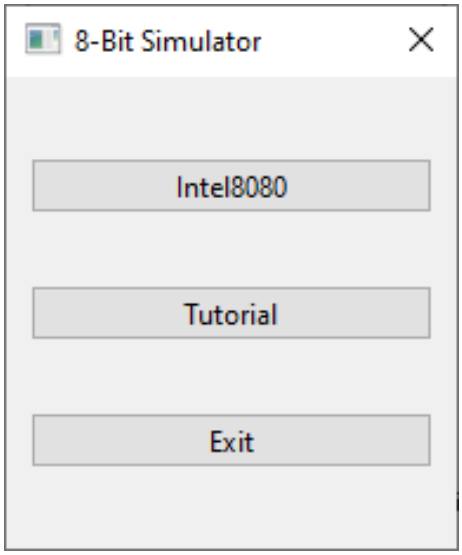


Abbildung 3: Hauptmenü

Die Aufgabe dieses Fensters ist simpel. Es gibt 3 verschiedene Knöpfe die jeweils eine Funktion erfüllen.

- Intel8080: Öffnet das Simulationsfenster des Intel8080 Prozessors
- Tutorial: Öffnet ein Fenster welches eine kurze Einweisung gibt in die Handhabung des Simulators für den Intel8080

- Exit: Schließt das Fenster und alle laufenden Hintergrundprozesse, für den Fall das ein Simulatorprozess in der selben Session verwendet wurde.

3.4.2 Intel8080 Simulationsfenster

In Abbildung 4 ist eine Version des Simulationsfensters für den Intel8080 zu sehen.

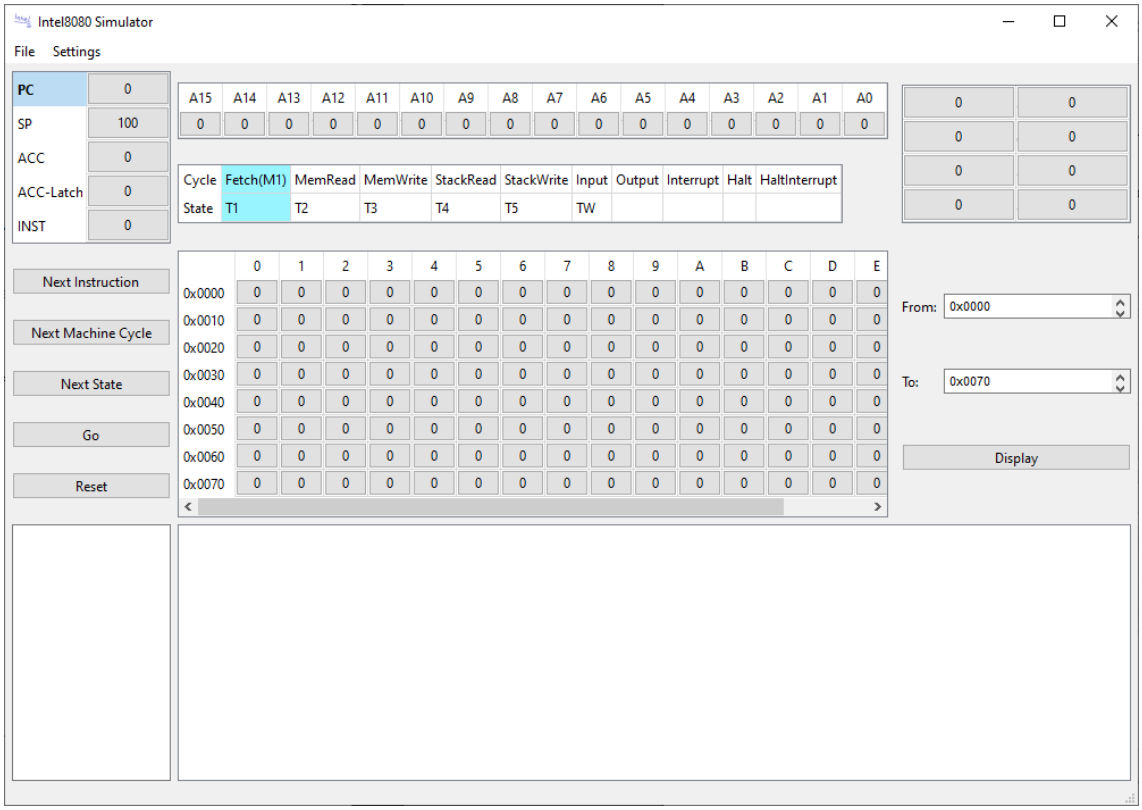


Abbildung 4: Intel8080 Sumulationsfenster

Über das Simulationsfenster für den Intel8080 können Programme die aus Bytebefehlen bestehen, welche ein Intel8080 Prozessor ausführen kann, simuliert werden. Dafür muss zuerst ein gültiges Programm geladen werden. Dies kann über den Reiter 'File' in der Menüleiste getan werden (siehe Abbildung 5).

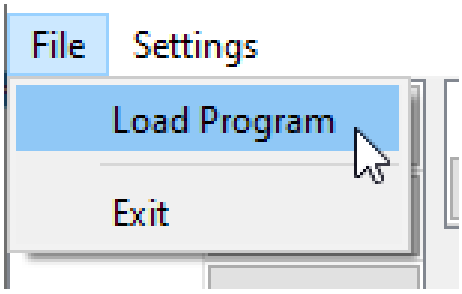


Abbildung 5: File Reiter

Dadurch öffnet sich ein Dateien-Explorer über den die gewünschte Datei ausgewählt werden kann. Dabei ist zu beachten, dass in der aktuellen Version nur Ausgabe-

dateien des verwendeten Assemblers genutzt werden können (Dateiendung '.com'). Abgesehen davon kann über 'Exit' auch ins Hauptmenü zurückgekehrt werden. Sobald ein Programm geladen wurde, wird dieses in der dafür vorgesehenen Tabelle angezeigt (siehe Abbildung 6).

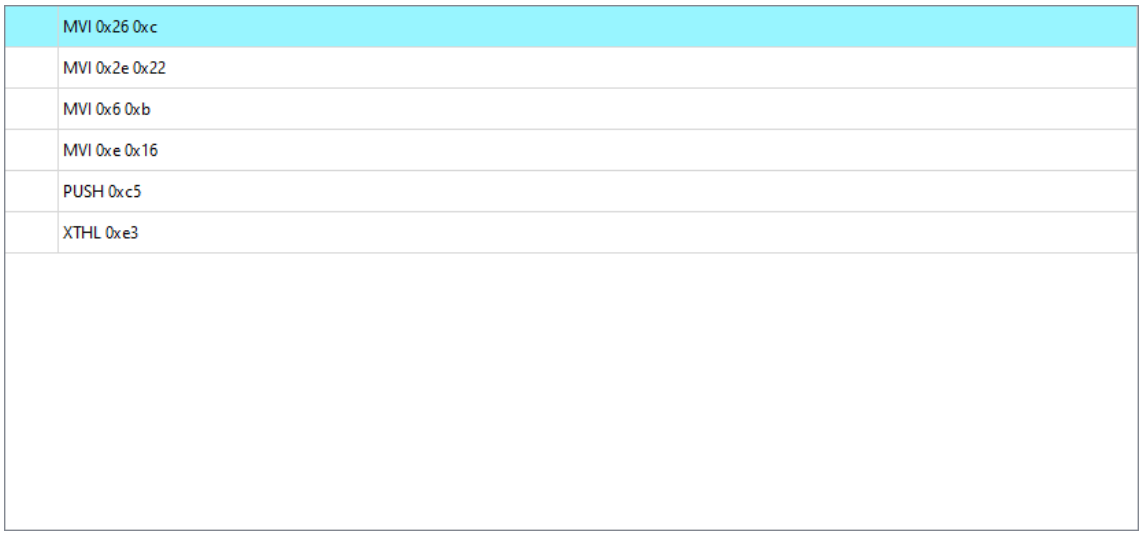


Abbildung 6: Programm Tabelle

Dabei ist für jeden Befehl eine Zeile vorgesehen. Angezeigt wird einerseits der Bytecode als auch die Bezeichnung des Befehls sowie zusätzliche Parameter falls der jeweilige Befehl welche besitzt. Außerdem wird die Zeile die als nächstes ausgeführt wird farbig markiert. Abgesehen von der Anzeige des Programms und des als nächstes ausgeführten Befehls ist es auch möglich Breakpoints zu setzen (siehe 7).

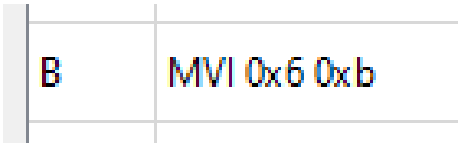


Abbildung 7: Breakpoint

Diese sind relevant wenn das Programm automatisch ausgeführt wird (sehe nächster Abschnitt). Wird ein Breakpoint erreicht, so stoppt das Programm bevor der Befehl ausgeführt wird und das Programm muss entweder manuell oder erneut automatisch ausgeführt werden. Um das Programm auszuführen werden drei Knöpfe verwendet (siehe Abbildung 8).

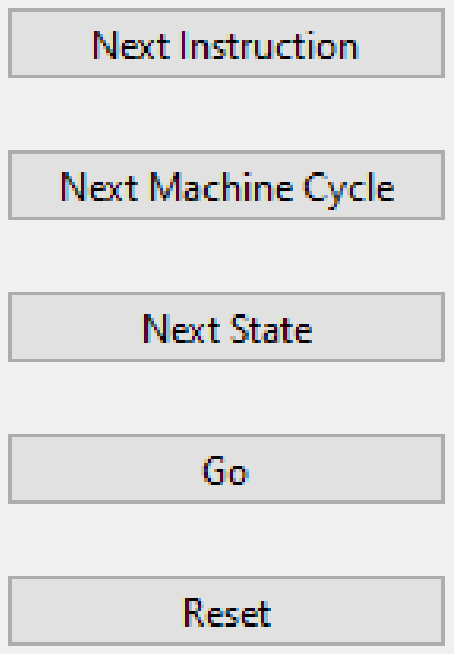


Abbildung 8: Bedien-Knöpfe

Über 'Next' wird der nächste Befehl, also der in der Programm Tabelle farbig markierte, ausgeführt und die Benutzeroberfläche wird entsprechend aktualisiert. Über Go wird automatisch das Programm ausgeführt. Bei erneutem drücken wird das Programm pausiert. Reset setzt den Simulator auf die Ausgangssituation zurück. Es ist auch möglich einzelne Register des Prozessors auszulesen. Wie in Abbildung 9 zu sehen sind diese mit deren entsprechender Bedeutung gekennzeichnet sowie deren Wert daneben.

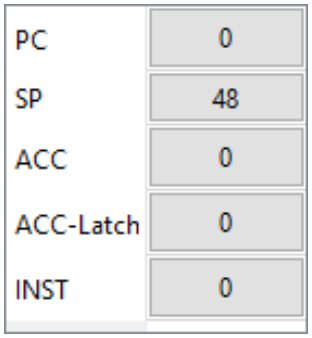


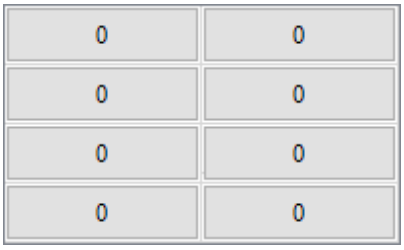
Abbildung 9: Ausgewählte besondere Register

Die gezeigten Einträge stehen für:

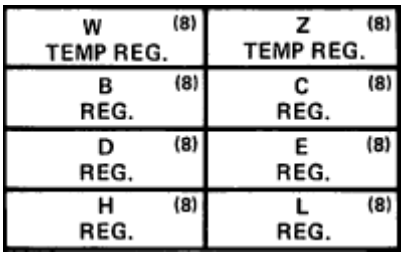
- PC: Program Counter Register
- SP: Stack Pointer Register
- ACC: Accumulator Register

- ACC-Latch: Accumulator-Latch Register
- INST: Instruction Register

Über die Knöpfe daneben in denen auch die Werte stehen lässt sich über klicken dieser ein Fenster öffnen, welches erlaubt den Wert zu ändern. Das in Abbildung 10 gezeigte Fenster ist für die restlichen Register zuständig.



(a) Simualtor generische Register



(b) Äquivalente Darstellung

Abbildung 10: Simulator Register und Darstellung

Damit gemeint sind die generischen Register ohne feste Funktion, welche jeweils eine Speicherkapazität von acht Bit haben. Das letzte Register, welches dem Nutzer angezeigt wird und welches bearbeitet werden kann ist in Abbildung 11 zu sehen.

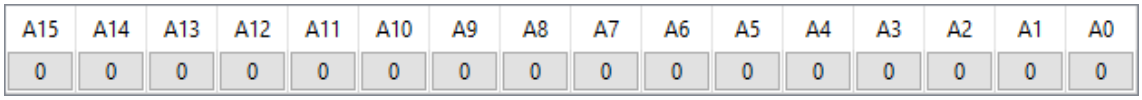


Abbildung 11: Adresspuffer

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, steuert man die Bits einzeln an. Generell ist noch zu sagen, dass als Bedingung für das Bearbeiten der Werte innerhalb der generischen oder besonderen Register gilt, dass ein Programm geladen ist sowie der Simulator gerade nicht im automatischen Betrieb für dieses ist.

3.5 Tutorial

4 Implementierung

Von Andreas
In diesem Kapitel wird gezeigt, was programmiert wurde, bis sich entschlossen wurde nur einen Prozessor zu simulieren. Obwohl dieser Simulator eigentlich mehrere Prozessoren simulieren können sollte, konnte dieser nie mehr als den Intel 8080 simulieren. Bei der Vorbereitung einen neuen Prozessor zu simulieren, ist aufgefallen, dass die Prozessoren sich nur minimal Unterscheiden werden. Mit dieser Erkenntnis wurde dann der bisherige Simulator so umgebaut, dass ein Prozessor detaillierter simuliert wird. Dessen Implementierung wurde in Kapitel 4.2 beschrieben.

4.1 Multi-Prozessor-Simulator

Mit diesem Simulator soll es möglich sein Assembler Code in ein Textfeld einzugeben und diesen dann ausführen zu können. Damit dies möglich ist muss der Assembler Code erst erstellt werden. Dies wird mit einem einfachen Assembler aus dem Internet gemacht. Dieser erstellt aus dem Source Code (.asm) eine .com-Datei, woraus die Befehle byteweise ausgelesen werden können. Manche Befehle verwenden aber auch mehr als ein Byte. Zusätzlich dazu wird noch eine Datei (.sym) erstellt, die die Lables/Sprungmarken abspeichert.

4.1.1 Programmablauf

Der eigentliche Simulator verwendet nur die .com-Dateien. Als aller erstes 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 jedem Befehl wird geschaut, ob ein Interrupt ausgelöst wurde. Falls das der Fall ist, 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 Befehl übergeben, und direkt ausgeführt wird. Während des Interrupts werden weitere Interrupts ignoriert außer in der Interrupt Service Routine werden Interrupts explizit wieder aktiviert.

4.1.2 Dekodieren eines Befehls

Um aus einem Byte ein Befehl analysieren zu können, wird dieser Wert mehrfach mit vielen Hexadezimalen Werten verglichen. Anhand der Dokumentation des Intel 8080 werden somit alle Befehle unterschieden. **[TODO]** Wie in Abbildung 12 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 12 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 Null und Acht. Entsprechend der Abbildung 1.

Abbildung 12: Lesen und decodieren eines Bytes

```
def nextInstruction(self):
    if (self.get_pc() < len(self.program)) and (self.get_memory_byte(self.get_pc()) != 0):
        instruction = self.get_memory_byte(self.get_pc())
    else:
        return

    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:
```

Dieses Vorgehen wird bei einigen Befehlen verwendet. In Kapitel 2.2.3 wird genauer gezeigt, wie die Register in den Befehlen codiert werden.

4.1.3 Programmaufbau

Der Simulator besteht hauptsächlich aus drei Klassen:

- Dem Prozessor (Intel8080.py)
- Der ALU (Intel8080_ALU.py)

- Die 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 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 (nur wenn die entsprechende Methode verwendet wird) wird der Programmzähler automatisch um eins erhöht siehe Abbildung 12. Damit 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.

Abbildung 13: Implementierung des Lesens eines Bytes

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

Der Befehl DCX reduziert das angegebene 16-Bit Register um 1. Abbildung 14 zeigt die Implementierung des DCX Befehls. Dort ist zu sehen, dass es bei den 16-Bit Registern ein 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 Parameter rp kann ermittelt werden ob es sich um den Stackpointer (rp == 3 bedeutet Stackpointer) handelt, der angepasst werden soll. Der Stackpointer kann einfach 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.

Abbildung 14: Implementierung des DCX-Befehls

```
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)
```

Die zusammengesetzten Register können alle gleich behandelt werden, da bei dem Programmkonzept daran gedacht wurde, dass dies öfter vorkommen kann. Deshalb wurden die Register so positioniert, dass auf die Register anhand ihrer Kodierung (siehe. 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 entsprechenden 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.

Abbildung 15: Register Array des Intel8080

```
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
                  ]
```

4.1.4 Befehlsvarianten

Der Intel 8080 kennt mehrere Befehle, die sowohl eine direkte als auch eine indirekte Variante besitzen. Der Unterschied ist nur, dass die Werte, mit denen gerechnet wird, von unterschiedliche Stellen geladen werden. Aus diesem Grund greifen beide Befehle im Hintergrund auf die gleichen Methoden zu. Somit ist sicher gestellt, dass falls etwas an der Implementierung geändert werden soll, das nur an einer Stelle gemacht werden muss.
Ein Beispiel dafür ist die „Subtract register/immediate from A with borrow (SBB/SBI)“. Es gibt für beide Mnemonics eine Methode in der Intel8080-Klasse aber die Methode von SBB leitet später in der ALU-Klasse nur noch auf die SBI-Methode weiter.

Abbildung 16: Zugrunde liegende Implementierung von SBB und SBI

```
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)
```

Die direkten Befehle lesen 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. Dies wird deshalb gemacht, da die ALU eigentlich keinen direkten Zugriff auf den Speicher hat. Mit dem gelesenem Wert wird dann in der ALU-Klasse die SBB-Methode aufgerufen. Diese leitet innerhalb der Klasse nur noch auf die SBI Methode weiter.

Ähnliches wird auch bei bei den Sprungbefehlen gemacht. Dort 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 17 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. Wenn nicht wird das Programm normal weitergeführt.

Abbildung 17: Implementierung der Sprungbefehle

```
def jump_general(self, low, high):
    self.set_pc(build_16bit_from_8bits(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())
```

4.1.5 Binäre Addition uns Subtraktion

Da die Additions- und Subtraktions-Befehle das Carry- und Auxiliary-Carry-Flag verändern, musste eine manuelle Addition implementiert werden. Das Carry-Flag, das anzeigt, ob ein Überlauf beim siebten Bit stattgefunden hat, hätte noch ohne eine eigene Implementierung der Addition funktioniert. Dafür müsste nur 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, dass 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. Sofern eines der beiden Flags verändert wird, 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.

Abbildung 18: Implementierung der binären Addition

```
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
```

4.1.6 GUI-Methoden

Die eine Seite des Simulators ist das Simulieren des Prozessors, mit diesem muss aber auch interagiert werden. Die in diesem Abschnitt erklärten Funktionen beziehen sich deshalb auf die in 3.4 beschriebenen Funktionen für die Benutzeroberfläche.

4.2 Single-Prozessor-Simulator

Von Andreas

In diesem Kapitel wird gezeigt, was die neue Version des Simulator kann. Dieser unterscheidet sich in sofern von dem Multi-Prozessor-Simulator, dass jeder Befehl aus mehreren Maschinen Zyklen und diese wiederum aus mehreren Zuständen bestehen. So ist es möglich, komplette Befehle aber auch einzelne Maschinen Zyklen 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 wie die Befehle ausgeführt worden sind, war weit weg von der Funktionsweise eines echten Prozessors.

4.2.1 Aufbau eines Befehls

Jeder Befehl besteht aus einem Array von Maschinen Zyklen. Zusätzlich enthält jeder Befehl auch einen Zähler, wie viele Maschinen Zyklen, der Befehl schon ausgeführt hat. Ein Befehl hat auch nur eine wichtige Methode. Die next_state()-Methode

sorgt dafür, dass der aktuelle Maschinen Zyklus seine next_state()-Methode ausführt. Wenn die Maschinen Zyklus next_state()-Methode True zurückliefert, bedeutet das, dass der letzte Zustand des Maschinen Zyklus ausgeführt wurde. Wenn dies der Fall ist, wird der Maschinen Zyklus Zähler um eins erhöht und es wird ebenfalls ein True zurückgeliefert. Dieses Signal des Maschinen Zyklus wird bis in die äußerste Schicht weitergeleitet, damit dort spezielle Ereignisse ausgeführt werden können z.B. die GUI aktualisieren. Beim nächsten Aufruf von next_state() wird somit der erste Zustand des nächsten Maschinen Zyklus ausgeführt.

Jeder Maschinen Zyklus besteht aus einem Array von Zuständen, einem Zähler, der die bisher ausgeführten Zustände zählt und eine Referenz auf den Prozessor. Diese Referenz ist notwendig, damit die Zustände, die in den Maschinen Zyklen 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 Maschinen Zyklus 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 Maschinen Zyklus verwendet werden kann, besitzt der Zustand selber keine weiteren Informationen.

4.2.2 Programmablauf

Die GUI besitzt vier Knöpfe, die das ausführen eines Befehls/Maschinen Zyklus oder Zustand auslöst. Bei dem „Next State“-Knopf wird genau ein Zustand ausgeführt. Bei dem „Next Machine Cycle“-Knopf wird der aktuelle Maschinen Zyklus bis zum Schluss ausgeführt. Der „Next Instruction“-Knopf verhält sich wie der Maschinen Zyklus-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 im Intel8080 gebunden, die den selben Namen trägt 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 zurückliefert. Diese wiederum führt solange die next_state_internal()-Methode aus, bis diese True zurückliefert. Diese liefert True, wenn der letzte Zustand eines Maschinen Zyklus ausgeführt wurde. Die next_state()-Methode ruft nur einmal die next_state_internal()-Methode auf.

4.2.3 Dekodieren eines Befehls

4.2.4 Programmaufbau

4.2.5 Befehlsvarianten

5 Fazit und Ausblick

Platzhalter

Literatur

[1] Google: https://www.google.com

[2] Grundlagen der Informatik: Herold, Lurz, Wohlrab und Hopf; 3. aktualisierte Auflage (2017), Pearson

[3] Instruction formats: https://www.geeksforgeeks.org/computer-organization-instruction-formats-zero-one-two-three-address-instruction/, zuletzt abgerufen: 25.12.2021