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 3. Mai 2022

Bearbeitungszeitraum	2 Semester
Kurs	TINF19B3
Betreuer der Ausbildungsfirma	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, 3. 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

Platzhalter

2.1 Auswahl geeigneter Varianten

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.

2.2 Intel 8080

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

Von Andreas
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. 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)]

2.2.2 Befehle

Von Andreas
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 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

2.2.4 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

2.2.5 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

2.2.6 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

2.2.7 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

2.2.8 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 1: 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
											Ⓝ 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

2.3 Beispielprozessor 2

Fällt vermutlich weg

2.4 Beispielprozessor 3 ...

s.o.

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

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 2 gezeigte Fenster geöffnet.

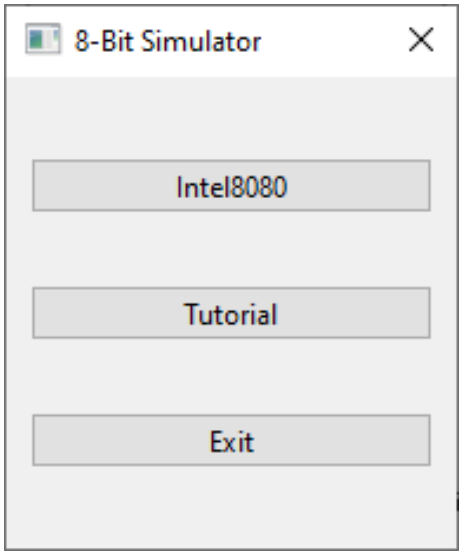

Abbildung 2: Hauptmenü

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 3 ist eine Version des Simulationsfensters für den Intel8080 zu sehen.

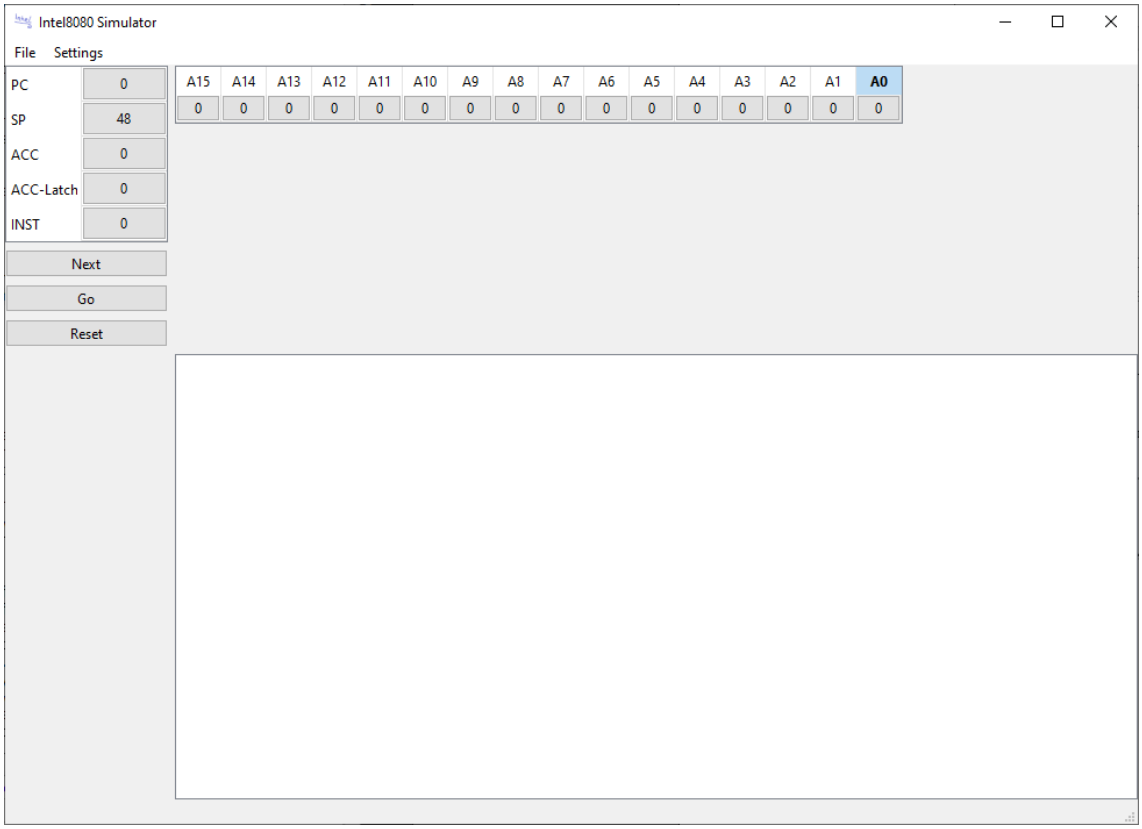

Abbildung 3: Intel8080 Sumulationsfenster

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 4).

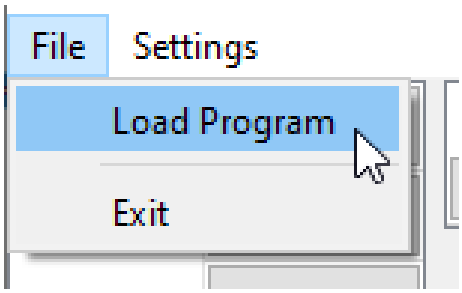

Abbildung 4: File Reiter

Dadurch öffnet sich ein Dateien-Explorer über den die gewünschte Datei ausgewählt

werden kann. Dabei ist zu beachten, dass in der aktuellen Version nur Ausgabedateien des verwendeten Assemblers genutzt werden können (Dateiendung '.com'). Abgesehen davon kann über 'Exit' auch ins Hauptmenü zurückgekehrt werden.

3.5 Tutorials

4 Implementierung

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.

Unser eigentlicher Simulator verendet nur die .com-Dateien. Aus dieser Datei wird als aller ersten nur ein Byte gelesen. Mithilfe des ersten Bytes wird analysiert um welchen Befehl es sich handelt. Je nach Befehl werden noch weitere Bytes eingelesen, die für den eigentlichen Befehl gebraucht werden. Somit werden nie zu wenig oder zu viele Bytes gelesen. Aus der Datei werden immer anhand des Programmzählers die nächsten Bytes ausgelesen. In unserem Programm wurde das so implementiert, dass direkt beim lesen eines Bytes der Programmzähler um eins erhöht wird. Damit muss aber darauf geachtet werden, dass Bytes nicht mehrfach gelesen werden sollen, da, zum einen zwei unterschiedliche Werte gelesen werden und zum anderen der Programmzähler um ein weiteres Mal erhäht wird und somit ein Byte überspringt.

Abbildung 5: 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()))
```

Nachdem erkannt wurde um welchen Befehl es sich handelt werden, werden zwei Arten unterschieden. Befehle wie Sprungbefehle, die die ALU nicht benötigen und Befehle wie Additionen, die die ALU verwenden. Der Simulator besteht hauptsächlich aus drei Klassen:

- Dem Prozessor (Intel8080.py)
- Der ALU (Intel8080_ALU.py)
- Die Register (Intel8080_Registers.py)

Die Befehle, die die ALU nicht verwenden wurden in in der Intel8080.py implementiert. Die anderen entsprechend in der ALU.

4.1 Erkennen eines Befehls aus einem Byte

Nachdem der Assembler Code in das Textfeld eingegeben wurde und [TODO] auf „Play“gedrückt wurde, wird der Assembler Code erstellt. Um damit später besser arbeiten zu können, werden diese Werte direkt in einem Array für den Prozessor gespeichert. In jeder Zelle steht dann genau ein Byte.

Um einen Befehl analysieren zu können muss zunächst ein Byte aus dem Array gelesen werden. Dieser Wert wird mehrfach mit vielen Hexadezimalen Werten verglichen. Anhand der Dokumentation des Intel 8080 werden somit alle Befehle unterschieden. [TODO] Wie in Abbildung 6 zu sehen ist, wird der Befehl „Add Immediate to A with carry (ACI)“mit dem Wert 0xCE beschrieben. Bei vielen Befehlen wie ACI wird auf genau einen Wert geprüft. Ebenfalls ist in Abbildung 6 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 der die ersten 5 Byte mit dem Wert 0x88 übereinstimmen.

Abbildung 6: Implementierung des Lesens des ersten Bytes eines Befehls

```
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:
```

Bei einigen anderen Befehlen wird diese Mechanik genauso verwendet. Es gibt aber auch Befehle, da werden die 8-Bit-Register in den Bits drei bis fünf codiert. Ebenso gibt es Befehlen, welche auch 16-Bit-Register verwenden z.B. [TODO] „Load Immediate register/stack pointer (LXI)“. Wie auf Seite 6 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. In diesen Befehlen werden diese Register mit zwei Bit kodiert. Damit gibt es noch eine Bitkombination, mit der noch ein weiteres Register verwendet werden kann. Mit der vierten noch übrig gebliebenen Kombination wird der Stackpointer als 16-Bit-Register ver-

wendet werden.

4.2 Mehrfach verwendete Methoden

Der Intel 8080 kennt mehrere Befehle, die sowohl eine direkte als auch eine indirekte Variante. Der Unterschied bei diesen Varianten sind 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 nochmal etwas an der Implementierung geändert werden soll, das nur an einer Stelle gemach 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 7: 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)
```

Zu Beginn der Analyse der Befehle, wird für SBI direkt das nächste Byte gelesen und damit die Methode SBI in der ALU-Klasse aufgerufen. Für SBB wird zunächst in der Intel8080-Klasse noch der Wert aus dem Speicher gelesen, da dies von der ALU nicht möglich ist. Mit dem gelesenem Wert wird dann in der ALU-Klasse die SBB-Methode aufgerufen. Diese leitet innnerhalb der Klasse aber 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 8 zeigt wie bei den 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 auf die Programmstelle weisen, auf die gesprungen werden soll. Nur sofern das übergebene Flag gesetzt ist wird der Sprung ausgeführt. Wenn nicht wird das Programm normal weitergeführt.

Abbildung 8: 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())
```

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