6502Simulator

Tobias Bäumlin

4. März 2024

Die vorliegende Simulation des 6502-Prozessors ist hauptsächlich für den Unterricht gedacht. Hauptzwecke sind eine Veranschaulichung der Von Neumann-Architektur und eine kleine Einführung in die Programmierung mit einer Assembler-Sprache. Zudem soll ein Einblick in die Geschichte der Informatik gegeben werden.

Inhaltsverzeichnis

1. Einleitung

Die Prinzipien der Von Neumann-Architektur gehören mit zu den fundamentalen Ideen der Informatik und sind daher unumgänglicher Bestandteil eines Informatik-Unterrichts auf gymnasialer Stufe. Damit deren Vermittlung nicht in der reinen Theorie verharrt, sondern durch die Schülerinnen und Schüler begreifbar wird, braucht es Hilfsmittel zur Veranschaulichung, die auch die eigene Tätigkeit ermöglichen.

Ich habe in meiner Unterrichtstätigkeit viele solcher Hilfsmittel getestet und einige auch im Unterricht verwendet. Mit keinem davon bin ich aber vollständig – ja nicht einmal weitgehend – zufrieden gewesen. Die wenigen wirklich anschaulichen Umsetzungen sind in den Funktionen zu eingeschränkt oder stellen zu wenig realistische Modelle einer Rechnerarchitektur dar.

Der vorliegende Simulator ist nun das Ergebnis eines Projekts, das ich lange mit mir herumgetragen habe: Es soll die Verbindung einer einigermassen realistischen Darstellung eines existierenden Rechnerarchitektur mit der notwendigen Anschaulichkeit ermöglichen. Gleichzeit soll es möglich sein, einfache Beispiele und Aufgaben zur Programmierung in einer „realen“ Assembler-Sprache umzusetzen.

Der Simulator ist in Python (3.11) geschrieben und besteht aus mehreren Teilen:

- Ein *Emulator*, der den 6502 Prozessor vollständig emuliert, das heisst den vollständigen Satz an Befehlen und Adressierungsarten umsetzt. Diese Emulation ist auf einer logischen Stufe angesiedelt und zielt nicht auf Effizienz sondern auf eine möglichst getreue Umsetzung auf Register-Ebene ab.

- Eine *GUI*, die den Zustand des Prozessors darstellt und zugleich dessen Manipulation erlaubt. Diese wurde mit der Library `PySide6` erstellt.

- Ein einfacher *Assembler*, der die Programmierung des Prozessors mit einer einfachen Assembler-Sprache erlaubt. Diese ist ist schlicht gehalten, erlaubt im Wesentlichen die Verwendung der üblichen mnemonischen Prozessor-Instruktionen, das Definieren von Labels und kennt nur wenige Direktiven, zum Beispiel solche zum Setzen des Programm-Zählers (`.org`) und zur Definition von benannten Konstanten und Datenbereichen.

Das vorliegende Dokument soll hauptsächlich den vorliegenden Simulator beschreiben und einige Ideen skizzieren, wie dieser im Unterricht eingesetzt werden könnte. Es ist nicht die Absicht, eine vollständige Beschreibung des 6502 Prozessors mit allen Details und Feinheiten zu geben, dafür sei auf die diversen online Verfügbaren Quellen verwiesen, zum Beispiel in [Jac08a] und [Lan02]. Ich selbst habe mich bei der Umsetzung meines Emulators massgeblich auf diese Quellen abgestützt.

Abbildung 1: Der MOS 6502-Prozessor
(Quelle: cpu-collection.de, `http://www.cpu-collection.de/?l0=i&i=198`)

2. Der 6502 Prozessor

Der 6502 Prozessor und seine Varianten der 6500 Familie wurde in Jahr 1975 von der Firma MOS Technology entwickelt. Er kam in legendären Computern wie dem Apple II, dem Commodore PET und C64 zum Einsatz. Bis heute werden Varianten davon produziert und verkauft.

Entsprechend hat er eine grosse Fangemeinde und man findet online eine grosse Anzahl von Webseiten, die sich damit befassen, seine Eigenschaften detailliert dokumentieren und Software zu seiner Emulation bereit stellen.

Er war in seiner ersten Version aus 3510 Transistoren[1] in CMOS-Technik aufgebaut und wurde — mangels entsprechender Computer — noch von Hand gezeichnet und danach mit foto-lithografischen Mitteln verkleinert für die Produktion. Seine Taktrate wurde zwischen 1MHz und 2MHz angegeben, je nach Version.

Eines der unglaublichsten Projekte ist eine Simulation des Prozessors auf der physikalischen Transistor ebene. Das heisst, es wurde seine komplette Schaltung durch *reverse engeneering* bestimmt und daraus eine virtuelle Kopie davon erstellt[Gre10].

2.1. Aufbau

Der 6502 Prozessor hat eine Register- und damit Datenbus-Breite von 8 Bit, also 1 Byte pro Speicherstelle. Die Breite des Adressbus beträgt 16 Bit, so dass ein maximaler Hauptspeicher von 2^{16}Byte = 64KiByte adressiert werden kann. Dieser Hauptspeicher wird in *Seiten* zu je 2^{8}Byte = 256Byte aufgeteilt, wovon die erste (mit Nummer 0, daher *Zeropage* genannt) und zweite (mit Nummer 1) spezielle Funktionen haben:

- Die Zeropage dient als eine Art *Cache*, da deren Speicherstellen schneller gelesen und geschrieben werden können, als die der übrigen Seiten (siehe 4.1).
- Die zweite Seite wird als *Stack* verwendet für den Mechanismus des Aufrufs von *Subroutinen* und *Interrupts*.

[1] Alternativ findet man auch die Zahl 4528, wenn die Pull-Up-Widerstände mitgezählt werden. Siehe [Cox11]

2.2. Arithmetisch-logische Einheit

Die arithmetisch-logische Einheit, kurz *ALU* (von *A*rithmetic and *L*ogical *U*nit) des Prozessors kann die folgenden Operationen ausführen:

- *Addition* zweier Operanden mit Übertrag (*carry*). Dieser Übertrag kann dazu gebraucht werden, um die Addition von Zahlen mit mehr als einem Byte Grösse zu programmieren.
- *Subtraktion* zweier Operanden mit Übertrag (*carry*, eigentlich eher *borrow*, siehe [Shi12]).
- Bitweise logische Operationen *Und*, *Oder* und *exklusives Oder* zweier Operanden.

  Interessanterweise fehlt aber die bitweise *Negation*.
- *Inkrement* und *Dekrement*, das heisst Erhöhen bzw. Vermindern eines Operanden um Eins.

2.3. Register

Der Prozessor besitzt ein sehr kleine Anzahl von *Registern*, die man zu seiner Programmierung verwenden kann:

- Der *Akkumulator*, auch `A`-Register genannt.

  Sein Inhalt wird für den ersten Operanden der arithmetischen und logischen Operationen verwendet.
- Zwei Index-Register, `X`- und `Y`-Register.

  Deren Inhalt wird für Wiederholungen bei der Verarbeitung von Listen gebraucht.
- Der Programmzähler, auch `PC`-Register genannt (von *p*rogram *c*ounter).

  Dieses Register kann als einziges 16bit, also 2Byte, halten, besteht technisch aus zwei separaten Registern, genannt `PCL` und `PCH` für `P`rogram `C`ounter `L`ow, beziehungsweise `H`igh.

  Es enthält die Adresse des Befehls, der im nächsten Schritt geladen und ausgeführt wird.
- Der *Stackpointer*, `SP`-Register genannt.

  Zeigt auf die nächste freie Speicherstelle des Stacks, also der zweiten Speicherseite.

  Dieser wird von `FF` nach unten gezählt, das heisst die erste Stack-Speicherstelle hat die absolute Adresse `01FF`.
- Das *Statusregister*, `S`-Register.

  Von dessen 8 Bit werden 7 als *Flag* gebraucht, um verschiedene Zustände des Prozessor zu signalisieren, siehe Abschnitt 2.4.

2.4. Status-Flags

Von den acht Bits des Statusregister dienen 7 als *Flags* (Markierungen) verschiedener Zustände des Prozessors oder Ergebnisse von Instruktionen.

So können zum Beispiel die Ergebnisse von Addition und Subtraktion sowohl vorzeichenlos als auch mit Vorzeichen behaftet interpretiert werden, siehe Tabelle 1.

Tabelle 1: Statusflags

Bit Nr.	Name	Funktion
0	`C`	Das *Carry*-Flag signalisiert einen Übertrag bei der letzten arithmetisch-logischen Operation. Um ein korrektes Resultat zu garantieren, muss dieses vor jeder Addition oder Subtraktion richtig gelöscht beziehungsweise gesetzt werden.
1	`Z`	Das *Zero*-Flag wird gesetzt, wenn das Resultat der letzten Operation Null war. Dieses Flag wird nicht nur durch die arithmetisch-logischen Operationen verändert sondern auch durch die Befehle, die Daten aus dem Speicher in ein Register holen.
2	`I`	Ist das *Interrupt Disable*-Flag gesetzt, werden normale Interrupt-Requests ignoriert.
3	`D`	Wenn das *Decimal Mode*-Flag gesetzt ist, werden die Addition und Subtraktion durch die ALU in einem speziellen dezimalen Modus ausgeführt[2].
4	`B`	Das *Break*-Flag wird durch den Break-Befehl gesetzt, um zu signalisieren, dass der es sich um einen Software-Interrupt handelt, der nicht durch die Interrupt-Leitung ausgelöst wurde.
5	—	Wird nicht verwendet und kann theoretisch beim Programmieren des Prozessors für benutzerdefinierte Zwecke verwendet werden.
6	`V`	Das *Overflow*-Flag wird für die Interpretation der Addition und Subtraktion als vorzeichenbehaftete Operationen verwendet. Das gesetzte `V`-Flag signalisiert, dass bei einer vorzeichenbehafteten Addition das Ergebnis grösser als 127, bzw. bei einer Subtraktion kleiner als -128 war[3].
7	`N`	Das `n`-Flag wird gesetzt, wenn das höchstwertige Bit nach einer Addition oder Subtraktion auf 1 steht.

[2] In der aktuellen Version dieses Simulators ist dieser Modus noch nicht implementiert und das *D*-Flag wird ignoriert.

[3] Da der Zweck dieses Flags auf die Interpretation der Addition und Subtraktion als vorzeichenbehaftete Operationen beschränkt ist, kann es im Unterricht auch ignoriert werden

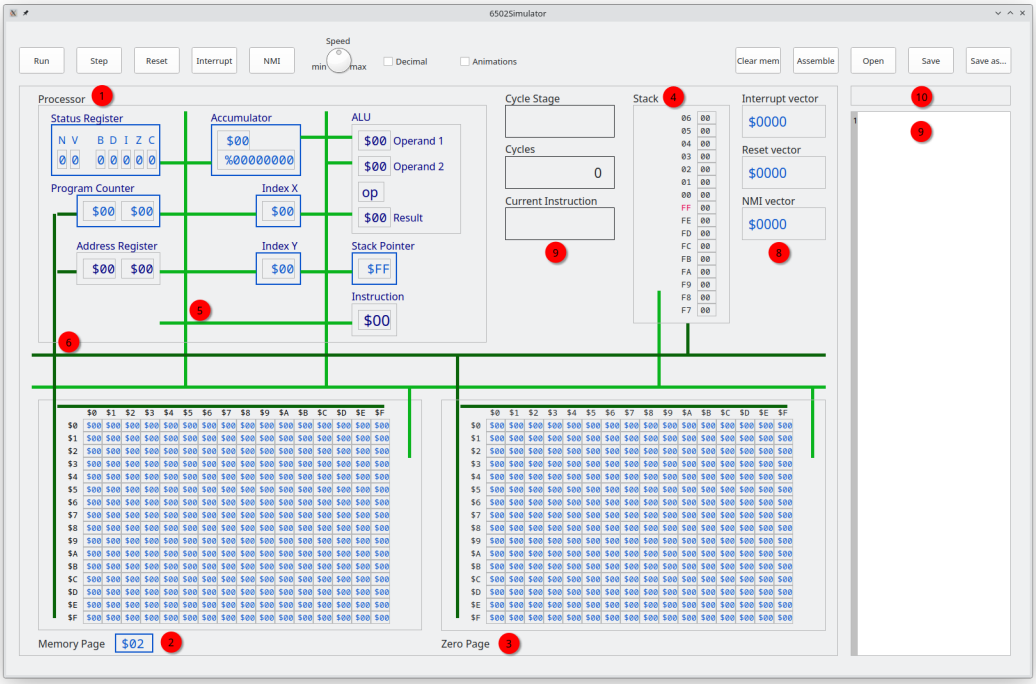

Abbildung 2: Die GUI des 6502Simulator

3. Der Simulator

Die graphische Benutzeroberfläche (GUI) des 6502Simulator stellt eine Sicht auf die relevanten Teile des 6502 Prozessors und seinen Hauptspeicher dar, siehe Abbildung 2.

Es wurde versucht, die Bedienung möglichst einfach zu halten. Es gibt kein Menü sondern nur eine kleine Reihe von Buttons oben, mit Hilfe deren die benötigten Aktionen ausgelöst werden können. Zudem sind blau dargestellte Elemente wie Register und Speicherstellen klickbar, um deren Wert zu ändern. Alle numerischen Eingaben können entweder dezimal (z. B. `134`), hexadezimal (z. B. `$A8` oder `0xFF10`) oder binär (z. B. `%100101` oder `0b11101010`) erfolgen.

3.1. Elemente

Innerhalb des Prozessors (1 in der Abbildung 2) sind die oben genannten Register `A`, `X`, `Y`, `PC`, `S` und `SP` sowie die arithmetisch-logische Einheit ALU dargestellt. Der Inhalt des Akkumulators wird zusätzlich in der binär gezeigt, um die bitweisen logischen Operationen einfacher nachvollziehen zu können.

Zusätzlich ist das Adressregister, das die aktuell adressierte Speicherstelle (zum Lesen oder Schreiben), und das *Instruktionsregister* gezeigt, das den zuletzt geladenen Opcode enthält.

Der Hauptspeicher kann aus Platzgründen natürlich nicht als Ganzes dargestellt werden:

Links unten (2 in der Abbildung 2) sind die 256 Stellen einer Seite des Hauptspeichers matrixförmig angezeigt, darunter ist die Nummer der angezeigten Seite ersichtlich und wählbar. Die dargestellte Seite wechselt dynamisch während der Ausführung eines Programms, je nach dem, auf welche Speicherstelle gerade zugegriffen wird.

Rechts daneben (3 in der Abbildung 2) ist in gleicher Form der Inhalt der Zeropage dargestellt.

Für die zweite Speicherseite, die ja den Stack enthält (4 in der Abbildung 2), wurde eine andere, spaltenförmige Darstellungsform gewählt: In der Mitte steht das Stack-Element, auf das der Stackpointer aktuell zeigt (rot hervorgehoben), nach unten nehmen die Adressen zu, nach oben ab. Die Darstellung muss man sich kreisförmig vorstellen, da oberhalb der höchsten Adresse `$01FF` wieder die tiefste Adresse `$0100` anschliesst.[4]

In hellgrüner Farbe wird der 8bit breite Datenbus dargestellt (5 in der Abbildung 2, während für den 16bit breiten Adressbus eine dunkelgrüne Farbe verwendet wird (6 in der Abbildung 2).[5]

Zwischen dem Prozessor und dem Stack (7 in der Abbildung 2 werden Informationen zum Programmablauf angezeigt: Der momentane Schritt innerhalb des Von Neumann-Zyklus „fetch“, „decode“ beziehungsweise „run“, die Anzahl abgearbeiteter Prozessorzyklen seit dem letzten Zurücksetzen sowie die aktuelle Instruktion in desassemblierter Form.

Die drei Vektoren, welche die Adressen enthalten, an denen nach einem Interrupt, einem Reset beziehungsweise einem nicht maskierbaren Interrupt (*NMI*) die Programmausführung fortgesetzt wird, sind rechts des Stacks gezeigt und können dort verändert werden (8 in der Abbildung 2).

Die Elemente des Assemblers (siehe Abschnitt 5 zuletzt nehmen den rechten Rand des Fensters ein: ein Texteingabefeld für den Assemblercode (9 in der Abbildung 2) und darüber der Name der aktuellen Assembler-Datei (10 in der Abbildung 2).

3.2. Bedienung

Die Bedienung des Simulators ist – wie gesagt – möglichst schlicht gehalten.

Es gibt im Wesentlichen drei Bereiche für die Bedienung des Simulators:

- Setzen von Werten für Register und Speicherstellen, siehe oben.
- Steuern und Darstellen des Programmablaufs.

  Run -Button: Durch Klicken darauf, wird der Programmablauf ab der aktuellen Adresse im Programmzähler gestartet. Das heisst die Instruktionen werden fortlaufend abgearbeitet.

[4] Dies in Übereinstimmung der Stack-Logik, die bei einem Unterschreiten der `$00` ebenfalls wieder nach `$FF` springt.

[5] Es wird auf eine unterschiedliche Darstellung des Bus innerhalb und ausserhalb des Prozessors verzichtet, obwohl dies in der Realität natürlich nicht dasselbe ist.

Step -Button: Die einzelne Instruktion an der Adresse im Programmzähler wird ausgeführt.

Reset -Button: Der Prozessor wird zurückgesetzt, das heisst die Register `A`, `X`, `Y`und `S` werden auf `$00` gesetzt, der Stackpointer `SP` auf `$FF` und der Programmzähler `PC` auf den Wert des Reset-Vektors.

Interrupt -Button: Ein externer Interrupt wird simuliert und die laufende Programm-Ausführung angehalten.

Speed -Regler: Die Geschwindigkeit der Simulation kann gesteuert werden.

Decimal -Checkbox: Der Inhalt von Datenregistern und Speicherstellen wird dezimal und nicht hexadezimal dargestellt. Der Programmzähler und das Adressregister hingegen werden immer hexadezimal dargestellt, genauso wie die Daten, auf dem Adressbus.

Animations -Checkbox: Datentransfers innerhalb des Prozessors und zwischen dem Hauptspeicher und dem Prozessor werden durch Animationen visualisiert.

- Arbeiten mit Assembler-Code. Im grossen Text-Eingabefeld kann zeilenweise Assemblercode geschrieben werden, wie im Abschnitt 5 beschrieben.

  Während der Programmausführung wird die aktuell ausgeführte Zeile farblich hervorgehoben.

  Damit der Code assembliert werden kann, muss er in einer Datei mit Endung `.asm` gespeichert werden. Der Name der aktuellen Datei wird im Text-Feld darüber angezeigt.

  Assemble -Button: Die aktuelle Assembler-Datei wird assembliert. Falls die Assemblierung fehlerfrei verläuft, wird der erhaltene Programm-Code in den Hauptspeicher geladen. Andernfalls werden Fehlermeldungen angezeigt, die die Zeilen mit Fehlern nennen.

  Open -Button: Eine bestehende Assembler-Datei kann gewählt und geöffnet werden.

  Save -Button: Der in Arbeit befindliche Assembler-Code wird in der aktuellen Datei gespeichert.

  Save as -Button: Der in Arbeit befindliche Assembler-Code kann in einer neuen Datei gespeichert werden.

  Clear mem -Button: Der Hauptspeicher wird gelöscht, das heisst es werden alle Speicherstellen auf `$00` gesetzt.

4. Instruktionen

Die *Instruktionen* oder *Befehle* des 6502 Prozessor bestehen aus einem Byte, das *Operation code* (kurz *Opcode*) genannt wird, gefolgt von bis zu zwei zusätzlichen Bytes, die einen *Operanden* angegeben.

Damit wir Menschen diese Instruktionen einfacher lesen und vor allem auch schreiben können, werden sowohl für die Opcodes als auch für allfällige Operanden nicht die Bytes als Zahlenwerte notiert, sondern es werden mnemonische Schreibweisen verwendet. Die Übersetzung dieser *Mnemonics* in die Maschinensprache, also in die entsprechenden Byte-Sequenzen nennt man *Assemblierung*, die Übersetzung in der umgekehrten Richtung *Desassemblierung*.

In Tabelle 2 sind eine Reihe von Instruktionen zusammen mit ihrer mnemonischen Schreibweise aufgelistet und kurz beschrieben. Details finden sich in den Abschnitten 4.2 und 4.1[6].

Tabelle 2: Beispiele von Instruktionen

Instruktion	Mnemonic	Beschreibung
`$A9 $10`	`LDA #$1`	Der Wert `$10` wird in den Akkumulator geladen
`$8E $00 $20`	`STX $0200`	Der Wert des Index-Registers `X` wird an die Speicherstelle `$200` geschrieben.
`$65 $B0`	`ADC $B0`	Der Wert in der Speicherstelle `$00B0` wird zum Akkumulator `A` addiert unter Berücksichtigung des Carry-Flags C
`$88`	`DEY`	Der Wert des Index-Registers `Y` wird um 1 verkleinert (decrement)
`$6C $34 $12`	`JMP ($1234)`	Die Programmausführung wird ab der Stelle fortgesetzt, deren Adresse an den Speicherstellen `$1234` und `$1235` steht.
`$10 $08`	`BPL` *`*10`*	Fall der `N`-Flag nicht gesetzt ist, werden die nächsten 8 Byte in der Programmausführung übersprungen. In der mnemonischen Schreibweise steht dafür `*10`, weil 2 Bytes schon durch die Anweisung selbst benötigt werden.
`$AA`	`TAX`	Kopiere den Wert des Akkumulators `A` in das Index-Register `X`.

4.1. Adressierungsarten

Viele Instruktionen brauchen einen zusätzlichen Operanden. Dieser wird in einem bis zwei zusätzlichen Adressbytes angegeben, die nach dem Opcode folgen. Der Opcode zusammen mit den Adressbytes machen zusammen die Instruktion aus.

[6] Es macht im Unterricht sicher Sinn, wenn man den Einstieg mit Hilfe dieser und ähnlicher Beispiele macht, mit dem Simulator diese zunächst Byte für Byte in den Hauptspeicher schreibt, ausführen und damit visualisieren lässt.

Der 6502 Prozessor kennt eine Reihe von verschiedenen *Adressierungsarten* (*Address mode*), die beschreiben, wie aus den Adressbytes letztlich der Operand bestimmt wird. Tabelle 3 gibt einen Überblick all dieser Adressierungsarten[7].

Tabelle 3: Adressierungsarten

Bezeichnung `Mnenomic`	Beschreibung	Grösse
Implicit `impl`	Implizite Adressierung Dabei sind die Operanden implizit im Befehl beinhaltet und es werden keine weitere Operanden benötigt.	1Byte
`TXS`	Kopiere das `X`-Register in das `S`-Register	
`CLC`	Lösche das `C`-Flag,	
`INC`	Erhöhe den Wert des Akkumulators	
Immediate `imm`	Unmittelbare Adressierung Ein nachfolgendes Byte direkt als nummerischer Wert interpretiert. Es handelt sich also nicht um eine Adresse im eigentlichen Sinn.	2Byte
`LDX #$10`	Lade den Wert `$10` in das `X`-Register	
`ADC #$80`	Addiere den Wert `$80` zum Akkumulator `A`	
`AND #$01`	Bitweises Und des Akkumulators mit dem Wert `$1`	
`CPY #$80`	Vergleiche das `Y`-Register mit dem Wert `$80`	
Absolute `abs` **Zeropage** `zp`	Absolute Adressierung Hier wird geben zwei Byte nach dem Opcode die vollständige Speicher-Adresse an, an der der Operand steht. Wird nur ein Byte angegeben, wird dieses als Adresse innerhalb der ersten Speicherseite interpretiert und man spricht man von Zeropage-Adressierung.	3Byte/2Byte
`LDX $4010`	Lade den Wert an der Adresse `$4010` in das `X`-Register	
`STY $FF`	Speichere den Wert des `Y`-Registers in der Adresse `$00FF`	

[7] Im Unterricht wird es nicht möglich sein, alle diese Adressierungsarten anzuschauen. Sie sind in daher in absteigender Wichtigkeit (für den Unterricht) in der Tabelle aufgelistet.

Die indirekten Adressierungsarten können zum Beispiel ohne grossen Schaden weggelassen werden.

Da der Assembler (siehe Abschnitt 5) uns die Berechnungen der relativen Adressierung bei Verzweigungsbefehlen abnimmt, kann auch darauf verzichtet werden, deren Details anzuschauen.

Tabelle 3: Adressierungsarten

Bezeichnung `Mnenomic`	Beschreibung	Grösse

Absolute indexed
`abs,X`, `abs,Y`
Zeropage indexed
`zp,X`, `zp,Y`
	Absolute Adressierung mit `X`-Indexierung beziehungsweise `Y`-Indexierung.
	Hierbei wird zu der Adresse, die in den beiden folgenden Bytes steht, noch der Wert des `p11.8cm`, bzw. `Y`-Registers als Index addiert.
	Wie bei `abs`gibt es davon auch die entsprechenden Zeropage-Varianten, bei denen wiederum nur ein Byte verwendet wird, um eine Adresse innerhalb der ersten Speicherseite anzugeben.
		3Byte/2Byte

`STA $2000,X`	Ist der Wert des Index-Registers `X` gleich `$A0`, so wird der Wert des Akkumulators an der Stelle `$20A0` gespeichert.	
`LDA $F0,X`	Ist der Wert des Index-Registers `X` gleich `$0044`, so wird an der Stelle `$34` in den Akkumulator `A`geladen. Beachte, dass eigentlich `$F0` + `$44` = `$134` ist. Der Übertrag wird also ignoriert!	
Indirect `(ind)`	Indirekte Adressierung Es wird mit zwei Bytes eine Speicherstelle angegeben, an der die effektive Adresse des Operanden beginnt (das heisst, das *Low Byte*steht an dieser Adresse, das *High Byte* in der darauf folgenden).	3Byte
`STA ($1234)`	Steht an der Stelle `$1234` der Wert `$10` und an der darauf folgenden Stelle `$1235` der Wert `$A8`, so wird der Wert der Stelle `$A810` in den Akkumulator geladen.	
X-Indexed Indirect `(ind,X)`	Indirekte Adressierung mit `X`-Indexierung Bei dieser Adressierungsart wird – wie bei `zp,X` – zuerst das nachfolgende Byte zum `X`-Register addiert. Danach wird der erhaltene Wert verwendet, um den Start der vollständigen Adresse (zwei Byte in *little endian*-Ordnung) des Operanden zu erhalten.	2Byte
`STA ($A0,X)`	Steht an der Stelle `$A0` der Wert `$40` und hat Indexregister `X` den Wert `$0A`, so werden die Werte an den Stellen `$4A` und `$4B` verwendet. Sind diese `$00` und `$20`, so wird der Wert des Akkumulators an die Stelle `$2000` des Speichers geschrieben.	

Tabelle 3: Adressierungsarten

Bezeichnung Mnenomic	Beschreibung	Grösse
Indirect Y-Indexed `(ind),Y`	`Y`-Indexierte indirekte Adressierung Bei dieser indirekten Adressierungsart, wird zuerst das nachfolgende Byte verwendet, um eine vollständige Adresse (2 Byte in *little endian* Ordnung) innerhalb der Zeropage zu bestimmen. Anschliessend wird zu dieser Adresse noch der Wert des Indexregister `Y` addiert, um die definitive Adresse des Operanden zu erhalten.	2Byte
`LDA ($70),Y`	Steht an der Stelle `$70` der Wert `$B1` und danach an der Stelle `$71` der Wert `$35` und hat Indexregister `X` den Wert `$20`, so ist der adressierte Operand `$35B1`+`$20`=`$35D1`.	
Akkumulator `A`	Adressierung des Akkumulators Einige wenige Instruktionen adressieren den Akkumulator separat. Diese Adressierungsart entspricht im wesentlichen der impliziten.	1Byte
`ASL A`	Schiebt die Bits des Akkumulators um eine Stelle nach links	

4.2. Befehlssatz

Der Befehls- oder Instruktionssatz des 6502 Prozessor ist relativ klein. Es gibt insgesamt 56 verschiedene Instruktionen, die zum Teil mehrere Adressierungsarten für ihre Parameter kennen. Daraus ergibt sich ein Total von 151 verwendeten *Opcodes* von den 256 möglichen.

Die restlichen Opcodes sind sogenannt *illegal* und nicht offiziell dokumentiert. Einige davon kann man tatsächlich für eher exotische Zwecke nutzbar machen (siehe??). Der vorliegende Emulator behandelt sie aber alle als *No operation* `NOP`, also als Befehle ohne Effekt.[8].

Einen Überblick über den vollständigen Befehlssatz zusammen mit den jeweiligen Adressierungsarten findet sich in den Tabellen des Anhangs A.

4.2.1. Datentransfer

Zu den wichtigsten Aufgaben eines Programms gehört das Kopieren von Daten von einer Speicherstelle zu einer anderen. Der 6502 kennt dafür eine Reihe von Befehlen.

- `LDA`, `LDX`, `LDY` dienen zum Laden eines Bytes in die Register `A`, `X`, `Y` respektive.

[8]Dies aus didaktischen Gründen, um den Einstieg in die Assemblerprogrammierung nicht unnötig zu verkomplizieren.

Diese Operationen beeinflussen zudem alle die Flags `Z` und `N`, je nach dem, ob die transferierten Werte Null bzw. negativ [9] sind.

- `STA`, `STX`, `STY` dienen zum Speichern des Bytes in einem der Register `A`, `X`, respektive `Y` an einer Speicherstelle des RAM.
- `TAX`, `TAY`, `TXA`, `TYA` `TSX`, `TSX`, kopieren das Byte des erstgenannten Registers in das zweitgenannte Register.

  Diese Befehle setzen je nach Wert wieder die Flags `Z` und `N`.

Eine Übersicht über diese Instruktionen mit ihren Adressierungsarten und zugehörigen Opcodes, findet man in Tabelle 4.

4.2.2. Arithmetisch-logische Operationen

Der 6502 kennt eine Reihe von Instruktionen, die arithmetische Operationen (Addition und Subtraktion[10], schrittweise Vergrösserung und Verkleinerung[11]), und bitweise logische Operationen durchführen. Details entnimmt man der Tabelle 5.

- `ADC` (`AD`dition with `C`arry) führt eine Addition mit Übertrag durch und `SBC` (`SuB`traction with `C`arry) eine Subtraktion mit Übertrag.

  Damit diese die erwarteten Ergebnisse liefern, muss vor der Addition das Carry-Flag `C` gelöscht und vor der Subtraktion gesetzt werden.

  Der Übertrag dient dazu, die arithmetischen Operationen auch mit mehr als einem Byte durchführen zu können.
- Für das schrittweise Erhöhen und Vermindern eines Registers um 1[12] gibt es die spezialisierten Befehle `INC` (für `INC`rement), `INX` (`IN`crement `X`-register) und `INY` (`IN`crement `Y`-register) respektive `DEC` (`DEC`rement), `DEX` (`DE`crement `X`-register) und `DEY` (`DE`crement `Y`-register)[13]

[9] Auf vorzeichenbehaftete Arithmetik wird in dieser Dokumentation nicht näher eingegangen.
Es genügt anzugeben, dass dabei die Zahlen in 2er-Komplement Darstellung interpretiert werden und dass somit Bytes, deren erstes, also höchstwertiges Bit 1 ist, als negativ angesehen werden.
Beim Unterricht z. B. in P/AM-Klassen kann dies natürlich zum Anlass genommen werden, diese 2er-Komplement Darstellung zu repetieren.

[10] Multiplikation und Division sind nicht implementiert.
Diese sind aufwändig in der Hardware-Implementierung und blieben lange Zeit spezialisierten externen Coprozessoren überlassen (Referenz?).
Entgegen der intuitiven Annahme gehören die arithmetischen Operationen auch nicht zu den am häufigst verwendeten in der Programmierung.

[11] Dieses Inkrementieren und Dekrementieren eines Werts sind in der Programmierung so häufig verwendete Operationen, dass sich die Definition dedizierter Befehle dafür lohnt.
Ausserdem entfällt dabei die Angabe eines Operanden, der immer 1 ist.

[12] Im Gegensatz zu der allgemeinen Addition und Subtraktion sind dies in der Programmierung sehr häufig verwendete Operationen und werden insbesondere bei Wiederholungen und bei der Bearbeitung von Listen überall gebraucht.

[13] Diese Inkrement- und Dekrement-Operationen sind zyklisch in dem Sinne, dass ein Inkrement des grössten Wert `$FF` wieder `$00` ergibt und umgekehrt ein Dekrement des kleinsten Werts `$00` das Resultat `$FF` liefert.

- Die bitweisen logische Operationen Und, Oder und exklusives Oder sind mit den Instruktionen `AND`, `ORA` (von `OR` with `A`ccumulator) beziehungsweise `EOR` (von `E`xclusive `OR`) möglich[14]. Diese operieren alle mit dem Akkumulator `A` und einem weiteren Wert, der durch den Operanden gegeben ist.

- Für bitweises Schieben nach links gibt es den Befehl `ASL` (`A`rithmetic `S`hift `L`eft) und nach rechts den Befehl `LSR` (`L`ogical `S`hift `R`ight); sollen die Bits aber rotiert werden, verwendet man die Befehle `ROL` (`RO`tate `L`eft) beziehungsweise `ROR` (`RO`tate `R`ight).

4.2.3. Sprungbefehle

Die Aufgabe dieser Kategorie von Instruktionen ist die Manipulation des Programmzählers, und den Programmablauf zu kontrollieren. Auf diese Weise können bedingte Anweisungen, Programmschleifen und Unterprogramme umgesetzt werden.

- Der `JMP`-Befehl (von `JuMP`) lädt einen Wert der Länge 2Byte in den Programmzähler und bewirkt so eine einfache Verzweigung des Programmablaufs.

- Der `JSR`-Befehl (von `J`ump to `S`ub`R`outine) tut dasselbe, der Programmzähler wird aber zusätzlich auf den Stack geschoben[15], damit der Rücksprung an die Stelle nach dieser Anweisung möglich ist.

- Mit dem `RTS`-Befehl (von `R`e`T`urn from `S`ubroutine) kann aus einem Unterprogramm die Programmausführung nach der Stelle des Aufrufs fortgesetzt werden.

- Die bedingten Verzweigungsbefehle `BEQ` (für `B`ranch on `EQ`ual), `BNE` (für `B`ranch on `N`ot `E`qual), `BMI` (für `B`ranch on `MI`nus), `BPL` (für `B`ranch on `PL`us)[16], `BCS` (für `B`ranch on `C`arry `S`et), `BCC` (für `B`ranch on `C`arry `C`lear) und zuletzt `BVS` beziehungsweise `BVC` lassen die Programmausführung an eine neue Stelle im Hauptspeicher verzweigen, je nach dem, ob das Flag `Z`, `N C` respektive `V` gesetzt (1) oder gelöscht (0) ist.

  Die Distanz zum Ziel der Verzweigung muss dabei zwischen -128 und +127 liegen[17].

4.2.4. Flag-Manipulation

Der 6502 Prozessor verfügt zudem über eine Reihe von Befehlen, deren Hauptzweck es ist die Status-Flags zu setzen oder löschen.

---

[14] Wie erwähnt fehlt die bitweise Negation oder Inversion.
Es ist eine sinnvolle Übung, zu überprüfen, dass diese durch ein bitweisen exklusives Oder mit dem Operanden `$ff` ersetzt werden kann.

[15] Effektiv werde die 2 Byte des Programmzähler in der Reihenfolge *High Byte-Low Byte*auf den Stack geschoben

[16] Eigentlich müsste es „branch on not minus“ heissen!

[17] Diese Verzweigungsbefehle verwenden als einzige eine relative Adressierungsart. Die Beschränkung kommt daher, dass dabei die Sprungdistanz als vorzeichenbehafteter Integer interpretiert wird.
Da der Assembler aber deren Berechnung übernimmt, können die Details hier getrost weggelassen werden.

- Die Befehle `CLC` (von `CL`ear `C`arry flag), `SEC` (von `SE`t `C`arry flag), `CLD` (von `CL`ear `D`ecimal flag), `SED` (von `SE`t `D`ecimal flag) und `CLV` (von `CL`ear o`V`erflow) löschen beziehungsweise setzen das entsprechende Flag direkt.

- Mit Hilfe der drei Vergleichsbefehle `CMP` (für `CoMP`are), `CPX` (für `C`om`P`are `X`-register) und `CPY` (für `C`om`P`are `Y`-register) kann der Inhalt eines der drei Register mit einem anderen Wert verglichen werden.

  Sind die Werte gleich, wird sowohl das Null-Flag `Z` als auch das Carry-Flag `C` gesetzt, ist der Wert im Register grösser, wird das `C`-Flag gesetzt und das `Z`-Flag gelöscht. Ist der Wert im Register kleiner, wird sowohl das `C`-Flag als auch `Z`-Flag gelöscht[18].

4.2.5. Stack-Operationen

Der 6502 Prozessor verfügt über einen Stapel-Bereich (*stack*), der in der zweiten Speicherseite angelegt ist, also die Adressen `$01FF` hinunter bis `$0100` belegt.

Dabei enthält das Stack-Register `SP` immer das *Low Byte* der Adresse des nächsten freien Speicherplatz[19] innerhalb dieses Bereichs und wird jedes Mal dekrementiert, wenn ein Element auf dem Stack abgelegt wird (*push*) verkleinert und umgekehrt jedes Mal inkrementiert, wenn ein Element vom Stack geholt wird (*pop*)[20] .

- Die Instruktionen `PHA` und `PHP` schieben den Akkumulator `A` beziehungsweise den Programmzähler `PC` auf den Stack[21].

- Umgekehrt holen die Instruktionen `PLA` und `PLP` den Wert des Akkumulators beziehungsweise Programmzählers wieder vom Stack.

4.2.6. Interrupts

Unterbrechungen des Programmablaufs werden durch so genannte Interrupts ermöglicht, wobei zwischen Hardware- und Software-Interrupts unterschieden wird[22].

Der Befehlssatz des 6502 Prozessor kennt zwei Instruktionen für den Umgang damit:

[18] Der gleiche Effekt auf das Statusregister kann auch durch die Subtraktion erreicht werden.
Intern sind diese Vergleichs-Operationen in der ALU effektiv via Subtraktion implementiert. Sie haben aber den Vorteil, dass man durch sie den Wert im Register nicht verliert.

[19] Man sagt, das Stack-Register *zeige* auf diese Adresse, weshalb dieses etwa auch *stack pointer* genannt wird

[20] Die Datenstruktur des *stack*, die nach dem *LIFO*-Prinzip (*l*ast *i*n, *f*irst *o*ut funktioniert und die beiden elementaren Operationen *push* und *pop* kann an dieser Stelle thematisiert oder vertieft werden, sofern es die zur Verfügung stehende Zeit erlaubt

[21] Die beiden Bytes des Programmzählers werden in ??-Manier, also zuerst das *Low Byte*und danach das *High Byte*, auf den Stapel geschoben.

[22] Das ganze Thema der Interrupt-Behandlung geht sicher über das hinaus, was sinnvollerweise im obligatorischen Fach Informatik behandelt werden kann, und wird hier nicht ausführlich beschrieben. Für eine umfassende Darstellung sei zum Beispiel auf [Wil24] verwiesen

- `BRK` (für `BR`ea`K` löst einen Software-Interrupt aus. Dabei werden der (um 2 vergrösserte) Programmzähler und anschliessend das Statusregister `S` auf den Stapel geschoben und anschliessend der NMI-Vektor in den Programmzähler geladen.
- `RTI` (für `r`eturn from `i`nterrupt) ermöglicht den Rücksprung nach der Beendigung eines Programmteils für die Interrupt-Behandlung. Hier werden in umgekehrter Reihenfolge zuerst das Statusregister `S` und danach der Programmzähler vom Stapel geholt, was bewirkt, dass die Programmausführung am Ort fortgesetzt wird, wo der Interrupt stattgefunden hatte.

4.2.7. Vermischte Instruktionen

Zuletzt seien noch die Befehle erwähnt, die nicht natürlich in eine der obigen Kategorien fallen:

- `NOP` (von `N`o `OP`eration) tut genau das, was der Name sagt, nämlich nichts.
- Der Befehl `BIT` (für `BI`t `T`est) testet einige Bits seines Operanden[23].

[23] Die genauen Spezifikationen dieser einigermassen exotischen Instruktion werden hier übergangen und können zum Beispiel bei [Jac08b] nachgeschlagen werden.

5. Der Assembler

Der Simulator verfügt über einen einfachen Assembler, der das Erstellen von Programmen vereinfacht. Bei dessen Implementierung habe ich mich bei dem Quellcode von James Salvino als Vorlage bedient (siehe [Sal22]), diesen aber weitgehend umstrukturiert und neu implementiert. Dies zuerst, um ihn überhaupt zu verstehen, und danach um einige Bugs zu eliminieren, verständlichere Fehlermeldungen zu haben und ihn an meine Bedürfnisse für den vorliegenden Simulator anzupassen.

5.1. Instruktionen

Der Hauptzweck des Assemblers besteht zunächst darin, Instruktionen mit den üblichen, oben beschriebenen mnemonischen Schreibweisen in der Reihenfolge ihrer Ausführung hinschreiben zu können.

Listing 1: Assembler, Instruktionen

```
1 LDA     #100
2 ; Kommentare folgen nach einem Semikolon
3             ADC $FF00
4
5    sta $ff     ; Klein- und Grossschreibung ist irrelevant
```

Durch dieses Programm werden die drei Anweisungen `LDA #100` (2Byte), `ADC $FF00` (3Byte) und `STA $FF` (2Byte) nach einander in die Stellen 0 bis 6 des Hauptspeichers geschrieben.

Alles, was nach einem Semikolon folgt, wird als *Kommentar* betrachtet durch den Assembler ignoriert.

Gross- und Kleinschreibung ist dabei unwichtig, es empfiehlt sich jedoch aus Gründen der Leserlichkeit, sich auf eine einheitliche Schreibweise festzulegen. Ebenso werden Leerzeichen am Anfang und Ende jeder Zeile, sowie gänzlich leere Zeilen ignoriert. Auch hier erleichtert Einheitlichkeit das Lesen und Verstehen enorm.

Daneben kennt der Assembler eine handvoll *Direktiven*, die der Programmiererin gerade bei Sprungbefehlen und beim Umgang mit Daten das Leben deutlich einfacher machen können.

5.2. Die `.org`-Direktive

Beginnt eine Zeile mit der Direktive `.org` (für *org*anize) gefolgt von einer gültigen Memory-Adresse, so fügt der Assembler die folgenden Zeilen ab dieser Adresse in den Speicher ein:

Listing 2: Assembler, `.org`-Direktive

```
1 .org $200
2 LDA #1
3 ADC #1
```

Die Anweisungen, um 1 und 1 zu addieren, stehen an den Adressen 512, 513, 514 und 515 im Hauptspeicher.

Ist gewünscht, dass die Programmausführung damit beginnt, muss zusätzlich der Reset-Vektor auf diesen Wert `$200` = 512 gesetzt werden.

5.3. Die `.db`-Direktive

Die Direktive `.db` (für *d*efine *b*yte) ermöglicht es, ab der aktuellen Speicherstelle eine Reihe von Werten als Daten zu schreiben:

Listing 3: Die `.db`-Direktive

```
1 .org $A000
2 .db $0A,100,'ABC',%1101
```

Dadurch werden die sechs Werte 16 (der Wert der Hexadezimalzahl `$A`), 100, 65 (der ASCII-Code des Buchstaben 'A'), 66, 67 und 13 (der Wert der Binärzahl `%1101`) ab der Stelle `$A000` = 40960 in den Speicher geschrieben.

5.4. Die `.ds`-Direktive

Mit Hilfe der Direktive `.ds` kann eine Anzahl Byte als Speicherplatz reserviert werden.

Listing 4: Die `.ds`-Direktive

```
1         LDA $00
2         JMP next
3 .ds 16
4 next    STA $01
```

Dadurch werden in den ersten fünf Bytes des Hauptspeichers die beiden Instruktionen 16 Byte frei gelassen. Die nächste Instruktion beginnt daher an der Stelle `$15` = 21 des Hauptspeichers.

5.5. Die `.equ`-Direktive

Die Direktive `.equ` Namen für Speicheradressen zu definieren.

Listing 5: Die `.equ`-Direktive

```
1  xx .equ $AA00
2  yy .equ $AAB0
3  LDA xx
4  CLC
5  ADC xx+1
6  SEC
7  SBC yy
8  .org $AA00
9  .db 42,50
10 .org $AAB0
11 .db 92
```

Es werden zuerst die Werte, die nacheinander an den Speicherstellen `$AA00` und `$AA01` stehen addiert und anschliessend der Wert an der Speicherstelle `$AAB0` subtrahiert. Beachte, dass vor den arithmetischen Operationen das `C`-Flag richtig gesetzt wird.

5.6. Labels

Die nächste Erleichterung, die der Assembler bietet, ist die Möglichkeit, die aktuelle Speicheradresse mit einem Namen (*Label*) zu versehen, und diese dann damit anzusprechen. Dies ist insbesondere für den relativen Adressierungsmodus hilfreich, da man so nicht mehr bei jeder Änderung die Sprungweite selbst neu berechnen muss.

Ein Label muss mit einem Buchstaben beginnen, gefolgt von bis zu 7 weiteren Buchstaben oder Ziffern[24].

Listing 6: Labels 1

```
1          LDX #10
2  loop    DEX
3          BNE loop
```

Das Programmfragment führt eine Schleife 10 Mal aus, indem in die Speicherstelle der 2. Zeile mit dem Namen `loop` versehen und dann in der 3. Zeile dorthin verzweigt wird, bis das Indexregister `X` den Wert 0 enthält.

Zudem ist einfache Arithmetik mit einem Label möglich, das heisst, man kann zu einem solchen entweder eine feste Zahl addieren oder davon subtrahieren.

[24] Ausgeschlossen ist zudem das Label, das nur aus dem Buchstaben `A` besteht, wegen der Verwechslung mit dem Adressierungsart `A`

Listing 7: Assembler, Beispiel 3

```
1 paar .db 10,15
2
3 LDA paar
4 LDX paar+1
5 STA paar+1
6 STX paar
```

Das Programmfragment vertauscht den Inhalt von zwei aufeinander folgenden Speicherstellen.

6. Beispiele

Im folgenden werden einige kurze Beispielprogramme gezeigt, die wichtige Konzepte und *Best Practices* der Assembler-Programmierung zeigen sollen.

6.1. Vorlage

Es lohnt sich, für Übungen und Aufgaben eine Vorlage zu haben, die ein einheitliches Erscheinungsbild der Programme erleichtern und eine einfache Handhabung in der GUI ermöglichen:

Listing 8: Vorlage

```
; Constants and data here

.org $0200
; Entry point here

; Set reset vector to entry point
.org $fffc
.db $00, $02
```

Der Einstiegspunkt für das ausführbare Programme wird hier als `$0200` gewählt, also am Beginn der Speicherseite 2. Dieser wird zuletzt an die Stellen `$FFFC` und `$FFFD` des Speichers geschrieben, wo der Reset-Vektor des Prozessors steht[25]. Auf diese Weise, startet die Programmausführung nach einem Reset immer an der gewünschten Stelle.

6.2. Maximum von zwei, drei oder mehr Werten

Den grösseren von zwei Werten kann man einfach durch Vergleichen der beiden bestimmen:

Listing 9: Maximum von 2 Werten

```
.org $20
x .db 4
y .db 7
max .equ $10

.org $0200
        LDA x
        STA max
        CMP y
        BPL end
        LDA y
        STA max
```

[25] In bekannter *little endian* Manier.

```
13 end     LDA max
14         BRK
15
16 .org $FFFC
17 .db $00, $02
```

Will man den grössten von drei Werten bestimmen, bestimmt man zuerst das Maximum der ersten beiden und vergleicht dieses anschliessend mit dem dritten Wert:

Listing 10: Maximum von 3 Werten

```
1  .org $20
2  lst .db 4, 5, 3
3  max .equ $10
4
5  .org $0200
6          LDA lst
7          STA max
8          LDA  lst+1
9          CMP max
10         BMI next
11         STA max
12 next    LDA lst+2
13         CMP max
14         BMI end
15         STA max
16 end     LDA max
17         BRK
18
19 .org $FFFC
20 .db $00, $02
```

Dies lässt sich nun auf n viele Werte verallgemeinern: Das erste Maximum ist der erste Wert in der Liste, danach vergleicht man alle folgenden Werte mit dem bisherigen Maximum und passt dieses an, wenn man einen Wert findet der noch grösser ist[26]:

Listing 11: Maximum einer Liste von Werten

```
1  .org $20
2  lst .db 4, 5, 3, 5, 2, 6,  1, 4
3  len .db 8
4  max .equ $10
5
6  .org $0200
7  LDX #0
```

[26]Damit hat man den bekannten *Bubble Sort*-Algorithmus implementiert.

```
8  LDA lst
9  STA max
10 loop INX
11           LDA lst,X
12           CMP max
13           BMI next
14           STA max
15 next CPX len
16           BMI loop
17 end  LDA max
18 BRK
19
20 .org $FFFC
21 .db $00, $02
```

Man beachte das Verwenden des Index-Registers `X` zusammen mit der indexierten Adressierungsart `(ind,X)`, um die Elemente der Liste der Reihe nach durchzugehen.

6.3. Unterprogramme

Listing 12: Unterprogramm zur Addition zweier 8Bit-Werte

```
1  x .equ $10
2  y .equ $11
3  z .equ $12
4
5  .org $0200
6  ; Store first summand in $10
7  LDA #$40
8  STA x
9  ; Store second summand in $11
10 LDA #$50
11 STA y
12 JSR add
13 ; Load result in accumulator
14 LDA z
15
16 .org $02A0
17 add      CLC
18          LDA x
19          ADC y
20          STA z
21          RTS
22
23 .org $fffc
```

```
.db $00, $02
```

Listing 13: Unterprogramm zur Addition zweier 16Bit-Werte

```
; Lobyte of first summand at $10
x .equ $10
; Lobyte of second argument in $12
y .equ $12
; Lobyte of result in $14
z .equ $14

.org $0200
; Store summands
LDA #$A0
STA x
LDA #$01
STA x+1
LDA #$81
STA y
LDA #$10
STA y+1
; Call subroutine
JSR add16bit
; Load lobyte of result
LDA z

.org $02A0
; Adds two 16bit numbers stored in
; x and y (little endian!)
; Result in z and z+1
add16bit        CLC
        LDA x
        ADC y
        STA z
        LDA x+1
        ADC y+1
        STA z+1
        RTS

.org $fffc
.db $00, $02
```

6.4. Multiplikation

Multiplikation kann einerseits durch fortlaufende Addition implementiert werden, was zwar katastrophal hinsichtlich der Laufzeit ist, aber eine gute einfache Übung darstellt.

Listing 14: Multiplikation mittels fortlaufender Addition

```
x .equ $10
y .equ $11

.org $0240
mult    LDX X
        INX
        LDA #0
loop    DEX
        BEQ end
        ADC y
        JMP loop
end     RTS

.org $0200
; Enter multiplicand in $10 and $11
; before running the program
; Result in A
        JSR mult
        BRK

.org $fffc
.db $00, $02
```

In der Praxis wird die Multiplikation als binäre Operation implementiert, nach dem Prinzip "Verdoppeln-Halbieren". Schieben um ein Bit nach links verdoppelt einen Wert, Schieben nach rechts halbiert ihn.

Listing 15: Multiplikation als binäre Operation

```
x .equ $10
y .equ $11
res .equ $12

.org $0240
mult    LDA #0
        STA res
loop    LDA x
        AND #1
        BEQ next
        LDA y
```

```
12          CLC
13          ADC res
14          STA res
15  next    LSR x
16          BEQ end
17          ASL y
18          JMP loop
19  end     LDA res
20          RTS
21
22  .org $0200
23  ; Enter multiplicand in $10 and $11
24  ; before running the program
25  ; Result in A
26          JSR mult
27          BRK
28
29  .org $fffc
30  .db $00, $02
```

6.5. Sortierung von zwei, drei oder mehr Werten

Listing 16: Sortieren von zwei Werten

```
1  ; Labels must be defined before
2  ; usage to use them with zeropage
3  ; mode
4  .org 50
5  xx .db 50, 40
6
7  .org $00
8  start  LDA xx
9             CMP xx+1
10            BMI end ; already sorted
11            LDX xx+1
12            STA xx+1
13            STX xx
14 end   BRK
```

Listing 17: Sortieren von drei Werten

```
1  .org 50
2  lst .db 50, 20, 30
3
```

```
.org $00
start   LDA lst
            CMP lst+1
            BMI next        ; 1st < 2nd
            LDX lst+1
            STA lst+1
            STX lst
next    LDA lst+1
            CMP lst+2
            BMI next2        ; 2nd < 3rd
            LDX lst+2
            STA lst+2
            STX lst+1
next2 LDA lst
            CMP lst+1
            BMI end          ; 1st < 2nd
            LDX lst+1
            STA lst+1
            STX lst
end     BRK
```

Listing 18: Sortieren von zwei Werten

```
.org $10
lst .db 3,1,5,2,6,4
.org $20
len .db 6


.org $200
; we loop to the second last element
start  DEC len
            LDA len
; if len is 0, we're done
            BEQ end
; index starts at 0
            LDX #0
loop   LDA lst,X
; compare consecutive elements
            CMP lst+1,X
            BMI cont
; switch them if necessary
            LDY lst+1,X
            STA lst+1,X
```

```
22              STY lst,X
23  cont    INX
24  ; compare index with length
25              CPX len
26  ; loop if index is smaller
27              BMI loop
28  ; start over from beginning
29              JMP start
30  end     BRK
31
32  .org $fffc
33  .db $00,$02
```

Verallgemeinert man das Vorgehen auf eine Liste von Werten erhält man auf natürliche Art eine einfache Implementierung des Bubble-Sort.

6.6. Summe einer Liste von Werten

Listing 19: Summe einer Liste von Werten

```
1   .org $20
2   lst .db 4, 1, 10, 4, 1, 4, 5
3   endlst
4   sum .equ $10
5
6   .org $0200
7   ; Compute length of list
8           LDA #endlst
9           SEC
10          SBC #lst
11  ; Length in X register
12          TAX
13  ; Sum is 0 initially
14          LDA #0
15          STA sum
16          LDX len
17  loop    LDA lst-1,X
18          ADC sum
19          STA sum
20          DEX
21          BNE loop
22          BRK
23
24  .org $FFFC
25  .db $00, $02
```

A. Befehlsreferenz

Tabelle 4: Instruktionen für den Datentransfer

Mnemonic	Adressierungsarten und Opcodes		Flags	Beschreibung
LDA	imm	$A9	N, C	Lädt einen Wert in den Akkumulator A, entweder direkt oder von einer Speicherstelle.
	zp	$A5		
	zp,X	$B5		
	abs	$AD		
	abs,X	$BD		
	abs,Y	$B9		
	(ind,X)	$A1		
	(ind),Y	$B1		
LDX	imm	$A2	N, C	Lädt einen Wert in das Index-Register X
	zp	$A6		
	zp,Y	$B6		
	abs	$AE		
	abs,Y	$BE		
LDY	imm	$A0	N, C	Lädt einen Wert in das Index-Register Y
	zp	$A4		
	zp,X	$B4		
	abs	$AC		
	abs,X	$BC		
STA	zp	$85	—	Kopiert den Wert des Akkumulators Aan eine Speicherstelle.
	zp,X	$95		
	abs	$8D		
	abs,X	$9D		
	abs,Y	$99		
	(ind,X)	$81		
	(ind),Y	$91		
STX	zp	$86	—	Kopiert den Wert des Akkumulators Aan eine Speicherstelle.
	zp,Y	$96		
	abs	$8E		
STY	zp	$84	—	Kopiert den Wert des Akkumulators Aan eine Speicherstelle.
	zp,X	$94		
	abs	$8c		
TAX	impl	$AA	—	Kopiert den Wert des Akkumulators Ain das Index-Register X.

Tabelle 4: Instruktionen für den Datentransfer

Mnemonic	Adressierungsarten und Opcodes		Flags	Beschreibung
`TAY`	`impl`	`$A8`	—	Kopiert den Wert des Akkumulators `A`in das Index-Register `Y`.
`TXA`	`impl`	`$8A`	—	Kopiert den Wert des Index-Registers `X`in den Akkumulator `A`.
`TYA`	`impl`	`$AA`	—	Kopiert den Wert des Index-Registers `Y`in den Akkumulator `A`.
`TSX`	`impl`	`$BA`	—	Kopiert den Wert des Stack-Register `SP`in das Index-Register `X`.
`TXS`	`impl`	`$AA`	—	Kopiert den Wert des Index-Registers `X`in das Stack-Register `SP`.

Tabelle 5: Instruktionen für arithmetisch-logische Operationen

Mnemonic	Adressierungsarten und Opcodes		Flags	Beschreibung
ADC	imm	$69	N, Z, C, V	Addiert einen Wert zu demjenigen im Akkumulator A, entweder direkt oder von einer Speicherstelle.
	zp	$65		
	zp,X	$75		
	abs	$6D		
	abs,X	$7D		
	abs,Y	$79		
	(ind,X)	$61		
	(ind),Y	$71		
SBC	imm	$A9	N, Z, C, V	Subtrahiert einen Wert von demjenigen im Akkumulator A, entweder direkt oder von einer Speicherstelle.
	zp	$A5		
	zp,X	$B5		
	abs	$AD		
	abs,X	$BD		
	abs,Y	$B9		
	(ind,X)	$A1		
	(ind),Y	$B1		
INC	zp	$A5	N, Z	Erhöht den Wert in einer Speicherstelle um 1.
	zp,X	$B5		
	abs	$AD		
	abs,X	$BD		
INX	impl	$A9	N, Z	Erhöht den Wert des Index-Register Xum 1.
INY	impl	$A9	N, Z	Erhöht den Wert des Index-Registers Yum 1.
DEC	zp	$C6	N, Z	Vermindert den Wert in einer Speicherstelle um 1.
	zp,X	$D6		
	abs	$CE		
	abs,X	$DE		
DEX	impl	$CA	N, Z	Vermindert den Wert des Index-Register Xum 1.
DEY	impl	$88	N, Z	Vermindert den Wert des Index-Registers Yum 1.

Tabelle 5: Instruktionen für arithmetisch-logische Operationen

Mnemonic	Adressierungsarten und Opcodes		Flags	Beschreibung
AND	imm	$29	N, Z	Berechnet das bitweise logische Und des Akkumulator A, entweder direkt oder mit einer Speicherstelle.
	zp	$25		
	zp,X	$35		
	abs	$2D		
	abs,X	$3D		
	abs,Y	$39		
	(ind,X)	$21		
	(ind),Y	$31		
ORA	imm	$09	N, Z	Berechnet das bitweise logische Oder[27] des Akkumulator A, entweder direkt oder mit einer Speicherstelle.
	zp	$05		
	zp,X	$15		
	abs	$0D		
	abs,X	$1D		
	abs,Y	$19		
	(ind,X)	$01		
	(ind),Y	$11		
EOR	imm	$49	N, Z	Berechnet das bitweise logische ausschliessliche Oder[28] des Akkumulator A, entweder direkt oder mit einer Speicherstelle.
	zp	$45		
	zp,X	$55		
	abs	$4D		
	abs,X	$5D		
	abs,Y	$59		
	(ind,X)	$41		
	(ind),Y	$51		
ASL	A	$0A	N, Z, C	Schiebt den Inhalt des Akkumulators oder einer Speicherstelle um ein Bit nach links; das höchstwertige Bit ganz links, wird in das Carry-Flag geschoben; das niedrigstwertige Bit ganz rechts wird zu 0.
	zp	$06		
	zp,X	$16		
	abs	$0E		
	abs,X	$1E		

Tabelle 5: Instruktionen für arithmetisch-logische Operationen

Mnemonic	Adressierungsarten und Opcodes		Flags	Beschreibung
LSR	A	$4A	N, Z, C	Schiebt den Inhalt des Akkumulators oder einer Speicherstelle um ein Bit nach rechts, das niedrigstwertige Bit ganz rechts, wird in das Carry-Flag geschoben, das höchstwertige Bit ganz links wird zu 0.
	zp	$46		
	zp,X	$56		
	abs	$4E		
	abs,X	$5E		
ROL	A	$2A	N, Z, C	Rotiert den Inhalt des Akkumulators oder einer Speicherstelle um ein Bit nach links; das höchstwertige Bit ganz links, wird in das Carry-Flag C geschoben; das niedrigstwertige Bit ganz rechts erhält den Wert des Carry-Flags.
	zp	$26		
	zp,X	$36		
	abs	$2E		
	abs,X	$3E		
ROR	A	$6A	N, Z, C	Rotiert den Inhalt des Akkumulators oder einer Speicherstelle um ein Bit nach rechts, das niedrigstwertige Bit ganz rechts, wird in das Carry-Flag geschoben, das höchstwertige Bit ganz links erhält den Wert des Carry-Flags.
	zp	$66		
	zp,X	$76		
	abs	$6E		
	abs,X	$7E		

[27] Gemeint ist wie normalerweise in der Informatik das *inklusive* Oder, das den Wert 1 ergibt, wenn mindestens einer der Operanden 1 ist

[28] Das Entweder-Oder wird in der Informatik auch ausschliessliches Oder (*exclusive or*) genannt. Dieses ergibt den Wert 1, wenn genau einer der Operanden 1 ist.

Tabelle 6: Verzweigungsbefehle

Mnemonic	Adressierungsarten und Opcodes		Flags	Beschreibung
`JMP`	`abs` `(ind)`	`$4C` `$6C`	—	Lädt eine Adresse in den Programmzähler[29]
`JSR`	`abs`	`$20`	—	Schiebt[30] die Rücksprungadresse, also den Wert des Programmzählers plus 2, auf den Stack und lädt eine Adresse in den Programmzähler.
`RTS`	`impl`	`$60`	—	Holt[31] die Rücksprungadresse in den Programmzähler.
`BEQ`	`*rel`	`$B0`	—	Verzweigt, falls der Zero-Flag `Z` gesetzt (1) ist, insbesondere falls bei der letzten Subtraktion oder beim letzten Vergleich der Wert des Registers gleich gross wie der Subtrahend war.
`BNE`	`*rel`	`$D0`	—	Verzweigt, falls der Zero-Flag `Z` gelöscht (0) ist; insbesondere falls bei der letzten Subtraktion oder beim letzten Vergleich der Wert des Registers gleich gross wie der Subtrahend war.
`BMI`	`*rel`	`$B0`	—	Verzweigt, falls der Negativ-Flag `N` gesetzt (1) ist, das heisst, falls bei der letzten Subtraktion oder beim letzten Vergleich der Wert des Registers kleiner als der Subtrahend war.
`BPL`	`*rel`	`$B0`	—	Verzweigt, falls der Negativ-Flag `N` gelöscht (0) ist; insbesondere falls bei der letzten Subtraktion oder beim letzten Vergleich der Wert des Registers mindestens so gross war wie der Subtrahend.
`BCS`	`*rel`	`$B0`	—	Verzweigt, falls der Übertrag-Flag `C` gesetzt (1) ist; insbesondere falls die letzte arithmetische Operation einen Übertrag ergab.

Tabelle 6: Verzweigungsbefehle

Mnemonic	Adressierungsarten und Opcodes		Flags	Beschreibung
`BCC`	`*rel`	`$B0`	—	Verzweigt, falls der Übertrag-Flag `C` gelöscht (0) ist; insbesondere falls die letzte arithmetische Operation keinen Übertrag ergab.
`BVS`	`*rel`	`$B0`	—	Verzweigt, falls der Overflow-Flag `V` gesetzt (1) ist; insbesondere, falls die letzte arithmetische Operationen einen Overflow ergab[32]
`BVC`	`*rel`	`$B0`	—	Verzweigt, falls der Overflow-Flag `V` gelöscht (0) ist; insbesondere, falls die letzte arithmetische Operationen keinen Overflow ergab.

[29] Effektiv wird nach dem Prinzip des *little endian* zuerst das nächste Bit als *Low Byte*und das darauf folgende Byte als *High Byte*in den 16-Programmzähler geladen. Auch hier nimmt der Assembler uns die Details ab.

[30] Gemeint ist ein *push* auf den Stack: Der Stackpointer `SP` wird anschliessend dekrementiert, um auf die nächste freie Stelle zu zeigen.

[31] Gemeint ist ein *pull* vom Stack, das heisst der Stackpointer `SP` wird zuerst inkrementiert, damit er auf das letzte Element zeigt.

[32] Die betrifft nur die vorzeichenbehaftete Interpretation dieser Operationen und kann im Unterricht schadlos übergangen werden

Tabelle 7: Befehle für die Flag-Manipulation

Mnemonic	Adressierungsarten und Opcodes		Flags	Beschreibung
SEC	impl	$38	C	Setzt den Carry-Flag C[33].
SEC	impl	$18	C	Löscht den Carry-Flag C[34].
SED	impl	$F8	C	Setzt den D-flag für den Dezimalmodus. Ist dieser gesetzt, werden die Argumente der Addition und Subtraktion nicht normal hexadezimal sondern als Paare von Dezimalzahlen interpretiert[35]
CLD	impl	$D8	C	Löscht den D-Flag.
SEI	impl	$78	C	Setzt den I-flag. Ist dieser gesetzt, werden normale Interrupts ignoriert und nur die nicht-maskierbaren Interrupts beachtet[36]
CLI	impl	$F8	C	Löscht den I-Flag.
CLV	impl	$F8	C	Löscht den Overflow-Flag V[37].

[33] Dieser Befehl sollte vor jeder Subtraktion ausgeführt werden, um die erwarteten Resultate zu garantieren.

[34] Dieser Befehl sollte vor jeder Addition ausgeführt werden, um die erwarteten Resultate zu garantieren.

[35] Für den Unterricht empfiehlt es sich sicher, den Dezimalmodus zu ignorieren

[36] Das ganze Thema der Interrupt-Behandlung geht wohl über den Zweck des gymnasialen Unterrichts hinaus.

[37] Der Overflow-Flag wird für die Interpretation der arithmetischen Operationen verwendet, die im gymnasialen Unterricht schadlos übergangen werden kann.

Literatur

[Cox11] Russ Cox. *The MOS 6502 and the Best Layout Guy in the World*. 2011. URL: https://research.swtch.com/6502 (besucht am 01. 03. 2024).

[Gre10] Brian Silverman Greg James Barry Silverman. *The Visual 6502*. 2010. URL: http://www.visual6502.org/JSSim/index.html (besucht am 19. 01. 2024).

[Jac08a] Andrew Jacobs. *Instruction Reference*. 2008. URL: http://www.6502.org/users/obelisk/6502/reference.html (besucht am 19. 01. 2024).

[Jac08b] Andrew Jacobs. *The Instruction Set*. 2008. URL: http://www.6502.org/users/obelisk/6502/instructions.html (besucht am 19. 01. 2024).

[Lan02] Norbert Landsteiner. *6502 Instruction Set*. 2002. URL: https://www.masswerk.at/6502/6502_instruction_set.html (besucht am 19. 01. 2024).

[Sal22] James Salvino. *6502Asm, An Assembler for the 6502 Microprocessor*. 2022. URL: https://github.com/SYSPROG-JLS/6502Asm (besucht am 01. 03. 2024).

[Shi12] Ken Shirriff. *The 6502 overflow flag explained mathematically*. 2012. URL: http://www.righto.com/2012/12/the-6502-overflow-flag-explained.html (besucht am 19. 01. 2024).

[Wil24] Garth Wilson. *Investigating Interrupts*. 20024. URL: http://www.6502.org/users/obelisk/6502/addressing.html (besucht am 01. 03. 2024).