Aufbau und Funktionsweise eines Prozessors

Facharbeit im Fach Informatik
(Fachlehrerin Frau Lohmann)

von
Leon Adam

Schuljahr 2016/2017

Inhaltsverzeichnis

Anlagen
Dokumentation der virtuellen Maschine.
Datenträger mit diesem Dokument sowie Quellcodes.

Abkürzungen, Synonyme, Zahlensysteme, etc.

In dieser Facharbeit wurden folgende Abkürzungen und Synonyme gewählt:

Wort	**Synonym(e)**
Prozessor	CPU, Mikroprozessor
MSB	Je nach Kontext: *Most Significant Bit/Byte* (höchstwertiges Bit/Byte)

Für verschiedene Zahlensysteme wurden folgende Schreibweisen genutzt:

Zahlensystem	**Format**
Hexadezimal (16er)	...h

1. Einleitung

Keine Technik hat die Menschheit in den vergangenen Jahren so beeinflusst wie die Digitaltechnik.
Eine besondere Rolle kommt hierbei nicht zuletzt jegliche Form von Prozessoren – vom ausgewachsenen System-on-Chip (SoC) aus dem Smartphone bis zum kleinen Mikrocontroller (z.B. in Funkmäusen) – zu.

Der Weg zu diesen Wunderwerken der Technik ist allerdings relativ lang und begann bereits im 17. Jahrhundert, zunächst allerdings rein mechanisch mit den Rechenmaschinen von Wilhelm Schickard und Plaise Pascal. Sie beherrschten bereits einfache mathematische Operationen.

Einen weiteren Schritt vorwärts machte 1941 Konrad Zuse mit dem Z3. Der Z3 besteht aus elektromagnetischen Relais und hatte eine (für heutige Verhältnisse langsame) Taktfrequenz von 5 Hertz. Die Programmierung erfolgte dabei über einen Lochstreifen. Die Geschwindigkeit war jedoch deutlich begrenzt, vor allem aufgrund der Relais, die ebenfalls mechanisch arbeiten (eine Spule, an der Spannung anliegt, erzeugt ein Magnetfeld, wodurch ein Anker an einen Kontakt gedrückt wird).

Der nächste Schritt in Richtung moderner Technik erfolgte u.a. mit dem berühmten Rechner „COLOSSUS“, der während des zweiten Weltkriegs ENIGMA-verschlüsselte Nachrichten der Deutschen dechiffrierte. Im Gegensatz zum Z3 arbeitet COLOSSUS mit Elektronenröhren. Ein „Mitarbeiter“ an dem Projekt war der bekannte Mathematiker Alan Turing.

Ein weiterer „Röhrenrechner“ war der ENIAC, an dem auch John von Neumann mitarbeitete, der später das Projekt verließ und die *von-Neumann-Maschine* entwickelte. Durch das neuartige Konzept, bei dem Programm und Daten im Arbeitsspeicher liegen, wurde die Programmierung deutlich einfacher. Bis heute werden Prozessoren nach diesem Prinzip entwickelt.

Den wahrscheinlich wichtigsten Schritt war aber die Entwicklung der Halbleitertechnik, insbesondere der Transistoren durch Bardeen, Brattain und Shockley 1948.
Der Vorteil von Transistoren und anderen Halbleiter: Sie sind klein, billiger und verbrauchen weniger Strom (Röhren müssen oft mit hohen Spannungen beheitzt werden). [a]

[a] Klaus Wüst: *Mikroprozessortechnik. Grundlagen, Architekturen, Schaltungstechnik und Betrieb von Mikroprozessoren und Mikrocontrollern.* Vieweg+Teubner Verlag. 4. Auflage, 2011. (ISBN 978-3-8348-0906-3)

23 Jahre später kam dann von Intel der 4004 auf den Markt, der erste „echte“ Mikroprozessor mit eine sehr kleinen Wortbreite von nur 4 Bit, 2.300 Transistoren und einer Taktfrequenz von 108 kHz.
Ein historischer Meilenstein bildet hier auch der MOS Technology 6502 von 1975 (4.000 Transistoren), der die Basis für viele bekannte Heimcomputer legte, z.B. den Commodore VC-20 oder den Commodore 64 (6510, Derivat des 6502).
Auch zu nennen ist in diesem Zusammenhang der 80386 von Intel aus dem Jahr 1985, welcher mit seinen nun 275.000 Transistoren erstmals Multitasking fähig war und auch heute noch weiterentwickelt in jedem Computer steckt. [b]

Betrachten wir Prozessoren letztlich von der untersten Ebene, so sind diese eigentlich nur Rechenmaschinen, die aus winzigen Schaltern (Transistoren) bestehen. Spielen wir also auf unserem Handy, ist dies lediglich ein Produkt von Schaltvorgängen, die das Berechnen von Zahlen bewirken.

2. Konzept

Für jedes Problem gibt es verschiedene Lösungen – so auch bei Prozessoren. Daher lassen sie sich auch in verschiedene Gruppen einteilen. Dabei teilt man Prozessoren nicht nach der Leistung oder der Anzahl der Kerne, sondern viel mehr nach grundlegenden Konzepten.
Hierzu gehören u.a. die Komplexität (siehe Kapitel 2.1.) und Aufteilung des Speichers (siehe Kapitel 2.2.). Darüber hinaus kann man Prozessoren auch noch danach unterteilen, ob es sich um Universalprozessoren oder Spezialprozessoren handelt. Beispielsweise gibt es sowohl Universalprozessoren für Smartphones als auch Spezialprozessoren, die z.B. zur Basisband-Signalgenerierung für Funksignale eingesetzt werden.

2.1. RISC & CISC

Betrachtet man Prozessoren, unterscheidet man vor allem zwischen RISC (**R**educed **I**nstruction **S**et **C**omputing) und CISC (**C**omplex **I**nstruction **S**et **C**omputing).

Das RISC-Prinzip zeichnet sich vor allem durch Minimalismus aus. Es gibt nur wenige und einfache Befehle, die dafür besonders schnell ausgeführt werden können. Sollen komplexere Operationen durchgeführt werden, müssen mehrere Befehle in Reihe benutzt werden.

Das Gegenteil bildet das CISC-Prinzip: Es wird versucht, einen möglichst mächtigen Befehlssatz (viele, komplexe Befehle) bereitzustellen, so dass für jede Operation ein Befehl zur Verfügung steht.
Der wesentliche Nachteil dieses Konzeptes ist der erhöhte Aufwand bei der Beschaltung. Aus diesem Grund ist es üblich, speziellere Befehle in Form von einem Programm (sog. „Mikrocode“) zu implementieren. Wird nun beispielsweise solch ein Befehl benutzt, führt der Prozessor ein kleines Programm aus, welches wiederum aus einfachen Befehlen besteht (sog. „Mikrobefehle“). [1]

[b] pc-erfahrung.de: *Geschichte und Entwicklung der Prozessoren seit dem Intel 4004.* (http://www.pc-erfahrung.de/prozessor/cpu-historie.html)
[1] Christian Richter: *Einführung in Aufbau und Funktionsweise von Mikroprozessoren.* (http://www.ch-r.de/et/nue-atmmk-mikroprozessoren.pdf)

In der Praxis findet man selten pure RISC oder CISC-Prozessoren. So gelten die bekannten ARM-Prozessoren z.B. als RISC-Prozessoren, obwohl diese inzwischen auch meistens schon komplexe Erweiterungen besitzen (bsp. zur digitalen Signalverarbeitung).

Eine Liste (weniger) typischer RISC-Prozessoren und deren Anwendungsgebiete: [2]

- ARM (**A**dvanced **R**ISC **M**achine): 32-bit (nun auch 64-bit) Prozessoren mit Stromsparfunktionen, vor allem bei Mobil- und IoT-Geräten häufig zu finden.
- MIPS (**M**icroprocessor without **i**nterlocked **p**ipeline **s**tages): Meistens bei eingebetteten System eingesetzt, vor allem Router, aber auch Spielekonsolen wie z.B. die Sony PlayStation oder die Nintendo 64.

Als Beispiel für einen CISC-Prozessor ist der bei Computern verwendete x86 zu nennen. Der Name leitet sich von der 80**86**-Reihe ab. Der Befehlssatz wurde im Laufe der Entwicklung immer wieder erweitert und wächst auch noch heute. [3]

2.2 Speicherkonzepte

Eine weitere Unterteilung lässt sich bei der Aufteilung in Daten- und Programmspeicher treffen.
Hier unterscheidet man zwischen der Harvard- und von Neumann-Architektur [4].

Bei dem Harvard-Modell ist der Befehlsspeicher physikalisch vom Datenspeicher getrennt,
d.h. dass die Befehle in einem anderen Speicher liegen als die Daten (z.B. Zeichenketten etc.).
Zum einen bietet dies einen Geschwindigkeitsvorteil (es können gleichzeitig Befehle und Daten geladen werden), zum anderen erhöht dies aber auch die Komplexität, sowohl in der Programmierung als auch in dem Aufbau des Prozessors. [5]

Das Gegenmodell stellt die von Neumann-Architektur dar. Befehlsspeicher und Datenspeicher sind nicht von einander getrennt. Ein Vorteil besteht darin, dass die Programmierung deutlich einfacher wird, allerdings müssen Befehle und Daten nun sequenziell geladen werden, wodurch sich die Geschwindigkeit reduziert.

[2] *RISC Vendor Examples.* (http://web.uvic.ca/~epurcell/group_web_page.html)
[3] *X86-Prozessor.* (https://de.wikipedia.org/wiki/X86-Prozessor)
[4] Christian Richter: *Mikroprozessoren. Aufbau und Funktionsweise.* (http://www.ch-r.de/et/nue-atmmk-mikroprozessoren_folien.pdf)
[5] Dr. Andreas Müller: *Rechnerarchitektur.* (https://www.tu-chemnitz.de/informatik/friz/Grundl-Inf/Rechnerarchitektur/Vorlesung/vorlesung_1.pdf)

3. Aufbau

3.1. Schaltungstechnik

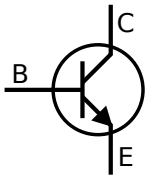
Betrachtet man moderne (Digital-)Technik, wird sofort deutlich, dass es sich bei Prozessoren und anderen Chips um komplexe Schaltung handelt. Aus diesem Grund werden „Chips“ unter Elektronikern auch als ICs (**I**ntegrated **C**ircuits) – also integrierte Schaltkreise – bezeichnet.

Die Basis dieser Schaltkreise bilden sog. Halbleiterelemente. Diese Elemente bestehen aus Stoffen, die bei 0 Kelvin (SI-Einheit für Temperatur), also -273,15 Grad Celsius, nicht leiten. Erhitzt man den Stoff z.B. durch Strom, wird diese leitend, d.h. es kann Strom fließen. [6]

3.1.1. Transistoren

Eine wichtige Basis für die heutige Digitaltechnik bilden Transistoren.
Wie der Name schon impliziert, können Transistoren entweder zu Verstärken von Leistungen (Transistorverstärker, Transistorradio etc.) oder zum Schalten eingesetzt werden.
In letzterem Fall arbeiten diese dann ähnlich wie Relais. Ein klassischer bipolarer Transistor hat drei Anschlüsse:

[7]



- Kollektor (C): Die Anode.
- Emitter (E): Die Kathode.
- Basis (B): Die Basis des Transistors. Liegt hier ein passender Strom/Spannung an, leitet der Transistor den Strom über den Kollektor und den Emitter.

3.1.2. Logikgatter

Ein weiterer wichtiger Schritt zum Mikroprozessor ist die Entwicklung der Logikgatter, welche eine Anwendung der Transistoren darstellen.
Logikgatter sind letztlich eine Implementierung von logischen Funktionen (z.B. UND, s. weiter unten), die beispielsweise in dem Prozessor zur Dekodierung von Befehlen oder bei arithmetischen Operationen verwendet werden.
Meistens besitzt ein Logikgatter zwei Eingänge (im Folgenden als X und Y bezeichnet) und einen Ausgang (im Folgenden als A bezeichnet).
Alle Ein- und Ausgänge kennen dabei nur zwei Zustände: wahr oder falsch bzw. „HIGH“ oder „LOW“. Die Bezeichnungen „HIGH“ und „LOW“ stammen aus der Elektrotechnik und beschreiben zwei verschiedene Spannungspegel.
Bei einer Transistor/Transistor-Logik-Schaltung (TTL-Schaltung) beträgt die Spannung für den HIGH-Zustand (binäre 1) größer als 2V (max. ca. 5V), für den LOW-Zustand (binäre 0) kleiner als 0,8V (max. ca. -0,5V). [8]

[6] Universität Oldenburg: *Exkurs: Was ist ein Halbleiter und was bedeutet Dotieren?* (http://olli.informatik.uni-oldenburg.de/weTEiS/weteis/halbleiter.htm)
[7] Omegatron: *Symbol for an NPN BJT.* Lizenz: CC-BY-SA 3.0 (https://upload.wikimedia.org/wikipedia/commons/e/e8/IGFET_N-Ch_Dep_Labelled.svg)
[8] Dipl.-Ing. Reinhard Gößler: *Elektronik – praxisnah.* Franzis-Verlag München, 1977. (ISBN 3-7723-6281-8)

Typische logische Operationen sind u.a.:

- **UND**: Sind sowohl X als auch Y wahr, ist A wahr.
  - $A = X \wedge Y$
- **ODER**: Sind X oder Y wahr, ist A wahr.
  - $A = X \vee Y$
- **NICHT**: Ist X wahr, ist A falsch. Ein zweiter Eingang existiert nicht.
- **XOR**: Wie **ODER**. Allerdings ist A nicht wahr, wenn sowohl X als auch Y wahr sind.

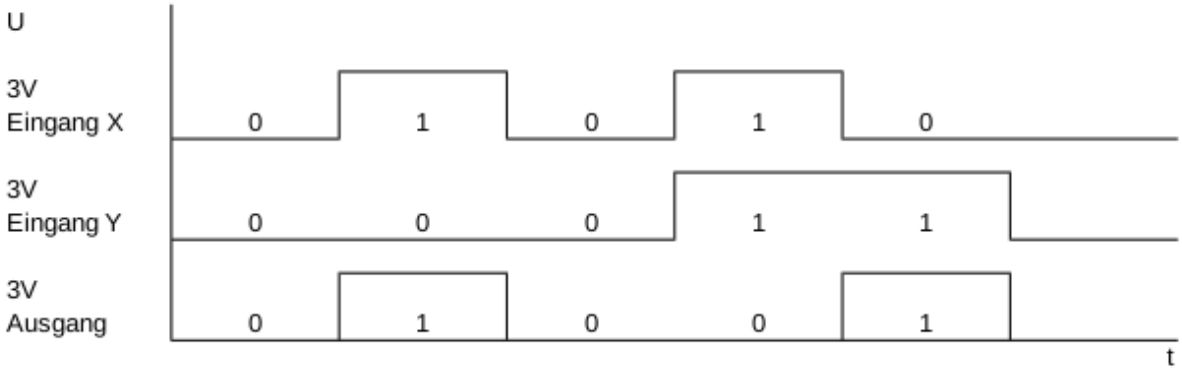


Abbildung 1: Spannungen am XOR-Gatter (mit 3V TTL-Pegeln).

Ein besonderes Gatter ist der sog. „**Flipflop**“. Ein Flipflop ist eine Art 1-Bit Speicher, d.h. es gibt wieder nur einen Ausgang der zwei Zustände annehmen kann (0 oder 1). Für eine Ausgangsänderung ist dabei ein ganzer Eingangsimpuls erforderlich. [9]
Steht der Flipflop also auf 1, muss zunächst der Eingang auf 1 und dann wieder auf 0 springen.
Danach sollte der Ausgang auf 0 stehen.
An dieser Stelle wird auch deutlich, was die Hertzangaben bei Prozessoren bedeuten: Um so höher
die Taktung (Frequenz) des Prozessors, desto schneller kann dieser Schalten und damit rechnen.

3.2. Einheiten

Während sich die vorherigen Kapitel sich fast ausschließlich mit den elektrotechnischen Aspekten von Prozessoren und Halbleitern im allgemeinen beschäftigt haben, sollen die folgenden Kapitel einen Blick auf den konkreten Aufbau behandeln.

3.2.1 Register

Eine wichtige Rolle bei Prozessoren spielen Register. Bei diesen handelt es sich um Speicherzellen mit einer festen Größe, die entweder „generelle“ Aufgaben (General-purpose registers) oder spezielle Aufgaben erfüllen. Diese Größe hängt von der Wortbreite ab. Bei heutigen Personal Computer und auch Smartphones 32- oder 64-bit, wobei für bestimmte Aufgaben auch heute noch geringere Wortbreiten von beispielsweise 8- oder 16-bit verwendet werden.

[9] Dipl.-Ing. Reinhard Gößler: *Elektronik – praxisnah.* Franzis-Verlag München, 1977. (ISBN 3-7723-6281-8)

General-purpose Register halten beispielsweise die Operanden oder das Ergebnis bei Rechenoperationen, können aber auch etwaige Addressen speichern.
Spezielle Register hingegen sind bestimmten Aufgaben zugeordnet, hierzu gehört z.B. das Speichern von Befehlen, der aktuellen Addresse des Befehls etc.

3.2.2 Steuerwerk

Bei dem Steuerwerk handelt es sich um die wichtigste Einheit in einem Prozessor. Es koordiniert den gesamten Ablauf und die „Kommunikation“ der Einheiten untereinander.
Das Steuerwerk ist auch für die Dekodierung der Befehle und das Holen der Befehle aus dem Speicher zuständig. [10]

Steht z.B. das Register, in dem die Addresse des aktuellen Befehls gespeichert ist (sog. Programmzähler) auf 1502h (h = Hexadezimal), holt sich das Steuerwerk aus dem Speicher den Befehl an der Stelle 1502h. Ist dieser Befehl beispielsweise die Addition aus zwei Registern B und C in das Register A, so übergibt das Steuerwerk dem Rechenwerk (s. Kapitel 3.2.3) die Werte der Register B und C, worauf das Rechenwerk diese addiert. Das Ergebnis speichert das Steuerwerk dann in Register A ein.

3.2.3 Rechenwerk (ALU)

Das Rechenwerk – oder auch kurz ALU für **A**rithmetical **L**ogical **U**nit – ist die eigentliche „Rechenmaschine“. Es führt die eigentlichen arithmetischen Operationen (Addition, Subtraktion etc.) durch. Meistens besitzt das Rechenwerk zwei Eingänge und einen Ausgang, außerdem sind die Operanden-Register sowie das Ziel-Register frei wählbar.
Es gibt jedoch auch Prozessoren – z.B. der MOS 6502 [11] – bei dem lediglich ein Operanden-Register frei wählbar ist, d.h. das andere Operanden-Register sowie das Ziel-Register sind immer gleich (meistens Akkumulator genannt).

3.3 Bussystem

Ein sehr bekanntes Prinzip in der Informtik ist das sogenannte „EVA-Prinzip“.
„EVA“ steht hierbei für *Eingabe*, *Verarbeitung* und *Ausgabe,* d.h. jedes Programm nimmt Daten vom Nutzer auf, verarbeitet die Daten und gibt diese aus. Ähnlich verhält es sich auch bei einem Prozessor. Um jedoch die Ein- und Ausgabe (typsicherweise auch I/O genannt – von engl. Input/Output) realisieren zu können, gibt es das Bussystem. Das Bussystem ist besteht aus mehreren digitalen Leitungen, die entweder Daten oder Adressen „leiten“.
So ist es gängig, diese in einen Daten- und Adressbus aufzuteilen.

[10] Schöningh Verlag: *Informatik 2*. 2015. S.234f. „Die digitale Welt 101 – Maschinennahe Programmierung“. (ISBN 978-3-14-037127-8)
[11] MOS TECHNOLOGY INC.: *MCS6500 MICROPROCESSORS PRELIMINARY DATASHEET.* Mai 1976. (Ersatzlink: http://archive.6502.org/datasheets/mos_6500_mpu_preliminary_may_1976.pdf)

Beispiel: Eine Eingabeeinheit ist an das Bussystem (8-Bit) angeschlossen und über die Adresse F0h ansprechbar. Möchte die CPU jetzt den Wert 1Fh an die Eingabeeinheit übergeben, legt die CPU die passenden Spannungen für F0h an den Adressbus (im folgenden als Array aus Bits, MSB zuerst: [1, 1, 1, 1, 0, 0, 0, 0]) und analog zum Addressbus an den Datenbus die passenden Spannungen für 1Fh an. Die Eingabeeinheit reagiert dann dementsprechend auf den Befehl.

3.4 Speicher

Eine weitere Beziehung zum Bussystem besteht bei dem Speicher, da dieser ebenfalls an das Bussystem angebunden ist.
Spricht man bei Prozessoren vom Speicher, so unterteilt man diesen in den Programm- und Datenspeicher. Im Programmspeicher wird der eigentliche Maschinencode gespeichert, während im Datenspeicher etwaige Ressourcen sowie für die Laufzeit benötigte Daten – z.B. Variablen – gespeichert werden. Eine besondere Art des Datenspeichers stellt hierbei der Stack dar, in dem z.B. Rücksprungadressen sowie lokale Datenstrukturen gespeichert werden (s. Kapitel 3.4.1 Der Stack).

Im Gegensatz zu Kommunikationsschnittstellen (z.B. Internetanschluss etc.) bemisst man den Speicher nicht in bits sondern in bytes.

3.4.1 Der Stack

Eine besondere Form des Datenspeichers ist der Stack (im deutschen Sprachraum auch Stapelspeicher genannt). Er ist wie ein Stapel aufgebaut, er arbeitet also nach dem *LIFO-Prinzip* („das, was als letztes hineinkommt, kommt als erstes wieder heraus“, Last In First Out).
In einem Prozessor kommt dem Stack eine besondere Rolle zu. Er hält zum einen lokale Daten (z.B. den Wert von Registern), zum anderen aber auch Rücksprungadressen. Rücksprungaddressen stellen die Speicherstellen dar, an denen nach der Ausführung einer Routine der Programmablauf fortgesetzt werden soll.
Meistens wird der Stack in Form eines zusätzlichen Registers realisiert, das in der Regel einen Zeiger auf einen Speicher darstellt (so z.B. bei dem 8086 als „SP“ [12]).

Auf der nächsten Seite befindet sich ein Beispiel für die Anwendung des Stacks. Ein Systemprogrammierer möchte eine Routine aufrufen. Der Routine muss ein 16-bit Wert übergeben werden, dies geschieht über den Stack. Um nun einen Wert auf dem Stack zu speichern, verwendet der Entwickler den „PUSH“-Befehl und gibt als Wert 1234h an. Die CPU liest den Befehl und speichert die Zahl auf dem Stack, indem diese den Stack Pointer (SP) um 1 verringert und dann das erste Byte auf dem Stapel speichert. Dieser Vorgang wiederholt sich für das zweite Byte.

[12] intel: *The 8086 Family User‘s Manual.* Oktober 1979. (Ersatzlink: http://bitsavers.informatik.uni-stuttgart.de/pdf/intel/8086/9800722-03_The_8086_Family_Users_Manual_Oct79.pdf)

Vor PUSH 1234h		
	Adresse	Inhalt
	1100h	40h
SP →	10FFh	10h

Nach PUSH 1234h		
	Adresse	Inhalt
	1100h	40h
	10FFh	10h
	10FEh	12h
SP →	10FDh	34h

Die Methode kann nun auf diesen Wert zugreifen, indem es einen „POP“-Befehl benutzt. In diesem Fall würde nun das erste Byte in das festgelegte Register gespeichert und der Stack Pointer wieder erhöht. Nun würde das zweite Byte vom Stack in das höherwertige Byte des Registers geladen und SP wird erneut inkrementiert.

4. Funktionsweise

Nachdem in den vorherigen Kapiteln vor allem der Aufbau im Vordergrund stand, soll nun die Funktionsweise eines Prozessors erläutert werden. Dabei wird auch auf die Programmierung von Prozessoren und Computern eingegangen, jedoch nur verallgemeinert auf einfache Mikroprozessoren.

4.1 Maschinencode

Im Kapitel 3 wurde bereits deutlich, dass ein Prozessor Befehle ausführt. Diese Befehle, häufig auch Instruktionen genannt, sind im Maschinencode kodiert, d.h. alle Befehle sind in der Regel binär und in einem festgelegten Format gespeichert. Dieses Format variiert je nach Architektur. Fast immer besteht ein Befehl aber zunächst aus einem speziellen Bitmuster, dem sog. Operationscode (kurz Opcode), der die durchzuführende Aktion beschreibt.
Die CPU lädt diesen Opcode (Opcode Fetch oder auch Befehlszyklus) und dekodiert diesen. Je nach Architektur und Befehl werden noch weiter Daten, zumeist Operanden, nachgeladen.
Addiert beispielsweise eine Instruktion einen als Operanden festgelegten Wert zum Akkumulator hinzu, muss der Prozessor noch den Operanden nachladen, der je nach Prozessor z.B. 16-bit lang sein kann.
Nach jedem Laden von Opcodes oder Operanden wird der Programmzähler, der die Adresse des aktuellen Opcodes/Operanden hält, um einen gewissen Wert erhöht, sodass am Ende der Programmzähler auf den Opcode des nächsten Befehls zeigt. [13]

[13] Klaus Wüst: *Mikroprozessortechnik. Grundlagen, Architekturen, Schaltungstechnik und Betrieb von Mikroprozessoren und Mikrocontrollern.* Vieweg+Teubner Verlag. 4. Auflage, 2011. (ISBN 978-3-8348-0906-3)

Da der Maschinencode jedoch schwierig für Menschen zu lesen ist, gibt es eine menschlich lesbare Darstellung die „Mnemocode“ genannt wird. Ein Beispiel für solch einen Mnemocode ist der bereits genannte, fiktive Befehl „PUSH“, der zum Beispiel den Opcode 2Ah haben kann.
Um diese Mnemocodes nun in Maschinencode umzuwandeln gibt es sog. Assembler, die aus dem Quelltext den fertig ausführbaren Maschinencode generieren. Andersherum gibt es auch sog. Disassembler, die Maschinencode in Mnemocode umwandeln.

4.2. Programmablauf

Wie am letzten Kapitel deutlich wird, versteht die CPU nur Maschinencode in Form von Befehlen, die (normalerweise) nacheinander ausgeführt werden. Wie können wir nun z.B. mit einer
Programmiersprache wie C++ einen Prozessor programmieren?
Hierfür gibt es Compiler, die in der Regel aus dem jeweiligen Code der Programmiersprache Assembler-Code generieren, der dann wiederum aus den einzelnen Mnemocodes einen Maschinencode generiert. Dabei gibt es bei modernen Systemen meistens einen Einstiegspunkt, an dem die Programmausführung beginnt.

Will ein heutiger „PC-Prozessor“ (z.B. x86-kompatible) das Programm ausführen, lädt das Betriebssystem es zunächst in den RAM, setzt einen eigenen geschützen Adressraum auf (damit Programme sich nicht gegenseitig direkt beeinflussen können) und trifft weitere Vorkehrungen die zur Ausführung des Programms nötig sind. Nun springt die CPU zum Programm und führt es damit aus.

4.2.1 Programmsprünge

In manchen Situation ist es sinnvoll, an andere Programmstellen zu springen. Dies ist insbesondere bei Kontrollstrukturen der Fall. Hier gibt es je nach Architektur verschiedene Lösungen des Problems. Allerdings ist es üblich Befehle einzuführen, die (bedingt) an eine bestimmte Adresse springen.

Zum Beispiel könnte folgender C-Code:

```
if (x == 42)
{
    aktion1();
}
else
{
    aktion2();
}
```

Folgenden ARM-Assembler-Code erzeugen (angenommen, dass r0 x speichert):
Hinweis: Dieser Code wurde „von Hand“ erzeugt. Das Semikolon weist auf einen Kommentar hin.

```
label_xyz:
        push {lr}               ; durch den Befehl bl (sowie bleq
etc.) wird das
                                ; Register LR (=Link Register)
verändert. Es
                                ; hält die Rücksprungadresse der
aufrufenden
                                ; Routine und muss daher umbedingt
                                ; zwischengespeichert werden.
                                ; Hierzu wird der Stack benutzt.


        ld r1, #42              ; r1 = 42
        cmp r1, r0              ; Vergleich von r0 (x) und r1 (42)

        blne falsch             ; else-Fall

wahr:
        bl aktion1              ; springe zu aktion1 und speichere die
                                ; Adresse der nächsten Instruktion in
LR.

        b fertig                ; springe zu fertig, damit nicht der
else-Fall
                                ; abgearbeitet wird


falsch:
        bl aktion2              ; springe zu aktion2 und speichere die
                                ; Adresse der nächsten Instruktion in
LR.


fertig:
        ...                     ; Sobald von aktion1 oder aktion2
                                ; die Kontrolle zurückgeben wird,
                                ; wird dieser code ausgeführt


        pop {pc}                ; Der Anfangszustand von LR wird vom
                                ; Stack genommen und in
                                ; den Programmzähler, der die aktuelle
Adresse
                                ; des Befehls enthält gespeichert,
                                ; also ähnlichen einem return.
                                ;
                                ; Alternativ ist auch folgender Code
denkbar:
                                ; pop { lr }
                                ; bx lr


aktion1:
        ...

aktion2:
        ...
```

4.2.2. Arithmetische Operationen

Eine wichtige Eigenschaft einer CPU ist nicht nur die Möglichkeit von Programmsprüngen, sondern auch das arithmetischen Rechnen. Hierzu hat jeder Prozessor in der Regel seine eigenen Befehle, die es z.B. erlauben, zwei Zahlen miteinander zu addieren. Diese Operationen laufen dabei immer binär ab, sodass es bei vielen Architekturen üblich ist, beispielsweise ein sog. „Carry-Bit“ zu besitzen. Das „Carry-Bit“, welches normalerweise in einem speziellen Register gespeichert wird, zeigt an, ob bei einer Rechenoperation ein Überlauf aufgetreten ist.

Addieren wir z.B. vorzeichenlos die Zahl 64000 zum 16-bit Register A mit dem Wert 3000 hinzu, so kommt es zum Überlauf, weil die höchste Zahl für ein 16-bit Register (ohne Vorzeichen) 65535 beträgt ($2^{16}=65536=[0;65535]$). [13, S. 19f.]

Darüber hinaus gibt es noch weitere Bits in sogenannten Flagregistern, meistens folgende:

- **Negative / Positive / Sign:** Dieses Bit weist auf das Vorzeichen der Rechenoperation (bzw. jeweiligen Registers) hin.
- **Zero / Non-zero:** Ist das Ergebnis (un-)gleich 0, ist das Bit gesetzt. Das kann beispielsweise bei Zählschleifen nützlich sein. Solange das Bit nicht gesetzt ist, wird zum Code in der Schleife zurückgesprungen.

Je nach Architektur gibt es noch weitere Bits.

4.3. Peripherien

Letztlich findet die Benutzerinteraktion mit einem Computer über Eingabe-/Ausgabe-Peripherien statt. Typische Beispiele sind hier Tastatur und Bildschirm bzw. der Touchscreen.
Diese Peripherien sind beim klassischen PC hierbei an dafür gemachte Schnittstellen (z.B. USB, VGA, PS/2 etc.) angeschlossen, die wiederum an die Southbridge angeschlossen sind. Die Southbridge ist dann direkt mit dem Prozessor verbunden. [13, S. 84f.]

Bei der Ansteuerung der Peripherien durch den Programmierer kann man grob zwei Modelle unterscheiden:

- **Memory-mapped I/O:** Die Peripherien können durch das Schreiben und Lesen von Werten von bestimmten Speicheradressen gesteuert werden, die auch Register genannt werden (nicht zu verwechseln mit den Prozessorregistern). Ein- und Ausgabe werden also wie Speicher behandelt.
- **Port I/O**: Die Peripherien werden über sogenannte Ports gesteuert. Es existiert also ein eigener Adressraum für die Ein- und Ausgabe.

Wollen wir nun z.B. Daten von einem seriellen Port empfangen, gibt es i.d.R. zwei verschiedene Möglichkeiten: Entweder wir überprüfen regelmäßig mithilfe eines speziellen Registers, ob Daten verfügbar sind (*Polling*) oder wir warten auf ein Signal von der Peripherie, dass Daten verfügbar sind (*Interrupts*, s. nächstes Kapitel).

4.4. Interrupts

Um schneller auf Benutzereingaben reagieren zu können, gibt es das Interrupt-Konzept. Die Peripherie gibt über einen Anschluss der CPU ein Signal, worauf der Prozessor die Ausführung des aktuellen Programms unterbricht und zur *Interrupt-Service-Routine* (ISR) springt.
Sie verarbeitet den Interrupt möglichst schnell, damit z.B. der Datenpuffer nicht überläuft, und setzt die Ausführung des eigentlichen Programms fort.

Mithilfe des Interrupt-Konzepts kann das *Busy-Waiting* verhindert werden, bei der der Prozessor einen großen Teil der Zeit für das Polling verwendet und somit unzureichend auf die Nutzerinteraktion reagieren kann. [13, S. 112]

Interrupts treten vor allem bei Eingabegeräten auf, beispielsweise wenn der Nutzer eine Taste auf der Tastatur drückt oder auf dem Netzwerkgerät (LAN/WLAN-Adapter etc.) Daten verfügbar sind.

4.5. Programmierung

Wenn wir unser Smartphone benutzen ist dies also letztlich nur das Schalten von Transistoren und Berechnen von Zahlen sowie das Schreiben und Lesen von/zu bestimmten Speicheradressen oder I/O-Ports.

Soll nun ein Programm entwickelt werden, brauchen wir entweder eine Laufzeit-Software (z.B. bei Java) oder einen Compiler (z.B. bei C/C++ usw.). Bei der Verwendung eines Compilers wird der Quelltext zunächst entweder in eine Zwischensprache (*Intermediate Language*) oder in Assemblercode übersetzt. Hiervon wird nun durch einen Assembler der Maschinencode gewonnen. Im letzten Schritt werden die Binärdateien durch einen *Linker* in die ausführbare Datei gebracht.
Meistens werden dabei zusätzliche Informationen in die ausführbare Datei geschrieben, insbesondere, welche Abhängigkeiten (d.h. Bibliotheken) nachzuladen sind.

Programmiert man hingegen in einer Sprache wie Java, so wird überlicherweise der Quelltext in einen *Bytecode* übersetzt. Java-Bytecode ist nicht direkt von der CPU ausführbar, deshalb ist eine zusätzliche Software nötig (die sogenannte *Java-Runtime*), welche den Code entweder interpretiert oder (mit dynamischen Optimierungen) in einen Maschinencode übersetzt (*Just in Time, JIT*).

5. Praktische Programmierung

5.1. Beispiel: Z80-ähnliche (virtuelle) Maschine

Ich möchte die praktische Programmierung einer virtuellen Maschine mit einem Z80-ähnlichen Prozessor erläutern. Eine grobe Dokumentation dieser virtuellen Maschine hängt diesem Dokument bei.

Der Z80 ist ein 8-Bit-Mikroprozessor aus den 70er-Jahren, der binär abwärtskompatibel zum Intel 8080 ist. Er steckt bis heute in einigen Taschenrechnern sowie als abgewandelter LR35902 von SHARP im GameBoy. [14]

Es soll der Quellcode eines Programms (hier als „ROM“ bezeichnet) erläutert werden, dass je nach gedrückter Taste die Farbe verändert. Als Assembler wurde *z80asm* aus den Ubuntu-Paketquellen verwendet, der Code kann jedoch auch mit einem Online-Assembler (z.B. http://www.asm80.com/ oder http://clrhome.org/asm/) unter leichten Anpassung assembliert werden.

(Fortsetzung s. nächste Seite)

[14] Michael Steil: *The Ultimate Game Boy Talk*. 30. Dezember 2016, 33C3 Hamburg. (https://media.ccc.de/v/33c3-8029-the_ultimate_game_boy_talk#video&t=600)

Quellcode-Listing: software/z80_emulator/examples/color-screen.asm

```
ORG $4000

; Vectors used here:
;       $E004 - gpu_fill_screen

ENTRY:
        IN A, ($02) ; get last pressed keyboard character
        LD B, A     ; save A to B for further operations

        SUB 'a'          ; if last character not A
        JP NZ, not_a     ; goto not_a

        LD BC, $4B00     ; count
        LD HL, $0000     ; address
        LD D, $C0        ; value
        CALL $E004       ; gpu_fill_screen

        JP ENTRY

not_a:
        LD A, B          ; restore A
        SUB 'b'
        JP NZ, ENTRY

        LD BC, $4B00     ; count
        LD HL, $0000     ; address
        LD D, $3         ; value
        CALL $E004       ; gpu_fill_screen

        JP ENTRY

        HALT    ; should never be reached
```

Zunächst wird die Startadresse festgelegt. Da der Bereich für die ROM bei 4000h beginnt, wurde hier also „ORG $4000“ geschrieben (*$* denotiert bei diesem Assembler eine hexadezimale Zahl).

Mit dem Label *ENTRY* wird der Einstiegspunkt der ROM definiert. Als nächstes liest das Programm über den I/O-Port 2 ein, welche Taste aktuell gedrückt ist. Das Ergebnis wird in B zwischengespeichert (Zeile 8), damit es für weitere Berechnungen zur Verfügung steht, da A im Laufe noch verändert wird. Danach wird der Wert des Zeichens „a“ von *A* abgezogen (Zeile 10). Ist das Ergebnis hieraus 0, bedeutet dies, dass diese Taste gedrückt ist. Andernfalls springt die CPU zu *not_a* (Zeile 11). Hier wird mithilfe von *B* dasselbe für die Taste „b“ durchgeführt. Ist dies ebenfalls falsch springt das Programm wieder zu „ENTRY“, also zum Programmanfang (*JP (...) ENTRY*).

Sollte einer der beiden Bedingungen erfüllt sein, so wird über die Funktion *gpu_fill_screen*, die an der Adresse E004h verfügbar ist, der Bildschirm vollständig gefüllt. Die Parameter werden mithilfe von 16-bittigen Registerpaaren sowie des Registers D übergeben. Das Registerpaar *BC* hält die Anzahl der zu schreibenden Pixel, *HL* die Adresse innerhalb des Bildschirms (relativ zur Adresse 0) und *D* den Wert der Pixel.

Die letzte Instruktion ist *HALT*. Erreicht die CPU aus irgendeinem Grund diese Instruktion, hält der Prozessor bis zur nächsten Interrupt-Anfrage (IRQ, *Interrupt ReQuest*).

Erklärung

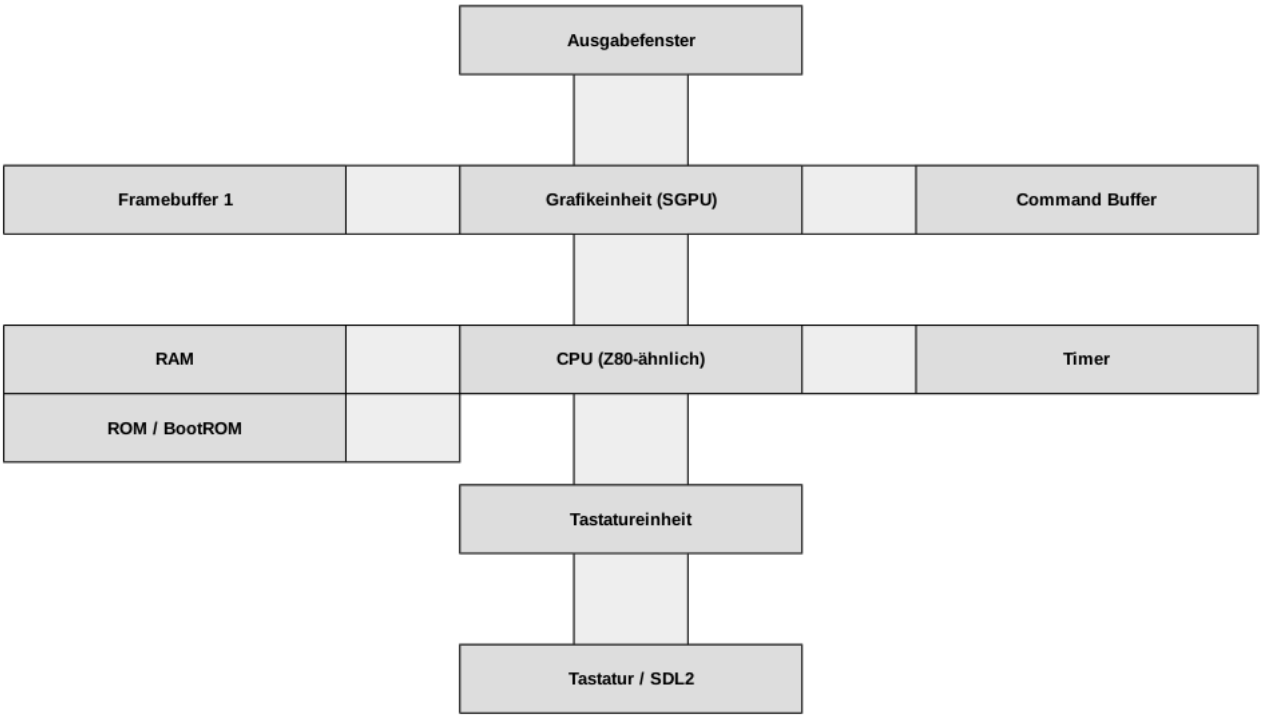
Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die im Literaturverzeichnis angegebenen Hilfsmittel verwendet habe. Ich versichere, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

_____, den_____	_____
Ort Datum	Unterschrift

Z80-ähnliche virtuelle Maschine
„Datenblatt“ und Spezifikation für die Standardkonfiguration
(Revision 1, deutsche Version)

1. Daten
- Z80-ähnliche CPU
- 12 KB RAM
- Grafikeinheit
  - 1 Framebuffer (160x120)
  - Kommando-Interface
- Tastatur-/Eingabeeinheit
- Timer (Zyklus-basiert)
- Banking/Paging-Mechanismus
  - Zugriff auf >64 KB (Boot-)ROM

2. Blockdiagramm



3. Prozessor
- Z80-ähnlicher Befehlsatz
  - unterstütze Opcodes:
    - alle nicht geprefixten Instruktionen außer `djnz *`
    - `RETI (Return from Interrupt).` Opcode: 0xED 4D
    - `DDS (Debug Dump State).` Gibt debug-Informationen über die CPU aus. Opcode: 0xED FF
  - Einen Interrupt mode: Sprung zu 0x0038 und speichern von Adresse zu Stack. (Ähnlich Mode 1 vom Original Z80)

4. Speicher-Layout

	O	S (bytes)	A	
			R	W
Zeropage-RAM	0000h 00FFh	256	X	X
RAM	1000h 3FFFh	12k	X	X
ROM page 0	4000h 5FFFh	8k	X	-
ROM page n	6000h 7FFFh	8k	X	-
Framebuffer	8000h 863Fh	1600	X	X
-	8640h BFFFh		-	-
BootROM page n	C000h DFFFh	8k	X	-
BootROM page 0	E000h FFFFh	8k	X	-

Legende:
O = Offset / Startadresse
S = Size / Größe
A = Attribute (R = Read / Lesen, W = Write / Schreiben)
X = o.g. Zugriff möglich / - = o.g. Zugriff nicht möglich

5. Peripherien

5.1. Tastatureinheit

Die Tastatureinheit erlaubt den Zugriff auf alle relevanten ASCII-Zeichen.

I/O-Port #	**Beschreibung**
0	Wählt das zu prüfende Zeichen aus.
1	Status des Keys (1 = gedrückt, 0 = nicht gedrückt)
2	Zuletzt gedrückter Key (0 = keine Taste gedrückt).

5.2. SGPU

5.2.1. Framebuffer

Der Zugriff auf den Framebuffer erfolgt in Zeilen. Eine Zeile ist 160x10 Zeichen groß, wobei nach dem x-Wert gezählt wird (z.B. entspricht die Adresse 000Fh dem Punkt (15 | 0)).
Das Pixelformat ist wie folgt: RRRG GGBB.

I/O-Ports #	Beschreibung
10	Setzt die Zeile / die Seite des Framebuffers.

5.2.2 Kommando-Interface

Die SGPU bietet ein Kommando-Interface mit dem bestimmte Vorgänge deutlich schneller durchführbar sind. Die Kommunikation erfolgt über einen Puffer, der über unten genannte I/O-Ports zugänglich ist.

I/O-Ports #	Beschreibung
11	Wählt das Byte für #12 im Kommando-Puffer aus. **Zulässige Werte:** 0...255
12	Der aktuelle Wert des ausgewählten Bytes im Puffer.

Allen Kommandos ist folgender Header vorangestellt:

Index #	Beschreibung
0	Status-Register. Ist das erste Bit 1, so wird das festgelegte Kommando ausgeführt (*Trigger-Bit*).
1	Kommando-Byte. Legt das Kommando fest (siehe nachfolgende Tabelle)

Folgende Kommandos sind verfügbar:

Kommando-Byte	Beschreibung	Struktur

01h	Füllt den Bildschirm mit festgelegter Farbe (*value*) von der Anfangsadresse (*address*) *count*-mal.	{
		byte reserved = 0;
		short address;
		short count;
		byte value;
		}

FFh	Zeigt den Erfolg des Kommandos an. (ACK)	-
FEh	Zeigt einen Fehler an. (NACK)	-