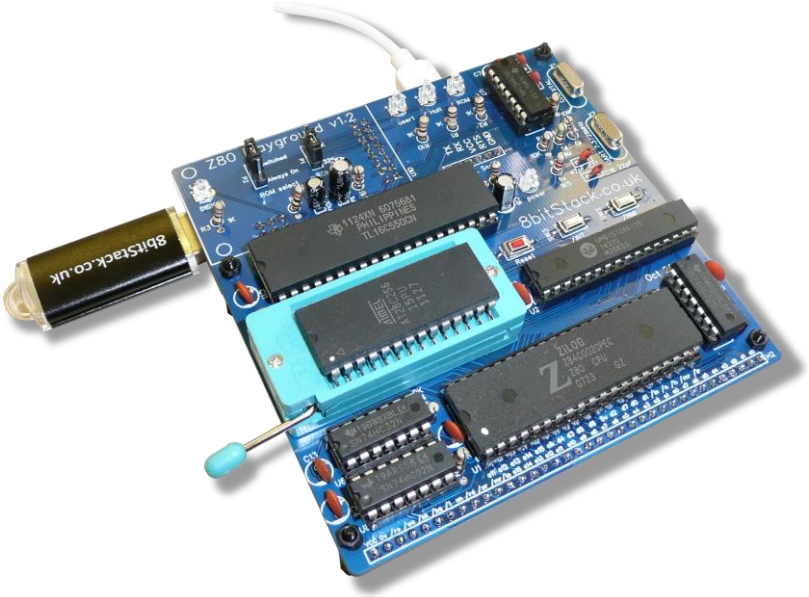
Erik Bartmann

Z80

Der Z80 mit dem Z80-Playground-PCB erklärt

Spaß mit der Programmierung des **Z80**

Eine etwas
andere Herangehensweise



Dieses Dokument	
Titel	Die Programmierung des Z80
Thema	Der Z80
Erstellt am	5. März 2021
Erstellt von	Bartmann, Erik
Version	1.02
Dateiname	Z80

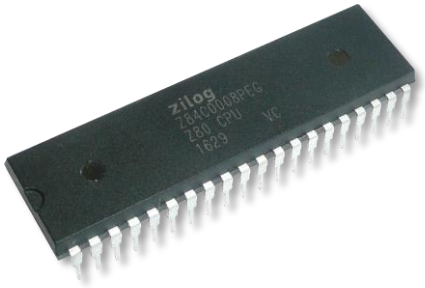
Ihr Ansprechpartner	
Name	Erik Bartmann
E-Mail	erk.bartmann@yahoo.de
Internet	https://erik-bartmann.de/

Inhalt

1 Der Sinn dieses Manuals

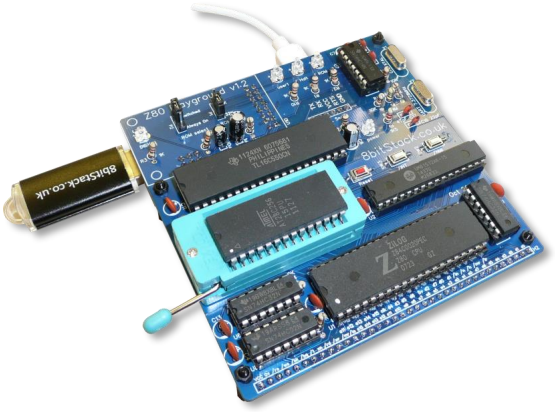
Wenn jemand den Reiz verspürt, sich mit einer etwas älteren, aber immer noch sehr interessanten Technik der Z80-CPU, auseinanderzusetzen, dann ist er hier genau richtig gelandet. Dieses Manual stellt die Z80-CPU mithilfe eines sehr günstigen Z80-Computers vor. Der Interessierte lernt sowohl etwas über das Betriebssystem CP/M, als auch ein wenig über die Z80-CPU mit einer grundlegenden Einführung in die Programmierung über die Maschinensprache des Z80. Die angesprochenen Themen füllen ganze Bücherschränke und es wäre illusorisch und vermessen zu behaupten, dass das auf ein paar hundert Seiten auch nur annähernd abzuhandeln wäre. Das soll es auch nicht und ist lediglich dazu da, einen geeigneten Einstieg in das Thema CP/M und Z80 zu liefern. Spaß ist auf jeden Fall die oberste Prämisse und das wird in meinen Augen auch geschehen. Wer wirklich grundlegende (Er)Kenntnisse darüber erlangen möchte, wie eine CPU funktioniert, wie Bits und Bytes hin und hergeschoben werden, der sollte hier etwas Zeit investieren. Es gibt in meinen Augen nichts Schlimmeres, als sich als angehender Entwickler nur mit Programmier-Hochsprachen zu beschäftigen und dann nicht zu wissen, was quasi unter der Haube eines Computers geschieht. Das ist heutzutage im Zeitalter „moderner“ Betriebssysteme wie *Windows*, *Mac OSX* oder *Linux*, sehr schwierig bis unmöglich, zu erfassen beziehungsweise zu realisieren, was der Computer denn so treibt, während wir gebannt und paralysiert auf den Bildschirm starren. Alle drei bis 5 Tage werden Updates eingespielt und auch nicht die vermeintlichen Spezialisten können ergründen, was denn wirklich im Hintergrund „gespielt“ wird. Eine in meinen Augen sehr verwerfliche Situation, die den Nutzern zugemutet wird. Das bedeutet nun nicht, alle modernen Computer mit dem Betriebssystem CP/M zu versehen, was sowieso nicht funktionieren würde. Doch zurück zum eigentlichen Thema. Der Z80 ist eine 8-Bit-CPU, was bedeutet, dass sein Datenbus eine Breite von 8 Bits besitzt. Mit diesen 8-Bits können eine Menge interessanter Dinge angestellt werden, wobei diese über einen 16-Bit breiten Adressbus quasi *gemanaged* werden. Auch mit Betriebssystem CP/M - DOS hat sich hier reichlich Innovationen *geholt* - können Dateien erstellt, gelöscht, kopiert oder umbenannt werden. Es gibt Programme für die Textverarbeitung, die Tabellenkalkulation und für Datenbanken. Sogar Spiele wurden entwickelt, die jedoch nur im Textmodus verfügbar sind, da CP/M keine Möglichkeit der Grafikausgabe hat. Das was nützen heutzutage die unzähligen hirnlosen Spiele mit wahnsinniger Grafik, die ohne Sinn und Verstand programmiert wurden. Da lobe ich mir doch ein kniffliges Text-Adventure, dass die kleinen grauen Zellen in Bewegung und einen in Grübeln bringt. Das alles ist jedoch nur meine persönliche Meinung und spiegelt nicht die öffentliche Meinung wider.

2 Die Geschichte des Z80



Der Z80-Prozessor wurde von der Firma *Zilog* unter der Leitung von *Federico Faggin* entwickelt und kam erstmals im Jahre 1976 auf den Markt. Er war ein direkter Konkurrent zu den bis dahin sehr beliebten und weit verbreiteten Intel-Prozessoren und bot sowohl Kompatibilität mit Intel-Systemen als auch mehr Funktionalität. Zum Starten eines CP/M-Systems ist nur ein minimales ROM erforderlich, was dann das Betriebssystem von Diskette lädt und startet, auf der die erforderlichen Dateien gespeichert sind. Im März des Jahres 1976 wurde der Z80 herausgebracht. Die Entwicklung des Z80 hatte das Ziel, dass diese CPU zum schon vorhandenen Intel 8080 abwärtskompatibel sein muss, wodurch alle auf dem 8080 entwickelten Programme auch auf dem Z80 lauffähig waren, was natürlich auch für das CP/M-System galt. Die ersten Z80-Versionen erlaubten eine Taktrate von 2,5 MHz, wobei spätere Versionen höhere Taktraten ermöglichten. Der Z80A konnte mit 4 MHz, der Z80B mit 6 MHz und der Z80H 8 MHz betrieben werden. Gerade auf dem Spielemarkt war der Z80 sehr beliebt und wurde bis Ende der 1980er Jahre in vielen Arcade-Konsolen und Heimcomputern verbaut. Nennenswerte Vertreter sind dabei der *ZX80* beziehungsweise *ZX81*, der *Tandy TRS-80* und der *PC-8801*, wobei es noch viele andere gab. Sogar für den sehr beliebten *Apple II* wurde eine Z80-Erweiterungskarte entwickelt, die ihn dazu befähigte, neben der hauseigenen 6502-CPU nun auch mit dem Z80 CP/M zu starten. Später wurde der Z80 sogar in den Taschenrechnern von Texas-Instruments verbaut und ist in den Modellen *TI-83 Plus*, *TI-84 Plus* und *TI-84 Plus Silver Edition* zu finden.

3 Der Z80-Playground



Dieses Manual - ich erwähnte es schon - soll und wird jetzt kein Handbuch für den Z80 sein, denn das Thema ist viel zu umfangreich und würde den Rahmen mit mehreren hunderten oder tausenden Seiten einfach sprengen. Zu diesem Thema gibt es sehr viele und gute Bücher, die in Form von PDFs im Internet frei verfügbar sind. Am Ende werde ich eine kleine Liste bereitstellen und auf diese Literatur verweisen. Dort sind alle Details zum Z80 sowohl für die Hard- als auch für die Software zu finden. Da das Ganze für einen Einsteiger jedoch sehr komplex scheint, ist es sehr leicht, sich zu verirren und die Lust an der sehr interessanten Thematik zu verlieren und ich rede hier aus eigener Erfahrung. Sehr viele Fachbuchautoren haben ein gigantisches Wissen, können dieses jedoch nicht in der Art didaktisch aufbereiten, dass ein Neuling einen geeigneten Einstieg findet. Meistens ist dann Frust angesagt und nach recht kurzer Zeit wird sich anderen Dingen gewidmet. Das ist in meinen Augen sehr schade, denn mit ein bisschen mehr Fingerspitzengefühl und Verständnis könnten die Menschen dort abgeholt werden, wo sie sich im Moment gerade befinden. Also quasi wissbegierig und wahrscheinlich ein wenig verloren. Nun bin ich schon seit vielen Jahren Fachbuchautor für elektronische Themen wie Arduino, Raspberry Pi oder auch ESP32, um nur einige wenige zu nennen. Für mich ist es außerordentlich wichtig und entscheidend, eine didaktische Linie zu fahren, wo ein Thema auf das andere aufbaut. Der Einsteiger wird also wie in einer geführten Meditation durch einen für ihn neuen Themenbereich geführt, wobei ich jedoch nicht hoffe, dass er dabei einschläft, wie das während einer Meditation durchaus einmal der Fall sein kann. Sollte man sich überhaupt mit der Programmierung einer veralteten CPU befassen? Macht es nicht mehr Sinn, sich mit aktuelleren IT-Themen zu befassen? Ja und Nein, in der Reihenfolge! Wer wirklich verstehen will, wie ein Computer quasi Low-Level – also auf unterster Ebene – arbeitet, sollte in Erwägung ziehen, einen anderen Weg einzuschlagen, als den, der vermeintlich Mainstream ist. Professionelle Programmiersprachen wie C/C++ oder C#, um nur wenige zu nennen, bieten zwar ebenfalls einen guten Einstieg in die Thematik der Arbeitsweise eines Computers, doch eben nicht so ganz. Natürlich ist das nur meine persönliche Meinung und kein Dogma, das nicht infrage gestellt werden darf. Wer also Lust und Zeit hat, sich mit der Programmierung einer CPU, respektive dem Aufbau eines Z80-Komplettsystems zu befassen, der sollte jetzt weiterlesen. Alle anderen können das Manual jetzt beiseitelegen und sich anderen netten Dingen widmen.

Pause!

Ok, es scheint also doch ein gewisses Interesse zu bestehen, was mich persönlich sehr freut! Ich möchte jedoch nicht weiter mit einer möglicherweise belanglosen und einschläfernden Einleitung fortfahren und nun zum eigentlichen Thema kommen. Der Z80! Wer sich mit dieser CPU beschäftigen möchte, kann das natürlich über sogenannte Emulatoren recht gut machen und einen geeigneten Einstieg finden. Es gibt umfangreiche Software, die frei verfügbar und kostenlos ist. Doch sich mit Software zu begnügen, die eine bestimmte Hardware nachbildet, ist für echte Bastler nicht so das wahre, denke ich. Nun komme ich auf den Punkt des Ganzen. Es gibt eine Entwicklungsplatine, die sich Z80-Playground nennt und von *John Squires* entwickelt wurde. Keine Sorge, ich habe mit ihm keine Vereinbarung getroffen und er hat mir - auf meinen persönlichen Wunsch hin - auch die Platine mit allen erforderlichen Bauteilen nicht frei zur Verfügung gestellt. Ich schätze die Arbeit und den Enthusiasmus

der Entwickler, die sich darum bemühen, etwas zu erschaffen, das bisher noch niemand in dieser Form gemacht hat und ich möchte deswegen nicht ausdrücklich betonen, dass ich keinen Nutzen daraus ziehen möchte und lediglich daran interessiert bin, seine Arbeit Wert zu schätzen. Das *PCB* (Printed-Circuit-Board), was er entwickelt hat, nennt sich *Z80-Playgound* und ist in meinen Augen eine wunderbare Möglichkeit, sich in die Thematik der Z80-Programmierung einzuarbeiten. Alle, für die es wichtig erscheint, sollten sich das Board einmal näher anschauen. Die Internetadresse lautet wie folgt.

http://8bitstack.co.uk/

Das Z80-Playgound-Board enthält alle Komponenten, die ein Z80-Computer benötigt. Hier die drei Hauptkomponenten, deren Funktionen gleich noch im Detail erläutert werden.

- Die **Z80** (CPU) - Zentraleinheit
- Den **ROM-Baustein 28C256** (32K EEPROM) - Nur-Lese-Speicher
- Den **RAM-Baustein UM61512** (32K RAM) - Schreib-Lese-Speicher

Zusätzlich werden noch weitere Komponenten benötigt, die eine Kommunikation mit der Außenwelt gestatten.

- **CH376S USB Pen-Drive Module** – Der CH376 wird als Dateiverwaltungs-Steuerungs-Chip verwendet, mit dem das MCU-System Dateien auf einem USB-Flash-Laufwerk oder einer SD-Karte lesen/schreiben kann. Hier sind die zahlreichen Programme und Dateien abgelegt.
- **16C550-UART** – Der 16550 UART ist eine integrierte Schaltung zur Implementierung der Schnittstelle für die serielle Kommunikation. Über diesen Weg wird die Verbindung zum Z80-Playground über das Terminal-Programm hergestellt.

3.1 Die Kommunikationswege

Auf der folgenden Abbildung sind die beiden Kommunikationswege abgebildet.

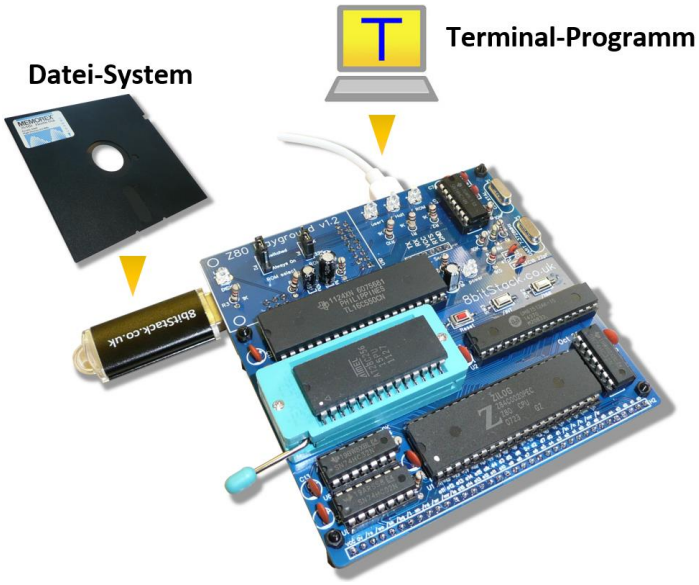


Abbildung 1 - Die Kommunikationswege

3.2 Die integrierten Schaltkreise

Nun befinden sich auf dem Z80-Playground-Board einige integrierte Schaltkreise, die sogenannten *IC*s. Die großen ICs sind auf der folgenden Abbildung markiert und beschriftet. Die drei kleinen Schaltkreise bilden die Logik zur Ansteuerung der Speicherbausteine (RAM und ROM) und des Pen-Moduls CH376.

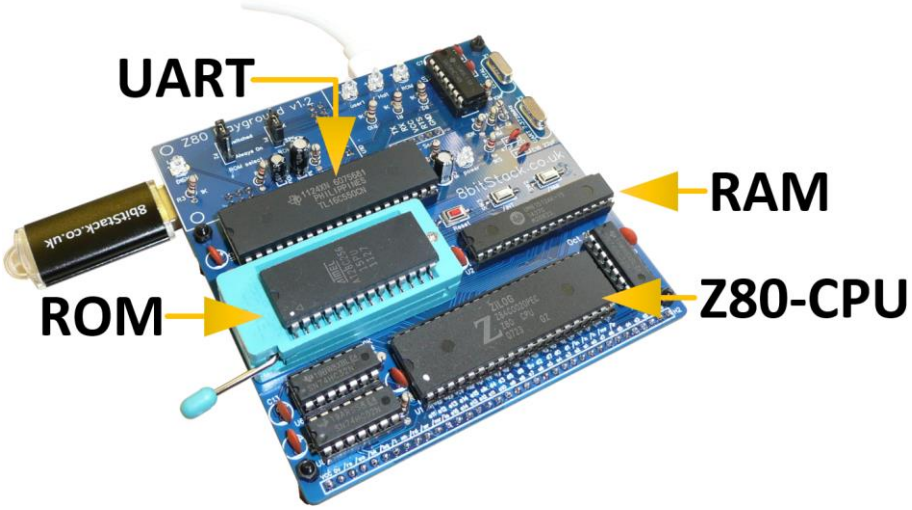


Abbildung 2 - Die grundlegenden integrierten Schaltkreise

3.3 Die Leuchtdioden

Natürlich befinden sich auf dem Z80-Playground auch einige Leuchtdioden, die sogenannten *LEDs*. Sie geben Informationen über den Status des Boards.

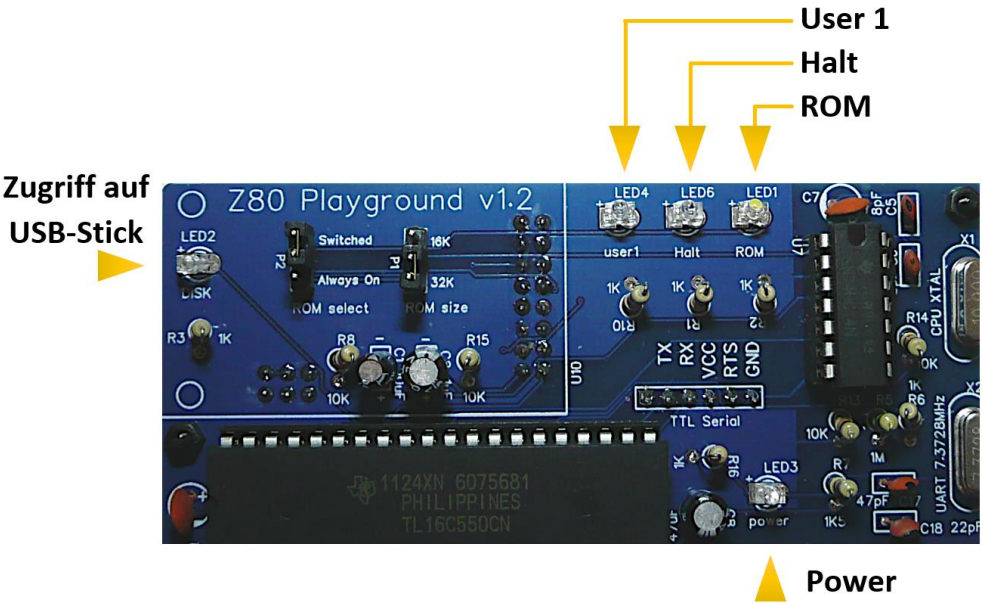


Abbildung 3 - Die LEDs

In der nachfolgenden Tabelle sind die Funktionen der LEDs erläutert.

LED	Bedeutung
LED 1	Leuchtet, wenn das ROM aktiviert ist. Kann unmittelbar nach dem Booten über die Tasten 3 (ROM ON) an- beziehungsweise 4 (ROM OFF) ausgeschaltet werden
LED 2	Leuchtet, wenn auf den USB-Stick zugegriffen wird
LED 3	Leuchtet, wenn das Board mit Spannung versorgt ist
LED 4	Leuchtet, wenn der User diese ansteuert
LED 6	Leuchtet, wenn sich die CPU im Halt-Modus befindet

3.4 Die Steckbrücken

Zur Konfiguration des Z80-Playground sind zwei Steckbrücken, die sogenannten *Jumper* vorhanden. Durch einfaches Umstecken, während das Board stromlos ist, kann eine Anpassung der Konfiguration vorgenommen werden.

Abbildung 4 - Die Jumper

Jumper	Bedeutung
ROM select	• Switched: Kann über Software gesteuert werden (Standard) • Always on: Ist immer aktiv
ROM size	• 16K: ROM nutzt 16K des Speichers • 32K: ROM nutzt 32K des Speichers (Standard)

Der RAM-Chip UM61512 besitzt eine Speichergröße von 64K. Er belegt den gesamten Z80-Speicheradressraum. Zusätzlich gibt es den ROM-Chip in Form eines EEPROMs vom Typ AT28C256, der 32K groß ist und entweder unteren 16K oder 32K des Adressraums belegt. Das Zusammenspiel zwischen diesen beiden Chips ist wie folgt.

ROM-Select

Dieser Jumper legt fest, ob das EEPROM immer eingeschaltet ist oder von der Z80 umgeschaltet werden kann. Es ist IMMER eingeschaltet, wenn das System zurückgesetzt wird. Wenn es schaltbar ist, kann die Software es jederzeit ein- oder ausschalten, indem sie Bit 3 des Modem-Steuerregisters des UART ein- oder ausschaltet (das sich an Z80-Port 12 befindet).

ROM-Size

Über diesen Jumper wird die Größe des EEPROMs eingestellt. Wenn er auf 16k gesetzt ist, belegt das EEPROM nur die unteren 16K des Adressraums, so dass 48K für das RAM übrigbleiben. Wenn er auf 32K gesetzt ist, belegt das EEPROM die untere Hälfte des Adressraums und das RAM die obere Hälfte.

Regel für das Lesen und Schreiben

Es gibt jedoch unterschiedliche Regeln für das Lesen und Schreiben aus beziehungsweise in den Speicher. Ein *WRITE* schreibt immer in das RAM, unabhängig davon, wie das ROM konfiguriert ist. Ein *READ* liest entweder aus dem ROM oder aus dem RAM, abhängig von der Konfiguration. Wenn das ROM umschaltbar und aktiviert ist, dann wird zum Beispiel beim Lesen von Speicherplatz 27 aus dem ROM gelesen, aber beim Schreiben auf Speicherplatz 27 in das RAM geschrieben. Das bedeutet, dass Sie das gesamte ROM in das RAM kopieren können, indem Sie das ROM in einer Schleife durchlaufen, jedes Byte lesen und es an dieselbe Stelle zurückschreiben. Sie können dann das ROM ausschalten und das Programm trotzdem aus dem RAM ausführen.

3.5 Die Mikro-Taster

Um von außen auf das Verhalten des Z80-Playground Einfluss zu nehmen, sind drei Mikro-Taster (Switches) vorhanden.

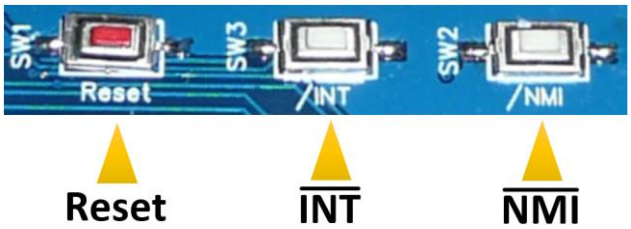


Abbildung 5 - Die Mikro-Taster

Taster	Bedeutung
SW1: Reset	Führt einen Reset des Boards durch (Kaltstart)
SW3: INT	Führt einen Interrupt durch
SW2: NMI	Führt einen Non Mascable Interrupt durch

3.6 Der Schaltplan

Der Schaltplan des Z80-Playground v1.2 ist unter der folgenden Internetadresse zu finden.

http://8bitstack.co.uk/wp-content/uploads/2021/01/Schematic_Z80-playground_v_1_2.pdf

Dort sind alle Komponenten zu finden und wie sie untereinander verbunden sind.

4 Das Bus-System

Wenn es darum geht, dass eine CPU, die ja das Herz eines Computersystems darstellt, funktionieren soll, dann kann sie das nicht alleine bewerkstelligen. Genau wie im menschlichen Körper gibt es ja auch nicht nur das Gehirn als zentrale Schaltzentrale, sondern weitere Organe, wie zum Beispiel das Herz, die Lunge, die Leber und viele andere. Alle zusammen bilden – recht materiell gesehen – den menschlichen Körper und stehen in einer bestimmten Verbindung untereinander. Es werden also Informationen ausgetauscht, die notwendig sind, dass das System Mensch lebensfähig ist und bleibt. Das Ganze hört sich sehr technisch an und soll lediglich als Beispiel diesen, auch, wenn es an vielen Stellen etwas hinkt. Im Körper eines Menschen gibt es natürlich viele Wege, um Informationen zwischen den einzelnen Organen auszutauschen. Da ist zum einen der Blutstrom, der durch viele Adern fließt und nicht nur Sauerstoff zu den „*Verbrauchern*“ leitet. Über Hormone beziehungsweise Botenstoffe können auf diesem Wege ebenfalls Informationen verbreitet werden, die von den entsprechenden Organen, die sie benötigen, in Empfang genommen und verarbeitet werden. Zum anderen gibt es auch das Nervensystem, das über elektrische Impulse wichtige Daten quasi in Echtzeit weiterleitet. Auch hier sind Sender und Empfänger in geeigneter Form miteinander verbunden und stehen in regem Austausch wichtiger Signale zur Aufrechterhaltung des organischen Systems. Warum erzähle ich das alles? Nun, in einem Computersystem sieht es – natürlich sehr vereinfacht – ähnlich aus. Es gibt Baugruppen, die Informationen versenden und andere, die diese empfangen. Nicht jede Information ist für jeden bestimmt und deswegen gibt es eine Logik, die bestimmt, wer was auszuwerten hat. Die einzelnen Baugruppen eines Computers kommunizieren über sogenannte *Bussysteme* miteinander. Busse bestehen in der Regel aus einzelnen elektrischen Leitungen, die zu funktionalen Gruppen zusammengefasst werden, wobei sich im Grunde genommen drei Einzelbusse unterscheiden.

- **Datenbus**: Auf diesen Leitungen werden die Daten (Informationen) zwischen den Baugruppen CPU, RAM und ROM untereinander ausgetauscht
- **Adressbus**: Die CPU legt fest, welche Baugruppe Sender beziehungsweise Empfänger der Daten ist. Jede einzelne Baugruppe besitzt eine oder mehrere Adressen, wobei durch die Angabe der Adressen auf dem Adressbus die gewünschte Baugruppe angesprochen wird. Die Anzahl der möglichen Adressen richtet sich nach der Anzahl der zur Verfügung stehenden Adressleitungen. Eine CPU mit 16 Adressleitungen kann zum Beispiel 2^{16} = 65536 Adressen (0x0000 bis 0xFFFF) adressieren.
- **Steuerbus**: Eine CPU besitzt eine Vielzahl an Steuerleitungen, die die Zugriffsart auf die einzelnen Baugruppen bestimmen. Um den Zugriff auf einen externen Speicher zu ermöglichen, wird zwischen zwei Leitungen unterschieden.
  - **RD**: Read (aus Speicher lesen)
  - **WR**: Write (in Speicher schreiben)

Auf der folgenden Abbildung ist die Kommunikation zwischen den Baugruppen CPU, RAM, ROM und PIO zu sehen.

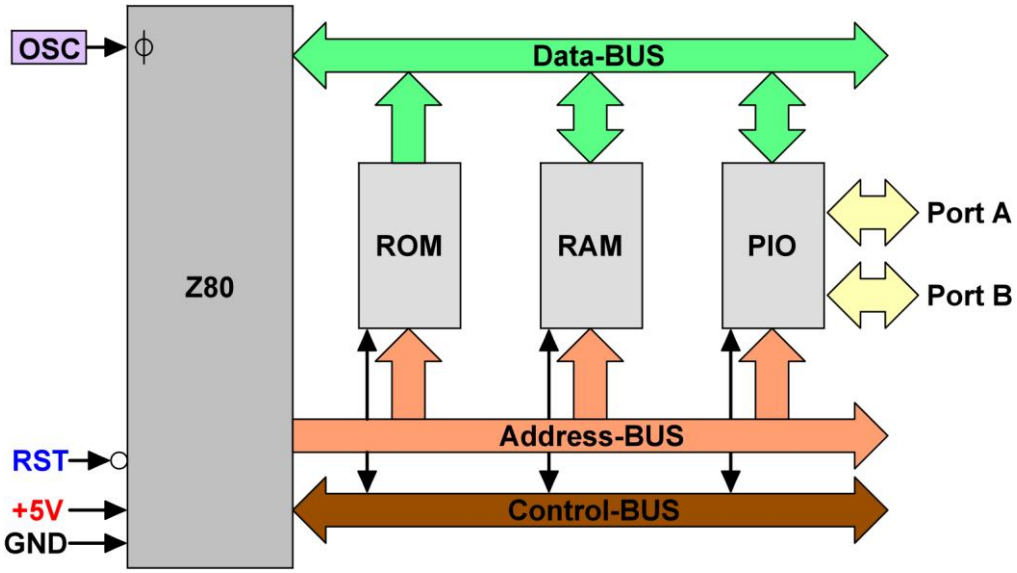


Abbildung 6 - Das grundlegende Bus-System des Z80

Der Adress- und Datenbus muss in einem Computersystem eine Vielzahl von Aufgaben erledigen, wobei alle Aufgaben etwas mit der Übermittlung von Daten beziehungsweise Informationen zu tun haben. Es müssen zum Beispiel die Informationen von einem Eingabegerät (z.B. Tastatur) in den Arbeitsspeicher, das RAM, gelangen, dann vom Z80 verarbeitet und wieder zurück in den Arbeitsspeicher und am Ende in das Ausgabegerät (z.B. Monitor) versendet werden. Dieser Datentransport im Zusammenspiel zwischen Adress- und Datenbus wird oftmals auch als *Datenautobahn* im Computer bezeichnet. Auf die genauen Unterschiede zwischen ROM und RAM und was sie bedeuteten, gehe ich gleich genauer ein.

5 Der Speicherbereich

Ein adressierbarer Speicherbereich ist naturgemäß abhängig, von den zur Verfügung stehenden Adressleitungen. Je mehr es davon gibt, desto mehr unterschiedliche Adressen können damit erreicht – adressiert – werden. Ein Z80 besitzt 16 Adressleitungen. Der mögliche Adressbereich berechnet sich wie folgt, wobei der Adressraum den adressierbaren Speicherbereich bezeichnet.

$$\text{Anzahl der möglichen Adressen} = 2^{Anzahl\ der\ Adressleitungen}$$

Konkret schaut das dann wie folgt aus.

$$\text{Anzahl der möglichen Adressen} = 2^{16} = 65.536$$

Da im Umfeld der Programmierung beziehungsweise der Adressierung nicht mit Dezimalzahlen, sondern mit hexadezimalen Zahlen gearbeitet wird, ist das ein Adressierungsbereich, der sich von *0x0000* bis *0xFFFF* erstreckt. Das Präfix 0x deutet darauf hin, dass es sich um eine hexadezimale Zahl handelt. Teilweise werden auch Angaben gemacht, die als Postfix ein *H* anhängen, wobei der Bereich dann von *0000H* bis *FFFFH* deklariert wird. Der Speicher eines Computersystems ist der Ort, an dem Daten gespeichert und/oder abgerufen werden können. Es gibt unterschiedliche Arten von Speicher, aber der Einfachheit halber erwähne ich lediglich das *ROM* und das *RAM*. Was aber bedeuten die Abkürzungen ROM beziehungsweise RAM überhaupt?

- ***ROM***: Dies ist die Abkürzung für *Read-Only-Memory* und bedeutet übersetzt *Nur-Lese-Speicher*. In diesem Speicher sind Daten abgelegt, die vom System nicht mehr verändert werden können und es ist der Speicherbereich, in dem zum Beispiel das Betriebssystem abgelegt ist. Die Daten bleiben nach der Unterbrechung von der Stromversorgung weiterhin bestehen.
- ***RAM***: Dies ist die Abkürzung für *Random-Access-Memory* und bedeutet übersetzt Speicher mit wahlfreiem Direktzugriff. Es handelt sich um eine Speicherform, der bei Computern als sogenannter Arbeitsspeicher Verwendung findet. Die Daten sind nach der Unterbrechung von der Stromversorgung verloren, da es sich um einen sogenannten flüchtigen Speicher handelt.

Nun ist es sicherlich sinnvoll, diese beiden Speicherformen der Z80-CPU irgendwie zugänglich zu machen. Man hat sich für das folgende Schema entschieden, obwohl das natürlich auch in einigen Spezialfällen geändert werden kann.

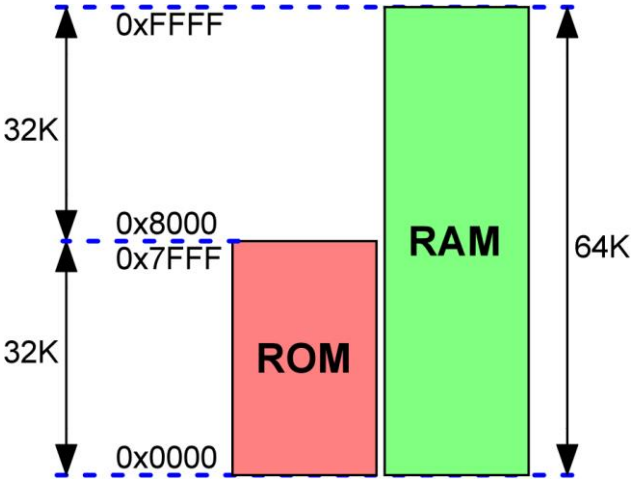


Abbildung 7 - Die Speicheraufteilung von ROM und RAM mit der Überlappung der Bereiche

Im unteren Bereich von *0x0000* bis *0x7FFF* ist das ROM mit seinen 32K angesiedelt. Parallel dazu befindet sich das RAM und deckt den Bereich von *0x0000* bis *0xFFFF* ab. Dieser gesamte Speicherbereich beträgt also in Summe 64KByte. Es sei hier nochmals erwähnt, dass das ROM in seiner Größe zwischen 16K und 32K dimensioniert werden kann. Auf der folgenden Abbildung sind sowohl die 32K-, als auch 16K-Konfiguration zu sehen. Standardmäßig sind bei der Auslieferung 32K für das ROM ausgewählt.

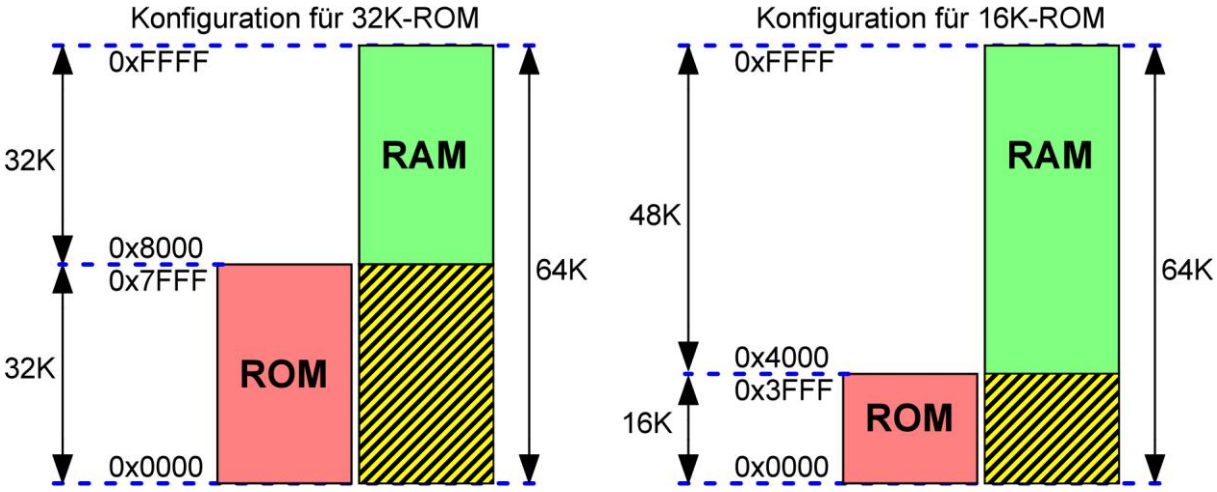


Abbildung 8 - Die Speicheraufteilung bei unterschiedlichen Konfigurationen

Die vorhandene Konfiguration kann über den Menüpunkt *m* (Memory Map) nach dem Booten des Z80-Playground abgerufen werden.

Abbildung 9 - Der Aufruf der Memory Map

Im Anschluss zeigt sich, je nach Konfiguration, die *Memory Map* mit der vorherrschenden Aufteilung von ROM und RAM.

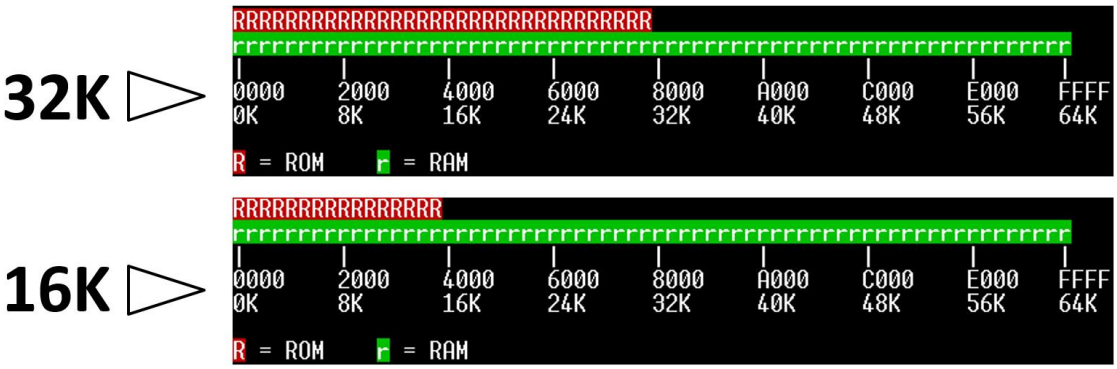


Abbildung 10 - Die Speicheraufteilung bei unterschiedlichen Konfigurationen

Um hier wieder eine Analogie zur Realität zu verwenden, kann gesagt werden, dass die Speicheradressierung ist wie die Zustellung einer Post ist. Jedes einzelne Haus besitzt in der jeweiligen Straße eine eindeutige Hausnummer und ist vergleichbar mit einer Speicheradresse im Z80-Computersystem, die der Postbote zum Beispiel bei der Zustellung eines Briefes nutzt. Beim Starten des Systems ist der RAM-Bereich, der ja den flüchtigen Speicher darstellt leer und aus diesem Grund ist es sinnvoll, der CPU über das ROM mitzuteilen, was sie denn überhaupt machen soll. Die Adresse 0x0000 wird also immer die erste Stelle sein, an der der Z80 beim Systemstart sucht, was zu tun ist. An dieser Stelle muss also eine Anweisung stehen, die ausgeführt werden kann. Wie ist das aber genau zu verstehen, denn eine einzelne Adresse auf dem Adressbus ist ja quasi nur eine Hausnummer. Was verbirgt eigentlich sich dahinter? Jetzt kommt der Datenbus ins Spiel! Bei der Wahl einer bestimmten Adresse - sei es ROM oder RAM – werden die dort gespeicherten Informationen in Form von Daten auf den Datenbus gelegt. Und da der Z80 ebenfalls an den Datenbus angeschlossen ist, kann der diese unmittelbar lesen. Der Datenbus besitzt eine Datenbreite von 8 Bits, was einem Byte entspricht. Über die eben genannte Formel können auch in diesem Fall die unterschiedlichen Bitmöglichkeiten berechnet werden.

$$\text{Anzahl der möglichen Daten} = 2^{Anzahl\ der\ Datenleitungen}$$

Konkret schaut das dann wie folgt aus.

Anzahl der möglichen Daten = $2^8 = 256$

Über die abgerufenen Daten des Datenbusses über die angeforderte Adresse kann der Z80 nun ermitteln, was er als nächstes zu tun hat. Wenn es darum geht, Daten zu speichern, so ist das zwar über interne Speicherbereiche – den sogenannten Registern – innerhalb des Z80 möglich, doch diese sind in ihrer Anzahl sehr gering. Aus diesem Grund ist ja das RAM vorhanden, dass die Speicherung von Informationen zulässt, was bei einem ROM nicht möglich ist, da es ja quasi schreibgeschützt ist. Das RAM wird beim Z80 durch einen speziellen Speicherbaustein gebildet, der sich *SRAM* nennt. SRAM steht für Static-RAM und benötigt im Gegensatz zum *DRAM* (Dynamic-RAM) keinen Refresh-Zyklus, um den Inhalt zu bewahren. Der Zugriff auf den RAM-Bereich ist in diesem Beispiel im Speicherbereich von 0x8000 bis 0xFFFF angesiedelt und kann nach Lust und Laune beschrieben und wieder gelöscht werden. Abschließend zu diesem Thema kann gesagt werden, dass aus diesem Grund ist die erste Anweisung für den Z80 entscheidend ist, ob und wie das ganze System funktioniert. Dann wollen wir im nächsten Schritt doch einmal sehen, wie das Ganze in logischer Abfolge – wenn auch etwas vereinfacht dargestellt - so funktioniert. Die Kombination zwischen Adress-, Daten- und Steuerbus wird auch als *Systembus* bezeichnet.

- ***Schritt 1:*** Der Z80 legt eine Adresse auf den Adressbus
- ***Schritt 2:*** Der Z80 gibt einen Lesebefehl auf den Steuerbus
- ***Schritt 3:*** Die angesprochene Baugruppe (ROM, RAM) übermittelt ihre Daten an den Datenbus
- ***Schritt 4:*** Der Z80 liest die Daten vom Datenbus und verarbeitet sie

Wenn man sich die Speicherbausteine von früher und von heute anschaut, dann sind da schon einige Unterschiede zu bemerken. Ich fange einmal mit den ROM-Bausteinen an. Früher waren das sogenannte EPROMs, die als Festspeicher genutzt wurden. *EPROM* ist die Abkürzung für *Erasable Programmable Read-Only Memory* und frei übersetzt löschbarer programmierbarer Nur-Lese-Speicher bedeutet. Es handelt sich um einen nichtflüchtigen elektronischen Speicherbaustein, der bis zur Mitte der 1990er-Jahre in der Computertechnik verwendet wurde, inzwischen jedoch durch sogenannte EEPROMs ersetzt wurden. *EEPROM* ist die Abkürzung für *Electrically Erasable Programmable Read-Only Memory* und bedeutet, dass es sich um einen elektrisch löschbaren und programmierbaren Nur-Lese-Speicher handelt. Ganz schön lange Beschreibung, nicht wahr?! Auf der folgenden Abbildung sind beide Bausteine zu sehen.

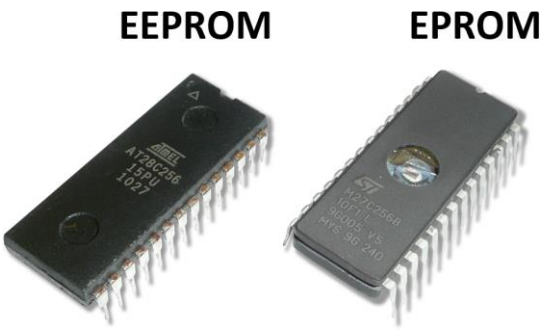


Abbildung 11 - Unterschiedliche ROM-Bausteine

Bei einem EPROM ist zu sehen, dass sich dort ein kleines Fenster befindet, was dafür da ist, bei entsprechender UV-Bestrahlung, den Inhalt des Speichers zu löschen. Derartiges ist heutzutage nicht

mehr erforderlich, obwohl das Ganze natürlich einen besonderen Charme aufweist und deswegen immer noch eine gewisse Herausforderung darstellt, der man sich schlecht entziehen kann.

6 Das Pinout – Die Pinbelegung

Die Z80-CPU mit ihren 40 Pins hat natürlich sehr viele unterschiedliche Pin-Funktionen, die im nachfolgenden Pinout zu sehen sind.

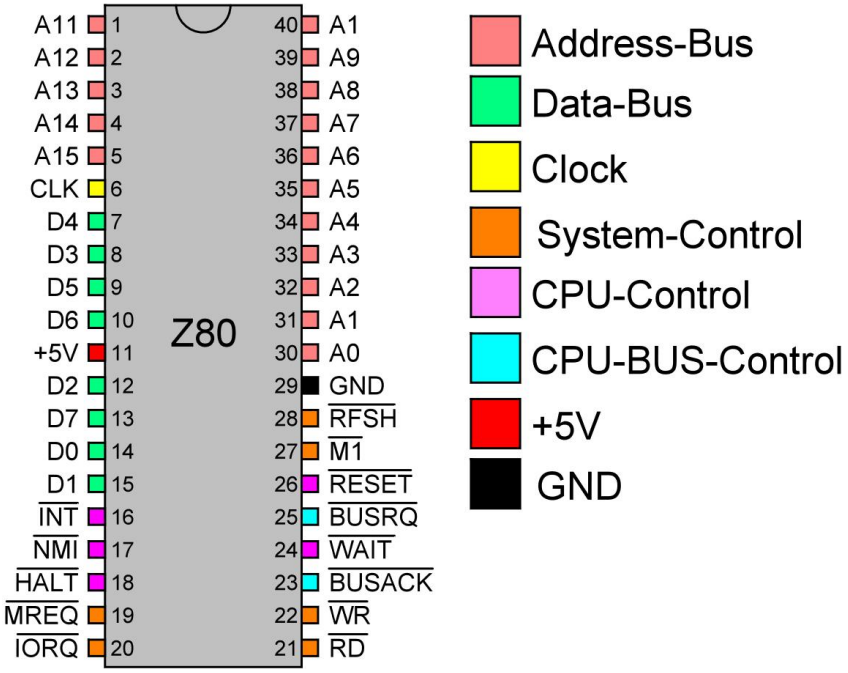


Abbildung 12 - Die Z80-Pinbelegung

In der nachfolgenden Tabelle sind die farblichen Kodierungen der einzelnen Pins erläutert.

Farbe	Bedeutung
	Die hellroten Pins sind der ***Adressbus***. A0 bis A15 werden verwendet, um eine Adresse im Speicher während eines Speicher-Lese- oder Schreibvorgangs anzugeben. A0 bis A7 werden auch verwendet, um während eines Port-Lese- oder Schreibvorgangs ein E/A-Gerät auszuwählen. Der Adressbus arbeitet nur unidirektional.
	Die grünen Pins sind der ***Datenbus***. D0 bis D7 werden zum Übertragen oder Empfangen von Daten während einer Speicher- oder Port-Lese- oder Schreiboperation verwendet. Der Datenbus kann auch verwendet werden, um anzuzeigen, welches Gerät einen Interrupt ausgelöst hat. Der Datenbus arbeitet bidirektional.
	Der Pin für den ***Takt*** (Clock)

	Der rote Pin ist die ***Spannungsversorgung*** für +5V
	Der schwarze Pin ist die ***Spannungsversorgung*** für Masse (GND)
	Die orangenen Pins sind für die ***System-Controls*** zuständig: • $\overline{\text{M1}}$ zeigt an, dass der Z80 den nächsten Befehl aus dem Speicher holt • $\overline{\text{MREQ}}$ zeigt an, dass der Z80 auf den Speicher zugreifen möchte • $\overline{\text{IORQ}}$ zeigt an, dass der Z80 auf einen I/O-Port zugreifen möchte • $\overline{\text{RD}}$ geht während eines Speicher- oder I/O-Lesevorgangs auf einen LOW-Pegel • $\overline{\text{WR}}$ geht während eines Speicher- oder I/O-Schreibvorgangs auf einen LOW-Pegel • $\overline{\text{RFSH}}$ wird dazu genutzt, um einen Refresh-Impuls an die Speichermodule zu senden. Die frühen Speicherschaltungen konnten ihren Inhalt nicht auf unbestimmte Zeit speichern und mussten einen Refresh-Impuls von Zeit zu Zeit erhalten. Dies erforderte normalerweise eine zusätzliche Schaltung, die sicherstellte, welche Speicheradressen aufgefrischt werden mussten
	Die lila Pins sind für die ***CPU-Controls*** zuständig: • $\overline{\text{HALT}}$ zeigt an, dass sich der Z80 in einem angehaltenen Zustand befindet, d. h. er wartet auf einen Interrupt • $\overline{\text{WAIT}}$ kann von Speicher- oder E/A-Geräten verwendet werden, um die Z80 während eines Lese- oder Schreibvorgangs warten zu lassen, während sich das Gerät darauf vorbereitet, eine Anforderung zu erfüllen • $\overline{\text{INT}}$ zeigt an, dass ein Hardware-Interrupt aufgetreten ist und veranlasst den Z80, auf diesen zu reagieren • $\overline{\text{NMI}}$ ist ein nicht maskierbarer Interrupt und hat eine höhere Priorität als der INT • $\overline{\text{RESET}}$ wird verwendet, um die Z80 in einen wohldefinierten Zustand zurückzusetzen
	Die cyan Pins sind für den ***Bus-CPU-Control*** zuständig: • $\overline{\text{BUSRQ}}$ wird von externen Geräten verwendet, um die Kontrolle über die Daten-, Adress- und Systemsteuerungsbusse anzufordern. Wenn die Z80 bereit ist, die Kontrolle zu übergeben, wird dies dem anfordernden Gerät über den BUSACK-Pin signalisiert • $\overline{\text{BUSACK}}$ zeigt an, dass die Z80 einem anderen Gerät die Kontrolle über die Busse überlassen hat

Auf der folgenden Abbildung sind die einzelnen Pins zu logischen Gruppen zusammengefasst.

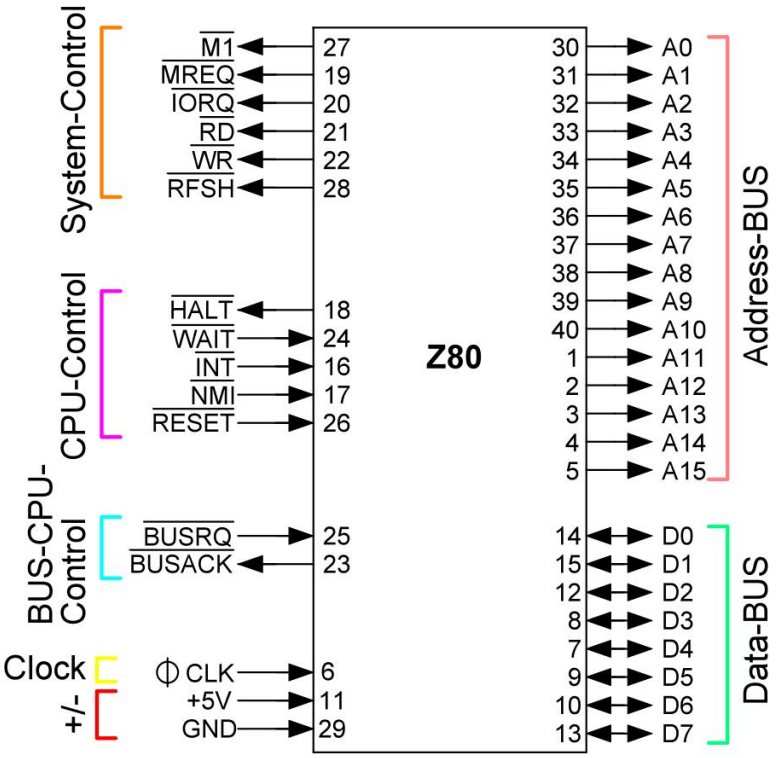


Abbildung 13 - Die Pinbelegung in logischen Gruppen

Im nächsten Schritt geht es darum, wie die Z80-CPU, die alleine für sich nicht wirklich einen Sinn ergibt, mit der Peripherie in Verbindung tritt. Damit die CPU also weiß, was sie beim nächsten Programmschritt ausführen soll, kommen sowohl Adress- als auch Datenbus ins Spiel. Eine an die Speicherbausteine gelegte Adresse liefert eine Antwort an den Datenbus. Diese Antwort veranlasst die CPU entsprechend des gelesenen Byte-Wertes - es können natürlich auch mehrere in Folge sein - etwas zu tun. Die Verschaltung von Z80, ROM und RAM schaut sehr vereinfacht wie folgt aus. Es ist zu erkennen, dass alle Adress- und alle Datenleitungen parallel miteinander verbunden sind. Welcher Speicherbaustein jedoch wann angesprochen werden soll, erfolgt über eine bestimmte Logik.

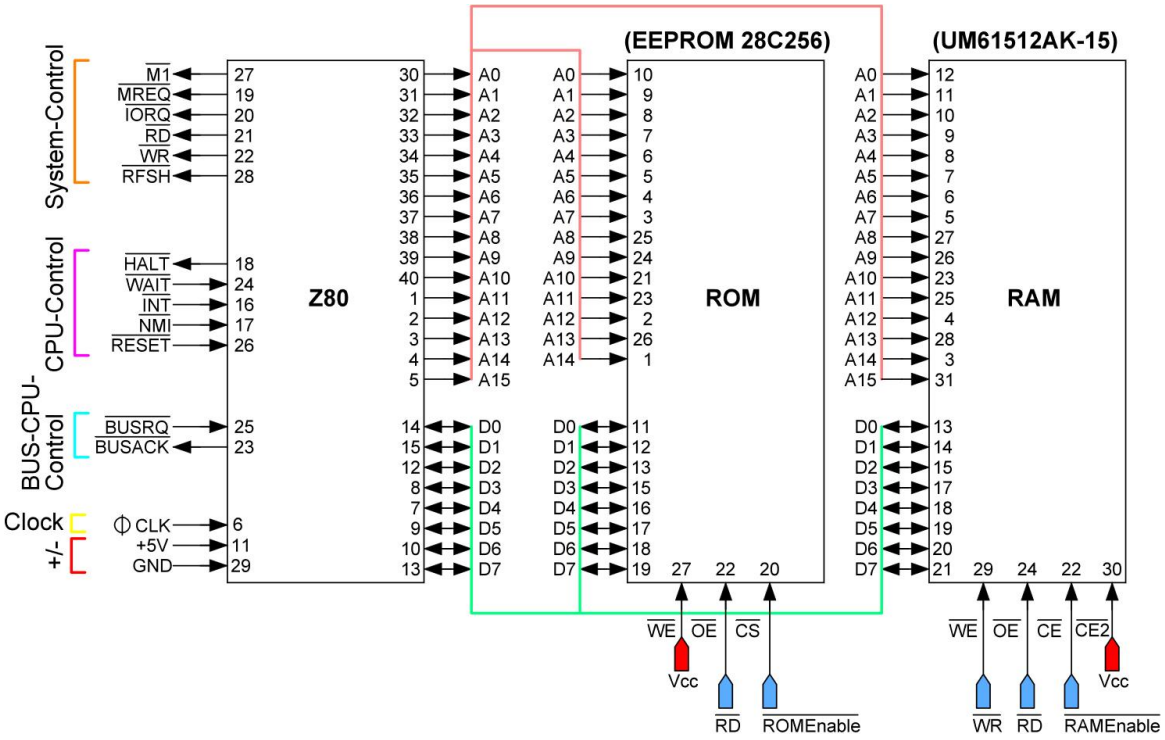


Abbildung 14 - Z80 + ROM + RAM

Für Detailinformationen zur Ansteuerung der Speicherbausteine, empfehle ich einen Blick auf den offiziellen Schaltplan zu werfen, der unter der folgenden Internetadresse zu finden ist.

http://8bitstack.co.uk/wp-content/uploads/2021/01/Schematic_Z80-playground_v_1_2.pdf

7 Das Betriebssystem CP/M

Um mit einem Computer in Kommunikation zu treten, muss nach Möglichkeit eine Software installiert sein, die es dem Nutzer gestattet, mehr oder weniger komfortabel eine Interaktion zwischen Mensch und Maschine zu führen. Man tippt also zum Beispiel Befehle in eine sogenannte Kommandozeile ein und versendet damit Anfragen an den Rechner, der entsprechende Aktionen ausführt und Antworten präsentiert. Natürlich geht das im Zeitalter der modernen Betriebssysteme wie *Windows*, *Mac OSX* oder Linux zumeist über eine grafische Benutzeroberfläche, die über das Eingabegerät *Maus* sehr einfach Programme startet um dann zum Beispiel über die Tastatur Texte in schicken Fenstern zu erstellen beziehungsweise zu verwalten, Grafiken zu kreieren oder auch Musik zu komponieren und abzuspielen. Das sind nur sehr wenige Beispiel einer Vielzahl von Einsatzmöglichkeiten eines modernen Computers. Die Geschichte der Betriebssysteme hat jedoch etwas anders angefangen. In den 1970er Jahren gab es einen amerikanischen Bastler und Computer-Freak mit Namen *Gary Kildall*. Man sagt, dass er ein Diskettenlaufwerk bekommen hatte und nun bestrebt war, dieses an seinen selbst gebauten Computer anzuschließen. Sicherlich kein einfaches und alltägliches Unterfangen dieser Tage. Es war jedoch so intelligent, dass er sich kurzerhand selbst eine passende Schaltung – einen Controller – bastelte, um das Laufwerk nutzen zu können. Doch derartige Hardware ist ohne eine geeignete Software eigentlich nicht viel Wert und unbrauchbar. Also programmierte er noch eine – man würde heute sagen – Treibersoftware, die in der Lage war, Informationen beziehungsweise Daten auf eine Diskette zu schreiben und auch wieder zu lesen. Das war quasi die Geburtsstunde für *CP/M*, was für ***C**ontrol-**P**rogram for **M**ikrocomputers* steht. Kildall gründete daraufhin eine Firma mit dem Namen *Digital Research*.

Bei CP/M handelte sich also um das erste *DOS* (Disk Operating System) und war somit der Vorläufer für die heute geläufigen Microsoft- DOS/Windows-Systeme, wobei MS-DOS selber von einem CP/M-Clone namens *QDOS* (Quick and Dirty Operating System) abstammt, das von *Tim Patterson* programmiert und von Microsoft für den ersten IBM PC eingekauft wurde. Microsoft hat sich für das Betriebssystem DOS an CP/M orientiert und diese Grundlagen gnadenlos ausgenutzt, um ein eigenes Produkt auf den Markt zu bringen. Wurde es „*geklaut*“? Das muss jeder für sich selbst entscheiden und recherchieren. Wer die Wahrheit sucht, für den sprechen die Fakten für sich! Es ist nichts so, wie es scheint! Und schon sind wir beim eigentlichen Thema dieses Kapitels angelangt. CP/M wurde sofort zum Standard-Betriebssystem für Prozessoren wie den Z80, 8080 und 8085 erkoren, die sich alle sehr ähnlich sind. Dadurch entstand eine sehr große Anzahl von Programmen, die auf Rechnern liefen, die diese CPUs besaßen. Zahllose Programme wie zum Beispiel Assembler, diverse Programmiersprachen wie BASIC, PASCAL, FORTRAN wurden für CP/M entwickelt oder portiert und CP/M erfreute sich dadurch großer Beliebtheit. Auch heute noch – 50 Jahre später – hat dieses Betriebssystem nichts von seinem Charme verloren und viele Entwickler basteln Platinen, die einen Z80 als CPU besitzen, um Software aus vergangenen Zeiten wieder aufleben zu lassen.

Da CP/M nun die Basis für den Betrieb des Z80-Playground darstellt, werden wir uns einigen grundlegenden Strukturen beziehungsweise Befehlen widmen, damit der Einstieg im Umgang mit dem Z80-Playground auch gelingt und der Spaßfaktor nicht zu kurz kommt. Natürlich wird das hier keine

umfassende und komplette Einführung in CP/M werden, doch es reicht dafür aus, erste Erfahrungen zu sammeln. Eine frei verfügbare Buchliste – ich erwähnte es schon – findet sich am Endes des Manuals.

Woraus setzt sich denn CP/M eigentlich zusammen? Nun, bei CP/M handelt es sich sowohl um ein spezielles Programm, als auch um eine Sammlung von Programmen, die den Umgang mit CP/M ermöglicht beziehungsweise erleichtert. Im Grunde genommen wird zwischen zwei Softwaregruppen unterschieden. Da ist zum einen die Systemsoftware, die über das CP/M mit internen Kommandos abgebildet wird und zum anderen die schon erwähnte Sammlung diverser (Dienst)-Programme, auch Tools genannt. Prominente Vertreter der Tools sind Programme wie *ED* oder *PIP*, auf die ich natürlich noch im Detail eingehe. Der Unterbau von CP/M ist in drei wesentliche Komponenten gegliedert.

- **BIOS**: *Basic Input Output System* - Basisfunktionen der Ein-/Ausgabe
- **BDOS**: *Basic Disk Operating System* - Grundlegende Diskettenverwaltung
- **CCP**: *Console Command Processor* - Eingabeaufforderung mit den wichtigsten CP/M-Kommandos

Die für den Z80-Playground verwendete Version ist CP/M 2.2.

Für den Nutzer des Z80-Playground ist diese Dreiteilung der genannten Komponenten quasi unsichtbar. Beim Systemstart von CP/M werden die drei Komponenten komplett in den Arbeitsspeicher geladen. Der verbleibende Speicher wird *TPA* - Transient Program Area - genannt und steht für nachzuladende Programme bereit, die als Programmdateien mit der Endung *COM* gespeichert werden. Das sind essentielle Dateien wie zum Beispiel, worauf ich noch detailliert eingehen werde.

- PIP.COM
- STAT.COM
- DDT.COM
- DUMP.COM
- usw.

7.1 Das Starten von CP/M

Wird der Z80-Playground mit der Versorgungsspannung verbunden und ist ein Terminal-Programm wie zum Beispiel *Terra Term* angeschlossen, dann ist nach kurzer Zeit ein unscheinbares Promptzeichen zu sehen, das die Bereitschaft zum Empfang eines Kommandos anzeigt. Die Übertragungsrate muss bei *Tera Term* dabei auf *460800* Baud gesetzt werden.

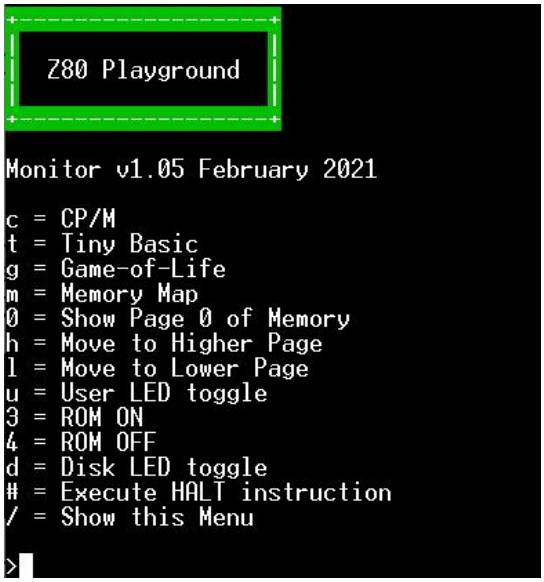


Abbildung 15 - Z80-Playground-Start

Um das Betriebssystem CP/M zu starten, muss die Taste *c* (für CP/M) gedrückt werden. Im Anschluss schaut die Anzeige im Terminalprogramm wie folgt aus. Es erscheinen zu Beginn ein paar grundlegende Informationen über die Größe des Systems - hier 64K - mit den möglichen Laufwerken von A bis P. Zudem werden Startadressen der Bereiche *CORE*, *BIOS*, *BDOS* uns *CCP* ausgegeben.

```
CP/M v2.2
Z80 Playground - 8bitStack.co.uk
Rel 1.08
Inspired by Digital Research

64K system with drives A thru P

CORE F600
BIOS F400
BDOS EA00
Z80CCP.BIN DE00

A0>
```

Das Prompt *A0* zeigt an, dass wir uns auf dem Laufwerk *A* befinden und die nachfolgende *0* zeigt den betreffenden User an. Da es bei CP/M noch keine Unterverzeichnisse gab, wurde dieses Manko über unterschiedliche User (0, 1, 2, usw.) realisiert. Später mehr dazu. Ich sprach darüber, dass in CP/M als Systemprogramm interne Kommandos verfügbar sind, die in der sogenannten *CCP* (Console Command Processor) implementiert sind. Es handelt sich dabei um den Kommandozeileninterpreter, der vergleichbar mit der Kommandozeile unter *DOS* ist. Die folgenden Kommandos sind intern, also fest eingebaut und beziehen sich auf Diskettenoperationen. Diese Befehle sind also nicht im Dateisystem zu finden, sondern sind im Betriebssystem integriert.

Kommando	Funktion
DIR	Anzeige des aktiven Disketteninhaltes
ERA	Löschen einer Datei
REN	Umbenennen einer Datei
SAVE	Speichern einer Datei
TYPE	Anzeigen einer ASCII-Datei auf der Konsole
USER	Ändern der Anwendernummer

Das zu sehende Promptzeichen **A0>** zeigt sowohl den Laufwerksnamen *A*, als auch den User *0* an, wobei CP/M 16 verschiedene Laufwerke von *A:* bis *P:* und User von *0* bis *15* unterstützt. Nach dem Booten ist immer User 0 aktiv. Gehen wir also die essentiell wichtigen internen Befehle einmal durch. Die übrigen Befehle liegen als Programme auf der CP/M-Diskette vor und werden bei Bedarf und auf Anforderung nachgeladen und werden *Transiente Befehle* genannt. Für den Anwender ist es eigentlich nicht wichtig, zu welcher Kategorie ein Befehl oder ein Programm gehört.

7.2 Das DIR-Kommando

Über das *DIR*-Kommando wird eine Liste der auf dem Standardlaufwerk vorhandenen Dateien erzeugt, die vierspaltig ist. Wird kein weiterer Parameter angegeben, enthält die Liste alle vorhandenen Dateien. Um eine Filterung nach bestimmten Mustern vorzunehmen, können die Zeichen * beziehungsweise ? als Platzhalter verwendet werden. Ein ? im Suchmuster ist stellvertretend für ein beliebiges Zeichen, ein * für beliebig viele beliebige Zeichen. Es ist dabei zu beachten, dass ein CP/M-Dateiname immer aus 8 Zeichen für den eigentlichen Namen und drei Zeichen für das Suffix besteht. Wird diese Konvention nicht erfüllt, werden Name und Suffix intern mit Leerzeichen aufgefüllt. Eine Datei mit dem Namen

xyz.txt

wird intern gespeichert als

xyz·····.txt

wobei die Punkte für Leerzeichen stehen. CP/M kennt keine Verzeichnisse, wie das von DOS vielleicht geläufig ist. Alle Dateien eines Verzeichnisses liegen demnach auf einer Ebene. Damit entfällt auch die Angabe von Pfadnamen. Soll eine Datei auf einem anderen als dem aktuellen Laufwerk angesprochen werden, so wird der Laufwerksbuchstabe gefolgt von einem Doppelpunkt vor den Dateinamen gestellt, also beispielsweise

B:dump.com

Ein Stern (*) steht also für eine komplette Zeichenfolge und das Fragezeichen (?) für ein einzelnes Zeichen. Hierzu einige Beispiele. Angenommen, es sind die folgenden Dateien auf Laufwerk ***A*:** vorhanden, was über das *DIR*-Kommando ohne weitere Parameter zur Anzeige gebracht wird.

```
A0>dir
CHK16550.COM : DELBR   .COM : PIP     .COM : SARGON  .COM
SD      .COM : STAT    .COM : TE      .COM : TECF    .COM
UNARC   .COM : UNCR    .COM : UNZIP   .COM : ZDE     .COM
ZDENST  .COM : DDT     .COM
A0>
```

Möchte ich jetzt zum Beispiel alle Dateien anzeigen, die mit dem Buchstaben *T* beginnen, ist das folgende Kommando mit den angezeigten Parametern abzusetzen.

```
A0>dir t*.*
TE      .COM : TECF    .COM
A0>
```

Ist es notwendig, dass alle Dateien, die mit einem *D* beginnen, *3* Zeichen lang sind und mit *COM* enden, dass ist das folgende Kommando abzusetzen. Zu Beginn habe ich noch einmal das DIR-Kommando ohne Parameter abgesetzt, damit auch klar ist, welche Datei dann letztendlich das Filterkriterium erfüllt.

```
A0>dir
CHK16550.COM : DELBR    .COM : PIP      .COM : SARGON   .COM
SD       .COM : STAT    .COM : TE       .COM : TECF     .COM
UNARC    .COM : UNCR    .COM : UNZIP    .COM : ZDE      .COM
ZDENST   .COM : DDT     .COM
A0>dir D??.com
DDT      .COM
A0>
```

8 Die Z80-Maschinensprache

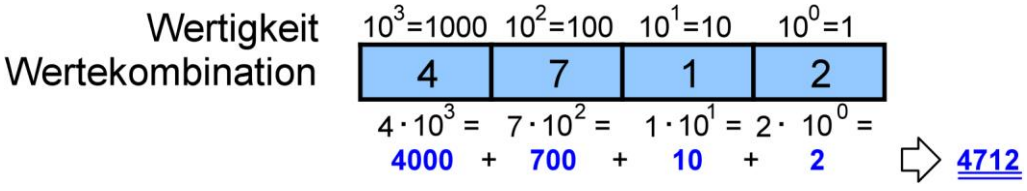
Nun geht es wirklich an's Eingemachte, denn wir beginnen mit der Programmierung von Z80-*Maschinensprache*. Nun, ganz so einfach ist das nicht und doch möchte ich einen geeigneten Einstieg vorstellen. Eigentlich stimmt die Überschrift nicht so ganz, denn wir programmieren nicht in Maschinensprache. Also was denn nun?! Was ist überhaupt Maschinensprache und wie schaut sie aus? Eine Maschinensprache, die genau genommen eigentlich keine Programmiersprache. Es handelt sich um einen Maschinencode, die ein Prozessor direkt ausführen kann. Wenn das so ist, dann muss es sich ja lediglich um Einsen und Nullen handeln, die eine Maschinensprache ausmacht. Ja und Nein! Wenn zum Beispiel die folgende Bitfolge vorliegt, dann handelt es sich beim Z80 schon um ein kleines Programm. Wir werden gleich im Detail sehen, was dieser Code so bewirkt.

```
00111110 00000100

00111100

11001001
```

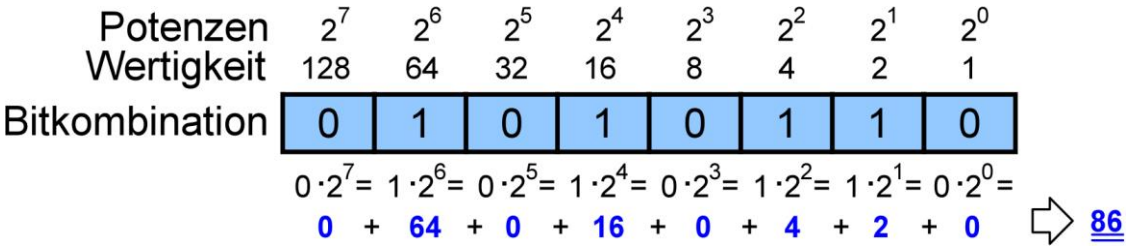
Nun ist es aber recht mühsam, wenn man sich als Programmierer mit diesen Einsen und Nullen herumschlagen muss, denn irgendwie ist das genauso aussagekräftig, wie ägyptische Hieroglyphen. Jedenfalls für einen Nicht-Sprachwissenschaftler.

8.1 Einführung in verschiedene Zahlensysteme

Unser bekanntes Dezimalsystem ist wie folgt aufgebaut, wobei jede einzelne Stelle die uns bekannten Wertigkeiten - von rechts nach links - Einer, Zehner, Hunderter, Tausender, usw. besitzt. Es arbeitet also in einem Stellenwertsystem zur Basis 10.



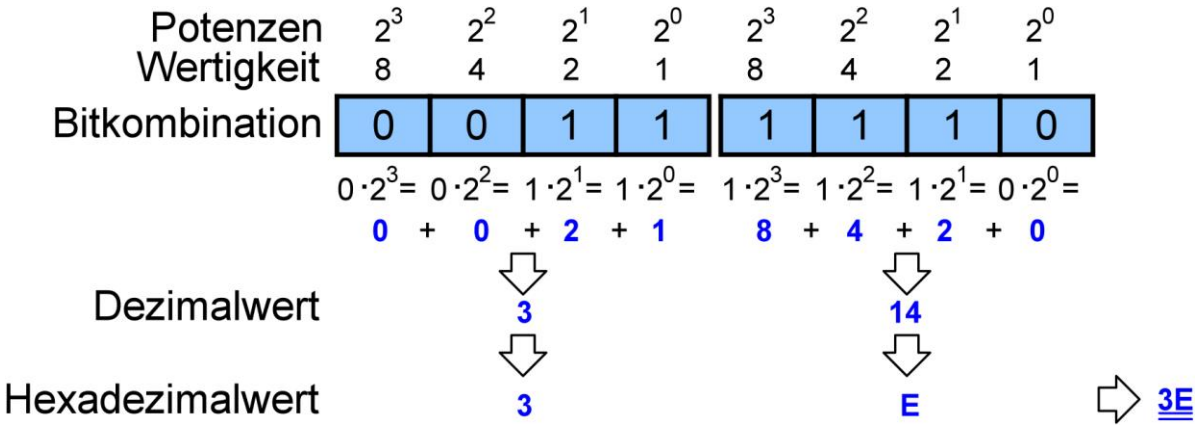
Das Ergebnis ist für uns Normalsterbliche natürlich sofort ablesbar, da wird dieses Zahlensystem in Form des Dezimalsystems täglich im Gebrauch haben. Eine Zahl hingegen, die lediglich aus den genannten Einsen und Nullen besteht, wird *Binärzahl* genannt. Hier schauen die Stellenwertigkeiten ein wenig anders aus und arbeitet in einem Stellenwertsystem zur Basis 2.



Um diese Bytefolgen, die ja aus vier 8-Bit-Gruppen besteht, etwas übersichtlicher zu gestalten, hat man die hexadezimalen Zahlen ins Leben gerufen. Diese beziehen sich immer auf 4-Bits - auch *Nibble* genannt. Im Hexadezimalsystem werden die einzelnen Zahlen in einem Stellenwertsystem zur Basis 16 abgebildet. Es gibt also statt 10 Ziffern (0 bis 9) des Dezimalsystems im Hexadezimalsystem 16 unterschiedliche Ziffern. Nun, das stimmt wiederum nicht ganz, denn es handelt sich dabei nicht nur um Ziffern, die ja auf 10 unterschiedliche Werte begrenzt sind. Es fehlen also noch 6 weitere und man hat sich dabei den Buchstaben *A* bis *F* bedient. In der folgenden Tabelle sind die dezimalen Werte den Hexadezimalen und Binärzahlen gegenübergestellt.

Dezimalzahl	Hexadezimalzahl	Binärzahl
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Hinsichtlich der ersten Bitkombination schaut das dann wie folgt aus.



Um nun die eben gezeigten Bitfolgen in Gänze in hexadezimale Werte (kurz: HEX-Werte) zu wandeln, gestaltet sich das Ergebnis wie folgt.

3	E	0	4
0011	1110	0000	0100

3	C
0011	1100

C	9
1100	1001

Der Maschinencode würde dementsprechend wie folgt lauten.

```
3E 04 3C C9
```

Ok, das ist schon ein wenig besser zu lesen, obwohl es immer noch einiges an Vorstellungskraft abverlangt. Auf diese Weise zu Programmieren ist sicherlich keine Freude, denn er handelt sich ja eigentlich um Befehle für die Z80-CPU, die sich dahinter verbergen. Gibt es keinen einfacheren Weg, der den Code irgendwie sprechender gestaltet? Den gibt es! Eine kleine Stufe über der Maschinensprache ist die sogenannte *Assemblersprache* angesiedelt. Die Assemblersprache - kurz auch *Assembler* genannt -, ist eine Programmiersprache, die auf den bestimmten Befehlsvorrat einer bestimmten CPU ausgerichtet ist. Hier natürlich die Z80-CPU. Diese Sprache bedeutet auf jeden Fall einen Fortschritt im Gegensatz zur Maschinensprache. Assembler ist leichter zu lesen und zu verstehen. Sehen wir uns doch einmal die ersten beiden Hexadezimalzahlen genauer an, denn sie gehören zusammen.

```
3E 04
```

Sie sagen der Z80-CPU, dass der Wert 04 in eine interne Speicherstelle geladen werden soll. Der Z80 besitzt einige unterschiedliche interne Speicherstellen, wobei eine spezielle Speicherstelle eine ganz besondere Stellung einnimmt. Es handelt sich um den sogenannten *Akkumulator* mit der Abkürzung *A*. In diesem Register, so werden die internen Speicherstellen auch genannt, werden alle Rechenoperationen durchgeführt. In Prosa ausgedrückt, würde die Anweisung wie folgt lauten.

„Lade den Wert 4 in den Akkumulator!“

Kommen wir noch einmal zum Maschinencode zurück, denn da haben die HEX-Werte die folgenden Bedeutungen. Der erste HEX-Wert *3E* ist der sogenannte OP-Code (Abkürzung für Operation-Code), der über die angegebene Zahl einen Maschinenbefehl repräsentiert, wobei die Summe aller OP-Codes den Befehlssatz des entsprechenden Prozessors bilden.

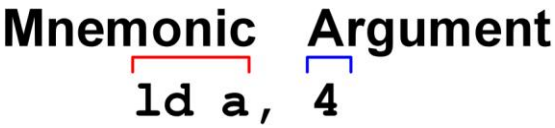
OP-Code	Befehl
3E	ld a

Der nachfolgende Wert *04* stellt das Argument für diesen Befehl dar. In Assembler lautet das dann wie folgt.

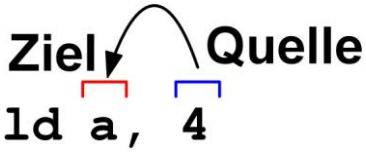
```
ld a, 4
```

Die beiden Buchstaben *ld* sind die Abkürzung für das englische Wort *Load*, was übersetzt „laden“ bedeutet. In Assembler wird jede Anweisung an den Computer durch ein sogenanntes mnemonisches

Symbol dargestellt, was ich gerade etwas lapidar *Befehl* genannt habe. Diese *Mnemonics* sind sehr kurze Wörter, die so ähnlich klingen, wie di zu repräsentierende Anweisung. Sollten wir uns diese Befehlszeile ein wenig genauer anschauen. Da befindet sich auf der linken Seite das Mnemonic und auf der rechten Seite das Argument.



Nun ist es in beim Assembler für den Z80 immer so, dass sich Ziel auf der linken und die Quelle auf der rechten Seite befindet. Das schaut dann wie folgt aus.



Die nächste Anweisung lautet binär

```
00111100
```

In der hexadezimalen Schreibweise wäre das der Maschinencode *3C*. Ok, keinen Schimmer, was das nun wieder zu bedeuten hat! Einen Blick in die Liste der OP-Codes für den Z80 zeigt, dass dieser Wert für *inc a* steht. Es handelt sich natürlich wieder um einen Mnemonic, wobei die Abkürzung *inc* für *Increment* steht, was übersetzt „Erhöhung“ bedeutet. Der Inhalt des Akkumulators soll also um den Wert 1 erhöht werden. Im Assembler wird das wie folgt geschrieben.

```
inc a
```

Die letzte Anweisung lautet binär

```
11001001
```

Das wird in HEX mit *C9* kodiert. Ein Blick in die Liste der OP-Codes zeigt, dass dies *ret* bedeutet und die Abkürzung für *return* ist. Return bedeutet übersetzt „zurück“ und gibt die Kontrolle zurück an den Aufrufer. In Assembler wird das wie folgt geschrieben.

```
ret
```

Eine Liste aller OP-Codes ist unter der folgenden Internetadresse zu finden.

http://map.grauw.nl/resources/z80instr.php

Ich zeige an dieser Stelle das Ergebnis noch einmal in einer Tabelle, damit es in Gänze übersichtlich erscheint.

Binärkombination(en)	HEX-Wert	Mnemonic
00111110 00000100	3E 04	ld a, 4
00111100	3C	inc a
11001001	C9	ret

Somit wäre das kleine Programm fertig. Doch wir haben das Pferd von hinten aufgezäumt und das hatte natürlich seinen Sinn. Ich wollte den Weg von den Einsen und Nullen über die HEX-Zahlen hin zu den Mnemonics des Assemblers aufzeigen. Ein Programmierer beschreitet genau den anderen Weg. Er

programmiert in Assembler mithilfe der Mnemonics und als Endergebnis fällt quasi hinten der Maschinencode raus. Genau diesen Weg werden wir jetzt gehen.

8.2 Die Programmierung über den Assembler

Die Programmierung in Assembler kann über zwei unterschiedliche Ansätze erfolgen, die ich beide vorstellen möchte. Zum einen - und damit werde ich beginnen - kann die Programmierung mit Bordmitteln, wie man das so schön sagt, erfolgen, also mit Programmen und Tools, die direkt für CP/M entwickelt wurden. Zum anderen gibt es noch die Möglichkeit, auf einem PC die Entwicklung zu starten und dann von dort aus mithilfe eines sogenannten Cross-Assemblers das Maschinenprogramm zu generieren. Bei einem Cross-Assembler handelt es sich um ist eine Spezialform eines Assemblers, der auf einer speziellen Computerplattform - also zum Beispiel auf einem PC - läuft und den Maschinencode für eine andere Computerplattform - dem Z80-Playground - generiert. Auch diesen Ansatz werde ich verfolgen. Zuerst sollten wir jedoch einen Blick auf die internen Speicherstellen des Z80 werfen, die ja bekanntermaßen *Register* genannt werden.

8.2.1 Die Z80-Register

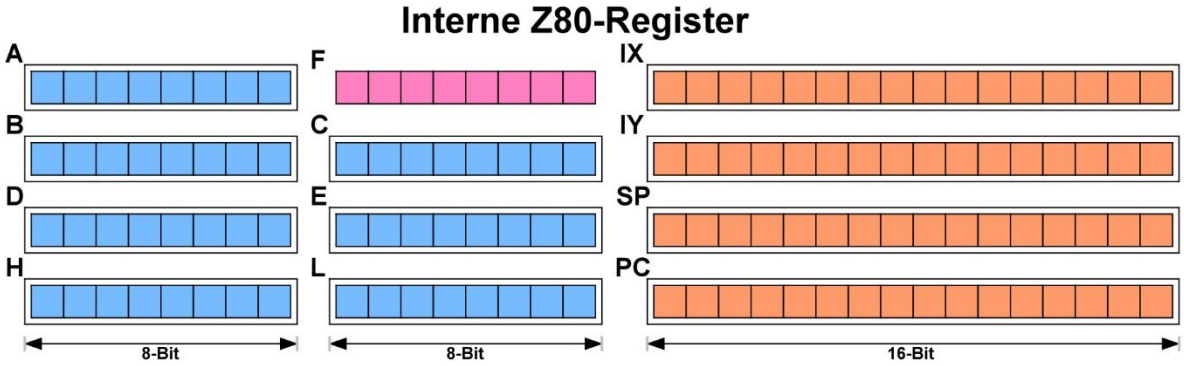


Abbildung 16 - Die Z80-Register

Diese Register haben unterschiedliche Datenbreiten, die 8- und 16-Bit breit sein können. Ein besonderes Register ist das *A*-Register, was für den *Akkumulator* steht, in dem alle Rechenoperationen durchgeführt werden und eine Breite von 8 Bits einnimmt. Das *PC*-Register, das die Abkürzung für *Program-Counter* ist, zeigt immer auf den nächsten Befehl, den die CPU abzuarbeiten hat und besitzt eine Datenbreite von 16-Bits. Dann gibt es noch die sogenannten Flags - hier mit *F* abgekürzt - in dem die Ergebnisse von verschiedenen Rechenoperationen in Form von Status-Bits abgebildet sind. Wir kommen natürlich noch im Detail darauf zu sprechen.

8.2.2 Der externe Speicher - ROM und RAM

Natürlich können nicht nur in den internen Registern des Z80 Daten gespeichert werden, es gibt ja auch noch den externen Speicher in Form der ROM und des RAM. Aber Vorsicht, denn nur das RAM kann Daten speichern. Das ROM besitzt bekannter Weise auch Daten, die jedoch nicht über den Z80 aktualisiert werden können.

8.2.3 Die Z80-Bordmittel verwenden

Bevor wir jedoch über den Assembler mithilfe der Mnemonics ein Programm erstellen, sind noch einige Details zu erwähnen. Das ursprüngliche CP/M war für den Prozessor 8080 entwickelt worden. Da wir

es nun jedoch mit einem Z80 zu tun haben, schaut die Sache - trotz der Abwärtskompatibilität vom Z80 zum 8080 - etwas anders aus. Der ursprüngliche Assembler *asm* für den 8080, besitzt andere Mnemonics als der Z80, was dazu führt, dass dieser Assembler für unsere Belange nicht genutzt werden kann, es sei denn, es werden die für den 8080 erforderlichen Mnemonics verwendet. Das möchte ich jedoch nicht, denn es geht hier um den Z80! Die Mnemonics für den Z80 wurden im Hinblick auf den 8080 vereinfacht. Es besteht jedoch keine Änderung der Technik des der Z80 zum 8080, denn nur die „Befehle“ wurden vereinfacht! Diese generieren jedoch hinsichtlich der Funktionalität den gleichen Maschinencode wie die 8080-Assembler-Mnemonics mithilfe der „alten“ Befehle. Kein Grund zur Sorge also. Es ist lediglich notwendig, einen anderen Assembler zur Programmierung zu verwenden. Alle erforderlichen Programme sind unter der folgenden Internetadresse zu finden. Notfalls auch auf meiner Internetseite.

https://github.com/MiguelVis/zsm

Doch bevor es losgeht, sollte ich einige Details besprechen. Ein ausführbares Programm unter dem Betriebssystem CP/M besitzt laut Konvention die Dateiendung *.COM*. Na, hat sich da das Betriebssystem *DOS* etwas abgeschaut?! Ein Schelm, wer Böses denkt! Es besteht also die Aufgabe, mithilfe eines Assemblers eine ausführbare Datei zu generieren, die die Dateiendung .COM besitzt. Das geht jedoch nicht, denn ein Assembler generiert immer nur eine sogenannte *Intel-HEX-Datei* mit der Dateiendung *.HEX*. Diese Datei ist quasi ein Zwischenprodukt auf dem Weg zur eigentlichen COM-Datei, die ja von uns beabsichtigt ist und die letztendlich einfach ausgeführt werden kann, was man von der Intel-HEX-Datei nicht behaupten kann. Das Intel HEX-Format wird zur Speicherung und Übertragung von binären Daten verwendet und wird auch heute noch dazu verwendet, um Programmierdaten für Mikrocontroller bzw. Mikroprozessoren zu speichern. Um nun eine Datei im Intel-HEX-Format in eine COM-Datei zu wandeln, wird ein kleines Tool - also Hilfsprogramm - benötigt, das für den 8080 *load* heißt. Doch sehen wir uns das in einem Arbeitsablauf - dem sogenannten *Workflow* - genauer an, der von links nach rechts abgearbeitet wird. Ganz links ist der Text-Editor *te* zu sehen, mit dessen Hilfe der Quellcode eingegeben wird. Ich würde auf keinen Fall den von CP/M standardmäßig vorhandenen Editor *ed* nutzen, denn er ist ein Graus und nur etwas für Programmierer, die mit körperlichen Schmerzen umzugehen wissen. Die erstellte Text-Datei muss die Endung *.asm* vorweisen, denn nur eine derartige Datei wird vom Assembler *asm* erkannt und beim Aufruf von asm darf die Dateierweiterung nicht mit angegeben werden. Das Ergebnis des Assembleraufrufes sind zwei Dateien, die die Endungen *.hex* und *.prn* besitzen. Die Datei .hex beinhaltet den schon erwähnten HEX-Code, der im nächsten Schritt mithilfe des Programms *load* dazu verwendet wird, eine ausführbare COM-Datei im Binärformat zu generieren. Die Datei mit der Endung .prn besitzt das ursprüngliche Quellprogramm, jedoch ergänzt um zusätzliche Assembler-Informationen in den äußersten 16 Spalten, die zum Beispiel die Programmadressen und den hexadezimalen Maschinencode enthalten. Diese Datei kann als Backup für die originale Quelldatei dienen. Wenn die Quelldatei versehentlich entfernt oder zerstört wird, kann die PRN-Datei durch Entfernen der äußersten linken 16 Zeichen jeder Zeile bearbeitet werden.

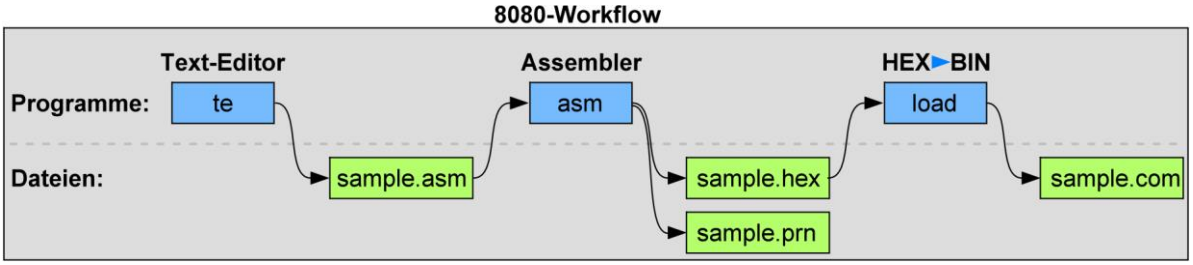


Abbildung 17 - Der Assembler-Workflow für den 8080

Der Assembler *asm* verarbeitet alle mnemonischen 8080-Befehle. Doch Stopp! Wir wollen ja nicht für den 8080 sondern für den Z80 programmieren. Zu diesem Zweck müssen ein paar zusätzliche

Programme installiert beziehungsweise einfach auf das entsprechende Laufwerk kopiert werden, die unter der gerade genannten Internetadresse zu finden sind. Es handelt sich dabei um die Programme

- ZSM.COM
- HEXTOCOM.COM
- DUMP.COM

Der Workflow schaut dabei ähnlich aus und verwendet anstatt *asm* das Programm *zsm* und anstatt *load* das Programm *hextocom*, wobei der Workflow ganz ähnlich ausschaut.

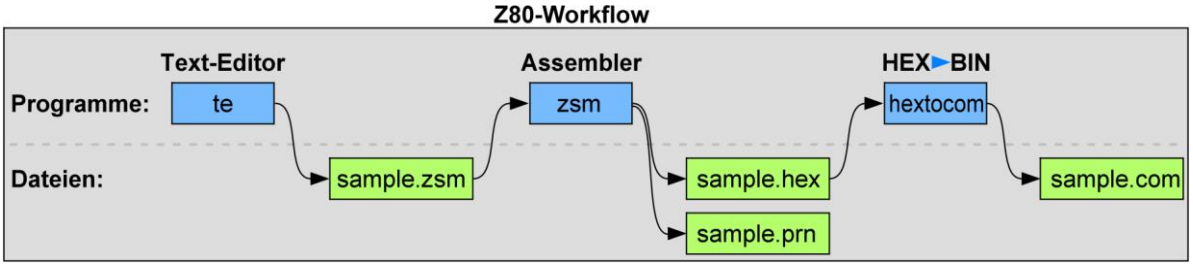


Abbildung 18 - Der Assembler-Workflow für den Z80

Ich denke, dass wir eine derartige Workflow-Session einmal komplett durchspielen sollten.

8.2.3.1 Die Quellcode-Eingabe über den Text-Editor TE

Ich möchte meine Z80-Programme auf dem Laufwerk *P:* speichern und verwalten und wechsle über die Eingabe *P:* in dieses Laufwerk. Der Text-Editor *te* befindet jedoch auf Laufwerk *A:*, was jedoch kein Problem darstellt, da CP/M zuerst immer auf Laufwerk *A:* nachschaut, ob sich das aufgerufene Programm dort befindet. Nach der Eingabe von *te* öffnet sich also der Text-Editor.

Abbildung 19 - Der Text-Editor TE

Es wird die momentan einzige vorhandene Zeilennummer 1 mit dem Cursor angezeigt. Nun kann der Quellcode eingegeben werden, doch ich muss zuvor noch ein paar Dinge erklären. Unter den vermeintlich modernen Betriebssystemen gibt es natürlich die Pfeiltasten, die es einem ermöglicht, den Cursor *rauf/runter* beziehungsweise *links/rechts* zu positionieren. Die ersten CP/M-Rechner besaßen jedoch keine derartigen Tasten, sodass es mit dem Navigieren hier nun etwas anders ausschaut. Es

wurden dazu die Tasten E und X beziehungsweise S und D genutzt, deren Positionen beziehungsweise Anordnung quasi dem Navigationskreuz der Pfeiltasten entspricht. Zusätzlich muss natürlich die Steuerungstaste *Strg* gedrückt werden, um die Eingabe von der normalen Texteingabe zu unterscheiden.

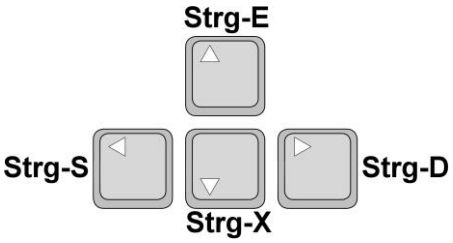


Abbildung 20 - Die Navigation im Text-Editor TE

Für das Löschen eines Zeichens - je nachdem, wo sich der Cursor gerade befindet - können sowohl die *Entfernen* - als auch die *Backspace*-Taste genutzt werden. Um eine Datei zu speichern muss das Menü über die *ESC*-Taste aufgerufen und später *Save* gewählt werden. Die gezeigten Großbuchstaben zeigen das jeweilige Tastenkürzel an. Dort sind auch andere wichtige Optionen zu finden.

Abbildung 21 - Die Optionen des Text-Editors TE

Alle wichtigen Menü-Befehle sind unter der Option *Help* zu finden. Bevor es nun wirklich losgehen kann, ist es wichtig zu wissen, wie die Struktur von Assemblerprogrammen respektive einer einzelnen Zeile denn so ausschaut. Eine Unterscheidung zwischen Groß- beziehungsweise Kleinschreibung findet hier übrigens nicht statt. Es ist also *nicht Case-Sensitiv*, wie man Neu-Deutsch so sagt. In Assembler werden die einzelnen Befehle also durch Textzeilen formuliert, wobei jeder Befehl einer einzigen Zeile entspricht. Eine derartige Zeile besitzt die folgende Struktur.

```
[<Label>[:]]<Space/Tab><Befehl>[[<Space/Tab><Argument1>],<Argument2>]
```

Kommentare werden im Assembler mit einem Semikolon eingeleitet. Also zum Beispiel wie folgt.

```
    ld a, 4  ; Akku laden
```

Da es wichtig ist, an welcher Stelle im Speicher sich das Programm später befinden soll, ist eine zusätzliche Angabe erforderlich. Sie wird über die eine sogenannte Assembler-Direktive - auch Pseudo-Anweisung genannt - angegeben, die sich *org* nennt. Wir nutzen dazu den Speicher ab 100h. Doch nun zum eigentlichen Programm, dass ich in den Text-Editor eingegeben und unter dem Namen *sample.zsm* abgespeichert habe. Da bei diesem Programm noch keine Labels - also Sprungmarken - zur Anwendung kommen, habe ich jede einzelne Zeile mit einem Tabulator nach rechts eingerückt. Wenn das nicht gemacht wird, liefert der Assembler *zsm* mehrere Fehler.

```
    org 100h
    ld a, 4
    inc a
    ret
```

Um zu sehen, wie es auf dem Laufwerk *P:* nun ausschaut, habe ich das Kommando *dir* abgesetzt.

```
P0>dir
ZSM     .COM : HEXTOCOM.COM : DUMP    .COM : TE      .BKP
SAMPLE  .ZSM
P0>
```

Die Datei *sample.zsm* ist also vorhanden.

8.2.3.2 Der Aufruf des Assemblers

Um nun diese Quelldatei zu assemblieren, wird das Programm *zsm* mit dem Namen der Datei ohne den Zusatz *.zsm* aufgerufen, was dann das folgende Ergebnis liefert.

```
P0>zsm sample
Zilog/Mostek Z80 Assembler Version 3.4 (Z80 CPU)

Pass 1
Pass 2

Errors    0

Finished


P0>
```

Der Assembler hat die Quelldatei also ohne einen Fehler (Errors 0) übersetzt. Dann wollen wir einmal sehen, wie es jetzt auf dem Laufwerk *P:* ausschaut.

```
P0>dir
ZSM     .COM : HEXTOCOM.COM : DUMP    .COM : TE      .BKP
SAMPLE  .ZSM : SAMPLE  .HEX : SAMPLE  .PRN
P0>
```

Es ist zu sehen, dass zwei neue Dateien erstellt wurden, die *sample.hex* und *sample.prn* lauten. Die für uns im Moment wichtige Datei ist die mit der Endung *.hex*, denn diese wird benötigt, um eine ausführbare COM-Datei zu generieren.

8.2.3.3 Eine COM-Datei generieren

Das erfolgt mit dem Aufruf des Programms *hextocom* mit dem Namen der Datei ohne den Zusatz *.hex*. Das schaut dann wie folgt aus.

```
P0>hextocom sample
HexToCom v1.05 / 10 Jan 2016

(c) 2007-2016 FloppySoftware

First address: 0100
Last address:  0103
Size of code:  0004 (4 dec) bytes

P0>
```

Es ist zu sehen, dass das Programm *sample.com* nach der Ausführung ab der Speicherstelle 0100h im Speicher liegt, wobei das letzte Byte die Adresse 0103h belegt und das Programm in Summe eine Länge von 4 Bytes besitzt. Ein erneuter Blick auf das Laufwerk *P:* zeigt, dass nun eine Datei mit dem Namen *sample.com* generiert wurde.

```
P0>dir
ZSM      .COM : HEXTOCOM.COM : DUMP     .COM : TE       .BKP
SAMPLE   .ZSM : SAMPLE   .HEX : SAMPLE   .PRN : SAMPLE   .COM
P0>
```

8.2.3.4 Ein Blick in die COM-Datei - DUMP

Kann man denn nun überhaupt einen Blick in die generierte binäre und ausführbare Datei mit der Endung *.COM* werfen? Das ist möglich, denn für diese Zwecke gibt es das Programm *dump*. Nähere Informationen zu *ddt* sind unter der folgenden Internetadresse zu finden.

http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch1.htm#Section_1.6.8

Dieses Programm zeigt den Inhalt einer Datei mit ihren binären Werten in Form von einzelnen Byte-Blöcken sowohl als HEX- als auch als ASCII-Werte an. Dann wollen wir einen Blick in die Datei *sample.com* werfen, in dem *dump* gefolgt vom entsprechenden Dateinamen aufgerufen wird.

```
P0>dump sample.com
0000 : 3E 04 3C C9 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A : >.<.............
0010 : 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A : ................
0020 : 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A : ................
0030 : 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A : ................
0040 : 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A : ................
0050 : 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A : ................
0060 : 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A : ................
0070 : 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A : ................

P0>
```

Die ersten vier Bytes zeigen genau die zuvor generierten Bytes in Form von HEX-Zahlen, die dem Maschinencode unseres Programms entsprechen. Jede, der hier gezeigten Zeilen besitzt 16 Bytes und eine führende Nummerierung, die ebenfalls eine HEX-Zahl ist und auf der rechten Seite befindet sich der Bereich, wo die einzelnen Bytes in Form von ASCII-Zeichen angezeigt werden. Die führende Nummerierung repräsentiert jedoch nicht die eigentlichen Speicherstellen, an denen sich das Programm befindet, das ja ab der Speicherstelle 100h abgelegt werden soll. Befindet sich das Programm *sample.com* denn überhaupt schon in diesem Bereich ab 100h? Das Programm *dump* zeigt nur den Inhalt einer Datei an und nicht die Inhalte von Speicherstellen im ROM beziehungsweise RAM. Ist das auch möglich? Natürlich!

8.2.3.5 Speicherbereiche anzeigen - DDT

Dazu wird das Programm *ddt* benötigt, was die Abkürzung für *Dynamic Debugging Tool* ist. Nähere Informationen zu *ddt* sind unter den folgenden Internetadressen zu finden.

- http://www.cpm.z80.de/randyfiles/DRI/DDT.pdf
- http://www.gaby.de/cpm/manuals/archive/cpm22htm/ch4.htm

Dann wollen wir einen Blick in die Datei *sample.com* werfen, in dem *ddt* gefolgt vom entsprechenden Dateinamen aufgerufen wird. Nach dem Strich, was das Bereitschaftszeichen von *ddt* ist, muss dann *d* eingegeben, um einen Dump zu erzeugen.

```
P0>ddt sample.com
DDT VERS 1.4
NEXT  PC
0180 0100
-d
0100 3E 04 3C C9 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A >.<.............
0110 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0120 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0130 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0140 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0150 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0160 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0170 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A ................
0180 1A 84 12 13 C3 69 01 D1 2E 00 E9 2A 7C 1D EB 0E .....i.....*|...
0190 1A CD 67 1B C9 3E 0C D3 01 3E 08 D3 01 DB 01 07 ..g..>...>......
01A0 07 07 1F DA A9 08 C3 9D 08 DB 03 E6 7F C9 21 83 ..............!.
01B0 1D 70 2B 71 2A 82 1D 44 4D CD A1 07 0E 3A CD 86 .p+q*..DM....:..
-
```

Der Inhalt sieht ähnlich dem aus, der mit *dump* erzielt wurde und doch beinhalten die Informationen weitere Details. Ganz zu Beginn werden zwei wichtige Hinweise gegeben.

- *NEXT*: Gibt die Adresse an, die als nächste belegt würde - hier *0180h*
- *PC*: Steht für *Programm-Counter* und ist der Programmzähler, der zeigt, an welcher Speicherstelle das Programm sich nach der Ausführung im Speicher befindet

Es ist über *ddt* auch möglich, sich den Inhalt in Form der Assembler-Mnemonics anzuschauen. Es muss dazu nur der Befehl *l* (kleines L) abgesetzt werden Hier wird jedoch der Code so angezeigt, als wäre er für den 8080 geschrieben worden. *MVI A* entspricht *LD A* und *INR A* entspricht *INC A*.

```
-l
  0100  MVI  A,04
  0102  INR  A
  0103  RET
  0104  LDAX D
  0105  LDAX D
  0106  LDAX D
  0107  LDAX D
  0108  LDAX D
  0109  LDAX D
  010A  LDAX D
  010B  LDAX D
-
```

Das interessante an *ddt* ist, dass das geladene Programm schrittweise beziehungsweise zeilenweise abgearbeitet werden kann und die Möglichkeit besteht, verschiedene Parameter wie *PC* (Program-Counter) und *A* (Akkumulator) zu inspizieren. Bei beiden handelt es sich um interne Register in im Z80 und es gibt noch einige andere mehr, auf die ich noch zu sprechen komme, die im Moment jedoch nicht von Bedeutung sind. Vorab zeige ich hier schon einmal eine verkürzte Liste mit einigen Tastenkürzeln, um *ddt* zu bedienen.

Kommando	Bedeutung
`D - Dump`	Zeigt den Speicher in Hexadezimal und ASCII an
`L - List`	Listet Speicher mit Assembler-Mnemonics auf
`R - Run`	Startet ein Programm schrittweise zum anschließenden Testen
`T - Trace`	Verfolgt die Programmausführung mit der Anzeige der Registerinhalte

8.2.3.6 Schrittweise Ausführung eines Programms - DDT

Im nächsten Schritt wollen wir das geschriebene Programm etwas erweitern, damit der Inhalt des Akkumulators erhöht und erniedrigt wird. Dann lassen wir uns den Inhalt anzeigen. Bevor das jedoch geschieht, sollten wir einen Blick auf den sogenannten *PC* (Program-Counter) werfen, denn das ist ein essentieller Zähler innerhalb einer CPU. Der Z80 muss quasi im Auge behalten, an welcher Stelle im Speicher er den Code denn gerade ausführt und was als nächstes auszuführen ist. Er speichert diese Adresse in dem 16-Bit breiten PC-Register, wobei es diese Breite besitzen muss, um den ebenfalls 16-Bit breiten Adressbus komplett adressieren zu können. Der *PC* zeigt immer auf die Adresse, die als nächstes auszuführen ist. Das schauen wir uns am besten am schon erwähnten und erweiterten Quellcode an, der wie folgt ausschaut.

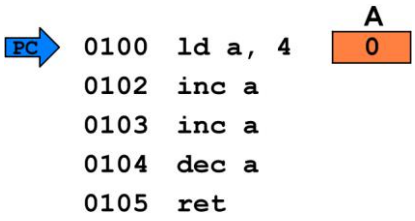
```
    org 100h
    ld a, 4
    inc a
    inc a
    dec a
    ret
```

Es ist zu sehen, dass 2x ein *inc a* erfolgt und dann 1x ein *dec a*, was für *Decrement* steht und Erniedrigung um den Wert 1 bedeutet. Der Inhalt des Akkumulators wird also schrittweise die Werte 4, 5, 6 und letztendlich 5 durchlaufen. Das hört sich sehr trivial an, was es ja auch ist, doch das Zusammenspiel von A und PC sollte genauer untersucht werden. Dabei hilft uns natürlich wieder das Programm *ddt*. Zuerst muss also der gezeigte Quellcode nach gezeigter Abfolge eingegeben, assembliert und in eine COM-Datei überführt werden, was ich hier nicht erneut zeigen werden. Im Anschluss erfolgt der Aufruf von *ddt*, wobei ich die neue Datei *sample2.zsm* genannt habe. Um die Mnemonics anzuzeigen, wurde in *ddt* wieder *l* eingegeben, was zur folgenden Ausgabe führte.

```
P0>ddt sample2.com
DDT VERS 1.4
NEXT  PC
0180 0100
-l
  0100  MVI  A,04
  0102  INR  A
  0103  INR  A
  0104  DCR  A
  0105  RET
  0106  LDAX D
  0107  LDAX D
  0108  LDAX D
  0109  LDAX D
  010A  LDAX D
  010B  LDAX D
-
```

Natürlich werden hier wieder die 8080-Mnemonics zur Anzeige gebracht, was jedoch nicht weiter stören soll, denn es geht primär um den Maschinencode und dessen Ausführung, der ja - wie schon erwähnt - bei 8080 und Z80 identisch ist. Nun wollen wir das Programm schrittweise ausführen, um dann die Register A und PC zu inspizieren. Bei der nachfolgen Erläuterung verwende ich jedoch wieder die Z80-Mnemonics, um die Verwirrung nicht allzu groß werden zu lassen. Wie wir sehen, zeigt der PC zu Beginn auf die Speicherstelle 0100h, wobei diese Ausführung jedoch noch nicht stattgefunden hat. Der Inhalt des Akkumulators ist noch 0 oder besitzt einen anderen Wert, der uns jedoch nicht weiter zu interessieren hat.

Schritt 1: Ausgangssituation



Um den Zustand mithilfe von *ddt* nun abzufragen, muss *t* für Trace eingegeben werden. Es ist zu sehen, dass *A=00* und *P(PC)=0100* sind. Rechts vom PC ist das Mnemonic *MVI A, 04* für den 8080 zu sehen, das im nächsten Schritt zur Ausführung kommt.

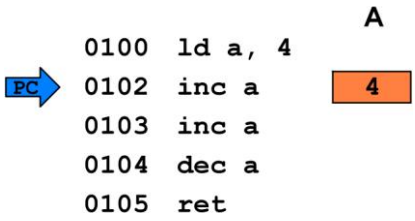
```
-t
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  A,04*0102
-
```

Schritt 2: Ausführung 1. Befehl ld a, 4

Um den 1. Befehl jetzt zur Ausführung zu bringen, wird mit *r* für Run diese Zeile abgearbeitet und der PC auf die nächste Zeile positioniert. Im Anschluss können mit *t* wieder die Registerinhalte angezeigt werden.

```
-r
NEXT  PC
0180 0102
-t
C0Z0M0E0I0 A=04 B=0000 D=0000 H=0000 S=0100 P=0102 INR  A*0103
-
```

Es ist zu sehen, dass der Akkumulator jetzt den Wert *04* besitzt und der PC auf die nächste Zeile *0102h* positioniert wurde. Es ist dabei zu beachten, dass der PC nicht immer um den Wert 1 erhöht wird, denn bei *ld a, 04* handelt es sich um einen 2-Byte-Befehl. Auf der folgenden Abbildung ist das noch einmal übersichtlicher dargestellt.

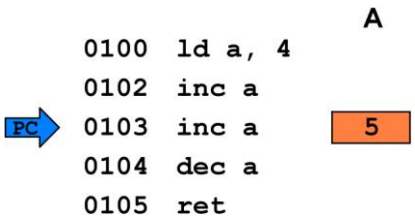


Schritt 3: Ausführung 2. Befehl inc a

Um den 2. Befehl zur Ausführung zu bringen, wird wiederum mit *r* diese Zeile abgearbeitet und der PC auf die nächste Zeile positioniert. Im Anschluss können mit *t* wieder die Registerinhalte angezeigt werden.

```
-r
NEXT  PC
0180 0103
-t
C0Z0M0E0I0 A=05 B=0000 D=0000 H=0000 S=0100 P=0103 INR  A*0104
-
```

Es ist zu sehen, dass der Akkumulator jetzt den Wert *05* besitzt, da er über *inc a* um den Wert 1 erhöht und der PC auf die nächste Zeile *0103h* positioniert wurde. Auf der folgenden Abbildung ist das erneut übersichtlicher dargestellt.

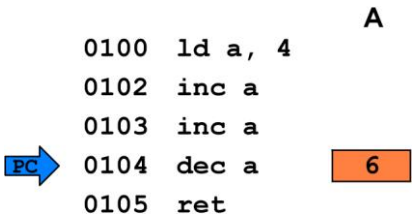


Schritt 4: Ausführung 3. Befehl inc a

Um den 3. Befehl zur Ausführung zu bringen, wird wiederum mit *r* diese Zeile abgearbeitet und der PC auf die nächste Zeile positioniert. Im Anschluss können mit *t* wieder die Registerinhalte angezeigt werden.

```
-r
NEXT  PC
0180 0104
-t
C0Z0M0E0I0 A=06 B=0000 D=0000 H=0000 S=0100 P=0104 DCR  A*0105
-
```

Es ist zu sehen, dass der Akkumulator jetzt den Wert *06* besitzt, da er über *inc a* wieder um den Wert 1 erhöht und der PC auf die nächste Zeile *0104h* positioniert wurde. Auf der folgenden Abbildung ist das erneut übersichtlicher dargestellt.

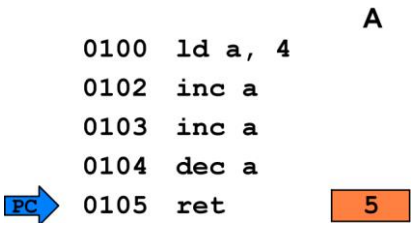


Schritt 5: Ausführung 4. Befehl dec a

Um den 4. Befehl zur Ausführung zu bringen, wird wiederum mit *r* diese Zeile abgearbeitet und der PC auf die nächste Zeile positioniert. Im Anschluss können mit *t* wieder die Registerinhalte angezeigt werden.

```
-r
NEXT  PC
0180 0105
-t
C0Z0M0E0I0 A=05 B=0000 D=0000 H=0000 S=0100 P=0105 RET  *043E
-
```

Es ist zu sehen, dass der Akkumulator jetzt den Wert *05* besitzt, da er diesmal über *dec a* um den Wert 1 erniedrigt und der PC auf die nächste Zeile *0105h* positioniert wurde. Auf der folgenden Abbildung ist das erneut übersichtlicher dargestellt.



Schritt 6: Ausführung 5. Befehl ret

Über den nächsten Befehl *ret* würde die Kontrolle zurück an den Aufrufer gegeben und das Programm wäre abgearbeitet.

Das hier vorgestellte Programm ist natürlich sehr einfach, doch es bietet sicherlich einen guten Einstieg in die Handhabung von Programmen, den Umgang mit dem Text-Editor *TE*, dem Programm *ddt* und vieles mehr, um aus einer Quelldatei eine COM-Datei über einen Assembler und weiteren Tools zu erstellen.