Pokèmon Mini Assembler

Oliver Skawronek

Inhalt

- Einführung
- Technische Daten
- Assembler - was ist das
- Benötigte Ausrüstung
- Auf in den Kampf - das erste Programm
- Stellensysteme
- Register - Speichermedien ohne Größenwahn
- Bits auf großer Reise - Move als Reiseleiter
- Labels
- Grundlagen des PMAs
- Staubtrockene Theorie - die ALU
- Logische Verknüpfungen
- Rechnen mit dem PMini
- Das große Bitgeschupse
- Rotieren bis einem schwindelig wird
- Sterne zählen für Romantiker
- Der Stack
- Variablen
- Unbedingte Sprünge
- Bedingte Sprünge
- Schleifen
- Schleifen
- Assembler-Direktiven

Einführung

Natürlich schreckt der Name „Pokèmon‟ fürs erste ab - jedoch zu Unrecht. In dem kleinen Meisterwerk von Nintendo steckt eine Menge an Features. Zu erwähnen sei hier die IR-Schnittstelle, der Rumblemotor(zum Erzeugen von Vibrationen), ein Schocksensor und eine RTC(Realtime Clock). Hauptsächlich macht der Preis der Plattform sie so interessant. Bei Preisen unter 10€ ist dieses System recht erschwinglich.

In diesem Tutorial soll es grundsätzlich um die Programmierung des Pokèmon Mini, oder „PMini‟, wie er im Folgenden bezeichnet wird, gehen. Es richtet sich an die absoluten Einsteiger in der Assemblerprogrammierung. Erfahrungen, die in anderen Programmiersprachen gesammelt wurden, tragen zu einem besseren Verständnis bei, sind aber nicht dringend notwendig.

Es sei hier erwähnt, dass die Architektur des PMini nicht vollständig „geknackt‟ ist. Nintendo gibt leider keine Informationen diesbezüglich frei. Der Prozessor scheint jedoch ein modifizierter Z80 zu sein.

Technische Daten

CPU (genannt „Minx‟)

* 8 Bit mit 4 MHz

ROM

* Bis zu 1 MByte Größe

RAM 4096 Bytes davon

* 1056 Bytes VideoRAM
* 3040 Bytes Benutzer

Video

* 96 x 64 Pixel Auflösung
* Monochrom (schwarz/weiß - über Flickering auch grau möglich)
* Hardware Tiles und Sprites
* Einstellbarer Kontrast per Software

Sound

* 3 Lautstärke Stufen
* Rechteck Impulsgenerator mit einstellbarer Pulsweite

Extras

* 1 x Rumblemotor
* 1 x Schocksensor
* 6 x Slots Savegame RAM
* 1 x RTC
* 1 x IR-Schnittstelle

Versorgung

* 1 x AAA 1,5 Batterie

Assembler - was ist das?

Generell ist jeder Prozessor, der programmiert werden kann, ein Computer. Somit ist auch unser PMini ein kleiner Computer. Dabei bringt jeder Prozessor seinen eigenen Befehlssatz mit. Diese Befehle sind im Wesentlichen in die Kategorien Logik, Arithmetik, Adresszugriff und Sprünge zu unterscheiden. Alles geschieht hier auf Binär Ebene.

Jedem Befehl ist eine Zahl, ein so genannten OpCode(= Operation Code) zugewiesen. Nun ist es jedoch sehr umständlich, mit Zahlen zu programmieren - jeder hat sicherlich schon von den alten Lochkarten gehört. Deswegen gibt es so genannte Assembler, die aus Befehlsnamen - also mnemonische Symbole(kurz Mnemonics) - diese Zahlen bilden. Sie wandeln Assemblercode in ausführbaren Maschinencode um.

Das Rückschließen von Maschinencode auf dessen Assemblercode wird als Disassemblierung bezeichnet. Dabei gehen Kommentare aus den Quellcode, sowie Namen für Operanden u. ä. verloren.

Wie darf man sich nun so einen Assemblerbefehl vorstellen? Befehle wie „Bewege Held nach rechts" wären zwar nicht schlecht, jedoch gäbe es sicherlich nicht den Beruf Programmierer, wenn dem so wäre. Vielmehr muss man die lästige, jedoch einfache Mathematik dahinter sehen. Auf unser Beispiel angewandt wäre es HeldX = HeldX+1 bzw. `INC A, 1`.

Zusammengefasst ist Assembler ein Begriff für die Programmiersprache oder den Übersetzer dieser Sprache.

Benötigte Ausrüstung

Grundausstattung eines jeden Programmierers ist natürlich viel Kaffee und/oder Nikotin. Hat man keine Sucht, sollte man auch gar nicht anfangen zu programmieren. Läuft noch gute Musik(in voller Lautstärke!) im Hintergrund kann es dann auch losgehen mit den weniger wichtigen Sachen.

Wie bereits erwähnt gäbe es da den Assembler, der unseren Assemblercode in ausführbaren Maschinencode umwandelt.

Da die wenigsten über entsprechende Hardware wie Flasher und Flashkarten verfügen, benötigen wir ein Emulator der den PMini auf dem PC emulieren kann(wir wollen ja auch sehen, was wir da für Mist zusammen geschustert haben).

Stinkt dieser Mist zu gewaltig in den Himmel, ist ein Debugger erforderlich, der den Code(nein hier ist absolut <u>kein</u> Wortspiel angebracht) schrittweise ausführen kann. Das hilft uns das stinkende Etwas einzugrenzen, und mit einem Stock([Entf]-Taste) vom Schuh zu kratzen.

Reverse Engeneering ist in Deutschland nicht verboten, also können wir unsere legal kopierten ROMs genauer unter die Lupe nehmen, und von den Profis lernen. Dies

geschieht mit einem (wer hätte es gedacht?) Disassembler.

Die tollsten Entwicklertools nützen einem jedoch gar nichts, solange man keine Ahnung hat, wie der PMini gestrickt ist. Dafür gibt es Hardware Dokumentationen. Unsere stammen jedoch nicht direkt vom Entwickler Nitnendo(alias „Big N"), sondern von motivierten Hobbyisten aus der Szene.

Open Source Assembler/Disassembler

* pmas-13b.zip

Open Source Emulator/Debugger

* minimon.zip

BIOS

* bios.min

Hardware Dokumentation

* www.sublab.net/pokemini/
* MindX13.txt

Auf in den Kampf, das erste Programm

Ich habe nun gut über 12 Programmiersprachen hinter mir, und bei keiner habe ich mir ein „Hello World" nehmen lassen. Bei einem Mikrocontroller ist so etwas jedoch ein etwas zu großes Projekt. Wir beschränken uns somit bei unserem ersten Programm auf das simple Addieren von 1 und 2.

Wir legen einen neuen Ordner namens *addieren* an und darin eine Datei *addieren.asm*. Die meisten Assemblercodes haben als Dateiendung *.asm oder *.s, die ausführbaren Dateien *.bin oder *.min.

Inhalt von addieren.asm:

```
.orgfill 0x02100
     .db "MN"       ; steht für MiNi
     jmp main       ; Springe zum Hauptprogramm

.orgfill 0x021A4
     .db "NINTENDO"
     .db "1234"     ; Spielcode + Firmencode
     .db "Addieren" ; Spielname

.orgfill 0x021BC
     .db "2P"

.orgfill 0x021D0
main:
     ; 1+2
```

```
     mov a, 1 ; lade 1 in Register A
     add a, 2 ; addiere 2 zu Register A

end:
     jmp end ; Endlosschleife

.end
```

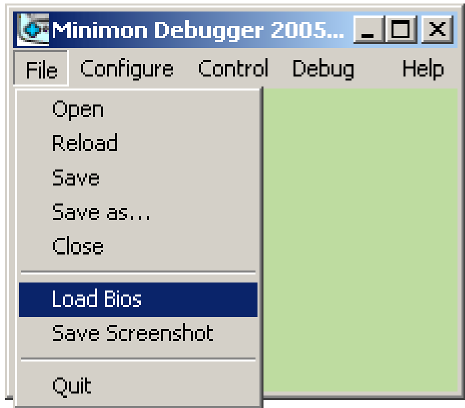
Nun ist es an der Zeit den Code zu übersetzen. Dafür ist PMAs zuständlich. PMAs steht für „Pika Macro ASsembler". Da wir nicht in einer bequemen Entwicklungsumgebung arbeiten, sondern im guten alten Notepad, müssen wir noch per Hand, sprich per Kommandozeile, assemblieren. Dazu öffnen wir die Eingabeaufforderung (am einfachsten per Start -> Ausführen -> cmd bzw. command) und starten den PMAs mit entsprechenden Parametern. z. B.

```
E:
cd Dev\PMini\PMAs\
PMAs.exe ..\Projekte\addieren\addieren.asm
..\Projekte\addieren\addieren.bin ..\Projekte\addieren\addieren.sym
```

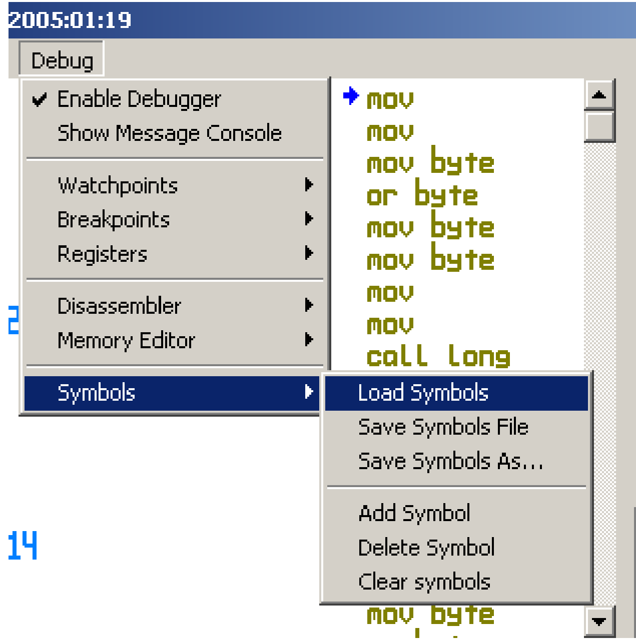
```
C:\WINNT\System32\cmd.exe
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>E:

E:\>cd Dev\PMini\pmas

E:\Dev\PMini\pmas>pmas.exe ..\Projekte\addieren\addieren.asm ..\Projekte\addiere
n\addieren.bin ..\Projekte\addieren\addieren.sym
Pika Macro ASsembler v0.13b (build Sep 15 2004) by Rafael Vuijk. http://darkfade
r.net/pm/

E:\Dev\PMini\pmas>_
```

Danach sollte im Ordner *E:\Dev\PMini\Projekte\addieren* die weiteren Dateien *addieren.bin* (ausführbarer Maschinencode) und *addieren.sym* (Symboldatei für den Debugger) zu finden sein.

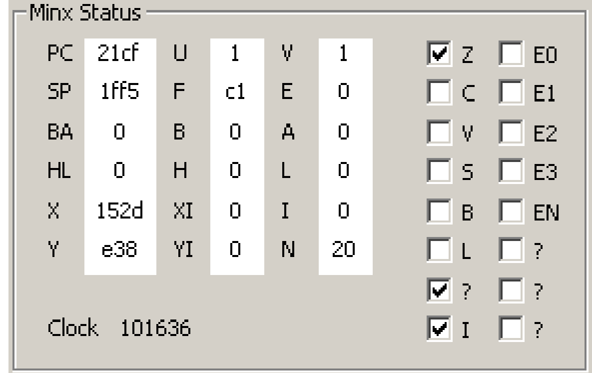
Mit den übersetzten Dateien geht es nun ab in den Emulator/Debugger - unseren Minimon. Also *Minimon.exe* starten und erst einmal das BIOS laden:



Damit wir vernünftig Debuggen können (auf dem Display zeigen wir ja nichts weiter an), schalten wir den Debugmode an, in dem wir Debug -> Enable Debugger wählen und die Symboldatei *addieren.sym* laden:



Nun können wir unsere ausführbare Datei *addieren.bin* per File -> Open öffnen. Um sie zu testen, geben wir die Startadresse 0x21CF in den Program Cursor ein und drücken [Enter]:

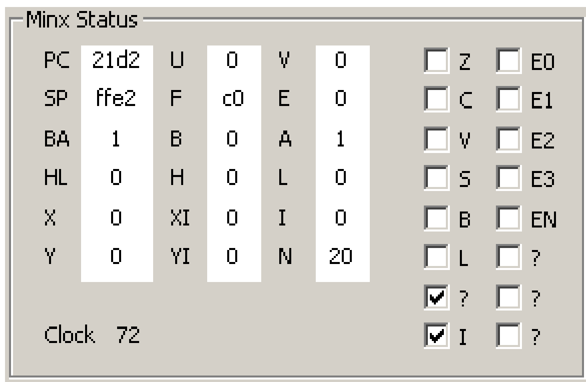


Nach dem die Adresse eingetragen wurde, klicken wir z. B. in das grüne Display. Es darf zumindest nicht mehr der Cursor im Eingabefeld blinken. Ist das geschehen, springen wir zu dieser Adresse durch drücken von [Space].

Drücken wir nochmal die [Space]-Taste, wird unser Code

```
mov a, 1 ; lade 1 in Register A
```

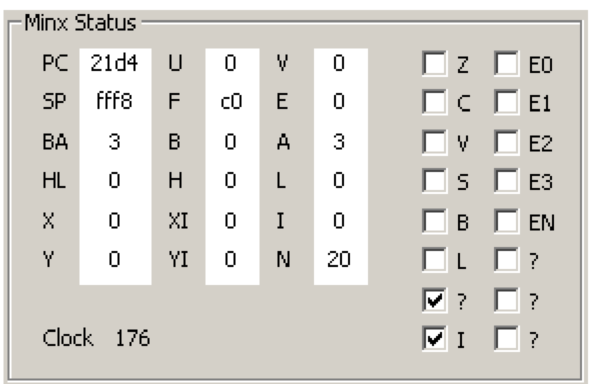
ausgeführt. Schauen wir in den Statusmonitor, können wir sehen, dass Register A den Wert 1 hat.



Nochmals die [Space]-Taste betätigen und

```
add a, 2 ; addiere 2 zu Register A
```

wird ausgeführt:



Es ist also geschafft, er hat 1 + 2 gerechnet. Das Ergebnis steht nun in Register A. Der weitere Code ist eine Endlosschleife, der immer wieder zu der selben Stelle springt. Generell ist es ratsam sich eine Batchdatei *make.bat* anzulegen, die den PMAs mit samt Parametern selber startet. Es genügt dann nur noch ein Doppelklick auf diese zu tätigen. Was überhaupt Register sind, welche weiteren Befehle es gibt usw., soll in den nächsten Abschnitten geklärt werden.

Stellensysteme

Als Assemblerprogrammierer ist es ein Muss, neben dem arabischen (unserem Dezimalsystem), auch das binäre und das hexadezimale Stellensystem zu kennen. Das Binärsystem basiert auf Einsen und Nullen und ist für Werte interessant. Das Hexadezimalsystem hingegen wird häufig bei Speicheradressen zum Einsatz kommen.

Das Dezimalsystem

Um die anderen Stellensysteme kennen zu lernen, sollte erst einmal geklärt werden, wie unseres Aufgebaut ist. Schon im alten Ägypten gab es eine Abwandlung dieses Systems. Es hatte nur noch keine Null(wie es z. B. beim römischen Zahlensystem auch der Fall ist). Ursprünglich kommt es jedoch aus Indien. Dezi steht für 10. Stellensysteme werden generell nach ihrer Basis benannt, so auch z. B. das Oktalsystem mit der Basis 8, das Binärsystem mit Basis 2 usw. Unser Stellensystem hat somit die Basis 10. Nehmen wir einmal die Zahl 5124, das ist nur die Kurzform von:

```
  4 * 10^0 (= 4)
+ 2 * 10^1 (= 20)
+ 1 * 10^2 (= 100)
+ 5 * 10^3 (= 5000)
```

(Nach dem Komma wird mit 10^-1, 10^-2 usw. multipliziert)

Das Binärsystem

Kommen wir zu dem Stellensystem, das unser Prozessor versteht. Das Binär-, oder auch Dualsystem genannt, besitzt die Basis 2 und hat die Nennwerte {0, 1}. Hiervon kommt auch die Speichergröße Bit. Bit steht für Binary Digit, und kann nur die Zustände 0 oder 1 abspeichern.

Nehmen wir die Zahl 100101b

```
  1 * 2^0 (= 1)
+ 0 * 2^1 (= 0)
+ 1 * 2^2 (= 4)
+ 0 * 2^3 (= 0)
+ 0 * 2^4 (= 0)
+ 1 * 2^5 (= 32)
----------------
  37
```

Ein Byte, das aus 8 Bit besteht, kann übrigens 2^8 = 256 verschiedene 0 und 1 Kombinationen speichern (und jetzt das ganze mal für eine 180 GByte Festplatte).

Das Hexadezimalsystem

Fälschlicherweise auch einfach nur als *Hexsystem* bezeichnet, hat die Basis 16 (Hex = 6 + Dezi = 10 = 16). Ein Hexadezimal Wert nimmt ein Halbbyte, Tetrade, oder bei den Mikrocontrollern auch als Nibble bezeichnet, ein. Die Nennwerte dieses Stellensystems sind {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. Wer zumindest einmal eine Webseite in HTML-Code verfasst hat, wird sicherlich Attribute wie `color="#F0D398"` kennen. Das sind einfach 3 Hexadezimal Werte. Der erste Wert ist der Rot-, der zweite der Grün- und der dritte der Blauanteil der Farbe. Die Buchstaben A bis F sind symbolisch für die Werte 10 bis 15.
Nehmen wir die Zahl 18A2h:

```
  2 * 16^0 (= 2)
+ A * 16^1 (= 160)
+ 8 * 16^2 (= 2048)
+ 1 * 16^3 (= 4096)
-------------------
  6306d
```

Vom Binär- ins Dezimalsystem

Oft ist es nützlich, Werte von einem Stellensystem ins andere zu konvertieren.

```
addb a, 0b0101
```

Hier wird der Binär Wert 0101b zu Register A dazu addiert. Für uns verständlicher ist jedoch der Dezimalwert 5.

Generell habe ich ja bereits gezeigt, wie man die Binär Werte in Dezimalwerte umrechnet, da aber Zeit kostbar ist, und das hier jeder kann, gibt es einen Trick:

Zuerst muss man die 2er Potenzen auswendig lernen, was sowieso jeder Informatiker können sollte. Da wären 0, 1, 2, 4, 8, 16, 32 ... bis 1024 ein guter Einstieg.

Nehmen wir jetzt den Binär Wert 11111011b. Ziemlich aufwendig hier 7 2er Potenzen zu addieren. Gehen wir jedoch den komfortableren Weg, und zählen erst einmal die Anzahl der Stellen - in dem Fall also 8 - und ignorieren die Null. 11111111b dieser Binär Wert entspricht 2^8-1, also 256-1 = 255. Weiterhin schauen wir, an welcher Stelle die 0 steht(von rechts aus). Das ist die Stelle 2. 2^2 = 4. Jetzt ziehen wir von 255 den Wert 4 ab, und erhalten 251. Der gewünschte Wert ist somit 251. Das ganze macht natürlich nur Sinn, wenn mehr Einsen als Nullen vorhanden sind. Für alle Faulen - und ja, Programmierer sind die faulsten Menschen überhaupt - gibt es da immer noch den Windows Rechner in der wissenschaftlichen Ansicht, der mit 4 Stellensystem umgehen kann.

Vom Binär- ins Hexadezimalsystem

Es sei 10011101b gegeben. Wie bereits erwähnt, lässt sich ein Nennwert eines Hexadezimal Wertes in eine Tetrade unterbringen. Eine Tetrade besteht aus 4 Bit. Gegeben waren 8 Bit. Somit ist es ein Leichtes, diese zu konvertieren.

```
1001b = 9 -> 9h
1101b = 13 -> Dh
```

Zusammengefasst also 9Dh als Hexadezimal Wert.

Vom Hexadezimal- ins Dezimalsystem

Leider kann ich hier nicht mit tollen Tricks glänzen, jedoch sollte man auch hier ein paar 16er Potenzen im Kopf haben. Es sei 0, 1, 16, 256 bis 4096 als ausreichend zu betrachten. Im Grunde habe ich es schon oben in der Erklärung verraten, wie man ein Hexadezimal Wert in einen Dezimalwert konvertiert. Weil es aber so viel Spaß macht(oder auch nicht), hier noch ein weiteres Beispiel 2DEh:

```
  E -> 14 * 16^0 (= 14)
+ D -> 13 * 16^1 (= 208)
+ 2 ->  2 * 16^2 (= 512)
------------------------
  734
```

Vom Hexadezimal- ins Binärsystem

Ähnlich wie beim Konvertieren vom Binär- ins Hexadezimalsystem, braucht man nur die

Nennwerte des Hexadezimal Wertes in 4 Bit Tetraden zu zerlegen. Beispiel sei A16Dh:

```
A -> 10 -> 1010
1 ->  1 -> 0001
6 ->  6 -> 0110
D -> 13 -> 1101
```

ist zusammengefasst der Binär Wert 1010000101101101b.

Spätestens hier sollte die Spreu vom Weizen getrennt sein. Die Stellensysteme zu verstehen ist wichtig, deswegen mit einem Blatt Papier und einem Stift bewaffnet, eigene Beispiele durchrechnen und konvertieren!

Wer sich nach dieser trockenen Theorie immer noch nicht abschrecken lies, der ist auf den richtigen Weg, sein erstes Spiel auf dem PMini zum laufen zu bringen. Es steht aber noch harte Arbeit vor uns. Jetzt geht es zum Mittelpunkt des PMinis - jedoch nicht ganz so spannend, wie mit Jules Verne.

Register - Speichermedien ohne Größenwahn

Register sind Speicherbereiche, die fest verdrahtet sind mit der CPU.

Jedoch besitzt auch unser PMini einen RAM. Man wird sich hier jedoch fragen, warum extra Speicher, wenn doch genügend RAM zur Verfügung steht. Die Antwort lautet Geschwindigkeit! Die Zugriffszeiten auf Register sind wesentlichen kürzer als die auf den RAM. Beschränkt sind diese jedoch auf wenige Bit Speichergröße.

Mit Register wird gerechnet, gezählt, Werte verglichen, Adressen berechnet usw. Sie sind praktisch unsere Hauptakteure.

Welche Register gibt es?

* 8 Bit: A, B, L, H, U, V, F, E, I, N, XI, YI
* 16 Bit: PC, SP, BA, HL, X, Y

Der ganze Buchstabensalat hat natürlich auch eine Bedeutung. Die 8 Bit Register A (=Akkumulator) und B sind für die ALU - die Arithmetic Logic Unit, gedacht. Das heißt, das größtenteils mit ihnen Berechnungen, Vergleiche usw. durchgeführt werden. Register B wird zudem oft als Zählregister für Zählschleifen verwendet. Zusammengefasst ergeben beide das 16 Bit Register BA, für das es eigene Befehle gibt.

Die Register L(= Low) und H(= High), die ebenfalls als 16 Bit Register HL aufgefasst werden, sind generelle Register, die dem Programmierer frei zur Verfügung stehen, können aber mit Register I zusammen eine Adresse bilden.

Das Register F(= Flag) speichert die Zustände der CPU ab. Hier wird gesichert, ob es z. B. zu einen Überlauf beim Addieren gekommen ist. Auch beim Vergleichen von 2 Werten kann hier ersehen werden, in welchen Verhältnis die beiden Werte zueinander stehen. Es gibt so genannte Branchbefehle, die nur zu einer Adresse springen, wenn ein bestimmtes Flag gesetzt ist. Somit kann man hiermit bedingte Verzweigungen in seinen Code einbringen.

Die Register HL und I(= Index) können zu einer 24 Bit Adresse zusammengefasst

werden. Ebenfalls gibt es einen Adressierungsmodus, welcher die Adresse aus Register I und Register N, zusammensetzt.

Register N(= Indexing) wird in Kombination mit Register I für einen Adressmodus verwendet.

Die Register X und Y lassen sich problemlos mit PMAs programmieren. Sie sind dort als X1 und X2 benannt. Man behandelt jedoch nicht den Indexregister I explizit.

Das PC(= Program Cursor) Register beinhaltet die Adresse des Befehls, der als nächstes ausgeführt werden soll. Man kann ihn nur auslesen, in dem man ihn ins Register BA oder HL kopiert. Das direkte setzen der Adresse ist nur mit den Jump oder Branch Befehlen möglich. Ist das höchste Bit, das so genannte MSB(= Most Significant Bit) gesetzt, so geht er in den erweiterten 23 Bit Modus. Davon werden die niedrigsten 8 Bit in das Register V geschrieben. Wenn ein Sprung erfolgt, wird das Register V im Register U vermerkt.

Das SP(= Stack Pointer) Register zeigt auf den aktuellen Stackeintrag. Später werden über ihn z. B. lokale Parameter adressiert.

Ein großes Spektrum an Register mit unterschiedlichen Aufgaben - manche werden wir aber gar nicht einsetzen. Hier seien die Register PC, V und U genannt. Weiterhin erlaubt zwar der Befehlssatz des Prozessors(auch des Emulators) den Umgang mit Register I, jedoch unser Assembler kennt ihn nicht, was ein Arbeiten mit ihm unmöglich macht. Wir merken uns, das Register X und Y einfach X1 und X2 genannt werden.

Das Flagregister

Wie wir wissen, ist es 8 Bit groß, und jedes Bit hat seine eigene Bedeutung. Generell kann man das Flagregister auch als Zustandsregister bezeichnen. Wie ist es nun aufgebaut?:

```
  7    6    5    4     3    2   1   0
+----+----+----+-----+---+---+---+---+
| IB | ID | LM | BCD | S | O | C | Z |
+----+----+----+-----+---+---+---+---+
```

* IB: Interrupt Branch Flag, wird von der CPU gesetzt, wenn ein Interrupt Request (= IRQ) durchgeführt worden ist.
* ID: Interrupt Disable Flag. Ist dieses Bit gesetzt können keine Interrupts aufgeführt werden.
* LM: Low Mask Flag wird gesetzt, wenn ein zu vergleichender Wert kleiner war.
* BCD: Binary Codet Digit. Bei gesetzten Bit, wird die CPU im so genannten BCD Mode gesetzt. In diesem Modus wird für eine Ziffer des Dezimalsystems 1 Byte verwendet - es findet jedoch kaum Anwendung.
* S: Signet Flag wird gesetzt, wenn das Ergebnis einer Instruktion negativ war.
* O: Overflow Flag wird gesetzt, wenn das Ergebnis einer Instruktionen einen Überlauf produziert hat.
* C: Carry Flag wird gesetzt, wenn das Ergebnis einer Instruktion zu einem Übertrag geführt hat.
* Z: Zero Flag wird gesetzt, wenn das Ergebnis einer Instruktion Null ergeben hat.

Bits auf großer Reise - Move als Reiseleiter

Klar dürfte sein, das wir Werte in Register schreiben, mit ihnen Arbeiten und zurückschreiben müssen - das gute alte Eingabe-Verarbeiten-Ausgabe(kurz EVA) - Prinzip. Nehmen wir das Beispiel nochmal von ganz weit oben mit dem Bewegen des Helden nach rechts. Die Variable, sprich Speicheradresse im RAM, hat die X Position des Helden gespeichert. Jetzt muss man diese X Position ins Register A bekommen, mit *INC A* z. B. um 1 erhöhen, und dann wieder zurück in die Variable schreiben. Dafür gibt es eine Menge Move-Instruktionen für verschiedene Adressmodi.

Welche der PMAs versteht, steht in der *mindx.txt*. Welche der PMini versteht, findet man unter:

* sublab.net/pokemini/instructions.html
* sublab.net/pokemini/instructionsce.html
* sublab.net/pokemini/instructionscf.html

Ich werde hier nur auf die wichtigsten Adressmodi eingehen, den Rest kann man selber nachschlagen.

Erst einmal kann man den Wert eines beliebigen 8 Bit Registers einem anderen 8 Bit Register zuweisen. Z. B. ist es möglich, dem Register A den Wert von Register L zuzuweisen.

```
mov a, l
```

Wichtig ist, dass bei Move das Ziel immer das erste und die Quelle immer das zweite Argument ist. Auch wenn man instinktiv meinen würde, der Wert wird von Register L nach Register A verschoben, ist es ein Kopieren des Wertes. Aber diese Unlogik gibt es bei fast allen Assemblern.

Man muss jedoch erst einmal einen Wert in Register L hinein bekommen, um ihn dann Register A zuweisen zu können. Das Zuweisen eines festen Wertes nennt man unmittelbares Adressieren(ist in dem Sinne keine Adressierungsart).

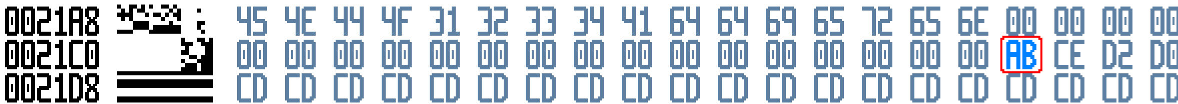
```
mov l, 123
```

Hier wird der Dezimalwert 123 unmittelbar dem Register L zugewiesen. Das geht wieder mit allen 8 Bit Registern.

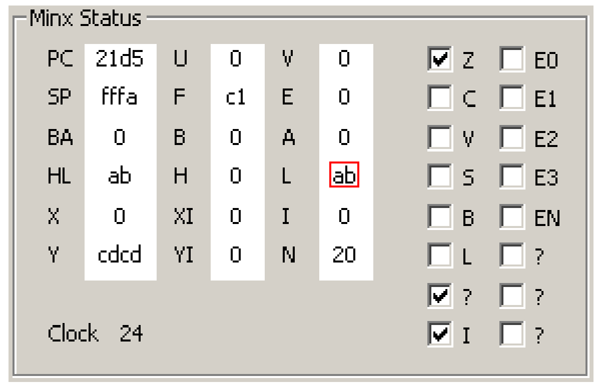
Bei der absoluten Adressierung wird dem Register der Wert an der angegebenen Speicheradresse zugewiesen. Wenn ein Zugriff auf eine Adresse erfolgt, geschieht dies immer in eckigen Klammern [Adresse]. Z. B. `[0x123456]` oder auch `[X1+L]`.

```
heldx:
     .db 0xAB

main:
     mov l, [heldx]

end:
     jmp end
```

Das so genannte Label *heldx* wird durch den Assembler durch die Adresse 0x21D0 ersetzt, und per `.DB 0xAB` mit dem Hexadezimal Wert ABh belegt:



Wie zu sehen, steht an der Adresse 21D0h der Hexadezimal Wert ABh. Nach ausführen der Move-Instruktion hat Register L ebenfalls den Wert an der Adresse 0x21D0 steht, also ABh:



Auch Register selbst können Adressen beinhalten bei dieser Adressierungsart. Beispielsweise weist man mit `MOV A, [HL]` den Wert der an Speicheradresse, die in HL steht, dem Register A zu. HL wird gedanklich mit seinem Wert ersetzt. Z. B. 1234h Somit ergibt dies `MOV A, [0x1234]`.

Es folgt die absolute, mit Register indizierte, Adressierung. Man hat eine absolute Adresse und kombiniert diese mit einem Wert aus einem Register. Hier sei [X1+ofs8] oder auch [X2+L] erwähnt.

```
mov x1, 0x1234
mov l, [x1+0x56]
```

In Register X steht nun der Wert 1234h. Jetzt wird in der nächsten Zeile diese Adresse mit dem Wert 56h kombiniert. Es wird folglich den Register L der Wert an Adresse 0x123456 zugewiesen.

Als weiteres Beispiel wäre noch

```
mov x1, 0x1234
mov l, 0x56
mov a, [x1+l]
```

Hier wird Register A der Wert an Adresse 0x123456 zugewiesen.

Zu guter Letzt noch einmal das Beispiel mit unseren Helden.

```
.orgfill 0x1460
heldx:
```

```
.orgfill 0x02100
     .db "MN"
     jmp main

.orgfill 0x021A4
     .db "NINTENDO"
     .db "1234"
     .db "Held"

.orgfill 0x021BC
     .db "2P"

.orgfill 0x021D0
startx:
     .db 12

.align 16
main:
     ; lade Startposition in heldx
     mov a, [startx] ; A = Startposition
     mov [heldx], a  ; heldx = A

     ; erhöhe Startposition
     mov a, [heldx] ; A = heldx
     inc a          ; A = A+1
     mov [heldx], a ; heldx = A

end:
     jmp end

.end
```

Was die ganzen Adressen und auch die Präprozessor-Direktive `ALIGN` zu bedeuten haben, komme ich später darauf zu sprechen.

Labels

Hier seien auch die Labels erwähnt. Eingesetzt haben wir diese schon bei z. B. *main* und *end*. Unverkennbar ist der Suffix von Labels, oder auch Sprungmarken genannt - ein Doppelpunkt. Sprungmarken sind beim BASIC Programmieren die Übeltäter, die zu so genannten Spaghetticode führen. Bei neuen BASIC Dialekten ist so etwas jedoch auch schon verpönt(BlitzBASIC z. B.). Beim Assembler-Programmieren sorgen sie jedoch für die nötige Übersicht, und sind somit unverzichtbar. Diese Sprungmarken werden über den Assembler durch Adressen ersetzt. Wird ein Sprung zu einem Label durchgeführt, wird eigentlich ein Sprung zu einer Adresse gemacht. Laden wir einmal keine Symboldatei in den Debugger, werden wir auch erkennen, dass z. B. `JMP END` durch `JMP 0x21DC` ersetzt wird.

Grundlagen des PMAs

Da wären zum einen die Kommentare. Diese werden mit einem Semikolon eingeleitet. Assemblercode ist zwar einfach zu verstehen, den Zusammenhang daraus abzuleiten ist

jedoch schwer. Hat man ein großes Spiel geschrieben, erstreckt sich oft der Code über tausende Zeilen. Da Übersicht zu gewährleisten, schafft man nur über Kommentare. Sie dokumentieren den Code, sollten jedoch nicht die Befehle selbst erläutern. Kommentare gehen beim Assemblieren jedoch verloren, da sie nicht zur Ausführung des Programms nötig sind. So kann man keine Kommentare in Minimon einsehen, so kann man auch keine per PMDis zurückgewinnen.

Auch der PMAs kann mit verschiedenen Stellensystem umgehen. Da wären das Dezimalsystem, das Binärsystem und das Hexadezimalsystem. Diese muss er durch Präfixe unterscheiden können:

* 0x oder $ steht für einen Hexadezimal Wert z. B. 0x1234
* 0b für einen Binär Wert z. B. 0b1110101
* Gibt man keinen Präfix an, so wird das Dezimalsystem verwendet z. B. 1234.

Staubtrockene Theorie - die ALU

Auch der Prozessor(nach Neumann Architektur) arbeitet nach dem EVA Prinzip. Dieses Prinzip besteht beim Prozessor aus 5 verschiedenen Phasen:

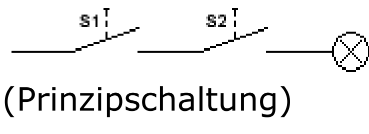
* Fetch Phase: Der Befehl wird geladen (anhand des PC Registers).
* Decode Phase: Der Befehl wird entschlüsselt (Unterscheiden zwischen arithemtischer, logischer oder Sprunginstruktion)
* Load Phase: Die Operanden werden gegebenenfalls geladen (bei `ADD A, 10` wird die 10 als Operand geladen).
* Execute: Der eigentliche Befehl wird ausgeführt.
* Write Back: Das Ergebnis wird zurück geschrieben.

Bei den arithmetischen und logischen Instruktionen kommt die ALU zum Einsatz. Sie kann auf und runter zählen, addieren, subtrahieren, multiplizieren, dividieren, Bits schieben und rotieren, logisch über UND, ODER und Exklusiv ODER verknüpfen sowie Vergleiche durchführen. All diese Funktionen lassen sich auf die logischen Funktionen UND, ODER und NICHT zurückführen. Diese werden mit Hilfe von Transistoren gebildet.

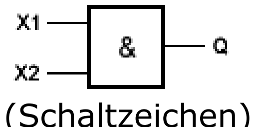
Logische Verknüpfungen

Bitzustände werden hier als Wahrheitszustände angesehen. Strom fließt - logisch 1 bzw. WAHR, Strom fließt nicht - logisch 0 bzw. FALSCH. Hier ist es z. B. möglich zu prüfen, ob der Benutzer die Feuertaste gedrückt hat UND ob noch genug Munition vorhanden ist. Nur dann kann der Benutzer einen Schuss abfeuern.

Das logische UND



(Prinzipschaltung)



(Schaltzeichen)

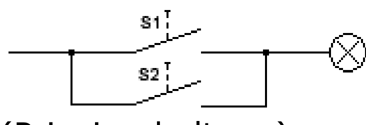
Die Lampe wird nur aufleuchten, wenn Taster S1 UND Taster S2 gedrückt sind. Bei uns wird der Ausgang Q nur dann auf logisch 1 gesetzt, wenn die Eingänge X1 UND X2 auf logisch 1 gesetzt sind. Ein Register besitzt jedoch mehrere Bits. Deswegen benutzt z. B. die ALU beim Befehl `AND A, imm8` genau 8 Logikgatter.

```
mov a, 0b01010101
and a, 0b11010011
```

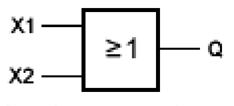
Beim letzten Befehl wird der Inhalt von Register A mit dem unmittelbaren Wert bitweise UND verknüpft und das Ergebnis in Register A zurückgeschrieben. Im Ergebnis sind nur da die Bits auf logisch 1 gesetzt, wo sie vorher im Register A UND im unmittelbaren Wert gesetzt waren.

```
    01010101
AND 11010011
    --------
    01010001
```

Das logische ODER



(Prinzipschaltung)



(Schaltzeichen)

Hier leuchtet die Lampe auf, wenn entweder Taster S1 ODER Taster S2 betätigt wird. Das ganze gilt bei uns wieder so, das Q auf logisch 1 gesetzt wird, wenn entweder Eingang X1 ODER X2 auf logisch 1 steht.

```
mov a, 0b01010101
or  a, 0b11010011
```

Wie auch bei AND, sind hier mehrere Logikgatter in der ALU tätig.

```
   01010101
OR 11010011
   --------
   11010111
```

Das logische Exklusiv ODER(XOR)

(Prinzipschaltung)

(Schaltzeichen)

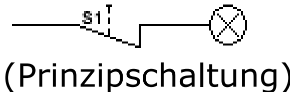
Zu den Namen kam es, weil es wie das logische ODER funktioniert, jedoch nicht die Schaltzustände des logischen UND beinhaltet. Sie sind exklusiv. Beim normalen ODER spricht man daher auch von Inklusiv ODER, da es die Schaltzustände vom logischen UND inklusive behandelt. Deswegen spricht man hier von Antivalenz. Q wird nur dann auf logisch 1 gesetzt, wenn die Signale der beiden Eingänge X1 und X2 verschieden von einander - also antivalent(Gegenteil wäre äquivalent) sind.

```
mov a, 0b01010101
xor a, 0b11010011
```

Wie auch bei `AND` und `OR`, stehen hier wieder mehrere Logikgatter zum Einsatz.

```
    00010101
XOR 11010011
    --------
    11000110
```

Das logische NICHT



(Prinzipschaltung)

(Schaltzeichen)

Das logisches NICHT invertiert den Eingang X. Die Lampe leuchtet nur dann, wenn der Taster(genauer: ein Öffner) nicht betätigt wird. Am Ausgang Q liegt nur dann logisch 1 an, wenn an Eingang X logisch 0 anliegt.

```
mov a, 0b01010101
not a
```

Alles Bits werden invertiert. Aus 1 wird 0, und aus 0 wird 1:

```
NOT 01010101 =
    10101010
```

Zur Zusammenfassung hier eine Wahrheitswerttabelle:

p	q	w(p UND q)	w(p ODER q)	w(p XOR q)	w(NICHT p)
F	F	F	F	F	W
F	W	F	W	W	W
W	F	F	W	W	F
W	W	W	W	F	F

(p und q = Operanden, w = Wahrheitswert, W = wahr, F = falsch)

Rechnen mit dem PMini

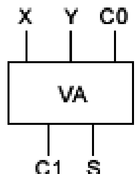
Ein Computer heißt nicht umsonst Computer, denn das ist seine Spezialität. Die ALU beschränkt sich jedoch hier auf die Grundrechenarten.

In unserem PC gibt es noch einen Coprozessor, die so genannte FPU (= Floating Point Unit), die auch mit reellen Werten rechnen kann und Funktionen zum berechnen von Sinus, Logarithmus usw. bereitstellt.

Wir sind also darauf beschränkt, mit Integer Werten zu rechnen. Die ALU des PMini kann Addieren, Subtrahieren, Multiplizieren und Dividieren.

Addieren

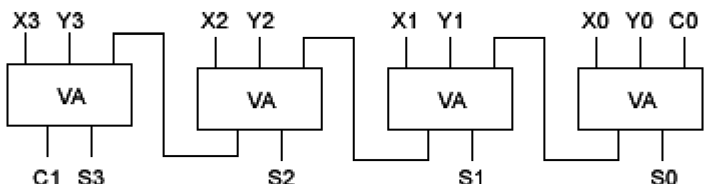
Das Addieren zweier Werte erfolgt durch so genannte Volladdierer. Als Eingang haben diese die beiden Summanden X und Y sowie den Übertrag C0. Als Ausgang gibt es dann S für die Summe und C1 für den berechneten Übertrag. C steht übrigens für Carry, dessen Begriff wir auch im Flagregister wiederfinden.




```
S = (X XOR Y) XOR C0
C1 = (X OR Y) AND (C0 OR X) AND (CO OR Y)
```

```
 X | Y | C0 | S | C1
---+---+----+---+---
 0 | 0 | 0  | 0 | 0
 0 | 0 | 1  | 1 | 0
 0 | 1 | 0  | 1 | 0
 0 | 1 | 1  | 0 | 1
 1 | 0 | 0  | 1 | 0
 1 | 0 | 1  | 0 | 1
 1 | 1 | 0  | 0 | 1
 1 | 1 | 1  | 1 | 1
```

In der ALU sind dann mehrere Volladdierer in Reihe geschaltet. Der Übertrag wird von rechts nach links immer weiter gereicht.



Diese Skizze ist repräsentativ für einen 4 Bit Addierer. In unserer ALU gibt es einen 16 Bit Addierer.

Jetzt muss man 2 Instruktionen von einander unterscheiden. `ADD` und `ADDC`. Letzteres

steht für Add with Carry. Ersteres setzt C0 auf logisch 0, Letzteres nimmt den Wert aus dem Flagregister. Beide Instruktionen speichern C1 in das Flagregister.

Nehmen wir z. B. die Addition 54 + 21. Binär ausgedrückt wäre dies 00110110b + 00010101b:

```
mov a, 0b00110110
add a, 0b00010101
```

In Register A steht dann 0x4B bzw. 75, das korrekte Ergebnis der Addition.

Rechnen wir das ganze noch einmal selber schriftlich:

```
  00110110
+ 00010101
  --------
  01001011
```

Auseinander genommen von rechts nach links:

```
0 + 1            = 1
1 + 0            = 1
1 + 1            = 0 -> Übertrag
0 + 0 + Übertrag = 1
1 + 1            = 0 -> Übertrag
1 + 0 + Übertrag = 0 -> Übertrag
0 + 0 + Übertrag = 1
0 + 0            = 0
0 + 0            = 0
```

Und ja, 001001011b ist 75 dezimal.

Jetzt müssen wir jedoch festhalten, das wir hier mit Vorzeichen behafteten Werten rechnen. Im Register A können wir nur Werte zwischen -128 bis +127 unterbringen. Das Vorzeichen wird im MSB gespeichert. Ist dies gesetzt, ist der Wert negativ. Die restlichen 7 Bit geben den absoluten Wert vor. Ist das Vorzeichen ein Minus, so muss man das 2er Komplement (alle Bits invertieren und 1 dazu addieren) vom absoluten Wert bilden, um den "richtigen" Wert zu erhalten.

Beispiele:

* 00001011b -> positives Vorzeichen, 0001011b absoluter Wert -> +11
* 01111111b -> positives Vorzeichen, 1111111b absoluter Wert -> +127
* 00000000b -> positives Vorzeichen, 0000000b absoluter Wert -> +0
* 10110111b -> negatives Vorzeichen, 0110111b absoluter Wert, 1001000b invertiert, 1001001b 2er Komplement -> -73
* 10011010b -> negatives Vorzeichen, 0011010b absoluter Wert, 1100101b invertiert, 1100110b 2er Komplement -> -102
* 11111111b -> negatives Vorzeichen, 1111111b absoluter Wert, 0000000b

invertiert, 0000001b 2er Komplement -> -1

* 10000000b -> negatives Vorzeichen, 0000000b absoluter Wert, 1111111b invertiert, 10000000b 2er Komplement -> -128

Das ganze erlaubt jedoch einen Trick. Wir können gleichzeitig auch mit Vorzeichen losen Werten von 0 bis 255 rechnen. Dafür dürfen wir nicht auf die Bedeutung des Vorzeichenbits eingehen, sondern die vollen 8 Bit als absoluten positiven Wert ansehen. Es lässt sich z. B. so auch 150+70 rechnen:

```
mov a, 150
add a, 70
```

Im Register A steht dann der Wert 11011100b bzw. 220 ohne Interpretation des Vorzeichenbits. Das Sign Flag wird, wie wir gleich sehen, jedoch mit gesetzt, obwohl das Ergebnis in unseren Sinne gar nicht negativ ist.

Wann wird welches Flag gesetzt?

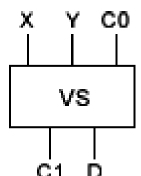
```
add a, b
```

Ergebnis = A+B(+ Carry)

* Zero Flag, wenn Ergebnis = 0 (Wenn Ergebnis AND FFh = 0)
* Sign Flag, wenn das höchste Bit des Ergebnis 1 ist (Wenn Ergebnis AND 80h = 80h)
* Carry, wenn Übertrag entstanden ist (Wenn Ergebnis AND (NOT FFh) ungleich 0)
* Overflow, wenn Überlauf oder Unterlauf eingetreten ist(Wenn ((A XOR B) AND 80h) und ((A XOR Ergebnis) AND 80h))

Subtraktion

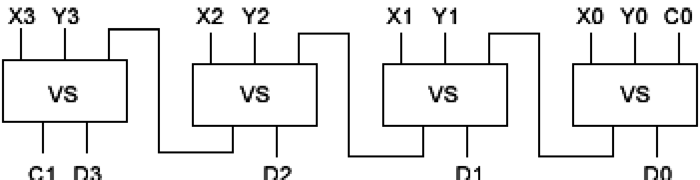
Zum Subtrahieren gibt es so genannte Vollsubtrahierer:




```
D = (X XOR Y) XOR C0
C1 = (X AND (NOT Y)) OR (C0 AND X) OR (C0 AND (NOT Y))
```

```
 X | Y | C0 | D | C1
---+---+----+---+---
 0 | 0 | 0  | 0 | 0
 0 | 0 | 1  | 1 | 1
 0 | 1 | 0  | 1 | 1
 0 | 1 | 1  | 0 | 1
 1 | 0 | 0  | 1 | 0
 1 | 0 | 1  | 0 | 0
 1 | 1 | 0  | 0 | 0
 1 | 1 | 1  | 1 | 1
```

Ebenfalls werden diese in Reihe geschaltet:



Wieder repräsentativ für einen 4 Bit Subtrahierer. Unsere ALU hat einen 16 Bit Subtrahierer.

Wer hätte es gedacht? Auch hier gibt es wieder 2 Instruktionen. Einmal `SUB` und `SUBC`. Funktionsweise wie bei den zusammengeschalteten Vollsubtrahierern.

Nehmen wir als Beispiel 74-23 also 01001010b - 00010111b

```
mov a, 0b01001010
sub a, 0b00010111
```

Lassen wir das ausführen, steht dann in Register A der Wert 33h also 51 dezimal. Das ganze noch einmal schriftlich subtrahiert:

```
  01001010
- 00010111
  --------
  00110011
```

Wieder sind die Spalten, wo Überträge entstehen, blau markiert. Das ganze im Detail vorgerechnet(von rechts nach links):

```
0 - 1            = 1 -> Übertrag
1 - 1 - Übertrag = 1 -> Übertrag
0 - 1 - Übertrag = 0 -> Übertrag
1 - 0 - Übertrag = 0
0 - 1            = 1 -> Übertrag
0 - 0 - Übertrag = 1 -> Übertrag
1 - 0 - Übertrag = 0
0 - 0            = 0
```

Jetzt kann es jedoch vorkommen, das man z. B. 5 - 12 rechnet was ja -7 ergibt.

```
  00000101
- 00001100
  --------
  11111001
```

Auch in der letzten Spalte gibt es einen Überlauf. Dieser ist C1 in unserer Schaltung. Bei ihm wird auch das Sign Bit im Flagregister gesetzt. Es kennzeichnet, dass das berechnete Ergebnis negativ ist. Auch hier wird das MSB als Vorzeichenbit erkannt. Also: 11111001b -> negatives Vorzeichen, 1111001b absoluter Wert, 0000110b invertierter Wert, 0000111b 2er Komplement -> -7

Wann wird welches Flag gesetzt?

```
sub a, b
```

Ergebnis = A-B bzw. Ergebnis = A-(B+Carry)

 * Zero Flag, wenn Ergebnis = 0 (Wenn Ergebnis AND FFh = 0)
 * Sign Flag, wenn das höchste Bit des Ergebnis 1 ist (Wenn Ergebnis AND 80h = 80h)
 * Carry, wenn Übertrag entstanden ist (Wenn Ergebnis AND (NOT FFh) ungleich 0)
 * Overflow, wenn Überlauf oder Unterlauf eingetreten ist(Wenn ((A XOR B) AND 80h) und ((A XOR Ergebnis) AND 80h))

Multiplikation

Multipliziert können nur die beiden 8 Bit Register L und A werden und das Ergebnis wird im 16 Bit Register HL abgelegt. Ich erspare mir hier den Schaltplan für Multiplikation zu zeigen, da dieser Rund eine Seite einnimmt für gerade mal 4 Bit, stattdessen werde ich anhand der schriftlichen Multiplikation das ganze erläutern.

Nehmen wir 15 * 41 also 00001111b * 00101001b

```
00001111 * 00101001
-------------------
    00000000
  +  00000000
  +   00001111
  +    00000000
  +     00001111
  +      00000000
  +       00000000
  +        00001111
-------------------
         1001100111
```

Das Ergebnis ist 1001100111b bzw. 615 dezimal. Die ALU macht sich hier einfach nur das kleine 1*1 zu Nutzen. Die zwischen Ergebnisse werden dann mit Hilfe von Schiebeoperatoren immer weiter nach rechts verschoben und miteinander addiert. Das selbe Beispiel als Code umgesetzt:

```
mov l, 0b00001111
mov a, 0b00101001
mul l, a
```

Nach dem Ausführen steht dann im Register HL der Wert 0x267 bzw. 615 dezimal.

Es wird hier nicht auf das Vorzeichen Rücksicht genommen. -128 wird z. B. als 0 interpretiert.

Wann wird welches Flag gesetzt?

```
mul l, a
```

 * Zero Flag, wenn Register HL = 0
 * Sign Flag, wenn das höchste Bit des Regsiter HL gesetzt ist (Wenn HL AND 8000h =

8000h)
    * Carry, nie
    * Overflow, nie

Division

Auch hier ist man einer Beschränkung unterlegen. Man kann nur das 16 Bit Register HL durch das 8 Bit Register A dividieren. Das Ergebnis wird wieder in Register HL abgelegt. Auch hier wäre es sinnlos den Schaltplan zu zeigen. Er hat ähnliche Ausmaße wie der, der Multiplikation. Deswegen wieder schriftliche Division, der Rechner macht auch nichts anderes.

Als Beispiel dient uns 23 / 3 was 7 Rest 2 ist. 00010111b / 00000011b

```
  00101111 / 00000011 = 00000111 Rest 00000010
- 00001100
  --------
  00001011
- 00000110
  --------
  00000101
- 00000011
  --------
  00000010
```

So ähnlich macht das auch die ALU. Hier der Algorithmus dafür:

    * Der Divisor wird solange nach links verschoben, bis eine 1 an höchster Stelle steht. Die Anzahl, die hierfür benötigt wird, + 1 wird als *N* gespeichert.
    * Ist Dividend >= Divisor, dann Dividend = Dividend – Devisor. Em Ergebnis wird eine 1 hinzugefügt. Ist dies nicht der Fall, so wird eine 0 ins Ergebnis hinzugefügt; Anschließend wird der Divisor um eine stelle nach rechts geschoben. Das ganze wiederholt sich *N* mal.
    * Der Rest ist der aktuelle Dividend

Angewandt auf unser Beispiel:

    * 00000011 muss 6 mal nach links geschoben werden, bis an höchster Stelle eine 1 steht. Also Devisor = 11000000 und N = 6+1

    * (Durchgang 1)
Ist 00010111 >= 11000000? Nein
`-> 0`
11000000 >> 1 = 01100000

(Durchgang 2)
Ist 00010111 >= 01100000? Nein
`-> 00`
01100000 >> 1 = 00110000

(Durchgang 3)
Ist 00010111 >= 00110000? Nein
`-> 000`

00110000 >> 1 = 00011000

(Durchgang 4)
Ist 00010111 >= 00011000? Nein
```
-> 0000
```
00011000 >> 1 = 00001100

(Durchgang 5)
Ist 00010111 >= 00001100? Ja
```
-> Dividend = 00010111 - 00001100
-> Dividend = 00001011
-> 00001
```
00001100 >> 1 = 00000110

(Durchgang 6)
Ist 00001011 >= 00000110? Ja
```
-> Dividend = 00001011-00000110
-> Dividend = 00000101
-> 000011
```
00000110 >> 1 = 00000011

(Durchgang 7)
Ist 00000101 >= 00000010? Ja
```
-> Dividend = 00000101-00000011
-> Dividend = 00000010
-> 0000111
```

  * Ergebnis = 00000111

Rest = 00000010

Das Ergebnis ist also 111b = 7 dezimal und der Rest beträgt 10b = 2 dezimal. Funktioniert also perfekt. Bei einer Division durch Null kann man in Schritt 1 den Devisor so lange nach links schieben, bis man grau wird. Es wird nie eine 1 an höchster Stelle stehen. Deswegen sind Divisionen durch Null nicht erlaubt, was mathematisch auch korrekt ist.

```
mov hl, 23
mov a, 3
div hl, a
```

Im Register L steht dann das Ergebnis 7 und im Register H der Rest 2. Ebenfalls kann wird das Vorzeichenbit beider Register nicht interpretiert.

Wann wird welches Flag gesetzt?

```
div hl, a
```

Ergebnis = HL/A

- Carry, nie

Wenn Ergebnis nicht mehr in 8 Bit passt(Ergebnis >= 100h)
  - Zero Flag, wenn Register L = 0
  - Sign Flag, immer

- Overflow, immer

Ansonsten

Register H = Rest der Division HL / A
Register L = Ergebnis
- Zero Flag, wenn Register L = 0
- Sign Flag, wenn das höchste Bit von Register L 1 ist (Wenn L AND 80h = 80h)

Grausame Theorie, ich weiß. Es folgt jedoch weiterer Nachschub davon. Jedoch muss man nicht wirklich wissen, was genau in der ALU abgeht.

Das große Bitgeschupse

Leider kommen in unserer ALU keine so genannten Barellshifter zum Einsatz, die es erlauben würden, die Bits gleich um mehrere Stellen in einem einzigen Takt zu verschieben. Wir können somit die Bits nur um eine Stelle nach links oder rechts schieben. Das Bit, was dabei heraus geschoben wird, wird im Carry Flag gespeichert, auf der anderen Seite rutscht eine 0 hinzu.

Wertemäßig ist eine Verschiebung um eine Stelle nach links eine Multiplikation mit 2, eine Verschiebung um eine Stelle nach rechts eine Division um 2. Das Shiften von Bits ist aber wesentlich schneller, als die Instruktionen MUL oder gar DIV.

```
mov a, 0b00101001
shl a
shr a
```

In der 2ten Zeile wird der Wert 00101001b um eine Stelle nach links verschoben. Im Register A steht somit 01010010b. In der 2te Zeile wird der Wert im Register A um eine Stelle nach rechts geschoben. Nun steht wieder 00101001b drin.

Rotieren bis einem schwindelig wird

Dies ist eine Eigenart des Bitshiftings. Hier wird nicht ein Bit verworfen, und es rückt auch keine Null nach, sondern das verworfene Bit wird auf der anderen Seite nachgerückt.

```
mov a, 0b00101001
ror a
rol a
```

In der 2ten Zeile wird 00101001b nach rechts verschoben. Dabei wird die rechte 1 gesichert und an der linke Seite wieder eingefügt. Es steht somit der Wert 10010100b im Register A. In der 3ten Zeile wird dann 10010100b wieder nach links rotiert. Dabei fällt die linke 1 heraus und wird rechts angehangen. Danach steht der Wert 00101001b wieder im Register A.

Der ganze Spaß lässt sich auch mit dem Carry Flag im Flagregister verknüpfen. Die Befehle hierfür heißen "RORC" und "ROLC". Das Bit, was nachrückt, wird direkt aus dem Carry Flag bezogen. Anschließend wird das Bit, was verworfen wird, in das Carryflag eingetragen.

Sterne zählen für Romantiker

Inkrementieren und Decrementieren - klingt schlimmer als es ist. Inkrementieren und Decrementieren kommen aus dem Lateinischen und stehen für Incrementare - der Zuwachs und Crementare - der Verfall / die Verminderung. Für diese beiden Aufgaben gibt es die Instruktionen `INC` und `DEC` - Increase und Decrease. `INC` erhöht also einen Wert um 1, und `DEC` vermindert einen Wert um 1. Oft wird hier das Zählregister B zum Einsatz kommen.

```
mov b, 1
inc b
inc b
inc b
dec b
```

In der ersten Zeile hat das Register noch den Wert 1. In der zweiten den Wert 2, in der dritten den Wert 3 in der vierten den Wert 4 und in der sechsten den Wert 3.

Hat Register B den Wert 11111111b, und es wird danach die `INC` Instruktion ausgeführt, so hat Register B anschließend den Wert 00000000b. Hat Register B den Wert 00000000b, und es wird danach die `DEC` Instruktion ausgeführt, so hat Register B anschließend den Wert 11111111b.

Es wird übrigens bei beiden Instruktionen nur das Zero Flag beeinflusst. Es wird nur dann gesetzt, wenn der Inhalt von Register B gleich 00000000b ist.

Der Stack

Ja, wie es auch jedes andere Tutorial besagt: man darf sich den Stack als ein Stapel Bücher vorstellen. Meinetwegen auch das ganze mit Bierkästen. Aber Stapel bleibt Stapel. Und was kann man mit einem Stapel machen? Richtig - Stapeln. Das ganze nennt sich bei uns Pushen, und das Herunter nehmen Popen. Und wer hätte es gedacht? Die Instruktionen dafür sind `PUSH` und `POP`.

Das ganze ist wichtig, wenn wir dann Prozeduren bzw. Funktionen kennen lernen. Rücksprungadressen, Übergabeparameter, lokale Variablen und Rückgabe Werte werden auf den Stack gepusht.

Aufgrund das der Stack Pointer nicht immer zu Beginn auf die gewünschte Adresse 0x2000 zeigt (z. B. durch Abarbeiten des BIOS o. ä.), müssen wir das selbst in die Hand nehmen mit der Instruktion `MOV SP, 0x2000`.

Diese Adresse heißt auch Nullelement. Pushen wir ein 2 Byte großen Wert auf den Stack, so wird ERST der Stack Pointer um 2 vermindert, und DANN an diese Adresse der gewünschte Wert geschrieben. In die Adresse 0x2000 wird somit nie ein Wert geschrieben, deswegen heißt es eben Nullelement.

Es lassen sich alle Register und auch ausgewählte Registerpaare(Siehe *mindx.txt*) auf den Stack legen und wieder herunter nehmen.

```
    mov  sp, 0x2000 ; Wichtig!
```

```
mov  x1, 0x1234 ; SP = 0x2000
push x1         ; SP = 0x1FFE
pop  a          ; SP = 0x1FFF, A = 0x34
pop  b          ; SP = 0x2000, B = 0x12
```

Was wird gemacht?

```
MOV SP, 0x2000          PUSH X1                 POP A                   POP B
+-------------+         +-------------+         +-------------+         +-------------+
|    leer     |         |    leer     |         |    leer     |         |    leer     |
+-------------+         +-------------+         +-------------+         +-------------+
|    leer     |         |    leer     |         |    leer     |         |    leer     |
+-------------+         +-------------+         +-------------+         +-------------+
|    leer     |         |    0x34     | <- SP   |    0x34     |         |    0x34     |
+-------------+         +-------------+         +-------------+         +-------------+
|    leer     |         |    0x12     |         |    0x12     | <- SP   |    0x12     |
+-------------+         +-------------+         +-------------+         +-------------+
| NullElement | <- SP   | NullElement |         | NullElement |         | NullElement | <- SP
+-------------+         +-------------+         +-------------+         +-------------+
```

Wie zu sehen ist, wird beim Popen nicht die Stelle mit Null o. ä. überschrieben. Die Werte bleiben so lange im Stack, bis sie wieder durch Pushen überschrieben werden.

Der Stack ist von 0x1420 bis 0x1FFF adressiert. Es bleiben also einem rund 3 KByte, bis es zum so genannten Stackoverflow, also einen Stacküberlauf, kommt. Und das wird in der Regel auch nicht erreicht, wenn man keine großen rekursiven Prozeduren/Funktionen ausführen lässt (später dazu mehr).

Variablen

Natürlich werden uns bald die Register ausgehen, wenn wir Sachen wie Lebenspunkte, Munition etc. in die Register packen. Dafür ist der WorkRAM da. Er ist adressiert von 0x1460 bis 0x1FFF und somit rund 3 KByte klein. Zu den RAM zählt übrigens auch der Stack und der VideoRAM. Jetzt wird auch klar, weswegen man keinen Fehler beim adressieren des Stacks machen sollte. Wenn der Stack Pointer zu Anfang auf 0x2000 zeigt, eine `POP` Instruktion folgt und danach `PUSH` ausgeführt wird, kann eine Variable mit einem falschen Wert überschrieben. Hier ist also Vorsicht geboten.

Der Zugriff auf Variablen ist recht einfach, es sind ja nur Adressen bzw. Labels.

In Hochsprachen gibt es unterschiedliche Datentypen für Variablen, die unterschiedliche Größen oder untersch. Interpretationen besitzen. Die meisten bieten welche für (Vorzeichen behaftete und Vorzeichen lose)Integer-, Float- und Stringvariablen an.

* Integer Werte sind, wir wir wissen, ganzzahlige Werte. Mit ihnen kann die ALU rechnen.
* Floatwerte sind die so genannten Fließkomma Werte, können somit reelle Werte speichern. Sie bestehen aus Vorzeichenbit, Mantisse sowie Exponent. Das Rechnen mit ihnen müsste manuell erfolgen, da wir keine FPU zur Verfügung haben. Eine etwas einfachere Lösung bieten Festkomma Werte.
* Strings sind Zeichenketten. Jedes Zeichen besteht hier aus einem Byte. Es lassen sich somit 256 verschiedene Zeichen unterbringen. Pro Zeichen gibt es einen eindeutigen ASCII-Code. So hat z. B. das große A den ASCII-Code 65. Ebenfalls beinhaltet der ASCII-Zeichensatz Sonderzeichen wie beispielsweise "ö", "ß", "ê" etc. und Steuerzeichen, die Zeilenumbrüche, Tabulatoren usw. kennzeichnen. Es gibt verschiedene Arten von Strings. Bei den *C-Strings* wird die Zeichenkette + Nullzeichen

gespeichert. Das ist platzsparend, jedoch sind Operationen wie Zusammenfügen usw. Perfomance lastig. Eine gute alternative sind *Pascal-Strings*. Pro String werden hier 256 Byte reserviert. Im ersten Byte steht die Länge des Strings, und in den restlichen 255 die Zeichenkette selber. Hier sind Operationen einfach durchzuführen, sie sind jedoch auf 255 Zeichen max. beschränkt(reicht in der Regel aber aus).

```
.orgfill 0x1460
varx:   .db 0
vary:   .db 0
varz:   .db 0

.orgfill 0x02100
     .db "MN"
     jmp main

.orgfill 0x021A4
     .db "NINTENDO"
     .db "1234"
     .db "Variablen"

.orgfill 0x021BC
     .db "2P"

.orgfill 0x021D0
main:
     ; varx = 1
     mov a, 1
     mov [varx], a

     ; vary = 2
     mov a, 2
     mov [vary], a

     ; varz = varx + vary
     mov a, [varx]
     add a, [vary]
     mov [varz], a

     jmp end

end:
     jmp end

.end
```

Im Grunde nichts neues.

Unbedingte Sprünge

Stellen wir uns vor, wir haben ein Spiel geschaffen, das über mehrere tausend Zeilen Code geht. Dürfte unübersichtlich werden, alle Programmteile nacheinander ablaufen zu lassen. Besser wäre es, verschiedene Aufgaben in Teilabschnitte zu gliedern. Z. B. das Laden und Speichern von Spielständern, die Eingabeverarbeitung, Kollisionsabfragen usw. Und da sind uns unbedingte Sprünge hilfreich. Wir springen einfach die Subroutinen ohne Bedingung an (das Abarbeiten der Eingabe z. B. ist ja ein Muss). Das

geschieht über den Befehl `JMP` abgeleitet von *Jump*. Er macht das softwaremäßig, was wir beim Ausprobieren manuell immer gemacht haben - den Program Cursor mit einer neuen Adresse belegen.

```
main:
     mov a, 0x10
     jmp seta
     mov a, 0x20

end:
     jmp end

seta:
     mov a, 0x30
     jmp end
```

In der ersten Zeile weisen wir Register A den Wert 10h zu. Dann springen wir zum Teilabschnitt *SetA*, der Register A den Wert 30h zuweist. Danach wird zum Label *End* gesprungen. Die dritte Zeile `MOV A, 0x20` wird hintergangen, Register A wird somit nie den Wert 0x20 annehmen.

Wenn eine Routine ausgeführt wurden ist, so sollte man da weiter machen können, wo man sie auch aufgerufen hat. Dazu gibt es die Instruktion `CALL` in Zusammenhang mit `RET`. `CALL` dürfte klar sein, und `RET` kommt von *Return*.

```
main:
     mov  sp, 0x2000

     mov  a, 0x10
     call seta
     mov  a, 0x30
     jmp  end

end:
     jmp  end

seta:
     mov  a, 0x20
     ret
```

Jetzt kommen wir langsam dahin, was einen guten Cracker(oder ohne böswillige Absicht, einen Hacker) ausmacht - das sofortige verstehen solcher Codes. Also was wird genau gemacht?

* Wieder wird in der ersten Zeile dem Register A der Wert 10h zugewiesen.
* Jetzt wird mit der Instruktion `CALL SetA` erst der Program Counter + 3(denn soviel Byte nimmt CALL ein) und dann das V Register(also insgesamt 3 Byte) auf den Stack gepusht. Anschließend wird der Program Counter auf die Adresse von *SetA* angesetzt.
* Nun wird `MOV A, 0x20` ausgeführt.
* Mit `RET` nimmt er zuerst das Register V und dann das Register PC vom Stack. Anschließend wird der Stack Pointer um 3 erhöht
* Als nächstes wird dann `MOV A, 0x30` ausgeführt und mit
* `JMP End` wieder in die Endlosschleife gesprungen.

Typischerweise kann ein Spiel folgendermaßen strukturiert sein:

```
main:
     call init
     jmp  mainloop

mainloop:
     call handleinput
     call handlecollisions
     jmp  mainloop

init:
     ; Hier werden z. B. Speicherstände geladen
     ret

handleinput:
     ; Hier wird die Eingabe verarbeitet
     ret

handlecollisions:
     ; Hier wird Kollisionserkennung durchgeführt
     ret
```

Bedingte Sprünge

Wir haben zwar noch keine Eingabe verarbeitet, geschweige denn was auf dem Display angezeigt. Aber stellen wir uns vor, wir haben ein Textadventure, und der User hat 2 mögliche Antworten vorgegeben.

„Edler Ritter, möchtest du die Burg angreifen?"

* „Ja"
* „Nein"

Auf beide Antworten muss entsprechend reagiert werden. Bei „Ja" wird dann zur Subroutine gesprungen, wo die Burg angegriffen wird, bei „Nein" eine weinende Prinzessin angezeigt.

Gehen wir davon aus, dass in Register A die Antwort gespeichert wird. 1 für „Ja" und 2 für „Nein". Jetzt müssen wir Register A mit dem Wert 1 und dem Wert 2 vergleichen. Dafür gibt es die Instruktion `CMP` - Compare. Im Eigentlichen macht sie nichts anderes, als Subtrahieren, und das Ergebnis zu verwerfen. Beim Subtrahieren werden bestimmte Bits im Flagregister gesetzt.

```
cmp a, 1
```

Hier wird A - 1 gerechnet. Wenn Register A den Wert 1 hat, so ist das Ergebnis 0, und der PMini müsste das Zero Flag im Flagregister setzen. Ist es also gesetzt, sind die zu vergleichenden Werte gleich. Ist Register A gleich 0, so würde er 0 - 1 rechnen. Das Ergebnis ist -1, also negativ. So wird das Sign Flag gesetzt. Das kennzeichnet, dass das Ergebnis negativ, und A kleiner als 1 ist. Sind Sign Flag und Zero Flag nicht gesetzt, so ist A größer als 1.

Am Ende läuft es darauf aus, das wir auf die gesetzten oder nicht gesetzten Flags im

Flagregister reagieren müssen. Hier eine komplette Liste mit allen Instruktionen. Der Unterschied zwischen Jump und Call dürfte klar sein:

```
 Befehl        | Bedeutung                         | Bedingung
--------------+----------------------------------+---------------------
 JL  / CALLL  | jump / call if lower              | SF <> OF
 JLE / CALLLE | jump / call if lower or equel     | ZF = 1 oder SF <> OF
 LG  / CALLG  | jump / call if greater            | ZF = 0 und SF = 0
 JGE / CALLGE | jump / call if greater or equel | SF = OF
 JO  / CALLO  | jump / call if overflow           | OF = 1
 JNO / CALLNO | jump / call if not overflow       | OF = 0
 JS  / CALLS  | jump / call if signed             | SF = 1
 JNS / CALLNS | jump / call if not signed         | SF = 0
 JC  / CALLC  | jump / call if carry              | CF = 1
 JNC / CALLNC | jump / call if not carry          | CF = 0
 JZ  / CALLZ  | jump / call if zero               | ZF = 1
 JNZ / CALLNZ | jump / call if not zero           | ZF = 0
```

In unserem Fall müssen wir eben die Antwort in Register A mit dem Wert 1 vergleichen. Die Subroutine für „Ja" wird nur ausgeführt, wenn Register A = 1 ist, und die Subroutine „Nein" wenn Register A <> 1 ist. Als wählen wir `CALLZ` und `CALLNZ`.

```
main:
     mov    sp, 0x2000

     mov    a, 1

     cmp    a, 1
     callz  attackclcastle
     callnz drawprincess

     jmp    end

end:
     jmp    end

attackclcastle:
     ; Grafik anzeigen, wie die Burg angegriffen wird
     mov    b, 0x10
     ret

drawprincess:
     ; Grafik der weinenden Prinzessin anzeigen
     mov    b, 0x20
     ret
```

Repräsentativ für die Antwort „Ja", habe ich hier banal `MOV B, 0x10` und für Antwort „Nein" `MOV B, 0x20` genommen, damit überhaupt was geschieht.

Was geschieht im Emulator?

* `MOV A, 1` -> Register A hat den Wert 1
* `CMP A, 1` -> Gleichbedeutend mit `SUB A, 1` -> Das Zero Flag wird gesetzt
* `CALLZ AttackCastle` -> Zero Flag ist gesetzt, er führt Subroutine *AttackCastle* aus
* `MOV B, 0x10` -> Register B hat den Wert 10h

* `RET` -> Es wird bei *Hauptprogramm* fortgeführt
* `CALLNZ DrawPrincess` -> Zero Flag ist gesetzt, ergo, es wird *DrawPrincess* nicht ausgeführt
* `JMP End` -> Sprung in die Endlosschleife

Nehmen wir statt `MOV A, 1` mal `MOV A, 2`, so wird *DrawPrincess* und damit verbunden `MOV B, 0x20` ausgeführt.

Pseudocode(in BASIC) dafür ist:

```
If RegisterA = 1 Then
     RegisterB = 0x10
Else
     RegisterB = 0x20
EndIf
```

Select Case

Nun, jedes gute Spiel hat natürlich ein Hauptmenü wo ein neues Spiel gewählt, ein gespeichertes Spiel fortgesetzt, die Soundeinstellungen verändert und die Credits Anzeige u. ä. aufgerufen werden kann.

„Hauptmenü"

* „Neues Spie" -> 1
* „Spiel fortsetzen" -> 2
* „Einstellungen" -> 3
* „Credits" -> 4

Jetzt haben wir gleich 4 Auswahlmöglichkeiten, auf die wir alle eingehen müssen. Da wäre es doch Quatsch, wenn nach „Neues Spiel" noch „Spiel fortsetzen" usw. geprüft werden. In höheren Programmiersprachen, wie z. B. C, gibt es dafür ein `break` Befehl, das weitere Prüfungen beendet. Bei anderen Sprachen wird es gar automatisch gesetzt. Für diesen Zweck benutzen wir nur eine einfache `JMP` Instruktion, die an das Ende der ganzen Abfragen springt. Weiterhin wollen wir auch noch auf eine unbehandelte Auswahl eingehen. In höheren Sprachen als `default` oder auch `else` bezeichnet.

```
main:
     mov sp, 0x2000

     mov a, 1

     cmp a, 1
     jz  newgame

     cmp a, 2
     jz  continue

     cmp a, 3
     jz  settings

     cmp a, 4
```

```
     jz  credits

     call default

break:
     jmp end

end:
     jmp end

newgame:
     ; Neues Spiel erzeugen
     mov b, 0x10
     jmp break

continue:
     ; Spiel fortsetzen
     mov b, 0x20
     jmp break

settings:
     ; Einstellungen von Sound und Co.
     mov b, 0x30
     jmp break

credits:
     ; Hier werden die Macher des Spiels aufgelistet
     mov b, 0x40
     jmp break

default:
     ; "Falsche Eingabe" ausgeben
     mov b, 0xFF
     jmp break
```

Wieder habe ich nur Register B repräsentativ andere Werte verpasst. Bei einer falschen Eingabe hat es FFh als Wert. Versuch einfach das Programm Schritt für Schritt auszuführen. Setze dabei für `MOV A, 1` 2, 3, 4 und 5 ein und schaue was passiert mit Register B am Schluss.

Pseudocode mäßig sieht das ganze wieder so aus:

```
Select RegisterA
     Case 1:
          RegisterB = 0x10
     Case 2:
          RegisterB = 0x20
     Case 3:
          RegisterB = 0x30
     Case 4:
          RegisterB = 0x40
     Default:
          RegisterB = 0xFF
End Select
```

Das *break* hätte man wegen der Default-Behandlung schon gar nicht wegfallen lassen

dürfen.

Schleifen

Schleifen sind die Teile im Programm, die Wiederholt werden sollen. Eingaberoutine, Kollisionserkennung usw. werden während des Spiels ständig wieder aufgerufen, bis der Benutzer das Spiel beendet. Diese Abfolge nennt man z. B. Hauptschleife oder Mainloop. Für unterschiedliche Zwecke gibt es unterschiedliche Schleifentypen. Befehls mäßig werden wir hier jedoch nichts neues kennenlernen.

Die For-Next-Schleife

Wenn bekannt ist, wie oft ein Programmteil ausgeführt werden soll, so kommt diese Schleife zum Einsatz. Gezählt werden kann entweder über eine Variable, oder falls nicht anders benötigt, über das Zählregister B.

```
.orgfill 0x1460
varx:   .db 0

.orgfill 0x02100
     .db "MN"
     jmp main

.orgfill 0x021A4
     .db "NINTENDO"
     .db "1234"
     .db "For-Next"

.orgfill 0x021BC
     .db "2P"

.orgfill 0x021D0
main:
     mov sp, 0x2000

     mov a, 1
     mov [varx], a

     mov b, 1
     jmp next

loop:
     mov a, [varx]
     shl a
     mov [varx], a

     inc b

next:
     cmp b, 5
     jle loop

     jmp end
```

```
end:
     jmp end

.end
```

Hier wird der Anfangswert 1 der Variable *VarX* 5 mal nach links verschoben. Also 1, 2, 4, 8, 16, 32. Wichtig ist, dass die Prüfung bei *next* zuerst ausgeführt wird. Erst dann kann der Schleifenkörper zum Einsatz kommen.

```
VarX = 1
For B = 1 To 5
     VarX = VarX Shl 1
Next
```

So sehe das Programm in einer Hochsprache aus.

Die Repeat-Until-Schleife

Sie ist eine Fuß gesteuerte Schleife. D. h., der Schleifenkörper wird mindestens 1 mal ausgeführt. Erst dann wird die Abbruchbedingung geprüft.

```
.orgfill 0x1460
varx:   .db 0

.orgfill 0x02100
     .db "MN"
     jmp main

.orgfill 0x021A4
     .db "NINTENDO"
     .db "1234"
     .db "Repeat-Until"

.orgfill 0x021BC
     .db "2P"

.orgfill 0x021D0
main:
     mov sp, 0x2000

     mov a, 1
     mov [varx], a

loop:
     mov a, [varx]
     add 2
     mov [varx], a

     cmp a, 20
     jle loop

     jmp end

end:
     jmp end
```

```
.end
```

Dieses Programm erhöht im Schleifenkörper die Variable *VarX* solange, bis sie größer 20 ist. Dabei wird der Schleifenkörper mindestens 1 mal ausgeführt, egal welchen Anfangswert *VarX* hatte.

Das Programm in einer Hochsprache:

```
VarX = 1
Repeat
     VarX = VarX + 2
Until VarX > 20
```

Die While-Wend-Schleife

Sie funktioniert ähnlich wie die Repeat-Until-Schleife, jedoch wird die Bedingung schon vor dem ersten Ausführen des Schleifenkörpers durchgeführt. Es ist somit durchaus möglich, dass der Schleifenkörper überhaupt nicht ausgeführt wird.

```
.orgfill 0x1460
varx:   .db 0

.orgfill 0x02100
     .db "MN"
     jmp main

.orgfill 0x021A4
     .db "NINTENDO"
     .db "1234"
     .db "While-Wend"

.orgfill 0x021BC
     .db "2P"

.orgfill 0x021D0
main:
     mov sp, 0x2000

     mov a, 1
     mov [varx], a

     jmp wend

loop:
     mov a, [varx]
     add 2
     mov [varx], a

wend:
     cmp a, 20
     jle loop

     jmp end
```

```
end:
     jmp end

.end
```

Hier wird zuerst die Bedingung geprüft, ob *VarX* > 20 ist. Wenn ja, wird die Schleife sofort unterbrochen.

```
VarX = 1
While VarX
     VarX = VarX + 2
Wend
```

Das ganze wieder in einer Hochsprache.

Prozeduren und Funktionen

Mit den Sprüngen haben wir schon eine Möglichkeit kennen gelernt, wie man sein Programm in mehrere Fragmente unterteilen kann. Der Nachteil wahr, man musste ihre Parameter global definieren. Prozeduren und Funktionen hingegen bieten die Möglichkeit, die Parameter lokal zu definieren. Nachdem die Prozedur/Funktion beendet ist, sind diese Parameter nicht mehr zugänglich.

Die Übergabeparameter werden hier auf den Stack ausgelagert. Da man nun mehrere Prozeduren/Funktionen aufrufen kann, müssen diese dann intern mit Hilfe des Stack Pointers adressiert werden.

Schauen wir uns ein einfaches Beispiel an. Die Prozedur soll 3 Übergabeparameter miteinander addieren, und das Ergebnis in die Variable *VarX* schreiben.

In einer Hochsprache wäre dies z. B.:

```
Addiere(10, 20, 30)
Procedure Addition(A, B, C)
     VarX = A + B + C
End Procedure
```

Wir müssen diese Parameter(10, 20 und 30) auf den Stack legen, bevor wir die Prozedur aufrufen:

```
mov ba, 30
pop ba

mov ba, 20
pop ba

mov ba, 10
pop ba

call addition
```

Als erstes sehen wir, dass die Reihenfolge der Parameter vertauscht ist. Zweitens, das wir 2 Byte-Werte benutzen.

Das Problem ist, wir können mit dem Stack Pointer nur 16 Bit Werte ansprechen. Egal, ob wir nur 8 Bit Werte benötigen.

```
.orgfill 0x1460
varx:   .db 0

.orgfill 0x02100
     .db "MN"
     jmp main

.orgfill 0x021A4
     .db "NINTENDO"
     .db "1234"
     .db "Procedure"

.orgfill 0x021BC
     .db "2P"

.orgfill 0x021D0
main:
     mov  sp, 0x2000

     mov  ba, 30
     push ba

     mov  ba, 20
     push ba

     mov  ba, 10
     push ba

     call addition
     add  sp, 6

     jmp  end

end:
     jmp  end

addition:
     mov ba, [sp + 3]

     mov hl, [sp + 5]
     mov b, l
     add a, b

     mov hl, [sp + 7]
     mov b, l
     add a, b

     mov [varx], a

     ret

.end
```

Wir sehen, der erste Parameter wird über [SP + 3] angesprochen, der zweite über [SP +

5] und der dritte über [SP + 7]. Generell können alle 16 Bit Register zum Adressieren mit dem Stack Pointer benutzt werden. `ADD SP, 6` kommt daher Zustande, da der Stack Pointer noch auf den letzten Parameter zeigt. Und 3 Parameter nehmen 6 Byte ein.

Der Unterschied zwischen Prozeduren und Funktionen ist der, dass Funktionen einen Rückgabe Wert besitzen. Das funktioniert wie in der Mathematik. Man übergibt z. B. den Parameter 90° der Funktion *Sinus*, und erhält dafür den Rückgabe Wert 1. In der Prozedur haben wir diesen Rückgabe Wert in eine globale Variable *VarX* gespeichert. Bei einer Funktion wird dieser Rückgabe Wert jedoch ebenfalls lokal auf den Stack gelegt.

Das ganze in einer Hochsprache sehe so aus:

```
Print Addition(10, 20, 30)
Function Addition(A, B, C)
     Return A + B + C
End Function
```

Der Aufruf ist der selbe, wie bei den Prozeduren, jedoch nach dem Aufruf muss der Rückgabe Wert vom Stack gepopt werden.

```
.orgfill 0x1460
varx:   .db 0

.orgfill 0x02100
     .db "MN"
     jmp main

.orgfill 0x021A4
     .db "NINTENDO"
     .db "1234"
     .db "Function"

.orgfill 0x021BC
     .db "2P"

.orgfill 0x021D0
main:
     mov  sp, 0x2000

     mov  ba, 30
     push ba

     mov  ba, 20
     push ba

     mov  ba, 10
     push ba

     call addition
     sub  sp, 5
     pop  ba
     add  sp, 9

     jmp  end

end:
```

```
    jmp  end

addition:
    mov ba, [sp + 3]

    mov hl, [sp + 5]
    mov b, l
    add a, b

    mov hl, [sp + 7]
    mov b, l
    add a, b

    push ba
    add sp, 2

    ret

.end
```

Hier sorgt `ADD SP, 2` dafür, dass der Stack Pointer beim Ausführen von `RET` auf die Rücksprungadresse zeigt. Schließlich haben wir nach der Rücksprungadresse von `CALL` noch 2 Byte des Registers BA auf den Stack gelegt. Nach `RET` zeigt der Stack Pointer auf den 3ten Parameter. Durch `SUB SP, 5` zeigt er dann wieder auf den Rückgabe Wert. Mit `ADD SP, 9` wird anschließend sichergestellt, dass die 6 Byte Parameter sowie 3 Byte Rücksprungadresse lokal überschrieben werden können.

Das ganze ist doch recht umständlich. Da bietet der X86er, hinter den wir gerade sitzen, doch komfortablere Möglichkeiten. Aber der ganze Aufwand rentiert sich bei größeren Projekten.

Damit wäre nun alles zum Thema PMini und Assembler gesagt. Jetzt folgen noch ein paar nützliche Tipps zum Umgang mit PMAs und dann kann es mit der Hardware Programmierung losgehen.

Assembler-Direktiven

Als Assemblerprogrammierer hat man es nicht besonders leicht. Deswegen stellen die meisten Assembler so geannte Direktiven zur Verfügung, die das Programmieren mit ihnen um einiges erleichtern. Vor der eigentlichen Assemblierung kommt ein Präprozessor zum Einsatz, der alle Direktiven auflöst. Direktiven stellen z. B. Konstanten dar. Ihre symbolische Namen werden durch ihre, für den Mikroprozessor verständlichen, Zahlenwerte ersetzt. Dies findet z. B. häufig Einsatz bei Adressen. Kein Programmierer will sich unnötige Hardware Adressen merken. Also verwendet man symbolische Namen wie *VRAM_ADDRESS*.

Direktiven erkennt man beim PMAs durch den vorangestellten Punkt. Z. B. bei `.orgfill`.

.include

Wenn ein Projekt bis ins unermessliche wächst, dann ist es von Vorteil, ähnlich wie bei

den Routinen/Prozeduren und Funktionen, den Code auf mehrere Quellcodes zu Verteilen. Die eine Quellcode Datei behandelt z. B. nur das Abarbeiten der Highscore, die andere nur die Kollisionserkennung usw.

framework.asm:

```
.orgfill 0x02100
     .db "MN"
     jmp main

.orgfill 0x021A4
     .db "NINTENDO"
     .db "1234"
     .db "Test"

.orgfill 0x021BC
     .db "2P"

.orgfill 0x021D0
```

test.asm:

```
.orgfill 0x1460
; Irgendwelche Variablen
varx:   .db 0
vary:   .db 0
varz:   .db 0

; Das Framework einbinden
.include "framework.asm"

main:
     mov a, 1
     add a, 2

end:
     jmp end
```

Ich habe die Erfahrung gemacht, dass es hier einen Fehler im PMAs gibt. Wenn man den PMAs über relative Pfade aufruft, meldet er, dass die einzubeziehende Datei *framework.asm* nicht geöffnet werden kann.

.equ

Kommt von Equal. Man definiert, wie oben schon erwähnt, einen symbolischen Namen, und einen dazugehörigen konstanten Wert. Symbolischer Name und konstanter Wert zusammen nennt man einfach Konstante.

```
; Adresse des Keypads ist 0x002052
.equ    KEYPAD 0x002052

main:
     mov a, [KEYPAD]
```

```
end:
     jmp end
```

Der Aufruf `MOV A, [KEYPAD]` ist gleichbedeutend mit `MOV A, [0x002052]`.

.org

Kommt von Origin. Legt die Startadresse des folgenden Quellcode Abschnitts fest. Die Adresse des WorkRAMs ist z. B. 0x1460, und nur dort können Variablen adressiert werden:

```
.org 0x1460
varx:   .db 0
vary:   .db 0
varz:   .db 0
```

Die Sprungmarke *VarX* hat die Adresse 0x1460, die Sprungmarke *VarY* die Adresse 0x1461 und die Sprungmarke *VarZ* die Adresse 0x1462.

```
001460  00      varx:      add      A, A
001461  00      vary:      add      A, A
001462  00      varz:      add      A, A
```

.orgfill

Ist gleichbedeutend mit `.org`, jedoch füllt sie „leere“ Abschnitte mit Nullen auf. So die Theorie. Im Quellcode des PMAs konnte ich jedoch erkennen, dass beide Direktiven mit Nullen auffüllen.

.end

Alles, was nach dieser Direktive folgt, wird vom Assembler ignoriert und somit auch nicht assembliert.

```
main:
     mov a, 1

.end
     ; Wird nicht assembliert
     mov a, 2
```

```
0021D0  B0 01   main:      mov      A, 01
0021D2  CD                 xchg     A, [HL]
0021D3  CD                 xchg     A, [HL]
```

.incbin

Kommt von Include Binary. Man kann hier eine beliebige Ressource zu seinem Programm hinzufügen. Vorzugsweise werden das Grafiken, oder für Testzwecke, das BIOS sein. Der Assembler bindet in die Ausgabedatei *.bin einfach diese Ressource mit

ein.

```
picture:
     .incbin "picture.dat"
```

Solche Ressource Dateien kann man sich mit Konverter erzeugen lassen. Diese wandeln dann z. B. Bitmap Grafiken in *.dat Dateien um.

Weitere Direktiven

Der PMAs ist, wie der Name schon sagt, ein Makroassembler. Das bedeutet, dass er auch Makros verarbeiten kann. Diese werden zwischen `.macro` und `.endm` definiert. Ich werde nicht weiter auf diese eingehen, jedoch mal ein kleines Beispiel zeigen:

```
.macro setcontrast contrast
     mov x1,   0x20fe
     mov [x1], 0x81
     inc x1
     mov [x1], contrast
.endm
```

Der Aufruf erfolgt z. B. durch `SetContrast 0x3F`.

Abschließende Worte

Erst einmal Respekt, wer sich dieses Tutorial von vorne bis hinten durchgelesen hat. Ich hoffe, mir ist es gelungen, ein wenig Licht ins Dunkle zu bringen. Das ganze mag trockene Theorie sein, wird sich aber hoffentlich mit dem 2ten Teil dieses Tutorials bezahlt machen.
(Bis jetzt gibt es noch keinen 2ten Teil ^^)