Tutorial: Die Grundlagen von ASM

von WhiteYoshiEgg

0. Notizen vorweg

- Dieses Tutorial wird dir (hoffentlich) **ASM**, die Programmiersprache der SNES, am Beispiel von Super Mario World beibringen.
- Geschrieben wurd's von mir: **WhiteYoshiEgg**, aka WYE, aka Eric, ein momentan 16 Jahre alter Super-Mario-World-Hacker aus Deutschland. Ich bin selbst kein Riesen-Experte in Sachen ASM, aber ich möchte doch sagen, dass ich mehr darüber weiß als der Durchschnitt.
- Dieses Tutorial ist nur was für dich, wenn du bereits mit den restlichen SMW-Hacking-Aspekten genügend vertraut bist. Wenn du noch nicht so viel Ahnung davon hast – oder nicht mal weißt, wovon ich hier überhaupt rede – lass die Finger davon.
- Ich bin meiner Meinung nach nicht besonders gut im Erklären – manche Abschnitte sind vielleicht zu lang, zu ausführlich oder zu sehr um den heißen Brei herumgeredet. Ich habe versucht, unter jedem Kapitel die neuen und wichtigsten Inhalte ganz kurz zusammenzufassen – ich empfehle daher, immer den „**Zusammengefasst**"-Teil zu lesen, damit ihr wisst, was wichtig ist und was nicht.
- Für weitere Fragen etc. findet ihr ganz am Ende ein paar Adressen, unter denen ich zu erreichen bin.

Inhalt

1. Zuallererst: Zahlensysteme – Bits und Bytes

Wir Menschen rechnen bekanntermaßen im Zehnersystem, auch **Dezimalsystem** genannt: Wir haben zehn Ziffern, von 0 bis 9, und die erste zweistellige Zahl ist die Zehn. Das erscheint uns selbstverständlich, aber wahrscheinlich haben wir das Zehnersystem nur deswegen, weil wir zehn Finger haben – wir hätten uns sicher auch leicht an einen Neuner- oder Elfersystem gewöhnen können, hätten wir die entsprechende Zahl an Fingern. Haben wir beim Zählen alle Finger ausgestreckt, ist die erste zweistellige Zahl erreicht, danach fangen wir wieder von vorn an.

Computer und andere Maschinen haben leider keine Finger, an denen sie irgendwas abzählen können. Sie kennen allerdings zwei Zustände: „Strom an“ (0) und „Strom aus“ (1). Und auch aus zwei Ziffern kann man ein Zahlensystem bilden! Dann ist die erste zweistellige Zahl eben schon die Zwei. Das Ganze nennt man dann **Binärsystem** (oder auch Dualsystem). Hier mal eine kurze Umrechnungstabelle:

Dezimal	Binär
0	0
1	1
2	10

4	100
7	111
8	1000
10	1010
20	10100
42	101010

Übrigens: Eine Stelle im Binärsystem wird **Bit** genannt. Klingelt da was bei dir? Der Ausdruck „Bits und Bytes“ sagt dir vielleicht was. Ja, auch **Byte** ist ein Begriff – Ein Byte besteht aus 8 Bits. So einfach ist das.

```
1          1111 1111
Bit        Byte
```

Aus Gründen der Übersichtlichkeit werden in einem Byte oft die beiden Hälften durch ein Leerzeichen voneinander getrennt. Übrigens: Eine Gruppe aus vier Bits nennt man „Nibble“ oder auch **Nybble**. Fragt mich nicht, woher der Name kommt. 4 Bits = 1 Nybble, 8 Bits = 1 Byte.

Vom Zehnersystem weißt du ja, dass man Nullen am Anfang weglassen kann – **075** heißt das gleiche wie **75**. Im Binärsystem ginge das theoretisch auch, aber – auch hier der Übersichtlichkeit halber – man macht’s anders. Normalerweise fügt man am Anfang immer so viele Nullen an, bis ganze Bytes erreicht sind – statt `10010` würde man also `0001 0010` schreiben.

So, über das Binärsystem weißt du jetzt hoffentlich Bescheid. Und ja, jede Computerdatei besteht aus Nullen und Einsen. Je nach Dateiformat werden diese Nullen und Einsen natürlich anders interpretiert.

Der Nachteil des Binärsystems ist allerdings, dass schon kleinere Zahlen elend lang sind – weil es eben nur zwei Ziffern gibt. Und so lange und gleich aussehende Zahlen will sich natürlich niemand antun müssen. Muss man glücklicherweise auch nicht – man zeigt die Zahlen einfach in einem anderen System an. Dafür nehmen wir nicht das Dezimalsystem, sondern eins, das dem Binärsystem ähnlicher ist – das Sechzehnersystem, wesentlich öfter **Hexadezimalsystem** oder kurz **Hex** genannt. Wie du vielleicht schon geahnt hast, hat Hex 16 Stellen, und die erste zweistellige Zahl ist die Sechzehn. Moment – die erste zweistellige Zahl ist die Sechzehn? Und wieso gibt es mehr als zehn Ziffern? Wie sehen die denn aus? Gute Fragen – und die Antwort? Zwischen der 10 und der 16 behilft man sich in Hex mit Buchstaben, genauer gesagt, mit den **Buchstaben von A bis F**. Das ganze sieht so aus:

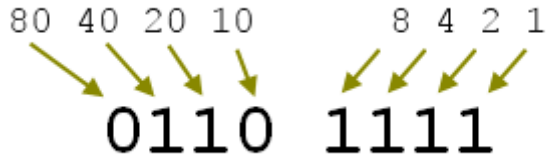
Dezimal	Binär	Hex
0	0000 0000	00
1	0000 0001	01
2	0000 0010	02
3	0000 0011	03
4	0000 0100	04
5	0000 0101	05
6	0000 0110	06
7	0000 0111	07
8	0000 1000	08
9	0000 1001	09
10	0000 1010	0A
11	0000 1011	0B
12	0000 1100	0C
13	0000 1101	0D
14	0000 1110	0E
15	0000 1111	0F
16	0001 0000	10
17	0001 0001	11
99	0110 0011	63
100	0110 0100	64
255	1111 1111	FF

Nach der 9 kommt das **A**, dann **B, C, D, E und F**, und dann erst die 10 (die im Zehnersystem 16 heißt). Wie ihr das ganze aussprecht – ob ihr die Hex-Zahl 12C jetzt „Zwölf C“, „Eins Zwei C“ oder „Hundertzehundzwanzig“ nennt – kann mir und euch übrigens wurscht sein. Das interessiert höchstens Mathelehrer.

Und wieso ist Hex jetzt dem Binärsystem ähnlicher? Schauen wir uns mal eine Zahl an:

```
Binär:    0110 1111
Hex:      6F
```

Jedes Bit steht für eine bestimmt Zahl – je weiter links, desto höher die Zahl. Das kennst du schon vom Dezimalsystem: In der Zahl 555 zum Beispiel steht auch nicht jede 5 für dasselbe. Die 5 ganz rechts steht für 5 ($5*10^0$), die daneben für 50 ($5*10^1$) und die 5 ganz links für 500 ($5*10^2$). Wenn du diese drei Zahlen addierst, kommt wieder 555 heraus. Genau so ist es im Binärsystem auch, nur dass hier statt der 10 die 2 als Basis verwendet wird. An unserem Beispiel oben: Die 1 ganz rechts steht für 1 ($1*2^0$), die daneben für 2 ($1*2^1$), die daneben für 4 ($1*2^2$), dann 8, 16, 32, 64 und schließlich 128. In Hex heißen diese Zahlen 1, 2, 4, 8, 10, 20 40 und 80. Sind einander schon ähnlicher als im Zehnersystem, oder?



Wir addieren bei jedem Bit, das 1 ist, die entsprechende Hex-Zahl, für die es steht. Oben wären es also 40 + 20 + 8 + 4 + 2 + 1 = 6F.

Wie du siehst, unterscheiden sich die Zahlen bei jedem Nybble nur um die 0 hinten dran – und das hat einen Vorteil: Jedes Nybble entspricht genau einer Hex-Stelle. Die 0110 steht für die Stelle 6 – wäre sie das rechte Nybble, wäre die dazugehörige Stelle immer noch 6. Und würde man von der Beispielzahl 6F die beiden Nybbles vertauschen, käme F6 dabei heraus, also auch nur mit vertauschten Stellen. Das ist alles andere als selbstverständlich! Im Dezimalsystem ginge das nicht so einfach.

Hinzu kommt, dass es in beiden Systemen zur selben Zeit einen Wechsel der Stellenanzahl gibt. Im Binärsystem muss man nach `1111 1111` ein neues Byte anfangen, so dass man jetzt zwei hat: `0000 0001 0000 0000`. In Hex wäre das `FF` zu `0100` – auch hier braucht man jetzt mehr Stellen. Im Dezimalsystem dagegen wäre das ein Wechsel zwischen 255 und 256 - Nix mit mehr Stellen.

Übrigens ist es in Hex üblich, die Zahlen immer eine gerade Zahl an Stellen (2, 4, 6 etc.) haben zu lassen – mal wieder aus Gründen der Übersichtlichkeit.

So, das erste Kapitel wäre geschafft. Ganz schön happig, oder? Und mit ASM selbst haben wir ja noch gar nicht angefangen. Aber keine Sorge – **du musst nicht jedes Wort im Kopf behalten**. Wichtig ist nur, dass du über die drei verschiedenen Systeme Bescheid weißt und

vielleicht in ganz groben Zügen zwischen ihnen umrechnen kannst. Und selbst dafür gibt es ja den Windows-Rechner.

Zusammengefasst:

- Drei verschiedene Zahlensysteme sind wichtig: **Dezimal** (das „Normale“), **Binär** (das, was ein Gerät versteht) und **Hex** (ein verhältnismäßig gut lesbares Zahlensystem, das gut zum Binärsystem passt).
- Eine Binärstelle (0 oder 1) heißt **Bit**.
- 4 Bits sind ein **Nybble**, 8 Bits sind ein **Byte**.
- Bytes können als **zweistellige Hex-Zahl**-dargestellt werden.

2. Programmieren mit Nullen und Einsen – Hex-Editoren

Wie gesagt, im Grunde besteht alles für den Computer – Dateien, Programme, das Betriebssystem und auch die Software, die nach dem Einschalten des Computers das Betriebssystem startet – aus Nullen und Einsen. Ja, sogar der Videorekorder wird mit Nullen und Einsen programmiert und auch die Daten auf CDs oder DVDs sind im Grunde nichts anderes als Nullen und Einsen, auf die Unterseite der Scheibe eingraviert. Wie nun diese Geräte aus den Binärzahlen Programmierbefehle machen, ist mir schleierhaft. Wie auch immer, freuen wir uns, dass es funktioniert.

Und weil eben jede Computerdatei aus denselben Ziffern besteht, gibt es auch Programme, die diese Nullen und Einsen darstellen können, egal, was für eine Datei es ist. Da Binärzahlen aber, wie gesagt, nicht gerade gut zu lesen sind, gibt es auch Programme, die sie – Byte für Byte – als Hex-Zahlen darstellen. Außerdem kann man die Zahlen dort auch ändern und so jede beliebige Datei bearbeiten (was natürlich ziemlich umständlich ist, aber theoretisch ginge das). Von diesen so genannten **Hex-Editoren** gibt es eine ganze Menge – ein recht bekannter, und der, den ich benutze, nennt sich **Translhextion**. Wenn du mein Hex-Tutorial gelesen hast, weißt du, dass man durch Bearbeiten der Hex-Zahlen den Programmiercode von SMW verändern kann. Denn – klar – auch SNES-Programme bestehen aus Binärzahlen, und die Programmiersprache, die sie verwendet – ASM – ist Hex sehr ähnlich. Jeder Programmierbefehl ist genau ein Byte lang, und die Zahlen, die den Befehlen folgen, 1, 2 oder 3 Bytes.

Zusammengefasst:

- Alles für Computer oder Spielekonsolen (jede Datei, jedes Programm) besteht aus **Binärzahlen** (Nullen und Einsen), die je nach Dateiformat anders interpretiert werden.
- Natürlich besteht auch der Programmiercode von **SNES-Spielen** aus Nullen und Einsen.
- **Hex-Editoren** zeigen jede beliebige Datei an, wobei die Nullen und Einsen im Hex-Format angezeigt werden.
- In ASM entspricht **jeder Programmierbefehl genau einem Byte**.

3. Was ist ASM? – ASM, Hex, Assembler, Disassembler

So, nach dieser viel zu langen Vorrede kommen wir nun endlich zum Hauptthema – ASM.

ASM steht für **As**se**m**bly und bezeichnet eigentlich eine ganze Reihe von Programmier-sprachen. Jedes Gerät – ob Computer, SNES oder Gameboy – hat seine eigene, ganz spezielle ASM-Sprache. In diesem Tutorial lernst du natürlich nur eine ASM-Sprache, nämlich die für die SNES. Sie wird auch **65c816-ASM** genannt. Wann immer ich ab jetzt von ASM rede, meine ich nur ASM für die SNES.

Wie du schon weißt, bestehen ASM-Befehle aus Nullen und Einsen, die man bequem als Hex-Zahlen darstellen und in einem Hex-Editor ansehen und bearbeiten kann. Aber mal ehrlich, nur mit Hex-Zahlen zu arbeiten ist auch nicht gerade verständlich und übersichtlich. Daher schreibt man die Befehle, wann immer man kann, mit Buchstaben. In 65c816-ASM ist jeder Befehl drei Buchstaben lang, und manchmal folgen ihm Zahlen. Der Befehl sagt, was gemacht werden soll, die Zahl dahinter, wie und wo. Auch wenn du die Befehle selbst noch nicht verstehst, zeige ich dir mal ein Diagramm:

Binär	Hex	ASM
1010 1001 0000 0001	A9 01	LDA #$01

Die Binärzahlen – der **Maschinencode** – ist das, was die Konsole versteht. In der mittleren Spalte steht dasselbe, nur in etwas leichter lesbare Hex-Zahlen umgeformt. Und in der Spalte ganz rechts steht der ASM-Code dazu. Es werden hier nur die Programmierbefehle durch Buchstaben dargestellt, was für Menschen noch leichter lesbar ist. Klar, den Befehl selbst verstehst du hier noch nicht, aber genau dafür schreibe ich ja dieses Tutorial.

Wenn du jetzt selbst ASM-Code schreibst, benutzt du immer die Buchstaben-Variante. Dazu kannst einfach den Windows-Editor benutzen und deinen Code dort hinein schreiben – ein eigenes Programmier-Tool brauchst du dafür nicht.

Nun versteht zwar der Mensch die Buchstaben, aber die SNES nicht – die braucht Binär- bzw. Hex-Zahlen. Wäre ja schön, wenn es ein Programm gäbe, das den ASM-Code wieder in Zahlen zurückübersetzt, oder? Gott sei Dank gibt es solche Programme! Sie nennen sich **Assembler**. Immer, wenn du mit Tools wie Blocktool oder Sprite Tool arbeitest, ist ein Assembler nötig: Er verwandelt den ASM-Code, mit dem der Block oder der Sprite geschrieben wurde, in Hex-Zahlen und fügt sie in die ROM ein.

ASM, Hex, ROM und RAM

Wie sind denn nun Programme für die SNES aufgebaut? Die ROMs (also die Kopien von SNES-Spielen im Format einer Computerdatei, die du mit einem Emulator ausführen kannst) enthalten den ganzen ASM-Code, in dem sie programmiert wurden.

65c816-ASM (die Programmiersprache der SNES) besteht aus dreibuchstabigen Befehlen und einigen Hex-Zahlen. Du kannst dir das so vorstellen: Der Befehl ist das, was gemacht werden soll, die Zahl dahinter ist die Zahl, mit der es gemacht werden soll. Mehr dazu natürlich später.

Wenn du den Code einer ROM ansehen willst, kannst du sie in einem Hex-Editor öffnen. Ein Hex-

RAM-Adressen

Wenn du die Seite smwcentral.net kennst, hast du vielleicht schon mal die ROM- und RAM-Maps gesehen. In der ROM-Map findet sich der Code von SMW (natürlich in ASM geschrieben, allerdings in Hex-Zahlen übersetzt), in der RAM-Map augenscheinlich auch. Und was ist jetzt der Unterschied zwischen ROM und RAM? Nun, den Code aus der ROM-Map kannst du nur außerhalb des Spiels verändern. Die Adressen in der RAM-Map allerdings sind ständig frei veränderbar, zum Beispiel Marios Powerup, die Anzahl der Münzen, die Anzahl der Leben, Marios Geschwindigkeit und so weiter. Der Originalcode von SMW verändert diese Eigenschaften selbst (er sagt zum Beispiel: Wenn man eine Münze einsammelt, dann

erhöhe die Anzahl der Münzen um 1). So etwas kannst du mit diesem Tutorial bald (vielleicht) selbst! Du fügst dann neuen ASM-Code ins Spiel ein, der macht, was du willst.

Die RAM-Map findest du auf http://www.smwcentral.net/?p=map&type=ram. Wie du vielleicht siehst, gibt es eine Menge RAM-Adressen, und jede ist für etwas anderes zuständig. Jede enthält einen Hex-Wert zwischen 00 und FF, und je nach Inhalt ist die Auswirkung anders.

Stell es dir vielleicht so vor: Jede RAM-Adresse ist ein Haus, und je nachdem, wer darin wohnt, verändert sich die Stadt (das Spiel). Mit ASM-Befehlen kannst du andere Leute in ein Haus einziehen lassen.

Laden und speichern

Und welche Befehle sind es, die so was tun, und wie benutzt man sie?

Nun ja, in 65c816 ASM gibt es drei sogenannte **Register**, mit denen du arbeiten kannst. Den **Akkumulator** (kurz **A**), und noch die **X**- und **Y**- Register. Zu X und Y komme ich später, jetzt ist erstmal A dran.

So, und hier ist dein erster Programmierbefehl:

```
LDA
```

LDA steht für „Load Data to Accumulator“ (Daten in A laden), und genau das macht er auch – er lädt einen Wert in A. Und wie bestimmt man, welcher Wert geladen wird? So:

```
LDA #$01
```

Jetzt enthält A den Wert 01. Die 0 davor ist wichtig – Alle Zahlen in ASM müssen entweder zwei-, vier- oder sechsstellig sein. Auch das #$ hat seine Bedeutung – es zeigt an, dass eine konkrete Zahl geladen wird. Geht’s auch anders? Ja, und zwar so:

```
LDA $19
```

Hier wird statt #$ nur $ benutzt. Jetzt wird keine absolute Zahl in A geladen, sondern der Inhalt einer RAM-Adresse. In diesem Beispiel ist es Adresse $19, die für Marios Powerup zuständig ist. So hat sie den Wert 00, wenn Mario klein ist, 01, wenn er Super-Mario ist, 02,

wenn er ein Cape hat, und 03, wenn er eine Feuerblume hat. Mit `LDA $19` wird also je nach Powerup eine andere Zahl geladen. Ist Mario klein, wird die Zahl 00 in A geladen, ist er Super-Mario, wird die Zahl 01 geladen und so weiter.

LDA verstanden? `LDA #$xx` lädt eine konkrete Zahl in den Akkumulator, `LDA $xx` lädt den Inhalt einer RAM-Adresse. Das geht, wie oben gesagt, auch mit 4- und 6-stelligen Zahlen: LDA $1DFC zum Beispiel. Vielleicht verwirrt es dich auch, das in der RAM-Map `$7E:0019` steht, obwohl ich hier `$19` benutzt habe. Du kannst statt `$19` genauso gut `$0019` oder `$7E0019` (nur ohne Doppelpunkt) schreiben, das macht keinen Unterschied.

Was allerdings nicht geht (zumindest normalerweise nicht), ist das Laden einer 4- oder 6-stelligen Zahl, zum Beispiel `LDA #$1234`. **Merke**: Wenn du **#$** benutzt, nimm nur zweistellige Zahlen!

So, jetzt weißt du, wie man Sachen in A lädt. Aber noch ist der Wert ja nur in A, du nicht im Spiel! Wie verändert man denn jetzt den Wert, der in RAM-Adressen ist? Ganz einfach…

```
STA $19
```

Der Befehl **STA** („Store Data from Accumulator“ – Daten aus A speichern) nimmt den Wert, der gerade in A ist, und speichert ihn in einer Adresse. Wenn A jetzt zum Beispiel den Wert 01 enthält und du `STA $19` benutzt, wird der Wert 01 in Adresse $19 gespeichert. Hier ein Beispiel:

```
LDA #$01   ; Wir laden den Wert 01 in A...
STA $19    ; ...und speichern ihn in Adresse $19.
```

(Hieran siehst du auch: Alles, was nach einem Semikolon steht, gilt als Kommentar und wird nicht mit ausgeführt.)

Hast du verstanden, was der Code macht? Jetzt hat $19 den Wert 01, und das heißt, dieser Code macht Mario zu Super-Mario.

Auch für STA gibt es Regeln: Du kannst auch hier wieder 4- und 6-stellige Zahlen verwenden, zum Beispiel `STA $0019` und `STA $7E0019`. Was allerdings nicht geht, ist, `STA #$xx` zu benutzen. Denk mal drüber nach: Was für einen Sinn hat es, einen Wert in einem Wert zu speichern? **Merke: #$** bei STA ist Schwachsinn.

Juhuu, jetzt weißt du über LDA und STA Bescheid! Ist das schon genug, um, sagen wir, deinen ersten Custom Block zu machen? Fast. Ein ganz wichtiger Befehl muss noch sein. Aber keine Sorge, er ist ganz einfach:

```
RTS
```

RTS heißt „Return from Subroutine“ – übersetzt etwa „Von Unterroutine zurückkehren“. RTS muss am Ende jedes Codes stehen, um anzuzeigen, dass der Code hier vorbei ist. Du braucht keine Zahl oder sonst was am Ende – einfach RTS, das genügt.

Wie sieht der Code jetzt aus, wenn wir RTS berücksichtigen?

```
LDA #$01   ; Wir laden 01...
STA $19    ; ...in Adresse 19, was Mario zu Super-Mario macht.
RTS        ; Code beenden.
```

Herzlichen Glückwunsch – dein erster kompletter Code!

Custom Blocks erstellen (.bin-Format)

Nur, wie fügt man den jetzt ins Spiel ein? Am einfachsten per Custom Block. Wenn man ihn berührt, wird der Code ausgeführt. Und wie macht man den? Einfach den Code in den Editor tippen und als .bin-Datei speichern geht nicht, das ist schon ein bisschen schwieriger.

Zuerst musst du dir Sprite Tool von SMW Central runterladen. Richtig, du brauchst Sprite Tool, um Blöcke zu erstellen! Na gut, nicht Sprite Tool selbst, sondern, ein Programm, das im Sprite Tool-Ordner mitgeliefert wird: **trasm**.

Um mit trasm einen Block zu erstellen, speichere zunächst deinen Code als .txt-Datei. Dann ziehe diese Datei über das trasm-Symbol, und es werden drei neue Dateien erstellt: Eine .smc-Datei, eine .err-Datei und eine .bin-Datei. Wie du vielleicht schon merkst, ist die .bin-Datei der Custom Block, den du haben willst. Ihn kannst du jetzt mit Blocktool oder BTSD in deinen Hack einfügen. Wie genau man das macht, will ich hier nicht erklären, darum geht’s hier ja auch nicht.

Die .smc- und .err-Dateien kannst du wegschmeißen, sie sind nutzlos.

Verzweigungen

Jetzt hast du einen Block programmiert, der Mario immer zu Super-Mario macht. Aber was, wenn man eine Feuerblume oder ein Cape hat und den Block dann berührt? Das wäre ja eher unvorteilhaft. Wie wär's mit einem Block, der Mario nur dann groß macht, wenn er klein ist? Dazu müsste der Code den Wert aus $19 laden, und je nachdem, welcher Wert es ist, zu anderen Teilen des Codes springen. Hups, das hört sich kompliziert an! Geht das überhaupt? Kann ASM so was? Natürlich! Dazu musst du ein paar neue Befehle lernen:

```
LDA $19    ; Wert aus $19 laden...
CMP #$00   ; ... und mit dem Wert 00 vergleichen.
```

Der Befehl **CMP** steht für „**C**o**mp**are" – Vergleichen. Wie der Name schon sagt, vergleicht er den Wert aus A mit dem Wert, den du festlegst. In diesem Fall ist es 00, und das steht für „kleiner Mario". Der Code oben hieße in Worten „*Wenn der Wert in $19 00 ist…*" oder besser „*Wenn Mario klein ist…*"

CMP kann man natürlich auch mit Adressen statt absoluten Werten verwenden. Du kannst also auch z. B. `CMP $0DBF` benutzen, was dann heißen würde „*Wenn der Wert in $19 und der Wert in $0DBF gleich sind…*"

Gut, jetzt haben wir $19 mit 00 verglichen. Nun brauchen wir noch einen Befehl, der Mario groß macht, wenn die beiden Werte gleich sind, und der gar nichts tut, wenn sie nicht gleich sind (Mario also klein ist). Und dieser Befehl heißt:

```
BEQ
```

BEQ steht für „Branch if Equal" – Springen, wenn gleich. Er springt zu einer bestimmten Stelle im Code, wenn die beiden Werte gleich sind, und führt den Code von dort aus weiter. Nur, wie bestimmen wir die Stelle, zu der gesprungen werden soll? Das ist wirklich überaus einfach. Schauen wir uns mal diesen Beispielcode an:

```
LDA $19          ; Wert aus $19 nehmen...
CMP #$00         ; ... und mit 00 vergleichen.
BEQ SuperMario ; Wenn gleich, springe zum Label „SuperMario".
RTS              ; Ansonsten Code beenden.
```

```
SuperMario:     ; Hier ist das Label „SuperMario“.
LDA #$01        ; Mario zu Super-Mario machen.
STA $19
RTS             ; Code beenden.
```

Verstanden? Nach **BEQ** muss ein Label stehen. Das kann aus Nummern bestehen, aus Zeichen, oder eben aus Worten, wie hier. Du definierst ein Label mit Doppelpunkt dahinter.

Wenn du dieses Label hinter **BEQ** schreibst, heißt das „Wenn die beiden Werte gleich sind, gehe zu diesem Label und mache da weiter, und wenn sie nicht gleich sind, mache von hier aus weiter“. Wenn der Wert von **$19** hier nicht 0 ist, würde einfach bei **RTS** weitergemacht, der Code also beendet. Wenn Mario aber klein ist, würde zum Label **SuperMario** gesprungen und Mario somit groß gemacht.

So, nun weißt du hoffentlich über BEQ Bescheid. Gibt es denn noch andere „Sprung-Befehle“? Oh ja, die gibt’s. Nehmen wir zum Beispiel

BNE

BNE steht für „Branch if not Equal“ – Springen, wenn ungleich. Er ist praktisch das Gegenteil von BEQ – Er springt zu einem Label, wenn die beiden Werte nicht übereinstimmen. Beispiel gefällig?

```
LDA $19
CMP #$00
BNE KleinMachen
RTS
KleinMachen:
STZ $19
RTS
```

Dieser Code macht Mario nur dann klein, wenn er nicht schon klein ist (Wenn also $19 nicht 0 ist). Okay, dieser Code ist etwas unnütz – *Wieso nur klein machen, wenn er nicht klein ist? Da kann man doch lieber einen Block machen, der Mario immer klein macht – kommt auf dasselbe raus und spart Code-Platz!* Na ja, das ist ja nur ein Beispiel.

Einen Sprung-Befehl solltest du noch kennen:

BRA

BRA steht einfach für „Branch“ – Springen. Ihm sind die Werte völlig egal - Er springt immer zu einem Label, und man muss noch nicht mal **CMP** davor benutzen.

Apropos CMP: Wenn du mit 0 vergleichst, kannst du die ganze CMP-Zeile auch weglassen. Beispiel:

```
LDA $19
CMP #$00
BEQ Test
```

kann ersetzt werden durch

```
LDA $19
BEQ Test
```

. Vergiss nicht, das geht nur mit 0!

Es gibt noch ein paar andere Sprung-Befehle, die zeige ich dir später. Diese drei reichen fürs Erste aus.

Scheinbar unnötig – Bytes und dcb

Jetzt kommen wir zu einem Kapitel, das für Sprungbefehle ganz nützlich sein kann und über das wohl jeder Bescheid wissen sollte, der ASM lernt. Okay, es ist trivial, also **könnt ihr’s meinetwegen auch überspringen**. Aber spätestens, wenn man xkas-Patches macht, sollte man sich damit auskennen!

Wenn ihr euren Code (per Custom Block oder was auch immer) in die ROM einfügt, wird der Code in Hex-Zahlen umgewandelt. Jede zweistellige Zahl wird Byte genannt, und jedes Byte steht für einen Befehl oder die Zahlen dahinter. Nicht verstanden? Gut, dann nochmal ein Beispiel. Dieser Code spielt einen Münz-Sound ab:

```
LDA #$01
STA $1DFC
RTS
```

Jetzt macht daraus mal einen Custom Block und öffnet die Datei in einem Hex-Editor. Ihr seht folgendes:

```
A9 01 8D FC 1D 60
```

Das ist euer Code, umgewandelt ins Hex-Format. Jedem Befehl und den Zahlen danach ist ein Byte zugeordnet. Schauen wir uns mal jedes Byte einzeln an:

A9 – Steht für den Befehl **LDA, absoluter Wert, zweistellig**. Wenn ihr stattdessen `LDA $19` (Also eine Adresse, keinen absoluten Wert) oder vier- bzw. sechsstellige Zahlen nehmen würdet, hätte das `LDA` eine andere Hex-Nummer.

01 – Der Wert nach dem `LDA`, in diesem Fall `01`.

8D – Steht für **STA, vierstellige Adresse**.

FC 1D – Die Adresse nach dem `STA`, in diesem Fall `1DFC`. Aber wieso sind die beiden Hälften in umgekehrter Reihenfolge? Das ist bei ASM nun mal so: Vier- bzw. sechsstellige Zahlen werden in zwei bzw. drei Bytes unterteilt und in umgekehrter Reihenfolge geschrieben. Wieso das so ist, weiß ich auch nicht, aber es ist so.

60 – Steht für **RTS**.

Jetzt weißt du also, dass jeder Befehl seine eigene Hex-Zahl hat, und die Werte und Adressen, die danach kommen, sowieso.

Jetzt gehen wir mal einer Frage auf den Grund, die ihr euch sicher auch schon gestellt habt: Wieso hat `LDA #$01` eine andere Hex-Zahl als zum Beispiel `LDA $19` oder `LDA $1DFC`? Wäre es nicht einfacher, wenn jeder Befehl eine Hex-Zahl hätte, egal, was dahinter steht? Damit spart man doch auch Platz! Also echt, blödes Nintendo!

So einfach ist es leider nicht. Wenn man ein bisschen nachdenkt, ahnt man schon, wieso. Also: Nehmen wir mal an, jedes `LDA` hätte das Byte `A9`, egal, was dahinter steht. Aus

```
LDA #$01
```

würde dann `A9 01`. Übersetzen wir das jetzt mal wieder zurück in ASM, stehen wir vor einem Problem: *„A9 heißt LDA, und danach kommt 01. Aber was für ein LDA ist das? LDA $01 oder LDA #$01?“* Vor diesem Problem der Zweideutigkeit würde die Konsole stehen, und zwar nicht nur bei LDA, sondern auch bei allen anderen Befehlen, die verschiedene Dinge hinter ihnen haben. Und es wird noch problematischer:

```
LDA $1DFC
```

würde, wenn man es sich einfach machen würde, zu `A9 FC 1D`. Dann haben wir auch noch das Problem, das wir nicht wissen, ob die Adresse hinter LDA zwei- oder vierstellig ist. *„Heißt es nun LDA $1DFC, oder heißt es LDA $FC, und das 1D ist schon der nächste Befehl? Oder heißt es gar LDA #$FC?“*

Das stiftet mächtig Verwirrung, oder? So eine Konsole weiß ja nicht, was man genau von ihr will, es sei denn, man macht es ihr eindeutig klar. Und das geht eben nur, wenn jeder Befehl je nach Typ und Länge dessen, was hinter ihm steht, ein anderes Byte bekommt.

Übrigens: **LDA, zweistellige Adresse** hat das Byte **A5** und **LDA, vierstellige Adresse** das Byte **AD**. Aber das muss man nun wirklich nicht auswendig können.

Und inwiefern ist das nun, wie oben angekündigt, für Sprung-Befehle nützlich? Nun, hinter BEQ, BNE und so weiter muss nicht unbedingt der Name eines Labels stehen – man kann stattdessen auch die Anzahl an Bytes angeben, die übersprungen werden soll. Hier ein Beispiel. Dieser Code spielt einen Münz-Sound ab, wenn Mario groß ist:

```
LDA $19    ; Wir laden Marios Powerup-Status.
CMP #$01   ; Wenn er Super-Mario ist...
BEQ $01    ; ...überspringe 1 Byte.
RTS        ; Ansonsten mache hier weiter, also RTS.
LDA #$01   ; Wir laden 01 in Adresse 1DFC...
STA $1DFC  ; ...was einen Sound abspielt.
RTS        ; Code beenden.
```

Was nach dem BEQ steht, also 01, steht für die Anzahl an Bytes, die übersprungen werden sollen, wenn Mario Super-Mario ist. Wie du siehst, hat es nur ein **$** davor, kein **#$**. So ist das halt.

Wenn du auf diese Weise arbeitest, musst du natürlich über die Anzahl der Bytes Bescheid wissen, die der Code beansprucht. Noch ein Beispiel: Jetzt soll Sound 1 gespielt werden, wenn Mario groß ist, ansonsten Sound 2.

```
LDA $19    ; Wenn Mario...
CMP #$01   ; ...Super-Mario ist...
BEQ $06    ; ...überspringe 6 Bytes.
LDA #$01   ; \
STA $1DFC  ;  | Ansonsten Sound spielen und Code beenden.
RTS        ; /
LDA #$02   ; \
STA $1DFC  ;  | Anderen Sound spielen und Code beenden.
RTS        ; /
```

Hier muss man wissen, wie viele Bytes der Code **LDA #$01 STA $1DFC RTS** hat, damit man auch die richtige Anzahl angibt. Übersetzen wir es in Hex… **A9 01 8D FC 1D 60** …

sehen wir, dass es 6 Bytes sind. Und das muss man auch hinter das BEQ schreiben, sonst passieren blöde Sachen. Wenn man in den Code oben statt `BEQ $06` beispielsweise `BEQ $04` schreibt, überspringt der Code nur 4 Bytes. Was passiert dann?

Er überspringt die Bytes `A9 01 8D FC` und macht mit `1D 60` weiter. Aber `1D 60` ergibt, zurückübersetzt in ASM, sinnlosen Code. Na ja, zumindest Code, der in diesem Fall sinnlos und unerwünscht ist. **Merke**: Immer schön Bytes zählen!

…Okay, eigentlich ist die ganze Bytes-Geschichte bei Sprungbefehlen **unnötig kompliziert**. Du kannst natürlich weiterhin Labels benutzen, da hab ich überhaupt nichts dagegen. Ich wollte es halt nur mal erklären, für den Fall, dass jemand neugierig ist.

Diesen Teil solltest du dir allerdings durchlesen, denn er ist wirklich wichtig und für fortgeschrittenes ASM relativ unerlässlich: Statt ASM-Code gleich mit Bytes programmieren. Für regulären Otto-Normal-LDA-STA-Code ist das natürlich umständlich, aber warte ab, es wird nochmal recht notwendig sein!

Also, hier kommt der Befehl: `dcb` („**D**ire**c**t **B**ytes“). Achtung: `dcb` ist kein richtiger ASM-Befehl wie LDA oder STA! Er hat keine eigene Hex-Nummer, und er ist auch nicht für die SNES gedacht, sondern für das Programm, das ASM zu Hex macht (Assembler genannt).

Und weil das so ist, ist je nach benutztem Assembler der Befehl anders: Bei trasm (das du schon vom Blöcke-machen kennst) heißt er `dcb`, bei xkas nur `db`. Manche Programme benutzen trasm, manche xkas – Genaueres in einem der letzten Kapitel. Wir benutzen jetzt mal trasm und nennen den Befehl `dcb`.

Also, wofür genau ist er gut? Er fügt, wie sein Name schon sagt, Bytes direkt in den Code ein. Hier mal wieder ein Beispiel:

```
dcb $A9,$01,$8D,$FC,$1D,$60
```

fügt die Bytes `A9 01 8D FC 1D 60` in den Code ein. Er bedeutet also genau dasselbe wie

```
LDA #$01
STA $1DFC
RTS
```

. Es ist nur eine andere, weniger übersichtliche Schreibweise.

Achte auf das Format, wenn du dcb benutzt: Die einzelnen Bytes haben nur ein $ davor und sind durch Kommas getrennt.

Jetzt fragst du dich zu recht: *„Wozu soll ich meinen Code so schreiben? Dazu muss ich erstmal die Bedeutung aller Bytes auswendig kennen, und ich sehe nicht auf den ersten Blick, was der Code bedeutet!“*

Du hast vollkommen recht. Für solch einen Code ist dcb sehr unpraktisch. Wirklich nützlich wird er erst später – Dazu komme ich in einem späteren Kapitel, ich wollte nur, dass ihr jetzt schon drüber Bescheid wisst. Also: Noch etwas Geduld haben!

Ein bisschen Mathe

Jetzt wird's mal wieder etwas praktischer: Wir lernen neue Befehle! Eine ganze Menge sogar. Mit höherer Mathematik hat das Ganze nicht viel zu tun – bloß ein bisschen Plus und Minus. Das kriegt auch der größte Mathemuffel noch hin. Später kommt noch Mal und Geteilt dran, aber ich will euch ja nicht überfordern.

Also, fangen wir mal einfach an:

```
INC
```

Dieser Befehl steht für „**Inc**rease" oder „**Inc**rement" (Erhöhen), und das macht er auch. Er nimmt den Wert aus A und erhöht ihn um 1. Für INC gibt es eine Menge verschiedener Schreibweisen. Die erste sieht so aus:

```
LDA $0DBF  ; Adresse 0DBF enthält die Anzahl an Münzen.
INC        ; Diese erhöhen wir nun um eins...
STA $0DBF ; ...und speichern sie wieder in $0DBF.
```

Nochmal etwas anschaulicher: `LDA $0DBF` lädt den Inhalt der Adresse $0DBF, also die Anzahl der Münzen, die der Spieler gerade hat, in A. Dann kommt der neue Befehl: `INC`. Er nimmt das, was in A ist, und gibt eins dazu. Hätte man also 5 Münzen, wäre jetzt der Wert 06 in A. Und dann wieder das altbekannte `STA` – es nimmt den Wert aus A – also 06 – und speichert ihn wieder in Adresse $0DBF. Kurz gesagt: Dieser Code gibt Mario eine Münze mehr. INC sei Dank!

Das ist, wie gesagt, aber nicht die einzige Schreibweise für INC. Die zweite, nur leicht abgewandelte, sieht so aus:

```
LDA $0DBF  ; Adresse 0DBF enthält die Anzahl an Münzen.
INC  A     ; Diese erhöhen wir nun um eins...
STA $0DBF ; ...und speichern sie wieder in $0DBF.
```

Das **INC A** drückt hier einfach nur noch mal aus, dass es auch der Akkumulator ist, der erhöht werden soll, und nicht etwas anderes. Welche Schreibweise du verwendest, ist eigentlich egal – manche Assembler allerdings wollen eine bestimmte Schreibweise haben. Probier einfach aus, welche ihm in den Kram passt.

So, und dann gibt es noch eine komplett andere Schreibweise für INC:

```
INC $0DBF
```

Diese eine Zeile macht gleich alles auf einmal – wenn man nach dem INC eine Adresse angibt, erhöht er den Wert dieser Adresse. Und das sogar, ohne den Wert in A zu beeinflussen! Man kann also zum Beispiel schreiben:

```
LDA #$01   ; Der Wert in A ist jetzt 01
INC $0DBF ; Der Wert in A ist immer noch 01
STA $19    ; Und 01 speichern wir jetzt in $19.
```

Auch nach dem **INC** ist der Wert in A noch 01. Das ist der Vorteil dieser Schreibweise – Man behält das, was in A ist. Dafür hat die andere Schreibweise den Vorteil, dass man den Wert nicht unbedingt in dieselbe Adresse zurückspeichern muss, sondern damit anstellen kann, was man will. Zum Beispiel könnte man die Anzahl der Münzen um eins erhöhen und sie dann in einer ganz anderen Adresse speichern - $19 zum Beispiel. So hat eben alles seine Vorzüge.

Kapiert? Sehr schön, dann machen wir mal gleich mit dem nächsten Befehl weiter:

```
DEC
```

DEC steht für „**Dec**rease“ oder „**Dec**rement“ (verringern) und schält, wie der Name schon sagt, Bananen.

…Nein, natürlich nicht. **DEC** funktioniert genauso wie **INC**, allerdings verringert er den Wert, statt ihn zu erhöhen. Man kann also alles das schreiben:

```
LDA $0DBF
DEC
STA $0DBF
```

```
LDA $0DBF
DEC  A
STA $0DBF

DEC $0DBF
```

Dieser Code würde die Anzahl der Münzen um eins verringern. Ansonsten gilt alles, was ich auch schon über `INC` gesagt habe, genauso hier. Schön einfach, nicht?

Nun kommt es aber wohl oft vor, dass du einen Wert nicht nur um eins erhöhen willst, sondern um eine andere, viel höhere Zahl, zum Beispiel 100. Muss man dann etwa 100 Mal INC schreiben? Und was, wenn man die Zahl, die addiert werden soll, vom Inhalt einer RAM-Adresse abhängig machen will? Dann hat man ja gar keine festgelegte Anzahl an INCs oder DECs!

Nicht verzagen, für all das gibt es eine Lösung. Klar, INC und DEC sind praktisch, aber wenn das mal nicht reicht, ist das hier deine Wahl:

```
CLC
ADC
```

Zu ihrer Bedeutung kommen wir gleich. Hier siehst du die beiden erstmal in einem Code-Beispiel:

```
LDA $0DBF       ; Anzahl der Münzen laden.
CLC             ; Wir addieren...
ADC #$05        ; ...fünf dazu...
STA $0DBF       ; ...und speichern den Wert wieder in $0DBF.
RTS             ; Code beenden.
```

Vielleicht hast du gedacht, einer der beiden Befehle stünde fürs Addieren und der andere fürs Subtrahieren. Falsch gedacht – beide Befehle sind zum Addieren nötig. Um CLC musst du dir noch keine Gedanken machen, aber merk dir, dass er wichtig ist – wenn du ihn nicht vor jedem ADC verwendest, wird manchmal ein falscher Wert addiert.

Nein, der Befehl, der hier die Hauptarbeit macht, ist **ADC** („**Ad**d with **C**arry“). Wie der Name schon sagt (was „Carry“ bedeutet, musst du noch nicht wissen), nimmt er den Wert, der gerade in A ist, und addiert so viel dazu, wie du willst. Und was du willst, machst du ihm klar, indem du eine Nummer dahinterschreibst. `ADC #$07` würde den Wert 7 hinzufügen, `ADC #$FF` den Wert 255 (denn, wie du weißt, 255 heißt in Hexadezimal FF). Der so veränderte

Wert ist jetzt wieder in A und steht zu deiner Verfügung – zum Speichern in eine RAM-Adresse zum Beispiel.

Du musst auch nicht unbedingt **#$** hinter **ADC** verwenden – du kannst auch den Inhalt einer RAM-Adresse addieren. Dann würde je nachdem, welcher Wert dort gerade ist, etwas anderes addiert werden. Beispiel:

```
LDA $0DBF  ; Die Anzahl der Münzen
CLC        ; wird je nach Powerup
ADC $19    ; um 0, 1, 2 oder 3 erhöht.
STA $0DBF
RTS
```

Wir erinnern uns - $19 enthält Marios Powerup. Ist Mario also klein, passiert gar nichts (da man ja 0 hinzuzählt), ist er Super-Mario, wird 1 hinzugezählt, und so weiter.