

ASM Cookbook

For the special bit(e).

Nun, ich habe mich endlich dazu bequemt dieses Dokument zu verfassen. Es ist nicht fertig und wird es auch nie sein, weil es einfach ein unendlich komplexes Thema ist. Ich weiß, so etwas sagt man nicht am Anfang eines 'Tutorials', eigentlich ist es nämlich ganz einfach.

Nach einer ersten sehr steilen Lernkurve fällt es einem schnell leicht, größere Anwendungen zu schreiben. Ich bleibe mir allerdings treu und werde auch in diesem Tutorial unkonventionellere Wege einschlagen, als es andere Anleitungen tun.

minx

Inhalt

Was ist eine Assemblersprache	2
Warum ASM lernen?	2
Memory	2
Zahlensysteme	4
Dezimal	4
Binär	4
Hexadezimal	5
Bits 'n' Bytes, Nibbles, Words, Doubles	6
Der Stack	7
Segment und Offset	8
Register	9
General Purpose Register	9
Segment Register	10
Index Register	10
Stack Register	11
Come in, freak out	11
ASM Grundlagen I	11
Registeroperationen	11
Stackoperationen	12
Arithmetische Operationen	13
Bitwise Operationen	14
Interrupts	15
ASM Grundlagen II	16
COM Dateiformat	16
Flow Control	17
Schleifen	19
Variablen	20
String Operationen	21
Subs	24
Benutzer Input	25
ASM Grundlagen III	26
Grafik	26
Grafik durch Interrupts	26
Direkt in den VRAM schreiben	27
Dateiverwaltung	29
File Handles	29
Dateien lesen	30
Schlusswort 16bit Assembler	32
32bit Assembler	32
LASM	32
Los geht's	33
Weiter geht's	34
Weitere ASM 8086 Befehle	41
Inline Assembler in Autolt	42
Autolt Funktionen im ASM Code	42
Autolt Variablen im ASM Code	43
CUID lesen / Übergabe von Structs / bearbeiten von Structs	43
ASM-Inline wie der Führer eines Geschäftes	44
64bit Assembler	44
Architektur	44
SIMD	46
Referenzen	47

Was ist eine Assemblersprache?

Der, die, das Assembler. Es hat unterschiedliche Bedeutungen. Eine Assemblersprache ist zunächst stets eine Low-Level Sprache (Sprache der (ersten/)/zweiten Generation), das heißt im Klartext, dass es die hardwarenächste Sprache ist, in welcher man Programme verfassen kann. Diese Assemblersprache bietet dem Programmierer die Möglichkeit, den mindestens gesamten Befehlssatz des genutzten Prozessors zu verwenden. Bei bekannten kompilierten Programmiersprachen wie C oder BASIC wird der lesbare Drittgenerationscode in ASM Code umgewandelt, von einem Assembler *assembliert* und von einem Linker mit Systemabhängigkeiten oder Bibliotheken *gelinkt*. 'Der Assembler' ist das Programm, welches aus ASM Code den für den Prozessor verständlichen Microcode, oder auch OP-Code erstellt. Dieser OP-Code lässt sich in Drittgenerationssprachen inline nutzen oder im Zusammenspiel verschiedener Header z.B. als Portable Executable (PE / COFF) als ausführbare Datei speichern. Neben den Standard-ASM-Befehlen können spezielle ASM Dialekte, vor allem Macro-enabled Assembler komplexere Konstrukte wie FOR-Schleifen verarbeiten.

Warum ASM lernen?

Wenn man heute Jemanden fragt was der Vorteil sei, Assembler zu lernen und vor allem zu nutzen, wird man oft auf die höhere Geschwindigkeit verwiesen. Das ist nur halbrichtig. Frühere Versionen von BASIC und die ersten Versionen von C warfen derart unoptimierten und langsamen Code aus, dass es einen erheblichen Unterschied bereitete schlichtweg gleich ASM zu nutzen. Heute allerdings optimieren die Compiler den ASM Code extrem, so dass es eigentlich überflüssig wird einfach nur ASM zu nutzen. Zumal es mit den unzähligen Funktionen der WinAPI, die sich auch in 32Bit ASM nutzen lassen nicht mehr viel gibt, was man eigentlich noch überhaupt selbst schreiben muss. Warum sollte man nun eigentlich ASM lernen?

- ASM zu lernen verstärkt erheblich das Wissen um die Funktionsweise eines PCs, speziell der CPU
- Wenn Windows einen Fehler feststellt, dann gibt es kryptische hexadezimale Adressen oder Befehle aus. Wäre es nicht gut zu wissen, was der Fehler war und wie man ihn vielleicht behebt?
- ASM Code zu verfassen lässt einen die volle Kontrolle über die Aktionen seines Programms erlangen. Bei Drittgenerationssprachen ist man immer dem vielleicht fehlerhaften Code-Generator untergeordnet
- Es macht einfach Spaß extrem große mathematische und/oder grafische Anwendungen in extrem wenig Code zu bringen. Mit genug Wissen und Ambition lassen sich beeindruckende Ergebnisse erzeugen, die auch geschätzt werden (z.B. auf sogenannten Demopartys wie Revision, Tokio Demo Fest, Function in Kategorien wie 4k, bei der es darum geht ein möglichst beeindruckendes Programm in 4096 Byte zu packen)
- Zudem macht es besonderen Spaß Windows auszutricksen. Ich werde am Ende noch einmal auf das Demoschreiben eingehen und wie man Programme so unerhört klein bekommt

Memory

In diesem Kapitel möchte ich dir von Grund auf eine neue Perspektive auf den PC eröffnen. Für viele sind PCs nur "magische" Kästen mit denen man spielen oder Officekram erledigen kann. Aber eigentlich sind es nur größere Taschenrechner, denn nichts anderes als Bit-Manipulationen sind möglich. Die einzige

mathematische Operation die so machbar wird, ist die Addition. Subtraktion ist bloß Addition mit negativen Summanden, Multiplikation ist simple wiederholte Addition und Division ist eigentlich bloß die wiederholte Addition negativer Zahlen.

Zahlensysteme

Dir ist mindestens ein Zahlensystem bekannt: Das Dezimalsystem. Um zu begreifen was im PC abläuft brauchen wir noch zwei andere Systeme: Das Hexadezimale und das Binäre. Zu Beginn werfen wir nochmal einen Blick auf das uns bekannte System.

Dezimal

Das Dezimale Zahlensystem ist Basis-10 System. Das bedeutet, dass es aus 10 verschiedenen Ziffern besteht (0...9). Untersuchen wir mal die Zahl 125 als Beispiel:

	Hunderter	Zehner	Einer
Ziffer	1	2	5
Berechnung	1×10^2	2×10^1	5×10^0
Wert	100	20	5

Binär

Das Binärsystem sieht zunächst viel komplizierter als das alltägliche Dezimalsystem aus, ist aber eigentlich einfacher. Denn es ist ein Basis-2 System (0...1). Die Zahlen eines System errechnen sich durch

Wert $\times$ Basis $^$ Position

Hier ist die Basis 2. Gehen wir damit mal ein Beispiel durch:

Binär: 10110

$1 \times 2^4 = 16$
 $0 \times 2^3 = 0$
 $1 \times 2^2 = 4$
 $1 \times 2^1 = 2$
 $0 \times 2^0 = 0$

Dezimal: 22

Ziemlich einfach, nicht? Dann machen wir das Ganze mal rückwärts, von Dezimal zu binär:

Dezimal: 238

$238 / 2$ Rest: 0
 $119 / 2$ Rest: 1
 $59 / 2$ Rest: 1
 $29 / 2$ Rest: 1
 $14 / 2$ Rest: 0
 $7 / 2$ Rest: 1
 $3 / 2$ Rest: 1
 $1 / 2$ Rest: 1
 $0 / 2$ Rest: 0

Binär: 11101110

Bei der Konvertierung wird also folgendermaßen vorgegangen:

1. Dividiere die Ausgangszahl durch 2, wenn es sich Restlos teilen lässt, dann ist der Rest 0 (BOOLSch gesehen: Kein Rest, also FALSCH, dann 0, sonst 1)
2. Dividiere das Ergebnis der ersten Division wieder durch 2. Wenn ein Rest bleibt, dann wird eine 1 vermerkt.
3. Wichtiger Schritt: Bei einem Ergebnis, bei welchem eigentlich ein Rest blieb, wird die Zahl für die nächste Division **abgerundet**. (siehe Beispiel '59')
4. Wiederholte Division, bis das Zwischenergebnis 0 ist

Übrigens: Beim Schreiben von Binärzahlen bleibt man bei der Länge eigentlich immer im Bereich der Vielfachen von 4. Das heißt aus 11001 wird 00011001. Das Zahlensystem wird immer durch einen kleinen Buchstaben gekennzeichnet. Das heißt die perfekte Schreibweise wäre: 00011001b.

Hexadezimal

Vielleicht ist es dir schon einmal aufgefallen, dass zum Beispiel bei Speichergrößen immer Vielfache von 2 angegeben werden. Du wirst z.B. 1024 Megabyte RAM finden, aber niemals 1023. Computer rechnen am liebsten in Vielfachen von 2, 4, 8, 16, 32, usw.

Da kommt das Hexadezimalsystem ins Spiel. Weil es ein Basis-16 System ist, ist es perfekt für diese Rechnungen. Wenn du das Binärsystem verstanden hast, dann wird das hier für dich kein Problem sein. Schauen wir uns mal die Zahlen des Hexadezimalsystem im Vergleich zu den zwei anderen System an:

Hexadezimal	Dezimal	Binär
0h	0	0000b
1h	1	0001b
2h	2	0010b
3h	3	0011b
4h	4	0100b
5h	5	0101b
6h	6	0110b
7h	7	0111b
8h	8	1000b
9h	9	1001b
Ah	10	1010b <- Nach 9 mit Buchstaben
Bh	11	1011b
Ch	12	1100b
Dh	13	1101b
Eh	14	1110b
Fh	15	1111b

Das h gehört nicht zur Zahl, sondern ist wieder Symbol für das Zahlensystem. Wieder betrachten wir die Konvertierung.

Weil das Hexadezimalsystem mit Buchstaben rechnet, müssen wir diese zunächst in das Dezimalsystem konvertieren, damit wir unsere Formel

Wert x Basis ^ Position

anwenden können:

Hex: 2A4F

F	x	16 ⁰	=	15	x	1	=	15	
4	x	16 ¹	=	4	x	16	=	64	
A	x	16 ²	=	10	x	256	=	2560	
2	x	16 ³	=	2	x	4096	=	8192	Summe bilden

Dez: 10831

Die Arbeitsschritte sind hier:

1. Die Hexadezimalzahl wird von hinten her aufgeschrieben
2. Alle Hex-Ziffern werden zu Dez-Ziffern konvertiert und die Basis-Formel angewendet
3. Letztendlich wird die Summe aller Zwischenergebnisse gebildet

Und andersherum:

Dez: 1324

1324	/	16	=	82.75
82	x	16	=	1312
1324	-	1312	=	12 = Ch

82	/	16	=	5.125
5	x	16	=	80
82	-	80	=	2 = 2h

5	/	16	=	0.3125
0	x	16	=	0
5	-	0	=	5 = 5h

Hex: 052Ch

Übrigens:

- Hex ist die Abkürzung für Hexadezimal (ich wurde da schonmal gefragt...)
- Wie für Binärzahlen gilt auch hier die Längenregel mit Vielfachen von 4
- In Assemblersprachen werden den Hexzahlen Präfixe vorangestellt, sonst gibts Fehler. In einigen Dialekten wird aus DEADh dann &HDEAD oder 0xDEAD. Das ist immer in der Referenz des Assemblers vermerkt.

Bits 'n' Bytes, Nibbles, Words, Doubles

Bits sind die kleinste Dateneinheit eines Computers. Ein Bit hat zwei Zustände: AN / AUS oder TRUE / FALSE oder eben 1 / 0. Bits allein sind eigentlich nutzlos, denn groß rechnen könnte man damit nicht. Deswegen gibt es die sogenannten *Nibbles*. Nibbles bestehen jeweils aus 4 Bits. Wenn wir uns nun die obrige Tabelle der Zahlen der verschiedenen System ins Gedächtnis rufen, so erkennen wir, dass durch ein Nibble jede Hexadezimale Ziffer binär dargestellt werden kann. Das ist auch schon der Sinn und Zweck der Nibbles.

Über den nibbles steht die wichtigste Dateneinheit eines PCs: Das *Byte*. Das Byte ist tatsächlich die kleinste Einheit, die von einem Prozessor gelesen und verarbeitet werden kann. Ein Byte besteht aus 8 Bits

oder 2 Nibbles. Alles was du irgendwo speicherst, sei es auf einer Festplatte oder einer CD wird in Bytes gespeichert und gelesen.

Sagen wir mal du speicherst die Zahl 170 z.B. in einer Textdatei auf deiner HDD, dann sähe das so aus:

Bits	1	0	1	0	1	0	1	0
Bitposition	7	6	5	4	3	2	1	0
Nibbles	HO Nibble				LO Nibble			

10101010 ist 170 binär geschrieben. Die Bits 0..3 werden als Low-Order Nibble bezeichnet, die höheren Bits dann logischerweise als High-Order Nibble. Gut, da habe ich mir jetzt den Optimalfall für zwei Nibbles rausgesucht, deswegen noch ein Beispiel für 43690:

1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
High Order Byte								Low Order Byte							

Was ist nun ein Double? Genau genommen ist es eine ganze 32Bit lange Zahl. Diesen Datentyp nennt man 32Bit Integer oder simpel: *Double*. Er besteht aus 32Bit, also 8 Nibbles, also 4 Bytes und 2 sogenannten *Words*.

Jetzt haben wir also schon die Datentypen BYTE, WORD und DWORD (Double) kennengelernt. Diese und andere Typen können wir später im ASM Code benutzen.

Der Stack

Beim Durchstöbern des Internats nach ASM Tutorials oder was auch immer ist dir bestimmt schon oft der Begriff 'Stack' begegnet. Ob schon erklärt oder nicht, ich mache es noch einmal.

Den Stack kann man sich am Besten als eine Art Bücherstapel vorstellen (Stack = Stapel). Du legst ein Buch obendrauf und es wird auch wieder das erste sein, wenn du es herunternimmst. Etwas auf den Stack zu schieben (also zu legen) nennt sich "pushen". Dem nachempfunden ist der ASM Befehl "push". Etwas vom Stack zu nehmen nennt sich "popen". Der zugehörige Befehl ist "pop".

Sagen wir mal du hast 5 Bücher. Die heißen A, B, C, D, und E und dein Stapel sieht so aus:

```

+--+
| A |
+--+
| B |
+--+
| C |
+--+
| D |
+--+
| E |

```

+--+

Jetzt nimmst du ein hypothetisches Buch "F" und legst es auf den Stapel:

```
+--+
|F| <- Buch liegt jetzt als erstes neues auf dem Stapel
+--+
|A|
+--+
|B|
+--+
|C|
+--+
|D|
+--+
|E|
+--+
```

Wenn du jetzt den Stack popst (hör auf zu lachen...), dann hast du nur das Buch F zurück:

```
+--+
|A|  - \
+--+  |
|B|   |      *Pop-Geräusch einfügen*
+--+  |
|C|   |
+--+  |
|D|   |
+--+  +-+
|E|   |F|
+--+  +-+
```

Dieses Prinzip nennt man kurz LIFO (Last In, First Out). Wozu wird jetzt der Stack benutzt? Sozusagen als temporäres Notizbuch für die Zwischenspeicherung von Daten.

Segment und Offset

Hardwaremäßig kommunizieren alle Teile deines PCs durch einen sogenannten BUS. Dieser BUS hat immer eine spezielle Breite. Am Meisten kommunizieren CPU und RAM. Der RAM-BUS ist 16Bit breit. Das heißt, wenn die CPU etwas in den RAM schreiben möchte, dann wird immer ein 16Bit Packet gesendet. Früher konnten die CPUs somit nur 65535 Bytes Speicher nutzen ($16\text{Bit} = 1111111111111111 = 65535\text{Byte}$). Später wurde dann der Weg eingeschlagen, mit einem Mal 20Bit an Daten, also 1MegaByte zu senden. Bei dieser neuen Variante ist der Speicher in Segmente eingeteilt, die einen Bund von Bytes darstellen. Zugriff auf diese Segmente kann über das Offset der Bytes in einem Segment erfolgen.

Wenn also die CPU auf den Speicher zugreifen will, so sendet sie zunächst die Segment-Nummer gefolgt von der Offset-Nummer. Beispielsweise will die CPU auf die Speicherstelle 1234:4321 zugreifen. Auf diese Anfrage sendet der RAM die Daten in Segment 1234 am Offset 4321 also das 4321. Byte im 1234. Segment. Man kann sich das also wie ein zweidimensionales Koordinatensystem vorstellen.

Um also eine physikalische Speicheradresse zu erhalten rechnen wir:

$\text{Segment} \times 10\text{h} + \text{Offset}$

Wollen wir also die Adresse von 1000:1234 rausfinden, dann setzen wir ein und erhalten 11234.

Diese Methode ist ziemlich einfach, aber auch veraltet. Bei neuen CPUs gibt es den sogenannten "Protected Mode". Dabei nutzt die CPU eine LookUp-Tabelle um die physikalische Adresse zu berechnen. Somit kann weit mehr als 1MB Speicher angesprochen werden. Der alte Modus nennt sich "Real Mode".

Register

Ganz essentiell und wichtig für die Arbeit mit Assembler sind Register. In jeder CPU gibt es kleine Speicherbereiche. Sie werden im Verlauf eines normalen Programm meist unheimlich oft mit vielen verschiedenen Daten beschrieben, obwohl sie nur winzige Datenmengen halten können. Es gibt verschiedene Arten von Registern:

General Purpose Register

Alle GP Register sind 16Bit breit und können in zwei 8Bit Register unterteilt werden. Zum Beispiel für das Register AX:

```
+-----+
|   AX   |
+-----+-----+
| AH | AL |
+-----+-----+
```

Wenn wir also z.B. DEADh in AX schreiben, enthält AH (H = High) DEh und AL (L = Low) ADh. Man kann auch nur die H und L Register einzeln beschreiben, der Gesamtwert wird auch in X übernommen.

```
+-----+-----+
| AX | Akkumulator |
+-----+-----+
| Für Mathematische Operationen, IO und INT 21 |
+-----+-----+

+-----+-----+
| BX | Basis |
+-----+-----+
| Für Basis oder Pointer |
+-----+-----+

+-----+-----+
| CX | Counter |
+-----+-----+
| Für Schleifen und repetetive Operationen |
+-----+-----+

+-----+-----+
| DX | Displacement |
+-----+-----+
| Alles mögliche und Ausgabe |
+-----+-----+
```

Aktuelle CPUs (seit dem 386) benutzen zudem 32Bit breite Register. Extended Registers heißen diese. Deswegen werden sie auch mit dem Präfix E geschrieben. E[A...D]X:

+-----+			
	EAX		
+-----+			
	AX		
+-----+			
	AH	AL	
+-----+			

Bitte beachten, dass es kein EAH oder EAL Register gibt! Weiter gehts.

Segment Register

+-----+		
CS	Code Segment	
+-----+		
Speicherzelle die Code enthält		
+-----+		

+-----+	
DS	Data Segment
+-----+	
	Speicherzelle die Daten enthält
+-----+	

+-----+		
ES	Extra Segment	
+-----+		
Hauptsächlich für Video-Operationen benutzt		
+-----+		

+-----+	
SS	Stack Segment
+-----+	
Enthält stets die zurückgegebene Adresse aufgerufener	
Routinen	
+-----+	

Index Register

+-----+	
SI	Source Index
+-----+	
Enthält die Quelle eines Strings oder Arrays	
+-----+	

+-----+	
DI	Ziel Index
+-----+	
Enthält das Ziel eines Strings oder Arrays	
+-----+	

+-----+	
IP	Instruction Pointer
+-----+	
Enthält die Adresse des nächsten Befehl (nicht be-	schreibbar
schreibbar	
+-----+	

Stack Register

+	----	+	-----	+
	BP		Base Pointer	
+	----	+	-----	+
	Wird in Verb. mit SP für Stackmanipulationen benutzt			
+	-----	+	-----	+
+	----	+	-----	+
	SP		Stack Pointer	
+	----	+	-----	+
	Enthält das Ziel eines Strings oder Arrays			
+	-----	+	-----	+

Das Register IP hält genauer gesagt das Offset des ausgeführten Befehls.

Ein paar Worte zu Flags. Sie unterscheiden sich von allen anderen Registern durch ihre Größe, die sind nämlich nur 1Bit breit. Genauer sind sie eigentlich BOOLSche Register. Es gibt verschiedene Flagregister, darunter Carry Flag, Overflow Flag, Parity Flag, Direction Flag und viele mehr.

Du wirst niemals selbst Werte in Flagregister schreiben, sie werden automatisch von ausgeführten Instruktionen beeinflusst. Sie dienen somit lediglich der Information. Sagen wir beispielsweise du möchtest BX mit CX vergleichen. Ist es gleich, dann springt ein Flagregister auf 1.

Übung: Lern die Register auswendig. Stets und ständig dieses Dokument zu öffnen ist nicht Sinn des Lernens der Sprache. Über die Umwandlung von Zahlensystemen wenn nötig.

Come in, freak out

Endlich haben wir das mist Theoriekram abgearbeitet und können loslegen. Ich baue das von 16Bit Code auf. Das wird keinesfalls mehr für 64Bit Systeme funktionieren und überhaupt gebe ich nur den Code vor. Ich stelle eine Menge Beispielcode vor. Im Abschnitt 16Bit kann eigentlich keiner davon mit irgendeinem Assembler kompiliert werden, denn er ist nur zu Anschauungszwecken gedacht, aber keine Angst, ich stelle dir am Ende ein komplettes Packet mit Beispielen, sowie den ausführbaren Dateien bereit, du brauchst dir also nicht extra einen 16Bit Assembler besorgen. Im 32Bit Abschnitt erörtere ich alles anhand meines Assemblers LASM (den Grund dafür nenne ich später). Dort verwende ich ausschließlich gültigen ASM Code. Auch hier stelle ich trotzdem noch ein Komplettpacket bereit.

Dateien die Kapitel Autolt-Inline LASM und Democoding betreffend habe ich ebenfalls jeweils zu einer downloadbaren ZIP zusammengefasst. Lies dir aber verdammt nochmal das ganze Tutorial von vorne bis hinten durch. Jedes Kapitel baut auf anderen auf und Hin&Her-Gehopse ist das Sinnloseste was man bei einem Tutorial machen kann. Also lade dir auch nicht gleich die Archive runter und pfusche unwissend im Code rum um dann später mir Fragen zu stellen, die im Text schon beantwortet sind. Das habe ich beim OpenGL Tutorial schon durch und beantworte solche Fragen jetzt schlicht nicht mehr.

ASM Grundlagen I

Registeroperationen

Du weißt bereits was Register sind, aber noch nicht wie du sie manipulierst. In Assembler bewegt man Daten aus einer Quelle in ein Ziel mit dem Befehl "mov":

```
MOV Ziel, Quelle
```

Zum Beispiel möchte ich 56h nach AX bewegen:

```
MOV AX, 56h
```

Man kann auch Registerwerte verschieben:

```
MOV AX, BX
```

Oder auf die H und L Register zugreifen:

```
MOV AL, 9
```

Jetzt enthält AL 09 und AX 0009.

MOV ist so ziemlich die wichtigste Operation.

Der nächste wichtige Befehl ist XCHG, welcher zwei Register in ihren Werten tauscht:

```
XCHG Register1, Register2
```

Zum Beispiel:

```
MOV DX, 56h
```

```
MOV AX, 3Fh
```

```
XCHG DX, AX
```

Jetzt enthält DX 3Fh und AX 56h. Beachte dabei, dass man keine H oder L mit X Registern tauschen kann.

Weiter gehts mit primitiven mathematischen Operationen. Zunächst INC und DEC. INC addiert 1 und DEC subtrahiert 1:

```
MOV DX, 50h
```

```
INC DX
```

Nun ist DX = 51h.

Stackoperationen

Jetzt wollen wir mal anwenden, was wir schon über den Stack wissen. Es gibt 6 Stack-Befehle von denen du nur 2 überhaupt benutzen wirst:

```
PUSH Thing
```

```
POP Thing
```

Sagen wir mal, wir möchten den Wert aus AX temporär speichern, damit zum Einen AX wieder frei ist, uns der Wert zum Anderen aber auch erhalten bleibt. Wir schreiben AX also einfach auf den Stack:

```
PUSH AX
```

Jetzt können wir mit AX wieder machen was wir wollen. Um den alten Wert wiederherzustellen POPen wir den Stack:

```
POP AX
```

Der Stack akzeptiert nur 16Bit und 32Bit Register. Sollte kein Problem sein, da sich H und L automatisch an die übergeordneten Register anpassen.

Üben wir das mal an ein wenig Code. Versuch mal rauszubekommen, was am Ende in AX und BX steht.

```
MOV  AX, 51h
MOV  BX, 4Fh
XCHG AX, BX
PUSH AX
MOV  AX, 34h
POP  BX
PUSH BX
POP  AX
```

Wenn du auf AX = 4Fh und BX = 4Fh kommst, dann: Sehr gut! Wenn nicht, dann lies nochmal weiter oben, bis du den Code verstehst. Du kannst dir auch Notizen machen.

Jetzt lernen wir noch eine Variation der Standard-Stack Befehle kennen: PUSHF und (wer hätte's gedacht) POPF. Diese Operatoren pushen das Flag Register auf den Stack. Sie sind Operandenlos:

```
PUSHF
POPF
```

Sagen wir mal, wir wollen das aktuelle Flagregister nach AX verschieben:

```
PUSHF
POP  AX
```

Die letzten zwei Stackbefehle sind PUSHA und POPA. PUSHA schiebt alle GP Register auf den Stack und POPA fordert alle GP Register vom Stack an.

Beachte: PUSHA und POPA sind 32Bit Operationen, das heißt sie funktionieren weder auf 16Bit System noch mit 16Bit Assemblern (also auch nicht in COM Dateien):

```
MOV  AX, 1h
MOV  BX, 2h
MOV  CX, 3h
MOV  DX, 4h
PUSHA
MOV  AX, 5h
MOV  BX, 6h
MOV  CX, 7h
MOV  DX, 8h
POPA
```

Am Ende des Programms ist eigentlich nichts passiert, alle Register enthalten wieder den Startwert vor PUSHA.

Arithmetische Operationen

Jeder liebt Mathe. Besonders binäre oder hexadezimale Mathematik! Du wirst jetzt die vier Grundrechenarten in ASM kennenlernen: ADD, SUB, MUL und natürlich DIV.

```
ADD Register1, Wert/Register2
Das Ergebnis wird in Register1 gespeichert.
```

```
SUB Register1, Wert/Register2
```

Das Ergebnis wird in Register1 gespeichert.

`MUL Register/Wert`

Das Ergebnis wird in AX gespeichert, wenn Register ein 16Bit Register ist. Das Ergebnis wird in AH oder AL gespeichert, wenn Register ein 8Bit Register ist. Das Ergebnis wird in EAX gespeichert, wenn Register ein 32Bit Register ist, oder bei 32Bit Assembler immer bei einem beliebigem Wert:

```
MOV AX, 5h
MOV BX, 4Fh
MUL BX
```

Ergebnis ist $AX = 18B$ ($5h \times 4Fh$).

`DIV Register/Wert`

Es gelten die selben Regeln wie für MUL:

```
MOV AX, 5h
MOV BX, 4Fh
DIV BX
```

Ergebnis ist $AX = Fh$ ($4Fh / 5h = Fh$), da abgerundet wird:

```
4Fh = 79
5h  = 5
79  / 5 = 15.8
15  = Fh
```

MUL und DIV haben komplizierte Bedingungen wie du siehst. Es ist zwar nicht so gewollt, aber, dass diese Operatoren so schwer zu benutzen sind hat einen guten Nebeneffekt. Nutze niemals MUL oder DIV! Es ist einfach, aber saulangsam, für grafische Manipulationen absolut unbrauchbar. Für solche Zwecke ist Shifting mit ADD und SUB deutlich besser geeignet und kann bis zu 5000% schnellere Ergebnisse erzielen. Darauf gehe ich später noch einmal ein. Für deine ersten Anwendungen ist das aber sicher nicht relevant.

Bitwise Operationen (dazu auch mein Tutorial "BitOR und Konsorten")

Es gibt 4 grundlegende Bitwise Operationen in Assembler: AND, OR, XOR und NOT.

Für AND:

```
AND Register1, Register2
AND Register, Wert
```

Ein Beispiel dazu:

```
MOV AX, 5h
MOV BX, 6h
AND AX, BX
```

Das Ergebnis ist $AX = 4$. Woha! Wie denn das (wird sich jemand denken, der besagtes Tutorial nicht gelesen hat). Ganz einfach:

```
Nach binär:
5h = 101b
6h = 110b
```

BOOLsche Operation durchführen:
101b AND 110b = 100b

Nach hexadezimal:
100b = 4h

Für OR:

OR Register1, Register2
OR Register, Wert

Beispiel:

MOV AX, 5h
MOV BX, 6h
OR AX, BX

Ergebnis: AX = 7h

Der selbe Syntax wird auf XOR appliziert.

Und zuletzt NOT. NOT negiert jedes Bit des gegebenen Werts oder Registerwerts:

NOT Register
NOT Wert

Beispiel:

MOV AX, F0h
NOT AX

Ergebnis: AX = Fh. Weil:

NOT F0h = NOT 11110000 = 00001111 = Fh

Interrupts

Interrupts sind die nützlichsten Dinge in Assembler. Ein Interrupt, sagt der Name ja schon, unterbricht die normale Ausführung eines Programms um etwas ... anderes zu tun. Was, bestimmt der Interruptcode. Die Interrupts die du benutzen wirst, wenn überhaupt, sind Software-Interrupts. Ein Beispiel dafür wäre das Schreiben oder Lesen einer Datei. Das wäre ein 16Bit System OS Interrupt, weil es vom OS ausgeführt wird. Andere Interrupts werden durch andere Software behandelt. Als Beispiel lässt sich der BIOS oder die GPU nehmen, die alle eigene Interrupts besitzen um spezielle Aktionen auszuführen (Pixel einfärben oder Ähnliches). Also wie weiß der PC welcher Interrupt was bewirkt? Jeder Interrupt wird durch einen numerischen Wert symbolisiert und in einer Interrupt Vector Table, kurz IVT gespeichert. Die IVT findet sich ab 000:000. Alle Interrupt handles sind 32Bit breit. Also findet sich Interrupt 1h bei 0000:0004. Der bekannteste Interrupt ist 21h und findet sich bei 0000:0084.

Wie benutzen wir nun Interrupts? Ganz einfach so:

INT Interrupt

Zum Beispiel

INT 20h

Dieser Interrupt unterbricht das Programm komplett, unternimmt nichts weiter und kehrt zum

Betriebssystem zurück.

Einige Interrupts wie dieses haben nur eine Funktion, aber andere können gleich mehrere erledigen. Da wir keine Parameter übergeben können, müssen wir AX manipulieren, damit es vom Interrupt ausgelesen werden kann und als Parameter genutzt wird:

```
MOV AH, 02
MOV DL, 41
INT 21
INT 20
```

Also was passiert hier? Ganz einfach: Das Programm gibt den Buchstaben A aus. Aber wie?

Das Interrupt 21 ist eine Funktionssammlung. Mit der ersten Zeile bestimmen wir die Funktion, die wir aufrufen wollen, hier die Textausgabe. Dann bewegen wir den Code für A nämlich 41 nach DL. Interrupt 21 zeigt den Buchstaben dann an und 20 kehrt zum OS zurück.

Das ist total verwirrend für einen Anfänger. Ich hab im Anhang noch ein paar Links dazu, also keine Panik.

Jetzt kannst du ein 4 Zeilen Programm schreiben, was kompiliert in 7 Byte passt. Das selbe ließe sich in BASIC in einer Zeile erledigen, erzeugt aber unverständlicherweise eine >10K große Datei! Und wer will denn, dass seine 1TB Festplatte **so schnell** voll wird - unerhört!

OK, Sarkasmusmodus aus und wieder ans Üben:

1. Erstell ein Programm, das den Wert aus AX ausliest, ihn nach DX und BX schreibt und dann die Werte in DX und BX multipliziert und das Ergebnis in CX speichert!
2. Erstelle ein Programm, welches AUTOIT ausgibt und sich beendet (A = 41, B = 42 etc.)!
3. Erstelle ein Programm, das alle bekannten Bitwise Operationen mit den Werten Bh und 4Ah durchführt!

Jetzt kannst du schon ein wenig Assembler. Eine exzellente Befehlsreferenz für die aktuellen 8086 CPUs (auf Deutsch) findest du im Anhang! Probier einfach ein wenig rum, so lernst du am Meisten.

ASM Grundlagen II

COM Dateiformat

COM steht für **CO**re **iM**age. Als Image = Abbild repräsentiert diese Binärdatei, was genau später im RAM stehen soll. COM Files sind "EXEn" von DOS, und sie funktionieren heute auch noch auf Windows7 (außer 64Bit). Nur ein paar Interrupts sind verschwunden, aber für sowas gibts ja DOSBox.

Beim Ausführen einer COM sucht das OS zunächst nach den Records in der Datei und prüft, ob ein 64k Block Speicher vorhanden ist. Dieser Block muss für das Programm frei sein, da sonst fremder Speicher überschrieben wird (ab Vista geregelt durch DEP Policy). Als nächstes erstellt das OS einen Program Segment Prefix (PSP), welcher 256 Bytes lang ist und wie folgt aussieht:

Adresse	Beschreibung
---------	--------------

00h-01h	Befehl der das Programm beendet (meist Interrupt 20h)
02h-03h	Segmentpointer auf den nächsten freien Speicherblock
04h	Reserviertes 0 Byte
05h-09h	Far call zum DOS Dispatcher
0Ah-0Dh	INT 22h vector (Beende Programm)
0Eh-11h	INT 23h vector (Ctrl+C Management)
12h-15h	INT 24h vector (Kritischer Error)
16h-17h	PSP Segment des Mutterprozesses
18h-2Bh	Pointer zur Dateibehandlung
2Ch-2Dh	DOS Segment
2Eh-31h	SS:SP Speicherplatz
32h-33h	Zahl der Dateihandles
34h-37h	Pointer zur Filehandle-Tabelle
40h-41h	OS Version
5Ch-6Bh	File Block 1
6Ch-7Bh	File Block 2
7Ch-7Fh	Reserviert
80h	Länge des Parameterbuffers
81h-FFh	Disk Transfer Area

Danach werden DS, ES und SS mit dem Pointer zum Speicherblock beschrieben, SP wird auf den Initialwert FFFFh gesetzt, auf den Stack wird 0000h gepusht (d.h. der Stack ist leer) und CS enthält den Pointer zum aktuellen Speichersegment, während IP das Offset 0100h speichert. Der Speicherpointer liegt also auf CS:IP.

Der PSP ist eigentlich 255 Byte lang, d.h. er passt in ein Speichersegment. Das wiederum heißt, dass es gültiges COM Programm nicht größer als 65280 Byte sein darf. Aber ehe du soviel Geduld und Wissen hast, um ASM Programme dieser Größe zu erstellen, bist du bereits bereit für eine EXE.

Wozu brauchst du diese Informationen? Nun, wie jeden Speicherbereich kannst du auch den PSP bearbeiten und auslesen. Wenn wir später zu Dateioperationen kommen, arbeiten wir mit DTA. Oder du willst einfach die Betriebssystemversion, dann liest du diese aus 40h-41h.

Flow Control

Flow Control Operationen sind genau das: sie kontrollieren oder bestimmen den Flow des Codes. Also wie wird wann welcher Code ausgeführt. In Drittgenerationssprachen haben wir da IF/THEN oder SWITCH oder andere hochkomplexe Konstrukte. In ASM sieht das richtig kompliziert aus, aber eigentlich ist es richtig einfach. Kenner und (verantwortungsvolle) Nutzer des GOTO Befehls in anderen Sprachen werden sich an ASM Flow Control keinen Zahn ausbeißen.

Um mal sowas wie IF/THEN zu realisieren lernen wir drei neue Dinge kennen: Labels, Compare- und Jump Operationen. Rufen wir uns noch einmal das Programm ins Gedächtnis, welches nur ein A ausgibt. Wir fügen jetzt dort ein Label ein um einen etwas aufgeräumteren Code zu bekommen:

```
(01) MOV AH, 02h
(02) MOV DL, 41h
(03) INT 21h

(04) JMP EXIT

(05) MOV AL, 30h

(06) EXIT:
```

(07) INT 20h

Ich habe einmal Zeilennummern eingefügt um das ganze besser erläutern zu können, diese zählen logischerweise nicht zum ASM Code. Zeile 1 bis 3 sind uns wohl bekannt und wir wissen, was passiert. In Zeile 4 finden wir einen neuen Operator `JMP`. `JMP` ist kurz für Jump, also Sprung und nicht anderes tut dieser Befehl. Mit diesem Befehl (`JMP Label`) springen wir zum Label `EXIT`, zu sehen in Zeile 6. Damit wird Zeile 5 niemals ausgeführt, weil sie schlicht übersprungen wird. Mit Zeile 7 wird das Programm dann beendet. Dass ich den Code in Zeile 7 eingerückt habe hat eher übersichtstechnische Gründe und ist irrelevant.

`JMP` ist ein konditionsloser Sprungoperator. Es gibt aber auch `JMPs` die sich an Bedingungen knüpfen und somit komplexen Flow Control ermöglichen. Ein paar hier:

<code>JA</code>	- Springe wenn Über
<code>JAE</code>	- Springe wenn Über oder Gleich
<code>JB</code>	- Springe wenn Unter
<code>JBE</code>	- Springe wenn Unter oder Gleich
<code>JC</code>	- Springe wenn Carry Flag
<code>JCXZ</code>	- Springe wenn <code>CX = 0</code>
<code>JE</code>	- Springe wenn gleich
<code>JG</code>	- Springe wenn größer
<code>JGE</code>	- Springe wenn größer oder gleich
<code>JL</code>	- Springe wenn kleiner
<code>JLE</code>	- Springe wenn kleiner oder gleich
<code>JMP</code>	- Springe
<code>JNA</code>	- Springe wenn nicht Über
<code>JNAE</code>	- Springe wenn nicht Über oder Gleich
<code>JNB</code>	- Springe wenn nicht Unter
<code>JNE</code>	- Springe wenn nicht gleich
<code>JNG</code>	- Springe wenn nicht (größer)
<code>JNGE</code>	- Springe wenn nicht (größer oder gleich)
<code>JNL</code>	- Springe wenn nicht (kleiner)
<code>JNLE</code>	- Springe wenn nicht (kleiner oder gleich)
<code>JNO</code>	- Springe wenn kein Overflow
<code>JNZ</code>	- Springe wenn nicht 0
<code>JO</code>	- Springe wenn Overflow
<code>JZ</code>	- Springe wenn 0

Es gibt mehr, aber das sind die wichtigsten. Einige von ihnen sind selbsterklärend, aber einige bedarfen bestimmter Vertiefung. Was wird womit verglichen und woher kennt der Jump das Ergebnis? Nun, eigentlich kann alles mit allem verglichen werden. Das Ergebnis jedes Vergleich wird im Flagregister gespeichert. Der Jump liest dieses aus und reagiert entsprechend. Machen wir mal ein einfach `IF/THEN`:

```
(01) MOV DL,41h
(02) MOV DH,41h
(03) CMP DH,DL
(04) JE Equal
(05) JMP NotEqual

(06) NotEqual:
(07)     MOV AH,02h
(08)     MOV DL,4Eh
(09)     INT 21h
(10)     INT 20h
```

```

(11) Equal:
(12)     MOV AH,02h
(13)     MOV DL,59h
(14)     INT 21h
(15)     INT 20h

```

Wir haben sogar ein IF THEN, ELSE hinbekommen. Gehen wir mal der Reihe nach den Code durch. Wir befüllen irgendwelche Register, hier DL und DH mit irgendwelchen Werte. Hier sinds zweimal die gleichen Werte. Der Inhalt ist also gleich. In Zeile 3 wird dann der entscheidende Vergleich ausgeführt. Das Ergebnis dieser BOOLSchen Operation steht nun im Flagregister. Der Erste Befehl der auf dieses Ergebnis zugreift ist JE in Zeile 4. JE springt, wenn das Ergebnis des letzten Vergleichs "gleich" ist, wenn also die zwei Operanden übereingestimmt haben (was hier der Fall ist, 41h = 41h). Sollten die Operanden nicht gleich sein, unternimmt JE gar nichts. Wirklich nichts. Deshalb bleibt das Ergebnis in den Flags erhalten und der Code wird einfach weiter ausgeführt. Jetzt brauchen wir kein konditionales Jump mehr, denn wenn es nicht WAHR war, dann muss es FALSCH sein. Von daher springen wir kompromisslos einfach nach NotEqual. Egal ob Equal oder NotEqual betreten wurde, das Programm wird immer sauber mit INT 20h beendet. Sollte der Vergleich positiv ausfallen (also Equal betreten werden), dann wird "Y" für "Yes" ausgegeben, sonst "N" für "No".

Schleifen

Schleifen sind das Grundelement einer Programmiersprache. In vielen Sprachen sind sie durch WHILE, REPEAT, DO, etc. gekennzeichnet. In ASM fehlen uns solche Befehle. Wir werden uns einmal auf WHILE und FOR konzentrieren und manuell den Job eines Compilers nachempfinden, indem wir versuch lesbaren Code in ASM Code umzuwandeln.

Nehmen wir uns mal diesen Pseudocode:

```

FOR counter = 1 TO 5
    PRINT "A"
NEXT counter

```

Dieser Code wird 5x den Buchstaben A ausgeben. Genauer gesagt wird eine Variable mit dem Wert 1 initialisiert, danach wird der Buchstabe A ausgegeben, dann die Variable um 1 erhöht und zum Anfang der Schleife gesprungen, so denn die Variable kleiner als 5 ist. Und so sähe das im ASM Code aus (Kommentare kennzeichnet man bei den meisten Assemblern mit ;):

```

MOV CX,5                ;CX ist der LOOP-Counter

LOOP_LABEL:             ; Label für den Schleifenrücksprung
    MOV AH,02h
    MOV DL,41h
    INT 21h
    LOOP LOOP_LABEL ; Springe zurück, so lange CX > 0 ist (ändert autom.)

    INT 20h              ; Beende Programm, wenn LOOP scheitert

```

Der Code macht folgendes:

1. Setze Loopcounter CX auf 5
2. Gib ein A aus

3. Prüfe ob CX = 0, wenn nicht dekrementiere CX
4. Springe zurück zu LOOP_LABEL
5. Prüfe ob CX = 0, wenn nicht dekrementiere CX
6. ect...

Als nächstes nehmen wir uns mal die WHILE Schleife vor. Pseudocode:

```
WHILE counter < 5 DO
    counter = counter + 1
    PRINT "A"
LOOP
```

Das lässt sich fast identisch zur FOR-Schleife nach ASM übersetzen:

```
MOV CX, 5

loop_label:
    MOV AH, 02h
    MOV DL, 41h
    INT 21h
    DEC CX
    CMP CX, 0
    JNZ loop_label
    INT 20h
```

Also was ist der Unterschied? Warum CMP benutzen? Ganz einfach: Die FOR-Schleife ist gut, wenn man eine feste Anzahl an Schleifendurchgängen haben möchte. Bei der WHILE-Schleife die oft ein ganzes Programm am Leben hält kann sich die Bedingung "live" ändern, z.B. durch User-Input.

Manchmal allerdings ändern Interrupts das CX Register (phöse Teile). Um zu vermeiden dass die Schleife unvorhersehbar oft ausgeführt wird machen wir uns wieder den Stack zu Nutzen:

```
MOV CX, 5

Loop_Label:
    PUSH CX                ; CX auf den Stack schieben
    MOV AH, 02h
    MOV DL, 41h
    INT 21h
    POP CX                 ; CX wiederherstellen
    LOOP Loop_Label
    INT 20h
```

Variablen

Ja, ich weiß es gibt keine echten Variablen in ASM, aber es ist nah dran. Variablen bezeichnen benannte Adressen zu Speicherzellen. Um also auf diesen oder jenen Speicherbereich zuzugreifen muss man nicht mehr die ganze Adresse angeben, sondern nur den lesbareren Namen. Das erhöht die Übersichtlichkeit immens.

Wir haben bereits die drei wichtigsten Variablentypen kennengelernt: DWORD, WORD und BYTE. Variablen werden stets im Header (bei TASM oder FASM im Trail) des ASM Codes deklariert. Die

Deklaration erfolgt entweder durch DB (Declare Byte), DW (Declare Word) oder DD (Declare DWord). Variablen müssen mit einem Buchstaben beginnen, können aber wie Labels auch Unterstriche "_" und Zahlen enthalten:

```
aNumber: db 1
```

Erstellt eine 1 Byte breite Variable, die den Wert 1 hat.

```
aString: db "Hallo, Welt!", 0
```

Erstellt eine 1 Byte breite Variable, die den nullterminierten String "Hallo, Welt!" enthält. Eigentlich enthält sie die ASCII Codes jeder der Buchstaben. (Und eigentlich ist es ein Array)

Man kann auch Konstanten erstellen. Konstanten sind eigentlich das gleiche wie Variablen, bloß ändern sie ihren Wert nicht und ihr Wert kann auch nicht geändert werden. In ASM wird eine Konstante durch EQU erstellt:

```
CONSTANT_1 EQU dword 12345
```

Erstellt die Konstante COSTANT_1 mit dem Datentyp DWORD und dem Inhalt 12345.

Konstanten und Variablen werden wie Register und Werte mit den ASM Befehlen benutzt. Um etwas in eine Variable zu schreiben muss ein Pointer auf die Variable erstellt werden. In LASM geschieht das durch []:

```
aNumber: db 0  
MOV [aNumber], 42
```

Jetzt hat die Variable aNumber den Wert 42. Das wird später immens wichtig bei der 32Bit WinAPI Programmierung.

Machen wir uns ein kleines Hallo, Welt - Programm:

```
shit: db "Hallo, Welt!", 0      ; Das Komma verknüpft (Au3: &)  
MOV AH, 09h  
MOV DX, OFFSET shit  
INT 21h  
INT 20h
```

Das ist kein mir bekannter ASM Dialekt, aber so funktioniert es. Die Funktion 09h aus dem Interrupt 21h benutzt den Anfangspointer, der jetzt in DX steht. OFFSET ist hier der Beginn der Variable im RAM.

String Operationen

Es gibt ein paar String Operationen die noch erwähnenswert sind. Ich werde mal 4 davon erklären. Fangen wir an mit MOVSB. Dieser Befehl verschiebt ein Byte von einer Position auf eine andere. Die Quellposition ist ES:SI und die Zielposition ist DS:DI:

```
OLDSTRING: DB "ABCDEFGHI ", 0  
NEWSTRING: DB "123456789 ", 0  
MOV AH, 9  
MOV DX, OFFSET NEWSTRING
```

```

INT 21h
MOV DX,OFFSET OLDSTRING
INT 21h
MOV CX,9
LEA SI,OLDSTRING
LEA DI,NEWSTRING
REP MOVSB
MOV DX,OFFSET NEWSTRING
INT 21h
MOV DX,OFFSET OLDSTRING
INT 21h
INT 20h

```

Output sollte "ABCDEFGHI 123456789 ABCDEFGHI ABCDEFGHI" sein. Was passiert im Code:

1. Wir machen den üblichen Startkram
2. Wir nutzen die altbekannte Methode um NEWSTRING auszugeben
3. Wir geben auch OLDSTRING aus
4. Wir setzen CX auf 9. Es werden als 9 Schleifendurchgänge durchgeführt
5. LEA steht für Load Effective Address. Dieser Befehl lädt den Inhalt einer Variable in ein Register. Weil DI das Ziel und SI die Quelle enthält, setzen wir die Adresse von NEWSTRING und OLDSTRING dementsprechend
6. MOVSB bewegt ein Byte von SI nach DI. REP (=REPeat) wiederholt MOVSB so oft, wie in CX angegeben
7. Wieder wird alles ausgegeben

Der nächste Stringbefehl ist nicht nur einfach zu benutzen, sondern auch sehr nützlich! Er sucht in einem String nach einem gegebenen Zeichen und setzt das EQUAL Flag Bit, wenn die Suche erfolgreich war. Eine eigentlich sehr komplexe Operation, wenn wir bedenken, dass es um ASM geht. Der Operator ist SCASB, die Position des String ist in DI und das Zeichen wird in AL gespeichert:

```

MOV CX,17h
LEA DI, STRING
MOV AL, SEARCH
REPNE SCASB
JE FOUND
JNE NOTFOUND

```

```

NOTFOUND:
    MOV AH,09h
    MOV DX,OFFSET NOTFOUND_S
    INT 21h
    INT 20h

```

```

FOUND:
    MOV AH,09h
    MOV DX,OFFSET FOUND_S
    INT 21h
    INT 20h

```

```

SEARCH:      DB  "!"
STRING:      DB  "AutoIt is good for you!"
FOUND_S:     DB  "Found", 0
NOTFOUND_S:  DB  "Not Found", 0

```

Das sollte eigentlich einfach zu verstehen sein:

1. Übliches Setup
2. CX auf 17h (23), weil unser String 17h Bytes lang ist
3. Position des Strings in DI
4. ASCII Wert des Such-Zeichens nach AL
5. 23x SCASB
6. JMP wertet die Suche aus

Der nächste Befehl, den ich dir näher bringen möchte ist CMPS. Wie der Name vermuten lässt, vergleicht dieser Operator zwei Strings miteinander:

```
MOV CX,17h
LEA SI,STRING1
LEA DI,STRING
REP CMPSB
JE EQUAL
JNE NOTEQUAL

NOTEQUAL:
    MOV AH,09h
    MOV DX,OFFSET NOT_EQUAL
    INT 21h
    INT 20h

EQUAL:
    MOV AH,09h
    MOV DX,OFFSET EQUAL1
    INT 21h
    INT 20h

STRING1:    DB 'AutoIt is good for you!'
STRING:     DB 'AutoIt is good for you!'
EQUAL1:     DB 'They are equal', 0
NOT_EQUAL:  DB 'They are not equal', 0
```

Und zu guter Letzt haben wir noch STOSB und LODSB. STOSB speichert ein Byte von AL an der Stelle ES:DI. STOSB speichert ein Byte aus der Stelle ES:DI in AL:

```
MOV AH,9
MOV DI,OFFSET STRING
MOV SI,DI
LODSB
INC AL
STOSB
MOV DX,DI
DEC DX
INT 21h
INT 20h

STRING DB "oesto! ", 0
```

Dieser Code wird "pesto!" ausgegeben. Nimm zur Kenntnis, dass die meisten Assembler, vor allem 16Bit Assembler keine nullterminierten Strings, sondern Strings verwenden, die auf \$ enden, also: "oesto! \$".

Also was macht der Code?

1. 9 nach AH um auf die Textausgabefunktion in Interrupt 21 zuzugreifen
2. Bewege die Position von STRING nach DI für die LODSB Funktion
3. Das Gleiche mit SI
4. Lade ES:DI in AL
5. Erhöhe AL und ändere somit o zu p
6. Bewege AL zurück zu ES:DI
7. Schreibe DI nach DX für INT21s Textausgabe
8. STOSB wird DI erhöhen, wenn die Ausführung erfolgreich war, also dekrementieren wir es wieder
9. Ausgabe & Ende

Diese Stringoperationen gibt es eigentlich für alle Datentypen. Wenn MOVSB ein **Byte** bewegt, bewegt MOVSW ein **Word**.

Subs

Subs sollten den BASIC Programmieren bekannt sein. Wenn nicht, auch nicht schlimm. Sub ist die Abkürzung für Subprozedur, also eine Prozedur (ebenfalls durch ein Label gekennzeichnet), die von einer Prozedur unter einem anderen Label aufgerufen wird.

Subs lassen sich mit nested functions vergleichen. Pseudocode:

```
Funktion1(Funktion2())
```

Subs können von jeder Stelle im ASM Code aufgerufen werden. Nach ihrer Beendigung wird der Hauptcode einfach weiter ausgeführt. Aufgerufen werden Subs durch den Operator CALL:

```
CALL DOENER
CALL DOENER
CALL DOENER
CALL DOENER
CALL DOENER
CALL DOENER
CALL DOENER
CALL DOENER
INT 20h
```

```
DOENER PROC
    MOV AH,09
    LEA DX,MSG
    INT 21h
    RET
DOENER ENDP
```

```
MSG DB 'Doener is good for you! $'
```

RET kehrt wieder zum Hauptcode zurück. Ohne RET würde der Sub nicht verlassen werden und das Programm erreicht INT 20h nicht.

Benutzer Input

Output hatten wir jetzt zu Genüge, jetzt widmen wir uns dem Input! Wir werden uns in diesem Kapitel mit simplen Eingabe-Interrupts aus dem BIOS beschäftigen. Der wichtigste Tastatur-Interrupt Handler ist 16h. Wir benutzen die Funktion 0h:

```
MOV AH,0
INT 16h
INT 20h
```

Dieses Programm wartet darauf, dass der Nutzer irgendeine Taste drückt und beendet sich dann selbst. Aber wir sehen nicht, was wir nun gedrückt haben. Um das zu ändern wollen wir das Programm noch dahingehend erweitern, dass es die gedrückte Taste ausgibt. Zum Ausgeben haben wir 21h mit der Funktion 2h benutzt, wobei das Zeichen in DL stand. Wir müssen also das empfangene Zeichen nach DL schreiben um es auszugeben. Das gelesene Zeichen von 16h/0h wird in AL geschrieben:

```
MOV AH,0h
INT 16h
MOV AH,2h
MOV DL,AL
INT 21h
INT 20h
```

Wenn du alles bis jetzt gelesen und verstanden hast, dann ist der Code oben natürlich kein Problem. Aber zeig das mal einem Laien, der wird anhand des Codes rein gar nichts verstehen. Du kannst dich also selbst loben (;-)).

Schon ganz gut, aber 16h hat noch viel mehr zu bieten. Es kann nämlich auch den Zustand einzelner Tasten wie STRG, ALT, CAPS LOCK usw. prüfen! Also schreiben wir mal so ein richtiges Hardcore-Programm:

```
MOV AH,0h
INT 16h
MOV KEY,AL
CMP KEY,90h
JE ITS_A_Z
JNE NOT_A_Z

ITS_A_Z:
    MOV AH,9h
    MOV DX,OFFSET NOTA
    INT 21h

    INT 20h

NOT_A_Z:
    MOV AH,2h
    MOV DL,KEY
    INT 21h

    INT 20h

KEY DB ?
NOTA DB "Du hast die Z-Taste gefunden!",10,13,"H3xXx0r! $"
```

Der Code in Stückchen:

1. KEY DB ? erstellt eine Variable ohne Anfangswert
2. AX auf Funktion 0h und calle INT 16h und warte auf Eingabe
3. ASCII Wert der Eingabe aus AL in die KEY Variable schreiben
4. KEY mit 90h (großes Z) vergleichen
5. Wenn es Z war, dann zu ITS_A_Z springen
6. Wenn nicht, zu NOT_A_Z springen, gedrückte Taste ausgeben und Ende
7. Die Zahlen zwischen den beiden Strings in NOTA sind CR und LF, zusammen ergeben sie den Zeilenumbruch CRLF

ASM Grundlagen III

Grafik

Das worauf du die ganze Zeit gewartet hast. Aber zunächst ein wenig Enttäuschung: Die meisten Grafikinterrupts oder Befehle funktionieren nur noch unter XP. Unter Win7 gibt es entweder Fehler oder Hänger. Da bleibt nur noch der Umweg über DOSBox oder eine kleine VBox für die ASM Spielereien (eine VBox passt sich auch von der Größe her automatisch an = sehr praktisch).

Grafik durch Interrupts

Das ist die einfachste Möglichkeit direkt in den Video-Speicher der Grafikeinheit zu schreiben. Wir werden dafür das BIOS Graphics Interrupt 10h benutzen. Um überhaupt etwas anzeigen zu können, müssen wir einen Grafikmodus betreten. Wir nutzen einfach Mal Modus 13 (bekannt aus BASIC "SCREEN 13"). Dieser Modus hat eine Auflösung von 320x200x256. Also 320 Pixel breit, 200 Pixel hoch und uns stehen 256 Farbtöne zur Verfügung. Um den Grafikmodus zu betreten müssen wir ihn in AL schreiben:

```
MOV AH,00h
MOV AL,13h
INT 10h
INT 20h
```

Ist erstmal nicht viel anders, die Auflösung wird bloß geringer. Färbern wir mal einen Pixel:

```
MOV AH,00h
MOV AL,13h      ; Modus 13
INT 10h         ; Modus ändern
MOV AH,0Ch      ; DrawPixel Funktion
MOV AL,10       ; Farbe
MOV CX,100      ; X Koordinate
MOV DX,100      ; Y Koordinate
MOV BX,1h       ; VGAPage 1 (nicht weiter kümmern)
INT 10h         ; Pixel zeichnen
INT 20h
```

Einmal im Grafikmodus kann man auch wieder mit AL = 03h in den Textmodus zurück. Ein Tweak gibt es selbst bei diesem simplen Programm noch: Man kann mit nur einem MOV AL und AH setzen, indem man einfach gleich AX bearbeitet, weniger übersichtlich, aber schneller und das Endprodukt ist ganze 3 Byte kleiner! Das folgende Programm zeichnet einen grünen Pixel bei 100, 100 auf VGAPage 1, wechselt zurück zum Textmodus und entfernt dabei den Pixel automatisch:

```

MOV AX,0013h
INT 10h
MOV AX,0C04h
MOV CX,100
MOV DX,100q
MOV BX,1h
INT 10h
MOV AX,0003h
INT 10h
INT 20h

```

Man würde nichts sehen, wenn dieses Programm lief. Einerseits haben wir schon ein wenig Optimierung betrieben, trotzdem ist die Routine verhältnismäßig langsam. Unter modernen CPUs sieht man nichts, weil sie trotzdem in Bruchteilen von Millisekunden abläuft. Wenn man aber Fullscreen damit zeichnen möchte kommt auch eine moderne CPU ins Schwitzen und man hat kaum noch CPU Zeit für Berechnungen.

Direkt in den VRAM schreiben

So, jetzt wirds das erste Mal etwas komplizierter (höhö). Denn direktes Schreiben in den VRAM, also das Bearbeiten des Grafikspeichers erfordert ein wenig Rechnen. Ich werde dir nun einige Befehle beibringen, die besagtes Pixelfärben noch leichter und vor allem viel schneller machen soll. Beim Pixel plotten mittels Interrupt gibts du nur die XY Koordinaten und die Farbe an. Wenn du allerdings direkt in den VRAM schreiben willst, dann musst du dir die physikalische Adresse des Pixels (das VRAM Offset) errechnen. Der VRAM ist einfach nur eine Art fortlaufendes Band, auf dem Grafikinformatoren stehen. Bei Mode 13 ist dieses Band somit so lang wie die Gesamtzahl an Pixeln und beginnt bei 0. Also reicht der VRAM von 0 bis $320 \times 200 = 64000$. Wir kriegen wir dann trotzdem XY Koordinaten rein? Ganz einfach: Der Bildschirm fängt stets oben links an und setzt sich dann mit dem VRAM nach rechts fort. Die erste Zeile Pixel auf dem Bildschirm ist also 0...320. Wollen wir nun einen Pixel bei P(100, 0) ändern, dann müssen wir im VRAM die Stelle 100 bearbeiten. Es was ist mit Y Koordinaten (die den Abstand zum oberen Bildschirmrand angeben) über 0? Dazu rechnen wir einfach so lange volle Pixelzeilen, bis wir in der gewünschten sind:

$$\text{Offset} = Y \times 320 + X$$

Sagen wir mal wir setzen den Pixel bei P(100, 100). Dann haben wir zunächst 100 volle Pixelzeilen ($Y = 100$. 100×320). Nun sind wir im VRAM bei 32000 und koordinatentechnisch bei P(0, 100). Jetzt müssen wir einfach noch den fehlenden X Abstand addieren. Also ist das Offset bei P(100, 100): 32100. A000 ist das Segment, an welchem der VRAM beginnt:

$$A000: Y \times 320 + X$$

Da PCs Multiplikationen hassen, formen wir um:
 $A000: Y \times 256 + Y \times 64 + X$

Das geht aber noch besser:

$$A000: Y \times 2^8 + Y \times 2^6 + X$$

Jetzt arbeiten wir mit Basis-2. Aber wie errechnen wir Potenzen in Assembler? Wir benutzen eine Technik namens *Shifting*. Shifting ist ein einfaches Konzept. Es gibt zwei Arten des Shiftings, nämlich Shift Left und Shift Right. Wenn man eine Zahl shiftet, dann addiert die CPU einfach eine 0 an das betreffende Ende. Beispiel:

256 = 100000000b

Shift Left:
1000000000b = 512

Shift Right:
0100000000b = 256

Shifts drücken immer dann Potenzen aus, wenn n aus 2^n die Zahl ist um welche geshiftet wird. Wir können dies nun einfach in unsere vorherige Formel einfließen lassen:

A000:Y SHL 8 + Y SHL 6 + X

Das Ergebnis bleibt das gleiche, bloß ist es viel schneller. Setzen wir das mal in ASM um. Wir bauen uns dazu ein Programm, das selbst alles vorbereitet, damit wir später ganz einfache Befehle nutzen können um den Pixel zu färben:

```
SET_VSEGMENT:                ; Video Setup
    MOV AX,0A000h             ; Pointer in ES auf das VGA Segment
    MOV ES,AX

VALUES:
    MOV AX,100                ; X
    MOV BX,100                ; Y

GET_OFFSET:                   ; Berechne Offset an XY
    MOV DI,AX                 ; X nach DI
    MOV DX,BX                 ; Y nach DX
    SHL BX,8                  ; Y x 2^8
    SHL DX,6                  ; Y x 2^6
    ADD DX,BX                 ; addieren
    ADD DI,BX                 ; und noch X addieren
```

Nun ist alles vorbereitet und wir können einfach den STOSB Befehl nutzen um den Pixel zu schreiben. Die Farbe wird dabei aus AL gelesen:

```
MOV AL, 4
STOSB
```

Wie ich sagte, etwas kompliziert, aber wir können uns immer Routinen bauen um alles zu vereinfachen. Diese Vorgehensweise hilft beim Assembler Programmieren. Aber zurück zum Code. Hier noch einmal ausführlicher uns sauberer Code, der alles ordentlich initialisiert und am Ende wieder aufräumt und zum Textmodus zurückkehrt um dann zu enden. Versuche den Code langsam durchzugehen und dir selbst Kommentare anzufertigen.

```
MOV AH,00h                   ; Video-Modus
MOV AL,13h                   ; Modus 13
INT 10h

SET_VSEGMENT:
    MOV AX,0A000h
    MOV ES,AX

VALUES:
```

```

MOV AX,100
MOV BX,100

GET_OFFSET:
MOV DI,AX
MOV DX,BX
SHL BX,8
SHL DX,6
ADD DX,BX
ADD DI,BX
MOV AL,4
STOSB
XOR AX,AX
INT 16h
MOV AX,0003h
INT 10h

INT 20h

```

Dateiverwaltung

Ach ja, das langweilige Kapitel gibts ja auch noch.

File Handles

Ein File Handle ist der Bezeichner, der einer Datei zugeordnet wird, sobald man sie öffnet. Eine Datei zu öffnen, heißt nicht dass sie angezeigt oder gelesen wird, sie bekommt einfach nur ein Handle:

```

MOV AX,3D00h
LEA DX,FILENAME
INT 21h
JC ERROR

INT 20h

ERROR:
MOV AH,09h
LEA DX,ERRORMSG
INT 21h

INT 20h

FILENAME DB 'TEST.TXT',0
ERRORMSG DB 'Unable to open [test.txt]$\

```

Wenn wir eine Datei namens text.txt im Verzeichnis hätten, würde sich das Programm einfach beenden. Ohne die Datei allerdings Gibt es die Fehlerbenachrichtigung aus und beendet sich dann. Also was passiert hier?

Schritt 1:

Wir setzen AX auf 3D00h, kurz für AH = 3Dh und AL = 00h. 3Dh ist die Funktion im Interrupt 21h, welche Dateien öffnet. Die Interruptfunktion liest AL aus und bestimmt damit, wie die Datei geöffnet werden soll:

Bit 0..2: Zugriffart

- 0 - Lesen
- 1 - Schreiben
- 2 - Beides

Bit 3: Reserviert (sprich 0)

Bit 4..6: Sharing Modus

- 0 - Kompatibilitätsmodus
- 1 - Exklusiv
- 2 - Schreibsperre
- 3 - Lesesperre
- 4 - Keine Einschränkungen

Bit 7: Interheritanz Flag

- 0 - Datei wird von einem Child-Prozess interheriert
- 1 - Exklusiv für den aktuellen Prozess

Nicht den Kopfzerbrechen über diese vielen Möglichkeiten, wir werden nur das Zugriffsbit benutzen.

Schritt 2:

Wir laden die Adresse des Dateinamens nach DX. Der Dateiname muss ein ASCII-Z String sein, d.h. er muss in jedem Fall nullterminiert werden. Sonst werden nach dem Namen weitere Befehle ausgeführt, und zwar immer die, die gerade in diesem Speichersegment sind und das kann böse Folgen haben.

Schritt 3:

Interrupt 21h aufrufen.

Schritt 4:

Das Carry Flag wird automatisch geändert, sobald ein Fehler beim Öffnen einer Datei auftritt und der Errorcode wird in AX gespeichert.

Schritt 5:

Wenn kein Error auftritt, wird das File Handle nach AX geladen und das Programm beendet.

Dateien lesen

Wenn wir nun das Handle der Datei haben können wir sie auch benutzen. Zum Beispiel lesen. Wenn du die Lesen-Funktion als Interrupt 21h benutzen willst, muss du die Register wie folgt anlegen:

AH = 3Fh

BX = File Handle

CX = Zahl der Bytes die gelesen werden sollen

DX = Pointer auf einen Buffer in welchem die gelesenen Daten gespeichert werden sollen

Dann kannst du den Inhalt zum Beispiel einfach ausgeben, mit der Print Funktion aus Interrupt 21h. Allerdings musst du die Länge der zu lesenden Daten angeben und das ist nicht perfekt, da wir die meiste

Zeit nicht wissen wie lang eine Datei ist. Wir können dafür aber einen kleinen Trick benutzen. Wenn ein Error auftritt wird der Errorcode in AX abgelegt. Der Errorcode 0 sagt dabei aus, dass die Lesenfunktion ein EOF, also das Ende der Datei festgestellt hat. Also bauen wir uns selbst eine Funktion die so lange jeweils ein Byte einliest, bis ein EOF auftritt. Am Ende aller Dateioperationen sollten wir noch die Datei freigeben, mit INT 21h's CloseFile Funktion. Dazu wieder der Code:

```
START:
    MOV AX, 3D00h
    LEA DX, FILENAME
    INT 21h
    JC ERROR
    MOV BX, AX

READFILE:
    MOV AH, 3Fh
    MOV CX, 0001h
    LEA DX, CHARACTER
    INT 21h
    CMP AX, 000h
    JE ENDPROGRAM
    MOV AH, 02h
    MOV DL, CHARACTER
    INT 21h
    JMP READFILE

ENDPROGRAM:
    MOV AH, 3Eh
    INT 21h
    INT 20h

ERROR:
    MOV AH, 09h
    LEA DX, ERRORMSG
    INT 21h
    INT 20h

FILENAME DB 'TEST.TXT', 0
ERRORMSG DB 'Failed to open file!$'
CHARACTER DB ? ; nicht initialisierte Variable (für andere Assembler 0)
```

Mit dem bisherigen Vorwissen solltest du den Code gut verstehen können, trotzdem schiebe ich mal die Analyse nach:

Schritt 1:

Wir öffnen die Datei und erhalten so das File Handle

Schritt 2:

Wir kopieren das File Handle nach BX. INT 21h's Funktion des Dateilesens fordert das Handle in BX.

Schritt 3:

Wir schieben den Bezeichner der Funktion "Lesen", nämlich 3Fh nach AH.

Schritt 4:

CX enthält die Anzahl der zu lesenden Bytes, wir wollen nur eines.

Schritt 5:

Das gelesene Byte wird in den Buffer auf welchem DX zeigt (CHARACTER) geschrieben.

Schritt 6:

Wir vergleichen AX mit 0, was positiv ausfällt wenn ein EOF erkannt wurde. Wenn dem so ist, wird das Programm beendet.

Schritt 7:

Wenn es kein EOF war, dann geben wir den Buchstaben / das Byte nochmal als Echo aus.

Schritt 8:

Zurück zu READFILE, damit wir noch ein Byte lesen können. Hier sollte dann ein EOF stehen. Darauf wird das Programm beendet, vorher noch die Datei geschlossen.

Schlusswort zu 16Bit Assembler

So, näher gehe ich nicht auf die 16Bit Basics ein. Zum Ersten weil sie dir jetzt garantiert zum Hals raushängen und zum Anderen, weil in Win32 Zeiten diese Art von Assembler, vor Allem das Jonglieren mit Interrupts, eine aussterbende Kunst ist. Klar gibt es noch einige Freaks, die COM Dateien produzieren (extrem kleine Grafikdemos, dazu später noch einmal), aber sonst ist das völlig überflüssig.

Die 16Bit Befehle an sich solltest du aber keinesfalls vergessen, die sind auch weiterhin Grundelemente in Assembler, auch in 32Bit.

32Bit Assembler

Also bisher haben wir ja die Funktionen die wir brauchten aus den Interrupts gewonnen. Unter Windows ist alles ganz anders. Microsoft stellt uns also das Betriebssystem bereit und einen Zugang dazu, den wir API nennen (Application Programming Interface). Diese WinAPI besteht aus einem Shitload an .DLL, .OCX, .EXE und .SYS Dateien. Am wichtigsten davon werden für uns die .DLL Dateien, in denen alle Funktionen, also alles was wir mit Windows anstellen können enthalten sind. DLL steht für Dynamic Link Library. Es sind einfach gesagt fertige Programme ohne Benutzeroberfläche, die ihre eigenen Funktionen jedem anderen Programm, das sie nutzt, offenbart. Der Rest vom Assembler bleibt gleich, bis auf die Tatsache, das unter einem 32Bit OS natürlich mit 32Bit Registern gerechnet wird.

LASM

Erster Schritt um mit Assembler irgendetwas zu tun, ist sich einen zu besorgen. Also warum LASM? Aus Gründen. Man muss sich weder irgendein Softwarepaket installieren, noch wird man von Drittgenerationssyntax verwirrt, wie es z.B. MASM hat. LASM ist bodenständig und tatsächlich ein echter

Assembler ohne Schnickschnack. Ich habe dazu noch eine IDE gebaut, in der wir unsere Projekte verwalten können.

Los gehts

Unser erstes Programm wird gleich ein "Hello, World!" Programm. Klingt angesichts der bisherigen Codes nach einem wahren Code-Monster, aber die WinAPI macht daraus wenige Zeilen. Wenn du jetzt die LASM IDE startest, und das Projekt "MessageBox" öffnest, siehst du, was ich meine:

```
01 extern "user32", MessageBoxW

02 DEFAULT          equ dword 0
03 MB_RETRYCANCEL  equ dword 5

04 Message: dw "16 Bit Unicode Message", 0
05 Title:   dw "LASM Message", 0

06 main:
07     push MB_RETRYCANCEL
08     push dword Title
09     push dword Message
10     push DEFAULT
11     call MessageBoxW
12 ret
```

Setzieren wir mal den Code, wie immer. Also ein LASM Programm muss immer die Sprungmarke `main` besitzen. Dies ist der sogenannte `main`-Entrypoint. Das Programm sollte immer mit einem einfachen `RET` enden, damit es sich beendet ohne dass wir extra Funktionen dazu aufrufen müssen.

Zeile 1

In Zeile 1 sagen wir LASM, es soll die Funktion `MessageBoxW` aus der WinAPI Datei "user32.dll" in das Programm einbinden. Hilfe zu dieser Funktion finden wir bei MSDN, dem Online Verzeichnis von Microsoft für Entwickler. Wie googeln: "(function) MessageBoxW". Der erste Treffer ist dann immer MSDN. Dort steht, wie die Funktion aufgerufen werden sollte. Wir finden den Code:

```
int WINAPI MessageBox(
    _In_opt_  HWND hWnd,
    _In_opt_  LPCTSTR lpText,
    _In_opt_  LPCTSTR lpCaption,
    _In_      UINT uType
);
```

Sieht auf den ersten Blick etwas kryptisch aus, ist auch völlig unnötig kompliziert formuliert. In der ersten Zeile sehen wir nochmal den Namen der Funktion "MessageBox". Aber hier fehlt doch das "W"? Nun, von manchen Funktionen in der WinAPI, vor allem jene, welche mit Strings arbeiten, gibt es meist zwei Varianten: FunktionA und FunktionW. A steht für ASCII/ANSI und W steht für Unicode (Western). Um das zu sehen, was auch im ASM Code steht, müssen wir Unicode benutzen, da sonst alles als ASCII Code interpretiert wird. Dann steht in der Vorschrift, es gäbe 4 Paramter: `hWnd`, `lpText`, `lpCaption` und `uType`. Das Geschreibsel vor den Paramtern sollen die Datentypen sein. Jetzt kommt das dunkle Geheimnis: Das

ist totaler Unsinn. Diese Datentypen sind so unnütz wie sonst nichts, weil sie schlichtweg nicht existieren. In der WinAPI lassen sich alle Funktionen eigentlich mit DWORD, also einem Double ansprechen. HWND ist DWORD, LPCTSTR ist DOWRD und UINT ist aber wirklich ein ... DWORD.

Zeile 2 und 3

Für uType gibt es verschiedene Werte, die das Aussehen der MsgBox beeinflussen. 0 (MB_OK) wäre wie der name schon sagt ein einfacher OK Button. Wir wollen es aber fancy und nehmen Retry & Cancel (MB_RETRYCANCEL). Wir definieren das als Konstante (dw = dword) in Zeile 3. Für hWnd könnten wir das Handle eines Fensters angeben, dem wir die MsgBox zuordnen wollen, haben wir aber nicht, also setzen wir hier 0 ein, oder hier DEFAULT, auch eine Konstante die wir im Voraus deklariert haben. "equ" heißt eigentlich nur "=".

Zeile 4 und 5

Wir bestimmen zwei Strings, die wir mit einem Null-byte terminieren. "," ist der Operator der in LASM Datenfelder miteinander verknüpft.

Zeile 7 bis 11

Die Parameter werden nun in verdrehter Reihenfolge auf den Stack gepusht (wenn du nicht weißt warum, ließ nochmal das Stack-Kapitel). Konstanten deren Datentyp wir explizit angeben müssen nicht mit der zusätzlichen Kennzeichnung des Datentyps gepusht werden. Die Buffer, deren Inhalt sich ändern kann, die Variablen also, müssen mit dem Datentyp gepusht werden. "call" ruft dann letztendlich die Funktion aus der WinAPI auf. Die Funktion hat eine sogenannte Std-Calling-Convention, d.h. sie räumt den Stack automatisch auf. Der Return der Funktion, insofern wir wie hier keinen Paramter ByRef übergeben haben, steht immer in EAX.

Zeile 12

ret verläßt den main-block. Kein weiterer Code ist vorhanden, das Programm terminiert automatisch.

Weiter geht's

Nun wollen wir mal ein etwas komplexeres Programm schreiben, mit GUI, Controls, Message Loop etc.

Dazu passend existiert das Beispiel namens "GUI.lasmprj". Lade jenes in die IDE. Nun erscheint folgender Code (ich kommentiere dazwischen, da es mit 200 Zeilen doch etwas umfangreicher wird):

```
(1) extern "kernel32", GetModuleHandleA
(2) extern "user32",   LoadCursorA
(3) extern "user32",   RegisterClassA
(4) extern "user32",   UnregisterClassA
(5) extern "user32",   CreateWindowExA
(6) extern "user32",   DestroyWindow
(7) extern "user32",   ShowWindow
(8) extern "user32",   GetMessageA
(9) extern "user32",   TranslateMessage
(10) extern "user32",  DispatchMessageA
(11) extern "user32",  DefWindowProcA
(12) extern "user32",  PostQuitMessage
```

```

(13) extern "user32",    MessageBeep    ; Sound? Sound!
(14) extern "user32",    SetWindowTextA
(15)

```

Bis hier hin werden ausschließlich Funktionen aus der WinAPI (kernel32.dll, user32.dll) importiert.

```

(16) AppName:           db "ASMGUI", 0
(17) WindowTitle:       db "LASM Window Demo", 0
(18) hInstance:         dd 0
(19) hCursor:           dd 0
(20) hWnd:              dd 0
(21) wParam:            dd 0
(22) hEdit1:            dd 0
(23) szEditClass:       db "EDIT", 0
(24) szButtonClass:     db "BUTTON", 0
(25) txtButton:         db "Button Control", 0
(26) txtSample:         db "A Editcontrol. AutoIt-like!", 0
(27) hNull:             dd 0
(28) hButton1:          dd 0
(29) hDC:               dd 0
(30)

```

Bis hier werden die Variablen deklariert, welche wir später im Code nutzen. `AppName` wird in der Fensterklasse benutzt, `WindowTitle` sollte klar sein, `hInstance` wird später die Instanznummer tragen, `hCursor` wird die Cursor ID enthalten, da wir einen anderen Cursor als den Standard-Mauspfeil benutzen, `hWnd` bekommt das Handle des Fensters, `wParam` wird die Message enthalten, die von der GUI ausgelöst wird, `hEdit1` erhält das Handle zum Edit-Feld, welches später erstellt wird, die `sz[...]Class` Variablen sind die Klassennamen der Fensterklassen der Controls die wir benutzen, die `txt[...]` enthalten den Text, der später den erstellten Controls zugewiesen wird, `hButton` wird mit dem Handle des Buttons gefüttert und `hDC` wird den Device-Context enthalten, wenn wir noch etwas zeichnen wollen. In diesem Programm benutzen wir allerdings keine Zeichenfunktionen.

```

(31) ; ->
(32) COLOR_APPWORKSPACE equ 12
(33) sizeof_WNDCLASS    equ 40
(34) sizeof_MSG          equ 28
(35) WM_DESTROY         equ &H2
(36) WM_CREATE          equ &H1
(37)

```

Bis hier her werden Standard Windows-Konstanten deklariert. (32) wird die Hintergrundfarbe der GUI, (33) und (34) sind Größenangaben für den Speicherplatz der für die Fensterklasse und für die Message benötigt wird. (35) und (36) sind die einzigen Messages auf die wir während der Ausführung prüfen.

```

(38) ; Fadenkreuz statt Pfeil
(39) IDC_CROSS          equ &H7f03
(40)

```

Hier wird die Konstante gesetzt, welche angibt, welches Aussehen später der Cursor auf der GUI Fläche haben soll.

```

(41) ; Sound ID fuer MessageBeep
(42) MB_ICONSTOP        equ &H10
(43)

```

MessageBeep spielt Standard-System-Töne ab. Jeder Ton hat seine eigene ID, welche auch mit einem Namen versehen ist. Wir wollen den Error-Ton hören, also setzen wir auch die entsprechende Konstante.

```
(44) DEFAULT                equ 0
(45)
(46)
```

DEFAULT wird für jeden optionalen Wert übergeben, den wir nicht brauchen.

```
(47) main:
```

Haupt-Einsprungpunkt, da alle Deklarationen abgeschlossen sind.

```
(48)    push ebp
(49)    mov  ebp, esp
```

EBP speichern und ESP in EBP sichern.

```
(50)
(51)    push dword 0
(52)    call GetModuleHandleA
(53)    mov [hInstance], eax
```

Ah! Eckige Klammern! So wird in LASM ein Pointer adressiert. Andere Assembler nutzen dafür ein @ o.ä. Hier wird eine neue Instance erstellt, aus dem Handle des Programms.

```
(54)
(55)    push dword IDC_CROSS
(56)    push dword 0
(57)    call LoadCursorA
(58)    mov [hCursor], eax
(59)
```

Hier laden wir den Cursor.

```
(60)
(61)    ;;;;;; WND CLASS ;;;;;;
(62)
(63)    sub esp, sizeof_WNDCLASS    ; WNDCLASS 40 Bytes
```

Wir reservieren 40 Bytes für unsere eigene Fensterklasse. Diese Klasse nutzen wir später beim Erstellen der GUI. Bitte sieh dir selbst MSDN zu den Funktionen an um zu verstehen warum und wie welche Werte gesetzt werden.

```
(64)
(65)    mov dword [esp+0], 3        ; style = CS_HREDRAW or CS_VREDRAW
```

Byte 0 bis 4 setzen wir den Style, der eine bitwise Kombination aus CS_HREDRAW und CS_VREDRAW ist und dafür sorgt, dass das Fenster stehts in beide Richtungen neu gezeichnet wird.

```
(66)    mov dword [esp+4], WndProc    ; lpfnWndProc = WndProc
```

Hier wird der Pointer zur Funktion WndProc übergeben, die später von der GUI ausgelöste Messages empfängt.

```
(67)    mov dword [esp+8], 0        ; cbClsExtra = 0
(68)    mov dword [esp+12], 0       ; cbWndExtra = 0
```

(69)

Diese Paramter werden nicht benötigt.

```
(70)    mov eax, [hInstance]
(71)    mov dword [esp+16], eax          ; hInstance
(72)
```

Kopieren von dem Handle aus hInstance in die Struct (Klasse).

```
(73)    mov dword [esp+20], 0          ; hIcon = 0
(74)
```

Standard Windows-Icon für das Fenster.

```
(75)    mov eax, [hCursor]
(76)    mov dword [esp+24], eax        ; hCursor
(77)
```

Unseren geladenen Cursor.

```
(78)    mov dword [esp+28], COLOR_APPWORKSPACE ; background color
```

Hintergrundfarbe setzen.

```
(79)    mov dword [esp+32], 0          ; lpzszMenuName = 0
```

Wir haben kein Menu.

```
(80)    mov dword [esp+36], AppName    ; lpzszClassName = AppName
```

Wir benennen unsere Klasse nach dem Wert in AppName.

```
(81)
(82)    push dword esp                ; register the window class
```

Lade die Klasse in esp als Argument.

```
(83)    call RegisterClassA
(84)
```

Die Klasse Programmweit registrieren.

```
(85)    ;;;;;; CREATEWINDOW ;;;;;;
(86)
(87)    push dword 0                  ; lParam
(88)    push dword [hInstance]       ; hInstance
(89)    push dword 0                 ; hMenu
(90)    push dword 0                 ; hWndParent
(91)    push dword 250               ; Height
(92)    push dword 400               ; Width
(93)    push dword 100               ; y
(94)    push dword 100               ; x
(95)    push dword &HCF0000         ; WS_OVERLAPPEDWND or
WS_BORDER or WS_CAPTION
(96)    push dword WindowTitle       ; lpWindowName
(97)    push dword AppName           ; ClassName
(98)    push dword &H40000          ; dwExStyle
```

Alle Argumente für CreateWindowEx laden (siehe MSDN).

```
(99)    call CreateWindowExA
(100)
(101)    mov [hWnd], eax
(102)
```

GUI erstellen und Handle nach hWnd schieben.

```
(103)
(104)    push dword 1                      ; show the window
(105)    push dword [hWnd]
(106)    call ShowWindow
```

GUI sichtbar machen.

```
(107)
(108)    ;;;;;; EDIT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(109)
(110)    push dword 0                      ; lParam
(111)    push dword [hInstance]           ; hInstance
(112)    push dword 2                      ; hMenu
(113)    push dword [hWnd]                ; hWndParent
(114)    push dword 180                    ; Height
(115)    push dword 275                    ; Width
(116)    push dword 5                      ; Y
(117)    push dword 5                      ; X
(118)    push dword &H50201045           ; WS_CHILD or WS_VISIBLE
or ES_AUTOHSCROLL
(119)    push dword 0                      ; lpWindowName
(120)    push dword szEditClass           ; ClassName
(121)    push dword &H00000200           ; dwExStyle
(122)    call CreateWindowExA
(123)
(124)    mov [hEdit1], eax
(125)
```

Edit Control erstellen und Handle nach hEdit1 schieben.

```
(126)    push dword 1                      ; show the Edit
(127)    push dword [hEdit1]
(128)    call ShowWindow
(129)
```

Edit anzeigen.

```
(130)    push dword txtSample
(131)    push dword [hEdit1]
(132)    call SetWindowTextA
(133)
```

Vordefinierten Text aus txtSample in das Edit schreiben.

```
(134)    ;;;;;; BUTTON ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(135)
(136)    push dword 0                      ; lParam
(137)    push dword [hInstance]           ; hInstance
(138)    push dword 1                      ; hMenu
```

```

(139)    push dword [hWnd]                ; hWndParent
(140)    push dword 20                    ; Height
(141)    push dword 275                   ; Width
(142)    push dword 187                   ; Y
(143)    push dword 5                     ; X
(144)    push dword &H50001000           ; WS_CHILD or WS_VISIBLE or BS_PUSHLIKE
(145)    push dword 0                     ; lpWindowName
(146)    push dword szButtonClass         ; ClassName
(147)    push dword 0                     ; dwExStyle
(148)    call CreateWindowExA
(149)
(150)    mov [hButton1], eax
(151)
(152)    push dword 1                     ; show the BUTTON
(153)    push dword [hButton1]
(154)    call ShowWindow
(155)
(156)    push dword txtButton
(157)    push dword [hButton1]
(158)    call SetWindowTextA
(159)
(160)    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(161)

```

Den Button auf die selbe Weise wie das Edit erstellen, anzeigen und mit Text aus txtButton füllen.

```

(162)
(163)    push dword MB_ICONSTOP
(164)    call MessageBeep
(165)

```

Error Ton ausgeben. Gleichzeitig die Bestätigung, dass alle GUI Befehle abgehandelt wurden.

```

(166)
(167)
(168)    ;;;;;; MESSAGELOOP ;;;;;;
(169)
(170)    sub esp, sizeof MSG                ; space for MSG Struct
(171)    mov [wMsg], esp
(172)

```

Struct der Größe einer Msg erzeugen und dann wMsg angleichen.

```

(173)    MessageLoop:

```

Sprungmarke des Message Loops.

```

(174)    push dword 0
(175)    push dword 0
(176)    push dword 0
(177)    push dword [wMsg]
(178)    call GetMessageA
(179)

```

GUI Message abfragen, mit wMsg als Referenzspeicher.

```

(180)    test eax, eax
(181)    jz ExitMessageLoop

```

(182)

Beenden, wenn es die Message befiehlt.

```
(183)      push dword [wMsg]                ; process the message
(184)      call TranslateMessage
(185)      push dword [wMsg]
(186)      call DispatchMessageA
(187)
```

Translate & Dispatch the message.

```
(188)      jmp  MessageLoop
(189)
```

Play it again, Sam.

```
(190)      ExitMessageLoop:
(191)      add  esp, sizeof_MSG              ; cleanup MSG struct
(192)
(193)      push dword [hWnd]                ; destroy the window
(194)      call DestroyWindow
(195)
(196)      push dword [hInstance]           ; unregister the window
class
(197)      push dword AppName
(198)      call UnregisterClassA
(199)
(200)      add  esp, sizeof_WNDCLASS        ; cleanup WNDCLASS
Struct
(201)      pop  ebp
(202)      ret
```

Aufräumen, wenn das Programm beendet wird. Struct der Message löschen (Speicher freigeben), Fenster löschen, Klasse wieder entfernen und die 40 Bytes der Klasse wieder freigeben. Die Rücklage in EBP wird ebenfalls gelöscht.

```
(203)
(204)      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(205)      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(206)
(207)      WndProc:
(208)      mov  eax, [esp+8]                  ; Message
(209)      cmp  eax, WM_DESTROY              ; WM_DESTROY?
(210)      jne  WndProcNext
(211)      push dword 0
(212)      call PostQuitMessage
(213)
(214)      WndProcNext:
(215)      push dword [esp+16]                ; copy arguments
(216)      push dword [esp+16]
(217)      push dword [esp+16]
(218)      push dword [esp+16]
(219)      call DefWindowProcA
(220)
(221)      ret 16
```

Hauptfunktion des Fensters. Empfängt die GUI Messages und beendet das Programm bei WM_DESTROY

mittels PostQuit Message, ansonsten leitet es einfach die Message durch.

Nun lasse ich dich damit alleine. LASM ist nur zum Einsteigen, schnell kannst du auf aktuelle Assembler wie MASM oder FASM umsteigen, zu welchen es auch eine große Gemeinschaft im Internet und fast immer Hilfe gibt.

[Weitere] ASM 8086 Befehle:

AAA	ASCII-Anpassung nach einer Addition
AAD	ASCII-Anpassung vor einer Division
AAM	ASCII-Anpassung nach einer Multiplikation
AAS	ASCII-Anpassung nach einer Subtraktion
ADC	Addiere mit Übertrag-Statusbit
ADD	Addiere
AND	Und
CALL	Rufe auf
CBW	Wandle Byte in Word um
CLC	Übertrag-Statusbit
CLD	Lösche Richtung-Statusbit
CLI	Lösche Interrupt-Statusbit
CMC	Invertiere Übertrag-Statusbit
CMP	Vergleiche
CMPS	Vergleiche Zeichenkette
CMPSB	Vergleiche Zeichenkette byteweise
CMPSW	Vergleiche Zeichenkette wordweise
CWD	Wandle Word in DWord um
DAA	Dezimale Anpassung nach einer Addition
DAS	Dezimale Anpassung nach einer Subtraktion
DEC	Vermindere
DIV	Dividiere (vorzeichenlos)
HLT	Halt
IDIV	Ganzzahlige Division (vorzeichenbehaftet)
IMUL	Ganzzahlige Multiplikation (vorzeichenbehaftet)
IN	Lese ein von
INC	Erhöhe
INT	Interrupt
INTO	Interrupt, wenn Überlauf
IRET	Kehre von Interrupt zurück
LAHF	Lade AH mit den Statusbits
LDS	Lade Pointer unter Verwendung von DS
LEA	Lade effektive Adresse in Register
LES	Lade Pointer unter Verwendung von ES
LOCK	Verriegle den Bus
LODS	Lade Zeichenkette in AL/AX
LODSB	Lade Zeichenkette byteweise in AL
LODSW	Lade Zeichenkette wordweise in AX
LOOPNE	Führe Schleife CX-mal aus solange ungleich
LOOPNZ	Führe Schleife CX-mal aus solange nicht 0
LOOPZ	Führe Schleife CX-mal aus solange 0
MUL	Multipliziere (vorzeichenlos)
NEG	Negiere
NOP	Keine Operation
OUT	Gebe aus nach
POPF	Pop flags
PUSHF	Push flags
RCL	Rotiere durch Übertrag-Statusbit nach links
RCR	Rotiere durch Übertrag-Statusbit nach rechts
REP	Wiederhole CX-mal

```

REPE Wiederhole CX-mal solange gleich
REPNE Wiederhole CX-mal solange nicht gleich
REPZ Wiederhole CX-mal solange nicht gleich 0
REPZ Wiederhole CX-mal solange 0
ROL Rotiere nach links
ROR Rotiere nach rechts
SAHF Speichere AH in die Statusbits
SAL Schiebe arithmetisch nach links
SAR Schiebe arithmetisch nach rechts
SHL Schiebe logisch nach links
SHR Schiebe logisch nach rechts
STC Setze Übertrag-Statusbit
STD Setze Richtung-Statusbit
STI Setze Interrupt-Statusbit
STOS Speichere Zeichenkette aus AL/AX
STOSB Speichere Zeichenkette byteweise aus AL
STOSW Speichere Zeichenkette wordweise aus AX
WAIT Warte auf Test-Signal
XLAT Setze Byte nach AL um

```

Inline Assembler in Autolt

Auch dies sei Teil dieses Tutorials. Aber nein, wir werden nicht gleich mit großen UDFs Sprünge in die Inline-Welt machen sondern uns zuerst die sehr kompakte LASM Inline DLL vornehmen, mit der wir ASM Code direkt in Autolt nutzen können. Später upgraden wir auf "bessere" Assembler.

So, was wir brauchen wäre zunächst einmal die LASM.au3, welche automatisch die LASM.dll installiert. Dazu gesellen sich wie du im Ordner siehst, 6 Beispiele. Nummiert nach ihrem Schwierigkeitsgrad.

Autolt Funktionen im ASM Code

Nun denn, öffnen wir das erste Beispiel "01 Use Autolt Functions.au3" und sehen was wir da haben:

```

#include <LASM.au3>

$PAu3Callback = LASM_Ptr("Test", "int", "")

# call $PAu3Callback$
# ret

$PTest = LASM_ASMTToMemory(LASM_GetInline())
MsgBox(0, "LASM", "Return = " & LASM_CallMemory($PTest, 0, 0, 0, 0))

Func Test()
    MsgBox(0, "Au3", "Hello!")
    Return 42
EndFunc

```

Erfrischend wenig Code, nicht? Dieses Beispiel verdeutlicht, wie man von Inline ASM Code auch Autolt Funktionen aufrufen kann. Die Autolt Funktion ist in diesem Fall `Test()`, welche einfach nur eine MessageBox ausgibt und den Wert 42 zurückgibt. LASM Inline Code wird durch ein # vor jeder ASM Zeile gekennzeichnet. Hier sind es nur zwei Zeilen, die eine ruft die Autolt Funktion auf, die andere kehrt zu Autolt zurück und gibt dabei das Ergebnis des Calls, also 42 aus. Autolt Variablen können direkt mittels \$Var\$ im Inline-Code genutzt werden, sie müssen aber vor dem Code definiert sein!

Nach dem Code wird LASM zum kompilieren aufgerufen. `LASM_ASMTToMemory(LASM_GetInline())` gibt einen Pointer zurück auf eine Speicherstelle, die den bereits kompilierten LASM Code enthält. `$pTest` kann nun immer mittels `LASM_CallMemory` aufgerufen werden und muss nicht erneut kompiliert werden. Es ist möglich vier reine Argumente an den ASM Code zu übergeben. Hier sind alle vier 0, da wir kein Argument übergeben wollen / im Code berücksichtigen.

Autolt Variabeln im ASM Code

Autolt Variablen können überall im ASM Code benutzt werden, wie Beispiel 2 zeigt:

```
#include <LASM.au3>

Local $Variable = 1111*1111 ; Random value

# mov eax, [Data]
# ret 16
# Data: dd $Variable$

$pTest = LASM_ASMTToMemory(LASM_GetInline())
MsgBox(0, "LASM", "Data = " & LASM_CallMemory($pTest, 0, 0, 0, 0))
```

Hier wird die Variable in ein Datenfeld geschrieben, vom ASM Code manuell nach EAX geladen und dann als Rückgabewert benutzt. Eigentlich ganz einfach & bequem.

CPUID lesen / Übergabe von Structs / bearbeiten von Structs

Hier haben wir Beispiel 3. Es liest den Namen der CPU mittels CPUID aus und gibt den String in einer Struct zurück:

```
#include <LASM.au3>

; Make room for returned string:
Local $strCPU = DllStructCreate("char[20]")

# pushad
# mov edi, [ebp+12]
# xor eax, eax
# cpuid ; This is a comment \o/
# mov [edi+0], ebx
# mov [edi+4], edx
# mov [edi+8], ecx
# popad
# ret 16

$pTest = LASM_ASMTToMemory(LASM_GetInline())
LASM_CallMemory($pTest, DllStructGetPtr($strCPU), 0, 0, 0, "ptr")

MsgBox(0, "LASM", DllStructGetData($strCPU, 1))
```

Für den String wird eine Struct erstellt. Jedes Zeichen stellt ein BYTE dar. Wir benutzen hier 20 als Länge, länger wird der Name nicht ausfallen. Wir übergeben die Struct als ersten Parameter und setzen den Typ von Parameter 1 auf ptr, da es ein Pointer auf die Struct ist.

Alle anderen Beispiele zu LASM Inline sollten selbsterklärend sein. Nimm dir einfach etwas Zeit sie durcharbeiten, benutze Sprachreferenzen aus dem Internet zu den Befehlen und stelle falls nötig die

Frage in einer ASM Community oder im AutoIt-Forum.

ASM-Inline wie der Führer eines Geschäftes

Hier kann ich nichts anderes tun als auf Andys Tutorial zu verweisen. AssembleIt und FASM UDF und wasweißich sind mir persönlich viel zu übertrieben, LASM kann da mit seiner Leichtigkeit trotz eingebautem Syntax-Check auch mithalten, wenn es nur um Inline-ASM geht. (/me sieht den Hammer-auf-Kopf-Smiley von Andy...)

64bit Assembler

Nun haben wir uns durch bis zu diesem Thema gearbeitet.

Lange wird nun schon 32bit Assembler genutzt, jedoch werden 32bit CPUs und Betriebssysteme zunehmend durch 64bit Systeme ersetzt und der zugrundeliegende ASM Code ändert sich ebenfalls. Ich werde nur einen kurzen Überblick über die Struktur des 64bit ASM geben.

Architektur

Die Register ändern sich zum 32bit ASM natürlich um einiges. 64bit Register können in vielen verschiedenen Größen und Arten angesprochen werden. So gesellen sich zu bekannten Typen noch einige erweiterte:

Typ	Breite	
Byte	8bit	
Word	16bit	
Double Word	32bit	
QuadWord	64bit	<- neu
Double QuadWord	128bit	<- neu

General Purpose Register gibt es folgende:

```
RAX [+++++ ]
RBX [+++++ ]
RCX [+++++ ]
RDX [+++++ ]
RBP [+++++ ]
RSI [+++++ ]
RDI [+++++ ]
RSP [+++++ ]
R8  [+++++ ]
...
R15 [+++++ ]
```

Es zeigen sich 16 GP Register. Die ersten 8 sind aus historischen Gründen nach den alten Registern benannt, während der Rest schlicht nummeriert ist. Wie man auch in 32bit den 16bit Modus nutzen kann, kann man hier durch Ersetzen des "R" bei den ersten acht Registern die 32bit Register ansprechen (z.B. EAX für RAX). Zusätzlich lässt sich auch das 16bit Register ansprechen. Dazu wird das R ganz entfernt und X eben durch L oder H ersetzt. Der Rest lässt sich fast genauso ansprechen:

```

R8  - QWord
R8D - lower QWord
R8W - lowest Word
R8B - lowest Byte Microsoft(r) Macro Assembler
R8L - lowest Byte Intel kompatible Assembler

```

Es gibt merkwürdige Beschränkungen beim Zugriff auf die Byte Register durch Komplikationen im REX OP-Code Prefix, welcher für neue Register benutzt wird. Ein Befehl kann auf kein geerbtes (historische Register) High-Byte (AH, BH, CH, DH) und auf ein neues Byte Register gleichzeitig zugreifen (z.B. R11B), außer bei geerbten Low Bytes (AL, BL, CL, DL). (AH, BH, CH, DH) müssen bei Befehlen mit REX Prefix stattdessen (BPL, SPL, DIL, SIL) benutzen.

Instruction Pointer und Flags

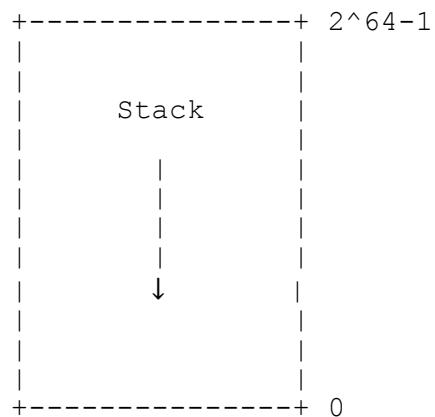
```

RIP      |           |           |
RFLAGS   | 0           | EFLAGS   |
          | 63          |           | 0

```

Der 64bit Befehlspointer RIP zeigt auf den jeweils nächsten Befehl, welcher ausgeführt wird und unterstützt das 64bit flat memory model (dazu später mehr).

Der Stackpointer RSP zeigt auf den letzten Eintrag, der auf den Stack gepusht wurde und inkrementiert sich selbst mit abnehmender Speicheradresse.



Das RFLAGS Register speichert flags, welche für Ergebnisse von Befehlen zurückgegeben werden, welche auf Prozessebene Rechnungen ausführen, genauso wie Flag Register in 32/16bit. Das RFLAGS Register formt sich aus dem x86 (32bit) Register EFLAGS und einem 32bit breiten höheren Speicher, der zur Zeit als reserviert und ungenutzt gilt. Im Folgenden siehst du die wichtigsten Flags. Alle anderen sollten vom Betriebssystem genutzt werden und immer den zuvor gelesenen Wert enthalten, falls geändert.

Symbol	Bit	Name	Gesetzt wenn...
CF	0	Carry	Befehl überträgt oder borgt
PF	2	Parity	letztes Byte eine gerade Zahl an Bits mit Wert 1 hat

AF	4	Adjust	es ein Übertragen von low Nibble zu high Nibble oder ein Borgen von high Nibble zu low Nibble der low order 8bit oder 16bit Zahl gab (z.B. in einer BCD [Rechnungen, bei denen Dezimalzahlen eine feste Anzahl an binär kodierten Bits einnehmen] Addition oder Subtraktion.)
ZF	6	Zero	Ergebnis war 0
SF	7	Sign	Bit des höchsten Wertes im Ergebnis war 1
OF	11	Overflow	Overflow in einer vorzeichenbehafteten Rechnungen stattgefunden hat
DF	20	Direction	Dieses flag wird benutzt um festzustellen, in welche Richtung (vorwärts oder rückwärts) mehrere Bytes von einer Speicherstelle zu einer anderen kopiert werden. Die Richtung ist eigentlich nur relevant, wenn die originale Position der Daten im Speicher und im Ziel gegenseitig überlappen.
ID	21	ID	Zeigt u.A. ob die CPUID Instruktion nutzbar ist

Die FPU (Floating Point Unit) enthält acht Register von FPR0 bis FPR7, Status- und Control Register und ein paar andere spezialisierte Register. FPR0-7 können Werte aufnehmen, die den folgenden Typen entsprechen:

Typ	Breite	Gen.bits	dez	Menge
Single Precision	32	24	7	$1.18 \cdot 10^{-38}$ - $3.40 \cdot 10^{38}$
Double Precision	64	53	15	$2.23 \cdot 10^{-308}$ - $1.79 \cdot 10^{308}$
Extended Precision	80	64	19	$3.37 \cdot 10^{-4932}$ - $1.18 \cdot 10^{4932}$

Floating Point Rechnungen sind konform zu IEEE 754 (ANSI/IEEE Std 754-1985; IEC-60559:1989 - International version). Viele Programmiersprachen unterstützen zwar die sog. Datentypen "float" und "double", allerdings meist nicht den 80bit-Typen. Es überlagern sich 8 weitere Register mit den Floating Point Registern, nämlich die 8 64bit breiten MMX Register. Sie teilen sich den Speicher miteinander.

Mehr zur 64bit Architektur, alle anderen Register und zusätzliche Befehle findest du z.B. im Intel® 64 and IA-32 Architectures Software Developer's Manual auf <http://www.intel.com/products/processor/manuals/> (Englisch). Dort gibt es fünf Teile, welche sich kostenlos als PDF herunterladen lassen. (Oder auf CD kaufen, aber das ist nicht nötig).

SIMD

Single Instruction Multiple Data (SIMD) Befehle führen einen einzelnen Befehl mit mehreren Daten parallel aus und sind nützlich für schnellere Assembler-Routinen. Von Intel eingeführte SSE oder MMX Befehle (welche die MMX und XMM Register benutzen) benutzen SIMD Operationen, welche bis zu 8 Datensätze parallel verarbeiten. Zum Beispiel können 8 Bytes zu den korrespondierenden 8 Bytes addiert werden, und das in einem Einzigen Befehl!

Die 64bit MMX Register MMX0-MMX7 überlagern den Speicherbereich über den FPR0-7 Registern, was heißt, dass ein Mix aus FP und MMX Operationen wohl durchdacht sein muss, damit nicht wichtige Werte überschrieben werden. Die MMX Register operieren mit Integer-Datentypen, erlauben als Byte, Word und DWord in parallelen Operationen. Die meisten MMX Befehle beginnen mit P für "Packed", z.B. PCMPGTB - "Compare packed signed byte integers for greater than" - "Vergleiche packed vorzeichenbehaftete Byte-breite Integer für "größer als" als wahre Aussage".

Die 16 128bit XMM Register erlauben parallele Ausführung von Operationen mit bis zu 4 Single oder 2 Double Precision Werten pro Befehl. Diese werden in SSE benutzt. Dazu finden sich einige Beispiele in Andys Tutorial.

- space for futher paragraphs -

Referenzen

- MASM32 Forum auf <http://masm32.com>
- AMD63 Architecture Tech Docs auf http://www.amd.com/usen/Processors/DevelopWithAMD/0,,30_2252_875_7044,00.html
- NASM Assembler (cp) auf <http://nasm.us>
- YASM auf <http://www.tortall.net/projects/yasm>
- FLAT Assembler FASM (cp) <http://flatassembler.net>
- - space for further references -

- EOF for now -

© minx (squarecode@web.de) - Document in Revision 5.9 - 7.9.2013

Dedicated to the people at AutoIt.de.

Dieses Dokument wurde mit WordPad erstellt.