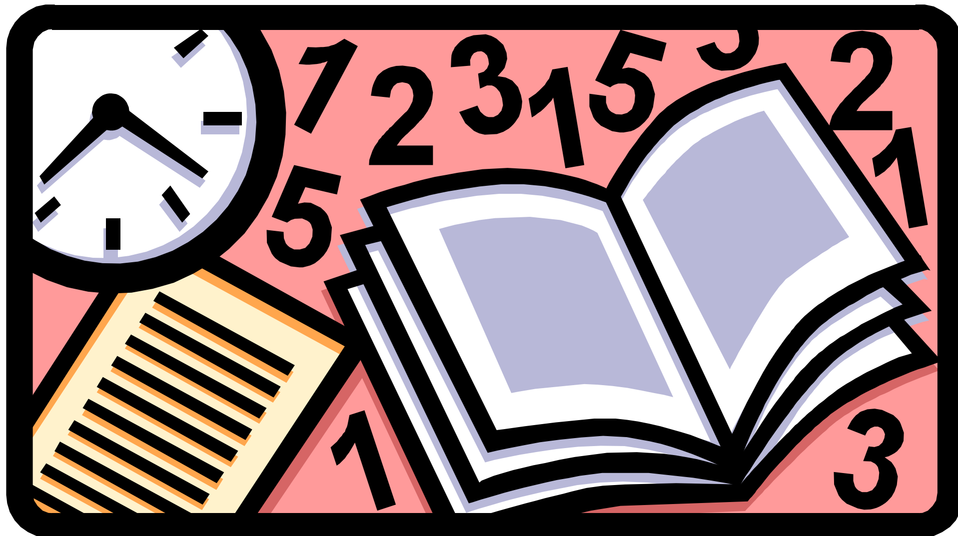
Assembler - Einführung 1. Teil

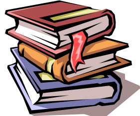
Vorstellung der wichtigsten Grundbegriffe und Grundkenntnisse sowie der wichtigsten Hardwarekomponenten



1. <u>Einleitung</u>

Wenn man durchs Internet surft kann man den Eindruck gewinnen, dass es keine deutschsprachige Internetseite gibt, die es fertig bringt eine Einführung in Assembler ins Netz zu stellen.

Aus diesem Grund hab' ich mich mit den letzten drei Büchern, die in der Buchhandlung noch zu haben waren bewaffnet und hoffe mit dieser kleinen Einführung und meinen bescheidenen Kenntnissen diesem Misstand ein Ende zu bereiten ;-)



Naja, viel Bestärkung habe ich allerdings bei meinen Recherchen nicht erhalten. Mal hörte ich „Was willste denn mit dem alten Quatsch. Mach lieber Java, das läuft immer und auf allem" oder „Äh Assembler ?. Das ist doch nur Tipperei !".

Wahr ist aber auch:

- ✓ Man hat endlich mal Gelegenheit zu verstehen wie ein Computer eigentlich funktioniert.
- ✓ Assembler gibt einem die **volle** Kontrolle über den Computer (einschließlich der Kontrolle über das Ticken der Systemuhr und vieles mehr...).
  → Daher ist der Assembler bei Systemprogrammieren immer noch geachtet.

- ✓ Assembler ermöglicht das Schreiben von sehr kompakten und extrem schnellen Programmen (..schneller als Programme, welche mit Hochsprachen erstellt wurden) .
  → Hacker lieben Assembler für diese Eigenschaften. Denn wer lässt sich schon einen Virus andrehen, der die Größe von Office 2000 hat und so lange geladen werden muss wie Windows 98 ??? .. ja ok, leicht übertrieben..;-)...und außerdem „ Allein der Gedanke ist verwerflich" ;-)

- ✓ Funktionen, welche man bei problemorientierten Hochsprachen schmerzlich vermisst sind in Assembler leicht zu realisieren.
- ✓ Assemblerprogramme lassen sich durchaus in Hochsprachenprogramme einbinden.

2. Die Maschinensprache - Grundlagen

Ja, welche Sprache versteht denn der Prozessor (CPU) des Computers?
Nun, es ist eigentlich keine Sprache sondern Zahlen. Wollen wir direkt der CPU des Computers einen Befehl erteilen, müssen wir dies mittels einer Zahl tun. Die Gesamtheit der Befehle die ein Prozessortyp kennt nennt man seinen Befehlssatz.
Die Dezimalzahl 10 bedeutet: addiere zwei Zahlen. Die Zahl 5 kann die CPU zum Beispiel anweisen den Inhalt eines Speicherplatzes an eine andere Speicherstelle zu verschieben usw.. Der 8086 , der Urahne unseres Pentium III, kennt 90 Grundbefehle. Jeder Befehl hat zu seiner Identifikation eine eigene Zahl. Diese Befehle versteht unser Pentium III auch, da er ja ein Nachkomme des 8086 ist. Der Pentium III hat jedoch noch zusätzliche Befehle bekommen, die jedoch dem 8086 überhaupt nichts sagen. Wir können also alle Befehle des 8086 durchaus auch bei nachfolgenden Prozessorgenerationen einsetzen.

Es ist völlig egal, welche Programmiersprache wir einsetzen. Am Ende übersetzt der jeweilige Sprachencompiler die Additionsanweisung des Quellcodes in die Zahl 10. Diese kann dann an die CPU geschickt werden. Nun, aber warum gibt es dann mehr Programmiersprachen als Sand am Meer ? Ganz einfach: Der Arbeitsschritt eines einzelnen Grundbefehls ist sehr klein. Es ist daher sehr verführerisch mehrere Grundbefehle zu einer Anweisung zusammenzufassen.
Jede Hochsprache (C, C++, Cobol, Pascal..) macht dies mit einem anderen Hintergedanken.
Cobol zum Beispiel fasst Grundbefehle so zusammen, dass Anweisungen entstehen, welche besonders gut kaufmännische Probleme lösen helfen. Will man aber wissen wie viel Speicherplatz auf der Festplatte noch vorhanden ist tut man sich mit Cobol etwas schwer.

Das menschliche Gehirn hat aber mit Zahlen in aller Regel Probleme. Vor allem, wenn es gleich 90 sind und es sich auch noch zu jeder Zahl den dazugehörigen Grundbefehl merken muss. Deshalb hat man zu einem Trick gegriffen. Und der heißt Assembler. Statt der Dezimalzahl 10 merkt man sich einfach den Mnemonic ADD (was natürlich für ADDition steht). Entsprechend wählt man für die Zahl 5 den Mnemonic MOV (was, wer hätt's gedacht, für MOVe steht). Nein falsch, Mnemonic ist kein Tippfehler, sondern ein Kunstwort aus memory und name (ob Keanu Reeves das weiß ? Ich mein „...und wie es funktioniert ?“). Wir schreiben in unserem Quellcode einfach ADD und MOV. Der Assembler macht daraus die Zahlen 10 sowie 5 und die CPU addiert und bewegt. Was mit wem addiert und von wo nach wohin bewegt wird kommt später.

Aber auch die Befehlssätze der einzelnen Prozessortypen unterscheiden sich voneinander.
Ein Prozessor für eine Spielekonsole hat mehr Befehle, welche die Erzeugung von Bildern, Tönen und Bewegungen unterstützen. Für einen Prozessor in einer Steuerung für Produktionsabläufe wären sie vermutlich überflüssig.

Man sieht also recht deutlich, dass nicht jeder Maschinencode auf jedem Prozessor läuft.

3. Die Central Processing Unit (CPU) oder einfach: Prozessor des PC's

Nähern wir uns nun vorsichtig dem eigentlichen Objekt der Begierde.

Eine CPU besitzt erwartungsgemäß ein **Rechenwerk**, denn sie soll ja addieren können. Das Rechenwerk kann aber auch Subtrahieren, Multiplizieren und Dividieren. Das Rechenwerk kann sogar zusätzlich noch logische Bedingungen erstellen und prüfen.

Was aber dekodiert die Grundbefehle die wir nacheinander in die CPU schieben und entscheidet was zu tun ist? Hierfür ist das **Steuerwerk** in der CPU zuständig.

Nun muss es aber auch noch Speicherplätze in der CPU geben, wo wir unsere Grundbefehle und Daten parken können bis die CPU Zeit hat sie von dort einzulesen. Zusätzlich sind auch Speicherplätze auf der CPU notwendig an denen die CPU ihre Arbeitsergebnisse für uns zur Abholung bereithält. Diese Speicherplätze nennt man **Register**.

Als besonderen Service der CPU gibt es noch Speicherplätze auf der CPU, deren Inhalt uns über den Erfolg oder Misserfolg eines Arbeitsschrittes informieren. Außerdem können an diesen Speicherplätzen noch generelle Anweisungen gegeben werden die dann für die gesamte Zeit der Ausführung der Grundbefehle gelten.
All diese Speicherplätze werden unter dem Begriff **Statusflags** zusammengefasst.

Schauen wir uns mal die Register zu Beginn etwas genauer an. Mit ihnen haben wir am meisten zu tun. Im Mittelpunkt stehen die 4 Allzweckregister.

Sie haben folgenden Aufbau:

Beim 8086 Prozessor galt:

Ein Register ist eine Speicherort bestehend aus 16 Bits. Ein Bit ist ein nicht mehr weiter zu unterteilender Speicherplatz. Der Inhalt eines Bits kann sozusagen nur eine 1 oder eine 0 sein.
Ein „leeres" Bit kann es nicht geben, denn entweder ist das Bit gesetzt oder es ist nicht gesetzt.

Ein Register:

Bit 15 Bit 14 Bit 13 Bit 12 Bit 11 Bit 10 Bit 9 Bit 8 Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0

Low – Teil

High – Teil

Ein Register lässt sich in zwei 8- Bit- Register unterteilen. Der hohe und der tiefe Teil lassen sich getrennt ansprechen. Man kann aber auch alle 16 Bits auf einmal auslesen oder beschreiben.

Jedes der 4 Allzweckregister hat einen eigenen Namen und einen eigenen Kennbuchstaben, sowie einen hauptsächlichen Verwendungszweck.

Das erste Register trägt den Namen Akkumulator und trägt den Kennbuchstaben A.
Will man zum Ausdruck bringen, dass man alle 16 Bit des Registers meint, so hängt man noch ein X an. Die Kennung lautet dann AX. Meint man nur Bit 0 bis einschließlich Bit 7, so schreibt man AL (Akkumulator- Low). Für Zugriffe auf Bit 8 bis Bit 15 schreibt man AH (Akkumulator – High). Das Register AX wird hauptsächlich im Zusammenhang mit arithmetischen Operationen und logischen Vergleichen verwendet.

Das zweite Register trägt den Namen Basisregister mit Kennbuchstaben B.
Für den niedrigen Teil schreibt man BL, für den hohen BH und für das Register als Ganzes BX.
Verwendung findet diese Register vor allem für Zugriffe auf den Arbeitsspeicher des PC's.

Das dritte Register trägt den Namen Countregister mit Kennbuchstaben C.
Für den niedrigen Teil schreibt man CL, für den hohen Teil CH und für das Register als Ganzes CX.
Verwendung findet diese Register vor allem wenn es sich ums Zählen dreht. Das ist vor allem bei Schleifen der Fall.

Das vierte Register trägt den Namen Datenregister mit Kennbuchstaben D.
Für den niedrigen Teil schreibt man DL, für den hohen Teil DH und für das Register als Ganzes DX.
Verwendung findet diese Register vor allem mit dem Register AX zusammen, wenn Multiplikations- und Divisionsaufgaben anstehen.

Bevor wir uns nun der Statusflags bemächtigen können müssen wir uns leider mit einem besonderen Zahlensystem (Binäres Zahlensystem) beschäftigen.

4. Das binäre Zahlensystem

Eine Dezimalzahl darf nur die Ziffern 0 bis 9 beinhalten.

Zahlen die **nur** aus den Ziffern **1** und **0** bestehen haben einen besonderen Namen.
Es handelt sich dabei um „**Dualzahlen**". Jedes Zahlensystem hat eine sogenannte Basis. Diese Basis entspricht der Anzahl der im Zahlensystem zur Verfügung stehenden Ziffern. Im binären Zahlensystem ist die Basis = 2.

Es stellt sich nun die Frage : „Wie rechnet man eine Dualzahl gebildet aus auf-
einander folgenden Bitnummern um in
eine Dezimalzahl ?"
Ganz einfach:
Betrachten wir doch einmal die Dualzahl in AL im Beispiel weiter oben.

$1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 10$

⇧ ⇧

Das erste Bit einer Dualzahl von rechts ist das niedrige Bit. Hier beginnt man immer mit der Bitnummer 0.

Kann man die Dezimalzahl 255 noch in das Register AH schreiben ?

255 : 2 = 127 Rest 1 ⇐ niedriges Bit (1. Ziffer von rechts)
127 : 2 = 63 Rest 1
63 : 2 = 31 Rest 1
31 : 2 = 15 Rest 1
15 : 2 = 7 Rest 1
7 : 2 = 3 Rest 1
3 : 2 = 1 Rest 1
1 : 2 = 0 Rest 1 ⇐ hochwertiges Bit

Antwort: Ja, denn sie benötigt nur 8 Bits. Die Dezimalzahl 256 könnte man nicht mehr nach AH schreiben.

4.1 Addition und Subtraktion im binären Zahlensystem

Die Addition und Subtraktion im binären Zahlensystem ist etwas eigenwillig und gewöhnungsbedürftig.

4.1.1 Addition

Erst mal das, was uns noch vertraut ist:

```
  0        0      1      jetzt aber:   1    Die 1 ist der Übertrag !
+ 0      + 1    + 0                + 1 1
---      ---    ---                -----
  0        1      1                  1 0
```

4.1.2 Subtraktion

```
  0        1      1          1 0
- 0      - 1    - 0          - 1
---      ---    ---          ---
  0        0      1            1
```

5. Das hexadezimale Zahlensystem

Im hexadezimalen Zahlensystem gibt es die Ziffern 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 sowie die „Ziffern“ A, B, C, D, E, F.

Die hexadezimale Zahl A steht für die dezimale Zahl 10.
Die hexadezimale Zahl B steht für die dezimale Zahl 11.
Die hexadezimale Zahl C steht für die dezimale Zahl 12.
Die hexadezimale Zahl D steht für die dezimale Zahl 13.
Die hexadezimale Zahl E steht für die dezimale Zahl 14.
Die hexadezimale Zahl F steht für die dezimale Zahl 15.

Welche Dezimalzahl entspricht der Hexadezimalzahl AFFE ?

$A * 16^3 + F * 16^2 + F * 16^1 + E * 16^0 = 45054$

⇧ ⇧

Das erste Bit einer Hexadezimalzahl von rechts ist das niedrige Bit. Hier beginnt man immer mit der Bitnummer 0.

Welche hexadezimale Zahl passt gerade noch in das Register AH ?

```
255 : 16 =   15  Rest  F   <==  niedriges Bit ( 1. Ziffer von rechts )
 15 : 16 =    0  Rest  F   <==  hochwertiges Bit
```

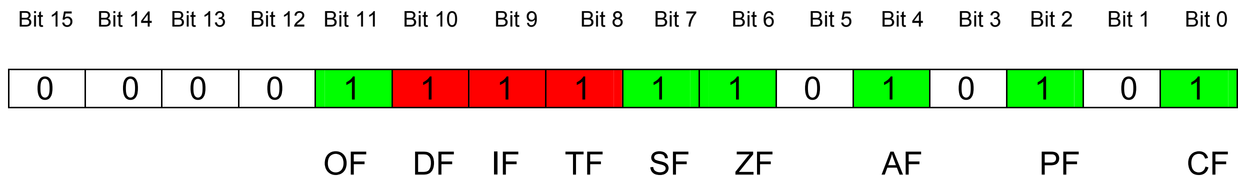
Antwort: Es ist die hexadezimale Zahl FF

Wie man Hexadezimalzahlen addiert und subtrahiert soll hier nicht gezeigt werden, da es für die Assemblerprogrammierung keine große Bedeutung hat.

So, nun sind wir gerüstet um uns mit den Statusflags auseinandersetzen zu können.

6. Die Statusflags

Ja, auch das Statusregister ist beim 8086er ein 16 Bit- Register. Von diesen 16 Bit werden jedoch nur 9 Bits benötigt. 6 Bits sind sogenannte Statusflags. Sie werden von der CPU gesetzt oder rückgesetzt. Die restlichen 3 Bits sind Kontrollbits. Sie werden vom Programmierer gesetzt oder gelöscht.



Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	1	1	1	1	1	1	0	1	0	1	0	1
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

6.1 Statusflags

Bit Nr. 0 nennt man Carry Flag (Übertragflag)

Es zeigt an, wenn nach einer Addition oder Subtraktion der Wertebereich in einem Register überschritten worden ist, wenn die Zahl des Ergebnisses zu viele Stellen hat. In diesem Fall steht in Bit 0 eine 1.
Mit anderen Worten: Das Carry Flag zeigt einen Übertrag **aus** dem höchstwertigen Bit an.

Beispiel: Im Register AH steht die Dualzahl 1111 0111. Es soll nun
zu dieser Zahl die Dualzahl 1000 0000 addiert werden.

```
    1 1 1 1   0 1 1 1

  + 1 0 0 0   0 0 0 0
  1
  -------------------
  1 0 1 1 1   0 1 1 1
```

Um das Ergebnis speichern zu können wären 9 Bits notwendig. Wir haben jedoch nur 8 Bits (1 Byte). Das Ergebnis kann nicht mehr in AH gespeichert werden.

Daraus folgt CF = 1. Dieses Flag kann jedoch durch den Befehl CLC gelöscht (CF = 0) und durch STC gesetzt werden (CF = 1).

Bit Nr. 2 Das Parity Flag

dient der Fehlerprüfung bei Datenübertragungen über sie serielle Schnittstelle. Dieses Flag wird auf 1 gesetzt, wenn das Ergebnis einer Operation in den niederwertigen 8 Bits (Low – Byte) eine gerade Anzahl an 1 en aufweist. Ansonsten ist PF = 0.

In unserem Beispiel: → PF = 0

```
    1 1 1 1   0 1 1 1

  + 0 0 0 1   0 0 0 0
    1 1 1 1
  -------------------
  1 0 0 0 0   0 1 1 1
```

Bit Nr. 4 Das Auxiliary Flag / (Hilfsübertragflag)

wird gesetzt (AF = 1), wenn ein Übertrag von Bit 3 nach Bit 4 erfolgt. Es wird hauptsächlich im Zusammenhang mit dem BCD Code verwendet.

```
    1 1 0 0  1 1 1 1
+   0 0 0 0  1 0 0 0
           1
--------------------
    1 1 0 1  0 1 1 1
```

Bit Nr. 6 Das Zero Flag (Nullflag)

Wenn das Ergebnis einer Vergleichs- oder arithmetischen Operation 0 ergab, wird dieses Bit gesetzt (ZF = 1).

```
    1 1 1 1  0 1 1 1
-   1 1 1 1  0 1 1 1
--------------------
    0 0 0 0  0 0 0 0
```

Bit Nr. 7 Das Sign Flag (Vorzeichenflag)

Auch Dualzahlen können Vorzeichen haben. Zur Darstellung von vorzeichenbehafteten Dualzahlen wird aber kein + oder – verwendet, sondern das höchstwertige Bit. Hierbei entspricht der Ziffer 1 das – und der Ziffer 0 das +.

Wenn wir die Dualzahl 0101 1110 als vorzeichenbehaftet betrachten erhalten wir:

1 0 1 0 0 0 1 0 ⇧ als Dezimalzahl – 94.

0 1 0 1 1 1 1 0 als Dezimalzahl + 94.

(Zweierkomplement)

Bit Nr. 11 Das Overflow Flag (Überlaufflag)

Dieses Bit wird gesetzt (OF = 1), wenn ein Übertrag **ins** höchstwertige Bit erfolgt.
(Nicht zu verwechseln mit CF → Übertrag **aus** dem höchstwertigen Bit).

```
    0111  0111

  + 0100  0000
    1
  ------------
    1011  0111
```

6.2 Kontrollflags

Bit Nr. 8 Trap Flag (Einzelschrittflag)

Durch setzten dieses Flags (TF = 1) schaltet der Prozessor in den Einzelschrittmodus. Dadurch kann man sich den Inhalt der Register nach jedem Grundbefehl anschauen und analysieren. Sehr vorteilhaft bei der Fehlersuche im Programmcode !

Bit Nr. 9 Das Interrupt Enable Flag (Unterbrechungsflag)

Gesetzt wird dieses Bit durch den Befehl STI. Rückgesetzt kann das IF mit dem Befehl CLI werden. Es kommt vor, dass ein Programm seinen vorgesehen Ablauf nicht einhalten kann. Durch (IF = 0) kann zum Beispiel verhindert werden, dass ein Programm durch das Drücken von STRG + C abgebrochen wird .

Bit Nr. 10 Das Direction Flag (Richtungsflag)

Wird eine Zeichenkette (String) im Speicher abgelegt, so muss jeder Speicherplatz eine eigene Adresse haben, damit man weiß an welchem Ort ein Zeichen abgelegt wurde. Soll eine solche Zeichenkette verarbeitet werden, müssen die einzelnen Adressen der Reihe nach aufgerufen werden.

DF = 1 Stringverarbeitung nach aufsteigenden Adressen.
DF = 0 Stringverarbeitung nach absteigenden Adressen.

7. Arbeitsspeicher

Unter dem Arbeitsspeicher kann man sich so etwas wie einen Notizblock des Prozessors vorstellen. Er kann in den Arbeitsspeicher schnell Daten auslagern, die in seinen Registern im Moment keinen Platz mehr haben. Andererseits kann er sie im Bedarfsfall auch schnell wieder zurückholen.

Natürlich muss man wissen an welchen Ort man die Daten geschrieben hat. Und da kommen die Adressen ins Spiel.

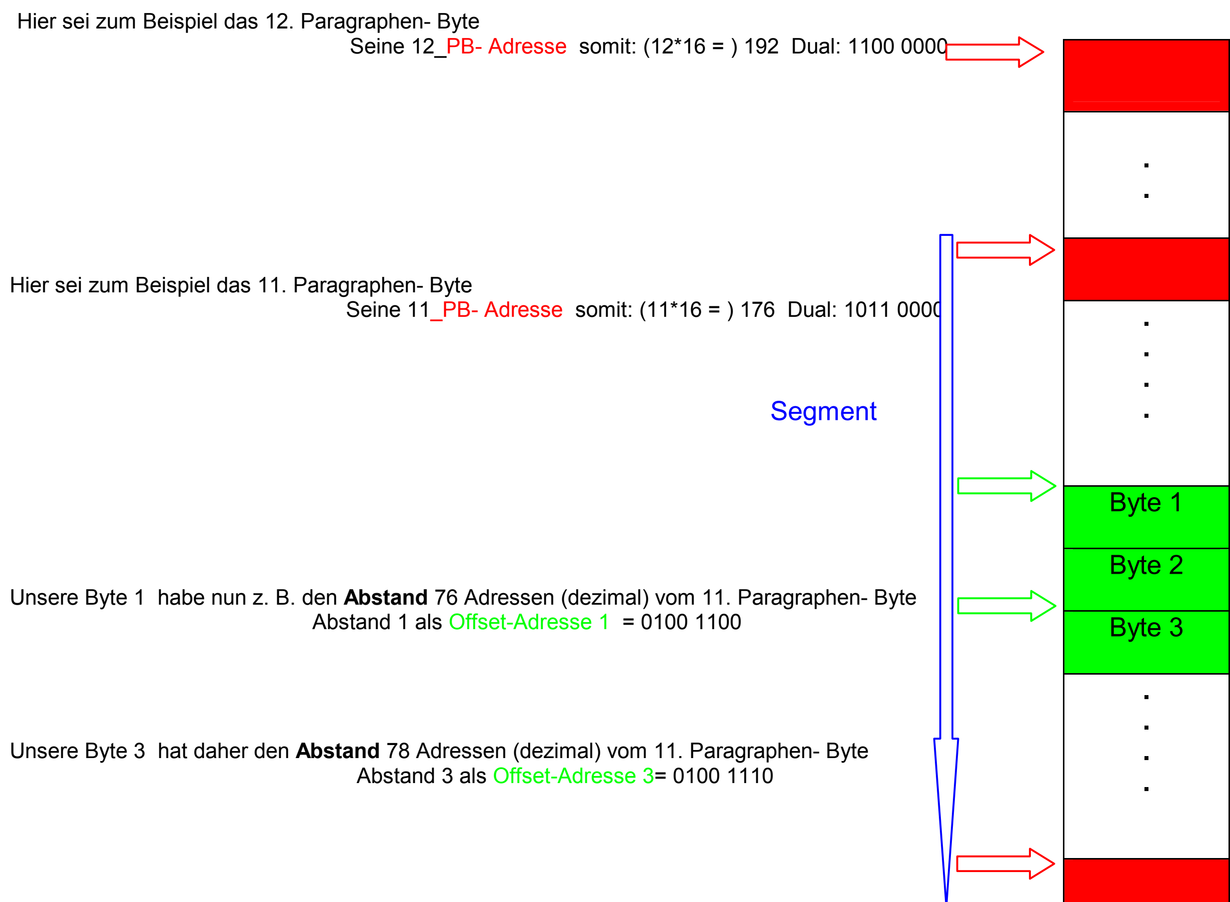
7.1 Adressbildung

Der Prozessor kann die Adressen von Speicherplätzen nur an Registern ausgeben. Die Register sind beim 8086er aber auf eine 16 Bit Architektur ausgerichtet. Mit 16 Bit können wir die Dezimalzahlen 0 bis 65 535 darstellen. Wir könnten also mit einem Register 65 535 Byte durchnummerieren. Jedes Byte hätte damit eine eigene Zahl und somit eine eigene Adresse.
Jetzt waren 65 535 Byte aber auch für damalige Verhältnisse keine berauschende Speicherkapazität. Schon für einen 1MB großen Arbeitsspeicher hätte man ein 20 Bit Register benötigt, denn man hätte 1 048 576 Byte adressieren können. Das war aber beim 8086er so eine Sache. Also musste man sich anders helfen.
Bei Intel hat sich deshalb ein schlauer Kopf damals etwas einfallen lassen.

Obwohl der 8086er nur 16 Bit Register hatte, war er an einen 20 Bit Adressbus angeschlossen. Das bedeutet der 8086er hätte eine 20 Bit Adresse abschicken können, was aber nicht ging, da er eben eigentlich nur mit 16 Bit breiten Adressen umgehen konnte.
Besagter schlauer Kopf hat sich nun folgendes überlegt: Wir nehmen die 1 048 576 Bytes und teilen diese Zahl durch 16. Was das bringt ? Na ja, das ergibt 65 535. Das bedeutet mit einem 16 Bit Register könnten wir jedem sechzehnten (16.) Byte eine Adresse geben. Diese Byte heißen Paragraphen. Die Paragraphen müssen also durch 16 teilbar sein.

Es entstand folgendes Konzept:

Das Bild soll einen Ausschnitt des Arbeitsspeicher darstellen. Die Fächer sind die einzelnen Speicherplätze.



Der blaue Pfeil bezeichnet einen Bereich im Arbeitsspeicher. So ein Bereich hat einen besonderen Namen: Segment. Das Segment beginnt mit einem Paragraphenbyte und umfasst maximal 64 KByte.

Mit der Adresse eines Paragraphenbyte kann man den Anfang eines Segmentes definieren. Diese Anfangsadresse nennt man daher **Segment- Adresse.**

Die **Offset- Adresse** eines Speicherplatzes ist der Abstand (gerechnet in Speicherplätzen) von diesem Paragraphenbyte im Arbeitsspeicher.

Die **vollständige** Adresse eines Speicherplatzes:
Segment- Adresse : Offset- Adresse

„Byte 1" wird daher in unserem Beispiel in folgender Form angegeben:
1011 0000 : 0100 1100

Übrigens:
Wenn in Windows 95 eine Anwendung an die Wand gefahren wird, kommt die allseits bekannte Meldung „Allgemeine Schutzverletzung". Nach anklicken von „Details" kommen dann einige Adressen in obiger Form (allerdings hexadezimal !) mit ihrem jeweiligen Inhalt (Hä ?).

Wie macht man jetzt eine 20- Bit- Adresse daraus ?

Es wird die Segment- Adresse mit 16 multipliziert . Im binären Zahlensystem bedeutet dies einfach das Anhängen von 4 Nullen. Dann addiert man einfach die so erhaltene Segment- Adresse und die Offset- Adresse und erhält eine 20- Bit- Adresse.

Segment- Adresse mal 16 :
+

0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0

Offset- Adresse:

				0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0

=

20- Bit- Adresse:

0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	0	1	1	0	0

Die so erhaltene 20- Bit- Adresse hat aber einen Schönheitsfehler. Und zwar kann es jetzt sein, dass ein und die selbe Speicherplatzstelle mehrere Adressen hat.

Sagen wir zum Beispiel : Das Segment beginnt am Paragraphenbyte 12. Unser „Byte1" hat zu dieser Segmentadresse einen Abstand von 92 Adressen.

Damit erhalten wir für die gleiche Speicherstelle „Byte 1“ die 20- Bit- Adresse:

```
  0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0    0 0 0 0

  0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1    1 1 0 0
+
-------------------------------------------

  0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 1    1 1 0 0
```

Zum Abschluss der Einführung kommen nun noch 4 Segmentregister sowie 5 Index- und Zeigerregister.

7.2 Die Zeiger- und Indexregister

Codesegment- Register (CS) → 16 Bit :

In diesem Register befindet sich eine Paragraphenbyte- Adresse mit der ein Segment beginnt in dem **Befehle** gespeichert werden.

Nun gibt es ein weiteres Register, den **Befehlszeiger (IP) → 16 Bit**.
Dieses Register beinhaltet die Offset- Adresse eines Befehls innerhalb dieses Codesegments.

Ein Speicherplatz an dem im Codesegment ein Befehl sitzt bekommt also die vollständige Adresse:
CS : IP

Datensegment- Register (DS) →16 Bit :

In diesem Register befindet sich eine Paragraphenbyte- Adresse mit der ein Segment beginnt in dem **Daten** gespeichert werden.

Extrasegment- Register (ES) → 16 Bit :

In diesem Register befindet sich eine Paragraphenbyte- Adresse mit der ein Segment beginnt in dem ebenfalls **Daten** gespeichert werden.

Reicht der Platz im Datensegment nicht aus, so kann das ES als zusätzlicher Datenspeicher für Variablen verwendet werden. Sehr wichtig ist dieses Register für das Kopieren, Verschieben und Vergleichen von Zeichenketten.

Nun gibt es zwei weitere Register,
das **Source- Index (SI)** und das **Destination- Index (DI) Register → jeweils 16 Bit**.

Das **SI** und das **DI** Register unterstützen unter anderem das Datensegment- und Extrasegmentregister als Offset- Register bei der Adressierung der Speicherstellen.

Auch das Register BX kann mit DS verwendet werden.

DS : SI
DS : DI
DS : BX

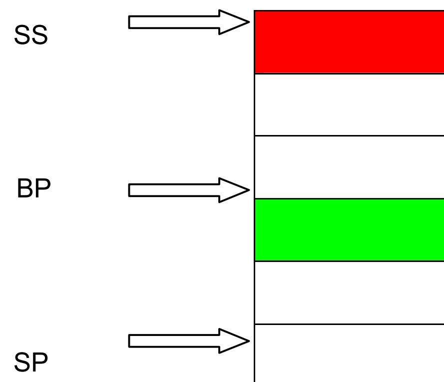
ES : SI
ES : DI
ES : BX

8. Der Stack

Der Stack ist ein Bereich des Arbeitsspeichers in dem Daten abgelegt werden, wie Schriftstücke auf einem Aktenstapel. Das neueste Schriftstück wird oben auf den Stapel gelegt. Das erste Schriftstück liegt somit ganz unten im Stapel. Im **Stacksegment- Register (SS)** steht wieder die Adresse eines Paragraphenbyte als Startadresse des Segments. Auch der Stack „wächst" von der höheren Adresse zur niedrigeren Speicheradresse!!! . Das **Stackpointer- Register (SP)** beinhaltet immer den Offset der niederwertigsten Speicherplatzstelle im Stacksegment, in die gerade Daten eingelesen wurden. Möchte man auf eine beliebige Speicherplatzstelle dazwischen zugreifen, muss zur Adressierung des Offsets das **Basepoint (BP)- Register** eingesetzt werden.

SS : SB
SS : BP

Im Stack sollen nur Daten nach dem LIFO (Last In , First Out) Prinzip abgelegt werden. Zuletzt eingelesene Daten sollen auch zuerst wieder ausgelesen werden.



9. Zusammenfassung