Inhaltsverzeichnis

1 Einführung

Während die CPUs (Central Processing Units) der ersten Computer noch mit Hilfe viele diskreter Bausteine aufgebaut und relativ einfach strukturiert waren, stieg im Laufe der Zeit die Integrationsdichte und immer weniger Chips bildeten einen Prozessor. Heutzutage sind vor allem Ein-Chip-Prozessoren, die Mikroprozessoren, die am weitesten verbreiteten CPUs. Aber nicht nur die Integrationsdichte, auch die Komplexität und Verarbeitungsgeschwindigkeit der Prozessoren nahm erheblich zu. Die ersten Mikroprozessoren arbeiteten mit einer Datenbreite von 8 Bit und boten wenig Funktionalität. Lange dominierten dann komplex aufgebaute 32-Bit-Prozessoren den Markt, aktuell sind heute 64-Bit-Prozessoren oder sogar noch größer. Das grundlegende Ziel dabei ist und bleibt, immer größere und kompliziertere Aufgaben in immer kürzerer Zeit zu erledigen.

Für den typischen Anwender ist es unwesentlich, wie seine CPU ihre Arbeit verrichtet und welche Vorgänge dabei ablaufen. Aus Anwendersicht zählt nur das Ergebnis. Der Programmierer ist dafür zuständig, dass das Programm die gewünschte Aufgabe erfüllt.

Jedoch selbst für Programmierer ist es in erster Näherung nicht entscheidend, auf welche Weise der Strom in seinem Rechner verbraucht wird, d.h. welche Signale über welche Leitungen gehen. Hauptsache, es erfolgt alles schnell genug. Es ist für ihn nicht so wichtig, wie die Daten vom Speicher in den Prozessor gelangen, was mit diesen Daten während ihrer Verarbeitung im Prozessor geschieht, was ein Cache ist, was er bewirkt, und welche Cache-Technologien oder -Größen beteiligt sind. Einem Programmierer, der ein Programm in einer höheren Programmiersprache (z.B. C oder C++) schreibt, sind die einzelnen Befehle, die der Prozessor tatsächlich ausführt, die sogenannten Maschinenbefehle, nicht mehr bekannt. Ein automatischer Übersetzer (Compiler) ist dafür zuständig, dass die Hochsprachenbefehle in für den Prozessor verständliche Einzelbefehle übersetzt werden. Bei Sprachen wie Java kommt sogar noch eine Indirektion hinzu, d.h. das compilierte Programm wird nicht vom Prozessor direkt, sondern von einer virtuellen Maschine ausgeführt, die ihrerseits erst die Übersetzung in die Plattform- und Prozessor-spezifischen Befehle vornimmt.

In manchen Situationen, z.B. beim Debugging, ist es jedoch sehr wohl von Vorteil, genau zu wissen, was ein Programm macht, d.h. die Befehle auf unterster Ebene zu verstehen. In dieser Vorlesung sollen deshalb die Vorgänge in Mikroprozessoren bei der Abarbeitung von High-Level-Befehlen genauer betrachtet werden. Darüber hinaus werden auch die im Laufe der Zeit entwickelten Maßnahmen zur Leistungssteigerung angesprochen. Dazu gehören dann auch die Fragen, wie die Befehle im Prozessor verarbeitet werden und welche Ressourcen zur Verfügung stehen. Die Beantwortung dieser Fragen beinhaltet zum einen einen Überblick über die Befehle eines Prozessors und deren interne Abarbeitung, zum anderen die Register und sonstigen Funktionseinheiten.

Um auf anschauliche Art und Weise einen Einblick in die Thematik zu bekommen, soll einmal betrachtet werden, wie ein Programmteil in einer höheren Programmiersprache (hier: C) in eine einem i386 oder vergleichbaren Prozessor verständliche Form gebracht werden kann. Betrachtet werden soll die folgende (mäßig sinnvolle) Schleife:

```
void test( void)
{
   int i, ende, var[10];
   for (i=0; i<ende; i++) {
      if (var[i] > 0) {
         var[i]++;
      } else {
         var[i]--;
      }
   }
}
```

Im Abschnitt "8086 Beispielprogramm" auf Seite 30 wird dieses Programm als Assembler-Programm gezeigt. Zum Verständnis sind dann zum einen die Kenntnis der (Assembler-)Befehle des Prozessors nötig, zum anderen gewisse Grundlagen seiner Architektur, z.B. welche Register für Daten im Prozessor zur Verfügung stehen.

Des weiteren ist es nützlich, einen allgemeinen Überblick über die Datentypen zu schaffen, d.h. über die verschiedenen Formen, in denen Daten von Prozessoren verarbeitet werden können (z.B. Integer oder Floating Point).

In den folgenden Kapiteln werden nach einer allgemeinen Einführung für jeden der behandelten Prozessoren diese Grundlagen geschaffen. Für die Intel-x86-Baureihe werden diese Grundlagen anhand des 8086 und 8087 diskutiert.

Beim der Diskussion des 8086 steht die Programmabarbeitung im Vordergrund, d.h. die Befehle, Register, Speicherzugriffe und Interrupts. Beim 8087 werden die Arbeitsweise einer Intel-Floating-Point-Unit sowie die Unterschiede zwischen Verarbeitung und Darstellung von fixed point (Integer) und floating point (Fließkommazahlen) betrachtet. Bei den Prozessoren 80286 und 80386 stehen die mit dem sog. "Protected Mode" eingeführten Sicherheitsmechanismen im Mittelpunkt.

Die von den Prozessor-Herstellern vorgenommenen Erweiterungen für 80286 und aufwärts (bis hin zu aktuellen Prozessoren) stehen im Fokus der folgenden Fragen:

- Wie können Systeme sicherer gemacht werden, d.h. welche Vorkehrungen können getroffen werden, dass nicht ein Programm ein ganzes System zum Absturz bringen kann. Zu der Zeit, als 8086 bzw. 80286 und 80386 aktuell waren, unter DOS und Windows bis Version 3.11, war dies ja ein häufiger Fall)?

  Diese Frage beinhaltet im Wesentlichen eine Betrachtung der Zugriffe eines Prozessors auf seine Daten und wird anhand des Protected Mode des i286 diskutiert.

- Wie kann die Performance gesteigert werden?

  Diese Frage stellte sich in einfacher Form natürlich schon für den 8086, ist dort jedoch von untergeordneter Bedeutung. In modernen Mikroprozessoren steht sie allerdings im Vordergrund. Ein wichtiges Thema ist deshalb die Verwirklichung dieser modernen Konzepte in Pentium und Pentium Pro.

2 Intel Mikroprozessoren Überblick

Die folgende Tabelle gibt einen Überblick über die behandelten Mikroprozessoren der Intel 80x86 Baureihe:

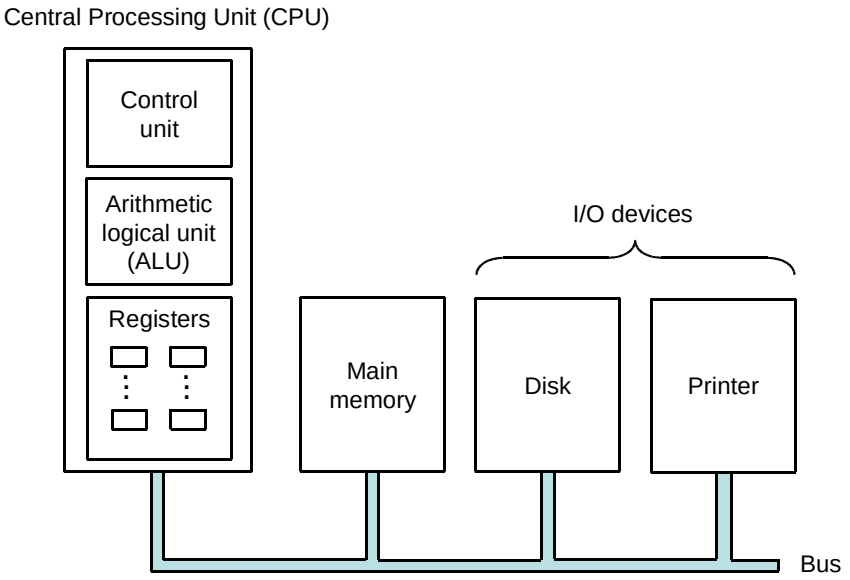
8086	Erster Prozessor der Baureihe; 16-Bit Prozessor; 1 MB Adressraum (20 Adressleitungen)
8088	Intern wie 8086, nach außen jedoch nur 8-Bit Datenzugriffe; 1 MB Adressraum (20 Adressleitungen)
8087	Mathematischer Co-Prozessor für den 8086 und 8088
80186	Intern wie 8086; verfügt über zusätzliche On-Chip Bausteine wie Taktgenerator, Timer, Interrupt-Controller, DMA und Chipselect-Logik; 1 MB Adressraum (20 Adressleitungen)
80286	16-Bit Prozessor; implementiert zusätzlich zum Real-Mode des 8086 den 80286 Protected Mode; 16 MB Adressraum (24 Adressleitungen)
80287	Mathematischer Co-Prozessor für den 80286 (auch für 80386 verwendbar)
80386 DX	32-Bit Prozessor; implementiert den 80386 Protected Mode; 4 GB Adressraum (32 Adressleitungen)
80386 SX	intern wie 80386 DX, extern jedoch nur 16-Bit Datenzugriffe; nur 24 Adressleitungen (16 MB physikalischer Adressraum)
80387	Mathematischer Co-Prozessor für den 80386 DX und SX
80486 DX	32-Bit Prozessor; implementiert den 80386 Protected Mode; enthält einen On-Chip Coprozessor (ähnlich dem 80387); 4 GB Adressraum (32 Adressleitungen);
80486 SX	wie 80486 DX, kein On-Chip-Co-Prozessor
80487	Mathematischer Co-Prozessor für den 80486 SX
Pentium (P5)	32-Bit Prozessor; implementiert den 80386 Protected Mode; Hardware-Design enthält viele RISC Konzepte, unter anderem eine Superscalar-Pipeline; 4 GB Adressraum (32 Adressleitungen)
Pentium Pro (P6)	Noch mehr RISC-Konzepte, intern ca. doppelt so schnell wie Pentium; ansonsten gleich wie P5: 32-Bit Prozessor, der den 386-Protected-Mode unterstützt

3 Grundlagen der Rechnertechnik

Bevor die Prozessoren im Einzelnen betrachtet werden, soll ein Modell über den grundsätzlichen Aufbau aller modernen Rechner skizziert werden.

3.1 Die Arithmetic Locigal Unit (ALU)

Ein Rechner besteht aus einer CPU (Central Processing Unit), die intern aus drei Komponenten aufgebaut ist: (a) Die Register, (b) die ALU (Arithmetic Logical Unit) und (c) einer Kontroll-Einheit, die steuert, welche Operation die ALU ausführt und welche Register benutzt werden.



Quelle: Stroetmann, Vorlesung Rechnertechnik, 2010

Die CPU ist über einen Bus mit dem Hauptspeicher und den Peripheriegeräten verbunden. Der Hauptspeicher enthält sowohl die zu verarbeitenden Daten als auch das Programm, das die Datenverarbeitung steuert. Eine Architektur, bei der Programm und Daten in derselben Einheit abgespeichert werden, wird üblicherweise als "Von-Neumann"-Architektur[1] bezeichnet.

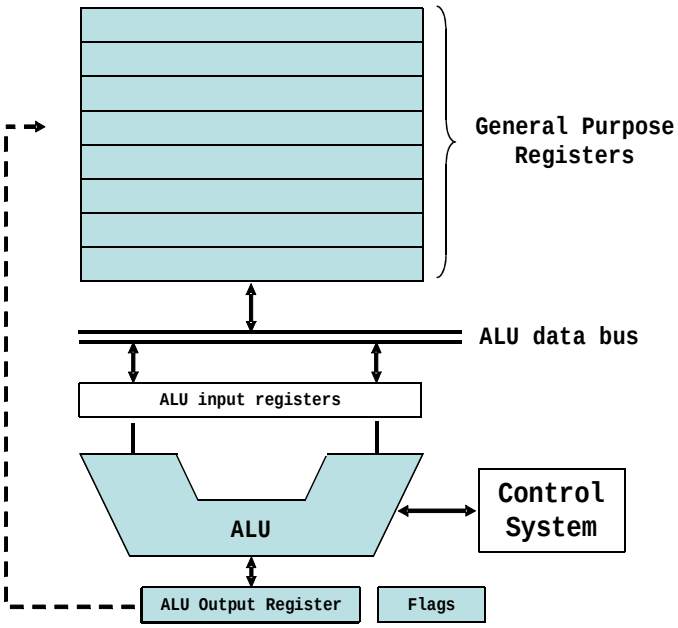
Erst die Tatsache, dass in dem Hauptspeicher sowohl Programm als auch Daten abgelegt werden, ermöglicht es, ein Benutzer-Programm zunächst als Daten einzulesen und dann diese Daten als Programm auszuführen. In der Entwicklung der Rechner war dies ein entscheidender Schritt. Bei den ersten Rechnern, beispielsweise bei dem von Konrad Zuse entwickelten Rechner Z3, waren Programmspeicher und Datenspeicher getrennt.

Der Ablauf einer Rechnung innerhalb der CPU läuft typischerweise von den allgemeinen, dem (Assembler-)Programmierer zugänglichen Registern über temporäre Register in die ALU. Von dort aus werden (üblicherweise über auch wieder über zwischengeschaltete temporäre Register) die

1 Diese Bezeichnung ist allerdings irreführend, da diese Architektur nicht von John von Neumann erfunden wurde. Der erste Rechner mit dieser Architektur war die Eniac, die von John William Mauchly und J. Presper Eckert gebaut wurde. Mauchly und Eckert hatten von Neumann die Eniac gezeigt und die Architektur erklärt. Dieser schrieb daraufhin ein Papier, das die Architektur der Eniac beschreibt.

Ergebnisse in die allgemeinen Register zurückgeschrieben. Dabei steuert eine Kontroll-Einheit, welche Register als Argumente der ALU ausgewählt werden, welche Operation durchgeführt und in welchem Register das Ergebnis abgelegt wird.

Die ALU enthält also logische Gatter, die die für eine Programm-Abarbeitung notwendigen arithmetischen, logischen und weiteren Operationen ermöglichen. Die Kontroll-Einheit enthält Gatter, die die ALU steuern und zwischen den verschiedenen zur Verfügung gestellten Operationen auswählen.



Last but not least soll noch angemerkt werden, dass eine ALU im weitesten Sinne nicht nur rein kombinatorische Schaltungen enthält, also Schaltungen "ohne Gedächtnis". Vielmehr kommen auch sequentielle Schaltungen mit Latches und Flipflops und endlichen Automaten zum Einsatz. Auf Details dieser Schaltungen soll im Rahmen dieser Vorlesung allerdings nicht weiter eingegangen werden. Hierzu sei auf die einschlägige Literatur verwiesen.

3.2 Maschinencode, Assembler, Programmiersprachen

Wie bereits erwähnt, wird die Funktion eines Prozessors durch den sogenannten Maschinencode gesteuert. Die Bits und Bytes aus dem Maschinencode steuern also letztendlich mehr oder weniger direkt die Kontroll-Einheit der ALU und geben damit vor, welche Operationen durchgeführt werden, ob z.B. eine Addition oder Multiplikation oder irgendeine andere Manipulation der eingehenden Daten erfolgen soll, um das Ergebnis zu bilden. Genauso erfolgt auch die Berechnung von Speicheradressen und das Laden und Abspeichern von Registern gesteuert durch den Maschinencode.

Man unterscheidet hierbei zwischen sogenannten RISC Prozessoren (Reduced Instruction Set Computer) und CISC Prozessoren (Complex Instruction Set Computer). Bei klassischen RISC Prozessoren werden nur relativ wenige Maschinenbefehle zur Verfügung gestellt, die alle direkt Gatter der ALU und Kontroll-Einheit steuern. Bei CISC Prozessoren erfolgt im Prozessor zunächst eine Übersetzung der Maschinensprache in sogenannten Mikro-Code, der letztendlich dann die Gatter steuert. Deshalb können CISC Prozessoren wesentlich mehr unterschiedliche Befehle zur Verfügung stellen, ohne dass grundsätzlich mehr Gatter benötigt werden.

Heutzutage enthalten alle aktuellen Prozessoren weitgehend eine Mischung von RISC und CISC Konzepten. Praktisch alle Prozessoren haben einen grundlegenden Satz von Instruktionen, die direkt in Hardware implementiert sind, als auch einen erweiterten Satz, der dann in Mikrocode umgesetzt ist.

Allerdings ist Maschinencode vom Menschen nur schwer lesbar. Deshalb werde Programme üblicherweise in Hochsprachen implementiert, die von sogenannten Compilern in Maschinencode übersetzt werden. Diese von einem Compiler zu leistende Übersetzung ist verhältnismäßig kompliziert, bekanntermaßen erleichtern Hochsprachen die Erstellung von Programmen allerdings erheblich bzw. machen umfangreichere Programme überhaupt erst möglich.

Zwischen Hochsprache und Maschinensprache gibt es allerdings noch eine Ebene, den sogenannten Mnemonischen Code, häufig auch Assembler-Code genannt, der vom Menschen noch hinreichend verstanden werden kann, der jedoch relativ 1:1 in Maschinencode übersetzbar ist.

Ein Befehl eines Assembler-Programms enthält meist nur die Anweisung für eine einzige Operation der ALU. Die dabei beteiligten Register müssen durch meist separate Anweisungen explizit geladen und wieder in den Speicher geschrieben werden. Soll der Ablauf des Programms verändert werden, muss oft der dazu notwendige Vergleich von Operanden und der darauf basierende bedingte "Sprung" an eine andere Stelle im Programm separat angewiesen werden.

Um also Programme in Assembler erstellen zu können, ist eine relativ genaue Kenntnis der Architektur und Möglichkeiten des Prozessors notwendig, für den das Programm erstellt wird. Assembler-Programme sind typischerweise auch nicht ohne größeren, meist manuellen, Aufwand von einer Prozessor-Architektur auf eine andere portierbar.

Beispiele für Befehle in mnemonischem Code sind:

- MOV R1, mem Kopiere Daten von Speicherstelle "mem" in das Register R1
- ADD R0, R3 Addiere Inhalt von R0 und R3 und speichere Ergebnis in R0
- JMP Zieladresse Springe an die durch "Zieladresse" angegebene Stelle im Programm

Anmerkungen:

- Üblicherweise steht das Ziel-Register ganz links
- Bei arithmetischen Operationen gibt es Prozessoren, die mehrere Operanden erlauben (ein ADD-Befehl, der die Operation R2=R1+R0 implementiert, könnte dann "ADD R2, R1, R0" sein) und Prozessoren, die grundsätzlich maximal zwei Operanden erlauben (ein ADD-Befehl, der die Operation R1=R1+R0 implementiert, könnte dann "ADD R1, R0" sein)
- Die Notation der mnemonischen Befehle als auch der Register unterscheidet sich nicht nur von Prozessor zu Prozessor, sondern sogar zwischen unterschiedlichen Assemblern für ein und denselben Prozessor

Mehr Details und Beispiele zur Assemblerprogrammierung werden anhand des 8086 Assemblers im Kapitel "4 Der Mikroprozessor 8086" auf Seite 15 behandelt und erläutert.

3.3 "Little Endian" vs. "Big Endian"

Eine wichtige Eigenschaft eines Prozessors ist es, wie Daten, die aus mehreren Bytes bestehen, im Speicher abgespeichert werden. Typischerweise ist die kleinste im Speicher adressierbare Einheit ein Byte.

Besteht nun ein Datum aus mehreren Bytes, so gibt es Prozessoren, die beginnend an einer Speicherstelle, zunächst das höchstwertige Byte abspeichern, danach alle weiteren Bytes. Soche Prozessoren speichern nach dem Big-Endian-Prinzip.

Dagegen gibt es auch Prozessoren, die zunächst das niederwertigste Byte speichern, also nach dem Little-Endian-Prinzip.

Historisch bedingt sind sowohl Little-Endian und Big-Endian ähnlich verbreitet, man muss sich also für einen Prozessor genau ansehen, wie Zahlen im Speicher abgelegt werden.

"Die Begriffe Big-Endian und Little-Endian benennen also dasjenige Ende der Zahlendarstellung, das in einer Reihenfolge an erster Stelle steht beziehungsweise an der kleinsten Adresse gespeichert wird." (aus Wikipedia)

Beide Varianten haben ihre Vor- und Nachteile. Während es für den Prozessor völlig egal ist, muss sich der Mensch beim Lesen von Speicherauszügen darauf einstellen.

Beispiele:

- Typischerweise werden Speicherauszüge, sogenannte Dumps, byteweise dargestellt, die Adressen aufsteigend von links nach rechts. Damit sieht ein Speicherauszug z.B. folgendermaßen aus:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1	**10**	**32**	**54**	**76**	00	00	00	00	00	00	00	00	00	00	00	00
2	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3	**76**	**54**	**32**	**10**	00	00	00	00	00	00	00	00	00	00	00	00
4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
6	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

- Die hexadezimale Zahl "76543210" würde von einem Big-Endian Prozessor wie an den Speicheradressen 30-33 gezeigt abgespeichert.
- Die gleiche Zahl würde von einem Little-Endian Prozessor wie an den Speicheradressen 10-13 gezeigt abgespeichert.
- Somit ist die im Big-Endian Format abgespeicherte Zahl für den Menschen im Speicherauszug leichter lesbar.

Hinweis:

Die in dieser Vorlesung näher betrachteten Prozessoren der x86-Baureihe arbeiten ausschließlich mit Little-Endian.

3.4 Interner Aufbau einer CPU

Um einen Prozessor, also eine CPU, vollständig zu beschreiben, sind folgende Schritte notwendig:

1. Beschreibung der Schnittstelle des Prozessors.
   Festlegen, über welche Signale der Prozessor mit der Peripherie verbunden ist und wann diese Signale welche Pegel annehmen.
   Dazu gehört zum Beispiel, dass ein Steuersignal "WR" oder "RD" beim Lesen bzw. Schreiben des Speichers bestimmte Pegel annimmt.
2. Festlegung der Register
   Welche Register sollen für die Daten innerhalb des Prozessors zur Verfügung gestellt werden?
   Dazu gehören die "von außen" sichtbaren Register, als auch die nur intern im Prozessor verwendeten "temporären" Register.
3. Spezifikation des Verhaltens des Prozessors.
   Erstellen einer mathematischen Beschreibung des Verhaltens des Prozessors durch bedingte Gleichungen.
   Für die Abarbeitung eines Maschinenbefehls benötigt der Prozessor mehrere Phasen. Für jede Phase wird mittels Gleichungen genau beschrieben, welche Register übertragen und welche Signale gesetzt werden.
4. Implementierung der verhaltensbasierten Beschreibung.
   Der Nutzen einer verhaltensbasierten Spezifikation liegt darin, dass eine solche Spezifikation es ermöglicht, das Verhalten der zu entwickelnden Schaltung zu testen. Dazu wird die in Schritt 3 erstellte mathematische Beschreibung des Prozessors in eine verhaltensbasierte Verilog-Beschreibung umgesetzt und implementiert.
5. Verhaltensbasierte Beschreibung der Peripherie
   Um die verhaltensbasierte Verilog-Implementierung testen zu können ist es erforderlich, auch die Peripherie des Prozessors, also den Hauptspeicher und die Einheiten zur Ein- und Ausgabe, verhaltensbasiert zu spezifizieren.
6. Implementierung des Prozessors auf Register-Transfer-Level (RTL)
   Um den Prozessor produzieren zu können, muss die verhaltensbasierte Beschreibung in eine Implementierung auf RTL-Ebene umgesetzt werden. Dazu müssen zunächst die Komponenten des Prozessors, wie z.B. das Register-File, der Datenpfad, und die Kontroll-Einheit, implementiert werden.
7. Prozessors implementieren.
   Der abschliessende Schritt ist die Beschreibung, wie die einzelnen Komponenten des Prozessors zu verbinden sind und welche Kontroll-Signale zur Steuerung des Datenflusses erforderlich sind.
   Die Steuerung der Kontroll-Signale wird von einer eigenen Einheit, der Kontroll-Einheit durchgeführt. Die Implementierung der Kontroll-Einheit ist ebenfalls in mehrere Schritte unterteilt:
   a) Spezifikation des Verhaltens der Kontroll-Einheit.
   b) Verhaltensbasierte Implementierung der Kontroll-Einheit.
   c) Implementierung der Kontroll-Einheit auf RTL-Ebene.

Die vollständige Behandlung aller o.g. Themen sprengt den Rahmen dieser Vorlesung.

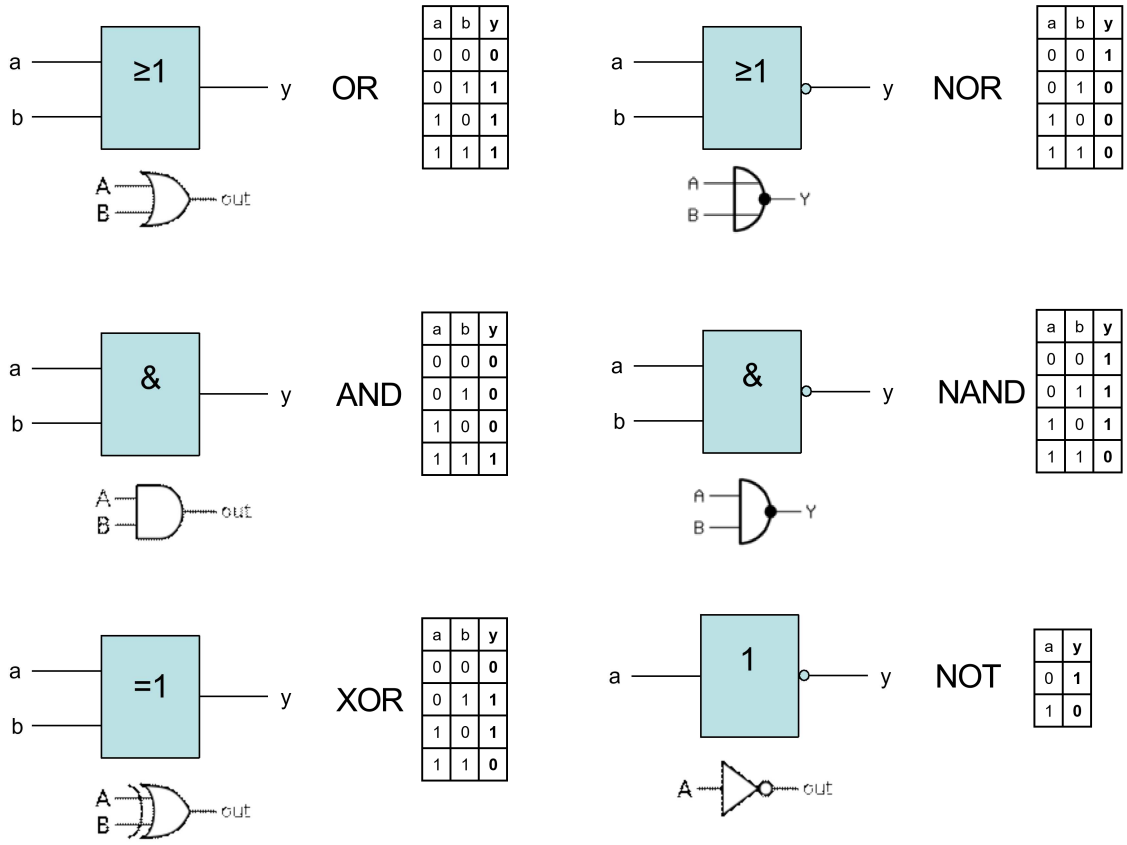
3.5 Logische Gatter

Die Hardware für Prozessoren, ihre CPUs und ALUs, wird durch logische Gatter implementiert. Diese werden dann wiederum über Maschineninstruktionen und Mikrocode gesteuert, um die notwendigen arithmetischen und logischen Operationen auszuführen, die der Prozessor zur Verfügung stellen soll.

In der Vorlesung werden die in der nachfolgenden Grafik dargestellten Gatter betrachtet. Wie man sieht, können Gatter durch unterschiedliche Symbole dargestellt werden.

- IEC (International Electrotechnical Commission)
- US ANSI
- DIN 40700 (nur vor 1976 verwendet)

In der Vorlesung werden bevorzugt die IEC Symbole verwendet.



Üblicherweise werden Schaltungen simuliert, bevor sie in Hardware erstellt werden. Eine Möglichkeit dieser Simulation ist die Beschreibung der Schaltung mittels der sogenannten "Hardware-Beschreibungs-Sprache" ***Verilog*.** Mit Hilfe von Verilog können Schaltungen auf verschiedenen Abstraktionsebenen beschrieben werden.

- Auf *Gatter-Ebene* – welche logischen Gatter werden verwendet und wie sind diese untereinander verbunden
- Auf *Register-Transfer-Level (RTL)* – die Beschreibung einer Schaltung mit Hilfe von Gleichungen
- Eine *verhaltensbasierte* Schaltungs-Beschreibung spezifiziert algorithmisch das Verhalten einer Schaltung

In dieser Vorlesung soll aus Zeitgründen nur eine kurze Einführung in die Thematik der Schaltungssimulation gegeben und nur der erste Punkt der obigen Liste, eine Beschreibung von Gattern mittels Verilog gezeigt werden.

Eine der grundlegenden Funktionen, die in Rechenwerken benötigt werden, ist die Addition von Binär-Zahlen. In den nachfolgenden Abschnitten werden die elementaren Bausteine vorgestellt, die für solche Addierer benötigt werden. Diese elementaren Bausteine sind die Grundlage, auf der bei Bedarf aufgebaut werden kann, um beliebig komplizierte arithmethische Schaltungen zu beschreiben.

Die folgende Tabelle zeigt eine Übersicht der logischen Operationen, der zugehörigen Verilog-Gatter-Funktionen und die arithmetische Spezifikation für jedes Gatter. Auf diese Tabelle wird in den nächsten Abschnitten und in Übungen zurück gegriffen:

Bezeichnung	**Logische Verknüpfung**	**Verilog-Gatter**	**Arithmetische Spezifikation**
Nicht-Gatter	c = ~a	not(c, a)	c = 1 − a
Und-Gatter	c = a & b	and(c, a, b)	c = a * b
Oder Gatter	c = a | b	or(c, a, b)	c = a + b − a * b
Exclusives-Oder-Gatter	c = a ^ b	xor(c, a, b)	c = a + b − 2 * a * b
Invertiertes Und-Gatter	c = ~(a & b)	nand(c, a, b)	c = 1 − a * b
Invertiertes Oder-Gatter	c = ~(a | b)	nor(c, a, b)	c = 1 − (a + b) + a * b
Invertiertes exclusives Oder-Gatter	c = ~(a ^ b)	xnor(c, a, b)	c = 1 − (a + b) + 2 * a * b

Die logische Verknüpfung erlaubt es, eine Schaltung auf der logischen Ebene zu beschreiben. Die Arithmetische Spezifikation ermöglicht es, die Ergebnisse der Schaltung mathematisch zu berechnen und die Korrektheit einer Schaltung zu beweisen. Die zugehörige Spezifikation in Verilog wird verwendet, um eine Schaltung zu simulieren.

Halb-Addierer

Eine Schaltung, die zwei einzelne Bits addieren kann, nennt man Halb-Addierer. Die beiden Eingänge A und B, die jeweils die Werte 0 und 1 annehmen können, werden addiert. Das Ergebnis liegt also in der Menge {0, 1, 2}. Im Binär-System sind dazu zwei Bits notwendig, die durch die beiden Ausgänge Sum und Carry repräsentiert werden.

Das Verhalten der Schaltung kann mathematisch beschrieben werden. Zwischen den Eingängen a und b und den beiden Ausgängen sum und carry besteht die folgende Beziehung:

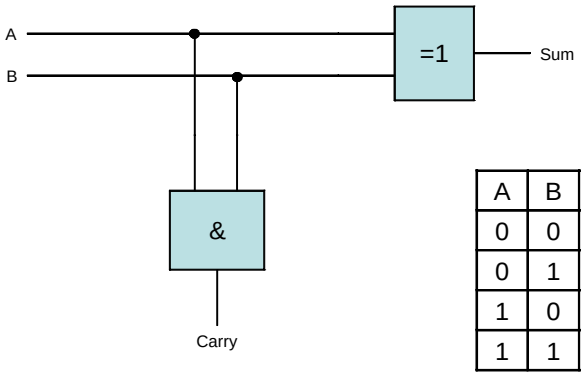
a + b = 2 * carry + sum

Diese Gleichung ist die Spezifikation eines Halb-Addierers, deren Korrektheit sich mathematisch (mit Hilfe der Tabelle arithmetischer Spezifikationen aus dem vorhergehenden Abschnitt) beweisen läßt:

2 * carry + sum		Spezifikation
=	2 * (a&b) + (a ^ b)	siehe Schaltung
=	2 * (a * b) + (a + b − 2 * a * b)	siehe Tabelle
=	~~2 * a * b~~ + a + b − ~~2 * a * b~~	
=	a + b	

Zudem soll die Schaltungsbeschreibung mittels einer Testbench getestet werden (siehe Übungen).

Die nachfolgende Abbildung zeigt die Schaltung eines Halb-Addierers, die Wahrheitstabelle, sowie die Implementierung in Verilog.



A	B	Sum	Carry
0	0	**0**	**0**
0	1	**1**	**0**
1	0	**1**	**0**
1	1	**0**	**1**

```
module half_adder( output sum, carry,
                   input  a, b);
   xor( sum, a, b);
   and( carry, a, b);
endmodule
```

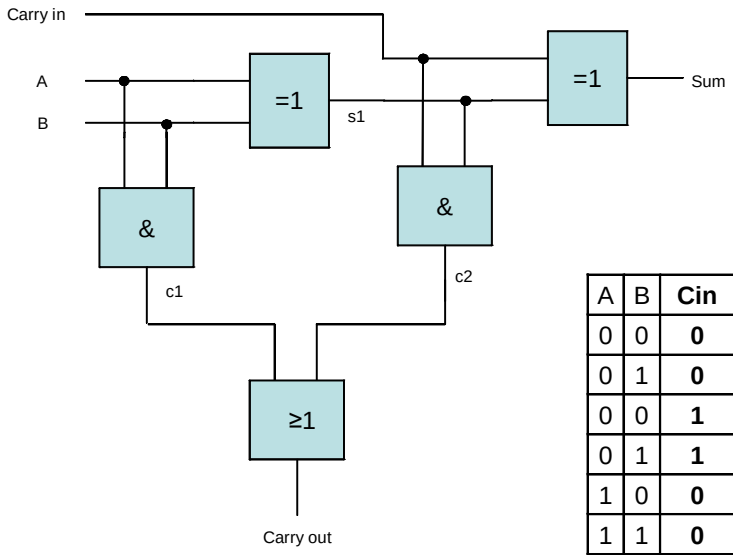
Voll-Addierer

Wenn zwei Zahlen im Zweier-System addiert werden sollen, so reicht die oben diskutierte Schaltung zur Addition zweier Bits nicht aus, denn schon beim zweiten Bit muss ein Übertrag mit berücksichtigt werden. Das führt zum Volladdierer, der intern aus zwei Halb-Addierern aufgebaut ist, wobei die beiden Carry-Signale der Halb-Addierer durch ein Oder-Gatter zusammengefasst werden.

Ein vollständiger Addierer für die Addition eines Bytes setzt sich aus 8 Voll-Addierern zusammen.

Das Verhalten eines Voll-Addierers wird wie folgt beschrieben:

$$a + b + cin = 2 * cout + sum$$



A	B	Cin	Sum	Cout
0	0	**0**	**0**	**0**
0	1	**0**	**1**	**0**
0	0	**1**	**1**	**0**
0	1	**1**	**0**	**1**
1	0	**0**	**1**	**0**
1	1	**0**	**0**	**1**
1	0	**1**	**0**	**1**
1	1	**1**	**1**	**1**

3.6 Weiterführende Informationen

Die in den vorhergehenden Abschnitten eingeführten Grundlagen können im Rahmen dieser Vorlesung und den ergänzenden Übungen nur kurz angerissen werden. Für weiterführende Informationen insbesondere zu den Kernthemen sind folgende Bücher zu empfehlen.

1. "Structured Computer Organization" von Andrew S. Tanenbaum,
   5. Auflage, erschienen 2005 bei Prentice Hall.
2. "Computer Organization & Design" von John L. Hennessy und David A. Patterson,
   3. Auflage, erschienen 2004 bei Morgan Kaufmann.
3. "Computer-Architektur: Modellierung, Entwicklung und Verifikation mit Verilog" von Karl Stroetmann, erschienen 2007 im Oldenbourg Wissenschaftsverlag.

4 Der Mikroprozessor 8086

4.1 Einführung

Der Mikroprozessor 8086 war der erste der Intel 80x86 Serie und wurde als Nachfolger des 8-Bit Prozessors 8085 von Intel etwa 1978 auf den Markt gebracht. Der 8086 ist ein 16-Bit Prozessor mit 16-Bit Bus- und Registerbreite. Es stehen allerdings 20 Adressleitungen zur Verfügung und somit kann 1 MB (= 2^{20} Bytes) Speicher adressiert werden.

Die Grundkonzepte des 8086, wie z.B. seine Registerstruktur, seine Speicheradressierungsmöglichkeiten und sein Interrupt-Konzept, sind bis zu den heutigen Prozessoren wieder zu erkennen, inzwischen natürlich deutlich erweitert, aber "kompatibel". Diese Grundkonzepte werden anhand des 8086 vorgestellt, weil dessen Strukturen noch überschaubar sind und im Rahmen der Vorlesung behandelt werden können.

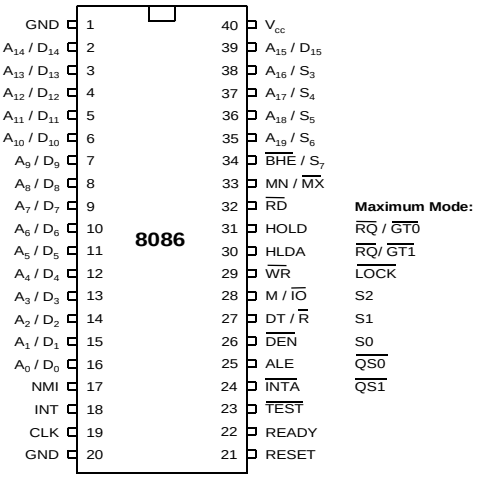
In diesem Kapitel werden nach einem kurzen Hardware-Überblick die die Programmierung betreffenden Funktionen des 8086 vorgestellt. Dazu gehören die Register und deren Verwendungsmöglichkeiten, die Implementierung des Stacks, das Interrupt-Konzept, der Aufbau eines Maschinenbefehls und natürlich der Befehlssatz des Prozessors. Als weiterführendes, von diesem Prozessor unabhängiges Element wird auf die Parameterübergabe beim Aufruf von Unterprogrammen eingegangen. Eine Einführung in die Assemblerprogrammierung mit FASM schließt das Kapitel ab.

Die folgende Liste gibt einen Überblick über den 8086 und die funktionellen Erweiterungen, die gegenüber dem 8085 implementiert wurden:

- 1 MByte Adressraum
- Speichersegmentierung
- 14 interne 16-Bit Register
- "Minimum Mode" und "Maximum Mode" zur Unterstützung einfacher Einprozessor- und aufwendigerer Mehrprozessorsysteme
- "Pipelining" durch getrennte EU (Execution Unit) und BIU (Bus Interface Unit)
- 24 verschiedene Speicher-Adressierungsmöglichkeiten (zur Unterstützung höherer Programmiersprachen)
- Multiplikation und Division von Binär- und (Festkomma-)Dezimalzahlen mit und ohne Vorzeichen
- Verschieben, Durchsuchen und Vergleichen von Zeichenketten (Strings) bis zu 64k-Byte Länge
- Vergleichsbefehle, die nur Flags setzen, jedoch keine Daten verändern
- Interrupts durch Software auslösbar
- Instruktionen zur Unterstützung von Multiprozessorsystemen

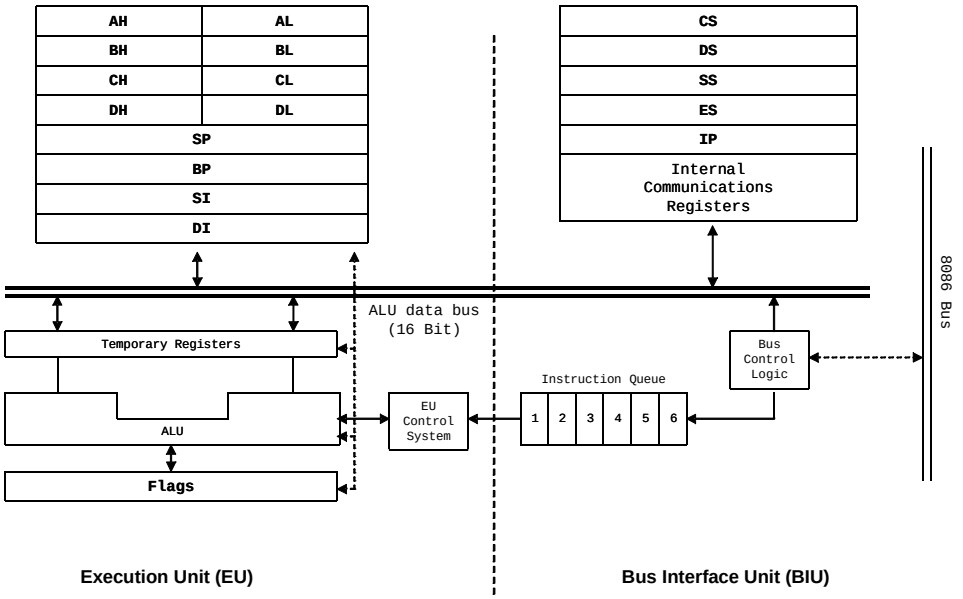
4.2 Hardware Überblick

Der folgende Abschnitt gibt einen kurzen Überblick über ein paar Hardwaremerkmale des 8086. Die Beschreibungen (wie auch die nachfolgenden Abschnitte) gelten auch für den kurz nach dem 8086 auf den Markt gebrachten 8088. Der 8088 ist intern nahezu ein 8086, tätigt extern jedoch nur 8-Bit Datenzugriffe und war dazu gedacht, kostengünstigere Systeme zu ermöglichen. Das folgende Bild zeigt das Pin-Layout des 8086:



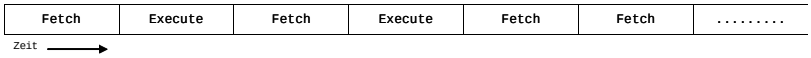
Der 8088 unterscheidet sich im wesentlichen nur dahingehend vom 8086, dass nur die ersten 8 Adressleitungen mit Datenleitungen gemultiplext sind.

Auch in dem gezeigten Blockbild des Prozessors gibt es nur einen kleinen Unterschied zwischen 8086 und 8088. Lediglich die Instruction-Queue, in der Befehle vorgehalten werden, ist beim 8088 nur 4, statt 6 Byte lang. Alle Register sowie der interne Bus sind bei beiden Prozessoren 16 Bit breit.

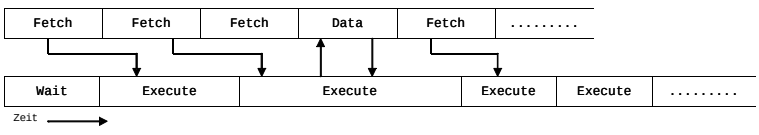


Die Aufteilung des Prozessors in BIU und EU ermöglicht eine Parallelisierung von Buszugriffen und Befehlsausführung. Während bei den bis damals gängigen Prozessoren Buszugriffe und Befehlsausführung immer nacheinander stattfanden, konnte der 8086 diese Operationen überlappen.

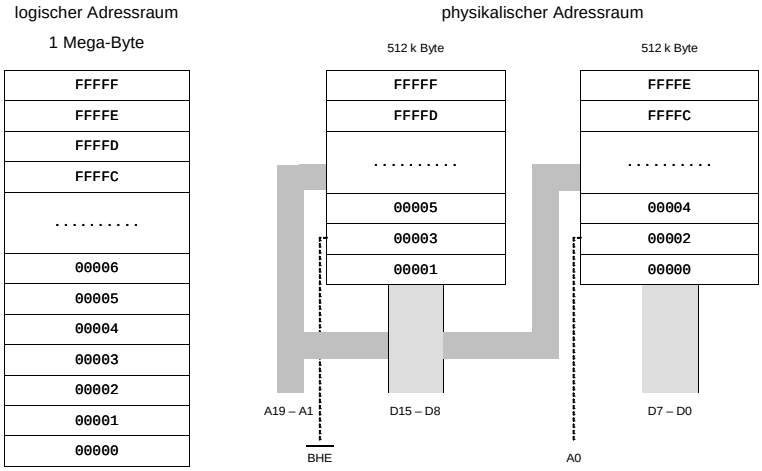
Herkömmlicher Prozessor:



8086 mit BIU und EU:



Die folgende Abbildung zeigt die physikalische Adressierung der Speicherbausteine in einem 8086- System. Darin wird deutlich, dass der Speicher zwar Byte-orientiert ist, Zugriffe jedoch mit 16-Bit- Breite erfolgen und deshalb auf Daten, die auf 2-Byte-Grenzen liegen, in einem Buszyklus erfolgen können. Für Zugriffe auf Daten-Worte, die nicht auf 2-Byte-Grenzen liegen, sind zwei Buszyklen nötig.



4.3 Allgemeine Daten- und Zeiger-Register

Der 8086 stellt 4 allgemeine Datenregister (AX, BX, CX, DX) und 4 Zeigerregister (SI, DI, BP, SP) zur Verfügung. Die vier allgemeinen, 16- Bit breiten Datenregister AX-DX können mit AL, AH, ..., DL, DH auch 8-bit-weise angesprochen werden. Alle 8 Daten- und Zeiger-Register lassen sich theoretisch gleichermaßen für Datenzwischenspeicherung verwenden.

Praktisch haben aber alle Register andere Eigenschaften. AX z.B. hat eine Sonderstellung für arithmetischen Berechnungen und wird deshalb auch bei Datentransferbefehlen bevorzugt (kürzere Befehle bei Verwendung des AX; AX wird auch Akkumulator genannt). CX wird bevorzugt als Schleifenzähler verwendet.

In der folgenden Skizze sind sowohl die Register schematisch dargestellt, als auch ihre bevorzugte, spezielle Verwendung in der Tabelle aufgelistet:

Datenregister

Accu	**AH**	**AL**	**AX**
Base	**BH**	**BL**	**BX**
Count	**CH**	**CL**	**CX**
Destination	**DH**	**DL**	**DX**

Instruction Pointer

IP

Zeigerregister

Stack Pointer	**SP**
Base Pointer	**BP**
Source Index	**SI**
Destination Index	**DI**

Register	Spezielle Verwendung
AX	Wort-Multiplikation, -Division, I/O Daten
AH, AL	Byte-Multiplikation, -Division
BX	Memory Pointer, Translate-Funktion (XLAT)
CX	Schleifenzähler, z.B. für String-Operationen
CL	variable Shift/Rotate-Operationen
DX	Wort-Multiplikation, -Division, Adressen für indirekte I/Os
SP, BP	Zeiger für Stack-Operationen
SI, DI	Zeiger für String-Operationen

4.4 Flag Register

Die Flags teilen sich auf in **Status-Flags**, die das Ergebnis einer Operation widerspiegeln, und in **Kontroll-Flags**, die den Betrieb des Prozessors beeinflussen. Die unteren 8 Bits des Flag-Registers decken sich mit den Flags des 8085.

X	X	X	X	**OF**	**DF**	**IF**	**TF**	**SF**	**ZF**	X	**AF**	X	**PF**	X	**CF**

Kontroll-Flags

TF	Trap Flag (Einzelschritt: löst speziellen Interrupt aus)
IF	Interrupt Enable Flag (maskierbare Interrupts 0=sperren, 1=freigeben)
DF	Direction Flag (Richtung von String-Operationen: 0=increment, 1=decrement)

Status-Flags

CF	Carry Flag (Überlauf im höchsten Bit einer Operation)
PF	Parity Flag (Parität des unteren Bytes eines Ergebnises)
AF	Auxiliary Flag (Überlauf in den untersten 4 Bit einer Operation)
ZF	Zero Flag (Ergebnis ist Null)
SF	Sign Flag (Vorzeichen des Ergebnisses (0=positiv))
OF	Overflow Flag (Überlauf bei Operationen mit Vorzeichen)

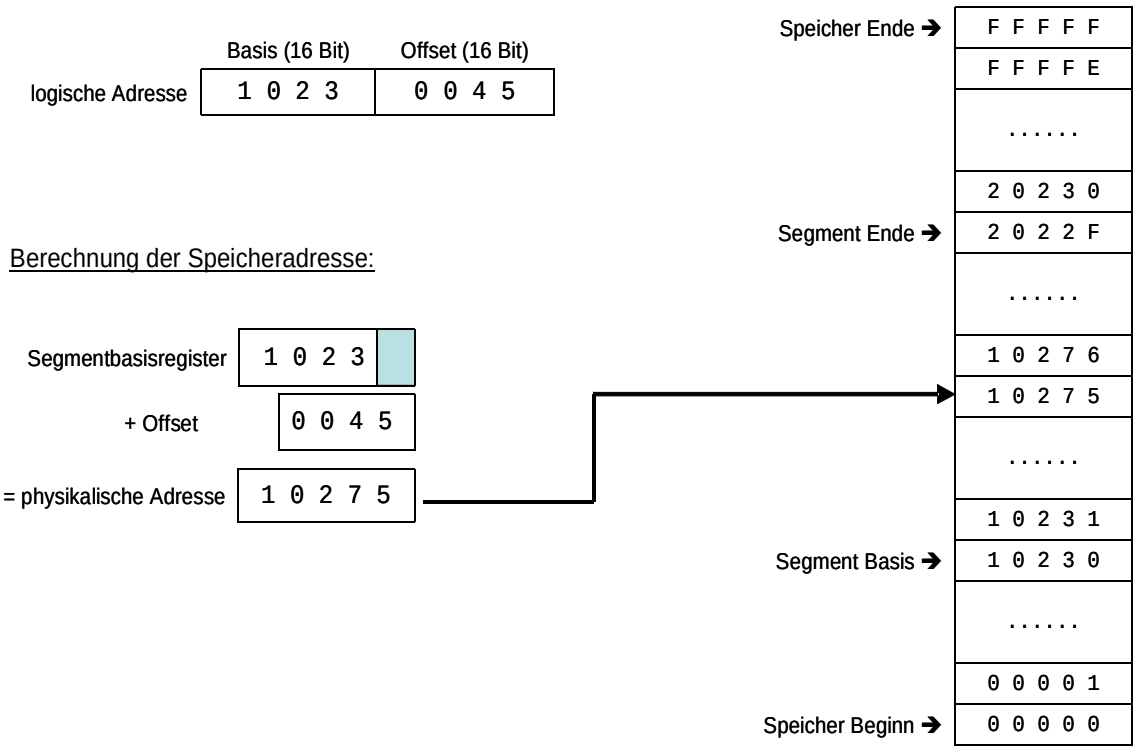
Die oberen 8 Bits sind neu. Außer den Kontroll-Flags wurde das Overflow-Flag (OF) neu eingeführt, das nur bei vorzeichenbehafteten Operationen beeinflusst wird.

4.5 Speichersegmentierung und Segmentregister

Der 1-MByte Adressraum des 8086 ist in Segmente aufgeteilt. Dies war notwendig, da mit 16-Bit Registern nur maximal 64 KByte große Datenblöcke adressierbar sind und Intel zum damaligen Zeitpunkt aus Kostengründen nicht 2 vollständige Register und damit 32 Pins zur Adressierung zur Verfügung stellen wollte. Also wurde mit Hilfe der Segment-Register (auch Segment-Basis-Register genannt) eine Adresserweiterung auf 20 Bit implementiert, womit man dann einen Adressraum von 1 MByte erreichen konnte (1 MByte = 2^{20} Byte).

Erzeugung einer physikalischen Adresse

Die Segmentierung des Speichers beim 8086 und die Berechnung der physikalischen 20-Bit Adressen die folgende Skizze. Ausgangspunkt ist dabei die aus Segmentbasis und Offset bestehende **logische Adresse.**



Eines der vier Segment-Register CS, SS, DS und ES enthält jeweils die Basisadresse eines Segments. Daten werden mit Hilfe des DS, SS oder ES Registers adressiert. Code wird über das CS Register zusammen mit dem Befehls-Zeiger (Instruction Pointer IP) gelesen.

	Segmentregister
Code Segment	**CS**
Data Segment	**DS**
Stack Segment	**SS**
Extra Segment	**ES**

Die folgende Tabelle zeigt die Verwendung der Segment-Register bei Speicherzugriffen:

Speicherzugriff	***Normale Segment Basis***	***Alternative Segment Basis***	***Offset***
Befehlszugriff	CS	keine	IP
Stack Operation	SS	keine	SP
Variable	DS	CS, ES, SS	effektive Adresse
String lesen	DS	CS, ES, SS	SI
String schreiben	ES	keine	DI
BP als Basis	SS	CS, DS, ES	effektive Adresse
BX als Basis	DS	CS, SS, ES	effektive Adresse
I/O Operation	keine	keine	direkt oder DX

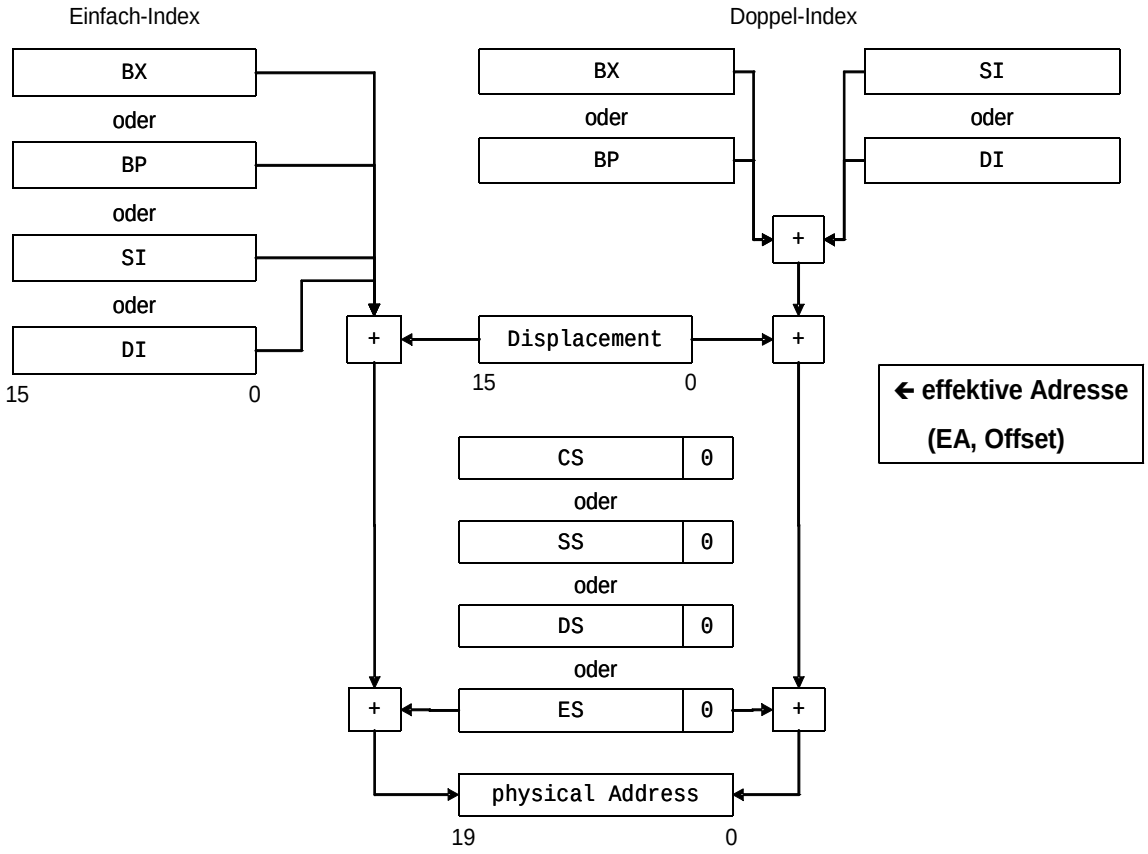
Laden von Daten

Die verschiedenen Möglichkeiten, Daten zu laden und Speicher zu adressieren, sind eine wesentliche Komponente bei der (Assembler-) Programmierung eines Prozessors. Unter Speicheradressierungsmöglichkeiten versteht man z.B. die verschiedenen Möglichkeiten, wie Zeigerregister und direkte Adressen miteinander verknüpft werden können, um eine Speicheradresse zu bilden.

Eine Charakteristik eines CISC-Prozessors, zu denen ja auch der 8086 gehört, ist, dass relativ viele verschiedene Arten der Adressierung bei Datenzugriffen möglich sind. Die beim 8086 zur Verfügung stehenden Adressierungsarten sind in der folgenden Tabelle aufgelistet. Ein jeweils mit angegebenes Beispiel eines MOV-Befehls demonstriert eine Verwendungsmöglichkeit der jeweiligen Adressierungsart.

Adressierung	***Daten befinden sich***	***Beispiel***
immediate	im Befehl	MOV AX, 5
register	in einem Register	MOV AX, SI
direct	im Speicher (Adresse im Befehl)	MOV AX, [var]
register indirect	im Speicher (Adresse in einem Register enthalten)	(LEA SI, [var]) MOV AX, [SI]
indexed or based with displacement	im Speicher (Adresse berechnet aus der Summe eines Registerinhalts und im Befehl enthaltenen Daten)	MOV AX, [SI+5] MOV AX, [BP+5]
based and indexed with displacement	im Speicher (Adresse ist Summe aus zwei Registern und im Befehl enthaltenen Daten)	MOV AX, [BP+SI+5]

Die nachfolgende Skizze stellt graphisch dar, welche Registerkombinationen beim 8086 zur Bildung einer Speicheradresse bei indizierter oder indirekter Adressierung möglich sind. In diese Darstellung sind auch die Segmentbasisregister mit aufgenommen worden.



4.6 Befehlssatz

Der 8086 stellt einen relativ umfangreichen Befehlssatz zur Verfügung, was ihn (zusammen mit anderen Merkmalen) als klassischen CISC-Prozessor kennzeichnet. In den folgenden Abschnitten wird dieser Befehlssatz vorgestellt und einige Befehle näher erläutert. Der komplette Befehlssatz ist in "8086 Befehlssatz" auf Seite 71 aufgelistet.

Datentransferbefehle

Unter Datentransferbefehlen versteht man all die Befehle, die Daten zwischen Registern und dem Speicher oder zwischen Registern austauschen. Beispiele für Datentransferbefehle sind MOV, PUSH, usw..

Beim 8086 gibt es für den MOV-Befehl nur den mnemonischen Code 'MOV' für alle Ausprägungen des Befehls, d.h. Zugriffe auf 'immediate data', Speicher oder Register werden nur durch die Operanden unterschieden. Der Assembler bestimmt die verschiedenen Maschinenbefehle selbständig. Auch die Entscheidung, ob ein 16-Bit oder ein 8-Bit Datenzugriff erfolgen soll, wird ausschließlich durch die Operanden festgelegt.

Außer dem MOV-Befehl stellt der 8086 als weitere Datentransferbefehle natürlich die Befehle 'PUSH' und 'POP' für Stack-Operationen zur Verfügung. Zusätzlich gibt es die Befehle 'PUSHF' und 'POPF', um die Flags auf dem Stack zu speichern bzw. wieder zurückzuholen.

Ganz besonders seien die Befehle **LEA**, **LDS**, und **LES** hervorgehoben. Mit dem Befehl LEA (Load Effective Address) wird (im Gegensatz zu MOV) nicht der Inhalt einer Speicherstelle,

sondern deren Adresse geladen. Die Pointer-Befehle LDS und LES laden ein Doppelwort aus dem Speicher in DS (bzw. ES) plus ein weiteres angegebenes Register (siehe auch die Erklärung zu LDS und LES in den Tabellen in Kapitel 11, "8086 Befehlssatz" auf Seite 71).

```
ptr    DD 0B8001234h       ; Pointer im Datensegment
...
LDS    SI,[ptr]            ; Pointer-Befehl: Lade DS+SI
```

Darüber hinaus seien noch zwei spezielle Datentransferbefehle erwähnt, die Befehle 'XCHG' und 'XLAT'. Beim XCHG-Befehl werden Quell- und Ziel-Operand gegeneinander ausgetauscht. Beim XLAT-Befehl wird ein 8-Bit-Wert aus einer Tabelle im Speicher in das AL-Register übertragen. Die Basisadresse der Tabelle im Speicher wird durch das BX-Register angegeben. Die Speicheradresse wird gebildet, indem der vor dem Befehl im AL-Register stehende Wert zum BX-Register addiert wird.

Programmtransferbefehle

Zu den Programmtransferbefehlen zählen die Sprungbefehle sowie die Befehle 'CALL' und 'RET'. Die Sprungbefehle gliedern sich in unbedingte und bedingte Sprünge. Eine unbedingte Sprunganweisung ist der folgende Befehl:

```
JMP  Sprungadresse
```

Außerdem stellt der 8086 eine ganze Reihe von bedingten Sprungbefehlen zur Verfügung, bei denen die Ausführung des Sprunges vom Zustand von Flags abhängt. Beispiele für bedingte Sprungbefehle sind:

```
JNZ  Sprungadresse    ; Jump if Not Zero
JBE  Sprungadresse    ; Jump if Below or Equal
JO   Sprungadresse    ; Jump if Overflow
JNE  Sprungadresse
...
```

Als besonderer, bedingter Sprungbefehl sei noch der Befehl 'JCXZ' erwähnt. Bei dieser Anweisung wird der Sprung dann ausgeführt, wenn das CX-Register Null enthält.

Der LOOP-Befehl

Der 8086 stellt des weiteren LOOP-Befehle zur Verfügung, die die Programmierung von Schleifen unterstützen. Die folgende Schleife

```
Schleife:   ...
            DEC  CX           ; Scleifenzähler
            JNZ  Schleife     ; zurück an Schleifenanfang
```

sieht mit einem LOOP-Befehl folgendermaßen aus:

```
Schleife:   ...
            LOOP Schleife
```

D.h., durch den LOOP-Befehl wird das CX Register dekrementiert und ein Sprung ausgeführt, solange CX größer 0 ist. Der Vorteil des LOOP-Befehls ist dabei nicht nur, dass die Ausführung schneller ist, sondern auch, dass das ZERO-Flag (ZF) nicht beeinflusst wird, was u.U. ein Programm vereinfacht.

Input und Output

Für I/O Operationen stehen die zwei Befehle **IN** und **OUT** zur Verfügung. Die Adressierung erfolgt dabei entweder direkt (solange die Adresse kleiner als 256 (0FFh) ist) oder indirekt über das DX-Register (gesamter I/O Bereich von 0000h-FFFFh).

```
Beispiele:     IN   AX, 080h              MOV  DX, 035Ah
               ...                        OUT  DX, AX
                                          ...
```

Stringbefehle

Eine Besonderheit im Befehlssatz des 8086 stellen die String-Befehle dar. String-Befehle werden verwendet, um ganze Speicherbereiche (Strings) von maximal 64k-Byte Länge zu beeinflussen. Die Funktionsweise von String-Befehlen soll an folgendem Beispiel verdeutlicht werden. Der folgende (herkömmlich programmierte) Programmteil kopiert einen Datenblock der Länge 100 Bytes von der Adresse 1800h nach 1A00h.

```
               MOV  AX, 2000h
               MOV  DS, AX
               MOV  ES, AX
               MOV  CX, 100
               MOV  SI, 01800h
               MOV  DI, 01A00h
Schleife:      MOV  AL, [SI]
               MOV  [ES:DI], AL
               INC  SI
               INC  DI
               DEC  CX
               JNZ  Schleife
```

Dieser Programmteil soll nun nach und nach optimiert werden. Ein LOOP-Befehl könnte die Iteration bereits beschleunigen. Eine wesentlich effektivere Ausführung eines solchen Programmteils wird jedoch mit einem **String**-Befehl erreicht. Mit dem String- Befehl MOVSB (***MOV S***tring Byte weise) sieht das Programm folgendermaßen aus:

```
               ...                                  CLD        ; Richtungsflag
Schleife:      MOV  AL, [SI]          Schleife:     MOVSB      ;    (siehe Text)
               MOV  [ES:DI], AL  --->
               INC  SI
               INC  DI
               LOOP Schleife                        LOOP Schleife
```

Der String-Befehl MOVSB beinhaltet demzufolge das Verschieben der Daten sowie die Anpassung der beiden verwendeten Zeigerregister. Dabei ***muss*** SI (***S***ource-***I***ndex) als Quell-Zeiger und DI (***D***estination- ***I***ndex) als Ziel-Zeiger verwendet werden. Außerdem wird in Verbindung mit DI immer das ES-Register als Segment-Basis benutzt. Hinweis: Die Verschiebung erfolgt direkt, d.h. AX wird nicht beeinflusst.

Strings können rückwärts oder vorwärts verschoben werden (nötig bei überlappenden Speicherbereichen). Die Richtung (und damit die Anpassung der Register SI und DI) wird durch das **Direction Flag** angezeigt, das mit den Befehlen **STD** und **CLD** gesetzt bzw. rückgesetzt wird. Ist das direction flag nicht gesetzt, erfolgt die Verschiebung vorwärts, d.h. SI und DI werden nach jeder Operation inkrementiert.

Darüber hinaus gibt es auch String-Befehle, die die Daten Wortweise behandeln, z.B. MOVSW (***MOV S***tring ***W***ortweise). Mit jeder Ausführung dieser Befehle werden die Register SI und DI jeweils um 2 inkrementiert bzw. dekrementiert.

Außer MOV gibt es noch die folgenden String-Befehle (jeweils für Byte- und Wort-Ausführung):

- CMPSB/CMPSW (***CMP S***tring, vergleicht zwei Strings)

- SCASB/SCASW (durchsucht einen String nach einem bestimmten Wort, das im ACCU steht)
- STOSB/STOSW (speichert Byte/Wort aus ACCU im Speicher ab)
- LODSB/LODSW (lädt Byte/Wort in ACCU)

Die letzten beiden Befehle sind z.B. sinnvoll in Verbindung mit einem Peripheriebaustein (z.B. UART), um die ausgehenden Daten zu laden bzw. die eingehenden Daten abzuspeichern.

Die letzte Optimierung wird jetzt noch durch den **Repetition-Prefix** (REP) erreicht, der bewirkt, dass neben der Ausführung des String-Befehls zusätzlich (im gleichen Befehl) das Register CX um 1 verringert wird. Der Befehl wird solange ausgeführt, solange CX größer Null ist.

Das obige Programm könnte dann folgendermaßen aussehen:

```
        MOV  CX, 100
        MOV  SI, 01800h
        MOV  DI, 01A00h
        CLD
REP     MOVSB
```

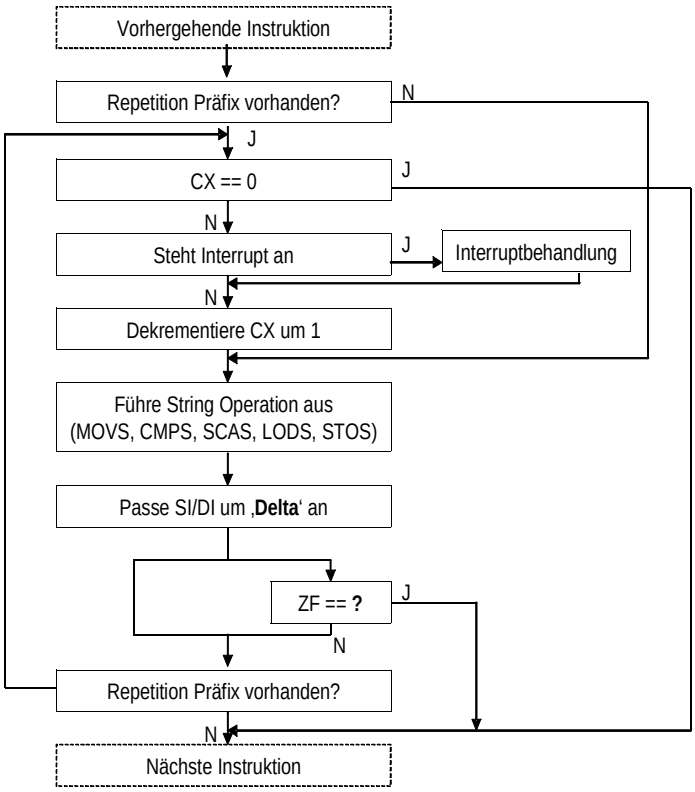
In Verbindung mit CMPS und SCAS gibt es auch den Repetition-Prefix REPZ, REPE, REPNZ und REPNE, wodurch nach jedem Einzelschritt zusätzlich das Zero-Flag abgefragt werden kann. Für LODS ist die Verwendung eines Repetition-Prefix nicht sinnvoll.

Zusammenfassung String-Befehle:

MOVS	Verschiebe (kopiere) Byte- oder Wort-String
CMPS	Vergleiche Byte- oder Wort-String
SCAS	Durchsuche Byte- oder Wort-String
LODS	Lade Byte oder Wort in ACCU
STOS	Speichere Byte oder Wort

SI	Index für Quell-String
DI	Index für Ziel-String
CX	Schleifenzähler
AX,AL	für SCAS, STOS und LODS
DF	Richtung der Korrektur von SI und DI (0=aufsteigend, 1=absteigend)
ZF	gesetzt bei CMPS und SCAS

Die interne Abarbeitung von String-Befehlen ist in der folgenden Skizze dargestellt.



String	DF	**Delta**
Byte	0	1
Byte	1	-1
Wort	0	2
Wort	1	-2

Präfix	**?**
REPE	0
REPZ	0
REPNE	1
REPNZ	1

4.7 Aufbau eines Maschinenbefehls

Befehle des 8086 bestehen aus einem bis sechs Bytes, mit Präfixen sogar bis zu 8 Bytes.

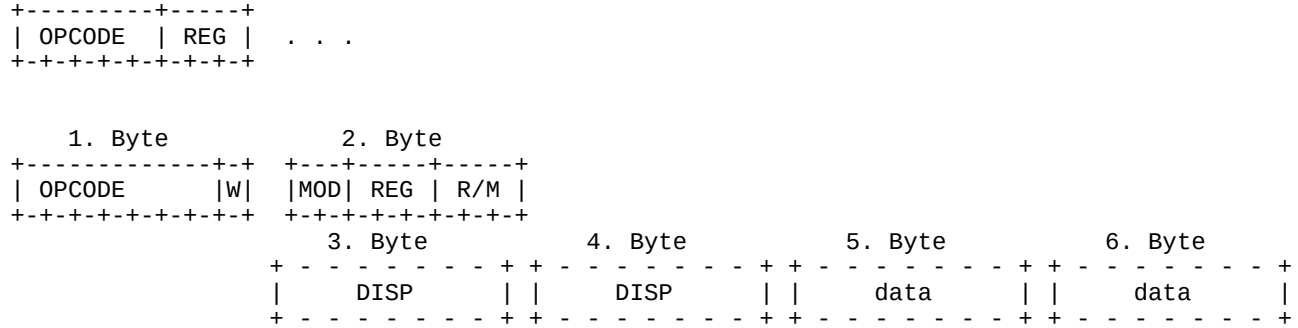
Beispiele für 1-Byte-Befehle sind:

```
INC AX        ; increment AX register
CLC           ; clear carry
MOVSW         ; move string wordwise
```

6-Byte-Befehle sind Befehle, in denen bei der Speicheradressierung ein Displacement verwendet wird und dann noch "immediate data" hinzukommen, z.B.:

```
CMP [BX]+ALPHA, 3456h      ; compare mem,data
```

Der interne Aufbau eines Befehls kann folgendermaßen dargestellt werden:



Das Bit W gibt an, ob es sich um einen Wortbefehl (W=1) oder einen Bytebefehl (W=0) handelt. Die Kombination von **REG** und **W** definiert ein Register (für W=1 ein ganzes 16-Bit-Register, für

W=0 eines der einzeln ansprechbaren 8-Bit-Register. Der Parameter **MOD** bestimmt die Art des Befehlsaufbaus. Die Kombination von **R/M** und **MOD** definiert eine Adressierungsart.

Die nachfolgenden Tabellen beschreiben diese Parameter.

MOD	2. Byte	3. Byte	4. Byte
00	mod reg r/m	-	-
00*	mod reg 110	addr(LSB)	addr(MSB)
01	mod reg r/m	disp 8	-
10	mod reg r/m	disp 16 (LSB)	disp 16 (MSB)
11	mod reg reg	-	-

REG	W = 1	W = 0
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

R/M	effektive Adresse
000	[BX] + [SI] + DISP
001	[BX] + [DI] + DISP
010	[BP] + [SI] + DISP
011	[BP] + [DI] + DISP
100	[SI] + DISP
101	[DI] + DISP
110	[BP] + DISP
111	[BX] + DISP

Nach dem darin gezeigten Regelwerk wäre die direkte Adressierung nicht möglich, 'r/m' gibt immer ein Register an. Aus diesem Grund wurde eine Ausnahme implementiert. Bei der Kombination 'mod=00' und 'r/m=110' wird die Adresse nicht im BP-Register angenommen, sondern die im Befehl angegebene Adresse zur direkten Adressierung verwendet. Dies hat zwar zur Folge, dass für [BP] immer ein Displacement mitgegeben werden muss, auch wenn dies 0000h ist. Bei einer Adressierung mit BP wird jedoch fast immer sowieso ein Displacement benötigt, da über BP normalerweise der Stack adressiert wird.

Bei mod=01 wird das 8-Bit Displacement "sign-extended" auf 16-Bit.

4.8 Stack des 8086

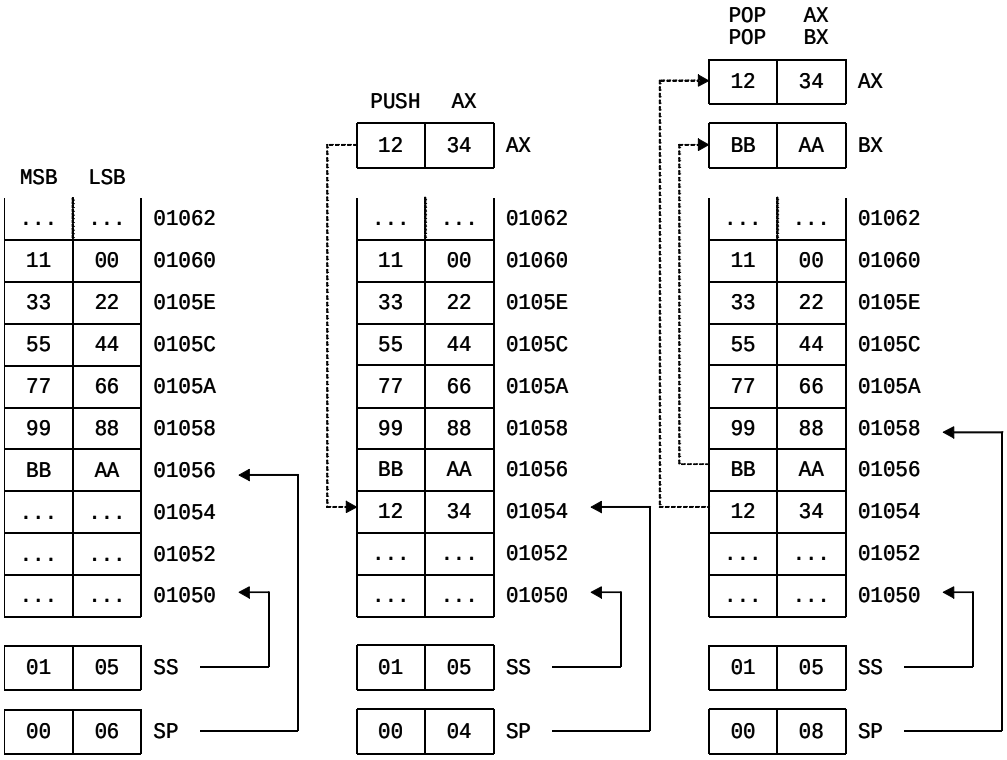
Der Stack ist ein Segment im Speicher, das mittels des Basisregisters SS adressiert wird. Der Stack des 8086 beginnt bei dem Offset FFFFh ("Bottom of Stack": SS:FFFFh) und reicht bis zum Offset 0000h des Segments. Als Top Of Stack (TOS) wird der aktuelle Zeiger SS:SP bezeichnet.

Beim Schreiben auf den Stack (PUSH) wird der Stackpointer SP immer zunächst dekrementiert, dann wird ein Datum auf den Stack geschrieben. Beim Lesen (POP) wird der Stackpointer **nach** dem Lesevorgang inkrementiert.

Um die Funktionsweise des 8086-Stacks zu verdeutlichen, sollen beispielsweise folgende Operationen nacheinander ausgeführt werden:

```
PUSH    AX
POP     AX
POP     BX
```

Während dieser Operationen verändert sich der Stack wie folgt:



4.9 Parameterübergabe an ein Unterprogramm

Bei älteren Compilern wurden alle Variablen, die von einem Programm an ein Unterprogramm übergeben wurden, auf dem Stack abgelegt. Moderne Compiler nutzen zunächst freie Register und benutzen den Stack erst, wenn mehr Variablen übergeben werden müssen, als Register verfügbar sind. Die letztere Methode soll hier jedoch nicht betrachtet werden.

Beim Aufruf eines Unterprogramms werden die nachfolgend beschriebenen Operationen auf dem Stack ausgeführt.

Im Hauptprogramm:

1. Ablegen von Übergabevariablen oder -parametern
2. Call: implizit Speichern der Rücksprungadresse (near call: IP, far call: CS:IP)

Im Unterprogramm:

3. Sichern des Basepointers (BP)
4. Reservieren eines Bereichs für lokale Variablen #(falls benötigt)
5. Sichern weiterer Register (falls nötig)
6. vor Verlassen des Unterprogramms 1.-3. rückgängig machen
7. Return (near call: RET, far call: RETF)

Im Hauptprogramm:

8. Übergabevariablen vom Stack aufräumen

Die folgenden Programmauszüge verdeutlichen dies.

Hauptprogramm

```
segment CSEG1
MAIN:   ...
        PUSH var1           ; Punkt 1 der obigen Liste
        PUSH var2
        CALL CSEG2:FUNC     ; "far" call von FUNC
        ...
```

Unterprogramm "FUNC"

```
segment CSEG2               ; Far call (Inter-Segment)
FUNC:   PUSH  BP            ; Punkt 3 der obigen Liste
        MOV   BP, SP
        SUB   SP, 6         ; 3 x 16-bit lokale Variablen
                            ; Punkt 4. der obigen Liste
        PUSH  CX
        PUSH  BX
        PUSHF

        MOV   CX, [BP+8]    ; Zugriff auf var1
        MOV   BX, [BP+6]    ; Zugriff auf var2
        ...
        POPF
        POP   BX
        POP   CX
        MOV   SP, BP        ; Stackpointer wiederherstellen
        POP   BP
        RETF  4             ; bei PASCAL-Calling-Convention
```

Der Befehl "RETF 4" am Ende des Unterprogramms beinhaltet den Rücksprung ins Hauptprogramm ("far" return)und das "Aufräumen" von 4 Bytes auf dem Stack (Übergabevariablen; das Hauptprogramm hat zwei 16-Bit Werte vor dem Aufruf des Unterprogramms auf den Stack geladen).

Dieses "Aufräumen" des Stacks vom Unterprogramm ist natürlich nur möglich, wenn die Anzahl der Übergabeparameter zum Compile-Zeitpunkt bekannt ist. Es gibt zwei Konventionen, wie Unterprogramme aufgerufen werden: Die PASCAL- und die C-Calling-Convention. Bei der

PASCAL-Konvention ist die Anzahl der Übergabeparameter fest und diese werden von links beginnend auf dem Stack abgelegt, d.h. der im PASCAL-Programm am weitesten links stehende Parameter wird zuerst gePUSHt.

Bei C werden die Übergabeparameter von rechts beginnend auf den Stack gePUSHt. Dies wird aus dem Grund gemacht, dass nach rechts Übergabeparameter weggelassen werden können, der Übergabe-Parameterstring also nicht vollständig sein muss. Aus diesem Grund wiederum kann bei der C-Calling-Convention nicht das Unterprogramm am Ende den Stack aufräumen, da dem Unterprogramm zum Compile-Zeitpunkt nicht bekannt ist, wie viele Parameter sich tatsächlich auf dem Stack befinden. In diesem Fall kann nur das Hauptprogramm den Stack wieder korrigieren. Im obigen Beispiel würde dies durch den folgenden Befehl im Hauptprogramm nach dem CALL-Befehl erreicht werden:

```
ADD   SP,4      ; korrigiert den Stackpointer um 4 Bytes
                ; d.h. macht 2 PUSHs rückgängig
```

Das Unterprogramm würde in diesem Fall mit einem normalen RET-Befehl (ohne Parameter) beendet werden.

4.10 8086 Beispielprogramm

Mit den bisher erworbenen Kenntnissen ist es jetzt möglich, den in der Einführung betrachteten Programmausschnitt genauer zu betrachten. Diese Schleife wird vom MSVC 1.5 ungefähr folgendermaßen übersetzt:

```
1  void test( void) {
                                    push  bp
                                    mov   bp,sp
                                    sub   sp,24
                                    push  bx
                                    push  ax

2      int i,                       ; bp-2
           ende,                    ; bp-4
           var[10];                 ; bp-24

3      for (i=0; i<ende; i++) {
                                    mov   word [bp-2], 0
                                    jmp   lbl_chkl
                               lbl_loop:
4         if (var[i] > 0) {
                                    mov   ax,word [bp-2]
                                    shl   ax,1
                                    lea   bx,word [bp-24]
                                    add   bx,ax
                                    cmp   word [ss:bx],0
                                    jle   lbl_else
5           var[i]++;
                                    mov   ax,word [bp-2]
                                    shl   ax,1
                                    lea   bx,word [bp-24]
                                    add   bx,ax
                                    add   word [ss:bx],1
                                    jmp   lbl_endl
6        } else {
                               lbl_else:
7           var[i]--;
                                    mov   ax,word [bp-2]
                                    shl   ax,1
                                    lea   bx,word [bp-24]
                                    add   bx,ax
                                    sub   word [ss:bx],1
8        } /* endif */
                               lbl_endl:
9     } /* endfor */
                                    add   word [bp-2],1
                               lbl_chkl:
                                    mov   ax,word [bp-4]
                                    cmp   word [bp-2],ax
                                    jl    lbl_loop
10 }
                                    pop   ax
                                    pop   bx
                                    mov   sp,bp
                                    pop   bp
                                    ret
```

Dieses Programm kann stark optimiert werden. Insbesondere das Laden und Speichern der Array-Elemente braucht nicht im if- und im else-Zweig erfolgen:

```
3      for (i=0; i<ende; i++) {
                                    mov   word [bp-2], 0
                                    jmp   lbl_chkl
                               lbl_loop:
                                    mov   ax,word [bp-2]
                                    shl   ax,1
                                    lea   bx,word [bp-24]
                                    add   bx,ax
                                    mov   ax,word [ss:bx]
```

```
4         if (var[i] > 0) {
                                      cmp   ax,0
                                      jle   lbl_else
5           var[i]++;
                                      inc   ax
                                      jmp   lbl_endl
6        } else {
                                lbl_else:
7            var[i]--;
                                      dec   ax
8        } /* endif */
                                lbl_endl:
9     } /* endfor */
                                      mov   word [ss:bx],ax
                                      add   word [bp-2],1
                                lbl_chkl:
                                      mov   ax,word [bp-4]
                                      cmp   word [bp-2],ax
                                      jl    lbl_loop
```

Werden zusätzlich die Maschineninstruktionen mit dargestellt, so sieht die nicht optimierte Version des Beispielprogramms folgendermaßen aus:

```
1  void test( void) {
                              0000  55                    push  bp
                              0001  89 E5                 mov   bp,sp
                              0003  83 EC 18              sub   sp,24
                              0006  53                    push  bx
                              0007  50                    push  ax

2      int i,                                           ; bp-2
           ende,                                        ; bp-4
           var[10];                                     ; bp-24

3      for (i=0; i<ende; i++) {
                              0008  C7 46 FE 0000         mov   word [bp-2], 0
                              000D  EB 34 90              jmp   lbl_chkl
                              0010                  lbl_loop:
4         if (var[i] > 0) {
                              0010  8B 46 FE              mov   ax,word [bp-2]
                              0013  D1 E0                 shl   ax,1
                              0015  8D 5E E8              lea   bx,word [bp-26]
                              0018  03 D8                 add   bx,ax
                              001A  36: 83 3F 00          cmp   word [ss:bx],0
                              001E  7E 11                 jle   lbl_else
5           var[i]++;
                              0020  8B 46 FE              mov   ax,word [bp-2]
                              0023  D1 E0                 shl   ax,1
                              0025  8D 5E E8              lea   bx,word [bp-24]
                              0028  03 D8                 add   bx,ax
                              002A  36: 83 07 01          add   word [ss:bx],1
6        } else {             002E  EB 0F 90
                                                          jmp   lbl_endl
                              0031                  lbl_else:
7            var[i]--;
                              0031  8B 46 FE              mov   ax,word [bp-2]
                              0034  D1 E0                 shl   ax,1
                              0036  8D 5E E8              lea   bx,word [bp-24]
                              0039  03 D8                 add   bx,ax
                              003B  36: 83 2F 01          sub   word [ss:bx],1
8        } /* endif */
                              003F                  lbl_endl:
9     } /* endfor */
                              003F  83 46 FE 01           add   word [bp-2],1
                              0043                  lbl_chkl:
                              0043  8B 46 FC              mov   ax,word [bp-4]
                              0046  39 46 FE              cmp   word [bp-2],ax
                              0049  7C C5                 jl    lbl_loop
10 }
                              004B  58                    pop   ax
                              004C  5B                    pop   bx
                              004D  89 EC                 mov   sp,bp
                              004F  5D                    pop   bp
                              0050  CB                    ret
```

4.11 Interrupts

Interrupts können beim 8086 durch Hardware und durch Software ausgelöst werden. Für Hardware-Interrupts stehen 2 Eingänge, NMI und INT, zur Verfügung. Für Software-Interrupts gibt es den Befehl INT [Nummer].

Interrupts werden beim 8086 aufgerufen, indem mit der Nummer des Interrupts der ***Interrupt-Vektor*** (die Adresse der aufzurufenden Interrupt-Routine) aus der ***Interrupt- Tabelle*** geladen wird. Die Nummer des Interrupts wird entweder von der Hardware geliefert (siehe Interrupt Controller 8259A) oder im Befehl vorgegeben (z.B. "INT 35").

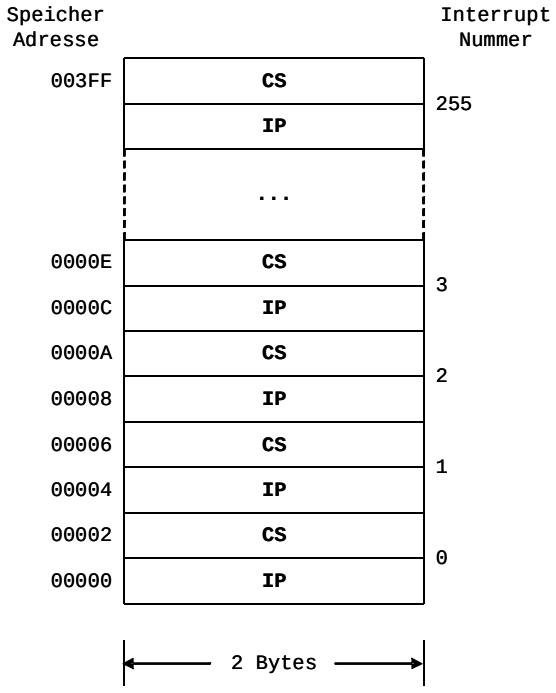
- Interrupts sind durch Software oder über Hardware auslösbar
- Vorgehensweise beim Aufruf
  - Momentane Flags, CS und IP auf den Stack
  - Neues CS und IP aus der Tabelle laden
  - ... (Routine ausführen)
  - Routine mit 'IRET' verlassen

Wird ein Interrupt ausgelöst, so werden automatisch zunächst die Flags auf den Stack gespeichert und das TF und das IF Flag zurückgesetzt. Dann wird die Rücksprungadresse auf dem Stack abgelegt (CS:IP; erst CS, dann IP). Danach wird die Einsprungadresse in die Interrupt-Routine aus der Interrupt-Tabelle geladen. Die Rückkehr aus einer Interrupt-Routine ins aufrufende Programm muss durch den Befehl IRET erfolgen, der (im Gegensatz zu RET) als letzte Aktion die Flags vom Stack holt.

Beim IBM-PC und kompatiblen und beim Betriebssystem DOS werden Interrupts auch dazu verwendet, Systemfunktionen aufzurufen. In den Kapiteln "BIOS Interrupts" auf Seite 79 und "DOS Interrupts" auf Seite 85 sind diese Funktionen aufgelistet.

Interrupt Tabelle des 8086

Die Interrupt-Tabelle beginnt beim 8086 fest bei Adresse 00000h im physikalischen Speicher. Sie enthält 4 Bytes für jeden Interrupt (CS:IP der definierten Interrupt-Routine).



Es sind 256 Interrupts möglich, d.h. die Interrupt-Tabelle kann maximal 1024 Bytes (000h-3FFh) lang sein.

Unter bestimmten Bedingungen, z.B. Division durch Null, werden automatisch bestimmte Interrupts erzeugt:

Interrupt	***Typ***	***Bedeutung***
Divide Error	0	Division durch Null (DIV, IDIV)
Single Step	1	Einzelschritt Interrupt (über Flag "TF" gesteuert)
NMI	2	non-maskable Interrupt
Breakpoint	3	1-Byte Interrupt (INT; benutzt für Breakpoints)
Overflow	4	Aufruf über INTO, wenn OF gesetzt
reserviert	5-31	"reserviert" für Intel (zukünftige Produkte, z.B. 80186)

Code Beispiele

Um einen Interrupt auslösen zu können, muss die Einsprungadresse, also der Beginn der Interrupt-Routine, in die Interrupt-Tabelle eingetragen werden. In diesem Abschnitt soll betrachtet werden, wie dies programmiert werden kann.

Als Beispiel soll folgende Interrupt-Routine betrachtet werden, die als INT 45 aufgerufen werden soll:

```
segment INTSEG            ; Interrupt Routinen in eigenes Segment
int_fkt:
        push ...
        ...
        pop  ...
        iret              ; Wichtig: iret; nicht ret, wg. Flags
```

Der entsprechende Zeiger auf die Routine kann nun folgendermaßen in die Interrupt-Tabelle eingetragen werden:

```
fkt_ptr dd INTSEG:int_fkt  ; praktisch: Pointer definieren

        mov  ax,0          ; "immediate data" direkt in ES geht nicht!
        mov  es,ax
        mov  di,180        ; Interrupt-Tabelle, Eintrag für INT 45
        mov  ax,word [fkt_ptr]
        mov  [es:di],ax
        add  di,2
        mov  ax,word [fkt_ptr+2]
        mov  [es:di],ax
```

Steht das Betriebssystem DOS zur Verfügung, geht dies allerdings auch eleganter mit Hilfe des DOS-Interrupt 21h, Funktion 25 (siehe "DOS Interrupts" auf Seite 85):

```
        push ds
        lds  dx,[fkt_ptr]
        mov  ah,25h        ; Funktion innerhalb INT 21h
        mov  al,45         ; Interrupt-Nummer
        int  21h           ; DOS Interrupt auslösen
        pop  ds
```

Der Aufruf des Interrupt 45 (hier: Dezimal) funktioniert dann wie folgt:

```
        int  45
```

4.12 Prozessor Reset

Nach einem RESET des Prozessors sind folgende Werte in den Registern enthalten:

alle Flags	0
Instruction Pointer	0000h
Prefetch Queue	Leer
CS Register	FFFFh
DS Register	0000h
SS Register	0000h
ES Register	0000h

Daraus folgt, dass der erste nach einem Reset ausgeführte Befehl die Instruktion an der Adresse FFFF0h ist.

5 Der Co-Prozessor 8087

5.1 Übersicht

Der als Co-Prozessor für den 8086 (oder 8088, 80186, 80188) ausgelegte 8087 stellt arithmetische Operationen und Vergleichsoperationen für verschiedene arithmetische Datentypen zur Verfügung. Auch trigonometrische und logarithmische Funktionen sind in diesem Prozessor implementiert. Diese Funktionen sind in Hardware relativ aufwendig und teuer zu realisieren und deshalb in der Basis-CPU nicht enthalten.

Der 8087 bietet jedoch mehr als nur höhere Geschwindigkeit für Floating-Point Operationen. Er bietet auch die Möglichkeit, mit sehr großen und sehr kleinen Zahlen wesentlich höhere Genauigkeiten zu erreichen. Die Verwendung eines 8087 könnte also sinnvoll sein, wenn

- eine große Bandbreite der zu verarbeitenden Werte erforderlich ist
- sehr große oder sehr kleine Zwischenergebnisse auftreten können
- eine hohe Genauigkeit und Stabilität der Ergebnisse gefordert ist
- sehr schnelle Fließkomma-Berechnungen ausgeführt werden müssen.

Der 8087 wird auch "Numeric Processor Extension (NPX)" genannt. Ein Programmierer sieht den 8087 im allgemeinen nicht als separate Einheit, sondern kann die CPU (8086) als um die Register und Funktionen des 8087 erweitert betrachten. Befehle, die zur Laufzeit vom 8087 ausgeführt werden, werden ganz normal im Assembler-Code eingebettet. Ein zu einem Assembler gehörendes Paket von Emulationsprozeduren ermöglicht es sogar, Programme, die 8087-Befehle enthalten, so zu assemblieren, dass sie auf Rechnern ohne Co-Prozessor lauffähig sind. Dazu muss der Source-Code nicht geändert werden.

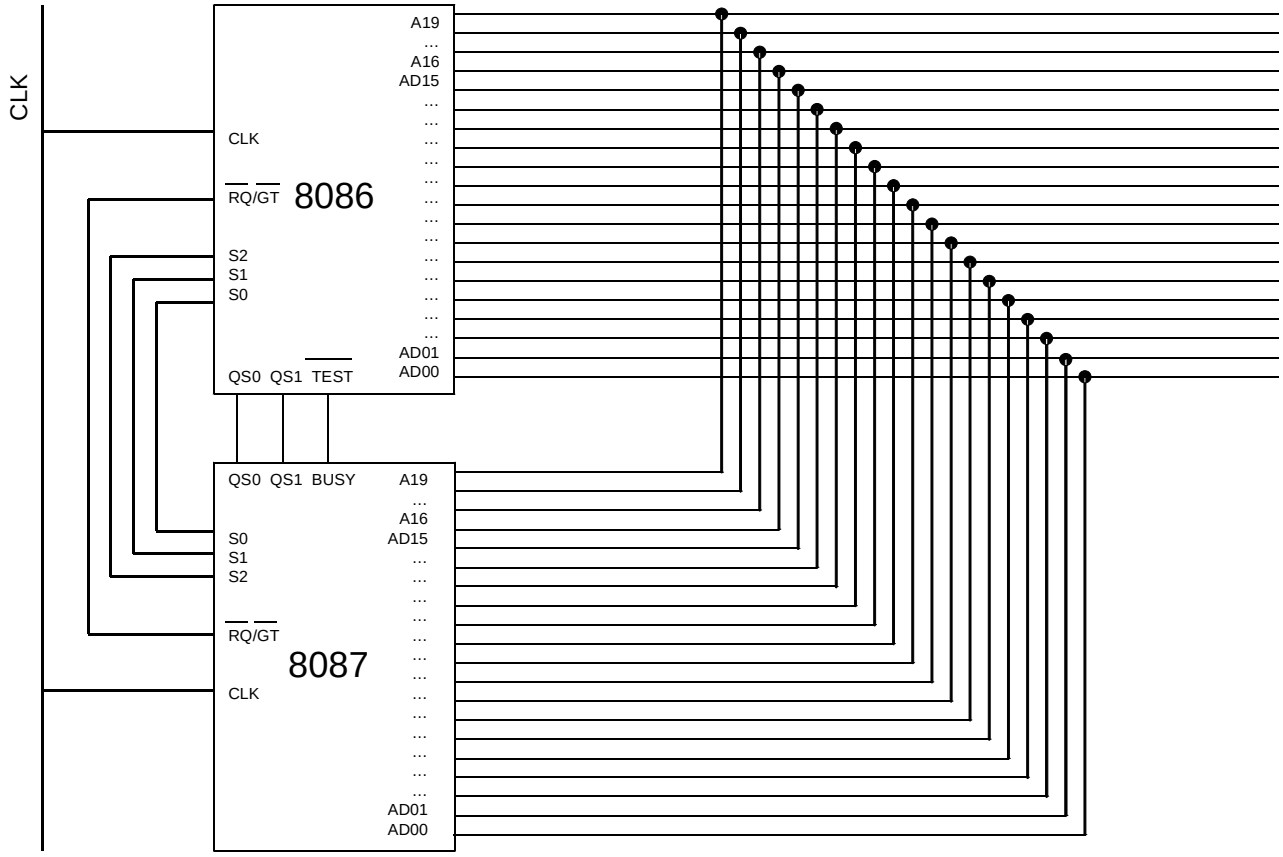
Die Funktionalität des 8087 kann also auch durch (hoch optimierte) Assembler Prozeduren auf dem 8086 emuliert werden (die Emulationsprozeduren vergrößern ein Programm um ca. 16kB). Einen Vergleich der Ausführungszeiten in µs dieser Emulation auf einem 5MHz 8086 mit den Operationen des 8087 zeigt die folgende Tabelle:

Instruction	***8087 (in µs)***	***8086 Emulation (in µs)***
Multiplikation (einfache Genauigkeit)	19	1600
Multiplikation (doppelte Genauigkeit)	27	2100
Addition	17	1600
Division (einfache Genauigkeit)	39	3200
Vergleich (Compare)	9	1300
Laden	9	1700
Speichern	18	1200
Quadratwurzel	36	19600
Tangens	90	13000
Potenzieren	100	17100

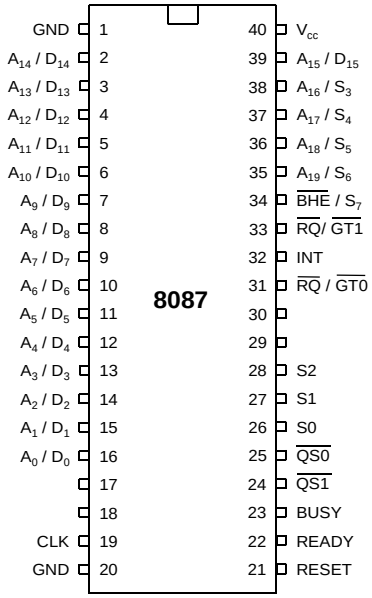
Anm.: "Einfache Genauigkeit" bedeutet 32-Bit Fließkommazahlen

8086 - 8087 Interface

In einem 8086/8087 System sind die beiden Prozessoren hardwaremäßig direkt parallel geschaltet. Die Synchronisation erfolgt über das Busy-Signal des 8087 und den TEST-Pin des 8086. Das TEST-Pin des 8086 wird mit der WAIT-Instruktion abgefragt (näheres siehe auch "8087 Befehlssatz" auf Seite 41). Die Buskontrolle wird über RQ/GT mit der CPU arrangiert.



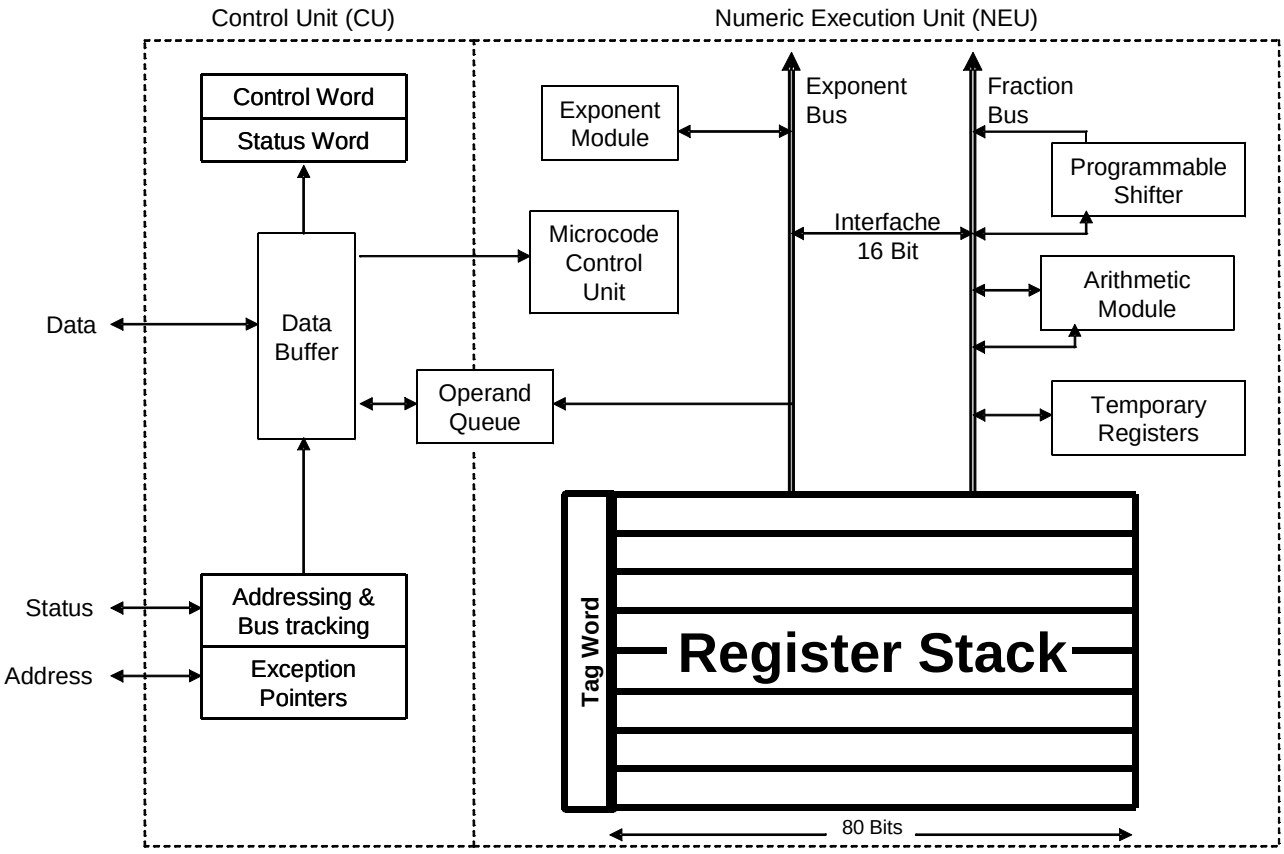
Die Kommunikation zwischen CPU und NPX erfolgt nicht über I/O der CPU, sondern über eine spezielle Instruktion (ESCAPE). Der Co-Prozessor "hört mit" (über eine eigene Befehlsqueue), welche Instruktionen von der CPU eingelesen werden. Wird ein ESCAPE erkannt, so wird der 8087 aktiv.



Soll der 8087 einen Speicherzugriff ausführen, so führt die CPU zunächst einen "dummy read" aus, d.h., der Beginn einer Variablen (die Adresse des ersten Bytes der Variablen im Speicher) wird von der CPU berechnet (wie im Abschnitt "Speichersegmentierung und Segmentregister" auf Seite 19 dargestellt) und auf den Bus gelegt, die Daten werden jedoch ignoriert. Der 8087 jedoch liest die Daten ein, merkt sich die Adresse und führt dann seine weiteren Datenzugriffe (wenn mehr als 16 Bits benötigt werden) ausgehend von dieser Adresse aus.

5.2 Prozessor Architektur

Der 8087 ist intern in zwei Einheiten geteilt, die Control Unit (CU) und die Numeric Execution Unit (NEU). Die NEU führt die numerischen Befehle aus, während die CU Befehle liest (bzw. die von der CPU gelesenen Befehle mithört) und Datenzugriffe ausführt.



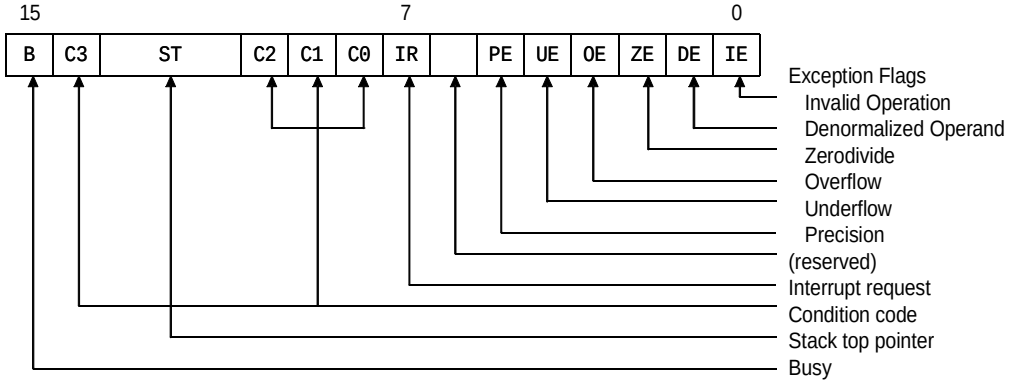
Die NEU unterhält einen ***Register Stack***, in dem alle arithmetischen Operationen des 8087 ablaufen. Der Register Stack besteht aus 8 Registern der Größe "temporary real", d.h. Register von 80-Bit Breite (für die Bedeutung von "temporary real" siehe 5.4, "Datenformate" auf Seite 39). Berechnungen, die mit dem 8087 ausgeführt werden, arbeiten immer mit dem Register Stack. Der Register Stack kann wie ein Stack angesprochen werden (PUSH und POP; oberstes Register ist Stack Top), die einzelnen Register können aber auch direkt angesprochen werden (ST(0) ist gleich Stack Top, danach kommt ST(1), ST(2), ..., ST(7)).

Zu jedem Register des Register Stack gibt das ***Tag Word*** an, ob der Wert gültig (00), gleich Null (01), unendlich oder eine andere ungültige Zahl (10), oder leer (11) ist.

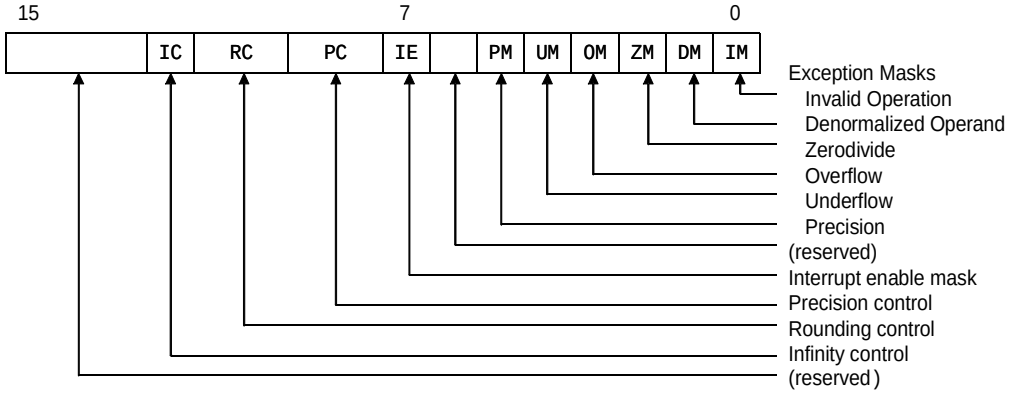
Das ***Status Word*** beinhaltet den allgemeinen Zustand des 8087. Einige 8087-Befehle (z.B. Vergleichsoperationen) setzen den "condition code" (Bits 8-10 und 14 des Status-Wortes), was dann von der CPU für bedingte Sprungbefehle benutzt werden kann. Auch das aktuelle Stack Top und das BUSY-Flag sind im Status-Wort abgelegt. Des weiteren können bei der Ausführung von

Befehlen einige "Exceptions" erkannt und mittels eines Interrupts zur CPU gemeldet werden (siehe 5.3, "Floating-Point Exceptions" auf Seite 38). Im Status-Wort ist dann ein Flag gesetzt, das anzeigt, welche "Exception" aufgetreten ist.

Das Status-Wort kann von der CPU ausgewertet werden, indem es zunächst (ausgelöst durch einen 8087-Befehl) im Speicher abgelegt wird und von dort in die CPU geladen wird.



Im ***Control Word*** wird die Operationsweise des 8087 definiert. Das Kontroll-Wort beschreibt, wie gerundet wird, wie unendliche Werte angezeigt werden, die Genauigkeit, mit der gerechnet werden soll und welche "exceptions" erkannt und im Status-Wort reflektiert werden sollen.



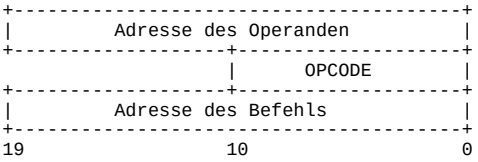
5.3 Floating-Point Exceptions

Exceptions zeigen Fehler bei floating-point Operationen an. Die folgenden Fälle können zu einer oder mehreren Exceptions führen:

- Über- oder Unterlauf
- Ein Register im Stack überladen werden soll, das nicht leer ist (Stack overflow)
- Ein Register mit POP vom Stack geholt werden soll, das leer ist (Stack underflow)
- Der Operand macht eine Operation ungültig (z.B. Wurzel einer negativen Zahl)
- Ein Operand ist unendlich oder das Ergebnis einer vorhergegangenen ungültigen Operation (undefinierter Wert; Not-A-Number)
- Division durch Null
- Genauigkeit überschritten (es musste gerundet werden)

Exceptions können über das Control Word maskiert werden und werden durch Bits im Status Word angezeigt.

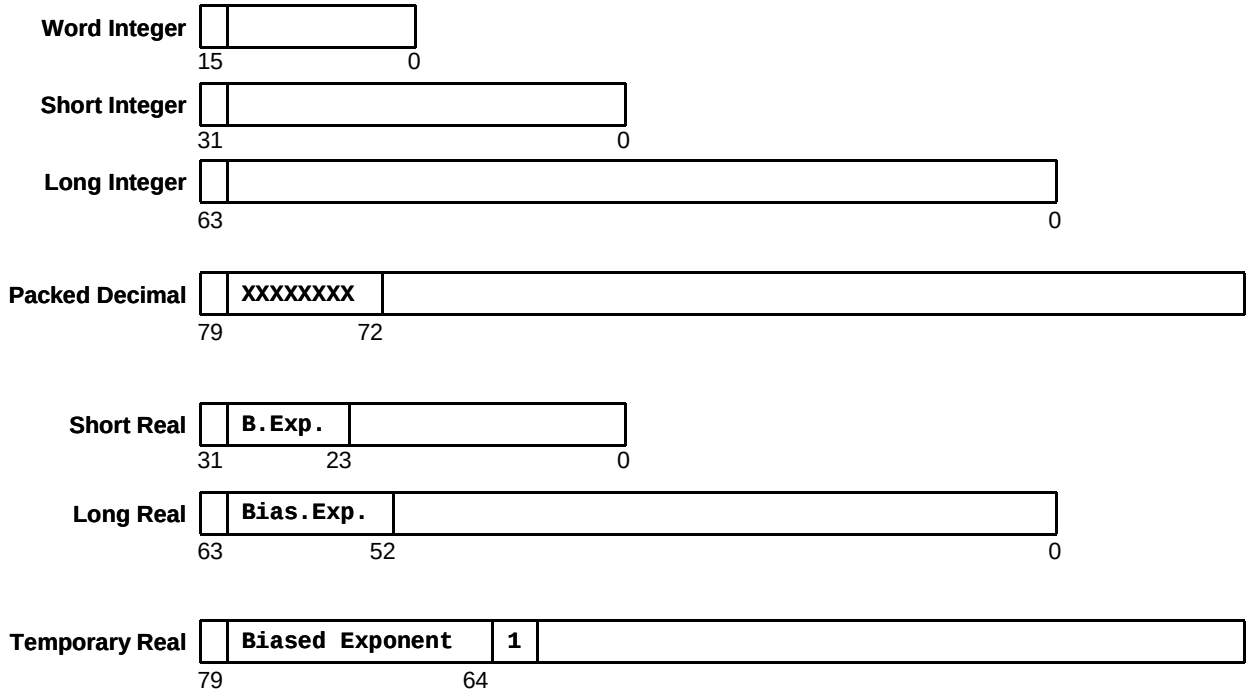
Tritt eine "Exception" auf, so werden außerdem die in der folgenden Abbildung gezeigten ***Exception Pointer***s gesetzt, die dann über die Instruktion **FSTENV** (siehe "8087 Befehlssatz" auf Seite 41) abgespeichert und von einem Exception Handler der CPU ausgewertet werden können.



Die in den Exception Pointers enthaltenen Adressen sind 20-Bit physikalische Adressen. OPCODE gibt die 11 "least significant" Bits des Maschinenbefehls an; die 5 "most significant" Bits sind immer der ESCAPE-Befehl (11011b)

5.4 Datenformate

Der 8087 verarbeitet verschiedene Datenformate. Die Tabelle am Ende des Abschnitts zeigt, in welchen Bereichen Zahlen bei den verschiedenen Darstellungsmodi liegen können.



Binäre Integer-Werte

Die drei binären Integer-Darstellungen sind bis auf die Länge (Wertebereich), d.h. den Bereich, in dem Zahlen in dem bestimmten Format dargestellt werden können, gleich. Binäre Integer-Werte werden als 2er-Komplement dargestellt.

Dezimale Integer-Werte

Für die dezimale Integer-Darstellung werden immer 2 Stellen pro Byte gepackt, außer im höchsten Byte, das lediglich das Vorzeichenbit enthält. Dezimale Integer-Werte werden nicht als 2er-Komplement dargestellt. Positive und negative Werte unterscheiden sich nur durch das Vorzeichen.

Real-Werte

Real-Zahlen werden dreigeteilt dargestellt, ein Vorzeichen, ein Exponent und die Mantisse. Das MSB gibt das Vorzeichen an. Die Aufteilung der restlichen Bits in Exponent und Mantisse erfolgt nach dem IEEE-Standard. Dabei stehen für den Exponent bei Short Real 8 und bei Long Real 11 Bits zur Verfügung. Der Exponent von Real-Zahlen wird mit einem Offset (Bias) versehen, damit keine negativen Werte entstehen.

Die folgende Abbildung zeigt, wie die Dezimalzahl 178,125 als Real-Zahl dargestellt wird.

Dezimalzahl	**`178,125`**
wissenschaftliche Darstellung:	**`1,78125 E+2`** (Normalisierung)
Binärdarstellung	**`10110010,001`**
wissenschaftliche Binärdarstellung	**`1,0110010001 E+111`**
... mit "biased" Exponent	**`1,0110010001 E10000110`**

Die daraus resultierende Darstellung als "Short Real" ist:

Vorzeichen	Exponent	Signifikante Stellen
`0`	**`10000110`**	**`011 0010 0010 0000 0000 0000`**

"1" implizit vorangestellt

Exponent Bias für	
Short Real	127 (7Fh)
Long Real	1023 (3FFh)
Temporary Real	16383 (3FFFh)

Die folgende Tabelle zeigt die Wertebereiche der 8087 Datentypen:

Datentyp	***Bits***	***Signifikante Stellen***	***Ungefährer Bereich (Dezimal)***
Word integer	16	4	-32768 <= X <= +32767
Short integer	32	9	$-2x10^{9} <= X <= 2x10^{9}$
Long integer	64	18	$-9x10^{18} <= X <= 9x10^{18}$
Packed decimal	80	18	-99..99 <= X <= 99..99 (18 Stellen)
Short real	32	6-7	$8.43x10^{-37} <= \|X\| <= 3.37x10^{38}$
Long real	64	15-16	$4.19x10^{-307} <= \|X\| <= 1.67x10^{308}$
Temporary real	80	19	$3.4x10^{-4932} <= \|X\| <= 1.2x10^{4932}$

5.5 8087 Befehlssatz

Der folgende Abschnitt beschreibt die Befehle, die vom 8087 verarbeitet werden. Jeder mnemonische Befehl beinhaltet zunächst den Maschinenbefehl "WAIT", wodurch sichergestellt wird, dass der vorhergehende 8087-Befehl komplett ausgeführt ist, bevor der aktuelle Befehl beginnt. Nach dem Maschinencode "WAIT" steht das Binärmuster für ESC (11011b), das zum einen von der CPU erkannt wird (evtl. "dummy read" ausführen oder nichts tun; siehe auch "8086 - 8087 Interface" auf Seite 36), zum anderen den 8087 aktiviert. Aus den restlichen Bits des Befehls erkennt der 8087 dann die konkrete Anforderung.

Wie schon in "Prozessor Architektur" auf Seite 37 erwähnt, werden alle Operationen auf dem Register Stack des 8087 ausgeführt. Die folgende Tabelle zeigt, wie die Operanden für einen 8087-Befehl angegeben werden können:

Form	***Mnemonic***	***Beispiel***
Classical stack	F*op* {ST,ST(1)}	FADD
Register	F*op* ST(i),ST oder F*op* ST,ST(i)	FSUB ST(3),ST FSUB ST,ST(3)
Register pop	F*op*P ST(i),ST	FMULP ST(2),ST
Real memory	F*op* {ST,} short/long-real	FDIV [AZIMUTH]
Integer memory	FI*op* {ST,} word/short-integer	FIDIV [N_PULSES]

Geschweifte Klammern { } schließen *implizite* Operanden ein; diese sind hier nur zur Information dargestellt und werden nicht codiert.

```
op =  FADD   destination <-- destination + source
      FSUB   destination <-- destination - source
      FSUBR  destination <-- source - destination
      FMUL   destination <-- destination * source
      FDIV   destination <-- destination / source
      FDIVR  destination <-- source / destination
```

Nachfolgend sind die Befehle des 8087 aufgelistet. Dabei sind die meisten Befehle selbsterklärend. Zwei der Befehle sollen jedoch herausgegriffen und näher erklärt werden.

- Die Operation **FSTENV** (store environment) legt das Environment des 8087 im Speicher ab. Environment beinhaltet das Kontroll-Wort, das Status-Wort und die Exception-Pointers.
- Die Operation **FSAVE** (save state) schreibt den gesamten 8087 Zustand - Environment plus die Register des Register Stack - in den Speicher. Diese Operation wird z.B. benutzt, wenn das Betriebssystem einen Kontext-Switch ausführt (-> Multitasking), oder wenn eine Interrupt-Routine den 8087 benötigt.

FLD	Load real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange registers
FILD	Integer load
FIST	Integer store
FISTP	Integer store and pop
FBLD	Packed decimal (BCD) load
FBSTP	Packed decimal (BCD) store and pop
FADD	Add real
FADDP	Add real and pop
FIADD	Integer add
FSUB	Subtract real
FSUBP	Subtract real and pop
FISUB	Integer subtract
FSUBR	Subtract real reversed
FSUBRP	Subtract real reversed and pop
FISUBR	Integer subtract reversed
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Integer multiply
FDIV	Divide real
FDIVP	Divide real and pop
FIDIV	Integer divide
FDIVR	Divide real reversed
FDIVRP	Divide real reversed and pop
FIDIVR	Integer divide reversed
FSQRT	Square root
FSCALE	Scale
FPREM	Partial remainder
FRNDINT	Round to integer
FXTRACT	Extract exponent and significant
FABS	Absolute value
FCHS	Change sign

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM	Integer compare
FICOMP	Integer compare and pop
FTST	Test
FXAM	Examine
FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	2^X-1
FYL2X	$Y * \log_2 X$
FYL2XP1	$Y * \log_2(X+1)$
FLDZ	Load zero
FLD1	Load 1
FLDPI	Load ¶
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$
FINIT	Initialize processor
FDISI	Disable interrupts
FENI	Enable interrupts
FLDCW	Load control word
FSTCW	Store control word
FSTSW	Store status word
FCLEX	Clear exceptions
FSTENV	Store environment
FLDENV	Load environment
FSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

5.6 Programmier-Beispiel

In diesem Abschnitt wird zunächst eine Schleife in C gezeigt. Anschließend wird dieser Code in Assembler dargestellt, um das Zusammenspiel von Floating Point- und Fixed Point-Instruktionen zu zeigen. Danach ist die Veränderung der Register des Register-Stack während eines Durchlaufs der Schleife grafisch dargestellt.

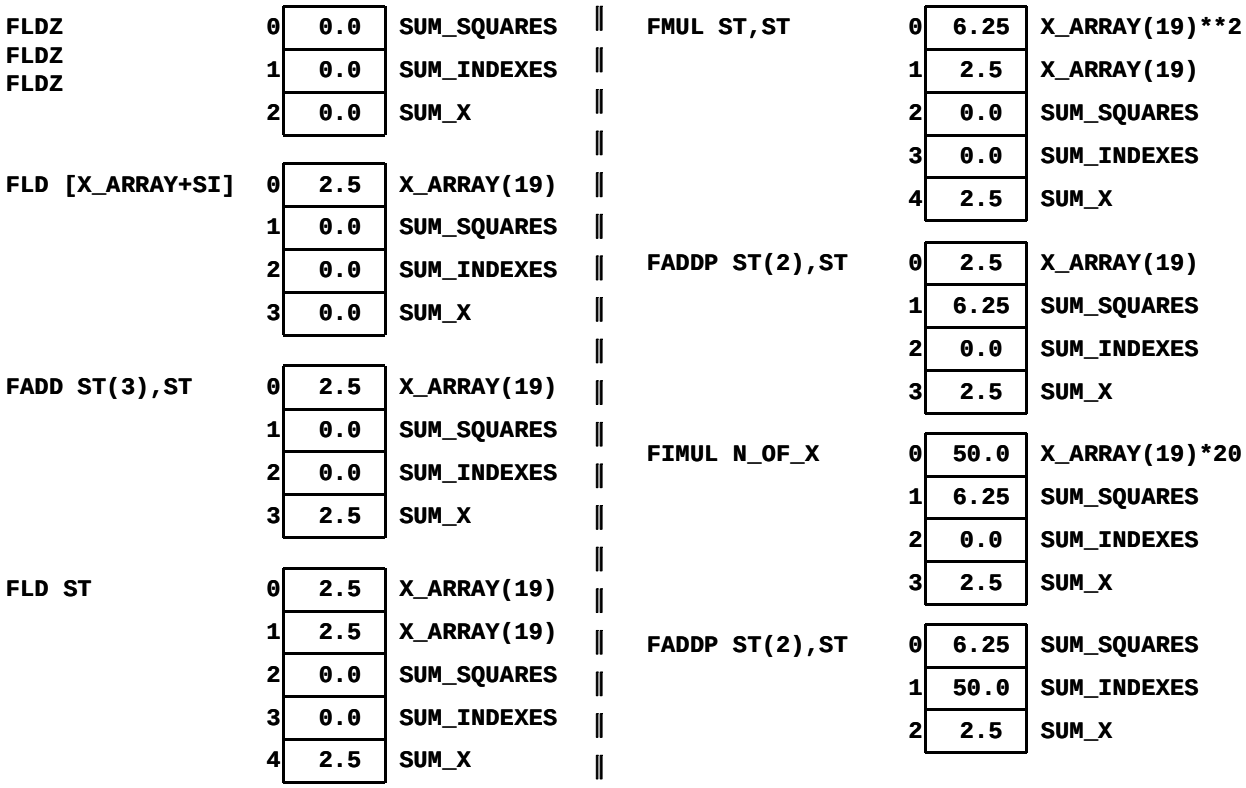
Eine Schleife in C:

```
for (CX=N_OF_X; CX>0; CX--) {                mit Anfangswerten:  N_OF_X = 20
                                                                 X_ARRAY(19) = 2.5
   SUM_X += X_ARRAY[ CX-1];
   SUM_INDEXES += X_ARRAY[ CX-1] * CX;
   SUM_SQUARES += X_ARRAY[ CX-1] * X_ARRAY[ CX-1];
} /* endfor */
```

Die gleiche Schleife in Assembler:

```
segment DATA_SEG
       N_OF_X      DW  20
       X_ARRAY     DD  100 DUP (?)

       SUM_X       DD  ?
       SUM_INDEXES DD  ?
       SUM_SQUARES DD  ?

segment CODE_SEG
       MOV  AX,DATA_SEG
       MOV  DS,AX
       .
       .
       FLDZ                      ; 3 mal 0 auf Stack Top laden
       FLDZ
       FLDZ

       MOV    CX,[N_OF_X]        ; CX ist Schleifenzähler
       JCXZ   exit               ; Exit, wenn ARRAY=0
       MOV    AX,4               ; SI als Index in X_ARRAY laden
       IMUL   CX                 ; multipliziere AX mit CX
       MOV    SI,AX

sum_next:
       SUB    SI,4               ; Index auf nächstes Element
                                 ;  (von hinten nach vorne)
       FLD    [X_ARRAY+SI]       ; Laden auf 8087 Stack
       FADD   ST(3),ST           ; Addieren zu SUM_X
       FLD    ST                 ; Duplizieren X_ARRAY auf ST(0)
       FMUL   ST,ST              ; Quadrieren
       FADDP  ST(2),ST           ; Addieren in SUM_SQUARES und löschen (pop)
       FIMUL  [N_OF_X]           ; Multiplizieren N_OF_X mit ST(0)
       FADDP  ST(2),ST           ; Summieren SUM_INDEXES und löschen (pop)
       DEC    [N_OF_X]
       LOOP   sum_next
exit:
       FSTP   [SUM_SQUARES]      ; Abspeichern
       FSTP   [SUM_INDEXES]
       FSTP   [SUM_X]
```

Veränderung des Register-Stack während eines Schleifendurchlaufs:



5.7 8087 Beispielprogramm

Zum Abschluss dieses Kapitels wird der in der Einführung vorgestellte Programmteil als 8086/8087 Programm betrachtet. Dieser Code wird erzeugt, wenn das Array "var" als "float" definiert ist. Jeglicher Datenaustausch zwischen 8086 und 8087 erfolgt über den Speicher. Auffallend ist dies an der Stelle, an der das 8087 Status Wort abgespeichert wird und vom 8086 als Grundlage für die TEST-Instruktion gelesen wird.

Eine Optimierung dieses Programms ist nur relativ beschränkt möglich, da der 8086 Daten nur über den Speicher mit dem 8087 austauschen kann. Diese Einschränkung besteht ab dem i386 nicht mehr. Die Prozessoren i386 und i486 können Daten aus dem Coprozessor direkt in ein Register laden, was oft mehr Möglichkeiten für Optimierungen eines Programms zulässt. Für den Pentium ist dann die Trennung zwischen Prozessor und Coprozessor sowieso völlig aufgehoben, jedes Floating-Point-Register kann direkt in jedes CPU-Register übertragen werden und umgekehrt.

```
1  void test( void) {
                                        push  bp
                                        mov   bp,sp
                                        sub   sp,50
                                        push  bx
                                        push  ax

2       int   i,                        ; bp-2
              ende,                     ; bp-4
        float var[10];                  ; bp-44
        int   temp1,                    ; bp-46
              temp2,                    ; bp-48
              temp3;                    ; bp-50

3       for (i=0; i<ende; i++) {
                                        mov   word [bp-2], 0
                                        jmp   lbl_chkl
                                  lbl_loop:
4          if (var[i] > 0) {
                                        mov   ax,0
                                        cwd
                                        mov   word [bp-48],ax
                                        mov   word [bp-46],dx
                                        fldi  dword [bp-48]
                                        mov   ax,word [bp-2]
                                        shl   ax,1
                                        shl   ax,1
                                        lea   bx,word [bp-44]
                                        add   bx,ax
                                        fld   dword [ss:bx]
                                        fxch  ST(1)
                                        ficompp
                                        fstsw [bp-50]
                                        test  [bp-50],04500h
                                        jbe   lbl_else
5             var[i]++;
                                        mov   ax,word [bp-2]
                                        shl   ax,1
                                        shl   ax,1
                                        lea   bx,word [bp-44]
                                        add   bx,ax
                                        fld   dword [ss:bx]
                                        fld1
                                        fadd  ST(0),ST(1)
                                        fstp  dword [ss:bx]
                                        fstp  ST(0)
6          } else {
                                        jmp   lbl_endl
                                  lbl_else:
7             var[i]--;
                                        mov   ax,word [bp-2]
                                        shl   ax,1
                                        shl   ax,1
                                        lea   bx,word [bp-44]
                                        add   bx,ax
                                        fld   dword [ss:bx]
                                        fld1
                                        fsubr ST(0),ST(1)
                                        fstp  dword [ss:bx]
                                        fstp  ST(0)
8          } /* endif */
                                  lbl_endl:
9       } /* endfor */
                                        add   word [bp-2],1
                                  lbl_chkl:
                                        mov   ax,word [bp-4]
                                        cmp   word [bp-2],ax
                                        jl    lbl_loop
10 }
                                        pop   ax
                                        pop   bx
                                        mov   sp,bp
                                        pop   bp
                                        ret
```

6 Der Mikroprozessor 80186

Der Prozessor 80186 stellt eine Besonderheit in der Intel x86-Baureihe dar. Er enthält als einziger Prozessor dieser Baureihe mehr als nur die CPU auf dem Chip. Außer der CPU (EU und BIU) des 8086 enthält er noch einige für ein Mikroprozessorsystem zentrale Peripheriebausteine auf seinem Chip:

- Bus Interface Unit des 8086 (mit geringfügigen Änderungen/Erweiterungen)
- Execution Unit des 8086
- Clock Generator
- Interrupt-Controller
- 3 programmierbare Timer
- DMA-Unit (2 20-Bit Kanäle)
- Programmierbare Chip Select Logik

Die Busstruktur, Bus-Zyklen, Interrupt-Struktur und Adressierungsmöglichkeiten sind gleich denen des 8086. Unterschiedlich zum 8086 ist, dass die Bus-Steuersignale, die der 8086 im Minimum Mode erzeugt und die Status Ausgänge des Maximum Mode gleichzeitig erzeugt werden. Außerdem ist die 80186 CPU ca. 30% schneller als die 8086 CPU, da die Ausführung vieler Instruktionen hardwaremäßig verbessert (benötigen weniger Zyklen) und die Adressberechnung beschleunigt wurde, indem sie durch einen Hardware-Addierer statt durch Mikrocode (wie beim 8086) erfolgt.

Gegenüber dem 8086 wurde beim 80186 auch der Befehlssatz erweitert. Es gibt zusätzlich:

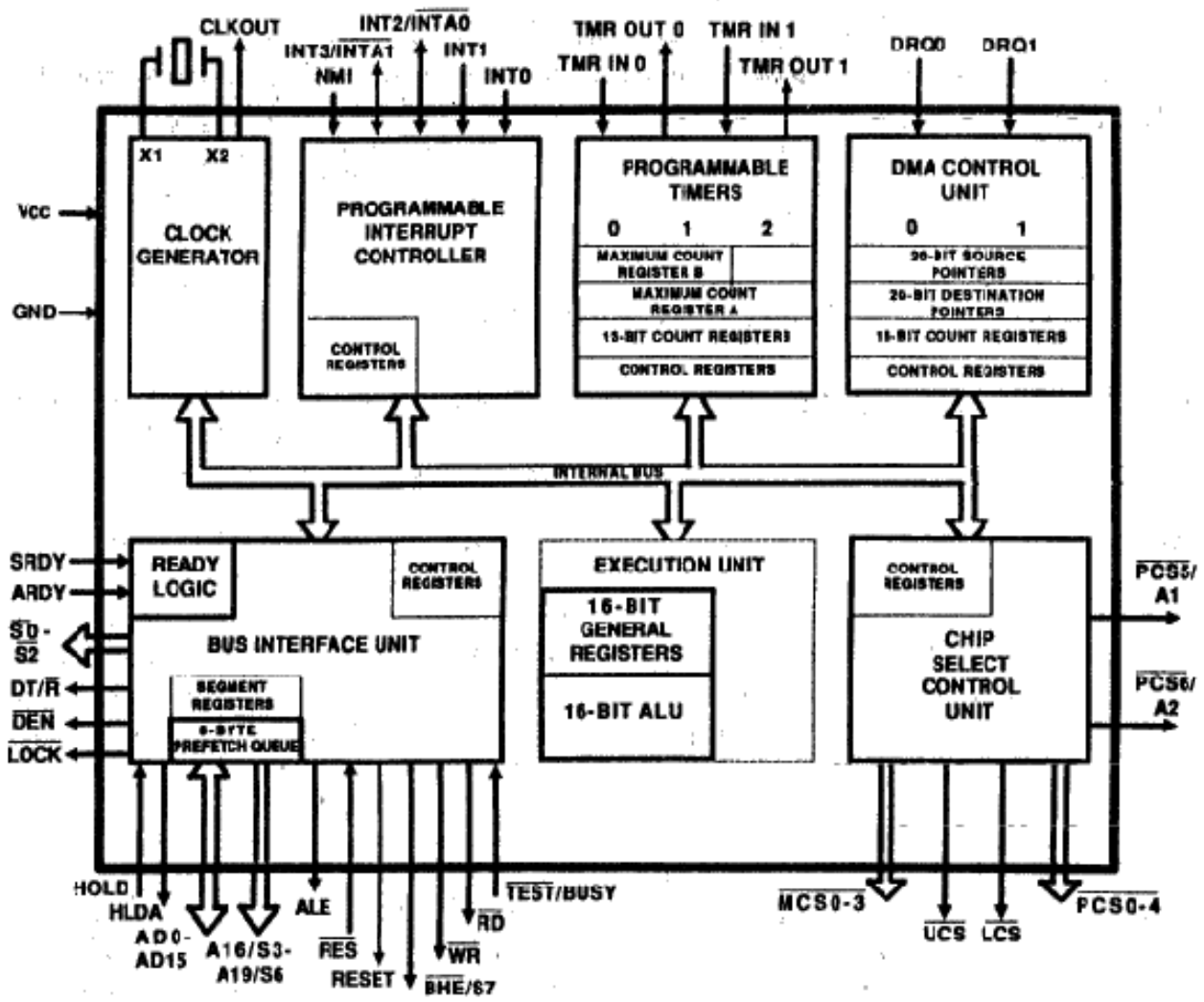
- PUSHI (PUSH von direktem Wert)
- PUSHA, POPA (alle 8 allgemeinen Register AX - DI)
- IMUL (Multiplikation mit direktem Wert)
- Shift/Rotate mit direktem Wert
- Block In/Out (INS/OUTS)
- Feld-Grenzen überprüfen (BOUND) und Interrupt auslösen in einem Befehl
- "High level" Befehle ENTER und LEAVE
- Interrupt, wenn undefinierter Befehl erkannt wird
- Interrupt möglich, wenn ESC erkannt wird

Die Funktionsweise der Interrupts ist gleich wie beim 8086. Durch die auf dem Chip integrierten Baugruppen sind jedoch noch weitere Interrupts festgelegt:

Interrupt	***Typ***	***Priorität***	***in Verbindung mit ... Instruktionen***
Divide Error	0	1	DIV, IDIV
Single Step	1	12	alle
NMI	2	1	alle
Breakpoint	3	1	INT
Overflow	4	1	INTO
Array Bounds	5	1	BOUND
Undef. Opcode	6	1	undefinierte Befehle
ESC Opcode	7	1	ESC + control bit
Timer 0	8	2a	

Interrupt	*Typ*	*Priorität*	*in Verbindung mit ... Instruktionen*
Timer 1	18	2b	
Timer 2	19	2c	
reserved	9	3	
DMA 0	10	4	
DMA 1	11	5	
INT 0	12	6	
INT 1	13	7	
INT 2	14	8	
INT 3	15	9	

Es gibt auch einen 80188, der die gleichen Unterschiede zum 80186 aufweist, wie der 8088 zum 8086 (intern 16-Bit, extern 8-Bit). Die folgende Abbildung zeigt das Blockdiagramm des 80186:



7 Der Mikroprozessor i286

7.1 Einführung

Der i286 ist ein 16-Bit Prozessor, der nicht nur voll kompatibel zum 8086 ist, sondern auch viele Architektur-Gemeinsamkeiten mit diesem hat, z.B. die byteweise, segmentierte Speicheradressierung (aufgrund der 16-Bit-Architektur immer noch max. 64kB große Segmente) oder die über Vektoren gesteuerten Interruptaufrufe. 8086-Programme laufen ohne Re-Kompilation auf dem i286. Der Modus, in dem der i286 für diese Programme betrieben wird, wird als "real addressing mode" oder kurz "**Real Mode**" bezeichnet, da die von einem Programm erzeugten Adressen direkt physikalische Speicheradressen sind, die wie beim 8086 über eine Addition von Segmentbasis und Offset errechnet werden.

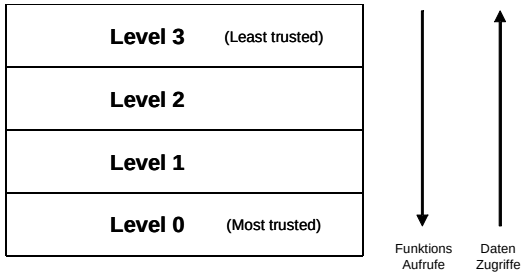
Diese Kompatibilität war zwar wichtig, aber da die Architektur des 8086 inzwischen zu starke Einschränkungen bedeutete, mussten natürlich auch Erweiterungen eingeführt werden. Da der Real Mode solche nicht zuließ, wurde der Operationsmodus "protected virtual addressing mode", oder kurz "**Protected Mode**", eingeführt. Im Protected Mode stehen nicht nur (wie der Name schon sagt) Schutzmechanismen zur Verfügung, die größere Systeme erst kontrollierbar machen. Auch hat der i286 24 Adressleitungen und kann damit physikalisch 16 MByte (2^{24}) adressieren.

Grundlage des Protected Mode ist, dass es eine "Hierarchie der Zuverlässigkeit" gibt, d.h. es gibt privilegierte und weniger privilegierte Programme. Weniger privilegierten Programmen stehen z.B. eine Reihe von Instruktionen nicht zur Verfügung. Außerdem sind Funktionsaufrufe zwischen privilegierten und weniger privilegierten Programmen mit genauen Regeln versehen. Der wesentliche Grund für die Einführung der Protection Mechanismen ist, so gut wie möglich Applikationen davon abzuhalten, das Operating System zu stören oder sogar den Rechner zum Absturz zu bringen.

Außerdem stellt der Prozessor Mechanismen zur Verfügung, die Multitasking ermöglichen. Das Umschalten zwischen verschiedenen Tasks wird durch die Prozessorarchitektur (eine einzelne Instruktion) sehr effizient unterstützt.

7.2 Protection

Der mit dem i286 eingeführte "Protection Mechanismus" unterscheidet vier sogenannte "Privilege Levels". Eine Task (das ist ein Programm oder ein Teil eines Programms) läuft zu einem bestimmten Zeitpunkt immer auf genau einem dieser vier Levels. Auch Daten sind immer einem Privilege Level zugeordnet.

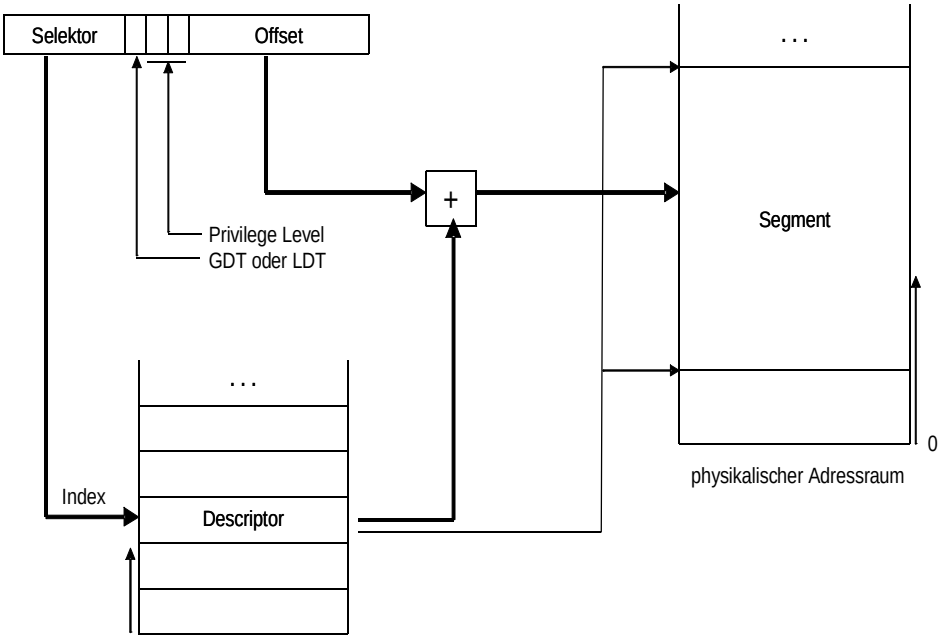


Auf Privilege Level 0 laufen die sichersten ("most trusted") Programme, normalerweise der Kern eines Operating-Systems. Auf Level 3 ("least trusted") laufen Applikations-Programme. Ein Programm kann nur Unterprogramme auf dem gleichen oder höherem (numerisch kleinerem)

Level aufrufen. Ein Programm kann nur auf Daten zugreifen, die sich auf gleichem oder niedrigerem (numerisch größerem) Level befinden.

7.3 Speicheradressierung im Protected Mode

Der Speicher wird im Protected Mode nur noch über Tabellen angesprochen, wodurch Speicherzugriffe besser kontrolliert werden können. Die vom 8086 her bekannten Segmentbasisregister werden für den i286 "Selektor" genannt. Durch den Selektor wird ein Descriptor (ein Eintrag in einer Descriptor-Tabelle) ausgewählt. Mit Hilfe dieses Descriptors und dem angegebenen Offset wird dann ein Speichersegment physikalisch adressiert:



Ein Selektor besteht insgesamt aus einem 16-Bit Wort. Die oberen 13 Bit ergeben, nach rechts mit Nullen aufgefüllt, den Index (Zeiger) in die Descriptor-Tabelle.

Ein weiteres Bit zeigt an, welche der zwei für eine Task möglichen Descriptor-Tabellen angesprochen werden soll:

- Die "Global Descriptor Table" (GDT), die global für das ganze System zur Verfügung steht, oder
- Die "Local Descriptor Table" (LDT), die lokal für jeweils eine Task des Prozessors ist.

Die 2 untersten Bits geben den "Requestor's Privilege Level" (RPL) an, d.h. der Level, der ein Segment anfordert ("requested"). Bei Codesegment-Selektor (CS-Register) geben diese Bits gleichzeitig auch den "Current Privilege Level" (CPL), d.h. der Level, auf dem eine Task gerade läuft.

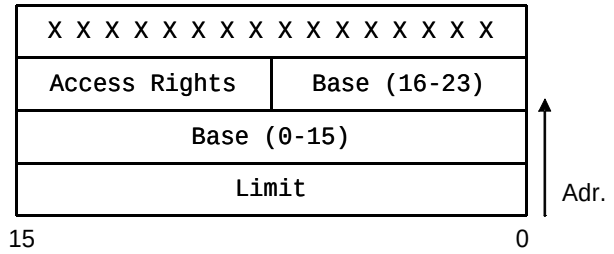
Ein einzelner Eintrag in einer Descriptor Table heißt "Descriptor". Es gibt zwei verschiedene Typen von Descriptoren, **Code- oder Daten-Segment Descriptoren** und **System Descriptoren** wobei es von den letzteren vier Ausprägungen gibt.

Code- und Daten-Segment Descriptoren werden im nächsten Abschnitt beschrieben. System Descriptoren werden in 7.5, "Programmtransfers im Protected Mode" auf Seite 52, in 7.6, "Interrupts und Exceptions" auf Seite 53 und 7.7, "Task Switches" auf Seite 55 behandelt.

7.4 Code- und Daten-Zugriffe im Protected Mode

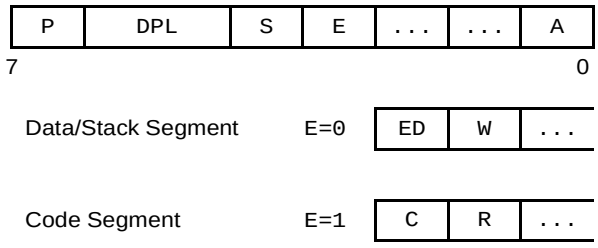
In diesem Abschnitt werden Segment-Descriptoren für Code- und Daten-Segmente beschrieben. Ein solcher Descriptor enthält folgende Werte:

- Physikalische Adresse der Basis des Segmentes (24 Bit)
- Größe des Segmentes ("Limit"; 16 Bit)
- Bits für die Zugriffsrechte ("access rights")



X = Reserviert, normalerweise 0, sollten als „Don‘t Cares“ betrachtet werden

Die Zugriffsrechte für ein Code- oder Daten-Segment (das Bit S ist gleich 1) sehen folgendermaßen aus (Descriptoren mit S=0 sind Descriptoren für System-Segmente, die später beschrieben werden.):



P	Present Flag
DPL	Descriptor Privilege Level
S	Segment (S=1) Control Descriptor (S=0)
E	Executable
A	Accessed
ED	Expand Down
W	Writeable
C	Conforming
R	Readable

Die beschriebene Methode der Speicheradressierung ermöglicht es dem Prozessor, beim Zugriff auf Segmente die Gültigkeit der Zugriffe auf verschiedene Art zu überprüfen. Zum Beispiel wird überprüft, ob der im Descriptor angegebene "Descriptor Privilege Level" (DPL), also der Privilege Level des adressierten Segments für den momentanen RPL oder CPL zulässig ist. Dabei müssen die folgenden Bedingungen erfüllt sein:

- CPL der Task <= DPL des Segments (laden in DS oder ES)
- CPL der Task == DPL des Segments (laden in SS)
- CPL der Task == DPL des Segments (laden in CS, Ausnahmen siehe Gate-Descriptoren)
- RPL im Selektor <= DPL des Segments

Weitere Überprüfungen beziehen sich auf die Zugriffsrechte, z.B. ob ein Segment beschreibbar ist (W=1) oder ob ein Segment ausführbaren Code enthält (E=1). Außerdem wird überprüft, ob sich der Zugriff innerhalb der Segmentgrenzen befindet.

Fällt eine dieser Prüfungen negativ aus, wird eine "General Protection Exception" (GP) ausgelöst. Dies ist ein spezieller Interrupt (Nr. 13, Trap 0Dh), dem die Fehlerart und der Verursacher übergeben werden. Der aufgerufene Exception-Handler kann dann z.B. das den Fehler erzeugende Programm beenden.

7.5 Programmtransfers im Protected Mode

Im vorhergehenden Abschnitt wurde beschrieben, wie Datenzugriffe (auch das Lesen der Befehle aus dem Speicher sind genau genommen Datenzugriffe) im Protected Mode kontrolliert werden. Damit sind alle Lese- und Schreibzugriffe behandelt. Für ein sicheres System ist das allerdings noch nicht ausreichend. Es müssen darüber hinaus auch Programmtransfers kontrolliert werden, d.h. ob ein Sprung oder ein Unterprogrammaufruf an einer bestimmten Stelle zulässig ist. Auch wenn das Sprungziel ein Programm ist, d.h. ein Weiterarbeiten dort theoretisch möglich wäre und Code-Zugriffe über die im vorhergehenden Abschnitt beschriebenen Mechanismen zu keinen Schutzverletzungen führen würden, könnte der Sprung grundsätzlich unzulässig oder unerwünscht sein.

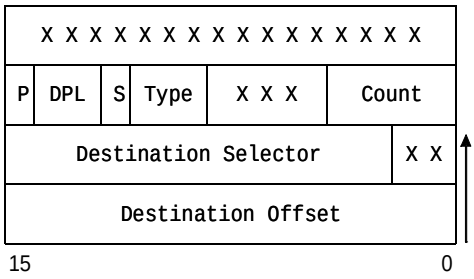
Die folgenden Arten von Programmtransfers sind möglich:

1. Innerhalb eines Segments (*short* jump, call, oder return)
2. Zwischen Segmenten des gleichen "Privilege Levels" (*long* jump, call oder return)
3. Zwischen Segmenten unterschiedlichen "Privilege Levels" (*long* call oder return)

Die ersten beiden Möglichkeiten funktionieren auf herkömmliche Art. Dort wird auch nicht kontrolliert, ob das Ziel der Beginn einer Routine ist. Bei der letzten der genannten Möglichkeiten, wozu dann auch Aufrufe von Betriebssystemroutinen gehören, kann dies jedoch nicht akzeptiert werden.

Programmtransfers zwischen Segmenten unterschiedlichen "Privilege Levels" können nur über **Gates** erreicht werden. D.h., in diesen Fällen wird ein Unterprogramm nicht direkt, sondern über einen Gate-Descriptor aufgerufen. Ein Jump zwischen unterschiedlichen Privilege-Levels ist nicht erlaubt.

Bei einem CALL-Befehl steht nach dem Opcode für den Call eine Adresse. Diese Adresse ist bei einem Long-Call zwischen unterschiedlichen Privilege-Levels jedoch nicht direkt die Einsprungadresse in eine Routine, d.h. es erfolgt kein direkter Zugriff auf ein Code-Segment. Vielmehr enthält die im Call angegebene Adresse einen Selektor, der auf ein Call-Gate zeigt. Der Offset wird vollständig ignoriert.



P	Descriptor gültig
DPL	Descriptor Privilege Level
S	Segment/Control Descriptor
Count	Words kopiert auf den Stack
Type	Descriptor Typ = 4-7
	4 = Call Gate
	5 = Task Gate
	6 = Interrupt Gate
	7 = Trap Gate

Spezielle Behandlung der Felder:

```
Task Gate:      Count ignoriert; Offset ignoriert; Selektor zeigt auf TSS-Descriptor
Interrupt Gate: Count ignoriert
Trap Gate:      Count ignoriert
```

x Reserviert, normalerweise 0, sollten als "Don't Cares" betrachtet werden

Es gibt vier verschiedene Typen von Gates: Call Gates, Task Gates, Interrupt Gates und Trap Gates. Wie man sieht, unterscheidet sich ein Gate-Descriptor (System-Descriptor Typen 4 bis 7) wesentlich von einem Descriptor für ein Code- oder Daten-Segment (z.B. enthält er keine Basisadresse für ein Segment). Wenn ein Call-Gate aufgerufen wird, wird der in der Call-Instruktion angegebene Offset ignoriert. Mit dem in der Call-Instruktion angegebenen Selektor wird das Call-Gate selektiert, das seinerseits den Selektor und den Offset für die gerufene Routine enthält. Mit diesem im Gate-Descriptor enthaltenen Selektor wird dann zunächst der das Code-Segment beschreibende Descriptor ausgewählt, es ist also noch ein weiterer Descriptor-Zugriff

nötig. Zu der in diesem (Segment-) Descriptor enthaltenen Basisadresse wird dann der im Call-Gate enthaltene Offset addiert und die physikalische Adresse des Call-Zieles berechnet.

Folgende Bedingungen müssen beim Transfer über ein Call-Gate erfüllt sein (protection checks):

1. CPL <= DPL des Gates
2. RPL <= DPL des Gates
3. DPL des angesprungenen Code-Segmentes <= CPL

Durch Bedingung 1 wird sichergestellt, dass kein Applikations-Programm Gates aufruft, die nur Systemroutinen zur Verfügung gestellt wurden. Mit Bedingung 2 ist sichergestellt, dass dies auch für durch gereichte Selektoren (Zeiger) gilt. Die Bedingung 3 stellt sicher, dass Unterprogrammaufrufe nie nach weniger privilegierten Programmen erfolgen (auch nicht über Gates).

Jeder Privilege Level besitzt einen eigenen Stack, da Stack-Zugriffe nicht Level-übergreifend möglich sind. Bei einem Unterprogramm-Aufruf wird die Adresse des neuen Stacks aus dem Task State Segment (siehe Abschnitt "Task Switches" auf Seite 55) geholt und der aktuelle Stackpointer und die Return-Adresse auf dem Stack des höheren Levels abgespeichert. Dementsprechend liest ein Return die Adresse von diesem Stack, bevor der Stack wieder umgeschaltet wird. Sollen bei einem Unterprogramm-Aufruf über ein Call-Gate Daten über den Stack übergeben werden, so müssen diese vom Stack des Hauptprogramms in den Stack des Unterprogramms kopiert werden. Damit dieses Kopieren vom Prozessor automatisch und selbständig ausgeführt werden kann, gibt es den Eintrag 'Count' im Gate-Descriptor, der die Anzahl der zu kopierenden Worte (Doppel-Bytes) angibt. 'Count' wird vom System beim Initialisieren des Call-Gates eingetragen (bereitgestellt werden die Einträge von Compilern oder Assemblern, die das ausführbare Programm erzeugt haben).

7.6 Interrupts und Exceptions

Aber nicht nur Jumps und Calls können den Programmfluss verändern. Auch die in diesem Abschnitt behandelten Exceptions und internen und externen Interrupts können die Ausführung eines Programms beeinflussen, genau genommen unterbrechen. Interne Interrupts werden durch die INT-Instruktion ausgelöst, externe Interrupts durch die Hardware über die INTR- oder NMI-Input-Pins. Exceptions werden durch das momentan ausgeführte Programm verursacht.

Ursache für Exceptions sind Fehlerbedingungen während der Ausführung eines Programms, die den Eingriff von System-Software erfordern. Es gibt drei Kategorien von Exceptions: Faults, Traps und Aborts.

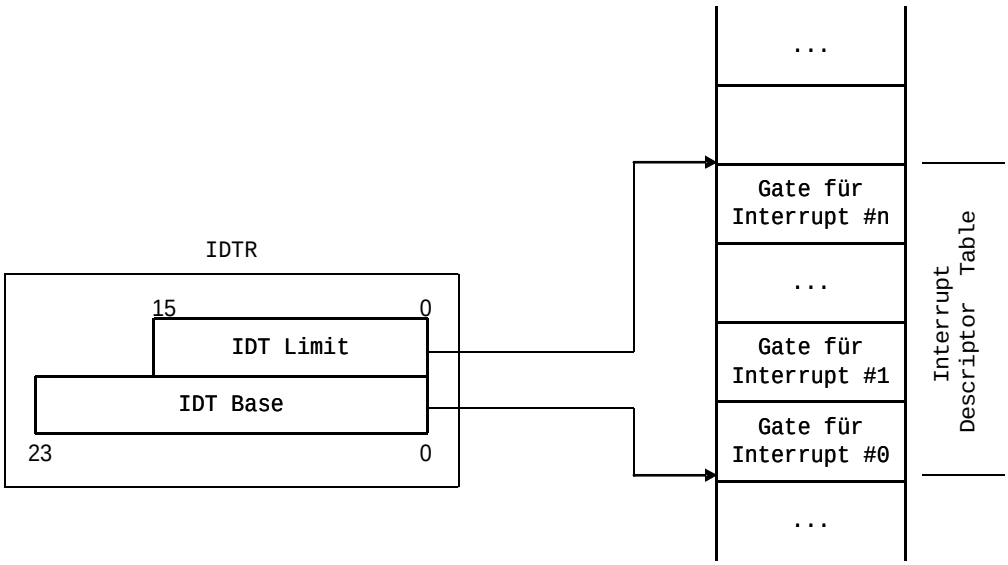
- Ein **Fault** wird ausgelöst, bevor der aktuelle Befehl vollständig ausgeführt ist. Ein Beispiel für einen Fault ist die "segment not present"-Exception, die ausgelöst wird, wenn auf ein Segment zugegriffen werden soll, das sich momentan nicht im Speicher befindet. Der aufgerufene Exception-Handler lädt dann das Segment in den Speicher, nach dem Rücksprung ins Hauptprogramm wird der aktuelle Befehl nochmals ausgeführt und der Speicherzugriff erneut versucht.
- Ein **Trap** wird durch einen Befehl ausgelöst, der nach der Exception nicht nochmals ausgeführt werden darf. Ein Trap wird z.B. ausgelöst, wenn Zugriffsrechte auf Segmente verletzt werden (siehe auch "Speicheradressierung im Protected Mode" auf Seite 50).
- Ein **Abort** liefert im Gegensatz zu Faults und Traps nicht immer die Adresse des Fehlers. Deshalb kann die Programmausführung nach einem Abort u.U. nicht wieder aufgenommen werden. Aborts können z.B. bei Hardware-Ausfällen oder ungültigen System-Tabellen ausgelöst werden.

Obgleich ihre Ursache unterschiedlich ist, der Ablauf beim Aufruf ist der Gleiche bis auf den Unterschied, dass bei einem Interrupt das Interrupt-Enable-Flag während dem Aufruf

zurückgesetzt wird (wie beim 8086), bei einer Exception passiert dies nicht. Prinzipiell funktioniert der Sprung in eine Interrupt-Routine oder einen Exception-Handler ähnlich wie beim 8086 (siehe "Interrupts" auf Seite 32), nämlich dass die Adresse der Ziel-Routine aus einer Tabelle entnommen wird. Allerdings sind die Zugriffe auf diese Tabelle natürlich mit Mechanismen des Protected Mode versehen.

Die Tabelle heißt im Protected Mode Interrupt Descriptor Table (IDT) und muss nicht mehr an der physikalischen Speicheradresse 0 beginnen, sondern kann an einer beliebigen Adresse stehen, die durch das Interrupt Descriptor Table Register (IDTR) angegeben wird.

Ein Software Interrupt oder ein vom Interrupt Controller erzeugter Interrupt Vektor wird dann vom Prozessor als Index in der IDT verwendet und die entsprechende Routine oder Task aufgerufen.



Die IDT kann Interrupt Gates, Trap Gates oder Task Gates enthalten. Interrupt- und Trap-Gates haben denselben Aufbau wie Call-Gates. Lediglich der Eintrag 'Count' hat keine Bedeutung. Interrupt-Gates unterscheiden sich von Trap-Gates darin, dass ein Aufruf über ein Interrupt-Gate die Flags IF und TF löscht, bei einem Trap-Gate geschieht dies nicht. Es sollte also sichergestellt werden, dass ein Hardware-Interrupt immer auf einem Interrupt-Gate landet, ein Software-Trap (oder allgemeiner: eine Exception) immer auf einem Trap-Gate.

Die folgenden Interrupts und Exceptions sind durch die Architektur vorgegeben:

Interrupt Nummer	***Beschreibung***
0	Divisions-Fehler
1	Debug Exception (Single Step)
2	Nonmaskable Interrupt
3	Breakpoint Interrupt
4	Overflow (INTO Befehl)
5	Feldgrenzen (BOUND Befehl)
6	Ungültiger Befehl
7	Coprozessor nicht verfügbar
8	Doppel-Fehler
9	Coprozessor Segment-Überlauf

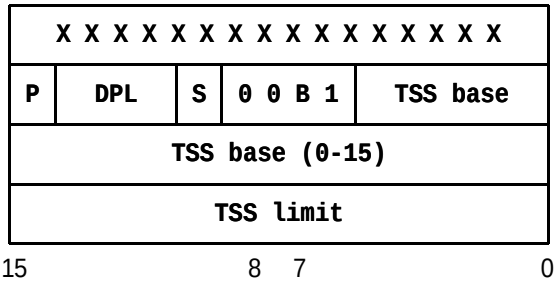
Interrupt Nummer	***Beschreibung***
10	ungültiges Task State Segment
11	Segment not present
12	Stack Exception
13	General Protection (Trap 0D)
14	(reserviert)
15	(reserviert)
16	Coprozessor Fehler
17-31	(reserviert)
32-255	Über Software oder INTR Pin verfügbare Interrupts

7.7 Task Switches

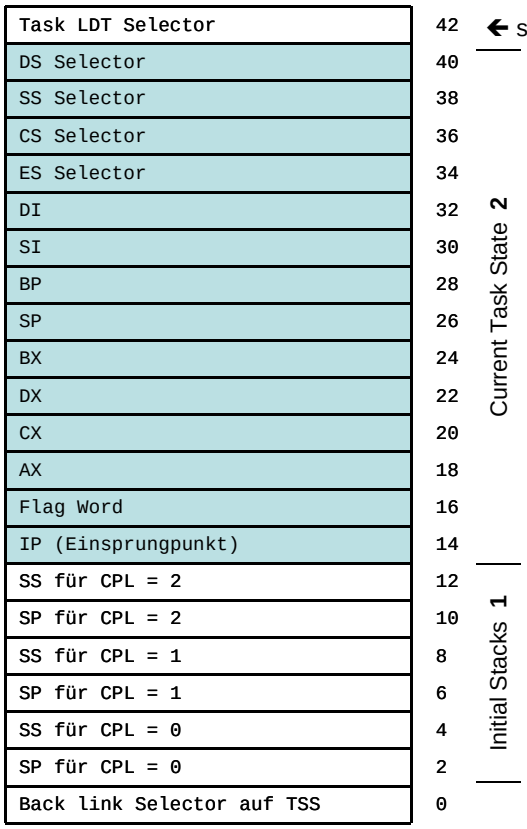
Eine wesentliche Komponente des i286 ist, dass er Multitasking unterstützt. Multitasking bedeutet, dass mehrere Tasks mehr oder weniger parallel ablaufen können. Tatsächlich erreicht man natürlich nur eine scheinbare Parallelität, d.h. einzelne Tasks werden (vom Betriebssystem kontrolliert) jeweils für kurze Zeit ausgeführt, unterbrochen und nach kurzer Zeit an der gleichen Stelle wieder gestartet. Um dies zu erreichen, muss der Zustand eines Tasks zum Zeitpunkt der Unterbrechung vollständig gesichert werden. Multitasking-Mechanismen müssen von der Prozessor-Hardware unterstützt werden. Mit einem 8086 wäre mit vertretbarem Aufwand kein vernünftiges Multitasking zu implementieren.

Die Sicherung des aktuellen Zustandes geschieht im **Task State Segment** (TSS). Jeder Task ist genau ein TSS zugeordnet. Ein TSS enthält alle Register, die Zeiger auf die Stacks für die Privilege Levels 0-2, einen Zeiger auf die Task-spezifische LDT und einen Link zu dem TSS der zuvor ausgeführten Task, falls Tasks verschachtelt werden (siehe auch Nested Task Flag).

Ein Task Switch erfolgt nun, wenn ein *long* Jump oder Call auf ein Task Gate oder einen TSS Descriptor ausgeführt wird. Bei einem Task Switch wird vom Prozessor automatisch und selbständig der Zustand des aktuellen Tasks abgespeichert und das neue Task State Segment geladen. Die folgenden Abbildungen zeigen ein TSS und einen TSS Descriptor (System-Descriptor Typ 1 und 3).



P	Present Flag
DPL	Descriptor Privilege Level
S=0	Segment/Control Descriptor
Type	Descriptor Typ = 1,3 (B: 1=Task busy/0=available)
Base	Phys. base addr. of TSS
Limit	Length of TSS



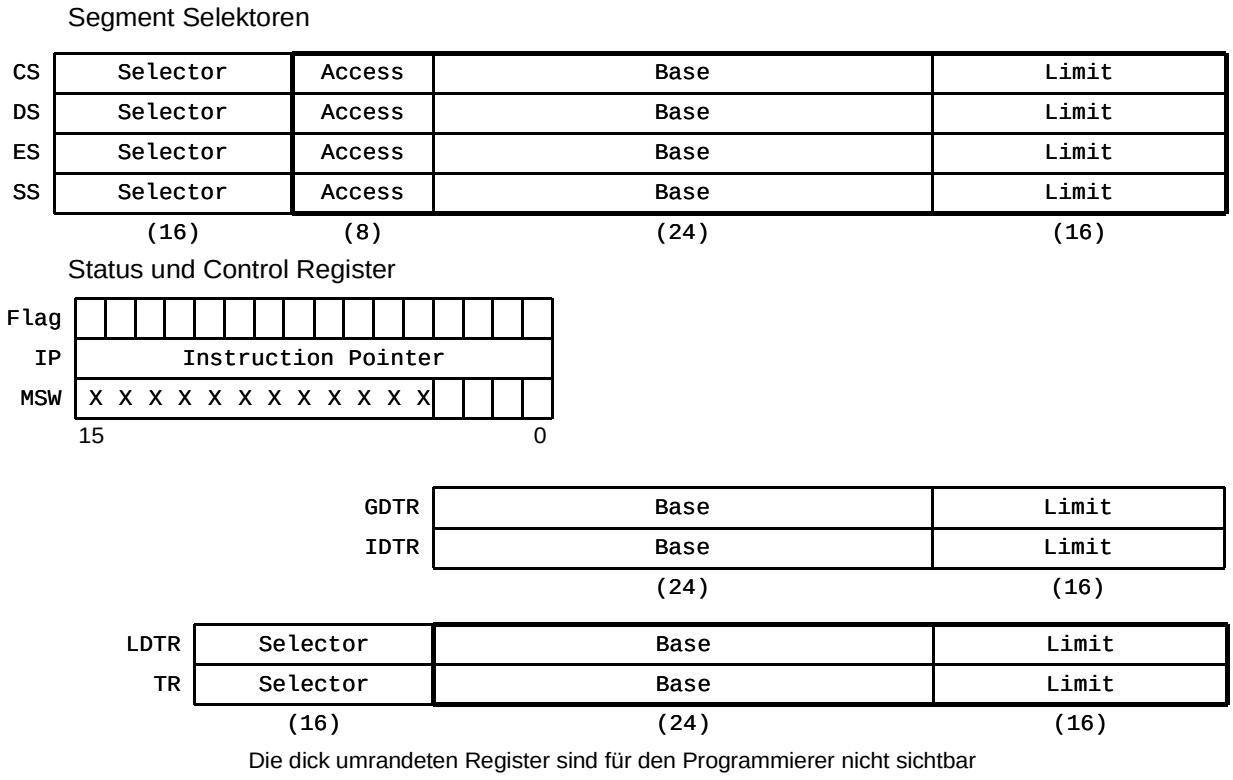
1 Unverändert nach Initialisierung durch Betriebssystem

2 Verändert während Task Switch

TSS Descriptoren können nur in der GDT und nicht in einer LDT stehen, da der Zugriff auf einen TSS Descriptor jederzeit möglich sein muss.

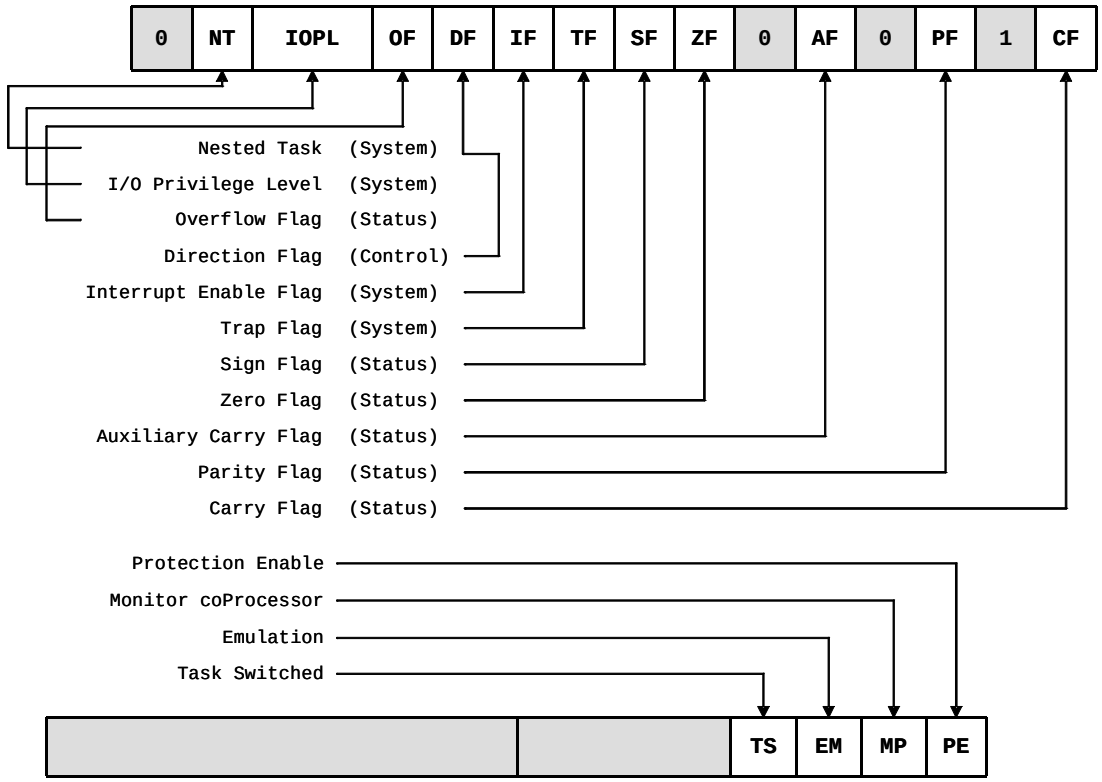
7.8 Registerstruktur des i286

Die allgemeinen Register AX - SP sind für den i286 gleich wie beim 8086. Darüber hinaus stellt der i286 die folgenden Register zur Verfügung.



Die dick umrandeten Register sind für den Programmierer nicht sichtbar

Das Machine Status Word (MSW) und einige Flags zur Steuerung der neuen Funktionen im Protected Mode wurden eingeführt:



8 Der Mikroprozessor i386

Der i386 stellt wie der i286 die zwei Modi "Real Mode" und "Protected Mode" zur Verfügung, wobei der Protected Mode für den i386 gegenüber dem i286 nochmals entscheidend erweitert wurde. Zusätzlich zu diesen beiden Modi gibt es beim i386 noch einen "virtual 8086 mode", in dem mehrere 8086 Prozessoren emuliert werden können.

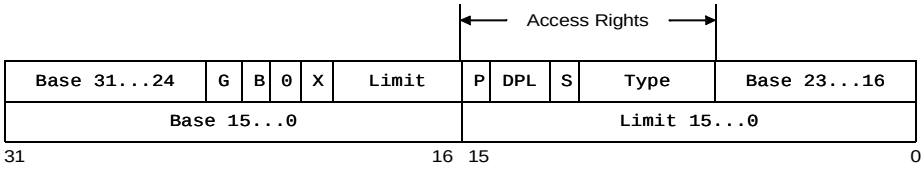
Der i386 ist ein 32-Bit Prozessor. Er stellt 32 Adressleitungen zur Verfügung und kann damit physikalisch 4 Giga-Byte adressieren. Im Protected Mode kann dieser Prozessor 2^{46} Bytes (oder 64 Tera-Bytes) virtuell adressieren.

Als erster Prozessor der x86-Serie ermöglicht der i386 ein "flaches" Speichermodell, da Segmente bis zu 4GByte groß sein können. Damit ist es möglich, die Segmentierung auszuschalten und den physikalischen Speicher als einen einzigen, "flachen" Adressraum zu sehen. Über die Möglichkeiten bei der Speichersegmentierung hinaus stellt der i386 einen Paging-Mechanismus zur Verfügung, der den Adressraum in feste Blöcke der Größe 4kByte, die "Pages", aufteilt.

8.1 Memory Management beim i386

Segment-Adressierung

Auch beim i386 wird Speicher im Protected Mode über Descriptor-Tabellen adressiert. Das Prinzip ist gleich dem des i286, es wurde im wesentlichen lediglich eine Erweiterung der Descriptoren-Einträge für die Segment-Basis und das Limit auf 32-Bit vorgenommen. Ein Descriptor sieht beim i386 also folgendermaßen aus:



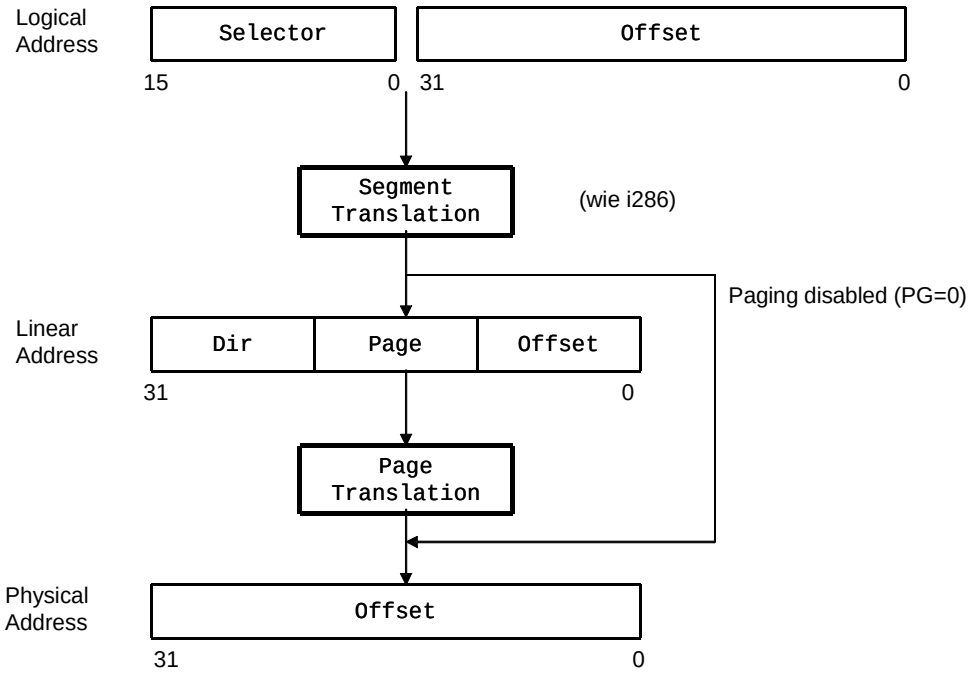
Base	Lineare Basis Adresse des Segments
Limit	Länge des Segments
G	Granularität von „Limit“ (0 = 1B, 1 = 4kB)
B	Big (Daten Segmente), Default (Code Segmente)
X	Verfügbar für Betriebssystem
P	Present Flag
DPL	Descriptor Privilege Level
Type	Descriptor Typ (wie i286)

Die Basisadresse für ein Segment ist 32 Bit breit. Das Limit besteht aus 20 Bit. Damit können entweder Segmente mit der Größe 1 MByte auf ein Byte genau definiert werden (Granularitäts-Bit G=0) oder Segmente mit einer Maximalgröße von 4 Giga-Byte auf 4kByte genau (G=1).

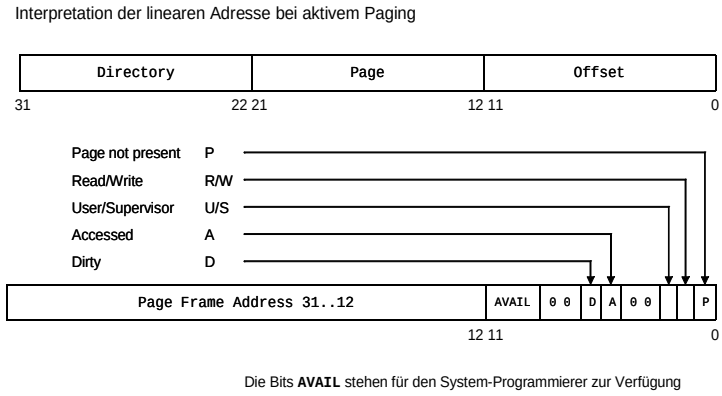
Das Bit B ist im Intel-Handbuch leider nicht beschrieben.

Paging

Für die Adressberechnung hat der i386 über die Segmentierung hinaus zusätzlich die Möglichkeit des "Pagings". Nach der Segmentbestimmung (die wie beim i286 abläuft) kann durch ein Kontroll-Flag (Paging enable) gesteuert noch eine weitere Adressübersetzung vorgenommen werden.



Die aus der Descriptor-Tabelle erhaltene lineare Adresse ist bei aktiviertem Paging nicht mehr gleich der physikalischen Speicheradresse, sondern wird folgendermaßen interpretiert:



Die Bits **AVAIL** stehen für den System-Programmierer zur Verfügung

Beim Paging gibt es unabhängig von den Segmenten wieder Protection Mechanismen. Es gibt zwei Ebenen, denen eine Page zugeordnet sein kann, eine User Ebene und eine Supervisor Ebene (Bit U/S). Außerdem wird auch spezifiziert, ob eine Page nur lesbar oder auch schreibbar ist (Bit R/W). Wird ein Paging-Schutzmechanismus verletzt, so wird ein "Page Fault" (Exception 0Eh) ausgelöst. Page- und Segment-Protection lassen sich auch kombinieren.

Auch in einem Page Table Entry gibt es wieder (wie auch in den Segment-Descriptoren) die Bits P und A, die angeben, ob sich eine Page momentan im Speicher befindet, und ob diese vor kurzem angesprochen (accessed) wurde.

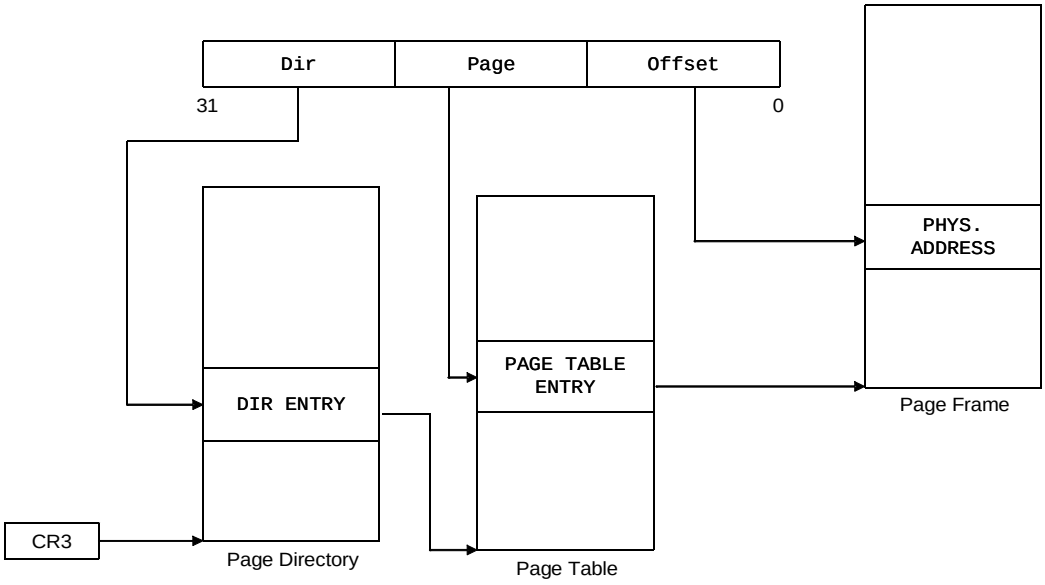
Das 'dirty'-Bit D gibt an, ob eine Page, die sich im Speicher befindet, modifiziert wurde.

Die zu Beginn erwähnte Möglichkeit des "flachen" Speichermodells bedeutet, dass die Segmentierung ausgeschaltet wird und die Speicherverwaltung im Wesentlichen über Paging erfolgt. Dazu werden vier, sich vollständig überlappende und den gesamten Speicher überdeckende Segmente definiert.

Das bedeutet, dass die Segment-Selektoren nie mehr umgeladen werden müssen. Speicherzugriffe werden (fast) ausschließlich über die Protection-Mechanismen beim Paging

kontrolliert. Selbstverständlich werden Calls, Interrupts und Task-Switches genau wie bisher über Gates ausgeführt.

Die folgende Abbildung zeigt, wie aus einer linearen Adresse bei aktiviertem Paging die physikalische Adresse bestimmt wird:



8.2 Der Virtual 8086 Mode

Neben der 32-Bit-Architektur und dem Paging ist der Virtual 8086 Mode (V86) die wesentliche Neuerung gegenüber dem i286. Im Virtual 8086 Mode unterstützt der i386 die Ausführung eines oder mehrerer 8086 Programme in einer Protected Mode Umgebung.

Eine V86 Task besteht aus drei Teilen:

- Dem Real Mode Programm (DOS-Anwendung)
- Einem Virtual-8086 Monitor (V86 Monitor)
- Betriebssystemfunktionen für das Real Mode Programm

Der V86 Monitor ist ein i386 Protected Mode Programm, das auf Privilege-Level 0 läuft. Die Aufgabe des V86 Monitors besteht darin, alle Schnittstellen der virtuellen Maschine zur realen Maschine zu überwachen. Das Real Mode Programm hat dann den Eindruck, auf einem gewöhnlichen 8086 zu laufen. Der V86 Monitor stellt sicher, dass sich die virtuelle 8086-Maschine als eigenständiger Task in das Protected Mode System einfügt.

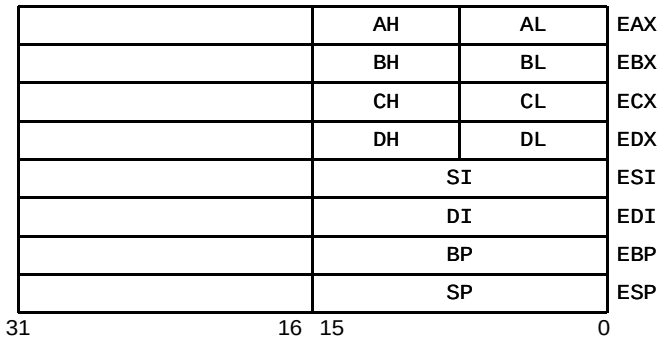
Die Betriebssystemfunktionen für den V86 Mode können entweder das Original Operating System für den 8086 sein, z.B. DOS, oder jedoch eine Nachbildung, die dann ein Teil des V86 Monitors sind und auch im Protected Mode laufen.

Im V86 Mode werden die vom Programm angesprochenen Adressen auf die 8086-Weise berechnet, d.h. die linearen Adressen werden durch Addition von Segmentbasis und Offset berechnet. Einer V86-Maschine müssen also die ersten 1 MByte linearen Adressen zur Verfügung gestellt werden. Daraus folgt auch, dass erst über Paging mehrere (theoretisch beliebig viele) parallel ablaufende Real Mode Programme möglich sind.

Zugriffe auf den I/O-Bereich werden im V86 Mode über die I/O-Permission-Bit-Map geschützt. Diese enthält für jedes Port ein Bit, das angibt, ob der Zugriff gestattet ist oder nicht. Ist ein Bit gesetzt, dann löst der i386 bei einem Zugriff auf das zugehörige Port eine Exception 13 aus, die dann wiederum vom V86 Monitor behandelt werden muss.

8.3 Die Register des i386

Die allgemeinen Register des i386 sind 32 Bit breit und mit EAX, EBX, etc. ansprechbar. Teile (Worte und Bytes) der Register sind (teilweise aus Kompatibilitätsgründen) über z.B. AX, BX, AH, DL adressierbar. Ein Register kann also ein Doubleword (zwei Words oder vier Bytes) enthalten.



Der i386 hat gegenüber dem i286 zwei zusätzliche Segment-Selektor-Register FS und GS und die Control-Register CR0-CR3. Die Register CR0 bis CR3 werden bis auf wenige Bits im MSW (=CR0) ausschließlich beim "Paging" verwendet.

Segment Selektoren

	Selector	Access	Base	Limit
CS	Selector	Access	Base	Limit
DS	Selector	Access	Base	Limit
ES	Selector	Access	Base	Limit
FS	Selector	Access	Base	Limit
GS	Selector	Access	Base	Limit
SS	Selector	Access	Base	Limit
	(16)	(12)	(32)	(20)

Control Register

CR0	reserved	
CR1	reserved	
CR2	Page Fault Linear Address	
CR3	Page Dir Base Reg.	reserved

EIP	Instruction Pointer
EF	Flags

System Register

GDTR		Base	Limit
IDTR		Base	Limit
		(32)	(20)

LDTR	Selector	Base	Limit
TR	Selector	Base	Limit
	(16)	(32)	(20)

Die dick umrandeten Register sind für den Programmierer nicht sichtbar

8.4 80386 Beispielprogramm

Wird das bekannte Beispielprogramm mit MSVC 2.0 für 32-Bit compiliert, sieht das Ergebnis etwa folgendermaßen aus:

```
1  void test( void) {
                                    push  ebp
                                    mov   ebp,esp
                                    sub   esp,48
                                    push  ebx
                                    push  esi
                                    push  edi

2       int i,                      ; bp-4
            ende,                   ; bp-8
            var[10];                ; bp-48

3       for (i=0; i<ende; i++) {
                                    mov   dword [ebp-4], 0
                                    jmp   lbl_loop2
                               lbl_loop1:
                                    inc   dword [ebp-4]
                               lbl_loop2:
                                    mov   eax, dword [ebp-4]
                                    cmp   dword [ebp-8], eax
                                    jle   lbl_endl
4         if (var[i] > 0) {
                                    mov   eax, dword [ebp-4]
                                    cmp   dword [ebp+eax*4-48], 0
                                    jle   lbl_else
5           var[i]++;
                                    mov   eax, dword [ebp-4]
                                    inc   dword [ebp+eax*4-48]
                                    jmp   lbl_endif
6        } else {
                               lbl_else:
7            var[i]--;
                                    mov   eax, dword [ebp-4]
                                    dec   dword [ebp+eax*4-48]
8        } /* endif */
                               lbl_endif:
                                    jmp   lbl_loop1
9     } /* endfor */
                               lbl_endl:
10 }
                                    pop   edi
                                    pop   esi
                                    pop   ebx
                                    leave
                                    ret
```

Interessant ist u.a., dass das Programm deutlich kürzer ist, als beim 8086. Auch wird hier die der LEAVE Instruktion vor dem Verlassen der Routine verwendet, die mit dem 80186 eingeführt wurde. Die ENTER Instruktion wurde hingegen nicht benutzt. Grund dafür war wohl, dass die ENTER Instruktion langsamer war, als einzelnen Befehle. LEAVE dagegen war gleich schnell, aber kürzer (weniger Maschinencode).

9 i486 und Pentium Prozessor Überblick

9.1 Überblick i486

Der i486 integriert eine verbesserte i386-CPU, einen gegenüber dem i387 leistungsfähigeren Coprozessor und einen Cache Controller samt 8kByte (Code- und Daten-) Cache auf einem Chip. Der i486 ist voll kompatibel zur Kombination i386/i387. Er kennt die dieselben Befehle und Datentypen, den Real Mode, Protected Mode und Virtual 8086 Mode und führt in gleicher Weise eine Segmentierung des Speichers und Paging aus. Selbst die Register des i486 sind die gleichen wie beim i386. Interne Unterschiede betreffen lediglich Flag- und Steuerregister-Erweiterungen, die hauptsächlich zur Cache-Unterstützung benötigt werden.

Der i486 ist vor allem wegen der Festverdrahtung häufig benutzter Befehle und einer Befehls-Pipeline etwa um den Faktor drei schneller als ein mit derselben Frequenz getakteter i386.

Die Befehls-Pipeline des i486 ist eine fünfstufige Pipeline und besteht aus den relativ lose gekoppelten Funktionseinheiten (im Gegensatz zu den stark gekoppelten Pipelines bei echten RISC-CPUs) **Instruction Fetch**, **Instruction Decode**, **Address Generation**, **Execution** und **Write Back**. Damit können die verschiedenen CPU-Einheiten in dem Sinne parallel arbeiten, dass sich z.B. ein Befehl bereits in der Ausführungsphase befindet, während der folgende noch dekodiert und der zweite folgende erst gelesen wird.

Für die Übertragung größerer Datenmengen wurde ein neuer Busmodus implementiert, der sogenannte "Burst-Modus". Normalerweise dauert ein Buszyklus zur Übertragung von Daten ohne Waitstates zwei Taktzyklen. Im Burst-Modus verringert sich die Zeit für die Übertragung eines Werts auf einen Taktzyklus, allerdings mit Einschränkungen: Maximal können in einem Burst-Zyklus 16 Bytes (4 x 32-Bits) eingelesen werden, die zudem alle in einem Bereich liegen müssen, der an einer 16-Byte-Grenze beginnt (also von xxxxxxx0h bis xxxxxxxFh). Das entspricht jedoch genau einer Zeile im internen Cache. Der Burst-Zyklus ist also besonders für das Füllen einer Cache-Zeile und das Einlesen des TSS bei einem Task Switch geeignet.

i486 Prozessorvarianten und Upgrading

Den i486 gab es für Taktfrequenzen von 25 bis 50MHz oder mit interner Taktverdopplung bis 66MHz.

Auch beim i486 hat Intel das SX-Konzept fortgesetzt und einen i486SX und den zugehörigen i487SX entwickelt. Beim i486SX fehlt nur die Gleitkommaeinheit auf dem Chip, die anderen Erweiterungen des i486DX gegenüber dem i386DX, wie z.B. On-Chip Cache, RISC-Core, etc., sind auch auf dem i486SX-Chip enthalten. Die Gleitkommafunktionen können mit dem "Coprozessor" i487 nachgerüstet werden. Interessanterweise steigert der i487 auch die Leistungsfähigkeit der CPU selbst - der i487SX ist nämlich nicht nur ein numerischer Coprozessor, sondern eine vollwertige i486-CPU, ist also mit einem i486DX vergleichbar und ersetzt die i486SX-CPU komplett. Das Nachrüsten eines i487SX zu einem i486SX nennt man auch "Upgrading". Der Upgrade-Prozessor teilt der bisherigen CPU über ein Hardware-Signal mit, dass er vorhanden ist, die alte CPU ist dadurch eigentlich nicht mehr notwendig und tritt in einen Power-Down-Modus ein.

Eine andere Möglichkeit der Leistungssteigerung ermöglichen die i486DX2 Prozessoren. Mit diesen CPUs begegnete Intel dem Problem, dass höher getaktete Motherboards auch aufwendiger und teurer sind. Die i486DX2 CPUs werden intern mit der doppelten Frequenz des Motherboards getaktet, ein i486DX2-66 wird intern mit 66MHz, das Motherboard mit 33MHz getaktet. Dadurch ergibt sich auch die Möglichkeit des Upgradings durch "Overdrives", wobei ein i486DX durch einen i486DX2 ersetzt wird.

9.2 Überblick Pentium Prozessor

Der Pentium Prozessor ist 100% binär kompatibel mit allen vorhergegangenen Prozessoren der 80x86 Serie. Der Pentium ist, wie seine Vorgänger i386 und i486, ein 32-Bit-Prozessor und kann physikalisch 4GByte adressieren. Software, die für den i386 oder i486 geschrieben wurde, läuft i.a. ohne Modifikationen auf dem Pentium Prozessor.

Die Register (Vielzweck-, Segment- und Speicherverwaltungsregister) wurden wie beim i486 implementiert, lediglich ein paar neue Bits in einigen Steuerregistern sind hinzugekommen. Auch die Floating-Point-Register sind noch die vom 8087 her bekannten. Sogar die Memory Management Unit (MMU) des Pentium Prozessors ist gleich der der i486-CPU.

Darüber hinaus enthält der Pentium Prozessor auch alle anderen Funktionen des i486 und bietet die folgenden Erweiterungen:

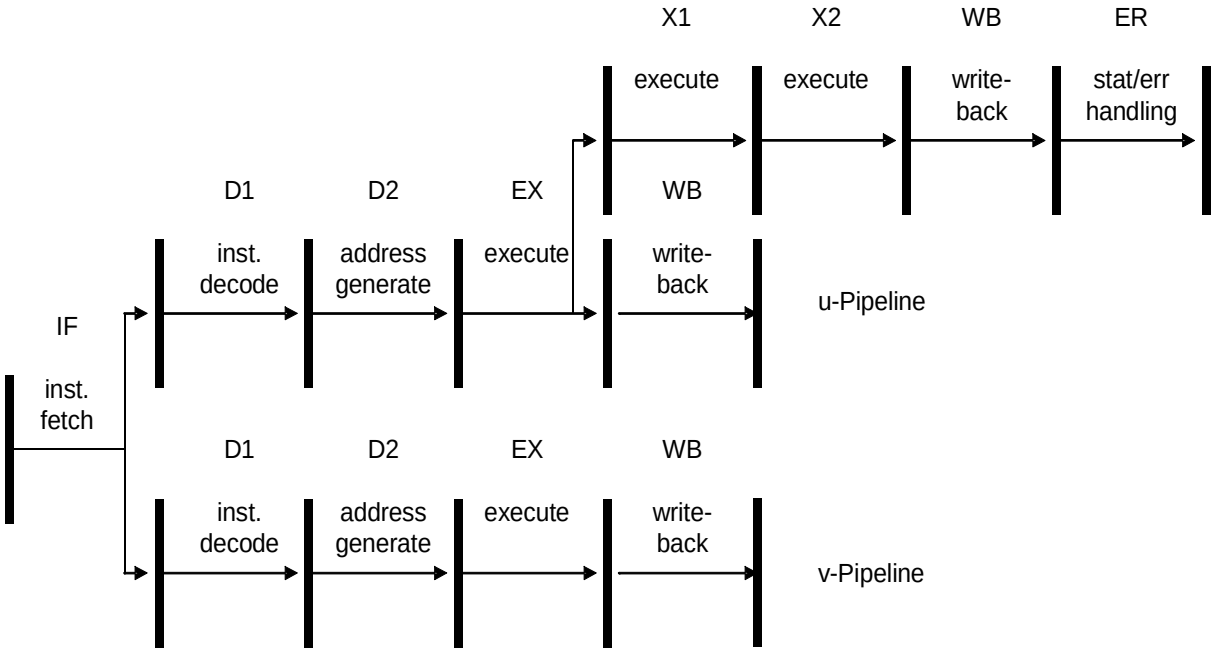
- Superscalar Architektur
- Dynamische Sprung-Vorhersage (Branch-Prediction)
- Floating point unit bis zu 5mal schneller als beim i486
- Neue Befehle CMPXCHG8B, CPUID, RDTSC, MOV CR4,r32, RDMSR, WRMSR, RSM
- Neue Flags: VIF, VIP und ID (Bits 19-21 in den Flags)
- Neue Register: CR4
- Verbesserte Befehls-Ausführungszeiten
- Separate 8k Code- und Daten-Caches
- Writeback MESI Kohärenz-Protokoll im Daten-Cache
- 64-Bit Daten-Bus
- Pipelining der Bus-Zyklen
- Interne Paritätsprüfung
- Erweiterte Debug-Unterstützung
- Execution Tracing
- Functional Redundancy Checking
- Performance Monitoring
- IEEE 1149.1 Boundary Scan
- System Management Mode
- Erweiterungen im Virtual 8086-Mode

Die Befehls-Pipelines und die getrennten Caches für Code und Daten sind Teile des klassischen RISC-Konzeptes. Auch die Hardware-Implementierung vieler Befehle und deren Ausführung in nur einem Taktzyklus sind RISC-Merkmale. Ansonsten ist auch der Pentium wie alle seine Vorgänger eher ein CISC-Prozessor, er stellt immerhin einen Befehlssatz von über 500 Befehlen zur Verfügung (incl. Gleitkommabefehle). Außerdem benötigen viele Befehle zur Ausführung mehr als einen Taktzyklus und sind auch oft mikro- codiert, was beides nur bei CISC-Prozessoren der Fall ist.

Die Pentium Pipelines

Der Pentium besitzt zwei 5-stufige Integer-Pipelines (die u- und die v-Pipeline) und eine 8-stufige Gleitkomma-Pipeline (wobei die ersten 4 Stufen der Gleitkomma-Pipeline mit denen der u-Pipeline identisch sind). Integer bezeichnet in diesem Zusammenhang alle Befehle, die keine Fließkommaoperationen beinhalten, also z.B. ADD-, CMP-, MOV- und JMP-Befehle. Der Pentium kann dadurch im Idealfall zwei Integer-Befehle in einem einzigen Takt abarbeiten, einen beliebigen

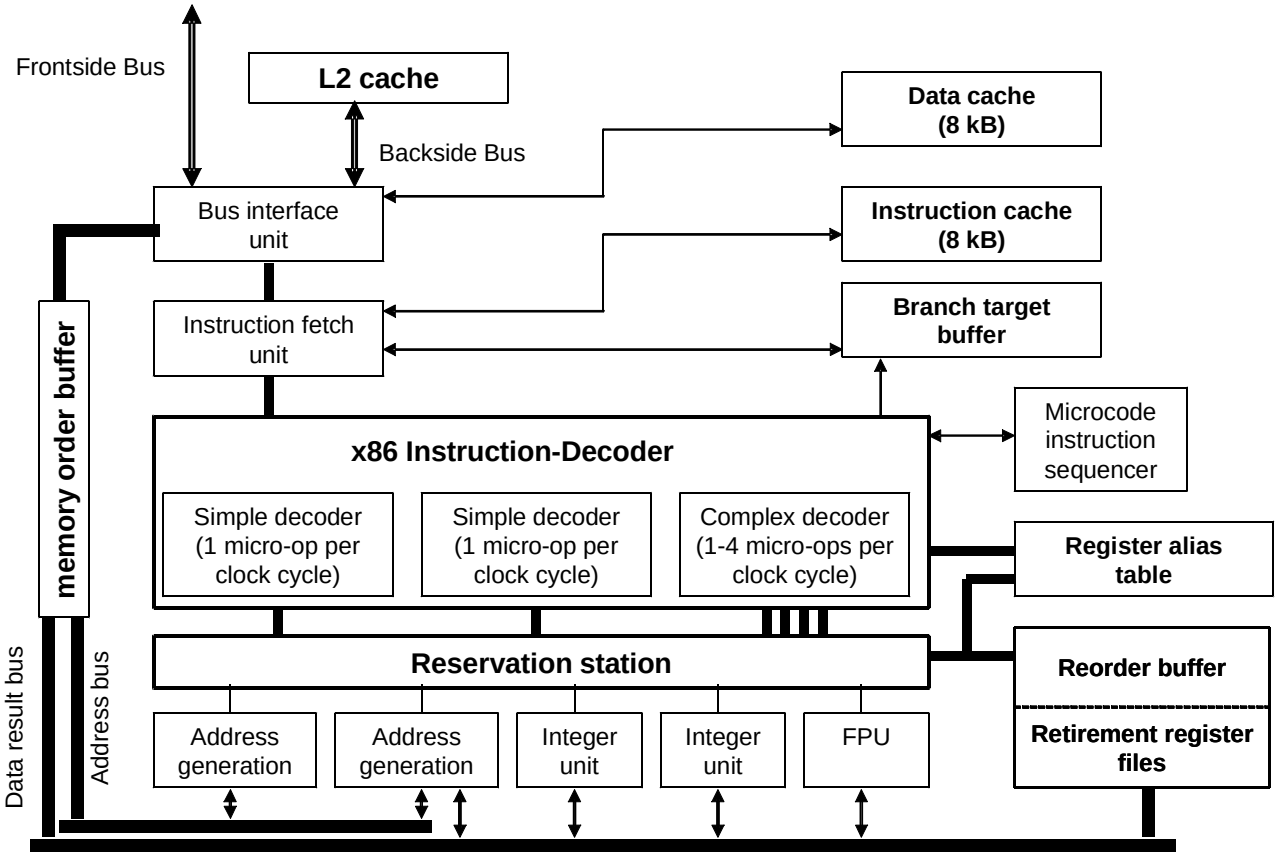
Befehl in der u-Pipeline und einen als einfach bezeichneten (nicht mikro- codierten) Befehl in der v-Pipeline. Damit kann theoretisch ungefähr die doppelte Performance verglichen mit einem i486 erreicht werden.



9.3 Überblick Pentium Pro (P6)

- Erhältlich als 150, 166, 180 und 200 MHz Version
- Optimiert für 32-Bit Anwendungen
- Drei-Wege Superscalar Architektur
- Bis zu 14-stufige Superpipeline
- "Out-of-order" Befehlsausführung
- dynamische Sprung-Vorhersage (Branch Prediction)
- Spekulative Befehlsausführung
- Aufbrechen der komplexen CISC-Befehle in **micro-ops**.
- 40 zusätzliche General-Purpose-Register
- 8-KB Zwei-Wege-Assoziativer Befehls-Cache; 8-KB Vier-Wege Primärer Daten-Cache
- Transaktions-orientierter I/O Bus und nicht-blockierende Cache-Hierarchie
- 387-Pin dual-cavity PGA package; 256-KB (oder 512-KB bei der 166 MHz Version) SRAM L2 Cache im gleichen Gehäuse
- 5.5 Millionen Transistoren im CPU Kern, 15.5 Millionen im Sekundär-Cache
- CPU Chip: 306mm^2 SRAM Chip: 202mm^2
- 2.9 V, 0.6µ BiCMOS Technologie
- 20 W bei 133 MHz

Die folgende Abbildung zeigt einen Überblick über den internen Aufbau des Pentium Pro:



10 Beispiele für Peripheriebausteine

Als Beispiele für Bausteine, die sich außer dem Prozessor in einem Rechner befinden, werden zwei Elemente des ursprünglichen IBM PC betrachtet. Auch aktuelle Chipsätze stellen diese Funktionalität so zur Verfügung (plus etliche Erweiterungen), so dass auch die Peripherie eines modernen PCs auf die gleiche Art programmiert werden kann.

Der **Timer 8254** wird u.a. für die Uhr eingesetzt. Im PC war der Vorläufer des 8254, ein 8253 eingebaut, der aber bis auf eine Erweiterung die gleichen Funktionen zur Verfügung stellt.

Der **Interrupt-Controller 8259A** empfängt z.B. Hardware-Anforderungen von Peripheriegeräten wie Platten oder Tastatur (siehe auch Interrupt 8 bis 15 in "BIOS Interrupts" auf Seite 79). Im IBM PC war ein 8259A eingebaut, seit PC AT stehen zwei zur Verfügung.

In diesem Kapitel werden einige Beispiele für die Programmierung dieser Bausteine gezeigt. Ausführliche Beschreibungen der Funktionen und weitere Details sind direkt aus den von Intel zur Verfügung gestellten Unterlagen zu entnehmen.

10.1 Timer 8254

Jeder Timer des Bausteins 8254 wird programmiert, indem ein Steuerwort und ein Zähleranfangswert ausgegeben wird.

Soll z.B. der Timer 1 im Mode 2 mit einem Anfangswert 1E01h im Binärmode programmiert werden, könnte dies wie folgt aussehen (die Basisadresse des 8254 soll 00B0h im I/O-Bereich des Prozessors sein):

```
MOV  AL, 01110100b     ; Laden AL mit Steuerwort:
                       ; Zähler 1, LSB+MSB, Mode 2, Binär
OUT  0B3h, AL          ; Ausgeben auf Adresse für Steuerwort

MOV  AL, 00000001b     ; Zählerregister LSB
OUT  0B1h, AL          ; Ausgeben auf Adresse Zähler 1
MOV  AL, 00011110b     ; Zählerregister MSB
OUT  0B1h, AL
```

Soll dann im laufenden Betrieb der Zähler ausgelesen werden, könnte dies mit einem Counter-Latch Command eingeleitet werden:

```
MOV  AL, 01000000b     ; Laden AL mit Steuerwort:
                       ; Zähler 1, Counter Latch
OUT  0B3h, AL          ; Ausgeben auf Adresse für Steuerwort

IN   AL, 0B1h          ; Einlesen Zählerregister LSB
...                    ; AL z.B. im Speicher ablegen
IN   AL, 0B1h          ; Einlesen Zählerregister MSB
...                    ; AL z.B. im Speicher ablegen
```

Verwendet man das Read-Back Command (das ist die erwähnte Funktion, die für den 8253 nicht zur Verfügung steht), könnte das Auslesen der Statusinformation aller Zähler folgendermaßen geschehen:

```
MOV  AL, 11101110b     ; Read-Back für Status; alle Zähler auswählen
OUT  0B3h, AL          ; Ausgeben auf Adresse für Steuerwort

IN   AL, 0B1h          ; Einlesen Status Zähler 1
...                    ; AL z.B. im Speicher ablegen
IN   AL, 0B0h          ; Einlesen Status Zähler 0
...                    ; AL z.B. im Speicher ablegen
IN   AL, 0B2h          ; Einlesen Status Zähler 2
...                    ; AL z.B. im Speicher ablegen
```

Die Reihenfolge, in der die Zähler ausgelesen werden, ist nicht relevant, da die Adresse jeweils eindeutig ist. Mit dem Read-Back Command ließe sich aber auch zusätzlich der Zählerinhalt auslesen:

```
MOV  AL, 11000010b     ; Read-Back für Status und Zähler; nur Zähler 0
OUT  0B3h, AL          ; Ausgeben auf Adresse für Steuerwort

IN   AL, 0B0h          ; Einlesen Status Zähler 0
...                    ; AL z.B. im Speicher ablegen
IN   AL, 0B0h          ; Einlesen LSB Zähler 0
...                    ; AL z.B. im Speicher ablegen
IN   AL, 0B0h          ; Einlesen MSB Zähler 0
...                    ; AL z.B. im Speicher ablegen
```

Hierbei muss beachtet werden, dass zunächst die Statusinformation, danach LSB und MSB des Zählers (in dieser Reihenfolge) gelesen wird.

10.2 Interrupt Controller 8259A

Zur Initialisierung eines 8259A müssen mindestens ein ICW1 und ein ICW2 (siehe Intel-Unterlagen), je nach Einsatz auch ein ICW3 oder ICW4 programmiert werden.

Soll z.B. ein Interrupt-Controller in einem gepuffert aufgebauten 8086-System so programmiert werden, dass Interrupts edge-triggered ausgelöst werden und eine Interrupt-Anforderung an IR0 des Controllers den Interrupt 128 im Prozessor auslöst, könnte dies folgendermaßen geschehen (die Basisadresse des Interrupt-Controllers soll 0380h im I/O-Bereich des Prozessors sein):

```
MOV  AL, 00010011b     ; ICW1
MOV  DX, 0380h         ; Ausgabe über DX (Adresse > 255)
OUT  DX, AL            ; (A0 = 0)

MOV  AL, 10000000b     ; ICW2
INC  DX                ; (A0 = 1)
OUT  DX, AL

MOV  AL, 00001101b     ; ICW4
OUT  DX, AL
```

Ein ICW3 wird in diesem Fall nicht benötigt. Soll für diesen Controller ein Interrupt maskiert werden, könnte dies so aussehen:

```
MOV  DX, 0381h         ; Ausgabe über DX (Adresse > 255)
                       ; (A0 = 1)
IN   AL, DX
OR   AL, 00001000b     ; Maske für IR3 setzen
OUT  DX, AL
```

Zunächst muss die Maske gelesen werden, da sonst alle Interrupts beeinflusst werden würden. Ein Freigeben würde entsprechend aussehen:

```
IN   AL, DX
AND  AL, 11110111b     ; IR3 freigeben
OUT  DX, AL
```

Wird eine Interrupt-Routine durch einen Hardware-Interrupt über einen 8259A aufgerufen, so ist es wichtig, dass vor Verlassen der Routine das ISR Bit im Interrupt-Controller rückgesetzt wird:

```
int_fkt:
        PUSH AX
        PUSH DX

        ...

        MOV  DX, 0380h
        MOV  AL, 00100000b     ; normales EOI
        OUT  DX, AL

        POP  DX
        POP  AX
        IRET
```

11 8086 Befehlssatz

In der ersten Tabelle dieses Abschnitts sind alle Befehle des 8086 in alphabetischer Reihenfolge aufgelistet. Die Tabelle enthält für jeden Befehl die möglichen Ausprägungen, den Maschinencode und eine kurze Erklärung. Zum Verständnis der mittleren Spalte siehe auch Abschnitt "Aufbau eines Maschinenbefehls" auf Seite 25. Die in der Tabelle verwendeten Symbole wie "r/m" oder "disp8" sind in der anschließenden zweiten Tabelle erklärt. Mit (*) markierte Mnemonics sind keine Befehle, sondern Präfixe.

Mnemonic	***Maschinencode***	***Beschreibung***
AAA	*37*	Passe AX nach ASCII-Addition an
AAD	*D5, 0A*	Passe AX nach ASCII-Division an
AAM	*D4, 0A*	Passe AX nach ASCII-Multiplikation an
AAS	*3F*	Passe AX nach ASCII-Subtraktion an
ADC AL/AX,data	*0001010w, data*	Addiere 'data' und Carry zu AL/AX
ADC r/m,data	*100000sw, mod 010 r/m [,disp], data*	Addiere 'data' und Carry zu 'r/m'
ADC r/m1,r/m2	*000100dw, mod reg r/m [,disp]*	Addiere 'r/m2' und Carry zu 'r/m1'
ADD AL/AX,data	*0000010w, data*	Addiere 'data' zu AL/AX
ADD r/m,data	*100000sw, mod 000 r/m [,disp], data*	Addiere 'data' zu 'r/m'
ADD r/m1,r/m2	*000000dw, mod reg r/m [,disp]*	Addiere 'r/m2' zu 'r/m1'
AND AL/AX,data	*0010010w, data*	Verknüpfe 'data' mit AL/AX über die logische AND-Funktion
AND r/m,data	*1000000w, mod 100 r/m [,disp], data*	Verknüpfe 'data' mit 'r/m' über die logische AND-Funktion
AND r/m1,r/m2	*001000dw, mod reg r/m [,disp]*	Verknüpfe 'r/m2' mit 'r/m1' über die logische AND-Funktion
CALL address	*9A, addr1, addr2*	Rufe Unterprogramm ab addr1 (CS=addr2) auf (Unterprogramm ist mit PROC FAR definiert)
CALL disp16	*E8, disp16*	Rufe Unterprogramm ab 'naddr+disp16' auf (Unterprogramm ist mit PROC NEAR definiert)
CALL mem	*FF, mod 011 r/m [,disp]*	Rufe Unterprogramm ab 'mem' (CS=mem+2) auf
CALL r/m16	*FF, mod 010 r/m [,disp]*	Rufe Unterprogramm ab 'r/m16' auf (r/m16 ist mit DW definiert)
CBW	*98*	Wandle Byte in AL in ein Wort für AX um
CLC	*F8*	Lösche Übertrag-Statusbit (Carry)
CLD	*FC*	Lösche Richtungs-Flag (Direction-Flag)
CLI	*FA*	Lösche Interrupt-Enable-Flag
CMC	*F5*	Invertiere Übertrag-Statusbit (Carry)
CMP AL/AX,data	*0011110w, data*	Vergleiche 'data' mit AL/AX
CMP r/m,data	*100000sw, mod 111 r/m [,disp], data*	Vergleiche 'data' mit 'r/m'
CMP r/m1,r/m2	*001110dw, mod reg r/m [,disp]*	Vergleiche 'r/m2' mit 'r/m1'
CMPS	*1010011w*	Vergleiche String ab SI mit String ab DI
CMPSB	*A6*	Vergleiche String ab SI byteweise mit String ab DI

Mnemonic	*Maschinencode*	*Beschreibung*
CMPSW	*A7*	Vergleiche String ab SI wortweise mit String ab DI
CWD	*99*	Wandle Wort in AX in ein Doppelwort für DX (MSB) und AX (LSB) um
DAA	*27*	Passe AL nach dezimaler Addition an
DAS	*2F*	Passe AL nach dezimaler Subtraktion an
DEC reg16	*01001reg*	Vermindere Register 'reg16' um 1
DEC r/m	*1111111w, mod 001 r/m [,disp]*	Vermindere 'r/m' um 1
DIV r/m	*1111011w, mod 110 r/m [,disp]*	Dividiere DX,AX bzw. AX durch 'r/m' (vorzeichenlos)
ESC	*11011xxx, mod xxx r/m [,disp]*	Übergebe 'r/m' an externe Einheit
HLT	*F4*	Halte Prozessor an (kann durch Interrupt wieder gelöst werden)
IDIV r/m	*1111011w, mod 111 r/m [,disp]*	Dividiere DX,AX bzw. AX durch 'r/m' (vorzeichenbehaftet)
IMUL r/m	*1111011w, mod 101 r/m [,disp]*	Multipliziere AL/AX mit 'r/m' (vorzeichenbehaftet)
IN AL/AX,DX	*1110110w*	Lese Wert über Port [DX] in AL/AX
IN AL/AX,port	*1110010w*	Lese Wert über Port 'port' in AL/AX ('port' < 256)
INC reg16	*01000reg*	Erhöhe Register 'reg16' um 1
INC r/m	*1111111w, mod 000 r/m [,disp]*	Erhöhe 'r/m' um 1
INT data8	*CD, data8*	Führe Software-Interrupt Nummer 'data8' aus
INT 3	*CC*	Führe Software-Interrupt Nummer 3 aus (Ein-Byte-Interrupt)
INTO	*CE*	Führe Software-Interrupt Nummer 4 aus, wenn Überlauf-Statusbit gesetzt wurde (Overflow-Interrupt)
IRET	*CF*	Kehre von Interruptroutine zurück
JA disp8	*77, disp8*	Verzweige zu 'naddr+disp8', wenn darüber (C=0 und Z=0)
JAE disp8	*73, disp8*	Verzweige zu 'naddr+disp8', wenn darüber oder gleich (C=0)
JB disp8	*72, disp8*	Verzweige zu 'naddr+disp8', wenn darunter (C=1)
JBE disp8	*76, disp8*	Verzweige zu 'naddr+disp8', wenn darunter oder gleich (C=1 oder Z=1)
JC		gleich wie JB
JCXZ disp8	*E3, disp8*	Verzweige zu 'naddr+disp8', wenn Inhalt von CX gleich 0
JE disp8	*74, disp8*	Verzweige zu 'naddr+disp8', wenn gleich (Z=1)
JG disp8	*7F, disp8*	Verzweige zu 'naddr+disp8', wenn größer (Z=0 und S=0)
JGE disp8	*7D, disp8*	Verzweige zu 'naddr+disp8', wenn größer oder gleich (S=0)
JL disp8	*7C, disp8*	Verzweige zu 'naddr+disp8', wenn kleiner (S!=0)
JLE disp8	*7E, disp8*	Verzweige zu 'naddr+disp8', wenn kleiner oder gleich (Z=0 oder S!=0)
JMP address	*EA, addr1, addr2*	Verzweige zu addr2:addr1 (CS=addr2, offset=addr1)

Mnemonic	*Maschinencode*	*Beschreibung*
JMP disp8	*EB, disp8*	Verzweige zu 'naddr+disp8'
JMP disp16	*E9, disp16*	Verzweige zu 'naddr+disp16'
JMP mem	*FF, mod 101 r/m [,disp]*	Verzweige zu 'mem' (CS=mem+2)
JMP r/m16	*FF, mod 000 r/m [,disp]*	Verzweige zu 'naddr+r/m16'
JNA disp8	*76, disp8*	Verzweige zu 'naddr+disp8', wenn nicht darüber (C=1 oder Z=1)
JNAE disp8	*72, disp8*	Verzweige zu 'naddr+disp8', wenn nicht darüber und nicht gleich (C=1)
JNB disp8	*73, disp8*	Verzweige zu 'naddr+disp8', wenn nicht darunter (C=0)
JNBE disp8	*77, disp8*	Verzweige zu 'naddr+disp8', wenn nicht darunter und nicht gleich (C=0 und Z=0)
JNC		gleich wie JAE
JNE disp8	*75, disp8*	Verzweige zu 'naddr+disp8', wenn nicht gleich (Z=0)
JNG disp8	*7E, disp8*	Verzweige zu 'naddr+disp8', wenn nicht größer (Z=1 oder O!=S)
JNGE disp8	*7C, disp8*	Verzweige zu 'naddr+disp8', wenn nicht größer und nicht gleich (O!=S)
JNL disp8	*7D, disp8*	Verzweige zu 'naddr+disp8', wenn nicht kleiner (O=S)
JNLE disp8	*7F, disp8*	Verzweige zu 'naddr+disp8', wenn nicht kleiner und nicht gleich (S=0 und Z=0)
JNO disp8	*71, disp8*	Verzweige zu 'naddr+disp8', wenn kein Überlauf (O=0)
JNP disp8	*7B, disp8*	Verzweige zu 'naddr+disp8', wenn keine Parität (P=0)
JNS disp8	*79, disp8*	Verzweige zu 'naddr+disp8', wenn positiv (S=0)
JNZ disp8	*75, disp8*	Verzweige zu 'naddr+disp8', wenn nicht Null (Z=0)
JO disp8	*70, disp8*	Verzweige zu 'naddr+disp8', wenn Überlauf (O=1)
JP disp8	*7A, disp8*	Verzweige zu 'naddr+disp8', wenn Parität (P=1)
JPE disp8	*7A, disp8*	Verzweige zu 'naddr+disp8', wenn Parität gerade (P=1)
JPO disp8	*7B, disp8*	Verzweige zu 'naddr+disp8', wenn Parität ungerade (P=0)
JS disp8	*78, disp8*	Verzweige zu 'naddr+disp8', wenn negativ (S=1)
JZ disp8	*74, disp8*	Verzweige zu 'naddr+disp8', wenn Null (Z=1)
LAHF	*9F*	Lade die unteren 8 Statusbit in AH
LDS reg16,mem	*C5, mod reg r/m [,disp]*	Lade 'reg16' mit 'mem' und DS mit 'mem'+2
LEA reg16,addr	*8D, mod reg r/m [,disp]*	Lade 'addr' (effektive Adresse einer Variablen) in 'reg16'
LES reg16,mem	*C4, mod reg r/m [,disp]*	Lade 'reg16' mit 'mem' und ES mit 'mem'+2
LOCK *	*F0*	Aktiviere den LOCK-Ausgang des Prozessors für den nachfolgenden Befehl (LOCK prefix)
LODS	*1010110w*	Lade String ab SI in AL/AX
LODSB	*AC*	Lade String ab SI byteweise in AL/AX

Mnemonic	*Maschinencode*	*Beschreibung*
LODSW	*AD*	Lade String ab SI wortweise in AL/AX
LOOP disp8	*E2, disp8*	Wiederhole Schleife bis CX=0 (CX wird jeweils um 1 dekrementiert)
LOOPE disp8	*E1, disp8*	Wiederhole Schleife bis CX=0 oder Z=0 (CX wird jeweils um 1 dekrementiert)
LOOPNE disp8	*E0, disp8*	Wiederhole Schleife bis CX=0 oder Z=1 (CX wird jeweils um 1 dekrementiert)
LOOPNZ disp8	*E0, disp8*	Wiederhole Schleife bis CX=0 oder Z=1 (CX wird jeweils um 1 dekrementiert)
LOOPZ disp8	*E1, disp8*	Wiederhole Schleife bis CX=0 oder Z=0 (CX wird jeweils um 1 dekrementiert)
MOV reg,data	*1011wreg, data*	Übertrage 'data' in reg
MOV r/m,data	*1100011w, mod 000 r/m [,disp], data*	Übertrage 'data' in 'r/m'
MOV AL/AX,addr	*1010000w, addr*	Übertrage Inhalt der durch 'addr' adressierten Speicherstelle in AL/AX
MOV addr,AL/AX	*1010001w, addr*	Übertrage AL/AX in die durch 'addr' adressierte Speicherstelle
MOV r/m1,r/m2	*100010dw, mod reg r/m [,disp]*	Übertrage 'r/m2' in 'r/m1'
MOV sr,r/m16	*8E, mod 0sr r/m [,disp]*	Übertrage 'r/m16' in sr (jedoch nicht für sr=01)
MOV r/m16,sr	*8C, mod 0sr r/m [,disp]*	Übertrage sr in 'r/m16'
MOVS	*1010010w*	Übertrage String ab SI nach DI
MOVSB	*A4*	Übertrage String ab SI byteweise nach DI
MOVSW	*A5*	Übertrage String ab SI wortweise nach DI
MUL r/m	*1111011w, mod 100 r/m [,disp]*	Multipliziere AL/AX mit 'r/m' (vorzeichenlos)
NEG r/m	*1111011w, mod 011 r/m [,disp]*	Negiere Wert von 'r/m' (bilde 2er-Komplement)
NOP	*90*	No Operation
NOT r/m	*1111011w, mod 010 r/m [,disp]*	Invertiere Wert von 'r/m' (bilde 1er-Komplement)
OR AL/AX,data	*0000110w, data*	Verknüpfe 'data' mit AL/AX über die logische OR-Funktion
OR r/m,data	*1000000w, mod 001 r/m [,disp], data*	Verknüpfe 'data' mit 'r/m' über die logische OR-Funktion
OR r/m1,r/m2	*000010dw, mod reg r/m [,disp]*	Verknüpfe 'r/m2' mit 'r/m1' über die logische OR-Funktion
OUT DX,AL/AX	*1110111w*	Gebe AL/AX über Port [DX] aus
OUT port,AL/AX	*1110011w, port*	Gebe AL/AX über Port "port" aus
POP reg16	*01011reg*	Lese in Register 'reg16' ein Wort vom Stack
POP sr	*000sr111*	Lese in Segmentregister sr ein Wort vom Stack
POP r/m16	*8F, mod 000 r/m [,disp]*	Lese in 'r/m16' ein Wort vom Stack
POPF	*9D*	Lese Statusbits vom Stack
PUSH reg16	*01010reg*	Lege Register 'reg16' auf den Stack
PUSH sr	*000sr110*	Lege Segmentregister sr auf den Stack
PUSH r/m16	*FF, mod 110 r/m [,disp]*	Lege 'r/m16' auf den Stack
PUSHF	*9C*	Lege Statusbits auf den Stack
RCL r/m,1 .br RCL	*110100cw, mod 010 r/m [,disp]*	Rotiere 'r/m' um 1 (oder CL) nach links und durch

Mnemonic	*Maschinencode*	*Beschreibung*
r/m,CL		das Carry-Flag
RCR r/m,1 .br **RCR r/m,CL**	*110100cw, mod 011 r/m [,disp]*	Rotiere 'r/m' um 1 (oder CL) nach rechts und durch das Carry-Flag
REP *	*F3*	Wiederhole String-Operation bis CX=0 (CX wird jeweils um 1 dekrementiert)
REPE *	*F3 (nur vor CMPS/SCAS)*	gleich wie REPZ
REPNE *	*F2 (nur vor CMPS/SCAS)*	gleich wie REPNZ
REPNZ *	*F2 (nur vor CMPS/SCAS)*	Wiederhole String-Operation bis CX=0 oder Z=1 (CX wird jeweils um 1 dekrementiert)
REPZ *	*F3 (nur vor CMPS/SCAS)*	Wiederhole String-Operation bis CX=0 oder Z=0 (CX wird jeweils um 1 dekrementiert)
RET	*CB*	Kehre aus Unterprogramm zurück, lese IP und CS vom Stack
RET	*C3*	Kehre aus Unterprogramm zurück, lese nur IP vom Stack
RET n	*CA, n*	Kehre aus Unterprogramm zurück, lese IP und CS vom Stack und addiere n zu SP
RET n	*C2, n*	Kehre aus Unterprogramm zurück, lese nur IP vom Stack und addiere n zu SP
ROL r/m,1 **ROL r/m,CL**	*110100cw, mod 000 r/m [,disp]*	Rotiere 'r/m' um 1 (oder CL) nach links
ROR r/m,1 **ROR r/m,CL**	*110100cw, mod 001 r/m [,disp]*	Rotiere 'r/m' um 1 (oder CL) nach rechts
SAHF	*9E*	Lade AH in die unteren 8 Statusbits
SAL r/m,1 **SAL r/m,CL**	*110100cw, mod 100 r/m [,disp]*	Schiebe 'r/m' um 1 (oder CL) arithmetisch nach links (entspricht SHL)
SAR r/m,1 **SAR r/m,CL**	*110100cw, mod 100 r/m [,disp]*	Schiebe 'r/m' um 1 (oder CL) arithmetisch nach rechts (entspricht SHR)
SBB AL/AX,data	*0001110w, data*	Subtrahiere 'data' und das Carry-Flag von AL/AX
SBB r/m,data	*100000sw, mod 011 r/m [,disp], data*	Subtrahiere 'data' und das Carry-Flag von 'r/m'
SBB r/m1,r/m2	*000110dw, mod reg r/m [,disp]*	Subtrahiere 'r/m2' und das Carry-Flag von 'r/m1'
SCAS	*1010111w*	Durchsuche String ab DI nach AL/AX
SCASB	*AE*	Durchsuche String ab DI byteweise nach AL/AX
SCASW	*AF*	Durchsuche String ab DI wortweise nach AL/AX
SEG: *	*001sr110*	Definiere sr als gültiges Segmentregister für den nächsten Befehl ("Segment override prefix")
SHL r/m,1 **SHL r/m,CL**		gleich wie SAL
SHR r/m,1 **SHR r/m,CL**		gleich wie SAR
STC	*F9*	Setze Carry-Flag
STD	*FD*	Setze Direction-Flag
STI	*FB*	Setze Interrupt-Flag
STOS	*1010101w*	Speichere AL/AX in String ab DI
STOSB	*AA*	Speichere AL byteweise in String ab DI

Mnemonic	*Maschinencode*	*Beschreibung*
STOSW	*AB*	Speichere AX wortweise in String ab DI
SUB AL/AX,data	*0010110w, data*	Subtrahiere 'data' von AL/AX
SUB r/m,data	*100000sw, mod 101 r/m [,disp], data*	Subtrahiere 'data' von 'r/m'
SUB r/m1,r/m2	*001010dw, mod reg r/m [,disp]*	Subtrahiere 'r/m2' von 'r/m1'
TEST AL/AX,data	*1010100w, data*	Vergleiche 'data' mit AL/AX über die AND-Funktion
TEST r/m,data	*1111011w, mod 000 r/m [,disp], data*	Vergleiche 'data' mit 'r/m' über die AND-Funktion
TEST r/m1,r/m2	*1000010w, mod reg r/m [,disp]*	Vergleiche 'r/m2' mit 'r/m1' über die AND-Funktion
WAIT	*9B*	Warte auf aktives Signal am TEST-Eingang des Prozessors
XCHG AX,reg16	*10010reg*	Tausche Register 'reg16' gegen AX aus
XCHG r/m1,r/m2	*1000011w, mod reg r/m [,disp]*	Tausche 'r/m2' gegen 'r/m1' aus
XLAT	*D7*	Übertrage den Inhalt der durch DS:[AL+BX] adressierten Speicherstelle in AL
XOR AL/AX,data	*0011010w, data*	Verknüpfe 'data' mit AL/AX über die logische XOR-Funktion
XOR r/m,data	*1000000w, mod 110 r/m [,disp], data*	Verknüpfe 'data' mit 'r/m' über die logische XOR-Funktion
XOR r/m1,r/m2	*001100dw, mod reg r/m [,disp]*	Verknüpfe 'r/m2' mit 'r/m1' über die logische XOR-Funktion

Parameter für die Tabelle der Assemblerbefehle:

Parameter	***Bedeutung***
addr	16-Bit-Adresse einer Speicherzelle
address	20-Bit-Adresse einer Speicherzelle, die sich aus einer 16-Bit-Adresse und einer nachfolgenden 16-Bit-Segmentbasis zusammensetzt
AL/AX	mit w=0: Register AL angesprochen mit w=1: Register AX angesprochen
c	c=0: Operand um eine Bitposition verschieben c=1: Operand um die in CL angegebene Anzahl von Positionen verschieben
d	"destination" d=0: Ergebnis steht im Speicher (im durch 'mod' und 'r/m' adressierten Operanden) d=1: Ergebnis steht im Register (das durch den Code 'reg' bezeichnete Register)
data	8-Bit oder 16-Bit Datenwert
data8	8-Bit Datenwert
disp	8-Bit-Adresse oder 16-Bit-Adresse, die zu der dem aktuellen Befehl folgenden Adresse naddr addiert wird
disp16	16-Bit-Adresse, die zu der dem aktuellen Befehl folgenden Adresse naddr addiert wird
mod	Code für eine Adressierungsart (siehe Aufbau eines Maschinenbefehls auf Seite 25)
mem	Inhalt der durch 'addr' oder 'disp' adressierten Speicherstelle(n) (mem ist mit DD definiert)
naddr	Nächste gültige Adresse hinter dem aktuellen Befehl
port	8-Bit-Adresse eines Ports
reg	Code für ein Register (siehe Aufbau eines Maschinenbefehls auf Seite 25)
r/m	Register (8- oder 16-Bit) oder Speicherinhalt (mit DB oder DW definiert)
r/m16	Register (16-Bit gefordert) oder Speicherinhalt (mit DW definiert)
r/m1, r/m2	Register oder Speicherinhalt; mindestens einer der beiden Operanden muß in einem Register stehen.
sr	Code für ein Segment-Register (CS=01, SS=10, DS=11, ES=00)
s	sign extension; die Kombination mit dem Bit w ergibt: sw=00: 'data' ist ein 8-Bit-Operand sw=01: 'data' ist ein 16-Bit-Operand sw=10: 'data' ist ein 8-Bit-Operand (s hat keine Wirkung) sw=11: 'data' ist ein 8-Bit-Operand, der auf 16-Bit erweitert wird
w	"word" w=0: Bytebefehl w=1: Wortbefehl
x	"undefiniert" Wahlweise gesetztes oder gelöschtes Bit

12 BIOS Interrupts

```
          +----------------+
          | BIOS DATA AREA |
          +----------------+

ADDR.  SIZE    DESCRIPTION

40:00  WORD    COM1 PORT ADDRESS
40:02  WORD    COM2 PORT ADDRESS
40:04  WORD    COM3 PORT ADDRESS
40:06  WORD    COM4 PORT ADDRESS
40:08  WORD    LPT1 PORT ADDRESS
40:0A  WORD    LPT2 PORT ADDRESS
40:0C  WORD    LPT3 PORT ADDRESS
40:0E  WORD    LPT4 PORT ADDRESS
40:10  WORD    EQUIPMENT FLAG (SEE INT 11)
40:13  WORD    MEMORY SIZE IN KBYTES
40:17  BYTE    KEYBOARD FLAG BYTE 0 (SEE INT 9)
40:18  BYTE    KEYBOARD FLAG BYTE 1 (SEE INT 9)
40:19  BYTE    STORAGE FOR ALTERNATE KEYPAD ENTRY
40:1A  WORD    POINTER TO KEYBOARD BUFFER HEAD
40:1C  WORD    POINTER TO KEYBOARD BUFFER TAIL
40:1E  20BYTES KEYBOARD BUFFER
40:3E  BYTE    DRIVE RECALIBRATION STATUS
40:3F  BYTE    MOTOR STATUS
40:3E  BYTE    DRIVE RECALIBRATION STATUS
40:3F  BYTE    MOTOR STATUS
40:40  BYTE    MOTOR OFF COUNTER (DECR. BY TIMER)
40:41  BYTE    STATUS OF LAST DISKETTE OPERATION
40:42  7 BYTES NEC STATUS
40:49  BYTE    CURRENT CRT MODE
40:4A  WORD    NUMBER OF COLUMNS ON SCREEN
40:4C  WORD    REGEN BUFFER LENGTH IN BYTES
40:4E  WORD    STARTING OFFSET OF REGEN BUFFER
40:50  8 WORDS CURSOR POSITION PAGES 1-8
40:60  BYTE    END LINE FOR CURSOR
40:61  BYTE    START LINE FOR CURSOR
40:62  BYTE    CURRENT PAGE BEING DISPLAYED
40:63  WORD    BASE PORT ADDRESS FOR ACTIVE DISPLAY
40:65  BYTE    CURRENT SETTING OF THE 3X8 REGISTER
40:66  BYTE    CURRENT PALETTE SETTING COLOR CARD
40:67  DWORD   TEMP. STORAGE FOR SS:SP DURING SHUTDOWN
40:6C  DWORD   TIMER COUNTER LOW WORD, HIGH WORD
40:70  BYTE    24 HOUR TIMER OVERFLOW
40:71  BYTE    BIOS BREAK FLAG (BIT 7 = BREAK KEY HIT)
40:72  WORD    RESET FLAG (1234 = SOFT RESET)
40:74  BYTE    STATUS OF LAST HARD DISK OPERATION
40:75  BYTE    NUMBER OF HARD FILES ATTACHED
40:77  BYTE    PORT OFFSET TO CURRENT HF ADAPTER
40:78  4 BYTES TIMEOUT VALUE FOR LPT1,LPT2,LPT3,LPT4
40:7C  4 BYTES TIMEOUT VALUE FOR COM1,COM2,COM3,COM4
40:80  WORD    KEYBOARD BUFFER START OFFSET (SEG=40)
40:82  WORD    KEYBOARD BUFFER END OFFSET (SEG=40H)
40:84  BYTE    ROWS ON THE SCREEN (EGA ONLY)
40:85  WORD    BYTES PER CHARACTER (EGA ONLY)
40:87  BYTE    MODE OPTIONS (EGA ONLY)
40:88  BYTE    FEATURE BIT SWITCHES (EGA ONLY)
40:8B  BYTE    LAST DISKETTE DATA RATE SELECTED
40:8C  BYTE    HARD FILE STATUS RETURNED BY CONTROLLER
40:8D  BYTE    HARD FILE ERROR RETURNED BY CONTROLLER
40:8E  BYTE    HARD FILE INTERRUPT (BIT 7=WORKING INT)
40:90  4 BYTES MEDIA STATE DRIVE 0,1,2,3
40:94  2 BYTES TRACK CURRENTLY SEEKED TO DRIVE 0,1
40:96  BYTE    KEYBOARD FLAG BYTE 3 (SEE INT 9)
40:97  BYTE    KEYBOARD FLAG BYTE 2 (SEE INT 9)
40:98  DWORD   POINTER TO USERS WAIT FLAG
40:9C  DWORD   USERS TIMEOUT VALUE IN MICROSECONDS
40:A0  BYTE    RTC WAIT FUNCTION IN USE
40:A1  BYTE    LANA DMA CHANNEL FLAGS
40:A2  2 BYTES STATUS LANA 0,1
40:A4  DWORD   SAVED HARDFILE INTERRUPT VECTOR
40:A8  DWORD   EGA POINTER TO PARAMETER TABLE
40:B4  BYTE    KEYBOARD NMI CONTROL FLAGS (CONVERTIBLE)
40:B5  DWORD   KEYBOARD BREAK PENDING FLAGS (CONVERTIBLE)
40:B9  BYTE    PORT 60 SINGLE BYTE QUEUE (CONVERTIBLE)
40:BA  BYTE    SCAN CODE OF LAST KEY (CONVERTIBLE)
40:BB  BYTE    POINTER TO NMI BUFFER HEAD (CONVERTIBLE)
40:BC  BYTE    POINTER TO NMI BUFFER TAIL (CONVERTIBLE)
40:BD  16BYTES NMI SCAN CODE BUFFER (CONVERTIBLE)
40:CE  WORD    DAY COUNTER (CONVERTIBLE AND AFTER)
50:00  BYTE    PRINT SCREEN STATUS BYTE
```

```
          +-----------------------+
          | INT 05 - PRINT SCREEN |
          +-----------------------+

 INPUT PARAMETERS:   NONE
OUTPUT PARAMETERS:   NONE
          MEMORY:

50:0 = 00 - PRINT SCREEN HAS NOT BEEN CALLED, OR
            UPON RETURN FROM A CALL THERE WERE NO
            ERRORS.

     = 01 - PRINT SCREEN IS ALREADY IN PROGRESS.

     = FF - ERROR ENCOUNTERED DURING PRINTING.


          +-----------------------+
          | INT 08 - SYSTEM TIMER |
          +-----------------------+

 INPUT PARAMETERS:   NONE
OUTPUT PARAMETERS:   NONE
          MEMORY:

40:6C = NUMBER OF INTERRUPTS SINCE POWER ON (4 BYTES)
40:70 = NUMBER OF DAYS SINCE POWER ON (1 BYTE)
40:67 = DAY COUNTER ON ALL PRODUCTS AFTER PC/AT
40:40 = MOTOR CONTROL COUNT - GETS DECREMENTED
        AND SHUTS OFF DISKETTE MOTOR IF ZERO


          +----------------------------+
          | INT 09 - KEYBOARD INTERRUPT |
          +----------------------------+

 INPUT PARAMETERS:   NONE
OUTPUT PARAMETERS:   NONE
          MEMORY:

 +-+-+-+-+-+-+-+-+
 |7|6|5|4|3|2|1|0| 40:17
 +-+-+-+-+-+-+-+-+
  | | | | | | | +----- RIGHT SHIFT KEY DEPRESSED
  | | | | | | +------- LEFT SHIFT KEY DEPRESSED
  | | | | | +--------- CONTROL SHIFT KEY DEPRESSED
  | | | | +----------- ALTERNATE SHIFT KEY DEPRESSED
  | | | +------------- SCROLL LOCK STATE HAS BEEN TOGGLED
  | | +--------------- NUM LOCK STATE HAS BEEN TOGGLED
  | +----------------- CAPS LOCK STATE HAS BEEN TOGGLED
  +------------------- INSERT STATE IS ACTIVE

 +-+-+-+-+-+-+-+-+
 |7|6|5|4|3|2|1|0| 40:18
 +-+-+-+-+-+-+-+-+
  | | | | | | | +----- LEFT CONTROL KEY DEPRESSED
  | | | | | | +------- LEFT ALT SHIFT KEY DEPRESSED
  | | | | | +--------- SYSTEM KEY DEPRESSED AND HELD
  | | | | +----------- SUSPEND KEY HAS BEEN TOGGLED
  | | | +------------- SCROLL LOCK KEY IS DEPRESSED
  | | +--------------- NUM LOCK KEY IS DEPRESSED
  | +----------------- CAPS LOCK KEY IS DEPRESSED
  +------------------- INSERT KEY IS DEPRESSED

 +-+-+-+-+-+-+-+-+
 |7|6|5|4|3|2|1|0| 40:96
 +-+-+-+-+-+-+-+-+
  | | | | | | | +----- LAST CODE WAS THE E1 HIDDEN CODE
  | | | | | | +------- LAST CODE WAS THE E0 HIDDEN CODE
  | | | | | +--------- RIGHT CTL KEY DOWN
  | | | | +----------- RIGHT ALT KEY DOWN
  | | | +------------- ALT GRAPHICS KEY DOWN (WT ONLY)
  | | +--------------- ENHANCED KEYBOARD INSTALLED
  | +----------------- FORCE NUM LOCK IF RD ID & KBX
  +------------------- LAST CHAR WAS FIRST ID CHAR
                       DOING A READ ID (MUST BE BIT0)

 +-+-+-+-+-+-+-+-+
 |7|6|5|4|3|2|1|0| 40:97
 +-+-+-+-+-+-+-+-+
  | | | | | | | +----- SCROLL LOCK INDICATOR
  | | | | | | +------- NUM LOCK INDICATOR
  | | | | | +--------- CAPS LOCK INDICATOR
  | | | | +----------- CIRCUS SYSTEM INDICATOR
  | | | +------------- ACK RECEIVED
  | | +--------------- RESEND RECEIVED FLAG
  | +----------------- MODE INDICATOR UPDATE
  +------------------- KEYBOARD TRANSMIT ERROR FLAG
```

```
40:1E = KEYBOARD BUFFER (20H BYTES)
40:1C = BUFFER TAIL POINTER
40:72 = 1234H IF CTL-ALT-DEL PRESSED ON KEYBOARD
  INT 05 INVOKED IF PRINT SCREEN KEY PRESSED.
  INT 1B INVOKED IF CTL-BREAK KEY SEQUENCE PRESSED.
  INT 15, AH=85 INVOKED ON PC/AT AND AFTER IF SYSTEM
     REQUEST KEY IS PRESSED.
  INT 15, AH=4F INVOKED ON MACHINES AFTER PC/AT WITH
     AL = SCAN CODE


              +---------------------+
              | INT 10 - VIDEO BIOS |
              +---------------------+

INPUT PARAMETERS:

AH = 00 - SET VIDEO MODE

       AL = 00 - 40x25 ALPHANUMERIC B/W
          = 01 - 40x25 ALPHANUMERIC COLOR
          = 02 - 80x25 ALPHANUMERIC B/W
          = 03 - 80x25 ALPHANUMERIC COLOR
          = 04 - 320x200 COLOR GRAPHICS
          = 05 - 320x200 B/W GRAPHICS
          = 06 - 640x200 B/W GRAPHICS
          = 07 - 80x25 MONOCHROME ALPHA
          = 08 - 160x200 COLOR GRAPHICS (PCJR)
          = 09 - 320x200 COLOR GRAPHICS (PCJR)
          = 0A - 640x200 COLOR GRAPHICS (PCJR)
          = 0D - 320x200 MONOCHROME GRAPHICS (EGA +)
          = 0E - 640x200 MONOCHROME GRAPHICS (EGA +)
          = 0F - 640x350 MONOCHROME GRAPHICS (EGA +)
          = 10 - 640x350 MONOCHROME HI-RES (EGA +)

AH = 01 - SET CURSOR TYPE

       CH = TOP LINE FOR CURSOR (BITS 4-0)
       CL = BOTTOM LINE FOR CURSOR (BITS 4-0)

AH = 02 - SET CURSOR POSITION

       DH = ROW
       DL = COLUMN
       BH = PAGE NUMBER (0 FOR GRAPHICS MODES)

AH = 03 - READ CURSOR POSITION

       BH = PAGE NUMBER (0 FOR GRAPHICS MODES)
     ON EXIT:
       DH = ROW #
       DL = COLUMN #
       CH = TOP LINE FOR CURSOR (BITS 4-0)
       CL = BOTTOM LINE FOR CURSOR (BITS 4-0)

AH = 04 - READ LIGHT PEN POSITION
     ON EXIT:  AH = 0 IF LIGHT PEN SWITCH IS NOT TRIGGERED
               AH = 1 IF REGISTERS HAVE VALID VALUES
               DH = ROW
               DL = COLUMN
               CH = RASTER LINE (0-199)
               CX = RASTER LINE ON NEW GRAPHICS MODES
               BX = PIXEL COLUMN (0-319,639)

AH = 05 - SELECT ACTIVE DISPLAY PAGE
          AL = NEW PAGE INFO, SEE AH=0 FOR PAGE INFO
     ON EXIT:
               BH = CRT PAGE REGISTER
               BL = CPU PAGE REGISTER

AH = 06 - SCROLL ACTIVE PAGE UP
          AL = NUMBER OF LINES, INPUT LINES BLANKED
          CX = ROW,COLUMN OF UPPER LEFT CORNER OF SCROLL
          DX = ROW,COLUMN OF LOWER RIGHT CORNER OF SCROLL
          BH = ATTRIBUTE TO BE USED ON BLANK LINE

AH = 07 - SCROLL ACTIVE PAGE DOWN
          AL = NUMBER OF LINES, INPUT LINES BLANKED
          CX = ROW,COLUMN OF UPPER LEFT CORNER OF SCROLL
          DX = ROW,COLUMN OF LOWER RIGHT CORNER OF SCROLL
          BH = ATTRIBUTE TO BE USED ON BLANK LINE

AH = 08 - READ ATTRIBUTE/CHARACTER AT CURSOR POSITION
          BH = DISPLAY PAGE
     ON EXIT:
          AL = CHARACTER AT CURRENT CURSOR POSITION
          AH = ATTRIBUTE OF CHARACTER (ALPHA MODES ONLY)

AH = 09 - WRITE ATTRIBUTE/CHARACTER AT CURSOR POSITION
          BH = DISPLAY PAGE
          CX = COUNT OF CHARACTERS TO WRITE
          AL = CHARACTER TO WRITE
          BL = ATTRIBUTE/COLOR OF CHARACTER. IF
               BIT 7=1 IN GRAPHICS MODE THEN XOR
               THE COLOR VALUE WITH ITS CURRENT VALUE

AH = 0A - WRITE CHARACTER ONLY AT CURRENT CURSOR POSITION
          BH = DISPLAY PAGE
          CX = COUNT OF CHARACTERS TO WRITE
          AL = CHARACTER TO WRITE
          NOTE: SHOULD NOT BE USED IN GRAPHICS MODES

AH = 0B - SET COLOR PALETTE
          BH = 0 FOR BACKGROUND, =1 FOR FOREGROUND
          BL = COLOR VALUE

AH = 0C - WRITE DOT
          BH = PAGE NUMBER
          DX = ROW NUMBER
          CX = COLUMN NUMBER
          AL = COLOR VALUE (XOR'D IF BIT 7=1)

AH = 0D - READ DOT
          BH = PAGE NUMBER
          DX = ROW NUMBER
          CX = COLUMN NUMBER
          AL = DOT READ UPON RETURN

AH = 0E - WRITE TELETYPE TO ACTIVE PAGE
          AL = CHARACTER TO WRITE
          BL = FOREGROUND COLOR IN GRAPHICS MODE
          BH = PAGE NUMBER

AH = 0F - RETURN CURRENT VIDEO STATE
     ON EXIT:  AL = MODE CURRENTLY SET (SEE AH=0)
               AH = NUMBER OF CHARACTER COLUMNS ON SCREEN
               BH = CURRENT ACTIVE DISPLAY PAGE

AH = 10 - SET/GET PALETTE REGISTERS (EGA AND AFTER)

          AL = 00 - SET INDIVIDUAL PALETTE REGISTER
                    BH = VALUE TO SET
                    BL = PALETTE REGISTER TO BE SET

          AL = 01 - SET OVERSCAN REGISTER
                    BH = VALUE TO SET

          AL = 02 - SET ALL PALETTE REGS AND OVERSCAN
                    ES:DX = POINTER TO 16-BYTE TABLE
                            OF REGISTER VALUES FOLLOWED
                            BY THE OVERSCAN VALUE

          AL = 03 - TOGGLE THE INTENSIFY/BLINKING BIT
                    BL = 0 - ENABLE INTENSIFY
                    BL = 1 - ENABLE BLINKING

AH = 11 - CHARACTER GENERATOR ROUTINE (EGA AND AFTER)

          AL = 00 - USER ALPHA LOAD
                    ES:BP = POINTER TO USER TABLE
                    CX = COUNT TO STORE
                    DX = CHARACTER OFFSET INTO TABLE
                    BL = BLOCK TO LOAD
                    BH = NUMBER OF BYTES PER CHARACTER

          AL = 01 - ROM MONOCHROME SET
                    BL = BLOCK TO LOAD
          AL = 02 - ROM 8X8 DOUBLE DOT
                    BL = BLOCK TO LOAD

          AL = 03 - SET BLOCK SPECIFIER
                    BL = CHARACTER GEN BLOCK SPECIFIER

          AL = 10 - USER ALPHA LOAD
                    ES:BP = POINTER TO USER TABLE
                    CX = COUNT TO STORE
                    DX = CHARACTER OFFSET INTO TABLE
                    BH = NUMBER OF BYTES PER CHARACTER
                    BL = BLOCK TO LOAD

          AL = 11 - ROM MONOCHROME SET
                    BL = BLOCK TO LOAD

          AL = 12 - ROM 8X8 DOUBLE DOT
                    BL = BLOCK TO LOAD

          AL = 20 - USER GRAPHICS CHARS INT 01FH (8X8)
                    ES:BP = POINTER TO USER TABLE

          AL = 21 - USER GRAPHICS CHARS
                    ES:BP = POINTER TO USER TABLE
                    CX = BYTES PER CHARACTER
                    BL = ROW SPECIFIER
                         BL = 0 - USER SPECIFIED
                                  DL = ROWS
                         BL = 1 - 0E ROWS
                         BL = 2 - 19 ROWS
                         BL = 3 - 2B ROWS
```

```
          AL = 22 - ROM 8X14 SET
                    BL = ROW SPECIFIER

          AL = 23 - ROM 8X8 DOUBLE DOT
                    BL = ROW SPECIFIER

          AL = 30 - RETURN INFORMATION
                    ES:BP = POINTER TO TABLE
                    DL = ROWS
                    CX = POINTS
                    BH = INFORMATION DESIRED:
                       = 0 - INT 1F POINTER
                       = 1 - INT 44 POINTER
                       = 2 - ROM 8X14 POINTER
                       = 3 - ROM DOUBLE DOT POINTER
                       = 4 - ROM DOUBLE DOT POINTER (TOP)
                       = 5 - ROM ALPHA ALTERNATE 9X14

AH = 12 - ALTERNATE SELECT (EGA AND AFTER)

          BL = 10 - RETURN EGA INFORMATION
          ON EXIT:  BH = 0 IF COLOR MODE IS IN EFFECT
                       = 1 IF MONO MODE IS IN EFFECT
                    BL = 00 IF 64K EGA MEMORY
                       = 01 IF 128K EGA MEMORY
                       = 10 IF 192K EGA MEMORY
                       = 11 IF 256K EGA MEMORY
                    CH = FEATURE BITS
                    CL = SWITCH SETTINGS

          BL = 20 - SELECT ALTERNATE PRINT SCREEN ROUTINE

AH = 13 - WRITE STRING (EGA AND AFTER)

          ES:BP = STRING TO BE WRITTEN
          CX = CHARACTER ONLY COUNT
          DX = POSITION TO BEGIN STRING, IN CURSOR TERMS
          BH = PAGE NUMBER
          BL = ATTRIBUTE IF AL BIT 1=0
          AL = TYPE OF WRITE:

          +-+-+-+-+-+-+-+-+
          |7|6|5|4|3|2|1|0| AL
          +-+-+-+-+-+-+-+-+
           | | | | | | | +----- MOVE CURSOR
           | | | | | | +------- STRING HAS ATTRIBUTES
           +-+-+-+-+-+--------- NOT USED

AH = 14 - LOAD LCD CHAR FONT (CONVERTIBLE ONLY)

          AL = 0 - LOAD USER SPECIFIED FONT
                   ES:DI = POINTER TO CHARACTER FONT
                   CX = NUMBER OF CHARACTERS TO STORE
                   DX = CHAR OFFSET INTO RAM FONT AREA
                   BH = NUMBER OF BYTES PER CHARACTER
                   BL = 0 - LOAD MAIN FONT (BLOCK 0)
                      = 1 - LOAD ALTERNATE FONT (BLOCK 1)

          AL = 1 - LOAD SYSTEM ROM DEFAULT FONT
                   BL = 0 - LOAD MAIN FONT (BLOCK 0)
                      = 1 - LOAD ALTERNATE FONT (BLOCK 1)

          AL = 2 - SET MAPPING OF LCD HIGH INT. ATTR.
                   BL = 0 - IGNORE HIGH INTENSITY ATTR.
                      = 1 - MAP HIGH INT. TO UNDERSCORE
                      = 2 - MAP HIGH INT. TO REVERSE VID.
                      = 3 - MAP HI INT. TO SEL. ALT FONT

AH = 15 - RETURN PHYSICAL DISPLAY PARMS (CONVERTIBLE)
     ON EXIT:
          AX = ALTERNATD DISPLAY ADAPTER TYPE
          ES:DI = POINTER TO PARAMETER TABLE:
              WORD #          INFORMATION
                1    MONITOR MODEL NUMBER
                2    VERTICAL PELS PER METER
                3    HORIZONTAL PELS PER METER
                4    TOTAL NUMBER OF VERTICAL PELS
                5    TOTAL NUMBER OF HORIZONTAL PELS
                6    HORIZ. PEL SEPERATION IN MICROMETERS
                7    VERT. PEL SEPERATION IN MICROMETERS
```

```
          +---------------------------------+
          | INT 11 - EQUIPMENT DETERMINATION |
          +---------------------------------+

 INPUT PARAMETERS: NONE
OUTPUT PARAMETERS:

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|F|E|D|C|B|A|9|8|7|6|5|4|3|2|1|0| AX
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 | | | | | | | | | | | | | | +- IPL DISKETTE INSTALLED
 | | | | | | | | | | | | | +--- MATH COPROCESSOR
 | | | | | | | | | | | | +----- NOT USED
 | | | | | | | | | | | +------- NOT USED
 | | | | | | | | | | +-+--------- INITIAL VIDEO MODE
 | | | | | | | | | +-+------------- # OF DISKETTE DRIVES
 | | | | | | | | +----------------- NOT USED
 | | | | +-+-+-+-------------------- NUMBER OF RS232 CARDS
 | | | +------------------------- NOT USED
 | | +--------------------------- INT. MODEM INSTALLED
 +-+----------------------------- NUMBER OF PRINTERS

  NOTE: INITIAL VIDEO MODE (BITS 4 & 5)
    00 - UNUSED              01 - 40x25 COLOR CARD
    10 - 80x25 COLOR CARD    11 - 80x25 MONOCHROME CARD


          +---------------------------------+
          | INT 12 - MEMORY SIZE DETERMINE |
          +---------------------------------+

    INPUT PARAMETERS:   NONE
   OUTPUT PARAMETERS:

    AX = NUMBER OF CONTIGUOUS 1K BLOCKS OF MEMORY

              MEMORY:   NONE


          +------------------------+
          | INT 13 - DISKETTE BIOS |
          +------------------------+

 INPUT PARAMETERS:

    AH = 00 - RESET DISKETTE SYSTEM
    ON EXIT:  AH = STATUS

    AH = 01 - READ STATUS OF LAST OPERATION
    ON EXIT:  AH = STATUS

    AH = 02 - READ THE DESIRED SECTORS INTO MEMORY
              AL = NUMBER OF SECTORS
              BX = ADDRESS OF BUFFER (ES=SEGMENT)
              CH = TRACK NUMBER
              CL = SECTOR NUMBER
              DH = HEAD NUMBER
              DL = DRIVE NUMBER (BIT 7=0)
    ON EXIT:
              AL = NUMBER OF SECTORS TRANSFERRED
              AH = STATUS

    AH = 03 - WRITE THE DESIRED SECTORS FROM MEMORY
              AL,BX,CX,DX SAME AS AH=02

    AH = 04 - VERIFY THE DESIRED SECTORS
              AL,BX,CX,DX SAME AS AH=02

    AH = 05 - FORMAT THE DESIRED TRACK
              AL,BX,CX,DX SAME AS AH=02
              BUFFER CONTAINS ADDRESS FIELDS OF THE
              FORM C,H,R,N FOR EACH SECTOR FORMATTED

    AH = 06 - RESERVED FOR FIXED DISK

    AH = 07 - RESERVED FOR FIXED DISK

    AH = 08 - READ DRIVE PARAMETERS
    ON EXIT:  AH = STATUS OF OPERATION

    AH = 15 - READ DASD TYPE
              DL = DRIVE NUMBER
    ON EXIT:
              AH = 00 IF DRIVE NOT PRESENT
                 = 01 IF DISKETTE WITHOUT CHANGE LINE
                 = 02 IF DISKETTE WITH CHANGE LINE

    AH = 16 - DISK CHANGE LINE STATUS
              DL = DRIVE NUMBER
    ON EXIT:
              AH = STATUS OF OPERATION
```

```
   AH = 17 - SET DASD TYPE FOR FORMAT
             DL = DRIVE NUMBER
             AL = 01 - 360KB DISKETTE IN 360KB DRIVE
                = 02 - 360KB DISKETTE IN 1.2M DRIVE
                = 03 - 1.2M DISKETTE IN 1.2M DRIVE
                = 04 - 720KB DISKETTE IN 720KB DRIVE
   ON EXIT:
             AH = STATUS OF OPERATION

   AH = 18 - SET MEDIA TYPE FOR FORMAT
             DL = DRIVE NUMBER
             CH = LOWER 8 BITS OF NUMBER OF TRACKS
             CL = HIGH 2 BITS OF NUMBER OF TRACKS (6,7)
                = SECTORS PER TRACK (BITS 0-5)
   ON EXIT:
             ES:DI = POINTER TO 11-BYTE PARM TABLE
             AH = 00 IF REQUESTED COMBINATION SUPPORTED
                = 01 IF FUNCTION NOT AVAILABLE
                = 0C IF NOT SUPP. OR DRIVE TYPE UNKNOWN
                = 80 IF THERE IS NO MEDIA IN THE DRIVE

               DISKETTE STATUS DEFINITIONS

  80 = TIME OUT               | 08 = DMA FAILURE
  40 = SEEK FAILURE           | 06 = MEDIA CHANGE
  20 = NEC CONTROLLER FAILURE | 04 = SECTOR NOT FOUND
  10 = CRC ERROR              | 03 = WRITE PROTECTED
  0C = UNSUPPORTED TRACK      | 02 = BAD ADDRESS MARK
  09 = DMA BOUNDARY ERROR     | 01 = INVALID FUNCTION

       CY = 1 IF STATUS PRESENTED IS NOT ZERO


            +--------------------------+
            | INT 13 - FIXED DISK BIOS |
            +--------------------------+

  UNLESS OTHERWISE NOTED, ALL FIXED DISK INTERRUPT CALLS
FUNCTION AS FOLLOWS:
 INPUT PARAMETERS:
            AH = FUNCTION REQUEST NUMBER
            AL = NUMBER OF SECTORS
            BX = ADDRESS OF USER BUFFER (ES = SEGMENT)
            CH = CYLINDER NUMBER
                 (BITS 9,A = BITS 7,8 OF CL)
            CL = SECTOR NUMBER
            DH = HEAD NUMBER
            DL = DRIVE NUMBER, BIT 7=1
OUTPUT PARAMETERS:
            AH = STATUS OF OPERATION:

   BB = UNDEFINED ERROR     | 09 = DMA BOUNDARY ERROR
   AA = DRIVE NOT READY     | 08 = DMA FAILURE
   80 = TIME OUT            | 07 = PARAMETER ACT. FAIL
   40 = SEEK FAILURE        | 05 = RESET FAILED
   20 = CONTROLLER FAILURE  | 04 = SECTOR NOT FOUND
   11 = DATA ECC CORRECTED  | 03 = WRITE PROTECT ERROR
   10 = BAD ECC ON DISK READ| 02 = BAD ADDRESS MARK
   0B = BAD TRACK DETECTED  | 01 = INVALID FUNCTION REQ.
   0A = BAD SECTOR DETECTED | 00 = NO ERROR

            CY = 1 IF STATUS NOT ZERO

 AH = 00 - RESET FIXED DISK SYSTEM

 AH = 01 - READ STATUS OF LAST OPERATION
 ON EXIT:  AL = STATUS OF LAST OPERATION

 AH = 02 - READ THE DESIRED SECTORS INTO MEMORY

 AH = 03 - WRITE THE DESIRED SECTORS FROM MEMORY

 AH = 04 - VERIFY THE DESIRED SECTORS

 AH = 05 - FORMAT THE DESIRED TRACK
       AL = INTERLEAVE VALUE (XT ONLY)

       BX = FORMAT BUFFER, SIZE = 512 BYTES
            THE FIRST 2*(SECTORS/TRACK) BYTES
            CONTAIN F,N FOR EACH SECTOR.
            F=00 FOR GOOD SECTOR, 80 FOR BAD SECTOR
            N=SECTOR NUMBER

 AH = 06 - FORMAT TRACK AND SET BAD SECTOR FLAGS
          (VALID FOR XT AND PORTABLE ONLY)
       AL = INTERLEAVE VALUE (XT ONLY)
       BX = FORMAT BUFFER, SIZE = 512 BYTES
            THE FIRST 2*(SECTORS/TRACK) BYTES
            CONTAIN F,N FOR EACH SECTOR.
            F=00 FOR GOOD SECTOR, 80 FOR BAD SECTOR
            N=SECTOR NUMBER

 AH = 07 - FORMAT THE DRIVE STARTING AT THE DESIRED TRACK
          (VALID FOR XT AND PORTABLE ONLY)
       AL = INTERLEAVE VALUE (XT ONLY)

       BX = FORMAT BUFFER, SIZE = 512 BYTES
            THE FIRST 2*(SECTORS/TRACK) BYTES
            CONTAIN F,N FOR EACH SECTOR.
            F=00 FOR GOOD SECTOR, 80 FOR BAD SECTOR
            N=SECTOR NUMBER

AH = 08 - READ DRIVE PARAMETERS
ON EXIT:
       DL = NUMBER OF CONSECUTIVE ACKNOWLEDGING DRIVES
       DH = MAXIMUM USABLE VALUE FOR HEAD NUMBER
       CH = MAXIMUM USEABLE VALUE FOR CYLINDER NUMBER
       CL = MAXIMUM USEABLE VALUE FOR SECTOR NUMBER
            AND CYLINDER NUMBER HIGH BITS

AH = 09 - INITIALIZE DRIVE PAIR CHARACTERISTICS

       INT 41 POINTS TO DATA BLOCK FOR DRIVE 80
       INT 46 POINTS TO DATA BLOCK FOR DRIVE 81

             DATA BLOCK DEFINITIONS:
    +0 = MAXIMUM NUMBER OF CYLINDERS (DW)
    +2 = MAXIMUM NUMBER OF HEADS (DB)
    +3 = STARTING REDUCED WRITE CURRENT CYLINDER
         (DW - XT ONLY)
    +5 = STARTING WRITE PRECOMP CYLINDER (DW)
    +7 = MAXIMUM ECC DATA BURST LENGTH (DB - XT ONLY)
    +8 = CONTROL BYTE:

         +-+-+-+-+-+-+-+-+
         |7|6|5|4|3|2|1|0|
         +-+-+-+-+-+-+-+-+
          | | | | | +-+-+----- DRIVE OPTION
          | | +-+-+----------- ALWAYS ZERO
          | +----------------- DISABLE ECC RETRIES
          +------------------- DISABLE ACCESS RETIRES

AH = 0A - READ LONG
ON EXIT:  AL = NUMBER OF SECTORS ACTUALLY TRANSFERRED

AH = 0B - WRITE LONG
ON EXIT:  AL = NUMBER OF SECTORS ACTUALLY TRANSFERRED

AH = 0C - SEEK

AH = 0D - ALTERNATE DISK RESET

AH = 0E - READ SECTOR BUFFER (XT, PORTABLE ONLY)
ON EXIT:  AL = NUMBER OF SECTORS ACTUALLY TRANSFERRED

AH = 0F - WRITE SECTOR BUFFER (XT, PORTABLE ONLY)
ON EXIT:  AL = NUMBER OF SECTORS ACTUALLY TRANSFERRED

AH = 10 - TEST DRIVE READY

AH = 11 - RECALIBRATE

AH = 12 - CONTROLLER RAM DIAGNOSTIC (XT, PORTABLE ONLY)

AH = 13 - DRIVE DIAGNOSTIC (XT, PORTABLE ONLY)

AH = 14 - CONTROLLER INTERNAL DIAGNOSTIC

AH = 15 - READ DASD TYPE (PC/AT ONLY)
ON EXIT:  AH = 00 IF DRIVE NOT PRESENT
             = 03 IF FIXED DISK PRESENT
          CX,DX = NUMBER OF 512 BYTE BLOCKS

AH = 16 - RESERVED FOR DISKETTE

AH = 17 - RESERVED FOR DISKETTE

AH = 18 - RESERVED FOR DISKETTE

AH = 19 - PARK FIXED DISK HEADS (XT MODEL 286)


  +-----------------------------------------+
  | INT 14 - ASYNCHRONOUS COMMUNICATIONS BIOS |
  +-----------------------------------------+

 FOR INT 14 THE FOLLOWING STATUS IS DEFINED:

     +-+-+-+-+-+-+-+-+
     |7|6|5|4|3|2|1|0| MODEM STATUS
     +-+-+-+-+-+-+-+-+
      | | | | | | | +----- DELTA CLEAR TO SEND
      | | | | | | +------- DELTA DATA SET READY
      | | | | | +--------- TRAILING EDGE RING DETECTOR
      | | | | +----------- DELTA RECEIVE LINE SIGNAL DET.
      | | | +------------- CLEAR TO SEND
      | | +--------------- DATA SET READY
      | +----------------- RING INDICATOR
      +------------------- RECEIVE LINE SIGNAL DETECT
```

```
     +-+-+-+-+-+-+-+-+
     |7|6|5|4|3|2|1|0| LINE STATUS
     +-+-+-+-+-+-+-+-+
      | | | | | | | +----- DATA READY
      | | | | | | +------- OVERRUN ERROR
      | | | | | +--------- PARITY ERROR
      | | | | +----------- FRAMING ERROR
      | | | +------------- BREAK DETECT
      | | +--------------- TRANSMIT HOLDING REG. EMPTY
      | +----------------- TRANSMIT SHIFT REGISTER EMPTY
      +------------------- TIME OUT *

  * NOTE: IF BIT 7 SET THEN OTHER BITS ARE INVALID

 ALL ROUTINES HAVE AH=FUNCTION NUMBER AND DX=RS232 CARD
 NUMBER (0 BASED). AL=CHARACTER TO SEND OR RECEIVED
 CHARACTER ON EXIT, UNLESS OTHERWISE NOTED.

AH = 00 - INITIALIZE COMMUNICATIONS PORT
          AL = PARMS FOR INITIALIZATION:
          +-+-+-+-+-+-+-+-+
          |7|6|5|4|3|2|1|0| AL
          +-+-+-+-+-+-+-+-+
           | | | | | | +-+----- WORD LENGTH
           | | | | | +--------- STOP BITS
           | | | +-+----------- PARITY
           +-+-+--------------- BAUD RATE

          WORD LENGTH: 10 = 7 BITS
                       11 = 8 BITS

            STOP BITS:  0 = 1 STOP BIT
                        1 = 2 STOP BITS

               PARITY: X0 = NONE
                       01 = ODD
                       11 = EVEN

            BAUD RATE: 000 = 110 BAUD
                       001 = 150 BAUD
                       010 = 300 BAUD
                       011 = 600 BAUD
                       100 = 1200 BAUD
                       101 = 2400 BAUD
                       110 = 4800 BAUD
                       111 = 9600 BAUD

 ON EXIT:  AL = MODEM STATUS
           AH = LINE STATUS

 AH = 01 - SEND CHARACTER IN AL
 ON EXIT:  AH = LINE STATUS

 AH = 02 - RECEIVE CHARACTER IN AL
 ON EXIT:  AH = LINE STATUS

 AH = 03 - READ STATUS
 ON EXIT:  AH = LINE STATUS
           AL = MODEM STATUS

 AH = 04 - EXTENDED INITIALIZE
           AL = BREAK STATUS
              = 1 IF BREAK
              = 0 IF NO BREAK
           BH = PARITY
              = 0 - NO PARITY
              = 1 - ODD PARITY
              = 2 - EVEN PARITY
              = 3 - STICK PARITY ODD
              = 4 - STICK PARITY EVEN
           BL = NUMBER OF STOP BITS
              = 0 - ONE STOP BIT
              = 1 - 2 STOP BITS (1&frac12. IF 5 BIT WORD
LEN)
           CH = WORD LENGTH
              = 0 - 5 BITS
              = 1 - 6 BITS
              = 2 - 7 BITS
              = 3 - 8 BITS
           CL = BAUD RATE
              = 0 - 110        = 5 - 2400
              = 1 - 150        = 6 - 4800
              = 2 - 300        = 7 - 9600
              = 3 - 600        = 8 - 19200
              = 4 - 1200
 ON EXIT:
           AL = MODEM STATUS
           AH = LINE CONTROL STATUS

 AH = 05 - EXTENDED COMMUNICATION PORT CONTROL
           AL = 00 - READ MODEM CONTROL REGISTER
                     BL = MODEM CONTROL REG (SEE AL=1)
           AL = 01 - WRITE MODEM CONTROL REGISTER
                     BL = MODEM CONTRO REGISTER:
          +-+-+-+-+-+-+-+-+
          |7|6|5|4|3|2|1|0| BL (FOR AL=00 AND AL=01)
          +-+-+-+-+-+-+-+-+
           | | | | | | | +----- DATA TERMINAL READY
           | | | | | | +------- REQUEST TO SEND
           | | | | | +--------- OUT1
           | | | | +----------- OUT2
           | | | +------------- LOOP
           +-+-+--------------- RESERVED
 ON EXIT:
          AH = STATUS


          +-------------------------------+
          | INT 15 - SYSTEM SERVICES BIOS |
          +-------------------------------+

 INPUT PARAMETERS:

 AH = 00 - TURN CASSETTE MOTOR ON (PC,PCjr ONLY)

 AH = 01 - TURN CASSETTE MOTOR OFF (PC,PCjr ONLY)

 AH = 02 - READ BLOCKS FROM CASETTE (PC,PCjr ONLY)
           ES:BX = OFFSET OF DATA BUFFER
           CX = COUNT OF BYTES TO READ
 ON EXIT:
           ES:BX = OFFSET OF LAST BYTE READ+1
           DX = COUNT OF BYTES ACTUALLY READ
           AH = ERROR CODE IF CY=1

 AH = 03 - WRITE BLOCKS TO CASETTE (PC,PCjr ONLY)
           ES:BX = OFFSET OF DATA BUFFER
           CX = COUNT OF BYTES TO WRITE
 ON EXIT:
           ES:BX = OFFSET OF LAST BYTE WRITTEN+1
           AH = ERROR CODE IF CY=1

 AH = 40 - READ / MODIFY PROFILES (CONVERTIBLE ONLY)
           AL = 00 - RETURN SYSTEM PROFILE IN CX,BX
           AL = 01 - MODIFY SYSTEM PROFILE
                     CX,BX = PROFILE INFO
           AL = 02 - RET. INTERNAL MODEM PROFILE IN BX
           AL = 03 - MODIFY INTERNAL MODEM PROFILE
                     BX = PROFILE INFO

 AH = 41 - WAIT ON EXTERNAL EVENT (CONVERTIBLE ONLY)
           AL = CONDITION TYPE:

          +-+-+-+-+-+-+-+-+
          |7|6|5|4|3|2|1|0| AL
          +-+-+-+-+-+-+-+-+
           | | | | | +-+-+----- CONDITION TO WAIT FOR
           | | | | +----------- RESERVED
           | | | +------------- 1=PORT ADDR./0=USER BYTE
           +-+-+--------------- RESERVED

           CONDITION CODES:

           0 = ANY EXTERNAL EVENT
           1 = COMPARE AND RETURN IF EQUAL
           2 = COMPARE AND RETURN IF NOT EQUAL
           3 = TEST AND RETURN IF NOT ZERO
           4 = TEST AND RETURN IF ZERO

           BH = CONDITION COMPARE OR MASK VALUE
           BL = TIMEOUT VALUE TIMES 55 MSEC.
              = 0 IF NO TIME LIMIT
           DX = I/O PORT ADDRESS (IF AL BIT 4 = 1)
           ES:DI = POINTER TO USER BYTE (IF AL BIT 4 = 0)

 AH = 42 - REQUEST SYSTEM POWER OFF (CONVERTIBLE ONLY)
           AL = 00 TO USE SYSTEM PROFILE
              = 01 TO FORCE SUSPEND REGARDLESS OF PROFILE

 AH = 43 - READ SYSTEM STATUS (CONVERTIBLE ONLY)

 ON EXIT:  AL = STATUS:
           +-+-+-+-+-+-+-+-+
           |7|6|5|4|3|2|1|0| AL
           +-+-+-+-+-+-+-+-+
            | | | | | | | +----- LCD DETACHED
            | | | | | | +------- RESERVED
            | | | | | +--------- RS232/PARALLEL POWERED ON
            | | | | +----------- INTERNAL MODEM POWERED ON
            | | | +------------- POWER ACTIVATED BY ALARM
            | | +--------------- STANDBY POWER LOST
            | +----------------- EXTERNAL POWER IN USE
            +------------------- BATTERY LOW

 AH = 44 - (DE)ACTIVATE INTERNAL MODEM POWER (CONVERTIBLE)
           AL = 00 TO POWER OFF
              = 01 TO POWER ON
```

```
AH = 4F - KEYBOARD INTERCEPT
          AL = SCAN CODE, CY=1
ON EXIT:
          AL = SCAN CODE, CY=1 IF PROCESSING DESIRED

AH = 80 - DEVICE OPEN
          BX = DEVICE ID
          CX = PROCESS ID

AH = 81 - DEVICE CLOSE
          BX = DEVICE ID
          CX = PROCESS ID

AH = 82 - PROGRAM TERMINATION
          BX = DEVICE ID

AH = 83 - EVENT WAIT
          AL = 0 TO SET INTERVAL
             = 1 TO CANCEL
          ES:BX = POINTER TO MEMORY FLAG (BIT 7 IS SET
                  WHEN INTERVAL EXPIRES)
          CX,DX = NUMBER OF MICROSECONDS TO WAIT
                  (GRANULARITY IS 976 MICROSECONDS)
ON EXIT:  CY = 1 IF FUNCTION ALREADY BUSY

AH = 84 - JOYSTICK SUPPORT
          DX = 0 TO READ THE CURRNENT SWITCH SETTINGS
ON EXIT:  AL = SWITCH SETTINGS (BITS 7-4)
ON ENTRY: DX = 1 TO READ THE RESISTIVE INPUTS
ON EXIT:  AX = A(X),BX = A(Y), CX = B(X), DX = B(Y)

AH = 85 - SYSTEM REQUEST KEY PRESSED
          AL = 00 MAKE OF KEY
             = 01 BREAK OF KEY

AH = 86 - ELAPSED TIME WAIT (PCAT AND AFTER)
          CL,DX = NUMBER OF MICROSECONDS TO WAIT

AH = 87 - MOVE BLOCK TO/FROM EXTENDED MEMORY
          CX = WORD COUNT OF BLOCK TO BE MOVED
          ES:SI = POINTER TO GLOBAL DESCRIPTOR TABLE

AH = 88 - EXTENDED MEMORY SIZE DETERMINE
ON EXIT:  AX = NUMBER OF CONTIGUOUS 1K BLOCKS OF
               MEMORY STARTING AT ADDRESS 1024K

AH = 89 - SWITCH PROCESSOR TO PROTECTED MODE
          ES:SI = POINTER TO GDT
          BH = OFFSET INTO IDT WHERE INTS 0-7 ARE
          BL = OFFSET INTO IDT WHERE INTS 8-15 ARE

AH = 90 - DEVICE BUSY
          AL = TYPE CODE:
             = 00 - DISK
             = 01 - DISKETTE
             = 02 - KEYBOARD
             = 80 - NETWORK (ES:BX = NCB)
             = FC - DISK RESET
             = FD - DISKETTE MOTOR START
             = FE - PRINTER
ON EXIT:  CY = 1 IF WAIT TIME SATISFIED

AH = 91 - INTERRUPT COMPLETE
          AL = TYPE CODE (SEE AH=90 ABOVE)

AH = C0 - RETURN SYSTEM CONFIGURATION PARAMETERS

ON EXIT:  ES:BX = POINTER TO SYSTEM DESCRIPTOR:
                  WORD - LENGTH OF DESCRIPTOR
                  BYTE - MODEL BYTE
                  BYTE - SECONDARY MODEL BYTE
                  BYTE - BIOS REVISION LEVEL
                  BYTE - FEATURE INFORMATION:
                  +-+-+-+-+-+-+-+-+
                  |7|6|5|4|3|2|1|0| FEATURE BYTE
                  +-+-+-+-+-+-+-+-+
                   | | | | +-+-+-+----- RESERVED
                   | | | +------------- INT 15 AH=4F USED
                   | | +--------------- RTC PRESENT
                   | +----------------- 2ND 8259 PRESENT
                   +------------------- DMA CHAN. 3 USED


           +------------------------+
           | INT 16 - KEYBOARD BIOS |
           +------------------------+

INPUT PARAMETERS:

AH = 00 - WAIT FOR KEYSTROKE AND READ
ON EXIT:
          AH = SCAN CODE
          AL = ASCII CHARACTER IF APPLICABLE

AH = 01 - GET KEYSTROKE STATUS
ON EXIT:
          ZF = 0 IF KEY PRESSED
          AH = SCAN CODE
          AL = ASCII CHARACTER IF APPLICABLE
          NOTE: CODE NOT REMOVED FROM BUFFER

AH = 02 - GET SHIFT STATUS
ON EXIT:

+-+-+-+-+-+-+-+-+
|7|6|5|4|3|2|1|0| AL
+-+-+-+-+-+-+-+-+
 | | | | | | | +----- RIGHT SHIFT KEY DEPRESSED
 | | | | | | +------- LEFT SHIFT KEY DEPRESSED
 | | | | | +--------- CONTROL SHIFT KEY DEPRESSED
 | | | | +----------- ALTERNATE SHIFT KEY DEPRESSED
 | | | +------------- SCROLL LOCK STATE ACTIVE
 | | +--------------- NUM LOCK STATE ACTIVE
 | +----------------- CAPS LOCK STATE ACTIVE
 +------------------- INSERT STATE IS ACTIVE

AH = 04 - KEYBOARD CLICK ADJUSTMENT
          AL = 1 FOR CLICK ON, =0 FOR CLICK OFF

AH = 05 - KEYBOARD BUFFER WRITE
          CH = SCAN CODE
          CL = ASCII CHARACTER
ON EXIT:
          AL = 01 IF BUFFER FULL

AH = 10 - EXTENDED WAIT FOR KEYSTROKE AND READ
ON EXIT:
          AH = SCAN CODE
          AL = ASCII CHARACTER IF APPLICABLE

AH = 11 - EXTENDED GET KEYSTROKE STATUS
ON EXIT:
          ZF = 0 IF KEY PRESSED
          AH = SCAN CODE
          AL = ASCII CHARACTER IF APPLICABLE
          NOTE: CODE NOT REMOVED FROM BUFFER

AH = 12 - EXTENDED GET SHIFT STATUS
ON EXIT:

+-+-+-+-+-+-+-+-+
|7|6|5|4|3|2|1|0| AL
+-+-+-+-+-+-+-+-+
 | | | | | | | +----- RIGHT SHIFT KEY DEPRESSED
 | | | | | | +------- LEFT SHIFT KEY DEPRESSED
 | | | | | +--------- CONTROL SHIFT KEY DEPRESSED
 | | | | +----------- ALTERNATE SHIFT KEY DEPRESSED
 | | | +------------- SCROLL LOCK STATE ACTIVE
 | | +--------------- NUM LOCK STATE ACTIVE
 | +----------------- CAPS LOCK STATE ACTIVE
 +------------------- INSERT STATE IS ACTIVE

+-+-+-+-+-+-+-+-+
|7|6|5|4|3|2|1|0| AH
+-+-+-+-+-+-+-+-+
 | | | | | | | +----- LEFT CONTROL KEY PRESSED
 | | | | | | +------- LEFT ALT KEY DEPRESSED
 | | | | | +--------- RIGHT CONTROL KEY PRESSED
 | | | | +----------- RIGHT ALT KEY DEPRESSED
 | | | +------------- SCROLL LOCK KEY DEPRESSED
 | | +--------------- NUM LOCK KEY DEPRESSED
 | +----------------- CAPS LOCK KEY DEPRESSED
 +------------------- SYSTEM REQUEST KEY DEPRESSED


           +------------------------+
           | INT 17 - PRINTER BIOS |
           +------------------------+

AH = 00 - PRINT CHARACTER
          AL = CHARACTER TO PRINT
          DX = PRINTER TO BE USED (0,1,2)
ON EXIT:
          AH = STATUS

AH = 01 - INITIALIZE THE PRINTER PORT
          DX = PRINTER PORT TO BE INITIALIZED (0,1,2)
ON EXIT:
          AH = STATUS

AH = 02 - READ PRINTER PORT STATUS
          DX = PRINTER PORT TO BE USED (0,1,2)
ON EXIT:
          AH = STATUS
```

```
       +-+-+-+-+-+-+-+-+
       |7|6|5|4|3|2|1|0| PRINTER STATUS BITS
       +-+-+-+-+-+-+-+-+
        | | | | | | | +----- TIME OUT
        | | | | | +-+------- NOT USED
        | | | | +----------- I/O ERROR
        | | | +------------- SELECTED
        | | +--------------- OUT OF PAPER
        | +----------------- ACKNOWLEDGE
        +------------------- NOT BUSY
```

```
          +---------------------------+
          | INT 19 - BOOTSTRAP LOADER |
          +---------------------------+
```

```
 INPUT PARAMETERS: NONE
OUTPUT PARAMETERS: NONE
           OTHER: TRACK 0, SECTOR 1 IS LOADED INTO
                  ADDRESS 0:7C00 AND CONTROL IS
                  TRANSFERRED THERE
```

```
    +-----------------------------------------+
    | INT 1A - SYSTEM AND REAL TIME CLOCK BIOS |
    +-----------------------------------------+
```

```
 INPUT PARAMETERS:

 AH = 00 - READ SYSTEM TIME COUNTER
 ON EXIT:  CX = HIGH PORTION OF COUNT
           DX = LOW PORTION OF COUNT
           AL = 0 IF 24 HOURS NOT PASSED SINCE RESET

 AH = 01 - SET SYSTEM TIME COUNTER
           CX = HIGH PORTION OF COUNT
           DX = LOW PORTION OF COUNT

 AH = 02 - READ REAL TIME CLOCK TIME (PC/AT ONLY)
 ON EXIT:  CH = HOURS IN BCD
           CL = MINUTES IN BCD
           DH = SECONDS IN BCD
           DL = 1 IF DAYLIGHT SAVINGS TIME OPTION
           CY = 1 IF CLOCK NOT OPERATING

 AH = 03 - SET REAL TIME CLOCK TIME (PC/AT ONLY)
           CH = HOURS IN BCD
           CL = MINUTES IN BCD
           DH = SECONDS IN BCD
           DL = 1 IF DAYLIGHT SAVINGS TIME OPTION

 AH = 04 - READ REAL TIME CLOCK DATE (PC/AT ONLY)
 ON EXIT:  CH = CENTURY IN BCD
           CL = YEAR IN BCD
           DH = MONTH IN BCD
           DL = DAY IN BCD
           CY = 1 IF CLOCK NOT OPERATING

 AH = 05 - SET REAL TIME CLOCK DATE (PC/AT ONLY)
           CH = CENTURY IN BCD
           CL = YEAR IN BCD
           DH = MONTH IN BCD
           DL = DAY IN BCD

 AH = 06 - SET REAL TIME CLOCK ALARM (PC/AT ONLY)
           CH = HOURS IN BCD
           CL = MINUTES IN BCD
           DH = SECONDS IN BCD
 ON EXIT:
           CY = 1 IF ALARM ALREADY SET OR CLOCK INOPERABLE
           INT 4A OCCURS AT SPECIFIED ALARM TIME

 AH = 07 - RESET REAL TIME CLOCK ALARM

 AH = 08 - SET RTC ACTIVATED POWER ON MODE (CONVERTIBLE)
           CH = HOURS IN BCD
           CL = MINUTES IN BCD
           DH = SECONDS IN BCD

 AH = 09 - READ RTC ALARM TIME AND STATUS (CONVERTIBLE)
 ON EXIT:  CH = HOURS IN BCD
           CL = MINUTES IN BCD
           DH = SECONDS IN BCD
           DL = ALARM STATUS:
              = 00 IF ALARM NOT ENABLED
              = 01 IF ALARM ENABLED BUT WILL NOT POWER
                UP SYSTEM
              = 02 IF ALARM WILL POWER UP SYSTEM

 AH = 80 - SET UP SOUND MULTIPLEXOR (PCjr ONLY)
           AL = 00 - SOURCE IS 8253 CHANNEL 2
              = 01 - SOURCE IS CASSETTE INPUT
              = 02 - SOURCE IS I/O CHANNEL "AUDIO IN"
              = 03 - SOURCE IS SOUND GENERATOR CHIP
```

13 DOS Interrupts

```
            +---------------+
            | ERROR RETURNS |
            +---------------+
```

```
OF THE FOLLOWING ERROR CODES, ONLY CODES 1-12 ARE
RETURNED IN AX UPON EXIT FROM INTERRUPT 21 OR 24.
THE REST ARE OBTAINED BY ISSUING THE "GET EXTENDED
ERROR" FUNCTION CALL (SEE INT 21, AH=59).

 01 - INVALID FUNCTION NUMBER
 02 - FILE NOT FOUND
 03 - PATH NOT FOUND
 04 - TOO MANY OPEN FILES (NO HANDLES LEFT)
 05 - ACCESS DENIED
 06 - INVALID HANDLE
 07 - MEMORY CONTROL BLOCKS DESTROYED
 08 - INSUFFICIENT MEMORY
 09 - INVALID MEMORY BLOCK ADDRESS
 0A - INVALID ENVIRONMENT
 0B - INVALID FORMAT
 0C - INVALID ACCESS CODE
 0D - INVALID DATA
 0E - RESERVED
 0F - INVALID DRIVE WAS SPECIFIED
 10 - ATTEMPT TO REMOVE THE CURRENT DIRECTORY
 11 - NOT SAME DEVICE
 12 - NO MORE FILES
 13 - ATTEMPT TO WRITE ON A WRITE-PROTECTED DISKETTE
 14 - UNKNOWN UNIT
 15 - DRIVE NOT READY
 16 - UNKNOWN COMMAND
 17 - CRC ERROR
 18 - BAD REQUEST STRUCTURE LENGTH
 19 - SEEK ERROR
 1A - UNKNOWN MEDIA TYPE
 1B - SECTOR NOT FOUND
 1C - PRINTER OUT OF PAPER
 1D - WRITE FAULT
 1E - READ FAULT
 1F - GENERAL FAILURE
 20 - SHARING VIOLATION
 21 - LOCK VIOLATION
 22 - INVALID DISK CHANGE
 23 - FCB UNAVAILABLE
24-4F RESERVED
 50 - FILE EXISTS
 51 - RESERVED
 52 - CANNOT MAKE
 53 - FAIL ON INT 24

       -------- ERROR CLASSES --------

 01 - OUT OF RESOURCE     | 08 - NOT FOUND
 02 - TEMPORARY SITUATION | 09 - BAD FORMAT
 03 - AUTHORIZATION       | 0A - LOCKED
 04 - INTERNAL            | 0B - MEDIA FAILURE
 05 - HARDWARE FAILURE    | 0C - ALREADY EXISTS
 06 - SYSTEM FAILURE      | 0D - UNKNOWN
 07 - APPLICATION ERROR   |


     --------- ACTION CODES ----------

 01 - RETRY               | 05 - IMMEDIATE EXIT
 02 - DELAY RETRY         | 06 - IGNORE
 03 - RE-ENTER INPUT      | 07 - USER INTERVENTION
 04 - ABORT               |


    -------------- LOCUS --------------

 01 - UNKNOWN             | 04 - SERIAL DEVICE
 02 - BLOCK DEVICE        | 05 - MEMORY
 03 - RESERVED            |
```

```
          +--------------------------+
          | PRE-DEFINED FILE HANDLES |
          +--------------------------+
```

```
0000 - STANDARD INPUT DEVICE - CAN BE REDIRECTED
0001 - STANDARD OUTPUT DEVICE - CAN BE REDIRECTED
0002 - STANDARD ERROR DEVICE - CANNOT BE REDIRECTED
0003 - STANDARD AUXILARY DEVICE
0004 - STANDARD PRINTER DEVICE
```

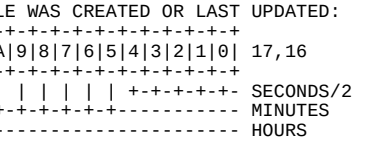
```
          +----------------------+
          | DOS INTERRUPT SUMMARY |
          +----------------------+

     20 - PROGRAM TERMINATE
     21 - FUNCTION REQUEST
     22 - TERMINATE ADDRESS
     23 - CTL-BREAK EXIT ADDRESS
     24 - CRITICAL ERROR HANDLER ADDRESS
     25 - ABSOLUTE DISK READ
     26 - ABSOLUTE DISK WRITE
     27 - TERMINATE BUT STAY RESIDENT
     2F - PRINTER


           +---------------------------+
           | INT 20 - PROGRAM TERMINATE |
           +---------------------------+

 DESCRIPTION: INT 20 RESTORES THE TERMINATE, CTL-BREAK,
              AND CRITICAL ERROR EXIT ADDRESSES, FLUSHES
              ALL BUFFERS, AND RETURNS TO DOS

 INPUT PARMS: CS = ADDRESS OF PROGRAM SEGMENT PREFIX

OUTPUT PARMS: NONE


           +---------------------------+
           | INT 21 - FUNCTION REQUEST |
           +---------------------------+

 DESCRIPTION: PERFORM A DOS FUNCTION.

 INPUT PARMS: AH = FUNCTION NUMBER. OTHER REGISTERS
              ARE SET AS DESCRIBED BELOW.

OUTPUT PARMS: IF ERROR ENCOUNTERED, CY IS SET TO 1 AND
              AX CONTAINS A RUDIMENTARY ERROR CODE.
              EXTENDED ERROR INFORMATION MAY BE OBTAINED
              BY ISSUING FUNCTION REQUEST 59.

   --------- FUNCTION REQUEST SUMMARY ----------

   00 - PROGRAM TERMINATE
   01 - WAIT FOR KEYBOARD INPUT
   02 - DISPLAY OUTPUT
   03 - WAIT FOR AUXILIARY DEVICE INPUT
   04 - AUXILARY OUTPUT
   05 - PRINTER OUTPUT
   06 - DIRECT CONSOLE I/O
   07 - WAIT FOR DIRECT CONSOLE INPUT WITHOUT ECHO
   08 - WAIT FOR CONSOLE INPUT WITHOUT ECHO
   09 - PRINT STRING
   0A - BUFFERED KEYBOARD INPUT
   0B - CHECK STANDARD INPUT STATUS
   0C - CLEAR KEYBOARD BUFFER, INVOKE KEYBOARD FUNCTION
   0D - DISK RESET
   0E - SELECT DISK
   0F - OPEN FILE
   10 - CLOSE FILE
   11 - SEARCH FOR FIRST ENTRY
   12 - SEARCH FOR NEXT ENTRY
   13 - DELETE FILE
   14 - SEQUENTIAL READ
   15 - SEQUENTIAL WRITE
   16 - CREATE A FILE
   17 - RENAME FILE
   19 - GET CURRENT DEFAULT DRIVE
   1A - SET DISK TRANSFER ADDRESS
   1B - GET ALLOCATION TABLE INFORMATION
   1C - GET ALLOCATION TABLE INFO FOR SPECIFIC DEVICE
   21 - RANDOM READ
   22 - RANDOM WRITE
   23 - GET FILE SIZE
   24 - SET RELATIVE RECORD FIELD
   25 - SET INTERRUPT VECTOR
   26 - CREATE NEW PROGRAM SEGMENT
   27 - RANDOM BLOCK READ
   28 - RANDOM BLOCK WRITE
   29 - PARSE FILENAME
   2A - GET DATE
   2B - SET DATE
   2C - GET TIME
   2D - SET TIME
   2E - SET/RESET VERIFY SWITCH
   2F - GET DISK TRANSFER ADDRESS
   30 - GET DOS VERSION NUMBER
   31 - TERMINATE PROCESS AND REMAIN RESIDENT
   33 - GET/SET CTL-BREAK CHECK STATE
   35 - GET VECTOR
   36 - GET DISK FREE SPACE
   38 - GET/SET COUNTRY DEPENDENT INFORMATION
   39 - CREATE SUBDIRECTORY (MKDIR)
   3A - REMOVE SUBDIRECTORY (RMDIR)
   3B - CHANGE CURRENT SUBDIRECTORY (CHDIR)
   3C - CREATE A FILE
   3D - OPEN A FILE
   3E - CLOSE A FILE HANDLE
   3F - READ FROM A FILE OR DEVICE
   40 - WRITE TO A FILE OR DEVICE
   41 - DELETE A FILE FROM A SPECIFIED SUBDIRECTORY
   42 - MOVE FILE READ/WRITE POINTER
   43 - CHANGE FILE MODE
   44 - I/O CONTROL FOR DEVICES
   45 - DUPLICATE A FILE HANDLE
   46 - FORCE A DUPLICATE OF A FILE HANDLE
   47 - GET CURRENT DIRECTORY
   48 - ALLOCATE MEMORY BLOCKS
   49 - FREE ALLOCATED MEMORY BLOCKS
   4A - MODIFY ALLOCATED MEMORY BLOCKS
   4B - LOAD/EXECUTE A PROGRAM
   4C - TERMINATE A PROCESS
   4D - GET RETURN CODE OF A SUB_PROCESS
   4E - FIND FIRST MATCHING FILE
   4F - FIND NEXT MATCHING FILE
   54 - GET VERIFY SETTING
   56 - RENAME A FILE
   57 - GET/SET A FILE'S DATE AND TIME
   59 - GET EXTENDED ERROR INFORMATION
   5A - CREATE A TEMPORARY FILE
   5B - CREATE A NEW FILE
   5C - LOCK/UNLOCK A FILE'S ACCESS
   62 - GET ADDRESS OF PROGRAM SEGMENT PREFIX

   AH = 00 - PROGRAM TERMINATE
       INPUT: NONE
      OUTPUT: NONE

   AH = 01 - WAIT FOR KEYBOARD INPUT
       INPUT: NONE
      OUTPUT: AL = CHARACTER FROM STANDARD INPUT DEVICE

   AH = 02 - DISPLAY OUTPUT
       INPUT: DL = ASCII CHARACTER TO OUTPUT
      OUTPUT: NONE

   AH = 03 - WAIT FOR AUXILIARY DEVICE INPUT
       INPUT: NONE
      OUTPUT: AL = CHARACTER FROM THE AUXILIARY DEVICE

   AH = 04 - AUXILIARY OUTPUT
       INPUT: DL = CHARACTER TO OUTPUT
      OUTPUT: NONE

   AH = 05 - PRINTER OUTPUT
       INPUT: DL = CHARACTER TO OUTPUT
      OUTPUT: NONE

   AH = 06 - DIRECT CONSOLE I/O
       INPUT: DL = CHARACTER TO OUTPUT IF <> FF
              DL = FF IF CONSOLE INPUT REQUEST
      OUTPUT: AL = INPUT CHARACTER IF DL = FF
              ZF = 1 IF DL = FF AND NO CHARACTER IS READY

   AH = 07 - WAIT FOR DIRECT CONSOLE INPUT WITHOUT ECHO
       INPUT: NONE
      OUTPUT: AL = CHARACTER FROM STANDARD INPUT DEVICE

   AH = 08 - WAIT FOR CONSOLE INPUT WITHOUT ECHO
       INPUT: NONE
      OUTPUT: AL = CHARACTER FROM STANDARD INPUT DEVICE

   AH = 09 - PRINT STRING
       INPUT: DS:DX = POINTER TO STRING ENDING IN "$"
      OUTPUT: NONE

   AH = 0A - BUFFERED KEYBOARD INPUT
       INPUT: DS:DX = POINTER TO INPUT BUFFER:
              +-+-+----------  +
              |1|2| BUFFER  ...|
              +-+-+----------  +
               | |     +-------- PLACE FOR INPUT CHARS
               | +-------------- NUMBER OF CHARS IN BUFFER
               +---------------- SIZE OF BUFFER
      OUTPUT: CHARACTERS UP TO AND INCLUDING A CR ARE
              PLACED INTO THE BUFFER BEGINNING AT BYTE
              3. BYTE 2 IS SET TO THE NUMBER OF CHARACTERS
              PLACED INTO THE BUFFER.

   AH = 0B - CHECK STANDARD INPUT STATUS
       INPUT: NONE
      OUTPUT: AL = FF IF CHARACTER IS AVAILABLE; 00 IF NOT

   AH = 0C - CLEAR KEYBOARD BUFFER, INVOKE KEYBOARD FUNCTION
       INPUT: AL = INT 21 FUNCTION # 01,06,07,08,OR 0A
      OUTPUT: KEYBOARD IS CLEARED,FUNCTION IN AL IS INVOKED
```

```
AH = 0D - DISK RESET
    INPUT: NONE
   OUTPUT: FLUSHES ALL FILE BUFFERS

AH = 0E - SELECT DISK
    INPUT: DL = DRIVE NUMBER (0=A,1=B,etc.)
   OUTPUT: AL = TOTAL NUMBER OF DRIVES INCL. HARDFILES

AH = 0F - OPEN A FILE
    INPUT: DS:DX = POINTER TO AN UNOPENED FCB
   OUTPUT: AL = 00 IF FILE OPENED; AL = FF IF NOT

AH = 10 - CLOSE A FILE
    INPUT: DS:DX = POINTER TO AN OPENED FCB
   OUTPUT: AL = 00 IF FILE IS FOUND; AL = FF IF NOT

AH = 11 - SEARCH FOR FIRST ENTRY
    INPUT: DS:DX = POINTER TO AN UNOPENED FCB
   OUTPUT: AL = 00 IF MATCHING FILENAME FOUND; FF IF NOT

AH = 12 - SEARCH FOR NEXT ENTRY
    INPUT: DS:DX = POINTER TO SAME FCB IN AH=11 ABOVE
   OUTPUT: AL = 00 IF MATCHING FILENAME FOUND; FF IF NOT

AH = 13 - DELETE A FILE
    INPUT: DS:DX = POINTER TO AN UNOPENED FCB
   OUTPUT: AL = 00 IF FILE DELETED
           AL = FF IF FILE NOT FOUND

AH = 14 - SEQUENTIAL READ
    INPUT: DS:DX = POINTER TO AN OPENED FCB
   OUTPUT: AL = 00 IF SUCCESSFUL READ
           AL = 01 IF END OF FILE (NO DATA READ)
           AL = 02 IF DTA IS TOO SMALL
           AL = 03 IF END OF FILE (PARTIAL RECORD READ)

AH = 15 - SEQUENTIAL WRITE
    INPUT: DS:DX = POINTER TO AN OPENED FCB
   OUTPUT: AL = 00 IF WRITE WAS SUCCESSFUL
           AL = 01 IF DISKETTE IS FULL
           AL = 02 IF DTA IS TOO SMALL

AH = 16 - CREATE A FILE
    INPUT: DS:DX = POINTER TO AN UNOPENED FCB
   OUTPUT: AL = 00 IF FILE CREATED; AL = FF IF NOT

AH = 17 - RENAME A FILE
    INPUT: DS:DX = POINTER TO A MODIFIED FCB
   OUTPUT: AL = 00 IF FILE RENAMED; AL = FF IF NOT

AH = 19 - GET CURRENT DEFAULT DRIVE
     INPUT: NONE
    OUTPUT: AL = CURRENT DEFAULT DRIVE (0=A,1=B,etc.)

AH = 1A - SET DISK TRANSFER ADDRESS
     INPUT: DS:DX = THE NEW DISK TRANSFER ADDRESS
    OUTPUT: NONE

AH = 1B - GET ALLOCATION TABLE INFORMATION
     INPUT: NONE
    OUTPUT: DS:BX = POINTER TO THE BYTE CONTAINING THE
                    FAT ID BYTE FOR THE DEFAULT DRIVE
            DX = THE NUMBER OF ALLOCATION UNITS
            AL = THE NUMBER OF SECTORS/ALLOCATION UNIT
            CX = THE SIZE OF THE PHYSICAL SECTOR

AH = 1C - GET ALLOCATION TABLE INFO FOR SPECIFIC DEVICE
     INPUT: DL = DRIVE NUMBER (0 FOR DEFAULT)
    OUTPUT: DS:BX = POINTER TO THE BYTE CONTAINING THE
                    FAT ID BYTE FOR THE REQUESTED DRIVE
            DX = THE NUMBER OF ALLOCATION UNITS
            AL = THE NUMBER OF SECTORS/ALLOCATION UNIT
            CX = THE SIZE OF THE PHYSICAL SECTOR

AH = 21 - RANDOM READ
     INPUT: DS:DX = POINTER TO AN OPENED FCB
    OUTPUT: AL = 00 IF READ WAS SUCCESSFUL
            AL = 01 IF EOF (NO DATA READ)
            AL = 02 IF DTA IS TOO SMALL
            AL = 03 IF EOF (PARTIAL RECORD READ)

AH = 22 - RANDOM WRITE
     INPUT: DS:DX = POINTER TO AN OPENED FCB
    OUTPUT: AL = 00 IF WRITE WAS SUCCESSFUL
            AL = 01 IF DISKETTE IS FULL
            AL = 02 IF DTA IS TOO SMALL

AH = 23 - GET FILE SIZE
     INPUT: DS:DX = POINTER TO AN UNOPENED FCB
    OUTPUT: AL = 00 IF THE DIRECTORY ENTRY IS FOUND
            AL = FF IF THE DIRECTORY ENTRY IS NOT FOUND

AH = 24 - SET RELATIVE RECORD FIELD
     INPUT: DS:DX = POINTER TO AN OPENED FCB
    OUTPUT: NONE

AH = 25 - SET INTERRUPT VECTOR
     INPUT: DS:DX = ADDRESS OF INTERRUPT HANDLER
            AL = INTERRUPT NUMBER
    OUTPUT: NONE

AH = 26 - CREATE A NEW PROGRAM SEGMENT
     INPUT: DX = SEGMENT NUMBER FOR THE NEW PROGRAM
    OUTPUT: NONE

AH = 27 - RANDOM BLOCK READ
     INPUT: DS:DX = POINTER TO AN OPENED FCB
            CX = THE NUMBER OF RECORDS TO BE READ
    OUTPUT: AL = 00 IF READ WAS SUCCESSFUL
                 01 IF EOF (NO DATA READ)
                 02 IF DTA IS TOO SMALL
                 03 IF EOF (PARTIAL RECORD READ)
            CX = ACTUAL NUMBER OF RECORDS READ

AH = 28 - RANDOM BLOCK WRITE
     INPUT: DS:DX = POINTER TO AN OPENED FCB
            CX = NUMBER OF RECORDS TO BE WRITTEN
    OUTPUT: AL = 00 IF WRITE WAS SUCCESSFUL
               = 01 IF DISKETTE IS FULL
               = 02 IF DTA IS TOO SMALL
            CX = ACTUAL NUMBER OF RECORDS WRITTEN

AH = 29 - PARSE A FILENAME
     INPUT: DS:SI = POINTER TO A COMMAND LINE TO PARSE
            ES:DI = POINTER TO A BUFFER TO BE FILLED
                    WITH AN UNOPENED FCB
            AL = BIT PATTERN TO CONTROL PARSING
                 (SEE DOS MANUAL FOR BIT MEANINGS)
    OUTPUT: AL = 00 IF NO GLOBAL CHARACTERS
               = 01 IF GLOBAL CHARACTERS PRESENT
               = FF IF DRIVE SPECIFIER IS INVALID
            DS:SI = POINTER TO THE FIRST CHARACTER
                    AFTER THE PARSED FILENAME
            ES:DI = POINTER TO THE UNOPENED FCB

AH = 2A - GET DATE
     INPUT: NONE
    OUTPUT: AL = DAY OF THE WEEK (0=SUNDAY)
            CX = YEAR (1980-2099)
            DH = MONTH (1-12)
            DL = DAY (1-31)

AH = 2B - SET DATE
     INPUT: CX = YEAR (1980-2099)
            DH = MONTH (1-12)
            DL = DAY (1-31)
    OUTPUT: AL = 00 IF DATE WAS VALID; AL=FF IF NOT

AH = 2C - GET TIME
     INPUT: NONE
    OUTPUT: CH = HOUR (0-23)
            CL = MINUTES (0-59)
            DH = SECONDS (0-59)
            DL = HUNDREDTHS (0-99)

AH = 2D - SET TIME
     INPUT: CH = HOUR (0-23)
            CL = MINUTES (0-59)
            DH = SECONDS (0-59)
            DL = HUNDREDTHS (0-99)
    OUTPUT: AL = 00 IF TIME WAS VALID; AL=FF IF NOT

AH = 2E - SET/RESET VERIFY SWITCH
     INPUT: AL = 01 TO SET VERIFY ON; AL=00 TO SET OFF
    OUTPUT: NONE

AH = 2F - GET DISK TRANSFER ADDRESS (DTA)
    INPUT: NONE
   OUTPUT: ES:BX = THE CURRENT DTA

AH = 30 - GET DOS VERSION NUMBER
    INPUT: NONE
   OUTPUT: AL = MAJOR VERSION NUMBER (LEFT OF DECIMAL)
           AH = MINOR VERSION NUMBER (RIGHT OF DECIMAL)
           BX,CX = 0000

AH = 31 - TERMINATE PROCESS AND REMAIN RESIDENT
    INPUT: AL = EXIT CODE (RETURNED TO BATCH FILES)
           DX = THE MEMORY SIZE IN PARAGRAPHS
   OUTPUT: NONE

AH = 33 - GET/SET CTL-BREAK CHECK STATE
    INPUT: AL = 00 TO REQUEST CURRENT STATE
           AL = 01 TO SET CURRENT STATE
           DL = 00 TO SET STATE OFF; DL=01 TO SET ON
   OUTPUT: DL = THE CURRENT STATE (00=OFF, 01=ON)

AH = 35 - GET VECTOR
    INPUT: AL = VECTOR NUMBER
   OUTPUT: ES:BX = POINTER TO INTERRUPT HANDLER
```

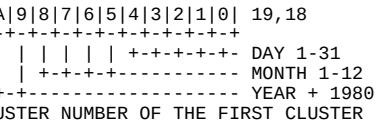
```
AH = 36 - GET DISK FREE SPACE
    INPUT: DL = DRIVE NUMBER (0=DEFAULT)
   OUTPUT: BX = AVAILABLE CLUSTERS
           DX = CLUSTERS PER DRIVE
           CX = BYTES PER SECTOR
           AX = FFFF IF DRIVE HAS INVALID SECTORS/CLUSTER

AH = 37 - GET COUNTRY DEPENDENT INFORMATION
    INPUT: DS:DX = POINTER TO BUFFER FOR RETURNED DATA
           AL = 00 TO GET CURRENT COUNTRY INFORMATION
           AL = 01-FE FOR COUNTRY CODES < 255
           AL = FF FOR COUNTRY CODES >255
           BX = COUNTRY CODE IF AL = FF
   OUTPUT: AX = ERROR CODE IF CARRY SET
           DS:DX = POINTER TO BUFFER OF RETURNED DATA
           BX = COUNTRY CODE

AH = 38 - SET CURRENT COUNTRY
    INPUT: DX = FFFF
           AL = 01-FE FOR COUNTRY CODES < 255
           AL = FF FOR COUNTRY CODES >255
           BX = COUNTRY CODE IF AL = FF
   OUTPUT: AX = ERROR CODE IF CARRY SET

   NOTE: SEE DOS MANUAL FOR COUNTRY INFORMATION
         BUFFER LAYOUT AND COUNTRY CODES

AH = 39 - CREATE SUBDIRECTORY (MKDIR)
    INPUT: DS:DX = POINTER TO ASCIIZ PATH NAME
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 3A - REMOVE SUBDIRECTORY (RMDIR)
    INPUT: DS:DX = POINTER TO ASCIIZ PATH NAME
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 3B - CHANGE THE CURRENT DIRECTORY (CHDIR)
    INPUT: DS:DX = POINTER TO ASCIIZ PATH NAME
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 3C - CREATE A FILE
    INPUT: DS:DX = POINTER TO ASCIIZ PATH NAME
           CX = THE ATTRIBUTE OF THE FILE
   OUTPUT: AX = ERROR CODE IF CARRY SET
           AX = FILE'S HANDLE IF CARRY NOT SET

AH = 3D - OPEN A FILE
    INPUT: DS:DX = POINTER TO AN ASCIIZ PATH NAME
                +-+-+-+-+-+-+-+-+
           AL = |7|6|5|4|3|2|1|0| OPEN MODE
                +-+-+-+-+-+-+-+-+
                 | | | | | +-+-+----- ACCESS MODE
                 | | | | +----------- ALWAYS 0
                 | +-+-+------------- SHARING MODE
                 +------------------- INHERITANCE FLAG
                (SEE DOS MANUAL FOR BIT VALUES)
   OUTPUT: AX = ERROR CODE IF CARRY SET
           AX = FILE HANDLE IF CARRY NOT SET

AH = 3E - CLOSE A FILE HANDLE
    INPUT: BX = THE FILE HANDLE TO CLOSE
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 3F - READ FROM A FILE OR DEVICE
    INPUT: BX = FILE HANDLE
           DS:DX = POINTER TO READ BUFFER
           CX = NUMBER OF BYTES TO READ
   OUTPUT: AX = ERROR CODE IF CARRY SET
           AX = NUMBER OF BYTES READ IF CARRY NOT SET

AH = 40 - WRITE TO A FILE OR DEVICE
    INPUT: BX = FILE HANDLE
           DS:DX = POINTER TO WRITE BUFFER
           CX = NUMBER OF BYTES TO WRITE
   OUTPUT: AX = ERROR CODE IF CARRY SET
           AX = NUMBER OF BYTES WRITTEN IF CARRY NOT SET

AH = 41 - DELETE A FILE FROM A SPECIFIED DIRECTORY
    INPUT: DS:DX = POINTER TO AN ASCIIZ FILENAME
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 42 - MOVE FILE READ/WRITE POINTER
    INPUT: CX:DX = DISTANCE TO MOVE IN BYTES (OFFSET)
           AL = ORIGIN OF MOVE:
                00 = BEGINNING OF FILE PLUS OFFSET
                01 = CURRENT LOCATION PLUS OFFSET
                02 = END OF FILE PLUS OFFSET
           BX = THE FILE'S HANDLE
   OUTPUT: AX = ERROR CODE IF CARRY SET
           DX:AX = NEW POINTER LOCATION IF CARRY NOT SET

AH = 43 - CHANGE FILE MODE
    INPUT: DS:DX = POINTER TO AN ASCIIZ PATH NAME
           CX = ATTRIBUTE
           AL = 00 TO GET ATTRIBUTE; 01 TO SET
   OUTPUT: AX = ERROR CODE IF CARRY SET
           CX = THE ATTRIBUTE

AH = 44 - I/O CONTROL FOR DEVICES
    INPUT: DS:DX = DATA OR BUFFER
           CX = NUMBER OF BYTES TO READ OR WRITE
           BX = FILE HANDLE
           BL = DEVICE NUMBER (0=DEFAULT)
           AL = FUNCTION VALUE
   OUTPUT: AX = ERROR CODE IF CARRY SET
           AX = # OF BYTES XFERRED IF CARRY NOT SET
     NOTE: SEE DOS MANUAL FOR FUNCTION VALUES

AH = 45 - DUPLICATE A FILE HANDLE
    INPUT: BX = FILE HANDLE
   OUTPUT: AX = ERROR CODE IF CARRY SET
           AX = NEW FILE HANDLE IF CARRY NOT SET

AH = 46 - FORCE A DUPLICATE OF A FILE HANDLE
    INPUT: BX = EXISTING FILE HANDLE
           CX = SECOND FILE HANDLE
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 47 - GET THE CURRENT DIRECTORY
    INPUT: DS:SI = POINTER TO A 64 BYTE USER AREA
           DL = DRIVE NUMBER (0=DEFAULT)
   OUTPUT: DS:SI = POINTER TO PATH NAME FROM ROOT
           AX = ERROR CODE IF CARRY SET

AH = 48 - ALLOCATE MEMORY
    INPUT: BX = # OF PARAGRAPHS OF MEMORY REQUESTED
   OUTPUT: AX:0 = POINTER TO ALLOCATED MEMORY BLOCK
           AX = ERROR CODE IF CARRY SET
           BX = SIZE OF THE LARGEST BLOCK OF MEMORY
                AVAILABLE (PARAGRAPHS) IF CARRY SET

AH = 49 - FREE ALLOCATED MEMORY
    INPUT: ES = SEGMENT OF THE BLOCK TO BE RETURNED
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 4A - MODIFY ALLOCATED MEMORY BLOCKS
    INPUT: ES = THE SEGMENT OF THE BLOCK
           BX = REQUESTED BLOCK SIZE IN PARAGRAPHS
   OUTPUT: AX = ERROR CODE IF CARRY SET
           BX = MAXIMUM SIZE POSSIBLE IF CARRY SET

AH = 4B - LOAD OR EXECUTE A PROGRAM
    INPUT: DS:DX = POINTER TO AN ASCIIZ FILENAME
           ES:BX = POINTER TO A PARAMETER BLOCK
           AL = 00 TO LOAD AND EXECUTE PROGRAM
           AL = 03 TO JUST LOAD
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 4C - TERMINATE A PROCESS
    INPUT: AL = RETURN CODE (FOR BATCH FILES)
   OUTPUT: NONE

AH = 4D - GET THE RETURN CODE OF A SUB-PROCESS
    INPUT: NONE
   OUTPUT: AL = EXIT CODE SENT BY SUB-PROCESS
           AH = 00 FOR NORMAL TERMINATION
           AH = 01 IF TERMINATED BY CTL-BREAK
           AH = 02 IF CRITICAL DEVICE ERROR
           AH = 03 IF TERMINATED BY FUNCTION REQ. 31

AH = 4E - FIND FIRST MATCHING FILE
    INPUT: DS:DX = POINTER TO ASCIIZ PATH/FILENAME
           CX = ATTRIBUTE USED DURING SEARCH
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 4F - FIND NEXT MATCHING FILE
    INPUT: DS:DX = UNCHANGED FROM PREVIOUS FUNCTION 4E
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 54 - GET VERIFY SETTING
    INPUT: NONE
   OUTPUT: AL = 00 IF VERIFY OFF, AL=01 IF VERIFY ON

AH = 56 - RENAME A FILE
    INPUT: DS:DX = POINTER TO OLD ASCIIZ PATH/FILENAME
           ES:DI = POINTER TO NEW ASCIIZ PATH/FILENAME
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 57 - GET/SET A FILE'S DATE AND TIME
    INPUT: AL = 00 TO GET; AL=01 TO SET
           BX = FILE'S HANDLE
           CX = TIME TO BE SET
           DX = DATE TO BE SET
   OUTPUT: AX = ERROR CODE IF CARRY SET
           CX = FILE'S TIME FROM TABLE
           DX = FILE'S DATE FROM TABLE
```

```
AH = 59 - GET EXTENDED ERROR INFORMATION
      INPUT: BX = 00 FOR VERSIONS  3.0, 3.1, 3.2
     OUTPUT: AX = EXTENDED ERROR CODE (SEE PANEL 1)
             BH = ERROR CLASS (SEE PANEL 1)
             BL = SUGGESTED ACTION (SEE PANEL 1)
             CH = LOCUS (SEE PANEL 1)

AH = 5A - CREATE A TEMPORARY FILE
      INPUT: DS:DX = POINTER TO ASCIIZ PATH ENDING IN \
             CX = ATTRIBUTE
     OUTPUT: AX = ERROR CODE IF CARRY SET
             DS:DX = POINTER TO ASCIIZ PATH/FILENAME

AH = 5B - CREATE A NEW FILE
      INPUT: DS:DX = POINTER TO ASCIIZ PATH/FILENAME
             CX = ATTRIBUTE
     OUTPUT: AX = ERROR CODE IF CARRY SET
             AX = HANDLE IF CARRY NOT SET

AH = 5C - LOCK/UNLOCK FILE ACCESS
    INPUT: AL = 00 TO LOCK; AL=01 TO UNLOCK
           BX = FILE HANDLE
           CX = OFFSET HIGH
           DX = OFFSET LOW
           SI = LENGTH HIGH
           DI = LENGTH LOW
   OUTPUT: AX = ERROR CODE IF CARRY SET

AH = 62 - GET PROGRAM SEGMENT PREFIX ADDRESS
    INPUT: NONE
   OUTPUT: BX = SEGMENT ADDRESS OF CURRENT PROCESS


         +---------------------------+
         | INT 22 - PROGRAM TERMINATE |
         +---------------------------+

INPUT: NONE. DO NOT EXECUTE THIS INTERRUPT DIRECTLY.
       SEE DOS MANUAL FOR MORE INFORMATION.


      +---------------------------------+
      | INT 23 - CTL BREAK EXIT ADDRESS |
      +---------------------------------+

INPUT: NONE. DO NOT EXECUTE THIS INTERRUPT DIRECTLY.
       SEE DOS MANUAL FOR MORE INFORMATION.


      +---------------------------------+
      | INT 24 - CRITICAL ERROR HANDLER |
      +---------------------------------+

INPUT: NONE. DO NOT EXECUTE THIS INTERRUPT DIRECTLY.
       SEE DOS MANUAL FOR MORE INFORMATION.


        +-----------------------------+
        | INT 25 - ABSOLUTE DISK READ |
        +-----------------------------+

INPUT: AL = DRIVE NUMBER (0=A)
       CX = NUMBER OF SECTORS TO READ
       DX = BEGINNING LOGICAL SECTOR NUMBER
       DS:BX = TRANSFER ADDRESS
OUTPUT: AL = DOS ERROR CODE IF CARRY SET
       AH = BIOS ERROR CODE IF CARRY SET:
            80 = ATTACHMENT FAILED TO RESPOND
            40 = SEEK FAILED
            08 = BAD CRC ON DISKETTE
            04 = SECTOR NOT FOUND
            03 = WRITE PROTECT
            02 = OTHER ERROR
  NOTE: CALLER FLAGS RETURNED ON STACK


        +-----------------------------+
        | INT 26 - ABSOLUTE DISK WRITE |
        +-----------------------------+

INPUT: AL = DRIVE NUMBER (0=A)
       CX = NUMBER OF SECTORS TO WRITE
       DX = BEGINNING LOGICAL SECTOR NUMBER
       DS:BX = TRANSFER ADDRESS
OUTPUT: AL = DOS ERROR CODE IF CARRY SET
       AH = BIOS ERROR CODE IF CARRY SET:
            80 = ATTACHMENT FAILED TO RESPOND
            40 = SEEK FAILED
            08 = BAD CRC ON DISKETTE
            04 = SECTOR NOT FOUND
            03 = WRITE PROTECT
            02 = OTHER ERROR
  NOTE: CALLER FLAGS RETURNED ON STACK


      +-----------------------------------+
      | INT 27 - TERMINATE BUT STAY RESIDENT |
      +-----------------------------------+

 INPUT: DS:DX = PROGRAM'S LAST ADDRESS PLUS ONE
OUTPUT: NONE


        +-----------------------------+
        | INT 2F - PRINT.COM INTERFACE |
        +-----------------------------+

AL = 00 - GET INSTALLED STATE
    INPUT: NONE
   OUTPUT: AL = 00 IF NOT INSTALLED, OK TO INSTALL
           AL = 01 IF NOT INSTALLED, NOT OK TO INSTALL
           AL = FF IF INSTALLED

AL = 01 - SUBMIT FILE
    INPUT: DS:DX = POINTER TO A SUBMIT PACKET CONSISTING
                   OF THE LEVEL (BYTE) AND A DWORD
                   POINTER TO AN ASCIIZ PATH/FILENAME
   OUTPUT: NONE

AL = 02 - CANCEL FILE
    INPUT: DS:DX = POINTER TO ASCIIZ FILENAME TO CANCEL
   OUTPUT: NONE                          |
                                         |
AL = 03 - CANCEL ALL FILES      | ---- ERROR CODES ----
    INPUT: NONE                 |
   OUTPUT: NONE                 | 01 - INVALID FUNCTION
                                | 02 - FILE NOT FOUND
AL = 04 - PAUSE / RETURN STATUS | 03 - PATH NOT FOUND
    INPUT: NONE                 | 04 - TOO MANY FILES
   OUTPUT: DX = ERROR COUNT     | 05 - ACCESS DENIED
           DS:SI = POINTER TO   | 08 - QUEUE FULL
                   PRINT QUEUE  | 09 - BUSY
                                | 0C - NAME TOO LONG
AL = 05 - END OF STATUS         | 0F - INVALID DRIVE
    INPUT: NONE                 |
   OUTPUT: AX = ERROR CODE      |


          +-------------------------------+
          | PROGRAM SEGMENT PREFIX LAYOUT |
          +-------------------------------+

  BYTE                    FUNCTION

 00-01 -  INT 20 INSTRUCTION
 02-03 -  TOP OF MEMORY IN SEGMENT (PARAGRAPH) FORM
   04  -  RESERVED
 05-09 -  LONG CALL TO THE DOS FUNCTION DISPATCHER
(06-07 -  NUMBER OF BYTES AVAILABLE IN THIS SEGMENT)
 0A-0D -  TERMINATE ADDRESS (IP,CS)
 0E-11 -  CTL-BREAK EXIT ADDRESS (IP,CS)
 12-15 -  CRITICAL ERROR EXIT ADDRESS (IP,CS)
 16-2B -  USED BY DOS
 2C-2D -  SEGMENT ADDRESS OF THE ENVIRONMENT
 2E-5B -  USED BY DOS
 5C-6B -  FORMATTED AS A STANDARD UNOPENED FCB
 6C-7F -  FORMATTED AS A STANDARD UNOPENED FCB
   80  -  NUMBER OF CHARACTERS ENTERED AFTER FILENAME
 81-FF -  ALL CHARACTERS ENTERED AFTER THE FILENAME


                +---------------------+
                | DIRECTORY STRUCTURE |
                +---------------------+

  BYTE              DESCRIPTION

   00  -  FILENAME STATUS:
             00 = FILENAME NEVER USED
             05 = FIRST CHARACTER OF FILENAME IS E5
             E5 = FILE HAS BEEN ERASED
             2E = THIS IS A SUBDIRECTORY ENTRY
 00-07 -  FILENAME, LEFT JUSTIFIED
 08-0A -  FILENAME EXTENTION, LEFT JUSTIFIED

   0B  -  FILE'S ATTRIBUTE:
             +-+-+-+-+-+-+-+-+
             |7|6|5|4|3|2|1|0| BYTE 0B
             +-+-+-+-+-+-+-+-+
              | | | | | | | |
              | | | | | | | +----- READ ONLY
              | | | | | | +------- HIDDEN
              | | | | | +--------- SYSTEM
              | | | | +----------- VOLUME LABEL
              | | | +------------- SUBDIRECTORY
              | | +--------------- ARCHIVE
              | +----------------- UNUSED
              +------------------- UNUSED
 0C-15 -  RESERVED BY DOS
```

16-17 - TIME THE FILE WAS CREATED OR LAST UPDATED:



18-19 - DATE THE FILE WAS CREATED OR LAST UPDATED:

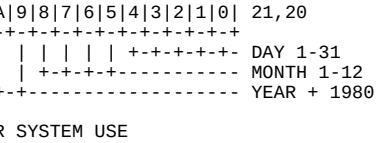


1A-1B - STARTING CLUSTER NUMBER OF THE FIRST CLUSTER OF THE FILE

1C-1F - FILE SIZE IN BYTES (LOW ORDER FIRST)

FILE CONTROL BLOCK (FCB) LAYOUT

BYTE	FUNCTION
-7	FLAG CONTAINING FF IF THIS IS AN EXTENDED FCB
-6 TO -2	RESERVED
-1	DIRECTORY ATTRIBUTE BYTE
0	DRIVE NUMBER (0 FOR DEFAULT DRIVE)
1 TO 8	FILENAME, LEFT JUSTIFIED WITH TRAILING BLANKS
9 TO 11	FILENAME EXTENSION, LEFT JUSTIFIED W/BLANKS
12,13	CURRENT BLOCK NUMBER RELATIVE TO START OF FILE
14,15	LOGICAL RECORD SIZE IN BYTES
16 TO 19	FILE SIZE IN BYTES
20,21	DATE THE FILE WAS CREATED OR LAST UPDATED



22 TO 31	RESERVED FOR SYSTEM USE
32	CURRENT RELATIVE RECORD NUMBER WITHIN CURRENT BLOCK
22 TO 36	RELATIVE RECORD NUMBER FROM START OF FILE

NOTE: POINT TO BYTE 0 FOR STANDARD FILE CONTROL BLOCK,
POINT TO BYTE "-7" FOR EXTENDED FILE CONTROL BLOCK

BOOT SECTOR LAYOUT

OFFSET	SIZE	DESCRIPTION
00	3 BYTES	JUMP TO EXECUTABLE CODE
03	8 BYTES	OEM NAME AND VERSION
0B	1 WORD	BYTES PER SECTOR
0D	1 BYTE	SECTORS PER ALLOCATION UNIT
0E	1 WORD	RESERVED SECTORS (STARTING AT 0)
10	1 BYTE	NUMBER OF FATS
11	1 WORD	NUMBER OF ROOT DIR. ENTRIES ALLOWED
13	1 WORD	TOTAL NUMBER OF SECTORS IN MEDIA
15	1 BYTE	MEDIA DESCRIPTOR
16	1 WORD	NUMBER OF SECTORS PER FAT
18	1 WORD	SECTORS PER TRACK
1A	1 WORD	NUMBER OF HEADS
1C	1 WORD	NUMBER OF HIDDEN SECTORS

14 ASCII Tabelle

Ein ASCII-Wert besteht aus einem Byte. Die Spalten der nachfolgenden Tabelle ergeben die oberen 4 Bits, die Zeilen die unteren 4 Bits des Bytes. Alle Werte sind in HEX angegeben.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	Ctrl-@ (NUL)	Ctrl-P (DLE)	SPACE	0	@	P	`	p
1	Ctrl-A (SOH)	Ctrl-Q (DC1)	!	1	A	Q	a	q
2	Ctrl-B (STX)	Ctrl-R (DC2)	"	2	B	R	b	r
3	Ctrl-C (ETX)	Ctrl-S (DC3)	#	3	C	S	c	s
4	Ctrl-D (EOT)	Ctrl-T (DC4)	$	4	D	T	d	t
5	Ctrl-E (ENQ)	Ctrl-U (NAK)	%	5	E	U	e	u
6	Ctrl-F (ACK)	Ctrl-V (SYN)	&	6	F	V	f	v
7	Ctrl-G (BEL)	Ctrl-W (ETB)	'	7	G	W	g	w
8	Ctrl-H (BSP)	Ctrl-X (CAN)	(	8	H	X	h	x
9	Ctrl-I (TAB)	Ctrl-Y (EM)	)	9	I	Y	i	y
A	Ctrl-J (LF)	Ctrl-Z (SUB)	*	:	J	Z	j	z
B	Ctrl-K (VT)	Ctrl-[(ESC)	+	;	K	[	k	{
C	Ctrl-L (FF)	Ctrl-\ (FS)	,	<	L	\	l	|
D	Ctrl-M (CR)	Ctrl-] (GS)	-	=	M	]	m	}
E	Ctrl-N (SO)	Ctrl-^ (RS)	.	>	N	^	n	~
F	Ctrl-O (SI)	Ctrl-_ (US)	/	?	O	_	o	DEL

Ein Zeichen auf einem ASCII-Bildschirm besteht immer aus 2 Bytes. Das 1. Byte (LSB) ergibt sich aus dem ASCII-Wert (siehe Tabelle oben), das 2. Byte (MSB) spezifiziert die Farb-Attribute und ergibt sich aus der folgenden Tabelle:

0	1	2	3	4	5	6	7
BLACK	BLUE	GREEN	CYAN	RED	MAGENTA	BROWN	WHITE

Anmerkungen:

Setzen von Bit 4 bzw. Bit 7 bewirkt "hell/intensiv" (wobei helles Schwarz = Dunkles Grau)