x86-Assemblerprogrammierung

von
Michael Röhrs
(Ergänzend zum Vortrag am 25.04.01)

Einleitung

Die Familie der x86-Prozessoren gehört zur Klasse der CISC-Prozessoren („Complex Instruction Set Computer“). Eine vollständige Beschreibung der Architektur dieser Prozessoren ist sehr umfangreich. In den folgenden Kapiteln soll daher nur ein Überblick über einige wichtige Aspekte dieser Architektur und ein Eindruck von der Komplexität der Prozessoren gegeben werden. Der Text bezieht sich im wesentlichen auf den 80386-Prozessor und die nachfolgenden Modelle, Abweichungen zu den Vorgängermodellen 8086 und 80286 sind nur teilweise ergänzend hinzugefügt.

1. Register

Die Programmungebung im 80386er Prozessor und den Folgemodellen besteht aus acht 32-Bit Allzweckregistern, sechs 16-Bit Segmentregistern, die zur Speicherverwaltung verwendet werden, einem 32-Bit Statusregister, einem 32-Bit Befehlszeiger sowie einem 32-Bit Adressbus mit dem entsprechenden linearen Adressraum bestehend aus 2**32 Adressen.

Die 8 Allzweckregister können durch den Programmierer frei verwaltet werden, sind jedoch durch einige Befehle implizit für bestimmte Inhalte vorgesehen. Beispielsweise dienen BP (Base Pointer) und SP (Stack Pointer) der Stackverwaltung , CX (Count Register) dient als Zähler für den Schleifenbefehl loop. Die Allzweckregister erlauben einen Zugriff auf die unteren 2 Byte durch die letzten beiden Buchstaben der Bezeichnung (AX,BX,...). Die Register EAX,EBX,ECX,EDX ermöglichen darüberhinaus einen Zugriff auf beide Bytes im unteren Bereich durch die Bezeichnungen AH,AL,BH,BL usw. (H=High,L=Low) (s.Abb.1).

Das Statusregister umfaßt im wesentlichen 6 Satusflags, 10 Systemflags sowie ein Controlflag. Die Statusflags enthalten Informationen über die letzten logischen und arithmetischen Befehle. So wird das „Zero Flag“ gesetzt, wenn das Ergebnis einer Operation Null ist, „Carry-Flag“ und „Overflow-Flag“ dokumentieren Überläufe. Die Statusflags dienen insbesondere als Grundlage für bedingte Befehle.
Die Systemflags kontrollieren intern die Prozessessorarbeit und werden in der Regel durch den Programmierer nicht benötigt. Das Direction Flag zeigt an, in welcher Richtung Strings bearbeitet werden und kann durch den Programmierer durch Befehle verändert werden (wie einige andere Flags ebenfalls).

In den Vorgängermodellen 8086 und 80286 wurden ausschließlich 16-Bit Register verwendet sowie 4 statt 6 Segmentregister. Der Adressbus war 20 Bit breit und konnte nur durch eine spezielle Speicherverwaltung über die 16 Bit Register genutzt werden, die im folgenden genauer erläutert wird.

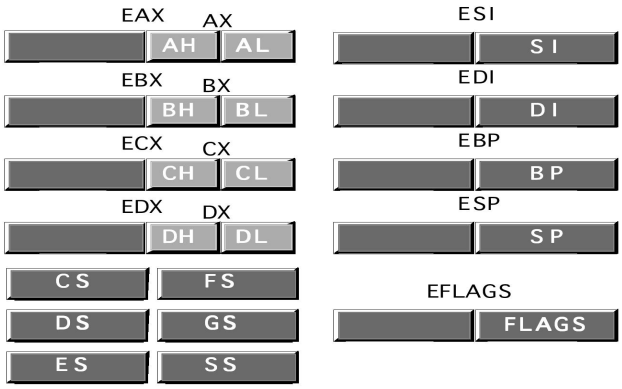

Abb.1: Schaubild zu den für den Programmierer sichtbaren Registern der x86-Prozessoren. Bei den 8086/286-Modellen sind bei den Allzweckregistern und dem Statusregister jeweils nur die unteren 2Byte vorhanden (AX,SI,...). Darüberhinaus fehlen die Segmentregister FS und GS.

2. Speicherorganisation

Der physikalische Speicher ist byteweise adressierbar. Als Little-Endian- Prozessoren speichern die x86er höherwertige Bytes an höheren Adressen ab. Der 80386er und die Nachfolger können in drei verschiedenen Operationsmodi betrieben werden: Real-Adress-Mode, Protected Mode und System Management Mode (SMM).
Im Real-Adress-Mode wird im wesentlichen der 8086-Prozessor simuliert und der Speicher wird im sogenannten Real-Adress-Memory-Model verwaltet.
Im Protected Mode stehen alle Verbesserungen im vollen Umfang zur erfügung, d.h. alle obigen Register, neue Befehle und erweiterte Speichermodelle.
Der SMM dient der Systemverwaltung durch das Betriebssystem und erlaubt beispielsweise Powermanagement- und Sicherheitsfunktionen.

Im folgenden sollen die Möglichkeiten der Speicherverwaltung im Real-Adress-Mode und im Protected Mode beschrieben werden.

2.1 Speicherorganisation im Real-Address-Mode

Durch die Simulation des 8086-Prozessors stehen in diesem Modus nur 16 Bit Register und ein 20 Bit Adressbus (1MB Speicher) zur Verfügung. Diese Konstellation erfordert eine spezielle Verwaltung des Speichers, da eine 20 Bit Adresse nicht in einem Register abgelegt werden kann. Man unterteilt daher den Speicher in zusammenhängende Speicherbereiche, die Segmente. Jedes Byte wird durch eine sogenannte logische Adresse adressiert. Diese besteht aus zwei Teilen: Im ersten Teil wird die Adresse des ersten Byte im jeweiligen Segment kodiert, den zweiten Teil bildet der Abstand des zu adressierenden Bytes vom ersten Byte des Segments. Kurz:

Logische Adresse = Segment:Offset

Der Code für die Adresse des ersten Bytes des Segments wird dabei durch die oberen 16 Bit der linearen Adresse dieses Bytes realisiert. Bei der Adressberechnung werden diese 16 Bit durch vier Nullen zu einer 20 Bit Adresse ergänzt und das Offset hinzu addiert. Diese Vorgehensweise erlaubt also nur solche Segmente, die bei einer linearen Adresse mit vier Nullen in den niederwertigsten Bits beginnen.
Der erste Teil einer logischen Adresse wird in den Segmentregistern gespeichert. Es werden dabei in der Regel getrennte Segmente für Daten,Befehlscode und den Stack angelegt und dabei entsprechend DS (Data Segment), CS (Code Segment) oder SS (Stack Segment) als Basisregister verwendet. Durch ES (Extra Segment) kann ein weiteres Datensegment realisiert werden. Das Offset wird in den Allzweckregistern gespeichert. Deren Größe von 16 Bit (im Real-Adress-Mode) ermöglichen eine Segmentgröße von 64 KB. Die einzelnen Segmente

dürfen sich dabei überlappen, so daß einer linearen Adresse meherere logische Adressen zugeordnet sein können.

2.2 Speicherorganisation im Protected Mode

Im Protected Mode steht ein 32 Bit Adressbus und entsprechend 4GB Speicher zur Verfügung. Der Speicher wird ebenfalls segmentiert und mittels logischer Adressen verwaltet, jedoch bestehen einige Unterschiede zur Organisation im Real-Adress-Modus. Zum einen erlauben die 32 Bit breiten Allzweckregister ein 32-Bit Offset und somit eine Segmentgröße von 4GB. Zum anderen enthält der erste Teil der logischen Adresse nicht die lineare Adresse des ersten Byte des Segments, sondern einen Verweis auf einen weiteren Speicherbereich, den sogenannten „Segment Descriptor“, der die 32 Bit-Basisadresse des Segments sowie weitere Informationen enthält. Die 32 Bit breite lineare Adresse berechnet der Prozessor im Protected Mode nun durch Addition der Basisadresse des Segments und des Offsets.

Die „Segment Descriptors“ sind in Arrays organisiert, die als „Segment Descriptor Tables“ bezeichnet werden. Man unterscheidet zwischen „Global Descriptor Table“ (GDT) für Betriebssysteme und „Local Descriptor Tables“ (LDT) für andere Programme. Die Adressierung der einzelnen Elemente dieser Arrays funktioniert analog zur logischen Adressierung. Die Adressen des ersten Bytes eines solchen Arrays wird in einem speziellen Register abgelegt, dem „Global Descriptor Table Register“ (GDTR) bzw. dem „Local Descriptor Table Register“ (LDTR). Der Index des adressierten Elements wird in den oberen 13 Bit eines Segmentregisters gespeichert. Da die einzelnen „Segment Descriptors“ eine einheitliche Größe von 8 Bytes besitzen, kann aus Index und Basisadresse die gewünschte Adresse ermittelt werden. Ein weiteres Bit im Segment Register kennzeichnet das Basisregister (LDTR oder GDTR). Die verbleibenden 2 Bit werden zur Kodierung von Zugriffsrechten verwendet.

In den „Segment Descriptors“ sind die Attribute der Segmente kodiert. Am wichtigsten ist dabei die 32Bit Basisadresse des Segments sowie das Segmentlimit. Darüberhinaus werden weitere Merkmale wie beispielsweise die Datengröße (16 Bit/ 32 Bit) festgelegt.

Die Speicherorganisation im Protected Mode erlaubt die Realisierung zweier unterschiedlicher Speichermodelle. Wird der Speicher in sich nicht überlappende Segmente unterteilt, spricht man vom „Multisegment Memory Model“. Die andere Möglichkeit ist die Simulation eines „flachen“ Speichers durch vollständige Überlappung der Segmente im sogenannten „Flat Memory Model“.

2.3 Paging

„Paging“ ist ein Mechanismus zur Implementation von virtuellem Speicher. Es wird ein linearer Adressraum verwaltet, der größer ist als der physikalische Adressraum. Dazu werden Daten auf Festplatte ausgelagert und eine zusätzliche Übersetzung einer linearen Adresse in die entsprechende physikalische Adresse vorgenommen.

Zunächst werden die Segmente im linearen Adressraum in Seiten mit einer Größe von 4KB unterteilt. Den Seiten werden Bereiche im Hauptspeicher und auf Festplatte zugeordnet. Diese Zuordnung wird in einem Seitenverzeichnis und in Seitentabellen vermerkt, welche durch das Betriebssystem verwaltet werden und eine Übersetzung der linearen Adressen ermöglichen. Wenn auf eine Seite zugegriffen werden soll, die sich nicht im Hauptspeicher befindet, wird die Programmausführung unterbrochen und die entsprechende Seite geladen.

Das Paging steht nur im Protected Mode zur Verfügung.

3. Adressierung

Es gibt drei grundsätzlich verschiedene Möglichkeiten der Speicherung von Operanden: Die Speicherung in einem Register („Register Addressing Mode“) , im Befehlscode („Immediate Addressing Mode“) und im Hauptspeicher („Memory Addressing Mode“). Die Speicheradressierung gliedert sich in direkte und indirekte Adressierung. Bei der direkten Adressierung ist die Speicheradresse Teil des Befehlscodes. Die Adressangabe erfolgt dabei im Assemblercode in der Regel über ein Label, das der Assembler dann in eine Adresse übersetzt.
Bei der indirekten Adressierung wird die Adresse nach einem festen Schema berechnet. Das Segment ist dabei bereits implizit durch den Befehl festgelegt, so dass die Berechnung sich nur auf das Offset bezieht. In *Abb.2* ist dieses Berechnungsschema für die 32-Bit-Adressierung mit den zugehörigen Registern dargestellt. Die Registerinhalte und das „Displacement“ werden dabei als 2er-Komplement Werte interpretiert.

Die verschiedenen Möglichkeiten der indirekten Adressierung erlauben eine effiziente Implementation verschiedener Datenstrukturen. Beispielsweise können die Elemente eines Arrays durch ein festes „Displacement“ zum ersten Byte des Arrays und einen Index adressiert werden. Durch Verwendung eines Skalenfaktors kann hierbei die Elementgröße im Array berücksichtigt werden.

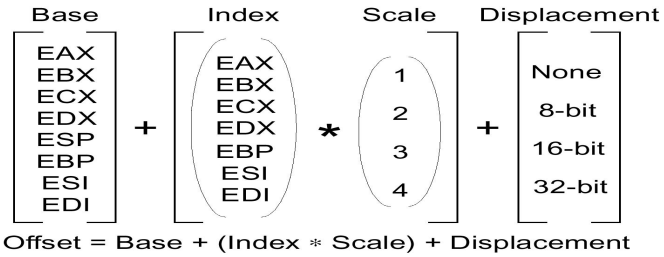

Abb.2: Schema zur Offset-Berechnung bei der indirekten Speicheradressierung (32 Bit). Bei 16-Bit-Adressierung können nur BX oder BP als Basis- und nur SI oder DI als Indexregister verwendet werden, der Skalenfaktor ist nicht verfügbar und das „Displacement“ auf 16 Bit beschränkt.

4. Stack

Der Stack ist eine LIFO-Datenstruktur („Last-In-First-Out“), in der temporäre Daten abgelegt werden. In den x86-Prozessoren spielt der Stack eine wichtige Rolle, weil durch die geringe Anzahl von Allzweckregistern zum einen eine häufige Sicherung von Registerinhalten nötig ist und zum anderen die Übergabe von Parametern an Unterprogramme in der Regel über den Stack vorgenommen werden muss. Durch das SS-Register wird für den Stack ein gesondertes Segment eingerichtet. Im SP-Register („Stack Pointer“) ist stets das Offset zum letzten Eintrag enthalten („Top of the Stack“). Die Befehle „push (source)“ und „pop (destination)“ haben einen impliziten Bezug zu dieser Adresse und führen hier Lese- und Schreib-operationen durch.
Der „push“-Befehl führt zunächst zu einer Dekrementierung des SP und dann zu einer Schreiboperation an dieser Adresse .Der Stack wächst in den x86-Prozessoren also zu niedrigeren Adressen hin. Die Höhe der Dekrementierung hängt von der Breite der Elemente im Stack ab. Diese kann durch ein Bit im „Segment Descriptor“ für das Stack-Segment (vgl.

Kap. 2.2) auf 16 Bit (Dekrementirung um 2) oder auf 32 Bit (Dekrementierung um 4) eingestellt werden.
Der „pop“-Befehl führt entsprechend zunächst eine Leseoperation durch, schreibt den Inhalt in ein angegebenes Register oder einen Speicherplatz und inkrementiert anschließend den SP entsprechend der Datenbreite, sodass er auf das vorhergehende Elemente zeigt.

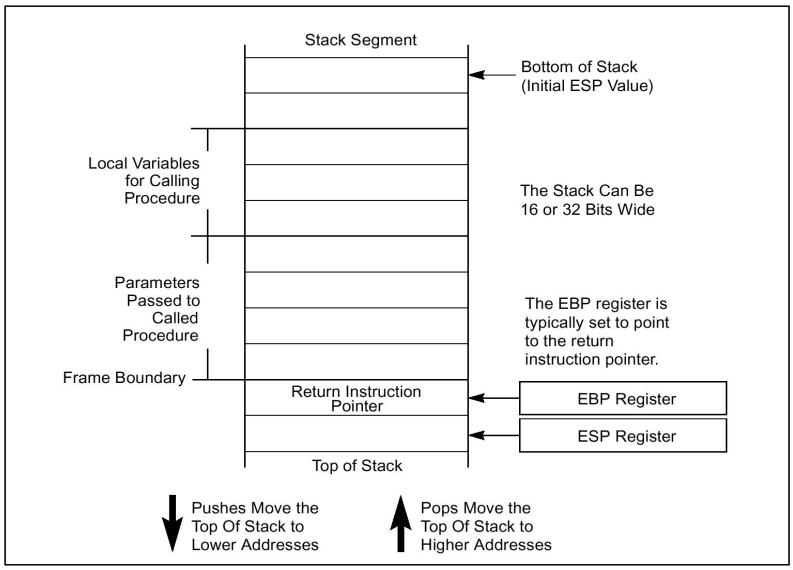

Abb.3: Schaubild zum Stack: Der Zugriff auf die Einträge wird bei Prozeduraufrufen über das (E)BP-Register als Basis-Register vorgenommen, dass hier auf den „Return Instruction Pointer“ zeigt.

4.1 Parameterübergabe

Der Aufruf von Prozeduren wird durch den „call (Prozedurname)“ –Befehl bewerkstelligt. Es wird der aktuelle Eintrag im IP-Register („Instruction Pointer“) auf dem Stack abgelegt, um das Programm nach Beendigung der Prozedur an der richtigen Stelle fortsetzen zu können („Return Instruction Pointer“). Anschließend wird das Offset zum ersten Befehl in der Prozedur in das IP-Register geschrieben.
Wenn Parameter an die Prozedur übergeben werden sollen, können diese vor dem Prozedur-aufruf auf dem Stack abgelegt werden. Um den Zugriff auf diese Parameter zu erleichtern, wird in einer Prozedur der aktuelle Stackpointer im BP-Register („Base Pointer“) gesichert, dessen alter Inhalt zuvor ebenfalls auf dem Stack gespeichert wird. Diese Vorgehensweise erlaubt dann den Zugriff auf die Parameter in der Form „Base+Displacement“ mit dem BP-Register als Basis. Auch lokale Variable einer Prozedur werden in der Regel auf dem Stack abgelegt und können über den BP adressiert werden. Den Bereich im Stack, in dem lokale Variable, BP,SP und Parameter abgelegt sind bezeichnet man als „Stack Frame“.

Die Rückkehr zur aufrufenden Prozedur wird über den „ret“-Befehl („Return“) erledigt. Dieser liest den alten Befehlszeiger wieder in das IP-Register ein und inkrementiert den SP um einen vom Programmierer festzulegenden Wert. Es ist Aufgabe des Programmierers dafür zu sorgen, dass der SP auf den „Return Instruction Pointer“ zeigt, wenn der „ret“-Befehl ausgeführt wird.

5. Datentypen und Befehlssatz

Auf der Ebene der Assemblerprogrammierung wird bei der Definition von Variablen keine Vorgabe für deren Interpretation gegeben. Die Befehle jedoch beinhalten eine feste Interpretation der Daten. In den x86-Prozessoren sind auf diese weise vergleichsweise viele verschiedene Datentypen mit den zugehörigen Befehlen vorhanden. Als Beispiele seien neben den „gewöhnlichen“ Datentypen wie Signed/Unsigned Integer und Floating Point noch BCD Integers („Binary Coded Decimal“), Near Pointer (Offset) und Far Pointer (Segment:Offset), MMX- und SIMD-Datentypen genannt.
Der Befehlssatz gliedert sich entsprechend der Datentypen in die Kategorien Integer,Floating Point, MMX, SIMD sowie System-Befehle. Die einzelnen Befehle sind dabei zum Teil sehr spezialisiert. So existiert mit „loop“ beispielsweise bereits auf dieser Ebene ein Schleifenbefehl, der das (E)CX-Register als Zähler verwendet.

5. Befehlscodierung

Das allgemeine Befehlsformat ist in *Abb.4* dargestellt. Das erste Feld in diesem Format ist der „Prefix“. Dieser erlaubt verschiedene Modifikationen der Befehle, beispielsweise die Umgehung der impliziten Segmentvorgabe für Befehle durch einen Segment-Override-Prefix. Hinter dem Opcode kommen mir Mod R/M, SIB und Displacement drei Felder für die Adressierung. Im Mod R/M-Feld wird zunächst der Adressierungsmodus festgelegt. Je nach Modus werden die folgenden Felder dann unterschiedlich interpretiert, z.B. können im verbleibenden Bereich des Mod R/M-Bytes ein oder zwei Register als Operanden kodiert und/oder der Adressierungsmodus eines Speicheroperanden festgelegt werden. Bei indirekter Speicher-Adressierung werden die Register und der Skalenfaktor im SIB-Byte (Scale,Index,Base) und das „Displacement“ im gleichnamigen Feld festgelegt. Im „Immediate“- Feld kann der Bezeichnung entsprechend direkt ein Operand (bis 32 Bit) gespeichert werden.

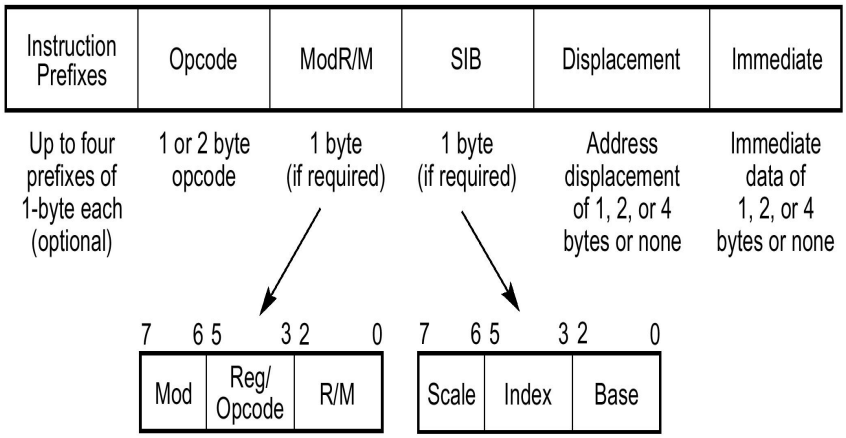

Abb.4: Allgemeines Befehlsformat: Die einzelnen Felder sind optional. Die einzelnen Befehle sind daher sehr unterschiedlich in der Code-Länge.

6. Literatur

- Dandamudi: „Introduction To Assembly Language Programming“
- „Intel Architecture Developer‘s Manual“, Band 1 bis 3 (Download bei http://developer.intel.com/design/pentium/manuals)
- Randall Hyde: „The Art Of Assembly Language Programming“ (Download bei http://webster.cs.ucr.edu/Page_asm/ArtofAssembly/HardCopy.html)