Assembler–Tutorial

Grundlagen und Theorie

http://www.amok.notrix.de

Vorwort

In diesem Tutorial werden wir Grundlagenarbeit betreiben. Diese besteht daraus, daß wir Begriffe wie Register, oder „Stack“ erläutern. Wir werden uns auch den Segmenten eines Programmes zuwenden, in Bezug auf den 8086 bzw. den neueren Prozessoren mit Windows 9x als Betriebssystem. Dieses Kapitel habe ich leider noch in keinem Tutorial vorgefunden, obwohl es sehr viel zum Verständnis eines Programmes auf der Assembler–Ebene beiträgt. Doch genug für den Anfang. Beginnen wir.

Zahlensysteme und deren Darstellung

Wie jeder PC Anfänger weiß, kann der Computer nur mit Einsen oder Nullen umgehen. Dieses Zahlensystem nennt man das Binär- oder Dualsystem. Weiterhin existieren das Dezimalsystem, welches uns am geläufigsten ist, und mit dem wir uns Tag für Tag auseinandersetzten, sowie das Hexadezimal–System (auf der Basis der Zahl 16) und das Oktal–System (auf der Basis 8). Ein genauere Einführung ist wohl eher uninteressant, daher lasse ich sie weg.

Nur eine Sache sollt geklärt werden: Die Darstellung negativer Zahlen im Binärformat. Damit man eine Zahl im Binärformat mit einem Vorzeichen „behaften“ kann, wurde folgende Lösung erdacht: Man bestimmt ein Bit, welches angibt, ob die Zahl - oder + ist. Dadurch verschiebt sich allerdings die Menge der darstellbaren Zahlen: So kann man mit einem 8 Bit großen Wert 2^8 = 265 mögliche Zahlen darstellen. Somit lassen sich bei einem vorzeichenlosen Wert die Zahlen 0 - 255 darstellen. Doch sobald ein Vorzeichen Bit dazu kommt, hat man nur noch 7 Bit übrig (Das erste Bit, ist das Vorzeichen Bit). Somit bleiben folgende Zahlen übrig: 00000000-01111111 für 0 bis 127; sowie 10000000-11111111 für die Zahlen -1 bis -127. Somit geht zwar nicht direkt die Menge der darstellbaren Zahlen zurück, doch es verschiebt sich der Bereich von 0 bis 255 auf -127 bis 127.

Will man nun eine positive Zahl in eine negative verwandeln, bildet man ein so genanntes Zwei–Komplement. Hierzu ein Beispiel: 5 ist im Dualsystem gleich 00000101. Um die Zahl -5 darzustellen, dreht man jedes Zeichen um; so wird aus 00000101 11111010. Das ist ein Zwei–Komplement.

Es werden solche Begriffe wie „Word“ („Wort“) oder „DWord“ („Doppelwort“) vorkommen, daher möchte ich sie gleich klären:

„Byte“: Ein 8 Bit großer Wert.

Ein Byte ist folgender maßen aufgeteilt:

7	6	5	4	3	2	1	0

Die Zählung der Bits beginnt bei 0 und endet bei 7. Das 8. Bit (Bit Nr. 7) ist das Vorzeichenbit.

„Word“: Ein 16 Bit großer Wert.

„Double–Word“ („DWord“): Ein 32 Bit großer Wert.

„Quad–Word“ („QWord“): Ein 64 Bit großer Wert.

„Integer“: Ein vorzeichenbehafteter Wert, welcher die Größe von einem „Byte“, „Word“ oder „DWord“ haben kann.

„Arithmetische Operation“: Eine der 4 Grundrechenarten: Addition, Subtraktion, Multiplikation oder Division.

Noch zu beachten:

In SoftICE und W32DASM werden Zahlen in hexadezimaler Schreibweise dargestellt.

Wenn in Assembler eine Zahl in einer eckigen Klammer ('[' und ']') steht, so wird diese als Zeiger („Pointer“) auf eine Speicherstelle interpretiert.

Assembler–Befehle folgen dem Syntax <Befehl> <Ziel>, <Quelle>.

„Operanden“ werden durch eine Komma getrennt.

Die Register

Ein „Register“ ist ein Teil des Prozessors, welcher Daten beinhaltet. Diese Daten dienen dem Prozessor dazu, um auf der eine Seite die vom Programmierer angeordneten Operationen durchzuführen und auf der anderen Seite, um ein Programm überhaupt zum Laufen zu bringen. Der Prozessor hält ab dem 80386 16 Register zur Programmierung bereit. Intel unterteilt die Register wie folgt (beschrieben im „Intel Architecture Software Developer Manual“):

„General-purpose data registers“ (Allzweckregister)

EAX	„accu register“	Akkumulator
EBX	„base register“	Basisregister
ECX	„counter register“	Zählregister
EDX	„data register“	Datenregister
ESI	„source index register“	Quell–Index
EDI	„destination index register“	Ziel–Index
EBP	„base pointer register“	Basiszeiger
ESP	„stack pointer register“	Stack–Zeiger

„Segment registers“ (Segmentregister)

CS	„code segment register“
DS	„data segment register“
ES	„extra segment register“
SS	„stack segment register“
FS	„help segment 1 register“
GS	„help segment 2 register“

„Status & control registers“ (Status- und Kontrollregister)

EFLAGS	„EFLAGS–Register“ (Flag–Register)
EIP	„instruction pointer register“ (Befehlszeiger)

Das Flag–Register hat eine besondere Bedeutung, und wird gleich gesondert behandelt. Daher gelten die folgenden Aussagen nicht für dieses Register.

Die Größe der Register

Sämtliche Allzweckregister sowie die Status- und Kontrollregister haben eine Größe von 32 Bit (Gekennzeichnet durch das 'E' am Anfang). Die Segmentregister sind nur 16 Bit groß. Alle 32 Bit großen Register lassen sich aber auch als 16 Bit Register ansprechen, indem man das 'E' am Anfang des Kürzels wegläßt. Die ersten 4 Register besitzen außerdem noch eine Besonderheit (deswegen der Absatz): Sie lassen sich zusätzlich als zwei 8 Bit Register ansprechen. Das heißt folgendes:

EAX	Ein maximal 32 Bit großer Wert kann in diesem Register gespeichert werden.
AX	Bedeutet, daß nur die ersten 16 Bit gemeint sind.

Diese beiden Möglichkeiten der Behandlung haben alle 32 Bit Register gemeinsam. Die beiden folgenden weiteren Möglichkeiten beschränken sich auf EAX, EBX, ECX und EDX.

AL	Dies sind nur die ersten 8 Bit des Registers. ('L' steht für „low“).
AH	Dies sind die zweiten 8 Bit des Registers. ('H' steht für “high“).

Die Verwendung bzw. die Aufgaben der Register

Die Allzweckregister

Die ersten 4 Allzweckregister sind die, mit denen der Programmierer am häufigsten arbeitet. Sie können beliebig kombiniert werden. Doch jedes dieser Register hat spezielle Aufgabengebiete, die durch Befehle deutlich gemacht werden.

EAX	Diese Register dient hauptsächlich für arithmetischen Operationen
EBX	EBX ist dazu gedacht, um auf Speicherstellen zu zeigen
ECX	ECX wird bei Schleifen eingesetzt
EDX	I/O Zeiger

Diese Aufgaben durch Beispiele darzustellen, halte ich an diesem Punkt für falsch, denn dafür benötigt man weiteres Wissen, welches erst später dazu kommt. Daher lassen wir es, und gehen zu den nächsten Registern.

Die vier weiteren Register dienen dazu um auf spezielle Bereiche in einem Segment (was das ist, wird gleicht behandelt) zu zeigen. Da das Wissen noch fehlt, um diese Sachen genauer zu erläutern sei nur folgendes gesagt: ESI und EDI sind die einzigen weiteren Register, die manchmal vom Programmierer wie EAX, EBX, ECX, oder EDX eingesetzt werden können. ESP und EBP haben spezielle Aufgaben, die im Kapitel über den Stack behandelt werden.

Die Segmentregister

Auch für diese Register gilt, daß sie erst im späteren Teil behandelt werden können.

Status- und Kontrollregister

Das Flag–Register wird ausführlich im Kapitel „Flags“ behandelt werden. Das EIP Register ist vielleicht das wichtigste Register für den Prozessor, da es auf die Stelle im Speicher zeigt, welches den nächsten Befehl beinhaltet. EIP ist für den Programmierer recht uninteressant, da man es nur indirekt beeinflussen kann.

Flag–Register

Einige werden sich vielleicht wundern, warum ich diesem Register ein eigenes Kapitel widme. Doch das ist gerechtfertigt, wie wir gleich sehen werden.

Ein Flag ist, wie der Name schon sagt eine Flagge, die etwas signalisiert. Das Flag–Register ist 32 Bit groß, und kann damit 32 mal 'ja' oder 'nein' sagen, in Form von einer 0 für ein 'nein' bzw. für eine nicht–gesetzte Flagge; oder in Form von einer 1 für ein 'ja', bzw. für eine gesetzte Flagge.

Doch wozu diese Flaggen? Ganz einfach: Sie dienen dazu um den Ausgang bestimmter Operationen anzuzeigen, oder bestimmen das Verhalten des Prozessors, bei besonderen Aktionen.

Es ist zwar möglich mit dem Register 32 Flaggen zu besetzten, doch es gibt nur 17 Stück. Die anderen Bits sind reservierte Positionen und dürfen nicht genutzt werden. Und auch von diesen 17 „Flags“ sind für uns nur einige relevant, nämlich 9 Stück. Diese 9 sind folgende:

OF	DF	IF	TF	SF	ZF	AF	PF

OF	„Overflow Flag“
DF	„Direction Flag“
IF	„Interrupt Enable Flag“
TF	„Trap Flag“
SF	„Sign Flag“
ZF	„Zero Flag“
AF	„Auxiliary Cary Flag“
PF	„Parity Flag“
CF	„Carry Flag“

Die Bedeutung der „Flags“

Was die „Flags“ im einzelnen bedeuten, oder bewirken, möchte ich natürlich niemandem vorenthalten: Allerdings ist das „Parity Flag“, „Trap Flag“ und das „Interrupt Enable Flag“ recht uninteressant für einen „Cracker“, und daher werden sie nur kurz angeschnitten. Und auch die anderen „Flags“ sind nur selten für einen „Cracker“ wirklich wichtig (ausgenommen das „Zero Flag“), trotzdem werden sie behandelt, damit im Fall der Fälle man immer noch auf dieses Dokument zurückgreifen kann.

In dem folgenden Abschnitt werden leider solche Begriffe wie String–Operationen und Interrupt benutzt. Diese werden nicht erklärt (Um bestimmte Befehle, oder Befehlsarten zu verstehen ist es Empfehlenswert, sich ein Buch darüber zu besorgen), trotzdem müssen sie benutzt werden, um die Funktionen der „Flags“ zu klären. Ein Anfänger sollte sich nicht davon abhalten lassen, das trotzdem zu lesen, da doch einiges erklärt wird, was auch ohne Vorwissen zu verstehen ist.

Das „Carry Flag“ (Übertrage–Flag)

Diese Flag wird gesetzt, wenn nach einer arithmetischen Operation ein Überlauf statt gefunden hat. Wenn man z.B. zwei 8 Bit Werte wie 100 und 200 addiert, ist das Ergebnis 300. Doch die Zahl 300 übersteigt, die mögliche Anzahl der darstellbaren vorzeichenlosen Zahlen um 45. Um 300 darzustellen benötigt man 9 Bit. Damit diese Zahl nicht verloren geht, wird das „Carry Flag“ gesetzt; somit erhält man die 9 Bit (8 Bit, das normale Byte + das „Carry Flag“).

Das „Parity Flag“ (Paritäts–Flag)

Diese „Flag“ dient zur Fehlerüberprüfung bei einer Datenübertragung auf eine serielle Schnittstelle.

Das „Auxiliary Cary Flag“ (Übertrage–Hilfs–Flag)

Dieses Flag ist gesetzt wenn ein übertrag von Bit 3 (das 4. Bit, da das 1. Bit, Bit 0 ist) auf Bit 4 erfolgt ist. Es wird im Umgang mit dem „BCD–System“ benutzt. Dieses Zahlensystem wird hier nicht behandelt, da es im PC-Bereich kaum Verwendung findet.

Das „Zero Flag“ (Null–Flag)

Dieses Flag ist zweifellos das wichtigste für einen „Cracker“, und wahrscheinlich auch das wichtigste für einen Assembler Programmierer. Dieses Flag zeigt an, ob das Ergebnis einer Operation Null ist, oder nicht. Falls es Null ist, ist das „Zero Flag“ gesetzt, falls das Ergebnis ungleich Null ist, wird es nicht gesetzt. Eine wichtige Benutzung des „Zero Flag“ aus der Sicht eines „Crackers“ ist sicherlich, wenn aus einem Unterprogramm der Rückgabewert ermittelt wird. Doch das ist für den Anfänger noch nicht so interessant.

Das „Sign Flag“ (Vorzeichen–Flag)

Dieses Flag ist gesetzt, wenn das Ergebnis einer arithmetischen oder logischen Operation negativ ist. Falls das Ergebnis positiv ist, ist das Flag nicht gesetzt.

Das „Trap Flag“ (Einzel–Schritt–Flag)

Dieses Flag (falls es gesetzt ist) versetzt den Prozessor in einen Zustand, wo er jeden Befehl einzeln ausführt (tut er ja sowieso) und danach den Interrupt 1 aufruft. Dadurch ist es möglich Debugger einzusetzen, um dem Programmierer z.B. bei der Fehlersuche, erheblich seine Arbeit zu erleichtern.

Das „Interrupt Enable Flag“ (Unterbrechungs–Flag)

Dieses Flag gibt an, ob das Programm durch Interrupts, z.B. von der Tastatur unterbrochen werden darf. Falls es gesetzt ist, kann das Programm unterbrochen werden. Falls es nicht gesetzt ist, läßt der PC keine Unterbrechung des Programms zu.

Das „Direction Flag“ (Richtungs–Flag)

Diese Flag gibt die Richtung bei sogenannten String–Operationen an. Falls es gesetzt ist, wird die String–Verarbeitung nach aufsteigenden Adressen durchgeführt. Falls es nicht gesetzt ist, wird die String–Verarbeitung nach absteigenden Adressen durchgeführt.

Das „Overflow Flag“ (Überlaufs–Flag)

Dieses Flag ist gesetzt, wenn das Vorzeichenbit durch eine „arithmetische Operation“ zerstört ist.

Noch etwas zu negativen und positiven Zahlen

Wie wir ganz am Anfang gelernt haben, wird die Darstellung von negativen und positiven Zahlen im Dualsystem durch ein Vorzeichenbit gelöst. Doch woher soll der Prozessor wissen, ob er bei einer Addition das Bit 7 (oder das Bit 15, oder Bit 31) als Vorzeichen, oder als Teil einer Zahl, die kein Vorzeichen hat behandeln soll. So kann 10011010 gleich 154 oder -102 sein. Der Prozessor kennt den Unterschied nicht. (In SoftICE wie folgt zu überprüfen: Wenn ein Register größer als 80000000h ist, so kann man sich durch den Befehl '? <Registername>' den Wert des Registers in Hex und Dezimal anzeigen lassen. Dadurch das der Wert des Registers mindestens 80000000h ist, wird hinter der Dezimal Zahl noch eine weitere Zahl in Klammern angezeigt. Diese Zahl ist negativ, da SoftICE bei ihr das höherwertigste Bit als Vorzeichen interpretiert hat. Da SoftICE genauso wenig wie der Prozessor weiß, ob die Zahl mit Vorzeichen oder ohne gedacht war, muß SoftICE beides in Betracht ziehen und zeigt beide Möglichkeiten an.) Doch was soll das Ganze dann, wenn der Prozessor den Unterschied nicht kennt? Ganz einfach: Es kommt auf den Befehl an. So z.B. führt der Befehl MUL eine Multiplikation durch, ohne dabei auf das höherwertigste Bit zu achten. Im Gegensatz zu IMUL, wo das höherwertigste Bit als Vorzeichen gewertet wird.

Die Segmente

Die Segmente beim 8086

Einige werden sich jetzt vielleicht Fragen, warum nehmen wir hier die Segmente beim 8086 durch, wobei doch in der heutigen Zeit dieser Prozessor hoffnungslos veraltet ist. Die Antwort ist relativ einfach: Das was wir jetzt besprechen gilt heute immer noch. Also aufgepaßt.
Im vorherigen Abschnitt habe ich bei den Segment–Registern darauf verwiesen, daß wir diese erst später behandeln. Und jetzt ist später. Wenn ich hier von Registern spreche, muß man noch folgendes Wissen: Beim 8086 gab es noch keine 32 Bit großen Register, sondern es waren alle 16 Bit groß. Das bedeutet es gab kein Register mit dem Namen EAX, sondern AX.

Der Bus

Beim 8086 gibt es folgende Busse: Der Steuerbus, der Adreßbus und der Datenbus. Davon interessieren uns nur der Adreßbus und der Datenbus. Der Steuerbus dient nur zur Regelung der Datenübertragung am Computer (nicht, das es nicht wichtig wäre, aber er ist für uns nicht relevant im Bezug auf die Segmente).
Beim 8086 ist der Adreßbus 20 und der Datenbus 16 Bit breit. Und das ist nicht gut. Der Adreßbus dient dazu eine Speicherstelle im Arbeitsspeicher zu adressieren. Der Adreßbus ist 20 Bit breit und 2^20 ergibt 1048576 Byte bzw. 1024 KB und dies wiederum ist 1 MB. Damit wäre klar, daß der 8086 maximal ein MB ansprechen konnte. Von der Adresse 0 bis 1048575.
Der Datenbus hat wiederum die Aufgabe den Inhalt der Stelle im RAM, der vom Adreßbus definiert worden ist, zu transportieren; entweder zum Prozessor hin, und von ihm weg. Dabei transportiert er Code (also ausführbare Befehle) und Daten (Informationen, welches das Programm benötigt, um mit den Befehlen vernünftig zu arbeiten).

Das Problem

Okay, wir haben einen 20 Bit großen Adreßbus und einen 16 Bit großen Datenbus. Dadurch können die Register maximal 16 Bit groß sein. Eine Adresse, auf die man mittels des Adreßbus zugreifen will, muß ja irgendwo gespeichert sein. Und dies irgendwo sind die Register. Doch wie will man mit einem 16 Bit großem Wert voll auf 20 Bit (= 1 MB) zugreifen? Genau das ist das Problem. Wenn wir z.B. auf die Adresse 10000h zugreifen wollen, welche 20 Bit zur Darstellung benötigt, so ist es nicht möglich diese Zahl in einem Register zu speichern, da ein Register beim 8086 maximal 16 Bit groß ist.

Die Lösung

Dieses Problem bedarf einer Lösung, die wie folgt aussah: Man benutzt 2 Register. Dies klingt nicht sehr einfallsreich; doch wie wir jetzt sehen werden, ist es nicht ganz so einfach. Wir haben einen Adreßraum vom 1 MB (= 1024 kB). Um diese 1 MB sinnvoll aufzuteilen auf 2 Register, dividierte man 2^20 (= 1024 KB; die maximale Größe des Adreßraums) durch 2^16 (die maximale Größe eines Registers) und erhält 16. Somit war klar, daß man den Adreßraum in Blöcke zu je 16 Byte aufteilte. Das bedeutet, daß der Adreßraum ab sofort nicht mehr nur 1024 KB groß ist, sondern auch aufgeteilt ist in Blöcke zu je 16 Byte. Diese Blöcke heißen Paragraphen. Doch wieviele dieser Paragraphen gibt es denn? Ganz einfach: Man dividiert die 2^20 durch 16, und bekommt die Zahl 65.536.
Nochmals zusammengefaßt: Man teilte den Adreßraum in Segmente auf, die je mindestens 16 Byte groß sind, deren Anzahl 65.535 beträgt. Soweit, so gut. Solch eine Zahl, die zwischen 0 und 65.535 groß sein kann, wird in den Segment–Registern gespeichert. Also CS, DS, ...
Doch das konnte ja nicht alles sein, denn mit Hilfe von diesen Segment–Registern, konnte man sich zwar im Adreßraum im Abstand von 16 Byte "bewegen", doch man mußte ja jedes einzelne Byte explizit ansprechen können (und nicht nur jedes 16.). An diesem Punkt kommen die sogenannten Offset–Register ins Spiel. Durch dieses zweite Register kann man sich innerhalb eines Segmentes bewegen. Denn es gibt den Abstand vom Beginn des Segments (welcher, wie wir ja wissen in den Segment–Registern gespeichert ist), bis zu der gewünschten Speicherstelle an. Dieses Offset–Register ist der Zeiger auf den Segment–Inhalt.

Somit erhalten wir eine Methode, bei der wir mit 2 Registern jede Stelle in dem 1 MB großen Adreßraum ansprechen können.

Die Segmentregister

Wir haben besprochen, wie die Segmentierung funktioniert, nun müssen wir noch die Aufgaben der Segmentregister klären. Es existieren folgende Segmentregister (wie wir am Anfang schon gelernt haben):

CS	„code segment register“
DS	„data segment register“
ES	„extra segment register“
SS	„stack segment register“
FS	„help segment 1 register“ (erst ab dem 80386 vorhanden)
GS	„help segment 2 register“ (erst ab dem 80386 vorhanden)

Die Aufgaben der Register CS und DS sind wohl klar: In ihnen ist die Segment–Grenze einmal für den Code und einmal für die Daten gespeichert. Das „stack segment register“ beinhaltet die Segment–Grenze für den Stack, welcher in einem extra Kapitel behandelt wird. Das „extra segment register“ wird bei String–Operationen benutzt. Die „help segment register“ dienen zur weiteren Aufnahme, und finden meines Wissens nach kaum Beachtung.

Der Leser, der die Segmentierung verstanden hat, wird sich jetzt fragen, welche die dazu gehörigen Offset–Register sind. Diese sind nur zweimal klar definiert: Nämlich für das „code segment register“, welches als Offset–Register das IP (EIP) Register hat. Und für das „stack segment register“ ist das SP (ESP) Register zuständig. Für die übrigen Register kommt es auf den Befehl an, von dem es dann abhängt, welches andere Register den Offset–Anteil bildet.

Glossar / Zusammenfassung

Das Kapitel über die Segmente beim 80386 ist sicherlich eines der schwierigeren Sorte, deshalb folgt jetzt noch einmal so etwas wie eine Zusammenfassung, mit einer näheren Erklärung:

Segment

Ein Segment ist mindestens 16 Byte groß, kann aber bis zu 65.536 Byte groß sein. Das hängt damit zusammen, daß ein Offset–Register (welches 16 Byte groß ist) einen maximalen Wert von 2^16 aufnehmen kann. Wenn dies der Fall ist, wird die nächste Segment–Grenze überschritten, und kann somit das Segment auf den Wert von 65.536 Byte erhöhen.

Segment–Grenze

Die Zahl, die in einem Segmentregister gespeichert wird, nennt man Segment–Grenze, da es auf den Anfang eines Segmentes zeigt.

Adresse

Die Adresse einer Speicherstelle wird in einem Segmentregister und einem Offset–Register gespeichert, welcher folgender Notation folgt: SEGMENTREGISTER : OFFSETREGISTER. z.B. merkt sich der Prozessor über die Registerkombination CS:IP (ab dem 80386 CS:EIP) welchen Befehl er als nächstes ausführen muß. Die physikalische Adresse bekommt man indem man den Wert im Segmentregister * 16 nimmt (da man den Adreßraum in Blöcke zu je 16 Byte aufgeteilt hat) und den Wert im Offset–Register hinzu addiert. Daraus folgt: Adresse = 16 * Segment + Offset.

Offset(–Register)

Der Offset–Anteil einer Adresse wird im Offset–Register gespeichert. Ein Offset–Register ist ein Zeiger um sich innerhalb eines Segmentes zu "bewegen". Der Offset–Anteil einer Adresse wird zu der „Segment–Grenze“ hinzu addiert.

Physikalische Adresse

Eine physikalische Adresse ist eine „wahre“ Adresse im Speicher. Eine normale Adresse besteht aus einem Segmentregister, und einem „Offset–Register“. Daraus läßt sich eine physikalische Adresse errechnen. (siehe „Adresse“)

Die Segmente ab dem 80386

Das was wir gerade besprochen haben war nicht sinnlos, da diese Regeln heute für DOS Programme immer noch gelten. Doch derjenige der sich am letzten Teil fast die Zähne ausgebissen hat, wird jetzt froh sein, denn für Windows Programme gelten
diese Regeln nicht mehr.

Der Bus und dessen Auswirkungen

Es gibt immer noch einen Adreßbus und einen Datenbus, doch sind seit dem 80386 beide gleich groß. Das bedeutet, daß man mit einem Adreßbus von 32 Bit maximal 2^32 Byte ansprechen kann. Dies sind 4 GB. Da nun auch der Datenbus 32 Bit groß ist, braucht man

keine Segmente mehr. Es existiert nun ein "flat memory model". Dieses neue Speichermodell hat zur Folge, daß alles linear ist. Doch wozu werden dann noch die Segmentregister benötigt, die ja noch da sind, und sogar Zuwachs bekommen haben mit FS und GS. Das hat folgenden Grund: Die 4 GB Adreßraum, die jedes Programm besitzt sind virtuell. Dies bedeutet, daß das Betriebssystem mit Adressen arbeitet, die gar nicht existieren. Und das Betriebssystem ist dafür verantwortlich, daß die aktuellen Daten im physikalischen Speicher zur Verfügung stehen. Um dies zu bewerkstelligen braucht Windows sogenannte Deskriptoren. In diesen Deskriptoren stehen Informationen die Windows dazu benötigt, wie z.B. Zugriffsberichtigungen. Genau an diesem Punkt kommen die Segmentregister ins Spiel; sie zeigen nämlich auf die Deskriptoren. Doch diese Vorgänge sind für eine „Cracker“ uninteressant. Es ist nur wichtig zu wissen, daß mit Windows ein "flat memory model" eingeführt wurde. Die Segmentregister können somit beim „Cracken“ außer acht gelassen werden. (Wer sich trotzdem für diese Vorgänge interessiert, findet Informationen im „Intel Architecture Software Developer`s Manual“ im Abschnitt „Protected-Mode Memory Management“.)

Der Stack

Der Stack allgemein

Der Stack ist ein wichtiger Bestandteil eines Programmes. Er ist so wichtig (und so vollkommen unbekannt in den „Cracking–Tutorials“), daß ich ihm ein eigenes Tutorial gewidmet habe. Ich werde ihn hier ausführlich behandeln, da er sehr wichtig ist im Bezug auf 'API Funktionen'. Das Stack–Ende findet man zu jeder Zeit in der Registerkombination SS:ESP. (Das Segmentregister SS, und das Offset–Register ESP).

Der Stack ist eine Art Ablageplatz. Dort können Daten gespeichert werden. Dies muß man zum Beispiel manchmal vor dem Aufruf eines Unterprogrammes tun, um die Inhalte wichtiger Register zu "retten", die sonst im Laufe des Unterprogrammes verloren gehen würden. Durch den Befehl PUSH kopiert man einen Wert auf den Stack; durch den Befehl POP holt man einen Wert wieder herunter. Mit dieser Erklärung könnte man sich zufrieden geben. Doch das werden wir nicht.

Was also passiert wenn man z. B. mittels des Befehls „PUSH EAX“ (Der Befehl PUSH erwartet natürlich eine Angabe, was er auf den Stack kopieren soll. Diese Angabe wird immer nach dem Befehl gemacht.) Durch diesen Befehl verändert sich der Zeiger des Stack. Danach zeigt er nämlich auf den Inhalt von EAX.

Vor dem Befehl:

```
??? <= ESP       ; ESP zeigt auf das Ende des Stack. Was dort auch immer
                 ; stehen mag.
```

Nach dem Befehl:

```
??? EAX <= ESP   ; ESP zeigt auf den Inhalt von EAX, welcher auf den Stack
                 ; kopiert wurde.
```

Doch bevor wir weiter in die Geheimnisse des Stack eingreifen, muß geklärt werden, wie der Stack 'wächst'. Jetzt denken sich vielleicht einige: Der Stack wächst ganz normal: ESP wird um soviel erhöht, wie der zu kopierende Wert an Bytes einnimmt. Doch diese Aussage ist so falsch, wie sie nur sein kann.

Der Stack wächst wie folgt: Er wächst nach unten, was bedeutet, daß ESP immer verringert wird, nachdem ein Wert kopiert wurde. (Bzw. der Stack wird erhöht, wenn ein Wert vom Stack genommen wird.) Der Stack wird immer um 4 verringert, bzw. erhöht. Diese Zahl ist also nicht dynamisch, sondern konstant. Egal, ob man jetzt nur ein Byte kopiert, z.B. durch den Befehl "PUSH AH", oder mehr, der Stack wird immer um 4 erhöht. Damit wäre dies geklärt.

Doch nicht nur der Programmierer sondern auch der Prozessor nutzt den Stack. Bei einem Aufruf von einem Unterprogramm durch den Befehl "CALL <Adresse>" wird die Adresse auf den Stack „gepusht“, an welcher der Prozessor nach dem Beenden des Unterprogrammes seine Arbeit fortsetzt. Jetzt werden vielleicht, die welche bereits etwas mehr Assembler–Erfahrung haben sagen, daß der Prozessor EIP auf den Stack kopiert, und somit die Adresse bewahrt, an der er später seine Arbeit fortsetzt. Doch einmal kurz nachgedacht, erkennt man, daß das in einer „Endlos–Schleife“ enden muß. So darf der Prozessor beim CALL Befehl nicht EIP auf den Stack kopieren, sondern die darauf folgende Adresse. Nach einem CALL Befehl sieht der Stack so aus:

```
.                               ; Vorherige Daten.
.
.
<Adresse>     <= ESP            ; Die nach dem CALL folgende Adresse.
```

ESP zeigt somit auf die gesicherte Adresse. Im folgenden wird diese gesicherte Adresse mit EIP bezeichnet, da diese später in dieses Register kopiert wird. Doch denken sie daran, daß es nicht dem Inhalt von EIP entspricht zu dem Zeitpunkt, als der CALL Befehl erfolgt ist; denn dann würde es in einer „Endlos–Schleife“ enden.

Um ein Unterprogramm zu beenden wird der RET Befehl verwendet. Dieser Befehl kopiert den Wert, auf den das „Stack–Ende“ zeigt (dieser Wert ist die Adresse, die von dem CALL Befehl auf den Stack kopiert wurde) nach EIP; somit wird das Programm nach dem CALL fortgesetzt.

Der Stack bei Hochsprachen

Die vorhergehenden Erläuterungen bezogen sich auf den Stack allgemein ab dem 80368. Doch wie sich der Stack bei Hochsprachen verhält, interessiert einen „Cracker“ ganz besonders. Denn heutzutage schreibt niemand man mehr ein größeres Programm in ASM, sondern höchstens Teile davon. Deshalb ist es wichtig zu verstehen, wie der Stack arbeitet, wenn das Programm mit einer Hochsprache geschrieben wurde.

Der Stack ist logisch aufgebaut. So besitzt jedes größere Unterprogramm einen eigenen Bereich auf dem Stack. Doch wie definiert man einen solchen? In SS:ESP ist immer das Ende des Stacks zu finden. So ist es vorgegeben. Doch wir benötigen noch den Anfang. Um diesen Anfang zu definieren benutzt man das EBP Register („base pointer register“.)

Kommen wir noch einmal zu der Situation von eben zurück: Es ist ein CALL Befehl erfolgt, und der Stack sieht so aus:

```
.               ; Vorherige Daten.
.
.
EIP   <= ESP    ; Die Rücksprung Adresse, wo das Programm
                ; später fortgesetzt wird.
```

Ich habe gerade gesagt, das man durch EBP den Anfang des private Stack definiert. Doch wo ist EBP? EBP ist noch nicht einbezogen worden, und zeigt deshalb auf eine Stelle, die nichts mit dem Stack bisher zu tun hat. Doch das ändert sich in der ersten Zeile des Unterprogrammes: Dort findet man häufig folgende 3 Zeilen:

```
PUSH EBP        ; Wir sichern EBP.
MOV EBP, ESP    ; EBP wird mit dem Wert von ESP überschrieben.
SUB ESP, ??     ; ?? wird von ESP abgezogen, und somit definiert man
                ; die Größe.
```

Doch eins nach dem anderen. Nach den ersten beiden Befehlen, sieht unser Stack so aus:

```
EIP              ; Die Rücksprung Adresse, wo das Programm später
                 ; fortgesetzt wird.
EBP=> EBP <=ESP  ; Der gesicherte Wert.
```

(Natürlich befinden sich weiterhin eventuell Daten oberhalb von EIP auf dem Stack, doch dies ist im Moment egal, und wird deswegen auch nicht weiter aufgezeigt.) Wir sehen, das jetzt EBP und ESP beide auf die gleiche Stelle zeigen. Somit hätten wir schon den Anfang definiert. Doch das Ende, welches in ESP sich befinden soll, ist noch nicht definiert. Das geschieht durch den Befehl SUB ESP, ??. Die zwei Fragezeichen stehen für eine Zahl, die von ESP subtrahiert wird. Durch diese Subtraktion wird ESP um ?? nach unten verschoben, und zeigt danach auf das Ende des Stack. Somit hat das Programm ?? Byte freier Speicher; und der Stack sieht wie folgt aus:

```
EIP              ; Die Rücksprung Adresse, wo das Programm später
                 ; fortgesetzt wird.
EBP=>EBP         ; Der gesicherte Wert.
.
.                ; Freier Speicher.
.
. <= ESP         ; Das Ende des Stack.
```

Um diesen Stack–Rahmen wieder aufzulösen werden folgende Befehle benutzt:

```
MOV ESP, EBP     ; Das Stack–Ende wird wieder an die ursprüngliche
                 ; Adresse gebracht.
POP EBP          ; Der vorher gesicherte Wert wird wieder in EBP geladen.
```

Durch diese beiden Befehle wird der ursprüngliche Zustand von ESP und EBP wieder hergestellt.

Anmerkung

Die beiden Befehlssequenzen kann man auch zusammenfassen in einem Befehl: Um einen Stack–Rahmen einzurichten wird auch der Befehl „ENTER“ verwendet. Dieser Befehl erwartet zwei Parameter, wobei der erste die Größe des Stack–Rahmens angibt, also die Zahl, die von ESP abgezogen ist. Der zweite Parameter ist meistens 0, da C/C++ von ihm keinen Gebrauch machen (Er hat eine ehe unwichtige Funktion für einen „Cracker“, und wird deshalb hier nicht weiter behandelt). Folgender Befehl:

```
ENTER 100, 0
```

ist also identisch mit folgenden Befehlen:

```
PUSH EBP
MOV EBP, ESP
SUB ESP, 100     ; 100h ist der erste Parameter.
```

Um einen Stack–Rahmen wieder zu löschen wird der Befehl „LEAVE“ verwendet, welcher keine Parameter erwartet. Somit ist

```
LEAVE
```

identisch mit:

```
MOV ESP, EBP
POP EBP
```

Die Größe des Stack–Rahmens, der durch den Befehl „ENTER“, oder durch die gleichbedeutende Befehlssequenz erzeugt wurde, ist nicht statisch, sondern kann, und wird auch sehr häufig verändert; und zwar durch die üblichen Befehle „PUSH“ bzw. „POP“.

Parameterübergabe

Die Einrichtung bringt einen Vorteil: Über das Register EBP kann man jederzeit auf lokale Daten zugreifen, und zwar indem man zu EBP einfach einen bestimmten Wert addiert oder subtrahiert. Und dies geschieht so: Bei Hochsprachen werden die Parameter für eine Funktion, über den Stack übergeben; dies bedeutet, daß diese vorher mittels des Befehls PUSH auf den Stack kopiert werden. Dazu ein Beispiel:

```
PUSH EAX          ; Der erste Parameter wird kopiert.
PUSH EBX          ; Der zweite Parameter wird kopiert.
PUSH ECX          ; Der dritte Parameter wird kopiert.
PUSH EDX          ; Der vierte Parameter wird kopiert.
CALL ??????       ; Die Funktion wird aufgerufen.
```

Diese Funktion wird in diesem Beispiel mit 4 Parametern aufgerufen. Danach erfolgt die Einrichtung eines Stack–Rahmens, wie oben besprochen. Danach sieht der Stack wie folgt aus:

```
EAX               ; Unsere vier Parameter.
EBX
ECX
EDX
EIP               ; Die Rücksprung–Adresse.
EBP=>EBP          ; EBP zeigt auf den gesicherten Wert.
.
.
.
.
. <= ESP          ; ESP markiert das Stack–Ende.
```

Somit zeigt EBP+8 auf das erste übergebene Argument. EBP+12 auf das zweite, usw. Genauso zeigt EBP-4 auf den ersten freien Speicherplatz und EBP-8 auf den zweiten, usw. Warum aber greifen wir nicht über ESP auf die Argumente, oder auf freie andere Speicherplätze zu? Wer bisher alles verstanden hat, und wer alles aufmerksam gelesen hat, wird es schon wissen: Da ESP durch die üblichen Befehle PUSH und POP verändert werden kann, ist er denkbar ungeeignet, da man sich immer merken müßte wieviele male man schon „PUSH“ und „POP“ verwendet hat. Doch EBP bleibt bis zur Auflösung des Stack–Rahmens am Ende des Unterprogrammes gleich. Somit ist klar, daß man EBP und nicht ESP verwendet.

Achtung

Bei C/C++ werden die Parameter für Funktionen nicht so übergeben, wie sie der Programmierer angibt. Ein Aufruf einer Funktion könnte z.B. so aussehen:

```
Hallo("Dies ist ein Test", "Hallo", 12345, 3.14);
```

Die Funktion hat 4 Parameter. In Assembler würde es so aussehen, da die Parameter umgekehrt übergeben werden:

```
PUSH <3.14>
PUSH <12345>
PUSH <"Hallo">
PUSH <"Dies ist ein Test!">
CALL <Hallo>
```

So sieht natürlich kein richtiger ASM–Code aus, er sollte nur noch einmal klarmachen, wie Parameter übergeben werden. In Wirklichkeit würden die Parameter in Form von Registern, oder Zeigern übergeben werden. Diese Regel gilt vor allem für API Funktionen. Bei API Funktionen werden die Parameter immer in umgekehrter Reihenfolge auf den Stack „gepusht“ (also übergeben) als es der Programmierer angibt.

Um das Kapitel vom Stack abzuschließen fehlt nur noch eine Sache: Das Bereinigen des „Stack“. Wenn ein Unterprogramm die Parameter auf den Stack „pusht“ und das Unterprogramm greift auf diese Parameter über EBP zu, bedeutet das, daß diese Werte nicht vom Stack wieder herunter genommen wurden; und das der Stack weiterhin diese Werte besitzt. Dazu ein Beispiel:

```
        PUSH EAX            ; Parameter
        PUSH 006
        PUSH EBX
        CALL ??????         ; Das Unterprogramm
```

Das Unterprogramm sieht so aus:

```
?????:  PUSH EBP            ; Erzeugung des Stack–Rahmens
        MOV EBP, ESP
        SUB ESP, 100
        .
        .
        <CODE>
        .
        .
        MOV ESP, EBP        ; Löschen des Stack–Rahmens
        POP EBP
        RET                 ; Rücksprung aus dem Unterprogramm
```

Das bedeutet doch, daß nachdem das Unterprogramm beendet ist, SS:ESP immer noch auf den letzten übergebenen Parameter zeigt. Dies muß bereinigt werden, da diese Werte nicht mehr benötigt werden. Und dies geschieht indem ein Wert zu ESP addiert wird, der bewirkt daß ESP wieder auf die Stelle zeigt bevor die Parameter auf den Stack „gepusht“ wurden. Wie groß dieser Wert ist, hängt davon ab, wieviele Parameter übergeben wurden. Die Anzahl dieser Parameter * 4, da ESP immer um 4 verringert wird. Bei unserem Beispiel wäre das die zahl 0Ch (= 12). Also würde unser Beispiel so aussehen:

```
        PUSH EAX            ; Parameter
        PUSH 006
        PUSH EBX
        CALL ??????         ; Das Unterprogramm
        ADD ESP, 0C         ; Reinigung des Stacks
```

Diese beschriebene Vorgehensweise gilt nur für C/C++ Programme. Andere Programmiersprachen gehen unter Umständen anders vor. Eine weitere Möglichkeit besteht durch den RET Befehl. Dieser oben beschriebene Befehl, kann auch eine solche Bereinigung des Stack durchführen, indem man dem Befehl einen Operand hinzufügt. Dieser Operand gibt an, um wieviel Bytes der Stack gereinigt werden soll. Somit würde sich bei unserem Beispiel diesmal das Unterprogramm ändern:

```
?????:  PUSH EBP            ; Erzeugung des Stack–Rahmens
        MOV EBP, ESP
        SUB ESP, 100
        .
        .
        <CODE>
        .
        .
```

```
MOV ESP, EBP        ; Löschen des Stack–Rahmens
POP EBP
RET 0C              ; Reinigung des Stack und Rücksprung aus dem
                    ; Unterprogramm
```

Wenn der RET Befehl diese Reinigung durchführt, ist die Addition von 0Ch nach dem „CALL“ natürlich nicht mehr nötig, und entfällt.

Damit ist das Kapitel über den Stack abgeschlossen.

Etwas übers „Cracken“ und der Schluß

Wenn Sie sich alle Kapitel aufmerksam durchgelesen haben, so besitzen Sie eine solide Grundlage fürs „Cracken“. Die einzelnen Kapitel führen teilweise weiter in Assembler hinein, als daß Sie es jemals beim „Cracken“ brauchen könnten, doch dies läßt sich nicht vermeiden, wenn man die Zusammenhänge verstanden haben will. Das „Cracken“ konzentriert sich heutzutage auf Win9x/NT Programme. Daher kurz noch eine Erklärung dazu:

Das „Cracken“ eines Programmes unter Win9x und NT setzt zu 99% auf das Windows API. Ich kann mir sicherlich vorstellen, daß einige „Cracker“ oder die, die von sich solches behaupten, noch nie etwas davon gehört haben.
API steht für "application programming interface". Und um das zu erklären folgender Ausschnitt aus der „MSDN–Bibliothek“:

API

Eine Reihe von Routinen, die eine Anwendung verwendet, um systemnahe Dienste anzufordern und aufzurufen, die dann vom Betriebssystem eines Computers bereitgestellt werden. Bei Computern mit einer grafischen Benutzeroberfläche verwaltet ein API die Fenster, Symbole, Menüs und Dialogfelder einer Anwendung.

Wem noch immer nicht klar ist was das API ist nochmals: Diese Funktionen sind im Betriebssystem integriert, so daß sie jeder Programmierer nutzen kann. Diese Funktionen erleichtern dem Programmierer seine Aufgaben ungemein, denn er kann so immer wieder auf vorgefertigte Menüs und Dialogboxen zurückgreifen. Und dies benutzen alle „Cracker“ als Einstiegs–Punkt für ihre „Cracks“.

Diese API Funktionen sind sehr wichtig, und daher sollte man sich eine Referenz darüber besorgen.

Schluß

Ich hoffe das Lesen hat Spaß gemacht und ihr Wissen erweitert.

Ich bin zu erreichen unter
ger.tuts@redseven.de

Formatierung, Korrekturen & PDF–Umsetzung:
seaw0lf@gmx.net

Hubertus Loobert,
13.10.1999