1 Überblick Assembler-Programmierung

Assembler-Programmierung: Programmierung auf Maschinencode-Ebene.

Argumente für Assembler-Programmierung:

Geschwindigkeits-Optimierung: Das wichtigste Argument, ein Programm in Assembler zu schreiben, war früher der Geschwindigkeits-Vorteil gegenüber einer Lösung in höheren Programmiersprachen.

Dieses Argument gilt für "einfache" Prozessoren (z. B. Mikrocontroller) immer noch, für moderne PC- und Server-Prozessoren hingegen seit einigen Jahren nicht mehr: Das zeitliche Verhalten eines Codestückes ist auf solchen Prozessoren mit ihren vielstufigen, parallelen Pipelines, Befehlsumordnung, Sprungvorhersage bzw. spekulativer Ausführung und anderen Tricks so komplex, daß es für Menschen nicht mehr überblickbar ist, und ein hochoptimierender Compiler daher oft schon schnelleren Code erzeugt als ein Assembler-Programmierer.

Merklich schneller ist handgeschriebender Code nur mehr dann, wenn er entweder Spezialinstruktionen (SSE usw.) verwendet oder sich nicht an die vom Compiler verwendeten Aufrufskonventionen hält.

Speicherplatz-Optimierung: Sorgfältig geschriebene Assembler-Programme belegen nach wie vor in den meisten Fällen signifikant weniger Speicherplatz für Code und Daten als gleichartige Programme in höheren Programmiersprachen.

Vor allem bei Programmen, die direkt in Hardware (ROM) gegossen werden, ist das ein wesentliches Argument, entweder, weil der zur Verfügung stehende Speicher (z. B. bei Single-Chip-Systemen) sehr begrenzt ist, oder weil bei entsprechend hohen Stückzahlen jedes Bit mehr Speicherplatz kostenmäßig ins Gewicht fällt.

Direkter Zugriff auf die Hardware: Assemblerprogramme können I/O-Bausteine, CPU-intere Spezialregister, Grafikkarten etc. ansprechen (wenn sie dürfen).

Verwendung von Spezialbefehlen: Assembler-Programme können CPU-Befehle nutzen, die Programmen in höheren Programmiersprachen nicht zur Verfügung stehen:

- Privilegierte Befehle auf Betriebssystem-Ebene zur Speicherverwaltung, zum Prozeßwechsel, zum Interrupt-Handling, ...
- Befehle zur Verarbeitung spezieller Datentypen ("Multimedia-Befehle": MMX, SSE).
- Befehle zur Synchronisation zwischen Prozessoren in Mehrprozessorsystemen.

Mangel an Compilern: Für einige ganz kleine Mikrocontroller gab bzw. gibt es keinerlei brauchbaren Compiler für höhere Programmiersprachen.

Argumente gegen Assembler-Programmierung:

Entwicklungs- und Wartungsaufwand: Software-Entwicklung in Assembler dauert sehr viel länger als in höheren Programmiersprachen. Die entstehenden Programme sind fehleranfälliger und schwerer zu lesen und zu warten, das Debugging ist mühsamer.

Keine Libraries: Es gibt keine standardisierten Libraries: Alles das, was in höheren Programmiersprachen an vordefinierten Funktionen zur Verfügung steht, muß man in Assembler u. U. selbst programmieren.

Maschinenabhängigkeit: Assembler-Programme sind nicht portabel: Sie müssen für jede Prozessor-Architektur (x86, IA64, Alpha, PowerPC, Sparc, HP-PA, MIPS, ARM, s390, ...) komplett neu entwickelt werden, typischerweise von neuen, auf die jeweilige Architektur spezialisierten Entwicklern.

Hochoptimierte Programmteile müssen sogar individuell auf einzelne Prozessor-Modelle angepaßt werden: Ein 386-Executable beispielsweise funktioniert zwar unverändert auf einem Pentium 4, aber was für den 386 optimaler Code war, kann auf dem Pentium 4 höchst ineffizient sein...

Betriebssystem-Abhängigkeit: Selbst innerhalb derselben Prozessorarchitektur müssen normalerweise alle Aufrufe des Betriebssystems und alle Interaktionen mit C-Code überarbeitet werden, wenn das Programm auf ein anderes Betriebssystem portiert wird.

Assembler-Abhängigkeit: Bei x86-Assembler kommt noch ein weiteres Problem dazu: Es gibt mehrere verschiedene, völlig unterschiedliche Syntax-Konventionen für die Notation von x86-Befehlen und -Adressierungsarten:

- Alle kommerziellen Unix-Systeme und der Standard-Linux-Assembler `gas` verwenden die unter Unix auch bei anderen Prozessoren übliche **AT&T-Syntax**[1].
- Intel selbst hat seit jeher eine komplett andere Syntax verwendet (u. a. mit vertauschter Reihenfolge der Operanden), der heute weit verbreitete, kostenlose `nasm` hält sich auch weitgehend an diese **Intel-Syntax**.
- Der Microsoft-Assembler `masm` und Borland's Turbo-Assembler `tasm` verwenden im Prinzip ebenfalls die Intel-Syntax, sind aber in Details inkompatibel zu `nasm`.

Arten von Assembler-Programmcode:

Assembler-Files: Im Normalfall wird der Sourcecode von Assembler-Programmen in eigene Files geschrieben (unter Linux: Endung `.s`) und vom ***Assembler*** in Maschinencode (unter Linux: `.o`) übersetzt. Mehrere solche *"Object-Files"* werden dann (u. U. zusammen mit Libraries) vom ***Linker*** zu einem ausführbaren *"Executable"* (Binärprogramm) (unter Linux: Keine Endung) kombiniert.

Inline-Code: Einige C-Compiler erlauben es, innerhalb von C-Sourcefiles mitten im C-Code einige Zeilen Assembler-Code einzufügen. Dieser wird dann zusammen mit dem C-Code vom Compiler übersetzt, gelinkt und und ausgeführt.
Manche Compiler enthalten dafür einen einfachen, eingebauten Assembler. Der `gcc` erzeugt ohnehin immer Assembler-Sourcecode aus dem C-Code und ruft dann einen Assembler auf: In diesem Fall wird der Inline-Assemblercode aus dem C-File einfach in den erzeugten Assembler-Code übernommen.

Verwendung von Assembler:

- In den Anfängen der EDV wurde praktisch ausschließlich in Assembler programmiert.
- Die Verwendung von Assembler geht seit Jahrzehnten ständig zurück, der Anteil von Assembler an Software-Neuentwicklungen ist heute vernachlässigbar gering (im Millionstel-Bereich): Sogar klassische "Assembler-Hochburgen" wie Betriebssysteme und Anlagensteuerungen werden heute praktisch nur mehr in Hochsprachen programmiert, nur ganz kleine Mikrocontroller werden manchmal noch immer in Assembler programmiert.
- Assembler kommt heute meist in einzelnen Spezialfunktionen innerhalb von Programmen in einer höheren Programmiersprache wie C vor.
- Am Beispiel Linux:
  - Der Linux-Kernel enthält pro Prozessor-Plattform an ein paar hundert Stellen kurze Stücke Inline-Assembler-Code im C-Source (meist nur ein oder zwei Zeilen) und gut zwanzig komplette Assembler-Sourcefiles (z. B. für das Booten, für die Emulation

[1] `gas` versteht inoffiziell inzwischen im wesentlichen auch die Intel-Syntax, aber sie ist nicht dokumentiert und noch fehlerhaft und unvollständig implementiert.

der Gleitkomma-Befehle auf Prozessoren ohne Gleitkommahardware, für das Starten von Programmen, für Syscalls und den Datentransfer zwischen Anwenderprogrammen und Kernel, für Semaphore, und für eine geschwindigkeitsoptimierte TCP/IP-Prüfsummen-Berechnung).

* Weiteren Assembler-Code gibt es beispielsweise in Lilo und Grub (den beiden Linux-Bootloadern) und im X-Windows (für die Pixelmanipulationen bei alten Grafikkarten, die das nicht selbst können).

Ziel dieses Gegenstandes:

- Das Verständnis für die Arbeitsweise eines Computers soll vertieft werden.
- Es soll verdeutlicht werden, wie ein Computer in höheren Programmiersprachen geschriebene Daten und Programme intern verarbeitet.
- Die Grundlagen der Assembler-Programmierung sollen vermittelt werden.
- Das Zusammenspiel von Assemblercode mit C-Code soll demonstriert und geübt werden.

Unsere Plattform:

Intel 32-Bit-Prozessoren der x86/Pentium-Familie: Leider eine der grauslichsten Prozessorarchitekturen, die es gibt (ein "historischer Unfall"), aber die Verbreitung zählt...

32-Bit-Mode: Die x86-Architektur kennt drei verschiedene Betriebsarten:

16-Bit-Real-Mode (DOS-Modus, 8086-Modus): 16 Bit breite Daten- und Adressregister, 1 MB maximaler Speicher mit 64 KB Segmenten *ohne* Speicherschutz, *ohne* Paging und *ohne* virtuelle Adressen.

16-Bit-Protected-Mode (286-Modus): 16 Bit breite Daten- und Adressregister, 16 MB maximaler Speicher mit 64 KB Segmenten *mit* Speicherschutz und virtuellen Adressen, aber *ohne* Paging.

32-Bit-Mode (386-Modus): 32 Bit breite Daten- und Adressregister, 4 bzw. 64 GB maximaler Speicher mit 4 GB Segmenten *mit* Speicherschutz, Paging und virtueller Adressierung.

Alle modernen Unix- und Windows-Betriebssysteme sowie OS/2 verwenden den 32-Bit-Mode, er ist auch anderen modernen Prozessorarchitekturen am ähnlichsten. DOS verwendet den 16-Bit-Real-Mode, Windows 3.*x* verwendete ein Gemisch aller Modes.

Linux:

Linux bietet einige Vorteile für unsere Zwecke:

- Bei Programmfehlern sollte nur das jeweilige Programm abgebrochen werden, nicht aber wie unter DOS das ganze System abstürzen.
- Der 32-Bit-Mode ist leichter zu verstehen und komfortabler zu verwenden als ein 16-Bit-Mode.
- Linux bietet einen guten, wohldokumentierten C-Compiler.
- Ein DOS-Environment ist heute nicht mehr zeitgemäß.

Es hat aber auch einen wesentlichen Nachteil:

- Normale Benutzerprogramme haben unter Linux keinen direkten Zugriff auf Hardware-Ressourcen, privilegierte CPU-Features usw.. Das Entwickeln von Device-Treibern oder der direkte Zugriff auf die Grafikkarte bzw. den Bildschirm-Speicher ist daher in unserem Gegenstand nicht möglich.

`nasm`: Im Unterschied zu `gas` (dem Standard-Assembler unter Linux) verwenden wir Intel-Syntax!

Inhalt dieses Gegenstandes:

- x86-Prozessor-Architektur (Register, Adressierungsarten, ...) und -Befehle
- `nasm`-Assembler-Syntax und Assembler-Befehle
- C-Funktionsaufrufs- und Datendarstellungs-Konventionen
- Linux-Kernel-Aufrufskonventionen
- Makros

Nicht Inhalt dieses Gegenstandes:

- Low-Level-Programmierung: I/O, Device-Driver, Kernel Modules, Interrupts, ...
- x87-Gleitkomma-Befehle: Die x87-Gleitkomma-Architektur ist derartig skurill und unterscheidet sich so grundlegend von der Gleitkommaverarbeitung in allen anderen Prozessorarchitekturen, daß eine genauere Beschäftigung damit nicht sinnvoll ist.
- Segmentierte Speicherverwaltung: Die segmentierte Speicherverwaltung ist zentrales (und leidiges) Thema bei der Programmierung von x86-Prozessoren im 16-Bit-Modus (z. B. unter DOS). Im 32-Bit-Modus ist sie zwar noch vorhanden, aber unnötig, Linux verwendet sie auch nicht. Nachdem auch die segmentierte Speicherverwaltung ein Spezifikum der x86-Architektur ist, das in anderen Prozessorarchitekturen so nicht vorkommt, wollen wir getrost darauf verzichten.
- MMX- und SSE-Befehle: Noch nicht für den allgemeinen Gebrauch etabliert (obwohl die SSE-Befehle auf lange Sicht vermutlich die x87-Gleitkommabefehle ablösen werden).

Software, Dokumentation, Unterlagen: Siehe Links auf meiner Webseite...

2 Unsere Werkzeuge

Details in den jeweiligen `man`-Pages und Manuals!

Assembler: `nasm -O2 -f elf` *asmfile*

Dieser Aufruf erzeugt aus dem Assembler-Sourcefile *asmfile* einen Objectfile. Dieser bekommt normalerweise den gleichen Namen wie der *asmfile*, aber mit der Extension `.o` (mit `-o` *ofile* könnte man einen anderen Namen angeben). Mit `-l` *listfile* kann man zusätzlich einen Listing-File erzeugen lassen.

`-f elf` legt das Format des Binärfiles fest, `elf` ist das Format von Linux (`nasm` könnte z. B. auch `.com`-Files für DOS erzeugen). `-O2` bewirkt, daß der Assembler versucht, alle Sprung-Offsets im erzeugten Code mit möglichst wenigen Bytes zu codieren (`nasm` muß dafür mitunter den Inputfile mehrmals lesen).

Wie jeder Unix-Compiler gibt `nasm` *nichts* am Terminal aus, wenn das Programm fehlerfrei war und erfolgreich übersetzt wurde.

Der erzeugte Objectfile ist nicht direkt ausführbar, er muß zuerst gelinkt werden.

Assembler-Sourcefiles haben unter Unix traditionell die Endung `.s` oder `.S`. Diese bezieht sich aber auf Files in AT&T-Syntax (für `gas`), man sollte daher für Files in Intel-Syntax eine andere Endung verwenden, z. B. `.asm` (`.a` ist leider schon anderweitig vergeben!).

C-Compiler: `gcc -Wall -g -c` *cfile*

Dieser Aufruf erzeugt aus dem C-Sourcefile *cfile* (mit der Extension `.c`) einen Objectfile (wieder mit dem gleichen Namen wie der *cfile*, aber mit der Extension `.o`, oder aber mit dem mit `-o` *ofile* angegebenen Namen).

`-c` sorgt dafür, daß `gcc` nur compiliert, aber nicht linkt. `-Wall` schaltet einige zusätzliche Warnings für dubiose C-Konstrukte ein, und `-g` bewirkt, daß Debug-Information in den Output-File geschrieben wird (weitere `gcc`-Optionen findet man auf meinen Webseiten zum Gegenstand *Programmieren-C*).

Mit der Option `-S` kann man den `gcc` dazu bringen, aus einem C-File statt eines Objectfiles einen Assembler-Sourcecode-File (allerdings in `gas`-Syntax, nicht `nasm`-Syntax) zu erzeugen: Auf diesem Weg kann man sich anschauen, welcher Maschinencode für ein bestimmtes Stück C-Code erzeugt wird.

Linker: Hier sind zwei Fälle zu unterscheiden:

`ld -o` *exefile ofile* ...
: Dieser direkte Linker-Aufruf gilt für Assembler-Programme, die direkt (mit `_start`) gestartet werden und keine C- oder C-Library-Funktionen aufrufen.

`gcc -o` *exefile ofile* ...
: Dieser Aufruf ist zu verwenden, wenn

- Das Assembler-Programm mit der C-Methode (`main`) gestartet wird.
- Das Assembler-Programm C-Library-Funktionen verwendet.
- Assembler-Code und eigener C-Code (z. B. ein C-Hauptprogramm) gemischt werden sollen.

(er hängt automatisch den C-Initialisierungscode und die C-Library dazu)

In beiden Fällen können mehrere Objectfiles *ofile* angegeben werden (bei "gemischten" Programmen der vom C-Programm und der vom Assembler-Programm, sowie bei großen, auf mehrere Sourcefiles verteilten Assembler-Programmen einer pro Sourcefile). Derjenige *ofile*, der das Hauptprogramm `main` oder `_start` enthält, muß dabei zuerst angegeben werden (egal, ob er von einem C- oder einem Assembler-Programm ist).

exefile ist der Name des zu erzeugenden Executables (in Linux üblicherweise ohne Extension), dieses kann anschließend wie ein normaler Unix-Befehl gestartet werden (`./`*exefile*, falls `.` nicht im `PATH` ist!).

Debugger: `ald` *exefile*

Startet den Debugger `ald` mit dem angegebenen Programm (man kann leider *keine* Argumente für das Programm angeben). `ald` meldet sich mit einem Prompt, bei dem man unter anderem folgende Befehle eintippen kann:

Run: `r` *[arg ...]* startet das Programm (von vorne) mit den angegebenen Argumenten. Das Programm wird bis zu einem Breakpoint oder bis zum Ende ausgeführt. Mit `set args` *arg* ... kann man die Argumente festlegen, damit man sie nicht bei jedem `r` angeben muß.

Continue: `c` setzt das Programm an der aktuellen Stelle fort.

Step: `s` *[n]* führt den nächsten Befehl / die nächsten *n* Befehle des Programmes aus. Zeigt danach die Register und den nächsten auszuführenden Befehl an.

Next: `n` *[n]* arbeitet wie `s`, aber steppt nicht in Funktionsaufrufe hinein.

Register: `re` *[reg]* zeigt den Inhalt aller Register / des Registers *reg* an. Fragt bei einem einzelnen Register, ob man einen neuen Wert in das Register schreiben will.

Examine: `e` *from [to]* oder `e` *from* `-n` *n* zeigt den Speicherinhalt zwischen den Adressen *from* und *to* oder *n* Elemente ab Adresse *from* an. Optionen:

`-s` *b*: Jedes Element ist *b* Bytes lang (z. B. `4` für Adressen oder Integers).

`-o x` oder `-o o` oder `-o d`: Ausgabe in Hex / Oktal / Dezimal.

Bei *from* kann statt einer Adresse auch ein Register-Name angegeben werden (`eax` usw.): Der Inhalt dieses Registers ist dann die Adresse, ab der der Speicher angezeigt wird. `e esp` zeigt also den aktuellen Inhalt des Stacks an, `e ebp` die Argumente des aktuellen Funktionsaufrufes.

Disassemble: `d` *from [to]* oder `d` *from* `-n` *n* disassembliert alle Befehle zwischen Adresse *from* und Adresse *to* oder *n* Befehle ab *from*.

Breakpoint: `b` *adr* oder `b` *label* setzt auf die angegebene Adresse oder auf das angegebene Label einen Breakpoint.

List Breakpoints: `lb` zeigt alle Breakpoints an.

Delete Breakpoint: `del` *n* oder `del all` löscht den Breakpoint Nummer *n* / alle Breakpoints.

Enable / Disable Breakpoint: `enab` *n* oder `enab all` oder `disab` *n* oder `disab all` macht den Breakpoint Nummer *n* / alle Breakpoints aktiv / inaktiv.

Ignore Breakpoint: `i` *n i* bewirkt, daß der Breakpoint Nummer *n* erst nach *i* Durchläufen aktiv wird.

File Sections: `f sec` zeigt die Sections (Adresse, Größe) des Programms an.

File Symbols: `f sym` zeigt die Symbole (Name, Wert) des Programms an.

Quit: `q` oder `Ctrl/D` beendet den Debugger.

Help: `h` oder `h` *command* zeigt eine Liste aller Befehle bzw. die Hilfe zu einem Befehl an.

Repeat: <*Return*> führt den letzten Befehl nochmals aus (praktisch bei `s`).

Bugs:
Die aktuelle Version von `ald` schafft es nicht immer, das Programm nochmals zu starten, wenn es einmal an einem Signal verstorben ist. `ald` beenden und frisch aufrufen!

Der Standard-Linux-Debugger ist `gdb`, er ist viel mächtiger und eignet sich wesentlich besser als `ald` für das Debuggen von C-Programmen, aber die Arbeit mit Assembler-Programmen ist in `gdb` wesentlich mühsamer (außerdem verwendet `gdb` beim Disassemblieren die AT&T-Syntax).

Syscall Tracer: `strace -i` *exefile arg ...*

`strace` führt das angegebene Programm *exefile* normal aus und schreibt zusätzlich am Terminal (auf `stderr`) oder auf einem mit `-o` *listfile* angegebenen File mit, welche Syscalls das Programm an welcher Codestelle mit welchen Argumenten und welchem Returnwert aufgerufen hat (die Hexzahl am Anfang jeder Zeile ist die Programmadresse, von der aus der Syscall-Aufruf erfolgt ist).

3 `nasm`-Syntax

Die Syntax einer Zeile:

Label: *Befehl Operanden* ; *Kommentar*

wobei gilt:

- *Label*: ist optional und nur dann notwendig, wenn der Befehl oder der Speicherplatz ein Label bekommen soll.
- *Label*: kann auch allein in einer Zeile stehen, es bezieht sich dann auf den Befehl in der folgenden Zeile.

- Ebenso kann ; *Kommentar* allein in einer Zeile stehen oder ganz weggelassen werden.
- *Befehl* kann ein *"Mnemonic"* (der Name eines Maschinenbefehls), eine Assembler-Anweisung oder ein Makro sein[2].
- Ob *Operanden* notwendig sind, hängt vom Befehl ab.
- Zwischenräume können nach Belieben eingefügt werden. Es ist zulässig, vor *Label* Zwischenräume zu lassen oder einen *Befehl* ohne Zwischenraum direkt an den Zeilenanfang zu schreiben, beides sollte man aber aus Gründen der Leserlichkeit tunlichst vermeiden!
- Auch der : nach *Label* ist optional, sollte aber aus Gründen der Leserlichkeit immer gemacht werden[3].

Folglich gibt es zwei sinnvolle "Stilrichtungen":

- *Befehl* weit eingerückt (mindestens 10 Spalten), *Label*: in der gleichen Zeile davor.
- *Befehl* wenig eingerückt (2 oder 3 Spalten), *Label*: immer in einer eigenen Zeile drüber (dadurch sind längere Labels und mehr Kommentare möglich).

Kommentare:

Alles nach einem ; bis zum Ende der Zeile ist Kommentar und wird ignoriert, bitte reichlich davon Gebrauch zu machen!

Labels:

- `nasm` erlaubt zwar diverse Sonderzeichen in Labels, aber man sollte sich an die C-Konventionen halten: Buchstaben (ohne Umlaute!), Ziffern, `_`, nicht mit einer Ziffer beginnend.
- Die Länge ist reichlich bemessen, man mache davon im Sinne der Leserlichkeit auch Gebrauch!
- Labels sind Case-sensitiv, `blabla`, `BlaBla` und `BLABLA` sind drei verschiedene Labels!
- Alle Register-Namen und Befehls-Kürzel sind reserviert und dürfen nicht als Label verwendet werden.
- Der *Wert* eines Labels ist jene *Adresse*, die beim Assemblieren an der Stelle, wo das Label steht, gerade aktuell ist:
  - Bei Labels vor Maschinenbefehlen in der Section `.text` ist der Wert des Labels eine Code-Adresse (nämlich die, an der der Befehl in dieser Zeile zu liegen kommt).
  - Bei Labels vor Assembler-Anweisungen in den Sections `.data` oder `.bss` ist es eine Adresse von Daten (nämlich jenen, die in dieser Zeile angelegt werden), also — in C-Terminologie — ein Pointer auf diese Daten.
- Folglich darf ein Label nur *einmal* definiert werden (links stehen), aber beliebig oft verwendet werden (als Operand eines Befehls).

Lokale Labels:

Beginnt ein Label mit . (beispielsweise `.endif`) so ist es ein *lokales Label*: Es ist nur zwischen den beiden umgebenden nichtlokalen Labels (Labels ohne . am Anfang) bekannt[4], kann aber in diesem Bereich wie ein normales Label verwendet werden.

---

[2] Hat der Maschinenbefehl einen Prefix-Befehl (z. B. `rep` bei den Stringbefehlen), dürfen auch zwei Mnemonics in einer Zeile stehen, man darf das Prefix aber auch allein in die vorhergehende Zeile schreiben.

[3] Aus diesen beiden Punkten ergibt sich eine ganz gemeine Fehlerquelle: Vertippt man sich bei einem *Befehl* ohne Operanden, sodaß `nasm` ihn nicht als Befehl erkennt, so liest `nasm` das — ohne Fehlermeldung! — als eine Zeile, die nur ein *Label* enthält (mit weggelassenem : und Zwischenraum davor), was zwar nach diesen Regeln zulässig, aber nicht im Sinne des Programmierers ist! Die Option `-w+orphan-labels` fängt solche Fehler ab.

[4] Will man es außerhalb dieses Bereiches ansprechen, muß man das letzte nichtlokale Label mit angeben.

Es ist kein Fehler, wenn zwischen zwei anderen nichtlokalen Labels ein lokales Label mit demselben Namen definiert wird: Es handelt sich um ein zweites Label, die beiden haben verschiedenen Gültigkeitsbereich und nichts miteinander zu tun.

In der Praxis verwendet man dieses Konzept, wenn ein Assembler-File mehrere Funktionen enthält: Für die Anfangs-Labels der Funktionen verwendet man nichtlokale Labels, für alle Labels innerhalb der Funktionen lokale Labels. Deren Gültigkeit ist damit auf die jeweilige Funktion beschränkt, Labels in verschiedenen Funktionen können nicht mehr kollidieren.

Zur Leserlichkeit:
In Assembler hängt die Leserlichkeit des Codes nicht nur von sprechenden Namen für Variablen ab, sondern vor allem auch von sprechenden Labels für Sprungziele.

Die beste mir bekannte Vorgangsweise lautet wie folgt:

- Für viele Labels ergibt sich aus der Bedeutung des Codes schon ein guter Name (`end`, `error`, `retry`, ...).
- Wenn nicht, sollte das Label nach dem Konstrukt benannt werden, für das es eingeführt wurde, z. B. `else` und `endif` oder `whilebeg` und `whileend`, vielleicht noch sprechender `iloop`, `jloop` usw..
- Bei mehreren Funktionen innerhalb eines Assembler-Files sollte man *lokale Labels* verwenden.
- Um die Labels eindeutig zu machen, und vor allem, um sie im Sourcecode schnell zu finden, bekommen sie bei längeren Funktionen (ab 1–2 Seiten Code) alle eine fortlaufende Nummer angehängt, und zwar von oben nach unten in Sourcecode-Reihenfolge und in Zehnerschritten (damit man nachträglich Labels einfügen kann)[5].

Operanden:

Je nach Befehl sind als Operanden zulässig:

Register: Werden per Namen angesprochen: `al`, `bp`, `edi`, ...
Speicher-Werte: Werden immer mit `[`*adr*`]` angesprochen: `[count]`, `[array+4*eax]`, ... (siehe Kapitel "Adressierungsarten")
Konstanten: `'a'`, `1`, `0xffffffff`, `myvar`, ... (auch Labels sind Konstanten!)
Arithmetische Ausdrücke: `base+4`, `100*4`, ...

Mehrere Operanden eines Befehls werden durch `,` (und eventuell Zwischenräume) getrennt.

Achtung, nicht verwechseln:

- `mov eax, myvar` speichert den Wert des Labels `myvar` (d. h. die *Adresse* von `myvar` bzw. einen Pointer auf `myvar`) in `eax`.
- `mov eax, [myvar]` speichert den *Inhalt* von `myvar` in `eax`. `[`*adr*`]` in `nasm` entspricht also in etwa `*(`*adr*`)` in C: Beides liefert den Inhalt der Speicherstelle, auf die *adr* zeigt.

Achtung:
Labels in `nasm` sind *typlos*! `nasm` weiß weder, ob ein Label auf ein Byte, Wort oder Doppelwort zeigt, noch, ob an dieser Adresse eine vorzeichenlose oder eine vorzeichenbehaftete Zahl liegt (oder gar eine Gleitkommazahl oder ein Codestück).

---

[5] Sind die Labels nicht aufsteigend numeriert, hat man beim Lesen eines längeren Programmes bei Sprungbefehlen ständig das Problem, daß man nicht weiß, ob man das angesprungene Label weiter oben oder weiter unten im Code suchen muß.

Daher gibt `nasm` auch keine Fehlermeldung aus, wenn man ein Label "falsch" (anders als gedacht) verwendet: Auch wenn man mit `myint: dd 1` ein Label auf ein Doppelwort angelegt hat, wird `nasm` bei `mov al, [myint]` kommentarlos ein einzelnes Byte (nämlich das niederwertigste Byte der `int`-Zahl) nach `al` kopieren ...

Aus demselben Grund muß man manchmal zusätzlich zum Operanden noch eine Längenangabe machen (`byte`, `word` oder `dword` bzw. `short` oder `near` für Sprünge):

- Bei Operationen mit einem Speicher- und einem Register-Operanden weiß `nasm` auf Grund der Größe des Register-Operanden, ob der Speicher-Operand ein Byte, Wort oder Doppelwort ist: Bei `mov eax, [var]` geht es um 4 Bytes, bei `mov dl, [var]` um 1 Byte.

  Hat ein Befehl hingegen nur einen Speicher-Operanden, so kann `nasm` dem Operanden nicht ansehen, wie viele Bytes an der angegebenen Speicherstelle zu verarbeiten sind, man muß es extra angeben:

```
inc     [var]           ; Fehlermeldung, Groesse von [var] unklar!
mov     [myint], 1      ; selbes Problem
inc     byte [var]      ; das Byte an der Adresse var wird um 1 erhoeht
inc     dword [var]     ; dasselbe fuer einen 4-Byte-int ebendort
```

- Der $x86$-Befehlssatz kennt einige 32-Bit-Befehle, die Varianten mit 8- und mit 32-Bit-Konstanten haben. Gibt man nichts an, hat `nasm` bisher in diesem Fall immer die lange Variante genommen, auch wenn die Konstante in einem Byte Platz gehabt hätte[6]:

```
add     esp, 4          ; erzeugt einen 6 Bytes langen Befehl
add     esp, byte 4     ; erzeugt einen 3 Bytes langen Befehl
```

- Weiters kennt der $x86$-Befehlssatz Adressierungsarten mit 1 und mit 4 Bytes langen Adress-Konstanten. Hier sollte `nasm` automatisch die kürzestmögliche Variante verwenden, aber um ganz sicher zu gehen, ist auch eine explizite Angabe möglich:

```
inc     [ebp-8]         ; sollte -8 als 1 Byte langes Offset codieren
inc     [byte ebp-8]    ; verwendet sicher ein 1 Byte langes Offset
inc     [dword ebp-8]   ; verwendet sicher ein 4 Bytes langes Offset
```

- Auch Sprungbefehle gibt es mit 1 Byte Offset (wenn das Sprungziel maximal 127 Bytes entfernt ist) und mit 4 Bytes Offset (für weiter entfernte Sprungziele).

  Bisher hat `nasm` per Default die 1-Byte-Variante erzeugt (und eine Fehlermeldung geliefert, wenn das Sprungziel zu weit weg war); für die längere Variante mußte man explizit `near` angeben.

  Die neuesten `nasm`-Versionen sollten automatisch die richtige Variante wählen, wenn man beim Aufruf die Option `-O2` angibt.

Achtung:
Im Unterschied zur AT&T-Syntax, bei der das Ergebnis im rechten Operanden gespeichert wird, steht beim `nasm` (Intel-Syntax) bei Maschinen-Befehlen mit mehreren Operanden *zuerst* (also *links*) der *Ziel-Operand* und *dann* (also *rechts*) der *Quell-Operand* (wie bei einer Zuweisung in C oder PASCAL: Da steht auch links die Zielvariable und rechts der zuzuweisende Wert)!

---

[6] In den neuesten `nasm`-Versionen wurde dieses Verhalten mehrmals geändert, mit der Option `-O2` sollte `nasm` automatisch die kürzere Variante nehmen, wenn der Operand klein genug ist.

Konstanten und Ausdrücke:

- Zahlen werden normalerweise als Dezimalzahlen betrachtet. Hex-Zahlen werden mit dem Prefix `0x` (wie in C) oder mit dem Suffix `h` geschrieben[7], Oktalzahlen mit dem Suffix `q`, Binärzahlen mit dem Suffix `b`.

  Achtung:
  Hexzahlen müssen in jedem Fall mit einer *Ziffer* beginnen, nicht mit einem Buchstaben. Ein Operand wie `ah` wäre sonst zweideutig: Handelt es sich um das Register `ah` oder die Hexzahl `a`?

- Buchstaben kann man in `'` oder `"` setzen, sie repräsentieren ihren ASCII-Wert. Die Assembler-Anweisung `db` verträgt auch Strings, ebenfalls wahlweise in `'` oder `"`.

  Achtung:
  `nasm` kennt *keine* \-Sequenzen wie C: `'\n'` sind *zwei* Buchstaben (`0x5c6e`)!

  Achtung:
  Strings sind in Assembler ***nicht*** automatisch `\0`-terminiert: Wenn man hinten ein `0`-Byte haben will, muß man es extra hinschreiben!

- Arithmetische Ausdrücke gleichen denen in C: `| & ^` berechnen bitweises Or, And und Xor, `<<` und `>>` stehen für bitweise Shifts, `+ - *` sind die üblichen arithmetischen Operatoren. `/` und `%` erlauben Division und Restbildung auf vorzeichenlosen Zahlen, `//` und `%%` machen das gleiche, aber betrachten ihre Argumente als Zweierkomplement-Zahlen. `+` und `-` funktionieren auch als einstellige Operatoren, ebenso `~` (bitweise Negation).

- Aktuelle Adresse und Anfangsadresse: `$` in einem Ausdruck wird durch die aktuelle Adresse (d. h. jenen Wert, den ein Label in derselben Zeile bekommen würde) ersetzt, `$$` steht für die Anfangsadresse der gerade verwendeten Section. `$-$$` liefert daher die Anzahl der Bytes zwischen Section-Anfang und aktueller Zeile.

Die "Sections" eines Assembler-Programms:

- Ein Assembler-Programm besteht unter Linux aus drei *Sections*:

  `.text` für den Programmcode (ausführbare Maschinenbefehle): Die `.text`-Section ist zur Laufzeit les- und ausführbar, aber nicht änderbar: Schreibzugriffe werden (bei fast allen Betriebssystemen) abgefangen[8].

  `.data` für initialisierte Daten (Variablen und Konstanten[9]), beispielsweise Strings: Diese Section ist zur Laufzeit les- und schreibbar, aber normalerweise nicht ausführbar.

  `.bss` für uninitialisierte Daten: Diese Section verhält sich zur Laufzeit wie `.data`, aber man kann dort weder Code noch initialisierte Daten hineinlegen, sondern nur Platz für Daten reservieren.

---

[7] Auch `$` wird als Prefix für Hexzahlen akzeptiert.

[8] Damit ist die früher so beliebte Unsitte, selbstmodifizierenden Code zu schreiben, hinfällig. Und selbst wenn es ginge, tun wir es nicht!

[9] Konstanten — vor allem Stringkonstanten — gehören genau genommen nach `.text`: Erstens sind sie dann wirklich gegen versehentliches Verändern geschützt, und zweitens liegen sie dann nur einmal im Speicher, wenn das Programm mehrmals gleichzeitig ausgeführt wird (`.text` ist geshart, `.data` wird für jeden Aufruf neu angelegt!).

Das funktioniert auch, nur muß man aufpassen, daß man alle Konstanten am Anfang oder zwischen Funktionen anlegt und nicht mitten zwischen Befehlen, damit sie nicht versehentlich "ausgeführt" werden, was schlimm endet!

Der GNU C Compiler `gcc` geht einen Mittelweg: Er legt die Stringkonstanten aus C-Programmen in eine eigene Section `.rodata` ("Read-only data").

Die `.bss`-Section wird vom Betriebssystem beim Programmstart automatisch mit lauter `0`-Bytes initialisiert, man kann sich also darauf verlassen, daß in `.bss` angelegte Variablen zu Programmbeginn den Wert `0` haben.

Der Vorteil von `.bss` ist, daß es — im Unterschied zu `.text` und `.data` — *keinen* Platz im Executable File belegt (große Arrays sollte man daher tunlichst in `.bss` und nicht in `.data` deklarieren)[10].

- Mit dem Befehl `section` *section* kann man im Assembler-Programm beliebig zwischen den Sections wechseln, alle Befehle nach einem `section` landen der Reihe nach in der angegebenen *section* (bis zum nächsten `section`).

  Das sollte man auch nutzen: Bei Programmen mit mehreren Funktionen sollte man unmittelbar vor dem Code einer jeden Funktion kurz nach `.data` oder `.bss` wechseln und die von der Funktion benötigten Variablen und Konstanten anlegen. Das ist leichter zu überblicken als ein einziger `.data`- und `.bss`-Block für die Daten aller Funktionen ganz am Anfang des Files.

- Für jede Section verwaltet der Assembler einen eigenen Zähler für die "aktuelle Adresse" in dieser Section (die Adresse, an der der nächste Befehl landet).

- Zusammen mit dem Stack und den Shared Libraries bilden die Sections (meist aufgerundet auf ein Vielfaches der Page-Größe) die "gültigen" Adressbereiche eines Programms: Speicherzugriffe außerhalb dieser Bereiche bewirken (je nach Betriebssystem) einen gewaltsamen Programmabbruch (Schutzverletzung, `SIGSEGV`)[11].

Assembler-Anweisungen:

Assembler-Anweisungen sehen aus wie Befehle (ein Mnemonic, eventuell mit einem Label davor und Operanden dahinter). Sie stellen aber Anweisungen an den Assembler selbst dar, es wird *kein* Maschinencode für sie erzeugt (deshalb werden sie mitunter auch *Pseudo-Operationen* genannt).

Die wichtigsten Assembler-Anweisungen sind:

`db` *expr* oder `dw` *expr* oder `dd` *expr* oder `dq` *expr* (*define byte, ...*) definieren ein mit dem angegebenen Wert *expr initialisiertes* Byte / Wort / Doppelwort / Quadwort, beispielsweise `db 'a'` oder `dd 0xFFFFFFFF`. Man verwendet sie normalerweise in der Section `.data`.

Werden mehr Daten angegeben, als Platz ist, werden entsprechend viele Bytes / Worte / Doppelworte angelegt: `str: db 'hallo', 0x0a, 0` (7 Bytes) oder `factor: dd -1, -2, -3, -4` (4 Doppelworte).

`resb` *n* oder `resw` *n* oder `resd` *n* oder `resq` *n* (*reserve byte, ...*) definieren *n uninitialisierte* Bytes / Worte / Doppelworte / Quadworte: `buffer: resb 1024` (1024 Bytes

---

[10] Die Größe der `.bss`-Section kann zur Laufzeit verändert werden: Mit dem Systemaufruf `sys_brk` kann man deren oberes Ende verschieben.

Auf diese Art und Weise fordern `malloc` usw. dynamisch Speicherplatz vom Betriebssystem an.

[11] Der Stack liegt unter Linux am oberen Ende des Adressraumes, der einem Programm zur Verfügung steht, alles andere am unteren Ende.

Der "gültige" Stackbereich wird vom Betriebssystem bei Bedarf automatisch nach unten vergrößert, man braucht also im Programm selbst keine maximale Stackgröße vorgeben.

Stack und `.bss` wachsen daher von beiden Enden gegeneinander in den unbenutzten Bereich des Adressraumes eines Programmes, stoßen sie zusammen (oder wird ein im System festgelegtes Speicherplatzlimit überschritten), beispielsweise bei einer endlosen Rekursion, wird das Programm abgebrochen.

Platz) oder `array: resd 256` (auch 1024 Bytes Platz). Man verwendet sie normalerweise in der Section `.bss`.

Label: `equ` *expr* (*equals*) definiert eine Konstante: Das *Label* ist in diesem Fall notwendig und wird besonders behandelt: Es bekommt als Wert nicht die aktuelle Adresse, sondern das aktuelle Ergebnis von *expr*.

Im Unterschied zu anderen Befehlen muß *Label* und `equ` in der selben Zeile stehen, man darf das *Label* nicht allein in die vorhergehende Zeile schreiben!

Eine typische Verwendung ist folgende:

```
msg:        db      'hello, world', 0x0a, 0
msglen:     equ     $-msg      ; Laenge von msg: Akt. Adr - Anfang von msg
```

`times` *expr Befehl* wiederholt *expr* mal den angegebenen Befehl *Befehl* (als ob man ihn textuell entsprechend oft hingeschrieben hätte): `array: times 256 db 0xFF` legt ein Array aus 256 Bytes mit dem Wert `0xFF` an.

Ein wichtiger Trick ist folgender:

```
buffer:     db      'hello, world'
            times buffer+256-$ db ' '
; ergaenzt 'hello, world' im Speicherbereich ab Adresse buffer
; mit ' ' auf eine Gesamtlaenge von 256 Bytes
```

Achtung: Kein `,` zwischen *expr* und *Befehl*!

`align` *n* oder `alignb` *n* bewirken durch entsprechendes Einfügen von Füllbytes, daß der Code oder die Daten der nächsten Programmzeile im Speicher auf der nächsten durch *n* teilbaren Adresse zu liegen kommt.

`align` ist für `.text` und `.data` und füllt den Platz mit `nop`-Befehlen (`nop` = "no operation": Ein Maschinenbefehl, der nichts tut), `alignb` ist für `.bss` und reserviert entsprechend viele uninitialisierte Bytes.

Elementare Daten sollten immer auf einer durch ihre Größe teilbaren Adresse liegen (4-Byte-Integer und -Pointer auf einer durch 4 teilbaren Adresse, 8-Byte-Gleitkommazahlen auf einer durch 8 teilbaren)[12], weil sonst zwei statt einem Speicherzugriff zum Lesen und zum Schreiben der Daten notwendig sind[13].

Code läuft dann am effizientesten, wenn wichtige Codestücke (z. B. Funktionsanfänge) auf einer durch 16 teilbaren Adresse liegen; das hängt mit der Cache-Organisation zusammen.

Große Datenblöcke für sektorenweises Disk-I/O sollten auf eigenen Pages liegen, ihre Anfangsadresse sollte daher durch die Pagegröße (bei der $x86$-Architektur 4096 Bytes) teilbar sein.

`section` *section* legt fest, in welche Section der folgende Code assembliert werden soll (`.text`, `.data` oder `.bss`; siehe eigenes Kapitel).

`extern` *label*`,` ... teilt dem Assembler mit, daß das in diesem File verwendete (aber nicht definierte!) Label *label* in einem anderen Assembler- oder C-File definiert wird.

---

[12] Sinnvollerweise legt man daher in `.data` und `.bss` zuerst die 8- und 4-Byte-Daten und erst dahinter die 2- und 1-Byte-Daten sowie die Strings an; damit hat man dieses Problem automatisch umgangen.

[13] Die $x86$-Architektur hat eine Option, mit der man "krumme" Speicherzugriffe abfangen und das Programm abbrechen kann, um derartige "Performance-Vernichter" zu erkennen. Auf den meisten anderen Architekturen sind solche Zugriffe grundsätzlich verboten und führen zu einem Software-Interrupt.

`global` *label*`,` ... ist das "Gegenteil" von `extern` und teilt dem Assembler mit, daß das in diesem File definierte Label *label* auch für andere Files ansprechbar sein soll (siehe Kapitel über die C-Anbindung).

4 *x*86-Architektur

4.1 Der Speicher

Zur Terminologie:

Nachdem die *x*86-Architektur historisch gesehen eine (nachträglich auf 32 Bit aufgeblasene) 16-Bit-Architektur ist, bezeichnet man mit *Wort ("Word")* nach wie vor nicht wie erwartet ein 32 Bit großes, sondern ein 16 Bit (2 Bytes) großes Datenobjekt. Ein 32 Bit (4 Bytes) großes Objekt wird demgemäß als *Doppelwort ("Doubleword")* bezeichnet, ein 64 Bit (8 Bytes) großes (z. B. eine `double` Gleitkommazahl oder ein `long long int`) als *"Quadword"*.

Adressierung:

Unter Linux läßt sich der Speicher eines *x*86-Rechners vereinfacht als einzeln direkt adressierbare Bytes betrachten[14]: Adressen sind *vorzeichenlose* 32-Bit-Zahlen, der Adressraum umfaßt daher 2^{32} einzelne Bytes oder 4 GB, das erste Byte hat die Adresse `0`, das letzte `FFFFFFFF`.

Trotzdem sind weder die Adressen im Assemblerprogramm noch die Adressen zur Laufzeit reale Speicheradressen:

- Im Assembler selbst werden die Adressen (und damit die Werte der Labels) in jeder Section für sich beginnend bei `0` vergeben (*"relative Adressen"*).
- Beim Linken und beim Starten des Programmes ändern sich diese Adressen aus zwei Gründen:
  - Die Inhalte der einzelnen Sections werden beim Linken aus allen beteiligten Programmen und Libraries zusammengesammelt und hintereinandergehängt, eine Section enthält also Code oder Daten aus mehreren Programm-Modulen. Damit beginnen Code und Daten eines Moduls nicht mehr bei `0`, sondern weiter hinten in der Section.
  - Beim Laden bekommen die Sections verschiedene Anfangsadressen zugewiesen, es können ja nicht sowohl Code als auch Daten an derselben Adresse `0` beginnen[15].

  Diese Vorgänge nennt man *Relozierung (Relocation)*.
- Die sich daraus ergebenden Adressen sind jene Adressen, mit denen das Programm wirklich arbeitet (und die man auch im Debugger sieht).
- Das sind aber nur *virtuelle Adressen*, und nachdem alle Programme gleichzeitig *dieselben* virtuellen Adressen verwenden, können diese nicht den tatsächlichen Hauptspeicheradressen entsprechen. An welchen *physikalischen (realen) Adressen* Programme und Daten wirklich liegen, weiß nur das Betriebssystem: Die realen Adressen ergeben sich aus dem

[14] Wir ignorieren die Speichersegmentierung der *x*86-Architektur, denn Linux verwendet das sogenannte "Flat Memory Model": Alle Segmentregister werden vom System auf ein und denselben Segment-Deskriptor initialisiert und vom Anwenderprogramm nie verändert. Dieser Deskriptor definiert ein Segment, das bei Adresse `0` beginnt und die vollen 4 GB groß ist. Alle Segmente liegen daher deckungsgleich übereinander; sie umfassen den gesamten Adressraum des Systems und tragen zur Adressberechnung nichts bei.

[15] Bei mir beginnt `.text` beispielsweise immer auf `0x08048080`, dann folgen — mit Löchern dazwischen — `.data` und `.bss`.

Segment- und Paging-Mechanismus und sind von Programmlauf zu Programmlauf verschieden, sie können sich sogar während des Programmlaufes ändern!

Byte-Anordnung:

Die $x86$-Architektur ist eine ***"Little-Endian"-Architektur***[16]: Aus mehreren Bytes bestehende Datenobjekte werden mit dem niederwertigsten Byte zuerst im Speicher abgelegt. Ist a die Adresse eines 4-Byte-Integers oder eines 4-Byte-Pointers, so liegt dessen niederwertigstes ("rechtes") Byte folglich an der Adresse a und sein höchstwertigstes ("linkes") Byte an der Adresse $a + 3$.

Andersherum ausgedrückt: Die Adresse eines Wortes oder Doppelwortes ist immer die numerisch kleinste bzw. erste der zwei oder vier Adressen, die dieses Wort oder Doppelwort im Speicher belegt, aber an dieser Stelle steht nicht das vorderste, sondern das hinterste Byte der Zahl!

Wenn man sich daher im Debugger einen Speicherbereich als Folge von Bytes anzeigen läßt, der das Doppelwort `0x12345678` enthält, wird man `0x78563412` sehen[17]!

Gleiches gilt für Konstanten (*imm*) in Befehlen: Wenn ein Befehl `0x00000001` als Doppelwort-Konstante enthält, wird das im Assembler-Listing als `0x01000000` angezeigt!

Stack:

Der Stack wächst auf der $x86$-Architektur *von oben nach unten*: Legt man Daten auf den Stack, wird der Stackpointer (Register `esp`) entsprechend heruntergezählt. Der *Stackpointer* `esp` zeigt dabei immer auf das unterste belegte Byte, nicht auf das oberste freie Byte.

Obwohl das zuletzt gepushte Element von der Adresse her das unterste Stack-Element wäre, spricht man aber trotzdem vom "obersten" Element bzw. vom "Top of Stack".

4.2 Die Register

Die $x86$-Architektur hatte ursprünglich acht allgemeine, 16 Bit breite Register (siehe Tabelle 1). Mit dem 80386 wurden dann alle Register auf 32 Bit erweitert, alle Namen bekamen ein `e` ("extended") vorne angehängt. Die "oberen Hälften" der Register sind nicht einzeln (als 16-Bit-Register) ansprechbar und haben keine eigenen Namen, die unteren Hälften können nach wie vor mit ihrem alten Namen als 16-Bit-Register verwendet werden.

`si`, `di`, `bp` und `sp` waren ursprünglich nur für Adressen gedacht (die nie 8 Bit, sondern immer 16/32 Bit lang sind), `ax`, `bx`, `cx` und `dx` hingegen teilte man für die Verarbeitung von einzelnen Bytes in zwei einzeln ansprechbare Hälften: Das höherwertige Byte ("high") und das niederwertige Byte ("low").

`ah`, `al` usw. sind daher *keine* eigenständigen Register: Verändert man `ah` oder `al`, ändert sich dadurch automatisch auch `ax` und `eax`. Schreibt man umgekehrt in `ax` oder `eax`, ändert sich dabei auch `ah` und `al`.

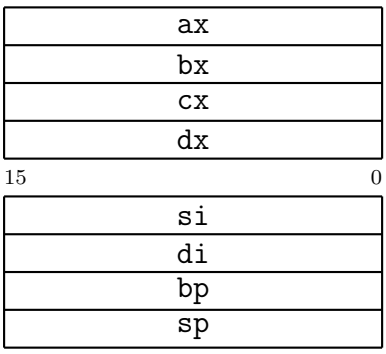
---

[16] Fast alle anderen Prozessor-Architekturen sind "Big Endian": Das höchstwertigste Byte einer Zahl wird auch als erstes gespeichert. Auch die TCP/IP-Standards schreiben vor, daß Integers am Netz mit dem vordersten Byte voran übertragen werden.

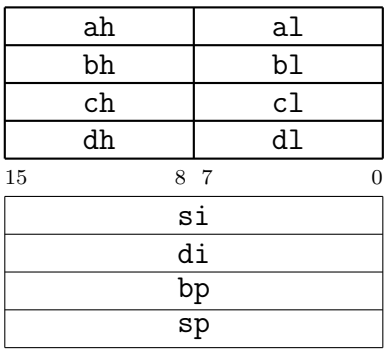
[17] Darum gibt es in Debuggern verschiedene Befehle zum Anzeigen eines Speicherbereiches als Bytes, Worte oder Doppelworte!

[18] Früher nannte man das "primäre" Register eines Prozessors, in dem die meisten Berechnungen durchgeführt wurden, "Akkumulator". Viele Befehle der $x86$-Architektur können zwar auf beliebige Register angewendet werden, haben aber nach wie vor eine "Kurzform" für den Akkumulator.

Die 16-Bit-Register:



Nutzung als 8-Bit-Register:



Erweiterung auf 32 Bit:

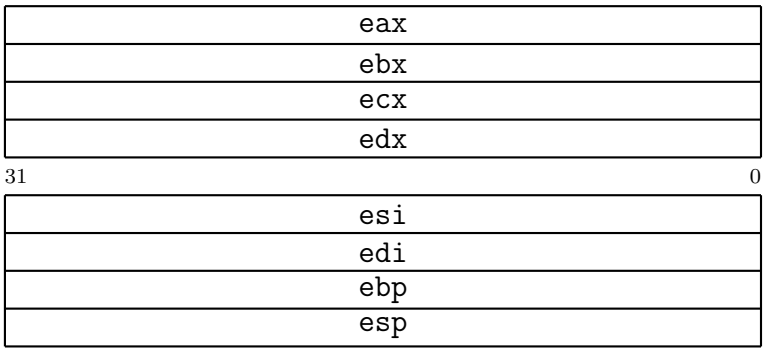


TABELLE 1: *Die x86-Register*

`ax`	*"Accumulator Register"*[18]	`si`	*"Source Index"*
`bx`	*"Base Register"*	`di`	*"Destination Index"*
`cx`	*"Count Register"*	`bp`	*"Base Pointer"*
`dx`	*"Data Register"*	`sp`	*"Stack Pointer"*

TABELLE 2: *Historische Register-Bezeichnungen*

Beim Großteil der Befehle und Adressierungsarten kann nach Belieben jedes der acht Register angegeben werden, einige Befehle verwenden aber fix festgelegte Register. Daher auch die historisch bedingten Namen der Register in Tabelle 2 (teilweise vom 8-Bit-Prozessor 8080 übernommen!).

Der Stackpointer `esp` und meist auch der Basepointer `ebp` haben eine fix vordefinierte Funktion und daher einen vorgegebenen Inhalt, den man nicht wahllos verändern darf. Die anderen Register kann man für beliebige Zwecke verwenden und ändern.

Dazu kommen noch:

- Der ***Program Counter***, der die Adresse des nächsten Befehls enthält. In Intel-Notation: `ip` (16 Bit) bzw. `eip` (32 Bit) (*"Instruction Pointer"*).
- Die ***Flags***, eine Sammlung einzelner Bits, die Status- und Steuer-Informationen enthalten (siehe unten). Von manchen Befehlen werden alle Flag-Bits zusammen ebenfalls als ein einziges Register behandelt (ursprünglich 16 Bit, später auch auf 32 Bit erweitert).

- Die 16 Bit breiten ***Segment-Register***[19]. Ursprünglich 4 Stück: `cs` ("Code Segment"), `ds` ("Data Segment"), `ss` ("Stack Segment") und `es` ("Extra Segment"). Später kamen noch zwei dazu: `fs` und `gs` (ohne tiefere Bedeutung des Namens).
- Die Gleitkomma-, MMX- und SSE-Register.
- Jede Menge Spezial-Register, die nur für Betriebssystem, BIOS und Hardware-Entwickler interessant sind ("Control Register" `cr`*n*, "Debug Register" `dr`*n* u. v. a. m.).

4.3 Die Flags

Im Flag-Register sitzen unter anderem fünf einzelne Bits, die bei jeder arithmetischen oder logischen Operation, jedem Vergleich, und einigen anderen Operationen abhängig vom Ergebnis der Operation gesetzt oder gelöscht werden[20]:

Das Zero-Flag *Z* wird gesetzt, wenn das Ergebnis gleich `0` war, und gelöscht, wenn es von `0` verschieden war.

Das Sign-Flag *S* ist gesetzt, wenn das Ergebnis — betrachtet als Zweierkomplement — negativ war, d. h. das höchstwertigste Bit gesetzt hatte, und gelöscht, wenn das Ergebnis `0` oder positiv war (vorderstes Bit `0`).

Das Carry-Flag *C* wird gesetzt, wenn die letzte Operation — wenn man die Argumente als ***vorzeichenlose*** Binärzahlen betrachtet — einen Überlauf bzw. Übertrag produziert hat, und sonst gelöscht[21].

Das Overflow-Flag *O* wird gesetzt, wenn die letzte Operation — bei Betrachtung der Argumente als Zweierkomplement-Zahlen, d. h. Zahlen ***mit Vorzeichen*** — einen Überlauf bzw. Übertrag produziert hat, und sonst gelöscht.

Das Parity-Flag *P* wird gesetzt, wenn das Ergebnis eine gerade Parität (gerade Anzahl von `1`-Bits) hat, und bei ungerader Parität des Ergebnisses gelöscht.

Diese Flags bleiben bis zur nächsten derartigen Operation erhalten und können von Sprungbefehlen geprüft werden, um abhängig vom Ergebnis der Operation zu verzweigen.

4.4 Die Befehle

Der Aufbau von Befehlen:

Ein $x86$-Maschinenbefehl kann aus 1–16 Bytes bestehen:

- 0–4 Stück je 1 Byte lange ***Prefixe***: Dazu gehören z. B. das Repeat-Prefix für die Stringbefehle, das Lock-Prefix zur Synchronisierung von Speicherzugiffen in Mehrprozessor-Systemen, das Segment-Prefix zur Auswahl des zu verwendenden Segmentregisters, und

[19] Im 16 Bit Real Mode enthalten die Segmentregister die um 4 Bit verschobene Anfangsadresse des jeweiligen Segmentes. Im 16 Bit Protected Mode und im 32 Bit Mode enthalten sie einen Verweis auf den Descriptor des jeweiligen Segmentes: Dieser legt Anfangsadresse, Größe, Zugriffsrechte usw. fest.

[20] Im Unterschied zu anderen Architekturen beeinflussen `mov`-Instruktionen bei einem $x86$-Prozessor die Flags *nicht*: Nach dem Kopieren eines Wertes spiegeln die Flags nach wie vor das Ergebnis der letzten arithmetischen / logischen Operation vor dem `mov` wider und *nicht* den Zustand des kopierten Wertes!

[21] Das Carry-Flag wird auch noch für andere Zwecke verwendet.

die Prefixe zum Umschalten auf 16-Bit-Operanden bzw. 16-Bit-Adressierungsarten im 32-Bit-Mode.
- 1 oder 2 Bytes für den eigentlichen Befehl (***Opcode***).
- 0, 1 oder 2 Bytes für die ***Adressierungsart*** (Intel-Bezeichnung: *"ModR/M-Byte"* und *"SIB-Byte"*).
- 0, 1, 2 oder 4 Bytes für eine ***Adresse*** oder ein ***Offset*** (Intel-Bezeichnung: *"Displacement"*).
- 0, 1, 2 oder 4 Bytes für eine ***Konstante*** als Operand (Intel-Bezeichnung: *"Immediate"*). Diese Konstanten stehen direkt im Befehl (und daher in der `.text`-Section, nicht in `.data`)!

Die "Zwei-Adress-Architektur":

Die $x86$-Architektur ist eine sogenannte "Zwei-Adress-Architektur": Die meisten arithmetischen und logischen Befehle der $x86$-Architektur sind *Zwei-Adress-Befehle*: Sie haben nur *zwei* Operanden (wobei nur *einer* davon im Speicher liegen kann, der zweite muß ein Register oder eine Konstante sein), das Ergebnis wird in einem der beiden Operanden gespeichert.

Es ist also nicht möglich, eine Operation der Form $c = a + b$ (mit verschiedenem a, b und c) mit einem einzigen Befehl durchzuführen, man muß sie in zwei oder drei Befehle aufspalten, beispielsweise $r = a$; $r \mathrel{+}= b$; $c = r$ (wenn alle Operanden im Speicher liegen) oder $c = a$; $c \mathrel{+}= b$ (wenn c in einem Register liegt).

16- und 32-Bit-Befehle:

Während für dasselbe Mnemonic zwei verschiedene binäre Maschinencodes existieren, wenn der Befehl eine Variante für 8-Bit-Daten und eine für 16/32-Bit–Daten hat, gibt es in der $x86$-Architektur *keine* Unterscheidung zwischen der 16- und 32-Bit-Variante eines Befehls: Für beide Datenbreiten wird derselbe Befehl verwendet!

Läuft der Prozessor im 32-Bit-Modus, wird ein solcher Befehl als 32-Bit-Befehl ausgeführt, läuft er im 16-Bit-Modus, bewirkt derselbe Befehl die entsprechende 16-Bit-Operation. Soll im 32-Bit-Modus ein Befehl nur auf 16-Bit-Daten oder -Registern ausgeführt werden, so wird er mit einem 1 Byte langen *Prefix-Befehl* versehen, der für diesen einen Befehl in den 16-Bit-Modus umschaltet.

16-Bit-Operationen benötigen daher im 32-Bit-Modus eine um 1 Byte längere Codierung als die entsprechenden 32-Bit-Befehle (und sind je nach Prozessor auch langsamer), 8-Bit-Operationen hingegen kosten nichts extra.

Weiters sind (außer bei einigen Spezialbefehlen) keine "gemischten" Operandengrößen erlaubt: Beide Operanden müssen gleich groß sein; es ist beispielsweise nicht möglich, eine 16-Bit-Zahl zu einem 32-Bit-Register zu addieren. Die meisten arithmetischen / logischen Operationen können allerdings eine 8-Bit-Konstante mit einem 16/32-Bit-Operanden verknüpfen, diese wird zuerst *mit Vorzeichen* auf die gewünschte Länge erweitert.

4.5 Die Adressierungsarten

Die zur Verfügung stehenden Adressierungsarten und deren Codierung im Befehl unterscheiden sich zwischen 16-Bit-Mode und 32-Bit-Mode, wir beschränken uns auf den 32-Bit-Fall.

Die Adresse eines Operanden im Speicher ist die Summe folgender Teile in beliebiger Kombination, wobei jeder Teil weggelassen werden kann:

Zielobjekt	**Adressierung**
Globale Variable `myvar`	`[myvar]`
Globales `int`-Array `arr`, Element mit Index `eax`	`[arr+4*eax]`
Globale Struktur `rec`, Feld mit Offset `12`	`[rec+12]`[23]
Lokale Variable, Offset `8` vom Basepointer	`[ebp-8]`
Lokales `char`-Array, Offset `1040`, Element mit Index `ecx`	`[ebp-1040+ecx]`
Pointer auf Variable in `esi`	`[esi]`
Pointer auf `int`-Array in `edi`, Element mit Index `eax`	`[edi+4*eax]`
Pointer auf Struktur in `edx`, Feld mit Offset `myfield`	`[edx+myfield]`
Pointer auf Struktur in `esi`, darin `short`-Array mit Offset `arr`, davon Element mit Index `edx`	`[esi+arr+2*edx]`

Tabelle 3: *Beispiele für Adressierungsarten*

- Einer 1 Byte oder 4 Bytes langen Zahl mit Vorzeichen (konstante ***Adresse*** oder ***Offset***[22]).
- Dem Inhalt eines ***Base-Registers***: Als Base-Register kann jedes 32-Bit-Register verwendet werden.
- Dem Inhalt eines ***Index-Registers*** mal `1`, `2`, `4` oder `8` (für 1/2/4/8 Bytes große Array-Elemente): Als Index-Register kann jedes 32-Bit-Register außer `esp` (dem Stackpointer) verwendet werden.

In `nasm`-Notation wird ein Operand im Speicher immer mit `[ ]` angegeben, zwischen den `[ ]` stehen die einzelnen Bestandteile der Adresse in beliebiger Reihenfolge als arithmetischer Ausdruck. Einige typische Fälle sind in Tabelle 3 aufgelistet.

In den Intel-Handbüchern wird bei einem Befehl *reg/mem* als Operand angegeben, wenn sowohl ein Register als auch eine Speicher-Adressierungsart für diesen Operanden zulässig ist, *reg* steht für einen Operanden, der nur ein Register sein kann, und *imm* ("immediate value") steht für eine Konstante als Operand. Ich verwende diese Notation im Folgenden ebenfalls.

5 x86-Befehle (die wichtigsten)

Befehle zum Datentransfer:

- `mov` *reg/mem*`,` *reg* oder `mov` *reg*`,` *reg/mem* oder `mov` *reg/mem*`,` *imm* kopiert ein Byte / Wort / Doppelwort vom zweiten zum ersten Operanden.
- `movzx` *reg*`,` *reg/mem* (*move with zero extension*) kopiert den kürzeren (Byte oder Wort) zweiten Operanden in den längeren (Wort oder Doppelwort) ersten Operanden (muß ein Register sein) und füllt dabei die oben dazugekommenen Bytes mit `0`-Bits, ist also für ***vorzeichenlose*** Zahlen gedacht.

  `movsx` *reg*`,` *reg/mem* (*move with sign extension*) macht das gleiche, aber füllt die neuen Bytes mit `1`-Bits, wenn das vorderste Bit des ursprünglichen Wertes `1` war (der Wert

[22] In Intel-Terminologie: *"Displacement"*.

[23] `nasm` zählt die beiden Werte zusammen und macht daraus eine Zahl.

betrachtet als Zweierkomplement-Zahl also negativ war), und mit `0`-Bits sonst. Eine *Sign Extension* braucht man also, wenn man aus einer kurzen Integer-Zahl ***mit Vorzeichen*** eine lange Integer-Zahl mit Vorzeichen mit dem gleichen Wert machen will: Eine Zero Extension würde aus negativen Werten falsche positive Werte machen, bei der Sign Extension bleibt das Vorzeichen erhalten.

`cbw` (*convert byte to word*) macht eine Sign Extension von `al` nach `ax`, `cwd` (*convert word to double*) eine von `ax` nach `dx:ax`, `cwde` (*convert word to double extended*) eine von `ax` nach `eax`, und `cdq` (*convert double to quad*) eine von `eax` nach `edx:eax` (notwendig vor allem in Kombination mit `idiv`).

- `xchg` *reg, reg/mem* (*exchange*) vertauscht den Inhalt der beiden Operanden (man erspart sich damit einen Dreieckstausch).
- `bswap` *reg* (*byte swap*) vertauscht Byte 1 und 4 sowie Byte 2 und 3 eines 32-Bit-Registers (verwandelt den Inhalt von "Big Endian" nach "Little Endian" und umgekehrt: Aus `0x12345678` wird `0x78563412`).

Arithmetische und logische Befehle:

- Folgende Befehle haben zwei Operanden:

  `add` (*add*) addiert den zweiten Operanden zum ersten.
  `adc` (*add with carry*) macht das gleiche, zählt aber zusätzlich noch das Carry-Bit dazu (als Übertrag in die niederwertigste Stelle).
  `sub` (*subtract*) subtrahiert den zweiten Operanden vom ersten.
  `sbb` (*subtract with borrow*) macht das gleiche, zieht aber zusätzlich noch das Carry-Bit ab (als Übertrag in die niederwertigste Stelle).
  `and` (*and*) speichert im ersten Operanden das bitweise logische Und der beiden Operanden. Es wird unter anderem dazu verwendet, Bits zu löschen: `and eax, 0x00000fff` läßt die letzten 12 Bits von `eax` unverändert und löscht alle anderen.
  Handelt es sich dabei wie in diesem Fall um die n hintersten Bits, ist das gleichbedeutend mit einer Restbildung modulo 2^n.
  `or` (*or*) speichert im ersten Operanden das bitweise logische Oder der beiden Operanden. Es wird unter anderem dazu verwendet, Bits zu setzen: `or eax, 0xfffff000` läßt die letzten 12 Bits von `eax` unverändert und setzt alle anderen.
  `xor` (*xor*) speichert im ersten Operanden das bitweise logische Exklusiv-Oder der beiden Operanden. Es wird unter anderem dazu verwendet, Bits zu invertieren: `xor eax, 0x00000001` invertiert das hinterste Bit und läßt alle anderen Bits von `eax` unverändert.
  Weiters sieht man oft `xor eax, eax` (oder dasselbe für ein anderes Register): Das ist ein beliebter Trick, um ein Register zu löschen (auf 0 zu setzen), dieser Befehl ist wesentlich kürzer als `mov eax, 0`.
  `cmp` (*compare*) subtrahiert den zweiten Operanden vom ersten (wie bei `sub`), speichert aber das Ergebnis nicht (beide Operanden bleiben unverändert): Es werden nur die Flags gesetzt (meist als Grundlage für einen nachfolgenden bedingten Sprungbefehl).
  `test` (*test*) berechnet wie `and` das bitweise logische Und der beiden Operanden, ignoriert aber wie `cmp` das Ergebnis: Der Befehl wirkt sich nur auf die Flags aus.
  Auch diesen Befehl sieht man oft mit zwei Mal demselben Register: `test eax, eax` setzt beispielsweise das Sign- und Zero-Flag abhängig davon, ob `eax` positiv, negativ oder `0` ist.

Für alle diese Befehle sind folgende Kombinationen von Operanden (jeweils beide Byte, Wort oder Doppelwort) zulässig:

  * *reg*, *reg/mem*
  * *reg/mem*, *reg*
  * *reg/mem*, *imm* (wobei *imm* entweder die volle Länge des anderen Operanden oder nur 8 Bit haben kann)

- Folgende Befehle haben einen einzigen *reg/mem*-Operanden (ebenfalls Byte, Wort oder Doppelwort):

  `inc` (*increment*) erhöht den Operanden um `1`.
  `dec` (*decrement*) erniedrigt den Operanden um `1`.
  `neg` (*negate*) ersetzt den Operanden durch sein Zweierkomplement (unäres −).
  `not` (*not*) ersetzt den Operanden durch sein Einserkomplement (alle Bits invertiert).

  Achtung:
  Im Unterschied zu allen anderen arithmetischen bzw. logischen Operationen lassen `inc` und `dec` das Carry-Flag *unverändert* (die anderen Flags werden wie üblich gesetzt). `not` läßt die Flags überhaupt völlig unverändert.

- `lea` *reg*, *mem* (*load effective address*) berechnet die *Adresse* des Operanden *mem* und speichert diese in *reg*. Mit anderen Worten: In *reg* wird ein Pointer auf *mem* abgelegt. Obwohl der zweite Operand ein Speicher-Operand ist, wird also *kein* Speicherzugriff gemacht! Da `lea` auf den Operanden selbst gar nicht zugreift, kann man auch keine Operanden-Größe angeben (die Adresse des Operanden ist unabhängig davon, ob es sich bei diesem um ein Byte, Wort oder Doppelwort handelt).

  Obwohl dieser Befehl primär zum Berechnen von Pointern gedacht ist, läßt er sich als universeller Additionsbefehl für Doppelworte (mit viel mehr Möglichkeiten als `add`) mißbrauchen (`lea` setzt allerdings die Flags *nicht*!): Man kann damit die Summe aus

  * einem Register,
  * einem zweiten Register (eventuell mal 2, 4 oder 8),
  * und einer Konstanten

  in einem Schritt berechnen und in einem beliebigen Register speichern.

- `mul` (*multiply*) multipliziert vorzeichenlose Zahlen, `imul` (*integer multiply*) multipliziert Zweierkomplement-Zahlen.

  Achtung:
  Der Assembler weiß nicht, ob eine Speicherstelle oder ein Register eine Zahl mit oder ohne Vorzeichen enthält, und auch zur Laufzeit ist das nicht unterscheidbar. Es kommt daher keine Fehlermeldung, wenn man den "falschen" Befehl verwendet[24]!

  Beide haben nur *einen reg/mem*-Operanden (*imm* ist nicht möglich!), der andere ist fix vorgegeben, ebenso das Ergebnis, das *doppelt so lang* wie die Operanden ist:

  * Die Byte-Variante multipliziert `al` mit dem Operanden und speichert das Ergebnis in `ax`.
  * Die Wort-Variante multipliziert `ax` mit dem Operanden und speichert das Ergebnis in `dx:ax` (das höherwertige Wort des Ergebnisses in `dx`, das niederwertige in `ax`).

[24] Bei der Addition und der Subtraktion funktioniert bekanntlich derselbe Befehl für vorzeichenlose und vorzeichenbehaftete Zahlen.

  - Die Doppelwort-Variante multipliziert `eax` mit dem Operanden und speichert das Ergebnis (64 Bit!) in `edx:eax`.

- Von `imul` gibt es auch Varianten mit 2 oder 3 Operanden:

  `imul` *reg*`,` *reg/mem/imm* multipliziert *reg* mit *reg/mem/imm* und speichert das Ergebnis in *reg*.
  `imul` *reg*`,` *reg/mem*`,` *imm* multipliziert *reg/mem* mit *imm* und speichert das Ergebnis in *reg* (Multiplikand und Ergebnis können also in *verschiedenen* Registern stehen!).

  Bei beiden ist das Ergebnis nur so lang wie die Operanden (und nicht doppelt so lang wie beim normalen `imul`), und es gibt beide nur für Wort- und Doppelwort-Operanden, nicht für 1 Byte lange Zahlen (aber sehr wohl mit vollem und mit auf 1 Byte verkürztem *imm*!).

  Von `mul` gibt es diese Varianten nicht, aber die untere Hälfte des Ergebnisses einer Multiplikation ist unabhängig davon, ob man die Operanden als vorzeichenbehaftetete oder vorzeichenlose Zahlen betrachtet, und die obere Hälfte (die sich tatsächlich unterscheidet) wird bei diesen `imul`-Varianten ohnehin ignoriert. Man kann also für beide Arten von Zahlen `imul` verwenden und braucht kein `mul`.

  Achtung:
  Nach `mul` und `imul` sind das Carry- und Overflow-Flag gesetzt, wenn das Ergebnis nicht in die untere Hälfte gepaßt hat, und sonst gelöscht. Die anderen Flags sind *undefiniert!*

- Spiegelbildlich dazu gibt es die Befehle `div` (*divide*) für vorzeichenlose Division und `idiv` (*integer divide*) für die Division von Zweierkomplementen.

  Beide dividieren einen fix vorgegebenen, doppelt langen Operanden durch einen *reg/mem*-Operanden (*imm* ist nicht möglich!) und liefern *zwei* Ergebnisse: Den Quotient und den Rest.

  - Die Byte-Variante dividiert `ax` durch den Operanden und speichert den Quotienten in `al`, den Rest in `ah`.
  - Die Wort-Variante dividiert `dx:ax` durch den Operanden und speichert den Quotienten in `ax`, den Rest in `dx`.
  - Die Doppelwort-Variante dividiert `edx:eax` (64 Bit!) durch den Operanden und speichert den Quotienten in `eax`, den Rest in `edx`.

  Die `imul`-Sonderformen gibt es für `idiv` nicht.

  Achtung:
  Die Flags sind nach `div` und `idiv` *undefiniert!*

  Achtung:
  Während alle anderen Overflows stillschweigend ignoriert werden, führt eine Division durch `0` oder eine Division, bei der der Quotient nicht in das dafür vorgesehene Register paßt, zum Programm-Abbruch!

  Achtung:
  Multiplikationen sind langsamer als andere arithmetische Befehle, und Divisionen sind nochmals wesentlich langsamer als Multiplikationen!

Jumps, Flags und Bedingungen:

- `jmp` *adr* setzt die Programmausführung mit dem Befehl an der angegebenen Adresse *adr*

fort[25]. Der Befehl nach dem `jmp` kann daher nur dann jemals ausgeführt werden, wenn er von anderer Stelle angesprungen wird, und wird daher in jedem Fall ein Label haben.

Für die segmentierten Betriebsarten der $x86$-Architektur gibt es Varianten von `jmp`, die auch das Code-Segment-Register ändern, diese brauchen wir aber unter Linux nicht.

- `jmp` *reg/mem* entnimmt die Adresse des nächsten auszuführenden Befehls aus dem 32-Bit-Operanden *reg/mem*, der einen Pointer auf einen Befehl enthalten sollte. Damit kann das Sprungziel zur Laufzeit berechnet werden, `switch`-Statements in C werden beispielsweise auf diese Art implementiert.

- *Bedingte Sprungbefehle* (siehe Tabelle 4) springen nur dann an die angegebene Adresse, wenn die Flags die angegebene Bedingung erfüllen (sie stehen daher normalerweise hinter einem Befehl, der die Flags setzt) — wenn nicht, ignoriert der Prozessor den Sprung und macht ganz normal mit dem nächsten Befehl nach dem bedingten Sprung weiter.

  Zum einen gibt es Sprungbefehle, die prüfen, ob ein einzelnes Flag gesetzt bzw. gelöscht ist, und genau so wie das Flag heißen. Da das Ergebnis eines Vergleiches mit `cmp` genau dann `0` ist, wenn die beiden Operanden gleich sind, wurden bei Zero — ebenso wie bei Parity — leichter zu merkende Namen für die gleichen Befehle eingeführt.

  Daneben gibt es noch Sprungbefehle, die bestimmte Kombinationen von Flags prüfen, um nach einem Vergleich von zwei Zahlen mit `cmp` auf $<$, $\leq$, $>$ und $\geq$ prüfen zu können:

  * "Below" und "above" bezieht sich immer auf die Vergleiche ***vorzeichenloser*** Zahlen,
  * "less" und "greater" beschreibt hingegen Vergleiche von Zahlen ***mit Vorzeichen***.
  * Vergleiche "gemischter" Zahlen sind nicht sinnvoll!

  Diese bedingten Sprünge werden dazu verwendet, um die Kontrollstrukturen höherer Programmiersprachen nachzubilden (siehe Tabelle 5). Auch `&&` und `||` werden durch bedingte Sprünge implementiert.

  Eine `for`-Schleife löst man im Prinzip in den Initialisierungscode und eine anschließende `while`-Schleife mit dem Inkrementierungscode am Ende des Schleifenrumpfes auf, aber es gibt sehr viele Tricks, um `for`-Schleifen zu optimieren (z. B. wie bei der `while`-Schleife mit dem Test am Ende in der Tabelle, um einen unbedingten Sprung in jedem Durchlauf zu sparen).

  Achtung:
  Sprünge sind auf modernen Prozessoren — im Vergleich zu anderen Operationen — relativ "teure", d. h. langsame Befehle. Bei kurzen, geschwindigkeitskritischen Schleifen ist es daher sinnvoll, die Anzahl der bei jedem Durchlauf ausgeführten Sprungbefehle sorgfältig zu minimieren.

- Neben den bedingten Sprüngen, die die Flags prüfen, gibt es auch solche für das `ecx`-Register:

  `jecxz` *adr* (*jump ecx zero*) springt, wenn `ecx 0` ist, und macht ansonsten mit dem nächsten Befehl weiter. *Achtung:* Von diesem Befehl gibt es *keine* negierte Variante, die bei ungleich `0` springt!

---

[25] *adr* wird dabei im Befehl nicht als absolute Adresse, sondern als 8 oder 32 Bit langer Offset (mit Vorzeichen) relativ zur aktuellen Programmadresse gespeichert (Intel-Bezeichnung: "Displacement") und beim Sprung zum Program Counter `eip` addiert.

Da die Sprungziele fast immer im Umkreis von ± 128 Bytes rund um den jeweiligen Sprungbefehl liegen, hat das den Vorteil, daß die Mehrzahl der Sprungbefehle in nur 2 statt 5 Bytes codieren werden können.

In Intel-Terminologie heißt ein Sprung mit 8-Bit-Offset `short`, einer mit 32-Bit-Offset `near` und einer in ein anderes Code-Segment `far`.

`jz`	Jump if zero	$\mathbf{Z} = 1$
`je`	Jump if equal	$\mathbf{Z} = 1$
`jnz`	Jump if not zero	$\mathbf{Z} = 0$
`jne`	Jump if not equal	$\mathbf{Z} = 0$
`js`	Jump if sign	$\mathbf{S} = 1$
`jns`	Jump if not sign	$\mathbf{S} = 0$
`jc`	Jump if carry	$\mathbf{C} = 1$
`jnc`	Jump if not carry	$\mathbf{C} = 0$
`jo`	Jump if overflow	$\mathbf{O} = 1$
`jno`	Jump if not overflow	$\mathbf{O} = 0$
`jp`	Jump if parity	$\mathbf{P} = 1$
`jpe`	Jump if parity even	$\mathbf{P} = 1$
`jnp`	Jump if not parity	$\mathbf{P} = 0$
`jpo`	Jump if parity odd	$\mathbf{P} = 0$
`jb`	Jump if below	$\mathbf{C} = 1$
`jnae`	Jump if not above or equal	$\mathbf{C} = 1$
`jnb`	Jump if not below	$\mathbf{C} = 0$
`jae`	Jump if above or equal	$\mathbf{C} = 0$
`ja`	Jump if above	$\mathbf{C} = 0 \wedge \mathbf{Z} = 0$
`jnbe`	Jump if not below or equal	$\mathbf{C} = 0 \wedge \mathbf{Z} = 0$
`jna`	Jump if not above	$\mathbf{C} = 1 \vee \mathbf{Z} = 1$
`jbe`	Jump if below or equal	$\mathbf{C} = 1 \vee \mathbf{Z} = 1$
`jl`	Jump if less	$(*)$
`jnge`	Jump if not greater or equal	$(*)$
`jnl`	Jump if not less	$\neg(*)$
`jge`	Jump if greater or equal	$\neg(*)$
`jg`	Jump if greater	$\neg(*) \wedge \mathbf{Z} = 0$
`jnle`	Jump if not less or equal	$\neg(*) \wedge \mathbf{Z} = 0$
`jng`	Jump if not greater	$(*) \vee \mathbf{Z} = 1$
`jle`	Jump if less or equal	$(*) \vee \mathbf{Z} = 1$

$$(*) \ldots (\mathbf{S} = 0 \wedge \mathbf{O} = 1) \vee (\mathbf{S} = 1 \wedge \mathbf{O} = 0)$$

TABELLE 4: *Bedingte Sprungbefehle*

`loop` *adr* dekrementiert `ecx` zuerst um `1`. Wenn `ecx` danach ungleich `0` ist, wird gesprungen, bei `ecx` gleich `0` nicht.

`loope` *adr* oder `loopz` *adr* oder `loopne` *adr* oder `loopnz` *adr* (*loop if equal / zero / not equal / not zero*) arbeiten wie `loop`, aber es wird nur gesprungen, wenn auch das Zero-Flag gesetzt / nicht gesetzt ist.

Es lohnt sich daher, bei Zählschleifen darüber nachzudenken, ob man nicht die übliche Zählrichtung umdrehen kann, damit man diese Befehle verwenden kann.

Achtung:
Im Unterschied zu den normalen bedingten Sprüngen gibt es bei `jecxz` und `loop`*cc* nur

`if (eax>=10) {`
thencode
`}`
	`cmp eax, 10`
	`jl endif ; Bedingung invertiert!`
	thencode
	`endif:`

`if (eax>=10) {`
thencode
`}`
`else if (eax<=-10) {`
elseifcode
`}`
`else {`
elsecode
`}`
	`cmp eax, 10`
	`jl elif ; Bedingung invertiert!`
	thencode
	`jmp endif`
	`elif:`
	`cmp eax, -10`
	`jg else ; Bedingung invertiert!`
	elseifcode
	`jmp endif`
	`else:`
	elsecode
	`endif:`

`if ((eax==1) ||`
`((eax>=10) &&`
`(eax<100))) {`
thencode
`}`
	`cmp eax, 1`
	`je then ; Bedingung normal`
	`cmp eax, 10`
	`jl endif ; Bedingung invertiert!`
	`cmp eax, 100`
	`jge endif ; Bedingung invertiert!`
	`then:`
	thencode
	`endif:`

`while (esi!=NULL) {`
loopcode
`}`
	`while:`
	`test esi, esi`
	`jz endwhile ; Bedingung invertiert!`
	loopcode
	`jmp while`
	`endwhile:`

`while (esi!=NULL) {`
loopcode
`}`
	Schneller, aber unschön:
	`jmp whiletest`
	`while:`
	loopcode
	`whiletest:`
	`test esi, esi`
	`jnz while ; Bedingung normal`

`do {`
loopcode
`} while (esi!=NULL)`
	`do:`
	loopcode
	`test esi, esi`
	`jnz do ; Bedingung normal`

Tabelle 5: *Beispiele für Kontrollstrukturen*

die "kurzen" Varianten mit 8-Bit-Offset, das Sprungziel muß daher innerhalb von ±128 Bytes liegen!

- Für den Umgang mit boolschen Werten gibt es den Befehl `set`*cc reg/mem* (wobei für *cc* die gleichen Bedingungs-Kürzel wie bei den bedingten Sprüngen `j`*cc* eingesetzt werden können). Erfüllen die Flags die Bedingung *cc*, so wird im angegebenen Operanden der Wert `1` gespeichert, sonst `0`. Diesen Befehl gibt es leider nur für 1 Byte große Operanden, nicht für Doppelworte. Da C boolsche Werte in `int`-Variablen speichert, braucht man in diesem Fall ein zusätzliches `movzx`.
- Das Carry-Flag kann man auch händisch manipulieren: `clc` (*clear carry*) löscht es, `stc` (*set carry*) setzt es, und `cmc` (*complement carry*) invertiert es.

Calls und Stack-Befehle:

- `call` *adr* ruft die an der Adresse *adr* beginnende Funktion auf: Zuerst wird der aktuelle Wert des Program Counters `eip` auf den Stack gepusht, damit das Programm weiß, wo es nach der Rückkehr aus der aufgerufenen Funktion weitermachen soll, und dann wird der Program Counter auf die angegebene Adresse *adr* gesetzt und dadurch der erste Befehl der angegebenen Funktion ausgeführt[26].
- Wie bei `jmp` gibt es auch ein `call` *reg/mem* für "indirekte" Funktionsaufrufe: Der Inhalt des Operanden liefert die Adresse der anzuspringenden Funktion, d. h. es wird keine fixe, sondern eine in einem Function Pointer gespeicherte Funktion aufgerufen.
- `ret` (*return*) kehrt aus einer Funktion zum Aufrufer zurück: Es poppt die vom `call` abgelegte Return-Adresse vom Stack nach `eip` und setzt die Programmausführung an dieser Adresse — dem Befehl unmittelbar hinter dem `call` — fort.
- Die Befehle `enter` *x*`,` *y* und `leave` dienen zum Anlegen von Stack-Frames für lokale Variablen und zum Verwalten des Basepointers in C- oder PASCAL-Programmen, sie werden im Kapitel über die Anbindung von C-Programmen besprochen.
- Neben den Calls gibt es für Systemaufrufe, fatale Fehler, Debugger-Breakpoints usw. auch noch den Software-Interrupt-Befehl `int` *n*, siehe Kapitel Systemaufrufe[27].
- `push` *reg/mem/imm* legt den Operanden als neues oberstes Element am Stack ab (und erniedrigt davor `esp` entsprechend), `pop` *reg/mem* speichert das oberste Element des Stacks im Operanden und erhöht den Stackpointer danach, um es vom Stack zu entfernen.

  `pusha` sichert alle Register (außer `eip` und den Flags) auf dem Stack, `popa` lädt sie wieder zurück[28].

  `pushf` legt die Flags auf den Stack, `popf` holt sie zurück.

  Achtung:
  Im 32-Bit-Mode darf man nur Doppelworte (4 Bytes) oder Vielfache davon pushen und poppen, niemals einzelne Bytes oder Worte!

---

[26] Wie bei `jmp` ist *adr* im `call`-Befehl nicht als absolute Adresse, sondern als Offset relativ zum aktuellen Program Counter gespeichert.

Weiters gibt es wie bei `jmp` auch ein `call` und ein `ret` mit Einbeziehung des Code-Segment-Registers, wir brauchen sie aber beim 32 Bit Flat Memory Model unter Linux nicht.

[27] *n* ist die Nummer des auszuführenden Software-Interrupts, `0` bis `255`: Diese Nummer wird als Index in die Interrupt-Tabelle im Kernel verwendet; dort ist die Anfangsadresse jener Kernelfunktion gespeichert, die für den jeweiligen Interrupt zuständig ist.

[28] Der am Stack abgelegte Wert für `esp` wird dabei ignoriert.

Achtung:
`push` und `pop` müssen schön paarweise zusammenpassen (oder man korrigiert `esp` entsprechend), sonst erfolgt spätestens beim nächsten `ret` ein Absturz, wenn `esp` nicht wie erwartet auf die Rücksprungadresse zeigt!

Bit-Operationen:

- *Shifts und Rotates* gibt es in 8 Varianten:

  `shl` oder `shr` (*shift left/right*) verschieben die Bits "logisch" nach links bzw. rechts[29]: Die herausgeschobenen Bits gehen verloren, und am anderen Ende werden `0`-Bits eingefügt.

  `sal` oder `sar` (*shift arithmetic left/right*) verschieben die Bits "arithmetisch" nach links bzw. rechts: `sal` ist ident zu `shl` (es wird rechts mit `0`-Bits aufgefüllt), aber `sar` schiebt links Kopien des vordersten Bits nach: Ist die Zahl negativ, wird sie links mit `1`-Bits aufgefüllt, ist sie positiv, mit `0`-Bits.

  `rol` oder `ror` (*rotate left/right*) bewirken eine 8- bzw. 32-Bit-Rotation nach links oder rechts: Bits, die an einem Ende herausfallen, werden am anderen wieder hineingesteckt.

  `rcl` oder `rcr` (*rotate carry left/right*) bewirken zusammen mit dem Carry-Flag eine 9- bzw. 33-Bit-Rotation nach links bzw. rechts: Der alte Wert des Carry-Flags wird an einem Ende hineingesteckt, und das am anderen Ende herausfallende Bit wird neu im Carry-Flag gespeichert.

  Allen gemeinsam ist, daß ein 8 oder 32 Bit langer *reg/mem*-Operand manipuliert wird, d. h. das Ergebnis ersetzt den ursprünglichen Wert. Außerdem wird in allen 8 Fällen das letzte herausgeschobene Bit im Carry-Flag gespeichert.

  Die Bits des Operanden können um eine fixe Anzahl von Stellen (`1` bis `31`) geschoben bzw. rotiert werden (`shl` *reg/mem*`,` *imm* usw.), oder um einen im Register `cl` gespeicherten variablen Wert (`shl` *reg/mem*`, cl` usw.).

  Shifts und Rotates dienen einerseits zum Einsetzen und Herausfiltern einzelner Bit-Bereiche eines Operanden (wenn eine Zahl beispielsweise in den Bits 11-27 eines Doppelwortes verpackt ist), andererseits erlauben sie eine schnelle Multiplikation und Division mit Zweierpotenzen (deutlich schneller als `mul` und `div`): `shl` multipliziert vorzeichenlose Zahlen mit 2^n, `shr` dividiert sie durch 2^n, `sal` und `sar` leisten das gleiche für Zweierkomplement-Zahlen.

- `bt` *reg/mem reg/imm* (*bit test*) testet das durch den zweiten Operanden angegebene Bit im ersten Operanden, indem es das betreffende Bit ins Carry-Flag kopiert.

  Ist der erste Operand ein Register oder der zweite Operand eine Konstante, muß der zweite Operand zwischen `0` und `31` liegen, wobei `0` ist das niederwertigste Bit ist.

  Ist der erste Operand ein Speicher-Operand und der zweite Operand ein Register, so darf dieses Register eine beliebige 32-Bit-Zahl n mit Vorzeichen enthalten: Der Prozessor rechnet zuerst von der für den ersten Operanden angegebenen Speicheradresse entsprechend viele Bytes vor oder zurück und betrachtet dann jenes Speicherwort, daß das n Bits vom ursprünglichen Operanden entfernte Bit enthält.

  Die Befehle `btc` (*bit test & complement*), `btr` (*bit test & reset*) und `bts` (*bit test & set*) leisten das gleiche, invertieren / löschen / setzen das angegebene Bit aber danach.

[29] Nach links heißt in Richtung höchstwertigstes Bit, nach rechts in Richtung niederwertigstes Bit.

- `bsf` *reg*`,` *reg/mem* (*bit scan forward*) und `bsr` *reg*`,` *reg/mem* (*bit scan reverse*) durchsuchen den 32-Bit-Operanden *reg/mem* nach dem hintersten (`bsf`) / vordersten (`bsr`) `1`-Bit, speichern dessen Index (`0`–`31`) im Register *reg*, und löschen das Zero-Flag. Sind alle Bits von *reg/mem* `0`, so wird das Zero-Flag gesetzt, und *reg* ist undefiniert.

Stringbefehle:

- Die Stringbefehle sind besonders effiziente Kurzbefehle für Operationen auf aufeinanderfolgenden Bytes, Worten oder Doppelworten im Speicher: Sie kombinieren eine Operation auf ein oder *zwei* Operanden im Speicher und das Rauf- oder Runterzählen der zur Adressierung verwendeten Register mit einer direkt in Hardware ausgeführten Schleife.

  Die Adresse der zu bearbeitenden Daten muß in den Registern `esi` bzw. `edi` stehen und wird bei jedem Stringbefehl automatisch entsprechend erhöht (bei Byte-Befehlen um `1`, bei Wort-Befehlen um `2`, bei Doppelwort-Befehlen um `4`).

  `lodsb` oder `lodsw` oder `lodsd` (*load string byte / ...*) lädt das durch `esi` adressierte Byte / Wort / Doppelwort in das Register `al` / `ax` / `eax`.
  `stosb` oder `stosw` oder `stosd` (*store string byte / ...*) speichert das Byte / Wort / Doppelwort aus dem Register `al` / `ax` / `eax` an der durch `edi` gegebenen Adresse.
  `movsb` oder `movsw` oder `movsd` (*move string byte / ...*) kopiert das durch `esi` adressierte Byte / Wort / Doppelwort an die durch `edi` gegebenen Adresse.
  `scasb` oder `scasw` oder `scasd` (*scan string byte / ...*) vergleicht das Register `al` / `ax` / `eax` mit dem durch `edi` adressierten Byte / Wort / Doppelwort: Wie bei `cmp` werden die beiden voneinander subtrahiert, das Ergebnis wird ignoriert, aber die Flags werden entsprechend gesetzt.
  `cmpsb` oder `cmpsw` oder `cmpsd` (*compare string byte / ...*) vergleicht (wie oben) das durch `esi` adressierte Byte / Wort / Doppelwort mit dem durch `edi` adressierten.

  Nachdem alle beteiligten Register und Operanden fix festgelegt sind, haben die Stringbefehle keine expliziten Operanden.

  Achtung:
  Es gibt ein Direction-Flag, das man mit `cld` löschen und mit `std` setzen kann, und das die "Arbeitsrichtung" der Stringbefehle steuert: Ist es `1`, werden die Adressen nach jedem Zugriff erniedrigt statt erhöht, der Befehl läuft "rückwärts" durch den Speicher.

  Der Inhalt dieses Flags ist nicht vorhersehbar. Man muß es vor *jeder* Verwendung von Stringbefehlen explizit setzen oder löschen, sonst ist es dem Zufall überlassen, ob der Befehl in die richtige oder in die falsche Richtung läuft!

- Zu den Stringbefehlen gibt es die *Repeat-Prefixes* `rep` (*repeat*), `repe` (*repeat while equal*) und `repne` (*repeat while not equal*). Diese können unmittelbar vor einem Stringbefehl (und nur dort!) angegeben werden und bewirken, daß dieser Stringbefehl in einer Hardware-Schleife `ecx` Mal wiederholt wird:

  * *Vor* jeder Ausführung des Stringbefehls wird geprüft, ob `ecx` gleich `0` ist. Wenn ja, endet die Schleife, wenn nein, wird der Befehl ausgeführt.
  * *Nach* jeder Ausführung des Befehls wird `ecx` um `1` heruntergezählt; danach beginnt die Schleife wieder mit dem Test von `ecx`.
  * `repe` und `repne` prüfen zusätzlich *nach* jeder Ausführung des Stringbefehls das Zero-Flag und beenden die Schleife, wenn die angegebene Bedingung nicht erfüllt ist (auch wenn `ecx` noch nicht `0` ist).

  Damit ergeben sich folgende sinnvolle Kombinationen:

`rep stos`*x* füllt `ecx` Bytes / Worte / Doppelworte im Speicher ab Adresse `edi` mit dem Wert aus dem Register `al` / `ax` / `eax`.

`rep movs`*x* kopiert `ecx` Bytes / Worte / Doppelworte vom Speicher ab Adresse `esi` in den Speicher ab Adresse `edi`.

`repe scas`*x* oder `repne scas`*x* sucht in den `ecx` Bytes / Worten / Doppelworten im Speicher ab Adresse `edi` das erste, das ungleich / gleich dem Wert in Register `al` / `ax` / `eax` ist.

`repe cmps`*x* oder `repne cmps`*x* vergleicht maximal `ecx` Bytes / Worte / Doppelworte im Speicher ab Adresse `esi` mit denen ab Adresse `edi`, stoppt beim ersten ungleichen / gleichen Paar.

Achtung:
Wenn `repe` oder `repne` auf Grund des Zero-Flags enden (und nicht auf Grund von `ecx`), sind `esi` bzw. `edi` sowie `ecx` trotzdem schon geändert, d. h. die Pointer zeigen schon auf das nächste Element, und nicht auf das gefundene! Man muß entsprechend zurückrechnen!

Gleitkomma-Befehle, MMX- und SSE-Befehle: ... lassen wir aus.

6 Programmstart und Programmende

Direkt als Assembler-Programm:

- Der Befehl, bei dem die Ausführung beginnen soll, bekommt das Label `_start` (in der Section `.text`), es wird mit `global _start` nach außen bekanntgemacht.
- Wenn `_start` angesprungen wird, enthält der Stack zuoberst die Anzahl der Argumente (`argc`), dann Pointer auf die `\0`-terminierten Argument-Strings (`argv[0]` bis `argv[argc-1]`), dann einen `NULL`-Pointer, und dann Pointer auf die Environment-Variablen (Strings der Form *name*=*wert*`\0`), ebenfalls mit einem `NULL`-Pointer abgeschlossen.

  Bei dieser Programmstart-Methode enthält der Stack ***keine Return-Adresse*** (weil `_start` mit einem Jump und nicht mit einem Call angesprungen wird)!
- Alle Register außer `esp` sind mit `0` initialisiert.
- Das Programm wird durch Aufruf des Syscalls `exit` (Syscall-Nummer `1`) mit einem Argument, dem Exit-Status, beendet.

Achtung:
Wenn man Funktionen aus der C-Library verwenden will, muß auch das Hauptprogramm mit dem C-Mechanismus und nicht direkt gestartet worden sein, weil sonst sind diverse interne Datenstrukturen der C-Library nicht initialisiert!

Mit C-Mechanismen:

- Der Befehl, bei dem die Ausführung beginnen soll, bekommt das Label `main` (in der Section `.text`), es wird mit `global main` nach außen bekanntgemacht.
- `main` wird beim Programmstart entsprechend der C-Aufrufskonventionen als Funktion mit den Argumenten `argc` und `argv` am Stack aufgerufen.
- Die Register-Inhalte (außer `esp`) sind undefiniert.

- Das Programm wird mit einem `ret`-Befehl beendet, der Exit-Status sollte dabei entsprechend den C-Aufrufskonventionen als Returnwert im Register `eax` abgelegt werden.
- Alternativ dazu kann man das Programm durch den Aufruf der C-Library-Funktion (*nicht* des Systemaufrufes!) `exit` mit dem Exit-Status als Argument beenden.

Unfreiwilliges Programmende:
Ein Programm kann auch durch eine Exception beendet werden. Eine Exception wird unter anderem ausgelöst bei

- Segment Violation (Zugriff auf einen Speicherbereich außerhalb des definierten Code- und Datenbereiches, `ret` mit falscher Adresse am Stack)
- Illegal Instruction (Ausführen eines Maschinenbefehls, der entweder gar kein gültiger Befehlscode oder ein privilegierter Befehl ist)
- Alignment-Fehler (Stackpointer nicht durch 4 teilbar oder "krummer" Speicherzugriff bei gesetztem Alignment-Check-Flag)
- Stack Overflow (Endlosrekursion?)
- Illegal Syscall (Aufruf eines Syscalls mit falscher Nummer oder fehlerhaften Argumenten)
- Signal von außen (`Ctrl/C`, `kill` usw.)
- Division durch `0`, Division mit Überlauf, Gleitkomma-Fehler
- Breakpoint oder unerwartetem Software-Interrupts

7 Systemaufrufe

Systemaufrufe erlauben — ohne Verwendung der C-Library! — eine direkte Nutzung aller im Betriebssystem selbst implementierten Funktionen (siehe Beispiel 6). Dabei handelt es sich im Wesentlichen um jene Funktionen, die eine `man`-Page in `man`-Section 2 haben (`read`, `write`, ...).

Da hierbei Kernel-Code angesprochen wird (und Privilegien, Speicherverwaltung usw. auf den Kernel umgeschaltet werden müssen), werden Systemaufrufe *nicht* mit einem normalen Call-Befehl aufgerufen, sondern über einen ***Software-Interrupt***, und zwar den Interrupt Nummer `0x80` (128)[30].

Dieser eine Software-Interrupt wird für alle Systemaufrufe gemeinsam verwendet: Man übergibt eine Nummer[31], und der Kernel weiß an Hand dieser Nummer, welche Funktion er ausführen soll. Die eigentliche Adresse der jeweiligen Funktion im Kernel bleibt dem aufrufenden Programm verborgen.

Im Detail[32]:

- Die Argumente des Syscalls werden in den Registern `ebx`, `ecx`, `edx`, `esi` und `edi` gespeichert (in dieser Reihenfolge, von links nach rechts, so viele Argumente der Syscall eben hat).
- Die Nummer des Syscalls wird in Register `eax` gespeichert.

---

[30] Linux unterstützt auch noch eine zweite Aufruf-Methode, und zwar durch einen Call mit Änderung des Code-Segment-Registers.

[31] Listen mit den Syscall-Nummern von Linux findet man am Internet oder in `/usr/include/asm/unistd.h` (oder durch Lesen der Kernel-Sourcen, File `entry.S`).

[32] Es gibt zwei Ausnahmen: Erstens bekommen Syscalls mit 6 oder mehr Argumenten ihre Argumente im Speicher anstatt in den Registern übergeben (wobei in `ebx` ein Pointer auf diesen Speicherbereich übergeben wird), und zweitens haben alle Netzwerk-Syscalls eine einzige Syscall-Nummer (`102`); die auszuführende Funktion wird durch eine weitere Nummer in `ebx` angegeben, und `ecx` enthält einen Pointer auf die Argumente.

```
section .data          ; fuer String-Konstanten
msg:                   ; unser String ("write" braucht kein \0)
   db      "Hello, world!", 0x0a
len equ    $ - msg     ; dessen Laenge

section .text          ; fuer Programmcode
global _start          ; Der Startpunkt muss nach aussen sichtbar sein

_start:                ; hier geht's los
   mov     ebx, 1      ; 1. Arg: File Nummer 1 (fuer stdout)
   mov     ecx, msg    ; 2. Arg: Pointer auf String
   mov     edx, len    ; 3. Arg: dessen Laenge
   mov     eax, 4      ; Syscall-Nummer: write = 4
   int     0x80        ; write(1, msg, len)
   mov     ebx, 0      ; 1. Arg: 0 (unser Exit-Status)
   mov     eax, 1      ; Syscall-Nummer: exit = 1
   int     0x80        ; exit(0)
```

TABELLE 6: *Ein komplettes* `Hello, world!`*-Programm mit Systemaufrufen*

- Dann kommt der Befehl `int 0x80`[33].
- Der Returnwert des Syscalls wird im Register `eax` zurückgeliefert; Werte von `-4095` bis `-1` zeigen einen Fehler an.
- Die restlichen Register werden vom Syscall nicht verändert, sie bleiben erhalten.

8 C Daten und Funktionen

Namen:

- Auf aktuellen Linux-Systemen sind C-Namen und Assembler-Namen ident. Auf vielen anderen Systemen ist der Assembler-Name gleich dem C-Namen mit einem _ vorn dran.
- Ein Assembler-Programm kann auf C- und C-Library-Funktionen sowie auf globale Variablen in C-Programmen zugreifen, nicht aber auf lokale Variablen von C-Funktionen.

  Mit der Assembler-Anweisung `extern` *name* wird dem Assembler mitgeteilt, daß *name* eine Funktion oder eine Variable in einem anderen Programm ist. Danach kann *name* wie ein normales Label verwendet werden.
- Ein C-Programm kann Funktionen aus einem Assembler-Programm aufrufen und auf Variablen in der `.text`- oder `.bss`-Section eines Assembler-Programms zugreifen.

[33] `int` ist ein Maschinenbefehl, der einen Software-Interrupt auslöst, der Operand ist die Nummer des Interrupts (0-255). Für uns ist nur `int 0x80` (Syscall) und `int3` (die 1-Byte-Kurzform von `int 3`) interessant: Wenn das Programm unter der Kontrolle eines Debuggers läuft und auf `int3` stößt, wird ein Breakpoint ausgelöst.

Damit ein Label in einem Assembler-Programm für andere Programme sichtbar ist, ist die Assembler-Anweisung `global` *name* nötig (per Default sind Labels nur innerhalb des Source-Files bekannt, in dem sie definiert wurden).

Im C-Programm ist eine entsprechende `extern`-Deklaration nötig, um dem C-Compiler die Variable oder Funktion bekannt zu machen.

Datenstrukturen:

- `char`-Variablen belegen 1 Byte.
- `short`-Variablen belegen ein Wort (2 Bytes).
- `int`- und `long`-Variablen belegen ein Doppelwort (4 Bytes), `float`-Variablen und Pointer ebenfalls.
- `double`- und `long long`-Variablen belegen 8 Bytes.
- Variablen eines `enum`-Typs belegen standardmäßig 4 Bytes, es gibt aber Compiler-Optionen, um sie mit nur 1 oder 2 Bytes zu speichern.
- Arrays eines Basis-Typs und Arrays von Arrays enthalten die entsprechenden Elemente "dicht auf dicht" (ohne Zwischenräume), Arrays mit `struct`- oder `union`-Elementen enthalten möglicherweise Füllbytes zwischen den Elementen, damit jedes Element auf einer durch 2 oder 4 teilbaren Adresse beginnt.
- `struct`-Variablen enthalten der Reihe nach alle Felder sowie möglicherweise Füllbytes zwischen den Feldern, und zwar in den meisten Fällen so, daß jedes Feld auf einer durch seine Größe teilbaren Adresse zu liegen kommt.

  Enthält eine `struct` also beispielsweise ein `char`-Feld gefolgt von einem Pointer-Feld, so liegen zwischen den beiden Feldern 3 Füllbytes, und die Gesamtgröße der Struktur ist 8 statt 5.

  Herausfinden kann man das entweder mit dem C-Makro `offsetof` oder durch Pointer-Arithmetik (Adresse des fraglichen Feldes einer Strukturvariable abzüglich Adresse des ersten Feldes).
- `union`-Variablen haben die Größe des größten Feldes, alle Felder beginnen am Beginn der `union`.

Für mehrdimensionale Arrays (oder Arrays mit zusammengesetztem Element-Typ) muß man die Adressrechnung selbst machen, die Adressierungsarten reichen dafür nicht aus. Will man beispielsweise das Element `a[ebx][ecx]` eines globalen Arrays `int a[100][100]` nach `eax` laden, nimmt man dazu ein `imul eax, ebx, 400` gefolgt von einem `mov eax, [a+eax+4*ecx]`. Hat man einen Pointer auf `a` in `esi`, ist statt dem `mov` ein `lea eax, [eax+4*ecx]` und ein `mov eax, [esi+eax]` notwendig, ähnlich bei einem lokalen Array.

`nasm` bietet die Assembler-Anweisungen `struc` und `endstruc` (ohne `t` am Schluß!), um komfortabel Labels für die Offsets von Strukturfeldern definieren zu können. Eine C-Struktur `list`, die einen `int` namens `cnt`, einen `char c`, einen Pointer `next` und einen 80 Zeichen langen String `line` enthält, kann man beispielsweise wie folgt nachbilden:

```
        struc   list
.cnt    resd    1
.c      resb    1
```

```
        alignb  4         ; Pointer auf Doppelwort-Grenze ausrichten!
.next   resd    1
.line   resb    80
        endstruc
```

Wenn `esi` dann auf eine solche Struktur zeigt, läßt sich der Inhalt des Feldes `next` wie folgt laden:

```
        mov     ebx, [esi+list.next]
```

Neben den Labels für die Felder definiert `struc` auch noch automatisch ein Label für die Gesamtgröße der Struktur in Bytes, im Beispiel oben `list_size`.

Call und Return, Argumente:

Unter Linux ist folgende Parameter-Übergabe standardisiert:

- Alle Argumente werden am *Stack* übergeben[34].

  Nachdem `esp` im 32-Bit-Mode immer ein Vielfaches von 4 sein muß, werden alle Datentypen als *Doppelwort* übergeben (oder als Vielfaches von 4 Bytes bei größeren Datentypen): `char` und `short` werden in `int` verwandelt!

  Auch `struct`-Variablen können im Prinzip am Stack auf diese Weise "by value" übergeben werden, es ist aber üblich (und normalerweise auch sinnvoller), in diesen Fällen das Argument stattdessen als Pointer auf die Struktur zu definieren!

- Der Aufrufer legt die Argumente ***von rechts nach links*** mittels `push` am Stack ab (das hinterste Argument wird zuerst gepusht, das vorderste als Letztes).

- Dann ruft er mittels `call`-Befehl die gewünschte Funktion auf.

- Wenn die Ausführung der aufgerufenen Funktion beginnt, liegt daher auf `[esp]` die Return-Adresse, auf `[esp+4]` das erste (linkeste) Argument, auf `[esp+8]` das zweite usw..

- Die Anzahl der Argumente wird *nicht* übergeben oder markiert.

  Die aufgerufene Funktion hat daher keine Möglichkeit, festzustellen, wie viele Argumente wirklich übergeben worden sind[35]: Hat der Aufrufer zu wenig Argumente am Stack abgelegt, verwendet der Aufgerufene einfach die zufällig dahinter am Stack liegenden Werte als Argumente (was in vielen Fällen zu einem Absturz führt). Hat der Aufrufer hingegen zu viele Argumente übergeben, ist das harmlos: Diese werden einfach ignoriert.

- Der Aufgerufene kann die übergebenen Argumente beispielsweise mit `[esp+4]` ansprechen, aber nicht mit `pop`: Er darf den Wert der Argumente zwar lesen und auch ändern, aber er darf sie nicht vom Stack entfernen!

---

[34] Bei einer anderen, vor allem in der Microsoft-Welt üblichen Konvention ("`regparm`") werden die ersten drei Argumente in Registern anstatt auf dem Stack übergeben.

[35] Das ist der Grund, warum die Argumente in verkehrter Reihenfolge übergeben werden:

Normalerweise läßt sich bei Funktionen mit variabel vielen Argumenten auf Grund des Wertes des ersten oder zweiten Argumentes feststellen, wie viele Argumente übergeben worden sein müßten, bei `printf` ergibt sich die Argumentanzahl beispielsweise aus der Anzahl der Formatbefehle im Formatstring.

Durch die Push-Reihenfolge steht das erste Argument nach dem `call` aber immer an Adresse `[esp+4]`, das zweite auf `[esp+8]` usw., ganz egal, wie viele Argumente übergeben wurden. Die aufgerufene Funktion kann daher von links nach rechts auf die Argumente zugreifen, *ohne* zu wissen, wie viele es eigentlich sind.

- Der Aufgerufene endet mit einem normalen `ret`-Befehl (ohne Operanden).
- Der *Aufrufer* ist nach dem `call` dafür verantwortlich, die von ihm gepushten Argumente wieder vom Stack zu entfernen[36]. Dazu wird kein `pop` verwendet (weil es unnötiger Aufwand wäre, die Argumente vom Stack zu lesen), sondern einfach eine Korrektur des Stackpointers: `add esp, byte 4` bei einem Argument, `8` bei zwei usw..

Returnwert:

- Im Fall von `char`, `short`, `int`, `long` usw. sowie bei Pointern und Enumerations wird der Returnwert im Register `eax` zurückgegeben.

  Dabei muß immer das gesamte `eax`-Register gesetzt sein: Bei 1 oder 2 Bytes großen Datentypen (`char` und `short`) reicht es also nicht, den Returnwert in `al` oder `ax` zu speichern, man muß die hochwertigen Bytes auf `0` oder die Sign-Extension des jeweiligen Wertes setzen!
- Im Fall von `long long` wird `eax` und `edx` verwendet.
- `float` und `double` werden in einem Gleitkomma-Register zurückgegeben.
- Für `struct` und `union` (als Wert, nicht als Pointer) gibt es je nach C-Compiler unterschiedliche Rückgabe-Konventionen, man sollte diese Fälle aber ohnehin vermeiden und stattdessen mit Pointern auf `struct` oder `union` arbeiten!
- Arrays werden nie als Wert zurückgegeben, sondern immer nur als Pointer.

Basepointer:

Das Konzept, alle Daten am Stack über den Stackpointer `esp` anzusprechen, hat einige Schwächen:

- Sobald der aufgerufene Code `push`- und `pop`-Befehle ausführt, ändern sich die Offsets der Argumente: Nach zwei `push` liegt das erste Argument nicht mehr auf `[esp+4]`, sondern auf `[esp+12]`.

  Bei längeren Funktionen ist es kaum möglich, einen Überblick über die korrekten `esp`-Offsets zu bewahren, und bei Code-Änderungen müssen mitunter jede Menge Offsets angepaßt werden.
- Variablen in `.data` oder `.bss` eignen sich nicht für rekursive Funktionen: Bei einer rekursiven Funktion braucht jeder einzelne Aufruf seine eigenen lokalen Variablen (weil er ja sonst die des umgebenden Aufrufs überschreibt), diese müssen daher bei jedem Aufruf irgendwo am Stack — und nicht statisch! —angelegt werden.
- Auch ein Debugger hat Probleme, den Stackinhalt richtig zuzuordnen. Er hat vor allem keine Chance, den Stackinhalt der umgebenden Funktionsaufrufe zu finden.

Um diese Probleme zu lösen, wird das Register `ebp` als ***Basepointer*** verwendet, und zwar nach folgender Konvention, die auch "Anlegen eines Stack-Frames" genannt wird:

---

[36] Nur der Aufrufer weiß mit Sicherheit, wie viele Argumente er gepusht hat. Bei Programmiersprachen, die keine variable Anzahl von Argumenten zulassen (z. B. PASCAL) ist normalerweise der *Aufgerufene* dafür verantwortlich, den Stack zu bereinigen.

Es gibt dafür einen eigenen `ret`-Befehl mit einer Zahl als Operand: Er führt ein normales `ret` aus und korrigiert `esp` danach um den angegebenen Wert.

- Jede Funktion muß sofort nach dem Aufruf den alten Wert von `ebp` (den Basepointer der aufrufenden Funktion) mit einem `push ebp` auf den Stack legen.
- Danach wird der aktuelle Wert des Stackpointers `esp` als neuer Basepointer `ebp` gespeichert: `mov ebp, esp`
- Schließlich wird am Stack Platz für die lokalen Variablen reserviert, indem der Stackpointer um die Anzahl der Bytes, die für die lokalen Variablen benötigt werden, heruntergezählt wird[37]: `sub esp, byte` n (wobei n immer ein Vielfaches von 4 sein muß!)

  Gibt es keine lokalen Variablen, kann man diesen Schritt weglassen.
- Für diese drei Schritte gibt es einen eigenen Maschinenbefehl: Durch `enter` n`, 0` wird `ebp` gesichert und neu gesetzt, und es werden n Bytes Platz am Stack reserviert[38].
- `ebp` bleibt dann bis zum Ende der Funktion unverändert; alle Parameter und lokalen Variablen werden mit fixen Offsets relativ zu `ebp` (anstatt relativ zu `esp`) angesprochen[39]: Die Parameter mit `[ebp+8]`, `[ebp+12]` usw., die lokalen Variablen mit `[ebp-4]`, `[ebp-8]` etc..

  Dementsprechend kann mit `lea` r`,[ebp-4]` ein *Pointer* auf die erste lokale Variable in Register r geladen werden.
- Am Ende der Funktion (vor dem `ret`) wird mit `mov esp, ebp` der lokale Speicherplatz wieder freigegeben (falls einer reserviert wurde).
- Dann folgt ein `pop ebp`, um den Framepointer der aufrufenden Funktion wiederherzustellen.
- Diese beiden Schritte lassen sich auch gemeinsam durch den Maschinenbefehl `leave` erledigen.

Da `ebp` stets auf den alten Wert von `ebp` (d. h. den Basepointer der vorigen Funktion) am Stack zeigt, dieser wiederum auf den Basepointer der vorvorigen Funktion usw., kann ein Debugger mühelos alle Stack-Frames bis zurück zur `main`-Funktion ermitteln und von jeder Funktion die Argumente und die lokalen Variablen anzeigen.

Register:

- Die Register `eax`, `ecx` und `edx` dürfen von jeder Funktion beliebig verändert werden:
  - Falls der Aufrufer in diesen Registern Dinge gespeichert hat, die er nach dem Aufruf noch braucht, muß er diese Register vor dem `call` speichern und nachher wieder laden.
    Dazu sind lokale Variablen oder `push` (*vor* den Argumenten!) und `pop` geeignet.
  - Der Aufgerufene darf diese Register beliebig verwenden, ohne den Inhalt, den sie beim Aufruf hatten, vorher zu sichern und nachher wiederherzustellen.

---

[37] Auch C implementiert seine Variablen nach diesen Regeln: Alle lokalen Variablen werden beim Aufruf am Stack angelegt und bei der Rückkehr wieder freigegeben. Nur globale Variablen und `static`-Variablen werden — je nachdem, ob ein Initialisierungswert angegeben ist oder nicht — in `.data` oder `.bss` abgelegt.

[38] Die Anzahl n der für lokale Variablen zu reservierenden Bytes ist bei `enter` als 2-Byte-Zahl codiert. Reicht das nicht (z. B. bei großen lokalen Arrays), muß man statt `enter` die Einzelbefehle verwenden.

[39] Auch die Codierung der Befehle ist darauf ausgerichtet: Eine Adressierung relativ zu `esp` resultiert in einem um 1 Byte längeren Maschinenbefehl als eine relativ zu `ebp`.

```
section .data          ; fuer String-Konstanten
msg:                   ; unser String, ohne \n und mit \0 fuer puts
   db      "Hello, world!", 0

section .text          ; fuer Programmcode
global main            ; Der Startpunkt muss nach aussen sichtbar sein
extern puts            ; diese Funktion aus der Clib verwenden wir

main:                  ; hier geht's los
   push    dword msg   ; Argument auf den Stack legen: Adr des Strings
   call    puts        ; die Funktion aufrufen
   add     esp, byte 4; Arg vom Stack nehmen
   xor     eax, eax    ; Returnwert = Exit Status = eax = 0
   ret                 ; das war's
```

TABELLE 7: *Ein komplettes* `Hello, world!`*-Programm mit C-Funktionen*

- Der Inhalt der Register `ebx`, `esi` und `edi` sowie `ebp` muß bei einem Funktionsaufruf *erhalten* bleiben:
  - Der Aufrufer darf sich darauf verlassen, daß der Inhalt dieser Register nach dem `call` unverändert erhalten ist.
  - Der Aufgerufene muß diese Register sichern (üblicherweise mittels `push` unmittelbar nach dem `enter`), wenn er sie verwenden will, und vor der Rückkehr wiederherstellen (mit `pop` in umgekehrter Reihenfolge unmittelbar vor dem `leave`).

Beispiele:

Beispiel 7 ist ein `Hello, world!`-Programm unter Verwendung von `puts`.

Als weiteres einfaches Beispiel zeigt Programm 8 eine Funktion `bswap_func`, die ein 4 Bytes langes Argument erwartet und dessen Bytes mit dem Befehl `bswap` in die umgekehrte Reihenfolge bringt[40], sowie deren Aufruf `myvar=bswap_func(myvar)` für eine globale Variable `myvar`. Nachdem die Funktion ganz einfach ist, keine lokalen Variablen braucht, und hoffentlich auch nicht debugged werden muß, verzichten wir auf das Einrichten eines Stack-Frames.

Jetzt zu einem umfangreicheren Fall (Programm 9):
Es soll ein Funktionsaufruf `count=myfunc(str, gstr, i, max, 3)` ausgeführt werden, wobei `count` die erste lokale Variable ist, `max` als erstes Argument an die aufrufende Funktion übergeben wurde, `i` im Register `edx` liegt, `gstr` das Label einer globalen Stringkonstante ist, und `str` einen Pointer auf ein lokal angelegtes Array mit 80 Bytes darstellt, das nach `count` liegt. `myfunc` soll ein Stackframe mit Platz für drei lokale Doppelwort-Variablen anlegen. Der aufrufende Code hat in allen Registern außer `eax` irgendwelche nachher noch benötigten Werte gespeichert, und auch `myfunc` benutzt alle Register. Tabelle 10 zeigt den Ablauf am Stack.

Verwendet ein Programm nur eigene Funktionen, muß man sich natürlich nicht an diese Konventionen halten (obwohl es sinnvoll ist), sondern kann eigene Regeln definieren, z. B. die Argumente in

[40] Das ist beispielsweise bei Zahlen notwendig, die über TCP/IP gesendet oder empfangen werden: Die TCP/IP-Protokolle legen fest, daß Mehr-Byte-Zahlen "Big-Endian" verschickt werden, also höchstwertiges Byte zuerst — für einen $x86$-Computer genau "verkehrtherum".

```
bswap_func:                  ; hier geht's los
; keine Initialisierung notwendig, keine Register zu retten:
; es geht gleich zur Sache
   mov    eax, [esp+4]       ; Argument laden
   bswap  eax                ; Bytes umordnen
; auch kein Aufraeumen notwendig
   ret                       ; Rueckkehr zum Aufrufer mit Ergebnis in eax

...
   push   dword [myvar]      ; Das Argument auf den Stack legen
   call   bswap_func         ; Funktion aufrufen
   add    esp, byte 4        ; Das Argument vom Stack entfernen
   mov    [myvar], eax       ; Den Returnwert speichern
...
```

TABELLE 8: *Funktion* `bswap_func` *mit Aufruf*

Registern übergeben oder den Aufgerufenen statt dem Aufrufer die Argumente vom Stack entfernen lassen (beides bringt etwas Performance).

9 Makros

Konzept:

`nasm` enthält so wie die meisten Assembler einen dem C-Preprozessor vergleichbaren Preprozessor, der das Assembler-Programm in einem eigenen Schritt *vor* dem eigentlichen Assembler-Lauf umformt[41].

Dieser Preprozessor unterstützt unter anderem:

- Verschiedene Formen von Makros (mit und ohne Parametern).
- Das Inkludieren von Files.
- Bedingte Codestücke.
- Eigene Preprozessor-Variablen und String-Funktionen sowie einen Stack für geschachtelte Makro-Konstrukte.

Hier werden nur seine wichtigsten Fähigkeiten besprochen[42].

File Inclusion:

`%include "`*file*`"` wird durch den Inhalt des angegebenen Files ersetzt. Man kann sich beispielsweise einen File `mymacs.inc`[43] mit einer Sammlung von Standard-Makros und -Konstanten

[41] Die mächtigen Makro-Fähigkeiten professioneller Assembler waren der Hauptgrund, warum man überhaupt jahrzehntelang große Softwarepakete erfolgreich in Assembler programmieren konnte; C hat sich das von Assembler abgeschaut und nicht umgekehrt!

[42] Die Makro-Fähigkeiten von Assemblern sind leider nicht standardisiert: Jeder Assembler bietet bezüglich Funktion und Syntax unterschiedliche Makro-Features.

[43] Die Extension kann nach Belieben gewählt werden, `.inc` ist sinnvoll.

```
myfunc:                         ; hier geht's los
   push   ebp                   ; alten Basepointer retten
   mov    ebp, esp              ; neuen Basepointer einrichten
   sub    esp, byte 12          ; Platz fuer 3*4 Bytes reservieren
;  enter  12, 0                 ; das waere das gleiche gewesen
   push   ebx                   ; ebx, esi, edi muessen erhalten bleiben
   push   esi
   push   edi
; hier kommt der eigentliche Funktionscode
; der Returnwert muss in eax abgelegt werden
   pop    edi                   ; alte Registerwerte wiederherstellen
   pop    esi                   ; (in umgekehrter Reihenfolge!)
   pop    ebx
   mov    esp, ebp              ; Platz fuer lokale Variablen wieder freigeben
   pop    ebp                   ; alten Basepointer wiederherstellen
;  leave                        ; das ginge auch statt dem mov und pop
   ret                          ; Rueckkehr zum Aufrufer

...
   push   ecx                   ; ecx und edx werden ueberschrieben:
   push   edx                   ; vor Funktionsaufruf retten!
   push   dword 3               ; letztes Argument "3" (mit 4 Bytes!) zuerst
   push   dword [ebp+8]         ; dann vorletztes: "max" ist 1. Parameter
   push   edx                   ; "i" liegt in edx
   push   dword gstr            ; Pointer auf "gstr" = Wert des Labels
   lea    eax, [ebp-84]         ; Pointer auf lokales Array berechnen
   push   eax                   ; ... und als oberstes Argument pushen
   call   myfunc                ; Jetzt der Aufruf
   add    esp, byte 20          ; die 5 Argumente vom Stack entfernen
   pop    edx                   ; die geretteten Register wiederherstellen
   pop    ecx                   ; (in umgekehrter Reihenfolge!)
   mov    [ebp-4], eax          ; Returnwert in "count" speichern
...
```

TABELLE 9: *Eine komplexere Funktion* `myfunc` *mit Aufruf*

schreiben und ihn mit `%include "mymacs.inc"` in jedem Assembler-Programm inkludieren[44].

Conditional Assembly:
Ähnlich wie in C gibt es die Preprozessor-Befehle `%ifdef` *name* oder `%ifndef` *name* oder `%elifdef` *name* oder `%elifndef` *name* oder `%else` oder `%endif`: Der Code zwischen diesen Konstrukten wird nur dann assembliert, wenn ein Makro mit dem Namen *name* definiert bzw. nicht definiert ist.

Typisches Beispiel:

[44] Der in C übliche Trick mit `%ifndef` gegen mehrfaches `%include` desselben Files ist auch in Assembler zu empfehlen!

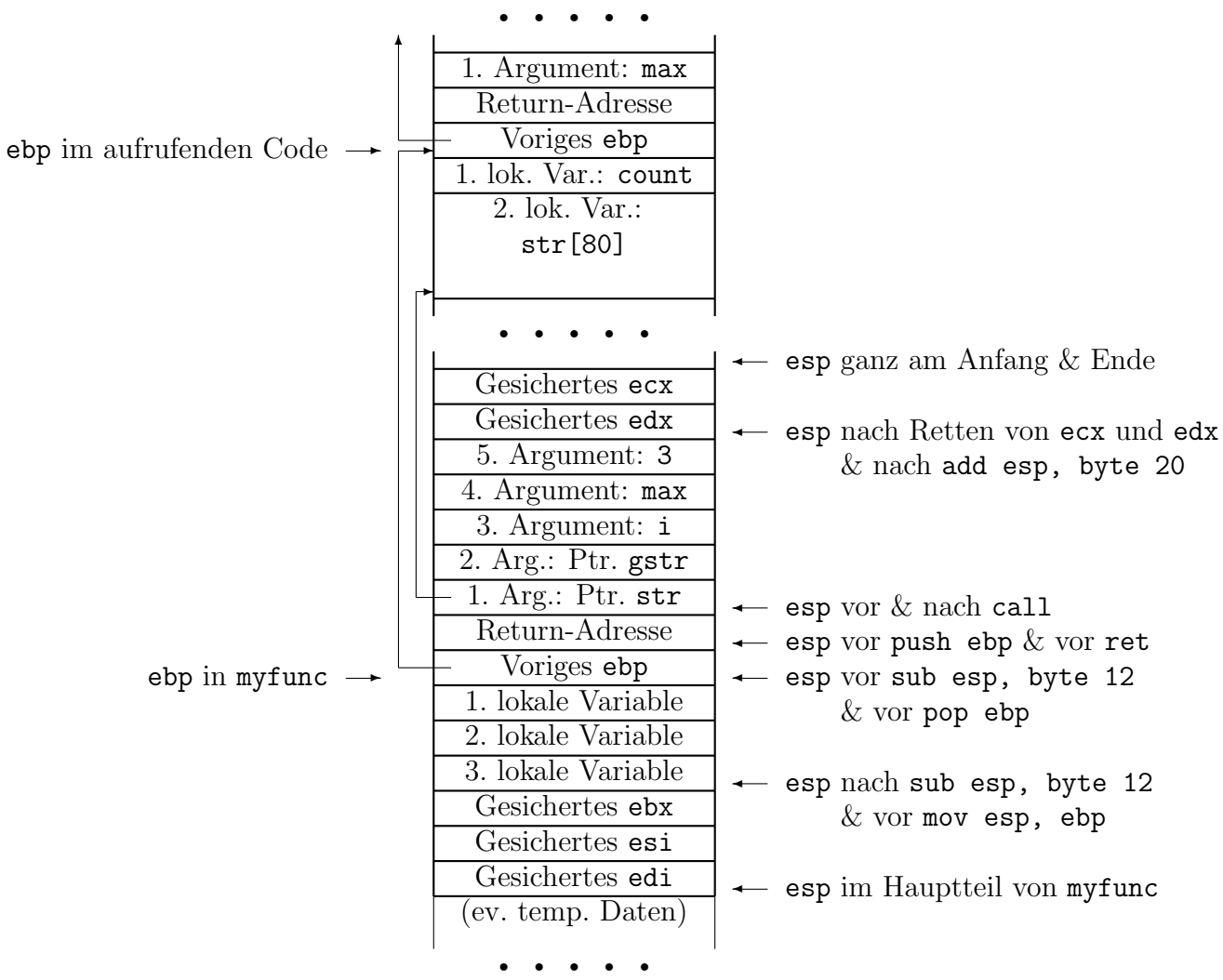


TABELLE 10: *Der Stack bei Ausführung von Programm 9*

```
%ifdef DEBUG
; Code hier wird nur mitassembliert, wenn das Makro DEBUG definiert ist
%endif
```

Makros kann man auch beim `nasm`-Aufruf auf der Commandline definieren, für das Beispiel oben mit `-dDEBUG`.

`%define`-Makros:

`%define` *macro definition* ist wie in C für "kurze" Makros (innerhalb einer Zeile) zuständig. Man kann es beispielsweise verwenden, um Namen für Parameter und lokale Variablen zu vergeben: Nach `%define i [ebp+12]` kann man `mov eax, i` schreiben.

Auch Parameter sind möglich (wie in C in `( )`): Für den Zugriff auf ein lokales Array eignet sich beispielsweise `%define a(ireg) [ebp-96+4*(ireg)]`, der Aufruf erfolgt beispielsweise mit `mov ecx, a(eax)`.

Ob man für numerische Konstanten `equ` oder `%define` verwendet, ist Geschmackssache, `%define newline 0x0a` (oder `%define newline byte 0x0a`, wenn man es immer als Byte-Konstante in Befehlen verwendet) hat jedenfalls den gewünschten Effekt.

Mit `%undef` *name* wird das Makro wieder "vergessen".

```
%macro funbeg 1
   enter  %1, 0
   push   ebx
   push   esi
   push   edi
%endmacro

%macro funend 0
   pop    edi
   pop    esi
   pop    ebx
   leave
   ret
%endmacro

myfun:
   funbeg 20      ; 20 Bytes lokale Variablen reservieren
; Code der Funktion
   funend
```

TABELLE 11: *Makros für Funktionsaufrufe*

%macro-Makros:

`%macro` *name param_cnt* ist für "lange" Makros (ein oder mehrere ganze Befehle) zuständig: Das Makro umfaßt alle folgenden Zeilen bis `%endmacro`. Der Name des Makros wird dann beim Aufruf *wie ein Befehl* verwendet: Im Unterschied zu `%define` werden die Argumente beim Aufruf daher auch nicht in `( )` angegeben, sondern ganz normal wie die Operanden eines Befehls. *param_cnt* ist die Anzahl der Parameter, die das Makro erwartet, die einzelnen Parameter werden im Rumpf des Makros mit `%1`, `%2` usw. angesprochen.

Einfachstes Beispiel:

```
%macro syscall 0
   int 0x80
%endmacro


...
   syscall
...
```

Ein weiteres praktisches Beispiel zeigt Programm 11.

Oft braucht man in Makros auch Labels. Damit mehrfache Aufrufe des Makros nicht zu Fehlermeldungen wegen doppelt definierter Label führen, muß man den Assembler dazu bringen, bei jedem Aufruf "neue" Labels anzulegen. Das erreicht man, indem man in der Makro-Definition `%%` vor jedes Label schreibt.

Weiters kommt es vor, daß ein Makro mit einem Argument aufgerufen werden soll, das Beistriche enthält (siehe Programm 12). Dafür gibt es zwei Möglichkeiten:

- Man schließt das Argument beim Aufruf in `{ }` ein.

```
; ein Makro mit 1 oder mehr Argumenten, alle in %1:
; gibt die angegebenen Bytes / Strings aus
%macro wrtstr 1+
; String im Speicher anlegen
section .data
%%str   db      %1
%%len   equ     $-%%str
; und via Syscall 'write' ausgeben
section .text
        mov     edx, %%len
        mov     ecx, %%str
        mov     ebx, 1
        mov     eax, 4
        int     0x80
%endmacro

...
        wrtstr  'hallo', 0x20
        wrtstr  'world', '!', 0x0a
...
```

TABELLE 12: *Makro* `wrtstr`

- Man definiert das Makro mit n`+` Argumenten, dann werden alle verbleibenden Argumente im letzten Parameter `%`n gespeichert.

Und noch ein paar Makro-Tricks zum Abschluß:

- Text zwischen `%rep` n und `%endrep` wird n mal wiederholt.
- Mit x-y als Argumentzahl in `%macro` kann man Makros mit variabel vielen Argumenten definieren, `*` steht dabei für beliebig viele[45].
- Die Anzahl der Argumente steht wie bei der Shell in `%0`.
- Mit `%rotate` n können die Parameter rotiert werden: Jeder Parameter rutscht um n nach vor, die vordersten n Parameter werden die letzten (bei negativem n rutschen die Parameter nach hinten, und die hintersten werden die ersten).

Damit ergeben sich die Makros in Programm 13, die beliebig viele Register pushen und in umgekehrter Reihenfolge (!) wieder poppen.

10 Wie programmiert man in Assembler?

1. Zuerst den Algorithmus konventionell niederschreiben (als C- oder PASCAL-Code, als Struktogramm, oder umgangssprachlich)!

2. Überlegen, welche Daten welchen Typ haben.

[45] Diese werden *nicht* zu einem einzigen Parameter zusammengefaßt, sondern in entsprechend vielen einzelnen Parametern gespeichert.

```
%macro pushx 1-*
%rep %0
   push %1
%rotate 1
%endrep
%endmacro

%macro popx 1-*
%rep %0
%rotate -1
   pop %1
%endrep
%endmacro

...
   pushx eax, ebx, ecx, edx
...
   popx eax, ebx, ecx, edx
...
```

TABELLE 13: *Makros für* `push` *und* `pop` *mehrerer Register*

---

3. Überlegen, welche Daten wohin kommen:
   - Was sind globale Daten (Namen vergeben und notieren!)?
   - Was sind Argumente oder lokale Daten (Basepointer-Offset notieren!)?
   - Welche Daten können in welchen Registern gehalten werden, ohne dafür Speicher anzulegen (Registername notieren!)? Achtung auf Befehle, die fixe Register erfordern (z. B. Multiplikation und Division oder Stringbefehle)!
   - Welche Daten werden temporär (mit `push` und `pop`) am Stack abgelegt?
4. Den Code nach Konstrukten absuchen, die sich durch spezielle Maschinenbefehle (String- oder Bit-Befehle, Pointer-Jumps für `switch`-Statements, usw.) realisieren lassen, und ev. entsprechend umbauen.
5. Ablauf-Struktur "Assembler-tauglich" machen: `for`-Schleifen in `while`- oder `until`-Schleifen verwandeln, `switch`-Statements eventuell durch `if`'s ersetzen[46], logische Ausdrücke (`&&` und `||`) in geschachtelte `if`'s auflösen, mehrbeinige `if`'s entsprechend anordnen, ...

   Am besten: Ein *Flußdiagramm* zeichnen (das ist zwar altmodisch und unstrukturiert, aber für Assembler viel besser geeignet als ein Struktogramm!), und zwar so, daß alles der Reihe nach untereinander und nicht nebeneinander steht!
6. Das Flußdiagramm (bzw. das umgeformte C-Programm oder Struktogramm) mit Labels ergänzen: Jedes sequentielle Stück Code bekommt vorbeugend ein Label!
7. Dann erst: In Assembler codieren!

---

[46] Üblicherweise werden `switch`-Statements durch ein Array von Code-Pointern und einen indirekten `jmp`-Befehl implementiert.