

Teil 1:

Hochsprache / Assembler-Code

Hochsprache / Assembler-Code

Hochsprache (z.B. C, C++, Java, Pascal, ...) nicht direkt ausführbar

*Übersetzung der Hochsprache auf die „Muttersprache“ (Maschinensprache) des Rechners
(symbolische Maschinensprache: Assembler-Code)*

Transformation normalerweise Aufgabe eines Compilers

Assembler-Code-Programmierung in Ausnahmefällen:

- Ausnutzen von Spezialfällen, für die die CPU bessere Befehle kennt
- Compiler erzeugt schlechten / falschen Code
- Programmteile, die in Hochsprache nicht programmierbar sind; z.B.
 - I/O-Operationen
 - Interrupt- / Exception- / System-Call-Handler

Hochsprache / Assembler-Code

Beispiel 1:

```
short int x;  
  
for (x = SHRT_MIN; x <= SHRT_MAX; x = x + 1) {  
    ...  
}
```

Hinweis vom Compiler (hier: gcc):

comparison is always true due to limited range of data type


*Das Programm durchläuft nicht –wie wohl erhofft– alle zulässigen Werte für x genau einmal!
Statt dessen ist obige Schleife eine Endlos-Schleife!*

Warum???

Hochsprache / Assembler-Code

Beispiel 2:

```
double array[10000][10000];  
double sum;  
int x; int y;  
  
sum = 0.0;  
for (x = 0; x < 10000; x++) {  
    for (y = 0; y < 10000; y++) {  
        sum = sum + array[y][x];  
    }  
}
```



Vertauscht man beim Array-Zugriff x und y ist das Programm u.U. um Größenordnungen schneller!

Warum???

Hochsprache / Assembler-Code

Beispiel 3:

```
int mean(int x, int y)
{
    return (x + y) / 2;
}
```

Diese Funktion liefert als „Mittelwert“ von 2.000.000.000 und 2.000.000.000 (beides korrekte 32-Bit-int-Zahlen) den Wert -147483648 zurück.

Warum?!?

Hochsprache / Assembler-Code

Beispiel 4:

```
struct element {
    char code[N];
    int val;
};
```

Für $N = 4$ braucht dieses Struktur / Klasse 8 Byte Speicher. Für $N = 5$ erhöht sich der Bedarf auf 12 Byte. Bei weiterer Vergrößerung von N auf z.B. $N = 6$, $N = 7$ oder $N = 8$ wird dagegen nicht nochmals mehr Speicher gebraucht.

Warum?!?

Hochsprache / Assembler-Code

Assembler-Code-Wissen unverzichtbar für

- Programmierung effizienter Algorithmen (auch in Hochsprachen!)
- Programmierung System-naher Programmteile (auch in Hochsprachen!)
- Hardware-Entwurf / -Bewertung
- Compiler-Bau, Programmiersprachen-Entwurf
- Betriebssystem-Programmierung / -Design
- Realzeit- / Embedded-System-Programmierung

Wissen hilft bei (Beispiele)

- Grafik-Programmierung
- Programmierung für Mustererkennung
- Datenbank-Programmierung

Hochsprache / Assembler-Code

Maschine soll in der Lage sein,

- alle in beliebigen (problemorientierten) Hochsprachen geschriebene Programme
 - effizient (schnell, geringer Speicheraufwand)
- abzuarbeiten

InstruktionsSatz-Architektur (ISA):

Ziel der CPU-Architekten sollte sein:

- grundlegende Befehle finden
- Balance zwischen
 - viele Befehle => aufwändige Dekodierung
 - wenige Befehle => ineffiziente Berechnung

Hochsprache / Assembler-Code

Konstrukte von Hochsprachen:

- Datenzugriffe
- (I/O-Operationen)
- Arithmetik
- Kontrollstrukturen
- Unterprogramme
- (Interrupts, Exceptions, System-Calls)

Zerlegung von Datenzugriffen, arithmetischen Ausdrücken, Schleifen-, If-Then-Else-, Case-Anweisungen usw. in einfache Bestandteile

Hochsprache / Assembler-Code

normalerweise Beispiele für Intel-Prozessoren der i80x86-Serie (ab i80386)

- in GNU-Assembler-Schreibweise
- geeignet für eigene Versuche auf Linux-PCs (Windows-PCs)
- viele Infos unter <http://linuxassembly.org/>

zum Teil Beispiele für Motorola-Mikro-Controller m68x05

- in Motorola-Schreibweise

Hinweise für eigene Versuche mit GNU-Assembler/Debugger unter Linux

Grundgerüst: Datei file.S

```
.text                /* Programmsegment */
_start:.globl _start
...                 /* <--- Testprogramm */
movl $0, %ebx       /* exit-Status des Programms */
movl $1, %eax       /* exit-Funktionsnummer */
int $0x80           /* System-Call */

.data               /* Datensegment */
...                 /* <--- Testdaten */
```

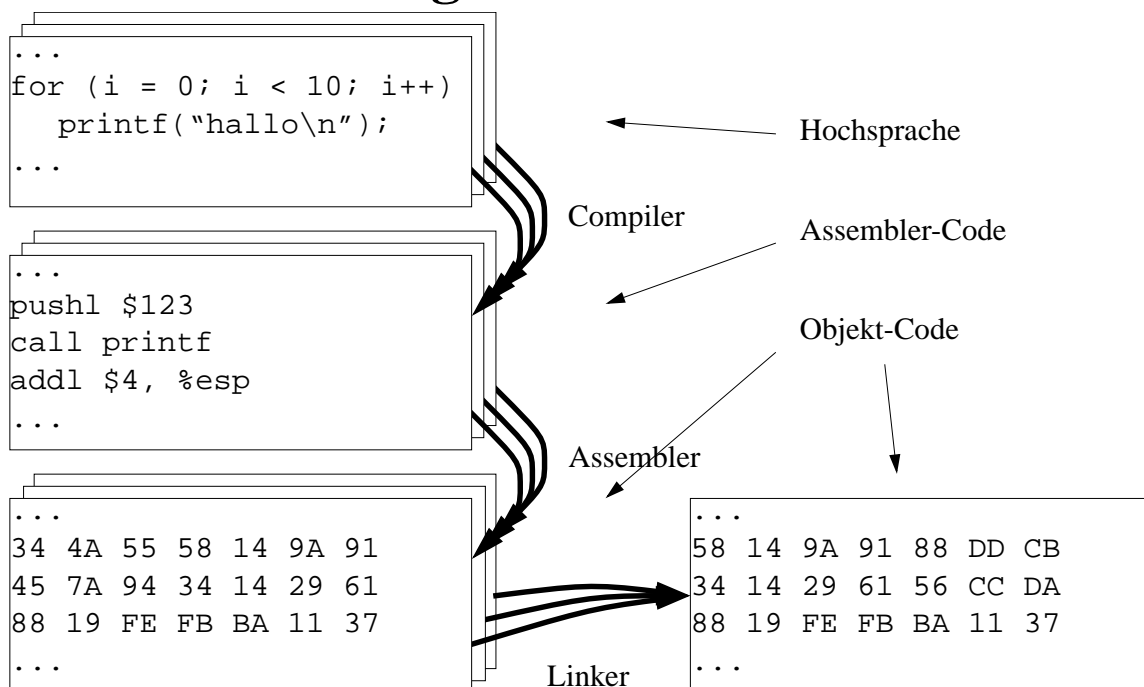
Assemblieren mit

```
as --gstabs -o file.o file.S
```

Linken mit

```
ld -o file file.o
```

Hinweise für eigene Versuche unter Linux



Hinweise für eigene Versuche mit GNU-Assembler/Debugger unter Linux

Testen des Programms `file` mit

- `./file` (normaler Testlauf)
- `gdb ./file` oder `ddd ./file` (Lauf mit Debugger)

Hinweise für eigene Versuche mit GNU-Assembler/Debugger unter Linux

Debugger (`gdb`) stellt mehrere Befehle zum Testen zur Verfügung:

- `help` (allgemeine Hilfe)
- `break func` (Break Point setzen)
- `run` (Lauf starten)
- `stepi` (Ausführung einer einzelnen Maschinen-Instruktion)
- `info register` (Anzeige der CPU-Register)
- `x &var` (Anzeige der Variablen `var`)
- `disassemble func` (Anzeige der Funktion `func`)
- `quit` (Beenden des Debuggers)

**Jeder(!) sollte eigene(!)
Versuche durchführen!**

**Verständnis ohne Programmierung
praktisch unmöglich!**

- C-Code schreiben – vom C-Compiler erzeugten Code verstehen
(oder andere problemorientierte Hochsprache)
- Assembler-Code schreiben – Programm testen