Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Systemprogrammierung: Assembler, Linker, Lader

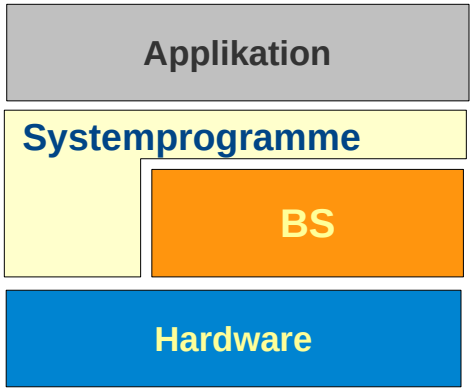
- Systemprogrammierung
- Von Quellcode zum aktiven Prozess: Compiler, Assembler, Linker, Lader
- Einige Grundbegriffe der Rechnerarchitektur

Systemprogrammierung – Definition

Systemprogrammierung / Systemprogramme

Als Systemprogrammierung bezeichnet man das Erstellen von Softwarekomponenten, die Teil des Betriebssystems sind oder die eng mit dem Betriebssystem bzw. mit der darunter liegenden Hardware kommunizieren müssen. (Wikipedia)

Systemprogramme: Hilfsprogramme die nicht selbst Anwendungen sind, sondern bei der Erstellung und dem Betrieb von Anwendungen helfen



Systemprogrammierung – Themen

- Umgang mit der Betriebssystem-Schnittstelle
- Hardware-nahe Programmierung
- System-Software
  - Assembler
  - Binder
  - Lader
  - Compiler
  - Betriebssystem

Systemprogrammierung und Compilerbau

Compilerbau

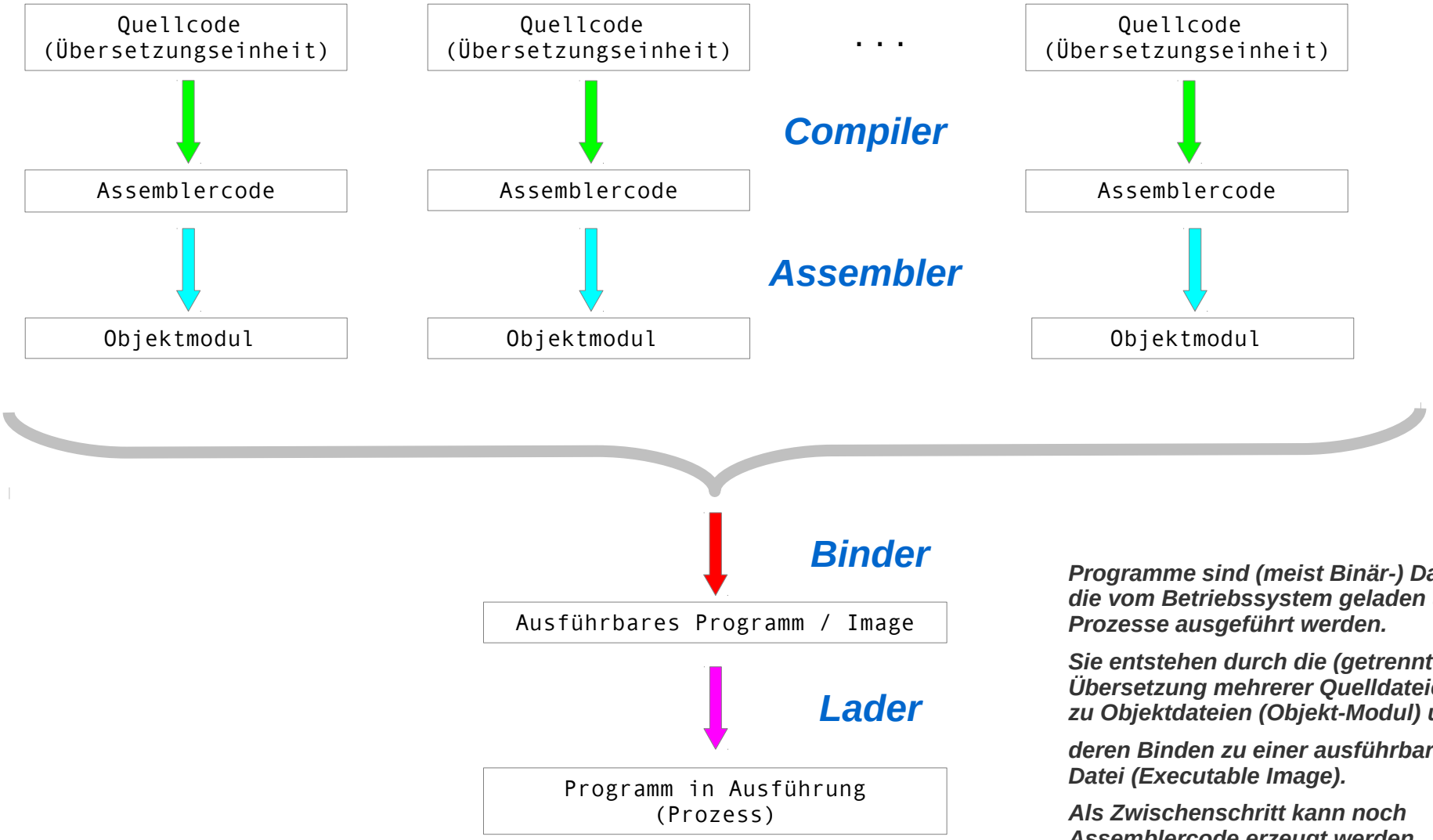
Compiler sind Systemprogramme – Compilerbau ist (auch) ein Teilgebiet der Systemprogrammierung

Compiler haben enge Bezüge zu anderen Themen und Werkzeugen der Systemprogrammierung:

- **Assembler**
- **Binder (Linker)**
- **Lader**
- **Virtuelle Maschinen**
- **…**

Vom Quellprogramm zum laufenden Code

Übersicht



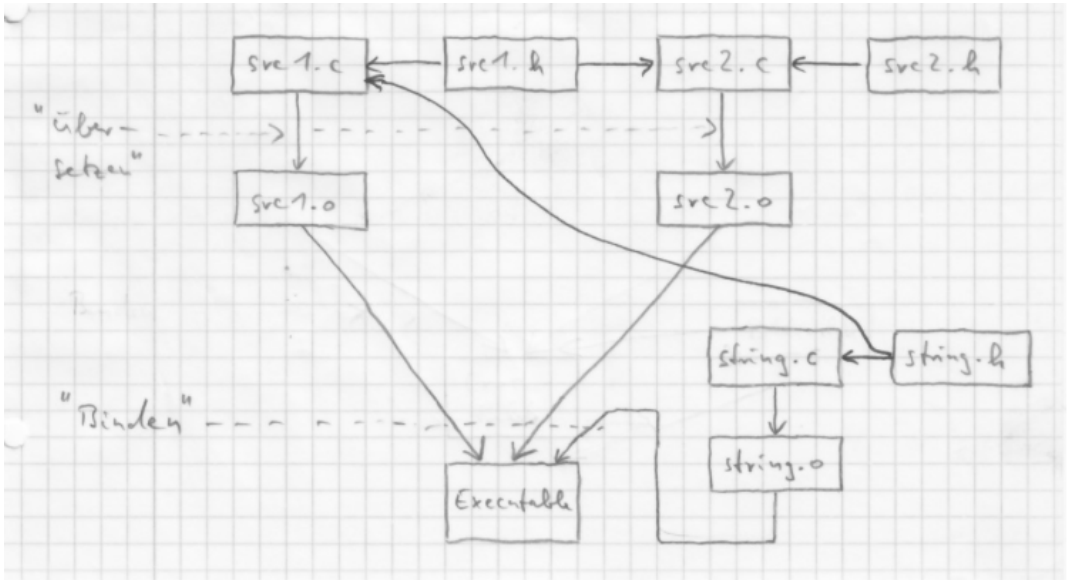
Programme sind (meist Binär-) Dateien, die vom Betriebssystem geladen und als Prozesse ausgeführt werden.

Sie entstehen durch die (getrennte) Übersetzung mehrerer Quelldateien und zu Objektdateien (Objekt-Modul) und

deren Binden zu einer ausführbaren Datei (Executable Image).

Als Zwischenschritt kann noch Assemblercode erzeugt werden.

Übersicht



Ausschnitt aus KSP-Skript zum Thema Übersetzen und Binden bei C-Programmen

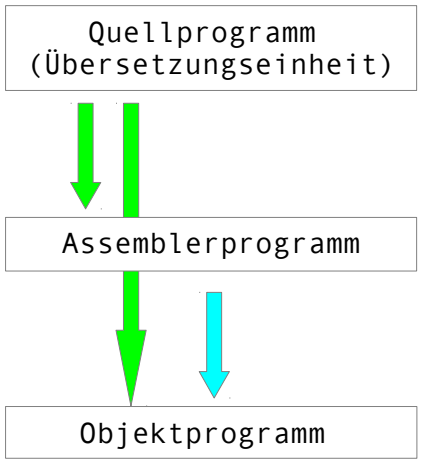
siehe: https://homepages.thm.de/~hg53/ksp-ws1617/manuskript.pdf

Vom Quellprogramm zum Objektcode

Quell-Programme werden als Text in eine Datei geschrieben

Handelt es sich um ein Programm einer Sprache, die in Maschinencode übersetzt wird, dann transformiert der Compiler den Text entweder in

- **Assembler-Code,**
  der dann vom Assembler in Objektcode übersetzt wird, oder
- **direkt in Objekt-Code**



Objekt-Code

ist eine Kombination aus:

- **Maschinencode**
- **Daten**
- **Informationen zur korrekten Ablage des Codes im Hauptspeicher**

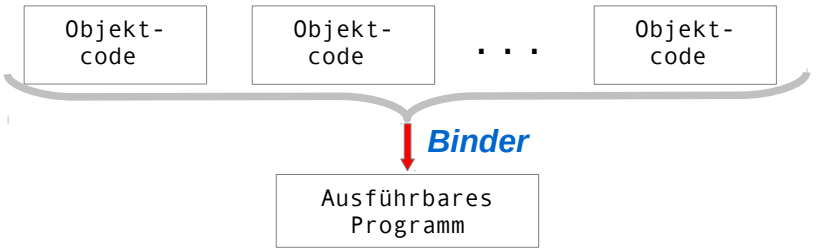
Einem Programm entsprechen in der Regel mehrere Objektdateien:

- **Getrennt übersetzte Programmteile,**
- **Bibliotheksfunktionen, etc.**

Der Objektcode enthält nur numerische (binäre) Daten:

- **keine symbolischen Adressen und**
- **keine symbolischen Maschinenbefehle**

Binder, ausführbare Datei



Binder / *Linker*

Der Binder (engl. *Linker*) erstellt

- aus der oder (meist) mehreren Dateien mit Objekt-Code (Objektdateien / Objekt-Module) eine
- ausführbare Binär-Datei (*executable image*)

Dabei werden vor allem die gegenseitigen Adressbezüge der Objektdateien aufgelöst

Ausführbare Datei / Ausführbares Programm / *(Program) Image*

Eine ausführbare Binär-Datei (engl. *program image* / *executable image* oder kurz *image*)

- ist eine Binär-Datei in einem speziellen Format das vom Betriebssystem abhängt.
- Beispiele
  - *.exe-Dateien* auf Windows-Systemen
  - Dateien im *a.out*- oder *Elf*-Format auf Unix / Linux-Systemen
  - *Mach-O 64-bit executables* auf Macs mit 64-Bit Intel-Prozessoren
  - …

Eine ausführbare Datei muss nicht zwingend eine Binäre-Datei sein.

- Beispiel: Shell-Skripte und andere Skripte werden von vielen Betriebssystemen auch als ausführbare Dateien anerkannt.

  Die Ausführung erfolgt dann nicht direkt durch die Hardware sondern indirekt durch einen einen Interpreter.

Lader

Lader / *Loader*

Der Lader (engl. *Loader*) lädt das *Image* in den Hauptspeicher und startet seine Ausführung.

Der Lader ist Teil des Betriebssystems:

Auf Unix-Systemen wird der Lader mit dem *exec*-Systemaufruf aktiviert.

Der Lade-Prozess umfasst

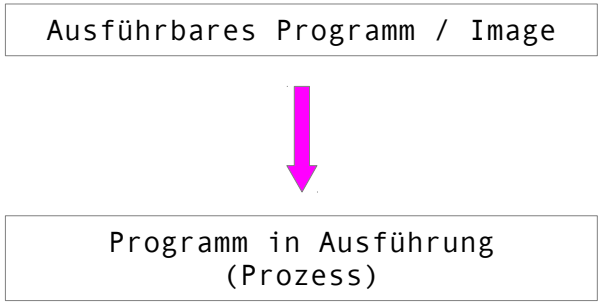
- Bereitstellen von Speicherplatz für Code und Daten
- Berechnen realer Adressen
  - **Absolut adressierte Programme**

    sind *images* mit festen realen Adressen die vom Binder berechnet wurden

    sie werden direkt in den zur Verfügung gestellten Platz im Speicher kopiert
  - **Relativ adressierte Programme**

    sind *images* mit relativen realen Adressen, die vom Binder berechnet wurden

    aus ihnen berechnet der Lader abhängig vom Zustand des Speichers und dem zugeteilten Platz die realen Adressen
- Sprung zur Startadresse des Programms



Lader: Segmente / Objektcode: Sektionen

Segmentierung

Ausführbare Programme (*images*) müssen nicht an einem Stück im Speicher platziert werden,
sie können in Segmente aufgeteilt im Speicher verstreut werden.
Sinn: Flexiblere Speicherzuteilung – mehrere kleine Speicherbereiche statt einem großen
Objektdateien

- enthalten dann Informationen über die Segmente und
- alle Adressen sind relativ zum Anfang ihres Segments

Virtueller Speicher

Ausführbare Programme werden auf modernen Plattformen *nicht* in Segmente geladen
Statt dessen werden sie in den virtuellen Speicher „geladen“
Der virtuelle Speicher

- ist ein privater großer Hauptspeicher für ein einzelnes Programm in Ausführung (= Prozess),
- dessen Existenz Hardware und Betriebssystem
  - dem Programm zur Laufzeit des Programms

  vorspielen.

Segmentierung, Segmente und Sektionen

Objekt-Dateien (und damit Assembler-Programme) sind in Sektionen (*Sections*) gegliedert.
Nicht verwechseln: Sektionen im Objektcode werden oft auch Segmente / *segments* genannt
Die Sektionen können in Segmente geladen werden – sie müssen aber nicht
Auf modernen Systemen spielen Segmente keine Rolle: Sektionen sind nur Gliederungen der Objekt-Dateien

Assembler

Assembler **ist eine Bezeichnung für zwei unterschiedliche Dinge:**

- **Assembler-Sprachen**

  Eine Klasse einfacher maschinennaher Programmiersprachen

- **Assembler**

  Übersetzer-Programme, die Programme in Assembler-Sprache in Maschinencode übersetzen

Assembler-Sprachen / Assembler-Programme

- **Symbolische Variante des Objektcodes**
- **Gehören immer zu einen bestimmten Maschinensprache + Objektcode-Format**
- **Unterscheiden sich vom Objektcode durch:**
  - **Verwendung von mnemonischen Namen statt Binärcodes**
  - **Verwendung von symbolischen Referenzen statt numerischen Speicheradressen**

Assembler

- **Übersetzen die symbolischen Namen in ihre Binärcodes**
- **Lösen die symbolischen Referenzen auf:**
  transformieren sie reale (absolute oder relative) Speicheradressen

Assemblercode und Objektcode

Objektcode

Binärdatei in einem festem Format

enthält typischerweise

- Header
- Text-Sektion („Segment“) mit Maschinencode
- Statische Daten (im Programm verwendete Strings, statische Arrays, …)
- Symbol- und Adresstabellen (importierte / exportierte Symbole, absolute Adressen, ...)

Assemblercode

Textdatei in festem Format

enthält prinzipiell die gleiche Information wie eine Objektdatei, allerdings

- in textueller Form und
- in einem eigenem Format

Assemblercode ist lesbar und etwas allgemeiner als Objekt-Code

- unabhängig vom dem BS
- aber immer abhängig von der CPU

Quell- und Assembler-Programme

Beispiel C-Programm und erzeugter Assembler-Code.

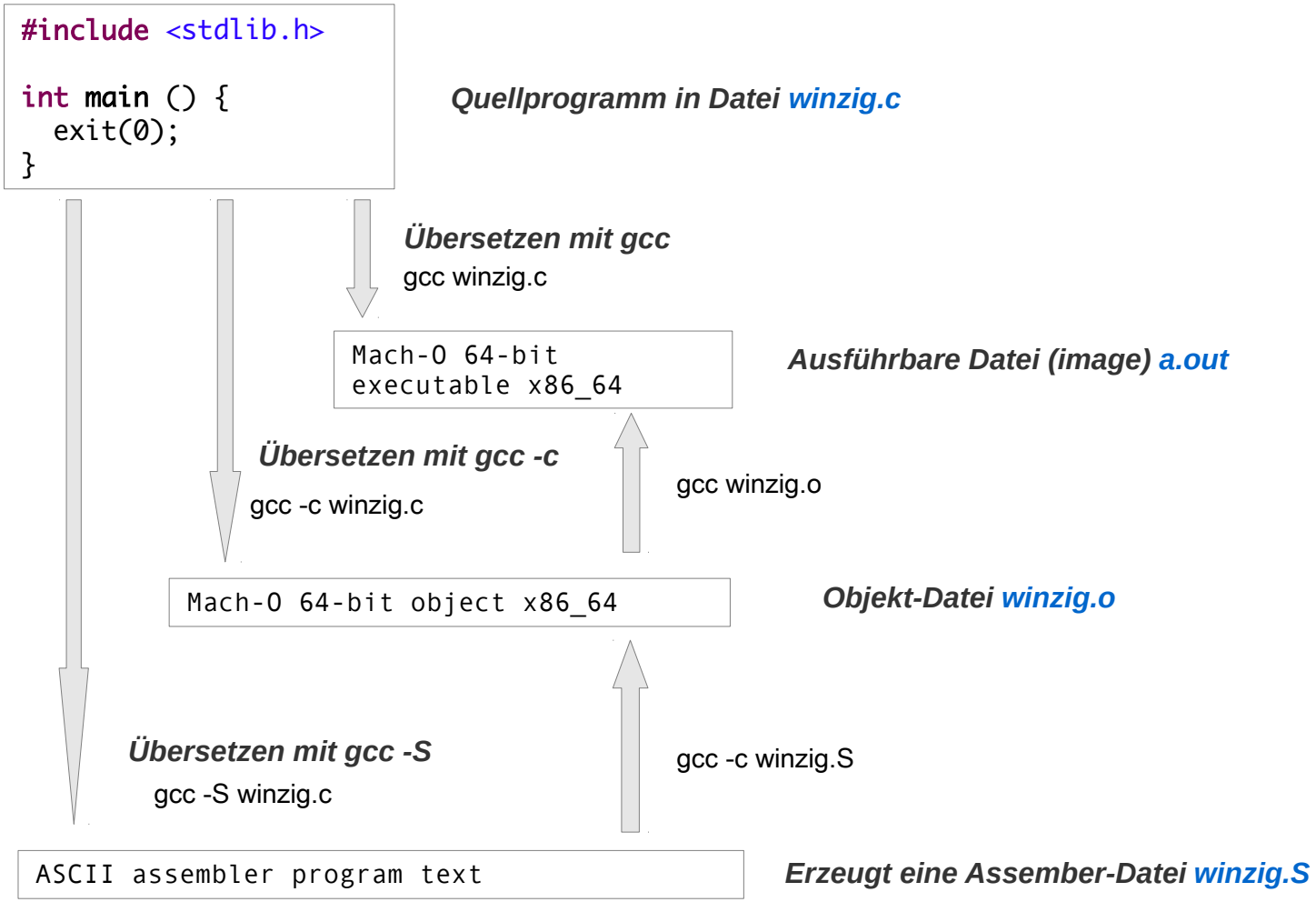
```
//
// prog1.asm -- an assembler example with global variables
//

//
// compute the gcd of two positive numbers
//
// global Integer x;
// global Integer y;
// x = readInteger();
// y = readInteger();
// while (x != y) {
//   if (x > y) {
//     x = x - y;
//   } else {
//     y = y - x;
//   }
// }
// writeInteger(x);
// writeCharacter('\n');
```

Assembler

```
// x = readInteger();
rdint
popg  0
// y = readInteger();
rdint
popg  1
// while ...
L1:
// x != y
pushg 0
pushg 1
ne
brf   L2
// if ...
pushg 0
pushg 1
gt
brf   L3
// x = x - y
pushg 0
pushg 1
sub
popg  0
jmp   L4
L3:
// y = y - x
pushg 1
pushg 0
sub
popg  1
L4:
jmp   L1
L2:
// writeInteger(x);
pushg 0
wrint
// writeCharacter('\n');
pushc '\n'
wrchr
halt
```

Aus der Veranstaltung Konzepte systemnaher Programmierung (Prof. Geisse) siehe https://homepages.thm.de/~hg53/ksp-ws1617/aufgabe3/prog1.asm

Quell-, Assembler-, Objekt-Programme / GCC Beispiel 1



Alle Beispiele auf einem Mac, andere Unix-Systeme entsprechend

Quell-, Assembler-, Objektprogramme und Images

GCC Beispiel 2

```
#include <stdlib.h>

int main () {
  exit(0);
}
```

winzig.c

gcc -S winzig.c

```
        .macosx_version_min 10, 10
        .globl  _main
        .align  4, 0x90
_main:                                  ## @main
        .cfi_startproc
## BB#0:
        pushq   %rbp
Ltmp0:
        .cfi_def_cfa_offset 16
Ltmp1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp2:
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        xorl    %edi, %edi
        movl    $0, -4(%rbp)
        callq   _exit
        .cfi_endproc

.subsections_via_symbols
```

winzig.S
GNU-Assember*

_main *ist ein exportiertes Symbol. Die main-Funktion wird von aussen aufgerufen, das C-Programm ist de-facto nur ein getrennt übersetztes Unterprogramm*

Direktiven *beginnen mit einem Punkt. Es sind Anweisungen zur Steuerung des erzeugten Objektcodes / Maschinencodes: Einträge in Tabellen, Ausrichtung des Codes, etc.*

Maschinencode *in symbolischer Form ist der Hauptbestandteil eines Assemblerprogramms.*

_exit *ist ein importiertes Symbol. Die main-Funktion ruft den Systemaufruf exit*

Erzeugung eines Assembler-Programms aus einem C-Programm. (Auf einem Mac, andere Unix-Systeme entsprechend.)

***Achtung**: gcc erzeugt nicht den Assemblercode für ein vollständiges Programm, sondern nur den Code für eine Objektdatei, die die Funktion **main** enthält.*
*Diese Objektdatei muss vom Binder mit Startup-Code (**crt0.o** o.ä.) gebunden werden, der dann main aufruft.*

* siehe https://de.wikipedia.org/wiki/GNU_Assembler

Quell-, Assembler-, Objektprogramme und Images

GCC Beispiel 3

```
#include <stdio.h>

int main () {
    printf("Hello\n");
}
```

hello.c

gcc -S hello.c

```
        .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 10
        .globl  _main
        .align  4, 0x90
_main:                                  ## @main
        .cfi_startproc
## BB#0:
        pushq   %rbp
Ltmp0:
        .cfi_def_cfa_offset 16
Ltmp1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp2:
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        leaq    L_.str(%rip), %rdi
        movb    $0, %al
        callq   _printf
        xorl    %ecx, %ecx
        movl    %eax, -4(%rbp)          ## 4-byte Spill
        movl    %ecx, %eax
        addq    $16, %rsp
        popq    %rbp
        retq
        .cfi_endproc

        .section        __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
        .asciz  "Hello\n"

.subsections_via_symbols
```

***Das Assemblerprogramm hat zwei Sektionen. Es exportiert ein Label* (_main) *und importiert ein Label* (_printf).**

GCC Beispiel 4

```c
#include <stdlib.h>

int main () {
  exit(0);
}
```

winzig.c

gcc -c winzig.c

winzig.o

Objektdatei

gcc -o winzig.exe winzig.o

```
Module table (0 entries)
(__TEXT,__text) section
_main:
0000000100000f60    pushq    %rbp
0000000100000f61    movq %rsp, %rbp
0000000100000f64    subq $0x10, %rsp
0000000100000f68    xorl %edi, %edi
0000000100000f6a    movl $0x0, -0x4(%rbp)
0000000100000f71    callq 0x100000f76          ## symbol stub for: _exit
```

winzig.exe

Lesbare Ausgabe erzeugt mit
otool -dtrVM winzig.exe

Auch dieses repräsentiert kein vollständiges Programm. Der Startcode wird getrennt übersetzt und dazu gebunden.

Rechnerarchitektur

Rechnerarchitektur: Aufbau und Funktionsweise eines Computers

Für den Compilerbau wird ein Basiswissen zum Aufbau eines Rechners (Computers) benötigt

Speicher

- **Register**

  Speicherplätze in der CPU, enthalten Daten und Anweisungen

  Nur Daten / Befehle in Registern können verarbeitet werden

- **Hauptspeicher**

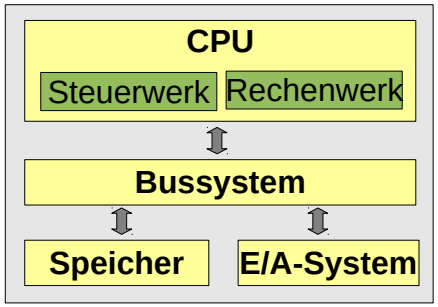
  Nur von hier können Daten / Befehle in die CPU / Register geladen werden.

  – **Cache-Speicher**

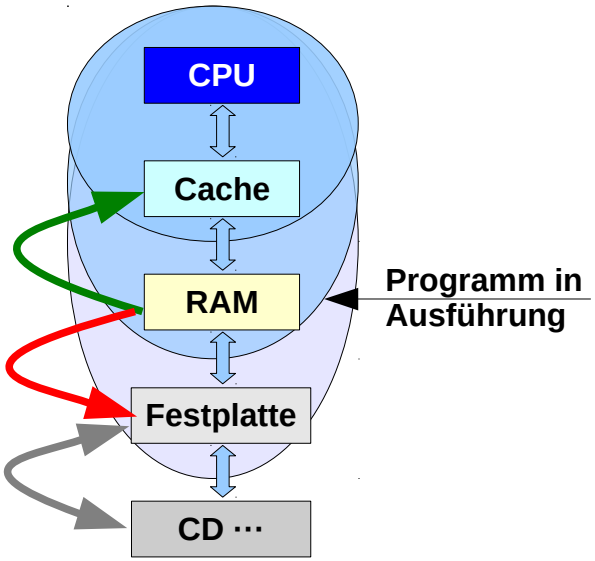
  Zwischenspeicher zwischen Hauptspeicher und externem Speicher, enthält häufig genutzte Daten.

  – **Sekundärspeicher / externer Speicher**

  Festplatte, USB-Stick etc.



Rechneraubau



Speicherhierarchie
- Register
- Level-1 Cche
- Level-2 Cache
- RAM (Speicher)
- Platte
- andere Datenträger

Registermaschine

- **Modell eines Rechners**

  CPU mit Registern, externer Speicher

  Übliche Struktur von Rechnern

- **Komponenten**
  - **Prozessor**
  - **Speicher**
  - **E/A-Einheiten**
- **Prozessor**
  - **Akkumulator*:***
    Vor der Ausführung einer zweistelligen Operation befindet sich der eine Operand im Datenspeicher und der andere im Akkumulator. Danach befindet sich das Ergebnis im Akkumulator.
  - **Befehlsregister*:***
    Enthält immer den momentan auszuführenden Befehl.
  - **Befehlszähler*:***
    Enthält den momentan auszuführenden Befehl.
- **Befehl**
  - **auszuführenden Operation**
  - **Adresse der Operanden (Register und Speicherstellen)**

Stack-Maschine

- **Alternatives Modell eines Rechners**

  CPU mit „unendlich“ vielen Registern die als Stack organisiert sind

  Un-übliche Struktur von realen Rechnern, oft bei virtuellem Maschinen verwendet

  Nachteil: Ineffizienter als Registermaschine

  Vorteil: einfache Bedienung bei Auswertungsauswertung

- **Befehl**

  auszuführenden Operation + (eventuell) eine Speicherstelle:

```
Load  <Adr>
Store <Adr>
Add
Sub
...
```

  alle Operation – ausser Laden und Speichern – werden auf dem Stack ausgeführt

Datenrepräsentation

- **Adressierbare Daten**

  Daten im (Haupt-) Speicher werden über feste Adressen angesprochen

  - **Bytes (Einheiten von 8 Bits) haben eine Adresse**

- **Ladbare / speicherbare Daten**

  Daten werden zwischen CPU-Register und dem Speicher ausgetauscht

  - **Bytes können geladen und gespeichert werden**
  - **Halbworte (2 Bytes) und Worte (4 Bytes) können meist auch mit einem Befehl transferiert werden**

    Alignment: Oft können Worte und Halbworte nicht von / zu beliebigen (Byte-) Adressen im Speicher transferiert werden, sondern nur zu solchen die durch 2 oder 4 teilbar sind.

- **Endian-ness**

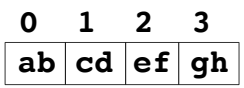
  Werden Einheiten mit mehr als einem Byte geladen oder gespeichert, dann kann das auf zwei Arten geschehen:

  - **Big-Endian**

    Bytes mit höherer numerischer Wertigkeit stehen an niedrigeren Adressen

  - **Little-Endian**

    Bytes mit höherer Wertigkeit stehen an Speicherstellen mit höherer Adresse



Halbwort im Speicher mit Byteadressen

Halbwort im Register bei big-endian

Halbwort im Register bei little-endian

Adressierungsarten

Maschinenbefehle beziehen sich auf Daten: laden, speichern, mit arithmetischen oder logischen Operationen verarbeiten.

Die Daten sind in Registern, auf dem Stack, oder im Hauptspeicher zu finden

Sie müssen also adressiert werden.

Hier sind unterschiedliche Verfahren üblich, die als die Adressierungsart einer Maschinenarchitektur bezeichnet.

Daten können auf unterschiedliche Art angesprochen werden:

- **Daten in Registern**

  werden über den Registernamen angesprochen

- **Daten**

  können angesprochen werden über

  - **die Adresse des ersten Bytes angesprochen, oder**
  - **über ein Register mit der Adresse der Daten, oder**
  - **über Basisadresse in einem Register + Offset-Wert / Offset-Register**
  - **oder als Direktwert in einem Befehl**
  - **…**

Bedingungen und Kontrollstrukturen

Mit

- **unbedingten (springe zu Adresse x) oder**
- **bedingten Sprüngen (springe zu Adresse x falls …), oder**
- **mit bedingten Anweisungen**

können Schleifen und If-Anweisungen realisiert werden.

Bedingte Sprünge können unterschiedliche Formen annehmen, üblich ist:

- **Mit Vergleichs- / Test-Anweisungen wird ein Statusregister gesetzt**
- **Der Sprung erfolgt dann abhängig vom Inhalt des Statusregisters**

Bedingte Anweisungen werden nur ausgeführt, wenn das Statusregister einen bestimmten Wert hat

Beispiel:

```c
#include <stdio.h>

int main () {
    int a = 0;
    int b = 1;
    if (a < b)
       printf("Hello\n");
}
```

hello.c

gcc -S hello.c

```
        .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 12
        .globl  _main
        .p2align        4, 0x90
_main:                                  ## @main
        .cfi_startproc
## BB#0:
        pushq   %rbp
Ltmp0:
        .cfi_def_cfa_offset 16
Ltmp1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp2:
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        movl    $0, -8(%rbp)
        movl    $1, -12(%rbp)
        movl    -8(%rbp), %eax
        cmpl    -12(%rbp), %eax
        jge     LBB0_2
## BB#1:
        leaq    L_.str(%rip), %rdi
        movb    $0, %al
        callq   _printf
        movl    %eax, -16(%rbp)         ## 4-byte Spill
LBB0_2:
        movl    -4(%rbp), %eax
        addq    $16, %rsp
        popq    %rbp
        retq
.cfi_endproc

        .section        __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
        .asciz  "Hello\n"
```

Statusregister mit dem Ergebnis eines Vergleichs belegen;
Bedingter Sprung