MIPS Programmierung
Ergänzungen zum Vorlesungsstoff

1. SPIM
2. Assembler
3. Registersatz
4. Gleitkommaoperationen
5. Assembler, Binder und Lader

1 SPIM

- Der **SPIM** Simulator gestattet ein einfaches Testen von MIPS Assemblerprogrammen
- SPIM wurde von James Larus an der University of Wisconsin-Madison entwickelt
- Version 1 bereits in 1990, seitdem ständig weiter verbessert, aktuell ist Version 9
- SPIM arbeitet **simulativ**, d.h. SPIM erzeugt kein binäres Maschinenprogramm, sondern simuliert die Abarbeitung von MIPS Assemblerprogrammen auf einem MIPS Prozessor
- SPIM basiert in der aktuellen Version auf dem GUI-Toolkit **Qt** und ist erhältlich für Windows, Linux und MacOS

1 SPIM (2)

- QtSpim kann heruntergeladen werden von SourceForge: https://sourceforge.net/projects/spimsimulator/
- Eine gute Dokumentation zu QtSpim findet man in http://www.egr.unlv.edu/~ed/MIPStextSMv11.pdf
- Alternativ gibt es die ältere Version PCspim (Vorgänger von QtSpim) für Windows als „*portable App*“ unter http://sourceforge.net/projects/portablepcspim/
- Allgemeine Informationen zu SPIM und dem MIPS-32 Instruktionssatz findet man im Anhang A von: *Hennessy & Patterson, Computer Organization and Design: The Hardware/Software Interface* (auch online verfügbar)

2 Assembler

- Ein Assembler übersetzt ein Programm aus **Assemblersprache** in **binäres** Maschinenprogramm:
  - erlaubt Verwendung von **symbolischen Namen** für Speicheradressen und Sprungmarken
  - realisiert eine Vielzahl von **Pseudobefehlen** zur Erweiterung des Instruktionssatzes

    Beispiel: Der Pseudobefehl `mov $s0,$t1` des MIPS Assemblers erlaubt einen Datentransfer zwischen Registern und kann z.B. durch den MIPS Maschinenbefehl `addi $s0,$t1,0` realisiert werden
  - stellt **Direktive** zur Steuerung der Assemblierung und zur Deklaration von Daten bereit
  - erlaubt Arbeiten mit Zahlen in unterschiedlichen Formaten, z.B. dezimal (default), binär (Präfix `0b`), hexadezimal (Präfix `0x`)
  - unterstützt die Möglichkeit von Betriebssystem-Aufrufen (*System Calls*)

2 Assembler (2)

- Auswahl einiger **Pseudobefehle** des MIPS Assemblers:
  - `li reg, const16` : Laden einer 16-Bit Konstante in ein Universalregister (*load immediate*)
  - `li reg, const32` : Laden einer 32-Bit Konstante in ein Universalregister
  - `la reg, addr32` : Laden einer 32-Bit Adresse in ein Universalregister (l*oad address*)
  - `move reg1, reg2` : kopiert den Inhalt des Universalregisters reg2 in das Register reg1
  - `neg reg1, reg2` : Laden des Universalregisters reg1 mit dem Inhalt –reg2 (*negate*)
  - `bgt reg1, reg2, label` : Sprung nach `label`, wenn Inhalt von reg1 größer ist als Inhalt von reg2 (*branch if greater*)

2 Assembler (3)

- Auswahl einiger **Direktive** des MIPS Assemblers:

  `.text` markiert Beginn des Programmsegments

  `.data` markiert Beginn des Datensegments

  `.align n` richtet nächstes Datum an 2^n Bytegrenze aus

  `.word w1,w2,...,wn` füllt Speicher mit den vorzeichenbehafteten `n` 32-Bit Worten `w1,w2,...,wn`

  `.byte b1,b2,...,bn` füllt Speicher byteweise mit den angegebenen Inhalten `b1,b2,...,bn`

  `.space num` reserviert Speicher für `num` Bytes (nicht initialisiert)

  `.asciiz str` stellt den String `str` in den Speicher (mit Terminierung durch Byte Null)

  `.globl symb` deklariert Sprungmarke oder Adresse `symb` als global

- Konstanten werden (auch bei `.byte`) ohne Präfix `0b` bzw. `0x` stets als Dezimalwerte angesehen.

2 Assembler (4)

- Mögliche **Systemaufrufe** im MIPS Assembler bei Verwendung des SPIM Simulators (Auswahl):

code	Bezeichnung	Argumente und Wirkung
1	`print_int`	gibt Inhalt von `$a0` aus (Integer)
4	`print_string`	gibt einen String (mit Terminierung durch Byte Null) ab Adresse `$a0` aus
5	`read_int`	liest in `$v0` einen Wert ein
8	`read_string`	liest ab Adresse `$a0` einen String aus `$a1` Zeichen ein
9	`sbrk`	reserviert `$a0` Byte im Speicher, Startadresse wird in `$v0` zurückgegeben
10	`exit`	Gibt Kontrolle an Betriebssystem zurück

- Syntax ist:

```
li $v0, code   # Pseudobefehl "load immediate"
lw $a0, addr   # ggf. Argument laden
syscall
```

Übung: Erste Schritte mit QtSpim

1) Laden Sie QtSpim herunter und installieren Sie das Softwarepaket auf Ihrem Rechner!

2) Auf der Webseite der Vorlesung finden Sie das MIPS Beispielprogramm `first-spim.s`; laden Sie dieses Programm zunächst in einen Editor Ihrer Wahl. Was leistet es?

3) Laden Sie das Programm nun in den SPIM Simulator und starten Sie es!

4) Erstellen Sie auf der Grundlage von `first-spim.s` ein Programm `calc.s`, das den arithmetischen Ausdruck Z = 2*A + 3*B + A*B berechnet. Das Ergebnis Z soll in den Speicher geschrieben und zusätzlich ausgegeben werden.

5) Testen Sie Ihr Programm mit SPIM! Kontrollieren Sie, ob das richtige Ergebnis für Z in den Speicher geschrieben wird.

3 Registersatz

- MIPS ISA hat 32 Universalregister, die wichtigsten sind:

Name	Registernummer	Nutzung
`$zero`	0	der konstante Wert 0
`$v0–$v1`	2–3	Werte für Ergebnisse und für die Auswertung von Ausdrücken
`$a0-$a3`	4–7	Argumente
`$t0–$t7`	8–15	temporäre Variablen
`$s0–$s7`	16–23	gespeicherte Variablen
`$t8–$t9`	24–25	weitere temporäre Variablen
`$gp`	28	globaler Zeiger
`$sp`	29	Kellerzeiger
`$fp`	30	Rahmenzeiger
`$ra`	31	Rücksprungadresse

- noch nicht behandelt wurden `$gp und $fp` ...

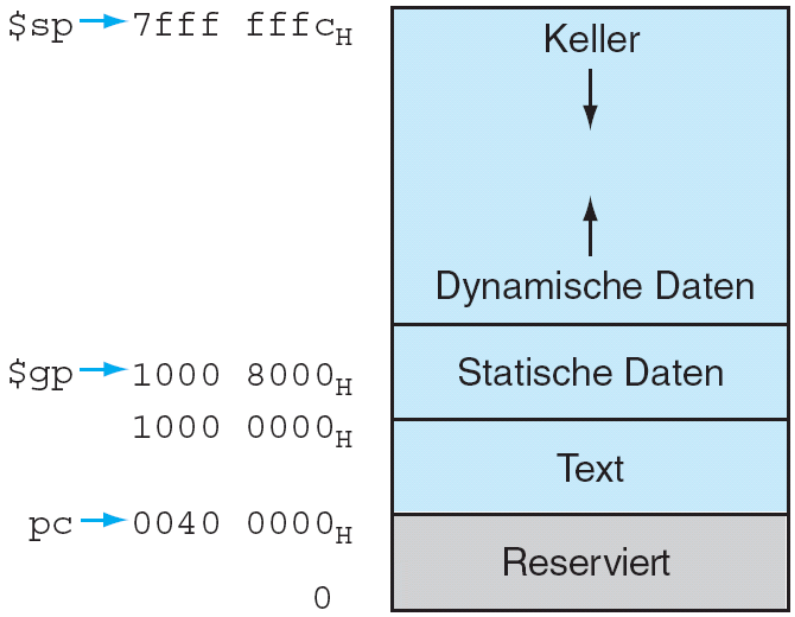
3 Registersatz (2)

- Das Register **`$gp`** (*global pointer*) dient in der MIPS ISA als Zeiger auf **statische Daten**, die im `.data` Segment deklariert wurden

- typische Speicheraufteilung MIPS ISA:

- Zeiger `$gp` wird i.a. mit dem Wert 10008000h initialisiert

  (mit positiven u. negativen Offsets kann auf den Speicherbereich von 10000000h bis 1000FFFFh zugegriffen werden)



3 Registersatz (3)

- Das Register **`$fp`** (*frame pointer*) wird oft als Zeiger auf den **Rahmen** (*frame*) einer Prozedur im Stack verwendet

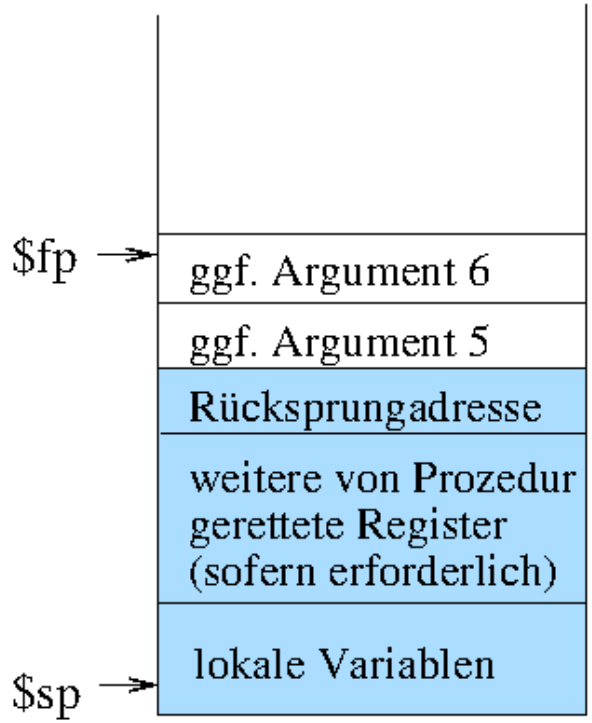
  (als Rahmen bezeichnet man den Platz im Stack, in dem beim Prozeduraufruf Argumente, gerettete Register, Rücksprungadressen und lokale Variablen der Prozedur gespeichert werden)

- Zeiger `$fp` zeigt i.a. auf den Beginn des Rahmens, d.h. auf das letzte Argument

- Zeiger `$fp` kann auch der einfachen Adressierung der geretteten Registerinhalte dienen

  (Zeiger `$sp` ist hierzu wenig geeignet, da er bei lokalen Prozedurvariablen auf den zuletzt belegten Stackplatz zeigt)

  im Bsp. rechts: `sw $ra, -8($fp)`



3 Registersatz (4)

- Bei Verwendung des **`$fp`** Registers werden in der aufgerufenen Prozedur **zu Beginn** z.B. folgende Schritte durchgeführt:

  1) Reserviere den weiteren für Rahmen benötigten Speicher auf Stack durch Subtraktion des Bytebedarfs *b* (für Register und lokale Variablen) von `$sp`

  2) Sichere `$ra`, `$fp` und ggf. weitere Register, sofern sie in Prozedur geändert werden, z.B. `$s0` bis `$s7` oder `$a0` bis `$a3`

  3) Setze `$fp` auf ersten Rahmeneintrag

- Vor dem Verlassen müssen **am Ende** der Prozedur folgende Schritte durchgeführt werden:

  1) Restauriere die geretteten Register

  2) Freigeben des Stacks durch Addition von *b* zu `$sp`, zusätzlich muss auch der Speicherplatz für die ggf. vom aufrufenden Programm auf den Stack geschriebenen Argumente freigegeben werden

  3) Rücksprung zur Adresse in `$ra`

3 Registersatz (5)

- Die Verwendung des `$fp` Registers ist **optional** und wird von einigen Compilern nicht genutzt.
- Bei einfachen Unterprogrammen, die keine oder nur wenige lokale Variablen haben, ist es oft einfacher, nur **relativ zu `$sp`** die Rahmeneinträge zu adressieren.
- Ruft ein Unterprogramm keine weiteren Unterprogramme auf, so muss der Inhalt von `$ra` auch nicht auf dem Stack gerettet werden.
- Ohne Optimierung legt ein Compiler **alle lokalen Variablen** eines Unterprogramms auf dem Stack an, bei höherer Optimierungsstufe werden diese i.a. bedeutend effizienter nur in Registern gehalten.

4 Gleitkommazahlen in MIPS ISA

- MIPS ISA unterstützt Gleitkommaformate für einfache und doppelte Genauigkeit
  - arithmetische Operationen für einfache Genauigkeit erhalten den Suffix **`.s`** (*single precision*), arithmetische Operationen für doppelte Genauigkeit den Suffix **`.d`** (*double precision*)
  - ansonsten werden die gleichen Befehle wie für Operationen auf ganzzahligen Daten verwendet: `add`, `sub`, `mul`, `div`
  - 32 zusätzliche Register für Gleitkommazahlen: **`$f0`** bis **`$f31`**

    (bei einigen MIPS CPUs haben diese nur eine Breite von 32 Bit, für Gleitkomma-zahlen doppelter Genauigkeit werden zwei Register konkateniert, daher sind dort nur `$f0`, `$f2`, `$f4`, ... zulässig)
  - Beispiele:

```
add.s $f2,$f3,$f4
mul.s $f5,$f2,$f3
sub.d $f6,$f8,$f12
div.d $f6,$f6,$f14
```

4 Gleitkommazahlen in MIPS ISA (2)

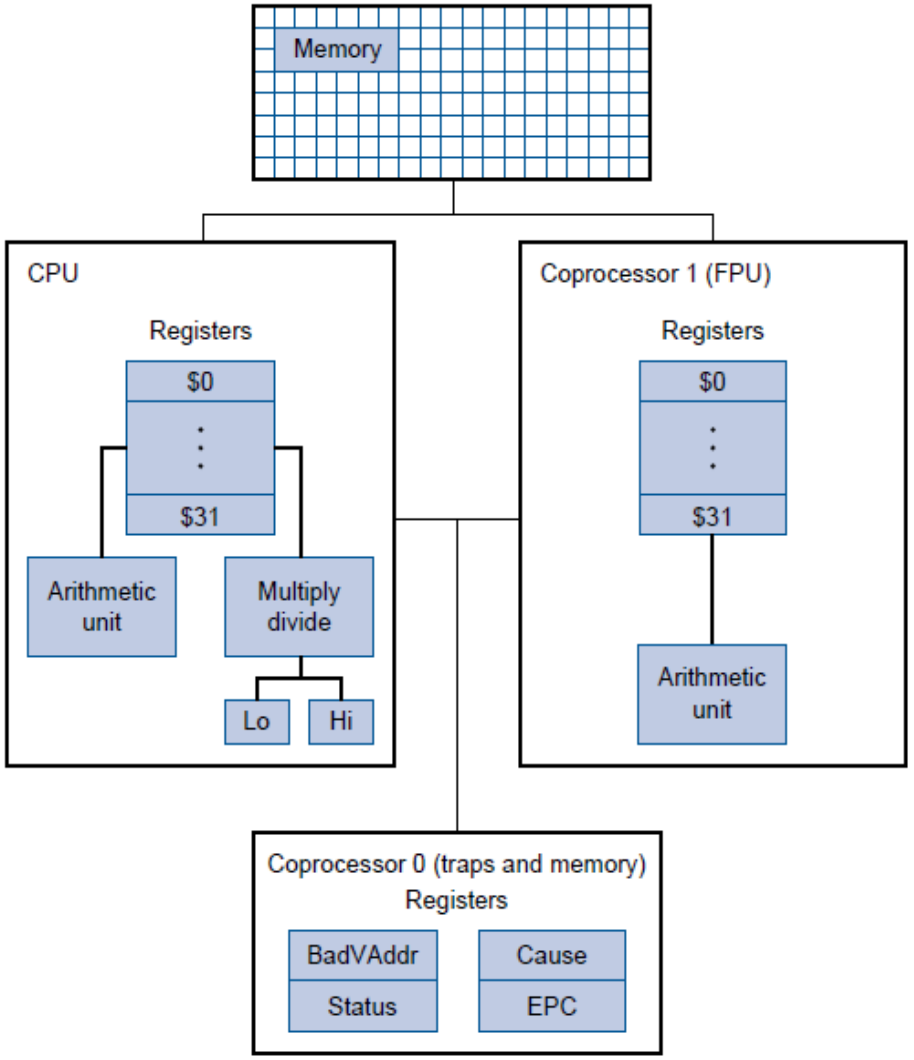
- Die Gleitkomma-Arithmetik war in der MIPS Architektur ursprünglich im separaten **Koprozessor 1** implementiert
  (in den aktuellen CPU-Version ist Koprozessor natürlich auf Chip integriert)
- Zugriff auf Gleitkommazahlen im MIPS Programm:
  - die zusätzlichen Assembler-Direktiven **`.float`** und **`.double`** ermöglichen das Ablegen von Gleitkommawerten im Speicher
  - Laden und Speichern von 32-Bit Gleitkommazahlen:

    ```
    lwc1 freg,disp(reg)      # load word to coprocessor 1
    swc1 freg,disp(reg)      # store word from coprocessor 1
    ```

    (für die Adressberechnung beim Speicherzugriff wird trotzdem ein Integer-Register `reg` verwendet)
  - Der Assembler bietet zusätzlich die Pseudo-Instruktionen **`l.s`**, **`l.d`**, **`s.s`** und **`s.d`** an

4 Gleitkommazahlen in MIPS ISA (3)

- Vollständige Architektur einer MIPS CPU mit 2 Koprozessoren:



4 Gleitkommazahlen in MIPS ISA (4)

- Bedingte Verzweigungen sind in der MIPS FPU mächtiger als in der Integer-ALU:
  - Es gibt für Gleitkommazahlen 3 verschiedene Vergleichsoperationen **`x`**: equal (**`eq`**), less than (**`lt`**), less than or equal (**`le`**)

    Ein Vergleich kann mit einer der folgenden zwei Instruktionen realisiert werden:

    ```
    c.x.s   freg1,freg2        # für einfache Genauigkeit
    c.x.d   freg1,freg2        # für doppelte Genauigkeit
    ```

    (diese Befehle setzen im Koprozessor ein internes Bedingungsflag `c`, wenn die Bedingung erfüllt ist)
  - für die bedingten Sprünge gibt es dann die Befehle

    ```
    bc1t   label               # Sprung, wenn Bedingung erfüllt (true)
    bc1f   label               # Sprung, wenn Bedingung nicht erfüllt
                               # ist (false)
    ```

4 Gleitkommazahlen in MIPS ISA (5)

- MIPS Beispielprogramm für Gleitkommazahlen: Umrechnung Fahrenheit in Celsius

```
float f2c (float fahr) {
  return ((5.0/9.0) * (fahr - 32.0));
}
```

Per Konvention wird Gleitkommaargument in `$f12` (ggf. + `$f13`), Rückgabewert in `$f0` (ggf. + `$f1`) abgelegt:

```
f2c:    lwc1   $f16,const5        # Konstante 5.0
        lwc1   $f18,const9        # Konstante 9.0
        div.s  $f16,$f16,$f18
        lwc1   $f18,const32       # Konstante 32.0
        sub.s  $f18,$f12,$f18     # fahr aus f12
        mul.s  $f0,$f16,$f18
        jr     $ra
```

4 Gleitkommazahlen in MIPS ISA (6)

- Zur Unterstützung von Gleitkommazahlen bietet SPIM folgende zusätzlichen Direktive an:

  `.float f1,f2,...,fn` füllt Speicher mit den `n` 32-Bit IEEE Gleitkommazahlen `f1,f2,...,fn`

  `.double d1,d2,...,dn` füllt Speicher mit den `n` 64-Bit IEEE Gleitkommazahlen `d1,d2,...,dn`

- Für die Ein/Ausgabe von Gleitkommazahlen gibt es folgende zusätzlichen Systemaufrufe:

`code`	Bezeichnung	Argumente und Wirkung
2	`print_float`	gibt Inhalt von `$f12` aus (float)
3	`print_double`	gibt Inhalt von `$f12/$f13` aus (double)
6	`read_float`	liest in `$f0` einen Wert (float) ein
7	`read_double`	liest in `$f0/$f1` einen Wert (double) ein

4 Gleitkommazahlen in MIPS ISA (7)

- Die folgenden MIPS Instruktionen können für die **Konvertierung** von Daten zwischen *integer*, *single* und *double* verwendet werden:

  `cvt.d.w $f0,$f2` : *Convert integer from* `f2` *to double in* `f0/f1`
  `cvt.s.w $f0,$f2` : *Convert integer from* `f2` *to float in* `f0`
  `cvt.d.s $f0,$f2` : *Convert float from* `f2` *to double in* `f0/f1`
  `cvt.s.d $f0,$f2` : *Convert double from* `f2/f3` *to float in* `f0`
  `cvt.w.d $f0,$f2` : *Convert double from* `f2/f3` *to integer in* `f0`
  `cvt.w.s $f0,$f2` : *Convert float from* `f2` *to integer in* `f0`

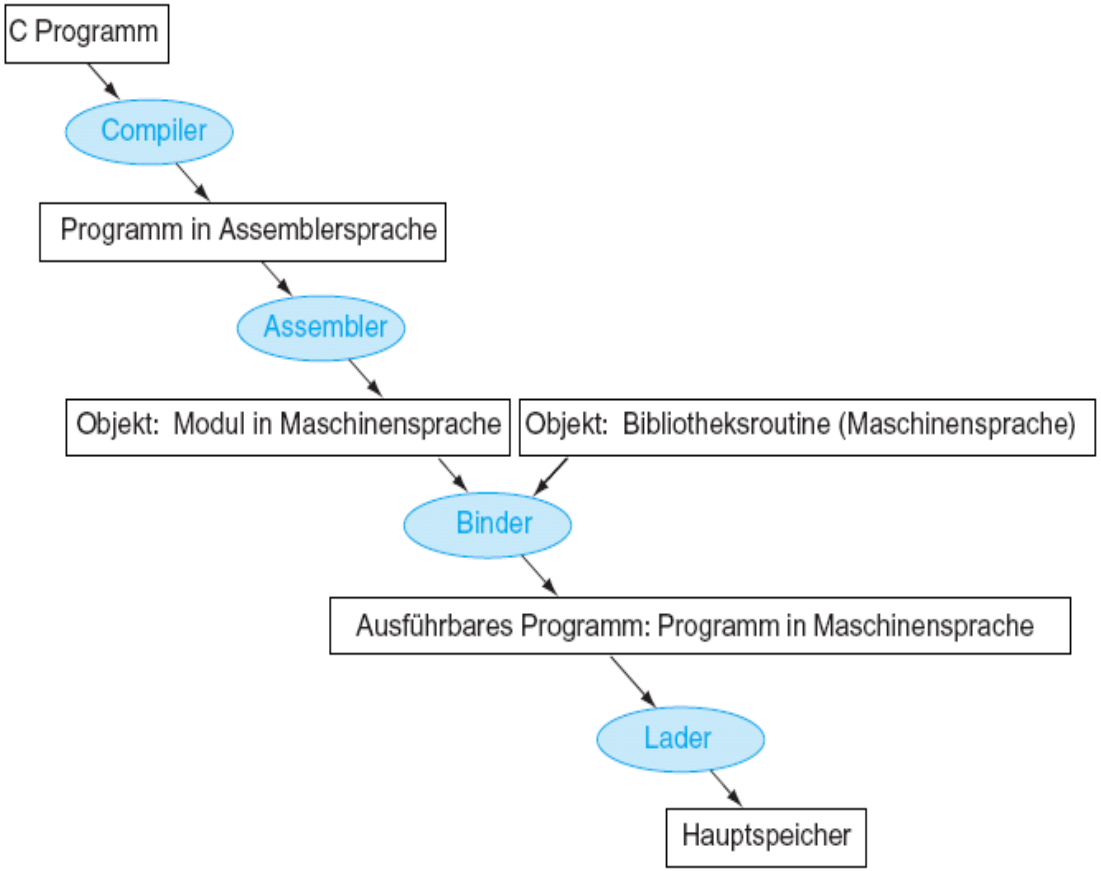
- Alle *integer* Werte müssen explizit vom Universalregister zu einem Gleitkommaregister tranportiert werden oder umgekehrt:

  `mtc1 $t0,$f0` : *Move from integer register* `t0` *to float register* `f0` *in coprocessor* 1

  `mfc1 $t0,$f0` : *Move from float register* `f0` *in coprocessor* 1 *to integer register* `t0`

5 Assembler, Binder und Lader

- Bei der Übersetzung eines C-Programms werden folgende Schritte ausgeführt:



5 Assembler, Binder und Lader (2)

- Vom Assembler erzeugte **Objektdatei** besteht aus:
  - **Header** mit Informationen über Größe und Positionen der einzelnen Teile der Objektdatei
  - **Programmsegment** (auch als **Textsegment** bezeichnet) mit binärem Maschinenprogramm
  - **Datensegment** mit statischen Daten
  - Informationen zur **Relokation** des Programms (Tabelle mit Instruktionen und Daten, die beim Laden von absoluten Adressen abhängen)
  - eine **Symboltabelle** mit Adressen von extern ansprechbaren Symbolen und intern nicht auflösbaren symbolischen Referenzen
  - ggf. Informationen zum **Debugging** des Assemblerprogramms
- Detailliertes Format der Objektdatei ist vom Betriebssystem abhängig!

5 Assembler, Binder und Lader (3)

- Eine vom Assembler erzeugte Objektdatei ist nicht ausführbar,
  - da sie noch keine absoluten Adressen enthält
  - da sie häufig externe Referenzen enthält (zu Daten und Prozeduren in anderen Objektdateien oder Bibliotheken)
- Der **Binder** (*Linker*) fügt mehrere Objektdateien zusammen und generiert ein **ausführbares Binärprogramm**
- Hierzu führt er folgende Schritte durch:
  - Abbilden der Text- und Datensegmente auf den Speicher und Festlegen der jeweiligen Speicherbereiche
  - Bestimmung globaler, absoluter Adressen für alle Marken unter Verwendung der Symboltabellen aller Objektmodule
  - Anpassen aller internen und externen Referenzen in den Textsegmenten
  - Erstellen eines ausführbaren Programms

5 Assembler, Binder und Lader (4)

- Bei der Ausführung eines Programms durch den **Lader** (*Loader*) werden folgende Schritte ausgeführt:
  - Einlesen des Headers, um die Größe des Daten- und Textsegmentes zu ermitteln
  - Bereitstellen von Arbeitsspeicher für Daten und Text
  - Laden von Code und initialisierten Daten in Arbeitsspeicher
  - Initialisieren einiger Register und des Stackzeigers
  - Aufruf einer Startprozedur, die eventuelle Aufrufparameter von der Benutzershell in Argumentregister kopiert und das Hauptprogramm aufruft