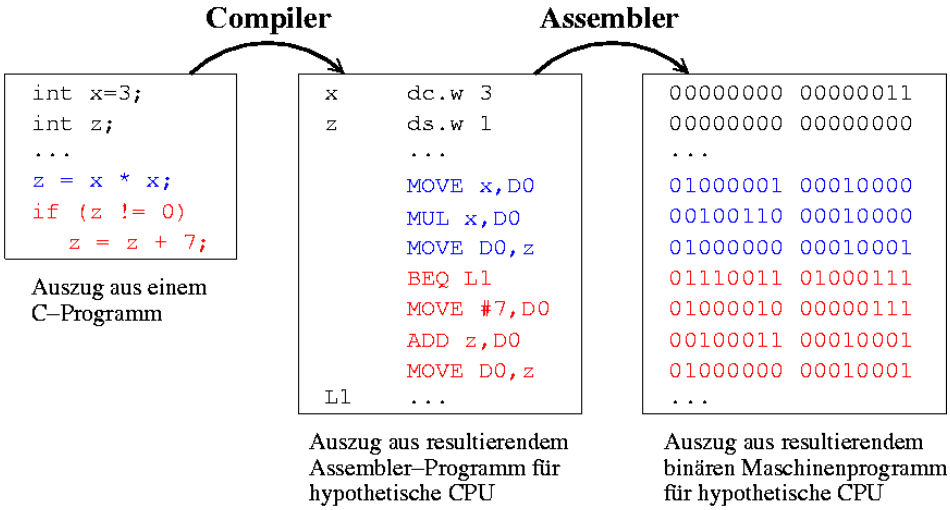
Teil 1: Prozessorstrukturen

Inhalt:

- Mikroprogrammierung
- Assemblerprogrammierung
- Motorola 6809: ein einfacher 8-Bit Mikroprozessor
- Mikrocontroller
- Koprozessoren
- CISC- und RISC-Prozessoren
- Intel Pentium

Compiler und Assembler

Übersetzung eines Programms (vereinfacht):



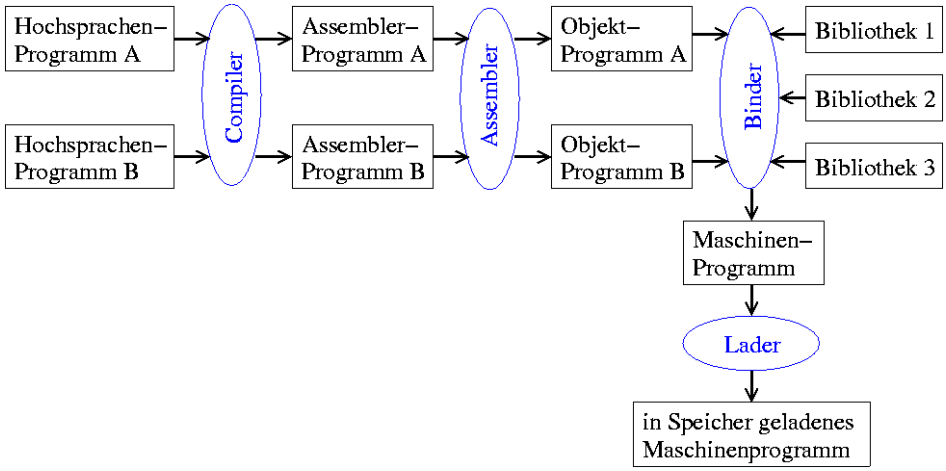
Auszug aus einem C–Programm

Auszug aus resultierendem Assembler–Programm für hypothetische CPU

Auszug aus resultierendem binären Maschinenprogramm für hypothetische CPU

Compiler, Assembler, Binder und Lader

Übersetzung von zwei Programmen aus einer Hochsprache in ein Maschinenprogramm:



Compiler

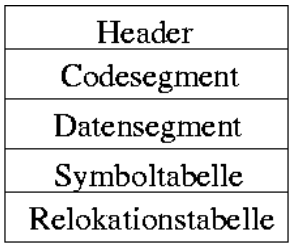
- Compiler übersetzt ein Programm aus einer Hochsprache in ein Assemblerprogramm in der Assemblersprache des Zielrechners
- Ziele einer Hochsprache:
  - Problemformulierung unabhängig vom Zielrechner
  - Kompakte, leicht verständliche und i.a. mit geringem Aufwand wartbare Formulierung
  - Sprache kann problemangepaßt sein
- Beispiele für Hochsprachen:
  - allgemein: C (relativ systemnah), C++ (objektorientierte Erweiterung von C), PASCAL, MODULA, JAVA, ...
  - problemangepaßt: VHDL (Hardwarebeschreibung), PROLOG (logisches Schließen), ...

Assembler

- Assembler übersetzt ein in Assemblersprache geschriebenes Programm in ein Objektprogramm
- Aufbau und Ziele einer Assemblersprache:
  - Symbolische Repräsentation eines Maschinenprogramms, enthält z.B. symbolische Operationscodes und symbolische Marken
  - angepaßt an die Instruktionssatz-Architektur (ISA) des Zielrechners
  - enthält Assemblerdirektiven, z.B. zur Datenorganisation
  - trotz der Maschinennähe eine noch lesbare Formulierung, jedoch hoher Aufwand für Wartung
- Beispiele für Assemblersprachen:
  - für Mikroprozessoren: x86-Assembler, 68k-Assembler, 6809-Assembler, SPARC-Assembler, MIPS-Assembler, ARM-Assembler, ...
  - für Großrechner: IBM 360/370/390 Assembler, ...

Assembler (Forts.)

- Typischer Aufbau des Objektprogramms:



- Objektprogramm enthält:
  - Kopf („*Header*“, mit Magic Number, Startadresse, Größenangaben aller Segmente)
  - Programmtext in Maschinensprache
  - zugehörige binäre Daten (Konstanten und initialisierte Variablen)
  - Namen der global deklarierten Funktionen und Daten
  - Namen der benötigten externen Funktionen und Daten
  - Verschiebungsinformationen
- Beispiele für UNIX-Objektformate: a.out (klassisches Format), COFF („*Common Objekt File Format*“), ELF („*Executable and Linkable Format*“)

Assembler (Forts.)

- Aufgaben eines Assemblers:
  - Ersetzung jeder symbolischen mnemotechnischen Instruktionsbezeichnung durch den entsprechenden Maschinenbefehl unter Berücksichtigung der verwendeten Registerbezeichnung und der Adressierungsart
  - Ersetzung symbolischer Operanden (Sprungziele, Variablen) durch Adressen (Referenzen) relativ zum Anfang des Programms
  - Umwandlung von Konstanten in maschineninterne Darstellung
  - Interpretation von Assemblerdirektiven
  - Erzeugen einer Tabelle nicht aufgelöster symbolischer Operanden
  - Ausgabe des übersetzten Programmes im Objektformat
  - Generierung eines Assemblerlistings
- Die meisten Assembler arbeiten in 2 Phasen:
  - **Phase 1**: Absuchen des Programms nach Marken und Zuordnung von relativen Adressen zu Marken
  - **Phase 2**: vollständige Übersetzung des Programms

Binder

- Aufgaben eines Binders („*Linker*“):
  - Zusammenfassen der Code- und Datensegmente mehrerer Objektprogramme, ggf. Verschiebung von Instruktionen und Daten bei Konflikten
  - Auflösen der Querbezüge zwischen Programmteilen, d.h. Ersetzen von symbolischen Bezeichnern, die in anderen Programmteilen definiert sind, durch Adressen
  - Überprüfung des resultierenden Programms auf noch nicht aufgelöste symbolische Bezeichner (externe Referenzen)
  - Suche in Bibliotheken nach externen Referenzen und Einfügen der gefundenen Code- oder Datenfragmente in das resultierende Maschinenprogramm

Lader

- Aufgaben eines Laders („*Loader*“):
  - Größenbestimmung von Code- und Datensegment
  - Anforderung eines entsprechend großen Speicherbereiches vom Betriebssystem
  - Relokation: Ersetzung aller als Operanden verwendeter Marken (Sprungziele und Daten) durch die absoluten Adressen im zugewiesenen Speicherbereich (mittels Relokationstabelle)
  - Kopieren von reloziertem Code- und Datensegment in Arbeitsspeicher
  - Programmausführung ab der relozierten Startadresse
- In einfachen Systemen sind häufig Assembler, Binder und Lader kombiniert.

Assemblerformat

- In einer Zeile befindet sich ein Maschinenbefehl oder eine Assemblerdirektive (d.h. Steueranweisung an Assembler)
- Jede Zeile besteht aus Markenfeld, Opcodefeld, Operandenfeld und Kommentarfeld:
  `[<label>] <opcode> [<operands>] [<comment>]`
- Optionales Markenfeld enthält Sprungmarken oder symbolische Bezeichner für Variablen bzw. Konstanten
- Bemerkungen:
  - gelegentlich ist festes Zeilenformat vorgegeben, z.B. Marke in Spalte 1-6, Opcode ab Spalte 8, usw.
  - Bezeichner für Marken haben i.a. eine maximale (signifikante) Länge
  - Definition von Marken häufig mit `<label>:` statt `<label>`
  - Assemblerdirektiven von Assembler zu Assembler unterschiedlich

Assemblerdirektiven

- Zuweisung („*equate*“) eines numerischen Wertes zu einem symbolischen Bezeichner durch `<name> equ <value>`

  Beispiele:
  ```
  a       equ 3              (Dezimalzahl)
  start   equ $1000          (Hexadezimalzahl)
  mask    equ %01010101      (Binärzahl)
  ```

- Festlegung einer Zieladresse („*origin*“) für Instruktionen und Daten durch `org <addr>`

  Beispiele: `org 1000, org $FFFF, org start`

- Kennzeichnung des Endes des zu assemblierenden Programms durch `end`

- Kommentare werden mit einem bestimmten Kommentarsymbol (z.B. `*` oder `;`) eingeleitet und laufen bis zum Zeilenende

Assemblerdirektiven (Forts.)

- Speichern von Konstanten („*define constant*“) mittels
  ```
  <label> dc.b <value>      (8-Bit Wert, „byte“)
  <label> dc.w <value>      (16-Bit Wert, „word“)
  <label> dc.l <value>      (32-Bit Wert, „long word“)
  ```

- Reservierung eines Speicherbereiches („*define storage*“) mittels
  ```
  <label> ds.b <num>        (<num> 8-Bit Werte)
  <label> ds.w <num>        (<num> 16-Bit Werte)
  <label> ds.l <num>        (<num> 32-Bit Werte)
  ```

- Beispiele:
  ```
  one   dc.b %00000001
  max   dc.l $FFFFFFFF
  array ds.w 100
  ```

- Hinweis: Im 6809-Assembler des TI-Praktikums müssen `fcb` („*form constant byte*“) und `fdb` („*form double byte*“) statt `dc.b` bzw. `dc.w` sowie `rmb` („*reserve memory byte*“) statt `ds.b` verwendet werden !

Beispiel

- einfaches Assemblerprogramm (Zielrechner: hypothetische CPU)

```
seven   equ    7
        org    $10              * Datemsegment ab Adresse 10H
x       dc.w   -3               * setze x=–3
y       dc.w   $A               * setze y=10
z       ds.w   1                * reserviere Speicher fuer 16–Bit Variable z
tmp     ds.w   1                * reserviere Speicher fuer 16–Bit Variable tmp

        org    $50              * Codesegment ab Adresse 50H
start   MOVE   x,D0
        MUL    y,D0             * berechne x*y
        BPL    L1               * springe nach L1, wenn x*y>0
        MOVE   D0, tmp          * rette x*y in tmp
        MOVE   #seven,D0        * lade 7
        ADD    tmp,D0           * berechne tmp+7=x*y+7
L1      MOVE   D0,z             * speichere Ergebnis in z
        end
```


Beispiel (Forts.)

- vom Assembler aus Beispiel generiertes Maschinenprogramm mit errechneten Adressen:

```
        Daten /                      Assemblerprogramm
Adr.    Opcode + Operand             (Adressen eingesetzt)
 10                                          org    $10
 10    11111111 11111101      x       dc.w   -3
 11    00000000 00001010      y       dc.w   $A
 12    00000000 00000000      z       ds.w   1
 13    00000000 00000000      tmp     ds.w   1

 50                                          org    $50
 50    01000001 00010000      start   MOVE   $10,D0
 51    00100110 00010001              MUL    $11,D0
 52    01110010 01010110              BPL    $56
 53    01000000 00010011              MOVE   D0,$13
 54    01000010 00000111              MOVE   #7,D0
 55    00100011 00010011              ADD    $13,D0
 56    01000000 00010010      L1      MOVE   D0,$12
                                      end
```

Weitere Fähigkeiten von Assemblern

- Moderne Assembler bieten weitere Hilfsmittel zur einfachen Assembler-Programmierung an:
  - Pseudo-Instruktionen (z.B. ein universelles `MOVE <src>,<dest>` für jeden Datentransport im Rechner)
  - Makros (für haüfig verwendete Sequenzen von Assemblerbefehlen)
  - Bedingte Assemblierung (z.B. für verschiedene Versionen):

```
if <condition>
...    (bedingt zu übersetzende Assemblerzeilen)
endif
```

  - Direktiven zur Definition von Zeichenketten und globalen symbolischen Bezeichnern
  - Direktiven zur expliziten Deklaration von Code- und Datensegment
- Cross-Assembler ermöglichen die Generierung von Maschinen-programmen, wenn Wirtsrechner nicht zugleich Zielrechner ist.

Assemblersprache vs. Hochsprache

- Vorteile der assembersprachlichen Beschreibung:
  - Reduktion der Programmgröße (im Vergleich zu übersetztem Hochsprachenprogramm)
  - Erhöhung der Ausführungsgeschwindigkeit
  - exakte Angabe aller Abläufe in der CPU
  - weitgehend vorhersagbares Zeitverhalten
    (⇒ Unterstützung von Echtzeitanforderungen möglich)
- Nachteile:
  - Schwieriger zu verstehen
  - Länger als hochsprachliche Beschreibung des Algorithmus
  - höhere Gefahr von Programmierfehlern
  - schlechtere Wartbarkeit
  - zielrechnerspezifisch, d.h. keine Portierbarkeit