Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Verfasser:
D. Gachet / HTA-FR - Telekommunikation

HTA-FR

Embedded systems 1

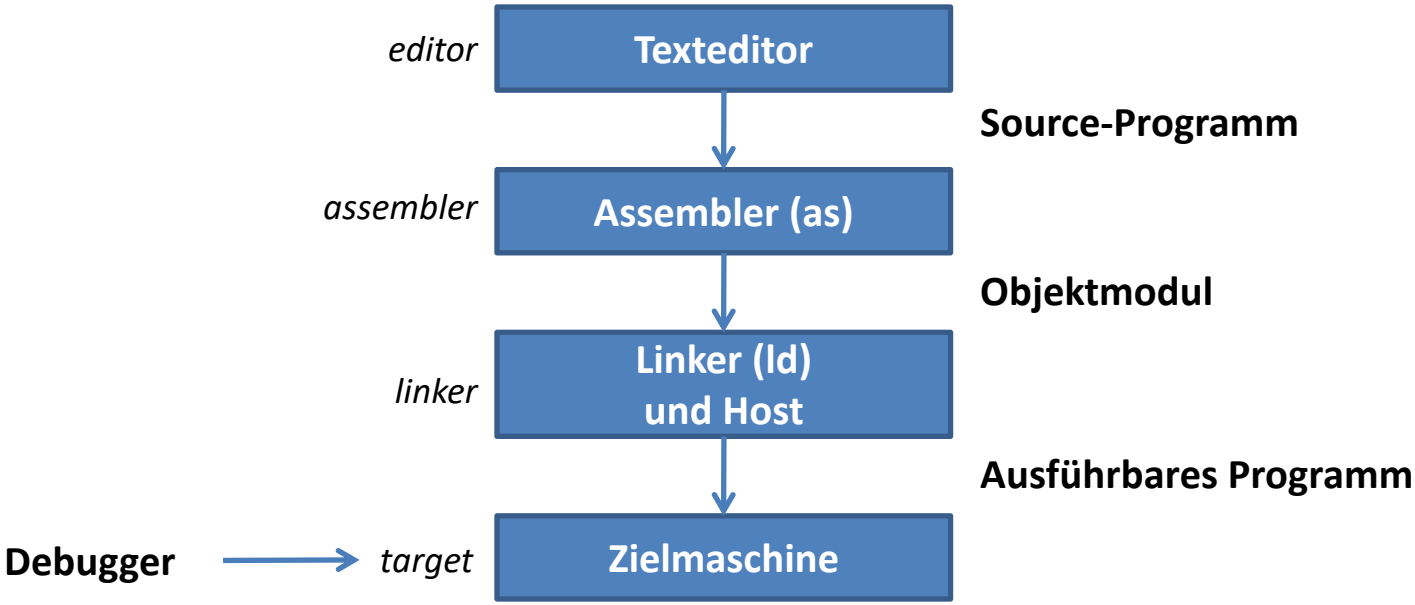
a.10 Einführung in die Assemblersprache (as)

Klassen I-2 / T-2 // 2018-2019

Inhalt

- Einführung
- Syntax
- Anweisungen
- Abschnitte
- Code-Beispiel

Entwicklungsprozess



- **Wenn der gesamte Prozess auf dem gleichen Rechner abläuft wird, spricht man von nativer Entwicklung → *native development***
- **Wenn die Texteingabe, das Assemblieren und das Linken auf einer Maschine erfolgt, die nicht für das Ausführen der Programme benutzt wird (Zielmaschine), spricht man von einer Cross-Entwicklungsumgebung → *cross development***

Syntax der Assemblersprache

- **Die Assemblersprachen unterscheiden sich abhängig von den Prozessoren und den Lieferanten von Assemblierungssoftware.**

- **Sie enthalten alle**
  - Assemblierungsanweisungen
  - Assemberbefehle
  - Anweisungen für das Speichern der Daten

- **Jede Zeile ist in 3 Felder aufgeteilt**

<LABEL> <BEFEHL> <KOMMENTAR>

- **Im vorliegenden Kurs werden wir speziell den GNU-Assembler (as) behandeln. Der Assembler ist ausführlich im Dokument "01_gnu-assembler.pdf" beschrieben.**

Symbole und Labels/Bezeichnungen

- **Ein Symbol**
  - Beginnt normalerweise mit einem Buchstaben und kann auch Ziffern und Sonderzeichen (._$), aber keine Leerzeichen enthalten
  - Ist "case sensitive"
  - Kennt keine Längenbegrenzung (alle Zeichen sind massgeblich)
  - Darf in einem Programm nur einmal definiert werden
  - Kann einen arbiträren Wert darstellen, der auf das Zeichen ‘ =‘ folgt

- **Ein Label/eine Bezeichnung**
  - Ist ein Symbol, auf das unmittelbar ein Doppelpunkt (:) folgt
  - Stellt den aktuellen Wert des Speicherplatzes dar
    - Adresse eines Datenelements im Speicher
    - Adresse eines Befehls im Code

- **Beispiele:**

```
symbol_1
symbol_2 = 19
label_1:
```

Konstanten

- **Der Assembler kennt 3 Arten von Konstanten**
  - Ganze Zahlen
    - Dezimalzahlen: eine Folge von Ziffern zwischen 0 und 9, die aber nicht mit einer 0 beginnen darf
    - Binärzahlen: 0b oder 0B, gefolgt von einer Folge von 0 und 1
    - Oktalzahlen: 0 gefolgt von einer Folge von Ziffern zwischen 0 und 7
    - Hexadezimalzahlen: 0x oder 0X gefolgt von einer Folge von Ziffern zwischen 0 und 9 und a und f oder A und F
  - Zeichen
    - Ein einzelnes ASCII-Zeichen wird entweder mit einem Apostroph und anschliessendem Zeichen (z. B. ‘a) oder mit einem Zeichen zwischen zwei Apostrophen (z. B. ‘a’) dargestellt
    - Um einen Backslash zu schreiben, muss dieser verdoppelt werden, d. h. ‘\\
  - Strings oder Zeichenketten
    - Ein ASCII-String wird einfach als Zeichenkette zwischen Anführungs- und Schlusszeichen geschrieben, z. B. "hello world!\n"
    - Die Escape-Zeichen können ebenfalls verwendet werden, d. h.:
      \b, \f, \n, \r, \t, \\, \" oder ein Oktalwert \000 oder Hexadezimalwert \x1a

Kommentare

- **Es bestehen zwei Möglichkeiten, um Kommentare in ein Assemblerprogramm einzufügen**
  - Die Form C mit '/*' gefolgt von einem '*/'.
    - Darf nicht verschachtelt werden.
    - Kann an jeder beliebigen Stelle des Programms eingefügt werden.
    - Kann sich über mehrere Zeilen erstrecken.
  - Zeilenkommentar. Er beginnt mit dem Zeichen '@' oder der Form C mit "//" und endet mit dem Zeilenende.

- **Beispiele**

```
/* dieses Beispiel eines Kommentars ist absolut zulässig
 * auf mehreren Zeilen geschrieben.
 */
      mov r0,#78 @ das ist ein Zeilenkommentar für den Befehl
      mov r1,#59 // das ist ebenfalls ein Zeilenkommentar
```

Ausdrücke

- In einem Assemblerbefehl kann das Operandfeld einen Ausdruck enthalten
- Zu den oft verwendeten Ausdrücken gehören die Addition (+), die Subtraktion (-), die Multiplikation (*), die ganzzahlige Division (/), der Rest (%) und die Verschiebungen (<<, >>)
- Beispiele

```
size = 100
list:     .space size+2,0xff        // list[102] = {0xff,..., 0xff}

          mov     r0, #size/4       // r0 = 25
          mov     r1, #2+4*size     // r1 = 402
          ldr     r2, =list+27      // r2 = &list[27]
          ldrb    r2, [r2]          // r2 = list[27]
          mov     r3,#'a'-'A'       // r3 = 32
          ldr     r4,=1<<30         // r4 = 0x40000000
```

Anweisungen für das Speichern der Daten

- **Für das Speichern von Daten sind sechs Arten von Anweisungen vorgesehen**
  - `.byte <expressions>` → Definition einer Reihe von 8-Bit-Worten
  - `.hword | .short <expressions>` → Definition einer Reihe von 16-Bit-Worten
  - .hword | .short <expressions> → Definition einer Reihe von 32-Bit-Worten
  - `.ascii <string>` → Definition eines Strings
  - `.ascii <string>` → Definition eines mit 0 terminierten Strings
  - `.space <number_of_bytes> [,<fill>]` → Definition eines Speicherbereichs

- **Diese *Pseudo-Operationen* werten ihre Argumente in der Assemblierungsphase aus und legen das Ergebnis beim Laden des Programms im Speicher ab.**

- **Beispiele**

```
Long:     .long   0x12345678        // definiert 1 Konstante von 32 Bit
Short:    .short  0xabcd            // definiert 1 Konstante von 16 Bit
Byte:     .short  0xabcd            // definiert 4 Konstanten von 8 Bit
msg:      .asciz  'Hello World'     // definiert einen mit 0 terminierten
String
table:    .space   3*4              // reserviert 3 longs (3*4*8 Bit)
list:     .space  100,0xff          // füllt 100 Bytes mit 255
```

Anweisungen

- **Die Assembler verfügen im Allgemeinen über eine grosse Anzahl von Anweisungen, die die Entwicklung komplexer Programme ermöglichen. Nachstehend sind einige der nützlichsten aufgeführt**

  - `.global symbol {, symbol}`
    erlaubt, das Symbol (Label) für den Linker sichtbar zu machen

  - `.align expression`
    erlaubt, Daten mit einer bestimmten Anzahl Bytes auszurichten

  - `.include "file"`
    erlaubt, eine Datei einzufügen

  - `.if absolute expression, .else, .elseif, .endif`
    erlaubt eine bedingte Assemblierung

Abschnitte

- **Die modernen Assembler sind in der Lage, Codeteile mit gleicher Qualität in getrennten Abschnitten zusammen zu fassen**

- **Sie sind auf die folgende Art definiert**

```
.section <Bezeichnung des Abschnitts>
```

- **Zu den 4 gebräuchlichsten gehören**
  - Codeabschnitt (`.section .text` **oder** `.text`)
  - Abschnitt der initialisierten Daten (`.section .data` **oder** `.data`)
  - Abschnitt der zurückgesetzten Daten (`.section .bss` **oder** `.bss`)
  - Abschnitt der Konstanten (`.section .rodata`)

Code-Beispiel

```
/** copyright & heading... */

// Export public symbols
        .global main, res, var2, i

// Declaration of the constants
        LOOPS = 10

// Initialized variables declaration
        .data
        .align   8
res:    .long    0
var2:   .short   30

// Uninitialized variables declaration
        .bss
        .align   8
i:      .space   4
```

```
// Assembler functions implementation
        .text
main:   nop
        mov  r0, #LOOPS
        ldr  r1, =var2
        ldrh r1, [r1]
        ldr  r3, =res
        ldr  r4, =i
        mov  r5, #0
        str  r5, [r4]
next:   ldr  r2, [r3]
        add  r2, r1
        str  r2, [r3]
        ldr  r5, [r4]
        add  r5, #1
        str  r5, [r4]
        cmp  r5, r0
        bne  next
1:      nop
        b    1b
```