Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Hardwarenahe Softwareentwicklung

Assembler-Direktiven

V5.1, ©2023 roger.weber@bfh.ch

Lernziele

Sie sind in der Lage:

- Assembler-Direktiven des GNU-ARM Assemblers zu erklären.
- Assembler-Direktiven in einfachen Assemblerprogrammen zweckmässig einzusetzen.

Inhaltsverzeichnis

Übersicht

Einleitung

- Assembler-Direktiven sind Pseudoinstruktionen zur Steuerung des Assembliervorganges.
  - Sie erzeugen keinen Maschinencode.
  - Sie sind vom verwendeten Assembler abhängig.
- Die wichtigsten Funktionen sind:
  - Steuerung des Assemblers
  - Steuerung des Linkvorganges (Sektionen, Startadressen)
  - Definition von Variablen und Konstanten

Kurzübersicht (1)

Namensfeld	Anweisung	Operandenfeld	Beschreibung
	.thumb		Verwendung des Thumb Befehlssatzes
	.syntax		Spezifikation der verwendeten Syntax
	.cpu	CPU-Familie	Vewendete CPU-Faumilie
	.global	Symbolname	Export eines Symbols
	.extern	Symbolname	Import eines Symbols (wird von GNU ignoriert)
	.include	PfadFilename	Fügt Dateiinhalte ein
Symbolname	.asciz	String	Definiert Stringkonstanten
Symbolname	.byte .hword .word	Wert	Reserviert Speicher und initialisiert Datenwert
Symbolname	.space	Anzahl, Wert	Reserviert Speicherplatz mit Initialisierung für ganzen Block
	.align	Ausrichtung, Füllwert	Ausrichten der Speicherplatzadresse

Kurzübersicht (2)

Namensfeld	Anweisung	Operandenfeld	Beschreibung
	.equ	Symbolname, Wert	Wertzuweisung für ein Symbol
	.org	Adresse	Zuweisung einer absoluten Adresse
	.section	Symbolname, Flags	Speicherbereich definieren oder wechseln.
	.text .data .bss		Standard-Sections
	.end		Quellcode-Ende
	.if .ifdef .ifndef .endif	Bedingung	Bedingte Assemblierung
	.macro .endm	Symbolname, Parameter	Makrodefinition
	.rep, .irp .endr	Anzahl Parameter	Wiederholung von Codeblöcken

Instruktionssatz, Import und Export

.thumb, .syntax, .cpu

- **.thumb** → Thumb-Instruktionssatz
- **.syntax unified** → Verwendung der UAL (Unified Assembler Language), Superset für ARM und Thumb Syntax
- **.cpu** → Verwendete CPU-Familie
- Beispiel:

```
.thumb
    @ Thumb-Instructions (16/32 bit)
.syntax unified
    @ Use Unified Assembler Language
.cpu cortex-m7
    @ Use Cortex-M7 family
```

.global, .extern

- **.global** exportiert ein lokal definiertes Symbol (Subroutine, Variable...).
  Es wird dadurch öffentlich erklärt und in anderen Modulen zugreifbar.
- **.extern** zeigt, dass das Symbol in einem anderen Modul definiert ist.
  - Beim GNU-Assembler hat .extern keine Wirkung (wird ignoriert).
  - Für die Dokumentation aber trotzdem sinnvoll

main.s

```
.extern InitDisplay
   ...
   ...
   ...
   ...
   BL InitDisplay
```

lcd.s

```
.global InitDisplay
   ...
   ...

InitDisplay:
   ...
   ...
   MOV pc,lr
```

.include

- Syntax: **.include “Pfad/Filename“**
- Bezugspfad: aktuelles Verzeichnis, wo das Objektfile abgelegt wird.

- Beispiel:

```
.include      "../src/myincludes.s"
```

- Im Quellcode wird die Zeile .include “../src/myincludes.s“ durch den Inhalt in der Datei “myincludes.s“ ersetzt.
- In Pfadangaben ist statt “\“ (Windows) ein “/“ (UNIX) zu verwenden.

Speicherplatz reservieren

.asciz

- **.asciz** :Stringliteral mit einem Nullbyte terminiert (wie C- Strings).
- Die ASCII-Werte der einzelnen Zeichen werden direkt aufeinander folgend im Speicher abgelegt.
- Syntax:
  {label:} <**.asciz**> <**string**> **{,**<**string**>**}**
- Beispiel (Auszug aus dem Listing):

```
                                    .data
0000    48616C6C        string1:    .asciz "Hallo"
        6F00
```

.byte

- **.byte**: Definition von Byte-Konstanten oder Variablen.
- Syntax:
  {label:} .byte <byte1> {,<byte2>} . . .
- Das Label verkörpert einen Namen, der als symbolische Adresse für den Zugriff benutzt werden kann.
- Beispiel:

```
                                .data
0000    416225          b1:     .byte 'A','b',37
0003    20D0            b2:     .byte 0x20, -0x30
0005    F608            b3:     .byte -10,010
0007    0A              b4:     .byte 0B1010
```

.hword

- **.hword**: Definition von 16-Bit Konstanten oder Variablen.
- Syntax:
  {label:} .hword <short1> {,<short2>}
- Beispiel:

```
                                .data
0000    1100            h1:     .hword  0x11
0002    33224400        h2:     .hword  0x2233,0x44
0006    ...
```

.word

- **.word**: Definition von 32-Bit Konstanten oder Variablen
- Syntax:
  {label:} .word <word1> {,<word2>}
- Beispiel:

```
                                .data
0000    11000000    w1:         .word   0x11
0004    55443322    w2:         .word   0x22334455,0x66
        66000000
000C    ...
```

.space

- **.space**: reserviert Speicherplatz in Form eines Byte-Blocks.
- Syntax:
  {label:} .space <AB> {,<FB>} ...
  - AB: Anzahl reservierte Bytes
  - FB: Füllbyte für den gesamten Block
- Beispiel:

```
                                    .bss
0000    00000000    label_1:        .space 0x10
        00000000
        00000000
        00000000
                                    .data
0010    AAAAAAAA    label_2:        .space 0x10000,0xAA
        AAAAAAAA
        AAAAAAAA
        ........
10010   FFFFFFFF    label_3:        .space 0x14,0xFF
        FFFFFFFF
        ........
```

.align

- ARM-CPU: Code und 32-Bit Daten werden immer auf 32-Bit Adresse im Speicher ausgerichtet.
- .align: Ausrichtung auf die nächste 32-Bit Adresse, einfügen von 0 bis 3 Null-Bytes.
- Beispiel:

```
0000    AA55        b1: .byte   0xaa,0x55
0002    0000            .align              @ Nun werden zwei Bytes
                                            @ 0x00 eingefuegt
0004    48616C6C    s1: .asciz  "Hallo"
        6F00
```

Symbole

.equ, .set, =

- **.equ**, **.set** oder = weisen einem Symbol einen Wert zu

Vorteile

→ erlauben zentrale, symbolische Definitionen
→ Übersichtlichkeit und Wartungsfreundlichkeit

- Beispiel:

```
1                       .text
2                       .equ newline, 0xa
3                       .set bitmask, 0b10010111
4                       TAB = 011
5
6  0000 F04F000A        MOV r0,#newline
7  0004 F04F0197        MOV r1,#bitmask
8  0008 EA000101        AND r1,r0,r1
9  000c F04F0309        MOV r3,#TAB
10
11                      .set bitmask, 0b00000111    @ bitmask redefined
12
13 0010 F04F0064        MOV r1,#bitmask
```

Aufteilung in Adressbereiche

Sections (1), Basisklassen

- Aufteilung der Adressbereiche für Programm und Daten in Blöcke (Segmente, Sektionen):
- Codesegmente (**.text**): ausführbarer Code, oft im Flash.
- Datensegmente : Variablen, Stack ... im RAM
  - initialisierte Datensegmente (**.data**)
  - nicht initialisierte Datensegmente (**.bss**)

Sections (2)

- Beispiel Sections:

```
        .data                           @ Initialisierte Daten / RAM, section .data
b1:     .byte 0x22,0x33
msg:    .asciz "hallo"

        .text                           @ Programmspeicher / Flash, section .text
main:
        MOV r0, #1
        BL  mysub1
        B   main

mysub1:
        ...
        MOV pc,lr
```

Papierübung

- Schreiben Sie folgenden C-Code in Assembler:

```
#define MAX 0x137
int     var1;
char    var2 = 'a';
int     var3 = 0x57;

void main(void){
    var1 = MAX;
    while(1){
        var1++;
    }
}
```

Operatoren

Operatoren

- Innerhalb von Assembleranweisungen stehen für Berechnungen zahlreiche Operatoren zur Verfügung.
- Für die Operatoren gelten die selben Regeln und Konventionen wie in ANSI-C.
- Bedingung für die Berechnung ist, dass die **Operanden zur Assemblierzeit einen definierten Wert aufweisen**.

Operator	Priorität	Beschreibung
/	1	Division (ganzzahlig)
*	1	Multiplikation
%	1	Divisionsrest (Modulus)
< <	1	Schieben nach links
> >	1	Schieben nach rechts
|	2	Bitweise ODER-Verknüpfung
^	2	Bitweise EXOR-Verknüpfung
&	2	Bitweise UND-Verknüpfung
!	2	Bitweise ODER-NICHT-Verknüpfung
+	3	Addition
-	3	Subtraktion
==	3	Vergleich auf Gleichheit
<>	3	Vergleich auf Ungleichheit
<	3	Relationalvergleich kleiner als
>	3	Relationalvergleich grösser als
<=	3	Relationsvergleich kleiner/gleich als
>=	3	Relationsvergleich grösser/gleich als
&&	4	Logisches UND
||	5	Logisches ODER

Kontrollstrukturen

Kontrollstrukturen

- Assembler Kontrollstrukturen dienen zur Steuerung des Assembliervorganges.
  - Bedingte Assemblierung (.if, .ifdef ...)
  - Makrodefinitionen (.macro ...)

.if

- **.if**: Assemblieren des nachfolgenden Codeblockes nur, wenn eine Bedingung erfüllt ist.
- Syntax:
  .if <expression>
  ...
  {.else}
  ...
  .endif
- Wenn der logische Ausdruck wahr ist, wird der Block assembliert.
- Andernfalls wird, wenn ein .else-Block vorhanden ist, der .else-Block bis zum zugehörigen .endif assembliert.

.ifdef

- **.ifdef**: Assemblieren des nachfolgenden Codeblockes nur, wenn ein Symbol definiert ist.
- Das Symbol muss zuvor als Label mit .set, .equ oder = definiert worden sein.
- Syntax:
  .ifdef <symbol>
  ...
  {.else}
  ...
  .endif

.ifdef (2)

- Beispiel: Für Testzwecke kann selektiv Code eingebunden werden. Nachfolgend wird beim Programmlauf ein Debug-Haltepunkt ausgeführt, wenn das Symbol DEBUG zuvor definiert worden ist.

```
.set DEBUG, 1
...
start:
    LDR r0,=0x80
.ifdef DEBUG
    BKPT                @ Execute only in DEBUG-mode
.endif
    LDR r1,=0x80
```

.ifndef

- **.ifndef**: der nachfolgende Code wird nur assembliert, wenn das Symbol undefiniert ist.
- Die Wirkung ist analog .ifdef.
- Syntax:
  .ifndef <symbol>
  ...
  {.else}
  ...
  .endif

.macro

- **.macro**: erzeugt für ein Symbol eine bestimmte Menge Code (textuelle Substitution, wie ANSI-C).
- Makros können parametrisiert erfolgen.
- Syntax:
  .macro <Name> {<arg1>}{,<arg2>}...{,<argN>}
  {Codeblock}
  .endm

.macro (2)

- **Name** gemäss Regeln für Assemblersymbole, oft in Grossschrift.
- Die **Argumente** verkörpern formale Parameter, ähnlich einer Funktion in C.
- Für den Parameter-Zugriff innerhalb des Makros ist ein Backslash '\' notwendig.
- Eine Makroanweisung kann vorzeitig mit .exitm beendet werden.

.macro (2)

- Beispiel: Ein Makro, welches die Inhalte von Core-Registern schiebt. Falls der Schiebewert positiv ist, nach links, falls der Wert negativ ist nach rechts.
- Vorher: r0 = 0x80, r1 = 0x80

```
.macro SHIFT a,b
    .if \b < 0
        MOV \a,\a,ASR #-\b
        .exitm
    .endif
    MOV \a,\a,LSL #\b
.endm
. . .
SHIFT r0,2
SHIFT r1,-2
```

- Nachher: r0 = 0x200, r1 = 0x20

Wiederholungsstrukturen

Wiederholungsstrukturen

- Mit Wiederholungsanweisungen kann ein Codeblock eine bestimmte Anzahl Male wiederholt werden.
- Der GNU-Assembler stellt hierzu zwei Anweisungen zur Verfügung:
  - .rept (Repeat)
  - .irp (Indefinite Repeat)

.rept

- **.rept**: Wiederholt einen Codeblock eine bestimmte Anzahl Male.
- Syntax:
  .rept <Anzahl>
  {Codeblock}
  .endr
- Beispiel 1: Das Datenwortmuster 0x55AA55AA, 0x00000000, 0xAA55AA55, 0x11111111 soll als Folge von Datenworten 16x im Speicher abgelegt werden.

```
.rept 16
.word 0x55AA55AA
.word 0x00000000
.word 0xAA55AA55
.word 0x11111111
.endr
```

.rept (2)

- Beispiel 2: Mit .rept kann unter Verwendung eines Symbols auch eine Tabelle mit berechneten Werten aufgebaut werden.
- Nachfolgende Sequenz zeigt den Aufbau einer Byte-Tabelle mit den Werten von 0..255:

```
    .set    L1,0        @ L1 ist lokales Symbol
TAB:                    @ mit Startwert 0
    .rept   256         @ Schleife fuer 256 Werte
    .byte   L1          @ Tabellenelement = L1
    .set    L1,L1+1     @ L1++
    .endr
```