



Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Systèmes Embarqués 1 & 2

a.10 - Introduction à l'assembleur

Classes T-2/I-2 // 2018-2019

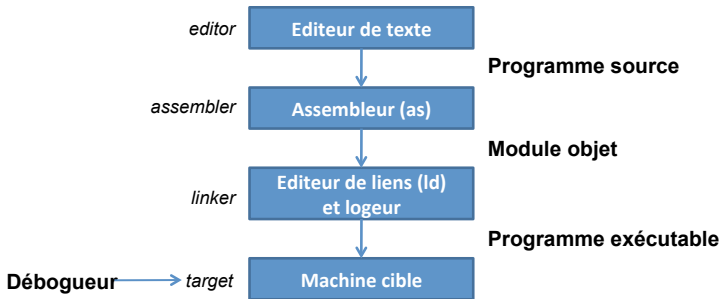
Daniel Gachet | HEIA-FR/TIC
a.10 | 16.11.2018



- Introduction
- Syntaxe
- Directives
- Sections



Processus de développement



- Lorsque tout le processus s'effectue sur le même ordinateur, on parle de développement natif → **native development**
- Lorsque l'édition de texte, l'assemblage et l'édition de liens s'effectuent sur une machine différente de celle qui exécute le programme (machine cible), on parle de développement croisé → **cross development**



- Les langages assembleurs sont dépendant du type de processeurs, mais également des concepteurs de logiciel
- Ils comportent tous
 - ▶ des directives d'assemblage
 - ▶ des instructions assembleur
 - ▶ des directives de stockage de données
- Chaque ligne est divisée en 3 champs

⟨ LABEL ⟩ ⟨ INSTRUCTION ⟩ ⟨ COMMENTAIRE ⟩
- Dans ce cours nous allons traiter plus spécialement l'assembleur GNU (as), assembleur décrit en détail dans le document «01_gnu-assembler.pdf»



■ Un symbole

- ▶ Débute généralement par une lettre ; peut contenir aussi des chiffres et des caractères spéciaux (._\$) mais pas d'espaces
- ▶ Est «case sensitive»
- ▶ Ne connaît pas de limites de taille (tous les caractères significatifs)
- ▶ Ne peut être défini qu'une fois dans un programme
- ▶ Peut représenter une valeur arbitraire, laquelle est écrite après le signe '=' suivant le symbole

■ Un label/une étiquette

- ▶ Symbole suivi immédiatement par deux points ':'
- ▶ Représente la valeur courante de la place mémoire
 - ◇ adresse d'une donnée en mémoire
 - ◇ adresse d'une instruction dans le code

■ Exemples...

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



■ L'assembleur connaît 3 types de constantes

▶ Nombres entiers

- ◇ Décimal : suite de chiffres entre 0 et 9, mais ne débutant pas par un 0
- ◇ Binaire : 0b ou 0B suivit d'une suite de 0 et 1
- ◇ Octal : 0 suivit par une suite de chiffres entre 0 et 7
- ◇ Hexa : 0x ou 0X suivit par une suite de chiffres entre 0 et 9 et a et f ou A et F

▶ Caractères

- ◇ Caractère ascii unique écrit soit avec un apostrophe suivie du caractère (p. ex. 'a), soit avec un caractère entre deux apostrophes (p. ex. 'a')
- ◇ Pour écrire le caractère backslash, on doit utiliser un 2^e backslash, soit '\\

▶ Strings ou chaînes de caractères

- ◇ Un string ascii est écrit simplement en plaçant une chaîne de caractères entre deux guillemets, p.ex. "hello world!\n"
- ◇ Les caractères d'échappement peuvent également être utilisés, p. ex. \b, \f, \n, \r, \t, \\, \" ou une valeur octale \000 ou hexa \x1a



- Il existe deux possibilités d'insérer des commentaires dans un programme assembleur
 - ▶ La forme C avec `/*` suivit d'un `*/`
 - ◇ Ne peut pas être imbriqué
 - ◇ Peut être placé n'importe où dans le programme
 - ◇ Peut être placé sur plusieurs lignes
 - ▶ Le commentaire de ligne. Il débute soit avec le caractère `'@'` ou soit avec `/*` (commentaire de ligne en C) et se termine à la fin de la ligne
 - ▶ Exemples...

```
/* ce commentaire, écrit sur plusieurs lignes,  
 * est tout à fait valide.  
 */  
  
mov r0, #78 @ commentaire de ligne  
mov r1, #59 // également un commentaire de ligne
```



- Dans une instruction assembleur, le champ opérande peut contenir une expression
- Les expressions couramment utilisées sont l'addition (+), la soustraction (-), la multiplication (*), la division entière (/), le reste (%) et les décalages (<<, >>)
- Exemples...

```
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
```




- Six types de directives sont proposés pour le stockage de données
 - ▶ `.byte <expressions>` → définition d'une série de mots de 8-bit
 - ▶ `.hword | .short <expressions>` → définition d'une série de mots de 16-bit
 - ▶ `.word | .long <expressions>` → définition d'une série de mots de 32-bit
 - ▶ `.ascii <string>` → définition d'un string
 - ▶ `.asciz <string>` → définition d'un string terminé par un 0
 - ▶ `.space <number_of_bytes> [, <fill>]` → définition d'un espace mémoire
- Ces *pseudo-opérations* évaluent leurs arguments lors de la phase d'assemblage et placent le résultat en mémoire lors du chargement du programme sur la cible
- Exemples...

```
Long : .long 0x12345678 //définit 1 constante de 32 bits
Short : .short 0xabcd //définit 1 constante de 16 bits
Byte : .byte 15,35,12,0 //définit 4 constantes de 8 bits
msg : .asciz "Hello World" //définit un string terminé par un 0
table : .space 3*4 //réserve 3 longs (3*4*8 bits)
list : .space 100,0xff //remplit 100 bytes avec 255
```



- Les assembleurs disposent généralement d'un grand nombre de directives permettant le développement de programmes complexes. Ci-dessous quelques-unes des plus utiles
 - ▶ `.global symbol {, symbol}`
→ permet de rendre accessible le symbole (label) pour le linker
 - ▶ `.align expression`
→ permet d'aligner des données sur un certain nombre de bytes
 - ▶ `.include "file"`
→ permet d'inclure un fichier
 - ▶ `.if absolute expression, .else, .elseif, .endif`
→ permet un assemblage conditionnel



- Les assembleurs modernes sont capables de regrouper des parties de code et/ou de données de même qualité dans des sections distinctes

- Elles sont définies de la manière suivante

`.section <nom_de_la_section>`

- Les 4 plus courantes sont

- ▶ `.section .text` ou `.text` → section pour le code
- ▶ `.section .data` ou `.data` → section pour les données initialisées
- ▶ `.section .bss` ou `.bss` → section pour les données mises à zéro
- ▶ `.section .rodata` → section pour les constantes



Exemple de code

```
/** <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  
next :  str     r5, [r4]  
        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
```