

Assembleur ARM

Directives d'assemblage
Etiquettes
Pseudo-instructions

Directives d'assemblage...



- Un fichier source assembleur n'est pas composé que d'instructions en langage assembleur
- Les **directives d'assemblage** sont des commandes ou des indications données au programme d'assemblage pour inclure des fichiers, définir des constantes, réserver des espaces en mémoire (données)
- Les **directives d'assemblage** sont toutes **préfixées par un point**

Directives d'assemblage...

- Inclure un fichier

```
.INCLUDE "fichier.s"
```

- Définir une constante symbolique
(équivalent au #define du C)

```
.EQU      MACONSTANTE, 0x0200
```

Directives d'assemblage...

- Réserver des emplacement pour des données (correspond aux variables locales du C)

```
.BYTE 56, 'a', -12, 255
```

```
.HWORD 0xB200, 0
```

```
.WORD 0x06000000
```

```
.FILL 8, 2, 0x0102
```

```
.ASCIZ "une chaine"
```

```
.ALIGN
```

Etiquettes



- Le programme d'assemblage décide des emplacements mémoire des données réservées et des instructions
- Il faut un système de référence symbolique pour désigner les emplacements mémoire
- Les étiquettes (*labels*) sont synonymes de l'adresse correspondant à leur emplacement
- Ce sont des identifiants **uniques** postfixés par un deux-point

Etiquettes

- Exemple de définition d'une étiquette :

MonEtiquette:

.WORD 0x89ABCDEF

.WORD 0xFFFFFFFF

- Si le 1er mot réservé est placé en mémoire par l'assemblage à l'adresse **0x03001A00** alors **MonEtiquette** est synonyme de **0x03001A00**
- Le second mot est à l'adresse **0x03001A04** car le placement se fait à des adresse consécutives

Variables en mémoire

- Le plus souvent pour avoir une variable en assembleur on utilise un registre...
- Quand ce n'est pas possible (pas assez de registres...) on utilise un stockage mémoire :

@ Une variable mémoire 32 bits

@ initialisée avec la valeur 10

@ *int MaVariable=10;*

MaVariable:

.WORD 10

Variables en mémoire



- Pour connaître la valeur stockée dans cette variable en mémoire il faut la charger dans un registre (ici r0 prend la valeur 10)

```
MaVariable:                @ int MaVariable=10;  
    .WORD 10
```

```
DebutCode:
```

```
...
```

```
LDR r0, MaVariable @ r0=MaVariable;
```

```
...
```

Variables en mémoire



- Pour écraser la valeur stockée dans cette variable en mémoire il faut l'écrire depuis un registre (ici `MaVariable` prend la valeur 20)

```
MaVariable:                @ int MaVariable=10;  
    .WORD 10
```

DebutCode:

...

```
MOV  r0, #20                @ r0=20;  
STR  r0, MaVariable         @ MaVariable=r0;
```

...

Variables en mémoire



- Pour modifier la valeur stockée il faut charger dans un registre, modifier le registre, écrire depuis le registre (ici MaVariable vaudra 11)

```
MaVariable:                @ int MaVariable=10;
    .WORD 10
```

DebutCode :

```
...
LDR  r0, MaVariable    @ r0=MaVariable;
ADD  r0, r0, #1        @ r0=r0+1;
STR  r0, MaVariable    @ MaVariable=r0;
...
```

Pseudo instructions LDR/STR

- Le processeur n'est pas capable de comprendre des identifiants symboliques comme "MaVariable"
- Les seules références mémoires à l'exécution sont données par des adresses (pointeurs)
- Le programme d'assemblage (à la compilation) transforme les identifiants symboliques en adresses concrètes :

"MaVariable" devient

"contenu de l'adresse 0x0300008C"

Pseudo instructions LDR/STR

- Cependant on ne peut faire tenir une adresse 32bits sur une instruction de 32bits...
- Il y a un système d'**adressage relatif** qui se fait sur 12 bits à partir du registre r15 (alias pc)

@ adr. variable 20 octets avant adr. instruction
LDR r0, [pc, #-20]

- Cette instruction **réelle** est écrite automatiquement par l'assembleur à partir de la **pseudo-instruction LDR r0, MaVariable**

Mémoire, LDR et constantes



- Une autre utilisation de LDR très utile permet (enfin !) de charger des valeurs constantes arbitraires dans un registre

```
LDR r0 ,=3141593 @ r0=3141593;
```

- Correspond également à une pseudo-instruction
- L'instruction réelle charge la constante à partir d'un autre emplacement mémoire réservé et initialisé automatiquement à la compilation

Mémoire, LDR et constantes

- Si valeur indiquée compatible avec <immed_8r>

LDR r0 , =20 → MOV r0 , #20

- Sinon

LDR r0 , =3141593



LDR r0 , [pc , #0x???

...

... @ Fin des instructions

.WORD 3141593

Déplacement relatif :
pointe sur l'adresse
où est stockée la constante

Variables en mémoire

- L'etiquette est synonyme de l'adresse (pointeur) ceci permet de charger une adresse dans un registre pour, par exemple, gérer un **tableau**

```
@ int tab[3]={10,15,20};
```

```
tab: .WORD 10,15,20
```

```
@ tab[1]=tab[0];
```

```
LDR r0,=tab
```

```
LDR r1,[r0]
```

```
ADD r0,r0,#4
```

```
STR r1,[r0]
```

Assembleur ARM

Exécution conditionnelle

Exécution conditionnelle



- Par défaut les opérations usuelles de traitements de données laissent les indicateurs de code condition NZCV inchangés
- En postfixant ces instructions par S (*Set condition flags*) le résultat de l'opération met à jour les codes conditions.

```
MOV    r1, #1
SUBS   r0, r1, #1
```

-> N=0 Z=1 C=1 V=0

Exécution conditionnelle



- La plupart des instructions assembleur ARM peuvent être exécutées conditionnellement à l'état des indicateurs NZCV
- Postfixe EQ : instruction exécutée si Z=1
- Postfixe NE : instruction exécutée si Z=0
- Voir support de TD pour les autres postfixes
- Exemple : si bit 5 de r1 est à 1 incrémenter r3

ANDS **r0 , r1 , #0x20** @ **Masquage**

ADDNE **r3 , r3 , #1** @ **Incrément**

Exécution conditionnelle

- On peut souhaiter réaliser une opération logique de masquage ou une comparaison entre valeurs uniquement pour modifier les indicateurs NZCV
- C'est le rôle des instructions :
 - **TST** *rn*, <Oprnd2> : *rn* AND Oprnd2
 - **TEQ** *rn*, <Oprnd2> : *rn* EOR Oprnd2
 - **CMP** *rn*, <Oprnd2> : *rn* - Oprnd2
 - **CMN** *rn*, <Oprnd2> : *rn* + Oprnd2
- Pas besoin de registre dest. pour le résultat

Exécution conditionnelle

- Exemple : si bit 5 de r1 est à 1 incrémenter r3

```
TST      r1 , #0x20      @ Masquage
ADDNE    r3 , r3 , #1     @ Incrément
```

- Exemple : si r1 divisible par 16 incrémenter r3

```
TST      r1 , #0x0F      @ Masquage
ADDEQ    r3 , r3 , #1     @ Incrément
```

Exécution conditionnelle



- L'instruction CMP permet de comparer les valeurs par soustraction
- Un tableau du support de TD résume les relations possibles et les codes correspondants

Comparaison de deux valeurs : `CMP r0, r1`
code conditionnel à utiliser en fonction du rapport des valeurs

condition	non signé	signé
<code>r0==r1</code>	EQ	EQ
<code>r0!=r1</code>	NE	NE
<code>r0>r1</code>	HI	GT
<code>r0>=r1</code>	CS	GE
<code>r0<r1</code>	CC	LT
<code>r0<=r1</code>	LS	LE

Exécution conditionnelle

- Exemple : $r2 \leftarrow \max(r0, r1)$

CMP	r0, r1	@	Comparaison r0-r1
MOVCS	r2, r0	@	si r0>=r1 : r2<-r0
MOVCC	r2, r1	@	si r0<r1 : r2<-r1

Assembleur ARM

Branchements,
Tests et Boucles

Branchements




- Le registre r15 (alias pc) est le compteur ordinal, il joue un rôle particulier car il désigne l'adresse de la prochaine instruction à exécuter.
- Il est ainsi possible de modifier le déroulement séquentiel par défaut du programme en mettant l'adresse d'un autre point du code dans r15
- On parle alors de **branchement** de l'exécution, ou de **saut**.

Branchements

- Exemple : sauter par dessus du code

```
MOV    r0, #0
LDR    r1, =EnAvant
MOV    r15, r1
ADD    r0, r0, #1
ADD    r0, r0, #1
EnAvant: ADD    r0, r0, #1
        ADD    r0, r0, #1
```



-> à la fin r0=2

Branchements



- Une instruction spéciale **B** (*Branch*) réalise directement cette modification du compteur ordinal pour sauter à l'instruction associée à une étiquette

```
MOV    r0 , #0
B      EnAvant    @Branchement
ADD    r0 , r0 , #1
ADD    r0 , r0 , #1
EnAvant: ADD    r0 , r0 , #1
ADD    r0 , r0 , #1
```

Branchements conditionnels

- Les **branchements conditionnels** permettent la réalisation des structures de contrôle
- Blocs exécutés plusieurs fois, **boucles** :
 - REPETER n fois ...
 - POUR cpt DE val1 A val2 ...
 - TANT QUE (condition) ...
 - FAIRE ... TANT QUE (condition)
- Blocs exécutés selon resultat d'un **test** :
 - SI (condition) ...
 - SI (condition) ... SINON ...

Branchements Conditionnels

- L'instruction **BNE** (*Branch on Not Equal*) permet de réaliser une boucle REPETER n fois. Exemple effacer l'écran pixel par pixel.

```
LDR    r2, =0x0
LDR    r1, =0x06000000
LDR    r0, =240*160
```

EffacerSuivant:

```
si r0≠0  STRH    r2, [r1]
          ADD     r1, r1, #2
          SUBS   r0, r0, #1
          BNE    EffacerSuivant
si r0=0  ...
```

Boucles



- REPETER n fois

```

                LDR        r0, =n
Repeter:
                ...
si r0≠0        ...
                SUBS      r0, r0, #1
                BNE       Repeter
                ...
si r0=0        ...
                @ Après la boucle
                @ Instructions du
                @ corps de boucle
```

Boucles

- FAIRE ... TANT QUE (condition)

FaireTantQue:

si condition	...	@ Corps de boucle
vraie	...	
	...	@ Mettre ZNCV à jour
	B{cond}	FaireTantQue
sinon	...	@ Après la boucle

Boucles

- POUR cpt DE val1 A val2

```
                MOV        r0, #val1
BouclePour:
                CMP        r0, #val2
si r0 ≥ val2 →  BCS        SortiePour
                ...        @ Corps de boucle
                ADD        r0, r0, #1
                B          BouclePour
SortiePour:
                ...        @ Après la boucle
```

Tests



- SI (condition) code1
- On saute par dessus code1 si la condition n'est **pas** respectée
- Exemple : Si $r0 \leq r1$ alors exécuter code1

```
si r0>r1    CMP    r0 , r1
            BHI    ApresSi
            ...
            ...
ApresSi:
            ...
```

@ Code1 exécuté
@ seulement si $r0 \leq r1$
@ Suite du programme

Tests



- SI (condition) code1 SINON code2
- Comme le SI mais à la fin de code1 on branche par dessus code2

```

                                CMP        r0, r1
si r0>r1 BHI        Sinon
                                ...
                                B          @ Code1 : si r0≤r1
Sinon:                          @ Code2 : si r0>r1
                                ...
ApresSiSinon:                   @ Suite ...
                                ...
```

Tests et Boucles

- L'assembleur autorise les branchements inconditionnels ou conditionnels de n'importe quel endroit du code à n'importe quel autre
- Contraste avec l'obligation des structures de contrôle du code C (blocs de code imbriqués)
- Ne pas abuser de cette possibilité : les réflexes acquis avec le C restent nécessaires en assembleur, il faut toujours **structurer** pour éviter l'effet spaghetti...

Dépassements, débordements

- Lors d'une opération arithmétique type addition,
 - L'indicateur C indique un problème non-signé
 - L'indicateur V indique un problème signé
- Ces dépassements et débordements peuvent avoir lieu également en C sur des variables int
- En général on ne testera pas la validité des calculs : on fera en sorte que les valeurs utilisées par le programme tiennent sur des intervalles compatibles avec le format d'accueil

Dépassements, débordements

- En pratique sur un format n bits, tant que les valeurs absolues restent inférieures à 2^{n-2} il est possible d'utiliser additions et soustractions sans risque de dépassement ou débordement
- Pour 32 bits (variable int ou registres) l'ordre de grandeur est de 10^9