Puisse ce document vous être utile.
Bonne lecture,
Chevalier Licorne.

Nous utilisons l'assembleur ARM 32 bits. Il s'agit d'effectuer la première étape de la compilation, c'est à dire d'écrire le code d'un programme en 'C' directement en assembleur ARM, en faisant comme si chaque instruction 'C' était indépendante de la précédente.
L'assembleur ARM 32 bits permet de manipuler 16 registres de 32 bits dans le processeur ainsi que d'écrire dans la RAM.
Pour lire ou écrire dans la RAM il faut utiliser l'adresse des variables dont on dispose.
Pour manipuler des données il faut le faire dans les registres du processeur.
Les registres sont 'r0', 'r1', …, 'r15'.

Partie 1 : Manipulations basiques, additions, soustraction, if then

On considère que « main » n'est pas une fonction et qu'elle n'a pas de paramètres.
« ; » permet d'indiquer que le reste de la ligne est un commentaire.

Pour déclarer des variables en début de programme on utilise toujours la même méthode pour le moment : pour « int a, b, c ; » voici le code assembleur ARM :

```
L1 :
.word a
.word b
.word c
.comm a, 4, 4 ; le premier 4 correspond à la taille en octets de la variable
.comm b, 4, 4 ; pour un tableau de 4 nombres on aurait : b, 16, 4
.comm c, 4, 4 ; la valeur du dernier 4 ne doit pas être modifiée
```

« mov » est une commande interne au processeur. Elle permet de :
-mettre une constante dans un registre :
mov r0, #10 ; met la valeur '10' dans le registre 'r0'
-copier le contenu d'un registre dans un autre :
mov r1, r0 ; met dans le registre 'r1' la valeur contenue dans 'r0', c'est à dire le nombre '10'

« ldr » permet de mettre dans un registre la valeur contenue à une adresse :
-ldr r0, L1+4 ; met dans 'r0' la valeur enregistrée dans la RAM à l'adresse de 'L1+4'
-ldr r0, [r1] ; met dans 'r0' la valeur enregistré dans la RAM à l'adresse contenue dans 'r1'

« str » fonctionne exactement à l'inverse de 'ldr'. Cette commande permet d'écrire dans la RAM :
-str r0, [r1] ; met la valeur contenue dans 'r0' à l'adresse contenue dans 'r1' qui se trouve dans la RAM

« add » permet d'effectuer une addition sur des valeurs contenues dans des registres :
add r0, r1, r2 ; met dans 'r0' le résultat de la somme des valeurs contenues dans 'r1' et 'r2'
On peut également faire :
add r0, r0, r1 ; (valeur de 'r0' + valeur de 'r1') va dans 'r0'
Ainsi que :
add r0, r0, #10 ; (valeur de 'r0' + 10) va dans 'r0'

« sub » permet d'effectuer une soustraction sur des valeurs contenues dans des registres :
-sub r0, r1, r2 ; met dans le registre 'r0' le résultat de la soustraction de la valeur contenues dans 'r1' moins celle contenue dans 'r2'
On peut également faire :
sub r0, r0, r1 ; (valeur de 'r0' - valeur de 'r1') va dans 'r0'
Ainsi que :
sub r0, r0, #10 ; (valeur de 'r0' - 10) va dans 'r0'

De même avec la commande « mul » qui multiplie.

Pour mettre une valeur dans une variable déclarée nous allons utiliser les 'a, b, c' du début. Elles se trouvent dans la RAM aux adresses 'L1' pour 'a', 'L1+4' pour 'b' et 'L1+8' pour 'c' :
-Mettre 10 dans 'a' :
mov r0, #10 ; met la valeur 10 dans le registre 'r0'
ldr r1, L1 ; met l'adresse de 'a' dans le registre 'r1'
str r0, [r1] ; met la valeur contenue dans 'r0' à l'adresse contenue dans 'r1'

-Mettre 12 dans 'b' :
mov r0, #12
ldr r1, L1+4 ; dans le cas d'un 'ldr' on peut écrire '+nombre'
str r0, [r1]

-Mettre a+b dans a :
ldr r0, L1 ; met l'adresse de 'a' dans 'r0'
ldr r1, [r0] ; met la valeur de 'a' dans r1
ldr r0, L1+4 ; met l'adresse de 'b' dans 'r0'
ldr r2, [r0] ; met la valeur de b dans 'r2'
add r1, r1, r2 ; met la valeur de 'a+b' dans 'r1'
ldr r0, L1 ; met l'adresse de 'a' dans 'r0'
str r1, [r0] ; met la valeur de 'a+b' à l'adresse de 'a'

Principe de non-optimisation du code :
pour faire 'a=a+1' puis 'a=a-2' on pourrait optimiser le code de la façon suivante :
ldr r0, L1 ; on met l'adresse de 'a' dans 'r0'
ldr r1, [r0] ; 'a' est dans 'r1'
add r1, r1, #1 ; 'a=a+1'
sub r1, r1, #2 ; 'a=a-2'
str r1, [r0] ; on remet dans 'a' (qui se trouve dans la RAM) la valeur de a+1-2
Cependant dans ce cas nous ne faisons pas la première étape du compilateur, or c'est ce qu'on nous demande de faire. Nous devons donc traiter ces deux instructions 'C' indépendamment :
ldr r0, L1
ldr r1, [r0]
add r1, r1, #1
str r1, [r0] ; fin de 'a=a+1', la valeur dans la RAM a été mise à jour
ldr r0, L1 ; on recommence à mettre l'adresse de 'a' dans 'r0' même si on sait qu'elle y était déjà. En effet il ne s'agit plus de la même instruction 'C'. Le compilateur se chargera plus tard d'optimiser tout ça.
ldr r1, [r0]
sub r1, r1, #2
str r1, [r0] ; fin de 'a=a-2'

Les étiquettes se placent devant une ligne de code, elles permettent de placer des ‘points de repère’ dans celui-ci. Cela permet ensuite d’y accéder en utilisant le nom de l’étiquette :
toto : mov r0, #10 ; le nom de l’étiquette est ‘toto’

« bra » permet de se rendre à une étiquette sans condition :
toto : mov r0, #10
bra toto ; ici nous avons une boucle infinie qui remet ‘10’ dans ‘r0’ en boucle

« cmp » (CoMPare) est une commande qui permet de comparer les valeurs contenues dans deux registres :
cmp r0, r1 ; va calculer la différence de ‘r0’ moins ‘r1’, comparer de toutes les façons possibles le résultat par rapport à 0 et stocker en mémoire ces résultats.
On peut aussi utiliser :
cmp r0, #10
Pour utiliser cette comparaison nous utilisons les commandes suivantes :

« beq » (Branch if EQual) va à une étiquette si les valeurs comparées sont les mêmes :
beq toto ; va à ‘toto’ si ‘r0==r1’

« bne » (Branch if Not Equal) va à une étiquette si les valeurs comparées sont différentes :
bne toto ; va à ‘toto’ si ‘r0!=r1’

« bgt » (Branch if Greater Than) va à une étiquette si la valeur du premier registre est strictement plus grande que le second :
bgt toto ; va à ‘toto’ si ‘r0>r1’

« bge » (Branch if Greater or Equal) va à une étiquette si la valeur de premier registre est plus grande ou égale au second :
bge toto ; va à ‘toto’ si ‘r0>=r1’

« blt » (Branch if Lower Than) va à une étiquette si la valeur de premier registre est strictement plus petite que le second :
blt toto ; va à ‘toto’ si ‘r0<r1’

« ble » (Branch if Lower or Equal) va à une étiquette si la valeur de premier registre est plus petite ou égale au second :
ble toto ; va à ‘toto’ si ‘r0<=r1’

Traduction du ‘if then’ :
On commence par faire la commande ‘cmp’ avec les registres à comparer, ou le registre à comparer à une nombre.
Ensuite on cherche pour quelle condition il ne faut pas exécuter le ‘then’ (la condition de sortie). Par exemple pour : ‘if a<5’ la condition de sortie est ‘a>=5’. Ensuite on utilisera les étiquettes. Cela aura la forme suivante :
cmp r0, #5 ; on va dire que ‘a’ est dans ‘r0’
bge fin_if ; la condition de sortie est ‘>=’
traitement du then
…
fin_if : ; notre étiquette de sortie du ‘if’
traitement après le if …

Partie 2 : if then else, for, while

Traduction du ‘if then else’ :
On utilise presque la même forme, avec un peu plus d’étiquettes :
cmp r0, #5 ; on va dire que ‘a’ est dans ‘r0’
bge else ; la condition du ‘else’ est ‘>=’
traitement du then…
bra fin_if ; car on ne doit pas exécuter le ‘else’
else :
traitement du else…
fin_if : ; notre étiquette de sortie du ‘if’
traitement après le if …

Traduction du for :
On va se baser sur ‘for (i=0 ; i<10 ; i++) traitement ;’ :
mov r0, #0
ldr r1, L1+4 ; disons que ‘i’ est à l’adresse de ‘L1+4’
str r0, [r1] ; on initialise ‘i=0’
debut_for : ldr r0, L1+4 ; on reprend l’adresse de ‘i’
ldr r1, [r0] ; on met la valeur de ‘i’ dans ‘r1’
cmp r1, #10
bge fin_for ; la condition de sortie est ‘i>=10’
traitement interne au ‘for’…
ldr r0, L1+4 ; on remet l’adresse de ‘i’ dans ‘r0’
ldr r1, [r0] ; on met la valeur de ‘i’ dans ‘r1’
add r1, r1, #1 ; on lui ajoute 1 pour faire i=i++
str r1, [r0] ; on enregistre le résultat dans la RAM à l’adresse de ‘i’
bra debut_for
fin_for :
traitement après le ‘for’…

Traduction du while :
On va se baser sur ‘while (i<10) traitement ;’ :
debut_while : ldr r0, L1+4 ; disons que ‘i’ est à l’adresse de ‘L1+4’
ldr r1, [r0] ; on met la valeur de ‘i’ dans ‘r1’
cmp r1, #10
bge fin_while ; la condition de sortie est ‘i>=10’
traitement interne au ‘while’…
bra debut_while
fin_while :
traitement après le ‘while’...

Partie 3 : tableaux, for imbriqué

Pour initialiser un tableau ‘int t[4] ;’ :
L1 :
.word t
.comm t, 16, 4 ; 16 car 4 nombres de 4 octets

Pour accéder au tableau : ‘t[0]=3 ; t[2]=21’ :
mov r0, #3
ldr r1, L1 ; adresse de t[0]
str r0, [r1] ; on met la valeur de ‘r0’ à l’adresse qui est dans ‘r1’ : t[0]=3
mov r0, #21
ldr r1, L1
str r0, [r1, #8] ; on met la valeur de ‘r0’ à l’adresse qui est dans ‘r1’ + 8 pour accéder à la troisième valeur du tableau : t[2]=21

For imbriqué : il suffit de faire un for, puis un autre for avec une variable d’incrémentation différente du premier et il faut placer ce nouveau for dans la partie ‘traitement...’ du premier for.
Par exemple avec :
‘int i, j ;
for (i=0 ; i<5 ; i++) {
    traitement1…
    for (j=0 ; j<10 ; j++) {
        traitement2…
    }
}’

On obtient :
mov r0, #0
ldr r1, L1
str r0, [r1] ; fin de i=0
debut_for1 : ldr r0, L1 ; on reprend l’adresse de ‘i’
ldr r1, [r0] ; on met la valeur de ‘i’ dans ‘r1’
cmp r1, #5
bge fin_for1 ; la condition de sortie est ‘i>=5’
traitement interne au ‘for 1’…
ldr r0, L1 ; on remet l’adresse de ‘i’ dans ‘r0’
ldr r1, [r0] ; on met la valeur de ‘i’ dans ‘r1’
add r1, r1, #1 ; on lui ajoute 1 pour faire i=i++
str r1, [r0] ; on enregistre le résultat dans la RAM à l’adresse de ‘i’
bra debut_for1
fin_for1 :
traitement après le ‘for 1’…

For imbriqué :

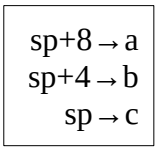
mov r0, #0
ldr r1, L1+4
str r0, [r1] ; fin de ‘j=0’
debut_for2 : ldr r0, L1+4
ldr r1, [r0]
cmp r1, #10
bge fin_for2
traitement interne au ‘for 2’…
ldr r0, L1+4
ldr r1, [r1]
add r1, r1, #1
str r1, [r0]
bra debut_for2
fin_for2 :
traitement après le ‘for 2’…

Partie 4 : variables locales, fonctions, pointeurs

Pour utiliser des variables locales ainsi que des fonctions, il faut utiliser l'empilement des environnements.
Pour cela on utilise le pointeur 'sp' dont le fonctionnement est le même que 'L1'.
Création d'un environnement pour la fonction main :

```
'int main() {
        int a, b, c ;
        a=10 ;
        b=20 ;
        c=a+10 ;
        return 0 ;
}'
```

Ce code donne :



stmfd sp!, {lr} ; il faut toujours remplacer le 'main(){' par cette ligne
sub sp, sp, #12 ; on réserve la place pour 3 variables de 4 octets dans la pile →
mov r0, #12
str r0, [sp, #8] ; on met 8 à l'adresse de 'sp' + 8 c'est à dire 'a'
mov r0, 20
str r0, [sp, #4] ; de même avec 'b=20'
ldr r0, [sp, #8] ; on met 'a' dans 'r0'
mov r1, #10
add r0, r0, r1 ; on met 'a+10' dans 'r0'
str r0, [sp] ; on met 'a+10' à l'adresse de 'c'
mov r0, #0 ; on met la valeur du 'return' dans 'r0'
add sp, sp, #12 ; comme au début mais avec un 'sub'
ldmfd sp!, {lr} ; la même ligne qu'au début en changeant 'stmfd' par 'ldmfd'
bx lr ; tout à la fin
On note que dans le '{lr}' on doit mettre 'lr' ainsi que tous les registres qui ont été modifiés et qu'on ne retourne pas. Pour la fonction 'main' cela ne change pas : on a toujours 'sp!, {lr}'

Création d'un environnement pour une autre fonction :

```
int fonction(int a, int b) {
        int c=10 ;
        return a+b+c
```

fonction : on commence par mettre une étiquettes
stmfd sp!, { lr} ; puis on laisse un peu de place car il faut attendre la fin pour savoir quels registres il faut mettre dans le stmfd
sub sp, sp, #12 ; disons que je vais entrer trois paramètres dans ma fonction, ils seront dans 'r0'et 'r1' et **il va falloir les sauvegarder dans la pile des registres** avant de commencer le traitement
str r0, [sp, #8] ; met le premier dans la pile
str r1, [sp, #4] ; met le deuxième dans la pile
mov r2, #10 ; on vient de modifier un registre, il faudra le mettre dans le { lr} → { r2, lr}
str r2, [sp] ; met 'c=10' dans la pile
add r0, r0, r1 ; modifie 'r0' mais comme on le retourne il ne faut pas le signaler
add r0, r0, r2 ; met 'a+b+c' dans 'r0' : 'return a+b+c'
add sp, sp, #12
ldmfd sp!, {r2, lr} ; il faudra mettre à jour cette ligne en début de programme
bx lr

« bl » est la commande qui permet d'appeler une fonction :
bl fonction ; appelle la fonction 'fonction'