Architecture des Ordinateurs

Chapitre 5 : Assembleur des processeurs ARM

Jacques Supcik, Daniel Gachet, Luca Haab

2022-11-25

Table des matières

5 Assembleur des processeurs ARM

La conception et réalisation de systèmes embarqués et de systèmes sur puce (SoC) a énormément évolué depuis ces débuts dans les années 1970. L'arrivée sur le marché de µP et µC excessivement performant (capacité mémoire, puissance de calcul, variété des périphériques d'entrées/sorties, etc.) a généralisé le développement d'applications mettant en oeuvre des systèmes d'exploitation riches, tel GNU/Linux, ou temps réels, tels FreeRTOS ou MbedOS. Pour des raisons de performances, ces applications sont principalement développées avec des langages de programmation compilés. C/C++ est actuellement le langage de prédilection, bien qu'une tendance se dessine vers des langages plus sûrs et plus robustes, tel que Rust.

Le code machine généré par les compilateurs de dernière génération est d'une qualité telle, qu'il devient difficile, voire impossible, de les égaler avec une programmation en assembleur. En effet, ces compilateurs ont une connaissance très pointue des processeurs et de leur architecture interne, qu'ils mettent à profit pour produire un code excessivement performant. Cependant, la réalisation de certaines opérations très proche du processeur et de ses unités de traitement internes doivent encore se programmer avec des instructions assembleur non disponibles dans les langages de programmation évolués.

L'assembleur est un langage de programmation très proche du code machine binaire directement exécutable par le processeur. Il existe généralement une relation univoque entre le code machine et les mnémoniques des instructions assembleur. Le langage est naturellement enrichi d'éléments facilitant l'écriture, la structuration et la modularisation des programmes, tels que la déclaration de sections de texte pour le code et les instructions, de sections de données pour les variables et les constantes, de symboles, macros ou d'informations facilitant le débogage.

Sans autres indications, les exemples en langage assembleur de ce chapitre sont réalisés pour des µP/µC ARMv7 à 32 bits avec le jeu d'instructions Thumb-2.

5.1 Rappel

Selon leur profil, les processeurs ARM supportent des jeux d'instructions différents. Les µP du profil A et R sont capables d'exécuter des instructions des trois jeux d'instructions ARM, Thumb et Thumb-2, tandis que les µC du profil M ne supportent que les jeux d'instructions Thumb et Thumb-2.

Avec les jeux d'instructions ARM (codé sur 32 bits) et Thumb-2 (utilisant un codage mixte 16 bits et 32 bits), toutes les instructions de traitement de données peuvent utiliser, sans distinction, l'ensemble des 16 registres (R0 à R15) du CPU à usage général et dédié (figure 5.1). Par contre le jeu d'instructions Thumb (codé sur 16 bits) ne peut utiliser que les 8 premiers registres à usage général (R0 à R7).

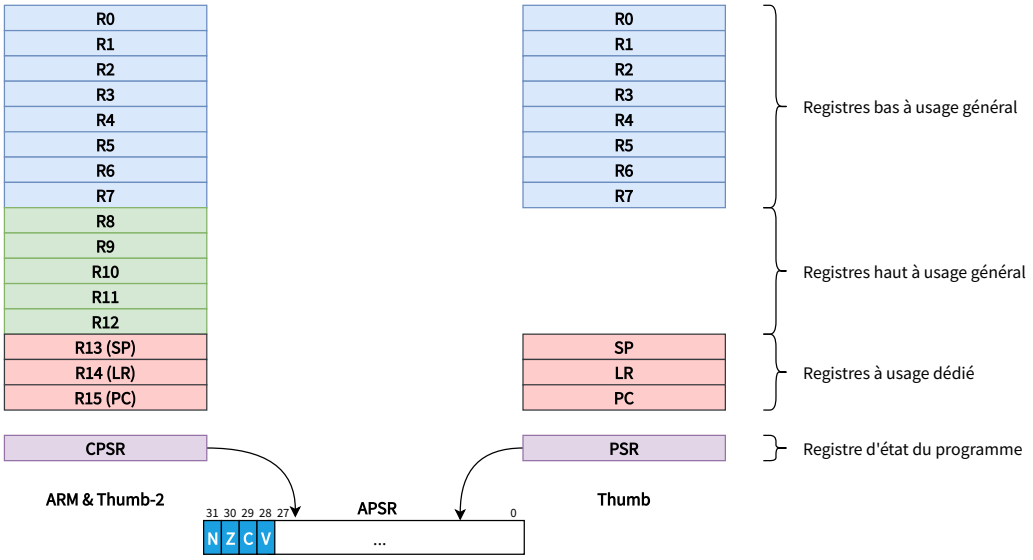


FIGURE 5.1 – Visibilité des registres

Tous les registres ont une taille fixe de 32 bits. Toutes les opérations arithmétiques et logiques ne s'exercent que sur ces 32 bits. Ce n'est qu'au chargement des données dans les registres du CPU que leur taille est adaptée, si nécessaire.

Lors de la réalisation d'un programme en langage assembleur, 4 registres requièrent une attention toute particulière et l'emploi d'instructions spécifiques :

- R13/SP (*Stack Pointer*) pointant sur le sommet de la pile (*Full Descending Stack*)
- R14/LR (*Link Register*) contenant l'adresse de retour lors d'appel de fonction
- R15/PC (*Program Compteur*) pointant sur la prochaine instruction à exécuter,
- APSR (*Application Program Status Register*) contenant les quatre fanions de conditions (*flags*) résultant d'opérations arithmétiques et logiques

Lors de branchements indirects, le bit 0 de l'adresse indique au processeur le jeu d'instructions à utiliser, 0 pour ARM et 1 pour Thumb/Thumb-2. Avec les µP ARM Cortex-M, il est dès lors impératif que ce bit soit toujours à 1.

5.2 Processus de développement

Lorsque tout le processus s'effectue sur le même ordinateur, on parle de développement natif (*Native Development*). Par contre, lorsque l'édition de texte, l'assemblage et l'édition de liens s'effectuent sur une machine différente, la machine hôte (*host*) de celle qui exécute le programme, la machine cible (*target*), on parle de développement croisé (*Cross Development*).

Le processus de développement en langage assembleur suit un processus classique et se réalise en 4 étapes (figure 5.2).

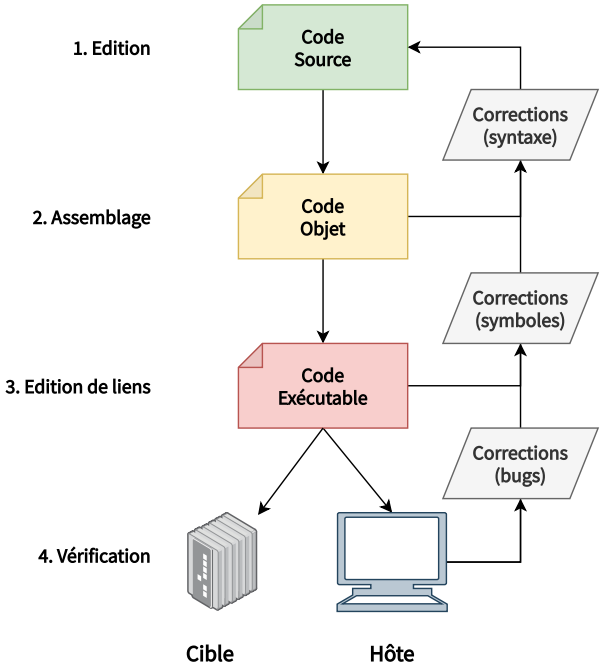


FIGURE 5.2 – Processus de développement

L'**édition** des fichiers source se réalise avec un éditeur de texte permettant de produire des fichiers de texte simple. Ces fichiers ne contiennent qu'une suite de caractères, généralement des caractères selon le standard ASCII (*American Standard Code for Information Interchange*).

L'**assemblage** consiste à transformer le code source afin d'obtenir le code machine sous forme binaire, nommé code objet (*Object Code*). Le code objet contient les instructions exécutables par le processeur

sous forme binaire ainsi qu'une série de symboles permettant de le lier à l'application finale et de le déboguer. Une phase de correction due à des erreurs d'écriture du code source, souvent des erreurs de syntaxe, est naturellement comprise dans cette partie du processus. Une fois assemblé, il est naturellement possible d'obtenir à nouveau le code source à partir du code objet. Cette opération inverse est nommée désassemblage.

L'**édition de liens** sert à lier les différents codes objet afin d'obtenir le code exécutable. Ce code est intimement lié à l'architecture de processeur ainsi qu'à sa réalisation et implémentation sur le système cible. Cette étape se divise en deux phases. La première doit résoudre les dépendances, les liens, entre les différents symboles (appel de fonction, adresses de variables globales, etc.). Si des symboles restent indéfinis, une phase de correction devra être entreprise. La deuxième phase sert à placer le code dans l'espace mémoire du processeur selon l'organisation de la cible. Avec l'utilisation de systèmes d'exploitation, cette phase peut également intervenir lors du chargement du code sur la cible.

La **vérification** est une étape essentielle dans la réalisation d'une application. Elle permet de garantir son bon fonctionnement, sa robustesse, sa fiabilité et finalement le respect du cahier des charges. Pour effectuer cette vérification, le code exécutable est chargé en mémoire sur le système cible, un système de test ou un simulateur. Des batteries de tests permettront ensuite de le vérifier de manière la plus exhaustive possible. Des corrections liées au mauvais fonctionnement du programme (*bugs*) peuvent intervenir à cette étape. Diverses techniques et outils existent pour localiser les erreurs. Le débogueur est un de ces outils.

La correction d'erreurs dues à une mauvaise conception ou design de l'application apparaissant lors de phases ultérieures peut être très complexe à identifier et coûter beaucoup d'énergie et d'argent ! Il est donc primordial d'effectuer un maximum de vérification lors du développement des applications. Une approche telle que CI/CD (*Continuous Integration / Continuous Deployment*) permet d'augmenter la fréquence de validation et de distribution des applications grâce à l'automatisation au niveau des étapes de développement des applications.

5.3 Structure d'un module d'assemblage

La conception de programme en langage assembleur s'effectue généralement sous une forme modulaire à l'aide de plusieurs fichiers (les modules). La décomposition de l'application en plusieurs modules permet d'en simplifier le développement, la validation et avant tout sa réutilisation. Cette technique permet également la réalisation d'applications utilisant différents langages de programmation, ceci pour autant que l'interface au niveau du code objet binaire (ABI - *Application Binary Interface*) soit identique.

5.3.1 Module

Les modules assembleur implémentent leur code dans un fichier avec l'extension "`.S`" (majuscule). Comme pour un code développé entièrement en C/C++, la déclaration des éléments publics passe par l'intermédiaire d'un fichier d'en-tête, un fichier "`.h`".

Un module assembleur se décompose généralement en sept sections distinctes :

1. En-tête avec le copyright, l'auteur, la date, la version, …
2. Inclusion de déclarations externes, si nécessaire
3. Déclaration des symboles ou expressions textuelles
4. Déclaration des constantes ou variables avec un accès en lecture uniquement
5. Déclaration des variables avec une valeur initiale non nulle
6. Déclaration des variables avec une valeur initiale nulle
7. Implémentation des fonctions ou routines

```
1  /*
2   * 1. header with copyright
3   */
4
5  // 2. Includes
6    .include "file"
7
8  // 3. Symboles (textual expressions)
9  #define MY_SYMBOL (12)
10
11 // 4. Declaration of constants
12   .section .rodata
13   .align 2
14 MY_CONSTANT:  .long 10
15
16 // 5. Declaration of variables with non-zero initial value
17   .section .data
18   .align 2
19 my_init_var:   .long 5
20
21 // 6. Declaration of variables with zero initial value
22   .section .bss
23   .align 2
24 my_zero_var:   .long 0
25
26 // 7. Function implementation
27   .section .text
28   .thumb
29   .syntax unified
30   .align 2
31
```

```
32    .global my_function
33    .type my_function, %function
34  my_function:
35    adds  r0, #10
36    movs  r3, #213
37    muls  r0, r3, r0
38    bx      lr
39    .size my_function, .-my_fonction
```

Le code assembleur s'écrit en une suite de lignes de code. Chaque ligne de code ne comprend qu'une seule instruction. L'indentation du code augmente grandement la lisibilité du code.

5.3.2 Commentaires

Les commentaires identifient des lignes de texte que l'assembleur doit ignorer. Il existe deux possibilités d'insérer des commentaires dans un module assembleur :

— Commentaire de bloc
— Commentaire de ligne

Le commentaire de bloc suit la même syntaxe que le code C/C++ :

— Débute par un "/*" et se termine par un "*/"
— Peut être placé n'importe où dans le programme
— Peut être placé sur plusieurs lignes
— Ne peut pas être imbriqué

Le commentaire de ligne débute avec un marqueur spécial et se termine à la fin de la ligne. Il existe plusieurs marqueurs selon la distribution de l'assembleur :

— "@" : caractère spécifique à l'assembleur GNU/AS
— ";" : caractère spécifique à l'assembleur ARM/AS (Keil)
— "//" : commentaire de ligne identique à C/C++

```
1    /* ce commentaire, écrit sur plusieurs lignes,
2     * est tout à fait valide.
3     */
4
5    mov r0,#78 @  commentaire de ligne à la GNU/AS
6    mov r2,#97 ;  commentaire de ligne à la ARM/AS
7    mov r1,#59 // commentaire de ligne à la C/C++
```

L'utilisation de la syntaxe C/C++ pour les commentaires de ligne est à privilégier, car la majorité des assembleurs la supporte.

5.3.3 Symboles

Tout comme en langage C/C++, les symboles permettent d'identifier les fonctions, les variables et les constantes. La visibilité d'un symbole est par défaut local au module dans lequel il est déclaré. Si sa visibilité doit être élargie à l'ensemble du programme, alors dans ce cas l'emploi de la directive "`.global symbol`" est nécessaire. Un symbole se déclare à l'aide d'un identifiant et se définit par une étiquette (*label*).

L'identifiant servant à la déclaration d'un symbole doit respecter certaines règles :

- Comme 1er caractère, une lettre ou un de ces caractères spéciaux : "`.`", "`_`", "`$`"
- Chiffres autorisés, mais pas comme premier caractère
- Espaces non autorisés
- Sensibilité à la casse (*case sensitive*)
- Pas de limites de taille (tous les caractères significatifs)
- Unique dans le module
- Unique dans le programme, si global

Une étiquette est un identifiant valide placé au début d'une nouvelle ligne et terminé par deux points ("`:`"). Cette étiquette représente une adresse mémoire. Elle permet de définir le symbole d'une donnée (variable ou constante) ou d'une fonction.

L'exemple ci-dessous déclare et définit trois symboles, "`my_variable`", "`my_constant`" ainsi que "`my_function`". Les deux premiers symboles sont locaux au module, tandis que le troisième est global.

```
1    .section .data
2  my_variable: .long 10
3
4    .section .rodata
5  my_constant: .asciz "constant string"
6
7    .section .text
8    .global my_function
9    .type my_function, %function
10 my_function:
11   bx  lr
```

Les symboles ne donnent aucune indication sur la nature de l'objet qu'ils représentent. Ils ne permettent pas de distinguer entre donnée et fonction, au demeurant information fort utile lors de sessions de débogage. La directive "`.type`" permet de pallier à ce manque. La directive "`.type str, %fonction`" indique que le symbole `str` est une fonction, tandis que la directive "`.type str, %object`" indique que le symbole "`str`" est une donnée (variable ou constante).

En assembleur, les symboles permettent également de représenter des expressions. Lors de l'assem-

blage du module, le symbole sera remplacé par son expression. L'exemple ci-dessous présente différentes manières de définir des symboles en assembleur.

```
1 SYMBOL_1 = 10
2
3 .equ   SYMBOL_2, 20
4 .set   SYMBOL_3, 30
5 .equiv SYMBOL_4, 40
6
7 #define SYMBOL_5 50
```

Avec l'utilisation du préprocesseur C/C++, il est également possible d'utiliser les symboles définis avec la pseudo-instruction "`#define`" (voir l'exemple ci-dessus).

5.3.4 Constantes

L'assembleur distingue deux types de constantes, les constantes numériques et les constantes caractères.

```
1 // numerical constants
2 an_integer:   .long 1234
3 a_float:      .float -1.05e4
```

```
1 // character constants
2 a_character:  .byte 'a'
3 a_string:     .asciz "hello world!\n"
```

Il existe principalement deux types de constantes numériques, les nombres entiers ("**`int`**" en C/C++) et les nombres à virgule flottante ("**`float`**" en C/C++).

- Nombres entiers
  - Décimal : suite de chiffres (*digits*) 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`", "`a`" et "`f`" ou "`A`" et "`F`"
- Nombres à virgule flottante
  - Normalement représentation décimale
  - Signe (optionnel), soit "`+`" ou "`-`"
  - Partie entière représentée par zéro ou plusieurs digits
  - Partie fractionnaire (optionnel) représentée par un "`.`" suivit de zéro ou plusieurs digits
  - Exposant en base décimale (optionnel) représenté par un "`E`" ou "`e`", un signe et un ou plusieurs digits

Il existe deux types de constantes de caractères, les caractères et les chaînes de caractères (*string*). Un caractère est contenu dans un octet. Sa valeur numérique peut être utilisée dans des expressions arithmétiques et logiques. Les chaînes de caractères sont représentées généralement par plusieurs octets. Leurs valeurs ne peuvent pas être utilisées dans des expressions numériques.

- Caractères
  - Caractère ASCII unique écrit soit avec un apostrophe (“`'`”) suivie du caractère, soit avec un caractère entre deux apostrophes
  - Pour écrire le caractère backslash, on doit utiliser un 2e backslash (“`\\`”)
- Chaînes de caractères
  - Une chaîne de caractères est écrite en plaçant une chaîne de caractères ASCII entre deux guillemets (‘`"`’)
  - Les caractères d’échappement peuvent également être utilisés :
  - “`\b`” : backspace (retour d’un caractère)
  - “`\f`” : formfeed (saut page)
  - “`\n`” : newline (nouvelle ligne)
  - “`\r`” : carriage-return (retour de chariot)
  - “`\t`” : horizontal tab (tabulateur horizontal)
  - “`\\`” : caractère ‘`\`’
  - “`\"`” : caractère ‘`"`’
  - Les caractères représentés par une valeur octale “`\000`” sont également autorisés
  - Les caractères représentés par une valeur hexadécimale “`\x1a`” sont également autorisés

5.3.5 Expressions

Le langage assembleur supporte l’utilisation d’expressions. Elles permettent de décrire plus explicitement une valeur et d’éviter ainsi de la calculer à la main et d’ajouter un commentaire d’explication. Les expressions peuvent s’utiliser aussi bien avec les symboles que comme opérande d’une instruction assembleur.

Les expressions couramment utilisées sont :

- Addition (“`+`”)
- Soustraction (“`-`”)
- Multiplication (“`*`”)
- Division entière (“`/`”)
- Modulo / reste d’une division entière (“`%`”)
- Décalages (“`>>`” et “`<<`”)
- Opérations logiques sur les bits : ou/or (“`|`”), et/and (“`&`”), ou exclusif/xor (“`^`”), inverse/not (“`!`”)

Quelques exemples :

```
1  #define SIZE 100
2
3  list:  .space SIZE+2,0xff // list[102] = {0xff,..., 0xff}
4
5    ldr     r0, =SIZE/4     // r0 = 25
6    ldr     r1, =2+4*SIZE   // r1 = 402
7    ldr     r2, =list+27    // r2 = &list[27]
8    ldrb    r2, [r2]        // r2 = list[27]
9    ldr     r3, ='a'-'A'    // r3 = 32
10   ldr     r4, =1<<30      // r4 = 0x40000000
```

5.3.6 Données

L'assembleur supporte la déclaration de variables et de constantes, mais ces données ne sont pas typées au sens de C/C++. Les seules indications nécessaires en assembleur sont la taille et si le type de la donnée est un nombre entier (signé ou non signé), un nombre à virgule flottante ou une chaîne de caractères.

Lors de leur déclaration, la directive "`.section`" permet de regrouper les données dans un segment de mémoire, une section, propre à leur qualité. Le contenu de ces sections sera ensuite placé en mémoire lors du chargement de l'application sur la cible.

L'assembleur distingue par défaut trois sections correspondant aux trois qualités de données standard :

- "`.section .data`" : pour les données avec une valeur initiale non nulle (!= 0)
- "`.section .bss`" : pour les données avec une valeur initiale nulle (== 0)
- "`.section .rodata`" : pour les constantes, données avec un accès en lecture uniquement

Il est cependant possible de définir d'autres sections si nécessaire. Dans ce cas, l'éditeur de liens devra en être informé afin de placer ces sections et leurs données à l'endroit souhaité dans la mémoire.

La déclaration d'une donnée nécessite trois champs, l'étiquette de la donnée (son symbole), le type de la donnée (sa taille) et finalement une expression ou plusieurs expressions séparées par des virgules ("`,`") (sa valeur initiale), par exemple :

```
1  var:    .long 10        // en C/C++: static long var = 10;
2  array:  .byte 1,2,3,4   // en C/C++: static char array[4] = {1,2,3,4}
```

Pour les données de type entier, quatre tailles sont disponibles :

- "`.byte`" : déclaration d'une donnée de 8 bits (1 octet)
- "`.hword`" ou "`.short`" : déclaration d'une donnée de 16 bits (2 octets)

— “`.word`” ou “**`.long`**” : déclaration d’une donnée de 32 bits (4 octets)
— “`.quad`” : déclaration d’une donnée de 64 bits (8 octets)

Pour les données de type virgule flottante selon le standard IEEE 754, deux tailles sont disponibles :

— “**`.float`**” : déclaration d’une donnée de 32 bits (4 octets)
— “**`.double`**” : déclaration d’une donnée de 64 bits (8 octets)

Pour les données de type chaîne de caractères, deux variantes sont disponibles :

— “`.ascii`” : chaîne de caractères non terminée
— “`.asciz`” : chaîne de caractères terminée avec un 0 (caractère “`\000`”)

Il est possible de réserver un espace mémoire d’une taille exprimée en octets et éventuellement de la remplir avec une valeur initiale, pour cela deux possibilités :

— “`.space number_of_bytes`”
— “`.fill number_of_data, data_size, fill_value"`

L’alignement des données en mémoire est un facteur important. Si les données sont bien alignées, le processeur pourra les accéder de façon optimale. La directive “`.align <expr>`” permet de définir l’alignement d’une donnée. L’alignement (“`<expr>`”) est indiqué en puissance de 2. Par exemple, la directive “`.align 3`” déplace l’emplacement de la donnée dans la mémoire afin que son adresse soit un multiple de 8 (2^3).

Voici quelques exemples de déclaration de données :

```
1 // Constant declaration
2   .section .rodata
3   .align 2
4
5 // C/C++: static const uint32_t my_long = 0x01020304;
6 my_long:     .long 0x01020304
7
8 // C/C++: static const char my_string[] = "hello world\n";
9 my_string:   .asciz "hello world\n"
```

```
1 // Declaration of data with initial value
2   .section .data
3   .align 3
4
5 // C/C++: static uint64_t my_longlong = 0x0102030405060708;
6 my_longlong: .quad 0x0102030405060708
7
8 // C/C++: static int8_t my_byte = -25;
9 my_byte:     .byte -25
```

```
1 // Declaration of data without initial value
2   .section .bss
```

```
3    .align 2
4
5  // C/C++: static int32_t my_word;
6  my_word:      .word
7
8  // C/C++: static int16_t my_hword = 0;
9  my_hword:     .hword 0
```

5.3.7 Code

La directive "`.section .text`" sert à regrouper tout le code d'une application et ses fonctions dans une même section. Une fois la section de code déclarée, il faut indiquer à l'assembleur le jeu d'instructions qu'il doit utiliser pour générer le code objet. Trois jeux d'instructions sont disponibles, ARM, Thumb et Thumb-2 :

```
1  // Instructions ARM      // Instructions Thumb    // Instructions Thumb-2
2    .section .text          .section .text           .section .text
3    .code 32                .code 16                 .code 16
4                                                     .syntax unified
5    .align 2                .align 2                 .align 2
```

ou

```
1  // Instructions ARM      // Instructions Thumb    // Instructions Thumb-2
2    .sections .text         .section .text           .section .text
3    .arm                    .thumb                   .thumb
4                                                     .syntax unified
5    .align 2                .align 2                 .align 2
```

Comme pour la déclaration des données, il est important que le code soit correctement aligné. La directive "`.align 2`" permet de garantir que le code soit placé à un multiple de 4. Là aussi, il est possible de placer le code dans des sections différentes. L'éditeur de liens devra également en être informé.

5.3.8 Fonctions

La déclaration d'une fonction s'effectue en plusieurs étapes. La fonction débute avec son étiquette (son symbole). Cette étiquette permet à d'autres fonctions de l'appeler. Si la fonction doit obtenir un accès public, externe au module, il est indispensable de le déclarer avec la directive "`.global`". Il sera également utile de déclarer cette fonction dans un fichier d'en-tête afin des codes développés en C/C++ puissent connaître sa signature. Pour faciliter le débogage, il est fortement conseillé d'entourer l'implémentation de la fonction avec les directives "`.type <str>, %function`" et "`.size <str>, .-<str>`".

L'exemple ci-dessus présente le squelette de la fonction publique "`my_fonction`".

```
1    .global my_function
2    .type my_function, %function
3  my_function:
4    bx  lr
5    .size my_function, .-my_fonction
```

5.3.9 Macro

Lors de développement en assembleur, il est courant de devoir répéter certaines séquences de code. Ces séquences peuvent être strictement identiques ou similaires, ne demandant que quelques petites adaptations, tels un identifiant ou une valeur numérique. Dans de tels cas, l'utilisation d'une fonction n'est pas forcément adaptée ou possible. Les macros offrent une solution appropriée.

Deux mots clefs à connaître : `.macro` et `.endm`.

```
1  // macro definition
2  .macro macname macarg1, macarg2
3    ldr r0, =\macarg1
4    ldr r1, =\macarg2
5  .endm
6
7  // usage of the macro
8  macname 10, 20
```

La définition d'une macro débute avec la directive "`.macro`" suivie d'un identifiant, un nom, dans l'exemple ci-dessus "`macname`". Si la macro nécessite des arguments, il est possible d'en passer en donnant leur nom après celui de la macro et en les séparant par une virgule ("`,`"), dans l'exemple "`macarg1`" et "`macarg2`". Pour utiliser et évaluer un argument à l'intérieur de la macro, il suffit de précéder son nom par une barre oblique inversée "`\`" (*backslash*). La directive "`.endm`" termine la définition de la macro.

Pour obtenir plus de détails sur la définition et l'utilisation de macro, le manuel en ligne [1] est un complément fort utile.

5.3.10 Assemblage conditionnel

Les directives pour un assemblage conditionnel peuvent s'avérer très utiles lors de développement de code en assembleur. Ces directives sont spécialement intéressantes lors de la réalisation de macros. Elles permettent de paramétriser l'expansion de la macro durant de la phase d'assemblage afin d'obtenir un code optimal.

1. https://sourceware.org/binutils/docs/as/Macro.html

Trois mots clefs à connaître : "**.if**", "**.else**" et "**.endif**".

```
1  // macro definition
2  .macro macname cond, macarg1, macarg2
3    ldr r0, =\macarg1
4    ldr r1, =\macarg2
5  .if \cond != 0
6    sub r0, #1
7  .else
8    add r0, #1
9  .endif
10 .endm
```

La directive "**.if** <expr>" marque le début d'un bloc de code conditionnel. Si l'évaluation de l'expression "<expr>" est non nulle, alors les instructions placées après la directive seront intégrées au code source, dans le cas contraire, elles seront tout simplement ignorées. La directive "**.endif**" marque la fin du bloc. La directive "**.else**" permet d'ajouter des instructions pour la condition inverse.

Le code ci-dessous donne deux exemples d'utilisation de la macro ainsi du résultat obtenu après son expansion lors de la phase d'assemblage.

```
1  // usage of the macro
2  _true:  macname 0, 10, 20
3  _false: macname 1, 10, 20
```

```
1  // generated code after assembly
2  _true:
3    ldr r0, =10
4    ldr r1, =20
5    add r0, #1
6
7  _false:
8    ldr r0, =10
9    ldr r1, =20
10   sub r0, #1
```

Il existe encore un grand nombre d'autres directives d'assemblage conditionnel [2].

5.4 Interfaçage C - Assembleur

De nos jours, le développement d'applications se réalise presque exclusivement en C/C++. Le langage assembleur ne s'utilise plus que pour l'implémentation de certaines fonctions très proches du proces-

2. https://sourceware.org/binutils/docs/as/If.html#If

seur. En suivant quelques règles et conventions, il est assez simple d'interfacer des routines développées en assembleur avec des fonctions réalisées en C/C++, ou vice-versa.

5.4.1 Conventions

Petit retour en arrière ! En programmation C/C++, la réalisation d'un programme modulaire nécessite bien naturellement un fichier d'implémentation (*.c/.cpp file*) pour les classes et leurs méthodes réalisant la fonction de l'application, mais également d'un fichier d'en-tête (*.h/.hpp file*). Ce fichier d'en-tête déclare quant à lui les parties publiques du module, tels que les types, les classes, les constantes ou les fonctions. Ces déclarations sont indispensables aux autres modules du programme afin qu'ils puissent connaître l'interface et interagir avec le module, son API (*Application Program Interface*).

Lors de la génération du code binaire, le compilateur doit respecter une ABI (*Application Binary Interface*) définissant des conventions d'appel des fonctions et méthodes lié à l'architecture du processeur cible. L'ABI définit également le rôle de chaque registre du processeur à usage général, tels que le passage d'arguments, le retour du résultat de la fonction ou des variables temporaires, ainsi que la responsabilité de leur intégrité, de leur sauvegarde, ceci aussi bien du côté de l'appelant (*caller*) que du côté de l'appelée (*callee*). Il définit également la structure de la pile, notamment l'emplacement des arguments de la fonction. Pour résoudre les dépendances entre les modules, l'appel des fonctions et l'accès aux variables globales, l'éditeur de liens doit connaître l'identifiant de leur symbole, de leur nom.

Identifiant des fonctions

C et C++ n'utilisent pas les mêmes conventions pour la génération des identifiants des fonctions (*Name Mangling*). C n'utilise que le nom de la fonction comme identifiant. En revanche C++ a des exigences plus strictes, dues en particulier au polymorphisme des fonctions (même nom de fonction, mais des arguments différents).

Pour interfacer des routines assembleur avec des fonctions C/C++, les conventions C sont généralement utilisées. Là également, la réalisation d'un fichier d'en-tête (*.h file*) est nécessaire. Afin de respecter les conventions C, alors que l'interface est utilisée par du code C++, il est indispensable d'enrober la déclaration des fonctions avec la directive "`extern "C"{ }`". Pour que l'interface puisse également être employée par du code C, il faut tester si l'interface est compilée par un compilateur C ou un compilateur C++ en testant le symbole "`__cplusplus`".

L'exemple ci-dessous présente la déclaration de la fonction "`my_routine`" dans un fichier d'en-tête ainsi que le squelette de son implémentation en assembleur.

```
1 #ifdef __cplusplus
```

```
2  extern "C" {
3  #endif
4
5  extern void my_routine(void);
6
7  #ifdef __cplusplus
8  }
9  #endif
```

en assembleur

```
1      .text
2      .align 2
3
4      .global my_routine
5      .type my_routine, %function
6  my_routine:
7      bx  lr
8      .size my_routine, .-my_routine
```

Usage des registres

Le document *"Procedure Call Standard for the Arm Architecture"* [3] décrit et définit l'utilisation des 16 registres à 32 bits des processeurs ARM visibles pour les jeux d'instructions ARM, Thumb et Thumb-2 (figure 5.3).

Les registres R0 à R3 servent à passer les quatre premiers arguments d'une fonction. Les registres R0 et R1 servent également à retourner le résultat de la fonction. Ces quatre registres peuvent également être utilisés librement sans devoir préalablement sauver leur contenu sur la pile (*scratch register*).

Les registres R4 à R10 servent à stocker le contenu de variables locales d'une routine. Leur contenu doit être préservé. Si la routine souhaite en faire usage, elle doit impérativement sauver leur contenu sur la pile avant de les utiliser.

Selon les options de compilation, le registre R11 sert de pointeur de trame (*Frame Pointer*). Il peut cependant également être utilisé pour stocker le contenu de variables locales. Dans tous les cas son contenu doit être préservé.

Le registre R12 (IP) peut être utilisé librement sans devoir préalablement sauver son contenu (*scratch register*).

Les registres R13, R14 et R15 (SP, LR et PC) ont des rôles bien définis et doivent être manipulés en conséquence. Concernant le pointeur de pile (SP), l'ABI spécifie que celui doit toujours être aligné sur

3. https://developer.arm.com/documentation/ihi0042/latest

Register	Synonym	Role
R0	a1	1st argument / result scratch
R1	a2	2nd argument / result scratch
R2	a3	3rd argument / result scratch
R3	a4	4th argument / result scratch
R4	v1	variable
R5	v2	variable
R6	v3	variable
R7	v4	variable
R8	v5	variable
R9	v6	variable
R10	v7	variable
R11	v8/FP	variable / frame pointer
R12	IP	scratch / intra procedure call scratch
R13	SP	stack pointer
R14	LR	link register
R15	PC	program counter

FIGURE 5.3 – Banque de registres

4 octets (`SP % 4 == 0`), mais qu'aux interfaces publiques cette contrainte est étendue à 8 octets (`SP % 8 == 0`).

5.4.2 Appel de fonction

Les jeux d'instructions des processeurs ARM proposent deux techniques pour l'appel d'une fonction ou d'une routine, l'appel direct et l'appel indirect. A chacune de ces techniques, une instruction :

— "`bl label`" : appel direct
— "`blx Rx`" : appel indirect

L'instruction "`bl label`" offre un appel direct à la fonction identifiée par son étiquette (`label`), mais souffre d'une restriction. L'offset entre la fonction appelante et la fonction appelée ne doit pas dépasser ±32 MiB pour le jeu d'instructions ARM, ±16 MiB pour Thumb-2 et ±4 MiB pour Thumb.

```
1 // direct call of the function "my_routine"
2 bl  my_routine
```

L'instruction "`blx Rx`" offre un appel indirect à une fonction. L'adresse de la fonction à appeler étant contenue dans un registre (`Rx`), cette technique permet d'accéder une fonction placée à une adresse quelconque dans l'ensemble de l'espace mémoire du processeur. Les pointeurs de fonction en C ou les méthodes virtuelles en C++, utilisent également cette technique pour l'appel de fonction par l'intermédiaire d'une variable, variable contenant l'adresse de la fonction à appeler.

```
1 // indirect call of the function "my_routine"
2 ldr  r0, =my_routine+1
3 blx  r0
```

Pour continuer l'exécution du code avec les jeux d'instructions Thumb/Thumb-2, l'adresse utilisée pour un appel indirect d'une fonction doit impérativement être impaire. Le code étant toujours aligné sur des adresses paires, il suffit d'incrémenter l'adresse de la fonction par 1 pour conserver l'exécution du code en Thumb/Thumb-2.

```
1 // function call through a function pointer
2 ldr  r0, =my_function_pointer
3 ldr  r0, [r0]
4 blx  r0
```

Les deux instructions ("`bl`" et "`blx`") sauvent automatiquement l'adresse de retour dans le registre LR, adresse pointant sur l'instruction placée juste après l'instruction d'appel. En exécutant l'instruc-

tion “`bx lr`”, la fonction appelée retourne à la fonction appelante. Cette dernière peut ainsi poursuivre son exécution.

```
1      .global my_routine
2      .type my_routine
3  my_routine:
4      // ...
5      bx  lr
6      .size my_routine, .-my_routine
```

Si la fonction appelée fait elle-même appel à d’autres fonctions, elle doit d’abord sauver le contenu du registre LR sur la pile avec l’instruction “`push {lr}`”. L’instruction “`pop {pc}`” lui permet de retourner à la fonction appelante.

```
1      .global my_routine
2      .type my_routine
3  my_routine:
4      push {lr}
5      // ...
6      pop  {pc}
7      .size my_routine, .-my_routine
```

5.4.3 Passage des arguments

Le passage des arguments à une fonction pose un certain nombre de questions :

— Quelles techniques existe-t-il pour passer des arguments ?
— Quels mécanismes existe-t-il pour réaliser ce passage d’arguments ?

Techniques

En C/C++, trois techniques existent pour le passage des arguments à une fonction :

— Passage par valeur (en C/C++)
— Passage par adresse (en C/C++)
— Passage par référence (en C++)

Avec la technique de **passage par valeur** (*call by value*), la fonction appelante crée une copie de la donnée réelle pour la fonction appelée. La fonction appelée reçoit une copie de la donnée réelle comme argument formel, qu’elle peut lire et modifier sans que la donnée réelle ne soit altérée. Pour la donnée réelle et l’argument formel, il existe deux emplacements distincts. La modification de l’argument formel par la fonction appelée n’a aucune conséquence sur la donnée réelle.

Les techniques de **passage par référence** et de **passage par adresse** sont du point vu du processeur strictement identique. Avec ces deux techniques (*call by reference*), la fonction appelante donne à la fonction appelée l'emplacement en mémoire de la donnée réelle, son adresse. La fonction appelée reçoit comme argument formel l'adresse de la donnée réelle en mémoire. La donnée réelle est ainsi partagée entre la fonction appelante et la fonction appelée. Il n'existe qu'un seul emplacement en mémoire (une seule adresse). Si la fonction appelée modifie la donnée, en passant par l'adresse contenue dans l'argument formel (accès indirect), cette modification sera visible et disponible pour la fonction appelante une fois la fonction appelée terminée.

Ces deux techniques sont naturellement également valables pour des appels de routines implémentées en assembleur.

Mécanismes

Les techniques pour le passage des arguments étant connues, il reste à imaginer des mécanismes pour les réaliser. En regardant l'infrastructure proposée par un processeur et selon le nombre et la taille des arguments, les compilateurs utilisent deux mécanismes :

- Passage par les registres
- Passage par la pile

Les registres offrent une méthode excessivement efficace et rapide pour le passage des arguments à une fonction. Par contre, ce mécanisme se bute au nombre limité de registres du processeur. A contrario, la pile ne souffre pas de limite de taille et offre une approche généralisée. Ces avantages sont contrebalancés par la lenteur des accès à la mémoire. Le choix d'une approche plutôt qu'une autre dépend fortement des possibilités du processeur et du langage de programmation. Les compilateurs modernes utilisent couramment les deux mécanismes.

Passage des arguments par les registres

Sur les processeurs ARM, le passage des arguments par les registres est limité. L'ABI permet le passage de seulement quatre arguments, mais au maximum quatre mots de 32 bits. Si le nombre d'arguments est supérieur à 4 ou que la taille des premiers arguments dépasse les quatre mots de 32 bits, le reste devra impérativement passer par la pile. L'ABI spécifie que les quatre mots de 32 bits doivent être placés dans les registres R0 à R3. R0 contient le premier argument/mot et R3 le dernier.

L'exemple ci-dessous présente l'appel d'une fonction avec quatre arguments, `a`, `b`, `c` et `d`, de type "`int`", soit 4 mots de 32 bits. Selon la convention, le registre R0 contient la valeur de l'argument `a`, R1 celui de `b`, R2 celui de `c` et R3 celui de `d`.

```
1 extern void my_function(int a, int b, int c, int d);
2
3 my_function(1, 2, 3, 4);
```

Le code assembleur ci-dessous représente l'implémentation cet appel.

```
1 ldr r0, =1
2 ldr r1, =2
3 ldr r2, =3
4 ldr r3, =4
5 bl  my_function
```

Le deuxième exemple présente l'appel d'une fonction avec trois arguments `a`, `b`, et `c`, de type "`uint64_t`" pour le premier et "`int32_t`" pour les 2 suivants, soit 1 mot de 64 bits et 2 mots de 32 bits. Selon la convention, les registres R0 et R1 contiennent l'argument `a`, R2 l'argument `b` et R3 `c`.

```
1 extern void my_function2(uint64_t a, int32_t b, int32_t c);
2
3 my_function2(0x1122334455667788, 5, 6);
```

voici l'implémentation assembleur

```
1 ldr r0, =0x55667788
2 ldr r1, =0x11223344
3 ldr r2, =5
4 ldr r3, =6
5 bl  my_function2
```

Passage des arguments par la pile

Le passage d'arguments par la pile suppose une convention précisant l'ordre dans lequel la fonction appelante doit pousser sur la pile le contenu de ces arguments. La fonction appelée accède ces valeurs en utilisant le pointeur de pile (SP).

L'ABI spécifie que dès le 5^{e} argument ou le 5^{e} mot de 32 bits les données doivent passer par la pile. Ces données doivent être placées sur la pile dans l'ordre inverse de leur déclaration ; le 5^{e} mot sera au sommet de la pile, suivi du 6e et ainsi de suite.

L'exemple ci-dessous présente une fonction "`my_sum`" effectuant l'addition de sept arguments de 32 bits chacun.

```
1 extern int my_sum(int a, int b, int c, int d, int e, int f, int g);
2
3 int sum = my_sum(1, 2, 3, 4, 5, 6, 7);
```

Les 4 premiers arguments sont passés par les registres et les 3 suivants par la pile (figure 5.4).

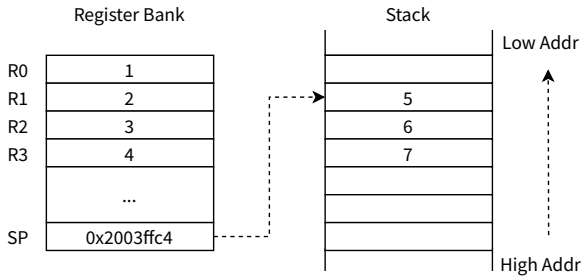


FIGURE 5.4 – Placement des arguments dans les registres et sur la pile

Pour passer les arguments `e`, `f` et `g`, la fonction appelante doit préalablement réserver l'espace suffisant sur la pile. Pour cela, elle décrémente le pointeur de pile (SP) du nombre de mots à placer sur la pile tout en respectant les conventions. Dans le cas présent, on décrémente le SP de 4 mots de 32 bits (3 mots pour les arguments et 1 mot pour respecter la convention d'alignement). Ensuite, il est possible de placer les arguments sur la pile. Après l'appel, la fonction appelante doit corriger le pointeur de pile pour rétablir sa valeur initiale.

```
1      subs    sp, #4*(3+1)    // reserve space on stack
2      movs    r0, #7          // push on stack g=7
3      str     r0, [sp, #8]
4      movs    r0, #6          // push on stack f=6
5      str     r0, [sp, #4]
6      movs    r0, #5          // push on stack e=5
7      str     r0, [sp, #0]
8      movs    r3, #4          // store in r3   d=4
9      movs    r2, #3          // store in r2   c=3
10     movs    r1, #2          // store in r1   b=2
11     movs    r0, #1          // store in r0   a=1
12     bl      my_sum          // call the fonction
13     adds    sp, #4*(3+1)    // restore value of SP
```

Pour la fonction appelée, les quatre premiers arguments sont directement disponibles dans les registres R0 à R3. Pour les trois suivants, elle doit les accéder indirectement en utilisant le pointeur de pile (SP) et un offset. Cet offset est le décalage entre le sommet de pile et l'emplacement de l'argument dans la pile.

```
1      .global my_sum
2      .type my_sum, %function
3  my_sum:
4      adds    r0, r1          // r0 = a + b
5      adds    r0, r2          // r0 = a + b + c
6      adds    r0, r3          // r0 = a + b + c + d
7      ldr     r1, [sp, #0]
8      adds    r0, r1          // r0 = a + b + c + d + e
9      ldr     r1, [sp, #4]
```

```
10      adds    r0, r1          // r0 = a + b + c + d + e + f
11      ldr     r1, [sp, #8]
12      adds    r0, r1          // r0 = a + b + c + d + e + f + g
13      bx      lr
14      .size my_sum, .-my_sum
```

5.4.4 Retour du résultat

Pour le retour du résultat d'une fonction, deux stratégies sont imaginables :

- Retour par les registres
- Retour par la mémoire

Le retour par les **registres** est la solution la plus efficace. Cependant elle se bute au nombre de registres disponibles. Selon l'ABI, seuls les registres R0 et R1 sont disponibles pour le retour des types de données fondamentaux du langage C/C++, tels que "`bool`", "**`int`**", "**`long`**" ou "**`float`**". Si la taille du type fondamental est plus petite ou égale à un mot de 32 bits, le retour s'effectue par le registre R0. Si elle est de 64 bits, le retour s'effectue par les registres R0 et R1. Le registre R0 est également utilisé pour le retour de données d'un type complexe pour autant que la taille du type soit inférieure ou égale à quatre octets.

Le retour par la **mémoire** est la solution pour le retour de données de types complexes dont la taille est supérieure à quatre octets. Pour retourner des données par la mémoire, la fonction appelante passe à la fonction appelée l'emplacement mémoire où stocker le résultat, l'adresse.

```
1  struct Result {
2      int r1;
3      int r2;
4  };
5
6  struct Result my_complex(int a);
7  void my_complex2(struct Result* r, int a);
```

Dans l'exemple ci-dessus, la fonction "`my_complex2`" et la fonction "`my_complex`" utilisent exactement les mêmes registres pour le passage des arguments. La fonction "`my_complex2`" l'effectue de façon explicite, tandis que pour la fonction "`my_complex`" cette tâche est laissée au compilateur.

5.4.5 Altérations des registres

Par nature, une fonction doit cohabiter avec les autres fonctions de l'application. Le nombre de registres d'un processeur étant limité, les fonctions ne peuvent pas en réserver pour leur usage exclusif. Elles ont l'obligation de se les partager. Pour éviter qu'une fonction appelée modifie le contenu de

registres utilisés par la fonction appelante, chaque fonction a l'obligation de sauver le contenu des registres dont elle a besoin avant de les modifier et de les restaurer avant de retourner vers la fonction appelante. Selon l'ABI le contenu des registres R4 à R11 doit être préservé.

```
1      .global my_context
2      .type my_context
3  my_context:
4      push {r4-r6, lr}
5      // ...
6      pop  {r4-r6, pc}
```

La pile sert naturellement de stockage pour ces sauvegardes. L'instruction "`push { ... }`", placée comme prologue de la fonction, sauve le contenu des registres placés entre les deux accolades. L'instruction "`pop { ... }`" les restaure. Elle se place comme épilogue de la fonction.

5.5 Jeux d'instructions

Les processeurs ARM implémentent une architecture "*Load & Store*". Cette architecture implique qu'une donnée soit préalablement chargée dans un registre interne du CPU avant de pouvoir la traiter (figure 5.5). Le design du CPU ne permet la manipulation de données que sur des mots de 32 bits. C'est donc lors du chargement de la donnée de la mémoire centrale ou d'un contrôleur de périphériques d'entrées/sorties que le CPU devra adapter la taille de cette donnée aux 32 bits de ses registres.

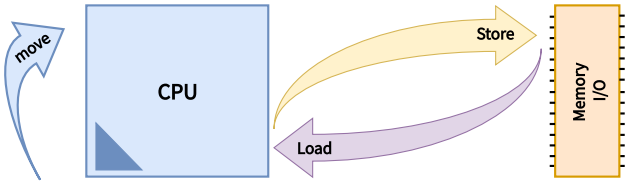


FIGURE 5.5 – Load & Store

Ce chapitre ne donne pas une description exhaustive des trois jeux d'instructions ARM, Thumb et Thumb-2. Car, hormis des différences sur l'exécution conditionnelle d'instructions et le nombre de registres accessible, les trois jeux d'instructions des processeurs ARM sont similaires. Dès lors, ce chapitre ne présente que les instructions principales, compatibles ARM et Thumb-2, permettant la réalisation des fonctions et routines nécessaires à la mise en œuvre du processeur et aux traitements des interruptions et exceptions.

5.5.1 Exécution conditionnelle

Deux différences majeures existent entre le jeu d'instructions ARM et le jeu d'instructions Thumb-2, l'exécution conditionnelle des instructions et la mise à jour optionnelle des fanions après l'exécution d'une instruction.

Conditions

Les fanions de conditions contenus dans le registre APSR, registre d'état du programme, permettent de dériver les conditions de branchement. Pour leurs utilisations, deux variantes sont possibles, soit un test direct du fanion, soit un test similaire aux conditions utilisées en C/C++. Les jeux d'instructions proposent des mnémoniques pour chacune de ces variantes (figure 5.6).

Le test des fanions consiste à regarder si le fanion est à 1 (*set*) ou s'il est à 0 (*clear*). L'alternative consistant à utiliser les conditions C/C++ est souvent plus évidente. Elle permet également d'effectuer des branchements conditionnels dépendant du résultat d'une opération sur des nombres entiers. Pour évaluer ces conditions, il est nécessaire de tester plusieurs fanions simultanément.

Mnemonic	Flag	Description
AL		Always (AL normally omitted)
EQ	Z==1	Equal
NE	Z==0	Not Equal
CS	C==1	Carry Set
CC	C==0	Carry Clear
MI	N==1	Minus (Negative Set)
PL	N==0	Plus (Negative Clear)
VS	V==1	Overflow Set
VC	V==0	Overflow Clear

Mnemonic	C/C++	Description
HI	>	Unsigned higher
HS	>=	Unsigned higher or same
LS	<=	Unsigned lower or same
LO	<	Unsigned lower

Mnemonic	C/C++	Description
GT	>	Signed greater than
GE	>=	Signed greater than or equal
LE	<=	Signed less than or equal
LT	<	Signed less than

FIGURE 5.6 – Fanions vs conditions

Il est dès lors important de distinguer un nombre entier signé d'un nombre entier non signé afin de choisir le mnémonique correspondant. Pour rappel, ces deux types d'entiers n'utilisent pas les mêmes fanions pour évaluer une même condition ; les nombres non signés utilisent les fanions Z et C, tandis que les signés emploient les fanions Z, V et N.

Jeu d'instructions ARM

Avec le jeu d'instructions ARM, pratiquement toutes les instructions peuvent s'exécuter de façon conditionnelle. Suite à un test ou une mise à jour des fanions de condition, l'instruction est capable de tester si son exécution peut se terminer ou doit s'interrompre.

La forme générale des instructions ARM est

```
opcode{cond}{S} Rd, <Sop>
```
ou
```
opcode{cond}{S} Rd, Rm, <Sop>
```

Le choix entre une exécution conditionnelle ou pas s'indique en plaçant la condition d'exécution "*{cond}*" après le mnémonique de l'instruction. Sans indication, l'instruction s'exécute toujours complètement, ce qui correspond à utiliser la condition "AL" (*always*). Voici une équivalence en C/C++ :

```
if (cond)Rd = opcode (Sop);
```
ou
```
if (cond)Rd = opcode (Rm, Sop);
```

Un "*S*" placé après le mnémonique de l'instruction indique que le changement de valeur des fanions, dû à l'exécution de l'instruction, doit être reporté dans le registre APSR pour un usage ultérieur.

Jeu d'instructions Thumb-2

Le jeu d'instructions Thumb-2 ne supporte qu'indirectement l'exécution conditionnelle. L'instruction "IT", (*If-Then*) permet une exécution conditionnelle pour les une à quatre instructions suivantes, évitant ainsi l'utilisation de branchements conditionnels. La syntaxe de l'instruction est

```
IT{pattern}{q} <cond>
```

La première instruction s'exécute si la condition "<*cond*>" est vraie. Pour les trois autres instructions, le choix est libre et défini par l'option "*{pattern}*", laquelle permet de spécifier jusqu'à trois conditions. Un "*T*" indique que l'instruction doit être exécutée si la condition est vraie, tandis qu'un "*E*" indique l'inverse. L'option "*{q}*" spécifie le codage des instructions ; "**.**n" (*narrow*) pour choisir un codage sur 16 bits de préférence et "**.**w" (*wide*) pour un codage sur 32 bits.

Le jeu d'instructions Thumb-2 supporte également la mise à jour optionnelle des fanions (N, Z, C, et V). Avec le suffixe "S", les fanions sont mis à jour et l'instruction est codée de manière compacte sur 16 bits. Sans le suffixe "S", les fanions ne sont pas mis à jour et l'instruction est codée sur 32 bits.

Exemple

La fonction C "`max`" ci-dessous, calculant le plus grand nombre entre deux entiers, démontre l'utilisation d'instructions conditionnelles.

```
1 int max(int a, int b) { return a > b ? a : b; }
```

et en assembleur

```
1 // Instructions ARM              // Instructions Thumb-2
2 max:                             max:
3    cmp     r0, r1                   cmp     r0, r1
4    movlt   r0, r1                   it.n    lt
5    bx      lr                       movlt   r0, r1
6                                     bx      lr
```

Avec le jeu d'instructions ARM, l'instruction "`movlt`" charge la valeur de `r1` dans le registre `r0` si `a` est plus petit que `b`. Avec le jeu d'instructions Thumb-2, if est nécessaire d'utiliser la combinaison des instructions "`it lt`" et "`movlt r0, r1`" pour réaliser cette condition.

Il peut sembler redondant de répéter la condition pour chaque instruction qui suit "`it`", mais ça permet de bien montrer dans le code que l'exécution est conditionnelle et ca permet à l'assembleur d'effectuer une vérification supplémentaire.

5.5.2 Opérande de décalage

Les jeux d'instructions ARM et Thumb-2 nomme le troisième opérande d'une instruction l'*opérande de décalage* (Sop - *Shifter Operand*). Cet opérande peut prendre trois formes différentes :

- Valeur immédiate
- Registre
- Décalage et rotation

Avec la forme **valeur immédiate**, l'opérande est généralement une valeur numérique entière comprise entre 0 et 255. Cependant, cette valeur peut subir une rotation par multiple de deux avant d'être utilisée. Quelques valeurs possibles :

```
1    movs  r0, #15        // r0 = 0x0000'000f
2    movs  r0, #983040    // r0 = 0x000f'0000
3    movs  r0, #245760    // r0 = 0x0003'c000
```

Avec la forme **registre**, l'opérande est simple un des seize registres du processeur. Par exemple :

```
1    mov   r0,  r2        // r0 = r2
2    mov   r15, r14       // pc = lr (return to the caller)
```

Avec la forme **décalage et rotation**, l'opérande est le résultat d'un décalage vers la droite, vers la gauche ou d'une rotation d'un des registres du processeur (`Rm`). Il est possible de spécifier la valeur de ce décalage ou de cette rotation, le nombre de bits, avec une valeur immédiate (`#<imm>`)ou avec le contenu d'un registre (`Rs`). Les formes possibles :

— Valeur immédiate : `Sop <= Rm, LSL|LSR|ASR|ROR #<imm>`
— Registre : `Sop <= Rm, LSL|LSR|ASR|ROR Rs`
— Rotation étendue : `Sop <= Rm, RRX`

LSL - Décalage logique (non signé) vers la gauche

Avecl'opérateur LSL, l'opérande de décalage est formé en décalant la valeur du registre `Rm` vers la gauche (figure 5.7). Un 0 est inséré à la place du bit vacant. Le dernier bit de poids fort décalé (`Rm[31]`, si le décalage est de 1) est, quant à lui, placé dans le carry (`APSR[C]`) si une mise à jour des fanions est souhaitée (`S==1`).

FIGURE 5.7 – LSL - Décalage logique (non signé) vers la gauche

Exemple :

```
1     movs  r0, r1, lsl #2    // r0 = r1 << 2
2  ou
3     lsls  r0, r1, #2
```

Si `R1` vaut `0xb000'8080`, alors l'exécution de l'instruction `R0` vaudra `0xc002'0200`. La valeur du carry est 0, car la valeur du 30e bit vaut 1 (`R1[30] == 0`).

En C/C++, cette opération correspond à un décalage d'un nombre entier non signé ou non signé vers la gauche.

```
1     unsigned r1 = 0xb0008080;
2     unsigned r0 = r1 << 2;
```

LSR - Décalage logique (non signé) vers la droite

Avec l'opérateur LSR, l'opérande de décalage est formé en décalant la valeur du registre `Rm` par la droite (figure 5.8). Un 0 est inséré à la place du ou des bits vacants. Le dernier bit de poids faible décalé

(`Rm[3]`, si le décalage est de 4) est, quant à lui, placé dans le carry (`APSR[C]`) si une mise à jour des fanions est souhaitée (`S==1`).

FIGURE 5.8 – LSR - Décalage logique (non signé) vers la droite

Exemple :

```
1     movs  r0, r1, lsr #8    // r0 = (unsigned)r1 >> 8
2  ou
3     lsrs  r0, r1, #8
```

Si `R1` vaut `0x8000'8080`, alors l'exécution de l'instruction `R0` vaudra `0x0080'0080`. La valeur du carry est 1, car la valeur du 8e bit vaut 1 (`R1[7] == 1`).

En C/C++, cette opération correspond à un décalage d'un nombre entier non signé vers la droite.

```
1     unsigned r1 = 0x80008080;
2     unsigned r0 = r1 >> 8;
```

ASR - Décalage arithmétique (signé) vers la droite

Avec l'opérateur ASR, l'opérande de décalage est formé en décalant la valeur du registre `Rm` par la droite (figure 5.9). Le bit du poids fort (`Rm[31]`) est inséré à la place du bit vacant. Le dernier bit de poids faible décalé (`Rm[0]`) est, quant à lui, placé dans le carry (`APSR[C]`) si une mise à jour des fanions est souhaitée (`S==1`).

FIGURE 5.9 – ASR - Décalage arithmétique (signé) vers la droite

Exemple :

```
1     mov   r0, r1, asr #8    // r0 = (signed)r1 >> 8
2  ou
3     asr   r0, r1, #8
```

Si `R1` vaut `0x8000'8080`, alors l'exécution de l'instruction `R0` vaudra `0xFF80'0080`. Par contre,

si R1 vaut 0x7000'8080, alors l'exécution de l'instruction R0 vaudra 0x0070'0080. La valeur du carry reste inchangée.

En C/C++, cette opération correspond à un décalage d'un nombre entier signé vers la droite.

```
1     int r1 = 0x80008080;
2     int r0 = r1 >> 8;
```

ROR - Rotation vers la droite

Avec l'opérateur ROR, l'opérande de décalage est formé en effectuant une rotation du contenu du registre Rm par la droite (figure 5.10). Le bit du poids faible (Rm[0]) est inséré à la place du bit de poids fort (Rm[31]). Cette opération se répète pour le nombre de bits correspondant à la rotation souhaitée. Le dernier bit de poids faible décalé (Rm[4], si la rotation est de 5) est, quant à lui, placé dans le carry (APSR[C]) si une mise à jour des fanions est souhaitée (S==1).

FIGURE 5.10 – ROR - Rotation vers la droite

Exemple :

```
1     movs   r0, r1, ror #4
2  ou
3     rors   r0, r1, #4
```

Si R1 vaut 0x0700'8083, alors l'exécution de l'instruction R0 vaudra 0x3070'0808. La valeur du carry est 0, car la valeur du 4e bit vaut 0 (R1[3] == 0).

Cette opération n'a pas d'équivalent en C/C++.

RRX - Rotation étendue vers la droite

Avec l'opérateur RRX, l'opérande de décalage est formé en effectuant une rotation de 1 bit du contenu du registre Rm par la droite. Le carry (APSR[C]) est inséré à la place du bit de poids fort Rm[31]. Le bit de poids faible (Rm[0]) est quant à lui placé dans le carry (APSR[C]) si une mise à jour des fanions est souhaitée (S==1).

Exemple :

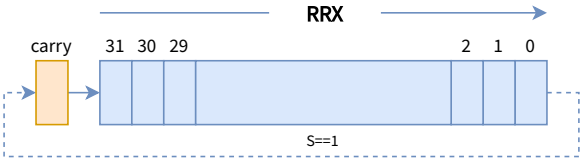


FIGURE 5.11 – RRX - Rotation étendue vers la droite

```
1     movs   r0, r1, rrx
2  ou
3     rrxs   r0, r1
```

Si `R1` vaut `0x0700'8080` et que le carry vaut 1, alors l'exécution de l'instruction `R0` vaudra `0x8380'4040`. La valeur du carry est 0, car la valeur du 1er bit vaut 1 (`R1[0] == 0`).

Cette opération n'a pas d'équivalent en C/C++.

5.5.3 Echange de données avec les registres

Les opérations "`move`" permettent d'échanger de données avec les registres. Elles supportent le chargement d'une valeur numérique immédiate dans un registre et la copie du contenu d'un registre dans un autre.

- **`MOV{S} Rd, <Sop>`** : l'instruction charge dans le registre `Rd` le résultat de l'opérande de décalage <`Sop`> (*shifter operand*).
- **`MOVW Rd, #<imm16>`** : l'instruction charge une valeur de 16 bits (0 à 65535) dans les deux octets inférieurs du registre `Rd` (`Rd[31:16] = 0; Rd[15:0] = #<imm16>`).
- **`MOVT Rd, #<imm16>`** : l'instruction charge une valeur de 16 bits (0 à 65535) dans les deux octets supérieurs du registre `Rd` (`Rd[31:16] = #<imm16>, Rd[15:0] = unchanged`).
- **`MVN{S} Rd, <Sop>`** : l'instruction charge dans le registre Rd le complément à 1 (valeur binaire inverse) de l'opérande de décalage <`Sop`>.

Quelques exemples :

```
1     movs  r0, r1              // r0 = r1
2     movs  r0, r1, lsl #5      // r0 = r1 << 5
3     movs  r0, #0              // r0 = 0
4     movw  r0, #0x034a         // r0 = 0x0000'034a
5     movt  r0, #0x1f00         // r0 = 0x1f00'034a
6     mvns  r1, #0              // r0 = 0xffff'ffff
```

5.5.4 Echange de données avec la mémoire

Par son architecture et pour traiter une donnée, le CPU des processeurs ARM doit préalablement copier la donnée (*load*) de la mémoire centrale ou d'un contrôleur de périphériques dans un de ses registres. Une fois disponible, le CPU peut la manipuler selon les instructions du programme. Le traitement terminé, le CPU peut stocker le résultat (*store*) dans la mémoire ou le contrôleur.

Le CPU ne pouvant opérer que sur des données de 32 bits, c'est lors de leur chargement (figure 5.12) que l'instruction doit connaître leur taille (8, 16, 32 ou 64 bits) et leur type (signée ou non signée). Lors du stockage du résultat, seule la taille de la donnée est nécessaire.

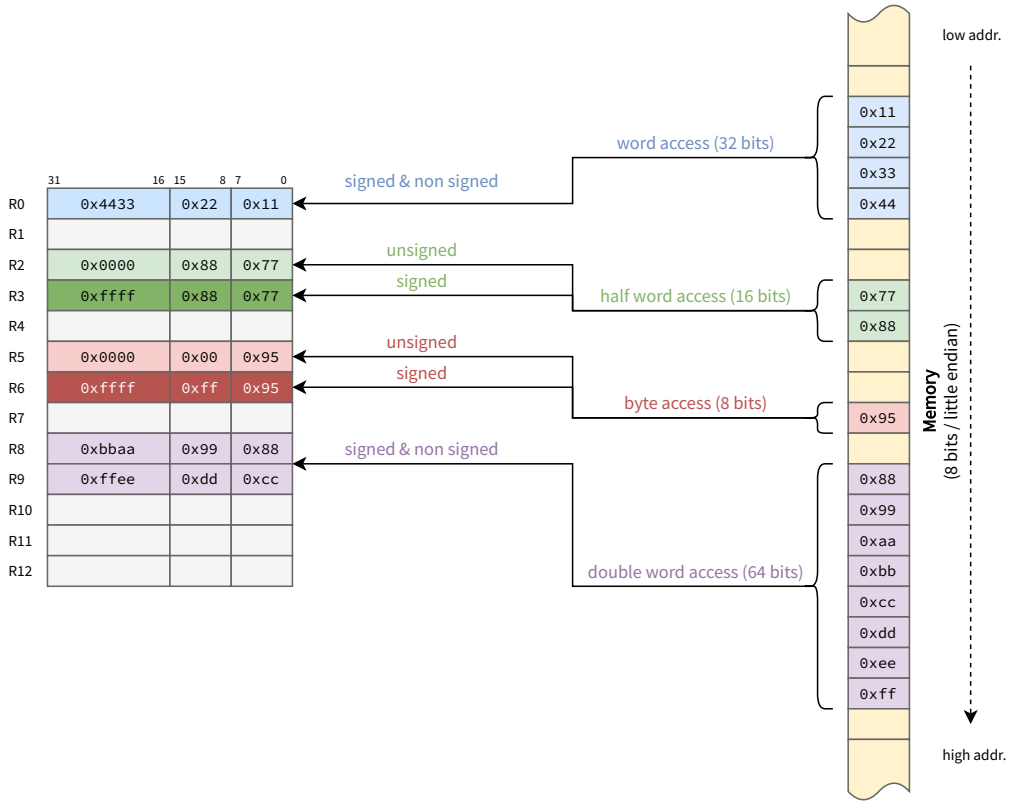


FIGURE 5.12 – Echange de données avec la mémoire

L'accès à une donnée requiert obligatoirement l'utilisation d'un registre (Rn) contenant l'adresse de cette donnée en mémoire centrale ou l'adresse du registre du contrôleur de périphériques. Pour charger cette adresse, il existe deux variantes selon qu'il s'agit d'une variable ou d'un contrôleur (figure 5.13).

Prenons un pseudo-code C pour illustrer ces accès

```
1    struct ctrl {
2       uint32_t stat;
```

```
3       uint32_t ctrl;
4       uint32_t pad[2];
5       uint32_t rx;
6       uint32_t tx;
7   };
8   static volatile struct ctrl* io = (struct ctrl*)0x40001000;
9   static uint16_t array[] = { 1, 2, 3, 4 };
10  static int8_t s_octet = -1;
11
12  array[1] = array[2];
13  io->tx = io->rx;
14  int8_t tmp = s_octet;
```

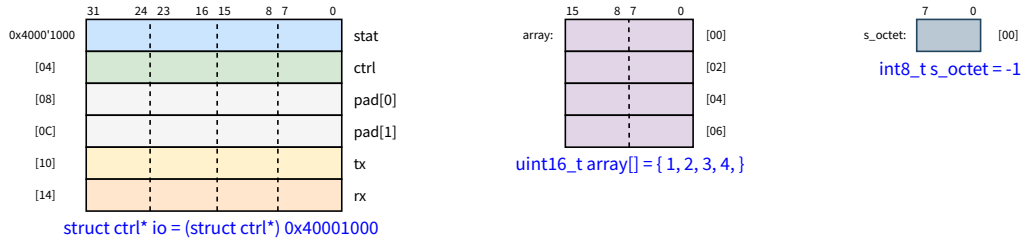


FIGURE 5.13 – Représentation mémoire de ces données

Si la donnée est une variable (*array* ou *s_octet*), l'assembleur permet d'utiliser son étiquette pour obtenir son adresse. Par contre, s'il s'agit d'un contrôleur (*io*), c'est plutôt que la valeur numérique de l'adresse de base du contrôleur qui est employée, adresse généralement disponible dans le manuel d'utilisateur du contrôleur.

```
1   ldr   r0, =array       // r0 = &array[0]
2   ldr   r1, #0x40001000  // r1 = 0x0x4000'1000
3   ldr   r2, =s_octet     // r2 = &s_octet
```

Pour l'instant, le code assembleur ci-dessus ne met à disposition que l'adresse de base de la variable ou du contrôleur. Alors en regardant le code C, il est légitime de se poser la question :

Comment procéder pour accéder un registre particulier du contrôleur ou un élément du tableau ?

Une solution est d'utiliser directement l'adresse de la variable, de l'élément d'un tableau ou du registre du contrôleur. Cette technique est simple, mais peu efficace, car pour chaque donnée à traiter une adresse doit être stockée avec le code et chargée dans un registre du CPU durant l'exécution du programme. Une autre approche nettement plus performante consiste à ne charger que l'adresse de base de la variable ou du contrôleur et d'y ajouter un offset (<ofs>) afin d'obtenir l'adresse de la donnée à traiter. Cette offset est le déplacement ou l'écart en octets entre l'adresse de base de la variable ou du contrôleur et l'adresse de l'élément/du registre. Cette offset peut être donnée soit par une valeur immédiate, soit par le contenu d'un registre du CPU (Rm).

Pour charger une donnée dans un registre, les jeux d'instructions ARM proposent les instructions :

- **`LDR Rd, [Rn, <ofs>]`** : pour charger un mot de 32 bits stocké à l'adresse (`Rn + <ofs>`) dans le registre `Rd`.
- **`LDRH Rd, [Rn, <ofs>]`** : pour charger un mot de 16 bits non signé stocké à l'adresse (`Rn + <ofs>`) dans les 16 premiers bits du registre `Rd` (`Rd[15:0]`), les 16 bits de poids fort seront simplement mis à 0 (`Rd[31:16] = 0`).
- **`LDRB Rd, [Rn, <ofs>]`** : pour charger un mot de 8 bits non signé stocké à l'adresse (`Rn + <ofs>`) dans les 8 premiers du registre `Rd` (`Rd[7:0]`), les 24 bits de poids fort seront simplement mis à 0 (`Rd[31:8] = 0`).
- **`LDRD Rd, [Rn, <ofs>]`** : pour charger un mot de 64 bits stocké à l'adresse (`Rn + <ofs>`) dans la paire de registres `Rd` et `Rd+1`. Le registre `Rd` doit être un registre pair (R0, R2, R4, ...).
- **`LDRSH Rd, [Rn, <ofs>]`** : pour charger un mot de 16 bits signé stocké à l'adresse (`Rn + <ofs>`) dans les 16 premiers bits du registre `Rd` (`Rd[15:0]`), la valeur des 16 bits de poids fort sera identique au seizième bit de la donnée à charger (`Rd[31:16] = data[15]`). On parle d'extension du bit de signe.
- **`LDRSB Rd, [Rn, <ofs>]`** : pour charger un mot de 8 bits signé stocké à l'adresse (`Rn + <ofs>`) dans les 8 premiers du registre `Rd` (`Rd[7:0]`), la valeur des 24 bits de poids fort sera identique au huitième bit de la donnée à charger (`Rd[31:8] = data[7]`).

Pour stocker le contenu d'un registre en mémoire, les jeux d'instructions ARM proposent les instructions :

- **`STR Rd, [Rn, <ofs>]`** : pour stocker les 32 bits du registre `Rd`.
- **`STRH Rd, [Rn, <ofs>]`** : pour stocker les 16 bits premiers du registre `Rd` (`Rd[15:0]`).
- **`STRB Rd, [Rn, <ofs>]`** : pour stocker les 8 bits premiers du registre `Rd` (`Rd[7:0]`).
- **`STRD Rd, [Rn, <ofs>]`** : pour stocker les 64 bits dans la paire de registres `Rd` et `Rd+1`.

Maintenant il est possible d'implémenter le pseudo-code C en code assembleur

```
1    ldrh  r3, [r0, #(2*2)]     // r3 = array[2]
2    strh  r3, [r0, #(1*2)]     // array[1] = r3 --> array[1] = array[2]
3
4    ldr   r3, [r1, #0x10]      // r3 = io->rx
5    str   r3, [r1, #0x14]      // io->tx = r3   --> io->tx = io->rx
6
7    ldrsb r3, [r2]             // r3 = s_octet
```

Le mode d'adressage présenté précédemment est le mode d'adressage élémentaire des processeurs ARM. Il devrait cependant suffire à réaliser une grande majorité des routines nécessitant une implémentation en code assembleur. Mais si toutefois des algorithmes plus complexes exigent des fonctionnalités plus avancées, le jeu d'instructions ARM offre d'autres modes d'adressage, tels que pré-indexé,

post-indexé ou par décalage. Alors avant de s'aventurer à écrire beaucoup de lignes de code, il est recommandé et judicieux de jeter un petit coup d'œil sur la documentation du processeur.

5.5.5 Echange multiple de données avec la mémoire

Les échanges multiples de données avec la mémoire servent à sauver, respectivement à restaurer, le contenu de plusieurs registres du CPU en une seule instruction. Il existe principalement deux paires d'instructions "`push`/`pop`" et "`stmia`/`ldmia`".

La première paire "`push`/`pop`" sert à la sauvegarde des registres du CPU sur la pile à l'entrée d'une fonction et à leur restauration à sa sortie.

- **`PUSH {<reglist>}`** : pousse sur la pile le contenu des registres indiqués avec l'instruction.
- **`POP {<reglist>}`** : place le contenu de la pile dans les registres indiqués avec l'instruction.

`<reglist>` spécifie la liste des registres à pousser sur la pile ou à retirer de la pile. Il est possible d'énumérer chaque registre en les séparant les uns des autres par une virgule (`,`) ou de donner une série en séparant le premier registre du dernier avec un trait d'union (−).

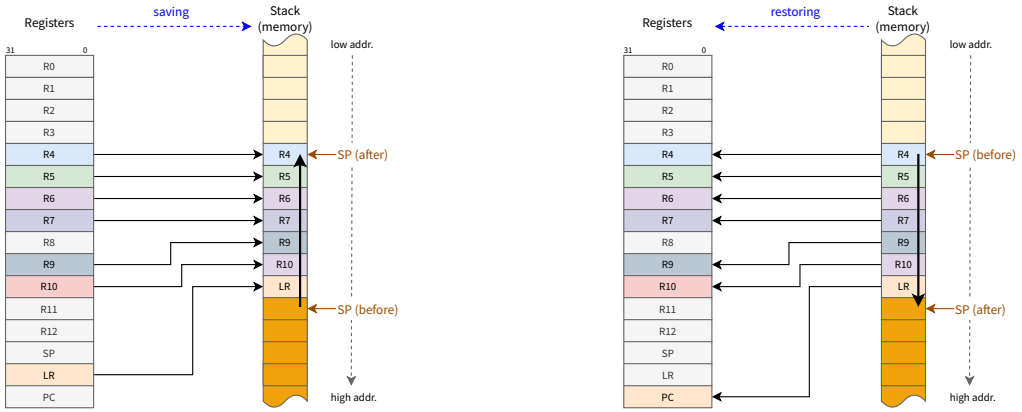


FIGURE 5.14 – Sauvegarde des registres sur la pile

Pour rappel, les processeurs ARM implémentent une pile pleine descendante (*full descending*) ; en poussant le contenu de registres sur la pile, le registre SP se décrémente, respectivement en retirant le contenu d'un registre, le registre SP s'incrémente (figure 5.14). Le registre SP pointe toujours sur le sommet de pile. L'instruction "`push`" place le contenu des registres sur pile dans l'ordre inverse (le plus grand numéro en premier et le plus petit en dernier). L'instruction "`pop`" travaille en sens inverse.

La deuxième paire "`stmia`/`ldmia`" sert à la sauvegarde des registres du CPU en mémoire à une adresse spécifiée par le registre `Rn`. Cette paire est généralement utilisée par les OS pour sauver le

contexte d'une tâche dans une structure de données servant à sa gestion (TCB - *Task Control Block*).

— **`STMIA Rn, {<reglist>}`** : stocke le contenu des registres indiqués avec l'instruction à l'emplacement mémoire spécifié par le registre Rn.
— **`LDMIA Rn, {<reglist>}`** : restaure le contenu des registres indiqués avec l'instruction avec les données stockées à l'emplacement mémoire spécifié par le registre Rn.

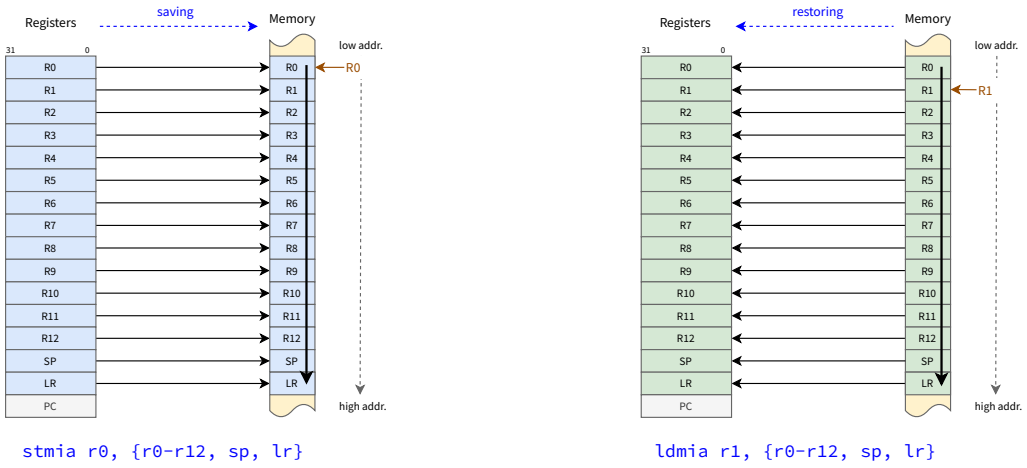


FIGURE 5.15 – Sauvegarde des registres en mémoire

Ce chapitre présente seulement les paires d'instructions les plus importantes. Les jeux d'instructions ARM et Thumb-2 offrent encore d'autres paires d'instructions. Il peut être judicieux de jeter un petit coup d'œil sur la documentation, si nécessaire.

5.5.6 Echange de données avec les registres spéciaux et des co-processeurs

L'accès aux registres spéciaux du CPU ainsi qu'aux registres des co-processeurs nécessite l'emploi d'instructions spécifiques.

— **`MRS`/`MSR`** : cette paire d'instructions donne accès aux registres APSR, CPSR et SPSR pour les µP Cortex-A et aux registres APSR, IPSR, EPSR, PSR, PRIMASK, FAULTMASK, BASEPRI et CONTROL pour les µC Cortex-M.
— **`CPS`** : cette instruction permet de gérer l'état du processeur et plus particulièrement l'activation des interruptions.
— **`MRC`/`MCR`** : cette paire d'instructions donne accès aux registres des co-processeurs.

L'utilisation de ces instructions étant très spécifique, il est impératif de jeter un coup d'œil sur la documentation avant de s'en servir.

5.5.7 Opérations arithmétiques

Les deux jeux d'instructions ARM et Thumb-2 proposent des instructions pour effectuer des opérations d'addition, de soustraction et de multiplication sur des nombres entiers signés et non signés. Par contre pour l'opération de division, seul le jeu d'instructions Thumb-2 la propose.

Ce chapitre présente seulement les instructions les plus importantes. Les jeux d'instructions ARM et Thumb-2 en offrent encore bien d'autres. Il peut être judicieux de jeter un petit coup d'œil sur la documentation, si nécessaire.

Additions

— **ADD{S} Rd, Rn, <Sop>** : additionne le contenu du registre Rn avec l'opérande de décalage <Sop> et stocke le résultat dans le registre Rd (Rd = Rn + Sop).

```
int32_t Rn  = 10;
int32_t Sop = 15;
int32_t Rd  = Rn + Sop;
```

```
ldr   r0, =Rn         // r0 = &Rn
ldr   r0, [r0]        // r0 = Rn
ldr   r1, =Sop        // r1 = &Sop
ldr   r1, [r1]        // r1 = Sop
adds  r0, r0, r1      // r0 = r0 + r1
ldr   r1, =Rd         // r1 = &Rd
str   r0, [r1]        // Rd = r0
```

— **ADC{S} Rd, Rn, <Sop>** : additionne le contenu du registre Rn avec l'opérande de décalage <Sop> et la valeur du fanion C (report - *carry*) et stocke le résultat dans le registre Rd (Rd = Rn + Sop + C). Cette instruction permet l'addition de nombre dont la taille est supérieure à 32 bits.

```
uint64_t Rn  = 0xffffffff;
uint64_t Sop = 1;
uint64_t Rd  = Rn + Sop;
```

```
ldr   r0, =Rn         // r0    = &Rn
ldrd  r0, [r0]        // r0,r1 = Rn
ldr   r2, =Sop        // r2    = &Sop
ldrd  r2, [r2]        // r2,r3 = Sop
adds  r0, r0, r2      // r0    = r0 + r2
adcs  r1, r1, r3      // r1    = r1 + r3 + carry
ldr   r2, =Rd         // r2    = &Rd
strd  r0, [r2]        // Rd    = r0,r1
```

Soustractions

— **SUB{S} Rd, Rn, <Sop>** : soustrait le contenu l'opérande de décalage <Sop> au contenu du registre Rn et stocke le résultat dans le registre Rd (`Rd = Rn - Sop`).

```
uint32_t Rn  = 10;
uint32_t Sop = 15;
uint32_t Rd  = Rn - Sop;
```

```
ldr   r0, =Rn        // r0 = &Rn
ldr   r0, [r0]       // r0 = Rn
ldr   r1, =Sop       // r1 = &Sop
ldr   r1, [r1]       // r1 = Sop
subs  r0, r0, r1     // r0 = r0 - r1
ldr   r1, =Rd        // r1 = &Rd
str   r0, [r1]       // Rd = r0
```

— **SBC{S} Rd, Rn, <Sop>** : soustrait le contenu l'opérande de décalage <Sop> et de l'inverse du fanion C (retenue - *inverse carry*) du contenu du registre Rn et stocke le résultat dans le registre Rd (`Rd = Rn - (Sop + ~C)`). Cette instruction permet la soustraction de nombre dont la taille est supérieure à 32 bits.

```
int64_t Rn  = 0x100000000;
int64_t Sop = 1;
int64_t Rd  = Rn - Sop;
```

```
ldr   r0, =Rn     // r0    = &rn
ldrd  r0, [r0]    // r0,r1 = Rn
ldr   r2, =Sop    // r2    = &Sop
ldrd  r2, [r2]    // r2,r3 = Sop
subs  r0, r0, r2  // r0    = r0 - r1
sbcs  r1, r1, r3  // r1    = r1 - (r3 + ~carry)
ldr   r2, =Rd     // r2    = &Rd
strd  r0, [r2]    // Rd    = r0,r1
```

— **RSB{S} Rd, Rn, <Sop>** : effectue la soustraction inverse entre le contenu de l'opérande de décalage <Sop> et contenu du registre Rn et stocke le résultat dans le registre Rd (`Rd = Sop - Rn`). Si <Sop> vaut 0, alors cette instruction calcule la valeur négative (`Rd = -Rn`).

```
int32_t Rn  = 10;
int32_t Rd  = -Rn;
```

```
ldr   r0, =Rn        // r0 = &Rn
ldr   r0, [r0]       // r0 = Rn
rsbs  r0, r0, #0     // r0 = 0 - r0
ldr   r1, =Rd        // r1 = &Rd
str   r0, [r1]       // Rd = r0
```

Multiplications

— **`MUL Rd, Rn, Rm`** : effectue la multiplication entre le contenu du registre `Rn` et le contenu du registre `Rm` et stocke le résultat dans le registre `Rd` (`Rd = Rn * Rm`).

```
int32_t Rn  = 10;
int32_t Rm  = -25;
int32_t Rd  = Rn * Rm;
```

```
ldr   r0, =Rn        // r0 = &Rn
ldr   r0, [r0]       // r0 = Rn
ldr   r1, =Rm        // r1 = &Rm
ldr   r1, [r1]       // r1 = Rm
mul   r2, r0, r1     // r2 = r0 * r1
ldr   r1, =Rd        // r1 = &Rd
str   r2, [r1]       // Rd = r2
```

Divisions

— **`SDIV Rd, Rn, Rm`**: effectue la division entre deux nombres signés contenus dans le registre `Rn` et le contenu du registre `Rm` et stocke le résultat dans le registre `Rd` (`Rd = Rn / Rm`).

```
int32_t Rn  = 250;
int32_t Rm  = -25;
int32_t Rd  = Rn / Rm;
```

```
ldr   r0, =Rn        // r0 = &Rn
ldr   r0, [r0]       // r0 = Rn
ldr   r1, =Rm        // r1 = &Rm
ldr   r1, [r1]       // r1 = Rm
sdiv  r2, r0, r1     // r2 = r0 / r1
ldr   r1, =Rd        // r1 = &Rd
str   r2, [r1]       // Rd = r2
```

— **`UDIV Rd, Rn, Rm`** : effectue la division entre deux nombres non signés contenus dans le registre `Rn` et le contenu du registre `Rm` et stocke le résultat dans le registre `Rd` (`Rd = Rn / Rm`).

```
uint32_t Rn  = 0xf0000000;
uint32_t Rm  = 0x00000010;
uint32_t Rd  = Rn / Rm;
```

```
ldr   r0, =Rn        // r1 = &Rn
ldr   r0, [r0]       // r1 = Rn
ldr   r1, =Rm        // r0 = &Rm
ldr   r1, [r1]       // r0 = Rm
udiv  r2, r0, r1     // r2 = r0 / r1
```

```
6  ldr   r1, =Rd        // r1 = &Rd
7  str   r2, [r1]       // Rd = r2
```

5.5.8 Opérations logiques

Les deux jeux d'instructions ARM et Thumb-2 proposent des instructions pour effectuer des opérations logiques (et, ou, ou-exclusif) sur des nombres entiers.

— **AND{S} Rd, Rn, <Sop>** : effectue un "et" logique entre deux nombres contenus dans le registre Rn et le contenu de l'opérande de décalage <Sop> et stocke le résultat dans le registre Rd (Rd = Rn & Sop).

```
1  uint32_t Rn  = 0xf800;
2  uint32_t Sop = 0x8800;
3  uint32_t Rd  = Rn & Sop;
```

```
1  ldr   r0, =Rn        // r0 = &Rn
2  ldr   r0, [r0]       // r0 = Rn
3  ldr   r1, =Sop       // r1 = &Sop
4  ldr   r1, [r1]       // r1 = Sop
5  ands  r0, r0, r1     // r0 = r0 & r1
6  ldr   r1, =Rd        // r1 = &Rd
7  str   r0, [r1]       // Rd = r0
```

— **ORR{S} Rd, Rn, <Sop>** : effectue un "ou" logique entre deux nombres contenus dans le registre Rn et le contenu de l'opérande de décalage <Sop> et stocke le résultat dans le registre Rd Rd (Rd = Rn | Sop).

```
1  uint32_t Rn  = 0xf800;
2  uint32_t Sop = 0x0700;
3  uint32_t Rd  = Rn & Sop;
```

```
1  ldr   r0, =Rn        // r0 = &Rn
2  ldr   r0, [r0]       // r0 = Rn
3  ldr   r1, =Sop       // r1 = &Sop
4  ldr   r1, [r1]       // r1 = Sop
5  orrs  r0, r0, r1     // r0 = r0 | r1
6  ldr   r1, =Rd        // r1 = &Rd
7  str   r0, [r1]       // Rd = r0
```

— **EOR{S} Rd, Rn, <Sop>** : effectue un "ou-exclusif" logique entre deux nombres contenus dans le registre Rn et le contenu de l'opérande de décalage <Sop> et stocke le résultat dans le registre Rd (Rd = Rn ^ Sop).

```
1  uint32_t Rn  = 0xf800;
2  uint32_t Sop = 0x8888;
3  uint32_t Rd  = Rn & Sop;
```

```
ldr   r0, =Rn        // r0 = &Rn
ldr   r0, [r0]       // r0 = Rn
ldr   r1, =Sop       // r1 = &Sop
ldr   r1, [r1]       // r1 = Sop
eors  r0, r0, r1     // r0 = r0 ^ r1
ldr   r1, =Rd        // r1 = &Rd
str   r0, [r1]       // Rd = r0
```

— **BIC{S} Rd, Rn, <Sop>** : efface les bits à 1 contenus dans l'opérande de décalage <Sop> du contenu du registre Rn. Cette instruction effectue un "et" logique entre le nombres contenu dans le registre Rn et l'inverse du contenu de l'opérande de décalage <Sop> et stocke le résultat dans le registre Rd (`Rd = Rn & ~Sop`).

```
uint32_t Rn  = 0xf800;
uint32_t Sop = 0xf000;
uint32_t Rd  = Rn & ~Sop;
```

```
ldr   r0, =Rn        // r0 = &Rn
ldr   r0, [r0]       // r0 = Rn
ldr   r1, =Sop       // r1 = &Sop
ldr   r1, [r1]       // r1 = Sop
bics  r0, r0, r1     // r0 = r0 & ~r1
ldr   r1, =Rd        // r1 = &Rd
str   r0, [r1]       // Rd = r0
```

Ce chapitre ne présente que les instructions principales. Les jeux d'instructions ARM et Thumb-2 en offrent encore d'autres. Il peut être judicieux de jeter un petit coup d'œil sur la documentation, si nécessaire.

5.5.9 Opérations de décalage et de rotation

Ces opérations mettent simplement en œuvre l'opérande de décalage pour effectuer les décalages et rotations sur les bits contenus dans les registres du CPU. Si on veut juste faire un décalage ou une rotation, un utilise l'instruction `MOV`. Par exemple `MOV R0, R0, LSL #2` effectue un décalage de deux bits vers la gauche du registre `R0`.

5.5.10 Opérations de contrôle de flux

L'exécution des instructions d'un programme se déroule principalement séquentiellement. Cependant, ce flux séquentiel peut être interrompu pour effectuer des appels de fonctions ou des branchements (figure 5.16).

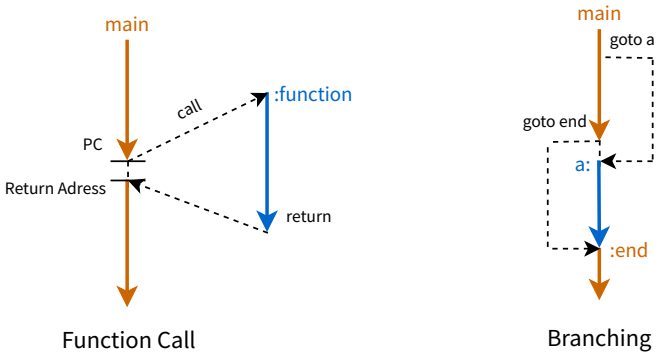


FIGURE 5.16 – Contrôle de flux

Appel de fonctions

En C/C++, deux techniques cohabitent pour l'appel de fonction, l'appel direct et l'appel indirect. L'appel direct utilise simplement le nom de la fonction, son adresse, lors de son appel. L'appel indirect utilise, quant à lui, une variable contenant l'adresse de la fonction à appeler. Cette variable est connue sous le nom de pointeur de fonction en C et de méthode virtuelle en C++.

Les deux jeux d'instructions ARM et Thumb-2 proposent une instruction pour chacune de ces techniques.

— **BL <label>** : appelle la fonction identifiée par l'étiquette <**label**>. L'adresse de retour est stockée dans le registre LR. Avec cette instruction, le processeur effectue un saut relatif par rapport à la valeur actuelle de son compteur ordinaire (registre PC) pour appeler la fonction spécifiée avec l'étiquette <**label**>. Le déplacement, offset entre l'adresse de l'instruction BL et celle du début de la fonction, est calculée lors de la génération de l'application. La plage de cette offset dépend du jeu d'instructions utilisé, ARM ±32MiB, Thumb-2 ±16Mib et Thumb ±4MiB.

```
1 extern void function(void);
2
3 function();
```

```
1 bl    function
```

— **BLX Rm** : appelle la fonction identifiée par le contenu du registre Rm. L'adresse de retour est stockée dans le registre LR. Avec cette instruction, le processeur effectue un saut absolu. Le contenu du registre Rm est simplement chargé dans le compteur ordinal (registre PC).

```
1 extern void function(void);
2
3 void (*fnct)(void) = function;
```

```
4 fnct();
```

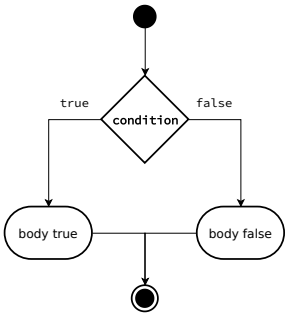
```
1 ldr   r0, =fnct+1
2 ldr   r0, [r0]
3 blx   r0
```

Pour continuer l'exécution du code avec le jeu d'instructions Thumb-2, l'adresse utilisée pour un appel indirect d'une fonction doit impérativement être impaire. Le code étant toujours aligné sur des adresses paires, il suffit d'incrémenter l'adresse de la fonction par 1 pour conserver l'exécution du code en Thumb-2.

Branchements

En C/C++, les branchements sont en général contrôlés par les instructions "**`if`**/**`else`**" et "**`switch`**". Le choix de la branche à exécuter dépend de l'évaluation d'une condition ou d'une valeur (figure 5.17).

```
if (condition) {
 // body true
} else {
  // body false
}
```




```
switch (value) {
case value_1:
   // body value 1
   break;
case value_2:
   // body value 2
   break;
default:
   // body default
   break;
}
```

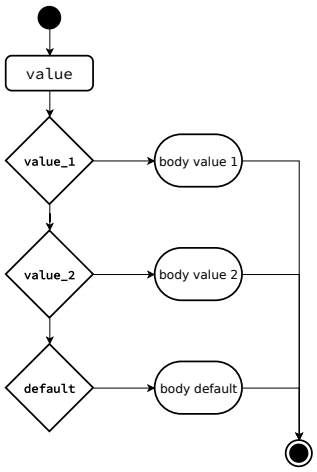


FIGURE 5.17 – Branchements

En langage assembleur, les branchements conditionnels s'effectuent en deux étapes, premièrement l'évaluation de la condition puis le branchement. L'évaluation de la condition est reflétée dans les fanions du registre APSR, ceci suite à l'exécution d'une opération arithmétique ou logique, d'une comparaison ou d'un test. Lors l'exécution de l'opération de branchement, l'instruction évalue les fanions, la condition <cond> (figure 5.6), et effectue le branchement si la condition est remplie. Si la condition n'est pas satisfaite, le CPU poursuit l'exécution des instructions placées à la suite de l'instruction de branchement (voir les exemples ci-dessous).

Comparaisons et tests

En ajoutant le suffixe "S" aux instructions, le CPU reflète, en plus de stocker le résultat de l'opération dans le registre Rd, la mise à jour des fanions dans le registre APSR. Pour éviter de modifier le contenu des registres du CPU, les deux jeux d'instructions ARM et Thumb-2 proposent des instructions pour comparer des valeurs et tester l'état de bits contenus dans les registres sans les modifier.

- **CMP Rn, <Sop>** : compare le contenu du registre Rn avec le contenu de l'opérande de décalage <Sop>. Les fanions reflètent le résultat de la soustraction de l'opérande de décalage <Sop> au registre Rn (Rn - Sop). Hormis les fanions, aucun registre n'est affecté par cette instruction.
- **TST Rn, <Sop>** : effectue un "et" logique entre le contenu du registre Rn et le contenu de l'opérande de décalage <Sop> (Rn & Sop) et reflètent le résultat dans les fanions. Hormis les fanions, aucun registre n'est affecté par cette instruction.
- **TEQ Rn, <Sop>** : effectue un "ou-exclusif" logique entre le contenu du registre Rn et le contenu de l'opérande de décalage <Sop> (Rn ^ Sop) et reflètent le résultat dans les fanions. Hormis les fanions, aucun registre n'est affecté par cette instruction.

Branchements conditionnels

Les deux jeux d'instructions ARM et Thumb-2 proposent une instruction pour effectuer des branchements conditionnels.

- **B<cond> <label>** : réalise un branchement vers l'étiquette <**label**> si la condition <cond> est satisfaite. L'instruction évalue le contenu des fanions du registre APSR selon la table présentée au chapitre "Exécution conditionnelle" (figure 5.6). Si la condition est vraie, le processeur effectue un saut relatif par rapport à la valeur actuelle de son compteur ordinaire (registre PC) pour exécuter les instructions placées à la suite de l'étiquette <**label**>. Le déplacement, offset entre l'adresse de l'instruction "B<cond>" et celle correspondante à l'étiquette, est calculé lors de la génération de l'application. La plage de cette offset dépend du jeu d'instructions utilisé, ARM ±32MiB, Thumb-2 ±16Mib et Thumb -252B à +256B.

L'assembleur permet l'utilisation de label "xf" et "xb", où "x" est un nombre entier. Ces labels permettent d'effectuer des branchements vers le label "x" le plus proche en cherchant en avant (*forward*) pour "f" ou en arrière (*backward*) pour "b".
Cela signifie que dans un code assembleur, l'instruction "b 1f" recherchera le premier "1" placé après l'instruction "b 1f" et que l'instruction "b 1b" recherchera le premier "1" placé avant l'instruction "b 1b".

Branchements inconditionnels

Les deux jeux d'instructions ARM et Thumb-2 proposent deux instructions pour effectuer des branchements inconditionnels.

- **B <label>** : réalise un branchement vers l'étiquette <**label**>. Le processeur effectue un saut relatif par rapport à la valeur actuelle de son compteur ordinaire (registre PC) pour exécuter les instructions placées à la suite de l'étiquette <**label**>. Le déplacement, offset entre l'adresse de l'instruction "B" et celle correspondante à l'étiquette, est calculé lors de la génération de l'application. La plage de cette offset dépend du jeu d'instructions utilisé, ARM ±32MiB, Thumb-2 ±16Mib et Thumb ±2MiB.
- **BX Rm**: réalise un branchement à l'adresse contenue dans le registre Rm. Avec cette instruction, le processeur effectue un saut absolu. Le contenu du registre Rm est simplement chargé dans le compteur ordinal (registre PC).

Pour rappel, l'adresse contenue dans le registre Rm doit être impair pour continuer l'exécution du code avec le jeu d'instructions Thumb-2.

Exemples

Le premier exemple ci-dessous présente la réalisation de l'instruction **if**/**else** en assembleur.

```c
unsigned exemple_1 (int i) {
   unsigned j = 0;
   if (i > 0) {
      j = i % 16;
   } else if (i == 0) {
      j = 0xaa;
   } else {
      j = -i / 16;
   }
   return j;
}
```

```
   .global exemple_1
   .type exemple_1, %function
exemple_1:
    cmp     r0, #0          // comparaison de l'argument i avec 0
    bgt     3f              // if (i >  0) alors va vers 3:
    beq     2f              // if (i == 0) alors va vers 2:
                            // sinon va vers 1:
    // else
1:  rsbs    r1, r0, #0      // i = -i
```

```
10      lsrs    r1, #4          // j = i / 16
11      b       4f
12
13      // if (i == 0)
14  2:  ldr     r1, =0xaa       // j = 0xaa;
15      b       4f
16
17      // if (i > 0)
18  3:  ands    r1, r0, #0xf    // j = i % 16;
19      b       4f
20
21  4:  movs    r0, r1
22      bx      lr
23     .size exemple_1, .-exemple_1
```

Le deuxième exemple présente la réalisation en assembleur de l'instruction **switch** évaluant un nombre entier borné (entre 0 à 3). L'implémentation ci-dessous est très intéressante, car l'évaluation du `cas` à exécuter est totalement indépendante de la valeur de l'expression **switch**.

```
1  int exemple_2 (int i)
2  {
3      int j = 0;
4      switch(i) {
5      case 0:  j=11; break;
6      case 1:  j=8;  break;
7      case 2:  j=25; break;
8      case 3:  j=99; break;
9      default: j=-1; break;
10     }
11     return j;
12 }
```

```
1     .global exemple_2
2     .type exemple_2, %function
3  exemple_2:
4      cmp     r0, #3          // teste si i entre 0 et 3
5      it      hi              // si vrai, prend cette valeur comme index
6      movshi  r0, #4          // si faux, utilise la valeur 4 comme défaut
7      ldr     r2, =lut        // prend l'adresse de base de la table "lut"
8      lsls    r1, r0, #2      // calcule offset dans la table "lut"
9      ldr     r2, [r2, r1]    // charge l'adresse case correspondant
10     bx      r2
11
12 // table avec les "switch-case"
13 // Note: les adresses des différentes "case" sont corrigées de 1 byte
14 // afin de garder le processeur en mode Thumb et éviter ainsi qu'il
15 // ne change pour le jeu d'instructions ARM, lequel n'est pas
16 // supporté un µP Cortex-M4
17 lut: .word case_0+1, case_1+1, case_2+1, case_3+1, case_d+1
18
```

```
19 case_0:
20     ldr   r1, =11
21     b     2f
22 case_1:
23     ldr   r1, =8
24     b     2f
25 case_2:
26     ldr   r1, =25
27     b     2f
28 case_3:
29     ldr   r1, =99
30     b     2f
31 case_d:
32     ldr   r1, =-1
33     b     2f
34
35 2:  movs  r0, r1
36     bx    lr
37    .size exemple_2, .-exemple_2
```

Ce troisième exemple présente une autre réalisation en assembleur de l'instruction **`switch`** évaluant un nombre entier, mais non borné.

```
1  int exemple_3 (int i)
2  {
3     j=0;
4     switch (i) {
5     case -10 : j=-20; break;
6     case   2 : j=-2;  break;
7     case 100 : j=0;   break;
8     case 801 : j=24;  break;
9     default:   j=-1;  break;
10    }
11    return j;
12 }
```

```
1     .global exemple_3
2     .type exemple_3, %function
3  exemple_3:
4      cmp     r0,#-10     // case i == -10
5      beq     case_m10
6      cmp     r0,#2       // case i == 2
7      beq     case_2
8      cmp     r0,#100     // case i == 100
9      beq     case_100
10     ldr     r1,=801
11     cmp     r0,#801     // case i == 801
12     beq     case_801
13     b       case_d      // default
14
```

```
15 case_m10:
16     ldr   r1, =-20
17     b     2f
18 case_2:
19     ldr   r1, =-2
20     b     2f
21 case_100:
22     ldr   r1, =0
23     b     2f
24 case_801:
25     ldr   r1, =24
26     b     2f
27 case_d:
28     ldr   r1, =-1
29     b     2f
30
31 2:  movs  r0, r1
32     bx    lr
33    .size exemple_3, .-exemple_3
```

Là également, il est possible de construire un code générique et indépendant de la valeur de l'expression **switch**. Pour cela, il faut à nouveau utiliser une table de recherche (LUT - *Lookup Table*). La LUT doit contenir, pour chaque cas de l'instruction **switch**, la valeur à tester ainsi que l'adresse des instructions correspondantes au cas. Le code ci-dessous ne présente que la recherche du cas.

```
 1     ldr     r1,  =lut      // adresse de base la table de recherche
 2     ldr     r2,  =lut_end  // adresse de fin
 3     ldr     r12, =case_d+1 // charge l'adresse du traitement par défaut
 4 1:  ldr     r3, [r1]       // charge la valeur à test
 5     cmp     r0, r3         // teste si le valeur correspond au cas
 6     it      eq             // si vrai, ...
 7     ldreq   r12, [r1, #4]  // ... charge l'adresse de traitement du cas
 8     adds    r1, #8         // charge l'adresse de la prochaine entrée
 9     cmp     r1, r2         // teste si toute la table a été examinée
10     bne     1b             // si faux, passe à la prochaine entée
11     bx      r12            // si vrai, traite le cas
12
13 lut:
14     .word -10, case_m10+1
15     .word   2, case_2+1
16     .word 100, case_100+1
17     .word 801, case_801+1
18 lut_end:
```

5.6 Exercices

Quelques exercices pour assimiler la matière de ce chapitre.

5.6.1 Exercice 1 : Utilisation des registres

Décrivez succinctement le rôle des 4 registres spéciaux (SP, LR, PC et APSR).

5.6.2 Exercice 2 : Développement croisé vs natif

Quelles différences existent-ils entre un développement croisé et un développement natif ?

5.6.3 Exercice 3 : Structure d'un module d'assemblage

Quelles sont les sections principales d'un module en langage assembleur ?
Que contiennent ces sections ?
Quelle est l'utilité de la directive `.align` ?

5.6.4 Exercice 4 : Fonction assembleur et interface C/C++

Implémentez en assembleur la fonction C "factorial". Implémentez également le fichier d'en-tête permettant d'appeler la fonction depuis du code C/C++. La récursivité doit impérativement être conservée et le fichier doit être compilable pour la cible des TPs.

```
unsigned long factorial (unsigned n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial (n-1);
}
```

5.6.5 Exercice 5 : Boucle "for"

Implémentez en assembleur la fonction C ci-dessous.

```
struct CheckSum {
    unsigned cks1;
    unsigned cks2;
};

struct CheckSum checksum (const char* msg, unsigned n) {
    struct CheckSum cks = {0, 0};

    for (unsigned i=0; i<n; i++) {
        cks.cks1 += msg[i];
        cks.cks2 += msg[i] * (i+1);
```

```
12      }
13
14      return cks;
15  }
```

Trouvez l'algorithme le plus performant et réalisez-le.

5.6.6 Exercice 6 : Boucle "while"

Implémentez en assembleur la fonction C ci-dessous.

```
1  char* strcpy (char* dest, const char* src) {
2      char* d = dest;
3      while (*dest++ = *src++);
4      return d;
5  }
```

5.6.7 Exercice 7 : Calcul de parité verticale

Implémentez en assembleur la fonction C ci-dessous.

```
1  uint8_t parity (const uint8_t* msg, unsigned n) {
2      char parity = 0;
3      while (n>0) {
4          n--;
5          parity ^=msg[n];
6      }
7      return parity;
8  }
```