CHAPITRE 3 : Les sous-programmes

1. Notions de base

Un sous-programme que l'on notera par la suite de façon abrégée **S/P**, est le codage en langage machine d'une fonction ou d'une procédure du langage algorithmique.

La mise en œuvre des S/P nécessite l'utilisation d'une structure de données de type **pile**, sauf dans quelques cas particuliers qui seront mentionnés dans la suite de ce chapitre.

Cette pile appelée **pile système** est utilisée pour gérer les appels et les retours de S/P.

Comme nous le verrons au paragraphe 3, l'utilisation d'une pile pour gérer les appels et les retours de S/P, permet de gérer simplement les appels de S/P **imbriqués** et de ce fait les S/P **récursifs**.

La pile système est aussi très souvent utilisée, notamment par les compilateurs, pour :

- la transmission des paramètres des S/P
- l'allocation des variables locales des S/P
- la sauvegarde temporaire de registres

A titre d'exemple, l'interface entre le Compilateur C **gcc** et l'assembleur pour les microprocesseurs ARM sera définie au paragraphe 6.

2. Structure et fonctionnement interne d'une pile

Une pile (**stack** en anglais) est une structure de donnée utilisée pour stocker temporairement des données. L'accès se fait par le **sommet** de la pile, c'est donc le dernier élément inséré (ou empilé) qui sera le premier supprimé (ou dépilé).

Cette structure est communément appelée structure **LIFO** (Last in, First out).

2.1. Représentation interne

Dans la représentation interne, on représente une pile par un tableau à une dimension, de N éléments en mémoire, auxquels on accède par un **pointeur**.

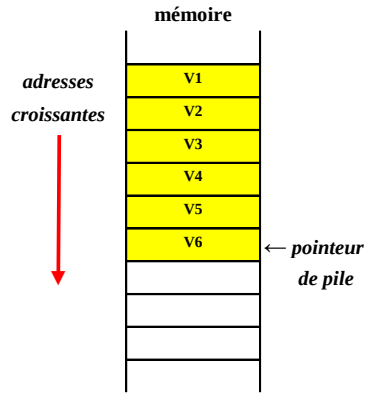
Il existe deux façons d'empiler des éléments dans cette pile :

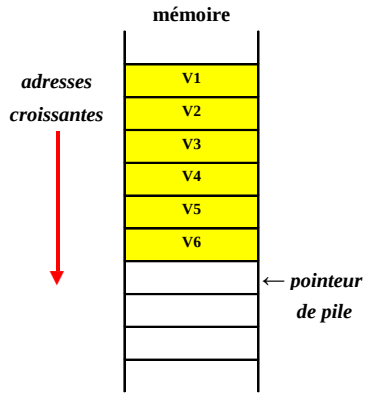
- en commençant au début du tableau et en procédant par **adresses croissantes**
- en commençant à la fin du tableau et en procédant par **adresses décroissantes**

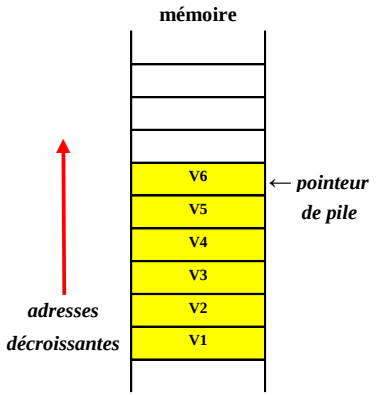
Il existe deux façons de définir le sommet de cette pile :

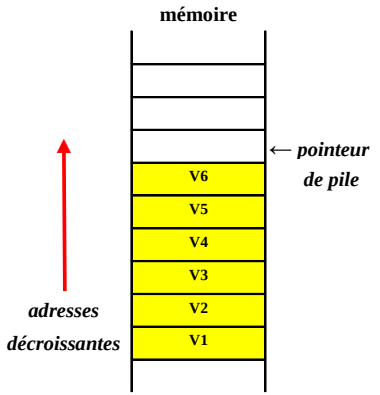
- le pointeur de pile (stack pointer) peut pointer le **dernier élément déposé**
- le pointeur de pile (stack pointer) peut pointer la **première place libre** juste au dessus du dernier élément empilé

Il existe donc quatre modes de représentions possibles d'une pile en mémoire qui sont illustrés par les figures ci dessous :









Remarques :

- certains microprocesseurs imposent une de ces représentations pour la **pile système**, en raison de leur jeu d'instruction. Ce n'est pas le cas des microprocesseurs ARM qui au contraire permettent le choix entre les différents modes.

- l'obligation d'utiliser une de ces représentations pour la **pile système** peut être imposée par certains compilateurs, c'est le cas de **gcc** qui utilise une pile par **adresses décroissantes** et un pointeur de pile sur le **dernier élément empilé**.

- pour les microprocesseurs ARM, il existe un protocole d'appel de procédure standardisé qui impose une **pile système** par **adresses décroissantes** avec un pointeur de pile sur le **dernier élément empilé**.
  On ne peut empiler que des mots de 32 bits contenus dans des registres et les éléments qu'on dépile sont chargés dans des registres. Le pointeur de la pile système est représenté par le registre **R13** qui peut être noté **SP** ou **sp** (stack pointer).

- la création de la **pile système** et l'initialisation du pointeur **R13** sont fait automatiquement par le système d'exploitation dés que le programme est chargé en mémoire pour être exécuté.

2.2. Fonctionnement

La gestion d'une pile nécessite au moins deux primitives :

- **empiler** un élément au sommet de la pile
- **dépiler** l'élément qui est au sommet de la pile

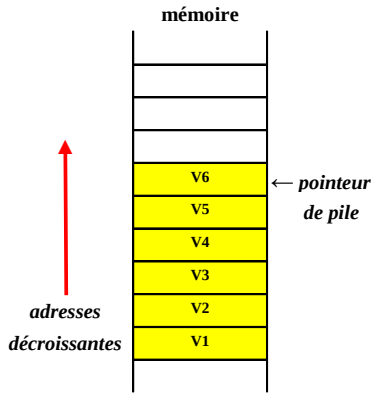
Le jeu d'instructions ARM 32 bits, n'a pas d'instructions spéciales **push** (empiler) et **pull** ou **pop** (dépiler) comme certains microprocesseurs. Ces instructions existent dans le jeu d'instructions Thumb 16 bits.

Les opérations empiler et dépiler sont réalisées par les instructions **LDM** et **STM** (load et store multiple) et les modes de transferts associés et qui sont codés par les 4 suffixes suivants :

- **IA** ≡ **incrémenter** le pointeur de pile **après** l'accès mémoire (Incremnet After)
- **IB** ≡ " " **avant** " (Increment Before)
- **DA** ≡ **décrémenter** le pointeur de pile **après** l'accès mémoire (Decrement After)
- **DB** ≡ " " **avant** " (Decrement Before)

Pour un mode donné de représentation de pile, le choix du mode de transfert de l'instruction LDM ou STM n'est pas simple à retenir.

Par exemple, pour une pile **descendante**, c'est à dire dans laquelle on empile par **adresses décroissantes** et dont le pointeur pointe le **dernier élément empilé** :



empiler ≡ STMDB → store multiple avec "decrement before"

dépiler ≡ LDMIA → load multiple avec "increment after"

Pour faciliter la programmation, il existe 4 suffixes à accoler aux instructions LDM et STM :

- **FD ≡ Full Descending**
- **FA ≡ Full Ascending**
- **ED ≡ Empty Descending**
- **EA ≡ Empty Ascending**

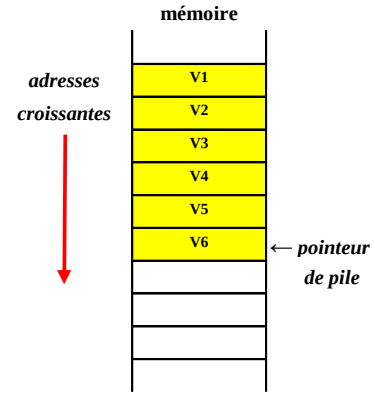
Ascending signifie qu'on empile par **adresses croissantes**

Descending signifie qu'on empile par **adresses décroissantes**

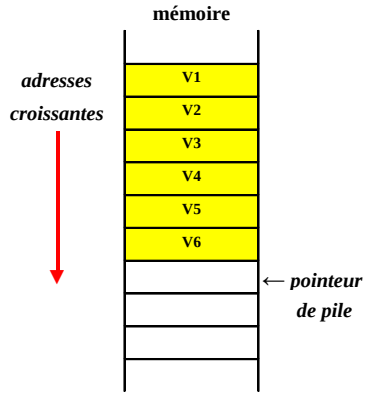
Full signifie que le pointeur pointe le **dernier élément déposé**

Empty signifie que le pointeur pointe la **première place libre** juste au dessus du dernier élément empilé

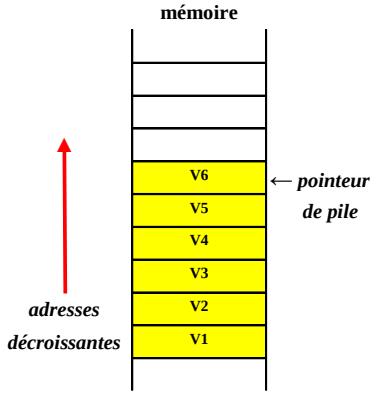
Les opérations empiler et dépiler pour les 4 modes de représentations possibles d'une pile sont donc les suivantes :



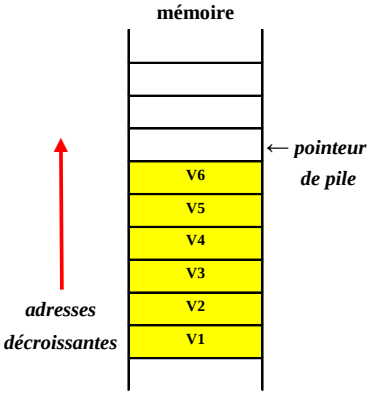
empiler ≡ STMFA SP !, {liste-registres}
dépiler ≡ LDMFA SP !, {liste-registres}



empiler ≡ STMEA SP !, {liste-registres}
dépiler ≡ LDMEA SP !, {liste-registres}



empiler ≡ STMFD SP !, {liste-registres}
dépiler ≡ LDMFD SP !, {liste-registres}



empiler ≡ STMED SP !, {liste-registres}
dépiler ≡ LDMED SP !, {liste-registres}

3. Structure et fonctionnement des S/P

Afin de garantir l'interopérabilité des programmes en langage machine des microprocesseurs ARM, la société ARM à défini un protocole standardisé d'appel de procédure qui spécifie l' **ABI** (Application Binary Interface) de l'architecture ARM :

- Les informations et les directives qui suivent sont conformes à ce protocole mais ne concernent uniquement que le jeu d'instructions ARM 32 bits (pas le Thumb).
- Ce protocole impose que la pile système soit implémentée en mode "**Full Descending**" et que le registre R13 nommé **SP** (ou **sp**), soit utilisé comme pointeur de pile.
  Les éléments empilés ou dépilés doivent toujours être des **mots de 32 bits**.

3.1. Structure d'un S/P

Un S/P est constitué d'un bloc d'instructions et éventuellement d'un bloc de données représentant en langage machine le corps d'une fonction ou d'une procédure du langage algorithmique.

La 1ère instruction du S/P doit être étiquetée avec une étiquette identique au nom donné à la fonction ou à la procédure du langage algorithmique.

3.2. Instructions d'appel et de retour d'un S/P

Appel d'un sous-programme

Syntaxe	*Sémantique*
BL etiquette	**LR ← @ retour ; PC ← @ etiquette**

L'**adresse de retour** est stockée dans le registre **LR** (Link Register) qui est le registre **R14**.

Le PC reçoit l'adresse de la première instruction du S/P.

Remarques :

- Le mode d'adressage utilisé par l'instruction BL est l'adressage relatif à la position courante.
- Le champ adresse contient un déplacement relatif égal à la différence entre l'adresse du S/P étiquetée par **étiquette** et l'adresse de l'instruction d'appel BL.
- Comme les instructions sont codées sur 32 bits et donc stockées à des adresses multiples de 4, le déplacement qui est codé sur 24 bits correspond à une adresse d'instruction multiple de 4. Pour pouvoir être ajouté au PC, il est décalé de 2 positions à gauche pour être converti en une adresse d'octet sur 26 bits. **Avec l'instruction BL, on ne peut appeler que des S/P dont la distance est comprise dans l'intervalle [- 32 Mo , +32 Mo].**

A partir de la **version 5** du jeu d'instructions, on peut appeler un S/P implanté à une adresse quelconque avec l'instruction BLX :

Syntaxe	*Sémantique*
BLX Rn	**LR ← @retour ; PC ← Rn**

On suppose que l'adresse de la première instruction du S/P à été chargée dans le registre **Rn**.

L'**adresse de retour** est stockée dans le registre LR (Link Register) qui est le registre R14.

Le PC reçoit l'adresse de la première instruction du S/P contenue dans registre Rn.

Remarques :

- cette instruction permet d'appeler un S/P en **mode Thumb** (jeu d'instructions 16 bits) si le bit 0 du registre Rn est à 1 (**Rn[0]=1**). Le microprocesseur change de mode et passe du mode ARM au mode Thumb d'où le mnémonique BLX (Branch with Link and eXchange).

- en **mode ARM** (jeu d'instructions 32 bits), les bits 0 et 1 d'une adresse de S/P sont toujours à 0 car les instructions sont stockées à des adresses multiple de 4.
  Le microprocesseur ne change pas de mode et reste en mode ARM.

Retour d'un sous-programme

Si le S/P n'appelle pas d'autres S/P (on dit que le **S/P est terminal**), il est possible de conserver l'adresse de retour dans le registre LR jusqu'à la fin de l'exécution du S/P.

Le retour dans l'appelant peut se faire simplement de la façon suivante :

Syntaxe	*Sémantique*
MOV PC, LR	**PC ← LR**

Si le S/P appelle un ou plusieurs autres S/P , il est n'est plus possible de conserver l'adresse de retour dans le registre LR jusqu'à la fin de l'exécution du S/P.

Il faut sauvegarder l'adresse de retour avant d'appeler tout autre S/P et la restaurer à la fin de l'exécution du S/P, soit directement dans le PC, soit dans le registre LR et utiliser l'instruction ci dessus pour retourner dans l'appelant.

Le paragraphe suivant décrit une méthode couramment utilisée.

Gestion des appels de S/P imbriqués

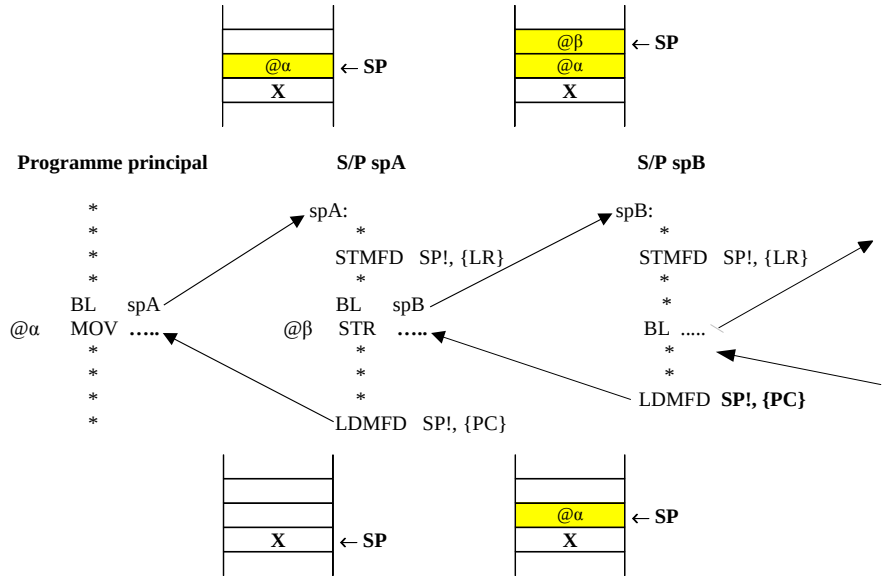
Il est courant qu'un S/P appelle un ou plusieurs autres S/P durant son exécution, lesquels S/P peuvent eux même appeler un ou plusieurs autres S/P et ainsi de suite.

Ces **appels imbriqués** se produisent aussi si le S/P est **récursif**, dans ce cas le S/P s'appelle lui même plusieurs fois de suite.

On pourrait penser à sauvegarder l'adresse de retour dans une **variable locale statique** du S/P.
Cette méthode serait suffisante pour des S/P indépendants mais pas dans le cas de S/P récursifs où à chaque appel l'appelé écraserait l'adresse de retour stockée par l'appelant dans la variable de sauvegarde.

Une méthode classique pour sauvegarder l'adresse de retour avant d'appeler tout autre S/P et la restaurer à la fin de l'exécution du S/P, est d'utiliser la pile système comme le montre la figure ci -dessous.

Certains microprocesseurs n'utilisent pas de registre pour transmettre l'adresse de retour au S/P appelé.
L'instruction d'appel d'un S/P empile directement l'adresse de retour dans la pile système.
Il existe de plus une instruction spéciale qui termine le S/P et fait le retour dans l'appelant en dépilant l'adresse de retour dans le PC.



4. Techniques de passage des paramètres des S/P

4.1. Concepts de base

Dans la plupart des cas, un S/P peut recevoir des informations du programme appelant et/ou lui en fournir. Ces échanges **internes** peuvent être réalisés de 2 façons **:**

- Par le biais de variables **globales** qui sont visibles dans le programme appelant et dans le SP.

  Cette technique est facile à mettre en œuvre mais elle restreint l'utilisation du S/P car les noms (identificateurs) de ces variables globales sont **fixes** .

- Par le biais de variables spéciales appelées **paramètres**.

  Le S/P est défini avec des variables **virtuelles** appelées paramètres **formels**.

  Le S/P est appelé avec des variables **réelles** appelées paramètres **effectifs** qui se substituent aux paramètres **formels** au moment de l'exécution de l'appel du S/P.

  Cette technique est moins facile à mettre en œuvre mais elle permet d'utiliser le S/P avec différents jeux de paramètres **effectifs**.

Dans le cas où on utilise des paramètres, il existe deux modes de transmission des informations par le biais de ces paramètres :

- la transmission **directe,** appelée transmission par **valeur,** qui consiste à transmettre la **valeur** du paramètre **effectif** (contenu de la variable utilisée comme paramètre **effectif**).
- la transmission **indirecte,** appelée transmission par **adresse** (appelée par **référence** dans les langages évolués**),** qui consiste à transmettre l'**adresse** du paramètre **effectif** (adresse de la variable utilisée comme paramètre **effectif**).

Remarques :

Il ne faut pas confondre les **modes de transmission** des paramètres avec le **sens des échanges** d'informations associés à chaque paramètre du S/P.

Le **sens de l'échange** (entrée, mise-à-jour ou sortie) attribué à un paramètre ne peut pas être choisi, il se déduit de la fonctionnalité de ce paramètre.

Le **mode de transmission** peut être choisi en fonction de différents critères quel que soit le sens de l'échange:

- un paramètre en **entrée** sera en général transmis par **valeur** s'il s'agit d'une variable simple ou d'un littéral numérique. Par contre s'il s'agit d'un tableau ou d'un enregistrement, il pourra être transmis par **adresse**, ce qui est plus rapide et plus économique en place mémoire qu'une transmission par valeur.
- un paramètre en **sortie** sera en général transmis par **adresse** quel que soit son type, ce qui permet au S/P de renseigner lui-même ce paramètre.
- Il n'y a pas de règles standardisées, chaque langage voire même chaque compilateur à ses propres règles.

4.2. Méthode de transmission des paramètres par registres

Les échanges d'informations entre le S/P et le programme appelant se font par l'intermédiaire de registres internes du processeur.

Dans le cas d'une transmission par **valeur,** le registre contiendra la **valeur** du paramètre **effectif** (contenu de la variable utilisée comme paramètre **effectif**).

Dans le cas d'une transmission par **adresse,** le registre contiendra l'**adresse** du paramètre **effectif** (adresse de la variable utilisée comme paramètre **effectif**).

Les paramètres **formels** sont les **noms des registres** du processeur utilisés pour la transmission.

Les paramètres **effectifs** sont les **valeurs** ou les **adresses** des variables utilisées comme paramètres **effectifs**, contenues dans ces registres.

Les avantages et inconvénients de cette technique sont les suivants :

- mise en œuvre facile.
- rapidité d'exécution (pas d'accès en mémoire).
- le nombre de paramètres est limité par le nombre de registres du processeur.
- le passage par valeur est complexe et limitatif pour un tableau ou un enregistrement s'il doit être stocké dans plusieurs registres concaténés.

déroulement du processus de la mise en œuvre

a)- dans le programme appelant, avant l'appel du S/P

Les valeurs ou les adresses des paramètres en **entrée** ou en **mise à jour** sont copiées dans les registres correspondants.

Les adresses des paramètres en **sortie** sont copiées dans les registres correspondants. Pour les paramètres en **sortie** transmis par valeur, la valeur retournée sera évidemment copiée dans le registre par le S/P.

b)- dans le S/P

Pour les paramètres en **entrée** ou en **mise à jour** transmis par **valeur**, les valeurs contenues dans les registres correspondants seront utilisées directement pour réaliser le traitement du S/P.

Pour les paramètres en **sortie** transmis par **valeur**, les valeurs résultant du traitement seront copiées dans les registres correspondants.

Pour les paramètres transmis par **adresse**, les adresses contenues dans les registres correspondant seront utilisées comme **pointeurs** pour lire et/ou écrire (suivant le sens de l'échange du paramètre) les informations utilisées dans la réalisation du traitement du S/P.

c)- dans le programme appelant, après le retour du S/P

Pour les paramètres en **sortie** transmis par **valeur**, les valeurs contenues dans les registres correspondants seront copiées respectivement dans les variables utilisées comme paramètres **effectifs**.

4.3. Méthode de transmission des paramètres par la pile système

Les échanges d'informations entre le S/P et le programme appelant se font par l'intermédiaire de la pile système.

On peut considérer que les paramètres sont structurés en **enregistrement** dont ils constituent les différents champs.

Dans le cas d'une transmission par **valeur,** le champ contiendra la **valeur** du paramètre **effectif** (contenu de la variable utilisée comme paramètre **effectif**).

Dans le cas d'une transmission par **adresse,** le champ contiendra l'**adresse** du paramètre **effectif** (adresse de la variable utilisée comme paramètre **effectif**).

Pour accéder aux différents champs d'un enregistrement en mémoire, il faut connaître :

- l'adresse de début du bloc d'octets qui a été alloué pour représenter l'enregistrement.
- pour chaque champ, le déplacement relatif au début du bloc à effectuer pour y accéder.

Les paramètres **effectifs** sont les **valeurs** ou les **adresses** des variables utilisées comme paramètres **effectifs**, contenues dans les champs correspondants de l'enregistrement.

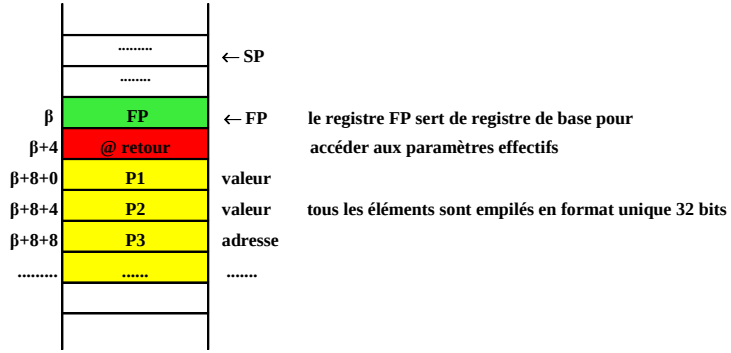
Les paramètres **formels** sont les **déplacements relatifs** à effectuer pour accéder à chaque champ.

L'ordre d'empilement des paramètres peut être l'ordre de leur position dans la liste d'appel du S/P ou l'ordre inverse. L'empilement des paramètres dans l'ordre inverse permet de gérer des S/P dont le nombre de paramètres effectifs peut varier d'un appel à un autre (cas des compilateurs C).Dans ce cas on est sûr que le premier champ contient le premier paramètre, etc …

On pourrait penser utiliser le pointeur de pile système SP pour repérer l'adresse de début du bloc d'octets qui a été alloué pour représenter l'enregistrement. Ce n'est pas raisonnable car le pointeur de pile système va changer de valeur chaque fois que le S/P va empiler ou dépiler des registres, ou appeler un autre S/P.

Il faut donc utiliser un registre différent souvent appelé **FP** (Frame pointer) qui sera initialisé avec une adresse qui restera fixe pendant toute la durée du S/P.

Exemple classique d'organisation de l'extrémité de la pile système en début d'un S/P



Les avantages et inconvénients de cette technique sont les suivants :

- le nombre de paramètres n'est pas limité.
- le passage par valeur des tableaux et des enregistrements n'est pas limitatif.
- mise en œuvre complexe.
- exécution ralentie par les accès en mémoire pour chaque paramètre.

déroulement du processus de la mise en œuvre

a)- dans le programme appelant, avant l'appel du S/P

Les valeurs ou les adresses des paramètres **effectifs** sont empilées pour constituer les différents champs de l'enregistrement.

Pour les paramètres en **sortie** transmis par valeur, la valeur empilée est quelconque car la valeur retournée sera évidemment copiée dans le champ correspondant par le S/P.

b)- dans le S/P

En début du S/P, le registre utilisé comme registre de base pour accéder à l'enregistrement contenant les paramètres, est sauvegardé au sommet de la pile, au dessus de l'adresse de retour du SP, puis initialisé avec l'adresse courante du sommet de pile.

Pour les paramètres en **entrée** ou en **mise à jour** transmis par **valeur**, les valeurs contenues dans les champs correspondants seront utilisées directement pour réaliser le traitement du S/P.

Pour les paramètres en **sortie** transmis par **valeur**, les valeurs résultant du traitement seront copiées dans les champs correspondants.

Pour les paramètres transmis par **adresse**, les adresses contenues dans les champs correspondant seront utilisées comme **pointeurs** pour lire et/ou écrire (suivant le sens de l'échange du paramètre) les informations utilisées dans la réalisation du traitement du S/P.

En fin du S/P, le registre utilisé comme registre de base pour accéder à l'enregistrement contenant les paramètres, est restauré puis le pointeur de pile est positionné sur l'adresse de retour du S/P.

c)- dans le programme appelant, après le retour du S/P

Pour les paramètres en **sortie** transmis par **valeur**, les valeurs contenues dans les champs correspondants seront copiées respectivement dans les variables utilisées comme paramètres **effectifs**.

L'enregistrement contenant les paramètres est ensuite **"éliminé"** de la pile en positionnant le pointeur de pile après le dernier champ de l'enregistrement.

4.4. Méthode de transmission des paramètres de l'ARM

La méthode est définie dans le protocole standardisé d'appel de procédure qui spécifie l' **ABI** (Application Binary Interface) de l'architecture ARM :

- Comme on l'a vu au **§3**, ce protocole impose que la pile système soit implémentée en mode "**Full Descending"** et que le registre R13 nommé **SP** (ou **sp**), soit utilisé comme pointeur de pile. Les éléments empilés ou dépilés doivent toujours être des **mots de 32 bits**.

- Les registres R0, R1, R2 et R3 sont utilisés en priorité pour transmettre les premiers paramètres et la pile système est utilisé en complément si nécessaire, de la façon suivante:

  - si les paramètres pris dans l'ordre de la liste d'appel du S/P, peuvent être stockés dans tout ou partie de ces 4 registres, la transmission se fait par ces registres et la pile système n'est pas utilisée.

  - si ce n'est pas possible, les paramètres sont empilés dans la pile système, dans **l'ordre inverse** de la liste d'appel du S/P, sauf les 4 derniers mots du bloc constitué par les paramètres qui eux sont stockés dans les registres R3, puis R2, puis R1, puis R0.

- Cette méthode se justifie par le constat que la majorité des S/P ont peu de paramètres et que les tableaux et enregistrements sont en général transmis par adresse. Par conséquent le passage par ces registres est dans la plupart des cas suffisant et évite les accès à la mémoire.

- Le protocole spécifie que les registres R0, R1, R2 et R3, étant donné leur fonction, n'ont pas besoin d'être préservés par le S/P. De plus, R0 et R1 sont utilisés pour retourner les valeurs des fonctions.

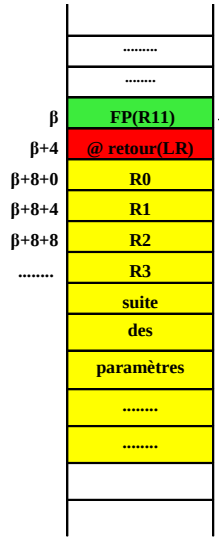
- Les registres R12 à R15 ont chacun une fonction et une utilisation particulière qui ont déjà été décrites dans les paragraphes précédents.

- Le registre R11 est en général utilisé comme registre **FP** (Frame Pointer) par les compilateurs.

- Par contre les registres R4 à R10 sont des registres banalisés. Ceux qui sont utilisés dans le S/P doivent être sauvegardés dans la pile système en début du S/P et restaurés en fin du S/P.

Exemple d'organisation de l'extrémité de la pile système en début d'un S/P

Le cas le plus complexe est celui d'un S/P **non terminal**, avec plusieurs paramètres transmis pour partie dans les registres R0 à R3 et pour partie dans la pile système.



le registre FP sert de registre de base pour accéder aux paramètres effectifs

La séquence d'instructions à exécuter en début du S/P est la suivante :

```
nom_du_S/P:
        STMFD   SP!, {R0-R3}
        STMFD   SP!, {LR, FP}
        MOV     FP, SP
```

La séquence d'instructions à exécuter en fin du S/P est la suivante :

```
        MOV     SP, FP
        LDMFD   SP!, {LR, FP}
        ADD     SP, SP, #4*4
        MOV     PC, LR
```


Remarques :

- L'empilement des registres R3 à R0 présente un triple intérêt :

  - uniformiser l'accès aux paramètres qui sont ainsi tous dans la pile système

  - sauvegarder leurs contenus car ces 4 registres vont être modifiés dés l'appel du premier S/P

  - ces 4 registres pourront ainsi être utilisés en priorité comme registres de travail car si on utilise d'autres registres il faudra les sauvegarder dans la pile système

- Si tous les paramètres contiennent dans les registres R0 à R3 et que le S/P n'est pas terminal cette organisation est encore intéressante pour les deux dernières raisons données ci dessus.

- Cette organisation peut être simplifiée voire inutile suivant la nature du S/P :

  - si le S/P est terminal, le registre LR n'a pas besoin d'être sauvegardé

  - si le S/P est terminal, si tous les paramètres contiennent dans les registres R0 à R3 et que le S/P est simple, l'utilisation de la pile système ne sera pas nécessaire.

5. Allocation des variables locales d'un S/P dans la pile système

5.1. Concept de base

Excepté dans des cas très particulier, les variables **locales** d'un S/P ne sont utilisées que pendant l'exécution du S/P.

Quand le S/P est terminé ces variables peuvent **"disparaître"** car comme elles ne sont visibles que dans le S/P, elles ne sont pas utilisables à l'extérieur du S/P.

D'où l'idée mise en œuvre dans la plupart des compilateurs, d'allouer les variables locales d'un S/P dans la pile système avant de commencer l'exécution des instructions du S/P.

En fin de S/P, les variables locales sont **"éliminées"** de la pile.

Par exemple, les compilateurs C allouent dans la pile système toutes variables locales des fonctions (y compris le **main**) qui ont l'attribut **auto** ou qui n'ont pas d'attribut (**auto** est l'attribut par défaut).

Par contre les variables locales d'une fonction C qui ont l'attribut **static** sont allouées dans le segment de mémoire des données statiques avec les variables globales et les littéraux chaînes de caractères.
Ces variables sont **rémanentes** et peuvent être réutilisées dans les appels successifs de la fonction.

5.2. Allocation de l'espace mémoire

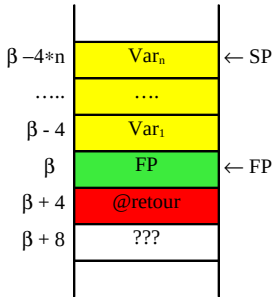
	Syntaxe	*Sémantique*
STR	**FP, [SP, # -4]!**	**SP ← SP - 4; SP↑ ← FP;**
MOV	**FP, SP**	**FP ← SP;**
SUB	**SP,SP, # taille**	**SP ← SP - taille**

Le registre **FP** (R11) qui va servir de registre de base pour accéder aux variables locales dans la pile système, est sauvegardé dans cette même pile.

Puis **FP** est initialisé avec le contenu courant de **SP**, auquel est ensuite retranchée la taille en octets de l'espace mémoire nécessaire aux différentes variables allouées.

Si on utilise l'organisation de la pile système présentée au paragraphe précédent, les 2 premières instructions ne doivent pas être faites car le registre **FP** est déjà initialisé pour servir de registre de base pour accéder aux paramètres dans la pile système.

Le contenu de la pile système, après l'allocation des variables locales est le suivant :



5.3. Libération de l'espace mémoire

	Syntaxe	*Sémantique*
MOV	**SP, FP**	**SP ← FP;**
LDR	**FP, [SP], # 4**	**FP ← SP↑; SP ← SP + 4;**

Le SP est restauré avec l'adresse contenue dans FP

Puis FP et SP sont restaurés avec la valeur qu'ils contenaient en début du S/P.

Si on utilise l'organisation de la pile système présentée au paragraphe précédent, ces 2 instructions ne doivent pas être faites.

6. Interface du langage C avec le langage d'assemblage des ARM

Le passage des paramètres se fait par **valeur,** excepté pour le type **tableau** :

- un entier (**int**) ou un réel (**float**) occupe 32 bits,
- un caractère (**char**) occupe 32 bits (seul l'octet de faible poids est significatif),
- un réel (**double**) occupe 64 bits,
- un pointeur occupe 32 bits.

Si une fonction C retourne une valeur simple, elle sera contenue dans :

R0 pour les types **int**, **char** et **float.**

R0 ‖ R1 pour le type **doubl**e.

R0 pour un pointeur quelque soit son type.

Les registres **R0**, **R1**, **R2** et **R3** n'ont pas besoin d'être **préservés** s'ils sont modifiés dans le corps de la fonction C.

Tous les autres registres doivent être **préservés** puis **restaurés** avant le retour de la fonction C s'ils sont modifiés dans le corps de celle-ci.

Rappels :

- dans le cas d'un tableau, le paramètre transmis est un **pointeur** sur le premier élément du tableau (passage par adresse).
- dans le cas d'un enregistrement, il est plus **efficace** de transmettre un pointeur contenant l'adresse de la structure plutôt que son contenu.

Exemple 1 : **codage d'une fonction en langage d'assemblage ARM**

Algorithmes

fonction somme (**entrée** a <Entier>, **entrée** b <Entier>)
retourne <Entier>

début

retourner(a + b);

fin

programme exemple1

glossaire

x <Entier> : ……;
y <Entier> : ……...…;
res <Entier> : ……....;

début

…….
res ← somme (x, y);
……

fin

Algorithmes détaillés

S/P somme

glossaire

paramètres :

R0 ≡ a, entier relatif 32 bits, passage par **valeur**

R1 ≡ b, entier relatif 32 bits, passage par **valeur**

retour de la fonction :

dans R0, entier relatif 32 bits, passage par **valeur**

début

R0 ← R0 add R1

fin

programme exemple1

glossaire

x : entier relatif 32 bits, ……...;
y : entier relatif 32 bits, ……..…..;
res : entier relatif 32 bits, ……..…...;
.Px : pointeur 32 bits, contient l'adresse de x;
.Py : pointeur 32 bits, contient l'adresse de y;
.Pres : pointeur 32 bits, contient l'adresse de res;

début

…….
R0 ← .Px
R0 ← R0↑
R1 ← .Py
R1 ← R1↑
appel somme
R1 ← .Pres
R1↑ ← R0
……

fin

Exemple 2 : **codage d'une procédure en langage d'assemblage ARM**

Algorithmes

procédure permuter (**mise à jour** a <Entier>, **mise à jour** b <Entier>)

glossaire

aux <Entier> : variable locale pour la permutation;

début

aux ← a;
a ← b;
b ← aux;

fin

programme exemple2

glossaire

x <Entier> : ……;
y <Entier>: ……..…;

début

…….
permuter (x, y);
……
fin

Algorithmes détaillés

S/P permuter

glossaire

paramètres :

R0 ≡ @a, pointeur sur entier relatif 32 bits, passage par **adresse**

R1 ≡ @b, pointeur sur entier relatif 32 bits, passage par **adresse**

variable locale :

R2 ≡ aux, entier relatif 32 bits,

début

R2 ← R0 ↑

R3 ← R1 ↑

R0 ↑ ← R3

R1 ↑ ← R2

fin

programme exemple2

glossaire

x : entier relatif 32 bits, …….;
y : entier relatif 32 bits, ……..…..;
.Px : pointeur 32 bits, contient l'adresse de x;
.Py : pointeur 32 bits, contient l'adresse de y;

début

…….

R0 ← .Px
R1 ← .Py
appel permuter

……

fin

Exemple3 : **codage d'une procédure non terminale en langage d'assemblage ARM**

Algorithmes

procédure traiter (**entrée** a <Entier>, **entrée** b <Entier>, **sortie** s <Entier>)

début

….....
appel .....
.....
s ← a + b;
…...

fin

programme exemple3

glossaire

x <Entier> : ……;
y <Entier> : ……..….;
som <Entier> : ……..….;

début

…….

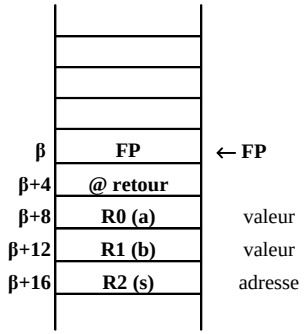
traiter (x, y, som);

……

fin

Algorithmes détaillés

Schéma de la pile système



S/P traiter

glossaire

paramètres :

A : position relative du paramètre a = 8+0 , passage par **valeur**
B : position relative du paramètre b = 8+4 , passage par **valeur**
S : position relative du paramètre s = 8+8 , passage par **adresse**

début

empiler(R0-R2)
empiler(FP, LR)
FP ← SP
….....
R0 ← (FP + #A)↑
R1 ← (FP + #B)↑
R0 ← R0 add R1
R2 ← (FP + #S)↑
R2 ↑ ← R0
.....
SP ← FP
dépiler(FP, LR)
SP ← SP add # 4*3
PC ← LR

fin

programme exemple3

glossaire

x : entier relatif 32 bits, …….;
y : entier relatif 32 bits, ……..…..;
som : entier relatif 32 bits, ……..……...;
.Px : pointeur 32 bits, contient l'adresse de x;
.Py : pointeur 32 bits, contient l'adresse de y;
.Psom : pointeur 32 bits, contient l'adresse de som;

début

…….

R0 ← .Px
R0 ← R0 ↑
R1 ← .Py
R1 ← R1 ↑
R2 ← .Psom

appel traiter

……

fin