Les procédures en langage d'assemblage

- **Comment traduire en langage d'assemblage la définition et les appels des procédures et fonctions des langages de haut niveau ?**

Les procédures/fonctions

- Une procédure (ou fonction) en langage de haut niveau :
  - Un nom
  - Des paramètres : Par valeur (ou donnée) ou par adresse (ou résultat)
  - Des variables locales
  - Un corps : liste d'instructions

Pour les fonctions : une valeur retournée

Introduction

- Exemple en C :

```
void B (int a, int *b)
    {
    int z=2 ;
    int t = 1 ;
    *b= z+a +t ; /*affectation de la case memoire dont l'adresse est dans le 2eme paramètre */
    }
 main()
 {
 int x=1;
 int y;
 B(x,&y); /* &y: adresse de y */
 }
```


1. Première approche

- **L'appel de procédure**
- Pas de variable locale, ni de paramètre

En C :	**En assembleur**
Void Proc() {*instructions*} ;	Etiq_Proc : traduction des *instructions* de Proc branchement à ad-retour
main() { … Proc() ; …. }	main : … Branchement à Etiq_Proc ad-retour : ….

Le problème du retour

- Il faut revenir à la fin de la procédure à l'instruction qui suit le branchement à etiq_Proc (en ad-retour).
- **Si plusieurs appels de la procédure**

  Il faut mémoriser quelque part cette adresse de retour (différente pour chaque appel)
- On peut le faire dans un registre ou à un endroit prédéfini en mémoire (comme une variable locale particulière de la procédure)
- Cela peut être fait dans l'appelant ou dans l'appelé
- Dans l'appelant c'est simple, puisque l'adresse de retour (l'adresse de l'instruction de branchement +/- n) est dans PC
- C'est en général réalisé par une instruction spécialisée

  Exemple: C'est le cas du SPARC (call, ret) ou du ARM (bl)

L'appel en ARM

- **BL etiquette_proc**
- L'adresse de la case mémoire qui suit l'instruction ***BL*** est sauvegardée dans le registre ***r14*** (appelé LR: LINK Register)
- A la fin de la procédure il suffit de mettre le contenu de ***r14*** dans ***PC***:

  Mov PC, LR

Exemple en ARM

En C:

ProcA () ;
{corps de ProcA}
Main() ;
{…ProcA ;…ProcA ;}

En assembleur :

Texte

main : …
bl ProcA @sauvegarde de Adr_ret1 (adresse de retour) dans lr(r14)
Adr_ret1: …
bl ProcA @sauvegarde de Adr_ret2 dans lr
Adr_ret2: ...
ProcA : corps de ProcA
Mov PC, LR

Problème des appels imbriqués

- Dans le cas d'appel imbriqué, le registre de sauvegarde LR est écrasé par l'appel suivant
- Il faut sauvegarder l'adresse de retour dans une case mémoire réservée pour cela
- Cette sauvegarde peut être fait dans l'appelé systématiquement ou dans l'appelant si nécessaire

Exemple en ARM

En C:

ProcA () ;
{...; ProcB;...}

ProcB () ;
{... }

Main() ;
{…; ProcA;.... }

En assembleur :

```
SauvAdRetA: .word 0
main : …
        bl ProcA  @sauvegarde de PC dans lr (r14)
        …
ProcA : ...
        ldr r1,=SauvAdRetA
        str lr, [r1]    @sauvegarde adresse retour de A en mémoire
        bl ProcB   @sauvegarde de PC dans lr (r14)
         ....
        ldr r1,=SauvAdRetA  @récupération de l'adresse de retour de A
        ldr PC,[r1]              @retour de procédure
```


Les variables locales

- On peut :
  - Soit les déclarer en zone data comme des variables globales
  - Soit utiliser des registres pour optimiser
  - Dans ce cas, il faut faire attention à l'utilisation possible des registres par l'appelant

Les registres (ou temporaires)

- L'appelant est peut être en train d'utiliser des registres.
  Si l'appelé veut utiliser ces registres, il faut les sauvegarder avant de les utiliser, puis les restaurer
- Plusieurs possibilités :
  - Fait dans l'appelant avant l'appel. Il sauvegarde tous les registres utilisés. Il restaure après le retour.
  - Fait dans l'appelé. Il sauvegarde les registres qui seront utilisés. Il restaure avant le retour.
- La deuxième solution est plus efficace (on ne sauvegarde que le strict nécessaire).
- Cette sauvegarde est effectuée dans une zone mémoire spécifique allouée statiquement

Les paramètres

- Un paramètre :
  - Soit par valeur (paramètre donnée)
  - Soit par adresse (paramètre résultat): ce qui va être passé en paramètre est l'**adresse d'une variable**, on pourra ainsi modifier la variable à l'intérieur de la procédure
- **Remarque:** Pour une fonction, la valeur retournée peut être vue comme un paramètre résultat de plus.
- Exemple:
  une fonction ***int fct() ;***

  équivalent à une procédure : ***void proc (int *val_ret) ;***

Paramètre par valeur

- On calcule le la valeur du paramètre effectif dans l'appelant
- L'appelant le stocke dans un endroit convenu :
  - case mémoire (allouée statiquement)
  - ou plus rapide : un registre
- L'appelé récupère cette valeur à l'endroit convenu
- Prenons comme convention que les paramètres sont stockés dans les registres r0, r1, r2 ....

Exemple de paramètres par valeur

```
Main()                          void Proc(int a, int b)
{ int y ;                           { int x ;
… Proc(3, y) ; … }                    x= a +b; }
```

```
x: .word 0
y: .word 0
main : …
    mov r0, #3   @ 1er paramètre dans r0
    ldr r1, =y   @r1 contient l'adresse de y
    ldr r1, [r1] @r1 contient la valeur de y :  2eme paramètre dans r1
    bl etiq_proc
    …
etiq_proc : ldr r3, =x     @r3 contient l'adresse de la variable locale x
            add r0, r0, r1 @r0= a+b
            str r0, [r3]   @x=a+b
            mov PC, LR     @retour
```


Exemple de paramètre par résultat

```
Main() ;                    void Proc( int* a) ;
  { int t = 2  ;                { int x =5;
  …Proc(&t) ; …                   *a=x;
  }                              }
```

```
t:  .word 2
x:  .word 5
main : …
        ldr r0, =t      @ r0 contient l'adresse de t
        bl etiq_proc
        …
etiq_proc :  ldr r1,[r1]   @ r1 contient la valeur de x
             str r1, [r0]  @ *a=x
             mov PC, LR @ retour
```


2. Cas à problème la récursivité

- Tout ce que l'on vient de dire ne tient pas si on doit traduire des procédures récursives (elle s'appelle elle-même).
- Cela peut être aussi de la récursivité indirecte : A appelle B qui appelle A
- Exemple de procédure récursive :

```
int fact (int n)
{
int res ;
if (n==1)
    res=1 ;
else
    res=n*fact(n-1) ;
return (res) ;
}
```

- Exemple : fact (3) appelle fact (2) qui appelle fact (1)

Les appels récursifs

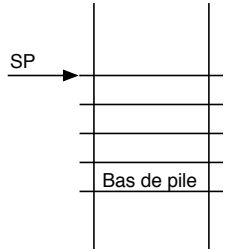
- Le registre (ou la case mémoire réservée statiquement) où est sauvegardé l'adresse de retour est écrasé au deuxième appel.

  Il n'y a qu'un emplacement en mémoire pour la variable locale *res*.
- On ne sait pas à l'avance combien d'appels seront fait
- Il faut donc allouer de la mémoire **dynamiquement** (au moment de l'exécution du programme) au moment de l'appel pour sauvegarder l'adresse de retour, les paramètres, les variables locales, et les temporaires.

- On invente la pile

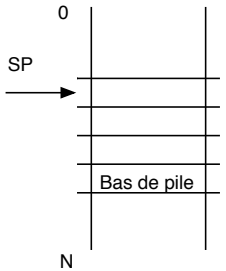
La pile, principes

- Pile d'assiette: on peut poser ou prendre une assiette sur la pile
- Pointeur de pile ; en général on spécialise un registre de calcul (SP : Stack Pointer)
- Initialisation à un endroit en mémoire (réservation d'un bout de la mémoire pour la pile)
- Si dépassement le système peut produire une erreur : *stack overflow* (protection matérielle des accès mémoire)



Convention de gestion de pile

- adresse croissante ou décroissante à l'empilement ?
- SP pointe sur dernier élément ou sur première case libre ?
- On choisit arbitrairement : empilement adresse décroissante , SP pointe sur dernier élément
- Avec cette convention:
  - Empilement de élément:
    SP= SP-1 (ou –4 si élément sur 4 octets et mémoire : tableau d'octets)
    Mem[SP]= élément
  - Dépilement dans élément:
    élément=Mem[SP]
    SP=SP+1



Instruction de gestion de la pile

- Dans de nombreux processeurs, il existe des instructions spécialisées pour gérer la pile

- Exemple le 68000 :
  - l'adresse de retour est sauvegardée sur la pile par l'instruction de branchement à une procédure:
  - JSR etiquette_proc : SP= SP-4, Mem[SP]= adresse qui suit le JSR, PC=Etiquette_proc
  - RTS : PC=Mem[SP], SP=SP+1

- Dans le ARM, BL ne gère pas la pile, c'est au programmeur de sauvegarder/restaurer le contenu de ce registre dans la pile
  - S'il n'y a pas d'appel imbriqué, c'est inutile

Exemple de sauvegarde de l'adresse de retour dans le ARM

- On choisit arbitrairement **r13** pour **SP**
- Conventions: Empilement adresse décroissante + Pointeur sommet case pleine
- **Au début de la procédure:**
  - **add sp, sp, #-4**
  - **str lr, [sp]**
- **Le retour de la procédure:**
  - **ldr pc, [sp]**
  - **Add sp, sp, #4** *@attention instruction non exécutée puisque on modifie PC*
  - On fait donc
  - **ldr pc , [sp], #4**

Dans le ARM

- Empilement multiple avec convention de pile diverses :
  instructions STM, LDM
  IA, IB, DA, DB pour increment/décrément after/before (après/avant)
  Autre dénomination :
  FA, FD, EA, ED pour full/empty ascending/descending (en référence au stockage : ascending : adresse croissante quand la pile augmente : la pile évolue vers les adresses hautes)
- Exemples:
- Empilement adresse croissante :
  - Pile à sommet case pleine : STMIB = STMFA = empiler, LDMDA = LDMFA = depiler
  - Pile à sommet case vide : STMIA = STMEA = empiler, LDMDB = LDMEA = depiler
- Empilement adresse décroissante :
  - Pile à sommet case pleine : STMDB (decrementer avant) = STMFD = empiler, LDMIA = LDMFD = depiler
  - Pile à sommet case vide : STMDA (decrémenter apres) = STMED = empiler, LDMIB = LDMED = depiler

Particularités des sauvegardes/ chargements multiple du ARM

- La liste de registre est encodée dans l'instruction sous la forme d'un vecteur de 17 bits
- Les registres sont stockés (ou chargés depuis) en mémoire de telle sorte que les **registres de numéros croissant** occupent des **adresses croissantes**
- Attention le pointeur de pile ne peut pas apparaître dans la liste des registres à stocker/charger si il est modifié
- Exemple :
  - STMFD SP !, {r1,r2 , r14}
    Empile d'abord r14 puis r2 puis r1 (ordre des numéros de registres décroissant puisque les adresses décroissent lors des empilements successifs)
  - **Le ! spécifie que SP doit être changé à la fin de l'empilement**
  - **LDMFD SP !, {r1,r2 , r14} dépile d'abord r1 puis r2 puis r14**

Exemple d'utilisation de STM et LDM

- Convention : empilement adresse décroissante , SP pointe sur dernier élément
- Empilement : **STMFD SP !, {r14}**
- Dépilement : **LDMFD SP !, {PC}** *@On dépile directement dans PC*

EtiqProc : STMFD SP !, {r14} *@empilement adresse de retour*

…

LDMFD SP !, {PC} *@dépilement adresse de retour*

Exemple d'appels imbriqués

- **En C:**

ProcA() ; ProcB() ; ProcC() ; Main() ;
{ … ; ProcB ;….} { … ; ProcC ;….} { …} {…; ProcA ;…}

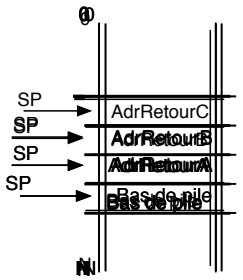
En assembleur :

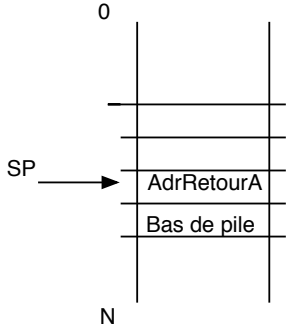
main : …
bl EtiqProcA
…

ProcA : STMFD SP !, {LR}
…
bl EtiqProcB
…
LDMFD SP !, {PC}

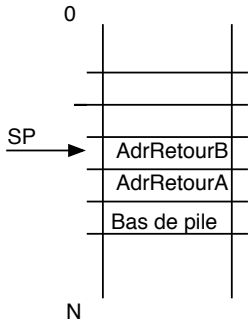
ProcB : STMFD SP !, {LR}
…
bl EtiqProcC
…
LDMFD SP !, {PC}

ProcC : STMFD SP !, {LR} @Non nécessaire ici : on pourrait ne pas sauvegarder
…
LDMFD SP !, {PC} @ou Move PC, LR s'il n'y a pas eu de sauvegarde





Après l'empilement dans ProcA



Après l'empilement dans ProcB

Gestion des variables locales sur la pile

- **Même problème que pour l'adresse de retour**
  - On peut utiliser des registres mais leur nombre est limité
  - Il faut pouvoir les sauvegarder en mémoire.
  - On ne connaît pas à l'avance le nombre d'appels de la procédure
  - Il est donc impossible de réserver au moment de la compilation les cases mémoires nécessaires aux variables locales

- **Même idée : on stocke les variables locales sur la pile**
  - Chaque variable locale aura une adresse calculée par rapport à SP
  - On réserve dans la pile la place nécessaire aux variables locales en entrant dans la procédure
  - On initialise ou non les variables locales
  - L'ordre des variables sur la pile dépend de l'ordre de déclaration et d'une convention arbitraire.

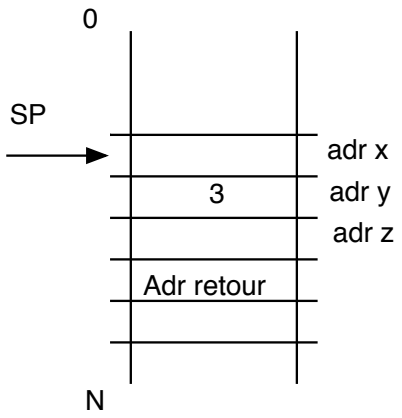
Exemple en ARM

- **Convention**: toutes les variables locales sur la pile (empilement en ordre inverse)
- Proc() ;
{int x; int y=3 ; int z ;
… ; x=5 ;…}

```
Etiq_proc: STMFD SP !, {LR} @empilement adresse de retour
           SUB SP, SP, #12   @reservation 12 case sur la pile pour x,y, z
                             @SP pointe sur x ; z à l'adresse SP+8 et y : SP+4
           MOV r0, #3
           STR r0, [SP,#4]   @Initialisation de y
           …
           MOV r0, #5
           STR r0, [SP]      @x=5
           …
           ADD SP, SP, #12   @Libération place des variables locales
           LDMFD SP !, {PC}  @dépilement adresse de retour
```

Etat de la pile

- **Après l'initialisation de y**



Gestion des paramètres dans la pile

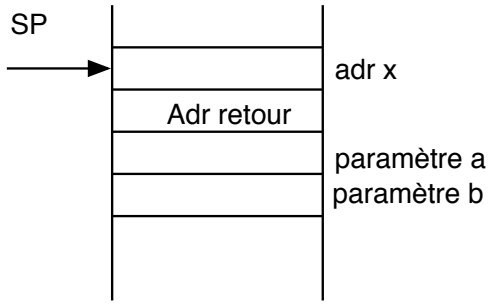
- Toujours même idée : les paramètres sont stockés dans la pile
  - Avant l'appel il faut calculer la valeur des paramètres « effectifs »
  - Ensuite il faut empiler ces valeurs pour que l'appelé puisse les récupérer
- Convention : on commence par empiler le dernier paramètre
- L'appelant trouvera les paramètres à des adresses calculées par rapport à SP
- Après le retour, il ne faut pas oublier de libérer la place des paramètres
- Souvent les compilateurs passent les premiers paramètres dans des registres et les suivants dans la pile
  - Exemple gcc pour le ARM : 4 premiers paramètres dans r0, r1, r2, r3, les autres sur la pile

Exemple

- **Convention** : Tous les paramètres sur la pile (ordre inverse)
- Main() void Proc(int a, int b)
  { … Proc(3, 2) ; …} { int x ; x= a+b;}
- **main : …**

```
    MOV r0, #3 @r0 contient le 1er paramètre
    MOV r1, #2 @r1 contient le 2eme paramètre
    STMFD SP !, {r0,r1}  @empilement des deux paramètres
    bl etiq_proc
    ADD SP, SP , #8      @libération paramètre
        …
etiq_proc : STMFD SP !, {LR}   @empilement adresse de retour
            SUB SP, SP, #4     @reservation 4 cases sur la pile pour x
                               @SP pointe sur x ; a à l'adresse SP+8 et b : SP+12
            LDR r0, [SP,#8]    @r0=a
            LDR r1, [SP,#12]   @r1=b
            ADD r0, r0, r1
            STR r0, [SP]       @x=a+b
            ADD SP, SP, #4     @Libération place des variables locales
            LDMFD SP !, {PC}   @dépilement adresse de retour
```


Etat de la pile



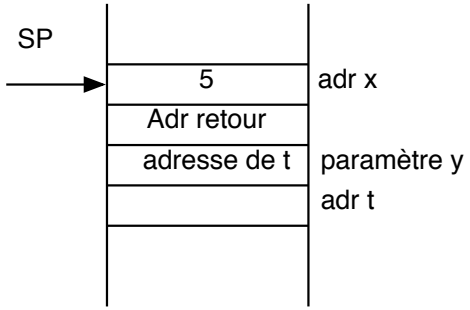
Pendant l'exécution du corps de Proc

Cas des paramètres résultats

- Main() { int t = 2 ;…Proc(&t) ; …} void Proc(int* y) { int x =5; *y=x}
- main : …

```
        MOV r0, #2
        STMFD SP !, {r0}    @empilement de t=2; variable locale de main sur la
      pile (main est aussi une procédure !)
        MOV r1, SP          @SP dans r1 car empilement de SP non autorisé
        STMFD SP !, {r1}    @empilement de l'adresse de t lui même déjà sur la
    pile (pointé par SP)
        bl etiq_proc
        ADD SP, SP , #4     @libération paramètre
        …
Etiq_proc : STMFD SP !, {LR}   @empilement adresse de retour
        SUB SP, SP, #4      @reservation 4 case sur la pile pour x
        MOV r0, #5
        STR r0, [SP]        @x=5; SP pointe sur x ;
        LDR r0, [SP,#8]     @r0=adresse de t, paramètre y à l'adresse SP+8
        LDR r1, [SP]        @r1= x
        STR r1, [r0]        @*a=x
        ADD SP, SP, #4      @Libération place des variables locales
        LDMFD SP !, {PC}    @dépilement adresse de retour
```


Etat de la pile



Pendant l'exécution du corps de Proc

Le problème des temporaires

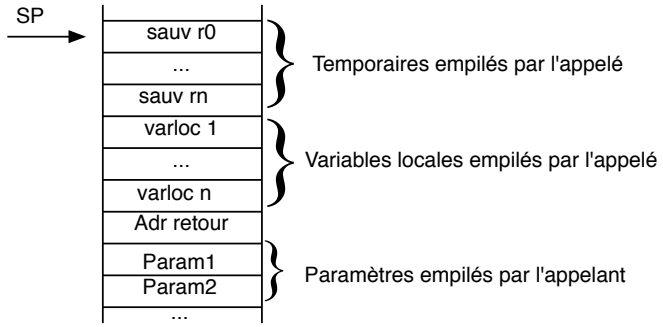
- Pour faire des calculs on a besoin d'utiliser des registres
- ceux qu'on utilise en lecture seule n'ont pas besoin d'être sauvegardés
- Dans les exemples précédents : r0 et r1
- Si l'appelant les utilisait, il faut les sauvegarder (encore une fois dans la pile)
- Solution avec sauvegarde dans l'appelé :
  - sur la pile: soit au dessus, soit en dessous des variables locales
- Attention du coup les calculs des adresses des paramètres et des variables locales par rapport à SP peuvent changer
- Il ne faut pas oublier de les restaurer et de libérer leur place sur la pile avant le retour

Résumé appel de procédure

- **Avant l'appel (dans l'appelant):**
  - Empilement des paramètres effectifs ou passage par registres
- **Appel : instruction particulière (sauvegarde l'adresse de retour)**
  - ARM:
    - Bl Etiquette_proc
    - Adresse de retour dans lr (r14)
- **Dans la procédure (dans l'appelé):**
  - **Prologue**:
    - Empilement de l'adresse de retour (indispensable si appel imbriqué)
    - Réservation et initialisation des variables locales sur la pile
    - Sauvegarde si nécessaire des registres utilisés dans la procédure
  - Corps de la procédure
    - Adresses des Variables locales et paramètres : pointeur de pile + déplacement
  - **Epilogue** (Retour de la procédure)
    - Restauration des temporaires
    - Libération des variables locales
    - Retour par dépilement de l'adresse de retour

Environnement d'une procédure sur la pile

- **Conventions:**
  - **Temporaires "en dessus" des variables locales**
  - **Ordre des variables locales et des paramètres**



Exercice

- Main()
  { int a, b=3;
  …Proc(&a, b) ; …}

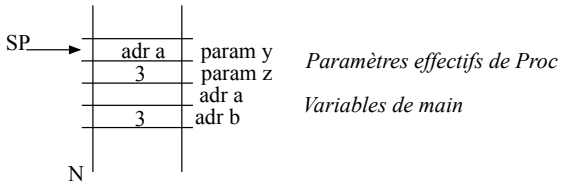
  void Proc(int* y, int z)
  { int x =5; *y= z+x }

- Ecrire le programme assembleur
  - En sauvegardant systématiquement l'adresse de retour
  - En stockant
    - les variables locales sur la pile : première variable "au dessus"
    - les paramètres sur la pile: premier paramètre "au dessus"

Correction

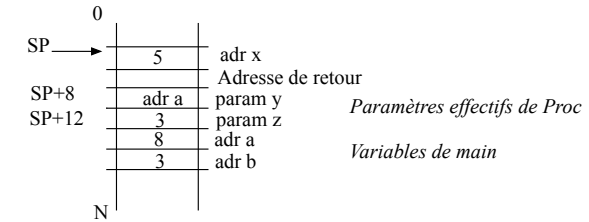
```
main : …
      MOV r0, #3
      STMFD SP !, {r0}     @empilement variable locale b=3
      SUB SP, SP, #4       @reservation pour var locale a (Adresse a= SP, adresse b= SP+4)
      LDR r1, [SP,#4]      @r1 contient la valeur de b
      MOV r0, SP           @SP dans r0 car empilement de SP non autorisé
                           @r0 contient l'adresse de a
      STMFD SP !, {r0, r1} @empilement des deux paramètres
      BL etiq_proc
      ADD SP, SP , #8      @libération paramètres
      …
```

- Etat de la pile au moment du ***bl etiq_proc***:




```
Etiq_proc : STMFD SP !, {LR}   @empilement adresse de retour
            SUB SP, SP, #4     @reservation 4 case sur la pile pour x
            MOV r0, #5
            STR r0, [SP]       @x=5; SP pointe sur x ;
            LDR r0, [SP,#8]    @r0= paramètre y à l'adresse SP+8 (contient adresse de a)
            LDR r1, [SP]       @r1= x
            LDR r2, [SP,#12]   @r2 contient le 2eme paramètre z
            ADD r2, r1, r2     @r2= z+x
            STR r2, [r0]       @*a=z+x
            ADD SP, SP, #4     @Libération place des variables locales
            LDMFD SP !, {PC}   @dépilement adresse de retour
```

- Etat de la pile après *STR r2, [r0]*:

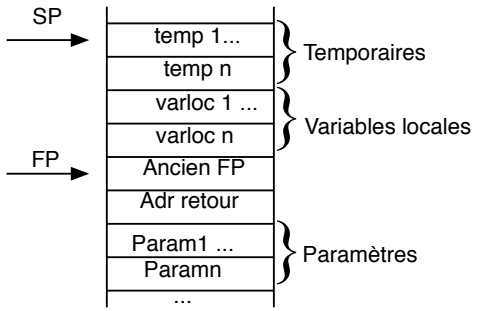


Simplification du calcul des adresses

- Le calcul des adresses des variables locales et des paramètres par rapport à SP peut varier pendant le déroulement de la procédure
- On définit un deuxième pointeur : FP (frame pointer)
- FP pointe sur l'environnement de la procédure. Il ne changera pas pendant le déroulement de la procédure.
- Quand on rentre dans la procédure, il faut sauvegarder son ancienne valeur et la restaurer avant de sortir.
- On peut libérer l'environnement complet d'une procédure en faisant SP = FP puis dépilement dans FP et PC
- Dans certains processeurs Instructions spécialisées (LINK UNLINK du 68000).

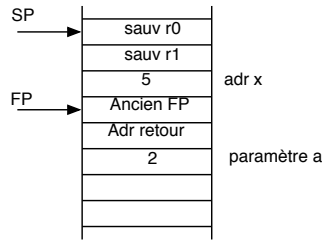
Utilisation de FP

- **Convention**: temporaires au dessus des variables locales
- Variables locales : à partir de FP - 4
- Paramètres : à partir de FP + 8



Exemple

- Main() / { / … / Proc(3) ; … / } — void Proc(int a) ; / { int x ; / … / x=a+1; …. / }
- **Conventions**:
  - Paramètres et variables locales sur la pile,
  - utilisation de FP,
  - Temporaires au dessus des variables locales



Pendant l'exécution du corps de Proc

Assembleur ARM

```
main :  ...
        MOV r0, #3
        STMFD SP !, {r0}      @empilement du paramètre effectif de Proc
        BL etiq_proc
        ADD SP, SP , #4       @libération paramètre
        ...
etiq_proc : STMFD SP !, {FP, LR } @empilement adresse de retour et FP
        MOV FP, SP            @FP=SP (pointe sur ancien fp)
        SUB SP, SP, #4        @reservation 4 cases sur la pile pour x;
                              @ x : FP –4, a : FP+8
        STMFD SP !, {r0, r1 } @ empilement des temporaires
        MOV r0, #5
        STR r0, [FP, #-4]     @x=5
        …
        LDR r1, [FP, #8]      @r1=premier parametre a en FP+8
        ADD r1, r1, #1        @r1= a+1

        STR r1, [FP, #-4 ]    @x= a+1
        …
        LDMFD SP !, {r0, r1 } @ depilement des temporaires
        MOV SP, FP            @Libération environnement SP=FP
        LDMFD SP !, {FP, PC}  @dépilement adresse de retour et FP
```

Exercice

- Main() void Proc(int* y, int z)

  { int a, b=3; { int x =5; *y= z+x }

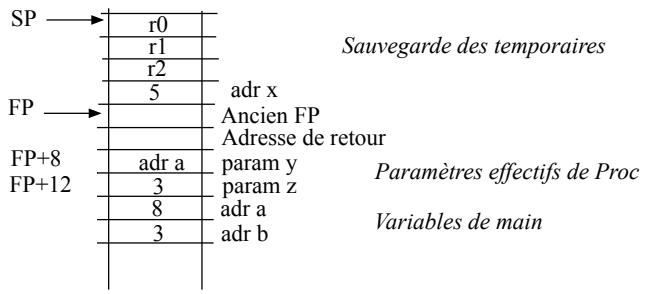
  …Proc(&a, b) ; …}

- Ecrire l'assembleur en utilisant FP et en sauvegardant systématiquement les temporaires dans l'appelé

Assembleur ARM

- **main ne change pas**

```
etiq_proc : STMFD SP !, {FP, LR }    @empilement adresse de retour et FP
            MOV FP, SP               @FP=SP (pointe sur ancien fp)
            SUB SP, SP, #4           @reservation 4 cases sur la pile pour x;
            STMFD SP !, {r0, r1, r2 } @ empilement des temporaires
```



Assembleur ARM

- **main ne change pas**

```
etiq_proc : STMFD SP !, {FP, LR }    @empilement adresse de retour et FP
            MOV FP, SP               @FP=SP (pointe sur ancien fp)
            SUB SP, SP, #4           @reservation 4 cases sur la pile pour x;
            STMFD SP !, {r0, r1, r2 } @ empilement des temporaires
            MOV r0, #5
            STR r0, [FP, #-4]        @x=5; FP-4 pointe sur x ;
            LDR r0, [FP,#8]          @r0= paramètre y à l'adresse FP+8 (contient adresse de a)
            LDR r1, [FP, #-4]        @r1= x
            LDR r2, [FP,#12]         @r2 contient le 2eme paramètre z
            ADD r2, r1, r2           @r2= z+x
            STR r2, [r0]             @*a=z+x
            LDMFD SP !, {r0, r1, r2 } @ dépilement des temporaires
            MOV SP, FP               @Libération place des variables locales
            LDMFD SP !, {FP, PC}     @dépilement adresse de retour et ancien FP
```

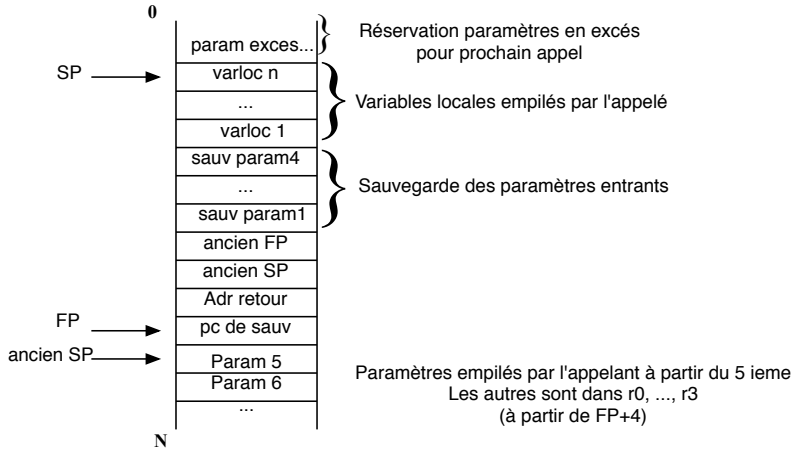

Convention du compilateur gcc ARM Optimisation "normale"

- **Convention de gestion de la pile**
  - SP pointe sur dernier empilé
  - empilement : adresse décroissante
- **Paramètres:**
  - Quatres premiers passés dans les registres r0, r1, r2, r3
  - Ceux en excès empilés par l'appelant avec convention : ordre inverse : cinquième paramètre au dessus
  - Dans le cas d'une fonction la valeur est retournée dans r0 (paramètres à partir de r1)
  - Variables locales sur la pile (empilées par ordre de déclaration)
  - Utilisation de SP (r14) et FP (r11)

Convention du compilateur gcc ARM

- **Le prologue**
  - sauvegarde : ancien fp, ancien sp, adresse de retour et le PC courant
    - Une seule instruction STM
  - Réservation de la place pour sauvegarder
    - les paramètres actuellement dans les registres (prochain appel),
    - les variables locales et les paramètres en excès pour un nouvel appel de fonction
    - Pas de temporaire, les registres sont “libérés” par l’appelant
- **L’épilogue**
  - Libération du contexte (variables locales, paramètres) sur la pile
  - Retour par mise à jour de PC
  - SP <- ancien SP (sur la pile)
  - FP <- ancien FP (sur la pile)
  - PC <- adresse de retour
  - Simplifié: Une seule instruction LDM

Contexte d’une fonction compilateur gcc ARM



Convention du compilateur gcc ARM

- **Le prologue**
  - Sauvegarde : ancien fp, ancien sp, adresse de retour et le PC courant (STM)
    - `mov ip, sp`
    - `stmfd sp!, {fp, ip, lr, pc}`
    - Utilisation du registre ip (r12) pour pouvoir sauvegarder SP
    - Sauvegarde de la valeur de PC au moment de l’empilement de SP et FP : cette sauvegarde n’est pas nécessaire à la gestion des appels mais facilite la mise au point des programmes avec un debogueur
  - Mise à jour de fp
    - `sub fp, ip, #4`
  - Réservation de la place pour sauvegarder
    - les paramètres actuellement dans les registres (prochain appel)
    - les variables locales et les paramètres en excès pour un nouvel appel de fonction
    - `sub sp, sp, #place-necessaire`
- **L’épilogue**
  - `ldmea fp, {fp, sp, pc}`
  - dépilement avec convention (empty, ascendant)