Institut
Mines-Télécom

ABI / Convention d'appel

ou comment interfacer C et assembleur

Alexis Polti

Licence de droits d'usage

Contexte académique } sans modification

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après, et à l'exclusion de toute utilisation commerciale.

Le droit d'usage défini par la licence autorise un usage dans un cadre académique, par un utilisateur donnant des cours dans un établissement d'enseignement secondaire ou supérieur et à l'exclusion expresse des formations commerciales et notamment de formation continue. Ce droit comprend :

- le droit de reproduire tout ou partie du document sur support informatique ou papier,
- le droit de diffuser tout ou partie du document à destination des élèves ou étudiants.

Aucune modification du document dans son contenu, sa forme ou sa présentation n'est autorisé.

Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité.

Le droit d'usage défini par la licence est personnel, non exclusif et non transmissible.

Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur :

alexis.polti@telecom-paristech.fr

tl;dr

- **Ce qu'on va apprendre :**
  - la gestion des fonctions en C
    - le passage d'argument
    - le passage de la valeur de retour
    - à quoi sert une pile
    - ce qu'est un activation record
  - les conventions d'appel ARM

ABI

- **ABI : application binary interface**
  - Convention regroupant :
    - les types de données, leur tailles et alignements
    - les formats des exécutables, objets et bibliothèques
    - les conventions d'usage de registres
    - les conventions d'appel de fonctions
    - les appels système
    - le format des informations de débug
    - le format et traitement des exceptions
    - etc.
  - Pour ARM :
    - OABI (APCS) : obsolète
    - EABI (AAPCS) : actuelle et plus performante
    - EABIHF : la même, le coprocesseur flottant en plus

Structure d'une fonction

- **Architecture d'une fonction C**
  - une fonction se compose :
    - d'un prologue : met en place un environnement d'exécution adéquat
    - du corps de la fonction
    - d'un épilogue :
      - fait le ménage,
      - place l'éventuelle valeur de retour au bon endroit,
      - et transfère le contrôle à la fonction appelante.
  - On peut demander la suppression des prologues / épilogues par l'attribut `naked`
    - exemple : `__attribute__ ((naked)) void f(void);`

Structure d'une fonction

- **Environnement d'exécution**
  - Une fonction a souvent besoin d'utiliser des registres,
  - mais elle ne sait pas si ces registres sont déjà utilisés ou non par les fonctions appelantes.
  - → Elle doit les remettre dans leur état original avant de retourner

  - En clair : *Merci de laisser cet endroit aussi propre en sortant que vous l'avez trouvé en entrant…*

Mécanisme d'appel de fonction

- La procédure appelante (***caller***) :
  - évalue les arguments et les place à des endroits appropriés,
  - sauve l'adresse de retour,
  - sauvegarde les registres qu'elle utilise (caller saved registers) et qui doivent être préservés,
  - passe le contrôle à la procédure appelée.

Mécanisme d'appel de fonction

- La procédure appelante (***caller***) :
  - évalue les arguments et les place à des endroits appropriés
  - sauve l'adresse de retour
  - sauvegarde les registres qu'elle utilise (caller saved registers) et qui doivent être préservés
  - passe le contrôle à la procédure appelée

- La procédure appelée (***callee***)
  - initialise les autres registres critiques (FP, …),
  - sauvegarde les registres qui seront utilisés et qui doivent être maintenus (callee saved registers),
  - alloue de la place en mémoire pour ses variables locales et temporaires (modifie SP),

  - **Le corps de la fonction est exécuté**

  - stocke l'éventuelle valeur de retour à un endroit approprié,
  - restaure les registres callee saved,
  - libère l'espace mémoire alloué (remet SP à son ancienne valeur),
  - redonne le contrôle à la fonction appelante.

prologue

épilogue

Mécanisme d'appel de fonction

- La procédure appelante (***caller***) :
  - évalue les arguments et les place à des endroits appropriés
  - sauve l'adresse de retour
  - sauvegarde les registres qu'elle utilise (caller saved registers) et qui doivent être préservés
  - passe le contrôle à la procédure appelée

- La procédure appelée (***callee***)
  - initialise les autres registres critiques (FP, …)
  - sauvegarde les registres qui seront utilisés et qui doivent être maintenus (callee saved registers)
  - alloue de la place en mémoire pour ses variables locales et temporaires

prologue

  - **Le corps de la fonction est exécuté**
  - stocke l'éventuelle valeur de retour à un endroit approprié
  - restaure les registres callee saved
  - libère l'espace mémoire alloué
  - redonne le contrôle à la fonction appelante

épilogue

- La procédure appelante :
  - restaure ses registres sauvegardés,
  - désalloue les éventuels arguments,
  - et continue son exécution.

Où en est-on ?

- **À venir :**
  - Que sauvegarder exactement ?
  - Où ?
  - Et par qui ?

  - Comment transmettre les arguments ?
  - Comment transmettre la valeur de retour ?

Pile / activation record

- **Pile et activation record:**
  - En C, les fonctions peuvent être récursives ou ré-entrantes. Chaque instance d'appel de fonction doit donc pouvoir disposer de son propre espace de stockage pour ses objets.
  - Les garder dans des endroits statiques est très difficile.
  - Il faut donc pouvoir allouer dynamiquement de la mémoire pour stocker ces objets.
  - La plupart du temps, cela est fait avec :
    - une ou plusieurs pile(s) : C, C++, Java, Ada, C#, Pascal, …
    - le tas : LISP, Scheme, …
  - L'espace contenant les objets propres à un appel de fonction est appelé « activation record » ou « activation frame ».

Pile / activation record

- **Rôle de "la" pile :**
  - Originellement, stocker l'adresse de retour d'une fonction : « call stack ».
  - Implémentation :
    - matérielle
    - logicielle
    - mixte
  - On en profite généralement pour y stocker aussi :
    - certains paramètres lors d'un appel de fonction,
    - les sauvegardes de registres / état du processeur,
    - des informations diverses…
  - Selon les langages et architectures, une ou plusieurs piles (Forth, Stackless Python).

Pile / activation record

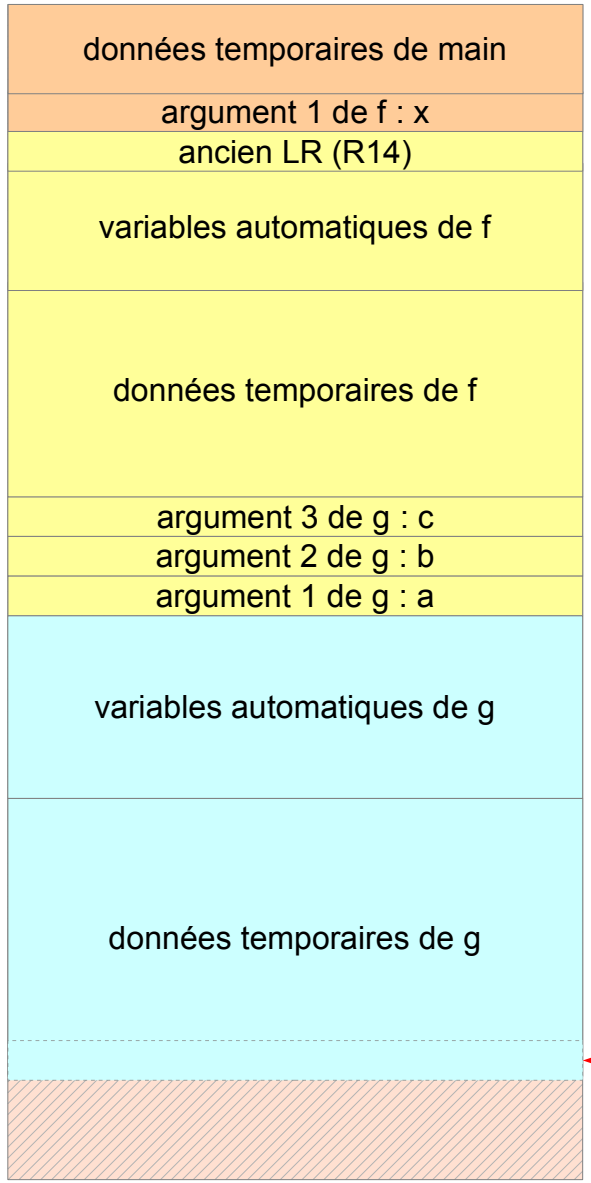
- **Contenu de l'activation record en C / C++**
  - adresse de retour (si nécessaire)
  - arguments
  - sauvegardes de registres et de l'état du processeur
  - dynamic link : lien vers l'AR de la fonction appelante
  - données locales
    - variables locales
    - valeurs intermédiaires dans certaines expressions
    - données dynamiques
  - informations diverses (pas en C)
    - `this` en C++
    - static link : lien vers l'AR de la définition de la fonction englobante
    - liens vers handlers d'exception
    - …

Pile / activation record

```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```



direction de la pile
(ici, full-descending)

ATTENTION
sera modifié plus tard !

Sauvegarde des registres

- **Stratégies**
  - *Caller saved registers* : la procédure appelante sait quels sont les registres qui ne doivent pas être modifiés par l'appel et les sauvegarde.
    - Inconvénient : on en sauvegarde généralement beaucoup trop.
  - *Callee saved registers* : la procédure appelée sait quels sont les registres qu'elle va modifier et les sauvegarde.
    - Inconvénient : on en sauvegarde généralement beaucoup trop.
  - Une stratégie optimale consiste en une approche intermédiaire.

Sauvegarde des registres

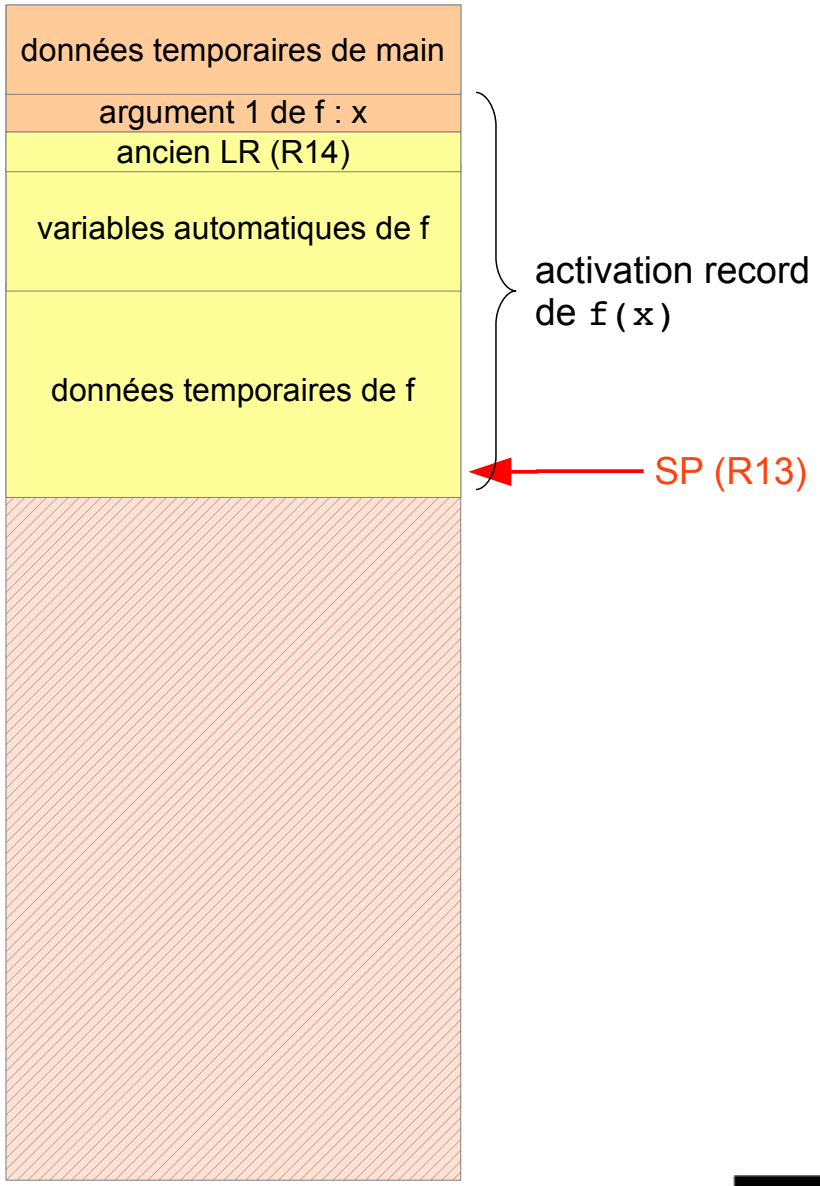
- **Stratégie intermédiaire (ARM)**
  - Les registres sont partagés en deux groupes :
    - `n` registres caller saved
    - `m` registres callee saved

Sauvegarde des registres

- Quelle répartition ?
  - Le code de sauvegarde / restauration est généré :
    - *caller saved register* : 1 fois par appel de procédure.
    - *callee saved register* : 1 fois par procédure.
    - → Si on cherche à optimiser la taille du code, on a intérêt à avoir un maximum de callee-saved registers.
  - Une procédure terminale a tout intérêt à utiliser des registres caller saved.
  - Une procédure non terminale a tout intérêt à utiliser :
    - Pour les registres devant survivre à un appel de fonction, des registres callee saved.
    - Pour le reste, des registres caller saved.
  - EABI ARM :
    - R0 à R3 : caller saved
    - R4 à R11 : callee saved

Activation frame / Frame pointer

- **Adressage des données**
  - Si nous sommes actuellement dans f(x), et si toutes ses données sont de taille connue à la compilation :
    - comment fait-on référence aux arguments de f ?
    - comment fait-on référence aux variables locales de f ?
    - comment fait-on référence aux données temporaires de f ?



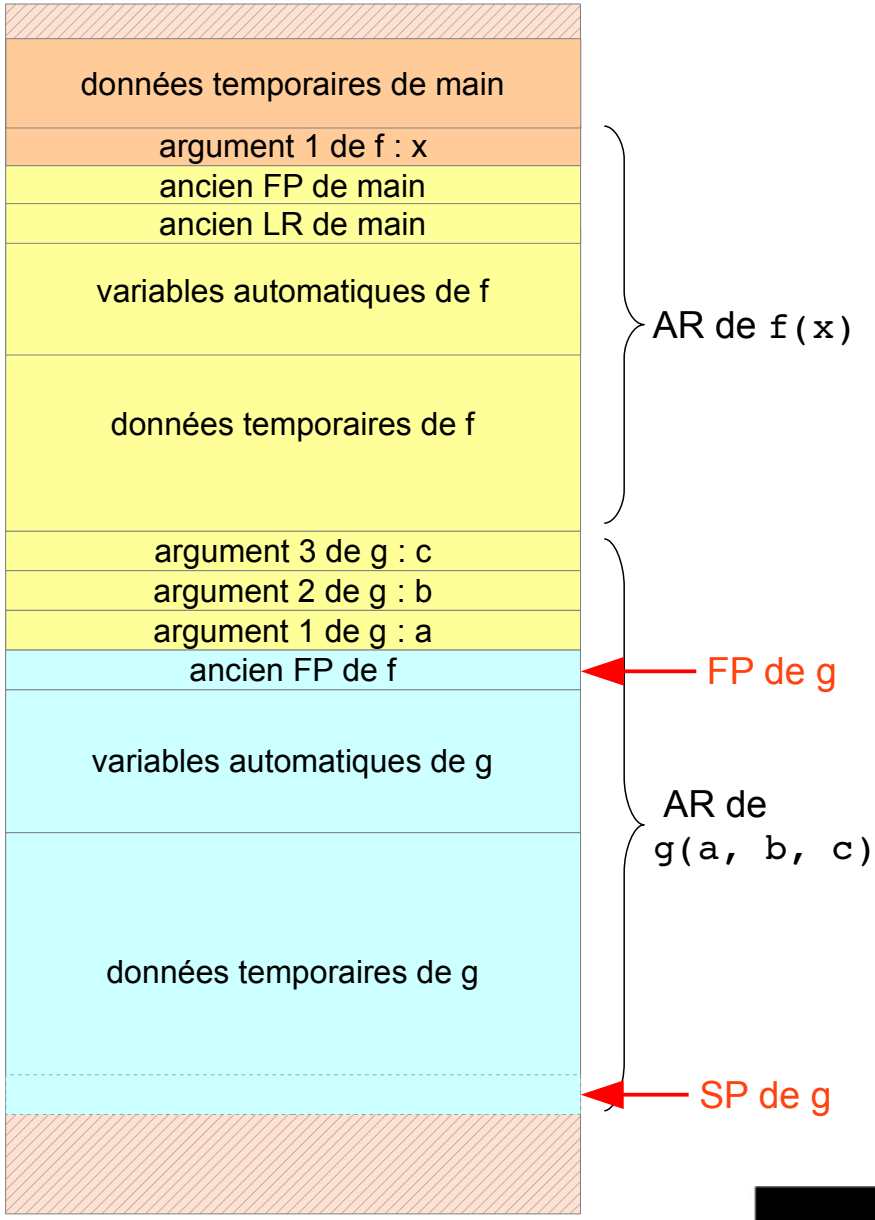
Réponse : ?

Activation frame / Frame pointer

- **Problème**
  - En C, comme dans beaucoup de langages, les données locales peuvent être de taille non connue à la compilation :
    - exemples ?
  - Conséquence :
    - La taille des activation record n'est donc pas connue à la compilation.
    - La position du SP n'est donc pas connue à la compilation.
    - Il est impossible de calculer les adresses des variables locales / arguments en se basant sur SP.
  - Il faut un pointeur stable : Frame Pointer (FP, R11 sur ARM)

Activation frame / Frame pointer

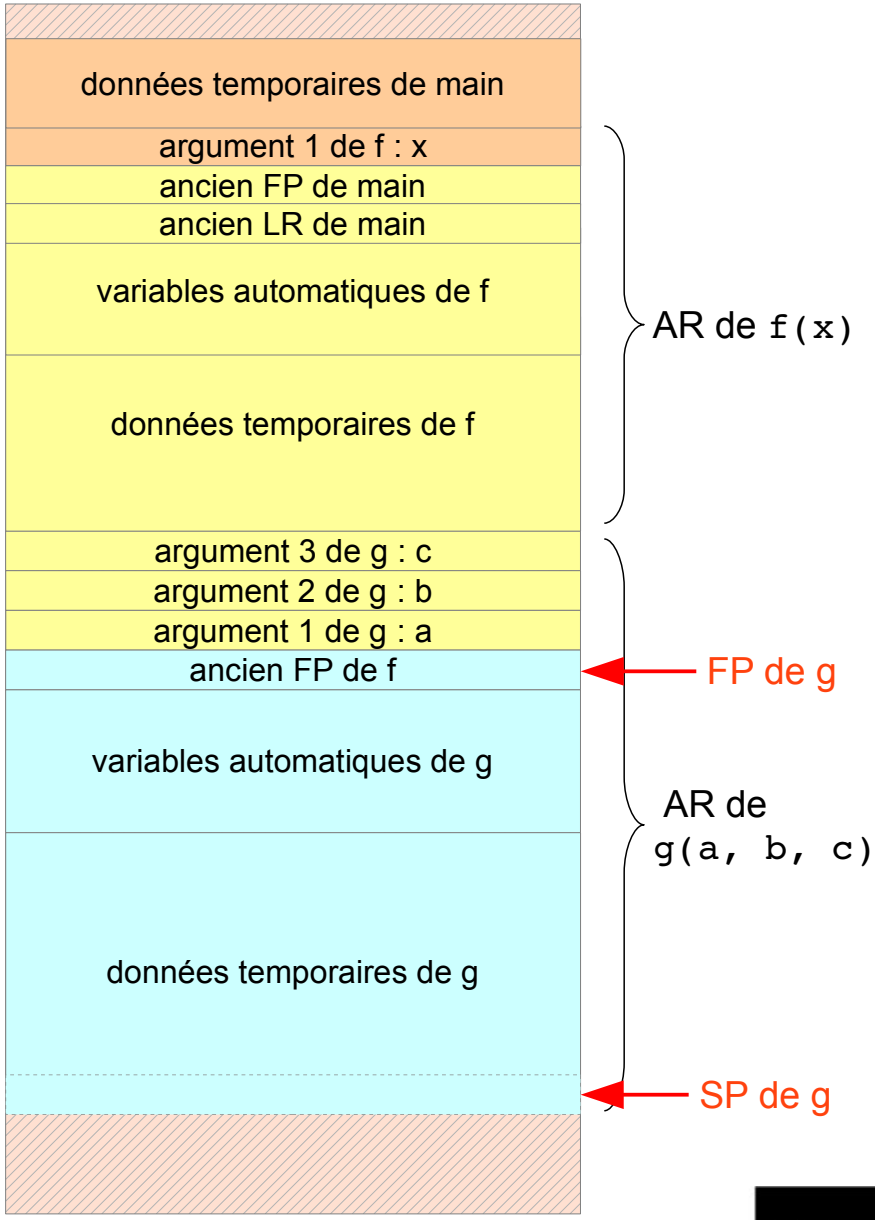
- **Frame Pointer**
  - permet de calculer les adresses des données d'une fonction, même en cas de SP variable
    - les arguments sont en FP + 4, FP + 8, FP + c, …
    - les variables locales et temporaires sont en FP – 4n.
  - Sur ARM, traditionnellement le Frame Pointer est R11.
  - Si on sait qu'il n'y aura pas d'allocation dynamique sur la pile, on peut supprimer le frame pointer grâce à l'option de compilation `-fomit-frame-pointer`.



Activation frame / Frame pointer

- **Retour de fonction**
  - SP <= FP+4
  - FP <= Ancien FP
  - L'appelant désalloue ensuite les arguments.

- **Quizz**
  - Où se trouve le dynamic link ?



Où en est-on ?

- **On a vu :**
  - rôle de la pile
  - structure des activation frames / records
  - rôle du SP / FP
  - stratégies de sauvegarde des registres

- **À venir :**
  - Comment transmettre les arguments ?
  - Comment transmettre la valeur de retour ?

Passage d'arguments

- **Plusieurs scénarios possibles**
  - L'appelant peut créer une structure, y stocker les arguments et passer l'adresse de cette structure (par un registre, qui deviendra important et qu'il faudra sauver en cas de sous-appel).
  - L'appelant connaît l'emplacement du stack frame de l'appelé, et peut stocker les argument juste à côté de cette frame, de façon à ce que l'appelé les trouve (cf. schémas précédents).
  - L'appelant peut stocker certains arguments dans des registres, et d'autres sur la pile (scénario maintenant le plus fréquent).

Passage d'arguments

- **Sur ARM**
  - Les 4 premiers arguments sont passés dans R0 à R3.
  - Les arguments suivants sont passés sur la pile.
  - Les types de 64 bits sont passés sur deux registres consécutifs.
  - Les grosses structures sont passées
    - soit sur la pile,
    - soit un bout par les registres et le reste sur la pile.
  - Pour les tableaux, on passe l'adresse du premier élément.

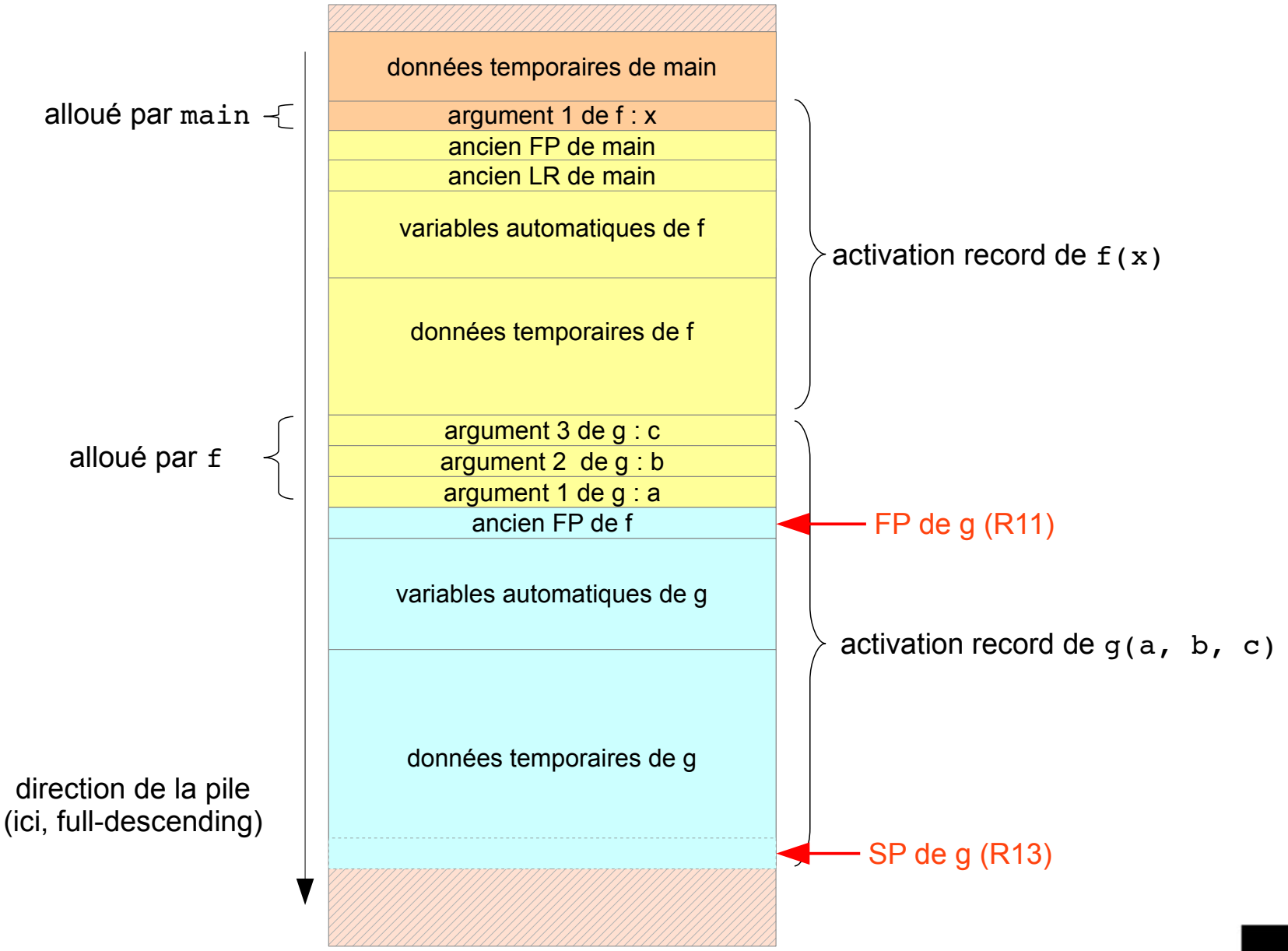
Passage de la valeur de retour

- **En général**
  - Dans un registre, pour les petites données.
    Sur ARM :
    - Donnée sur 32 bits : R0
    - Donnée sur 64 bits : R0+R1
    - Autre : cf. prochain slide.

Passage de la valeur de retour

- **En général**
  - Pour les structures / grosses données, il est tentant de les allouer sur la pile et de retourner un pointeur dessus.
    - Pourquoi est-ce une très mauvaise idée ?
    - Comment le faire correctement ?

En résumé



Résumé convention ARM

- **En général sur ARM**
  - PC = R15, LR = 14, SP = R13, IP = R12, FP = R11 (pour gcc).
  - R4 à R11 : utilisés pour les variables locales, callee saved.
  - R0 à R3 : utilisés pour passer les arguments, caller saved.
  - Évidemment :
    - FP et SP sont callee saved.
    - LR et IP sont caller saved.
  - La valeur de retour est transmise dans R0 (ou R0 + R1).
  - Les procédures ne sauvegardent leur LR que si elles appellent d'autres routines.
  - Le pointeur de pile doit être aligné sur 8 octets aux frontières d'unités de compilation et pour une table de handlers d'IRQ.

Licence de droits d'usage

Contexte académique } sans modification

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après, et à l'exclusion de toute utilisation commerciale.

Le droit d'usage défini par la licence autorise un usage dans un cadre académique, par un utilisateur donnant des cours dans un établissement d'enseignement secondaire ou supérieur et à l'exclusion expresse des formations commerciales et notamment de formation continue. Ce droit comprend :

- le droit de reproduire tout ou partie du document sur support informatique ou papier,
- le droit de diffuser tout ou partie du document à destination des élèves ou étudiants.

Aucune modification du document dans son contenu, sa forme ou sa présentation n'est autorisé.

Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité.

Le droit d'usage défini par la licence est personnel, non exclusif et non transmissible.

Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur :

alexis.polti@telecom-paristech.fr