Conventions d'appels et interruptions

…faire cohabiter du C et de l'assembleur

Tarik Graba
tarik.graba@telecom-paris.fr
Année scolaire 2020/2021

Plan

Flot d'exécution linéaire

- Un processeur exécute les instructions les unes après les autres de façon linéaire.
  - Le **PC** est incrémenté (+4 ou +2)
- On peut faire des branchements pour aller vers une instruction précise.
  - sur ARM on a l'instruction **b** (branch)
- Ces instructions de branchement permettent d'implémenter des tests, boucles….
  - Voir le cours sur la cross-compilation

Les routines/sous-routines

Une **routine** est un fragment du programme à qui on donne la main.

- on donne le contrôle à la routine à partir du programme,
  - le programme appelant sait comment appeler cette routine
- la routine s'exécute,
  - le programme appelant ne sait pas ce qu'elle fait vraiment
  - cette **routine** peut, elle-même, appeler une **sous-routine**.
- à la fin on revient au programme appelant (juste après l'appel) et poursuit son exécution.
  - le programme appelant est en attente de la fin de la sous-routine

Les routines/sous-routines

Une **routine** peut être une:

- **fonction** si elle retourne un résultat
- **une procédure** si elle ne retourne pas de résultat

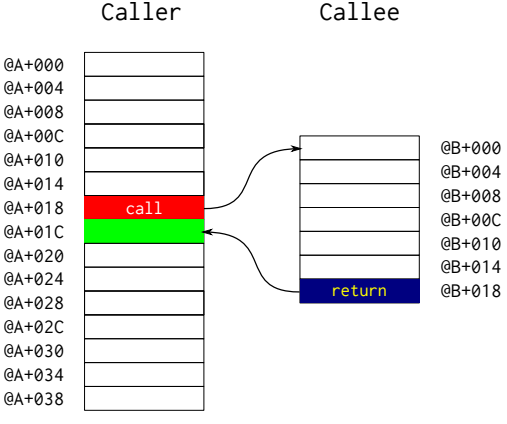
En langage `C` *le terme fonction est utilisé dans les deux cas.*

Les routines/sous-routines

Pour se simplifier la vie:

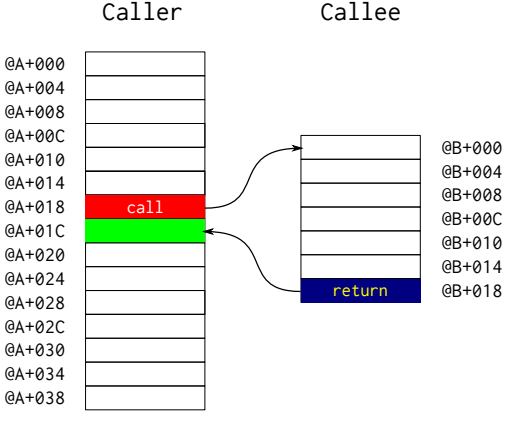
- **L'appelant** (**Caller**):
  - la partie du programme où on appelle la sous-routine
- **L'appelée** (**Callee**):
  - la sous-routine qui est appelée

Les routines/sous-routines



- On appelle la sous-routine en gardant l'adresse de retour
  - L'instruction **BL**(*Branch and Link*) sauvegarde l'adresse de retour dans le registre **`lr`**
- Au retour, on revient à l'adresse qui suit l'appel à la sous-routine
  - Il faut remettre **`lr`** dans **`pc`**
  - **`MOV pc, lr`** ou **`BX lr`**

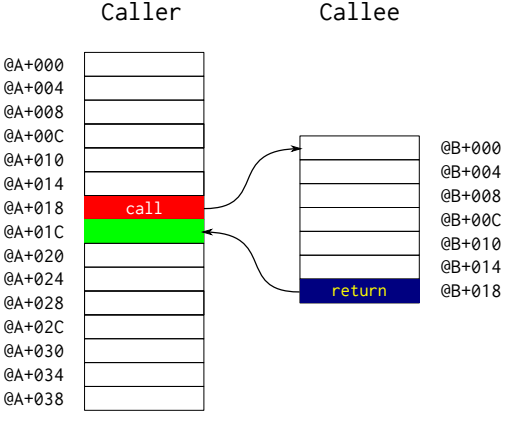
Les routines/sous-routines



Questions

- Où stocker les variables locales?
  - Utiliser des registres? Lesquels?
  - Les stocker en mémoire? où?
- Comment transmettre des arguments et récupérer la valeur de retour?
  - Utiliser des registres? Lesquels?
  - Les stocker en mémoire? où?
- Tout n'est pas connu à la compilation:
  - des appels imbriqués,
  - des appels récursifs, des fonctions réentrantes

Les routines/sous-routines



Il nous faut

- Des conventions:
  - où stocker les choses,
  - comment utiliser la mémoire,
  - quels registres utiliser,
  - qui en est responsable,
  - où se trouvent les arguments, les valeurs de retour…

Plan

Pourquoi une convention?

Permettre l'interopérabilité de sous-routines:

- compilées séparément,
- générées à partir de langages différents (`C` et assembleur par exemple),
- des compilateurs différents, ou des versions différentes du même compilateur.

En définissant:

- comment l'appelant doit configurer l'environnement de l'appelé,
- et ce qu'a le droit de faire l'appelé et comment il doit restaurer l'environnement avant de rendre la main.

Pourquoi une convention?

Pour les processeurs ARM

Des conventions définies dans plusieurs documents [Lien vers le site d'ARM]

AAPCS : Procedure Call Standard for the Arm Architecture [Lien]

- définit les conventions bas niveaux

ABI : Application Binary Interface

- des extensions de compatibilité liées aux exécutables et binaires
- les formats et organisation des binaires (elf par exemple),
- les bibliothèques, certains langages (C++ par exemple), les OS (Linux par exemple)

EABI : ABI ARM pour les applications embarquées

- utilisée dans le contexte de systèmes embarqué sans système d'exploitation (baremetal) ou avec des OS temps réel.

Plan

La pile

La pile est un espace mémoire dynamique qui permet de stocker l'environnement d'une sous-routine:

- Ses variables locales/automatiques
- Certains arguments
- Des sauvegardes de contexte

Sur les processeurs ARM, elle est gérée de façon logicielle. Des instructions doivent être ajoutées pour empiler (sauvegarder en mémoire) ou dépiler (restaurer) des éléments.

La pile

Principe

Un exemple:

- une fonction **f** avec ses variables locales
- elle appelle la fonction **g** qui a aussi des variables locales
- **g** appelle une sous-fonction **h**

```
void f(){
  int vf0, vf1, vf2;
  ....
  g();
  ...
}

void g(){
  int vg0, vg1, vg2, vg3;
  ....
  h();
  ...
}

void h(){
  int vh0, vh1;
  ....
}
```

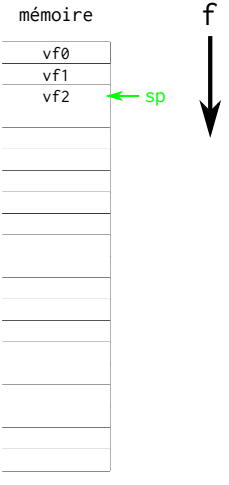
La pile

Principe

Dans la fonction **f** on place les variables en mémoire:

- à la suite
- un pointeur nous donne l'adresse de la dernière variable
- on connait la position de toutes les variables par rapport à ce pointeur

C'est notre pointeur de pile (*stack pointer* **sp**)



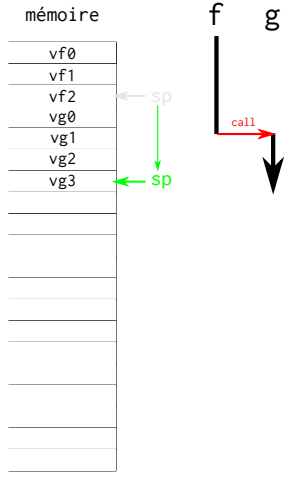
La pile

Principe

À l'appel de la fonction **g**:

- on fait avancer le pointeur de pile suffisamment pour réserver de l'espace pour les variables de **g**
  - on connait le nombre de variables locales
- les variables locales sont référencées par rapport au même pointeur **sp**

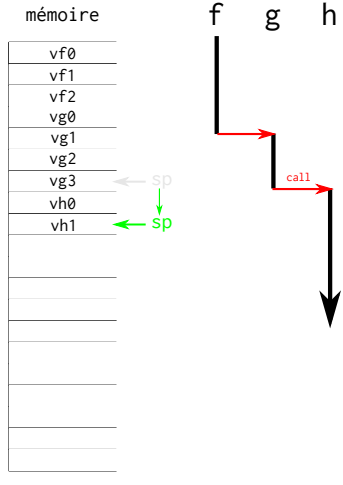
f n'a pas d'information sur l'espace nécessaire pour les variables locales de **g**.
Le code qui ***crée*** la pile de **g** appartient à **g**.
Il est appelé **prologue**.



La pile

Principe

À l'appel de la fonction **h** on refait la même chose.



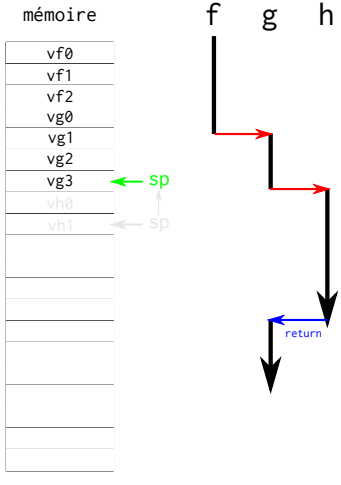
La pile

Principe

À la fin de l'exécution de **h**:

- on restaure la valeur de **sp** en le faisant reculer du bon nombre de cases
- on rend la main à **g**

Le code qui restaure le contexte de l'appelant est appelé **épilogue**. Il fait forcément partie de **h** car comme le prologue il doit savoir combien d'éléments sont empilés.

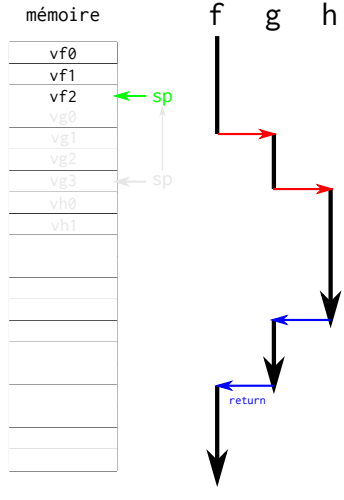


La pile

Principe

À la fin de l'exécution de **g** on refait la même chose.

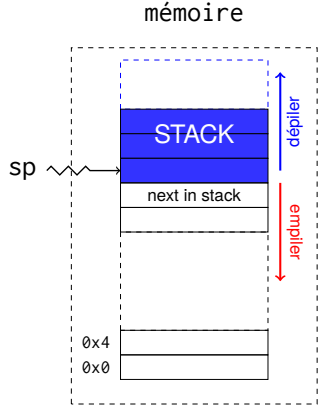
Pour ne pas dégrader les performances, les données empilées ne sont pas effacées. On n'a juste plus de référence pour y accéder.



La convention de pile sur ARM

Full Descending stack

- Le registre `r13(sp)` est le pointeur de pile (stack pointer)
- Le pointeur de pile contient l'adresse de la dernière donnée empilée (case pleine)
- Avant chaque empilement le pointeur de pile doit être décrémenté (la pile descend)



La convention de pile sur ARM

D'autres contraintes

- La valeur registre `r13(sp)` doit être multiple de 4
  - la pile servant à sauvegarder des registres, elle doit être alignée sur des adresses de mots de 32 bits
- La pile doit être alignée sur une un double mot (8 octets) aux interfaces
  - lors de l'appel d'une sous-routine
  - nécessaire pour certains mécanismes liés aux exceptions
- pour assurer l'alignement le compilateur va ajouter dans la pile des éléments normalement inutiles

L'activation record/pile d'appel

Du jargon:

- Activation record
- Pile d'appel (Call frame)

L'espace mémoire utilisée par une sous-routine pour la sauvegarde du contexte et ses variables locales.

- Alloué à chaque activation (après l'appel) de la sous-routine.
- Pour les processeurs ARM (et beaucoup d'autres) sur la pile.

L'activation record/pile d'appel

Prologue : code ajouté au début de la fonction pour la sauvegarde de contexte et la *création* de l'activation record.

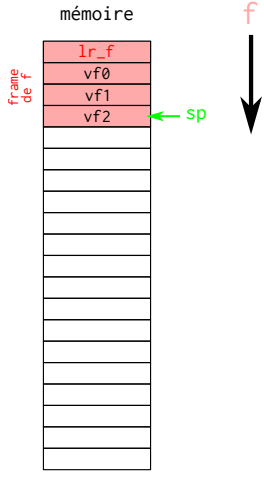
Épilogue : code ajouté à la fin de la fonction pour la *destruction* de l'activation record et la restauration du contexte et le retour.

Remarque: Avec **gcc** pour une cible ARM, vous pouvez utiliser l'attribut de fonction `naked` pour qu'ils ne soint pas ajoutés.

L'activation record/pile d'appel

Qu'y trouve-t-on?

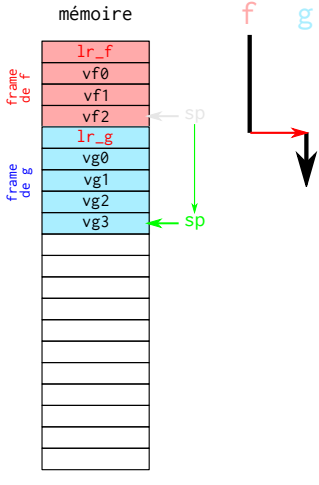
- L'appelant (*caller*) doit sauvegarder son adresse de retour
  - lorsqu'on appelle **BL** le registre **`lr`** est modifié
  - toute fonction qui appelle une sous-routine doit donc prévoir de sauvegarder **`lr`** dans sa frame
  - la sauvegarde se fait dans le prologue et la restauration dans l'épilogue de l'appelant



L'activation record/pile d'appel

Qu'y trouve-t-on?

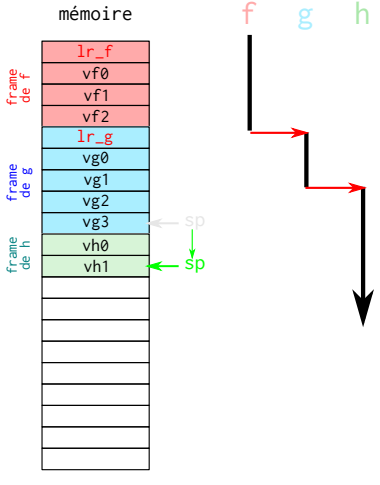
- L'appelant (*caller*) doit sauvegarder son adresse de retour
  - lorsqu'on appelle **BL** le registre **`lr`** est modifié
  - toute fonction qui appelle une sous-routine doit donc prévoir de sauvegarder **`lr`** dans sa frame
  - la sauvegarde se fait dans le prologue et la restauration dans l'épilogue de l'appelant



L'activation record/pile d'appel

Qu'y trouve-t-on?

- L'appelant (*caller*) doit sauvegarder son adresse de retour
  - lorsqu'on appelle **BL** le registre **lr** est modifié
  - toute fonction qui appelle une sous-routine doit donc prévoir de sauvegarder **lr** dans sa frame
  - la sauvegarde se fait dans le prologue et la restauration dans l'épilogue de l'appelant
- Une fonction qui ne fait pas d'appel, peut s'économiser cette sauvegarde.



L'activation record/pile d'appel

le frame pointer (fp)

Tout n'est pas forcément connu à la compilation!

Par exemple, un tableau local dont la taille dépend d'un argument:

```
int f(){
  int vf0, vf1;
  ...
  vf1 = g(4);
  ....
  vf0 = g(2);
  ...
}

int g(int n){
  int T[n];
  ....
  h(...);
  ...
}
```

Il existe aussi la fonction **alloca** qui permet d'allouer de l'espace sur la pile.

L'activation record/pile d'appel

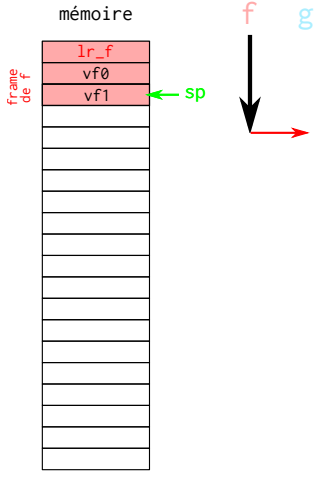
le frame pointer (fp)

- On a besoin d'une référence fixe à la frame
  - on garde donc l'adresse du début de la frame dans le *frame pointer*
  - sur ARM, on utilise le registre **r11(fp)**
  - si une sous-routine utilise le frame pointer, elle doit sauvegarder sa valeur
- le pointeur de pile avance toujours et pointe vers la fin de la frame
  - ce qui est connu à la compilation est référencé par rapport à **fp**
  - ce qui dynamique est référencé par rapport à **sp**

L'activation record/pile d'appel

le frame pointer (fp)

Après l'appel on entre dans le prologue la fonction.



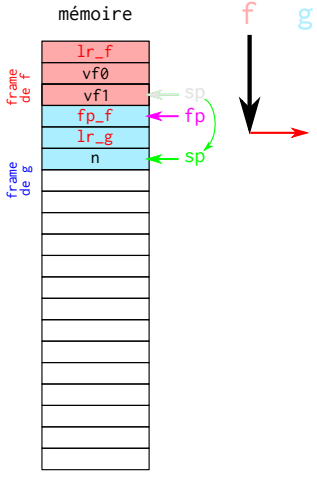
L'activation record/pile d'appel

le frame pointer (fp)

Comme précédemment sauvegarde les registres et on réserve l'espace pour les variables locales

- l'adresse de retour
- comme on va utiliser **fp** on le sauvegarde aussi

On positionne **sp** à la fin de la frame et **fp** au début.

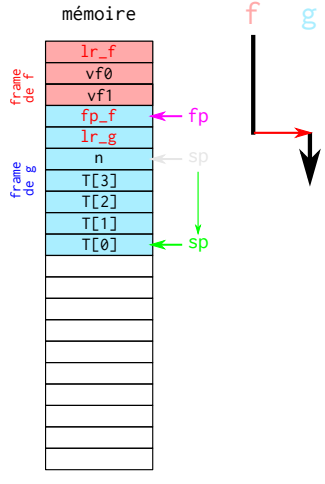


L'activation record/pile d'appel

le frame pointer (fp)

On fait avancer le pointeur de pile (**sp**) pour allouer l'espace nécessaire au tableau.
On a donc:

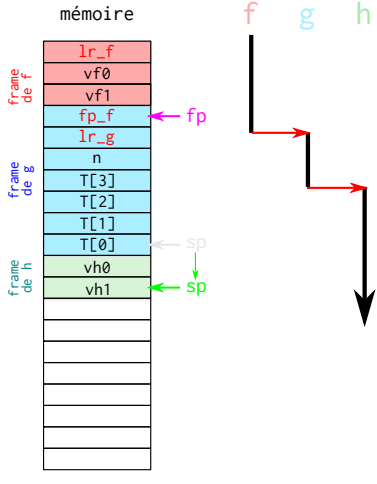
- **fp** qui est stable et permet d'accéder aux variables automatiques
- **sp** qui permet d'accéder aux variables dynamiques sur la pile



L'activation record/pile d'appel

le frame pointer (fp)

comme le pointeur de pile a avancé, on est compatible avec le comportement précédent lors de l'appel à une sous-routine.



Plan

Les registres

Quel rôle?

Sur ARM, il y a 16 registres utilisables par le programmeur. Si on veut de l'interopérabilité, il faut se mettre d'accord sur leur usage:

registre	alias	rôle
r15	pc	compteur programme
r14	lr	adresse de retour (Link register)
r13	sp	pointeur de pile (stack pointer)
r12	ip	Intra-procedure-call register
r11	fp/v8	pointeur de frame (ou registre variable 8)
r10	v7	registre variable 7
r9	v6	registre variable 6
r8	v5	registre variable 5
r7	v4	registre variable 4
r6	v3	registre variable 3
r5	v2	registre variable 2
r4	v1	registre variable 1
r3	a4	argument/registre temporaire 4
r2	a3	argument/registre temporaire 3
r1	a2	argument/résultat/registre temporaire 2
r0	a1	argument/résultat/registre temporaire 1

Les registres

Qui en est responsable?

Qui doit sauvegarder le contenu des registres?

- **L'appelant (caller)**, connait les registres dont il a besoin, il les sauvegarde tous avant de donner la main à la sous-routine. Il les restaurera au retour de l'appel.
  - Si la sous-routine ne s'en sert pas, il les aura sauvegardés pour rien
- **L'appelé (callee)**, connait les registres qu'il va modifier, il les sauvegarde tous puis les restaure avant de rendre la main.
  - Il les a peut-être sauvegardés pour rien.

Pour les processeurs ARM (et pour la majorité des processeurs RISC) on a "beaucoup" registres internes. On peut se partager la responsabilité des registres entre caller et callee.
On n'a besoin de sauvegarder un registre que si on a utilisé tous *nos registres*.

Les registres

Qui en est responsable?

registre	save
r10/v7	callee
r9/v6	callee
r8/v5	callee
r7/v4	callee
r6/v3	callee
r5/v2	callee
r4/v1	callee
r3/a4	caller
r2/a3	caller
r1/a2	caller
r0/a1	caller

- Les registres temporaires, contenant les arguments peuvent être modifiés par l'appelé (callee),
- si l'appelé a besoin de plus de registres, il doit les sauvegarder avant.

Les registres

Qui en est responsable?

Pour les registres spéciaux:

registre	save
`r14/lr`	caller
`r13/sp`	callee
`r12/ip`	caller
`r11/fp`	callee

- `lr`: l'appelant (**caller**) le sauvegarde avant d'appeler une sous-routine,
- `sp/fp`: l'appelé (**callee**) les sauvegarde avant de construire sa frame.

Les registres

Stratégies de sauvegarde

- Le code de sauvegarde et de restauration est ajouté:
  - à chaque appel dans le **caller**,
  - une fois au début et à la fin du **callee**.
  - Si on veut optimiser la taille du programme, on favorisera les registres **callee-save**.
- Une procédure terminale (qui n'appelle pas de sous-procédure) favorisera les **caller-save**.
- Une procédure non terminale favorisera:
  - les **callee-save** si le contenu des registres est nécessaire après l'appel,
  - les **caller-save** pour le reste.

registre	save
r10/v7	callee
r9/v6	callee
r8/v5	callee
r7/v4	callee
r6/v3	callee
r5/v2	callee
r4/v1	callee
r3/a4	caller
r2/a3	caller
r1/a2	caller
r0/a1	caller

Plan

Arguments

Sans trop entrer dans le détail:

- Les 4 premiers arguments s'ils tiennent sur 32 bits utilisent les registres **`r0`**, **`r1`**, **`r2`**, **`r3`**
- Les arguments suivants sont mis sur la pile (**`sp`** pointe sur le premier).
- Si l'argument tient sur 64 bits il utilise deux registres consécutifs.
- Pour les tableaux (en C) c'est l'adresse du premier élément qui est passée.
- Les grosses (plus que 128 bits) structures de données sont copiées sur la pile.

La valeur de retour

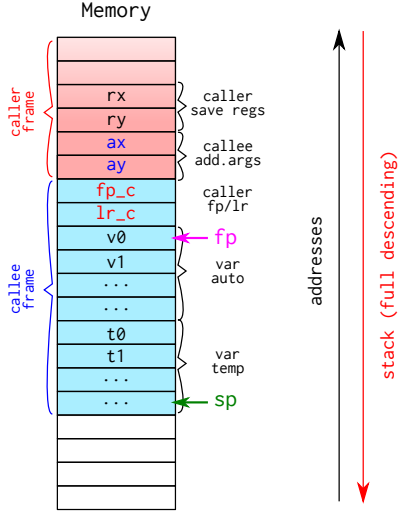
Sans trop entrer dans le détail:

- Si le retour tient sur 32 bits on utilise **`r0`**
- Si le retour tient sur 64 bits on utilise **`r0`** et **`r1`**
- Si c'est plus grand
- Pour les grosses structures de données, l'espace nécessaire sera alloué par l'appelant et un argument supplémentaire sera ajouté pour y passer l'adresse.

L'activation record/pile d'appel

Résumons

- Partage des registres entre caller et callee et convention de sauvegarde
- Organisation de la pile et pointeurs utilisant des registres
- Convention pour les passages d'arguments et des valeurs de retour



Plan

interruptions/exceptions

- Les exceptions sont un mécanisme qui permet d'interrompre le programme en cours et d'exécuter une routine (handler) particulière pour la prendre en charge.
- On va parler d'interruption, quand c'est un évènement intentionnel pour lequel on doit réagir.
  - Souvent cet évènement est provoqué par une source extérieure au processeur.
  - Après avoir réagi à l'interruption on retourne au programme principal
- Alors qu'on parle d'exception quand c'est un évènement imprévu (une erreur par exemple) qui empêche la poursuite du programme.
  - Souvent on doit remédier à l'erreur et on ne revient pas dans l'état initial

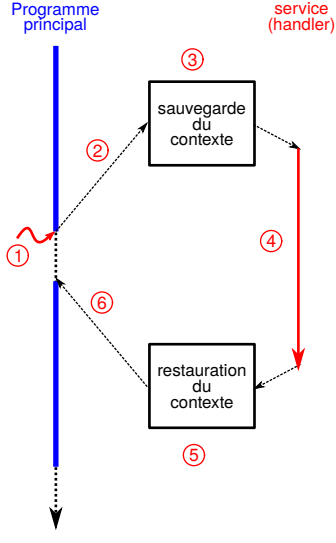
interruptions/exceptions

Par exemple

- une interruption générée par un périphérique
  - un timer qui génère une interruption à intervalles réguliers
  - une unité de communication qui génère une interruption à la réception d'un message
- une instruction qui permet de générer une interruption de façon logicielle
  - ce qui permet de déclencher des mécanismes gérés par un moniteur ou un système d'exploitation
- une exception due à la division par zéro
- un accès interdit en mémoire
  - alignement non respecté
  - rien à cette adresse
  - zone protégée
  - …

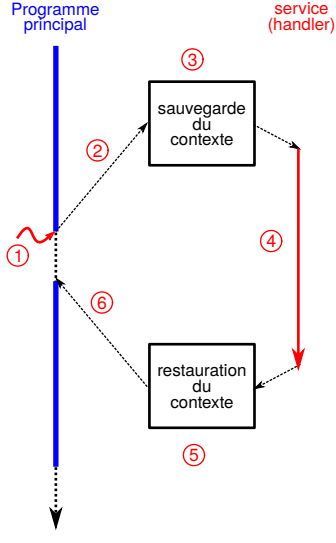
interruptions/exceptions

1. Un évènement survient durant l'exécution



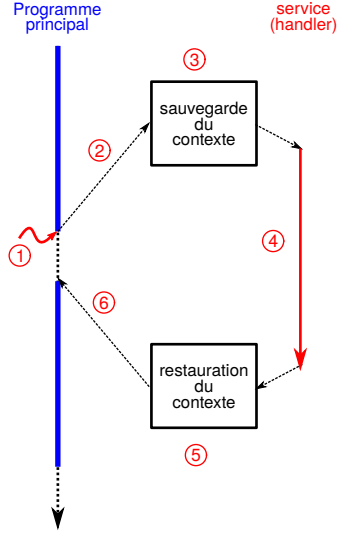
interruptions/exceptions

1. Un évènement survient durant l'exécution
2. On arrête l'exécution du programme



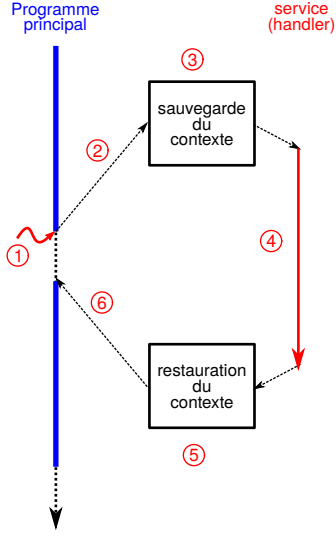
interruptions/exceptions

1. Un évènement survient durant l'exécution
2. On arrête l'exécution du programme
3. On sauvegarde le contexte
   - On peut être amené à identifier l'origine de l'exception



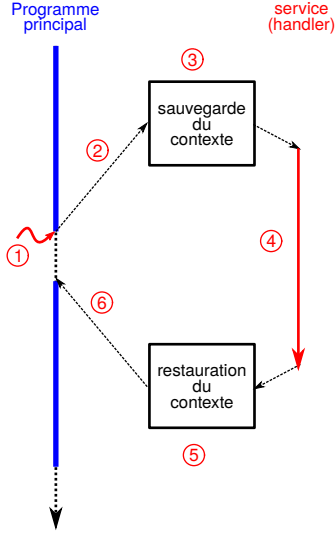
interruptions/exceptions

1. Un évènement survient durant l'exécution
2. On arrête l'exécution du programme
3. On sauvegarde le contexte
   - On peut être amené à identifier l'origine de l'exception
4. On sert l'interruption (*handle*)



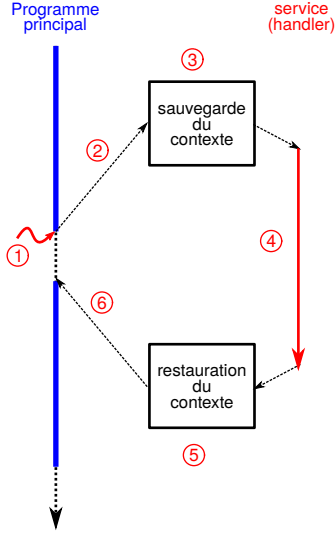
interruptions/exceptions

1. Un évènement survient durant l'exécution
2. On arrête l'exécution du programme
3. On sauvegarde le contexte
   - On peut être amené à identifier l'origine de l'exception
4. On sert l'interruption (*handle*)
5. On restaure le contexte



interruptions/exceptions

1. Un évènement survient durant l'exécution
2. On arrête l'exécution du programme
3. On sauvegarde le contexte
   - On peut être amené à identifier l'origine de l'exception
4. On sert l'interruption (*handle*)
5. On restaure le contexte
6. On redonne la main au programme



Plan

Exception ARM

ARM historiques, Cortex-A et Cortex-R

- L'architecture défini un nombre fixe de sources d'exceptions
- Pour plus de granularité, la gestion se fait de façon logicielle

Exception	**source**
`Reset`	Remise à zéro du processeur
`Undefined`	Instruction non définie
`Supervisor Call`	Interruption logicielle (**svc**)
`Prefetch Abort`	Erreur d'accès aux instructions
`Data Abort`	Erreur d'accès aux données
`IRQ interrupt`	Interruption HW
`FIQ interrupt`	Interruption *rapide*

Exception ARM

ARM historiques, Cortex-A et Cortex-R

- En cas d'exception, le processeur change de mode et **exécute** une instruction de la table des vecteurs
  - Chaque case de la table ne peut contenir **qu'une seule** instruction
  - Il suffit d'y mettre un **branchement** vers un *handler* plus complet
- La table se trouve à une adresse prédéfinie par l'architecture (ou le fabricant)
  - souvent en **0x00000000** et/ou **0xFFFF0000**

Offset	vector
0x00	Reset
0x04	Undefined
0x08	Supervisor Call
0x0C	Prefetch Abort
0x10	Data Abort
0x14	Hyper. trap if supported
0x18	IRQ interrupt
0x1C	FIQ interrupt

Exception ARM

ARM historiques, Cortex-A et Cortex-R

- En fonction de l'exception quelques registres sont sauvegardés en interne du processeur
  - Au moins **cpsr** (dans **spcr**), **pc** et **sp**
  - On peut avoir une pile différente en fonction du mode
- Tout autre sauvegarde de contexte, nécessaire à l'appel d'une fonction en C doit être fait de façon logicielle
- Le retour d'une exception nécessite une instruction particulière (**ERET, MOVS, LDR**)
  - restaure le mode et le **cpsr**

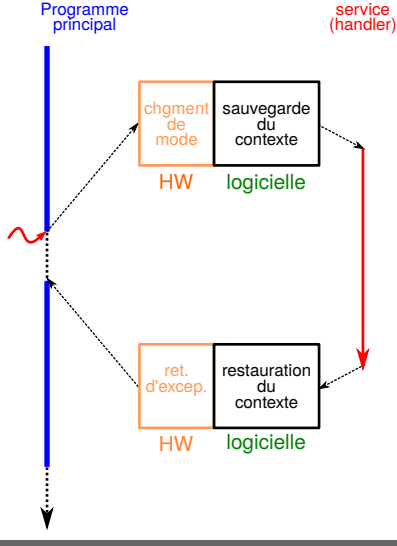
User/Syst.	Superv.	FIQ	IRQ	Abort	Undef
r_0	r_0	r_0	r_0	r_0	r_0
r_1	r_1	r_1	r_1	r_1	r_1
r_2	r_2	r_2	r_2	r_2	r_2
r_3	r_3	r_3	r_3	r_3	r_3
r_4	r_4	r_4	r_4	r_4	r_4
r_5	r_5	r_5	r_5	r_5	r_5
r_6	r_6	r_6	r_6	r_6	r_6
r_7	r_7	r_7	r_7	r_7	r_7
r_8	r_8	r_{8fiq}	r_8	r_8	r_8
r_9	r_9	r_{9fiq}	r_9	r_9	r_9
r_{10}	r_{10}	r_{10fiq}	r_{10}	r_{10}	r_{10}
r_{11}	r_{11}	r_{11fiq}	r_{11}	r_{11}	r_{11}
r_{12}	r_{12}	r_{12fiq}	r_{12}	r_{12}	r_{12}
r_{13} (sp)	r_{13svc}	r_{13fiq}	r_{13irq}	r_{13abt}	r_{13und}
r_{14} (lr)	r_{14svc}	r_{14fiq}	r_{14irq}	r_{14abt}	r_{14und}
r_{15} (pc)	r_{15} (pc)	r_{15} (pc)	r_{15} (pc)	r_{15} (pc)	r_{15} (pc)

Ça ne concerne bien sûr pas le reset

Exception ARM

ARM historiques, Cortex-A et Cortex-R

- En fonction de l'exception quelques registres sont sauvegardés en interne du processeur
  - Au moins **cpsr** (dans **spcr**), **pc** et **sp**
  - On peut avoir une pile différente en fonction du mode
- Tout autre sauvegarde de contexte, nécessaire à l'appel d'une fonction en C doit être fait de façon logicielle
- Le retour d'une exception nécessite une instruction particulière (**ERET, MOVS, LDR**)
  - restaure le mode et le **cpsr**



Ça ne concerne bien sûr pas le reset

Plan

Exceptions sur les Cortex-M

Architecture ARMv6m et ARMv7m

Le fonctionnement des exceptions pour les Cortex-M a été entièrement revu pour:

- simplifier la gestion des exceptions par les programmeurs
  - en gérant certains aspects matériellement,
- garantir l'interopérabilité
  - en supprimant la nécessité d'écrire du code ad hoc pour la sauvegarde et la restauration de contexte,
- permettre de gérer un grand nombre d'interruptions,
  - utile dans un contexte micro-contrôleur,
  - sauvegarde et la restauration de contexte,
  - gestion des priorités.

Exceptions sur les Cortex-M

Architecture ARMv6m et ARMv7m

- plus de granularité permet d'identifier plus simplement la source de l'exception
- les interruptions sont différentiées
- en fonction de l'implémentation on peut avoir jusqu'à 256 interruptions

Num	Exception	source
1	`Reset`	Remise à zéro du processeur
2	`NMI`	Interruption non masquable
3	`HardFault`	Une erreur…
4	`MemManage`	Accès en mémoire interdit
5	`BusFault`	Autres erreurs d'accès
6	`UsageFault`	Instruction non définie ou mal utilisée
7-10	`Réservés`	…
11	`SVCall`	Supervisor Call `SVC`
12	`DebugMonitor`	Pour le débugueur
13	`Réservé`	…
14	`PendSV`	Changement de contexte asynchrone
15	`SystTick`	Interruption du timer interne
16	`Ext. Interrupt 0`	Ext. Inter. 0
…	…	…
16+N	`Ext. Interrupt 0`	Ext. Inter. N

Exceptions sur les Cortex-M

Architecture ARMv6m et ARMv7m

Table des vecteurs

- Contient **l'adresse** des handlers
- Par défaut à la position **`0x00000000`**
  - Le registre **`VTOR`** (*Vector Table Origin*) permet de changer l'adresse de base
- La case **`0`** contien la valeur de **sp** au reset

Offset	Exception
`0x00`	**sp** au reset
`0x04`	`Reset`
`0x08`	`NMI`
`0x0C`	`HardFault`
`0x10`	`MemManage`
`0x14`	`BusFault`
`0x18`	`UsageFault`
…	…
`0x2C`	`SVCall`
`0x30`	`DebugMonitor`
…	…
`0x38`	`PendSV`
`0x3C`	`SystTick`
`0x40`	`Ext. Interrupt 0`
…	…

Exceptions sur les Cortex-M

Architecture ARMv6m et ARMv7m

Table des vecteurs

- Permet de déclarer les handlers comme des fonctions C
- Définir la table des hablers comme une table de pointeurs de fonctions

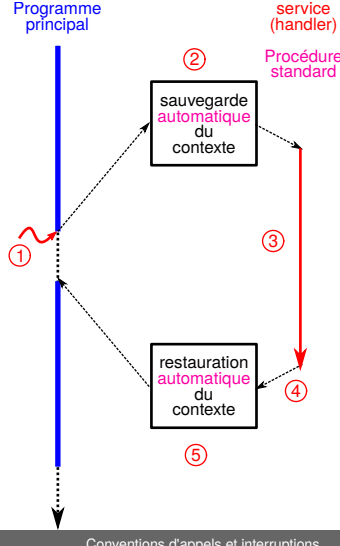
```
extern unsigned int _stack;
void init();
void NMI_Handler();
void HardFault_Handler();
...
void IRQ0_Handler();
...

void* vectors[] = {
   (void*)&_stack     , // initial stack pointer
   init               , // 1  init is called after reset
   NMI_Handler        , // 2  NMI
   HardFault_Handler  , // 3  Hard Fault
   ...
   IRQ0_Handler       , // 16 IRQ0
   ...
```

Si les handlers sont des fonctions en C, il faut **respecter les conventions d'appel**.

Exceptions sur les Cortex-M

Architecture ARMv6m et ARMv7m

1. Un évènement déclenche l'exception

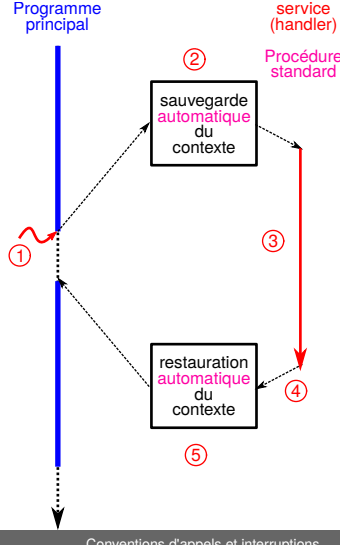


Exceptions sur les Cortex-M

Architecture ARMv6m et ARMv7m

1. Un évènement déclenche l'exception
2. Sauvegarde automatique du contexte sur la pile (l'alignement est conservé)

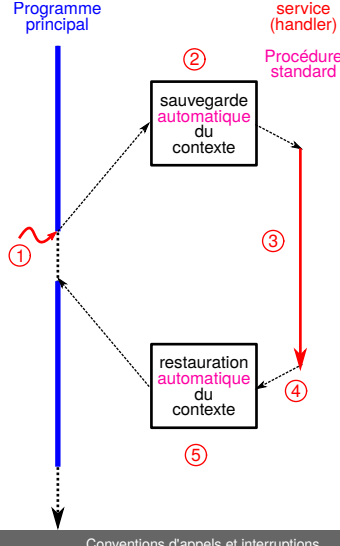
`reg. xPSR`
`Adresse de retour` *dans le programme principal*
`reg. lr(r14)`
`reg. r12`
`reg. r3`
`reg. r2`
`reg. r1`
`reg. r0`



Exceptions sur les Cortex-M

Architecture ARMv6m et ARMv7m

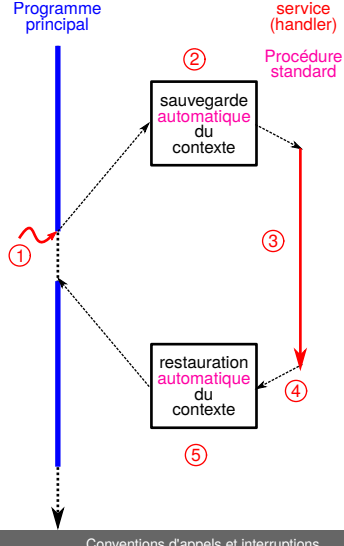
1. Un évènement déclenche l'exception
2. Sauvegarde automatique du contexte sur la pile (l'alignement est conservé)
   - fait par le matériel
   - **sp** est décrémenté
   - le contenu de **lr** est remplacé par une valeur spéciale



Exceptions sur les Cortex-M

Architecture ARMv6m et ARMv7m

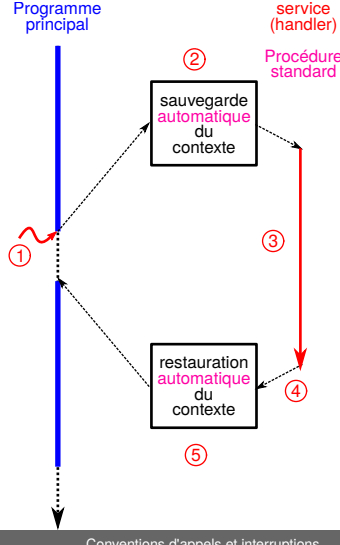
1. Un évènement déclenche l'exception
2. Sauvegarde automatique du contexte sur la pile (l'alignement est conservé)
   - fait par le matériel
   - **sp** est décrémenté
   - le contenu de **lr** est remplacé par une valeur spéciale
3. charge le **pc** avec l'adresse du handler



Exceptions sur les Cortex-M

Architecture ARMv6m et ARMv7m

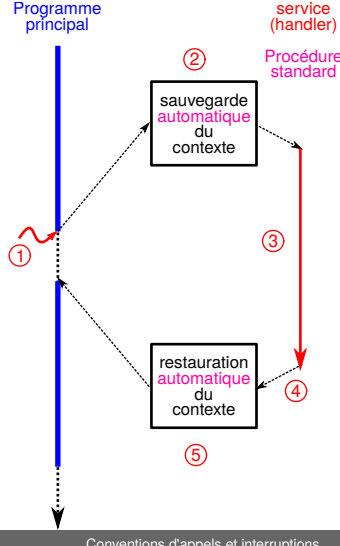
1. Un évènement déclenche l'exception
2. Sauvegarde automatique du contexte sur la pile (l'alignement est conservé)
   - fait par le matériel
   - **sp** est décrémenté
   - le contenu de **lr** est remplacé par une valeur spéciale
3. charge le **pc** avec l'adresse du handler
4. Le retour du handler est détecté en identifiant la valeur spéciale de **lr**



Exceptions sur les Cortex-M

Architecture ARMv6m et ARMv7m

1. Un évènement déclenche l'exception
2. Sauvegarde automatique du contexte sur la pile (l'alignement est conservé)
   - fait par le matériel
   - **sp** est décrémenté
   - le contenu de **lr** est remplacé par une valeur spéciale
3. charge le **pc** avec l'adresse du handler
4. Le retour du handler est détecté en identifiant la valeur spéciale de **lr**
5. Le contexte est restauré à partir de la pile et on retourne au programme d'origine



Exceptions sur les Cortex-M

Architecture ARMv6m et ARMv7m

Plus?

- **NVIC**: *Nested Vectored Interrupt Controler*
  - Pour la gestion automatique des interruptions imbriquées et des priorités
- ARMv7-M Architecture Reference Manual [Lien vers le site d'ARM]