

# Programmation assembleur (ASM)

## *Instructions de branchement*

Prof. Daniel Rossier  
Version 1.7 (2017-2018)

## Plan

- Instructions de **branchement** (*x86 & ARM*)
- Gestion de la pile (*x86 & ARM*)
- Appels de fonction
- Contexte de fonction
- Conventions *ABI*

2

Cours ASM - Institut REDS/HEIG-VD

Ce chapitre est dédié aux instructions de branchement des assembleurs *x86* et *ARM*. Ces instructions permettent de rediriger l'exécution à des endroits différents dans le code, notamment lors d'exécution de boucles, de sauts conditionnels ou inconditionnels, d'appels de fonctions, etc.

Nous étudierons également de manière détaillée le comportement d'une pile (*stack*) en relation avec les fonctions. Les conventions *ABI* (*Application Binary Interface*) principales seront analysées dans ce contexte.

## Instructions de branchement x86 (1/5)

- Branchement **inconditionnel**
  - `jmp label`
- Branchement **conditionnel**
  - `j<cc> label` où **<cc>** est un code de condition
- Branchement **avec lien** (**mémorisation** de l'adresse de retour)
  - *Inconditionnel* ou *conditionnel*
  - Appel de "sous-programme"

3

Cours ASM - Institut REDS/HEIG-VD

L'instruction de base pour l'exécution d'un saut sur x86 est l'instruction ***jmp***.

Le saut est effectué de manière inconditionnelle, c-à-d quelque soit les valeurs du registre d'état *eflags*.

Un saut conditionnel, basé sur les valeurs du registre *eflags*, est possible avec l'instruction ***j<cc>*** où **<cc>** représente un code de condition.

Il faut distinguer les sauts nécessitant de mémoriser l'adresse de retour, des sauts sans mémorisation. Typiquement, un appel de fonction nécessitera de préserver l'adresse de retour afin que l'appelant puisse poursuivre son exécution correctement après l'exécution de la fonction.

## Instructions de branchement x86 (2/5)

- Codes de condition
- Basés sur le registre *eflags*

Instructions	Description	Signe	Flags
JO	Jump if overflow		OF = 1
JNO	Jump if not overflow		OF = 0
JS	Jump if sign		SF = 1
JNS	Jump if not sign		SF = 0
JE	Jump if equal		ZF = 1
JZ	Jump if zero		ZF = 1
JNE	Jump if not equal		ZF = 0
JNZ	Jump if not zero		ZF = 0
JB	Jump if below		
JNAE	Jump if not above or equal	unsigned	CF = 1
JC	Jump if carry		
JNB	Jump if not below		
JAE	Jump if above or equal	unsigned	CF = 0
JNC	Jump if not carry		
JBE	Jump if below or equal		
JNA	Jump if not above	unsigned	CF = 1 or ZF = 1
JA	Jump if above		
JNBE	Jump if not below or equal	unsigned	CF = 0 and ZF = 0
JL	Jump if less	signed	SF <> OF
JNGE	Jump if not greater or equal		
JGE	Jump if greater or equal	signed	SF = OF
JNL	Jump if not less		
JLE	Jump if less or equal		
JNG	Jump if not greater	signed	ZF = 1 or SF <> OF
JG	Jump if greater		
JNLE	Jump if not less or equal	signed	ZF = 0 and SF = OF
JP	Jump if parity		
JPE	Jump if parity even		PF = 1
JNP	Jump if not parity		
JPO	Jump if parity odd		PF = 0
JCXZ	Jump if %CX register is 0		%CX = 0
JECHZ	Jump if %ECX register is 0		%ECX = 0

4

Cours ASM - Institut REDS/HEIG-VD

Sur *x86*, les codes de conditions sont utilisés principalement avec les instructions de saut. Ils se basent exclusivement sur les *flags* du registre *eflags*, *flags* qui peuvent être modifiés durant l'exécution des instructions de traitement. Une instruction de comparaison par exemple met à jour le registre d'état de telle manière à pouvoir "récupérer" les résultats de la comparaison à l'aide des codes de condition inclus dans l'instruction de saut.

Les *flags* de la colonne de droite sont basés sur les bits du registre *eflags*.

Les principales instructions de saut conditionnel sont les suivantes:

*je, jne, jg, jge, jl, jle*

## Instructions de branchement x86 (3/5)

- Différentes formes d'utilisation de l'instruction *jmp*
  - Saut à un emplacement déterminé par un *label*
    - *jmp <label>*
  - Saut à un emplacement défini par une adresse
    - *jmp \*<reg>* où *<reg>* représente un registre
    - *jmp <addr>*
    - Obligatoirement inconditionnel
  - Saut à un emplacement défini indirectement
    - *jmp \*(<reg>)* où *<reg>* est un registre contenant une adresse
    - *jmp \*<addr>*
    - Obligatoirement inconditionnel
  - Saut à l'adresse **stockée en mémoire à l'adresse <addr>**

5

Cours ASM - Institut REDS/HEIG-VD

Le saut à un emplacement dans la mémoire nécessite l'utilisation d'une adresse. La référence à une adresse peut se faire directement dans l'instruction de saut, ou à une adresse stockée dans un registre, ou encore à un *label* déterminant une certaine position.

On distinguera deux modes dans l'utilisation d'une instruction de saut : le mode d'adressage direct ou indirect. Dans le premier mode, l'adresse est codée dans l'instruction (c'est le cas d'un *label* ou d'une adresse donnée sous la forme d'une valeur immédiate sans préfixe). Le second mode nécessite l'utilisation du symbole \*

Le symbole étoile sera utilisé en lien avec un registre; si le registre est encadré par des parenthèses, il s'agit d'une **référence indirecte**. Dans ce cas, c'est l'adresse stockée en mémoire, à l'adresse définie dans le registre, qui sera utilisée comme adresse de saut. Les modes d'adressage indirect étudiés précédemment avec les instructions de transfert peuvent être aussi utilisés avec les instructions de saut. Si une valeur immédiate est donnée après le symbole \*, il s'agira également d'une indirection à cette adresse-ci.

Les instructions de saut faisant référence à un registre ou à une adresse directe ne peuvent être conditionnels (*je*, *jne*, etc.). Ce type de saut est donc toujours **inconditionnel**.

## Instructions de branchement x86 (4/5)

- Soit un registre I/O 32 bits accessible à l'adresse  $0x4000'1000$
- ⇒ Codez en assembleur x86 le comportement suivant:
- Si le bit no. 15 du registre I/O vaut 1, le remettre à 0 et mettre le bit no. 12 à 1.

## Instructions de branchement x86 (5/5)

- Branchement avec stockage de l'adresse de retour
  - ***call*** *<label>*
  - ***call*** \**<reg>*
  - ***call*** \*(*<reg>*)
    - Sauvegarde (**empilement**) de l'adresse de retour sur la pile (descendante)
    - Saut à l'adresse spécifiée
- ***ret***
  - Restitution (**dépilement**) de l'adresse de retour sur la pile
  - Saut à l'adresse de retour

7

Cours ASM - Institut REDS/HEIG-VD

Lorsqu'un saut avec mémorisation de l'adresse de retour est nécessaire, l'instruction ***call*** permet de stocker celle-ci sur la pile avant d'effectuer le branchement à l'adresse correspondante.

L'instruction ***ret*** permet d'effectuer le branchement à l'adresse de retour stockée sur la pile (sommet de la pile). Le branchement s'effectue après avoir retirée l'adresse de la pile.

L'opérande de l'instruction ***call*** est de même forme que celle utilisée dans l'instruction ***jmp***.

## Instructions de branchement *ARM* (1/5)

- Branchement **inconditionnel**
- Branchement **conditionnel**
- Branchement **avec lien** (adresse de retour)
  - *Inconditionnel* ou *conditionnel*
  - **Appel de sous-programme**

Sur *ARM*, les fonctions de redirection de l'exécution sont similaires à ceux étudiées précédemment. Les branchements peuvent s'effectuer de manière conditionnels ou inconditionnels, et la mémorisation de l'adresse de retour peut être nécessaire.



## Instructions de branchement ARM (2/5)

- **b** <label>
- **b<cc>** <label>
  - Branchement à une adresse dans le code spécifié par le *label*.
  - Un code de condition peut être apposé comme suffixe.
- **bl** <label>
- **bl<cc>** <label>
  - Branchement **avec stockage** de l'adresse de retour (soit l'instruction qui suit *bl*) dans le registre **lr (r14)**.
  - Un code de condition peut être apposé comme suffixe.

9

Cours ASM - Institut REDS/HEIG-VD

Le branchement s'effectue avec l'instruction **b** ou **bl** si la mémorisation de l'adresse de retour est nécessaire.

Sur *ARM*, on se rappelle que **toute instruction peut être exécutée de manière conditionnelle** (exécution conditionnelle des instructions), en rajoutant un code de condition comme suffixe au mnémonique de base. Il en va de même pour les instructions *b* et *bl*. De cette manière, il est aisé de réaliser un saut conditionnel se basant sur l'état des bits du registre *cpsr*.

Lors de l'exécution d'un *bl* (*branch with link register*) avec mémorisation de l'adresse de retour, cette dernière est stockée dans le **registre r14 (lr)** (et non sur la pile). Ainsi, lorsqu'une fonction termine son exécution, il suffit de placer la valeur du registre *lr* dans **r15 (pc)** pour effectuer le branchement à l'adresse de retour.

## Instructions de branchement ARM (3/5)

- Format des instructions
  - Branchement avec (*bl*) et sans lien (*b*)



- Les adresses de branchement doivent être **alignées** (sur 4 octets).
- L'instruction est codée **relativement au PC**
  - L'offset est sur une plage de  $\pm 32$ Mbytes

10

Cours ASM - Institut REDS/HEIG-VD

Pareil aux autres instructions sur *ARM*, l'instruction de branchement est encodé sur 32 bits; ce qui signifie que l'adresse contenu dans l'instruction pour le saut ne peut pas s'étendre sur les 32 bits. C'est pourquoi le compilateur d'assemblage traduit l'adresse (correspondant au *label*) en une adresse relative par rapport à la position courante du *PC*. On voit sur le format de l'instruction que seul 24 bits sont à disposition pour exprimer le déplacement relatif. Le déplacement pouvant être avant la position courante, il faut donc stocker un bit de signe. Il reste 23 bits.

Les instructions ARM doivent être alignées (sur 4 *bytes*); ainsi les deux derniers bits de poids faible d'une adresse pointant vers une instruction seront **toujours à 0**. Il n'est donc pas nécessaire de "stocker" ces deux bits dans l'adresse: au final, on pourra donc exprimer une adresse relative sur 23 bits + 2 bits = 25 bits, soit  $2^5$  Mo = 32 Mo.

## Instructions de branchement ARM (4/5)

- Exemples d'utilisation

```
b <label>          @ pc = <label>
bne <label>       @ si ne alors pc = <label>
                          @ sinon pc = pc + 4

bl  une_fonction  @ lr = adresse de l'inst. suivante
                          @ pc = adresse de une_fonction
...

une_fonction:
...

mov  pc, lr        @ pc = lr (adresse de retour)
```

Comme le montre l'exemple ci-dessus, il est possible sur ARM de modifier directement le registre *pc* (*r15*). Ce n'est pas le cas sur x86.

## Instructions de branchement *ARM* (5/5)

- Soit un registre I/O 32 bits accessible à l'adresse  $0x4000'1000$
- ⇒ Codez en assembleur *ARM* le comportement suivant:
- Si le bit no. 15 du registre I/O vaut 1, le remettre à 0 et mettre le bit no. 12 à 1.

## Gestion de la pile sur x86 (1/6)

- Gestion de la pile
  - Stockage d'une adresse de retour
  - Stockage des registres
  - Stockage des variables locales (mémoire processus)
- Instructions de **transfert**
  - La pile est une **zone mémoire**.
  - Accès via des adresses comme toute autre zone

13

Cours ASM - Institut REDS/HEIG-VD

La pile est une structure de donnée dynamique bien connue et sujette à des utilisations très diverses.

Sa première utilisation intervient dans l'appel d'une fonction : la sauvegarde de l'adresse de retour, la sauvegarde des valeurs de registre, le passage d'arguments, les variables locales, etc. Etant donné qu'elle joue un rôle crucial dans le branchement de l'exécution courante à certaines adresses, elle est également une zone de donnée critique. Sa corruption peut aboutir à une exécution erronée, voire à l'injection de code malveillant.

La gestion de la pile est essentiellement assurée à un niveau logiciel. Dans le langage assembleur, le programmeur doit prendre en charge la gestion intégrale de la pile : il peut également décider de ne pas en utiliser. Dans les langages de plus haut niveau, c'est le compilateur qui produira le code nécessaire lié à la gestion de la pile.

Il est à noter que la pile est une zone mémoire pouvant être accéder en lecture/écriture avec les instructions de transfert conventionnelles. En revanche, on peut trouver des instructions de transfert spécifiques pour la gestion de la pile, comme *push* ou *pop* (comme sur *x86*). De plus, la pile doit contenir des valeurs *alignées* sur le mot (ou parfois demi-mot) machine (32 bits ou 16 bits), ce qui s'implifie notamment l'encodage des instructions.

## Gestion de la pile sur x86 (2/6)

- Instructions **d'empilement / dépilement**
  - Pile **descendante**
  - Registre *esp* (*stack pointer*) pointe vers une **case pleine**
- **push** *<src>*
  - 1)  $esp = esp - 4$  (si opérande sur 4 octets)
  - 2) Empilement de la valeur correspondant à *<src>* sur la pile
- **pop** *<dest>*
  - 1) Dépilement de la valeur dans *<dest>*
  - 2)  $esp = esp + 4$  (si opérande sur 4 octets)
- Opérande à **16 ou 32 bits** (uniquement)

14

Cours ASM - Institut REDS/HEIG-VD

Dans la plupart des environnements, la pile est **descendante** (les adresses décroissent lorsque l'on rajoute des éléments sur la pile). Du point de vue du processeur, un registre est généralement réservé pour le stockage de l'adresse "pointant" vers le sommet de la pile; c'est une registre de type "*stack pointer*".

Rajouter ou enlever une valeur de la pile implique donc une mise à jour du *stack pointer*. Sur *x86*, c'est le registre *esp* qui est dédié au *stack pointer*.

Un autre aspect lié à la gestion du pointeur de pile est par rapport à sa position courante : réfère-t-elle une "case" pleine ou vide ? En effet, sur *x86*, le pointeur de pile réfère toujours une **case pleine**, c-à-d qu'il pointe vers le dernier élément inséré sur la pile. L'empilement d'une nouvelle valeur commence donc par la **décréméntation** du registre *sp* du nombre d'octets correspondants.

L'instruction **push** met à jour le registre *sp*, puis stocke la valeur sur la pile. A l'inverse, l'instruction **pop** récupère la dernière valeur sur la pile, la transfère dans l'opérande de l'instruction, puis met à jour le registre *sp*.

## Gestion de la pile sur x86 (3/6)

- Exemples d'utilisation

- `pushl %eax`           # `sp = sp - 4`
- `pop %bx`           # `sp = sp + 2`
  
- `popw $0x1654`       # `sp = sp + 2`
- `pushl $0xaabb`     # `sp = sp - 4`
  
- `pushw (%eax)`       # `sp = sp - 2`  
                          (accès mémoire à l'adresse contenu  
                          dans `%eax`)
  
- `popl -5(%eax)`       # `sp = sp - 2`  
                          (accès mémoire à l'adresse `(%eax) - 5`)

## Gestion de la pile sur x86 (4/6)

- Instructions **d'empilement / dépilement**
  - Empilement/dépilage des 8 registres principaux  
*eax – ecx – edx – ebx – esp- ebp – esi – edi* (dans cet ordre)
  - ***pusha***
    - 1)  $esp = esp - 32$
    - 2) Empilement des 8 registres 32 bits
  - ***popa***
    - 1) Dépilement des 8 registres 32 bits
    - 2)  $esp = esp + 32$
  - Variantes
    - 32 bits : ***pushal/popal***
    - 16 bits : ***pushaw/popaw***

16

Cours ASM - Institut REDS/HEIG-VD

Il existe des variantes des instructions *push* et *pop*.

Les instructions *pusha* et *popa* permettent le transfert de l'ensemble des registres principaux, à l'exception du registre d'état *eflags*. Ce dernier peut être sauvegardé avec les instructions ***pushf*** et ***popf***.

La variante 32 bits peut également être explicitement spécifiée avec les instructions *pushl* et *popl*. Logiquement, la variante 16 bits est utilisée avec les instructions *pushw* et *popw*.

Sur les modèles ci-dessus, les instructions de transfert – dédiées à la gestion de la pile – suivantes sont disponibles :

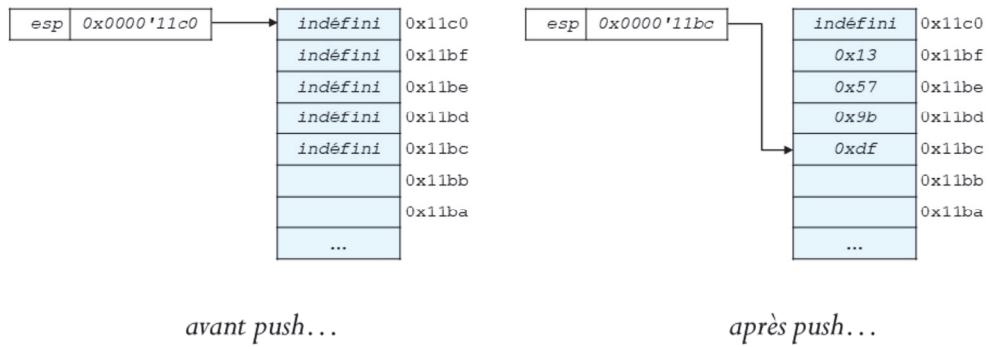
- ***pushal / popal***
- ***pushaw / popaw***
- ***pushfl / popfl***
- ***pushfw / popfw***



## Gestion de la pile sur x86 (5/6)

- `push %eax`

<code>eax</code>	<code>0x1357'9bdf</code>
------------------	--------------------------



17

Cours ASM - Institut REDS/HEIG-VD

Le pointeur `sp` réfère la dernière valeur stockée sur la pile.

## Gestion de la pile sur x86 (6/6)



- Considérez les valeurs dans les registres, la zone mémoire et les instructions ci-dessous.  
⇒ Déterminez les valeurs des registres à chaque exécution d'une instruction.

<i>eax</i>	<i>0x0002'0000</i>
<i>ebx</i>	<i>0x0004'0000</i>
<i>ecx</i>	<i>0x0006'0000</i>
<i>esp</i>	<i>0x0010'0124</i>

	0x10'0125
	0x10'0124
	0x10'0123
	0x10'0122
	0x10'0121
	0x10'0120
	0x10'011f
...	0x10'011e

```
pushl $0x13487
pop %ax
pop %bx
push %ebx
pop %ecx
pushl $0x100122
pop %ebx
pushw $0x987
pushw (%ebx)
popw -1(%ebx)
```

## Gestion de la pile sur *ARM* (1/6)

- Gestion du pointeur de pile
  - **sp** pointe vers une zone mémoire *réservée*.
  - Position courante: case **vide** ou **pleine**
  - Pile **ascendante** ou **descendante**

Sur *ARM*, la pile n'a pas de sémantique particulière. C'est donc au développeur qu'incombe la responsabilité totale de gérer la pile à l'aide des instructions de transfert disponibles.

La pile peut évoluer de manière croissante ou décroissante. Typiquement, pour des raisons d'organisation mémoire (mais sans rapport avec le processeur), la pile est descendante, ce qui signifie que le pointeur *sp* est décrémenté à chaque empilement, et incrémenté à chaque dépilement.

Afin de faciliter sa gestion, et pour des raisons de performance également, les éléments de la pile sont *alignés* sur la taille d'un registre, à savoir 32 bits pour *ARM*. Il est donc **indispensable** que les adresses des éléments de la pile (le premier élément à fortiori) soient des multiples de 4, et que l'empilement et le dépilement se fasse par éléments de cette taille.

## Gestion de la pile sur ARM (2/6)

- **Transferts multiples**
- Plusieurs **mots** peuvent être transférés à la fois
  - `<ldm|stm>{cond}<IA|IB|DA|DB>Rn{!},<Rlist>{^}`

Nom	Mnémonique
Pre-increment load	LDMIB
Post-Increment load	LDMIA
Pre-Decrement load	LDMDB
Post-Decrement load	LDMDA
Pre-Increment store	STMIB
Post-Increment store	STMIA
Pre-Decrement store	STMDB
Post-Decrement store	STMDA

20

Cours ASM - Institut REDS/HEIG-VD

Les instructions **stm** et **ldm** sont utilisées pour des transferts multiples de registres en mémoire (ou vice-versa). Bien qu'elles peuvent être utilisées avec n'importe quelle registre, elles sont utilisées principalement pour gérer la pile, en conjonction avec le registre utilisé comme pointeur de pile.

C'est pourquoi, le registre *Rn* ci-dessus correspond la plupart du temps à **r13**, ou **sp**.

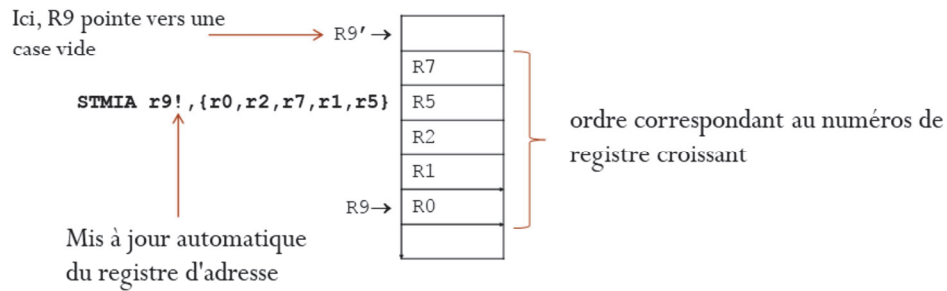
L'opérande `<Rlist>` décrit le-s registre-s impliqués dans le transfert. Les registres sont encadrés avec des accolades. Voici quelques exemples d'opérande de ce type :

```
{r0}
{r0, r3, r9, lr}
{r0-r2, r5}
{r0-r10}
etc.
```

Il est à noter que l'ordre des registres à l'intérieur des accolades n'est pas important. En effet, l'encodage des instructions **stm** et **ldm** comprend un champ de 15 bits qui permettent de déterminer quels registres sont concernés. L'ordre dans lequel ces registres seront transférés en mémoire suivront ainsi un ordre croissant (cf exemples ci-après).

## Gestion de la pile sur ARM (3/6)

- Fonctionnement
  - Les registres énoncés entre accolade sont placés selon leur ordre de numéro (donc selon l'ordre  $r_0, r_1, r_2, \dots$ ).
  - L'incrément correspond au nombre de registres entre accolade.



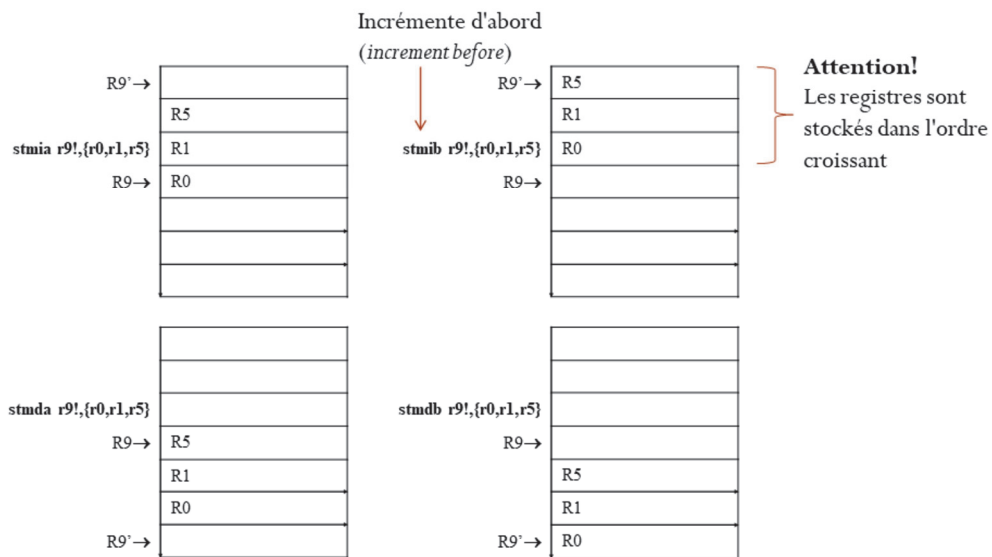
21

Cours ASM - Institut REDS/HEIG-VD

Dans l'exemple ci-dessus, le fait que le registre  $r9$  pointe vers une case vide correspond à un choix du programmeur; il pourrait très bien pointer vers une case pleine (et devrait dans ce cas être incrémenté avant).

**Attention!** Il ne faut pas confondre l'incrément du registre lié à l'opération de stockage (code *IA*, *IB*, *DA*, *DB*) avec le symbole **!** qui signifie, dans ce dernier cas, que le registre d'adresse est **mis à jour** après l'opération de transfert.

## Gestion de la pile sur ARM (4/6)



22

Cours ASM - Institut REDS/HEIG-VD

Le sens de lecture/écriture est imposé par le suffixe de l'instruction (**ia**, **ib**, **da**, **db**).

On observera qu'une écriture de type **IA** (*Increment Before*) nécessitera l'utilisation d'une lecture de type **DB** (*Decrement Before*) afin de restituer les données correctement. Respectivement, une écriture de type **IB** (*Increment Before*) nécessitera une lecture de type **DA** (*Decrement After*).

De plus, on remarque une particularité au niveau de stockage des registres : dans le cas d'une pile décroissante, les registres impliqués dans le transfert sont toujours placés dans un ordre ascendant; cet ordre ne dépend pas du sens de la pile.

## Gestion de la pile sur ARM (5/6)

- Pile décroissante
  - C'est l'approche adoptée dans les OS afin de gérer au mieux la gestion dynamique du tas (*heap*) et de la pile (*stack*) (cf cours SYE).
- Le pointeur de pile **sp** pointe sur le sommet de la pile (case pleine).
  - Lorsque la pile est vide, le **sp** pointe 4 octets avant.
- Instructions correspondantes

```
stm{cond}fd sp!, {<regs>} @ push  
ldm{cond}fd sp!, {<regs>} @ pop
```

Bien que les instructions **stmfd** et **ldmfd** peuvent être traduites avec des **stmdb** et **ldmia** respectivement, elles sont de loin plus pratiques à utiliser. En effet, le terme **fd** est identique pour l'opération *load* et *store*. Elle met bien en évidence que, dans le cadre d'une utilisation "standard" d'une pile, celle-ci est "*full descending*", c-à-d descendante avec une référence courante sur la dernière valeur (sommet de la pile).

## Gestion de la pile sur ARM (6/6)



- Soit une architecture *little-endian* et le code *ARM* ci-dessous.

⇒ Montrez l'évolution de la pile en mémoire (avec les adresses).

```
1
2 ldr sp, =0x10124          (1 case = 1 byte)
3 mov r0, #0x26
4 ldr r1, =0x1234
5 ldr r2, =0xffee
6
7 stmfd sp!, {r0-r2}
8 ldr r3, =0x1011f
9 ldr r2, [r3]
10 strb r0, [r3, #2]
11 ldmfd sp, {r2}
12
13 ldmfd sp!, {r3-r5}
14
```



↑  
sens de  
l'adressage



## Appels de fonction (1/5)

- Schéma général d'un appel de fonction
  - Stockage des **arguments** (pile et/ou registres)
  - Saut à une adresse avec **mémorisation de l'adresse de retour**
    - *ARM*: via un registre (instruction *bl*)
    - *x86*: via la pile (instruction *call*)
  - **Réservation** des variables locales sur la pile
  - **Utilisation des registres de travail**
    - Préservation éventuelle des valeurs courantes sur la pile (*ARM*)
  - Gestion de la valeur de retour et restitution des registres sauvegardés
  - **Saut à l'adresse de retour**
    - Réajustage de la pile, récupération de la valeur de retour

25

Cours ASM - Institut REDS/HEIG-VD

La traduction d'un appel de fonction écrit dans un langage de haut niveau (*C*, *C++*, etc.) conduit à la production d'un ensemble d'instructions assembleur respectant une certaine méthodologie. Cette dernière est souvent dictée par des standards de compilation dépendant de l'architecture machine sur laquelle le code est compilé. C'est pourquoi, l'utilisation d'une *toolchain* correspondante au processeur cible est importante.

L'approche générale pour la réalisation d'un appel de fonction en assembleur est décrite ci-dessus. Elle fait intervenir le stockage des **arguments** de la fonction, via des registres ou via la pile, la mémorisation de **l'adresse de retour** lors du branchement (on doit revenir de la fonction), la réservation de la place sur la pile pour le stockage des **variables locales**, la sauvegarde éventuelle des **registres de travail**, c-à-d les registres qui vont être utilisés/modifiés à l'intérieur de la fonction, et dont il est important de préserver les valeurs courantes (avant l'appel de la fonction).

Finalement, la fin de la fonction doit restaurer les registres, récupérer l'adresse de retour, réajuster la pile (ôter les variables locales) et effectuer le branchement de retour.

Le code d'entrée de la fonction est aussi appelée **prologue** de la fonction et le code de sortie **épilogue** de la fonction.

## Appels de fonction (2/5)

**ARM**

```
.globl start
start:
    mov r0, #-13          @ 1er argument
    mov r1, #0x25        @ second argument
    bl une_fonction
    @ r0 contient la valeur de retour
    ...
une_fonction:           @ point d'entrée de la fonction
    stmfid sp!, {r3, r4} @ sauvegarde des registres de travail
    add r3, r0, #10
    ldr r4, =0x35
    ...
    ldmfd sp!, {r3, r4} @ restitution des registres de travail
    mov pc, lr          @ fin de la fonction, retour à l'appelant
```

26

Cours ASM - Institut REDS/HEIG-VD

Le passage des arguments s'effectue par les registres (*r0*, *r1*). Les registres de travail sont sauvegardés sur la pile à l'entrée de la fonction.

L'adresse de retour est stockée dans le registre *lr* lors de l'exécution de l'instruction *bl* (branchement avec mémorisation de l'adresse de retour dans *lr*).

Bien que le pointeur de pile puisse pointer vers n'importe quel octet de la mémoire, un alignement sur 4, 8 ou 16 octets peut être exigé selon les conventions d'appel de fonction (cf plus loin).

## Appels de fonction (3/5)

x86

```
.globl start

start:
    pushl $0x25
    pushl $-13
    subl $4, %esp      # Stockage de la valeur de retour
    call  une_fonction

    popl  %eax        # Valeur de retour
    addl  $8, %esp    # Réajustage de la pile
    ...

une_fonction:        # point d'entrée de la fonction

    movl  8(%esp), %ecx # premier argument
    movl  12(%esp), %ebx # second argument
    ...
    movl  %edx, 4(%esp) # valeur de retour
    ret   # fin de la fonction, retour à l'appelant
```

27

Cours ASM - Institut REDS/HEIG-VD

Le passage d'arguments s'effectue cette fois-ci sur la pile. Dans l'exemple ci-dessus, les arguments sont passés **de droite à gauche**, c-à-d que le premier argument sera le plus proche du sommet de la pile. C'est la **convention habituelle sur x86**.

Dans l'exemple ci-dessus, on réserve également un emplacement sur la pile pour réceptionner la valeur de retour. Il suffit pour cela de décrémenter le pointeur de pile de 4 octets (il n'est pas utile de stocker quoique soit pour la valeur de retour, à l'appel de la fonction).

Ici, l'utilisation de l'instruction *call* a pour effet de stocker l'adresse de retour sur la pile. C'est pourquoi, dans la fonction, le premier argument se trouve juste avant la valeur de retour qui se trouve elle-même juste avant l'adresse de retour (*offset* de 8 octets).

L'instruction *ret* dépile l'adresse de retour et effectue le branchement. La valeur de retour pourra ainsi être récupérée directement à l'exécution du prochain *pop*.

## Appels de fonction (4/5)

- Fonctions imbriquées
  - Plusieurs adresses de retour
    - Préservation des adresses de retour sur la pile
    - **ARM** Le registre *lr* doit être sauvegardé
    - **x86** Le registre *%ebp* permet de maintenir la base de la pile à chaque niveau d'appel.

28

Cours ASM - Institut REDS/HEIG-VD

L'imbrication de fonctions nécessite de sauvegarder le contexte de la fonction en cours d'exécution. L'utilisation d'une pile permet de faciliter grandement cela. Dans le cas de *ARM*, il faudra préserver le registre *lr* sur la pile avant d'effectuer le branchement avec *bl* puisque cette instruction va stocker l'adresse de retour dans *lr*.

Sur *x86*, on peut garder la trace de chaque contexte de fonction (que l'on appelle également *stack frame* en anglais) grâce au **registre *ebp***. Le registre *ebp* (*base pointer*) permet en fait de mémoriser le *stack pointer* à l'entrée d'une fonction afin de faciliter l'accès aux arguments (*offset* positif) et aux variables locales (*offset* négatif) comme nous le verrons plus tard.

Il est utile de mentionner que sur *ARM*, un registre analogue (*r11*, avec l'alias *fp* pour *frame pointer*) est également souvent utilisé.

## Appels de fonction (5/5)

**ARM**

```
main:
  ...
  bl  une_fonction
  ...

une_fonction:
  stmfd sp!, {r0-r2, lr}
  bl  autre_fonction
  ... @ On utilise r0-r2
  ldmfd sp!, {r0-r2, pc}

autre_fonction:
  stmfd sp!, {r0-r4}
  ... @ On utilise r0-r4
  ldmfd sp!, {r0-r4}

  mov  pc, lr
```

29

Cours ASM - Institut REDS/HEIG-VD

Dans l'exemple ci-dessus, la sauvegarde du registre *lr* est effectuée à l'entrée de la fonction. Ainsi, le branchement suivant (avec mémorisation) pourra sans autre altérer le contenu du registre *lr*.

La sauvegarde de *lr* dans la fonction de second niveau n'est pas nécessaire puisqu'on suppose qu'aucun branchement n'est effectué par la suite.

A l'épilogue de la fonction de premier niveau (*une\_fonction*), la valeur sauvegardée du registre *lr* est directement placée dans le registre *pc* par l'instruction *ldmfd*.

## Contexte de fonction (1/5)

x86

```
main:
  ...
  call une_fonction
  ...

une_fonction:
  push %ebp
  mov  %esp, %ebp
  ...
  call autre_fonction
  ...
  mov  %ebp, %esp  } instruction leave
  pop  %ebp
  ret

autre_fonction:
  push %ebp
  mov  %esp, %ebp
  ...

  leave
  ret
```

30

Cours ASM - Institut REDS/HEIG-VD

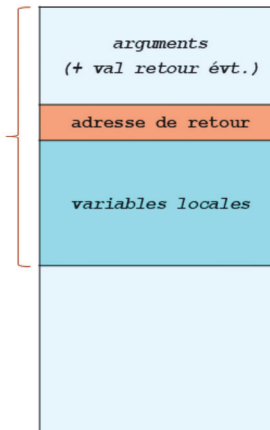
Le code ci-dessus présente la manière conventionnelle de gérer le registre *ebp* à l'entrée et à la sortie d'une fonction. On commence par sauvegarder l'état courant de *ebp* (celui-ci peut maintenir la référence vers un contexte de fonction en cours d'exécution). Puis, la valeur courante de *esp* est sauvegardée dans ce registre, devenant ainsi la référence vers le nouveau contexte de fonction.

La restauration du registre *ebp* en épilogue suit toujours le même schéma : on réajuste le pointeur de pile avec la valeur de *ebp* – ce qui a pour effet d'ôter le contenu lié aux variables locales – puis on restaure l'ancienne valeur de *ebp* avant d'effectuer le branchement vers l'adresse de retour. Le jeu d'instruction x86 offre l'instruction *leave* qui est une équivalence à l'exécution du *mov* et du *pop* liée à la restauration de la pile et du registre *ebp*.

## Contexte de fonction (2/5)

- Variables **locales**
- Stockage sur la pile
  - Simplifie l'utilisation des registres
  - Utilisation restreinte à la fonction
- Accès aux variables avec des instructions de transfert

Stack frame



- Organisation de la pile
  - *Stack frame*

**x86**

16 (%ebp)	- 3 <sup>ème</sup> paramètre de la fonction
12 (%ebp)	- 2 <sup>ème</sup> paramètre de la fonction
8 (%ebp)	- 1 <sup>er</sup> paramètre de la fonction
4 (%ebp)	- adresse de retour
0 (%ebp)	- valeur de %ebp préservée (ancienne)
-4 (%ebp)	- 1 <sup>ère</sup> variable locale
-8 (%ebp)	- 2 <sup>ème</sup> variable locale
-12 (%ebp)	- 3 <sup>ème</sup> variable locale

31

Cours ASM - Institut REDS/HEIG-VD

Le passage d'arguments dans une fonction et la récupération d'une valeur de retour nécessitent une convention de passage entre le compilateur et l'architecture matérielle. Par exemple, les arguments peuvent être passés par des registres, pour autant qu'il y a suffisamment de registre à disposition, ou par la pile si l'on veut augmenter la portabilité.

Le choix d'utiliser des registres ou une pile aura bien entendu une conséquence sur les performances: la manipulation de registres ne nécessite **pas** de transfert mémoire. Lors d'appels fréquents à des fonctions – comme par exemple à des appels système – on préférera alors une solution basée sur des registres. En revanche, la structure d'une pile permet de gérer plus facilement les appels récursifs.

L'utilisation d'une zone de mémoire dédiée est aussi une possibilité, mais l'organisation des données (arguments/valeur de retour) doit faire l'objet d'une convention et peut réduire la portabilité du code. Cette approche peut être utilisée pour le transfert de paramètres volumineux (peu utilisé).

## Contexte de fonction (3/5)

x86

```
1
2 int une_fonction(int arg1, char arg2) {
3     int count;
4     char current;
5
6     count = arg1;
7     current = arg2;
8
9     ...
10
11    return count;
12 }
13
14 int main(int argc, char *argv[]) {
15     int ret;
16
17     ret = une_fonction(0x26, 'z');
18 }
```

```
1
2 une_fonction:
3     push %ebp           # Préserve %ebp courant
4     mov %esp, %ebp
5     sub $8, %esp       # Zone pour les var. locales
6
7     movl 8(%ebp), %eax  # count = arg1
8     movl %eax, -4(%ebp)
9
10    movl 12(%ebp), %eax # current = arg2
11    movb %al, -5(%ebp)
12
13    ...
14
15    mov -4(%ebp), %eax  # Valeur de retour dans %eax
16
17    leave               # Réajustage pointeur de pile
18    ret
19
20 main:
21    push %ebp
22    mov %esp, %ebp
23
24    pushl $'z'          # Empilement des arguments
25    pushl $0x26
26
27    call une_fonction
28
29    add $8, %esp
30
31    mov %eax, (%esp)
32
33    mov %ebp, %esp
34    pop %ebp
35    ret
36
```

32

Cours ASM - Institut REDS/HEIG-VD

Le code ci-dessus montre des analogies avec l'assembleur *ARM* dans le fonctionnement des appels de fonction. Le registre *ebp* représente l'équivalent du registre *fp*. Il n'est en revanche pas nécessaire de stocker l'adresse de retour puisque celle-ci a déjà été stockée lors du *call*.

Contrairement à l'exemple *ARM* précédent, les arguments sont stockés sur la pile avant l'appel de la fonction.

L'épilogue de la fonction *main* montre l'équivalent de l'instruction *leave*.



## Contexte de fonction (4/5)

**ARM**

- *Stack frame* sur ARM
- Utilisation de la pile pour les arguments seulement si nécessaire
- Exemple général

[fp, #12]		- 3ème paramètre de la fonction
[fp, #8]		- 2ème paramètre de la fonction
[fp, #4]		- 1er paramètre de la fonction
fp	lr	- adresse de retour
[fp, #-4]	ip	- valeur courante de sp
[fp, #-8]	fp	- valeur courante de fp
[fp, #-12]		- 1ère variable locale
[fp, #-16]		- 2ème variable locale
[fp, #-20]		- 3ème variable locale

33

Cours ASM - Institut REDS/HEIG-VD

A la différence du *x86*, le registre *sp* sera stocké sur la pile. Mais le stockage de *sp* sur la pile **pose problème avec l'instruction *stmfd sp!*** Pour contourner ce problème, on a recours au registre *r12* (ou *ip* pour *Intra-Procedure-call*) pour placer temporairement la valeur de *sp* afin de le stocker sur la pile.

De plus, la mémorisation de *sp* dans *ip* facilite le positionnement du registre *fp* qui maintient une référence vers le contexte de fonction. Celui-ci pourra récupérer l'ancienne valeur de *sp* diminuée de 4 octets (c'est le début du contexte).

Ainsi, *fp+0* pointera vers l'adresse de retour (*lr*), *fp-4* vers le *stack pointer* dont la valeur est celle à l'entrée de la fonction, *fp-8* vers l'ancienne valeur de *fp*, et *fp-12* sera la première variable locale.

L'épilogue de la fonction montre que le pointeur de pile est simplement réajusté à l'aide de la valeur stockée dans *fp*. En effet, les 12 octets soustraits à *fp* permet de se positionner juste après les 3 registres ( $3 * 4$ ) qui doivent être restaurés avec l'instruction de transfert *ldmfd sp!, {fp, ip, lr}*

## Contexte de fonction (5/5)

**ARM**

```
1
2 int une_fonction(int arg1, char arg2) {
3     int count;
4     char current;
5
6     count = arg1;
7     current = arg2;
8
9     ...
10
11     return count;
12 }
13
14 int main(int argc, char *argv[]) {
15     int ret;
16
17     ret = une_fonction(0x26, 'z');
18 }
```

```
3  une_fonction:
4      mov ip, sp
5      stmfd sp!, {fp, ip, lr}
6      sub fp, ip, #4 @ Début du stack frame
7
8      sub sp, sp, #8 @ Réserve pour count et current
9      str r0, [fp, #-12] @ count = arg1
10     strb r1, [fp, #-13] @ current = arg2
11
12     ...
13     ldr r0, [fp, #-12]
14
15     sub sp, fp, #8 @ Libère la zone des variables
16     @ locales et prépare la restauration
17     ldmfd sp!, {fp, ip, lr}
18     mov sp, ip
19     mov pc, lr
20
21 main:
22     mov ip, sp
23     stmfd sp!, {fp, ip, lr}
24     sub fp, ip, #4
25
26     sub sp, sp, #4
27
28     mov r0, #0x26
29     mov r1, #'z'
30     bl une_fonction
31     str r0, [fp, #-12] @ Valeur de retour
32
33     sub sp, fp, #8
34
35     ldmfd sp, {fp, sp, pc}
```

34

Cours ASM - Institut REDS/HEIG-VD

Le code assembleur *ARM* ci-dessus illustre – à quelques différences près – le code objet qui serait produit par un compilateur C à partir du code de gauche.

L'épilogue de la fonction *main* montre une alternative plus rapide.

## Conventions *ABI* (1/5)

- *Application Binary Interface (ABI)*
  - Interfaces bas niveau entre :
    - Applications et OS
    - Applications et bibliothèques
    - Différentes parties d'applications
- **Conventions des appels de fonction** dans le langage C
  - Une spécification de l'*ABI* dédiée aux appels de fonction
    - Passage des arguments
    - Stockage des variables locales
    - Retour de fonction
    - ...

L'*Application Binary Interface (ABI)* décrit une interface bas niveau entre les applications et le système d'exploitation, entre une application et une bibliothèque ou bien entre différentes parties d'une application.

Dans le contexte des appels de fonction, l'*ABI* décrit également comment les arguments doivent être passés (selon le processeur), comment les variables locales doivent être gérées, et d'une manière générale comment la pile est gérée.

## Conventions ABI (2/5)

- Conventions pour **x86**
  - **cdecl** (prononcé "see-DECK-'ll"), *fastcall*, *stdcall*, *thiscall*, *etc.*
  - **Pile descendante**
  - Passage des arguments sur la **pile**
  - Valeur de retour dans le registre **%eax**
  - Variables locales sur la pile
  - Registres **scratch** : **%eax**, **%ecx** et **%edx**
  - Alignement de la pile sur 16 octets

L'ABI pour une architecture *x86* et spécifiée ci-dessus. On notera l'utilisation des registres *%eax*, *%ecx* et *%edx* en tant que *scratch register*, c-à-d des registres de travail que le compilateur peut utiliser librement à l'intérieur d'une fonction.

## Conventions ABI (3/5)

- Conventions pour **ARM**
  - **AAPCS** (*ARM Architecture Procedure Call Standard*)
    - **GNU-EABI** (appropriés pour les systèmes embarqués)
- **Pile descendante**
- Passage des arguments par les **registres** (*r0, r1, r2, r3*)
- Valeur de retour dans le registre **r0**
- Variables locales dans les **registres** (*r4-r11*) ainsi que sur la **pile**
- Alignement sur 4 octets dans tous les cas

37

Cours ASM - Institut REDS/HEIG-VD

L'ABI pour une architecture ARM est spécifié ci-dessous (correspond à GNU-EABI). On remarque la présence de quatre registres (*r0-r3*) pour le passage d'arguments à une fonction. Si toutefois davantage d'arguments sont requis, les arguments suivants (à partir du cinquième), la pile sera sollicitée.

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

## Conventions *ABI* (4/5)



**ARM**

- Ecrire un programme en assembleur *ARM* qui implémente la fonction `void fill(char *buffer, int size)` qui remplit *buffer* avec *size* caractères 'X' .
- Implémentez un code qui appelle cette fonction à titre d'exemple.
- La fonction doit suivre les conventions de l'*ABI ARM*.

## Conventions ABI (5/5)



**x86**

- Ecrire un programme en assembleur *x86* qui implémente la fonction `void fill(char *buffer, int size)` qui remplit *buffer* avec *size* caractères 'X' .
- Implémentez un code qui appelle cette fonction à titre d'exemple.
- La fonction doit suivre les conventions de l'ABI *x86*.

## Références

- ARM Infocenter. Site ARM principal : <http://infocenter.arm.com>
- Intel x86 32 & 64 bits.  
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
- Rajat Moona, *Assembly Language Programming in GNU/Linux for IA32 Architectures*, Eastern Economy Edition, New Dehli, 2009.
- Andrew N.Sloss, Dominic Symes, *ARM System Developer's Guide, Designing and Optimizing System Software*, Elsevier, 2004.