Langage d'assemblage

Vincent Migliore

vincent.migliore@insa-toulouse.fr

Organisation du cours

- ASM (responsable Vincent Migliore) :
  - 7 CM ;
  - 4 TD.
- ArchiMat (responsable Vincent Migliore) :
  - Archi ARM/x86 (enseignant Vincent Migliore) :
    - 2 CM ;
    - 3 TP.
  - Archi Programmation Orientée Matérielle (enseignant Arthur Bit-Monnot) :
    - 2 CM ;
    - 3 TP.
  - Archi Sécurité (enseignant Vincent Migliore) :
    - 2 CM ;
    - 2 TP.

Introduction au langage d'assemblage

Représentation interne des données et des instructions [1]

Les données et instructions dans un ordinateur sont représentées sous forme binaire (0 ou 1). Les premiers ordinateurs possédaient d'ailleurs une interface utilisateur binaire.

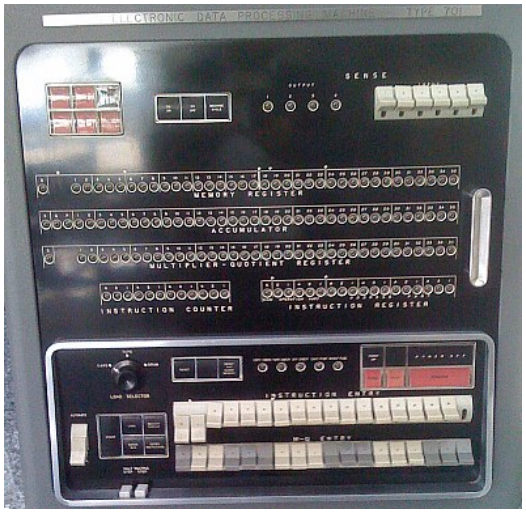


Figure: IBM 701 (1952)

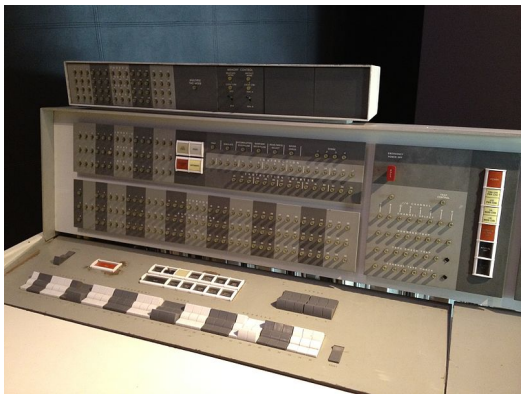


Figure: IBM 7094 (1962)

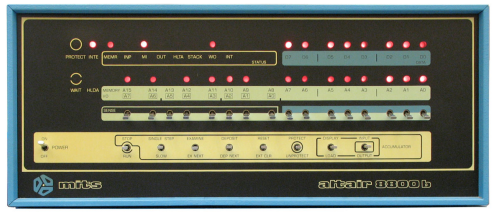


Figure: Altaïr 8800 (1975)

Représentation interne des données et des instructions [2]

Depuis l'avènement des mots de données multiples de 8 bits (systèmes UNIX), les représentations octales et hexadécimales se sont démocratisées.

Octal

Regroupement de 3 bits :

10	101	010
2	5	2

Hexadécimal

Regroupement de 4 bits :

1010	0101	1111
A	5	F

Langage machine

C'est le seul langage compréhensible par la machine. Ce langage est constitué d'instructions dont la taille est un multiple de 8 bits (1 octet). L'ensemble des instructions exécutables par une machine s'appelle le jeu d'instructions. La représentation binaire des instructions peut varier d'une machine à l'autre, tout comme le jeu d'instructions.

```
f3 0f 1e fa
55
48 89 e5
41 57
41 56
41 54
53
```

Figure: Exemple de langage machine, 1 ligne par instruction

Langage d'assemblage

Afin de simplifier l'écriture de programmes, une représentation textuelle du code a été développée (langage d'assemblage). Elle est constituée :

- De symboles fixes pour les opérations : mnémoniques ou opcodes ;
- De symboles fixes pour les registres et modes d'adressage ;
- De symboles créés par l'utilisateur pour les adresses : Labels, et pour les constantes.

```
5431: f3 0f 1e fa          endbr64
5435: 55                   push   %rbp
5436: 48 89 e5             mov    %rsp,%rbp
5439: 41 57                push   %r15
543b: 41 56                push   %r14
543d: 41 54                push   %r12
543f: 53                   push   %rbx
```

Figure: Exemple de langage machine (1 ligne par instruction) et sa représentation en langage d'assemblage

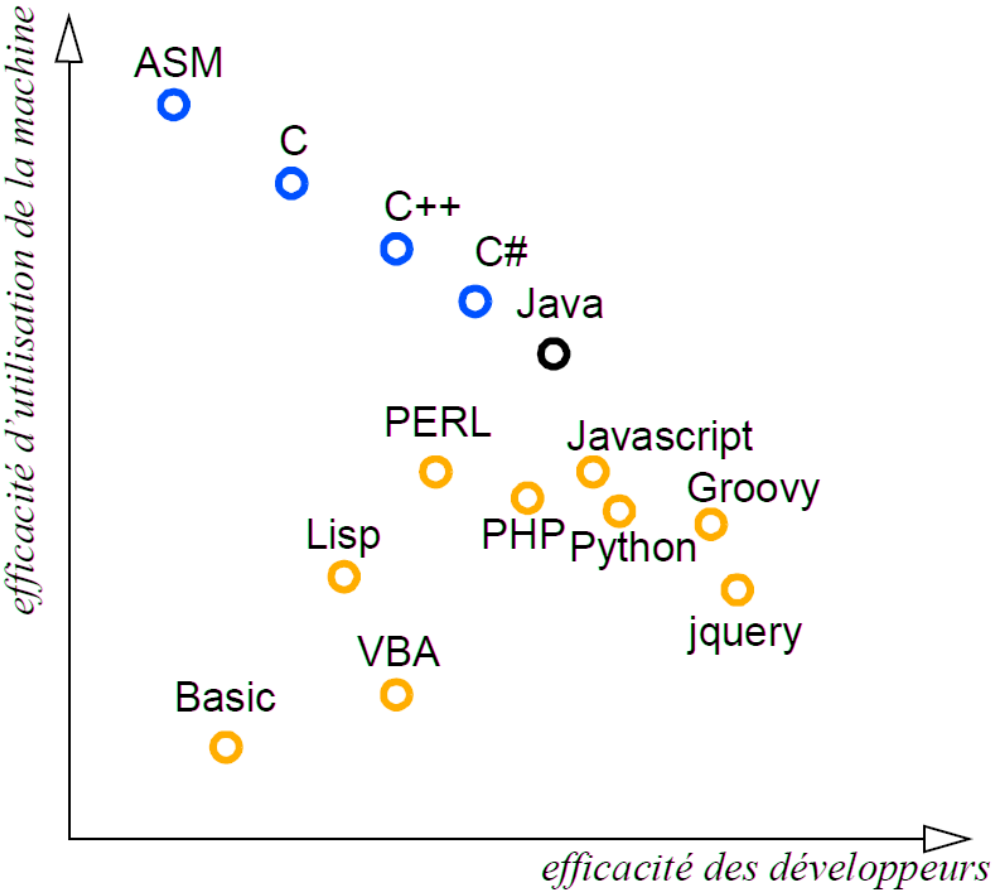
Mauvaise réputation de l'ASM

- Rébarbatif, hermétique, hostile ;
- Verbeux mais opaque ;
- Incitant à de mauvaises pratiques (programmation non structurée) ;
- Non-portable, un dialecte différent pour chaque architecture.

Pourquoi l'étudier ?

En se restreignant aux langages de haut niveau uniquement, vous ne pourrez pas expliquer certains comportements observés sur la machine (bugs, durée d'exécution excessive, ...) car les technologies modernes cachent les aspects bas niveau.
De plus, les technologies modernes reposant sur le langage d'assemblage, vous pourrez mieux comprendre et mieux exploiter ces outils.

Comparaison des langages



Version simplifiée car chaque axe est en principe évalué sur plusieurs dimensions :

- Efficacité d'utilisation de la machine : temps d'execution + utilisation mémoire ;
- Efficacité du développeur : vitesse de développement + réutilisation de code.

Dans la pratique, les langages sont combinés pour optimiser le compromis efficacité sur la machine et temps de développement.

Combinaison des langages

Cas d'un site web


```
1
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-str
3  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" lang="fr"><head><meta http-equiv="Content-T
4
5                  $j = jQuery.noConflict();
6
7  --></script><script src="/plugins/web/resources/js/jquery.cookie.js" type="text/javascript"></script
8
9                  $j(document).ready(function() {
10                     $j().piroBox_ext({
11                         piro_speed: 600,
12                         bg_alpha: 0.5,
13                         piro_scroll: true,
14                         prev_label: 'Précédent',
15                         next_label: 'Suivant',
16                         close_label: 'Fermer'
17                     });
18                 });
19
20 --></script><script type="text/javascript"><!--
21
22             ametysFOUser = null;
23
24
25 --></script><script type="text/javascript"><!--
26
27             ametysCookieConsentListener = [];
```

- La mise en page est pilotée par un programme en javascript,
- L'interpréteur javascript a été créé en C++,
- La fonction de décodage JPEG a été créée en C,
- La fonction de manipulation de pixels sur la mémoire d'écran a été créée en langage d'assemblage.

Desktop/Laptop (PC), tablette, smartphone

- Noyau du système d'exploitation : commutation de tâches par manipulation de pile, traitement des interruptions
- drivers de périphériques
- calcul intensif : multimedia (codecs, traitement de son et d'image)
- graphique 2D/3D (GPU)
- interpréteur de java-bytecode
- émulateur

Systèmes embarqués (au sens large)

- traitement du signal (DSP Digital Signal Processing) ;
- temps réel (OS et applications) ;
- microcontrôleur de petite capacité (8 bits).

Domaine d'application [2]

L'ASM reste fondamental pour :

- La création, le portage et l'amélioration des compilateurs des autres langages (même si le compilateur lui-même n'est pas développé en ASM)
- L'interpretation des sessions de debug
- L'analyse de programmes compilés en vue de les optimiser (désassemblage)
- La compréhension des incidents causés par l'excés d'optimisation des compilateurs
- Le reverse-engineering, la gestion de la sécurité

Lorsque l'objectif est la performance machine et qu'on dispose d'un compilateur C avec de bonnes capacités d'optimisation, la solution ASM ne sera retenue que si elle surclasse la solution C optimisée.

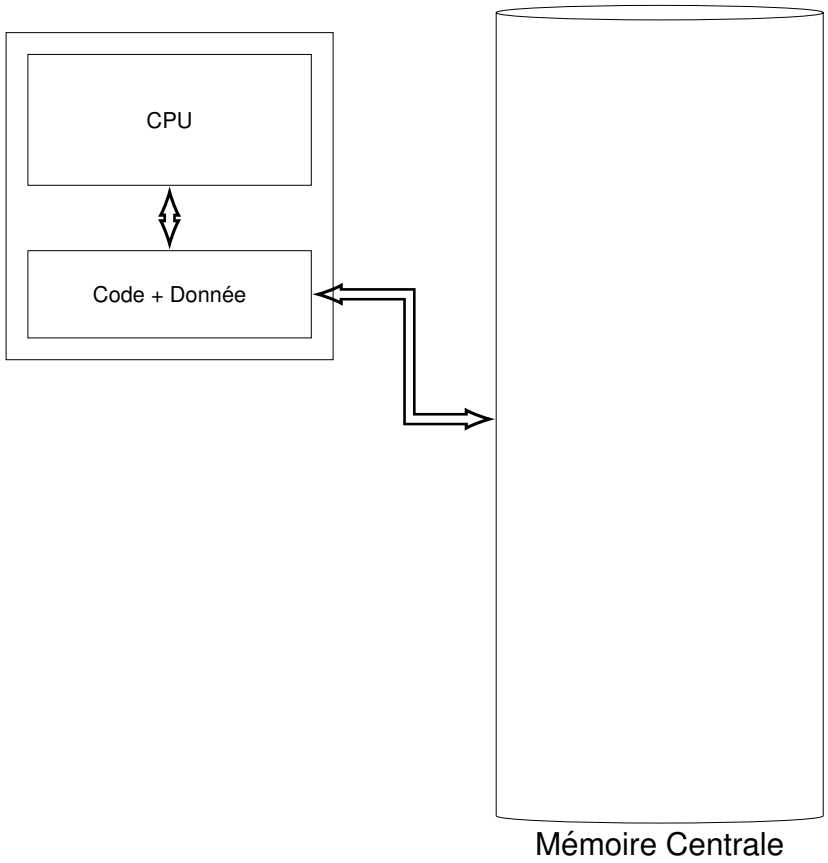
Cas où le compilateur atteint certaines limites:

- Utilisation d'instructions spécialisées non gérées par le compilateur :
  - Contrôle du processeur : reset, sleep (pause), watchdog, gestion de privilège, interruption soft, etc...
  - arithmétique saturée (bornage d'un résultat),
  - opérations arithmétiques hétérogènes (par exemple multiplication 32 * 32 $\rightarrow$ 64 bits),
  - division avec reste,
  - multiplication-accumulation (DSP Digital Signal Processing),
  - SIMD (Single Instruction Multiple Data), opérations vectorielles
- Hypothèse sur les valeurs;
- Relâchement des contraintes sur la sauvegarde du contexte.

Elements d'architecture à considérer

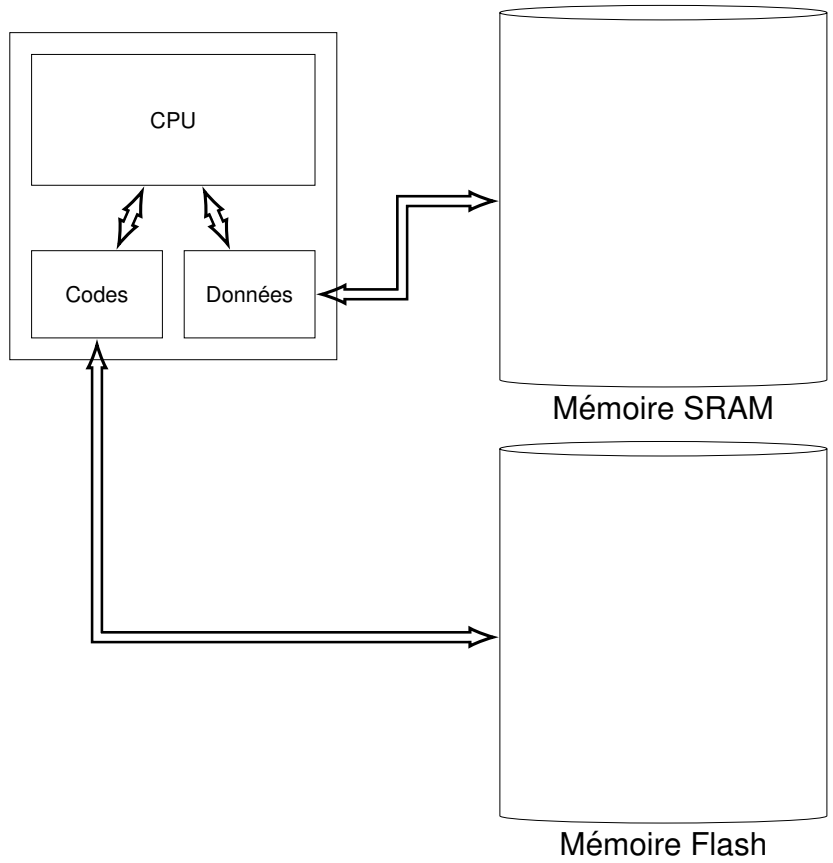
Rappels sur les architectures

Von Neumann



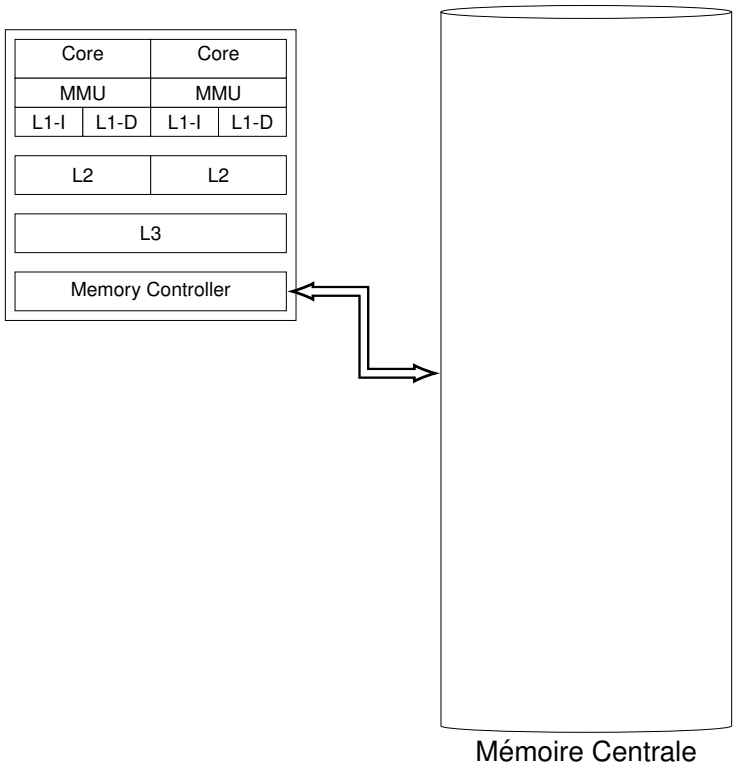
Espace mémoire linéaire Code+Données.
Souplesse et facilité d'utilisation.

Harvard



Deux espaces mémoires linéaires indépendants.
Plus performant car accès concurrent.

Harvard Modifiée



Espace mémoire linéaire Code+Données, mais séparation des chemins via des caches. Souplesse et facilité d'utilisation + amélioration des performances.

Jeu d'instructions

C'est l'ensemble des instructions supportées par un processeur donné.

Architecture de jeu d'instructions

Appelé Instruction Set Architecture (ISA) en anglais, c'est la description fonctionnelle du processeur du point de vu du programmeur, c'est-à-dire tout ce qui est visible au niveau du langage machine.
Ce qui est **visible** au niveau de l'ISA :

- Les instructions supportées par le processeur;
- Les registres utilisables par les instructions et leur rôle ;
- L'organisation de la mémoire et des entrées/sorties ;
- Les éléments architecturaux (présence ou non de plusieurs coeurs, présence ou non de caches,$\cdots$).

(Exemple) de ce qui est **invisible** au niveau de l'ISA :

- Les éléments internes du pipeline processeurs (nombre d'étages, exécution in-order ou out-of-order, prédiction de branchement,$\cdots$) ;
- Le nombre de cache et leur associativité ;
- Description interne du circuit du processeur (ALU, contrôleur mémoire, unité de contrôle,$\cdots$).

RISC

Reduced **I**nstruction **S**et **C**omputer

- Premier jeu d'instruction formalisé (Université de Stanford et Berkeley, 1981).
- code d'instructions de longueur fixe = 1 mot de mémoire programme, pour faciliter l'utilisation d'un pipe-line
- durée d'exécution d'une instruction = 1 cycle d'horloge registres banalisés
- éventuellement architecture load-store (2 instructions seulement ont accès à la mémoire)

CISC

Complex **I**nstruction **S**et **C**omputer

- Concept non formalisé : Acronyme donné aux familles non RISC.
- code d'instructions de longueur variable, partant de 1 byte (pas d'alignement mémoire).
- développement incrémental d'une famille de CPUs de plus en plus perfectionnés.
- registres spécialisés.
- ajout d'instructions facilité par le microcodage.
- durée d'exécution d'une instruction très variable.

RISC

Reduced **I**nstruction **S**et **C**omputer

- Premier jeu d'instruction formalisé (Université de Stanford et Berkeley, 1981).
- code d'instructions de longueur fixe = 1 mot de mémoire programme, pour faciliter l'utilisation d'un pipe-line
- durée d'exécution d'une instruction = 1 cycle d'horloge registres banalisés
- éventuellement architecture load-store (2 instructions seulement ont accès à la mémoire)

CISC

Complex **I**nstruction **S**et **C**omputer

- Concept non formalisé : Acronyme donné aux familles non RISC.
- code d'instructions de longueur variable, partant de 1 byte (pas d'alignement mémoire).
- développement incrémental d'une famille de CPUs de plus en plus perfectionnés.
- registres spécialisés.
- ajout d'instructions facilité par le microcodage.
- durée d'exécution d'une instruction très variable.

Aujourd'hui la barriere entre les deux jeux d'instruction s'est réduite :

- les processeurs CISC se sont dotés de pipelines efficaces ;
- le jeu d'instruction des RISC est devenu de plus en plus complexe, les règles ont été assouplies (Arm autorise des instructions sur 16 et 32 bits + spécialisations de registres) ;
- vers un dépassement des performances du RISC par rapport à CISC (Apple M1).

Jusqu'à ARMv7

Jusqu'à ARMv7, seulement des architectures jusqu'à 32 bits étaient supportés, avec les jeux d'instructions suivants :

ARM	:	Jeu d'instructions de taille fixe (32 bits). C'est le jeu d'instructions historique de Arm.
Thumb	:	Jeu d'instructions de taille fixe (16 bits). Il est constitué d'un sous-ensemble d'instructions du ARM (réduction de la taille du code, mais réduction des performances)
Thumb-2	:	Jeu d'instructions de taille variable (16 et 32 bits). Il s'agit d'une extension du jeu d'instruction Thumb avec des instructions 32 bits (issus de ARM) afin d'obtenir un compromois temps/taille du code.

ARMv8 et + (à partir d'octobre 2011)

Arrivée des architectures 64 bits, avec refonte de la nomenclature : Processeur 32 bits $\rightarrow$ AArch32, Processeur 64 bits $\rightarrow$ AArch64. Chaque architecture ne supporte pas le même jeu d'instruction :

AArch32		
A32 ("ARM")	:	Jeu d'instructions de taille fixe (32 bits) avec support de ARM
T32 ("THUMB")	:	Jeu d'instructions de taille variable (16 et 32 bits) avec support du Thumb-2

AArch64		
A64	:	Jeu d'instructions de taille fixe (32 bits)

Jeu d'instructions pour architectures 32 bits (x86)

Le jeu d'instruction des processeurs Intel d'architecture 32 bits est le x86. Rappel : Il s'agit d'un jeu d'instruction incrémental avec un jeu d'instructions historiques pour les processeurs 8086/8088, puis des extensions spécifiques pour les processeurs suivants.

Jeu d'instructions pour architectures 64 bits (x86-64, I64)

La société AMD proposa une extension pour processeurs 64 bits du jeu d'instruction x86 (nommé x86-64). Intel l'a renommé Intel 64 depuis (I64).

Une Architecture et un jeu d'instructions totalement configurables

L'université de Berkeley (oui encore eux) a récemment proposé un modèle d'architecture de jeu d'instruction libre et fortement configurable : C'est le RISC-V. L'objectif était de proposer un standard ouvert pour la recherche mais son aspect ouvert intéresse de plus en plus l'industrie. La taille des instructions est fixe (32 bits) est possède un modèle 32 bits et 64 bits.

Le RISC-V est composé d'un ensemble d'instructions de base sur les entiers noté I (opérations arithmétiques et logiques sur les entiers + accès à la mémoire) puis en fonction des opérations souhaitées, possède de nombreuses extensions qui viennent enrichir le jeu d'instruction en gardant toujours un même format d'instructions :

- M : Support de la multiplication entière ;
- A : Support des opérations atomiques (c'est-à-dire non-interrompables) qui permettent typiquement la synchronisation entre plusieurs coeurs RISC-V ;
- F : Support des opérations à virgule flottante simple précision ;
- D : Support des opérations à virgule flottante double précision ;
- C : Support des instructions compressées (c'est-à-dire sur 16 bits).

En particulier, un RISC-V 64 bits ayant les modules MFC sera noté RV64ICMF.

De nombreuses extensions restent encore à décrire, notamment la gestion du calcul vectoriel, binaire ou encore les opérations cryptographiques.

RISC : exemple de ARM

Registres généraux :

- Low Registers : R0 $\rightarrow$ R7
- High Registers : R8 $\rightarrow$ R12

Registres spéciaux :

- R13 Stack Pointer (SP) : Pointeur de pile
- R14 Link Register (LR) : Adresse de retour de fonction
- R15 Program Counter (PC) : Compteur Ordinal (adresse de l'instruction)
- Program Status Register (xPSR)

CISC : exemple de INTEL

Registres généraux :

- EAX, EAB, ECX, EDX

Registres de segment :

- CS : Code Segment
- DS : Data Segment
- SS : Stack Segment

Registres d'indices et de pointeur :

- ESP : Extended Stack Pointer (pointeur vers le haut de la pile)
- EIP : Extended Instruction Pointer (adresse de l'instruction)

Registres de controle :

- CR0 $\rightarrow$ CR4
- CR0 : Flags controllant les opérations de base du processeur (protection en ecriture, activation des caches, activation de la pagination, ...)
- CR3 : Adresse de la table des pages.

Registre de controle : Les flags

Etat du calcul	ARM Cortex-M3	Intel x86 (i32)	PIC24
Résultat Zéro	Z	ZF	Z
Retenue addition ou soustraction (Carry)	C	CF	C
Signe négatif	N	SF	N
Débordement signé (Overflow)	V	OF	OV
Saturation (écrêtage)	Q	-	-

- N (Negative) : Activé si le résultat est négatif (bit de poids fort à 1)
- Z (Zero) : Activé si le résultat est égal à zéro.
- C (Carry) : Activé 1 lorsqu'une opération **non signée** déborde (dépasse la valeur maximale représentable par le registre).
- V (Signed Overflow) : Activé lorsqu'une opération **signée** déborde (dépasse la valeur maximale représentable par le registre).

Remarque : En principe le fait que le flag soit actif revient à mettre le bit associé au flag à 1, mais il y a des exceptions.
Remarque 2 : Les opérations signées se font toujours en complément à 2.

Les code-conditions

Les codes-conditions sont des codes composés de quelques lettres permettant de modifier le comportement de certaines opérations en fonction des flags du registre de contrôle. On va le retrouver typiquement dans opérations de branchement (saut pour Intel), permettant de faire partir l'exécution d'un programme à une nouvelle adresse.

Exemple avec Arm

Code	**Meaning (for cmp or subs)**	**Flags Tested**
eq	Equal.	Z==1
ne Not equal.	Z==0	
cs or hs	Unsigned higher or same (or carry set).	C==1
cc or lo	Unsigned lower (or carry clear).	C==0
mi	Negative. The mnemonic stands for "minus".	N==1
pl	Positive or zero. The mnemonic stands for "plus".	N==0
vs	Signed overflow. The mnemonic stands for "V set".	V==1
vc	No signed overflow. The mnemonic stands for "V clear".	V==0
hi	Unsigned higher.	(C==1) && (Z==0)
ls	Unsigned lower or same.	(C==0) —— (Z==1)
ge	Signed greater than or equal.	N==V
lt	Signed less than.	N!=V
gt	Signed greater than.	(Z==0) && (N==V)
le	Signed less than or equal.	(Z==1) —— (N!=V)
al (or omitted)	Always executed.	None tested.

Exemple d'instructions modifiant les flags (Arm)

Pour Arm, les opcodes avec le suffixe "s" mettent à jour le registre d'état, de manière cohérente par rapport à l'opération. Quelques exemples :

- **adds** *dest, op1, op2* : Opération d'addition entière.
- **subs** *dest, op1, op2* : Opération de soustraction entière.
- **muls** *dest, op1, op2* : Opération de multiplication entière.
- **adcs** *dest, op1, op2* : Opération d'addition entière avec prise en compte de la retenue (flag C).
- **sdcs** *dest, op1, op2* : Opération de soustraction entière avec prise en compte de la retenue (flag C).
- **cmp** *op1, op2* : Comparaison de deux entiers. Cela revient à évaluer *op2* - *op1*.

Exemple d'instructions utilisant les flags (Arm)

- **Opérations Arithmétiques** :
  - **adc** *dest, op1, op2* (add with carry) : Opération d'addition entière avec prise en compte de la retenue (flag C) sans mise à jour du registe de controle.
  - **sbc** *dest, op1, op2* (sub with carry) : Opération de soustraction entière avec prise en compte de la retenue (flag C) sans mise à jour du registe de controle.
- **Opérations de gestion du flux de controle :**
  - **beq** (branch if equal) : Saut du code à une adresse donnée si le flag Z (zero) est à 1.
  - **bne** (branch if not equal) : Saut du code à une adresse donnée si le flag Z (zero) est à 0.
  - **ble** (branch if less or equal) : Saut du code à une adresse si Z = 1 et N !=V.
  - **bcs** (branch if carry set) : Saut du code à une adresse donnée si le flag C (carry) est à 1.
  - **bcc** (branch if carry clear) : Saut du code à une adresse donnée si le flag C (carry) est à 0.
  - **bvs** (branch if oVerflow set) : Saut du code à une adresse donnée si le flag V (overflow) est à 1.

Cas de Intel x86

- **Opérations Arithmétiques** :
  - **adc** (add with carry) : Addition avec prise en compte de la carry.
  - **sbc** (sub with carry) : Soustraction avec prise en compte de la carry
- **Opérations de gestion du flux de controle** :
  - **cmp** *op1*, *op2* (compare) : Comparaison de deux entiers. Cela revient à évaluer *op2* - *op1*.
  - **jz** (jump if zero) : Saut du code à une adresse donnée si le résultat de l'instruction précédente est nulle (Flag Z à 1).
  - **jc** (jump if carry) : Saut du code à une adresse donnée si le résultat de l'instruction précédente a provoqué une carry (Flag C à 1).
  - **jo** (jump if overflow) : Saut du code à une adresse donnée si le résultat de l'instruction précédente a provoqué un débordement (Flag V à 1).
  - **jnz** (jump if zero) : Saut du code à une adresse donnée si le résultat de l'instruction précédente est non nulle (Flag Z à 0).
  - **jnc** (jump if carry) : Saut du code à une adresse donnée si le résultat de l'instruction précédente n'a pas provoqué une carry (Flag C à 0).
  - **jno** (jump if overflow) : Saut du code à une adresse donnée si le résultat de l'instruction précédente n'a pas provoqué un débordement (Flag V à 0).

Gestion de la mémoire

Gestion mémoire : le stockage de l'information

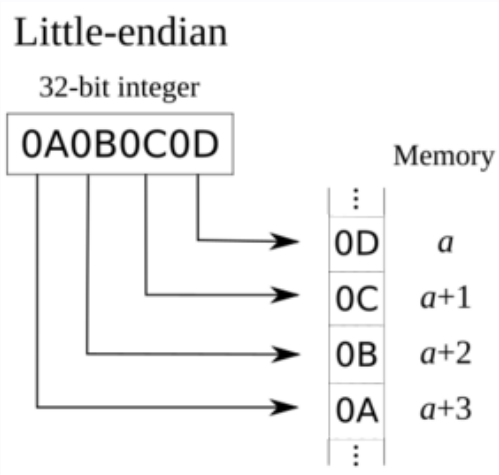
La montée en puissance des machines a fait progresser la taille des mots et des bus de 8 bits à 64 bits aujourd'hui.

Le découpage en octet (8 bits) des données est resté la norme même sur les machines plus récentes pour faciliter l'interopérabilité des systèmes.

C'est ainsi que les machines Von-Neumann, Harvard ou Harvard Modifiée ont un adressage à l'octet.

Pour des données sur 2 octets ou plus, il existe deux sens possibles pour le stockage des données : le format little-endian et big-endian. Elles imposent un contrôleur mémoire du processeur particulier et donc ne peut pas être changé après la conception du processeur.

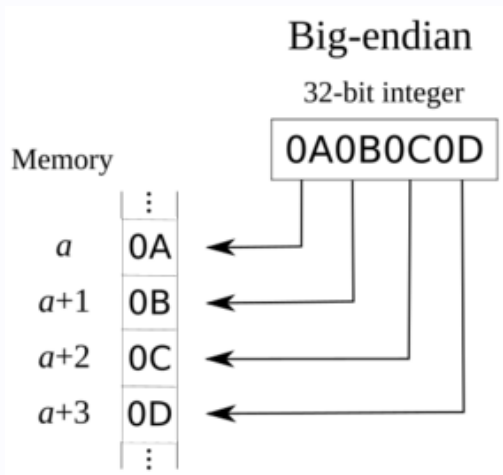
Little-endian



L'adresse mémoire de la donnée pointer vers l'octet de poids **faible**.

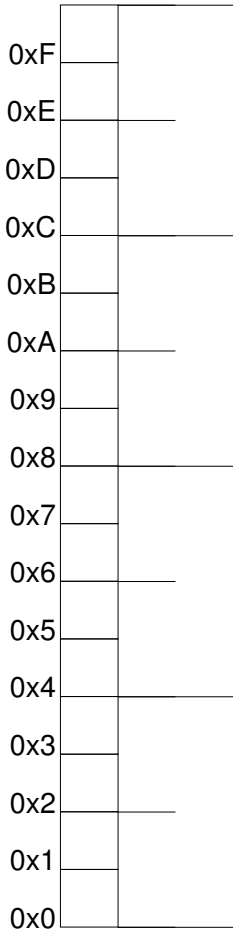
- Intel (du 8080 au Pentium)
- Microchip PIC, ATmega328
- Cortex M3 (ARM)

Big-endian

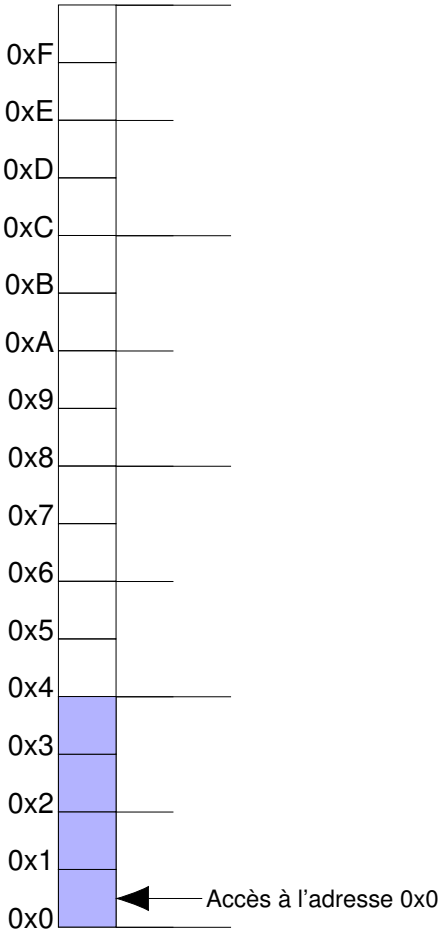


L'adresse mémoire de la donnée pointer vers l'octet de poids **fort**.

- Sun Sparc CPU (avant la version 9)
- IBM-Motorola PowerPC
- headers des paquets Ethernet, IP, UDP, TCP : "network order".

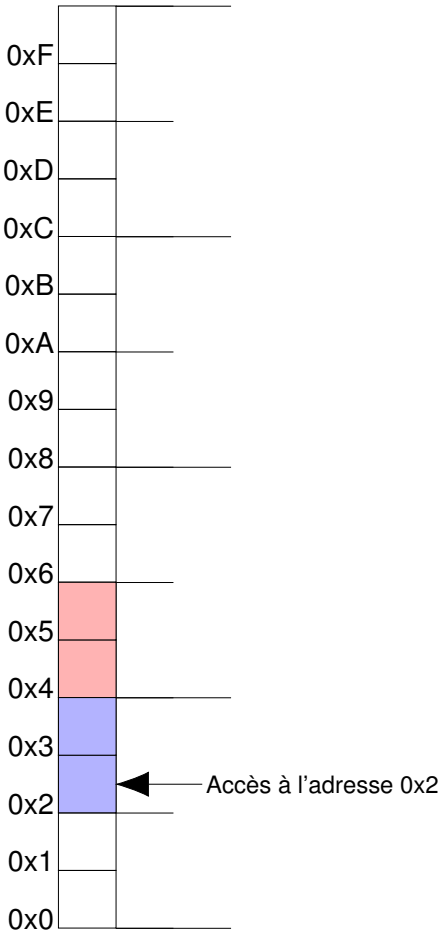


Le nombre d'octets que peut lire un processeur par opération est définit par la taille du bus mémoire.
Si le bus mémoire est sur 32 bits, le processeur pourra lire 4 octets d'un coup si l'adresse est un multiple de 4.



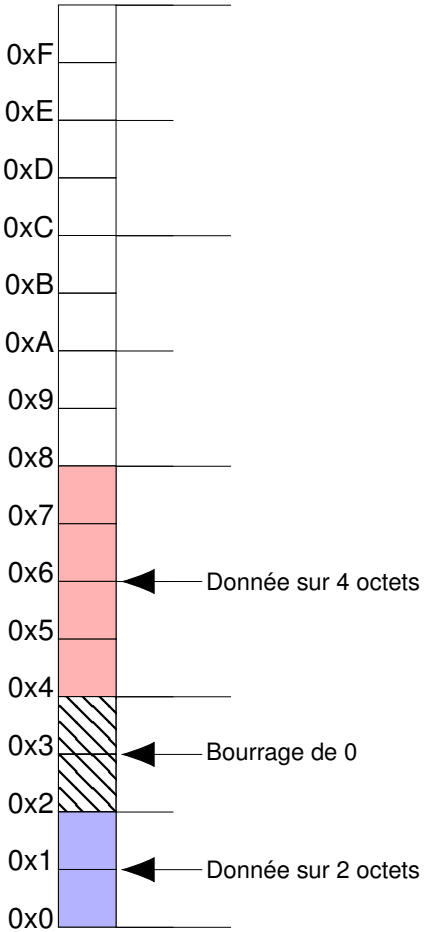
Le nombre d'octets que peut lire un processeur par opération est définit par la taille du bus mémoire.
Si le bus mémoire est sur 32 bits, le processeur pourra lire 4 octets d'un coup si l'adresse est un multiple de 4.

Ici, le processeur demande 4 octets à l'adresse 0x0. L'accès est dit aligné et se fera en un coup.



Le nombre d'octets que peut lire un processeur par opération est définit par la taille du bus mémoire.
Si le bus mémoire est sur 32 bits, le processeur pourra lire 4 octets d'un coup si l'adresse est un multiple de 4.

Dans ce second exemple, si le processeur demande 4 octets, il y aura un chevauchement entre deux données de 4 octets alignées sur le bus mémoire, et donc 2 accès seront nécessaires pour récupérer la donnée.



Le nombre d'octets que peut lire un processeur par opération est définit par la taille du bus mémoire.
Si le bus mémoire est sur 32 bits, le processeur pourra lire 4 octets d'un coup si l'adresse est un multiple de 4.

En cas de stockage de données hétérogènes, il est commun d'ajouter des espaces inutilisés (en pratique mis à 0) pour que les données soient toutes alignées.

Les modes d'adressage

Définition : Opérande

Mathématiquement, un opérande est un élément sur lequel s'applique une opération (Exemple : 1 et 2 sont les deux opérandes de l'opération 1+2 = 3). En informatique, l'opérande fait plus précisément référence aux arguments d'une fonction. En langage d'assemblage, un opérande fait référence soit à un registre, soit une adresse mémoire.

Définition : Code opération (opcode)

Le code opération (ou opcode en anglais) est la partie de l'instruction qui définie l'opération à réaliser.

Définition : Instruction

L'instruction est une étape d'un programme informatique. Il est constitué d'un code opération et éventuellement d'opérandes.

Définition : Modes d'adressage

Les modes d'adressages définissent comment l'instruction en langage machine identifie les opérandes.

Figure: Exemple d'un format possible pour une instruction

Adressage immédiat ou littéral

Il n'y a pas à proprement parler d'adressage, la donnée étant directement codée dans l'instruction.

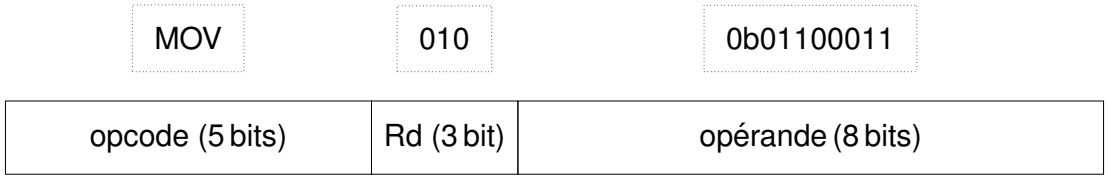


Figure: Exemple d'adressage immédiat avec l'instruction mov de l'ARMv7m (instruction sur 16 bits)

Dans cet exemple :

- L'opération à exécuter est un déplacement de valeur (MOV).
- L'opérande est une constante codée dans l'instruction (adressage immédiat).
- La destination est un registre (Rd = R2).

En langage d'assemblage ARM, cette instruction s'écrit :

```
mov R2, #0b01100011
```

Adressage immédiat ou littéral

Il n'y a pas à proprement parler d'adressage, la donnée étant directement codée dans l'instruction.

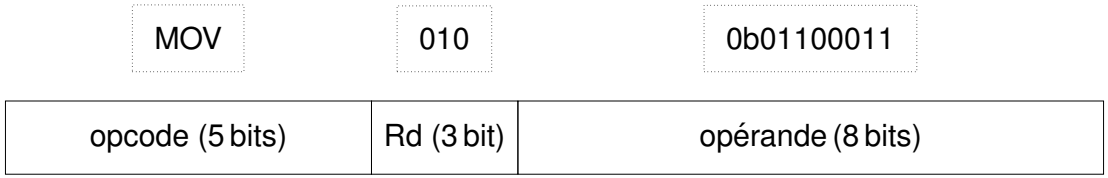


Figure: Exemple d'adressage immédiat avec l'instruction mov de l'ARMv7m (instruction sur 16 bits)

Question

Selon la documentation, comme le registre de destination est sur 32 bits, l'opération suivante est réalisée : imm32 = ZeroExtend(imm8, 32). En fonction des contraintes précisées ci-dessus, donnez :

- **La plage des registres utilisables.**
- **La plage des valeurs que l'on peut affecter au registre de destination.**
- **Comment se gère le stockage d'un nombre négatif.**

Adressage immédiat ou littéral

Il n'y a pas à proprement parler d'adressage, la donnée étant directement codée dans l'instruction.

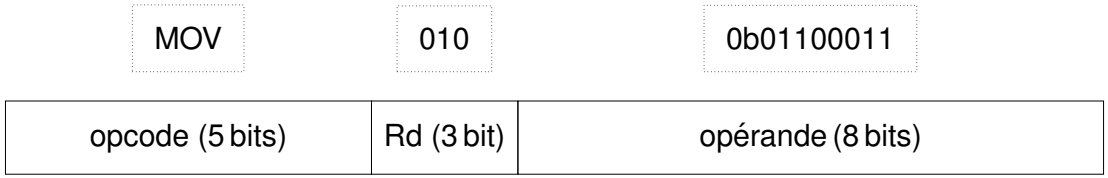


Figure: Exemple d'adressage immédiat avec l'instruction mov de l'ARMv7m (instruction sur 16 bits)

Réponse

imm32 = ZeroExtend(imm8, 32).

- **La plage des registres utilisables :** Rd étant codé sur 3 bits, le numéro de 8 registres peuvent être codés → R0 à R7.
- **La plage des valeurs que l'on peut affecter au registre de destination :** l'opérande étant codée sur 8 bits, la valeur maximale est 255 → 0 à 255.
- **Comment se gère le stockage d'un nombre négatif :** L'extension de la valeur immédiate de 8 bits en 32 bits se faisant en ajoutant des 0 à gauche, le codage de nombres négatifs est impossible.

Adressage immédiat ou littéral

Il n'y a pas à proprement parler d'adressage, la donnée étant directement codée dans l'instruction.

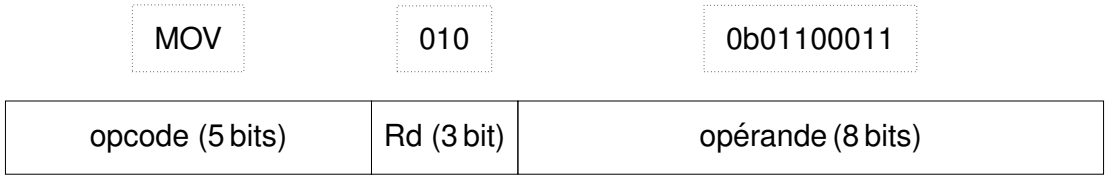


Figure: Exemple d'adressage immédiat avec l'instruction mov de l'ARMv7m (instruction sur 16 bits)

Remarque

Il existe une variante de mov sur 32 bits pour l'architecture ARMv7m pour étendre à la fois la plage des registres adressables (4 bits) et également la valeur immédiate (11 bits) :

mov.w Rd, #imm11

Il existe un grand nombre de variantes qui sont documentées à la page 291 du manuel de l'ARMv7m (notamment une variante permettant le codage de nombres négatifs).

Adressage direct

Dans ce mode d'adressage, l'adresse mémoire de la donnée est codée dans l'instruction. Remarque : Ce mode d'adressage n'est pas supporté par ARM.

Adressage indirect par registre

Dans ce mode d'adressage, l'adresse mémoire de la donnée est déterminée à partir de la valeur d'un registre (appelé registre de base). Il est également possible de coder dans l'instruction of décalage (offset) qui sera ajouté à la valeur du registre (on parle d'adressage indirect par registre avec offset).

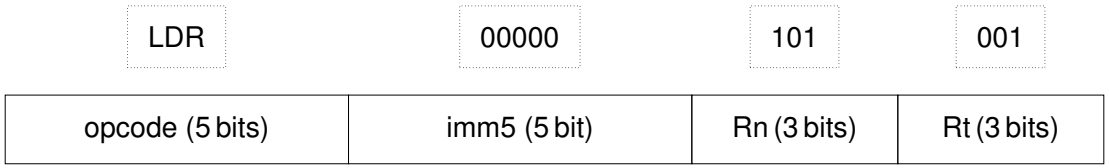


Figure: Exemple d'adressage indirect par registre avec offset avec l'instruction ldr de l'ARMv7m (instruction sur 16 bits)

Dans cet exemple :

- L'opération à exécuter est une lecture en mémoire (LDR).
- L'opérande est un registre qui contient l'adresse à lire (Rn = R5).
- La destination (target) est un registre (Rt = R1).
- Il n'y a pas d'offset, donc il s'agit d'un adressage indirect par registre.

En langage d'assemblage ARM, cette instruction s'écrit :

```
ldr R1, [R5]
```

Adressage indirect par registre

Dans ce mode d'adressage, l'adresse mémoire de la donnée est déterminée à partir de la valeur d'un registre (appelé registre de base). Il est également possible de coder dans l'instruction of décalage (offset) qui sera ajouté à la valeur du registre (on parle d'adressage indirect par registre avec offset).

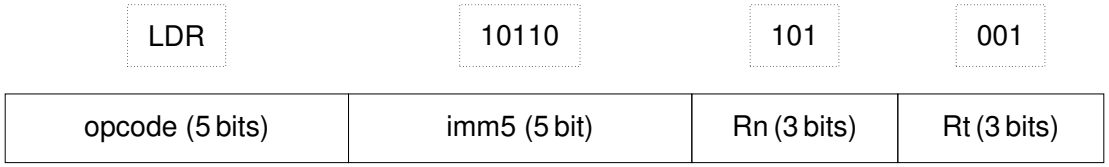


Figure: Exemple d'adressage indirect par registre avec offset avec l'instruction ldr de l'ARMv7m (instruction sur 16 bits)

Dans cet exemple :

- L'opération à exécuter est une lecture en mémoire (LDR).
- L'opérande est un registre qui contient l'adresse à lire (Rn = R5).
- La destination (target) est un registre (Rt = R1).
- Il y a un offset de 22, donc il s'agit d'un adressage indirect par registre avec offset, et l'adresse finale sera le contenu de R5 + 22.

En langage d'assemblage ARM, cette instruction s'écrit :

```
ldr R1, [R5,#22]
```

Adressage indirect par registre

Dans ce mode d'adressage, l'adresse mémoire de la donnée est déterminée à partir de la valeur d'un registre (appelé registre de base). Il est également possible de coder dans l'instruction of décalage (offset) qui sera ajouté à la valeur du registre (on parle d'adressage indirect par registre avec offset).

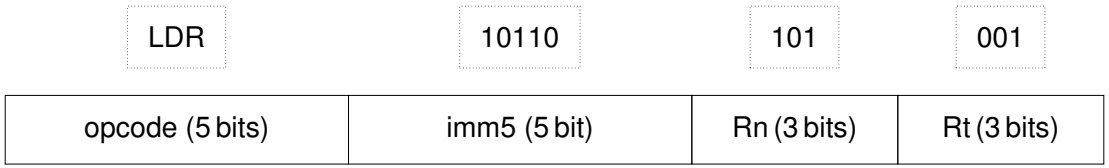


Figure: Exemple d'adressage indirect par registre avec offset avec l'instruction ldr de l'ARMv7m (instruction sur 16 bits)

Remarque

L'adressage indirect par registre avec offset est la manière la plus simple d'accéder à un élément particulier d'une structure de données.

Adressage indirect par registre

Dans ce mode d'adressage, l'adresse mémoire de la donnée est déterminée à partir de la valeur d'un registre (appelé registre de base). Il est également possible de coder dans l'instruction of décalage (offset) qui sera ajouté à la valeur du registre (on parle d'adressage indirect par registre avec offset).

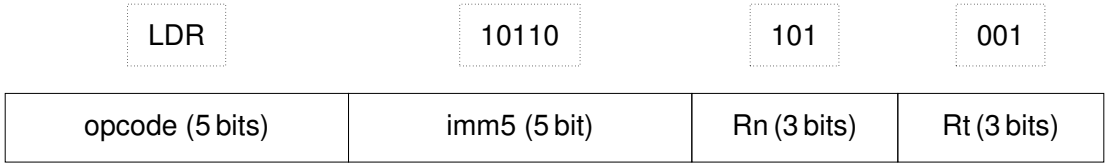


Figure: Exemple d'adressage indirect par registre avec offset avec l'instruction ldr de l'ARMv7m (instruction sur 16 bits)

Question

On considère la structure ci-dessous :

```
struct personne{
uint32_t id;
uint32_t age;
uint32_t taille;
}
```

En supposant que l'adresse de début de la structure personne soit stockée dans R5 et que la destination soit le registre R1, donnez l'instruction en langage d'assemblage correspondant.

Adressage indirect par registre

Dans ce mode d'adressage, l'adresse mémoire de la donnée est déterminée à partir de la valeur d'un registre (appelé registre de base). Il est également possible de coder dans l'instruction of décalage (offset) qui sera ajouté à la valeur du registre (on parle d'adressage indirect par registre avec offset).

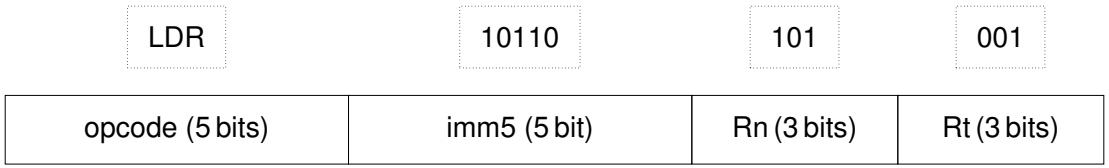


Figure: Exemple d'adressage indirect par registre avec offset avec l'instruction ldr de l'ARMv7m (instruction sur 16 bits)

Réponse

On considère la structure ci-dessous :

```
struct personne{
uint32_t id;
uint32_t age;
uint32_t taille;
}
```

En supposant que l'adresse de début de la structure personne soit stockée dans R5 et que la destination soit le registre R1, donnez l'instruction en langage d'assemblage correspondant.

LDR R1, [R5,#4]

Adressage indirect à deux registres

Ce mode d'adressage est similaire à l'adressage indirect par registre mais cette fois-ci un registre est utilisé comme registre de base et le deuxième sert index. Son utilisation typique est l'accès à un élément d'un tableau.

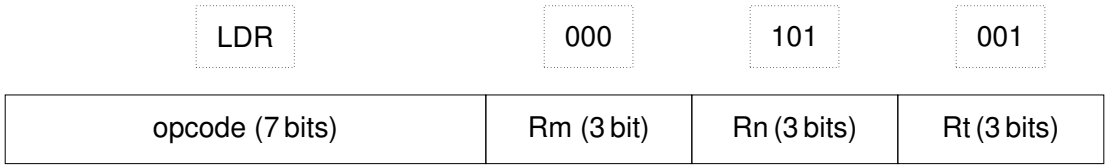


Figure: Exemple d'adressage indirect à deux registres avec l'instruction ldr de l'ARMv7m (instruction sur 16 bits)

Dans cet exemple :

- L'opération à exécuter est une lecture en mémoire (LDR).
- L'opérande est un registre qui contient l'adresse à lire (Rn = R5).
- La destination (target) est un registre (Rt = R1).
- L'index est donné par la registre Rm = R0.

En langage d'assemblage ARM, cette instruction s'écrit :

```
ldr R1, [R5,R0]
```

Adressage indirect à deux registres

Ce mode d'adressage est similaire à l'adressage indirect par registre mais cette fois-ci un registre est utilisé comme registre de base et le deuxième sert index. Son utilisation typique est l'accès à un élément d'un tableau.

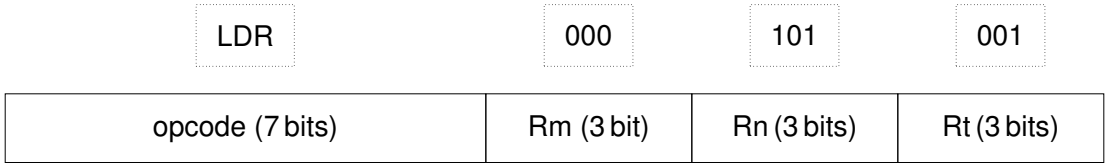


Figure: Exemple d'adressage indirect à deux registres avec l'instruction ldr de l'ARMv7m (instruction sur 16 bits)

Remarque

Il est plus naturel d'avoir un index qui correspond à l'élément du tableau désiré. Hors les élém ents de tableaux peuvent être des multiples de l'octet (4 octets pour des entiers 32 bits par exemple), il est donc nécessaire de multiplier la valeur de index par 4 avant l'accès à la mémoire. Cela se fait par le biais d'une instruction de décalage binaire à gauche (shift en anglais) de 2, ce qui donne en assembleur ARM :

```
ldr R1, [R5,R0, LSL 2]
```

2 est codé dans l'instruction de manière immédiate, et ne peut dépasser 2 bits.

Adressage indirect par registre pré/post-indexés

Le mode d'adressage indirect à deux registres peut être utilisé pour le parcours de tableaux, mais cela monopolise 2 registres. Dans ce cas il est plus efficace d'utiliser l'adressage indirect par registre pré/post-indexés qui permet d'incrémenter ou décrémenter le registre de base avant (pré-indexé) ou après (post-indexé) l'opération. Dans ce cas, un seul registre sera nécessaire, mais il faut garder à l'esprit qu'il et modifié.

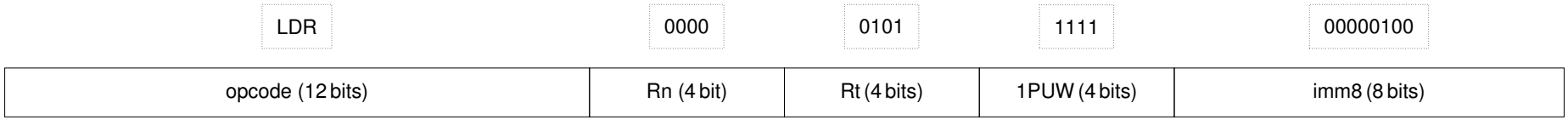


Figure: Exemple d'adressage indirect par registre pré-post indexé avec l'instruction ldr de l'ARMv7m (instruction sur 32 bits)

Dans cet exemple :

- L'opération à exécuter est une lecture en mémoire (LDR).
- L'opérande est un registre qui contient l'adresse à lire (Rn = R0).
- La destination (target) est un registre (Rt = R5).
- Le pré-post incrément est codé dans le champ imm8.

En langage d'assemblage ARM, l'instruction avec pré-incrément s'écrit :

ldr Rt, [Rn,#imm8]!

Adressage indirect par registre pré/post-indexés

Le mode d'adressage indirect à deux registres peut être utilisé pour le parcours de tableaux, mais cela monopolise 2 registres. Dans ce cas il est plus efficace d'utiliser l'adressage indirect par registre pré/post-indexés qui permet d'incrémenter ou décrémenter le registre de base avant (pré-indexé) ou après (post-indexé) l'opération. Dans ce cas, un seul registre sera nécessaire, mais il faut garder à l'esprit qu'il et modifié.

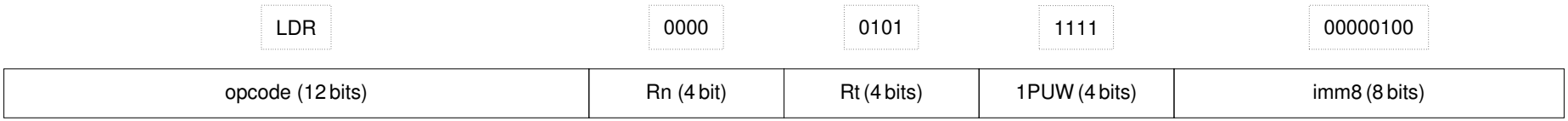


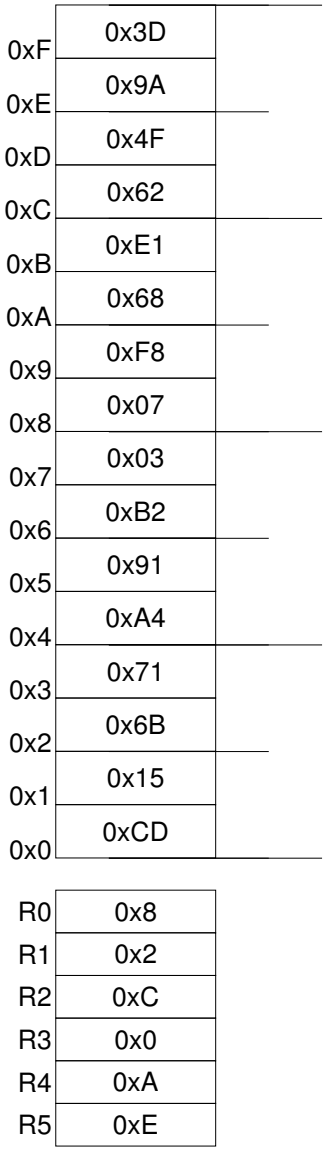
Figure: Exemple d'adressage indirect par registre pré-post indexé avec l'instruction ldr de l'ARMv7m (instruction sur 32 bits)

En langage d'assemblage ARM, l'instruction avec pré-incrément s'écrit :

ldr Rt, [Rn,#imm8]!

Et pour le post-incrément :

ldr Rt, [Rn],#imm8



R0	0x8
R1	0x2
R2	0xC
R3	0x0
R4	0xA
R5	0xE

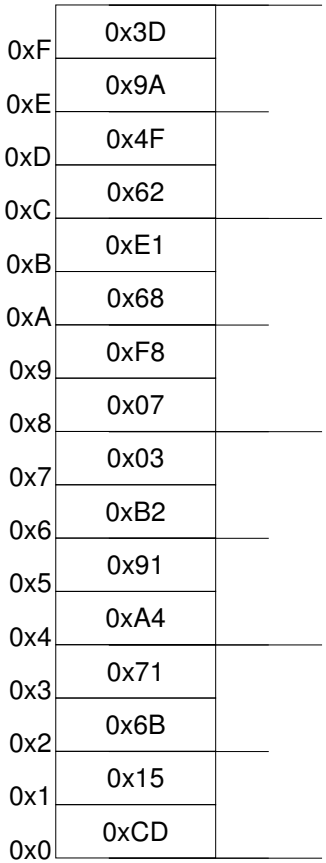
Configuration

Processeur ARM Cortex-M d'architecture 32 bits little-endian.

Question

Donnez l'état des registres après l'instruction :

MOV R3, #12



R0	0x8
R1	0x2
R2	0xC
R3	0x0
R4	0xA
R5	0xE

Configuration

Processeur ARM Cortex-M d'architecture 32 bits little-endian.

Question

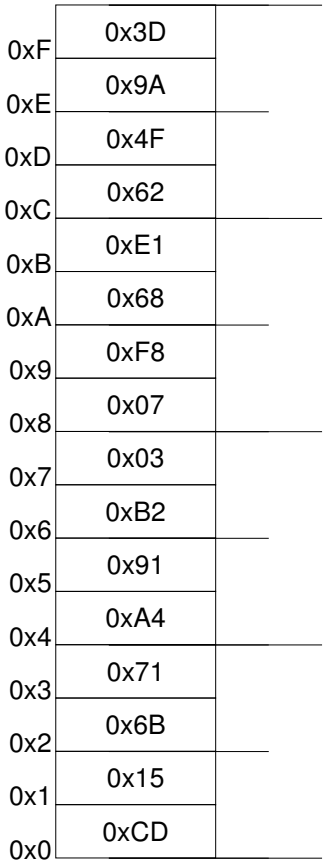
Donnez l'état des registres après l'instruction :

MOV R3, #12

Réponse

C'est un adressage immédiat, et le registre R3 est modifié :

R3 = 0xC



R0	0x8
R1	0x2
R2	0xC
R3	0x0
R4	0xA
R5	0xE

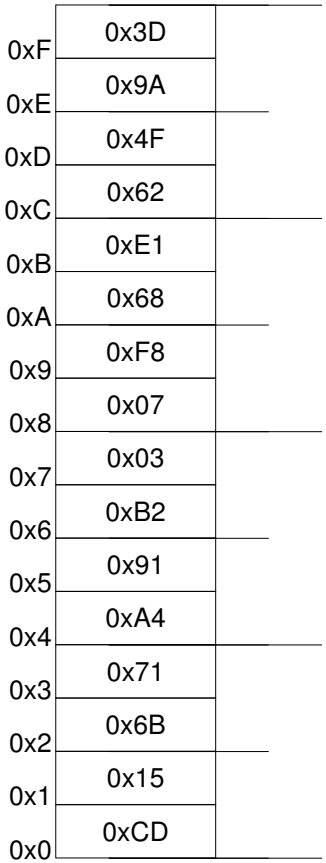
Configuration

Processeur ARM Cortex-M d'architecture 32 bits little-endian.

Question

Donnez l'état des registres après l'instruction :

LDR R1,[R1,#-2]



R0	0x8
R1	0x2
R2	0xC
R3	0x0
R4	0xA
R5	0xE

Configuration

Processeur ARM Cortex-M d'architecture 32 bits little-endian.

Question

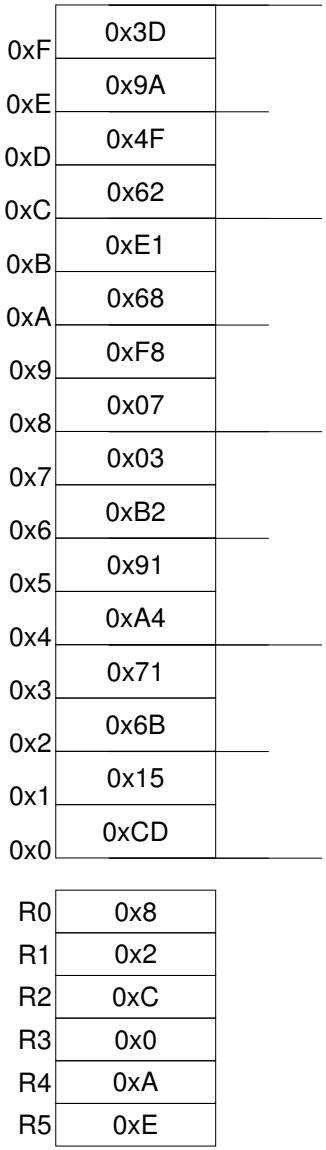
Donnez l'état des registres après l'instruction :

LDR R1,[R1,#-2]

Réponse

C'est un adressage indirect par registre avec offset, avec R1 modifié :

R1 = 0x716B15CD



R0	0x8
R1	0x2
R2	0xC
R3	0x0
R4	0xA
R5	0xE

Configuration

Processeur ARM Cortex-M d'architecture 32 bits little-endian.

Question

Donnez l'état des registres après l'instruction :

LDR R4,[R2,#-6]!

0xF	0x3D
0xE	0x9A
0xD	0x4F
0xC	0x62
0xB	0xE1
0xA	0x68
0x9	0xF8
0x8	0x07
0x7	0x03
0x6	0xB2
0x5	0x91
0x4	0xA4
0x3	0x71
0x2	0x6B
0x1	0x15
0x0	0xCD

R0	0x8
R1	0x2
R2	0xC
R3	0x0
R4	0xA
R5	0xE

Configuration

Processeur ARM Cortex-M d'architecture 32 bits little-endian.

Question

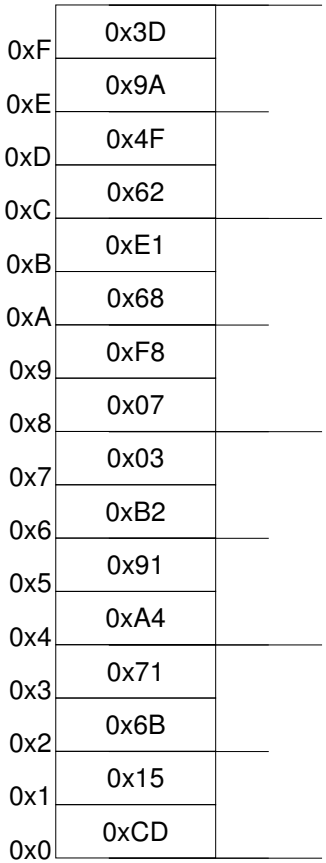
Donnez l'état des registres après l'instruction :

LDR R4,[R2,#-6]!

Réponse

C'est un adressage indirect par registre pré-incrémenté. Le registre R2 est d'abord modifié pour servir de registre de base pour l'adressage indirect par registre. Le registre R4 est modifié également :

R4 = 0xF80703B2
R2 = 0x6



R0	0x8
R1	0x2
R2	0xC
R3	0x0
R4	0xA
R5	0xE

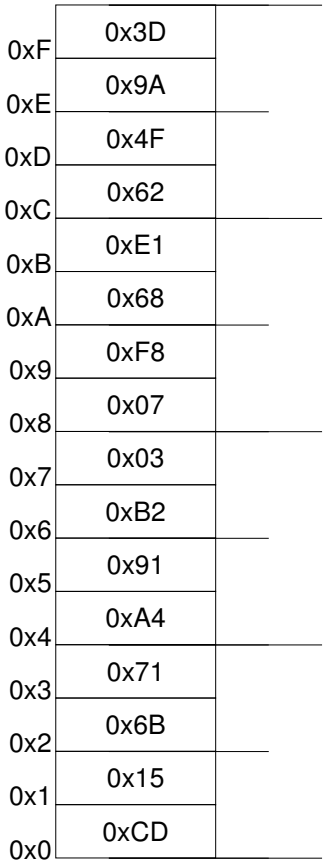
Configuration

Processeur ARM Cortex-M d'architecture 32 bits little-endian.

Question

Donnez l'état des registres après l'instruction :

LDR R2,[R0,R1]



R0	0x8
R1	0x2
R2	0xC
R3	0x0
R4	0xA
R5	0xE

Configuration

Processeur ARM Cortex-M d'architecture 32 bits little-endian.

Question

Donnez l'état des registres après l'instruction :

LDR R2,[R0,R1]

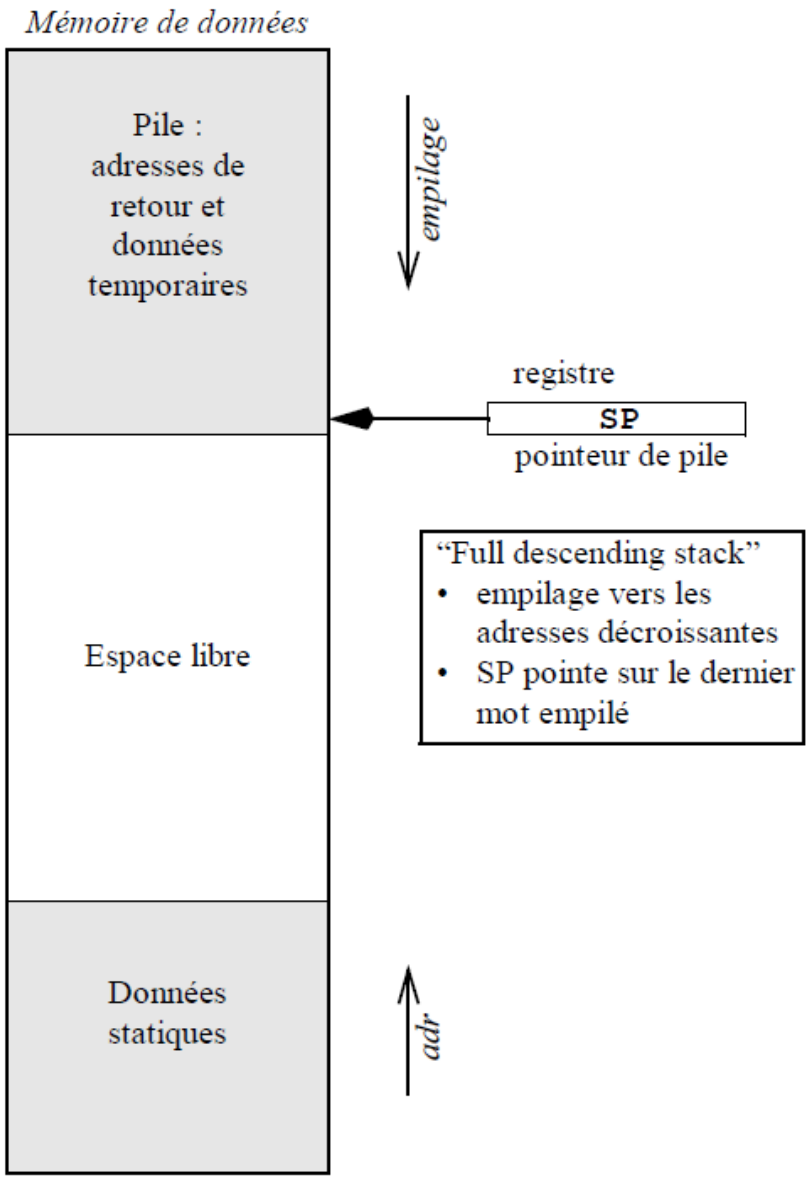
Réponse

C'est un adressage indirect à deux registres. Seul le registre R2 est modifié :

R0 + R1 = 0xA
R2 = 0x4F62E168

Pile et sous-programmes

La pile

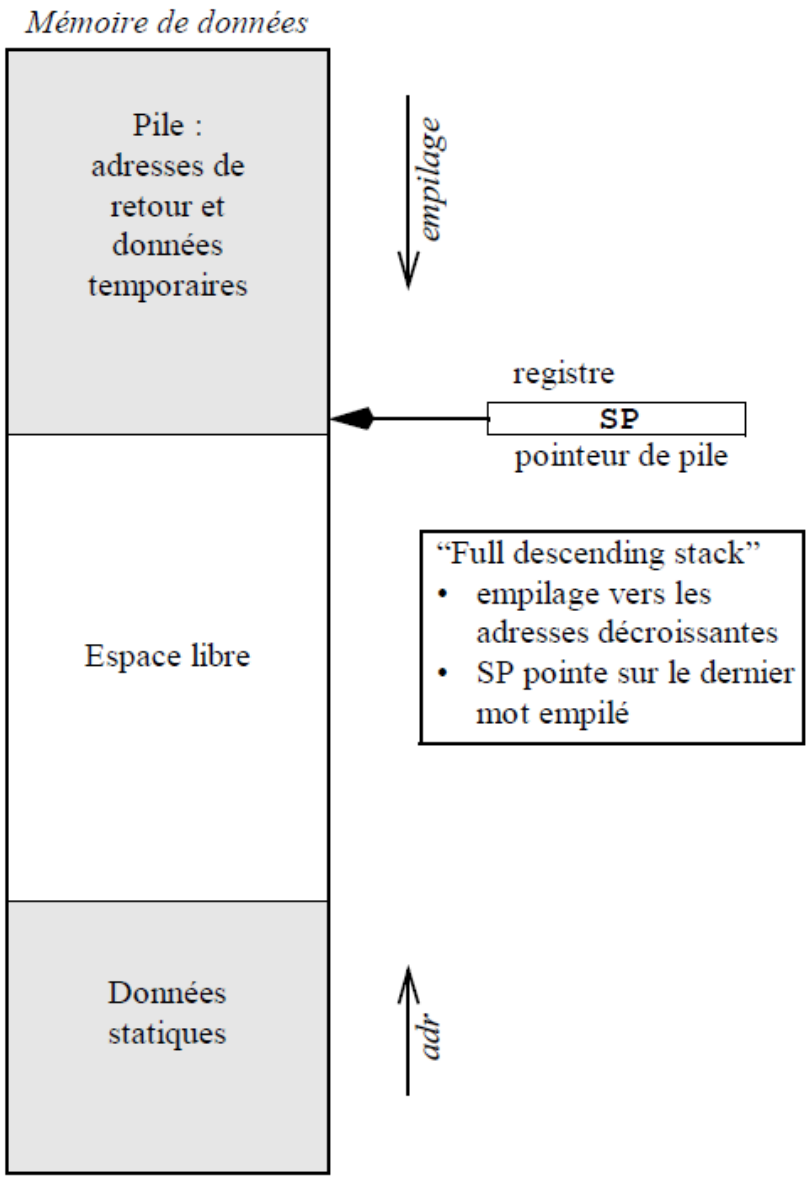


La pile (stack) est une structure de données à accès séquentiel du type LIFO (Last-In, First-Out).

Pour gérer une pile dans la mémoire de données (RAM), il suffit d'un registre pour indiquer la position courante ("pointeur de pile"), qu'on utilise pour empiler ou dépiler des données par adressage indexé.

Afin de définir une pile, deux conventions sont à définir :

- La direction de l'empilage : adresses croissantes (ascending) ou décroisantes (descending) ;
- Le pointer de pile se positionne sur adresse de la dernière valeur empilée (full) ou la future valeur (empty).



Cas du Arm Cortex-M

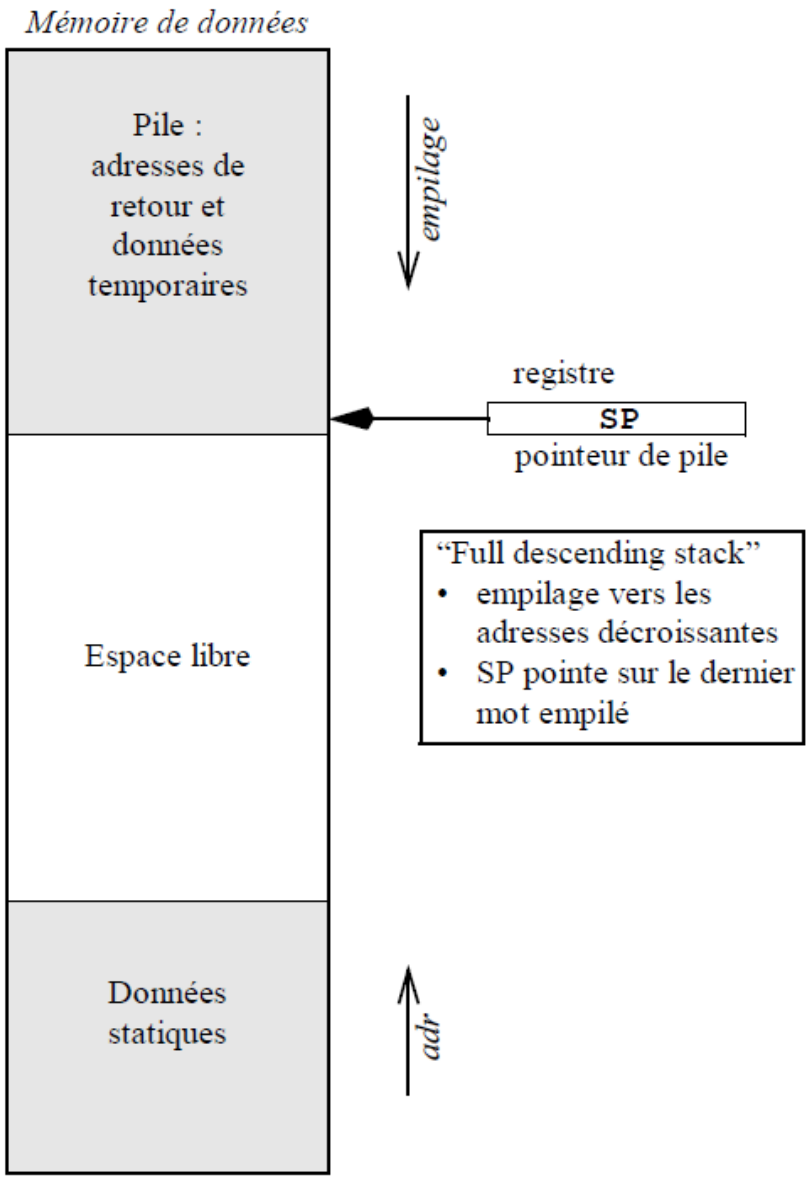
C'est une pile **full descending**, c'est-à-dire que le pointeur de pile pointe sur la dernière valeur empilée, et l'empilage se fait vers les adresses plus faibles (comme l'exemple à droite).

L'action d'empiler est l'opération dite de PUSH. Elle demande :

- Le décrément du pointeur de pile.
- L'écriture de la valeur en mémoire.

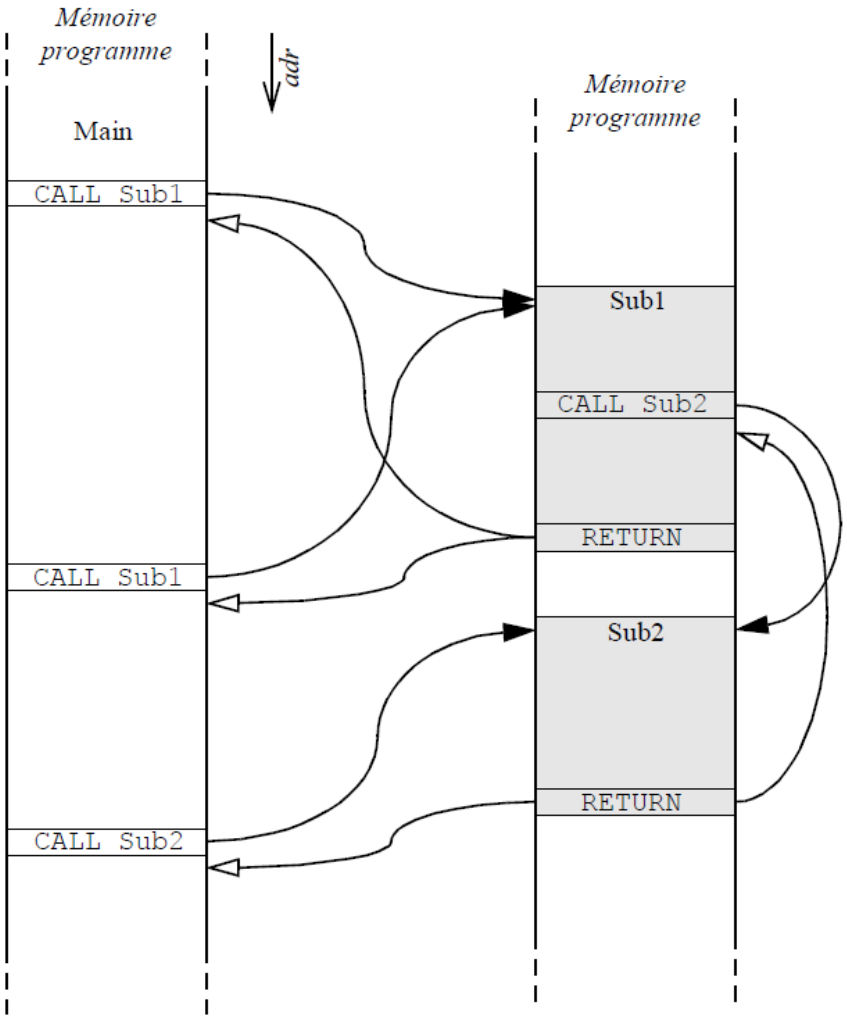
L'action de dépiler est l'opération dite de POP. Elle demande :

- La lecture de la valeur en mémoire.
- L'incrément du pointeur de pile.



Remarques

- Les valeurs empilées ou dépilées sont en principe des registres.
- Permet d'artificiellement gérer plus de registre que ce que possède le processeur en empilant/dépilant temporairement certains registres.
- En empilant tous les registres en cours d'utilisation (y compris le compteur programme et le pointeur de pile), il est possible de sauvegarder l'état d'un programme pour restaurer son état plus tard : c'est le changement de contexte (context switch).
- L'opération de dépilage POP ne signifie pas que la valeur disparait en mémoire.

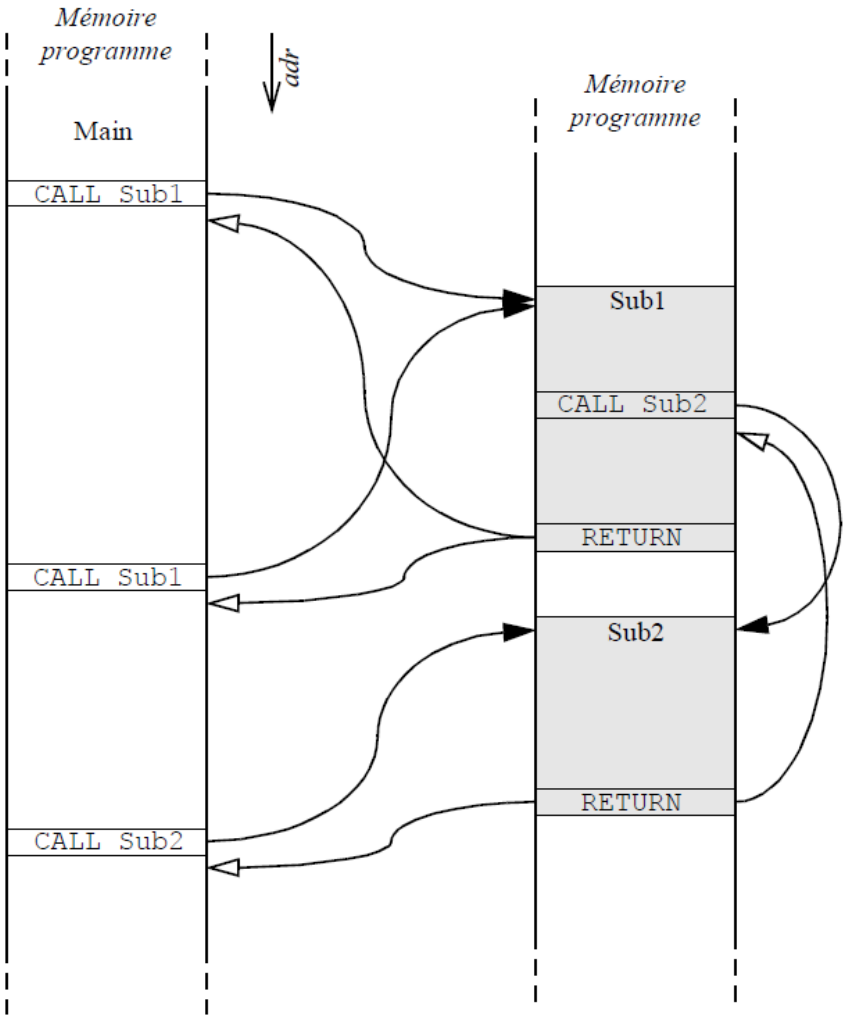


Sous-programme (subroutine)

C'est une suite d'instruction que l'on souhaite utiliser plusieurs fois dans un programme. Dans les langages structurés, ils sont appelés fonctions.

Saut vs appel de sous-programme

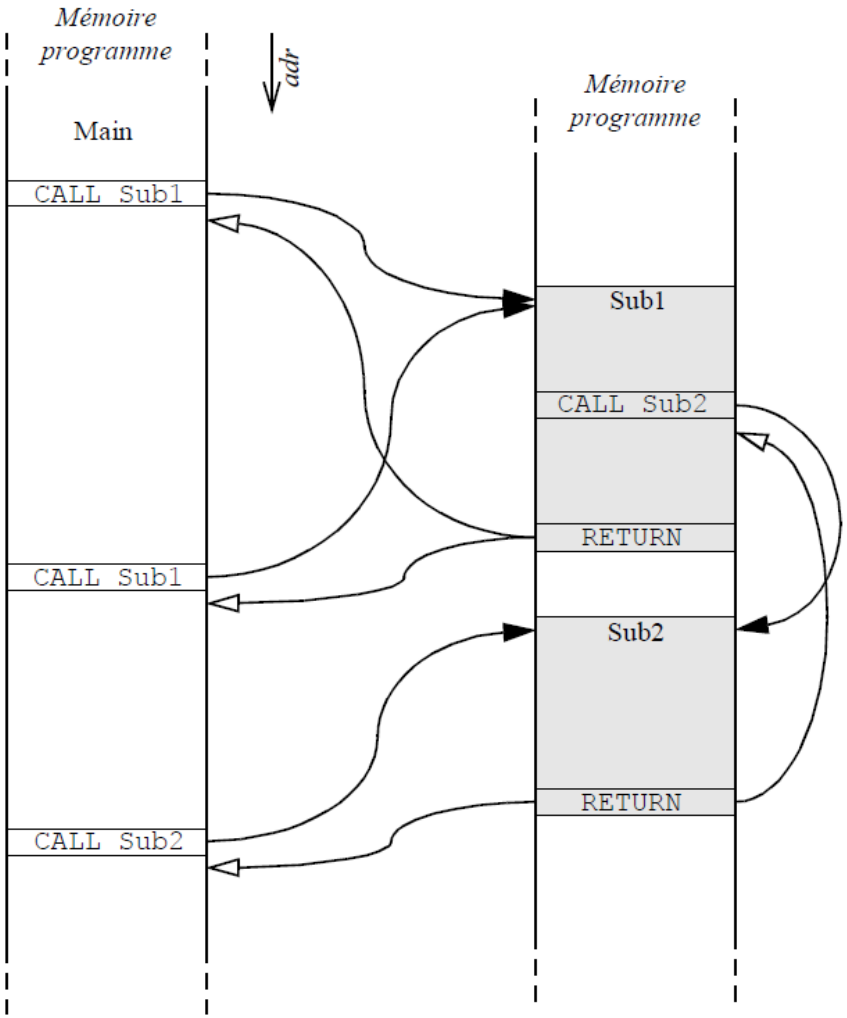
- Lors d'une opération de saut, seul le compteur programme est modifié et positionné à l'adresse désirée dans le code ;
- Lors d'un appel à un sous-programme, la seule modification du compteur programme est insuffisante : Il faut également stocker l'adresse de retour (= adresse de l'instruction suivant l'instruction de l'appel au sous-programme) dans un registre spécifique nommé adresse retour (LR, Link register).



Cas de Intel x86

Les instruction CALL et RETURN permettent d'appeler un sous-programme, et d'en sortir :

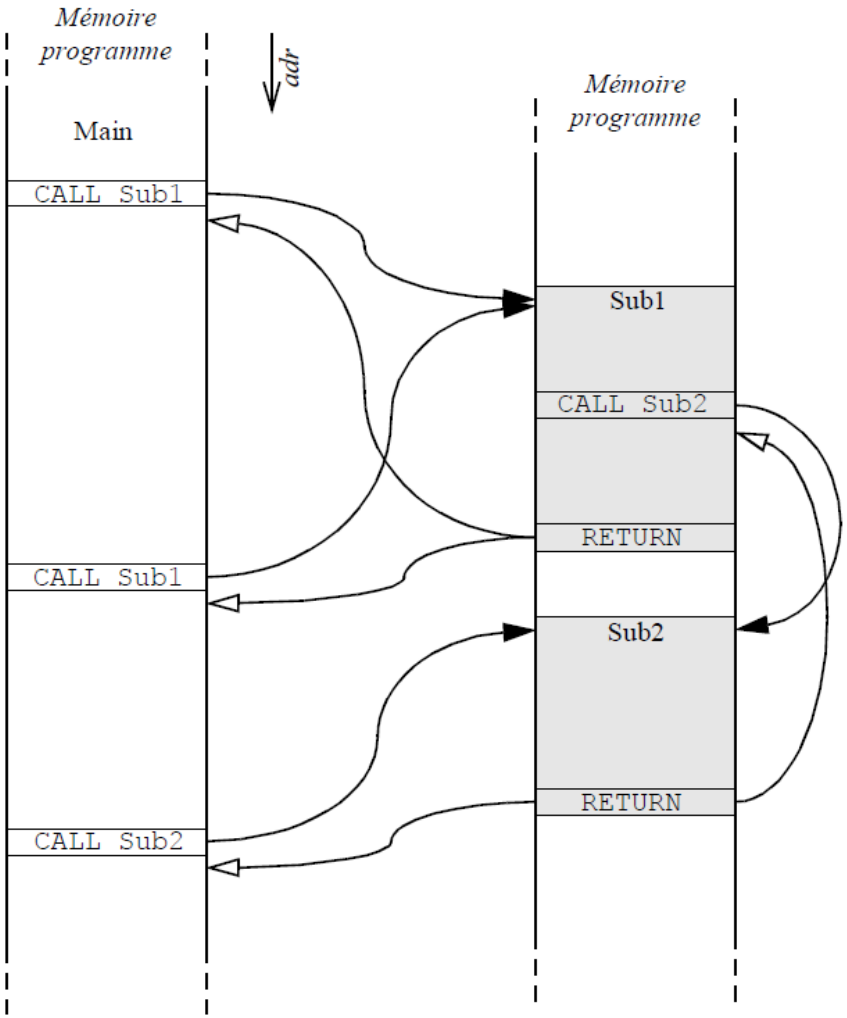
- CALL *Label* : Le compteur programme est positionné sur l'adresse de début du sous-programme et le registre d'adresse retour positionné juste à l'adresse de l'instruction suivant le CALL ;
- RETURN : Le compteur programme est écrasé par la valeur présente dans le registre d'adresse retour.



Cas de Arm

Arm n'a pas à proprement parlé d'instruction dédiée à l'appel de fonction, mais utilise deux instructions qui fonctionnellement apportent la même chose :

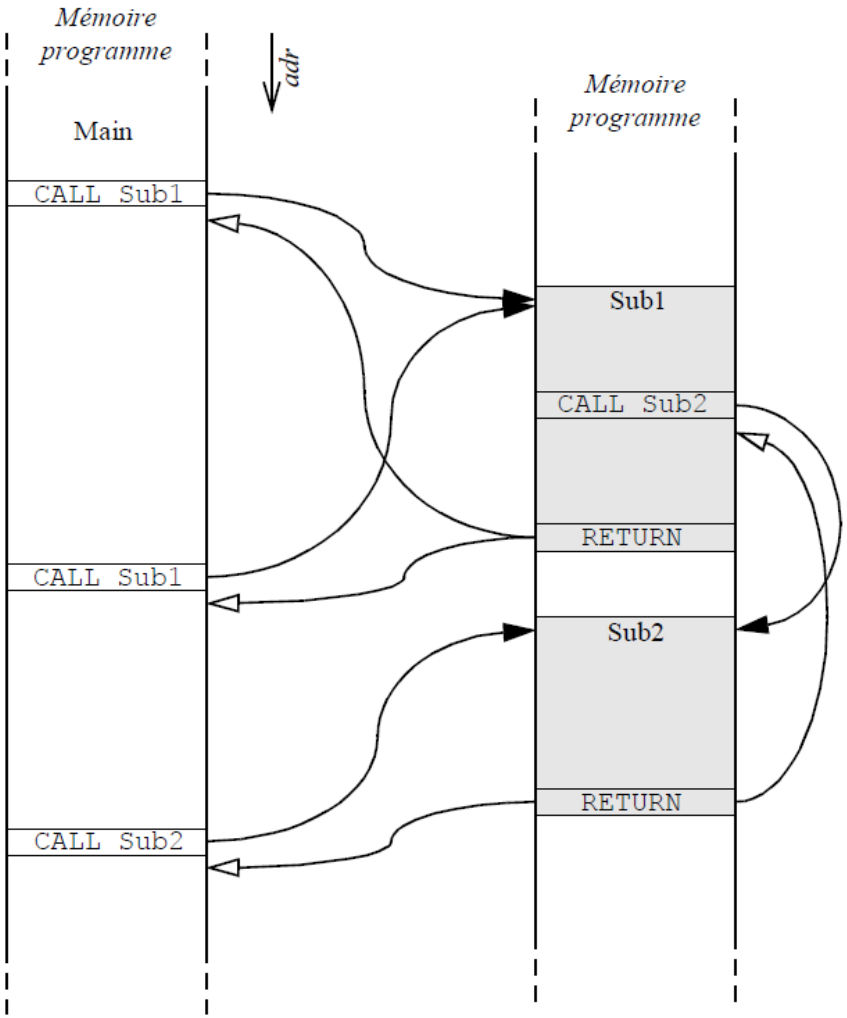
- BL *Label* (branch with link) : Le compteur programme (PC) est positionné sur l'adresse de début du sous-programme et le registre d'adresse retour (LR) positionné juste à l'adresse de l'instruction suivant l'instruction BLX ;
- BX LR : Le compteur programme (PC) est écrasé par la valeur présente dans le registre d'adresse retour (LR).



Cas de l'appel imbriqué (nested)

Question

Cette méthode fonctionne à partir du moment où il n'y a pas appel imbriqué de sous-programmes (comme le cas de *sub1* et *sub 2* sur l'exemple de gauche). A votre avis, comment peut-on réaliser cette imbrication ?



Cas de l'appel imbriqué (nested)

Question

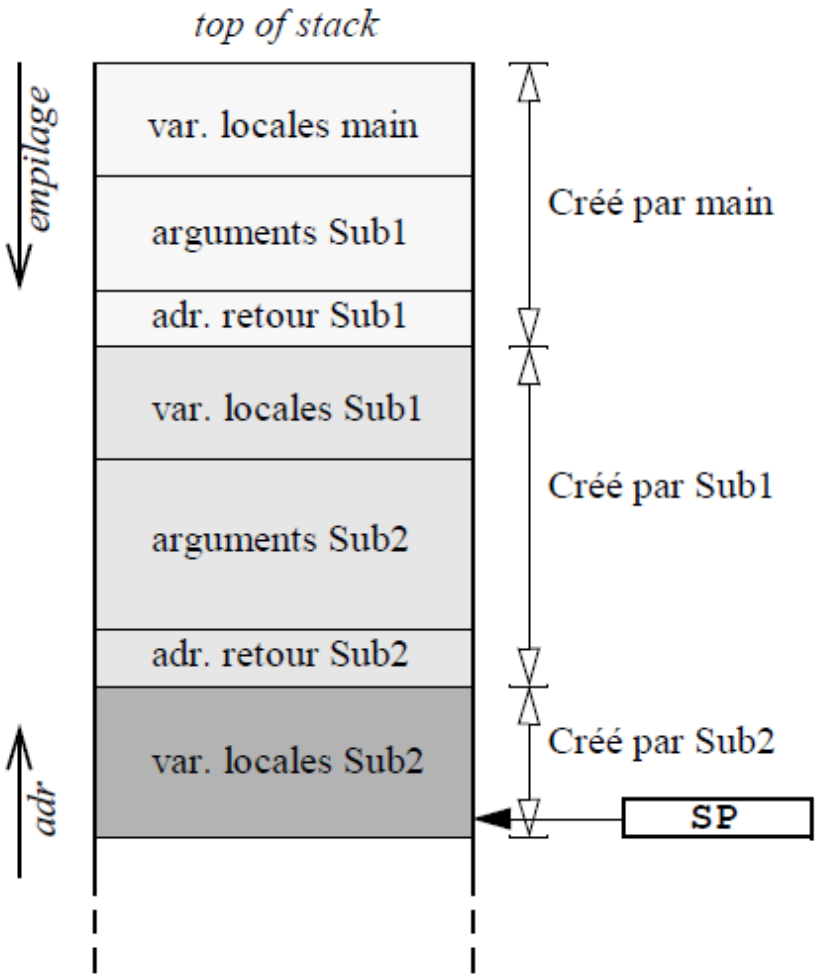
Cette méthode fonctionne à partir du moment où il n'y a pas appel imbriqué de sous-programmes (comme le cas de *sub1* et *sub 2* sur l'exemple de gauche). A votre avis, comment peut-on réaliser cette imbrication ?

Cas de Arm

On utilise la pile :

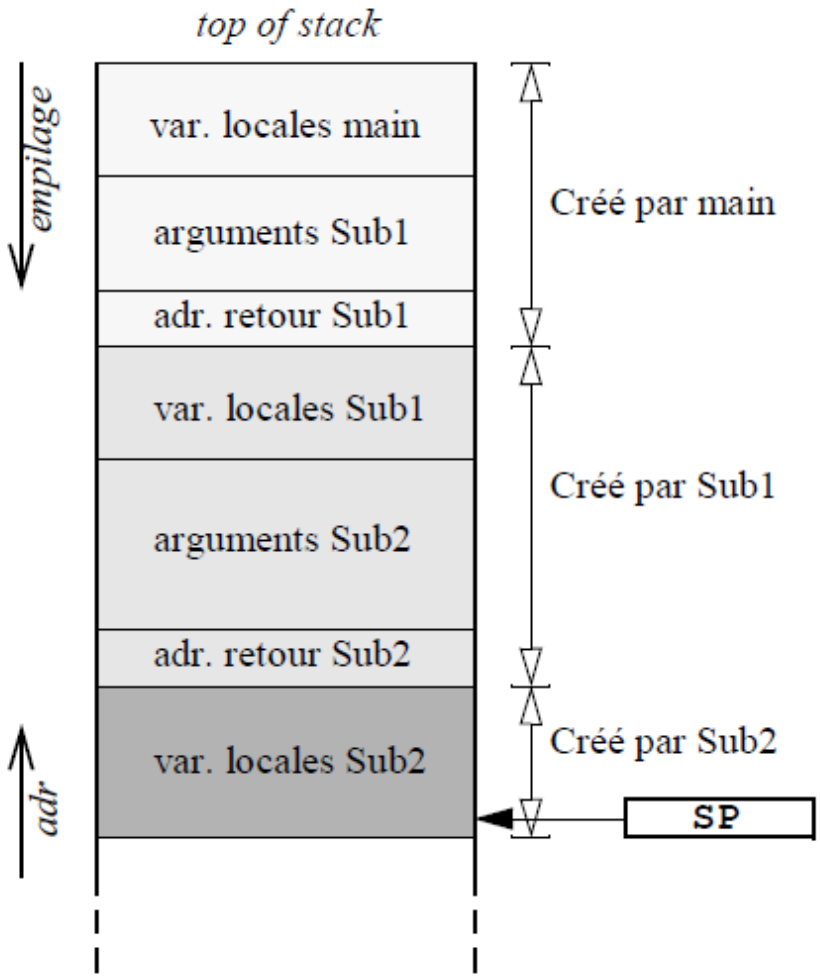
- En début de sous-programme, on utilise PUSH LR pour empiler l'adresse de retour. On peut maintenant appeler un sous-programme sans risque d'écraser sa valeur ;
- On remplace la dernière instruction par POP PC, ce qui aura pour conséquence de dépiler la valeur de LR dans le registre PC.

Remarque : Il faut bien entendu que le nombre d'appels PUSH et POP soient identiques.



Pile et arguments de fonction

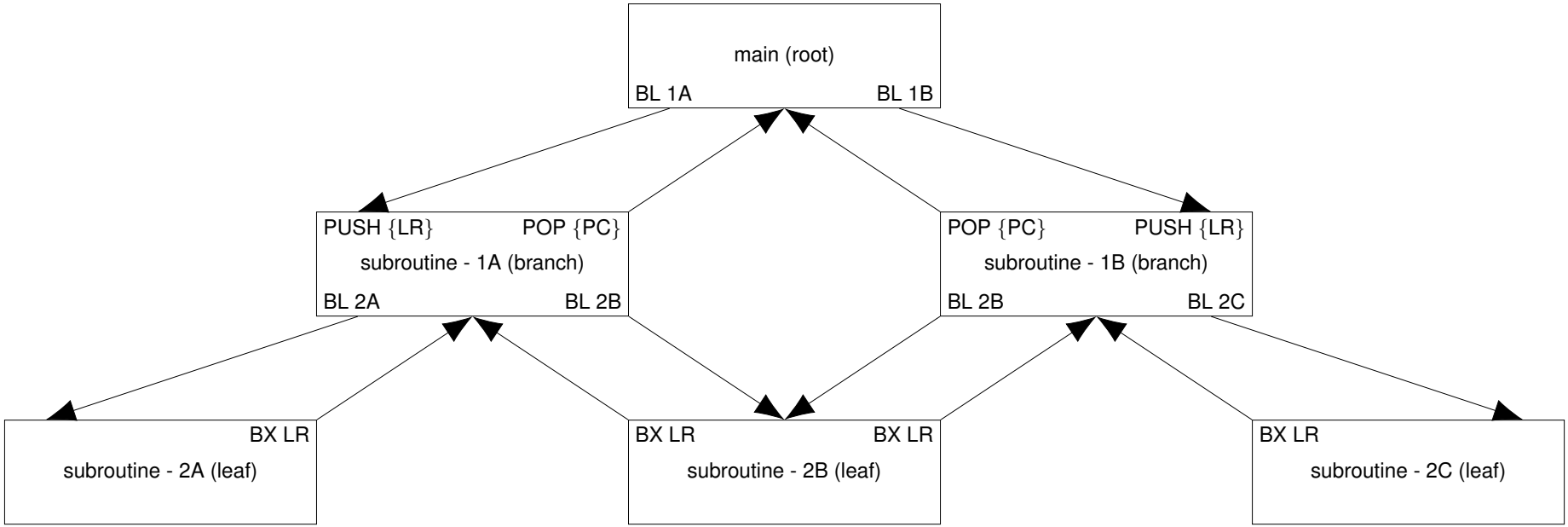
La pile est également un moyen de passer des arguments aux sous-programmes, surtout les architectures x86/x86_64 qui possèdent peu de registres généraux. En revanche, sur les architectures d'Arm et RISC-V, les arguments se passent principalement par registre (puis la pile si trop d'arguments).



Remarques

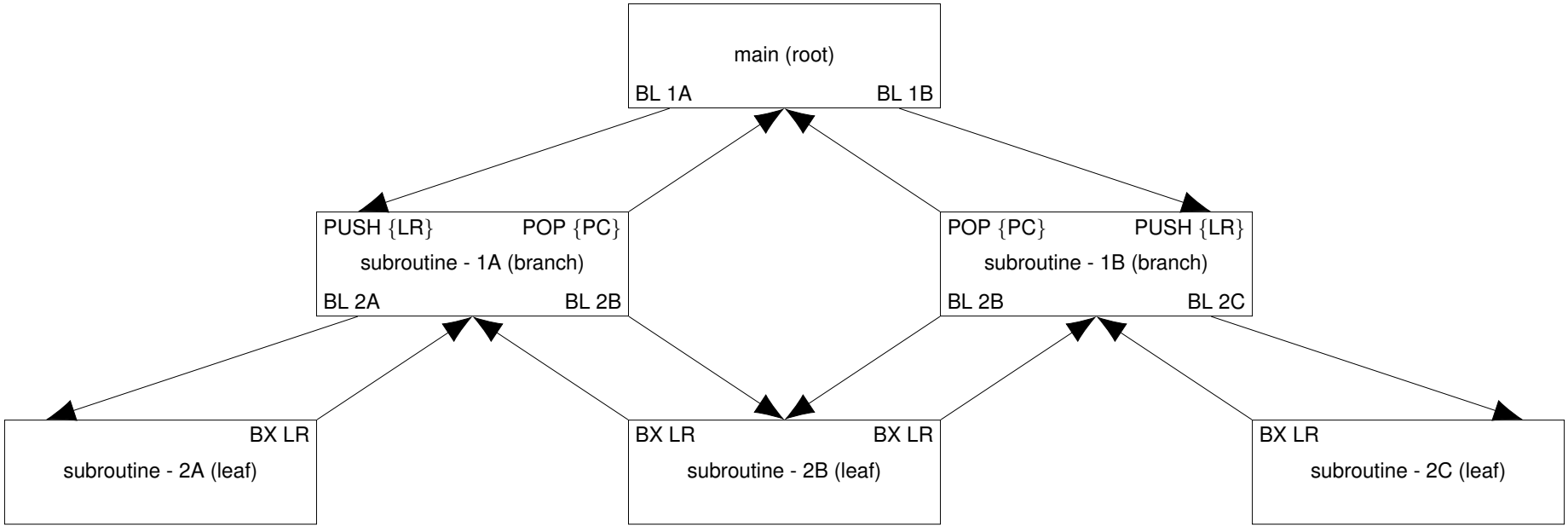
- Il est nullement nécessaire de passer par l'empilage/dépilage pour accéder à un élément de la pile : l'utilisation du pointeur de pile + un offset suffit.
- Avec le concept de pile, la seule limite à l'appel imbriqué de fonctions est la taille de la mémoire RAM.
- Il est tout à fait possible de posséder 1 pile par tâche, afin d'avoir des contextes d'exécutions distincts.

Exemple d'appel récursif (Arm)



- appel de fonction (dans tous les cas) :
  BL myfunction ; branch-and-link, update LR
- corps de la fonction :
  - fonction "feuille" :
    ...
    BX LR ; indirect jump to [LR]
  - fonction quelconque :
    PUSH LR ; empiler l'adresse de retour
    ...
    BL myfunction2
    ...
    POP PC ; dépiler directement dans PC

Exemple d'appel récursif (Arm)



Note

Chez ARM, le bit de faible poids de PC n'est pas utilisé pour accéder aux instructions (dont les adresses sont obligatoirement paires) mais il sert à signaler l'utilisation du jeu d'instructions Thumb ou Thumb-2 (Cortex M3).

Principe

Un code exécutable pouvant être issu de plusieurs langages différents, il est fondamental de se fixer une convention pour l'appel aux fonctions. En particulier, il est nécéssaire de se fixer :

- La manière de passer les arguments d'une fonction au sous-programme ;
- La manière de passer le résultat de la fonction au programme appelant.

ABI

L'Application Binary Interface (ABI) est l'interface bas niveau qui décrit comment les différents composants logiciels communiquent au niveau binaire. Les conventions d'appel de fonctions en font partie. Il ne faut pas confondre ABI avec API, qui définie l'interface entre un code source et une bibliothèque.

Cas de ARM

ARM propose au développeur son standard nommé standard AAPCS (ARM Architecture Procedure Call Standard), qui propose en plus des conventions d'appels, des conventions pour la représentation types de données et les structures du langage C/C++.

On se donne la fonction *var_r* fonction(*var_1* , *var_2*)
Pour se fixer les idées, on suppose que var_1, var_2 sont deux entiers de 32_bits stockés avant l'appel de fonction dans respectivement R4 et R5.
var_r est également un entier de 32 bits qui sera stocké dans R3.
Ci-dessous à quoi ressemble l'appel au sous-programme en langage d'assemblage :

```
mov R0, R4
mov R1, R5
BL fonction
mov R3,R0
```

Question

A partir de l'exemple proposé, expliquez comment fonctionne le passage d'argument selon l'AAPCS.

On se donne la fonction *var_r* fonction(*var_1* , *var_2*)
Pour se fixer les idées, on suppose que var_1, var_2 sont deux entiers de 32_bits stockés avant l'appel de fonction dans respectivement R4 et R5.
var_r est également un entier de 32 bits qui sera stocké dans R3.
Ci-dessous à quoi ressemble l'appel au sous-programme en langage d'assemblage :

```
mov R0, R4
mov R1, R5
BL fonction
mov R3,R0
```

Question

A partir de l'exemple proposé, expliquez comment fonctionne le passage d'argument selon l'AAPCS.

Réponse

Le passage d'argument se fait par registre, en commençant par R0 puis R1 et ainsi de suite. Le retour est également un registre, R0.

Conclusion sur l'appel de fonction

- Le passage des arguments se fait par registres pour les 4 premières variables, en commençant par R0 et jusqu'à R3.
- Le passage des arguments suivants se fait par la pile. La valeur de retour se fait par le registre R0.
- Le contenu des registres R4 à R11, SP et LR doivent être préservés, c'est-à-dire qu'ils doivent avoir une valeur identique au retour de fonction qu'au moment de l'appel.

Remarque

En cas de modification d'un des registres parmi $\{\text{R4}, \cdots, \text{R11}, \text{SP}, \text{LR}\}$ sans rétablir leur valeur, alors il y a un risque de comportement inattendu de l'application. Dans le cas où il s'avère impossible leur préservation, il est possible d'utiliser :

- les registres R0 à R3 ainsi que R12 ;
- la pile.

Question

Que pensez-vous du code suivant ?

```
mul_add:
MUL R4,R1,R2
ADD R0,R0,R4
BX
```

Question

Que pensez-vous du code suivant ?

```
mul_add:
MUL R4,R1,R2
ADD R0,R0,R4
BX
```

Réponse

Il utilise le registre R4 ans le sauvegarder, ce qui est incompatible avec l'AAPCS

```
mul_add:
push {R4}
MUL R4,R1,R2
ADD R0,R0,R4
POP {R4}
BX
```

Quiz

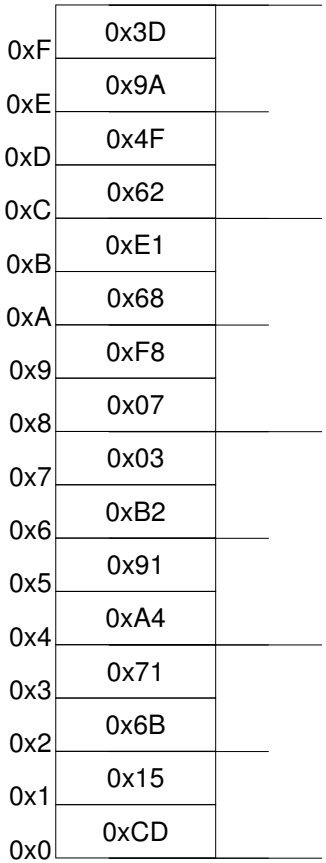
La représentation du langage d'assemblage est :	
A. textuel	B. binaire
Pour chaque architecture de processeur, le langage est :	
C. Unique	D. le même

La représentation du langage machine est :	
A. textuel	B. binaire
Pour chaque architecture de processeur, le langage est :	
C. Unique	D. le même

L'architecture du jeu d'instructions (ISA) est :	
A. Connue du programmeur	B. Inconnue du programmeur
Elle donne notamment :	
C. Les instructions supportées par le processeur	D. L'associativité des caches
E. L'organisation de la mémoire	F. La structure du pipeline

L'architecture RISC possède un jeu d'instructions de taille fixe :	
A. Oui	B. Non
On retrouve ce type d'architecture dans :	
C. Les ordinateurs généralistes	D. Les systèmes embarqués

L'architecture CISC possède un jeu d'instructions de taille fixe :	
A. Oui	B. Non
On retrouve ce type d'architecture dans :	
C. Les ordinateurs généralistes	D. Les systèmes embarqués



R0	0x8
R1	0x2
R2	0xC
R3	0x0
R4	0x4
R5	0xE

Configuration

Processeur ARM Cortex-M d'architecture 32 bits little-endian.

Question

On suppose que l'instruction suivante est exécutée :

LDR R5, [R4,R1,LSL 2]

Donnez quelles opérations en langage c seraient à l'origine de cette utilisation ? On suppose que R1 fait référence à une variable *i*, R4 une variable *j* et R5 une variable *k*.

0xF	0x3D
0xE	0x9A
0xD	0x4F
0xC	0x62
0xB	0xE1
0xA	0x68
0x9	0xF8
0x8	0x07
0x7	0x03
0x6	0xB2
0x5	0x91
0x4	0xA4
0x3	0x71
0x2	0x6B
0x1	0x15
0x0	0xCD

R0	0x8
R1	0x2
R2	0xC
R3	0x0
R4	0x4
R5	0xE

Configuration

Processeur ARM Cortex-M d'architecture 32 bits little-endian.

Question

On suppose que l'instruction suivante est exécutée :

LDR R5, [R4,R1,LSL 2]

Donnez quelles opérations en langage c seraient à l'origine de cette utilisation ? On suppose que R1 fait référence à une variable *i*, R4 une variable *j* et R5 une variable *k*.

Réponse

C'est un adressage indirect à deux registres, utilisé principalement pour l'accès à des tableaux. *j* est donc l'adresse du tableau, et $i << 2$ est l'index. Pour plus de lisibilité, on pose $I = i << 2$ Un exemple possible :

```
uint32_t j[4]
k = j[I]
```

Jeu d'instructions : Classification, codage

Il n'existe pas de standard portable en langage d'assemblage (i.e. indépendant de l'architecture), mais certains nom d'opérations reviennent régulièrement.

Remarque

Les opérandes et la destination d'une opération sont soit des registres, soit des adresses. Hormis quelques rares exceptions, ARM et RISC-V n'acceptent que les registres en tant qu'opérande ou destination.

Copie de données

- **MOV (Move)** : Opération de copie d'une valeur dans un registre. Il peut provenir :
  - D'un autre registre (MOV Rd,Rm).
  - D'une valeur immédiate (MOV Rd, imm).
- **LDR (Load)** : Opération de lecture en mémoire dont la destination est un registre. Il est composé d'une adresse de base, et d'un offset qui sera ajouté à cette adresse pour calculer l'adresse de lecture. L'adresse de base est en principe contenu dans un registre (obligatoire chez ARM), et pour l'offset, il existe différentes méthodes :
  - **Immédiate :** l'offset est une constante codée dans l'instruction (LDR Rt,[Rn,#imm])
  - **Registre :** un autre registre est utilisé comme offset (LDR Rt,[Rn,Rm]). Celui-ci peut subir un décalage avant l'ajout au registre de base (LDR Rt,[Rn,Rm,LSL #imm])
  - **Pré-indexé :** L'adresse finale est calculée, le registre de base est mis-à-jour avec cette valeur, puis l'opération de lecture est effectuée (LDR Rt,[Rn,#imm]!).
  - **Post-indexé :** L'opération de lecture est effectuée, puis le registre de base est mis-à-jour en y ajoutant l'offset (LDR Rt,[Rn],#imm).
- **STR (Store)** : Opération d'écriture, suit une utilisation similaire que que LD, mais la destination est une adresse. Par exemple STR Rd,[Rn] copiera la valeur de Rd à l'adresse qui est stockée dans Rn.
- **CPY :** equivalent à MOV.

Copie de données

- **MOV (Move)** : Opération de copie d'une valeur dans un registre. Il peut provenir :
  - D'un autre registre (MOV Rd,Rm).
  - D'une valeur immédiate (MOV Rd, imm).
- **LDR (Load)** : Opération de lecture en mémoire dont la destination est un registre. Il est composé d'une adresse de base, et d'un offset qui sera ajouté à cette adresse pour calculer l'adresse de lecture. L'adresse de base est en principe contenu dans un registre (obligatoire chez ARM), et pour l'offset, il existe différentes méthodes :
  - **Immédiate :** l'offset est une constante codée dans l'instruction (LDR Rt,[Rn,#imm])
  - **Registre :** un autre registre est utilisé comme offset (LDR Rt,[Rn,Rm]). Celui-ci peut subir un décalage avant l'ajout au registre de base (LDR Rt,[Rn,Rm,LSL #imm])
  - **Pré-indexé :** L'adresse finale est calculée, le registre de base est mis-à-jour avec cette valeur, puis l'opération de lecture est effectuée (LDR Rt,[Rn,#imm]!).
  - **Post-indexé :** L'opération de lecture est effectuée, puis le registre de base est mis-à-jour en y ajoutant l'offset (LDR Rt,[Rn],#imm).
- **STR (Store)** : Opération d'écriture, suit une utilisation similaire que que LD, mais la destination est une adresse. Par exemple STR Rd,[Rn] copiera la valeur de Rd à l'adresse qui est stockée dans Rn.
- **CPY :** equivalent à MOV.

Copie de données

- **MOV (Move)** : Opération de copie d'une valeur dans un registre. Il peut provenir :
  - D'un autre registre (MOV Rd,Rm).
  - D'une valeur immédiate (MOV Rd, imm).
- **LDR (Load)** : Opération de lecture en mémoire dont la destination est un registre. Il est composé d'une adresse de base, et d'un offset qui sera ajouté à cette adresse pour calculer l'adresse de lecture. L'adresse de base est en principe contenu dans un registre (obligatoire chez ARM), et pour l'offset, il existe différentes méthodes :
  - **Immédiate :** l'offset est une constante codée dans l'instruction (LDR Rt,[Rn,#imm])
  - **Registre :** un autre registre est utilisé comme offset (LDR Rt,[Rn,Rm]). Celui-ci peut subir un décalage avant l'ajout au registre de base (LDR Rt,[Rn,Rm,LSL #imm])
  - **Pré-indexé :** L'adresse finale est calculée, le registre de base est mis-à-jour avec cette valeur, puis l'opération de lecture est effectuée (LDR Rt,[Rn,#imm]!).
  - **Post-indexé :** L'opération de lecture est effectuée, puis le registre de base est mis-à-jour en y ajoutant l'offset (LDR Rt,[Rn],#imm).
- **STR (Store)** : Opération d'écriture, suit une utilisation similaire que que LD, mais la destination est une adresse. Par exemple STR Rd,[Rn] copiera la valeur de Rd à l'adresse qui est stockée dans Rn.
- **CPY :** equivalent à MOV.

Copie de données

- **MOV (Move)** : Opération de copie d'une valeur dans un registre. Il peut provenir :
  - D'un autre registre (MOV Rd,Rm).
  - D'une valeur immédiate (MOV Rd, imm).
- **LDR (Load)** : Opération de lecture en mémoire dont la destination est un registre. Il est composé d'une adresse de base, et d'un offset qui sera ajouté à cette adresse pour calculer l'adresse de lecture. L'adresse de base est en principe contenu dans un registre (obligatoire chez ARM), et pour l'offset, il existe différentes méthodes :
  - **Immédiate :** l'offset est une constante codée dans l'instruction (LDR Rt,[Rn,#imm])
  - **Registre :** un autre registre est utilisé comme offset (LDR Rt,[Rn,Rm]). Celui-ci peut subir un décalage avant l'ajout au registre de base (LDR Rt,[Rn,Rm,LSL #imm])
  - **Pré-indexé :** L'adresse finale est calculée, le registre de base est mis-à-jour avec cette valeur, puis l'opération de lecture est effectuée (LDR Rt,[Rn,#imm]!).
  - **Post-indexé :** L'opération de lecture est effectuée, puis le registre de base est mis-à-jour en y ajoutant l'offset (LDR Rt,[Rn],#imm).
- **STR (Store)** : Opération d'écriture, suit une utilisation similaire que que LD, mais la destination est une adresse. Par exemple STR Rd,[Rn] copiera la valeur de Rd à l'adresse qui est stockée dans Rn.
- **CPY :** equivalent à MOV.

Copie de données

- **MOV (Move)** : Opération de copie d'une valeur dans un registre. Il peut provenir :
  - D'un autre registre (MOV Rd,Rm).
  - D'une valeur immédiate (MOV Rd, imm).
- **LDR (Load)** : Opération de lecture en mémoire dont la destination est un registre. Il est composé d'une adresse de base, et d'un offset qui sera ajouté à cette adresse pour calculer l'adresse de lecture. L'adresse de base est en principe contenu dans un registre (obligatoire chez ARM), et pour l'offset, il existe différentes méthodes :
  - **Immédiate :** l'offset est une constante codée dans l'instruction (LDR Rt,[Rn,#imm])
  - **Registre :** un autre registre est utilisé comme offset (LDR Rt,[Rn,Rm]). Celui-ci peut subir un décalage avant l'ajout au registre de base (LDR Rt,[Rn,Rm,LSL #imm])
  - **Pré-indexé :** L'adresse finale est calculée, le registre de base est mis-à-jour avec cette valeur, puis l'opération de lecture est effectuée (LDR Rt,[Rn,#imm]!).
  - **Post-indexé :** L'opération de lecture est effectuée, puis le registre de base est mis-à-jour en y ajoutant l'offset (LDR Rt,[Rn],#imm).
- **STR (Store)** : Opération d'écriture, suit une utilisation similaire que que LD, mais la destination est une adresse. Par exemple STR Rd,[Rn] copiera la valeur de Rd à l'adresse qui est stockée dans Rn.
- **CPY :** equivalent à MOV.

Copie de données

- **MOV (Move)** : Opération de copie d'une valeur dans un registre. Il peut provenir :
  - D'un autre registre (MOV Rd,Rm).
  - D'une valeur immédiate (MOV Rd, imm).
- **LDR (Load)** : Opération de lecture en mémoire dont la destination est un registre. Il est composé d'une adresse de base, et d'un offset qui sera ajouté à cette adresse pour calculer l'adresse de lecture. L'adresse de base est en principe contenu dans un registre (obligatoire chez ARM), et pour l'offset, il existe différentes méthodes :
  - **Immédiate :** l'offset est une constante codée dans l'instruction (LDR Rt,[Rn,#imm])
  - **Registre :** un autre registre est utilisé comme offset (LDR Rt,[Rn,Rm]). Celui-ci peut subir un décalage avant l'ajout au registre de base (LDR Rt,[Rn,Rm,LSL #imm])
  - **Pré-indexé :** L'adresse finale est calculée, le registre de base est mis-à-jour avec cette valeur, puis l'opération de lecture est effectuée (LDR Rt,[Rn,#imm]!).
  - **Post-indexé :** L'opération de lecture est effectuée, puis le registre de base est mis-à-jour en y ajoutant l'offset (LDR Rt,[Rn],#imm).
- **STR (Store)** : Opération d'écriture, suit une utilisation similaire que que LD, mais la destination est une adresse. Par exemple STR Rd,[Rn] copiera la valeur de Rd à l'adresse qui est stockée dans Rn.
- **CPY :** equivalent à MOV.

Copie de données

- **MOV (Move)** : Opération de copie d'une valeur dans un registre. Il peut provenir :
  - D'un autre registre (MOV Rd,Rm).
  - D'une valeur immédiate (MOV Rd, imm).
- **LDR (Load)** : Opération de lecture en mémoire dont la destination est un registre. Il est composé d'une adresse de base, et d'un offset qui sera ajouté à cette adresse pour calculer l'adresse de lecture. L'adresse de base est en principe contenu dans un registre (obligatoire chez ARM), et pour l'offset, il existe différentes méthodes :
  - **Immédiate :** l'offset est une constante codée dans l'instruction (LDR Rt,[Rn,#imm])
  - **Registre :** un autre registre est utilisé comme offset (LDR Rt,[Rn,Rm]). Celui-ci peut subir un décalage avant l'ajout au registre de base (LDR Rt,[Rn,Rm,LSL #imm])
  - **Pré-indexé :** L'adresse finale est calculée, le registre de base est mis-à-jour avec cette valeur, puis l'opération de lecture est effectuée (LDR Rt,[Rn,#imm]!).
  - **Post-indexé :** L'opération de lecture est effectuée, puis le registre de base est mis-à-jour en y ajoutant l'offset (LDR Rt,[Rn],#imm).
- **STR (Store)** : Opération d'écriture, suit une utilisation similaire que que LD, mais la destination est une adresse. Par exemple STR Rd,[Rn] copiera la valeur de Rd à l'adresse qui est stockée dans Rn.
- **CPY :** equivalent à MOV.

Copie de données

- **MOV (Move)** : Opération de copie d'une valeur dans un registre. Il peut provenir :
  - D'un autre registre (MOV Rd,Rm).
  - D'une valeur immédiate (MOV Rd, imm).
- **LDR (Load)** : Opération de lecture en mémoire dont la destination est un registre. Il est composé d'une adresse de base, et d'un offset qui sera ajouté à cette adresse pour calculer l'adresse de lecture. L'adresse de base est en principe contenu dans un registre (obligatoire chez ARM), et pour l'offset, il existe différentes méthodes :
  - **Immédiate :** l'offset est une constante codée dans l'instruction (LDR Rt,[Rn,#imm])
  - **Registre :** un autre registre est utilisé comme offset (LDR Rt,[Rn,Rm]). Celui-ci peut subir un décalage avant l'ajout au registre de base (LDR Rt,[Rn,Rm,LSL #imm])
  - **Pré-indexé :** L'adresse finale est calculée, le registre de base est mis-à-jour avec cette valeur, puis l'opération de lecture est effectuée (LDR Rt,[Rn,#imm]!).
  - **Post-indexé :** L'opération de lecture est effectuée, puis le registre de base est mis-à-jour en y ajoutant l'offset (LDR Rt,[Rn],#imm).
- **STR (Store)** : Opération d'écriture, suit une utilisation similaire que que LD, mais la destination est une adresse. Par exemple STR Rd,[Rn] copiera la valeur de Rd à l'adresse qui est stockée dans Rn.
- **CPY :** equivalent à MOV.

Copie de données

- **MOV (Move)** : Opération de copie d'une valeur dans un registre. Il peut provenir :
  - D'un autre registre (MOV Rd,Rm).
  - D'une valeur immédiate (MOV Rd, imm).
- **LDR (Load)** : Opération de lecture en mémoire dont la destination est un registre. Il est composé d'une adresse de base, et d'un offset qui sera ajouté à cette adresse pour calculer l'adresse de lecture. L'adresse de base est en principe contenu dans un registre (obligatoire chez ARM), et pour l'offset, il existe différentes méthodes :
  - **Immédiate :** l'offset est une constante codée dans l'instruction (LDR Rt,[Rn,#imm])
  - **Registre :** un autre registre est utilisé comme offset (LDR Rt,[Rn,Rm]). Celui-ci peut subir un décalage avant l'ajout au registre de base (LDR Rt,[Rn,Rm,LSL #imm])
  - **Pré-indexé :** L'adresse finale est calculée, le registre de base est mis-à-jour avec cette valeur, puis l'opération de lecture est effectuée (LDR Rt,[Rn,#imm]!).
  - **Post-indexé :** L'opération de lecture est effectuée, puis le registre de base est mis-à-jour en y ajoutant l'offset (LDR Rt,[Rn],#imm).
- **STR (Store)** : Opération d'écriture, suit une utilisation similaire que que LD, mais la destination est une adresse. Par exemple STR Rd,[Rn] copiera la valeur de Rd à l'adresse qui est stockée dans Rn.
- **CPY :** equivalent à MOV.

Copie de données

- **MOV (Move)** : Opération de copie d'une valeur dans un registre. Il peut provenir :
  - D'un autre registre (MOV Rd,Rm).
  - D'une valeur immédiate (MOV Rd, imm).
- **LDR (Load)** : Opération de lecture en mémoire dont la destination est un registre. Il est composé d'une adresse de base, et d'un offset qui sera ajouté à cette adresse pour calculer l'adresse de lecture. L'adresse de base est en principe contenu dans un registre (obligatoire chez ARM), et pour l'offset, il existe différentes méthodes :
  - **Immédiate :** l'offset est une constante codée dans l'instruction (LDR Rt,[Rn,#imm])
  - **Registre :** un autre registre est utilisé comme offset (LDR Rt,[Rn,Rm]). Celui-ci peut subir un décalage avant l'ajout au registre de base (LDR Rt,[Rn,Rm,LSL #imm])
  - **Pré-indexé :** L'adresse finale est calculée, le registre de base est mis-à-jour avec cette valeur, puis l'opération de lecture est effectuée (LDR Rt,[Rn,#imm]!).
  - **Post-indexé :** L'opération de lecture est effectuée, puis le registre de base est mis-à-jour en y ajoutant l'offset (LDR Rt,[Rn],#imm).
- **STR (Store)** : Opération d'écriture, suit une utilisation similaire que que LD, mais la destination est une adresse. Par exemple STR Rd,[Rn] copiera la valeur de Rd à l'adresse qui est stockée dans Rn.
- **CPY :** equivalent à MOV.

Arithmétique

- **ADD (Addition)** : Opération d'addition entière dont la destination est un registre. Le code-condition de retenue est mis à jour après calcul. Les opérandes peuvent être de différente natures :
  - **Deux registres :** les registres sont additionnés et le resultat est copié dans le registres de destination (ADD Rd,Rn,Rm).
  - **Un registre et une valeur immédiate :** le registre est additionné avec une constante et le resultat est copié dans le registre de destination (ADD Rd,Rn,#imm).
  - **Une valeur immédiate :** dans ce cas il s'agit d'une opération d'accumulation (ADD Rn,#imm).
- **SUB (Subtraction)** : Opération de soustraction entière dont l'utilisation est similaire à l'addition. Rappel : La représentation des nombre signés se fait avec le **complément à deux**.
- **ADC (Add with Carry)** : Opération d'addition avec prise en compte du code-condition de retenue. Permet de faire des additions sur des entiers dont la taille est supérieure à un registre.
- **SBC (Sub with Carry)** : Opération de soustraction avec prise en compte du code-condition de retenue. Permet de faire des soustractions sur des entiers dont la taille est supérieure à un registre.

Arithmétique

- **ADD (Addition)** : Opération d'addition entière dont la destination est un registre. Le code-condition de retenue est mis à jour après calcul. Les opérandes peuvent être de différente natures :
  - **Deux registres :** les registres sont additionnés et le resultat est copié dans le registres de destination (ADD Rd,Rn,Rm).
  - **Un registre et une valeur immédiate :** le registre est additionné avec une constante et le resultat est copié dans le registre de destination (ADD Rd,Rn,#imm).
  - **Une valeur immédiate :** dans ce cas il s'agit d'une opération d'accumulation (ADD Rn,#imm).
- **SUB (Subtraction)** : Opération de soustraction entière dont l'utilisation est similaire à l'addition. Rappel : La représentation des nombre signés se fait avec le **complément à deux**.
- **ADC (Add with Carry)** : Opération d'addition avec prise en compte du code-condition de retenue. Permet de faire des additions sur des entiers dont la taille est supérieure à un registre.
- **SBC (Sub with Carry)** : Opération de soustraction avec prise en compte du code-condition de retenue. Permet de faire des soustractions sur des entiers dont la taille est supérieure à un registre.

Arithmétique

- **ADD (Addition)** : Opération d'addition entière dont la destination est un registre. Le code-condition de retenue est mis à jour après calcul. Les opérandes peuvent être de différente natures :
  - **Deux registres :** les registres sont additionnés et le resultat est copié dans le registres de destination (ADD Rd,Rn,Rm).
  - **Un registre et une valeur immédiate :** le registre est additionné avec une constante et le resultat est copié dans le registre de destination (ADD Rd,Rn,#imm).
  - **Une valeur immédiate :** dans ce cas il s'agit d'une opération d'accumulation (ADD Rn,#imm).
- **SUB (Subtraction)** : Opération de soustraction entière dont l'utilisation est similaire à l'addition. Rappel : La représentation des nombre signés se fait avec le **complément à deux**.
- **ADC (Add with Carry)** : Opération d'addition avec prise en compte du code-condition de retenue. Permet de faire des additions sur des entiers dont la taille est supérieure à un registre.
- **SBC (Sub with Carry)** : Opération de soustraction avec prise en compte du code-condition de retenue. Permet de faire des soustractions sur des entiers dont la taille est supérieure à un registre.

Arithmétique

- **ADD (Addition)** : Opération d'addition entière dont la destination est un registre. Le code-condition de retenue est mis à jour après calcul. Les opérandes peuvent être de différente natures :
  - **Deux registres :** les registres sont additionnés et le resultat est copié dans le registres de destination (ADD Rd,Rn,Rm).
  - **Un registre et une valeur immédiate :** le registre est additionné avec une constante et le resultat est copié dans le registre de destination (ADD Rd,Rn,#imm).
  - **Une valeur immédiate :** dans ce cas il s'agit d'une opération d'accumulation (ADD Rn,#imm).
- **SUB (Subtraction)** : Opération de soustraction entière dont l'utilisation est similaire à l'addition. Rappel : La représentation des nombre signés se fait avec le **complément à deux**.
- **ADC (Add with Carry)** : Opération d'addition avec prise en compte du code-condition de retenue. Permet de faire des additions sur des entiers dont la taille est supérieure à un registre.
- **SBC (Sub with Carry)** : Opération de soustraction avec prise en compte du code-condition de retenue. Permet de faire des soustractions sur des entiers dont la taille est supérieure à un registre.

Arithmétique

- **ADD (Addition)** : Opération d'addition entière dont la destination est un registre. Le code-condition de retenue est mis à jour après calcul. Les opérandes peuvent être de différente natures :
  - **Deux registres :** les registres sont additionnés et le resultat est copié dans le registres de destination (ADD Rd,Rn,Rm).
  - **Un registre et une valeur immédiate :** le registre est additionné avec une constante et le resultat est copié dans le registre de destination (ADD Rd,Rn,#imm).
  - **Une valeur immédiate :** dans ce cas il s'agit d'une opération d'accumulation (ADD Rn,#imm).
- **SUB (Subtraction)** : Opération de soustraction entière dont l'utilisation est similaire à l'addition. Rappel : La représentation des nombre signés se fait avec le **complément à deux**.
- **ADC (Add with Carry)** : Opération d'addition avec prise en compte du code-condition de retenue. Permet de faire des additions sur des entiers dont la taille est supérieure à un registre.
- **SBC (Sub with Carry)** : Opération de soustraction avec prise en compte du code-condition de retenue. Permet de faire des soustractions sur des entiers dont la taille est supérieure à un registre.

Arithmétique

- **ADD (Addition)** : Opération d'addition entière dont la destination est un registre. Le code-condition de retenue est mis à jour après calcul. Les opérandes peuvent être de différente natures :
  - **Deux registres :** les registres sont additionnés et le resultat est copié dans le registres de destination (ADD Rd,Rn,Rm).
  - **Un registre et une valeur immédiate :** le registre est additionné avec une constante et le resultat est copié dans le registre de destination (ADD Rd,Rn,#imm).
  - **Une valeur immédiate :** dans ce cas il s'agit d'une opération d'accumulation (ADD Rn,#imm).
- **SUB (Subtraction)** : Opération de soustraction entière dont l'utilisation est similaire à l'addition. Rappel : La représentation des nombre signés se fait avec le **complément à deux**.
- **ADC (Add with Carry)** : Opération d'addition avec prise en compte du code-condition de retenue. Permet de faire des additions sur des entiers dont la taille est supérieure à un registre.
- **SBC (Sub with Carry)** : Opération de soustraction avec prise en compte du code-condition de retenue. Permet de faire des soustractions sur des entiers dont la taille est supérieure à un registre.

Arithmétique

- **ADD (Addition)** : Opération d'addition entière dont la destination est un registre. Le code-condition de retenue est mis à jour après calcul. Les opérandes peuvent être de différente natures :
  - **Deux registres :** les registres sont additionnés et le resultat est copié dans le registres de destination (ADD Rd,Rn,Rm).
  - **Un registre et une valeur immédiate :** le registre est additionné avec une constante et le resultat est copié dans le registre de destination (ADD Rd,Rn,#imm).
  - **Une valeur immédiate :** dans ce cas il s'agit d'une opération d'accumulation (ADD Rn,#imm).
- **SUB (Subtraction)** : Opération de soustraction entière dont l'utilisation est similaire à l'addition. Rappel : La représentation des nombre signés se fait avec le **complément à deux**.
- **ADC (Add with Carry)** : Opération d'addition avec prise en compte du code-condition de retenue. Permet de faire des additions sur des entiers dont la taille est supérieure à un registre.
- **SBC (Sub with Carry)** : Opération de soustraction avec prise en compte du code-condition de retenue. Permet de faire des soustractions sur des entiers dont la taille est supérieure à un registre.

Arithmétique [2]

- **NEG (Negative)** : Opération d'inversion de signe.
- **CMP (Compare)** : Opération de comparaison de deux registres (équivalent à la soustraction sans destination) avec mise à jour du registre de contrôle.
- **MUL (Multiplication)** : Opération de multiplication entière entre les registres.
- **UDIV (Unsigned Division)** : Opération de division entière (quotient de la division euclidiène) sur les entiers.
- **SDIV (Multiplication)** : Opération de division entière signée (quotient de la division euclidiène) sur les entiers.

Arithmétique [2]

- **NEG (Negative)** : Opération d'inversion de signe.
- **CMP (Compare)** : Opération de comparaison de deux registres (équivalent à la soustraction sans destination) avec mise à jour du registre de contrôle.
- **MUL (Multiplication)** : Opération de multiplication entière entre les registres.
- **UDIV (Unsigned Division)** : Opération de division entière (quotient de la division euclidiène) sur les entiers.
- **SDIV (Multiplication)** : Opération de division entière signée (quotient de la division euclidiène) sur les entiers.

Arithmétique [2]

- **NEG (Negative)** : Opération d'inversion de signe.
- **CMP (Compare)** : Opération de comparaison de deux registres (équivalent à la soustraction sans destination) avec mise à jour du registre de contrôle.
- **MUL (Multiplication)** : Opération de multiplication entière entre les registres.
- **UDIV (Unsigned Division)** : Opération de division entière (quotient de la division euclidiène) sur les entiers.
- **SDIV (Multiplication)** : Opération de division entière signée (quotient de la division euclidiène) sur les entiers.

Arithmétique [2]

- **NEG (Negative)** : Opération d'inversion de signe.
- **CMP (Compare)** : Opération de comparaison de deux registres (équivalent à la soustraction sans destination) avec mise à jour du registre de contrôle.
- **MUL (Multiplication)** : Opération de multiplication entière entre les registres.
- **UDIV (Unsigned Division)** : Opération de division entière (quotient de la division euclidiène) sur les entiers.
- **SDIV (Multiplication)** : Opération de division entière signée (quotient de la division euclidiène) sur les entiers.

Arithmétique [2]

- **NEG (Negative)** : Opération d'inversion de signe.
- **CMP (Compare)** : Opération de comparaison de deux registres (équivalent à la soustraction sans destination) avec mise à jour du registre de contrôle.
- **MUL (Multiplication)** : Opération de multiplication entière entre les registres.
- **UDIV (Unsigned Division)** : Opération de division entière (quotient de la division euclidiène) sur les entiers.
- **SDIV (Multiplication)** : Opération de division entière signée (quotient de la division euclidiène) sur les entiers.

Logique

- **AND :** Opération ET bit-à-bit (AND Rd,Rn,Rm / AND Rdn,Rm).

op1	op2	res
0	0	0
0	1	0
1	0	0
1	1	1

- **OR :** Opération OU bit-à-bit (OR Rd,Rn,Rm / OR Rdn,Rm).

op1	op2	res
0	0	0
0	1	1
1	0	1
1	1	1

- **EOR :** Egalement appelé le XOR. Opération OU EXCLUSIF bit-à-bit (EOR Rd,Rn,Rm / EOR Rdn,Rm).

op1	op2	res
0	0	0
0	1	1
1	0	1
1	1	0

Logique

- **AND :** Opération ET bit-à-bit (AND Rd,Rn,Rm / AND Rdn,Rm).

op1	op2	res
0	0	0
0	1	0
1	0	0
1	1	1

- **OR :** Opération OU bit-à-bit (OR Rd,Rn,Rm / OR Rdn,Rm).

op1	op2	res
0	0	0
0	1	1
1	0	1
1	1	1

- **EOR :** Egalement appelé le XOR. Opération OU EXCLUSIF bit-à-bit (EOR Rd,Rn,Rm / EOR Rdn,Rm).

op1	op2	res
0	0	0
0	1	1
1	0	1
1	1	0

Logique

- **AND :** Opération ET bit-à-bit (AND Rd,Rn,Rm / AND Rdn,Rm).

op1	op2	res
0	0	0
0	1	0
1	0	0
1	1	1

- **OR :** Opération OU bit-à-bit (OR Rd,Rn,Rm / OR Rdn,Rm).

op1	op2	res
0	0	0
0	1	1
1	0	1
1	1	1

- **EOR :** Egalement appelé le XOR. Opération OU EXCLUSIF bit-à-bit (EOR Rd,Rn,Rm / EOR Rdn,Rm).

op1	op2	res
0	0	0
0	1	1
1	0	1
1	1	0

Décalage

- **LSL (Logical Shift Left) :** Opération de décalage à gauche "logique" (i.e. sans préserver le signe). Il existe plusieurs variantes :
  - **Immediat :** La valeur du décalage est codé dans l'instruction (LSL Rd,Rm,#imm).
  - **Registre :** La valeur du décalage est lue dans un registre (LSL Rd,Rm). Dans l'Armv7m, le décalage est déterminé par l'octet de poids faible de Rm uniquement.
- **LSR (Logical Shift Right) :** Opération de décalage à droite "logique" (i.e. sans préserver le signe). Fonctionnement similaire à LSL.
- **ASL (Arithmétique Shift Left) :** Opération de décalage à gauche "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.
- **ASR (Arithmétique Shift Right) :** Opération de décalage à droite "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.

Décalage

- **LSL (Logical Shift Left) :** Opération de décalage à gauche "logique" (i.e. sans préserver le signe). Il existe plusieurs variantes :
  - **Immediat :** La valeur du décalage est codé dans l'instruction (LSL Rd,Rm,#imm).
  - **Registre :** La valeur du décalage est lue dans un registre (LSL Rd,Rm). Dans l'Armv7m, le décalage est déterminé par l'octet de poids faible de Rm uniquement.
- **LSR (Logical Shift Right) :** Opération de décalage à droite "logique" (i.e. sans préserver le signe). Fonctionnement similaire à LSL.
- **ASL (Arithmétique Shift Left) :** Opération de décalage à gauche "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.
- **ASR (Arithmétique Shift Right) :** Opération de décalage à droite "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.

Décalage

- **LSL (Logical Shift Left) :** Opération de décalage à gauche "logique" (i.e. sans préserver le signe). Il existe plusieurs variantes :
  - **Immediat :** La valeur du décalage est codé dans l'instruction (LSL Rd,Rm,#imm).
  - **Registre :** La valeur du décalage est lue dans un registre (LSL Rd,Rm). Dans l'Armv7m, le décalage est déterminé par l'octet de poids faible de Rm uniquement.
- **LSR (Logical Shift Right) :** Opération de décalage à droite "logique" (i.e. sans préserver le signe). Fonctionnement similaire à LSL.
- **ASL (Arithmétique Shift Left) :** Opération de décalage à gauche "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.
- **ASR (Arithmétique Shift Right) :** Opération de décalage à droite "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.

Décalage

- **LSL (Logical Shift Left) :** Opération de décalage à gauche "logique" (i.e. sans préserver le signe). Il existe plusieurs variantes :
  - **Immediat :** La valeur du décalage est codé dans l'instruction (LSL Rd,Rm,#imm).
  - **Registre :** La valeur du décalage est lue dans un registre (LSL Rd,Rm). Dans l'Armv7m, le décalage est déterminé par l'octet de poids faible de Rm uniquement.
- **LSR (Logical Shift Right) :** Opération de décalage à droite "logique" (i.e. sans préserver le signe). Fonctionnement similaire à LSL.
- **ASL (Arithmétique Shift Left) :** Opération de décalage à gauche "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.
- **ASR (Arithmétique Shift Right) :** Opération de décalage à droite "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.

Décalage

- **LSL (Logical Shift Left) :** Opération de décalage à gauche "logique" (i.e. sans préserver le signe). Il existe plusieurs variantes :
  - **Immediat :** La valeur du décalage est codé dans l'instruction (LSL Rd,Rm,#imm).
  - **Registre :** La valeur du décalage est lue dans un registre (LSL Rd,Rm). Dans l'Armv7m, le décalage est déterminé par l'octet de poids faible de Rm uniquement.
- **LSR (Logical Shift Right) :** Opération de décalage à droite "logique" (i.e. sans préserver le signe). Fonctionnement similaire à LSL.
- **ASL (Arithmétique Shift Left) :** Opération de décalage à gauche "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.
- **ASR (Arithmétique Shift Right) :** Opération de décalage à droite "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.

Décalage

- **LSL (Logical Shift Left) :** Opération de décalage à gauche "logique" (i.e. sans préserver le signe). Il existe plusieurs variantes :
  - **Immediat :** La valeur du décalage est codé dans l'instruction (LSL Rd,Rm,#imm).
  - **Registre :** La valeur du décalage est lue dans un registre (LSL Rd,Rm). Dans l'Armv7m, le décalage est déterminé par l'octet de poids faible de Rm uniquement.
- **LSR (Logical Shift Right) :** Opération de décalage à droite "logique" (i.e. sans préserver le signe). Fonctionnement similaire à LSL.
- **ASL (Arithmétique Shift Left) :** Opération de décalage à gauche "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.
- **ASR (Arithmétique Shift Right) :** Opération de décalage à droite "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.

Contrôle de flux d'éxecution (control transfer instruction)

- **B (Branch) :** Opération de saut programme à une adresse définie par un label (B Label)
- **BL (Branch with Link) :** Opération d'appel à sous-programme défini par un label (BL Label). Sauvegarde de l'adresse de retour dans LR.
- **BX (Branch and Exchange) :** Opération de retour de sous programme à partir de la valeur d'un registre (BL Rm).

Contrôle de flux d'éxecution (control transfer instruction)

- **B (Branch) :** Opération de saut programme à une adresse définie par un label (B Label)
- **BL (Branch with Link) :** Opération d'appel à sous-programme défini par un label (BL Label). Sauvegarde de l'adresse de retour dans LR.
- **BX (Branch and Exchange) :** Opération de retour de sous programme à partir de la valeur d'un registre (BL Rm).

Contrôle de flux d'éxecution (control transfer instruction)

- **B (Branch) :** Opération de saut programme à une adresse définie par un label (B Label)
- **BL (Branch with Link) :** Opération d'appel à sous-programme défini par un label (BL Label). Sauvegarde de l'adresse de retour dans LR.
- **BX (Branch and Exchange) :** Opération de retour de sous programme à partir de la valeur d'un registre (BL Rm).

Accès à la pile

- **PUSH :** Opération d'empilement d'un registre dans la pile. Différentes méthodes existes :
  - Un seul registre :PUSH {Ra}
  - Plusieurs registres :PUSH {Ra,Rb,Rc}
- **POP :** Opération de dépilement d'un registre de la pile. Utilisation similaire à PUSH.

Accès à la pile

- **PUSH :** Opération d'empilement d'un registre dans la pile. Différentes méthodes existes :
  - Un seul registre :PUSH {Ra}
  - Plusieurs registres :PUSH {Ra,Rb,Rc}
- **POP :** Opération de dépilement d'un registre de la pile. Utilisation similaire à PUSH.

Accès à la pile

- **PUSH :** Opération d'empilement d'un registre dans la pile. Différentes méthodes existes :
  - Un seul registre :PUSH {Ra}
  - Plusieurs registres :PUSH {Ra,Rb,Rc}
- **POP :** Opération de dépilement d'un registre de la pile. Utilisation similaire à PUSH.

Accès à la pile

- **PUSH :** Opération d'empilement d'un registre dans la pile. Différentes méthodes existes :
  - Un seul registre :PUSH {Ra}
  - Plusieurs registres :PUSH {Ra,Rb,Rc}
- **POP :** Opération de dépilement d'un registre de la pile. Utilisation similaire à PUSH.

Jeu d'instructions - Tableau comparatif

Opération	Arm Thumb2	x86 - Intel (Microsoft)	x86 - AT&T (Linux)
Copie registre	mov r0, r1 ←	mov eax, ebx ←	mov %ebx, %eax →
Adr immédiat	mov r0, #0x1C ←	mov eax, 0x1C ←	mov $0x1C, %eax →
Load indirect	ldr r0, [r2] ←	mov eax, [eci] ←	mov (%eci), %eax →
Store indirect	str r0, [r2] →	mov [ecx], eax ←	mov %eax,(%ecx) →
Load indirect + offset	ldr r0, [r2,0x18] ←	mov eax, [ecx+0x18] ←	mov 0x18(%ecx), %eax →
Store indirect + offset	str r0, [r2,0x1C] →	mov [ecx+0x1C], eax ←	mov %eax,0x1C(%ecx) →

Opération	Arm Thumb2	x86 - Intel (Microsoft)	x86 - AT&T (Linux)
Load indexé	ldr r0, [r2,r3,LSL #2] ←	mov eax, [ecx+edx*4] ←	mov (%ecx,%edx,4), %eax →
Store indexé	str r0, [r2,r3,LSL #2] →	mov [ecx+edx*4], eax ←	mov %eax,(%ecx,%edx,4) →
Base + Index + offset		mov eax, [ecx+edx*8+0x24] ←	mov 0x24(%ecx,%edx,8), %eax →
Load effective address		lea eax, [ecx+edx*8+0x24] ←	lea 0x24(%ecx,%edx,8), %eax →

Remarque sur load effective address

"Load Effective Address" permet de calculer l'adresse finale, mais stocke le résultat dans un registre plutot que de faire un accès mémoire. Conçu pour gérer le & en langage C.

Opération	Arm Thumb2	x86 - Intel (Microsoft)	x86 - AT&T (Linux)
Load indexé	ldr r0, [r2,r3,LSL #2] ←	mov eax, [ecx+edx*4] ←	mov (%ecx,%edx,4), %eax →
Store indexé	str r0, [r2,r3,LSL #2] →	mov [ecx+edx*4], eax ←	mov %eax,(%ecx,%edx,4) →
Base + Index + offset		mov eax, [ecx+edx*8+0x24] ←	mov 0x24(%ecx,%edx,8), %eax →
Load effective address		lea eax, [ecx+edx*8+0x24] ←	lea 0x24(%ecx,%edx,8), %eax →

Remarque sur load effective address

Elle peut être utilisée astucieusement pour faire des calculs n'ayant rien à voir avec le calcul d'adresse, exemple que fait (%eax,%eax,4),%eax ?

Opération	Arm Thumb2	x86 - Intel (Microsoft)	x86 - AT&T (Linux)
Load indexé	ldr r0, [r2,r3,LSL #2] ←	mov eax, [ecx+edx*4] ←	mov (%ecx,%edx,4), %eax →
Store indexé	str r0, [r2,r3,LSL #2] →	mov [ecx+edx*4], eax ←	mov %eax,(%ecx,%edx,4) →
Base + Index + offset		mov eax, [ecx+edx*8+0x24] ←	mov 0x24(%ecx,%edx,8), %eax →
Load effective address		lea eax, [ecx+edx*8+0x24] ←	lea 0x24(%ecx,%edx,8), %eax →

Remarque sur load effective address

Elle peut être utilisée astucieusement pour faire des calculs n'ayant rien à voir avec le calcul d'adresse, exemple que fait (%eax,%eax,4),%eax ? Il multiplie le registre eax par 5.

Jeu d'instructions - Tableau comparatif

Opération	Arm Thumb2	x86 - Intel (Microsoft)	x86 - AT&T (Linux)
Appel de sous-programme	bl Label	call label	call label
Retour de sous-programme	bx LR (feuille) pop {PC} (branche)	ret	ret
Retour de sous-programme avec incrément du pointeur de pile	 POP {PC} (branche)	ret 0x1C ret	ret 0x1C ret
Saut inconditionnel	b Label	jmp Label	jmp Label
Saut conditionnel (exemple : Z=1)	beq Label	je Label	je Label

Construction d'un programme : La chaîne de compilation

Principe

La chaîne de compilation (ou toolchain en anglais), est l'ensemble des procédures permettant le construction d'un fichier exécutable pour un processeur donné à partir de fichiers sources.

Chaîne de compilation GNU

La chaîne de compilation GNU est un ensemble d'outils libres permettant la compilation de fichier décrits en langage C/C++ et d'assemblage. Elle est composée de :

- **binutils :** suite d'outils permettant de manipuler les fichiers binaires. En particulier, contient un assembleur et un éditeur de lien.
- **gcc :** compilateur pour langage C/C++.
- **glibc :** bibliothèque permettant de gérer les appels systèmes ainsi que la gestion de fonctionnalités requises par certaines normes logicielles (POSIX, ISO C99, ...).
- **gdb :** debugger de fichiers binaires compilés avec GNU.
- **make :** interpréteur de scripts qui facilite l'automatisation des séquences de construction du projet (build).

Remarque

En principe, une chaîne de compilation est requise par architecture cible. En pratique, les chaînes de compilations peuvent gérer plusieurs architectures mais il faut leur fournir des directives au moment de la compilation (exemple : Arm fourni une unique chaîne de compilation GNU pour les Cortex-M, R et A).

Principe

La compilation croisée permet d'utiliser une chaîne de compilation pour un processeur particulier sur une autre machine (dite hôte). Cela permet par exemple de compiler pour une cible Arm sur votre ordinateur personnel.

Avantages

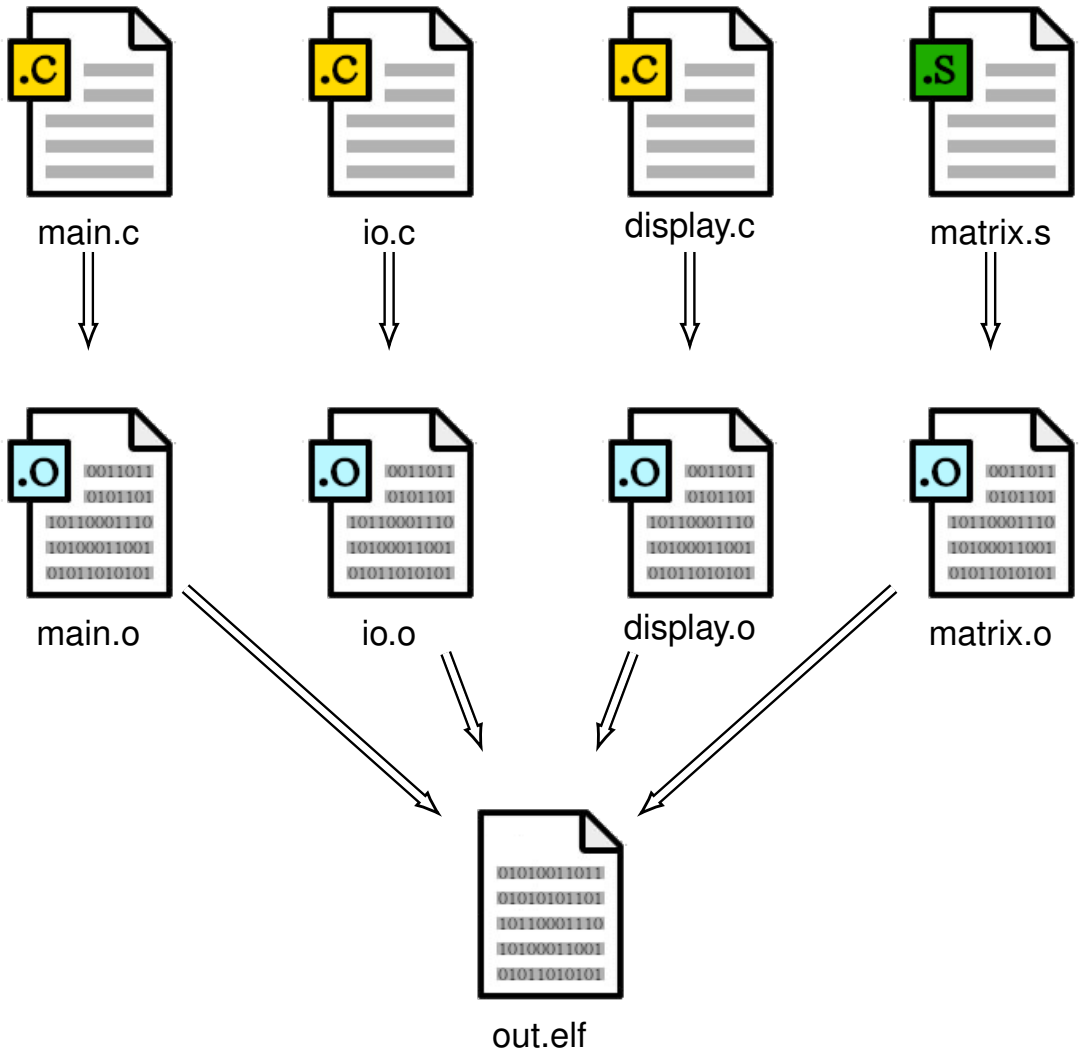
- Vous bénéficiez des performances de la machine hôte pour le processus de compilation.
- Vous bénéficiez d'un environnement de debug généralement plus puissant que s'il était réalisé sur la machine cible (interface graphique, utilisation d'outils complémentaires performants, ...).

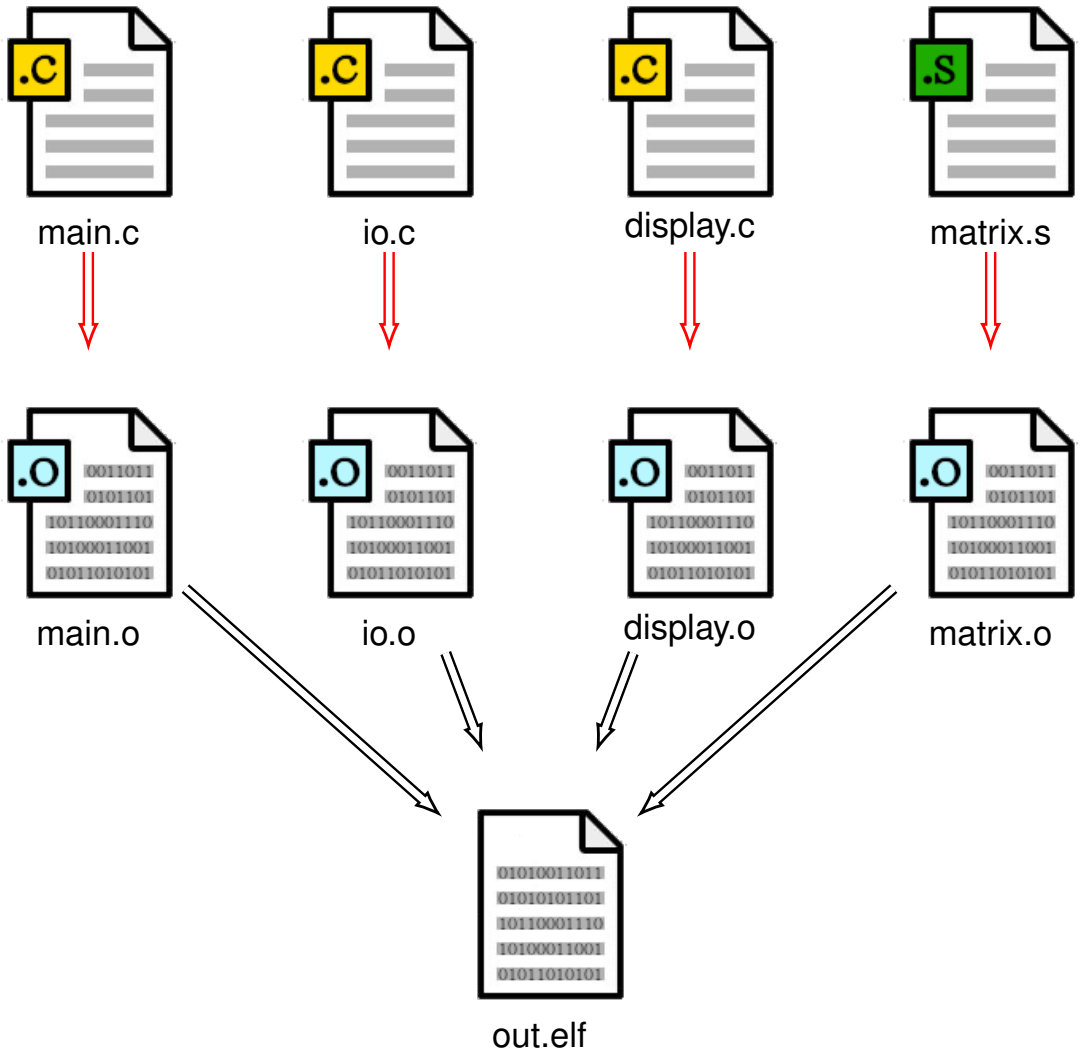
Points de vigilance

Le code exécutable généré par la compilation croisée n'est pas exécutable par votre machine. Deux stratégies de test existent :

- Utiliser une cible compatible avec votre binaire et l'interfacer avec votre machine pour le test.
- Utiliser un simulateur d'architecture compatible avec votre binaire.

Exemple d'une chaîne de compilation



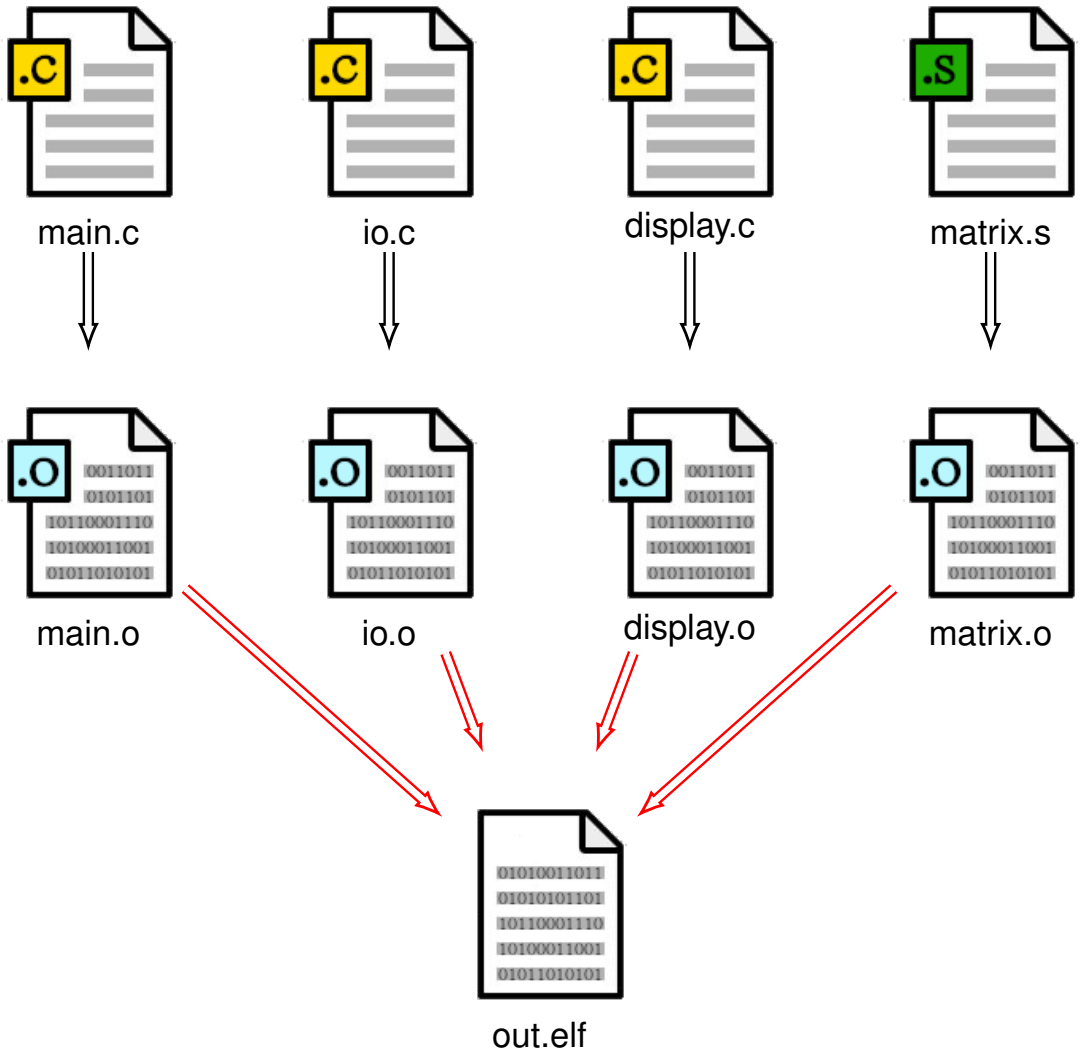


Génération des fichiers objets

L'assembleur et le compilateur traduisent le fichier source en un fichier intermédiaire dit objet contenant du code en langage machine. Il possède en plus :

- Une table de symboles pour exporter les adresses des fonctions utilisables par d'autres fonctions ainsi que les variables statiques (export).
- Une table de symboles non résolus pour les appels de fonctions décrits dans un autre fichier ainsi que les variables statiques (import).

Exemple d'une chaîne de compilation

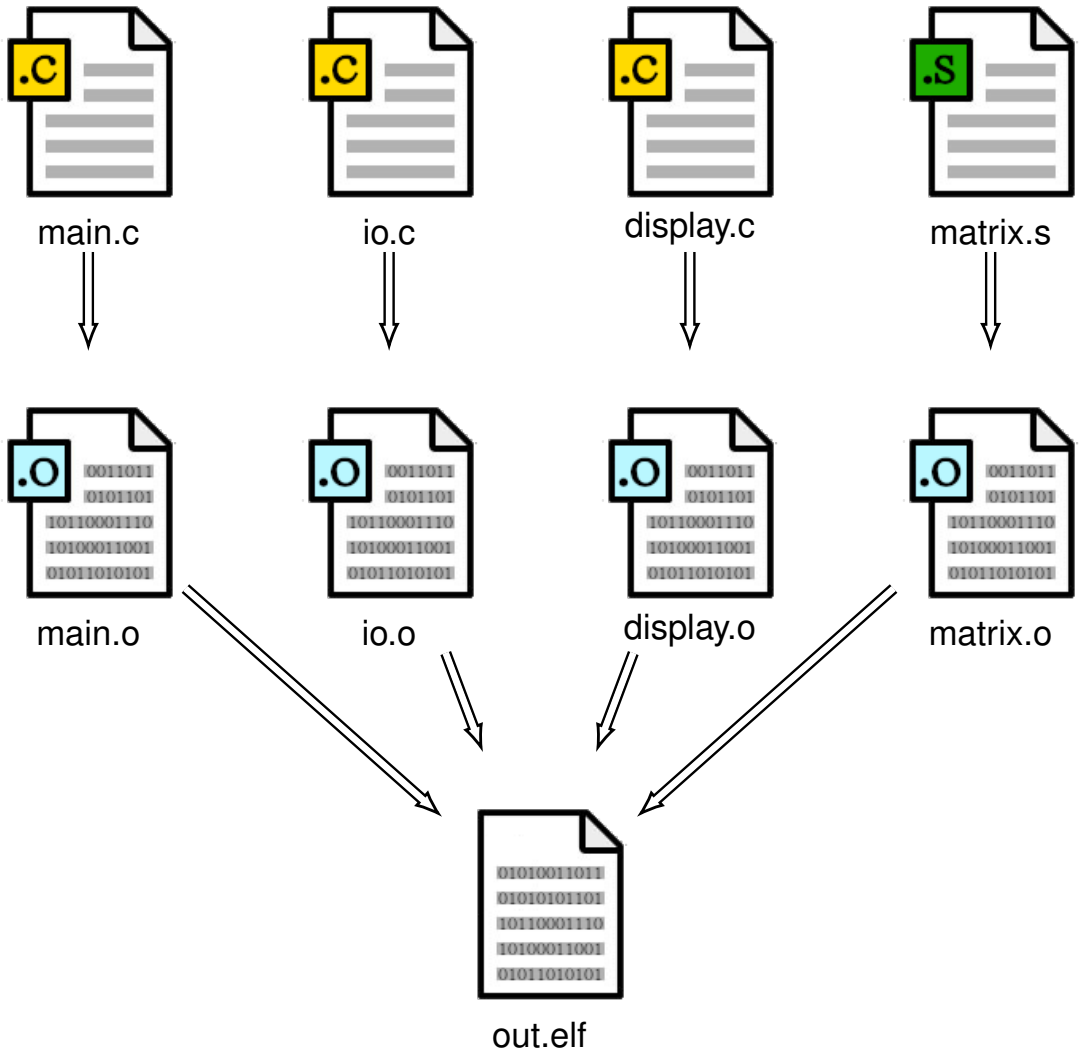


L'édition des liens

Construction du binaire à partir des fichiers objets. Il permet :

- D'attribuer des plages d'adresse aux codes et données des différents fichiers objets.
- Concaténer les objets.
- Mettre à jour les symboles encore non résolus maintenant que les adresses des fonctions sont connues.

Exemple d'une chaîne de compilation

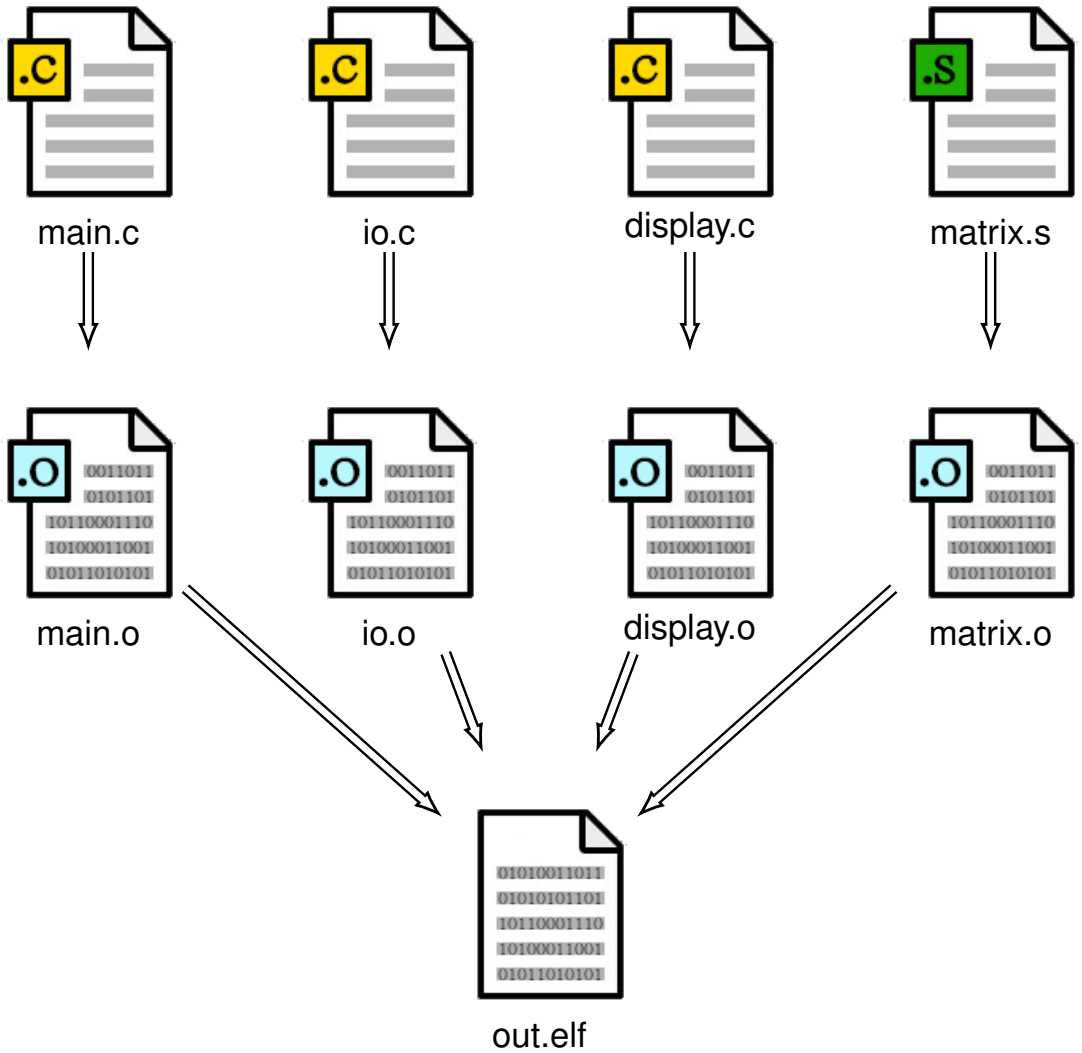


L'édition des liens C/ASM - export

Adresses exportées sont mises à la disposition des autres objets du projet :

- En langage C : Par défaut, toutes les fonctions et les variables statiques globales sont exportées (attribut **static** pour l'éviter).
- En langage d'assemblage : Pas d'export automatique (directive à ajouter, différent selon l'assembleur : **export**, **global**)

Exemple d'une chaîne de compilation

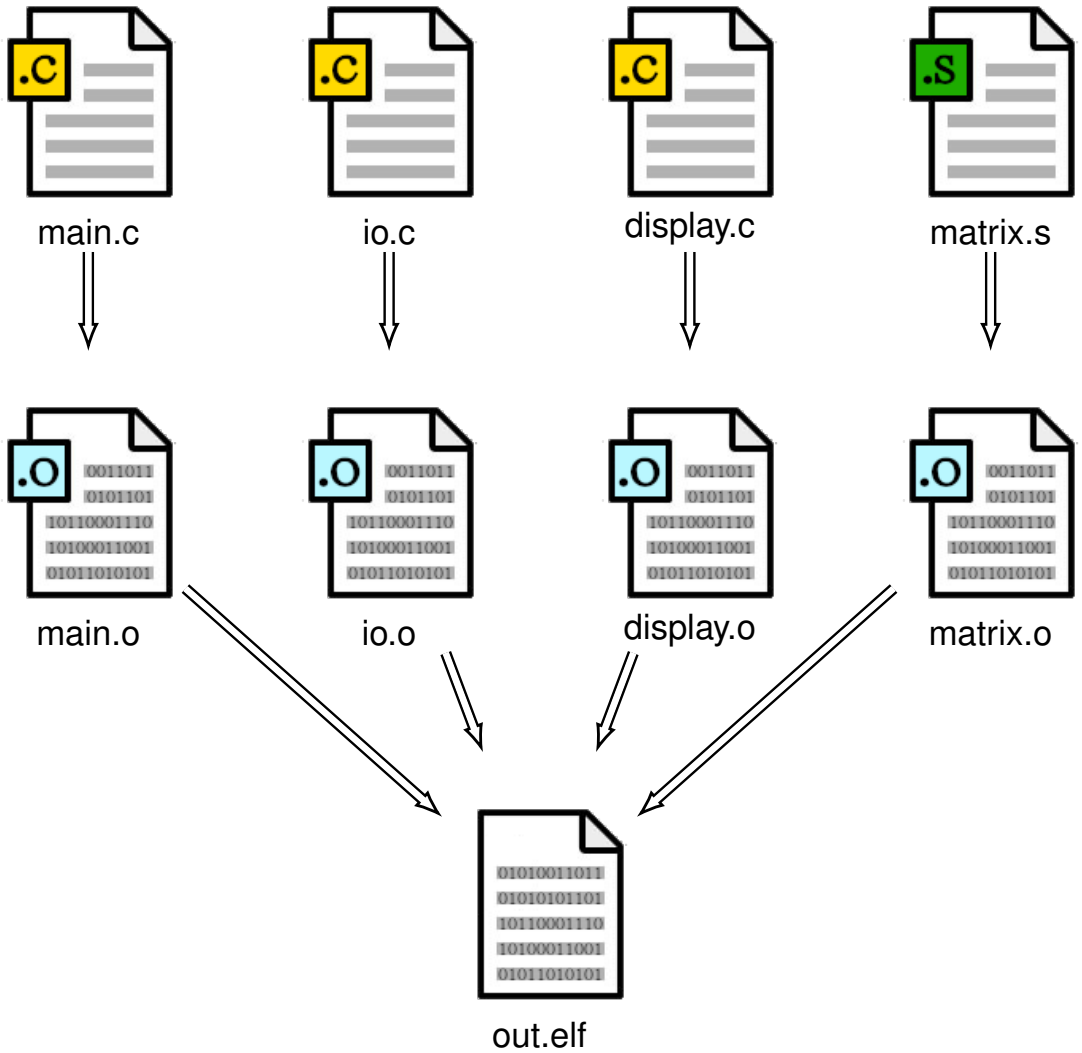


L'édition des liens C/ASM - import

Le compilateur et l'assembleur peuvent accepter des "symboles non résolus" c'est à dire des références à des adresses inconnues (externes) :

- En langage C : Les fonctions doivent être déclarées, la déclaration précise le type de valeur retournée et de chaque argument. Les variables doivent être déclarées avec l'attribut **extern**.
- En langage d'assemblage : certains assembleurs exigent l'utilisation d'une directive **import**, d'autres acceptent tous les symboles non résolus sans déclaration.

Exemple d'une chaîne de compilation



Notes sur Import-Export

- Ne pas confondre symboles pour import-export et constantes manipulées par le pré-processeur du C (#define). Les #define ne sont pas propagés dans le code objet.
- Il est recommandé d'exporter le moins possible de symboles, pour réduire le risque de conflits de noms au linkage (en C, utiliser static aussi souvent que possible)
- Contrairement au compilateur C, l'assembleur ne peut faire aucune vérification du type de retour et des arguments d'une fonction.
- L'appel en assembleur d'une fonction écrite en C ou C++ implique un risque rarement justifié.

Principe

Certains compilateurs C supportent l'insertion de portions de code en langage d'assemblage au milieu d'un texte source en C. Ceci peut de faire :

- au niveau instruction : inline assembly
- au niveau fonction : embedded assembly

Avantages

- possibilité d'insérer des instructions spéciales que le compilateur C ne sait pas manipuler
- possibilité de manipuler directement les variables du C

Inconvénients

- syntaxe très variable d'un outil à l'autre
- utilisation difficile en dehors des cas simples

Inline assembly

Permet d'introduire du code en langage d'assemblage à l'intérieur de fonctions c. Il faut définir les variables/constantes d'entrées et de sortie pour s'interfacer au code c et leur type (adresse, registre, ...). Ci-dessous un exemple pour GCC.

```
int main() {
    int a,b,c;
    asm (
    "mul R1,%1,%2 \n\t" // R1 <- a x b
    "add %0,R1    \n\t" // c += R1
    : "=r" (c)          // variables de sortie (variable c)
    : "r" (a)           // variable d'entrée (variable a)
    , "r" (b)           // variable d'entrée (variable b)
    : "R1"              // registres utilisés
    );
}
```

Embedded assembly

Permet d'introduire du code en langage d'assemblage en tant que fonction. Ce code doit respecter l'appel de fonctions propre à l'architecture. Ci-dessous un exemple avec Keil (Realview).

```
// multiplication 32 bits X 32 bits -> 64 bits
__asm long long smull_func( int a, int b )
{
   smull r0, r1, r0, r1
   bx lr
}

int main (void)
{
   static long long r; // 64 bits
   r = smull_func( 543210, 1000000001 );
   return 0;
}
```

Keil micro-vision (Windows only)

Environnement de développement de Arm pour micro-contrôleur Arm possédant une chaîne de compilation croisée, un simulateur ainsi qu'une interface pour le debug de cibles matérielles.

Alternatives libres

Il est possible également de reposer sur des outils libres permettant notamment le debug :

- GDB : Outil de debug de code compilé via la chaîne de compilation GNU.
- QEMU : Outil de simulation et de virtualisation de machine particulièrement puissant, prenant en charge un grand nombre d'architectures différentes et est même capable de émuler des systèmes exploitation. qemu peut s'interfacer directement avec GDB pour le debug.
- OpenOCD : Outil permettant la traduction de commandes GDB vers des outils de debug de cibles matérielles (exemple STM32F).

Définition

Le linker script (ou scatter file) est un fichier permettant de donner des directives pour l'édition des liens (linker) afin de positionner les codes compilés dans la mémoire.

```
MEMORY
{
    rom : ORIGIN = 0x00000000, LENGTH = 0x1000
    ram : ORIGIN = 0x20000000, LENGTH = 0x1000
}

SECTIONS
{
    .text : { *(.text*) } > rom
    .rodata : { *(.rodata*) } > rom
    .bss : { *(.bss*) } > ram
}
```

MEMORY

Définit l'adresse et la taille des blocs mémoires de la cible.

```
MEMORY
{
    rom : ORIGIN = 0x00000000, LENGTH = 0x1000
    ram : ORIGIN = 0x20000000, LENGTH = 0x1000
}

SECTIONS
{
    .text : { *(.text*) } > rom
    .rodata : { *(.rodata*) } > rom
    .bss : { *(.bss*) } > ram
}
```

SECTIONS

Une section est un espace mémoire continu possédant des droits d'accès fixes. Sections classiques :

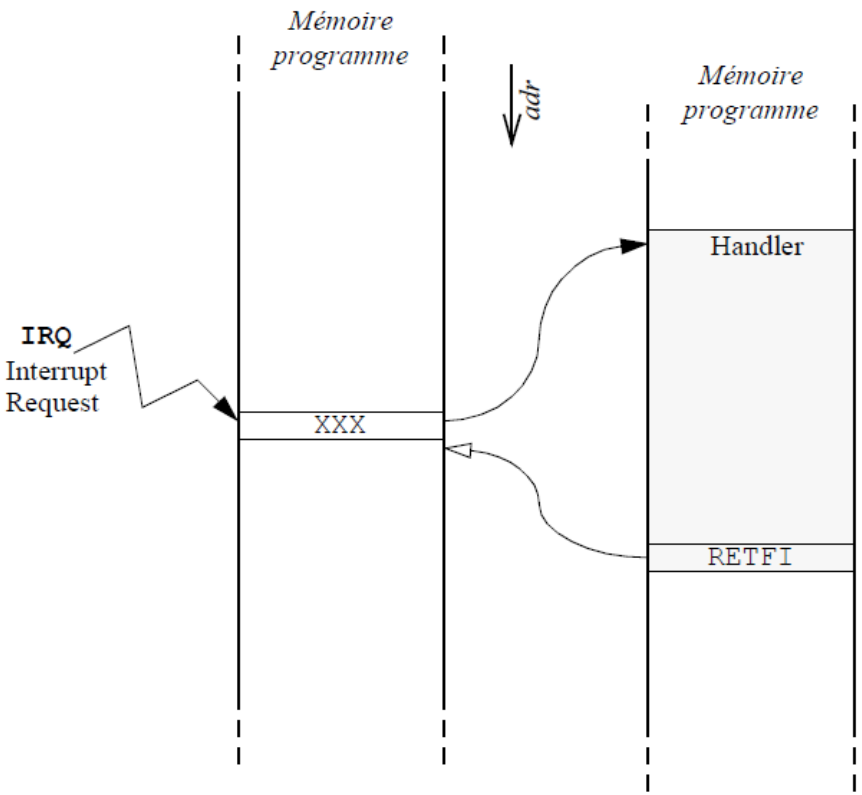
.text	code du programme	.bss	variables allouées statiquement
.data	variables globales et constantes locales globales	.rodata	constantes globales

```
MEMORY
{
    rom : ORIGIN = 0x00000000, LENGTH = 0x1000
    ram : ORIGIN = 0x20000000, LENGTH = 0x1000
}

SECTIONS
{
    .text : { *(.text*) } > rom
    .rodata : { *(.rodata*) } > rom
    .bss : { *(.bss*) } > ram
}
```

Remarque sur les interruptions et cas particulier de ARM

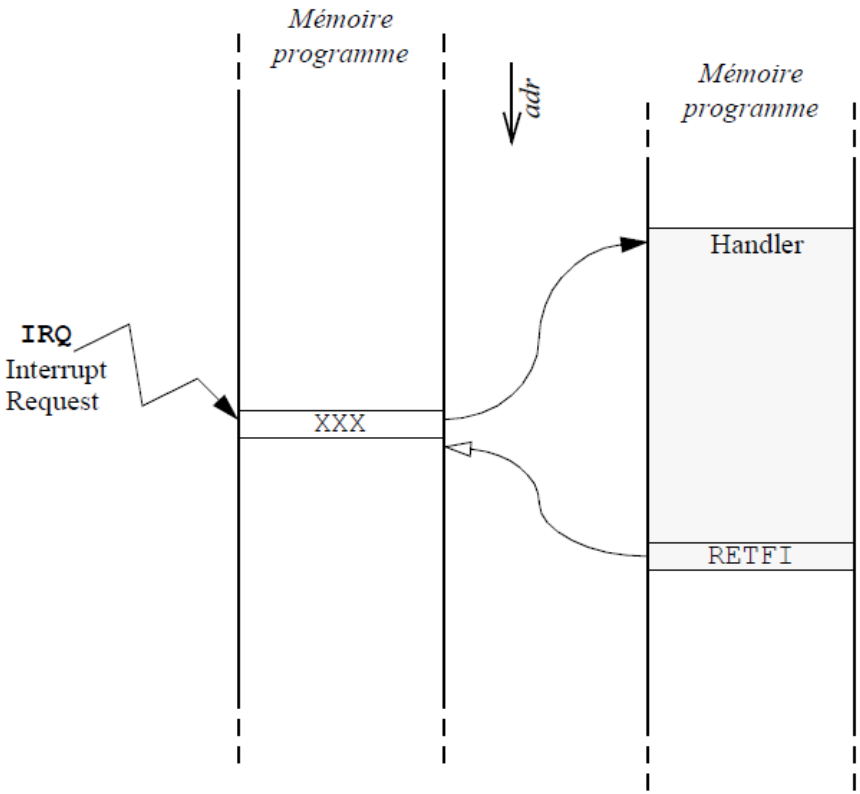
Rappel sur les interruptions



Principe

- Une interruption est un événement matériel permettant à un périphérique de forcer l'exécution d'un code par le processeur.
- Ce code est appelé routine d'interruption (**interrupt routine** ou **interrupt handler**) dont le rôle est de traiter la requête du périphérique.
- Le processeur peut lui-aussi générer une interruption pour lui-même (interruption logicielle).

Rappel sur les interruptions

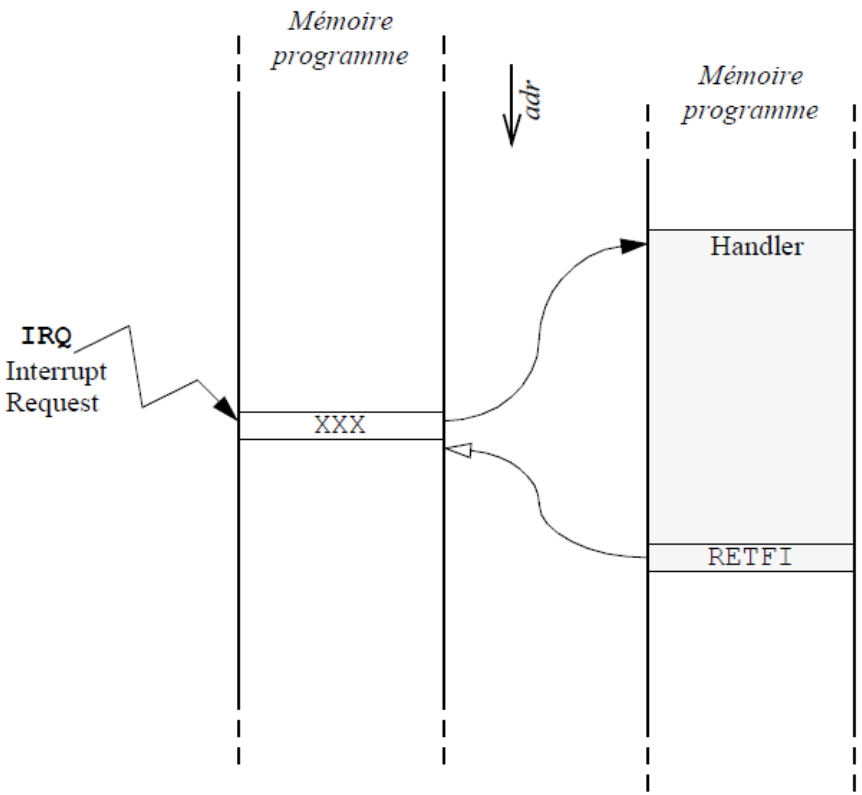


Principe

- Une interruption est un événement matériel permettant à un périphérique de forcer l'exécution d'un code par le processeur.
- Ce code est appelé routine d'interruption (**interrupt routine** ou **interrupt handler**) dont le rôle est de traiter la requête du périphérique.
- Le processeur peut lui-aussi générer une interruption pour lui-même (interruption logicielle).

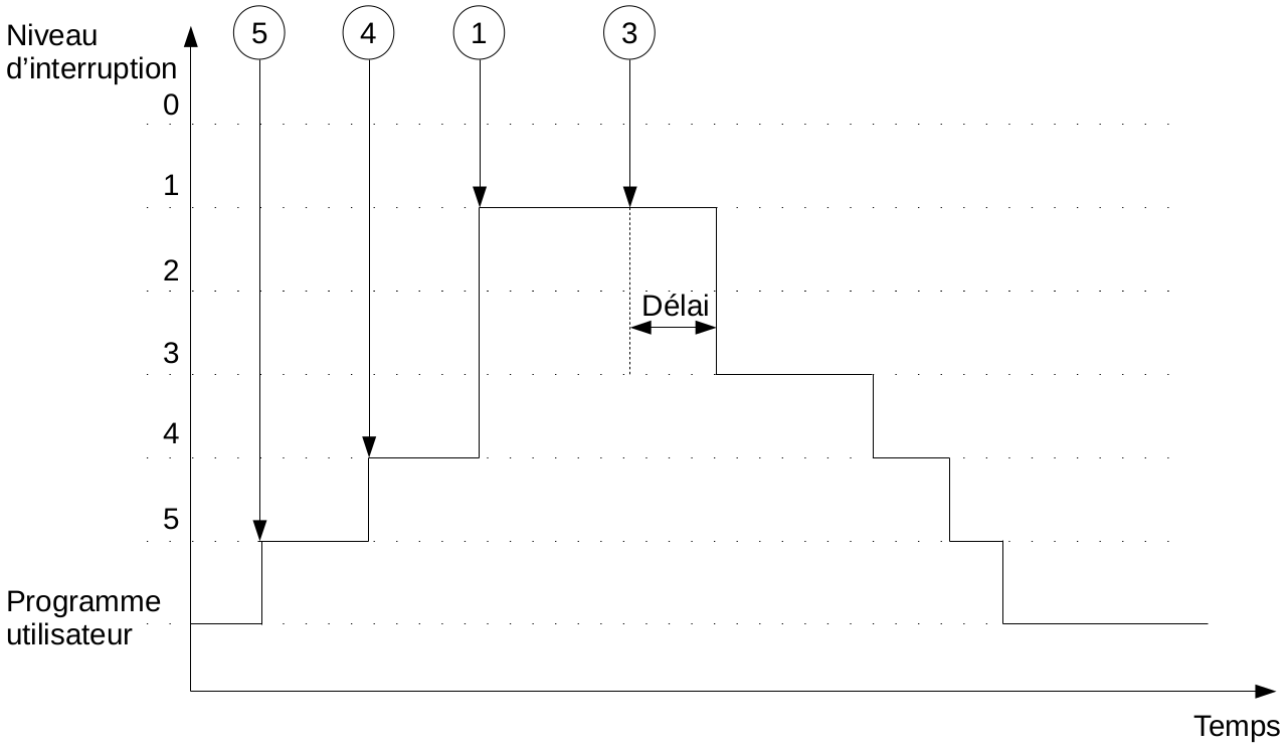
Exemples

- Carte réseau demandant une lecture d'un packet réseau stocké dans sa mémoire interne.
- Exécution du code noyau depuis l'espace utilisateur (interruption logicielle).
- Gestion du multi-tâche en stoppant l'exécution du processus courant toutes les X millisecondes (interruption via compteurs matériels).
- Réveil d'un processeur mis en sommeil pour réduire la consommation énergétique.



Vectorisation

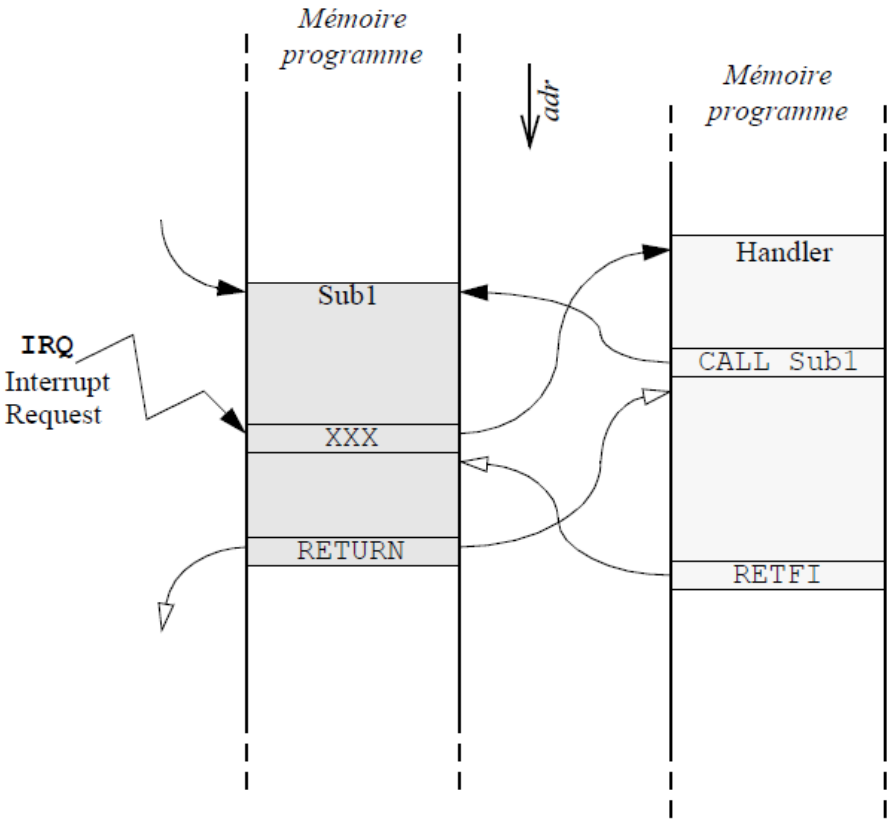
Pour appliquer un traitement specifique pour chaque source d'interruption, la solution la plus efficace est une table des points d'entrées (**vecteurs**) des handlers, souvent installée à une adresse fixe.



Niveaux d'interruption

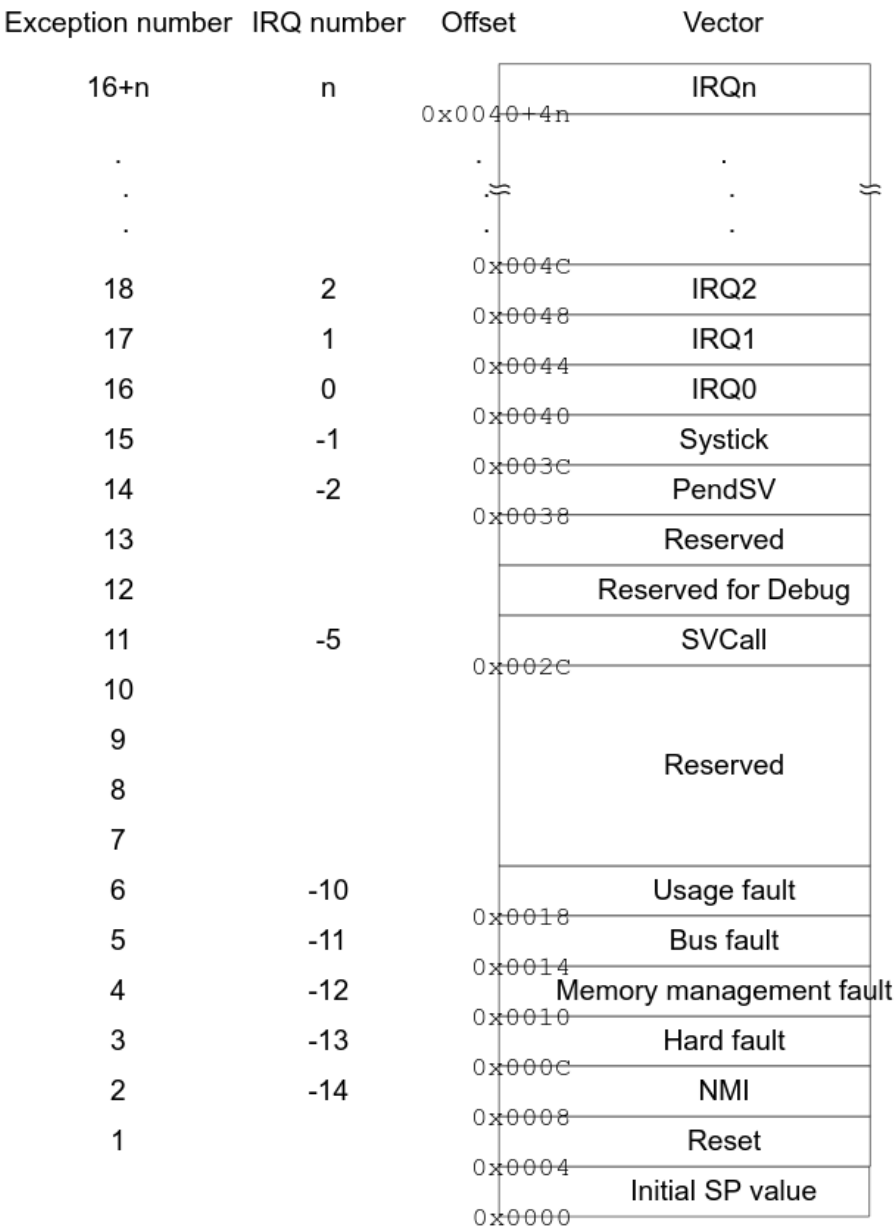
Des niveaux de priorité peuvent être attribués aux différentes sources d'interruption, pour les situations suivantes :

- arbitrage entre deux requêtes simultanées,
- possibilité pour une source prioritaire d'interrompre le traitement d'une requête moins prioritaire (**interrupt nesting**).



Réentrance

Il est possible qu'une routine d'interruption appelle directement ou indirectement la procédure qui a été interrompue.
Dans ce cas celle-ci se trouve exécutée 2 fois simultanément. On appelle procédure réentrante (**reentrant function**) une procédure qui supporte cette situation sans dommage (Une autre application des procédures réentrantes est le traitement récursif).
Attention : une routine d'interruption ne doit appeler que des procedures réentrantes, ou strictement privées.



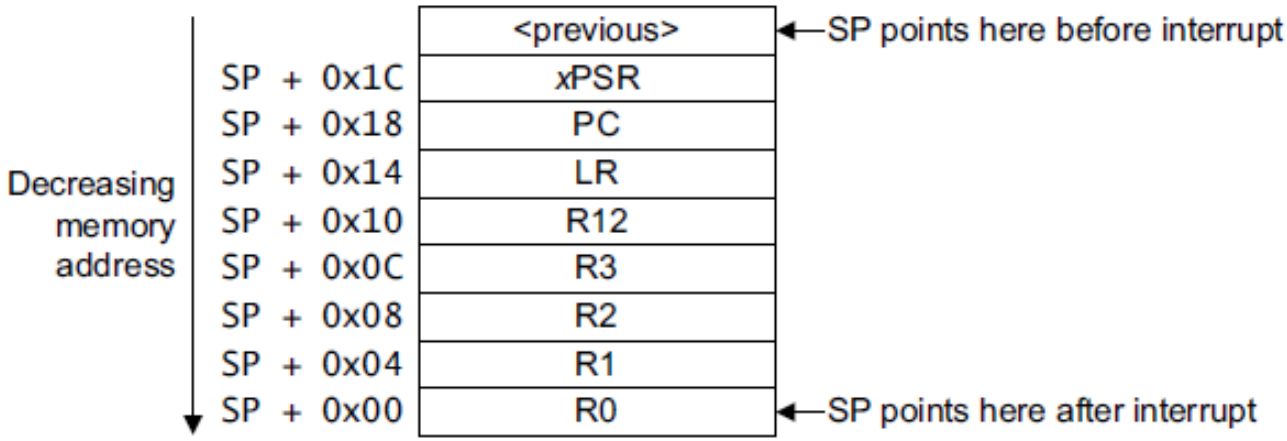
Vecteur d'interruption (Vector Table)

Pour le cortex-M3, le vecteur d'interruption est fixé à l'adresse 0x0, et contient notamment :

- L'adresse initiale de la pile (à l'adresse 0x0).
- L'adresse de la routine pour gérer les reset (à l'adresse 0x4).
- Les adresses pour gérer les exceptions (erreur de gestion mémoire, erreur du bus mémoire, ...).
- L'adresse de la routine à lancer en cas de débordement de compteur matériel (Systick).
- Les adresses pour gérer les interruptions configurables (IRQ0, IRQ1, IRQ2, ...).

Sauvegarde de contexte

Toute interruption déclenche une sauvegarde de contexte pour sauvegarder l'état du programme interrompu. Pour le cortex-M3, les registres sauvegardés sont R0 à R3, R12, LR, PC et xPSR. Ainsi, la routine d'interruption peut suivre la convention d'appel de fonctions de ARM sans modifier l'état du processus interrompu.

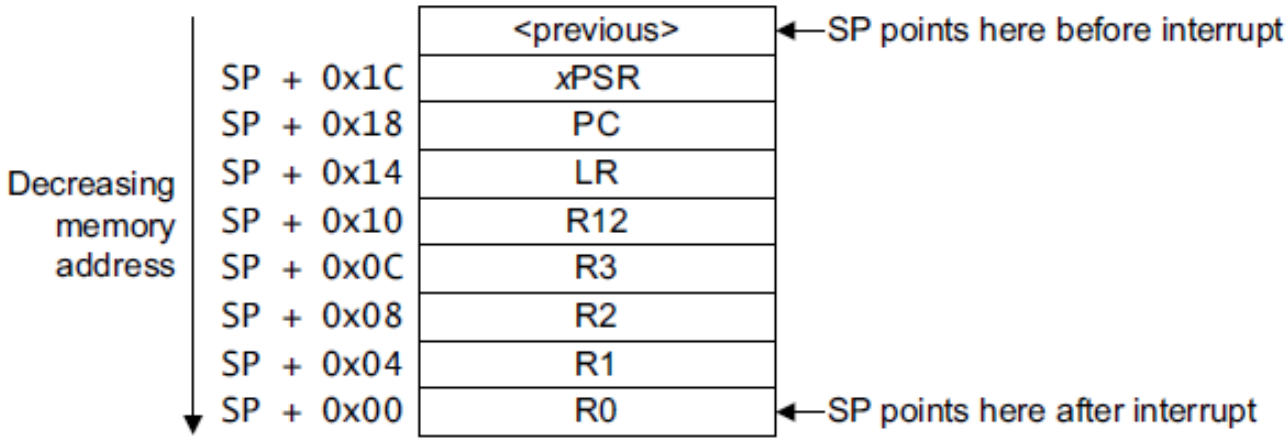


Remarques

- Exceptionnellement(!) l'adresse de retour n'est pas mise dans LR puisqu'elle est déjà dans la pile
- LR contient un code spécial (0xFFFFFFFx) indiquant qu'il s'agit du traitement d'un exception ou interruption
- Il n'y a pas d'instruction spéciale pour le retour d'interruption, un BX LR ou un dépilage équivalent suffit (N.B. le saut à une adresse de la forme 0xFFFFFFFx est par contre interdit dans toute autre situation)

Sauvegarde de contexte

Toute interruption déclenche une sauvegarde de contexte pour sauvegarder l'état du programme interrompu. Pour le cortex-M3, les registres sauvegardés sont R0 à R3, R12, LR, PC et xPSR. Ainsi, la routine d'interruption peut suivre la convention d'appel de fonctions de ARM sans modifier l'état du processus interrompu.



Remarques

- Exceptionnellement(!) l'adresse de retour n'est pas mise dans LR puisqu'elle est déjà dans la pile
- LR contient un code spécial (0xFFFFFFFx) indiquant qu'il s'agit du traitement d'un exception ou interruption
- Il n'y a pas d'instruction spéciale pour le retour d'interruption, un BX LR ou un dépilage équivalent suffit (N.B. le saut à une adresse de la forme 0xFFFFFFFx est par contre interdit dans toute autre situation)

Points à retenir

Représentation interne des données et des instructions

Les données et instructions dans un ordinateur sont représentées sous forme binaire (0 ou 1). Les premiers ordinateurs possédaient d'ailleurs une interface utilisateur binaire.

Langage machine

C'est le seul langage compréhensible par la machine. Ce langage est constitué d'instructions dont la taille est un multiple de 8 bits (1 octet). L'ensemble des instructions exécutables par une machine s'appelle le jeu d'instructions. La représentation binaire des instructions peut varier d'une machine à l'autre, tout comme le jeu d'instructions.

Langage d'assemblage

Afin de simplifier l'écriture de programmes, une représentation textuelle du code a été développée (langage d'assemblage). Elle est constituée :

- De symboles fixes pour les opérations : mnémoniques ou opcodes ;
- De symboles fixes pour les registres et modes d'adressage ;
- De symboles créés par l'utilisateur pour les adresses : Labels, et pour les constantes.

Rôle du langage d'assemblage au XXIe siècle

Il permet de compléter les outils mis à disposition du développeur pour tester, optimiser et sécuriser son code :

- L'analyse de programmes compilés en vue de les optimiser ;
- La compréhension des incidents causés par l'excès d'optimisation des compilateurs ;
- Le reverse-engineering, la gestion de la sécurité.

Applications Desktop/Laptop (PC), tablette, smartphone

- Noyau du système d'exploitation : commutation de tâches par manipulation de pile, traitement des interruptions
- drivers de périphériques
- calcul intensif : multimedia (codecs, traitement de son et d'image)
- graphique 2D/3D (GPU)
- interpréteur de java-bytecode
- émulateur

Applications Systèmes embarqués (au sens large)

- traitement du signal (DSP Digital Signal Processing) ;
- temps réel (OS et applications) ;
- microcontrôleur de petite capacité (8 bits).

Cas où le compilateur atteint certaines limites:

- Utilisation d'instructions spécialisées non gérées par le compilateur :
  - Contrôle du processeur : reset, sleep (pause), watchdog, gestion de privilège, interruption soft, etc...
  - arithmétique saturée (bornage d'un résultat),
  - opérations arithmétiques hétérogènes (par exemple multiplication 32 * 32 $\rightarrow$ 64 bits),
  - division avec reste,
  - multiplication-accumulation (DSP Digital Signal Processing),
  - SIMD (Single Instruction Multiple Data), opérations vectorielles
- Hypothèse sur les valeurs;
- Relâchement des contraintes sur la sauvegarde du contexte.

Jeu d'instruction

C'est l'ensemble des instructions supportées par un processeur donné.

Architecture de jeu d'instruction

Appelé Instruction Set Architecture (ISA) en anglais, c'est la description fonctionnelle du processeur du point de vu du programmeur, c'est-à-dire tout ce qui est visible au niveau du langage machine.
Ce qui est **visible** au niveau de l'ISA :

- Les instructions supportées par le processeur;
- Les registres utilisables par les instructions et leur rôle ;
- L'organisation de la mémoire et des entrées/sorties ;
- Les éléments architecturaux (présence ou non de plusieurs coeurs, présence ou non de caches,$\cdots$).

(Exemple) de ce qui est **invisible** au niveau de l'ISA :

- Les éléments internes du pipeline processeurs (nombre d'étages, exécution in-order ou out-of-order, prédiction de branchement,$\cdots$) ;
- Le nombre de cache et leur associativité ;
- Description interne du circuit du processeur (ALU, contrôleur mémoire, unité de contrôle,$\cdots$).

RISC

Reduced **I**nstruction **S**et **C**omputer

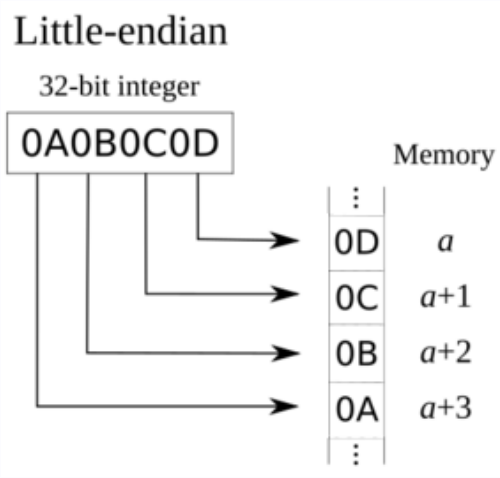
- Premier jeu d'instruction formalisé (Université de Stanford et Berkeley, 1981).
- code d'instructions de longueur fixe = 1 mot de mémoire programme, pour faciliter l'utilisation d'un pipe-line
- durée d'exécution d'une instruction = 1 cycle d'horloge registres banalisés
- éventuellement architecture load-store (2 instructions seulement ont accès à la mémoire)

CISC

Complex **I**nstruction **S**et **C**omputer

- Concept non formalisé : Acronyme donné aux familles non RISC.
- code d'instructions de longueur variable, partant de 1 byte (pas d'alignement mémoire).
- développement incrémental d'une famille de CPUs de plus en plus perfectionnés.
- registres spécialisés.
- ajout d'instructions facilité par le microcodage.
- durée d'exécution d'une instruction très variable.

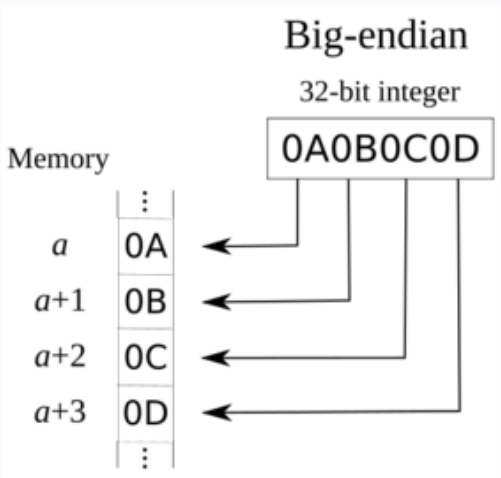
Little-endian



L'adresse mémoire de la donnée pointer vers l'octet de poids **faible**.

- Intel (du 8080 au Pentium)
- Microchip PIC, ATmega328
- Cortex M3 (ARM)

Big-endian



L'adresse mémoire de la donnée pointer vers l'octet de poids **fort**.

- Sun Sparc CPU (avant la version 9)
- IBM-Motorola PowerPC
- headers des paquets Ethernet, IP, UDP, TCP : "network order".

Adressage immédiat ou littéral

L'opérande est une constante codée dans l'instruction. Il n'y a pas à proprement parlée de notion d'adresse.

Adressage directe

L'adresse est codée dans l'instruction. Répandu dans les architectures CISC.

Adressage indirecte par registre

L'adresse est stockée dans un registre. Exemple : LDR R0,[R1]

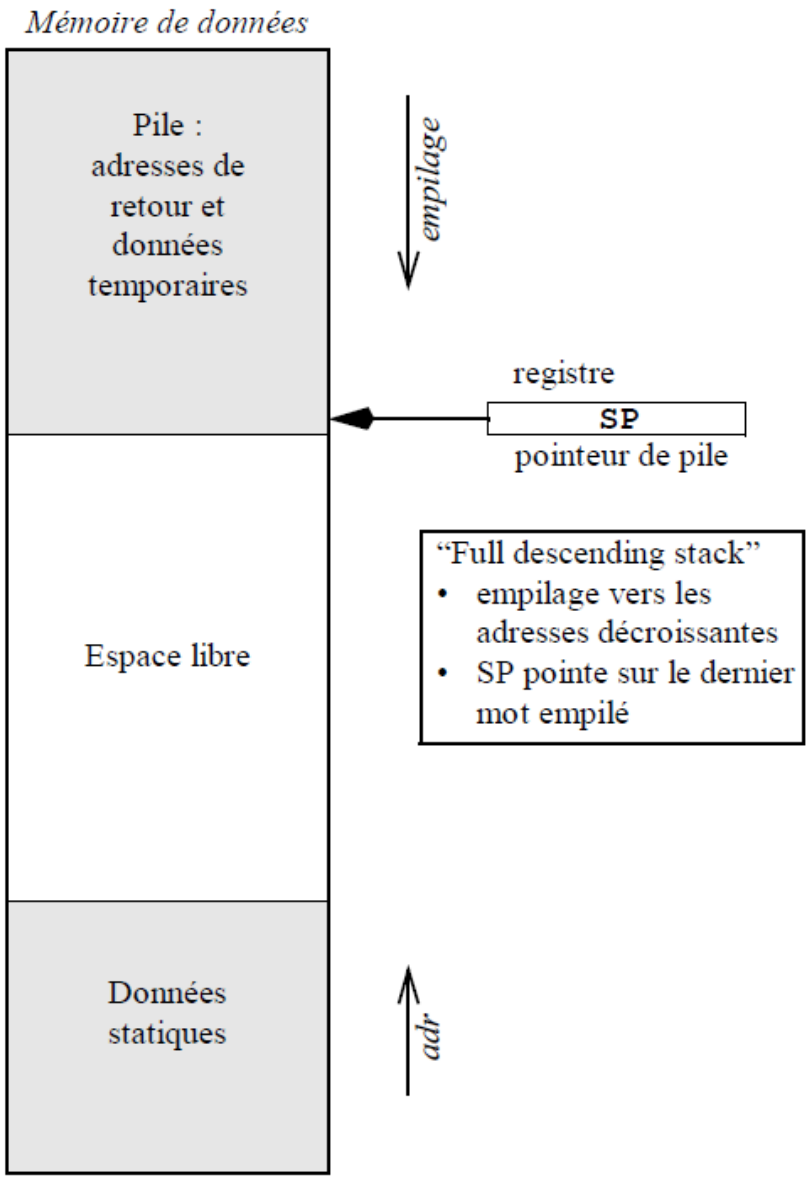
Adressage indirecte à deux registres

L'adresse de base stockée dans un registre. L'adresse effective est obtenue après l'ajout d'un décalage (offset) qui stocké dans un autre registre. Exemple : LDR R0,[R1,R2]. Utilisation : Récupération d'un élément d'un tableau.

Adressage indirecte par registre pré-post-indexé

L'adresse de base stockée dans un registre, auquel on va appliquer un décalage avant (pré-indexé) ou après (post-indexé) modifiant définitivement le registre stockant l'adresse. Exemple : LDR R0,[R1,#4]! (pré-indexé) LDR R0,[R1],#4 (post-indexé) Utilisation : Parcours d'un tableau avec un itérateur.

La pile

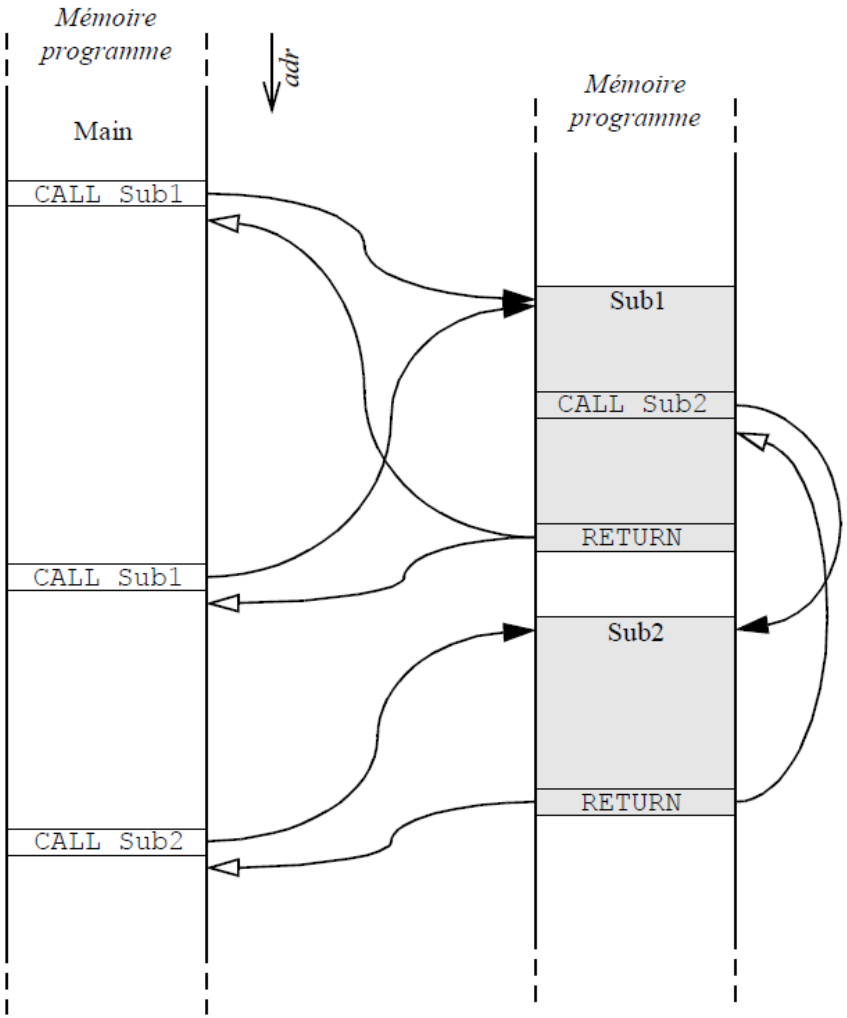


La pile (stack) est une structure de données à accès séquentiel du type LIFO (Last-In, First-Out).

Pour gérer une pile dans la mémoire de données (RAM), il suffit d'un registre pour indiquer la position courante ("pointeur de pile"), qu'on utilise pour empiler ou dépiler des données par adressage indexé.

Afin de définir une pile, deux conventions sont à définir :

- La direction de l'empilage : adresses croissantes (ascending) ou décroisantes (descending) ;
- Le pointer de pile se positionne sur adresse de la dernière valeur empilée (full) ou la future valeur (empty).



Sous-programme (subroutine)

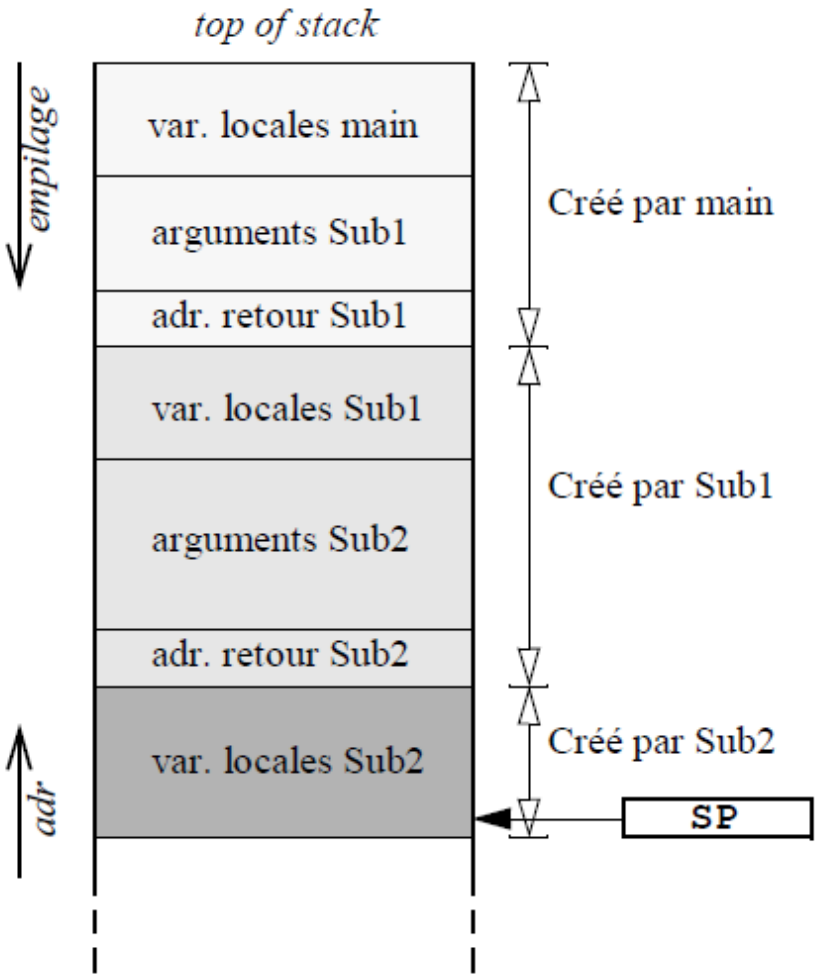
C'est une suite d'instruction que l'on souhaite utiliser plusieurs fois dans un programme. Dans les langages structurés, ils sont appelés fonctions.

Cas de Intel x86

- CALL *Label*
- RETURN

Cas de Arm

- BL *Label*
- BX LR



Pile et arguments de fonction

La pile est également un moyen de passer des arguments aux sous-programmes, surtout les architectures x86/x86_64 qui possèdent peu de registres généraux. En revanche, sur les architectures d'Arm et RISC-V, les arguments se passent principalement par registre (puis la pile si trop d'arguments).

Conventions d'appel chez ARM : l'AAPCS

- Le passage des arguments se fait par registres pour les 4 premières variables, en commençant par R0 et jusqu'à R3.
- Le passage des arguments suivants se fait par la pile. La valeur de retour se fait par le registre R0.
- Le contenu des registres R4 à R11, SP et LR doivent être préservés, c'est-à-dire qu'ils doivent avoir une valeur identique au retour de fonction qu'au moment de l'appel.

Copie de données

- **MOV (Move)**
- **LDR (Load)**
- **STR (Store)**

Arithmétique

- **ADD (Addition)**
- **SUB (Subtraction)**
- **ADC (Add with Carry)**
- **SBC (Sub with Carry)**
- **NEG (Negative)**
- **CMP (Compare)**
- **MUL (Multiplication)**
- **UDIV (Unsigned Division)**
- **SDIV (Multiplication)**

Logique

- **AND :** Opération ET bit-à-bit
- **OR :** Opération OU bit-à-bit
- **EOR :** Egalement appelé le XOR. Opération OU EXCLUSIF bit-à-bit

Décalage

- **LSL (Logical Shift Left)** Opération de décalage à gauche "logique" (i.e. sans préserver le signe).
- **LSR (Logical Shift Right) :** Opération de décalage à droite "logique" (i.e. sans préserver le signe).
- **ASL (Arithmétique Shift Left) :** Opération de décalage à gauche "arithmétique" (i.e. avec préservation du signe). Fonctionnement similaire à LSL.
- **ASR (Arithmétique Shift Right) :** Opération de décalage à droite "arithmétique" (i.e. avec préservation du signe).

Contrôle de flux d'éxecution (control transfer instruction)

- **B (Branch) :** Opération de saut programme à une adresse définie par un label
- **BL (Branch with Link) :** Opération d'appel à sous-programme défini par un label. Sauvegarde de l'adresse de retour dans LR.
- **BX (Branch and Exchange) :** Opération de retour de sous programme à partir de la valeur d'un registre.

Accès à la pile

- **PUSH :** Opération d'empilement d'un registre dans la pile.
- **POP :** Opération de dépilement d'un registre de la pile.

Principe

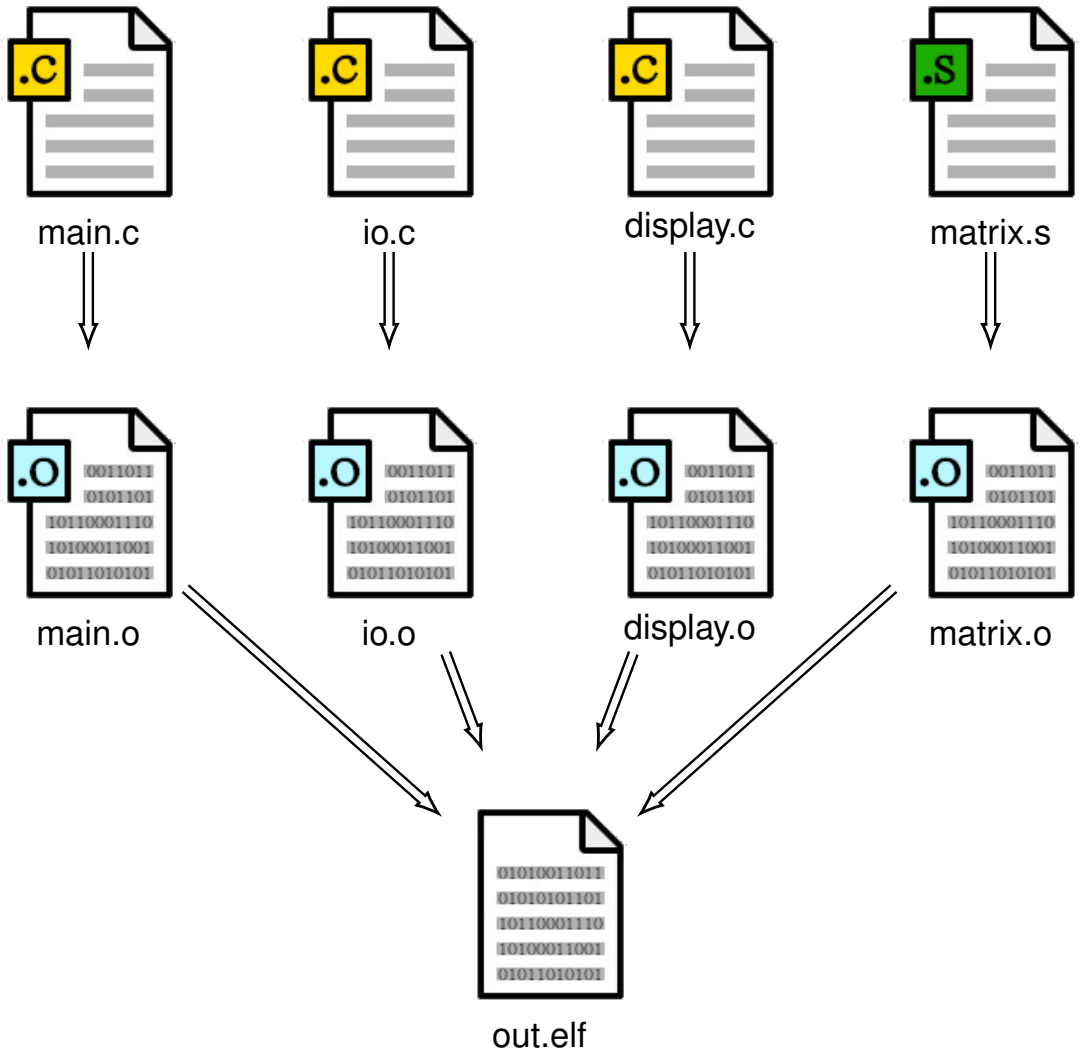
La chaîne de compilation (ou toolchain en anglais), est l'ensemble des procédures permettant le construction d'un fichier exécutable pour un processeur donné à partir de fichiers sources.

Chaîne de compilation GNU

La chaîne de compilation GNU est un ensemble d'outils libres permettant la compilation de fichier décrits en langage C/C++ et d'assemblage. Elle est composée de :

- **binutils :** suite d'outils permettant de manipuler les fichiers binaires. En particulier, contient un assembleur et un éditeur de lien.
- **gcc :** compilateur pour langage C/C++.
- **glibc :** bibliothèque permettant de gérer les appels systèmes ainsi que la gestion de fonctionnalités requises par certaines normes logicielles (POSIX, ISO C99, ...).
- **gdb :** debugger de fichiers binaires compilés avec GNU.
- **make :** interpréteur de scripts qui facilite l'automatisation des séquences de construction du projet (build).

Etapes lors de la compilation

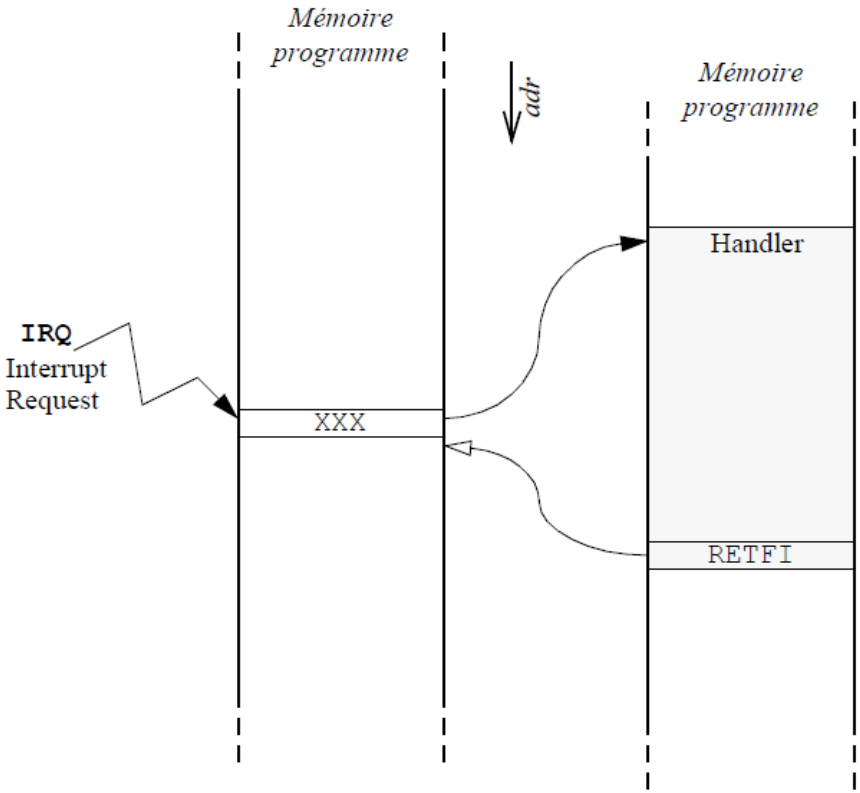


Etapes clefs

- Génération des fichiers objets contenant code compilé + symboles non résolus + symboles exportés.
- Édition des liens pour attribuer les plages d'adresse des codes + concaténer les objets + résoudre les symboles non résolus.

Linker Script

Permet de spécifier à l'éditeur des liens des directives pour positionner les différents segments en mémoire. Remarque : La pile et le tas ne sont pas inclus dans le linker script car la pile est toujours aux adresses hautes (sauf pile ascendante mais c'est très rare), et le tas commence après que tous les segments aient été positionnés.



Principe

- Une interruption est un événement matériel permettant à un périphérique de forcer l'exécution d'un code par le processeur.
- Ce code est appelé routine d'interruption (**interrupt routine** ou **interrupt handler**) dont le rôle est de traiter la requête du périphérique.
- Le processeur peut lui-aussi générer une interruption pour lui-même (interruption logicielle).
- Cela implique une sauvegarde du contexte.

Vecteur d'intérruption

Pour appliquer un traitement spécifique pour chaque source d'interruption, la solution la plus efficace est une table des points d'entrées (**vecteurs**) des handlers, souvent installée à une adresse fixe.

L'objectif

L'objectif du TD Quiz est d'implémenter en C/ASM l'addition entière sur 64 bits sur un processeur Arm cortex-M3 32 bits.

Organisation

L'enseignant posera des questions à des moments clefs. Vous répondrez à ces questions via iquiz. Vous pouvez utiliser le cours pour vous aider à répondre aux questions. La réponse la plus fréquente l'emporte, pouvant amener à un code fonctionnel, ou pas !