Introduction à l'Assembleur ARM

Vincent Berry et Guillaume Gourmelen

Objectifs du cours

1. Manipuler un langage de programmation de bas niveau ;
2. Utiliser des jeux d'instructions pour écrire des mini-programmes ;
3. Comprendre la segmentation logique de la mémoire (pile, tas, etc.) ;
4. Comprendre le mécanisme d'appel de procédure (sous-programme) ;
5. Percevoir les principes de programmation universelle.

Pourquoi ? [Strandth & Durand, 2005]

Une petite citation

Le rôle d'un informaticien n'est pas de concevoir des architectures, en revanche il a besoin d'un modèle de fonctionnement de l'ordinateur qui lui donne une bonne idée de la performance de son programme et de l'impact que chaque modification du programme aura sur sa performance.

Assimiler un tel modèle suppose un certain nombre de connaissances sur le fonctionnement d'un ordinateur, notamment le mécanisme d'appel de fonction, la transmission des paramètres d'une fonction à l'autre, l'allocation ou la libération d'espace mémoire, etc.

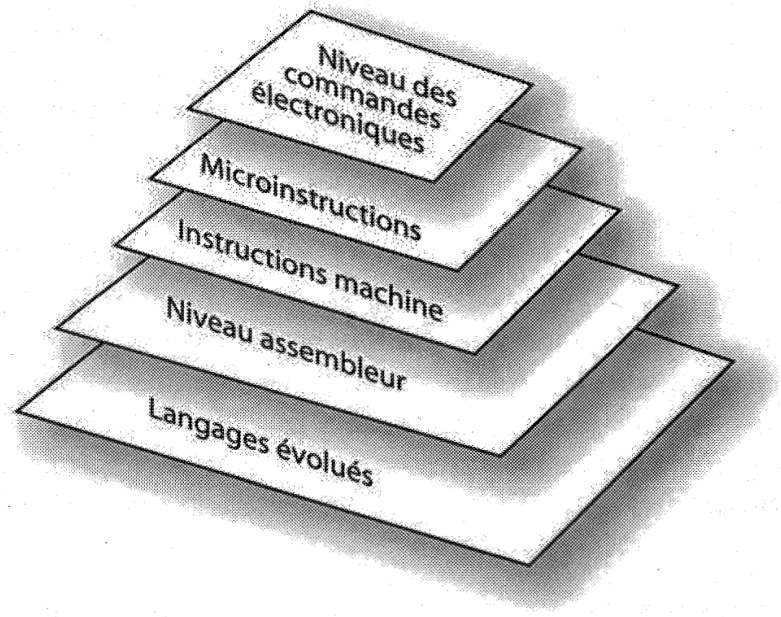
Conclusion

⇒ Apprendre l'architecture et un langage machine permet cela.

Niveaux de programmation

Langage machine

Le programmeur a le choix entre différents langages (Assembleur, Python, Java, C, etc.). La machine ne comprend que le langage machine (i.e., instructions binaires) !



Niveaux de programmation

- L'assembleur (langage d'assemblage) est le premier langage non binaire accessible au programmeur ;
- Code mnémoniques et symboles ;
- L'assembleur (programme traducteur) convertit le langage d'assemblage en langage machine ;
- Permet d'exploiter au maximum les ressources de la machine ;
- Dépend de la machine, de son architecture.

Compilation et Interprétation

Compilation

Génération d'un code objet intermédiaire équivalent au code source

- Traduction réalisée une seule fois ;
- Exécution rapide et efficace.

Interprétation

Conversion et exécution d'un code source en une seule étape : les instructions sont lues les unes après les autres, traduites et exécutées au passage.

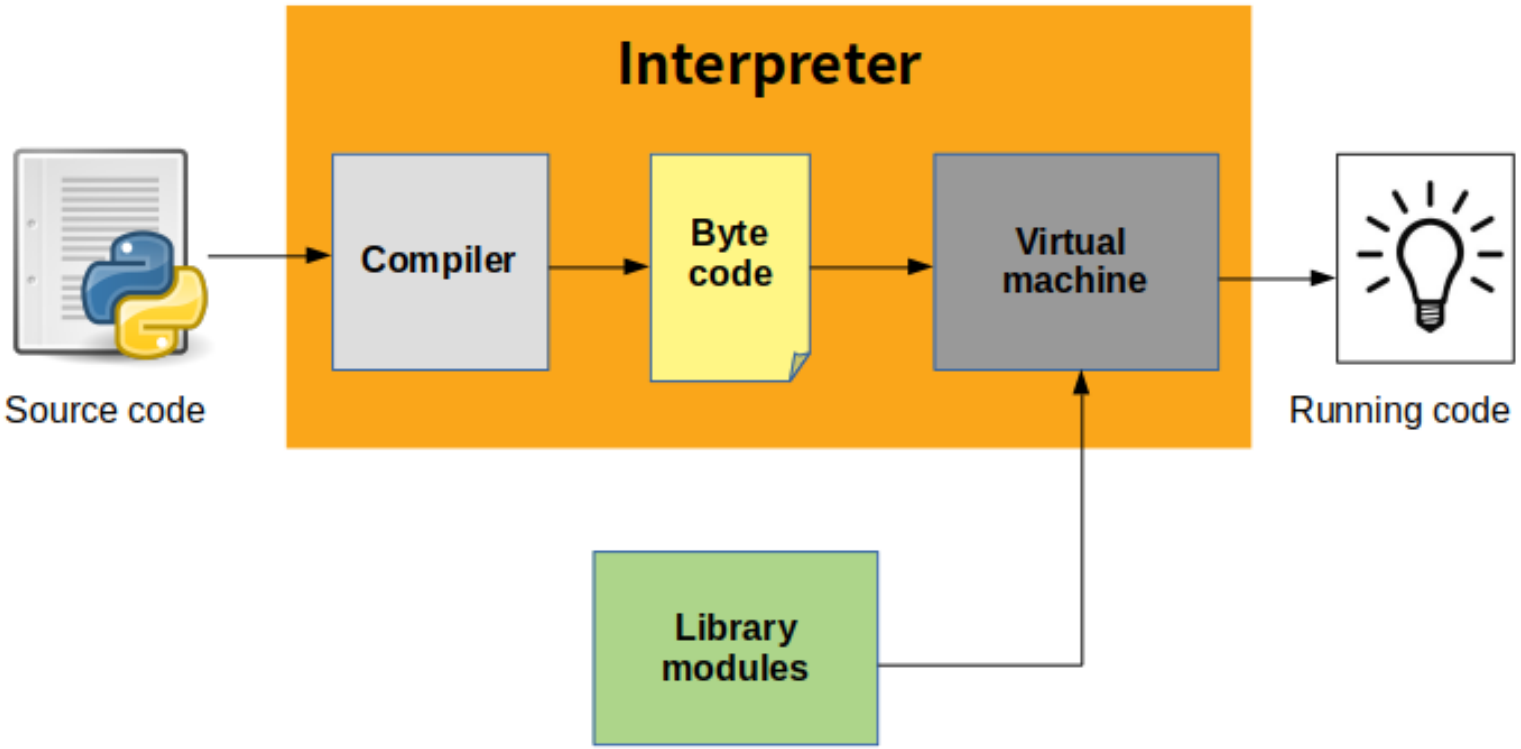
- Pas de fichier de code intermédiaire ;
- Répétition du travail de traduction à chaque exécution.

Types de compilation

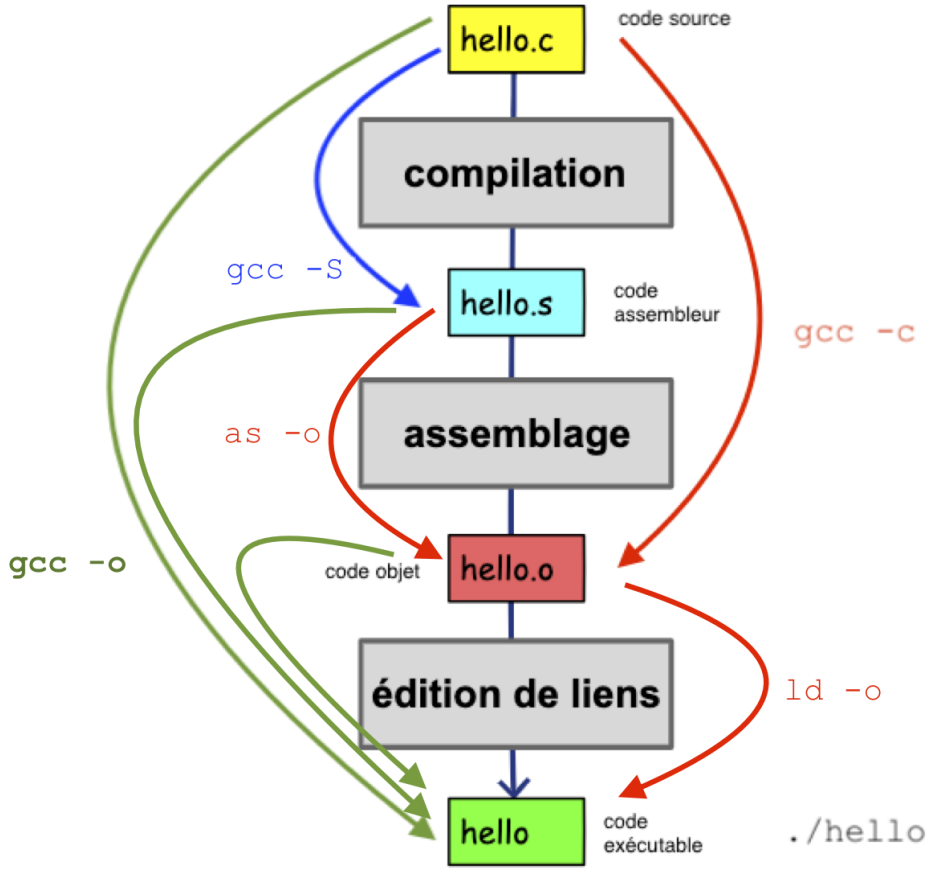
Compilation

- Native : le code objet est de l'assembleur et sera traduit en langage machine. Cette compilation est donc dépendante de l'architecture, mais est la plus efficace ;
- « Bytecode » : le code objet est exécuté par une machine virtuelle. Cette compilation est donc indépendante de l'architecture, mais est en moyenne 5 fois moins efficace qu'une compilation native.

Interprétation : cas de Python



Compilation native : cas de C



Warning : `main` nécessaire pour gcc mais `_start` pour ld.

Langage d'assemblage

- Utilisés par les spécialistes ⇒ optimisation ;
- Pour valoriser l'architecture spécifique de la machine ;
- Diagnostic d'erreurs (i.e., examen du contenu de la mémoire) ;
- L'assembleur est une variante symbolique du langage machine ⇒ même jeu d'instructions ;
- Propre à chaque type de machine ;
- Permet d'accéder aux ressources de la machine (i.e., registres) ;
- Permet d'accéder aux facilités de traitement (e.g., décalage) ;

Le programmeur peut utiliser :

- Codes mnémoniques (jeu d'instruction) ;
- Étiquettes/*labels* (adresses symboliques) ;
- Littéraux (constantes numériques ou caractères ASCII) ;
- Directives.

Processeurs CISC et RISC

CISC

- « Complex Instruction Set Computer » ;
- Jeu étendu d'instructions complexes ;
- Chaque instruction peut effectuer plusieurs opérations élémentaires ;
- Jeu d'instructions comportant beaucoup d'exceptions ;
- Instructions codées sur une taille variable.

Processeurs CISC et RISC

RISC

- « Reduced Instruction Set Computer » ;
- Jeu d'instructions réduit ;
- Chaque instruction effectue une seule opération élémentaire ;
- Jeu d'instructions plus uniforme ;
- Instructions codées sur la même taille et s'exécutant dans le même temps (un cycle d'horloge en général).

Processeurs CISC et RISC

Répartition des principaux processeurs

CISC (pré-1985)	RISC (post-1985)
S/360 (IBM)	Alpha (DEC)
VAX (DEC)	PowerPC (IBM)
68xx, 680x0 (Motorola)	MIPS
x86, Pentium (Intel)	PA-RISC (Hewlett-Packard)
	SPARC (Sun)
	ARM

Les processeurs ARM

- Langage ARM : assembleur des processeurs ARM
- Processeurs avec une architecture de type RISC 32 bits (ARMv1 à ARMv7) et 64 bits (ARMv8)
- Introduits en 1983 par Arcon Computers
- Développés par ARM Ltd depuis 1990
- Processeurs à faible consommation : équipent les smartphones, les iPads et les raspberries !

Processeurs avec une **architecture** (licence) **ARM**, mais fabriqués par des entreprises différentes (Apple, Broadcom (Raspberries), Huawei, Nvidia, Qualcomm, Samsung, STMicroelec., Texas Instr., Toshiba, ...)

Les registres dans l'ARM

Registres dans l'ARMv7 (ce qui équipe nos rasp.)

- L'architecture dispose de 16 registres généraux de 32 bits, nommés *r*0 à *r*15
- Les registres *r*0 à *r*12 peuvent être utilisés dans les calculs (registres non dédiés)
- Les trois registres *r*13, *r*14 et *r*15 sont utilisés dans des tâches spécifiques réalisés par le processeur :
  - *r*15 (nommé pc aussi) est dédié en permanence au stockage du compteur ordinal (PC : *Program Counter*)
  - *r*14 (nommé lr – *Link Register*) est utilisé lors de l'exécution d'une instruction précise (*branch with link*) pour stocker une copie du compteur ordinal
  - *r*13 (nommé sp – *stack pointer*) : prochain cours

Les registres dans l'ARM -suite-

Compteur ordinal

- Compteur ordinal = 00....0 au départ
- Il est incrémenté de 4 (4 octets) à chaque fois
- Certaines instructions altèrent la valeur du PC (instr. de saut)

Autre registre : cpsr (current program status register)

Contient des fanions/flags pour indiquer l'état des calculs :

- Bit 31 : N ⟶ résultat **n**égatif
- Bit 30 : Z ⟶ résultat nul (**z**ero)
- Bit 29 : C ⟶ résultat avec retenue (**c**arry)
- Bit 28 : V ⟶ dépassement de capacité (o**V**erflow)
- ... (IRQ, user mode, system mode, ...)

Les instructions dans l'ARM

Jeu d'instructions ARM

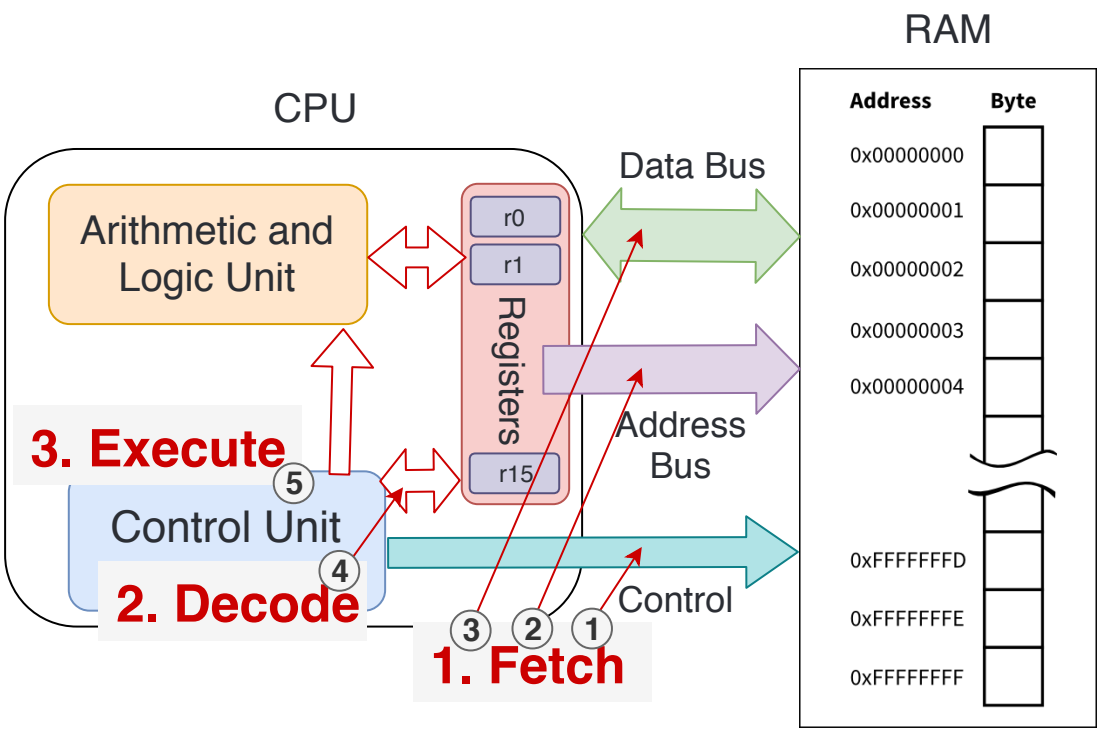
ARM propose trois types principaux d'instructions (sur 32 bits principalement) :

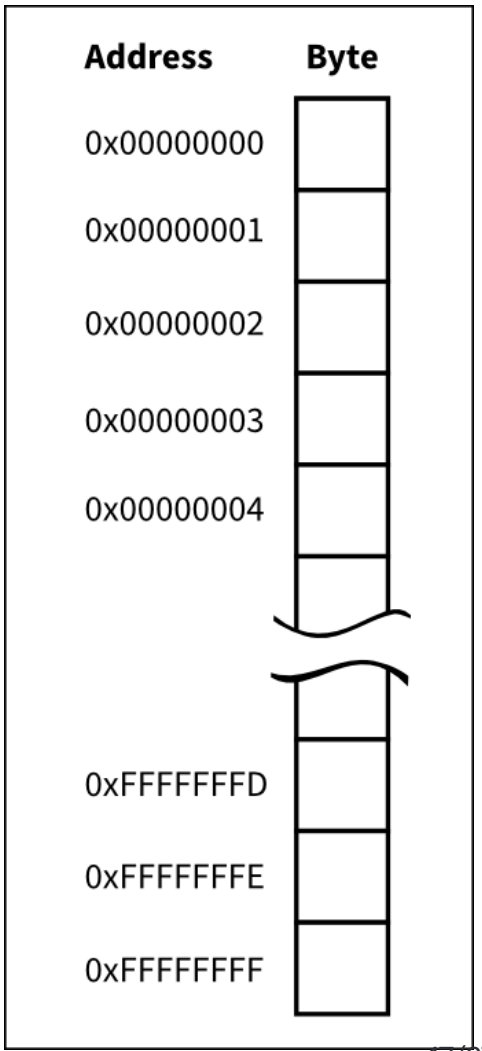
1. Les instructions de transfert entre registres et mémoire ;
2. Les instructions de calcul ;
3. Les instructions de saut (branchement) ;

Seules les instructions de transfert permettent d'accéder à la mémoire ; les autres opèrent uniquement sur les registres.

Mémoire et processeur

- La mémoire est organisée en cellules d'un octet (8 bits), chacune possédant une adresse
- Un processeur accède aux programmes stockés en mémoire pour les exécuter en respectant un cycle : **fetch-decode-execute**





Mémoire et adressage

- Le code d'un programme en Assembleur ARM comporte deux sections : `.data` (variables utilisées par le programme) et `.text` (instructions du programme)

```
/* - - mon_programme.s */
/* - - Data section */
.data
/* - - Instruction section */
.text
.global main
main:
  bl exit
```

- Chaque unité du programme (une donnée ou une instruction) est représentée sur 32 bits (possibilité sur 8/16 bits)
  ⇒ occupe 4 octets (PC incrémenté de 4 après un *execute* si l'instruction n'affecte pas le PC)

Sur vos Raspberries

- Se connecter en `ssh`
- On va utiliser un éditeur de texte pour écrire le code source des programmes (fichiers .s)
- Installer emacs (si pas déjà présent : quel scandale !) : `sudo apt-get install emacs`
- Créer un fichier hello.s et y ajouter le code de la diapo précédente

Pour exécuter le code Assembleur

- Pour exécuter un programme Assembleur, on utilisera l'outil Gnu **as** en ligne de commande pour générer le code machine (objet)

```
as -o hello.o -g hello.s
```

- On utilisera l'outil Gnu **gcc** pour faire l'édition des liens

```
gcc -o hello hello.o
```

  Conformément au schéma de la diapo 8, vous pouvez aussi utiliser `ld -o hello hello.o`

- Pour exécuter :

```
./hello
```

Pour debugger

- On utilisera l'outil Gnu **gdb** en ligne de commande en mode interactif

```
gdb ./hello
```

- ça donne accès à un Shell qui permet de taper des commandes de débogage (start, disassemble, info registers, modifier les registres, avancer pas à pas dans l'exécution, ...)
- Consulter le site Web ci-dessous et faire l'ensemble des étapes :

  https://thinkingeek.com/2013/01/12/arm-assembler-raspberry-pi-chapter-4/

  Explorer d'autres commandes (avec la touche Tabulation)

Format des déclarations de variables

- Une déclaration de données est faite de la façon suivante :

```
/* - - Section pour declarer les variables globales */
.data
/* S'assure que la prochaine variable declaree sera placee a une
     adresse multiple de 4. C'est important car ARM ne peut utiliser
      que des données commencant a de tels emplacements  */
.balign 4
/* Declare une variable nommee myvar1 */
myvar1:
  /* myvar1 correspond a 4 octets contenant le nombre 3 */
  .word 3
```

- .word est une directive indiquant que le nombre 3 va occuper un mot (4 octets). Possibilités : `.byte` et `.2byte`
- myvar1 est une étiquette/label : symbole représentant l'adresse mémoire du premier octet du mot dans lequel est stocké le nombre 3
- TP : Tester le debugger avec cette déclaration

Format des instructions

- Toute instruction prend la forme suivante :
  Code_Operation Operande1, Operande2
  ou bien *Code_Operation Operande1, Operande2, Operande3*
  Les opérandes sont soit des registres (r0, r1, ...), des variables ou des valeurs littérales
- Les valeurs littérales : #3 pour le nombre entier 3, #'c' pour le caractère 'c', #0b10110 pour un nombre binaire, #0x19DA pour un hexadécimal

Important : accès à la mémoire

Les données à manipuler par un programme sont en général dans la mémoire principale (RAM). Mais les instructions de calculs ne manipulent que des données situées dans les registres du CPU.
Donc, toute donnée (variable) doit d'abord être transférée de la mémoire vers un registre avant de servir à un calcul. On peut stocker une nouvelle valeur pour cette variable en faisant après les calculs un transfert registre ⟶ mémoire principale.
Donc pour incrémenter une valeur stockée en mémoire (exemple : `i=i+1`), on procède en 3 temps :

1. chargement de la valeur de i dans un registre r (*load in register*)
2. on ajoute 1 à la valeur de r
3. stockage du contenu de r à l'adresse de i en RAM (*store register*)

Load et store sont les seules instructions autorisées à accéder à la RAM

Instructions de transfert : Mémoire ⟷ Registres

Lecture et écriture

- Lecture (« LoaD Register – ldr ») :

  ldr r_{dest}, *source*

  - Si *source* est définie sans crochets ([et]) : une adresse est transférée dans le registre
    source peut être une variable ou un registre qui contiennent une adresse,
    source peut être "`=variable`" : c'est l'adresse de la variable qui sera transférée dans le registre (équiv. &variable en C)
  - Si *source* est définie avec crochets ([et]) : c'est la valeur se trouvant à l'adresse (et non l'adresse) qui est transférée dans le registre (équiv. *variable en C)

Instructions de transfert : Mémoire ⟷ Registres

Lecture et écriture

- Écriture (« STore Register – str ») :

  str r_{source}, r_{dest} `/* ordre des args inverse de ldr */`

- Le mot stocké dans le registre r_{source} est copié vers la cellule mémoire dont l'adresse est indiquée dans r_{dest}.

Lecture et écriture avec un offset

- Écriture avec offset (lecture avec offset possible aussi) :

  str r_{source}, [r_{dest}, #*offset*]

- Le mot stocké dans le registre r_{source} est copié vers la cellule mémoire dont l'adresse est indiquée dans r_{dest} + un décalage égal à #offset.

Application : incrémentation de variable

Exercice

Écrivez un programme qui déclare une variable i en lui donnant la valeur 5, puis incrémente le contenu de cette variable en mémoire.

Indication :

add r1, r1, #1 // équiv. à r1++

c'est-à-dire incrémente de 1 la valeur contenue dans le registre r1

Instructions de calcul

Principe

Ces instructions lisent la valeur de 0, 1 ou 2 registres dits arguments, effectuent un calcul, puis écrivent le résultat dans un registre dit destination.

Un même registre peut figurer plusieurs fois parmi les arguments et destination.

Instructions de calcul

Instruction de déplacement

- Déplacement (« mov ») :

  mov dest, source

  Le contenu de source est transféré vers le registre dest.

  - source peut être un opérande “immédiat” (littéral)
    Ex : mov r0, #15
    Transférer l’entier 15 dans le registre r0
  - *source* peut être un registre

Instructions de calcul

Opérations arithmétiques

- Addition (« add ») :

  add dest, op1, op2

  Produit la somme de op1 et op2 et range le résultat dans le registre dest.
  dest et op1 doivent être des registres et op2 peut être :
  - un opérande “immédiat” :
    add r1, r1, #1 // Incrémentation : équiv. à r1++
  - ou un registre :
    add r0, r0, r1

- Autres opérations arithmétiques : sub, mul, div, asl (arithmetic shift left), asr, ror (rotate right), ...

Instructions de calcul

Opérations logiques

- Même principe que pour les opérations arithmétiques
- **and** (ET logique) :
  and r0, r1, r2
  produit un “et-logique” à chaque bit des opérandes r1 et r2 et range le résultat dans r0
- **bic** (et logique entre l’opérande 1 et le not de l’opérande 2)
- **orr** (ou inclusif), **eor** (ou exclusif), ...

Instructions de calcul

Comparaisons

- Comparaison (« cmp ») :

  cmp op1, op2

  Compare op1 et op2 (en effectuant $r1 - r2$) et met à jour les drapeaux du registre *cpsr* en fonction du résultat (voir diapo 15)
  op1 doit être un registre et op2 un registre ou une valeur
- Le résultat de la comparaison est exploité par les instructions qui s'exécuteront après (instructions avec codes mnémoniques suffixées)

Suffixes aux mnémoniques – instructions conditionnelles

- Tout mnémonique dans une instruction peut être suffixé d'une condition (deux lettres, comme **addne** ou **mullt**)
- Si cette condition est vraie, l'instruction est exécutée. Sinon, elle est ignorée
- Ces conditions sont souvent liées aux drapeaux d'état (résultat d'un calcul ou une comparaison précédents, qui ont produits un résultat nul, négatif, une retenue ou un débordement, ...)
- Exemple :
  cmp r0, #0
  addgt r1,r1,r2 /* *r1* ← *r1* + *r2* */
  Exécuter l'addition seulement si `r0 > 0` (**gt** = *greater than*)

Suffixes aux mnémoniques (Mise à jour des drapeaux/*flags*)

- Un code mnémonique peut être suffixé aussi de la lettre s pour indiquer que l'instruction doit mettre à jour les drapeaux (s = *set flags*)
  mov fait un transfert sans toucher aux drapeaux
  movs modifie les drapeaux
- L'instruction **cmp** modifie systématiquement les drapeaux (pas besoin d'ajouter un s)

Instructions de saut

Principe

Une instruction de saut (**branch**) permet de faire un saut à n'importe quel endroit dans le programme (changer le flux de contrôle)

- Le code mnémonique de la forme la plus simple est **b** ou **bal** (le **al** est un suffixe pour dire *Branch ALways*)
- Une seule opérande = une expression qui indique où aller dans le programme (c'est souvent une étiquette genre `main`)
- Possibilités : blt (*branch if less than*), bge (*branch if greater than or equal*), beq (*if equal*), bne (*if not equal*), ...
- ça sert à écrire des if-then-else et des boucles

Instructions de saut

Saut avec lien/retour

- Code mnénmonique : **bl** (*branch with link*)
- Avant le saut, l'adresse de l'instruction qui suit l'instruction courante (obtenue du PC=r15) est stockée dans le registre r14
- De cette façon le programme peut revenir au point de départ du saut
- ça sert à écrire des appels de fonctions et des *returns*
- Pour faire un return, il suffit d'écrire :
  movs r15,r14 ou **mov** r15,r14

Instructions de saut

Exercices : structures de contrôle usuelles

- Écrire un programme Assembleur avec un si-alors
  (si(r4 <> r5) alors r3=r4 fin-si)
- Écrire un programme Assembleur avec un si-alors-sinon
  (si(r4 < r5) alors r3=r4 sinon r3=r5 fin-si)
- Écrire un programme Assembleur avec une boucle tant-que
  (tant-que(r4 >= r5) r3=r3+r4 ; r4-=1 fin-tant-que)

Instructions de saut

Une instruction spéciale

- Appel système (« system call ») :

  swi

- Provoque un appel au noyau du système d'exploitation
- Le code mnémnique est suivi d'un seul opérande : une expression qui dépend du système d'exploitation
- Appels systèmes courants : lire et écrire depuis/vers l'E/S standards, lire et écrire des fichiers/périph. sur le disque, ...

Programme « Hello World »

Code

```
/* -- hello.s */
.data
msg: .ascii "Hello World\n"
.text
.global main
main:
  mov r2, #12 // Nb de caractères à afficher
  mov r7, #4 // Dire à l'OS que l'IRQ ci-dessous est un syscall print
  mov r0, #1 // Dire à l'OS que l'IRQ est un syscall print
  ldr r1, =msg // Message à afficher stocké dans r1
  swi 0 // SoftWare Interrupt (IRQ)
  mov r7, #1 // IRQ = syscall exit
  swi 0
```

Programme « Hello World » plus simple

On utilise des fonctions système de la bibliothèque standard du langage C (`printf` et `exit`)

Note : dans ce cas compiler avec gcc (fait le lien avec la lib standard du c) et pas avec as + ld

Code

```
/* -- hello2.s */

.data
msg:     .ascii "Hello World\n"

.text
.global main

main:
        ldr r0, =msg
        bl printf
        bl exit
```

Un autre programme

Code

Demander la saisie d'un entier et afficher s'il est positif (>=0).

```
/* -- positif.s */
.data
enter:  .asciz "Entrer un nombre : "
scan:   .asciz "%d"
print:  .asciz "C'est un nombre %s"
msg1:   .asciz "positif ou nul\n"
nb:     .word 0 // La variable dans laquelle le nombre saisi sera stocké
.text
.global main
main:
        ldr r0, =enter
        bl printf
        ldr r0, =scan
        ldr r1, =nb // On range dans r1 un pointeur vers nb
        bl scanf // Après ça, dans nb on a le nombre saisi
```

Un autre programme -suite-

Code

```
        ldr r1, =nb // On repointe dans r1 vers nb (après le scanf)
        ldr r2, [r1] // On récupère dans r2 le nombre pointé par r1
        cmp r2, #0
        blt fin
        ldr r0, =print
        ldr r1, =msg1
        bl printf
fin:
        bl exit
```

À vous de jouer !

Exercices

1. Modifier le programme positif.s pour afficher si le nombre est négatif
2. Demander la saisie d'un entier et rendre la valeur absolue de cet entier (afficher le résultat) ;
3. Afficher les *n* premiers entiers (en partant de 1), où l'entier *n* sera demandé à l'utilisateur ;
4. Demander la saisie d'un entier et dire si cet entier est pair ou non (afficher le résultat).

Diapos et références

Diapos construites sur la base du cours de :

David Delahaye, Prof FdS

Références en ligne

- https://thinkingeek.com/arm-assembler-raspberry-pi/
- http://www.peter-cockerell.net/aalp/
- https://community.arm.com/
- https://www.youtube.com/watch?v=ViNnfoE56V8
- https://azeria-labs.com/writing-arm-assembly-part-1/