Université Grenoble Alpes (UGA)

UFR en Informatique, Mathématique et Mathématiques Appliquées (IM2AG)
Département Licence Sciences et Technologies (DLST)

Architectures des ordinateurs (une introduction)

Unité d'Enseignement INF401 pour les Parcours INM, MIN et MIN-int

Année 1, l'exécution des programmes en langage machine.

(comprendre pour programmer efficacement et sans bug)

Sujets des Travaux Dirigés

Sujets des Travaux Pratiques

Année Universitaire 2025 / 2026

Table des matières

I Travaux Dirigés **3**

TD séance 1 : Codages **4**

TD séance 2 : Représentation des nombres **6**

TD séances 3 et 4 : Langage machine, codage des données **16**

TD séances 5 et 6 : Codage des structures de contrôle **23**

TD séance 7 : Fonctions : paramètres et résultat **27**

TD séance 8 : Appels/retours de procédures, actions sur la pile **33**

TD séance 9 : Correction du partiel **36**

TD séance 10 : Paramètres dans la pile, paramètres passés par adresse **37**

TD séances 11 et 12 : Organisation d'un processeur : une machine à pile **39**

II Travaux Pratiques **45**

TP séance 1 : Représentation des informations (ex. : images, programmes, entiers) **46**

TP séance 2 : Codage et calculs en base 2 **52**

TP séances 3 et 4 : Codage des données **56**

TP séance 5 : Codage de structures de contrôle et metteur au point `gdb` **62**

TP séances 6 et 7 : Parcours de tableaux **66**

TP séances 8 et 9 : Procédures, fonctions et paramètres, liens entre ARM et C **70**

TP séances 10 et 11 : Programmation fonctionnelle (MapRed) **74**

TP séance 12 : Etude du code produit par `gcc`, optimisations **77**

Première partie

Travaux Dirigés

TD séance 1 : Codages

1.1 Codage binaire, hexadécimal de nombres entiers naturels

Ecrire les 16 premiers entiers en décimal, binaire et hexadécimal.

1.2 Codage ASCII

Regarder la table de codes ascii qui est en annexe. Sur combien de bits est codé un caractère ?
Soit la fonction : `code_ascii : un caractère --> un entier` $\in [0, 127]$.
Comment passe-t-on du code d'une lettre majuscule au code d'une lettre minuscule ou l'inverse.

1.3 Codage par champs : codage d'une date

On veut coder une information du style : `lundi 12 janvier`.
Codage du jour de la semaine : lun :0,...,dim :6, il faut 3 bits
Codage du quantième du jour dans le mois : 1..31, 5 bits
Codage du mois : 1..12, 4 bits
Quel est le code de la date : `lundi 12 janvier` ? Quelle est la date associée au code 010 00011 0101 ?
Quel est la date associée au code 111 11111 1111 ?

1.4 Code d'une instruction ARM

En utilisant la documentation technique, coder en binaire les instructions : `MOV r5, r7`, et `MOV r5, #7`.

1.5 Codage des entiers naturels entre 16 et 255

Combien faut-il de bits pour coder un entier entre 16 et 255 ? Coder les entiers naturels 17, 67, 188 en binaire et en hexadécimal. En déduire une méthode rapide de passage binaire vers hexadécimal ainsi que l'inverse. Calculer la valeur décimale de l'octet 10101010 représentant un entier naturel.

1.6 Codage de nombres à virgule

On représente des nombres à virgule de l'intervalle [0, 16[par un octet selon le code suivant : les 4 bits de poids forts codent la partie entière, Les 4 bits de poids faibles codent la partie après la virgule

Par exemple 01101010 représente 6,625. En effet $x_3x_2x_1x_0x_{-1}x_{-2}x_{-3}x_{-4} = 01101010$ donne X = $4 + 2 + \frac{1}{2} + \frac{1}{8} = 6{,}625$. (Rappelons que les anglo-saxons le notent 6.625)

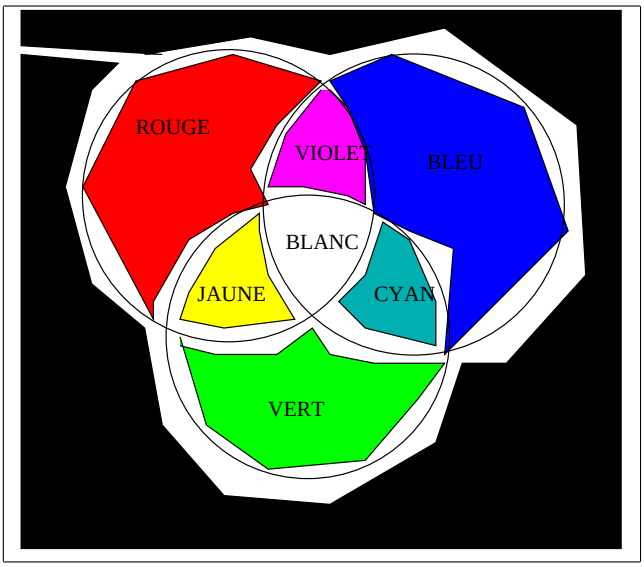


FIGURE 1.1 – Codage de couleurs

rang du bit	3	2	1	0	-1	-2	-3	-4
bit	0	1	1	0	1	0	1	0
valeur arithmétique correspondante	8	4	2	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$

Dans le cas général on a : $X = \sum_{i=-4}^{3} 2^i \times x_i$

Que représente le vecteur 00010100 ?

Donner l'écriture binaire de 5,125.

Quel est le plus grand nombre représentable selon ce code ?

Peut-on représenter $\frac{7}{3}$ ou $\frac{8}{5}$?

1.7 Codage de couleurs

Codage des 16 couleurs sur les premiers PC couleurs : Ici, il y a un bit de rouge, un bit de vert, un bit de bleu et un bit de clair. Ainsi on voit que cobalt est cyan pale, rose est rouge pale, mauve est violet pale, jaune est brun pale et blanc est gris pale. La figure 1.1 montre les "mélanges".

B	$b_3b_2b_1b_0$		B	$b_3b_2b_1b_0$		B	$b_3b_2b_1b_0$	
0	0 0 0 0	noir	5	0 1 0 1	violet	10	1 0 1 0	vertpâle
1	0 0 0 1	bleu	6	0 1 1 0	brun	11	1 0 1 1	cobalt
2	0 0 1 0	vert	7	0 1 1 1	gris	12	1 1 0 0	rose
3	0 0 1 1	cyan	8	1 0 0 0	noir pâle	13	1 1 0 1	mauve
4	0 1 0 0	rouge	9	1 0 0 1	bleu pâle	14	1 1 1 0	jaune
						15	1 1 1 1	blanc

Important : Le TD2 comporte une longue partie à lire avant les exercices, prévoir de faire cette lecture pendant la semaine, avant le TD !

TD séance 2 : Représentation des nombres

2.1 Introduction

Un entier E peut être représenté par une suite de n chiffres (ou digits) e_i, tous inférieurs à la base utilisée ($0 \leq e_i \leq B-1$) et tels que $E = \sum_{i=0}^{n-1} e_i * B^i$. Chaque chiffre e_i représente le reste de la division entière de E/B^i par B. La base B est éventuellement précisée en indice à droite du dernier chiffre ou entre parenthèses. Par défaut, il s'agit de la base 10.

L'entier composé des k chiffres de poids faibles de E est E modulo [1] 2^k et celui composé des n-k chiffres de poids forts de E est E / 2^k. Exemple pour n=5 et k=2 :
$23_{10} = 5*4+3 = 10111_2$. ($\underline{101}\overline{11}$) : $23/4 = 5 = 101_2$. $23\ modulo\ 4 = 3 = 11_2$

Les organes d'un ordinateur sont dimensionnés à un nombre fixe n de bits. Par exemple, les registres, les unités de calcul, le bus d'accès à la mémoire d'un ARM7 sont tous dimensionnés à 32 bits : le résultat d'une opération est stocké avec le même nombre de bits que ses opérandes. Tous les calculs sont donc réalisés modulo 2^n (environ quatre milliards pour $n = 32$ bits).

La table en annexe **??** donne les principales puissances de 2, ainsi que la valeur binaire et décimale de chaque chiffre hexadécimal.

$$\begin{array}{llllll} 101_2 & = & 1\times 2^2 + 0\times 2^1 + 1\times 2^0 & = & 4+1 & = & 5_{10} \\ 101_{10} & = & 1\times 10^2 + 0\times 10^1 + 1\times 10^0 & = & 100+1 & = & 101_{10} \\ 101_{16} & = & 1\times 16^2 + 0\times 16^1 + 1\times 16^0 & = & 256+1 & = & 257_{10} \\ A4_{16} & = & 10\times 16^1 + 4\times 16^0 & = & 10*16+4 & = & 164_{10} \end{array}$$

Conversions entre bases 2, 10 et 16 :

1. 2 vers 16 : ajouter éventuellement des 0 à gauche pour avoir 4k bits, convertir les k quartets en k chiffres hexadécimaux.
2. 16 vers 2 : convertir chaque chiffre hexadécimal en quartet de bits
3. 10 vers B=2 ou B=16 : diviser par B, le reste donne un chiffre (poids faible), recommencer avec le quotient, etc : le dernier reste non nul donne le chiffre de poids fort.

Exemples :
— $178_{10} = B2_{16}$: $178/16\ quotient\ 11, reste\ \boxed{2}\ (poids\ faible), 11/16$: $quotient\ 0\ reste\ 11(\boxed{B}\ (poids fort))$
— $11_{10} = 1011_2 = 5*2+\boxed{1}\ (poids\ faible), 5 = 2*2+\boxed{1}, 2 = 1*2+\boxed{0}, 1 = 0*2+\boxed{1}\ (poids\ fort)$
— $1001101101_2 \rightarrow \mathbf{00}10\ 0110\ 1101 \rightarrow 26D_{16}$

1. modulo = reste de la division entière

2.1.1 Propriété remarquable

$$\sum_{i=0}^{n-1} a^i = \frac{a^n - 1}{a - 1} \; et \; \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

En effet $(a^{n-1} + a^{n-2} + \ldots + a^1 + 1)(a - 1) = (a^n - a^{n-1} + a^{n-1} - a^{n-2} \ldots + a - 1) = (a^n - 1))$

L'entier dont la représentation est constituée de n bits à 1 est $2^n - 1$: $11111111_2 = 2^8 - 1 = 255_{10}$

2.1.2 Compléments à 1 et à 2

Soit $E = \sum_{i=0}^{n-1} e_i * 2^i$ un entier naturel représenté sur n chiffres en base 2. On appelle *complément à* $2^n - 1$ de E (on dit habituellement *complément à 1 de E*) l'entier $\overline{E} = \sum_{i=0}^{n} \overline{e}_i$ obtenu en remplaçant les 1 par des 0 et les 0 par des 1 ($\overline{e}_i = 1 - e_i$) dans la représentation en binaire de E. Il s'écrit ˜E en langage C. On a $E + \overline{E} = \sum_{i=0}^{n} e_i 2^i + \sum_{i=0}^{n} (1 - e_i) 2^i = \sum_{i=0}^{n-1} 2^i = 2^n - 1$, d'où $\overline{E} = 2^n - 1 - E$.

On appelle *complément à* 2^n de E (on dit habituellement *complément à deux*) l'entier naturel $2^n - E$, noté $\overline{E}^2$. Par définition, $\overline{E}^2 = \overline{E} + 1$. Soit u la position du premier un [2] dans la représentation en binaire de E. La représentation de $\overline{E}^2$ est obtenue à partir de celle de E en inversant les $n - u$ bits de poids forts et en conservant les u bits de poids faibles.

2.2 Addition

On rappelle le principe de calcul dans l'addition : colonne par colonne, de droite à gauche. Les retenues, habituellement placées au dessus de l'opérande gauche, sont placées ici en dessous de l'opérande droit. Dans chaque colonne, on fait la somme des chiffres du premier (a_i) et du deuxième (b_i) opérande, ainsi que la retenue entrante (c_i).

Le chiffre (r_i) du résultat est égal à :
— cette somme, la retenue sortante (c_{i+1}) étant 0, si $somme < base$,
— cette somme moins la base, la retenue sortante (c_{i+1}) étant 1, si $somme \geq base$.

	a_3	a_2	a_1	a_0	opérande gauche
$+_{base}$	b_3	b_2	b_1	b_0	opérande droit
$C = c_4$	c_3	c_2	c_1	c_0	retenues
	r_3	r_2	r_1	r_0	résultat apparent

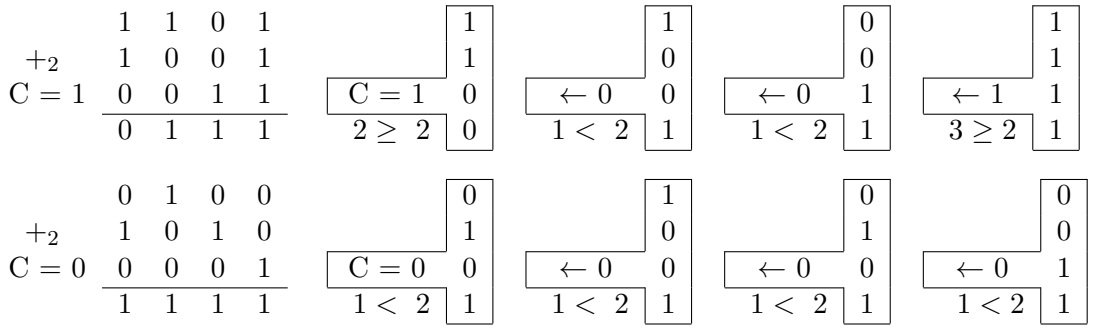
sortante c_{i+1} | a_i / b_i / c_i / r_i | entrante

	3	6	4	3		3		6		4		3				
$+_{10}$	5	7	8	5		5		7		8		5				
C = 0	1	1	0	0	C = 0	1	← 1	1	← 1	0	← 0	0				
	9	4	2	8	$9 < 10$	9	$14 \geq 10$	4	$12 \geq 10$	2	$8 < 10$	8				
	0	1	1	1		0		1		1		1				
$+_2$	0	1	1	0		0		1		1		0				
C = 0	1	1	0	0	C = 0	1	← 1	1	← 1	0	← 0	0				
	1	1	0	1	$1 < 2$	1	$3 \geq 2$	1	$2 \geq 2$	0	$1 < 2$	1				

Dans une addition normale, la retenue entrante initiale (c_0, colonne de droite) est nulle. L'utilisation d'une retenue initiale à 1 permet de calculer l'expression $op_{gauche} + op_{droit} + 1$ (pour réaliser des

2. $\forall i, e_i = 1 \Rightarrow i \geq u$, (u=0 si E=0)

soustractions par addition du complément à deux).



2.3 Conventions d'interprétation (entiers naturels et relatifs)

Soit $e = \sum_{i=0}^{i=n-2} e_i 2^i$. Sur n bits, on peut coder 2^n valeurs différentes. Mais l'interprétation de ce codage n'est pas unique. En pratique, l'entier écrit $e_{n-1}\ e_{n-2}\ e_{n-3} \dots\ e_1\ e_0$ en base deux représente la valeur $E = \alpha e_{n-1} 2^{n-1} + e$.

Les règles de calcul pour l'addition et la soustraction sont les mêmes quel que soit α : seule l'interprétation des valeurs des opérandes et du résultat change.

2.3.1 Pour entiers naturels (N) : $\alpha = 1$ et $E = \sum_{i=0}^{i=n-1} e_i 2^i$.

En pratique, il n'est pas rare que les entiers manipulés dans la vie courante sortent de l'intervalle de valeurs représentables dans les formats inférieurs à 64 bits. A titre d'exemple, les capitalisations boursières des sociétés ne sont pas toutes représentables sur 32 bits.

Pour stocker une valeur entière toujours positive ou nulle [3], le programmeur peut décider d'utiliser une variable entière en interprétant son contenu comme un entier naturel (attribut *unsigned* de type entier en langage C) afin de maximiser l'intervalle de valeurs représentables : $[0 \dots 2^n - 1]$.

Le bit de poids fort n'a pas de signification particulière : il indique simplement si la valeur représentée est supérieure à 2^{n-1} ou pas.

Dans le langage C, le type entier naturel est spécifié avec l'attribut **unsigned**, ou les types entier naturel de taille précise **uint***xx*_**t** ($x \in \{8, 16, 32, 64\}$) définis dans stdint.h (révisions récentes du langage).

La figure en cercle 2.1 illustre les $2^4 = 16$ codes binaires possibles sur 4 bits (incluses dans le cercle intérieur) et (sur la couronne extérieure,en décimal) les valeurs d'entiers naturels représentées.

Chaque entier correspond à un angle de rotation depuis l'origine dans le sens trigonométrique [4]. L'addition de 2 entiers peut être interprétée comme la sommation des angles de rotation des opérandes. A partir d'un tour complet, il y a débordement (résultat apparent obtenu modulo 2^4 et C=1 qui indique qu'il faudrait un bit de plus (à 1) pour représenter le vrai résultat).

3. Les constantes adresse et les variables pointeurs entrent dans cette catégorie.
4. ou anti-horaire

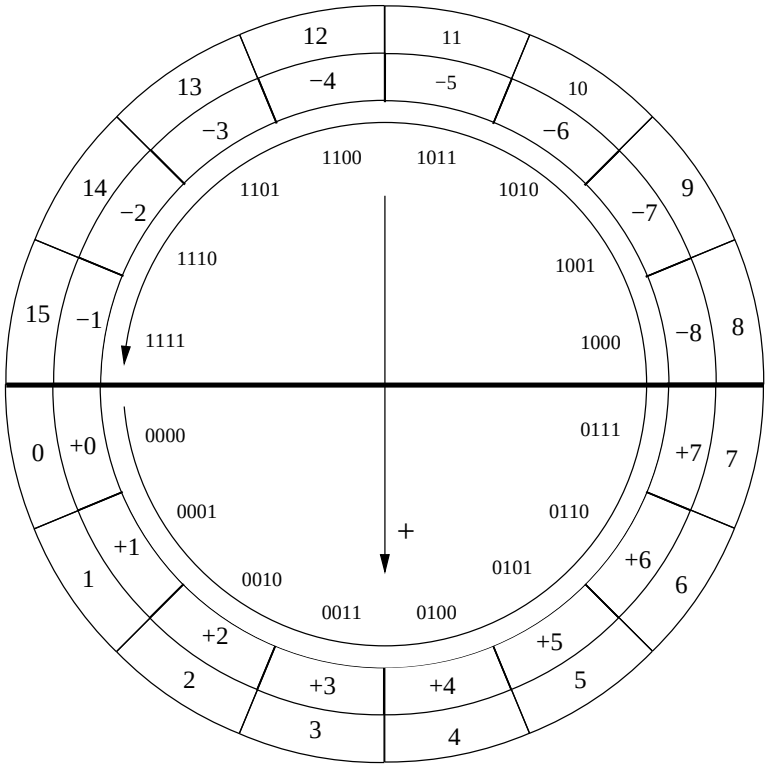


FIGURE 2.1 – Représentation d'entiers naturels et signés sur 4 bits

2.3.2 Une convention à oublier : signe et valeur absolue

La convention la plus intuitive pour l'ensemble Z est de coder le signe dans le bit de poids fort ($0 : \geq 0$, $1 : \leq 0$) et la valeur absolue sur les n-1 bits de poids faibles.

Elle présente deux inconvénients majeurs pour le codage des entiers [5] :
— La valeur 0 a deux codages (sur 4 bits : 0000 et 1000)
— l'addition des relatifs est traitée différemment de celle des naturels

C'est pourquoi les entiers relatifs sont codés selon la méthode du complément à 2.

2.3.3 Pour entiers relatifs (Z) par la méthode du complément à 2 : $\alpha = -1$ et $E = -e_{n-1}2^{n-1} + \sum_{i=0}^{i=n-2} e_i 2^i$.

Le bit de poids fort représente maintenant le signe de l'entier et le principe consiste à retrancher 2^n à la valeur associée aux entiers dont le bit de poids fort est à 1. Cette convention représente les entiers négatifs selon la technique du *complément à deux*[6] : l'entier relatif -x est représenté comme l'entier naturel $2^n - x$. Dans les langages, cette convention d'interprétation est généralement utilisée par défaut (type entier sans attribut unsigned ou type **int***xx*_**t** en langage C).

Un entier relatif E dont le bit de signe est 0 (≥ 0) appartient à l'intervalle $[0 \dots 2^{n-1} - 1]$ et sa valeur associée est la même que dans la convention pour entier naturels.

5. elle est utilisée dans certains formats de représentation des nombres à virgule flottante

6. La convention alternative "signe et valeur absolue" (resp. codé dans le bit de poids fort et codée sur les n-1 bits de poids faibles) a l'inconvénient de définir deux zéros : +0 et -0. Rarement utilisée pour les entiers, elle peut s'appliquer à la représentation des nombres à virgule flottante.

Un entier E dont le bit de signe est 1 (< 0) appartient à l'intervalle $[-2^{n-1}, +2^{n-1} - 1]$ et sa valeur associée est $-\overline{e}^2 = -(2^n - e)$.

— Pour calculer l'opposé d'un entier, il faut prendre le complément à deux de cet entier (et non inverser simplement le bit de signe).
— Sur n bits, l'entier -2^{n-1} est son propre complément à deux et l'entier relatif $+2^{n-1}$ n'est pas représentable.
— L'ajout d'un bit à 0 en poids fort d'un entier relatif négatif inverse son signe et change sa valeur.

La couronne intérieure de la figure 2.1 illustre le codage des entiers relatifs sur 4 bits. A chaque entier peut être associé un angle de rotation dans le sens trigonométrique pour les entiers positifs ou nuls et dans le sens horaire pour les entiers négatifs.

Le code 1001 peut représenter selon la convention d'interprétation soit l'entier naturel 9 soit l'entier relatif -7. De même, 0101 est le code commun aux entiers naturels 5 et relatif +5.

L'addition de deux entiers relatifs de même signe donne une erreur lorsque la somme des angles correspondant aux entiers va au-delà d'une rotation d'un demi-tour, avec un résultat apparent de signe opposé à celui des opérandes. Cette erreur vient du fait que le résultat attendu n'appartient pas à l'intervalle des valeurs représentables sur le nombre de bits de codage utilisé.

2.3.4 Intervalles représentables

n	Convention naturels			Convention relatifs		
n	0	à	$2^n - 1$	-2^{n-1}	à	$+2^{n-1} - 1$
8	0	à	255	-128	à	+127
16	0	à	65535 (64K_b-1)	-32768 (-32K_b)	à	+32767 (32K_b-1)
32	0	à	4294967295 (4G_b-1)	-2147483648 (-2G_b)	à	+2147483647 (2G_b-1)
64	0	à	$1,8 \times 10^{19}(16E_b - 1)$	$-9 \times 10^{18}(-4E_b)$	à	$+9 \times 10^{18}(+4E_b - 1)$

Pour les bornes de l'intervalle sur 64 bits, le tableau mentionne l'ordre de grandeur (préfixé par) : la valeur exacte représente une vingtaine de chiffres. Les préfixes K_b (kilo) et M_b (méga) représentent $2^{10} = 1024_{10}$ et $2^{20} = 1048576_{10}$, dont la valeur est proche de 1000 (1K) et 1000000 (1M). Même principe pour G_b (giga : 2^{30}) et E_b (eta : 2^{60}).

2.4 Changement de format, manipulation booléenne des bits, décalages

2.4.1 Extension et réduction de format

Des conversions de taille sont nécessaires lors de la copie d'un entier entre 2 contenants de tailles différentes, par exemple un registre de 32 bits et un emplacement mémoire de 8 ou 16 bits.

La réduction de format élimine les bits de poids forts excédentaires (opération modulo). La valeur entière n'est pas modifiée si elle est représentable sur le contenant de plus petite taille.

En sens inverse, la représentation de l'entier doit être étendue en ajoutant des bits de poids forts selon la nature de l'entier :

— ajout de bits à 0 pour un entier naturel
— duplication de l'ancien bit de poids fort (bit de signe) pour un entier relatif

Il existe ainsi 3 instructions ARM de transfert d'un entier entre un registre 32 bits **regx** et un emplacement de 16 bits en mémoire **mem[y]**. L'instruction ldrh est destinée aux entiers naturels codés sur 16 bits, et ldrsh aux entiers relatifs.
— strh : regx modulo $2^{16} \rightarrow mem[y]$
— ldrh : regx $\underleftarrow{extension\ en\ ajoutant\ 16\ fois\ un\ bit\ 0}$ mem[y]
— ldrsh : regx $\underleftarrow{extension\ en\ ajoutant\ 16\ fois\ le\ bit\ de\ poids\ fort\ de}$ mem[y]

2.4.2 Décalage et rotation

Le décalage logique à gauche de k bits (Logic Shift Left #k en langage d'assemblage ARM, $\ll\ k$ en C) d'un entier e ajoute k bits à 0 à droite, ce qui revient à multiplier e par 2^k. Le format de représentation restant inchangé, le décalage supprime les k bits de poids forts de e. Si l'un de ces bits éjectés n'est pas 0, le résultat de la multiplication n'est pas représentable sur le nombre de bits utilisé.

Remarque : l'entier 2^k est l'entier 1 décalé de k bits à gauche (((uint32_t)1 $\ll\ k$) en C).

Une opération de rotation est un décalage dans lequel les bits ajoutés à une extrémité sont ceux qui sont éjectés de l'autre extrémité de l'entier. Une rotation à gauche de k bits et une rotation à droite de $n - k$ bits ont le même effet.

Le décalage de k bits à droite correspond à la division par 2^k : les k bits de poids faibles sont éjectés.

Le décalage logique de k bits à droite (Logic Shift Right #k en langage d'assemblage ARM, $\gg k$ sur une variable unsigned en C) est destiné aux entiers naturels : k bits à 0 sont ajoutés en poids forts et l'entier est divisé par 2^k.

Le décalage arithmétique à droite (Arithmetic Shift Right en langage d'assemblage ARM) est destiné aux entiers relatifs : le bit de poids fort (signe) d'origine est recopié dans les bits ajoutés à gauche. L'entier est divisé par 2^k s'il en était un multiple au départ.

2.4.3 Opérations booléennes bit à bit

Un chiffre de la base 2 (bit d'un entier) et un booléen ont la même écriture : 0 ou 1.
Les opérateurs bit à bit traitent un entier sur n bits comme une collection de n booléens : chaque bit de rang j du résultat correspond à une opération booléenne sur les bits de rang j des opérandes.

La négation (un seul opérande) bit à bit (~ en C) inverse tous les bits de l'entier : elle réalise le complément à 1 de celui-ci.

Les autres opérations booléennes classiques à 2 deux opérandes existent aussi en version bit à bit :
— Et bit à bit (& en C)
— Ou bit à bit (— en C)
— Ou exclusif bit à bit (^ en C, remarquer que ce n'est pas l'opérateur d'élévation à la puissance)

Noter la différence avec les opérateurs booléens classiques du C :
— 11 && 13 donne 1 ($11 \neq 0$) : vrai, $13 \neq 0$: vrai, vrai et vrai : vrai $\rightarrow$ 1
— 11 & 13 donne 9 : seuls les bits 0 et 3 sont à 1 dans les deux entiers.

2.4.4 Exemples d'application

Tous les entiers des exemples suivant sont supposés déclarés de type unsigned.

Tester si le bit b de e est à 1 (3 méthodes différentes) :
— décaler une copie de e de b bits à droite, puis ET bit à bit avec $2^0 = 1$: si $\neq 0$, le bit à tester est 1 ou
— décaler une copie de e pour amener le bit b en poids fort : le résultat interprété comme un entier relatif est < 0 si le bit b de e est à 1 ou
— ET bit à bit entre e et 1 décalé à gauche de b bits : bit testé à 1 si résultat non nul

Création de masque : entier dont les bits x à y ($y \geq x$) inclus sont à 0, les autres à 1 (utilisé dans un ET bit à bit, permet de forcer à 0 les bits x à y d'un entier) :

1. Complément à 1 de 0 (~0) : entier composé uniquement de bits à 1
2. décalage à gauche pour supprimer les bits à 1 au-delà du rang y
3. décalage à droite pour supprimer les bits à 1 en deçà du rang x
4. décalage à gauche pour ramener le premier bit à 1 au rang x
5. complément à 1 : inversion de tous les bits

2.5 Soustraction

Dans chaque colonne, on fait la somme du chiffre du deuxième (b_i) opérande et de l'emprunt entrant (e_i) et l'emprunt entrant initial e_0 est nul. Le chiffre (r_i) du résultat est égal :
— au chiffre du premier opérande (a_i) moins cette somme, l'emprunt sortant (e_{i+1}) étant 0, si $somme \leq a_i$,
— au chiffre du premier opérande (a_i) plus la base moins cette somme, l'emprunt sortant (e_{i+1}) étant 1, si $somme > a_i$,

	a_3	a_2	a_1	a_0	opérande gauche			a_i								
$+_{base}$	b_3	b_2	b_1	b_0	opérande droit			b_i								
$E = e_4$	e_3	e_2	e_1	e_0	emprunts	sortant	e_{i+1}	e_i	entrant							
	r_3	r_2	r_1	r_0	résultat apparent			r_i								

	8	6	4	8			8		6		4		8	
$-_{10}$	5	7	9	5			5		7		9		5	
E = 0	1	1	0	0		E = 0	1		$\leftarrow 1$	1	$\leftarrow 1$	0	$\leftarrow 0$	0
	2	8	5	3		$6 \leq 8$	2		$8 > 6$	8	$9 > 4$	5	$5 \leq 8$	3

	1	1	0	1			1		1		0		1	
$-_2$	0	1	1	0			0		1		1		0	
E = 0	1	1	0	0		E = 0	1		$\leftarrow 1$	1	$\leftarrow 1$	0	$\leftarrow 0$	0
	0	1	1	1		$0 \leq 1$	0		$2 > 1$	1	$1 > 0$	1	$0 \leq 1$	1

	0	1	0	0			0		1		0		0	
$-_2$	0	1	0	1			0		1		0		1	
E = 1	1	1	1	0		E = 1	1		$\leftarrow 1$	1	$\leftarrow 1$	1	$\leftarrow 1$	0
	1	1	1	1		$1 > 0$	0		$2 > 1$	1	$1 > 0$	1	$1 > 0$	1

2.6 Soustraction par addition du complément à deux

En pratique, toutes les soustractions sont réalisées par addition du complément à 2. On exploite la propriété suivante (calculs sur n bits) : $x + \overline{y}^2 = x + 2^n - y$.

Les résultats étant obtenus modulo 2^n, on peut calculer l'expression $x-y$ en effectuant une addition comme suit :

- Premier opérande : x
- Deuxième opérande : $\overline{y}$
- Retenue initiale : 1 (pour faire $x + \overline{y} + 1$)
- On observe que la ligne des retenues dans cette addition de $\overline{y}$ est le complément de la ligne des emprunts dans la soustraction normale.

Le calcul de $13 - 6$ (réalisable) et $4 - 5$ (impossible pour des entiers naturels) est illustré par les deux derniers exemples des paragraphes 2.5 (soustraction normale) et 2.2 (soustraction par addition du complément à deux).

2.7 Indicateurs et débordements

Lors d'une opération (addition ou soustraction) sur les entiers, l'unité de calcul d'un processeur synthétise quatre indicateurs booléens à partir desquels il est possible de prendre des décisions.

2.7.1 Nullité et indicateur : Z

L'indicateur Z (**Z**éro) et vrai si et seulement tous les bits du résultat apparent sont à 0, ce qui signifie que ce dernier est nul.

2.7.2 Signe du résultat apparent : N

L'indicateur N est égal au bit de poids fort du résultat apparent. Si ce dernier est interprété comme un entier relatif, N=1 signifie que le résultat apparent est négatif.

2.7.3 Débordement en convention d'entiers naturels : C

L'indicateur C (**C**arry) est la dernière retenue sortante de l'addition. Il n'a de sens que dans une interprétation de l'opération sur des entiers naturels.

Après une addition, $C = 1$ indique un débordement : le résultat de l'opération est trop grand pour être représentable sur n bits. Le résultat apparent est alors faux : il correspond au vrai résultat à 2^n près.

E est le dernier emprunt sortant d'une soustraction. $E = 1$ indique que la soustraction est impossible parce que le deuxième opérande est supérieur au premier. Les soustractions sont en pratique réalisées par addition du complément à deux. C correspond alors à $\overline{E}$. Après une soustraction par addition du complément à deux, $C = 0$ indique que la soustraction est impossible, $C = 1$ que l'opération est correcte [7].

7. Attention : les instructions de soustraction ou de comparaison de certains processeurs (dont le SPARC) stockent dans C le **complément** de la retenue finale. Pour ces processeurs, $C = 1$ indique toujours une erreur, que ce soit après une addition ou une soustraction.

2.7.4 Débordement en convention d'entiers relatifs : V

Pour les entiers, la soustraction est toujours réalisée par addition de l'opposé du deuxième opérande.

La valeur absolue de la somme de deux entiers relatifs de signes opposés est inférieure ou égale à la la plus grande des valeurs absolues des opérandes et le résultat est toujours représentable sur n bits. La somme de deux entiers relatifs de même signe peut ne pas être représentable sur n bits, auquel cas le résultat apparent sera faux :

— Sa valeur n'est égale à celle du vrai résultat de l'opération qu'à 2^n près.
— Son bit de signe (bit de poids fort) est également faux : la somme de deux entiers positifs donnera un résultat apparent négatif et la somme de deux entiers négatifs donnera un résultat apparent positif ou nul.

L'indicateur V (o**V**erflow [8]) est l'indicateur de débordement destiné à la convention d'interprétation pour entiers relatifs. $V = 1$ indique un débordement, auquel cas les deux dernières retenues sont de valeurs différentes.

	0	0	1	1	+3
$+_2$	1	0	1	1	-5
V=0	$0 = 0$	1	1	0	
	1	1	1	0	-2
	1	0	1	0	-6
$+_2$	1	1	0	0	-4
V=1	$1 \neq 0$	0	0	0	
	0	1	1	0	+6

	0	1	1	0	+6
$+_2$	0	1	0	0	+4
V=1	$0 \neq 1$	0	0	0	
	1	0	1	0	-6

Le signe du vrai résultat (sans erreur) de l'opération s'écrit : $V \oplus N = \overline{V}.N + V.\overline{N}$. Ainsi, le signe du résultat de l'opération sans erreur est N signe du résultat apparent s'il n'y a pas de débordement ($\overline{V}$), ou le signe opposé $\overline{N}$ de celui du résultat apparent en cas de débordement (V).

2.7.5 Expressions des conditions avec les indicateurs ZNCV

Après synthèse des indicateurs lors du calcul de $x - y$, il est possible de tester diverses conditions.

Par exemple ,l'expression de la condition "strictement inférieur" ($x < y$) est :
— $\overline{C}$ si x et y sont considérés comme des entiers naturels (la soustraction est impossible)
— $V \oplus N$ si x et y sont considérés comme des entiers relatifs (le vrai résultat est négatif).

2.8 Exercices

2.8.1 Addition d'entiers naturels

Quels entiers naturels peut-on représenter sur 4 bits ?

Choisir deux entiers naturels représentables sur 4 bits (si vous n'avez pas d'idées, prendre 11 et 13), faire la somme en faisant apparaître les retenues propagées. Quand la somme n'est-elle pas représentable sur 4 bits ?

On pourra reprendre l'exercice pour des nombres représentés sur 8, 16 ou 32 bits...

8. L'initiale O n'a pas été retenue pour éviter une confusion avec zéro

2.8.2 Représentation des entiers relatifs en *complément à deux*

Quels entiers relatifs peut-on représenter sur 4 bits ? Donner pour chacun leur codage en complément à 2.

Quels entiers relatifs peut-on représenter sur 8 bits ? Comment s'y prendre pour coder un entier relatif en complément à 2 sur 8 bits ? Comment passer d'un relatif négatif à son opposé ?

Choisir un entier relatif positif représentable sur 4 bits. Donnez sa représentation sur 8 bits.

Choisir un entier relatif négatif représentable sur 4 bits. Donnez sa représentation sur 8 bits.

2.8.3 Addition d'entiers relatifs

Choisir deux entiers relatifs un positif et un négatif représentables sur 4 bits (si vous n'avez pas d'idées, prendre 6 et -4), faire la somme. Quand la somme n'est-elle pas représentable sur 4 bits ?

Choisir deux entiers relatifs positifs représentables sur 4 bits, faire la somme. Identifier les cas où la somme n'est pas représentable sur 4 bits ?

Même question pour deux entiers relatifs négatifs.

On pourra reprendre les exercices pour des nombres représentés sur 8, 16 ou 32 bits...

2.8.4 Soustraction de naturels

Choisir deux entiers naturels représentables sur 4 bits (si vous n'avez pas d'idées, prendre 11 et 13), faire la différence. Quand la différence n'est-elle pas représentable sur 4 bits ?

Pour comparer deux nombres a et b on peut calculer la différence $a - b$; $a > b$ ssi $a - b > 0$.

Dans le tableau de la figure 2.3 de votre documentation retrouvez les lignes correspondant à des comparaisons $(>, <, \leq, \geq)$ de nombres dans $\mathbb{N}$. Faire le lien avec la réponse que vous avez donnée précédemment.

2.8.5 Comparaisons d'entiers relatifs

Choisir deux entiers relatifs représentables sur 4 bits (si vous n'avez pas d'idées, prendre 6 et -4), faire la différence. Exprimer quand la différence n'est pas représentable est un peu plus complexe : on trouve les expressions logiques nécessaire dans le tableau de la figure 2.3 de votre documentation. Prendre un exemple par exemple le cas $\leq$ et chercher des entiers relatifs correspondant à chacun des cas de l'expression $Z \vee ((N \wedge \overline{V}) \vee (\overline{N} \wedge V))$.

2.8.6 Multiplier et diviser par une puissance de deux

Choisir un entier naturel n représentable sur 8 bits. Quelle est la représentation de $2 * n$, de $4 * n$, de $8 * n$? Quelle est la représentation de $n/2$, de $n/4$, de $n/8$?

Choisir un entier relatif (essayer avec un positif puis avec un négatif) x représentable sur 8 bits. Quelle est la représentation de $2 * x$, de $4 * x$, de $8 * x$? Quelle est la représentation de $x/2$, de $x/4$, de $x/8$?

TD séances 3 et 4 : Langage machine, codage des données

3.1 Sujet global des TD 3 et 4

On considère l'instruction : `x := (a + b + c) - (x - a - 214)`.

`x, a, b` et `c` sont des variables représentées sur 32 bits et rangées en mémoire aux adresses (fixées arbitrairement) : `0x50f0` `0x2fa0`, `0x3804`, `0x4050`.

Il existe un espace mémoire libre à partir de l'adresse `0x6400`.

On veut écrire un programme en langage machine qui exécute l'instruction considérée. Le programme ne doit pas changer les valeurs des variables `a, b` et `c` (i.e. ne doit pas changer le contenu des cases mémoire correspondantes).

Exercice : Dans chacun des langages machines décrits dans la suite, écrire systématiquement le programme qui exécute l'instruction ci-dessus.

3.2 Un premier style de langage machine : machine dite à accumulateur

La figure 3.1 donne la structure de la machine. Cette machine possède un registre spécial appelé accumulateur (on notera `ACC`) utilisé dans les opérations à la fois comme un des deux opérandes et pour stocker le résultat.

Dans une telle machine une instruction de calcul est formée du code de l'opération à réaliser (addition ou soustraction) et de la désignation d'un opérande. Il y a deux façons de désigner une information : on donne son adresse en mémoire ou on donne une valeur.

instruction	opération réalisée
`add adr`	`ACC <-- ACC + MEM[adr]`
`add# vi`	`ACC <-- ACC + vi`
`sub adr`	`ACC <-- ACC - MEM[adr]`
`sub# vi`	`ACC <-- ACC - vi`

Par ailleurs, on peut aussi charger une information dans l'accumulateur depuis la mémoire ou avec une valeur appelée **valeur immédiate**.

instruction	opération réalisée
`load adr`	`ACC <-- MEM[adr]`
`load# vi`	`ACC <-- vi`

Et enfin, on peut ranger la valeur contenue dans l'accumulateur en mémoire :

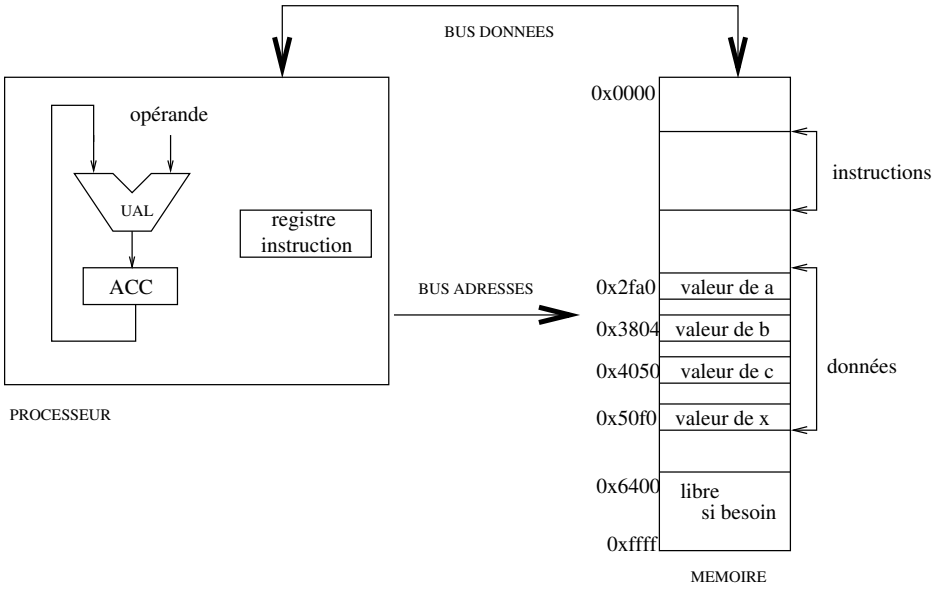


FIGURE 3.1 – Structure d'une machine à accumulateur

instruction	opération réalisée
`store adr`	`MEM[adr] <-- ACC`

1. Calculer la taille du programme si on suppose que les adresses sont représentées sur 16 bits (2 octets), les valeurs immédiates sont aussi représentées sur 2 octets et le code instruction est lui codé sur 1 octet.
2. Quelle est la différence entre `sub 0x2fa0` et `sub# 214` ?
3. Une instruction `store# 6` a-t-elle une signification ?
4. Ecrire un programme qui réalise le même calcul en commençant par évaluer le second membre de la soustraction globale.

Les microprocesseurs des années 70/80 ressemblent à ce type de machine : type 6800, 6501 (APPLE 2), Z80. Il en existe encore dans les petits automatismes, les cartes à puce, ... Les adresses sont souvent sur 16 bits, les instructions sur 1,2,3,4 octets, le code opération sur 1 octet.

3.3 Machine avec instructions à plusieurs opérandes

On va s'intéresser maintenant à une machine dans laquelle on indique dans l'instruction : le code de l'opération à réaliser, un opérande dit destination et deux opérandes source.

On pourrait imaginer une instruction de la forme : `add adr1, adr2, adr3` dont la signification serait : `mem[adr1] <-- mem[adr2] + mem[adr3]`.

Cela coûterait cher en taille de codage d'une instruction (6 octets pour les adresses si une adresse est sur 2 octets + le reste) mais surtout en temps d'exécution d'une instruction (3 accès mémoire).

Dans ce type de machine, il y a en fait des registres, proches du processeur et du coup d'accès plus rapide. On peut y stocker les informations avec lesquelles sont faits les calculs (Cf. figure 3.2). Il y a de plus des opérations de transfert d'information de la mémoire vers les registres (et inversement).

Les registres sont repérés par des numéros. On note `reg5` le registre de numéro 5 par exemple. On notera aussi `reg5` la valeur contenue dans le registre.

Une instruction de calcul est formée du code de l'opération à réaliser, et de la désignation des registres intervenant dans le calcul. On trouve deux formes de telles instructions :

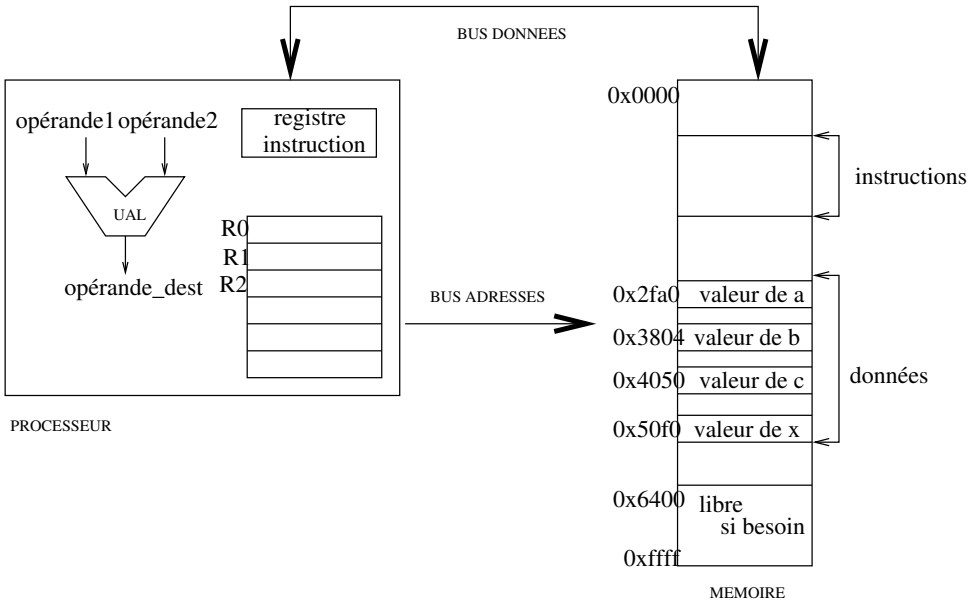


FIGURE 3.2 – Structure d'une machine générale à registres

— deux opérandes sources dans des registres (on écrira regs1 et regs2) et un registre pour le résultat du calcul (on écrira regd pour registre destination).
— un opérande source dans un registre et l'autre donné dans l'instruction (valeur immédiate) et toujours un registre destination.

instruction	opération réalisée
`add regd regs1 regs2`	`regd <-- regs1 + regs2`
`sub regd regs1 regs2`	`regd <-- regs1 - regs2`
`add# regd regs1 vi`	`regd <-- regs1 + vi`
`sub# regd regs1 vi`	`regd <-- regs1 - vi`

Au niveau du codage, il faut coder : le code de l'opération à réaliser et les numéros des registres. Par exemple sur ARM il y a 16 registres, d'où 4 bits pour coder leur numéro.

Nous avons besoin aussi d'effectuer des transferts entre mémoire et registres. En général, dans ce genre de machine les adresses (et les données) sont représentées sur 32 bits (question d'époque...). Le problème est que pour représenter l'instruction *amener le mot mémoire d'adresse 0x2fa0 dans le registre 2*, il faut : 1 codeop + 1 numéro de registre sur x bits + 1 adresse (0x2fa0) sur 32 bits pour former l'instruction... codée elle aussi sur 32 bits.

Les opérations de transfert sont réalisées en deux étapes : mettre l'adresse du mot mémoire concerné dans un registre (ci-dessous reg1) puis charger un registre avec le contenu du mot mémoire à cette adresse (load) ou ranger le contenu du mot mémoire à cette adresse dans un registre (store).

instruction	opération réalisée
`METTRE reg1, adr`	`reg1 <-- adr`
`load reg2, [reg1]`	`reg2 <-- Mem[reg1]`
ou	
`METTRE reg1, adr`	`reg1 <-- adr`
`store [reg1], reg2`	`Mem[reg1] <-- reg2`

1. Si on suppose qu'une instruction est codée sur 4 octets, quelle est la taille du programme ?
2. Discuter de la taille de codage des numéros de registres.

3. Discuter de la taille de codage des valeurs immédiates.
4. Pourquoi en général n'y a-t-il qu'une valeur immédiate ?

Les microprocesseurs des années 90 sont de ce type : machines RISC, type Sparc, ARM. Les adresses sont en général sur 32 bits, toutes les instructions sont codées sur 32 bits, et il y a beaucoup de registres.

Remarque : Attention, pour le processeur ARM, dans la syntaxe de l'instruction `store` les opérandes sont inversés par rapport au choix fait ci-dessus ; on écrit `str reg2, [reg1]`. Ainsi l'ordre d'écriture des opérandes est le même pour l'instruction store (`str`) et l'instruction load (`ldr`).

Dans les années 70/80 il y a eu des processeurs (pas micro du tout) de type VAX (inspirés de, avec beaucoup de variantes). Une instruction peut être codée sur 4 mots de 32 bits et donc contenir 3 adresses.

Il a été construit dans les années 80/90 des microprocesseurs avec deux opérandes pour une instruction : un opérande source servant aussi de destination (type 68000, 8086). Les adresses sont sur 16, 24 ou 32 bits, les instructions sur 1,2,3 ou 4 mots de 16 bits. Le code opération est généralement sur 1 mot de 16 bits. Il y a 8 ou 16 registres.

3.4 Codage de `METTRE` ?

Il reste à comprendre comment coder : `METTRE une adresse de 32 bits dans un registre ?`

Même si on n'a plus que le code de METTRE, un seul numéro de registre, l'adresse reste sur 32 bits et ça ne tient toujours pas...

Par exemple, on veut coder : `charger reg2 avec le mot mémoire d'adresse 0x2fff2765`. On va donc coder : `METTRE reg1, 0x2fff2765` puis `load reg2, [reg1]`.

Chercher plusieurs solutions.

4.1 Machine ARM : accès et opérations sur les données en mémoire

On converge vers le programme (plus loin) écrit en langage d'assemblage `ARM` qui exécute l'instruction : `x := (a + b + c) - (x - a - 214)`.

Pour traduire le programme en binaire en fixant les adresses de début de la zone `text` et de la zone `data` on utilise :

```
arm-eabi-as -o exp_arm.o exp_arm.s -mbig-endian
arm-eabi-ld -o exp_arm exp_arm.o -e main -Ttext 0x800000 -Tdata 0x0 -EB
```

La zone `text` étant stockée à partir de l'adresse `0x800000` (option `-Ttext 00800000`) et la zone `data` à partir de l'adresse `00000000` (option `-Tdata 0x0`).

Dans la pratique, ce n'est pas nous qui fixons les adresses, mais les outils de traduction et/ou de chargement en mémoire. Pour simuler ce fonctionnement et fixer les adresses que l'on a choisi en assembleur, on peut utiliser des étiquettes et des directives `.org`. La directive `.org` permet de fixer l'adresse relative où sera stockée la valeur qui suit. Par exemple, le mot étiquetté `a` sera rangé à l'adresse de début de la zone `data` + 0x2fa0.

1. Ajouter des commentaires au programme explicitant chacune des lignes de code.
2. Dessiner le contenu de la zone de données en exprimant les valeurs des différentes données en hexadécimal. Vous pouvez regarder la traduction obtenue par les commandes `as` et `ld` (après le programme) et vérifier la cohérence avec votre dessin.

Le programme `ARM` :

```
1     .text
2     .global main
3  main: ldr r1, LD_a
4        ldr r1, [r1]
5        ldr r2, LD_b
6        ldr r2, [r2]
7        ldr r3, LD_c
8        ldr r3, [r3]
9        add r4, r1, r2
10       add r4, r4, r3
11       ldr r2, LD_x
12       ldr r3, [r2]
13       sub r3, r3, r1
14       sub r3, r3, #214
15       sub r4, r4, r3
16       str r4, [r2]
17       bx lr
18       .org 0x1000
19 LD_a: .word a
20 LD_b: .word b
21 LD_c: .word c
22 LD_x: .word x
23    .data
24    .org 0x2fa0
25 a:    .word 10
26    .org 0x3804
27 b:    .word 20
28    .org 0x4050
29 c:    .word 30
30    .org 0x50f0
31 x:    .word 1000
```

Zone text :

```
$ arm-eabi-objdump -d -j .text exp_arm

exp_arm:     file format elf32-bigarm

Disassembly of section .text:

00800000 <main>:
  800000:   e59f1ff8    ldr   r1, [pc, #4088]   ; 801000 <LD_a>
  800004:   e5911000    ldr   r1, [r1]
  800008:   e59f2ff4    ldr   r2, [pc, #4084]   ; 801004 <LD_b>
  80000c:   e5922000    ldr   r2, [r2]
  800010:   e59f3ff0    ldr   r3, [pc, #4080]   ; 801008 <LD_c>
  800014:   e5933000    ldr   r3, [r3]
  800018:   e0814002    add   r4, r1, r2
  80001c:   e0844003    add   r4, r4, r3
  800020:   e59f2fe4    ldr   r2, [pc, #4068]   ; 80100c <LD_x>
  800024:   e5923000    ldr   r3, [r2]
  800028:   e0433001    sub   r3, r3, r1
  80002c:   e24330d6    sub   r3, r3, #214   ; 0xd6
  800030:   e0444003    sub   r4, r4, r3
  800034:   e5824000    str   r4, [r2]
  800038:   e1a0f00e    bx lr
   ...

00801000 <LD_a>:
  801000:   00002fa0    andeq r2, r0, r0, lsr #31
```

```
00801004 <LD_b>:
  801004:   00003804    andeq r3, r0, r4, lsl #16

00801008 <LD_c>:
  801008:   00004050    andeq r4, r0, r0, asr r0

0080100c <LD_x>:
  80100c:   000050f0    streqd   r5, [r0], -r0
```

Zone data :

```
$  arm-eabi-objdump -s -j .data exp_arm

exp_arm:     file format elf32-bigarm

Contents of section .data:
 0000 00000000 00000000 00000000 00000000  ................
 ...
 2f90 00000000 00000000 00000000 00000000  ................
 2fa0 0000000a 00000000 00000000 00000000  ................
 2fb0 00000000 00000000 00000000 00000000  ................
 ...
 2fc0 00000000 00000000 00000000 00000000  ................
 ...
 37f0 00000000 00000000 00000000 00000000  ................
 3800 00000000 00000014 00000000 00000000  ................
 3810 00000000 00000000 00000000 00000000  ................
 ...
 4040 00000000 00000000 00000000 00000000  ................
 4050 0000001e 00000000 00000000 00000000  ................
 4060 00000000 00000000 00000000 00000000  ................
 ...
 4070 00000000 00000000 00000000 00000000  ................
 ...
 50e0 00000000 00000000 00000000 00000000  ................
 50f0 000003e8                             ....
```

1. En fin de zone `text` on trouve le binaire correspondant aux déclarations des adresses en zone `data`. Repérez les valeurs (attention : ce sont des adresses) associées aux étiquettes `LD_a`, `LD_b`, `LD_c` et `LD_x`.
2. Retrouvez les valeurs rangées à ces adresses dans la zone `data`.
3. En utilisant la documentation technique, donnez la traduction de l'instruction `ldr r1, LD_a` ? Etudiez le codage binaire de cette instruction et retrouvez-y les différents éléments : le code de `ldr`, le code des registres `r1` et `pc` et la valeur du déplacement.
4. Quelle est la traduction de l'instruction `ldr r1, [r1]` ? Etudiez le codage binaire de cette instruction et retrouvez-y les différents éléments : le code de `ldr`, le code des registres `r1` et `r1` et la valeur du déplacement.
5. Comprendre le déplacement codé dans l'instruction `ldr r1, LD_a` ?
6. Recommencer le même travail avec l'instruction `ldr r2, LD_x` ?

4.2 Codage des tableaux

Pour les chaines de caractères et les tableaux "simples" (à 1 dimension), le placement en mémoire est immédiat : à partir d'une adresse de départ qui est associée à la chaine de caractère, resp. au tableau, les caractères, resp. les éléments de tableaux, sont disposés dans la mémoire à la suite. Pour les caractères, octet après octet. Pour un tableau d'entiers 32 bits, chaque entier sera donc placé sur un bloc de 4 octets en suivant la convention grand-boutiste (big-endian) ou petit-boutiste (little-endian) adoptée.

1. Combien faudra-t-il de place pour placer la chaine de caractère "Bonjour !" en mémoire (prendre en compte le codage de fin de chaîne par un 0).
2. Si l'adresse de départ est multiple de 4, quelle sera la prochaine adresse multiple de 4 ? (en déduire la place occupée en pratique).
3. Dessiner la mémoire obtenue pour placer la chaine de caractère "Bonjour" à partir de l'adresse 0x8080
4. Dessiner la mémoire obtenue pour placer le tableau [0x12345678,0xAABBCCDD,0x0,0x1] à partir de l'adresse 0xA000 (par exemple avec la convention grand-boutiste).
5. Quelles sont les valeurs du tableau vues par Bob si Bob pense que le tableau a été écrit avec la convention petit-boutiste) ?

Pour les tableaux à 2 dimensions, c'est presque la même chose, il suffit de voir le tableau comme un tableau à une dimension de tableaux à une dimension : soit un tableau de lignes, soit un tableau de colonnes.

1. Dessiner la mémoire obtenue pour placer le tableau ci dessous comme tableau de lignes
2. Dessiner la mémoire obtenue pour placer le tableau ci dessous comme tableau de colonnes
3. Dans un tableau d'entiers (32 bits) de 100 lignes et 1000 colonnes quelle sera la distance (en nombre d'octets) entre 2 éléments voisins sur une même ligne si la tableau a été placé en mémoire en utilisant la disposition "tableau de lignes".
4. Même question pour la disposition "tableau de colonnes".
5. Idem pour 2 entiers voisins dans une même colonne pour les deux dispositions.

0x01020304	0x05060708	0x090A0B0C	0x0D0E0F10
0x11121304	0x15161718	0x191A1B1C	0x1D1E1F10
0x21222324	0x25262728	0x292A2B2C	0x2D2E2F10

FIGURE 4.1 – Tableau de nombres

TD séances 5 et 6 : Codage des structures de contrôle

5.1 Codage d'une instruction conditionnelle

On veut coder l'algorithme suivant : `si a = b alors c <-- a-b sinon c <-- a+b`.

L'évaluation de l'expression booléenne `a = b` est réalisée par une soustraction `a-b` dont le résultat ne nous importe guère ; on veut juste savoir si le résultat est 0 ou non. Pour cela on va utiliser l'indicateur `Z` du code de condition arithmétique positionné après une opération :
`Z = 1` si et seulement si le résultat est nul
`Z = 0` si et seulement si le résultat n'est pas nul.

De plus nous allons utiliser l'instruction de rupture de séquence `BCond` qui peut être conditionnée par les codes de conditions arithmétiques `Cond` (`EQ`, `NE`, `GT`, `GE`, ...)

On peut proposer beaucoup de solutions dont les deux suivantes assez classiques :

```
@ a dans r4, b dans r5, c dans r6
       CMP r4, r5 @ a-b ??            CMP r4, r5 @ a-b ??
       BNE sinon                      BEQ alors
alors:
      @  a completer                @  a completer
       B finsi                        B finsi
sinon:                         alors:
      @  a completer                 @  a completer
finsi:                         finsi:
```

Exercices :

1. Compléter et commenter les 2 programmes. Comprendre l'évolution du contrôle (compteur de programme, valeur des codes de conditions arithmétiques) pour chacune des deux solutions. Prendre deux exemples, l'un avec le test valide, l'autre avec le test faux.
2. Quel est l'effet du programme suivant :

   ```
          CMP r4, r5
          BNE sinon
          SUB r6, r4, r5
   sinon: ADD r6, r4, r5
   ```

3. Coder en langage d'assemblage ARM l'algorithme suivant :

   `si x est pair alors x <-- x div 2 sinon x <-- 3 * x + 1`

   la valeur de la variable `x` étant rangée dans le registre `r7`.

5.2 Notion de tableau et accès aux éléments d'un tableau

Considérons la déclaration de tableau suivante :
`TAB : un tableau de 5 entiers représentés sur 32 bits`.
Il s'agit d'un ensemble d'entiers stockés dans une zone de mémoire contiguë de taille 5×32 bits (ou 5×4 octets). La déclaration en langage d'assemblage d'une telle zone pourrait être :
`debutTAB: .skip 5*4`
où `debutTAB` représente l'adresse du premier élément du tableau (considéré comme l'élément numéro 0). `debutTAB` est aussi appelée adresse de début du tableau.

Quelle est l'adresse du 2^{eme} élément de ce tableau ? du 3^{eme} ? du i^{eme}, $0 \leq i \leq 4$?

On s'intéresse à l'algorithme suivant :

```
TAB[0] <-- 11
TAB[1] <-- 22
TAB[2] <-- 33
TAB[3] <-- 44
TAB[4] <-- 55
```

Les deux premières affectations peuvent se traduire :

```
1     .bss
2  debutTAB: .skip 5*4
3
4     .text
5     .global main
6  main:
7     ldr r4, LD_debutTAB
8     mov r5, #11
9     str r5, [r4]
10
11    mov r5, #22
12    add r6, r4, #4  @ *
13    str r5, [r6]    @ *
14
15    @ a completer
16
17 fin: bx lr
18
19 LD_debutTAB : .word debutTAB
```

À la place des lignes marquées (*) on peut écrire une des deux solutions suivantes :
— `str r5, [r4, #4]` ; le registre `r4` n'est alors pas modifié.
— ou `mov r6, #4` puis `str r5, [r4, r6]` ; le registre `r4` n'est pas modifié.
D'autres solutions modifiant `r4` sont également possibles.

Exercices : Compléter ce programme de façon à réaliser les dernières affectations. Proposer d'autres solutions. Reprendre le même problème avec un tableau d'octets.

6.1 Codage d'une itération

Si notre tableau était formé de 10000 éléments, la méthode précédente serait bien laborieuse ... On utilise alors un algorithme comportant une itération.

```
lexique local :
  i : un entier compris entre 0 et 4
  val : un entier
algorithme :
  val <-- 11
  i parcourant 0..4
      TAB[i] <- val
      val <- val + 11
```

ce qui peut aussi s'écrire :

```
  val <-- 11
  i <-- 0
  tant que i <> 5   @ ou bien : tant que i <= 4 ou encore i < 5
      TAB[i] <- val
      val <- val + 11
      i <-- i + 1
```

A noter : si i était mal intialisé avant le tant que (par exemple i = 6), on obtiendrait une boucle infinie avec le test $\neq$, et une terminaison sans exécuter le corps du tant que avec les conditions $<$ ou $\leq$.

Nous exprimons le même algorithme en faisant apparaître explicitement l'adresse d'accès au mot de la mémoire : `TAB[i]`.

```
  val <-- 11
  i <-- 0
  tant que i <> 5
     MEM [debutTAB + 4*i] <-- val
     val <- val + 11
     i <-- i + 1
```

Exercices :

1. Coder cet algorithme en langage d'assemblage, en installant les variables `val`, `i` et `debutTAB` respectivement dans les registres : `r7`, `r6` et `r4`.

   Pour évaluer l'expression booléenne `i <> 5`, on calcule `i-5`, ce qui nous permet de tester la valeur de `i <> 5` en utilisant l'indicateur `Z` code de condition arithmétique : si `Z = 1`, `i-5` est égal à 0 et si `Z = 0`, `i-5` est différent de 0.
2. Dérouler l'exécution en donnant le contenu des registres à chaque itération.
3. Modifier le programme si le tableau est un tableau de mots de 8 bits ?
4. Lors de l'exécution du programme précédent on constate que la valeur contenue dans le registre `r4` reste la même durant tout le déroulement de l'exécution ; il s'agit d'un calcul constant de la boucle. On va chercher à l'extraire de façon à ne pas le refaire à chaque fois. Pour cela on introduit une variable `AdElt` qui contient à chaque itération l'adresse de l'élément accédé.

   ```
   val <-- 11; i <-- 0
   AdElt <- debutTAB
   tant que i <= 4
      { invariant : AdElt = debutTAB + 4 * i }
      MEM [AdElt] <-- val
      i <-- i + 1
      val <- val + 11
      AdElt <- AdElt + 4
   ```

On peut alors supprimer la variable d'itération `i` en modifiant le test d'arrêt de l'itération. D'une boucle de parcours de tableau par indice on passe à une boucle de parcours par pointeur (la variable indice i peut être supprimée) :

— multiplication des deux membres de l'inéquation par 4 : $4 * i \leq 4 * 4$
— ajout de debutTAB : $debutTAB + 4 * i \leq debutTAB + 4 * 4$
— remplacement de debutTAB+4*i par AdElt

```
{ i = 0 }
val <-- 11; AdElt <- debutTAB; finTAB <- debutTAB+4*4
tant que AdElt <= finTAB
   { invariant : AdElt = debutTAB + 4 * i }
   MEM [AdElt] <-- val
   val <- val + 11
   AdElt <- AdElt + 4
```

Remarques :

— On peut aussi utiliser les conditions $AdElt \neq finTAB$ ou $AdElt < finTAB$ avec $finTAB < -debutTAB + 4 * 5$, en transformant la condition de départ $i \neq 5$ ou $i < 5$.
— dans le corps du tant que, d'après l'invariant, on pourrait recalculer i à partir de AdElt ($i = (AdElt - debutTAB)/4$).

Après avoir compris chacune de ces transformations, traduire la dernière version de l'algorithme en langage d'assemblage. Traduire aussi la variante 8 bits.

6.2 Calcul de la suite de "Syracuse"

La suite de Syracuse est définie par :

U_0 = *un entier naturel* > 0
U_n = $U_{n-1}/2$ *si* U_{n-1} *est pair*
= $U_{n-1} \times 3 + 1$ *sinon*

Cette suite converge vers 1 avec un cycle.

Calculer les valeurs de la suite pour $U_0 = 15$.

Pour calculer les différentes valeurs de cette suite, on peut écrire l'algorithme suivant (à traduire en langage d'assemblage) :

```
lexique :
  x : un entier naturel
algorithme :
  tant que x <> 1
     si x est pair
     alors x <-- x div 2
     sinon x <-- 3 * x + 1
```

Important : Le TD suivant, premier TD sur les fonctions, est long et comporte de multiples variantes possibles (mais aussi des raccourcis). Prévoir de regarder le sujet du TD jusqu'au bout et peut-être même de le commencer pendant la semaine, avant de venir au prochain TD !

TD séance 7 : Fonctions : paramètres et résultat

7.1 Branchements aller et retour à une routine (sans question)

L'instruction permettant l'appel de fonction ou de procédure est nommée `bl`. Son effet est de sauvegarder l'adresse de l'instruction qui suit l'instruction `bl ...` (on parle de l'adresse de retour) dans le registre `r14` aussi nommé `lr` (Link Register) avant de réaliser le branchement à la fonction ou procédure. Dans la fonction appelée, le retour à l'appelante se fait alors par l'instruction `bx lr`.

Le schéma standard d'un programme `P` appelant une fonction ou procédure `Q` peut s'écrire :

```
P: ...          Q: ...          Pequiv : mov lr,pc  @ équivalent de bl
   bl Q            ...                   b   Q      @ en 2 instructions
   ...             ...                   ...        @ lr repère ici (pc+2 instr)
                   bx lr                             @ car pc avance pendant l'exécution du mov
```

7.2 Un seul niveau d'appel : paramètres et variables locales en registres

7.2.1 Présentation et structure du code

La fonction **coderfonc** et la procédure **coderproc** effectuent une translation circulaire [1] à droite dans l'alphabet : 'a' $\underset{\rightarrow}{3}$ coder('a',3)='d', 'g' $\underset{\rightarrow}{4}$ coder('g',4)='k' et 'z' $\underset{\rightarrow}{2}$ coder('z',2)='b'.

Le premier paramètre c est un code ASCII de lettre minuscule [2], ainsi que le résultat. Le deuxième paramètre n est un entier naturel : $0 \leq n \leq 26$.

Les deux routines ont la même structure de code ci-dessous, seule la manière de transmettre le résultat à l'appelante change. La traduction de la procédure coderproc peut être traitée dans ce TD ou reportée au TD 8.2 après le cours présentant les paramètres de type adresse.

```
codefonc:     @ prologue (sauvegarder)
code:    ...  @
         ...  @ code de corps commun
edoc:         @
              @ traiter résultat (fonc)
              @ épilogue (restaurer)
              @ branchement retour
```

```
codeproc:     @ prologue (sauvegarder)
code:    ...  @
         ...  @ code de corps commun
edoc:         @
              @ traiter résultat (proc)
              @ épilogue (restaurer)
              @ branchement retour
```

1. A la base d'une méthode de cryptage simple (code de César)
2. c ∈ ['a' ...'z'], soit $0x61 \leq c \leq 0x7a$

Le corps des routines est habituellement encadré par une séquence (laissée vide dans cette première partie de td) de sauvegarde en mémoire (prologue) et de restauration (épilogue) des registres modifiés dans le corps de la routine. Le branchement de retour à l'appelante termine l'épilogue.

7.2.2 Traduction du code commun aux 2 routines

`Traduire le code suivant : cdec = c + n;  if (cdec > 'z') {cdec = cdec - 26;}`

Convention de stockage :
- registre `r0` : paramètre c
- registre `r1` : paramètre n
- registre `r12` : variable locale cdec

```
code:    ...      @ cdec = c + n
         ...      @ if (cdec > 'z') {
         ...      @     cdec = cdec - 26
edoc:             @ }
```

A noter : pour copier le contenu de `cdec` dans une variable `res` stockée en mémoire (section `data` ou `bss`[3]), il faut 2 instructions ARM :

```
1            ldr  r4,LD_res   @ r4 = adresse de res (&res en C)
2            strb r12,[r4]     @ Mem[r4] = cdec  (*r4 = cdec en C)
3            ...
4   LD_res: .word res
```

L'instruction `strb` d'écriture en mémoire à l'adresse de `res` appartiendra soit au code de l'appelante de la fonction, soit au code de la procédure.

7.2.3 Transmission du résultat : fonction versus procédure

Il existe deux méthodes pour stocker le résultat `coder('d',5)` à une variable résultat en mémoire :

1. la fonction **coderfonc** stocke la valeur de retour à un endroit convenu (`r2`) par la convention d'appel associée à `coderfonc` et la fonction appelante exécute sa recopie dans `res2`.
2. l'écriture dans `res2` est effectuée directement par la procédure **coderproc**, à laquelle l'appellante passe en troisième paramètre l'adresse de la variable où stocker le résultat.

```
char lu,res1,res2;

// L'appelante écrit
void main () {
  res1 = coderfonc('b',4);
  LireChar(&lu);
  res2 = coderfonc(lu,2);
  EcrireChar(res2);
}
```

```
char lu,res1,res2;

// L'appelante passe l'adresse
void main () {
  coderproc('b',4,&res1);
  LireChar(&lu);
  coderproc(lu,2,&res2));
  EcrireChar(res2);
}
```

3. `bss` : section analogue à `data`, mais dans laquelle tous les octets seront nuls (pas de valeur initiale). Sert principalement à réduire la taille des fichiers exécutables.

Dans le code ci-dessous :

— considérer le type char comme un entier relatif sur 8 bits

— `pres` est de type **char *** : `pres` contient une adresse de variable de type `char` où stocker le résultat

```
// L'appelée n'écrit pas
// Convention d'appel :
// c : r0, n : r1,
// valeur de retour : r2
char coderfonc (char c, unsigned n) {
  char cdec;  // stockée dans r12
  cdec = c + n;
  if (cdec > 'z') {
    cdec = cdec - 26;
  }
  return cdec;
  // valeur retournée dans r2
}
```

```
// L'appelée écrit
// Convention d'appel :
// c : r0, n : r1
// pres : r2
void coderproc (char c, unsigned n, char *pres) {
  char cdec;  // stockée dans r12
  cdec = c + n;
  if (cdec > 'z') {
    cdec = cdec - 26;
  }
  *pres= cdec; // Mem[pres] = cdec
  // Résultat dans Mem[r2]
}
```

Voici un squelette de code d'appel dans l'appelante pour les deux versions de routine. Il est instancié 2 fois (pour x=1 avec c ← 'b' et n ← 4, et pour x=2 avec c ← lu et n ← 2) :

```
parfoncx:  ... @ c_de_coderfonc = ...
           ... @ n_de_coderfonc = ...
               @ pas de 3ème paramètre
brfoncx:   ... @ saut a coderfonc
resfoncx:  ... @ resx = valeur_retour
```

```
parprocx:  ... @ c_de_coderproc = ...
               @ n_de_coderproc = ...
               @ pres_de_coderproc = adr(resx)
brprocx:   ... @ saut a coderproc
resprocx:      @
```

Quelle(s) instruction(s) faut-il mettre dans le bloc branchement retour des routines. **Pourquoi** ne peut-on pas utiliser un branchement ordinaire à une étiquette (b resfonc) ?

Traduire côte à côte (et comparer) en code ARM pour les deux versions de routine :

1. le passage des paramètres explicites dans le code de main (blocs parfonc et parproc)
2. le branchement à coderfonc ou coderproc
3. pour la fonction : le code traiter_résultat de coderfonc et le bloc resf
4. pour la procédure : le code traiter_résultat de coderproc (le bloc resproc est vide)

7.3 Exemple à deux niveaux d'appel : factorielle itérative

7.3.1 Utilisation d'une fonction pour le produit de deux entiers

Une instruction machine et le circuit matériel de multiplication sont devenus courants sur les processeurs RISC modernes, mais leur présence constituait l'exception plutôt que la norme sur les processeurs RISC de première génération. Le code de calcul de $n!$ ci-dessous illustre l'appel d'une fonction `mult` de calcul du produit. Les variables n, res et i sont de type entier naturel.

```
// Code de calcul de n!
res=1;
i = n;
while (i != 1) {
  res = mult(res,i);  // res = res * i
  i = i-1;
 }
```

```
// Prototype de la fonction mult
// et convention d'appel
//  x : registre r0    y : registre r1
// Valeur de retour :
//  registre r0 (remplace x)
// Tous entiers naturels 32 bits
uint32_t mult (uint32_t x, uint32_t y);
```

Vous trouverez une réalisation de `mult` en annexe, mais vous n'avez pas besoin d'en comprendre le fonctionnement pour faire l'exercice.

```
calcul:  ... @ res=1;
         ... @ i = n;
         ... @ while (i != 1) {
             @   res = mult(res,i);
corpsw:  ... @     x_de_mult=res
         ... @     y_de_mult=i
         ... @     saut à mult
suitem:  ... @     res = valeur_retour_mult
         ... @   i = i-1;
condw:   ... @ }
         ... @
luclac:      @
```

Traduire le code de calcul de $n!$ ci-contre, avec 2 contraintes :

1. **Placer** le test de la condition après le corps de while.
2. **Respecter** la convention de stockage suivante :
   (a) res dans le registre r5
   (b) i dans le registre r6
   (c) n dans le registre r7

7.3.2 Variables en mémoire : un niveau d'appel

Le code est complété pour calculer `factx = x!`, `x` et `factx` étant des variables stockées en mémoire. **Traduire** les affectations n=x et factx=res.

```
// entiers naturels 32 bits

uint32_t x;
uint32_t factx;

void main () {
  //  Lire32(&x);

  n=x;       // {

  ...   // Calcul


  factx=res; // }

  // EcrNdecimal32(factx);
}
```

```
        .bss
        .balign 4
x:      .skip 4
factx:  .skip 4

main:   @sauvegardes omises
        @ Lire32(&x) omis
          ...  @ r12  = &x
          ...  @ n = *r12  (n=Mem[r12])
calcul:  ...  @ res = 1
          ...
luclac:
          ... @ r12 = &factx
          ... @ *r12 = res (Mem[r12] = res)

             @restaurations omises
         b   exit   @ à améliorer

px:     .word x
pfactx: .word fcatx
```

7.3.3 Deux niveaux d'appel avec fonction fact

Le bloc calcul est transformé en une fonction fact avec convention d'appel :

1. Paramètre n dans registre r7
2. Valeur de retour dans registre r5

Traduire l'affectation factx=fact(x).

```
uint32_t mult (uint32_t x, uint32_t y){
  ...
  return ;..;
}

uint32_t fact(uint32_t n) {
  ...   // Calcul

  return res;
}

void main () {
   // Lire32(&x);

 factx=fact(x);

   // EcrNdecimal32(factx);
}
```

```
mult:
          ...  @ >1 instructions

          bx lr

fact:          @ sauvegarde omise
calcul:   ...   @ res = 1
          ...
luclac:

               @ restauration omise
          bx lr

main:          @sauvegarde omise
               @ Lire32(&x)

          ...  @ r12 = &x
          ...  @ n_de_fact = *r12
brfact:   ...  @ saut à fact
suitef:   ...  @ r12 = & factx
               @ *r12 = val_retour_fact

               @ EcrNDecimal32(factx)
          bx lr
```

Donner le contenu (quelle étiquette) du registre `lr` à différents instants d'exécution :

1. au début de la fonction `fact`, lors de l'affectation `res=1`
2. lors de l'exécution de la première instruction de la fonction `mult`
3. lors de l'exécution de l'instruction de retour `bx lr` à la fin de `fact` : quelle sera la prochaine instruction exécutée ?

Indiquer à quels endroits le registre lr doit être sauvegardé puis restauré pour que l'exécution du programme n'entre pas dans une boucle infinie.

7.4 Appel de routines d'entrées/sorties

On précise les spécifications suivantes :
— la procédure `LireCar` lit un caractère dans le mot mémoire dont l'adresse est donnée en paramètre, dans le registre r1.
— la procédure `Ecrcar` prend en paramètre d'entrée le caractère à écrire, dans le registre r1.

```
// Type des procédures d'entrées/sorties :
void LireCar (char *destination);   // Mem[destination] = caractère lu au clavier
```

```
void EcrCar (char a_ecrire);        // Affiche a_ecrire à l'écran
```

```
1                    @ variante possible en section data
2         .bss                      .data
3  lu:    .skip 1           lu:    .byte 0
4  res1:  .skip 1           res1:  .byte 0
5  res2:  .skip 1           res2:  .byte 0
6      .text
7  main:
8      @ LireCar(&lu)
9      ...                    @ A COMPLETER
10     bl Lirecar
11     @ le caractere lu est dans la zone data a l adresse lu
12     @ EcrCar(res2) : le caractere a écrire doit etre dans r1
13     ...                    @ A COMPLETER
14     bl EcrCar
15     ...
16 LD_lu:     .word lu
17 LD_res2:   .word res2
```

Compléter la traduction des appels à `LireCar` et `EcrCar` dans la première partie du TD.

Traduire les appels aux routines de lecture et écriture dans le code fonction de `main` qui appelle `fact`.

TD séance 8 : Appels/retours de procédures, actions sur la pile

8.1 Mécanisme de pile

La pile est une zone de la mémoire. Elle est accessible par un registre particulier appelé `pointeur de pile` (noté `sp`, pour `stack pointer`) : le registre `sp` contient une adresse qui repère un mot de la zone mémoire en question.

On veut effectuer les actions suivantes :

— empiler : on range une information (en général le contenu d'un registre) au sommet de la pile.
— dépiler : on "prend" le mot en sommet de pile pour le ranger par exemple dans un registre.

Le tableau ci-dessous décrit les différentes façons de mettre en oeuvre une pile en fonction des conventions possibles pour le sens de progression (vers les adresses croissantes ou décroissantes) et pour la contenu de la case mémoire pointée (vide ou pleine).

sens pointage	croissant 1^{er} vide	croissant dernier plein	décroissant 1^{er} vide	**décroissant dernier plein**	**instruction Full Desc.**
empiler reg	M[sp]←reg sp←sp+1	sp←sp+1 M[sp]←reg	M[sp]←reg sp←sp-1	**sp ←sp-1** **M[sp]←reg**	**push {reg}**
dépiler reg	sp←sp-1 reg←M[sp]	reg←M[sp] sp←sp-1	sp←sp+1 reg←M[sp]	**reg←M[sp]** **sp←sp+1**	**pop {reg}**

Dans le TD et dans tout le semestre, on travaille avec un type de mise en oeuvre. On choisit celle qui est utilisée dans le compilateur `arm-eabi-gcc` c'est-à-dire "décroissant, dernier plein" (Full Desc.) (Cf. figure 8.1).

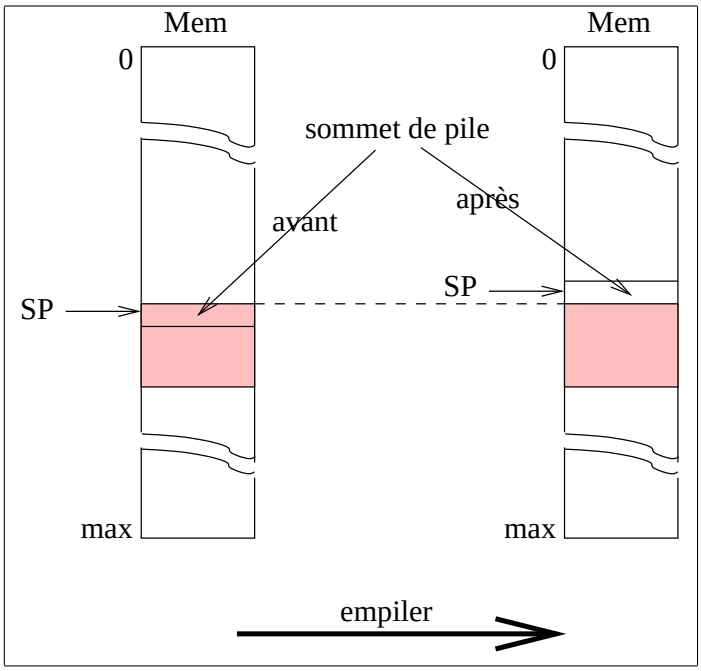


FIGURE 8.1 – Mise en oeuvre de la pile. La pile progresse vers les adresses **décroissantes**, le pointeur de pile repère la **dernière** information empilée

Exercices : utilisation de la pile

Supposons que la pile soit comprise entre les adresses `3000` comprise et `30F0` exclue. Le pointeur de pile est initialisé avec l'adresse `30F0`. Dans cet exercice on empile des informations de taille 1 octet.

Questions :

- Quelle est la valeur de `sp` quand la pile est pleine ?
- De combien de mots de 32 bits dispose-t-on dans la pile ?
- De combien d'octets dispose-t-on dans la pile ?
- Ecrire en ARM les deux instructions élémentaires permettant d'empiler le contenu de l'octet de poids faible du registre `r0`. Dans la suite du TD on ecrira `empiler r0` ou `push {r0}`.
- Ecrire en ARM les deux instructions élémentaires permettant de dépiler le sommet de pile dans le registre `r0`. Dans la suite du TD on écrira `depiler r0` ou `pop {r0}`..
- Dessiner l'état de la mémoire après chacune des étapes du programme suivant : `mov r0, #7; push {r0}; mov r0, #2; push {r0}; mov r0, #5; push {r0}; mov r0, #47; pop {r0}; pop {r0}; mov r0, #9; push {r0}`
- Reprendre l'exercice si on travaille avec des informations codées sur 4 octets. Comment modifier le code de `empiler` et `depiler` ?

8.2 Appel et retours de procédures

On travaille avec le programme ci-dessous ; les procédures "A", "B" et "C" sont rangés aux adresses 10, 60 et 80.

Remarque : il s'agit du programme donné en cours dans lequel on a remplacé les Ai, Bi et Ci par des vraies instructions.

```
10  A1= mov r0, #0          60  B1= push {r0}          80  C1= mov r0, #47
14  A2= push {r0}           68  B2= add r0, r0, #1     84  bl 60 (B)
1c  bl 60 (B)               6c  B3= pop {r0}           88  C2= push {r0}
20  A3= mov r5, #28         70  bx lr                  90  bl si condX 80 (C)
24  bl 80 (C)                                          94  C3= mov r2, r5
28  A4= pop {r0}                                       98  C4= pop {r0}
                                                       a0  bx lr
```

Questions : Le programme C est incorrect. Expliquer pourquoi et le corriger en conséquence.

Donner une trace de l'exécution du nouveau programme en indiquant après chaque instruction le contenu des registres et de la pile.

pc	inst	sp	r0	r2	r5	lr	m[30f0]	m[30ec]	m[30e8]	m[30e4]	m[30e0]
?	?	30f0	?	?	?	?	?	?	?	?	?
10	`mov r0, # 0`	30f0	0	?	?	?	?	?	?	?	?
14	`push {r0}`	30ec	0	?	?	?	?	0	?	?	?

TD séance 9 : Correction du partiel

Correction du contrôle donné pendant la semaine de partiel.

TD séance 10 : Paramètres dans la pile, paramètres passés par adresse

10.1 Gestion des paramètres et des variables dans la pile

Reprendre les fonctions `codefonc`, `fact` et `mult` du TD7.

Exercices :

- transformer la traduction de ces fonctions en langage ARM pour gérer le passage des paramètres et les variables locales dans la pile. Écrire les appels qui correspondent.
- reprendre les exemples traités précédemment dans ce TD et effectuer les sauvegardes nécessaires de temporaires dans la pile.

10.2 Paramètre passé par adresse

10.2.1 Un premier exemple

Traduire (si cela n'a pas été fait au TD7) la fonction `coderproc` et son appel en passant les paramètres par les registres.

Reprendre cette traduction en supposant que les 3 paramètres explicites sont à présent passés dans la pile (paramètre de gauche c en sommet de pile). On pourra si nécessaire passer par une première version dans laquelle les paramètres sont dans `data` ou `bss`.

10.2.2 Une version récursive de procédure calculant factorielle

On considère la version suivante du calcul de la factorielle d'un entier :

```
procedure fact2 (donnée n: entier, adresse fn: entier) {
int fnmoins1;
    si (n == 1)
    alors mem[fn] = 1;
    sinon
        fact2 (n-1, adresse de fnmoins1);
        mem[fn] = n * fnmoins1;
}

n, fn : entier
   Lire (n)
   fact2 (n, adresse de fn)
   Ecrire (fn)
```

Exercice : donner une traduction en langage d'assemblage ARM de cette procédure.

TD séances 11 et 12 : Organisation d'un processeur : une machine à pile

11.1 Description du processeur

Cette machine dispose de registres visibles par le programmeur :

— `acc` : accumulateur pour stocker des valeurs,
— `pc` : compteur de programme,
— `sp` : pointeur de pile.

`pc` est initialisé à 0 et repère la prochaine instruction à exécuter.

La pile suit la convention *progression décroissante, dernier plein.* `sp` est initialisé à `0xFE` (la pile commence donc à `0xFD`).

Il y a aussi des registres non visibles par le programmeur, c'est-à-dire, qui ne peuvent pas être utilisés dans un programme en langage machine :

— `Rinst` : registre instruction qui contient l'instruction en cours d'exécution,
— `ma` et `mb` : registres qui servent aux accès mémoire,
— `mk1` et `mk2` : registres servant à des calculs internes au processeur.

La mémoire est composée de mots de taille un octet. Les adresses sont aussi sur un octet.

Il existe des entrées sorties rudimentaires : la lecture du mot mémoire d'adresse `0xFE` correspond à une lecture au clavier et l'écriture dans le mot mémoire d'adresse `0xFF` correspond à un affichage sur l'écran.

Le répertoire d'instructions est donnée dans la figure 11.1.

Le compteur programme indique la prochaine instruction à exécuter. Ainsi, lors de l'exécution de l'instruction `jumpifAccnul`, la valeur du déplacement est calculée par rapport à l'adresse de l'instruction suivante (c'est-à-dire, l'instruction qui sera exécutée ensuite si la condition de saut n'est pas vérifiée).

Le code d'une instruction est choisi de telle façon que le décodage soit facilitée par le test d'un bit : `load` : 1_{10}, `input` : 2_{10}, `output` : 4_{10}, `push-acc` : 8_{10} `pop-acc` : 16_{10}, `add` : 32_{10}, `dup` : 64_{10} et `jumpifAccnul` : 128_{10}.

La figure 11.2 décrit l'organisation générale du processeur et de la mémoire.

instruction	signification	code opération (valeurs en décimal)	taille codage
`load# vi`	`acc <-- vi`	1	2 mots
`input`	`acc <-- Mem[0xFE]`	2	1 mot
`output`	`Mem[0xFF] <-- acc`	4	1 mot
`push-acc`	empiler acc	8	1 mot
`pop-acc`	dépiler vers acc	16	1 mot
`add`	ajouter le sommet et le sous-sommet de la pile, ils sont dépilés, empiler la somme l'accumulateur n'est pas modifié	32	1 mot
`dup`	dupliquer le sommet de pile	64	1 mot
`jumpifAccnul depl`	saut conditionnel à pc+depl la condition est "accumulateur nul"	128	2 mot

FIGURE 11.1 – Les intructions de la machine à pile

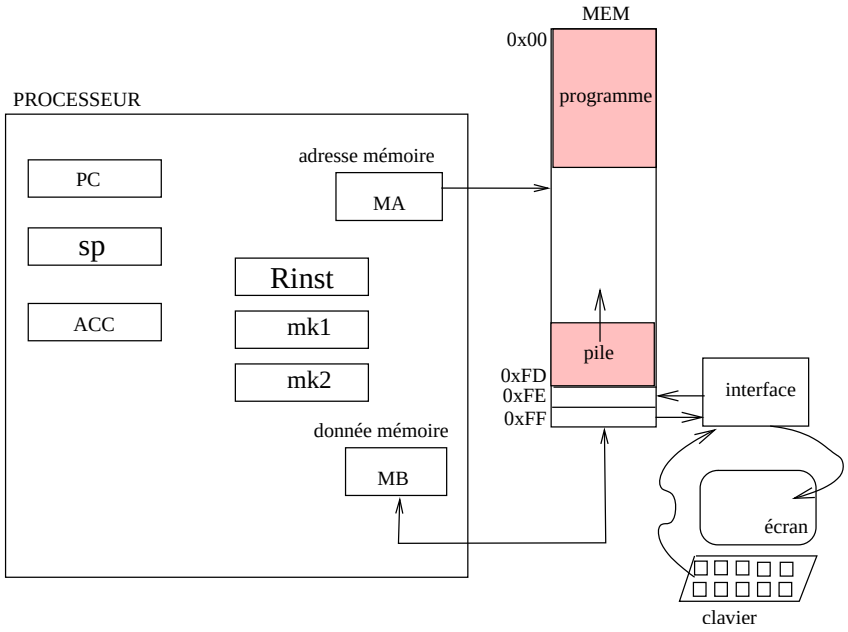


FIGURE 11.2 – La machine à pile et sa mémoire

11.1.1 Représentation en mémoire d'un programme

Donner la représentation en mémoire (en binaire et en hexadécimal) du programme en langage d'assemblage suivant :

```
load# 0x05
push-acc
push-acc
add
pop-acc
```

11.1.2 Evolution des valeurs des registres lors d'une exécution

Décrire l'évolution des registres `acc`, `pc`, `sp` et de la pile lors de l'exécution du programme précédent. On supposera la pile vide et le programme à l'adresse 0.

11.1.3 Plus d'addition et de pile ?

Supposons que le programme précédent soit plus long et qu'il empile beaucoup. Par exemple qu'il empile `N` fois avant d'effectuer `N-1` additions.

Combien doit-il faire d'opérations pour dépiler ce qu'il a empilé ?

Quel est le résultat obtenu ?

Est-ce qu'il y a une taille maximale pour le programme ? (quel est le `N` maximum ?)

Est-ce que le programme et la pile ne risquent pas d'entrer en conflit à l'exécution ? Comment peut-on éviter ce problème ? (quel est le `N` maximum ?)

Est-ce que les calculs ne risquent pas de déborder ? Comment peut-on éviter ce problème ? (quel est le `N` maximum ? préciser si cela dépend de la représentation des entiers, naturels vs relatifs ?)

L'organisation du processeur et les instructions disponibles sont-elles suffisantes ? (sinon, décrire brièvement les extensions possibles)

11.2 Interprétation des instructions sous forme d'un algorithme

Afin de comprendre comment évoluent les différents registres du processeur au cours de l'exécution d'un programme, on peut donner une interprétation du fonctionnement du processeur sous forme d'un algorithme.

11.2.1 Algorithme

Donner l'algorithme d'interprétation des instructions.

Soigner la description des accès mémoires en utilisant `ma, mb` et en s'appuyant sur le schéma 11.2.

Pour les dernières instructions `add, dup, jumpifAccnul` vous pouvez utiliser les registres `mk1, mk2` si nécessaire.

11.2.2 Fonctionnement de l'algorithme

Compléter la description obtenue en 11.1.1 avec les différentes valeurs contenues dans les registres non visibles du processeur (`ma, mb, mk1, mk2, rinst`) au cours de l'interpétation du programme donné en 11.1.1.

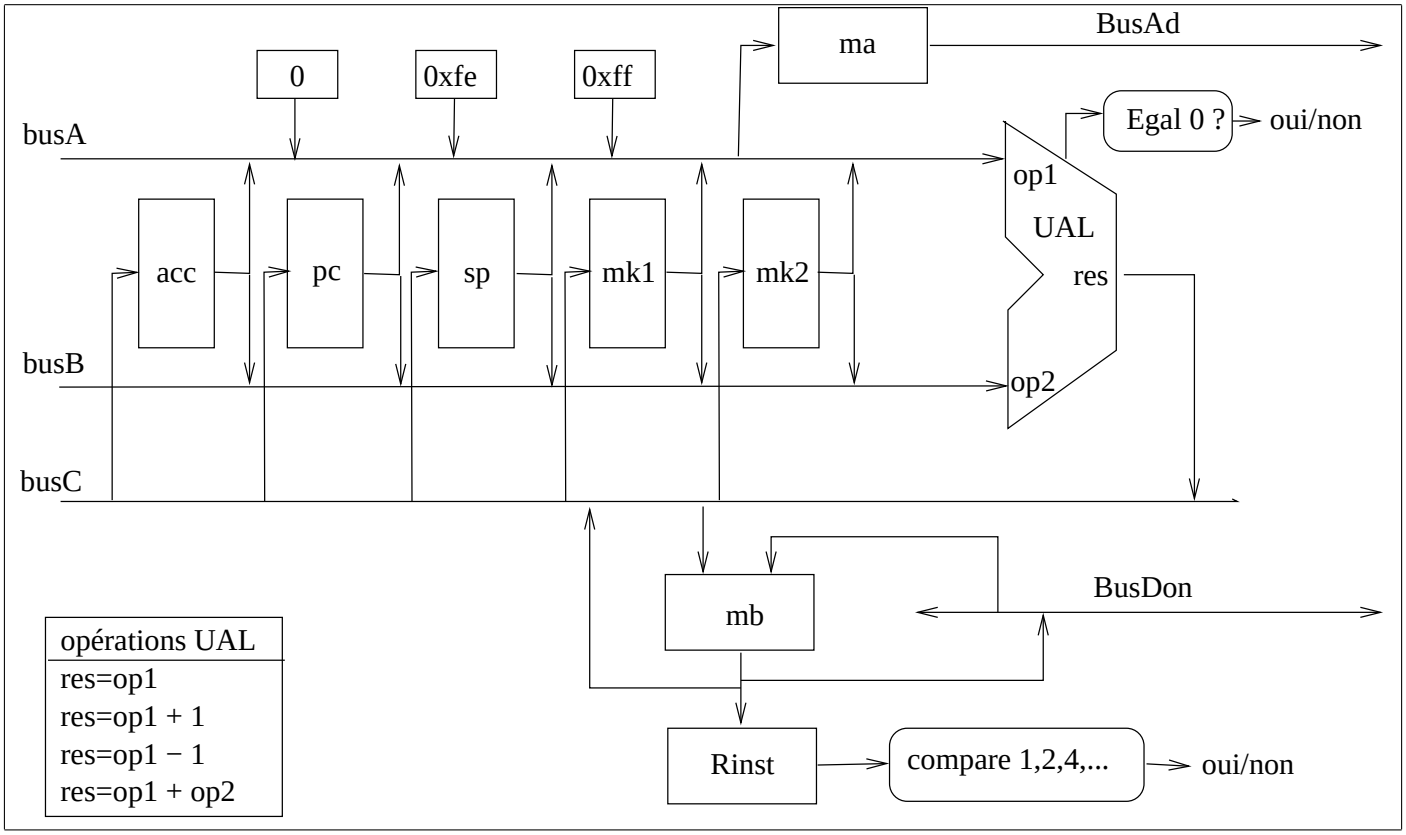


FIGURE 11.3 – Organisation de la machine à pile

11.3 Interprétation des instructions sous forme d'un automate

On précise les opérations de base que le processeur peut effectuer : les **micro-action**. Une micro-action dure un cycle d'horloge.

L'ensemble des micro-actions possibles dépend de l'organisation physique du processeur (cf. figure 11.3).

Pour notre exemple, les actions élémentaires sont les suivantes :

1. micro-actions internes au processeur :
   - `reg_i` ← `0`
   - `reg_i` ← `reg_j`
   - `reg_i` ← `reg_j + 1`
   - `reg_i` ← `reg_j - 1` note : -1≡+ff
   - `reg_i` ← `reg_j + reg_k`
   - `mb` ← `reg_i` (via l'UAL)
   - `reg_i` ← `mb`
   - `rinst` ← `mb`
   - `ma` ← `0`
   - `ma` ← `reg_i`
   - `ma` ← `0xff`
   - `ma` ← `0xfe`
2. micro-actions permettant l'accès à la mémoire :
   - lecture mémoire : `mb` ← `Mem [ma]`
   - écriture mémoire : `Mem [ma]` ← `mb`

avec `reg_i`, `reg_j`, `reg_k` ∈ { `sp`, `pc`, `mk1`, `mk2`, `acc` }.

On dispose des tests de la valeur contenue dans le registre `Rinst` : `Rinst` = code de `load#`, code de `add`, ...Par ailleurs, le "calcul" `acc = acc + 0` permet de tester si `acc` est nul où non.

11.3.1 Séquence de micro-actions pour une instruction

Vérfier que le programme d'interprétation proposé à la section précédente est compatible avec les actions possibles, sinon le ré-écrire avec une suite d'actions élémentaires autorisées.

11.3.2 Automate d'interprétation, graphe de contrôle

Proposer un automate d'interprétation des instructions pour la machine à pile. Il s'agit de rassembler l'ensemble des séquences de micro-actions en mettant en évidence des sous-séquences communes.

Donner la trace d'exécution du programme donné en 11.1.1 avec les états de cet automate.

Fin prévue pour le TD 11, indications pour la semaine prochaine : après ce TD d'introduction aux processeurs, une séance d'approfondissement des notions associées à cette organisation est prévue avec le TD 12. Ultime TD de l'année, cette séance peut aussi réserver un temps pour des questions/réponses et un exercice de révision (Note aux étudiants : prévoir les questions et exercices à l'avance).

12.1 Un autre exemple

Voici un second programme pour la machine à pile :

```
      load# -1
      push
      dup
      load# 4
      push
TITI: add
      pop
      dup
      push
      jumpifAccnul TOTO
      load# 0
      jumpifAccnul TITI
TOTO: load# 5a
      output
      load# 0
FIN: jumpifAccnul FIN
```

12.1.1 Questions

Donner le code en hexadécimal ainsi que son implantation en mémoire à partir de l'adresse 0. (La question intéressante est la valeur du déplacement pour les instructions de branchements)

Donner l'évolution des valeurs dans les registres et dans la pile lors de l'exécution de ce programme.

Donner la trace lors de l'exécution de ce programme avec les états de l'automate.

12.2 Optimisation du graphe de contrôle

Nous pouvons envisager plusieurs types d'optimisations : diminuer le temps de calcul des instructions ou diminuer le nombre d'états du graphe de contrôle.

12.2.1 Temps de calcul d'une instruction, d'un programme.

Une micro-action dure le temps d'une période d'horloge. Choisir une fréquence et calculer le temps de calcul de chaque instruction du processeur étudié pour l'exemple donnée dans le paragraphe 11.3.2.

Pouvez-vous améliorer ce temps de calcul ? Quelles sont les parties incompressibles ?

Pour les deux programmes vus dans ce TD, donner le temps de calcul nécessaire pour atteindre la dernière instruction (hors exécution de cette dernière instruction pour le dernier programme).

12.2.2 Nombre d'états du graphe de contrôle

Est-il possible de diminuer le nombre d'états du graphe proposé :

— avec la même partie opérative ?

— en modifiant la partie opérative : ajoût de registres, de bus, etc. ?

Faire un dessin.

12.3 Extension de la machine à pile

La prochaine génération de machine à pile devra avoir 4 nouvelles instructions :

— `resetStack` : vide la pile (simple réinitialisation du pointeur de pile)

— `eraseStack` : remplit la pile de 0 et réinitialise le pointeur de pile

— `jmpifEmptystack depl` : exécute un saut conditionnel à pc+depl, la condition est "pile vide"

— `jmpifFullstack depl` : exécute un saut conditionnel à pc+depl, la condition est "pile plein"

Discuter et choisir une zone pour la pile.

Discuter et choisir des codes opérations pour ces nouvelles instructions.

Donner l'algorithme (ou la partie d'algorithme) pour prendre en compte ces nouvelles instructions.

Compléter l'automate avec l'interprétation de ces nouvelles instructions. (Ajouter éventuellement une constante pour le haut de la pile, dans le schéma de la machine, et les actions associées)

Proposer un programme utilisant l'une ou l'autre de ces instructions, le représenter en mémoire et donner la trace de son exécution.

Deuxième partie

Travaux Pratiques

TP séance 1 : Représentation des informations (ex. : images, programmes, entiers)

1.1 Comment est représentée une image ?

On va utiliser le format `bitmap`. Ce format permet de décrire une image extrêmement simple en noir et blanc ; on peut par exemple l'utiliser pour décrire une icône.

Une image est un ensemble de points répartis dans un rectangle. L'image est définie par un texte de programme en langage `C` comprenant la taille du rectangle et la valeur de chacun des points : noir ou blanc. Un point est décrit par 1 bit (vrai = 1 = noir).

On donne ci-dessous le contenu du fichier `image.bm` codant une image de dimensions 16×16 dans laquelle tous les points sont blancs.

```
#define image_width 16
#define image_height 16
static unsigned char image_bits[] = {
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
```

1.1.1 Modifier une image "à la main"

Le naturel 0 codé sur 8 bits s'écrit `0x00` en hexadécimal et `0000 0000` en binaire ; il représente 8 points blancs contigus alignés horizontalement.

— Au moyen d'un éditeur de texte (`nedit` par exemple), modifiez le fichier `image.bm` de façon à ce qu'il contienne la description d'une image de dimensions 16×16 dans laquelle la troisième ligne est noire. Vous afficherez votre image avec la commande : `bitmap image.bm`.
Pour sortir, sélectionner `Quit` dans le menu `File`.

— Quelles modifications avez-vous apportées au fichier `image.bm` ?

1.1.2 Codage d'une image

Effectuez la manipulation suivante :

— Créez au moyen de votre éditeur de texte un fichier `monimage.bm` contenant une image de dimensions 16×16 au format `bitmap`.
Tous les points de cette image doivent être blancs, exceptés ceux de la première ligne qui doivent représenter le motif ayant l'aspect suivant :
■■□□□■■■□□□□□■□■□

— Utilisez le programme `bitmap` pour afficher l'image contenue dans le fichier `monimage.bm`. Vérifiez que l'image affichée correspond bien au résultat attendu.
— Ecrivez en binaire les valeurs que vous avez codées dans le fichier. Expliquez le codage que vous avez utilisé pour obtenir l'image demandée.
— **Indications** (au cas où) : si vos premiers essais ne sont pas concluants, essayez d'obtenir un seul pixel du motif, par exemple le premier pixel, puis le second pixel, puis les deux ensembles etc. ou essayer de mettre `0x01` comme première valeur du fichier, puis `0x02`, ou `0x03`.

1.2 Comment est représenté un programme ?

Considérons un programme écrit en langage C : `prog.c`.

```
/* prog.c */
int NN = 0xffff;
char CC[8] = "charlot";

int main() {
    NN = 333;
    NN= NN + 5;
}
```

Vous allez le compiler, c'est-à-dire le traduire dans un langage interprétable par une machine avec la commande : `arm-eabi-gcc -c prog.c`. Vous obtenez le fichier `prog.o`.

C'est du "binaire"... Nous allons le regarder avec différents outils.

1.2.1 Une première expérience

Essayez successivement les quatre expériences suivantes :
— `nedit prog.o`
— `more prog.o`
— `less prog.o`
— `cat prog.o`
Qu'avez-vous observé ? Qu'en concluez-vous ?

1.2.2 Affichage en hexadécimal

Tapez : `hexdump -C prog.o`. Vous observez des informations affichées en hexadécimal et les caractères correspondants sur la droite. Plus précisément, sur la gauche vous avez des adresses, c'est-à-dire des numéros qui comptent les octets (paquets de 8 bits), et au centre l'information qui est affichée en hexadécimal.

Combien d'octets sont codés sur une ligne affichée ? Combien de mots de 32 bits cela représente-t-il ?

Les caractères de la chaine de caractères `"charlot"` sont codés en `ASCII` : chaque caractère est représenté sur un octet (8 bits, 2 chiffres hexadécimaux). Pour avoir le code ascii d'un caractère, tapez `man ascii` ou consultez votre documentation technique.

Repérez la chaîne `"charlot"` dans l'affichage à droite et trouvez l'information correspondante au centre. A quelles adresses est rangée cette chaîne ?

La valeur `ffff` de l'entier `NN` n'est pas bien loin de `"charlot"`, la trouver.

La chaine `NN` est-elle dans ce fichier ? Au même endroit que les deux valeurs précédentes ?

Grâce à la commande `xterm &` on peut ouvrir plusieurs fenêtres et comparer ce qu'affiche `hexdump` et ce qu'affiche `nedit` pour un même fichier. Comparer les caractères dont le code est compris entre `0x20` et `0x7f` et ceux qui ne sont pas dans cet intervalle.

1.2.3 Affichage plus “lisible”

Le programme a été traduit dans le langage machine `ARM`.

La commande `arm-eabi-objdump -S prog.o` donne la séquence d'instructions `ARM` qui correspond à nos instructions C. On peut lire l'adresse de l'instruction, puis son code en hexadécimal et enfin les mnémoniques correspondants en langage d'assemblage.

On va repérer le code qui correspond à l'instruction `NN = 333`. C'est fait en deux fois :

```
10:   e3a03f53        mov     r3, #332
14:   e2833001        add     r3, r3, #1
```

La question suivante étant un peu plus complexe, vous en chercherez la réponse chez vous ou en fin de séance si vous avez terminé en avance. Pour cela il faut lire en détail le paragraphe 2.3 de la documentation technique et la remarque du paragraphe **??**. Pourquoi le processeur ARM ne permet-il pas d'écrire `mov r3, #333` ?

`arm-eabi-gcc` traduit un programme écrit dans le langage C en un programme écrit en langage machine du processeur `ARM`.

Avec d'autres options le compilateur `gcc` traduit dans le langage machine d'autres processeurs. Par exemple, par défaut, `gcc` effectue la traduction pour le processeur contenu dans la machine sur laquelle vous travaillez ; dans votre cas c'est le langage machine du processeur `INTEL`... Effectuez l'expérience suivante :

```
gcc -c prog.c
hexdump -C prog.o
objdump -S prog.o
```

Regardez la ligne ci-dessous :

```
  11:   c7 05 00 00 00 00 4d    movl   $0x14d,0x0
```

Quel nombre représente `0x14d` ?

Combien d'instructions a-t-il fallu pour traduire l'affectation `NN=333` pour chacun des deux processeurs ?

1.3 Langage d'assemblage/langage machine

Le but de cette partie est de traduire des instructions écrites en langage d'assemblage `ARM` en langage machine.

Une instruction en langage d'assemblage est traduite en une instruction en langage machine par une suite de bits. On exprime cette suite de bits en hexadécimal car c'est plus facile à lire.

A l'aide de la documentation technique ARM, traduire les instructions ci-dessous. Pour chacune, mettre en évidence le codage de la valeur immédiate, la valeur du bit S, la valeur du bit I, le codage du numéro des registres.

```
ADD r10, r2, #10
ADD r10, r2, #17
ADDS r10, r2, #10
ADD r10, r2, r3
```

Pour vérifier vos résultats effectuez l'expérience suivante :

— On fabrique 2 programmes en langage d'assemblage presque identiques. Par exemple, le programme `prog1.s` se différencie du programme `prog1.var1.s` par une instruction. Laquelle ?

— Produire les 2 programmes en langage machine : `arm-eabi-gcc -c prog1.s` et `arm-eabi-gcc -c prog1.var1.s`).
— Observer leur contenu : `arm-eabi-objdump -S prog1.o` et `arm-eabi-objdump -S prog1.var1.o`.
— Comment interprétez-vous les résultats de cette expérience ?

Reproduire la même expérience pour les instructions : `ADDS r10, r2, #10` (programme `prog1.var2.s`) et `ADD r10, r2, r3` (programme `prog1.var3.s`).

Ci-dessous le contenu des fichiers servant à cette expérience.

```
@-------------------- prog1.s ----------------------
   .text
   .global main
main:
      ADD r10, r2, r0
      ADD r10, r2, #10
fin:  BX LR

@-------------------- prog1.var1.s ----------------------
   .text
   .global main
main:
      ADD r10, r2, r0
      ADD r10, r2, #17
fin:  BX LR

@-------------------- prog1.var2.s ----------------------
   .text
   .global main
main:
      ADD r10, r2, r0
      ADDS r10, r2, #10
fin:  BX LR

@-------------------- prog1.var3.s ----------------------
   .text
   .global main
main:
      ADD r10, r2, r3
      ADD r10, r2, #10
fin:  BX LR
```

1.4 Codage des couleurs

Nous nous intéressons ici au codage des couleurs. On utilise le codage dit `RGB`. Il s'agit pour coder une couleur de donner une proportion des trois couleurs rouge (`Red`), vert (`Green`) et bleu (`Blue`) pour les images fixes ou animées.

Une couleur est codée par un nombre exprimé en hexadécimal sur 3×2 chiffres dans l'ordre : Rouge, Vert, Bleu ; `ff` représentant la proportion maximale. Par exemple, `00ff00` code la couleur verte, `000012` code une nuance de bleu.

Pour plus d'informations vous pouvez regarder :
`http://en.wikipedia.org/wiki/RGB_color_model`

La figure 1.1 donne la description d'une image dans le format `xpm`. Cette image comporte deux segments. L'image est décrite dans le langage `C` à l'aide d'un tableau de caractères. On y trouve la taille de l'image (32×32), le nombre de caractères utilisés pour la représenter (3), la couleur (au codage `RGB`) associée à chacun des caractères, et enfin la matrice de points, un caractère étant associé à un point. `None` désigne une couleur prédéfinie dans le système.

On remarque que la proportion d'une couleur est codée sur deux chiffres hexadécimaux, `ff` représentant le maximum.

1.4.1 Fabriquer une image à la main

1. Récupérez le fichier `lignes.xpm` et affichez l'image avec `xli`, `xpmview`, `eog` ou `mirage`.
2. Quelles sont les couleurs des deux segments ? Donnez le code de chacune de ces deux couleurs.
3. Editez ce fichier avec un éditeur classique et modifiez la couleur d'un segment en modifiant les proportions de la couleur de ses points puis affichez à nouveau l'image. Faites éventuellement plusieurs essais et observez qu'une légère modification de la proportion d'une couleur de base n'est pas visible à l'oeil. A partir de quelle proportion distingue-t-on une différence ?
4. Remplacez un des caractères codant un point d'une couleur donnée par un autre. Par exemple, remplacer `a` par `s`. Pensez à effectuer ce remplacement dans la définition de la couleur du point et dans la matrice de points. Quel est l'effet d'une telle modification ?

Important : Le TP2 commence par quelques calculs "à la main", i.e. hors machine ; vous pouvez les commencez pendant la semaine, chez vous, avant le TP.

```
1
2  /* XPM */
3  static char *lignes[]={      /* le fichier doit s'appeler lignes.xpm */
4  "32 32 3 1",
5  ". c None",
6  "# c #f0f000",
7  "a c #ff0000",
8  "................................",
9  "................................",
10 "................................",
11 "................................",
12 "................................",
13 "................................",
14 "...############.................",
15 "................................",
16 "................................",
17 "................................",
18 "................a...............",
19 "................a...............",
20 "................a...............",
21 "................a...............",
22 "................a...............",
23 "................a...............",
24 "................a...............",
25 "................a...............",
26 "................a...............",
27 "................a...............",
28 "................................",
29 "................................",
30 "................................",
31 "................................",
32 "................................",
33 "................................",
34 "................................",
35 "................................",
36 "................................",
37 "................................",
38 "................................",
39 "................................"};
```

FIGURE 1.1 – Le fichier : `lignes.xpm`

TP séance 2 : Codage et calculs en base 2

Rappel : vous êtes censés avoir lu entièrement le sujet du td sur la base 2, qui rappelle entre autres la définition et la signification des 4 indicateurs (Z,N,C V) et la définition des opérations booléennes bit à bit et de décalage.

Garder également à l'esprit que la représentation d'un entier $x = \sum_{i=0}^{n-1} x_i.B^i$ codé sur n chiffres en base B est une suite de n chiffres x_i, x_0 étant le chiffre de droite (des unités) et x_{n-1} le chiffre de gauche (de poids fort).

S'exercer (notament pour les contrôles) à effectuer les conversions de base sans calculette.

2.1 Calculs à effectuer à la main (rappel : 0x → base 16)

2.1.1 Conversions

Convertir 277_{10} en bases 2 et 16, 0x4E en bases 2 et 10 et 11011001_2 en bases 10 et 16.

2.1.2 Addition

Poser en décimal sur 3 chiffres, en hexadécimal sur 2 chiffres et en binaire sur 8 bits l'addition 57_{10} + `0xA3`. Sur l'opération binaire :

1. faire apparaître les retenues c_i, la dernière retenue $C = c_8$
2. les deux dernières retenues sont-elles égales ?
3. que valent les indicateurs N, Z et V ?
4. que valent les opérandes et le résultat apparent sur 8 bits et l'opération est-elle correcte ?
   — s'il s'agit d'entiers naturels (addition dans $\mathbb{N}$)
   — s'il s'agit d'entiers relatifs (addition dans $\mathbb{Z}$)

Quelles réponses changent si cette même addition est réalisée sur 9 bits ?

2.1.3 Soustraction

Poser en décimal sur 2 chiffres, en hexadécimal sur 1 chiffre et en binaire sur 4 bits les soustractions suivantes : 0xD - 0x8, 0x8 - 0xD, 0x7 - 0x7. Sur l'opération binaire sur 4, puis sur 5 bits en observant les changements :

1. faire apparaître les emprunts e_i et le dernier emprunt E=e_4 (puis e_5)
2. que valent les indicateurs N, Z et V ?
3. interpréter l'opération en supposant qu'il s'agisse d'entier naturels, puis d'entiers relatifs

4. la soustraction est-elle possible dans l'ensemble $\mathbb{N}$?
5. la soustraction donne-t-elle un résultat correct dans l'ensemble $\mathbb{Z}$?

2.1.4 Soustraction par addition du complément à 2

Poser à la main en binaire sur n=4 puis sur n=5 bits l'addition 0xD + 0x7 en utilisant une retenue initiale non nulle (i.e. avec $c_0 = 1$, attention, ne pas oublier ce détail qui permet de faire le "+1"). **Comparer** la ligne des retenues (dont C=c_n retenue sortante) avec celles des emprunts (dont E=e_n) emprunt sortant) de la soustraction précédente (0xD-0x8).
Pour vérifier avec la calculette qui ne permet pas de spécifier une retenue initiale non nulle dans l'addition, utiliser subc2 4 0xD 0x8.
Sur 5 bits, comparez aussi l'addition 0x1D + 0x17 + retenue initiale avec la soustraction -5 - -8.

Recommencer avec 0x8 + 0x2 (+ une retenue initiale non nulle $c_0 = 1$) et la soustraction précédente 0x8 - 0xD.

Quelle doit être la valeur de C pour que la soustraction dans $\mathbb{N}$ soit possible ?

2.2 Présentation de la calculette binaire

Voici le mode d'emploi de la calculette binaire qui vous permettra de vérifier les opérations manuelles précédentes.

Syntaxe d'utilisation

La syntaxe d'utilisation de la calculette est la suivante :

opération nombre_de_bits opérande_gauche opérande_droit

Les opérations disponibles sont les suivantes :

1. **add** et **addsc** : addition
2. **sub** et **subsc** : soustraction
3. **subc2** et **subc2sc** : soustraction par addition du complément à deux

Le nombre de bits spécifie la taille de la machine utilisée. Le suffixe **sc** (show carries) pour les additions et soustractions spécifie de détailler la génération des retenues et des emprunts.

Format des opérandes et du nombre de bits

Les entiers utilisés par la calculette binaire peuvent être spécifiés sous trois formats : décimal (par défaut), hexadécimal (avec le préfixe **0x**, comme en langage C) et binaire (avec le préfixe **0b**, inconnu du langage C).

Il est de plus possible de spécifier le complément à 1 (préfixe **/**) ou le complément à deux (ou opposé : préfixe **-**) d'un entier [1].

A titre d'exemple, voici plusieurs manières de spécifier l'entier 111100001001_2 sur 12 bits :

1. en binaire : **0b111100001001 /0b000011110110** ou **−0b000011110111**
2. en hexadécimal : **0xf09 /0x0f6** ou **−0x0f7**
3. et en décimal : **3849 /246** ou **−247**.

1. Ceci s'applique aux opérandes, mais pas au nombre de bits

2.3 Compréhension des indicateurs N,Z,V

Réaliser les additions sur 4 bits détaillées dans le tableau ci-dessous (a et b sont les opérandes codés en binaire sur n=4 bits, à interpréter soit comme des entiers naturels, soit comme des entiers relatifs).
Pour chaque d'entre elles, compléter les informations suivantes :

1. valeurs décimales de a et b en tant qu'entiers naturels et relatifs
2. écriture binaire du résultat apparent Σ, et valeurs décimales en tant qu'entier naturel et relatif
3. bit de poids fort du résultat apparent $N = \Sigma_{n-1}$, bit de poids poids fort des opérandes a_{n-1} et b_{n-1}.
4. indicateur V, dernière $C = c_n$ (sortante) et avant-dernière c_{n-1} retenue
5. le résultat apparent Σ est-il correct pour l'addition dans ℕ, pour l'addition dans ℤ ?

a			b			Σ			N=	a_{n-1}	b_{n-1}	V	C=c_n	c_{n-1}	Erreur ?	
bin	rel	nat	bin	rel	nat	bin	rel	nat	Σ_{n-1}						rel	nat
0100			0011													
0100			0101													
1001			0110													
1111			1110													
0100			1000													
1000			1000													

Compléter les phrases suivantes : après une addition

1. Le résultat est correct pour des entiers naturels si et seulement si l'indicateur $= 0$
2. Le résultat est correct pour des entiers relatifs si et seulement si l'indicateur $= 0$
3. L'indicateur N n'est pertinent que pour une addition d'entiers .....(nature d'entiers)
4. L'indicateur V est vrai si et seulement si .....(propriété portant sur (C,c_{n-1}))
5. L'indicateur V est vrai si $a_{n-1} =$ *et* $b_{n-1} =$ *et* $\Sigma_{n-1} =$ ou $a_{n-1} =$ *et* $b_{n-1} =$ *et* $\Sigma_{n-1} =$.

Effectuer sur 5 bits les additions de relatifs +4 + +5 et -8 +-8. Observer qu'en passant de 4 à 5 bits le résultat apparent passe de faux à correct ("vrai" résultat). En cas de débordement (V=1 de l'exemple sur 4 bits) quelle conclusion peut-on tirer de la comparaison de N sur 4 bits avec le signe du vrai résultat ?

Expliquer pourquoi après avoir effectué le calcul $a - b$ sur des entiers relatifs, la condition $a < b$ s'écrit "(N et $\overline{V}$) ou ($\overline{N}$ et V)" (cela concerne les cas (N,V) = (1,0) ou (0,1))

Expliquer pourquoi après avoir effectué le calcul $a - b$ sur des entiers naturels, la condition $a < b$ s'écrit $\overline{C}$

2.4 Extension de format et opérations bit à bit

Observer que l'addition sur 4 bits de naturels (5 + 11) et celles des relatifs (+5 + -5) donne en binaire la même opération : 0101 + 1011.

Poser en binaire sur 8 bits :
— l'addition des mêmes entiers naturels (5,11)

— l'addition des mêmes entiers relatifs (+5,-5)

Dans la méthode d'extension de format à 8 bits, le bit de poids fort de départ a-t-il une importance ?

Effectuer sur 8 bits entre les opérandes 0xcc et 0x55 les opérations booléennes bit à bit suivantes : or, and, xor. Expliquer le résultat obtenu

Effectuer sur 12 bits les opérations de décalage suivantes entre les opérandes 0x1f5 (entier décalé) et 3 (nombre de bits de décalage), puis 0x842 et 2 :

1. lsl (Logic Shift Left)
2. lsr (Logic Shift Right)
3. asr (Arithmetic Shift Right)

Observer les multiplications ou division effectuées (entiers naturels et entiers relatifs).

TP séances 3 et 4 : Codage des données

3.1 Déclaration de données en langage d'assemblage

Les données sont déclarées dans une zone appelée : `data`. Pour déclarer une donnée on indique la taille de sa représentation et sa valeur ; on peut aussi déclarer une zone de données non initialisées (sans valeur initiale) ce qui correspond à une réservation de place en mémoire.

Soit le lexique suivant en notation algorithmique :

```
aa : le caractère 'A'
oo : l'entier 15 sur 8 bits (1 octet)
cc : la chaine "bonjour"
rr : <'B', 3> de type <un caractère, un entier sur 1octet>
T : le tableau d'entiers sur 16 bits [0x1122, 0x3456, 0xfafd]
xx : l'entier 65 sur 8 bits (1 octet)
```

On le traduit en langage d'assemblage ARM. Le fichier `donnees.s` contient une zone `data` dans laquelle sont déclarées les données correspondant aux déclarations ci-dessus.

La directive `.byte` (respectivement `.hword`, `.word`) permet de déclarer une valeur exprimée sur 8 bits (respectivement 16, 32 bits). Une valeur peut être écrite en décimal (65) ou en hexadécimal (0x41).

Pour déclarer une chaîne, on peut utiliser la directive `.asciz` et des guillemets.

Le caractère `@` marque le début d'un commentaire, celui-ci se poursuivant jusqu'à la fin de la ligne.

```
    .data
aa: .byte 65   @ .byte 0x41
oo: .byte 15   @ .byte 0x0f
cc: .asciz "bonjour"
rr: .byte 66   @ .byte 0x42
    .byte 3
T:  .hword 0x1122
    .hword 0x3456
    .hword 0xfafd
xx: .byte 65
```

Nous allons maintenant observer le codage en mémoire de cette zone `data`. Traduire le programme `donnees.s` en binaire avec la commande :
`arm-eabi-gcc -c -mbig-endian donnees.s`.
Notez que nous utilisons l'option `-mbig-endian` dans le but de faciliter la lecture (rangement par "grands bouts"). Vous pourrez en observer le fonctionnement dans la partie 4.2.2. Vous obtenez le fichier `donnees.o`. Observez le contenu de ce fichier :
`arm-eabi-objdump -j .data -s donnees.o`

Chaque ligne comporte une adresse puis un certain nombre d'octets et enfin leur correspondance sous forme d'un caractère (quand cela a un sens). Dans quelle base les informations sont-elles affichées ? Combien d'octets sont-ils représentés sur chaque ligne ? Donnez pour chacun des octets affichés la correspondance avec les valeurs déclarées dans la zone data. Comment est codée la chaîne de caractères, en particulier comment est représentée la fin de cette chaîne ?

Nous voulons maintenant représenter tous les entiers sur 32 bits ; d'où le nouveau lexique :

```
aa : le caractère 'A'
oo : l'entier 15 sur 32 bits
cc : la chaine "bonjour"
rr : <'B', 3> de type <un caractère, un entier sur 32 bits>
T : le tableau d'entiers sur 32 bits [0x1122, 0x3456, 0xfafd]
xx : l'entier 65 sur 32 bits
```

La directive de déclaration pour définir une valeur sur 32 bits est `.word`.

Copiez le fichier `donnees.s` dans `donnees2.s` et modifiez `donnees2.s`. Compilez `donnees2.s`. Quelle est maintenant la représentation de chacun des entiers de la zone data modifiée ?

3.2 Accès à la mémoire : échange mémoire/registres

3.2.1 Lecture d'un mot de 32 bits

Le problème est le suivant : la zone `data` contient des données dont plus particulièrement un entier représenté sur 32 bits à l'adresse `xx` ; on veut copier cet entier dans un registre.

Le programme `accesmem.s` montre comment résoudre le problème. On commence par charger dans le registre `r5` l'adresse `xx` (`LDR r5, LD_xx`), puis on charge dans `r6` le mot mémoire à cette adresse (`LDR r6, [r5]`). La suite du programme permet d'afficher le contenu des registres `r5` et `r6`.

```
1  @ accesmem.s
2        .data
3  aa:  .word 24
4  xx:  .word 266
5  bb:  .word 42
6
7        .text
8        .global main
9  main:
10       LDR r5, LD_xx
11       LDR r6, [r5]
12
13   @ impression du contenu de r5
14       MOV r1, r5
15       BL EcrHexa32
16
17   @ impression du contenu de r6
18       MOV r1, r6
19       BL EcrHexa32
20
21 fin: B exit  @ terminaison immédiate du processus (plus tard on saura faire mieux)
22 LD_xx: .word xx
```

Ce programme utilise une fonction d'affichage `EcrHexa32` qui est définie dans un autre module `es.s` (Cf. chapitre **??**). Cette fonction affiche à l'écran en hexadécimal la valeur contenue dans le registre `r1` obligatoirement.

Produisez l'exécutable `accesmem` :

```
arm-eabi-gcc -c es.s
arm-eabi-gcc -c accesmem.s
arm-eabi-gcc -o accesmem accesmem.o es.o
```

Exécutez ce programme : `arm-eabi-run accesmem`. Notez les valeurs affichées. Que représente chacune d'elle ?

3.2.2 Lecture de mots de tailles différentes

Voilà un programme (`accesmem2.s`) utilisant les instructions : `LDR`, `LDRH` et `LDRB`.

Le programme `es.s` vous fournit également les fonctions d'affichage en décimal de la valeur contenue dans le registre `r1` sur 32 bits, 16 bits ou 8 bits : `EcrNdecimal32`, `EcrNdecimal16` et `EcrNdecimal8` (Cf. chapitre **??**).

Ajoutez des instructions permettant l'affichage des adresses et des valeurs lues dans la mémoire de la même façon que dans le programme précédent. Compilez de la même façon que précédemment et exécutez.

Relevez les valeurs affichées et en particulier donnez les adresses mémoire où sont rangées les valeurs 266, 42 et 12. Expliquez les différences entre elles.

```
1  @ accesmem2.s
2      .data
3  D1:     .word  266
4  D2:     .hword 42
5  D3:     .byte 12
6
7      .text
8      .global main
9  main:
10       LDR r3, LD_D1
11       LDR r4, [r3]
12
13       LDR r5, LD_D2
14       LDRH r6, [r5]
15
16       LDR r7, LD_D3
17       LDRB r8, [r7]
18
19 fin:  B exit  @ terminaison immédiate du processus (plus tard on saura faire mieux)
20
21 LD_D1:    .word D1
22 LD_D2:    .word D2
23 LD_D3:    .word D3
```

3.2.3 Ecriture en mémoire

L'instruction `STR` (respectivement `STRH`, `STRB`) permet de stocker un mot représenté sur 32 (respectivement 16, 8) bits dans la mémoire.

Le programme `ecrmem.s` affiche la valeur rangée à l'adresse `DW`, puis range à cette adresse la valeur -10 ; le mot d'adresse `DW` est alors lu et affiché. Exécutez ce programme et constatez qu'effectivement le mot d'adresse `DW` a été modifié. Modifiez le programme `ecrmem.s` pour faire le même genre de travail mais avec un mot de 16 bits rangé à l'adresse `DH` et un mot de 8 bits rangé à l'adresse `DB`.

Remarque : les adresses sont toujours représentées sur 32 bits.

```
1       .data
2  DW:      .word  0
3  DH:      .hword 0
4  DB:      .byte 0
5
6       .text
7       .global main
8  main:
9         LDR r0, LD_DW
10        LDR r1, [r0]
11        BL EcrHexa32
12        BL EcrZdecimal32
13
14        MOV r4, #-10
15        LDR r5, LD_DW
16        STR r4, [r5]
17
18        LDR r0, LD_DW
19        LDR r1, [r0]
20        BL EcrHexa32
21        BL EcrZdecimal32
22
23 fin:  B exit  @ terminaison immédiate du processus (plus tard on saura faire mieux)
24
25 LD_DW:     .word DW
26 LD_DH:     .word DH
27 LD_DB:     .word DB
```

4.1 Un premier programme en langage d'assemblage

Considérons le programme `caracteres.s`.

```
1       .data
2  cc: @ ne pas modifier cette partie
3       .byte 0x42
4       .byte 0x4f
5       .byte 0x4e
6       .byte 0x4a
7       .byte 0x4f
8       .byte 0x55
9       .byte 0x52
10      .byte 0x00          @ code de fin de chaine
11           @ la suite pourra etre modifiee
12           .word 12
13           .word 0x11223344
14      .asciz "au-revoir..."
15
16      .text
17      .global main
18 main:
19
20 @ impression de la chaine de caracteres a une adresse cc
21       LDR r1, LD_cc
22       BL EcrChaine
23
```

```
24 @ impression de la chaine "au-revoir..."
25 @ A COMPLETER
26
27 @ modification de la chaine cc
28 @ A COMPLETER
29
30 @ impression de la chaine cc modifiee
31     LDR r1, LD_cc
32     BL EcrChaine
33
34 fin: B exit  @ terminaison immédiate du processus (plus tard on saura faire mieux)
35
36 LD_cc: .word cc
```

Compilez ce programme et exécutez-le ; vous constatez qu'il affiche deux fois la chaîne de caractères d'adresse `cc`.

Modifiez ce programme pour qu'il affiche la chaîne `"BONJOUR"` sur une ligne puis la chaine `"au revoir..."` sur la ligne suivante. Il y a plusieurs façons de traiter cette question, vous pouvez essayer plusieurs solutions (c'est même conseillé) mais celle qui nous intéresse le plus ici consiste à utiliser l'indirection avec un pointeur relais. Plus précisément vous devez identifier l'adresse de début de chaque chaine (avec une étiquette) et utiliser cette étiquette pour réaliser l'affichage souhaité.

La chaîne d'adresse `cc` est formée de caractères majuscules. Modifiez le programme en ajoutant une suite d'instructions qui transforme chaque caractère majuscule en minuscule. On peut résoudre ce problème sans écrire une boucle. Compilez et exécutez votre programme.

Indication : inspirez-vous de l'exercice fait en TD1. L'opération `OU` peut être réalisée avec l'instruction `ORR`.

4.2 Alignements et "petits bouts"

4.2.1 Questions d'alignements

Voici une nouvelle zone de données à définir en langage d'assemblage :

```
x: l'entier 0x01 sur 8 bits
y: l'entier 0x02 sur 8 bits
z: l'entier 0x04 sur 8 bits
a: l'entier 0x0A0B0C0D sur 32 bits
b: l'entier 0x08 sur 8 bits
```

En vous inspirant du programme `accesmem.s`, écrivez le programme `alignements1.s` comportant la zone de données décrite ci-dessus, et une zone `text` consistant à lire les mots d'adresse `x, y, z, a, b` et à afficher sa valeur. Que constatez-vous ?

Le problème vient du fait que les mots de 32 bits doivent être placés à des adresses multiples de 4. De même les entiers représentés sur 2 octets doivent être stockés à des adresses multiples de 2.

Pour rétablir l'alignement insérer la ligne suivante :
`.balign 4`
juste avant la déclaration de l'entier `a`, et appelez ce nouveau programme `alignements2.s`.

Vérifiez que le problème est résolu.

Ecrivez un programme dans lequel vous déclarez une valeur représentée sur 16 bits et dont l'adresse n'est pas un multiple de 2 (il suffit de placer un octet devant). Reproduisez une expérience similaire à la précédente (pour rétablir un alignement sur une adresse multiple de 2 utilisez la directive `.balign 2`).

4.2.2 Questions de "petits bouts"

Reprendre l'exercice de lecture de mots de 32 bits dans la mémoire (paragraphe 3.2.1).

Observer le contenu de la zone `data` du fichier `accesmem` avec la commande : `arm-eabi-objdump -j .data -s accesmem` et retrouver les valeurs et les adresses des 3 mots déclarés dans la zone `data`.

Noter que la convention utilisée est le rangement par "petits bouts" (Cf. paragraphe **??**). Pour obtenir un rangement par "grands bouts" recompiler les fichiers source avec l'option `-mbig-endian`.

Faire le même exercice avec l'exercice du paragraphe 3.2.2.

TP séance 5 : Codage de structures de contrôle et metteur au point gdb

5.1 Accès à un tableau

On considère l'algorithme suivant :

```
lexique:
TAB : un tableau de 5 entiers représentés sur 32 bits

algorithme:
TAB[0] <-- 11
TAB[1] <-- 22
TAB[2] <-- 33
TAB[3] <-- 44
TAB[4] <-- 55
```

1. Récupérez le fichier `tableau.s`. On y a traduit en langage d'assemblage les deux premières affectations.
2. Complétez le programme de façon à réaliser l'algorithme donné en entier.
3. Compilez avec la commande : `arm-eabi-gcc -Wa,--gdwarf2 tableau.s -o tableau`[1]
4. Observez son exécution pas à pas sous `gdb` ou `ddd` (lire ce qui suit et vous référer au paragraphe dans la documentation technique).

`gdb` est un metteur au point (ou "débogueur") ; il permet de suivre l'exécution d'un programme en pas à pas c'est-à-dire une ligne de programme après l'autre ou à modifier un programme en cours d'exécution.

Nous verrons par la suite qu'un metteur au point sert aussi à chercher des erreurs dans un programme en stoppant celui-ci justement à l'endroit où l'on soupçonne l'erreur...

Pour utiliser `gdb` le programme doit avoir été compilé avec l'option `-g`, ce que vous avez fait (option `-gdwarf2`).

Exécutez le programme sous `gdb` en tapant les commandes suivantes :

1. `arm-eabi-gdb tableau`
   On lance le débogueur, nous sommes désormais dans l'environnement `gdb`.
2. `target sim`
   On active le simulateur, ce qui permet d'exécuter des instructions en langage d'assemblage ARM.
3. `load`
   On charge le programme à exécuter dont on a donné le nom à l'appel de `gdb`.

1. attention, ne pas mettre d'espace avant le `--gdwarf2`

4. `break main`
   On met un point d'arrêt juste avant l'étiquette `main`.
5. `run`
   Le programme s'exécute jusqu'au premier point d'arrêt exclu : ici, l'exécution du programme est donc arrêtée juste avant la première instruction.
6. `list`
   On voit 10 lignes du fichier source.
7. `list`
   On voit les 10 suivantes.
8. `list 10,13`
   On voit les lignes 10 à 13.
9. `info reg`
   Permet d'afficher en hexadécimal et en décimal les valeurs stockées dans tous les registres. Notez la valeur de `r15` aussi appelé `pc`, le compteur de programme. Elle représente l'adresse de la prochaine instruction qui sera exécutée.
10. `s`
    Une instruction est exécutée. `gdb` affiche une ligne du fichier source qui est la prochaine instruction (et qui n'est donc pas encore exécutée).
11. etc.

Pour l'observation de l'exécution du programme `tableau`, notez, en particulier, les valeurs successives (à chaque itération) de `r0`, le contenu de la mémoire à partir de l'adresse `debutTAB` en début de programme et après l'exécution de toutes les instructions. Sous `gdb`, pour afficher 5 mots en hexadécimal, à partir de l'adresse `debutTAB`, utilisez la commande : `x/5w &debutTAB`.

5.2 Codage d'une itération

On considère l'algorithme suivant :

```
val <-- 11
i <-- 0
tant que i <> 5
    TAB[i] <- val
    i <-- i + 1
    val <- val + 11
```

Après transformations, on l'a codé en langage d'assemblage par :

```
1     .data
2  debutTAB: .skip 5*4
3
4     .text
5     .global main
6  main:
7
8        mov r3, #11              @ val <- 11
9        mov r2, #0               @ i <- 0
10 tq:   cmp r2, #5               @ i-5 ??
11       beq fintq
12          @ i-5 <> 0
```

```
13          ldr r0, LD_debutTAB          @ r0 <- debutTAB
14          add r0, r0, r2, LSL #2  @ r0 <- r0 + r2*4 = debutTAB + i*4
15          str r3, [r0]                 @ MEM[debutTAB+i*4] <- val
16          add r2, r2, #1               @ i <- i + 1
17          add r3, r3, #11              @ val <- val + 11
18      b tq
19 fintq:   @ i-5 = 0
20
21 fin:   BX LR
22
23 LD_debutTAB : .word debutTAB
```

1. Récupérez le fichier `iteration.s`. Compilez ce programme et exécutez-le sous `gdb` ou `ddd`.
2. Quelle est la valeur contenue dans `r0` à chaque itération ?
3. Quelle est la valeur contenue dans `r2` à chaque itération ?
4. Quelle est la valeur contenue dans `r2` à la fin de l'itération, c'est-à-dire lorsque le contrôle est à l'étiquette `fintq` ?
5. Supposons que l'algorithme soit écrit avec `tant que i <= 4` au lieu de `tant que i <> 5` ; le tableau contient-t-il les même valeurs à la fin de l'itération ? Comment doit-on alors traduire ce nouveau programme ?
6. Supposons que le tableau soit maintenant un tableau de mots de 16 bits. Comment devez-vous modifier le programme ? Faire la modification et rendre le nouveau programme et les valeurs dans les registres.
7. Même question pour un tableau d'octets.

5.3 Calcul de la suite de "Syracuse"

La suite de Syracuse est définie par :

$U_0 = \textit{un entier naturel} > 0$

$U_n = U_{n-1}/2 \textit{ si } U_{n-1} \textit{ est pair}$

$\quad = U_{n-1} \times 3 + 1 \textit{ sinon}$

Cette suite converge vers 1 avec un cycle.

Calculer les valeurs de la suite pour $U_0 = 15$.

Pour calculer les différentes valeurs de cette suite, on peut écrire l'algorithme suivant :

```
lexique :
  x : un entier naturel
algorithme :
  tant que x <> 1
     si x est pair
     alors x <-- x div 2
     sinon x <-- 3 * x + 1
```

Vous allez traduire cet algorithme en langage d'assemblage et vérifier que son exécution calcule bien les éléments de la suite de Syracuse.

Quelques indications :

— L'algorithme comporte une itération dans laquelle est incluse une instruction conditionelle. Vous pouvez traduire chacune des deux constructions en utilisant un des schémas de traduction précédents. N'hésitez pas à utiliser autant d'étiquettes que vous voulez si cela vous rend le travail plus lisible.

— Pour tester si un entier est pair il suffit de regarder si son bit de poids faible (le plus à droite) est égal à 0. Pour cela vous pouvez utiliser une instruction “et logique” avec la valeur 1 ou l'instruction `TST` qui exécute la même chose.
— Pour diviser un entier par 2 il suffit de le décaler à droite de 1 position.
— Pour calculer $3 * x$ on peut calculer $2 * x + x$ et pour multiplier un entier par 2, il suffit de le décaler à gauche de 1 position.

TP séances 6 et 7 : Parcours de tableaux

6.1 Tables de multiplications

On vous propose de réaliser un programme qui remplit un tableau avec les tables de multiplication de 1 à 10 et qui l'affiche à l'écran comme sur la figure 6.1.

Le remplissage du tableau peut être réalisé de façon itérative, suivant l'algorithme :

```
 //Remplissage d'un tableau des multiplications de 1 à 10
 //table[n-1,m-1] = n*m pour n et m compris entre 1 et 10.

LEXIQUE :
   N_MAX    : l'entier 10
   Ligne    : le type tableau  sur [0..N_MAX-1] d'entiers
   table    : le tableau sur [0..N_MAX-1] de Ligne
   n_lig,n_col : deux entiers

ALGORITHME :
  pour n_lig parcourant [1..N_MAX] :
    pour n_col parcourant [1..N_MAX] :
      // produit de n_col par n_lig
      table[n_lig-1][n_col-1] <-- n_lig * n_col;
```

Questions concernant le remplissage du tableau

1. Dans quelle case du tableau `table` se trouve le produit 1*1 ?

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

FIGURE 6.1 – Tables de multiplication de 1 à 10

2. Dans quelle case du tableau `table` se trouve le produit 1*2 ?
3. Dans quelle case du tableau `table` se trouve le produit 7*9 ?
4. Dans quelle case du tableau `table` se trouve le produit 10*10 ?

Organisation du travail Dans ce TP, vous avez à écrire une séquence de deux blocs de codes distincts :

1. le premier initialise (remplit) un tableau en mémoire
2. le second affiche à l'écran le contenu du tableau stocké en mémoire

Il est fortement recommandé de ne tester qu'un bloc de code à la fois : affichage puis remplissage ou remplissage puis affichage. Les binômes peuvent même effectuer le travail en parallèle et fusionner les codes ensuite.

6.2 Affichage du tableau

On donne ci-dessous un algorithme pour afficher le tableau conformément à la figure 6.1, une fois celui-ci rempli. On utilise les fonctions d'entrée/sortie suivantes :

— `ecrire_car(c)` : affiche sans retour à la ligne le caractère de code ascii c.
— `ecrire_chn(s)` : affiche sans retour à la ligne la chaine de caractères s.
— `ecrire_int(e)` : affiche sans retour à la ligne la forme décimale de l'entier e.
— `a_la_ligne()` : provoque un retour à la ligne

```
LEXIQUE :
 N_MAX    : l'entier 10
 ESPACE   : le caractère ' '   // code ascii 32
 BARRE    : le caractère '|'   // code ascii 124
 TIRETS   : le caractère '---' // code ascii 45
 Ligne    : le type tableau  sur [0..N_MAX-1] d?entiers
 table    : le tableau sur [0..N_MAX-1] de Ligne
 n_lig,n_col : deux entiers
 mult : un entier

ALGORITHME :
  pour n_lig parcourant [0..N_MAX-1] :
       pour n_col parcourant [0..N_MAX-1] :
            ecrire_car(BARRE);
            mult <-- table[n_lig][n_col];
            si mult < 100 alors ecrire_car(ESPACE);
            si mult < 10 alors ecrire_car(ESPACE);
            ecrire_int(mult);
       ecrire_car(BARRE);
       a_la_ligne();
       répéter N_MAX fois :
            ecrire_car(BARRE);
            ecrire_chn(TIRETS);
       ecrire_car(BARRE);
       a_la_ligne();
```

Traduire en langage d'assemblage cet algorithme, récupérer le fichier `tabmult.s` et compléter la partie affichage. Tester cette partie, pour le tableau évidemment pour l'instant vide ; c'est-à-dire que l'affichage que vous devez observer est le même que celui de la figure 6.1 mais avec des zéros.

Pour la traduction des fonctions d'entrées-sorties utiliser les fonctions suivantes définies dans le fichier `es.s` :

- `EcrChn` pour implémenter `ecrire_car` et `ecrire_chn`. Pour écrire un caractère sans retour à la ligne déclarer le caractère comme chaîne.
- `EcrNdecim32` pour implémenter `ecrire_int`.
- `AlaLigne` pour implémenter `a_la_ligne`.

6.3 Remplissage du tableau

Il s'agit maintenant de traduire en langage d'assemblage l'algorithme de remplissage du tableau donné au paragraphe 6.1.

Transformer cet algorithme dans une forme adaptée à la traduction en langage d'assemblage (i.e. suppression des constructions `pour`).

Dans un premier temps, on garde telle quelle l'écriture de l'accès à un élément du tableau (`table[n_lig][n_col]`).

Pour réaliser la multiplication de deux entiers positifs vous pouvez utiliser l'instruction `MUL Rdest, Rgauche, Rdroite`, l'algorithme de multiplication par additions successives selon l'algorithme qui suit, ou un algorithme par additions et décalages (non présenté ici).

Algorithme de multiplication par additions successives :

```
LEXIQUE :
  mult, a et b : trois entiers positifs ou nuls

ALGORITHME :
    mult <-- 0;
    répéter a fois : mult <-- mult + b;
```

Important : Votre compte-rendu comportera cette version intermédiaire de la traduction. Vous devrez donc fournir deux versions (Sections 6.3.2 et 6.3.3).

6.3.1 Codage d'un tableau à 2 dimensions

Pour stocker en mémoire un tableau à 2 dimensions, on peut le transformer en un tableau à une dimension en rangeant les lignes du tableau, les unes après les autres. Chaque ligne est une suite de cases contenant chacune un élément du tableau.

Par exemple, un tableau avec 4 lignes et 6 colonnes pourra être représenté par un tableau de 4*6=24 cases.

e00	e01	e02	e03	e04	e05
e10	e11	e12	e13	e14	e15
e20	e21	e22	e23	e24	e25
e30	e31	e32	e33	e34	e35

table : e00
e01
e02
e03
e04
e05
e10
e11
...
e35

Questions

1. `table` étant l'adresse de début du tableau (i.e. du premier élément), exprimer la formule du donne l'adresse de `table[x][y]` en fonction de `table`, `x` et `y` ?.
2. Donner le langage d'assemblage correspondant à la ligne suivante (calcul d'adresse, puis écriture de la valeur en mémoire) : `table[x][y] <-- valeur`.

6.3.2 Codage du programme de multiplication (version 1)

En rassemblant les différents algorithmes que vous avez traduits, vous avez maintenant une version complète et vous pouvez compléter le fichier `tabmult.s`, le compiler, l'exécuter et vérifier vos résultats ...

Pour vérifier que votre tableau est correctement rempli, vous pouvez utiliser `gdb` ou `ddd` pour afficher le contenu de la mémoire à l'adresse `debutTab`. Vous pouvez aussi utiliser la partie affichage si celle-ci a été complètement testée, car dans le cas contraire vous n'êtes pas à l'abri d'un bug dans cette première partie.

6.3.3 Codage du programme de multiplication (version 2)

Pour parcourir le tableau à 2 dimensions, on pourrait aussi parcourir le tableau à 1 dimension du début à la fin, en utilisant une seule boucle. L'algorithme de remplissage du tableau peut alors être récrit sans utiliser de multiplication.

Questions

1. Donnez la nouvelle forme de l'algorithme complet.
2. Traduire cette version en langage d'assemblage. Reprenez la version initiale du fichier `tabmult.s`, complétez-le avec la traduction de votre algorithme.
3. Compilez votre programme, exécutez le et vérifiez vos résultats ...

Pour le compte-rendu :
Les différentes lignes de vos algorithmes doivent apparaître de façon claire sous forme de commentaire dans votre programme en langage d'assemblage. Vous donnerez aussi les conventions d'implantation des différentes variables dans les registres (c'est à dire, quel registre contient quelle variable)

6.4 Fichier `tabmult.s`

```
1  N_MAX= 10
2      .data
3  barre :   .byte '|'
4            .byte 0
5  espace :  .byte '-'
6            .byte 0
7  tirets :  .asciz "----"
8  debutTab:     .skip N_MAX*N_MAX*4   @ adresse du debut du tableau
9      .text
10     .global main
11 main: push {lr}
12     @ Programme tabmult : Affiche les tables de multiplication de de 1 a 10
13     @ remplissage du tableau
14         @ a completer...
15     @ affichage du tableau
16         @ a completer...
17     pop {lr}
18     bx lr
19
20 LD_debutTab : .word debutTab
21 LD_barre :    .word barre
22 LD_espace :   .word espace
23 LD_tirets :   .word tirets
```

TP séances 8 et 9 : Procédures, fonctions et paramètres, liens entre ARM et C

8.1 Traitement des fichiers au format bitmap : objectifs

Une image donnée au format bitmap peut recevoir des traitements spéciaux via l'utilisation de plusieurs types d'algorithmes. L'objectif de ces TP est de réaliser plusieurs fonctions en langage d'assemblage ARM capables de traiter différemment une image donnée en entrée au format bitmap.

Pour faciliter votre tâche la consigne est d'utiliser des hauteurs et largeurs d'image multiples de 8 pixels, et égales dans le cas de la réalisation des rotations.

En partant des exemples des sections suivantes, vous devez produire les fonctions qui réalisent les effets ci-dessous sur l'image :

1. Négatif ;
2. Symétries comme dans la figure 8.1 ci-après ;
3. Rotations comme dans la figure 8.1 ci-après.

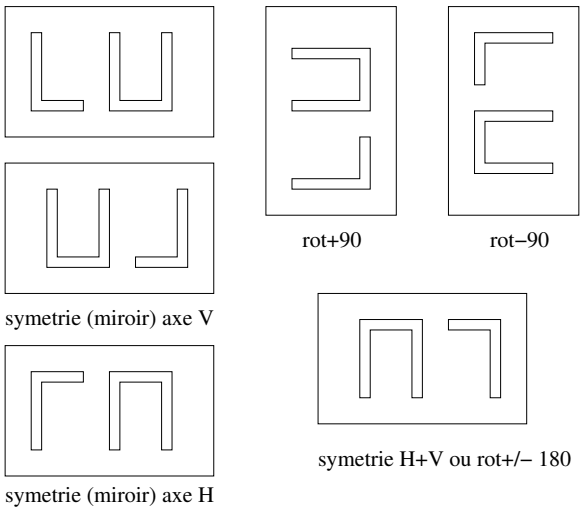


FIGURE 8.1 – Exemples d'opérations de symétrie et rotation d'une image

8.2 Découverte des transformations écrites en C

1. Extraire l'archive fournie : **tar xvzf tp_image_etu.tgz** ; et se placer sans le répertoire SRC_ETU.
2. Créer un lien symbolique de image_test.bm vers l'image à traiter :
   ln -s ../IMAGES/charlot.bm image_test.bm
3. Compiler le programme : **make**
4. Exécuter : **arm-eabi-run transformer n**
5. Afficher le résultat : **xli -pixmap resultat.bm** ou **bitmap resultat.bm**
6. Exécuter et afficher d'autres traitements : **arm-eabi-run transformer** *option*
   (*option* parmi `n`, `h`, `v`, `nv`, `nh`, `hv`, `nhv`)

Le travail consiste à coder en langage d'assemblage les traitements d'inversion vidéo (`negatif_asm.S`), les symétries (`symetrie_asm.S`) et en dernier lieu, l'écriture du fichier résultat (`afficher_contenu_asm.S`).

8.3 Négatif d'une image

Afficher l'exemple d'image qui vous est fourni : **xli charlot.bm**.

La commande **./extract_image.sh image_test.bm** génère deux fichiers indispensables à la génération du fichier exécutable (à refaire à chaque changement d'image à traiter) :

1. `image_bits_content.c` inclus dans `main.c`, qui déclare un tableau d'octets représentant les point de l'image (1 bit par pixel)
2. `image_bits_include.h` à inclure dans les fichiers `.c` et `.S`, qui définit les constantes symboliques liées à la taille de l'image (`WIDTH`, `HEIGHT`, `BYTES` et `BYTES_PER_LINE`), qui seront donc utilisables en langage d'assemblage [1]

Les fichiers `negatif.c` et `negatif_asm.S` (à compléter) sont tous les deux traités par le préprocesseur C. Le code du corps d'une fonction *fonc* compilé dépend des macros définies dans le fichier `Makefile` : par défaut le code C dans `negatif.c`, le code (à compléter) dans `negatif_asm.S` lorsque la ligne `C_NEGATIF+=-DC_`*fonc* est commentée (# en début de ligne) dans le Makefile

Par exemple, en commentant uniquement `C_NEGATIF+=-DC_NEG_OCTET`, vous pouvez tester (`make; arm-eabi-run transformer n`) le code C fourni de la fonction neg_image et votre fonction neg_octet écrite en langage d'assemblage.

En commentant les **2 lignes** de la forme `C_NEGATIF+=`, vous compilez votre version totalement en langage d'assemblage du traitement d'inversion.

Compléter, compiler et tester le fichier `negatif_asm.S`.

8.4 Symétries d'une image

Compléter, compiler et tester le fichier `symetrie_asm.S`.

1. Exemple : `ldr r8,LD_bytes` avec plus loin `LD_bytes: .word BYTES`

Sur le même principe que précédemment, les lignes `C_SYMETRY+=` permettent pour chaque fonction de conserver le code C initial ou de tester vôtre traduction en langage d'assemblage.

La symétrie la plus intéressante à traiter est celle selon l'axe vertical.

8.5 Affichage de contenu

Ecrire votre propre version de la fonction afficher_contenu. Pour la tester, remplacer afficher_contenu.o par afficher_contenu_asm.o dans la définition de OBJS.

8.6 Annexes

negatif.c

```
1  unsigned char neg_octet(unsigned char c) {
2     return ~c;}
3
4  void neg_image (unsigned char *address) {
5    unsigned char *adr;
6    for (adr=address;adr <address+BYTES;adr++)
7       *adr = neg_octet(*adr);}
```

symetrie.c

```
1  void symetrie_octet (unsigned char *adresse) {
2    unsigned char octet;
3      octet = *adresse;
4    // echange de quartets adjacents
5    octet = (octet & 0xF0) >> 4 | (octet & 0x0F) <<4;
6    // echange de doublets adjacents
7    octet = (octet & 0xCC) >> 2 | (octet & 0x33) <<2;
8    // echange de bits adjacents
9    octet = (octet & 0xAA)>> 1 | (octet & 0x55) <<1;
10   *adresse = octet;}
11
12 void permuter_cols (unsigned char *tab, unsigned int col) {
13     unsigned char tmp;
14     tmp = tab[BYTES_PER_LINE -1 - col];
15     tab [BYTES_PER_LINE -1 - col] = tab [col];
16     tab[col] = tmp;}
17
18 void symetrie_axe_v (unsigned char *image) {
19     unsigned int position;
20     unsigned int li,col;
21     unsigned char *adresse;
22     // symetriser chaque octet
23     for (position = 0; position < BYTES; position++)
24         symetrie_octet (image+position);
25     // symetrie verticale octet par octet
26     for (li=0;li<HEIGHT;li++) {
27        adresse = image+li*BYTES_PER_LINE;
28        for (col=0; col<BYTES_PER_LINE/2;col++) {
29           permuter_cols (adresse, col); }}}
30
31 void permuter_lignes (ligne_t *tab, unsigned int li) {
```

```
32    unsigned char tmp;
33    tmp = tab[li][0];
34    tab[li][0] = tab [HEIGHT -1 - li][0];
35    tab[HEIGHT -1 - li][0] = tmp;}
36
37 void symetrie_axe_h (unsigned char *image) {
38    unsigned int li,col;
39    for (li=0;li<HEIGHT/2;li++)
40       for (col=0; col<BYTES_PER_LINE;col++)
41         permuter_lignes ((ligne_t *) (image+col), li);}
```

TP séances 10 et 11 : Programmation fonctionnelle (MapRed)

10.1 Application d'une fonction à tous les éléments d'un tableau

On considère le lexique suivant :

```
. Ent8 : entier sur [-128 .. + 127]
. NMAX : constante de type entier >= 0 et <= 255
. EntN : entier sur [0 .. NMAX]
. TabEnt8 : tableau sur [0 .. NMAX - 1] d'entiers du type Ent8
. FoncMapEnt8 : adresse d'une fonction avec un paramètre du type Ent8 et un
retour du type Ent8 aussi
. saisir_tab : procédure avec deux paramètres des types : TabEnt8 et EntN {
saisir_tab(t, n) : saisit le contenu des n premiers entiers du type Ent8
dans un tableau t}
. afficher_tab : procédure avec deux paramètres des types : TabEnt8 et EntN {
afficher_tab(t, n) : affiche le contenu des n premiers entiers du type Ent8
qui sont dans un tableau t}
. map : procédure avec 4 paramètres : TabEnt8, EntN, TabEnt8 et FoncMapEnt8 {
map(t1, n, t2, f) :t1 est un tableau qui contient une séquence de n entiers
du type Ent8, déjà t2 est un tableau qui contient la séquence avec les résultats
du type Ent8 de la fonction f du type FoncMapEnt8 : [f(t1[0]), f(t1[1]), ..., f(t1[n-1])]}
```

On s'intéresse à la procédure `map`. Une réalisation en est donnée par l'algorithme suivant :

```
map(t1, n, t2, f) :
   i : entier du type EntN
   i <- 0
   tant que i != n
      t2[i] <- f(t1[i])
      i <- i + 1
```

On se propose de coder cette procédure en langage d'assemblage. On convient que :
- l'adresse du tableau `t1` est passée dans le registre `r0`
- la taille n de la séquence est passée dans le registre `r1`
- l'adresse du tableau résultat `t2` est passée dans le registre `r2`
- l'adresse de la fonction `f` est passée dans le registre `r3`

D'autre part, pour l'appel de la fonction `f` on convient que :
- l'entier donné est passé dans le registre `r3`
- le résultat calculé par la fonction est produit dans le registre `r4`

Note : Pour appeler une fonction dont l'adresse est dans un registre on utilise l'instruction ARM `BLX` (Cf. paragraph **??**).

De même, la convention d'appel des procédures `saisir_tab` et `afficher_tab` est la suivante :
- l'adresse du tableau `t` est passée dans le registre `r0`
- le nombre n d'éléments à afficher est passé dans le registre `r1`

Exercice 1 :

1. Traduire l'algorithme en langage d'assemblage Arm : compléter la définition de la procédure `map` (fichier `map.s`).
2. Compléter le corps de la procédure principale `main` du fichier `essai-map.s` : vous devez ajouter aux endroits voulus de ce fichier deux appels à la procédure `map` et pour chacun d'eux un appel à la procédure auxillaire `afficher_tab` (fichier `gestion_tab.s`).

   La procédure `map` sera invoquée une première fois avec la fonction `plus_un` comme un paramètre telle que : $plus\_un(x) = x+1$ et une seconde fois avec la fonction `carre` telle que : $carre(x) = x^2$ (fichier `fg.s`).
3. Compilez et testez le programme (faites un fichier `Makefile` permettant d'utiliser la commande `make` pour compiler le programme).

10.2 Réduction d'un tableau à une valeur

10.2.1 Calcul de $\sum_{i=0}^{n-1} T[i]$

On ajoute les éléments suivants dans le lexique de la question précédente :

```
. Ent32 : entier naturel sur 32 bits
. FoncRedEnt8 : adresse d'une fonction avec deux paramètres du type Ent8 et un
retour du type Ent32
. red : fonction avec quatre paramètres des types TabEnt8, EntN, Ent8 et
FoncRedEnt8 et un retour du type Ent32 {
   si n > 0 : red(t, n, vi, g) = g( ... (g(g(g(vi, t[0]), t[1]), t[2]), ..., t[n - 1]) ... )
   si n = 0 : red(t, 0, vi, g) = vi}
```

Si g est la fonction *somme* telle que : $somme(x, y) = x + y$, alors $red(t, n, 0, somme) = \sum_{i=0}^{n-1} t[i]$.

Une réalisation de la fonction `red` est donnée par l'algorithme suivant :

```
red(t, n, vi, g) :
   i : entier du type EntN
   acc : entier du type Ent32
   i <- 0
   acc <- vi
   tant que i != n
      acc <- g(acc, t[i])
      i <- i + 1
   retour acc
```

Pour coder cette fonction en langage d'assemblage, on convient que :
- l'adresse du tableau `t` est passée dans le registre `r0`
- la taille n de la séquence à traiter est passée dans le registre `r1`
- la valeur initiale du calcul est passée dans le registre `r2`
- l'adresse de la fonction `g` est passée dans le registre `r3`

— le résultat calculé par la fonction est produit dans le registre `r4`

D'autre part, pour l'appel de la fonction `g` on convient que :
— les données sont passées dans les registres `r0` et `r1`
— le résultat calculé par la fonction est produit dans le registre `r2`

Exercice 2 :

1. Traduire l'algorithme en langage d'assemblage Arm : compléter la définition de la fonction `red` (fichier `red.s`).
2. Compléter le corps de la procédure principale `main` du fichier `essai-red.s`. Vous devez ajouter aux endroits voulus de ce fichier un appel à la fonction `red` ainsi qu'un appel à l'une des deux procédures auxillaires `EcrZdecimal32` ou `EcrZdecim32` (cf. fichier `es.s`).

   La fonction `red` sera invoquée avec la fonction `somme` comme un paramètre telle que : $somme(x, y) = x + y$ (fichier `fg.s`).
3. Compilez et testez le programme.

10.2.2 Calcul de $\prod_{i=0}^{n-1} T[i]$

Exercice 3 :

1. Quels sont les paramètres à passer à la fonction `red` pour effectuer le calcul du produit $\prod_{i=0}^{n-1} T[i]$?
2. Quelle est la fonction qui remplace `g` de l'exercice précédent ?
3. Ajoutez au programme main de l'exercice précédent une invocation de la fonction `red` permettant de réaliser ce calcul.
4. Compilez et testez le programme modifié.

10.3 Passage de paramètres par la pile

Exercice 4 :

On veut réaliser une nouvelle version de la procédure `map`, ou de la fonction `red`, en utilisant cette fois-ci le passage de paramètres *par la pile* plutôt que par les registres. Ainsi, vous pourrez avoir une définition récursive de ces fonctions. Choisissez une organisation de la pile appropriée pour le passage des paramètres (le schéma correspondant sera à inclure dans le compte-rendu du TP), puis réalisez une nouvelle version du programme (dans les fichier `map2.s` et `essai-map2.s`, ou `red2.s` et `essai-red2.s`, par exemple). S'il vous reste du temps, faites deux versions de vos programmes : une version itérative et une version récursive et comparez ces deux versions.

TP séance 12 : Etude du code produit par gcc, optimisations

Le code en langage d'assemblage ARM d'un programme en C peut être produit sans optimisation par le compilateur `gcc` avec l'option explicite `-O0`. Dans ce TP, nous allons observer ce qu'un compilateur peut aussi faire en optimisant en mode `-O2` pour voir l'effet des optimisations.
À la première compilation (avec `-O0`), vous pouvez faire le pari de ce qui peut se produire avec les optimisations. Si quelques questions surgissent sur les optimisations et résultats possibles, **noter** ces questions dans un coin, et **prévoir** les expérimentations à faire pour lever ces doutes et observer quelles hypothèses étaient les bonnes.
Dans tous les cas, pour ce TP, n'hésitez pas à faire d'autres essais que ceux qui sont proposés.

12.1 Un premier exemple

Soit un programme écrit en langage C dans le fichier `premier.c`.

```
 1 #include "stdio.h"
 2 #include "string.h"
 3
 4 #define N 10
 5
 6 int main () {
 7 char chaine [N] ;
 8 int i ;
 9
10     printf ("Donner une chaine de longueur inferieure a %d:\n", N);
11     fgets (chaine, N, stdin);
12     printf ("la chaine lue est : %s\n",chaine);
13     i = strlen (chaine) ;
14     printf ("la longueur de la chaine lue est : %d\n", i);
15 }
```

Sans optimisation : Nous compilons ce fichier sans optimisations (option `-O0`) et produisons le code en langage d'assemblage ARM (option `-S`) avec la commande suivante : `arm-eabi-gcc -O0 -S premier.c`.
Le code en langage d'assemblage est produit dans le fichier `premier.s`. Faites une copie de ce fichier puis observez ce fichier.

1. Le premier appel à la fonction `printf` a 2 paramètres : une chaîne de caractères et un entier. Ces paramètres sont passés dans des registres, lesquels ?
2. Observez maintenant l'appel à la fonction `fgets`. Retrouvez ses paramètres dans le code.
3. La fonction `strlen` a un paramètre et un résultat. Où sont rangées ces informations dans le code ?

4. Déduire du code la convention utilisée par le compilateur pour le passage des paramètres et le retour des résultats de fonctions.
5. Quel est l'effet des 3 premières instructions du code assembleur de `main` ?
6. Quel est l'effet des 3 dernières instructions du code assembleur de `main` ?

Avec les optimisations. Compilez ce programme avec un bon niveau d'optimisations (option `-O2`) avec la commande suivante : `arm-eabi-gcc -O2 -S premier.c`. Etudiez le code en langage d'assemblage ARM produit dans le fichier `premier.s` et comparez avec la copie du code étudié précédement.

12.2 Programme avec une procédure qui a beaucoup de paramètres

12.2.1 Premier essai

Considérons le programme `bcp_param.c` écrit en langage `C` suivant :

```
 1 include "stdio.h"
 2
 3 static long int Somme (long int a1, long int a2, long int a3, long int a4, long int a5,
 4                        long int a6, long int a7, long int a8, long int a9, long int a10) {
 5 long int x1,x2,x3,x4,x5,x6,x7,x8,x9,x10;
 6 long int y;
 7
 8     x1=a1+1; x2=a2+1; x3=a3+1; x4=a4+1; x5=a5+1;
 9     x6=a6+1; x7=a7+1; x8=a8+1; x9=a9+1; x10=a10+1;
10     y = x1+x2+x3+x4+x5+x6+x7+x8+x9+x10;
11     return (y);
12 }
13
14 int main () {
15 long int z;
16
17     z = Somme (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
18     printf("La somme des entiers de 1 a 10 plus 10 vaut %d\n", z);
19 }
```

Sans optimisation : Le code produit sans optimisation est dans le fichier `bcp_param.s`, il est obtenu par la commande : `arm-eabi-gcc -O0 -S bcp_param.c`. Faites une copie de ce fichier puis observez ce fichier.

1. Observez le code du `main`. Etudier le contenu de la pile avant l'appel `bl Somme`. Comment sont passés les paramètres à la fonction `Somme` ?
2. Où est rangé le résultat rendu par la fonction `Somme` ?
3. Où est rangée la variable locale `z` ?
4. Observez le code de la fonction `Somme`. Dessiner la pile et retrouvez comment sont récupérés les paramètres. Où sont rangées les variables locales : `x1,x2,x3,x4,x5,x6,x7,x8,x9,x10` et `y` ?

Avec les optimisations. Compilez ce programme avec la commande : `arm-eabi-gcc -O2 -S bcp_param.c`. Etudiez le programme produit dans `bcp_param.s` et comparez avec la copie du code étudié précédement.

12.2.2 Suite

Modifier le programme précédent de la façon suivante :

```
 1 #include "stdio.h"
 2
 3 static long int Somme (long int a1, long int a2, long int a3, long int a4, long
int a5,
 4                        long int a6, long int a7, long int a8, long int a9, long
int a10) {
 5 long int x1,x2,x3,x4,x5,x6,x7,x8,x9,x10;
 6 long int y;
 7
 8     x1=a1+1; x2=a2+1; x3=a3+1; x4=a4+1; x5=a5+1;
 9     x6=a6+1; x7=a7+1; x8=a8+1; x9=a9+1; x10=a10+1;
10     y = x1+x2+x3+x4+x5+x6+x7+x8+x9+x10;
11     return (y);
12 }
13
14 int main () {
15 long int z; long int u1, u2, u3, u4, u5, u6, u7, u8, u9, u10;
16
17     scanf ("%d", &u1);
18     scanf ("%d", &u2);
19     scanf ("%d", &u3);
20     scanf ("%d", &u4);
21     scanf ("%d", &u5);
22     scanf ("%d", &u6);
23     scanf ("%d", &u7);
24     scanf ("%d", &u8);
25     scanf ("%d", &u9);
26     scanf ("%d", &u10);
27     z = Somme (u1, u2, u3, u4, u5, u6, u7, u8, u9, u10);
28     printf("La somme des entiers vaut %d\n", z);
29 }
```

Compilez ce programme avec ou sans optimisation, étudiez les codes produits, comparez avec les versions précédentes.

12.2.3 Dernière tentative

Modifier le programme précédent de la façon suivante :

```
 1 #include "stdio.h"
 2
 3 static long int Somme (long int a1, long int a2, long int a3, long int a4, long
int a5,
 4                        long int a6, long int a7, long int a8, long int a9, long
int a10) {
 5 long int x1,x2,x3,x4,x5,x6,x7,x8,x9,x10;
 6 long int y;
 7
 8     x1=a1+1; x2=a2+1; x3=a3+1; x4=a4+1; x5=a5+1;
 9     x6=a6+1; x7=a7+1; x8=a8+1; x9=a9+1; x10=a10+1;
10     y = x1+x2+x3+x4+x5+x6+x7+x8+x9+x10;
11     return (y);
12 }
```

```
13
14 int main () {
15 long int z; long int u;
16
17     scanf ("%d", &u);
18     z = Somme (1, 2, 3, 4, u, 6, 7, 8, 9, 10);
19     printf("La somme des entiers vaut %d\n", z);
20 }
```

Compilez ce programme, étudiez le code produit, comparez avec les versions précédentes.

12.3 Les variables locales peuvent prendre beaucoup de place

Considérons le programme `var_pile.c` écrit en langage `C` suivant :

```
 1 #include "stdio.h"
 2
 3 #define N 100
 4
 5 short int Compare2Chaines (char *s1, char *s2) {
 6 char *p1, *p2 ;
 7
 8    p1 = s1 ; p2 = s2 ;
 9    while ( *p1 && *p2 && (*p1 == *p2) ) {
10       p1++ ; p2++ ;
11    }
12    return (*p1 == 0) && (*p2 == 0) ;
13 }
14
15 int main () {
16 short int      r ;
17 char    chaine1 [N], chaine2[N] ;
18
19    printf("Chaine 1, de moins de 99 caracteres : \n");
20    fgets (chaine1, N, stdin);
21    printf("Chaine 2, de moins de 99 caracteres : \n");
22    fgets (chaine2, N, stdin);
23
24    r = Compare2Chaines (chaine1,chaine2);
25
26    printf("Sont-elles egales ? %s !\n", (r ? "oui" : "non"));
27 }
```

Sans optimisation : Le code produit sans optimisation est dans le fichier `var_pile.s`, il est donné par la commande : `arm-eabi-gcc -O0 -S var_pile.c`.

1. Dans le `main` le compilateur réserve 208 octets. Comment sont-ils utilisés ?
2. Quels sont les paramètres de la fonction `Compare2Chaines` ?
3. Observez le code suivant le retour de l'appel à `Compare2Chaines`. Commentez précisément les lignes entre `mov r3, r0` et `mov r1, r3`. Quels sont les paramètres passés à la fonction `printf` qui suit ?
4. Commentez le code de la fonction `Compare2Chaines`. Comment est généré le code d'une instruction `while` ?

Avec optimisation : Compilez le programme avec la commande : `arm-eabi-gcc -O2 -S var_pile.c`. Etudiez le programme produit dans `var_pile.s` et comparez avec le code étudié précédemment.

12.4 Le programmeur peut aussi aider ...

Soit le programme suivant, en langage C, qui compte le nombre de bits 0 et 1 (N_0 et N_1) à 1 pour les entiers N entre 1 et NMax à l'aide d'une double boucle.

```
 1 #include "stdio.h"
 2 #define NMax 1000000000
 3
 4 int main() {
 5 int i,j,zero,tab[2]; i=NMax;zero=0;tab[0]=0;tab[1]=0;
 6 for(i=NMax;i;i--)   {
 7   for(j=1;j>=0;j--) {
 8     if (i&(1<<j)) {
 9       tab[j]++;}
10     if (j==0) {
11         zero++;}}}
12 printf("%d - %d - %d\n",tab[0],tab[1],zero);
13 return;}
```

La boucle extérieure, sur i, passe en revue tous les entiers entre 1 et Nmax (en sens inverse). La boucle intérieure, sur j, compte les 0 pour les bits en position j (j=1 puis j=0). Selon la théorie, deux boucles imbriquées peuvent être réécrites en une seule boucle et parfois cela permet d'éviter des branchements coûteux ...

- Réécrire le programme en conservant le traitement effectué avec une seule boucle globale. Conserver la gestion de i et de j (en particulier les lignes à l'intérieur de la boucle interne, i.e. les lignes 8, 9, 10 et 11, ne doivent pas être modifiées)
- Réécrire ce programme une seconde fois en déroulant la boucle intérieure, c'est à dire en faisant disparaître j.
- Discuter sur le gain en temps susceptible d'être obtenu avec les deux programmes précédents selon les options de compilations utilisées.