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 INF, MIN et MIN-int

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

(comprendre pour programmer efficacement et sans bug)

Documentation Technique
Sujets des Travaux Dirigés
Sujets des Travaux Pratiques

Année Universitaire 2020 / 2021

Table des matières

Première partie

Travaux Dirigés

Chapitre 1

TD séance 1 : Codage

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. Quelle opération faut-il faire ?

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 001 00011 0001 ?
Quel est la date associée au code 111 11111 1111 ?

1.4 Code d'une instruction ARM

C'est un autre type de codage par champs.

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

Exercice à faire à la maison : codage d'une instruction `add`.

1.5 Codage d'un nombre entre 16 et 255

Combien faut-il de bits ? Coder les valeurs 17, 67, 188 en binaire et en hexadécimal. En déduire une méthode rapide de passage binaire vers hexadécimal ainsi que l'inverse.

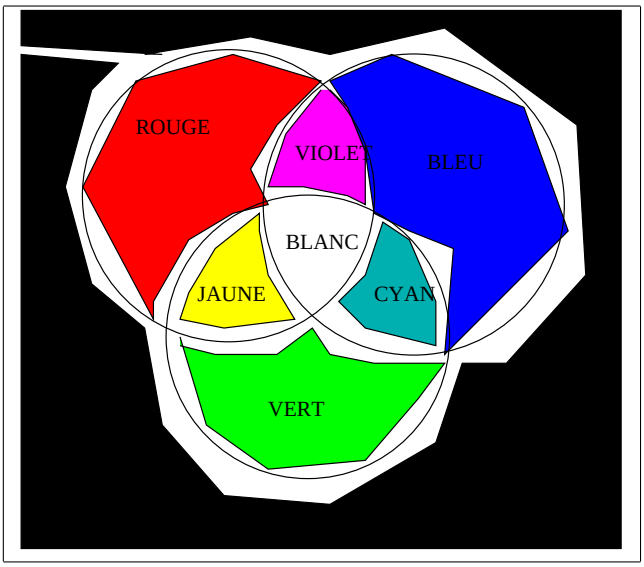


FIGURE 1.1 – Codage de couleurs

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 [1].

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)

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

1. on ne dit pas décimale car ce mot est impropre ici mais c'est quand même le mot habituel

Chapitre 2

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 :
$178_{10} = B2_{16}$: $178/16\ quotient\ 11, reste\ \boxed{2}\ (poids\ faible), 11/16$: $quotient\ 0\ reste\ 11(\boxed{\text{B}}\ (poids fort))$

1. modulo = reste de la division entière

$11_{10} = 1011_2 = 5 * 2 + \boxed{1}$ (*poids faible*), $5 = 2 * 2 + \boxed{1}, 2 = 1 * 2 + \boxed{0}, 1 = 0 * 2 + \boxed{1}$ (*poids fort*)
$1001101101_2 \rightarrow \mathbf{00}10\ 0110\ 1101 \rightarrow 26B_{16}$

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 $somme \geq base$.

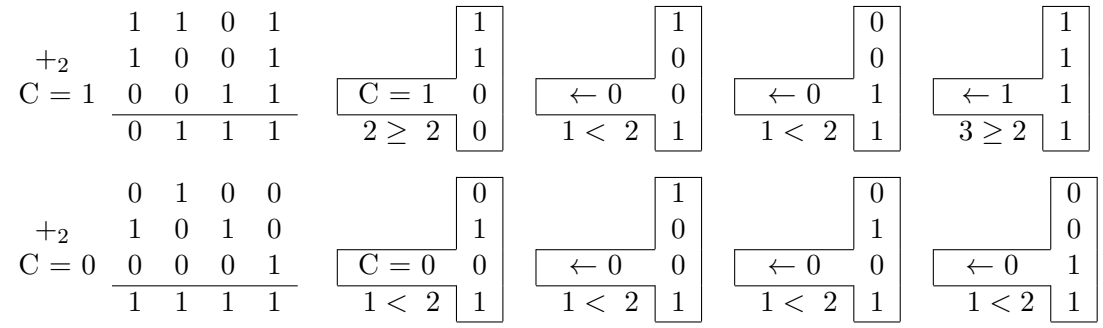
	a_3	a_2	a_1	a_0	opérande gauche
$+_{base}$	b_3	b_2	b_1	b_0	opérande droit
$C = c_4$	c_3	c_2	c_1	c_0	retenues
	r_3	r_2	r_1	r_0	résultat apparent

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

	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 ≥ 10	4	12 ≥ 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 ≥ 2	1	2 ≥ 2	0	1 < 2	1			

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

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 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 $e_{n-1}\ e_{n-2}\ e_{n-3} \ldots\ e_1\ e_0$ 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 : $[0 \ldots 2^n - 1]$.

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

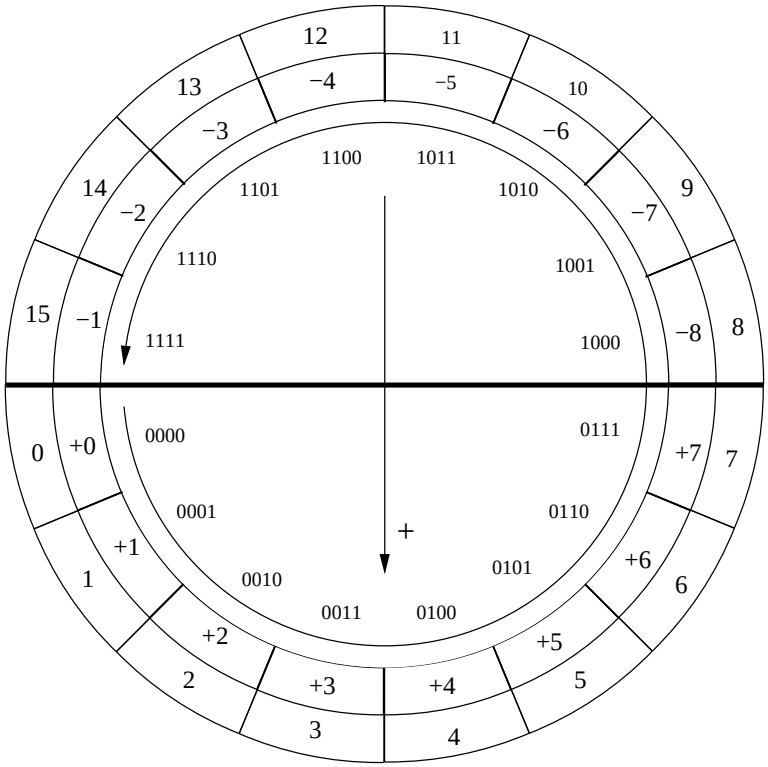


FIGURE 2.1 – Représentation d'entiers naturels et signés sur 4 bits

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 $somme > a_i$,

						sortant		entrant
	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	e_{i+1}	e_i	
	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		← 1	1		← 1	0		← 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		← 1	1		← 1	0		← 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		← 1	1		← 1	1		← 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, 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...

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.

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

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, 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, 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 2.2 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, 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 2.2 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 $x/8$?

Chapitre 3

TD séance 3 : Langage machine, codage des données

3.1 Sujet du TD

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`

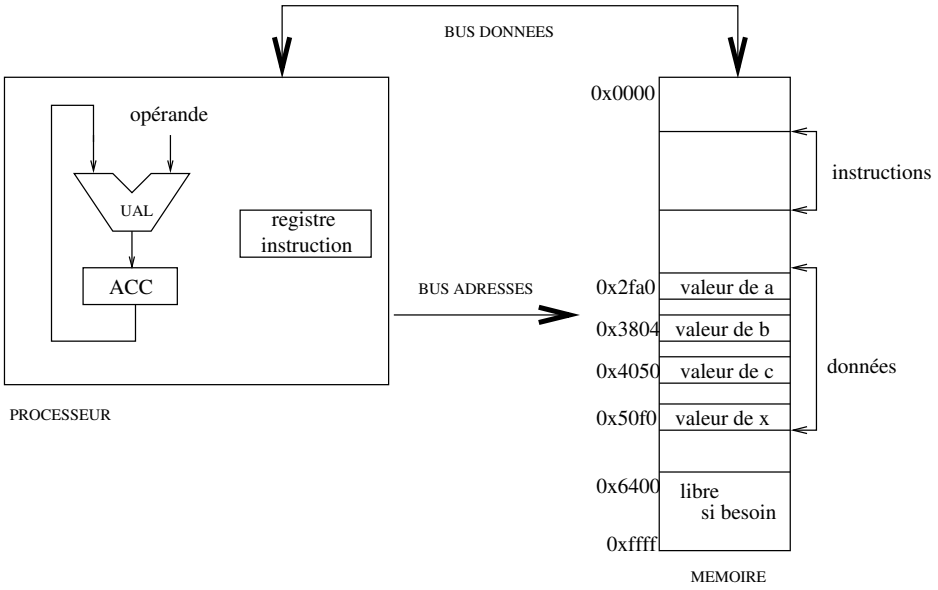


FIGURE 3.1 – Structure d'une machine à accumulateur

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

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 la soustraction.

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.

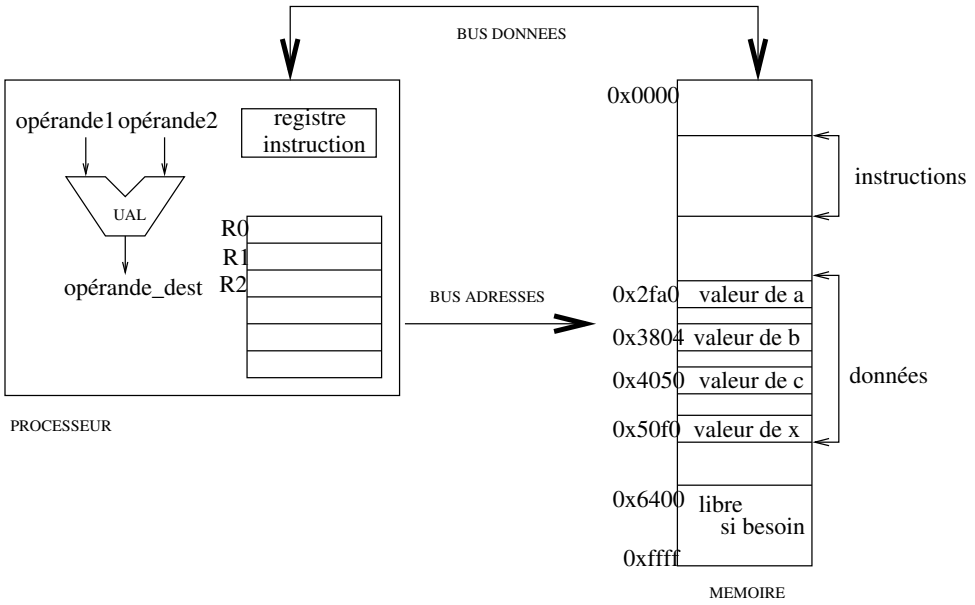


FIGURE 3.2 – Structure d'une machine générale à registres

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 :

— 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]`.

Chapitre 4

TD séance 4 : Codage des données (suite)

4.1 Données en mémoire

(rappel) On travaille sur un programme écrit en langage d'assemblage `ARM` qui exécute l'instruction : `x := (a + b + c) - (x - a - 214)`. Dans la pratique, ce n'est pas nous qui fixons les adresses, mais les outils de traduction et/ou de chargement en mémoire et nous, on peut utiliser des étiquettes ...

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
```

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. Dessiner le contenu de la zone de données en exprimant les valeurs des différentes données en hexadécimal (en faisant apparaître les différents octets).
2. Ajouter des commentaires au programme explicitant chacune des lignes de code.

On traduit le programme en binaire en fixant les adresses de début de la zone `text` et de la zone `data` :

```
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`), on regarde la traduction obtenue.

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. Quelle est 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` ?

Codage des instructions `ldr` et `str` : La figure 4.1 donne un sous-ensemble des règles de codage des instructions `ldr` et `str`, suffisant pour traiter les exercices précédents. On peut par exemple coder : `ldr rd, [rn, +/-déplacement]` ; le bit `U` code le signe du déplacement (1 pour +, 0 pour −) et le bit `L` vaut 1 pour `ldr` et 0 pour `str`.

31 28	27 24	23	22 21	20	19 16	15 12	11 0
cond	0 1 0 1	U	0 0	L	rn	rd	deplacement

FIGURE 4.1 – Codage des instructions `ldr` et `str`

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 4.2 – Tableau de nombres

Chapitre 5

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=b : c <-- a-b        sinon: @ a!=b : c <-- a+b
       B finsi                           B finsi
sinon: @ a!=b : c <-- a+b       alors: @ a=b : c <-- a-b
finsi:                          finsi:
```

Exercices :

1. Comprendre l'évolution du contrôle (compteur de programme, valeur des codes de conditions arithmétiques) pour chacune des deux solutions.
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     .data
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 r4, r4, #4  @ *
13    str r5, [r4]    @ *
14
15    @ a completer
16
17 fin: bx lr
18
19 LD_debutTAB : .word debutTAB
```

A 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.

Exercices : Compléter ce programme de façon à réaliser les dernières affectations. Reprendre le même problème avec un tableau de mots de 16 bits. Reprendre le même problème avec un tableau d'octets.

5.3 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 16 bits ?
4. Lors de l'exécution du programme précédent on constate que la valeur contenue dans le registre `r0` 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 condtion 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.

5.4 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
```

Chapitre 6

TD séance 7 : Fonctions : paramètres et résultat

6.1 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: ...          Pbis :  mov lr,pc  @ équivalent de bl
   bl Q            ...                  b   Q      @ en 2 instructions
   ...             ...                  ...        @ lr repère ici (pc+2 instr)
                   bx lr
```

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

6.2.1 Présentation et structure du code

La fonction **coderfonc** et la procédure **coderproc** effectuent une translation circulaire [1] à droite dans l'alphabet : 'a' $\underrightarrow{3}$ coder('a',3)='d', 'g' $\underrightarrow{4}$ coder('g',4)='k' et 'z' $\underrightarrow{2}$ 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 9 après le cours présentant les paramètres de type adresse.

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

```
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
```

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.

6.2.2 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 : copier le contenu de cdec dans une variable res stockée en mémoire (section data ou bss [3]) nécessite 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.

6.2.3 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 résultat.
2. l'écriture dans résultat 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)

6.3 Exemple à deux niveaux d'appel : factorielle itérative

6.3.1 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

6.3.2 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
```

6.3.3 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'affecation 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.

6.4 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.

Chapitre 7

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

7.1 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 7.1).

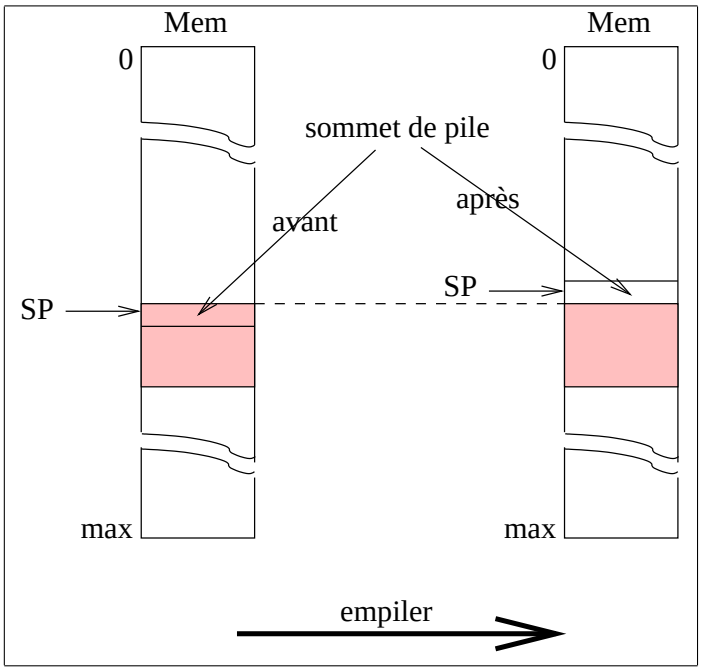


FIGURE 7.1 – Mise en oeuvre de la pile. La pile progresse vers les adresses **décroissantes**, le pointeur de pile repère la **dernière** information empilée

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 ecrira `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` ?

7.2 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
14  A2= push {r0}
1c  bl 60 (B)
20  A3= mov r5, #28
24  bl 80 (C)
28  A4= pop {r0}
```

```
60  B1= push {r0}
68  B2= add r0, r0, # 1
6c  B3= pop {r0}
74  bx lr
```

```
80  C1= mov r0, # 47
84  bl 60 (B)
88  C2= push {r0}
90  bl si condX 80 (C)
94  C3= mov r2, r5
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	?	?	?

Chapitre 8

TD séance 9 : Correction partiel

Correction du contrôle donné pendant la semaine de partiel.

Chapitre 9

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

9.1 Gestion des paramètres et des variables dans la pile

Reprendre les fonctions `codefonc`, `fact0` et `mul` 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.

9.2 Paramètre passé par adresse

9.2.1 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.

9.2.2 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.

Chapitre 10

TD séance 11 : Organisation d'un processeur : une machine à pile

10.1 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 composé 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 donné dans la figure 10.1.

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 facilité 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 10.2 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 10.1 – Les intructions de la machine à pile

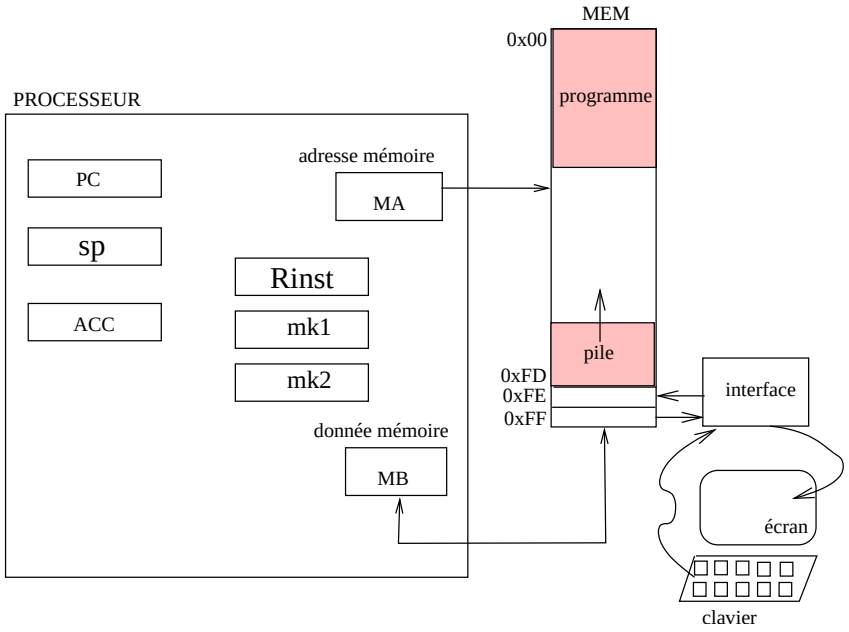


FIGURE 10.2 – La machine à pile et sa mémoire

10.1.1 Représentation en mémoire d'un programme

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

```
load# 3
push-acc
push-acc
add
pop-acc
```

10.1.2 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.

Que se passe-t-il si le programmeur empile beaucoup ? Et que la pile "marche" sur le programme qui commence lui à l'adresse 0 ? Comment peut-on éviter ce problème ?

10.2 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.

10.2.1 Algorithme

Donner l'algorithme d'interprétation des instructions.

10.2.2 Fonctionnement de l'algorithme

Donner les différentes valeurs contenues dans les registres non visibles du processeur au cours de l'interpétation du programme donné en 10.1.1.

10.3 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 10.3).

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`

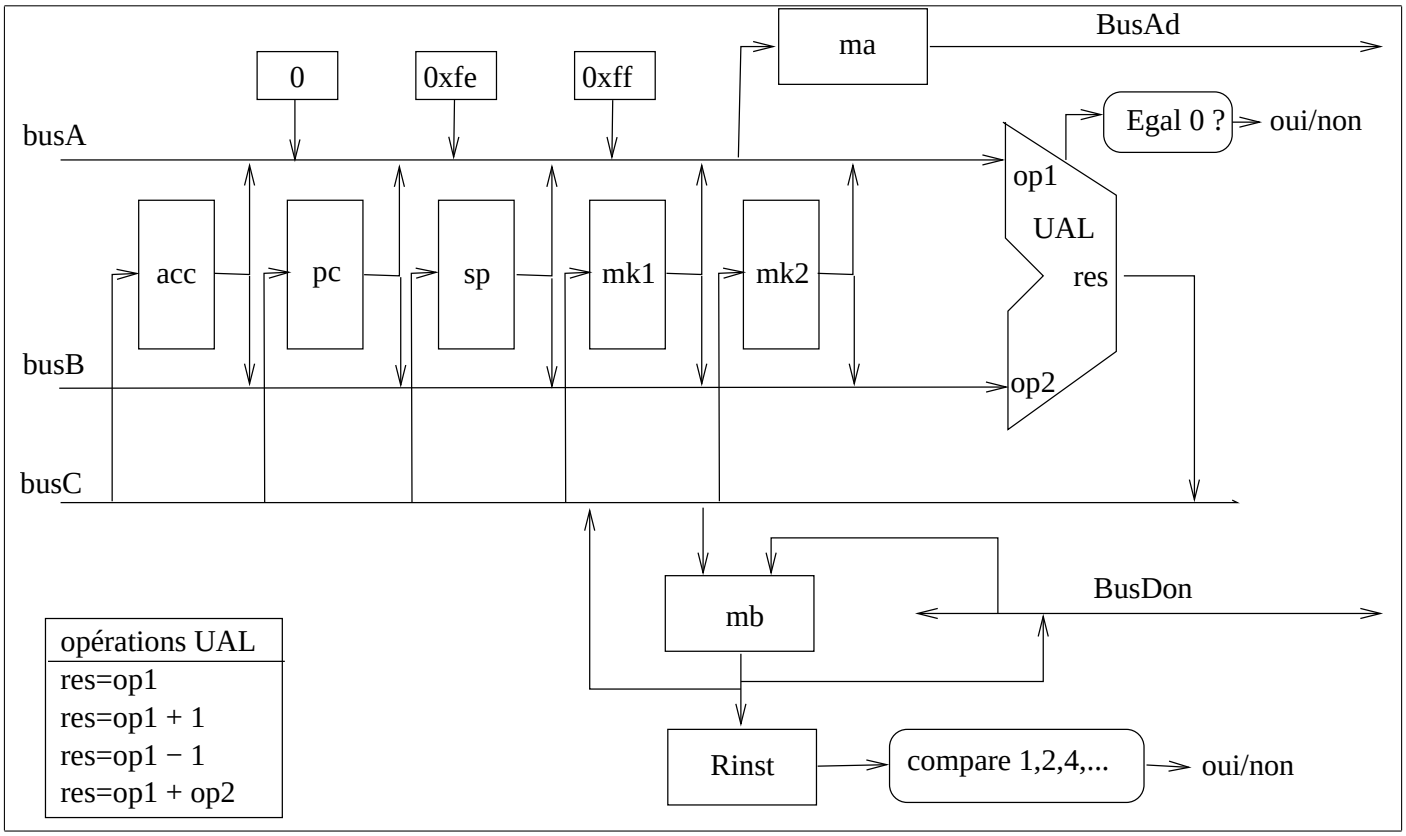


FIGURE 10.3 – Organisation de la machine à pile

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`, etc.

Par ailleurs, le "calcul" `acc = acc + 0` permet de tester si `acc` est nul où non.

10.3.1 Séquence de micro-actions pour une instruction

Ecrire la suite d'actions élémentaires (micro-actions) de la liste ci-dessus pour chacune des instructions.

10.3.2 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.

10.4 Un autre exemple

Voici un programme pour cette machine :

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

10.4.1 Questions

1. 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.
2. Donner l'évolution des valeurs dans les registres et dans la pile lors de l'exécution de ce programme.
3. Donner la trace en terme d'états du graphe de contrôle du processeur lors de l'exécution de ce programme.

10.5 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.

10.5.1 Temps de calcul d'une instruction

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 le graphe de contrôle proposé dans le paragraphe 10.3.2.

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

10.5.2 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. ?

Chapitre 11

TD séance 12 : Etude du code produit par le compilateur `arm-eabi-gcc`, optimisations

TD préparatoire au TP (si possible respecter l'ordre et faire le TD avant le TP)

Le TP consiste à refaire les compilations en mode `-O2` poru voir l'effet des optimisations. En TD, vous pouvez faire le pari de ce que peut se produire (vous aurez le résultat en TP). Si quelques doutes surgissent en TD, les **noter** dans un coin, avec les hypothèses de résolution associées et **prévoir** les expérimentations à faire en TP pour lever ces doutes et observer quelles hypothèses étaient les bonnes.

11.1 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 }
```

Nous le compilons 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` (Cf. Annexe 11.5).

Questions :

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` ?

11.2 Programme avec une procédure qui a beaucoup de paramètres

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 }
```

Le code produit dans le fichier `bcp_param.s` par la commande : `arm-eabi-gcc -O0 -S bcp_param.c` est dans le paragraphe 11.6.

Questions :

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` ?

11.3 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 }
```

Le code produit dans le fichier `var_pile.s` par la commande : `arm-eabi-gcc -O0 -S var_pile.c` est dans le paragraphe 11.7.

Questions :

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` ?

11.4 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 (pour la réponse effective, attendre le TP)

11.5 Annexe : `premier.s`

```
 1         .cpu arm7tdmi
 2         .fpu softvfp
 3         .eabi_attribute 20, 1
 4         .eabi_attribute 21, 1
 5         .eabi_attribute 23, 3
 6         .eabi_attribute 24, 1
 7         .eabi_attribute 25, 1
 8         .eabi_attribute 26, 1
 9         .eabi_attribute 30, 6
10         .eabi_attribute 18, 4
11         .file   "premier.c"
12         .section        .rodata
13         .align  2
14 .LC0:
15         .ascii  "Donner une chaine de longueur inferieure a %d:\012\000"
16         .align  2
17 .LC1:
18         .ascii  "la chaine lue est : %s\012\000"
19         .align  2
20 .LC2:
21         .ascii  "la longueur de la chaine lue est : %d\012\000"
22         .text
23         .align  2
24         .global main
25         .type   main, %function
26 main:
27         @ Function supports interworking.
28         @ args = 0, pretend = 0, frame = 16
29         @ frame_needed = 1, uses_anonymous_args = 0
30         stmfd   sp!, {fp, lr}
31         add     fp, sp, #4
32         sub     sp, sp, #16
33         ldr     r0, .L2
34         mov     r1, #10
35         bl      printf
36         ldr     r3, .L2+4
37         ldr     r3, [r3, #0]
38         ldr     r3, [r3, #4]
39         sub     r2, fp, #20
40         mov     r0, r2
```

```
41          mov     r1, #10
42          mov     r2, r3
43          bl      fgets
44          sub     r3, fp, #20
45          ldr     r0, .L2+8
46          mov     r1, r3
47          bl      printf
48          sub     r3, fp, #20
49          mov     r0, r3
50          bl      strlen
51          mov     r3, r0
52          str     r3, [fp, #-8]
53          ldr     r0, .L2+12
54          ldr     r1, [fp, #-8]
55          bl      printf
56          mov     r0, r3
57          sub     sp, fp, #4
58          ldmfd   sp!, {fp, lr}
59          bx      lr
60  .L3:
61          .align  2
62  .L2:
63          .word   .LC0
64          .word   _impure_ptr
65          .word   .LC1
66          .word   .LC2
67          .size   main, .-main
68          .ident  "GCC: (GNU) 4.5.3"
```

11.6 Annexe : bcp_param.s

```
 1    .cpu arm7tdmi
 2    .fpu softvfp
 3    .eabi_attribute 20, 1
 4    .eabi_attribute 21, 1
 5    .eabi_attribute 23, 3
 6    .eabi_attribute 24, 1
 7    .eabi_attribute 25, 1
 8    .eabi_attribute 26, 1
 9    .eabi_attribute 30, 6
10    .eabi_attribute 18, 4
11    .file "bcp_param.c"
12    .text
13    .align   2
14    .type Somme, %function
15  Somme:
16    @ Function supports interworking.
17    @ args = 24, pretend = 0, frame = 64
18    @ frame_needed = 1, uses_anonymous_args = 0
19    @ link register save eliminated.
20    str   fp, [sp, #-4]!
21    add   fp, sp, #0
22    sub   sp, sp, #68
23    str   r0, [fp, #-56]
24    str   r1, [fp, #-60]
25    str   r2, [fp, #-64]
```

```
26    str   r3, [fp, #-68]
27    ldr   r3, [fp, #-56]
28    add   r3, r3, #1
29    str   r3, [fp, #-8]
30    ldr   r3, [fp, #-60]
31    add   r3, r3, #1
32    str   r3, [fp, #-12]
33    ldr   r3, [fp, #-64]
34    add   r3, r3, #1
35    str   r3, [fp, #-16]
36    ldr   r3, [fp, #-68]
37    add   r3, r3, #1
38    str   r3, [fp, #-20]
39    ldr   r3, [fp, #4]
40    add   r3, r3, #1
41    str   r3, [fp, #-24]
42    ldr   r3, [fp, #8]
43    add   r3, r3, #1
44    str   r3, [fp, #-28]
45    ldr   r3, [fp, #12]
46    add   r3, r3, #1
47    str   r3, [fp, #-32]
48    ldr   r3, [fp, #16]
49    add   r3, r3, #1
50    str   r3, [fp, #-36]
51    ldr   r3, [fp, #20]
52    add   r3, r3, #1
53    str   r3, [fp, #-40]
54    ldr   r3, [fp, #24]
55    add   r3, r3, #1
56    str   r3, [fp, #-44]
57    ldr   r2, [fp, #-8]
58    ldr   r3, [fp, #-12]
59    add   r2, r2, r3
60    ldr   r3, [fp, #-16]
61    add   r2, r2, r3
62    ldr   r3, [fp, #-20]
63    add   r2, r2, r3
64    ldr   r3, [fp, #-24]
65    add   r2, r2, r3
66    ldr   r3, [fp, #-28]
67    add   r2, r2, r3
68    ldr   r3, [fp, #-32]
69    add   r2, r2, r3
70    ldr   r3, [fp, #-36]
71    add   r2, r2, r3
72    ldr   r3, [fp, #-40]
73    add   r2, r2, r3
74    ldr   r3, [fp, #-44]
75    add   r3, r2, r3
76    str   r3, [fp, #-48]
77    ldr   r3, [fp, #-48]
78    mov   r0, r3
79    add   sp, fp, #0
80    ldmfd sp!, {fp}
81    bx lr
```

```
 82    .size Somme, .-Somme
 83    .section .rodata
 84    .align   2
 85 .LC0:
 86    .ascii   "La somme des entiers de 1 a 10 plus 10 vaut %d\012\000"
 87    .text
 88    .align   2
 89    .global  main
 90    .type main, %function
 91 main:
 92    @ Function supports interworking.
 93    @ args = 0, pretend = 0, frame = 8
 94    @ frame_needed = 1, uses_anonymous_args = 0
 95    stmfd sp!, {fp, lr}
 96    add   fp, sp, #4
 97    sub   sp, sp, #32
 98    mov   r3, #5
 99    str   r3, [sp, #0]
100    mov   r3, #6
101    str   r3, [sp, #4]
102    mov   r3, #7
103    str   r3, [sp, #8]
104    mov   r3, #8
105    str   r3, [sp, #12]
106    mov   r3, #9
107    str   r3, [sp, #16]
108    mov   r3, #10
109    str   r3, [sp, #20]
110    mov   r0, #1
111    mov   r1, #2
112    mov   r2, #3
113    mov   r3, #4
114    bl Somme
115    str   r0, [fp, #-8]
116    ldr   r0, .L3
117    ldr   r1, [fp, #-8]
118    bl printf
119    mov   r0, r3
120    sub   sp, fp, #4
121    ldmfd sp!, {fp, lr}
122    bx lr
123 .L4:
124    .align   2
125 .L3:
126    .word .LC0
127    .size main, .-main
128    .ident   "GCC: (GNU) 4.5.3"
```

11.7 Annexe : `var_pile.s`

```
 1         .cpu arm7tdmi
 2         .fpu softvfp
 3         .eabi_attribute 20, 1
 4         .eabi_attribute 21, 1
 5         .eabi_attribute 23, 3
 6         .eabi_attribute 24, 1
```

```
 7         .eabi_attribute 25, 1
 8         .eabi_attribute 26, 1
 9         .eabi_attribute 30, 6
10         .eabi_attribute 18, 4
11         .file   "var_pile.c"
12         .text
13         .align  2
14         .global Compare2Chaines
15         .type   Compare2Chaines, %function
16 Compare2Chaines:
17         @ Function supports interworking.
18         @ args = 0, pretend = 0, frame = 16
19         @ frame_needed = 1, uses_anonymous_args = 0
20         @ link register save eliminated.
21         str     fp, [sp, #-4]!
22         add     fp, sp, #0
23         sub     sp, sp, #20
24         str     r0, [fp, #-16]
25         str     r1, [fp, #-20]
26         ldr     r3, [fp, #-16]
27         str     r3, [fp, #-8]
28         ldr     r3, [fp, #-20]
29         str     r3, [fp, #-12]
30         b       .L2
31 .L4:
32         ldr     r3, [fp, #-8]
33         add     r3, r3, #1
34         str     r3, [fp, #-8]
35         ldr     r3, [fp, #-12]
36         add     r3, r3, #1
37         str     r3, [fp, #-12]
38 .L2:
39         ldr     r3, [fp, #-8]
40         ldrb    r3, [r3, #0]    @ zero_extendqisi2
41         cmp     r3, #0
42         beq     .L3
43         ldr     r3, [fp, #-12]
44         ldrb    r3, [r3, #0]    @ zero_extendqisi2
45         cmp     r3, #0
46         beq     .L3
47         ldr     r3, [fp, #-8]
48         ldrb    r2, [r3, #0]    @ zero_extendqisi2
49         ldr     r3, [fp, #-12]
50         ldrb    r3, [r3, #0]    @ zero_extendqisi2
51         cmp     r2, r3
52         beq     .L4
53 .L3:
54         ldr     r3, [fp, #-8]
55         ldrb    r3, [r3, #0]    @ zero_extendqisi2
56         cmp     r3, #0
57         bne     .L5
58         ldr     r3, [fp, #-12]
59         ldrb    r3, [r3, #0]    @ zero_extendqisi2
60         cmp     r3, #0
61         bne     .L5
62         mov     r3, #1
```

```
 63          b       .L6
 64 .L5:
 65          mov     r3, #0
 66 .L6:
 67          mov     r3, r3, asl #16
 68          mov     r3, r3, lsr #16
 69          mov     r3, r3, asl #16
 70          mov     r3, r3, asr #16
 71          mov     r0, r3
 72          add     sp, fp, #0
 73          ldmfd   sp!, {fp}
 74          bx      lr
 75          .size   Compare2Chaines, .-Compare2Chaines
 76          .section        .rodata
 77          .align  2
 78 .LC0:
 79          .ascii  "Chaine 1, de moins de 99 caracteres : \000"
 80          .align  2
 81 .LC1:
 82          .ascii  "Chaine 2, de moins de 99 caracteres : \000"
 83          .align  2
 84 .LC2:
 85          .ascii  "oui\000"
 86          .align  2
 87 .LC3:
 88          .ascii  "non\000"
 89          .align  2
 90 .LC4:
 91          .ascii  "Sont-elles egales ? %s !\012\000"
 92          .text
 93          .align  2
 94          .global main
 95          .type   main, %function
 96 main:
 97          @ Function supports interworking.
 98          @ args = 0, pretend = 0, frame = 208
 99          @ frame_needed = 1, uses_anonymous_args = 0
100          stmfd   sp!, {fp, lr}
101          add     fp, sp, #4
102          sub     sp, sp, #208
103          ldr     r0, .L10
104          bl      puts
105          ldr     r3, .L10+4
106          ldr     r3, [r3, #0]
107          ldr     r3, [r3, #4]
108          sub     r2, fp, #108
109          mov     r0, r2
110          mov     r1, #100
111          mov     r2, r3
112          bl      fgets
113          ldr     r0, .L10+8
114          bl      puts
115          ldr     r3, .L10+4
116          ldr     r3, [r3, #0]
117          ldr     r3, [r3, #4]
118          sub     r2, fp, #208
```

```
119         mov     r0, r2
120         mov     r1, #100
121         mov     r2, r3
122         bl      fgets
123         sub     r2, fp, #108
124         sub     r3, fp, #208
125         mov     r0, r2
126         mov     r1, r3
127         bl      Compare2Chaines
128         mov     r3, r0
129         strh    r3, [fp, #-6]   @ movhi
130         ldrsh   r3, [fp, #-6]
131         cmp     r3, #0
132         beq     .L8
133         ldr     r3, .L10+12
134         b       .L9
135 .L8:
136         ldr     r3, .L10+16
137 .L9:
138         ldr     r0, .L10+20
139         mov     r1, r3
140         bl      printf
141         mov     r0, r3
142         sub     sp, fp, #4
143         ldmfd   sp!, {fp, lr}
144         bx      lr
145 .L11:
146         .align  2
147 .L10:
148         .word   .LC0
149         .word   _impure_ptr
150         .word   .LC1
151         .word   .LC2
152         .word   .LC3
153         .word   .LC4
154         .size   main, .-main
155         .ident  "GCC: (GNU) 4.5.3"
```