Université Grenoble Alpes (UGA)

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

Unité d'Enseignement INF401 aux Parcours INF et MIN :

Introduction aux Architectures Logicielles et Matérielles

Documentation Technique
Sujets des Travaux Dirigés
Sujets des Travaux Pratiques

Année Universitaire 2016 / 2017

Table des matières

Première partie

Documentation Technique

Chapitre 1

Environnement informatique pour les travaux pratiques

1.1 Connexion au serveur

Un serveur Linux : `turing` est disponible pour les TPs de 401_INF_PUBLIC.

Depuis un poste du DLST : utiliser un logiciel de connexion à distance (typiquement `putty`, `xming` ou `cygwin/X`) disponible sur le pc windows pour obtenir une bannnière de connexion sur turing.

Depuis une machine de type POSIX (unix,linux,macOS) : lancer la commande `ssh -X -C turing.e.ujf-grenoble.fr` dans un terminal (Xterm, Konsole, etc) pour vous connecter.

1.2 Emplacement des fichiers

Les fichiers nécessaires pour effectuer chaque TP sont situés dans le répertoire :
`/Public/401_INF_PUBLIC/TP`$<i>$, ou $<i>$ désigne le numéro du TP.
Par exemple, les fichiers nécessaires pour la première séance sont situés dans le répertoire `/Public/401_INF_PUBLIC/TP1`. Lorsque le TP dure plus d'une séance le nom du répertoire porte les numéros des deux séances, comme par exemple `/Public/401_INF_PUBLIC/TP3et4`.

1.3 Configuration de la session de travail

Les opérations suivantes doivent être effectuées pour configurer *une fois pour toutes* votre environnement de travail :

1. Se connecter à `turing`
2. Exécuter les commandes de configuration contenues dans le fichier `config.sh` au moyen de la commande : `source /Public/401_INF_PUBLIC/config.sh`
   Vérifier que le répertoire `/opt/gnu/arm/bin` est bien en tête du chemin d'accès aux exécutables, au moyen de la commande : `echo $PATH`
   Cette opération installe une fois pour toutes l'environnement requis pour les TPs. Elle n'est à exécuter qu'une seule fois. *Elle n'aura pas à être ré-exécutée lors des autres séances.*

Votre binôme doit ensuite répéter la même opération, afin que vous puissiez tous deux travailler avec un environnement correct dans la suite du semestre :

— Il doit se connecter à son tour à `turing` (depuis un autre poste ou sur le même après que vous vous soyez vous-même déconnecté).

— Il doit ensuite exécuter sous son identité la commande :
`source /Public/401_INF_PUBLIC/config.sh`

1.4 Démarrage d'une session de travail

Les opérations suivantes sont à effectuer **au début de chaque séance** :

1. Se connecter à `turing`.
2. Ouvrir une deuxième fenêtre au moyen de la commande : `xterm &` (Ctrl+clic central ou droit pour options de configuration).
3. Copier le répertoire `/Public/401_INF_PUBLIC/TP`<i> dans votre répertoire de travail. Par exemple, pour le 1^{er} TP utilisez la commande :
`cp -r /Public/401_INF_PUBLIC/TP1 .`
(*ne pas oublier le point à la fin de la commande précédente* !...)
4. Effectuer les exercices décrits dans l'énoncé du TP.

1.5 TP1 : ressources disponibles

1.5.1 Fichiers

— Exo 1.1.1 : `image.bm`
— Exo 1.2.1 : `lignes.xpm`
— Exo 2.x : `prog.c`
— Exo 3 : `prog1.s`, `prog1.var1.s`, `prog1.var2.s`, `prog1.var2.s`

1.5.2 Outils

— `nedit` : création et modification de fichiers au format texte (`gedit` également disponible)..
— `cat` et `less` : visualisation de fichiers texte.
— `bitmap` : affichage d'une image monochrome au format *bitmap*.
— `xli` : visualisation de fichiers contenant une image.
— `hexdump` : visualisation en hexadécimal d'un fichier binaire ou texte.
— `arm-eabi-gcc` : compilateur C et assembleur pour processeur Arm.
— `arm-eabi-objdump` : utilitaire permettant d'observer le contenu d'un fichier binaire ayant été produit par `arm-eabi-gcc`.

1.6 Quelques commandes utiles (rappels)

— Créer une copie d'un fichier existant (sans utiliser `nedit` :
`cp` <*nom fichier original*> <*nom nouveau fichier*>
Exemple : `cp fich1.c fich2.c`
ou bien :
`cp` <*nom fichier original*> <*nom répertoire destination*>
Exemple : `cp fich1.c repA`

— Renommer un fichier :
`mv` <*nom original du fichier*> <*nouveau nom du fichier*>
Exemple : `mv fich1.c fich2.c`

— Déplacer un fichier :
`mv` <*nom du fichier*> <*nom du répertoire destination*>
Exemple : `mv fich1.c ../repB`

— Afficher (sans le modifier) le contenu d'un *gros* fichier (inutile d'utiliser `nedit`) :
`less` <*nom du fichier*> Exemple : `less fich1.c`

— Afficher (sans le modifier) le contenu d'un *petit* fichier (inutile d'utiliser `nedit`) :
`cat` <*nom du fichier*> Exemple : `cat fich2.s`

1.7 Commandes raccourcies

— `armas` <*nom du fichier source .s*>
Assemblage d'un fichier source. Le résultat est un fichier binaire translatable dont le nom est suffixé par `.o` (`prog.o` dans l'exemple).
Exemple : `armas prog.s`

— `armcc` <*nom du fichier source .c*>
Compilation d'un fichier en langage C. Le résultat est un fichier binaire translatable dont le nom est suffixé par `.o` (`prog.o` dans l'exemple).
Exemple : `armcc prog.c`

— `armbuild` <*nom du fichier exécutable*> <*nom du fichier source*> [<*liste de fichiers* `.o` *à ajouter éventuellement*>]
Assemblage (ou compilation, pour un fichier en langage C) *et* production d'un exécutable.
Exemple 1 : `armbuild prog1 prog1.s lib.o`
Exemple 2 : `armbuild prog2 prog2.c`

— `armrun` <*nom du fichier exécutable*>
Exécution/simulation d'un fichier binaire exécutable.
Exemple : `armrun prog`

— `armgdb` <*nom du fichier exécutable*>
Mise au point d'un fichier binaire exécutable.
Exemple : `armgdb prog`

— `armddd` <*nom du fichier exécutable*>
Mise au point d'un fichier binaire exécutable (mode graphique).
Exemple : `armddd prog`

— `armdata` <*nom du fichier "objet" .o*>
Observation de la section .data d'un fichier binaire translatable. Le résultat est affiché à l'écran.
Exemple : `armdata prog.o`

— `armdisas` <*nom du fichier "objet" .o*>
Observation/désassemblage de la section .text d'un fichier binaire translatable. Le résultat est affiché à l'écran.
Exemple : `armdisas prog.o`

Chapitre 2

Langage machine et langage d'assemblage ARM

2.1 Résumé de documentation technique ARM

2.1.1 Organisation des registres

Dans le mode dit "utilisateur" le processeur `ARM` a 16 registres visibles de taille 32 bits nommés `r0, r1, ..., r15` :

- `r13` (synonyme `sp`, comme "stack pointer") est utilisé comme registre pointeur de pile.
- `r14` (synonyme `lr` comme "link register") est utilisé par l'instruction "branch and link" (`bl`) pour sauvegarder l'adresse de retour lors d'un appel de procédure.
- `r15` (synonyme `pc`, comme "program counter") est le registre compteur de programme.

Les conventions de programmation des procédures (ATPCS="ARM-Thumb Procedure Call Standard, Cf. Developer Guide, chapitre 2) précisent :

- les registres `r0, r1, r2 et r3` sont utilisés pour le passage des paramètres (données ou résultats)
- le registre `r12` (synonyme `ip`) est un "intra-procedure call scratch register" ; autrement dit il peut être modifié par une procédure appelée.
- le compilateur `arm-eabi-gcc` utilise le registre `r11` (synonyme `fp` comme "frame pointer") comme base de l'environnement de définition d'une procédure.

Le processeur a de plus un registre d'état, `cpsr` pour "Current Program Status Register", qui comporte entre autres les codes de conditions arithmétiques. Le registre d'état est décrit dans la figure 2.1.

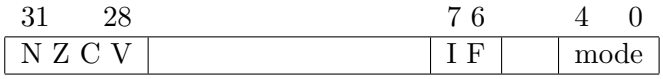

FIGURE 2.1 – Registre d'état du processeur ARM

Les bits `N, Z, C` et `V` sont les codes de conditions arithmétiques, `I` et `F` permettent le masquage des interruptions et `mode` définit le mode d'exécution du processeur (`User`, `Abort`, `Supervisor`, `IRQ`, etc).

2.1.2 Les instructions

Nous utilisons trois types d'instructions : les instructions arithmétiques et logiques (paragraphe 2.1.5), les instructions de rupture de séquence (paragraphe 2.1.6) et les instructions de transfert

d'information entre les registres et la mémoire (paragraphe 2.1.7).

Les instructions sont codées sur 32 bits.

Certaines instructions peuvent modifier les codes de conditions arithmétiques `N, Z, C, V` en ajoutant un `S` au nom de l'instruction.

Toutes les instructions peuvent utiliser les codes de conditions arithmétiques en ajoutant un mnémonique (Cf. figure 2.2) au nom de l'instruction. Au niveau de l'exécution, l'instruction est exécutée si la condition est vraie.

2.1.3 Les codes de conditions arithmétiques

La figure 2.2 décrit l'ensemble des conditions arithmétiques.

code	mnémonique	signification	condition testée
0000	EQ	égal	Z
0001	NE	non égal	$\overline{Z}$
0010	CS/HS	$\geq$ dans `N`	C
0011	CC/LO	$<$ dans `N`	$\overline{C}$
0100	MI	moins	N
0101	PL	plus	$\overline{N}$
0110	VS	débordement	V
0111	VC	pas de débordement	$\overline{V}$
1000	HI	$>$ dans `N`	$C \wedge \overline{Z}$
1001	LS	$\leq$ dans `N`	$\overline{C} \vee Z$
1010	GE	$\geq$ dans `Z`	$(N \wedge V) \vee (\overline{N} \wedge \overline{V})$
1011	LT	$<$ dans `Z`	$(N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
1100	GT	$>$ dans `Z`	$\overline{Z} \wedge ((N \wedge V) \vee (\overline{N} \wedge \overline{V}))$
1101	LE	$\leq$ dans `Z`	$Z \vee (N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
1110	AL	toujours	

FIGURE 2.2 – Codes des conditions arithmétiques

Toute instruction peut être exécutée sous une des conditions décrites dans la figure 2.2. Le code de la condition figure dans les bits 28 à 31 du code de l'instruction. Par défaut, la condition est `AL`.

2.1.4 Description de l'instruction de chargement d'un registre

Nous choisissons dans ce paragraphe de décrire en détail le codage d'une instruction.

L'instruction `MOV` permet de charger un registre avec une valeur immédiate ou de transférer la valeur d'un registre dans un autre avec modification par translation ou rotation de cette valeur.

La syntaxe de l'instruction de transfert est : `MOV [<COND>] [S] <rd>, <opérande>` où `rd` désigne le registre destination et `opérande` est décrit par la table ci-dessous :

`opérande`	commentaire
`#immédiate-8`	entier sur 32 bits (Cf. remarque ci-dessous)
`rm`	registre
`rm, shift #shift-imm-5`	registre dont la valeur est décalée d'un nombre de positions représenté sur 5 bits
`rm, shift rs`	registre dont la valeur est décalée du nombre de positions contenu dans le registre `rs`

Dans la table précédente le champ `shift` de l'opérande peut être `LSL, LSR, ASR, ROR` qui signifient respectivement "logical shift left", "logical shift right", "arithmetic shift right", "rotate right".

Une valeur immédiate est notée selon les mêmes conventions que dans le langage C ; ainsi elle peut être décrite en décimal (15), en hexadécimal (0xF) ou en octal (O17).

Le codage de l'instruction `MOV` est décrit dans les figures 2.3 et 2.4. `<COND>` désigne un mnémonique de condition ; s'il est omis la condition est `AL`. Le bit `S` est mis à 1 si l'on souhaite une mise à jour des codes de conditions arithmétiques. Le bit `I` vaut 1 dans le cas de chargement d'une valeur immédiate. Les codes des opérations `LSL, LSR, ASR, ROR` sont respectivement : $00, 01, 10, 11$.

Remarque concernant les valeurs immédiates : Une valeur immédiate sur 32 bits (opérande `#immediate`) sera codée dans l'instruction au moyen, d'une part d'une constante exprimée sur 8 bits (bits 7 à 0 de l'instruction, figure 2.4, 1^{er} cas), et d'autre part d'une rotation exprimée sur 4 bits (bits 11 à 8) qui sera appliquée à la dite constante lors de l'exécution de l'instruction.

La valeur de rotation, comprise entre 0 et 15, est multipliée par 2 lors de l'exécution et permet donc d'appliquer à la constante une rotation à **droite** d'un nombre pair de positions compris entre 0 et 30. La rotation s'applique aux 8 bits placés initialement à droite dans un mot de 32 bits (qui n'est pas celui qui contient l'instruction).

Il en résulte que ne peuvent être codées dans l'instruction toutes les valeurs immédiates sur 32 bits...

Une rotation nulle permettra de coder toutes les valeurs immédiates sur 8 bits.

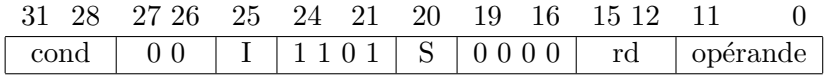

FIGURE 2.3 – Codage de l'instruction `mov`

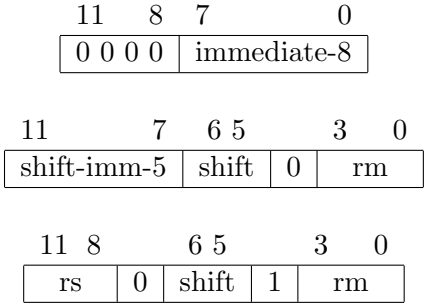

FIGURE 2.4 – Codage de la partie `opérande` d'une instruction

Exemples d'utilisations de l'instruction `mov`

```
MOV r1, #42         @ r1 <-- 42
MOV r3, r5          @ r3 <-- r5
MOV r2, r7, LSL #28 @ r2 <-- r7 décalé à gauche de 28 positions
MOV r1, r0, LSR r2  @ r1 <-- r0 décalé à droite de n pos., r2=n
MOVS r2, #-5        @ r2 <-- -5 + positionnement N, Z, C et V
MOVEQ r1, #42       @ si cond(EQ) alors r1 <-- 42
MOVLTS r3, r5       @ si cond(LT) alors r3 <-- r5 + positionnement N, Z, C et V
```

2.1.5 Description des instructions arithmétiques et logiques

Les instructions arithmétiques et logiques ont pour syntaxe :
`code-op[<cond>][s] <rd>, <rn>, <opérande>`, où `code-op` est le nom de l'opération, `rn` et `opérande` sont les deux opérandes et `rd` le registre destination.

Le codage d'une telle instruction est donné dans la figure 2.5. `opérande` est décrit dans le paragraphe 2.1.4, figure 2.4.

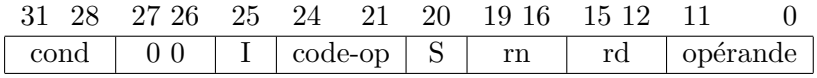

FIGURE 2.5 – Codage d'une instruction arithmétique ou logique

La table ci-dessous donne la liste des intructions arithmétiques et logiques ainsi que les instructions de chargement d'un registre. Les instructions `TST, TEQ, CMP, CMN` n'ont pas de registre destination, elles ont ainsi seulement deux opérandes ; elles provoquent systématiquement la mise à jour des codes de conditions arithmétiques (dans le codage de l'instruction les bits 12 à 15 sont mis à zéro). Les instructions `MOV` et `MVN` ont un registre destination et un opérande (dans le codage de l'instruction les bits 16 à 19 sont mis à zéro).

code-op	Nom	Explication du nom	Opération	remarque
0000	`AND`	AND	et bit à bit	
0001	`EOR`	Exclusive OR	ou exclusif bit à bit	
0010	`SUB`	SUBstract	soustraction	
0011	`RSB`	Reverse SuBstract	soustraction inversée	
0100	`ADD`	ADDition	addition	
0101	`ADC`	ADdition with Carry	addition avec retenue	
0110	`SBC`	SuBstract with Carry	soustraction avec emprunt	
0111	`RSC`	Reverse Substract with Carry	soustraction inversée avec emprunt	
1000	`TST`	TeST	et bit à bit	pas `rd`
1001	`TEQ`	Test EQuivalence	ou exclusif bit à bit	pas `rd`
1010	`CMP`	CoMPare	soustraction	pas `rd`
1011	`CMN`	CoMpare Not	addition	pas `rd`
1100	`ORR`	OR	ou bit à bit	
1101	`MOV`	MOVe	copie	pas `rn`
1110	`BIC`	BIt Clear	et not bit à bit	
1111	`MVN`	MoVe Not	not (complément à 1)	pas `rn`

Exemples d'utilisations

```
ADD r1, r2, r5           @ r1 <-- r2 + r5
ADDS r0, r2, #4          @ r0 <-- r2 + 4 + positionnement NZCV
SUB r3, r7, r0           @ r3 <-- r7 - r0
SUBS r3, r7, r0          @ r3 <-- r7 - r0 + positionnement NZCV
SUBGES r3, r7, r0        @ si cond(GE) r3 <-- r7 - r0 et positionnement NZCV
CMP r1, r2               @ calcul de r1-r2 et positionnement NZCV
TST r3, #1               @ calcul de r3 ET 1 et positionnement NZCV
ANDS r1, r2, #0x0000ff00 @ r1 <-- r2 ET 0x0000ff00 et positionnement NZCV
```

2.1.6 Description des instructions de rupture de séquence

Nous utilisons trois instructions de rupture de séquence : `B[<cond>] <déplacement>`, `BL[<cond>] <déplacement>`, `BLX[<cond>] <registre>`.

a) Instruction `B[<cond>] <déplacement>` L'instruction `B` provoque la modification du compteur de programme si la condition est vraie ; le texte suivant est extrait de la documentation ARM :

```
if ConditionPassed(cond) then
   PC <-- PC + (SignExtend(déplacement) << 2)
```

L'expression `(SignExtend(déplacement) << 2)` signifie que le `déplacement` est tout d'abord étendu de façon signée à 32 bits puis mutiplié par 4. Le `déplacement` est en fait un entier relatif (codé sur 24 bits comme indiqué ci-dessous) et qui représente le nombre d'instructions (en avant ou en arrière) entre l'instruction de rupture de séquence et la cible de cette instruction.

Dans le calcul du déplacement, il faut prendre en compte le fait que lors de l'exécution d'une instruction, le compteur de programme ne repère pas l'instruction courante mais deux instructions en avant.

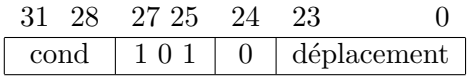

FIGURE 2.6 – Codage de l'instruction de rupture de séquence `b{cond}`

Exemples d'utilisations

```
BEQ +5      @ si cond(EQ) alors pc <-- pc + 4*5
BAL -8      @ pc <-- pc - 4*8
```

Dans la pratique, on utilise une étiquette (Cf. paragraphe 2.2.4) pour désigner l'instruction cible d'un branchement. C'est le traducteur (i.e. l'assembleur) qui effectue le calcul du déplacement.

La figure 2.7 résume l'utilisation des instructions de branchements conditionnels après une comparaison.

Conditions des instructions de branchement conditionnel				
Type	Entiers relatifs ($\mathbb{Z}$)		Naturels ($\mathbb{N}$) et adresses	
Instruction C	Bxx	Condition	Bxx	Condition
goto	BAL	1110	BAL	1110
if (x== y) goto	BEQ	0000	BEQ	0000
if (x != y) goto	BNE	0001	BNE	0001
if (x < y) goto	BLT	1011	BLO, BCC	0011
if (x<= y) goto	BLE	1101	BLS	1001
if (x > y) goto	BGT	1100	BHI	1000
if (x>= y) goto	BGE	1010	BHS,BCS	0010

FIGURE 2.7 – Utilisation des branchements conditionnels après une comparaison

b) Instruction `BL[<cond>] <déplacement>` L'instruction `BL` provoque la modification du compteur de programme avec sauvegarde de l'adresse de l'instruction suivante (appelée `adresse de retour`) dans le registre `lr` ; le texte suivant est extrait de la documentation ARM :

```
lr <-- address of the instruction after the branch instruction
PC <-- PC + (SignExtend(déplacement) << 2)
```

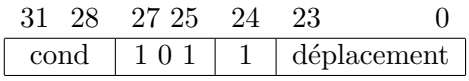

FIGURE 2.8 – Codage de l'instruction de branchement à un sous-programme `bl`

Exemples d'utilisations

```
BL 42       @ lr <-- pc+4 ; pc <-- pc +4*42
```

c) Instruction `BLX[<cond>] <registre>` L'instruction `BLX Rm` provoque la modification du compteur de programme avec sauvegarde de l'adresse de l'instruction suivante (appelée `adresse de retour`) dans le registre `lr` ; le texte suivant est extrait de la documentation ARM :

```
lr <-- address of the instruction after the branch instruction
PC <-- Rm
```

Attention : Dans le cadre des TP, l'adresse passée en paramètre à `BLX` doit être paire. En effet, l'exécution de `BLX` avec une adresse impaire active un mode spécial du processeur (THUMB) avec un autre jeu d'instructions (codées sur 16 bits). Ce type d'erreur peut avoir des effets assez variés en fonction du programme concerné : on peut obtenir un message d'erreur relatif au mode THUMB ou un comportement arbitraire du simulateur.

Exemples d'utilisations

```
BLX R5       @ lr <-- pc+4 ; pc <-- R5
```

Pour désigner une procédure on utilisera une étiquette ; des exemples sont donnés dans le paragraphe 2.2.4.

2.1.7 Description des instructions de transfert d'information entre les registres et la mémoire

Transfert entre un registre et la mémoire

L'instruction `LDR` dont la syntaxe est : `LDR <rd>, <mode-adressage>` permet le transfert du mot mémoire dont l'adresse est spécifiée par `mode-adressage` vers le registre `rd`. Nous ne donnons pas le codage de l'instruction `LDR` parce qu'il comporte un grand nombre de cas ; nous regardons ici uniquement les utilisations les plus fréquentes de cette instruction.

Le champ `mode-adressage` comporte, entre crochets, un registre et éventuellement une valeur immédiate ou un autre registre, ceux-ci pouvant être précédés du signe + ou −. Le tableau ci-dessous indique pour chaque cas le mot mémoire qui est chargé dans le registre destination. L'instruction `ldr` permet beaucoup d'autres types de calcul d'adresse qui ne sont pas décrits ici.

mode-adressage	opération effectuée
`[rn]`	rd <– mem [rn]
`[rn, #offset12]`	rd <– mem [rn + offset12]
`[rn, #-offset12]`	rd <– mem [rn - offset12]
`[rn, rm]`	rd <– mem [rn + rm]
`[rn, -rm]`	rd <– mem [rn - rm]

Il existe des variantes de l'instruction `LDR` permettant d'accéder à un octet : `LDRB` ou à un mot de 16 bits : `LDRH`. Et si l'on veut accéder à un octet signé : `LDRSB` ou à un mot de 16 bits signé : `LDRSH`. Ces variantes imposent cependant des limitations d'adressage par rapport aux versions 32 bits (exemple : valeur immédiate codée sur 5 bits au lieu de 12).

Pour réaliser le transfert inverse, registre vers mémoire, on trouve l'instruction `STR` et ses variantes `STRB` et `STRH`. La syntaxe est la même que celle de l'instruction `LDR`. Par exemple, l'instruction `STR rd, [rn]` provoque l'exécution : `MEM [rn] <-- rd`.

Exemples d'utilisations

```
LDR r1, [r0]           @ r1 <-32bits-- Mem [r0]
LDR r3, [r2, #4]       @ r3 <-32bits-- Mem [r2 + 4]
LDR r3, [r2, #-8]      @ r3 <-32bits-- Mem [r2 - 8]
LDR r3, [pc, #48]      @ r3 <-32bits-- Mem [pc + 48]
LDRB r5, [r3]          @ 8bits_poids_faibles (r5) <-- Mem [r3],
                       @ extension aux 32 bits avec des 0
STRH r2, [r1, r3]      @ Mem [r1 + r3] <-16bits-- 16bits_poids_faibles (r2)
```

L'instruction `LDR` est utilisée entre autres pour accéder à un mot de la zone text en réalisant un adressage relatif au compteur de programme. Ainsi, l'instruction `LDR r2, [pc, #depl]` permet de charger dans le registre `r2` avec le mot mémoire situé à une distance `depl` du compteur de programme, c'est-à-dire de l'instruction en cours d'exécution. Ce mode d'adressage nous permet de recupérer l'adresse d'un mot de données (Cf. paragraphe 2.2.4).

Pré décrémentation et post incrémentation

Les instructions `LDR` et `STR` offrent des adressages post-incrémentés et pré-décrémentés qui permettent d'accéder à un mot de la mémoire et de mettre à jour une adresse, en une seule instruction. Cela revient à combiner un accès mémoire et l'incrémentation du pointeur sur celle-ci en une seule instruction.

instruction ARM	équivalent ARM	équivalent C
`LDR r1, [r2, #-4]!`	`SUB r2, r2, #4`	`r1 = *--r2`
	`LDR r1, [r2]`	
`LDR r1, [r2], #4`	`LDR r1, [r2]`	`r1 = *r2++`
	`ADD r2, r2, #4`	
`STR r1, [r2, #-4]!`	`SUB r2, r2, #4`	
	`STR r1, [r2]`	
`STR r1, [r2], #4`	`STR r1, [r2]`	
	`ADD r2, r2, #4`	

La valeur à incrémenter ou décrémenter (4 dans les exemples ci-dessus) peut aussi être donnée dans un registre.

Transfert multiples

Le processeur ARM possède des instructions de transfert entre un ensemble de registres et un bloc de mémoire repéré par un registre appelé registre de base : `LDM` et `STM`. Par exemple, `STMFD r7!, {r0,r1,r5}` range le contenu des registres `r0, r1` et `r5` dans la mémoire et met à jour le registre `r7` suite le transfert (i.e. `r7` = `r7` - 12) ; après l'exécution de l'instruction `MEM[ r7 à jour ]` contient `r0` et `MEM[ r7 à jour + 8 ]` contient `r5`.

Il existe 8 variantes de chacune des instructions `LDM` et `STM` selon que :

— les adresses de la zone mémoire dans laquelle sont copiés les registres croissent (Increment) ou décroissent (Decrement).

— l'adresse contenue dans le registre de base est incrémentée ou décrémentée avant (Before) ou après (After) le transfert de chaque registre. Notons que l'adresse est décrémentée avant le transfert quand le registre de base repère le mot qui a l'adresse immédiatement supérieure à celle où l'on veut ranger une valeur (Full) ; l'adresse est incrémentée après le transfert quand le registre de base repère le mot où l'on veut ranger une valeur (Empty).

— le registre de base est modifié à la fin de l'exécution quand il est suivi d'un `!` ou laissé inchangé sinon.

Ces instructions servent aussi à gérer une pile. Il existe différentes façons d'implémenter une pile selon que :

— le pointeur de pile repère le dernier mot empilé (`Full`) ou la première place vide (`Empty`).

— le pointeur de pile progresse vers les adresses basses quand on empile une information (`Descending`) ou vers les adresses hautes (`Ascending`).

Par exemple, dans le cas où le pointeur de pile repère l'information en sommet de pile (case pleine) et que la pile évolue vers les adresses basses (lorsque l'on empile l'addresse décroit), on parle de pile `Full Descending` et on utilise l'instruction `STMFD` pour empiler et `LDMFD` pour dépiler.

Les modes de gestion de la pile peuvent être caractérisés par la façon de modifier le pointeur de pile lors de l'empilement d'une valeur ou de la récupération de la valeur au sommet de la pile. Par exemple, dans le cas où le pointeur de pile repère l'information en sommet de pile et que la pile évolue vers les adresses basses, pour empiler une valeur il faut décrémenter le pointeur de pile avant le stockage en mémoire ; on utilisera l'instruction `STMDB` (`Decrement Before`). Dans le même type d'organisation pour dépiler on accède à l'information au sommet de pile puis on incrémente le pointeur de pile : on utilise alors l'instruction `LDMIA` (`Increment After`).

Selon que l'on prend le point de vue gestion d'un bloc de mémoire repéré par un registre ou gestion d'une pile repérée par le registre pointeur de pile, on considère une instruction ou une autre ... Ainsi, les instructions `STMFD` et `STMDB` sont équivalentes ; de même pour les instructions `LDMFD` et `LDMIA`.

Les tables suivantes donnent les noms des différentes variantes des instructions `LDM` et `STM`, chaque variante ayant deux noms synonymes l'un de l'autre.

nom de l'instruction	synonyme
LDMDA (decrement after)	LDMFA (full ascending)
LDMIA (increment after)	LDMFD (full descending)
LDMDB (decrement before)	LDMEA (empty ascending)
LDMIB (increment before)	LDMED (empty descending)

nom de l'instruction	synonyme
STMDA (decrement after)	STMED (empty descending)
STMIA (increment after)	STMEA (empty ascending)
STMDB (decrement before)	STMFD (full descending)
STMIB (increment before)	STMFA (full ascending)

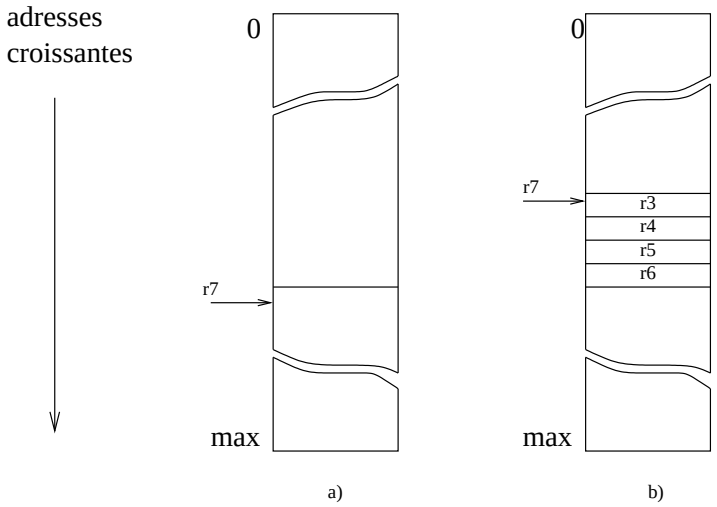

FIGURE 2.9 – Transfert multiples mémoire/registres : `STMFD r7!, {r3,r4,r5,r6}` ou (`STMDB ...`) permet de passer de l'état `a)` de la mémoire à l'état `b)`. `LDMFD r7!, {r3,r4,r5,r6}` (ou `LDMIA ...`) réalise l'inverse.

La figure 2.9 donne un exemple d'utilisation.

2.2 Langage d'assemblage

2.2.1 Structure d'un programme en langage d'assemblage

Un programme est composé de trois types de sections :
— données intialisées ou non (`.data`)
— données non initialisées (`.bss`)
— instructions (`.text`)

Les sections de données sont optionnelles, celle des instructions est obligatoire. On peut écrire des commentaires entre le symbole `@` et la fin de la ligne courante. Ainsi un programme standard a la structure :

```
   .data
@ déclaration de données
@ ...

   .text
@ des instructions
@ ...
```

2.2.2 Déclaration de données

Le langage permet de déclarer des valeurs entières en décimal (éventuellement précédées de leur signe) ou en hexadécimal ; on précise la taille souhaitée.

Exemple :

```
    .data
  .word 4536   @ déclaration de la valeur 4536 sur 32 bits (1 mot)
```

```
  .hword -24   @ déclaration de la valeur -24 sur 16 bits (1 demi mot)
  .byte 5      @ déclaration de la valeur 5 sur 8 bits (1 octet)
  .word 0xfff2a35f @ déclaration d'une valeur en hexadécimal sur 32 bits
  .byte 0xa5   @ idem sur 8 bits
```

On peut aussi déclarer des chaînes de caractères suivies ou non du caractère de code ASCII 00. Un caractère est codé par son code ASCII (Cf. paragraphe 1).

Exemple :

```
    .data
  .ascii "un texte" @ déclaration de 8 caractères...
  .asciz "un texte" @ déclaration de 9 caractères, les mêmes que ci-dessus
                    @ plus le code 0 à la fin
```

La définition de données doit respecter les règles suivantes, qui proviennent de l'organisation physique de la mémoire :

— un mot de 32 bits doit être rangé à une adresse multiple de 4
— un mot de 16 bits doit être rangé à une adresse multiple de 2
— il n'y a pas de contrainte pour ranger un octet (mot de 8 bits)

Pour recadrer une adresse en zone `data` le langage d'assemblage met à notre disposition la directive `.balign`.

Exemple :

```
   .data
@ on note AD l'adresse de chargement de la zone data
@ que l'on suppose multiple de 4 (c'est le cas avec les outils utilisés)
  .hword 43    @ après cette déclaration la prochaine adresse est AD+2
  .balign 4    @ recadrage sur une adresse multiple de 4
  .word 0xffff1234  @ rangé à l'adresse AD+4
  .byte 3      @ après cette déclaration la prochaine adresse est AD+9
  .balign 2    @ recadrage sur une adresse multiple de 2
  .hword 42    @ rangé à l'adresse AD+10
```

On peut aussi réserver de la place en zone `.data` ou en zone `.bss` avec la directive `.skip`. `.skip 256` réserve 256 octets qui ne sont pas initialisés lors de la réservation. On pourra par programme écrire dans cette zone de mémoire.

2.2.3 La zone text

Le programmeur y écrit des instructions qui seront codées par l'assembleur (le traducteur) selon les conventions décrites dans le paragraphe 2.1.

La liaison avec le système (chargement et lancement du programme) est réalisée par la définition d'une étiquette (Cf. paragraphe suivant) réservée : `main`.

Ainsi la zone `text` est :

```
  .text
  .global main
main:

@ des instructions ARM
@ ...
```

2.2.4 Utilisation d'étiquettes

Une donnée déclarée en zone `data` ou `bss` ou une instruction de la zone `text` peut être précédée d'une étiquette. Une étiquette représente une adresse et permet de désigner la donnée ou l'instruction concernée.

Les étiquettes représentent une facilité d'écriture des programmes en langage d'assemblage.

Expression d'une rupture de séquence

On utilise une étiquette pour désigner l'instruction cible d'un branchement. C'est le traducteur (i.e. l'assembleur) qui effectue le calcul du déplacement. Par exemple :

```
etiq: MOV r0, #22
      ADDS r1, r2, r0
      BEQ etiq
```

Accès à une donnée depuis la zone `text`

```
   .data

DD: .word 5

   .text

@ acces au mot d'adresse DD
   LDR r1, relais  @ r1 <-- l'adresse DD
   LDR r2, [r1]    @ r2 <-- Mem[DD] c'est-à-dire 5

   MOV r3, #245    @ r3 <-- 245
   STR r3, [r1]    @ Mem[DD] <-- r3
                   @ la mémoire d'adresse DD a été modifiée

@ plus loin
relais: .word DD  @ déclaration de l'adresse DD en zone text
```

L'instruction `LDR r1, relais` est codée avec un adressage relatif au compteur de programme : `LDR r1, [pc, #depl]` (Cf. paragraphe 2.1.7).

Appel d'une procédure

On utilise l'instruction `BL` lorsque la procédure appelée est désignée directement par une étiquette :

```
   ...
ma_proc: @ corps de la procedure
         mov pc, lr
    ...
    @ appel de la procedure ma_proc
    BL ma_proc
```

On utilise l'instruction `BLX` lorsque l'adresse de la procédure est rangée dans un registre, par exemple lorsqu'une procédure est passée en paramètre :

```
    LDR   r1, adr_proc    @ r1 <-- adresse ma_proc
    BLX   r1
    ...
adr_proc: .word ma_proc
```

2.3 Organisation de la mémoire : petits bouts, gros bouts

La mémoire du processeur ARM peut être vue comme un tableau d'octets repérés par des numéros appelés **adresse** qui sont des entiers naturels sur 32 bits. On peut ranger dans la mémoire des mots de 32 bits, de 16 bits ou des octets (mots de 8 bits). Le paragraphe 2.2.2 indique comment déclarer de tels mots.

Dans la mémoire les mots de 32 bits sont rangés à des adresses multiples de 4. Il y a deux conventions de rangement de mots en mémoire selon l'ordre des octets de ce mot.

Considérons par exemple le mot `0x12345678`.

— convention dite "Big endian" (Gros bouts) :
les 4 octets `12, 34, 56, 78` du mot `0x12345678` sont rangés aux adresses respectives 4x, 4x+1, 4x+2, 4x+3.

— convention dite "Little endian" (Petits Bouts) :
les 4 octets `12, 34, 56, 78` du mot `0x12345678` sont rangés aux adresses respectives 4x+3, 4x+2, 4x+1, 4x.

Le processeur ARM suit la convention "Little endian". La conséquence est que lorsqu'on lit le mot de 32 bits rangé à l'adresse `4x` on voit : `78563412`, c'est-à-dire qu'il faut lire "à l'envers". Selon les outils utilisés le mot de 32 bits est présenté sous cette forme ou sous sa forme externe, plus agréable...

En général les outils de traduction et de simulation permettent de travailler avec une des deux conventions moyennant l'utilisation d'options particulières lors de l'appel des outils (option `-mbig-endian`).

2.4 Commandes de traduction, exécution, observation

2.4.1 Traduction d'un programme

Pour traduire un programme écrit en C contenu dans un fichier `prog.c` :
`arm-eabi-gcc -g -o prog prog.c`. L'option `-o` permet de préciser le nom du programme exécutable ; `o` signifie "output". L'option `-g` permet d'avoir les informations nécessaires à la mise au point sous débogueur (Cf. paragraphe 2.4.2).

Pour traduire un programme écrit en langage d'assemblage ARM contenu dans un fichier `prog.s` :
`arm-eabi-gcc -Wa,--gdwarf2 -o prog prog.s`.

Lorsque l'on veut traduire un programme qui est contenu dans plusieurs fichiers que l'on devra rassembler (on dit "lier"), il faut d'abord produire des versions partielles qui ont pour suffixe `.o`, le `o` voulant dire ici "objet". Par exemple, on a deux fichiers : principal.s et biblio.s, le premier contenant l'étiquette `main`. On effectuera la suite de commandes :

```
arm-eabi-gcc -c -Wa,--gdwarf2 biblio.s
arm-eabi-gcc -c -Wa,--gdwarf2 principal.s
arm-eabi-gcc -g -o prog principal.o biblio.o
```

La première produit le fichier `biblio.o`, la seconde produit le fichier `principal.o`, la troisième les relie et produit le fichier exécutable `prog`.

Noter que les deux commandes suivantes ont le même effet :
`arm-eabi-gcc -c prog.s` et

`arm-eabi-as -o prog.o prog.s`. Elles produisent toutes deux un fichier objet `prog.o` sans les informations nécessaires à l'exécution sous débogueur.

2.4.2 Exécution d'un programme

Exécution directe

On peut exécuter un programme directement avec :
`arm-eabi-run prog`. S'il n'y a pas d'entrées-sorties, on ne voit évidemment rien...

Exécution avec un débogueur

Nous pouvons utiliser deux versions du même débogueur : `gdb` et `ddd`. On parle aussi de metteur au point. C'est un outil qui permet d'exécuter un programme instruction par instruction en regardant les "tripes" du processeur au cours de l'exécution : valeur contenues dans les registres, dans le mot d'état, contenu de la mémoire, etc.

`gdb` est la version de base (textuelle), `ddd` est la même mais graphique (avec des fenêtres, des icônes, etc.), elle est plus conviviale mais plus sujette à des problèmes techniques liés à l'installation du logiciel...

Soit le programme objet exécutable : `prog`. Pour lancer `gdb` :
`arm-eabi-gdb prog`. Puis taper successivement les commandes : `target sim` et enfin `load`. Maintenant on peut commencer la simulation.

Pour éviter de taper à chaque fois les deux commandes précédentes, vous pouvez créer un fichier de nom `.gdbinit` dont le contenu est :

```
# un diese débute un commentaire
# commandes de demarrage pour arm-eabi-gdb
target sim
load
```

Au lancement de `arm-eabi-gdb prog`, le contenu de ce fichier sera automatiquement exécuté.
Voilà un ensemble de commandes utiles :

- placer un point d'arrêt sur une instruction précédée d'une étiquette, par exemple : `break main`. On peut aussi demander `break no` avec `no` un numéro de ligne dans le code source. Un raccourci pour la commande est `b`.
- enlever un point d'arrêt : `delete break numéro_du_point_d'arrêt`
- voir le code source : `list`
- lancer l'exécution : `run`
- poursuivre l'exécution après un arrêt : `cont`, raccourci : `c`
- exécuter l'instruction à la ligne suivante, en entrant dans les procédures : `step`, raccourci `s`
- exécuter l'instruction suivante (sans entrer dans les procédures) : `next`, raccourci `n`
- voir la valeur contenue dans les registres : `info reg`
- voir la valeur contenue dans le registre `r1` : `info reg $r1`
- voir le contenu de la mémoire à l'adresse `etiquette` : `x &etiquette`
- voir le contenu de la mémoire à l'adresse `0x3ff5008` : `x 0x3ff5008`
- voir le contenu de la mémoire en précisant le nombre de mots et leur taille.
  `x /nw adr` permet d'afficher `n` mots de 32 bits à partir de l'adresse `adr`.
  `x /ph adr` permet d'afficher `p` mots de 16 bits à partir de l'adresse `adr`.
- modifier le contenu du registre `r3` avec la valeur `0x44` exprimée en hexadécimal : `set $r3=0x44`
- modifier le contenu de la mémoire d'adresse `etiquette` : `set *etiquette = 0x44`
- sortir : `quit`
- La touche `Enter` répète la dernière commande.

Et ne pas oublier : `man gdb` sous Unix (ou Linux) et quand on est sous `gdb` : `help nom_de_commande...`

Pour lancer `ddd` : `ddd --debugger arm-eabi-gdb`. On obtient une grande fenêtre avec une partie dite "source" (en haut) et une partie dite "console" (en bas). Dans la fenêtre "console" taper successivement les commandes : `file prog`, `target sim` et enfin `load`.

On voit apparaître le source du programme en langage d'assemblage dans la fenêtre "source" et une petite fenêtre de "commandes". Maintenant on peut commencer la simulation.

Toutes les commandes de `gdb` sont utilisables soit en les tapant dans la fenêtre "console", soit en les sélectionnant dans le menu adéquat. On donne ci-dessous la description de quelques menus. Pour le reste, il suffit d'essayer.

- placer un point d'arrêt : sélectionner la ligne en question avec la souris et cliquer sur l'icône break (dans le panneau supérieur).
- démarrer l'exécution : cliquer sur le bouton Run de la fenêtre "commandes". Vous voyez apparaître une flèche verte qui vous indique la position du compteur de programme i.e. où en est le processeur de l'exécution de votre programme.
- le bouton Step permet l'exécution d'une ligne de code, le bouton Next aussi mais en entrant dans les procédures et le bouton Cont permet de poursuivre l'exécution.
- enlever un point d'arrêt : se positionner sur la ligne désirée et cliquer à nouveau sur l'icône break.
- voir le contenu des registres : sélectionner dans le menu Status : Registers ; une fenêtre apparaît. La valeur contenue dans chaque registre est donnée en hexadécimal (0x...) et en décimal.
- observer le contenu de la memoire étiquettée `etiquette` : apres avoir sélectionné memory dans le menu Data, on peut soit donner l'adresse en hexadecimal 0x... si on la connaît, soit donner directement le nom `etiquette` dans la case `from` en le précédant du caractère &, c'est-à-dire `&etiquette`.

2.4.3 Observation du code produit

Considérons un programme objet : `prog.o` obtenu par traduction d'un programme écrit en langage C ou en langage d'assemblage. L'objet de ce paragraphe est de décrire l'utilisation d'un ensemble d'outils permettant d'observer le contenu du fichier `prog.o`. Ce fichier contient les informations du programme source codées et organisées selon un format appelé `format ELF`.

On utilise trois outils : `hexdump`, `arm-eabi-readelf`, `arm-eabi-objdump`.

`hexdump` donne le contenu du fichier dans une forme brute.
`hexdump prog.o` donne ce contenu en hexadécimal complété par le caractère correspondant quand une valeur correspond à un code ascii ; de plus l'outil indique les adresses des informations contenues dans le fichier en hexadécimal aussi.

`arm-eabi-objdump` permet d'avoir le contenu des zones `data` et `text` avec les commandes respectives :
`arm-eabi-objdump -j .data -s prog.o` et
`arm-eabi-objdump -j .text -s prog.o`. Ce contenu est donné en hexadécimal. On peut obtenir la zone `text` avec le désassemblage de chaque instruction :
`arm-eabi-objdump -j .text -d prog.o`.

`arm-eabi-readelf` permet d'avoir le contenu du reste du fichier.
`arm-eabi-readelf -a prog.o` donne l'ensemble des sections contenues dans le fichier sauf les zones `data` et `text`.
`arm-eabi-readelf -s prog.o` donne le contenu de la table des symboles.

Deuxième 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.

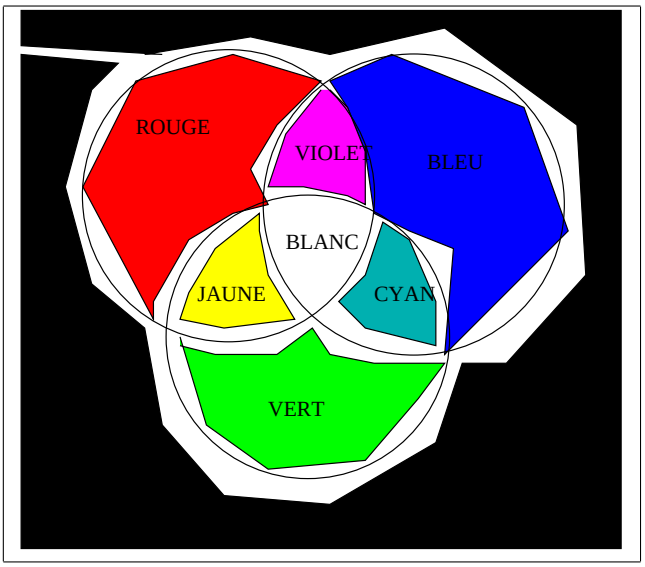

FIGURE 1.1 – Codage de couleurs

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 Ecriture des entiers naturels en base 2

2.1.1 Introduction

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. Tous les calculs sont alors réalisés modulo 2^n (environ quatre milliards pour $n = 32$ bits).

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.

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

2.1.2 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

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

2.1.3 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 (on dit habituellement *complément à deux*) de E l'entier naturel E 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 [1] 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

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).

	1	1	0	1		1		1		0		1
$+_2$	1	0	0	1		1		0		0		1
C = 1	0	0	1	1	C = 1	0	← 0	0	← 0	1	← 1	1
	0	1	1	1	2 ≥ 2	0	1 < 2	1	1 < 2	1	3 ≥ 2	1
	0	1	0	0		0		1		0		0
$+_2$	1	0	1	0		1		0		1		0
C = 0	0	0	0	1	C = 0	0	← 0	0	← 0	0	← 0	1
	1	1	1	1	1 < 2	1	1 < 2	1	1 < 2	1	1 < 2	1

2.3 Conventions d'interprétation

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

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

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 [2], 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.

2.3.2 Pour entiers relatifs (Z) : $\alpha = -1$ 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* [3]

Dans les langages, cette convention d'interprétation est généralement utilisée par défaut (type *int* en langage C).

Un entier 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.

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.

2.3.3 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	$+9 \times 10^{18}(+4E_b - 1)$

2. Les constantes adresse et les variables pointeurs entrent dans cette actégorie.

3. La convention alternative "signe (codé dans le bit de poids fort) et valeur absolue (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 les nombres à virgule flottante.

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 Soutraction

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		

2.5 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 2.4 (soustraction normale) et 2.2 (soustraction par addition du complément à deux).

2.6 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.

2.6.1 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.

2.6.2 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.

2.6.3 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 [4].

2.6.4 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 [5]) 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.

4. 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.

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

	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).

2.6.5 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).

2.7 Exercices

2.7.1 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...

2.7.2 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 représentaion sur 8 bits.

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

2.7.3 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...

2.7.4 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 `N`. Faire le lien avec la réponse que vous avez donnée précédemment.

2.7.5 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 $Z \vee ((N \wedge \overline{V}) \vee (\overline{N} \wedge V))$.

2.7.6 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

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`

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

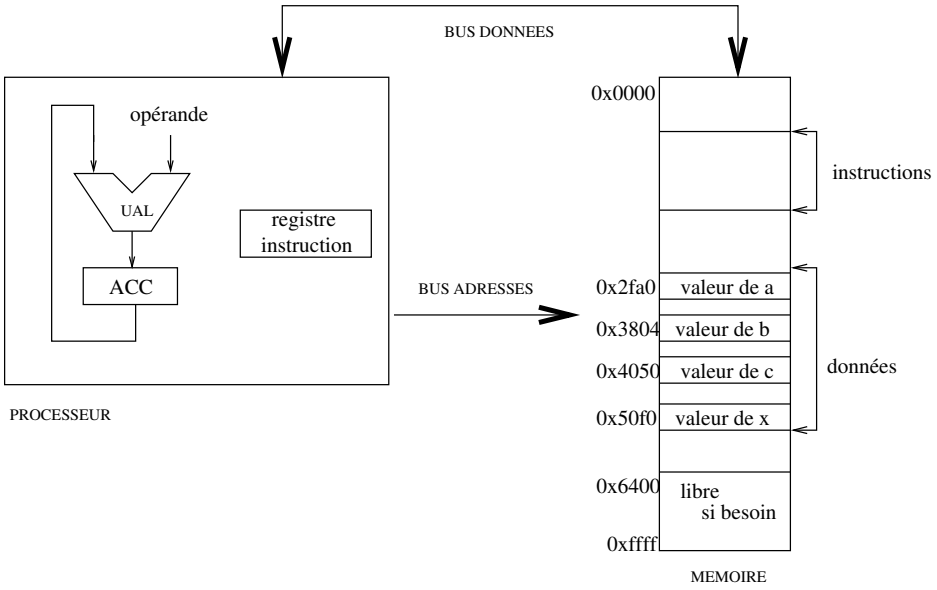

FIGURE 3.1 – Structure d'une machine à accumulateur

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 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.

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 :

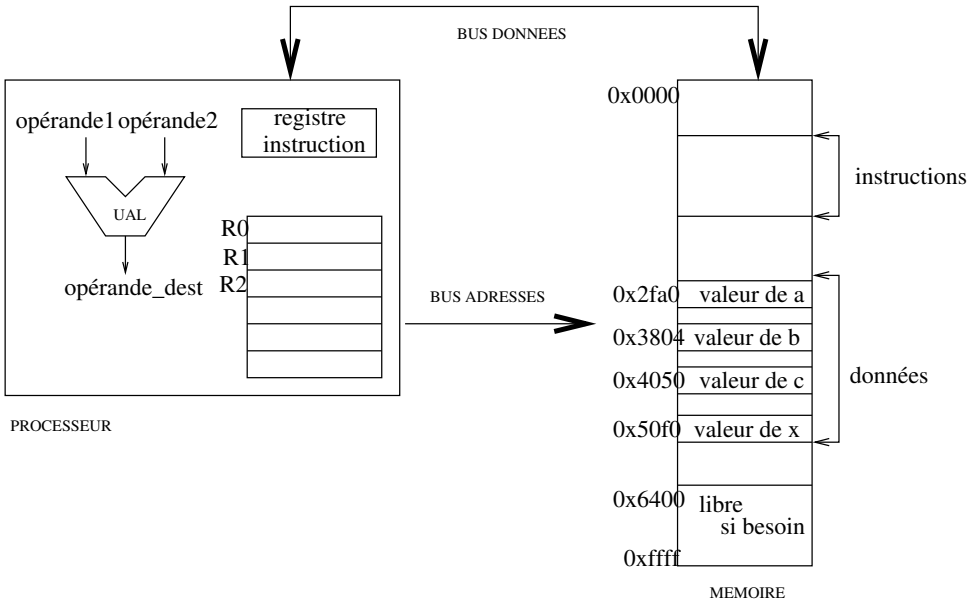

FIGURE 3.2 – Structure d'une machine générale à registres

— 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]`.

Chapitre 4

TD séance 4 : Langage machine (suite)

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, elles sont fixées par les outils de traduction et/ou de chargement en mémoire et nous on peut utiliser des étiquettes... c'est plus agréable à lire.

Le programme `ARM` :

```
@ programme calculant x <-- (a + b + c) + - (x - a - 214)

   .text
   .global main
main: ldr r1, ptr_a
      ldr r1, [r1]
      ldr r2, ptr_b
      ldr r2, [r2]
      ldr r3, ptr_c
      ldr r3, [r3]
      add r4, r1, r2
      add r4, r4, r3
      ldr r2, ptr_x
      ldr r3, [r2]
      sub r3, r3, r1
      sub r3, r3, #214
      sub r4, r4, r3
      str r4, [r2]
      mov pc, lr
      .org 0x1000
ptr_a: .word a
ptr_b: .word b
ptr_c: .word c
ptr_x: .word x

      .data
      .org 0x2fa0
a:    .word 10
      .org 0x3804
b:    .word 20
      .org 0x4050
c:    .word 30
      .org 0x50f0
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 <ptr_a>
  800004:   e5911000    ldr   r1, [r1]
  800008:   e59f2ff4    ldr   r2, [pc, #4084]   ; 801004 <ptr_b>
  80000c:   e5922000    ldr   r2, [r2]
  800010:   e59f3ff0    ldr   r3, [pc, #4080]   ; 801008 <ptr_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 <ptr_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    mov   pc, lr
   ...

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

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

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

0080100c <ptr_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 `ptr_a`, `ptr_b`, `ptr_c` et `ptr_x`.
2. Retrouvez les valeurs rangées à ces adresses dans la zone `data`.
3. Quelle est la traduction de l'instruction `ldr r1, ptr_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, ptr_a` ?
6. Recommencer le même travail avec l'instruction `ldr r2, ptr_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`

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 `Bcc` qui peut être conditionnée par les codes de conditions arithmétiques `cc`.

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

```
@ a dans r0, b dans r1
       CMP r0, r1 @ a-b ??
       BNE sinon
alors: @ a=b : c <-- a-b
       BAL finsi
sinon: @ a!=b : c <-- a+b
finsi:
```

```
       CMP r0, r1 @ a-b ??
       BEQ alors
sinon: @ a!=b : c <-- a+b
       BAL finsi
alors: @ a=b : c <-- a-b
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 r0, r1
          BNE sinon
          SUB r2, r0, r1
   sinon: ADD r2, r0, r1
   ```

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 `r5`.

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 :

```
   .data
debutTAB: .skip 5*4

   .text
   .global main
main:
   ldr r0, ptr_debutTAB
   mov r1, #11
   str r1, [r0]

   mov r1, #22
   add r0, r0, #4  @ *
   str r1, [r0]    @ *

   @ a completer

fin: bal fin

ptr_debutTAB : .word debutTAB
```

A la place des lignes marquées (*) on peut écrire une des deux solutions suivantes :
— `str r1, [r0, #4]` ; le registre `r0` n'est alors pas modifié.
— ou `mov r2, #4` puis `str r1, [r0, r2]` ; le registre `r0` n'est pas modifié.

Exercices : Compléter ce programme de façon à réaliser la dernière affectation. 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 : `r3`, `r2` et `r0`.
   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 Appel de fonction ou procédure en ARM

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. Le retour se fait alors par l'instruction `mov pc, lr`.

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

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

6.2 Codage d'une fonction avec paramètres passés par les registres

On considère une fonction qui retourne un code pour un caractère donné.

```
fonction code (c: caractère, n: entier naturel) --> caractère
{ code(c, n) est le caractère obtenu par une translation de n caractères
à partir de c en considérant l'ordre alphabétique usuel.
Par exemple code('a', 3) est le caractère 'd'.
préconditions : c est une lettre minuscule, 0 <= n <= 26 }
```

On donne ci-dessous un algorithme pour la fonction `code`. On donne aussi la version correspondante en langage `C` et en langage Ada.

Pseudocode :

```
code (ascii_c, n) :
  si ascii_c + n <= 122 alors acsii_c + n sinon acsii_c + n - 26
```

Lors de l'appel, le (code ASCII du) caractère est étendu au format 32 bits et déposé dans un registre. Le (code ASCII du) caractère résultat occupe l'octet de poids faible du registre contenant la valeur retournée par la fonction. Ceci correspond aux conversions de types explicites du code ADA (et implicites en C).

Langage C :

```
char code (char ascii_c, unsigned int n) {
int res;                          /* utiliser de preference r5 pour stocker res */
   res = ascii_c + n;             /* conversion implicite de ascii_c en int     */
   if (res > 'z') res = res - 26;
return res;                       /* conversion implicite de res en char       */
}
```

Langage Ada :

```
function code (ascii_c: in Character; n: in Natural) return Character is
res Character;
begin
   res := Character'Pos (ascii_n) + n;
   if (res > Character'Pos ('z'))      -- Character'Pos ('z') : Ascii(z) = 122
      then   res := res - 26;
   endif
return Character'Val (res);
end code;
```

Exercice 1 : On convient que le code ascii du caractère `c` est dans le registre `r0`, que l'entier `n` est dans le registre `r1` et que le résultat de la fonction est dans le registre `r2`. Traduire en langage d'assemblage ARM la fonction `code`.

On considère le programme suivant :

```
cc1 : caractère; rr1 : caractère; /* char cc1,rr1; */
   LireCar(cc1);                       /* LireCar (&cc1); ou scanf ("%c",&cc1); */
   rr1 = code (cc1, 3);                /* rr1 = code (cc1,3);  */
   EcrCar (rr1);                       /* EcrCar(rr1); ou printf ("%c",rr1); */
```

Exercice 2 : On donne ci-dessous un squelette de traduction en langage d'assemblage ARM de ce programme, avec en particulier les parties de code correspondants à `Lire(cc1)` et `Ecrire(rr1)`. Compléter le programme en langage d'assemblage ARM .

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.

```
    .data
cc1: .byte 0
rr1: .byte 0

    .text
main:
    @ Lire(cc1)
    ldr r1, ptr_cc1
    bl Lirecar
    @ le caractère lu est dans la zone data à l'adresse cc1
       @  ...... appel de code (cc1, 3) ......
```

```
 @ nA COMPLETER
    @ on range le resultat dans rr1
       @ A COMPLETER
    @ EcrCar(rr1) : le caractère à écrire doit être dans r1
       @ A COMPLETER
    bl EcrCar
    ......
ptr_cc1: .word cc1
ptr_rr1: .word rr1
```

6.3 Un autre exemple : factorielle

6.3.1 Utilisation d'une fonction de calcul du produit de deux entiers

On considère le programme suivant qui calcule la factorielle d'un entier en utilisant un algorithme itératif.

```
res, x, n : entiers @ n est l'entier dont on veut calculer la factorielle
   Lire (n)
   res = 1
   x = n
   tantque x != 1
       res = res * x
       x = x - 1
   Ecrire (res)
```

Exercice : donner une traduction de ce programme en langage d'assemblage ARM. Les variables res, x et n seront rangées respectivement dans les registres : `r5, r6, r7` (voir le squelette de programme ci-dessous). On utilise une fonction `mul` qui calcule le produit de deux entiers. Vous en trouverez une réalisation au paragraphe 7.3 mais vous n'avez pas besoin d'en comprendre le fonctionnement pour faire l'exercice. La spécification est la suivante :

```
@ fonction mul (a,b : entiers) --> un entier
@ calcule le produit des deux entiers donnés a et b
@ paramètres données : a<-->r0  b<-->r1
@ résultat dans r2

    .data
n:   .skip 4
res:  .skip 4

    .text

main:
    @ Lire(n)
    ldr r1, ptr_n
    bl Lire32
    @ l'entier lu est dans la zone data à l'adresse n
 .......
 @ Ecrire(res), l'entier à écrire est dans r1
    ldr r12, ptr_res
    ldr r1, [r12]
```

```
    bl EcrNdecimal32
 ...........

ptr_n: .word n
ptr_res: .word res
```

6.3.2 Une fonction en appelle une autre

On reprend l'exercice précédent en définissant une fonction qui calcule la factorielle, cette fonction étant appelée dans le programme principal.

```
fonction fact0 (n : entier) --> entier {
int res, x
  res = 1; x = n
  tantque (x != 1)
     res = res * x;
     x = x - 1;
  retour res;
}

n, f0: entier
   Lire (n)
   f0 = fact0 (n)
   Ecrire (f0);
```

Exercice 1 : transformer le programme précédent pour donner un code ARM de la fonction fact0 et écrire le programme principal ci-dessus en rangeant les variables n et f0 dans la zone data.

On convient des conventions suivantes pour la fonction `fact0` : le paramètre `n` est passé dans le registre `r0`, le résultat de la fonction est rangé dans le registre `r1`.

Exercice 2 : imaginez l'exécution : le programme appelle la fonction `fact0` qui appelle la fonction mul. Quelles sont les adresses que l'on trouve successivement dans le registre `lr`. Conclure...

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**
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**
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**

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" (Cf. figure 7.1).

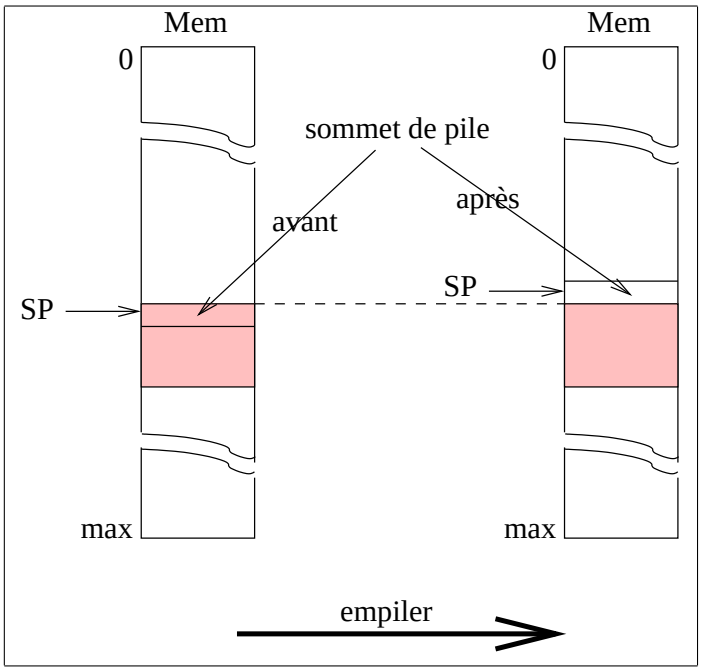

FIGURE 7.1 – Mise en oeuvre de la pile. La pile progresse vers les adresses **décroissantes**, le pointeur de pile repère la **dernière** information empilée

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 permettant d'empiler le contenu d'un octet du registre `r0`. Dans la suite du TD on ecrira : `empiler r0`.
- Ecrire en ARM les deux instructions permettant de depiler le sommet de pile dans le registre `r0`. Dans la suite du TD on ecrira : `depiler r0`.
- Dessiner l'état de la mémoire après chacune des étapes du programme suivant : `mov r0,# 7; empiler r0; mov r0, # 2; empiler r0; mov r0, # 5; empiler r0; mov r0, # 47; depiler r0; depiler r0; mov r0, # 9; empiler 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        60  B1= empiler r0          80  C1= mov r0, # 47
14  A2= empiler r0         68  B2= add r0, r0, # 1     84  bl 60 (B)
1c  bl 60 (B)              6c  B3= depiler r0          88  C2= empiler r0
20  A3= mov r5, #28        74  mov pc, lr              90  bl si condX 80 (C)
24  bl 80 (C)                                          94  C3= mov r2, r5
28  A4= depiler r0                                     98  C4= depiler r0
                                                       a0  mov pc, 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	m[30f0]	m[30ec]	m[30e8]	m[30e4]	m[30e0]
?	?	30f0	?	?	?	?	?	?
10	`mov r0, # 0`	30f0	0	?	?	?	?	?
14	`empiler r0`	30ec	0	?	0	?	?	?

Chapitre 8

TD séances 11 : Paramètres dans la pile, paramètres passés par l'adresse

8.1 Gestion des paramètres et des variables dans la pile

Reprendre la fonction `code` programmée au paragraphe 6.2 et les fonctions `fact0` et `mul` du paragraphe 6.3.2.

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.

8.2 Paramètre passé par adresse

8.2.1 Un premier exemple

On transforme la fonction `code` du paragraphe 6.2 en procédure avec un paramètre résultat.

```
procedure coder (c : in caractère, n: in entier, cres: out caractère)
{ après l'appel coder(c, n, cres), cres est le caractère c translaté de n
positions }
```

On donne ci-dessous un algorithme pour la procédure `coder`. On donne aussi la version correspondante avec un passage de paramètre par adresse.

```
coder (ascii_c, n, cres) :
  si ascii_c + n <= 122 alors cres <-- acsii_c + n
  sinon cres <-- acsii_c + n - 26

procedure coder (données ascii_c: caractère, entier : entier naturel,
                 adresse cres: caractère) {
   si (ascii_c + n <= 122)
   alors mem[cres] = ascii_c + n;
   sinon mem[cres]  = ascii_c + n - 26;
}
```

```
/* La même procédure écrite en C */
void  coder (char ascii_c, unsigned int n, char *cres) {
  unsigned int res;                        /* utiliser de preference r5 pour stocker res */
  res = ascii_c + n;
  if (res > 'z') res = res - 26;
  *cres = res;
}
```

Exercice : On convient que le paramètre `ascii_c` est dans le registre `r0`, que l'entier `n` est dans le registre `r1` et que l'adresse `cres` est dans le registre `r2`. Traduire en langage d'assemblage ARM la procédure `coder`.

Exercice : On considère le programme suivant :

```
cc, rr: entier                          /* char cc, rr; */
   LireCar (cc)                         /* LireCar(&cc); ou scanf ("%c",&cc); */
   coder (cc, 3, adresse de rr)         /* coder (cc,&rr);      */
   EcrCar (rr)                          /* EcrCar(rr); ou printf ("%u",rr); */
```

Traduire ce programme en langage d'assemblage ARM. Les variables `cc` et `rr` sont dans la zone `data` ou `bss`.

Exercice : reprendre la procédure et le programme précédent en passant les paramètres `ascii_c`, `n` et `cres` dans la pile. On pourra si nécessaire écrire une première version en passant les paramètres dans la zone `data` ou `bss`.

8.2.2 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, &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 9

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

9.1 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-aebi-gcc -O0 -S premier.c`. Le code en langage d'assemblage est produit dans le fichier `premier.s` (Cf. Annexe 9.4).

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` ?

9.2 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-aebi-gcc -O0 -S bcp_param.c` est dans le paragraphe 9.5.

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` ?

9.3 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-aebi-gcc -O0 -S var_pile.c` est dans le paragraphe 9.6.

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` ?

9.4 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"
```

9.5 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"
```

9.6 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"
```

Chapitre 10

TD séances 13 et 14 : 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 (on fait semblant... !!!).

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, test d'un bit ; d'où, pour le codage des instructions, les codes : `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} s'imposent...

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

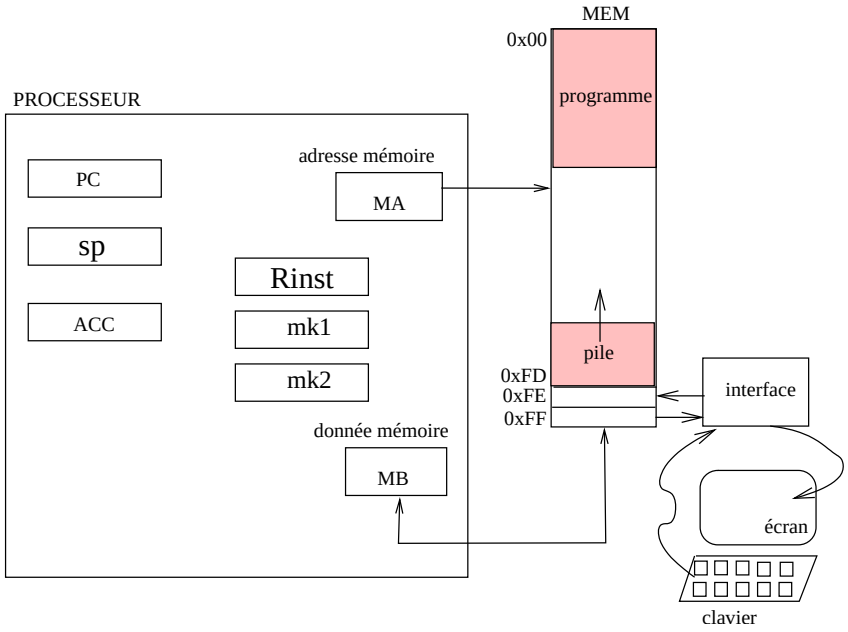

FIGURE 10.2 – La machine à pile et sa mémoire

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

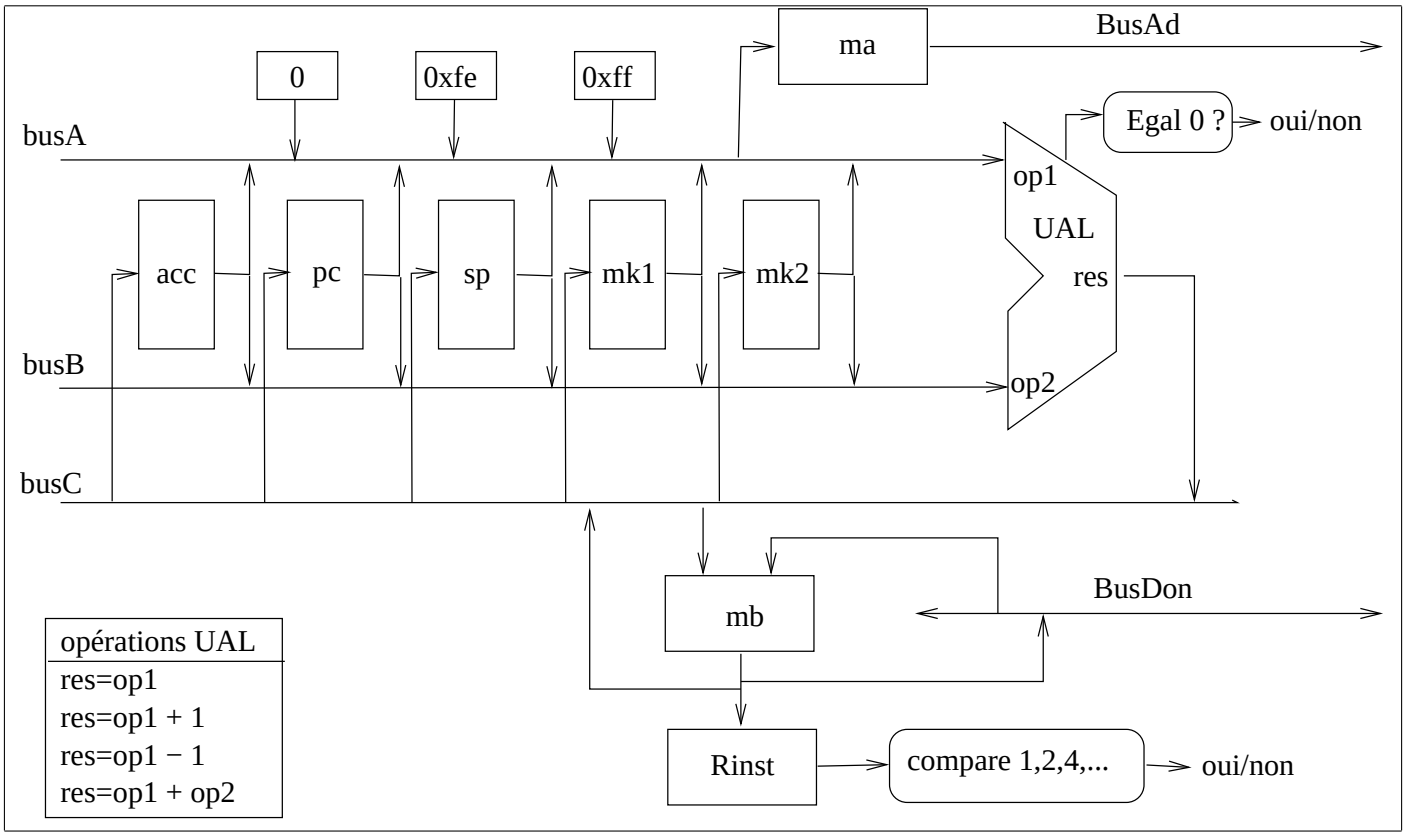

FIGURE 10.3 – Organisation de la machine à pile

— 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, 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. ?

Troisième partie

Travaux Pratiques

Chapitre 1

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, excepté ceux de la première ligne qui doit avoir 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.

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 2.1.4. 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:  BAL fin

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

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

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

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` 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 ?

```
/* XPM */
static char *lignes[]={    /* le fichier doit s'appeler lignes.xpm */
"32 32 3 1",
". c None",
"# c #f0f000",
"a c #ff0000",
"................................",
"................................",
"................................",
"................................",
"................................",
"................................",
"...############.................",
"................................",
"................................",
"................................",
"..................a.............",
"..................a.............",
"..................a.............",
"..................a.............",
"..................a.............",
"..................a.............",
"..................a.............",
"..................a.............",
"..................a.............",
"..................a.............",
"................................",
"................................",
"................................",
"................................",
"................................",
"................................",
"................................",
"................................",
"................................",
"................................",
"................................",
"................................"};
```

FIGURE 1.1 – Le fichier : `lignes.xpm`

Chapitre 2

TP séance 2 : Codage et calculs en base 2

2.1 Présentation de la calculette binaire

Syntaxe d'utilisation

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

operation 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 **subsc2** : 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 préfix **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

2.2 Expérimentation sur l'addition

Voici deux séries d'opérations à réaliser avec la calculette pour être en mesure de répondre aux questions ci-dessous.

No	opération			Calculette	
1	7	$+_{5bits}$	6	addsc	5 7 6
2	7	$+_{4bits}$	6	add	4 7 6
3	7	$+_{3bits}$	6	add	3 7 6
4	8	$+_{4bits}$	8	add	4 8 8
5	8	$+_{5bits}$	8	add	5 8 8

No	opération			Calculette	
6	/8	$+_{6bits}$	/8	add	6 /8 /8
7	-8	$+_{6bits}$	-8	add	6 -8 -8
8	/8	$+_{6bits}$	-8	add	6 /8 -8
9	-7	$+_{5bits}$	-6	add	5 -7 -6
10	-7	$+_{4bits}$	-6	add	4 -7 -6

Comment représente-t-on en base 2 les entiers naturels 7 et 8 ?
Quel est le complément de 7 à $2^3 - 1$, de 7 à $2^4 - 1$, de 8 à $2^4 - 1$ et de 8 à 2^4 ?

Combien de bits faut-il au minimum pour représenter correctement :

1. les intervalles d'entiers naturels $[0 \ldots 7], [0 \ldots 8]$, $[0 \ldots 15]$?
2. les intervalles d'entiers relatifs $[-7 \ldots 7]$, $[-7 \ldots 8]$, $[-8 \ldots +8]$?
3. les entiers relatifs $+7$, -7, $+8$ et -8 ?
4. les sommes d'entiers naturels $6 + 7$ et $8 + 8$?
5. les sommes d'entiers relatifs $+6 + +7$, $+8 + +8$ et $+8 + -5$?

Que peut-on dire des indicateurs Z, N, C et V lorsque le résultat apparent d'une addition est :
— correct sur des entiers naturels et sur des entiers relatifs ?
— correct sur des entiers naturels et faux sur des entiers relatifs ?
— faux sur des entiers naturels et correct sur des entiers relatifs ?
— faux sur des entiers naturels et sur des entiers relatifs ?
— négatif ?
— nul ?

Quel lien existe-t-il entre l'indicateur V et les deux dernières retenues de l'addition ?

2.3 Expérimentation sur la soustraction

Voici quatre opérations de soustraction, à effectuer par la méthode normale et par l'addition du complément à deux. **Observer** les indicateurs **Z**, **N**, **V** et (selon la méthode) **B** (**E**mprunt final) ou **C** (**R**etenue finale).

Soustractions ordinaires $x - y$					
No	opération			Calculette	
1	0x9	$-_{4bits}$	0x8	subsc	4 0x9 0x8
2	0x8	$-_{4bits}$	0x9	sub	4 0x8 0x9
3	0x3	$-_{4bits}$	0x9	sub	4 0x3 0x9
4	0x8	$-_{4bits}$	0x1	sub	4 0x8 0x1

Soustractions par $x + \overline{y} + 1$					
No	opération			Calculette	
1	0x9	$-_{4bits}$	0x8	subc2sc	4 0x9 0x8
2	0x8	$-_{4bits}$	0x9	subc2	4 0x8 0x9
3	0x3	$-_{4bits}$	0x9	subc2	4 0x3 0x9
4	0x8	$-_{4bits}$	0x1	subc2	4 0x8 0x1

Pour des entiers de type relatif, le **signe** du résultat apparent est-il correct ?

Quelle condition N, B et V doivent-ils vérifier pour que la condition $x \geq y$ soit vraie :
— avec x et y de type entier naturel ?

— avec x et y de type entier relatif ?

Quel lien existe-t-il entre les **retenues** de l'addition du complément à deux et les **emprunts** de la soustraction normale ? **Quelle condition** doit vérifier **C** après une soustraction $x - y$ pour que la condition $x \geq y$ soit vraie pour des entiers naturels ?

Chapitre 3

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 3.4.2. 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, ptr_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`.

```
@ accesmem.s
     .data
aa: .word 24
xx: .word 266
bb: .word 42

     .text
     .global main
main:
     LDR r5, ptr_xx
     LDR r6, [r5]

  @ impression du contenu de r5
     MOV r1, r5
     BL EcrHexa32

  @ impression du contenu de r6
     MOV r1, r6
     BL EcrHexa32

fin: BAL exit
```

```
ptr_xx: .word xx
```

Ce programme utilise une fonction d'affichage `EcrHexa32` qui est définie dans un autre module `es.s` (Cf. chapitre 3). 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 3).

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.

```
@ accesmem2.s
    .data
D1:    .word  266
D2:    .hword 42
D3:    .byte 12

    .text
    .global main
main:
      LDR r3, ptr_D1
      LDR r4, [r3]

      LDR r5, ptr_D2
      LDRH r6, [r5]

      LDR r7, ptr_D3
      LDRB r8, [r7]

fin:  BAL exit

ptr_D1:   .word D1
ptr_D2:   .word D2
ptr_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 1048576 ; 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.

```
    .data
DW:    .word  0
DH:    .hword 0
DB:    .byte 0

    .text
    .global main
main:
      LDR r0, ptr_DW
      LDR r1, [r0]
      BL EcrNdecimal32

      MOV r4, #1048576   @  1048576 = 2^20
      LDR r5, ptr_DW
      STR r4, [r5]

LDR r0, ptr_DW
      LDR r1, [r0]
BL EcrNdecimal32

fin:  BAL exit

ptr_DW:   .word DW
ptr_DH:   .word DH
ptr_DB:   .word DB
```

3.3 Un premier programme en langage d'assemblage

Considérons le programme `caracteres.s`.

```
    .data
cc: @ ne pas modifier cette partie
    .byte 0x42
    .byte 0x4f
    .byte 0x4e
    .byte 0x4a
    .byte 0x4f
    .byte 0x55
    .byte 0x52
    .byte 0x00         @ code de fin de chaine
```

```
 @ la suite pourra etre modifiee
 .word 12
 .word 0x11223344
    .asciz "au revoir..."

    .text
    .global main
main:

@ impression de la chaine de caracteres d'adresse cc
     LDR r1, ptr_cc
     BL EcrChaine

@ modification de la chaine
@ A COMPLETER

@ impression de la chaine modifiee
     LDR r1, ptr_cc
     BL EcrChaine

fin: BAL exit

ptr_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`.

3.4 Alignements et "petits bouts"

3.4.1 Questions d'alignements

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

```
x: l'entier 9 sur 8 bits
   l'entier 8 sur 8 bits
   l'entier 3 sur 8 bits
z: l'entier 1024 sur 32 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 le mot d'adresse `z` 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 `z`, 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`).

3.4.2 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 2.3). 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.

Chapitre 4

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

4.1 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 2.4.2).

`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`.

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

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`.
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`.

4.2 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 :

```
   .data
debutTAB: .skip 5*4

   .text
```

```
   .global main
main:

     mov r3, #11             @ val <- 11
     mov r2, #0              @ i <- 0
tq:  cmp r2, #5              @ i-5 ??
     beq fintq
        @ i-5 <> 0
        ldr r0, ptr_debutTAB        @ r0 <- debutTAB
        add r0, r0, r2, LSL #2  @ r0 <- r0 + r2*4 = debutTAB + i*4
        str r3, [r0]            @ MEM[debutTAB+i*4] <- val
        add r2, r2, #1          @ i <- i + 1
        add r3, r3, #11         @ val <- val + 11
     bal tq
fintq:  @ i-5 = 0

fin:  bal exit

ptr_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'agorithme 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.

4.3 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.

Chapitre 5

TP séances 6 et 7 : Parcours de tableaux

5.1 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 5.1.

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;
```

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 5.1 – Tables de multiplication de 1 à 10

Questions concernant le remplissage du tableau

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

5.2 Affichage du tableau

On donne ci-dessous un algorithme pour afficher le tableau conformément à la figure 5.1, 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 5.1 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`.

5.3 Remplissage du tableau

Il s'agit maintenant de traduire en langage d'assemblage l'algorithme de remplissage du tableau donné au paragraphe 5.1.

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 des additions successives selon l'algorithme suivant :

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

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

Votre compte-rendu comportera cette version intermédiaire de la traduction.

5.3.1 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`.

5.3.2 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 `gbd` 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.

5.3.3 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

5.4 tabmult.s

```
@ Programme tabmult : Affiche les tables de multiplication de de 1 a 10
N_MAX= 10
   .data
barre :  .byte '|'
         .byte 0
espace : .byte ' '
         .byte 0
tirets : .asciz "---"

debutTab:    .skip N_MAX*N_MAX*4   @ adresse du debut du tableau

   .text
   .global main
main:

    @ remplissage du tableau
        @ a completer...
```

```
    @ affichage du tableau
        @ a completer...

   BAL exit

ptr_debutTab : .word debutTab
adr_barre :    .word barre
adr_espace :   .word espace
adr_tirets :   .word tirets
```

Chapitre 6

TP séances 8, 9 et 10 : Procédures, fonctions et paramètres

6.1 Traitement des fichiers au format bitmap

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.

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 6.1 ci-après ;
3. Rotations comme dans la figure 6.1 ci-après.

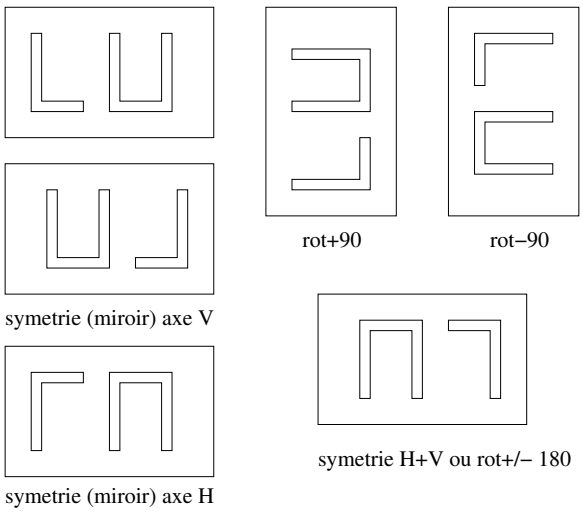

FIGURE 6.1 – Exemples d'opérations de symétrie et rotation d'une image

6.2 Négatif d'une image au format bitmap

Le premier objectif de ces TP est de réaliser un programme capable de produire un fichier au format bitmap qui contienne le négatif de l'image donnée en entrée au format bitmap aussi.

Le programme principal écrit en C, `negatif.c`, est disponible ci-après, il fait appel à une procédure écrite en langage d'assemblage dans le fichier `util.s`, que vous devez le compléter par la procédure `NEG` et la fonction `NON_OCT` décrites ci-dessous.

6.2.1 Spécifications de NEG et NON_OCT

La procédure NEG(tt, n) transforme le tableau d'adresse `tt` en son négatif, `n` étant la taille du tableau en nombre d'octets. Cette procédure est exportée vers le programme principal qui l'utilise.

```
NEG : procédure (tt: adresse d'un tableau d'octet, n: entier)
   pour i de 0 à n-1 : tt[i] <-- not tt[i]
```

Les conventions de passage des paramètres sont : l'adresse du tableau est dans `r0`, la valeur de `n` est dans `r1`.

La fonction `NON_OCT` est utilisée par la procédure `NEG` pour calculer la négation bit à bit d'un octet. Cette fonction est locale à ce fichier.

```
NON_OCT : fonction (x : un octet) --> un octet
```

Les conventions de passage du paramètre et du résultat sont : l'octet donnée est dans r4, l'octet résultat est dans r5.

6.2.2 Consignes de compilation

Le fichier contenant une image exemple `charlot.bm` et le fichier contenant le programme principal `negatif.c` sont disponibles sur le site de l'UE.

Un petit rappel pour compiler vos programmes :

```
arm-eabi-gcc -c -Wa,-gdwarf2 util.s
arm-eabi-gcc -c -g negatif.c
arm-eabi-gcc -o exec -g negatif.o util.o
arm-eabi-run exec
```

Le programme lit le fichier `charlot.bm` et produit le fichier `resultat.bm`. Pour afficher ces fichiers, utiliser la commande `bitmap` (cf. le premier TP).

6.2.3 util.s

```
   .text
   .global NEG

@ NEG : procedure (tt: tableau d'octet, n: entier)
@ NEG(tt, n) transforme le tableau d'adresse tt en son negatif
@ n est la taille du tableau en nombre d'octets
@ adresse du tableau dans r0, n dans r1

NEG:
```

```
@ a completer

@ NON_OCT : fonction (x : un octet) --> un octet
@ NON_OCT(x) est la negation bit a bit de l'octet x
@ donnee dans r4, resultat dans r5

NON_OCT:

@ a completer
```

6.2.4 negatif.c

```
/* transformation d'un image au format bitmap en son image negative
 *
 * le programme produit un fichier resultat.bm
 *
 * ce programme fait appel a la procedure NEG qui transforme
 * un tableau de bits en sa negation bit a bit
 *
 */

#include <stdio.h>
#include "charlot.bm"

/* NEG : procedure (tt : tableau d'octet, n : entier)
 * NEG(tt) transforme le tableau d'adresse tt en son negatif
 * n est la taille du tableau en nombre d'octets
 */
extern void NEG (unsigned char *ptr_tab, int n);

int main(int argc, char *argv[]) {
int nboctets, i, nblignes, j;
FILE *fich_res;

   /* un pixel = 1 bit d'ou 1 octet vaut pour 8 pixels
    * et nombre d'octets de l'image = (longueur * largeur) / 8 */

   nboctets = (charlot_width*charlot_height) >> 3;

   printf ("Calcul du negatif de l'image\n");

   /* ************************************** */
   /* appel d'une procedure ecrite en assembleur */
   /* ************************************** */

   NEG (charlot_bits, nboctets);

   /* ************************************** */
   /* version en C */
```

```
   /*
      for (i=0;i<nboctets;i++) {
         charlot_bits[i] = ~charlot_bits[i];
      }
   */
   /* ************************************** */

   printf ("Production du fichier resultat.bm\n");
   if ( (fich_res = fopen("resultat.bm", "w")) == NULL ) {
       printf ("%s : impossible d'ouvrir le fichier %s\n",
                argv[0], "resultat.bm");
      exit(1);
   }

   fprintf (fich_res, "#define charlot_width %d\n", charlot_width);
   fprintf (fich_res, "#define charlot_height %d\n", charlot_height);
   fprintf (fich_res, "static unsigned char charlot_bits[] = {\n");
   nblignes = nboctets / 12;   /* on ecrit des lignes de 12 octets */
   /* les nb-1 premieres lignes */
   for (j=0; j < nblignes; j++) {
      for (i=0; i<12; i++) {
         fprintf (fich_res, "0x%x, ", charlot_bits[(j*12) + i]);
      };
      fprintf (fich_res, "\n");
   }
   /* la derniere ligne (peut etre moins de 12 octets) */
   for (i=0; i < (nboctets - (nblignes*12) -1)   ; i++) {
      fprintf (fich_res, "0x%x, ", charlot_bits[(nblignes*12)+i]);
   }
   /* le dernier n'est pas suivi d'une virgule */
   fprintf (fich_res, "0x%x", charlot_bits[nboctets-1]);
   fprintf (fich_res, "};\n");
   fclose (fich_res);

}
```

6.3 Miroir vertical d'une image au format bitmap

Le code ci-après en langage C propose deux méthodes de calcul du symétrique d'un octet :

1. Avec une constante tableau préinitialisée (cf. tabsym.h) ;
2. Par des opérations bit à bit + décalage.

Adaptez le code main de la section précédente pour valider votre programme en langage d'assemblage ARM qui fait l'algorithme décrit ci-après en langage C.

6.3.1 symetrie.c

```
#include <stdio.h>
#include "commun.h"
#include "image_test.bm"
```

```
#ifndef SYMTAB
void symetrie_octet (unsigned char *adresse) {
  unsigned char octet;

  octet = *adresse;

  // echange de quartets adjacents
  octet = (octet & 0xF0) >> 4 | (octet & 0x0F) <<4;
  // echange de doublets adjacents
  octet = (octet & 0xCC) >> 2 | (octet & 0x33) <<2;
  // echange de bits adjacents
  octet = (octet & 0xAA)>> 1 | (octet & 0x55) <<1;

  *adresse = octet;
}
#else
#include "tabsym.h"

void symetrie_octet (unsigned char *adresse) {
  unsigned char octet;

  octet = *adresse;
  octet = tabsym_octet[octet];
  *adresse = octet;
}
#endif


void permuter_ligne (unsigned char *tab, unsigned int octets_par_ligne, unsigned int col) {
   unsigned char tmp;

   tmp = tab[octets_par_ligne -1 - col];
   tab [octets_par_ligne -1 - col] = tab [col];
   tab[col] = tmp;
}


void symetrie (unsigned char *image) {
   unsigned int position;
   unsigned int li,col;

   // symetriser chaque octet
   for (position = 0; position < height*octets_par_ligne_image; position++) {
       symetrie_octet (image+position);
       }
   // symetrie verticale octet par octet
   for (li=0;li<height;li++) {
      for (col=0; col<octets_par_ligne_image/2;col++)
         permuter_ligne (image+li*octets_par_ligne_image, octets_par_ligne_image,col);
```

```
    }
}
```

Chapitre 7

TP séance 11 : Passage de paramètres par les registres

7.1 Calcul de $n!$ au moyen d'une action itérative

On considère le lexique suivant :

```
. fact1 : une action(la donnée : un Entier > 0,
                     le résultat : un Entier > 0)
  { fact1(n, facn) :
    état initial : r5 = n, r7 = @facn
    état final :   facn = n! }
. mult : deux Entiers >= 0 -> un Entier >= 0
  { mult(a, b) = a * b
    pré-condition : r0 = a, r1 = b
    post-condition : r0 = a * b }
```

On s'intéresse à l'action `fact1`. Une réalisation itérative en est donnée par l'algorithme suivant :

```
fact1(n, facn) :
   x : un Entier sur [0 .. n - 1]
   res : un Entier > 0
   x <- n - 1
   res <- n
   tant que x <> 0
      res <- mult(res, x)
      x <- x - 1
   facn <- res
```

On se propose de coder cette action en langage d'assemblage.

Exercice 1 :

1. Traduire l'algorithme en langage d'assemblage Arm : compléter la définition de la procédure `fact1` (fichier `fact1.s`, cf. annexe I) : cette procédure devra faire appel à la fonction `mult` (fichier `multiplication.s`, cf. paragraphe 7.3).
2. Compléter le corps de la procédure principale `main` du fichier `essai-fact1.s` (cf. annexe III) : vous devez ajouter à l'endroit voulu de ce fichier l'appel à la procédure `fact1`.

3. Compilez et testez le programme. Vous disposez d'un fichier `Makefile` vous permettant d'utiliser la commande `make` pour compiler le programme. Vous pouvez (vous devez !) utiliser `arm-eabi-gdb` pour éliminer les erreurs d'exécution.

Remarque : pour ce premier programme, il est inutile de sauvegarder les registres temporaires dans la procédure `fact1`.

7.2 Calcul de $n!$ au moyen d'une fonction récursive

On ajoute à présent dans le lexique la déclaration suivante :

```
. fact0rec : un Entier > 0 -> un Entier > 0
  { fact0rec(n) = n!
    pré-condition : r0 = n
    post-condition : r1 = n! }
```

Une réalisation récursive de la fonction `fact0rec` est donnée par l'algorithme suivant :

```
fact0rec(n) :
   facn : un Entier > 0
   si n = 1 alors
      facn <- 1
   sinon
      facn <- mult(fact0rec(n - 1), n)
   fact0rec(n) : facn
```

Exercice 2 :

1. Traduire l'algorithme en langage d'assemblage Arm : compléter la définition de la fonction `fact0rec` (fichier `fact0rec.s`, cf. annexe IV).
2. Compléter le corps de la procédure principale `main` du fichier `essai-fact0rec.s` (cf. annexe V). Vous devez ajouter à l'endroit voulu de ce fichier l'appel à la fonction `fact0rec`.
3. Compilez et testez le programme.

7.3 Calcul de $n!$ au moyen d'une action récursive

On reprend la spécification de l'action `fact1` réalisée en section 7.1 en la modifiant comme suit :

```
. fact1rec : une action(la donnée : un Entier > 0,
                        le résultat : un Entier > 0)
  { fact1rec(n, facn) :
    état initial : r0 = n, r1 = @facn
    état final :   facn = n! }
```

On veut maintenant donner une réalisation récursive de cette action, selon l'algorithme suivant :

```
fact1rec(n) :
   facn_1 : un Entier > 0
   si n = 1 alors
      facn <- 1
   sinon
      fact1rec(n - 1, facn_1)
      facn <- mult(facn_1, n)
```

Exercice 3 :

1. Traduire le nouvel algorithme en langage d'assemblage Arm : compléter la définition de la fonction `fact1rec` (fichier `fact1rec.s`, cf. annexe VI).
2. Compléter le corps de la procédure principale `main` du fichier `essai-fact1rec.s` (cf. annexe VII). Vous devez ajouter à l'endroit voulu de ce fichier l'appel à la fonction `fact1rec`.
3. Compilez et testez le programme.

Annexe I : le fichier `fact1.s`

```
@ fact1.s : realisation iterative d'une action
@           sans sauvegarde des registres

@ procedure fact1
@ parametres : A COMPLETER
@ algorithme : A COMPLETER
@ allocation des registres : r3 <-> x
@                            r4 <-> res

         .text
         .global  fact1
fact1:
         @@@@@@@@@@@@@@
         @ A COMPLETER
         @@@@@@@@@@@@@@
```

Annexe II : le fichier `multiplication.s`

```
@  Algorithme de multiplication par addition et decalage
@
@  Principe : si a = Sigma de i=0 a n-1 des a_i * 2^i
@  alors  resultat = Sigme de i=0 a n-1 des (a_i * 2^i) * b_i
@  Pour a entier relatif : a_(n-1) est a mutiplier par -2^(n-1)
@
@  Principe : pour chaque bit i de a a 1, ajouter   b << i
@             (ajouter b si a_i et faire b = b*2 a chaque passage)
@
@ int mult (int a, int b) {
@   int resultat;
@   unsigned int au;     // a pris comme un naturel (pour >>)
@
@   au = a; resultat = 0;
@   if (a <0) {          // se ramener au cas a >= 0
@     au = -a; b = -b;
@   }
@   while (au != 0) {
@     if ((au & 1) != 0) // ajouter b si a_0 == 1
@        resultat = resultat + b;
@     b = b * 2;         // ou b = b << 1
@     au = au / 2;       // ou au = au >> 1
@   }
@   return resultat;
@ }
@
```

```
@ Convention d'appel :
@                  a : r0, b : r1, valeur retour : r0
@ Registres temporaires :
@                  resultat : r2, au: confondu avec a: r0
   .text
   .global   mult
mult:
   stmfd   sp!, {r1,r2}   @ sauver aussi b par precaution
   mov   r2,#0        @ resultat =0
   cmp   r0,#0        @ if (a<0)
   rsblt   r0,r0,#0   @   au = -a
   rsblt   r1,r1,#0   @   b = -b
    b   condtq         @ while (au != 0)
tq: tst   r0,#1        @ if ((au&1) != 0)
      addne   r2,r2,r1      @ resultat = resultat + b
      mov   r1,r1, LSL #1   @ b = b *2
      mov   r0,r0, LSR #1   @ au = au /2
condtq: cmp   r0,#0
    bne   tq
   mov   r0,r2      @ return resultat
   ldmfd   sp!,{r1,r2}
   mov   pc,lr
```

Annexe III : le fichier `essai-fact1.s`

```
@ essai-fact1.s
        .data
n:      .word    0              @ donnee
facn:   .word    0              @ resultat
invite: .asciz   "Saisir un entier > 0 :"

        .text
        .global  main
@ procedure principale
main:
        @ saisir n
        ldr      r1, adr_invite
        bl       EcrChaine
        ldr      r1, adr_n
        bl       Lire32

        @ appel de la procedure fact1(n, facn)
        @@@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@@

        @ afficher n!
        ldr      r1, adr_facn
        ldr      r1, [r1]
        bl       EcrNdecimal32

        @ fin de la procedure principale
        bal      exit

@ adresses pour l'acces en zone data
```

```
adr_n:
         .word    n
adr_facn:
         .word    facn
adr_invite:
         .word    invite
```

Annexe IV : le fichier `fact0rec.s`

```
@ fact0rec.s : realisation recursive d'une fonction

@ fonction fact0rec : A COMPLETER
@ parametres : A COMPLETER
@ algorithme : A COMPLETER
@ allocation des registres : r0 <-> n, n - 1
@                            r1 <-> n!, (n - 1)!
@                            r2 <-> n * (n - 1)!

         .text
         .global  fact0rec
fact0rec:
         @@@@@@@@@@@@@@
         @ A COMPLETER
         @@@@@@@@@@@@@@
```

Annexe V : le fichier `essai-fact0rec.s`

```
@ essai-fact0rec.s
         .data
n:       .word    0              @ donnee
invite:  .asciz   "Saisir un entier >= 0 :"

         .text
         .global  main
@ procedure principale
main:
         @ saisir n
         ldr      r1, adr_invite
         bl       EcrChaine
         ldr      r1, adr_n
         bl       Lire32

         @ appel de la fonction fact0rec(n)
         @@@@@@@@@@@@@@
         @ A COMPLETER
         @@@@@@@@@@@@@@

         @ afficher n!
         bl       EcrNdecimal32

         @ fin de la procedure principale
         bal      exit

@ adresses pour l'acces en zone data
adr_n:
```

```
        .word    n
adr_invite:
        .word    invite
```

Annexe VI : le fichier `fact1rec.s`

```
@ fact1rec.s : realisation recursive d'une action

@ procedure fact1rec : A COMPLETER
@ parametres : A COMPLETER
@ algorithme : A COMPLETER
@ allocation des registres : r0 <-> n, n - 1
@                            r1 <-> @facn, @facn_1
@                            r2 <-> n * (n - 1)!

        .text
        .global  fact1rec
fact1rec:
        @@@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@@
```

Annexe VII : le fichier `essai-fact1rec.s`

```
@ essai-fact1rec.s
        .data
n:      .word    0              @ donnee
facn:   .word    0              @ resultat
invite: .asciz   "Saisir un entier >= 0 :"

        .text
        .global  main
@ procedure principale
main:
        @ saisir n
        ldr      r1, adr_invite
        bl       EcrChaine
        ldr      r1, adr_n
        bl       Lire32

        @ appel de la procedure fact1rec(n, facn)
        @@@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@@

        @ afficher n!
        ldr      r1, [r1]
        bl       EcrNdecimal32

        @ fin de la procedure principale
        bal      exit

@ adresses pour l'acces en zone data
adr_n:
        .word    n
```

```
adr_facn:
         .word    facn
adr_invite:
         .word    invite
```

Annexe VIII : version C des procédures et fonctions

```
// Protoype de la fonction mult
unsigned int mult (unsigned a, unsigned b);

void fact1 (unsigned int n, unsigned int *facn)
{
  unsigned int x,res;
  x = n-1;
  res = n;
  while (x != 0) {
    res = mult(res,x);
    x = x-1;
  }
  *facn = res;
}

unsigned int fact0rec(unsigned int n)
{
  unsigned int facn;
  if (n==1) {
    facn=1;
  } else {
    facn = mult(fact0rec(n-1),n);
  }
  return facn;
}

void fact1rec (unsigned int n, unsigned int *facn)
{
  unsigned int facn_1;
  if (n==1) {
    *facn = 1;
  } else {
  fact1rec(n-1,&facn_1);
  *facn = mult(facn_1,n);
  }
}
```

Chapitre 8

TP séance 12 : Code en langage d'assemblage produit par un compilateur C

Le code en langage d'assemblage ARM d'un programme en C peut être produit sans optimisation par le compilateur `gcc` avec l'option `-O0`. Dans ce TP, nous allons observer ce qu'un compilateur peut optimiser. Nous allons reprendre les mêmes programmes et codes du chapitre II.9, éventuellement modifiés mais en les compilant avec un niveau d'optimisation important (option `-O2`).
Pour ce TP n'hésitez pas à faire d'autres essais que ceux qui sont proposés.
Un autre objectif du TP est de distinguer :

- ce qui est du domaine du "statique", c'est-à-dire ce qui peut être calculé lors de la compilation,
- ce qui est du domaine du "dynamique", c'est-à-dire ce qui ne peut être calculé que lors de l'exécution.

8.1 Un premier exemple

Récupérez le programme écrit en langage C dans le fichier `premier.c`.
Compilez ce programme avec un bon niveau d'optimisations (option `-O2`) avec la commande suivante : `arm-aebi-gcc -O2 -S premier.c`. Etudiez le code en langage d'assemblage ARM produit dans le fichier `premier.s` et comparez avec le code étudié dans le chapitre II.9.

8.2 Programme avec une procédure qui a beaucoup de paramètres

8.2.1 Premier essai

On considère le programme contenu dans le fichier `bcp_param.c`. Compilez ce programme avec la commande : `arm-aebi-gcc -O2 -S bcp_param.c`. Etudiez le programme produit dans `bcp_param.s` et comparez avec le code étudié dans le chapitre II.9.

8.2.2 Deuxième essai

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, étudiez le code produit, comparez avec la version précédente.

8.2.3 Troisième essai

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 la version précédente.

8.3 Les variables locales peuvent prendre beaucoup de place

Soit le programme contenu dans le fichier `var_pile.c`. Compilez ce programme avec la commande : `arm-aebi-gcc -O2 -S bcp_param.c`. Etudiez le programme produit dans `bcp_param.s` et comparez avec le code étudié dans le chapitre II.9.

Chapitre 9

TP séances 13 et 14 : Procédures et paramètres

9.1 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 quatre paramètres des types : 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 2.1.6).

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`, cf. annexe I).
2. Compléter le corps de la procédure principale `main` du fichier `essai-map.s` (cf. annexe III) : 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`, cf. annexe VI).
   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`, cf. annexe II).
3. Compilez et testez le programme (faites un fichier `Makefile` permettant d'utiliser la commande `make` pour compiler le programme).

9.2 Réduction d'un tableau à une valeur

9.2.1 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 un paramètre 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`, cf. annexe IV).
2. Compléter le corps de la procédure principale `main` du fichier `essai-red.s` (cf. annexe V). 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`, cf. annexe II).
3. Compilez et testez le programme.

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

Exercice 3 :

1. Quels sont-ils 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.

9.3 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. 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).

Annexe I : le fichier `map.s`

```
@ procedure map
@ parametres : A COMPLETER
@ algorithme : A COMPLETER
@ allocation des registres : A COMPLETER

        .text
        .global  map
map:
        @@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@
```

Annexe II : le fichier `fg.s`

```
        .global plus_un, carre
        .global somme, produit
        .text
@ fonction plus_un : incremente l'entier passe en parametre
@ r3 : donnee
@ r4 : resultat
plus_un: add      r4, r3, #1
        mov      pc, lr

@ fonction carre : eleve au carre l'entier passe en parametre
@ r3 : donnee
@ r4 : resultat
carre:   sub      sp, sp, #4
        str      lr, [sp]
        sub      sp, sp, #4
        str      r0, [sp]
        sub      sp, sp, #4
        str      r1, [sp]
        sub      sp, sp, #4
        str      r2, [sp]

        mov      r0, r3
        mov      r1, r3
        bl       mult
        mov      r4, r0

        ldr      r2, [sp]
        add      sp, sp, #4
        ldr      r1, [sp]
```

```
        add     sp, sp, #4
        ldr     r0, [sp]
        add     sp, sp, #4
        ldr     lr, [sp]
        add     sp, sp, #4
        mov     pc, lr

@ fonction somme : ajoute les deux entiers passes en parametre
@ r0, r1 : donnees
@ r2 : resultat
somme:  add     r2, r0, r1
        mov     pc, lr

@ fonction produit : multiplie les deux entiers passes en parametre
@ r0, r1 : donnees
@ r2 : resultat
produit:
        @@@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@@
```

Annexe III : le fichier `essai-map.s`

```
        .set    NMAX, 10        @ nombre d'elements

        .data
invite1: .asciz  "Saisir une sequence de "
invite2: .asciz  " entiers :"
afftab1: .asciz  "Sequence donnee S :"
afftab2: .asciz  "map(S, plus_un) :"
afftab3: .asciz  "map(S, carre) :"
tab1:   .skip   NMAX            @ tableau de NMAX octets
tab2:   .skip   NMAX            @ tableau de NMAX octets

        .text
        .global  main
@ procedure principale
main:
        @ saisir la sequence donnee
        ldr     r1, adr_invite1
        bl      EcrChn
        mov     r1, #NMAX
        bl      EcrNdecim32
        ldr     r1, adr_invite2
        bl      EcrChaine
        ldr     r0, adr_tab1
        mov     r1, #NMAX
        bl      saisir_tab

        @ afficher la sequence donnee
        bl      AlaLigne
        ldr     r1, adr_afftab1
        bl      EcrChaine
        ldr     r0, adr_tab1
        mov     r1, #NMAX
```

```
        bl      afficher_tab

        @ appel de la procedure map(tab1, NMAX, tab2, plus_un)
        @@@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@@

        @ afficher la sequence resultat
        bl      AlaLigne
        ldr     r1, adr_afftab2
        bl      EcrChaine
        @@@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@@

        @ appel de la procedure map(tab1, NMAX, tab2, carre)
        @@@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@@

        @ afficher la sequence resultat
        bl      AlaLigne
        ldr     r1, adr_afftab3
        bl      EcrChaine
        @@@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@@

        @ fin du programme principal
        bal     exit

@ relais vers la zone data
adr_invite1:
        .word   invite1
adr_invite2:
        .word   invite2
adr_afftab1:
        .word   afftab1
adr_afftab2:
        .word   afftab2
adr_afftab3:
        .word   afftab3
adr_tab1:
        .word   tab1
adr_tab2:
        .word   tab2

@ relais vers la zone text
adr_plus_un:
        .word   plus_un
adr_carre:
        .word   carre
```

Annexe IV : le fichier `red.s`

```
        .text
```

```
@ fonction red
@ parametres : A COMPLETER
@ algorithme : A COMPLETER

red:
        @@@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@@
```

Annexe V : le fichier `essai-red.s`

```
        .set    NMAX, 10        @ nombre d'elements

        .data
invite1:   .asciz   "Saisir une sequence de "
invite2:   .asciz   " entiers :"
afftab:    .asciz   "Sequence donnee S :"
affres1:   .asciz   "red(S, somme) = "
tab :      .skip    NMAX           @ tableau de NMAX octets
var_somme: .byte    0

        .text
        .global  main
@ procedure principale
main:
        @ saisir la sequence donnee
        ldr     r1, adr_invite1
        bl      EcrChn
        mov     r1, #NMAX
        bl      EcrNdecim32
        ldr     r1, adr_invite2
        bl      EcrChaine
        ldr     r0, adr_tab
        mov     r1, #NMAX
        bl      saisir_tab

        @ afficher la sequence donnee
        bl      AlaLigne
        ldr     r1, adr_afftab
        bl      EcrChaine
        mov     r1, #NMAX
        bl      afficher_tab

        @ appel de la fonction red(tab, NMAX, 0, somme)
        @@@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@@

        @ afficher le resultat
        bl      AlaLigne
        ldr     r1, adr_affres1
        bl      EcrChn
        @@@@@@@@@@@@@@
        @ A COMPLETER
        @@@@@@@@@@@@@@
```

```
        @ fin du programme principal
        bal     exit

@ relais vers la zone data
adr_invite1:
        .word   invite1
adr_invite2:
        .word   invite2
adr_afftab:
        .word   afftab
adr_affres1:
        .word   affres1
adr_tab:
        .word   tab
adr_var_somme:
        .word   var_somme

@ relais vers la zone text
adr_somme:
        .word   somme
```

Annexe VI : le fichier gestion_tab.s

```
        .data
entier: .byte   0             @ entiers de la sequence

        .text
        .global  saisir_tab, afficher_tab

@ procedure saisir_tab : saisit une sequence d'entiers
@ r0 = T : adresse de debut du tableau contenant la sequence
@ r1 = N : nombre d'elements de la sequence
@ algorithme : i parcourant 0 .. N - 1 : Lire8(T[i])

saisir_tab:
        sub     sp, sp, #4     @ sauvegarde adresse de retour
        str     lr, [sp]
        sub     sp, sp, #4     @ sauvegarde temporaires
        str     r1, [sp]
        sub     sp, sp, #4
        str     r2, [sp]
        sub     sp, sp, #4
        str     r5, [sp]
        sub     sp, sp, #4
        str     r6, [sp]

        mov     r5, #0         @ indice dans le tableau
        mov     r6, r1         @ nombre d'elements

tantque1:
        cmp     r5, r6
        beq     fintq1
        ldr     r1, adr_entier @ lire un entier
        bl      Lire8
        ldrb    r2, [r1]
        strb    r2, [r0, r5]   @ le ranger dans le tableau
        add     r5, r5, #1     @ octet suivant
        bal     tantque1
fintq1:
        ldr     r6, [sp]       @ restauration temporaires
        add     sp, sp, #4
        ldr     r5, [sp]
        add     sp, sp, #4
        ldr     r2, [sp]
        add     sp, sp, #4
        ldr     r1, [sp]
        add     sp, sp, #4
        ldr     lr, [sp]       @ restauration adresse de retour
        add     sp, sp, #4
        mov     pc, lr         @ retour a l'appelant
adr_entier:
        .word   entier

@ procedure afficher_tab : affiche une sequence d'entiers
@ r0 = T : adresse de debut du tableau contenant la sequence
@ r1 = N : nombre d'elements de la sequence
@ algorithme : i parcourant 0 .. N - 1 : EcrZdecimal8(T[i])
```

```
afficher_tab:
        sub     sp, sp, #4      @ sauvegarde adresse de retour
        str     lr, [sp]
        sub     sp, sp, #4      @ sauvegarde temporaires
        str     r1, [sp]
        sub     sp, sp, #4
        str     r5, [sp]
        sub     sp, sp, #4
        str     r6, [sp]
        sub     sp, sp, #4
        str     r7, [sp]

        @ corps de la procedure
        mov     r5, #0          @ indice dans tableau
        mov     r6, r0          @ adresse de debut
        mov     r7, r1          @ taille du tableau

tantque2:
        cmp     r5, r7
        beq     fintq2
        ldrb    r1, [r6, r5]    @ on recupere l'octet courant
        bl      EcrZdecim8     @ qui est imprime
        mov     r1, #' '
        bl      EcrCar          @ en le separant du suivant par un espace
        add     r5, r5, #1      @ octet suivant
        bal     tantque2

fintq2:
        bl      AlaLigne
        ldr     r7, [sp]        @ restauration temporaires
        add     sp, sp, #4
        ldr     r6, [sp]
        add     sp, sp, #4
        ldr     r5, [sp]
        add     sp, sp, #4
        ldr     r1, [sp]
        add     sp, sp, #4
        ldr     lr, [sp]        @ restauration adresse de retour
        add     sp, sp, #4
        mov     pc, lr          @ retour a l'appelant
```

Quatrième partie

Annexes

Chapitre 1

Annexe I : Codage ASCII des caractères

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL	32	20	SPACE	64	40	@	96	60	‘
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	”	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	’	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Chapitre 2

Annexe II : Représentation des nombres en base 2

La figure 2.1 illustre les représentations d'entiers naturels et signés pour une taille de mot de 4 bits. A chaque entier peut être associé un angle. Effectuer une addition revient à ajouter les angles correspondant. Un débordement de produit au-delà d'un demi-tour en arithmétique signée ou d'un tour complet en arithmétique naturelle.
Le tableau suivant récapitule les principales puissances de 2 utiles, avec leur représentation en hexadécimal et les puissances de 10 approchées correspondantes.

n				2^n		
décimal	hexa	octal	binaire	décimal	hexa	commentaire
0	0	00	0000	1	1	
1	1	01	0001	2	2	
2	2	02	0010	4	4	
3	3	03	0011	8	8	
4	4	04	0100	16	10	un quartet = un chiffre hexa
5	5	05	0101	32	20	
6	6	06	0110	64	40	
7	7	07	0111	128	80	
8	8	10	1000	256	100	un octet = deux chiffres hexa
9	9	11	1001	512	200	
10	A	12	1010	1024	400	$1K_b$
11	B	13	1011	2048	800	$2K_b$
12	C	14	1100	4096	1000	$4K_b$
13	D	15	1101	8192	2000	$8K_b$
14	E	16	1110	16384	4000	$16K_b$
15	F	17	1111	32768	8000	$32K_b$
16	10	20	10000	65536	10000	$64K_b$
20	14	24	10100	1048576	100000	$1M_b = 1K_b^2 = 5$ chiffres
30	1E	36	11110	$\tilde{}1.07 \times 10^9$	40000000	$1G_b = 1K_b^3$

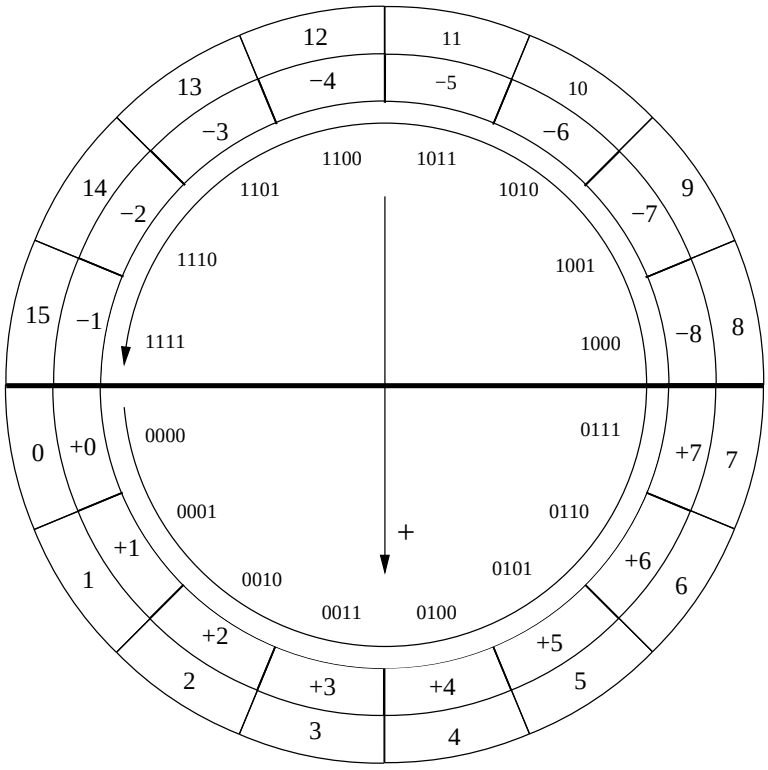

FIGURE 2.1 – Représentation d'entiers naturels et signés sur 4 bits

Chapitre 3

Annexe III : Spécification des fonctions d'entrée/sortie définies dans `es.s`

```
@ fichier es.s
@ fonctions d'entrees sorties

@ AlaLigne :
@    retour a la ligne

@ EcrCar :
@    ecriture d'un caractère dont la valeur est dans r1

@ EcrChn :
@    ecriture de la chaine sans retour à la ligne dont l'adresse est dans r1

@ EcrChaine :
@    ecriture de la chaine dont l'adresse est dans r1

@ EcrHexa32 :
@    ecriture d'un mot de 32 bits en hexadécimal
@    la valeur a afficher est dans r1

@ EcrZdecimalf32 :
@    ecriture en decimal d'un entier relatif represente sur 32 bits
@    l'entier est dans r1

@ EcrZdecimal16 :
@    ecriture en decimal d'un entier relatif represente sur 16 bits
@    l'entier est dans les 16 bits de poids faibles de r1

@ EcrZdecimal8 :
@    ecriture en decimal d'un entier relatif represente sur 8 bits
@    l'entier est dans les 8 bits de poids faibles de r1
@    attention : les bits 15 a 8 de r1 sont eventuellement modifies

@ EcrNdecimal32 :
@    ecriture en decimal d'un entier naturel represente sur 32 bits
@    l'entier est dans r1

@ EcrNdecimal16 :
@    ecriture en decimal d'un entier naturel represente sur 16 bits
@    l'entier est dans les 16 bits de poids faibles de r1
```

```
@ EcrNdecimal8 :
@    ecriture en decimal d'un entier naturel represente sur 8 bits
@    l'entier est dans les 8 bits de poids faibles de r1
@    attention : les bits 15 a 8 de r1 sont mis a 0

@ Lire32 :
@    lecture d'un entier represente sur 32 bits
@    l'adresse de l'entier doit etre donnee dans r1

@ Lire16 :
@    lecture d'un entier represente sur 16 bits
@    l'adresse de l'entier doit etre donnee dans r1

@ Lire8 :
@    lecture d'un entier represente sur 8 bits
@    l'adresse de l'entier doit etre donnee dans r1

@ LireCar :
@    lecture d'un caractere tape au clavier
@    l'adresse du caractere (code en ascii) doit etre donnee dans r1
```