Introduction au langage C et au langage machine ARM : du C au langage d'assemblage

Philippe WAILLE
Université Joseph Fourrier
UFR IMA

6 juillet 2006

Objectifs de ce document

Ce document est une introduction au langage machine et au langage C. Le but est de donner au lecteur une notion des actions élémentaires effectuées dans la machine et un aperçu du processus de traduction manuelle des contructions d'un langage évolué (C en l'occurence) en suite d'instructions élémentaires du langage machine d'un processeur RISC.

En pratique, la traduction est automatisée et confiée à des programmes traducteurs : les compilateurs, dont il n'est pas question d'aborder ici le fonctionnement interne. Les notions présentées ici sont en revanche suffisantes pour permettre la compréhension du code machine généré par le compilateur gnu en l'absence d'optimisation poussée.

Ce document couvre un surensemble des prérequis du module "Architectures Logicielles et Matérielles" de troisième année de la licence d'informatique de Grenoble, enseignées dans le module éponyme de deuxième année (INF241).

Les deux premiers chapitres introduisent la représentation des nombres en base 2 et les types et expressions correspondant du langage C.

Quelques concepts essentiels d'architecture des ordinateurs sont ensuite présentés dans les chapitres 3 et 4 : l'organisation générale d'un ordinateur, et les notions de langage machine et d'assemblage, de jeux d'instructions et de modes d'adressage.

La suite du document est consacrée à la programmation en langage d'assemblage d'un processeur RISC fictif inspiré des processeurs 32 bits ARM et SPARC v7. Elle insiste sur les primitives importantes du langage C et sur la technique de traduction systématique du C en langage d'assemblage.

La notion d'étiquette, les directives de réservation de mémoire en langage d'assemblage et la traduction des déclarations de variables sont abordées dans le chapitre 5.

Les chapitres 6 et 8 regroupent les principes de base de la traduction du C en langage d'assemblage. Les opérateurs ***** et **&** du C, la notion de pointeur et la traduction des accès aux variables stockées en mémoire font l'objet du chapitre 6. Les constructeurs algorithmiques du C (tels que for, while, etc), leur traduction et le concept de branchement sont présentées dans le chapitre 8.

Quelques points particuliers sont traités dans des chapitres spécifiques. Il s'agit des structures, des tableaux et de leurs liens avec les pointeurs, de la gestion des procédures et de la compilation séparée.

Le dernier chapitre explicite quelques spécificités du processeur ARM par rapport au processeur RISC fictif de référence considéré dans les chapitres précédent.

Outre le dernier chapitre, le lecteur déjà familiarisé avec le langage d'assemblage d'un autre processeur, (par exemple 680x0 ou famille 80x86), pourra se concentrer sur les chapitres 4 et 6, ainsi que 5 s'il ne maîrise pas la syntaxe GNU des directives de réservation mémoire, et enfin les instructions ldm et stm (présentées avec les procédures sans récursion),

Chapitre 1

Codage des nombres et calcul en base 2

Un ordinateur peut être vu comme une machine qui manipule des (représentations d') informations. Il s'agit souvent de réaliser un certain nombre de calculs.

La manipulation de grandeurs analogiques ou continues est malaisée. Il est diffcile de réaliser des circuits électroniques analogiques à la fois précis, rapides et insensibles aux variations de température. Les ordinateurs sont des calculateurs digitaux (ou numériques) qui travaillent sur des informations de nature discrète codées sous forme de nombres. Les circuits électroniques digitaux fonctionnant en tout ou rien offrent l'avantage d'une bonne immunité au bruit et une vitesse de calcul élevée.

Un ordinateur manipule donc des variables booléennes qui ne peuvent prendre chacune que deux valeurs : 0 ou 1. Ces deux symboles représentent la propriété vraie (1) ou fausse (0) d'une proposition logique. Mais ils correspondent aussi aux chiffres de la base 2.

Une variable booléenne dont les valeurs vrai et faux sont équiprobables encode une unité élémentaire d'information : le bit. Mais le bit est plus souvent défini comme un chiffre élémentaire (BInary digiT) dans la représentation des nombres en base deux. Les paquets de 4 bits et 8 bits sont appelés respectivement quartet et octet.

Les circuits des ordinateurs traitent donc les chiffres des entiers (écrits en binaire) comme des variables booléennes. Ils réalisent des fonctions boolénnes qui correspondent aux règles de calcul arithmétique en base 2.

Les ordinateurs sont dimensionnés pour traiter des mots (paquets de bits accolés) de largeur fixe (généralement 32 et 64 bits), ou des sous-multiples de 8 ou 16 bits. Ils savent gérer efficacement les entiers naturels (non signés) et relatifs (signés). Il existe également une norme de représentation de nombres à virgule (avec une précision limitée).

1.1 Conversion d'entiers naturels dans les bases 2 et 16

Dans une base de numération B donnée, tout entier naturel (non signé, ≥ 0) X peut être représenté par une suite de chiffres (ou digits) x_i, tous inférieurs à la base utilisée ($0 \leq x_i \leq B-1$) et tels que $X = \sum_{i=0}^{n-1} x_i * B^i$. La base est éventuellement précisée en indice à droite du dernier chiffre ou entre parenthèses. Par défaut, il s'agit de la base 10.

Le rang d'un chiffre sera également désigné sous le nom de poids en se référant à la puissance de la base par lequel il doit être multiplié. Le chiffre de poids fort x_{n-1} est écrit à gauche et celui

de poids faible x_0 à droite. Les chiffres sont 0 et 1 pour la base 2, 0 à 7 pour la base 8 (octale), 0 à 9 pour la base 10 (décimale) et 0 à 9 et A à F pour la base 16 (hexadécimale).

Il suffit d'appliquer la définition pour convertir un entier exprimé en base B en base 10. Les 0 à gauche peuvent être éliminés sans changer la valeur de l'entier naturel représenté.

$$
\begin{array}{lcllcll}
101_2 & = & 1\times 2^2+0\times 2^1+1\times 2^0 & = & 4+1 & = & 5\\
101_8 & = & 1\times 8^2+0\times 8^1+1\times 8^0 & = & 64+1 & = & 65\\
101_{10} & = & 1\times 10^2+0\times 10^1+1\times 10^0 & = & 100+1 & = & 101\\
101_{16} & = & 1\times 16^2+0\times 16^1+1\times 16^0 & = & 256+1 & = & 257\\
A4_{16} & = & 10\times 16^1+4\times 16^0 & = & 10*16+4 & = & 164\\
0023_5 & = & 0\times 5^3+0\times 5^2+2\times 5^1+3\times 5^0 & = & 10+3 & = & 13.
\end{array}
$$

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	$1G_b = 1K_b^3$

TAB. 1.1 – Chiffres hexadécimaux et principales puissances de deux

La traduction des chiffres hexadécimaux et les principales puissances de 2 sont regroupées dans le tableau 1.1. Les préfixes K,M,G indiquent une multiplication par respectivement un millier, un million et un milliard. On notera que 2^{10} vaut presque 1000 (2.4% d'erreur), d'où l'idée de définir un préfixe K_b pour les valeurs binaires. Les capacités de mémoire d'un ordinateur s'expriment toujours en préfixes binaires et nous omettrons généralement l'indice indiquant qu'il s'agit d'un préfixe binaire. 1 Koctet de mémoire signifie par défaut 1024 octets.

La conversion d'un entier de base 10 en une autre base est effectuée en divisant répétitivement l'entier par la base jusqu'à 0 et en alignant les restes des divisions successives (notés $\rightarrow$ *reste* dans l'exemple ci-dessous), du premier écrit à droite au dernier écrit à gauche.

La représentation en binaire peut aussi être obtenue en essayant de décomposer l'entier en somme de puissances de 2, chacune correspondant à un chiffre à mettre à 1. Exemple : on peut se rendre compte que $1099 = 1024(2^{10}) + 64(2^6) + 8(2^3) + 2(2^1) + 1(2^0)$. Dans la représentation en

binaire, seuls les chiffres de poids 0, 1, 3, 6 et 10 seront à 1, d'où $1099 = 10001001011_2$.

$$
\begin{array}{lcl}
116_{10} = 431_5 & \Rightarrow & 116 \div 5 = 23 \rightarrow \boxed{1} \rightsquigarrow 23 \div 5 = 4 \rightarrow \boxed{3} \rightsquigarrow 4 \div 5 = 0 \rightarrow \boxed{4} \\
171_{10} = AB_{16} & \Rightarrow & 171 \div 16 = 10 \rightarrow \boxed{11\ (\mathrm{B})} \rightsquigarrow 10 \div 16 = 0 \rightarrow \boxed{10\ (\mathrm{A})} \\
171_{10} = 253_8 & \Rightarrow & 171 \div 8 = 21 \rightarrow \boxed{3} \rightsquigarrow 21 \div 8 = 2 \rightarrow \boxed{5} \rightsquigarrow 2 \div 8 = 0 \rightarrow \boxed{2} \\
71_{10} = 1000111_2 & \Rightarrow & 71 \div 2 = 35 \rightarrow \boxed{1} \rightsquigarrow 35 \div 2 = 17 \rightarrow \boxed{1} \rightsquigarrow 17 \div 2 = 8 \rightarrow \boxed{1} \\
 & & \rightsquigarrow 8 \div 2 = 4 \rightarrow \boxed{0} \rightsquigarrow 4 \div 2 = 2 \rightarrow \boxed{0} \rightsquigarrow 2 \div 2 = 1 \rightarrow \boxed{0} \rightsquigarrow 1 \div 2 = 0 \rightarrow \boxed{1} \\
1099_{10} = 44B_{16} & \Rightarrow & 1099 \div 16 = 68 \rightarrow \boxed{11\ (\mathrm{B})} \rightsquigarrow 68 \div 16 = 4 \rightarrow \boxed{4} \rightsquigarrow 4 \div 16 = 0 \rightarrow \boxed{4}
\end{array}
$$

FIG. 1.1 – Conversion entre bases par le calcul des restes de division

L'écriture de 32 chiffres binaires côte à côte est difficile à lire et il est recommandé de grouper les bits en quartets (à partir de la droite), séparés par des espaces. Elle facilite le passage entre les formes hexadécimales et binaires : chaque quartet correspond à la traduction en binaire d'un chiffre hexadécimal de la représentation en base 16. La figure 1.1 illustre la conversion pour 1099 :

$1099_{10} =$	0	0	0	0	0	4	4	B_{16}
$1099_{10} =$	0000	0000	0000	0000	0000	0100	0100	1011_2

FIG. 1.2 – Expression binaire et hexadécimale de 1099

1.2 Additions d'entiers naturels

1.2.1 Principe de l'addition en base B

L'addition de deux nombres a et b écrits en base B est effectuée colonne par colonne des poids faibles au poids fort, donc de droite à gauche. Chaque colonne comprend un chiffre de retenue entrante, issue de la colonne précédente, qui vaut 0 ou 1, et un chiffre ($\leq B-1$) de chacun des deux opérandes.

La somme s des chiffres d'une colonne vérifie $0 \leq s \leq 2 * B - 1$ et s est un nombre codable sur deux chiffres. Celui de poids fort correspond à la retenue sortante de la colonne. Il vaut 0 si $s < B$ et 1 si $B \leq s < 2 \times B$. Celui de poids faible est le chiffre du résultat dans cette colonne et vaut s *modulo*[1] B, que nous noterons $s\%B$. La retenue sortante est prise en compte dans la colonne suivante, ce qui revient à la multiplier par la base B.

Les lignes de la figure 1.3 représentent dans l'ordre les deux opérandes, les retenues entrantes (retenues sortantes décalées d'un cran à gauche), le résultat et les retenues sortantes. La dernière retenue sortante, habituellement désignée c (carry) est encadrée.

Ainsi, dans la colonne de droite du calcul en base 10, la retenue entrante est nulle (pas de colonne précédente). La somme des chiffres de la colonne (8, 4 et 0) vaut 12, qui excède 10. Donc le chiffre du résultat dans la colonne est 12 modulo 10, soit 2 et la retenue sortante 1. Cette retenue sortante est propagée à la colonne suivante, dans laquelle elle représente l'addition d'une fois la base 10. L'addition dans la colonne suivante (5, 2 et 1 de retenue propagée) donne 8, inférieur à la base, d'où 8 pour le chiffre de résultat et pas de retenue sortante (propagation de 0 dans la colonne de gauche).

[1] Rappel : modulo désigne le reste de la division entière notée /

opérande 1		4	5	8		1	1	1	0	0	1	0	1	0
+ opérande 2		1	2	4		0	0	1	1	1	1	1	0	0
Ret décalé ←		0	1	(0)		1	1	1	1	1	0	0	0	(0)
Résultat	0	5	8	2	1	0	0	1	0	0	0	1	1	0
Ret		[0]	0	1		[1]	1	1	1	1	1	0	0	0

FIG. 1.3 – Addition en bases 10 et 2

1.2.2 Addition en base 2

La même technique est appliquée en base 2. La somme de trois chiffres 1 donne 3, soit un résultat de 1 (3 modulo 2) et une retenue sortante à 1. Deux chiffres à 1 et un à 0 donnent 2, soit une retenue sortante à 1 et un résultat nul.

Une retenue sortante finale, notée c_n ou encore c (encadrée), non nulle indique que la représentation du résultat nécessite un chiffre de plus que celle de ces opérandes. Elle correspond au chiffre à rajouter à gauche de la ligne du résultat.

Tous les entiers naturels traités par les ordinateurs sont stockés dans des contenants (registres du processeur ou emplacements de la mémoire) dont le nombre de bits n est figé. Au besoin la représentation binaire des opérandes sera complétée à gauche par des 0 pour correspondre au format du contenant. Les résultats sont obtenus à B^n (donc 2^n en base 2) près.

Une retenue finale c à 1 indique un débordement de la capacité de représentation (pour des entiers naturels) et que le résultat apparent est faux (la véritable somme devrait valoir 2^n de plus). Soit R le résultat final apparent et c la retenue finale issus de l'additionneur :
$R = (a + b)\ \%\ 2^n$, $c = (a + b)/2^n$ (division entière) et $a + b = R + c \times 2^n$.

1.2.3 Addition en hexadécimal

L'addition en base 16 fonctionne exactement de la même manière que dans les autres bases, et il y a retenue lorsque la somme dans la colonne égale ou dépasse 16_{10} (somme de la colonne supérieure à F). En base 16, on utilise la table d'addition (1.4) au lieu de la table d'addition décimale habituelle. Si on trouve dans une colonne les chiffes A, 9 et 1 de retenue entrante, la somme vaut 0x13+1, soit 0x14 : résultat 4 et une retenue sortante.

1.2.4 Addition multilongueur

Il arrivent que les ordinateurs travaillent sur des entiers codés sur un nombre de bits excédant la largeur du circuit de calcul. Les entiers sont alors découpés en tranches de bits de la largeur de l'additionneur. Les additions sont réalisées tranche par tranche, de la droite vers la gauche.

La retenue sortante de l'addition d'une tranche est utilisée comme retenue entrante pour l'addition de la tranche suivante. Seule la première addition utilise une retenue entrante initiale nulle. Si l'on réduit les tranches à un seul bit, on retrouve l'algorithme séquentiel classique d'addition colonne par colonne.

+	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

FIG. 1.4 – Table d'addition des chiffres hexadécimaux

1.3 Soustraction d'entiers naturels

Soit un entier naturel Y à soustraire d'un entier X. Chaque colonne comprend dans l'ordre un chiffre de X, un chiffre de Y, un emprunt entrant propagé par la colonne précédente, un chiffre du résultat apparent et un emprunt sortant.

opérande 1 ($X = \sum x_i$)		4	1	4		1	1	0	0	1	1	1	1	0
opérande 2 ($Y = \sum y_i$)	-	2	4	3	-	0	1	1	1	1	0	0	1	1
Emprunt décalé ←		1	0	(0)		1	1	1	0	0	0	1	1	(0)
Résultat ($R = \sum res_i$)		1	7	1		0	1	0	1	0	1	0	1	1
Emprunt (e_i)		0	1	0		0	1	1	1	0	0	0	1	1

La soustraction est effectuée de droite à gauche, tout comme l'addition. L'emprunt entrant est ajouté au chiffre de Y. Leur somme est otée du chiffre de X et cette soustraction donne le chiffre du résultat apparent. Il n'y a pas d'emprunt sortant ($e_i = 0$) lorsque la soustraction est possible ($Y_i + e_i \leq X$). Dans le cas contraire, la base B est préalablement ajoutée au chiffre de X et un emprunt ($e_{i+1} = 1$) est propagé à la colonne suivante.

Exemple de la colonne [2] numéro 5 dans la première soustraction : $x_5 = e_5 = 0$ et $y_5 = 1$, d'où 1 à oter de 0 : $res_5 = 1$ et $e_6 = 1$. En colonne 6 $x_6 = 0$ et $y_6 = e_6 = 1$, d'où $(1 + 1)$ à oter de 0 : $res_6 = 0$ et $e_7 = 1$.

Remarque : un emprunt final (sortant de la dernière colonne de gauche et noté e_n ou e) nul indique que la soustraction peut être réalisée (deuxième opérande inférieur ou égal au premier).

[2] Rappel : les colonnes sont numérotées de droite à gauche à partir de 0

Si nous réalisions la soustraction inversée (243 - 414), un emprunt final non nul indiquerait que celle-ci n'est pas réalisable (deuxième opérande supérieur au premier). Le résultat apparent correspond à l'opération réalisée en ajoutant B^n au premier opérande, autrement dit en supposant l'existence d'un 1 supplémentaire à gauche du premier opérande.

opérande 1 ($X = \sum x_i$)		2	4	3		0	1	1	1	1	0	0	1	1
opérande 2 ($Y = \sum x_i$)	-	4	1	4	-	1	1	0	0	1	1	1	1	0
Emprunt décalé ←		0	1	(0)		0	0	0	1	1	1	0	0	(0)
Résultat ($R = \sum res_i$)		8	2	9		1	0	1	0	1	0	1	0	1
Emprunt (e_i)		$\boxed{1}$	0	1		$\boxed{1}$	0	0	0	1	1	1	0	0

L'emprunt final indique que $101010101 = 1011110011 - 110011110$, soit 341=755 - 414[3]. Dans la version décimale, il signifie que $829 = 1243 - 414$[4].

La soustraction multilongueur est réalisée de la même manière que pour l'addition, en propageant les emprunts au lieu des retenues.

1.4 Notion de complément à 1 et à 2

Rappelons une formule utile pour la compréhension des calculs arithmétiques en base 2 : $\sum_{i=0}^{n-1} a^i = \frac{a^n-1}{a-1}$. Cette équation est aisément démontrable en multipliant $(a-1)$ par $(1+a+a^2+\ldots+a^{n-1})$. Pour $a = 2$, nous en déduisons que $\sum_{i=0}^{n-1} 2^i = 2^n - 1$.

Soient n le nombre de chiffres stockables dans un contenant (registre ou mot mémoire) et soit X l'entier $\sum_{i=0}^{n-1} x_i B^i$. Notons $\bar{}_B$ l'opération de complémentation qui à un chiffre x_i associe le chiffre $x_{iB} = B - 1 - x_i$, B étant la base de numération : $\overline{3}_{10} = 6$, $\overline{6}_{10} = 3$, $\overline{0}_3 = 2$, $\overline{2}_3 = 0$ et $\overline{1}_3 = 1$, Par définition, nous avons $x_i + \overline{x}_{iB} = B - 1$.

La base 2 exhibe une propriété très utile pour la réalisation efficace de circuits de calculs. La table de vérité de l'opérateur $\bar{}_2$ défini sur les chiffres binaires 0 et 1 se confond avec celle de l'opérateur $\bar{}$ défini sur les booléens 0 et 1 : $\overline{0}_2 = 2 - 1 - 0 = 1$ et $\overline{1}_2 = 2 - 1 - 1 = 0$. Dans la suite du document, lorsque la base de complémentation ne sera pas précisée, il s'agira par convention de la base 2 : $\overline{X}$=$\overline{X}_2$.

Le complément à $B-1$ de X est l'entier $\overline{X}_B = \sum_{i=0}^{n-1} \overline{x}_{iB} B^i$. L'entier $\overline{X}^B = \overline{X}_B + 1$ est appelé complément à B de X. En base 2, $\overline{X} = \overline{X}_2$ et $\overline{X}^2 = \overline{X}_2 + 1$ sont donc respectivement les complément à 1 et complément à 2 de X.

Les ordinateurs exploitent la propriété suivante de la somme $X + \overline{X}_B + 1$ pour la réalisation des soustractions.

[3]avec 755=243 + 512

[4]avec 1243=243 + 1000

$$
\begin{aligned}
X + \overline{X}_B &= \sum_{i=0}^{n-1} (x_i + \overline{x}_{iB}) B^i && (1.1)\\
&= (B-1) \sum_{i=0}^{n-1} B^i && (1.2)\\
&= (B-1) \frac{B^n - 1}{B-1} = B^n - 1 && (1.3)\\
X + \overline{X}^2_B &= X + \overline{X}_B + 1 = B^n && (1.4)\\
X + \overline{X}^2 &= X + \overline{X}_2 + 1 = 2^n && (1.5)
\end{aligned}
$$

Ce résultat indique que le complément à 2 (respectivement complément à 1) de X est le nombre qu'il faut ajouter à X pour obtenir 2^n (respectivement $2^n - 1$). On pourrait aussi les appeler complément à 2^n et complément à $2^n - 1$: $\overline{X}^2 = 2^n - X$ et $\overline{X} = 2^n - 1 - X$.

Montrons à titre d'exemple comment calculer $\overline{X}^2 = \overline{X} + 1$ pour $X = 124$ pour $n = 8$ bits ($124_{10} = 0111\ 1100_2$ et $\overline{124} = 10000011$). Une première méthode exploite la propriété $X + \overline{X}^2 = 2^n$. D'où $\overline{124}^2 = 2^8 - 124 = 256 - 124 = 132 = 132_{10} = 1000\ 0100_2$. Il est également possible de poser l'addition $\overline{X} + 1$ (figure).

	1	0	0	0	0	0	1	1	$\overline{124}$
+	0	0	0	0	0	0	0	1	1
+	0	0	0	0	0	1	1	(0)	Retenues
	1	0	0	0	0	1	0	0	$\overline{124}^2$

Il existe une méthode plus rapide : parcourir la représentation binaire de 124 de droite à gauche, conserver les chiffres de droite jusqu'au premier 1 inclus et inverser les autres. Pour 124, les 3 chiffres de droite 100 sont conservés et les chiffres 0111 1 inversés en 1000 0.

Notons au passage qu'un entier est le complément à 2 de son complément à 2 et que sur n bits, 2^{n-1} est son propre complément à 2 : $\left(\overline{\overline{X}^2}\right)^2 = 2^n - (2^n - X) = X$ et $\overline{2^{n-1}}^2 = 2^n - 2^{n-1} = 2^{n-1}$.

1.5 Soustraction par addition du complément à 2

Rappelons que les ordinateurs travaillant sur n bits réalisent les additions à modulo 2^n près. Les ordinateurs effectuent en base 2 la soustraction $X - Y$ via un additionneur calculant $X + \overline{Y}^2$. Additionner le complément à 2 de Y revient en fait à soustraire Y (rappel : en travaillant sur n chiffres, toutes les additions sont réalisées à modulo 2^n près).

$$X + \overline{Y} + 1 = \sum_{i=0}^{n-1} (x_i + 1 - y_i)\ 2^i + 1 \quad (1.6)$$

$$= \sum_{i=0}^{n-1} 2^i + \sum_{i=0}^{n-1} (x_i - y_i)\ 2^i + 1 \quad (1.7)$$

$$= 2^n - 1 + X - Y + 1 = X - Y + 2^n \quad (1.8)$$

$$(X + \overline{Y} + 1)\ \%\ 2^n = (2^n + X - Y)\ \%\ 2^n = X - Y \quad (1.9)$$

La figure suivante illustre les deux méthodes (soustraction normale à gauche et soustraction par addition du complément à 2 à droite) pour retancher 134 de 243.

243		1	1	1	1	0	0	1	1		1	1	1	1	0	0	1	1	243
134	-	1	0	0	0	0	1	1	0	+	0	1	1	1	1	0	0	1	$\overline{134}$
Emp	0	0	0	0	1	1	0	0	(0)	+	1	1	1	1	0	0	1	1	(1)	Ret ($\overline{Emp}$)
	↓										↓								
109	0	0	1	1	0	1	1	0	1		1	0	1	1	0	1	1	0	1	109

FIG. 1.5 – Soustraction normale à gauche et par addition du complément à 2 à droite

Les lignes du premier opérande et celle du résultat sont naturellement identiques dans les deux versions. La deuxième ligne correspond à gauche au deuxième opérande (134) et à droite à son complément à 1 ($\overline{134}$).

La troisième ligne correspond à droite aux retenues entrantes pour l'addition (le +1 est obtenu en utilisant une retenue entrante initiale non nulle dans la colonne de droite) et à gauche aux emprunts entrants pour la soustraction classique. Les retenues dans l'addition du complément à 2 de Y présentent la particularité d'être exactement égales au complément des emprunts dans la soustraction normale.

Cette propriété s'applique en particulier à l'emprunt final e et à la retenue finale c : $e = \overline{c}$. Lors de la réalisation d'une soustraction par addition du complément à 2, une retenue finale nulle ($c = \overline{e} = 0 \Longrightarrow e = 1$) signifie que la soustraction est impossible ($X < Y$), alors qu'une retenue finale à 1 ($c = \overline{e} = 1 \Longrightarrow e = 0$) indique que le résultat est correct et la soustraction possible.

1.6 Interprétation des indicateurs C et Z

Les instructions arithmétiques peuvent positionner les indicateurs booléens Z et C, stockés dans un registre spécial dit d'état, du processeur. L'ensemble des indicateurs est quelquefois dénommé code condition.

L'indicateur de résultat nul est Z (Zéro). Il est égal au produit du complément des chiffres (interprétés comme des booléens) du résultat fourni par l'additionneur : $Z = \prod_{i=0}^{n-1} \overline{res_i}$. On a $Z = 1$ si et seulement si le résultat apparent est l'entier 0 (tous les res_i sont à 0).

Il existe deux interprétations possibles de C (Carry) selon les processeurs. La famille ARM le définit comme la valeur de la retenue sortante c de l'additionneur. L'autre convention utilisée entre

autres par les SPARC et 680x0, le définit comme un indicateur de débordement en arithmétique sur les entiers naturels.

Après une addition sur n bits, C=1 indique un débordement : le vrai résultat n'est pas représentable sur n bits et qu'il faudrait ajouter 2^n au résultat apparent.

La différence porte sur la soustraction (toujours réalisée par addition du complément à 2) : C représente le complément de l'emprunt dans le premier cas (ARM) et l'emprunt lui-même dans l'autre cas (SPARC).

Les comparaisons sont des soustractions sans stockage du résultat apparent, dont le seul effet est de positionner les indicateurs pour déterminer si une expression conditionnelle est vraie ou fausse.

x,y : unsigned	SPARC		ARM	
condition	expression symbolique	bool	expression symbolique	bool
$x = y$	E Z	Z	EQ	Z
$x \neq y$	NE NZ	$\overline{Z}$	NE	$\overline{Z}$
$x < y$	LU (Less Unsigned) CS (Carry set)	C	LO (Lower) CC (Carry Clear)	$\overline{C}$
$x \leq y$	LEU (Less or Equal)	$C + Z$	LS (Lower or same)	$\overline{C} + Z$
$x > y$	GU (Greather)	$\overline{C}.\overline{Z} = \overline{C + Z}$	HI (Higher)	$C.\overline{Z} = \overline{\overline{C} + Z}$
$x \geq y$	GEU (Greather or Equal) CC (Carry Cleared)	$\overline{C}$	HS (Higher or same) CS (Carry Set)	C

TAB. 1.2 – Table des conditions pour entiers naturels après calcul de x-y

La construction **si (rg<rd) sauter à ...** est traduite en langage machine par une instruction de branchement conditionnel précédée d'une instruction de comparaison qui essaie de retrancher rd de rg. Le saut aura lieu si et seulement si les valeurs de Z et C correspondent à la condition $rg < rd$.

Cette condition correspond au seul cas dans lequel la soustraction n'est pas réalisable parce que $rd > rg$, soit $C_{SPARC} = e = 1$ et $C_{ARM} = \overline{(e)} = 0$. La ligne $x < y$ du tableau des conditions ARM montre qu'il faut utiliser l'instruction de branchement conditionnel blo (bcc est un synonyme) qui effectue un saut si $\overline{C}$ (C vaut 0).

Pour une condition de type inférieur ou égal, il suffit d'ajouter le cas d'égalité ($Z = 1$) à l'expression booléenne. Pour une condition strictement supérieur, il faut que le soustraction soit possible (C à 0) et que le résultat ne soit pas nul ($Z = 0$).

1.7 Opérations sur les vecteurs de bits

Certaines opérations considèrent les représentations des entiers comme des collections de chiffres binaires ou de booléens.

1.7.1 Extension et réduction de format

L'extension de format consiste à passer d'une représentation d'un entier naturel en binaire sur m bits à une représentation sur $n > m$ bits. Il suffit pour celà d'ajouter $n - m$ chiffres 0 à gauche (poids forts), ce qui ne modifie pas la valeur de l'entier représenté. L'extension de format est par exemple utilisée lors d'un transfert d'un entier stocké en mémoire sur 8 ou 16 bits vers un registre à 32 bits.

L'opération inverse de réduction tronque la représentation en éliminant les $n - m$ chiffres de gauche. Elle revient à calculer la valeur *entier modulo* 2^n. Elle est notamment utilisée lors du rangement du contenu d'un registre à 32 bits dans une variable mémoire sur 8 ou 16 bits. La valeur de l'entier est conservée si et seulement si ses $n - m$ chiffres de gauche sont nuls : l'entier est alors représentable sur m bits (valeur inférieure à 2^m).

1.7.2 Décalages logiques et rotations

Le décalage logique de d bits à droite supprime les d chiffres de droite et rajoute d chiffres 0 à gauche, ce qui revient à diviser l'entier par 2^d.

Le décalage à gauche élimine au contraire d chiffres de poids fort et rajoute d chiffres 0 à droite. Il correspond à une multiplication par 2^d (en l'absence de débordement, autrement dit si les d chiffres de poids forts du nombre de départ sont des 0).

La sommation de deux décalages logiques respectivement de d bits à droite et $n-d$ bits à gauche correspond à une rotation à droite qui déplace d chiffres de l'extrémité droite vers l'extrémité gauche. Réciproquement, une opération de rotation à gauche déplace les chiffres de poids fort (gauche) en poids faible (droite). Une rotation ne correspond à aucune opération arithmétique simple. Il existe enfin des rotations sur $n + 1$ bits au travers de l'indicateur C considéré comme un bit supplémentaire de rang n de l'entier.

1.7.3 Opérations booléennes bit à bit

Les chiffres de même rang de deux entiers codés sur n bits peuvent être considérés comme n paires de booléens sur lesquelles il est possible d'appliquer les opérateurs booléeens : et , ou, ou exclusif.

Dans chaque colonne, le chiffre du résultat est obtenu en appliquant l'opération booloéenne sur les chiffres des deux opérandes.

$$\begin{array}{rcccc} \&_{bb} & 1 & 0 & 1 & 0 \\ & 1 & 1 & 0 & 0 \\ \hline & 1 & 0 & 0 & 0 \end{array} \qquad \begin{array}{rcccc} +_{bb} & 1 & 0 & 1 & 0 \\ & 1 & 1 & 0 & 0 \\ \hline & 1 & 1 & 1 & 0 \end{array} \qquad \begin{array}{rcccc} \oplus_{bb} & 1 & 0 & 1 & 0 \\ & 1 & 1 & 0 & 0 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

FIG. 1.6 – Et, ou et ou exclusif bit à bit sur 4 bits

Dans la même catégorie, mais agissant sur un seul entier, citons l'opérateur de complémentation qui inverse tous les chiffres de l'entier : cet opérateur calcule le complément à 1 de l'entier.

Remarque : le complément à 1 équivaut à un ou exclusif ($x \oplus 1 = \overline{x}$) avec un nombre tel que tous ses chiffres binaires sont à 1. Ce nombre est l'entier naturel $2^n - 1$ ou (en anticipant sur le paragraphe consacré à l'arithmétique signée) l'entier relatif -1.

1.7.4 Manipulation de champs de bits

Les décalages et les opérations bit à bit permettent de manipuler des champs de bits à l'intérieur des entiers. Ce type d'opération sera décrite dans le chapitre 2 en utilisant la syntaxe du langage C.

1.7.5 Extraction de la parité

Par définition de la représentation en binaire, un entier exprimé en binaire est pair si et seulement si son chiffre de poids faible (rang 0) est nul. Le ET bit à bit entre l'entier x et la constante entière 1 est donc nul si et seulement si x est pair.

1.7.6 Comparaison avec 2^r

Un entier x est supérieur ou égal à 2^r si et seulement si au moins un de ses bits de rang supérieur ou égal à r est non nul. Autrement dit, le décalage de r bits à droite de x donne un résultat non nul si et seulement si $x \geq 2^r$.

1.7.7 Modulo 2^r

Par définition le reste de la division de x par 2^r (x *modulo* 2^r) est l'entier dont les $n + 1 - r$ premiers chiffres sont à 0 et les $r - 1$ chiffres de poids faibles identiques à ceux de x.

L'expression x *modulo* 2^r est calculable par un ET bit à bit entre x et $2^r - 1$ (dont la représentation contient $r - 1$ bits à 1 en poids faible et des bits à 0 en poids forts.

1.7.8 Nombre de 0 à gauche et logarithme binaire

Certains processeurs (sont la famille ARM) sont dotés d'une instruction clz (count leading zeros) qui détermine le nombre de 0 à gauche (en poids forts) d'un entier (écrit en binaire). Soit r la valeur retournée par clz(x) pour un entier x non nul.

L'expression $n - 1 - r$ (n étant le nombre de bits, soit 32 pour le ARM) est la position du chiffre à 1 de rang le plus élevé. Elle varie comme la partie entière de $log_2()x$.

1.8 Nombres entiers naturels et relatifs et nombres à virgule

Un contenant à n bits ne peut prendre que 2^n valeurs différentes. A ce paquet de n bits sera associée une valeur numérique selon une convention d'interprétation de contenu. Le tableau 1.8 donne les intervalles de valeurs entières codables pour différentes tailles de contenant.

n	Entiers naturels			Entiers signés		
8	0	à	255	-128	à	+127
16	0	à	65535 (64K-1)	-32768 (-32K)	à	+32767 (32K-1)
32	0	à	4294967295 (4G-1)	-2147483648 (-2G)	à	+2147483647 (2G-1)
64	0	à	$\tilde{}1,8 \times 10^{19}$	$\tilde{} - 9 \times 10^{18}$	à	$\tilde{} + 9 \times 10^{18}$

FIG. 1.7 – Intervalles des entiers représentables selon le nombre de bits

La convention la plus simple consiste à interpréter le contenu sur n bits comme un entier naturel en utilisant la convention de représentation des entiers en base 2. L'intervalle des entiers naturels représentables va de 0 à $2^n - 1$ (le premier entier nécessitant $n + 1$ chiffres est 2^n).

Le contenu aussi peut être interprété comme la valeur d'un entier relatif (signé) représenté en binaire selon une convention à définir. Si l'on choisit d'avoir un codage unique de la valeur nulle, on obtient un intervalle de valeurs représentables allant de -2^{n-1} à $2^{n-1} - 1$.

Il est enfin possible d'interpréter le contenu comme un triplet {signe, mantisse ou partie fractionnaire, exposant} représentant un nombre à virgule selon une norme telle que ANSI/IEEE 754 de 1985.

A titre d'illustration, sur 32 bits, cette norme interprète le bit 31 comme un bit de signe s, et les paquets de bits 23 à 30 d'une part et 0 à 22 d'autre part, comme des entiers naturels codant respectivement l'exposant e et la partie fractionnaire f. Elle associe au triplet {s,e,f} ainsi défini la valeur $(-1)^s \times 2^{e-127} \times 1.f$.

31 s	30 exposant e 23	22 mantisse f 0

FIG. 1.8 – Format de nombres à virgule flottante

1.9 Convention de représentation des entiers relatifs

1.9.1 Signe et valeur absolue

Une convention imaginable pour la représentation des entiers signés dans un contenant serait de considérer le bit de rang $n - 1$ comme un bit de signe (par exemple 0 pour + et 1 pour -) et les bits 0 à $n - 2$ comme un entier naturel représentant la valeur absolue du nombre.

Cette technique de représentation a deux inconvénients : elle oblige à prévoir un cas spécial pour le traitement des entiers signés dans les circuits de calcul, et surtout elle exhibe deux représentations de la valeur nulle : $+0$ et -0, ce qui complique nettement la réalisation des comparaisons.

La représentation en signe et valeur absolue ne subsiste dans les ordinateurs actuels que dans la représentation de la mantisse des nombres à virgule.

1.9.2 Représentation en complément à deux

La quasi-totalité des ordinateurs modernes utilise la méthode du complément à deux pour représenter les entiers signés. Dans cette convention, le chiffre de poids fort représente le signe de l'entier (1 indique $x < 0$ et 0 indique $x \geq 0$).

La suite de chiffres $x_{n-1}x_{n-2}\ldots x_1x_0$, qui représentait jusqu'à présent l'entier naturel $X_{non\_signé} = \sum_{i=0}^{n-1} x_i B^i$ est interprétée dans cette convention de représentation comme l'entier signé $X_{signé} = -\ x_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} x_i B^i$.

Cette convention permet de représenter sur n bits l'intervalle d'entiers signés $[-2^{n-1}, +2^{n-1}-1]$.

Soit $y = \sum_{i=0}^{n-2} y_i 2^i$ et $-z = \overline{y}^2 = 2^n - y$.

Un contenu d'une variable entière de la forme $0y_{n-2}y_{n-3}\dots y_1y_0$ sera toujours interprété comme l'entier y, que la variable soit de type naturel ou signé.

Un contenu de la forme $1y_{n-2}y_{n-3}\dots y_1y_0$ sera interprété comme l'entier $Y = 2^{n-1} + y$ si variable est de type entier naturel et comme l'entier relatif négatif $z = -2^{n-1} + y$, si la variable est de type entier relatif.

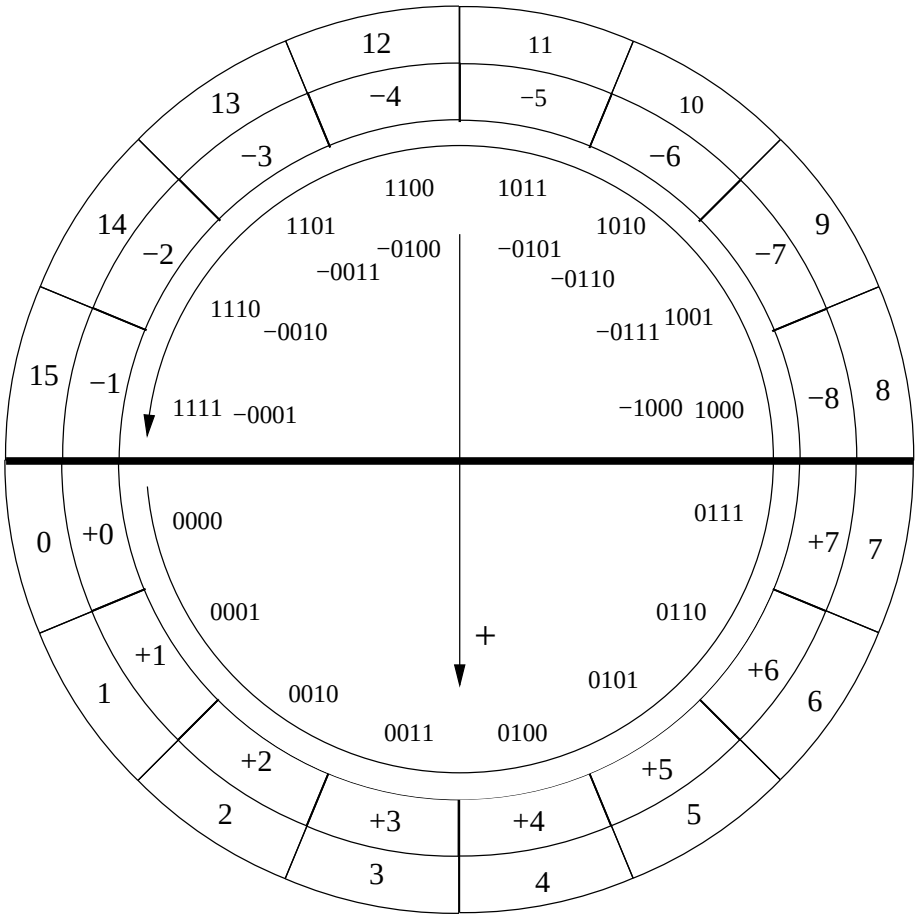


FIG. 1.9 – Représentation d'entiers naturels et signés sur 4 bits

Sur la figure 1.9, chaque entier (codé ici sur 4 bits) peut être représenté par une rotation dans le sens horaire pour les entiers signés négatifs et dans le sens trigonométrique pour les autres (entiers naturels ou entiers signés non négatifs).

Exemple sur 4 bits (figure 1.9) : 1100 peut représenter soit l'entier signé -4 (rotation horaire d'un quart de tour sur cercle intérieur des entiers signés), soit l'entier naturel 12 (rotation trigonométrique sur le cercle extérieur des entiers naturels). De même, 0011 peut représenter soit l'entier naturel 3, soit l'entier signé +3.

Cette méthode de représentation des entiers signés par le complément à 2 présente un gros avantage : les opérations d'addition et de soustractions sont posées de la même manière quelque soit la nature, signée ou non, des entiers manipulés. Seule l'interprétation des valeurs entières associées change.

1.10 Opérations sur les entiers relatifs et débordements

La figure 1.10 illustre une addition entre deux variables sur 4 bits contenant respectivement 1110 et 1100. Au milieu, les opérations effectuées par l'additionneur sur les chiffres binaires.

La gauche de la figure montre l'interprétation classique en arithmétique non signée : la somme de 14 et 12 donne un résultat apparent de 10 avec une retenue finale à 1 indiquant un débordement (et qu'il faudrait ajouter 16 au résultat apparent pour obtenir un résultat exact).

La droite de la figure montre l'interprétation en arithmétique signée. Le bit de fort à 1 des opérandes indique qu'il s'agit d'entiers négatifs (dont la valeur absolue peut être trouvée en prenant le complément à deux) : -2 et -4. De même pour le résultat apparent, correct, interprété comme -6.

Il n'y a pas débordement pour cette opération en arithmétique signée. En effet, le résultat -6 appartient effectivement à l'intervalle des entiers représentables sur 4 bits : $[-2^3, +2^3 - 1]$. Notons que les deux dernières retenues sont égales.

Rappelons que dans les deux cas, l'additionneur a effectué le même travail sur les mêmes booléens et produit le même résultat apparent. Seule la grille d'interprétation de la représentation en binaire change.

Interprétation	naturels		binaires	signés
opérande 1	1 4		1 1 1 0	- 2
opérande 2	1 2	+	1 1 0 0	- 4
Résultat	1 0	1	[1] 0 1 0	- 6
retenues		↑ ↖	[1] [1] 0 0	

FIG. 1.10 – Addition entière naturelle et signée

La technique de soustraction en arithmétique signée est la même que pour les entiers naturels : par addition du complément à 2, ce qui revient en arithmétique signée à ajouter l'opposé du deuxième opérande. Seule l'interprétation des indicateurs de débordement change.

Pour évaluer les conditions portant sur la valeur relative de deux entiers signés $S1$ et $S2$, on procède comme pour les entiers naturels en calculant $S1 - S2$.

1.10.1 Indicateur Z

L'indicateur Z présenté dans le chapitre sur le calcul sur les entiers naturels reste utilisable pour des variables de type signé : la valeur nulle est codée de la même manière dans les deux conventions. $Z == 1$ indique un résultat nul, donc l'égalité des deux entiers dans le cas d'une soustraction.

1.10.2 Indicateur de signe N

Nous avons vu que le signe d'un entier est codé dans son bit de poids fort. Le bit de poids fort du résultat apparent (encadré sur la figure 1.10) fourni par l'unité arithmétique est stocké dans

l'indicateur N (Négatif).

$N == 0$ correspond à un résultat apparent positif ou nul. Après une soustraction $S1 - s2$, $N == 0$ indique que d'après le résultat apparent (≥ 0), nous avons $S1 \geq S2$. De $N == 1$ on tirerait au contraire la conclusion que $S1 < S2$ du résultat apparent (<0) de la soustraction.

1.10.3 Indicateur de débordement signé V

La figure 1.9 permet de visualiser le phénomène de débordement.

Pour la représentation d'entiers naturels, l'intervalle des entiers représentables correspond à tout le cercle. Il y aura débordement (signalé par une retenue finale c à 1) si la somme des deux rotations égale ou excède un tour complet.

Dans la représentation d'entiers relatifs, l'intervalle des valeurs absolues disponibles est réduit de moitié. La limite de rotation avant débordement est d'un demi-tour seulement : au delà la valeur absolue du vrai résultat excède la capacité de représentation et le signe du résultat apparent est faux. Ce type d'erreur est signalé par l'indicateur de débordement signé V (oVerflow : l'initiale O n'a pas été retenue pour éviter des confusions avec zéro). $V == 1$ indique un débordement et $V == 0$ un résultat correct.

La somme de 0100 (1/4 de tour) et 0100 (1/4 tour) donne 1000 (1/2 de tour) : le résultat apparent de la somme des entiers naturels 4 et 4 est 8 (correct : la rotation est inférieure à un tour complet) et le résultat apparent de la somme des entiers signés +4 et +4 est -8 (faux à 2^4 près : rotation atteignant un demi-tour). Dans cet exemple, il y a débordement uniquement en arithmétique signée (V == 1) et pas en arithmétique naturelle (C=0).

$$\begin{array}{rrrr} + & 4 & 0\,1\,0\,0 & +4 \\ & 4 & 0\,1\,0\,0 & +4 \\ & C=\underline{0} & \underline{1}\,0\,0\,0 & V=1 \\ & & \overline{\qquad\qquad} & \\ & 8 & 1\,0\,0\,0 & -8 \end{array}$$

La valeur absolue de la somme de deux entiers relatifs de signes opposés (exemples sur 8 bits : 16 + -37, 72 + -37) est toujours inférieure ou égale à celle des deux opérandes. Le résultat est toujours représentable et il ne peut y avoir de débordement (V == 0). L'interprétation des mêmes opérations en entiers naturels peut aboutir à un débordement non signé ($C == 1$ pour 72+219) ou pas ($C == 0$ pour 16+219).

$$\begin{array}{rrrr} + & 16 & 0\,0\,0\,1\,0\,0\,0\,0 & +16 \\ & 219 & 1\,1\,0\,1\,1\,0\,1\,1 & -37 \\ & C=\underline{0} & \underline{0}\,0\,1\,0\,0\,0\,0\,0 & V=0 \\ & & \overline{\qquad\qquad\qquad} & \\ & 235 & 1\,1\,1\,0\,1\,0\,1\,1 & -21 \end{array} \qquad \begin{array}{rrrr} + & 72 & 0\,1\,0\,0\,1\,0\,0\,0 & 72 \\ & 219 & 1\,1\,0\,1\,1\,0\,1\,1 & -37 \\ & C=\underline{1} & \underline{1}\,0\,1\,1\,0\,0\,0\,0 & V=0 \\ & & \overline{\qquad\qquad\qquad} & \\ & 35 & 0\,0\,1\,0\,0\,0\,1\,1 & +35 \end{array}$$

Si l'on additionne la constante -2^{n-1} à elle-même, on fait exactement un tour complet. Le résultat obtenu est 0 et il y a débordement signé.

La valeur absolue de la somme de deux entiers relatifs de même signe est supérieure ou égale à celle des opérandes. Excluons l'addition de l'entier -2^{n-1} à lui-même : la valeur absolue de la somme est strictement inférieure à 2^n . Le résultat vrai appartient à l'intervalle $[-(2^n-1), +2^n-1]$. Il ne sera représentable sur n bits que s'il appartient à l'intervalle $[-2^{n-1}, +2^{n-1}-1]$. Dans le cas

contraire, il y a aura débordement et le signe N du résultat apparent sera faux.

Une étude de cas d'opérandes de même signe (-72 + -37, -37 + -37, +72 + +37) llustre le fait qu'il n'y a pas de débordement en arithmétique signée lorsque les deux dernières retenues c_{n-1} et c_n sont égales. Il en va de même pour l'addition d'un entier et de son opposé (+72 + -72).

+	184	1 0 1 1 1 0 0 0	−72	+	219	1 1 0 1 1 0 1 1	−37
	219	1 1 0 1 1 0 1 1	−37		219	1 1 0 1 1 0 1 1	−37
	$C = \underline{1}$	1 1 1 1 0 0 0 0	$V = 0$		$C = \underline{1}$	$\underline{1}$ 0 1 1 0 1 1 0	$V = 0$
	147	1 0 0 1 0 0 1 1	−109		182	1 0 1 1 0 1 1 0	−74

+	72	0 1 0 0 1 0 0 0	+72	+	72	0 1 0 0 1 0 0 0	+72
	37	0 0 1 0 0 1 0 1	+37		184	1 0 1 1 1 0 0 0	−72
	$C = \underline{0}$	$\underline{0}$ 0 0 0 0 0 0 0	$V = 0$		$C = \underline{1}$	$\underline{1}$ 1 1 1 0 0 0 0	$V = 0$
	109	0 1 1 0 1 1 0 1	+109		0	0 0 0 0 0 0 0 0	+0

L'indicateur V peut être défini à partir des bits de poids fort (signe) des opérandes x et y et du résultat r selon l'expression booléenene $V = \overline{x_{n-1}}.\overline{y_{n-1}}.r_{n-1} + x_{n-1}.y_{n-1}.\overline{r_{n-1}}$ (+ indique le ou booléen) Cette expression signifie que $V == 1$ uniquement quand les deux opérandes sont de même signe et le résultat apparent de signe opposé.

+	72	0 1 0 0 1 0 0 0	+72	+	184	1 0 1 1 1 0 0 0	−72
	72	0 1 0 0 1 0 0 0	+72		184	1 0 1 1 1 0 0 0	−72
	$C = \underline{0}$	$\underline{1}$ 1 0 1 0 0 0 0	$V = 1$		$C = \underline{1}$	$\underline{0}$ 1 1 1 0 0 0 0	$V = 1$
	144	1 0 0 1 0 0 0 0	−112		112	0 1 1 1 0 0 0 0	+112

Le premier cas de débordement correspond à des opérandes positifs ou nuls et un résultat apparent négatif (72 + 72). Dans la dernière colonne, les opérandes sont 0 et le résultat 1. Ceci implique que la retenue entrante c_{n-1} est 1 et que la retenue finale c_n est 0.

Examinons l'autre cas de débordement : deux entiers négatifs donnent un résultat apparent positif (-72 + -72). Dans la dernière colonne, nous avons deux opérandes à 1 et un résultat à 0, d'où obligatoirement $c_{n-1} = 0$ et $c_n = 1$.

On peut définir une autre expression de V en fonction des deux dernières retenues : V est vrai uniquement si celles-ci sont différentes : $V = c_{n-1} \oplus c_n = \overline{c_{n-1}}.c_n + c_{n-1}.\overline{c_n}$

1.11 Résumé sur les indicateurs et les débordements

Rappelons que le circuit de calcul effectue sur paquets de n bits des additions ou des soustractions (par addition du complément à 2) et fournit pour chaque opération réalisée quatre indicateurs : Z, C, N et V.

Seule l'interprétation du sens de l'addition ou la soustraction réalisée sur les paquets de n bits change selon que l'on considère que ces paquets de n bits représentent des entiers naturels ou des

entiers relatifs.

Le programmeur utilise donc la même instruction d'addition et la même instruction de soustraction pour les deux types d'entiers. Après une comparaison, on utilise en revanche deux sortes de branchements conditionnels différents : l'un testant Z et C pour les entiers naturels et l'autre testant Z, N et V pour les entiers relatifs.

1. Une retenue finale c égale à 1 signale un débordement en arithmétique sur une addition d'entiers naturels : elle indique que la valeur absolue du vrai résultat dépasse $2^n - 1$ et n'est pas représentable sur n bits.
2. Une retenue finale c égale à 0 signale une erreur sur une soustraction d'entiers naturels (réalisée par addition du complément à 2) : elle indique que le dernier emprunt est à 1 et que la soustraction n'est réalisable qu'en ajoutant 2^n au premier opérande.
3. Un indicateur V à 1 indique un débordement lors d'une addition ou d'une soustraction sur des entiers relatifs. Celà se produit si et seulement si les deux opérandes sont de même signe et le résultat apparent de signe opposé. La limite de valeur absolue d'entiers représentables sur n bits est 2^{n-1}.
4. L'indicateur N est le signe du résultat en arithmétique sur les entiers relatifs. Il est faux (et le résultat apparent est faux à 2^n près) si V=1.
5. Z == 1 si et seulement si le résultat apparent est nul

1.12 Comparaisons d'entiers relatifs avec Z, N et V

Après une soustraction $x - y$, il est possible d'évaluer des conditions portant sur les valeurs relatives de deux entiers signés.

Les conditions d'égalité et de non égalité sont les mêmes pour les entiers naturels et signés.

La condition $x < y$ est vraie si le résultat vrai est négatif. En l'absence de débordement ($V = 0$), c'est le cas si le signe du résultat apparent est négatif ($N = 1$). En cas de débordement ($V = 1$), on sait que le résultat apparent est faux, y compris son signe, la condition est alors vraie lorsque $N = 0$. D'où l'expression de la condition $N \oplus V = N.\overline{V} + \overline{N}.V$.

La condition inverse $x \geq y$, incluant le cas d'égalité est naturellement associée à l'expression complémentaire $\overline{N \oplus V} = N.V + \overline{N}.\overline{V}$. Pour la condition strictement supérieur, il faut naturellement éliminer le cas d'un résultat nul, d'où le produit avec la condition complémentaire $\overline{Z}$.

Pour la condition $x \leq y$, le cas nul est au contraire rajouté, d'où $Z + \overline{N \oplus V}$.

1.13 Propriétés diverses des entiers relatifs

1.13.1 Opposé

D'après la convention de représentation des entiers signés l'opposé (-X) d'un entier relatif se représente comme le complément à 2 de l'entier (X).

Condition	SPARC	ARM	eq. booléenne
$x = y$	EQ ou Z	EQ	Z
$x \neq y$	NE ou NZ	NE	$\overline{Z}$
$x < y$	L (Less)	LT (Lower Than)	$N \oplus V$
$x \leq y$	LE (Less or Equal)	LE (Less or Equal)	$Z + N \oplus V$
$x > y$	G (Greather)	GT (Greather Than)	$\overline{Z}.(\overline{N \oplus V})$
$x \geq y$	GE (Greather or Equal)	GE (Greather or Equal)	$\overline{N \oplus V}$

TAB. 1.3 – Table des conditions pour entiers signés

1.13.2 Valeur absolue

Si le bit de signe est 0, le nombre est positif ou nul et égal à sa valeur absolue. Un bit de signe à 1 indique un nombre négatif : il suffit d'en prendre le complément à 2 pour obtenir sa valeur absolue (sauf cas particulier de -2^{n-1}).

1.13.3 Valeurs particulières : $0, -1, -2^{n-1}$

0 est son propre complément à 2, ce qui est logique puisque 0 est son propre opposé en arithmétique ($0 = +0 = -0$). Il s'écrit $0 \ldots 0$ (n chiffres à 0).

-2^{n-1} est également son propre complément à 2, alors que son opposé est 2^{n-1}. Cette propriété inattendue résulte de l'asymétrie des intervalles positifs et négatifs d'entiers représentables : 2^{n-1} n'est pas représentable sur n bits.

L'entier signé 2^{n-1} s'écrit en binaire $10 \ldots 0$ (1 suivi de $n-1$ chiffres à 0) et l'entier signé -1 s'écrit $1 \ldots 1$ (n chiffres à 1).

1.13.4 Extension de format et décalage arithmétique

Rappel : les extensions et réductions de format de représentation sont surtout utilisées lors des transferts entre registres et variables en mémoire de taille inférieure à celle des registres.

Pour passer à un codage sur $m > n$ bits, la représentation de l'entier signé doit être complétée par $m - n$ copies du bit de poids fort (bit de signe). A titre d'exemple, -5 s'érit 1011 sur 4 bits et 11111011 sur 8 bits.

L'opération inverse de réduction de format supprime les $m - n$ chiffres de gauche. La valeur est conservée si les chiffres supprimés sont tous identiques.

Le décalage arithmétique à droite est un décalage qui recopie à gauche le bit de signe, donc le bit de poids fort (au lieu de 0 pour un décalage logique). Si les d chiffres de poids faibles de l'entier sont nuls, un décalage arithmétique à droite de d bits divise la valeur de l'entier signé par 2^n.

La constante -2^{n-1} peut être générée par une rotation d'un bit à droite de la constante 1 ; les constantes -2^{n-x} par décalage arithmétique de x bits à droite de la constante -2^{n-1}.

1.13.5 Récupération du signe

Définissons une fonction signe(x) qui retourne 0 pour $x \geq 0$ et -1 pour $x < 0$.

Signe(x) correspond à un décalage arithmétique de x d'au moins $n-1$ bits. Le bit de poids fort (bit de signe) est dupliqué sur les n bits. D'où $00\ldots00$ (0) pour $x \geq 0$ et $11\ldots11$ (-1) pour $x < 0$.

Notons qu'un décalage logique (normalement destiné aux entiers naturels) de $n-1$ bits à droite de x retournerait 0 pour $x \geq 0$ et 1 pour $x < 0$.

Chapitre 2

Variables et expressions en langage C

2.1 Types, variables et constantes

2.1.1 Les types numériques en C

En C, les variables et les constantes sont typées. Le type indique la taille (nombre de bits) et la convention de codage qui permet d'associer une valeur au paquet de bits qu'elle contient. Toutes les variables représentent des nombres.

Les types entiers précisent la taille (nombre de bits) et la convention d'interprétation (naturel ou signé) du nombre entier. Le type nombre à virgule flottante existe également en deux variantes de taille. La taille d'entier la plus efficace à manipuler dépend des machines et correspond au type int. Le tableau 2.1 résume la correspondance pour des machines à processeur RISC 32 bits tels que ceux de la famille ARM.

Appliqué à un type, l'opérateur C sizeof donne le nombre d'unités adressables nécessaire à sa représentation. Autrement dit, sizeof retourne la taille du type exprimée en octets. Sizeof est beaucoup employé dans les programmes utilisant l'allocation dynamique de mémoire (par exemple pour des tableaux de structures).

Type	Synonymes	Taille(bits)	Sizeof()	Interprétation
char		8	1	entier signé
short int	short	16	2	
long int	long	32	4	
int		32	4	
unsigned char		8	1	entier naturel
unsigned short int	unsigned short	16	2	
unsigned long int	unsigned long	32	4	
unsigned int		32	4	
float		32	4	nombre à virgule
double		64	8	flottante

TAB. 2.1 – Les types de variables et de constantes en C

2.1.2 Caractères et chaînes

Contrairement à d'autres langages, C ne définit pas de type caractère spécifique, ni de type chaîne de caractères. Le type char appartient à la famille des types entiers. Bien que celà présente rarement le moindre intérêt, il parfaitement possible d'accéder à un petit tableau avec un indice

de type char plutôt que int.

Le type char correspond à un type entier de petite taille suffisant pour stocker le code ASCII d'un caractère non accentué. Il correspond à la notion de byte en anglais. Le terme anglais octet désigne un paquet de 8 bits. Dans le passé, la taille d'un byte a pu aller de 6 à 12 bits. Aujourd'hui la taille de byte universellement adoptée est de 8 bits, ce qui explique que les termes anglais byte et octet sont tous les deux traduits en français par le même mot : octet.

La notation utilisée pour une constante ASCII est une paire de caractères apostrophe (') encadrant le caractère. Le code ASCII de A se note 'A' et correspond à la valeur entière 0x41. La commande unix man ascii affiche le code du même nom.

caractère	hexa	octal	notation	commentaire
'	0x27	047	\'	
\	0x5c	0134	\\	
"line feed"	0x0a	012	\n	(passage à la ligne)
"carriage return"	0x0d	015	\r	(retour en début de ligne)
tabulation	0x09	011	\t	
"backspace"	0x08	010	\b	(retour en arrière d'un caractère),
"form feed"	0x0c	014	\f	(saut de page)

TAB. 2.2 – Notation de caractères non imprimables et spéciaux

Le tableau 2.2 indique la notation C utilisée pour les caractères spéciaux les plus courants. Il est aussi possible de noter tout caractère par son code octal. Ainsi le caractère '\n'("Line-Feed") peut aussi être noté '\012'.

Le caractère "line feed" correspond à un déplacement vertical d'une ligne vers le bas et "carriage return" un retour en début de ligne courante. Les lignes sont séparées par la séquence "\n\r" (qui réalise à l'affichage un classique passage au début de ligne suivante) dans un fichier de texte au format Windows, ou par le caractère "line feed" seul dans un texte à la norme posix (fichiers unix/linux).

En C, une constante chaîne de n caractères (encadrée par des guiillemets) est en réalité une constante tableau de n+1 éléments de type char contenant le code des n caractères de la chaîne, suivie d'un marqueur de fin de chaîne. Ce dernier est le caractère nul, dont le code ASCII est zéro. Ainsi, la notation **"abc"** n'est qu'un raccourci d'écriture pour **{'a','b','c',0}**.

A noter : la gestion des caractères en C est été conçue pour des caractères ASCII. Le type char et des routines de manipulation de chaînes de la bibliothèque C standard permettent de gérer les carcactères accentués codés sur 8 bits avec une extension ISO du code ASCII, mais pas le codage unicode qui nécessite 16 bits par caractère.

2.1.3 Constantes

Par défaut, en C, les constantes entières s'écrivent en base décimale. Les préfixes 0 et 0x perment de spécifier une constante respectivement en octal et en hexadécimal et les suffixes L et U de spécifier les attributs long et unsigned.

Exemples : La constante entière 171 s'écrit aussi 0253 ou encore 0xAB. La constante 171.0 note la même valeur dans le format à virgule flottante.

Il y a deux manières de définir des constantes symboliques :

```
#define PI 3.14
const int DEUX_PUISSANCE_DIX = 1024;
const float TROIS_FLOTTANT   = 3.0;
```

Le mécanisme de définition de constante #define réalise grosso modo l'équivalent d'une substitution de toutes les occurences de la chaîne PI par la chaîne 3.14 dans toute la suite du fichier, telle qu'on pourrait la réaliser avec la commande chercher/remplacer d'un éditeur de fichiers.

L'attribut const appliqué à une variable indique que son contenu ne doit pas être modifié dans le programme.

2.1.4 Conversions

Pour convertir explicitement une valeur d'un type vers un autre, il suffit de la faire précéder d'un forceur de type (nouveau type entre parenthèses).

```
const   TROIS_VIRG_FLOT 3.0;
const   TROIS_ENTIER (int) TROIS_VIRG_FLOT;
```

La conversion se fait sans perte d'information lorsque toutes les valeurs du type initial sont représentables dans le nouveau type. Il y a par exemple perte de la partie décimale lors d'une conversion de type flottant vers entier. En revanche, tous les entiers courts (short et unsigned short) sont codables dans un flottant (la mantisse est de 23 bits).

Les conversions sont utilisées pour corriger les divergences de type entre opérandes d'une expression. Pour l'affectation à e de la somme de a et d, il est possible de convertir d en unsigned short et le résultat de la somme en float, ou au contraire de convertir a en entier float.

```
float d,e;
short int a;

/* conversion a vers float sans perte */
e = (float) a + d; /* version choisie par le compilateur */
e = (float) (a + (unsigned short) d); /* d vers short : avec perte */
```

Dans les cas simples, les conversions de types omises par le programmeur seront implicitement insérées par le compilateur en privilégiant les conversions de types sans perte d'information.

L'absence de type est notée **void**. C définit le type fonction, mais pas le type procédure. La déclaration d'une fonction spécifie le type de résultat retourné par la fonction. En C, une procédure est déclarée comme une fonction retournant void.

En C, les constantes adresses et les pointeurs sont de type **t ***, t étant le type de l'objet pointé. Tous les pointeurs ont la même taille : celle d'une adresse (32 bits pour un processeur ARM). Un pointeur de type **void *** peut pointer sur n'importe quel type de variable. Mais pour faire un accès à la variable pointée, le contenu du pointeur doit au préalable être converti en adresse du

bon type d'entité pointée.

Le forceur (type *) convertit un pointeur générique en pointeur d'objet de type t. Le forceur ne change pas l'adresse à laquelle il est appliqué : il permet simplement au compilateur de déduire du type t le nombre d'octets à lire ou écrire et comment en interprétrer le contenu. Considérons à titre d'exemple l'allocation dynamique de mémoire pour des structures : un forceur de type (struct st *) permet de convertir le pointeur de char retourné par la routine malloc en pointeur de structure st.

2.1.5 Déclaration des variables et attributs de stockage

Une déclaration de variable(s) sans initialisation comprend un attribut de stockage optionnel, suivi du type et du nom de la variable déclarée (ou d'une liste de noms pour déclarer plusieurs variables de même type) et se termine par un caractère point-virgule.

Par défaut, une variable est supposée stockée en mémoire centrale. L'attribut register permettait au programmeur de suggérer au compilateur de stocker la variable dans un registre plutôt qu'en mémoire centrale, pour en accélérer l'accès.

Cet attribut est aujourd'hui tombé en désuétude avec les compilateurs modernes capables de réaliser une optimisation poussée du code et de l'allocation des registres aux variables. Dans ce document, nous utiliserons l'attribut register pour guider la traduction de programmes C ordinaires en langage d'assemblage.

A noter : l'attribut register interdit l'application de l'opérateur "adresse de" (&) à la variable (un registre du processeur n'a pas d'adresse mémoire).

```
register vreg1, vreg2;  /* vreg1 et vreg2 a stocker dans des registres */
int varint1, varint2;   /* varint1 et varint2 stockees en mémoire centrale */
int avecinit = 12345;   /* avecinit en mémoire avec contenu initial */
```

La déclaration permet de spécifier le contenu initial (précédé du signe =) de la variable au début de l'exécution du programme.

2.1.6 Définition de types par typedef

Typedef permet de définir de nouveaux types à partir de types C de base.

```
typedef unsigned short int age;    /* age est une variante    du type short */
typedef unsigned short int taille; /* taille est une variante du type short */

age age1, age2;           /* deux variables de type age         */
taille taille1,taille2;   /* deux variables de type taille      */
taille2 = (taille) 165;
age1 = taille2;           /* types différents : peut-être une erreur */
```

On peut définir des synonymes de types de base, par exemple des synonymes de int, mais pour un intervalle de valeurs plus restreint (par exemple de 0 à 150 pour un âge et de 0 à 250 pour une taille).

Le compilateur ne fera pas forcément plus de vérifications avec la définition du type synynonyme (age ou taille) qu'avec le type short, mais le programme est plus lisible. Le programmeur voyant un paramètre de fonction de type âge saura implicitement qu'il est censé être inférieur à 150.

Typedef permet aussi de donner un nom à des types complexes construits à partir des types de bases et des opérateurs d'indirection, d'appels de fonction et d'indiçage de tableau (*, ( ), [ ]). Il est en effet difficile d'écrire des déclarations de variables lisibles ou des conversions de type de pointeur sans nommer ce genre de type complexe.

```
typedef  int vect3 [3];  /* le type vect3 est un tableau de 3 entiers */
vect3 v1,v2,v3;          /* déclaration de 3 tableaux de 3 entiers    */

typedef (int *) fonc_pti_de_i_pti (int, int*);
/* le type fonction ayant un argument entier et                 */
/*                       un argument pointeur d'entier          */
/*          et retournant un résultat de type pointeur d'entier */


(int *) calcul (int x, int *y)    /* une fonction de ce genre */
{
...
}

fonc_pti_de_i_pti *ptfonc[4];      /* un tableau de 4 pointeurs */
                                   /*      de telles fonctions */

pt_fonc[1] = &calcul;              /* pt_fonc[1] repere calcul */
```

Ecrire (ou même relire et comprendre) la déclaration du tableau pt_fonc de pointeurs de fonctions sans utiliser le type fonc_pti_de_i_pti n'est pas forcément évident[1].

2.1.7 Enumération de constantes nommées

Supposons que nous voulions déclarer une variable représentant une couleur appartenant à l'ensemble rouge, bleu , vert, noir, blanc.

```
#define ROUGE 0
#define BLEU  1
#define VERT  2
#define NOIR  3
#define BLANC 4
#define BRUN  5

unsigned short int ma_couleur;

...
ma_couleur = VERT;         /* ma_couleur = 2 */
...
if (ma_couleur == NOIR)  traiter_noir();  /* if (ma_couleur == 3) */
```

[1] Elle devrait ressembler à ceci : ((int *) (*pt_fonc) (int, int *)) [4] ...

La technique habituelle consiste à déclarer une variable d'un type entier et à définir des constantes entières symboliques correspondant aux couleur, comme l'illustre le fragment de code précédent.

Enum permet de définir un type synonyme de type entier et une liste de n constantes symboliques qui seront implicitement associées aux constantes 0 à n-1 d'après leur rang d'apparition :

```
enum couleur {ROUGE, BLEU, VERT, NOIR, BLANC, BRUN};
/* rouge 0  bleu 1  vert 2  noir 3  blanc 4  brun 5 */
couleur ma_couleur;


...
ma_couleur = VERT;          /* ma_couleur = 2 */
...
if (ma_couleur == NOIR)  traiter_noir();  /* if (ma_couleur == 3)
```

Il est également possible d'associer explicitement à une constante symboliques une valeur différente de son rang :

```
enum couleur {ROUGE, BLEU, VERT=4, NOIR, BLANC=2, BRUN};
/* rouge 0  bleu 1  vert 4  noir 5  blanc 2  brun 3 */
```

Malgré l'utilisation de enum, le code généré par le compilateur ne vérifiera pas forcément que toute expression affectée à la variable de type enum (ici ma_couleur) appartient à l'intervalle des valeurs définies pour le type couleur. Mais le programme écrit avec enum est plus lisible.

2.2 Opérateurs de calcul et expressions C

2.2.1 Opérateurs arithmétiques

Pour le calcul arithmétique, le langage C offre les opérateurs habituels +, -,*,/. L'opérateur / est interprété comme une division entière ou en virgule flottante selon le type de ses opérandes. L'opérateur % (ou modulo) est le reste de la division entière.

Le signe - représente à la fois un opérateur unaire (un seul opérande : changement de signe) et un opérateur binaire (deux opérandes : soustraction).

```
float x,y,div_ent div_float;
unsigned int f,g,h;
h = f / g;                              /* division entiere de f par g */
div_float = x / y;                      /* division flottante de x par y */
div_ent = (float) ((int x) /(int) y); /* division entiere de x par y */
```

2.2.2 Opérateurs bit à bit et décalages

Les opérateurs logiques binaires travaillant bit à bit sont : & (et), | (ou), ^ (ouex). L'expression a & b retourne un entier dont le bit 0 correspond au produit booléen des bits 0 de a et b, le bit 1 au produit booléen des bits 1 de a et b, etc.

Il existe aussi l'opérateur unaire ~ de complément à un[2]. L'entier signé -1 étant codé en binaire avec tous les chiffres à 1 (voir chapitre 1.8), on peut aussi obtenir ~ x par x ^ -1 (en exploitant la

[2]Rappel : le complément à 1 inverse tous les bits de l'entier

propriété $x \oplus 1 = \overline{x}$).

On trouve aussi deux opérateurs de décalage de l'opérande gauche d'un nombre de bits spécifié par l'opérande droit, respectivement à gauche (opérateur $\ll$) et à droite (opérateur $\gg$).

L'opérateur $\gg$ réalise un décalage arithmétique (recopie du bit de signe en poids fort) si l'opérande est de type entier relatif (int) et un décalage logique (ajout de chiffres à 0 en poids fort) si l'opérande est de type entier naturel (unsigned int).

2.2.3 Gestion de champs de bits

Les décalages et les opérations bit à bit permettent de manipuler des champs de bits à l'intérieur d'un contenant de type entier.

Soit un bit x_r de rang r à manipuler dans un contenant entier x. La représentation de la constante 2^r contient un seul chiffre à 1 de rang r et des 0 à tous les autres rangs. Elle est facilement générée en décalant la constante 1 de r bits à gauche ($1 \ll r$ en C).

Pour forcer x_r à 1, il suffit d'effectuer un OU bit à bit entre x et 2^r (x_i OU $1 = 1$). Pour inverser la valeur de x_r, on utilisera un OU exclusif ($x_i \oplus 1 = \overline{x}$). Un ET bit à bit entre le complément à 1 de 2^r, qui ne contient que des chiffres à 1 excepté au rang r, force x_r à 0 (x_i ET $0 = 0$). Pour tester la valeur de x_r, il à noter que le ET bit à bit de x et 2^r retourne une valeur non nulle si et seulement si $x_r = 1$.

L'entier x peut aussi être considéré comme une juxtaposition de champs de bits représentant chacun une constante entière sur quelques bits.

Soit B un tel champ de b bits occupant les rangs r à $r + b - 1$. Les macros suivantes peuvent être définies pour sa manipulation :

```
#define MASQUE_b_BITS = ((1<<b) -1);
#define MASQUE_B_DANS_X = (MASQUE_b_BITS << r);
#define get_B(x) ((x>>r) & MASQUE_b_BITS)
#define set_b(x,vb) ((x & ~MASQUE_B_DANS_X) | ((vb & MASQUE_b_BIS) << r))
```

Les seuls bits à 1 de la représentation en binaire de la constante $1 << B - 1$ sont ceux de B.

La valeur de B peut être récupérée en calculant le ET bit à bit entre MASQUE_b_BITS et x décalé de r bits à droite (il est aussi possible réaliser le ET bit à bit de x et MASQUE_B_DANS_X et décaler le résultat de r bits à droite ensuite).

L'affectation d'une valeur v_b sera réalisée en deux opérations : mise à zéro de B par ET bit à bit entre x et le complément à 1 de MASQUE_B_DANS_X, suivie d'un OU bit à bit avec v_b décalée de r bits à gauche.

2.2.4 Expressions booléennes

Les opérateurs relationnels binaires incluent les comparaisons habituelles <, <=, >, >=. La comparaison d'égalité se note ==. La négation logique se note ! et la comparaison d'inégalité !=. Ainsi a != b équivaut à ! (a == b).

Les opérateurs && (ET) et || (OU) permettent de construire des conditions composées. Ils considèrent chaque entier sur n bits comme un seul booléen[3], faux ou vrai selon que l'entier est nul ou différent de 0.

La norme C spécifie que l'opérande gauche de l'opérateur && ou || est évalué d'abord et que si cette évaluation retourne faux, l'opérande droit de && n'est pas évalué (il s'agit donc d'un ET puis et d'un OU puis).

C n'offre pas de type booléen comme c'est le cas dans d'autres langages. Il existe cependant une interprétation booléenne de valeurs entières. L'entier 0 est assimilé à faux et toute autre valeur à vrai. Les opérateurs relationnels retournent une valeur entière : 1 pour vrai et 0 pour faux.

Il existe aussi en C des expressions conditionnelles. Le terme précédant ' ?' est la condition à tester. L'expression conditionnelle retourne la valeur de l'expression entre ' ?' et ' :' si la condition est vraie et celle après le ' :' dans le cas contraire.

A titre d'exemple, voici deux manières de calculer dans d le double du maximum de deux entiers a et b. La première (originale mais pas du tout recommandable) exploite le fait que les opérateurs relationnels retournent un entier 0 ou 1. La deuxième utilise une expression conditionnelle.

```
d = 2*((a>=b) * a + (a<b) *b);
d = (a >= b) ? 2*a : 2*b;
```

2.2.5 Opérateur d'affectation

En langage C, l'opérateur d'affectation s'écrit = , (le test d'égalité étant noté ==). **Attention :** dans de nombreux autres langages, tels que ADA ou PASCAL, = correspond à une comparaison et l'affectation se note :=.

Le membre de gauche de l'affectation désigne un contenant qui sera un registre ou un emplacement mémoire. Il peut par exemple s'agir d'une simple variable ou d'un tableau indicé par une expression. Le membre de droite est une expression donnant la valeur à affecter au contenant.

Particularité : en C, l'affectation est une expression (au même titre qu'une opération arithmétique) qui a pour effet de bord de modifier un contenant et qui retourne la valeur de l'expression affectée a la variable. Ainsi, l'instruction C suivante est parfaitement légale. L'opérateur = évalue l'expression x+3, l'affecte à la variable y et retourne cette valeur comme opérande de la multiplication.

```
z = 2 * (y = x + 3) - 4;  /* équivaut à la séquence ci-dessous */

y = x + 3;
z = 2 * y - 4;
```

Cette particularité peut quelquefois améliorer la clarté du programme en cas d'affectations multiples de la même valeur à un ensemble de variables et ou de mémorisation d'expression utilisée dans les tests de boucles, comme le montrent les deux variantes de l'exemple ci-dessous.

```
                                    /* z = lire_valeur();          */
y = z = lire_valeur ();             /* y = z;                      */
```

[3] Contrairement aux opérateurs bit à bit & et | qui traitent chacun des n bits d'un entier comme un booléen.

```
while ((x = (z<<3 - y/2)) != VALEUR_FINALE)  /* x = z << 3 - y/2;          */
                                             /* while (x != VALEUR_FINALE) */
   {                                         /*   {                        */
   corps_du_while;                           /*   corps_du_while;          */
   }                                         /*   x = z << 3 - y/2;        */
                                             /*   }                        */
```

Chaque argument d'appel d'une fonction C est une expression qui est évaluée et dont le résultat est passé à la fonction appelée. Passer une affectation comme paramètre d'appel d'une fonction est parfaitement légal en C (mais à proscrire pour la lisibilité du programme).

```
z = f (1, x + 2, y = g(x) + 1); /* signifie */      y = g(x) + 1;
                                                    z=f (1, x+2, y);
```

2.2.6 Formes abrégées de l'affectation

La notation x += 1 est une abréviation de l'expression x = x + 1. Le principe est applicable à d'autres opérations (-=, *=, &=, |=,...). Si le membre de gauche est défini par une expression, cette expression n'est évaluée qu'une seule fois.

Les opérateurs ++ et — à gauche (notation préfixée) ou à droite (notation postfixée) d'une variable sont des raccourcis d'écriture pour += 1 et -= 1, souvent utilisés pour mettre à jour les variables de boucle à chaque itération. La variable incrémentée peut faire partie d'une expression. La position gauche ou droite de l'opérateur ++ définit l'ordre dans lequel l'incrémentation et l'évaluation de l'expression utilisant la variable seront effectuées.

```
y = tab [x++];       /* correspond a */     y = tab[x]; x = x + 1;
z = tab [++y];                              y = y + 1; z = tab[y];
                                            tampon = f(x);
tab[f(x)] += 3;                             tab[tampon] = tab[tampon] + 3;
```

Le programeur veillera à ne pas abuser de ces raccourcis d'écriture qui n'améliorent pas toujours la lisibilité du programme. Pire, la sémantique d'instructions utilisant ++ et – peut être ambigüe, notamment en version postfixée.

L'effet des instructions ci-dessous dépend de l'ordre dans lequel le compilateur décide de réaliser l'incrémentation de x et le calcul de l'adresse de t[x] dans l'autre membre de l'affectation. Dans la dernière instruction, le compilateur peut choisir de calculer $y + 1$ dans un temporaire, appliquer l'opérateur ++ sur y, puis affecter le temporaire à y (ce qui annule l'effet du ++). Il peut au contraire affecter $y + 1$ à y d'abord et appliquer ensuite ++ sur y déjà modifié.

```
tab[x]   = tab[x++] + 1; /* ambigu :     t[x]=t[x]+1   ou t[x+1]=t[x]+1   ? */
tab[++x] = tab[x]   + 1; /* ambigu :     t[x+1]=t[x]+1 ou t[x+1]=t[x+1]+1 ? */
tab[x++] +=1;            /* non ambigu : t[x] = t[x]+1    puis x= x+1       */
tab[++x] +=1;            /* non ambigu : x = x +1   puis t[x] = t[x] + 1    */
y = ++y + 1;             /* non ambigu : y += 2                             */
y = y++ + 1;             /* ambigu :     y += 1        ou y += 2  ?         */
```

2.2.7 Instructions simples et composées, opérateur virgule

En C, une instruction simple est soit une expression (cette expression pouvant être une affectation), soit un saut de la forme goto étiquette. Dans les deux cas elle se termine par ';'. Il est possible de grouper une séquence d'instructions simples par des accolades, l'ensemble est alors considéré comme une seule instruction.

Une instruction C peut aussi être formée d'un assemblage d'instructions et d'expressions avec les constructeurs algorithmiques (if, while, for, switch etc).

On peut aussi former une expression composée d'expressions simples séparées par des virgules. L'opérateur virgule évalue les expressions de gauche à droite, et retourne la valeur de l'expression de droite. L'ensemble est alors considéré comme une expression unique.

```
while (c=getchar (), c != EOF) /* equivalent a */     c = getchar ();
  {                                                     while (c != EOF)
  ... traiter                                           {
  }                                                     ... traiter
                                                        c = getchar ();
                                                        }
/* on pourrait aussi ecrire */
while ((c=getchar ()) != EOF)  { ... traiter }
```

2.3 Expressions avec parenthèses et priorité des opérateurs

Les expressions que l'on peut rencontrer dans un programme C pourraient être ambigües parce que non parenthésées.

```
unsigned register reg_x, reg_y, reg_z, reg_a, reg_b,reg_c,reg_result;
...
reg_result = reg_a * reg_b + reg_c / reg_x % reg_y - reg_z;
/*****************************************************************/
/* Deux manières parmi d'autres de parenthéser l'expression      */
/* reg_result = (reg_a * reg_b) + (reg_c /reg_x) % (reg_y - reg_z) ou  */
/* reg_result = (reg_a * (reg_b + reg_c)) / (reg_x % reg_y)) - reg_z   */
/*****************************************************************/
```

Obliger à parenthéser totalement chaque expression alourdirait nettement la programmation des expressions mathématiques. En mathématiques, l'usage est d'interpréter l'expression $x-3y/z+a$ comme $(x-((3*y)/z))+a$ plutôt que par exemple $x-(3*(y/(z+a)))$, en considérant qu'on privilégie l'application des opérateurs * et / à celle des opérateurs + et -. On dit aussi que * et / ont priorité sur + et -.

2.3.1 Priorité des opérateurs C

Les opérateurs appliqués en priorité sont les opérateurs unaires d'appel de fonction ( ), d'indiçage de tableau/pointeurs [ ] et d'accès aux champ d'une structure via un pointeur -> (cf table 2.3).

Ils sont suivis des formes unaires des opérateurs - (opposé), * (déréférencement de pointeur), l'opérateur inverse & de prise d'adresse, l'opérateur de détermination de taille (sizeof), les formes préfixées et suffixées des opérateurs ++ et – et le forceur (conversion) de type.

Priorité	Opérateurs	Associativité
16	( ) [ ] ->	
15	++ et – postfixés	
14	++ et – préfixés ! - unaire * unaire & unaire (forceur de type) sizeof()	
13	* / %	Gauche
12	+ -	Gauche
11	« »	Gauche
10	< <= > >=	Gauche
9	== !=	Gauche
8	& bit à bit	Gauche
7	^ bit à bit	Gauche
6	| bit à bit	Gauche
5	&& booléen	Gauche
4	|| booléen	Gauche
3	? :	Droite
2	= += -= *= /= %= »= «= &= ^= |=	Droite
1	,	Gauche

TAB. 2.3 – Table des priorités

Viennent ensuite les opérateurs arithmétiques avec la précédence habituelle de la multiplication et de la division sur l'addition et la soustraction et les opérateurs relationnels de comparaison.

Les versions bit à bit des opérateurs booléens précèdent les opérateurs booléen logiques utilisés pour combiner les conditions.

Les opérateurs appliqués avec la plus faible priorité sont les affectations et les expressions condtionnelles.

A priorité égale, les règles d'associativité spécifie que les opérateurs à gauche sont appliqués avant les opérateurs à droite : l'expression $a + b + c$ sera interprétée comme $(a + b) + c$.

Les opérateurs d'affectation et d'expression conditonnelle font logiquement exception : $x = y = z$ est interprété comme $x = (y = z)$.

2.3.2 Exemples d'application des priorités

L'application de la priorité de () (appel de fonction) sur * unaire (accès à un objet à partir de son adresse) indique que l'expression *f(x) correspond à *(f(x)) (obtention d'un objet dont la fonction f appliquée à x calcule l'adresse) plutôt que (*f) (x) (valeur retournée par une fonction, dont f contient l'adresse, appliquée à x).

La priorité relative des opérateurs arithmétiques (+ - * / %) reflète l'usage adopté en mathématiques et permet d'économiser quelques parenthèses dans l'écriture d'expressions arithmétiques sans risque d'erreur. Pour comparer la parité de deux entiers en utilisant l'opérateur modulo (% : reste de la division entière), on pourra ainsi écrire $x\%2 == y\%2$ que le compilateur interprétera $(x\%2) == (y\%2)$.

Malheureusement, les règles de priorité pour les autres opérateurs ne correspondent pas tou-

jours à l'intuition du programmeur et il est vivement recommandé de parenthéser explicitement les expressions.

A titre d'exemple, la parité peut aussi être obtenue en prenant le chiffre de poids faible par masquage avec 1. Il semblerait naturel d'écrire la comparaison de parité sur le même modèle qu'avec l'opérateur % : $x\&1 == y\&1$. L'opérateur $==$ étant prioritaire sur &, cette expression sera interprétée comme $(x\&(1 == y))\&1$ et non $(x\&1) == (y\&1)$.

Les règles de priorité appliquées aux opérations de décalage ne sont pas plus intuitives. Un programmeur souhaitant forcer à 1 le bit n de x sera probablement tenté d'écrire $x = x + 1 \ll (n-1)$: il semble naturel de supposer que l'opérateur $\ll$ se comporte comme l'opérateur de multiplication et prime sur l'addition. Malheureusement, il n'en est rien et cette expression est interprétée comme $x = ((x+1) \ll (n-1))$ plutôt que $x = (x + (1 \ll (n-1)))$.

2.4 Décomposition d'une affectation en opérations élémentaires

Traduire une expression C en langage machine implique de détailler la décomposition de son évaluation en une séquence d'opérations de calcul élémentaires.

Nous supposerons que toutes les variables sont stockées dans des registres. Ce paragraphe détaille uniquement le stockage des résultats intermédiaires issus de l'évaluation des sous-expressions. Les points spécifiquement liés à gestion des accès aux variables stockées en mémoire centrale font l'objet des chapitres 5 et 6.

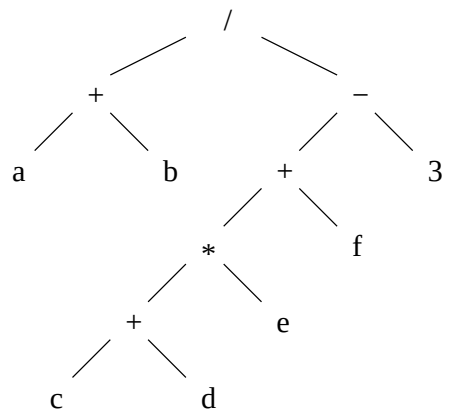
2.4.1 Description arborescente et notation polonaise inversée

Les expressions peuvent être représentées sous une forme arborescente, la racine indiquant l'opérateur à appliquer en dernier et les feuilles les opérandes des opérateurs les plus prioritaires.

L'expression $(a+b)/((c+d)*e+f-3)$ s'écrit $(a+b)/((((c+d)*e)+f)-3)$ avec ses parenthèses. L'opérateur de division occupe la racine de l'arbre associé. Son opérande gauche est le résultat de l'expression $(a+b)$ décrite par son sous-arbre fils gauche et de $(((c+d)*e)+f)-3$ décrite par son sous-arbre fils droit.

La notation polonaise inversée est une notation plus compacte des arbres d'évaluation. Elle consiste à dérire l'arbre dans l'ordre fils_gauche fils_droit nœud_père.

La notation polonaise inversée de la formule ci-dessus s'écrit a b + c d + e * f + 3 - /.



$(a+b)/((c+d)*e+f-3)$

2.4.2 Gestion des temporaires

L'évaluation et le stockage de la valeur de l'expression de droite dans une affectation de la forme $reg_{r3} = reg_{r1} - reg_{r2}$ ne pose pas de problème particulier et se traduit en une unique ins-

truction machine prenant ses opérandes et déposant son résultat dans trois registres généraux du processeur.

Mais dans le cas général, l'évaluation de l'expression nécessite des (emplacements de stockage) temporaires pour mémoriser les résultats intermédiaires (de l'évaluation des sous-expressions).

Pour guider la traduction en langage d'assemblage, il est souvent utile d'écrire en C une variante du programme d'origine ne contenant qu'une opération de calcul simple par affectation et détaillant l'utilisation de registres temporaires. Chaque affectation se traduit alors en une instruction de calcul du processeur.

La stratégie la plus simple consiste à affecter un temporaire distinct à chaque noeud de l'arbre. Ainsi, pour la première affectation, temp1 est utilisée pour évaluer $a + b$, temp2 pour $c + d$ et temp3 pour mémoriser le résultat de la division.

Notons cependant qu'allouer un registre temporaire distinct à chaque nœud de l'arbre est un gaspillage de ressources : lorsque toutes les sous-expressions utilisant la valeur qu'il contient ont été calculées, un registre temporaire peut être réutilisé pour stocker un autre résultat intermédiaire.

Lorsque le membre de gauche de l'affectation est une variable logée dans un registre, il peut remplacer le temporaire associé à la racine de l'arbre. Ainsi, on peut directement calculer r1 = temp1 * temp2 au lieu de passer par un temporaire temp3 dans l'exemple qui suit.

Lorsque sa valeur actuelle n'est pas utilisée, la variable à gauche de l'affectation peut servir de temporaire. Dans la première affectation, r1 peut ainsi remplacer le temporaire temp1. En revanche, r2 ne peut jouer le rôle de temporaire (pour la première affectation) tant que l'expression r2 % 2 n'a pas été évaluée.

2.4.3 Exemple de traduction en langage d'assemblage

Considérons à titre d'exemple la traduction de quatre affectations.

```
register int a,b,c,d,e,f,r1,r2,r3;

r1 = (a+b) * (c+d);                    /* a b + c d + *      */
r2 = (c+d) * (r2 % 2);                 /* c d + r2 2 % *     */
r3 = (a+b) / ((c+d) * e + f -3);       /* a b + c d + e * f + 3 - / */
r4 =  (a-b) * (c-d) + (f-3) / (a-b);   /* a b - c d + f 3 - a b - / */
```

Chaque affectation d'origine de cet exemple se décompose en quatre à six instructions C élémentaires.

```
/* expansion avec les temporaires */
register int temp1, temp2, temp3, temp4, temp5, temp6, temp7;

temp1 = a+b;                           /* ou r1 = a + b;      */
temp2 = c+d;                           /*    temp1 = c + d;   */
temp3 = temp1 * temp2;                 /*    r1 = r1 * temp1  */
r1 = temp3;
```

```
temp1 = c + d;                          /* la valeur de r2 est */
temp2 = r2 % 2;                         /* <--- utilisée ici   */
r2 = temp1 * temp2;

                                        /* variante possible    */
temp1 = a+b;                            /* temp1 = c + d ;      */
temp2 = c+d;                            /* temp1 = temp1 * e ;  */
temp3 = temp1 * e;                      /* temp1 = temp1 + f;   */
temp4 = temp2 + f;                      /* temp1 = temp1 - 3;   */
temp5 = temp4 -3;                       /* r3 = a+b;            */
r3 = temp1 / temp5;                     /* r3 /= temp1          */

temp1 = a-b;                            /* reutilisation de la  */
temp2 = c-d;
temp3 = f-3;
temp2 = temp1 * temp2;                  /* sous expression commune */
temp3 = temp3 / temp1;                  /* a-b calculee dans temp1 */
r4    = temp2*temp3;
```

L'ordre d'évaluation des sous-expressions a une incidence sur le nombre de temporaires à prévoir. L'arbre sera parcouru en privilégiant les nœuds et les branches les plus profonds, la notation polonaise inversée suggérant l'ordre dans lequel effectuer les calculs. Ainsi, l'évaluation optimisée de $(a+b)/((c+d)*e+f-3)$ n'utilise qu'un seul registre temporaire en plus de r3 pour réaliser les six opérations.

A titre d'illustration, voici à quoi peut ressembler la traduction en langage d'assemblage de l'affectation de r1 (non optimisée) de l'exemple.

```
@ r1 : r1
@ r2 : temp1
@ r3 : temp2
@ r4 : temp3
@ r5 : a
@ r6 : b
@ r7 : c
@ r8 : d

add  r2, r5, r6    @ temp1 = a+b
add  r3, r7, r8    @ temp2 = c+d
mul  r4, r2, r3    @ temp3 = temp1 * temp2
mov  r1, r4        @ r1 = temp3
```

Chapitre 3

Ordinateur, langages machine et d'assemblage

3.1 Organisation générale d'un ordinateur

3.1.1 Composants d'un ordinateur

Les ordinateurs sont des calculateurs numériques qui manipulent deux sortes d'informations :

1. Les données regroupent les opérandes et résultats des opérations de calcul, de tri etc. Elles correspondent aux variables et tableaux des programmes (stockées en mémoire centrale) et aux fichiers (stockés en mémoire secondaire).
2. Les programmes exécutables sont des suites d'instructions de la machine (dites "instructions machine") décrivant les suites d'opérations à effectuer sur les données. Les instructions machine sont codées en binaire. Une instruction d'un langage de programmation est traduite en une suite d'instructions machine, chacune effectuant un travail souvent beaucoup plus élémentaire qu'une instruction des langages de programmation tels que C ou ADA.

Il existe des automates qui exécutent une suite d'actions prédéfinies selon un algorithme figé (par exemple sous la forme de cames d'un programmateur). Un tel automate peut être qualifié de machine à programme câblé : il faut en recâbler les circuits électroniques ou changer les rouages mécaniques pour en modifier le comportement. L'orgue de barbarie est au contraire un exemple de machine à programme enregistré : il suffit de changer les perforations du ruban cartonné pour changer le morceau de musique joué par l'orgue.

L'ordinateur appartient à la famille des machines à programmes enregistrés et se caractérise par une grande souplesse de programmation avec la possibilité de réaliser des traitements conditionnels en fonction de résultats de calcul ou de données extérieures, ainsi que des répétitions de séquences (boucles de programmation).

Un ordinateur comprend une mémoire qui stocke les données et les instructions. En pratique, cette mémoire est organisée en une mémoire électronique rapide et volatile (le contenu est perdu lors de la coupure de l'alimentation) et une mémoire secondaire permanente.

La mémoire rapide dite centrale ou principale est utilisée durant l'exécution des programmes et des calculs sur les données. Pour le programmeur, elle est assimilable à un tableau dont les éléments (mots et octets) sont accessibles individuellement.

La mémoire secondaire plus lente, plus économique et non volatile, stocke les programmes exécutables et les données persistentes (fichiers et bases de données) entre deux exécutions et en

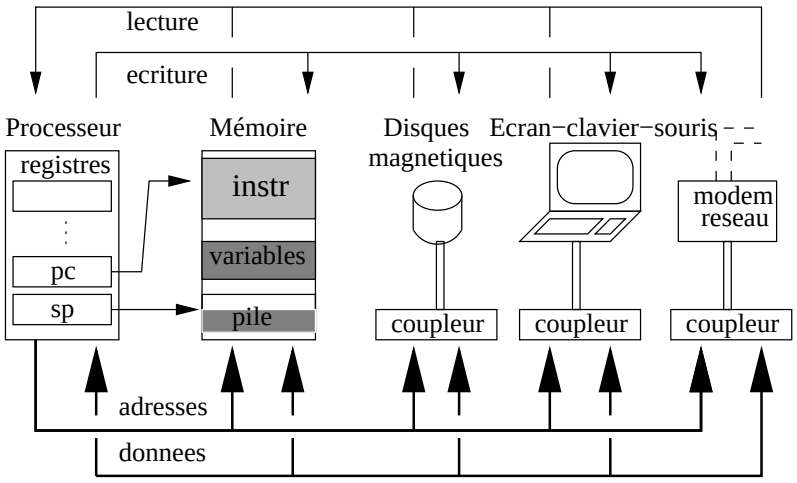


FIG. 3.1 – Schéma synoptique d'un ordinateur

particulier pendant l'arrêt de la machine.

L'ordinateur est aussi doté de périphériques, dispositifs de dialogue avec l'environnement, à savoir le ou les utilisateurs, d'autres ordinateurs ou des processus industriels. Les écrans, claviers, souris, scanners et imprimantes sont les principaux périphériques de dialogue avec l'utilisateur dont sont dotés les ordinateurs personnels. Les périphériques sont connectés à l'ensemble processeur et mémoire via des coupleurs d'entrées/sorties.

Le cœur de l'ordinateur est le processeur, qui comprend les organes de calcul et la circuiterie de pilotage du reste de la machine. Outre les organes de calcul et de séquencement du travail, le processeur est doté de quelques mémoire rapides d'une capacité d'un mot chacune : les registres.

3.1.2 Microactions et instructions

Le processeur est un circuit séquentiel cadencé par un signal périodique dit d'horloge. A chaque cycle (période) d'horloge, le processeur effectue une action très élémentaire (ou microaction). Il peut s'agir d'un calcul interne au processeur sur le contenu de ses registres ou d'un accès à la mémoire.

Lors d'un accès en lecture, la mémoire fournit une copie du contenu d'un mot au processeur. Lors d'un cycle d'accès en écriture, le processeur envoie à la mémoire la nouvelle valeur d'un emplacement à modifier.

Un processeur travaillant à une fréquence de 2 GigaHertz a une période d'horloge de la moitié d'une nanoseconde (0.5×10^{-9} seconde).

Un ordinateur simple exécute les instructions séquentiellement, dans l'ordre où elles sont rangées en mémoire. Un registre (noté PC : Program Counter), appelé compteur ordinal, compteur programme ou encore pointeur d'instruction contient l'adresse (position dans la mémoire) de l'instruction courante (instruction en cours d'exécution).

Chaque exécution d'une instruction représente une séquence de plusieurs microactions. L'exécution d'une instruction simple de calcul, travaillant sur les registres, représente au moins deux ou trois microactions (lecture de l'instruction en mémoire et mise à jour du compteur ordinal pour

passer à l'instruction suivante, réalisation du calcul).

A la mise sous tension, PC est initialisé avec l'adresse de la première instruction du programme de démarrage de l'ordinateur et incrémenté après chaque instruction pour pointer sur l'instruction suivante.

3.1.3 Fonctionnement en pipeline

On peut imaginer de décomposer très finement le déroulement d'une instruction de calcul en microactions : par exemple lecture de l'instruction en mémoire, transfert du contenu des registres vers l'unité de calcul, calcul du résultat, écriture du résultat dans le registre destination.

En pratique, le processeur est souvent capable de travailler "à la chaîne" sur les séquences d'instructions, à savoir dans le même cycle d'horloge effectuer le calcul sur les opérandes de l'instruction courante, ranger le résultat du calcul associé à l'instruction précédente, consulter le contenu de registres spécifiés par l'instruction suivante et lire à l'avance l'instruction d'après.

On parle de traitement en "pipeline[1]". Les instructions progressent le long des circuits du processeur comme un flux : l'instruction qui occupait l'étage de recherche d'instruction au cycle C occupera le circuit de calcul au cycle $C+1$ et l'écriture dans les registres aura lieu au cycle $C+2$.

La durée effective d'une instruction qui correspond au délai de mise à jour du registre résultat n'a pas changé (quatre cycles dans notre exemple). En revanche, le processeur "pipeliné" peut éventuellement commencer l'exécution d'une nouvelle instruction à chaque cycle d'horloge. Le débit théorique du processeur est quadruplé et la durée apparente d'une instruction (utilisée pour estimer le temps de calcul) se réduit à un seul cycle.

Dans une famille de processeurs partageant le même langage machine binaire, on pourrait trouver des processeurs peu optimisés qui exécutent les instructions en mode séquentiel et d'autres qui sont "pipelinés" pour accroître les performances. Certains détails dans la définition du jeu d'instructions des machines résultent d'une conception de processeurs "pipelinés".

Dans le cas de la famille ARM, on notera que lorsqu'une instruction utilise la valeur du compteur ordinal (par exemple pour calculer une adresse de branchement relatif) PC pointe deux instructions plus loin que l'instruction qui en utilise la valeur. La valeur de PC lue est donc celle de l'instruction courante plus huit (deux instructions d'avance occupant 4 octets chacune).

3.2 Organisation et structuration du contenu de la mémoire

3.2.1 Von neumann : un modèle séquentiel à mémoire unique

Conformément au modèle défini par Von Neumann, la mémoire centrale stocke à la fois les instructions et les données. Les emplacements de la mémoire principale ne sont pas typés et peuvent contenir une information de n'importe quelle nature.

En particulier, rien ne distingue les données proprement dites des instructions : le contenu d'un mot mémoire prend simplement le statut d'instruction et est interprété comme tel par le

[1] Ce mot se traduit en français par le suffixe duc, ou éventuellement par le mot conduite. Le terme officellement préconisé pour traduire pipeline n'a pas été retenu dans ce document. Il sonne assez bizarrement à l'oreille : bitoduc ...

processeur lorsqu'il est pointé par le registre PC.

D'une manière générale, le contenu d'un mot mémoire est dépourvu de tout sens en l'absence de convention d'interprétation. En l'absence d'information de type, il est impossible de déterminer si le contenu du mot représente une variable numérique entière signée ou non, un nombre à virgule flottante, une chaîne de 4 caractères, une instruction machine, ou autre chose.

Les contenus des mots transitent entre la mémoire et le processeur par un ensemble de fils porteurs de signaux bidirectionnels : le bus de données. Le numéro d'emplacement de la mémoire accédé par le processeur est appelé adresse et transite via le bus unidirectionnel du même nom (du processeur vers la mémoire).

L'usage a consacré le terme de bus de données (par opposition à adresse), mais le terme de bus de contenu aurait été plus correct. Une information contenue dans une variable pointeur peut être lue puis utilisée ensuite par le processeur comme numéro d'emplacement mémoire à écrire lors d'un cycle d'écriture. Elle sera alors considérée comme une donnée (et voyagera sur le bus du même nom) lors du cycle de lecture, puis comme une adresse (émise par le proceseur sur le bus d'adresse) lors du cycle d'écriture.

3.2.2 Unité de transfert et unité adressable

Rappelons que la mémoire est organisée à la manière d'un tableau, dont les numéros de cases sont les adresses.

L'unité de transfert normale (information transférée en un seul cycle d'accès mémoire) entre le processeur et la mémoire est le mot (32 bits pour ARM), de même taille que les registres, l'unité de calcul et les adresses. Mais la majorité des processeurs est conçue de telle manière qu'un cycle mémoire puisse lire ou modifier un sous-multiple d'un mot dans toucher au reste du mot.

L'unité adressable définit la taille des cases numérotées. Elle correspond à la plus petite taille d'emplacement mémoire qu'il est possible de consulter ou modifier individuellement en un seul cycle d'accès.

Son choix résulte d'un compromis entre plusieurs critères :
- maximiser la quantité de mémoire adressable à taille de mot constante, qui conduit à adopter une unité adressable la plus grande possible (le mot),
- simplifier l'interface mémoire, ce qui s'oppose à une subdivision trop fine du mot, et
- faciliter la manipulation des variables de taille sous-multiples du mot, qui incite à numéroter chaque bit de mémoire d'une adresse de telle sorte que les variables booléennes aient chacune son adresse.

L'unité adressable universellement adoptée aujourd'hui est l'octet. Utiliser des adresses de mot alourdirait notablement la gestion des variables de type caractère, largement utilisées dans de nombreuses applications, dont les éditeurs de fichiers et les traducteurs (compilateurs)[2].

A l'inverse, la réduction de la quantité de mémoire adressable (division par 8 ou 32 par rapport à l'octet et au mot) induite par un adressage individuel de chaque bit de mémoire :
- limiterait la mémoire des machines 32 bits actuels à 512 Moctets,
- serait tolérable sur des processeurs récents à 64 bits,

[2] A moins de ne stocker qu'un caractère par mot, ce qui gaspillerait inutilement de la mémoire.

– était totalement rédhibitoire sur les anciennes machines travaillant sur 16 bits.

Les quelques variables booléennes habituellement présentes dans les programmes peuvent être dotées facilement d'adresses individuelles pour en conserver une gestion simple et efficace. Il suffit simplement d'allouer à chaque variable booléenne un octet ou un mot entier. Le surcoût en mémoire consommée reste très raisonnable compte tenu de la relative rareté de ce type de variable dans la majorité des applications. Cette technique se retrouve dans la définition du langage C (se reporter à l'interprétation booléenne de valeurs entières).

Rappelons que l'opérateur C sizeof(type) prend en paramètre un type ou une variable et en retourne la taille, exprimée en nombre d'unités adressables, donc en octets (colonne Sizeof de la table 2.1).

3.2.3 Ordre de stockage (big/little endian)

Tout objet[3] O, de taille X et d'adresse A, occupe une suite d'octets d'adresses $\{A, A+1, \ldots, A+X-1\}$. Un épisode du voyage de Gulliver relate un conflit portant sur la manière de manger les œufs : en les gobant par le grand bout ou par le petit bout. Les noms des deux méthodes de stockage d'un entier s'en inspirent.

La méthode dite "gros boutiste[4]" ("big endian" en anglais) range les bits de poids forts de l'entier en tête alors que la convention "petit boutiste" (little endian) stocke les bits de l'entier dans l'ordre de poids croissant. Certains parlent aussi de "sexe des machines" à propos de l'ordre de rangement des entiers. Certaines familles de processeurs, dont ARM, autorise le choix de la convention. Dans la suite de ce document, la configuration ARM sera supposée "petit boutiste".

Chiffres de l'entier	Adresse de l'octet	
	GB/BE	PB/LE
$x_{31}x_{30}\ldots x_{25}x_{24}$	A	$A+3$
$x_{23}x_{22}\ldots x_{17}x_{16}$	$A+1$	$A+2$
$x_{15}x_{14}\ldots x_{9}x_{8}$	$A+2$	$A+1$
$x_{7}x_{6}\ldots x_{1}x_{0}$	$A+3$	A

TAB. 3.1 – Conventions gros et petit "boutiste"

Lorsque le contenu de la mémoire est affiché octet par octet, par adresses croissant de gauche à droite (et de haut en bas), le codage "gros boutiste" facilite la lecture des entiers : l'ordre des chiffres hexadécimaux de l'entier est respecté alors qu'avec le codage "petit boutiste", les paires de chiffres hexadécimaux apparaissent dans l'ordre inverse.

Octet d'adresse	0x1000	0x1001	0x1002	0x1003
Gros Boutiste	12	34	56	78
Petit Boutiste	78	56	34	12

TAB. 3.2 – Stockage de l'entier 0x12345678 à l'adresse 0x1000

Les propriétés respectives des deux conventions ne donnent aucun avantage décisif à l'une ou l'autre des deux méthodes. La majorité des familles de processeurs et d'ordinateurs, depuis l'IBM 360 est de type "gros boutiste", excepté entre autres la famille d'INTEL (exception devenue très

[3] Objet pris dans le sens le plus large (entité), hors du contexte de la programmation orientée objet.

[4] La traduction "boutiste" laisse à désirer, mais écrire "grands indiens" ou garder "big endian" ne semble pas plus satisfaisant.

répandue !) 80x86 et Pentium utilisée dans les ordinateurs personnels compatibles IBM PC.

La présence de machines utilisant des conventions différentes complique les échanges de données et doit notamment être prise en compte par les protocoles de transfert d'informations via le réseau.

3.2.4 Contraintes d'alignement

Compte tenu de la manière dont la mémoire est connectée aux bus d'adresses et de données, seuls les octets peuvent être lus ou écrits en un seul cycle à n'importe quelle adresse.

Dans la majorité des machines, les mots et sous-multiples du mot autres autres que l'unité adressable ne sont accessibles en un seul cycle que s'ils sont stockés à une adresse multiple de leur taille.

Le programmeur devra faire en sorte que tout entier sur 16 bits soit stocké à une adresse paire, et tout mot de 32 bits à une adresse multiple de 4.

3.2.5 Sections d'instructions et de données

Un programme en langage d'assemblage décrit (dans l'ordre croissant des adresses) le contenu de la mémoire tel qu'il sera initialisé à partir du fichier binaire exécutable au début de chaque lancement du programme. L'exécution proprement dite commence lorsque le registre PC est initialisé à l'adresse de la première instruction du programme.

Les informations sont habituellement regroupées par nature dans des régions de la mémoire appelées zones ou sections. En fonctionnement normal, les instructions et les constantes ne sont accédées qu'en lecture et regoupées dans une section commune.

Sur les machines dotées d'un système d'exploitation (tel que unix) et des fonctions matérielles associées à la protection, toute tentative d'écriture dans cette zone déclenchera l'arrêt de l'exécution du programme (pour éviter son auto-destruction) et l'affichage d'un message d'erreur à l'intention de l'utilisateur.

Selon la terminologie en vigueur pour les outils GNU dans le monde unix, la section contenant le code des instructions est appelée text[5] et possède automatiquement les attributs de droit d'accès lecture seule et exécution (lecture d'instructions).

Une autre section correspond aux variables globales du programme, et dont la déclaration spécifie une valeur initiale. Cette valeur est stockée dans le fichier exécutable de telle sorte que les variables sont initialisées avant l'exécution de la première instruction du programme. Le nom de cette section est data (accessible en lecture et écriture) dans le monde unix/posix (et par exemple une section avec les attributs DATA et READWRITE dans un autre contexte).

Les variables peuvent être déclarées sans valeur initiale (note : l'utilisation dans une expression de la valeur d'une variable déclarée sans valeur initiale avant qu'elle ne soit initialisée par une affectation constitue en principe une erreur de programmation). La section correspondante s'appelle bss (variante hors gnu/unix : DATA, READWRITE et NOINIT). Le fichier exécutable indique la taille de la section bss, pour que le chargeur/lanceur lui alloue de la mémoire, mais pas

[5]Nom historique. Hors de l'environnement gnu et unix, on décrira par exemple cette section par les attributs CODE et READONLY.

de contenu initial de bss.

Pour des raisons de confidentialité, les systèmes d'exploitation multiutilisateurs initialisent toute la zone bss à 0, mais dans le cas général, (notamment pour des applications embarquées) lors de l'exécution de la première instruction du programme les mots mémoire de la zone bss sont potentiellement susceptibles de contenir des valeurs quelconques.

La gestion des appels de procédure utilise une troisième zone appelée pile. Cette zone est souvent gérée automatiquement par le système d'exploitation ou le chargeur/lanceur, et absente du fichier exécutable. Cete zone est référencée via un registre pointeur de pile (sp sur la figure 3.1).

Remarques :

1. Les zones text, data et pile n'occupent pas forcément des emplacements adjacents en mémoire.
2. Rien n'interdit de mettre des instructions dans une section prévue pour les données. Mais une erreur de manipulation de pointeurs peut venir écraser le code d'une séquence d'instructions stockées en zone dédiée aux données modifiables, ce qui peut compliquer sérieusement la mise au point, et interdit de partager le code entre plusieurs utilisateurs exécutant le même programme sur des données différentes. Cette possibilité est en particulier utilisée par certains interprètes de langages tels que JAVA (compilation "just in time"), qui compilent en cours d'exécution les procédures les plus souvent utilisées.
3. Il possible d'alterner les déclarations de sections dans le texte du programme. L'assembleur regroupera ensemble toutes les séquences étiquetées avec le même nom de section.

3.3 Langages et cycle de vie d'un programme

3.3.1 Fichier binaire exécutable

Un programme exécutable se présente sous la forme d'un fichier binaire stocké en mémoire secondaire. Il représente une image de ce que devra contenir la mémoire principale au début de l'exécution du programme.

A chaque lancement d'une exécution du programme, le contenu du fichier binaire exécutable est recopié en mémoire centrale, puis l'adresse de la première instruction à exécuter dans ce programme est chargée dans le registre PC. Ensuite, le processeur exécute séquentiellement les instructions du programme dans l'ordre où elles sont stockées en mémoire.

3.3.2 Langages machine et d'assemblage

Le jeu d'instructions est l'ensemble des instructions défini par les concepteurs du processeur considéré et que ce dernier sait interpréter en effectuant la suite de microactions correspondante.

En mémoire et dans les fichiers exécutables, ces instructions sont codées en binaire. Elles forment le langage machine du processeur. Le code binaire d'une instruction rassemble un certain nombre de champs de bits : nature de l'opération et opérandes (numéros de registres, constante numérique).

La programmation directe en langage machine binaire ou hexadécimal est extrèmement fastidieuse puisque le programme doit détailler chaque action élémentaire en instruction machine :

un simple appel de procédure peut demander plusieurs dizaines d'instructions en langage machine.

Le langage machine (en binaire ou en hexadécimal) est de plus quasiment illisible pour un humain : il faut une grande habitude du langage machine ARM pour reconnaître par exemple le code d'une addition dans le code hexadécimal 0xE282102A.

C'est pourquoi on utilise un langage dit d'assemblage qui est une représentation textuelle lisible du langage machine. Le langage d'assemblage a le même pouvoir d'expression que le langage machine binaire : chaque instruction du langage machine existe également sous forme symbolique en langage d'assemblage. Le langage d'assemblage permet aussi de donner un nom symbolique aux adresses (les étiquettes) et d'automatiser le calcul des déplacements dans les branchements.

3.3.3 Cycle de vie d'un programme

Chaque famille de processeurs possède son propre langage machine et les programmes en langage machine binaire ou d'assemblage ne sont pas portables entre machines dotées de processeurs de familles différentes.

Les langages de programmation dits "de haut niveau" tels que C permettent l'écriture de logiciels portables d'une machine à l'autre et de programmer à partir de primitives (constructeurs algorithmiques, appels de procédures,...) plus puissantes que les actions élémentaires réalisées par les instructions du langage machine.

Le cycle de vie (simplié) des programmes est le suivant :

1. Le cahier des charges est analysé, puis la structuration du logiciel et les principaux algorithmes sont définis.
2. Les fichiers contenant le texte du programme sont saisis avec un éditeur de texte (tel que vi, emacs ou nedit). Ce type de fichier est appelé fichier source. L'extension .s (langage d'assemblage) ou .c indique le langage dans lequel le programme est écrit.
3. Les fichiers .c sont traduits en langage d'assemblage (fichiers .s) par un programme traducteur appelé compilateur.
4. Les fichiers .s en langage d'assemblage sont traduits en fichiers binaires. Ce type de fichier est appelé fichiers objet (extension .o). Le traducteur est appelé assembleur[6].
5. Les fichiers .o sont réunis en un fichier binaire exécutable par l'éditeur de liens. Généralement, le fichier exécutable porte le nom du fichier source principal, sans extension.
6. L'exécution est déclenchée par le chargeur/lanceur du système d'exploitation. Invoqué par l'interprète de commande, le chargeur copie le contenu du fichier exécutable en mémoire centrale et initialise PC à l'adresse du point d'entrée du programme (première instruction à exécuter) pour en lancer l'exécution.

Lors de la phase de développement, les premiers tests d'exécution révèlent généralement des erreurs de conception ou de programmation. Les corrections sont alors apportées à l'étape correspondante et les étapes de génération de l'exécutable et de test sont reprises. Dans la phase d'exploitation normale du logiciel, seule subsiste l'étape de chargement-exécution du fichier binaire exécutable.

[6]La programmation en langage d'assemblage est souvent appelée programmation en assembleur ou même programmation assembleur. Ces raccourcis de langage, très usités, sont cependant des abus de langage : l'assembleur n'est que le programme traducteur du langage d'assemblage.

Notons que la commande de compilation est généralement capable d'enchaîner toutes les étapes de génération du fichier exécutable, de la compilation en langage d'assemblage proprement dite jusqu'à l'édition de liens. Il s'agit généralement du comportement par défaut, diverses options permettant de limiter le processus à une étape particulière (typiquement -o et -S stoppent le travail du compilateur après la génération des fichiers respectivement .o et .s.).

3.3.4 Syntaxe d'assemblage multiples

Chaque processeur ou famille de processeurs n'admet qu'une syntaxe de langage machine binaire et dont l'interprète est figé dans le matériel du processeur. En revanche il est possible de définir plusieurs variantes de langages d'assemblage pour décrire le même langage machine.

Ainsi, il existe des différences notables, en particulier dans la syntaxe des directives de réservation de mémoire, entre les langages d'assemblage pour ARM d'origine GNU et ceux fournis par ARM. La suite du document utilise la syntaxe GNU.

3.3.5 Justification de l'étude du langage machine

Depuis de nombreuses années déjà, les applications ordinaires ne sont plus écrites en langage machine (ni binaire, ni d'assemblage). On peut donc s'interroger sur l'utilité d'étudier les langages machine et d'assemblage.

L'étude du langage machine est une base indispensable pour comprendre l'architecture et le fonctionnement interne d'un processeur ou de la hiérarchie mémoire (dont la mémoire virtuelle), ainsi que pour écrire les traducteurs générant du code en binaire (assembleurs, compilateurs, éditeurs de liens, gestionnaires de bibliothèques).

En outre, l'apprentissage de la programmation en langage d'assemblage reste cependant utile pour :

1. toutes les opérations nécessitant la manipulation directe de ressources spéciales de la machine (telles que les registres ou instructions spéciaux des processeurs, relatifs par exemple à la commande du système de gestion des interruptions), notamment dans le noyau (cœur du) du système d'exploitation ou dans le cas d'applications embarquées.
2. écrire des bibliothèques optimisées (graphique, calcul), utilisant des instructions spécifiques du processeur. Par exemple, l'absence d'opérateur C correspondant oblige le programmeur en C à réaliser les rotations par des paires de décalages. Le compilateur C n'est pas forcément capable de reconnnaître que la paire de décalages est une opération de rotation réalisable en une seule instruction machine.
3. observer et éventuellement optimiser à la main le code généré par un compilateur pour une procédure dont les performances sont critiques.
4. comprendre la programmation en C, en particulier la gestion des tableaux, pointeurs et paramètres de procédures.

Chapitre 4

RISC, CISC et modes d'adressage

4.1 Interprétation d'une affectation

4.1.1 Exemple d'affectations

Considérons l'extrait de programme C suivant. Il manipule trois variables entières nommées pour simplifier r6 à r8, que nous supposerons stockées dans les registres de même nom du processeur.

```
long int m1 = 55667788;
long int m2 = 11223344;
long int m3;

register long int r6,r7,r8;

...
r6 = r7 - r8;          /* reg_no_6 <- reg_no_7 - reg_no_8 */
m3 = m1 - m2;          /* Mem[120000] <- Mem[100000] - Mem[100004] */
r7 = r8 - 11112222;    /* reg_no_7 = reg_no_8 - 11112222 */
m1 = m2 - 12345678;    /* Mem[100000] = Mem[100004] - 12345678 */
...
```

Il déclare aussi deux variables m1 et m2 stockées en mémoire avec une valeur initiale spécifique et une variable m3 stockée elle aussi en mémoire, mais sans spécification de valeur initiale. Les six variables, de type entier long, sont représentées sur 32 bits.

Nous supposerons que, lors de l'exécution considérée, m1 et m2 sont stockées dans la section data, respectivement aux adresses 00100000 et 00100004. Le contenu de la section data est intialisé avec le contenu du fichier exécutable.

Nous supposerons que m3 occupe l'adresse 00120000 de la section bss (pas de valeur initiale de bss dans le fichier exécutable : initialisation à 0 par défaut).

Ce fragment de code contient quatre affectations, dont nous allons détailler la signification en termes d'actions élémentaires dans la machine et l'expression en instructions du langage machine.

4.1.2 Signification d'une affectation

L'affectation évalue la valeur de son membre droit et l'assigne comme nouveau contenu au contenant désigné dans son membre gauche. Un contenant est désigné par un numéro. Il peut

s'agir d'un registre identifié par son numéro de registre ou d'un emplacement mémoire identifié par son adresse.

Le membre droit de l'affectation prend l'une de ces trois formes :
- une constante (par exemple 11112222) à utiliser directement
- un contenant dont on va utiliser le contenu (par exemple r7)
- une expression utilisant un opérateur de calcul dont chaque opérande est lui-même une constante, un contenant ou une expression.

L'affectation r6 = r7 - r8 correspond à une opération interne au processeur : soustraire le contenu du registre numéro 8 du contenu du registre numéro 7 et stocker le résultat comme nouveau contenu du registre numéro 6.

L'affectation r7 = r8 - 11112222 soustrait la constante 11223344 (contenue dans l'instruction) du contenu du registre numéro 8 et range le résultat dans le registre numero 7.

L'affectation m3 = m1 - m2 lit les contenus des emplacements mémoire de m1 et m2 (respectivement Mem[100000] et Mem[100004]) et les soustrait l'un de l'autre (à l'intérieur du processeur). Elle effectue une écriture en mémoire qui copie le résultat comme nouveau contenu de l'emplacement de m3 (Mem[120000]).

L'affectation m1 = m2 - 12345678 lit en mémoire le contenu de de m2 (Mem[100004]), lui soustrait la constante 12345678 contenue dans l'instruction, et écrit le résultat en m1 (Mem[100000]).

4.1.3 Informations contenues dans la section text

Chacune des affectations C du programme d'origine est traduite dans la section text par une instruction ou une séquence d'instructions de la machine, codée(s) sur un ou plusieurs mots, et décrivant les informations suivantes :

1. la nature du calcul à effectuer (addition, soutraction, etc),
2. la séquence de microactions à effectuer pour récupérer les opérandes et stocker le résultat ou
3. le type d'emplacement (registre, constante incluse dans l'instruction, emplacement mémoire dans une section de donnée) choisi pour le résultat et les opérandes, ce qui définit implicitement la séquence de microactions à effectuer pour y accéder,
4. les numéros de registres utilisés,
5. les constantes adresses des variables stockées en mémoire (exemple 100004 pour m2),
6. les constantes valeurs utilisées dans l'affectation (par exemple 12345678).

Les variantes possibles pour définir les informations autres que la nature du calcul correspondent aux différents modes d'adressage.

4.1.4 Exécution : une séquence de microactions

Les actions élémentaires (ou microactions) réalisables par le matériel de la machine sont les suivantes :
- calcul interne : affecter à un registre du processeur un nouveau contenu calculé à partir du contenu des registres du processeur.
- lire dans la mémoire : copier dans un registre du processeur le contenu d'un emplacement mémoire (dont l'adresse est spécifiée par le contenu d'un registre)

– écrire dans la mémoire : copier le contenu d'un registre du processeur dans un emplacement mémoire (dont l'adresse est spécifiée par le contenu d'un registre du processeur)

Dans une machine simple, chaque microaction correspond à un cycle d'horloge.

Dans la description des affectations, les r_{tempi} désignent des registres du processeur utilisés pour le stockage temporaire d'informations (autres que les variables déclarées du programme C).

L'affectation r6 = r7 - r8 effectue les actions élémentaires suivantes :

– lire en mémoire (dans la section text) chacun des mots spécifiant la nature de l'instruction,
– affectuer la soustraction entre les registres internes au processeur.

L'affectation C r7 = r8 - 11112222 implique une microaction supplémentaire :

– lire en mémoire (dans la section text) chacun des mots spécifiant la nature de l'instruction,
– lire en mémoire (dans la section text) la constante 11112222 incluse dans l'instruction et la stocker dans un registre r_temp1,
– faire la soutraction entre les registres internes au processeur.

L'affectation C m3 = m1 - m2 réalise davantage de microactions :

– lire en mémoire (dans la section text) chacun des mots spécifiant la nature de l'instruction
– lire en mémoire (dans la section text) la constante 00100000, et la copier dans un registre $r_{temp1}1$ du processeur ,
– stocker Mem[r_{temp1}] dans un registres r_{temp2} : il s'agit d'une lecture en mémoire (dans la section de donnée) du contenu de m1 (en l'occurence l'entier 55667788),
– lire en mémoire (dans la section text) la constante 00100004, et la copier dans un registre r_{temp3} du processeur,
– stocker Mem[r_{temp3}] dans un registre r_{temp4} : lecture en mémoire (dans la section de donnée) du contenu de m2 (en l'occurence l'entier 11223344),
– faire la différence $r_{temp2} - r_{temp4}$ (soit 44224444) entre ces deux contenus et la stocker dans un registre r_{temp6},
– lire en mémoire (dans la section text) la constante 00120000, et la copier dans un registre r_{temp5} du processeur,
– écrire r_temp6 dans Mem[120000] : écriture en mémoire (dans la section de données) du nouveau contenu de m3.

L'instruction C m1 = m2 - 12345678 demande une étape de moins :

– lire en mémoire (dans la section text) chacun des mots spécifiant la nature de l'instruction
– lire en mémoire (dans la section text) la constante 00100004, et la copier dans un registre $r_{temp1}1$ du processeur ,
– stocker Mem[r_{temp1}] dans un registres r_{temp2} : il s'agit d'une lecture en mémoire (dans la section de donnée) du contenu de m2 (en l'occurence l'entier 11223344),
– lire en mémoire (dans la section text) la constante 12345678, et la stocker dans un registre r_{temp4} du processeur,
– calculer $r_{temp2} - r_{temp4}$ et ranger le résultat dans le registre r_{temp6},
– lire en mémoire (dans la section text) la constante 00120000, et la copier dans un registre r_{temp5} du processeur,
– écrire r_temp6 dans Mem[120000] : écriture en mémoire (dans la section de données) du nouveau contenu de m3.

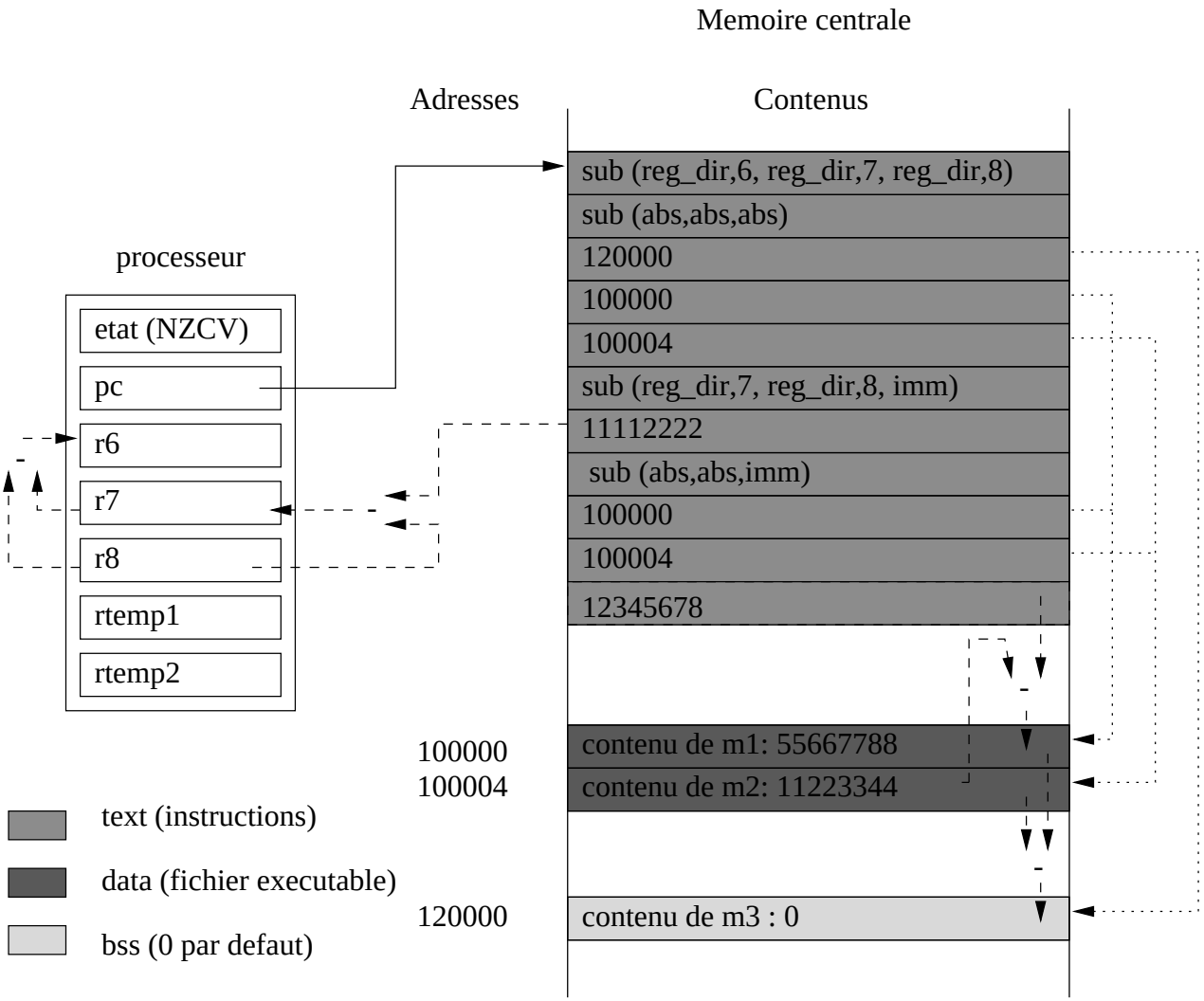


FIG. 4.1 – Registres et contenu de la mémoire : text, data et bss

4.2 Notion de mode d'adressage

Le compteur ordinal repère[1] l'instruction machine courante, composée d'un ou plusieurs mots, généralement stockée dans la section text. Le premier mot de l'instruction contient le code opération de l'instruction (ou un préfixe spécifique à une famille d'instructions additionnelles[2]). Ce dernier définit la nature du calcul effectué et le mode d'accès aux opérandes et au résultat. Selon les modes d'accès utilisés, il est éventuellement suivi d'informations d'adressage complémentaires.

4.2.1 Méthodes d'adressage

L'information d'adressage peut être omise lorsque la méthode d'accès est définie par convention. C'est par exemple le cas d'une instruction "empiler" utilisant systématiquement le registre sommet de pile. On parle alors d'adressage **implicite**.

L'instruction peut spécifier un registre à usage général. L'information d'adressage se limite à un numéro de registre. Il s'agit le plus souvent d'un nombre entier codé sur 3 à 5 bits, les proces-

[1] contient l'adresse de

[2] Les concepteurs de processeurs ont parfois recours à la surcharge de code opération pour ajouter de nouvelles instruction (calcul graphique, vectoriel, extension d'instructions à des entiers sur 64 bits, etc), via un préfixe (un code opération inaffecté dans le jeu d'instruction initial) modifiant la signification du code opération qu'il précède.

seurs étant majoritairement dotés de 8, 16 ou 32 registres généraux. Cette taille réduite permet généralement de l'encoder dans le même mot que le code opération.

En mode **registre indirect**, le registre **repère** le contenant en mémoire : il contient l'adresse à laquelle est stocké l'opérande ou le résultat. En mode **registre direct**, l'opérande est stocké dans le registre : le registre **est** le contenant.

L'utilisant du compteur ordinal (pc) en mode indirect permet d'accéder au contenu (de la section text) d'un emplacement mémoire voisin du code opération de l'instruction. Il est commode de stocker une constante immédiatement derrière le code opération de l'instruction.

Dans le cas d'un adressage dit **immédiat** (souvent noté avec un caractère #) cette constante **est** l'opérande de l'instruction de calcul[3]. L'adressage immédiat permet de gérer des calculs sur des constantes. La taille de l'information d'adressage est celle de la constante (typiquement un mot).

Si l'adressage est de type **absolu** (ou direct mémoire) cette constante est l'adresse mémoire du contenant et **pointe** l'opérande. La taille de l'information d'adressage est alors celle d'une adresse, soit un mot (32 bits pour un processeur ARM). L'adressage absolu permet d'accéder à des variables ordinaires stockées en mémoire.

Les adressages de type **indexé** et **base plus déplacement** sont des variantes du type registre indirect utiles pour l'accès aux élements d'un tableau ou aux membres d'une struture. Ils définissent l'**adresse** de l'opérande comme la **somme** d'une constante et du contenu d'un registre ou la somme du contenu de deux registres.

L'adressage **relatif** est une variante d'adressage indexé utilisant le registre **compteur ordinal**. Il est rarement utilisé pour accéder à des contenus de variables[4]. Il est principalement utilisé pour définir la cible des branchements par rapport à l'emplacement de l'instruction de saut.

4.2.2 Notations et exemple

Il n'existe pas de syntaxe universelle respectée par l'ensemble des langages d'assemblages, en particulier pour les notations des modes d'adressage. Nous définirons donc notre propre notation pour la suite de ce document.

Pour l'adressage registre direct, nous écrirons simplement le nom du registre. Pour l'adressage immédiat, nous utiliserons le caractère # suivi de la constante immédiate.

Nous utiliserons une paire de crochets pour décrire les autres modes d'adressage explicites. Les crochets contiennent une liste d'éléments séparés par des virgules. L'addition de ces éléments définit l'adresse de l'emplacement mémoire accédé. Ainsi, l'adressage (absolu) d'une variable stockée à l'adresse 12 se note [12]. L'adressage (registre indirect) d'une variable pointée par le registre r3 s'écrit [r3].

Illustrons cette syntaxe par quelques exemples en supposant que les instructions de calcul du processeur offrent un large choix de modes d'adressage :

```
@ r6 = r7 - r8
```

[3] L'adressage immédiat n'est pas applicable au résultat qui n'est pas une constante.

[4] Excepté pour écrire des programmes qui ne dépendent pas de l'endroit où ils sont chargés en mémoire

```
     sub r6, r7, r8                          @ (reg dir, reg dir, reg dir)

@ m3 = m1 - m2
     sub [120000], [100000], [100004]  @ (abs, abs, abs)

@ r7 = r8 - 11112222
     sub r7, r8, #11112222                   @ (reg dir, reg dir, imm)

@ m1 = m2 - 12345678
     sub [100000], [100000], #12345678 @ (abs, abs, imm)

@ un autre exemple de modes d'adressages
     add [r1], [r2,r3], [r2, 12]             @ (reg ind, index, base+depl)
```

La dernière instruction spécifie de retrancher le contenu de la mémoire dont l'adresse est donnée par le contenu du registre r2 auquel on ajoute 12 (adressage base plus déplacement), du contenu de la mémoire dont l'adresse donnée est la somme des contenus des registres r2 et r3 (adressage indexé), et de stocker le résultat dans l'emplacement mémoire pointé par r1.

Le contenu de la section text correspondant aux quatre premières instructions ce programme est décrit sur la figure 4.1. Le code opération de chaque instruction occupe un mot. Celui de la deuxième instruction est suivie dans la section text des trois adresses (de m3, m1 et m2) et celui de la quatrième des deux adresses (de m1 et m2) et de la constante immédiate. L'instruction d'addition finale pourrait occuper deux mots : un code opération et la constante 12.

4.3 Jeux d'instructions CISC et RISC

4.3.1 Approche CISC

Dans l'approche qualifiée de CISC (Complex Instruction Set Code), chaque instruction de calcul offre une un large choix de modes d'adressages, ce qui permet de manipuler directement des variables rangées en mémoire.

Chaque instruction de calcul existe en autant de variantes que de combinaisons de choix de modes d'adressage pour et les opérandes gauche et droit. La complexité réside dans la combinatoire de choix : avec 8 modes d'adressages on peut déjà définir de l'ordre de 500 variantes[5] de la même instruction d'addition, avec des tailles et des durées d'exécution différentes.

L'affectation r6 = r7 - r8 est réalisable en une seule instruction machine de soustraction. Cette dernière n'utilise que l'adressage registre direct et n'occupe qu'un mot (le code opération). Elle effectue un cycle d'accès mémoire (un en section text et aucun accès en section de données).

L'affectation m3 = m1 - m2 est réalisable en une seule instruction machine n'utilisant que le mode d'adressage absolu. Cette dernière occupe quatre mots (le code opération et les 3 adresses 1000000, 10004 et 120000) et correspond à sept cycles d'accès à la mémoire (4 dans la zone text et 3 dans la section de données).

[5] 7x8x8, l'adressage immédiat étant disponible pour les opérandes et exclu pour le résultat

4.3.2 Approche RISC : load/store et calcul sur les registres

La stratégie RISC (Reduced Instruction Set Code) consiste au contraire à contraindre fortement le choix de modes d'adressage. Le principe est de disposer d'un ensemble minimal d'instructions de taille fixe réalisant chacune des actions très élémentaires. Le programmeur décrit l'accès à ses données par une séquence d'instructions simples plutôt qu'une instructions dotée de modes d'adressages sophistiqués.

Dans une machine RISC typique, les instructions de calcul sont codées sur un seul mot et ne travaillent que sur les contenus des registres (adressage registre direct imposé). Leur durée apparente[6] est d'un cycle d'horloge.

Une instruction load (que nous noterons ldr : LoaD Register) lit le contenu d'un emplacement mémoire et le copie dans un registre du processeur. Une instruction store (que nous noterons str : STore Register) écrit une copie du contenu d'un registre dans un emplacement mémoire. Les modes d'adressages disponibles sont registre indirect et immédiat[7].

Les approches CISC ou RISC correspondent à deux "granularités" différentes de description des actions à réaliser : séquence courte d'instructions machines complexes et puissantes décrivant chacune une (potentiellement) longue suite de microactions ou séquence plus longue d'instructions simples décrivant chacune une action relativement élémentaire.

Un jeu d'instructions RISC n'est pas plus restrictif qu'un jeu d'instructions CISC : toute instruction de calcul CISC, quelle que soit la variante de modes d'adressage utilisée, admet une séquence d'instructions RISC équivalente.

4.3.3 Exemple de programme pour une machine RISC

Reprenons l'exemple pour notre jeu d'instructions dans la philosopie RISC.

```
@ r6 = r7 - r8
    sub r6, r7, r8          @ idem CISC avec adressage registre direct

@ m3 = m1 - m2
    ldr  r1, #100000        @ r1 = 1000000
    ldr  r2, [r1]           @ r2 = contenu de m1 (Mem[1000000])
    ldr  r3, #100004        @ r3 = 1000000
    ldr  r4, [r3]           @ r4 = contenu de m2 (Mem[1000000])
    sub  r0, r2, r4         @ r0 = m1 - m2
    ldr  r5, #120000        @ r5 = 1200000
    str  r0, [r6]           @ m3 (Mem[1200000] = m1 - m2

@ r7 = r8 - 11112222
    ldr  r0, #11112222
    sub  r7, r8, r0

@ m1 = m2 - 11223344
    ldr  r1, #100004        @ r1 = 1000004
    ldr  r2, [r1]           @ r2 = contenu de m2 (Mem[1000004])
```

[6] cf le fonctionnement pipeline

[7] immédiat : uniquement pour load

```
ldr  r3, #11223344       @ r3 = la constante
sub  r4, r2, r3          @ r4 = m1 - m2
ldr  r5, #120000         @ r5 = 1200000
str  r4, [r5]            @ m3 (Mem[1200000] = m1 - 11223344
```

Dans la section text, le code opération de chaque instruction occupe un mot. Pour une instruction de type ldr immédiat, le mot suivant le code opération contient la constante immédiate. Toutes les autres instructions sont codées sur un seul mot.

4.3.4 Taille des opérandes et choix du type de variables entières

Les calculs peuvent porter sur des variables stockées en mémoire de tailles diverses, allant de l'octet (char) au mot complet (long).

La famille 680x0 illustre une philosophie CISC : chaque instruction de calcul existe en autant de variantes que de tailles d'entier supportées.

La famille ARM illustre une approche opposée commune à de nombreux processeurs RISC. L'instruction load effectue une conversion au format 32 bits lors de la lecture en mémoire de variables entières de format inférieur au mot. Tous les calculs sont réalisés entre registres, sur des mots complets. L'instruction store réalise une réduction de format et tronque les bits de poids fort du résultat lors de l'écriture en mémoire du contenu d'un registre dans une variable de taille inférieure.

Un processeur RISC 32 bits typique offrira donc huit instructions d'accès à la mémoire : **ldr** (entiers 32 bits), **ldrh** et **ldrsh** (entiers 16 bits naturels et signés), **ldrb** et **ldrsb** (entiers 8 bits naturels et signés), **str**, **strh**, **strb**. A noter : la distinction entre entiers naturels et signés ne s'applique qu'à l'extension à 32 bits du format de l'entier, donc uniquement à l'instruction ldr.

L'approche RISC a cependant un petit inconvénient si le processeur n'est pas doté d'une paire d'indicateurs C et V (et d'un jeu de branchements conditionnels pour les tester) spécifiques à chaque taille d'opérande. C'est le cas du ARM, dont les indicateurs C et V ne sont significatifs que dans le cas d'un calcul sur des entiers de la taille du mot.

Pour que C et V du processeur ARM signalent les débordements correspondant effectivement à un calcul sur 8 (respectivement 16) bits, le programmeur peut décaler les opérandes à gauche de 24 (respectivement 16) bits, réaliser l'opération, puis décaler de 24 (respectivement 16) bits à droite le résultat.

Notons que pour respecter la compatilibité le jeu d'instructions 32 bits, les concepteurs de la version 64 bits du SPARC ont choisi de la doter d'indicateurs et d'instructions de branchement distincts pour les modes 32 et 64 bits.

4.3.5 Choix du type des variables entières

Lors de la traduction de son algorithme en langage C, le programmeur doit choisir le type (naturel ou signé) et la taille de ses variables entières .

Le type unsigned est destiné aux grandeurs qui ne prennent pas de valeur négative : âge d'une personne, date, indice de boucle de parcours de tableau, adresse d'une variable (pointeurs), etc. Il

permet de représenter des valeurs entières maximales doubles de celles de type signé.

La taille des variables logées en mémoire peut être ajustée au plus près des besoins pour réduire la consommation de mémoire du programme. Pour les variables stockées dans des registres, l'argument ne tient plus et on pourra faire l'impasse sur les sous-multiples du mot.

Il convient de garder à l'esprit que les intervalles de valeurs entières codables sur 8 et 16 bits sont sévèrement limitées (cf tableau 1.8). L'éventail de valeurs représentables sur 32 bits peut sembler important et mettre le programmeur à l'abri des débordements. La lecture d'un journal financier donnant la capitalisation boursière de sociétés cotées dissipera vite cette illusion : celles dont le capital dépasse les 4 milliards ne sont pas rares.

Les anciennes architectures travaillant sur 16 bits ont disparu des postes de travail[8] parce qu'incapable de gérer plus de 64 Koctets de mémoire. Pour les mêmes raisons que à l'époque des machines à 16 bits, la migration progressive de 32 bits à 64 bits est en cours : aujourd'hui, des serveurs dotés de plus de 4 Goctets de mémoire centrale sont disponibles à un coût raisonnable dans le commerce.

Le risque de débordement est nettement plus faible sur 64 bits : les grandeurs utilisées dans la vie courante susceptibles de dépasser 10^{18} sont rares. Sur 64 bits, le choix entre entiers naturels et signés est moins crucial.

4.3.6 Mise à jour des indicateurs arithmétiques

Pour certaines opérations, il est important de tester l'absence de débordement ou la nullité du résultat. Il est alors indispensable que l'instruction de calcul mette à jour les indicateurs (ZNCV) à partir desquels une décision sera prise. Ce genre de décision correspond souvent à la traduction d'une condition d'une construction algorithmique, du genre **if (a <= b) ...**.

Ce genre de test est en revanche rarement effectué sur certains types de calcul, tels que les calculs d'adresse pour l'indiçage des tableaux.

La majorité des machines RISC laisse le choix de mettre à jour les indicateurs ou non et offre deux variantes de chaque instruction de calcul. Les instruction addS (ARM) et addcc (SPARC) mettent à jour le registre d'état. L'instruction add (ARM et SPARC), au contraire, laisse les indicateurs inchangés.

Seules instructions de calcul sans mise à jour des indicateurs peuvent être librement insérées entre un branchement conditionnel et la comparaison à partir de laquelle le branchement prend une décision.

Les machines CISC n'offrent généralement pas cette souplesse et la liste des indicateurs modifiés est définie instruction par instruction par le concepteur du jeu d'instructions. Cette restriction économise un bit dans le code opération (souvent limité à 8 ou 16 bits).

[8] Mais sont encore utilisées dans des systèmes embarqués

4.4 Jeux d'instructions limités à un ou deux opérandes

Les concepteurs de jeux d'instructions ont été confrontés à la taille du code opération, qui dépend du nombre de registres, de modes d'adressages utilisables et de type de calculs réalisables.

Considérons un processeur doté de 32 registres, supportant 16 variantes de modes d'adressage et capable de réaliser 32 types d'opérations différentes. Supposons qu'il soit doté d'un jeu d'instructions dit à trois opérandes (opérandes gauche et droit, résultat). La taille minimale du code opération d'un tel processeur est de 5 bits (nature du calcul) plus 4 bits de mode d'adressage et 5 bits de numéro de registre[9] par opérande, soit 32 (3x9+5) bits.

Dans le passé, la taille et le débit de la mémoire étaient sévèrement limités. Les concepteurs de processeurs ont alors cherché à concevoir un encodage des instructions très compact en restreignant le choix des modes d'adressage et/ou en ne spécifiant pas tous les opérandes de manière implicite.

4.4.1 Le 68000 : exemple d'instructions à deux opérandes

Examinons à titre d'exemple les contraintes appliquées au jeu d'instruction de la famille 680x0 pour que la taille du code opération soit de 16 bits.

Les instructions de calcul ne spécifient que l'opérande gauche et l'opérande droit. Une convention implicite spécifie que le résultat est stocké à la place de l'opérande gauche. Toutes les opérations sont alors de la forme destination = destination opération source (ce qui correspond aux opérateurs +=, -=, *=, ...du langage C). L'initialisation de la destination est réalisée par une instruction move qui réalise une simple copie de la source vers destination.

Il existe un deuxième restriction[10] : l'adressage d'au moins un des deux opérandes (source ou destination) est de type registre direct (parmi 8 registres de données).

Le code opération sur 16 bits des instructions de calcul est structuré comme suit :

- 4 bits spécifient la nature de l'instruction (type de calcul)
- 2 bits si le calcul porte sur des entiers de 8, 16 ou 32 bits,
- 3 bits encodent le numéro du registre,
- 1 bit spécifie si le numéro de registre précédent correspond à l'opérande source ou à la destination,
- 6 bits (3 pour encoder le mode d'adressage parmi 8 et 3 autres pour le numéro du registre éventuellement utilisé par le mode d'adressage) spécifient le mode d'adressage de l'autre opérande.

Avec la syntaxe de description des modes d'adressage que nous avons définie dans ce chapitre[11], les quatre affectations de notre exemple seraient traduites en dix instructions 68000. Le sufixe l indique une taille de calcul de 32 bits.

```
move.l    d6, d7  @ destination a gauche : reg_d6 <- reg_d7
sub.l     d6, d8  @ reg_d6 <- reg_d6 - reg_d8  (reg direct x 2)

move.l    [120000], [100000]  @ m3 <- m1       (absolu, absolu)
move.l    d0, [100004]        @ reg_d0 <- m2   (red direct, absolu)
```

[9] Pour les modes d'adressage spécifiant explicitement un registre

[10] Cette restriction ne s'applique pas à l'instruction move

[11] Les assembleurs 68000 disponibles dans le commerce utilisent des notations différentes

```
    sub.l     [120000], d0        @ m3 <- m3 - reg_d0  (absolu, reg direct)

    move.l    d7, d8              @ d7 <- d8        (reg direct, reg direct)
    sub.l     d7, #11223344       @ d7 <- d7 - cte  (reg direct, immediat)

    move.l    [120000], [100000]  @ m3 <- m1        (absolu, absolu)
    move.l    d0, 100004          @ reg_d0 <- m2    (red direct, absolu)
    sub.l     [120000], d0        @ m3 <- m3 - reg_d0 (absolu, reg direct)
```

4.4.2 Machines à accumulateur : instructions à un opérande

Les microprocesseurs 8 bits des années 70 étaient doté d'un registre de travail unique appelé accumulateur et éventuellement d'un ou deux registre(s) d'index utilisé avec les modes d'adressage de type registre indirect.

Par convention implicite, les instructions de calcul sont de la forme accu = accu opération opérande et le code opération spécifie un seul mode d'adressage (pour l'opérande droit).

Le code opération est un octet, dont on peut utiliser par exemple 3 bits pour choisir un mode d'adressage de l'opérande droit parmi 8 possibles, et 5 bits pour spécifier un type d'opération parmi 32.

L'instruction load effectue une lecture de la mémoire et la copie dans l'accumulateur. L'instruction store écrit dans la mémoire et y copie le contenu de l'accumulateur. Le fragment de code ci-dessous illustre la traduction des affectations de variables m1 et m3.

La traduction de notre d'exemple (toujours avec les même notations de mode d'adressage) n'a de sens que pour les variables stockées en mémoire (en l'absence de registres de travail utilisables pour stocker les variables).

```
@ m3 = m1 - m2

    load      [100000]  @ accu = contenu de m1  (absolu)
    sub       [100004]  @ accu = accu - contenu de m2 (absolu)
    store     [120000]  @ m3 <- accu  (absolu)

@ m1 = m1 - 12345678

    load      [100004]  @ accu = contenu de m2  (absolu)
    sub       #12345678  @ accu = accu - 12345678  (immédiat)
    store     [100000]  @  m1  = accu (absolu)
```

Chapitre 5

Réservation et initialisation de la mémoire

5.1 Notion d'étiquette et de fichier relogeable

5.1.1 Notion d'etiquette

L'utilisation d'adresses numériques rend notre exemple de programme machine présenté au chapitre 4 difficile à lire. Considérons à titre d'exemple **sub [120000], [100000], [100004]** : le lecteur doit consulter en permanence la table des adresses auxquelles sont rangées les variables pour remonter de cette instruction machine à l'instruction C d'origine **m1 = m2 - m3**.

Les erreurs de transcription des adresses numériques sont tout aussi difficiles à repérer : si la première adresse est accidentellement remplacée par 1200000, la présence d'un chiffre à zéro excédentaire dans la constante adresse risque fort de passer inaperçue.

Le langage d'assemblage permet de définir des étiquettes. Une étiquette permet d'associer un nom symbolique à une adresse. Lors de l'assemblage, l'assembleur sustituera chaque utilisation de l'étiquette (comme opérande d'une instruction ou d'une directive de réservation de mémoire) par l'adresse numérique associée à l'étiquette.

5.1.2 Notion de programme relogeable

Le fichier exécutable décrit le contenu initial que devra avoir la mémoire à l'instant auquel l'exécution du programme sera lancée. Programmer en langage d'assemblage revient à décrire, dans l'ordre croissant des adresses, le contenu initial des sections text et data.

La description de la section text débute par la directive text et celle de la section data par la directive data. Chaque ligne contenant une directive de réservation de mémoire décrit le contenu d'un ou plusieurs octets et définit éventuellement une étiquette.

Ce contenu inital de text et data peut dépendre des adresses auxquelles sont placées les sections text, data et bss et le contenu initial de tous les mots contenant une adresse dans ces sections est à modifier si l'implantation des sections dans la mémoire centrale change.

Dans la section text, il peut s'agir d'une instruction CISC utilisant un mode d'adressage absolu[1] pour accéder au contenu d'une variable stockée dans la section data (ou bss) : un des mots de l'instruction machine contient l'adresse mémoire de la variable.

[1] ou d'une séquence équivalente d'instructions RISC

Dans la section data, ce cas figure correspond généralement à une variable pointeur initialisée dans sa déclaration avec l'adresse d'une autre variable.

A titre d'exemple, le contenu de la section text de l'exemple détaillé figure 4.1 n'est valable que pour une exécution telle que les sections data et bss débutent aux adresses 100000 et 120000. Si la section bss débutait à l'adresse 140000, tous les adresses 120000 (correspondant à l'adresse de m3) contenues dans les instructions de la section text devraient être remplacées par 140000.

Il existe au moins trois cas de figures tels que les adresses des sections text, data et bss ne sont pas connues du programmeur au moment de l'assemblage du fichier.

Le programme peut être conçu pour être exécuté sur plusieurs machines dans lequelles l'intervalle d'adresses correspondant à la mémoire vive utilisable varie d'une machine à l'autre. Il s'agit alors d'un problème de portabilité entre machines.

D'autre part, le programme peut être écrit sous la forme de modules compilés séparément et fusionnés en un fichier binaire exécutable unique lors d'une étape d'édition de liens. Même si l'adresse de début des sections est connue à l'avance, l'adresse à laquelle une variable sera stockée à l'intérieur de sa section dépend de l'ordre dans lequel la fusion est effectuée.

Le programme peut aussi être destiné à un système multitâche dans lequel l'espace mémoire est partagé entre plusieurs programmes en cours d'exécution. L'adresse de chargement d'une section du programme est susceptible de varier d'une exécution à l'autre, en fonction de ce que le système d'exploitation aura alloué aux autres programmes.

Les assembleurs et compilateurs génèrent donc un fichier binaire dit relogeable, dans lequel tous les emplacements dont le contenu initial dépend de l'adresse de chargement d'une section sont repérés.

Le système d'exploitation et la chaîne de compilation collaborent de telle sorte que le contenu du fichier relogeable est corrigé lorsque les adresses des sections text, data et bss sont connues. Au plus tard, cette correction, appelée réimplantation, est effectuée juste avant l'exécution de la première instruction du programme.

Outre une lisibilité accrue du code, le mécanisme d'étiquette permet au programmeur de spécifier une adresse dans une section, même si l'adresse de début de section n'est pas encore connue à l'assemblage.

5.2 Réservation et initialisation de mémoire dans data

5.2.1 Directive byte et définition d'étiquette

La directive byte permet de réserver de la place et de spécifier la valeur initiale d'un octet. Chaque directive de réservation peut petre précédée d'une définition d'étiquette, auquel cas le nom de l'étiquette apparaît en début de ligne, suivi de deux points (caractère :). La valeur initiale doit être une constante entière codable sur 8 bits.

L'exemple ci-dessous décrit une table des 6 première puissances de 2, codées chacune sur un octet, suivie de 254 et 255 et du code ASCII de 'A' et de 'B'.

```
ASCII_DE_B = 0x42
```

```
            .data
debut_tab:  .byte 1
            .byte 2
            .byte 4
huit:       .byte (1 << 3)      @ 1 << 3 : la constante 8
fin_tab:    .byte 0x10          @ meme effet que .byte 16
            .byte 254           @ 254 pris comme entier naturel
            .byte -1            @ -1 pris relatif équivalent à 255
            .byte 'A'           @ 'A' signifie 0x41
            .byte  ASCII_DE_B   @ meme effet que .byte 0x42
```

La troisième ligne de l'exemple définit une étiquette : elle associe le nom d'étiquette debut_tab à l'adresse du premier octet de la section data. La directive **.byte 1** réserve ce premier octet et spécifie que son contenu initial est 1.

La quatrième ligne spécifique le contenu initial (2) du deuxième octet de la section data, et ainsi de suite. La sixième ligne associe le nom d'étiquette huit à l'adresse de la section data plus 3 (adresse de l'octet initialisé à 8). De même, la dernière ligne définit l'étiquette fin_tab comme l'adresse de la section data, plus quatre.

La valeur initiale peut être une constante numérique ou une expression ne contenant que des constantes. La directive = ne réserve pas de place, mais permet de donner un nom symbolique à une constante : l'assembleur remplacera chaque occurence ultérieure du symbole ASCII_DE_B par 0x42. Son équivalent C s'écrirait **#define ASCII_DE_B 0x42**.

5.2.2 Directive word et utilisation d'étiquette

La directive word permet de réserver de la place et spécifier une valeur initiale pour un mot. Elle équivaut à une séquence de quatre[2] directives .byte spécifiant chacune un octet du mot[3]. L'opérande de .word est une constante numérique ou une étiquette.

```
            .data
debut:      .word  1234
adr_x:      .word  0            @ int x = 0
adr_ptr_x:  .word  adr_x        @ int *ptr_x = &x
```

La ligne **.word 1234** réserve quatre octets et les initialise à la valeur entière 1234 (codée sur 32 bits). La troisième ligne associe à l'étiquette adr_x l'adresse de début de la section data, plus quatre et réserve de la place pour un mot, initialisé à 0.

La dernière ligne définit l'étiquette adr_ptr_x comme l'adresse de la section data, plus huit. Elle réserve un mot dont le contenu est l'adresse de la section data, plus 4. Notons au passage que nous aurions pu remplacer l'opérande **adr_x** de la directive **.word** par **debut + 4**.

Si la section data débute à l'adresse 2000_{10} (respectivement 4000), le troisième mot est stocké à l'adresse 2008_{10} (respectivement 4008) et son contenu initial est 2004 (respectivement 4004). L'étiquette adr_x est un nom symbolique de l'adresse du deuxième mot de la section data (soit 2004 ou 4004 selon l'adresse à laquelle débute la section data).

[2] pour une machine 32 bits

[3] en tenant compte de l'ordre de stockage des entiers en mémoire, cf 3.2.3

5.2.3 Directive .short

Outre byte et word, il existe une directive pour chaque autre sous-multiples du mot. Pour réserver des paquets de 16 bits, le langage d'assemblage GNU pour ARM supporte la directive **.short** (synonyme **.hword**).

5.3 Réservation et initialisation de mémoire dans text

Il est possible de décrire tout un programme avec les directives byte, short et word, utilisables de la même manière dans les sections text et data.

La section text est habituellement réservée aux constantes[4] et aux instructions.

Le programmeur peut écrire chaque instruction machine sous forme symbolique. L'assembleur se charge alors de convertir cette description symbolique en une séquence équivalente d'une ou plusieurs directives .word décrivant chacune un mot de la représentation en binaire de l'intruction.

L'exemple suivant calcule dans r3 le quadruple de la valeur de r2, plus 2. La deuxième instruction (**add r3, r3, #1**) incrémente r3 de 1. Elle est ici décrite par son code en hexadécimal, via une directive .word. Réciproquement, à la place de la dernière instruction, nous aurions pu utiliser une directive .word spécifiant son code machine (0xe2833003 pour add r3, r3, r3).

```
                                    @ calcul de 4r2 +2
          .text
fois2:     add       r3, r2, r2         @ r3 <- 2 x r2
plus_un:   .word     0xe2833001         @ ou add   r3, r3, #1  : r3 <- r3 + 1
re_fois2:  add       r3, r3, r3         @ ou .word 0xe2833003
```

Notons qu'il est possible d'associer plusieurs étiquettes à la même adresse, ce qui est très utile dans la traduction des contructions algorithmiques. Dans l'exemple suivant, les étiquettes corps_while et debut_if (respectivement fin_if et test_while) sont toutes les deux associées à l'adresse de la section text plus quatre (respectivement plus 24).

```
                b        test_while         @ x : r0
corps_while:                                @ y : r1
debut_if:       cmp      r0, r1             @ while (x != y)
                ble      sinon              @ if (x > y)
alors:          sub      r0, r0, r1         @    x = x - y;
                b        fin_if             @ else
sinon:          sub      r1, r1, r0         @    y = y - x;
fin_if:
test_while:     cmp      r0, r1
                bne      corps_while
```

5.4 Réservation de mémoire sans valeur initiale

La directive **.skip** permet de réserver n octets consécutifs. Elle ne specifie pas de valeur initiale : les octets seront implicitement initialisés à 0.

[4] Les constantes peuvent cohabiter avec les instructions dans la section text, ou être regroupées dans une section de données initialisées protégée contre les écritures (rodata).

La directive skip est utilisable pour toutes les déclarations de variables en mémoire sans initialisation. Voici un exemple de réservation de place pour deux variables entières x et y sur 32 bits et 2 variables a et b sur 16 bits.

```
            .data
adr_x:      .skip 4             @ long int x
adr_y:      .skip 4             @ long int y
adr_a:      .skip 2             @ short int a
adr_b:      .skip 2             @ short int b
```

Le même effet aurait pu être réalisée avec une seul directive skip spécifiant 12 octets, mais ne permettrait pas de définir une étiquette pour chaque emplacement.

5.5 Alignement

Considérons l'ensemble de déclarations de variables C suivant.

```
                                    @           .data
short int s1 = 0x1234;              @ s1:       .short    0x1234
char c1 = 'a';                      @ c1:       .byte     'a'
                                    @ sauter ici 3 octets pour X4
long int l = 0x12345678;            @ l:        .word     0x12345678
char c2 = 'b';                      @ c2:       .byte     'b'
                                    @ sauter ici 1 octet pour X2
short int s2 = 3;                   @ s2:       .short    3
```

Prise individuellement, chacune des déclarations C est facile à traduire en langage d'assemblage, comme le montrent les commentaires. En revanche, la traduction de l'ensemble pose un problème d'alignement des demi-mots et des mots (qui doivent être stockés à des adresses respectivement paires et multiples de 4).

Supposons que la section data débute à une adresse multiple de 4 (donc une adresse A telle que A = 4X). La variable C1 sera stockée à une adresse paire (4X+2) et l sera à l'adresse suivante (4X+3), qui n'est pas multiple de 4. Il existe un problème potentiel d'alignement chaque fois qu'une variable est suivie en mémoire par une autre variable d'une taille supérieure (short après byte, word après byte ou short).

Une stratégie consiste à réordonner les déclarations par ordre décroissant de taille pour éviter tout problème d'alignement. Cette stratégie présente l'inconvénient de ne pas respecter l'ordre des déclarations du programme C d'origine.

L'autre technique consiste à sauter le nombre d'octets nécessaires pour rétablir l'alignement. Dans l'exemple, il convient d'insérer une réservation de 3 octets (**.skip 3**) entre c1 et l, pour que l'adresse l redevienne un multiple de 4 , et une réservation d'un octet (**.skip 1**) entre c2 et s2 de telle sorte que l'adresse s2 soit paire.

Supposons que nous insérions après coup une variable c3 de type char après c1 : il ne faudra laisser plus que deux octets d'alignement avant l. La vérification manuelle des contraintes d'alignement, fastidieuse, est simplifiée par l'existence d'une directive d'alignement : **.align**.

La directive **.align T** saute automatiquement le bon nombre d'octets pour aligner l'adresse de l'objet qui suit align sur un multiple de T. Dans notre exemple initial, il suffit d'insérer une directive **.align 4** (respectivement **.align 2**) devant tout réservation .word (respectivement .short) précédée d'une réservation de taille inférieure.

Dans l'exemple initial, la déclaration de l sera précédée de **.align 4** qui générera l'équivalent de **.skip 3** et celle de s2 suivra **.align 2** dont l'effet sera identique à **.skip 1**.

Attention : pour le processeur ARM, il faut utiliser la variante balign car la directive GNU align pour ARM aligne sur 2^n au lieu de n.

5.6 Réservation et initialisation de chaînes

Les directives **.asciz** et **.ascii** permettent de réserver de la place pour les chaînes de caractères, respectivement avec et sans marque de fin de chaîne.

```
          .data                          @ équivalent avec .byte
aurevoir: .asciz          "bye"          @ aurevoir:  .byte 'b'
                                         @            .byte 'y'
                                         @            .byte 'e'
                                         @            .byte 0   (ou .byte '\0')
salut:    .ascii          "hop"          @ salut:     .byte 'h'
                                         @            .byte 'o'
                                         @            .byte 'p'
```

5.7 Contenu d'un fichier exécutable relogeable et section bss

5.7.1 Contenu d'un fichier objet

Un fichier objet issu de l'assemblage comprend un en-tête, le contenu initial de la section text, celui de la section data, les tables de réimplantation et une table des symboles.

L'en-tête du fichier contient les rensignements suivant :

- des informations spécifiques permettant de reconnaître le format de fichier exécutable utilisé (typiquement ELF pour un unix ou linux récent),
- des informations sur la nature de la machine auquel il est destiné (par exemple un ARM 32 bits configuré en mode big endian),
- la position respective des tables et des contenus de text et data dans le fichier,
- la taille des sections text, data et bss,
- le point d'entrée, autrement la position (dans la section text) de l'instruction par laquelle l'exécution doit commencer.

La table des symboles recense les définitions des étiquettes. Elle est utilisée par l'éditeur de liens lors de la fusion de fichiers compilés séparément ou par les débogueurs (tels que gdb).

Il existe une table de réimplantation pour le contenu de la section text et une pour celui de data. Ces tables indiquent les corrections à appliquer au contenu initial de text et data en fonction des adresses de chargement des sections text, data et bss.

Le lancement de l'exécution d'un programme met en jeu les étapes suivantes :

1. ouverture du fichier et lecture de l'en-tête,
2. allocation de portions de mémoire centrale aux sections text, data et bss,
3. recopie du contenu initial des sections text et data du fichier exécutable en mémoire centrale
4. forçage à 0 de tout le contenu de bss
5. application des corrections spécifiées dans les tables de réimplantation au contenus de text et de data,
6. initialisation du compteur ordinal avec l'adresse du point d'entrée du programme : l'exécution des instructions du programme commence.

5.7.2 Bss : une section destinée aux variables non initialisées

Le fichier exécutable ne contient pas de copie du contenu initial de bss, qui est implicitement 0 pour tous les octets. L'utilisation des directives byte, short et word dans la section bss est donc prohibée.

En théorie, on peut se passer de section bss. Toutes les variables (y compris les tableaux) peuvent être stockées dans la section data, même si elles sont déclarées sans initialisation, mais au détriment de l'espace disque.

En effet, le fichier exécutable contient une copie de la valeur initiale de chaque octet de la section data, même si celle-ci est nulle. Supposons que l'on déclare un tableau d'un million d'éléments de type entier 32 bits, sans valeur initiale.

Si le tableau est déclaré dans la section data, le fichier exécutable contiendra une copie du contenu initial du tableau, donc quatre millions d'octets à 0. Si le même tableau est déclaré dans la section bss, l'en-tête du fichier exécutable indiquera simplement que la taille de la section bss est de quatre mégaoctet. Mais le fichier exécutable ne contiendra pas les quatre millions d'octets à 0 : lors du lancement de l'exécution, le système d'exploitation se contentera d'exécuter une boucle d'initialisation à 0 de la section bss en mémoire centrale.

En théorie, la section bss pourrait ne contenir qu'une unique directive skip spécifiant en une seule fois la taille totale bss. Il est cependant intéressant d'utiliser autant de directives skip que de déclarations de variables sans initialisation : celà permet d'associer une étiquette à chaque emplacement de variable. Ces étiquettes seront utilisées comme contenus des pointeurs de variables initialisés, présents dans la section data ou dans les instructions de la section text.

Chapitre 6

Variables et pointeurs, opérateurs * et &

Les variables des programmes C peuvent être stockées dans les registres du processeur ou dans la mémoire centrale (dans les sections data ou bss).

Dans le chapitre 2 , nous avons vu comment gérer les expressions et affectations C n'utilisant que des variables stockées dans des registres du processeur.

La traduction des déclarations de ces variables se limite à de simples commentaires indiquant quels registres du processeurs sont alloués à quelles variables. Après réécriture dans un format C intermédiaire faisant apparaître les temporaires, la traduction en langage d'assemblage est triviale, chaque opération en C se traduisant par une instruction machine de calcul dont les opérandes et le résultat sont stockés dans des registres.

Ce chapitre détaille au contraire les aspects spécifiques au stockage des variables en mémoire, la méthode de décomposition des expressions en opérations élémentaires étant supposée acquise.

6.1 Processeur RISC fictif de référence

Sauf précision contraire, nous utiliserons le langage machine et d'assemblage d'un processeur RISC 32 bits fictif, légèrement simplifié par rapport aux familles de processeurs RISC réelles, et fortement inspiré de la famille ARM. Ce processeur dispose de trente-deux registres généraux de travail, notés $r0$ à $r31$

Toutes les instructions, à l'exception de mov32, sont codées sous la forme d'un mot de 32 bits encodant la nature de l'instruction, les numéros de registres utilisés et éventuellement une petite constante entière. L'instruction mov32 (en fait une instruction load immédiat) charge une entière constante quelconque dans un registre. Mov32 occupe deux mots de 32 bits, le deuxième contenant la constante à transférer dans le registre[1].

Toutes les instructions de calcul existent en deux versions : avec et sans mise à jour des indicateurs NZCV (exemple : sub sans mise à jour, subS avec). Elles prennent leurs opérandes et déposent leur résultat dans l'un des seize registres généraux (notés r0 à r15) du processeur. Il existe une version réduite d'adressage immédiat pour l'opérande droit qui peut être au choix un registre ou une petite constante entière codée 8 bits.

[1]Nous détaillerons ultérieurement comment charger une constante 32 bits quelconque dans un registre d'un processeur SPARC ou ARM, dont toutes les instructions sans exception sont codées sur un seul mot.

Les instructions arithmétiques permettent de réaliser l'addition, la soustraction et la soustraction inversée (opérande droit - opérande gauche) avec (addc, subc, rsubc) ou sans (add,sub,rsub) prise en compte de C comme retenue initiale. Après une soustraction, C indique la retenue finale, comme pour un processeur ARM.

Le jeu d'instructions comprend aussi les décalages logique à gauche (lsl), à droite (lsr) et arithmétique à droite (asr), ainsi que les opérations bit à bit et (and), ou (or) et ou exclusif (eor).

L'instruction move not (mvn) n'a pas d'opérande gauche. Elle copie le complément à 1 de l'opérande droit dans le registre résultat. Sa variante move (mov) copie sans complémenter. Comme pour les instructions de calcul, l'adressage immédiat est limité à des constantes codables sur 8 bits.

L'instruction cmp est une forme de subS qui n'utilise pas de registre résultat : elle met à jour les indicateurs comme la soustraction, mais ne stocke pas le résultat apparent de l'opération dans un registre.

Les instructions d'accès à la mémoire sont load (ldr) et store (str). L'adresse de load ou store, notée entre crochets, est la somme ou la diférence entre le contenu d'un registre général d'une part et le contenu d'un autre registre général ou une petite constante immédiate entière codable sur 8 bits d'autre part. Cette constante est supposée nulle si les crochets ne contiennent qu'un registre.

La taille des variables stockées en mémoire est le mot ou le sous-multiple du mot, celle des registres est de 32 bits. La lecture en mémoire des sous-multiples du mot réalise une extension à 32 bits du format de représentation[2] et l'écriture une réduction par troncature (élimination des bits de poids forts excédentaires[3]). Les variantes de tailles pour load et store sont : les mots de 32 bits (ldr, str), les entiers sur 16 bits (ldrh pour les entiers naturels, ldrsh pour les entiers signés, strh) et les octets (ldrb, ldrsb, strb).

Il existe des variantes de ldr et str avec préincrémentation et postincrémentation, présentées à la fin de ce chapitre.

Il existe aussi des variantes de load et store sur 32 bits pour effectuer des transferts de contenu entre une liste ordonnée de registres et un ensemble d'emplacements mémoire contigüs. L'utilisation de ces instructions (ldm et stm) est présenté dans le chapitre 10.

```
@ Cette instruction range en mémoire les registres
@ r0, r2, r3, r4, r5, r6 et r8 dans une zone mémoire
@ pointée par r12. Les registres sont rangés par ordre
@ croissant de numéros.

stm  {r0, r2-r6, r10}, [r12]

@ Elle remplace cette suite d'instructions :

str  r0 ,[r12,#0]
str  r2 ,[r12,#4]
str  r3 ,[r12,#8]
str  r4 ,[r12,#12]
str  r5 ,[r12,#16]
```

[2]ajout en poids fort de 0 pour un entier naturel ou recopie du bit de signe pour un entier signé

[3]quelque soit la nature de l'entier transféré

```
        str  r6 ,[r12,#20]
        str  r10,[r12,#24]

        @ Le transfert en sens inverse s'écrit :

        ldm  {r0, r2-r6, r10}, [r12]
```

Les instructions de branchement seront détaillées au chapitre 8.

6.2 Traduction des déclarations de variable en mémoire

6.2.1 Exemple de déclarations de variables en C

```
unsigned char c3 = 'a'
char cs1;
unsigned short int s1;
short int ss1, ss2;
unsigned short int s3 = 0x1234;
unsigned short int s4 = 3;
short int ss3;
unsigned long int l1, l2;
long int ls1,ls2;
unsigned long int l3 = 0x12345678;
unsigned long int l4 = 1;
long int ls3 = -2;
```

6.2.2 Principe de traduction

Chaque déclaration de variable C (sans attribut register, donc supposée stockée en mémoire) sans initialisation est traduite par une directive skip de réservation dans la section bss d'autant d'octets que la taille du type de la variable.

Chaque déclaration avec initialisation sera traduite par une directive (byte, short, word, ascii, asciz) de réservation de place dans data spécifiant la valeur initiale de la variable.

Dans les deux cas, la directive de réservation sera accompagnée d'une définition d'étiquette donnant un nom symbolique à l'adresse de stockage de la variable.

En présence de plusieurs variables, la principale précaution à prendre concerne le respect des contraintes d'alignement, détaillées au chapitre 5.

Notons que pour la réservation de place, seule la taille (cf l'opérateur C sizeof) de la variable compte, (la nature de la variable est sans importance). A titre d'exemple la directive **.skip 4** réserve un emplacement pour n'importe quelle variable codée sur 32 bits, qu'elle soit de type float, long int ou unsigned long int, ou encore de type pointeur.

6.2.3 Sections data et bss de l'exemple

Examinons l'application de ce principe sur l'exemple précédent. Chaque directive de réservation de place peut être précédée d'une directive d'alignement sur une adresse multiple de sa taille,

mais cette précaution n'est nécessaire que si la réservation précédente est de taille inférieure.

Notons au passage que l1 étant précédé de 8 octets (char, octet d'alignement, 3 fois short), il serait naturellement à une adresse multiple de 4.

Rappelons également que si les variables C étaient déclarées dans l'ordre décroissant de tailles, il n'y aurait aucun octet d'alignement perdu dans data et bss.

```
                .data
adr_de_c3:      .byte   'a'
                .align   2          @ saute 1 octet devant short qui suit
adr_de_s3:      .short   0x1234
adr_de_s4:      .short   3
                .align   4          @ saute 2 octets devant long qui suit
adr_de_l3:      .word    0x12345678
adr_de_l4:      .word    1
adr_de_ls3:     .word    -2

                .bss
adr_de_cs1:     .skip   1
                .align  2           @ saute 1 octet devant short qui suit
adr_de_ss1:     .skip   2
adr_de_ss2:     .skip   2
adr_de_ss3:     .skip   2
                .align  4           @ saute 0 octet, pourrait être omis
adr_de_l1:      .skip   4
adr_de_l2:      .skip   4
adr_de_ls1:     .skip   4
adr_de_ls2:     .skip   4
```

Pour bien faire apparaître dans cet exemple que les étiquettes sont des noms symboliques des **adresses** des variables, leurs noms sont tous préfixés par adr_de_.

6.3 Opérateur &, * et type adresse

6.3.1 Opérateur & : "adresse de"

L'opérateur C & donne l'adresse de son opérande : si v est une variable stockée en mémoire, &v désigne l'adresse à laquelle est est stockée. En d'autres termes, appliqué à un opérande de type Mem[adr], il permet d'en désigner l'adresse adr.

La déclaration d'une variable C de type T sans attribut register a deux effets :
- réserver (et éventuellement spécifier une valeur initiale différente de 0) sizeof(T) unités adressables (donc T octets) et
- associer la constante symbolique &v à l'adresse du premier octet réservé.

Soit XX le nom d'une variable de notre exemple. Par définition, il est clair que la constante &XX en langage C et l'étiquette adr_de_XX en langage d'assemblage sont deux noms symboliques de l'adresse du premier octet réservé pour stocker XX.

A titre d'exemple &ss1 en C et l'étiquette adr_de_ss1 en langage d'assemblage sont deux noms symboliques de l'adresse du troisième octet de la section bss, (soit 1002 si la section bss débute à l'adresse 1000).

6.3.2 Affectation et opérateur *

Appliqué à une adresse adr, l'opérateur * permet de désigner le contenant mémoire Mem[adr] à cette adresse. L'opérande adr peut être une constante ou une expression (qui sera évaluée pour calculer l'adresse). L'opérateur unaire[4] ***** fonctionne à l'inverse de celui de l'opérateur &.

Une variable **var** stockée en mémoire peut être désignée en utilisant l'opérateur * sur son adresse (&var) et par définition, **var** est synonyme de***&var**. Sa signification est la suivante : **Mem[adresse_de_stockage_de_var]**.

Une affectation est de la forme **membre_gauche = membre_droit**. Membre_gauche spécifie un contenant qui prend pour nouveau contenu le résultat de l'évaluation du membre droit. Un contenant est soit un registre (pour les variables de type register), soit un emplacement mémoire identifié par son adresse.

Membre_droit est une expression composée d'opérateurs de calcul, de constantes et de contenants. L'affectation évalue l'expression en appliquant les opérateurs de calcul sur les contantes et sur les contenus des contenants appartenant à membre_droit.

A titre d'exemple, s1 et s3 étant stockées en mémoire, l'affectation **s1 = s3 - 2** peut ausi s'écrire sous la forme C détaillée ***&s1 = *&s3 - 2**. On pourrait l'écrire plus explicitement *** (short int *) &s1 = *(short int *) &s3 - (short int) 2** pour faire apparaître les types spécifiés par les déclarations de ss1 et ss3.

Cette dernière notation C signifie **Mem[adresse_de_s1]** prend pour nouvelle valeur **Mem[adresse_de_s3] - 2** : lire le contenu (entier relatif sur 2 octets) de la mémoire à l'adresse à laquelle est stockée s3, retancher la constante (entière sur 16 bits) 2 et écrire le résultat (entier sur 2 octets) en mémoire à l'adresse à laquelle est stockée s1.

L'opérateur unaire * tout à gauche du membre gauche de l'affectation correspond à une écriture (du résultat de la soustraction) en mémoire (dans la variable affectée).

Le ou les autres opérateurs unaires * présents dans la forme détaillée de l'affectation correspondent à des lectures (du contenu des variables utilisée dans l'expression).

L'affectation prise en exemple pourrait être traduite en langage d'assemblage par en une instruction CISC de calcul spécifiant un mode d'adressage absolu pour l'opérande gauche ainsi que pour le résultat et un mode d'adressage immédiat pour l'opérande droit : **sub [s1], [s3], #2**.

Pour notre processeur RISC fictif, nous devons décomposer le travail en plusieurs étapes : charger les adresses s1 et s3 dans des registres, lire en mémoire le contenu de s3, effectuer la soustraction, écrire en mémoire le résultat dans s1.

Cette traduction suppose l'introduction de variables intermédiaires de type pointeur, dont nous allons au préalable présenter la syntaxe de déclaration.

[4] unaire : à seul opérande

6.3.3 Type "adresse de"

Notons que l'opérande * utilise deux informations : l'adresse et le type d'objet à accéder à cette adresse. Le type d'élément pointé définit le nombre d'octets à lire ou à écrire et la manière d'en interpréter le contenu.

Les constantes adresses sont du type "adresse de", mais il existe autant de variantes de types "adresse de" que de types d'élément adressable.

Il semblerait logique de noter (& T) le type "adresse d'élément de type T". Cette notation a été retenue par les concepteurs du langage C++ (extension orientée objet de C) pour le type "référence à un objet de la classe T".

Les concepteurs du langage C ont préféré noter le type "adresse d'élément de type T" (T *). La justification de cette notation est la suivante : en appliquant l'opérateur * à une entité de ce type, on obtient un élément de type T.

Notons que tous les types "adresse de" ont la même taille : celle d'une adresse, à ne pas confondre avec la taille de l'objet pointé. Ainsi, pour une machine 32 bits **sizeof(char) = 1** et **sizeof(long) = 4**, mais **sizeof(long *) = sizeof(char *) = 4**.

6.3.4 Conversion de type pointeur

Le forceur de type (T *) ne modifie pas la constante adresse ou le contenu du pointeur auquel il s'applique, mais indique au compilateur de considérer cette adresse comme l'adresse d'une entité de type T.

Le type **(void *)** représente le type "adresse d'un élément de type non spécifié". Mais on ne peut appliquer l'opérateur * sur une variable ou une constante de type (void *) : le type (void *) doit d'abord être converti en (T *), T étant un type autre que void.

Soit une primitive d'allocation dynamique de mémoire retournant un pointeur de type **void *** ou **char *** : ce pointeur sera converti en **T *** lors de l'allocation d'un objet de type T.

6.3.5 Constante NULL

L'opérande de l'opérateur unaire * doit être l'adresse d'une entité du programme (variable ou procédure). Elle doit appartenir à une des sections mémoire du programme exécuté (text, data, bss ou tas[5] , pile) : une variable stockée dans un registre n'a pas d'adresse.

Le langage C définit un nom de constante adresse invalide : NULL. NULL est utilisée pour indiquer qu'un pointeur ne répère aucun élément. L'application de l'opérateur unaire * à NULL est illégale.

NULL est le plus souvent définie comme **(void *) 0**. Tout pointeur stocké dans bss (implicitement initialisée à 0) sera donc automatiquement initialisé à NULL et le matériel est généralement capable de générer une interruption au premier accès à l'adresse 0, qui a pour effet d'arrêter immédiatement l'exécution de programme.

[5]Le tas est la zone utilisée pour l'allocation dynamique de mémoire (fonction malloc). Le tas constitue le plus souvent une extension de la section bss qui croît avec les demandes d'allocation.

6.4 Variables pointeur stockées en registre

On appelle variable pointeur de type T une variable de type (T *) : elle contient une adresse (d'un élément supposé être) de type T.

Comme toute variable, une variable de type pointeur peut être stockée dans un registre (nous la déclarerons alors avec l'attribut de stockage register) ou en mémoire. Dans ce chapitre, nous n'utiliserons que des pointeurs stockés dans des registres.

Après affectation de l'adresse &v d'une variable v à un pointeur p, p repère v : v peut être accédée en appliquant l'opérateur * à p. V et *p (en remplaçant p par son contenu) équivalent en C à *&v et désignent tous les deux l'emplacement mémoire (Mem[&v]) réservé au stockage de v.

Voici un exemple de code C dans lequel le pointeur ptrss repère successivement et permet de manipuler les variables ss1, puis ss3.

```
unsigned short int ss1, ss2, ss3;
register unsigned short int rss;
register unsigned short int *ptrss; /* un pointeur d'entier court     */
ptrss = &ss1                        /* ptrss repere ss1              */
*ptrss = 2;                         /* ss1 = 2  :  *(&ss1) =2        */
ss2 = ss1;                          /* ss2 = ss1 : *&ss2 = *&ss1     */
ss3 = *ptrss;                       /* ss3 = ss1 : *&ss3 = *(&ss1)   */
ptrss = &ss3;                       /* ptrss repere ss3              */
*ptrss += 3;                        /* ss3 +=3   : *&ss3 = *&ss3 + 3 */
```

Trois opérations élémentaires peuvent être réalisées sur une variable pointeur stockée dans un registre :

1. affecter l'adresse d'une variable à un pointeur : **ptrss = &ss1**,
2. lire : prendre une copie du contenu de la variable pointée (par exemple pour l'affecter à une variable stockée dans un registre) : **rss = *ptrss**,
3. écrire : affecter à la variable pointée un nouveau contenu (par exemple celui d'une variable stockée dans un registre) : ***ptrss = rss**.

```
@ r5 : ptrss
@ r2 : rss

@ exemple d'affectation d'une adresse  : ptrss = &ss1
mov32   r5, #adr_de_ss1      @ adresse : ldr 32 bits

@ exemple de lecture : rss = *ptrss
ldrsh   r2, [r5]             @   sh : signed halfword

@ exemple d'écriture : *ptrss = rss
strh    r2, [r5]             @    h : halword
```

Chacune de ces opérations se traduit en une seule instruction machine de notre processeur RISC :

1. Une instruction mov32 pour charger une constante adresse sur 32 bits dans un registre. En langage C la constante adresse &ss1 correspond à l'étiquette adr_de_ss1 en langage d'assemblage.

2. L'instruction de lecture est de type load avec adressage registre indirect, correspondant à la taille et au type d'objet pointé (entier naturel court signé dans le cas de ptrss).
3. L'instruction d'écriture est de type store avec adressage registre indirect, correspondant à la taille d'objet pointé.

Rappelons que le format des entiers stockés sur 8 ou 16 bits est étendu à la taille d'un mot lors de leur lecture de la mémoire dans un registre, en tenant compte de la nature de l'entier (naturel ou signé).

6.5 Décomposition d'une affectation en instructions RISC

Pour guider la traduction des instructions C en langage d'assemblage, nous utiliserons uen forme de C intermédiaire équivalente (et acceptée par le compilateur) mettant en évidence la décomposition en étapes élémentaires et l'utilisation de registres de stockage d'informations temporaires. Chaque instruction C de cette forme intermédiaire correspond à une instruction machine de notre processeur RISC virtuel.

Dans une expression n'utilisant que des variables stockées dans des registres, les temporaires ne servent qu'à stocker des résultats de calcul intermédiaire (cd 2).

Dans le cas général, des registres sont utilisés pour stocker temporairement les adresses et les contenus de variables rangées en mémoire.

Considérons à titre d'exemple les deux affectations suivantes.

```
/* affectations a traduire */
ss1 = 2;                /* decomposee en *&ss1 = 2 */
ss3 = ss1 - ss2;        /* decomposée en *&ss3 = *&ss1 - *&ss2 */
```

Voici la version décomposant ces affectations en instructions C plus élémentaires.

```
short int  r1, r2, r3;          /* --> r1, r2, r3 de la machine */
short int *rp0;                 /* --> r0 de la machine */

r1 = 2;
rp0 = &ss1;
*rp0 = r1;

rp0 = &ss1;                     /* peut etre omis : deja fait */
r1  = *rp0;                     /* inutile : r1 contient deja Mem[rp0] */
rp0 = &ss2;
r2 = *rp0;
r3 = r1 - r2;
rp0 = &ss3;
*rp0 = r3;
```

Chaque instruction C de la forme intermédiaire a une traduction directe en langage d'assemblage.

```
@ r1, r2, r3 temporaires pour contenus de ss1, ss2, ss3
@ r0         temporaire  pour adresses de ss1, ss2, ss3
```

```
mov32 r1, #2              @ r1 = 2
mov32 r0, #adr_de_ss1     @ rp0 = &ss1
strh  r1, [r0]            @ *rp0 = r1

mov32 r0, #adr_de_ss1     @ rp0 = &ss1
ldrsh r1, [r0]            @ r1 = *rp0
mov32 r0, #adr_de_ss2     @ rp0 = &ss2
ldrsh r2, [r0]            @ r2 = *rp0
sub   r3,r1,r2            @ r3 = r1 - r2
mov32 r0, #adr_de_ss3     @ rp0 = &ss3
strh  r3, [r0]            @ *rp0 = r3
```

La consommation de registres peut être optimisée en examinant finement les instants auxquels ils contiennent simultanément une information utile. Voici à tire d'exmple une traduction de l'affectation **ss3 = ss1 - ss2 ;** qui n'utilise que deux registres.

```
mov32 r1, #adr_de_ss1
ldrsh r1, [r1]
mov32 r2, #adr_de_ss2
ldrsh r2, [r2]
sub   r2,r1,r2
mov32 r1, #adr_de_ss3
strh  r2, [r1]
```

Le registre r1 est utilisé pour stocker l'adresse de ss1 jusqu'à la fin du cycle de lecture mémoire spécifié par la première instruction ldrsh. Lorsque le cycle de lecture mémoire réalisé par ldrsh se termine, l'adresse contenue dans r1 n'est plus nécessaire et peut être remplacée dans r1 par le contenu de ss1 lu en mémoire. De même, à la fin du calcul de la différence entre les contenus des deux registres, le résultat peut prendre la place d'un des opérandes (dans r2).

6.6 Variables pointeur stockées en mémoire

Les pointeurs sont des variables C comme les autres. En l'absence de l'attribut de stockage register, une variable pointeur est stockée en mémoire et possède une adresse.

La variable pointeur est stockée dans la section data si la déclaration spécifie une valeur initiale[6] et dans la section bss sinon.

Une variable pointeur stockée en mémoire a une adresse, que l'on peut donc éventuellement stocker dans une variable pointeur (de pointeur). La déclaration d'une variable ppt pointeur de pointeur de T est de forme **T**ppt = &var_de_type_T***, la spécification de valeur initiale (= &var_type_T*) étant facultative.

Le principe de traduction des accès aux variables stockées en mémoire s'applique de la même manière aux variables pointeurs stockées en mémoire.

6.6.1 Exemple d'utilisation de pointeurs stockés en mémoire

Illustrons la gestion de pointeurs stockés en mémoire sur un exemple simple.

[6] Cette valeur initiale étant l'adresse d'une autre variable

```
char x = 2;
char y;
char *ptr_init = &x;   /* pointeur initialise dans la declaration    */
char *ptr;             /* pointeur non initialise                    */
...
y = *ptr_init + 1;     /* equivaut a y = x + 1                       */
x = x + 4;             /* la valeur pointée peut changer             */
ptr = ptr_init;        /* copie de pointeur : ptr repère egalement x */
*ptr += 5;             /* equivaut a x = x + 5                       */
ptr = &y;              /* ptr repère y                               */
*ptr = *ptr_init;      /* equivaut a y = x;                          */
```

La première étape préalable à la traduction en langage d'assemblage de notre processeur RISC fictif consiste à remplacer dans les instructions chaque occurence d'une variable v stockée en mémoire par *&v. Dans notre exemple, cette étape concerne quatre variables : x, y, ptr_init et ptr.

```
/* déclarations omises */
*&y = **&ptr_init + 1;  /* Mem[&y] = Mem[Mem[&ptr_init]]              */
*&x = *&x + 4;          /* Mem[&x] = Mem[&x] + 4                      */
*&ptr = *&ptr_init;     /* Mem[&ptr] = Mem[&ptr_init]                 */
**&ptr += 5;            /* Mem[Mem[&ptr]] = Mem[Mem[&ptr]] +5         */
*&ptr = &y;             /* Mem[&ptr] = &y                             */
**&ptr = **&ptr_init;   /* Mem[Mem[&ptr] = Mem[Mem[&ptr_nit]]         */
```

6.6.2 Introduction des temporaires et traduction

L'étape suivante consiste à introduire les temporaires stockés dans des registres pour stocker les contenus de variables entières (contenus de x et y), les contenus des pointeurs (ptr, ptr_init) et les adresses des pointeurs (&ptr, &ptr_init), puis à décomposer en instructions élémentaires.

```
                                          .data
char x = 2;                       adr_x:          .byte    2
char y;                                           .align   4
char *ptr_init = &x;              adr_ptr_init:   .word adr_x
char *ptr;
                                                  .bss
                                  adr_y:          .skip    1
                                                  .align   4
                                  adr_ptr:        .skip    4

register char t0;                 @ t0 : registre r0
register char *t2 ,  *t3;         @ t2 : registre r2
                                  @ t3 : registre r3
register char **t4, **t5;         @ t4 : registre r4
                                  @ t5 : registre r5
```

Dans cet exemple, quatre temporaires pointeurs ont été introduits, tous stockés dans des registres : deux pointeurs de short (t2 et t3) et deux pointeurs de pointeurs de short (t4 et t5).

```
/* *&y = **&ptr_init + 1 */                    .text
t4 = &ptr_init;                               mov32   r4, #padr_tr_init
t2 = *t4;                                     ldr     r2, [r4]
t0 = *t2;                                     ldrsb   r0, [r2]
t0 = t0 + 1;                                  add     r0, r0, #1
t1 = &y;                                      mov32   r1, #adr_y
*t1 = t0;                                     strb    r0, [r1]

/* *&x = *&x + 4 */
t2 = &x;                                      mov32   r2, #adr_x
t0 = *t2;                                     ldrsb   r0, [r2]
t0 = t0 + 4;                                  add     r0, r0, #4
*t2 = t0;                                     strb    r0, [r2]

/* *&ptr = *&ptr_init */
t4 = &ptr_init;                               mov32   r4, #adr_ptr_init
t2 = *t4;                                     ldr     r2, [r4]
t5 = &ptr;                                    mov32   r5, #adr_ptr
*t5 = t2;                                     str     r2, [t5]

/* **&ptr += 5 */
t4 = &ptr;                                    mov32     r4, #adr_ptr
t2 = *t4;                                     ldr     r2, [r4]
t0 = *t2;                                     ldrsb   r0, [r2]
t0 = t0 + 5;                                  add     r0, r0, #5
*t2 = t0                                      strb    r0, [r2]

/* *&ptr = &y */
t2 = &y;                                      mov32   r2, #adr_y
t4 = &ptr;                                    mov32   r4, #adr_ptr
*t4 = t2;                                     str     r2, [r4]

/* **&ptr = **&ptr_init */
t4 = &ptr_init;                               mov32   r4, #ptr_init
t2 = *t4;                                     ldr     r2, [r4]
t0 = *t2;                                     ldrb    r0, [r2]
t5 = &ptr;                                    mov32   r5, #adr_ptr
t3 = *t5;                                     ldr     r3, [r5]
*t3 = t0;                                     strb    r0, [r3]
```

6.7 Symboles étiquettes sans préfixe adr_

Jusqu'à présent nous avons préfixé les symboles étiquettes qui correspondent à des adresses de variables (&var en C) par adr_. Dans la suite du document nous abandonnerons ce préfixe qui s'appliquerait à toutes les étiquettes. Mais le lecteur devra se souvenir que l'étiquette x en langage d'assemblage est l'adresse de stockage de x (et non sa valeur) et correspond à &x en C.

6.8 Pointeurs de pointeurs en mémoire

L'exemple précédent a illustré l'utilisation de pointeurs de pointeurs stockés dans des registres. Voici maintenant un exemple d'utilisation de pointeurs de pointeurs stockées en mémoire.

6.8.1 Programme à traduire

```
char c = 'a';
char d;
char *ptr_c = &c;                    /* ptr_c repère c                        */
char *ptr_car;                       /* un autre pointeur de char             */
char **ptr_ptr_car;                  /* un pointeur de pointeur de char       */
register char **reg_ptr_ptr;
                                     /* le meme dans un registre              */
...
c = c + 1;                           /* equivaut a c = 'b'                    */
*ptr_c ++;                           /* manipulation par pointeur : c = 'c'   */
ptr_car = &d                         /* ptr_car repère d                      */
*ptr_car = c + 2;                    /* manip de d par ptr : d = 'e'          */
reg_ptr_ptr_car = &ptr_car;          /* reg_ptr_ptr_car repère ptr_car        */
ptr_ptr_car = &ptr_car;              /* ptr_de_ptr_car  repère ptr_car        */
ptr_c = *ptr_ptr_car;                /* ptr_c affecte par pointeur            */
                                     /* ptr_c repère maintenant d             */
ptr_car = &c;                        /* ptr_carx repère c                     */
**ptr_ptr_car = 'x';                 /* c modifie via un ptr ptr char         */
d = **reg_ptr_ptr_car;               /* d = c par reg ptr ptr char            */
```

6.8.2 Mise en évidence des accès à la mémoire

```
*&c = *&c + 1;                       /* 1 */
**&ptr_c = **&ptr_c +1;              /* 2 */
*&ptr_car = &d;                      /* 3 */
**&ptr_car = *&c + 2;                /* 4 */
reg_ptr_ptr_car = &ptr_car;          /* 5 */
*&ptr_ptr_car = &ptr_car;            /* 6 */
*&ptr_c = **&ptr_ptr_car;            /* 7 */
*&ptr_car = &c;                      /* 8 */
***&ptr_ptr_car = 'x';               /* 9 */
*&d = **reg_ptr_ptr_car;             /* 10 */
```

6.8.3 Traduction en langage d'assemblage RISC

```
   char c = 'a';                                .data
   char *ptr_c = &c;            c:              .byte  'a'
                                                .align  4
                                ptr_c:          .word   c
```

```
                                                .bss
char d;                                 d:      .skip 1
                                                .align  4
char *ptr_car;                          ptr_car:        .skip   4
char **ptr_ptr_car                      ptr_ptr_car:    .skip   4

register char **reg_ptr_ptr_car;        @ r8 : reg_ptr_ptr_car;

register char r0;                       @ r0 à r6 : registres de même noms
register char *r1  ,   *r2;
register char **r3 ,  **r4;
register char ***r5, ***r6;


r1 = &c;                                        mov32   r1, #c
r0 = *r1;                                       ldrsb   r0, [r1]
r0 = r0 + 1;                                    add     r0, r0, #1
*r1 = r0;                                       strb    r0, [r1]


r3= &ptr_c;                                     mov32   r3, #ptr_c
r1 = *r3;                                       ldr     r1, [r3]
r0 = *r1;                                       ldrsb   r0, [r1]
r0 = r0 + 1;                                    add     r0, r0, #1
*r1 = r0;                                       strb    r0, [r1]


r1 = &d;                                        mov32   r1, #d
r3 = &ptr_car;                                  mov32   r3, #ptr_car
*r3 = r1;                                       str     r1, [r3]


r1 = &c;                                        mov32   r1, #c
r0 = *r1;                                       ldrsb   r0, [r1]
r0 = r0 + 2;                                    add     r0, r0, #2
r3 = &ptr_car;                                  mov32   r3, #ptr_car
r2 = *r3;                                       ldr     r2, [r3]
*r2 = r0;                                       strb    r0, [r2]


reg_ptr_ptr_car = &ptr_car;                     mov32   r8, #ptr_car



r3 = &ptr_car;                                  mov32   r3, #ptr_car
r5 = &ptr_ptr_car;                              mov32   r5, #ptr_ptr_car
*r5 = r3;                                       str     r3, [r5]


r5 = &ptr_ptr_car;                              mov32   r5, #ptr_ptr_car
r3 = *r5;                                       ldr     r3, [r5]
r1 = *r3;                                       ldr     r1, [r3]
r6 = &ptr_c;                                    mov32   r6, #ptr_c
*r6 = r1;                                       str     r1, [r6]
```

```
   r1 = &c;                                    mov32   r1, #c
   r3 = &ptr_car;                              mov32   r3, #ptr_car
   *r3 = r1;                                   str     r1, [r3]

   r0 = 'x';                                   mov32   r0, #'x'
   r5 = &ptr_ptr_car;                          mov32   r5, #ptr_ptr_car
   r3 = *r5;                                   ldr     r3, [r5]
   r1 = *r3;                                   ldr     r1, [r3]
   *r1 = r0;                                   strb    r0, [r1]

   r1 = *reg_ptr_ptr_car;                      ldr     r1, [r8]
   r0 = *r1;                                   ldr     r0, [r1]
   r2 = &d;                                    mov32   r2, #d
   *r2 = r0;                                   strb    r0, [r2]
```

6.9 Préincrémentation et postincrémentation des pointeurs

Il est possible de combiner les opérateurs **- -** et ++ avec l'opérateur ***** sur les pointeurs. L'opérateur ***** est appliqué sur la valeur initiale du pointeur si l'opérateur **- -** ou ++ est à droite du pointeur ou sur la valeur modifiée s'il est à gauche du pointeur.

```
void copie2 (int *s, int *d, int t)
{
int i;
  *d++ = *s++;                          /*  *d = *s   avant   */
}                                       /*  d++ et s++        */

void copie3 (int *s, int *d, int t)
{
int i;
for {i=t; i>=1; i--}
  *--d = *--s++;                        /*  d-- et s-- avant  */
}                                       /*  *d = *s           */
```

La traduction en langage d'assemblage ne pose pas de problème particulier :

```
@  r0 = *++r1
ldr r0,[r1,#-4]   @ adresse = registre pointeur - 4
sub r1, r1, #4    @ mise à jour registre pointeur

@  r0 = *r1++
ldr r0, [r1]      @ adresse = registre
sub r1, r1, #4    @ mise à) jour registre pointeur
```

Nous supposerons que notre processeur fictif dispose de variantes de ldr et str avec accès mémoire et préincrémentation (**[reg,#ajout] !**) ou postincrémentation (**[ ],#ajout**) du registre d'adresse utilisé, le tout en une seule instruction.

```
@  r0 = *++r1
ldr r0, [r1,#-4]!   : préincrémentation
@  r0 = *r1++
ldr r0, [r1], #-4   : postincrémentation
```

Chapitre 7

Structures et unions

Le type enregistrement, appelé structure en C et record dans d'autres langages, permet de regrouper en un seul objet stocké dans un espace mémoire contigu un ensemble d'objets de types différents, appelés membres de la structure.

7.1 Structures

7.1.1 Syntaxe de déclaration

La déclaration d'objets de type structure est normalement réalisé en deux étapes : définition d'un type de structure et déclaration de variables en utilisant le type défini.

La syntaxe de la définition d'un type de structure est la suivante : le mot clé struct, précédant le nom du type de structure défini et suivi, entre accolades, d'une liste de déclarations de membres de la structure.

```
#define TAILLE_NOM 50
#define TAILLE_PRENOM 30
/* Definition du type struct prsonne */
struct personne {
  char nom[TAILLE_NOM];
  char prenom[TAILLE_PRENOM];
  unsigned short int age;
  unsigned int code_postal;
  ...   /* d'autres membres éventuels */
}
/* Declaration de 3 variables de type struct personne */
struct personne proprietaire, locataire, client;
```

L'utilisation de typedef permet éventuellement d'éviter de répéter le mot struct dans chaque déclaration de variable.

```
typedef struct s_point{
int x;
int y;
} point;
                /* ecriture equivalente */
point px,py;    /* equivaut a struct s_point px, py;   */
```

```
point centre;   /* equivaut a struct s_point centre; */
point sommet;   /* equivaut a struct s_point sommet  */
```

Il est possible de combiner la définition du type de structure et la déclaration des variables en une seule déclaration, le nom du type de structure pouvant éventuellement être omis. Il suffit d'ajouter après la définition de type les noms des variables à déclarer.

La séparation de la déclaration du type structure de la déclaration des variables proprement dite est cependant recommandée. Elle permet d'identifier clairement la définition de type et rend la déclaration des variables plus lisible.

```
typedef enum colori {BLEU, BLANC, ROUGE, JAUNE, NOIR};
/* Declaration combinee du type struct pixel                    */
/* et des variables sommet1, sommet2 et centre                  */
/* ainsi que du pointeur de struct pixel ptr_pixel              */

struct pixel {
  int x;
  int y;
  colori couleur;
  int intensité;
  } sommet1, sommet2, centre, *ptr_pixel;

/* Declaration de deux variables structure cercle1 et cercle2   */
struct {                        /* pas de nom de type struct    */
  int x;
  int y;
float rayon; } cercle1, cercle2;
```

L'omission d'un nom de type de structure est vivement déconseillée : nommer le type de structure permet ensuite de déclarer facilement des champs de type pointeur du type de structure en cours de définition.

```
typedef struct s_doublet {
    long valeur;                    /* contenu */
    struct s_doublet *suivant;      /* pointeur de chaînage */
    } doublet;

doublet cellules [NB_CELLULES];
doublet *tete, *queue;
```

Notons que dans cet exemple, le membre suivant ne pourrait être déclaré par **doublet *suivant**, le type doublet n'étant pas encore défini à ce stade.

L'omission d'un nom de type de structure est également gênant pour la passage de paramètres aux procédures. En l'absence de nom de type, toute la spécification des membres de structure devra être dupliquée, avec tous les risques d'erreur que cela comporte.

```
/* Declaration d'une procédure d'effacement de pixel */
void effacer_pixel(struct pixel *p)
{
p -> intensité = 0;
```

```
}

/* Déclaration d'une fonction acceptant cercle1 ou cercle2         */
/* comme paramètre et retournant la surface                        */
/* L'absence de nommmage du type de structure impose de            */
/* redéclarer les membres de la structure                          */

float aire_du_cercle (struct{int x; int y; float rayon;} *cercle)
{
float surface;
surface = PI * cercle-> rayon * cercle->rayon;
return(surface);
}
```

7.1.2 Structures contenant des structures

Il est possible de construire des structures dont les membres sont eux-mêmes des structures. Dans un logiciel graphique, on peut définir un vecteur par ses extrémités et il peut être commode de regrouper les coordonnées (x,y) de chaque extrémité dans une structure. On peut aussi imaginer qu'une structure décrivant un employé contienne une structure représentant son adresse.

```
struct fleche {
  struct s_point origine;
  struct s_point destination;
  }
```

7.1.3 Initialisation

Une constante structure est une liste de valeurs initiales des membres de la structure, entre accolades et séparées par des virgules. Si un membre est lui-même une structure, sa valeur initiale est elle-même une liste de valeurs initiales entre accolades.

```
#define XINITIAL 4
#define YINITIAL 5

point origine;
point p = {XINITIAL, YINITIAL};
struct s_point q = {6, 2};

point p1 = {3,4};
point p2, p3,
point *ptpoint = &p3;
int z;

struct fleche f1 = {{1,2},{3,4}};
```

7.1.4 Réservation de mémoire, alignement et taille

La réservation de mémoire pour une structure n membres est traitée comme n déclarations de variables de même type que les membres, à ceci près qu'on ne définit pas une étiquette pour

chaque membre.

En revanche, il est commode d'utiliser des constantes symboliques pour repérer la position de chaque membre dans la structure, ainsi que pour la taille totale de la structure.

```
       S_POINT_X=0       /* le membre x est au début de la structure */
       S_POINT_Y=4       /* le membre y est à 4 octets du début      */
       TAILLE_S_POINT=8  /* la structure occupe 8 octets             */

       S_FLECHE_ORIGINE=0
       S_FLECHE_DESTINATION=TAILLE_S_POINT
       TAILLE_S_FLECHE=2*TAILLE_S_POINT

         .data

         XINITIAL=4
         YINITIAL=5
                        /* tout est de type mot : align 4 inutile */
                        /* entre variables                        */
p:       .word  XINITIAL
         .word  YINITIAL

q:       .word  6
         .word  2

p1:      .word  3
         .word  4

f1:      .word  1
         .word  2
         .word  3
         .word  4

ptpoint: .word  p3

         .bss
origine: .skip TAILLE_S_POINT
p2:      .skip TAILLE_S_POINT
p3:      .skip TAILLE_S_POINT
z:       .skip 4
```

La position relative des membres par rapport à l'adresse de la structure est fonction de la taille et des contraintes d'alignement qui s'appliquent aux membres.

```
struct s16et32 {
   short int h;
   long  int w;}

short s = 10000;
struct s16et32 st1 = {0x3344, 0x55667788};
struct s16et32 st2 = {0xaabb, 0xccddeeff};
```

```
/* Déclaration en langage d'assemblage */

        S_S16ET32_H=0
        S_S16ET32_W=4

       .data
s:     .hword  10000
       /* aligner st1 sur une adresse multiple de 4 */
       .align  4                    /* AL1 */
st1:   .hword  0x3344
ici:   .align  4                    /* AL2 */
w1:    .word   0x55667788
       /* l'adresse de st2 est déjà alignée sur un multiple de 4 */
       .align 4                     /* ne saute aucun octet */
st2:   .hword  0x3344
ici2:  .align  4                    /* AL4 */
w2:    .word   0x55667788           /* membre w de st2 */
```

Pour que le nombre d'octets séparant deux membres soit le même dans tous les exemplaires d'un même type de structure, l'adresse d'une structure doit respecter les mêmes contraintes d'alignement que son membre de plus grande taille.

Dans l'exemple précédent, si la directive d'alignement **AL1** était omise, l'adresse **ici** serait déjà un multiple de 4 et la directive **AL2** ne sauterait aucun octet. En revanche l'adresse **ici2** n'est pas un multiple de 4 et la directive **AL3** sauterait 2 octets : **w1 - st1** ne serait pas égal à **w2 - st2**.

La taille d'une structure peut être supérieure à la somme des tailles de ses membres. Appliqué à un type structure, l'opérateur **sizeof** tient compte de tous les octets d'alignement à prévoir pour le stockage d'un ensemble de structures de ce type dans un tableau.

7.1.5 Affectation, opérateurs . et ->

Un membre **m** d'une structure **s** est désigné en suffixant le nom de la structure par l'opérateur point ('**.**') suivi du nom du membre : **s.m**.

Soient deux structures de même type à n membres. L'affectation d'une structure à une autre équivaut à n affectations portant chacune sur un membre de la structure.

Pour manipuler un membre **m** d'une structure repérée par un pointeur **p**, il faut appliquer d'abord l'opérateur *****, puis l'opérateur **.** pour accéder au membre : **(*p).m**. L'opérateur **.** étant prioritaire sur l'opérateur *****, les parenthèses sont obligatoires : ***s.ptr** est interprété comme ***(s.ptr)** et retourne le contenant repéré par le membre ptr (qui doit être de type pointeur) de la structure **s**.

Pour faciliter la manipulation de structures avec des pointeurs (par exemple pour gérer des listes chaînées), on peut utiliser l'opérateur **->** . La notation **p -> m** n'est qu'un raccourci syntaxique de **(*p).m**.

```
p2.x = 3;
p2.y = p1.y;
```

```
p3 = p1;                /* p3.x = p1.x; p3.y = p1.y */

z  = (*ptpoint).x       /* z = p1.x */
ptpoint->y = 5;         /* p1.y = 5  */
ptpoint.x++;            /* p1.x ++ */
```

Les opérateurs de comparaison ne sont pas applicables aux structures : seuls les membres peuvent être comparés.

7.1.6 Traduction en langage d'assemblage des accès aux structures

Comme pour les variables ordinaires, la traduction en langage d'assemblage sera guidée par deux réécritures du programme C d'origine, l'une mettant en évidence tous les accès à la mémoire, l'autre détaillant toutes les utilisations de variables temporaires.

```
*&(p2.x) = 3;
*&(p2.y) = *&(p1.y);
*&(p3.x) = *&(p1.x);
*&(p3.y) = *&(p1.y);
*&z = (**&ptpoint).x;                 /* z = (*ptpoint).x */
(**&ptpoint).y = 5;                   /* (*ptpoint).y = 5 */
(**&ptpoint).x = (**&ptpoint).x + 1;  /* (*ptpoint).x++   */
```

Comme pour les variables ordinaires, chaque opérateur * correspond à une instruction load ou store, l'opérateur . ayant pour effet d'ajouter à l'adresse la position relative du membre par rapport au début de la structure.

```
register int  rint;                   /* r0 */
register int *rpint;                  /* r1 */
register struct point *rpoint;        /* r2 */
register struct point **rppoint;      /* r3 */

rint = 3;                             mov   r0, #3
rpoint = &p2;                         mov32 r2, p2
*rpoint.x = rint;                     str   r0, [r2, #S_POINT_X]

rpoint = &p1;                         mov32 r2, p1
rint = *rpoint.y;                     ldr   r0, [r2, #S_POINT_Y]
rpoint = &p2;                         mov32 r2, p2
*rpoint.y = rint;                     str   r0, [r2, #S_POINT_Y]

rpoint = &p1;                         mov32 r2, p1
rint = *rpoint.x;                     ldr   r0, [r2, #S_POINT_X]
rpoint = &p3;                         mov32 r2, p3
*rpoint.x = rint;                     str   r0, [r2, #S_POINT_X]

rpoint = &p1;                         mov32 r2, p1
rint = *rpoint.y;                     ldr   r0, [r2, #S_POINT_Y]
rpoint = &p3;                         mov32 r2, p3
*rpoint.y = rint;                     str   r0, [r2, #S_POINT_Y]
```

```
rppoint = &ptpoint;                    mov32 r3, ptpoint
rpoint = *rppoint;                     ldr   r2, [r3]
rint = (*rpoint).x;                    ldr   r0, [r2, #S_POINT_X]
rpint = &z;                            mov32 r1, z
*rpint = rint;                         str   r0, [r1]

rint = 5;                              mov   r0, #5
rppoint = &ptpoint;                    mov32 r3, ptpoint
rpoint = *rppoint;                     ldr   r2, [r3]
*rpoint.y = rint;                      str   r0, [r2, #S_POINT_Y]

rppoint = &ptpoint;                    mov   r3, ppoint
rpoint = *rppoint;                     ldr   r2, [r3]
rint = *rpoint.x;                      ldr   r0, [r2, #S_POINT_X]
rint = rint + 1;                       add   r0, r0, #1
*rpoint.x = rint;                      str   r0, [r2, #S_POINT_X]
```

7.2 Unions

Les unions sont des variantes particulières de structures dont les membres sont tous logés à la même adresse en mémoire (à l'adresse de l'union). Elles remplacent les structures dont certain champs ne peuvent pas contenir de valeur simultanément et permettant d'économiser de la mémoire.

Considérons à titre d'exemple une procédure de dessin de flèches. Les extrémités de la flèche peuvent être décrites en coordonnées cartésiennes (x,y) ou polaires (rayon, angle).

```
struct xy {    /* point en coordonnées cartésienne (x,y) */
   int x;
   int y;
          }

struct rt {    /* point en coordonnées polaires (r, theta) */
   float r;
   float t;
          }

struct fleche {
   int polaire;           /* type coordonnées  : polaire si != 0 */
   struct xy orig_xy;     /* coordonnées origine */
   struct rt orig_rt;
   struct xy dest_xy;     /* coordonnées destination */
   struct rt dest_rt;
   }

void fleche (struct fleche f)
{
int xorig,yorig;
int xdest,ydest;
if (f.polaire)
  {
```

```
  /* calculer xorig, xorig a partir de f.orig_rt.r */
  /* calculer xdest, xdest a partir de f.dest_rt   */
  }
else
  {
  xorig = f.orig_xy.x; yorig = f.orig_xy.y;
  xdest = f.dest_xy.x; ydest = f.dest_xy.y;
  }
/* dessiner la flèche */
}
```

L'allocation simultanée de place pour les deux types de coordonnées n'est pas utile. Il convient de définir une union dont les membres sont les coordonnées à usage mutuellement exclusif.

```
union point {
  struct_xy xy;
  struct_rt rt;
            }

struct fleche {
   int polaire;
   union point orig;
   union point dest;
              }

fleche f;

/*  Allocation de mémoire pour f : */
/*  f    : polaire                 */
/*  f+4  : xorig ou rorig          */
/*  f+8  : yorig ou torig          */
/*  f+12 : xdest ou rdest          */
/*  f+16 : ydest ou tdest          */

f.dest.rt.t = 3.0;   /* accès au membre t du membre rt de l'union dest */
f.dest.xy.y = 4;     /* accès au membre y du membre xy de l'union dest */



    S_XY_X=0
    S_XY_Y=4
    TAILLE_S_XY=8

    S_RT_R=0
    S_RT_R=4
    TAILLE_S_RT=8

    /* Dans l'union point, xy et rt sont tous les deux au déplacement 0 */
    S_POINT_XY=0
    S_POINT_RT=0
    TAILLE_S_POINT=8   /* max (TAILLE_S_XY,TAILLE_S_RT) */
```

```
    S_FLECHE_POLAIRE=0
    S_FLECHE_ORIG=4
    S_FLECHE_DEST=12
    TAILLE_S_FLECHE=20 /* sizeof(int) + 2*TAILLE_S_POINT */

    .bss
f:  .skip TAILLE_S_FLECHE

    TROIS_FLOTTANT=0x.....   /* écrire représentation hexa de 3.0 */

    .text
    mov    r0, #TROIS_FLOTTANT
    mov32  r1, f
    str    r0, [r2, #S_FLECHE_DEST+S_POINT_RT+S_RT_T]

    mov    r0, #4
    mov32  r1, f
    str    r0, [r2, #S_FLECHE_DEST+S_POINT_XY+S_XY_Y]
```

Chapitre 8

Sauts et constructeurs algorithmiques

8.1 Notion de saut ou branchement

8.1.1 Définition

Le processeure exécute les instructions séquentiellement (par adresses croissantes) dans l'ordre où elles sont rangées en mémoire. L'instruction courante est par le registre compteur ordinal (PC). L'exécution de l'instruction courante se termine par une incrémentation du compteur ordinal pour pointer sur l'instruction suivante.

Une instruction de branchement (on dit aussi de saut) est une instruction qui affecte au compteur ordinal l'adresse d'une instruction cible (autre que la suivante en mémoire) qui sera exécutée après l'instruction courante de banchement. Une instruction de branchement introduit une rupture de séquence dans le flot d'instructions exécutées.

On parle de branchement en avant lorsque l'adresse de branchement est supérieure à celle de l'instruction qui suit celle de branchement, et en arrière dans le cas contraire. Un branchement en avant saute une séquence d'instructions du programme. Un branchement en arrière réexécute une séquence d'instructions qui précédent l'instruction de saut et sert à réaliser les boucles dans l'exécution du programme.

On parle de branchement absolu lorsque l'instruction de branchement spécifie directement l'adresse à charger dans le compteur ordinal. La destination du branchement reste la même quelque soit l'emplacement de l'instruction de saut. Une instruction de branchement relatif (sous entendu au compteur ordinal) spécifie un déplacement (négatif pour un saut en arrière, positif pour saut en avant) à ajouter au compteur ordinal, donc par rapport à l'emplacement de l'instruction de saut.

Illustrons ce concept par une analogie. Supposons que vous soyez dans la rue de la poste et que quelqu'un vous demande l'emplacement du bureau de poste. Vous pouvez lui donner le numéro du bâtiment ou sa distance en mètres depuis le début de la rue (adresse absolue) et votre réponse ne dépend pas d'où vous vous trouvez. Vous pouvez aussi lui donner l'information par rapport (adressage relatif) à l'endroit auquel vous vous trouvez : la poste est à plus ou moins x mètres (ou numéros) dans telle direction.

La rupture de séquence peut avoir lieu seulement dans certaines situations (branchement conditionnel) ou au contraire systématiquement (branchement inconditionnel). Pour les sauts conditionnels, la décision est prise en fonction des indicateurs ZNCV du registre d'état du processeur.

Les processeurs performants utilisent une technique plus ou moins poussée de travail à la

chaîne (pipeline) qui leur permet de débuter l'exécution d'une nouvelle instruction à chaque cycle d'horloge. Il en résulte que lorsque instruction courante consulte le contenu du compteur ordinal, ce dernier est en avance d'une ou plusieurs instructions par rapport à l'instruction courante. Le calcul du déplacement dans les branchements relatifs doit en tenir compte.

8.1.2 Les instructions de saut du processeur RISC de référence

Notre processeur RISC fictif possède des instructions de branchement relatif conditionnels, notée b_{cond}[1] **destination**. En langage d'assemblage, l'opérande de l'instruction est soit déplacement (entier signé), soit une étiquette que l'assembleur convertit en un déplacement par rapport à l'instruction b_{cond}.

L'instruction b_{cond} est codée sur un seul mot dont quatre bits indiquent qu'il s'agit d'un branchement relatif conditionnel, quatre autres spécifient la nature de condition testée et les vingt quatre autres encodent le déplacement signé exprimé en nombre d'instructions (à multiplier par quatre avant ajout au compteur ordinal).

Les branchements conditionnels sont généralement utilisés après une comparaison (cf figure 8.3). Les pincipales conditions sur les entier naturels testent Z et C (LOwer ou Carry Clear, Lower or Same, HIgher, Higher or Same ou Carry Set), celles sur les entiers relatifs testent Z, N et V (Less Than, Less or Equal, Greater Than, Greater or Equal). Le test de Z est valable pour les deux types d'entiers (EQual, Not Equal). La condition ALways est toujours vraie : elle correspond à un branchement relatif inconditionnel.

L'instruction de branchement (inconditionnel) non relatif est **jmp reg1 + reg2_ou_#cte8**. Elle est identique à une instruction d'addition qui déposerait son résultat dans le compteur ordinal plutôt que dans un registre de travail ordinaire : l'adresse de la prochaine instruction est la somme du contenu du premier registre et au choix du contenu d'un autre registre ou d'une petite constante entière sur 8 bits. En langage d'assemblage, l'opérande de jmp peut se limiter à un registre : jmp reg sera interprétrée comme jmp reg + #0.

Les variantes $b_{cond}l$ (branch and link) et **jmpl** (jump and link) sont destinées aux appels de procédures. Avant le branchement, elles sauvegardent le compteur ordinal (qui repère alors l'instruction qui suit $b_{cond}l$ ou **jmpl**) dans un général nommé **lr** (link register).

8.2 Etiquettes, goto et programmation structurée

Nous avons vu que le langage d'assemblage permet d'associer des noms symboliques aux adresses : les étiquettes. Via les pointeurs, le langage C offre l'équivalent des étiquettes pour les adresses de variables et de fonctions.

Le langage C permet de plus de définir des étiquettes dans le corps des procédures et l'instruction **goto** permet de réaliser des sauts à ces étiquettes. Il est possible de décrire des branchements conditionels sous la forme **if** (condition) **goto** étiquette, chacun correspondant à une instruction machine.

Dans les langages de programmation de première génération (exemple : FORTRAN IV), les étiquettes (souvent numériques) et les instructions de branchement étaient les seules primitives offertes aux programmeurs pour la prise de décision dans les programmes. Depuis, les techniques

[1] beq, bne, bcc, bge, ... (voir tables 1.2 et 1.12)

dites de programmation structurée ont été généralisées. Elles évitent l'utilisation anarchique de branchements et permettent d'obtenir des programmes plus facile à lire et maintenir.

Les programmes sont aujourd'hui bâtis à partir d'un ensemble de constructeurs algorithmiques intégrés dans la définition des langages, tels que **si alors sinon**, **tant que**, ou les boucles itérées et la programmation à base de if et goto est à proscrire.

La suite de ce chapitre montre comment passer d'un programme C normal à un programme C équivalent en remplaçant toutes les occurences des constructeurs algorithmiques classiques par l'utilisation d'étiquettes et de goto, pour guider sa traduction en langage d'assemblage.

L'instruction goto et les étiquettes ne sont pas évidement pas destinées à la programmation ordinaire d'applications en C, bien que leur utilisation ponctuelle peut parfois simplifier la gestion de la remontée des erreurs[2]. Nous n'utilisons la forme intermédiaire avec if et goto que comme une notation (que le compilateur peut traduire, ce qui permet d'en vérifier la justesse) de décomposition en actions élémentaires pour la traduction en langage d'assemblage.

Le programme C de départ à traduire en langage d'assemblage doit être écrit normalement, autrement dit sans goto, en utilisant les constructeurs algorithmiques classsiques (if, while, for, switch, etc). Réservez les étiquettes et instructions goto à l'étape intermédiaire de traduction d'un programme C normal en langage d'assemblage : **n'utilisez jamais une programmation non structurée à base de if et goto pour programmer une application.**

8.3 Constructeurs algorithmiques C

8.3.1 Instruction vide, instruction composée et accolades

Les accolades permettent de délimiter une séquence d'instructions (séparées par des caractères point-virgule) à considérer comme une unique instruction composée. Il est ainsi possible de considérer que le corps d'une boucle ou d'une autre construction ne contient qu'une instruction : instruction simple ordinaire ou suite d'instructions délimitée par les accolades.

Le caractère point virgule (;) est un marqueur de fin d'instruction. Utilisé seul, il correspond à une instruction vide.

8.3.2 if (condition) instr_alors else instr_sinon

Noter l'absence de mot-clé *then* entre la condition et la branche alors.

Le comportement de la construction dépend de la valeur de la condition placée entre parenthèses. Les flèches illustrent le flot d'exécution, les sauts étant représentés en trait pointillé.
Les flèches sont étiquetées avec un condition vraie (c) ou fausse (/c).

L'expression condition c est évaluée. Une valeur non nulle est interprétée comme VRAI [3], et indique que seule instruction_alors doit être exécutée. Une valeur nulle est interpétée comme FAUX et signifie que seule instruction_sinon doit être exécutée.

[2]pour laquelle le mécanisme d'exception a été indroduit dans les langages orientés objet.
[3]cf 2.2.4 : interprétation booléenne de variables entieres

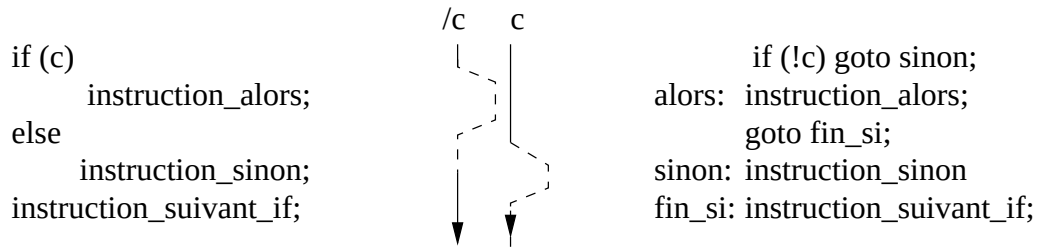


FIG. 8.1 – Transformation d'un si alors sinon

```
/* Passage a la forme goto + etiquettes */
if (x<y)              /* debut_si:  if (x>=y) goto sinon  */
  {                   /*                                  */
  m = y;              /*                m = y;            */
  inferieur ++;       /*                inferieur ++;     */
  }                   /*                goto fin_si;      */
else                  /* sinon:                           */
  m = x;              /*                 m = x;           */
  x++;                /*                 x++;             */
maximum = m;          /* fin_si:    maximum = m;          */
```

Le lecteur notera le branchement au sinon sur la condition (x >= y) opposée à celle du if (x < y) (la condition inverse d'une inégalité stricte inclut aussi le cas d'égalité).

La branche sinon est facultative, auquel cas else et instr_sinon sont omis. Instr_alors et instr_sinon peuvent elles-mêmes être des constructions if, while, for, etc. En cas d'ambiguïté, le else se rattache à l'instruction if la plus proche.

```
if ((d <= a) && (d <= b) && (d <= c))
  mini = d;
else if ((c <= a) && (c <= b) && (c <= d))
  mini = c;
else if ((b <= a) && (b <= c) && (b <= d))
  mini = b;
else
  mini = a;

/* signifie */

if ((d <= a) && (d <= b) && (d <= c))
  mini = d;
else
  {
  if ((c <= a) && (c <= b) && (c <= d))
     mini = c;
  else
     {
     if ((b <= a) && (b <= c) && (b <= d))
        mini = b;
     else
        mini = a;
```

```
    }
  }
```

D'autre part, compte tenu de l'interprétation booléenne des valeurs entières, si var_ent est une variable entière, alors if (var_ent) et if (var_ent != 0) sont synonymes.

Notons enfin une variante de traduction possible basée sur un branchement conditionnel utilisant la condition du if, non inversée, pour sauter à la branche alors. Cette variante peut paraître plus simple à comprendre, mais présente l'inconvénient de générer deux instructions de branchement même en la branche sinon de l'instruction if est absente.

```
if (c)  goto alors;
sinon:  instruction_sinon;
        goto fin_si;
alors:  instruction_alors;
fin_si: instruction_suivant_if;
```

8.3.3 Tant que et répéter jusqu'à (while et do...while)

La condition du tant que est évaluée avant exécution du corps du tant que. Si l'évaluation retourne vrai, le corps du tant que est exécuté. Le processus est répété jusqu'à ce que la condition retourne la valeur faux.

Le lecteur notera l'absence de ';' après la condition et que la condition n'est pas la condition de sortie de boucle, mais celle de continuation.

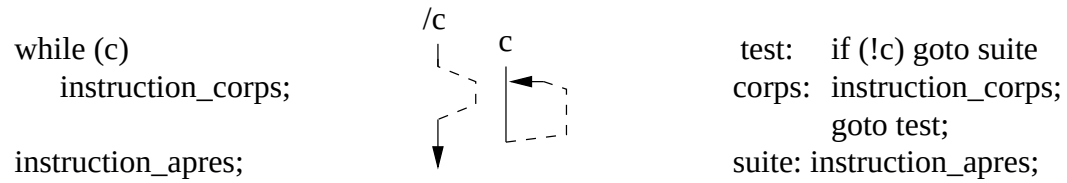


FIG. 8.2 – Transformation d'un tant que

Le schéma de traduction précédent présente l'inconvénient d'exécuter deux instructions goto par tour de boucle. Pour l'éviter, il suffit de placer le test de la condition (non inversée) après le corps du tant que et d'ajouter un branchement inconditionnel à ce test devant le corps du tant que.

```
/* Passage a la forme goto + etiquettes */

while (x != y)          /*          goto test;             */
  {                     /*    corps:                       */
  x = x + 1;            /*          x = x + 1;             */
  y = y / 2;            /*          y = y / 2;             */
  }                     /*    test: if (x!=y) goto corps;  */
z = x;                  /*    z = x;                       */
```

Remarque : while(1) définit une boucle infinie (voir aussi l'instruction break).

Le constructeur do...while correspond à la primitive répéter jusqu'à. Il se comporte comme while, mais le corps est exécuté au moins une fois avant évaluation de la condition. Sa transformation est identique à celle de while, à l'omission du branchement initial au test près.

```
/* Passage a la forme goto + etiquettes */
do
  {                       /*    corps:                          */
  x = x + 1;              /*          x = x + 1;                */
  y = y / 2;              /*          y = y / 2;                */
  }
while (x != y);           /*    test:  if (x!=y) goto corps;    */
z = x;                    /*    z = x;                          */
```

8.3.4 Boucles for itérées et génériques

Considérons l'exemple de boucle suivant.

```
/* Calcul des puissances de 2 */

int tab_puissance[32];  /* un tableau a remplir          */

int i;                  /* variable de boucle principale */
int deux_puissance_i;   /* variable de boucle secondaire */

/* Initialisation (des variables) de la boucle */
i = 0;
deux_puissance_i = 1;

/* Condition de boucle */
while (i < 32)
  {
  /* corps de la boucle */
  tab_puissance[i] = deux_puissance_i;

  /* mise à jour des variables de boucle */
  i ++;
  deux_puissance_i <<= 1;   /* multiplication par 2 par decalage */
  }
```

Une boucle générique manipule une ou plusieurs variables de (contrôle de la) boucle et combine quatre éléments :

1. la séquence d'initialisation des variables de boucle (i = 0 et deux_puissance_i = 1),
2. la condition de continuation[4], testée avant chaque exécution du corps de la boucle (i<32),
3. la mise à jour (souvent une incrémentation) des variables après exécution du corps du boucle (i++ et deux_puissance_i «= 1)
4. le corps de la boucle (affectation à tab_puissance[i]).

Le corps de la boucle for regroupe les instructions exécutées à chaque tour de boucle, test de la condition et mise à jour des variables de boucles exclues. Il est possible de spécifier un corps vide avec une paire d'accolades n'encadrant aucune instruction. A noter : on peut aussi spécifier (souvent involontairement) un corps vide en ajoutant un point-virgule après la parenthèse fermante.

[4]For spécifie une condition de continuation : si elle est vraie le corps est exécuté (répéter tant que la condition reste vraie). NB : en algorithmique, il est courant de spécifier à l'inverse une condition de sortie (répéter jusqu'à ce que la condition devienne vraie).

La syntaxe du for C est la suivante (remarquer l'absence de ';' après la parenthèse fermante) :

```
for (init1, init2, ..., initn ; condition ; maj1, maj2, ..., majn)
  instruction_simple_ou_composee;
```

Réécrivons notre exemple de boucle avec une instruction for :

```
/* le meme exemple avec une boucle for */
for (i=0, deux_puissance_i = 1; i < 32; i++, deux_puissance_i <<= 1)
  tab_puissance[i] = deux_puissance_i;
```

La partie gauche du contenu des parenthèses (init1, init2, ..., initn) est la séquence des instructions d'initialisation des variables, séparées par des virgules. Dans l'exemple ci-dessus, init1 correspond à $i = 0$ et init2 à $deux\_puissance\_i = 1$.

Le premier ';' sépare l'initialisation de la condition de continuation de la boucle, ici $i < 32$.

La partie droite après le deuxième ';' est la séquence des instructions de mise à jour des variables à chaque tour de boucle. Notre exemple en comprend deux : maj1 ($i++$) et maj2 ($deux\_puissance\_i \ll= 1$).

A titre d'exemple, l'itération C classique de 0 à n-1 de parcours d'un tableau de n éléments [5] s'écrit comme suit :

```
/* calcul de la somme des elements d'un tableau */
const TAILLE_TAB = 10;
int tableau[TAILLE_TAB] = { ... }; /* Liste de valeurs d'élements */
...
int cumul_elements = 0;
int i;
...
for (i=0; i<TAILLE_TAB; i++)
     cumul_elements += tableau[i];
...
```

Chacun des trois composants (initialisation, condition, mise à jour) peut être omis. La boucle for (;;) est une boucle infinie [6].

8.3.5 Instructions continue et break

L'instruction continue est utilisée dans les boucles for, while ou do...while. Elle arrête l'exécution de l'itération courante (dans le cas d'une boucle for, les instructions de mise à jour des variables de boucles sont cependant exécutées).

```
/* Calcul de la somme des éléments et remplacement */
/* par la valeur absolue */
...
for (i = 0; i< TAILLE_TAB; i++)
  {
```

[5] Nous verrons que les tableaux C sont indicés à partir de 0.

[6] Voir aussi l'instruction break.

```
  somme += tableau[i];              /* version equivalente        */
  if (tableau[i] >= 0) continue;    /* sauter changement de signe */
  tableau[i] = - tableau[i];        /* si élément déjà >= 0       */
  }
```

Son exécution est suivie de l'évaluation de la condition de boucle et (selon le résultat) de l'exécution de l'itération suivante ou de la sortie de la boucle. L'instruction continue correspond à un branchement incondtionnel au test de la condition de boucle.

```
      i = 0;
test: if (i >= TAILLE_TAB) goto suite;
      somme += tableau[i];
      if (tableau[i] >=0) goto test;
      tableau[i] = - tableau[i];
      goto test;
suite: /* ... */
```

L'instruction break a pour effet de terminer immédiatement l'exécution de l'instruction switch, while, do...while ou for à l'intérieur de laquelle elle est exécutée.

```
                          /* version equivalente        */
while (x < N)             /* while(1)  {                */
  {                       /*  if (!(x < N)) break;      */
  y += x;                 /*  y += x;                   */
  }                       /*  }                         */
```

L'utilisation de l'instruction break revient en général à exécuter une instruction goto vers une étiquette après la fin de la boucle.

```
test_while:   /* neant : condition toujours vraie */
corps_while:  if (x >= N) goto suite;
              y += x;
              goto test_while;
suite:        /* ... */
```

8.3.6 Constructeur selon (switch ... case)

L'instruction switch a pour paramètre une expression (en général une variable), notée entre parenthèses, dont le contenu sera comparé à différentes constantes entières[7].

Le corps du switch est une séquence d'instructions C étiquetées par des gardes. Une garde est un sorte d'étiquette un peu particulière. Une garde ordinaire est de la forme **case constante :** .

Deux gardes distinctes ne doivent pas porter sur la même constante. Une instruction peut être étiquetée par plusieurs gardes.

La valeur de l'expression est comparée aux constantes des différentes gardes dans l'ordre où elles sont écrites. En cas d'égalité avec la constante d'une garde, l'exécution se poursuit à partir de cette garde . En l'absence de garde répondant au critère, l'exécution de l'instruction switch se

[7]Il peut s'agir d'expressions évaluables à la compilation.

termine.

Il est possible de définir une garde particulière, identifiée par **default :**, à placer en dernier, et qui est sélectionnée si aucune autre garde ne correspond à la valeur de la variable.

Chaque séquence d'instructions gardées se termine en principe par une instruction **break**, de telle sorte que les instructions associées aux gardes suivantes ne soient pas exécutées.

A titre d'exemple, décrivons un programme ayant le comportement suivant :
- les nombres pairs à un seul chiffre sont traités individuellement,
- le chiffre 1 est affiché,
- un message est affiché pour tous les nombres impairs à un seul chiffre (1 inclus),
- un traitement commun est appliqué à tous les nombres à plusieurs chiffres.

```
...
switch (v) {
case 0:     printf ("v est égale a 0\n");
            traiter_la_valeur_nulle ();
            break;                          /* fin de branche normale */
case 2:     printf ("v est egale a 2\n");
            traiter_la_valeur_deux ();
            break;
case 4:     printf ("v est egale a 4\n");
            traiter_la_valeur_quatre ();
            break;
case 6:     printf ("v est egale a 6\n");
            traiter_la_valeur_six ();
            break;
case 8:     printf ("v est egale a 8\n");
            traiter_la_valeur_huit ();
            break;
case 1:     printf ("v est egale a 1\n");
            /* pas de break : poursuite en ligne avec afficher impair */
case 3:
case 5:
case 7:
case 9:     printf ("C'est un chiffre impair\n");
            break;
default:    traiter_pas_un_seul_chiffre ();
}
```

L'absence volontaire de break en fin de branche d'un switch passe facilement inaperçue à la lecture du programme, et mérite un commentaire.

Comme les autres constructeurs, switch sera converti en version à goto + étiquettes en vue de sa traduction en langage d'assemblage :

```
test_0: if (v != 0) goto test_2;
            printf ("v est égale a 0\n");
            traiter_la_valeur_nulle ();
            goto fin_switch;
```

```
test_2: if (v != 2) goto test_4;
            printf ("v est égale a 2\n");
            traiter_la_valeur_deux ();
            goto fin_switch;
test_4: if (v != 4) goto test_6;
            printf ("v est égale a 4\n");
            traiter_la_valeur_quatre ();
            goto fin_switch;
test_6: if (v != 6) goto test_8;
            printf ("v est égale a 6\n");
            traiter_la_valeur_six ();
            goto fin_switch;
test_8: if (v != 8) goto test_1;
            printf ("v est égale a 8\n");
            traiter_la_valeur_huit ();
            goto fin_switch;
test_1: if (v != 1) goto test_3;
            printf ("v est egale a 1\n");
            goto trouve_impair;
test_3: if (v == 3) goto trouve_impair;
test_5: if (v == 5) goto trouve_impair;
test_7: if (v == 7) goto trouve_impair;
test_9: if (v != 9) goto default;
trouve_impair:
        printf ("C'est un chiffre impair\n");
        goto    fin_switch;
default:
             traiter_pas_un_seul_chiffre ();
fin_switch:   /* instructions après le switch */
```

Remarque : Lorsque les valeurs des gardes sont nombreuses et forment un sous-ensemble dense d'un intervalle de valeurs entières, il est plus intéressant de recourir à un tableau de branchement indicé par la variable du switch. Cette technique économise l'exécution d'une longue séquence de if (...) goto, et permet de sélectionner la branche du switch en temps constant.

8.4 Quelques pièges liés à la syntaxe C

Les particularités syntaxiques du langage C réservent quelques pièges classiques aux programmeurs habitués à d'autres langages de programmation tels que PASCAL ou ADA.

Le programmeur novice en langage C ou confronté à un comportement anormal de son programme a tout intérêt à relire son code pour vérifier s'il n'a pas commis une des étourderies classiques passées revues ici.

8.4.1 Affectations dans les conditions

Une erreur classique consiste à écrire les affectations i := et les comparaisons = comme dans d'autres langages.

L'utilisation de := dans les affectations est facile à détecter (le compilateur C signalera l'erreur) et à corriger par une simple commande de substitution de texte.

En revanche, l'oubli de doubler le signe = dans les comparaisons est un piège redoutable à cause de l'absence d'erreur de compilation. L'utilisation d'une affectation C, (qui est une aussi une expression retournant la valeur de son membre droit) comme condition est parfaitement légale en C.

L'utilisateur habituel de langage PASCAL ou ADA croira programmer une comparaison entre x et y en écrivant le code C suivant.

```
if (x=y)                  /* Erreur : if (x=y) au lieu de if (x==y) */
  printf("egalite \n");
else
  printf("x et y sont différents\n");
```

Il n'est rien : l'affectation C sera éxécutée et (en tant qu'expression) retournera la valeur de y. Une valeur de y différente de 0 sera considérée comme une condition vraie, une valeur nulle comme une condition fausse. Le fragment de programme précédent est en réalité équivalent à la séquence C suivante, dont le comportement est très différent de ce que souhaitait le programmeur :

```
x = y;
if (y != 0)
  printf ("egalite \n");
else
  printf ("x et y sont différents\n");
```

8.4.2 Corps de boucle vide

Il est très facile d'écrire des boucles dont le corps est vide. La condition d'une boucle while, notée entre parenthèses, est suivie du corps de boucle, sans caractère de séparation entre les deux. Le corps peut se limiter à une simple instruction terminée par le caractère ';'.

Or le caractère ';' utilisé seul est une instruction : l'instruction vide. Ajouter un ';' derrière la condition d'une boucle constitue une autre étourderie classique. Un programmeur souhaitant écrire un calcul du plus grand commun diviseur (PGCD) de x et y et maîtrisant encore mal l'emploi du ';' en C, pourrait écrire ce genre de code :

```
while (x != y);            /* Erreur : ajout d'un ; après la condition */
  if (x >= y)
     x = x - y;
  else
     y = y - x;
```

Il sera sans doute très surpris en découvrant que son programme se comporte comme le code suivant :

```
while (x != y)
  {
  /* corps vide : la boucle est infinie si x != y au depart */
  }
if (x >= y)
```

```
  x = x - y;
else
  y = y - x;
```

Il est recommandé de mettre en évidence l'utilisation volontaire de corps de boucles vides en utilisant les accolades ou via un commentaire approprié.

```
/* Attendre la disponibilité du périphérique */
while ((*REG_ETAT_PERIPHERIQUE) & PERIPH_PRET);  /* corps vide */

/* ou encore */
while ((*REG_ETAT_PERIPHERIQUE) & PERIPH_PRET)
  {
  }
```

8.4.3 Enchaînement d'alternatives d'un selon (switch)

En l'absence de *break*, l'exécution d'une branche d'un switch C se poursuit en séquence avec le code de l'alternative suivante.

```
...
switch {meteo}
  {
  case NEIGE :  enfiler_anorak ();    /* poursuite dans pluie */
  case PLUIE :  ouvrir_parapluie ();
                /* oubli de break ici */
  case SOLEIL:  mettre_maillot_de_bain ();
                mettre_creme_solaire ();
                break;
  default:      ;
  }
```

Le comportement voulu dans cet exemple est assez simple : sous la neige, le promeneur partira abrité de son parapluie et équipé de son anorak. Si la température est assez élevée pour qu'il pleuve au lieu de neiger, le parapluie suffira et l'anorak restera dans sa penderie.

L'oubli de l'instruction break donnera un résultat étonnant : un observateur sera surpris de voir un promeneur enfiler un maillot de bain et s'enduire d'huile solaire après avoir passé un anorak et ouvert son parapluie pour affronter la neige ....

8.5 Traduction de if...goto en langage d'assemblage

Chaque instruction C **if (condition) goto etiquette** se traduit en langage d'assemblage par une séquence d'instructions mettant à jour les indicateurs Z, N, C et V en fonction de la condition à tester, et (au moins) une instruction b_{cond} **etiquette**.

8.5.1 Traduction de if...goto avec une comparaison

La majorité des conditions sont des comparaisons d'entiers.

```
register unsigned int r1, r2, r3;
unsigned int maxi;
...
r3 = r1;
if (r1 < r2) goto trouve;
r3 = r2;
trouve:
maxi = r3;
```

Pour l'expression des conditions en fonction des indicateurs, se reporter aux chapitres traitant de l'arithmétique entière, et aux tables 1.2 et 1.12.

Conditions des instructions de branchement conditionnel				
Type	Entiers signés		Naturels 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

FIG. 8.3 – Utilisation des branchements conditionnels après une comparaison

Les if (condition) goto dans lesquelles la condition porte sur la valeur relative de deux entiers seront traduits en langage d'assemblage par une instruction **cmp** de comparaison des deux entiers, suivie d'une instruction de branchement conditionnel appropriée.

```
          mov   r3, r1
          cmp   r1, r2
          blo   trouve
          mov   r3, r2
trouve:   ldr   r0, #maxi
          str   r3, [r0]
```

Les indicateurs à tester dépendent de la condition à tester et de la nature (signée ou non) des entiers à comparer. La table 8.3 résume les règles de choix des instructions de branchement et le codage en binaire du champ condition.

8.5.2 Autres conditions testables par b_{cond}

Il est aussi possible de tester quelques conditions particulières résumées dans la table 8.5.2.

8.5.3 Choix de condition inadaptée à la nature des entiers

L'existence pour une même inégalité de deux variantes de branchement conditionnel est mal comprise par de nombreux programmeurs en langage d'assemblage.

Bxx	Codage	Indicateurs	Sigle	Commentaire
BMI	0100	N	Minus	Résultat apparent < 0
BPL	0101	$\overline{N}$	Plus	Résultat apparent ≥ 0
BVS	0110	V	V Set	Débordement signé
BVC	0111	$\overline{V}$	V Clear	Pas de débordement signé
BAL	1110	1	Always	Branchement inconditionnel

FIG. 8.4 – Branchements testant des conditions particulières

Le mauvais réflexe classique est de prendre parmi les instructions de branchement conditionnel, et ce quelque soit la nature des entiers à comparer, celle dont le sigle évoque le mieux la comparaison écrite en C, soit BEQ, BNE, BLT, BLE, BGT et BGE.

Que se passe si l'on utilise par exemple BGE au lieu de BHS pour une condition $x \geq y$ portant des entiers x et y de type unsigned ?

Lorsque x et y appartiennent à la même moitié de l'intervalle des entiers naturels repésentables tout se passe bien. En effet, BGE interpréte x et y comme deux entiers de même signe et prend la même décision que BHS.

Dans le cas contraire, les deux entiers différent au moins par leur bit de pids fort (supposons que $x_{n-1} = 0$ et $y_{n-1} = 1$). BHS détecte un entier y supérieur à l'entier x, alors que BGE interpréte y comme un entier signé négatif, donc inférieur à l'entier signé x positif ou nul. Les deux instructions ont dans ce cas des comportements opposés.

Si la comparaison porte sur des adresses dans le cas d'un parcours de tableau via un pointeur, les problèmes se poseront lorsque le tableau est implanté à cheval sur les deux moitiés de l'espace mémoire adressable (début avant 0x7FFFFFFF et fin après 0x80000000).

8.5.4 Gestion de conditions quelconques

Bien que les comparaisons ordinaires représentent la grande majorité des cas, l'expression de la condition des if, while, do et for peut être plus complexe.

Comparaison implicite à 0

Une condition se limitant à une simple variable est une comparaison implicite avec 0 : if (x) ... est par définition équivalent à if (x != 0) ....

```
register int r1;
...                        /*      cmp  r1, #0      */
if (x) goto ailleurs;      /*      bne  ailleurs    */
```

Calculs et affectations dans les conditions

La condition peut être une expression à calculer, éventuellement affectée à une variable au passage, et dont la valeur est implicitement à comparer à 0.

Le calcul peut être déplacé avant l'évaluation de la condition, en introduisant au besoin un temporaire.

```
register int r1, r2;
...
if ((r1 + r2) & 1) goto somme_impaire;
...
/* transformation */
register int reg_temp;           /* r3 */
...
reg_temp = (r1 + r2) & 1;
if (reg_temp != 0) goto somme_impaire;
...
```

Dans ce cas de figure, il est possible d'économiser une instruction de comparaison si la dernière instruction de calcul de l'expression met à jour les indicateurs.

```
...
     add   r3, r1, r2
     andS  r3, #1          ; met à jour ZNCV
     bne   somme_paire
...
```

8.5.5 Conditions composées (|| et &&)

Les constructons algorithmiques à conditions composites obtenues par assemblage de conditions simples seront transformées en un assemblage équivalent de constructeurs et if...goto utilisant chacun une condition simple.

```
if ((a<b) && (b<c)) milieu = c;

/* version equivalente */
if (a<b)
  if (b<c) milieu = c;

/* version equivalente transformee en if() goto */
if (a >= b) goto fin_si;
   if (b >= c) goto fin_si;
      milieu = c;
fini_si:

if ((a>9) || (a<0)) printf ("Pas un chiffre\n");

/* transformation */
if (a>9) goto print;
if (a>=0) goto fin_si;
print: printf ("Pas un chiffre\n");
fin_si:
```

Chapitre 9

Tableaux et arithmétique sur les pointeurs

La notion de tableau correspond à une collection d'objets de même type, identifiés par des numéros (ou indices), et stockés à des emplacements mémoire consécutifs[1].

En C, il n'existe pas à proprement parler de type tableau : les accès aux tableaux sont en fait réalisés via les primitives de manipulation des pointeurs. Pour permettre la gestion de tableaux, le langage C offre les facilités suivantes :

1. la réservation d'emplacements contigus en mémoire pour n éléments de même type, avec ou sans valeurs initiales,
2. une convention de numérotation : les éléments d'un tableau C de n éléments sont indicés à partir de 0 et jusqu'à $n-1$,
3. l'opérateur d'indiçage et l'arithmétique associée sur les adresses et les pointeurs ([ ]),
4. une convention d'ordre de stockage des tableaux de tableaux, pour les tableaux à plusieurs dimensions.

9.1 Déclaration de tableau à une dimension

9.1.1 Syntaxe de la déclaration

Une déclaration de tableau avec réservation de place comprend dans l'ordre :
- le type des éléments du tableau,
- le nom du tableau déclaré,
- sa taille encadrée par des crochets et exprimée en nombre d'éléments, qu'il est conseillé de définir par une constante symbolique.
- une spécifiation optionnelle de valeur initiale, à savoir le signe égal suivi d'un contenu de tableau,
- le marqueur de fin (caractère ';').

Le rôle de la déclaration est triple :
- associer une information de type (type d'éléments et dimensions) au tableau en vue de vérifier la concordance de type dans les instructions utilisant le tableau,
- réserver de la mémoire pour contenir le tableau et l'initialiser le cas échéant,
- associer une étiquette du nom du tableau à l'adresse du contenant : l'adresse du premier élément du tableau est celle du premier des octets réservés au stockage du tableau : la déclaration d'un tableau t définit implicitement t comme synonyme de la constante adresse **&(t[0])** [2].

[1] ...en respectant les contraintes d'alignement...

[2] C'est la raison pour laquelle l'opérateur & ne s'applique par à un tableau : &t signifierait &&(t[0]).

Pour la traduction en langage d'assemblage, une déclaration de tableau de n éléments est traitée comme n réservations de place pour un élément. En l'absence de valeur initiale explicite, le tableau sera stocké dans la section bss.

Attention : la taille du tableau n'est pas stockée en mémoire avec ses éléments et aucune vérification dynamique de validité de l'indice n'est effectuée à l'exécution. L'exécution d'un accès à t[i] avec i égal à -1 n'est pas légal, mais réalisera vraisemblablement un accès à la variable déclarée avant le tableau t, sans générer d'erreur durant l'exécution.

9.1.2 Syntaxe de l'initialisation

La syntaxe C décrivant un contenu de tableau est un ensemble d'autant de constantes que d'éléments, séparées par des virgules, et encadré par des accolades.

La partie initialisation d'une déclaration de tableaux peut spécifier moins de valeurs que d'éléments déclarés (dimension) du tableau, les derniers éléments du tableau seront implicitement initialisés à 0.

Lorqu'un tableau est déclaré avec initialisation, la taille de sa dimension peut être omise : elle est alors définie implicitement à partir du nombre d'éléments dans la partie initialisation.

9.1.3 Exemple sans initialisation

```
#define TAILLE_NOM 35
char nom[TAILLE_NOM];
int  puis2 [4];
```

Dans la traduction en langage d'assemblage, il est possible de définir une étiquette pour chaque élément du tableau.

```
            .bss
nom:                            /* adr tableau = adr 1er element */
adr_nom_0:      .skip  1            /* stockage de nom[0] */
adr_nom_1:      .skip  1            /* stockage de nom[1] */
adr_nom_2:      .skip  1            /* stockage de nom[2] */
...
adr_nom_34:     .skip  1            /* stockage de nom[34] */
                .align 4
puis2:
adr_puis2_0:    .skip  4            /* stockage de puis2[0] */
adr_puis2_1:    .skip  4            /* stockage de puis2[1] */
adr_puis2_2:    .skip  4            /* stockage de puis2[2] */
adr_puis2_3:    .skip  4            /* stockage de puis2[3] */
```

Toutefois, l'adresse d'un élément est calculée à partir de son indice et de l'adresse (du premier élément) du tableau. Il suffit donc de ne définir qu'une étiquette, du nom du tableau, sur l'emplacement du premier élément.

```
            .bss
nom:        .skip  35
```

```
            .align 4
puis2:      .skip  16
```

9.1.4 Exemple avec initialisation

```
#define TAILLE_NOM 35
#define TAILLE_PUIS2 4
char nom[TAILLE_NOM]={'c','o','n','t','e',n','u'};
int  puis2 [TAILLE_PUIS2] = {1,2,4,8};
```

La déclaration d'un tableau initialisé de n éléments est traitée comme n déclarations d'éléments initialisés séparéments. Cependant les éléments doivent être contigus : si un des éléments est initialisé explicitement, tous les éléments du tableau seront stockés dans la section data.

```
            .bss
nom:        .byte  'c'    @  7 elements avec initialisation
            .byte  'o'
            .byte  'n'
            .byte  't'
            .byte  'e'
            .byte  'n'
            .byte  'u'
            .skip   28     @ 28 elements sans initialisation
            .align 4
puis2:      .word   1
            .word   2
            .word   4
            .word   8
```

Notons que nous aurions pu remplacer les sept directives .byte par la directive .ascii "contenu".

9.1.5 Un autre exemple

```
short int tab[5] = {3,6,10,15,-5};
short int x = 4;
short int y;

          .data
adr_tab_0:              @ cette etiquette est inutile mais rappelle
                        @ que tab est equivalent a &(tab[0])
tab:      .short  3
          .short  6
          .short 10
          .short 15
          .short -5

x:        .short  4

          .bss
y:        .skip   2
```

9.1.6 Tableaux de chaînes de caractères

Rappelons que les chaînes de caractères sont représentées comme des tableaux, la fin de chaîne étant signalée par le code ASCII 0.

Pour stocker une chaîne de n caractères, on doit déclarer un tableau de n+1 éléments de type char (ou unsigned char). La taille retournée par la fonction strlen n'inclut pas le 0 de fin de chaîne (strlen("ab") retourne 2).

```
char reponse[4]="oui"; /* ou char reponse[4] = {'o','u','i',0}; */

                .data
reponse:        .asciz  "abc"
```

9.2 Indiçage de tableau et arithmétique sur les adresses

9.2.1 Adresse du $i^{ème}$ élément d'un tableau

La différence entre deux adresses d'éléments consécutifs d'un tableau de type type_t est sizeof(type_t). Les éléments étant indicés à partir de 0, l'adresse du i^{ime} élément d'un tableau est égale à l'adresse du (premier élément du) tableau plus i fois la taille d'un élément.

$$\&(t[i])\ vaut\ t + i * sizeof(type\_t)$$

9.2.2 Arithmétique sur les adresses et indiçage

L'opérateur d'indiçage de tableau en C est [ ] : t[i] est la notation du i^{ime} élément de t. Rappelons que l'adresse du premier élément (d'indice 0) est celle du tableau : t est synonyme de &(t[0]).

L'opérateur C d'addition d'un entier i à un pointeur p (ou une constante adresse) multiplie implicitement l'entier par la taille du type d'objet reperé par p avant d'effectuer l'addition. Il permet de se déplacer de i éléments dans un tableau : si le contenu de p est l'adresse d'un élément de tableau t[k], alors l'expression $p + i$ est l'adresse de l'élément de tableau t[k+i]. Pour ce faire, l'entier ajouté à un pointeur ou une constante adresse est implicitement multiplié par la taille du type d'élément pointé.

Considérons l'affectation $\mathbf{p = p + 3}$. Elle incrémente le contenu de p de 3 si p est de type (void *), (char *) ou (unsigned char *), de 6 si p est de type (short int *) ou (unsigned short int *) et de 12 si p est un pointeur d'entier sur 32 bits.

Par définition de [ ] et *, t[i] est strictement équivalent à *(t+i) que l'on peut aussi écrire *(&(t[0]) + i).

9.3 Traduction des accès aux tableaux

La méthode de traduction en langage d'assemblage des manipulations de tableaux en passe par deux réécritures du programme C d'origine dans des formes de C intermédiaire.

La première réécriture remplace les opérateurs d'indiçage [] par des opérateurs *. La deuxième fait apparaître les variables intermédiaires pour les adresses et les contenus et met en évidence les multiplications par la taille des éléments.

9.3.1 Exemple à traduire

```
short int tab[5] = {3,6,10,15,-5};
short int x = 4;
short int y;

register short int reg_ptr;
...
y = tab[3];
tab[4] = 5;
*tab = -2;                /* tab[0] = -2 */
reg_ptr = tab;            /* reg_ptr repere tab[0] */
*reg_ptr = 3              /* tab[0] = 3 */
reg_ptr += 2;             /* reg_ptr repere tab[2] */
tab[x] = (*reg_ptr) + 4; /* tab[x] = tab[2] + 4 */
y = *(reg_ptr+2)          /* y = tab[4] */
...
}
```

9.3.2 Elimination des opérateurs [ ]

La première version intermédiaire est obtenue remplaçant systématiquement toute expression t[i] par *(t+i) et toute variable v stockée en mémoire par *&v.

```
*&y = * (tab + 3);
*(tab + 4) = 5;
reg_ptr = tab;
*reg_ptr = 3;
reg_ptr += 2;
* (tab + *&x) = *reg_ptr+4;
*&y = * (reg_ptr+2);
```

9.3.3 Forme intermédiaire pour la traduction

Cette deuxième version intermédiaire destinée à la traduction met en évidence l'utilisation des temporaires, comme pour la traduction des affectations et expressions utilisant des variables ordinaires. Elle met aussi en évidence les multiplications implicites par la taille d'un élément de tableau.

Dans cette forme intermédiaire, nous utilisons un registre pointeur de type (void *), dont le type est converti en (short *) avant application de l'opérateur *. Ainsi, lorsqu'il est ajouté à reg_adr, le contenu de reg_ajout n'est pas multiplié par sizeof(short int).

La conversion du contenu de reg_ptr en (void *) indique au compilateur de ne pas multiplier implicitement l'ajout par sizeof(short).

```
register void *reg_adr;
register int reg_val1, reg_val2;
register int reg_ajout;


reg_adr = tab;                          @ *&y = * (tab+3)
reg_ajout = 3 * 2;                      @ 3 *sizeof (short int)
```

```
reg_val1 = * (short *) (reg_adr + reg_ajout);
reg_adr = &y;
*(short *) reg_adr = reg_val1;

reg_val1 = 5;                                @ *(tab+4) = 5
reg_adr = tab;
reg_ajout = 4 * 2;                           @ 4* sizeof (short int)
* (short *) (reg_adr + reg_ajout) = reg_val1;

reg_ptr = tab;

reg_val1 = 3                                 @ *reg_ptr = 3
*reg_ptr = reg_val1;

reg_ptr = (short *) ((void *) reg_ptr + 2 * 2); @ + 2 * sizeof(short )

                                             @ *(tab + *&x) = *reg_ptr+4
reg_ajout = 4 * 2;                           @ 4 *sizeof (short)
reg_val1 = * (short *)((void *)reg_ptr + reg_atout);
reg_adr = &x;
reg_val2 = *reg_adr;
reg_adr = tab;
reg_ajout = reg_val2 * 2;                    @ 2 *sizeof (short)
* (short*) (reg_adr + reg_ajout) =  reg_val1;

reg_ajout = 2 * sizeof (short int);          @ *&y = *(reg_ptr+2)
reg_val1 = * (short int *)(reg_ptr + reg_ajout);
reg_adr = &y;
*reg_adr = reg_val1;
```

9.3.4 Traduction en langage d'assemblage

La traduction en langage d'assemblage suppose une attribution arbitraire des registres généraux du processeur aux variables reg_adr, reg_ajout, reg_val1, reg_val2 et reg_ptr.

Les opérations entre registres sont traduites en instructions mov et add. Les affectations du type ***regad = regval** et **regval = *regad** correspondent respectivement aux instructions store et load.

Les multiplications par 2^x correspondent à des instructions de décalage de x bits à gauche (lsl reg, reg, #x). Les constantes adresses sur 32 bits sont chargées par mov32, les constantes codables sur 8 bits par mov.

```
@ r0 : reg_ajout
@ r1, r2 : reg_val1 , reg_val2
@ r3 : reg_adr
@ r6 : reg_ptr

     .text
     mov32 r3, #tab       @ reg_adr = tab
     mov   r0, #6         @ reg_ajout = 3 *sizeof(short int) : 3 x 2
```

```
ldrsh r1, [r3, r0]   @ reg_val1 = * (short *) ((void *) reg_adr + reg_ajout)
mov32 r3, #adr_y     @ reg_adr = &y
strh  r1, [r3]       @ * reg_adr = reg_val1

mov   r1, #5         @ reg_val1 = 5
mov32 r3, #tab       @ reg_adr = tab
mov   r0, #8         @ reg_ajout = 4 *sizeof(short int) : 4 x 2
strh  r1, [r3, r0]   @ * (short *) ((void *) reg_adr + reg_ajout) = reg_val1

mov32 r6, #tab       @ reg_ptr = tab

mov   r1, #3         @ reg_val1 = 3
strh  r1, [r6]       @ *reg_ptr = regval1

add   r6, r6, #4     @ reg_ptr += 2

mov   r0, #8         @ reg_ajout = 4 * sizeof(short int) : 4 x 2
ldrsh r1, [r6, r0]   @ reg_val1 = * (short *) ((void *) reg_ptr + reg_ajout)

mov32 r3, #adr_x     @ reg_adr = &x
ldrsh r2, [r3]       @ reg_val2 = * reg_adr
mov32 r3, #tab       @ reg_adr = tab
lsl   r0, r2, #1     @ reg_ajout = reg_val2 * 2  (decale G 1 bit)
strh  r1, [r3, r0]   @  * (short *) ((void *) reg_adr + reg_ajout) = reg_val1

mov   r0, #4         @ reg_ajout = 2 *sizeof (short int)
ldrsh r1, [r6,r0]    @ reg_val1 = * (short *) ((void *) reg_ptr + reg_ajout)
mov32 r3, #adr_y     @ reg_adr = &y
strh  r1, [r3]       @ * reg_adr = reg_val1
```

Il est à remarquer que toutes les instructions store correspondent à un opérateur * situé tout à gauche des affectations dans la première forme C intermédiaire du programme.

9.4 Boucles de parcours d'un tableau à une dimension

9.4.1 Boucle de parcours avec indice

La manière habituelle de parcourir les éléments d'un tableau de N éléments est d'utiliser une variable de boucle parcourant l'intervalle des indices [0, N-1] (autrement dit $[0, N[$: intervalle de 0 inclus à N exclus).

A titre d'exemple, l'extrait de code suivant calcule la somme des puissances de deux contenues dans le tableau puis2 (de taille paire).

```
short int somme_puis2 = 0;
int indice;
...
/* Rappel : indice ++ pourrait s'ecrire indice = indice + 1 */
for (indice = 0; indice < TAILLE_PUIS2; indice++)
  {
  somme += puis2 [i];
  }
```

9.4.2 Boucle de parcours avec pointeur

Une autre manière simple de parcourir un tableau est d'utiliser un pointeur parcourant les adresses des éléments successifs (ici de **&puis2[0]** jusqu'à **&puis2[TAILLE_PUIS2]** exclue). Rappelons que ces adresses peut aussi s'écrire **puis2** et **puis2 + TAILLE_PUIS2**.

```
short int somme_puis2 = 0;
short int *ptr_short;
...

/* Rappel : ptr_short ++ pourrait s'ecrire ptr_short = ptr_short + 1 */
/*          ptr_short ++ signifie se decaler d'un element dans le    */
/*                                tableau repere par ptr_short       */

for (ptr_short = puis2; ptr_short < puis2 + TAILLE_PUIS2; ptr_short++)
  {
   somme += *ptr_element;
  }
```

9.4.3 Conversion de boucle : indice vers pointeur

Il est possible de transformer progressivement un parcours par indice en parcours par pointeur. Considérons à titre d'exemple, la boucle suivante qui ajoute à l'élément d'indice i le contenu de l'élément d'indice TAILLE_PUIS2 - i.

```
register int indice;
...
for (indice = 0;                          /* initialisation */
     indice < TAILLE_PUIS2/2;             /* condition      */
     indice ++)                           /* mise a jour    */
  {
  puis2 [i] = puis2 [i] + puis2 [TAILLE_PUIS2 - i];
  }
```

Remarquons que le calcul des adresses dans l'affectation implique d'effectuer une soustraction et deux multiplications (indice fois taille d'un élément de tableau) par tour de boucle[3].

Chaque tour de boucle met en jeu six calculs :

1. une comparaison (condition sur l'indice)
2. une incrémentation de l'indice
3. deux multiplications et une soustraction pour le calcul des adresses
4. une addition des contenus des éléments de tableau

La première étape consiste à déclarer deux pointeurs qui repèrent en permanence puis2[i] et puis2[TAILLE_PUIS2 -i]. Ils sont initialisés en début de boucle d'après la valeur initiale (0) de indice. A chaque tour de boucle, ils sont mis à jour en fonction de l'évolution de la variable indice.

[3]Plus deux additions de l'adresse de début de tableau réalisables dans une instruction load et dont la présence n'affecte donc pas le temps d'exécution.

```
int indice;
register short int *ptdebut, *ptfin;
...
for (indice = 0, ptdebut = puis2, ptfin = puis2 + TAILLE_PUIS2;
     indice < TAILLE_PUIS2/2;
     indice ++, ptdebut++, ptfin --)
  {
  /* Invariant : ptdebut = puis2 + indice                    */
  /*      ==>    ptdebut repere puis2[i]                     */
  /* Invariant : ptfin   = puis2 + TAILLE_PUIS2 - indice     */
  /*      ==>    pftin repere puis2[TAILLE_PUIS2 - indice]   */
  *ptdebut = *ptdebut + *ptfin;
  }
```

L'invariant nous permet de remplacer indice par ptdebut - puis2 dans la condition qui devient **ptdebut - puis2** $<$ **TAILLE_PUIS2 /2**, puis **ptdebut** $<$ **puis2** $+$ **TAILLE_PUIS2 /2**. Après modification de la condition, la variable indice n'est plus utilisée ni dans le corps de boucle, ni dans la condition de continuation et peut être supprimée.

```
register short int *ptdebut, *ptfin;
...
for (ptdebut = puis2, ptfin = puis2 + TAILLE_PUIS2;
     ptdebut < puis2 + TAILLE_PUIS2/2;
     ptdebut++, ptfin --)
  {
  *ptdebut = *ptdebut + *ptfin;
  }
```

Cette nouvelle version n'effectue que quatre calculs par tour de boucle :

1. une comparaison (condition sur le pointeur ptdebut)
2. une incrémentation (ptdebut++)
3. une soustraction (ptfin–)
4. une addition des contenus des éléments de tableau

On trouve aujourd'hui des compilateurs optimisant capables de réaliser eux-même ce genre de transformation et aboutir à une boucle différente, respectant la sémantique du programme d'origine, mais permettant de générer un code machine plus efficace.

9.5 Contraintes d'alignement et types d'éléments particuliers

Tous les éléments d'un tableau doivent respecter les contraintes d'alignement sur un multiple de leur taille. La réservation de place pour le tableau sera donc éventuellement précédée d'une directive d'alignement.

Les tableaux de pointeurs sont gérés de la même manière que les tableaux d'entiers longs. Rappelons que la taille d'un pointeur est celle d'une adresse, à ne pas confondre avec la taille du type d'objet pointé : quelque soient T et T' deux types C, la propriété suivante est vérifiée :

sizeof (T *) = sizeof (T') = sizeof (void *). Ainsi, pour une machine 32 bits, tous les pointeurs ont une taille de 4 octets.

L'opérateur **sizeof** tient compte des contraintes d'alignement propres aux structures[4]. Ces contraintes s'appliquent évidement entre deux champs appartenant à la même structure, mais elles s'appliquent aussi entre deux éléments consécutifs d'un tableau de structures. Il peut être nécessaire de laisser des octets inutilisés entre le dernier champ d'une structure d'indice i et le premier champ de la structure de rang $i+1$ pour respecter les contraintes d'alignement de ce dernier.

L'opérateur sizeof, utilisé par l'arithmétique sur les pointeurs et les calculs d'adresse des éléments de tableau, doit en tenir compte comme l'illustre l'exemple suivant. Bien que les champs d'une structure s soient déclarés par ordre décroissant de taille et ne nécessitent donc pas d'octet d'alignement entre eux, sizeof(struct s) retourne 8, soit la somme des tailles des champs plus trois pour les octets d'alignement nécessaires entre deux structures de type s consécutives.

```
typedef struct s {   /* sizeof (struct s) = 8 */
  long l;            /* un long de taille 4   */
  char c;            /* un char de taille 1   */
} type_s

type_s tab [2] = {{4,'a'},{5,'b'}};

/* Traduction en langage d'assemblage */

           SIZEOF_STRUCT_S=8
           .data
tab:       .word   4     /* tab[0] */
           .byte  'a'
           .align  4     /* 3 octets d'alignement entre les 2 */
           .byte  'b'    /* tab[1] */
           .align  4
fin_tab:
```

9.6 Tableaux à deux dimensions

9.6.1 Déclaration

Un tableau à une dimension permet de représenter un vecteur.

On peut considérer une matrice (m,n) comme un vecteur de m vecteurs comprenant n éléments chacun. Le langage C permet de déclarer directement des tableaux à 2 dimensions correspondant à de telles matrices sans nommer le type tableau à n éléments.

Une déclaration **type_elem mat [M][N]** définit un tableau mat de taille M dont chacun des éléments est lui-même un tableau de N éléments de type type_elem.

```
#define N 3
#define M 4
```

[4] Ce cas incluant celui des unions

```
/* Deux manières équivalentes de déclarer des tableaux à deux dimensions */

typedef long vect [N];   /* vect : le type tableau de N entiers */

long matrice1 [M][N];    /* matrice (m,n) declaree directement */
vect matrice2 [M];       /* matrice (m,n) en tableau de tableaux */
```

Le procédé est généralisable à plus de deux dimensions : la déclaration **int t [P][M][N]** définit un tableau t de P tableaux à deux dimensions de tailles M et N.

En langage machine, ces déclarations génère les mêmes directives de réservation que la déclaration d'un tableau à une dimension du même nombre d'éléments.

```
            N=3
            M=4

            .bss
matrice1:   .skip  M*N
matrice2:   .skip  M*N
```

9.6.2 Ordre de rangement et calcul d'adresse d'un élément

En mémoire, on trouve d'abord les N éléments du premier des M tableaux, puis les N éléments du deuxième tableau de N éléments, et ainsi de suite.

On trouve donc au début de la mémoire allouée au tableau l'élément d'indices**[0][0]**, suivi de l'élément d'indices **[0][1]** et suivants jusqu'à l'élément d'indices **[0][N-1]**, puis l'élément d'indices **[1][0]** suivi de l'élément d'indices **[0][1]** et ainsi de suite jusqu'à **[M-1][N-1]**. Autrement dit, lorsque l'on parcourt les éléments du tableau dans l'ordre de leur rangement en mémoire, c'est l'indice de la dernière dimension (indice le plus à droite) qui varie le plus vite.

Soit $type\_t[T_n]...[T_2][T_1]$ un tableau à n dimensions, l'adresse de l'élément $t[i_n][i_{n-1}]\dots[i_1][i_0]$ est : $t + sizeof(type\_t) * ((\dots(i_n * T_{n-1} + i_{n-1}) * T_{n-2} + i_{n-2})\dots) * T_1 + i_1) * T_0 + i_0)$.

Comme le montre l'expression ci-dessus, pour calculer l'adresse d'un élément de tableau à n dimensions, il faut connaître la taille des n-1 dernières dimensions.

$$\left(\begin{array}{c|ccc} & \multicolumn{3}{c}{\overrightarrow{\quad j \quad}} \\ i\downarrow & T_{00} & T_{01} & T_{02} \\ & T_{10} & T_{11} & T_{12} \\ & T_{20} & T_{21} & T_{22} \\ & T_{30} & T_{31} & T_{32} \end{array}\right) \Longrightarrow \left(\begin{array}{ll} & \overbrace{\quad\quad\quad}^{j\rightarrow} \\ T[i=0]: & \underbrace{T_{00}\quad T_{01}\quad T_{02}} \\ T[i=1]: & \underbrace{T_{10}\quad T_{11}\quad T_{12}} \\ T[i=2]: & \underbrace{T_{20}\quad T_{21}\quad T_{22}} \\ T[i=3]: & \underbrace{T_{30}\quad T_{31}\quad T_{32}} \end{array}\right) \Longrightarrow \left\{\begin{array}{l} \overbrace{\underbrace{T[0][0]\quad T[0][1]\quad T[0][2]}} \\ \overbrace{\underbrace{T[1][0]\quad T[1][1]\quad T[1][2]}} \\ \overbrace{\underbrace{T[2][0]\quad T[2][1]\quad T[2][2]}} \\ \overbrace{\underbrace{T[3][0]\quad T[3][1]\quad T[3][2]}} \end{array}\right\}$$

FIG. 9.1 – Décomposition d'un tableau 4,3 en tableau de 4 tableaux

9.6.3 Initialisation

Pour l'initialisation, un tableau à deux dimensions est considéré comme un tableau de tableaux. La valeur initiale du tableau est une suite d'éléments entre accolades, chacun de ces éléments étant lui-même une suite de valeurs initiales entre accolades, séparées par des virgules.

Voici à titre d'exemple l'initialisation de la matrice de la figure 9.1.

```
#define valT00 1000
#define valT01 1001
#define valT02 1002
#define valT10 1010
#define valT11 1011
...
#define valT32 1032

long T [4][3] = {  {valT00, valT01, valT02},   /* T[0] */
                   {valT10, valT11, valT12},   /* T[1] */
                   {valT20, valT21, valT22},   /* T[2] */
                   {valT30, valT31, valT32}    /* T[3] */
                };
```

En langage d'assemblage, la réservation avec initialisation est traitée comme celle d'un tableau à une dimension.

```
        valT00=1000
        valT01=1001
        valT02=1002
        valT10=1010
        valT11=1011
        ...
        valT32 1032

        .data
T:      .word valT00     @ debut du tableau T[0]
        .word valT01
        .word valT02
        .word valT10     @ debut du tableau T[1]
        .word valT11
        ...
        .word valT32     @   fin du tableau T[3]
```

9.6.4 Boucle de parcours par pointeur

Considérons le problème du calcul de la somme des éléments d'un tableau à 2 dimensions. La méthode classique utilise deux boucles emboîtées parcourant chacune les indices d'une dimension.

```
for (ligne = 0; ligne < 4; ligne++)
  for (colonne = 0; colonne < 3; colonne++)
     somme += t[ligne][colonne];
```

Mais on peut mettre à profit le fait que les éléments du tableau sont parcourus dans l'ordre de l'ordre stockage en mémoire et n'utiliser qu'une seule boucle avec un pointeur :

```
for (ptr = t; ptr < &(t[4][3]);ptr++)
  somme += *ptr;
```

9.6.5 Passage de tableau à n dimensions en paramètre

Une procédure recevant un tableau passé en paramètre ne peut l'indicer correctement sans information sur la taille des $n-1$ dernières dimensions du tableau. Lorsque celles-ci sont constantes, elles peuvent être indiquées à la déclaration des arguments de la procédure.

Une déclaration d'un paramètre t de type int t [ ][4][3] dans un prototype de fonction n'est pas une déclaration réservant de la place pour stocker un tableau. Elle indique simplement la géométrie du tableau dont l'adresse est passée à la procédure.

Chapitre 10

Procédures sans récursion

Ce chapitre présente la gestion simplifiée des appels de procédures, qui ne serait applicable qu'aux appels sans récursion. La méthode de gestion générale appliquée à tous les cas sera présentée dans le chapitre suivant.

10.1 Notion de procédure

10.1.1 Principe

Soit une même suite d'opérations à exécuter dans différentes parties d'un programme. Pour réduire la taille des programmes et éviter les problèmes de mise à jour de copies multiples en cas de modification, il est préférable de n'inclure qu'un seul exemplaire de la suite d'instructions correspondante, qui constituera le corps d'une procédure. Le programmeur peut passer des arguments à la procédure pour en paramétrer le fonctionnement.

Chaque utilisation de la suite d'opérations constitue un point d'appel de la procédure. En chaque point d'appel est inséré une instruction de branchement aller vers le début (ou prologue) de la procédure. L'épilogue de la procédure contient une instruction de branchement retour vers l'instruction qui suit le point de branchement.

10.1.2 Exemple sans procédure

Soient les déclarations de variables suivantes.

```
long t = 2;
long x = 4;
long y = 5;
long z = 6;

int a = 1;
int b = 0;
int c = 1;
int d = 0;

register long *p1, *p2;
register long i1, i2;
```

Considérons un extrait de code, qui réalise quatre échange de contenus de deux variables.

```
void main ()
{
x--;

/* échanger les contenus de x et y si a != 0 */
if (a != 0)
{
p1  = &x;
p2  = &y;
i1  = *p1;        /* debut sequence d'instructions commune */
i2  = *p2;
*p1 = i2;
*p2 = i1;         /* fin   sequence d'instructions commune */
}

y = y-4;

/* échanger les contenus de x et z si b != 0 */
if (b !=0)
{
p1  = &x;
p2  = &z;
i1  = *p1;        /* debut sequence d'instructions commune */
i2  = *p2;
*p1 = i2;
*p2 = i1;         /* fin   sequence d'instructions commune */
}
z = z+4;

/* échanger les contenus de y et z  si c != 0 */
if (c != 0)
{
p1  = &y;
p2  = &z;
i1  = *p1;        /* debut sequence d'instructions commune */
i2  = *p2;
*p1 = i2;
*p2 = i1;         /* fin   sequence d'instructions commune */
}
x++;

/* échanger les contenus de t et z si d != 0*/
if (d != 0)
{
p1  = &t;
p2  = &z;
i1  = *p1;        /* debut sequence d'instructions commune */
i2  = *p2;
*p1 = i2;
*p2 = i1;         /* fin   sequence d'instructions commune */
}
```

```
t++;
}
```

10.1.3 Exemple avec procédures sans paramètre

Déclarons une procédure d'échange et une procédure milieu qui englobe les deux deuxième et troisième échanges :

```
void echanger ()
{                 /* prologue */
                  /* début du corps */
i1  = *p1;                              /* 4 , 13 , 22, 32 */
i2  = *p2;                              /* 5 , 14 , 23, 33 */
*p1 = i2;                               /* 6 , 15 , 24, 34 */
*p2 = i1;                               /* 7 , 16 , 25, 35 */
                  /* fin du corps */
}                 /* épilogue + retour */  /* 8 , 17 , 26, 36 */

void milieu()
{                 /* prologue */
                  /* debut du corps */

if (b != 0)
{
p1 = &x;                                                    /* 11 */
p2 = &z;                                                    /* 12 */
echanger();       /* B */                                   /* 13 */
suiteB:
}

z = z+4;                                                    /* 18 */

if (c != 0)
{
p1  = &y;                                                   /* 19 */
p2  = &z;                                                   /* 20 */
echanger();       /* C  */                                  /* 21 */
suiteC:
x++;                                                        /* 27 */
}
}                 /* epilogue+retour    */                  /* 28 */
```

Le programme modifié contient un appel de la procédure milieu et quatre de la procédure échanger : la suite d'instructions dans le corps de la procédure échanger est exécutée quatre fois.

L'ordre d'exécution des instructions est noté en commentaire. La fin de chaque procédure contient une instruction de branchement retour. Chaque appel de procédure contient une instruction de branchement vers le prologue de la procédure.

L'ordre des branchements de retour est l'inverse de celui des branchements aller. Le premier retour est exécuté par la dernière procédure appelée : cette propriété est dite "LIFO" (Last In First Out).

```
void main ()
{
x--;                                                    /* 0 */

if (a != 0)
{
p1 = &x;                                                /* 1 */
p2 = &y;                                                /* 2 */
echanger();          /* A  : point d'appel de echanger */ /* 3 */
suiteA:
}

y = y-4;                                                /* 9 */

milieu ();                                              /* 10 */
suiteM:

if (d != 0)
{
p1  = &t;                                               /* 29 */
p2  = &z;                                               /* 30 */
echanger();         /* D */                             /* 31 */
suiteD:
t++;                                                    /* 37 */
}
}                                                       /* 38 */
```

10.1.4 Gestion des branchements aller et retour

La traduction de cet exemple en langage d'assemblage de notre processeur RISC fictif n'appelle pas de commentaire particulier excepté pour la traduction des branchements aller et retour.

Il convient de définir une convention de stockage pour le paramètre implicite qu'est l'adresse de retour : elle pourrait être stockée en mémoire à une adresse définie statiquement (par exemple dans bss), ou dans un registre du processeur. Nous choisissons de la stocker dans le dernier registre général du processeur : que nous appelerons link register. Le symbole **lr** sera un synonyme de $r31$..

Les instructions $b_{cond}l$ (branch and link) et **jmpl** (jump and link) sont destinées aux appels de procédures. Avant de modifier le compteur ordinal, elles le sauvegardent dans le registre **lr**. Au moment de la sauvegarde, le compteur ordinal repère l'instruction qui suit l'instruction de branchement $b_{cond}l$ ou **jmpl**. L'adresse de branchement de retour contenue dans **lr** à la fin de la procédure échanger est suite1 lors de l'appel A, suiteB lors de l'appel B, suiteC lors de l'appel C et suiteD lors de l'appel D.

Dans l'épilogue de la procédure, le branchement retour à l'adresse contenue dans **lr** correspond à l'instruction **jmp lr**.

10.1.5 Traduction de l'exemple

```
          .data
t:        .word  2
x:        .word  4
y:        .word  5
z:        .word  6
a:        .word  1
b:        .word  0
c:        .word  1
d:        .word  0


          @ affectation arbitraire des registres aux variables
          @ r0 : i1    r1: p1   r2:i2   r3:p3    r4,r5: temporaires */
          .text


/* traduction de l'appel de la procédure échanger sans paramètre */


echanger:  @ prologue de echanger : vide ici
           @ il faudra ajouter sauver registres  modifiés */
           ldr r0, [r1]       @ i1 = *p1
           ldr r2, [r3]       @ i2 = *p2
           str r2, [r1]       @ *p1 = i2
           str r0, [r3]       @ *p2= r3
           @ epilogue de echanger : vide ici
           @ il faudra ajouter restaurer registres modifiés */
           jmp   lr           @ branchement retour :
                              @ à suiteA au premier retour
                              @ à suiteB au deuxième retour
                              @ à suiteC au troisième retour
                              @ à suiteD au quatrième retour

milieu:    @ prologue de milieu : vide ici
           @ il faudra ajouter sauver registres modifiés */
           mov32 r5, #b
           ldr   r4, [r5]
           cmp   r4, #0
           beq   suiteB
           mov32 r1, #x        @ p1 =&x
           mov32 r3, #z        @ p2 =&z
           bl    echanger      @ echanger() : lr <- suiteB; b echanger

suiteB:                        @ z = z + 4
           mov32  r5, #z
           ldr    r4, [r5]
           sub    r4, r4, #4
           str    r4, [r5]

           mov32 r5, #c
```

```
          ldr   r4, [r5]
          cmp   r4, #0
          beq   suiteC
          mov32 r1, #y        @ p1 =&y
          mov32 r3, #z        @ p2 =&z
          bl    echanger      @ echanger() : lr <- suiteC; b echanger

suiteC:                       @ x ++
          mov32  r5, #x
          ldr    r4, [r5]
          sub    r4, r4, #1
          str    r4, [r5]

          @ epilogue de milieu : vide ici
          @ il faudra ajouter restaurer registres modifiés */
          jmp   lr           @ branchement retour (a suiteM)

main:     @ prologue de main : vide ici
          @ prévoir l'ajout de sauver registres modifiés
                              @ x--
          mov32  r5, #x
          ldr    r4, [r5]
          sub    r4, r4, #1
          str    r4, [r5]

          mov32 r5, #a
          ldr   r4, [r5]
          cmp   r4, #0
          beq   suiteA
          mov32 r1, #x        @ p1 =&x
          mov32 r3, #y        @ p2 =&y
          bl    echanger      @ echanger() : lr <- suiteA; b echanger

suiteA:                       @ y = y - 4
          mov32  r5, #y
          ldr    r4, [r5]
          sub    r4, r4, #4
          str    r4, [r5]

          bl     milieu       @ milieu () : lr <- suiteM; b milieu

suiteM:   mov32 r5, #d
          ldr   r4, [r5]
          cmp   r4, #0
          beq   suiteE
          mov32 r1, #t        @ p1 =&t
          mov32 r3, #z        @ p2 =&z
          bl    echanger      @ echanger() : lr <- suiteD; b echanger

suiteD:                       @ t ++
          mov32  r5, #t
          ldr    r4, [r5]
          sub    r4, r4, #1
          str    r4, [r5]
```

```
          @ épilogue de main : vide ici
          @ prévoir la restauration des registres modifiés
          jmp    lr           @ retour au code qui a appelé main
```

Notons que les branchements aller pourraient être réalisés en utilisant l'instruction **jmpl** au lieu de **bl**.

```
@ Variante utilisant jmpl au lieu de bl
suiteM:   mov32 r5, #d
          ...
          mov32 r3, #z        @ p2 =&z
          mov32 r4, #echanger
          jmpl  r4            @ lr <- suiteD, saut absolu à échanger
suiteD:
```

10.2 Passage de paramètre par valeur et par adresse

Il est souvent utile de passer explicitement lors de l'appel des arguments permettant de paramétrer le fonctionnement de la procédure. En C, il n'existe qu'un seul mode de passage de paramètre : par valeur. Chaque argument passé lors de l'appel est une expression qui est évaluée et dont la valeur est copiée à l'endroit convenu pour le stockage des paramètres.

La procédure appelée peut modifier le contenu de son paramètre, mais le passage par valeur ne lui permet pas de modifier les variables utilisées dans l'expression passée en argument. Même si l'expression passée à l'appel se limite à une variable[1], la procédure ne peut modifier qu'une copie du contenu de cette variable stockée dans son paramètre, ce qui n'affecte pas la variable passée en paramètre.

Il existe une notion de paramètre de type résultat dans des langages tels que PASCAL ou ADA, mais pas en C. Dans d'autre langages tels que FORTRAN, les variables sont passées par adresse, ce qui permet à la procédure appelée de les modifier.

Bien le C ne propose que le passage par valeur, il est possible de réaliser un passage par adresse grâce aux pointeurs. En passant en paramètre la valeur d'un pointeur qui repère une variable, il est possible de modifier cette variable dans le corps de la procédure appelée via l'opérateur ***.

Voici à titre d'exemple un programme définissant et utilisant deux procédures lire_entier et écrire_entier.

Les procédures qui doivent modifier une variable de l'appelante utilisent un paramètre de type pointeur : les procédures de lecture au clavier entrent dans cette catégorie. Celles qui n'ont besoin que d'une valeur utilisent des paramètres de type normal : c'est notament le cas des procédures d'affichage à l'écran.

```
/*************************************************************/
/* procédure ecrire_entier                                   */
/* affiche en binaire un entier 32 bits à l'écran            */
/*************************************************************/
```

[1] variable simple ou élément de tableau ou membre de structure

```
void ecrire_entier (int x)
{
int i;
for (i=0; i<32;i++)
  {
  if (x<0)              /* x<0 si bit de poids fort == 1 */
     putchar ('1');
  else
     putchar ('0');
  }
  x = x << 1;           /* décaler d'un bit à gauche */
}

/*****************************************************************/
/* procédure lire_entier                                         */
/* lit en binaire un entier 32 bits au clavier                   */
/*****************************************************************/

void lire_entier (int *x)
{
int r;
r = 0;
for (i=0; i<32;i++)
  {
  /* Vérification omise : getchar doit retourner '0' ou '1' */
  r = r << 1 + (getchar () - '0');
  }
*x = r;
}

int a;

void tester ()
{
lire_entier (&a);
ecrire_entier (3*a+1);
}
```

10.3 Sauvegarde et restauration des registres

10.3.1 Principe

L'intérêt du mécanisme d'appel de procédure ne se limite pas à la réduction de la taille du code machine. Les procédures constituent également un élément de structuration des programmes. Une fois les interfaces d'appel clairement définies, le corps de la procédure appelée peut être écrit sans connaître le corps de la ou les procédures qui l'appelent, et réciproquement. Le mécanisme de gestion des procédures ne doit pas imposer de contraintes sur les appels : n'importe quelle procédure doit pourvoir être appelée dans le corps de n'importe quelle autre procédure.

Toutes les procédures utilisent au moins une partie d'un ensemble de ressources communes : les registres du processeur. Ceci pose un problème lorsque l'appelée modifie le contenu d'un registre que l'appelante a déjà utilisé : sans précaution particulière, au retour de l'appel de procédure l'information que l'appelante y avait stockée est perdue.

On pourrait imaginer de faire en sorte que l'appelante et l'appelée utilise des ensembles de registres disjoints. Cette stratégie n'est pas applicable en pratique : si l'intersection de leurs ensembles de registres modifiés n'est pas vide, deux procédures ne peuvent pas s'appeler entre elles.

La solution générale consiste à effectuer une sauvegarde en mémoire du contenu des registres avant exécution du corps de la procédure appelée et une opération inverse de restauration après exécution du corps.

Plusieurs stratégies sont envisageables :
- sauvegarde par l'appelante : avant d'exécuter le branchement aller, l'appelante sauvegarde en mémoire le contenu des registres dans lesquelles elle a stocké des informations.
- sauvegarde par l'appelée : dans le prologue et l'épilogue, la procédure appelée sauvegarde et restaure les registres sont elle modifie le contenu.
- absence de sauvegarde : l'appelante fait en sorte de ne stocker dans les registres que des informations qui ne sont plus utiles au moment où un appel de procédure est exécuté (cette stratégie n'est applicable qu'à un ou quelques-uns des registres du processeur)
- approches mixtes : des politiques de sauvegarde différentes sont appliquées chacune à une partition de l'ensemble des registres du processeur.

Pour notre processeur RISC de référence, nous choisissons de ne pas sauver un des registres généraux lors des appels de procédures. Ce registre sera désigné sous le nom **ip** (intraprocédure : conserve sa valeur tant que l'on effectue pas d'appel de procédure), synonyme de **r28**. Ce registre sera entre autre utilisé comme temporaire de stockage de l'adresse de la zone de sauvegarde des registres.

Nous utiliserons la convention suivante : la procédure appelée sauvegarde dans le prologue et restaure dans l'épilogue le contenu de tous les registres généraux qu'elle modifie, excepté **ip**.

10.3.2 Instructions ldm et stm

Pour la sauvegarde des registres, nous supposerons que notre processeur RISC fictif dispose des instructions ldm et stm pour transférer une suite de mots entre un ensemble de registres et un bloc de mots contigus en mémoire.

L'instruction **stm rp, {liste de registres}** écrit le contenu des x registres spécifiés dans la liste dans un bloc de x mots consécutifs en mémoire dont la limite est repérée par un registre général quelconque **rp**. L'instruction **ldm rp, {liste de registres}** effectue le transfert en sens inverse.

La liste de registres est composée d'éléments séparés par des virgules. Chaque élément peut être un registre unique ou un ensemble de registres dont les numéros forment un intervalle : **{r0, r3-r6, r8}** décrit l'ensemble de registres **{r0, r3, r4, r5, r6, r8}**. L'ordre de rangement est prédéfini : les registres de numéros croissants sont stockés à des adresses croissantes.

Le registre **rp** peut repérer une case à l'intérieur ou à l'extérieur au début ou à la fin du bloc en mémoire. La position de **rp** par rapport aux cases mémoires à accéder définit quatre possibilités

illustrées par la figure 10.1 : increment after (**ia**), increment before (**ib**), decrement after (**db**), decrement before (**db**). Le suffixe indique si l'adresse fournie par **rp** doit être incrémentée ou décrémentée avant ou après le transfert du premier mot.

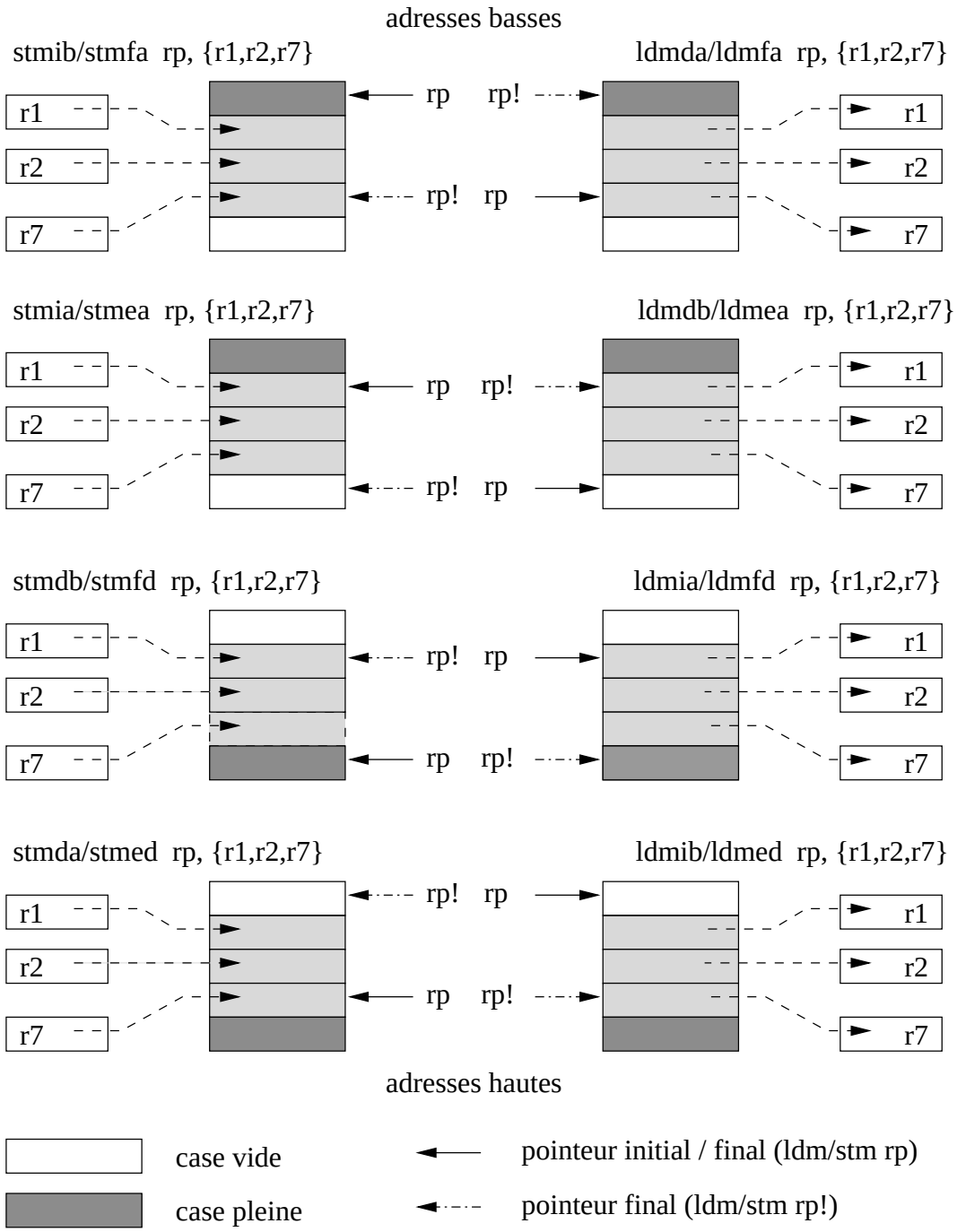


FIG. 10.1 – Comportement des instructions ldm et stm

Les versions normales de ldm et stm ne modifient pas le contenu du registre pointeur **rp**. Les variantes **ldmxx/stmxx rp !, {liste de registres}** décalent l'adresse contenue dans le registre pointeur **rp** d'autant de mots que de registres transférés. Il peut alors être réutilisé tel que pour un nouveau transfert de même type.

Notons qu'un transfert décrit par une instruction stm peut être réalisé par une séquence d'ins-

truction str ordinaires [2]

```
@ séquence d'instructions équivalentes à stmdb sp!, {r1, r2, r7}
@
       str  r7, [sp,#-4]!         @ ou str  r7, [sp, #-4]
       str  r2, [sp,#-4]!         @    str  r7, [sp, #-8]
       str  r1, [sp,#-4]!         @    str  r7, [sp, #-12]
                                  @    sub  sp, sp, #12
@
@ séquence d'instructions équivalentes à stmda sp!, {r1, r2, r7}
@
       str  r7, [sp], #-4         @ ou str  r7, [sp, #0]
       str  r2, [sp], #-4         @    str  r7, [sp, #-4]
       str  r1, [sp], #-4         @    str  r7, [sp, #-8]
                                  @    sub  sp, sp, #12
@
@ séquence d'instructions équivalentes à ldmia  r0, {r5-r7}
@
       ldr r5, [r0, #0]
       ldr r6, [r0, #4]
       ldr r7, [r0, #8]
@
@ séquence d'instructions équivalentes à ldmib  r0, {r5-r7}
@
       ldr r5, [r0, #4]
       ldr r6, [r0, #8]
       ldr r7, [r0, #12]
```

Les instructions **ldm** et **stm** admettent des suffixes synonymes **fd, fa, ea, ed** dont la signification se rapporte à la notion de pile (utilisée pour les appels de procédures autorisant la récursion).

10.4 Gestion des paramètres et des variables locales

La déclaration d'une procédure définit une liste (éventuellement vide) de paramètres formels en spécifiant leur type. L'instruction d'appel fournit alors une liste de paramètres réels définissant les valeurs des paramètres pour cet appel.

Notons que l'appel fournit systématiquement un paramètre implicite : l'adresse destination du branchement de retour en fin de procédure.

10.4.1 Convention d'appel et stockage des paramètres

Une convention d'appel doit être définie de telle sorte que la procédure appelée puisse déterminer où trouver le contenu des paramètres que la procédure appelante lui a passé.

Les deux conventions de passage des paramètres les plus simples sont :

1. stockage dans les registres du processeur

[2] cf 6.9 pour les variantes de ldr et str avec préincrémentation et postincrémentation

2. stockage dans une zone mémoire allouée statiquement à chaque procédure, par exemple dans la section bss.

3. approche mixte : passage des p premiers paramètres dans les registres du processeur et stockage des n-p autres paramètres dans bss.

La premère stratégie est efficace, mais le nombre d'arguments ne peut pas dépasser le nombre de registres disponibles. La deuxième stratégie accepte un nombre quelconque (mais fixé) d'arguments, mais au prix de deux accès mémoire par argument (une écriture par l'appelante et une lecture par l'appelée).

Dans ce chapitre, nous utiliserons la convention suivante : l'adresse de retour (paramètre implicite) est passée dans le registre **lr** du processeur, le premier paramètre explicite est passé dans le premier registre **r0**, les autres paramètres sont passés en mémoire. Le choix du nombre de paramètres explicites passés dans les premiers registres du processeur (ici un) est arbitraire[3] et tient compte du nombre total de registres.

10.4.2 Stockage des variables locales

Comme pour les paramètres, trois approches sont possibles pour le stockage des variables locales :

1. stockage dans les registres du processeur

2. stockage dans une zone mémoire allouée statiquement à chaque procédure,

3. approche mixte : quelques variables stockées dans les registres du processeur et les autres en mémoire.

Notons que la première la première méthode est plus efficace que la deuxième, mais n'est pas applicable

- lorsque le nombre de variables excède le nombre de registres disponibles
- aux variables dont on prend l'adresse.

Dans ce chapitre, nous utiliserons la convention suivante : toutes les variables locales sont stockées en mémoire.

10.5 Exemple avec paramètres et variables locales

Dans notre exemple, il est judicieux de définir des procédures recevant des paramètres explicites plutôt que de passer les informations via les variables globales p1 et p2. Les variables i1 et i2, qui ne sont utilisées que dans la procédure échanger, devennent des variables locales de celle-ci.

Cond, cond1 et cond2 illustrent le passage de paramètre par valeur, et les paramètres pointeurs p et q correspondent à un passage d'adresses des variables à échanger.

[3]Dans la convention utilisée par le compilateur C GNU, il est de 4 pour le ARM et 6 pour le SPARC

```
long t = 2;
long x = 4;
long y = 5;
long z = 6;

int a = 1;
int b = 0;
int c = 1;
int d = 0;
```

```
void echanger (int cond, long *p, long *q)
{
long i1;
long i2;
if (cond != 0)
  {
  i1  = *p;
  i2  = *q;
  *p  = i2;
  *q  = i1;
  }
}
```

```
void main ()
{
x--;
echanger(a,&x,&y);
y = y-4;
milieu(b,c);
echanger(d,&t, &z);
t++;
}
```

```
void milieu (int cond1, int cond2)
{
echanger(cond1,&x,&z);
z = z+4;
echanger(cond2,&y,&z);
x++;
}
```

10.6 Traduction de l'exemple

Examinons la traduction des deux procédures de l'exemple avec paramètres.

La procédure échanger reçoit deux paramètres de type entier long, il convient donc d'allouer 8 octets dans la section bss pour cette procédure.

Dans le corps de échanger, on ne prend pas l'adresse des variables i1 et i2. Il serait donc possible de les stocker dans les registres r0 et r2 plutôt que dans bss, auquel on peut supprimer quatre instructions d'accès à la mémoire (commentées "opt" dans le code) et réduire à 0 la constante ECHANGER_TAILLE_VAR.

Pour répérer les paramètres recus de l'appelante et les variables locales, nous utiliserons un registre à usage général, de nom **fp** (function parameters)[4], synonyme de **r29**.

Pour repérer les paramètres passés à la procédure appelée, nous utiliserons un registres à usage général de nom **sp** (sommet de pile[5]), synonyme de **r30**.

La figure 10.2 représente l'état de la mémoire et des registres juste après la troisième exécution de **bl echanger**.

La procédure main a utilisé le registre **sp** pour initialer les paramètres **cond1** et **cond2** de milieu. Dans le prologue de milieu, le registre **lr** qui contenait l'adresse de retour dans main a été sauvegardé dans la zone **param_milieu**.

[4] le nom officiel est frame pointer : soit pointeur de cadre en français

[5] l'explication de ce nom est expliqué dans le chapitre consacré à la gestion de procédures avec récursion

Le corps de **milieu** utilise le registre pointeur **fp** pour repérer les paramètres reçus de **main** et **sp** pour initialiser les paramètres passés à **échanger**.

La procédure **échanger** sauvegardera les registres qu'elle modifie et allouera de la mémoire pour ses variables **i1** et **i2** dans la zone mémoire réservée avant **param_échanger**. Elle modifiera **fp** pour repérer les paramètres qu'elle a reçus de **milieu**.

```
                ECHANGER_TAILLE_PARAM=8          /* taille des args */
                ECHANGER_P    = 0   /*  emplacement relatif de p    */
                ECHANGER_Q    = 4   /*  emplacement relatif de q    */

                ECHANGER_TAILLE_REGS=16 /* sauvegarde des registres */

                ECHANGER_TAILLE_VAR=8            /* taille des vars */
                                        /* emplacement relatif de i1 */
                ECHANGER_I1=-(4+ECHANGER_TAILLE_REGS)
                                        /* emplacement relatif de i2 */
                ECHANGER_I2=-(8+ECHANGER_TAILLE_REGS)


                .bss
                .skip  ECHANGER_TAILLE_VAR + ECHANGER_TAILLE_REGS
param_echanger:   .skip  ECHANGER_TAILLE_PARAM

          .text
echanger: @ prologue de echanger

          @ affectation des registres :
          @ r0 : paramètre cond
          @ r1 : copie du paramètre p
          @ r2 : valeur de i2
          @ r3 : copie du paramètre q
          @ r4 : valeur de i1
          @ fp : pointeur de paramètres

          @ sauvegarde des registres
          mov32  ip, #param_echanger - ECHANGER_TAILLE_REGS
          stmia ip, {r1-r4,fp}

          @ test de cond
          cmp r0, #0
          beq fin_echanger
          mov32 fp, #param_echanger
          @ récupération des paramètres p et q dans r1 et r3
          ldr r1, [fp, #ECHANGER_P]
          ldr r3, [fp, #ECHANGER_Q]

          ldr r4, [r1]                  @ i1 = *p
          str r4, [fp, #ECHANGER_I1]                  @ opt

          ldr r2, [r3]                  @ i2 = *q
```

```
            str r2, [fp, #ECHANGER_I2]                  @ opt


            ldr r2, [fp, #ECHANGER_I2]   @ *p = i2      @ opt
            str r2, [r1]

            ldr r4, [fp, #ECHANGER_I1]   @ *q = i1      @ opt
            str r4, [r3]

fin_echanger:
            @ epilogue de echanger :
            @ restaurer contenu initial des registres
            ldmia ip, {r1-r4, fp}

            jmp   lr           @ branchement retour :

                  MILIEU_TAILLE_PARAM=4          /* taille des args */
                  MILIEU_COND2=0               /* position de cond2 */
                  MILIEU_TAILLE_REGS=24 /* sauvegarde des registres */

                  MILIEU_TAILLE_VAR=0            /* taille des vars */

                 .bss
                 .skip  MILIEU_TAILLE_VAR + MILIEU_TAILLE_REGS
param_milieu:    .skip  MILIEU_TAILLE_PARAM

milieu:     @ prologue de milieu
            @ Affectation des registres
            @ r0 : premier paramètre reçu/passé
            @ r1 : temporaire adresse
            @ r2 : temporaire valeur
            @ sauvegarde des registres
            mov32  ip, #param_milieu - MILIEU_TAILLE_REGS
            stmia ip, {r0-r2,sp,fp,lr}
            @ corps de milieu
            mov32 sp, #param_echanger
            mov32 fp, #param_milieu
            @ le paramètre cond1 est déjà dans r0
            mov32 r1, #x                    @ echanger (cond1,&x, &z)
            str   r1, [sp, #ECHANGER_P]
            mov32 r1, #z
            str   r1, [sp, #ECHANGER_Q]
            bl    echanger                  @ detruit le contenu de ip

            mov32  r1, #z                   @ z = z + 4
            ldr    r2, [r1]
            sub    r2, r2, #4
            str    r2, [r1]

            ldr   r0, [fp, #MILIEU_COND2]   @ echanger (cond2,&x, &z)
            mov32 r1, #y
            str   r1, [sp, ECHANGER_P]
```

```
            mov32 r1, #z
            str   r1, [sp, ECHANGER_Q]
            bl    echanger                  @ detruit le contenu de ip

            mov32  r1, #x                   @ x++
            ldr    r2, [r1]
            sub    r2, r2, #1
            str    r2, [r1]

            @ epilogue de milieu
            @ restaurer registres modifiés */
            mov32  ip, #param_milieu - MILIEU_TAILLE_REGS
            ldmia ip, {r0-r2,sp,fp,lr}
            jmp   lr

main:       ...
            @ extrait du code de main : appel de milieu
            @
            mov32  sp, param_milieu
            mov32  r1, #b             @ param cond1 = b
            ldr    r0, [r1]
            mov32  r1, #c             @ param cond2 = c
            ldr    r1, [r1]
            str    r1, [sp, #MILIEU_COND2]
            bl milieu
            ...
```

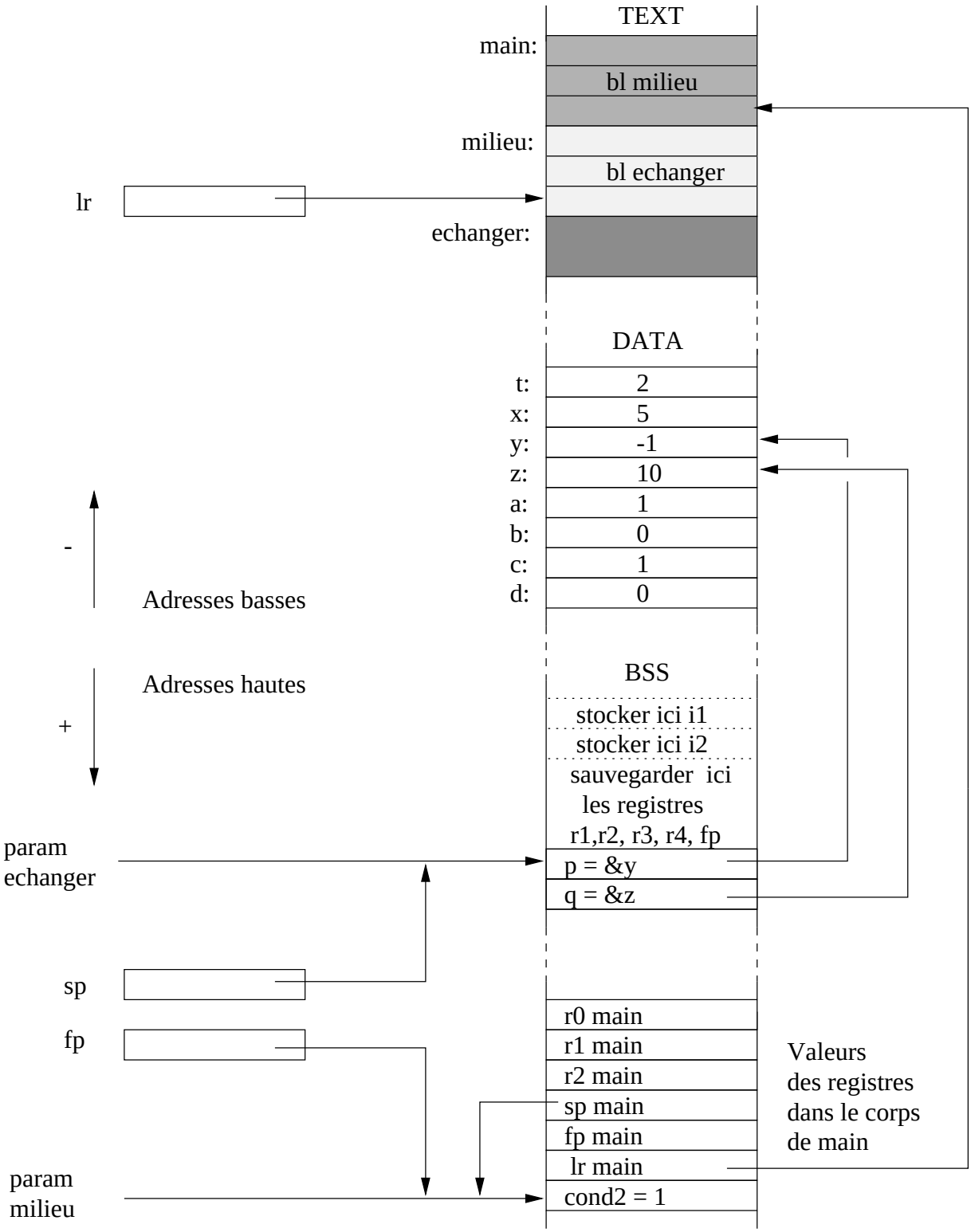


FIG. 10.2 – Etat de la machine dans le prologue de échanger

Chapitre 11

Procédures avec récursion

11.1 Notion de récursion

Les appels de fonctions définissent une relation de filiation entre les fonctions. Le graphe de cette relation possède un arc d'une fonction g vers une fonction f si et seulement si g contient un appel de la procédure f.

Dans un programme récursif, le graphe de filiation contient (au moins) un cycle (d'appels en cascade). Un cycle de longueur un correspond à une récursion directe et un cyle plus long à une récursion indirecte. Une fonction récursive directe contient un appel à elle-même.

Considérons par exemple un ensemble de fonctions récursives impliquées dans un cycle de longueur trois : la première fonction contient un appel à la deuxième fonction, qui contient un appel à la troisième, cette dernière incluant un appel à la première.

La notion de récursion en programmation correspond à la notion mathématique de récurrence.

11.1.1 Exemple de récursion directe : la suite de Fibonacci

La suite de Fibonacci est définie comme suit : $u_0 = u_1 = 1$ et $u_n = u_{n-1} + u_{n-2}$ pour $n > 1$.
Voici un exemple de récursion directe pour calculer cette suite.

```
unsigned long x = 4;
unsigned long resultat;
int m = 0;

void fibo (unsigned long *s, unsigned long n)
{
unsigned long f;
unsigned long res;
res = n;
if (n > 1)
  {
  fibo (&res,n-1);
  fibo (&f,n-2);
  res = res + f;
  }
*s = res;
}
```

```
int main ()
{
long m;
   m = 6;
   fibo (&resultat,x);
}
```

Voici une trace des appels et des retours de procédure générés par le calcul de Fibonacci(4).

```
Appel de fibo (...,4)
   Appel de fibo (...,3)
      Appel de fibo (...,2)
         Appel de fibo (...,1)
         Retour de fibo(...,1)
         Appel de fibo (...,0)
                                /* point d'observation F */
         Retour de fibo(...,0)
      Retour de fibo(...,2)
      Appel de fibo (...,1)
      Retour de fibo(...,1)
   Retour de fibo(...,3)
   Appel de fibo (...,2)
      Appel de fibo (...,1)
      Retour de fibo(...,1)
      Appel de fibo (...,0)
      Retour de fibo(...,0)
   Retour de fibo(...,2)
Retour de fibo(...,4)
```

11.1.2 Exemple de récursion indirecte : calcul de $\Sigma_{i=0}^{n} i$

Voici un exemple de calcul de la somme $1+2+3\cdots+n-1+n$ par un programme utilisant une récursion indirecte [1].

```
unsigned long x = 4;
unsigned long resultat;
int m=0;

extern void sigma_pair (unsigned long, unsigned long *);
extern void sigma_impair (unsigned long, unsigned long *);

void sigma_pair (unsigned long n, unsigned long *s)
{
unsigned long f;
if (n==0)
  f = 0;
else
  {
  sigma_impair (n-1,&f);
```

[1] Cette somme peut évidement être calculée itérativement ou avec la formule $\Sigma_{i=0}^{n} i = \frac{n(n+1)}{2}$.

```
  f = f + n;
  }
*s = f;
}

void sigma_impair (unsigned long n, unsigned long *s)
{
unsigned long f;
if (n==1)
  f = 1;
else
  {
  sigma_pair (n-1,&f);
  f = f + n;
  }
*s = f;
}

void sigma (unsigned long n, unsigned long *sigma)
{
if ((n %2) == 0)
  sigma_pair(n,sigma);
else
  sigma_impair(n,sigma);
}

int main ()
{
   sigma (x,&resultat);
   printf ("sigma(%d) = %d\n",x,resultat);
}
```

Voici une trace des appels et des retours générés par l'appel de sigma(4,...).

```
Appel de sigma(4,...)
   Appel de pair (4,...)
                                    /* point d'observation de la pile */
      Appel de impair (3,...)
         Appel de pair (2,...)
            Appel de impair (1,...)
                                    /* point d'observation S */
            Retour de impair(1,...)
         Retour de pair(2,...)
      Retour de impair(3,...)
   Retour de pair(4,...)
Retour de sigma(4,...)
```

11.1.3 Contraintes spécifiques liées à la récursion

La récursion se traduit par la coexistence simultanée de plusieurs instances d'appel de la même procédure (avec des paramètres différents).

Au point d'observation F de l'exemple Fibonacci, cinq instances d'exécutions de la procédure fibo sont en cours, avec différentes valeurs du paramètre n : 0, 1, 2, 3 et 4. Il en va de même dans l'autre exemple : au point d'observation S, il existe deux instances d'exécution de sigma_impair (pour n=1 et n=3) et deux de sigma_pair (pour n=0 et n=2).

Il n'est donc pas possible d'allouer statiquement une zone de stockage de paramètres et de variables locales à chaque procédure comme nous l'avons fait au chapitre précédent. Il est nécesaire d'allouer dynamiquement une zone de stockage à chaque nouvel appel de procédure et de la libérer lors du retour.

Chaque instance d'exécution de procédure utilise trois blocs de mémoire distincts :

1. un bloc mémoire de paramètres reçus partagé avec et alloué par la procédure appelante,
2. un espace privé de stockage de variables locales, de temporaires et de sauvegarde de registres, alloué par l'instance d'exécution de la procédure en cours,
3. un bloc de paramètres transmis, alloué par la procédure courante lorsqu'elle appelle à son tour une autre procédure, partagé avec cette dernière.

De plus, la mise en œuvre de l'allocation et la libération de mémoire lors des appels et retours ne peut pas être réalisé sous forme d'appel de procédure ordinaire[2].

Remarquons que les allocations et libérations respectent la propriété LIFO des appels et des retours : le dernier bloc de mémoire alloué (lors du dernier appel) est le premier bloc libéré lors du (premier) retour.

11.2 Allocation et libération de blocs dans la pile

11.2.1 Notion de pile

La zone mémoire utilisée pour l'allocation dynamique de mémoire liée aux appels de procédures est appelée la pile. La pile se présente comme un tableau d'octets alloué implicitement par le système d'exploitation ou déclaré explicitement par le programmeur (cas d'applications embarquées sans système d'exploitation).

```
@ exemple de déclaration explicite d'une pile dans bss par le programmeur
@ l'allocation est normalement realisee implicitement par le systeme
@ d'exploitation, dans une zone pile distincte de bss.

TAILLE_PILE = 100000         @ taille arbitraire : 100 Ko

                  .bss
debut_file:       .skip  TAILLE_PILE
fin_pile:
                  .text
       @ initialisation du registre sommet de pile a réaliser
       @ avant exécution du programme principal (main)
       @ ici pour une pile de type "full descending"
init:             move32  sp, fin_pile
```

[2]Le mécanisme ordinaire d'appel de procédure ne peut pas à la fois utiliser le mécanisme d'allocation de mémoire et servir à le construire.

Les blocs alloués sont contigus et rangés en mémoire dans l'ordre de leur allocation. Le registre sommet de pile (sp) repère la limite entre la portion pleine de la pile allouée aux appels de procédures en cours et la portion vide utilisables pour de nouveaux appels.

Autrement dit, le sommet de la pile permet de repérer la limite entre le premier bloc à libérer parmi ceux déjà alloués et le prochain bloc qui sera alloué. Allouer ou libérer un bloc de T octets revient simplement à décaler le sommet de pile de T octets.

L'opération **empiler(x)** consiste à allouer un bloc dans le tableau pile, et à en initialiser le contenu avec la valeur de x. L'opération **x = dé(sem)piler()**[3] consiste à lire le contenu du dernier bloc et à libérer ensuite.

Pour empiler un ensemble de n mots, on peut effectuer n fois l'opération empiler, ou allouer un bloc de n mots, et remplir les mots de ce bloc ensuite.

Le registre sommet peut repérer le dernier octet déjà alloué ou le premier octet libre. D'autre part, l'allocation des premiers blocs peut être réalisée en début de pile (pile croissante) ou en fin de pile (pile décroissante).

Il existe donc quatre conventions possibles de pile :

1. pleine, croissante (full, ascending) : premiers blocs alloués en début de pile, sp repère le dernier octet déjà alloué (case pleine),
2. vide, croissante (empty, ascending) : premiers blocs alloués en début de pile, sp repère le premier octet libre (case vide),
3. pleine, décroissante (full descending) : premiers blocs alloués en fin de pile, sp repère le dernier octet déjà alloué,
4. vide, décroissante (empty descending) : premiers blocs alloués en fin de pile, sp repère le premier octet libre.

La figure 11.1 illustre l'allocation de trois blocs dans l'ordre 1, 2, 3 dans les quatre conventions de pile :

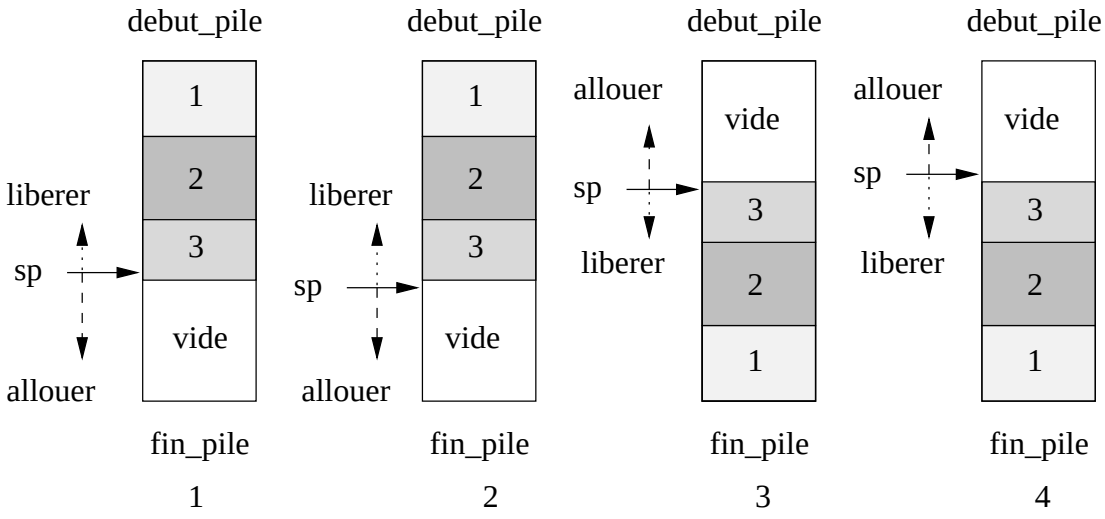


FIG. 11.1 – Les quatre conventions de pile

[3]Le terme correct est désempiler, mais l'usage du terme impropre dépiler est très répandu

Dans la suite de ce document, sauf précision contraire, nous utiliserons une pile décroissante avec sommet de pile repérant une case pleine (numéro 3 sur la figure).

Avec ce type de pile, **allouer(T)** signifie **sp = sp - T** et **libérer(T)** correspond à **sp = sp + T**. Il faut utiliser les instructions **str registre, [sp,#-4]!** ou **stmdb sp!, {liste_regs}** pour empiler le contenu d'un (ensemble de) registre(s). L'opération inverse (dé(sem)piler[4]), correspond aux instructions **ldr rd, [sp], #4** ou **ldmia sp!, {liste_regs}**

Croissance pile	Sommet	Empiler	Dé(sem)piler
descendante	plein	str rd, [sp,#-4]!	ldr rd, [sp], #4
descendante	vide	str rd, [sp],#-4	ldr rd, [sp, #4]!
ascendante	plein	str rd, [sp,#4]!	ldr rd, [sp], #-4
ascendante	vide	str rd, [sp],#4	ldr rd, [sp,#-4]!

TAB. 11.1 – Utilisation de ldr et str selon les conventions de pile

Pour le programmeur de procédures, il est plus commode de spécifier le type de pile utilisée que l'ordre dans lequel l'incrémentation ou la décrémentation et le premier accès à la mémoire doivent être réalisés. C'est pourquoi l'assembleur accepte les synonymes suivant pour ldm et stm :

Croissance pile	Sommet		Empiler	Synonyme	Dépiler	Synonyme
descendante	plein	(full)	stmfd	stmdb	ldmfd	ldmia
descendante	vide	(empty)	stmed	stmda	ldmed	ldmib
ascendante	plein	(full)	stmfa	stmib	ldmfa	ldmda
ascendante	vide	(empty)	stmea	stmia	ldmea	ldmdb

TAB. 11.2 – Utilisation de ldm et stm selon les conventions de pile

11.2.2 Allocation, libération, notion de lien dynamique

A l'état initial, le sommet de pile repère fin_pile [5]. L'allocation d'un bloc de taille T consiste à déplacer le sommet de pile de T octets vers les adresses basses : (**sp = sp - T**). Le sommet de pile repère alors le premier octet du bloc qui d'etre alloué. La libération consiste à l'inverse à déplacer le sommet de pile en sens inverse (**sp = sp + T**).

Chaque procédure reçoit un bloc de paramètres alloué et rempli par la procédure apppelante et s'alloue deux blocs contigus : l'un de taille L pour ses variables locales, temporaires et sauvegardes de registres, l'autre de taille A pour passer des arguments lors des appels de procédure qu'elle exécute dans son corps.

La figure 11.2 illustre l'état de la pile lors de la première exécution de la procédure sigma_pair. La partie gauche de la figure illustre l'état de la pile juste avant que le corps de la procédure sigma exécute le branchement à sigma_pair. La partie droite de la figure montre l'état de la pile lorsque le prologue de sigma_pair a été exécuté.

Au retour de la procédure, l'appelante libère l'espace mémoire alloué aux arguments dans la pile. La partie utilisée de la pile croît à chaque appel de procédure et décroit à chaque retour. Dans le cas d'une récursion directe, tous les blocs sont de même taille et toutes les instances d'exécutions

[4] le fonctionnement de ldm et stm a été présenté dans le chapitre consacré aux procédures sans récursion

[5] Rappel : nous avons choisi arbitrairement la convention (pile descendante, case pleine)

de la procédure récursive situées à la même profondeur dans l'arbre des appels utiliseront la même portion de mémoire.

Il est souvent commode de gérer deux registres pointeurs de pile : le sommet de pile **sp** qui pointe sur le début du bloc d'arguments à passer lors d'un prochain appel de procédure, et le pointeur d'arguments **fp** qui repère la limite entre le bloc de paramètres reçus et le bloc de variables locales. Cette technique est connue sous le terme de lien dynamique.

11.2.3 Principe de codage avec lien dynamique

Le code d'une procédure se compose d'un prologue, d'un corps et d'un épilogue.

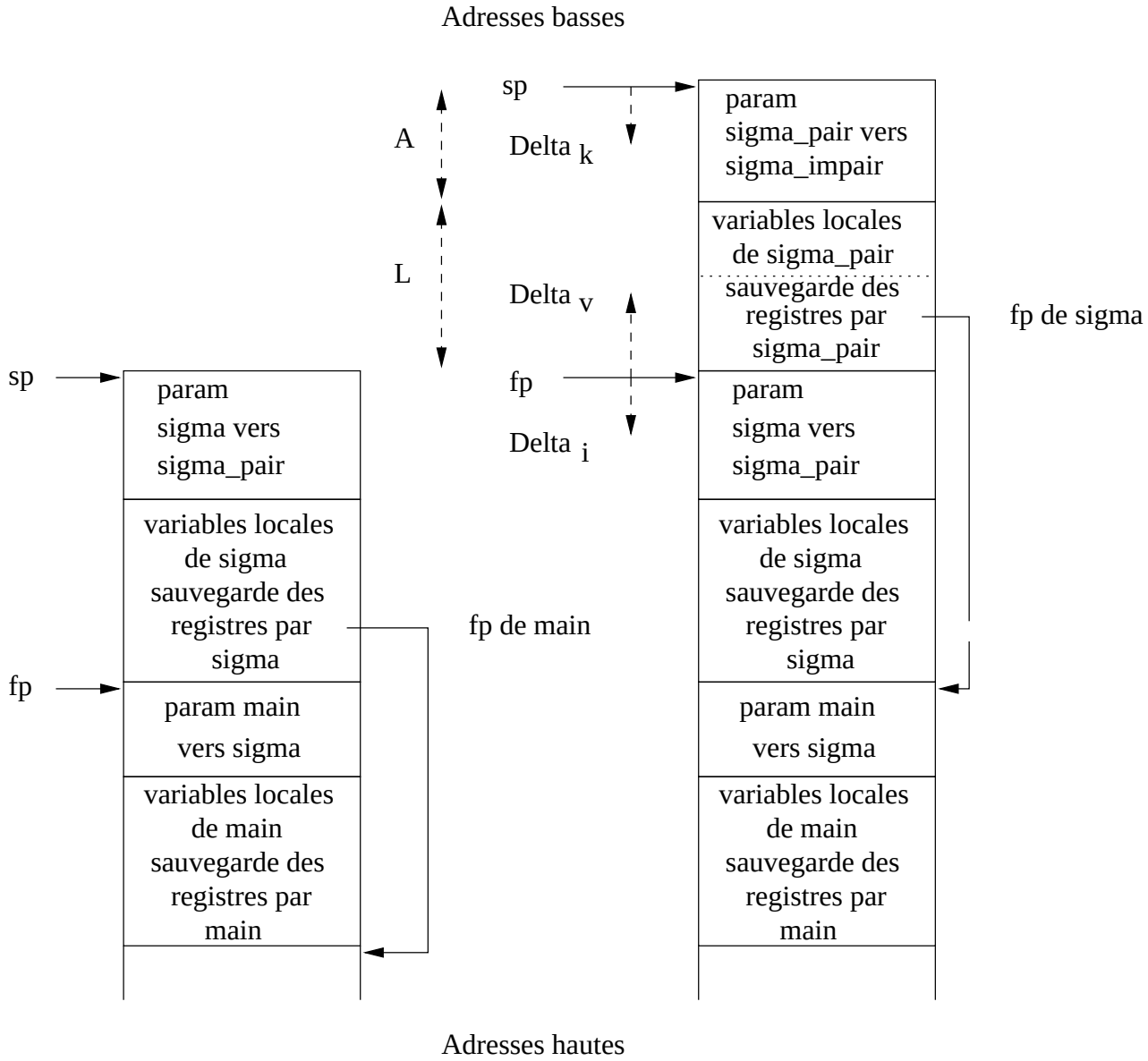


FIG. 11.2 – La pile avant et après exécution du prologue de sigma_pair

Au début du prologue, le sommet de pile **sp** repère le début du tableau d'arguments reçus que l'appelante a stockés dans la pile. Le code du prologue

1. sauvegarde les registres dans les mots mémoire qui précèdent les arguments reçus,

2. copie **sp** dans **fp** de telles sorte que **fp** repère la limite entre les arguments reçus et les sauvegardes de registres et
3. déplace le sommet de pile **sp** de L octets, L correspondant à l'espace mémoire occupé par les sauvegardes de registres et les variables locales (et éventuels temporaires en mémoire) de la procédure.

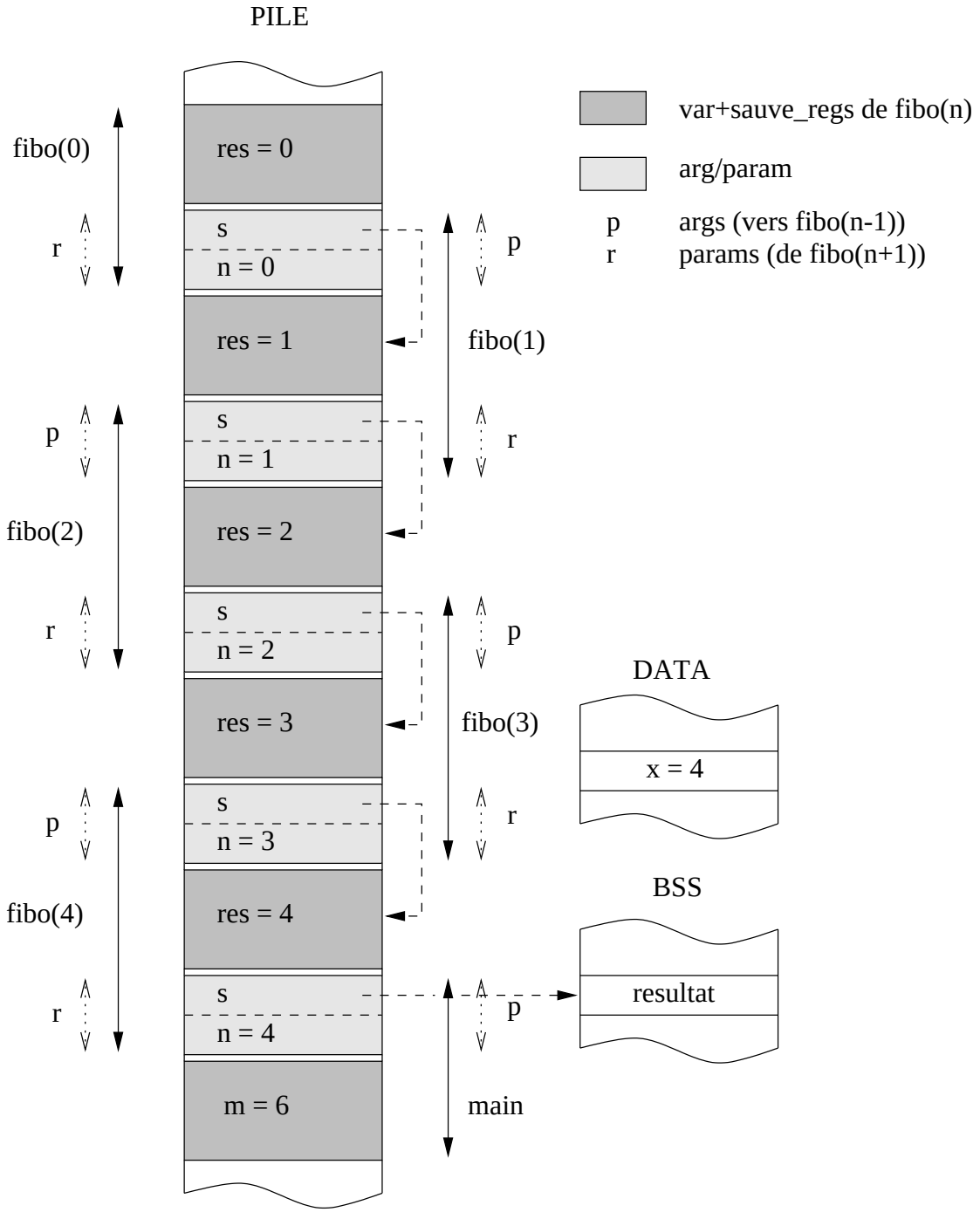


FIG. 11.3 – Fibonacci : état de la pile au point F

L'épilogue effectue le travail inverse du prologue et effectue le branchement de retour.

Chaque appel de procédure :

1. empile un tableau d'arguments,
2. effectue de branchement aller à la procédure et

3. libère la place allouée au tableau d'arguments

Dans le corps de la procédure, le $i^{ième}$ argument reçu de l'appelante est accessible à l'adresse $fp + \Delta_i$. L'adresse d'une variable locale v est $fp - \Delta_v$. Le $k^{ième}$ argument passé dans la pile lors d'un appel de procédure est à l'adresse $sp + \Delta_k$.

Le sommet de pile se déplace chaque fois que la procédure alloue ou libère de la place pour un argument a passer à une procédure qu'elle appelle. La position relative des arguments reçus et des variables locales de la procédure par rapport au sommet de pile change au gré des appels exécutés par la procédure. Il est alors commode d'y accéder via le deuxième pointeur de pile (fp), qui reste fixe dans tout le corps de la procédure.

La figure 11.3 illustre l'état de la pile au point S du calcul de la suite de fibonacci.

11.2.4 Exemple avec lien dynamique

```
extern void a1 (p0, p1, p2);
extern void a2 (p0, p1, ..., p5);

void b (int recu0, recu1, ..., recu5)
{
int v0, v2, ...,v7;

v2 = recu3;
...
a1 (10,11,12);
a2 (20,recu0,recu2+3,23,24,25);  /* 6 paramètres au total */
}

@ On suppose que b a 8 variables locales

          B_TAILLE_VARS = 8*4

          .text
b:        @ prologue de b           le sommet de pile est en sp0

          @ sauvegarde des registres
          @ On suppose que le corps de b modifie le registre r0
          @ ==> a sauver ainsi que fp et lr (adresse de retour)
          stmdb sp, {r0,fp,lr}          @ sp non modifié

          @ stm a sauvegardé 4 registres
          B_TAILLE_REGS = 3*4

          @ fp repere les arguments reçus et regs sauvegardés
          mov   fp, sp

          @ l'adresse de retour est accessible en fp - 4
          @ l'ancien fp         est accessible en fp - 8
          @ la sauvegarde de r0 est accessible en fp - B_TAILLE_REGS

          B_DELTA_lr = -4
```

```
B_DELTA_r0 = -TAILLE_REGS

@ les positions des paramètres reçus par rapport à fp

B_DELTA_recu0 = B_DELTA_r0  @ reçu0 dans la sauvegarde de r0
B_DELTA_recu1 = 0           @ reçu1 en début de tableau empilé
B_DELTA_recu2 = 4           @ par l'appelante de a
B_DELTA_recu3 = 8
B_DELTA_recu4 = 12
B_DELTA_recu5 = 16          @ reçu5 en fin de tableau empilé

@ allocation de mémoire pour sauvegarde, locaux et
@ les arguments pour l'appel de a1 et a2

sub   sp, sp, #B_TAILLE_REGS+B_TAILLE_VARS

@ les positions des variables de b stockées dans la pile
B_DELTA_V0 = - (TAILLE_REGS+ TAILLE_VARS)
B_DELTA_V1 = B_DELTA_V0 + 4
B_DELTA_V2 = B_DELTA_V0 + 8
...                          @ etc jusqu'à 8

                             @ ici le sommet de pile est en sp1
@ corps de b

@ affectation : v2 = reçu3
ldr  r0, [fp, #B_DELTA_RECU3]
str  r0, [fp, #B_DELTA_V2]

@ ...

@ appel de a1 : 2 arguments en pile, le 1er dans r0
mov r0, #12                   @ empiler p2 = 12
str r0, [sp, #-4]!
mov r0, #11                   @ empiler p1 = 11
str r0, [sp, #-4]!
mov r0, #10                   @ p0 dans r0 = 10
bl  a1
add sp, sp, #8                @ libérer le bloc (p1,p2)



@ appel de a2 : 5 arguments en pile, le 1er dans r0

mov r0, #25                   @ empiler p5 = 25
str r0, [sp, #-4]!
mov r0, #24                   @ empiler p4 = 24
str r0, [sp, #-4]!
mov r0, #23                   @ empiler p3 = 23
str r0, [sp, #-4]!
ldr r0, [fp, #B_DELTA_RECU2]    @ empiler p2 = recu2+3
```

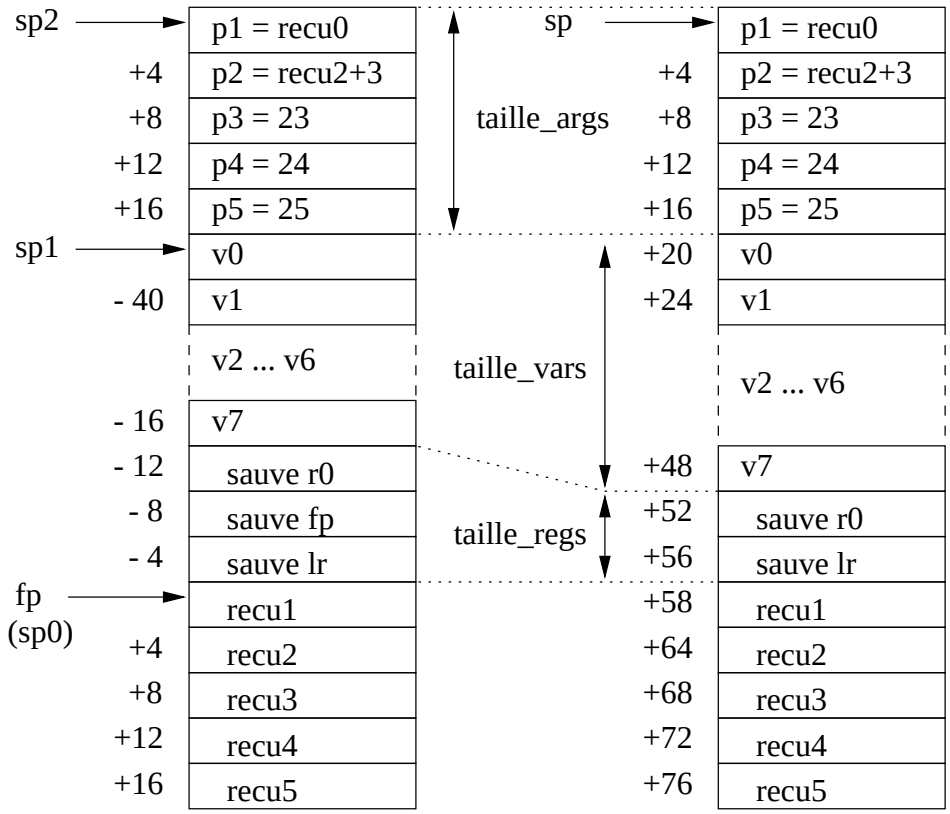


FIG. 11.4 – Extrait de la pile avec (à gauche) et sans (à droite) lien dynamique

```
add r0, r0, #3
str r0, [sp, #-4]!
ldr r0, [fp, #B_DELTA_RECU0]    @ empiler p1 = recu0
str r0, [sp, #-4]!
mov r0, #20                     @ p0 dans r0 = 20
                                @ ici le sommet de pile est en sp2
bl  a2
add sp, sp, #20                 @ libérer bloc (p1,p5)

@ epilogue de b                 @ libérer locaux
mov sp, fp
@ sp pointe juste au_dessus de la zone de sauvegarde
ldmdb sp, {r0,fp,lr}
@ rétablit les contenus de fp et lr avant l'appel
jmp   lr
```

11.2.5 Technique de codage sans lien dynamique

Une autre technique consiste à n'utiliser que le registre sp pour accéder à tous les éléments stockés dans la pile, auquel cas le codage est nettement facilité si la position du sommet de pile reste fixe dans tout le corps de la procédure.

Le principe consiste à préallouer dans le prologue de la mémoire pour le tableau d'arguments à passer. Si le corps de la procédure effectue plusieurs appels de procédure, ce tableau sera dimensionné pour l'appel qui passe le plus grand nombre d'arguments[6].

[6]au prix d'un léger gaspillage de place dans la pile, ce tableau étant évidement surdimensionné pour les autres

Le bloc d'arguments passés est alors accessible à l'adresse **sp**, celui des variables locales à l'adresse **sp** + **A** et celui des paramètres reçus à l'adresse **sp** + **A** + **L**.

```
        A2_NB_ARGS = 6           @ proceure A2 a 6 arguments

        B_NB_ARGS = 6                   @ procédure B à 6 arguments
        B_TAILLE_VARS = 8*4             @ 8 variables locales en pile
        B_TAILLE_REGS = 2*4             @ 2 registres sauvegardés
        B_TAILLE_ARGUMENTS=(A2_NB_ARGS-1)*4    @ param p1 à p5 de a2

        B_TAILLE_ALLOC=B_TAILLE_ARGUMENTS+B_TAILLE_VARS+B_TAILLE_REGS

        .text
b:      stmdb sp, {r0,lr}
        sub   sp, sp, #B_TAILLE_ALLOC

        @ emplacement de sauvegarde de r0 : debut de TAILLE_REGS
        B_DELTA_r0 = B_TAILLE_ARGUMENTS+B_TAILLE_VARS

        @ les positions des paramètres reçus

        B_DELTA_recu0 = B_DELTA_r0  @ reçu0 dans la sauvegarde de r0
                                    @ les autres dans la pile
        B_DELTA_recu1 = B_TAILLE_ALLOC + 0
        B_DELTA_recu2 = B_TAILLE_ALLOC + 4
        ...                         @ etc jusqu'à recu5 et +16

        @ les positions des variables de b stockées dans la pile
        B_DELTA_V0 = B_TAILLE_ARGUMENTS
        B_DELTA_V0 = B_TAILLE_ARGUMENTS+4
        ...                         @ etc jusqu'à v8 et +8

        @ corps de b

        @ affectation : v2 = reçu3
        ldr  r0, [sp, #B_DELTA_RECU3]
        str  r0, [sp, #B_DELTA_V2]

        @ ...
        @ appel de a1 : 2 arguments en pile, le 1er dans r0

        mov r0, #12                     @ passer p2 =12
        str r0, [sp, #4]
        mov r0, #11                     @ et p1 =11
        str r0, [sp, #0]
        mov r0, #10                     @ p0 dans r0 =10
        bl  a1

        @ appel de a2 : 5 arguments en pile, le 1er dans r0
```

appels

```
mov r0, #25                     @ passer p5 =25
str r0, [sp, #16]
mov r0, #25                     @ et p4 =24
str r0, [sp, #12]
mov r0, #25                     @ et p3 =23
str r0, [sp, #8]
ldr r0, [sp, #B_DELTA_RECU2]    @ et p2 = recu2+3
add r0, r0, #3
str r0, [sp, #4]
ldr r0, [sp, #B_DELTA_RECU0]    @ et p1 = recu0
str r0, [sp, #0]
mov r0, #20                     @ p0 dans r0
bl  a2

@ epilogue de b

add sp, sp, #B_TAILLE_ALLOC
@ sp pointe juste au_dessus de la zone de sauvegarde
ldmdb sp, {r0,lr}

jmp   lr
```

11.3 Taille de pile et débordement

La taille de la pile à prévoir dépend de la profondeur de l'arbre des appels, qui peut être élevée dans le cas de programmes récursifs. Il existe donc deux zones de données susceptibles de grossir au cours de l'exécution d'un programme : la pile, avec les appels de procédure, et le tas (généralement une extension de bss) au gré des allocations dynamique de mémoire.

Deux cas de figure peuvent alors se présenter :
- Le programmeur est capable de déterminer la borne supérieure de la consommation de mémoire de son algorithme et a dimensionné les sections bss et pile en conséquence
- Le programmeur ne connaît pas la consommation maximale de mémoire de son programme (par exemple parce qu'elle dépend des données à traiter) et il doit vérifier avant chaque allocation de mémoire (explicite ou lors d'un appel de procédure) qu'il reste assez de mémoire libre pour le faire.

En général, on choisit de faire croître le tas vers les adresses hautes et la pile vers les adresses basses. En l'absence de système d'exploitation, le programmeur devrait inclure des tests de débordement dans son code. Une technique de gestion simple consiste à placer la mémoire libre entre le tas et la pile : la mémoire libre est épuisée lorsque le sommet de la pile et l'extrémité du tas se rejoignent.

Sur les machines dotées d'un mécanisme de mémoire virtuelle, les débordements sont détectés automatiquement par le matériel, qui en informe le noyau du système d'exploitation et ce dernier interrompt l'exécution du programme fautif : le programmeur (ou le compilateur) peut se dispenser de tester les débordements de mémoire.

Chapitre 12

Procédures : cas particuliers

12.1 C ANSI et blocs à la PASCAL

Dans les langages à structure de blocs tels que PASCAL, il est possible de définir des procédures à l'intérieur d'autres procédures. Chaque procédure a alors automatiquement accès aux variables locales et aux procédures locales des procédures qui l'englobent.

Le langage C tel qu'il a été défini initialement par Kernighan et Ritchie a une structure "plate" : les procédures sont déclarées les unes à coté des autres et ne s'englobent pas. Le passage de paramètres de type pointeur via un appel (des appels en cascade) est l'unique méthode permettant à une procédure de modifier les variables locales d'une autre procédure.

La norme ANSI définissant le langage C a enrichi la définition initiale et le C ANSI est devenu un langage à stucture de bloc autorisant la déclaration de procédures à l'intérieur d'autres procédures. Nous ne détaillerons pas la traduction en langage d'asemblage des accès aux variables et procédures déclarés par les procédures englobantes.

12.2 Fonctions

Un fonction est une routine retournant un résultat utilisable dans une affectation. Outre la liste des paramètres et leur type, la déclaration d'une fonction précise le type de résultat qu'elle retourne. En C, les procédures (qui ne retournent pas de résultat) sont déclarées comme des fonctions particulières retournant void (absence de type : pas de résultat).

Une fonction remplace habituellement une procédure qui ne calcule qu'un seul résultat.

```
int b, a=3;

/* variante procédure */

void proc_fois2 (int x, int *res)
{
*res = x + x;
}

void essaip ()
{
proc_fois2 (a, &b);
}
```

```
/* variante fonction */

int fonc_fois2 (int x)
{
return (x + x);
}

void essaif ()
{
b = func_fois2 (a);
}
```

La fonction appelante et la fonction appelée doivent convenir d'un emplacement de stockage du résultat. Une convention répandue consiste à stocker le résultat en lieu et place du premier paramètre reçu. Dans notre exemple, le résultat d'une fonction sera stocké dans le premier registre **r0**.

```
            .data
a:          .word 3

            .bss
b:          .skip 4

            .text
            @ x dans r0, résultat dans r0
fonc_fois2: add  r0, r0, r0
            jmpl lr

essaif:     ...               @ prologue : sauvegarde regs
            mov32 r1, #a
            ldr   r0, [r1]
            bl    fonc_fois2
            mov32 r1, #b
            str   r0, [r1]
            ...               @ epilogue : restauration regs + retour
```

12.3 Pointeurs de fonctions

Comme tout objet C ayant une adresse mémoire, une fonction[1] peut être repérée par un pointeur. L'entité obtenue en appliquant l'opérateur * au contenu d'un pointeur de fonction est une fonction que l'on peut appeler.

```
int oppose (int x)
{
return (-x);
}

/* Définition du type intfint : */
/* fonction d'un entier retournant un entier */
typedef int intfint (int);

intfint *pointe_fonc =&oppose;       /* un pointeur de fonction */

void main ()
{
/* appel de oppose */
b = (*pointe_fonc) (3);          /* b = - 3 */

pointe_fonc = &fonc_fois2;

/* appel de fois2 */
```

[1] ou une procédure

```
b = (*pointe_fonc) (b);          /* b = - 6 */
}

              .data
pointe_fonc:  .word  oppose      @ intfint *pointe_fonc = &oppose

              .text
              ...
                                 @ extraits du code de main
                                 @ appel  b = (*pointe_fonc) (3)
              mov   r0, #3
              mov32 r1, #pointe_fonc
              ldr   r1, [r1]
              jmpl  r1
              mov   r1, #b
              str   r0, [r1]

                                 @ pointe_fonc = &fonc_fois2
              mov   r0, #fonc_fois2
              mov32 r1, #pointe_fonc
              str   r0, [r1]

                                 @ b = (*pointe_fonc) (b)
              mov   r0, #b
              ldr   r0, [r0]
              mov32 r1, #pointe_fonc
              ldr   r1, [r1]
              jmpl  r1
              mov   r1, #b
              str   r0, [r1]
              ...
```

12.4 Paramètre de type tableau

Rappelons qu'une déclaration de tableau définit simplement une constante pointeur du nom du tableau, repérant son premier élément. Ceci a deux conséquences :

- Un paramètre tableau est toujours passé par adresse. Le prototype de la fonction peut déclarer le paramètre comme un tableau sans préciser sa dimension, ou comme un pointeur : les deux notations sont synonymes.
- Si la procédure a besoin de connaître la taille du tableau, celle-ci doit être passée à part comme un deuxième paramètre.

```
/* calcul du maximum et du minimum des éléments d'un tableau */
#define TAILLE_TAB 4
long t [TAILLE_TAB] = { 5, 2, 0, 9};

long maxtab (unsigned long t[], long taille)
{
int i;
unsigned long max;
max = 0;
```

```
for (i=0; i< taille; i++)
  if (*(t+i) > max) max = t[i];     /* t[i] et *(t+i) synonymes */
return max
}

long mintab (unsigned long *t, long taille)
{
int i;
unsigned long min;
max = 0;
for (i=0; i< taille; i++)
  if (t[i] > min) min = *(t+i);
return min
}

void main ()
{
unsigned long l;
l = mintab(t,TAILLE_TAB);
l = maxtab(t,TAILLE_TAB);
}
```

12.5 Paramètre et résultat de type structure

Dans la définition initiale du C, une fonction ne pouvait accepter que des paramètres scalaires[2] : entier, nombre flottant ou pointeur.

Dans la version initiale du langage C, pour passer le contenu d'une structure à une procédure, le programmeur devait passer explicitement autant de paramètres que de membres dans la structure. Le C ANSI autorise les paramètres de stype structures, le traducteur devant se charger de le transformer en passage individuel de chacun des membres.

```
struct point { float x,y;};
struct droite {
   struct point orig;
   struct point dest;
              };
struct droite dd = {{1.0, 2.0}, {3.5, 6.0}};

/* calcul de longueur : norme C ANSI */
float longueur (struct droite d)
{
float l;
l = (d.dest.x-d.orig.x)*(d.dest.x-d.orig.x);
l += (d.dest.y-d.orig.y)*(d.dest.y-d.orig.y);
return sqrt(l);
}

/* calcul de longueur : */
```

[2]scalaire signifie ici stockable dans un mot ou un registre

```
/* conversion à l'ancienne norme C Kernighan & Ritchie */
float longueur_ancien (float xorig, float yorig, float xdest, float ydest)
{
float l;
l = (xdest-xorig)*(xdest-xorig);
l += (ydest-yorig)*(ydest-yorig);
return sqrt(l);
}

void main ()
{
float norme;
norme = longueur (dd);
norme = longueur_ancien (dd.orig.x, dd.orig.y,dd.dest.x, dd.dest.y);
}
```

Le même principe s'applique au fonctions devant retourner un objet de type structure : ces fonctions seront converties en procédures recevant un paramètre de type pointeur, soit explicitement par le programmeur (norme K&R), soit implicitement par le traducteur (C ANSI).

```
/* exemple de resultat de type structure, nouvelle norme */
struct droite decaler ()
{
struct droite res;
res.orig.x = dd.orig.x + 2.0;
res.orig.y = dd.orig.y + 3.0;
res.dest.x = dd.dest.x + 2.0;
res.dest.y = dd.dest.y + 3.0;
return(res);
}

/* idem , norme C K&R */
void decaler_ancien(struct droite *res)
{
*res.orig.x = dd.orig.x + 2.0;
*res.orig.y = dd.orig.y + 3.0;
*res.dest.x = dd.dest.x + 2.0;
*res.dest.y = dd.dest.y + 3.0;
}

void main()
{
dd = decaler ();      /* norme ANSI */
decaler_ancien(&dd);  /* conversion norme K&R */
}
```

12.6 Gestion des appels à nombre variable d'arguments

Certaines fonctions sont conçues pour accepter une liste d'arguments de taille quelconque. La longueur de la liste peut être définie par le contenu du premier argument, ou la liste peut être

délimitée par un marqueur de fin (typiquement l'entier 0 ou le pointeur NULL). A titre d'illustration de la première méthode, citons les fonctions printf et scanf. La primitive posix execl est un bon exemple d'utilisation de marqueur de fin de liste.

Dans le prototype de la fonction, on note **...** la liste de paramètres de longueur variable. Cette liste est précédée d'au moins un paramètre nommé explicitement.

Les fonctions à nombre de paramètres quelconques exploitent le fait que les arguments sont stockés sous forme de tableau dans la pile, bien que de nombreuses conventions d'appel stipulent que les arguments r à n sont empilés et que les arguments 0 à r-1 sont passés dans r registres du processeur.

Pour se ramener dans tous les cas à un cas unique simple de parcours de tableau de paramètres dans la pile, les fonctions à nombre quelconque de paramètres recopient les premiers paramètres des registres dans la pile avant d'effectuer la sauvegarde de l'adresse de retour et des autres registres modifiés dans le corps de la procédure. La figure 12.1 illustre cette recopie pour r = 4.

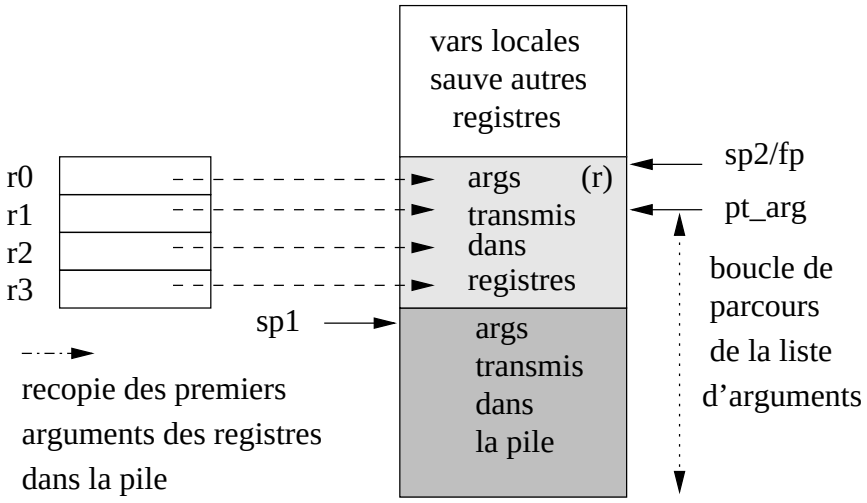


FIG. 12.1 – Fonctions varargs : boucle de parcours des arguments

Dans le corps d'une telle fonction, pour parcourir le tableau d'arguments de taille variable, on utilise une variable pointeur (nommée **pt_arg** dans l'exemple) de type **va_list**.

Pour initialiser le pointeur d'argument à l'adresse du dernier argument nommé explicitement (**r** dans l'exempe), on utilise la macro **va_arg(pt_arg,r)**. Cette macro génère un code que l'on pourrait écrire : **pt_arg = (va_list) &r ; pt_arg++** : pt_arg repère l'argument dans la pile qui suit immédiatement l'argument **r**.

Pour consulter l'argument courant pointé par **pt_arg**, l'affecter à une variable **v** de type **t** et passer à l'argument suivant dans la liste, on utilise la macro **va_arg** : **v=va_arg(pt_arg,t)**. L'expansion de cette macro correspond à **v= *(t *)pt_arg ; ((t *)pt_arg)++**.

La macro **va_end(pt_arg)** ne génère pas de code : elle permet d'indiquer au compilateur la portée de la déclaration du pointeur d'arguments **pt_arg**

```
#include <stdio.h>
#include <stdarg.h>
```

```
/* Calcul la somme d'une liste d'entiers terminée par 0 */
void somme (unsigned *r,...)
{
register int s, arg_lu; /* code equivalent aux macros va_xxx */
va_list pt_arg;      /* { void * pt_arg */
                     /* affecter à pt_arg l'adresse du premier argument */
                     /* de la liste, qui se trouve juste après r        */
va_start(pt_arg,r);  /* pt_arg = &r; *((int *) pt_arg)++ */
s=0;
do
  {
  arg_lu = va_arg(pt_arg,int);            /* arg_lu = * (int *) pt_arg; */
                                          /* ((int *) pt_arg)++         */
  s+= arg_lu;
  }
while (arg_lu != 0);
va_end(pt_arg);      /* } fin de la portée de la déclaration de pt_arg */
*r = s;
}

void main ()
{
int x;
somme (&x, 1,2,3,4,5,0);   /* liste de 5 arguments + argument initial */
printf ("somme=%d\n",x);
somme (&x, 4,5,0);         /* liste de 2 arguments + argument initial */
printf ("somme=%d\n",x);
}

@ Traduction de somme
@ Convention d'appel : quatre premiers arguments dans registres r0 à r3
@
@ Allocation des registres (arbitraire)
@ r0 : arg_lu
@ r1 : s
@ r2 : pt_arg
@ r3 : r

            .text
somme:      @ prologue particulier

            @ ici l'appelante a laissé le sommet de pile en sp1
            stmfd  sp!, {r0-r3}   @ recopie des 4 premiers args dans la pile

            @ ici le sommet de pile est en sp2 : repère tableau d'arguments
            stmfd  sp!, {fp,lr}  @ sauvegarde des autres registres

            @ fp repère les premiers arguements
            add    fp, sp, #8
```

```
            DELTA_R = 0
            DELTA_DEBUT_LISTE_ARGS=4

            @ corps de somme

            add   r2, fp, #DELTA_DEBUT_LISTE_ARGS    @ va_start(pt_arg,r)
            mov   r1, #0                             @ s = 0;

corps_do:   ldr   r0, [r2], #4                       @ va_arg(pt_arg,int)
            add   r1, r1, r0                         @ s += arg_lu

            cmp   r0, #0                             @ while (arg_lu != 0)
            bne   corps_do

            ldr   r3, [fp, #DELTA_R]                 @ *r = s
            str   r1, [r3]

            @ epilogue
            ldmfd  sp!, {fp, lr}                     @ restaurer registres
            add    sp, sp, #16                       @ libérer place des
                                                     @ 4 premiers args
            @ retour
            mov   pc, lr
```

12.7 Paramètres de main

Main est une fonction appelée par le code d'initialisation standard ajouté par défaut à tout programme C. Main retourne un code d'erreur que le processus qui a lancé l'exécution du programme peut récupérer. Un code de retour nul indique habituellement l'absence d'erreur.

La figure 12.2 illustre la pile lors de l'exécution d'un programme de calcul lancé avec la ligne de commande ci-dessous. La figure suppose que la convention d'appel spécifie que le premier paramètre est transmis dans r0.

```
mamachine> calcul 1234 56
mamachine>
```

Les paramètres de main lui permettent d'accéder aux arguments de lancement. Dans un environnement posix, ces arguments correspondent aux chaînes de caractères représentant les mots de la ligne de commande ("calcul", "1234" et "56") et aux définitions de variables d'environnement (du genre "PATH=/usr/.../bin", "TERM=xterm", "SHELL=/bin/tcsh").

Les trois paramètres standard de main sont :

1. **argc** : le nombre d'éléments du tableau argv,
2. **argv** : un tableau de argc pointeurs sur les chaînes de caractères représentant les arguments de la ligne de commande,
3. **envp** : un tableau de argc pointeurs sur les chaînes de caractères représentant les arguments de la ligne de commande, terminé par la constante NULL.

```
#include <stdio.h>
```

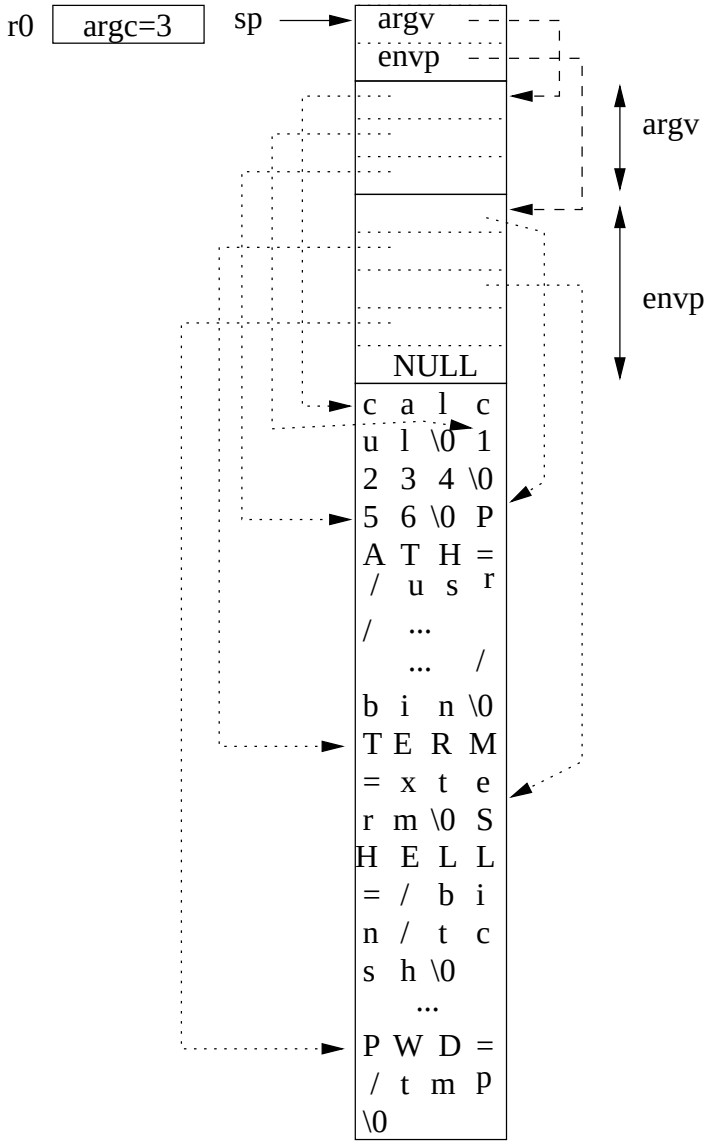


FIG. 12.2 – Les paramètres de main dans la pile

```
int main (int argc, char *argv[], char *envp[])
{
int i;
printf ("argc : %d\n",argc);
for (i=0;i<argc;i++)
  printf ("argv[i]  : %s\n", argv[i]);
i=0;
while (envp[i] != NULL)
  {
  printf ("envp[i] : %s\n", envp[i]);
  i++;
  }
return 0;
}

/* Ce programme affiche le texte suivant : */
/* argc : 3                                */
```

```
/* argv[0] : calcul                        */
/* argv[1] : 123                           */
/* argv[2] : 456                           */
/* envp[0] : PATH=/usr/...../bin           */
/* envp[1] : TERM=/xterm                   */
/* envp[2] : SHELL=/bin/tcsh               */
```

Chapitre 13

Compilation séparée et attributs de stockage

Un programme est habituellement décomposé en modules compilés séparément en fichiers objets relogeables, fusionnés en un fichier exécutable unique lors de l'édition de liens.

Certains langages offrent une gestion des modules élaborée, telle la gestion des "paquetages" dans le langage ADA. La notion des modules en C et en langage d'assemblage est moins élaborée et ce confond avec celle de fichier : chaque fichier est un module. Dans ce chapitre, les termes modules et fichiers sont synonymes.

La traduction séparée des fichiers pose deux problèmes dictincts :

1. lors de la compilation du C : la cohérence de type des objets définis (et exportés) dans un fichier et utilisés (importés) dans un autre
2. lors de l'assemblage ou de la compilation du C : la définition de la portée des symboles pour la phase d'édition de liens,

Un symbole partagé correspond typiquement à une variable commune utilisée dans plusieurs fichiers ou à une fonction définie dans un fichier et utilisée dans un autre.

13.1 Cohérence de type et compilation séparée en C

Un compilateur a besoin de connaître le type d'une variable (respectivement les types des paramètres et du résultat pour une fonction) pour traduire correctement les accès (respectivement les appels) à celle-ci : le type spécifie la taille et l'interprétation du contenu de la variable (respectivement des paramètres à passer et du résultat à retourner) à transférer.

13.1.1 Déclaration de type d'une variable

Une déclaration ordinaire de variable a un double rôle :

1. définir le type de la variable et
2. réserver de la mémoire pour la stocker.

Le partage entre plusieurs fichiers compilés séparéments nécessite deux types de déclarations :

1. une déclaration ordinaire avec allocation de mémoire dans le module qui définit (exporte) la variable et

2. une déclaration spéciale sans réservation de mémoire pour spécifier le type dans les modules qui accèdent à (importent) cette variable sans la définir.

Dans le langage C, une déclaration de variable du deuxième type est précédée de **extern** : cet attribut de stockage indique que la déclaration spécifie uniquement le type de la variable et qu'un autre module doit déjà réserver de la mémoire pour stocker cette variable.

13.1.2 Déclaration de (proto)type d'une fonction

Une déclaration ordinaire définit le type d'une fonction et alloue de la mémoire aux instructions qu'elle contient. La déclaration de compose de deux parties :

1. **le prototype**, qui spécifie le nom de la fonction et le type de résultat qu'elle retourne, ainsi que (délimités par une paire de parenthèses) la liste des types et noms des arguments.
2. le **corps** de la fonction, composé d'une ou plusieurs instructions (précédées éventuellement de déclarations de variables locales) encadrées par une paire d'accolades.

Le type de la fonction (autrement dit la manière de l'appeler) est entièrement défini par son prototype. Le corps de la fonction spécifie le contenu initial de la mémoire allouée à cette fonction. Il est traduit en une suite d'autant de valeurs initiales que de mots réservés, qui correspondent au codes des instructions machines (et éventuellement aux constantes) utilisées dans la fonction.

Une déclaration de fonction sans corps ne spécifie que le type et n'alloue pas de mémoire pour la fonction. Pour définir le (proto)type de la fonction, la liste des types des paramètres suffit et leur nom peut être omis.

L'attribut **extern** est optionnel : une déclaration sans corps ne peut être qu'une simple spécification du type de la fonction.

La seule déclaration du type d'une fonction est nécessaire dans deux contextes :

1. l'utilisation dans un module d'une fonction définie dans un autre fichier, comme pour les variables et,
2. la déclaration de fonctions mutuellement récursives.

13.1.3 Gestion de la cohérence

Lors de la compilation séparée de fichiers, il convient de détecter toute discordance de type entre la déclaration qui réserve la mémoire dans le module qui définit (exporte) une variable ou une fonction et la ou les déclarations de type dans les modules qui l'utilisent (importent).

La technique habituelle en C consiste à placer les déclarations de type dans un fichier suffixé **.h** et à l'inclure ce dernier dans tous les fichiers qui utilisent (y compris celui qui définit) les variables ou fonctions citées dans le fichier **.h**. Toute divergence entre la spécification de type incluse dans le fichier **.h** et la déclaration d'une variable ou d'une fonction dans le module qui la définit déclenchera une erreur lors de la compilation de ce dernier.

Un fichier **truc.h** inclus par la directive **#include<truc.h>** spécifie les types et prototypes associés aux bibliothèques standard livrées avec le système d'explitation et les chaînes de compilation. Il appartient à un des répertoires prédéfinis décrivant ces bibliothèques.

Un fichier **monprog.h** inclus par la directive **#include"monprog.h"** est stocké avec les fichiers **.c** du programme et décrit les variables et fonctions partagées du programme à compiler.

13.1.4 Exemple

Considérons à titre d'exemple le squelette de programme suivant, composé de trois fichiers :

1. un fichier prog.h définissant le type des variables et fonctions partagées,
2. un fichier prog2.c définissant x et f et utilisant y et calcul,
3. un fichier prog.c définissant y et calcul et utilisant x et f.

```
/*****************************************************************/
/* fichier prog.h                                                */
/*****************************************************************/

/* Définition du type des variables partagées */
extern long x;
extern long y;

/* Définition du prototype des fonctions partagées */
extern void calcul (long, long *);
extern long f (long);
```

On peut remarquer l'absence de nom des paramètres des fonctions dans le fichier de définition des prototypes.

```
/*****************************************************************/
/* fichier prog2.c                                               */
/*****************************************************************/

#include "prog.h"

/* Variable définie et exportée */
long x = 4;

/* Fonction définie et exportée */
long f(long t)
{
long r;
/* ... */
calcul (y+2,&r);   /* calcul et y sont définies dans prog.c */
/* ... */
return (r);
}
```

Le fichier prog.c illustre l'utilisation des prototypes pour prédéclarer le type de fonctions mutuellement récursives.

```
/*****************************************************************/
/* fichier prog.c                                                */
/*****************************************************************/
```

```
#include "prog.h"

/* Variable définie et exportée */
long y = 13;

/* Procédure définie et exportée */
void calcul (long x, long *r)
{
*r = x +3;
}

/* Ce prototype est nécessaire pour déclarer correctement */
/* les procédures mutuellement récursives locales         */

static void recursive1 (long);  /* Cette déclaration est indispensable */
static void recursive2 (long);  /* Cette déclaration est facultative   */

static void recursive2 (long b)
{
/* ... */
if (b>1) recursive1(b-1);
/* ... */
}

static void recursive1 (long a)
{
/* ... */
recursive2(a/2);
/* ... */
}

/* Main utilise f et x définies dans prog2.c */
int main ()
{
/* ... */
recursive1(x);
f(11);
/* ... */
return 0;
}
```

13.2 Exportation de symboles

La définition d'un symbole (variable ou fonction) dans un module correspond à la définition d'un nom symbole associé à l'emplacement mémoire alloué au stockage du contenu de la variable ou des instructions de la fonction.

Le langage utilisé doit permettre de spécifier si la définition d'un symbole dans un fichier est utilisable dans les autres fichiers (définition globale/exportée) ou si la portée de la définition est limitée au seul fichier qui la contient (définition à portée locale).

Notons que tout symbole non défini utilisé dans un module sera implicitement réputé importé d'un autre module. En C, en l'absence de spécification de type, le symbole sera supposé de type int (ou retourner un résultat de type int à partir de paramètres de type int pour une fonction).

13.2.1 Exportation en langage d'assemblage

En **langage d'assemblage**, une étiquette a **par défaut** une **portée limitée** au module qui la définit. La **directive d'exportation** d'une définition d'étiquette vers les autres fichiers compilés séparément est **.global étiquette**.

Si deux variables[1] définies dans deux modules différents ont le même nom, elles sont considérées comme deux variables distinctes. Tout se passe comme si l'on avait implicitement préfixé le nom de variable par le nom du fichier dans laquelle la variable est définie, pour donner deux noms de variables différents.

Si un symbole est utilisé dans un fichier sans être défini, il est supposé importé depuis un autre fichier (dans lequel il aura l'attribut global) et sa définition sera connue lors de la phase d'édition de liens.

13.2.2 Exportation en langage C

Les variables et les fonctions définies à l'intérieur d'une fonction ne sont accessibles que dans le corps de cette fonction.

Les variables et fonctions définies à l'extérieur de toute fonction sont **par défaut exportables** vers d'autres fichiers. Une variable ou une fonction locale à un fichier doit être définie avec l'attribut de stockage **static** pour en **cacher la définition** aux autres modules.

Le mécanisme d'exportation par défaut du langage C est cependant criticable. Supposons que deux programmeurs écrivant deux modules d'un même programme utilisent fortuitement le même nom de variable privée (éventuellement de types différents) dans leurs fichiers respectifs.

Si dans les deux fichiers la variable est déclarée avec l'attribut static, tout est correct : elles sont considérées comme deux variables distinctes. Si la variable est déclarée dans les deux fichiers sans l'attribut **static**, l'édition de liens signalera une erreur de double définition.

En revanche, si l'un des programmeurs ne spécifie pas l'attribut **static** et que l'autre oublie de déclarer sa variable, aucune erreur ne sera générée et les deux modules utiliseront une variable partagée (du type spécifiée par le permier module) au lieu de deux variables indépendantes.

La traduction en langage d'assemblage d'une déclaration C génère donc :

1. une réservation de mémoire
2. une définition de symbole étiquette et
3. une directive **.global** d'exportation du symbole, omise en présence de l'attribut **static**.

Notons que dans les chapitres précédents, aucune des déclarations C ne spécifiait l'attribut **static**. Cependant les directives **.global** correspondant à ces déclarations ont été omises dans la traduction en langage d'assemblage pour ne pas perturber le lecteur.

[1] même chose pour des fonctions

13.2.3 Exemple

Voici à titre d'exemple deux des modules (export1.c et export2.c) d'un programme C (qui en comporte d'autres). Ces deux modules partagent une variable (variable_partagee) et deux fonctions (f1_partagee et f2_partagee). En revanche, chacun déclare une variable (var_locale) et deux fonctions de même nom (f_locale) et deux fonctions de noms différents (essai et essai2), privées toutes les cinq.

```
/*****************************************************************/
/*  fichier export.h                                             */
/*****************************************************************/

extern long var_partagee;

extern long f1_partagee (long);
extern long f2_partagee (long);

/*****************************************************************/
/*  fichier export1.c                                            */
/*****************************************************************/

#include "export.h"

long var_partagee = 1234;

long f1_partagee (long a)
{
return (a & 1);
}

static long  var_locale = 3;

static int f_locale (int x)
{
return (3*x);
}

static void essai ()
{
/* ... */
var_locale = f1_partagee (4);
var_partagee = f2_partagee (3);
var_locale = f_locale (2);
/* ... */
}

/*****************************************************************/
/*  fichier export2.c                                            */
/*****************************************************************/

#include "export.h"
```

```
long f2_partagee (long a)
{
return (a/2);
}

static long  var_locale = 5;

static int f_locale (int x)
{
return (x+2);
}

static void essai2()
{
/* ... */
var_locale=f_locale(8);
var_partagee=f1_partagee(6);
/* ... */
}
```

Voici à quoi ressemble sa traduction en langage d'assemblage :

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ fichier export1.s
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

              .data
              .global  var_partagee

var_partagee: .word    1234
var_locale:   .word    3

              .text
              .global f1_partagee

f1_partagee:  and      r0, r0, #1
              jmp      lr

f_locale:     add      r0, r0, r0, LSL #1
              jmp      lr

essai:        @ sauvegarde et restauration
              @ des registres modifies omises
              @ ...
              mov      r0, #4
              bl       f1_partagee
              mov32    r1, #var_locale
              str      r0,[r1]
              mov      r0, #3
              bl       f2_partagee
              mov32    r2, #var_partagee
```

```
                str     r0,[r2]
                mov     r0, #2
                bl      f_locale
                mov32   r1, #var_locale
                str     r0,[r1]
                @ ...
                jmpl    lr

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ fichier export2.s
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

                .data

var_locale:     .word   5

                .text
                .global f2_partagee

f2_partagee:    mov     r0, r0, ASR #1
                jmp     lr

f_locale:       add     r0, r0, #2
                jmp     lr

essai2:         @ sauvegarde et restauration
                @ des registres modifies omises
                @ ...
                mov     r0, #8
                bl      f_locale
                mov32   r2, #var_locale
                str     r0,[r2]
                mov     r0, #6
                bl      f1_partagee
                mov32   r1, #var_partagee
                str     r0,[r1]
                @ ...
                jmpl    lr
```

13.3 Attributs de stockage

La déclaration d'une variable peut être précédée d'un attribut ou d'un qualificateur de stockage : **auto**, **static**, **extern**, **register**, **const** ou **volatile**.

13.3.1 Classes de stockage

Il n'existe que deux classes de stockage d'une variable C :

1. statique : la mémoire de stockage de la variable est allouée statiquement (sections data ou bss) et le contenu de la variable est accessible pendant toute la durée du programme,

2. automatique : la mémoire de stockage est allouée dynamiquement (dans la pile[2]) au début et libérée à la fin de l'exécution du bloc d'instructions qui déclare la variable (après quoi son contenu est détruit).

La classe de stockage d'une variable définie dans une fonction (ou dans un bloc d'instructions inclus dans le corps d'une fonction) est implicitement automatique. D'autre part, la classe de stockage d'une variable déclarée à l'extérieur de toute fonction est toujours statique.

La classe automatique étant implicite dans tous les cas de figure où elle peut être utilisée, l'attribut de stockage **auto** ne sert en fait à rien et peut être systématiquement omis.

L'attribut **register**, quelque peu tombé en désuétude, spécifie également une classe de stockage automatique, mais indique au compilateur de placer si possible la variable dans un registre. L'opérateur "adresse de" (**&**) n'est pas applicable à une variable **register**.

13.3.2 Attribut de stockage static

L'attribut de stockage **static** a deux significations totalement différentes selon que la variable est déclarée à l'intérieur d'une fonction ou à l'extérieur de toute fonction. Cette double sémantique ne constitue pas le choix de conception du langage C le plus heureux.

Comme exposé précédemment, à l'extérieur de toute fonction, il masque la définition de la variable aux autres modules composant le programme.

Une variable déclarée avec l'attribut **static** à l'intérieur d'une fonction a un statut hybride :
- Sa classe de stockage est statique et elle conserve son contenu entre deux appels de la fonction comme si elle était déclarée statiquement à l'extérieur de toute fonction.
- Son nom n'est connu et elle n'est accessible qu'à l'intérieur du corps de la fonction : elle reste bien une variable privée de la fonction contrairement à une variable qui serait déclarée à l'extérieur de celle-ci.

Ce genre de déclaration est par exemple utile pour définir une fonction capable de compter combien de fois elle a déjà été exécutée.

```
/*****************************************************************/
/* gestion de tickets de file d'attente                          */
/*****************************************************************/

static unsigned long donner_un_ticket ()
{
static unsigned long numero_courant = 1;
return numero_courant++;
}

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                      .data
numero_courant_de_donner_un_ticket:         .word 1
```

[2]un compilateur optimisant peut aussi décider de placer la variable dans un registre, sans changer pour autant la durée du stockage

```
                    .text
donner_un_ticket:   stmfd   sp!, {...,lr}
                    mov32   r1, numero_courant_de_donner_un_ticket
                    ldr     r2, [r1]
                    mov     r0, r1
                    add     r2, r2, #1
                    str     r2, [r1]
                    ldmfd   sp,! {....,lr}
                    jmp     lr
```

13.3.3 Qualificateurs de stockage const et volatile

Outre un attribut de stockage, une déclaration de variable peut inclure un qualificateur de stockage : **const** ou **volatile**.

L'attribut **const** indique que la variable contient une constante : modifier son contenu est illégal. Le compilateur peut alors vérifier l'absence d'affectation directe d'une valeur à cette variable[3].

L'attribut **volatile** indique au compilateur que le contenu de la variable peut être modifié en l'absence de tout accès dans le code qu'il génère. Ce cas de figure se rencontre en particulier en cas de partage d'une variable entre plusieurs processus ou lorsqu'une addresse correspond à un dispositif d'entrée/sortie.

Le compilateur n'a pas le droit d'optimiser l'accès à une variable **volatile** stockée en mémoire en supposant que son contenu vient d'être lu et stocké dans un registre par les instructions précédentes.

```
static long nonvol=0;
static volatile long vol=0;

if (nonvol != 0) nonvol --;
if (vol != 0) vol --;
```

```
            .data
nonvol:     .word  0
vol:        .word  0

            .text
            mov32  r1, #nonvol
            ldr    r0, [r1]
            cmp    r0, #0
            beq    finnonvol
            @ r0 contient encore la valeur actuelle de nonvol
            @ on peut donc omettre l'instruction ldr ci-dessous
            @ ldr r0,[r1]
            sub   r0, r0, #1
            str   r0, [r1]
finonvol:   mov32  r1, #vol
```

[3]L'attribut **const** n'est pas une assurance "tous risques" : une modification de la variable via un pointeur à l'insu du compilateur reste généralement possible.

```
            ldr    r0, [r1]
            cmp    r0, #0
            beq    finvol
            @ la valeur de vol lue dans r0 peut etre obsolete
            @ on ne peut donc pas omettre l'instruction ldr ci-dessous
            ldr   r0,[r1]
            sub   r0, r0, #1
            str   r0, [r1]
finvol:
```

Chapitre 14

Spécificités du jeu d'instructions ARM

14.1 Registres, compteur ordinal et instructions de calcul

14.1.1 Registres

Un processeur ARM est doté de 16 registres généraux. Contrairement à la majorité des autres processeurs, le **compteur ordinal** d'un processeur ARM est accessible en tant que **registre général : r15** (en langage d'assemblage on peut écrire au choix **pc** ou **r15**).

Lorsque pc/r15 est utilisé comme opérande, son contenu est l'adresse de l'instruction courante qui l'utilise plus huit. Pour optimiser les performances, les processeurs sont conçus de manière à exécuter une nouvelle instruction par cycle (techique de pipeline). Avec cette technique, le +8 s'explique par le fait que lorsque le compteur ordinal est lu par l'instruction courante, le processeur est déjà en train de lire la deuxième instruction qui suit.

La convention de nommage des registres est la suivante : **lr** est synonyme de **r14**, **sp** de **r13**, **ip** de **r12** et **fp** de **r11**.

Les quatre octets du registre d'état, appelé **cpsr**[1], sont notés c,x,s,f, le dernier étant celui de poids forts contenant les indicateurs NZCV. L'instruction spéciale **mrs** copie le contenu du registre d'état dans un registre général. L'instruction effectuant le transfert inverse est **mrs**, et il est possible de ne modifier que certains octets du registre d'état.

```
@ transfert du registre d'état  dans r2
     mrs  r2, cpsr
@ transfert inverse, affectant les quatre octets du registre d'état
     msr  cpsr_fsxc, r2
```

14.1.2 Constantes entières sur 32 bits

Toutes les instructions ARM sont sur un seul mot de 32 bits, sans exception. Pour charger une constante 32 bits quelconque dans un registre, on utilise une instruction ldr avec une adresse relative au compteur ordinal, qui permet d'accéder à tout mot (de la section text) dans le voisinage de l'instruction ldr (l'entier ajouté à pc restant codable sur 12 bits).

L'assembleur fournit une pseudo instruction de la forme **ldr reg, etiquette** avec **etiquette** dans le voisinage de l'instruction.

[1] Current Program Status Register

```
@ int ajout1 (int x)          int ajout2 (int x)
@   {                           {
@   return (x+0x11111111);      return (x+0x22222222);
@   }                           }
@
         .text
ajout:   ldr r1, [pc, #(cte1-ajout-8)]   @ r1 <- 11111111
         add r0, r0, r12
         mov pc, lr
@
@  Utilisation de la pseudo-instruction ldr reg, étiquette
ici:     ldr r2, cte2          @ génère ldr r2, [pc, #(cte2-ici-8)]
         add r0, r0, r12
         mov pc, lr

cte1:    .word   0x11111111    @ ces .word sont dans le voisinage
cte2:    .word   0x22222222    @ de ajout et ici
```

Pour faciliter encore plus le travail, l'utilisation de la directive **.ltorg** et de la pseudo-instruction **ldr reg,=cte** évite de déclarer les mots contenants les constantes à charger dans les registres. Ainsi, la traduction ARM de l'instruction **mov32 r0,#12345678** s'écrit **ldr r0,= 0x12345678**, suivi de (une seule fois et un peu plus loin dans la section text) **.ltorg**.

```
@ charger 11112222 dans r1
@ charger 33334444 dans r2
@ charger 55556666 dans r3
@
@  avec pseudo ldr =          @ code expansé par l'assembleur
       ldr  r1, =0x11112222   @ ici1:  ldr r1, [pc, #(c1 -ici1 -8)]
       ldr  r2, =0x33334444   @ ici2:  ldr r2, [pc, #(c2 -ici2 -8)]
       ldr  r3, =0x33334444   @ ici3:  ldr r3, [pc, #(c3 -ici3 -8)]
       ...                                ...
@ stocker les constantes ici
       .ltorg                 @ c1:    .word   0x11112222
                              @ c2:    .word   0x33334444
                              @ c3:    .word   0x44445555
```

14.2 Instructions de calcul

Le tableau 14.1 résume les instructions ARM de calcul.

Les deux variantes d'addition **add** et **adc** utilisent respectivement 0 et l'indicateur C comme retenue initiale. Le même principe d'applique aux instructions de soustraction (**sub**, **sbc**) qui retranchent l'opérande droit de l'opérande gauche. Il est également possible de soustraire l'opérande gauche de l'opérande droit (**rsb**, **rsc**[2]).

Le jeu d'instruction comprend une instruction pour chaque opération bit à bit (**and**, **or**, **eor**), ainsi que andnot (**bic**[3] : $op_{gauche} \& \tilde{} op_{droit}$).

[2]Reverse SuBstract, Reverse Substract with Carry
[3]BIt Clear

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`

TAB. 14.1 – Instructions de calcul : **nom rd, rn, op_droit**

L'instruction **mov** de copie d'un opérande (de type opérande droit) dans un registre destination admet une variante **mvn**[4] qui complémente l'opérande.

Toutes ces instructions (par exemple de soustraction) déposent le résultat de l'opération dans un registre destination et existent en deux versions (**subS**, **sub**) qui mettent à jour ou non les indicateurs **ZCNV**.

Les instructions **tst**, **teq** et **cmp**[5] sont des variantes de **andS**, **eorS** et **subS** qui mettent à jour les indicateurs mais ne déposent pas le résultat apparent dans un registre (ce qui évite de détruire un contenu utile d'un registre pour faire de simples comparaisons). L'instruction **cmn**[6] compare avec le complément de l'opérande droit.

14.2.1 Opérande droit, décalages et rotations

L'opérande droit d'une instruction de calcul peut être un registre général ou un opérande immédiat, à savoir un entier naturel codable sur 8 bits à une rotation à droite d'un nombre pair de bits près.

Il n'y a pas d'instruction ARM spécifique de décalage et de rotation, mais il est possible de les appliquer à l'opérande droit de toute instruction de calcul, s'il est de type registre (le contenu du registre n'est pas modifié : le décalage est appliqué sur la copie du contenu envoyée à l'unité de calcul)

```
@     Quelques exemples d'opérandes droits dans les calculs
@
      mov   r3, #255            @ 255 : 0xff
      add   r3, r3, #520        @ 520 : 0x210, rotation de 0x21
```

[4] MoV Not
[5] TeST, Test EQuivalence, CoMPare
[6] CoMpare Not

```
add   r2, r1, r1, LSL #3  @ r2 = 9 * r1
mov   r1, r2, ASR r3      @ r1 = r2 >> r3  (r3 int)
mov   r1, r2, LSR r3      @ r1 = r2 >> r3  (r3 unsigned int)
and   r1, r1, r2, ROR #4  @ rotation à droite 4 bits
orr   r1, r1, r2, ROR r3  @ rotation à droite de r3 bits
mov   r1, r2, RRX         @ rotation 1 bit à droite de (r2,C)
```

Sont disponibles le décalage arithmétique à droite (**ASR**), la rotation à droite (**ROR** et le décalage logique à gauche ou à droite **LSL, LSR**[7]) de **b** bits. Le nombre de bits **b** doit appartenir à l'intervalle [1, 32][8]. Il peut s'agir d'une constante entière codée sur 5 bits ou du contenu d'un troisième registre.

L'absence de modification de l'opérande droit est traité comme un décalage de zéro bits. Une rotation à droite de zéro bit est interprétée comme un rotation à droite d'un bit appliquée à un entier de 33 bits obtenu en concaténant l'opérande droit et l'indicateur C (rotation notée RRX).

14.3 Branchements et conditions

Les branchements relatifs utilisent un déplacement signé sur 24 bits exprimé en nombre d'instructions et qui tient compte du fait que PC pointe deux instructions en avance. Soit *dest* l'adresse de l'instruction destination du branchement et *source* l'adresse de l'instruction de branchement relatif, l'expression du déplacement est $\frac{dest-source-8}{4}$. En pratique, le programmeur utilise des étiquettes et l'assembleur se charge de calculer la valeur du déplacement.

En pratique, toutes les instructions ARM ordinaires sont conditionnelles : si la condition est fausse, l'instruction n'a pas d'autre effet que de faire passer le compteur ordinal à l'instruction suivante. Si la condition est vraie, l'instruction est exécutée normalement. En l'absence de suffixe de condition dans le mnémonique, la condition toujours vraie est implicitement utilisée (sub est synonyme de subal). Cette facilité permet de traduire de petites séquences conditionnelles sans branchement.

```
@ traduction de si (r0==r1) r0 = r0*2; else r1 = r1 + 1;
@
@ sans branchement                      avec branchement
        cmp  r0, r1                     cmp   r0, r1
                                        bne   sinon
alors:  addeq  r0, r0, r0       alors:  add   r0, r0, r0
                                        bal   finsi
sinon:  addne  r1, r1, #1       sinon:  add   r1, r1, #1
finsi:                          finsi:
@
@  Equivalent ARM de l'instruction jmp  r1+r2  du RISC fictif :
@       add    pc, r1, r2
```

Notons que toute addition ou soustraction stockant son résultat dans pc/r15 est aussi un branchement relatif, mais dont le déplacement est exprimé en octets et sur 8 bits seulement.

L'instruction de branchement abolu jmp de notre processeur RISC fictif correspond à une instruction ARM add ou mov affectant pc.

[7] Arithmetic Shift Right, ROtate Right, Logic Shift Left/Right, ROtate with eXtension

[8] 32 codé comme 0, intervalle [0, 31] pour LSL et [1,31] pour ROR

```
@ RISC de référence              @ instruction ARM équivalente
jmp  1r                            mov  pc, 1r
jmp  r1+r2                         add  pc, r1, r2
jmpl r1                            mov  lr,pc    @ sauver adresse retour
                                   mov pc, r1    @ branchement
```

14.4 Variantes des instructions load et store

Les instructions ARM ldm et stm se comportent comme celles du processeur RISC fictif avec une limitation : si le registre pointeur est mis à jour (variante ldm/stm reg !,liste-regs), il ne doit pas faire partie de la liste de registres à transférer (**stmfd sp !, {fp,sp}** n'est pas autorisé).

Il existe quelques différences mineures entre les instructions ldr et str du jeu d'instructions ARM et celui de notre processeur RISC fictif.

Les seules instructions d'accès à la mémoire dans la première définition du jeu d'instructions ARM permettaient uniquement :

1. la lecture (ldr) dans un registre et l'écriture (str) en mémoire d'un mot de 32 bits,
2. l'écriture en mémoire d'un octet, [9] (strb),
3. la lecture en mémoire d'un octet considéré comme un entier naturel (ldrb), avec extension de format à 32 bits par remplissage de 0 des 24 bits de poids forts du registre destinataire.

Le jeu d'instructions ARM a ensuite étendu pour inclure :

1. la lecture d'un octet considéré comme un entier relatif (ldrsb) avec extension de format à 32 bits par remplissage des 24 bits de poids fort du registre destinataire avec le bit de signe (bit 7) de l'octet,
2. l'écriture d'un demi-mot de 16 bits (strh),
3. la lecture d'un demi-mot de 16 bits avec extension à 32 bits appropriée, selon sa nature : entier naturel (ldrh) ou entier relatif (ldrsh).

Ce deuxième groupe d'instruction offre des possibilités d'adressages restreintes : il n'est pas possible d'appliquer un opérateur de décalage (LSL, LSR, ASR ou ROR[10], noté déc dans le tableau :) sur le deuxième registre et la constante entière codable est plus petite.

Mode d'adressage	adresse utilisée	nouvelle valeur reg1	Limitations sur ldrsb, ldrh, lrsh, sth
[reg1, ± reg2]	reg1 ± reg2	reg1	aucune
[reg1, ± reg2] !	reg1 ± reg2	reg1 ± reg2	
[reg1], ± reg2	reg1	reg1 ± reg2	
[reg1, # ±entier]	reg1 ± entier	reg1	Codage de l'entier sur 8 bits au lieu de 12
[reg1, # ± entier] !	reg1 ± entier	reg1 ± entier	
[reg1], # ± entier	reg1	reg1 ± entier	
[reg1, ± reg2, déc #n]	reg1 ± reg2 $\overset{n}{\leftrightarrow}$	reg1	Indisponible
[reg1, ± reg2, déc #n] !	reg1 ± reg2 $\overset{n}{\leftrightarrow}$	reg1 ± reg2 $\overset{n}{\leftrightarrow}$	
[reg1], ± reg2, déc #n	reg1	reg1 ± reg2 $\overset{n}{\leftrightarrow}$	

[9] pris dans l'octet de poids faible d'un registre

[10] En pratique, seul le décalage à gauche présente un intérêt dans un calcul d'adresse

Voici quelques exemples d'utilisation de ces modes d'adressage :

```
ldrh   r0, [r1, r2]
ldrsb  r0, [r1, -r2]!
ldrsh  r0, [r1], -r2
ldr    r0, [r1, #-4095]
ldrh   r0, [r1, #255]!
ldr    r0, [r1], #4095
ldr    r0, [r1, r2, LSL #3]!
```

14.5 Convention d'appel et pile du compilateur ARM GNU

La convention d'appel utilisée par le compilateur gcc est la suivante :

1. Les quatre premiers paramètres explicites sont stockés dans les registres r0 à r3.
2. les paramètres explicites suivant sont empilés et sp repère le premier d'entre eux.
3. L'adresse de retour est déposée dans le registre **lr**.
4. La sauvegarde des registres est à la charge de l'appelée, excepté pour ip.
5. Le résultat d'une fonction est retourné dans le registres r0.

Le prologue standard d'une fonction empile après les arguments transmis par la procédure appelante :

1. l'adresse de l'instruction d'allocation mémoire dans le prologue (cette information n'est pas nécessaire à la gestion des appels, mais facilite la mise au point des programmes avec un débogueur),
2. l'adresse de retour dans l'appelante,
3. le sommet de pile **sp** laissé par l'appelante,
4. le pointeur de paramètres **fp** de l'appelante.

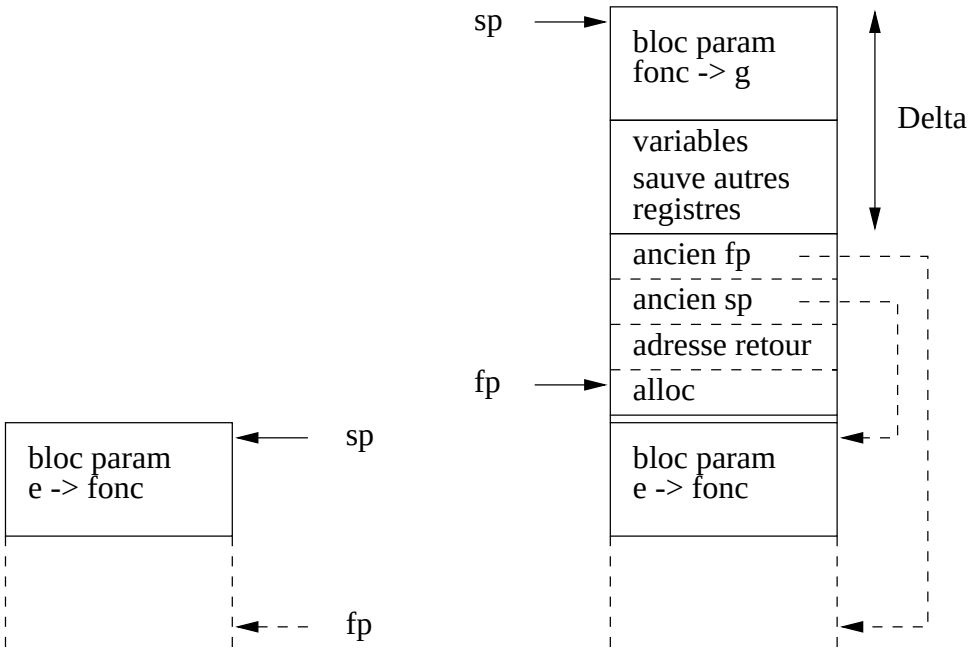


FIG. 14.1 – Etat de la pile ARM au début (à gauche) et à la fin (à droite) du prologue

Le pointeur de paramètres repère le premier item de ce bloc de quatre sauvegardes de registres.

Le squelette typique d'une procédure à nombre fixe d'arguments est le suivant :

```
fonc:          @ prologue
               mov     ip, sp
               stmfd   sp!, {fp,ip, lr, pc}
               sub     fp, ip, #4
alloc:         sub     sp, sp, #AUTRES_REGS+LOCAL+PARAM_PASSES

               @ corps de la procédure
               ...

               @ épilogue
               ldmea   fp, {fp, sp, pc}
```

14.6 Divers

La directive d'alignement **align x** aligne sur un multiple de 2^n au lieu d'un multiple de n, qui est obtenu par la directive **.balign x**.

14.7 Exemple de code ARM

Considérons à titre d'exemple la séquence C suivante :

```
short int s = 15;
static long int x = 1234;

long int proc (long l)
{
s = s + 3;
...
return (5*l);
}
```

En voici la traduction en langage d'assemblage ARM :

```
               .data
               .global s
s:             .short  15
               .balign 4    @ ou .align 2
x:             .word   1234

               .text
               .global proc
               DELTA_PROC=8      @ sauvagarde : r4 et r5

proc:          mov     ip, sp
               stmfd   sp!, {fp,ip, lr, pc}
               sub     fp, ip, #4
```

```
sub     sp, sp, #DELTA_PROC
str     r4, [sp, #(DELTA_PROC-4)]
str     r5, [sp, #(DELTA_PROC-8)]

ldr     r4,= s                      @ mov32 r4, #s
ldrsh   r5, [r4]
add     r5, r5, #3
strh    r5, [r4]
...

add     r0, r0, r0, LSL #2

ldr     r5, [sp, #(DELTA_PROC-8)]
ldr     r4, [sp, #(DELTA_PROC-4)]
                        @ retour + restaurer fp,sp
ldmea   fp, {fp, sp, pc}

.ltorg
```

Table des matières

Liste des tableaux

Table des figures