Architecture des ordinateurs

TD 5 : Assembleur SPARC

Arnaud Giersch, Benoît Meister et Frédéric Vivien

1 L'architecture SPARC

1.1 Caractéristiques générales

SPARC est une architecture du type RISC (*Reduced Instruction Set Computer*). Ce type d'architecture se caractérise par un jeu réduit d'instructions câblées qui opèrent sur un ensemble de registres.

Basiquement SPARC fait intervenir deux unités de calcul distinctes. La première est l'unité « entière » (*Integer Unit* notée IU) qui est consacrée au traitement des nombres entiers et des booléens. La seconde est l'unité « flottante » (*Floating Point Unit* notée FPU) qui est dédiée au traitement des nombres flottants. Les seules instructions arithmétiques qui seront présentées par la suite sont celles qui sont traitées par l'unité entière.

L'architecture SPARC est une architecture 32 bits [1] au niveau de son bus d'adresse ainsi que de son bus de données. Son espace d'adressage est linéaire et paginé, c'est-à-dire que 2^{32} octets sont adressables individuellement. Les mots mémoire sont de 32 bits et doivent être alignés sur des limites de mots (c.-à-d. que les adresses doivent être des multiples de 4). Enfin, la mémoire est de type *big-endian*, donc l'octet de poids faible d'un mot est à l'adresse la plus haute du mot.

SPARC est une architecture du type *chargement/rangement* (*load/store*) — aussi appelée architecture de type *registre-registre* — ce qui signifie que tous les calculs s'effectuent entre des registres et que les accès à la mémoire sont explicites et sont réalisés par des instructions dédiées.

> Les instructions opératoires (arithmétiques, logiques et de décalages) n'admettent que des registres — ou des constantes signées codées sur 13 bits — pour opérandes, mais **en aucun cas** une référence à la mémoire.

> Les seules instructions qui ont accès à la mémoire sont les instructions de chargement d'une zone référencée de la mémoire dans un registre et les instructions de rangement du contenu d'un registre dans une zone référencée de la mémoire.

Toutes les instructions et tous les registres ont une longueur de 32 bits. Il est néanmoins possible d'accéder en mémoire à des mots de 8, 16, 32 ou 64 bits, ce dernier type d'accès utilisant deux registres consécutifs.

SPARC est un architecture de type *pipeline* imposant des branchements et des appels de procédures retardés[2]. Une instruction qui suit immédiatement une instruction retardée au sein du code est appelée *instruction de délai* (*delay slot*). La conséquence de l'existence de délais implique que l'instruction qui suit un branchement ou un appel de routine (une procédure ou une fonction au sens du langage de programmation PASCAL) est toujours exécutée. Par conséquent, si le branchement est conditionnel, il ne faut pas que l'instruction de délai modifie le registre contenant les codes de condition. C'est la raison pour laquelle la plupart des instructions existent en deux versions, une avec modification d'état, l'autre sans.

[1] Les nouvelles versions comme SPARC v9 sont des architectures 64 bits.

[2] Certaines architectures (MIPS par exemple) imposent également des accès retardés à la mémoire.

1.2 Les registres de l'unité entière

SPARC dispose d'un mécanisme de « fenêtrage recouvrant des registres » utilisé pour améliorer le passage des paramètres lors de l'appel d'une routine. Dans une architecture CISC[3] standard, les paramètres sont passés par le biais de la pile, ce qui génère de très nombreux accès à la mémoire. Le nombre de registres étant conséquent au sein des architectures RISC, ceux-ci sont utilisés pour passer les paramètres aux routines lors de leur appel. Le mécanisme de fenêtrage recouvrant limite la visibilité des registres : SPARC limite l'accès à 32 registres pour une routine donnée. Un registre interne de 5 bits, le « pointeur de fenêtre courante » (*Current Window Pointer* désigné par CWP) référence la fenêtre de registres courante.

Les 32 registres de la fenêtre courante (c.-à-d. la fenêtre de registres associée à la routine courante) sont divisés en 4 groupes :

- 8 *registres globaux* (*global registers*) désignés par `%g0`, …, `%g7` et susceptibles de contenir des pointeurs et des variables globales, utilisés pendant toute l'exécution du programme (l'ensemble des routines du code ont accès à ces huit registres, c'est-à-dire que ces huit registres sont communs à l'ensemble des routines du programme) ;
- 8 *registres d'entrée* (*input registers*) désignés par `%i0`, …, `%i7` et susceptibles de contenir les paramètres entrants de la routine à laquelle est associée la fenêtre (c.-à-d. les paramètres reçus par la routine à laquelle est associée la fenêtre) ;
- 8 *registres locaux* (*local registers*) désignés par `%l0`, …, `%l7` et susceptibles de contenir les variables locales à la routine à laquelle est associée la fenêtre ;
- 8 *registres de sortie* (*output registers*) désignés par `%o0`, …, `%o7` et susceptibles de contenir les paramètres sortants de la routine à laquelle est associée la fenêtre (c.-à-d. les paramètres passés à une routine appelée par la routine à laquelle est associée la fenêtre).

Le mécanisme de fenêtrage recouvrant des registres procède du fait que les registres d'entrée associés à une routine appelée correspondent aux registres de sortie de la routine appelante. Ce mécanisme permet de passer les paramètres entre deux routines : l'appelante passe le premier paramètre dans son registre de sortie désigné par `%o0`, le second dans `%o1`, … S'il y a trop de paramètres, ceux qui ne peuvent être placés dans les registres de sortie sont placés sur la pile. Du fait du recouvrement, l'appelée trouvera son premier paramètre dans son registre d'entrée `%i0`, le second dans `%i1`, … et le reste dans la pile si les paramètres sont trop nombreux. La figure 1 page 3 illustre un appel de routine avec mécanisme de fenêtrage recouvrant. Notez sur cette figure que le contenu du registre CWP est décrémenté à chaque nouvel appel d'une routine.

Certains de ces registres ont des caractéristiques particulières.

- Le registre `%g0` est câblé à la valeur 0 si bien que le spécifier comme registre de destination dans une instruction ne le modifiera pas.
- Les registres `%g1` à `%g7` sont globaux, c'est-à-dire communs à toutes les fenêtres.
- Le registre `%o6` est le « pointeur de pile » (*stack pointer* désigné par `%sp`). Lors d'un appel de routine, le pointeur de pile de l'appelante devient le « pointeur de cadre » (*frame pointer* désigné par `%fp`) de l'appelée. `%i6` est donc le pointeur de cadre de l'appelée.
- Le registre `%i0` est utilisé par la routine appelée pour renvoyer sa valeur de retour lorsqu'elle est de type simple. S'il s'agit d'un agrégat, d'une structure par exemple, celle-ci sera retournée via la pile. C'est à la routine appelante de placer une adresse valide pointant sur une zone de la mémoire qui contiendra la valeur de retour.
- Le registre `%o7` est utilisé par la routine appelante pour placer l'adresse de l'instruction d'appel qu'elle exécute. Cette adresse permet de calculer l'adresse de retour une fois la routine appelée exécutée.

Le tableau de la figure 2 page 4 fait un inventaire des principaux registres de l'unité entière SPARC. La fonction de chacun des registres inventoriés est également proposée.

1.3 Format des instructions SPARC

Toutes les instructions (d'assemblage) d'une architecture SPARC occupent un mot mémoire — soit 4 octets — et sont de ce fait placées à des adresses multiple d'un mot (i.e. multiples de 4 dans le cas présent). Il existe quatre formats d'instructions distincts qui se distinguent par les deux premiers bits de leur code (voir figure 3).

[3] CISC est le sigle de *Complex Instruction Set Computer*. Contrairement aux architectures RISC, les architectures CISC favorise l'implémentation d'un nombre important d'instructions d'assemblage au détriment du nombre de registres.

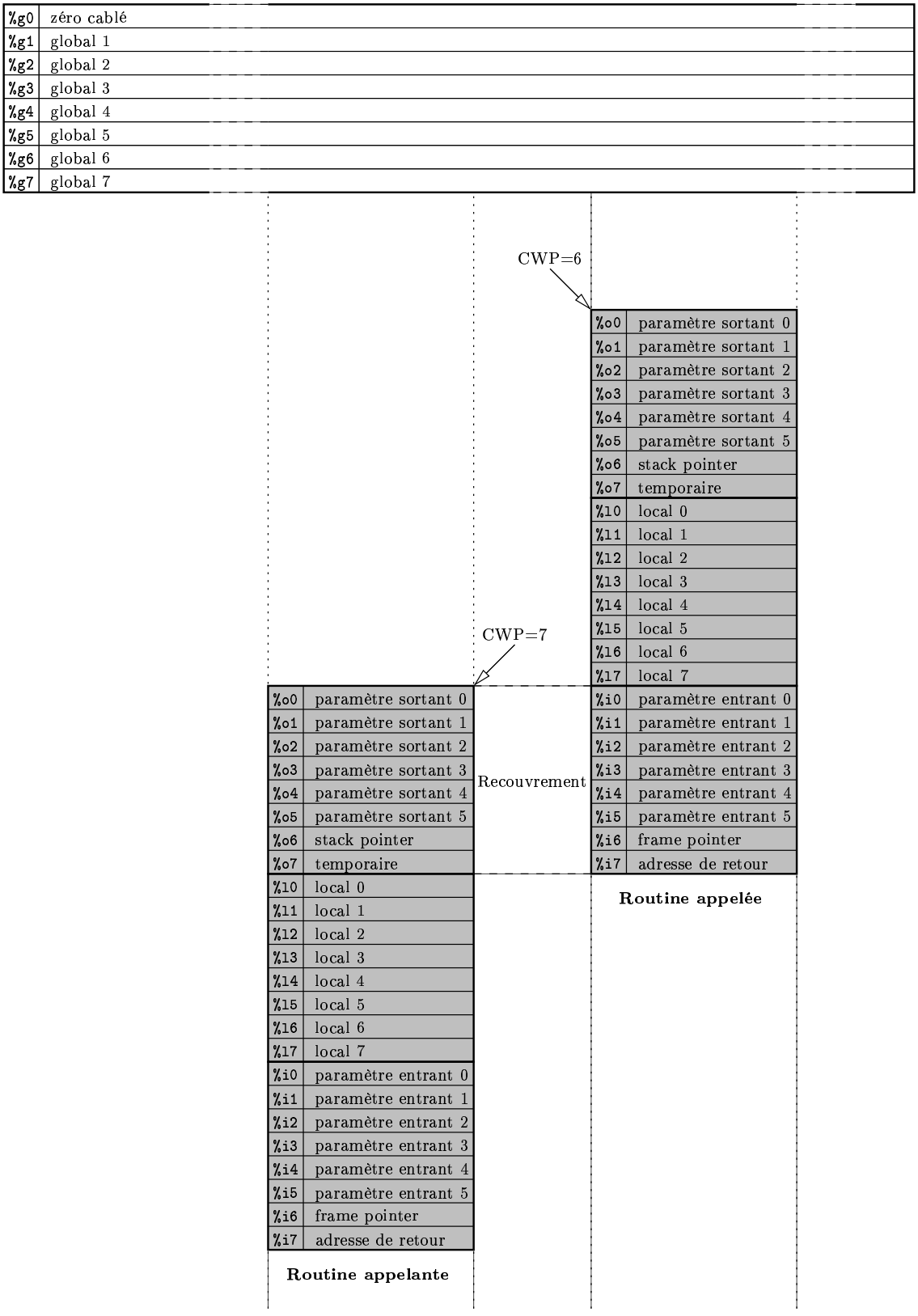
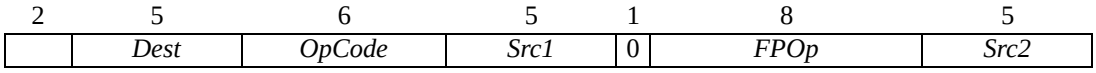


FIG. 1 – Mécanisme de fenêtrage recouvrant lors d'un appel de routine.

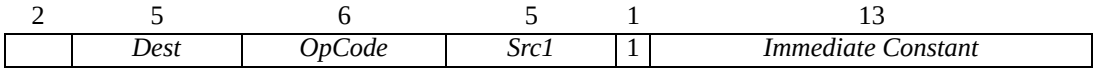
Désignation	Fonction
%y	Utilisé pour la multiplication et la division entière.
%psr	*Processor Status Register* : contient les codes de conditions entiers aussi appelés *Integer Condition Codes* et désignés par ICC.
%pc	*Program Counter* : contient l'adresse, au sein de la mémoire des instructions, de l'instruction en cours d'exécution.
%npc	*Next Program Counter* : contient l'adresse, au sein de la mémoire des instructions, de l'instruction qui sera exécutée à la suite de l'instruction en cours d'exécution.
%g0	Contient la valeur entière zéro. Cette valeur est câblée. Le registre %g0 peut être utilisé comme registre de destination au sein d'une instruction opératoire afin d'y écrire le résultat de l'opération, le contenu de %g0 demeurera toutefois égal à zéro.
%g1 ... %g7	Registres entiers globaux, c'est-à-dire accessibles en lecture et en écriture par l'ensemble des routines du programme.
%o0	Peut contenir soit le premier paramètre sortant de la routine courante, (c'est-à-dire de la routine à laquelle est associée la fenêtre de registre à laquelle appartient ce registre), auquel cas la routine appelée par la routine courante récupérera ce paramètre dans son registre d'entrée %i0 ; soit la valeur retournée par la routine appelée (la routine appelée ayant nécessairement, dans ce cas, rangé la valeur quelle devait retourner dans son registre d'entrée %i0).
%o1 ... %o5	Contiennent les paramètres sortants 1 à 5 de la routine courante, c'est-à-dire les paramètres passés à la routine appelée par la routine courante.
%o6 ou %sp	Contient le pointeur de pile de la routine courante.
%o7	Contient l'adresse, au sein de la mémoire des instructions, de l'instruction call de la routine courante, instruction qui a appelé la routine appelée. Le contenu de ce registre est utilisé par la routine appelée pour revenir à la routine courante une fois achevée l'exécution de la routine appelée.
%l0 ... %l7	Registres entiers locaux, c'est-à-dire accessibles en lecture et en écriture uniquement par la routine à laquelle est associée la fenêtre de registres à laquelle appartiennent ces registres.
%i0	Peut contenir soit le premier paramètre entrant de la routine courante, (c'est-à-dire de la routine à laquelle est associée la fenêtre de registre à laquelle appartient ce registre), auquel cas la routine ayant appelé la routine courante devra avoir placé ce paramètre dans son registre de sortie %o0 ; soit la valeur retournée par la routine courante (la routine appelante pourra dans ce cas récupérer la valeur retournée par la routine courante dans son registre de sortie %o0).
%i1 ... %i5	Contiennent les paramètres entrants 1 à 5 de la routine courante, c'est-à-dire les paramètres reçus de la routine ayant appelé la routine courante.
%i6 ou %fp	Contient le pointeur de cadre de la routine courante, c'est-à-dire le pointeur de pile de la routine ayant appelé la routine courante.
%i7	Contient l'adresse, au sein de la mémoire des instructions, de l'instruction call de la routine appelante, instruction qui a appelé la routine courante. Le contenu de ce registre est utilisé pour retourner à la routine appelante une fois achevée l'exécution de la routine courante.

FIG. 2 – Désignation et fonction des principaux registres de l'unité entière SPARC.

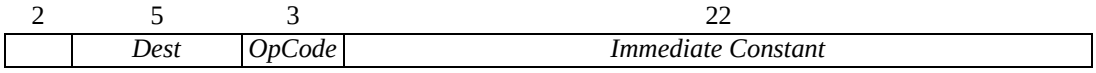
Format 1a : indirect



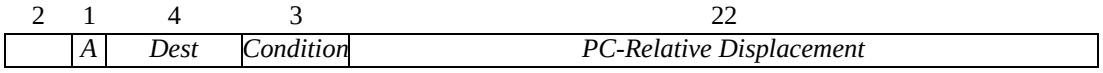
Format 1b : immédiat



Format 2 : SETHI



Format 3 : Branchements (A : *Annul Bit*)



Format 4 : CALL

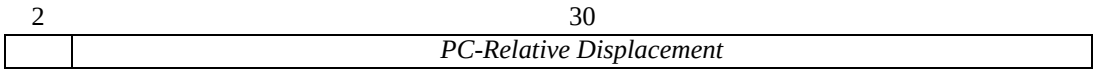


FIG. 3 – Formats des instructions SPARC.

Format 1 : Le format 1 regroupe toutes les instructions arithmétiques, logiques et flottantes, ainsi que les accès à la mémoire (*load* et *store*). Toutes ces opérations admettent deux opérandes « source » et un opérande « destination ». Ce format est divisé en deux sous-formats suivant la valeur du bit 13 : le format `1a` correspond au cas où les deux opérandes sont des registres et le format `1b` correspond au cas où le second opérande source est une valeur immédiate (c.-à-d. une constante signée codée sur 13 bits et donc comprise entre -4096 et +4095).

– Les instructions arithmétiques, logiques et flottantes réalisent le calcul suivant :

$$\boxed{\textit{Dest} = \textit{Src1 op Src2}}$$

où l'opérateur *op* est codé dans le champ *OpCode*.

– Les instructions *load* et *store* utilisent le champ *Dest* pour désigner le registre à charger ou à ranger. Chacun des deux sous-formats correspond à un mode d'adressage différent, c'est-à-dire détermine le mode de calcul de l'adresse mémoire désirée :
  – le format `1a` utilise comme adresse la valeur $\textit{Src1} + \textit{Src2}$;
  – le format `1b` utilise comme adresse la valeur de $\textit{Src1} + \textit{Immediate Constant}$.

Tous les registres peuvent être utilisés comme source ou destination.

Format 2 : Le format 2 est exclusivement dédié à l'instruction `sethi` qui sera détaillée à la section 2.6 page 12.

Format 3 : Le format 3 regroupe toutes les instructions de « branchement » conditionnels et inconditionnels (cf. section 2.4.1 page 11). Le champ *Condition* permet de spécifier la condition de branchement (égal, supérieur ou égal, ...). Le champ *PC-Relative Displacement* permet de spécifier une adresse relative de branchement sur 22 bits, ce déplacement étant exprimé en mots. Le bit « A » (*Annul Bit*) est une astuce permettant de remplir certains intervalles de délai. Si ce bit n'est pas positionné pour une instruction de branchement conditionnel, alors l'instruction de délai est normalement exécutée. En revanche, si ce bit est positionné alors l'instruction de délai n'est exécutée que si le branchement est réalisé.

Format 4 : Un premier moyen d'appeler une routine est d'utiliser l'instruction `call` (cf. section 2.5 page 12), qui répond au format 4 et dont le déplacement relatif doit être exprimé en mots sur 30 bits. De la sorte, 4 giga-octets

peuvent être adressés. Cette instruction place l'adresse de retour de la routine appelée dans le registre de sortie `%o7` de la routine appelante (cette adresse est donc accessible par la routine appelée dans son registre d'entrée `%i7`). Un second moyen d'appeler une routine est de recourir à l'instruction `jmpl` qui répond au format 1. Dans ce cas l'adresse de branchement (dans la mémoire des instructions) peut être calculée lors de l'exécution du programme. L'instruction `jmpl` ne place pas l'adresse de retour dans le registre `%o7` de la routine appelante, il est donc possible de placer cette adresse de retour dans n'importe quel registre de sortie de la routine appelante[4].

2 Syntaxe de l'assembleur SPARC

2.1 Caractéristiques lexicales : notations et directives

Dans la suite de ce document, *simm13* désignera une constante signée codée en binaire sur 13 bits par son complément à 2, *const22* désignera une constante codée sur 22 bits, et *valeur* fera référence à une valeur codée sur 32 bits. La notation *reg* sera utilisée pour désigner un registre de la fenêtre des registres. reg_{s1} représentera un registre source contenant le premier opérande d'une opération et *reg_ou_imm* représentera soit un registre source contenant le second opérande d'une opération, soit une constante signée codée en binaire sur 13 bits par son complément à 2. La notation reg_{dest} fera référence à un registre destination, c'est-à-dire un registre destiné à contenir le résultat d'une opération. Enfin, *adresse* désignera une adresse mémoire et *label* désignera une étiquette.

Chaînes de caractères : Les chaînes de caractères doivent être entourées de simples quotes « ' » ou de doubles quotes « " ». Leur valeur numérique est la valeur du code ASCII de leur premier caractère.

Expressions constantes : Il est possible de construire des expressions constantes. Les opérateurs reconnus au sein d'une expression constante sont les suivants : `+`, `-`, `*`, `/`, `%`, `^`, `<<`, `>>`, `&`, `|`, `-` (unaire : complément à 2), `~` (unaire : complément à 1), `%lo` (retourne les 10 bits de poids faible de son argument), et `%hi` (retourne les 22 bits de poids fort de son argument).

Adresses : Une adresse mémoire peut avoir la forme suivante : reg_{s1}, $reg_{s1} + reg_{s2}$, $reg_{s1} + simm13$, $reg_{s1} - simm13$, *simm13* ou $simm13 + reg_{s1}$.

Commentaires : Des commentaires peuvent être introduits dans le code de deux manières différentes. Un commentaire sur plusieurs lignes sera entouré des symboles « `/*` » et « `*/` » ou à la suite du symbole « `!` » pour commenter la fin d'une ligne.

Étiquettes : Les étiquettes sont considérées comme étant des constantes dont la valeur est leur propre adresse. Les étiquettes sont suffixées par le symbole « `:` » (« `LABEL_0:` » par exemple).

Pour toutes les instructions, l'opérande destination est toujours placée en dernier (c.-à-d. le plus à droite dans l'instruction). Lorsque le contenu d'une zone de la mémoire est spécifié par son adresse, celle-ci doit être placée entre des symboles « `[` » et « `]` » (par exemple : `ld [%fp-8], %o0`, ou `sth %l3, [%sp+96]`, ou encore `swap [%sp+8], %o2`). Une référence à une adresse mémoire est donnée directement.

Comme dans tout langage d'assemblage, la notion de *section* (ou *segment*) est présente. Une section représente la plus petite unité relogeable d'un programme objet. Lors de l'écriture d'un programme en langage d'assemblage, il est impératif de spécifier par une directive la section courante, c'est-à-dire la section qui va recevoir le code généré lors de la phase d'assemblage. La liste des sections prédéfinies les plus importantes est la suivante.

- `".bss"` : section qui contient des données non-initialisées accessibles en lecture et en écriture.
- `".data"` ou `".data1"` : section qui contient des données initialisées accessibles en lecture et en écriture.
- `".rodata"` ou `".rodata1"` : section qui contient des données initialisées accessibles en lecture seulement.
- `".text"` : section qui contient le code exécutable.

L'assembleur SPARC reconnaît un certain nombre de *directives* (aussi appelées *pseudo-instructions*). Ces directives ne doivent pas être confondues avec les *macro-instructions* qui génèrent du code (cf. section 2.7 page 13).

- `.align` *limite* : aligne le mot suivant sur une frontière de *limite* octets.
- `.ascii "`*string*`"` : génère la chaîne de caractères *string*.
- `.asciz "`*string*`"` : génère la chaîne de caractères *string* et y concatène le caractère nul « `\0` ».
- `.byte` *valeur_codée_sur_8_bits* : génère l'octet de valeur *valeur_codée_sur_8_bits*.

[4] En fin d'exécution d'une routine appelée par l'instruction `call`, le retour à la routine appelante est obtenu en exécutant l'instruction `jmpl %i7, 8, %g0`.

– `.global` *symbole* : déclare le symbole *symbole* global, c'est-à-dire accessible de l'extérieur ou provenant de l'extérieur.
– `.half` *valeur_codée_sur_16_bits* : génère le demi-mot de valeur *valeur_codée_sur_16_bits*.
– `.local` *symbole* : déclare le symbole *symbole* local.
– `.section "`*section*`"` : change la section courante (`.section ".text"` par exemple).
– `.word` *valeur_codée_sur_32_bits* : génère le mot de valeur *valeur_codée_sur_32_bits*.

2.2 Instructions de transfert de données entières

2.2.1 Instructions de chargement d'entiers

Les instructions du tableau de la figure 4 copient un octet, un demi-mot ou un mot de la mémoire vers le registre reg_{dest}.

Instruction	Arguments	Description
`ldsb`	*adresse*, reg_{dest}	*LoaD Signed Byte*
`ldsh`	*adresse*, reg_{dest}	*LoaD Signed Half word*
`ldub`	*adresse*, reg_{dest}	*LoaD Unsigned Byte*
`lduh`	*adresse*, reg_{dest}	*LoaD Unsigned Half word*
`ld`	*adresse*, reg_{dest}	*LoaD word*
`ldd`	*adresse*, reg_{dest}	*LoaD Double word*

FIG. 4 – Instructions de chargement des entiers.

Un octet ou un demi-mot est justifié à droite dans le registre destination reg_{dest}, c'est-à-dire que les bits de l'octet ou du demi-mot sont copiés dans les bits de poids faible du registre reg_{dest}. Ainsi, l'instruction `ldsb` charge, depuis la mémoire des données, l'octet situé à l'adresse *adresse* dans les 8 bits de poids faible du registre reg_{dest} puis étend le bit de signe — bit de poids fort — de cet octet aux 24 bits de poids fort du registre reg_{dest}. Par la suite, toute instruction qui préserve de la sorte le signe de son argument sera appelée *instruction signée*. L'instruction `ldsh` est un autre exemple d'instruction signée. `ldsh` charge, depuis la mémoire des données, le demi-mot situé à l'adresse *adresse* dans le registre reg_{dest} puis étend le bit de signe de ce demi-mot aux 16 bits de poids fort du registre reg_{dest}.

Alors qu'une instruction signée agit comme si elle opérait sur un argument signé, une instruction non signée agit comme si elle opérait sur un argument non signé. Ainsi avec une instruction non signée (par exemple `ldub` ou `lduh`) les bits de poids fort du registre reg_{dest} sont remplis avec des zéros.

L'intruction `ldd` charge un double-mot depuis la mémoire des données vers une paire de registres consécutifs. Le mot de poids fort du double-mot est chargé dans le registre reg_{dest}[5] et le mot de poids faible, c'est-à-dire le mot situé à l'adresse $adresse + 4$ dans la mémoire des données, est chargé dans le registre de numéro impair suivant.

2.2.2 Instructions de rangement d'entiers

Les instructions du tableau de la figure 5 copient en mémoire le mot, le demi-mot de poids faible ou l'octet de poids faible du registre *reg*.

Instruction	Arguments	Description
`stb`	*reg*, *adresse*	*STore Byte*
`sth`	*reg*, *adresse*	*STore Half word*
`st`	*reg*, *adresse*	*STore word*
`std`	*reg*, *adresse*	*STore Double word*

FIG. 5 – Instructions de rangement des entiers.

[5]Le numéro de ce registre doit nécessairement être pair (par exemple `%o2`, `%l6`, ...)

Le rangement en mémoire d'un double-mot s'effectue de manière symétrique par rapport au chargement d'un double-mot dans un registre : le mot de poids fort du double-mot, initialement contenu dans le registre *reg* — registre dont le numéro doit impérativement être pair — est rangé dans la mémoire des données à l'adresse *adresse*, le mot de poids faible du double-mot, initialement contenu dans le registre de numéro suivant, est rangé pour sa part dans la mémoire des données à l'adresse $adresse + 4$.

2.2.3 Instruction atomique de chargement-rangement d'un octet non signé

L'instruction du tableau de la figure 6 copie dans le registre reg_{dest} l'octet situé à l'adresse *adresse* dans la mémoire des données puis écrit en mémoire la valeur hexadécimale FF à la place de l'octet.

Instruction	Arguments	Description
`ldstub`	*adresse*, reg_{dest}	*Atomic LoaD STore Unsigned Byte*

FIG. 6 – Instruction atomique de chargement-rangement d'un octet non signé.

Cette instruction est exécutée de manière *atomique*, c'est-à-dire qu'aucune interruption (logicielle ou matérielle) ne peut interrompre l'exécution de cette instruction.

2.2.4 Instruction d'échange du contenu d'un registre et d'un mot mémoire

L'instruction du tableau de la figure 7 échange le contenu du registre reg_{dest} avec le mot mémoire situé à l'adresse *adresse* dans la mémoire des données.

Instruction	Arguments	Description
`swap`	*adresse*, reg_{dest}	*SWAP register with memory*

FIG. 7 – Instruction d'échange du contenu d'un registre et d'un mot mémoire.

À l'instar de l'instruction `ldstub`, l'instruction `swap` est exécutée de manière atomique.

2.3 Instructions arithmétiques entières et logiques

Toutes les instructions arithmétiques et logiques rangent le résultat de leur calcul dans le registre destination reg_{dest} passé en troisième opérande.

2.3.1 Instructions logiques

Les instructions du tableau de la figure 8 page 9 réalisent les opérations logiques bit-à-bit. Les opérandes sont soit deux registres, soit un registre et une constante signée codée sur 13 bits. Seules les instructions dont le suffixe est `cc` modifient les codes de condition. Enfin, les instructions `andn`, `orn`, `andncc`, `orncc` calculent la négation logique de leur second opérande avant d'appliquer l'opérateur spécifié.

2.3.2 Instructions de décalages

Les instructions du tableau de la figure 9 effectuent les opérations de décalages arithmétiques et logiques sur le registre reg_{s1}. Le nombre de décalages opérés est donné par les 5 bits de poids faible du second opérande *reg_ou_imm*, qui est soit un registre soit une constante codée sur 13 bits. Il n'est donc pas possible d'appliquer au contenu d'un registre un décalage de plus de 32 bits vers la droite ou la gauche. Il ne s'agit toutefois en rien d'une restriction dans la mesure où un registre ne contient que 32 bits. Dans le cas où *reg_ou_imm* est un registre, ses 27 bits de poids forts sont ignorés et dans le cas où *reg_ou_imm* est une constante codée sur 13 bits, les 8 bits de poids fort de cette constante doivent être positionnés à zéro (c'est-à-dire que cette constante doit être comprise entre 0 et 31).

Alors que les décalages logiques remplacent les bits libérés par des zéros, le décalage arithmétique les remplace par le bit de poids fort du registre reg_{s1}. Les instructions de décalages ne modifient pas les codes de condition.

Instruction	Arguments	Description
`and`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*AND*
`andcc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*AND with ICC updating*
`andn`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*AND Not*
`andncc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*AND Not with ICC updating*
`or`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*OR*
`orcc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*OR with ICC updating*
`orn`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*OR Not*
`orncc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Or Not with ICC updating*
`xor`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*EXclusive OR*
`xorcc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*EXclusive OR with ICC updating*
`xnor`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*EXclusive NOR*
`xnorcc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*EXclusive NOR with ICC updating*

FIG. 8 – Instructions logiques.

Instruction	Arguments	Description
`sll`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Shift Left Logical*
`srl`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Shift Right Logical*
`sra`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Shift Right Arithmetic*

FIG. 9 – Instructions de décalages arithméthiques et logiques.

2.3.3 Instructions d'addition entière

Les instructions `add` et `addcc` du tableau de la figure 10 page 9 calculent la somme de leurs deux premiers opérandes et rangent le résultat dans le troisième opérande qui est le registre reg_{dest}. Si le second opérande est une constante signée codée sur 13 bits, alors le codage de cette constante est étendu sur 32 bits avec conservation du signe de la constante (c.-à-d. le bit de signe de la constante est reproduit 19 fois à gauche de son codage).

Les instructions `addx` et `addxcc` du tableau de la figure 10 ajoutent à la somme de leurs deux premiers opérandes la valeur du bit de retenue (*carry*) contenu dans les codes de condition. Les seules instructions de sommation qui modifient les codes de condition sont `addcc` et `addxcc`. Une addition engendre un débordement (*overflow*) si les deux premiers opérandes de l'instruction ont même signe et que le signe du résultat est différent de celui des opérandes.

Instruction	Arguments	Description
`add`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*integer ADD*
`addcc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*integer ADD with ICC updating*
`addx`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*integer ADD with carry*
`addxcc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*integer ADD with carry and ICC updating*

FIG. 10 – Instructions arithméthiques d'addition entière.

2.3.4 Instructions de soustraction entière

Les instructions du tableau de la figure 11 page 10 soustraient de la valeur de leur premier opérande la valeur de leur second opérande ($reg_{s1} - reg\_ou\_imm$) et rangent le résultat dans leur troisième opérande qui est le registre reg_{dest}. Si le second opérande est une constante signée codée sur 13 bits, alors le codage de cette constante est étendu sur 32 bits avec conservation du signe de la constante (c.-à-d. le bit de signe de la constante est reproduit 19 fois à gauche de son codage).

Les instructions `subx` et `subxcc` du tableau de la figure 11 soustraient à la différence de leurs deux premiers opérandes la valeur du bit de retenue (*carry*) contenu dans les codes de condition. Les seules instructions de différence qui modifient les codes de condition sont `subcc` et `subxcc`. Une différence engendre un débordement (*overflow*) si les deux premiers opérandes de l'instruction ont un signe opposé et que le signe du résultat est différent de celui du premier opérande reg_{s1}.

Instruction	Arguments	Description
`sub`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*integer SUBstract*
`subcc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*integer SUBstract with ICC updating*
`subx`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*integer SUBstract with carry*
`subxcc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*integer SUBstract with carry and ICC updating*

FIG. 11 – Instructions arithméthiques de soustraction entière.

2.3.5 Instructions de multiplication entière

Les instructions du tableau de la figure 12 page 10 opèrent le produit de leurs deux premiers opérandes. Ces instructions calculent donc le produit de deux entiers codés sur 32 bits. Elles fournissent cependant un résultat codé sur 64 bits. Les 32 bits de poids faible du résultat sont rangés dans le registre reg_{dest} passé en troisième opérande à l'instruction de multiplication. Les 32 bits de poids fort du résultat sont rangés dans le registre spécial `%y`. De même que pour les instructions d'addition et de soustraction, si le second opérande est une constante signée codée sur 13 bits, alors le codage de cette constante est étendu sur 32 bits avec conservation du signe de la constante (c.-à-d. le bit de signe de la constante est reproduit 19 fois à gauche de son codage).

Les seules instructions de multiplication qui modifient les codes de condition sont `umulcc` et `smulcc`.

> Les instructions `umulcc` et `smulcc` modifient les codes de condition suivant la valeur du mot de *poids faible* du résultat.

Le bit « `n` » (*negative*) est positionné (c.-à-d. $\texttt{n} = 1$) si le bit de signe de reg_{dest} est égal à 1. Le bit « `z` » (*zero*) est positionné si le contenu du registre reg_{dest} est égale à zéro. Les bits « `v` » (*overflow*) et « `c` » (*carry*) ne sont jamais positionnés par `umulcc` et `smulcc`.

Instruction	Arguments	Description
`umul`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Unsigned integer MULtiply*
`umulcc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Unsigned integer MULtiply with ICC updating*
`smul`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Signed integer MULtiply*
`smulcc`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Signed integer MULtiply with ICC updating*

FIG. 12 – Instructions arithméthiques de multiplication entière.

2.3.6 Instructions de division entière

Les instructions du tableau de la figure 13 opèrent la division d'un entier codé sur 64 bits par un entier codé sur 32 bits. Le double-mot de 64 bits est représenté par les registres `%y` pour le mot de poids fort du double-mot et par reg_{s1} pour le mot de poids faible du double-mot. *reg_ou_imm* contient le diviseur. Si *reg_ou_imm* est une constante codée sur 13 bits alors le codage de cette constante est étendu sur 32 bits avec conservation du signe de la constante (c.-à-d. le bit de signe de la constante est reproduit 19 fois à gauche de son codage). Le résultat de la division, le quotient, est rangé dans reg_{dest}. Le reste de la division ainsi que le mot de poids fort du dividende sont perdus. La division peut être opérée sur des entiers non-signés ou sur des entiers signés. Dans le deuxième cas, le résultat n'est pas correct lorque les opérandes ne sont pas de même signe. À titre d'exemple, le résultat de la division de -3 par 2 est -1 (le reste étant nécessairement -1) alors que le résultat devrait être -2 et le reste $+1$.

Les seules instructions de division qui modifient les codes de condition sont `udivcc` et `sdivcc`.

Instruction	Arguments	Description
udiv	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Unsigned integer DIVide*
udivcc	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Unsigned integer DIVide with ICC updating*
sdiv	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Signed integer DIVide*
sdivcc	reg_{s1}, *reg_ou_imm*, reg_{dest}	*Signed integer DIVidewith ICC updating*

FIG. 13 – Instructions arithméthiques de division entière.

2.4 Instructions de tranfert de contrôle

2.4.1 Instructions de branchement suivant les codes de condition entiers

Les instructions du tableau de la figure 14 page 11 effectuent un branchement, conditionné par la valeur des codes de condition, vers l'instruction repérée par l'étiquette *label*. Les instructions de branchement sont des instructions retardées. Elles sont donc suivies d'une instruction de délai.

Instruction	Arguments	Description	Test effectué sur les codes de condition entiers (ICC)
ba,a	*label*	*Branch Always*	1
bn,a	*label*	*Branch Never*	0
bne,a	*label*	*Branch on Not Equal*	$\overline{\mathtt{z}}$
be,a	*label*	*Branch on Equal*	z
bg,a	*label*	*Branch on Greater*	$\overline{\mathtt{z}+(\mathtt{n}\oplus\mathtt{v})}$
bge,a	*label*	*Branch on Greater or Equal*	$\overline{\mathtt{n}\oplus\mathtt{v}}$
bl,a	*label*	*Branch on Less*	$\mathtt{n}\oplus\mathtt{v}$
ble,a	*label*	*Branch on Less or Equal*	$\mathtt{z}+(\mathtt{n}\oplus\mathtt{v})$
bgu,a	*label*	*Branch on Greater Unsigned*	$\overline{\mathtt{c}+\mathtt{z}}$
bcc,a	*label*	*Branch on Carry Clear (branch on greater or equal unsigned)*	$\overline{\mathtt{c}}$
bcs,a	*label*	*Branch on Carry Set (branch on less unsigned)*	c
bleu,a	*label*	*Branch on Less or Equal Unsigned*	$\mathtt{c}+\mathtt{z}$
bpos,a	*label*	*Branch on POSitive*	$\overline{\mathtt{n}}$
bneg,a	*label*	*Branch on NEGative*	n
bvc,a	*label*	*Branch on oVerflow Clear*	$\overline{\mathtt{v}}$
bvs,a	*label*	*Branch on oVerflow Set*	v

FIG. 14 – Instructions de branchement suivant les codes de condition entiers.

Pour les branchements conditionnels, cette instruction de délai est toujours exécutée dès lors que le branchement est effectué, c'est-à-dire dès lors que l'expression du test effectué sur les codes de condition (ICC) est vraie. Si le branchement n'est pas effectué, l'instruction de délai est exécutée suivant la valeur du champ *Annul Bit*, positionné en suffixant le mnémonique de l'instruction avec « ,a ». Si le branchement n'est pas effectué et que le champ *Annul Bit*

est positionné alors l'instruction de délai sera sans effet. Si le branchement n'est pas effectué et que le champ *Annul Bit* n'est pas positionné alors l'instruction de délai sera effectivement exécutée.

Pour les branchements inconditionnels `ba` et `bn`, si le champ *Annul Bit* est positionné alors l'instruction de délai n'est jamais exécutée.

2.5 Instructions d'appel de routine et de saut d'instructions avec lien

L'instruction `call`, d'appel de routine avec lien, du tableau de la figure 15 page 12 effectue un saut inconditionnel, retardé, à l'étiquette *label* passée en argument. De plus, elle écrit dans le registre `%o7` la valeur courante du compteur de programme `%pc`.

Du point de vue du mécanisme de fonctionnement de l'instruction `call`, tout se passe comme si l'étiquette *label* passée en argument était une référence à l'adresse — dans la mémoire des intructions — de la première instruction à exécuter au sein de la routine appelée.

Ainsi, l'instruction `call` place le contenu du registre `%pc` dans le registre `%o7` de la fenêtre de registres associée à la routine en cours d'exécution, puis effectue un saut à l'adresse désignée par l'étiquette *label*, ce qui revient à affecter au compteur de programme `%pc` l'adresse — dans la mémoire des instructions — de la première instruction à exécuter de la routine appelée.

Instruction	Arguments	Description
`call`	*label*	*CALL and link*
`jmpl`	*adresse*, reg_{dest}	*JuMP and link*

FIG. 15 – Instructions d'appel de routine et de saut d'instructions avec lien.

L'instruction `jmpl`, de saut d'instructions avec lien, du tableau de la figure 15 effectue un saut inconditionnel, retardé, à l'adresse *adresse* passée en argument. De plus, elle copie dans le registre reg_{dest} la valeur courante du compteur de programme `%pc`.

L'instruction `jmpl` peut être utilisée afin de réaliser l'appel d'une routine dont l'adresse est déterminée lors de l'exécution du programme. Dans ce cas, le registre reg_{dest} est utilisé pour contenir la valeur courante du compteur de programme `%pc`. Avec l'instruction `jmpl` il est donc possible d'effectuer un appel de routine en plaçant le compteur de programme dans un autre registre de sortie que `%o7`.

L'instruction jmpl est communément utilisée à la suite d'un appel de routine pour retourner à la routine appelante. Si le registre `%i7` contient la valeur du compteur de programme de la routine appelante lors de l'appel de la routine courante, alors la routine courante peut « rendre la main » à la routine appelante en invoquant l'instruction `jmpl [%i7 + 8], %g0`. En effet, `%i7 + 8` est l'adresse de l'instruction qui suit l'instruction d'appel de la routine courante et de son instruction de délai (qu'on ne souhaite en général pas exécuter deux fois). Le compteur de programme courant étant sans intêret dans le cas d'un retour de routine, il est oublié dans le registre `%g0`.

2.6 Instructions particulières

Instruction vide : La syntaxe de l'instruction vide est simplement `nop`. Cette instruction est généralement utilisée comme instruction de délai d'une instuction retardée.

Instruction `sethi` : (cf. tableau de la figure 16) L'instruction `sethi` annule les 10 bits de poids faible du registre reg_{dest} et remplace ces 22 bits de poids fort par la valeur constante donnée en premier argument. L'opérateur unaire `%hi` extrait les 22 bits de poids fort de son opérande *valeur* (une étiquette en générale).

Instruction	Arguments	Description
`sethi`	*const22*, reg_{dest}	*SET HIgh-order 22 bits*
`sethi`	`%hi(`*valeur*`)`, reg_{dest}	*SET HIgh-order 22 bits*

FIG. 16 – Instruction `sethi`.

Instructions `save` et `restore` : (cf. tableau de la figure 17) Ces deux instructions permettent de manipuler la fenêtre de registres. L'instruction `save` retranche 1 (modulo le nombre de fenêtres de registres dont est doté le processeur) au registre d'état contenant le pointeur de fenêtre courant (*Current Window Pointer* ou CWP) et compare le résultat avec le bit correspondant du registre contenant le masque de fenêtres invalides (*Window Invalid Mask* ou WIM). Si ce bit est positionné, alors une exception est générée. Dans le cas contraire, le CWP est valide et une nouvelle fenêtre de registre est disponible. L'instruction `restore` présente un comportement symétrique.

Instruction	Arguments	Description
`save`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*SAVE caller's window*
`restore`	reg_{s1}, *reg_ou_imm*, reg_{dest}	*RESTORE caller's window*

FIG. 17 – Instructions save et restore.

Instructions d'accès au registres d'état : Il est possible d'accéder au registres spéciaux du processeur (tels que `%y`, `%psr`, `%wim`, ...) en invoquant les instructions `rd` (*ReaD*) et `wr` (*WRite*). Ces accès pouvant engendrer des comportements instables du programme, ils sont fortement déconseillés.

2.7 Macro-instructions

Les macro-instructions ne sont que des pseudo-instructions, c'est-à-dire qu'elles représentent une écriture plus simple de certaines instructions fréquemment employées. Ces macro-instructions (ainsi que les instructions qui leur sont équivalentes) sont présentées dans le tableau 18 page 14.

3 Fonctionnement de la pile

Dans l'architecture SPARC, la pile est une zone de la mémoire qui croit des adresses supérieures vers les adresses inférieures. De la place sur la pile appelée « *cadre de pile* » (*stack frame*) est reservée pour chaque procédure afin de :

- stocker les variables locales à la procédure ;
- passer les paramètres (au-delà du sixième) aux procédures appelées ;
- permettre au système de sauvegarder les registres *in* et *local* de la procédure s'il n'y a plus assez de fenêtres de registres disponibles.

La liste suivante décrit plus précisemment ce qu'il faut allouer sur la pile :

- Pour toutes les procédures, il faut *toujours* allouer :
  - 16 mots (64 octets), toujours à partir de `%sp` pour sauvegarder, le cas échéant, les registres *in* et *local* de la procédure.
- Pour toutes les procédure non-feuilles (c.à-d. qui appellent d'autres procédures), il faut allouer :
  - 1 mot (4 octets) pour passer un paramètres « caché ». C'est utilisé si la procédure appelée doit retourner une donnée complexe (structure en C) ;
  - 6 mots dans lesquels la procédure appelée peut copier les paramètres devant être adressés (on a besoin d'une adresse mémoire).
- Pour les procédures en ayant besoin, il faut allouer de la place pour :
  - les paramètres « sortants » au-delà du sixième ;
  - toutes les variables locales à la procédure (tableaux, variables devant être adressables ou pour lesquelles il n'y a plus assez de registres disponibles)

Les variables locales sur la pile sont adressées avec des déplacements négatifs par rapport à `%fp`, le reste est adressé avec des déplacements positifs par rapport à `%sp`.

Le pointeur de pile `%sp` doit toujours être alignée sur une frontière de double mots (c.-à-d. sur des adresses multiples de 8).

Macro	**Arguments**	**Instruction SPARC**	**Description**
cmp	*reg$_{s1}$, reg_ou_imm*	subcc *reg$_{s1}$, reg_ou_imm*, %g0	*CoMPare*
jmp	*adresse*	jmpl *adresse*, %g0	*JuMP*
call	*adresse*	jmpl *adresse*, %o7	*CALL*
tst	*reg$_{s2}$*	orcc %g0, *reg$_{s2}$*, %g0	*TeST*
ret		jmpl %i7 + 8, %g0	*RETurn from subroutine*
retl		jmpl %o7 + 8, %g0	*RETurn from Leaf subroutine*
set	*valeur*, *reg$_{dest}$*	sethi %hi(*valeur*), *reg$_{dest}$* or *reg$_{dest}$*, %lo(*valeur*), *reg$_{dest}$*	*SET*
not	*reg$_{s1}$, reg$_{dest}$*	xnor *reg$_{s1}$*, %g0, *reg$_{dest}$*	*one's complement*
not	*reg$_{dest}$*	xnor *reg$_{dest}$*, %g0, *reg$_{dest}$*	*one's complement*
neg	*reg$_{s2}$, reg$_{dest}$*	sub %g0, *reg$_{s2}$*, *reg$_{dest}$*	*two's complement*
neg	*reg$_{dest}$*	sub %g0, *reg$_{dest}$*, *reg$_{dest}$*	*two's complement*
inc	*reg$_{dest}$*	add *reg$_{dest}$*, 1, *reg$_{dest}$*	*INCrement by 1*
inc	*simm13*, *reg$_{dest}$*	add *reg$_{dest}$*, *simm13*, *reg$_{dest}$*	*INCrement by simm13*
inccc	*reg$_{dest}$*	addcc *reg$_{dest}$*, 1, *reg$_{dest}$*	*INCrement by 1 with ICC updating*
inccc	*simm13*, *reg$_{dest}$*	addcc *reg$_{dest}$*, *simm13*, *reg$_{dest}$*	*INCrement by simm13 with ICC updating*
dec	*reg$_{dest}$*	sub *reg$_{dest}$*, 1, *reg$_{dest}$*	*DECrement by 1*
dec	*simm13*, *reg$_{dest}$*	sub *reg$_{dest}$*, *simm13*, *reg$_{dest}$*	*DECrement by simm13*
deccc	*reg$_{dest}$*	subcc *reg$_{dest}$*, 1, *reg$_{dest}$*	*DECrement by 1 with ICC updating*
deccc	*simm13*, *reg$_{dest}$*	subcc *reg$_{dest}$*, *simm13*, *reg$_{dest}$*	*DECrement by simm13 with ICC updating*
btst	*reg_ou_imm*, *reg$_{s1}$*	andcc *reg$_{s1}$*, *reg_ou_imm*, %g0	*Bit TeST with ICC updating*
bset	*reg_ou_imm*, *reg$_{dest}$*	or *reg$_{dest}$*, *reg_ou_imm*, *reg$_{dest}$*	*Bit SET*
bclr	*reg_ou_imm*, *reg$_{dest}$*	andn *reg$_{dest}$*, *reg_ou_imm*, *reg$_{dest}$*	*Bit CLeaR*
btog	*reg_ou_imm*, *reg$_{dest}$*	xor *reg$_{dest}$*, *reg_ou_imm*, *reg$_{dest}$*	*Bit TOGgle*
clr	*reg$_{dest}$*	or %g0, %g0, *reg$_{dest}$*	*CLeaR register*
clrb	*adresse*	stb %g0, *adresse*	*CLeaR Byte*
clrh	*adresse*	sth %g0, *adresse*	*CLeaR Half-word*
clr	*adresse*	st %g0, *adresse*	*CLeaR word*
mov	*reg_ou_imm*, *reg$_{dest}$*	or %g0, *reg_ou_imm*, *reg$_{dest}$*	*MOVe*

FIG. 18 – Macro-instructions.

La cadre de pile est réservé grâce à l'instruction `save %sp, -taille, %sp`. Cette instruction a pour effet de déplacer la fenêtre de registres (cf. section 2.6, page 12) et de soustraire *taille* à `%sp` (`%fp` contient la valeur de l'ancien `%sp`). La valeur *taille* doit être un multiple de 8.

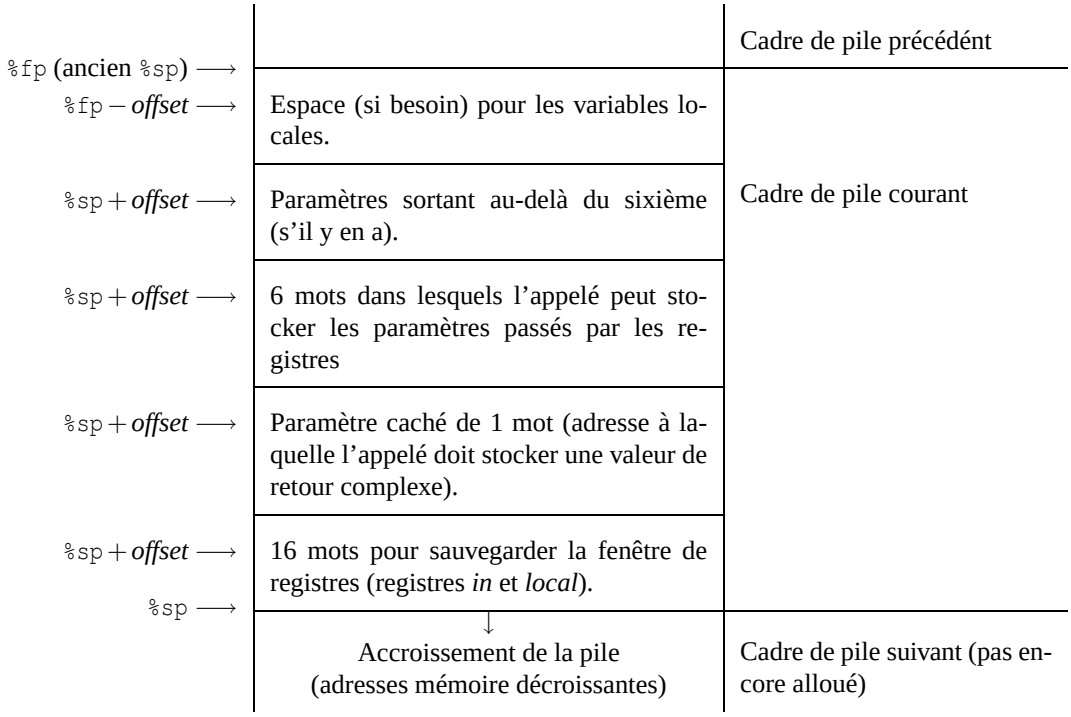


FIG. 19 – Le cadre de pile

La figure 19 représente l'état de la pile pour une procédure non-feuille.

4 Application

La figure 20 page 16 propose un exemple de programme écrit en assembleur SPARC . Ce programme invite l'utilisateur à saisir un entier naturel *n* au clavier. Il appelle ensuite une routine qui calcule la somme des *n* premiers entiers de manière itérative. Enfin, le programme affiche à l'écran le résultat du calcul.

Ce programme n'a pas été élaboré pour être stable ou même performant. La seule raison pour laquelle il est proposé est de permettre au lecteur d'assimiler les bases de la syntaxe générale de l'assembleur SPARC

5 Références

`http://www.sparc.org/standards.html`

```
.section        ".text"
        .align 4
        .global somme

somme:
        save %sp, -64, %sp
        mov %i0, %l0
        mov 0, %i0
        cmp %l0, %g0
        be .fin_somme
        nop

.debut_iteration:
        cmp %l0, %g0
        be .fin_somme
        nop
        add %i0, %l0, %i0
        dec %l0
        ba .debut_iteration
        nop

.fin_somme:
        ret
        restore


.section        ".rodata"
        .align 8

.PRINTF0:
        .asciz "Entrez un entier naturel : "
.PRINTF1:
        .asciz "La somme des i de 1 à %d est %d.\n"
.SCANF0:
        .asciz "%d"

.section        ".text"
        .align 4
        .global main

main:
        save %sp, -104, %sp

        sethi %hi(.PRINTF0), %o0
        or %o0, %lo(.PRINTF0), %o0
        call printf
        nop
        sethi %hi(.SCANF0),%o0
        or %o0, %lo(.SCANF0), %o0
        add %fp, -4, %o1
        call scanf
        nop
        ld [%fp-4], %o0
        mov %o0, %o1
        call somme
        nop
        mov %o0, %o2
        sethi %hi(.PRINTF1), %o0
        or %o0, %lo(.PRINTF1), %o0
        call printf
        nop
.fin_main:
        clr %i0
        ret
        restore
```

FIG. 20 – Listing d'un programme calculant de manière itérative la somme des *n* premiers entiers.