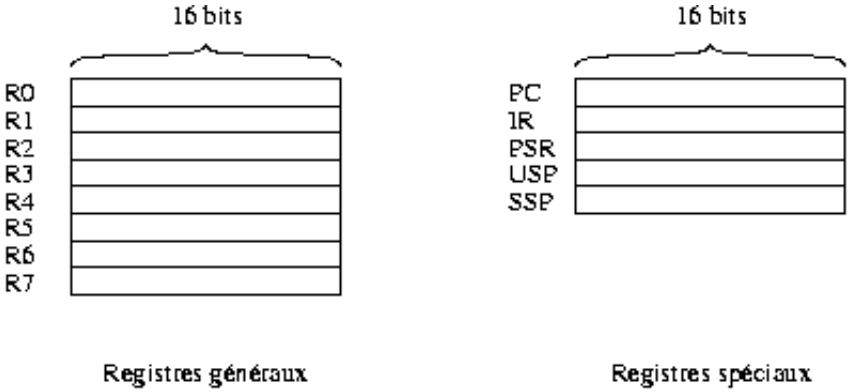
10 Micro-processeur LC-3

Le micro-processeur LC-3 est à vocation pédagogique. Il n'existe pas de réalisation concrète de ce processeur mais il existe des simulateurs permettant d'exécuter des programmes. L'intérêt de ce micro-processeur est qu'il constitue un bon compromis de complexité. Il est suffisamment simple pour qu'il puisse être appréhendé dans son ensemble et que son schéma en portes logiques soit accessible. Il comprend cependant les principaux mécanismes des micro-processeurs (appels système, interruptions) et son jeu d'instructions est assez riche pour écrire des programmes intéressants.

10.1 Registres

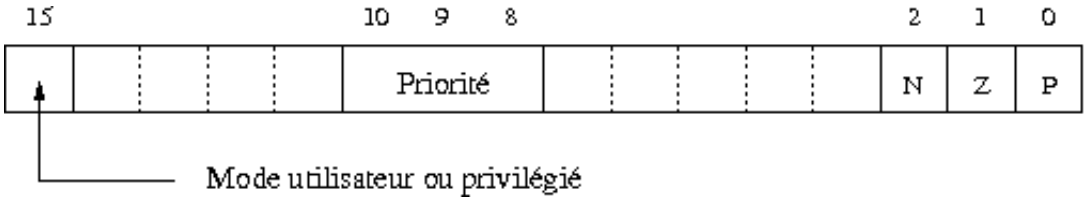
Le micro-processeur LC-3 dispose de 8 registres généraux 16 bits appelés R0,…,R7. Il possède aussi quelques registres spécifiques dont l'utilisation est abordée plus tard. Le registre PSR (Program Status Register) regroupe plusieurs indicateurs binaires dont l'indicateur de mode (mode utilisateur ou mode privilégié), les indicateurs n, z et p qui sont testés par les branchements conditionnels ainsi que le niveau de priorité des interruptions. Les registres USP (User Stack Pointer) et SSP (System Stack Pointer) permettent de sauvegarder le registre R6 suivant que le programme est en mode privilégié ou non. Comme tous les micro-processeurs, le LC-3 dispose d'un compteur de programme PC et d'un registre d'instruction IR qui sont tous les deux des registres 16 bits.



Registres du LC-3

10.1.1 Registre PSR

Le registre PSR regroupe plusieurs informations différentes. Le bit de numéro 15 indique si le processeur est en mode utilisateur (0) ou privilégié (1). Les trois bits de numéros 10 à 8 donnent la priorité d'exécution. Les trois bits de numéros 2 à 0 contiennent respectivement les indicateurs n, z et p.



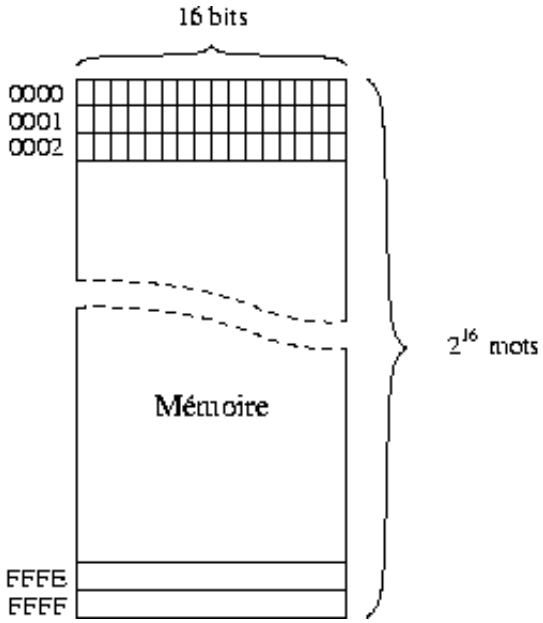
Registre PSR

10.2 Indicateurs N, Z et P

Les *indicateurs* sont des registres 1 bit. Les trois indicateurs n, z et p font, en fait, partie du registre spécial PSR. Ils sont positionnés dès qu'une nouvelle valeur est mise dans un des registres généraux R0,…,R7. Ceci a lieu lors de l'exécution d'une instruction logique (`NOT` et `AND`), arithmétique (`ADD`) ou d'une instruction de chargement (`LD`, `LDI`, `LDR` et `LEA`). Ils indiquent respectivement si cette nouvelle valeur est négative (n), nulle (z) et positive (p). Ces indicateurs sont utilisés par l'instruction de branchement conditionnel `BR` pour savoir si le branchement doit être pris ou non.

10.3 Mémoire

La mémoire du LC-3 est organisée par mots de 16 bits. L'adressage du LC-3 est également de 16 bits. La mémoire du LC-3 est donc formée de 2^{16} mots de 16 bits, c'est-à-dire 128 KiB avec des adresses de 0000 à FFFF en hexadécimal. Cette organisation de la mémoire est inhabituelle mais elle simplifie le câblage. La mémoire de la plupart des micro-processeurs est organisée par octets (mots de 8 bits). Par contre, ces micro-processeurs ont la possibilité de charger directement 2, 4 ou 8 octets. Pour certains, ce bloc doit être aligné sur une adresse multiple de 2, 4 ou 8 alors que ce n'est pas nécessaire pour d'autres.



Mémoire du LC-3

10.4 Instructions

Les instructions du LC-3 se répartissent en trois familles : les instructions arithmétiques et logiques, les instructions de chargement et rangement et les instructions de branchement (appelées aussi *instructions de saut* ou encore *instructions de contrôle*).

Les instructions arithmétiques et logiques du LC-3 sont au nombre de trois. Le LC-3 possède une instruction `ADD` pour l'addition, une instruction `AND` pour le *et logique* bit à bit et une instruction `NOT` pour la négation bit à bit. Le résultat de ces trois opérations est toujours placé dans un registre. Les deux opérandes peuvent être soit les contenus de deux registres soit le contenu d'un registre et une constante pour `ADD` et `AND`.

Les instructions de chargement et rangement comprennent des instructions (avec des noms commencent par `LD`) permettant de charger un registre avec une valeur et des instructions (avec des noms commencent par `ST`) permettant de ranger en mémoire le contenu d'un registre. Ces instructions se différencient par leurs *modes d'adressage* qui peut être immédiat, direct, indirect ou relatif. Les instructions de chargement sont `LD`, `LDI`, `LDR`, `LEA` et les instructions de rangement sont `ST`, `STI` et `STR`.

Les instructions de branchement comprennent les deux instructions de saut `BR` et `JMP`, les deux instructions d'appel de sous-routine `JSR` et `JSRR`, une instruction `TRAP` d'appel système, une instruction `RET` de retour de sous-routine et une instruction `RTI` de retour d'interruption.

10.4.1 Description des instructions

10.4.1.1 Instructions arithmétiques et logiques

Ces instructions du LC-3 n'utilisent que les registres aussi bien pour les sources que pour la destination. Ceci est une caractéristique des architectures RISC. Le nombre d'instructions arithmétiques et logiques du LC-3 est réduit au strict nécessaire. Les micro-processeurs réels possèdent aussi des instructions pour les autres opérations arithmétiques (soustraction, multiplication, division) et les autres opérations logiques (*ou logique*, ou exclusif). Ils possèdent aussi des instructions pour l'addition avec retenue, les décalages.

10.4.1.1.1 Instruction `NOT`

L'instruction `NOT` permet de faire le *non logique* bit à bit d'une valeur 16 bits. Sa syntaxe est `NOT DR,SR` où `DR` et `SR` sont les registres destination et source.

10.4.1.1.2 Instruction `ADD`

L'instruction `ADD` permet de faire l'addition de deux valeurs 16 bits. Elle convient pour les additions pour les nombres signés ou non puisque les nombres sont représentés en complément à 2. Elle a deux formes différentes. Dans la première forme, les deux valeurs sont les contenus de deux registres généraux. Dans la seconde forme, la première valeur est le contenu d'un registre et la seconde est une constante (adressage immédiat). Dans les deux formes, le résultat est rangé dans un registre.

La première forme a la syntaxe `ADD DR,SR1,SR2` où `DR` est le *registre destination* où est rangé le résultat et `SR1` et `SR2` sont les *registres sources* d'où proviennent les deux valeurs. La seconde forme a la syntaxe `ADD DR,SR1,Imm5` où `DR` et `SR1` sont encore les registres destination et source et `Imm5` est une constante codée sur 5 bits ($-16 \leq Imm5 \leq 15$). Avant d'effecteur l'opération, la constante `Imm5` est étendue de façon signée sur 16 bits en recopiant le bit 4 sur les bits 5 à 15.

10.4.1.1.3 Instruction `AND`

L'instruction `AND` permet de faire le *et logique* bit à bit de deux valeurs 16 bits. Elle a deux formes similaires à celles de l'instruction `ADD` de syntaxes `AND DR,SR1,SR2` et `AND DR,SR1,Imm5`.

10.4.1.2 Instructions de chargement et rangement

Les instructions de chargement permettent de charger un des registres généraux avec un mot en mémoire alors que les instructions de rangement permettent de ranger en mémoire le contenu d'un de ces registres. Ce sont les seules instructions faisant des accès à la mémoire. Ces différentes instructions se différencient par leur mode d'adressage. Un *mode d'adressage* spécifie la façon dont l'adresse mémoire est calculée. Le micro-processeur LC-3 possède les principaux modes d'adressage qui sont les modes d'adressage immédiat, direct, relatif et indirect. La terminologie pour les modes d'adressage n'est pas fixe car chaque constructeur a introduit ses propres termes. Le même mode peut avoir des noms différents chez deux constructeurs et le même nom utilisé par deux constructeurs peut recouvrir des modes différents.

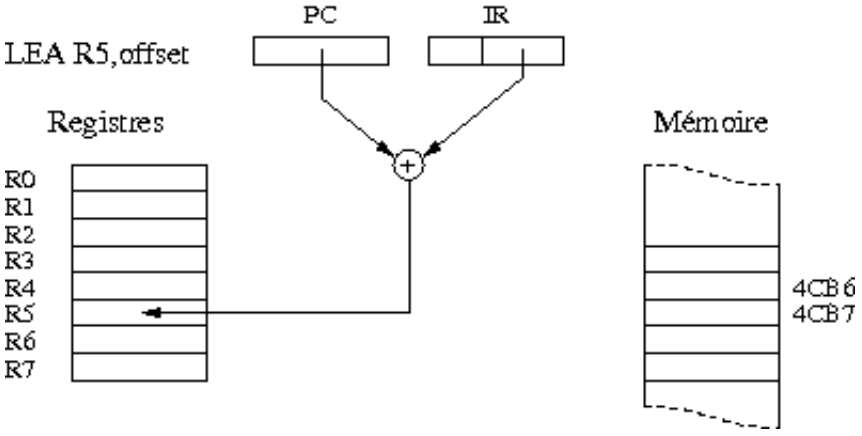
Il faut faire attention au fait que presque tous les modes d'adressage du LC-3 sont relatifs au compteur de programme. Cette particularité est là pour compenser la petite taille des offsets qui permettent un adressage à un espace réduit.

Exceptée l'instruction `LEA`, les instructions de chargement et rangement vont par paire. Pour chaque mode d'adressage, il y a une instruction de chargement dont le mnémonique commence par `LD` pour *LoaD* et une instruction de rangement dont le mnémonique commence par `ST` pour *STore*.

Toutes les instructions de chargement et rangement contiennent un offset. Cet offset n'est en général pas donné explicitement par le programmeur. Celui-ci utilise des étiquettes et l'assembleur se charge de calculer les offsets.

10.4.1.2.1 Instruction `LEA`

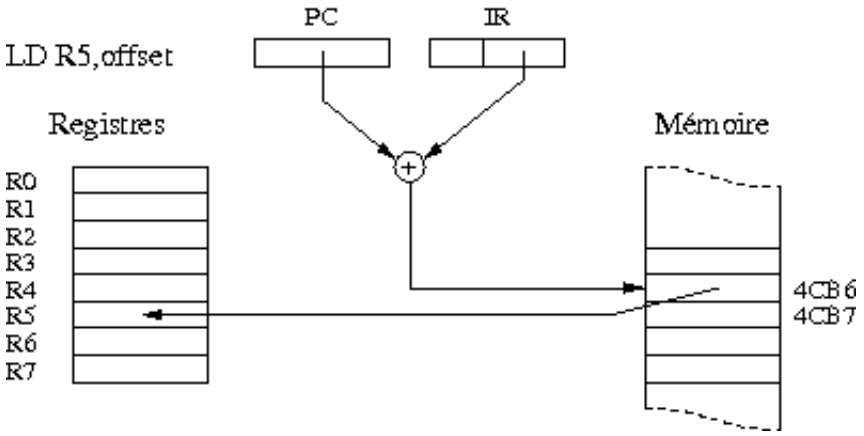
L'instruction `LEA` pour *Load Effective Address* charge dans un des registres généraux la somme du compteur de programme et d'un offset codé dans l'instruction. Ce mode adressage est généralement appelé adressage *absolu* ou *immédiat* car la valeur chargée dans le registre est directement contenue dans l'instruction. Aucun accès supplémentaire à la mémoire n'est nécessaire. Cette instruction est par exemple utilisée pour charger dans un registre l'adresse d'un tableau.



Adressage immédiat

10.4.1.2.2 Instructions LD et ST

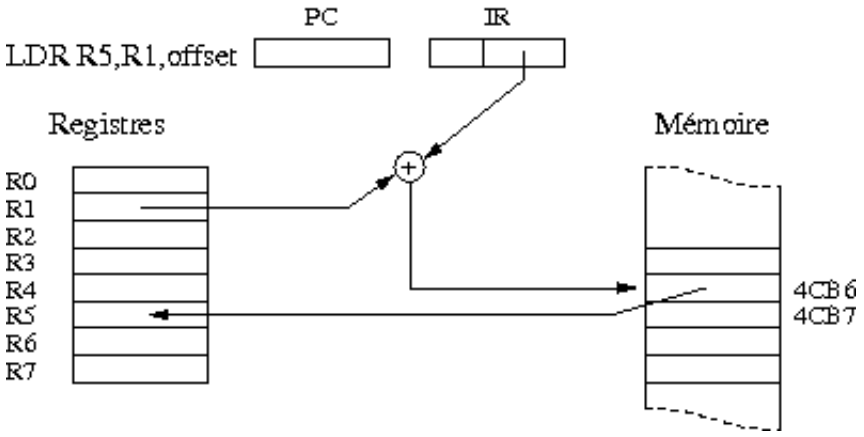
Les instructions LD et SD sont les instructions générales de chargement et rangement. L'instruction LD charge dans un des registres généraux le mot mémoire à l'adresse égale à la somme du compteur de programme et d'un offset codé dans l'instruction. L'instruction ST range le contenu du registre à cette même adresse. Ce mode adressage est généralement appelé adressage *direct*. Il nécessite un seul accès à la mémoire.



Adressage direct

10.4.1.2.3 Instructions LDR et STR

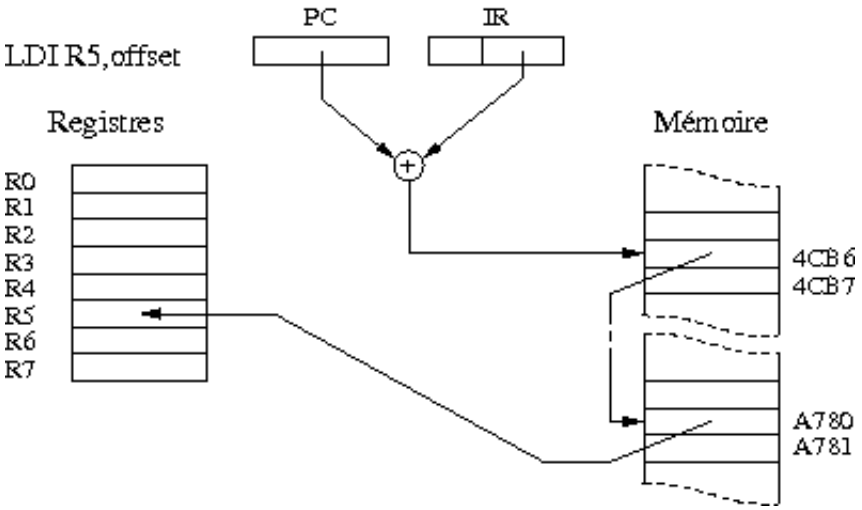
L'instruction LDR charge dans un des registres généraux la mot mémoire à l'adresse égale à la somme du registre de base et d'un offset codé dans l'instruction. L'instruction STR range le contenu du registre à cette même adresse. Ce mode adressage est généralement appelé adressage *relatif* ou *basé*. Il nécessite un seul accès à la mémoire. La première utilité de ces instructions est de manipuler les objets sur la pile comme par exemple les variables locales des fonctions. Elles servent aussi à accéder aux champs des structures de données. Le registre de base pointe sur le début de la structure et l'offset du champ dans la structure est codé dans l'instruction.



Adressage relatif

10.4.1.2.4 Instructions `LDI` et `STI`

Les instructions `LDI` et `STI` sont les instructions les plus complexes de chargement et rangement. Elle mettent en œuvre une double indirection. Elles calculent, comme les instructions `LD` et `SD`, la somme du compteur de programme et de l'offset. Elle chargent la valeur contenue à cette adresse puis utilisent cette valeur à nouveau comme adresse pour charger ou ranger le registre. Ce mode adressage est généralement appelé adressage *indirect*. Il nécessite deux accès à la mémoire.



Adressage indirect

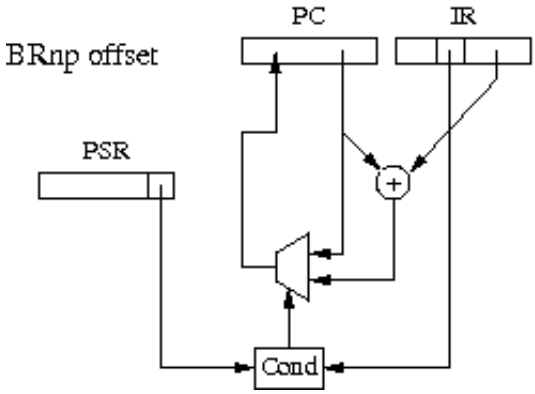
10.4.1.3 Instructions de branchements

10.4.1.3.1 Instruction `BR`

L'instruction générale de branchement est l'instruction `BR`. Elle modifie le déroulement normal des instructions en changeant la valeur contenue par le registre PC. Cette nouvelle valeur est obtenue en ajoutant à la valeur de PC un offset codé sur 9 bits dans l'instruction.

Le branchement peut être inconditionnel ou conditionnel. Le cas *conditionnel* signifie que le branchement est réellement exécuté seulement si une condition est vérifiée. Certains des trois indicateurs n, z, et p sont examinés. Si au moins un des indicateurs examinés vaut 1, le branchement est pris. Sinon, le branchement n'est pas pris et le déroulement du programme continue avec l'instruction suivante.

On rappelle que les trois indicateurs n, z et p sont mis à jour dès qu'une nouvelle valeur est chargée dans un des registres généraux. Ceci peut être réalisé par une instruction arithmétique ou logique ou par une instruction de chargement. Les indicateurs qui doivent être pris en compte par l'instruction `BR` sont ajoutés au mnémonique `BR` pour former un nouveau mnémonique. Ainsi l'instruction `BRnp` exécute le branchement si l'indicateur n ou l'indicateur p vaut 1, c'est-à-dire si l'indicateur z vaut 0. Le branchement est alors pris si la dernière valeur chargée dans un registre est non nulle.



Branchement conditionnel

Les programmes sont écrits en langage d'assembleur qui autorise l'utilisation d'étiquettes (*labels* en anglais) qui évitent au programmeur de spécifier explicitement les offsets des branchements. Le programmeur écrit simplement une instruction `BR label` et l'assembleur se charge de calculer l'offset, c'est-à-dire la différence entre l'adresse de l'instruction de branchement et l'adresse désignée par l'étiquette `label`.

Comme l'offset est codé sur 9 bits, l'instruction `BR` peut uniquement atteindre une adresse située de -255 à 256 mots mémoire. Le décalage d'une unité est dû au fait que le calcul de la nouvelle adresse est effectué après l'incrémentation du compteur de programme qui a lieu lors de la phase de chargement de l'instruction.

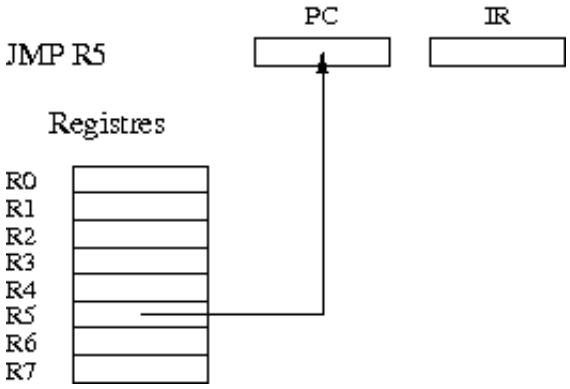
10.4.1.3.2 Instruction `JMP`

L'instruction `JMP` permet de charger le compteur de programme PC avec le contenu d'un des registres généraux. L'intérêt de cette instruction est multiple.

1. L'instruction `JMP` compense la faiblesse de l'instruction `BR` qui peut uniquement effectuer un branchement proche. L'instruction `JMP` permet de sauter à n'importe quelle adresse. Si l'étiquette `label` désigne une adresse trop éloignée, le branchement à cette adresse peut être effectuée par le code suivant.

```
        LD R7,labelp
        JMP R7
labelp: .FILL label
        ...
label:  ...
```

2. L'instruction `JMP` est indispensable pour tout les langages de programmation qui manipulent explicitement comme C et C++ ou implicitement des pointeurs sur des fonctions. Tous les langages de programmation objet manipulent des pointeurs sur des fonctions dans la mesure où chaque objet a une table de ses méthodes.
3. L'instruction `JMP` permet aussi les retours de sous-routines puisque l'instruction `RET` est en fait une abréviation pour l'instruction `JMP R7`.



Branchement par registre

10.4.2 Récapitulatif des Instructions

La table suivante donne une description concise de chacune des instructions. Cette description comprend la syntaxe, l'action ainsi que le codage de l'instruction. La colonne *nzp* indique par une étoile * si les indicateurs n, z et p sont affectés par l'instruction.

Syntaxe	Action	nzp	Codage															
			Op-code				Arguments											
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOT DR,SR	DR ← not SR	*	1	0	0	1	DR			SR			1	1	1	1	1	1
ADD DR,SR1,SR2	DR ← SR1 + SR2	*	0	0	0	1	DR			SR1			0	0	0	SR2		
ADD DR,SR1,Imm5	DR ← SR1 + SEXT(Imm5)	*	0	0	0	1	DR			SR1			1	Imm5				
AND DR,SR1,SR2	DR ← SR1 and SR2	*	0	1	0	1	DR			SR1			0	0	0	SR2		
AND DR,SR1,Imm5	DR ← SR1 and SEXT(Imm5)	*	0	1	0	1	DR			SR1			1	Imm5				
LEA DR,label	DR ← PC + SEXT(PCoffset9)	*	1	1	1	0	DR			PCoffset9								
LD DR,label	DR ← mem[PC + SEXT(PCoffset9)]	*	0	0	1	0	DR			PCoffset9								
ST SR,label	mem[PC + SEXT(PCoffset9)] ← SR		0	0	1	1	SR			PCoffset9								
LDR DR,BaseR,Offset6	DR ← mem[BaseR + SEXT(Offset6)]	*	0	1	1	0	DR			BaseR			Offset6					
STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] ← SR		0	1	1	1	SR			BaseR			Offset6					
LDI DR,label	DR ← mem[mem[PC + SEXT(PCoffset9)]]	*	1	0	1	0	DR			PCoffset9								
STI SR,label	mem[mem[PC + SEXT(PCoffset9)]] ← SR		1	0	1	1	SR			PCoffset9								
BR[n][z][p] label	Si (cond) PC ← PC + SEXT(PCoffset9)		0	0	0	0	n		z	p	PCoffset9							
NOP	No Operation		0	0	0	0	0		0	0	0	0	0	0	0	0	0	0
JMP BaseR	PC ← BaseR		1	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RET (≡ JMP R7)	PC ← R7		1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
JSR label	R7 ← PC; PC ← PC + SEXT(PCoffset11)		0	1	0	0	1	PCoffset11										
JSRR BaseR	R7 ← PC; PC ← BaseR		0	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RTI	cf. interruptions		1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TRAP Trapvect8	R7 ← PC; PC ← mem[Trapvect8]		1	1	1	1	0	0	0	0	Trapvect8							
Réservé			1	1	0	1												

10.5 Codage des instructions

Chaque instruction est représentée en mémoire par un code. Il est important que le codage des instructions soit réfléchi et régulier pour simplifier les circuits de décodage.

Toutes les instructions du LC-3 sont codées sur un mot de 16 bits. Les quatre premiers bits contiennent l'*op-code* qui détermine l'instruction. Les 12 bits restant codent les paramètres de l'instruction qui peuvent être des numéros de registres ou des constantes.

À titre d'exemple, les codages des deux instructions `ADD` et `BR` sont détaillés ci-dessous.

10.5.1 Codage de `ADD`

L'instruction `ADD` peut prendre deux formes. Une première forme est `ADD DR,SR1,SR2` où `DR`, `SR1` et `SR2` sont les trois registres destination, source 1 et source 2. Une seconde forme est `ADD DR,SR1,Imm5` où `DR` et `SR1` sont deux registres destination et source et `Imm5` est une constante signée sur 5 bits.

Le codage de l'instruction `ADD` est le suivant. Les quatre premiers bits de numéros 15 à 12 contiennent l'op-code 0001 de l'instruction `ADD`. Les trois bits suivants de numéros 11 à 9 contiennent le numéro du registre destination `DR`. Les trois bits suivants de numéros 8 à 6 contiennent le numéro du registre source `SR1`. Les six derniers bits contiennent le codage du troisième paramètre `SR2` ou `Imm5`. Le premier de ces six bits de numéro 5 permet de distinguer entre les deux formes de l'instruction. Il vaut 0 pour la première forme et 1 pour la seconde forme. Dans la première forme, les bits de numéros 2 à 0 contiennent le numéro du registre `SR2` et les bits de numéros 4 et 3 sont inutilisés et forcés à 0. Dans la seconde forme, les cinq derniers bits de numéro 4 à 0 contiennent la constante `Imm5`.

Le tableau ci-dessous donne un exemple de codage pour chacune des formes de l'instruction `ADD`.

Instruction	**Codage**															**Code en hexa**
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	Op-code				DR			SR1			0	0	0	SR2			
ADD R2,R7,R5	0	0	0	1	0	1	0	1	1	1	0	0	0	1	0	1	0x16C5
	Op-code				DR			SR1			1	Imm5					
ADD R6,R6,-1	0	0	0	1	1	1	0	1	1	0	1	1	1	1	1	1	0x1DBF

10.5.2 Codage de `BR`

Le codage de l'instruction `BR` est le suivant. Les quatre premiers bits de numéros 15 à 12 contiennent l'op-code 0000 de l'instruction `BR`. Les trois bits suivants de numéros 11 à 9 déterminent respectivement si l'indicateur n, z et p est pris en compte dans la condition. Si ces trois bits sont b_{11}, b_{10} et b_9, la condition *cond* est donnée par la formule suivante.

$$cond = (b_{11} \wedge n) \vee (b_{10} \wedge z) \vee (b_9 \wedge p)$$

Les 9 derniers bits de numéros 8 à 0 codent l'offset du branchement.

Si les trois bits b_{11}, b_{10} et b_9 valent 1, la condition est toujours vérifiée puisque $n + z + p = 1$. Il s'agit alors de l'instruction de branchement inconditionnel `BR`. Si au contraire les trois bits b_{11}, b_{10} et b_9 valent 0, la condition n'est jamais vérifiée et le branchement n'est jamais pris. Il s'agit alors de l'instruction `NOP` qui ne fait rien. En prenant un offset égal à 0, le code de l'instruction `NOP` est 0x0000.

Le tableau ci-dessous donne un exemple de codage pour trois formes typiques de l'instruction `BR`. Dans les exemples ci-dessous, on a donné des valeurs explicites aux offsets afin d'expliquer leur codage mais cela ne correspond pas à la façon dont l'instruction `BR` est utilisée dans les programmes.

Instruction	Codage																Code en hexa
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	Op-code				n	z	p	PCoffset9									
BRnz -137	0	0	0	0	1	0	1	1	0	1	1	1	0	1	1	1	0x0B77
BR 137	0	0	0	0	1	1	1	0	1	0	0	0	1	0	0	1	0x0E89
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0x0000

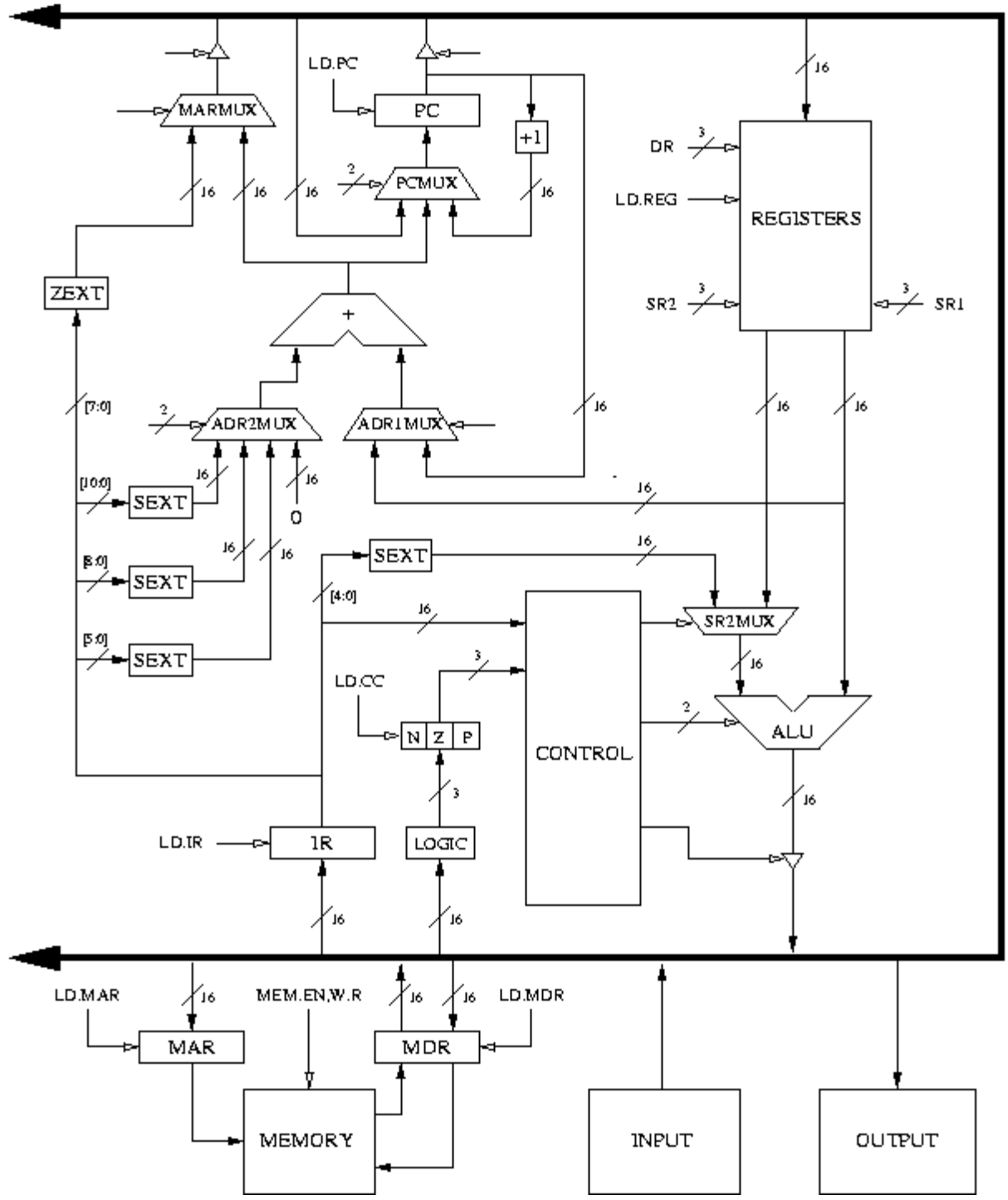
10.5.3 Répartition des op-code

On peut remarquer les *op-code* des instructions n'ont pas été affectés de manière aléatoire. On peut même observer une certaine régularité qui facilite le décodage. Les *op-code* des instructions de chargement et de rangement qui se correspondent (`LD` et `ST`, `LDR` et `STR`, …) ne différent que d'un seul bit.

MSB	LSB			
	0 0	0 1	1 0	1 1
0 0	BR	ADD	LD	ST
0 1	JSR(R)	AND	LDR	STR
1 0	RTI	NOT	LDI	STI
1 1	JMP		LEA	TRAP

10.6 Schéma interne du LC-3

Le schéma ci-dessous donne une réalisation du processeur LC-3 avec les différents éléments : registres, ALU, unité de contrôle, …. Les détails des entrées/sorties sont donnés dans un schéma ultérieur.



Chemins de données du LC-3

11 Programmation du LC-3

11.1 Programmation en assembleur

La programmation en langage machine est rapidement fastidieuse. Il est inhumain de se souvenir du codage de chacune des instructions d'un micro-processeur même simplifié à l'extrême comme le LC-3 et *a fortiori* encore plus pour un micro-processeur réel. Le *langage d'assemblage* ou *assembleur* par abus de langage rend la tâche beaucoup plus facile même si cette programmation reste longue et technique.

L'assembleur est un programme qui autorise le programmeur à écrire les instructions du micro-processeur sous forme symbolique. Il se charge de traduire cette écriture symbolique en codage hexadécimal ou binaire. Il est beaucoup plus aisé d'écrire `ADD R6,R1,R2` plutôt que 0x1C42 en hexadécimal ou encore 0001 1100 0100 0010 en binaire.

L'assembleur permet aussi l'utilisation d'*étiquettes symboliques* pour désigner certaines adresses. L'étiquette est associée à une adresse et peut alors être utilisée dans d'autres instructions. On peut par exemple écrire un morceau de code comme ci-dessous.

```
        ...
label:  ADD R6,R1,R2
        ...
        BRz label
        ...
```

L'étiquette *label* est alors associée à l'adresse ou se trouvera l'instruction `ADD R6,R1,R2` lors de l'exécution du programme. Cette adresse est alors utilisée pour calculer le codage de l'instruction `BRz label`. Comme le codage de l'instruction `BR` contient un offset, l'assembleur calcule la différence entre les adresses des deux instructions `ADD R6,R1,R2` et `BRz label` et place cette valeur dans les 9 bits de poids faible du codage de `BRz`.

11.1.1 Syntaxe

La syntaxe des programmes en assembleur est la même pour tous les assembleurs à quelques détails près.

Les instructions d'un programme en assembleur sont mises dans un fichier dont le nom se termine par l'extension `.asm`. Chaque ligne du fichier contient au plus une instruction du programme. Chaque ligne est en fait divisée en deux champs qui peuvent chacun ne rien contenir. Le premier champ contient une étiquette et le second champ une instruction. L'instruction est formée de son nom symbolique appelé *mnémonique* et d'éventuel paramètres.

Les espaces sont uniquement utiles comme séparateurs. Plusieurs espaces sont équivalents à un seul. Les programmes sont indentés de telle manière que les étiquettes se trouvent dans une première colonne et les instructions dans une seconde colonne.

L'instruction peut être remplacée dans une ligne par une directive d'assemblage. Les directives d'assemblage permettent de donner des ordres à l'assembleur. Elles sont l'équivalent des directives commençant par `'#'` comme `#include` ou `#if` du langage C.

11.1.1.1 Constantes

Il existe deux types de constantes : les constantes entières et les chaînes de caractères. Les chaînes de caractères peuvent uniquement apparaître après la directive `.STRINGZ`. Elles sont alors délimitées par deux caractères `'"'` comme dans `"Exemple de chaine"` et sont implicitement terminées par le caractère nul `'\0'`.

Les constantes entières peuvent apparaître comme paramètre des instructions du LC3 et des directives `.ORIG`, `.FILL` et `.BLKW`. Elles peuvent être données soit sous la forme d'un entier écrit dans la base 10 ou 16, soit sous la forme d'un caractère. Les constantes écrite en base 16 sont préfixées par le caractère `x`. Lorsque la constante est donnée sous forme d'un caractère, sa valeur est le code ASCII du caractère. Le caractère est alors entouré de caractères `''` comme dans `'Z'`.

```
; Exemple de constantes
        .ORIG x3000     ; Constante entière en base 16
        AND R2,R1,2     ; Constante entière en base 10
        ADD R6,R5,-1    ; Constante entière négative en base 10
        .FILL 'Z'       ; Constante sous forme d'un caractère
        .STRINGZ "Chaine"  ; Constante chaîne de caractères
        .END
```

11.1.1.2 Commentaires

Les commentaires sont introduits par le caractère point-virgule `';'` et il s'étendent jusqu'à la fin de la ligne. Ils sont mis après les deux champs d'étiquette et d'instruction mais comme ces deux champs peuvent être vides, les commentaires peuvent commencer dès le début de la ligne ou après l'étiquette. Voici quelques exemples de commentaires

```
; Commentaire dès le début de la ligne
label:  ADD R6,R1,R2    ; Commentaire après l'instruction

loop:                   ; Commentaire après l'étiquette (avec indentation)
        BRz label
```

11.1.2 Directives d'assemblage

`.ORIG` ⟨adresse⟩
: Cette directive est suivi d'une adresse constante. Elle spécifie l'adresse à laquelle doit commencer le bloc d'instructions qui suit. Tout bloc d'instructions doit donc commencer par une directive `.ORIG`.

`.END`
: Cette directive termine un bloc d'instructions.

`.FILL` ⟨valeur⟩
: Cette directive réserve un mot de 16 bits et le remplit avec la valeur constante donnée en paramètre.

`.STRINGZ` ⟨chaine⟩
: Cette directive réserve un nombre de mots de 16 bits égal à la longueur de la chaîne de caractères terminée par un caractère nul et y place la chaîne. Chaque caractère de la chaîne occupe un mot de 16 bits.

`.BLKW` ⟨nombre⟩
: Cette directive réserve le nombre de mots de 16 bits passé en paramètre.

11.1.3 Étiquettes (labels)

Les étiquettes sont des identificateurs formés de caractères alphanumériques et commençant par une lettre. Elles désignent une adresse qui peut alors être utilisée dans les instructions.

L'adresse est spécifiée en mettant l'étiquette suivie du caractère `':'` dans la première colonne d'une ligne du programme. L'étiquette est alors affectée de l'adresse à laquelle se trouve l'instruction. L'étiquette peut également être mise avant une directive d'assemblage comme .BLKW, `.FILL` ou `.STRINGZ`. Elle désigne alors l'adresse à laquelle sont placés les mots mémoires produits par la directive.

```
; Exemple d'étiquettes
label:  ADD R0,R1,R2    ; 'label' désigne l'adresse de l'instruction ADD
char:   .FILL 'Z'       ; 'char'  désigne l'adresse du caractère 'Z'
string: .STRINGZ "Chaine"  ; 'char' désigne l'adresse du premier caractère 'C'
```

11.2 Exemples de programmes

On étudie dans cette partie quelques exemples de programmes (très) simples afin d'illustrer quelques techniques classiques de programmation en langage machine. Les codes complets de ces programmes ainsi que d'autres sont disponibles ci-dessous

- Longueur d'une chaîne de caractères
- Tours de hanoï
- Multiplication non signée
- Multiplication signée
- Multiplication logarithmique
- Calcul du logarithme en base 2

Les programmes ci-dessous sont présentés comme des fragments de programmes puisque les routines n'ont pas encore été présentées. Il faudrait les écrire sous forme de routines pour une réelle utilisation.

11.2.1 Longueur d'une chaîne

On commence par rappeler le programme en C pour calculer la longueur d'une chaîne de caractères terminée par `'\0'`.

```
int strlen(char* p) {
  int c = 0;
  while (*p != '\0') {
    p++;
    c++;
  }
  return c;
}
```

Le programme ci-dessous calcule la longueur d'une chaîne de caractères terminée par le caractère nul `'\0'`. Le registre R0 est utilisé comme pointeur pour parcourir la chaîne alors que le registre R1 est utilisé comme compteur.

```
; @param R0 adresse de la chaîne
; @return R1 longueur de la chaîne
        .ORIG x3000
        LEA R0,chaine   ; Chargement dans RO de l'adresse de la chaîne
        AND R1,R1,0     ; Mise à 0 du compteur : c = 0
loop:   LDR R2,R0,0     ; Chargement dans R2 du caractère pointé par R0
        BRz fini        ; Test de fin de chaîne
        ADD R0,R0,1     ; Incrémentation du pointeur : p++
        ADD R1,R1,1     ; Incrémentation du compteur : c++
        BR loop
fini:   NOP
chaine: .STRINGZ "Hello World"
        .END
```

Le programme C pour calculer la longueur d'une chaîne de caractères peut être écrit de manière différente en utilisant l'arithmétique sur les pointeurs.

```
int strlen(char* p) {
  char* q = p;
  while (*p != '\0')
    p++;
  return p-q;
}
```

Ce programme C se transpose également en langage d'assembleur. Le registre R0 est encore utilisé comme pointeur pour parcourir la chaîne et le registre R1 sauvegarde la valeur initiale de R0. Il contient en fait l'opposé de la valeur initiale de R0 afin de calculer la différence. Ce programme a le léger avantage d'être plus rapide que le précédent car la boucle principale contient une instruction de moins.

```
; @param R0 adresse de la chaîne
; @return R0 longueur de la chaîne
        .ORIG x3000
        LEA R0,chaine   ; Chargement dans R0 de l'adresse de la chaîne
        NOT R1,R0       ; R1 = -R0
        ADD R1,R1,1
loop:   LDR R2,R0,0     ; Chargement dans R2 du caractère pointé par R0
        BRz fini        ; Test de fin de chaîne
        ADD R0,R0,1     ; Incrémentation du pointeur : p++
        BR loop
fini:   ADD R0,R0,R1    ; Calcul de la différence q-p
chaine: .STRINGZ "Hello World"
        .END
```

11.2.2 Nombre d'occurrences d'un caractère dans une chaîne

Le programme ci-dessous calcule le nombre d'occurrences d'un caractère donné dans une chaîne de caractères terminée par le caractère nul `'\0'`. Le registre R0 est utilisé comme pointeur pour parcourir la chaîne. Le registre R1 contient le caractère recherché et le registre R2 est utilisé comme compteur. La comparaison entre chaque caractère de la chaîne et le caractère recherché est effectuée en calculant la différence de leurs codes et en testant si celle-ci est nulle. Dans ce but, le programme commence par calculer l'opposé du code du caractère recherché afin de calculer chaque différence par une addition.

```
; @param R0 adresse de la chaîne
; @param R1 caractère
; @return R2 nombre d'occurrences du caractère
        .ORIG x3000
        LEA R0,chaine   ; Chargement dans RO de l'adresse de la chaîne
        LD R1,caract    ; Chargement dans R1 du code ASCII de 'l'
        AND R2,R2,0     ; Mise à 0 du compteur
        NOT R1,R1       ; Calcul de l'opposé de R1
        ADD R1,R1,1     ; R1 = -R1
loop:   LDR R3,R0,0     ; Chargement dans R3 du caractère pointé par R0
        BRz fini        ; Test de fin de chaîne
        ADD R3,R3,R1    ; Comparaison avec 'l'
        BRnp suite      ; Non égalité
        ADD R2,R2,1     ; Incrémentation du compteur
suite:  ADD R0,R0,1     ; Incrémentation du pointeur
        BR loop
fini:   NOP
chaine: .STRINGZ "Hello World"
caract: .FILL 'l'
        .END
```

11.2.3 Multiplication

Tous le micro-processeurs actuels possèdent une multiplication câblée pour les entiers et pour les nombres flottants. Comme les premiers micro-processeurs, le LC-3 n'a pas de telle instruction. La multiplication doit donc être réalisée par programme. Les programmes ci-dessous donnent quelques implémentation de la multiplication pour les entiers signés et non signés.

11.2.3.1 Multiplication naïve non signée

On commence par une implémentation très naïve de la multiplication qui est effectuée par additions successives. On commence par donner une implémentation sous forme de programme C puis on donne ensuite une traduction en langage d'assembleur.

```
// Programme sous forme d'une fonction en C
int mult(int x, int n) {
  int r = 0;            // Résultat
  while(n != 0) {
    r += x;
    n--;
  }
  return r;
}
```

Dans le programme ci-dessous, les registres R0 et R1 contiennent respectivement x et n. Le registre R2 contient les valeurs successives de r.

```
; @param R0 x
; @param R1 n
; @return R2 x × n
        .ORIG x3000
        AND R2,R2,0     ; r = 0
        AND R1,R1,R1    ; Mise à jour de l'indicateur z pour le test
        BRz fini
loop:   ADD R2,R2,R0    ; r += x
        ADD R1,R1,-1    ; n--
        BRnp loop       ; Boucle si n != 0
fini:   NOP
        .END
```

11.2.3.2 Multiplication naïve signée

Le programme précédent fonctionne encore si la valeur de x est négative. Par contre, il ne fonctionne plus si la valeur de n est négative. Les décrémentations successives de n ne conduisent pas à 0 en le nombre d'étapes voulues. Le programme suivant résout ce problème en inversant les signes de x et n si n est négatif. Ensuite, il procède de manière identique au programme précédent.

```
; @param R0 x
; @param R1 n
; @return R2 x × n
        .ORIG x3000
        AND R2,R2,0     ; r = 0
        AND R1,R1,R1    ; Mise à jour de l'indicateur z pour le test
        BRz fini
        BRp loop
        ; Changement de signe de x et n
        NOT R0,R0       ; x = -x
        ADD R0,R0,1
        NOT R1,R1       ; n = -n
        ADD R1,R1,1
loop:   ADD R2,R2,R0    ; r += x
        ADD R1,R1,-1    ; n--
        BRnp loop       ; Boucle si n != 0
fini:   NOP
        .END
```

11.2.3.3 Multiplication logarithmique non signée

Le problème majeur des deux programmes précédents est leur temps d'exécution. En effet, le nombre d'itérations de la boucle est égal à la valeur de R1 et le temps est donc proportionnel à cette valeur. Le programme ci-dessous donne une meilleure implémentation de la multiplication de nombres non signés. Ce programme pourrait être étendu aux nombres signés de la même façon que pour l'implémentation naïve. Son temps d'exécution est proportionnel au nombre de bits des registres.

On commence par donner un algorithme récursif en C pour calculer une multiplication qui est inspiré de l'exponentiation logarithmique.

```
// Programme sous forme d'une fonction récursive en C
// Calcul de x * n de façon logarithmique
float mult(float x, int n) {
  if (n == 0)
    return 0;
  // Calcul anticipé de la valeur pour n/2 afin
  // d'avoir un seul appel récursif.
  float y = mult(x, n/2);
```

```
  if (n%2 == 0)
    return y + y;
  else
    return y + y + x;
}
```

Pour éviter d'écrire un programme récursif, on exprime le problème d'une autre façon. On suppose que l'écriture binaire du contenu de R1 est $b_{k-1} \ldots b_0$ où k est le nombre de bits de chaque registre, c'est-à-dire 16 pour le LC-3. On a alors la formule suivante qui exprime le produit $x \times n$ avec uniquement des multiplications par 2 et des additions. Cette formule est très semblable au schéma de Horner pour évaluer un polynôme.

$$x \times n = ((\ldots((xb_{k-1}2 + xb_{k-2})2 + xb_{k-3})2 + \ldots)2 + xb_1)2 + xb_0 .$$

On note $x_1, \ldots, x_k$ les résultats partiels obtenus en évaluant la formule ci-dessus de la gauche vers la droite. On pose $x_0 = 0$ et pour $1 \leq j \leq k$, le nombre x_j est donné par la formule suivante.

$$x_j = ((\ldots((xb_{k-1}2 + xb_{k-2})2 + xb_{k-3})2 + \ldots)2 + xb_{k-j+1})2 + xb_{k-j} .$$

Le nombre x_k est égal au produit $x \times n$. Il est de plus facile de calculer x_j en connaissant x_{j-1} et b_{k-j}. On a en effet les relations suivantes.

$$x_j = 2x_{j-1} \quad \text{si } b_{k-j} = 0$$
$$x_j = 2x_{j-1} + x \text{ si } b_{k-j} = 1$$

Le programme suivant calcule le produit $x \times n$ en partant de $x_0 = 0$ puis en calculant de proche en proche les nombres x_j grâce aux formules ci-dessus. Les registres R0 et R1 contiennent respectivement x et n. Le registre R2 contient les valeurs successives $x_0, \ldots, x_k$ et le registre R3 est utilisé comme compteur. Pour accéder aux différents bits $b_{k-1}, \ldots, b_0$ de n, on utilise le procédé suivant. Le signe de n vu comme un entier signé donne la valeur de b_{k-1}. Ensuite, l'entier n est décalé vers la gauche d'une position à chaque itération. Le signe des contenus successifs de R2 donne les valeurs $b_{k-2}, \ldots, b_0$.

```
; @param R0 x
; @param R1 n
; @return R2 x × n
        .ORIG x3000
        AND R2,R2,0     ; Initialisation x0 = 0
        LD R3,cst16     ; Initialisation du compteur
        AND R1,R1,R1
        BRzp bit0
bit1:   ADD R2,R2,R0    ; Addition de x si bk-j = 1
bit0:   ADD R3,R3,-1    ; Décrémentation du compteur
        BRz fini
        ADD R2,R2,R2    ; Calcul de 2xj-1
        ADD R1,R1,R1    ; Décalage de n vers la gauche
        BRn bit1
        BR bit0
fini:   NOP
cst16:  .FILL 16
        .END
```

11.2.4 Addition 32 bits

Dans la majorité des micro-processeurs réels, il y a outre les flags n, z et p un flag c qui reçoit la retenue (Carry) lors d'une addition par une instruction `ADD`. Cela permet de récupérer facilement cette retenue en testant le flag c. Certains micro-processeurs possèdent aussi une instruction `ADC` qui ajoute au résultat de l'addition la valeur du flag c. Ceci permet d'enchaîner facilement des additions pour faire des calculs sur des entiers codés sur plusieurs mots. Le micro-processeur LC-3 n'a pas de telles facilités et il faut donc exploiter au mieux ses possibilités.

```
; @param R1,R0 poids fort et poids faible du premier argument
; @param R3,R2 poids fort et poids faible du second argument
; @return R5,R4 poids fort et poids faible du résultat
        .ORIG x3000
        ; Addition des poids faibles
        ADD R4,R2,R0
        ; Addition des poids forts
        ADD R5,R3,R2
        ; Test s'il y a une retenue sur les poids faibles
        AND R0,R0,R0
        BRn bit1
        ; Cas où le bit de poids fort de R0 est 0
        AND R2,R2,R2
        ; Si les deux bits de poids fort de R1 et R0 sont 0,
        ; il n'y a pas de retenue.
        BRzp fini
        ; Si un seul des deux bits de poids fort de R1 et R0 est 1,
        ; il faut tester le bit de poids fort de R4 pour savoir
        ; s'il y a eu une retenue.
testR4: AND R4,R4,R4
        BRzp carry
        BR fini
        ; Cas où le bit de poids fort de R0 est 1
bit1:   AND R2,R2,R2
        ; Si le bit de poids fort de R1 est 0, on est ramené au cas
        ; un seul des deux bits de poids fort de R1 et R0 est 1.
        BRzp testR4
        ; Si les deux bits de poids fort de R1 et R0 sont 1,
        ; il y a nécessairement une retenue.
        ; Correction de la retenue
carry:  ADD R5,R5,1
fini:   NOP
        .END
```

11.2.5 Conversion en hexa du contenu d'un registre

```
; @param R0 valeur à convertir
        ADD R1,R0,0             ; R1 <- R0
        ld R2,cst4              ; Nombre de caractères
loopo:  ld R3,cst4              ; Nombre de bits par caractère
        ; Rotation de 4 bits
loopi:  AND R1,R1,R1
        BRn $1
        ADD R1,R1,R1
        BR $2
$1:     ADD R1,R1,R1
        ADD R1,R1,1
$2:     ADD R3,R3,-1
        BRp loopi
        ; Recupération de 4 bits
```

```
        AND R0,R1,0xf
        ADD R0,R0,-0xa
        BRn $3
        ; Chiffres de 0 à 9
        ld R3,cst40             ; 0x40 = '0' + 0xa
        BR $4
        ; Chiffres de A à F
$3:     ld R3,cst51             ; 0x51 = 'A' + 0xa
$4:     ADD R0,R0,R3
        TRAP 0x21               ; Affichage
        ADD R2,R2,-1
        BRp loopo
        NOP
        TRAP 0x25               ; Arrêt
cst4:   .fill 4
cst40:  .fill 0x40
cst51:  .fill 0x51
```

12 Les sous-routines et la pile

12.1 Instruction JMP

L'instruction JMP permet de transférer l'exécution du programme à une adresse contenue dans un des 8 registres R0,…,R7. Ce genre d'instruction est nécessaire à tout programme qui gère dynamiquement les appels de fonction. C'est le cas de tout programme en C qui manipule des pointeurs sur des fonctions. C'est aussi le cas des langages objets comme C++ dont les objets contiennent une table des méthodes sous forme de pointeurs de fonctions. Dans le cas du LC-3, cette instruction permet de compenser la faiblesse de l'instruction BR qui peut uniquement sauter à une adresse éloignée d'au plus 256 mots (l'offset est codé sur 9 bits en complément à 2). L'instruction JMP est aussi indispensable pour réaliser les retours de sous-routine comme nous allons le voir.

La réalisation d'une boucle se fait par exemple de la manière suivante.

```
        LD R0,nbiter    ; Nombre d'itérations de la boucle
; Boucle
loop:   ...             ; Début de la boucle
        ...
        ADD R0,R0,-1    ; Décrémentation du compteur
        BRp loop        ; Retour au début de la boucle
; Suite du programme
        ...
```

12.2 Sous-routines

12.2.1 Appel

Lorsqu'un programme appelle une sous-routine, il faut d'une certaine manière mémoriser l'adresse à laquelle doit revenir s'exécuter le programme à la fin de la routine. Dans le micro-processeur LC-3, l'adresse de retour, c'est-à-dire l'adresse de l'instruction suivante est mise dans le registre R7 lors d'un appel de sous-routine. Ce registre a donc un rôle particulier.

Il existe deux instructions JSR et JSRR permettant d'appeler une sous-routine. L'instruction JSR est semblable à l'instruction BR de branchement inconditionnel. Il n'y a pas de variante conditionnelle de l'instruction JSR. Les trois bits utilisés pour la condition dans le codage l'instruction BR sont récupérés pour avoir un offset de 11 bits au lieu de 9 bits pour BR. Un bit est aussi nécessaire pour distinguer JSR et JSRR qui utilisent le même op-code (cf. codage des instructions du LC-3). L'adresse de la sous-routine est stockée dans le code l'instruction sous forme d'un offset de 11 bits. L'instruction JSRR est semblable à l'instruction JMP. Elle permet d'appeler une sous-routine dont l'adresse est contenue dans un des 8 registres. Cette deux instructions ont en commun de transférer la valeur du compteur de programme PC incrémenté (adresse de l'instruction suivante) dans le registre R7 avant de charger PC avec l'adresse de la sous-routine.

12.2.2 Retour

Lors d'un appel à une sous-routine, l'adresse de retour est mise dans le registre R7. La sous-routine se termine donc par un saut à cette adresse avec l'instruction JMP R7. Cette instruction peut aussi être désignée par le mnémonique RET.

```
; Appel de la sous-routine sub
        ...
        JSR sub         ; Adresse de l'instruction suivante dans R7
        ...

; Sous-routine sub
sub:    ...
        ...
        RET             ; JMP R7
```

12.2.3 Exemple

On reprend le fragment de programme pour calculer la longueur d'une chaîne sous la forme d'une sous-routine appelée par un programme principal.

```
        .ORIG x3000
; Programme principal
        ...
        ; Premier appel
        LEA R0,chaine   ; Chargement dans R0 de l'adresse de la chaîne
        JSR strlen      ; Appel de la sous-routine
        ...
        ; Autre appel
        JSR strlen
        ...

; Chaine
chaine: .STRINGZ "Hello World"
        .END

; Sous-routine pour calculer la longueur d'une chaîne terminée par '\0'
; @param R0 adresse de la chaîne
; @return R0 longueur de la chaîne
; L'adresse de retour est dans R7
strlen: AND R1,R1,0     ; Mise à 0 du compteur : c = 0
loop:   LDR R2,R0,0     ; Chargement dans R2 du caractère pointé par R0
        BRz fini        ; Test de fin de chaîne
        ADD R0,R0,1     ; Incrémentation du pointeur : p++
        ADD R1,R1,1     ; Incrémentation du compteur : c++
        BR loop
fini:   ADD R0,R1,0     ; R0 = R1
        RET             ; Retour par JMP R7
```

La routine `strlen` utilise le registre R1 pour effectuer ses calculs. Si le programme principal utilise également ce registre R1, le contenu de ce registre doit être sauvegardé pendant l'appel à `strlen`. Cette question est abordée ci-dessous.

12.3 Sauvegarde des registres

Les registres du micro-processeur LC-3 sont en nombre limité. Lorsque qu'une routine a besoin de registres pour faire des calculs intermédiaires, il est préférable de sauvegarder le contenu de ces registres. Cela évite de détruire le contenu du registre qui est peut-être utilisé par le programme appelant la routine. En particulier si une routine doit appeler une autre sous-routine, elle doit préserver le registre R7 qui contient l'adresse de retour.

12.3.1 Registres

Une routine `sub` qui appelle une autre routine `subsub` peut utiliser un registre, par exemple R5 pour sauvegarder le registre R7 qui contient l'adresse de retour de `sub`. Cette méthode donne un code efficace car il n'utilise que les registres. Par contre, on arrive rapidement à cours de registres. Il faut donc utiliser d'autres techniques.

```
sub:    ADD R5,R7,0     ; Sauvegarde de R7 dans R5
        ...
        JSR subsub
        ...
        ADD R7,R5,0     ; Restauration de R7
        RET

subsub: ...

        RET
```

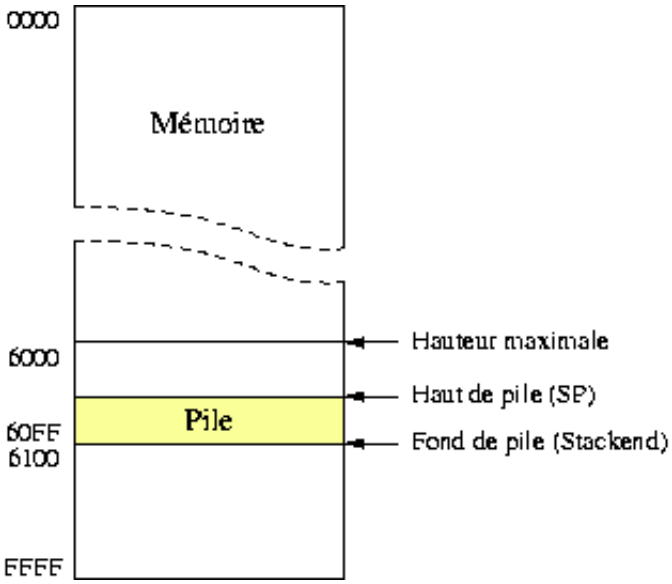
12.3.2 Emplacements mémoire réservés

Une autre méthode consiste à réserver des emplacements mémoire pour la sauvegarde des registres pendant l'exécution d'une routine. Les contenus des registres sont rangés dans ces emplacements au début de la routine et ils sont restaurés à la fin de celle-ci. Cette méthode a deux inconvénients majeurs. D'une part, elle nécessite de réserver de la mémoire pour chaque routine. Cela peut gaspiller beaucoup de mémoire si le programme est conséquent. De plus, l'espace réservé pour les routines non appelées est perdu. D'autre part, cette méthode conduit à du code qui n'est pas réentrant.

```
; Sous-routine sub sauvegardant R0 et R1
sub:    ST R0,saveR0    ; Sauvegarde de R0
        ST R1,saveR1    ; Sauvegarde de R1

        ...             ; Corps de la procédure

        LD R1,saveR1    ; Restauration de R1
        LD R0,saveR0    ; Restauration de R0
        RET
saveR0: .BLKW 1         ; Emplacement de sauvegarde de R0
saveR1: .BLKW 1         ; Emplacement de sauvegarde de R1

; Appel de la sous-routine
; Le registre R7 est détruit
        JSR sub
```

12.3.3 Utilisation d'une pile

La meilleure méthode est d'utiliser une pile. Cela à l'apparente à la méthode des emplacements réservés mais dans le cas d'une pile, c'est un espace global qui est partagé par toutes les sous-routines. Par contre, cette méthode nécessite de réserver un des registres à la gestion de la pile. Dans le cas du micro-processeur LC-3, c'est le registre R6 qui est généralement utilisé.



Pile en mémoire

12.3.3.1 Utilisation du registre R6

Le registre R6 est utilisé comme pointeur de pile. Pendant toute l'exécution du programme, le registre R6 donne l'adresse de la dernière valeur mise sur la pile. La pile croît vers les adresses décroissantes. N'importe quel autre registre (hormis R7) aurait pu être utilisé mais il est préférable d'utiliser R6 en cas d'interruption.

12.3.3.2 Empilement d'un registre

Un empilement est réalisé en déplaçant vers les adresses basses le pointeur de pile pour le faire pointer sur le premier emplacement libre. Ensuite le contenu du registre y est rangé en employant un rangement avec adressage relatif avec offset de 0.

```
ADD R6,R6,-1    ; Déplacement du haut de pile
STR Ri,R6,0     ; Rangement de la valeur
```

12.3.3.3 Dépilement d'un registre

Le dépilement est bien sûr l'opération inverse. Le contenu du registre est récupéré avec un chargement avec adressage relatif. Ensuite le pointeur de pile est incrémenté pour le faire pointer sur le haut de la pile.

```
LDR Ri,R6,0     ; Récupération de la valeur
ADD R6,R6,1     ; Restauration du haut de pile
```

Aucune vérification n'est faite pour savoir si la pile déborde. Il appartient au programmeur de vérifier que dans chaque routine, il effectue autant de dépilements que d'empilements.

12.3.3.4 Empilement ou dépilement de plusieurs registres

Lors de l'empilement ou du dépilement de plusieurs registres, l'offset de l'adressage relatif permet de faire en une seule instruction les différentes incrémentations ou décrémentations du registre de pile. L'empilement des registres R0, R1 et R2 (dans cet ordre peut par exemple être réalisé de la manière suivante.

```
ADD R6,R6,-3    ; Déplacement du haut de pile
STR R0,R6,2     ; Empilement de R0
STR R1,R6,1     ; Empilement de R1
STR R2,R6,0     ; Empilement de R2
```

Le fait de placer la décrémentation du registre de pile R6 avant de placer les contenus des registres R0, R1 et R2 sur la pile n'est pas anodin. Il semble à première vue que l'instruction `ADD R6,R6,-3` pourrait être placée après les trois instructions `STR` en changeant bien sûr les offsets utilisés par ces trois dernières par les valeurs -1, -2 et -3. Ceci est en fait faux. Si une interruption survient entre le placement des contenus des registres sur la pile et la décrémentation de R6, le contenu de la pile peut être altéré par de valeurs que l'interruption mettrait sur la pile. Le fait de décrémenter d'abord R6 peut être considéré comme une façon de réserver sur la pile les emplacements pour les contenus de R0, R1 et R2. Dans le cas du LC-3, les programmes utilisateur sont en quelque sorte protégés de ce type de problèmes car les interruptions utilisent la pile système qui est distincte de la pile utilisateur. Par contre, le code exécuté en mode privilégié doit respecter cette contrainte.

Le dépilement des registres R2, R1 et R0 se fait la même manière. L'ordre des trois dépilements effectués par les instructions `LDR` n'est pas imposé. Le résultat serait le même en les mettant dans un ordre différent. Par contre, on a respecté l'ordre inverse des empilements pour bien montrer qu'il s'agit de l'utilisation d'une pile.

```
LDR R0,R6,2     ; Dépilement de R0
LDR R1,R6,1     ; Dépilement de R1
LDR R2,R6,0     ; Dépilement de R2
ADD R6,R6,3     ; Restauration du haut de pile
```

La raison pour laquelle l'incrémentation de R6 est placée après le chargement des registres avec les valeurs sur la pile est identique à la raison pour laquelle la décrémentation de R6 est placée avant le placement des contenus des registres sur la pile.

La pile permet en particulier d'échanger les contenus de deux registres sans utiliser de registre supplémentaire autre que le pointeur de pile R6. Le morceau de code suivant échange par exemple les contenus des registres R1 et R2.

```
ADD R6,R6,-2    ; Déplacement du haut de pile
STR R1,R6,1     ; Empilement de R1
STR R2,R6,0     ; Empilement de R2
LDR R2,R6,1     ; Dépilement du contenu de R1 dans R2
LDR R1,R6,0     ; Dépilement du contenu de R2 dans R1
ADD R6,R6,2     ; Restauration du haut de pile
```

Si un micro-processeur possède une opération de ou exclusif bit à bit appelée `XOR`, l'échange des contenus de deux registres peut aussi se faire de la manière compliquée suivante.

```
                ; R1 : x        R2 : y
XOR R1,R1,R2    ; R1 : x ^ y    R2 : y
XOR R2,R1,R2    ; R1 : x ^ y    R2 : x
XOR R1,R1,R2    ; R1 : y        R2 : x
```

12.3.3.5 Appels de sous-routine imbriqués

Dans le cas d'appels de sous-routine imbriqués, c'est-à-dire d'une sous-routine `sub` appelant une (autre) sous-routine `subsub`, il est nécessaire de sauvegarder sur la pile l'adresse de retour de `sub` contenue dans le registre R7. À l'appel de la seconde sous-routine `subsub`, le registre R7 reçoit l'adresse de retour de celle-ci et son contenu précédent est écrasé.

```
sub:    ADD R6,R6,-2    ; Sauvegarde sur la pile de
        STR R7,R6,1     ; - l'adresse de retour
        STR R0,R6,0     ; - registre R0
        ...
        JSR subsub
        ...
        LDR R0,R6,0     ; Restauration de la pile de
        LDR R7,R6,1     ; - registre R0
        ADD R6,R6,2     ; - l'adresse de retour
        RET             ; Retour par JMP R7

subsub: ...             ; R7 contient l'adresse de retour
                        ; c'est-à-dire l'adresse de l'instruction
                        ; suivant JSR subsub

        RET             ; Retour par JMP R7
```

12.3.3.6 Initialisation de la pile

L'initialisation de la pile se fait en mettant dans le registre R6 l'adresse du haut de la pile. Il s'agit en fait de l'adresse du premier emplacement mémoire après l'espace réservé à la pile. Comme tout utilisation de la pile commence par décrémenter le registre R6, cet emplacement n'est normalement jamais utilisé par la pile.

```
; Initialisation de la pile au début du programme
psp:   .FILL stackend
main:   LD R6,psp       ; Équivalent à LEA R6,stackend
        ...


; Réservation de l'espace pour la pile
        .ORIG 0x6000
        .BLKW 0x100     ; Taille de la pile : 256 octets
stackend:               ; Adresse de mot mémoire suivant
```

12.4 Programmation

Pour illustrer l'utilisation de la pile, un programme récursif (source) calculant la solution du problème des *tours de Hanoï* est donné ci-dessous.

```
; Tours de Hanoï
        .ORIG x3000
; Programme principal
; Le pointeur de pile R6 est initialisé par le simulateur
hanoim: LD R0,nbrdisk   ; Nombre de disques
        LD R1,startst   ; Piquet de depart
        LD R2,endst     ; Piquet d'arrivée
        JSR hanoi
        TRAP x25        ; HALT
; Constantes
nbrdisk:.FILL 3         ; Nombre de disques
startst:.FILL 1         ; Piquet de depart
endst:  .FILL 2         ; Piquet d'arrivée

; Calcul du piquet intermédiaire avant de lancer la procédure récursive
; @param R0 nombre de disques
; @param R1 piquet de départ
; @param R2 piquet d'arrivée
hanoi:
```

```
        ; Calcul du troisième piquet : R3 = 6 - R1 - R2
        ADD R3,R2,R1
        NOT R3,R3
        ADD R3,R3,7     ; 7 = 6 + 1

; Procédure récursive
; @param R0 nombre de disques
; @param R1 piquet de départ
; @param R2 piquet d'arrivée
; @param R3 piquet intermédiaire
hanoirec:
        ; Décrémentation du nombre de disques
        ADD R0,R0,-1
        ; Test si le nombre de disques est 0
        BRn hanoiend
        ; Empilement de l'adresse de retour
        ADD R6,R6,-1
        STR R7,R6,0
        JSR swapR2R3
        JSR hanoirec
        JSR swapR2R3
        JSR print
        JSR swapR1R3
        JSR hanoirec
        JSR swapR1R3
        ; Dépilement de l'adresse de retour
        LDR R7,R6,0
        ADD R6,R6,1
        ; Restauration du nombre de piquets
hanoiend:
        ADD R0,R0,1
        RET

; Échange des contenus de R1 et R3
swapR1R3:
        ADD R4,R3,0
        ADD R3,R1,0
        ADD R1,R4,0
        RET
; Échange des contenus de R2 et R3
swapR2R3:
        ADD R4,R3,0
        ADD R3,R2,0
        ADD R2,R4,0
        RET

; Affichage de R1 -> R2
print:  ; Empilement de R7 et R0
        ADD R6,R6,-2
        STR R7,R6,1
        STR R0,R6,0
        ; Affichage du caractère '0' + R1
        LD R0,char0
        ADD R0,R0,R1
        TRAP x21        ; Appel système putc
        ; Affichage de la chaîne " --> "
        LEA R0,arrow
        TRAP x22        ; Appel système puts
        ; Affichage du caractère '0' + R2
        LD R0,char0
        ADD R0,R0,R2
        TRAP x21        ; Appel système putc
```

```
        ; Retour à la ligne
        LD R0,charnl
        TRAP x21        ; Appel système putc
        ; Dépilement de R0 et R7
        LDR R0,R6,0
        LDR R7,R6,1
        ADD R6,R6,2
        RET
; Constantes pour l'affichage
char0:  .FILL '0'
charnl: .FILL '\n'
arrow:  .STRINGZ " --> "
        .END
```

12.5 Comparaison avec d'autres micro-processeurs

Beaucoup de micro-processeurs (surtout CISC comme le 80x86) possèdent un registre, généralement appelé SP, dédié à la gestion de la pile. Il y a alors des instructions spécifiques pour manipuler la pile. La pile est alors systématiquement utilisée pour les appels à des sous-routines. Certaines instructions permettent en effet les appels et les retours de sous-routines. D'autres instruction permettent d'empiler ou de dépiler un ou plusieurs registres afin de simplifier la sauvegarde des registres.

12.5.1 Instructions `CALL` et `RET`

Les instructions d'appels de sous-routine empilent alors l'adresse de retour plutôt que de la mettre dans un registre particulier. Comme l'adresse de retour d'une sous-routine se trouve toujours sur la pile, il existe une instruction spécifique, généralement appelée `RET` pour terminer les sous-routines. Cette instruction dépile l'adresse de retour puis la met dans le compteur de programme PC. Ceci explique pourquoi l'instruction `JMP R7` du LC-3 est aussi appelée `RET`.

12.5.2 Instructions `PUSH` et `POP`

Les micro-processeurs ayant un registre dédié à la pile possèdent des instructions, généralement appelées `PUSH` et `POP` permettant d'empiler et de dépiler un registre sur la pile. Ces instructions décrémentent et incrémentent automatiquement le registre de pile pour le mettre à jour. Certains micro-processeurs comme le 80x86 ont même des instructions permettant d'empiler et de dépiler tous les registres ou un certain nombre d'entre eux.

12.6 Appels système

L'instruction `TRAP` permet de faire un appel au système. Les mots mémoire des adresses 0x00 à 0xFF contiennent une table des appels système. Chaque emplacement mémoire contient l'adresse d'un appel système. Il y a donc 256 appels système possibles. L'instruction `TRAP` contient le numéro d'un appel système, c'est-à-dire l'indice d'une entrée de la table des appels systèmes.

Comme l'instruction `JSR`, l'instruction `TRAP` sauvegarde le compteur de programme PC dans le registre R7 puis charge PC avec l'entrée de la table des appels système dont le numéro est indiqué par l'instruction. Le retour d'un appel système se fait par l'instruction `RET`.

La table des appels système procure une indirection qui a l'intérêt de rendre les programmes utilisateurs indépendant du système. Dans la mesure où l'organisation (c'est-à-dire la correspondance entre les numéros de la table et les appels système) de la table reste identique, il n'est pas nécessaire de

changer (recompiler) les programmes utilisateurs.

Sur les micro-processeurs usuels, l'instruction `TRAP` fait passer le micro-processeur en mode privilégié (aussi appelé mode système). C'est d'ailleurs souvent la seule instruction permettant de passer en mode privilégié. Ceci garantit que seul le code du système d'exploitation soit exécuté en mode privilégié. Par contre, il faut une instruction capable de sortir du mode privilégié avant d'effectuer le retour au programme principal. Le processeur LC-3 ne possède pas de telle instruction.

12.7 Interruptions

Les interruptions sont un mécanisme permettant à un circuit extérieur au micro-processeur d'interrompre le programme en cours d'exécution afin de faire exécuter une routine spécifique. Les interruptions sont en particulier utilisées pour la gestion des entrées/sorties et pour la gestion des processus.

12.7.1 Initiation d'une interruption

Le micro-processeur possède une ou plusieurs broches permettant de recevoir des signaux provenant des circuits extérieurs susceptibles de provoquer des interruptions. Avant d'exécuter chaque instruction, le micro-processeur vérifie s'il y a un signal (de valeur 1) sur une de ces broches. Si aucun signal n'est présent, il continue le programme et il exécute l'instruction suivante. Si au contraire un signal est présent, il interrompt le programme en cours et il commence à exécuter une sous-routine spécifique appelée *sous-routine d'interruption*. Lorsque cette sous-routine se termine, le micro-processeur reprend l'exécution du programme en cours à l'instruction où il en était.

L'exécution d'une sous-routine d'interruption s'apparente à l'exécution d'une sous-routine du programme. Par contre elle n'est initiée par aucune instruction du programme. Elle est initiée par le signal d'interruption et elle peut intervenir à n'importe quel moment de l'exécution du programme interrompu. Pour cette raison, cette sous-routine d'interruption ne doit modifier aucun des registres R0,…,R6 et R7 du micro-processeur. De la même façon, elle ne doit pas modifier aucun des indicateurs n, z et p car l'interruption peut intervenir entre une instruction qui positionne ces indicateurs (comme `LD` et `ADD`) et une instruction de branchement conditionnel (comme `BRz`) qui les utilise.

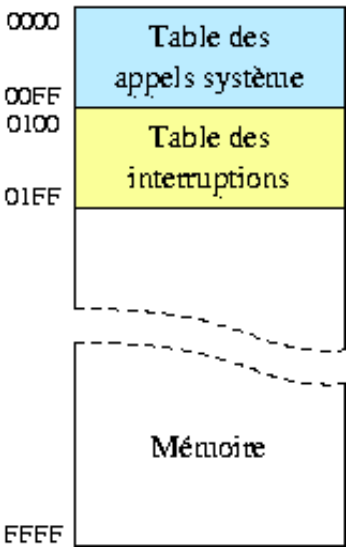
12.7.2 Retour d'une interruption

Afin de pouvoir revenir à l'instruction à exécuter, le micro-processeur doit sauvegarder le compteur de programme. Comme aucun registre ne peut être modifié, le compteur de programme ne peut être transféré dans le registre R7 comme les instructions `JSR`, `JSRR` et `TRAP` le font. Le compteur de programme est empilé sur la pile ainsi que le registre PSR qui contient les indicateurs n, z et p. Ensuite le compteur de programme est chargé avec l'adresse de la routine d'interruption. La routine d'interruption se charge elle-même d'empiler les registres dont elle a besoin afin de les restaurer lorsqu'elle se termine. La routine d'interruption se termine par une instruction spécifique `RTI` (pour *ReTurn Interrupt*). Il faut d'abord dépiler le registre PSR puis sauter à l'instruction dont l'adresse est sur la pile. Comme le registre R7 doit être préservé, l'instruction `RET` ne peut être utilisée. L'instruction `RTI` se charge de dépiler PSR et de dépiler l'adresse de retour pour la charger dans le registre PC.

12.7.3 Vecteurs d'interruption

Dans les micro-processeurs très simples, l'adresse de la routine d'interruption est fixe. S'il y a plusieurs circuits extérieurs susceptibles de provoquer une interruption, le premier travail de la routine d'interruption est de déterminer quel est le circuit qui a effectivement provoqué l'interruption. Ceci est fait en interrogeant les registres d'état (Status Register) de chacun de ces circuits.

S'il y a beaucoup de circuits pouvant provoquer une interruption, il peut être long de déterminer lequel a provoqué l'interruption. Comme le temps pour gérer chaque interruption est limité, les micro-processeurs possèdent plusieurs routines d'interruption. Chacune d'entre elles gère un des circuits extérieurs. Comme pour les appels systèmes, il existe une table des interruptions qui contient les adresses des différentes routines d'interruption. Pour le micro-processeur LC-3, celle-ci se trouve aux adresses de 0x100 à 0x1FF. Lorsqu'un circuit engendre une interruption, il transmet au micro-processeur via le bus de données un *vecteur d'interruption*. Ce vecteur est en fait l'indice d'une entrée de la table des interruptions. Pour le micro-processeur LC-3, il s'agit de la d'une valeur 8 bits auquel le micro-processeur ajoute 0x100 pour trouver l'entrée dans la table. Chaque circuit externe reçoit le vecteur d'interruption qui lui est attribué lors de son initialisation par le micro-processeur.



Tables des appels systèmes et interruptions

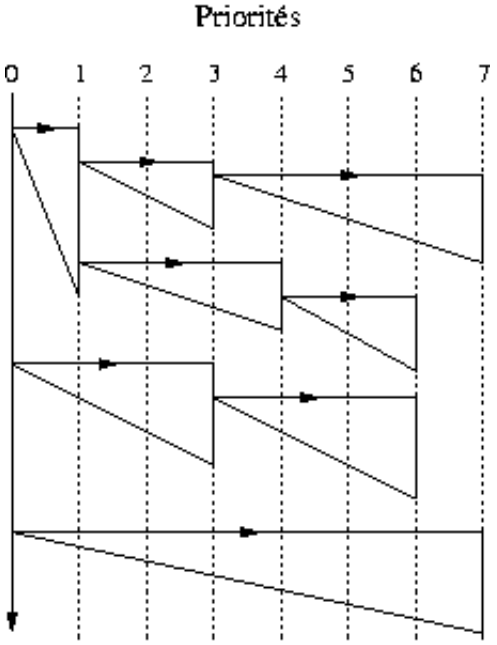
12.7.4 Priorités

Dans le micro-processeur LC-3, chaque programme est exécuté avec une certaine priorité allant de 0 à 7. Celle-ci est stockée dans 3 bits du registre PSR. Lorsqu'une interruption est demandée par un circuit extérieur, celle-ci a aussi une priorité. Elle est alors effectivement effectuée si sa priorité est supérieure à la priorité du programme en cours d'exécution. Sinon, elle est ignorée.

Lorsqu'une interruption est acceptée, la routine d'interruption est exécutée avec la priorité de l'interruption. Cette priorité est mise dans le registre PSR après la sauvegarde de celui-ci sur la pile. À la fin de la routine, l'instruction `RTI` restaure le registre PSR en le dépilant. Le programme interrompu retrouve ainsi sa priorité.

Il est possible qu'une demande d'interruption intervienne pendant l'exécution d'une autre routine d'interruption. Cette interruption est prise en compte si elle a une priorité supérieure à celle en cours. Ce mécanisme permet de gérer les urgences différentes des interruptions. Une interruption liée à un

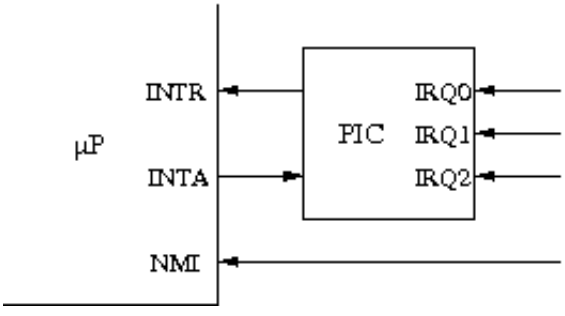
défaut de cache est nettement plus urgente qu'une autre liée à l'arrivée d'un caractère d'un clavier.



Imbrications des interruptions

12.7.5 Contrôleur d'interruption

Certains micro-processeurs comme le 8086 ne disposent que de deux interruptions. Chacune de ses deux interruptions correspond à une broche du circuit du micro-processeur. Une première interruption appelée NMI (Non Maskable Interrupt) est une interruption de priorité maximale puisqu'elle ne peut pas être ignorée par le micro-processeur. Celle-ci est généralement utilisée pour les problèmes graves comme un défaut de mémoire. Une seconde interruption appelée INTR est utilisée par tous les autres circuits extérieurs. Afin de gérer des priorités et une file d'attente des interruptions, un circuit spécialisé appelé PIC (Programmable Interrupt Controler) est adjoint au micro-processeur. Ce circuit reçoit les demandes d'interruption des circuits et les transmet au micro-processeur. Le micro-processeur dispose d'une sortie INTA (Interrupt Acknowledge) pour indiquer au PIC qu'une interruption est traitée et qu'il peut en envoyer une autre.



Contrôleur d'interruptions

12.7.6 Séparation des piles système et utilisateur

Pour plusieurs raisons, il est préférable que la pile utilisée par le système soit distincte de la pile de chacun des programmes utilisateurs. D'abord, si un programme gère mal sa pile et que celle-ci déborde, cela n'empêche pas le système de fonctionner correctement. D'autre part, si la pile est partagée, un programme peut en inspectant sa pile trouver des informations de sécurité qu'il ne devrait pas avoir.

Pour ces raisons, le micro-processeur LC-3 permet d'utiliser deux piles distinctes. Ceci est mis en œuvre de la manière suivante. Le micro-processeur dispose de deux registres USP (User Stack Pointer) et SSP (System Stack Pointer) qui sauvegardent le registre R6 utilisé comme pointeur de pile. Lorsque le micro-processeur LC-3 passe en mode privilégié (lors d'une interruption), le registre R6 est sauvegardé dans USP et R6 est chargé avec le contenu de SSP. Lorsqu'il sort du mode privilégié, R6 est sauvegardé dans SSP et R6 est chargé avec le contenu de USP. Ce mécanisme impose bien sûr que le pointeur de pile soit le registre R6.