Examen d'Architecture des Ordinateurs
Majeure 1 – Polytechnique
Lundi 10 Décembre 2001

- L'examen dure 3 heures.
- Le sujet comporte 7 pages dont 3 pages de rappels sur le LC-2 et la microprogrammation.
- Tous documents autorisés.
- Le barème est donné à titre indicatif, il sert surtout à évaluer le poids respectif des sections ; tel quel, l'examen est noté sur 40.
- L'examen contient un problème et un exercice. Dans le problème, les questions 1 et 2 peuvent être traitées indépendamment, mais il est recommandé de bien comprendre le fonctionnement des instructions PUSH et POP avant d'aborder la question 2.
- Il est impératif de commenter les programmes en assembleur et les microprogrammes ; normalement, presque chaque instruction ou microinstruction doit être suivie d'un commentaire.
- `Les corrections sont dans les cadres en dessous des questions.`

Problème. Pile et LC-2 (30 points)

On considère le processeur LC-2 vu en cours et en TD ; la structure du LC-2 et le jeu d'instructions sont rappelés en Annexe. Dans ce processeur on veut rajouter des instructions destinées à faciliter l'utilisation d'une pile, que l'on utilisera pour le stockage de données temporaires quelconques (destinées au calcul, aux appels de procédures,...). Dans une telle pile, on ne peut accéder qu'au dernier élément (ici, on visualisera une pile dont le dernier élément est situé au *bas* de la pile). On dispose de deux nouvelles instructions pour utiliser cette pile : PUSH Rs et POP Rd :

- l'effet de PUSH Rs est d'ajouter le contenu du registre Rs au bas de la pile ; le registre Rs n'est pas modifié.
- l'effet de POP Rd est de placer la donnée située dans le dernier élément de la pile dans le registre Rd et d'enlever cette donnée de la pile.

Le format et l'opcode des instructions PUSH et POP sont indiqués ci-dessous :

Bit	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
PUSH	1011				Rs1			1	0	1	1	0	0	0	0	1
POP	1010				Rd			1	0	1	1	1	1	1	1	1

Voici un exemple de fonctionnement d'une pile à 4 éléments de 16 bits chacun (les valeurs sont données en hexadécimal):

Etat initial	*Etat après PUSH R1 (R1 contient 0A2F)*	*Etat après PUSH R2 (R2 contient 19C3)*	*Etat après POP R3 (R3 contient alors 19C3)*
0000	0000	0000	0000
0000	0000	0000	0000
0000	0000	0A2F	0000
0000	0A2F	19C3	0A2F

Dans l'exemple ci-dessus, on déplace les données dans la pile afin de donner l'intuition du fonctionnement de la pile ; en pratique et selon les implémentations de la pile, les données sont fixes (les données sont stockées en mémoire) ou se déplacent effectivement (il existe un composant matériel sur le processeur représentant la pile); ce point est sans importance pour la question 1, il sera abordé en détail à la question 2.
On suppose maintenant que le LC-2 dispose des instructions PUSH et POP et d'une implémentation de la pile. Dans la section 1 on utilise cette pile pour la programmation en assembleur, et dans la question 2 on étudie les modifications à apporter à l'architecture pour implémenter la pile.

1. Utilisation de la pile (15 points)

Dans cette section, on ne se préoccupe pas de la taille de la pile que l'on suppose suffisamment grande pour tous les programmes ci-dessous. Comme la pile sert de zone de stockage, elle permet souvent de réduire le nombre de registres nécessaires dans une architecture ; dans certaines architectures, il peut n'y avoir pratiquement aucun

registre, on n'utilise alors que la pile ; c'est pourquoi, dans plusieurs questions, on restreint volontairement le nombre de registres dont on peut disposer.
Dans toutes les questions, on suppose qu'initialement tous les éléments de la pile contiennent la valeur hexadécimale 0000.
De manière générale, on privilégie ici la compacité (et la lisibilité) du programme sur le temps d'exécution : on préfère un programme comportant peu d'instructions à un programme plus rapide mais comportant plus d'instructions.

1.1 En utilisant un seul registre (le registre R0), les instructions de gestion de la pile décrites ci-dessus et les instructions arithmétiques et logiques du LC-2, modifier la pile pour que ses 4 derniers éléments aient les valeurs décimales suivantes :

....
28
56
8
-9

On veillera à minimiser le nombre d'instructions nécessaires au remplissage de la pile.

```
AND R0, R0, #0    ; R0=0
ADD R0, R0, #15   ; R0=15
ADD R0, R0, #13   ; R0=28
PUSH R0
ADD R0, R0, R0    ; R0=56
PUSH R0
AND R0, R0, #15   ; R0=8
PUSH R0
NOT R0, R0        ; R0=-9
PUSH R0
```

1.2 On veut écrire un programme qui effectue la division entière de b par a, a et b étant deux nombres représentés sur 16 bits et tels que $0 < a, b < 2^{15}$. On suppose que a et b sont respectivement stockés dans les registres R0 et R1.

1.2.1 Proposer un algorithme, et le présenter sous forme d'organigramme. Il existe un algorithme symétrique à celui vu en cours pour la multiplication, mais on conseille ici d'utiliser un algorithme plus élémentaire, dont le temps d'exécution peut éventuellement être plus long. Ecrire le programme assembleur LC-2 correspondant. En fin de programme, le quotient sera stocké dans R0 et le reste dans R1.

On implémente l'algorithme de division en comptant combien de fois on peut soustraire *a* à *b*.

```
          quotient = 0
BOUCLE    reste < diviseur ?
OUI       FIN
NON       reste = reste – diviseur
          quotient++
          aller à BOUCLE

          .ORIG x3000
          AND R2, R2, #0    ; quotient=0
BOUCLE    NOT R3, R0        ; R3=-diviseur
          ADD R3, R3, #1    ;
          ADD R3, R1, R3    ; R3=reste-diviseur
          BRn FIN
          ADD R1, R3, #0    ; reste=reste-diviseur
          ADD R2, R2, #1    ; quotient++
          JMP BOUCLE
FIN       ADD R0, R2, #0    ; R0=quotient
          .END
```

1.2.2 Ecrire une nouvelle version de ce programme en n'utilisant que les registres R0, R1, R2 et R3 (on n'utilise ni la mémoire, ni la pile).

Cette question concerne les programmes du 1.2.1 comportant plus de 4 registres.

1.2.3 Ecrire à nouveau le programme en n'utilisant que les registres R0, R1, R2 et la pile. En fin de programme, le quotient sera dans R0 et le reste dans R1.

Une solution :

```
          .ORIG x3000
          AND R2, R2, #0    ; quotient=0
          PUSH R2           ; quotient sauvegardé sur la pile
BOUCLE    NOT R2, R0        ; R2=diviseur
          ADD R2, R2, #1    ;
          ADD R2, R1, R2    ; R2=reste-diviseur
          BRn FIN
          ADD R1, R2, #0    ; reste=reste-diviseur
          POP R2            ; Récupération du quotient
          ADD R2, R2, #1    ; quotient++
          PUSH R2           ; Sauvegarde du quotient
          JMP BOUCLE
FIN       POP R0            ; R0=quotient
          .END
```

1.3 On a vu en cours que les appels de procédure, une fois codés en assembleur LC-2, utilisent une pile de contextes. Dans le LC-2, on gère normalement les contextes à l'aide d'instructions mémoire (LD, ST, LDR, STR). On veut maintenant utiliser les instructions PUSH et POP pour gérer les contextes.
On suppose ici que l'on n'a pas de variables locales. Dans cette question, les seules zones de stockage dont l'on dispose sont les registres et la pile, on n'utilise pas les instructions mémoire du LC-2.

1.3.1 Par rapport au contexte décrit en cours, quelle information est-il maintenant inutile de garder dans le contexte s'il est géré à l'aide des instructions PUSH et POP ? Expliquer brièvement pourquoi.

En l'absence d'instructions de gestion de la pile, la position en mémoire des contextes du LC-2 est indiquée par un pointeur stocké dans R6. Ce pointeur indique le début du contexte courant. Avec PUSH et POP, il n'est plus nécessaire d'indiquer explicitement la position en mémoire du contexte, le dernier élément de la pile va correspondre au dernier élément du contexte courant, c'est un pointeur implicite sur le contexte courant.

1.3.2 Si l'on ne dispose que des instructions PUSH et POP, peut-on utiliser la pile pour le passage de paramètres à la procédure appelée ? Expliquer brièvement pourquoi. Même question pour le passage du résultat à la procédure appelante.

- Passage de paramètres : supposons que l'on place un paramètre sur la pile à l'aide d'un PUSH dans la procédure appelante ; en début de procédure appelée, on doit sauver les registres utilisés sur la pile (ils font partie du contexte de la procédure appelée) avant toute opération sur les registres ; on ne pourra donc accéder aux paramètres qui se situeront, dans la pile, au-dessus des registres sauvés. Si la procédure appelée ne modifie aucun registre, le passage de paramètres est possible. Cependant, si la sauvegarde des registres est effectuée dans la procédure appelante, il est possible d'utiliser la pile pour passer les paramètres
- Passage de résultat : le raisonnement est similaire ; en fin de procédure appelée, on doit restaurer les registres modifiés avec l'instruction POP avant de retourner à la procédure appelante ; le résultat étant nécessairement dans un registre modifié, la restauration va écraser ce résultat ; inversement, si l'on sauve le résultat sur la pile avant la restauration des registres, on ne pourra accéder aux valeurs des registres à restaurer. Cependant, si la sauvegarde des registres est effectuée dans la procédure appelante, il est possible d'utiliser la pile pour renvoyer le résultat.

1.3.3 On veut effectuer le calcul suivant : *((a / b + c) / d)* en utilisant des divisions entières. Ecrire le programme correspondant en utilisant la procédure de division entière de la question 1.2. Initialement, *a, b, c, d* sont respectivement dans les registres R0, R1, R2, R3. On passe les paramètres à la procédure appelée directement par les registres R0 et R1, et cette procédure renvoie directement le résultat par les registres R0 et R1 comme indiqué à la question 1.2.1. En fin de programme, le résultat devra être dans R0. Distinguer la procédure appelante et la procédure appelée ; indiquer clairement les instructions à rajouter dans le programme de division entière de la question 1.2.2 (ou 1.2.1 si vous n'avez pas fait le 1.2.2) pour le transformer en procédure. Dans la procédure appelante, on ne pourra utiliser que les registres R0, R1, R2, R3 et la pile.

Procédure appelée :
On utilise la procédure du 1.2.2 :

```
DIV       PUSH R2           ; sauvegarde des registres R2 et R3
          PUSH R3
                            ; Il n'est pas nécessaire de sauver l'adresse
                            ; de retour contenue dans R7
          AND R2, R2, #0    ; quotient=0
BOUCLE    NOT R3, R0        ; R3=diviseur
          ADD R3, R3, #1    ;
          ADD R3, R1, R3    ; R3=reste-diviseur
          BRn FIN
          ADD R1, R3, #0    ; reste=reste-diviseur
          ADD R2, R2, #1    ; quotient++
          JMP BOUCLE
FIN       ADD R0, R2, #0    ; R0=quotient
          POP R3            ; Restauration des registres dans l'ordre inverse
          POP R2            ; de la sauvegarde
```

Procédure appelante :

Dans la question 1.2, on effectue la division de *b* par *a*, ici on effectue d'abord la division de *a* par *b*, il faut donc intervertir les valeurs des registres R0 et R1 avant d'appeler la procédure de division ; comme les registres R2 et R3 sont utilisés, il faut passer par la pile pour intervertir R0 et R1.

```
          .ORIG x3000
          PUSH R1           ; Sauvegarde de R0 et R1 pour interversion
          PUSH R0
          POP R1            ; Interversion
          POP R0
          JSR DIV
          ADD R1, R0, R2    ; R1=(a/b+c), R0 contient le résultat
          ADD R0, R3, #0    ; R0=d
          JSR DIV
          .END              ; Le résultat est dans R0
```

1.4 On veut maintenant réaliser un programme en assembleur qui effectue la conversion en base *b* d'un nombre *A* (*A* et *b* sont des entiers strictement positifs). On veut effectuer cette conversion en utilisant une succession de divisions entières.

1.4.1 Ecrire un algorithme de conversion qui utilise la division entière. On présentera cet algorithme sous forme d'organigramme.

On utilise une succession de divisions entières :
$A = q_0 \times b + r_0$
$q_0 = q_1 \times b + r_1$
$q_1 = q_2 \times b + r_2$
... arrêt lorsque $q_n=0$; le nombre en base *b* est $r_n r_{n-1} \ldots r_1 r_0$

```
          quotient=A
          diviseur=b
BOUCLE    quotient < b ?
OUI       FIN, le quotient est le dernier chiffre (poids fort)
NON       reste = RESTE(quotient/b)
          quotient = QUOTIENT(quotient/b)
          reste est le nouveau chiffre
          aller à BOUCLE
```

1.4.2 Initialement, *A* est stocké dans le registre R1 et *b* dans le registre R0. Ecrire le programme de conversion en utilisant le programme de division entière de la question 1.2. Le programme de division entière de la question 1.2 sera considéré comme une procédure appelée par le programme de conversion ; on utilise les hypothèses de la question 1.3 pour le passage des paramètres et la récupération des résultats de la procédure de division entière. En fin de programme, le résultat sera stocké dans la pile avec un chiffre par élément de la pile et le chiffre le plus significatif (par exemple le bit de poids fort si *b*=2) sera situé en bas de la pile. Dans la procédure appelante, on ne pourra utiliser que les registres R0, R1, R2, R3 et la pile.

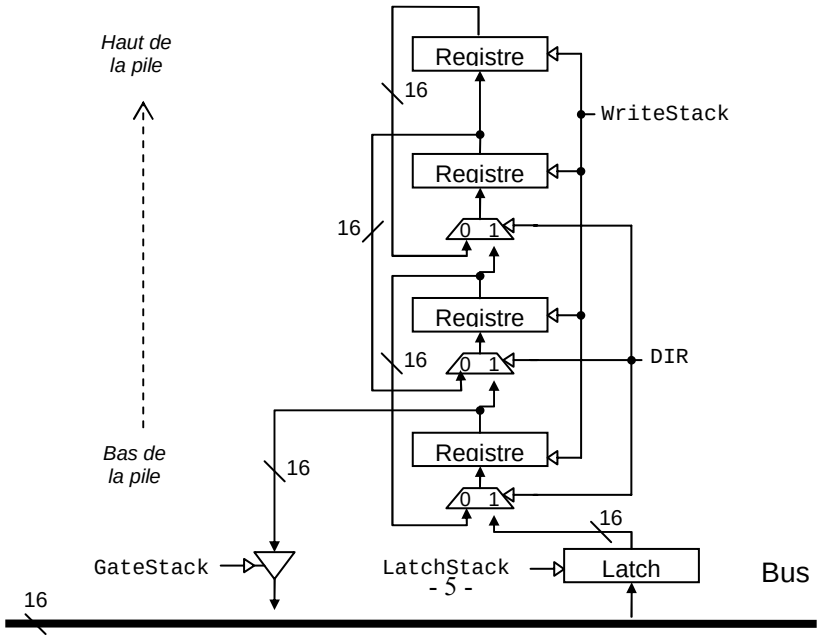
```
          .ORIG x3000
          ADD R2, R0, #0    ; b dans R2
BOUCLE    JSR DIV           ; le reste est dans R1, le quotient dans R0
          PUSH R1
          ADD R1, R0, #0    ; quotient dans R1
          ADD R0, R2, #0    ; b dans R0
          ADD R1, R1, #0    ; quotient != 0 ?
          BRp BOUCLE
          .END
```

1.4.3 Même question en n'utilisant que les registres R0, R1, R2 et la pile dans la procédure appelante. Peut-on écrire la procédure appelante avec seulement R0, R1 et la pile ?

```
Pour la première partie de la question, voir question 1.4.2
Pas de solution avec 2 registres (à démontrer).
```

2. Implémentation de la pile (15 points)

On utilise le processeur LC-2 vu en cours et en TD. Tout comme en TD, le LC-2 dispose d'un contrôle



microprogrammé, et on utilise le microséquenceur vu en TD et rappelé en annexe; il est impératif d'utiliser la syntaxe des microinstructions spécifiée en annexe (ne pas écrire les microinstructions en binaire). Dans le LC-2, le signal de contrôle DRMX a été modifié. Enfin, en ce qui concerne la durée des étapes, on précise ici que l'écriture dans un registre nécessite un cycle ; on considérera également que le passage à travers le bus suivi de l'écriture dans un latch ou un registre nécessite un cycle ; enfin, on négligera le temps nécessaire à la traversée de l'ALU sans effectuer de calcul. D'une manière générale, on considérera qu'il n'est pas vital de minimiser la durée d'une instruction et donc le nombre d'étapes, même s'il faut éviter un nombre excessivement grand d'étapes.

Figure 1. Implémentation de la pile.

2.1 **Pile dans un composant dédié**. Dans cette question, on veut ajouter au processeur LC-2 une véritable pile matérielle, implémentée sous la forme d'une succession de 4 registres 16-bits, comme indiqué Figure 1. Le circuit de la pile possède 4 signaux de contrôle :

- WriteStack qui indique que l'on modifie le contenu de la pile (WriteStack=0 pas de modification, WriteStack=1 en cas de modification) ; le signal WriteStack commande l'écriture dans les registres de la pile.
- DIR qui indique le sens de progression des éléments dans la pile (DIR=0 pour une progression vers le bas, DIR=1 pour une progression vers le haut) ;
- GateStack qui contrôle des *tristates* reliant la sortie du dernier élément de la pile au bus (GateStack=0 lorsqu'il n'y a pas de connexion au bus, GateStack=1 lorsqu'il y a connexion) ;
- LatchStack qui commande l'écriture dans un latch relié au dernier élément de la pile ; ce latch stocke la valeur provenant du bus et à placer dans la pile (si LatchStack=0 pas d'écriture, si LackStack=1 on écrit dans le latch).

On suppose que l'on a ajouté ces signaux de contrôle dans le champ *Signaux* des microinstructions.

2.1.1 On veut définir le microprogramme de contrôle correspondant à l'instruction POP. Donner les différentes étapes de l'exécution de l'instruction POP en précisant les signaux de contrôle à activer. Il est inutile de donner la valeur de tous les signaux de contrôle à chaque étape, seuls les signaux intervenant dans une étape seront mentionnés.

2.1.2 Ecrire le microprogramme de contrôle pour l'instruction POP (on ne se préoccupera pas des autres instructions assembleur LC-2 ; on écrit le microprogramme en supposant qu'il n'y a qu'une seule instruction assembleur dans le LC-2).

Adresse	Nom	Condition	Adresse	Signaux	
0	MS	-	-	GatePC=1, LD.MAR=1, PCMX=00, LD.PC=1	*PC→MAR ; PC+1→PC*
1	MS	-	-	LD.MDR=1, R.W=0	*Mémoire→MDR*
2	MS	-	-	GateMDR=1, LD.IR=1	*MDR→IR*
3	MS	-	-	GateStack=1, LD.REG=1, LD.CC=1, DRMX=0	*Dernier élément pile→DR*
4	MS	-	-	DIR=0, WriteStack=1	*Décalage pile vers le bas*
5	BI	-	0		*Retour au début*

2.1.3 Reprendre la question 2.1.2 pour l'instruction PUSH, en indiquant en commentaire de chaque microinstruction l'étape à laquelle elle correspond.

Adresse	Nom	Condition	Adresse	Signaux	
0	MS	-	-	GatePC=1, LD.MAR=1, PCMX=00, LD.PC=1	*PC→MAR ; PC+1→PC*
1	MS	-	-	LD.MDR=1, R.W=0	*Mémoire→MDR*
2	MS	-	-	GateMDR=1, LD.IR=1	*MDR→IR*
3	MS	-	-	SR1MUX=00	*Lecture Rs1*
4	MS	-	-	ALUK=11, GateALU=1, LatchStack=1	*Rs1→Latch pile (par l'ALU)*
5	MS	-	-	DIR=1, WriteStack=1	*Décalage pile vers le haut et empilement de la donnée contenue dans le latch de la pile*
6	BI	-	0		*Retour au début*

2.1.4 Ecrire le microprogramme de contrôle du LC-2 en supposant qu'il ne contient que deux instructions : POP et PUSH.

Adresse	Nom	Condition	Adresse	Signaux	
0	MS	-	-	GatePC=1, LD.MAR=1, PCMX=00, LD.PC=1	*PC→MAR ; PC+1→PC*
1	MS	-	-	LD.MDR=1, R.W=0	*Mémoire→MDR*
2	MS	-	-	GateMDR=1, LD.IR=1	*MDR→IR*
3	BC	I_{12}	8		*Test sur opcode pour différencier PUSH et POP (dans le LC-2 le test est placé après la phase 3, ici, on essaie de factoriser au maximum les étapes)*
4	MS	-	-	GateStack=1, LD.REG=1, LD.CC=1, DRMX=0	*Début POP* *Dernier élément pile→DR*
5	MS	-	-	DIR=0, WriteStack=1	*Décalage pile vers le bas*
6	BI	-	0		*Fin POP* *Retour au début*
7	MS	-	-	SR1MUX=00	*Début PUSH* *Lecture Rs1*
8	MS	-	-	ALUK=11, GateALU=1, LatchStack=1	*Rs1→Latch pile (par l'ALU)*
9	MS	-	-	DIR=1, WriteStack=1	*Décalage pile vers le haut et empilement de la donnée contenue dans le latch de la pile*
10	BI	-	0		*Fin PUSH* *Retour au début*

2.1.5 On suppose que le registre R0 contient la valeur hexadécimale 000A, R1 la valeur 000B, R2 la valeur 000C, R3 la valeur 000D et R4 la valeur 000E. Donner le contenu de la pile de la Figure 1 après la séquence : PUSH R0, PUSH R1, PUSH R2, PUSH R3, PUSH R4. On exécute ensuite POP R0. Donner le contenu de la pile de la Figure 1. On exécute encore une fois POP R0. Donner à nouveau le contenu de la pile.

000B
000C
000D
000E

Après PUSH R0, PUSH R1, PUSH R2, PUSH R3, PUSH R4, 000A a disparu

000C
000B
000C
000D

Après POP R0

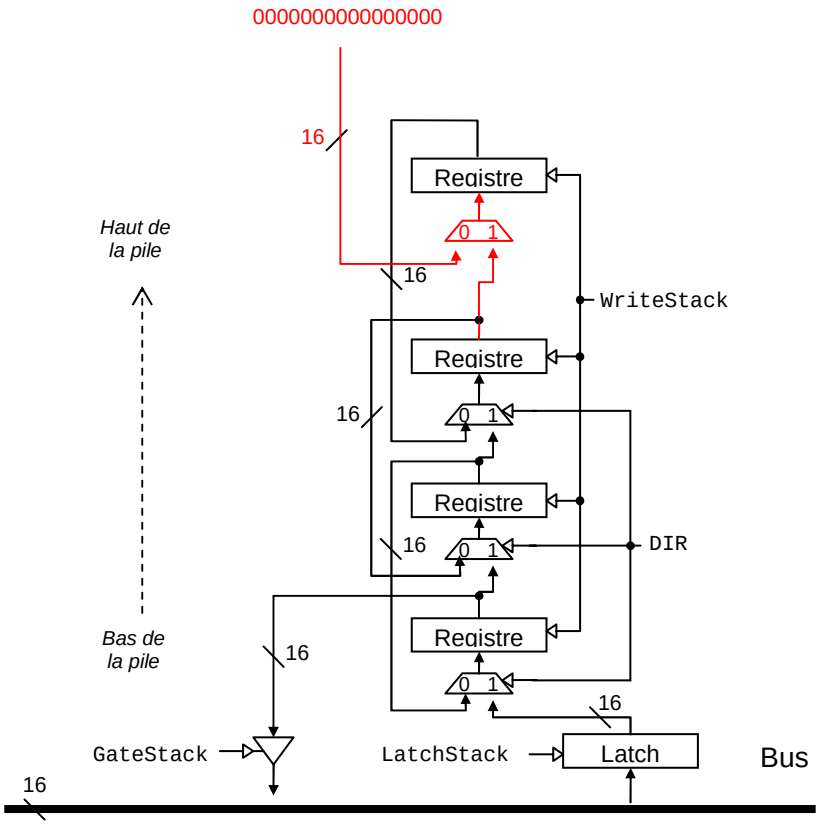
000B
000C
000B
000C

Après un nouveau POP R0

L'absence de multiplexeur en entrée de l'élément haut de la pile (le 4ème élément à partir du bas) fait que le 3ème élément de la pile est systématiquement injecté vers le haut, même en cas de POP, i.e., même lorsque la pile est décalée vers le bas.

2.1.6 On suppose qu'à l'état initial, tous les éléments de la pile contiennent la valeur hexadécimale 0000. On veut qu'un élément vide de la pile contienne toujours la valeur 0000. Comment modifier l'implémentation de la pile de la Figure 1 pour que ce soit le cas ? Répondre notamment avec un schéma correspondant à une version modifiée de la Figure 1. Cette modification de la pile ne sera pas utilisée dans le reste de l'énoncé.

Il faut placer un multiplexeur en entrée du 4ème élément, comme pour les autres éléments de la pile. En cas de POP (DIR=0), il faut fournir en entrée la valeur 0 (0000000000000000 en binaire).



2.2 **Pile en mémoire**. Dans cette question, on suppose que la pile est stockée dans une zone de la mémoire principale. Il n'existe pas de composant spécifique dans le LC-2 matérialisant la pile, contrairement à la question précédente. En cas de PUSH et POP, on ne déplace pas les données en mémoire, on se contente de déplacer un *pointeur de pile*. Ce pointeur indique l'adresse en mémoire du dernier élément de la pile (le bas de la pile). On va considérer que le registre R5 contient ce pointeur de pile (attention, le rôle du registre R5 n'est pas exactement le même que celui vu en cours pour le registre R6 et la pile des contextes, il s'agit de deux notions différentes). On précise que le pointeur de pile se déplace dans le sens des adresses croissantes au fur et à mesure que l'on ajoute des éléments à la pile.

2.2.1 Indiquer brièvement les atouts et les inconvénients de cette implémentation de la pile par rapport à celle de la question 2.1.

Principal inconvénient :
- Accès plus lent (requête à la mémoire)

Principal atout :
- Grande taille (nombreux appels de procédure imbriqués, nombreuses opérandes de calcul)

2.2.2 Reprendre la question 2.1.2 avec cette implémentation de la pile.

Adresse	Nom	Condition	Adresse	Signaux	
0	MS	-	-	GatePC=1, LD.MAR=1, PCMX=00, LD.PC=1	*PC→MAR ; PC+1→PC*
1	MS	-	-	LD.MDR=1, R.W=0	*Mémoire→MDR*
2	MS	-	-	GateMDR=1, LD.IR=1	*MDR→IR*
3	MS	-	-	SR1MX=01	*Lecture R5, immédiat (les bits 8 à 6 contiennent 101, soit la valeur 5 ; les bits 4 à 0 contiennent 11111 soit la valeur -1 en complément à 2 sur 5 bits ; le bit 5 est à 1, donc l'immédiat est sélectionné plutôt que SR2 ; l'immédiat est envoyé sur le 2ème port ALU et utilisé à l'étape 7)*
4	MS	-	-	ALUK=11, GateALU=1, LD.MAR=1	*R5→MAR par ALU*
5	MS	-	-	R.W=0, LD.MDR=1	*Mémoire→MDR*
6	MS	-	-	GateMDR=1, LD.REG=1, LD.CC=1, DRMX=0	*MDR→DR*
7	MS	-	-	ALUK=00	*R5 + (-1)*
8	MS	-	-	GateALU=1, LD.REG=1, DRMUX=1	*R5-1→R5*
9	BI	-	0		*Retour au début*

2.2.3 Reprendre la question 2.1.3 avec cette implémentation de la pile.

Pour l'instruction PUSH, il faut prendre garde à déplacer le pointeur de pile *avant* d'écrire dans la pile, car R5 pointe sur le dernier élément de la pile, donc sur un élément occupé, et il faut d'abord le déplacer sur un élément libre.

Adresse	Nom	Condition	Adresse	Signaux	
0	MS	-	-	GatePC=1, LD.MAR=1, PCMX=00, LD.PC=1	*PC→MAR ; PC+1→PC*
1	MS	-	-	LD.MDR=1, R.W=0	*Mémoire→MDR*
2	MS	-	-	GateMDR=1, LD.IR=1	*MDR→IR*
3	MS	-	-	SR1MX=01	*Lecture R5, immédiat (00001 spécifié dans bits 4 à 0 de l'instruction PUSH)*
4	MS	-	-	ALUK=00	*R5+1 (R5+immédiat) ; on doit faire R5+1 avant l'accès à la mémoire car R5 pointe sur le dernier élément de la pile, donc un emplacement occupé, et il faut l'amener sur le premier emplacement libre*
5	MS	-	-	GateALU=1, LD.REG=1, DRMUX=1	*R5+1→R5*
6	MS	-	-	ALUK=11, GateALU=1, LD.MAR=1	*R5→MAR par ALU ; on récupère R5 depuis le banc de registres (on a toujours SR1MX=01), donc la nouvelle valeur de R5 qui pointe maintenant sur le premier emplacement libre de la pile*
7	MS	-	-	SR1MX=00	*Lecture Rs1*
8	MS	-	-	ALUK=11, GateALU=1, LD.MDR=1	*Rs1→MDR (le conflit de ressources sur le port SR1 du banc de registres et sur le bus empêche l'envoi simultané de l'adresse et de la donnée dans MAR et MDR, d'où l'étape 5)*
9	MS	-	-	R.W=1	*Ecriture Mémoire*
10	BI	-	0		*Retour au début*

2.3 **Pile = Composant dédié + Mémoire**. On considère à nouveau que l'on dispose du composant matériel de la Figure 1. En outre, en cas de dépassement de la capacité de la pile matérielle, on veut que l'élément en excès soit envoyé à la mémoire. On suppose donc que la mémoire contient une pile annexe qui prolonge la pile matérielle. Comme dans la question 2.2, le pointeur de cette pile annexe est R5 ; il contient l'adresse en mémoire de l'élément situé au bas de cette pile annexe (le dernier élément de la pile) ; on précise enfin que l'adresse en mémoire du premier élément de la pile annexe est 0000, et que la pile annexe ne peut contenir plus de 1024 éléments (mais on ne se préoccupera pas de la gestion du dépassement de capacité de la pile annexe). On dispose d'un signal de condition supplémentaire *StackFull* qui vaut 1 lorsque la pile matérielle est pleine (lorsque les 4 éléments sont utilisés).

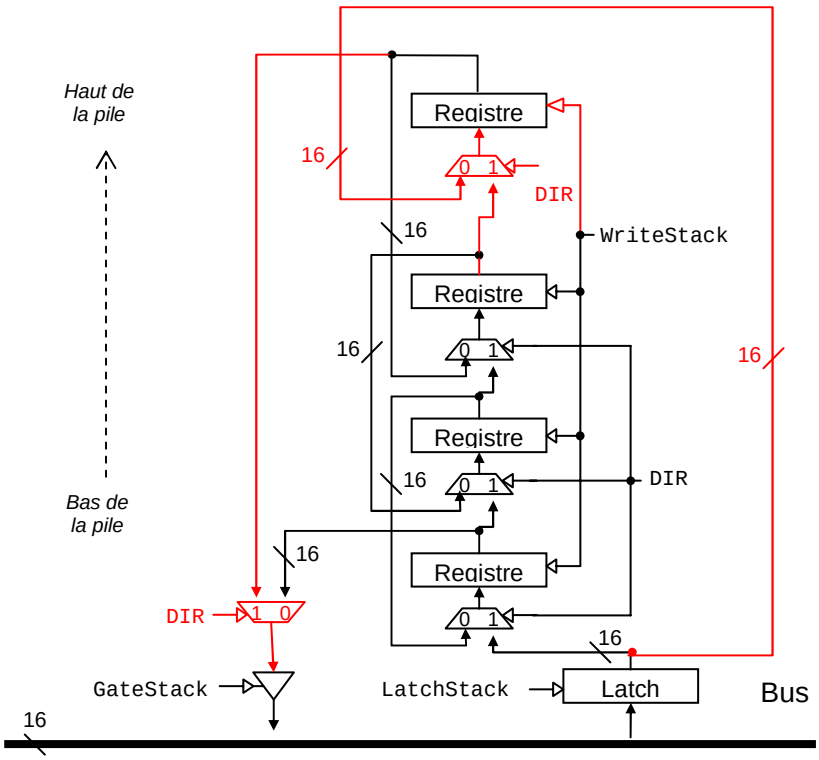
2.3.1 Quel élément de la pile matérielle faut-il envoyer à la mémoire en cas de dépassement de capacité de la pile matérielle (élément en excès) ? Quand est-il nécessaire de tester si la pile annexe est vide ? Expliquer comment tester si la pile annexe est vide.

L'élément en excès est celui situé en haut de la pile matérielle.
Il faut tester si la pile annexe est vide en cas de POP : si la pile annexe n'est pas vide, il faut ramener le dernier élément de la pile annexe dans le premier élément (en haut) de la pile matérielle, et dépiler la pile annexe d'un élément.
R5 pointe sur le dernier élément de la pile. On précise que l'adresse en mémoire du premier élément de la pile est 0000. Par conséquent, si la pile ne contient qu'un seul élément, R5=0000. Si la pile est vide, R5 est donc égal à FFFF. Il faut donc tester si R5 est égal à FFFF. Comme la pile ne peut contenir plus de 1024 éléments, on ne peut avoir le cas ambigu où la pile n'est pas vide et R5=FFFF (de toute façon, R5=FFFF signifie que la pile annexe remplit toute la mémoire ; ce cas est absurde car il signifie nécessairement que la pile a écrasé le programme)
Pour effectuer ce test, on lit R5 sur le port SR1 du banc de registres, et on positionne l'ALU sur l'opération NOT. Si R5=FFFF, le résultat de l'ALU est égal à 0000. D'après la description du microséquenceur, le bit *z* est un bit de condition. Donc si le bit *z* est à 1, on sait que R5 est égal à 0.

2.3.2 Indiquer comment modifier le schéma de la Figure 1 pour permettre le mécanisme de pile annexe. Encore une fois, on privilégiera des modifications simples de l'architecture décrite dans la Figure 1.

- POP : si la pile annexe n'est pas vide, on doit amener le dernier élément de la pile annexe dans le latch, puis décaler la pile matérielle vers le bas, en plaçant l'élément dans le latch en haut de la pile annexe. Les modifications matérielles nécessaires pour POP sont donc : (1) un multiplexeur en entrée de l'élément haut de la pile, (2) un chemin du latch au multiplexeur.

- PUSH : Le raisonnement est similaire. Si la pile matérielle est pleine (signal de condition StackFull), il faut envoyer l'élément haut de la pile matérielle dans la pile annexe. Il faut donc un chemin de l'élément haut vers le bus, d'où le multiplexeur placé avant GateStack.



2.3.3 Reprendre les questions 2.1.2 et 2.1.3 avec cette implémentation de la pile.

POP :

On teste si la pile annexe est vide ; si ce n'est pas le cas, il faut ramener le dernier élément de la pile annexe en haut de la pile matérielle, décaler vers le bas la pile matérielle, puis modifier le pointeur de la pile annexe contenu dans R5.

On peut également accélérer l'instruction en testant StackFull avant de tester si la pile annexe est vide ; si StackFull n'est pas à 1, il est inutile de tester si la pile annexe est vide ; il s'agit seulement d'une optimisation et ce n'est pas non plus la solution retenue ci-dessous.

Adresse	**Nom**	**Condition**	**Adresse**	**Signaux**	
0	MS	-	-	GatePC=1, LD.MAR=1, PCMX=00, LD.PC=1	*PC→MAR ; PC+1→PC*
1	MS	-	-	LD.MDR=1, R.W=0	*Mémoire→MDR*
2	MS	-	-	GateMDR=1, LD.IR=1	*MDR→IR*
3	MS	-	-	SR1MX=01	*Début test pour savoir si pile annexe vide* *Lecture R5*
4	MS	-	-	ALUK=10	*ALU effectue NOT*
5	MS	-	-	GateALU=1, LD.CC=1	*Calcul de CC, notamment de z*
6	BC	z	13		*Si z=1, la pile annexe est vide, il n'est pas nécessaire de la décaler ; Aller au décalage de la pile matérielle*
7	MS	-	-	SR1MX=01	*Décalage pile annexe :* *Lecture R5*
8	MS	-	-	ALUK=11, GateALU=1, LD.MAR=1	*R5→MAR par ALU*
9	MS	-	-	R.W=0, LD.MDR=1	*Mémoire→MDR*
10	MS	-	-	GateMDR=1, LatchStack=1,	*MDR→Latch de la pile*
11	MS	-	-	ALUK=00	*R5 + (-1)*
12	MS	-	-	GateALU=1, LD.REG=1, DRMUX=1	*Fin décalage de la pile annexe* *R5-1→R5*
13	MS	-	-	DIR=0, WriteStack=1	*Décalage pile matérielle (idem 2.1) :* *Décalage pile vers le bas (l'élément contenu dans le latch passe dans l'élément haut de la pile ; c'est utile si la pile annexe n'était pas vide)*
14	MS	-	-	GateStack=1, LD.REG=1, LD.CC=1, DRMX=0	*Pile→DR*
15	BI	-	0		*Retour au début*

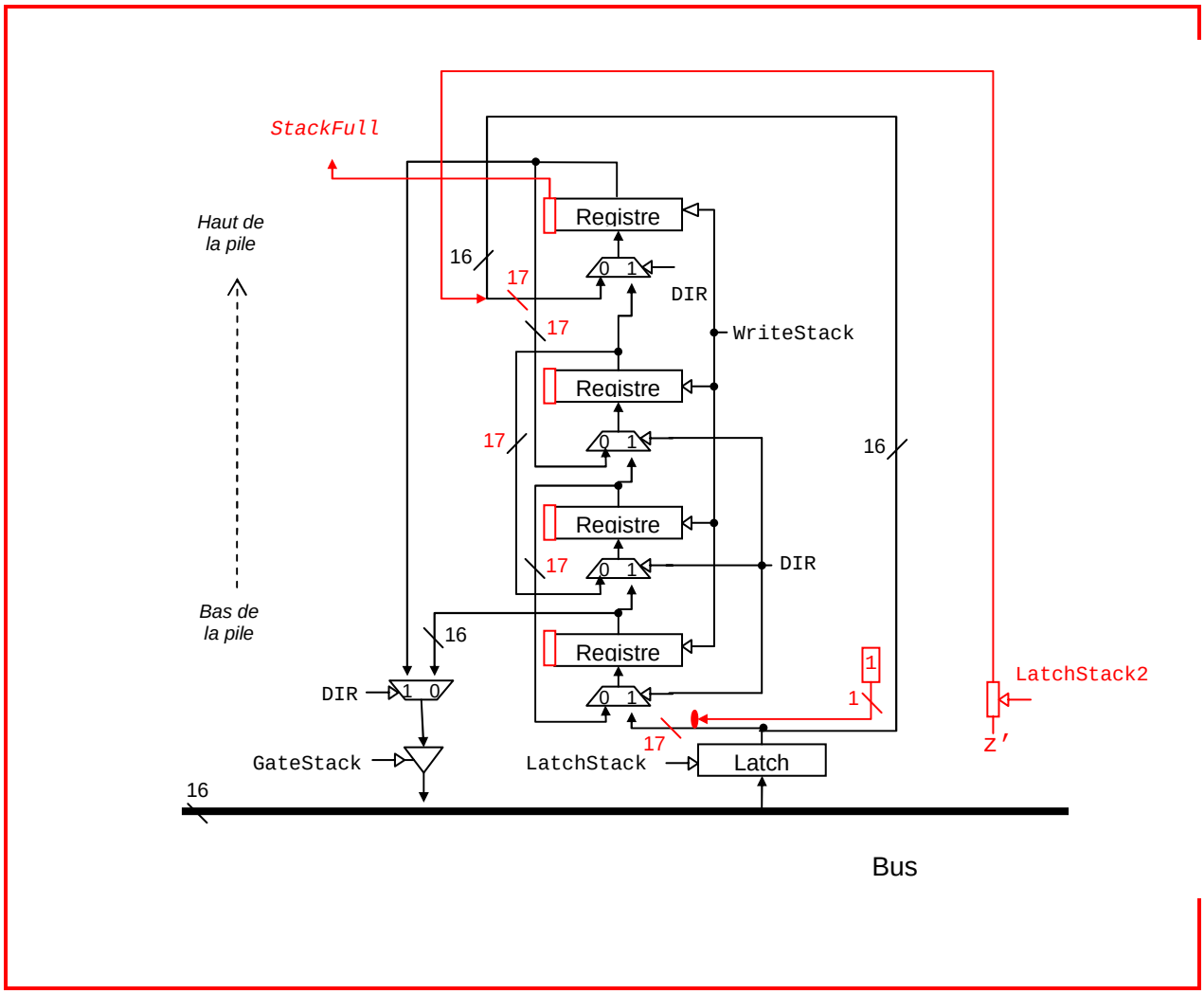
PUSH :

Ici, il faut nécessairement tester si StackFull=1 *avant* de décaler la pile matérielle vers le haut. Si StackFull=0, il suffit de décaler la pile matérielle. Si StackFull=1, il faut d'abord envoyer le haut de la pile matérielle dans la pile annexe, puis décaler la pile matérielle.

Adresse	Nom	Condition	Adresse	Signaux	
0	MS	-	-	GatePC=1, LD.MAR=1, PCMX=00, LD.PC=1	*PC→MAR ; PC+1→PC*
1	MS	-	-	LD.MDR=1, R.W=0	*Mémoire→MDR*
2	MS	-	-	GateMDR=1, LD.IR=1	*MDR→IR*
3	BC	StackFull	5		*Test pour savoir si pile matérielle pleine*
4	BI	-	11		*Aller directement à décalage pile matérielle*
5	MS	-	-	SR1MX=01	*Vider le haut de la pile matérielle dans la pile annexe*
6	MS	-	-	ALUK=00	*Lecture R5, immédiat* *R5+1 (R5+immédiat)*
7	MS	-	-	GateALU=1, LD.REG=1, DRMUX=1	*R5+1→R5*
8	MS	-	-	ALUK=11, GateALU=1, LD.MAR=1	*R5→MAR par ALU*
9	MS	-	-	DIR=1, GateStack=1, LD.MDR=1	*Haut de la pile→MDR*
10	MS	-	-	R.W=1	*Ecriture Mémoire*
11				SR1MUX=00	*Décaler la pile matérielle vers le haut (idem 2.1)* *Lecture Rs1*
12	MS	-	-	ALUK=11, GateALU=1, LatchStack=1	*Rs1→Pile (par l'ALU)*
13	MS	-	-	DIR=1, WriteStack=1	*Décalage pile vers le haut et empilement de la donnée contenue dans LatchStack*
14	BI	-	0		*Retour au début*

2.3.4 Indiquer comment modifier le schéma de la question **Erreur ! Source du renvoi introuvable.** pour générer le signal de condition *StackFull*.

Il faut que le signal soit à 1 lorsque l'élément du haut de la pile est utilisé. On ajoute un registre 1-bit à chaque étage de la pile. Le signal StackFull est fourni par le registre 1-bit de l'élément haut de la pile. Ces registres 1-bit sont connectés comme les registres contenant les données de la pile, exceptés l'élément du haut et celui du bas ; pour ces deux éléments, on s'inspire de la question 2.1.6. L'élément du bas reçoit 1 en entrée à chaque PUSH (lorsque DIR=1) ; celui du haut doit recevoir 0 à chaque POP sauf si la pile annexe n'est pas vide (auquel cas, il doit recevoir 1). On le relie à un nouveau latch 1-bit situé à côté du latch de la pile ; l'écriture dans ce latch 1-bit est contrôlé par un signal LatchStack2, et ce latch reçoit en entrée la négation de la sortie du registre 1-bit de condition z. Le signal LatchStack2 doit être activé à l'étape 6 de l'instruction POP (après stockage du résultat dans le registre 1-bit z), au moment du test de z. Ainsi, si la pile annexe est vide, ce latch 1-bit va recevoir 0, et après décalage de la pile matérielle, le signal StackFull vaudra 0 également. Réciproquement, si la pile annexe n'est pas vide, ce latch va recevoir 1 et StackFull continuera à valoir 1. Pour les instructions PUSH, on fait progressivement monter la valeur 1 à partir de l'élément bas de la pile.



Exercice. Processeur SISC : *Single Instruction Set Computer* (10 points)

On considère un jeu d'instructions ne contenant qu'une seule instruction: *SBN A,B,S*, où *SBN* signifie « *Substract and Branch if Negative* » (Soustrait et Saute si Négatif). L'instruction effectue l'opération suivante:

- *Mem(A) = Mem(A) - Mem(B)*, et
  - si (*Mem(A) < 0*) *PC = PC + S* ;
  - sinon PC = PC + 1 (saut à l'instruction suivante) ;

où *Mem(A)* correspond au contenu de la mémoire à l'adresse *A*, et *PC* au compteur de programme ; le test est effectué *après* la soustraction ; on suppose que la largeur de la mémoire est telle qu'une adresse mémoire correspond à une donnée ou une instruction ; on ne se préoccupe pas non plus de la taille du mot que l'on suppose suffisamment grand pour les calculs ; enfin, on considère qu'un programme se termine quand sa dernière instruction est exécutée.
On suppose que *Mem(0)=1*, et on peut utiliser les adresses 1 à 9 pour stocker des valeurs temporaires. Dans les questions suivantes, on suppose que $10 \leq A, B, C \leq 20$.

1. Ecrire le programme permettant d'effectuer *Mem(A)* ← *0*.

```
        SBN A,A,1   ; Mem(A)=Mem(A)-Mem(A)=0
```

2. Ecrire le programme permettant d'effectuer *Mem(A)* ← *Mem(B)*.

```
        SBN A,A,1   ; Mem(A)=0
        SBN 1,1,1   ; Mem(1)=0
        SBN 1,B,1   ; Mem(1)=-Mem(B)
        SBN A,1,1   ; Mem(A)=-Mem(1)=Mem(B)
```

3. Ecrire le programme permettant d'effectuer *Mem(A)* ← *3*.

```
        SBN A,A,1  ; Mem(A)=0
        SBN 1,1,1  ; Mem(1)=0
        SBN 1,0,1  ; Mem(1)=-Mem(0)=-1
        SBN 1,0,1  ; Mem(1)=Mem(1)-Mem(0)=-2
        SBN 1,0,1  ; Mem(1)=Mem(1)-Mem(0)=-3
        SBN A,1,1  ; Mem(A)=-Mem(1)=3
```

4. Ecrire le programme permettant d'effectuer *Mem(A) ← Mem(B) + Mem(C)*.

```
        SBN A,A,1  ; Mem(A)=0
        SBN 1,1,1  ; Mem(1)=0
        SBN 1,B,1  ; Mem(1)=-Mem(B)
        SBN 2,2,1  ; Mem(2)=0
        SBN 2,C,1  ; Mem(2)=-Mem(C)
        SBN A,1,1  ; Mem(A)=-Mem(1)=Mem(B)
        SBN A,2,1  ; Mem(A)=Mem(B)-Mem(2)=Mem(B)+Mem(C)
```

5. Ecrire le programme permettant d'effectuer *Mem(A) ← Mem(B) x Mem(C)* en supposant que *0≤Mem(B)*.

```
On additionne Mem(B) fois Mem(C) à Mem(A).

        SBN A,A,1  ; Mem(A)=0
        SBN 1,1,1  ; Mem(1)=0
        SBN 1,C,1  ; Mem(1)=-Mem(C)
        SBN 3,3,1  ; Mem(3)=0
BCL     SBN 2,2,1  ; Mem(2)=0
        SBN 2,B,ADD; Mem(2)=-Mem(B), -Mem(B)<0 ? (⇔ Mem(B)>0 ?)
        SBN 3,0,FIN; Aller à FIN (branchement inconditionnel)
ADD     SBN A,1,1  ; Mem(A)=Mem(A)-Mem(1)=Mem(A)+Mem(C)
        SBN B,0,1  ; Mem(B)=Mem(B)-1
        SBN 3,0,BCL; Aller à BCL (branchement inconditionnel)
FIN
```

Annexe

Les informations ci-dessous ont été vues en TD, exceptée la syntaxe des microinstructions ; on rappelle que le signal DRMX a été modifié.

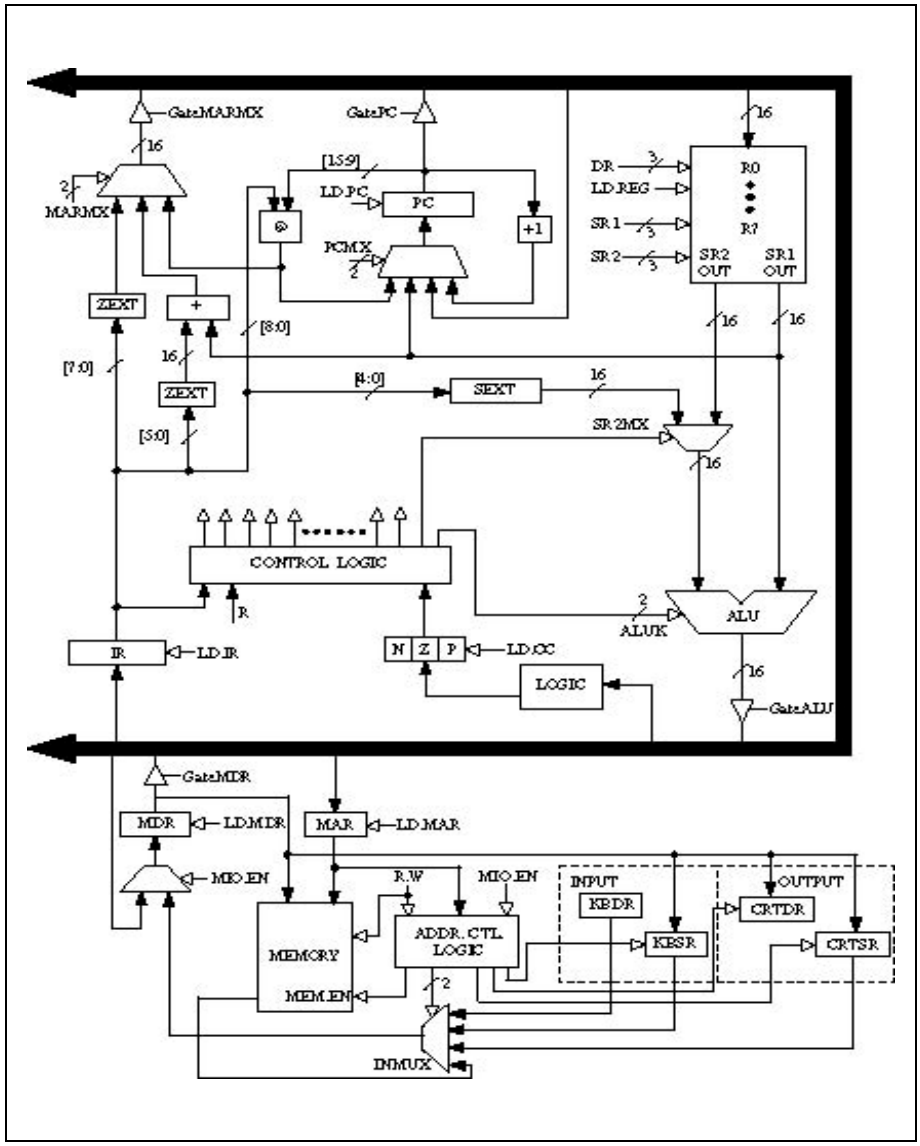
- On rappelle le format et la signification des principales instructions du LC-2 :

Syntaxe assembleur *Signification*	*Bit :*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
ADD Rd, Rs1, Rs2 *Rd←Rs1+Rs2*	ADD	0001				Rd			Rs1			0	00		Rs2		
ADD Rd, Rs1, immédiat *Rd←Rs1+SEXT(immédiat)*	ADD	0001				Rd			Rs1			1	immédiat				
AND Rd, Rs1, Rs2 *Rd←ET(Rs1,Rs2)*	AND	0101				Rd			Rs1			0	00		Rs2		
AND Rd, Rs1, immédiat *Rd←ET(Rs1,SEXT(immédiat))*	AND	0101				Rd			Rs1			1	immédiat				
BRnzp label ou immédiat *Saut si OU(ET(n,N),ET(z,Z),ET(p,P))=1* *adresse=PC[15:9]@ZEXT(immédiat)* *PC←adresse*	BR	0000				n	z	p	immédiat								
JMP/JSR label ou immédiat *Si L=1 (JSR) R7←PC* *adresse=PC[15:9]@ZEXT(immédiat)* *PC←adresse*	JMP/JSR	0100				L	00		immédiat								
JMPR/JSRR label ou immédiat *Si L=1 (JSRR) R7←PC* *adresse=Rs1+ZEXT(immédiat)* *PC←adresse*	JMPR/JSRR	1100				L	00		Rs1			immédiat					
LD Rd, label ou immédiat *adresse=PC[15:9]@ZEXT(immédiat)* *Rd←Mémoire(adresse)*	LD	0010				Rd			immédiat								
LDR Rd, Rs1, immédiat *adresse=Rs1+ZEXT(immédiat)* *Rd←Mémoire(adresse)*	LDR	0110				Rd			Rs1			immédiat					
LEA Rd, label ou immédiat *Rd← PC[15:9]@ZEXT(immédiat)*	LEA	1110				Rd			immédiat								
NOT Rd, Rs1 *Rd←NOT(Rs1)*	NOT	1001				Rd			Rs1			111111					
RET *PC←R7*	RET	1101				000000000000											
ST Rs1, label ou immédiat *adresse=PC[15:9]@ZEXT(immédiat)* *Rs1→Mémoire(adresse)*	ST	0011				Rs1			immédiat								
STR Rs1, Rs2, immédiat *adresse=Rs2+ZEXT(immédiat)* *Rs1→Mémoire(adresse)*	STR	0111				Rs1			Rs2			immédiat					

où SEXT signifie *Signed Extension* (extension signée à 16 bits en complément à 2), et ZEXT signifie *Zero Extension* (extension non signée à 16 bits) , @ est l'opérateur de concaténation (de deux champs de bits), IR[x:y] dénote les bits *x* à *y* du registre instruction (IR).

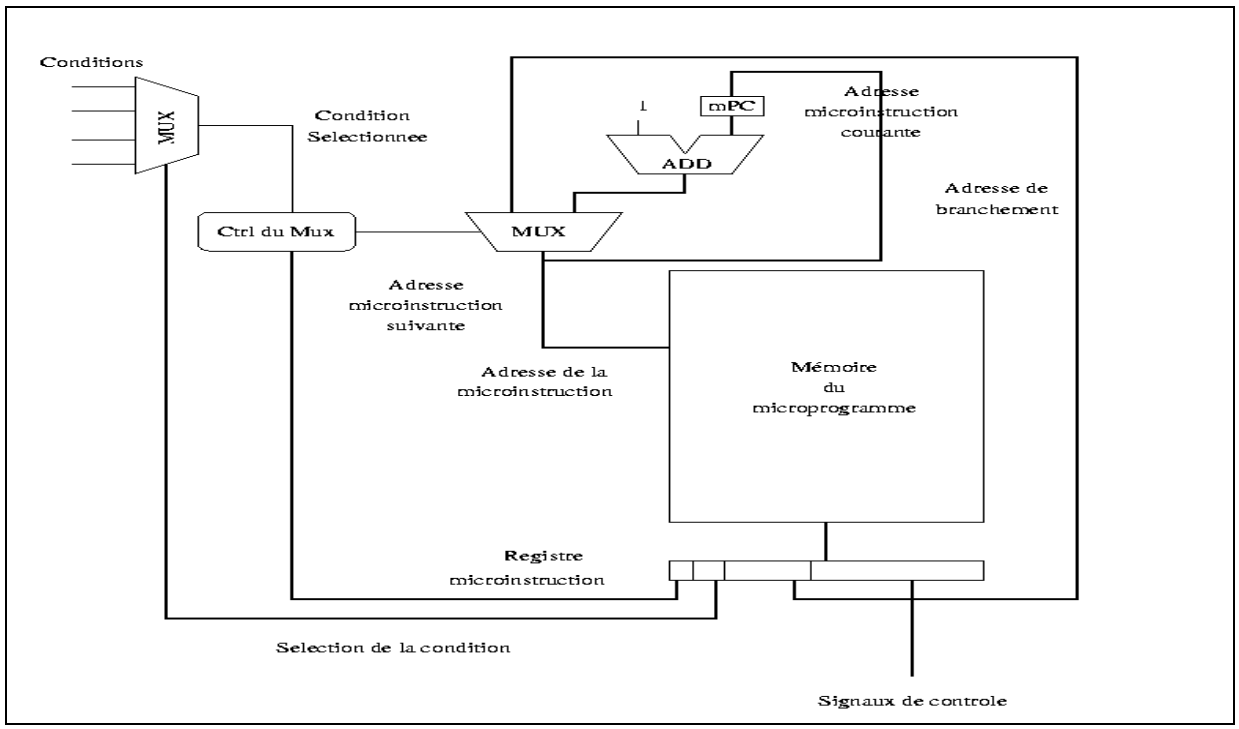
- On rappelle la structure du LC-2 et le rôle des signaux de contrôle :

- LD.MAR, LD.MDR, LD.IR, LD.REG, LD.CC, et LD.PC commandent l'écriture dans les divers registres et latchs du LC-2 (1 si écriture, 0 sinon) ;
- GatePC, GateMDR, GateALU et GateMARMX commandent l'écriture sur le bus ;
- R.W : 0 pour une lecture et 1 pour une écriture en mémoire.
- ALUK (2 bits) : 00 pour ADD, 01 pour AND, 10 pour NOT, 11 pour que l'opérande de droite (SR1) traverse l'ALU sans calcul ;
- PCMX (2 bits) : 00 pour sélectionner PC+1, 01 pour le bus, 10 pour la sortie du banc de registres, 11 pour la concaténation PC[15:9]@IR[8:0] ;
- MARMX (2 bits) : 00 pour sélectionner IR[7:0], 01 pour l 'addition d'un registre et IR[5:0], 10 pour la concaténation PC[15:9]@IR[8:0], 11 est inutilisé ;
- SR1MX (2 bits) : multiplexeur non représenté sur la figure et alimentant le signal SR1 (numéro registre source 1) : 00 pour sélectionner IR[11:9], 01 pour IR[8:6], 10 et 11 inutilisés ;
- DRMX (1 bit) : multiplexeur non représenté sur la figure et alimentant le signal DR (numéro registre destination) : 0 pour sélectionner IR[11:9], 1 pour IR[8:6] ;
- SR2MX (1 bit) : multiplexeur permettant de choisir entre SR2 (registre source 2) et l'immédiat ; ce signal est toujours égal à la valeur du bit IR[5], il est inutile de spécifier la valeur de ce signal ;
- on ignore le rôle des autres signaux.



- On rappelle que les différentes étapes possibles de l'exécution d'une instruction dans le LC-2 sont les suivantes (chacune requiert un cycle processeur) :
  - Chargement de l'instruction (3 étapes, donc 3 cycles) :
    - Envoi adresse instruction (PC→MAR) ; on effectue également PC←PC+1
    - Chargement instruction (Mémoire→MDR)
    - Stockage instruction (MDR→IR)
  - Décodage et lecture des opérandes (lecture des registres Rs1, Rs2 ; appelés SR1, SR2 sur le schéma du processeur)
  - Calcul d'adresse (les différents calculs d'adresse destinés aux instructions d'accès à la mémoire sont sélectionnés avec MARMX)
  - Accès mémoire : l'accès mémoire pour les données se décompose en 3 étapes (3 cycles) comme l'accès mémoire pour les instructions :
    - Envoi adresse de la donnée (dans le latch MAR) ; en cas d'écriture, il faut également envoyer la donnée dans MDR dans une autre étape
    - Accès mémoire (écriture, ou lecture et stockage de la donnée dans le latch MDR)
    - Récupération de la donnée dans le processeur à partir de MDR en cas de lecture
  - Exécution (ALU)
  - Ecriture du résultat (dans le banc de registres)

- On rappelle la structure générale du microséquenceur, on donne une syntaxe pour les microinstructions et un exemple de microprogramme (contrôle de l'instruction ADD) :



- Une microinstruction comporte 4 champs : opcode, numéro de condition, adresse de branchement, signaux de contrôle.
- Il existe 4 types de microinstruction : microinstruction simple (MS), branchement inconditionnel (BI), branchement conditionnel (BC), microinstruction d'arrêt (STOP).
- Pour le microséquenceur du LC-2, on considère ici que l'on dispose de 6 signaux de conditions correspondant aux 5 bits de poids fort du registre d'instruction (IR) et au bit *z* du registre code-condition (CC); ces signaux sont appelés I_{15}, I_{14}, I_{13}, I_{12}, I_{11}, *z*.
- **Syntaxe à utiliser** : on ne représentera pas une microinstruction avec un format binaire mais en utilisant le format suivant :

nom microinstruction	nom condition	numéro microinstruction	signaux activés

où *nom microinstruction* correspond aux acronymes MS, BI, BC et STOP, *nom condition* aux bits I_{15}, I_{14}, I_{13}, I_{12}, I_{11}, *z*, *numéro microinstruction* à l'adresse en mémoire ROM de la microinstruction de destination (en cas de branchement) exprimée en décimal (la première microinstruction sera stockée à l'adresse 0), et *signaux activés* aux signaux qui sont activés par la microinstruction en ignorant les signaux sans importance pour cette microinstruction.

- **Exemple de microprogramme** : contrôle de l'instruction ADD du LC-2 :

Nom	Condition	Adresse	Signaux	
MS	-	-	GatePC=1, LD.MAR=1, PCMX=00, LD.PC=1	*PC→MAR ; PC+1→PC*
MS	-	-	LD.MDR=1, R.W=0	*Mémoire→MDR*
MS	-	-	GateMDR=1, LD.IR=1	*MDR→IR*
MS	-	-	SR1MX=01	*SR1,SR2→ALU*
MS	-	-	ALUK=00	*Calcul*
MS	-	-	GateALU=1, LD.REG=1, LD.CC=1	*ALU→DR*