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.
- **Faire le problème et l'exercice sur des copies séparées**
- Il est impératif de commenter les programmes en assembleur et les microprogrammes ; normalement, presque chaque instruction ou microinstruction doit être suivie d'un commentaire.

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.

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.

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).

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.

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.

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.

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.

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.

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.

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 ?

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. 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. De même, lorsque des modifications de l'architecture sont demandées, on privilégiera des modifications simples, le coût et la performance étant ici des critères secondaires.

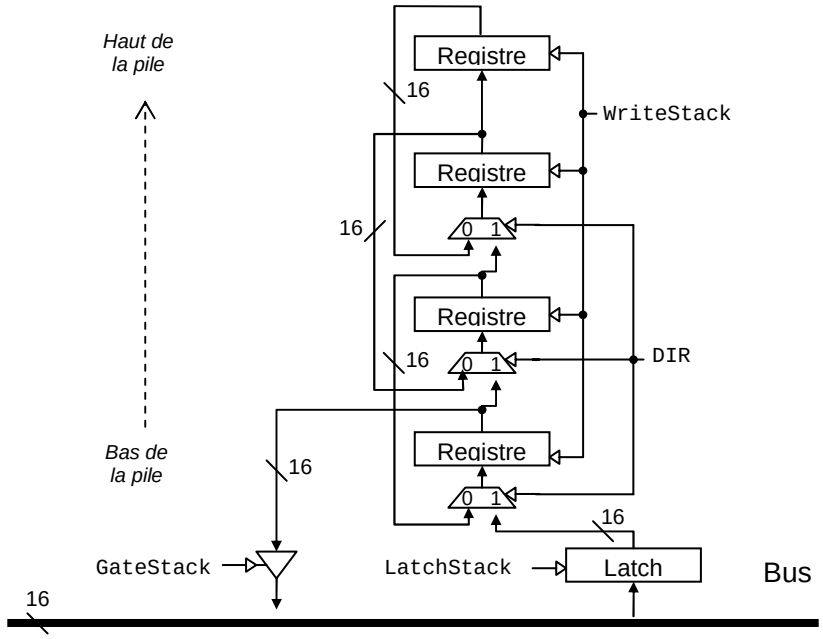


Figure 1. Implémentation de la pile.

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).

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.

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

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.

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.

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 ce pointeur est toujours stocké dans le registre R5. 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 2.2.

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

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

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 capacité de la pile matérielle à la suite d'un PUSH, 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 dans R5 ; R5 contient l'adresse en mémoire de l'élément situé au bas de cette pile annexe (le dernier élément de la pile annexe) ; 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.

2.3.2 Indiquer comment modifier le schéma de la Figure 1 pour permettre le mécanisme de pile annexe. On répondra notamment avec un schéma correspondant à une version modifiée de la Figure 1.

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

2.3.4 Indiquer comment modifier le schéma de la question 2.3.2 pour générer le signal de condition *StackFull*.

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 $Mem(A) \leftarrow 0$.
2. Ecrire le programme permettant d'effectuer $Mem(A) \leftarrow Mem(B)$.
3. Ecrire le programme permettant d'effectuer $Mem(A) \leftarrow 3$.
4. Ecrire le programme permettant d'effectuer $Mem(A) \leftarrow Mem(B) + Mem(C)$.
5. Ecrire le programme permettant d'effectuer $Mem(A) \leftarrow Mem(B) \times Mem(C)$ en supposant que $0 \leq Mem(B)$.

Annexe

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

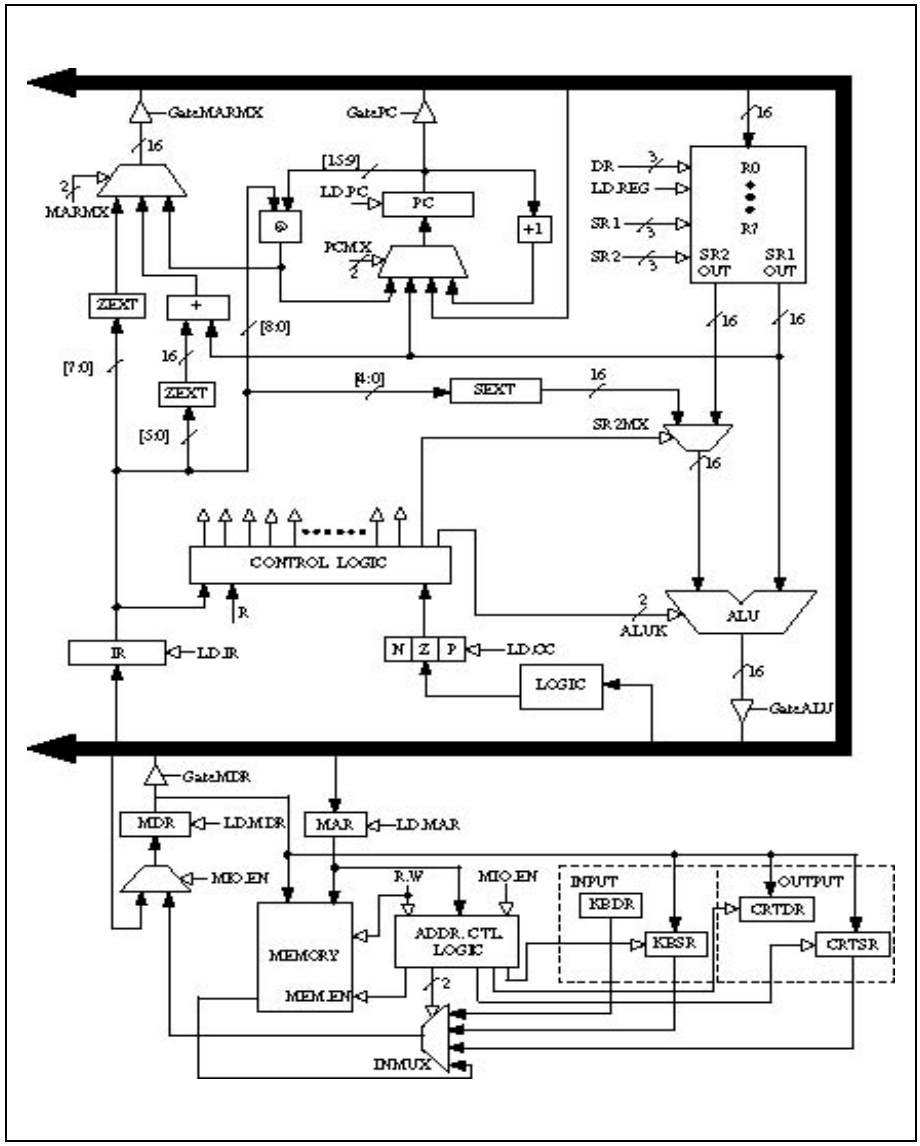
- On rappelle le format et la signification des principales instructions du LC-2 :

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).

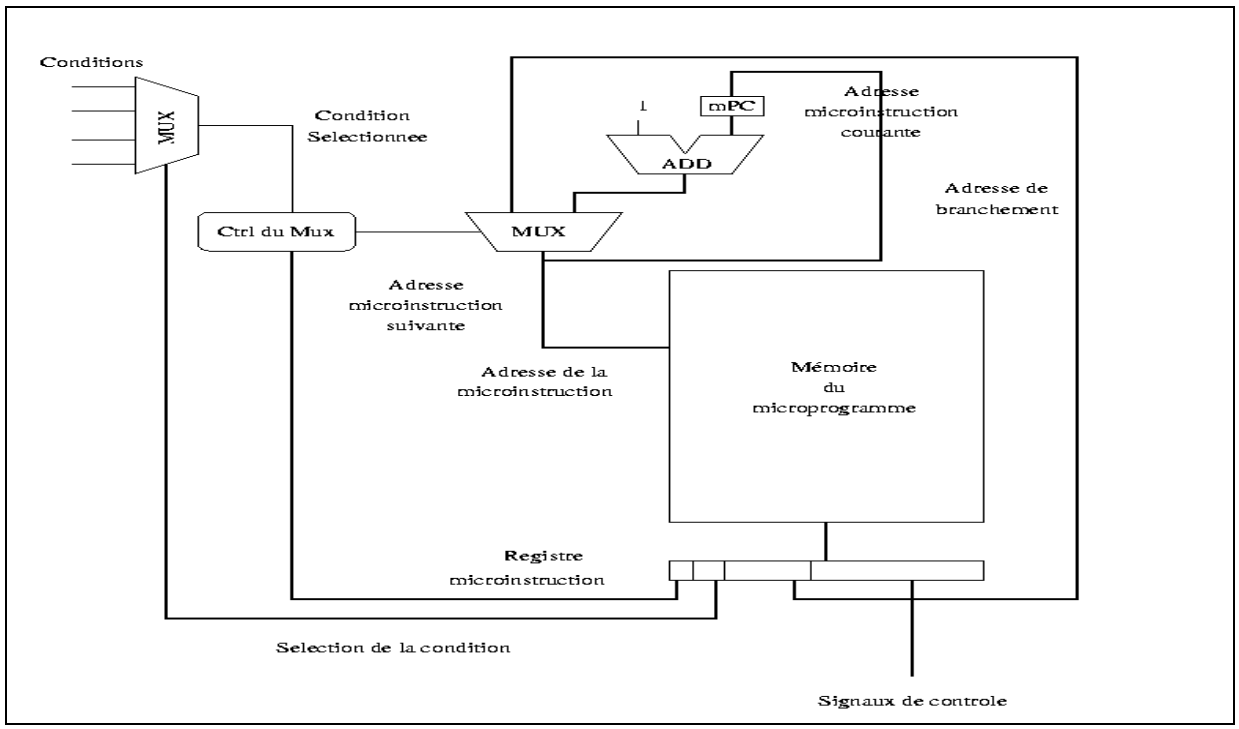
- On rappelle la structure du LC-2 et le rôle des signaux de contrôle :

- 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*