Examen d'Architecture des Ordinateurs
Majeure 1 – École Polytechnique

2006–2007

- L'examen dure 3 heures.
- Le sujet comporte 8 pages dont des rappels sur le LC-2.
- Tous documents autorisés.
- Le barème est donné à titre indicatif, il sert surtout à évaluer le poids respectif des sections.
- L'examen contient 2 exercices distincts que vous devez rendre sur **2 COPIES SÉPARÉES**.
- Il est impératif de commenter les programmes en assembleur et les microprogrammes ; la plupart des instructions doivent être suivies d'un commentaire permettant de comprendre leur rôle.

Exercice 1 - Machine Virtuelle (12 points)

On suppose que de nombreux systèmes embarqués ont été conçus à l'aide d'un microprocesseur 8-bits, appelé Pico, et donc que beaucoup de programmes ont été développés pour ce microprocesseur ; ces programmes ont été écrits en assembleur. La fabrication de Pico a été arrêtée, mais ces systèmes embarqués continuent à être vendus ; il faut donc remplacer le microprocesseur Pico par un autre microprocesseur du marché. On choisit le LC-2 ; cependant, le jeu d'instructions du Pico est différent de celui du LC-2, mais il serait trop lourd de convertir tous les programmes écrits en assembleur Pico en assembleur LC-2. On décide donc de créer une *machine virtuelle* Pico, qui va tourner sur le LC-2, et qui sera capable d'exécuter des programmes assembleur Pico directement sur le LC-2, sans les réécrire. Le but de l'exercice est d'écrire cette machine virtuelle.

Une machine virtuelle reproduit, de façon logicielle, le fonctionnement matériel d'un processeur. Elle opère instruction par instruction, et pour chaque instruction effectue les étapes suivantes :
- chargement de l'instruction
- décodage (récupération des opérandes, analyse de l'opcode)
- exécution (calcul ou accès mémoire)
- écriture du résultat

En outre, la machine virtuelle gère un PC (*Program Counter*) virtuel, ainsi qu'un banc de registres virtuel. Ces différentes informations sont stockées dans la mémoire du processeur hôte (ici, le LC-2).

Quelques détails supplémentaires sur le Pico. Il s'agit d'un processeur 8-bits (taille des instructions et des données), donc la mémoire adressable est de 256 octets. Les instructions et leur format sont indiqués Figure 1. Les bits 7 et 6 contiennent l'opcode ; le bit 5 est parfois également utilisé comme bit d'opcode. Le Pico utilise 4 registres 8-bits nommés `pr0` à `pr3`. Les opérations arithmétiques sont effectuées en complément à 2.

Instruction	7	6	5	4	3	2	1	0
`pADD pRs3, pRs2, pRs1` (`pRs3` ← `pRs2 + pRs1`)	0	0	pRs3		pRs2		pRs1	
`pNAND pRs3, pRs2, pRs1` (`pRs3` ← `NOT(AND(pRs2, pRs1))`)	0	1	pRs3		pRs2		pRs1	
`pLOAD pRs2, pRs1` (`pRs2` ← `MEM(pRs1)`)	1	0	0	0	pRs2		pRs1	
`pSTORE pRs2, pRs1` (`pRs2` → `MEM(pRs1)`)	1	0	1	0	pRs2		pRs1	
`pBNZ pRs2, pRs1` (branch to `pRs1` if `pRs2` ≠ `0`)	1	1	0	0	pRs2		pRs1	
`pSET val, pRs1` (`pRs1 = val`)	1	1	1	val			pRs1	

FIG. 1 – Format des instructions du microprocesseur Pico

Dans la machine virtuelle qui tourne sur le LC-2, on considère que le PC du Pico, appelé `pPC` pour éviter les confusions avec le PC du LC-2, sera stocké dans le registre `R4` du LC-2. La mémoire du Pico sera stockée à partir de l'adresse `0x4000` dans la mémoire du LC-2. Chaque octet de la mémoire du Pico sera stocké dans les bits de poids faible d'une case mémoire de taille 16-bits de la mémoire du LC-2 (on perd donc 8 bits à chaque fois, mais c'est sans importance). Les 4 registres du Pico seront stockés aux adresses suivantes de la mémoire du LC-2 :

- `pr0` à l'adresse `0x3FFC`
- `pr1` à l'adresse `0x3FFD`
- `pr2` à l'adresse `0x3FFE`
- `pr3` à l'adresse `0x3FFF`

Le programme de la machine virtuelle lui-même sera stocké à partir de l'adresse `0x3000`. Ce programme sera notamment suivi des labels et directives suivants :

- `mem_offset .FILL 0x4000`
- `regs_offset .FILL 0x3FFC`

Les questions ci-dessous ont pour but de vous aider à décomposer votre programme de machine virtuelle. Il est impératif de **strictement** respecter les indications, notamment de nommage. Pour les appels de procédure, on n'utilisera pas de pile (bien qu'il s'agisse de la méthode la plus employée) afin de simplifier le programme ; on se contentera de sauver les registres à préserver, au début de la procédure **appelée**, dans un emplacement mémoire statique (nommer ces emplacements mémoire `bak_Rx_<éventuel label complémentaire de votre choix>` où `x` est le numéro du registre à sauver).

Note importante : Le terme *fonction* ci-dessous indique un ensemble d'instructions appelé par `JSR` et terminant par `RET`. Le terme *sous-programme* correspond simplement à plusieurs instructions d'un programme, il ne s'agit *pas* d'une fonction appelée par `JSR`.

Question 1.1

Ecrire le sous-programme d'initialisation de la machine virtuelle du Pico. Ce sous-programme met tous les registres du Pico à la valeur 0, et initialise `pPC` en sachant que la première instruction du programme Pico à exécuter est toujours stockée à l'adresse `0x00` de la mémoire du Pico.

Réponse

Voir label `init_mv`.

Question 1.2

Ecrire le sous-programme de chargement d'une instruction Pico. Le registre `R2` sera utilisé comme registre instruction (`IR`).

On supposera que ce sous-programme est placé en mémoire LC-2 immédiatement après le sous-programme de la question 1.1, donc la valeur des registres en début de sous-programme correspond à celle en fin du sous-programme précédent.

Réponse

Voir label `fetch`.

Question 1.3

On veut écrire une fonction permettant de décaler à gauche de i bits un registre 16-bits. Lors de l'appel de cette fonction, on supposera que i est stocké dans le registre `R0`, la valeur 16-bits dans `R2` et que le résultat est renvoyé dans `R3`.

On supposera que cette fonction est placée en mémoire LC-2 après l'ensemble des sous-programmes décrits dans les questions ci-dessus et ci-dessous.

Réponse

Voir label `f_shift`.

Question 1.4

On veut écrire une fonction permettant de passer j bits de poids fort d'un registre 16-bits en poids faible et d'annuler les autres bits. Par exemple, si $j = 4$ pour `1011000001111000`, on obtiendra `0000000000001011`.

Lors de l'appel de cette fonction, on supposera que j est stocké dans le registre `R1`, la valeur 16-bits dans `R3` et que le résultat est renvoyé dans `R3`.

On supposera que cette fonction est placée en mémoire LC-2 après la fonction de la question 1.3.

Réponse

Voir label `f_swap`.

Question 1.5

On commence le décodage des instructions. Dans cette question, on va charger les *adresses* en mémoire LC-2 où sont stockés les registres `pRs1` et `pRs2` (voir Figure 1), et les placer respectivement dans les registres `R5` et `R6` du LC-2.

Dans quels cas et pourquoi cette façon de décoder est inefficace ?

On supposera que ce sous-programme est placé en mémoire LC-2 immédiatement après le sous-programme de la question 1.2, donc la valeur des registres en début de sous-programme correspond à celle en fin du sous-programme précédent.

Réponse

`pSet` n'a qu'un registre source, donc débugger `pRs2` dans ce cas est inutile.
Voir label `di_rs12`.

Question 1.6

On continue le décodage des instructions, on charge les opérandes et on exécute les instructions. On traite pour le moment uniquement les instructions ALU (`pNAND` et `pADD`).

On commencera par mettre l'opcode dans le registre `R3`, et on écrira le sous-programme de décodage de l'opcode. Ce sous-programme identifiera d'abord s'il s'agit d'une instruction ALU, et ensuite laquelle des instructions ALU. Pour le moment, on laissera le résultat du calcul dans le registre `R6` du LC-2, on ne le réécrira pas dans le registre du Pico. On terminera le sous-programme en sautant vers le sous-programme d'écriture du résultat qui commence au label `wb_alu`.

On supposera que ce sous-programme est placé en mémoire LC-2 immédiatement après le sous-programme de la question 1.5, donc la valeur des registres en début de sous-programme correspond à celle en fin du sous-programme précédent.

Réponse

Voir label `di_opcode`.

Question 1.7

Ecrire le sous-programme d'écriture du résultat pour les instructions ALU qui commence au label `wb_alu`. Le résultat est contenu dans le registre `R6`.

On supposera que ce sous-programme est placé en mémoire LC-2 immédiatement après le sous-programme de la question 1.6, donc la valeur des registres en début de sous-programme correspond à celle en fin du sous-programme précédent.

Réponse

Voir label `wb_alu`.

Question 1.8

Continuer le décodage des instructions. Ecrire maintenant le sous-programme qui identifie les instructions mémoire (`pLOAD` ou `pSTORE`), puis décoder et exécuter ces instructions.

On supposera que ce sous-programme est placé en mémoire LC-2 immédiatement après le sous-programme de la question 1.7, donc la valeur des registres en début de sous-programme correspond à celle en fin du sous-programme précédent.

Réponse

Voir label `di_1x`.

Question 1.9

Continuer le décodage des instructions. Ecrire maintenant le sous-programme qui décode puis exécute l'instruction `pBNZ`.

On supposera que ce sous-programme est placé en mémoire LC-2 immédiatement après le sous-programme de la question 1.8, donc la valeur des registres en début de sous-programme correspond à celle en fin du sous-programme précédent.

Réponse

Voir label `di_11`.

Question 1.10

Que faudrait-il changer à la question précédente si l'adressage du branchement était relatif et non absolu ? Dans les questions suivantes, on reste sur un adressage absolu pour le branchement.

Réponse

En plus du calcul qui est différent, il faut penser que le `pPC` a déjà été incrémenté juste après le chargement. Il faut donc le décrémenter.

Question 1.11

Terminer le décodage avec l'instruction `pSET`. La valeur immédiate `val` stockée dans le champ de l'instruction est une valeur signée en complément à 2.

On supposera que ce sous-programme est placé en mémoire LC-2 immédiatement après le sous-programme de la question 1.9, donc la valeur des registres en début de sous-programme correspond à celle en fin du sous-programme précédent.

Réponse

Voir label `pSET`.

```
;-------------------------------------------------------------------
.ORIG x3000
init_mv LEA  R0, regs_offset ; mise à 0 des registres
AND  R1, R1, #0 ; R1 = 0
STR  R1, R0, #0 ; pr0 = 0
STR  R1, R0, #1 ; pr0 = 1
STR  R1, R0, #2 ; pr0 = 2
STR  R1, R0, #3 ; pr0 = 3
LEA  R4, mem_offset ; initialisation pPC

;-------------------------------------------------------------------

fetch LDR  R2, R4, #0 ; chargement inst. adresse pPC dans IR
ADD  R4, R4, #1 ; pPC = pPC + 1

;-------------------------------------------------------------------

func_extract ST  R2, bak_R2_extract ; sauver R2
ST  R5, bak_R5_extract ; sauver R5
AND  R5, R5, #0
ADD  R5, R5, #15
NOT  R5, R5
ADD R5, R5, #1 ; R5 = -15 (complément à 2)
ADD R3, R0, R5 ; R3 = ifort - 15
ifort_to_15 ADD R3, R3, #1
BRn init_extract
ADD R2, R2, R2 ; décaler R2 à gauche
JMP  ifort_to_15
init_extract NOT R5, R1
ADD R5, R5, #1
ADD R5, R5, R0 ; R5 = ifaible - ifort (<=0)
AND R3, R3, #0 ; R3 = 0
extract ADD R2, R2, #0
BRzp bit0 ; bit poids fort R2 = 0
ADD R3, R3, #1 ; transférer 1 dans bit poids faible R3
bit0 ADD R3, R3, R3 ; décaler R3 à gauche
ADD R2, R2, R2 ; décaler R2 à gauche
ADD R5, R5, #1 ; incrémenter R5
BRn extract ; continuer extraction
LD  R2, bak_R2_extract ; restaurer R2
LD  R5, bak_R5_extract ; restaurer R5
RET ; R3 contient le champ extrait en bits de poids faible

bak_R2_extract .BLKW 1
bak_R5_extract .BLKW 1

;-------------------------------------------------------------------

decode_rs12 AND R0, R0, #0
ADD R0, R0, #1
AND R1, R1, #0
JSR func_extract ; R3 contient numéro pRs1
ADD R5, R5, #0
ADD R5, R5, R3 ; R5 contient numéro pRs1
decode_rs2 ADD R0, R0, #2 ; R0 = 3
ADD R1, R1, #2 , R1 = 2
JSR func_extract ; R3 contient numéro pRs2
ADD R6, R6, #0
ADD R6, R6, R3 ; R6 contient numéro pRs2
```

```
;-----------------------------------------------------------------

decode_opcode ADD R0, R0, #4 ; R0 = 7
ADD R1, R1, #4 ; R1 = 6
JSR func_extract ; R3 contient l'opcode
ADD R3, R3, #0
BRn decode_1x
; decode_0x correspond aux opérations ALU, on charge les opérandes
decode_0x LDR R5, R5, #0 ; R5 = pRs1
LDR R6, R6, #0 ; R6 = pRs2
ADD R3, R3, R3 ; décaler R3 à gauche
BRn pnand
padd ADD R6, R6, R5 ; R6 = pRs2 + pRs1
JMP wb_alu
; inutile annuler 8 bits poids fort
pnand AND R6, R6, R5 ; R6 = AND (pRs2, pRs1)
NOT R6, R6 ; R6 = NAND (pRs2, pRs1)
; inutile annuler 8 bits poids fort
JMP wb_alu

;-----------------------------------------------------------------

wb_alu ADD R0, R0, #-2 ; R0 = 5
ADD R1, R1, #-2 ; R1 = 4
JSR func_extract ; R3 = numéro de pRs3
LEA R5, regs_offset
ADD R5, R5, R3 ; R5 = adresse pRs3
STR R6, R5, #0 ; pRs3 = résultat (R6)
JMP fetch ; fin exécution, retour début

;-----------------------------------------------------------------


decode_1x ADD R3, R3, R3 ; décaler R3 à gauche
BRn decode_11
; ici, c'est pLOAD ou pSTORE
decode_10 LEA R0, mem_offset
ADD R5, R5, R0 ; R5 = adresse(mem_offset + pRs1)
LEA R0, regs_offset
ADD R0, R0, R6 ; R0 = adresse pRs2
ADD R3, R3, R3 ; décaler R3 à gauche
BRn pSTORE
pLOAD LDR R5, R5, #0 ; R5 = MEM(pRs1)
STR R5, R0, #0 ; pRs2 = MEM(pRs1)
JMP fetch
pSTORE LDR R6, R6, #0 ; R6 = pRs2
STR R6, R5, #0 ; MEM(pRs1) = pRs2
JMP fetch

;-----------------------------------------------------------------

decode_11 ADD R3, R3, R3 ; décaler R3 à gauche
BRn pSET
pBNZ LDR R6, R6, #0 ; R6 = pRs2 (condition branchement)
LD R0, mask_0_7
AND R6, R6, R0 ; annuler 8 bits poids fort
; nécessaire pour que le test (pRs2 == 0 ?) soit correct
BRz fetch ; branchement non pris
LDR R5, R5, #0 ; R5 = pRs1 (adresse cible branchement)
LEA R0, mem_offset
```

```
ADD R4, R5, R0 ; pPC = adresse(mem_offset + pRs1)
JMP fetch

;-------------------------------------------------------------------

pSET ADD R0, R0, #-3 ; R0 = 4
ADD R1, R1, #-4 ; R1 = 2
JSR func_extract ; R3 = val; note: tous les bits R3 à 0 sauf 3 poids faible
LD R0, mask_2
AND R0, R3, R0 ; masquer tous les bits sauf 3ème (bit signe)
BRz wb_set
LD R0, complement_12
ADD R0, R3, R0 ; mettre tous les bits poids fort à 1 sauf 3 poids faible
LEA R1, regs_offset
ADD R1, R1, R5 ; R1 = adresse pRs1
STR R0, R1, #0 ; pRs1 = SEXT(val);
JMP fetch
mask_2 .FILL x0004
complement_12 .FILL xFFF8

;-------------------------------------------------------------------
```

Exercice 2 - Protection mémoire dans le LC-2 (8 points)

La robustesse des systèmes d'exploitation repose en grande partie sur la capacité du processeur à interdire l'accès à certaines zones mémoire aux processus non autorisés. Ainsi, sous UNIX, un processus *utilisateur* peut accéder à son espace mémoire privé, mais ni à l'espace mémoire des autres processus ni à celui du *noyau* du système.

Ce mécanisme de *protection mémoire* nécessite ;
- une notion de *privilège*, le noyau s'exécutant à un niveau plus privilégié que les processus utilisateurs ;
- un partitionnement de la mémoire en *pages* contiguës de taille 2^n.

On souhaite étendre le LC-2 avec une protection mémoire simplifiée.

Question 2.1

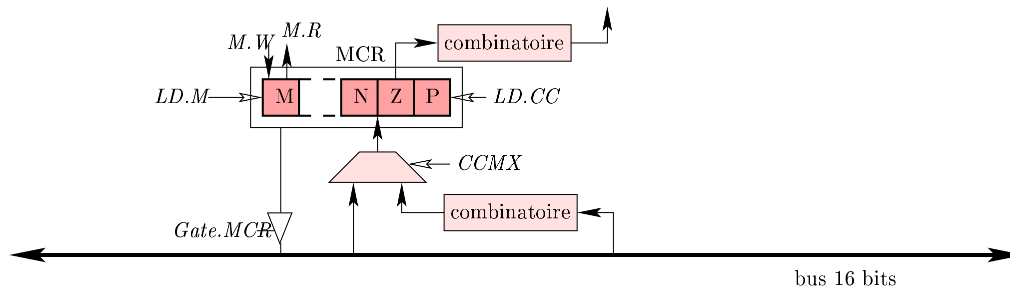
On s'intéresse d'abord à la gestion du niveau de privilège. Notre LC-2 étendu possède un registre d'état de 16 bits, `MCR` (*machine control register*) structuré de la manière suivante :
- les bits 0 à 3 correspondent aux codes de condition `N`, `Z` et `P`, respectivement ;
- le bit 8, appelé `M` est le niveau (ou mode) de privilège courant ;
- les autres bits (certains sont définis dans le LC-2 standard) sont ignorés dans cet exercice.

Les appels système et interruptions agissent sur le niveau de privilège :
- `M` $= 0$ indique que l'exécution est privilégiée, on dit aussi *mode superviseur* ;
- `M` $= 1$ indique que l'exécution n'est pas privilégiée, on dit aussi *mode utilisateur* ;
- l'instruction `TRAP` <*vecteur*> (appel système) est modifiée pour sauvegarder le `PC` sur la pile, dont le pointeur est `R6`, voir rappels sur le LC-2), au lieu de le transférer dans `R7` ; l'instruction sauvegarde également le registre d'état `MCR`, puis bascule `M` à 0 ; ensuite, l'adresse du code de l'appel système proprement dit est lue à l'adresse <*vecteur*> et transférée dans le `PC` (par exemple, `TRAP x25` effectue `PC`←`Mem[x25]`) ;
- l'instruction `RTI` permet normalement de revenir d'une interruption vers le programme en cours d'exécution au moment où l'interruption s'est produite ; contrairement à `RET`, le PC de retour est stocké sur la pile et non dans `R7` ; ici, `RTI` est désormais utilisée non seulement au retour d'une routine d'interruption, mais également au retour d'un appel système initié par `TRAP` ; elle restaure d'abord `MCR` puis le `PC` à partir de la pile.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`RTI`	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`TRAP` trapvect8	1	1	1	1	0	0	0	0	vecteur d'interruption ou d'appel système 8 bits							

On note que ce mécanisme permet de superposer des appels systèmes et des interruptions, tout en préservant les niveaux de privilège. Par la suite, on ne s'intéressera qu'aux instructions `TRAP` et `RTI` en ignorant les interruptions.

Le chemin de données et les signaux pour la lecture et l'écriture de `MCR` sont décrits dans la figure suivante. Le registre `MCR` est contrôlé par les signaux *LD.M* et *LC.CC* qui agissent sur les sous-registres `M` et `NZP`, par *Gate.MCR* qui permet de transférer `MCR` tout entier sur le bus, et *CCMX* qui permet de choisir entre un chargement de `MCR` depuis le bus — $CCMX = 1$ — ou la mise à jour exclusive de `NZP` — $CCMX = 0$. Les fils *M.R* et *M.W* permettent respectivement de consulter ou de modifier le niveau de privilège.



Indiquez la séquence d'actions de contrôle nécessaire à l'exécution de l'instruction`TRAP` à partir du moment où l'instruction a été placée dans le registre `IR`. Même question pour `RTI`. Pour simplifier, vous n'indiquerez le détail des signaux de contrôle que pour les manipulations du registre `MCR` ; pour les autres opérations vous indiquerez informellement les transferts de registres et les calculs effectués ; par exemple, vous pourrez écrire `R6`←`R6+1` bien que le circuit ne contienne pas tous les multiplexeurs et la constante 1 nécessaire pour effectuer cette opération sur l'ALU (on ne s'intéresse à ce niveau de détails que pour `MCR`).

Réponse

`TRAP` <vecteur> :

1. *MAR←R6 ;*
2. *PC←MDR ;*
3. *écriture mémoire, R6←R6+1 ;*
4. *MAR←R6 ;*
5. *MDR←MCR : Gate.MCR=1 ;*
6. *écriture mémoire, R6←R6+1, LD.M=1, M.W=1 (le changement de niveau de privilège peut s'effectuer plus tard mais pas plus tôt) ;*
7. *MAR←<vecteur> ;*
8. *lecture mémoire ;*
9. *PC←MDR.*

`RTI` :

1. *MAR←R6-1, R6←R6-1 ;*
2. *lecture mémoire ;*
3. *MCR←MDR : LD.M=1, LD.CC=1, CCMX=1 ;*
4. *MAR←R6-1, R6←R6-1 ;*
5. *lecture mémoire ;*
6. *PC←MDR.*

Question 2.2

Le mécanisme de pagination utilise les mêmes pages de taille 512 mots (de 16 bits) que celles de l'adressage direct des instructions `LD`, `JMP`, `BR`, etc. Le numéro de page étant donc sur 7 bits (bits 15 à 9 de `IR`), la mémoire du LC-2 est partitionnée en 128 pages.

Afin de garantir l'isolation des accès effectués par un processus envers les pages des autres processus et du noyau, on étend le processeur avec un registre de 128 bits appelé `PR` (*page register*). Ce registre est *ignoré en mode superviseur*. En revanche, en mode utilisateur, chaque bit de `PR` indique si la page correspondante est accessible pour le processus en cours d'exécution (le bit 0 pour la page d'adresse x0000, le bit 1 pour la page d'adresse x0200, etc.). (On ne s'occupe pas de la mise à jour de `PR` dans cette question.)

L'accés à une page interdite (en mode utilisateur) déclenche une interruption de vecteur `x9` en activant le signal *PF* (*page fault*)

Construisez un circuit combinatoire calculant le signal PF à partir du bus, du signal $LD.MAR$, de PR et M, en faisant en sorte que le signal ne puisse être activé que lors de l'écriture du registre MAR.

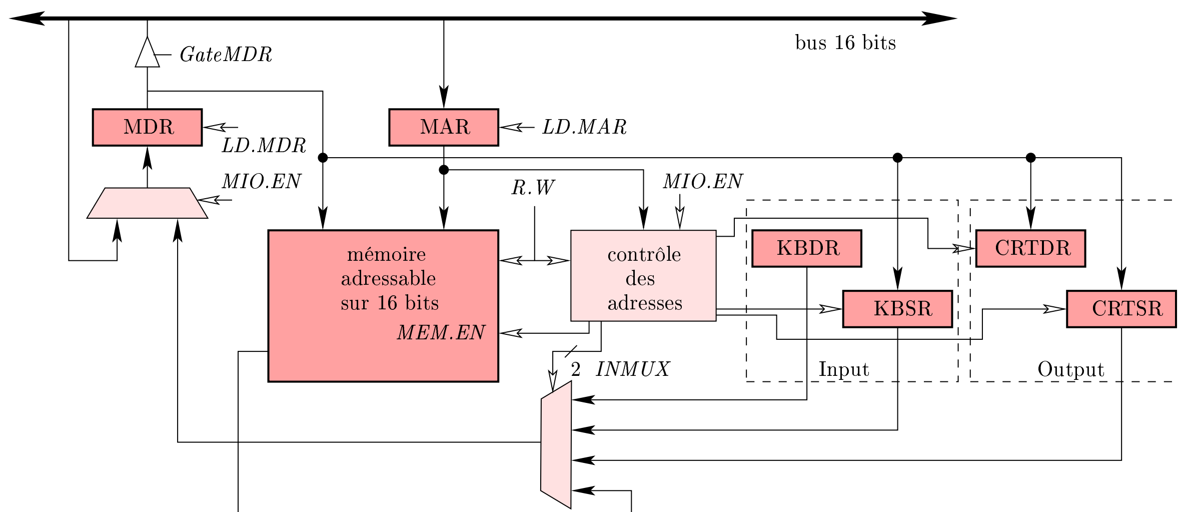
Réponse

$PF = M.R \wedge LD.MAR \wedge \bigvee_{128} \left(Decode_{7\rightarrow 128}(\texttt{bus[15 :9]}) \wedge_{128} \overline{\texttt{PR}}\right)$.

Question 2.3

Le registre PR est accédé via un détournement des accès en mémoire dans l'intervalle d'adresses xFFF0–xFFF7 ; toute lecture/écriture à une adresse dans cet intervalle est traduite en une lecture/écriture au sous-mot de 16 bits correspondant du regitre PR (de la même manière que les registres d'entrées-sorties clavier/écran du LC-2, cf. figure ci-dessous).

Afin de réaliser ce détournement des accès mémoire, on interpose un circuit entre le registre MAR et la mémoire proprement dite. Ce circuit décide s'il s'agit d'un registre mappé en mémoire on d'un accès véritable à la RAM. Construisez un chemin de données reliant PR à MDR, puis réalisez le circuit permettant de décoder les adresses de l'intervalle xFFF0–xFFF7 et d'accéder aux 16 bits de PR correspondant à l'adresse décodée. Vous pourrez considérer que le registre PR est subdivisé en 8 sous-mots de 16 bits, et compléter le diagramme des blocs du système mémoire du LC-2 rappelé dans la figure suivante.



Réponse

Une partie du circuit vérifie que les 12 bits de poids fort de MAR sont à 1 et que le 13ème est à 0, et le résultat est utilisé pour multiplexer l'entrée et la sortie de MDR avec, d'une part l'entrée et la sortie de la mémoire, et d'autre part un décodeur et un multiplexeur commandant l'écriture et la lecture du sous-registre de PR identifié par les 3 bits de poids faible de MAR.

Le reste du chemin de données est le même que celui du mapping en mémoire des registres d'entrées-sorties du LC-2.

Question 2.4

Les pages des adresses x0000 à x0E00, et xF000 à xFE00 sont réservées au noyau du systeème d'exploitation, et les bits 0 à 7 et 120 à 127 correspondants de PR sont toujours nuls. Ainsi, le contenu des vecteurs d'appels système et d'interruption et les registres mappés en mémoire sont inaccessibles aux processus utilisateurs.

Montrez qu'il reste toutefois possible de passer outre le mécanisme de protection par niveau de privilège, dès lors que la pile se situe dans une page accessible au processus utilisateur. Réalisez ainsi une fonction (appelée par JSR) basculant en mode superviseur et poursuivant l'exécution du code appelant en restant dans ce mode. Pour simplifier, vous n'êtes pas obligés de sauvegarder les registres utilisés par cette fonction.

Proposez un remède à cette faille de sécurité, en modifiant l'architecture du processeur afin d'interdire l'accès des processus utilisateurs à toute zone de la pile utilisée par le noyau.

Note : une méthode très similaire au "hack" précédent est souvent utilisé par les noyaux de systèmes d'exploitation, pour basculer à un niveau de privilège plus faible avant d'exécuter un processus utilisateur.

Réponse

```
Hack        ADD R6, R6, #1
            STR R7, R6, #0
            AND R7, R7, #0
            ADD R6, R6, #1
```

```
STR R7, R6, #0
RTI
```

Cette version ne modifie que R7*. Pour protéger ce registre, il faut nécessairement utiliser un* frame pointer*, afin de ranger les valeurs dans l'ordre correct pour* RTI*.*

Pour régler le problème, il suffit par exemple de séparer la pile des processus utilisateurs de la pile du noyau (au niveau superviseur). Techniquement, cela nécessite un mécanisme complexe de sauvegarde de R6 *et de basculement. Le LC-3 implémente ce mécanisme.*

Rappels sur le LC-2

Dans cette section, on rappelle le jeu d'instructions du LC-2, voir Figure 4.

On utilise le schéma du processeur LC-2 (hors gestion des interruptions) de la figure 2.

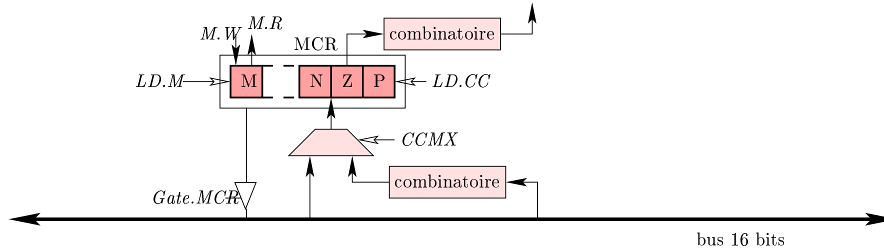


FIG. 2 – Diagramme des blocs du LC-2

Note : les circuits `PC`, `IR`, `MAR`, `MDR`, `BEN`, `N`, `Z` et `P` sont des registres.

Le rôle des signaux de contrôle de la figure 2 est explicité dans la liste suivante.

- $LD.MAR_{/1}$, $LD.MDR_{/1}$, $LD.IR_{/1}$, $LD.REG_{/1}$, $LD.CC_{/1}$ et $LD.PC_{/1}$ commandent l'écriture dans les divers registres du LC-2.
- $LD.BEN_{/1}$ commande l'écriture dans le registre BEN (*branch enable*) ; ce registre vaut 1 lorsque le branchement doit être pris (si l'instruction courante est un branchement conditionnel).
- $GatePC_{/1}$, $GateMDR_{/1}$, $GateALU_{/1}$ et $GateMARMX_{/1}$ commandent les accès en écriture sur le bus.
- $MIO.EN_{/1}$ doit être mis à 1 lorsque l'on souhaite accéder à la mémoire, en lecture ou en écriture.
- $R.W_{/1}$ est mis à 0 pour une lecture et 1 pour une écriture en mémoire.
- $ALUK_{/2}$: 00 pour `ADD`, 01 pour `AND`, 10 pour `NOT`, 11 pour faire « traverser » l'entrée 1 sans calcul.
- $PCMX_{/2}$ sélectionne l'une des quatre entrées du multiplexeur : de *droite* à *gauche*, 00, 01, 10 et 11.
- $MARMX_{/2}$ sélectionne l'une des trois entrées du multiplexeur : de *gauche* à *droite*, 00, 01 et 10 (11 est inutilisé).
- $SR1MX_{/2}$ sélectionne l'une des quatre entrées du multiplexeur : de *haut* en *bas*, 00 et 01 (10 et 11 ne sont pas utilisés).
- $DRMX_{/2}$ sélectionne le registre destination (signal *DR*) à partir de $IR[11:9]$, qui correspond à la valeur 00 (01, 10 et 11 ne sont pas utilisés)
- $SR2MX_{/1}$ est à part : ce signal détermine si la deuxième opérande vient du banc des registres ou du champ immédiat, il est par construction égal au bit 5 du registre d'instruction IR et n'intervient pas dans la microprogrammation.

On dispose du même *microcontrôleur* que celui utilisé au TD 4. Il permet de réaliser le contrôle microprogrammé du LC-2 à l'aide de microinstructions simples et de branchements conditionnels ou non. Il comporte 8 signaux de condition de branchement : le signal $\overline{BEN}$ (qui vaut 1 lorsque le branchement ne doit *pas* être pris) est sélectionné lorsque $Condition_{/3} = 000$; les 5 bits de poids fort du registre d'instruction IR sont respectivement sélectionnés lorsque $Condition_{/3}$ prend des valeurs de 011 à 111. Enfin, les signaux associés aux valeurs 001 et 010 ne sont pas utilisés (pour l'instant). En sortie, ce microcontrôleur est capable de générer les 23 signaux de contrôle du LC-2. Les adresses des microinstructions sont codées sur 8 bits : $J[7:0]$ indique l'adresse de l'instruction suivante en cas de branchement. Les microinstructions sont codées sur 48 bits et leur format détaillé est indiqué figure 3 (les `x` correspondent à des signaux inutilisés, disponibles pour étendre le microcontrôleur) :

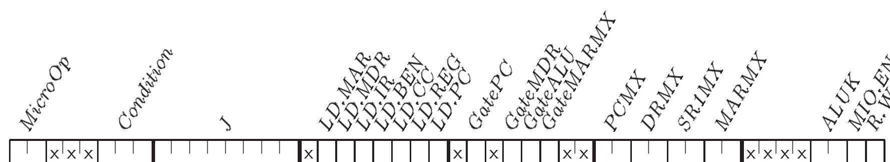


FIG. 3 – Format des mirco-instructions

On rappelle également la liste des instructions du LC-2 (dans sa version simplifiée) en figure 4.

À titre d'exemple, on rappelle le microprogramme pour l'exécution complète de l'instruction `ADD`, vu au TD 4. L'exécution commence à l'adresse 0, et les 4 premières microinstructions ne sont pas spécifiques

Instruction	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`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 : immédiat 5-bits signé				
`AND DR, SR1, SR2` (`DR` ← `AND(SR1, SR2)`)	0	1	0	1	DR			SR1			0	0	0	SR2		
`AND DR, SR1, imm5` (`DR` ← `AND(SR1, SEXT(imm5))`)	0	1	0	1	DR			SR1			1	imm5 : immédiat 5-bits signé				
`NOT DR, SR` (`DR` ← `NOT(SR)`)	1	0	0	1	DR			SR			1	1	1	1	1	1
`BRnzp label` (`PC = PC[15:9]@offset9` `if n.N+z.Z+p.P`)	0	0	0	0	n	z	p	offset 9-bits non signé dans la page courante								
`JMP label` (`PC = PC[15:9]@offset9`)	0	1	0	0	0	0	0	offset 9-bits non signé dans la page courante								
`JSR label` (`R7` ← `PC` and `PC = PC[15:9]@offset9`)	0	1	0	0	1	0	0	offset 9-bits non signé dans la page courante								
`JMPR indexed address` (`PC = BaseR + ZEXT(offset6)`)	1	1	0	0	0	0	0	BaseR			index 6-bits non signé					
`JSR indexed address` (`R7` ← `PC` and `PC = BaseR + ZEXT(offset6)`)	1	1	0	0	1	0	0	BaseR			index 6-bits non signé					
`LEA DR, label` (`DR` ← `PC[15:9]@offset9`)	1	1	1	0	DR			offset 9-bits non signé dans la page courante								
`LD DR, label` (`DR` ← `MEM(PC[15:9]@offset9)`)	0	0	1	0	DR			offset 9-bits non signé dans la page courante								
`LDR DR, indexed address` (`DR` ← `MEM(BaseR + ZEXT(offset6))`)	0	1	1	0	DR			BaseR			index 6-bits non signé					
`ST SR, label` (`SR` → `MEM(PC[15:9]@offset9)`)	0	0	1	1	SR			offset 9-bits non signé dans la page courante								
`STR SR, indexed address` (`SR` → `MEM(BaseR + ZEXT(offset6))`)	0	1	1	1	SR			BaseR			index 6-bits non signé					
`RET` (`PC` ← `R7`)	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0

FIG. 4 – Format des instructions du LC-2

à l'instruction ADD. Les microinstructions effectuant un branchement conditionnel sont associées aux états ????, 0???, 00?? et 000? : à l'aide de 4 microinstructions conditionnelles successives, on saute à l'état $DR \leftarrow SR1 + SR2 / SEXT, CC$ lorsque l'opcode (0001) correspond à l'instruction ADD. Les microinstructions et leurs adresses sont décrites par le tableau de la figure 5 (toutes les valeurs sont en binaire, sauf l'état) :

état	adresse	signaux de contrôle	adresse suivante	*LD.?*	*Gate ?*	*?MX*	divers
$MAR \leftarrow PC, PC \leftarrow PC + 1$	00000000	01xxx000	00000000	x1000001	x1x000xx	00000000	xxxx0000
$MDR \leftarrow MAR$	00000001	01xxx000	00000000	x0100000	x0x000xx	00000000	xxxx0010
$IR \leftarrow MDR$	00000010	01xxx000	00000000	x0010000	x0x100xx	00000000	xxxx0000
opcode ????, $[\overline{BEN}]$	00000011	11xxx111	00000111	x0001000	x0x000xx	00000000	xxxx0000
opcode 0???	00000100	11xxx110	00000111	x0000000	x0x000xx	00000000	xxxx0000
opcode 00??	00000101	11xxx101	00000111	x0000000	x0x000xx	00000000	xxxx0000
opcode 000?	00000110	11xxx100	00001000	x0000000	x0x000xx	00000000	xxxx0000
Stop	00000111	00xxx000	00000000	x0000000	x0x000xx	00000000	xxxx0000
opcode 0001, $DR \leftarrow SR1 + SR2 / SEXT, CC$	00001000	10xxx000	00000000	x0000110	x0x010xx	00000100	xxxx0000

FIG. 5 – Microprogramme de l'instruction ADD

Les 4 microinstructions de branchement conditionnel testent successivement les 4 bits de poids fort du registre IR, en posant $Condition_{/3} = 111$, puis $Condition_{/3} = 110$, puis $Condition_{/3} = 101$ et enfin $Condition_{/3} = 100$. L'addition proprement dite a lieu à la microinstruction d'adresse 00001000. L'adresse 00000111 implémente la microinstruction *Stop* ; elle est atteinte lorsque l'instruction à exécuter n'est pas une addition.

On supposera que la pile du LC-2 est implémentée de la façon simplifiée suivante. Il y a seulement un pointeur de pile `R6` (pas de pointeur de contexte, pas de notion de contexte en fait) qui pointe sur le premier emplacement libre de la pile. Allouer un élément consiste à incrémenter `R6`. La même pile est utilisée par le système comme par les processus utilisateur.