

Examen d'Architecture des Ordinateurs

Majeure 1 – École Polytechnique

2005–2006

- L'examen dure 3 heures.
- Le sujet comporte 9 pages dont 3 pages de 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.

Rappels sur le LC-2

Dans cette section, on effectue plusieurs rappels sur le LC-2. Ces rappels sont essentiellement destinés au 2ème exercice ; pour le 1er exercice, le seul rappel nécessaire est le jeu d'instructions.

On utilise le schéma du processeur LC-2 (hors gestion des interruptions) de la figure 1.

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

Le rôle des signaux de contrôle de la figure 1 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 2 (les x correspondent à des signaux inutilisés, disponibles pour étendre le microcontrôleur) :

On rappelle également la liste des instructions du LC-2 (dans sa version simplifiée) en figure 3.

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

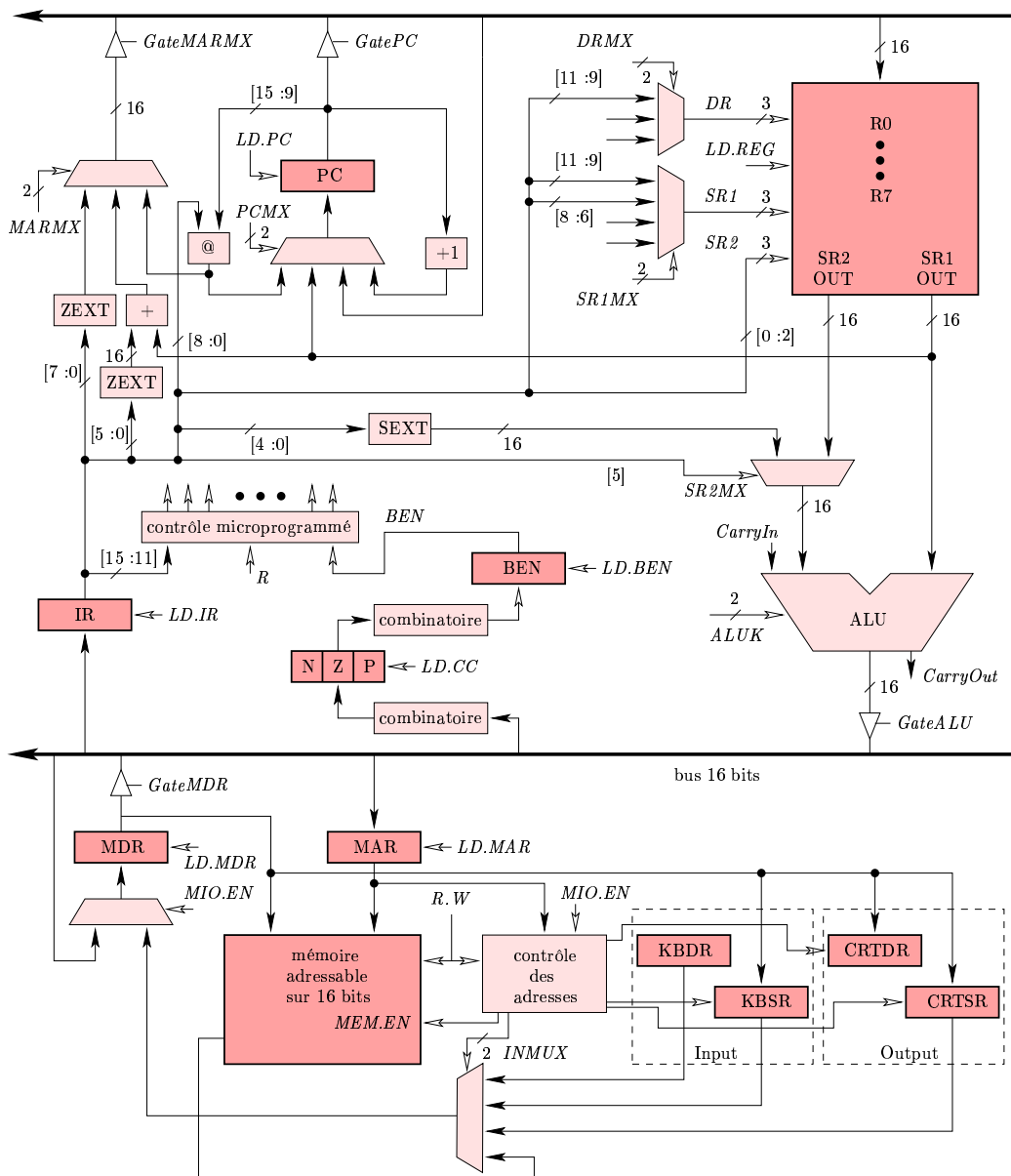


FIG. 1 – Diagramme des blocs du LC-2

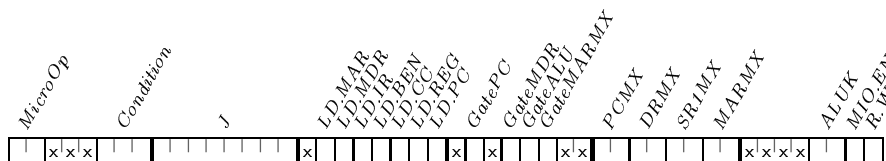


FIG. 2 – Format des mirco-instructions

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 4 (toutes les valeurs sont en binaire, sauf l'état) :

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.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	3	
ADD DR, SR1, SR2	0	0	0	1	DR			SR1			0	0	0	SR2				
ADD DR, SR1, imm5	0	0	0	1	DR			SR1			1	imm5 : immédiat 5-bits signé						
AND DR, SR1, SR2	0	1	0	1	DR			SR1			0	0	0	SR2				
AND DR, SR1, imm5	0	1	0	1	DR			SR1			1	imm5 : immédiat 5-bits signé						
NOT DR, SR	1	0	0	1	DR			SR			1	1	1	1	1	1		
BRnzp label	0	0	0	0	n	z	p	offset 9-bits non signé dans la page courante										
JMP label	0	1	0	0	0	0	0	offset 9-bits non signé dans la page courante										
JSR label	0	1	0	0	1	0	0	offset 9-bits non signé dans la page courante										
JMPR offset	1	1	0	0	0	0	0	BaseR			index 6-bits non signé							
JSRR offset	1	1	0	0	1	0	0	BaseR			index 6-bits non signé							
LEA DR, label	1	1	1	0	DR			offset 9-bits non signé dans la page courante										
LD DR, label	0	0	1	0	DR			offset 9-bits non signé dans la page courante										
LDR DR, BaseR, offset	0	1	1	0	DR			BaseR			index 6-bits non signé							
ST SR, label	0	0	1	1	SR			offset 9-bits non signé dans la page courante										
STR SR, BaseR, offset	0	1	1	1	SR			BaseR			index 6-bits non signé							
RET	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0		

FIG. 3 – Format des instructions du LC-2

	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 0???, [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. 4 – Microprogramme de l'instruction ADD

Exercice 1 - Division dans le LC-2 (6 points)

On veut écrire un programme de division entière, en assembleur LC-2, en utilisant une méthode proche de celle utilisée pour effectuer la division à la main.

Les deux opérandes sont appelés le *divisé* et le *diviseur* ; le résultat est composé du *quotient* et du *reste*. Les étapes sont les suivantes :

1. On commence par le bits de poids fort du divisé, et on considère un nombre croissant de bits de poids faible, jusqu'à ce que le nombre formé de l'ensemble de ces bits soit supérieur au quotient.
2. On effectue la division de ce nombre par le quotient ; on met à jour le quotient et on calcule le reste intermédiaire.
3. On recommence jusqu'à ce que tous les bits du nombre divisé aient été considérés.

Initialement, la mémoire contient le divisé et le diviseur aux labels respectifs *divisé* et *diviseur*. Vous êtes libres de configurer d'autres emplacements mémoire si vous le jugez nécessaire. Après initialisation des registres, on adoptera les conventions suivantes :

- R0 contiendra le divisé ;
- R1 contiendra le diviseur ;

- R2 contiendra le reste, ou le reste concaténé avec des bits du nombre à diviser ;
- R3 contiendra le quotient ;

On supposera que les deux opérandes sont strictement positifs et strictement inférieurs à 2^{15} .

On prendra garde à commenter presque chaque ligne de code assembleur de façon claire et utile (par exemple, éviter des commentaires du type $R0 \leftarrow R0 + 1$ pour l'instruction `ADD R0, R0, #1`, ils sont sans intérêt). Pour les éventuelles fonctions, ne pas utiliser la pile comme vue en cours. Les opérandes seront toujours directement passés par registres, ainsi que le résultat. Si des registres doivent être sauvegardées, on utilisera des emplacements statiques en mémoire intitulés `bak_Rn` pour le registre `Rn`.

Un programme peu commenté ou ne respectant pas **strictement** les conventions ci-dessus sera considéré comme nul.

Question 1.1

En utilisant les conventions ci-dessus, écrire une fonction qui prend en entrée deux nombres stockés dans les registres `R0` et `R2`, et qui décale vers la gauche les bits de `R2`, transfère le bit de poids fort de `R0` dans le bit de poids faible de `R2` et décale `R0`.

Question 1.2

Ecrire le programme complet de la division, en utilisant la fonction précédente. Les facteurs d'évaluation seront, notamment, la correction, la clarté, la concision et la rapidité (nombre d'instructions à exécuter) du programme. Il est recommandé de déboguer votre programme avec quelques exemples simples (e.g., 54 divisé par 5).

Réponse

```
.ORIG    x3000

init
    LD     R0, divide ; chargement nb à diviser
    LD     R1, diviseur ; chargement diviseur
    AND    R2, R2, #0 ; reste/résultat intermédiaire (res) = 0
    AND    R3, R3, #0 ; quotient = 0
    LD     R4, val16 ; compteur = 16

division
    ADD    R4, R4, #-1 ; décrémenter compteur
    BRn    fin ; tous les bits du diviseur utilisés ?
    ADD    R3, R3, R3 ; décaler à gauche quotient pour nouveau bit (par défaut = 0)
    JSR    decal ; transfère bit poids fort diviseur dans bit poids faible res
    NOT    R6, R1
    ADD    R6, R6, #1 ; R6 = -diviseur (complément à 2)
    ADD    R6, R2, R6 ; R6 = res - diviseur
    BRn    division ; res < diviseur ?
    ADD    R2, R6, #0 ; res = res - diviseur
    ADD    R3, R3, #1 ; nouveau bit quotient = 1
    JMP    division

decal
    ; transfère bit poids fort divisé dans bit poids
faible diviseur
    ADD    R2, R2, R2 ; décalage à gauche res
    ADD    R0, R0, #0 ; NZP = divisé
    BRzp   decalfin ; bit poids fort divisé = 0
    ADD    R2, R2, #1 ; bit poids fort divisé = 1 et transféré dans bit poids faible res
decalfin
    ADD    R0, R0, R0 ; décalage à gauche divisé
    RET

fin
    HALT
    NOP    ; NOP inutile, ajouté à cause bug LC-2 (pas de label avant .end)

divide
    .fill  x7337
diviseur
    .fill  x0000
val16
    .fill  x0010
```

Question 1.3

Expliquer, sans le programmer, comment il faudrait modifier votre programme si l'on supposait que les opérandes étaient deux nombres non signés sur 16 bits.

Réponse

Il est plus difficile de vérifier si le résultat intermédiaire est supérieur ou égal au quotient parce que le LC-2 ne fonctionne qu'avec des nombres signés. Comme pour le TD sur la multiplication 32 bits, il faut envisager les 4 cas possibles.

Exercice 2 - Du LC-2 au LC-2D (14 points)

Le but de cet exercice est d'accroître les performances des systèmes construits autour de l'architecture LC-2. On propose ainsi une extension parallèle du processeur LC-2, le *multi-processeur* LC-2D, comportant deux *cœurs* de processeur LC-2 sur une même puce (*chip multi-processor*). Ces deux cœurs peuvent exécuter des programmes totalement indépendants, et les seules communications possibles se font via la mémoire. Dans tout l'exercice, on supposera que les deux cœurs sont synchronisés sur une *horloge commune*, et pour simplifier, on ignorera le traitement des interruptions.

On commencera par étudier le cas où les deux cœurs sont strictement identiques au LC-2, mis à part l'interface avec la mémoire. Pour plus de commodité, tous les registres et signaux sont suffixés par le numéro x du cœur auquel ils appartiennent : par exemple le registre d'instruction s'appelle IR. 1 dans le premier cœur et IR. 2 dans le deuxième. Bien entendu, le cœur 1 n'a pas accès aux registres et signaux du cœur 2, et réciproquement.

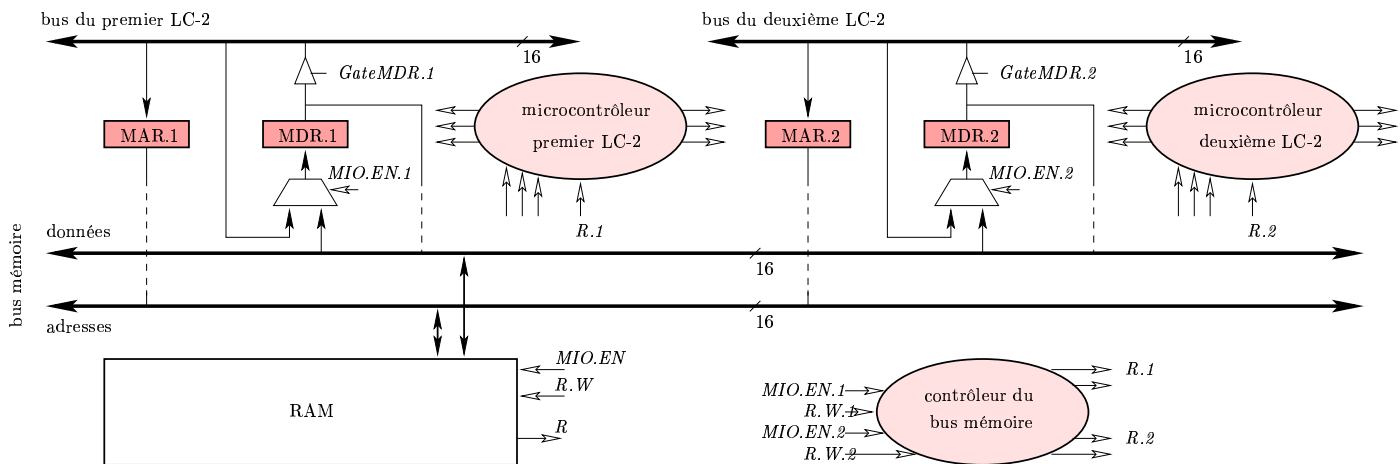


FIG. 5 – Schéma du LC-2D

Comme indiqué sur la figure 5 (incomplète), chaque cœur dispose de ses propres registres MAR. x et MDR. x . Chaque cœur dispose également de son propre signal $R.x$, émis par le contrôleur du bus, indiquant que l'accès mémoire qu'il avait initié précédemment est terminé.

En revanche, la mémoire et les registres d'entrée-sortie sont communs et repoussés au delà d'un *bus* reliant les deux cœurs. Ce bus sert bien sûr de chemin de données et d'adresses entre les cœurs et la mémoire, mais il comporte également un petit *contrôleur de bus* chargé d'arbitrer entre les accès mémoire déclenchés indépendamment par chaque cœur. La mémoire n'est plus contrôlée directement par les LC-2 mais par ce contrôleur de bus ; lorsqu'un cœur effectue un accès mémoire, ce contrôleur garantit que toute requête d'accès mémoire effectuée par l'autre cœur reste bloquée en attente tant que l'accès en cours n'est pas terminé. Ainsi, les signaux $MIO.EN.x$ et $R.W.x$ émis par chaque cœur de LC-2 sont interprétés par le contrôleur de bus, lequel est en charge des signaux $MIO.EN$ et $R.W$ agissant directement sur la mémoire. Inversement, lorsque l'accès (en lecture ou en écriture) est terminé, la mémoire en informe le contrôleur du bus via le signal R ; le contrôleur du bus répercute l'information via le signal $R.x$ du cœur x qui avait émis la requête d'accès mémoire.

On suppose que la latence de la mémoire est constante pour tout accès (en lecture ou écriture, à n'importe quelle adresse) et égale à 10 cycles ; c'est-à-dire que le signal R est renvoyé 10 cycles après que l'émission du signal $MIO.EN$ qui matérialise le début de l'accès mémoire.

Question 2.1

On ajoute le signal $R.x$ (où x vaut 1 ou 2) aux signaux de condition que peuvent interroger les microinstructions de branchement conditionnel, en l'associant à la valeur $Condition_3 = 001$ (les signaux de condition associés aux valeurs 001 et 010 ne sont pas utilisés sur le LC-2). Le signal $R.x$ indique au cœur x que la donnée lue ou écrite a été transférée via le bus.

Proposez un microprogramme pour l'instruction LDR, en vous assurant que la donnée lue en mémoire n'est pas transférée dans le registre MDR. x tant que $R.x$ ne vaut pas 1. Vous pourrez vous inspirer de la correction du TD 4 pour l'instruction STR, et vous ignorez les premières étapes du microprogramme nécessaires chargement et au décodage de l'instruction (la lecture de l'instruction en mémoire, son transfert vers IR, et les branchements conditionnels sur les 4 bits d'opcode). Il suffira d'indiquer le type et l'adresse

relative de chaque microinstruction ainsi que les signaux dont l'activation/désactivation est spécifique à cette microinstruction.

Par la suite, on supposera que toutes les instructions effectuant des accès mémoire — et tous les chargements d'instructions en mémoire — ont été mis à jour de cette manière.

Réponse

Il suffit de trois états, implémentés chacun par une microinstruction.

- adresse a , microinstruction simple pour l'état 16 : l'offset sur 6 bits étendu avec des zéros est ajouté au registre de base et le résultat est rangé dans MAR ($SR1MX.x=01$, $LD.MAR.x=1$, $GateMARMX.x=1$ et $MARMX.x=01$);
- adresse $a+1$, microinstruction de branchement conditionnel à l'adresse $a+1$ sur $R.x$ pour l'état 29 : la valeur lue est rangée dans MDR ($LD.MDR.x=1$, $MIO.EN.x=1$, $R.W.x=0$);
- adresse $a+2$, microinstruction de branchement inconditonnnel à l'adresse 1 pour l'état 30 : la valeur de MDR est transférée dans le banc de registres et dans le registre CC ($LD.REG.x=1$, $LD.CC.x=1$, $GateMDR.x=1$ et $DRMX.x=00$).

Question 2.2

Dessinez le chemin de données reliant les deux cœurs au bus et le bus à la mémoire, et donnez la liste des signaux additionnels nécessaires au contrôle de ce bus.

Réponse

Quatre signaux de contrôle supplémentaires : $GateA.x$, $GateD.x$, $MAR.x$ vers le bus d'adresses via gate commandée par $GateA.x$, $MDR.x$ vers le bus de données via gate commandée par $GateD.x$.

Question 2.3

Proposez un automate de contrôle (déterministe) pour le contrôleur de bus. Cet automate doit garantir l'équité des accès entre les deux cœurs, en évitant de donner toujours la priorité aux accès issu d'un même cœur.

Réponse

Première solution, non déterministe en cas d'accès concurrents (état initial Q_0) :

- $Q_0 - \overline{MIO.EN.x} \rightarrow Q_0 : MIO.EN=0$;
- $Q_0 - MIO.EN.x \rightarrow Q_x : MIO.EN=1, R.W=R.W.x$;
- $Q_x - \overline{R} \rightarrow Q_x : MIO.EN=1, R.W=R.W.x$;
- $Q_x - R \rightarrow Q_0 : R.x=1$.

Pour garantir l'équité, il suffit de dédoubler l'état Q_0 en Q'_1 et Q'_2 , en spécialisant chaque nouvel état pour n'accepter que les accès du cœur x , en les munissant de transitions vers l'état jumeau si un tel accès n'est pas demandé, et en remplaçant Q_0 par Q_{3-x} dans la 4ème transition ci-dessus (état initial Q'_1) :

- $Q'_x - \overline{MIO.EN.x} \rightarrow Q'_{3-x} : MIO.EN=0$;
- $Q'_x - MIO.EN.x \rightarrow Q_x : MIO.EN=1, R.W=R.W.x$;
- $Q_x - \overline{R} \rightarrow Q_x : MIO.EN=1, R.W=R.W.x$;
- $Q_x - R \rightarrow Q'_{3-x} : R.x=1$.

Question 2.4

Réalisez le contrôle câblé associé à cet automate. Vous vous efforcerez de minimiser la taille du circuit (en nombre de bascules D et logique combinatoire).

Réponse

C'est une question de cours.

Au minimum, deux registres d'état : le premier indique qu'un accès est en cours, le deuxième quel cœur l'a effectué.

Chaque registre est associé à une fonction booléenne : la disjonction des conditions des transitions dont il est la destination. Le calcul et la minimisation des fonctions booléennes ne posent pas de problème particulier.

Question 2.5

On considère le programme de la figure 6, et on suppose que ses deux fonctions s'exécutent en parallèle sur le LC-2D : *Fonction1* est exécutée sur le cœur 1 et *Fonction2* sur le cœur 2.

Montrez qu'un comportement non-déterministe peut se produire lorsque les deux cœurs accèdent à la même adresse mémoire, et que l'un au moins des accès est une écriture (noter que l'ordre des additions de 1 ou de 2 aux éléments de A n'a pas d'importance du fait de l'associativité et la commutativité de l'addition). On pourra par exemple écrire les séquences d'instructions correspondant à deux scénarios possibles et fournissant des résultats différents.

Réponse

Deux séquences au résultat incompatible, en supposant que R1 contient la même adresse dans les deux cœurs : LDR.1, LDR.2, ADD.1, ADD.2, STR.1, STR.2 ; LDR.1, ADD.1, STR.1, LDR.2, ADD.2, STR.2.

```

        .ORIG    x3000
Fonction1
        LEA     R1,A
        LD      R2,C
Boucle1 LDR      R0,R1,#0
        ADD     R0,R0,#1
        STR     R0,R1,#0
        ADD     R1,R1,#1
        ADD     R2,R2,#-1
        BRnp   Boucle1
        RET
Fonction2
        LEA     R1,A
        LD      R2,C
Boucle2 LDR      R0,R1,#0
        ADD     R0,R0,#2
        STR     R0,R1,#0
        ADD     R1,R1,#1
        ADD     R2,R2,#-1
        BRnp   Boucle2
        RET
C       .FILL   1000
A       .BLKW   1000
        .END

```

FIG. 6 – Exemple

Question 2.6

Il y a de nombreuses sources de non-déterminisme en programmation parallèle, et pour simplifier, on ne considèrera que celles du type étudié à la question précédente.

La solution naturelle consiste tout d'abord à identifier au préalable toutes les zones de mémoire « partagées », c'est-à-dire les zones susceptibles d'être accédées à la fois par des codes exécutés sur des cœurs différents ; ensuite, on exige que chaque accès à une zone partagée soit précédé d'une « demande d'exclusivité » et suivi de la « libération de cette exclusivité ». On dit alors que le programme accédant à la zone partagée « entre en section critique ».

Proposez un protocole de demande/libération d'exclusivité utilisant une unique variable S permettant d'indiquer que l'un des cœurs est dans une section critique. Pourquoi ne peut-on pas implémenter ce protocole avec la version actuelle du LC2D ?

Réponse

Pour implémenter ce mécanisme d'exclusivité, on utilise une variable partagée supplémentaire S , en faisant en sorte que S vaille 0 si et seulement si aucun programme est dans une section critique (c'est-à-dire, aucun programme ne cherche à accéder à une zone partagée). La demande d'exclusivité consiste à attendre que S soit nul, puis dès que S est nul, à mettre S à 1. La libération de l'exclusivité consiste à mettre S à 0.

Question 2.7

Il est en réalité impossible de supprimer le non-déterminisme sans un support matériel supplémentaire.

On introduit donc une nouvelle instruction, appelée *test-and-set* et notée $\text{TSR } R_x, R_y$ qui effectue *atomiquement*, c'est-à-dire sans libérer l'accès au bus, une lecture mémoire à l'adresse indiquée par le registre R_y , immédiatement suivie, si la valeur lue est nulle, de l'écriture de la valeur de R_x à cette adresse (l'instruction équivaut à un NOP si la valeur lue est non nulle) ; TSR modifie les bits NZP selon les résultats du test ($Z = 1$ si la valeur est nulle). On ne décrira le format de cette instruction qu'à une question ultérieure.

Modifiez *Fonction1* et *Fonction2* pour implémenter le protocole décrit ci-dessus avec l'instruction TSR (il suffit de montrer les nouvelles parties de code) ; notez que l'ordre des opérations sera toujours indéterminé, mais cela n'a pas d'importance du fait de l'associativité et la commutativité de l'addition.

Réponse

On remplace le code d'entrée en section critique par

```

AND     R3, R3, #0 ; peut être sorti de la boucle
ADD     R3, R3, #1 ; peut être sorti de la boucle
LEA     R4, S      ; peut être sorti de la boucle
TSR     R4, R3

```

La sortie de section critique est inchangée.

Question 2.8

Comment faut-il modifier le bus mémoire et/ou son contrôleur pour implémenter cette instruction ? Expliquer en détail ces modifications ; on considérera ensuite qu'elles ont été effectuées.

Réponse

Le contrôleur de bus doit refuser tout accès issu de l'autre cœur entre l'étape de lecture et d'écriture de l'instruction TSR. Pour cela, on peut ajouter deux signaux $ATOM.x$ indiquant que le bus doit être réservé pour plusieurs accès consécutifs. Les états Q'_x sont dupliqués en Q''_x avec de nouvelles transitions (état initial Q'_1) :

- $Q'_x - \overline{MIO.EN.x} \rightarrow Q'_{3-x} : MIO.EN=0;$
- $Q'_x - MIO.EN.x \rightarrow Q_x : MIO.EN=1, R.W=R.W.x;$
- $Q_x - \overline{R} \rightarrow Q_x : MIO.EN=1, R.W=R.W.x;$
- $Q_x - R \overline{ATOM} \rightarrow Q'_{3-x} : R.x = 1.$
- $Q_x - R ATOM \rightarrow Q''_{3-x} : R.x = 1.$
- $Q''_x - MIO.EN.x \rightarrow Q_x : MIO.EN=1, R.W=R.W.x;$

Question 2.9

On utilise l'op-code 1111 (inutilisé dans cet exercice) pour l'instruction TSR.

Proposez un format pour l'instruction TSR et réalisez son microprogramme. Vous vous efforcerez de minimiser l'impact de l'implémentation de cette instruction sur le chemin de données des cœurs (étudier en particulier le rôle des signaux BEN et $SR1MX$ dans l'implémentation de TSR, et vous ne donnerez pas le code binaire complet des microinstructions)

Indiquez les modifications éventuelles à apporter au microcontrôleur et/ou au chemin de données du processeur.

Réponse

On utilise les deux opérandes $SR1$ et $SR2$ habituelles pour simplifier les modifications ultérieures, et on met le bit z à 1 pour pouvoir utiliser le signal BEN sans modifier le chemin de données.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TSR SR1, SR2	1	1	1	1	0	1	0	SR1			0	0	0	SR2		

On ajoute une entrée au multiplexeur $SR1MX$, associée à la valeur 100, sélectionnant les bits 0 à 2 de IR . Enfin, le signal de contrôle $ATOM.x$ doit être ajouté.

- adresse a , microinstruction simple : l'offset sur 6 bits étendu avec des zéros est ajouté au registre de base et le résultat est rangé dans MAR ($SR1MX.x=001$, $LD.MAR.x = 1$, $GateMARMX.x = 1$ et $MARMX.x = 01$);
- adresse $a + 1$, microinstruction de branchement conditionnel à l'adresse $a + 1$ sur $R.x$: la valeur lue est rangée dans MDR ($LD.MDR.x = 1$, $MIO.EN.x = 1$, $R.W = 0$, $ATOM.x = 1$);
- adresse $a + 2$, microinstruction simple; la valeur de MDR est transférée dans le registre CC ($LD.CC.x = 1$, $GateMDR.x = 1$ et $DRMX.x = 00$);
- adresse $a + 3$, microinstruction simple : le registre source est copié dans MDR et le registre BEN est chargé ($GateMDR.x = 1$, $LD.REG = 1$, $SR1MX.x = 100$, $ALUK = 00$);
- adresse $a + 4$, microinstruction de branchement conditionnel à l'adresse 1 sur \overline{BEN} ;
- adresse $a + 5$, microinstruction de branchement conditionnel à l'adresse $a + 5$ sur $R.x$: la valeur de MDR est écrite en mémoire ($MIO.EN.x = 1$, $R.W = 1$);
- adresse $a + 6$, microinstruction de branchement inconditonnal à l'adresse 1 : $ATOM.x = 0$.

Question 2.10

Comptez le nombre d'accès mémoire effectués par chaque fonction (on négligera les instructions précédant chaque boucle). Au maximum, quel gain peut-on espérer avec l'exécution concurrente par rapport à l'exécution séquentielle, en supposant que, hors accès mémoire (LD/ST et chargement de l'instruction), les instructions requièrent 5 cycles chacune ?

Réponse

Chaque boucle exécute 13000 accès (lecture d'instructions et load/store, en incluant les instructions de gestion de l'exclusivité).

L'exécution parallèle ou séquentielle prendra donc 260000 cycles d'accès mémoire. Le gain de l'exécution parallèle sera limité au recouvrement des cycles opératoire d'un cœur par les cycles d'accès mémoire de l'autre, soit environ 5 cycles par instruction. L'exécution séquentielle prendra donc environ $260000 + 130000 = 390000$ cycles contre $260000 + 65000 = 325000$ pour la version parallèle : un maigre bénéfice.

Question 2.11

Les performances sont pénalisées fortement par la latence d'accès mémoire et par la séquentialité de ces accès. Décrivez comment l'introduction de caches permet, en général, de remédier à chacun de ces

problèmes. On considèrera d'abord un unique cache avec une latence de 2 cycles (stockant les instructions et les données ensemble) partagé entre les deux cœurs, puis des caches distincts pour chaque cœur, toujours avec une latence de deux cycles (stockant toujours instructions et données ensemble).

Il est inutile de décrire précisément le fonctionnement de ces caches, mais vous indiquerez encore, dans le cas de lectures/écritures concurrentes en mémoire, les gains maximaux attendus par rapport à une exécution séquentielle.

Réponse

Dans le premier cas, la latence réduite du cache permet de minimiser l'impact de la séquentialité des accès mémoire. Avec une latence de 2 cycles, les temps d'exécution ci-dessus deviennent respectivement $52000 + 130000 = 182000$ et $52000 + 65000 = 117000$ cycles.

Avec des caches distribués, seuls les accès concurrents à la même ligne de cache sont séquentiels. On supposera que cela n'arrive jamais, en remarquant que si cela arrive, il y a de fortes chances que l'un des cœurs prenne une avance définitive sur l'autre et que cela n'arrivera donc plus ultérieurement (pour ces fonctions). Les temps d'exécution approximatifs deviennent $26000 + 130000 = 156000$ cycles et $26000 + 65000 = 91000$ cycles. L'accélération est excellente dans ce dernier cas.

Question 2.12

Montrez que l'introduction de caches distincts pour chaque cœur peut conduire l'exécution à un résultat faux, si l'on ne dispose pas d'un mécanisme pour garantir la cohérence des valeurs stockées dans les deux caches.

En ne modifiant que le comportement des écritures, proposez un mécanisme simple garantissant la cohérence des caches, puis évaluez ses performances.

Réponse

C'est le problème de la cohérence des caches : par exemple, l'accès par le cœur 1 à une ligne de son cache privé qui n'a pas été synchronisée avec une écriture à la même adresse par le cœur 2, ou encore, l'écrasement de la valeur correcte en mémoire par une ligne non synchronisée.

Le mode write-through (écriture à travers) force chaque écriture à modifier directement la mémoire. Cela ralentit fortement les écritures mais corrige simplement le problème de cohérence. Attention, les lectures de l'autre cœur doivent être bloquées lors d'une écriture, afin d'attendre l'invalidation éventuelle d'une ligne de son cache depuis la mémoire, à moins d'implémenter un mécanisme plus complexe de signalisation. Comme chaque itération de boucle n'effectue que 2 écritures parmi les 13 accès mémoire, et que celles-ci sont espacées de plus de 10 cycles, les temps d'exécution deviennent respectivement $22000 + 20000 + 130000 = 176000$ et $22000 + 20000 + 65000 = 111000$ cycles.

Note : il existe une solution moins radicale et plus performante au problème de cohérence de caches sur bus partagés : les snoopy caches.

Question 2.13 (facultative)

Adaptez l'implémentation de l'instruction TSR (et le bus mémoire) pour qu'elle fonctionne en présence de caches distribués.

Réponse

Il faut bloquer les accès en lecture de l'autre cœur dès le début de l'accès en lecture de l'instruction TSR, comme pour un store pass-through. Les détails sont sensiblement plus complexes que lors de l'implémentation de TSR sans caches.