Numéro :

Examen ASR1 2016/2017

10/01/2017

Durée 3h.
Répondez sur le sujet.
Tous documents autorisés, mais en papier seulement.
Crayon à papier accepté, de préférences aux ratures et surcharges.
Les questions de cours sont de difficulté variable et dans le désordre.
Les cadres donnent une idée de la taille des réponses attendues.

1 Questions de cours (7 points)

Dans les questions QCM, entourez soit V pour vrai, soit F pour faux. Si la proposition ne veut rien dire, entourez F. Il y a une pénalité pour les mauvaises réponses.

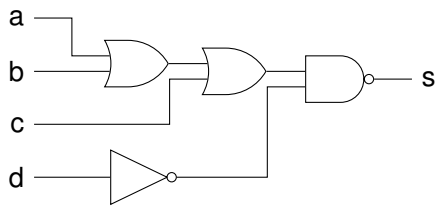
Q1.1

Dessinez ici une machine de von Neumann :

Q1.2 Le circuit suivant implémente l'expression :

V F [a] $s = \overline{((a \wedge b) \wedge c) \vee \overline{d}}$ V F [b] $s = (\overline{a} \wedge \overline{b} \wedge \overline{c}) \vee \overline{d}$

V F [c] $s = \overline{a \wedge b \wedge c} \vee \overline{d}$ V F [d] $s = \overline{(a \vee b \vee c) \wedge \overline{d}}$



Q1.3 Donnez la valeur du nombre binaire 11111011
- — interprété comme un entier positif : [ ]
- — interprété en complément à 2 sur 8 bits : [ ]

Donnez la valeur binaire du nombre hexadécimal **F2D** :

Q1.4 Donnez une expression booléenne implémentant la table de vérité suivante.

s=

a	b	c	d	s
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Q1.5 Précisez les termes suivants (comme pour l'exemple)

ENSL : École Normale Supérieure de Lyon

FIFO :

RAM :

ALU :

VLIW :

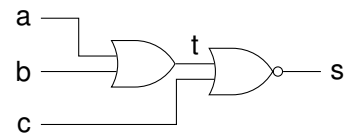
Q1.6 La plus petite valeur non nulle qu'on peut mettre dans une variable de type `float` est 2^{-150}.
Cela fait en gros combien en décimal ? Justifiez.

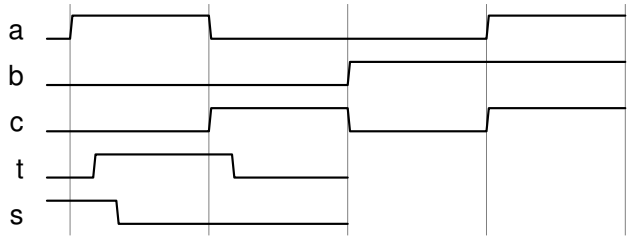
	1	0	1	1
x	0	1	0	1

Posez la multiplication ci-contre en binaire.

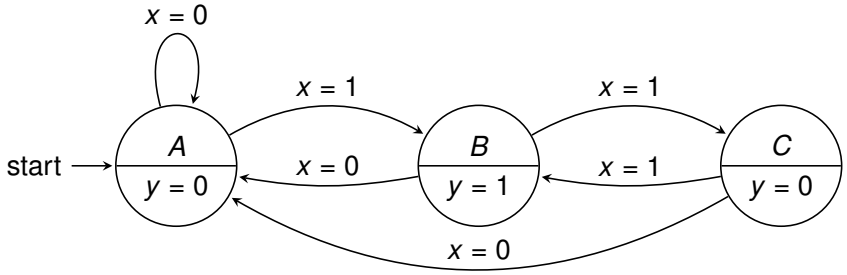
Q1.7 Les nombres sont codés en binaire non signé.
Bien faire figurer les éventuelles retenues.

Q1.8 On considère le circuit ci-contre, dans lequel chaque porte calcule sa sortie avec un petit délai. Le chronogramme suivant illustre le fonctionnement de ce circuit. Complétez les lignes pour t et s.

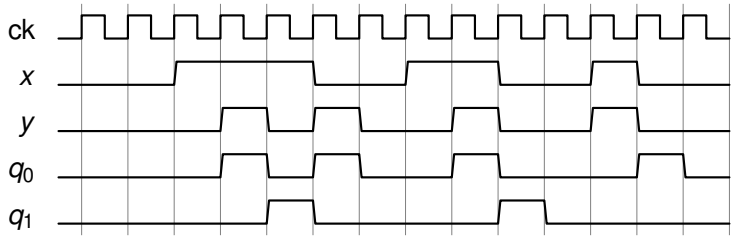




Q1.9 On considère l'automate ci-dessous. Son entrée est un booléen x, et sa sortie est un booléen y. Les transitions ont lieu sur front montant de l'horloge. Son état est codé par les bits q_0 et q_1.



Le chronogramme ci-dessous est supposément obtenu en simulant cet automate depuis l'état start, mais il y a une erreur sur la ligne Y dans ce chronogramme : entourez-la. Vous remplirez au passage la table qui donne l'encodage des états.



état	q_1	q_0
A		
B		
C		

Q1.10

V F a À cause du cycle de von Neumann, il y a toujours de la localité spatiale sur le code

V F b La localité temporelle sur le code vient essentiellement des boucles

V F c Pour un même problème, on peut écrire du code avec plus ou moins de localité sur les données

V F d Si on sépare les caches programme et donnée, il faut le faire pour tous les niveaux de la hiérarchie

Q1.11

V F a Un processeur superscalaire est forcément RISC.

V F b Un processeur superscalaire peut exécuter des dizaines d'instructions chaque cycle.

V F c Un processeur superscalaire doit être capable de lancer plusieurs instructions chaque cycle.

V F d Un processeur superscalaire n'a pas besoin d'analyser les dépendances entre instructions.

Q1.12
Complétez les phrases suivantes avec IF, ID, OL, EX, WB :

— Une dépendance RAW peut provoquer une bulle à l'étage ____ .

— L'étage ____ sert essentiellement à analyser les dépendances et renommer les registres.

— Une dépendance de contrôle peut provoquer une bulle à l'étage ____ .

— Un court-circuit consiste à envoyer un résultat produit par l'étage ____ à l'étage OL sans attendre la fin de l'étage ____ .

Q1.13 Pour le code patassembleur suivant, donnez

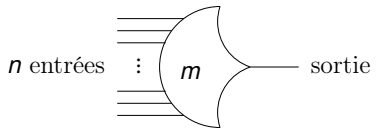
1. les dépendances RAW *(read after write)* (je veux des flèches de registre à registre sur le code).
2. un arbre d'expression
3. une expression algébrique

```
; entrées: R10, R11, R12
MUL R10, R10 -> R9
MUL R11, R11 -> R10
ADD R9,  R10 -> R9
MUL R12, R12 -> R10
ADD R9,  R10 -> R9
; sortie: R9
```

Q1.14 Le code précédent tourne sur un processeur avec les caractéristiques suivantes. L'addition prend un cycle. La multiplication prend 5 cycles mais est pipelinée : on peut en lancer une par cycle.
Proposez un réordonnancement de ce code, possiblement avec renommage des registres, qui sera plus rapide (on ne demande pas de compter les cycles précisément).

2 La logique nulle (5 points)

Ce problème traite de l'utilisation de la *Null Convention Logic* (ou NCL) pour réaliser des circuits asynchrones, c'est-à-dire sans horloge.
La NCL est basée sur un seul type de porte, les seuils *(threshold)*, paramétrés par le nombre d'entrées $n \geq 2$ et un entier m, et représentés de manière générique par le dessin suivant :



(elle est assez pénible à dessiner donc je vous autorise à l'approximer par un triangle avec un entier dedans).
Le fonctionnement de cette porte, notée T_{nm}, est le suivant :

- La sortie comme les n entrées peuvent valoir NULL ou DATA (on ne les appelle pas 0 et 1 pour éviter la confusion avec les 0 et 1 logiques qui viennent plus bas, mais ce sont bien deux valeurs booléennes).
- La porte a une mémoire (ou hysteresis).
- La sortie descend à NULL si et seulement si toutes les entrées sont à NULL.
- La sortie monte à DATA dès que m des n entrées montent à DATA : m est la valeur de seuil.
- Désormais, on dira juste "monter" et "descendre".
- Une fois montée, la sortie reste à DATA jusqu'à ce que toutes les entrées soient descendues. Je sais, je l'ai déjà dit.

Pour le reste, ces portes s'assemblent au moyen de fils selon les mêmes règles que les portes classiques.

C'est un peu pauvre pour faire de la logique. Dans tout le problème, on utilisera donc pour transporter une information binaire s un codage *dual rail* c'est-à-dire deux fils s^0 et s^1 utilisés de la manière suivante :

- (s^0, s^1) = (NULL, NULL) : état de repos
- (s^0, s^1) = (DATA, NULL) : transmission d'un 0 logique
- (s^0, s^1) = (NULL, DATA) : transmission d'un 1 logique
- (s^0, s^1) = (DATA, DATA) : invalide. Chaque fois que l'un de vos circuits passe par cet état vous perdez 3 points.

2.1 Questions cadeau

Q2.1 Combien de portes seuil à 2 entrées ? Combien de portes seuil à 3 entrées ? Combien de portes seuil à n entrées ?

Q2.2 Proposez des implémentations de T_{31} et T_{32} à base de T_{21} et T_{22}.

2.2 Logique dual-rail

Q2.3 Comment implémenter le OU logique à deux entrées dual-rail ? On cherchera des circuits qui calculent séparément les deux fils s^0 et s^1 de la sortie double-rail à partir des entrées double-rail (a^0, a^1) et (b^0, b^1).

Q2.4 Comment implémenter le ET logique à deux entrées dual-rail ?

Q2.5 Comment implémenter un XOR dual-rail ?

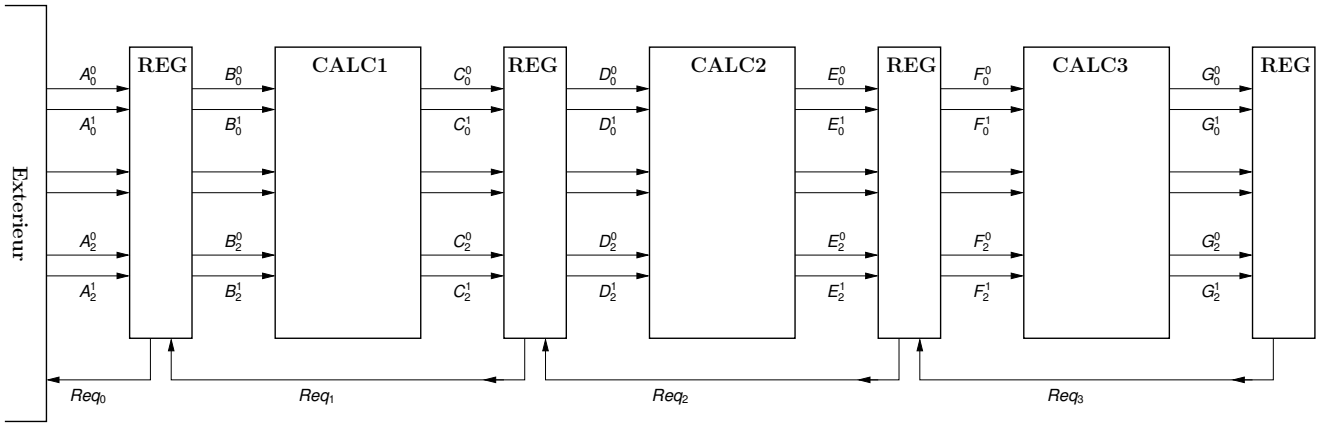
Q2.6 Soit une fonction booléenne à *n* entrées donnée par sa table de vérité. Décrivez une technique générale pour l'implémenter par des portes à seuil en logique dual-rail.

2.3 Circuits auto-synchronisés

Q2.7 Soit une sortie dual-rail d'un circuit. Comment détecter si sa donnée est prête ?

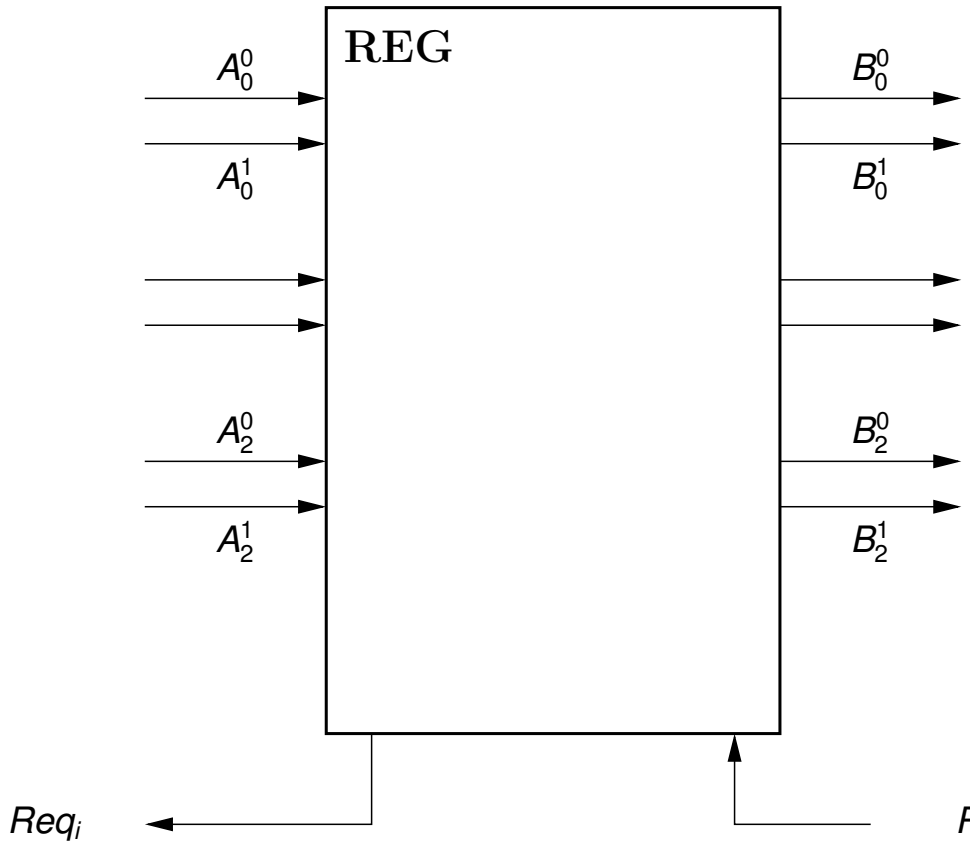
Q2.8 Soit un circuit à *n* sorties dual-rail. En supposant qu'il est correct (c'est-à-dire qu'aucune sortie ne peut passer par un état ou les deux rails sont à DATA), comment détecter en une porte seuil si le calcul de ce circuit est terminé ?

L'intérêt de détecter que le calcul est terminé est de passer cette information en amont du circuit, pour que ce qu'il y a en amont nous envoie les prochaines données. Ceci doit passer par une phase de reset de toutes les entrées à NULL, seule manière de faire descendre les sorties. Le fonctionnement d'un circuit complexe passe donc par des successions d'ondes de calcul et d'ondes de remise à NULL.
Un micropipeline se présente de la manière suivante :



Toutes les données sont en dual-rail. On suppose que l'extérieur réagit à Req_0 de la manière suivante : lorsque Req_0 monte, l'extérieur fournit un jeu de données après au plus 3 patasecondes. Lorsque Req_0 tombe, il met toutes les entrées à NULL après au plus 3 patasecondes.
Au début ($t < 0$) tous les Req_i sont à DATA et tous les autres signaux sont à NULL.
Lorsque Req_{i+1} monte, la boîte REG recopie ses entrées vers ses sorties au fur et à mesure qu'elles arrivent, en les mémorisant. Lorsque toutes ses sorties sont positionnées (plus précisément, lorsqu'un rail par bit dual-rail est monté), elle fait tomber sa sortie Req_i. Cela invite la tranche de calcul précédente à revenir à l'état NULL. En effet, lorsque Req_{i+1} tombe, la boîte REG fait tomber ses sorties au fur et à mesure que les entrées correspondantes tombent. Lorsque toutes ses sorties soient tombées, elle fait monter son Req_i pour initier un nouveau cycle de calcul.

Q2.9 Relisez 17 fois le paragraphe précédent puis construisez l'intérieur d'une des boites Reg. Vous avez droit aux portes seuil, et à l'inverseur DATA/NULL que vous dessinerez comme un inverseur normal.



Q2.10 Pourquoi cette usine à gaz serait-elle plus performante qu'un pipeline classique (obtenu en replaçant tous les Req par une horloge unique, et les REG par des registres classiques) ?

Q2.11 (question bonus à garder pour la fin pour ceux qui auront fini en avance) Proposez la construction d'un additionneur à insérer dans le pipeline précédent. Il devra être capable de finir vite les additions qui peuvent se finir vite, c'est-à-dire dont les chaînes de propagation de retenue sont courtes.
On rappelle l'observation déjà utilisée par Zuse en 1942 : lors de l'addition de deux nombres binaires $A = a_n...a_0$ et $B = b_n...b_0$,

- si $a_i = 1$ ET $b_i = 1$ on sait qu'on sortira une retenue à 1 sans avoir besoin d'attendre la retenue entrante ;
- si $a_i = 0$ ET $b_i = 0$ on sait qu'on sortira une retenue à 0 sans avoir besoin d'attendre la retenue entrante ;
- si a_iXORb_i on sait qu'on devra propager la retenue entrante à l'identique.

Attachez des feuilles en plus si nécessaire mais METTEZ VOTRE NOM DESSUS.

3 Les instructions de synchronisation à travers les âges (8 points)

Dans toute cette partie on considère un processeur RISC 32 bits à 3 opérandes, à 32 registres de 32 bits notés R0 à R31. Ce processeur possède toutes les instructions utiles, qui seront écrites dans un assembleur qui ressemble à ça :

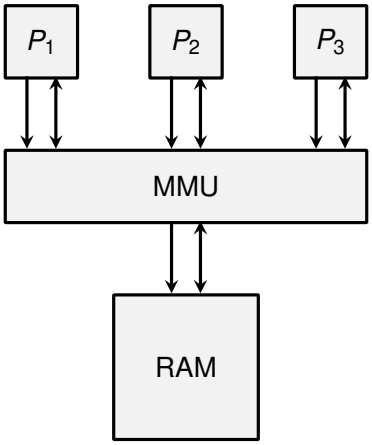
```
R0   <- R1+R2      ; opérations arithmétiques, en complément à 2
R0   <- R1+1       ; opérations arithmétiques immédiates
R0   <- 17         ; constante immédiate dans un registre
[R0] <- R21        ; écriture mémoire
R21  <- [R0]       ; lecture mémoire
CMP  R1, R2        ; compare R2 à R1 met à jour les drapeaux
JRGT -42           ; saut relatif (PC<-PC-42) si R1 > R2 (greater than)
CMP R1, -1         ; comparaison à un immediat en complément à 2
```

Les instructions arithmétiques ne travaillent que sur les registres, tandis que les instructions mémoire ne réalisent aucun calcul, juste des transferts entre registres et mémoire.
Toutes les instructions arithmétiques mettent à jour les drapeaux, et seuls les sauts sont conditionnels. Vous pourrez utiliser les sauts conditionnels JRZ ou JREQ (comparaison d'égalité, ce qui revient à dire que la différence est 0), JRNZ ou JRNE (not equal), JRGT (strictly greater than), JRGE (greater or equal). Le saut inconditionnel se note juste JR.
Dans tout ce problème on se satisfera de sauts relatifs, et on aura le droit d'abstraire les destinations de saut derrière des étiquettes, par exemple :

```
boucle: R0   <- R1+R2   ; opérations arithmétiques
        (...)           ; pas envie de compter combien d'instructions
        JR boucle       ; saut vers l'instruction d'étiquette "boucle"
```

Le `;` dans tous ces exemples marque le début d'un commentaire qui se finit à la fin de la ligne, et vous êtes encouragés à l'utiliser.

À présent que même les téléphones portables sont multicœurs, il faut s'intéresser au fonctionnement correct d'un logiciel réparti sur plusieurs cœurs se partageant une mémoire.
Chacun des cœurs est un processeur complet, avec ses registres, son PC, etc. Par contre, il n'y a qu'une mémoire pour tous les cœurs, avec un seul bus d'accès à cette mémoire (adresse+données). C'est donc une ressource partagée qu'il faut arbitrer. C'est le rôle de la MMU *(memory management unit)* : si plusieurs coeurs veulent accéder à la mémoire au même cycle, la MMU en choisit un, le sert, et fait patienter les autres.



3.1 Les enfants, comptez vous !

Attaquons-nous à un problème élémentaire : on veut que nos cœurs/threads [1] se comptent. Pour cela ils vont chacun incrémenter une case mémoire partagée, mettons à l'adresse contenue dans R0, et contenant initialement 0. Autrement dit, chacun doit faire, en syntaxe C [2],

```
*R0 = *R0+1;
```

Q3.1 Donnez l'assembleur correspondant à `*R0 = *R0+1;` (trois instructions).

Q3.2 Décrivez une exécution parallèle de ce même assembleur sur trois coeurs, qui fait que le compte final est tout faux. Les trois cœurs peuvent démarrer ensemble, ou pas. Ils peuvent exécuter leurs instructions en parallèle, mais attention, il est impossible de faire deux accès à la mémoire au même cycle : la MMU y veille. Il peuvent donc passer un cycle ou plus à ne rien faire parce que la MMU les fait patienter.

cycle	processeur 1	processeur 2	processeur 3	`[R0]`
1				0
2				
3				
4				
5				
6				
7				
8				
9				

La solution classique est la suivante. Un verrou *(lock)* est un mécanisme qui garantit qu'au plus un thread/cœur a accès à une ressource (par exemple le compteur évoqué ci-dessus) à un instant donné. On parle aussi d'exclusion mutuelle, ou parfois *mutex*.
Un verrou est soit ouvert (0) soit fermé (1).
Pour un identifiant de verrou V (en pratique une adresse mémoire), votre OS vous fournira trois primitives :

- InitLock(V) : initialiser le verrou à 0 (ouvert).
- Lock(V) : verrouiller
- UnLock(V) : déverrouiller

qu'un thread utilise typiquement comme ceci :

```
Lock(V);
(... accès à la ressource critique ...)
UnLock(V);
```

À l'appel à `Lock(V)`,

- soit le verrou est ouvert. Alors le thread le ferme et continue.
- soit le verrou est fermé. Alors le thread attend qu'il s'ouvre, et dès qu'il s'ouvre, il le ferme et continue. Si plusieurs threads attendaient, un seul ferme le verrou, les autres continuent à attendre.

L'appel de `UnLock(V)` ouvre le verrou.

1. Ici, un thread est le programme qui tourne sur un cœur.
2. Si vous ne connaissez pas la syntaxe C, `*R0;` dénote le contenu de la case mémoire dont l'adresse est donnée par R0. Cette ligne ne change donc pas R0 mais la casse mémoire pointée par R0.

3.2 Une instruction qui n'est pas dans le poly

On peut implémenter un verrou en utilisant une instruction à deux opérandes `TST Rd, [Rn]` *(test and set)* dont la sémantique est la séquence d'actions décrite par le pseudo-C suivant :

```
Rd=*Rn; if (Rd==0) {*Rn=1; Z=0}
```

(Z est le drapeau Z). Cette instruction est atomique, c'est-à-dire ininterruptible. Durant son exécution, les interruptions sont désactivées. En particulier, le processeur ne changera pas de contexte/processus/thread. De plus, le processeur qui exécute un TST a l'exclusivité de l'accès à la mémoire tant que cette instruction n'est pas terminée.

Q3.3 Donnez du code assembleur qui implemente Lock() et Unlock() en utilisant cette instruction. En entrée le registre R0 contiendra l'adresse du verrou. Convainquez-moi en une phrase que deux processeurs ne peuvent prendre le verrou en même temps.

```
Lock:                                   Unlock:
```

3.3 Synchronisation

J'ai à présent 32 cœurs, donc 32 threads, numérotés de 0 à 31. On optimise le rendu graphique de votre jeu de baston préféré en coupant la scène en 32 morceaux et en donnant 1/32ième du travail à chaque processeur. Mais il est difficile de prévoir exactement combien de temps va prendre le traitement de chaque morceau. Heureusement il y a un chef (disons le thread 0). Il veut savoir quand tous les threads ont fini. Pour cela, je ressors mon problème de comptage : les 32 threads vont se partager une case mémoire contenant un compteur, initialisé à 0 et incrémenté par chaque thread lorsqu'il finit son travail.

Q3.4 Écrivez en pseudo-C la synchronisation utilisant les primitives Lock() et Unlock(). L'adresse du verrou sera dans R0, et l'adresse du compteur dans R1. On supposera ces deux cases mémoires initialisées à 0. Vous écrirez à gauche le code exécuté par le thread 0 (le chef) et à droite le code exécuté par les threads 1 à 31, lorsqu'ils ont fini.

```
// end of work for thread 0
```

```
// end of work for thread 1 to 31
```

```
// Wait for the other threads to finish
while (*R1!=31) {};
// and now, proceed
```

```
// and now, wait for more work
```

Q3.5 Pour chaque boucle cachée dans les deux questions précédentes, convainquez-moi que ce n'est pas une boucle infinie. Pouvez-vous borner le nombre d'itérations ?

3.4 Une autre instruction qui n'est pas dans le poly

Avec l'arrivée des processeurs RISC à trois opérandes, on a vu arriver des instructions plus compliquées. Par exemple, dans le jeu d'instruction ARM, on trouve cette instruction *swap* bizarre :
`SWP Rd, Rm, [Rn]` dont la sémantique est décrite ainsi dans la doc :
`temp = *Rn;  *Rn = Rm;  Rd = temp;`
Le `temp` ci-dessus est utile lorsque `Rd` et `Rm` sont le même registre, auquel cas on échange *(swap)* le contenu de ce registre et la case mémoire.
À nouveau, cette séquence d'action est atomique, et garantit au processeur qui l'exécute l'exclusivité de l'accès à la mémoire.

Q3.6 Utilisez `SWP` pour écrire en assembleur le code de `Lock` et `Unlock`. En entrée le registre R0 contiendra l'adresse du verrou. Convainquez-moi en une phrase que deux processeurs ne peuvent prendre le verrou en même temps.

```
Lock:                                   Unlock:
```

Q3.7 Implémenter la synchronisation du **??** utilise deux cases mémoires : une pour le verrou, et une pour le compteur partagé. Montrez qu'avec SWP on peut mieux faire, en utilisant *une seule case mémoire* qui sera pointée par R0 (et qu'on suppose toujours initialisée à 0). On demande le code correspondant en assembleur.

```
; end of work for thread 1 to 31









; and now, wait for more work
```

3.5 L'atomicité en deux instructions

La mémoire étant de plus en plus lente comparativement au processeur, un SWP, qui réalise deux accès mémoire, peut durer des dizaines de cycles. Et pendant ce temps les interruptions ne sont pas servies. Pour certaines applications, c'est un problème. De plus, le SWP bloque tout le bus mémoire alors qu'on ne veut en fait bloquer qu'une adresse. Les processeurs récents introduisent plus de flexibilité en basant les synchronisations sur deux nouvelles instructions que voici :

- `Rd <-! [Rn]` réalise `Rd <- [Rn]`. De plus, l'adresse pointée par `Rn` se retrouve marquée *(tagged)* dans ce processeur comme "tentative d'accès exclusif".
- Toute écriture ultérieure à cette adresse (par ce processeur ou par un autre) effacera cette marque/*tag*.
- `[Rn] <-! Rm` fonctionne comme suit :
  - Si l'adresse pointée par `Rn` est toujours marquée comme exclusive dans ce processeur, alors cette instruction réalise `[Rn] <- Rm`, et positionne le drapeau Z à 1.
  - Sinon, elle ne modifie pas la mémoire, et positionne le drapeau Z à 0.

Q3.8 Écrivez en assembleur le code de synchronisation par compteur partagé (toujours stocké dans R0) des threads 1 à 31, en utilisant ces deux nouvelles instructions. Convainquez-moi en une phrase que cela marche.

```
; end of work for thread 1 to 31




; and now, wait for more work
```

Q3.9 Le marquage exclusif est associé à chaque adresse dans chaque processeur. Pourquoi ne pas l'associer à une adresse de manière globale ? Ce serait plus économique. Pour se convaincre que ce n'est pas une bonne idée, supposons donc que c'est le cas. Donnez alors un exemple d'exécution de votre code de comptage qui donne le mauvais compte. Indice : il faut au moins trois processeurs.

3.6 Yapuka le construire

Associer un bit à chaque adresse dans chaque processeur nécessiterait une mémoire énorme.
Mais l'ISA ARM précise qu'on n'a le droit d'utiliser un `[Rn] <-! Rm` que pour l'adresse qui correspond au plus récent `Rd <-! [Rn]`. Sinon, le comportement n'est pas défini. Normalement, ce devrait être le cas de vos utilisations jusqu'ici. Vérifiez.

Q3.10 Cette restriction permet une implémentation peu coûteuse de ces deux instructions. Que suffit-il d'ajouter à la MMU ? Soyez précis. On ignorera ici le problème des interruptions qui est traité dans la question suivante (allez la lire).

Q3.11 On peut imaginer qu'une interruption arrive entre un `Rd <-! [Rn]` et le `[Rn] <-! Rm` correspondant. La routine de traitement d'interruption peut avoir besoin elle-même d'accéder à un autre verrou, et donc réaliser elle-même un `Rd <-! [Rn]`. Au retour de cette routine, on aura alors une violation de la condition précédente "on n'a le droit d'utiliser un `[Rn] <-! Rm` que pour l'adresse qui correspond au plus récent `Rd <-! [Rn]`".
Suggérez une solution matérielle simple à ce problème.

3.7 Bonus

Q3.12 Il y a (au moins) deux réponses correctes à la question Q3.7. Donnez la seconde, et dites laquelle est la plus performante.