Circuits Numériques et Éléments d'Architecture
Examen

ENSIMAG 1A

Année scolaire 2021–2022

Informations concernant l'épreuve

- durée : 3h ;
- documents autorisé : une feuille A4 manuscrite recto-verso ;
- le barème est donné à titre indicatif ;
- les questions « bonus » ne sont à traiter que lorsque toutes les questions du même exercice ont été traitées et ne sont pas obligatoires ;
- les exercices sont indépendants et peuvent être traités dans le désordre, et certaines questions au sein du même exercice peuvent aussi être traitées indépendamment.

Ex. 1 : Questions de cours (2 pts)

Question 1 *(1.5pt)* Soit un processeur RISC-V possédant un cache de 1er niveau de 32 KiB (soit 32768 octets) avec des lignes de 16 mots, sachant qu'un mot est constitué de 4 octets. Ce cache est à correspondance directe, comme celui présenté en cours. Combien de lignes contient ce cache ? En supposant que sur cette machine l'adresse est d'une taille de 39 bits (même si on la vend pour 64 !), sur combien de bits sont codés (a) le champ *offset*, (b) le champ *index* et (c) le champ *tag* ?

16 mots par ligne ⇒ 64 octets par ligne, d'où $\frac{32\times1024}{64} = 512$ lignes.
(a) 4 pour identifier 1 mot parmi 16 dans la ligne ; (b) 9 pour identifier 1 ligne parmi 512 ; (c) $39-9-4-2$ (ce dernier 2 est nécessaire pour identifier 1 octet parmi 4 dans le mot), soit 24 bits.

Question 2 *(0,5pt)*

- classez par ordre de vitesse d'une part et de nombre de transistors d'autre part : registre, DRAM, SRAM (genre x > y > z)

vitesse : registre > sram > dram, surface registre > sram > dram

Question 3 *(0,5pt)* (bonus)

- Pour sauver la planète, vaut-il mieux que l'humanité arrête de prendre l'avion ou d'utiliser un ordinateur ?

informatique = 4% des émissions de GES, transport aérien = 2% => travaillez chez Airbus plutôt que chez Cap Gémini ! Fallait faire l'ENAC plutôt que l'Ensimag, ...

Ex. 2 : Circuits combinatoires (2 pts)

Les 7 premières valeurs de la séquence de Recamán [1] sont $0,1,3,6,2,7,13$. On cherche à réaliser un circuit combinatoire dont l'entrée e codée sur 4 bits ($e_3e_2e_1e_0$) reçoit un nombre entre 0 et 13 (inclus) et produit une sortie codée sur 2 bits notés s_1s_0. Le bit de poids faible (s_0) indique si le nombre appartient aux 7 premières valeurs de la suite (1 s'il y appartient, 0 sinon), celui de poids fort (s_1) si le nombre de bits à 1 dans la représentation binaire du nombre est pair (par ex. 1110 est tel que $s_1 = 0$, 1001 est tel que $s_1 = 1$). Lorsque le nombre n'appartient pas à la suite, la valeur de s_1 n'est pas spécifiée et on peut la choisir à 0 ou 1.
On notera e_i le bit en position i de l'entrée.

Question 1 *(0,5pt)* Donnez la table de vérité des s_i en fonction des e_i.

$e_3e_2e_1e_0$	s_1	s_0
0000	1	1
0001	0	1
0010	0	1
0011	1	1
0100	Φ	0
0101	Φ	0
0110	1	1
0111	0	1
1000	Φ	0
1001	Φ	0
1010	Φ	0
1011	Φ	0
1100	Φ	0
1101	0	1
1110	Φ	Φ
1111	Φ	Φ

Question 2 *(1pt)* Donnez les expressions simplifiées des s_i à l'aide d'une table de Karnaugh.

Question 3 *(0,5pt)* Faites les schémas en portes logiques qui réalisent ces expressions booléennes. Vous pouvez utiliser des portes à plus de 2 entrées.

Pour s_1 :

e_3e_2 \ e_1e_0	**00**	**01**	**11**	**10**
00	1	0	1	0
01	ϕ	ϕ	0	1
11	ϕ	0	ϕ	ϕ
10	ϕ	ϕ	ϕ	ϕ

D'où $s_1 = \overline{e_1}\,\overline{e_0} + e_2\overline{e_0} + \overline{e_2}e_1e_0$.

1. Séquence A005132 de *The On-Line Encyclopedia of Integer Sequences.*

Pour s_0 :

e_3e_2 \ e_1e_0	**00**	**01**	**11**	**10**
00	1	1	1	1
01	0	0	1	1
11	0	1	ϕ	ϕ
10	0	0	0	0

D'où $s_0 = \overline{e_3}\,\overline{e_2} + \overline{e_3}e_1 + e_3e_2e_0$.

Ex. 3 : Circuits séquentiels (2 pts)

Le schéma de la figure 1 représente un « encodeur Manchester différentiel ». Ce type de codage est par exemple utilisé pour échanger des données entre un chargeur USB à la norme *Power Delivery* et votre téléphone. Le signal d'horloge est noté `ck`, l'entrée du circuit `e` et sa sortie `s`. Complétez le chronogramme de la figure 3, en annexe, qui donne l'état de la sortie `s` et des signaux `q0` et `q1` au cours du temps, le comportement de l'entrée `e` et la valeur initiale des bascules (`q0` et `q1`) étant définis sur ce chronogramme par des traits épais.

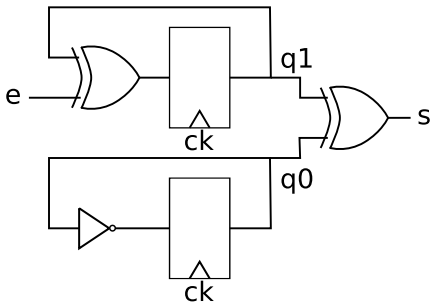
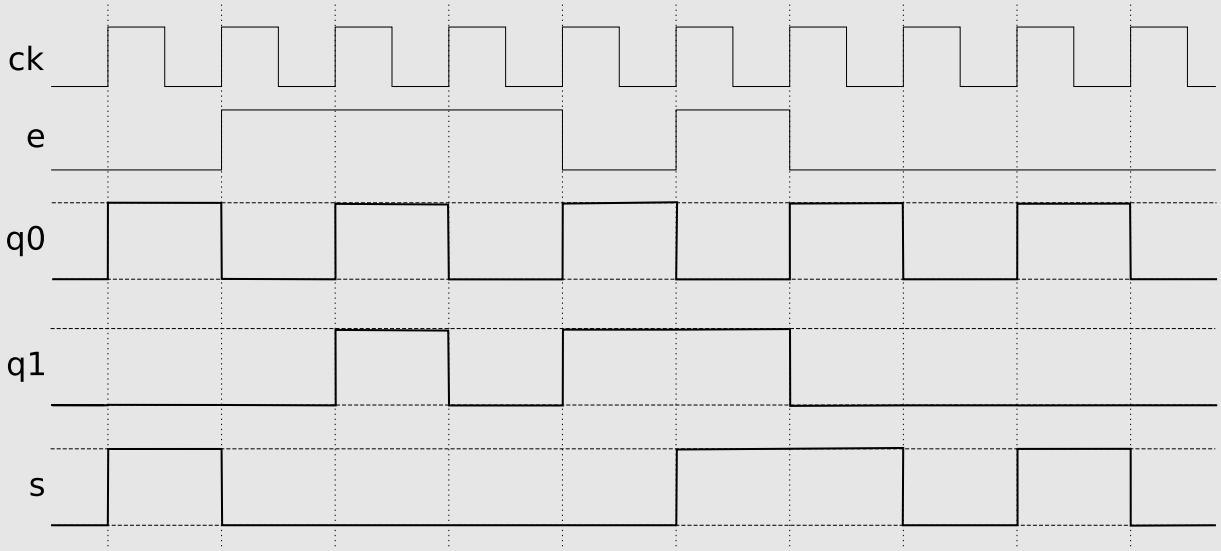


FIGURE 1 – Schéma à étudier



Ex. 4 : Synthèse d'automate (4 pts)

Les processeurs modernes sont des boules de cristal, ils passent leur temps à spéculer ! Ils spéculent en particulier si un branchement conditionnel sera pris ou non, grâce à des « prédicteurs de branchement ». La tâche est titanesque, et ils utilisent classiquement deux prédicteurs, l'un local et l'autre global [2]. Ces deux prédicteurs étant indépendants, pour un branchement donné il faut prédire, parmi les deux, celui qui donnera la meilleure prédiction ! C'est là qu'intervient l'automate d'état de la figure 2 que nous nous proposons d'étudier.
On note l une entrée qui vaut 1 si la prédiction locale précédente était incorrecte, 0 sinon, et g une entrée qui vaut 1 si la prédiction globale précédente était incorrecte, 0 sinon. On note p la sortie de l'automate, qui vaut 0 si on choisit le prédicteur local (états L1 et L0), et 1 si on choisit le prédicteur global (états G1 et G0). Sur l'automate d'état, les conditions de transitions sont étiquetés avec des conditions de ces variables.

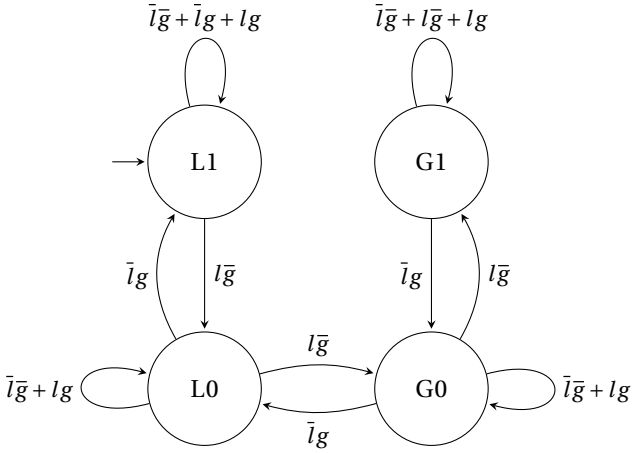


FIGURE 2 – Machine d'états du prédicteur de prédicteurs

On fait l'hypothèse que l'état initial est L1. On encode les états ainsi : L1 = 00, L0 = 01, G0 = 11, G1 = 10. On note $\mathtt{d}_i$ les bits d'entrée du registre d'état, et $\mathtt{q}_i$ ses sorties.

Question 1 *(1.5pt)* Donnez la table de transition exprimant les $\mathtt{d}_i$ et la sortie p en fonction des $\mathtt{q}_i$ et des entrées l et g.

état courant	$\mathtt{q}_1\mathtt{q}_0$	p	lg	état futur	$\mathtt{d}_1\mathtt{d}_0$
L1	00	0	00	L1	00
L1			01	L1	00
L1			10	L0	01
L1			11	L1	00
L0	01	0	00	L0	01
L0			01	L1	00
L0			10	G0	11
L0			11	L0	01
G1	10	1	00	G1	10
G1			01	G0	11
G1			10	G1	10
G1			11	G1	10
G0	11	1	00	G0	11
G0			01	L0	01
G0			10	G1	10
G0			11	G0	11

2. Pour les détails suivre l'excellent cours d'archi de 2A !

Question 2 *(2,5pt)* Donnez les équations simplifiées des d_i et de la sortie p, et le schéma en portes des d_i.

Trivialement, $p = q_1$. Voyons les entrées du registre d'état à présent.

$d_0 = q_0\bar{l}\bar{g} + q_1\bar{l}g + q_0lg + \overline{q_1}l\bar{g}$

q_1q_0 \ lg	00	01	11	10
00	0	0	0	1
01	1	0	1	1
11	1	1	1	0
10	0	1	0	0

$d_1 = q_1\overline{q_0} + q_1l + q_1\bar{g} + q_0l\bar{g}$

q_1q_0 \ lg	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	1	0	1	1
10	1	1	1	1

Question 3 *(1pt)* (bonus) On fait à présent l'hypothèse que l'encodage choisi est de type *one-hot*. Donnez le schéma complet de l'automate.

Ex. 5 : Conception PC/PO d'un algorithme d'« extraction » (6 pts)

On se propose d'implanter un algorithme d'« extraction » en matériel. Cet algorithme prend deux entiers non signés en entrée : une valeur x et un masque m, tous deux considérés comme des vecteurs de bits. Il produit en sortie un nouveau vecteur de bits tels que, en partant de la gauche vers la droite, les bits de x qui sont à une position à laquelle le bit correspondant de m est à 1 se trouvent « entassés » sur la droite. Prenons deux exemples sur 8 bits :

	x = 11001100	x = 11011010
	m = 10011000	m = 11110000
	1 01	1101
alors	r = 00000101	r = 00001101

Voici un algorithme naïf pour faire cette opération :

```
1  uint32_t extract(uint32_t x, uint32_t m)
2  {
3      uint32_t r, s, b;
4      r = 0;
5      s = 0;
6      while (m != 0) {
7          b = m & 1;
8          if (b == 1) {
9            r = r | ((x & 1) << s);
10           s = s + 1;
11         }
12         x = x >> 1;
13         m = m >> 1;
14     }
15     return r;
16 }
```

— on ne cherchera pas à comprendre comment fonctionne cet algorithme, on le prendra donc comme une entrée de l'exercice sans plus se préoccuper de sa fonction ;

— les variables sont typées, elles sont sur 32 bits, non signées ;

— l'opérateur `&` (respectivement `|`) effectue le *et* (respectivement *ou*)-logique bit à bit entre ses opérandes ;

— l'opération `v >> p` décale le mot binaire `v` à droite de `p` positions, `p` '0' étant injectés à gauche et les `p` bits de poids faibles étant perdus ;

— l'opération `v << p` décale le mot binaire `v` à gauche de `p` positions, `p` '0' étant injectés à droite et les `p` bits de poids forts étant perdus.

Pour mémoire, des instructions qui peuvent s'effectuer en parallèle peuvent être écrites sous forme d'affectations concurrentes (ex. $(a, b) = (z, t)$), et l'on peut aussi affecter conditionnellement une variable (ex. $n = z > t \ ? \ 1 : 0$).

Question 1 *(0,5pt)* D'un point de vue matériel, à quoi correspond le calcul de `b` ligne 7 ?

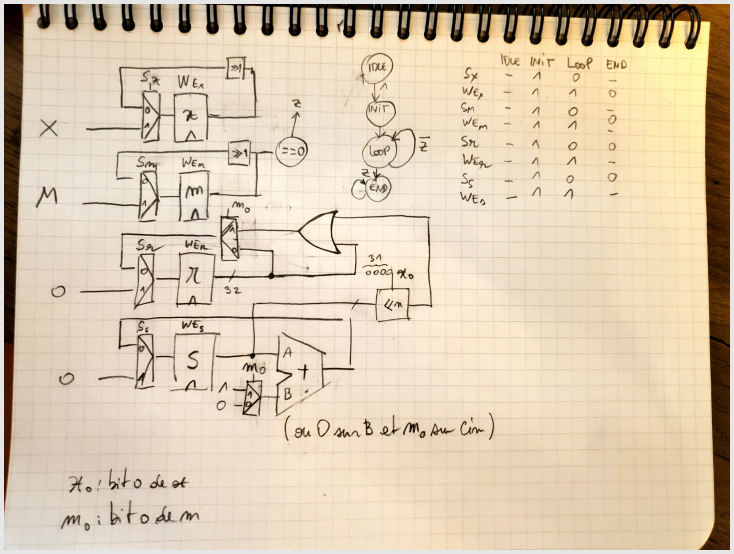
C'est juste la lecture du bit de poids faible de `m`.

Question 2 *(1,5pt)* On considère `r, s, x, m` comme des registres. On cherche à exécuter concurremment les lignes qui le peuvent, en prenant en compte les affectations conditionnelles lorsque cela est possible. Réécrivez l'algorithme avec les assignations concurrentes et affectations conditionnelles.

```
1  uint32_t extract(uint32_t x, uint32_t m)
2  {
3      uint32_t r, s, b;
4      (r, s) = (0, 0);
5      while (m != 0) {
6          (r, s, x, m) = (m & 1 ? (r | ((x & 1) << s) : r, m & 1 ? s + 1 : s, x >> 1, m >> 1);
7      }
8      return r;
9  }
```

Question 3 *(1,5pt)* Proposez une partie opérative permettant de réaliser cet algorithme. Pensez à faire apparaître et nommer les différents signaux de contrôle et de compte-rendus, et à préciser les numéros d'entrées sur les mux. On propose les conventions de nommage suivantes : l'autorisation d'écriture du registre `A` est notée `WEa`, le signal de sélection des multiplexeurs à l'entrée d'un registre `A` est noté `Ma`, et le signal de sélection du multiplexeur devant l'entrée `A` de operateur `X` est noté `SXa`.

Question 4 *(1,5pt)* Donnez la machine à états permettant de contrôler votre chemin de données. Comme en TD, faites une table avec les signaux et les états et donnez les valeurs des signaux de contrôle pour chaque état.



Question 5 *(1pt)* (bonus) Modifiez votre algorithme de la question 2 de sorte à faire disparaître l'affectation conditionnelle. Sans faire l'implantation en matériel, expliquez en une phrase l'intérêt de cette nouvelle version du point de vue du matériel.

```
1  uint32_t extract(uint32_t x, uint32_t m)
2  {
3      uint32_t r, s;
4      (r, s) = (0, 0);
5      while (m != 0) {
6          (r, s, x, m) = (r | ((x & m & 1) << s), s + (m & 1), x >> 1, m >> 1);
7      }
8      return r;
9  }
```

D'un point de vue matériel, il n'y a plus besoin de multiplexeurs, on met m0 sur cin pour s (ou on utilise un incrémenteur), l'entrée du shifter un bête *et* entre m0 et x0, et la sortie du *ou* est directement connectée à l'entrée de r.
Cette implèm est aussi plus efficace en soft, car on n'a plus besoin de branchement.

Ex. 6 : Conception de processeur (4 pts)

L'objectif de cet exercice est d'ajouter deux nouvelles instructions au processeur 2-adresses étudié durant les TD 8, 9, 10 et 11. Pour mémoire, le jeu d'instructions, la partie opérative ainsi que la partie contrôle (sans signaux et conditions) sont rappelés en annexe. La première instruction, `call AD`, appelle la fonction (au sens langage de programmation) dont l'adresse est `AD`. Simultanément, elle sauvegarde dans un registre dédié l'adresse de l'instruction qui suit le `call`. La seconde instruction, `ret`, est utilisée à la fin de la fonction, elle permet de revenir à l'instruction qui suit le `call`. Puisqu'on utilise un seul registre pour sauvegarder une adresse de retour, il est impossible d'imbriquer les appels de fonctions (programme qui appelle f, f qui appelle g, etc.).

Question 1 *(0,5pt)* Proposez un encodage des instructions `call AD` et `ret`.

`call` nécessite la valeur de l'adresse `AD` sur 2 octets et rien d'autre. Comme ça ressemble à un jump, on le met, par convention, après le jz.

Instruction	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
`call AD`	1	0	0	0	0	1	1	0
	ADH							
	ADL							

`ret` n'a besoin d'aucune autre informaton que de son opcode, mais comme il ne reste pas bezef d'espace d'encodage, on le met après call.

Instruction	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
`ret`	1	0	0	0	0	1	1	1

Question 2 *(1,5pt)* On s'intéresse à l'instruction `call AD`.
Quels sont éventuellement les éléments à ajouter dans la partie opérative pour pouvoir réaliser cette instruction ? Vous pouvez ajouter ces éléments sur l'annexe et la rendre avec votre copie.

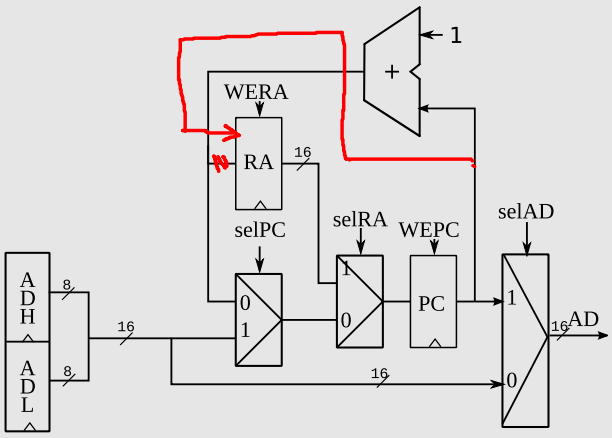
On doit ajouter un registre (16 bits) qui recopie la sortie de l'additionneur qui suit `pc` pour sauver l'adresse de retour. On l'appelle RA.

Question 3 *(0,5pt)* Ajoutez dans la partie contrôle (à compléter sur l'annexe et à rendre avec votre copie) les états nécessaires à l'exécution de l'instruction `call AD` en précisant la valeur des différents signaux (valeurs des éventuels nouveaux signaux pour chaque état et valeur de tous les signaux dans les éventuels nouveaux états), en suivant le pricipe de ce qui a été fait en TD et qui est rappelé en annexe.

On doit ajouter une transition depuis l'état ADL vers un nouvel état call si IR code `call AD`. Dans cet état, on fait PC <= AD // RA <= PC + 1, et l'état futur est fetch
Après avoir corrigé 2 groupes : il y a des solutions rigolotes avec réutlisation de l'état JUMP. Ca fait un état de plus mais c'est malin.

On passe maintenant à l'instruction `ret`.

Question 4 *(1pt)* Quels sont éventuellement les élements à modifier dans la partie opérative pour pouvoir faire cette instruction ? Vous pouvez ajouter ces éléments sur l'annexe et la rendre avec votre copie.



Question 5 *(0,5pt)* Ajoutez dans la partie contrôle (à compléter sur l'annexe et à rendre avec votre copie) les états nécessaires à l'exécution de l'instruction `ret` en précisant la valeur des différents signaux (valeurs des éventuels nouveaux signaux pour chaque état et valeur de tous les signaux dans les éventuels nouveaux états).

Après l'état `décode` si `IR` code `ret`, il faut aller dans un état `ret` dans lequel on fait PC <= RA, puis on retourne dans fetch. On ne touche à rien, sauf à SELra qui doit présenter RA en sortie, et WEpc qui est mis à 1 (comme en gros dans tous les états qui retournent à fetch).

Signaux/États	Fetch	Decode	OP	ADH	ADL	ST	LD	JMP	CALL	RET	Init
CE^*	0	1	1	0	0	0	0	1	1	1	0
WE^*	1	ϕ	ϕ	1	1	0	1	ϕ	ϕ	ϕ	ϕ
$selPC$	0	ϕ	ϕ	0	0	ϕ	ϕ	1	1	ϕ	ϕ
$WEPC$	1	0	0	1	1	0	0	1	1	1	0
$selRA$	0	ϕ	ϕ	0	0	ϕ	ϕ	0	0	1	ϕ
$WERA$	0	0	0	0	0	0	0	0	1	ϕ	ϕ
$selAD$	1	ϕ	ϕ	1	1	0	0	ϕ	ϕ	ϕ	ϕ
$WEAdH$	ϕ	ϕ	ϕ	1	0	ϕ	ϕ	ϕ	ϕ	ϕ	0
$WEAdL$	ϕ	ϕ	ϕ	ϕ	1	ϕ	ϕ	ϕ	ϕ	ϕ	0
$WEIR$	1	0	0	0	0	0	0	0	0	0	0
$selR$	ϕ	ϕ	0	ϕ	ϕ	ϕ	1	ϕ	ϕ	ϕ	ϕ
$WER0$	0	0	$\overline{IR_0}.\overline{IR_1}$	0	0	0	$\overline{IR_0}.\overline{IR_1}$	0	0	0	0
$WER1$	0	0	$IR_0.\overline{IR_1}$	0	0	0	$IR_0.\overline{IR_1}$	0	0	0	0
$WER2$	0	0	$\overline{IR_0}.IR_1$	0	0	0	$\overline{IR_0}.IR_1$	0	0	0	0
$WER3$	0	0	$IR_0.IR_1$	0	0	0	$IR_0.IR_1$	0	0	0	0
$selA$	$\phi\phi$	$\phi\phi$	$IR_1 IR_0$	$\phi\phi$	$\phi\phi$	$IR_1 IR_0$	$\phi\phi$	$\phi\phi$	$\phi\phi$	$\phi\phi$	$\phi\phi$
$selB$	$\phi\phi$	$\phi\phi$	$IR_3 IR_2$	$\phi\phi$	$\phi\phi$	$\phi\phi$	$\phi\phi$	$\phi\phi$	$\phi\phi$	$\phi\phi$	$\phi\phi$
op	$\phi\phi\phi$	$\phi\phi\phi$	$IR_6 IR_5 IR_4$	$\phi\phi\phi$	$\phi\phi\phi$	$\phi\phi\phi$	$\phi\phi\phi$	$\phi\phi\phi$	$\phi\phi\phi$	$\phi\phi\phi$-	$\phi\phi\phi$

Question 6 *(1pt)* (bonus) (question à ne faire que si vous avez fini tout le reste, ...)
On fait l'hypothèse que l'on veut à présent supporter des appels à `call` imbriqués. Proposez conceptuellement un mécanisme qui le permette, et faites une proposition architecturale pour le supporter.

Des calls imbriqués impliquent des retours imbriqués, soit un mécanisme de pile. On a une solution générale, qui passe à l'échelle avec la quantité de mémoire : Il faut ajouter un nouveau registre d'adresse dans notre processeur, et un mux supplémentaire qui va vers AD. Il faut ensuite incrémenter (de 2) ce registre et sauver PC + 1 lors des calls, et récupérer l'ancienne valeur de PC pointée par ce registre et le décrémenter (de 2) lors d'un ret. Une solution ad-hoc utilisant une pile matérielle destinée à cet usage est également possible, mais alors on est limité intrinsèquement par la taille de cette pile.

Annexe

NOM : PRENOM : GROUPE :

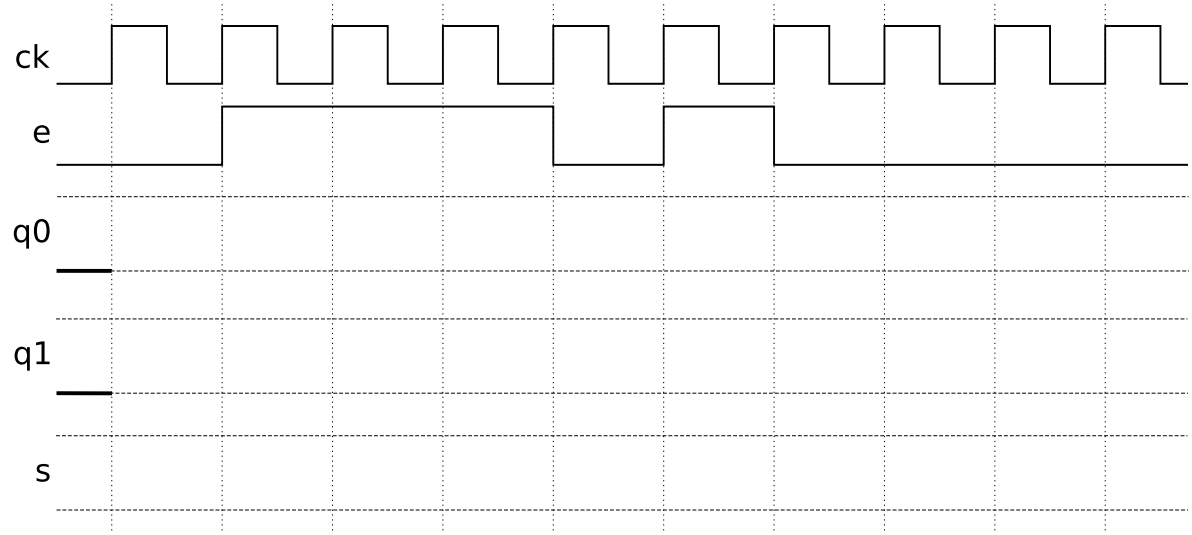
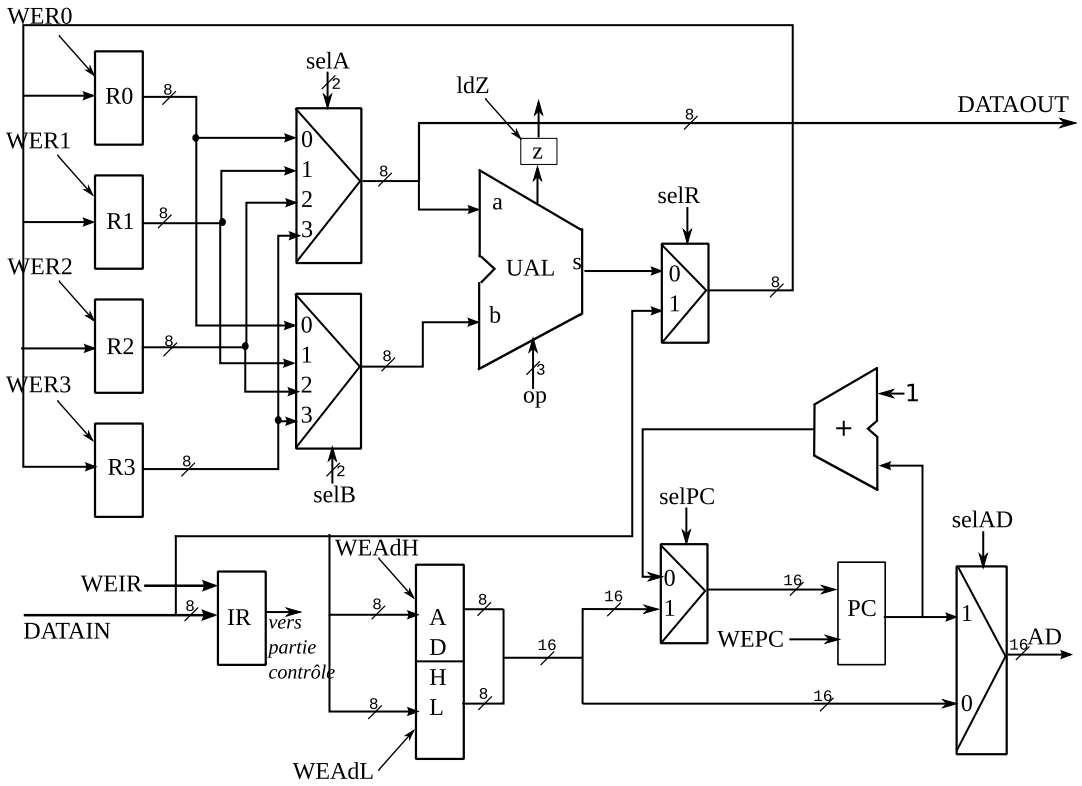


FIGURE 3 – Chronogramme

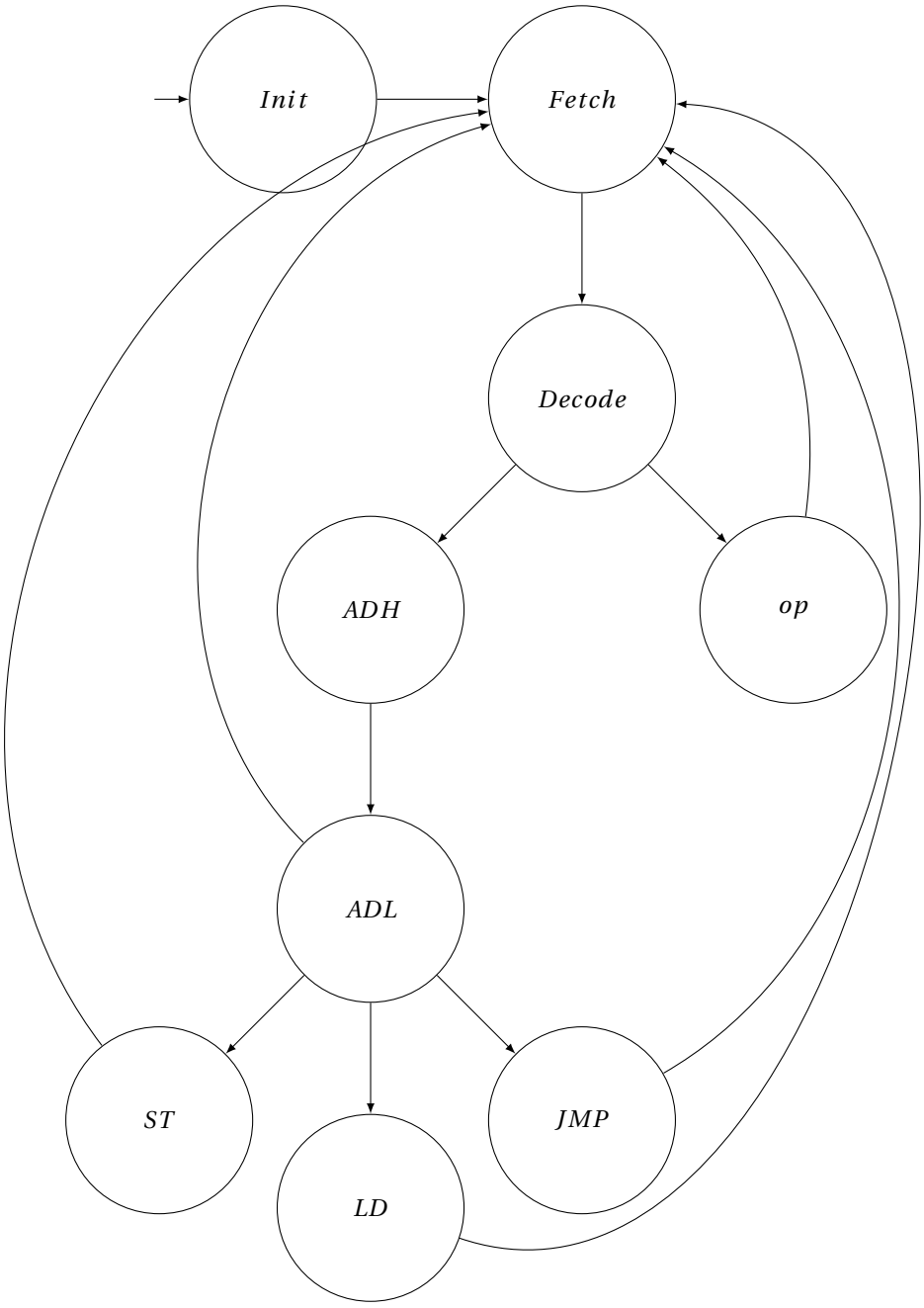
Jeu d'instructions et encodage

Instruction	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Opérations à 2 registres : $rd := rd\ op\ rs$								
`or rs, rd`	0	0	0	0	rs_1	rs_0	rd_1	rd_0
`xor rs, rd`	0	0	0	1	rs_1	rs_0	rd_1	rd_0
`and rs, rd`	0	0	1	0	rs_1	rs_0	rd_1	rd_0
`add rs, rd`	0	1	0	0	rs_1	rs_0	rd_1	rd_0
`sub rs, rd`	0	1	0	1	rs_1	rs_0	rd_1	rd_0
Opérations à 1 registre : $rd := op\ rd$								
`not rd`	0	0	1	1	0	0	rd_1	rd_0
`shl rd`	0	1	1	0	0	0	rd_1	rd_0
`shr rd`	0	1	1	1	0	0	rd_1	rd_0
Stockage : $MEM(AD) := rs$								
`st rs, AD`	1	1	0	0	0	0	rs_1	rs_0
					ADH			
					ADL			
Chargement : $rd := MEM(AD)$								
`ld AD, rd`	1	0	0	0	0	0	rd_1	rd_0
					ADH			
					ADL			
Chargement de constante : $rd := imm8$								
`li imm8, rd`	1	0	1	0	0	0	rd_1	rd_0
					imm8			
Branchement inconditionnel : $PC := AD$								
`jmp AD`	1	0	0	0	0	1	0	0
					ADH			
					ADL			
Branchement conditionnel : si $COND$ alors $PC := AD$								
`jz AD`	1	0	0	0	0	1	0	1
					ADH			
					ADL			

Partie opérative



Partie Contrôle



Signaux/États	Fetch	Decode	OP	ADH	ADL	ST	LD	JMP	Init
CE^*	0	1	1	0	0	0	0	1	1
WE^*	1	ϕ	ϕ	1	1	0	1	ϕ	ϕ
$selPC$	0	ϕ	ϕ	0	0	ϕ	ϕ	1	ϕ
$WEPC$	1	0	0	1	1	0	0	1	0
$selAD$	1	ϕ	ϕ	1	1	0	0	ϕ	ϕ
$WEAdH$	ϕ	ϕ	ϕ	1	0	ϕ	ϕ	ϕ	0
$WEAdL$	ϕ	ϕ	ϕ	ϕ	1	ϕ	ϕ	ϕ	0
$WEIR$	1	0	0	0	0	0	0	0	0
$selR$	ϕ	ϕ	0	ϕ	ϕ	ϕ	1	ϕ	ϕ
$WER0$	0	0	$\overline{IR_0}.\overline{IR_1}$	0	0	0	$\overline{IR_0}.\overline{IR_1}$	0	0
$WER1$	0	0	$IR_0.\overline{IR_1}$	0	0	0	$IR_0.\overline{IR_1}$	0	0
$WER2$	0	0	$\overline{IR_0}.IR_1$	0	0	0	$\overline{IR_0}.IR_1$	0	0
$WER3$	0	0	$IR_0.IR_1$	0	0	0	$IR_0.IR_1$	0	0
$selA$	$\phi\phi$	$\phi\phi$	IR_1IR_0	$\phi\phi$	$\phi\phi$	IR_1IR_0	$\phi\phi$	$\phi\phi$	$\phi\phi$
$selB$	$\phi\phi$	$\phi\phi$	IR_3IR_2	$\phi\phi$	$\phi\phi$	$\phi\phi$	$\phi\phi$	$\phi\phi$	$\phi\phi$
op	$\phi\phi\phi$	$\phi\phi\phi$	$IR_6IR_5IR_4$	$\phi\phi\phi$	$\phi\phi\phi$	$\phi\phi\phi$	$\phi\phi\phi$	$\phi\phi\phi$	$\phi\phi\phi$