EXAMEN D'ELECTRONIQUE NUMERIQUE INTEGREE – Janvier 2011 *Note* ____ *Barême* x *points*

Durée 2h30 - Tous documents et calculatrice autorisés sauf Ordinateur - **Répondre exclusivement sur ces feuilles à rendre**
(∇) : Tous les circuits sont supposés de vitesse grande vis à vis de l'horloge pour négliger les brefs états transitoires.
En revanche, les temps de propagation des circuits ne doivent pas être négligés.

NOM - Prénom : ____ **Groupe :** ____

1. Transcodage BCD → Excess3 sur 4 bits

On considère le transcodeur du code binaire naturel BCD sur 4 bits *DCBA* vers le code Excess3 sur 4 bits *dcba.*
On donne le codage Excess3 :

Decimal	Binary	Decimal	Binary
0	0011	9	1100
1	0100	8	1011
2	0101	7	1010
3	0110	6	1001
4	0111	5	1000

Le code Excess3 (*Excédent 3*) est un code issu du code BCD auquel on ajoute systématiquement 3 à chaque chiffre.
Le code Excess3 prend en charge uniquement le codage des chiffres (0 à 9) (l'entrée du transcodeur est toujours comprise entre 0 et 9).

1. Compléter les tableaux de Karnaugh décrivant les variables *d,c,b,a* du code Excess3. 1 *point*

d DC \ BA	00	01	11	10
00				
01				
11				
10				

c DC \ BA	00	01	11	10
00				
01				
11				
10				

b DC \ BA	00	01	11	10
00				
01				
11				
10				

a DC \ BA	00	01	11	10
00				
01				
11				
10				

2. Donner l'expression la plus simple de chacune de ces variables. 1 *point*

$d =$ ____ $c =$ ____ $b =$ ____ $a =$ ____

3. La conversion inverse se fait en ajoutant quelle valeur *positive* v exprimée en *décimal* (i.e. en base 10) au code Excess3 ? **0.5** *point*

Commentaires obligatoires :

$v =$ ____

2. Multiplexeur pour fonction logique

Soit la fonction logique x de 4 variables a_3, a_2, a_1, a_0 définie par sa table de vérité :

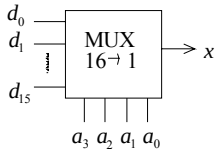
a_3	a_2	a_1	a_0	x
0	0	0	0	**0**
0	0	0	1	**0**
0	0	1	0	**1**
0	0	1	1	**1**

a_3	a_2	a_1	a_0	x
0	1	0	0	**0**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**0**

a_3	a_2	a_1	a_0	x
1	0	0	0	**1**
1	0	0	1	**1**
1	0	1	0	**0**
1	0	1	1	**0**

a_3	a_2	a_1	a_0	x
1	1	0	0	**1**
1	1	0	1	**1**
1	1	1	0	**0**
1	1	1	1	**0**

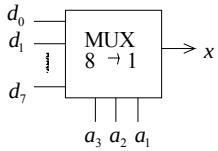
1. On souhaite réaliser la fonction logique x avec un multiplexeur `16 → 1` : (a_0, d_0 : LSB: bits de plus faible poids)



Compléter la table suivante décrivant les données d_0 à d_{15} du multiplexeur `16 → 1` : **0.5** *point*

d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}	d_{15}

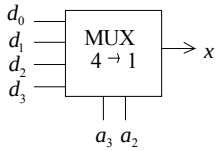
2. On souhaite réaliser la fonction logique x avec un multiplexeur `8 → 1` : (a_1, d_0 : LSB: bits de plus faible poids)



Compléter la table suivante décrivant les données d_0 à d_7 les plus simples du multiplexeur `8 → 1` : **0.5** *point*

d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7

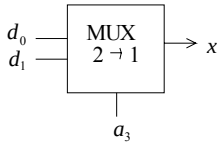
3. On souhaite réaliser la fonction logique x avec un multiplexeur `4 → 1` : (a_2, d_0 : LSB: bits de plus faible poids)



Compléter la table suivante décrivant les données d_0 à d_3 les plus simples du multiplexeur `4 → 1` : **0.5** *point*

d_0	d_1	d_2	d_3

4. On souhaite réaliser la fonction logique x avec un multiplexeur `2 → 1` : (d_0 : LSB: bits de plus faible poids)

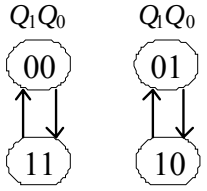


Compléter la table suivante décrivant les données d_0 à d_1 les plus simples du multiplexeur `2 → 1` : **0.5** *point*

d_0	d_1

3. Synthèse de Compteur synchrone 2 bits à bascules JK

Synthétiser le compteur synchrone 2 bits avec des bascules JK *positive edge triggered* qui compte dans la séquence :



en complétant les tables suivantes et le schéma ci-dessous, sous la contrainte d'obtenir le circuit le plus simple possible :

J_0	Q_0 = 0	Q_0 = 1
Q_1 = 0		
Q_1 = 1		

K_0	Q_0 = 0	Q_0 = 1
Q_1 = 0		
Q_1 = 1		

J_1	Q_0 = 0	Q_0 = 1
Q_1 = 0		
Q_1 = 1		

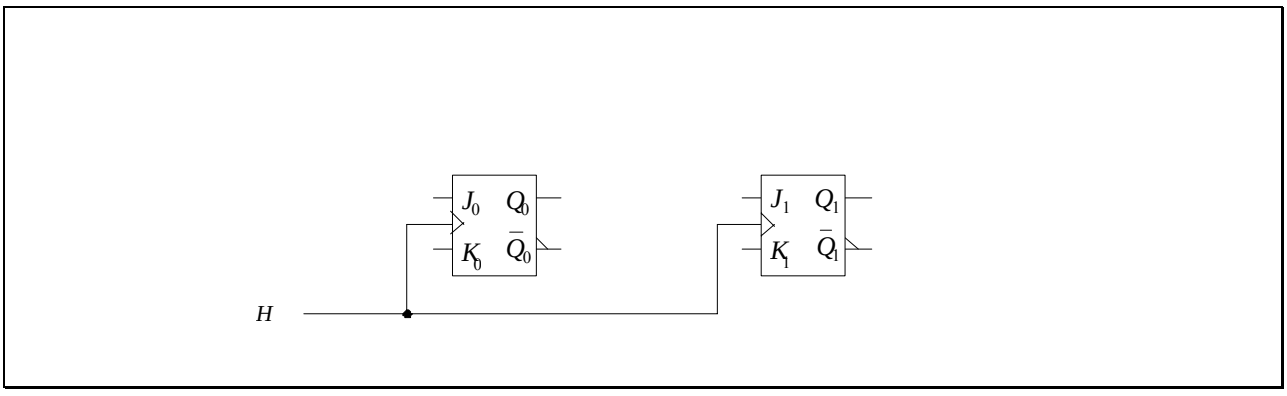
K_1	Q_0 = 0	Q_0 = 1
Q_1 = 0		
Q_1 = 1		

1 *point*

$$\begin{cases} J_0 = \\ K_0 = \end{cases}$$

$$\begin{cases} J_1 = \\ K_1 = \end{cases}$$

1 *point*



4. Compteur programmable

On considère un compteur binaire 4 bits synchrone, désigné par CNT. Il dispose de 2 commandes synchrones :

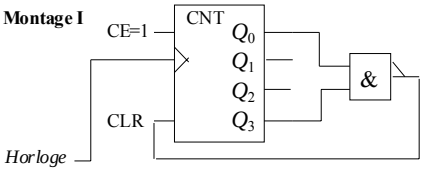
. CE (Count Enable) : Autorisation de comptage
. CLR (Clear) : Reset (Mise à 0) du Compteur

La table de fonctionnement du compteur CNT est la suivante :

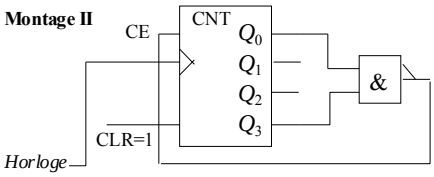
CE	CLR	*Fonction du compteur CNT*
1	1	Comptage (incrémentation de 1 du compteur) à chaque front montant d'horloge
0	1	Mémorisation
X	0	Reset (Mise à 0 du compteur) au front montant d'horloge suivant

Le compteur est initialisé à 0 : $Q_3Q_2Q_1Q_0 = 0000_2 = 0_{10}$. Q_3 est le MSB (bit de plus fort poids). On donne les 4 montages :

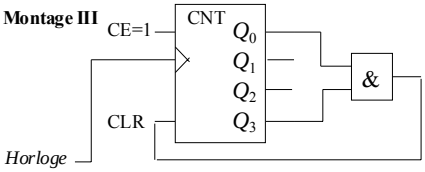
Montage I



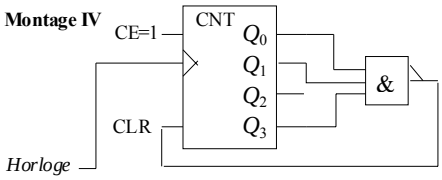
Montage II



Montage III



Montage IV



Pour chacun des montages, compléter la table suivante décrivant les états $S = Q_3Q_2Q_1Q_0$ du compteur exprimés en *décimal* (i.e. en base 10) en fonction de l'horloge :

Montage I : 1 *point*

Front montant d'horloge	*Init*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Etat S (en décimal)	**0**															

Commentaires obligatoires :

Montage II : 1 *point*

Front montant d'horloge	*Init*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Etat S (en décimal)	**0**															

Commentaires obligatoires :

Montage III : 1 *point*

Front montant d'horloge	*Init*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Etat S (en décimal)	**0**															

Commentaires obligatoires :

Montage IV : 1 *point*

Front montant d'horloge	*Init*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Etat S (en décimal)	**0**															

Commentaires obligatoires :

5. Représentation des nombres en machine : somme et produit en flottant

1. Donner (*en Hexadécimal*) le codage IEEE754 simple précision des nombres flottants 5 et 6.

5 = (*HEXA*)

0.5 *point*

6 = (*HEXA*)

Commentaires obligatoires :

2. Donner (*en Hexadécimal*) le codage IEEE754 simple précision de la somme 5 + 6 en flottant. Lors du calcul, y-a-t-il normalisation ?

5 + 6 = (*HEXA*) **1** *point* Normalisation : *OUI / NON*

Commentaires obligatoires :

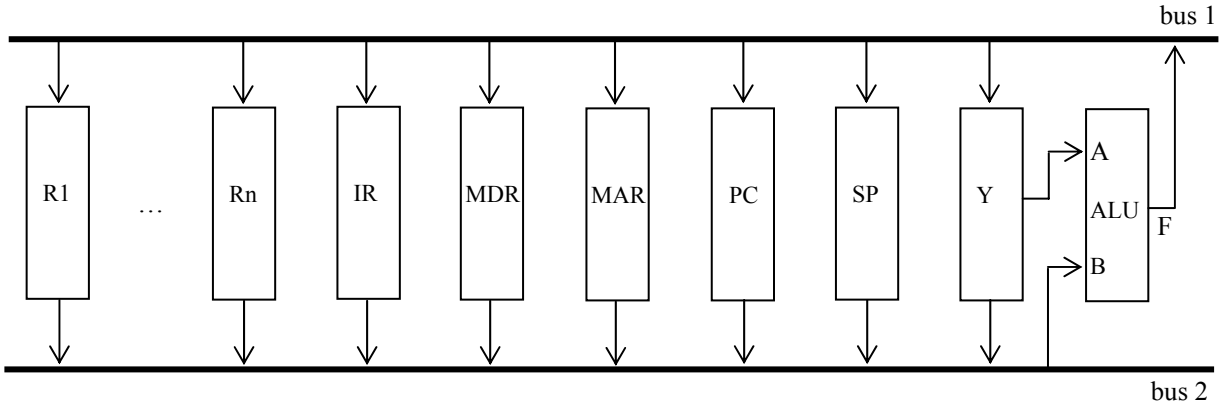
3. Donner (*en Hexadécimal*) le codage IEEE754 simple précision de la multiplication 5 x 6 en flottant. Lors du calcul, y-a-t-il normalisation ?

5x6 = (*HEXA*) **1** *point* Normalisation : *OUI / NON*

Commentaires obligatoires :

6. Machine de Von Neumann : chemin des données

On considère une organisation de CPU à 2 bus (bus 1 et bus 2 à la fois bus de données/adresses) représentée par la figure suivante :



Registres et ALU :

- R1 à Rn : registres généraux
- IR : registre d'instruction
- MDR : registre de donnée d'échange avec la mémoire
- MAR : registre d'adresse d'échange avec la mémoire
- PC : compteur ordinal
- SP : pointeur de pile
- Y : registre pour mémoriser l'entrée A de l'ALU
- ALU : Unité Arithmétique et Logique

Micro-instructions (étapes possibles du traitement d'une instruction machine) :

- Reg out : sort le contenu du registre Reg sur le bus 2
- Reg in : met la valeur du bus 1 dans le registre Reg
- Lecture : effectue une lecture de la mémoire à l'adresse MAR et met le résultat dans MDR
- Ecriture : effectue une écriture dans la mémoire à l'adresse MAR de la valeur contenue dans MDR
- ADD : additionne les entrées A et B de l'ALU, met le résultat sur la sortie de l'ALU (F = A+B)
- INCRA : incrémente l'entrée A de l'ALU (F = A+1)
- DECRA : décrémente l'entrée B de l'ALU (F = A-1)
- INCRB : incrémente l'entrée B de l'ALU (F = B+1)
- DECRB : décrémente l'entrée B de l'ALU (F = B-1)
- REPA : reporte l'entrée A de l'ALU sur sa sortie (F = A)
- REPB : reporte l'entrée B de l'ALU sur sa sortie (F = B)

Pour l'ALU, l'entrée A est directement lue sur la sortie du registre Y (aucune opération à effectuer par rapport au registre Y) ; l'entrée B est lue directement sur le bus 2 ; la sortie F est inscrite directement sur le bus 1 quand une opération est déclenchée.
Pour simplifier, on considère une machine de Von Neumann manipulant des données sur 8 bits et des adresses sur 8 bits. On s'intéresse à l'exécution de l'instruction suivante : ADD R1 (R2) qui additionne le contenu du registre R1 et la donnée en mémoire dont l'adresse est contenue dans le registre R2. En fin de traitement de cette instruction, le résultat en sortie de l'ALU est stocké dans le registre accumulateur R1. L'addition se fait sur des entiers relatifs codés en complément à 2. L'instruction entière ADD R1 (R2) est codée par le mot 0xFA en mémoire, et est située à l'adresse 0x03 en mémoire. Les registres R1 et R2 ont été préalablement chargés par le programme avec les valeurs suivantes : R1 contient 0xDD et R2 0xAB. Voici un extrait du contenu de la mémoire :

(le préfixe 0x indique une valeur exprimée en *Hexadécimal*)

0x00	
0x01	
0x02	
0x03	0xFA
0x04	…
…	
0xAB	0x27
…	…
0xFF	

On s'intéresse à l'évolution des données dans cette machine lors du cycle de Chargement-Décodage-Exécution (*Fetch-Decode-Execute*) de cette instruction (la partie décodage de l'instruction n'est pas ici traitée, elle est supposée réalisée à la ligne 10 du tableau ci-dessous). On donne le début du micro-programme qui commence par aller chercher l'opérande (R2) avant d'aller ensuite chercher l'opérande R1 pour enfin exécuter l'instruction d'addition.

Compléter le tableau suivant à partir de la ligne 12 et jusqu'à l'écriture du résultat de l'addition dans le registre R1 en ne remplissant dans les cases que les **changements de valeurs** comme cela est fait pour les lignes précédentes. Les premières lignes déjà remplies correspondent au début de l'exécution du micro-programme : **3** *points*

R1	**R2**	**IR**	**MDR**	**MAR**	**PC**	**Y**	**bus 1**	**Bus 2**	**micro-instruction**	**Commentaire**	
0xDD	0xAB	indéfini	indéfini	indéfini	0x03	indéfini	indéfini	Indéfini	indéfini	Etat initial	1
								0x03	PC out	(PC) → bus 2	2
							0x03		REPB	F = (PC) = 0x03	3
				0x03					MAR in	MAR = 0x03	4
							0x04		INCRB	F = 0x04	5
					0x04				PC in	PC = 0x04	6
			0xFA						Lecture	0xFA → MDR	7
								0xFA	MDR out	B = 0xFA	8
							0xFA		REPB	F = 0xFA	9
		0xFA							IR in	0xFA → IR; + Décodage	10
								0xAB	R2 out	B = 0xAB	11
											12
											13
											14
											15
											16
											17
											18
											19
											20

Vérification du résultat obtenu : le résultat du calcul de l'addition est-il correct ? *OUI / NON* **0.5** *point*

Commentaires obligatoires :

7. Pipelining (1) : Processeur à 8 étages

On considère un processeur avec un pipeline à 8 étages possédant un temps de cycle de $0.5ns$. Les étages 6 et 7 du pipeline sont consacrés à l'accès (*lecture/écriture*) à la mémoire cache de données.

1. Quel est le nombre maximal d'instructions qui peuvent s'exécuter *simultanément* dans ce processeur ?

	instructions simultanées

0.5 *point*

2. Quelle est la cadence maximale d'exécution d'instructions (*en nombre d'instructions/seconde*) pour ce processeur ?

	instructions / seconde

0.5 *point*

3. Quelle doit être la valeur maximale du temps d'accès au cache en cas de succès (*cache-hits*) ?

0.5 *point*

8. Pipelining (2) : Architectures Pipelines P1 et P2

On considère les pipelines P1 et P2 :

Pipeline P1

Soit le pipeline d'instructions (**P1**) représenté ci-dessous, ayant un débit théorique maximum égal à 1 instruction/cycle :

Un cycle est le temps nécessaire au traitement d'un étage du pipeline.

	étage	*pipeline P1*		
	1	**lit cache d'instructions**		
	2	**decode instruction**		
	3	**lit registres**		
instruction ≠ LOAD/STORE →	**4**	**Execute / calcul adresse**		←)
	5	**Acces cache de donnees**	)	←) LOAD/STORE
	6	**Acces cache de donnees**	) 2 cycles	←)
	7	**ecrit registre**		

On suppose que toutes les instructions, sauf les instructions LOAD et STORE, sont exécutées à l'étage 4. Pour les instructions LOAD/STORE, le calcul d'adresse du/des opérandes du LOAD/STORE se fait à l'étage 4 et l'accès au cache de données est pipeliné sur 2 cycles (étages 5 et 6). Pour les instructions qui ne sont pas des LOAD/STORE, les étages 5 et 6 se comportent comme des étages vides.

On suppose qu'il y a un mécanisme de *bypass* permettant de transmettre le résultat d'une instruction aux instructions suivantes sans attendre l'écriture registre. Si un opérande source d'une instruction n'est pas disponible au moment où celle-ci va rentrer dans l'étage 4, les étages 1 à 3 sont bloqués jusqu'à ce que l'opérande indisponible soit accessible via le mécanisme de *bypass*, ce qui conduit à l'insertion de *bulles* dans le pipeline.

2 bulles sont insérées dans le pipeline P1 si l'instruction immédiatement après un LOAD/STORE utilise comme opérande le résultat du LOAD/STORE.

On supposera un prédicteur de branchement parfait. On supposera également que toutes les lectures d'instructions font des *hits* dans le cache d'instructions et que tous les LOAD/STORE font des *hits* dans le cache de données (cela signifie qu'il n'y a que des *cache-hits* et pas de *cache-miss*, c'est-à-dire que les accès cache sont toujours tels que le cache respectivement d'instructions / données contient les cibles, instructions / données).

Pipeline P2

On considère une version modifiée du pipeline d'instructions **P1** comme représenté ci-dessous (pipeline **P2**), c'est-à-dire en déplaçant l'étage d'exécution de l'étage 4 à l'étage 6. Le calcul d'adresse continue à se faire à l'étage 4 :

étage	*pipeline P2*
1	**lit cache d'instructions**
2	**decode instruction**
3	**lit registres**
4	**calcul adresse**
5	**Acces cache de donnees**
6	**execute / acces cache de donnees**
7	**ecrit registre**

2 bulles sont insérées dans le pipeline P2 si l'instruction immédiatement avant un LOAD/STORE met à jour un opérande du LOAD/STORE.

I. Pipeline P1

On s'intéresse dans cette 1ère partie au pipeline **P1**.

On considère le programme ci-dessous écrit dans un assembleur donné, qui calcule la somme des N éléments d'un tableau contenant des entiers. Le registre R4 est initialisé avec N, le registre R1 est initialisé avec l'adresse A du 1er élément du tableau et le registre R3 contenant en fin d'exécution le résultat de la somme est initialisé à 0. Les éléments du tableau sont stockés sur 2 octets. Enfin, le calcul de la somme est interrompu si celle-ci égale une valeur seuil donnée. Le registre R6 est initialisé avec ce seuil :

```
1:      boucle:  R2 = LOAD R1+0          // lit element du tableau
2:               R4 = R4 SUB 1           // R4 = R4 - 1
3:               R3 = R3 ADD R2          // ajoute a la somme
4:               R7 = R3 SUB R6          // R7 = R3 - R6
5:               BZ R7, fin              // BZ (Branch Zero) : saut à l'étiquette fin si R7 = 0
6:               R1 = R1 ADD 2           // R1 = R1 + 2
7:               BNZ R4, boucle          // BNZ (Branch Not Zero) : boucle si R4 ≠ 0
8:      fin:     …                       // fin de traitement
```

1. En supposant N grand et en partant d'une configuration où le calcul de la somme n'est pas interrompu, quel est le débit d'exécution de la boucle de programme en *instructions par cycle* ?

	instructions / cycle	**0.5** *point*

Commentaires obligatoires :

2. On modifie ainsi le programme précédent, le registre R5 étant initialisé à 0 :

```
1:      boucle:  R2 = LOAD R1+0        // lit element du tableau
2:               R3 = R3 ADD R2        // ajoute a la somme
3:               R5 = R5 ADD R2        // recalcul et sauvegarde du resultat dans R5
4:               R7 = R3 SUB R6        // R7 = R3 - R6
5:               BZ R7, fin            // BZ (Branch Zero) : saut à l'étiquette fin si R7 = 0
6:               R1 = R1 ADD 2         // R1 = R1 + 2
7:               R4 = R4 SUB 1         // R4 = R4 - 1
8:               BNZ R4, boucle        // BNZ (Branch Not Zero) : boucle si R4 ≠ 0
9:      fin:     …                     // fin de traitement
```

Dans les mêmes hypothèses que précédemment, indiquer le débit d'exécution de la boucle de programme en *instructions par cycle* ?

	instructions / cycle	**0.5** *point*

Commentaires obligatoires :

II. Pipeline P2

On s'intéresse dans cette 2nde partie au pipeline **P2**.

Mêmes questions pour le pipeline **P2** que pour le pipeline **P1** :

1.

	instructions / cycle	**0.5** *point*

Commentaires obligatoires :

2.

	instructions / cycle	**0.5** *point*

Commentaires obligatoires :

EXAMEN D'ELECTRONIQUE INTEGREE - CORRIGE

1. Transcodage BCD → Excess3 sur 4 bits

	Code BCD	Code Excess3
N	*DCBA*	*dcba*
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100
10	1010	XXXX
11	1011	XXXX
12	1100	XXXX
13	1101	XXXX
14	1110	XXXX
15	1111	XXXX

1.

d (DC \ BA)	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

c (DC \ BA)	00	01	11	10
00	0	1	1	1
01	1	0	0	0
11	X	X	X	X
10	0	1	X	X

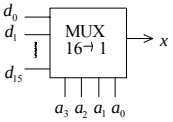
b (DC \ BA)	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	X	X	X	X
10	1	0	X	X

a (DC \ BA)	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	X	X	X	X
10	1	0	X	X

2. $\boxed{d = D + CA + CB = D + C(A + B)}$ $\boxed{c = \overline{C}A + C\overline{B}\,\overline{A} + \overline{C}B = \overline{C}(A + B) + C\overline{B}\,\overline{A}}$ $\boxed{b = \overline{B}\,\overline{A} + BA = \overline{A \oplus B}}$ $\boxed{a = \overline{A}}$

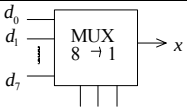
3. La conversion en sens inverse se fait en soustrayant 3 au code Excess3 ou bien en ajoutant le nombre binaire $\boxed{v = 13}$ (= 1101_2) : 0011 (Excess3) → 0000 (BCD) donc 0011 + **1101** = 0000.

2. Multiplexeur pour fonction logique



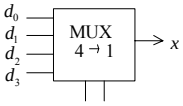
1. MUX 16 →1: $x = \overline{a}_3\overline{a}_2a_1\overline{a}_0 + \overline{a}_3\overline{a}_2a_1a_0 + a_3\overline{a}_2\overline{a}_1\overline{a}_0 + a_3\overline{a}_2\overline{a}_1a_0 + a_3a_2\overline{a}_1\overline{a}_0 + a_3a_2\overline{a}_1a_0$

d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}	d_{15}
0	0	1	1	0	0	0	0	1	1	0	0	1	1	0	0



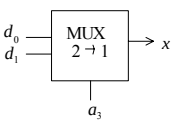
2. MUX 8 →1: $x = \overline{a}_3\overline{a}_2a_1(\overline{a}_0 + a_0) + a_3\overline{a}_2\overline{a}_1(\overline{a}_0 + a_0) + a_3a_2\overline{a}_1(\overline{a}_0 + a_0) = \overline{a}_3\overline{a}_2a_1(1) + a_3\overline{a}_2\overline{a}_1(1) + a_3a_2\overline{a}_1(1)$

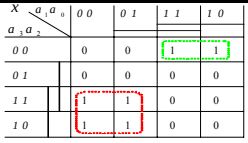
d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
0	1	0	0	1	0	1	0



3. MUX 4 →1: $x = \overline{a}_3\overline{a}_2(a_1\overline{a}_0 + a_1a_0) + a_3\overline{a}_2(\overline{a}_1\overline{a}_0 + \overline{a}_1a_0) + a_3a_2(\overline{a}_1\overline{a}_0 + \overline{a}_1a_0) = \overline{a}_3\overline{a}_2(a_1) + a_3\overline{a}_2(\overline{a}_1) + a_3a_2(\overline{a}_1)$

d_0	d_1	d_2	d_3
a_1	0	$\overline{a}_1$	$\overline{a}_1$

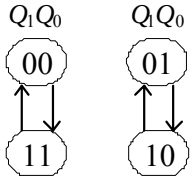




4. MUX 2 →1: $x = \overline{a}_3\overline{a}_2a_1 + a_3\overline{a}_1 = \overline{a}_3(\overline{a}_2a_1) + a_3(\overline{a}_1)$

d_0	d_1
$\overline{a}_2a_1$	$\overline{a}_1$

3. Synthèse de Compteur synchrone 2 bits à bascules JK



$J_0\ K_0$

Q_1 \ Q_0	0	1
0	1 X	X 1
1	1 X	X 1

$J_1\ K_1$

Q_1 \ Q_0	0	1
0	1 X	1 X
1	X 1	X 1

Table des transitions (Synthèse) - (Horloge active)

Transition $Q_{n-1} \rightarrow Q_n$	***J***	***K***
$0 \rightarrow 0$	0	X
$0 \rightarrow 1$	1	X
$1 \rightarrow 1$	X	0
$1 \rightarrow 0$	X	1

J_0

Q_1 \ Q_0	0	1
0	1	X
1	1	X

K_0

Q_1 \ Q_0	0	1
0	X	1
1	X	1

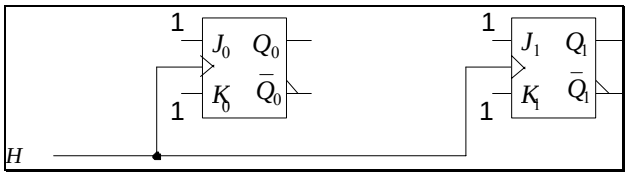
J_1

Q_1 \ Q_0	0	1
0	1	1
1	X	X

K_1

Q_1 \ Q_0	0	1
0	X	X
1	1	1

$$\begin{cases} J_0 = 1 \\ K_0 = 1 \end{cases} \qquad \begin{cases} J_1 = 1 \\ K_1 = 1 \end{cases}$$



4. Compteur programmable

N	$Q_3Q_2Q_1Q_0$
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Montage I : CLR = $\overline{Q_0Q_3}$ CE = 1 $\overline{Q_0Q_3} = 0$ pour $N = 9,11,13,15$

Front montant d'horloge	*Init*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Etat S (en décimal)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**

Montage II : CE = $\overline{Q_0Q_3}$ CLR = 1 $\overline{Q_0Q_3} = 0$ pour $N = 9,11,13,15$ **Attention au décalage dû au temps de propagation**

Front montant d'horloge	*Init*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Etat S (en décimal)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**9**	**9**	**9**	**9**	**9**	**9**

Montage III : CE = 1 CLR = Q_0Q_3 $Q_0Q_3 = 0$ pour $N \neq 9,11,13,15$ **Attention au décalage dû au temps de propagation**

Front montant d'horloge	*Init*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Etat S (en décimal)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Montage IV : CE = 1 CLR = $\overline{Q_0Q_1Q_3}$ $\overline{Q_0Q_1Q_3} = 0$ pour $N = 11,15$

Front montant d'horloge	*Init*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Etat S (en décimal)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**0**	**1**	**2**	**3**

5. Représentation des nombres en machine : somme et produit en flottant

1.

X = 5 = $2^k \cdot (1, \cdots)$ = 4 x 1,25 = 2^2 x (1 + 0,**25**) = $2^{129-127}$ x (1 + 0,**25**)

s (1) = **0**

e (8) = 129 = **1000 0001**

m (23) = 0,25 = **010 0000 0000 0000 0000 0000** = m_X

sem = **0100 0000 1010 0000 0000 0000 0000 0000 = 40 A0 00 00**$_{(H)}$

$5 = 40\ A0\ 00\ 00_{(HEXA)}$

Y = 6 = $2^k \cdot (1, \cdots)$ = 4 x 1,5 = 2^2 x (1 + 0,**5**) = $2^{129-127}$ x (1 + 0,**5**)

s (1) = **0**

e (8) = 129 = **1000 0001**

m (23) = 0,5 = **100 0000 0000 0000 0000 0000** = m_Y

sem = **0100 0000 1100 0000 0000 0000 0000 0000 = 40 C0 00 00**$_{(H)}$

$6 = 40\ C0\ 00\ 00_{(HEXA)}$

2. X +Y = 5 + 6 =

1,010 0000 0000 0000 0000 0000 * $2^{129-127}$

\+ 1,100 0000 0000 0000 0000 0000 * $2^{129-127}$

10,110 0000 0000 0000 0000 0000 * $2^{129-127}$

= 1,011 0000 0000 0000 0000 0000 * $2^{130-127}$ Normalisation

soit pour le résultat : **5 + 6** =

s (1) = **0**

e (8) = 130 = **1000 0010**

m (23) = **011 0000 0000 0000 0000 0000**

sem = **0100 0001 0011 0000 0000 0000 0000 0000 = 41 30 00 00**$_{(H)}$

$5 + 6 = 41\ 30\ 00\ 00_{(HEXA)}$ Normalisation : *OUI*

Vérification : A comparer au résultat **11** :

11 = $2^k \cdot (1, \cdots)$ = 8 x 1,375 = 2^3 x (1 + 0,**375**) = $2^{130-127}$ x (1 + 0,**375**)

s (1) = **0**

e (8) = 130 = **1000 0010**

m (23) = 0,375 = **011 0000 0000 0000 0000 0000**

sem = **0100 0001 0011 0000 0000 0000 0000 0000 = 41 30 00 00**$_{(H)}$

même résultat que 5 + 6) **OK**

3. Z = X x **Y = 5** x 6 =

signe : **0** (xor entre les 2 signes des 2 opérandes)

exposant : **1000 0001 + 1000 0001 - (127)$_2$ = 1000 0001 + 1000 0001 − 0111 1111**
= 1000 0001 + 1000 0001 + (-127)$_{C2}$ = 1000 0001 + 1000 0001 + 1000 0001
= 1000 0011 (= 131$_{10}$) = 4 + 127 (**OK : 4 = 2 + 2**)

mantisse : Multiplication des mantisses m_X et m_Y :

rq: $(1 + m_X)(1 + m_Y) = 1 + m_X.m_Y + m_X + m_Y \rightarrow \mathbf{m_Z = m_X.m_Y + m_X + m_Y}$

$1,\mathbf{m_X}$ = 1,**01** 0…0

x $1,\mathbf{m_Y}$ = x 1,**1** 0…0

101

101

$1,\mathbf{m_Z}$ = 1,**111** 0…0 Pas de Normalisation à faire

→ $\mathbf{m_Z}$ = **111** 0…0 → $\mathbf{m_Z}$ **(23) = 111 0000 0000 0000 0000 0000**

Interprétation du résultat : **Z**

s (1) : **0**

e (8) : **1000 0011** → **e = 131** → **exposant = 131-127 = 4** % 127

m (23): **111 0000 0000 0000 0000 0000** → **mantisse =** $2^{-1} + 2^{-2} + 2^{-3} =$ 0,875

sem = 0100 0001 1111 0000 0000 0000 0000 0000 = 41 F0 00 00$_{(H)}$

Z = $1{,}875 \times 2^4 = 1{,}875 \times 16 =$ **30**

$5x6 = 41\,F0\,00\,00_{(HEXA)}$ Normalisation : *NON*

Vérification du résultat : **Z**

Z (exact) = 30 = $2^k \cdot (1, \cdots) = 16 \times 1{,}875 = 2^4 \times (1 + 0{,}875) = 2^{131-127} \times (1 + 0{,}875)$

s (1) = **0**

e (8) = 131 = **1000 0011 = 4** % 127

m (23) = 0,875 = $2^{-1} + 2^{-2} + 2^{-3}$ = **111 0000 0000 0000 0000 0000**

sem = **0100 0001 1111 0000 0000 0000 0000 0000 = 41 F0 00 00**$_{(H)}$

6. Machine de Von Neumann : chemin des données

R1	R2	IR	MDR	MAR	PC	Y	bus 1	bus 2	micro-instruction	Commentaire	
0xDD	0xAB	indéfini	indéfini	indéfini	0x03	indéfini	indéfini	indéfini	indéfini	Etat initial	1
								0x03	PC out	(PC) → bus 2	2
							0x03		REPB	F = (PC) = 0x03	3
				0x03					MAR in	MAR = 0x03	4
							0x04		INCRB	F = 0x04	5
					0x04				PC in	PC = 0x04	6
			0xFA						Lecture	0xFA → MDR	7
								0xFA	MDR out	B = 0xFA	8
							0xFA		REPB	F = 0xFA	9
		0xFA							IR in	0xFA → IR; + Décodage	10
								0xAB	R2 out	B = 0xAB	11
							0xAB		**REPB**	**F = 0xAB**	**12**
				0xAB					**MAR in**	**MAR = 0xAB**	**13**
			0x27						**Lecture**	**MDR = 0x27**	**14**
								0x27	**MDR out**	**B = 0x27**	**15**
							0x27		**REPB**	**F = 0x27**	**16**
						0x27			**Y in**	**A = 0x27**	**17**
								0xDD	**R1 out**	**B = 0xDD**	**18**
							0x04		**ADD**	**F = 0x27 + 0xDD = 0x04**	**19**
0x04									**R1 in**	**R1 = 0x04**	**20**

Vérification du résultat obtenu : le résultat du calcul de l'addition est-il correct ? *OUI*

```
Addition  : 11111111    (retenues)
   0xDD   :  11011101  -> (- 11011100 = 00100011 = 35) -> -35
+  0x27   :  00100111  -> 39
=  0x0    :  00000100  -> 4
```

7. Pipelining (1) : Processeur à 8 étages

1. Jusqu'à 8 *instructions simultanées*.

2. Jusqu'à 1 *instruction* toutes les $0.5\,ns$, soit $2 \cdot 10^9 = 2\,milliards$ *instructions/seconde*.

3. $0.5\,ns \times 2 = 1\,ns$ puisque l'accès au cache se fait sur 2 étapes.

8. Pipelining (2) : Architectures Pipelines P1 et P2

I. Pipeline P1

1. En négligeant le temps de remplissage du pipeline, lorsqu'un LOAD/STORE est immédiatement suivi d'une instruction utilisant le résultat du LOAD/STORE, 2 bulles sont insérées dans le pipeline.

Dans le programme considéré, 1 bulle est donc introduite du fait de l'instruction LOAD car 1 instruction indépendante du résultat du LOAD est intercalée entre le LOAD et l'instruction 3 dépendante du résultat du LOAD (1 bulle sur les 2 insérées par le LOAD a pu être éliminée).

Le corps de la boucle comporte 7 instructions, le débit d'exécution vaut : $\frac{7}{7+1} = \frac{7}{8} = 0.875$ *instructions / cycle*.

2. L'instruction 1 (LOAD) est suivie immédiatement d'une instruction utilisant le résultat du LOAD, 2 bulles sont insérées dans le pipeline. L'instruction 3 n'introduit pas de bulles supplémentaires car les 2 bulles précédentes sont éliminées après l'instruction 2.

En négligeant le temps de remplissage du pipeline : le corps de la boucle comporte 8 instructions, 2 bulles sont donc générées toutes les 8 instructions.

Le débit d'exécution vaut : $\frac{8}{8+2} = \frac{8}{10} = \frac{4}{5} = 0.8$ *instructions / cycle*.

II. Pipeline P2

Rappel :

étape	*pipeline P1*
1	**Lit cache d'instructions**
2	**decode instruction**
3	**Lit registres**
4	**execute / calcul adresse**
5	**acces cache de donnees**
6	**acces cache de donnees**
7	**ecrit registre**

étape	*pipeline P2*
1	**lit cache d'instructions**
2	**decode instruction**
3	**lit registres**
4	**calcul adresse**
5	**Acces cache de donnees**
6	**execute / acces cache de donnees**
7	**ecrit registre**

. Pipeline initial (P1) : 2 *bulles* sont introduites si le résultat de l'exécution d'une instruction LOAD/STORE dépend de l'instruction précédente.

. Nouveau Pipeline (P2) :2 *bulles* sont introduites avec le calcul d'adresse pour chaque opérande d'une instruction LOAD/STORE dépendant de l'instruction précédente.

Si les 2 instructions précédant un LOAD/STORE sont indépendantes du calcul d'adresse de ou des opérandes de l'instruction LOAD/STORE, aucune *bulle* n'est introduite.

1. L'instruction 6 avant le LOAD met à jour l'opérande R1 du LOAD et 1 instruction indépendante du LOAD est intercalée, 1 bulle est donc générée.
En négligeant le temps de remplissage du pipeline : le corps de la boucle comporte 7 instructions. 1 bulle est donc générées toutes les 7 instructions.

Le débit d'exécution vaut : $\frac{7}{7+1} = \frac{7}{8} = 0.875$ *instructions / cycle*.

2. Aucune instruction parmi les 2 instructions précédant le LOAD (instructions 7 et 8) ne met à jour l'opérande R1 du LOAD, aucune bulle n'est donc générée.

En négligeant le temps de remplissage du pipeline : le corps de la boucle comporte 8 instructions, 0 bulle est donc générée toutes les 8 instructions.

Le débit d'exécution vaut : $\frac{8}{8+0} = \frac{8}{8} = 1$ *instructions / cycle*.