Architecture Matérielle et Logicielle (LIF6)

Cahier d'exercices, printemps 2016

Table des matières

1 Vue d'ensemble de l'ordinateur **4**
1.1 Petits calculs autour d'une carte mère 4
1.1.1 Bande passante d'un bus 4
1.1.2 Lecture d'un film 5
1.2 Langage machine 5
1.3 Instructions de branchement et boucles 6
1.4 Codage des instructions 7
1.4.1 Codage des instructions 8
1.5 Taille du bus, volume de mémoire centrale 8
1.6 Langage machine 8

2 Codage des nombres **11**
2.1 Représentation positionnelle des entiers naturels 11
2.2 Représentation en complément à deux sur p bits 11
2.2.1 Définition 11
2.2.2 Addition et calcul de l'opposé 11
2.2.3 Extension de signe en complément à 2 12
2.3 Représentation positionnelle des rationnels 12
2.4 Représentation à virgule flottante, introduction 12
2.4.1 Définition 12
2.4.2 Représentation en machine, norme IEEE-754 : 13
2.4.3 Arrondi correct 13
2.5 Représentation à virgule flottante 13
2.6 Codage de nombres en machine 14
2.7 Conversion base 10 vers base 2 d'entiers naturels 14
2.8 Représentation à virgule flottante 14
2.8.1 14
2.8.2 15

3 Circuits combinatoires **16**
3.1 Addition-soustraction en complément à 2 sur n bits 16
3.1.1 Dépassement de capacité 16
3.1.2 Additionneur-soustracteur 17
3.2 Comparaison d'entiers naturels 17
3.3 Décalage 18
3.4 Générateur de parité impaire 19
3.5 Des circuits combinatoires 19
3.5.1 Analyse d'un circuit combinatoire 19
3.5.2 Décodeurs 19
3.6 Circuit décrémenteur 20
3.7 Petits circuits combinatoires 20
3.8 Encodeur octal 20

4 Circuits séquentiels **21**
4.1 Un générateur de séquence simple 21
4.2 Détection de séquences spécifiques 22
4.2.1 Avec un automate séquentiel classique 22
4.2.2 Avec des registres à décalage 22
4.3 Circuit compteur 23
4.4 Génération d'un signal périodique 24
4.5 Un contrôleur 24
4.6 Addition séquentielle 24

4.7 Analyse d'un circuit séquentiel 25
4.8 Génération d'un circuit séquentiel 26

5 Programmation en assembleur 28
5.1 Autour de la sommation 30
5.2 Ajouter 1 aux éléments d'un tableau 30
5.3 Programme Mystère LC3 31
5.4 Dessin de carrés et de triangles 31
5.5 Multiplication par 6 des entiers d'un tableau 32
5.6 Décompte de bits non-nuls 33
5.7 Palindromes 33
5.8 Affichage et Récursivité 34
5.9 Nombre de chiffres dans une chaîne de caractères 35
5.10 Renversement d'une chaîne de caractères grâce à la pile 35

6 Mémoire cache et pipeline 38
6.1 Mémoire cache 38
6.2 Somme de deux vecteurs 38
6.3 Somme des éléments d'une matrice 39
6.4 Mémoire cache - petits calculs 40
6.5 Calculs autour de l'exécution pipelinée 40
6.5.1 Calcul sur des pipelines 40
6.5.2 Calibrage d'un pipeline 41
6.6 Pipeline à 4 étages 41
6.7 Boucles et exécution pipelinée 41

Thème 1
Vue d'ensemble de l'ordinateur

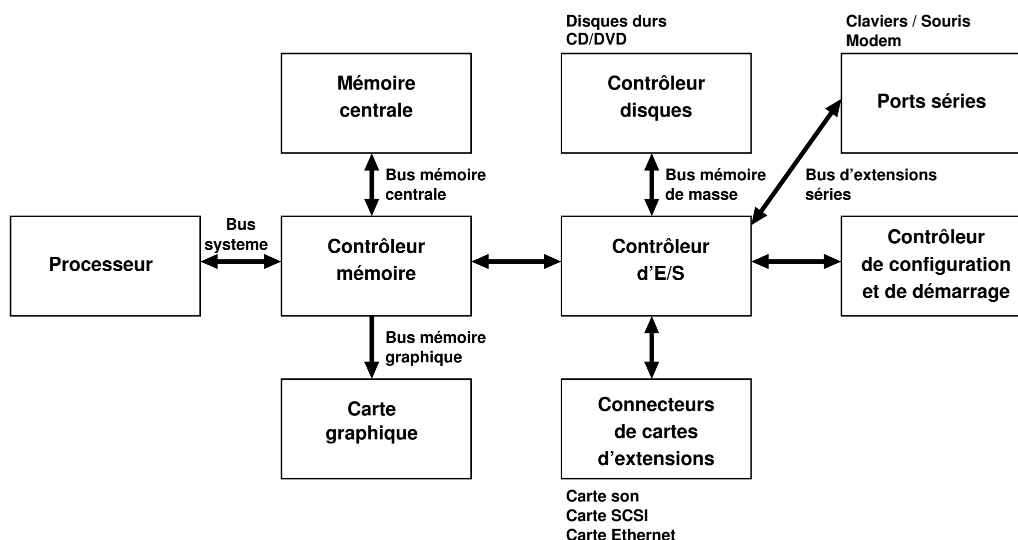
Exercice 1 : Petits calculs autour d'une carte mère

Les tailles des mémoires sont des puissances de deux ou des multiples de telles puissances. On utilise parfois les préfixes SI en leur associant la puissance de 2 la plus proche : par exemple 1024 octets est souvent abusivement noté 1 ko, bien que 1 ko = 1000 o... La norme CEI-60027-2 définie des *préfixes binaires* pour éviter la confusion :

préfixe standardisé	symbole standardisé	puissance de 2	puissance de 10 la plus proche
tébi	Ti	2^{40}	$10^{12} = \mathrm{T}$
gibi	Gi	2^{30}	$10^{9} = \mathrm{G}$
mébi	Mi	2^{20}	$10^{6} = \mathrm{M}$
kibi	ki	2^{10}	$10^{3} = \mathrm{k}$

Ainsi, 1024 octets = 1 kibi-octet = 1 kio. On utilisera ces préfixes binaires par la suite.

La *carte mère* d'un PC regroupe des fonctionnalités de base d'un ordinateur : processeur, mémoire centrale et gestion des E/S.



Partie 1 : Bande passante d'un bus

La bande passante d'un bus, aussi appelée débit crête, est la quantité de données pouvant circuler sur ce bus par unité de temps. Sur la carte mère d'un PC, le bus reliant le processeur au contrôleur mémoire est le bus système, souvent appelé *Front Side Bus* (FSB). Supposons ici que le FSB d'un certain ordinateur est capable d'assurer le transfert de 8 octets à la fréquence de 400 MHz.

Un contrôleur mémoire prend à sa charge les échanges entre le processeur et la mémoire centrale, entre le processeur et le contrôleur d'E/S, et entre le processeur et la mémoire vidéo. Le contrôleur mémoire peut en outre mettre en place un accès direct entre le contrôleur d'E/S et la mémoire centrale ou la mémoire vidéo (accès DMA pour *Direct Memory Access*) : le contrôleur d'E/S pourra par exemple transférer directement des données d'un périphérique à la mémoire vidéo sans qu'elles transitent par le processeur.

1) Quelle est la bande passante du bus FSB considéré exprimée en Go/s ? Rappelons que 1 Go = 10^9 o.

2) Quelle est la bande passante du bus FSB considérée exprimée en Gio/s ? Rappelons que 1 Gio = 2^{30} o.

3) Supposons que le bus mémoire centrale permet le transfert de mots de 32 bits à la fréquence de 266 MHz. Quelle est la bande passante du bus mémoire centrale en Go/s ?

4) Que penser de la différence de bande passante entre le bus de la mémoire centrale et celle du FSB ?

Partie 2 : Lecture d'un film

Un film est lu à partir d'un disque dur, connecté via le bus IDE au contrôleur de disque. Le film est non-compressé, et constitué d'une succession d'images de 512 × 384 pixels en 256 couleurs. On suppose que le défilement des images se fait en 24 images par seconde.

1) Quels sont les bus utilisés pour le transfert ?

2) Quel est le débit (en Mo/s) requis pour le transfert du film du disque dur à la mémoire vidéo ?

3) Supposons que le bus de la mémoire vidéo a une bande passante identique à celle du bus de la mémoire centrale. Quelle est la part (en pourcentage) de la bande passante du bus de la mémoire vidéo est consommée par la lecture du film ?

Exercice 2 : Langage machine

On se place sur un processeur hypothétique, qui accède à une mémoire centrale dans laquelle la taille d'une case mémoire est de 2 octets. Ce processeur dispose des registres suivants :

— **IR**, le registre d'instruction ;

— **PC**, le compteur de programme ;

— **A** (comme accumulateur), un registre temporaire pour le stockage du résultat des opérations.

Les instructions sont codées comme suit :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode				adresse											

On détaille les instructions suivantes :

mnémonique	opcode		opération réalisée
LOAD	$(0001)_2$	$(1)_H$	charger le mot dont l'adresse est donnée dans **A**
STORE	$(0010)_2$	$(2)_H$	stocker le mot contenu dans **A** à l'adresse donnée
ADD	$(0101)_2$	$(5)_H$	ajouter l'entier naturel à l'adresse donnée à **A**.

1) Combien d'instructions, suivant le codage indiqué ci-dessus, peut compter le jeu d'instruction ?

2) Quel est le nombre maximal d'adresses auxquelles une telle instruction peut faire référence ?

3) On considère le morceau de programme suivant, écrit en « langage d'assemblage » :

LOAD $(130)_H$
ADD $(131)_H$
ADD $(132)_H$
STORE $(133)_H$

Que fait ce programme ?

4) Traduisez ce programme en langage machine, et représentez le dans la mémoire centrale en plaçant la première instruction à l'adresse $(100)_H$.

5) On suppose que le contenu des cases mémoires $(130)_H$ à $(133)_H$ est initialement le suivant :

$(130)_H$	$(0002)_H$
$(131)_H$	$(0003)_H$
$(132)_H$	$(0001)_H$
$(133)_H$	$(0022)_H$

Représentez le contenu des cases mémoires $(130)_H$ à $(133)_H$, ainsi que celui des registres **PC**, **IR** et **A** après l'exécution de chacune des instructions considérées.

6) Proposez un morceau de programme pour échanger le contenu des cases mémoires $(130)_H$ et $(131)_H$. Écrivez votre proposition en langage d'assemblage.

Exercice 3 : Instructions de branchement et boucles

Un processeur 8 bits est doté d'un espace mémoire de 64 Kio ; à chaque adresse en mémoire centrale correspond une case de 1 octet. Le processeur dispose de 4 registres de travail, indicés de 0 à 4 et notés `R0`... `R3`, ainsi que d'un *program status register* (le PSR), qui comporte un drapeau nommé `z` : ce drapeau est à 1 si la dernière valeur écrite dans l'un des registres était nulle, 0 sinon. Les instructions sont codées sur 1 ou 2 octets, comme indiqué dans le tableau ci-dessous :

		premier octet								second octet
assembleur	action	7	6	5	4	3	2	1	0	7...0
`LD RD,adr`	`RD <- Mémoire[adr]`	0	0	0	RD		0	0	0	adr
`ST RS,adr`	`Mémoire[adr] <- RS`	0	0	1	SR		0	0	0	adr
`ADD RD,RS`	`RD <- RD + RS`	0	1	0	RD		RS		0	—
`SUB RD,RS`	`RD <- RD - RS`	0	1	1	RD		RS		0	—
`BR dec5`	`PC <- adins + 1 + dec5`	1	0	0	dec5					—
`BRZ dec5`	Si z = 1 alors `PC <- adins + 1 + dec5`	1	0	1	dec5					—

Dans ce tableau :

— `RD` désigne un registre de destination et `RS` un registre source (`R0`... `R3`) ;

— `PC` désigne le compteur de programme (adresse de la prochaine instruction) ;

— `adr` et `adins` désignent des adresses en mémoire, codées sur 1 octet ;

— `Mémoire[adr]` désigne la case mémoire d'adresse `adr`.

Principe des instructions de branchement `BR` et `BRZ` : ces instructions permettent de faire des sauts dans le programme, en modifiant la valeur de `PC`. On note `adins` l'adresse de l'instruction de branchement dans le programme, et `dec5` est un entier (décalage) codé en complément à 2 sur 5 bits par rapport à cette adresse :

— Dans le cas de `BR dec5`, lorsque l'instruction est exécutée, alors l'instruction qui sera exécutée ensuite est celle à l'adresse `adins + 1 + dec5` (branchement inconditionnel).

— Dans le cas de `BRZ dec5` : si `z` = 1 lorsque l'instruction est exécutée, alors la prochaine instruction exécutée est celle à l'adresse `adins + 1 + dec5` (branchement pris), sinon la prochaine instruction sera celle à l'adresse `adins + 1` (branchement non-pris).

Un programmeur qui écrit en langage d'assemblage n'a pas besoin de calculer lui même les adresses de son programme : il utilise simplement une étiquette pour désigner l'adresse d'une instruction ou d'une donnée, comme dans le programme ci-dessous. Dans le cas des instructions de branchement, l'assembleur calculera lui-même les valeurs de décalage (`dec5`).

```
one:     1            // case mémoire contenant l'entier 1
n:       15           // case mémoire contenant l'entier 15
r:       0            // case mémoire contenant l'entier 0
         LD R0,r      // ........................................
         LD R1,one    // ........................................
         LD R2,n      // ........................................
loop:    BRZ endloop  // ........................................
         ADD R0,R2    // ........................................
         SUB R2,R1    // ........................................
         BR loop      // ........................................
endloop: ST R0,r      // ........................................
```

1) Si un programme comporte une instruction `BR dec5` à l'adresse `adins`, dans quel intervalle pourra se trouver l'adresse `adsuiv` de la prochaine instruction à être exécutée ? Justifiez votre réponse.

2) Les instructions `LD` et `ST` décrites dans l'énoncé ne permettent pas d'accéder à tout l'espace mémoire dont dispose le processeur : pourquoi ?

3) Ajouter des commentaires succincts au programme ci-dessus, en indiquant l'action algorithmique effectuée par chaque instruction. Expliquez ensuite ce que calcule ce programme.

4) On suppose que l'étiquette `one` sera rangée en mémoire à l'adresse $(001A)_H$. Complétez le tableau ci-dessous, en indiquant pour chaque donnée son adresse, et pour chaque instruction l'adresse de son premier octet.

adresse	instruction	
$(001A)_H$	`one:`	`1`
	`n:`	`15`
	`r:`	`0`
	`begin:`	`LD R0,r`
		`LD R1,one`
		`LD R2,n`
	`loop:`	`BRZ endloop`
		`ADD R0,R2`
		`SUB R2,R1`
		`BR loop`
	`endloop:`	`ST R0,r`

5) Calculez la valeur de décalage `dec5` pour l'instruction `BRZ endloop`, puis pour l'instruction `BR loop` : détaillez bien vos calculs. Donnez votre résultat en décimal puis en binaire.

6) Donnez le codage en binaire du programme (rayez les octets inutiles) :

instruction		premier octet								second octet							
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
`one:`	`1`																
`n:`	`15`																
`r:`	`0`																
`begin:`	`LD R0,r`																
	`LD R1,one`																
	`LD R2,n`																
`loop:`	`BRZ endloop`																
	`ADD R0,R2`																
	`SUB R2,R1`																
	`BR loop`																
`endloop:`	`ST R0,r`																

7) Donnez le codage en mémoire du programme, en donnant le contenu des cases de la mémoire en hexadécimal :

adresse	contenu
$(001A)_H$	

Exercice 4 : Codage des instructions

Soit un processeur 8 bits, équipé de 4 registres de 8 bits (nommés `r0`, `r1`, `r2` et `r3`) et d'un bus d'adresses sur 16 bits. Son jeu d'instructions comporte :

— deux instructions d'accès à la mémoire :

  — « `st rs, @mem` » : stocke la valeur du registre `rs` dans l'octet d'adresse `@mem` ;
  — « `ld rd, @mem` » : stocke le contenu de l'octet d'adresse `@mem` dans le registre `rd` ;
— des instructions réalisant des opérations arithmétiques ou logiques :
  — « `op rd, rs` » avec op appartenant à :
    — `add` (`rd = rd + rs`),
    — `sub` (`rd = rd - rs`),
    — `and` (`rd = rd & rs`),
    — `or` (`rd = rd | rs`),
    — `xor` (`rd = rd ⊕ rs`).
  — « `op rd` » avec op appartenant à :
    — `not` (`rd = ~ rd`),
    — `shr` (`rd = rd » 1`, décalage arithmétique vers la droite),
    — `shl` (`rd = rd « 1`, décalage arithmétique vers la gauche).

Partie 1 : Codage des instructions

On veut écrire un programme destiné à ce processeur et qui réalise le calcul : `A = 2 * (B + C) - 18`. On suppose que le contenu de la variable A est à l'adresse 1230_H, celui de B à l'adresse 1231_H et celui de C à l'adresse 1232_H. On peut stocker des constantes à partir de l'adresse 2000_H.

1) Quel doit être le contenu de la mémoire avant l'exécution du programme si on veut que `B` soit égal à 10 et `C` à 26 ?

2) Écrire un programme utilisant le jeu d'instructions du processeur et calculant `A = 2 * (B + C) - 18` (pour aller plus loin : comment faire pour multiplier par 5, ou une autre « petite » constante entière ?).

3) Proposez un codage de chaque instruction en essayant de minimiser sa taille (qui sera néanmoins forcément multiple de l'octet). Vous devrez pour cela proposer un codage de taille variable, pouvant comporter plusieurs octets par instruction.

4) Combien de codages restent pour coder d'autres instructions ?

5) En supposant que le processeur va chercher la première instruction à l'adresse 4000_H, donner la représentation en mémoire du programme pour le calcul de `A = 2 * (B + C) - 18`.

Exercice 5 : Taille du bus, volume de mémoire centrale

Sur un certain ordinateur, on suppose que le bus entre le processeur et la mémoire centrale comporte 32 fils d'adresse.

1. Si à chaque adresse de la mémoire centrale correspond un octet :
   (a) Quel est le nombre d'octets adressables ?
   (b) Quelle est la taille maximale de la mémoire ? Exprimez votre réponse en octet puis en Gio.
   (c) Combien de fils de donnée doit comporter le bus ?
2. Si à chaque adresse correspond un mot de 32 bits :
   (a) Quel est le nombre de mots adressables ?
   (b) Quelle est la taille maximale de la mémoire ? Exprimez votre réponse en octet puis en Gio.
   (c) Combien de fils de donnée doit comporter le bus ?

Exercice 6 : Langage machine

Un processeur 8 bits est doté d'un espace mémoire de 64 Kio ; à chaque adresse en mémoire centrale correspond une case de 1 octet. Le processeur dispose de 4 registres de travail, indicés de 0 à 4 et notés `R0`...`R4`. Les instructions sont codées sur 1 ou 2 octets, comme indiqué dans le tableau ci-dessous :

		premier octet								second octet
assembleur	action	7	6	5	4	3	2	1	0	7…0
`HALT`	Met fin à l'exécution du programme	0	0	0	0	0	0	0	0	—
`ADD RD,RS`	`RD <- RD + RS`	0	1	0	1	RD		RS		—
`LD RD,adr`	`RD <- Mémoire[adr]`	1	0	1	0	RD		0	0	adr
`ST RS,adr`	`Mémoire[adr] <- RS`	1	0	1	1	RS		0	0	adr

Dans ce tableau :

— `RD` désigne un registre de destination et `RS` un registre source (`R0`…`R4`) ;

— `adr` désigne une adresse en mémoire, codée sur 1 octet ;

— `Mémoire[adr]` désigne la case mémoire d'adresse `adr`.

Un programmeur qui écrit en langage d'assemblage n'a pas besoin de calculer lui-même les adresses de son programme : il utilise simplement une étiquette pour désigner l'adresse d'une instruction ou d'une donnée, et l'assembleur se charge du calcul des adresses. Dans le programme ci-dessous, les étiquettes `zero`, `a` et `b` désignent l'adresse de donnée ou de résultats en mémoire.

```
         LD R0,zero   // ..........................................
         LD R1,a      // ..........................................
         ADD R1,R1    // ..........................................
         ADD R0,R1    // ..........................................
         ADD R1,R1    // ..........................................
         ADD R0,R1    // ..........................................
         ST R0,b      // ..........................................
         HALT         // ..........................................
zero:    0            // constante décimal 0 sur 8 bits
a:       43           // constante décimal 43 sur 8 bits
b:       0            // case de 8 bits où est stocké le résultat
```

1) Ajouter des commentaires au programme ci-dessus, en indiquant l'action effectuée par chaque instruction. Vous utiliserez les étiquettes `a` et `b` comme des noms de variables pour clarifier vos commentaires. Exprimez ensuite la valeur de `b` en fonction de `a` en fin de programme.

2) On regarde le contenu des registres et des cases mémoire comme des entiers naturels codés en binaire sur 8 bits. Donnez la valeur décimal que prend `b` après l'exécution du programme. Détaillez vos calculs.

3) On suppose que le programme sera rangé en mémoire à partir de l'adresse $(1A)_H$. Complétez le tableau ci-dessous, en indiquant pour chaque instruction l'adresse de son premier octet et pour chaque donnée son adresse.

adresse	instruction
$(1A)_H$	`LD R0,zero`
	`LD R1,a`
	`ADD R1,R1`
	`ADD R0,R1`
	`ADD R1,R1`
	`ADD R0,R1`
	`ST R0,B`
	`HALT`
	`zero: 0`
	`a: 43`
	`b: 0`

4) Donnez le codage en binaire du programme (rayez les octets inutiles) :

instruction	premier octet								second octet							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
`LD R0,zero`																
`LD R1,a`																
`ADD R1,R1`																
`ADD R0,R1`																
`ADD R1,R1`																
`ADD R0,R1`																
`ST R0,b`																
`HALT`																
`zero: 0`																
`a: 43`																
`b: 0`																

5) Donnez le codage en mémoire du programme, en donnant le contenu des cases de la mémoire en hexadécimal :

adresse	contenu
$(1A)_H$	

Thème 2
Codage des nombres

Dans cette partie, le petit encadré « **à savoir !** » est utilisé au début des paragraphes qui peuvent être considérés comme du cours.

Exercice 1 : Représentation positionnelle des entiers naturels

à savoir ! Soit $\beta \in \mathbf{N}$, $\beta > 1$, une base. Tout $n \in \mathbf{N}$ peut être représenté de manière unique par sa *représentation positionnelle en base* β :

$$(x_{p-1}x_{p-2}\cdots x_1x_0)_\beta := \sum_{i=0}^{p-1} x_i\beta^i.$$

Les $x_i \in \{0,1,\ldots,\beta-1\}$ sont les *chiffres* de l'écriture de n en base β, p est le nombre de chiffres nécessaires pour écrire de l'entier naturel n. On attribue un symbole à chaque chiffre : chiffres 0 et 1 en binaire, chiffres de 0 à 9 en décimal, chiffres de 0 à F en hexadécimal.

1) La définition permet directement d'effectuer des changements de base : si on connaît l'écriture $(x_{p-1}\ldots x_0)_\beta$ de n, et qu'on veut connaître cette écriture en base γ, il suffit de convertir les chiffres x_i en base γ, puis de calculer $\sum_{i=0}^{p-1} x_i\beta^i$ en base γ. En utilisant cette technique, convertissez $(11011)_2$ puis $(56)_9$ en décimal.
2) Une autre technique de changement de base est celle des divisions euclidiennes successives. Rappelez et justifiez cette méthode. Utilisez là pour convertir $n = (423)_{10}$ en binaire, puis $(3452)_{10}$ en base 8.
3) Entre les bases 2, 8 et 16, des méthodes plus directes peuvent être utilisées : par exemple, tout chiffre octal est représenté par un entier sur trois bits, et tout entier sur trois bits est représenté par un chiffre octal. Justifiez cette méthode de convertion entre les bases 2 et 8. Convertissez $(34521)_8$ en base 2 puis 16.

Exercice 2 : Représentation en complément à deux sur p bits

Partie 1 : Définition

à savoir ! Soit un entier relatif n à coder en machine sur p bits. On adopte la définition suivante pour le codage de n en complément à 2 sur p bits :

$$(c_{p-1}c_{p-2}\ldots c_1c_0)_{\bar{2}} := -c_{p-1}2^{p-1} + \sum_{i=0}^{p-2} c_i 2^i.$$

Une manière d'interpréter le codage en complément à 2 est donc de considérer que le bit le plus à gauche a un poids négatif (-2^{p-1}). On peut montrer que $n \geq 0$ ssi $c_{p-1} = 0$, et $n < 0$ ssi $c_{p-1} = 1$.

1) Quelle est la valeur décimale codée par $(10000011)_{\bar{2}}$? Par $(00001010)_{\bar{2}}$?
2) Comment coder $(-120)_{10}$ en complément à 2 sur 8 bits ?
3) Quelle est le plus grand entier représentable en complément à 2 sur p bits ? Le plus petit ?

Partie 2 : Addition et calcul de l'opposé

à savoir ! Soient $m = (c_m)_{\bar{2}}$ et $n = (c_n)_{\bar{2}}$ en complément à 2 sur p bits. On dit qu'il y a dépassement de capacité dans une opération en complément à 2 sur p bits lorsque le résultat de l'opération ne peut pas être représenté sous cette même forme (il est soit trop petit, soit trop grand).

— En l'absence de dépassement de capacité, le codage de $m+n$ en complément à 2 sur p bits est le codage de l'entier naturel $(c_m+c_n) \mod 2^p$. Si m et n sont de même signe, il y dépassement ssi le signe du résultat calculé diffère du signe des opérandes ; s'ils sont de signes opposés, aucun dépassement n'est possible.

— En l'absence de dépassement de capacité, le codage de $-n$ en complément à 2 sur p bits est le même que celui de l'entier naturel $(\overline{c}_{p-1}\overline{c}_{p-2}\dots\overline{c}_1\overline{c}_0)_2 + 1 \mod 2^p$. Il y dépassement ssi le signe du résultat calculé est le même que celui de l'opérande.

1) Posez, en complément à deux sur 8 bits, les additions suivantes : $(10001010)_{\overline{2}} + (00001011)_{\overline{2}}$, $(10001010)_{\overline{2}} + (10001011)_{\overline{2}}$, $(01001010)_{\overline{2}} + (11001010)_{\overline{2}}$.

2) Calculez l'opposé de $(10001010)_{\overline{2}}$ en complément à 2 sur 8 bits, et vérifiez que votre résultat est correct.

3) En complément à 2 sur p bits, quel est le seul cas produisant un dépassement de capacité pour le calcul de l'opposé ?

Partie 3 : Extension de signe en complément à 2

1) Comment sont représentés $(34)_{10}$ et $(-42)_{10}$ en complément à 2 sur 8 bits (complément à 2^8) ?

2) Comment sont représentés $(34)_{10}$ et $(-42)_{10}$ en complément à 2 sur 12 bits (complément à 2^{12}) ?

3) Proposez et justifiez une règle permettant de passer de l'écriture d'un entier relatif en complément à 2 sur p bits à son écriture en complément à 2 sur $p+k$ bits (en conservant l'égalité des valeurs bien-entendu).

Exercice 3 : Représentation positionnelle des rationnels

à savoir ! Les rationnels sont les nombres de la forme $\frac{p}{q}$, avec $p \in \mathbf{Z}$ et $q \in \mathbf{N} - \{0\}$. Tout $x \in \mathbf{Q}$ positif peut être décomposé en une partie entière $\lfloor x \rfloor \in \mathbf{N}$ ($\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$) et une partie fractionnaire $\{x\} = x - \lfloor x \rfloor$ ($0 \leq \{x\} < 1$). On utilise la notation positionnelle pour l'écriture de $\{x\}$: s'il existe $q \in \mathbf{N}$ t.q.

$$\{x\} = (0, x_{-1} \dots x_{-q})_\beta = \sum_{i=1}^{q} x_{-i}\beta^{-i},$$

alors $x = (x_{p-1}x_{p-2}\dots x_0,x_{-1}x_{-2}\dots x_{-q})_\beta$ est l'écriture x en base β. Le problème est que l'écriture d'un rationnel en base β n'est pas forcément finie : en machine, il faut souvent se contenter d'une approximation. Par contre, on sait que l'écriture d'un rationnel est nécessairement périodique.

à savoir ! Soit un rationnel $0 \leq x < 1$. On veut déterminer son écriture $(0, x_{-1}x_{-2}x_{-3}\dots)_2$ en binaire. On a $2 \times x = (x_{-1}, x_{-2}x_{-3}\dots)_2$, donc $x_{-1} = \lfloor 2 \times x \rfloor$. En procédant par des multiplications successives par 2, on peut ainsi extraire un par un les bits de l'écriture (binaire) de x.

1) Quelle est l'écriture de $13/7$ en base $\beta = 10$? L'écriture est périodique, on soulignera la période.

2) Convertir $1/10 = (0,1)_{10}$ en écriture binaire.

3) Donnez l'écriture en décimal de $(11,1001)_2$.

4) Donnez une écriture sous la forme d'une fraction de deux nombres décimaux de $(0,0\underline{101})_2$.

Exercice 4 : Représentation à virgule flottante, introduction

Partie 1 : Définition

à savoir ! On s'intéresse aux nombres à virgule flottante binaires à p bits de précision, que nous appellerons aussi *nombres flottants*. Un *nombre flottant normalisé* x est soit 0, soit un rationnel de la forme

$$x = (-1)^s \times (1, x_1 \dots x_{p-1})_2 \times 2^e,$$

où $s \in \{0,1\}$ est son *signe*, $(1, x_1 \dots x_{p-1})_2$ est sa *mantisse*, et $e \in \mathbf{Z}$ est son *exposant*, t.q. $e_{\min} \leq e \leq e_{\max}$.

1) Lister l'ensemble des flottants normalisés positifs avec 3 bits de précision, $e_{\min} = -1$ et $e_{\max} = 1$:

2) Déterminez le plus petit flottant normalisé positif non nul λ, le plus grand flottant normalisé σ, et l'écart entre deux flottants consécutifs de même exposant e.

Partie 2 : Représentation en machine, norme IEEE-754 :

à savoir ! La norme IEEE-754 définit la manière dont sont représentés les flottants sur les ordinateurs actuels. On considère le code suivant :

$c =$	s	code de l'exposant : c_e	code de la mantisse : c_m
	1 bit	k bits	$p-1$ bits

Pour un flottant normal, la valeur codée est $f(c) = (-1)^{s(c)} \times m(c) \times 2^{e(c)}$, avec

— $s(c) = s$ le signe du flottant.

— $m(c) = (1, c_m)_2$, ce qui signifie que le 1 de tête est codé implicitement.

— $e(c)$ dépend de $(c_e)_2$, mais doit être interprété avec moultes précautions...

Comme c_e est un code sur k bits, notons que $0 \le (c_e)_2 \le 2^k - 1$.

— $(c_e)_2 = 0$ et $(c_m)_2 = 0$ indique que le nombre représenté est 0. Il y a deux codages de 0 : -0 ou $+0$ selon s.

— $(c_e)_2 = 2^k - 1$ indique une valeur exceptionnelle (+Inf, -Inf, NaN) : $1/0 = +\text{Inf}$, $0/0 = \text{NaN}$, $a + \text{Inf} = \text{Inf}$, ...

— $1 \le (c_e)_2 \le 2^k - 2$ indique que le nombre représenté est normal. Alors,

$$m(c) = (1, c_m)_2 \quad \text{et} \quad e(c) = (c_e)_2 - \underbrace{(2^{k-1} - 1)}_{\text{biais}}.$$

Les types float et double du C correspondent à deux formats prévus par la norme IEEE-754, appelés *simple précision* et la *double précision* :

format	mantisse		exposant				taille
	précision	$p-1$	k	biais	$e_{\min}$	$e_{\max}$	
simple	$p = 24$	23	8	127	-126	127	32 bits
double	$p = 53$	52	11	1023	-1022	1023	64 bits

1) Donnez les valeurs extrêmes λ et σ pour la simple et la double précision.

2) Donnez le codage de $(1,5)_{10}$ en simple précision, sous la forme d'un code en hexadécimal.

Partie 3 : Arrondi correct

à savoir ! Pour représenter un réel r par un nombre flottant, il faut en général arrondir r. La norme IEEE-754 propose 4 modes d'arrondi, mais on n'utilise que l'*arrondi au plus proche* (RN) dans ce TD : RN(r) est le flottant le plus proche de r, et si r est équidistant de deux flottants consécutifs alors RN(r) est celui dont la mantisse se termine par un 0. La norme impose *l'arrondi correct* pour les opérations +, -, ×, / et $\sqrt{\cdot}$: le résultat calculé doit être le résultat exact arrondi selon le mode d'arrondi courant.

1) Donnez la représentation positionnelle en base 2 de $(0,1)_{10}$.

2) Est-il possible de représenter exactement $(0,1)_{10}$ par un flottant binaire ?

3) Représentez $(0,1)_{10}$ par un flottant sur 8 bits de précision. S'il y a lieu, effectuez un arrondi au plus proche.

Exercice 5 : Représentation à virgule flottante

On travaille sur une machine où les flottants binaires sont codés sur 12 bits. On considère le code :

$c =$	11	10	9	8	7	6	5	4	3	2	1	0
	s	b				m						

Le nombre représenté est $f(c) = (-1)^s \times (1, m)_2 \times 2^e$, avec $e = (b)_2 - 2^3$ si $1 \le (b)_2 \le 14$, (on ne se préoccupe pas du codage des sous-normaux ni des valeurs exceptionnelles ici).

1) Quel nombre x_1 est représenté par le code hexadécimal $c = 637_H$, si on le regarde comme le codage d'un flottant ? Donnez votre résultat sous la forme d'une représentation positionnelle à virgule en décimal.

2) Quel est le nombre flottant x_2 immédiatement supérieur à x_1 ?

— Donnez votre résultat sous la forme $x_2 = (-1)^s \times (1, m)_2 \times 2^e$,

— puis donnez le codage de x_2 en hexadécimal.

3) Comment est représenté le nombre $(13/3)_{10}$ en représentation positionnelle à virgule en binaire ? Détaillez un peu vos calculs.

4) Comment peut-on représenter le nombre $(13/3)_{10}$ dans le format flottant binaire décrit ci-dessus ? Vous effectuerez un arrondi au plus proche, et exprimerez votre résultat sous la forme d'un code binaire, puis en hexadécimal.

Exercice 6 : Codage de nombres en machine

1) Convertir les nombres suivants en base 10 : $(1011)_2$, $(10110)_2$, $(101.1)_2$, $(0.1101)_2$, $(110.01)_2$.

2) Convertir les nombres suivants en base 10 : $(FF)_{16}$, $(1A)_{16}$, $(789)_{16}$, $(0.13)_{16}$, $(ABCD.EF)_{16}$.

3) Convertir les nombres suivants (exprimés en base 10) en base 2 et en base 16 : 12, 24, 192, 2079, 0.25, 0.375, 0.376 et 17.150.

Exercice 7 : Conversion base 10 vers base 2 d'entiers naturels

Le principe de l'algorithme conversion de Horner de l'entier naturel $n = (x_3x_2x_1x_0)_\beta$ est le suivant. On a :

$$n = x_3{\cdot}\beta^3 + x_2{\cdot}\beta^2 + x_1{\cdot}\beta^1 + x_0, = (x_3{\cdot}\beta^2 + x_2{\cdot}\beta^1 + x_1){\cdot}\beta + x_0, = ((x_3{\cdot}\beta + x_2){\cdot}\beta + x_1){\cdot}\beta + x_0.$$

Si on exécute l'algorithme suivant,

$r_3 = x_3$
$r_2 = r_3 \times \beta + x_2$
$r_1 = r_2 \times \beta + x_1$
$r_0 = r_1 \times \beta + x_0$

alors par construction on a l'égalité $r_0 = x_3{\cdot}\beta^3 + x_2{\cdot}\beta^2 + x_1{\cdot}\beta^1 + x_0$. Si on effectue tous les calculs dans une base d'arrivée γ, on obtient l'écriture de n en base γ.

1) Testez l'algorithme de Horner pour convertir l'entier $n = (11010101)_2$ en décimal.

2) Plus fastidieux à faire à la main : testez l'algorithme de Horner pour convertir l'entier $n = (567)_{10}$ en binaire. Combien de multiplications avez vous posées ? Combien de multiplications auriez vous posées si vous aviez utilisé la définition de la notation positionnelle pour faire cette conversion ?

3) Ecrivez maintenant deux fonctions en `C` pour convertir une chaîne de caractères (tableau de codes ASCII terminé par un `0`) contenant l'écriture d'un entier positif en décimal, en un entier non-signé en machine [1] : l'une utilisera naïvement la définition de la notation positionnelle, l'autre l'algorithme de Horner. Dans les deux cas, combien de multiplications sont utilisées ?

4) Pour poursuivre : dans les deux fonctions de conversion, peut-il se produire un dépassement de capacité ? Le cas échéant, que vont retourner ces fonctions ? Dans la même veine, mettre au point une fonction pour convertir un entier naturel écrit en hexadécimal vers un entier naturel du type `unsigned int`.

Exercice 8 : Représentation à virgule flottante

On travaille sur une machine où les flottants binaires sont codés sur 12 bits. On considère le code :

$c =$

11	10	9	8	7	6	5	4	3	2	1	0
s	b				m						

Le nombre représenté est $f(c) = (-1)^s \times (1,m)_2 \times 2^e$, avec $e = (b)_2 - 2^3$ si $1 \le (b)_2 \le 14$, (on ne se préoccupe pas du codage des sous-normaux ni des valeurs exceptionnelles ici).

Partie 1 :

1) Comment est représenté le nombre $(11,1)_{10}$ en représentation positionnelle à virgule en binaire ? Détaillez vos calculs.

2) Comment peut-on représenter le nombre $x_1 = (11,1)_{10}$ dans le format flottant binaire décrit ci-dessus ? Vous effectuerez un arrondi au plus proche, et exprimerez votre résultat sous la forme d'un code binaire, puis en hexadécimal.

1. Vous avez peut-être déjà utilisé les fonctions `sscanf` ou `atoi` du C99 pour faire cela, mais il s'agit ici de s'en passer.

Partie 2 :

On souhaite ajouter les deux nombres flottants $x_1 = (1,1110001)_2 \times 2^2$ et $x_2 = (1,1100101)_2 \times 2^4$, et calculer l'arrondi au plus proche du résultat exact.

1) Posez l'opération demandée en utilisant le « formulaire » ci-dessous, de manière à obtenir le résultat exact (avant arrondi). Il vous est demandé de décaler la virgule dans x_2 de manière à ce qu'il partage le même exposant que x_1. On notera x_3 le résultat exact de l'opération.

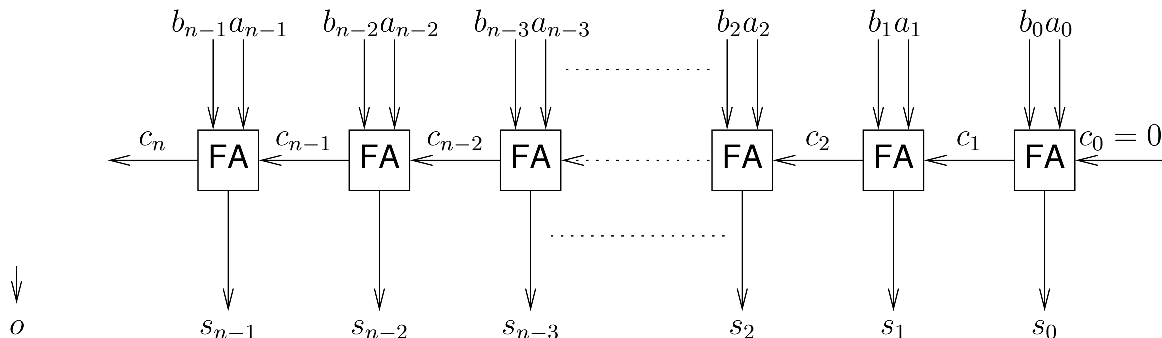
$$\begin{array}{rl} & (\sqcup\sqcup\sqcup\sqcup\sqcup,\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup)_2 \times 2^{\sqcup} \\ + & (\sqcup\sqcup\sqcup\sqcup\sqcup,\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup)_2 \times 2^{\sqcup} \\ \hline & (\sqcup\sqcup\sqcup\sqcup\sqcup,\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup)_2 \times 2^{\sqcup} \end{array}$$

2) Donnez une approximation de x_3 par un flottant binaire, en effectuant un arrondi au plus proche. Vous donnerez le résultat sous la forme $(1,m)_2 \times 2^e$.

Thème 3
Circuits combinatoires

Exercice 1 : Addition-soustraction en complément à 2 sur n bits

Soient a et b deux entiers relatifs connus par leur codage en complément à 2 sur n bits ($n \geq 2$) : $a = (a_{n-1}a_{n-2}\dots a_1a_0)_{\overline{2}}$ et $b = (b_{n-1}b_{n-2}\dots b_1b_0)_{\overline{2}}$. On étudie dans un premier temps l'addition de a et de b à l'aide d'un additionneur à propagation de retenue classique, fabriqué à l'aide de composants *full-adders* (FA) :



On rappelle les éléments suivants sur la représentation en complément à 2.

— Par définition, $(a_{n-1}a_{n-2}\dots a_1a_2)_{\overline{2}} = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$.

— Le bit a_{n-1} est appelé le bit de signe dans la représentation de a en complément à 2 sur n bits.

— Soit $s = (s_{n-1}s_{n-2}\dots s_1s_0)_{\overline{2}}$ le résultat calculé par l'additionneur représenté ci-dessus. On dit qu'il y a dépassement de capacité dans le calcul de $a+b$ en complément à 2 sur n bits si $s \neq a+b$ (cela se produit si $a+b$ est soit « trop grand », soit « trop petit » pour être représenté en complément à 2 sur n bits).

Partie 1 : Dépassement de capacité

On souhaite ajouter à notre additionneur une sortie o (comme *overflow*) permettant de détecter si on se trouve en présence d'un cas de dépassement de capacité.

1) Posez, en complément à 2 sur 8 bits, les additions avec les opérandes a et b indiquées ci-dessous ; A chaque fois, donnez la valeur des opérandes et celle du résultat calculé (qui peut être différent du résultat exact) en décimal, en précisant bien le signe de ces valeurs. Indiquez clairement les opérations pour lesquelles il y a eu dépassement de capacité.

— $a = (10001111)_{\overline{2}}$, $b = (10010000)_{\overline{2}}$,

— $a = (01111111)_{\overline{2}}$, $b = (10000000)_{\overline{2}}$,

— $a = (01111111)_{\overline{2}}$, $b = (00000001)_{\overline{2}}$.

2) En complément à 2 sur n bits, quel est le plus petit entier (négatif) représentable, et quel est le plus grand entier (positif) représentable ?

3) Montrez que si a et b sont de signes opposés (ou tous deux égaux à 0), alors il ne peut pas y avoir de dépassement de capacité dans le calcul de $a+b$ en complément à 2 sur n bits.

4) Déduire de la question précédente une condition nécessaire pour qu'il y ait dépassement de capacité.

5) Donnez une condition nécessaire et suffisante portant sur a_{n-1}, b_{n-1} et s_{n-1} pour qu'il y ait dépassement de capacité. Notez que a_{n-1}, b_{n-1} et s_{n-1} sont les bits de signe dans la représentation en complément à 2 sur n bits de a, b et s respectivement.

6) Complétez le tableau ci-dessous, dans lequel on ne s'intéresse qu'au calcul effectué par le FA le plus à gauche. L'indicateur de dépassement o doit prendre la valeur 1 en cas de dépassement de capacité, la valeur 0 sinon.

a_{n-1}	b_{n-1}	c_{n-1}	s_{n-1}	c_n	o
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

7) En vous référant à la table de vérité précédente, exprimez o en fonction de c_{n-1} et c_n à l'aide d'une formule booléenne. Justifiez votre réponse.

8) Complétez le circuit de l'additionneur afin qu'il produise la sortie o.

Partie 2 : Additionneur-soustracteur

On souhaite maintenant compléter le circuit de manière à ce qu'il puisse aussi effectuer des soustractions en complément à 2 sur n bits. Pour cela, on ajoute une entrée e qui indiquera si le circuit doit calculer l'addition ou la soustraction des deux entiers placés sur ses entrées. En l'absence de dépassement de capacité,

— si $e = 1$ alors $s = a + b$,

— si $e = 0$ alors $s = a - b$.

On note $d = (d_{n-1} d_{n-2} \cdots d_1 d_0)$ le mot de n bits tel que

— si $e = 1$ alors $d = b$,

— si $e = 0$ alors $d = \overline{b}$, où $\overline{b}$ désigne le complément à 1 de b.

1) Rappelez comment peut être effectuée la soustraction $a - b$ en complément à 2 sur n bits. Vous supposerez dans vos explications qu'aucun problème de dépassement de capacité se produit.

2) Comment peut-on détecter les cas de dépassement de capacité dans le cas de la soustraction ?

3) Soit i un entier tel que $0 \leq i < n$. Complétez la table de vérité ci-dessous, et exprimez d_i par une fonction booléenne.

b_i	e	d_i
0	0	
0	1	
1	0	
1	1	

4) Expliquez en quelques phrases comment vous allez calculer le résultat de la soustraction dans le cas $e = 0$.

5) Donnez le circuit d'un additionneur-soustracteur en complément à 2 sur 8 bits, avec une sortie permettant de détecter les cas de dépassement de capacité. Veillez à annoter votre schéma (il faut faire apparaître en particulier : les a_i, b_i, c_i, d_i, s_i, ainsi que e et o).

Exercice 2 : Comparaison d'entiers naturels

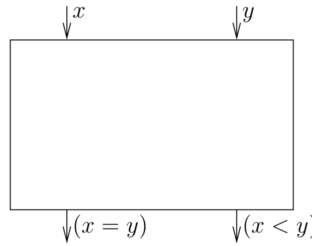
On souhaite mettre au point un circuit permettant de comparer deux entiers naturels codés en binaire, $a = (a_3 a_2 a_1 a_0)_2$ et $b = (b_3 b_2 b_1 b_0)_2$. La sortie du circuit devra prendre pour valeur 1 si $a \leq b$, 0 sinon. On va d'abord s'intéresser à la comparaison de deux bits x et y avant de passer à celle des entiers naturels : l'expression $(x = y)$ vaut 1 si x et y sont égaux, 0 sinon ; l'expression $(x \leq y)$ vaut 1 si $(x)_2 \leq (y)_2$, 0 sinon.

1) Complétez les tables de vérités suivantes, puis donnez des formules booléennes pour $(x = y)$ et pour $(x < y)$.

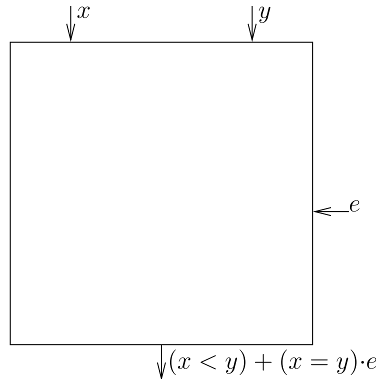
x	y	$(x = y)$
0	0	
0	1	
1	0	
1	1	

x	y	$(x < y)$
0	0	
0	1	
1	0	
1	1	

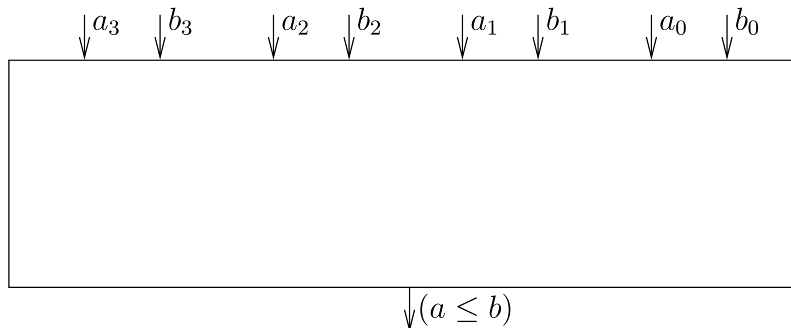
2) Complétez le circuit ci-dessous, de manière à ce que ses sorties produisent les valeurs de $(x = y)$ et $(x < y)$. Dans la suite, on appellera `HC` le composant obtenu.



3) Complétez le circuit ci-dessous, qui prend en entrée les deux bits x et y, ainsi qu'une entrée supplémentaire e, et dont la sortie prend la valeur $s = (x < y) + (x = y) \cdot e$. Dans la suite, on appellera FC le composant obtenu.



4) Complétez le circuit ci-dessous, de manière à réaliser la comparaison entre $a = (a_3 a_2 a_1 a_0)_2$ et $b = (b_3 b_2 b_1 b_0)_2$: la sortie prendra pour valeur 1 si $a \leq b$, 0 sinon. Expliquez votre raisonnement.



Exercice 3 : Décalage

On souhaite mettre au point un circuit permettant le décalage à droite ou à gauche d'un mot de 5 bits. Le circuit prend en entrée un mot $(e_4 e_3 e_2 e_1 e_0)$ et présente en sortie le résultat du décalage $(s_4 s_3 s_2 s_1 s_0)$. Une ligne de contrôle c permet de spécifier la direction du décalage.

— $c = 1$ correspond à un décalage à droite de 1 bit ; un 0 est alors inséré au niveau de s_4.

— $c = 0$ correspond à un décalage à gauche de 1 bit ; un 0 est alors inséré au niveau de s_0.

1) Pour $3 \leq i \leq 1$, proposez un circuit à base de portes NOT, AND et OR permettant de déterminer s_i en fonction de e_{i+1}, e_{i-1} et c. On voit que dans les deux cas chaque bit de la sortie s est calculé à l'aide des deux bits adjacents de l'entrée e. Par exemple, le bit s_2 vaut soit e_1, soit e_3 (cela dépend de la valeur de c).

2) Proposez un circuit permettant de déterminer s_0 en fonction de e_1, et c. Proposez un circuit pour déterminer s_4 en fonction de e_3, et c.

3) Représentez le décaleur 5 bits demandé.

Vous pouvez réfléchir à la manière d'associer les décaleurs, de manière par exemple à pouvoir fabriquer un décaleur 4 bits avec deux décaleurs 2 bits.

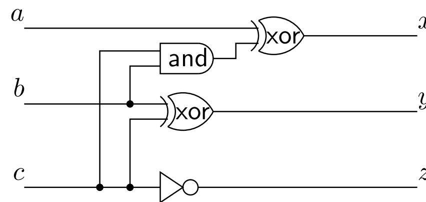
Exercice 4 : Générateur de parité impaire

Un générateur de parité impaire est une fonction qui retourne 1 si le nombre de bits à 1 parmi ses entrées est impair, 0 sinon. Définissez cette fonction pour un mot de 8 bits, et donnez un circuit logique implantant cette fonction. Vous pouvez commencer par donner une version naïve de votre circuit, mais vous essayerez ensuite de réduire sa latence.

Exercice 5 : Des circuits combinatoires

Partie 1 : Analyse d'un circuit combinatoire

On considère le circuit combinatoire suivant, dont les entrées sont a, b et c, et les sorties x, y et z :



1) Donnez des expressions booléennes pour x, y et z en fonction de a, b et c.

2) Complétez la table de vérité suivante :

a	b	c	x	y	z
⋮	⋮	⋮	⋮	⋮	⋮

3) Quelle fonction réalise le circuit considéré ?

Partie 2 : Décodeurs

Un décodeur k bits est un circuits a k entrées $e_{k-1}, \ldots, e_0$ et 2^k sorties $s_{2^k-1}, \ldots, s_0$: la sortie $s_{(e_{k-1},\ldots,e_0)_2}$ dont l'indice est indiqué par les entrées est activée et toutes les autres restent inactives.

1) On représente de la manière indiquée ci-dessous un décodeur 1 vers 2.

e_0	s_1	s_0
⋮	⋮	⋮

Complétez sa table de vérité, et donnez un logigramme pour un décodeur 1 vers 2.

On souhaite fabriquer un décodeur 2 vers 4 à partir de deux décodeurs 1 vers 2. Cela n'est pas facile à réaliser avec des décodeurs classiques, il faut leur ajouter une entrée supplémentaire CS (*Chip Select*). Le rôle de cette entrée est le suivant :

— quand CS=0, les sorties du décodeur restent à 0, quelles que soient les autres entrées ;

— quand CS=1, le décodeur se comporte comme un décodeur classique.

1) On représente de la manière indiquée ci-dessous un décodeur 1 vers 2 avec CS.

CS	e_0	s_1	s_0
⋮	⋮	⋮	⋮

Complétez sa table de vérité, et donnez un logigramme pour un décodeur 1 vers 2 avec CS.

2) On considère un décodeur 2 vers 4 classique, représenté ci-dessous : dressez sa table de vérité.

e_1	e_0	s_3	s_2	s_1	s_0
⋮	⋮	⋮	⋮	⋮	⋮

3) En utilisant deux décodeurs 1 vers 2 avec CS, ainsi qu'un décodeur 1 vers 2 classique, proposez un circuit réalisant un décodeurs 2 vers 4 classique. Justifiez brièvement votre réponse en vous basant sur la table de vérité de la question précédente.

Exercice 6 : Circuit décrémenteur

La décrémentation est un opération fréquente pour certains registres du processeur : on peut donc dans certains cas souhaiter disposer d'un circuit spécialisé. Ici, on va simplement construire un décrémenteur à propagation de retenue linéaire. On considère $x = (x_{n-1}, x_{n-2}, \ldots, x_0)_2$ le nombre binaire, et on note $s = (s_{n-1}, s_{n-2}, \ldots, s_0)_2$ le résultat $s = x - 1$. On note r_i la retenue qu'il faut éventuellement propager du rang i au rang $i+1$ ($r_i \in \{0, 1\}$).

1) Posez les soustraction binaires $(10101)_2 - 1$ et $(10100)_2 - 1$.

2) Dressez la table de vérité de s_0 et r_0 en fonction de x_0. En déduire les expressions de s_0 et de r_0 en fonction de x_0

3) Dressez la table de vérité de s_i et r_i en fonction de x_i et de r_{i-1}. En déduire les expressions de s_i et r_i en fonction de x_i et de r_{i-1}.

4) Dans quel cas la retenue la plus à gauche r_n est non nulle ?

5) Représentez le circuit d'un décrémenteur 4 bits.

Exercice 7 : Petits circuits combinatoires

1) Construire un circuit combinatoire à trois entrées x_0, x_1, x_2 capable de détecter si le nombre $(x_2 x_1 x_0)_2$ est divisible par 3.

2) Le but est de construire un circuit combinatoire à base de portes NOT, AND et OR pour la porte logique XOR à trois entrées a, b, c : commencez par écrire la table de vérité, puis proposez un circuit en vous basant sur la forme normale disjonctive. La porte XOR à trois entrées réalise la fonction dite d'imparité : pourquoi ce nom ?

Exercice 8 : Encodeur octal

Le but est de concevoir un encodeur octal : il s'agit d'un circuit à 8 entrées $e_0, \ldots, e_7$ et 3 sorties s_0, s_1, s_2. Si l'entrée e_i est à 1 et que toutes les autres sont à 0, on veut que les sorties soient telles que $(s_2 s_1 s_0)_2 = i$.

1) Complétez un tableau de la forme suivante, en supposant qu'au plus l'une des entrées e_i peut être activée à la fois (dans les colonnes $e_0, \ldots, e_7$, n'indiquez que les 1, pas les 0).

e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	s_2	s_1	s_0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

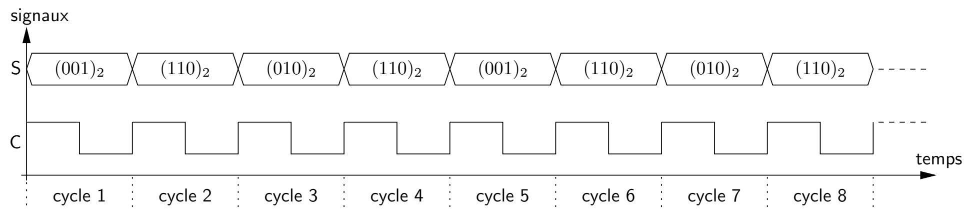
2) Exprimez les sorties s_2, s_1 et s_0 chacune sous la forme d'une somme de quatre littéraux.

3) Représentez le circuit logique demandé (vous pouvez utiliser des portes OR à 4 entrées).

Thème 4
Circuits séquentiels

Exercice 1 : Un générateur de séquence simple

On souhaite mettre au point un circuit séquentiel permettant de générer la séquence de valeurs $(001)_2$, $(110)_2$, $(010)_2$, $(110)_2$, et la répéter de façon périodique. On note $S = (s_2, s_1, s_0)$ la sortie du circuit. Un chronogramme représentant l'évolution de la valeur de S au cours du temps sera par exemple le suivant :



Pour concevoir ce circuit séquentiel, on va le modéliser à l'aide d'un automate fini séquentiel.

1) Quatre états sont suffisants pour réaliser l'automate fini demandé : pourquoi ?

Comme quatre états sont suffisants, on utilise un registre (à base de bascules flip-flops) à 2 bits pour le stockage de l'état courant $Q = (q_1, q_0)$ du circuit séquentiel. On choisit la correspondance suivante entre chacun des états de l'automate et sa sortie.

état Q	sortie S
00	001
01	110
10	010
11	110

1) Donnez une représentation graphique de l'automate fini demandé (graphique comportant les états et les transitions de l'automate).

2) Soit F la fonction de transition de l'automate : donnez la table de vérité de $F(Q)$ en fonction de Q. Complétez pour cela le tableau de vérité suivant :

Q		$F(Q)$	
q_1	q_0	f_1	f_0
⋮	⋮	⋮	⋮

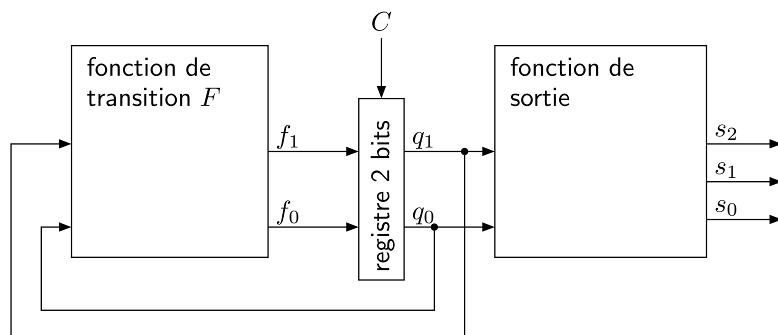
3) Exprimez f_1 et f_0 en fonction de q_1 et de q_0 à l'aide de formules booléennes simples.

4) Complétez la table de vérité de s_2, s_1 et s_0 en fonction de q_1 et q_0.

Q		$S(Q)$		
q_1	q_0	s_2	s_1	s_0
⋮	⋮	⋮	⋮	⋮

5) Exprimez s_2, s_1 et s_0 en fonction de q_1 et q_0 à l'aide de formules booléennes simples.

6) Complétez le circuit ci-dessous, afin d'obtenir le circuit séquentiel demandé.



7) Sur un chronogramme, représenter l'évolution de Q, $F(Q)$ et $S(Q)$ sur 8 cycles d'horloge, en supposant que l'état initial est $Q = (0,0)$.

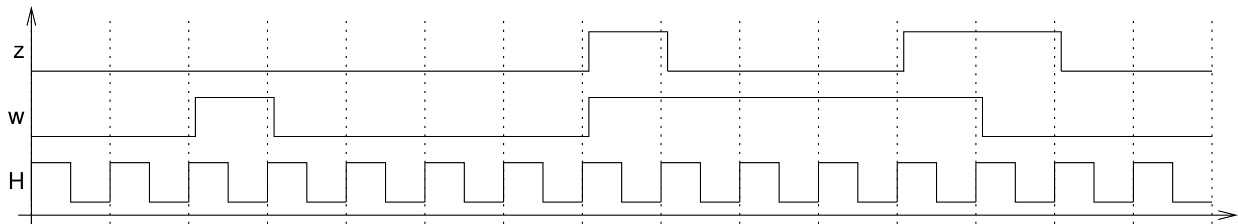
Exercice 2 : Détection de séquences spécifiques

On souhaite implanter un circuit séquentiel qui reconnaît deux séquences spécifiques quand celles-ci sont reçues à l'entrée w. Ces séquences consistent en quatre 1 consécutifs ou quatre 0 consécutifs. Le circuit possède une sortie z sur laquelle il génère le signal suivant :

— z prend la valeur 1 sur un cycle si lors des quatre cycles précédents w a conservé la valeur 1, ou a conservé la valeur 0 ;

— sinon, z prend la valeur 0.

Le chronogramme suivant illustre le fonctionnement du circuit (H désigne le signal d'horloge) :



Partie 1 : Avec un automate séquentiel classique

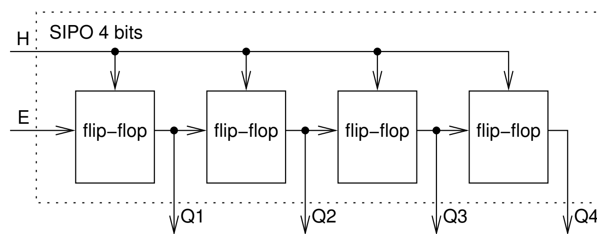
1) Donnez un automate fini séquentiel permettant de modéliser le circuit demandé. Vous choisirez un codage (simple et logique) des états de l'automate, et vous indiquerez la valeur de la sortie z dans chacun des états de l'automate. Établissez la table de transition de l'automate en encodant les états en binaire, puis la table de vérité pour la sortie z en fonction de l'état courant.

2) Exprimez la fonction de sortie, c'est à dire z en fonction de l'état courant (q_3, q_2, q_1, q_0), par une formule booléenne.

3) Combien de termes comportent les formes normales disjonctives pour chaque fonction partielle de la fonction de transition $F : (q_3, q_2, q_1, q_0) \mapsto (F_3, F_2, F_1, F_0)$? Est-ce la bonne façon de concevoir notre circuit ?

Partie 2 : Avec des registres à décalage

Au lieu d'implanter le circuit considéré sous la forme d'un automate, on va essayer d'utiliser un registre à décalage.



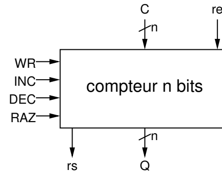
Un registre à décalage est un registre de taille fixe, fabriqué à l'aide de bascules flip-flops, dans lequel les bits sont décalés à fin de cycle de l'horloge H. On considère un registre de type SIPO (serial in, paralel out) : l'entrée est en série et la sortie est en parallèle.

Prenons l'exemple d'une suite de 4 bits, 1101. Le fait que l'entrée soit en série signifie qu'on utilise un seul fil pour insérer l'entrée dans le registre : on envoie 1, suivi de 0, puis de 1, et encore de 1 sur l'entrée E. La sortie en parallèle permet de récupérer plusieurs bits en même temps. Donc, pour cet exemple après 4 périodes de l'horloge du circuit on a 4 fils qui renvoient chacun un bit : $Q_1 = 1$, $Q_2 = 1$, $Q_3 = 0$, $Q_4 = 1$. Il peut être intéressant de dessiner un chronogramme pour illustrer cela.

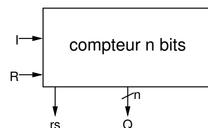
1) Proposez un circuit utilisant un registre à décalage pour détecter une suite de quatre 1 ou de quatre 0, comme dans la première partie de l'exercice. Vous vérifierez le bon fonctionnement de votre circuit à l'aide d'un chronogramme, en prenant pour séquence d'entrée : 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0 (c'est le signal d'entrée w du chronogramme de la partie 1).

Exercice 3 : Circuit compteur

Un compteur se présente, vu de l'extérieur, comme un registre de mémorisation particulier auquel on adjoint, en plus de la fonction d'écriture, les fonctions d'incrémentation, de décrémentation et de mise à zéro. Ces fonctions sont mises en œuvre par les signaux correspondants : INC, DEC, RAZ, ECR. Lorsqu'aucun de ces signaux n'est activé, le compteur reste à sa valeur courante. Le code mémorisé par un compteur est celui d'un nombre entier naturel, et l'incrémentation le remplace par le nombre suivant, la décrémentation par le précédent.



Pour simplifier, on n'étudiera que l'incrémentation et la mise à zéro. On appelle Q le compte courant (code sortant) de taille 8 bits (compteur modulo 2^8), R le signal de mise à 0, et I le signal d'incrémentation. On suppose que R est prioritaire sur I. Un signal de sortie supplémentaire rs devra indiquer si Q est maximum ($2^8 - 1$), et l'incrémentation dans ce cas remettra le compte à zéro (compteur modulo 2^8).

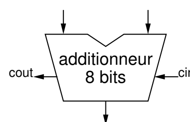


1) Interprétez en termes d'actions (incrémenter, mettre à zéro, laisser tel quel) les combinaisons de valeurs des signaux R et I. Caractérisez chaque action par une expression logique en R et I, vraie seulement pour cette action. Formalisez chaque action par l'expression algorithmique de l'évolution de Q que cette action doit provoquer.

2) Le vecteur de 8 signaux Q code un entier naturel. Sa mémorisation nécessite un registre : sortie Q, entrée D, commande d'écriture W, horloge C. Exprimez W en fonction de R et I.

3) Supposons que R ou I est activé. Donnez une expression algorithmique de N(D) en fonction de Q et de R, comme une fonction de sélection.

4) En plus du registre 8 bits déjà décrit, on dispose d'un multiplexeur 2 × 8 vers 8, dont le signal de sélection est s. On dispose également d'un additionneur 8 bits, avec retenue entrant cin (pour *carry in*) et retenue sortante cout (pour *carry out*).



Dans le circuit séquentiel demandé, on suppose que l'additionneur opère de tout façon sur Q à chaque cycle d'horloge. Quelles entrées doivent être données à l'additionneur, et que doit-on faire de sa sortie. Précisez également comment vous comptez utiliser le multiplexeur ; donnez le code de sélection que vous choisissez et exprimez s en fonction de R et I.

5) Proposez un circuit séquentiel implantant le compteur demandé, en laissant pour l'instant de côté le signal de sortie rs.

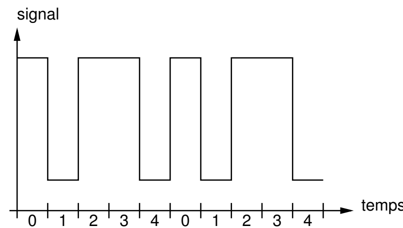
6) Complétez votre circuit en prenant en compte la sortie rs. Ajouter également une entrée re, telle que le fonctionnement du circuit reste inchangé, si ce n'est que le compteur est maintenant incrémenté uniquement si $\overline{R}$·I·re.

7) Proposez un chronogramme pour illustrer le fonctionnement du compteur de la question précédente. Vous supposerez que l'entrée re est maintenue à 1, que R est maintenue à 0, et que I est activé à chaque cycle. Au « cycle 0 » du chronogramme, vous supposerez que N(Q)=253, et vous compléterez le chronogramme sur 7 cycles : représentez en particulier les valeurs de N(Q) et de rs.

8) Comment fabriquer un compteur 16 bits à partir de deux compteurs 8 bits ? On appelle Q0 les 8 bits de poids faible du compteur, et Q1 les 8 bits de poids forts ; on appelle ri la retenue qui se propage d'un compteur 8 bits à l'autre. Illustrer le fonctionnement du compteur 16 bits à l'aide d'un chronogramme et des valeurs initiales des compteurs bien choisies : représentez en particulier les valeurs de N(Q0), ri, N(Q1) et N(Q).

Exercice 4 : Génération d'un signal périodique

Un circuit séquentiel est dépourvu d'entrée, et possède une sortie s sur laquelle il génère le signal périodique suivant : la valeur 1 est émise pendant une période de l'horloge du circuit, puis 0 est produite pendant la période suivante. Ensuite, le signal passe à 1 pendant deux périodes de l'horloge et à nouveau à 0 pendant la période suivante. Enfin, le signal est répété à partir de la première étape pour commencer un nouveau cycle.



1) Déterminez le nombre d'états nécessaires pour concevoir un automate réalisant le circuit séquentiel décrit. Donnez une correspondance entre les états de l'automate, et la sortie s du circuit. Finalement, établir la table de transition de l'automate.

2) Exprimez la fonction de transition sous forme de formules booléennes (d_2, d_2 et d_0 en fonction de q_0, q_1 et q_2), ainsi que la fonction de sortie (s en fonction de q_0, q_1 et q_2)

3) Proposez un circuit séquentiel implantant l'automate mis au point.

4) La période du signal généré devant être de 50 ns, quelle doit être la fréquence de l'horloge du circuit construit ci-dessus ?

Exercice 5 : Un contrôleur

L'objet de cet exercice est d'implémenter un contrôleur basique de lavage de voiture [1]. Ce contrôleur effectue les actions suivantes :

— Il attend l'insertion d'une pièce (événement d'entrée $coin$)

— Il met à 0 (sortie $clear$) un timer et commence à envoyer de l'eau (sortie $water$)

— Quand le timer a atteint son max (entrée $stop$), il recommence.

1) Coder cet automate avec deux états ($attente$ et $lavage$). On n'oubliera pas de mettre les sorties (sur les états) et les entrées correspondantes (sur les transitions).

2) Quelle est la taille (en bits) du registre stockant l'état courant ? Justifier.

3) Écrire la table de transition. On précisera bien les entrées et les sorties, et on écrira un tiret lorsque la valeur d'une entrée ou d'une sortie importe peu.

4) Dessiner le circuit.

Exercice 6 : Addition séquentielle

On veut mettre au point un circuit séquentiel pour d'effectuer une addition binaire en séquentiel ; en n cycles d'horloge l'addition $(s_{n-1} \dots s_1 s_0)_2 = (x_{n-1} \dots x_1 x_0)_2 + (y_{n-1} \dots y_1 y_0)_2$. Au cycle i $(n-1 \geq i \geq 0)$:

— l'état courant du circuit est donné par la valeur de la retenue c_i (on suppose $c_0 = 0$) ;

— $s_i = x_i + y_i + c_i \mod 2$;

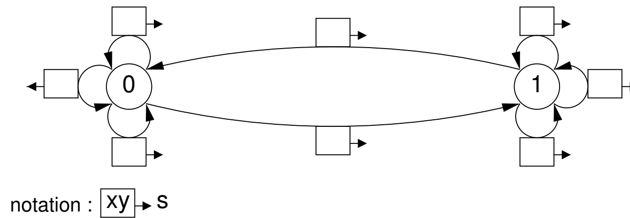
— la nouvelle valeur de la retenue à propager c_{i+1} est telle que : $x_i + y_i + c_i = 2 \times c_{i+1} + s_i$.

Dans la suite, on s'intéresse essentiellement à ce qui se passe à un cycle i quelconque, donc on oublie les indices : c désigne l'état courant du circuit, c^+ l'état suivant, x et y sont les deux bits à additionner (entrées), s est le résultat de l'addition au rang i (sortie).

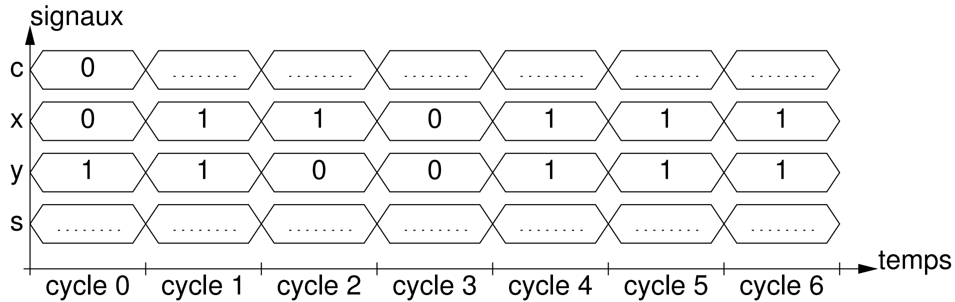
La fonction $(x, y, c) \mapsto c^+$ est la fonction de transition du circuit, et la fonction $(x, y, c) \mapsto s$ est la fonction de sortie.

1) Complétez la représentation graphique de l'automate fini séquentiel pour le circuit séquentiel demandé, en indiquant les valeurs prises par la fonction de sortie.

1. adapté de `http://ece224web.groups.et.byu.net/`



2) Complétez le chronogramme suivant.



3) Complétez le tableau suivant :

c	x	y	s	c⁺
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

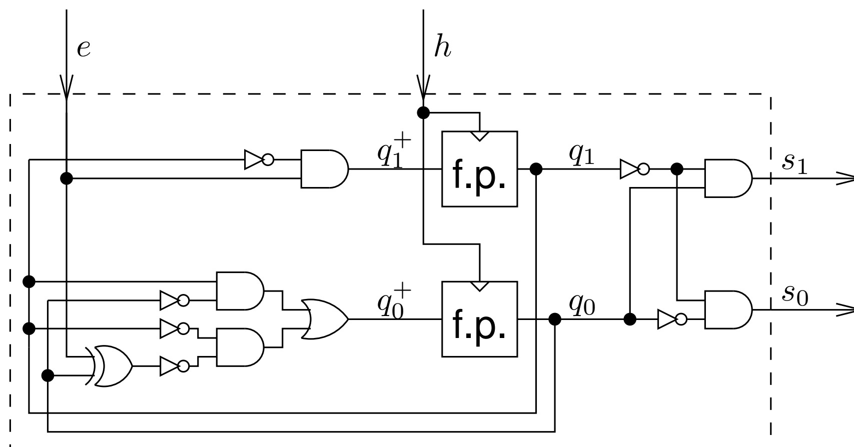
4) Montrez (en justifiant) que $s = x \oplus y \oplus c$.

5) En partant de la forme norme disjonctive pour s, montrez (en justifiant) que $c^+ = (x \oplus y)c + xy$.

6) Déduisez des questions précédentes un circuit implantant un additionneur séquentiel.

Exercice 7 : Analyse d'un circuit séquentiel

On considère le circuit séquentiel ci-dessous. Ce circuit reçoit un signal d'horloge h, prend une entrée e, et produit une sortie $s = (s_1 s_0)$. Il comporte 2 bascules flip-flops, régies par le front montant de l'horloge h, qui stockent l'état courant $(q_1 q_0)$ du circuit.



La fonction de transition du circuit est la fonction f telle que $(q_1^+, q_0^+) = f(q_1, q_0, e)$. La fonction de sortie g est telle que $s = g(q_1, q_0, e)$.

1) Exprimez q_1^+ et q_0^+ en fonction de q_1, q_0 et e par des expressions booléennes.

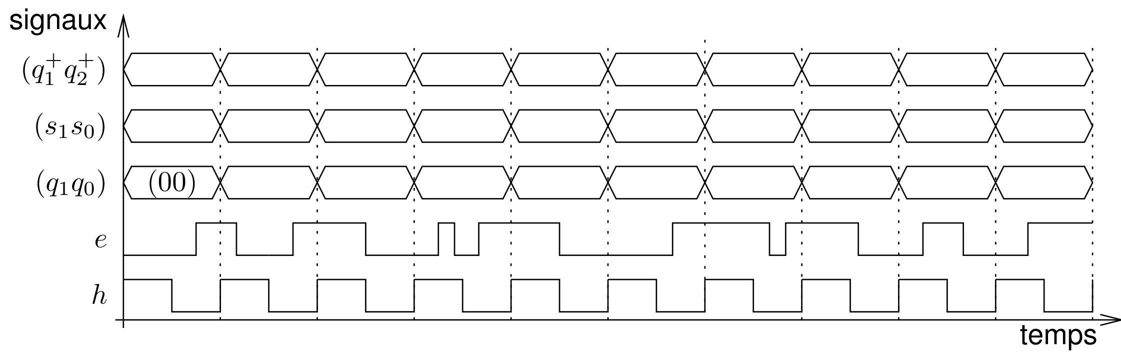
2) Exprimez s_1 et s_2 en fonction de q_1 et q_0 par des expressions booléennes.

3) Complétez la table de vérité suivante.

q_1	q_0	e	q_1^+	q_0^+	s_1	s_0
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

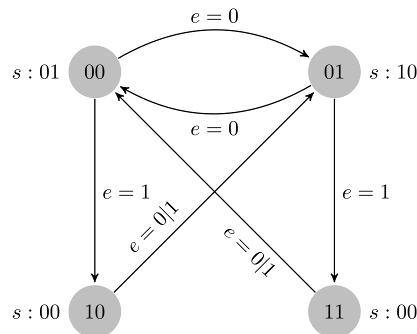
4) Représentez le comportement du circuit sous la forme d'un automate fini séquentiel. Indiquez bien sur chaque flèche indiquant une transition la valeur de e provoquant cette transition, ainsi que la valeur de la sortie s pour chaque état.

5) Complétez le chronogramme suivant.

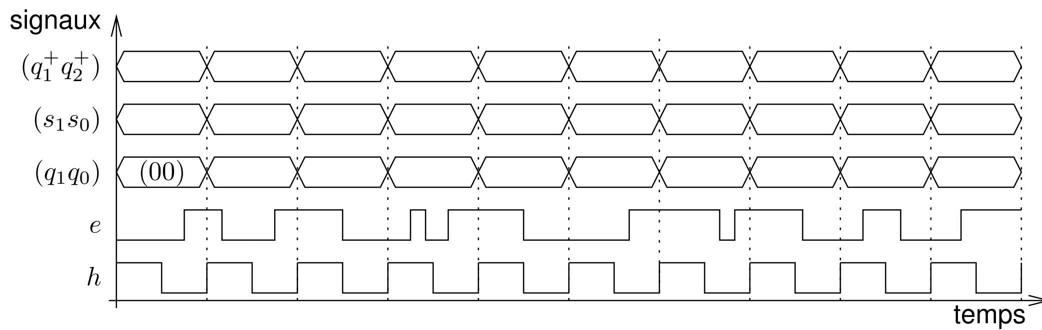


Exercice 8 : Génération d'un circuit séquentiel

On considère l'automate fini séquentiel ci-dessous. Le circuit séquentiel correspondant reçoit un signal d'horloge h, prend une entrée e, et produit une sortie $s = (s_1 s_0)$. Il comporte 2 bascules flip-flops, régies par le front montant de l'horloge h, qui stockent l'état courant $(q_1 q_0)$ du circuit. La fonction de transition du circuit est la fonction f telle que $(q_1^+, q_0^+) = f(q_1, q_0, e)$. La fonction de sortie g est telle que $s = g(q_1, q_0, e)$.



1) Complétez le chronogramme suivant.



2) Complétez la table de vérité suivante.

q_1	q_0	e	q_1^+	q_0^+	s_1	s_0
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

3) Exprimez q_1^+ et q_0^+ en fonction de q_1, q_0 et e par des expressions booléennes.

4) Exprimez s_1 et s_2 en fonction de q_1 et q_0 par des expressions booléennes.

5) Donnez un circuit séquentiel implantant l'automate fini étudié.

Thème 5
Programmation en assembleur

Tous les exercices de ce TD sont à faire en utilisant le langage d'assemblage du LC3.

Description du LC-3

La mémoire et les registres : La mémoire du LC-3 est organisée par mots de 16 bits, avec un adressage également de 16 bits (adresses de $(0000)_H$ à $(FFFF)_H$).

Le LC-3 comporte 8 registres généraux 16 bits : R0, ..., R7. R6 est réservé pour la gestion de la pile d'exécution, et R7 pour stocker l'adresse de retour des routines. Il comporte aussi des registres spécifiques 16 bits : PC (*Program Counter*), IR (*Instruction Register*), PSR (*Program Status Register*) qui regroupe plusieurs drapeaux.

Le PSR contient trois bits N, Z, P, indiquant si la dernière valeur (regardée comme le code d'un entier naturel en complément à 2 sur 16 bits) placée dans l'un des registres, R0, ..., R7 est négative strictement pour N, nulle pour Z, ou positive strictement pour P.

Les instructions :

syntaxe	action	NZP
NOT DR,SR	DR <- not SR	*
ADD DR,SR1,SR2	DR <- SR1 + SR2	*
ADD DR,SR1,Imm5	DR <- SR1 + SEXT(Imm5)	*
AND DR,SR1,SR2	DR <- SR1 and SR2	*
AND DR,SR1,Imm5	DR <- SR1 and SEXT(Imm5)	*
LEA DR,label	DR <- PC + SEXT(PCoffset9)	*
LD DR,label	DR <- mem[PC + SEXT(PCoffset9)]	*
ST SR,label	mem[PC + SEXT(PCoffset9)] <- SR	
LDR DR,BaseR,Offset6	DR <- mem[BaseR + SEXT(Offset6)]	*
STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] <- SR	
BR[n][z][p] label	Si (cond) PC <- PC + SEXT(PCoffset9)	
NOP	No Operation	
RET	PC <- R7	
JSR label	R7 <- PC; PC <- PC + SEXT(PCoffset11)	

Directives d'assemblage :

`.ORIG adresse`	Spécifie l'adresse à laquelle doit commencer le bloc d'instructions qui suit.
`.END`	Termine un bloc d'instructions.
`.FILL valeur`	Réserve un mot de 16 bits et le remplit avec la valeur constante donnée en paramètre.
`.BLKW nombre`	Cette directive réserve le nombre de mots de 16 bits passé en paramètre.
`;`	Les commentaires commencent par un point-virgule.

Les interruptions prédéfinies : `TRAP` permet de mettre en place des *appels système*, chacun identifié par une constante sur 8 bits, gérés par le système d'exploitation du LC-3. On peut les appeler à l'aide des macros indiquées ci-dessous.

instruction	macro	description
`TRAP x00`	`HALT`	termine un programme (rend la main à l'OS)
`TRAP x20`	`GETC`	lit au clavier un caractère ASCII et le place dans R0
`TRAP x21`	`OUT`	écrit à l'écran le caractère ASCII placé dans R0
`TRAP x22`	`PUTS`	écrit à l'écran la chaîne de caractères pointée par R0
`TRAP x23`	`IN`	lit au clavier un caractère ASCII, l'écrit à l'écran, et le place dans R0

Constantes : Les constantes entières écrites en hexadécimal sont précédées d'un `x`, en décimal elles sont précédées d'un `#` (qui peut être implicite) ; elles peuvent apparaître comme paramètre : des instructions du LC3 (opérandes immédiats, attention à la taille des paramètres), des directives `.ORIG`, `.FILL` et `.BLKW`.

Codage des instructions LC3

On donne ici un tableau récapitulatif du codage des instructions LC3.

syntaxe	action	NZP	codage															
			opcode				arguments											
			F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
NOT DR,SR	DR ← not SR	*	1 0 0 1				DR			SR			1 1 1 1 1 1					
ADD DR,SR1,SR2	DR ← SR1 + SR2	*	0 0 0 1				DR			SR1			0	0 0		SR2		
ADD DR,SR1,Imm5	DR ← SR1 + SEXT(Imm5)	*	0 0 0 1				DR			SR1			1	Imm5				
AND DR,SR1,SR2	DR ← SR1 and SR2	*	0 1 0 1				DR			SR1			0	0 0		SR2		
AND DR,SR1,Imm5	DR ← SR1 and SEXT(Imm5)	*	0 1 0 1				DR			SR1			1	Imm5				
LEA DR,label	DR ← PC + SEXT(PCoffset9)	*	1 1 1 0				DR			PCoffset9								
LD DR,label	DR ← mem[PC + SEXT(PCoffset9)]	*	0 0 1 0				DR			PCoffset9								
ST SR,label	mem[PC + SEXT(PCoffset9)] ← SR		0 0 1 1				SR			PCoffset9								
LDR DR,BaseR,Offset6	DR ← mem[BaseR + SEXT(Offset6)]	*	0 1 1 0				DR			BaseR			Offset6					
STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] ← SR		0 1 1 1				SR			BaseR			Offset6					
BR[n][z][p] label	Si (cond) PC ← PC + SEXT(PCoffset9)		0 0 0 0				n	z	p	PCoffset9								
NOP	No Operation		0 0 0 0				0	0	0	0 0 0 0 0 0 0 0 0								
RET (JMP R7)	PC ← R7		1 1 0 0				0 0 0			1 1 1			0 0 0 0 0 0					
JSR label	R7 ← PC ; PC ← PC + SEXT(PCoffset11)		0 1 0 0				1	PCoffset11										

Traduction de programmes en langage d'assemblage

Il vous est demandé de toujours commencer par écrire un pseudo-code pour le programme ou la routine demandé, en faisant apparaître les registres que vous allez utiliser pour effectuer vos calculs, et en ajoutant tous les commentaires utiles. Vous traduirez ensuite votre pseudo-code vers le langage d'assemblage du LC3 en utilisant les règles de traduction suivantes.

Traduction d'un « bloc if » : On suppose que la condition d'entrée dans le bloc consiste simplement en la comparaison du résultat d'une expression arithmétique `e` à 0. Dans ce qui suit, `cmp` désigne une relation de comparaison : <, ≤, = , ≠, ≥ , >. On note `!cmp` la relation contraire de la relation `cmp`, traduite dans la syntaxe des bits `nzp` de l'instruction `BR`. Si par exemple `cmp` est <, alors `BR!cmp` désigne `BRpz` (pour « positive or zero »).

```
/* En pseudo-code */
if e cmp 0 {
  corps du bloc
}
```

```
; En langage d'assemblage du LC3
     evaluation de e
     BR!cmp endif    ; branchement sur la sortie du bloc
     corps du bloc
endif:
```

Traduction d'un bloc « if-else » :

```
/* En peudo-code */
if e cmp 0 {
  corps du bloc 1
}
else {
  corps du bloc 2
}
```

```
; En langage d'assemblage du LC3
     evaluation de e
     BR!cmp else    ; branchement sur le bloc else
     corps du bloc 1
     BR endif       ; branchement sur la sortie du bloc
else:
     corps du bloc 2
endif:
```

Traduction d'une « boucle while » :

```
/* En peudo-code */
while e cmp 0 {
  corps de boucle
}
```

```
; En langage d'assemblage du LC3
loop:
     evaluation de e
     BR!cmp endloop  ; branchement sur la sortie de boucle
     corps de boucle
     BR loop         ; branchement inconditionnel
endloop:
```

Quelques « astuces » à connaître :

- Initialisation d'un registre à 0 : `AND Ri,Ri,#0`
- Initialisation d'un registre à une constante `n` (représentable en complément à 2 sur 5 bits) :

```
AND Ri,Ri,#0
ADD Ri,Ri,n
```

- Calcul de l'opposé d'un entier (on calcule le complément à 2 de Rj dans Ri) :

```
NOT Ri,Rj
ADD Ri,Ri,#1
```

- Multiplication par 2 de Rj, résultat dans Ri : `ADD Ri,Rj,Rj`
- Copie du contenu de Rj dans Ri : `ADD Ri,Rj,#0`

Exercice 1 : Autour de la sommation

1) On considère le programme incomplet ci-dessous. On souhaite pouvoir ajouter les entiers aux adresses add0, add1, add2, et placer le résultat à l'adresse `resultat`. Donnez un code assembleur pour cela.

```
           .ORIG x3000     ; adresse de début de programme
; partie dédiée au code
                           ; R1 sera utilisé comme accumulateur
           ### A COMPLETER ###

; partie dédiée au résultat et aux données
resultat:  .BLKW #1        ; espace pour stocker le résultat (résultat attendu ici : 73 soit x49)
add0:      .FILL #12
add1:      .FILL #45
add2:      .FILL #16
           .END
```

2) Supposons que R0 contient un entier positif x, et R1 un entier positif y. On souhaite affecter à R2 l'entier $x - y$: proposez un morceau de code pour cela.

3) On considère le programme incomplet ci-dessous. On souhaite pouvoir ajouter les entiers compris entre les adresses `debut` et `fin` à l'aide d'une boucle, et placer le résultat de la sommation à l'adresse `resultat` Il vous est demandé de commencer par donner un pseudo-code pour votre programme, en respectant les directives d'utilisation des registres données ci-dessous, avant de le traduire en langage d'assemblage.

```
           .ORIG x3000     ; adresse de début de programme
; partie dédiée au code
; R0 sera utilisé comme un pointeur pour une case du tableau
; R1 contiendra l'opposé de l'adresse fin, pour comparaison avec R0
; R2 servira à l'accumulation des entiers du tableau
; R3 servira de registre temporaire
           ### A COMPLETER ###
; partie dédiée au résultat et aux données
resultat:  .BLKW #1        ; espace pour stocker le résultat (résultat attendu ici : 157 soit x9D)
debut:     .FILL #12
           .FILL #45
           .FILL #16
           .FILL #06
fin:       .FILL #78
           .END
```

Exercice 2 : Ajouter 1 aux éléments d'un tableau

Il s'agit de compléter la routine add pour qu'elle permette d'ajouter 1 à chacune des cases mémoire dont les adresses sont comprises entre l'adresse contenu dans `R0` et celle contenue dans `R1` lors de l'appel. Il vous est demandé de commencer par donner un pseudo-code pour votre routine, en respectant les directives d'utilisation des registres données ci-dessous, avant de le traduire en langage d'assemblage.

```
           .ORIG x3000     ; adresse de début de programme
; partie dédiée au code
           LEA R0,debut    ; charge l'adresse du premier élément dans R1
           LEA R1,fin      ; charge l'adresse du dernier élément dans R2
           JSR add         ; appel la routine add
           HALT            ; termine le programme

; partie dédiée aux données
debut:     .FILL #1
           .FILL #3
           .FILL #5
           .FILL #7
fin:       .FILL #9

; Routine add, pour ajouter 1 au contenu des cases dont les adresses sont
; comprises entre l'adresse contenue dans R0 et celle contenue dans R1.
; On suppose que R0 <= R1
; paramètres d'entrée : R0 et R1, adresses de début et de fin du tableau
;                       R0 et R1 ne doivent pas être modifiés
; registre temporaire : R2 servira de pointeur pour parcourir les cases du tableau
;                       R3 registre dans lequel on effectura des calculs
;                       R4...R7 ne doivent pas être modifiés
         ### PARTIE A COMPLETER ###
         .END
```

Exercice 3 : Programme Mystère LC3

On [1] fournit un programme LC3 sous forme d'un bout de mémoire rempli en hexa. Il est demandé de décoder ce programme et de dire ce qu'il fait. On fera attention aux sauts de PC (le cas échéant, on rajoutera des *labels* à certaines adresses mémoires).

Adresse	Contenu	Contenu binaire	Détails des instructions	pseudo-code
x3000	x5020	0101 000 000 1 00000	AND (mode cst), DR=SR=R0, Imm5=x00	$R_0 \leftarrow R_0 \& 0 = 0$
x3001	x1221			
x3002	xE404			
x3003	x6681			
x3004	x1262			
x3005	x16FF			
x3006	x03FD			
x3007	xF025			
x3008	x006	donnée	-	

FIGURE 5.1 – Un programme en binaire/hexa

Exercice 4 : Dessin de carrés et de triangles

Le but de cet exercice est de programmer deux routines : l'une pour l'affichage d'un carré d'étoiles, l'autre pour l'affichage d'un triangle. Vous utiliserez pour cela « l'appel système » OUT qui permet d'afficher le caractère dont le code ASCII se trouve dans R0. Cet appel système se passe un peu comme un appel de fonction : il écrase le contenu du registre R7. On considère le programme à compléter ci-dessous.

```
          .ORIG x3000     ; adresse de début de programme
; partie dédiée au code
          LD R6,spinit    ; initialisation du pointeur de pile
          LD R1,taille    ; charge la taille du carre dans R1
          JSR carre       ; appel à la routine carre (dessine un carré)
          JSR trilo       ; appel à la routine trilo (dessine un triangle)
          HALT            ; termine le programme

; partie dédiée aux données
taille:   .FILL #5
etoile:   .FILL x2A       ; caractère '*'
cr:       .FILL x0D       ; caractère '\n'

; pile
spinit:  .FILL stackend
         .BLKW #5
```

1. Source : http://castle.eiu.edu/~mathcs/mat3670/index/index.html, avec l'aimable autorisation des auteurs

```
stackend: .BLKW #1      ; adresse du fond de la pile

; Routine carre, pour dessiner un carre de R1*R1 étoiles.
; paramètre d'entrée : R1, la taille du côté du carré.
carre:
         ; *** A COMPLETER ***

; Routine trilo, pour dessiner un triangle inférieur sur R1 lignes
; et R1 colonnes.
; paramètre d'entrée : R1, la taille du côté du triangle.
carre:
         ; *** A COMPLETER ***
         .END
```

1) Complétez la routine `carre` pour qu'elle affiche un carré d'étoiles dont le nombre de lignes est contenu dans R1. Vous traduirez pour cela le pseudo-code suivant :

```
R2 <- -R1
R3 <- 0
while( R3 < R1 ) { // (R3 < R1) <=> (R5 = R3-R1 = R3+R2 < 0)
  R4 <- 0
  while( R4 < R1 ) { // (R4 < R1) <=> (R5 = R4-R1 = R4+R2 < 0)
    print('*');
    R4++
  }
  print('\n');
  R3++;
}
```

Attention, il ne suffit pas de traduire ce code pour obtenir celui de la routine : il faut également gérer correctement l'adresse de retour.

2) Proposez une routine `trilo`, permettant d'afficher un triangle de la forme

```
*
**
***
```

sur le nombre de lignes contenu dans R1 (ici, R1=3). Il vous est demandé de modifier le pseudo-code de la question précédente avant de vous livrer à sa traduction.

Exercice 5 : Multiplication par 6 des entiers d'un tableau

On considère le programme à compléter ci-dessous.

```
          .ORIG x3000     ; adresse de début de programme
; partie dédiée au code
          LD R6,spinit    ; initialisation du pointeur de pile
          LEA R0,debut    ; charge l'adresse de début du tableau
          LEA R1,fin      ; charge l'adresse de fin du tableau
          JSR mul6tab     ; appel à une routine
          HALT            ; termine le programme

; partie dédiée aux données
mask:     .FILL x000F     ; constante x000F
debut:    .FILL 4         ; adresse de début de tableau
          .FILL 5
          .FILL 6
fin:      .FILL 3         ; adresse de fin de tableau

; pile
spinit:   .FILL stackend
         .BLKW #5
stackend: .BLKW #1      ; adresse du fond de la pile

; Routine mul5tab, pour multiplier les entiers d'un tableau par 6 modulo 16.
; Les multiplications se feront sur place.
; paramètres d'entrée   : R0, adresse de début du tableau
;                         R1, adresse de fin du tableau
mul6tab:
         : *** A COMPLETER ***
         .END
```

Le but est de compléter la routine `mul6tab` pour qu'elle multiplie les entiers d'un tableau par 6 modulo 16 : en entrée R0 contient l'adresse de la première case du tableau, R1 l'adresse de la dernière case. Vous traduirez pour cela le pseudo-algorithme suivant :

```
; R2 <- R0
; while( R2 <= R1 ) {  // (R2 <= R1) <=> (R2-R1 <= 0)
;   R3 <- mem[R2];
;   R3 <- 2*R3+4*R3;   // R3 <- 6*R3
;   R3 <- R3 & 0x000F; // R3 <- R3 modulo 16
;   mem[R2] <- R3;
;   R3++;
; }
}
```

1) En utilisant uniquement R4 comme registre intermédiaire, montrez comment traduire la ligne `R3 <- 2*R3+4*R3`.

2) A la ligne `R3 <- R3 & 0x000F`, le `&` désigne le AND bit-à-bit : justifier le fait que l'opération `R3 & 0x000F` calcule bien le reste dans la division euclidienne de R3 par 16. Comment traduirez-vous cette ligne en langage d'assemblage ?

3) Traduisez l'algorithme proposé. Vous pouvez utiliser R4 et/ou R5 pour les calculs intermédiaires.

Exercice 6 : Décompte de bits non-nuls

Écrire une fonction dans l'assembleur du LC3 afin de compter le nombre de bits non-nuls dans un entier. Votre programme principal doit appeler votre fonction afin de compter le nombre de bits non-nuls de l'entier à l'adresse `n`, et placer le résultat à l'adresse `r`. Vous compléterez le code ci-dessous, en le commentant.

```
         .ORIG x3000
; Programme principal
         ...
; Données
n:       .FILL #78
r:       .BLKW #1

; Sous-routine pour calculer le nombre de bit non-nuls dans R1
; paramètre : R1, inchangé
; retour : R0, le nombre de bits non nuls dans R1
; registres temporaires : R2, R3, R4
cmptbits: ...
.END
```

Exercice 7 : Palindromes

L'objet de l'exercice est d'écrire un programme LC3 [2] qui permet de déterminer si une chaîne donnée est un palindrome, *i.e.* son retournement est égal à lui-même. L'algorithme utilisé sera le suivant :

```
bool isPalindrome(char *string){
    int left,right ;
    left = 0 ;
    right = len(string) - 1 ;
    while (left < right){
        if (string[left] != string[right])
            return(false);
        left += 1;
        right -= 1;
    }
    return(true)
}
```

On suppose écrites les routines suivantes :

— `len` : l'adresse du premier caractère d'une chaîne étant stockée dans le registre R_1, cette routine calcule et place la taille de la chaîne dans le registre R_2.

2. adapté de `http://home.wlu.edu/~lambertk/classes/210/exercises/hw7.htm`

— `gets` : cette primitive réalise la lecture au clavier d'une chaîne de caractère de taille au maximum R_2, et la stocke dans un tableau commençant à l'adresse R_1.

La routine considérera que l'adresse de base de la chaîne est stockée dans R_1, que sa longueur est stockée dans R_2, et stockera le résultat dans le registre R_0 (1 pour vrai, 0 pour faux).

1) Expliquer sur des exemples/dessins comment fonctionne cet algorithme.

2) Remplir les trous du pseudo-code-LC3 suivant :

```
R2 <- R1 + R2 - 1;
 while(R2 > R1) {
   if(...............) {
     R0 <-  ................
     ...........
   }
   R1++;
   R2--;
  }
  R0 <-  .........
  return;
}
```

et commentez-le (dire en particulier ce que stockent les différents registres).

3) Écrire la routine en assembleur LC-3.

4) Écrire le reste du programme. On pourra utiliser la primitive `puts` du LC3. On bornera la taille des chaînes à 10 caractères.

Exercice 8 : Affichage et Récursivité

On considère le programme incomplet ci-dessous : il s'agit de compléter les sous-routines `affn0` et `aff0n` de manière à ce qu'elles affichent respectivement les chiffres décimaux de `n` à `0` et de `0` à `n` (On supposera toujours `n` ≥ 0). Le programme ci-dessous devra donc afficher :

```
76543210
01234567
```

Les deux routines doivent être récursives, et utiliser la pile mise en place dans le programme pour gérer la récursivité : la routine doit commencer par sauvegarder l'adresse de retour contenue dans `R7`, ainsi que le paramètre-donnée `R1`, et doit restaurer ces registres avant son retour. Vous devez commencer par écrire le pseudo-code de ces routines, avant de les traduire en langage d'assemblage.

```
  .ORIG x3000
; Programme principal
main:     LD R6,spinit  ; on initialise le pointeur de pile
          LD R1,n       ; on initialise R1 avant l'appel à la sous-routine affn0
          JSR affn0     ; appel à la sous-routine affn0
          LEA R0,strcr  ; charge l'adresse strcr dans R0 (chaine de caractères "\n")
          PUTS          ; affiche un retour-chariot
          JSR aff0n     ; appel à la sous-routine aff0n
          LEA R0,strcr  ; charge l'adresse strcr dans R0 (chaine de caractères "\n")
          PUTS          ; affiche un retour-chariot
          HALT

; routine affn0 : affiche les chiffres de n à 0
; paramètre-donnée  : R1(n)
; tous les registres sauf R1 et R6 peuvent être écrasés par l'appel
; ### A COMPLETER ###

; routine aff0n : affiche les chiffres de 0 à n
; paramètre-donnée  : R1(n)
; tous les registres sauf R1 et R6 peuvent être écrasés par l'appel.
; ### A COMPLETER ###

; Données
n:        .FILL #7
carzero:  .FILL #48     ; code ASCII de '0'
```

```
strcr:    .STRINGZ "\n" ; pour afficher un retour chariot
spinit:   .FILL stackend
          .BLKW #32
stackend: .BLKW #1      ; adresse du fond de la pile
          .END
```

Exercice 9 : Nombre de chiffres dans une chaîne de caractères

Le but est d'écrire une routine pour compter le nombre de chiffres présents dans une chaîne de caractères se terminant par un `'\0'`. Par exemple, le nombre de chiffres dans la chaîne `"2 fois 3 fait 6"` est 3. On rappelle que le code ASCII de `\0` est 0, celui de `'0'` est 48, et celui de `'9'` est 57. Pour la routine `nbchiffres` en question, il vous est demandé de traduire l'algorithme suivant :

```
paramètres d'entrée   : R0 contient l'adresse du début de la chaîne
paramètre de sortie   : R1 le nombre de chiffres comptés
registres temporaires : R2, R3, R4
R1 <- 0
R2 <- R0
R3 <- mem[R2];
while(R3 != '\0') {
  if('0' <= R3) {
    if(R3 <= '9') R1++;
  }
  R2++;
  R3 <- mem[R2];
}
```

Complétez le programme ci-dessous dans le langage d'assemblage du LC3. Vous devez respecter les commentaires présents dans le code, et ajouter vos propres commentaires. N'oubliez pas de compléter le programme principal.

```
 .ORIG x3000
; Programme principal
         .................... ; charge l'adresse ch dans R0
         .................... ; appel à la routine nbchiffres
         .................... ; range le résultat à l'adresse r
         HALT                 ; rend la main au système d'exploitation

; Partie dédiée aux données
ch:      .STRINGZ "2 fois 3 fait 6"
r:       .BLKW 1              ; réserve une case pour le résultat
zero:    .FILL 48             ; code ASCII de '0'
neuf:    .FILL 57             ; code ASCII de '9'

; Routine pour compter le nombre de chiffres dans une chaîne terminée par un '\0'.
nbchiffres:
;######## à compléter ########
         .END
```

Exercice 10 : Renversement d'une chaîne de caractères grâce à la pile

Le but est d'écrire en langage d'assemblage du LC3 une routine permettant de renverser une chaîne de caractères terminée par le caractère `'\0'` (dont le code ASCII est l'entier 0), *via* la pile. Par exemple, le renversé de la chaîne `saper` et la chaîne `repas`. La pile utilisée sera la pile d'exécution manipulée à l'aide du registre `R6`. Le principe est le suivant :

- initialisation : on empile un `'\0'`, qui permettra de détecter qu'il faudra arrêter de dépiler ;
- boucle d'empilement : tant que l'on n'a pas rencontré le `'\0'` de la chaîne initiale (à renverser), on empile un a un ses caractères.

— boucle de dépilement : tant que l'on n'a pas rencontré le '\0' dans la pile, on dépile un a un les caractères de la pile, et on les range dans le chaîne de destination (renversée).

— on n'oublie pas d'ajouter un '\0' final à la chaîne de destination.

Vous n'oublierez pas, au début de la routine, d'empiler l'adresse, et de la dépiler à la fin de la routine.

1) Donner le pseudo-code pour la boucle d'empilement telle qu'expliquée ci-dessus. Vous utiliserez R2 comme pointeur pour parcourir la chaine initiale, en supposant que l'adresse de son premier caractère est contenue dans R0. Vous utiliserez R3 pour le stockage temporaire des caractères.

2) Donner le pseudo-code pour la boucle de dépilement telle qu'expliquée ci-dessus. Vous utiliserez R2 comme pointeur pour parcourir la chaine, en supposant que l'adresse de son premier caractère est contenue dans R1. Vous utiliserez R3 pour le stockage temporaire des caractères.

3) Complétez maintenant le code ci-dessous (programme principal et code de la routine rev) en vous conformant aux commentaires.

```
; programme principal
main:     LD R6,spinit  ; on initialise le pointeur de pile
          ............. ; on charge l'adresse effective du label chaine dans R0
          ............. ; on charge l'adresse effective du label chrev dans R1
          ............. ; appel à la sous-routine rev
          ............. ; on charge l'adresse effective du label chrev dans R0
          PUTS          ; on affiche la chaîne de caractères résultat à l'écran
          HALT          ; rend la main au système d'exploitation
chaine:    .STRINGZ "saper"
chrev:     .BLKW #6     ; espace réservé pour la chaîne renversée

; routine rev, pour renverser une chaîne de caractères
; paramètre d'entrée    : R0 (adresse du premier caractère de la chaîne initiale)
;                         R1 (adresse de début de la chaîne de destination)
;                         R0 et R1 doivent être préservés par l'appel de la routine
; registres temporaires : R2 et R3
rev:
; empilement de l'adresse de retour
          ............. ; ......................................
          ............. ; ......................................
; initialisation de la pile pour le renversement d'une chaîne
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
; boucle d'empilement des caractères de la chaîne initiale
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
; boucle de dépilement des caractères vers la chaîne de destination
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
; on n'oublie pas d'ajouter un '\0' à la chaîne de destination
```

```
          ............. ; ......................................
          ............. ; ......................................
; retour au programme appelant
          ............. ; ......................................
          ............. ; ......................................
          ............. ; ......................................
; fin de la routine rev
```

Thème 6

Mémoire cache et pipeline

Exercice 1 : Mémoire cache

Un ordinateur dispose d'une mémoire centrale de 64 kio : les adresses sont codées sur 16 bits, et la taille de chaque case mémoire est de 1 o. Entre le processeur et la mémoire centrale est placée une petite mémoire cache directe, composée de 8 entrées pouvant chacune contenir 32 o. On décompose les adresses en trois champs :

INDICATEUR								ENTREE			OCTET				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

On précise que : la concaténation des champs INDICATEUR et ENTREE donne en binaire l'indice d'une ligne de cache ; le champ ENTREE donne en binaire l'indice d'une entrée dans la mémoire cache ; le champ OCTET donne l'indice d'un octet dans une ligne de cache (ou un entrée dans la mémoire cache).

1) Pourquoi le champ OCTET comporte-t-il 5 bits, et le champ ENTREE 3 bits ?

2) En combien de lignes de cache se décompose la mémoire centrale ?

3) Complétez le tableau suivant ; notez les numéros de lignes en indice de la lettre ℓ, et les numéros d'entrées en indice de la lettre e.

adresse (hexa.)	adresse (binaire)																ligne (décimal)	entrée (décimal)
$0BBF_H$																		
$0BC0_H$																		
$0BC1_H$																		
$05AE_H$																		
$05BF_H$																		
$05C0_H$																		

4) Le processeur effectue successivement des accès aux adresses $0BBF_H$, $0BC0_H$, $0BC1_H$, $05AE_H$, $05BF_H$, $05C0_H$ (à celles-là seulement). On suppose le cache initialement vide. Indiquez quelle est la ligne contenue dans chacune des entrées du cache après chaque accès, et signalez chaque défaut de cache (par une croix dans la dernière colonne).

adresse	entrées								défaut
	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	
$0BBF_H$									
$0BC0_H$									
$0BC1_H$									
$05AE_H$									
$05BF_H$									
$05C0_H$									

Exercice 2 : Somme de deux vecteurs

On considère le programme C suivant :

```
int main(void) {
  int A[16], B[16], C[16];
  int i;
  for(i=0; i<16; i++) C[i] = A[i] + B[i];
  ...
}
```

On suppose que l'on compile et que l'on exécute ce programme sur un ordinateur qui ne dispose que d'un seul niveau de cache de données direct : ce cache comporte 4 entrées de 32 octets (on ne se préoccupe pas du chargement des instructions, ni de la variable `i` dans cet exercice). On note les lignes de cache $\ell_0, \ell_1, \ell_2, \ldots$, et les entrées du cache e_0, e_1, e_2, e_3. Les entiers de type `int` sont d'une taille de 32 bits. Les tableaux sont stockés de manière contiguë en mémoire, dans l'ordre de leur déclaration.

1) Combien d'entiers de type `int` peut contenir une ligne de cache ?

2) Combien de lignes de caches consécutives occupe chacun des tableaux `A`, `B` et `C` ?

3) En supposant que `A[0]` est aligné sur le début de la ligne de cache ℓ_0, indiquez sur un schéma à quelles lignes de cache appartiennent les élément de `A`, `B` et `C`. Indiquez également sur ce même schéma dans quelles entrées du cache seront rangés ces éléments lorsqu'ils seront accédés.

4) Dans un tableau de la forme indiquée ci-dessous, indiquez quelle ligne de cache contient chaque entrée du cache à la fin de chaque itération de la boucle `for(i=0; i<16; i++) C[i] = A[i] + B[i];` (Vous placerez un « ? » quand on ne peut pas savoir). Indiquez également le nombre de *cache miss* qui ont eu lieu au cours de l'itération.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
e_0																
e_1																
e_2																
e_3																
cache miss																

Exercice 3 : Somme des éléments d'une matrice

On considère le programme C suivant :

```
#define n 16

int main(void) {
  int i, j;
  double A[n][n];      /* A va permettre de stocker une matrice de doubles */
  double s;

  for(i=0; i<n; i++) for(j=0; j<n; j++) A[i][j] = i+j;   /* On initialise A */
  flush_cache();           /* fonction qui a pour effet de vider le cache ! */

  /* On calcule la somme des éléments de A */
  s = 0.0;
  for(j=0; j<n; j++)
    for(i=0; i<n; i++) s += A[i][j];

  /* On affiche le résultat */
  printf("%f\n", s);

  return(0);
}
```

On suppose que l'on compile et que l'on exécute ce programme sur un ordinateur qui ne dispose que d'un seul niveau de cache de données direct : ce cache comporte 8 entrées de 64 octets. L'ordinateur dispose d'un cache d'instruction séparé : on ne se préoccupe pas du chargement des instructions dans cet exercice.

1) Comment est stockée la matrice `A` en mémoire centrale ? Représentez la situation dans le cas où `n`= 5.

2) On suppose que `n`= 16, comme indiqué dans le programme. Combien d'octets sont occupés par une ligne de la matrice `A` ?

3) En supposant que l'élément `A[0][0]` de la matrice est stockée sur les 8 premiers octets de la ligne de cache 0, indiquez à quelle ligne appartient chacun des éléments de la matrice. Indiquez également dans quelle entrée du cache sera stocké chaque élément de la matrice.

4) On fixe un certain `j`, $0 \le$ `j` ≤ 7, et on considère la boucle `for(i=0; i<n; i++) s += A[i][j]` : combien de *cache-misses* sont provoqués par les accès en lecture à la matrice `A` ? Que se passe-t-il si on fixe `j` tel que $8 \le$ `j` ≤ 15 ?

5) Au total, combien de *cache-misses* sont provoqués par les opérations d'accès en lecture à la matrice `A` dans la double boucle de sommation ? En déduire le taux de *cache-miss*, c'est-à-dire le pourcentage de *cache-misses* qui se sont produits sur le nombre total d'accès à la matrice.

6) Comment transformer le programme de manière à ramener le nombre de *cache-misses* à 32 ? Quel est désormais le taux de *cache-miss* ?

7) On compile et on exécute le programme initial sur un autre ordinateur, dont la micro-architecture comporte un cache associatif à 8 entrées de 64 octets et 2 voies. Combien de *cache-misses* provoquent les accès en lecture à la matrice `A` ?

Exercice 4 : Mémoire cache - petits calculs

On considère, dans cet exercice [1], que les mots mémoire sont de taille un octet. Soit un cache direct de 4ko (= 2^{12} octets) dont les lignes font 128 (= 2^7) octets. Soient l'adresses hexadécimales de deux données :

— $xA23847EF$

— $x7245E824$

1) Sur combien de bits les adresses sont-elles codées ? Donnez le nombre de bits codant les champs INDICATEUR, ENTREE, et OFFSET (selon la terminologie du cours sur les caches directs).

2) Pour chacune de ces adresses, donnez (en hexadécimal) le numéro de la ligne du cache où on peut la trouver ainsi que l'étiquette (INDICATEUR) de cette ligne et l'offset (OCTET) de la donnée correspondante dans le cache. *Certains calculs fastidieux peuvent être évités...*

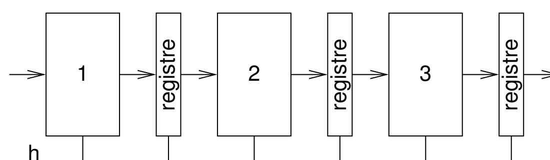
3) Si la donnée correspondant à l'adresse $(A23847EF)_{16}$ considérée est effectivement dans le cache, donnez (en hexadécimal) les adresses qui seront stockées dans la même ligne du cache que celle-ci. *On pourra se contenter d'une réponse où les calculs ne sont pas effectués jusqu'au bout.*

Exercice 5 : Calculs autour de l'exécution pipelinée

On suppose que le chemin de données d'un processeur peut être divisé pour former un pipeline à k étages : chaque étage est un circuit logique, qui consomme les résultats de l'étage précédent, et produit des résultats à destination de l'étage suivant. Après chaque étage i est placé un registre pour stocker les résultats intermédiaires produits à l'étage i à un certain cycle de, et pour qu'il puissent être utilisés par l'étage $i+1$ au cycle suivant de l'horloge h.

Partie 1 : Calcul sur des pipelines

Le temps de stabilisation d'un registre est de 20 ps. Dans le cas de 3 étages, la situation peut être représentée comme suit (h est le signal d'horloge).



Dans les trois questions suivantes, le temps de traitement total d'une instruction, hors passage par les registres, reste de 300 ps.

1) On suppose que le pipeline comporte 3 étages et qu'ils ont tous la même durée de traitement de 100 ps. Quelle est la durée minimale du cycle d'horloge ? Quel est le débit maximal de ce pipeline, exprimé en Gop/s (Giga opérations par seconde) ? Quelle est dans ce cas la latence, c'est-à-dire la durée d'exécution d'une instruction ?

2) On suppose toujours que le pipeline comporte 3 étages, mais maintenant la durée de traitement est variable : 50 ps pour l'étage 1, 150 ps pour l'étage 2, 100 ps pour l'étage 3. Répondez aux mêmes questions que précédemment.

3) On suppose maintenant que l'on double le nombre d'étages et qu'ils ont tous la même durée de traitement. Que deviennent la latence et le débit du pipeline ?

4) Quel débit ne dépassera-t-on jamais, même en continuant d'augmenter la profondeur du pipeline ?

1. Source : `http://www.liafa.jussieu.fr/~amicheli/Ens/Archi/td11.pdf`

Partie 2 : Calibrage d'un pipeline

On considère un chemin de données non-pipeliné qui possède une latence de 10 ns. On suppose que l'on est capable de diviser ce chemin de données en k étage, en répartissant équitablement la latence des circuits logiques entre chaque étage. Le temps de stabilisation d'un registre est de 500 ps.

1) Quel est le temps de cycle d'une version pipelinée du processeur avec un pipeline à k étage ? Faites le calcul pour $k = 2, 4, 8$ et 16.

2) Combien d'étages de pipeline sont requis pour atteindre un temps de cycle de 2 ns ? Combien en faut-il pour atteindre la fréquence de 1 GHz ?

Exercice 6 : Pipeline à 4 étages

On considère un pipeline à 4 étages, comme celui du cours. Les étages du pipeline sont FE (*fetch*, chargement), DE (*decode*, décodage), EX (*execute*, exécution) et SR (*store register*, stockage du résultat en registre). On rappelle que la phase EX d'une instruction ne peut être effectuée avant que ses opérandes n'aient été effectivement calculées et stockées (phase SR terminée) ; si ce n'est pas le cas on parlera ici de *défaut de pipeline*.

1) En l'absence de défaut, si une instruction entre dans le pipeline au cycle i, à quel cycle se finit son exécution ?

2) En ignorant les défauts de pipeline, représentez le traitement des instructions dans le pipeline en remplissant avec FE, DE, EX ou SR dans les cases du tableau suivant. Vous ajouterez entre parenthèses les registres lus dans la phase EX et ceux écrits dans la phase SR :

Instruction	Cycle 1	2	3	4	5	6	7	8	9	10	11
`ADD R4,R1,R2`											
`NOT R4,R4`											
`ADD R3,R3,R4`											
`ADD R4,R5,R1`											
`ADD R5,R5,R2`											

3) Entourez dans le tableau précédent les défauts de pipeline et expliquez-les rapidement.

4) Résoudre les défauts de cache en rajoutant des attentes de pipeline (un *stall*, que vous noterez S). On considère qu'un stall bloque une instruction à un étage précis du pipeline, ainsi que toutes les instructions qui la suivent dans le pipeline.

5) Proposez (en expliquant) un ré-ordonnancement des instructions minimisant le temps total d'exécution. Combien de cycle(s) avez-vous gagné ?

Exercice 7 : Boucles et exécution pipelinée

On ce place sur un processeur dont le chemin de données est implanté à l'aide d'un pipeline à 3 étages :

— chargement de l'instruction pointée par le compteur de programme et décodage ;

— exécution de l'instruction ;

— mise à jour des registres ou de la mémoire d'après le résultat de l'exécution.

On rappelle que la phase exécute d'une instruction ne peut pas commencer avant que la phase de mise à jour d'une instruction dont elle dépend soit terminée. Le chemin de données ne gère pas seul les problèmes de dépendances dans les programmes : on peut néanmoins les éviter en utilisant judicieusement l'instruction `NOP`, qui ne modifie pas l'état du processeur. On considère la portion de programme ci-dessous, donnée dans le langage d'assemblage du LC3 :

```
      AND R0, R0, 0
      ADD R1, R0, 2
loop: ADD R0, R0, R1
      ADD R1, R1, -1
      BRp loop
      ADD R0, R0, 2
```

On souhaite l'adapter, pour permettre son exécution sur le processeur pipeliné décrit.

1) Mettre en évidence les dépendances qui existent dans ce programme. Quels conflits vont se produire si l'on tente d'exécuter en l'état le programme ?

2) Donnez un ordonnancement de l'exécution du code dans le pipeline en insérant au fur et à mesure les instructions `NOP` nécessaires. En déduire une version modifiée du programme qui pourra être exécutée sans problème.

3) En oubliant la seconde instruction du programme, donnez la latence de la boucle en fonction de `R1`.