Nom :	Prénom :
N° Étudiant :	Groupe de TD :

Partiel 2022 – 2023

Architecture des ordinateurs 1 – LU3IN029

Durée : 1h30

Documents autorisés : Aucun document ni machine électronique n'est autorisé à l'exception du mémento MIPS.

Le sujet comporte 19 pages. Ne pas désagrafer les feuilles. Répondre directement sur le sujet. Le barème indiqué pour chaque question n'est donné qu'à titre indicatif. Le barème total est lui aussi donné à titre indicatif : 39 points.

Le partiel est composé d'exercices indépendants.

- Exercice 1 - 8 points : Instruction mémoire et codage binaire – (p. 1)
- Exercice 2 - 17 points : Programme assembleur non optimisé et optimisation – (p. 6)
- Exercice 3 - 14 points : Circuits logiques et ou-exclusif – (p. 13)

Exercice 1 : Instruction d'accès mémoire et codage binaire – 8 points

On s'intéresse à l'introduction d'une nouvelle instruction dans le jeu d'instructions MIPS : `lws`. Sa syntaxe asssembleur est :

```
lws Rdst, (Rsrc1, Rsrc2 lsl Imm)
```

Cette instruction effectue une lecture mémoire d'un mot (4 octets) à l'adresse `A = Rsrc1 + (Rsrc2 « Imm)`. Le signe << désigne l'opération de décalage à gauche et l'immédiat `Imm` a une valeur comprise entre 0 et 31 inclus. Ainsi, l'adresse accédée résulte de l'addition du contenu de `Rsrc1` avec le contenu de `Rsrc2` préalablement décalé à gauche de `Imm`. Le mot lu est placé dans le registre `Rdst`.

Question 1.1 : 2 points

On suppose que le contenu de la mémoire est le suivant :

	Vue par octet				Vue par mot
Adresse (de mot)	+ 0	+ 1	+ 2	+3	
0x10010004	0x01	0x02	0x03	0x04	
0x10010008					0x05060708
0x1001000c					0x10F0E0D0
0x10010010	0x14	0x13	0x12	0x11	

Remplir les cases vides du tableau ci-dessus.

Solution:

	Vue par octet				Vue par mot
Adresse (de mot)	+ 0	+ 1	+ 2	+3	
0x10010004	0x01	0x02	0x03	0x04	0x04030201
0x10010008	0x08	0x07	0x06	0x05	0x05060708
0x1001000c	0x0D	0x0E	0x0F	0x10	0x10F0E0D0
0x10010010	0x14	0x13	0x12	0x11	0x11121314

En supposant que le registre `$4` contient `0x10010000` et `$6` contient `0x00000004`, quelle est la valeur contenue dans `$7` après l'exécution de l'instruction `lws $7, ($4, $6 lsl 2)` ? Justifier votre réponse.

Solution:

L'accès mémoire se fait à 0x10010000 + 4 « 2 soit 0x10010000 + 4 * 4 = 0x10010010.

Le contenu de `$7` est `0x11121314` (ou la valeur donnée pour cette adresse dans le tableau)

Question 1.2 : 3 points

Combien d'opérandes a cette instruction ?

Pour chacun des opérandes, indiquer sa nature (dans l'instruction assembleur) et le nombre de valeurs qu'il peut prendre. En déduire le nombre minimal de bits nécessaires au codage de chacun de ces opérandes dans l'instruction binaire associée à l'instruction assembleur.

Solution:

Cette instruction a 3 opérandes registres (1 destination et 2 sources) ainsi qu'un immédiat.
Les registres ont des numéros compris entre 0 et 31, donc il faut au minimum 5 bits pour encoder le numéro

d'un registre.
L'immédiat a une valeur comprise entre 0 et 31, il faut donc au minimum 5 bits aussi.

Existe-t-il un format de codage MIPS adapté au codage de cette instruction ? Justifier votre réponse.

Solution:

Le format R convient pour coder l'instruction qui serait alors `lws Rd, (Rs, Rt lsl sh)` (on peut échanger la place de Rs et de Rt). Il faut alors un codage pour l'opération dans la table SPECIAL.

Question 1.3 : 2 points

L'opération indiquée par une instruction est associée à un code de 6 bits qui est une entrée soit dans la table OPCOD, soit dans la table SPECIAL (voir le mémento MIPS).

On suppose que pour l'instruction `lws` ce code sur 6 bits est `0b111111`.

En utilisant votre réponse à la question précédente, encoder en binaire l'instruction `lws $17, ($13, $10 lsl 4)` en justifiant votre réponse.

Enfin donner le codage de cette instruction sous forme hexadécimale.

Solution:

La solution ici dépend de la réponse à la question précédente (qui est Rs et qui est Rt). En supposant le codage `lws Rd, (Rs, Rt lsl sh)` :

`lws $17, ($13, $10 lsl 4)` en supposant le codage `lws Rd, (Rs, Rt lsl sh)` avec le format R.
La valeur du CODOP est SPECIAL soit `0b000000` et FUNC vaut `0b111111`
La valeur de Rs est 13 soit `0b01101`
La valeur de Rt et 10 soit `0b01010`
La valeur de Rd est 17 soit `0b10001`
La valeur de Sh est 3 soit `0b00100`

Le codage binaire est donc :

OPCOD	RS	RT	RD	SH	FUNC
000000	01101	01010	10001	00100	111111.

En regroupant par quartet : `0000 0001 1010 1010 1000 1001 0011 1111`
En hexadécimal, cela donne :`0x01AA893F`

En supposant le codage `lws Rd, (Rt, Rs lsl sh)` :
`lws $17, ($13, $10 lsl 4)` en supposant le codage `lws Rd, (Rs, Rt lsl sh)` avec le format R.
La valeur du CODOP est SPECIAL soit `0b000000` et FUNC vaut `0b111111`
La valeur de Rs et 10 soit `0b01010`
La valeur de Rt est 13 soit `0b01101`
La valeur de Rd est 17 soit `0b10001`
La valeur de Sh est 3 soit `0b00100`

Le codage binaire est donc :

OPCOD	RS	RT	RD	SH	FUNC
000000	01010	01101	10001	00100	111111.

En regroupant par quartet : `0000 0001 0100 1101 1000 1001 0011 1111`
En hexadécimal, cela donne :`0x014D893F`

Question 1.4 : 1 point

Donner une suite d'instructions MIPS équivalente à l'instruction `lws $17, ($13, $10 lsl 4)`

Solution:

```
sll  $8, $10, 4  # decalage à gauche de 4
addu $8, $8, $13 # addition du résultat avec $13
lw   $17, 0($18) # acces mémoire
```

Exercice 2 : Programmation assembleur – 17 points

Question 2.1 : 12 points

Soit le code C suivant, qui étant donné un tableau d'entiers `tab` de taille `size` (qui vaut au plus 255 car de type `unsigned char`), compte le nombre d'éléments de `tab` qui sont impairs et le nombre d'élements de `tab` qui sont pairs. Ces nombres sont affichés à la fin du programme.

```
unsigned char size = 8;
int tab[] = {3, 5, 8, 10, 7, 25, 31, 24};

void main() {
  unsigned char i = 0;
  unsigned char nb_impair = 0;
  unsigned char nb_pair = 0;

  while (i < size)
  {
    if ((tab[i] & 0x1) == 0) /* test parité */
    {
      nb_pair++;
    }
    else
    {
      nb_impair++;
    }
    i++;
  }

  printf("%d", nb_pair);    /* affichage d'un entier */
  printf("%d", nb_impair);  /* affichage d'un entier */
  exit();
}
```

Donner le code assembleur correspondant au programme ci-dessus. Le programme assembleur NE DOIT PAS être optimisé. Vous traduirez donc de manière fidèle le programme C en assembleur.

Vous devez commenter un minimum votre code assembleur afin d'indiquer ce que réalise chaque instruction (lien avec le code C ou convention MIPS).

Tout code illisible ou non commenté recevra la note 0.

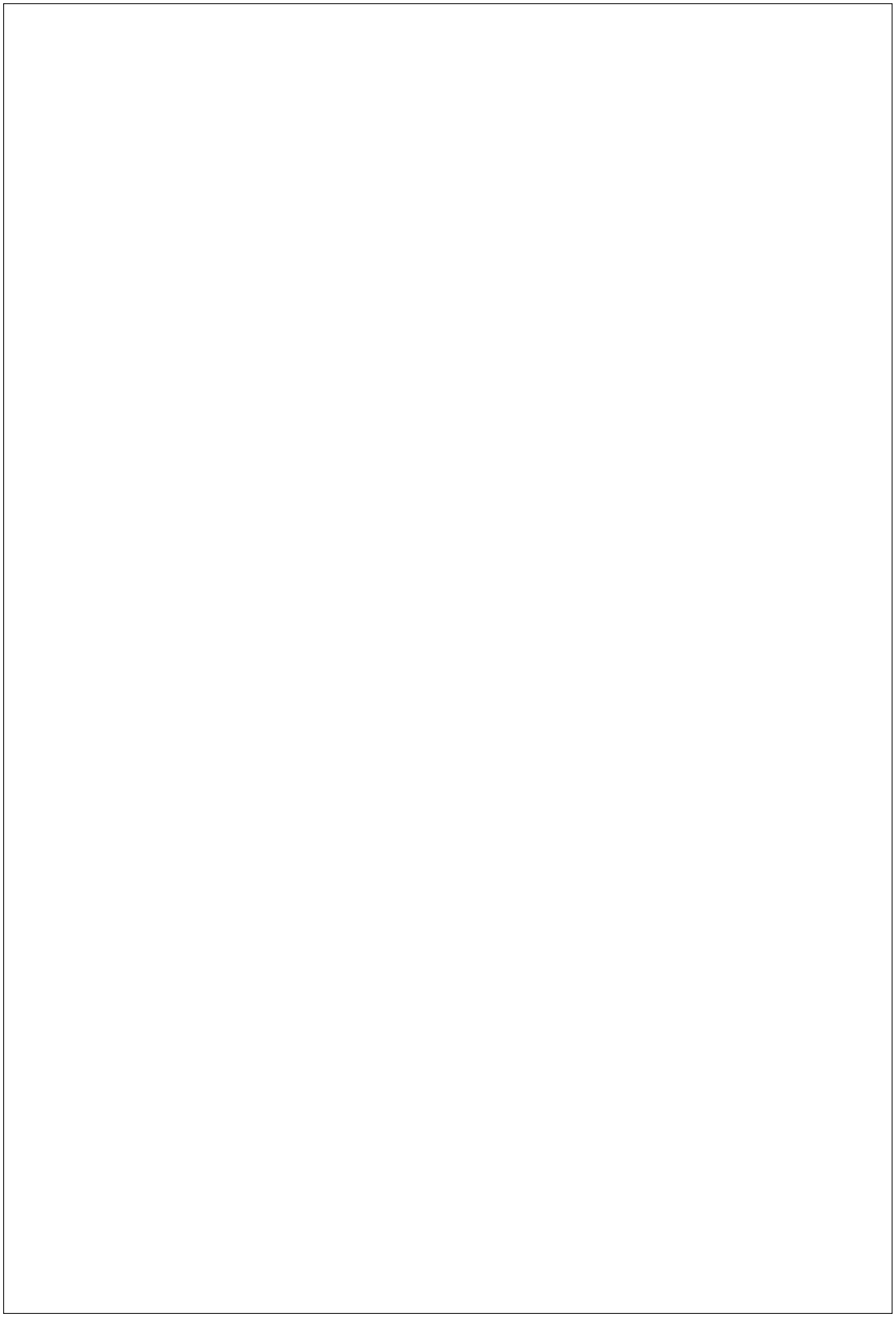
Solution:

```
.data
size: .byte 8
tab:  .word 3, 5, 8, 10, 7, 25, 31, 24
```

```
.text
   addiu $29, $29, -4    # nv = 3 char = 3 octets = 1 mot
   sb    $0, 0($29)      # i = 0
   sb    $0, 1($29)      # nb_impair = 0
   sb    $0, 2($29)      # nb_pair = 0

while:  # i < size ?
   lui  $8, 0x1001
   lbu  $8, 0($8)        # size
   lbu  $9, 0($29)       # i
   slt  $10, $9, $8      # $10 = 1 si i < size
   beq  $0, $10, finwhile
   # tab[i] & 0x01
   lui  $8, 0x1001
   ori  $8, $8, 4        # tab
   lbu  $9, 0($29)       # i
   sll  $9, $9, 2        # i * 4
   addu $8, $9, $8       # &tab[i]
   lw   $8, 0($8)        # tab[i]
   andi $8, $8, 1
   beq  $8, $0, pair
   #nb_impairs++
   lbu   $9, 1($29)
   addiu $9, $9, 1
   sb    $9, 1($29)
   j suite
pair:    #nb_pairs++
   lbu   $9, 2($29)
   addiu $9, $9, 1
   sb    $9, 2($29)
suite:   #i++
   lbu   $8, 0($29)
   addiu $8, $8, 1
   sb    $8, 0($29)
   j while

finwhile:
   # affichage nb_pairs
   lbu  $4, 2($29)
   ori  $2, $0, 1
   syscall
   #affichage nb_impairs
   lbu  $4, 1($29)
   ori  $2, $0, 1
   syscall
   # fin du programme
   addiu $29, $29, 4
   ori   $2, $0, 10
   syscall
```



Question 2.2 : 5 points

On s'intéresse à l'optimisation du code donné en réponse à la question précédente.

Pour cela, numéroter les instructions de votre code réponse de la question précédente.

Donner les optimisations possibles de votre code : pour chaque optimisation possible, indiquer la ou les lignes concernées par cette optimisation. Expliquer ce qu'il faudrait changer au code pour mettre en oeuvre cette optimisation, ainsi que, le cas échéant les conséquences sur d'autres lignes du code.

Solution:

Les variables locales peuvent etre optimisées en registre :

- Pour chacune des 3 variables, on choisit un registre qu'on initialise à 0 juste après l'allocation de la pile.
- On élimine alors les lectures depuis pile de ces variables, on utilise le registre associé à la variable.
- De la meme manière, les écritures en pile pour ces variables ne sont plus nécessaires, c'est le registre qui est mis à jour.

Ainsi on peut :

- éliminer des lectures en pile pour l'utilisation de `i` dans la condition de boucle et dans le calcul de l'adresse de `tab[i]`,
- éliminer des lectures et écritures en la pile pour incrémentations de `nb_pairs`, `nb_impairs` et `i`.

Il faut faire attention lors du choix du registre pour la variable `nb_impairs` car lors de l'affichage, si elle a été mise dans un registre non persistant, sa valeur est perdue par l'affichage de `nb_pairs`, il faut soit l'écrire dans son emplacement sur la pile, soit choisir un registre persistant...

La valeur de la variable `size` peut être lue en mémoire avant la boucle et conservée dans un registre, ce registre est utilisé directement dans le calcul de la condition de boucle.

De la même manière, l'adresse du tableau `tab` peut etre calculée avant la boucle et conservée dans un registre. Ce registre est alors utilisé dans la boucle pour la calcul de l'adresse de `tab[i]`.

Ci-dessous une version optimisée du code (non demandée)

```
.text
   addiu $29, $29, -4    # nv = 3 char = 3 octets = 1 mot
   xor   $9,  $9, $9      # i = 0
   xor   $16, $16, $16    # nb_impair = 0
   xor   $17, $17, $17    # nb_pair = 0

   lui  $8, 0x1001
   lbu  $8, 0($8)        # size

   lui  $11, 0x1001
   ori  $11, $11, 4       # tab

while2:  # i < size ?
   slt  $10, $9, $8      # $10 = 1 si i < size
   beq  $0, $10, finwhile2

   # tab[i] & 0x01
   sll  $12, $9, 2        # i * 4
   addu $12, $11, $12     # adressse de tab[i]
   lw   $12, 0($12)       # tab[i]
   andi $12, $12, 1
   beq  $12, $0, pair2
   #nb_impairs++
   addiu $16, $16, 1
   j     suite2

pair2:   #nb_pairs++
   addiu $17, $17, 1

suite2: #i++
   addiu $9, $9, 1
   j while2

finwhile2:
   # affichage nb_pairs
   ori  $4, $17, 0
   ori  $2, $0, 1
   syscall
   #affichage nb_impairs
   ori  $4, $16, 0
   ori  $2, $0, 1
   syscall
```

```
# fin du programme
addiu $29, $29, 4
ori   $2, $0, 10
syscall
```

Exercice 3 : Circuits logiques et ou-exclusif – 14 points

L'opération OU-EXCLUSIF sur n entrées a deux versions :

1. Le résultat vaut 1 si et seulement si une seule des n entrées vaut 1. On appelle `XOR1` cette version.
2. Le résultat vaut 1 si un nombre impair d'entrées vaut 1. On appelle `XORimpair` cette version.

On s'intéresse au cas où `n` vaut 4. On appelle `a`, `b`, `c`, `d` les 4 entrées sur 1 bit.

Question 3.1 : 3 points

Combien de lignes les tables de vérités des fonctions booléennes `XOR1` et `XORimpair` à 4 entrées (sur 1 bit) ont-elles ? Donner ces tables de vérité.

Solution:

16 lignes dans les tables de vérités qui sont :

a	b	c	d	XOR1	XORimpair
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	0	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	0	1
1	0	0	0	1	1
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	0	1
1	1	1	1	0	0

Donner la forme normale disjonctive de XOR1 et XORimpair

Solution:

XOR1(a,b,c,d) =
not(a).not(b).not(c).d + not(a).not(b).c.not(d) + not(a).b.not(c).not(d) + a.not(b).not(c).not(d)

XORimpair(a,b,c,d) =
not(a).not(b).not(c).d + not(a).not(b).c.not(d) + not(a).b.not(c).not(d) + not(a).b.c.d +
a.not(b).not(c).not(d) + a.not(b).c.d + a.b.not(c).d + a.b.c.not(d)

Question 3.2 : 3 points

Simplifier les expressions obtenues à la question précédente pour faire apparaître le maximum de portes XOR à deux entrées dans les deux expressions. Vous détaillerez vos simplifications que vous justifierez.

Solution:

XOR1(a,b,c,d) =
not(a).not(b).not(c).d + not(a).not(b).c.not(d) + not(a).b.not(c).not(d) + a.not(b).not(c).not(d)

= not(a).not(b).[not(c).d + c.not(d)] + not(c).not(d).[not(a).b + a.not(b)]

= not(a).not(b).[c xor d] + not(c).not(d).[a xor b]

XORimpair(a,b,c,d) =
not(a).not(b).not(c).d + not(a).not(b).c.not(d) + not(a).b.not(c).not(d) + not(a).b.c.d +
a.not(b).not(c).not(d) + a.not(b).c.d + a.b.not(c).d + a.b.c.not(d)

= [not(a).not(b) + a.b][c xor d] + [a.not(b) + not(a).b].[not(c).not(d) + c.d]

= [not(a xor b)][c xor d] + [a xor b][not(c xor d)]

= (a xor b) xor (c xor d)

NB : plusieurs réponses sont bonnes selon la factorisation réalisée par les étudiants mais la formule finale doit contenir 3 xor et rien d'autres.

On souhaite réaliser un circuit `XOR_4` qui, à partir d'une commande `cmd` et 4 entrées `a`, `b`, `c` et `d` sur 1 bits, réalise le `XOR1` si `cmd` = 0, et réalise le `XORimpair` si `cmd` = 1.

Question 3.3 : 1 point

Avec quel circuit peut on réaliser la sélection d'une valeur parmi deux ? Donner une expression booléenne de ce circuit.

Solution:

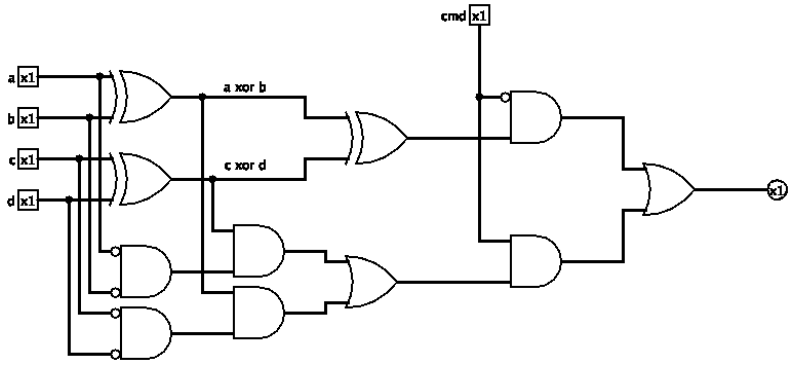
C'est le MUX2(a,b,s) = a.not(s) + b.s

Question 3.4 : 4 points

Donner une réalisation schématique du circuit `XOR_4` permettant de réaliser la sélection d'une des deux versions du `XOR` à 4 entrées. Votre réalisation doit être "optimisée" : aucun calcul ne doit être réalisé deux fois. Votre circuit ne doit contenir que des portes NOT, XOR, ET et OU. Vous pouvez utiliser des portes ET et OU avec des entrées ou sorties complémentées (mais elles doivent être dessinées de façon lisible).

Solution:

On peut factoriser selon les réponses aux questions précédentes le calcul de (a xor b) et de (c xor d). Cela donne le circuit suivant :



Le chemin critique d'un circuit correspond au plus long chemin en nombre de portes ET-OU entre une

entrée et une sortie du circuit. La taille de ce chemin est le nombre de portes ET-OU. Quel est le chemin critique de votre circuit et sa taille ? Justifier votre réponse.

Solution:

Le plus long chemin dans une porte XOR est de longueur 2 (1 ET et 1 OU), donc le plus long chemin dans la solution donnée est celui composé des portes XOR-XOR-ET-OU soit 6 portes.

Question 3.5 : 3 points

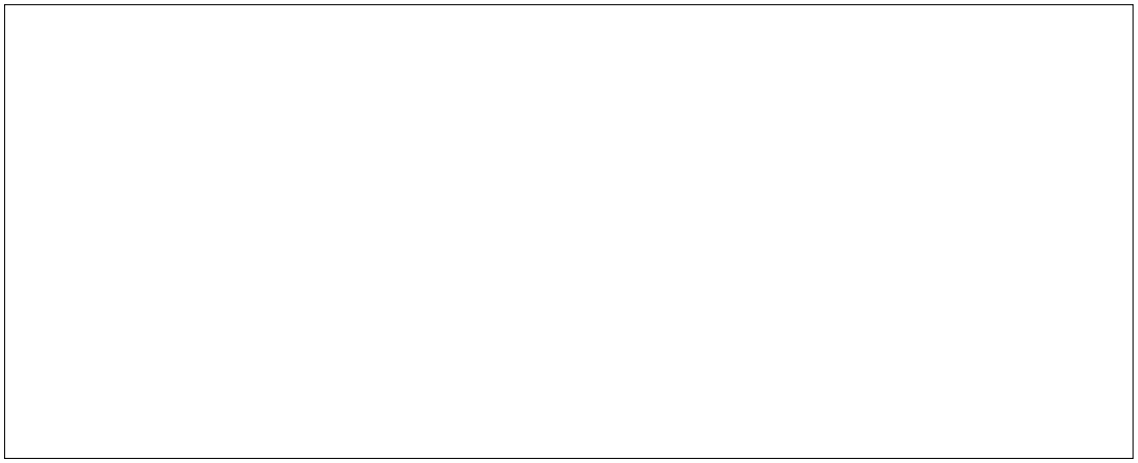
On s'intéresse désormais uniquement au `XORimpair`, plus précisément au circuit qui à partir d'un mot binaire de $n = 2^k$ bits, avec $k \geq 1$, a pour sortie 1 si le nombre de bits à 1 dans le mot est impair, 0 si ce nombre est pair.

Dans le cas où $n = 2^3$, combien de portes `xor` contient le circuit ? Justifier.

Solution:

Il faut un 7 XOR pour déterminer si le nombre de bits à 1 est impair. On peut xorer les bits à partir de la droite (ou gauche) et xorer le résultat avec le bit suivant. On peut aussi xorer les bits $2i$ et $2i$+1 pour tout i dans $[0, 2^{k-1} - 1]$, et xorer les résultats deux à deux jusqu'à n'avoir qu'un seul bit de résultat. Il faut aussi 7 xor bien sur. (Arbre binaire dégénéré dans le 1er cas et arbre équilibré dans le deuxième)

Dans le cas général où $n = 2^k$, combien de portes `xor` contient ce circuit ? Comment réduire le chemin critique (le plus long en nombre de portes ET-OU) de ce calcul ? Quel est dans ce cas sa taille ?



Solution:

Il faut $2^k - 1$ (ou $n - 1$) portes xor. Le chemin le plus long est minimisé si lorsque l'on compare les bits 2 à 2, puis les résultats deux à deux jusqu'à obtenir le résultat. Le bits et les portes XOR forment alors un arbre binaire équilibré. La profondeur de l'arbre (et donc du circuit) est de k, comme ce sont des portes XOR, le chemin le plus long est composé de $2k$ portes ET-OU.