MINISTÈRES
ÉDUCATION
JEUNESSE
SPORTS
ENSEIGNEMENT
SUPÉRIEUR
RECHERCHE
Liberté
Égalité
Fraternité

EAE INF 1

SESSION 2026

AGREGATION
CONCOURS EXTERNE

Section : INFORMATIQUE

COMPOSITION EN INFORMATIQUE

Durée : 5 heures

L'usage de tout ouvrage de référence, de tout dictionnaire et de tout matériel électronique (y compris la calculatrice) est rigoureusement interdit.

Il appartient au candidat de vérifier qu'il a reçu un sujet complet et correspondant à l'épreuve à laquelle il se présente.

Si vous repérez ce qui vous semble être une erreur d'énoncé, vous devez le signaler très lisiblement sur votre copie, en proposer la correction et poursuivre l'épreuve en conséquence. De même, si cela vous conduit à formuler une ou plusieurs hypothèses, vous devez la (ou les) mentionner explicitement.

NB : Conformément au principe d'anonymat, votre copie ne doit comporter aucun signe distinctif, tel que nom, signature, origine, etc. Si le travail qui vous est demandé consiste notamment en la rédaction d'un projet ou d'une note, vous devrez impérativement vous abstenir de la signer ou de l'identifier.
Le fait de rendre une copie blanche est éliminatoire.

Tournez la page S.V.P.

INFORMATION AUX CANDIDATS

Vous trouverez ci-après les codes nécessaires vous permettant de compléter les rubriques figurant en en-tête de votre copie.

Ces codes doivent être reportés sur chacune des copies que vous remettrez.

Concours	Section/option	Epreuve	Matière
EAE	6200A	101	9422

IMPRIMERIE NATIONALE – 26 0113 – D'après documents fournis

Ce sujet est constitué de trois parties.

Dépendances. Les trois parties sont indépendantes et doivent être traitées toutes les trois. On veillera à bien indiquer sur la copie les changements de partie. On veillera également à bien se conformer aux attendus de chaque exercice en terme de précondition de fonction et de justification de complexité.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

Réseaux de processus synchrones périodiques

Cette partie doit être traitée dans le langage C.

Un réseau de processus est un système logiciel formé d'un certain nombre de processus parallèles liés par des canaux de communication soumis à certaines restrictions. Ces restrictions rendent l'exécution du réseau globalement déterministe, ce qui facilite grandement le débogage, et a rendu ce modèle de programmation populaire pour l'implémentation de systèmes embarqués. Dans cette partie, on se propose de programmer une petite bibliothèque écrite en langage C permettant l'analyse de performance de tels réseaux en supposant un modèle d'exécution synchrone et des communications périodiques.

Généralités

On va s'intéresser à des processus dont les communications peuvent être décrites par des *mots ultimement périodiques*, comme dans l'exemple figurant sur la partie gauche de la fig. 1. Le processus T dispose de deux canaux d'entrée nommés a et b et d'un canal de sortie nommé c. Son exécution est cyclique. À chaque cycle, le processus T consomme un nombre fixé de messages sur chacun de ses canaux d'entrée et produit simultanément un nombre fixé de messages sur son canal de sortie. Ces nombres sont déterminés par les *mots ultimement périodiques* figurant au dessus de chaque canal à la fig. 1.

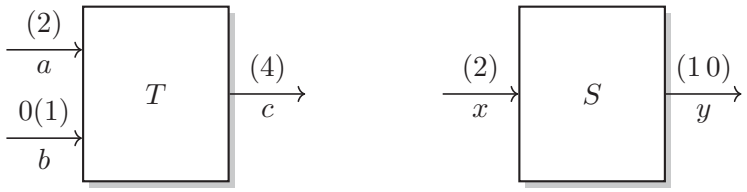


Figure 1: Exemples de processus à communication périodique

La notation $u(v)$, où u et v sont des suites finies d'entiers avec u potentiellement vide et v non vide, désigne la suite infinie d'entiers commençant par le *préfixe* u suivi de la *partie périodique* v répétée infiniment. Une telle suite est appelée *mot ultimement périodique*. Par exemple, les mots ultimement périodiques $0(1)$, (2) et $(0\,1)$ désignent respectivement les suites infinies d'entiers $[0, 1, 1 \dots]$, $[2, 2, 2 \dots]$ et $[0, 1, 0, 1 \dots]$. Lors de son premier cycle d'exécution,

le processus T va donc consommer 2 messages sur le canal a, aucun message sur le canal b, et produire 4 messages sur le canal c. Puis, à chaque cycle d'exécution à partir du second, il consommera 2 messages sur le canal a, un seul sur le canal b, et en produira 4 sur le canal c.

L'analyse des réseaux de processus cherche à déterminer l'existence ou non d'exécutions sans interblocages, c'est-à-dire où chaque processus progresse infiniment souvent. Considérons par exemple le processus S figurant sur la droite de la fig. 1. Peut-on relier par des canaux de capacité finie la sortie c du processus T à l'entrée x du processus S, ainsi que la sortie y du processus S à l'entrée b du processus T, sans créer d'interblocage ? La réponse est oui. En revanche, ce n'est pas le cas si on cherche maintenant à relier la sortie y de S à l'entrée a de T. En effet, dans ce cas, pour exécuter le premier cycle du processus T, il faut exécuter au moins un cycle du processus S, puisque T doit lire sur son canal a deux données produites par S sur son canal y. Mais, symétriquement, pour exécuter le premier cycle du processus S, il faut exécuter au moins un cycle du processus T, puisque S doit lire sur son canal x les données produites par T sur son canal c. Il n'existe donc pas d'exécution globale du réseau où S et T progressent au moins une fois chacun, et a fortiori aucune exécution où ils progressent infiniment souvent !

On peut détecter les interblocages dans le cas général en étudiant les mots ultimement périodiques caractérisant les transmissions de données sur les canaux du réseau. On se propose dans cette partie d'implémenter les fonctions élémentaires de manipulation de ces mots sur lesquelles une telle analyse repose.

Mots ultimement périodiques

On représentera un mot ultimement périodique , en anglais *ultimately periodic word*, en langage C sous la forme d'une structure contenant le préfixe et la partie périodique.

```
typedef struct up_word { uint_vect_ptr_t u; uint_vect_ptr_t v; } *up_word_ptr_t;
```

Cette représentation suppose l'existence d'une structure `uint_vect` représentant des séquences finies d'entiers positifs, d'un type de pointeur associé `uint_vect_ptr_t`, ainsi que d'une petite bibliothèque de fonctions permettant de le manipuler. Une telle bibliothèque est proposée à la fig. 2. On rappelle que `size_t` est le type des entiers non signés pouvant contenir la plus grande taille d'un objet adressable.

Si w est un mot fini ou infini et $i \geq 0$, on note w_i pour son $i$ème élément. Si w est un mot fini, on note $|w|$ pour sa longueur et $|w|_1$ pour son *poids*, c'est-à-dire la somme totale de tous ses éléments.

Question 1.1. Proposer une implémentation de la structure `uint_vect` et des fonctions spécifiées à la fig. 2 telle que toutes les fonctions à l'exception de `uint_vect_init` s'exécutent en temps constant (non amorti).

```
/* La déclaration du type des mots finis contenant des entiers positifs. */
typedef struct uint_vect *uint_vect_ptr_t;

/* La fonction `uint_vect_init(len)` alloue et initialise un mot fini de
   taille fixée `len`. La fonction renvoie un pointeur vers le mot nouvellement
   alloué ou NULL en cas d'erreur.
   Les entiers du mot sont tous initialisés à 1. */
uint_vect_ptr_t uint_vect_init(size_t);

/* La fonction `uint_vect_len(pw)` renvoie la longueur du mot pointé par
   `pw`. */
size_t uint_vect_len(uint_vect_ptr_t);

/* La fonction `uint_vect_weight(pw)` renvoie le poids du mot pointé par `pw`,
   c'est-à-dire la somme de tous les entiers qu'il contient. */
unsigned int uint_vect_weight(uint_vect_ptr_t);

/* La fonction `uint_vect_set(pw, i, v)` modifie le mot fourni pour écrire
   l'entier `v` à l'indice `i` du mot pointé par `pw`. Les écritures hors des
   bornes doivent être ignorées, tout comme la valeur `UINT_MAX` pour `v`. */
void uint_vect_set(uint_vect_ptr_t, size_t, unsigned int);

/* La fonction `uint_vect_get(pw, i)` renvoie le contenu du `i`ème indice du mot
   pointé par `pw`. La fonction renverra la valeur UINT_MAX si l'indice `i`
   est en dehors des bornes du mot. */
unsigned int uint_vect_get(uint_vect_ptr_t, size_t);
```

Figure 2: Bibliothèque de manipulation de mots finis

Tournez la page S.V.P.

Question 1.2. Programmer la fonction `unsigned int up_word_get(up_word_ptr_t, size_t)` telle que `up_word_get(pw, i)` renvoie w_i avec w le mot ultimement périodique vers lequel pointe pw.

Fonctions de cumul

À chaque mot infini w, ultimement périodique ou non, est associé une *fonction de cumul*, notée $\mathcal{O}_w$ et définie par

$$\mathcal{O}_w(i) = \sum_{j<i} w_j.$$

En particulier, on a $\mathcal{O}_w(0) = 0$ pour tout mot w, par définition.

Les fonctions de cumul jouent un rôle crucial dans la modélisation des réseaux de processus. Si w caractérise les émissions (ou réceptions) de message d'un processus sur un canal de communication, alors $\mathcal{O}_w(i)$ désigne la quantité totale de messages émis (ou reçus) après i cycles d'exécution du processus. En particulier, si p caractérise le nombre d'émissions et q le nombre de réceptions pour le même canal, alors la quantité $\mathcal{O}_p(i) - \mathcal{O}_q(i)$ désigne la quantité totale de données stockées au sein du canal après i exécutions du processus émetteur et du processus récepteur.

Question 1.3. Définir une fonction `unsigned int cumulative_sum(up_word_ptr_t, size_t)` telle que `cumulative_sum(p, i)` renvoie $\mathcal{O}_p(i)$.

Pour a et b entiers naturels, on note $\mathrm{lcm}(a, b)$ le plus petit commun multiple de a et b. Soient $p = u_1(v_1)$ et $q = u_2(v_2)$ deux mots ultimement périodiques. L'entier $\max(|u_1|, |u_2|) + \mathrm{lcm}(|v_1|, |v_2|)$ est appelé une *hyperpériode* de p et q.

On suppose définie une fonction `size_t lcm(size_t, size_t)` telle que `lcm(a, b)` renvoie $\mathrm{lcm}(a, b)$.

Question 1.4. Utiliser la fonction `lcm` pour définir une fonction `size_t hyperperiod(up_word_ptr_t, up_word_ptr_t)` telle que `hyperperiod(p, q)` renvoie une hyperpériode des mots ultimement périodiques p et q.

Une hyperpériode donne une borne supérieure sur le nombre de cycles après lequel les comportements de p et q seront resynchronisés.

Question 1.5. Soient $p = u_1(v_1)$ et $q = u_2(v_2)$ deux mots ultimement périodiques, montrer qu'il existe des mots finis u'_1, v'_1, u'_2, v'_2 tels que $p = u'_1(v'_1)$ et $q = u'_2(v'_2)$ où $|u'_1| = |u'_2| = \max(|u_1|, |u_2|)$ et $|v'_1| = |v'_2| = \mathrm{lcm}(|v_1|, |v_2|)$.

Soit p et q respectivement le rythme d'écriture et de lecture dans le même canal. Le rythme d'écriture ne doit pas être trop lent par rapport au rythme de lecture, sans quoi on risquerait de

lire plusieurs fois les mêmes données, ou bien de lire des données non initialisées. Symétriquement, le rythme de lecture ne doit pas être trop lent par rapport au rythme d'écriture, sans quoi on risquerait d'écraser des données non encore lues. On peut caractériser cette situations à l'aide de la relations de *synchronisabilité*, notée $p \bowtie q$, et définie par

$$p \bowtie q \iff \exists B \in \mathbb{N}, \forall i, |\mathcal{O}_p(i) - \mathcal{O}_q(i)| \leq B$$

Soit $p = u(v)$ un mot ultimement périodique, on appelle *taux d'accroissement* de p le rationnel $\mathrm{rt}(p) = \frac{|v|_1}{|v|}$. On admet que ce nombre ne dépend pas de l'écriture de p.

Question 1.6. Définir une fonction `double rate(up_word_ptr_t)` telle que `rate(p)` renvoie $\mathrm{rt}(p)$.

Question 1.7. Soient p et q deux mots ultimement périodiques, montrer que $p \bowtie q \iff \mathrm{rt}(p) = \mathrm{rt}(q)$.

Question 1.8. Définir une fonction `bool synchronizes(up_word_ptr_t, up_word_ptr_t)` de sorte que `synchronizes(p, q)` renvoie `true` si et seulement si p et q sont synchronisables ($p \bowtie q$).

Si p et q sont synchronisables, il est facile de placer en attente dans un tampon de capacité finie les écritures en attente d'être lues. Par contre, les lectures en attente sont problématiques. On définit ainsi la relation p précède q, notée $p \preceq q$, par

$$\forall i \in \mathbb{N}, \mathcal{O}_p(i) \geq \mathcal{O}_q(i)$$

Si p précède q, il est garanti que chaque lecture pourra s'effectuer.

Question 1.9. Soit p et q deux mots ultimement périodiques tels que $p \bowtie q$ et $H_{p,q}$ une hyperpériode de p et q, montrer que

$$\forall i \geq H_{p,q}, \exists j < H_{p,q}, \mathcal{O}_p(i) - \mathcal{O}_q(i) = \mathcal{O}_p(j) - \mathcal{O}_q(j)$$

Question 1.10. Définir une fonction `bool precedes(up_word_ptr_t, up_word_ptr_t)` de sorte que `precedes(p, q)` renvoie `true` si et seulement si p précède q ($p \preceq q$).

p (écriture) q (lecture)	(1 0) (0 1)	(1) (1)	(2 0) (1)	(3) 0(3)
`capacity(p, q)`	1	0	1	3

Table 1: Exemples d'écritures et lectures ultimement périodiques

Question 1.11. Définir une fonction `size_t capacity(up_word_ptr_t, up_word_ptr_t)` telle que, pour tous les mots ultimement périodiques p et q synchronisables, l'appel `capacity (p, q)` renvoie la capacité minimale d'un canal vers lequel on écrit au rythme p et depuis lequel on lit au rythme q. On supposera que les messages sont transmis instantanément au cours d'un instant. La table 1 donne plusieurs exemples de résultats attendus.

Tournez la page S.V.P.

Tas persistants

Cette partie doit être traitée dans le langage OCaml.

Dans cette partie, on souhaite réaliser une implémentation du type de données abstrait tas persistant. C'est-à-dire que l'on souhaite définir un type `'a heap` munit des quatre opérations suivantes :

- `insert : 'a heap -> 'a -> 'a heap` telle que l'appel `insert t x` renvoie le tas obtenu après insertion de l'élément `x` dans le tas `t` ;
- `find_max : 'a heap -> 'a` telle que l'appel `find_max t` renvoie la valeur maximale dans le tas `t` supposé non vide ;
- `merge : 'a heap -> 'a heap -> 'a heap` telle que l'appel `merge t1 t2` renvoie un tas contenant les éléments du tas `t1` et les éléments du tas `t2` ;
- `delete_max : 'a heap -> 'a heap` tel que l'appel `delete_max t` renvoie le tas obtenu en supprimant dans le tas `t` son plus grand élement.

Tas binomiaux

On définit un *arbre binomial de rang* r de la manière suivante :

- un arbre binomial de *rang* 0 est un arbre réduit à sa racine ;
- un arbre binomial de rang $r+1$ est la *liaison* de deux arbres binomiaux de rang r de sorte que l'un des deux arbres est placé comme fils le plus à gauche de la racine de l'autre.

Une représentation d'arbres binomiaux est donnée dans la figure 3.

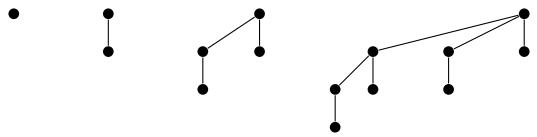


Figure 3: Arbres binomiaux de rang 0 à 3

Question 2.1. Donner le nombre de nœuds d'un arbre binomial de rang r. Donner le nombre de nœuds de profondeur k d'un arbre binomial de rang r.

Sur un ensemble totalement ordonné on dit qu'un arbre binomial est *ordonné en tas* si l'étiquette de chaque nœud est supérieure ou égale à celles de ses descendants.

Lors d'une *liaison* de deux arbres ordonnés en tas, l'arbre ayant la plus petite racine devient le fils de celui ayant la plus grande racine.

Un *tas binomial* est une liste d'arbres binomiaux ordonnés en tas triée par ordre strictement croissant de rangs (deux arbres quelconques d'un même tas n'ont donc jamais le même rang).

La *taille* du tas est le nombre total de nœuds contenus dans le tas.

Question 2.2. Dessiner un tas binomial de taille 21. En généralisant cet exemple, expliciter le lien entre l'écriture binaire de la taille n d'un tas binomial et le rang des arbres qui apparaissent dans ce tas. Quel est le nombre maximal d'arbres binomiaux d'un tas binomial de taille n ?

Dans la suite, on considérera que le type générique `'a` est totalement ordonné. On définit alors un arbre binomial par la donnée de l'étiquette de sa racine, un entier représentant son rang et la liste de ses fils. On introduit les types et valeurs suivantes :

```
type rank = int
type 'a tree = N of 'a * rank * 'a tree list
let root (N(x,r,c)) = x
let rank (N(x,r,c)) = r
exception Empty
```

Question 2.3. Écrire la fonction `link` réalisant la liaison de deux arbres de même rang. On supposera que les deux arbres sont de même rang sans le vérifier.

On introduit à présent le type tas binomial.

```
type 'a bin_heap = 'a tree list
```

On rappelle que les tas binomiaux sont triés par ordre croissant de rang, et que deux arbres du tas ne peuvent pas avoir le même rang.

Question 2.4. Écrire la fonction `find_max` renvoyant le maximum d'un tas non vide. Cette fonction devra être de complexité $O(\log n)$, où n est la taille du tas. Si le tas est vide, on lèvera l'exception `Empty`.

On veut à présent écrire une fonction `insert` d'insertion d'un élément dans un tas binomial. On commence par écrire la fonction `insert_tree` prenant un arbre binomial de rang r et un tas binomial et qui renvoie un tas binomial contenant tous les nœuds de l'arbre et du tas initial.

Pour cela, on distingue deux cas :

- si le tas ne contient pas d'arbre de rang r, on insère l'arbre dans le tas en préservant la stricte croissance des rangs ;
- sinon, on crée un arbre de rang $r+1$ en liant l'arbre de rang r présent et le nouvel arbre qu'on souhaite insérer, puis on insère récursivement cet arbre de rang $r+1$.

Question 2.5. Détailler les étapes de l'insertion d'un arbre binomial de rang 1 dans un tas binomial de taille 22. Combien de liaisons sont réalisées ? Donner le pire cas.

Question 2.6. Écrire la fonction `insert_tree` qui insère un arbre binomial dans un tas en maintenant la propriété d'ordre des rangs.

Tournez la page S.V.P.

Question 2.7. En déduire la fonction `insert` d'un élément dans un tas.

On s'intéresse maintenant à la suppression du maximum.

Pour supprimer le maximum, on cherche l'arbre du tas dont la racine est le maximum et on réinsère ses fils par fusion dans la file.

Question 2.8. Montrer que si l'on parcourt de la droite vers la gauche les enfants de la racine d'un arbre binomial de rang r, on obtient un tas binomial valide.

Question 2.9. Écrire la fonction `merge` réalisant la fusion de deux tas binomiaux.

Question 2.10. Écrire la fonction `delete_max`.

Tas binomiaux asymétriques

Dans cette partie, nous étudions une variante qui permet d'obtenir une insertion en $O(1)$.

Les *tas binomiaux asymétriques* sont des listes d'*arbres binomiaux asymétriques*. Ces derniers sont définis récursivement de la manière suivante :

- un arbre binomial asymétrique de rang 0 est un arbre réduit à sa racine ;
- un arbre binomial asymétrique de rang $r+1$ est formé de l'une des trois manières suivantes :
  - une *liaison simple* qui place un arbre binomial asymétrique de rang r comme fils le plus à gauche d'un arbre binomial asymétrique de rang r,
  - une *liaison asymétrique de type A* qui place deux arbres binomiaux asymétriques de rang r comme fils d'un arbre binomial asymétrique de rang 0,
  - une *liaison asymétrique de type B* qui place un arbre binomial asymétrique de rang 0 et un arbre binomial asymétrique de rang r comme fils d'un autre arbre binomial asymétrique de rang r.

La Figure 4 illustre les trois liaisons possibles.

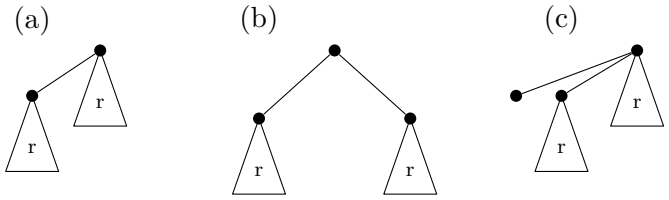


Figure 4: Les trois méthodes de construction d'un arbre binomial asymétrique de rang $r + 1$. (a) Liaison simple ; (b) liaison asymétrique de type A ; (c) liaison asymétrique de type B

Question 2.11. Donner un encadrement optimal du nombre de nœuds d'un arbre binomial asymétrique de rang r. Expliciter des exemples atteignant les bornes. Quel est le lien entre hauteur et rang ?

Un arbre binomial asymétrique est ordonné en tas si l'étiquette de chaque nœud est supérieure ou égale à celles de ses descendants.

Lors d'une liaison simple entre deux arbres de rang r, l'arbre de plus petite racine devient fils de l'arbre de racine la plus grande.

Lors d'une liaison asymétrique, les deux arbres de racines les plus petites deviennent fils de l'arbre de plus grande racine. On a une liaison de type A si l'arbre de rang 0 a la plus grande racine, B sinon.

Un *tas binomial asymétrique* est une liste d'arbres binomiaux asymétriques ordonnés en tas où aucun arbre n'a le même rang, à l'exception éventuelle des deux arbres de plus petit rang. On maintient cette liste triée par ordre croissant de rang.

L'insertion d'un élément dans un tas binomial asymétrique se fait selon les modalités suivantes. On commence par créer un arbre de rang 0 contenant l'élément à insérer. On cherche ensuite les deux plus petits rangs des arbres du tas. Si les deux rangs sont égaux, on fait une liaison asymétrique et on obtient un arbre de rang $r + 1$ à insérer. Sinon, on ajoute l'arbre de rang 0 au tas.

Question 2.12. Justifier que l'algorithme précédent renvoie bien un tas binomial asymétrique valide.

Question 2.13. Écrire la fonction `insert` et donner sa complexité. Quel est le nombre maximal de liaisons effectuées ?

Tas binomiaux enracinés

Un tas binaire classique permet de déterminer le maximum avec une complexité en $O(1)$. On décrit maintenant une amélioration des structures précédentes permettant d'obtenir cette complexité pour les tas persistants.

On introduit le type *tas binomial enraciné*

```
type 'a rbin_heap = Empty | R of 'a * 'a bin_heap
```

Autrement dit, un tas binomial enraciné est soit vide, soit un couple composé d'un élément unique (la racine) et d'un tas binomial. On maintient l'invariant que le maximum d'un tas non vide est à la racine.

Question 2.14. Écrire les fonctions d'insertion d'un élément, de recherche de maximum et de suppression d'un élément avec ce nouveau type.

Tournez la page S.V.P.

Circuits logiques efficaces pour l'addition des entiers

L'addition de nombres entiers naturels est un des éléments essentiels de l'architecture de tout processeur, car beaucoup d'opérations sur les entiers ou les nombres flottants s'y ramènent. Dans cette partie, on s'intéresse à l'implémentation en tant que circuit logique de l'addition et son efficacité.

Pour éviter l'ambiguïté entre disjonction et addition, on notera $a + b$ la somme de deux nombres, même lorsqu'il s'agit de valeurs dans $\{0, 1\}$, et $a \vee b$ le **ou logique**. Par symétrie on notera alors $a \wedge b$ le **et logique**. La **négation** du booléen a est notée $\overline{a}$. On note également $a \oplus b$ le **ou exclusif**, c'est-à-dire le booléen qui vaut 1 si et seulement si $a \neq b$.

Si $a_0, \ldots, a_{n-1} \in \{0, 1\}$, on note $\overline{a_{n-1} \ldots a_0}^2 = \sum_{i=0}^{n-1} a_i 2^i$ l'entier représenté par le mot binaire $a_{n-1} \ldots a_0$.

Tous les circuits logiques considérés dans cette partie ont un ensemble d'entrées, un ensemble de sorties et un assemblage de portes. Les portes logiques considérées sont la porte unaire NOT, réalisant la **négation**, et les portes binaires AND, OR et XOR réalisant respectivement le **et logique**, le **ou logique** et le **ou exclusif**.

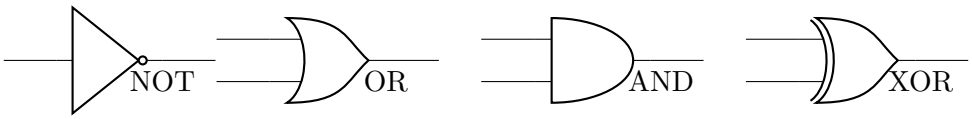


Figure 5: Les quatre types de portes logiques considérées

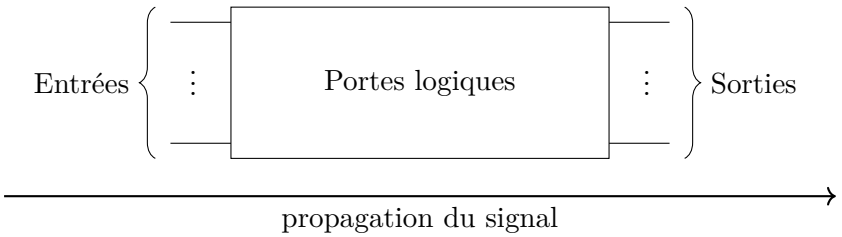


Figure 6: Forme générale des circuits considérés

Les circuits logiques qu'on étudie sont des assemblages de portes logiques dont certains fils sont appelés des entrées et d'autres des sorties. On suppose que les circuits ne présentent ni cycles, ni connexions directes entre deux sorties ou entre deux entrées. De plus, une entrée ne peut pas être reliée à la sortie d'une porte logique et une sortie ne peut pas être reliée à une des entrées d'une porte logique. On peut alors orienter schématiquement un tel circuit de la gauche vers la droite et voir le signal comme se propageant dans cette direction. Cette vision schématique est présentée dans la figure 6.

Stabilisation de circuit

D'un point de vue matériel, la sortie d'une porte logique met un certain temps avant de se stabiliser sur la valeur correcte lorsque ses entrées changent. Si la sortie d'une porte logique est l'entrée d'une seconde porte logique, il est nécessaire d'attendre que la première porte se stabilise pour que la seconde puisse se stabiliser.

Pour simplifier, on va supposer ici que toutes les portes logiques que l'on considère se stabilisent au bout d'un temps T_s.

On appelle *hauteur logique* la longueur maximale, en nombre de portes logiques du type de celles présentes dans la figure 5, d'un chemin reliant une entrée du circuit à une de ses sorties. On admet qu'un tel chemin existe toujours dans les circuits considérés. Si un circuit est de hauteur logique h, on peut considérer qu'il s'est stabilisé au bout d'un temps hT_s.

Question 3.1. On considère ici $T_s = 50ps = 5.10^{-11}s$, sachant que l'horloge d'un processeur est à 1 Ghz, c'est-à-dire un milliard de cycles par seconde, en déduire un majorant de la hauteur logique pour un circuit se stabilisant en un cycle d'horloge pour ce processeur.

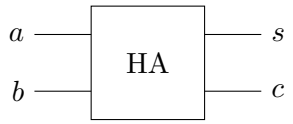
Pour simplifier les constructions qui vont suivre, on considère des portes logique n-aires.

Question 3.2. Déterminer une implémentation de hauteur logique minimale, qu'on précisera, d'une porte logique AND n-aire à l'aide de portes AND binaires.

Additionneur à propagation de retenue

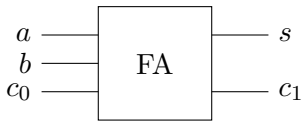
Pour réaliser une addition de nombres entiers sur n bits, on va reproduire l'addition posée : on ajoute bit par bit en partant du bit de poids faible et en propageant les retenues.

Question 3.3. Écrire un circuit logique HA, pour *half-adder* ou *demi-additionneur*, ayant deux entrées a et b et deux sorties s et c tel que $\overline{cs}^2 = a + b$. Le circuit devra être de hauteur logique 1.



Tournez la page S.V.P.

Question 3.4. En déduire un circuit logique FA, pour *full-adder* ou *additionneur complet*, ayant trois entrées a, b et c_0 et deux sorties s et c_1 tel que $\overline{c_1 s}^2 = a + b + c_0$. Le circuit devra être de hauteur logique 3.



Question 3.5. A l'aide d'*additionneurs complets* et d'une propagation des retenues, réaliser un circuit permettant de calculer la somme de deux nombres sur n bits. C'est-à-dire que ce circuit aura des entrées $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, c_0$, des sorties $s_0, \ldots, s_{n-1}, c_n$ et il est tel que

$$\overline{c_n s_{n-1} \ldots s_0}^2 = \overline{a_{n-1} \ldots a_0}^2 + \overline{b_{n-1} \ldots b_0}^2 + c_0$$

Question 3.6. Déterminer la hauteur logique du circuit précédent. En déduire un majorant sur le nombre de bits pour qu'une addition puisse s'effectuer en un cycle sur un processeur de fréquence d'horloge 1Ghz avec $T_s = 50ps$.

Additionneur à anticipation de retenue

On considère ici des entiers $a, b < 2^n$ qui s'écrivent sur n bits $\overline{a_{n-1} \ldots a_0}^2$ et $\overline{b_{n-1} \ldots b_0}^2$, ainsi qu'une retenue initiale $c_0 \in \{0, 1\}$.

On note s_i le ième bit de $a + b + c_0$ et c_{i+1} la valeur de la retenue éventuelle issue du calcul de s_i.

La hauteur logique importante du circuit précédent est liée au fait que le calcul de s_i et c_{i+1} dépend de $a_i + b_i + c_i$, et c_i n'est connu qu'une fois la valeur de s_{i-1} déterminée.

Connaissant a_i et b_i, on peut toutefois en déduire deux nouveaux booléens :

- p_i qui vaut 1 lorsque la somme $a_i + b_i + 1 = 2$, c'est-à-dire qu'on est dans un cas de pure propagation de la retenue ;
- g_i qui vaut 1 lorsque la somme $a_i + b_i$ génère une retenue.

Il est possible de calculer simultanément tous les couples (p_i, g_i), puis d'en déduire les c_i et enfin de calculer les s_i pour obtenir la somme. On obtient ainsi une architecture *en trois étages* présentée sur la figure 7.

Question 3.7. Exprimer p_i et g_i en fonction de a_i et b_i.

En déduire un circuit de hauteur logique 1 réalisant le bloc de pré-calcul.

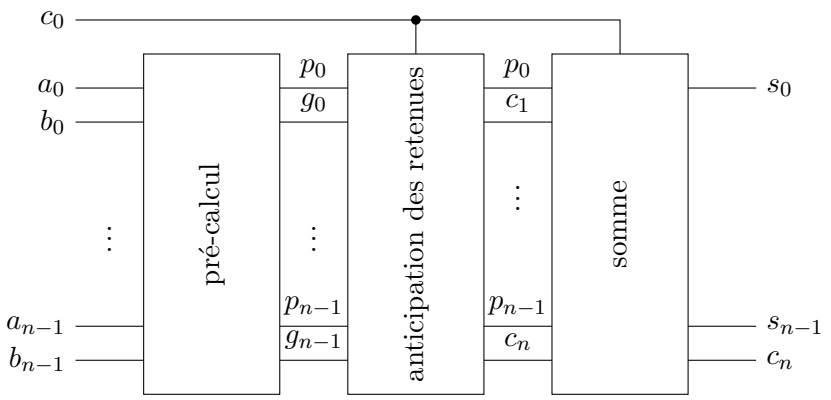


Figure 7: L'architecture à trois étages d'un additionneur à anticipation de retenues

Question 3.8. Exprimer s_i en fonction de p_i et c_i.

Question 3.9. En déduire un circuit de hauteur logique 1 réalisant le bloc somme.

Question 3.10. Exprimer c_{i+1} en fonction de c_i, p_i et g_i.

Question 3.11. En déduire une première implémentation naïve du bloc d'anticipation des retenues de hauteur logique $2n$.

Question 3.12. Exprimer, en forme normale disjonctive, c_{i+1} en fonction de $g_i, \ldots, g_0, p_0, \ldots, p_i$ et c_0.

Question 3.13. En déduire une implémentation plus efficace du bloc d'anticipation des retenues. On pourra utiliser des portes n-aires pour simplifier l'écriture du circuit.

Question 3.14. Déterminer la hauteur logique du circuit global et en déduire un majorant sur le nombre de bits pour qu'une addition puisse s'effectuer en un cycle sur un processeur de fréquence d'horloge 1Ghz avec $T_s = 50ps$.

En pratique, l'intensité du signal et la consommation des portes logiques ne permettent pas de réaliser de tels branchements et un additionneur par anticipation de retenue est en fait constitué de $\log_2 n$ étages d'anticipations, ce qui permet tout de même un gain important par rapport à un additionneur à propagation de retenue.

* *
*