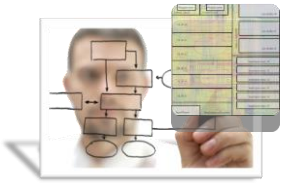
Utilisation des outils Lustre

A. Prise en main

Dans ce premier exercice, on va analyser l'exemple fourni en cours : `edge`. Ce programme détecte un front montant, c'est à dire le passage d'une variable booléenne de la valeur `false` à la valeur `true`.

1. Spécification Lustre



Nous en rappelons la spécification Lustre :

```
node EDGE (b : bool) returns (edge : bool);
let
      edge = false -> b and not pre b;
tel
```

Dans l'expression qui définit `EDGE`, on retrouve plusieurs opérateur Lustre dont nous rappelons la signification:

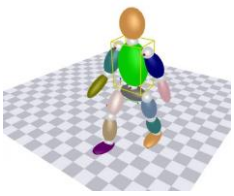
- Le "et logique" (`and`) et la négation (`not`), la constante booléenne `false`.
- La flèche (`->`) permet de distinguer la valeur initiale du flot (en partie gauche) des autres valeurs (en partie droite). La première valeur du flot `edge` est donc `false`, les valeurs successives sont définies par l'expression " `b and not pre b` ".
- L'opérateur `pre` (pour précédent) permet de faire référence au "passé" : la valeur à l'instant précédent.

Au final, le noeud EDGE définit le flot modélisé par l'équation suivante :

```
EDGE(1) = false
∀t > 1,EDGE(t) = b(t) and not b(t - 1)
```

Écrire le noeud `EDGE` dans un fichier `edge.lus`

2. Simulation Lustre

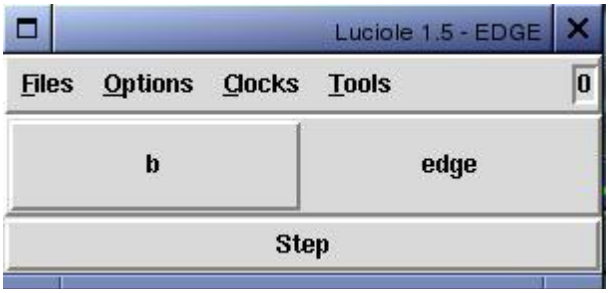


Dans cette partie, nous nous intéressons à la simulation de programmes Lustre à l'aide du simulateur graphique **luciole**. Luciole requiert le nom du fichier `.lus` à lire, et le nom du noeud à simuler.

```
luciole edge.lus EDGE
```

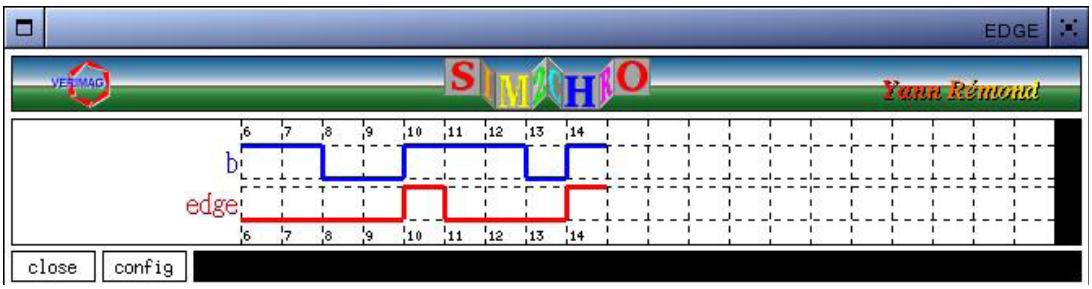
Cette commande ouvre une fenêtre de simulation. Elle présente une série de boutons correspondant aux entrées/ sorties du noeud. Dans le cas du noeud `EDGE`, on dispose d'un bouton pour l'entrée `b` et une "lampe" pour la sortie `edge` (Note : une lampe est un label qui s'affiche en rouge quand `edge` est vrai, et en gris quand `edge` est faux).

Par défaut, avec cette interface, cliquer sur le bouton "`b`" provoque un cycle de calcul avec `b` vrai, cliquer sur le bouton "`Step`" provoque un cycle de calcul avec `b` faux. Le résultat est renvoyé sur la lampe `edge`.



3. Chronogrammes

La commande "Tools → sim2chro" ouvre l'outil de visualisation de chronogramme. L'évolution des variables `b` et `edge` est alors automatiquement visualisée sous forme de chronogramme, indexé par les entrées du noeud.



<u>Modes auto-step et modes compose</u> :
Par défaut, l'interface d'exécution de luciole est en mode « *auto-step* », c'est-à-dire que le noeud est activé dès qu'on active sur une entrée booléenne ou sur `Step`.
Avec la commande "Clocks → Compose", on passe en mode *compose* : les entrées booléennes deviennent des interrupteurs qu'on peut activer/désactiver sans provoquer un pas de calcul. Le pas de calcul est déclenché en appuyant sur le bouton `Step`. Ce mode est nécessaire pour des programmes qui ont plusieurs entrées, pour pouvoir composer des états où plusieurs entrées sont vraies simultanément.

Horloge temps-réel :
Utiliser le bouton `Step` peut devenir gênant, on peut alors utiliser une horloge "temps-réel".
Le menu "Clocks →Real time clock" permet d'activer/désactiver le mode temps-réel. Dans ce mode, le pas de calcul est automatiquement déclenché à intervalles réguliers.
La période de l'horloge peut être modifiée avec "Clocks → Change period". Elle est exprimée en milli-secondes.
Attention ! *L'aspect temps-réel est relatif et dépend de la plate-forme utilisée. On utilise un système multitâches multi-utilisateur qui n'offre aucune garantie temps-réel. En pratique, si la machine est un peu trop chargée, le simulateur aura du mal à soutenir une période inférieure à quelques ms.*

---

4. Compilation en C

La compilation d'un programme Lustre se déroule en plusieurs phases :
- La pré-compilation transforme un programme Lustre en programme "Lustre noyau", aussi appelé code expansé. Cette phase est réalisée par la commande :

```
lus2ec edge.lus EDGE
```

Elle produit un fichier `EDGE.ec`, qui dans ce cas précis est pratiquement très proche du programme initial `edge.lus`.

- La compilation proprement dite transforme un programme `ec` en programme C. La commande est :

```
ec2c EDGE.ec -v
```

L'option `-v` permet de passer en mode "verbeux" : des informations additionnelles sur le déroulement de la compilation sont affichées (Note : tous les outils Lustre disposent d'une option -v).

Le résultat de la compilation est un fichier `EDGE.c` qui contient le noyau réactif du programme, ainsi que le fichier `EDGE.h` qui contient les informations nécessaires à l'utilisation du noyau.

- La compilation en C ne produit que le noyau réactif du système, c'est normalement à l'utilisateur d'écrire un programme principal qui se charge de l'acquisition des entrées et de la visualisation des sorties. Le compilateur `ec2c` propose une option `-loop` qui produit, en plus du noyau réactif, un programme principal standard. La commande :

```
ec2c EDGE.ec -loop -v
```

produit `EDGE.h`, `EDGE.c` plus un fichier `EDGE_loop.c` qui contient une fonction main standard. Pour des programmes simples (comme `edge`) ce programme principal peut être utilisé tel quel.

- Pour obtenir un exécutable, il faut finalement utiliser un compilateur C, par exemple `GCC` sur GNU/Linux.
La commande :

```
gcc EDGE.c EDGE_loop.c -o EDGE
```

compile les deux fichiers C, et produit l'exécutable `EDGE`. Le programme principal standard est en fait très rudimentaire : l'utilisateur doit taper au clavier, une par une, les entrées du programme.

- Lux permet d'enchaîner toutes ces phases automatiquement. Il gère en particulier la phase de liaison avec les différentes bibliothèques nécessaires à la construction de l'exécutable. La commande :

```
lux edge.lus EDGE
```

enchaîne automatiquement toutes les phases et produit un exécutable EDGE.

---

5. Inclusion du code C dans un programme Lustre

Dans cette section, on s'intéresse à l'inclusion de code C dans un programme Lustre. Une telle action est souvent nécessaire pour interfacer le code avec d'autres portions logicielles, dont des drivers.

La construction d'une telle application est un peu plus complexe, du fait du modèle de compilation des applications C qui impose à l'utilisateur de définir les dépendances entre les différents modules à compiler puis lier.

On cherche à interfacer le code C suivant, défini dans gett.c :

```
#include <stdio.h>

int gett (int a)
{
        usleep(200000);

        return a;
}
```

Et comme fichier prototype (gett.h) :

```
int gett(int a);
```

avec le code Lustre suivant :

```
function gett (val : int) returns(num : int);

node c_code (a : int) returns (x : int);
let
        x = gett(a);
tel
```

On note la définition de la fonction gett *qui indique à* Lustre *l'existence de* gett *sans pour autant la définir.*

Pour compiler ce programme, il faut au préalable fournir la définition C du code que l'on souhaite inclure. Le fichier Lustre ayant comme point d'entrée c_code, vous définirez le fichier c_code_ext.h fournissant les définitions au code extérieur, il a pour contenu:

```
#include "gett.h"
```

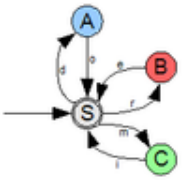
Pour compiler ce programme, il est nécessaire de suivre la démarche suivante:

```
lustre c_code.lus c_code
poc -loop c_code.oc
gcc -c gett.c
gcc -c c_code.c
gcc c_code_loop.c gett.o c_code.o -o c_code
```

On peut alors exécuter le programme `c_code`.

---

6. Compilation en automate



Une solution alternative pour la compilation d'exécutable consiste à générer le noyau réactif du noeud Lustre sous la forme d'un automate à états finis :
- La phase de pré-compilation est la même :

```
lus2ec edge.lus EDGE
```

produit le fichier `EDGE.ec`. La commande :

```
ec2oc EDGE.ec
```

produit un automate, dans un format particulier appelé oc (dans un fichier `EDGE.oc`). On génère le code C correspondant avec la commande :

```
poc EDGE.oc
```

On obtient alors, comme avec `ec2c` un fichier `EDGE.c` et un fichier `EDGE.h`.

- La commande `poc` dispose aussi d'une option `-loop`. On peut donc enchaîner les commandes suivantes pour obtenir un programme exécutable `edge` :

```
poc EDGE.oc -loop
gcc EDGE.c EDGE_loop.c -o EDGE
```

Cette méthode de génération de code est assez proche de la précédente, à ceci près qu'un automate fournit un modèle pertinent pour raisonner sur les exécutions possibles du programme, et ainsi analyser son espace d'états.

---

7. Visualisation de l'automate

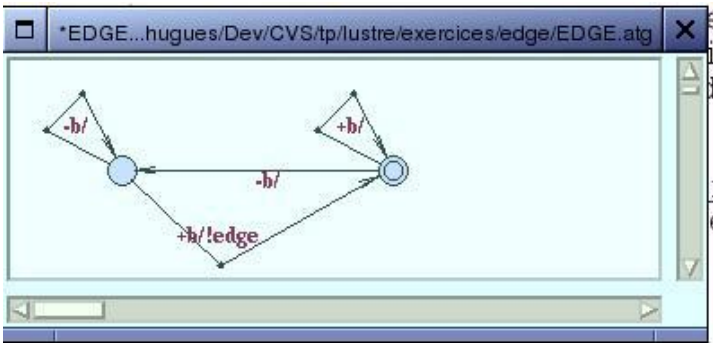
Si on dispose de l'outil **autograph**, on peut visualiser assez simplement l'automate, et voir s'il correspond bien à ce qu'on désirait. Pour cela, on peut utiliser `lus2atg` :

```
lus2atg edge.lus EDGE
```

Cette commande produit un fichier `edge.atg` directement exploitable par autograph. On tape alors la commande :

```
atg EDGE.atg
```

qui ouvre l'explorateur d'automate. L'exploration se fait à la souris, choisir "Placing → Explore" puis interagir avec le diagramme pour faire apparaître les différents états. On obtient alors le graphique suivant :



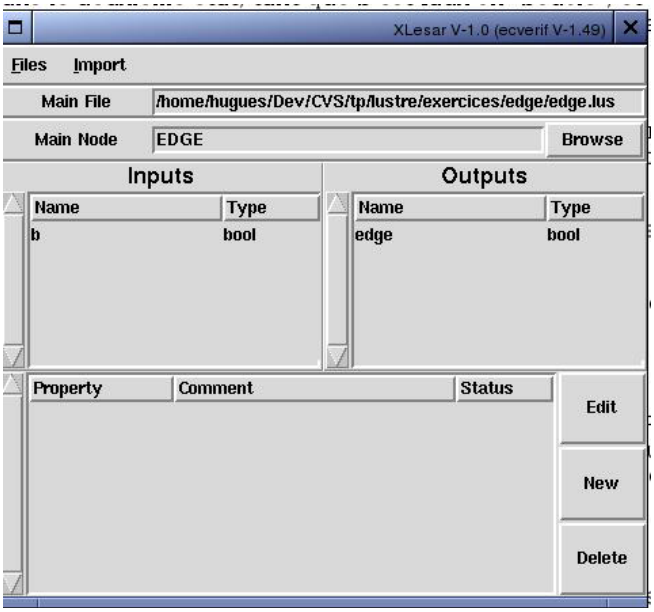
Sur cette figure, on remarque les choses suivantes :

- Dans l'état initial (double cercle), si `b` est vrai, on reste dans cet état, et si `b` est faux on passe à l'état suivant.
- Dans le deuxième état, tant que `b` est faux on "boucle", et si `b` est vrai on "émet" `edge` (c'est-à-dire que `edge` est vrai), et on retourne dans l'état initial.

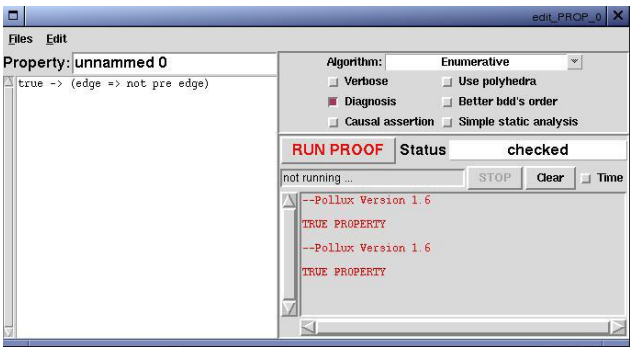
---

8. Vérification de propriétés

Lustre dispose d'un outil de vérification formelle de propriétés. L'utilisateur exprime un ensemble de propriétés que doit vérifier le programme, et l'outil indique si ces propriétés sont satisfaites, ou fournit un contre-exemple.
On utilise ici l'outil **xlesar** : une interface graphique du vérificateur `lesar`. Pour lancer l'outil, taper : `xlesar`.



Sélectionner le fichier et le noeud à analyser (commande `Browse`). Les entrées/sorties du noeud sont affichées.
La partie basse de la fenêtre permet de rentrer les différentes propriétés que l'on souhaite vérifier (commandes New , puis sur Edit). Cette dernière commande ouvre une fenêtre d'édition de propriété. La partie gauche est l'éditeur de propriété proprement dit, la partie droite permet de lancer la vérification.



<u>Une propriété triviale (`true`)</u> : Le point important est que la propriété doit être exprimée sous la forme d'une expression booléenne Lustre. Par défaut, cette définition est `true`, et est donc vraie. On peut tout de même essayer de lancer le vérificateur en appuyant sur « RUN PROOF ». On obtiendra naturellement (dans la fenêtre de dialogue) le résultat "`TRUE PROPERTY`".

<u>Une propriété vraie</u> : On va maintenant exprimer et vérifier une propriété plus complexe. On cherche à évaluer la propriété suivante : *« La sortie edge ne peut pas être vraie à deux instants consécutifs »*.
Pour utiliser le vérificateur, on doit tout d'abord traduire cette propriété en une expression Lustre. On va simplement dire que, initialement, la propriété est toujours vérifiée, puis à chaque instant, si `edge` est vraie alors `edge` était faux à l'instant précédent. En utilisant l'opérateur "implication logique" (=>), cela donne :

```
true -> (edge => not pre edge)
```

On peut alors lancer le prouveur, en utilisant l'option « Verbose » pour avoir des informations sur le déroulement de la preuve. On peut essayer les différents algorithmes proposés : Enumerative, Symbolic forward et Symbolic backward. On doit évidemment obtenir la même réponse pour chaque algorithme.

<u>Complexité de la preuve</u> : Grâce au mode verbeux, on peut se faire une idée de la "complexité" de la preuve : avec l'algorithme énumératif, le nombre d'états et de transitions sont une bonne mesure de la complexité de la propriété (4 états et 8 transitions pour cet exemple). En symbolique, c'est le nombre d'itérations qui mesure la complexité (2 pas de calculs en mode "forward" et en mode "backward").

Un autre exemple de propriété vraie est *«edge ne peut être vrai que si b est vrai »*, ce qu'on traduit simplement par :

```
edge => b
```

Cette propriété est beaucoup plus évidente que la précédente : elle est, pour ainsi dire, écrite dans le programme.
On peut d'ailleurs observer cette simplicité en regardant la complexité de la preuve : 2 états et 2 transitions en énumératif, 2 pas de calcul en "forward", et 0 pas de calcul en mode "backward".

<u>Une propriété fausse</u> : On va maintenant voir ce qui se passe pour une propriété fausse, par exemple : *« si b est vrai, alors edge est vrai »*. Ce qu'on traduit par :

```
b => edge
```

Si on n'utilise aucune option, le prouveur se contente de répondre `FALSE PROPERTY`.
Il faut utiliser l'option « Diagnosis » pour obtenir un contre exemple :

```
DIAGNOSIS:
--- TRANSITION 1 ---
b
```

L'interprétation est la suivante : dès la première réaction du programme, si `b` est vrai, la propriété n'est pas satisfaite. Ce résultat est attendu : la sortie est toujours fausse à l'instant initial. Il faut donc modifier la propriété : *« Sauf à l'instant initial, si b est vrai, alors edge est vrai »*. Traduite en Lustre, cette propriété devient :

```
true -> (b => edge)
```

Là encore on va obtenir un résultat négatif, avec un nouveau contre-exemple :

```
DIAGNOSIS:
--- TRANSITION 1 ---
b
--- TRANSITION 2 ---
b
```

qui stipule que la propriété est fausse si b est vrai deux fois de suite. Mieux vaut alors se rentre à l'évidence : la propriété est bien totalement fausse.

B. Exercices

Exercice 1 : Initiation

1. Écrire et simuler un noeud `ABS` qui prend un flot d'entrée réel et renvoie sa valeur absolue.
2. Tester le programme suivant :

```
node Test (x:int) returns (z,t:int);
let
      z =  0 -> 1 -> 2;
      t =  0 -> pre(1 -> 2);
tel
```

3. Écrire un noeud `Osc` qui génère le signal (true, false, true, false, ...).
4. Écrire un noeud `Osc2` qui génère le signal (true, true, false, false, true, true, false, ...).
5. Écrire un noeud `Compteur` qui prend en entrée un flot booléen `Reset`, en sortie un flot d'entiers `N`. A chaque instant le compteur incrémente la sortie `N` sauf quand `Reset` est vrai. Dans ce dernier cas `N` est remis à zéro. La première fois que le compteur est appelé et si il n'y a pas de remise à zéro alors `N` doit valoir `1`.
6. Écrire un noeud `Compteur2` qui reprend les caractéristiques de `Compteur` mais va deux fois plus lentement (utiliser les opérateurs `current` et `when`).
7. Différence entre le `when` et le `if . then . else .` : Testez le programme suivant avec luciole et comprenez ce qui ce passe...

```
node ChangeCompteur(RESET,C :bool) returns(N :int) ;
let
      N = if C then Compteur(RESET) else Compteur2(RESET) ;
tel
```

8. Vérification sur programme :
   Écrire un noeud qui formalise la proposition sur le noeud `Osc2` suivante : " Quand `Osc2` produit un front montant alors à l'instant suivant le signal est encore vrai. "

Exercice 2 : Additionneur

On veut implémenter un additionneur 4 bits à l'aide d'additionneurs 1 bits.
L'additionneur 1 bit va prendre en entrée les 2 bits à additionner `a` et `b` et la retenue de l'étage précédent `c_i`. Il donne en sortie le résultat `s` et la retenue `c_o`.

1. Implémenter l'additionneur 1 bit en Lustre (nœud `ADD1`)
2. Implémenter l'additionneur 4 bits (nœud `FIRST_ADD4`) à l'aide de `ADD1` . Cet additionneur va prendre les 8 bits `a_0`, ..., `a_3`, `b_0`, ..., `b_3` en entrée et fournir les 4 bits `s_0`, ..., `s_3` et le `carry` final en sortie.
3. Simuler à l'aide de luciole (horloge en mode *compose*)

On va voir maintenant la possibilité offerte par Lustre d'utiliser des tableaux et des nœuds récursifs. Ce n'est qu'un artifice facilitant la vie du programmeur car le compilateur transforme les tableaux en autant de variables qu'il y a d'éléments et la récursivité se traduit par l'instanciation d'autant de nœuds que nécessite la récursion.

`A` et `B` vont être des tableaux de 4 booléens déclarés par des expressions du type `A: bool^4`. Les éléments de ce tableau sont `A[0]`, `A[1]`, `A[2]` et `A[3]`. Le nœud pour l'additionneur va devenir `ADD4`.
La notation `A[1..3]` représente la « tranche » des éléments du tableau `A` constituée des éléments `A[1]`, `A[2]` et `A[3]`.

L'équation :
```
(S[1..3], C[1..3]) = ADD1(A[1..3], B[1..3], C[0..2]);
```
est une notation pour représenter les 3 équations :
```
(S[1], C[1]) = ADD1(A[1], B[1], C[0]);
(S[2], C[2]) = ADD1(A[2], B[2], C[1]);
(S[3], C[3]) = ADD1(A[3], B[3], C[2]);
```

Rappel : les équations logiques donnant `s` et `c_o` à partir de `a`, `b` et `c_i` sont les suivantes :

$s = a \oplus b \oplus c\_i$

$c\_i_0 = a.b + b.c\_i + a.c\_i$

Exercice 3 : Contrôleur de souris

Implémenter en Lustre le contrôleur de souris (nœud `Souris`) qui retourne `double` si on a reçu 2 `click` en moins de 4 `top`, sinon `single`.
Indication : Utiliser un compteur pour compter le nombre de `top`.

Exercice 4 : Chien de garde

Un chien de garde est un dispositif qui permet de gérer le temps de réponse d'un système et de déclencher éventuellement une alarme si une échéance n'est pas respectée. Il est utilisé pour détecter automatiquement une anomalie du logiciel et réinitialiser le processeur.

Dans un premier temps, on va considérer un nœud à 3 entrées : deux commandes : `set` et `reset`, et une variable booléenne signalant l'occurrence de l'échéance : `deadline`. Il a une seule sortie signalant le dépassement de l'échéance : `alarm`. L'alarme se produit quand le watchdog est actif (a reçu la commande `set`) et que l'événement `deadline` se produit. L'état *actif* ou *passif* du watchdog est identifié par une variable booléenne `is_set`. L'alarme se déclenchera donc quand `is_set` et `deadline` seront vrais. `is_set` est initialement dans un état vrai ou faux suivant que `set` est vrai ou faux. Elle reste identique à elle même tant que `set` ou `reset` ne sont pas vrais.

1. Implémenter le nœud `WD1` en tenant compte du fait que `set` et `reset` ne peuvent pas être présents simultanément.
2. Simuler le nœud avec luciole.

La 2ème version du watchdog va recevoir les mêmes commandes `set` et `reset` mais l'alarme sera déclenchée si l'événement `reset` ne s'est pas produit avant un certain temps `delay` (mesuré en nombre de cycles de l'horloge de base) après l'occurrence de `set`. Le temps écoulé depuis l'occurrence de `set` est mesuré par un registre `remain` dont le contenu est décrémenté à chaque cycle de l'horloge. On va réutiliser le nœud `WD1`. `deadline` sera produit quand le registre atteint 0. On réutilisera aussi le nœud `EDGE` du détecteur de front (vu en cours) pour détecter le fait que `remain` atteint 0.

1. Implémenter le nœud `WD2`.
2. Simuler le nœud avec luciole.

Dans la dernière version, on va exprimer le délai suivant une échelle de temps correspondant à l'occurrence d'un signal `time_unit`. On va reprendre `WD2` en utilisant une horloge appropriée générée par la variable booléenne `clock`.

1. Implémenter le nœud `WD3`.
2. Simuler le nœud avec luciole.

Exercice 5 : Délai

L'opérateur `DELAY` va prendre en entrée un entier `d` (une constante) et un flot booléen `X` et va retourner une version « retardée » du flot, c'est-à-dire un flot booléen `Y` tel que $y_n = x_{n-d}$ pour tout $n>d$. On suppose que y_n = `false` pour tout $n \leq d$.

1. Dans un premier temps, on peut utiliser une variable auxiliaire `A`, tableau booléen de dimension `d` tel que $A[i]_n = X_{n-i}$. Implémenter le nœud `DELAY` en Lustre
2. Implémenter le nœud `MAIN_DELAY`, nœud principal qui appelle le nœud `DELAY` pour `d=3` par exemple.