Programmation de systèmes hétérogènes CPU/FPGA avec Lustre

Emile GUILLAUME

Verimag Research Report n^o TR-2025-1

28/08/2025

Reports are downloadable at the following address
http://www-verimag.imag.fr

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UGA
Bâtiment IMAG
Université Grenoble Alpes
700, avenue centrale
38401 Saint Martin d'Hères
France

Programmation de systèmes hétérogènes CPU/FPGA avec Lustre

Emile GUILLAUME

28/08/2025

Abstract

Ce rapport présente le développement d'une chaîne de compilation pour traduire des programmes écrits en Lustre vers des descriptions matérielles en Verilog, afin de cibler des architectures CPU/FPGA. Le travail décrit l'étude des outils existants, l'identification des contraintes liées à la synthèse matérielle, ainsi que la conception d'un prototype capable de générer du code FPGA à partir d'un sous-ensemble de Lustre. Les résultats montrent la faisabilité de l'approche et ouvrent des perspectives pour l'exploitation du parallélisme implicite du langage dans les systèmes embarqués critiques.

Keywords: Lustre, CPU/FPGA, programmation hétérogène, synthèse matérielle, Verilog, systèmes embarqués

Reviewers: Erwan JAHIER, Bruno FERRES

Notes: Rapport réalisé dans le cadre du stage assistant ingénieur à Verimag.

How to cite this report:

```
@techreport {TR-2025-1,
  title = {Programmation de systèmes hétérogènes CPU/FPGA avec Lustre},
  author = {Emile GUILLAUME},
  institution = {{Verimag} Research Report},
  number = {TR-2025-1},
  year = {}
}
```

Contents

1 Introduction et objectifs du stage

Lustre [1] est un langage synchrone développé au sein du laboratoire Verimag, historiquement conçu pour la modélisation de systèmes embarqués temps réel. Ce langage déclaratif repose sur la notion de flot de données et permet de décrire des relations entre variables qui restent valides à chaque instant logique de l'exécution. Il se distingue par son caractère déterministe, son adéquation aux systèmes critiques, et sa capacité à exprimer clairement le parallélisme.

Dans le cadre de ce stage, l'objectif principal a été d'explorer la possibilité de compiler automatiquement du code écrit en Lustre vers un langage de description matériel (HDL), en particulier Verilog [2], afin de cibler des architectures reconfigurables comme les FPGA. L'idée sous-jacente est de tirer parti du parallélisme implicite du langage Lustre pour produire des circuits numériques efficaces, tout en conservant la sémantique originale du programme.

Le travail a consisté dans un premier temps à étudier les outils existants, à identifier les contraintes liées à la compilation vers du matériel, puis à concevoir un prototype de chaîne de compilation traduisant un sous-ensemble de Lustre vers Verilog. Ce rapport technique détaille les étapes de cette démarche, les choix réalisés, les défis rencontrés ainsi que les perspectives ouvertes par ce travail.

2 Présentation de Lustre

Avant d'entrer dans le cœur du rapport, il est utile de rappeler brièvement les principaux éléments du langage Lustre v6, utilisé dans nos développements.

Lustre est un langage synchrone à flot de données, pensé pour la modélisation de systèmes réactifs et embarqués. Il repose sur une syntaxe déclarative, structurée autour de blocs appelés nodes (nœuds), qui représentent des fonctions déterministes.

Le langage prend en charge des types de base (`int`, `bool`, `real`), ainsi que des types énumérés définis par l'utilisateur via le mot-clé enum. La structure générale d'un programme s'appuie sur les blocs node, entourés des mots-clés let et tel.

Lustre offre plusieurs mécanismes pour exprimer la logique temporelle :

- `pre` permet d'accéder à la valeur précédente d'un flot ;
- `->` permet d'initialiser un flot avec une valeur particulière au premier pas ;
- `fby` permet de définir des séquences d'initialisation suivies d'un comportement régulier.

La gestion du temps repose également sur le concept d'horloges (clocks). Le mot-clé when permet de conditionner l'activation d'un flot à une certaine horloge. D'autres constructions comme merge ou current permettent de structurer la logique multi-clock et l'activation conditionnelle de sous-composants.

Des structures de contrôle comme if permettent d'exprimer des comportements dépendants de conditions ou de valeurs énumérées.

Enfin, Lustre-v6 introduit la gestion des tableaux ainsi que des nœuds récursifs lorsque le nombre d'itérations est connu, ce qui permet une meilleure expressivité et un traitement plus efficace des structures répétitives.

3 Mise en place des outils et d'une banque de test

Dans la première partie de ce stage, une première phase de lecture a permis de se familiariser avec les travaux existants sur la génération de circuits à partir du langage Lustre. Deux articles ont été étudiés à cette occasion. L'article de Rocheteau et Halbwachs [3] constitue une référence historique, proposant une méthode pour produire des circuits à partir de programmes Lustre. Le second, plus récent, est l'article de Winandy et al. [4] , qui s'inscrit dans une démarche similaire, en ciblant spécifiquement les architectures FPGA. Cependant, ce travail ne prend pas en charge certaines fonctionnalités clés de Lustre v6, comme les tableaux, ce qui limite son applicabilité.

Par ailleurs, ce stage s'inscrit dans le contexte des FPGA récents, avec pour objectif de pouvoir contrôler finement le processus de compilation. Cette maîtrise est essentielle pour envisager à terme la génération de code hétérogène déployé simultanément sur CPU et FPGA, ce qui explique l'intérêt d'intégrer directement les travaux réalisés dans le compilateur Lustre.

Après une phase d'apprentissage de la syntaxe du langage Verilog, nous avons réalisé plusieurs tests de traduction manuelle de petits programmes Lustre vers Verilog, afin d'en comprendre les mécanismes et les divergences sémantiques. Un exemple simple est celui de la détection de front montant (rising edge) :

3.1 Code Lustre :

```
node rising_edge(x : bool) returns (res: bool);
let
  res = false -> (x and not pre(x));
tel
```

3.2 Traduction Verilog correspondante :

```
module rising_edge (
  input wire veri_rst,
  input wire veri_clk,
  input wire x,
  output wire res
);
  reg  pre_x;
  assign res = veri_rst ? (1'b0) : ( x && !(pre_x));
  always @(posedge veri_clk) begin
    pre_x <= x;
  end
endmodule
```

Cet exemple met en évidence plusieurs différences fondamentales entre les deux langages. Tout d'abord, Lustre ne manipule pas explicitement de signaux d'horloge ou de remise à zéro, puisqu'il repose sur une abstraction de temps logique : chaque "tick" est implicite. En revanche, dans une implémentation matérielle en Verilog, il est nécessaire d'introduire un signal d'horloge (`veri_clk`) ainsi qu'un signal de reset (`veri_rst`) pour contrôler explicitement la progression du temps et l'initialisation des registres. Ces signaux sont donc ajoutés artificiellement pour assurer que le circuit généré se comporte correctement.

Par ailleurs, le comportement du `pre` (mémoire d'un pas de temps) doit être traduit manuellement à l'aide d'un registre et d'un bloc `always` déclenché sur le front montant de l'horloge (la syntaxe des registre en Verilog est assez lourde. Il faut donc générer le bon morceau de code pour chaque `pre`). Enfin, contrairement à Verilog où les calculs se font généralement avant le front d'horloge et sont capturés au moment du front d'horloge, en Lustre, le front logique déclenche l'ensemble des calculs synchrones dans un ordre déterministe, ce qui implique un effort d'adaptation dans la traduction.

Ces premiers tests nous ont permis d'identifier les exigences minimales d'une traduction correcte, en particulier concernant la gestion du temps logique, de l'état et de l'initialisation, qui seront centrales dans le développement du compilateur.

Pour valider la cohérence entre le programme Lustre d'origine et sa traduction en Verilog, nous avons dans un premier temps mis en place une chaîne de test simple basée sur les outils Icarus Verilog (compilation et simulation Verilog) et GtkWave (visualisation de signaux). Cette approche permettait de simuler les modules traduits en Verilog et d'observer graphiquement l'évolution de leurs signaux au cours du temps (Figure 1). Toutefois, bien que pratique pour les premières vérifications, cette méthode s'est rapidement révélée insuffisante pour une validation systématique, car elle repose essentiellement sur une inspection visuelle et manuelle des résultats, ce qui est peu rigoureux et impossible à automatiser à grande échelle.

Afin de passer à une méthode de test plus automatisée et fiable, nous nous sommes tournés vers Verilator, un outil de simulation Verilog capable de générer un simulateur en C++ à partir d'un module Verilog, permettant ainsi une interaction directe avec le module simulé via un testbench écrit en C++ données à Verilator en complément du module. Grâce à cette approche, il devient possible d'automatiser les tests de

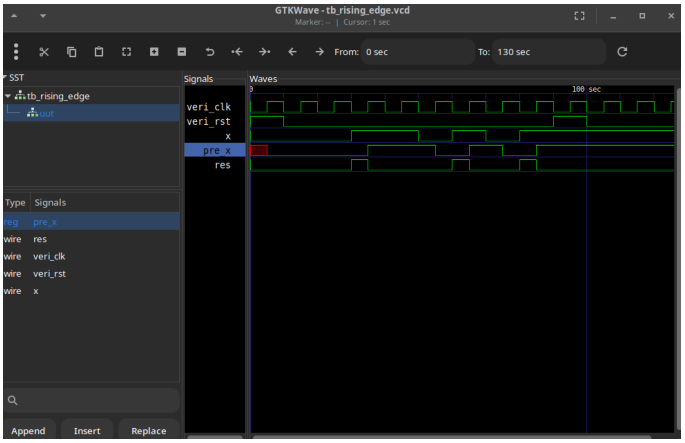

Figure 1: Exemple d'exécution de icarus Verilog et gtkwave sur rising edge

manière plus fine, en comparant les sorties attendues et les sorties obtenues, sans passer par une visualisation graphique.

Dans ce cadre, nous avons généré des testbenchs en C++ qui reproduisent des séquences d'entrées connues, en capturant les sorties du module Verilog simulé. Les résultats produits sont ensuite enregistrés dans un fichier au format RIF (Reactive Input Format), un format d'échange utilisé dans l'écosystème Lustre, spécifiquement conçu pour décrire des scénarios d'exécution synchrones.

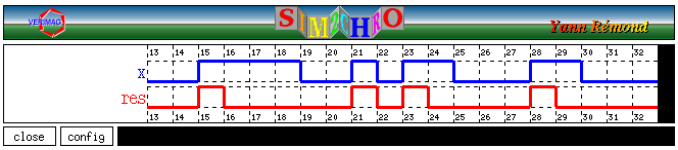

Figure 2: Exemple d'exécution de rising edge avec le testbench C++ sous sim2chro grâce au format .rif

Le choix du format RIF permet de tirer parti d'outils existants (Figure 2), comme Lurette [5], qui peut alors être utilisé pour comparer les traces de comportement du programme Lustre initial et du module Verilog généré. Cette étape de validation constitue un point clé du projet : elle nous permet de vérifier que la traduction ne modifie pas le comportement fonctionnel du programme et respecte bien les spécifications initiales. Grâce à cette stratégie, nous avons mis en place une boucle de test automatisée, reproductible, et adaptée à une démarche de prototypage rapide.

Lurette est donc utilisé dans un but de validation d'un programme Verilog, en comparant sa sortie avec celle du programme lustre de base (Figure 3).

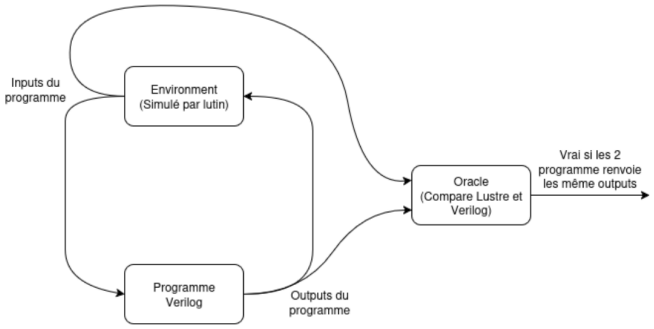

Figure 3: Processus de validation d'un test

Comme le montre le schéma, nous avons un environnement aléatoire géré par Lutin [6] qui est un langage servant à décrire le comportement d'un environnement.

Dans le cadre de Lurette, la vérification repose sur un Oracle, c'est-à-dire un composant chargé de vérifier automatiquement si la sortie du programme testé correspond à celle attendue.

Concrètement, notre programme Verilog produit ses sorties à partir des entrées générées par Lutin. Ces mêmes entrées sont envoyées à la version Lustre du programme, dont les sorties servent de référence pour l'Oracle. Celui-ci compare les deux jeux de sorties : s'ils sont identiques, le test est réussi ; sinon, il échoue. Le but étant ainsi de répéter ce procédé sur un grand nombre de tests afin d'avoir une base de confiance solide.

4 Compilateur Lustre vers Verilog

Dans la deuxième partie du stage, on a ajouté une option de compilation de Lustre vers Verilog. L'idée est de pouvoir générer directement du code matériel (HDL) à partir d'un programme Lustre.

Pour commencer, il a fallu choisir où se placer dans la chaine de compilation (Figure 4) pour faire le compilateur. Ainsi, nous avons choisie de nous placer au niveau de la représentation intermédiaire appelée LIC (Lustre Internal Code). C'est une version transformée du programme, représentée sous forme d'arbre syntaxique. Elle sert de base aux différentes étapes de compilation. Le choix de cette version du code permet de ne pas perdre les informations concernant le parallélisme du code de base. En effet, le SOC (Sequential Object Code) correspond à la transformation finale de Lustre en code séquentiel, ce qui n'est pas adapté à un circuit dont les performances reposent sur son parallélisme.

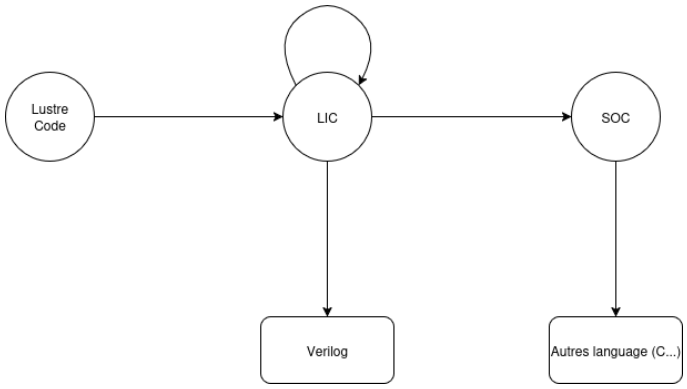

Figure 4: Chemins de compilation du Lustre

La Première étape a donc été de repérer la structure correspondant aux nœuds Lustre (les unités principales du langage, équivalentes à des fonctions avec de la mémoire) dans ce LIC. Plutôt que de tout détailler d'un coup, les éléments importants seront expliqués au fur et à mesure de l'implémentation.

La structure représentant un Noeud est sous cette forme :

```
node_exp = {
 node_key_eff : node_key;
 inlist_eff   : var_info list;
 outlist_eff  : var_info list;
 loclist_eff  : var_info list option;
 def_eff      : node_def;
 has_mem_eff  : bool;
 is_safe_eff  : bool;
 lxm          : Lxm.t;
}
```

La première étape de la génération consiste à récupérer les informations de base du nœud Lustre :

- son nom,

- ses entrées,
- ses sorties,
- ses variables locales.

Plusieurs fonctions s'occupent de ça dans le code :

- `init_info_vars` analyse les types des variables, notamment pour détecter les types énumérés (qui seront traduits en define Verilog),
- `head_module_verilog` écrit l'en-tête du module Verilog : nom du nœud, entrées et sorties,
- `local_var2verilog` s'occupe de générer la déclaration des variables locales.

Pour chaque variable, on regarde son type afin de produire la bonne déclaration Verilog :

- les entiers sont traduits en `wire [63:0]`,
- les booléens en `wire`,
- les types énumérés sont gérés séparément via des macros.

À ce stade, seuls ces trois types sont supportés dans le compilateur. Une évolution possible du compilateur serait la prise en charge des types real, ce qui permettrait de supporter une plus large partie des programmes Lustre, et des types structurés, qui éviteraient une expansion systématique en variables élémentaires.

`init_info_vars` détermine aussi l'horloge associée à chaque variable. Ce point est important, car Verilog impose une gestion explicite des événements d'horloge, ce qui influence la façon dont le code est généré pour chaque expression (ce sera détaillé dans la partie suivante).

La Deuxième étape consiste à identifier les variables qui nécessitent une version `pre` d'elles-mêmes. En Lustre, l'opérateur `pre` permet d'accéder à la valeur d'une variable au cycle précédent. Verilog ne dispose pas directement de cette notion. Pour la simuler, il faut stocker la valeur courante dans un registre à chaque front d'horloge.

Pour repérer ces variables, on utilise la fonction `pre_handler_verilog`. Elle analyse les équations du programme et ajoute les variables concernées dans un ensemble spécifique :

```
List.iter (fun eq -> let (_, val_e) = eq.it in pre_handler_verilog val_e when_set pre_set
    hashClkSet) body.eqs_eff;
```

Chaque variable détectée sera ensuite déclarée comme un registre en Verilog, mis à jour à chaque front d'horloge. Ce mécanisme permet de retrouver la valeur précédente de la variable, comme attendu par le `pre` en Lustre.

Il y a cependant un cas particulier : les variables qui ne sont pas sur l'horloge de base (`BaseClock`) nécessitent forcément un pre. Cela vient d'une différence fondamentale entre les deux langages. En Lustre, une horloge conditionnelle (`when`) permet de désactiver certains calculs si la condition n'est pas remplie. En Verilog, tout est calculé tout le temps : on ne peut pas "couper" un chemin de calcul.

Pour simuler ce comportement, on utilise un registre `pre` qui ne s'actualise que lorsque la condition d'horloge est vraie. Si ce n'est pas le cas, la valeur précédente est conservée. Ensuite, un multiplexeur est utilisé pour choisir entre la valeur nouvellement calculée et l'ancienne, en fonction de la condition d'horloge. Cela permet de simuler correctement le comportement du `when` de Lustre tout en restant dans le modèle d'exécution de Verilog. (Voir fin de rapport pour plus de précision)

Pour finir cette étape, on déclare les pre dans le fichier cible.

La Troisième étape consiste à s'occuper des équations, pour cela, on va pour chaque équation lancer les fonctions qui s'occupent de traduire les équations en chaîne de caractères. Les traductions effectuées seront plus détaillées dans la partie suivante.

Une fois toutes les équations traduites et les registres `pre` déclarés, il reste à compléter la mise à jour des registres. Chaque variable marquée comme nécessitant un pre doit être actualisée à chaque front montant de l'horloge associée. Cette logique est directement traduite en blocs `always @(posedge clk)` dans Verilog. Cette syntaxe, qui représente un processus de simulation réveillé à chaque front montant de l'horloge, est la syntaxe reconnue par les outils de synthèse de l'industrie pour décrire un registre matériel. Une fois toutes les mises à jour correctement définies, le module est clôturé par l'instruction `endmodule`.

Dans le cas où plusieurs nœuds Lustre doivent être traduits, le processus est simplement répété pour chacun d'eux. Or, Verilog permet, comme Lustre, de générer et d'instancier hiérarchiquement des unités à l'aide de modules. Chaque nœud devient alors un module Verilog indépendant.

Cependant, une attention particulière doit être portée aux nœuds qui ne s'exécutent pas sur l'horloge de base (`BaseClock`). Dans Lustre, lorsqu'un nœud est activé conditionnellement via un `when`, cela signifie qu'il ne doit s'exécuter que si une certaine condition est vraie. Or, comme mentionné précédemment, Verilog n'offre pas nativement ce type de contrôle : toutes les opérations sont évaluées à chaque cycle.

Pour pallier cela, nous introduisons un signal supplémentaire `enable` dans les modules concernés. Ce signal est utilisé pour simuler le comportement conditionnel de Lustre. Si `enable` est à true, alors le module calcule ses sorties normalement ; sinon, il renvoie la valeur précédente, conservée dans des registres.

Concrètement, cela est réalisé via des multiplexeurs dans le code Verilog, qui sélectionnent entre la valeur nouvellement calculée (si `enable` est vrai) et la valeur mémorisée (si `enable` est faux). Cette approche permet d'imiter le mot-clé `when` de manière fidèle, tout en respectant la sémantique de Verilog.

5 Détails sur la traduction

Dans cette partie, je vais revenir en détail sur le sous-ensemble de Lustre v6 qui a été traduit en Verilog, et comment il l'a été.

5.1 Typage et Variable

Comme dit précédemment, le premier enjeu de la traduction Lustre vers Verilog est le typage. Dans notre cas, il reste relativement simple, bien que non trivial. La fonction se chargeant de cette étape est la fonction `type_to_string_verilog`, son comportement est décrit comme tel :

- Une variable de type booléen (par exemple `b`), est traduite par un fil (`wire b;`). En effet, dans le cadre de la description de circuit, nous parlons d'objets réels, en l'occurence, un fil.
- Une variable de type `int` (par exemple `i`), sera traduit par un tableau de fil de taille 64, avec la mention `signed` devant (`wire signed [63:0] i;`), car physiquement, un `int` en Lustre est un entier sur 64 bits.
- Une variable de type énuméré est un cas un peu particulier. Comme dit plus haut, des macros sont ajoutées avant l'écriture du code afin d'imiter le comportement d'un type énuméré. En réalité, le type énuméré est traduit sous forme d'entier sur un tableau de taille $\lceil log_2(Tailledutype) \rceil$. Par exemple, pour une variable z de type énuméré de taille 3, on a `wire [1:0] z;`.

Les variables sont, quant à elles, facilement traduisibles de Lustre vers Verilog. En effet, les variables d'entrée d'un nœud deviennent les entrées du module, les variables de sortie deviennent ses sorties, et les variables locales deviennent des variables internes au module. Il faut uniquement indiquer `input` pour les entrées, `output` pour les sorties et rien pour les variables locales. Chaque variable est un `wire` et non un `reg` car il n'y a pas de mémoire dans les variables, il y en a uniquement dans leur version `pre`.

Exemple de traduction des types via le compilateur

Version Lustre :

```
type color = enum {A, B, C};

node test(a:int) returns(b:bool)
var c: color;
```

```
let
   c = A;
   b = c = A;
tel;
```

Version Verilog :

```
`define A 2'd0
`define B 2'd1
`define C 2'd2
module testV (
  input wire veri_rst,
  input wire veri_clk,
  input wire signed [63:0] _Va,
  output wire _Vb
);
  wire [1:0] _Vc;
  assign _Vb = (_Vc == `A);
  assign _Vc = `A;
endmodule
```

5.2 Traduction directe

Certaine traduction se font directement, en voici la liste :

- Les variable et constante se traduisent directement, à l'exception des types énumérés où il est nécessaire de mettre un ` avant la constante (syntaxe Verilog)
- Les opérations (+, − , ==, <=, `modulo` ...) se traduisent de manière simple et directe. En effet, il suffit de faire un simple *pattern matching* en OCaml, le langage dans lequel le compilateur Lustre est écrit, pour traduire ces opérations.
- Le `if` est un cas très direct aussi. En effet, le comportement d'un if en Lustre n'est pas un if "classique", il fonctionne de manière à définir la valeur d'un variable en fonction d'une expression booléenne (on ne fait pas de choix d'exécution avec, juste un choix de résultat). C'est exactement le même cas dans Verilog avec le point d'interrogation (opérateur d'affectation ternaire), il nous suffit donc de remplacer les `if` par des points d'interrogation.

C'est la fonction `op2verilog` qui se charge de la traduction :

```
let (op2verilog: string -> string) = function
 | "iplus" | "plus" | "+" -> "+"
 | "minus" | "iminus" | "iuminus" | "uminus" | "-" -> "-"
 | "times" | "itimes" | "*" -> "*"
 | "slash" | "islash" | "/" | "div" -> "/"
 | "mod" | "%" -> "%"
 | "and" | "&&" -> "&&"
 | "or" | "||" -> "||"
 | "xor" | "^" -> "^"
 | "not" -> "!"
 | "=" | "eq" -> "=="
 | "<>" | "neq" -> "!="
 | "<" | "lt" | "ilt" | "rlt" -> "<"
 | ">" | "gt" | "igt" | "rgt" -> ">"
 | "<=" | "lte" | "ilte" | "rlte"  -> "<="
 | ">=" | "gte" | "igte" | "rgte" -> ">="
 | "if" -> "if"
 | "=>" | "impl" -> "=>"
 | "==" -> "=="
 | "!=" -> "!="
 | s -> ((Printf.eprintf "op2verilog: unsupported op string: %s" s);
                assert false;)
```

Selon la valeur de sortie, la fonction `equation_2_verilog` se charge de traduire (par exemple le `if`, qui est un cas particulier).

Exemple de traduction simple :

Version Lustre

```
node test(a:int) returns(b:int)
var tmp : int;
let
   tmp = a + 2;
   b = if tmp mod 2 = 0 then a else tmp;
tel;
```

Version Verilog

```
module testV (
   input wire veri_rst,
   input wire veri_clk,
   input wire signed [63:0] _Va,
   output wire signed [63:0] _Vb
);
   wire _V_X_2;
   wire signed [63:0] _V_X_1;
   wire signed [63:0] _Vtmp;
   assign _Vb = _V_X_2 ? _Va : _Vtmp;
   assign _V_X_1 = (_Vtmp % 2);
   assign _V_X_2 = (_V_X_1 == 0);
   assign _Vtmp = (_Va + 2);
endmodule
```

Dans Lustre comme dans Verilog, l'ordre des affectations n'a pas d'importance : c'est une propriété commune aux deux langages.

5.3 Traduction indirecte

Cependant, toutes les traductions ne sont pas aussi simples que les précédentes. En effet, Lustre possède une couche de logique temporelle que Verilog n'a pas nativement.

Ainsi, la traduction de ces fonctionnalités sont faites comme tel :

- Traduction des `pre` :

  Comme dit précédemment, les `pre` du programme Lustre sont cherchés, dans un premier temps, dans tout le nœud cible. Ensuite, ils sont stockés dans un ensemble (`pre_set`) afin de se souvenir de quelles variables ont un `pre` ou non. Ainsi, dans cet ensemble, on peut savoir quelle variable a un `pre`, ce qui peut apparaître comme un problème à première vue quand on a une variable `a` telle que l'on veut obtenir `pre pre a`. Cependant, comme les expressions sont expansées par le compilateur Lustre grâce à une option du compilateur, ce n'est pas un problème (`pre pre a` devient `pre X1` avec `X1 = pre a`).

  La fonction s'occupant de cette tâche est `pre_handler_verilog`.

  Ensuite, les variables `pre` sont décrites dans le fichier destination (comme expliqué plus tôt).

  Chaque fois qu'un `pre` est utilisé dans le code Lustre, on le remplace par sa version Verilog (stockée dans un registre).

  Chaque `pre` est actualisé au front montant de la horloge (sauf s'il dépend d'une autre horloge avec un `when`, par exemple).

  Exemple de génération avec pre :

Version Lustre

```
node test(a:int) returns(b:int)
let
   b = pre pre a;
tel;
```

Version Verilog

```
module testV (
   input wire veri_rst,
   input wire veri_clk,
   input wire signed [63:0] _Va,
```

```
    output wire signed [63:0] _Vb
);
    wire signed [63:0] _V_X_1;
    reg signed [63:0]  pre_V_X_1;
    reg signed [63:0]  pre_Va;
    assign _Vb = pre_V_X_1;
    assign _V_X_1 = pre_Va;
    always @(posedge veri_clk) begin
        pre_V_X_1 <= _V_X_1;
        pre_Va <= _Va;
    end
endmodule
```

- Traduction de `when` et `current` :

  La traduction des constructions `when` et `current` est sans doute la plus complexe à gérer.

  Le vrai souci ici, c'est qu'un circuit matériel ne peut pas, par nature, interrompre son flot de calculs. Or, c'est exactement ce que cherche à faire le `when` dans Lustre : ne calculer une valeur que lorsque la condition (horloge booléenne) est vraie.

  Pour imiter ce comportement, l'idée est donc de ne pas utiliser directement la valeur calculée, mais de la remplacer par la valeur précédente quand la condition d'horloge n'est pas satisfaite. Autrement dit, on écrase le résultat avec l'ancienne valeur. Pour faire ça, on crée un registre `pre` pour chaque variable concernée.

  Ensuite, on utilise un multiplexeur dans le code Verilog pour choisir dynamiquement entre la nouvelle valeur calculée et la valeur stockée précédemment dans le `pre`, selon la valeur du booléen de l'horloge.

  On fait aussi en sorte que la valeur contenue dans le `pre` ne soit mise à jour que lorsque le signal d'horloge associé est à true. Concrètement, cela se traduit en Verilog par des blocs `always @(posedge veri_clk)` contenant une condition `if (clk)`. Pour gérer ce comportement dans le code, on utilise deux structures :

  - Le `pre_set` (déjà mentionné précédemment), qui contient les variables ayant un pre explicite dans le code Lustre.
  - Une table de hachage (`hashClkSet`) , qui associe chaque horloge à l'ensemble des variables qui lui sont rattachées. Cela permet de ne pas mettre dans le `pre_set` des variables qui sont déjà gérées par horloge.

  Le mot-clé `when` en tant que tel n'est donc jamais directement détecté. C'est plutôt l'horloge d'activation de chaque variable, telle qu'elle est définie dans le LIC code, qui nous permet d'en déduire le comportement équivalent. Pour le `current`, la traduction est beaucoup plus simple. Une variable étant toujours définie dans Verilog, on peut considérer que chaque ligne est équivalente à un `current`. Il n'y a pas besoin de traitement particulier, mis à part qu'on ne peut pas représenter le `nil` de Lustre, qui n'a pas d'équivalent en Verilog. Mais `nil` n'est pas une vrai valeur en Lustre, elle n'est pas censé arriver, c'est plus un "debug" pour l'utilisateur. Quand il s'agit d'écrire les équations, quand l'une d'entre elle dépend d'une horloge, on rajoute un multiplexeur. Celui ci décide entre prendre l'ancienne valeur et la nouvelle

  Exemple de traduction de When et Current :

Version Lustre

```
node test(a:int; clk:bool) returns(b:int)
var tmp : int when clk;
let
    tmp = a when clk;
    b = current(tmp);
tel;
```

Version verilog

```
module testV (
   input wire veri_rst,
   input wire veri_clk,
   input wire signed [63:0] _Va,
   input wire _Vclk,
   output wire signed [63:0] _Vb
);
   wire signed [63:0] _Vtmp;
   reg signed [63:0]  pre_Vtmp;
   assign _Vb = _Vtmp;
   assign _Vtmp = _Vclk ? (_Va) : pre_Vtmp;
   always @(posedge veri_clk) begin
       if (_Vclk) begin
           pre_Vtmp <= _Vtmp;
       end
   end
endmodule
```

Cas particulier des `when` sur type énuméré :

Quand l'horloge est un type énuméré, le `when` vérifie si l'horloge est égale à une valeur précise de l'énumération, et non si elle est simplement vraie.

En Verilog, cela se traduit par une condition du type `if (clk` = VAL)=. Le reste reste identique : usage d'un pre et choix via un multiplexeur.

Exemple de traduction :

Version Lustre

```
type color = enum{yellow, red, green};

node when_enum(c:color) returns (res : int when red(c));
let
  res = 0 when red(c) -> pre res + 1 when red(c);
tel;
```

Version Verilog

```
`define yellow 2'd0
`define red 2'd1
`define green 2'd2
module when_enumV (
  input wire veri_rst,
  input wire veri_clk,
  input wire [1:0] _Vc,
  output wire signed [63:0] _Vres
);
  wire signed [63:0] _V_X_4;
  wire signed [63:0] _V_X_3;
  wire signed [63:0] _V_X_2;
  wire signed [63:0] _V_X_1;
  reg signed [63:0]  pre_V_X_1;
  reg signed [63:0]  pre_V_X_2;
  reg signed [63:0]  pre_V_X_3;
  reg signed [63:0]  pre_V_X_4;
  reg signed [63:0]  pre_Vres;
  assign _Vres = (_Vc == `red ) ? (veri_rst ? (_V_X_1) : (_V_X_4)) : pre_Vres;
  assign _V_X_1 = (_Vc == `red ) ? (0) : pre_V_X_1;
  assign _V_X_2 = (_Vc == `red ) ? (pre_Vres) : pre_V_X_2;
  assign _V_X_3 = (_Vc == `red ) ? (1) : pre_V_X_3;
  assign _V_X_4 = (_Vc == `red ) ? ((_V_X_2 + _V_X_3)) : pre_V_X_4;
  always @(posedge veri_clk) begin
   if ((_Vc == `red )) begin
    pre_V_X_1 <= _V_X_1;
    pre_V_X_2 <= _V_X_2;
    pre_V_X_3 <= _V_X_3;
    pre_V_X_4 <= _V_X_4;
    pre_Vres <= _Vres;
   end
  end
endmodule
```

- Traduction du Merge :

  Il existe deux cas de `merge` en Lustre : les `merge` sur booléen et ceux sur type énuméré.

Dans les deux situations, la traduction vers Verilog est globalement similaire, car un `merge` peut être remplacé directement par une structure conditionnelle, de type `if`.

Dans le cas d'un `merge` booléen, c'est assez intuitif : si le booléen est à `true`, on prend la branche associée à `true`, sinon celle pour `false`.

Pour les merge sur type énuméré, c'est un peu plus verbeux, car il faut empiler plusieurs `if` imbriqués (un pour chaque valeur possible de l'énumération). Mais ça reste une traduction simple à générer automatiquement.

C'est d'ailleurs la même logique de transformation que celle utilisée lors de l'expansion classique du code par le compilateur Lustre, donc sans surprise particulière ici.

De plus, même si la génération de code peut produire des structures complexes, comme une succession d'=if= imbriqués aboutissant à de nombreux multiplexeurs, ce n'est pas un réel problème. Les outils de synthèse Verilog appliquent en effet des optimisations poussées et transforment ce code en une implémentation matérielle efficace, en simplifiant et en réorganisant intelligemment les multiplexeurs générés.

Exemple de traduction de `merge` :

Version Lustre

```
type color = enum {yellow, blue, red};

node test(c:color; b:bool) returns(a,d:int);
let
   a = merge b (false -> 0) (true -> 1);
   d = merge c
            (yellow -> 1 when yellow(c))
            (blue -> 2 when blue(c))
            (red -> 3 when red(c));
tel;
```

Version Verilog

```
`define yellow 2'd0
`define blue 2'd1
`define red 2'd2
module testV (
  input wire veri_rst,
  input wire veri_clk,
  input wire [1:0] _Vc,
  input wire _Vb,
  output wire signed [63:0] _Va,
  output wire signed [63:0] _Vd
);
  wire signed [63:0] _V_X_3;
  wire signed [63:0] _V_X_2;
  wire signed [63:0] _V_X_1;
  reg signed [63:0]  pre_V_X_2;
  reg signed [63:0]  pre_V_X_1;
  reg signed [63:0]  pre_V_X_3;
  assign _Vd = (_Vc == 2'd2) ? (_V_X_3) : ((_Vc == 2'd1) ? (_V_X_2) : (_V_X_1));
  assign _V_X_1 = (_Vc == `yellow ) ? (1) : pre_V_X_1;
  assign _V_X_2 = (_Vc == `blue ) ? (2) : pre_V_X_2;
  assign _V_X_3 = (_Vc == `red ) ? (3) : pre_V_X_3;
  assign _Va = (_Vb == 1'd1) ? (1) : (0);
  always @(posedge veri_clk) begin
   if ((_Vc == `blue )) begin
    pre_V_X_2 <= _V_X_2;
   end
  end
  always @(posedge veri_clk) begin
   if ((_Vc == `yellow )) begin
    pre_V_X_1 <= _V_X_1;
   end
  end
  always @(posedge veri_clk) begin
   if ((_Vc == `red )) begin
    pre_V_X_3 <= _V_X_3;
   end
  end
endmodule
```

6 Traduction avec nœuds non expansés

Jusqu'à présent, nous avons utilisé la stratégie de l'expansion complète. Cependant, le compilateur Lustre propose plusieurs options pour choisir d'expanser ou non un nœud. Lorsqu'un nœud est expansé, on fusionne tout dans un seul nœud principal (le top) qui contient l'intégralité du programme. Cette approche peut entraîner une duplication importante de code. En revanche, dans la version non expansée, les nœuds sont conservés tels quels et appelés via des appels de nœuds, ce qui évite la duplication de parties de code.

Cette approche non expansée permet donc de réduire la taille du code généré en limitant la duplication, et génère un code Verilog plus modulaire et plus lisible, facilitant la maintenance et le débogage. Par ailleurs, elle pourrait laisser aux outils de synthèse la possibilité d'optimiser efficacement les appels de modules

Quand les nœuds ne sont pas expansés, la traduction reste assez directe. On applique simplement la fonction de génération de module à chaque nœud individuellement. Pour éviter les doublons, un ensemble `deja_vu` garde la trace des nœuds déjà traités.

Pendant le parcours des équations, les appels à des nœuds non encore traduits sont ajoutés à une liste `a_voir`. Une fois le nœud principal généré, on boucle sur cette liste pour traduire les autres :

```
List.iter
  (fun node_elm_and_clk ->
    Printf.fprintf oc "\n";
    let node_elm, clk_m = node_elm_and_clk in
    generate_module node_elm oc deja_vu (Some et) licprg clk_m)
  !a_voir
```

Il y a cependant un point important à gérer à cause du `when` : un module Verilog ne peut pas arrêter ses calculs comme un nœud Lustre le ferait. Pour imiter ce comportement, on ajoute un signal `enable` au module. Il s'intègre aux multiplexeurs utilisés pour les horloges conditionnelles, permettant de choisir entre la valeur calculée et la valeur précédente.

Tous les nœuds compilés avec un `enable` doivent obligatoirement avoir un registre `pre` pour chaque variable locale ou de sortie. Sinon, on ne pourrait pas restaurer la valeur précédente.

Pour limiter les conflits de noms, les modules compilés avec un `enable` prennent comme nom `<NomLustre>E`, et ceux sans `enable` deviennent `<NomLustre>V`. Le module principal (`top`) se termine toujours par `V`, ce qui permet aussi d'éviter les noms réservés de Verilog.

Un même nœud peut donc être généré sous deux versions distinctes (avec ou sans `enable`), selon l'usage qui en est fait dans le code.

Version Lustre

```
node incr(a:int) returns (res : int);
let
   res = a + 1;
tel;

node test(a:int; ck:bool) returns(res : int when ck);
let
   res = incr(a when ck);
tel;
```

Version Verilog

```
module testV (
  input wire veri_rst,
  input wire veri_clk,
  input wire signed [63:0] _Va,
  input wire _Vck,
  output wire signed [63:0] _Vres
);
  wire signed [63:0] _V_X_1;
  reg signed [63:0]  pre_V_X_1;
  reg signed [63:0]  pre_Vres;
  incrE _incrE0 (
   ._Va(_V_X_1),
   ._Vres(_Vres),
   .veri_clk(veri_clk),
   .enable(_Vck),
   .veri_rst(veri_rst)
  );
```

```
    assign _V_X_1 = _Vck ? (_Va) : pre_V_X_1;
    always @(posedge veri_clk) begin
     if (_Vck) begin
      pre_V_X_1 <= _V_X_1;
      pre_Vres <= _Vres;
     end
    end
  endmodule

  module incrE (
    input wire veri_rst,
    input wire enable,
    input wire veri_clk,
    input wire signed [63:0] _Va,
    output wire signed [63:0] _Vres
  );
    reg signed [63:0]  pre_Vres;
    assign _Vres =  enable ? ((_Va + 1)) : pre_Vres;
    always @(posedge veri_clk) begin
     if (enable) begin
      pre_Vres <= _Vres;
     end
    end
  endmodule
```

Le signal `enable` d'un nœud est naturellement défini par l'horloge utilisée lors de son appel dans le nœud parent. Cependant, si ce nœud parent est lui-même compilé avec un signal `enable`, alors le signal effectif devient une conjonction logique (et) entre l'horloge de l'appel et le signal `enable` du parent.

Autrement dit, un nœud compilé avec `enable` transmet ce mode de fonctionnement à tous les nœuds qu'il appelle : ils seront eux aussi générés avec un enable.

7 Détails supplémentaires sur les noms de variables

Afin d'éviter tout conflit avec les noms réservés en Verilog ou ceux générés par Verilator, tous les noms de variables provenant de Lustre sont modifiés. On ajoute systématiquement le préfixe `_V` devant chaque nom de variable.

Cela permet non seulement d'éviter les collisions avec les signaux ajoutés automatiquement par le compilateur (comme l'horloge Verilog, le `reset` ou encore le signal `enable`), mais aussi de garantir une distinction claire entre les éléments générés et ceux issus directement du code Lustre.

8 Container GUIX

Pour garantir un environnement stable et reproductible pour les tests, un container GUIX a été mis en place.

Pour l'utiliser, il faut avoir GUIX installé sur la machine. Ensuite, il suffit de lancer la commande suivante :

```
guix time-machine -C channels.scm -- shell -CP -m guixVerilo/manifest.scm -- bash -c ". ./
    guixVerilo/build.sh; exec bash;"
```

Cette commande suppose que les fichiers `manifest.scm` et `build.sh` sont présents, ce qui est le cas sur le GitLab du projet Lustre, branche Verilog.

Une fois dans le container, il suffit de se rendre dans le dossier test du dépôt et d'exécuter les tests avec :

Pour la version sans expansion des modules :

```
make -j N test_verilog 2> res.txt
```

Pour la version avec expansion des modules :

```
make -j N test_verilog_module 2> res.txt
```

Remplacer N par le nombre de cœurs à utiliser.

Les programmes qui ne passent pas les tests sont listés dans le fichier `res.txt`.

Avec le fichier `liste_verilog.test` fourni dans le dépôt Git, on devrait en avoir environ 6 en échec pour la version sans expansion, et 7 pour la version avec expansion de modules.

Ces échecs ne viennent a priori pas du compilateur Verilog lui-même, mais d'autres sources : problèmes dans la génération de l'oracle ou de l'environnement Lurette, erreurs de correspondance entre les versions expansées et non expansées, etc.

References

[1] E. Jahier, L. Mandel, N. Halbwachs, and P. Raymond, "Verimag manual,"

[2] S. Palnitkar, *Verilog HDL: a guide to digital design and synthesis*. Prentice Hall Professional, 2003, vol. 1.

[3] F. Rocheteau and N. Halbwachs, "Implementing reactive programs on circuits a hardware implementation of lustre," in *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, Springer, 1991, pp. 195–208.

[4] I. Winandy, A. Dion, P.-L. Garoche, and F. Manni, "A reactive system-specific compilation chain from synchronous dataflow models to fpga netlist," in *2024 IEEE 10th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, IEEE, 2024, pp. 11–21.

[5] E. Jahier, "The lurette v2 user guide," *Lurette V2*, vol. 4, no. 4, 2004.

[6] P. Raymond, Y. Roux, and E. Jahier, "Lutin: A language for specifying and executing reactive scenarios," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, p. 753 821, 2008.