INITIATION AU LANGAGE VERILOG

Jean Louis Noullet - 1995-2005 - 5ème édition rev.5

1. Introduction- -2
   1.1 Objectifs 2
   1.2 Les supports de données en Verilog 3

2. Des exemples, pour commencer - - - - - - - - - - - - -3
   2.1 Simulation d’un “latch RS” 3
   2.2 Un exemple de conception descendante ("TOP-DOWN") 6
   2.3 Simulation d’un circuit synchrone 8

3. Déclarations - 10

4. Expressions et Constantes - - - - - - - - - - - - - - - - 11
   4.1 opérations logiques “verticales” ou “bitwise” 11
   4.2 opérations logiques “horizontales” ou “unary” 11
   4.3 opérations arithmétiques et décalage 11
   4.4 opération booléennes 12
   4.5 opérations relationnelles 12
   4.6 vecteurs 12
   4.7 constantes numériques 12
   4.8 extension - troncature 13

5. Communication reg <--> net - - - - - - - - - - - - - - - 13

6. Etats et Forces - 14

7. Blocs Procéduraux - 15
   7.1 Règles 15
   7.2 Temporisation procédurale 16

8. Constructions Procédurales - - - - - - - - - - - - - - - 17
   8.1 Itérations ou boucles (for, while, repeat, forever) 17
   8.2 Décisions (if, expr. conditionnelle, case) 18

9. Encore quelques exemples- - - - - - - - - - - - - - - - 19
   9.1 bascule D avec clear asynchrone 19
   9.2 Buffer tri-state 20
   9.3 Cellule paramétrée 20

10. Introduction à la Synthèse Logique - - - - - - - - - - 21
   10.1 Modules combinatoires 21
   10.2 Modules séquentiels synchrones 23
   10.3 Machine à états finis 24
   10.4 Registre avec clear asynchrone 25

11. Ce qui resterait à voir - - - - - - - - - - - - - - - - - - 26

© J.L.N.2005

INITIATION AU LANGAGE VERILOG

1. Un langage de description de matériel

1.1 Objectifs

Les langages de description de matériel (**HDL** = **H**ardware **D**escription **L**angage) font partie des outils de base pour la conception de systèmes logiques intégrés câblés, que le produit final soit construit sur des composants électriquement configurables (**FPGA** = **F**ield **P**rogrammable **L**ogic **A**rray) ou des circuits intégrés spécifiques (**ASIC** = **A**pplication **S**pecific **I**ntegrated **C**ircuit).

Ces langages doivent être utilisables pour :
- écrire la *spécification* d'un système (description *comportementale* ou *fonctionnelle*)
- le ***construire*** par interconnexion de cellules élémentaires (description *structurelle* ou *physique*)
- gérer des description ***hiérarchisées*** (dont la description structurelle est un cas particulier)
- ***valider*** par ***simulation*** les différents types de description
- permettre l'utilisation de programmes de ***synthèse automatique***

De plus les descriptions comportementales doivent se faire dans deux modes :
- mode ***concurrent*** : les données se propagent dans des élements dont le comportement est décrit par des déclarations qui ont un effet permanent (comme dans les descriptions structurelles)
- mode ***procédural*** : les données sont manipulées par des séquences d'instructions (comme dans les programmes d'ordinateurs)

Dans le mode concurrent les données sont actives, on parle de ***"data flow"***, alors que dans le mode procédural c'est le programme qui contrôle les évenements, on parle alors de ***"program flow"***.

Il existe deux standards de langages de description de matériel :
- **VHDL**
- **Verilog**

Leurs syntaxes sont assez différentes mais mettent en oeuvre les mêmes concepts.

© J.L.N.2005

1.2 Les supports de données en Verilog

Le langage Verilog utilise deux classes de supports de données distinctes pour les modes concurrents et procedural.

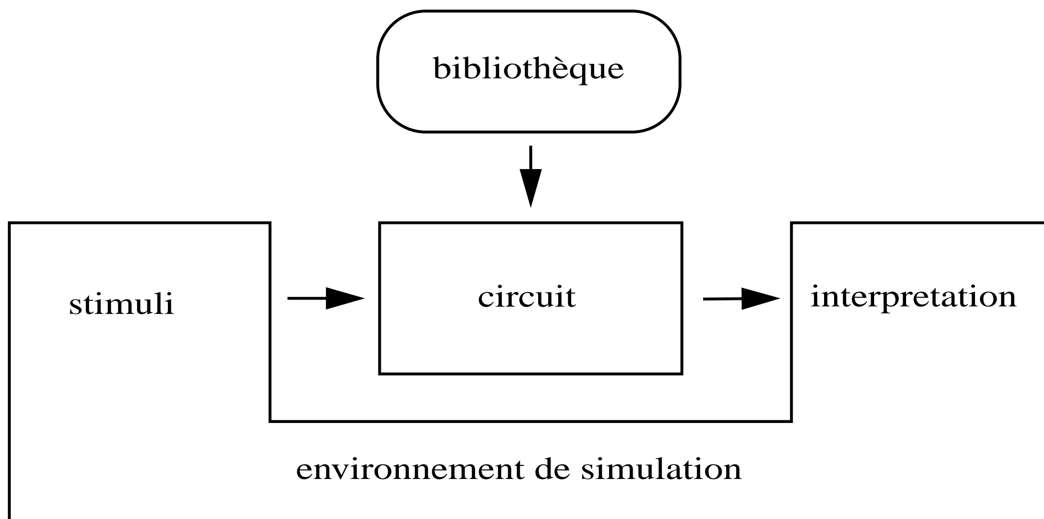
- classe ***"net"*** dans le mode concurrent : un net est équivalent à un noeud d'un circuit électrique, son état est contrôlé en permanence par les éléments aux sorties desquels il est connecté.

- classe ***"reg"*** dans le mode procédural : un reg est équivalent à une variable d'un programme informatique : il subit des affectations instantanées par instructions et conserve son état jusqu'à la prochaine affectation.

Le bon usage des nets et regs est la seule réelle difficulté de Verilog. On y a consacré le chapitre **5.**

2. Des exemples, pour commencer

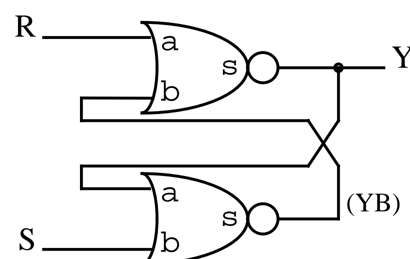
2.1 Simulation d'un "latch RS"



Dans cette exemple, on utilisera le langage Verilog pour décrire un petit circuit, la bibliothèque de cellules standard qui sert à le construire, et l'environnement de simulation qui sert à le valider.

2.1.1 Description structurelle de la bascule rs_latch :

```
module rs_latch( R, S, Y );
output Y;
wire YB;
input R, S;
  LibNo2 I1 ( Y, R, YB );
  LibNo2 I2 ( YB, Y, S );
endmodule
```



© J.L.N.2005

Un ***module*** ou cellule est un noeud de la hiérarchie structurelle.
Sa description commence par la déclaration des ***ports*** qui servent à le connecter au monde extérieur (les ports sont de la classe *"net"*).
Suivent des ***instances*** ou "appels" de modules situés en dessous dans la hiérarchie. A chaque instance est associé un nom d'instance unique (ici I1 et I2).
Ici, la cellule LibNo2 est supposée faire partie d'une bibliothèque de cellules standard.
Dans ce module, un net interne YB est déclaré en tant que ***wire*** (classe *"net"*).
Les identifieurs créés dans le module (R, S, Y, YB, I1, I2) sont locaux.
La connexion des nets de rs_latch aux ports de chaque instance de LibNo2 est régie par l'**ordre** des éléments dans la liste. Le langage permet une variante : la connexion par noms, plus explicite mais plus lourde :

```
module rs_latch( R, S, Y );
output Y; input R, S;
wire YB;
  LibNo2 I1 ( .a(R),  .b(YB), .s(Y) );
  LibNo2 I2 ( .s(YB), .a(Y),  .b(S) );
endmodule
```

2.1.2 Description comportementale des cellules standard

Pour simuler notre rs_latch, nous avons besoin d'une description comportementale de LibNo2 :

```
`timescale 1ns/10ps
module LibNo2 ( s, a, b );   // porte NOR
output s; input a, b;
  assign #(2.5, 2.0) s = ~(a|b);
endmodule
```

La directive de compilation ***`timescale*** sert à indiquer qu'on veut donner les temps en ns avec une résolution de 10 ps.
L'instruction ***assign*** établit une connexion permanente ("data-flow") entre l'expression `~(a|b)` et le net `s`, avec un délai de 2.5 ns à la montée et 2 ns à la descente.
(La syntaxe des expressions est détaillée dans le chapitre **4.**)

2.1.3 Environnement de simulation

L'environnement de simulation de notre circuit devra assurer la génération des stimuli et l'affichage des résultats, ce qu'on fera de manière procédurale.
Ces éléments doivent être hébergés dans un module dit module de test qui constituera la racine de la hiérarchie. Ce module n'a ni entrée ni sortie.
Pour que les signaux injectés dans le circuit (stimuli) puissent être contrôlés de manière procédurale, le module de test doit contenir des regs connectés aux entrées du circuit.

© J.L.N.2005

Par contre les sorties du circuit doivent être connectées à des nets (déclarés comme “wire”).

```
module test;
reg r, s; wire y;
rs_latch lat( r, s, y );  // instance de notre circuit
initial
  begin
  $display("simulation RS");
  $monitor("t = %4g ns : rs = %b%b --> %b (%b)",
            $realtime, r, s, y, lat.YB );
  r = 0; s = 0;
  #5 r = 1;
  #5 r = 0;
  #5 s = 1;
  #5 s = 0;
  #5 s = 1; r = 1;
  #5 $display("t=%g ns : travail fini", $realtime );
  end
endmodule
```

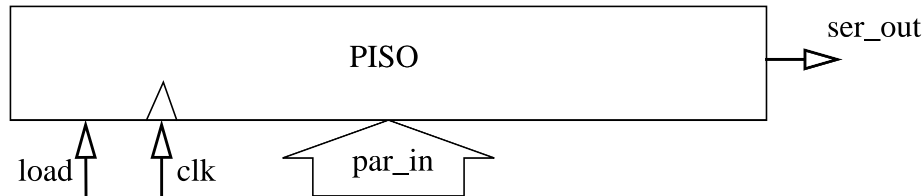
Après ces déclarations de signaux, le module test contient une instance du circuit rs_latch lui-même, et enfin un ***bloc procédural*** `(initial begin ......... end)`.
Dans ce bloc procédural, il y a une séquence d'instructions d'affectation dont certaines sont précédées d'une ***condition de temporisation*** `#5` exprimant une attente de 5 ns.
Les identifieurs commençant par $ désignent des fonctions ou tâches prédéfinies dans le simulateur Verilog-XL.
`$display` affiche du texte à l'instant où il est exécuté, alors que `$monitor` déclenche un processus automatique qui va réitérer l'affichage chaque fois qu'un des signaux concernés change d'état.
L'expression `lat.YB` permet de désigner un signal situé "en dessous" dans la hiérarchie en indiquant un chemin constitué de noms d’instances.

Dans cet exemple très simple, on a rencontré les 3 niveaux de description :
- structurelle (rs_latch)
- comportementale "data-flow" (LibNo2)
- procédurale (test)
Le langage permet de mélanger librement les 3 niveaux.
N.B. :
- Toute description doit être hébergée par un module
- Toute description procédurale doit être hébergée par un bloc procédural ***"initial"*** ou ***"always"*** (chapitre **7.**).

© J.L.N.2005

2.2 Un exemple de conception descendante ("TOP-DOWN")

Un registre à décalage à chargement parallèle de 4 bits ("PISO", pour "Parallel-In, Serial-Out").



Le signal load commande le chargement d'un mot par_in (ici de 4 bits) que le registre devra restituer bit par bit sur la sortie ser_out, en commençant par le bit de faible poids, avec un fonctionnement strictement synchrone (horloge clk).

Considérons une conception manuelle, partant d'une spécification consistant en une description comportementale globale (§2.2.1), et aboutissant à une structure basée sur des cellules standard disponibles (§2.2.3), avec une étape intermédiaire de découpage en "étages" (§2.2.2).

2.2.1 Modèle global du registre en Verilog :

```
// description comportementale du registre piso
module piso4( clk, load, par_in, ser_out );
input clk, load; input [3:0] par_in;
output ser_out;
reg [3:0] contenu;
always @ (posedge clk)
  begin
  if   ( load )
       contenu = par_in;         // chargement
  else
       contenu = contenu >> 1;   // decalage
  end
assign #3 ser_out = contenu[0];
endmodule
```

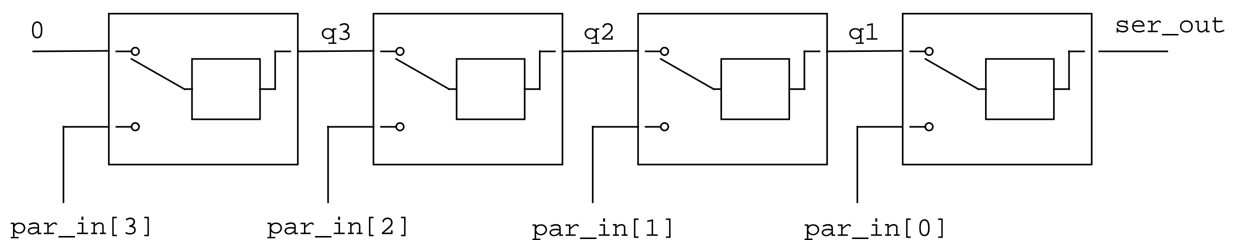
L'entrée parallèle `par_in` de ce registre est un groupe de 4 connexions représentées par un seul nom. Cette manière de désigner les signaux comme des tableaux indicés est souvent appelée ***"bus"***, ou mieux : ***"vecteur"***.
La déclaration comprend les valeurs extrêmes de l'indice entre crochets. Le premier indice correspondant au bit de fort poids, il est vivement conseillé de toujours mettre le plus fort indice en première position.
Quelques exemples d'utilisation :
`contenu[0]` est le *"lsb"* (less significant bit = bit de plus faible poids) de `contenu`,
`contenu[2]` est son bit de poids 4,

© J.L.N.2005

`par_in[2:0]` est un bus de 3 bits contenant les 3 *lsb* de `par_in`,
`par_in[3:0]` est synonyme de `par_in`.

Le coeur du modèle est un bloc procédural ***always***, dont la condition de déclenchement est le front montant de l'horloge `clk`. (cf chapitre **7.**)
La mémoire interne du registre est modélisée par le reg `contenu`. La sortie série `ser_out` est connectée de façon permanente au bit de faible poids de `contenu` par une directive assign.

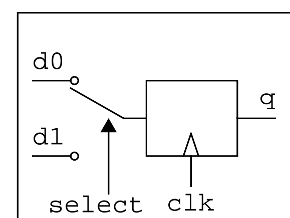
2.2.2 Modèle découpé en étages :




```
// description structurelle du registre piso
module piso4( clk, ld, par_in, ser_out );
input clk, load; input [3:0] par_in;
output ser_out; wire q1, q2, q3;
  etage e3 ( clk, load, par_in[3], 0,  q3 );
  etage e2 ( clk, load, par_in[2], q3, q2 );
  etage e1 ( clk, load, par_in[1], q2, q1 );
  etage e0 ( clk, load, par_in[0], q1, ser_out );
endmodule
```

N.B. `q3, q2, q1` sont des nets internes de piso4. On a relié à `0` (zéro) l'entrée de décalage de l'étage `e3` pour obtenir un comportement semblable à celui de l'opérateur `>>` de la description comportementale.

```
// decription comportementale d'un etage de registre
module etage ( clk, select, d1, d0, q);
input clk, select, d1, d0; output q;
reg rq;
always @ (posedge clk)
  if ( select ) rq = d1;
  else          rq = d0;
assign #1 q = rq;
endmodule
```



© J.L.N.2005

2.2.3 Modèle niveau cellules standard

```
// decription structurelle d'un etage de registre
module etage ( ck, select, d1, d0, q);
input ck, select, d1, d0; output q; wire d;
  LibDFF bascule ( .q(q), .ck(ck), .d(d) );
  LibMux multipl ( .y(d), .i0(d0), .i1(d1), .s(select) );
endmodule
```

Ceci suppose l'existence des cellules standard LibDFF et LibMux, dont les descriptions comportementales pourraient être :

```
// bascule D sensible au front montant
module LibDFF( q, ck, d );
input ck, d; output q;
reg resu;
  always @ ( posedge ck ) resu = d;
  assign #1 q = resu;
endmodule

// multiplexeur à deux entrées
module LibMux( i0, i1, s, y );
input i0, i1, s; output y;
  assign #1 y = ( i0 & ~s ) || ( i1 & s );
endmodule
```

2.3 Simulation d'un circuit synchrone

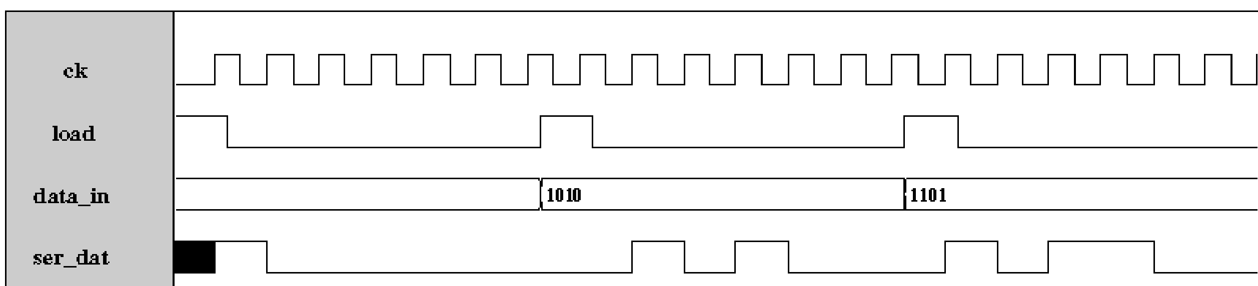
On va profiter du parallélisme offert par le langage pour produire le signal d'horloge périodique avec une boucle séparée de la séquence de test principale.
D'autre part on doit impérativement éviter toute collision entre les transitions des entrées et les fronts actifs d'horloge.
Il est donc commode et recommandable de "caler" toutes les transitions des entrées sur les temps multiples de la période d'horloge, et de décaler le générateur d'horloge d'une fraction de période.
Pour rendre plus lisible les signaux internes et éventuellement les sorties (machine de Mealy), on placera le front actif d'horloge *un peu avant* les multiples de la période.

Dans cet exemple, l'horloge de période `p` est calée de manière à ce que son premier front actif arrive à l'instant `p/4 +p/2` soit 3/4 de période. Les fronts actifs seront donc en avance de 1/4 de période par rapport aux transitions des entrées.
On a utilisé l'instruction `parameter` pour créer un ***symbole*** `p` représentant la période d'horloge.

© J.L.N.2005

Exemple : simulation du registre à décalage à chargement parallèle :

```
module test;
reg [3:0] data_in;      // reg vectoriel de 4 bits
reg ck, load; wire ser_data;
parameter p=5000;       // periode horloge

piso4 piso ( ck, load, data_in, ser_data ); // instance

initial   //============= sequence principale
  begin
         load = 1; data_in = 1;
  #p     load = 0;
  #(p*6) load = 1; data_in = 10;
  #p     load = 0;
  #(p*6) load = 1; data_in = 'b1101;
  #p     load = 0;
  #(p*6) $finish;
  end

initial   // ============= generation horloge
  begin
  ck = 0;
  #(p/4) forever #(p/2) ck = ~ck;
  end

always @ (posedge ck) // ===== affichage echantillonne
  $display("ld=", load, " data_in=%b", data_in,
           " ser_data=", ser_data );

endmodule
```



© J.L.N.2005

Sortie texte : on a mis dans cet exemple un affichage de résultats échantillonné, produisant une ligne de texte par période d'horloge : un bloc `always` qui appelle la fonction `$display` à chaque front actif d'horloge. Dans le cas d'un système synchrone, ceci facilite l'interprétation des résultats (contrairement à la fonction `$monitor` qui produit un nombre variable de lignes par période).
De cette manière, on "voit" les sorties du circuit exactement comme un autre module synchrone les "verrait".

3. Déclarations

ATTENTION : le langage ***distingue*** les majuscules et minuscules.

Les regs et nets doivent être déclarés :

- à **l'intérieur** des modules, mais
- à **l'extérieur** des blocs procéduraux

La déclaration des regs est toujours obligatoire.

La déclaration des nets est obligatoire sauf dans un cas : dans une représentation structurelle, les nets internes du module qui apparaissent dans les instances sans avoir été déclarés explicitement sont considérés comme déclarés implicitement (avec le type de net "par défaut").
Attention Danger : il faut déclarer explicitement les nets vectoriels, car la déclaration implicite leur donne une capacité de 1 bit seulement, sans avertissement.
Les *ports* du module sont des nets, et les déclarations de direction des ports (*input, output, inout*) valent comme déclarations de nets.

Les types de regs sont les suivants :

- reg "simple" ou "scalaire" de 1 bit
- reg vecteur (nombre de bits arbitraire) (non-signé)
- table de regs vectoriels = "RAM"
- integer (reg de 32 bits, signé)

Il y a de nombreux types de nets, qui diffèrent principalement par la manière dont sont résolus les conflits et les indéterminations. Le type de net le plus approprié pour les circuits CMOS statiques est le ***wire***, qui résout les conflits en fonction des forces des sources en présence (cf §6) et prend l'état **Z** (haute impédance) lorsqu'aucune force ne détermine son état.

N.B. Le mot *net* n'est pas un nom de type, mais de "classe de types".

© J.L.N.2005

exemple de déclaration :

```
reg [15:0] a, b; reg c; reg [1:200] bizar;
reg [7:0] babar[0:127];
wire [23:0] leBus;
```

`a` et `b` sont des regs vectoriels de 16 bits, dont le *lsb* porte l'indice 0,

`c` est un reg scalaire, `bizar` est un reg de 200 bits dont le *lsb* porte l'indice 200,

`babar` est un tableau de 128 mots de 8 bits, pouvant être utilisé comme modèle d'une mémoire RAM ou ROM, `leBus` est un net de 24 bits.

4. Expressions et Constantes

Les règles de constructions des expressions sont similaires dans les mondes concurrent et procédural. Elles sont fortement compatibles avec le langage C.

N.B. En cas de doute sur la priorité des opérateurs, utiliser des parenthèses.

4.1 opérations logiques “verticales” ou “bitwise”

comme en C :

opérateurs `~, &, |, ^` pour respectivement not, and, or, xor

exemple :

```
c = a & b;
d = ~c | ~( a ^ b );
```

chaque bit de c sera le "et" du bit de même poids pris sur a et sur b.

L'opérateur ~ est prioritaire sur les autres.

4.2 opérations logiques “horizontales” ou “unary”

opérateurs `&, |, ^` pour respectivement and, or, xor

exemple :

```
reg [7:0] b; reg azero;
azero = ~( | b );
```

azero est à 1 si et seulement si tous les 8 bits de b sont nuls.

4.3 opérations arithmétiques et décalage

comme en C :

opérateurs `+, -, *, /, %, <<, >>`

Attention : les opérateurs d'incrémentation et de décrémentation (++, --) n'existent pas en Verilog.

© J.L.N.2005

4.4 opération booléennes

comme en C :
opérateurs `!`, `&&`, `||` pour respectivement not, and, or
étant entendu que la valeur zéro représente *"false"* et que toute valeur différente de zéro représente *"true"*.

4.5 opérations relationnelles

comme en C :
opérateurs `==`, `!=`, `<`, `>`, `<=`, `>=` pour respectivement égal, différent, inférieur, supérieur, inf. ou égal, sup. ou égal.
avec en plus :
opérateurs `===`, `!==` pour respectivement littéralement égal, littéralement différent.
(au sens de `==` et `!=`, l'état X implique un résultat de comparaison toujours égal à X, alors que au sens de `===` et `!==`, X est égal à lui-même)
N.B. les comparaisons sont non signées, sauf pour le type integer.

4.6 vecteurs

Les opérateurs `{` et `}` servent à former un vecteur par concaténation de bits ou de vecteurs (**msb**=*most significant bit* en premier).
exemple :

```
reg [8:0] resu; reg [2:0] rev_a; reg q0, q1, q2;
wire [7:0] sum; wire [2:0] a;
resu = { carry, sum[7:0] };
{ q2, q1, q0 } = a + b;
rev_a = { a[0], a[1], a[2] };
c = sum[ n - 1 ];
```

`resu` est un vecteur de 9 bits dont le msb est `carry`,
`q2, q1, q0` reçoivent séparément les 3 lsb de la somme `a` + `b`,
`rev_a` contient les trois lsb de `a` dans l'ordre inverse,
`c` est le `n`-ième bit de `sum` (`n` étant un entier >= 1).

4.7 constantes numériques

Les constantes numériques sont décimales par défaut.
Les opérateurs `'b`, `'h` ,`'o` préfixent les constantes binaires, hexadécimales et octales.
(ces opérateurs peuvent être précédés d'un entier indiquant le nombre de digits.)

Le caractère "_" ("souligné") peut être inséré librement pour rendre la constante plus lisible.
La déclaration ***parameter*** permet d'associer un symbole à une constante.

© J.L.N.2005

exemples :

```
parameter msb8 = 'b1000_0000;
a = a + 'hFC0;
b = 'bz;  // etat haute impédance
```

4.8 extension - troncature

- si un élément d'une expression comporte plus de bits que la destination de cette expression, cet élément va être tronqué "par la gauche", c'est à dire que ce sont les bits de faible poids qui vont subsister.
- si un élément d'une expression comporte moins de bits que la destination de cette expression, cet élément va être complété par des zéros "à gauche" (du coté des forts poids).

Attention : ces opérations ne s'accompagnent d'aucun avertissement et peuvent dissimuler des erreurs fatales.

5. Communication reg <--> net

Règles fondamentales :

- l'état d'un reg n'est modifié que par assignement procédural (une instruction exécutée de manière séquentielle). L'assignement procédural est ***instantané***, le reg conserve son état jusqu'au prochain assignement.
- l'état d'un net n'est modifié que par connexion structurelle ("branchement" d'un instance) ou par assignement continu ("branchement" d'un net à une expression). Dans ces deux cas, l'action sur le net est ***permanente***.

Conséquences :

- un reg ne peut pas être branché à un port de sortie d'une instance, ni à un port bidirectionnel (*inout*),
- un net ne peut pas modifié directement dans une séquence d'instructions constituant un modèle procédural.

Transmission net --> reg :

ne présente aucune difficulté dans un bloc procédural, car le membre de droite d'un assignement procédural peut être une expression contenant librement regs et nets.

Transmission reg --> net :

- si le net est en dessous dans la hiérarchie structurelle, la transmission peut se faire par connexion du reg à un port d'entrée d'une instance.

C'est ainsi qu'on procède en général pour injecter les stimuli dans le module à tester : voir pour exemples §2.1.3 et §2.3.

© J.L.N.2005

- si le net est au dessus ou au même niveau dans la hiérarchie structurelle, la transmission peut se faire par assignement continu (le membre de droite d'un assignement continu peut être une expression contenant librement regs et nets).
C'est ainsi qu'on procède en général pour acheminer le résultat du modèle procédural d'un module vers les ports de sortie de ce module : voir pour exemples §2.2.1, §2.2.2 et §2.2.3.
(On en profite souvent pour introduire dans cet assignement le temps de propagation des sorties du module).

N.B. Ne pas confondre assignement procédural et assignement continu.
L'assignement procédural est dans un bloc procédural, il ne contient pas le mot-clef "assign" mais seulement le signe "=", son membre de gauche est un reg.
L'assignement continu est en dehors de tout bloc procédural, il commence par le mot-clef "**assign**", son membre de gauche est un net.

6. Etats et Forces

Le simulateur Verilog connaît quatre états logiques : 0, 1, X et Z.

L'état X est l'état "indéterminé" ou "inconnu".
C'est une pure abstraction : il n'existe pas dans le circuit réel, mais sert à traduire une absence de déterminisme à un instant particulier de la séquence de simulation.
C'est l'état initial de tous les regs et nets. C'est aussi l'état qui apparaît lorsqu'un conflit entre 1 et 0 ne peut être résolu.
Cet état n'est pas considéré comme une erreur, mais comme un état à part entière qui est propagé par application d'une extension de l'algèbre de Boole :

```
1 & X --> X          1 | X --> 1          ~X --> X
0 & X --> 0          0 | X --> X
```

En sortie, il sert à détecter un manque d'initialisation ou un conflit.
En entrée, il peut être utilisé pour faire une analyse de déterminisme : exemple :
Supposons qu'on veuille prouver que lorsque le bit `a[7]` de l'entrée `a[7:0]` d'un circuit est nul le résultat est indépendant des 7 autres bits de `a`. Pour cela il suffit de faire :

```
a = 'b0xxx_xxxx;
```

et d'observer que aucune des sorties n'est à X (on évite ainsi de devoir essayer 128 valeurs de `a`).

L'état Z est l'état haute-impédance. Il traduit un phénomène physique réel : le noeud "flottant". Il apparait lorsqu'une entrée est "en l'air" ou lorsqu'un bus bidirectionnel n'est contrôlé par aucun des drivers tri-state qui y sont connectés.
Lors de l'évaluation des expressions logiques, il est d'abord converti en état X.

© J.L.N.2005

Les ***conflits*** apparaissent lorsque plusieurs sources de signal (sorties ou ports bidirectionnels ou assignements continus) sont connectées sur le même net, et tendent à le forcer à des états contradictoires.
Bien entendu, l'état Z ne contredit aucun autre vu son caractère passif.
(Mais l'état X contredit tous les autres).

Une échelle de ***forces*** permet de contrôler la résolution des conflits.
Si les sources ont des forces différentes, la plus forte impose sa valeur, si elles ont des forces égales, la contradiction se traduit par l'état X du noeud.
Echelle des 5 forces : `supply, strong, pull, weak, highz.`
La force par défaut est : `strong`.

Exemple :
Modélisation de portes NMOS ou DCFL-MESFET ou "open-collector" avec pull-up passif :

```
module DCFL_nor2( s, a, b );
output s; input a, b;
assign (strong0, pull1) #(2.5, 2.0) s = ~( a | b );
endmodule
```

En "court-circuitant" les sorties de plusieurs de ces portes, on effectue un "et câblé" puisque les conflits se résolvent en faveur du 0.

Autre exemple : forçage des noeuds d'un circuit pour initialisation :
On ajoute un assignement continu (conflictuel) sur un noeud déjà connecté à une sortie de cellule, avec une force `supply` supérieure à la force par défaut.
Pour relâcher le forçage, on donne à cette source la valeur Z.

7. Blocs Procéduraux

7.1 Règles

La modélisation procédurale est contenue dans des blocs procéduraux (ou dans les tâches et fonctions appelées par ces blocs). Les blocs procéduraux sont des portions de programme qui sont exécutées en parallèle.
Il n'y a pas de hiérarchie des blocs procéduraux, un bloc ne peut pas en appeler un autre.
(Par contre un bloc procédural peut appeler des tâches et fonctions).
Les blocs procéduraux ne sont pas appelés, mais démarrent tous simultanément à l'instant zéro.
Les blocs procéduraux sont obligatoirement contenus dans des modules.
L'ordre de leurs déclarations n'a aucune incidence sur leur exécution.
Ils ne contiennent ni instances, ni assignements continus.

© J.L.N.2005

Exemples :
bloc ***initial*** (exécuté une seule fois) : voir §2.1.3
bloc ***always*** (réitéré indéfiniment) : voir §2.2

7.2 Temporisation procédurale

Les conditions de temporisation procédurale `#, @ et wait` peuvent apparaître soit dans l'en-tête d'un bloc `initial` ou `always`, soit en préfixe d'une instruction dans un bloc

7.2.1 délai relatif

Le caractère # suivi d'un nombre ou d'un expression numérique introduit une attente avant l'exécution d'une instruction ou d'un bloc (et retarde d'autant toute la suite du bloc).
On ne peut spécifier le temps de manière absolue (mais en cas de besoin, on peut créer un bloc initial contenant seulement l'instruction dont on veut spécifier l'instant d'exécution de façon absolue.)

7.2.2 attente de transitions ou de front

Le caractère @ suivi d'une liste de signaux entraîne l'attente de la prochaine transition d'un des signaux de la liste. Sous cette forme, cette condition permet la modélisation de circuits combinatoire de manière procédurale, exemple :

```
always @ ( A or B or C )
  begin
  .....
  RESU = ....
  end
```

Chaque fois que l'un des signaux d'entrée A, B, ou C change d'état, la sortie RESU est re-calculée (comparable au "process" de VHDL).

Avec les mots-clefs `posedge` (front montant) et `negedge` (front descendant) on peut choisir quel front va être attendu, ce qui nous fournit l'outil de base de modélisation des systèmes **synchrones** :

```
@ (posedge clk)
```

attendra le prochain front actif d'horloge.

7.2.3 attente d'un niveau

Le mot-clef `wait` suivi d'un expression logique va faire attendre que cette expression prenne une valeur "vraie" (non-nulle). Si l'expression est vraie au moment de l'exécution, il n'y a pas d'attente.

N.B. ne pas confondre `wait` et `if` : `wait` bloque l'exécution tant que la condition n'est pas satisfaite, alors que `if` saute immédiatement à l'instruction suivante.

© J.L.N.2005

8. Constructions Procédurales

A l'intérieur d'un bloc procédural, on trouve des instructions simples ou composées, éventuellement préfixées de conditions de temporisation.

Une instruction simple peut être :
- un assignement procédural
- un appel de tâche
- une construction itérative (boucle) ou décisionnelle (*if* ou *case*).

Dans les deux premiers cas, elle doit être terminée par le caractère ";"

Une instruction composée est une séquence d'instructions encadrée par les mots-clefs `begin` et `end`.
Les constructions sont très similaires au langages C, à deux différences près :
- présence des conditions de temporisations,
- `begin` et `end` à la place de { et }.

N.B. Il n'y a pas de danger à mettre plus de couples `begin - end` qu'il n'est strictement nécessaire, et cela peut améliorer la lisibilité du texte (exemple § 2.2.1)

8.1 Itérations ou boucles

8.1.1 boucle “for”

similaire au C :
```
for ( <assignement>; <condition>; <assignement> )
    <instruction>
```
exemple :
```
for ( i = 0; i < IMAX; i = i+1 )
    begin
    @ (posedge clk) a = a >> 1;
    #11 x = a & 1;
    end
```
N.B. la variable de contrôle doit avoir été déclarée (integer en général)

8.1.2 boucle “while”

similaire au C :
```
while ( <condition> )
     <instruction>
```
exemple :
```
while ( a[7] == 0 )  // justification à gauche de a[7:0]
      a = a << 1;
```

© J.L.N.2005

8.1.3 boucle “repeat”

```
repeat ( <expr> )
       < instruction >
```
exemple :
```
repeat (3) begin
              a = 1;
           #t a = 0;
           #(t*3) ;   // le ";" représente une inst. vide
           end
```
La variable qui sert a compter les "tours de boucle" est cachée, on ne peut ni la déclarer ni y accéder explicitement.

8.1.4 boucle “forever”

```
forever <instruction>
```
exemple
```
forever #(p/2) clk = ~clk;
```

Attention : Il faut impérativement éviter une boucle capable de faire un nombre illimité de "tours" en un temps de simulation nul. Ceci <u>bloque</u> irrémédiablement la simulation sans que l'utilisateur soit averti.
Exemple :
```
while ( a == 0 ) begin end // attendre que a passe a 1
```
bloque la simulation sans attendre que a passe à 1, doit être remplacé par
```
wait ( a ) ;
```
Autre exemple : l'exemple du § 8.1.2 va bloquer la simulation si a[7:0] est nul. Si cela risque de se produire, il faut modifier le programme comme suit :
```
i = 7;
while (  ( a[7] == 0 ) && ( i )  ) // justification à gauche
      begin  a = a << 1; i = i - 1;  end
```

8.2 Décisions

8.2.1 décision “if”

similaire au C :
```
if ( <condition> ) <instruction>
else               <instruction>
```
ou simplement :
```
if ( <condition> ) <instruction>
```
exemple : cf § 2.2.1

© J.L.N.2005

8.2.2 expression conditionnelle

similaire au C :

```
( <condition> ) ? <instruction> : <instruction>
```

N.B. peut aussi s'utiliser dans un assignement continu !

exemple : cf § 9.2 et § 10.1.1

8.2.3 décision “case”

très différent du C !

```
case ( <expression> )
<constante> : <instruction>
   ....           ....
default     : <instruction>
endcase
```

exemple :

```
case ( opcode )       // ALU combinatoire
  1 : s = a & b;
  2 : s = a | b;
  5 : s = a + b;
  7 : s = ~a;
default : s = 'bXXXXXXXX_XXXXXXXX;
endcase
```

9. Encore quelques exemples

9.1 bascule D avec clear asynchrone

Soit à faire le modèle d'une bascule D sensible au front montant, avec une entrée de remise à zéro asynchrone (prioritaire) sensible au niveau :

```
module LibDFFc ( q, ck, d, clear );
input ck, d, clear; output q;
reg iq;
  always @ ( posedge ck )     // <== sensible au front
     if ( clear == 0 ) iq = d;
  always wait ( clear == 1 )  // <== sensible au niveau
     begin
     iq = 0;
     wait ( clear == 0 ) ;
     end
  assign #8 q  =  iq;
  specify  // timing a verifier : setup 5.2 ns, hold 1.5 ns
    $setuphold( posedge ck, d, 5.2, 1.5 );
  endspecify
```

© J.L.N.2005

```
endmodule
```

N.B. Si on oublie la condition `wait ( clear == 0 )` à la fin du second bloc always, celui-ci va boucler un nombre indéfini de fois dès que clear sera à 1 ==> simulation bloquée (cf § 8.1). Remarquer aussi que cette condition préfixe une instruction vide, représentée par un `";"` (obligatoire).
Le bloc "specify" sert ici à demander au simulateur des vérifications de "timing", dans le cas présent il devra signaler par un "warning" toute transition de `d` qui surviendrait entre 5.2 ns avant et 1.5 ns après le front montant d'horloge.

9.2 Buffer tri-state

```
module LibTRI ( out, in, out_ena );
inout out; input in, out_ena;
  assign (strong0, highz1)  #1 out = in | ~out_ena;
  assign (highz0,  strong1) #1 out = in &  out_ena;
endmodule
```

variante :

```
module LibTRI ( out, in, out_ena );
inout out; input in, out_ena;
  assign #1 out = ( out_ena ) ? in : 'bz;
endmodule
```

9.3 Cellule paramétrée

L'utilisation de paramètres permet de faire des descriptions “génériques”.
exemple : le registre à décalage du § 2.2.1 généralisé à N bits :

```
// registre piso de N bits
module pisoN( clk, load, par_in, ser_out );
parameter N=4;
input clk, load; input [(N-1):0] par_in;
output ser_out;
reg [(N-1):0] contenu;
  always @ (posedge clk)
    contenu = (load)?(par_in):(contenu >> 1);
  assign #3 ser_out = contenu[0];
endmodule
```

exemple d'instance d'un piso de 16 bits :

```
defparam piso_inst.N = 16;
pisoN piso_inst ( ck, load, data_in, ser_data );
```

La directive `defparam` a permis de fixer une valeur de N différente de la valeur par défaut.

© J.L.N.2005

10. Introduction à la Synthèse Logique

C'est la technique qui consiste à utiliser un logiciel de synthèse pour obtenir automatiquement une description structurelle (utilisant les cellules d'une bibliothèque préalablement définie) à partir d'une description comportementale.
(Accessoirement cette technique peut aussi être utilisée pour optimiser une description structurelle existante, ou l'adapter à une bibliothèque différente ("reciblage").)
Il faut savoir qu'il ne suffit pas qu'une description comportementale donne un résultat correct à la simulation pour qu'elle soit "synthétisable", il y a de nombreuses restrictions dans l'emploi des primitives du langage. De plus un logiciel de synthèse utilisé sans précautions est capable de produire un circuit N fois plus compliqué qu'il n'est nécessaire.
On donnera ici quelques directives générales, en précisant bien que chaque logiciel de synthèse impose des contraintes particulières.

10.1 Modules combinatoires

10.1.1 description concurrente

La solution "sans problèmes" est la description concurrente à base d'**assign.**
Elle s'applique spécialement bien aux cas où on dispose déjà d'expressions booléennes (non nécessairement minimales) des fonctions à réaliser : exemple : le multiplexeur du § 2.2.3.

```
module LibMux( i0, i1, s, y );
input i0, i1, s; output y;
  assign #1 y = ( i0 & ~s ) || ( i1 & s );
endmodule
```

On peut aussi exploiter d'autres possibilités des expression Verilog :

```
module LibMux( i0, i1, s, y );
input i0, i1, s; output y;
  assign #1 y = (s)?(i1):(i0);
endmodule
```

variante (généralisable à N entrées) :

```
module LibMux( ivec, s, y );
input s; input [1:0] ivec; output y;
  assign #1 y = ivec[s];
endmodule
```

autre exemple : additionneur-soustracteur 8 bits :

```
module addsous( A, B, sign, X );
input [7:0] A, B; input sign; output [8:0] X;
  assign X = (sign)?(A+B):(A-B);
endmodule
```

© J.L.N.2005

10.1.2 description procédurale

Si la spécification du bloc combinatoire se présente sous forme d'une table de vérité, on peut vouloir éviter l'extraction manuelle des expressions booléennes des sorties.
Pour décrire le module directement sous forme de table, on peut utiliser la décision "case" (voir § 8.2.3) dans un bloc procédural déclenché par les transitions des entrées :

```
module tableau( A, B, Z );
input [2:0] A; input B; output [4:0] Z;
reg [4:0] rZ;
always @ ( A or B )
  case ( { A, B } )
     'b0001 : rZ = 'b10001;
     'b0100 : rZ = 'b01001;
     'b0110 : rZ = 'b01111;
    default : rZ = 'b10000;
  endcase
assign Z = rZ;
endmodule
```

N.B : on a dû déclarer un reg pour la sortie de la table. Le reg ayant en général une capacité de mémorisation, le programme de synthèse risque de mettre des bascules, ce qui est contraire à l'objectif de faire un circuit combinatoire.
Heureusement l'effet de mémorisation est évité si on respecte :

Les deux conditions de modélisation procédurale d'un bloc combinatoire :

- **toutes les variables** intervenant dans l'évaluation du reg apparaissent dans la condition de déclenchement du bloc always,
- le reg est re-évalué dans **tous les cas** d'éxécution du bloc

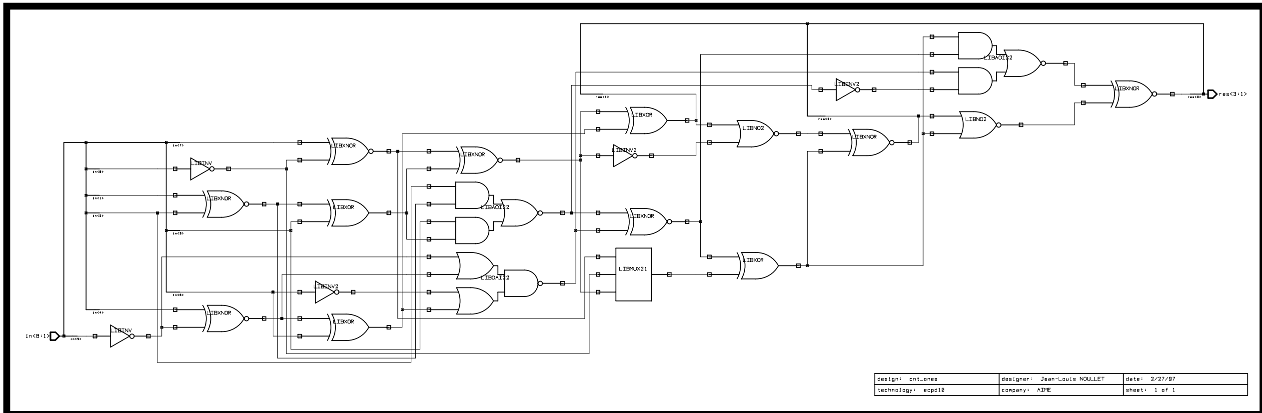
Dans l'exemple ci-dessus, la clause default du case permet de garantir la seconde condition.

D'une façon plus générale, un circuit combinatoire peut être décrit par une construction procédurale si les deux conditions ci-dessus sont satisfaites.
Un autre exemple : supposons qu'on veuille faire un module comptant les "uns" du vecteur d'entrée :

```
module compte_uns( in, res );
input [2:0] in; output [1:0] res;
reg [1:0] r; integer cnt;
always @ (in)
  begin
  r = 0;
  for ( cnt = 0; cnt < 3; cnt = cnt + 1 )
      r = r + in[cnt];
  end
assign res = r;
endmodule
```

© J.L.N.2005

Bien que l'algorithme soit séquentiel, le circuit synthétisé par logiciel *Synopsys* sera purement conbinatoire.



10.2 Modules Séquentiels Synchrones

La description fait appel à un bloc procédural, de type

```
always @ (posedge clk)
```

décrivant les actions à effectuer entre deux fronts actifs d'horloge.
Ce bloc ne doit contenir **aucune temporisation** autre que celle de l'en-tête.

Ce bloc constitue la boucle unique de la description, il ne doit pas en général contenir de boucle interne genre "while" ou "for".

Les signaux de type `reg` étant doués de mémoire, le logiciel va avoir tendance à placer une bascule pour chaque bit de `reg` recevant des valeurs à l'intérieur du bloc always.
Il faut donc soigneusement évaluer les besoins en mémoire du système projeté et ne déclarer que le minimum de `reg`s nécessaires.
exemple : le registre du § 2.2.1 est décrit dans un style parfaitement synthétisable.

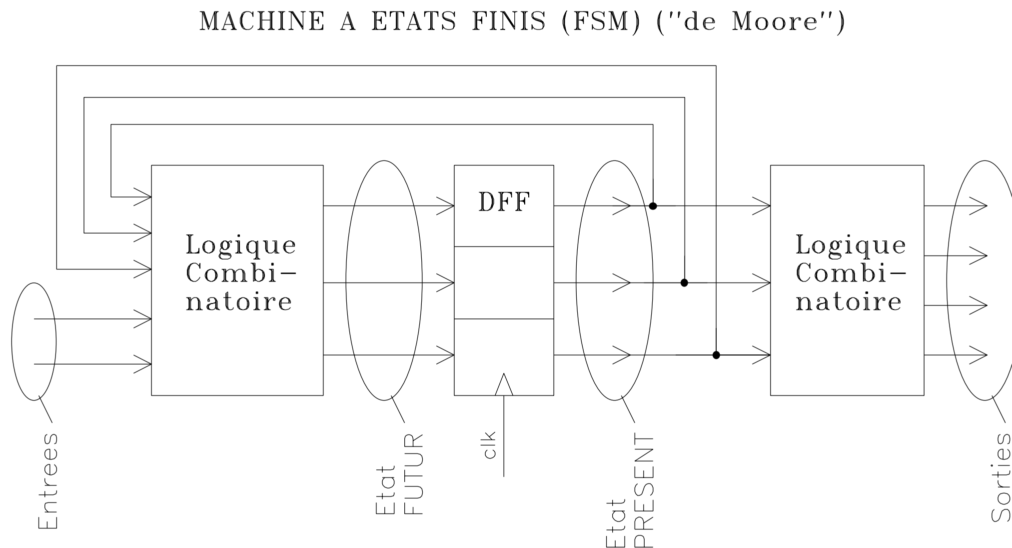
Il est également possible qu'un même module contienne plusieurs blocs procéduraux, dont un seul déclenché par le font d'horloge, et un ou plusieurs autres représentant des parties combinatoires respectant les deux conditions expliquées au § 10.1.2.

Attention : un même reg ne doit pas subir des affectations dans deux blocs procéduraux ayant des conditions de déclenchement différentes. Ceci aboutirait a une **impossibilité de synthèse.**
(Ainsi l'exemple du § 9.1 n'est pas synthétisable)

© J.L.N.2005

10.3 Machine à états finis

Considérons une machine à états finis ou automate synchrone selon le modèle “de Moore” et décrivons la dans le style “RTL” :




```
module moore_fsm( entrees, sorties, clk );
input [1:0] entrees; input clk; output [3:0] sorties;
reg [2:0] etat_futur, etat_present;
reg [3:0] r_sorties;
always @ (posedge clk)
     etat_present = etat_futur;
always @ (etat_present or entrees)
     begin
       < calcul etat_futur >
     end
always @ (etat_present)
     begin
       < calcul r_sorties >
     end
assign sorties = r_sorties;
endmodule
```

Naturellement il est possible de remplacer le second ou le troisième bloc procedural par une description concurrente (à base d’*assigns*).

© J.L.N.2005

Variante ou l'état futur n'apparait pas explicitement :

```
module moore_fsm( entrees, sorties, clk );
input [1:0] entrees; input clk; output [3:0] sorties;
reg [2:0] etat_present;
reg [3:0] r_sorties;
always @ (posedge clk)
     begin
       < calcul etat_present >
     end
always @ (etat_present)
     begin
       < calcul r_sorties >
     end
assign sorties = r_sorties;
endmodule
```

10.4 Registre avec clear asynchrone

La description de bascule du 9.1 est correcte mais n'est pas acceptée par certains programmes de synthèse. On utilise alors la variante suivante (qui n'est pas strictement correcte) :

```
always @ ( posedge ck or posedge clear )
   if    ( clear )
         iq = 0;
   else  iq = d;
```

De cette manière on peut facilement introduire le clear asynchrone dans une description de type RTL comme celle du § 10.3.

© J.L.N.2005

11. Ce qui resterait à voir

Ce document n'est pas le manuel de référence du langage Verilog mais seulement une introduction. Il faut savoir qu'il existe de nombreuses possibilités complémentaires...

- primitives du simulateur, soit modules pré-définis représentant les portes logiques de base
- primitives définies par l'utilisateur (UDP) par représentation tabulaire (comb. et seq.)
- modélisation MOS au niveau transistor, modélisation MOS dynamique (effets capacitifs)
- échelle des 3 cas de temps de propagation : min, typ, max
- modèles de propagation (inertiel et transport)
- "path delay specify" : spécifications des temps de propagation par chemins
- assignement continu incorporé à la déclaration de net
- délais, forces incorporés à la déclaration de net
- types de net autre que wire
- instances avec ports non connectés
- conflits de types de net aux ports d'une instance
- output port déclaré comme reg --> assignement continu implicite
- nombres réels
- "command line options" : options d'invocation, organisation des bibliothèques ("libraries")
- mode interactif et mise au point ("debugging", "patching")
- "tasks", "functions" : tâches et fonctions définies par l'utilisateur (= sous-programmes)
- fonctions et tâches pré-définies autres que celles utilisées dans les exemples ($xxxx)
- accès à des fichiers de données
- directives de compilation `default_nettype, `define, `unconnected_drive, etc..
- événements nommés ("named events")
- parallélisme à l'intérieur d'un bloc procédural ("fork", "join")
- "intra-assignement timing" : délai à l'intérieur d'un assignement procédural

© J.L.N.2005