SystemVerilog pour la verification

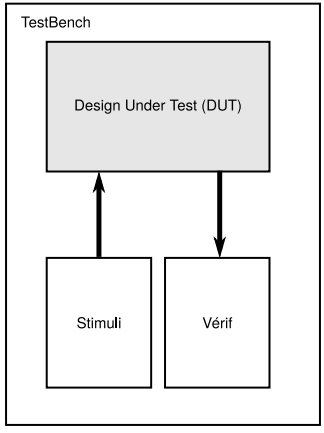
Testbenches avancés

Tarik Graba

Année 2019/2020

Plan

Testbenches Verilog



Testbenches Verilog

- Un module pour le **testbench**
- Une instance du module testé (DUT)
- Générer des stimuli vers les entrées du DUT et récupérer les sorties
  - **initial**, **always**...

```
module testbench;

  // déclaration des signaux
  ...

  module_a_tester DUT(.*);

  initial
  begin
     // générer des stimuli
     ...
  end

  initial
  begin
     // observer les sorties
     ...
  end

endmodule
```

Plus qu'un simple testbench

Pourquoi ?

Cette façon de faire des testsbench n'est plus suffisante :

- Complexité :
  - des fonctionnalités complexes
  - des protocoles de bus complexes
  - des fournisseurs différents
- Aller au-delà du test fonctionnel :
  - détecter des bugs avant fabrication
  - en dehors des spécifications
- Diviser le travail :
  - qui définit les tests ?
  - comment les rendre modulaires et réutilisables ?

Plus qu'un simple testbench

Comment ?

- Constrained Random Verification :
  - des stimuli générés aléatoirement
    - des contraintes pour cibler les cas intéressants
  - des assertions
    - pour détecter les cas interdits/problématiques
  - analyser la couverture des tests
    - a-t-on testé tous les cas intéressants

Vérification en SystemVerilog

Qu'apporte le langage ?

- Standardisation de l'ordonnancement du simulateur pour séparer «*design*» et «*testbench*»
- Le langage intègre des constructions pour :
  - les assertions
  - la génération d'aléas sous contraintes
  - l'analyse de la couverture
- Le langage intègre des constructions pour construire et maintenir des tests complexes
  - classes/programmation orientée objet

Vérification en SystemVerilog

Comment on s'en sert ?

Comment ne pas réinventer la roue systématiquement !

- Des méthodologies ont été proposées et standardisées :
  - VMM : Verification Methodology Manual…
  - OVM : Open Verification Methodology
  - UVM : Universal Verification Methodology
- C'est UVM qui est le standard le plus utilisé actuellement.

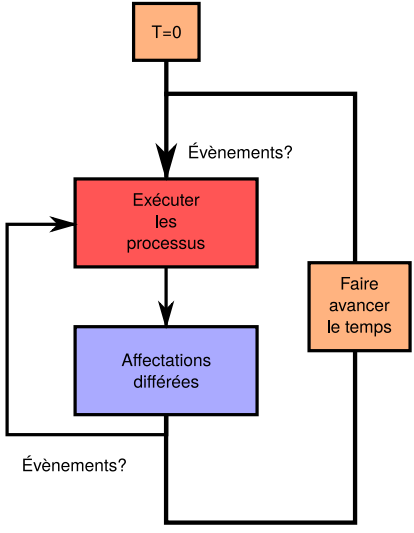
Ce cours ne présente pas ces méthodologies mais plutôt les briques du langage qui les permettent !

Plan

Simulation évènementielle

Rappel

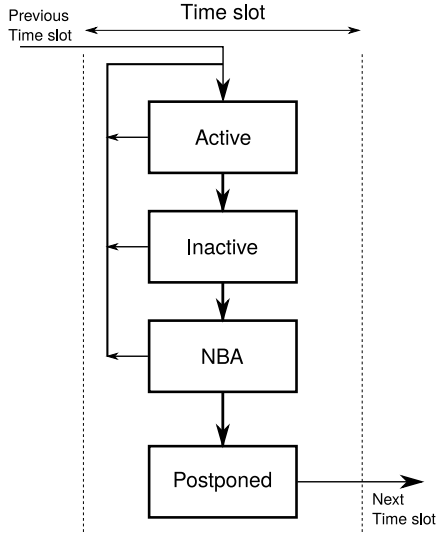
```
T = 0;
While( Event ) {
   While( Event at t=T ) {
      RunProcess
      ApplyDelayedAssignment
   }
   T = AdvanceToNextTime
}
End
```



Simulation évènementielle

En Verilog 2001

Pour un temps physique donné («*time slot*»), l'ordonnancement est divisé en plusieurs régions.

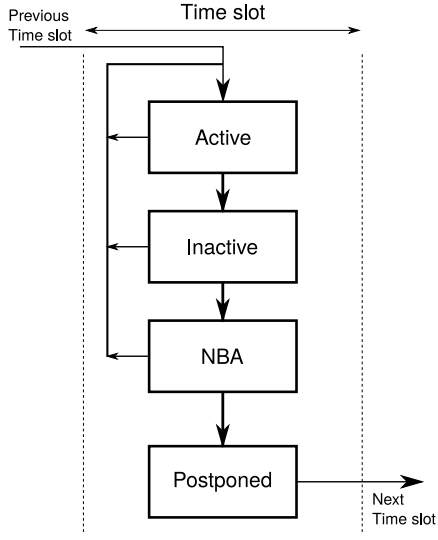


Simulation évènementielle

En Verilog 2001

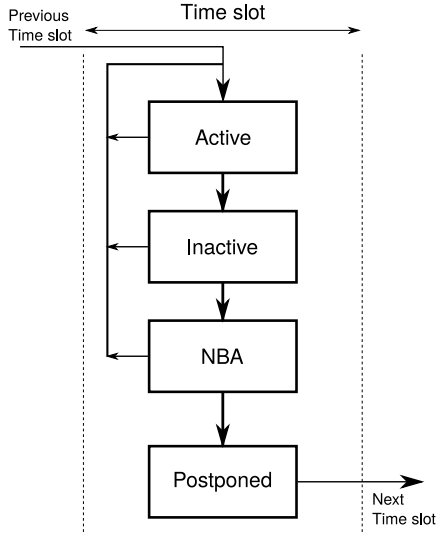
Active

- les affectations immédiates des processus **`always`** ou **`initial`**,
- les affectations concurrentes **`assign`**,
- propagation des I/O (**`in`**, **`out`**...)
- l'évaluation de ce qui se trouve à droite des affectations



Simulation évènementielle

En Verilog 2001

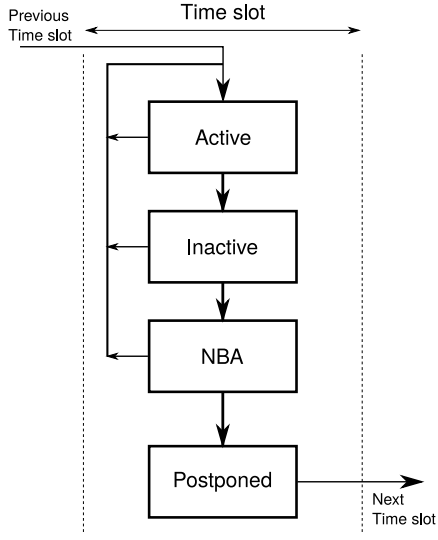


Inactive

- les affectations à retard nul (**#0**)

Simulation évènementielle

En Verilog 2001



NBA : Non blocking assignment

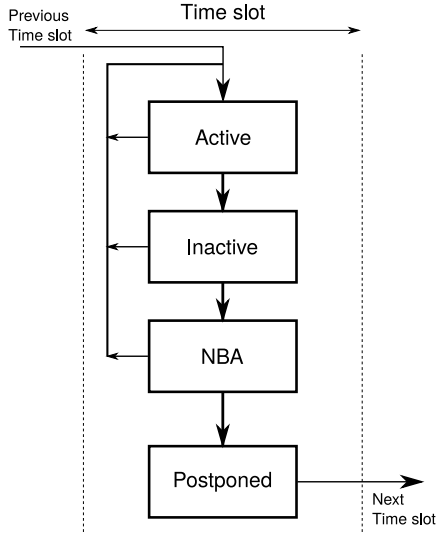
- les affectations différées sont appliquées (<=)

Simulation évènementielle

En Verilog 2001

Postponed

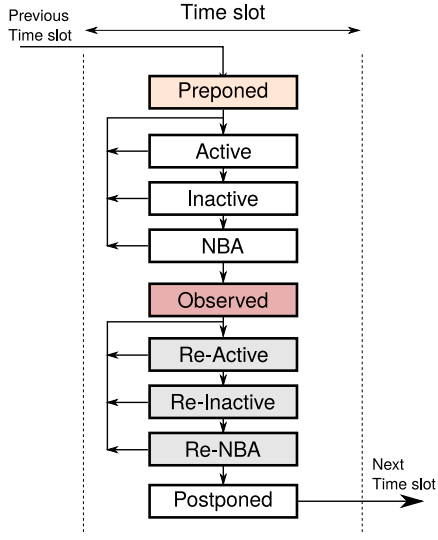
- pour la fonction système **`$monitor`**



Simulation évènementielle

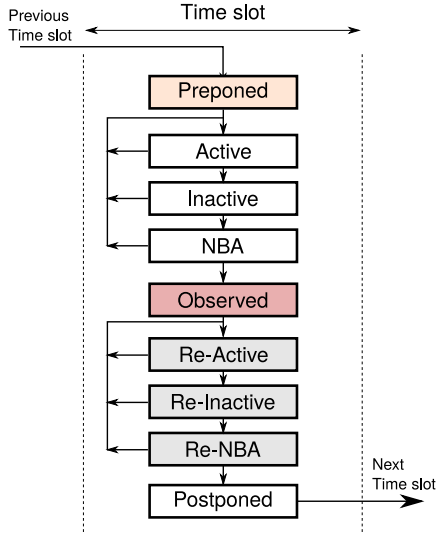
En SystemVerilog

Ajouts de régions (non blanches dans le diagramme) pour permettre de nouvelles fonctionnalités.



Simulation évènementielle

En SystemVerilog



Preponed

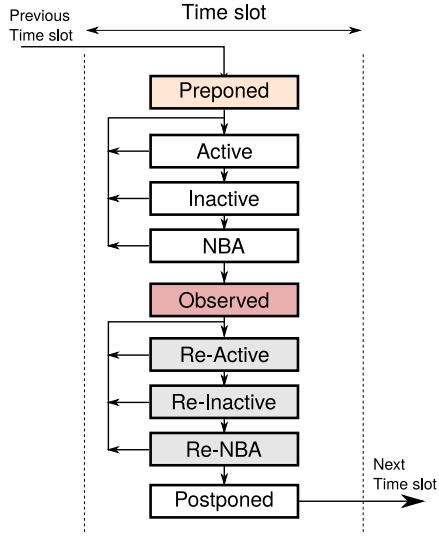
- permet de lire les valeurs (échantillonner) avant le début du *time slot*.

Simulation évènementielle

En SystemVerilog

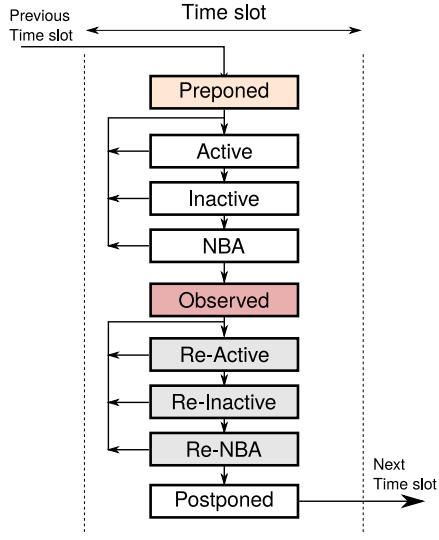
Les régions actives

- celles de Verilog,
- destinées au RTL



Simulation évènementielle

En SystemVerilog



Observed

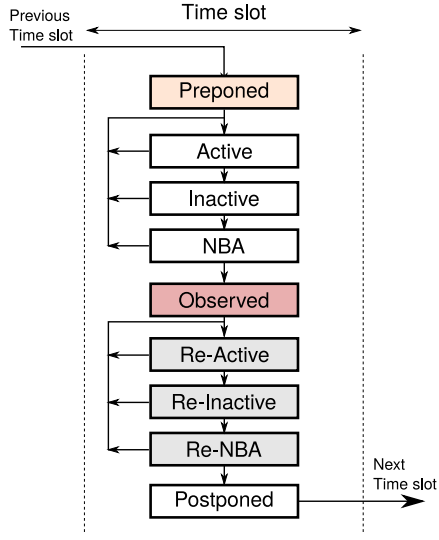
- permet d'observer le résultat de la région active
- destinée aux assertions

Simulation évènementielle

En SystemVerilog

Les régions réactives

- permet d'exécuter des processus liés à la simulation
- les mêmes régions que la zone active



Simulation évènementielle

Pourquoi cette complexité ?

Pour séparer le Design du testbench

- Des constructions différentes (**`program`**)
- On évite les problèmes de concurrence (*race conditions*)

Ajouter des fonctionnalités

- Des assertions concurrentes (*concurrent assertions*)
- Simplifier la simulation de designs synchrones (*clocking blocs*)

Plan

Les «program»

Un module pour la simulation

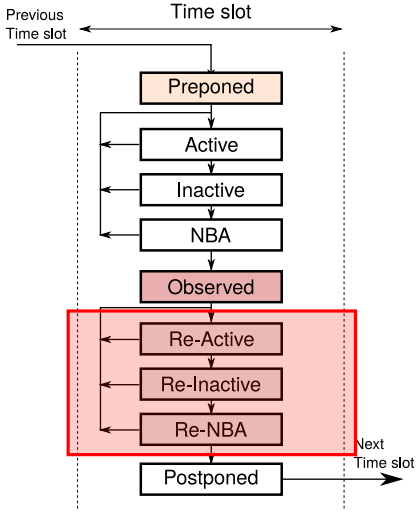
+ Se déclare comme un module
+ Comme un module, il a une interface
+ On peut l'instancier dans un autre module

- Ne peut pas contenir de sous modules ni de sous-programmes
- Ne doit pas contenir de processus **always**, **always_comb**, **always_ff**, … seulement des processus **initial** (ou **final**)

```
program tester (...);

  initial
  begin
     // générer des stimuli
     ...
  end

  initial
  begin
     // observer les sorties
     ...
  end

endprogram

module testbench;

  // déclaration des signaux
  ...

  module_a_tester DUT(.*);

  tester TESTER_i(.*);

endmodule
```

Les «program»

Un module pour la simulation



Pourquoi ?

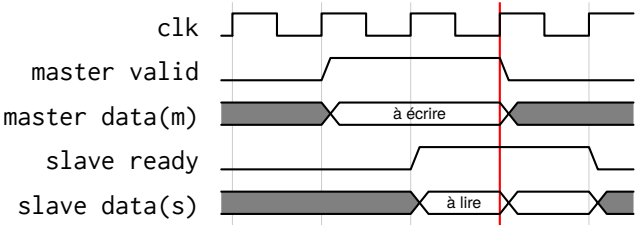
- Isoler/séparer le «design» matériel du «testbench»
- Ses processus sont exécutés dans la région réactive
  - moins de problèmes de concurrence

Remarque : La simulation s'arrête automatiquement quand tous les processus des programmes se finissent.

Les «program»

Exemple : Un protocole

Le protocole de l'interface **foo** permet au maitre d'écrire une donnée et d'en récupérer une autre.



Les «program»

Exemple : Une interface

- une entrée d'horloge `clk` et de reset `nrst`
- 2 **`modport`** pour un maitre et un esclave

```
interface foo (input bit clk,
               input logic nrst
              );

    logic [7:0] m;
    logic [7:0] s;
    logic  valid;
    logic  ready;

   modport M (
    input   clk,
    input   nrst,
    output  m,
    input   s,
    output  valid,
    input   ready
    );

   modport S (
    input   clk,
    input   nrst,
    input   m,
    output  s,
    input   valid,
    output  ready
    );

endinterface:foo
```

Les «program»

Exemple : Un esclave

```
module slave( foo.S I );

localparam [7:0] Sec = 8'b1111_0000;

logic[7:0] R;

always_ff@(posedge I.clk or negedge I.nrst)
   if (!I.nrst)
   begin
      R       <= '0;
      I.ready <=  1'b1;
   end
   else
   begin
      I.ready <= 1'b1;

      if(I.valid & I.ready)
      begin
         R       <= I.m;
         I.ready <= 1'b0;
      end
   end

assign I.s = R ^ Sec;

endmodule
```

- Renvoie valeur précédemment écrite sur laquelle un calcul est effectué
- `ready` par défaut
- insère un cycle d'attente après chaque requête valide

Les «program»

Exemple : testbench de base

- Un module sans entrée ni sortie
- Les signaux et interfaces
- Une instance de l'esclave (DUT : Design Under Test)
- Une instance du testeur
- Générer l'horloge

```
module testbench();

  bit   clk;
  logic nrst;

  foo I(.*);

  tester tester_i(.*);
  slave  DUT(.*);

  always #10ns clk = !clk;

endmodule
```

Les «program»

Exemple : program pour un test de base

```
program tester( foo.M I , output logic nrst);
   bit[7:0] data;

   initial
   begin:main
      I.m     <= '0;
      I.valid <=  0;
      nrst    <=  0;
      // Attendre 2 cycles d'horloge
      repeat(2) @(negedge I.clk);
      nrst    <= 1;
      repeat(2) @(negedge I.clk);
      repeat(10)
      begin
         I.valid <= 1;
         I.m     <= data;

         while(!I.ready) @(posedge I.clk);

         $display("w %0h & r %0h",data, I.s);
         data++;
         @(negedge I.clk);
         I.valid <= 0;
         repeat(1+$random%3)@(negedge I.clk);
      end
   end:main
endprogram
```

- contrôle le signal de reset
- génère des séquences vers l'esclave
- récupère les sorties de l'esclave

On aurait pu déclarer le **program** à l'intérieur du testbench pour qu'il ait directement accès à ses signaux

Les «program»

Est-ce suffisant ?

Quand on simule de la logique synchrone :

- Comment être sûr du synchronisme ?
- Comment le garantir quand le testbench se complexifie ?
- Comment le garantir quand :
  - quand l'horloge change ?
  - quand les fronts utilisés changent ?
  - quand on a des temps de propagation ?

Plan

Les «clocking block»

Un «clocking block» est une construction qui permet dans un testbench de :

- déclarer l'évènement de synchronisation (le front de l'horloge)
- les entrées à capturer au front d'horloge
- les sorties à modifier après le front

Un «clocking block» est déclaré entre les mots clés **`clocking`** et **`enclocking`** dans un **`module`**, **`program`** ou **`interface`**.

Les «clocking block»

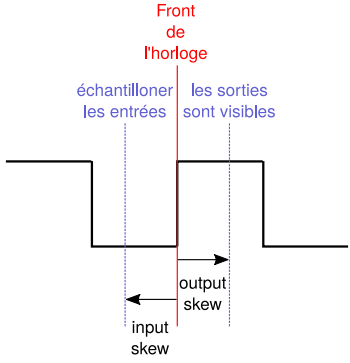
La syntaxe

```
clocking foo @(posedge clk);
   input  sig1;
   input  sig2;
   output sig3;
   output sig4;
   ....
endclocking

// avec I/O skews
clocking bar @(negedge clk);
   default input 1ns output 2ns;
   input  sig1;
   input  sig2;
   output sig3;
   output sig4;
   ....
endclocking

...
foo.sig3 <= x;
y <= bar.sig1;
...
@(foo) ...
```

- `foo` est un «clocking block» déclenché par le front montant de `clk`
- `bar` est un «clocking block» déclenché par le front descendant de `clk`
- on peut optionnel ment définir des «skews»
- On accède aux signaux à travers le «clocking block» par des affectations differées.
- `@(foo)` fait référence au front qui déclenche «clocking block»

Les «clocking block»

Les *skews*

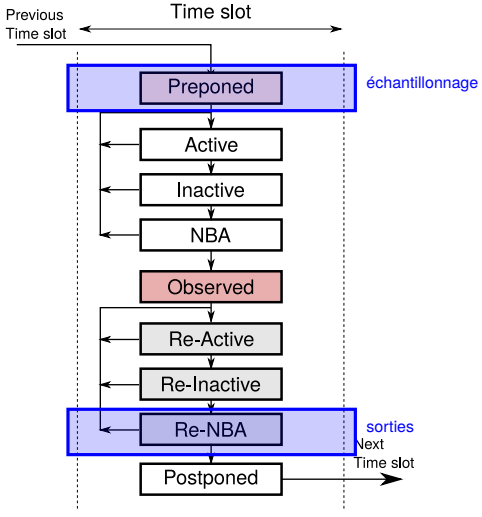


Par défaut :

- en entrée 1 cycle du simulateur (`1step`)
- en sortie un temps de `0`

Les «clocking block»

Comment est-ce fait par le simulateur



Les «clocking block»

Exemple : améliorons notre interface

- Clocking block dans l'interface
- Nouveau **`modport`** pour le testbench
- On peut y ajouter le reset

```
interface foo (input bit clk);

    logic nrst;
    logic [7:0] m;
    logic [7:0] s;
    logic  valid;
    logic  ready;

  // ici rien ne change
   modport M ( ... );
   modport S ( ... );

    clocking tb_cb @(posedge clk);
       output  nrst;
       output  m;
       input   s;
       output  valid;
       input   ready;
    endclocking

    // Le testeur est synchrone
    modport TB ( clocking tb_cb );

endinterface:foo
```

Les «clocking block»

Exemple : améliorons notre `program`

- On a la garantie que les sorties changeront après le front d'horloge
- On a la garantie que les entrées sont correctement échantillonnées

```
program tester( foo.TB I );

   bit[7:0] data;

   initial
   begin:main
      I.tb_cb.m     <= '0;
      I.tb_cb.valid <=  0;
      I.tb_cb.nrst  <=  0;
      // Attendre que les affectations se fassent
      @(I.tb_cb);
      // Attendre 2 cycles d'horloge
      repeat(2) @(I.tb_cb);
      I.tb_cb.nrst <= 1;
      repeat(2) @(I.tb_cb);
      repeat(10)
      begin
         I.tb_cb.valid <= 1;
         I.tb_cb.m     <= data;
         @(I.tb_cb);
         // ready ne change qu'au front d'horloge
         wait(I.tb_cb.ready==1);
         $display("w %0h & r %0h",data, I.tb_cb.s);
         data++;
         I.tb_cb.valid <= 0;
         repeat(1+$random%3)@(I.tb_cb);
      end
   end:main
endprogram
```

Les «clocking block»

Exemple : simplifions notre testbench

- Plus besoin de déclarer le signal de reset
- Le reset fait aussi partie du «clocking block»
- Comment faire si on veut simuler un reset asynchrone ?
  - On l'ajoute au **`modport`**

```
module testbench();

  bit   clk;
  foo I(.*);

  tester tester_i(.*);
  slave  DUT(.*);

  always #10ns clk = !clk;

endmodule
```

Autre choses ?

Que pouvons-nous ajouter dans une interface ?

Presque comme un module :

- des tâches (**task**)
- des fonctions (**function**)
- des processus (**always**, **assign**)
- …

Par exemple :

```
interface foo (input bit clk);
   ...
    function void init();
       nrst  = 0;
       valid = 0;
       m     = $random();
    endfunction
    ...
    // fonction visible dans le modport
    modport TB ( clocking tb_cb,
                 import init
               );
   ...
endinterface:foo
```

La fonction `init` permet d'initialiser de façon asynchrone les signaux du maitre.

Plan

Les classes en SystemVerilog

Orienté objet

```
class Message;
   byte text[];
   int id;
   logic[31:0] address;

   function void dump();
      $diplay("%p",this);
   endfunction
endclass

...
// On déclare une référence (handle)
Message m;
// on alloue l'objet
m = new;

m.text = "hello world";
...
xx = m.id;
...
m.dump();
```

SystemVerilog les classes sont réservées a la simulation.

- définition entre les mots clés **`class`** et **`endclass`**
- contient des «propriétés» (les données) et des «méthodes» (**`function`** ou **`task`**)
- le constructeur s'appelle toujours **`new`** (implicite si par défaut)
- un «garbage collector»
- peut avoir des paramètres (**`parameter`**)

Les classes en SystemVerilog

Orienté objet

Les classes SystemVerilog supporte l'héritage simple

- on doit utiliser le mot clé **`extends`**
- **`super`** fait référence à la classe père
- certains champs peuvent être marqués :
  - **`local`** (private en C++)
  - **`protected`**

```
program foo;
   class A;
      int i;
      function new (int i);
         this.i = i;
      endfunction

      function void dump();
         $display("%p", this);
      endfunction
   endclass

   class B extends A;
      int j = 0;
      function new();
         super.new(33);
      endfunction
   endclass

   A a;

   initial
   begin
     a = new(44);
     a.dump();
     a = B::new;
     a.dump();
   end
endprogram
```

Les classes en SystemVerilog

La génération d'aléa

Certaines propriétés d'une classe peuvent être tirées aléatoirement en y ajoutant l'attribut :

- **rand** : uniformément distribuées
- **randc** : cyclique (une valeur ne reviendra pas avant la fin du cycle)
- La méthode implicite **randomize** permet de tirer une nouvelle valeur
  - Elle renvoie la valeur `0` en cas d'échec, 1 sinon.

```
program foo;
   class A;
      rand  bit[3:0] i;
      randc bit[3:0] j;

      function void dump();
         $display("%p", this);
      endfunction
   endclass

   A a;

   initial
   begin
     a = new;
     repeat(20) begin
        if (a.randomize()) a.dump();
     end
   end
endprogram
```

Les classes en SystemVerilog

Contraintes à la randomisation

A la génération on peut ajouter des contraintes en utilisant le mot clé **`with`**

```
program foo;
   class A;
      rand  bit[3:0] i;
      randc bit[3:0] j;

      function void dump();
         $display("%p", this);
      endfunction
   endclass

   A a;
   int res;

   initial
   begin
     a = new;
     repeat(20) begin
        res = a.randomize()
            with {i[1:0] == 0 && j%2 ==1;};
        if (res) a.dump();
     end
   end
endprogram
```

Les classes en SystemVerilog

Contraintes à la déclaration

On peut ajouter des contraintes à la déclaration de la classe en utilisant le mot clé **`constraint`**

Si les contraintes ne sont pas compatibles, la randomisation échouera.

```
program foo;
   class A;
      rand  bit[3:0] i;
      randc bit[3:0] j;

      function void dump();
         $display("%p", this);
      endfunction

      constraint C  {i[1:0] == 0 && j%2 ==1;}
      // dans un ensemble/intervalle
      constraint C1 {j inside {[1:10],15} ;}
      // implication
      constraint C2 {i<5 -> j>5 ;};
      //  la distribution
      constraint C3
      { i dist {0:=1, [1:10]:= 5, [11:15]:=2 }; };
   endclass

   A a;
   int res;

   initial
   begin
     a = new;
     repeat(40) begin
        if (a.randomize()) a.dump();
     end
   end
endprogram
```

Les classes en SystemVerilog

Comment faire évoluer notre TB

On peut utiliser les classes pour les générer des requêtes aléatoires.

```
program tester( foo.TB I );
   // classe de base pour les requêtes
   class BaseRequest;
      rand bit[7:0] data;
      rand int      delay;
   endclass

   // Spécialisation des requêtes en fonction du délai
   typedef enum {nodelay, shortdelay, longdelay} delay_t;

   class Request extends BaseRequest;
      rand delay_t dtype;
      constraint delay_range {
          (dtype == nodelay   ) -> delay == 0;
          (dtype == shortdelay) -> delay inside {[1:4]};
          (dtype == longdelay ) -> delay inside {[5:15]};
      }
   endclass
...
```

Les classes en SystemVerilog

Comment faire évoluer notre TB

On peut utiliser les classes pour les générer des requêtes aléatoires.

```
...
   // des tâches pour cacher les détails
   task reset();
      I.tb_cb.m     <= '0;
      I.tb_cb.valid <=  0;
      I.tb_cb.nrst  <=  0;
      @(I.tb_cb);
   endtask

   task unreset();
      I.tb_cb.nrst  <=  1;
      @(I.tb_cb);
   endtask

   task buswrite(input BaseRequest req, output [7:0] sdata );
         I.tb_cb.valid <= 1;
         I.tb_cb.m     <= req.data;
         @(I.tb_cb);
         wait(I.tb_cb.ready==1);
         sdata = I.tb_cb.s;
         I.tb_cb.valid <= 0;
         repeat(req.delay)@(I.tb_cb);
   endtask

   initial
      I.init();
...
```

Les classes en SystemVerilog

Comment faire évoluer notre TB

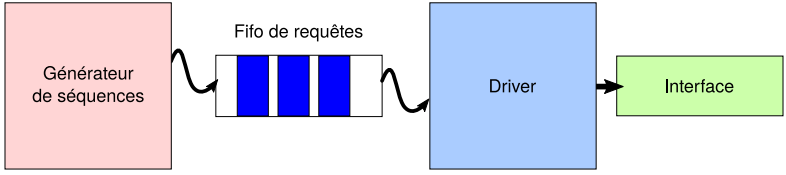
On peut utiliser les classes pour les générer des requêtes aléatoires.

```
...
   Request req = new;
   int res;
   bit[7:0] sdata;
   delay_t c_delay;

   initial
   begin:main
      reset();
      unreset();

      // On itère sur les différents type de délais
      c_delay = c_delay.first;
      do begin
         $display("Test with %p",c_delay);
         repeat(10) begin
            res = req.randomize() with {dtype == c_delay;};
            if ( res == 1 ) buswrite(req, sdata);
            else            $fatal(1,"Rondomistion failed");
            $display("w %02h & r %02h",req.data, sdata);
         end
         c_delay = c_delay.next;
      end
      while(c_delay != c_delay.first);
   end:main
endprogram
```

Les classes en SystemVerilog

Comment faire évoluer notre TB

On peut aussi augmenter l'abstraction en séparant la génération des séquences de test de leur application.



- Une classe «générateur de séquences»
- Une classe «driver» qui accède au bus
- Une fifo entre les deux pour les découpler temporellement

Les classes en SystemVerilog

Comment faire évoluer notre TB

Pour la file de requêtes on peut utiliser les **mailbox** :

- méthodes d'accès bloquantes (**put**, **get**)
- méthodes d'accès non-bloquantes (**try_put**, **try_get**)
- pas de contraintes de type
- par défaut de taille infinie

```
program babar;
   class A;
      rand bit[3:0]x;
      function void dump();
         $info("%p", this);
      endfunction
   endclass

   A a, b;
   mailbox m = new(5);

   initial
   begin
      fork begin
         repeat(10) begin
            a = new; void'(a.randomize());
            m.put(a);
            #10ns;
         end
      end
      begin
         #200ns;
         forever begin
            m.get(b); b.dump();
            #3ns;
         end
      end
      join_any
   end
endprogram
```

Les classes en SystemVerilog

Comment faire évoluer notre TB

```
class Driver;
   virtual foo.TB I;
   mailbox req_mbx;
   BaseRequest req;
   // références vers l'interface et la mailbox
   function new( virtual foo.TB I, mailbox m );
      this.I = I;
      this.req_mbx = m;
   endfunction

   // manipulation de l'interface
   task reset ...  endtask
   task unreset ...  endtask
   task buswrite ...  endtask

   task run();
      logic[7:0] sdata = 'x;

      reset();
      unreset();
      forever
      begin
         req_mbx.get(req);
         buswrite(req, sdata);
         $display("w %02h & r %02h",
                  req.data, sdata);
      end
   endtask
endclass
```

```
class SequenceGenerator;
   mailbox req_mbx;
   Request req;

   // référence vers une mailbox
   function new( mailbox req_mbx );
      this.req_mbx = req_mbx;
   endfunction

   task run();
      int res;
      delay_t c_delay;

      c_delay = c_delay.first;
      do begin
         repeat(10)
         begin
            req = new;
            res = req.randomize()
                  with {dtype == c_delay;};
            if ( res == 1 )
               req_mbx.put(req);
            else
               $fatal(1,"Rondomistion failed");
         end
         c_delay = c_delay.next;
      end
      while(c_delay != c_delay.first);
   endtask
endclass
```

Les classes en SystemVerilog

Comment faire évoluer notre TB

Le testeur devient :

```
program tester( foo.TB I);

   mailbox req_mbx;
   Driver driver;
   SequenceGenerator gene;

   initial
      I.init();

   initial
   begin:main
      req_mbx  = new;
      driver   = new (I, req_mbx);
      gene     = new (req_mbx);

      fork
         driver.run();
         gene.run();
      join_any

   end:main
endprogram
```

Plan

Les assertions

Les assertions sont des constructions qui permettent :

- de vérifier (ou de prouver) des propriétés (au sens logique)

En SystemVerilog, ces propriétés peuvent être

- statiques ou
- des séquences temporelles.

Elles permettent de définir un objectif de couverture des simulations effectuées.

Langage dédié

PSL : Property Specification Language

PSL : existe et est standard depuis plus longtemps.
Compatible avec d'autres langages RTL (VHDL, Verilog)
En fonction des outils :

- fichiers indépendants
- commentaires magiques

Les **SVA** sont intégrées aux langage SystemVerilog

Langage dédié

PSL : Property Specification Language

PSL : existe et est standard depuis plus longtemps.
Compatible avec d'autres langages RTL (VHDL, Verilog)
En fonction des outils :

- fichiers indépendants
- commentaires magiques

Les **SVA** sont intégrées aux langage SystemVerilog

Les assertions

Les assertions procédurales

Dans un processus, elles sont évaluées au moment de son exécution.

```
initial/always
begin
...
  assert(xx == aa) else $info("au fait");
...
  assert(xx == yy) else $error("pas bien!");
...
  assert(xx == zz) else $fatal(1,"vraiment pas bien!");
...
```

Sans **`else`** génère une erreur avec un message générique.

Normalement, ignorées par les outils de synthèse.

Permettent de vérifier en permanence des règles dans un **`module`**, un **`program`** ou une **`interface`**.

- On définit des propriétés (**`property`**)
- Les propriétés sont forcément liées à un évènement déclencheur.
  - Les simulateur imposent qu'il soit lié à une horloge
- On définit ensuite une assertion (**`assert`**) sur cette propriété.

Les assertions

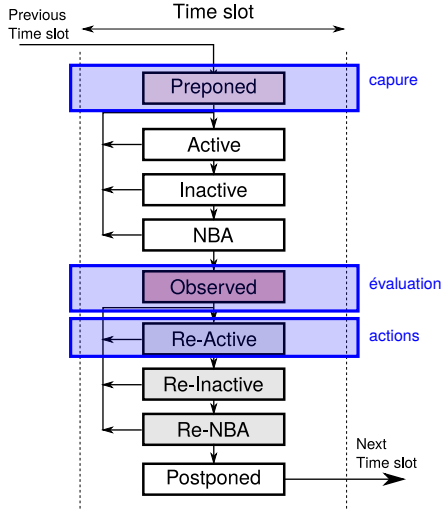
Les assertions concurrentes

Exemple

```
// au front d'horloge
// a ou b doit être vrai
property P0;
   @(posedge clk)
   a || b;
endproperty

assrt_p0: assert property(P0) else $info("Est-ce normal?");
```

Les assertions

Les assertions concurrentes



Implication (|->)

```
// au front d'horloge si a est vrai
// alor b doit être faux
property P1;
  @(posedge clk)
      a |-> !b;
endproperty

assrt_p1: assert property(P1) else $error("pas bien");
```

Dans la norme, «**Overlapped implication**»

L'évaluation de **b** se fait sur l'évènement pour lequel **a** est vrai.

Implication (|=>)

```
// au front d'horloge si a
// alors au prochain front
// c doit être faux
property P2;
   @(posedge clk)
   a |=> !c;
endproperty

assrt_p2: assert property(P2) else $error("vraiment pas bien");
```

Dans la norme, «**Nonverlapped implication**»

L'évaluation de **c** se fait l'évènement suivant l'évènement pour lequel **a** est vrai.

Capture

```
// au front d'horloge si req et !ack
// on capture la donnée
// on vérifie quelle reste stable au coup suivant
property P3;
 bit [7:0] s;
   @(posedge clk)
   (req && !ack , s = bus) |=> s == bus;
endproperty

assrt_p3: assert property(P3) else $error("Ça a changé!");
```

Dans l'exemple, la variable locale **s** permet de capturer la valeur de **bus** si la condition **(req && !ack)** est vérifiée. Au cycle suivant on vérifie que **bus** n'a pas changé de valeur.

À chaque fois que la propriété est déclenché, une nouvelle capture est faite.

Les séquences

```
// au front d'horloge si stb
// alors ack doit être vrai au cycle suivant
// ou dans les 5 cycles
property P4;
  @(posedge clk)
      stb |-> ##[1:5] ack;
endproperty

assrt_p4: assert property(P4) else $error("Trop tard");
```

Si **stb** alors **ack** doit arriver au cycle suivant ou dans les 5 cycles.

Attention ce code n'est pas efficace.

Les séquences

```
property P5;
  @(posedge clk)
     (a ##1 b) |-> (c ##2 d);
endproperty

assrt_p5: assert property(P5) else $error("Trop tard");
```

a suivi de **b** au cycle suivant, implique, **c** suivi de **d** deux cycles plus tard.
Détecter le changement d'état d'un signal peut alors être écrit (**`!stb ##1 stb`**)

On peut avoir des séquences infinies. Par exemple, **`##[1:$]x`** veut dire **x** vrai à partir du cycle suivant.

Des raccourcis

$rose	le signal est passé de `0` à `1`
$fell	le signal est passé de `1` à `0`
$stable	la valeur du signal n'a pas changée
$changed	la valeur du signal a changée
…	

Les assertions

Comment faire évoluer notre TB

Ajouter des assertions :

- − dans le testeur (ce n'est pas son rôle)
- + dans l'interface,
- + dans un module observateur qui espionne l'interface (c'est souvent ce qui est fait)

Les assertions

Comment faire évoluer notre TB

```
module testbench();
  bit   clk;

  foo I(.*);

  tester tester_i(.*);
  slave  DUT(.*);
  monitor monitor_i  (.*);

  always #10ns clk = !clk;

endmodule
```

Les assertions

Comment faire évoluer notre TB

```
module monitor( foo.MONITOR I );

   property slave_data_notunknown_when_ready;
      @(posedge I.clk)
      I.ready |-> $isunknown(I.s) == 0;
   endproperty

   assert_slave_data_notunknown_when_ready: assert property (slave_data_notunknown_when_ready)
   else $error("%m: ready is asserted but data from slave is non valid");

   property slave_ready_until_valid;
      @(posedge I.clk)
      $rose(I.ready) |-> I.ready throughout I.valid [->1]; //ou I.ready [*0:$] ##1 I.valid;
   endproperty

   assert_slave_ready_until_valid: assert property(slave_ready_until_valid)
   else $error("%m:slave's ready must be held until valid is set");

   property slave_data_held_when_ready;
      bit [7:0] s;
      @(posedge I.clk) disable iff (I.nrst == 0)
      (I.ready && !I.valid , s = I.s) |=> s == I.s; //ou $stable(I.s);
   endproperty

   assert_slave_data_held_when_ready: assert property(slave_data_held_when_ready)
   else $error("%m: data must be held stable when slave is ready");

endmodule
```

Plan

coverpoint/covergroup

covergroup : groupe les éléments pour lesquels on veut une analyse.
On precise quel évènement déclenche l'enregistrement.

coverpoint : l'élément qu'on veut analyser.

coverpoint/covergroup

Exemple

```
module ALUCovMonitor(
  input clk,
  input [4:0] opcode,
  input bypass
);

  covergroup ALUCov @(posedge clk);
    coverpoint opcode;
    coverpoint bypass;
  endgroup

  ...
  ALUCov cov_i = new;
  ...

endmodule
```

- l'enregistrement est déclenché à chaque front de **clk**
- on obtient l'histogramme de **opcode** et **bypass**
- peut être déclaré dans un module, une classe ou une interface.
- on doit l'instancier (et dans une classe le déclencher)

coverpoint/covergroup

Exemple 2

```
module ALUCovMonitor(
  input clk,
  input [4:0] opcode,
  input bypass
);

  covergroup ALUCov @(posedge clk);
    coverpoint opcode {
      bins group0          = {5'h0};
      bins group1          = {[5'h1:5'h1e]};
      ignore_bins notused = {5'h1f};
    }
    coverpoint bypass {
      bins active   = {0};
      bins inactive = {1};
    }
  endgroup

  ...
  ALUCov cov_i = new;
  ...

endmodule
```

- préciser les catégories que l'on veut tester
- ignorer/interdire certaines catégories
- plein d'autres choses…

coverpoint/covergroup

Exemple 3

```
module tb();
bit clk;
logic [4:0] opcode;
logic bypass;

ALUCovMonitor2 mon(.*);

always #10ns clk = !clk;

initial
begin
  repeat(100)
  begin
    @(negedge clk);
    opcode = $random();
    bypass = $random();
  end

  $display("Coverage %P", mon.cov_i);
  $display("Coverage %d%%",
    mon.cov_i.get_inst_coverage());
  $finish();
end
endmodule
```

- On peut demander aux outils le taux de couverture
- il peut aussi être obtenu durant la simulation

coverpoint/covergroup

Exemple dans une interface

```
interface foo (input bit clk);

  logic nrst;
  logic [7:0] m;
  logic [7:0] s;
  logic  valid;
  logic  ready;

  ...

  covergroup foo_itf_cov @(tb_cb iff nrst);
    master_data: coverpoint m iff (valid && ready){
      bins ZERO  = {8'h00};
      bins VAL[4] = {[8'h01:8'hfe]};
      bins FFFF  = {8'hff};
    }
    slave_data: coverpoint s iff (valid && ready);
  endgroup

  foo_itf_cov foo_cov = new;

endinterface:foo
```

Travail à faire

Reprendre les slides et complétez le testbench pour que tout fonctionne.