eBook Gratuit

APPRENEZ verilog

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#verilog

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: verilog

It is an unofficial and free verilog ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official verilog.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Verilog

Remarques

Verilog est un langage de description de matériel (HDL) utilisé pour concevoir, simuler et vérifier des circuits numériques à un niveau comportemental ou de transfert de registre. Il est à noter pour une raison qui le distingue des langages de programmation "traditionnels":

- Il existe deux types d'affectation, le blocage et le non-blocage, chacun avec ses propres utilisations et sémantique.
- Les variables doivent être déclarées comme étant à un seul bit ou avec une largeur explicite.
- Les conceptions sont hiérarchiques, avec la possibilité d'instancier des modules ayant un comportement souhaité.
- Dans la simulation (pas généralement en synthèse), les variables de `wire` peuvent être dans l'un des quatre états suivants: 0, 1, flottant (`z`) et indéfini (`x`).

Versions

Version	Date de sortie
Verilog IEEE-1364-1995	1995-01-01
Verilog IEEE-1364-2001	2001-09-28
Verilog IEEE-1364.1-2002	2002-12-18
Verilog IEEE-1364-2005	2006-04-07
SystemVerilog IEEE-1800-2009	2009-12-11
SystemVerilog IEEE-1800-2012	2013-02-21

Examples

Installation ou configuration

Les instructions détaillées sur la configuration ou l'installation de Verilog dépendent de l'outil que vous utilisez, car il existe de nombreux outils Verilog.

introduction

Verilog est un langage de description de matériel (HDL) utilisé pour modéliser des systèmes électroniques. Il décrit le plus souvent un système électronique au niveau du transfert de registre (RTL) de l'abstraction. Il est également utilisé dans la vérification des circuits analogiques et des

circuits à signaux mixtes. Sa structure et ses principes fondamentaux (décrits ci-dessous) sont conçus pour décrire et mettre en œuvre avec succès un système électronique.

- Rigidité
  Un circuit électronique est une entité physique ayant une structure fixe et Verilog est adapté pour cela. Les modules (module), les ports (input / output / inout), les connexions (fils), les blocs (@always), les registres (reg) sont tous fixés au moment de la compilation. Le nombre d'entités et d'interconnexions ne change pas de manière dynamique. Il y a toujours un "module" au niveau supérieur représentant la structure de la puce (pour la synthèse) et un au niveau du système pour la vérification.
- Parallélisme
  Les opérations simultanées inhérentes à la puce physique sont imitées dans le langage par des blocs toujours (la plupart du temps), initiaux et fork / join.

```
 module top();
 reg r1,r2,r3,r4; // 1-bit registers
   initial
   begin
     r1 <= 0 ;
   end
   initial
   begin
     fork
        r2 <= 0 ;
        r3 <= 0 ;
     join
   end
    always @(r4)
       r4 <= 0 ;
endmodule
```

Toutes les déclarations ci-dessus sont exécutées en parallèle dans la même unité de temps.

- Chronométrage et synchronisation
  Verilog prend en charge diverses constructions pour décrire la nature temporelle des circuits. Les temporisations et les délais dans les circuits peuvent être implémentés dans Verilog, par exemple par #delay constructs. De même, Verilog prend également en charge des circuits et des composants synchrones et asynchrones tels que les flops, les verrous et la logique combinatoire en utilisant différentes constructions, par exemple des blocs "toujours". Un ensemble de blocs peut également être synchronisé via un signal d'horloge commun ou un bloc peut être déclenché en fonction d'un ensemble spécifique d'entrées.

  ```
  #10   ;                 // delay for 10 time units
  always @(posedge clk )  // synchronous
  always @(sig1 or sig2 ) // combinatorial logic
  @(posedge event1)       // wait for post edge transition of event1
  wait (signal  == 1)     // wait for signal to be 1
  ```

- Incertitude
  Verilog soutient certaines des incertitudes inhérentes aux circuits électroniques. "X" représente l'état inconnu du circuit. "Z" est utilisé pour représenter l'état non contrôlé du

circuit.

```
reg1 = 1'bx;
reg2 = 1'bz;
```

- Abstraction
  Verilog prend en charge la conception à différents niveaux d'abstraction. Le niveau d'abstraction le plus élevé pour une conception est le niveau de transfert de résistance (RTL), le niveau suivant étant le niveau de la porte et le plus bas le niveau de la cellule (User Define Primitives). Verilog prend également en charge le niveau comportemental d'abstraction sans tenir compte de la réalisation structurelle de la conception, principalement utilisée pour la vérification.

```
 // Example of a D flip flop at RTL abstraction
module dff (
 clk    , // Clock Input
 reset  , // Reset input
 d       , // Data Input
 q        // Q output
 );
 //-----------Input Ports---------------
 input d, clk, reset ;

 //-----------Output Ports---------------
 output q;

 reg q;

 always @ ( posedge clk)
 if (~reset) begin
   q <= 1'b0;
 end  else begin
   q <= d;
 end

endmodule


// And gate model based at Gate level abstraction
module and(input x,input y,output o);

wire  w;
// Two instantiations of the module NAND
nand U1(w,x, y);
nand U2(o, w, w);

endmodule

// Gate modeled at Cell-level Abstraction
primitive udp_and(
a, // declare three ports
b,
c
);
output a;   // Outputs
input b,c;  // Inputs
```

```
// UDP function code here
// A = B & C;
table
 // B  C    : A
    1  1    : 1;
    0  1    : 0;
    1  0    : 0;
    0  0    : 0;
endtable

endprimitive
```

Il existe trois principaux cas d'utilisation pour Verilog. Ils déterminent la structure du code et son interprétation et déterminent également les ensembles d'outils utilisés. Les trois applications sont nécessaires à la mise en œuvre réussie de toute conception Verilog.

1. Conception physique / back-end
   Ici, Verilog est principalement utilisé pour voir la conception comme une matrice de portes d'interconnexion mettant en œuvre une conception logique. RTL / logic / Design passe de différentes étapes, de la synthèse -> placement -> à la construction de l'horloge -> au routage -> DRC -> LVS -> à la sortie de bande. Les étapes et les séquences précises varient en fonction de la nature exacte de la mise en œuvre.
2. Simulation
   Dans ce cas d'utilisation, l'objectif principal est de générer des vecteurs de test pour valider la conception conformément à la spécification. Le code écrit dans ce cas d'utilisation n'a pas besoin d'être synthétisable et reste dans la sphère de vérification. Le code ici ressemble davantage à des structures logicielles génériques comme les boucles for / while / do, etc.
3. Conception
   La conception implique la mise en œuvre de la spécification d'un circuit généralement au niveau d'abstraction RTL. Le code Verilog est alors donné pour la vérification et le code entièrement vérifié est donné pour l'implémentation physique. Le code est écrit en utilisant uniquement les constructions synthétisables de Verilog. Certains styles de codage RTL peuvent provoquer une non-concordance entre la simulation et la synthèse, et il faut veiller à les éviter.

Il existe deux principaux flux de mise en œuvre. Ils affecteront également la manière dont le code Verilog est écrit et implémenté. Certains styles de codage et certaines structures conviennent mieux dans un flux que dans l'autre.

- ASIC Flow (circuit intégré spécifique à l'application)
- Flux FPGA (matrice de portes programmable sur site) - inclut les FPGA et les CPLD

Bonjour le monde

Cet exemple utilise le compilateur icarus verilog.

Étape 1: Créez un fichier appelé hello.v

```
module myModule();

initial
  begin
    $display("Hello World!");   // This will display a message
    $finish ; // This causes the simulation to end.  Without, it would go on..and on.
  end

endmodule
```

Étape 2. Nous compilons le fichier .v en utilisant icarus:

```
>iverilog -o hello.vvp hello.v
```

L'option -o attribue un nom au fichier d'objet de sortie. Sans ce commutateur, le fichier de sortie serait appelé a.out. Hello.v indique le fichier source à compiler. Il ne devrait y avoir pratiquement aucune sortie lorsque vous compilez ce code source, sauf s'il y a des erreurs.

Étape 3. Vous êtes prêt à simuler ce programme Hello World verilog. Pour ce faire, invoquer en tant que tel:

```
>vvp hello.vvp
Hello World!
>
```

Installation du compilateur Icarus Verilog pour Mac OSX Sierra

1. Installez Xcode depuis l'App Store.
2. Installer les outils de développement Xcode

```
> xcode-select --install
```

Cela fournira des outils de ligne de commande de base tels que `gcc` et `make`

3. Installer les ports Mac `https://www.macports.org/install.php`

Le package d'installation OSX Sierra fournira une méthode open-source d'installation et de mise à niveau de logiciels supplémentaires sur la plate-forme Mac. Pensez `yum` ou `apt-get` pour le Mac.

4. Installer icarus en utilisant les ports Mac

```
> sudo port install iverilog
```

5. Vérifier l'installation à partir de la ligne de commande

```
$ iverilog
iverilog: no source files.

Usage: iverilog [-ESvV] [-B base] [-c cmdfile|-f cmdfile]
                [-g1995|-g2001|-g2005] [-g<feature>]
                [-D macro[=defn]] [-I includedir] [-M depfile] [-m module]
```

```
                [-N file] [-o filename] [-p flag=value]
                [-s topmodule] [-t target] [-T min|typ|max]
                [-W class] [-y dir] [-Y suf] source_file(s)

See the man page for details.
$
```

Vous êtes maintenant prêt à compiler et à simuler votre premier fichier Verilog sur le Mac.

Installer GTKWave pour l'affichage graphique des données de simulation sur Mac OSx Sierra

GTKWave est un logiciel de visualisation graphique complet prenant en charge plusieurs normes de stockage de données graphiques, mais il prend également en charge le format VCD, qui est le format que `vvp` affichera. Donc, pour ramasser GTKWave, vous avez quelques options

1. Allez sur http://gtkwave.sourceforge.net/gtkwave.zip et téléchargez-le. Cette version est généralement la plus récente.
2. Si vous avez installé MacPorts (https://www.macports.org/), lancez simplement `sudo port install gtkwave` . Cela voudra probablement installer sur les dépendances à. Notez que cette méthode vous donnera généralement une version plus ancienne. Si vous ne disposez pas de MacPorts, il existe un exemple de configuration de l'installation pour ce faire sur cette page. Oui! Vous aurez besoin de tous les outils de développement xcode, car ces méthodes vous "construiront" une source GTKWave.

Une fois l'installation terminée, vous pouvez être invité à sélectionner une version python. J'avais déjà installé 2.7.10 et je n'ai donc jamais "sélectionné" un nouveau.

À ce stade, vous pouvez démarrer gtkwave depuis la ligne de commande avec `gtkwave` . Au démarrage, vous pouvez être invité à installer ou mettre à jour XQuarts. Faites-le Dans mon cas, XQuarts 2.7.11 est installé.

Note: J'ai eu besoin de redémarrer pour obtenir XQuarts correctement, puis j'ai tapé à nouveau `gtkwave` et l'application apparaît.

Dans l'exemple suivant, je créerai deux fichiers indépendants, un banc d'essai et un module à tester, et nous utiliserons gtkwave pour visualiser la conception.

Utiliser Icarus Verilog et GTKWaves pour simuler et visualiser graphiquement une conception

Cet exemple utilise Icarus et GTKWave. Les instructions d'installation de ces outils sur OSx sont fournies ailleurs sur cette page.

Commençons par la conception du module. Ce module est un affichage BCD vers 7 segments. J'ai codé le design de manière obtuse simplement pour nous donner quelque chose qui se casse facilement et nous pouvons passer quelque temps à réparer graphiquement. Nous avons donc une horloge, une réinitialisation, une entrée de 4 données représentant une valeur de BCD et une sortie de 7 bits représentant l'affichage à sept segments. Créez un fichier nommé bcd_to_7seg.v

et placez-y la source ci-dessous.

```
module bcd_to_7seg (
   input clk,
   input reset,
   input [3:0] bcd,
   output [6:0] seven_seg_display

);
   parameter TP = 1;
   reg seg_a;
   reg seg_b;
   reg seg_c;
   reg seg_d;
   reg seg_e;
   reg seg_f;
   reg seg_g;


   always @ (posedge clk or posedge reset)
      begin
      if (reset)
         begin
            seg_a <= #TP 1'b0;
            seg_b <= #TP 1'b0;
            seg_c <= #TP 1'b0;
            seg_d <= #TP 1'b0;
            seg_e <= #TP 1'b0;
            seg_f <= #TP 1'b0;
            seg_g <= #TP 1'b0;
         end
      else
         begin
            seg_a <= #TP  ~(bcd == 4'h1 || bcd == 4'h4);
            seg_b <= #TP  bcd < 4'h5 || bcd > 6;
            seg_c <= #TP   bcd != 2;
            seg_d <= #TP   bcd == 0 || bcd[3:1] == 3'b001 || bcd == 5 || bcd == 6 || bcd == 8;
            seg_e <= #TP  bcd == 0 || bcd == 2 || bcd == 6 || bcd == 8;
            seg_f <= #TP  bcd == 0 || bcd == 4 || bcd == 5 || bcd == 6 || bcd > 7;
            seg_g <= #TP  (bcd > 1 && bcd < 7) || (bcd > 7);
         end
    end

    assign seven_seg_display = {seg_g,seg_f,seg_e,seg_d,seg_c,seg_b,seg_a};
endmodule
```

Ensuite, nous avons besoin d'un test pour vérifier si ce module fonctionne correctement. La déclaration de cas dans le banc d’essai est en fait plus facile à lire à mon avis et plus claire sur ce qu’elle fait. Mais je n'ai pas voulu mettre la même déclaration de cas dans la conception ET dans le test. C'est une mauvaise pratique. Deux modèles indépendants sont plutôt utilisés pour valider les uns les autres.

Avec le code ci-dessous, vous remarquerez deux lignes `$dumpfile("testbench.vcd");` et `$dumpvars(0,testbench);` . Ces lignes sont ce qui crée le fichier de sortie VCD qui sera utilisé pour effectuer une analyse graphique de la conception. Si vous les omettez, vous ne recevrez pas de fichier VCD. Créez un fichier appelé testbench.v et placez-y la source ci-dessous.

```
`timescale 1ns/100ps
module testbench;
reg clk;
reg reset;
reg [31:0] ii;
reg [31:0] error_count;
reg [3:0] bcd;
wire [6:0] seven_seg_display;
parameter TP = 1;
parameter CLK_HALF_PERIOD = 5;

   // assign clk = #CLK_HALF_PERIOD ~clk;  // Create a clock with a period of ten ns
   initial
   begin
     clk = 0;
     #5;
     forever clk = #( CLK_HALF_PERIOD )  ~clk;
   end

   initial
     begin
       $dumpfile("testbench.vcd");
       $dumpvars(0,testbench);
       // clk  = #CLK_HALF_PERIOD ~clk;
       $display("%0t, Reseting system", $time);
       error_count = 0;
       bcd  = 4'h0;
       reset = #TP 1'b1;
       repeat (30) @ (posedge clk);
       reset  = #TP 1'b0;
       repeat (30) @ (posedge clk);
       $display("%0t, Begin BCD test", $time); // This displays a message


       for (ii = 0; ii < 10; ii = ii + 1)
          begin
          repeat (1) @ (posedge clk);
          bcd  = ii[3:0];
          repeat (1) @ (posedge clk);
          if (seven_seg_display !== seven_seg_prediction(bcd))
             begin
                $display("%0t, ERROR: For BCD %d, module output 0b%07b does not match
prediction logic value of 0b%07b.",$time,bcd, seven_seg_display,seven_seg_prediction(bcd));
                error_count = error_count + 1;
             end
          end
       $display("%0t, Test Complete with %d errors", $time, error_count);
       $display("%0t, Test %s", $time, ~|error_count ? "pass." : "fail.");
       $finish ; // This causes the simulation to end.
     end


parameter SEG_A = 7'b0000001;
parameter SEG_B = 7'b0000010;
parameter SEG_C = 7'b0000100;
parameter SEG_D = 7'b0001000;
parameter SEG_E = 7'b0010000;
parameter SEG_F = 7'b0100000;
parameter SEG_G = 7'b1000000;

function [6:0] seven_seg_prediction;
```

```
   input [3:0] bcd_in;

   //    +--- A ---+
   //    |         |
   //    F         B
   //    |         |
   //    +--- G ---+
   //    |         |
   //    E         C
   //    |         |
   //    +--- D ---+

   begin
      case (bcd_in)
         4'h0: seven_seg_prediction = SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F;
         4'h1: seven_seg_prediction = SEG_B | SEG_C;
         4'h2: seven_seg_prediction = SEG_A | SEG_B | SEG_G | SEG_E | SEG_D;
         4'h3: seven_seg_prediction = SEG_A | SEG_B | SEG_G | SEG_C | SEG_D;
         4'h4: seven_seg_prediction = SEG_F | SEG_G | SEG_B | SEG_C;
         4'h5: seven_seg_prediction = SEG_A | SEG_F | SEG_G | SEG_C | SEG_D;
         4'h6: seven_seg_prediction = SEG_A | SEG_F | SEG_G | SEG_E | SEG_C | SEG_D;
         4'h7: seven_seg_prediction = SEG_A | SEG_B | SEG_C;
         4'h8: seven_seg_prediction = SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G;
         4'h9: seven_seg_prediction = SEG_A | SEG_F | SEG_G | SEG_B | SEG_C;
         default: seven_seg_prediction = 7'h0;
      endcase
   end
endfunction


bcd_to_7seg u0_bcd_to_7seg (
.clk               (clk),
.reset             (reset),
.bcd               (bcd),
.seven_seg_display (seven_seg_display)
);


endmodule
```

Maintenant que nous avons deux fichiers, un testbench.v et un bcd_to_7seg.v, nous devons compiler, élaborer en utilisant Icarus. Pour faire ça:

```
$ iverilog -o testbench.vvp testbench.v bcd_to_7seg.v
```

Ensuite, nous devons simuler

```
$ vvp testbench.vvp
LXT2 info: dumpfile testbench.vcd opened for output.
0, Reseting system
6000, Begin BCD test
8000, Test Complete with          0 errors
8000, Test pass.
```

À ce stade, si vous souhaitez valider le fichier en cours de test, allez dans le fichier bcd_2_7seg.v et déplacez une partie de la logique et répétez ces deux premières étapes.

Par exemple, je change la ligne `seg_c <= #TP bcd != 2; seg_c <= #TP bcd != 4;` . Recompiler et simuler effectue les opérations suivantes:

```
$ iverilog -o testbench.vvp testbench.v bcd_to_7seg.v
$ vvp testbench.vvp
LXT2 info: dumpfile testbench.vcd opened for output.
0, Reseting system
6000, Begin BCD test
6600, ERROR: For BCD  2, module output 0b1011111 does not match prediction logic value of
0b1011011.
7000, ERROR: For BCD  4, module output 0b1100010 does not match prediction logic value of
0b1100110.
8000, Test Complete with          2 errors
8000, Test fail.
$
```

Alors maintenant, permet de visualiser la simulation en utilisant GTKWave. Depuis la ligne de commande, émettez un

```
gtkwave testbench.vcd &
```

Lorsque la fenêtre GTKWave apparaît, vous verrez dans le coin supérieur gauche le nom du module testbench. Cliquez dessus. Cela révélera les sous-modules, tâches et fonctions associés à ce fichier. Les fils et les registres apparaîtront également dans la zone inférieure gauche.

Faites maintenant glisser, clk, bcd, error_count et seven_seg_display dans la zone de signal située à côté de la fenêtre de forme d'onde. Les signaux vont maintenant être tracés. Error_count vous montrera quelle entrée BCD particulière a généré la mauvaise sortie seven_seg_display.

Vous êtes maintenant prêt à dépanner graphiquement un bug de Verilog.

Lire Démarrer avec Verilog en ligne: https://riptutorial.com/fr/verilog/topic/1080/demarrer-avec-verilog

Chapitre 2: Blocs procéduraux

Syntaxe

- toujours @ (posedge clk) begin / * instructions * / end
- always @ (negedge clk) begin / * instructions * / end
- always @ (posedge clk ou posedge reset) // peut synthétiser moins efficacement que la réinitialisation synchrone

Examples

Compteur simple

Un compteur utilisant une initialisation à bascule de type FPGA:

```
module counter(
    input clk,
    output reg[7:0] count
)
initial count = 0;
always @ (posedge clk) begin
    count <= count + 1'b1;
end
```

Un compteur implémenté en utilisant des réinitialisations asynchrones adaptées à la synthèse ASIC:

```
module counter(
  input            clk,
  input            rst_n, // Active-low reset
  output reg [7:0] count
)
always @ (posedge clk or negedge rst_n) begin
  if (~rst_n) begin
    count <= 'b0;
  end
  else begin
    count <= count + 1'b1;
  end
end
```

Les blocs procéduraux dans ces exemples incrémentent le `count` à chaque front montant.

Affectations non bloquantes

Une affectation non bloquante (`<=`) est utilisée pour l'affectation dans les blocs `always` sensibles aux contours. Dans un bloc, les nouvelles valeurs ne sont pas visibles tant que le bloc entier n'a pas été traité. Par exemple:

```
module flip(
    input clk,
    input reset
)
reg f1;
reg f2;

always @ (posedge clk) begin
  if (reset) begin // synchronous reset
    f1 <= 0;
    f2 <= 1;
  end
  else begin
    f1 <= f2;
    f2 <= f1;
  end
end
endmodule
```

Notez l'utilisation des assignations non bloquantes (`<=`) ici. Étant donné que la première affectation n'entre en vigueur qu'après le bloc procédural, la deuxième attribution fait ce qui est prévu et échange les deux variables - contrairement à une affectation de blocage (`=`) ou à des affectations dans d'autres langues; `f1` toujours sa valeur d'origine sur le côté droit de la deuxième affectation dans le bloc.

Lire Blocs procéduraux en ligne: https://riptutorial.com/fr/verilog/topic/2512/blocs-proceduraux

Chapitre 3: Bonjour le monde

Examples

Compiler et exécuter l'exemple

En supposant un fichier source de `hello_world.v` et un module de premier niveau de `hello_world` . Le code peut être exécuté en utilisant différents simulateurs. La plupart des simulateurs sont des simulateurs compilés. Ils nécessitent plusieurs étapes pour compiler et exécuter. Généralement le

- La première étape consiste à compiler le design de Verilog.
- La deuxième étape consiste à élaborer et à optimiser la conception.
- La troisième étape consiste à exécuter la simulation.

Les détails des étapes peuvent varier en fonction du simulateur, mais l'idée générale reste la même.

Processus en trois étapes à l'aide de Cadence Simulator

```
ncvlog hello_world.v
ncelab hello_world
ncsim hello_world
```

- La première étape ncvlog consiste à compiler le fichier hello_world.v
- La deuxième étape ncelab consiste à élaborer le code avec le module de niveau supérieur hello_world.
- La troisième étape ncsim consiste à exécuter la simulation avec le module de niveau supérieur hello_world.
- Le simulateur génère tout le code compilé et optimisé dans une librairie de travail. [INCA_libs - nom de la bibliothèque par défaut]

pas à pas en utilisant Cadence Simulator.

La ligne de commande appelle en interne les trois étapes requises. Cela imite l'ancien style d'exécution du simulateur interprété (ligne de commande unique).

```
irun hello_world.v
or
ncverilog hello_world.v
```

Bonjour le monde

Les sorties du programme Hello World! à la sortie standard.

```
module HELLO_WORLD(); // module doesn't have input or outputs
  initial begin
    $display("Hello World");
```

```
    $finish; // stop the simulator
  end
endmodule
```

Le module est un élément de base de Verilog. Il représente une collection d'éléments et est entouré par le mot-clé module et module de fin. Ici, hello_world est le module le plus (et le seul) le plus important.

Le bloc initial s'exécute au début de la simulation. Le début et la fin sont utilisés pour marquer la limite du bloc initial. `$display` le message sur la sortie standard. Il insère et fin de ligne "\ n" au message.

Ce code ne peut pas être synthétisé, c'est-à-dire qu'il ne peut pas être mis dans une puce.

Lire Bonjour le monde en ligne: https://riptutorial.com/fr/verilog/topic/1819/bonjour-le-monde

Chapitre 4: Incohérence entre la synthèse et la simulation

Introduction

Une bonne explication de ce sujet se trouve dans http://www.sunburst-design.com/papers/CummingsSNUG1999SJ_SynthMismatch.pdf

Examples

Comparaison

fil d = 1'bx; // dire du bloc précédent. Sera 1 ou 0 dans le matériel

if (d == 1'b) // faux dans la simulation. Peut être vrai de faux dans le matériel

Liste de sensibilité

```
wire a;
wire b;
reg q;

always @(a) // b missing from sensativity list
 q = a & b; // In simulation q will change only when a changes
```

Dans le matériel, q changera à chaque changement de a ou b.

Lire Incohérence entre la synthèse et la simulation en ligne: https://riptutorial.com/fr/verilog/topic/9220/incoherence-entre-la-synthese-et-la-simulation

Chapitre 5: Souvenirs

Remarques

Pour les FIFO, vous instanciez généralement un bloc spécifique au fournisseur (également appelé "core" ou "IP").

Examples

Simple Dual Port RAM

Simple port double RAM avec adresses et horloges séparées pour les opérations de lecture / écriture.

```
module simple_ram_dual_clock #(
  parameter DATA_WIDTH=8,                 //width of data bus
  parameter ADDR_WIDTH=8                  //width of addresses buses
)(
  input      [DATA_WIDTH-1:0] data,       //data to be written
  input      [ADDR_WIDTH-1:0] read_addr,  //address for read operation
  input      [ADDR_WIDTH-1:0] write_addr, //address for write operation
  input                       we,         //write enable signal
  input                       read_clk,   //clock signal for read operation
  input                       write_clk,  //clock signal for write operation
  output reg [DATA_WIDTH-1:0] q           //read data
);

  reg [DATA_WIDTH-1:0] ram [2**ADDR_WIDTH-1:0]; // ** is exponentiation

  always @(posedge write_clk) begin //WRITE
    if (we) begin
      ram[write_addr] <= data;
    end
  end

  always @(posedge read_clk) begin //READ
    q <= ram[read_addr];
  end

endmodule
```

RAM synchrone à port unique

Simple Single Port RAM avec une adresse pour les opérations de lecture / écriture.

```
module ram_single #(
  parameter DATA_WIDTH=8,          //width of data bus
  parameter ADDR_WIDTH=8           //width of addresses buses
)(
  input  [(DATA_WIDTH-1):0] data,  //data to be written
  input  [(ADDR_WIDTH-1):0] addr,  //address for write/read operation
  input                     we,    //write enable signal
```

```
  input                     clk,   //clock signal
  output [(DATA_WIDTH-1):0] q      //read data
);

  reg [DATA_WIDTH-1:0] ram [2**ADDR_WIDTH-1:0];
  reg [ADDR_WIDTH-1:0] addr_r;

  always @(posedge clk) begin //WRITE
      if (we) begin
          ram[addr] <= data;
      end
      addr_r <= addr;
  end

  assign q = ram[addr_r]; //READ

endmodule
```

Registre à décalage

Registre à décalage profond N bits avec réinitialisation asynchrone.

```
module shift_register #(
  parameter REG_DEPTH = 16
)(
  input clk,       //clock signal
  input ena,       //enable signal
  input rst,       //reset signal
  input data_in,   //input bit
  output data_out  //output bit
);

  reg [REG_DEPTH-1:0] data_reg;

  always @(posedge clk or posedge rst) begin
    if (rst) begin //asynchronous reset
      data_reg <= {REG_DEPTH{1'b0}}; //load REG_DEPTH zeros
    end else if (enable) begin
      data_reg <= {data_reg[REG_DEPTH-2:0], data_in}; //load input data as LSB and shift
(left) all other bits
    end
  end

  assign data_out = data_reg[REG_DEPTH-1]; //MSB is an output bit

endmodule
```

Port simple Async Lecture / écriture RAM

RAM simple port simple avec opérations de lecture / écriture asynchrones

```
module ram_single_port_ar_aw #(
  parameter DATA_WIDTH = 8,
  parameter ADDR_WITDH = 3
)(
  input                       we,    // write enable
  input                       oe,    // output enable
```

```
  input  [(ADDR_WITDH-1):0]   waddr, // write address
  input  [(DATA_WIDTH-1):0]   wdata, // write data
  input                       raddr, // read adddress
  output [(DATA_WIDTH-1):0]   rdata  // read data
);

  reg [(DATA_WIDTH-1):0]      ram [0:2**ADDR_WITDH-1];
  reg [(DATA_WIDTH-1):0]      data_out;

  assign rdata = (oe && !we) ? data_out : {DATA_WIDTH{1'bz}};

  always @*
  begin : mem_write
    if (we) begin
      ram[waddr] = wdata;
    end
  end

  always @* // if anything below changes (i.e. we, oe, raddr), execute this
  begin : mem_read
    if (!we && oe) begin
      data_out = ram[raddr];
    end
  end

endmodule
```

Lire Souvenirs en ligne: https://riptutorial.com/fr/verilog/topic/2654/souvenirs

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Verilog	Brian Carlton, Community, hexafraction, Morgan, Qiu, Rahul Menon, Rich Maes, wilcroft
2	Blocs procéduraux	Brian Carlton, hexafraction, Morgan, wilcroft
3	Bonjour le monde	Brian Carlton, Morgan, Rahul Menon
4	Incohérence entre la synthèse et la simulation	Brian Carlton
5	Souvenirs	Brian Carlton, jclin, Kamil Rymarz, Morgan, Qiu, wilcroft