FDS - Guide de Verilog

Ce document décrit les concepts de base pour l'utilisation de Verilog.

Contact pour tout signalement de typo ou erreur : simon.lefort@epfl.ch.

1. Qu’est-ce que Verilog ?

Verilog est un langage de description matériel utilisé pour la modélisation et la conception de circuits électroniques, tout comme VDHL (l’ancien langage utilisé en BA2 à l’EPFL, avec une courbe d’apprentissage un peu plus rude).

2. Comment installer Verilog ?

Pour exécuter votre code Verilog, vous aurez besoin d’un compilateur. Le compilateur recommandé pour le cours de FDS BA2 au printemps 2024 est **iverilog**, version 11 ou 12.

Pour l’installer :
- Sur MacOS, via Homebrew
- Sur Windows, non.
- Sur Linux, via make depuis le repository GitHub est probablement le plus safe

3. Comment exécuter du code Verilog ?

Un projet Verilog est composé de fichiers `.v`, contenant le code source. Dans un premier temps, vous devez compiler vos fichiers `.v` pour pouvoir les rendre exécutables.

Supposons que vous ayez écrit un fichier `my_super_test.v`. La commande suivante à taper dans le terminal :

```
iverilog -o build my_super_test.v
```

vous permettra d’obtenir un fichier exécutable `build` que vous pouvez exécuter en tapant `./build` dans le même terminal.

> Une fois le terminal ouvert, attention à bien vous déplacez dans le dossier qui contient vos fichiers `.v` (à l’aide de la commande `cd`) pour que la commande de build fonctionne.

Pour compiler plusieurs fichiers, comme dans le cas d’un test bench (voir Section 9.), vous pouvez simplement préciser plusieurs fichiers `.v`:

```
iverilog -o build my_super_test_testbench.v my_super_test.v
```

4. Comment écrire un programme Verilog ?

Il existe plusieurs façons d’écrire un programme Verilog :

- le **Verilog structurel** : nous devons décrire physiquement ce qu’il se passe (à l’aide de portes logiques comme `or`, `and`, etc.). C’est utile si on nous présente un circuit sans la fonction booléenne associée.
- le **Verilog d’affectation continue** : utilise le mot-clef `assign` pour définir des relations continues entre les signaux. C’est utile pour implémenter des fonctions logiques simples et pour connecter des modules ensemble.

- le **Verilog comportemental** : se concentre sur ce que le circuit doit faire, et pas sur la façon de l'implémenter (complètement l'inverse du verilog structurel). Il utilise des structures de contrôle de haut niveau (type `for`, `while`, etc.) et se situe typiquement dans un bloc `always` ou `initial`.

Nous verrons plus tard quelles sont, en pratique, les différences.

5. Déclaration de modules

La base d'un programme Verilog est un module : une portion de circuit, similaire à une fonction. Chaque déclaration de module commence par le mot-clef `module`, suivi de son nom.

5.1. Spécification des entrées

Les entrées sont utilisées pour se brancher sur un signal existant envoyé par un autre module (par exemple un autre module qui se charge de tester votre module).

Elles sont…

- déclarées à l'aide du mot-clef `input`.
- de type **wire**, câble
- d'une taille spécifique (ou 1 si non spécifiée)

Les tailles sont spécifiées entre crochet, **[a:b]**, où **a** est l'index du MSB, et **b** l'index du LSB.

5.1.1. Exemple

```
module my_super_module (
  input [4:0] A, B, // A and B will have the same size, 5 bit
  input D // D will have a size of 1 bit
);
// your code here
endmodule
```

5.2. Spécification des outputs

Les sorties sont utilisées pour envoyer un signal à d'autres modules.

Elles sont…

- déclarées à l'aide du mot-clef `output`.
- de type **wire** ou **reg**.
- d'une taille spécifique (ou 1 sinon)

5.2.1. Exemple

```
module my_super_module (
  input A,
  output B,
  output [2:0] G,
  output reg K
);
// your code here
endmodule
```

Nous verrons plus tard comment choisir entre reg et wire.

6. Le Verilog structurel

Comme évoqué précédemment, le Verilog structurel décrit physiquement comment est construit le circuit.

Ainsi, il faudra généralement déclarer :
- des câbles, déclaré en utilisant **wire** suivi du nom du câble.
- des portes logiques, déclarée à l'aide des mot-clefs **not**, **or**, **and**, **nor**, **xnor**, etc. suivis du nom de la porte, puis de l'output et des inputs entre parenthèses.

> En Verilog structurel, déclarez les entrées et les sorties en tant que **wires** (puisque nous connectons physiquement les signaux entre eux !)

6.1. Exemple

```
module structural_example (
    // 3 signaux d'entrée
    input a,
    input b,
    input c,
    // un signal de sortie
    output f
);

    // on créé nos câbles
    wire p1;
    wire p2;
    wire not_a;
    wire not_b;

    // ici on connecte l'entrée A au câble not_a en ajoutant au passage
    // une porte logique "NOT" pour inverser le signal
    not gA (not_a, a);
    not gB (not_b, b);

    // ici f (la sortie) est connectée aux deux câbles
    // p1 et p2, eux-mêmes connectés aux entrées ou à d'autres câbles
    nor g3 (f, p1, p2);
    and g2 (p1, not_a, b);
    and g1 (p2, not_b, c);

endmodule
```

Pour savoir comment tester ce circuit, allez voir la Section 9.

Tout fonctionne bien, mais… si on connaît l'expression booléenne derrière ce circuit, l'écrire en Verilog structurel est très long inutilement.

7. Le Verilog d'affectation continue

Comme mentionné précédemment, le Verilog d'affectation continue permet d'utiliser le mot-clef un peu magique `assign` qui permet d'utiliser des expressions booléennes directement dans le module.

> En Verilog d'affectation continue, déclarez également les entrées et les sorties en tant que **wires** (on continue de connecter physiquement des câbles).

7.1. Exemple

```
module structural_example (
    // 3 signaux d'entrée
    input a,
    input b,
    input c,
    // un signal de sortie
    output f
);

    assign p1 = !a && b;
    assign p2 = !b && c;
    assign f = !(p1 || p2);

endmodule
```

8. Verilog comportemental

Le Verilog comportemental permet de modéliser des circuits plus complexes que ceux vus jusqu'ici. Comme énoncé précédemment, on se focalise plus sur le comportement du circuit en lui-même que son implémentation hardware.

> En Verilog structurel, déclarez les inputs en tant que **wires** et les outputs en tant que **reg** ! En effet, on veut pouvoir leur assigner une valeur numérique sans les connecter physiquement à un câble.

8.1. Ecrire du code réactif

La façon dont fonctionne Verilog là-dessus parlera peut-être aux étudiants qui ont déjà fait du React, Vue, etc.

Ici, nous voulons que notre code soit **exécuté quand un signal d'entrée change** (par exemple lorsqu'une variable passe de 0 à 1).

Nous devons donc définir une liste de variables de dépendance (*comme `useEffect()` avec React !).*

Pour cela, on écrit un bloc `always @` et on écrit entre parenthèses le nom des variables de dépendance, à surveiller.

8.1.1. Exemple

```
module structural_example (
    // 3 signaux d'entrée
    input a,
    input b,
    input c,
    // un signal de sortie
    output reg f
);

// ces variables ne sont plus des câbles !
// elles ne connectent plus physiquement des câbles
// elles stockent uniquement des valeurs numériques
reg p1;
reg p2;

always @(a, b, c) begin
  p1 = !a && b;
  p2 = !b && c;
  // on assigne directement la valeur d'output à f
  f = !(p1 || p2);
end

endmodule
```

Notez qu'il est possible d'utiliser `always @*` pour surveiller toutes les variables utilisées dans le bloc always.

8.2. Ecrire du code exécuté à l'initialisation

Ici, nous voulons exécuter du code lorsque notre module s'initialise (au démarrage du programme). Ce sera particulièrement utile lorsque nous écrirons notre premier module de test.

Pour cela, nous devons écrire `initial` suivi de notre code.

8.2.1. Exemple

```
module structural_example;

initial begin
  $display("Hello, world.");
end

endmodule
```

> Comme vous pouvez le voir, nous utilisons une nouvelle fonction, **$display**, qui permet d'afficher un message textuel dans le terminal.

9. Tester ses circuits

Il nous faut maintenir écrire ce qu'on appelle un "test bench", TB, un environnement de simulation pour notre circuit.

Tout d'abord, il nous faut créer une instance du module à tester (qu'on appelle le DUT, le "Design Under Test"). La syntaxe pour initialiser le DUT est `nom_du_module nom_de_l_instance`, suivi d'un mapping des inputs et des outputs aux variables du TB.

Il y a deux autres parties importantes du TB : le stimulus generator (qui s'occupe de générer toutes les combinaisons d'entrée possibles) et le checker (qui vérifie que le résultat est bien celui attendu).

9.1. Exemple

Essayons de tester le code de la section Section 8.1.1.

> On écrit le test dans un bloc `initial` car on veut exécuter notre bloc de test une fois, quand il est lancé.

```
module tb_structural_example;

    // on définit les variables qui vont faire varier le DUT
    reg a = 0;
    reg b = 0;
    reg c = 0;
    // on définit le câble qui représente le signal de sortie
    // ce n'est pas une reg car on ne veut pas changer
    // la variable de F autrement qu'en
    // la connectant physiquement au module my_test
    wire F;

    // on initialise le D.U.T.
    structural_example my_super_structural_under_testing (
        // .variable_dans_dut(variable_dans_tb)
        .a(a),
        .b(b),
        .c(c),
        // .output_dans_dut(output_dans_tb)
        .f(f)
    );

    reg expected;

    initial begin
        for (integer i = 0; i < 8; i = i + 1) begin
            {a, b, c} = i;
            #1;
            // check if the output is correct
            expected = !(!a && b && !b && c);
            if (f != expected) begin
                $display("Error! Expected: [%b], Result: [%b]", expected, f);
            end else begin
                $display("Correct! Expected: [%b], Result: [%b]", expected, f);
            end
        end
    end
```

```
endmodule
```

Le résultat :

```
Correct input! Expected: [1], Result: [1]
Error, wrong input! Expected: [1], Result: [0]
Error, wrong input! Expected: [1], Result: [0]
Error, wrong input! Expected: [1], Result: [0]
Correct input! Expected: [1], Result: [1]
Error, wrong input! Expected: [1], Result: [0]
Correct input! Expected: [1], Result: [1]
Correct input! Expected: [1], Result: [1]
```

9.2. GTKWave

GTKWave est un logiciel permettant de visualiser graphiquement la sortie d'un module Verilog.

9.2.1. Installation

Pour installer GTKWave :
- sur MacOS, utilisez Homebrew
- sur Windows, non.
- sur Linux, build from source fonctionne comme d'habitude

9.2.2. Utilisation

Pour cela, commencez par écrire un test bench classique comme vu précédemment. Maintenant, il nous faut **exporter les résultats** du test bench pour pouvoir les lire avec GTKWave. Pour cela, nous utiliserons deux instructions :

- **`$dumpfile("nom_de_fichier.vcd");`** pour exporter les résultats dans un fichier VCD (Value Change Dump).
- **`$dumpvars`** pour exporter toutes les variables du programme dans le fichier (une nouvelle ligne sera écrite à chaque fois qu'une variable change)
- **`$monitor`** est très pratique, il permet d'afficher les changements des variables passées en argument dans la console (sans écrire de logs manuellement à chaque fois)
- **`$finish`** permet de finir proprement la simulation (on peut placer `$finish` à d'autres endroits dans le code pour interrompre la simulation)

```
module test_tb;

    reg a = 0;
    reg b = 0;
    reg c = 0;
    wire F;

    structural_example my_test (
        .a(a),
        .b(b),
        .c(c),
        .f(f)
    );

    reg expected;

    initial begin
        $dumpfile ("structural_example.vcd");
        $dumpvars;

        $monitor ("Time %2t, a=%b, b=%b, c=%b, f=%b", $time, a, b, c, f);

        for (integer i = 0; i < 8; i = i + 1) begin
            {a, b, c} = i;
            #1;
            expected = a & b | ~a & c;
            if (f != expected) begin
                $display("Error! Expected %b, got %b", expected, f);
            end else begin
                $display("Correct! Expected %b, got %b", expected, f);
            end
        end

        $finish;
    end


endmodule
```

Exécutez ensuite la commande suivante :

```
gtkwave structural_example.vcd
```

10. Compléments

10.1. Que se passe-t-il si on déclarer un input avec une taille “inversée” ?

`[0:2] au lieu de [2:0]`

```
module test (
    input [0:2] in, // si on envoie 011 au module...
    output [2:0] F
);

  always @* begin
      $display(in[0]); // ...on aura 0 ici au lieu de 1 !
      $display(in); // par contre ici on aura toujours 3 (et pas 6 !)
  end

endmodule;
```

10.2. Différence entre double et triple égalité

```
module test;

reg a = 1'bz;
reg b = 1'bz;

  initial begin

    (a == b) // false!
    (a === b) // true!
    // this is because by default you should not compare unknown values together but
during simulation it can be useful

  end

endmodule;
```

10.3. Différence entre double et triple égalité

```
module test;

reg a = 1'bz;
reg b = 1'bz;

  initial begin

    (a == b) // false!
    (a === b) // true!
    // this is because by default you should not compare unknown values together but
during simulation it can be useful

  end

endmodule;
```