

Additionneur en Verilog, style structurel

Michel Billaud (michel.billaud@laposte.net)

28 juin 2022

Table des matières

1 Objectifs	1
2 Demi-additionneur	2
2.1 Fonction de transfert	2
2.2 Description structurelle du demi-additionneur	2
2.3 Tester le module	3
2.4 Compilation et exécution	4
3 Additionneur complet	4
3.1 Fonction de transfert	5
3.2 Câblage du l'additionneur	5
3.3 Description structurelle en Verilog	5
3.4 Simulation	5
3.5 Compilation et exécution	7
4 Additionneur 4 bits	7
4.1 Description en Verilog	8
4.2 Vérification du fonctionnement	9
4.3 Exécution du test	10



Ce texte fait partie d'une petite collection de notes mise à disposition selon les termes de la Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 2.0 France.

- Les notes sont publiées dans <https://www.mbillaud.fr/notes/>
- Sources dans <https://github.com/MichelBillaud/notes-diverses>

1 Objectifs

Dans cette note, on regarde comment

- décrire un additionneur en Verilog en style “structurel” ;
- tester son fonctionnement en affichant tous les cas possibles.

Ceci en utilisant le compilateur `iverilog` que l'on trouve dans les packages Debian.

Pour compléter, on construit un additionneur 2 x 4 bits, avec une autre méthode de vérification. Avec 9 entrées (2 nombres de 4 bits et une retenue entrante),

- entrer les 512 cas à la main serait fastidieux (on va faire des boucles)
- afficher les 512 lignes ne permettrait pas de repérer les problèmes. on ne fera afficher que les lignes où il y a des anomalies.

2 Demi-additionneur

Le circuit minimal pour faire une addition

- prend en entrée deux nombres **a**, **b** de 1 bit chacun,
- fournit un nombre de deux bits : le chiffre de gauche **co** est la retenue sortante (output carry), le chiffre de droite **s** la somme.

Ce circuit est appelé **demi-additionneur**, on verra pourquoi plus loin.

2.1 Fonction de transfert

La fonction de transfert indique la valeur des sorties en fonction de celle des entrées. Sous forme de table :

a	b	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

2.2 Description structurelle du demi-additionneur

Ce circuit peut être construit par assemblage de **portes logiques** de base. Ici la sortie **r** s'obtient par un "et" des deux entrées, la sortie **s** par un ou-exclusif.

Cette description peut être codée en Verilog, en mettant le code suivant dans un fichier source que nous nommerons `half-adder.vl` :

```
// half-adder.vl
// description structurelle d'un demi-additionneur

module half_adder
(
    output co, s,
    input  a, b
);

    xor (s, a, b);
    and (co, a, b);
endmodule // half_adder
```

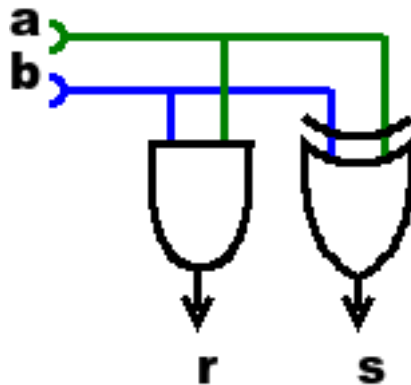


FIG. 1 : Circuit Demi-Additionneur

décrit un “module” qui

- a 4 ports de branchement : 2 entrées **a** et **b**, et 2 sorties **co** et **s**.
- **contient** une porte **xor** et une porte **and** qui ont **a** et **b** en entrée, et dont les sorties sont reliées respectivement à **s** et **co**.

C’est une description **structurelle** : le circuit est décrit comme une composition d’autres circuits. Verilog permet d’autres styles de description.

Notes

- les portes **and**, **or** sont prédéfinies en Verilog ;
- il en existe d’autres (**not**, **xor**, **nand**, **nor**) ;
- le premier paramètre est la sortie.

C’est pour des raisons d’homogénéité que nous avons choisi, pour **half_adder**, de mettre les sorties en premier.

2.3 Tester le module

Pour tester le module **half_adder**, nous allons écrire une **simulation**, qui

- envoie successivement toutes les combinaisons de 0 et de 1 en entrée d’un circuit demi-additionneur ;
- affiche les valeurs des entrées et sorties du circuit

```
# test-half-adder.vl
# test d'un circuit half_adder

`include "half-adder.vl"

module test_half_adder;
    reg a, b;
    wire co, s;
    half_adder h(co, s, a, b);
```

```

initial begin
    $monitor("%04t\t%b + %b = %b %b",
             $time, a, b, co, s);
    a = 0; b = 0;
    #10 a = 0; b = 1;
    #10 a = 1; b = 0;
    #10 a = 1; b = 1;
end
endmodule // test_half_adder

```

Explications : ce module de test décrit

- le matériel utilisé : un demi-additionneur (nommé **h**) connecté à deux entrées et deux sorties;
- dans un bloc “**initial**”, la séquence d’actions sur les entrées :
 - mettre **a** et **b** à 0;
 - attendre 10 unités de temps, puis mettre **a** à 0 et **b** à 1
 - attendre 10 unités de temps, puis mettre **a** à 1 et **b** à 0
 - etc. tout en affichant les changements de valeurs quand ils se produisent (**\$monitor**)

Les variables **a** et **b** ont des valeurs que nous pilotons explicitement à notre gré au moment voulu de la simulation. Ce sont des **registres** dans la terminologie Verilog.

Pour **co** et **s**, ce sont de simples fils (**wire**) dont l’état ne dépend que des sorties auxquelles ils sont reliés.

Complément : l’instruction **\$monitor** utilise une chaîne de format similaire à celle de **printf** en C. Ici on trouve

- “%04t” pour afficher le temps (**\$time**) sur 4 chiffres avec des 0 en tête;
- “\t” pour une tabulation,
- “%b” pour chacun des bits **a**, **b**, **co** et **s**.

2.4 Compilation et exécution

La suite de commandes

```

$ iverilog -o test-half-adder test-half-adder.vl
$ ./test-half-adder

```

compile le fichier source et le fait exécuter, ce qui affiche

```

0000    0 + 0 = 0 0
0010    0 + 1 = 0 1
0020    1 + 0 = 0 1
0030    1 + 1 = 1 0

```

3 Additionneur complet

Si on veut additionner des nombres de plus d’un bit, le demi-additionneur ne suffit pas. En effet, il y a des retenues à prendre en compte.

Exemple d'addition en binaire $1010 + 1111 = 11001$ (en décimal $10 + 15 = 25$).

Si on procède selon la méthode habituelle qui va de droite à gauche, à chaque étape (sauf tout à fait à droite), on doit additionner **trois** bits : un pour chaque nombre, et la retenue entrante obtenue au chiffre précédents.

3.1 Fonction de transfert

Dans le tableau ci-dessous, on distingue les retenues entrante **ci** et sortante **co**,

a	b	ci	co	s	a	b	ci	co	s
0	0	0	0	0	1	0	0	0	1
0	0	1	0	1	1	0	1	1	0
0	1	0	0	1	1	1	0	1	0
0	1	1	1	0	1	1	1	1	1

3.2 Câblage du l'additionneur

Le schéma montre comment, avec deux demi-additionneurs et une porte ou, on peut construire un additionneur complet.

3.3 Description structurelle en Verilog

En suivant le schéma, nous construisant l'additionneur avec deux instances nommées **h1** et **h2** du module demi-additionneur, et une porte ou :

```
# full-adder.vl
# additionneur complet (3 bits en entrée, somme et retenue en sortie)

`include "half-adder.vl"

module full_adder
(
    output co, s,
    input a, b, ci
);

    wire c1, c2, s1, s2;

    half_adder ha1(c1, s1, a, b);
    half_adder ha2(c2, s , s1, ci);
    or(co, c1, c2);

endmodule // full_adder
```

3.4 Simulation

La simulation est construite selon le même principe que précédemment

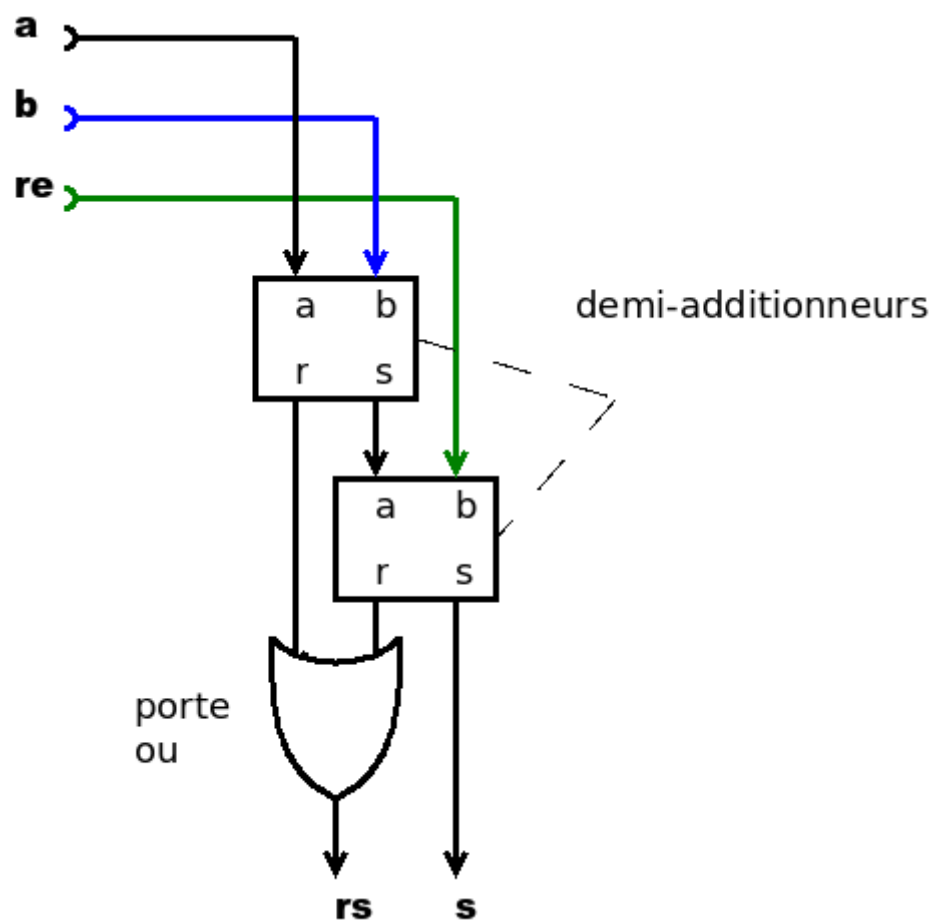


FIG. 2 : Circuit Additionneur

```

# test-full-adder.vl
# test de l'additionneur

`include "full-adder.vl"

module test_full_adder;
    reg a, b, ci;
    wire co, s;

    full_adder adder(co, s, a, b, ci);

    initial begin
        $monitor("%04t\t%b + %b + %b = %b%b", $time, a, b, ci, co, s);

        #0      a = 0; b = 0; ci = 0;
        #10      ci = 1;
        #10      b = 1; ci = 0;
        #10      ci = 1;
        #10      a = 1; b = 0; ci = 0;
        #10      ci = 1;
        #10      b = 1; ci = 0;
        #10      ci = 1;
    end

endmodule // test_full_adder

```

3.5 Compilation et exécution

```

$ iverilog -o test-full-adder test-full-adder.vl
$ ./test-full-adder
0000    0 + 0 + 0 = 00
0010    0 + 0 + 1 = 01
0020    0 + 1 + 0 = 01
0030    0 + 1 + 1 = 10
0040    1 + 0 + 0 = 01
0050    1 + 0 + 1 = 10
0060    1 + 1 + 0 = 10
0070    1 + 1 + 1 = 11

```

4 Additionneur 4 bits

Un additionneur de 4 bits est constitué de 4 additionneurs 1 bit, en connectant la retenue sortante de l'un à la retenue entrante de son voisin.

Pour pouvoir chaîner des additionneurs 4 bits, en plus des 2 nombres de 4 bits on a aussi une retenue entrante et une retenue sortante.

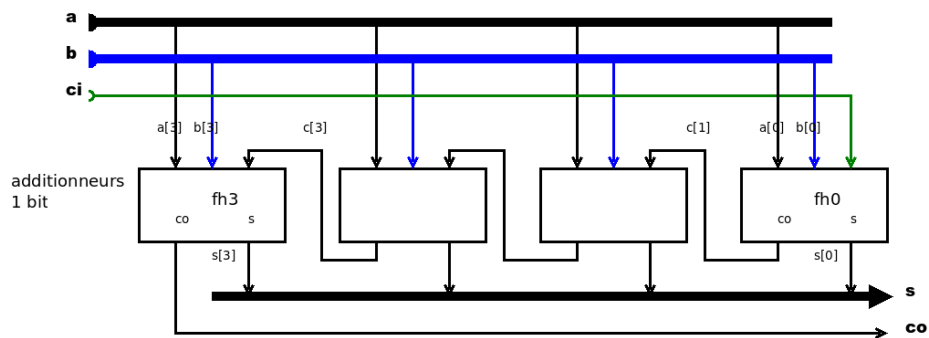


FIG. 3 : Circuit Additionneur 4 bits

4.1 Description en Verilog

```
// four-bit-adder.vl
// Additionneur 2 x 4 bits, avec retenues entrante et sortante

`include "full-adder.vl"

module four_bit_adder
(
    output          co,
    output[NB_BITS-1:0] s,
    input [NB_BITS-1:0] a,
    input [NB_BITS-1:0] b,
    input           ci
);

    parameter NB_BITS = 4;

    wire [NB_BITS : 1] c;

    full_adder fh0(c[1], s[0], a[0], b[0], ci);
    full_adder fh1(c[2], s[1], a[1], b[1], c[1]);
    full_adder fh2(c[3], s[2], a[2], b[2], c[2]);
    full_adder fh3(co, s[3], a[3], b[3], c[3]);

endmodule // four_bit_adder
```

Notes :

1. Pour la lisibilité, on utilise une constante `NB_BITS` qui vaut 4.
2. Les entrées `a` et `b` et la sortie `s` sont des “bus”, des faisceaux de fils indicés de 0 à `NB_BITS-1`.
3. Les retenues sont interconnectées par un bus interne `c` dont les fils sont numérotés de 1 à `NB_BITS-1`.

Remarque : Les connexions des 4 instances de “full_adder” étant similaires, on pourrait utiliser une possibilité plus avancée de Verilog, consistant à écrire une

boucle de génération.

4.2 Vérification du fonctionnement

Ce circuit possède 2 entrées de 4 bits, plus une pour la retenue entrante, soit 9 bits.

Le test va consister à vérifier que pour chaque combinaison possible (qui représente deux entiers entre 0 et 15 et une retenue entrante qui vaut 0 ou 1), ce que le circuit calcule correspond bien à la somme de ces trois valeurs.

Au lieu de tout faire afficher (512 cas, soit une dizaine de pages de texte à raison d'une ligne par cas...), nous faisons seulement afficher les anomalies.

```
// test-four-bit-adder.vl

`include "four-bit-adder.vl"

module test_four_bit_adder;
    parameter NB_BITS = 4;
    parameter MAX_INT = (1 << NB_BITS) - 1;

    reg [NB_BITS - 1:0] a;
    reg [NB_BITS - 1:0] b;
    reg ci;
    wire [NB_BITS - 1:0] s;
    wire co;
    four_bit_adder adder(co, s, a, b, ci);

    initial begin : simulation
        integer i, j, k;
        integer expected;
        integer errors, cases;
        $display("# Testing four-bit adder");
        errors = 0;
        cases = 0;

        for (i = 0; i <= MAX_INT; i = i + 1) begin
            for (j = 0; j <= MAX_INT ; j = j + 1) begin
                for (k = 0; k <= 1; k = k + 1) begin
                    a = i;
                    b = j;
                    ci = k;
                    expected = i + j + k;
                    cases = cases + 1;
                    #1 ;

                    if ({co,s} != expected) begin
                        $display("- error %1d + %1d + %1d is %1d instead of %1d",
                                a, b, ci, {co, s}, expected);
                        errors = errors + 1;
                    end
                end
            end
        end
    end
endmodule
```

```

        end

        end // k loop
    end // j loop
end // i loop

$display("- %1d failures over %1d test cases", errors, cases);
end
endmodule // test_four_bit_adder

```

Pour bien faire, nous affichons à la fin un récapitulatif du nombre de cas testés et du nombre d'anomalies détectées.

Explications

- Les cas sont générées par la triple boucle sur *i*, *j* et *k*.
- L'affectation (exemple *a = i*) d'un entier à un bus revient à placer les bits de sa représentation binaire sur les fils du bus.
- une expression comme *{co,s}* représente la concaténation du bit de *co* avec ceux du bus *s*.
- Sa valeur numérique - par exemple dans le test *if ({co,s} != expected)* - correspond aussi au codage binaire des entiers.

Importance du délai : dans la boucle, nous insérons un délai (#1) parce qu'après avoir placé les données dans les registres en entrée du circuit (*a = i; b = j; ci = k;*) il faut un peu de temps avant que les résultats soient disponibles sur les fils de sortie. La commutation de portes logiques n'est pas instantanée.

4.3 Exécution du test

```

$ iverilog -o test-four-bit-adder test-four-bit-adder.vl
$ ./test-four-bit-adder
# Testing four-bit adder
- 0 failures over 512 test cases

```

Tout va bien.