Tom GAULTIER

TP : Régression Linéaire & Tranpilers Verilog

IA et systèmes embarqués

PARIS PANTHÉON-ASSAS UNIVERSITÉ

Verilog est un Hardware description language comme le VHDL. Il est très utilisé dans l'industrie donc on va voir comment implémenter des modèles simples en verilog.

Nous allons utiliser un logiciel de simulation nommé icarus iverlog qui permet à partir de spécification verilog de générer une simulation exécutable. Cette simulation nous permettra de générer des tests bench éventuellement affichables avec des logiciels de type waveform.

Exercice 0

Installer icarus iverilog et gtkwave (via apt ou le lien suivant : https://bleyer.org/icarus/)
En cas de problème demander une VM Linux sur lequel ce sera installé

Exercice 1

Récupérer le code accessible à ce lien. Il implémente une porte logique ET et le fichier de simulation correspondant.
Pour afficher les waveform d'un test bench il faut faire trois étapes :

1. Compiler les modules verilog avec le test bench
2. lancer le binaire obtenu et afficher la trace textuelle du dump
3. lancer vvp sur l'executable obtenu
4. Charger dans gtkwave le fichier vcd obtenu.

Exercice

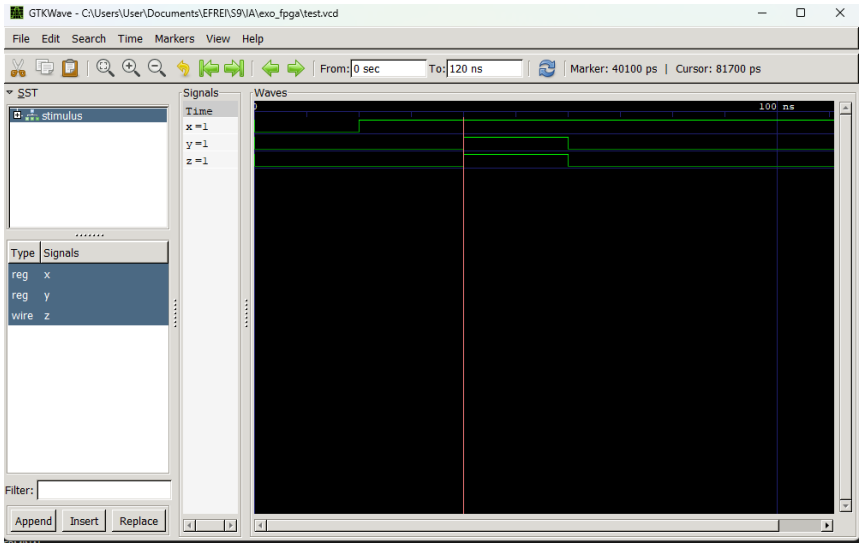
1. Faire en sorte de compiler le module et le testbench avec iverilog

```
iverilog -o test_and_gate.vvp test_and_gate.v
```

2. Lancer la simulation avec vvp sur le fichier binaire obtenu

```
PS C:\Users\User\Documents\EFREI\S9\IA\exo_fpga> vvp test_and_gate.vvp
VCD info: dumpfile test.vcd opened for output.
t=  0 x=0,y=0,z=0

t= 60 x=1,y=0,z=0
```

3. Afficher la waveform gtkform et vérifier que la porte logique se comporte de manière cohérente



4. Noter comment le module de la porte ET est défini

A 20 ns, le signal *x* monte à 1, 20 ns après, soit à 40 ns, le signal *y* monte également à 1. Au même instant, le signal *z* prend la valeur 1. Encore 20 ns après, le signal *y* redescend à 0, et le signal *z* a la même attitude. Le signal *z* dans la structure *and_gate* est la sortie représentant la porte ET, et le signal monte bien à 1 lorsque les deux entrées, signaux *x* et *y*, sont à 1. La porte ET est bien définie.

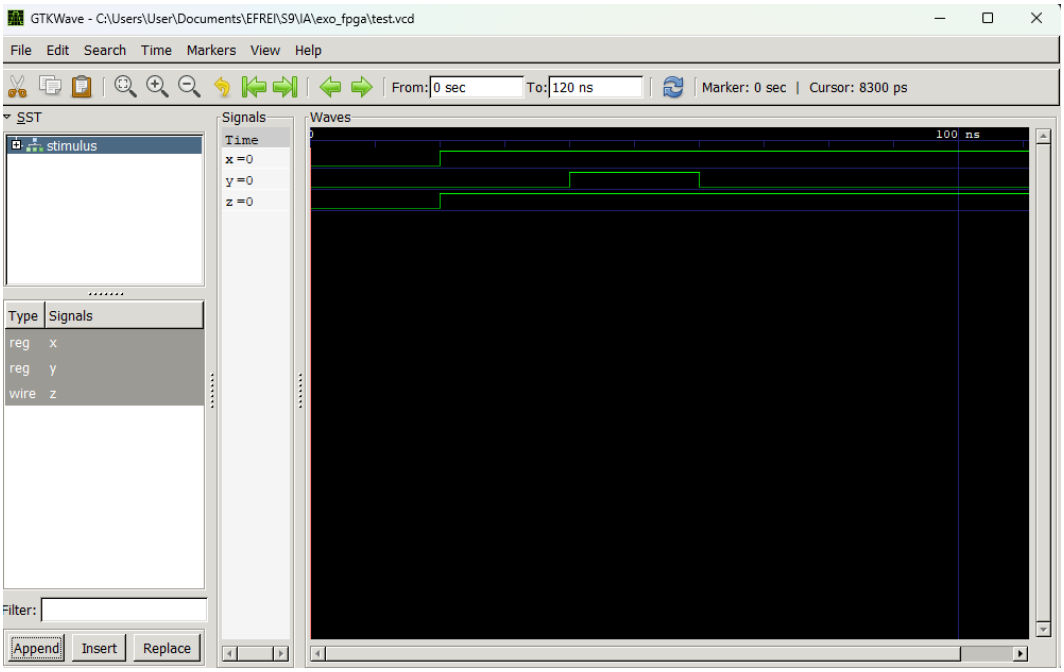
Exercice 2 - Porte logique Ou

1. Copier le module de la portie logique OU et le modifier pour qu'il implémente une porte logique OU

or_gate.v

```
1  module and_gate (e1, e2, s);
2      input e1;
3      input e2;
4      output s;
5
6
7      assign s = e1 | e2;
8
9  endmodule
```

2. Reprendre le testbench de l'exercice précédent et vérifier que le module se comporte convenablement.



Le comportement du module suit bien le fonctionnement de la porte OU. La sortie z monte à la valeur 1 lorsque au moins l'une des deux entrées, x ou y, est à 1.

Exercice Reg vs wire

En faisant des recherches si nécessaire, expliquer la différence entre reg versus wire. Dans quels cas est-il plus pertinent d'utiliser l'un que l'autre ?

Les reg et wire sont des sous-types de signaux qui peuvent être définis en utilisant les mots-clés respectifs. Le signal reg permet de stocker l'état précédent du signal et met à jour la valeur à chaque coup d'horloge. On utilise des signaux wire pour des signaux séquentiels, notamment lorsqu'ils dépendent de leur état précédent comme une machine d'état d'un bloc ou une entrée du système. Le signal wire quant à lui est utilisé pour les signaux combinatoires, dont leur valeur dépend d'une combinaison d'entrées. Sa valeur n'a pas besoin d'être stockée, comme les signaux de sorties de la conception.

Exercice - assign vs always

En faisant des recherches si nécessaire, expliquer la différence entre assign vs always. Dans quels cas est-il plus pertinent d'utiliser l'un que l'autre ?

Assign et always sont des mots-clés utilisés pour assigner une valeur à une variable.
La méthode la plus utilisée est Assign, puisqu'elle permet d'attribuer la valeur directement à la lecture de la ligne. Alors que Always attribue la valeur à l'apparition de l'évènement associé au block Always.

Exercice - Porte logique ET à trois entrées

1. Faire un module qui implémente une porte logique ET à trois entrées

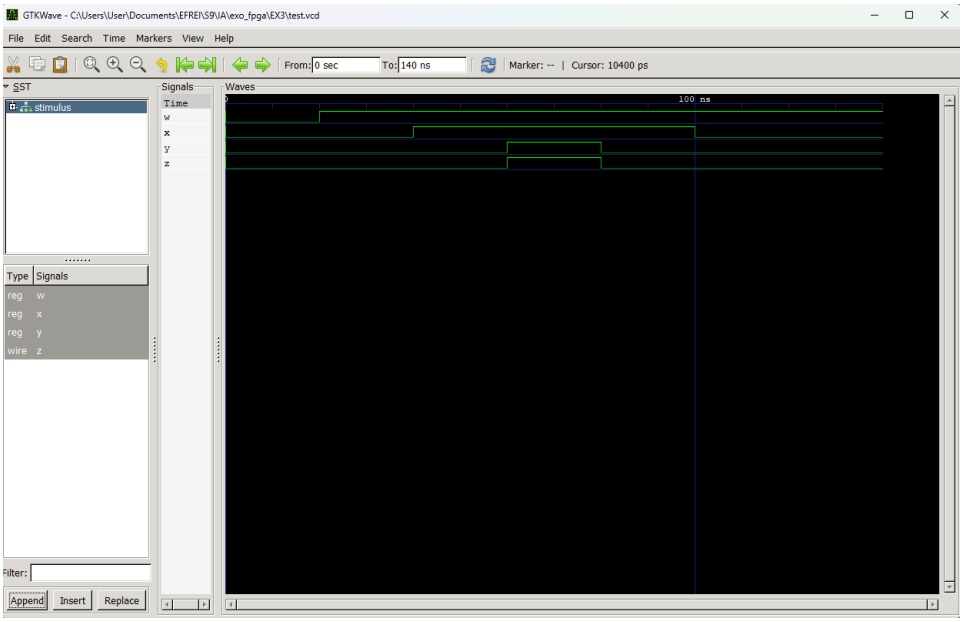
and_3entries_gate.v

```v
module and_3entries_gate (e1, e2, e3, s);
    input e1;
    input e2;
    input e3;
    output s;


    assign s = e1 & e2 & e3;

endmodule
```

2. Reprendre et modifier le fichier de test précédent pour qu'il marche avec trois entrées

test_and_3entries_gate.v

```v
`timescale 1ns / 1ps
`include "and_3entries_gate.v"

module stimulus;
    // Inputs
    reg w;
    reg x;
    reg y;
    // Outputs
    wire z;
    // Instantiate the Unit Under Test (UUT)
    and_3entries_gate uut (
        w,
        x,
        y,
        z
    );

    initial begin
    $dumpfile("test.vcd");
    $dumpvars(0,stimulus);
    // Initialize Inputs
    w = 0;
    x = 0;
    y = 0;

    #20 w = 1;
    #20 x = 1;
    #20 y = 1;
    #20 y = 0;
    #20 x = 0;
    #40 ;

    end

    initial begin
    $monitor("t=%3d w=%d,x=%d,y=%d,z=%d \n",$time,w,x,y,z, );
    end

endmodule
```

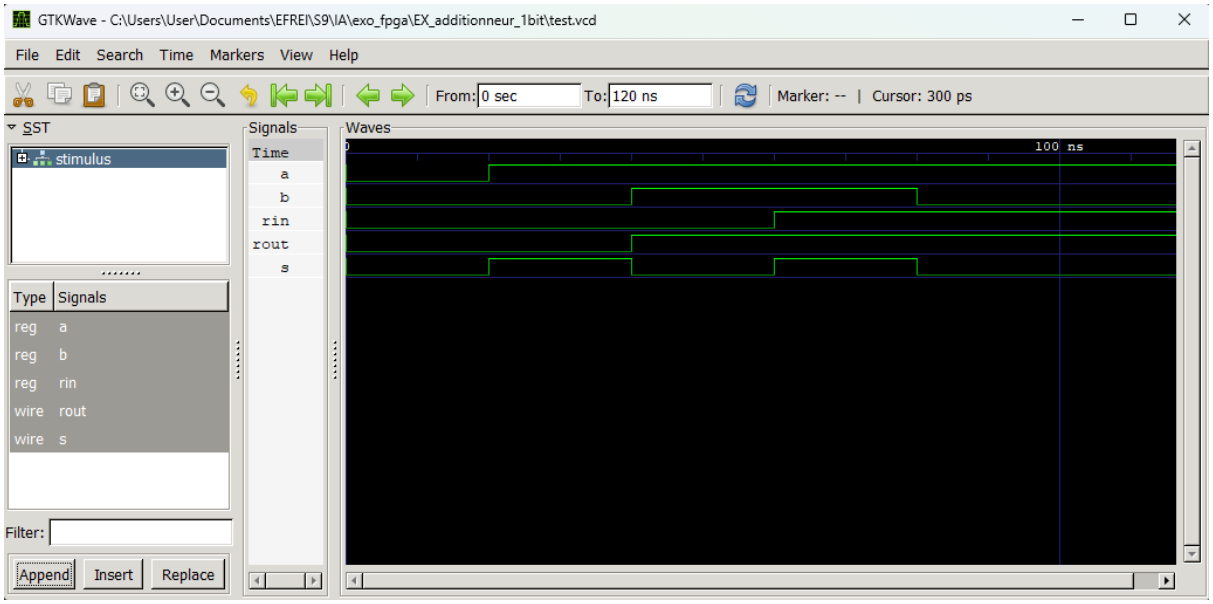


Exercice - Additionneur 1 bit

1. On veut faire un additionneur 1 bit qui prend en entrée 2 nombre a et b, une retenue, et qui en sortie renvoie l'addition des deux nombres et l'éventuelle retenue de sortie. Implémenter ce module

```
`timescale 1ns / 1ps
`include "additionneur_1bit.v"

module stimulus;
    // Inputs
    reg a;
    reg b;
    reg rin;
    // Outputs
    wire s;
    wire rout;
    // Instantiate the Unit Under Test (UUT)
    additionneur_1bit uut (
        a,
        b,
        rin,
        s,
        rout
    );

    initial begin
    $dumpfile("test.vcd");
    $dumpvars(0,stimulus);
    // Initialize Inputs
    a = 0;
    b = 0;
    rin = 0;

    #20 a = 1;
    #20 b = 1;
    #20 rin = 1;
    #20 b = 0;
    #40 ;

    end

    initial begin
    $monitor("t=%3d a=%d,b=%d,rin=%d,s=%d,rout=%d \n",$time,a,b,rin,s,rout, );
    end

endmodule
```

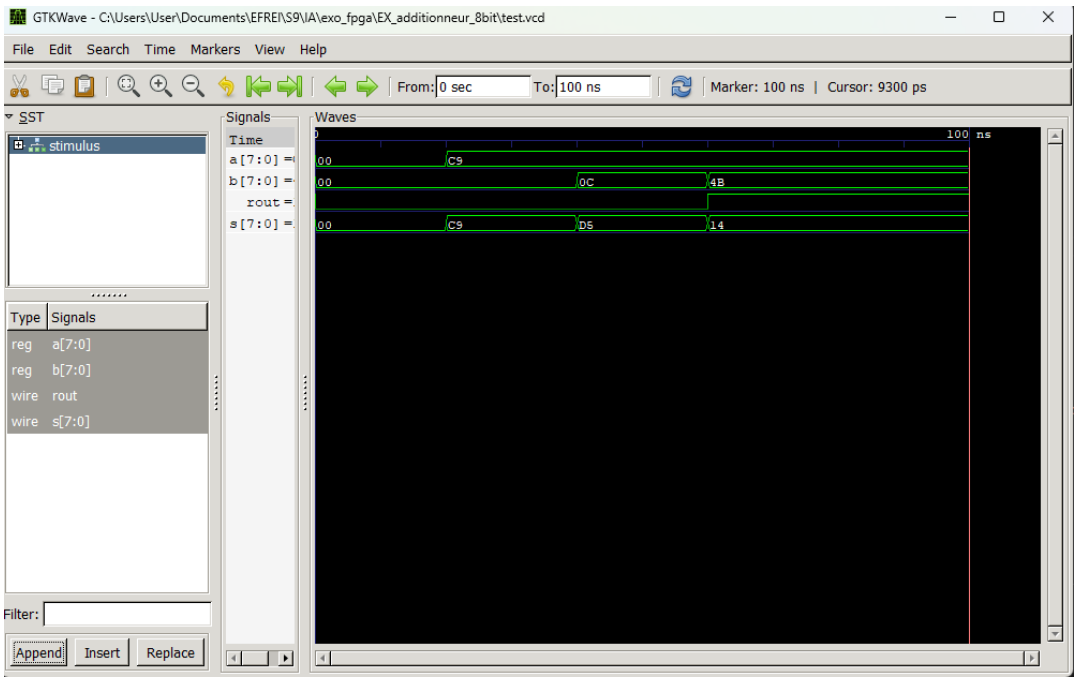
2. Reprendre et modifier le fichier de test précédent pour faire en sorte de simuler le comportement de l'additionneur 1bit



Additionneur 8bit

faire en sorte d'avoir un module qui fait l'addition de deux nombres 8bit et qui renvoie le resultat et une éventuelle retenue.

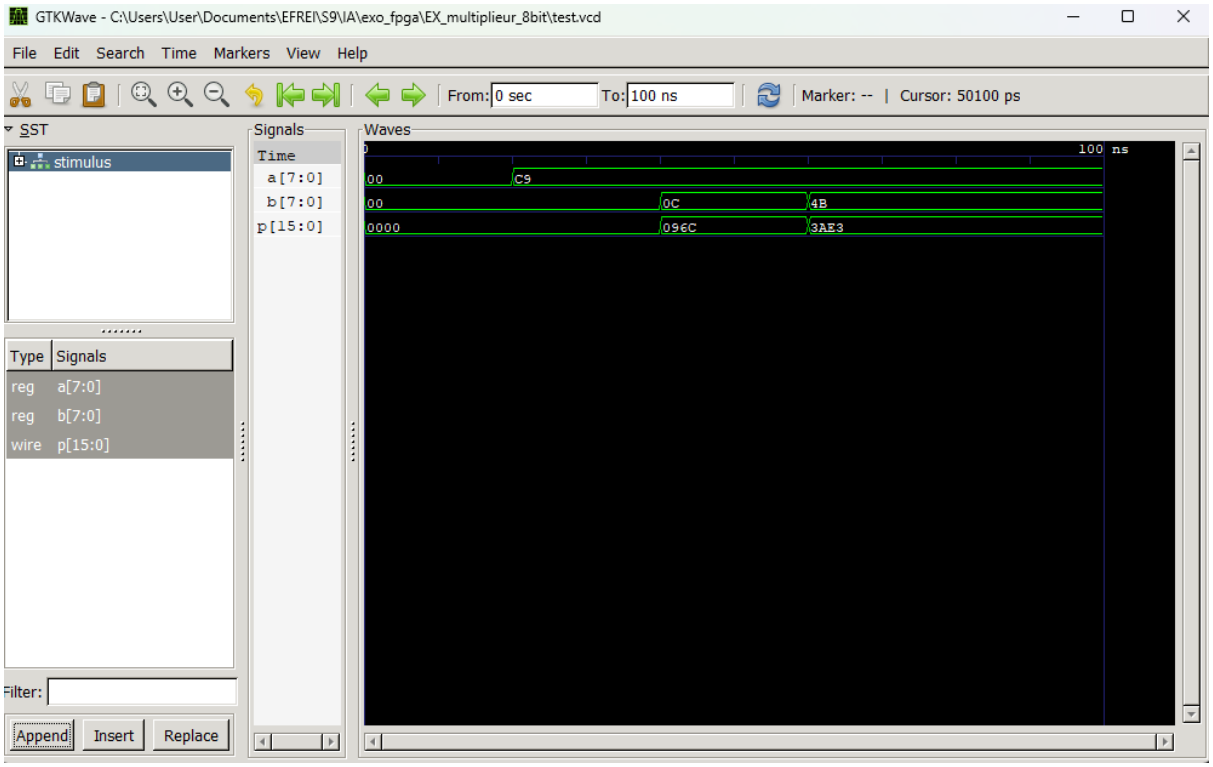
En reprenant le block Additionneur_1bit, nous pouvons faire l'addition bit par bit.



Multiplieur 8 bit

Faire un module multiplier_8b qui prend deux vecteurs de bit de taille 8 et a en sortie un vecteur de bit de taille 16 et qui effectue la multiplication.

La multiplication 8 bits revient à faire des additions en décalant le second opérant, donc en multipliant le second opérant par une puissance de 2.



201 * 12 = 2412

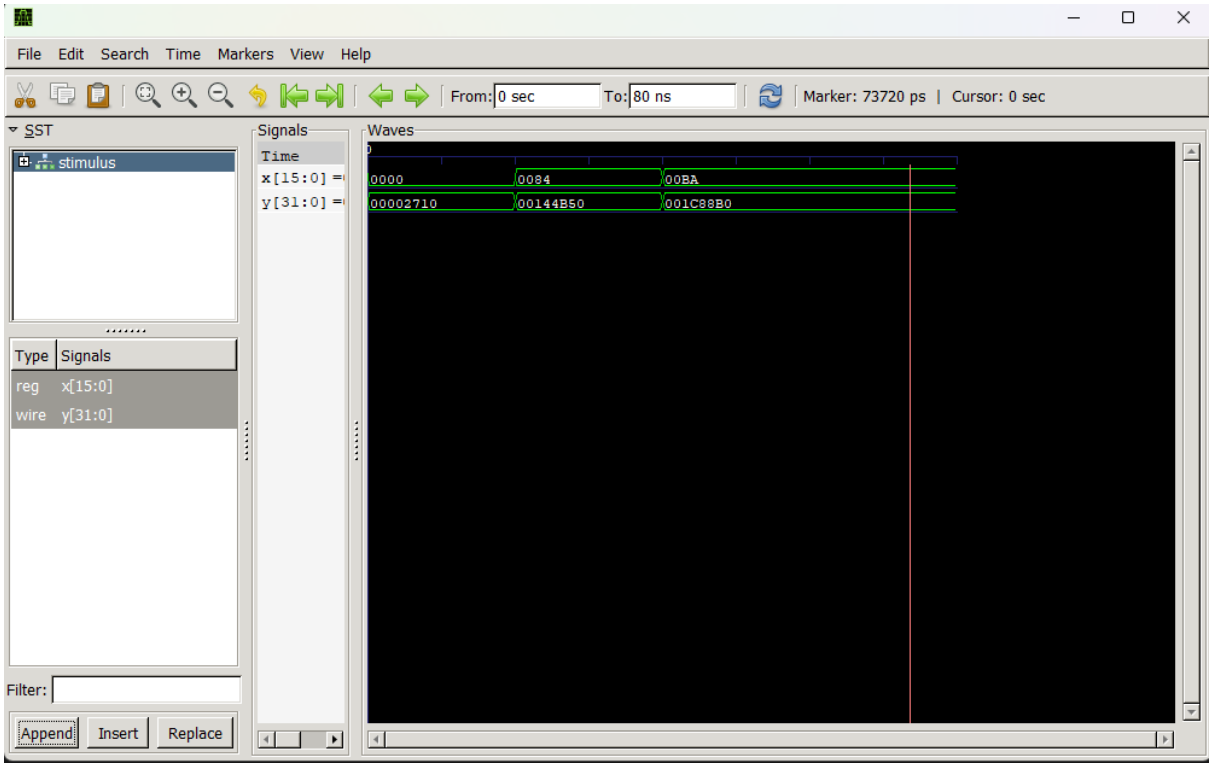
201*75=15075

Régression linéaire simple

Implémenter une régression linéaire qui utilise les modules précédents qui calcule

y = 10000 + 10000 * X1 ou le prix est le prix de la maison et size est sa taille

Afin de pouvoir faire les calculs de la régression linéaire, il a fallu produire de plus grand additionneur et multiplicateur, car le facteur 10 000 ne tient pas sur 8 bits.



Y = 10 000 + 10 000 * X = 10 000 + 10 000 * 186 = 1 870 000 = 001C 88B0

Transpileur

Faire un script python qui serait capable de générer un module regression_lineare à partir d'un modèle scikit-learn entraîné et qui permettrait de gérer un nombre quelconque de feature.

Le fichier transpiler.py crée un modèle de régression linéaire avec le csv houses.csv et créer le module verilog du modèle de régression en ajoutant le nombre d'additionneurs et de multiplicateurs selon le nombre de coefficients dans le modèle.