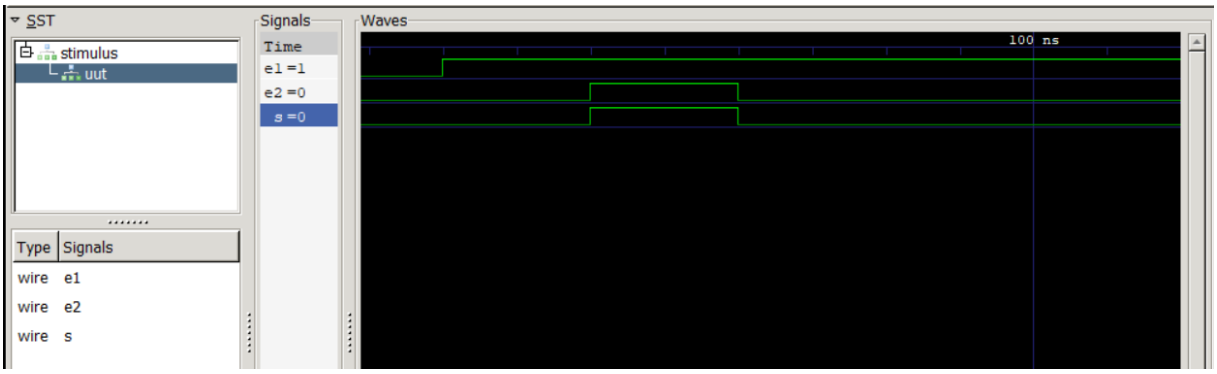
TP2 : Exercice et réponse

Exercice 1 :

Il est important de vérifier que tout fonctionne. Nous avons bien 0 lorsque x et/ou y sont à 0. Et nous avons bien 1 lorsqu'ils valent tous les deux 1. Donc cette porte ET fonctionne.

```
C:\Users\Marie\Documents\AI>cd TP2

C:\Users\Marie\Documents\AI\TP2>vvp a.out

C:\Users\Marie\Documents\AI\TP2>iverilog and_gate.v test_and_gate.v

C:\Users\Marie\Documents\AI\TP2>vvp a.out
VCD info: dumpfile test.vcd opened for output.
t=  0 x=0,y=0,z=0

t= 20 x=1,y=0,z=0

t= 40 x=1,y=1,z=1

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



Exercice 2

Pour la porte OU, j'ai simplement changé le symbole en remplaçant le ET par OU.

Nous retrouvons bien les réponses d'une porte OU :

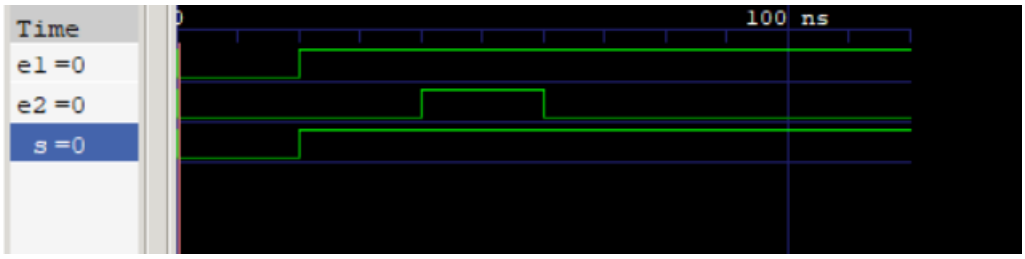
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

C: > Users > Marie > Documents > AI > TP2 > and_gate.v

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


    assign s = e1 | e2;

endmodule
```

```
C:\Users\Marie\Documents\AI\TP2>iverilog and_gate.v test_and_gate.v
```

```
C:\Users\Marie\Documents\AI\TP2>vvp a.out
VCD info: dumpfile test.vcd opened for output.
t=  0 x=0,y=0,z=0

t= 20 x=1,y=0,z=1

t= 40 x=1,y=1,z=1

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


C:\Users\Marie\Documents\AI\TP2>gtkwave
```



Exercice Reg vs wire

Reg permet de créer des variables de type registre alors que wire représente des variables de type filaire. Ainsi Reg est plutôt pour des variables qui vont rester dans leur module alors que wire permet d'avoir des variables qui peuvent passer d'un module à un autre.

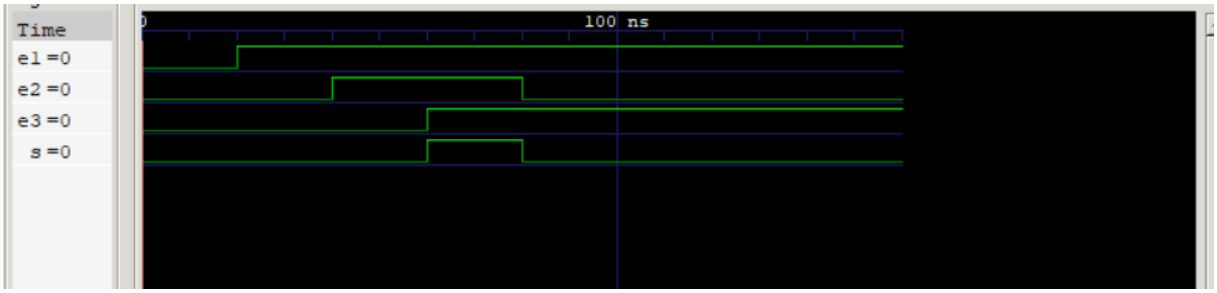
Exercice – assign vs always

Assign correspond au return d'un module alors que always permet de décrire des circuits qui ont besoin de la sortie dans l'entrée. Ainsi, Assign est plus continu alors que always est séquentiel et à notamment besoin d'une clock pour changer d'état.

Exercice 3

```
module exercice3(e1, e2, e3, s);
    input e1;
    input e2;
    input e3;
    output s;

    assign s = e1 & e2 & e3; // Exercice 3

endmodule
```

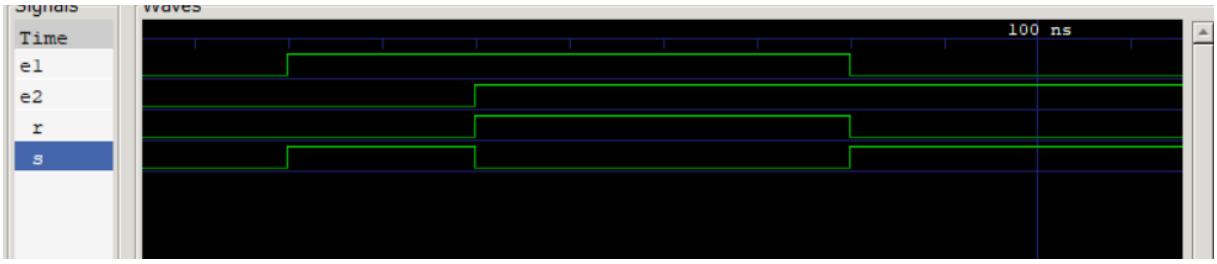
```
C:\Users\Marie\Documents\AI\TP2>iverilog and_gate.v test_exercice3.v

C:\Users\Marie\Documents\AI\TP2>vvp a.out
VCD info: dumpfile test.vcd opened for output.
t=  0 x=0,y=0,a=0,z=0

t= 20 x=1,y=0,a=0,z=0

t= 40 x=1,y=1,a=0,z=0

t= 60 x=1,y=1,a=1,z=1

t= 80 x=1,y=0,a=1,z=0
```



Additionneur 1 Bit

```
module Add1B(e1, e2, r, s);
    input e1;
    input e2;
    output r;
    output s;

    assign r = e1 & e2;
    assign s = e1 ^ e2;

endmodule
```

```
C:\Users\Marie\Documents\AI\TP2>iverilog and_gate.v test_Add1B.v

C:\Users\Marie\Documents\AI\TP2>vvp a.out
VCD info: dumpfile test_Add1B.vcd opened for output.
t=  0 x=0,y=0, retenue=0, result=0

t= 20 x=1,y=0, retenue=0, result=1

t= 40 x=1,y=1, retenue=1, result=0

t= 80 x=0,y=1, retenue=0, result=1
```



Additionneur 8 bit

Pour des raisons pratiques, j'ai modifié mon additionneur 8 bits et l'est passé en additionneur 32 bits pour la suite du TP. Maintenant, en changeant simplement les valeurs des vecteurs, nous obtenons la même chose.

```
module adder8bit(A, B, Cin, Sum, Cout);

input [31:0] A, B;
input Cin;
output [31:0] Sum;
output Cout;

wire [32:0] temp_sum;
wire carry;

assign temp_sum = A + B + Cin;
assign Sum = temp_sum[31:0];
assign Cout = temp_sum[32];

endmodule
```

```
PS C:\Users\Marie> cd Documents\AI\TP2
PS C:\Users\Marie\Documents\AI\TP2> iverilog and_gate.v test_Add8B.v
PS C:\Users\Marie\Documents\AI\TP2> vvp a.out
VCD info: dumpfile test_Add8B.vcd opened for output.
t=  0 x=         0,y=         0, retenue=0, result=         0

t= 20 x=         1,y=         0, retenue=0, result=         1

t= 40 x=         1,y=         2, retenue=0, result=         3

t= 60 x=         1,y=     65535, retenue=0, result=     65536

t= 80 x=       255,y=     65535, retenue=0, result=     65790
```

Multiplieur 8 bit

Pour le multiplieur, l'idée est simplement de décaler les bits tout en les additionnant aux précédents.

Et, tout comme l'additionneur, j'ai dû augmenter sa capacité jusqu'à 16 bits pour pouvoir faire l'exercice de régression.

```
module mult8bit(A, B, P);

input [15:0] A, B;
output reg [31:0] P;

integer i;

always @(*) begin
  P = 0;
  for (i = 0; i < 16; i = i + 1) begin
    if (B[i] == 1) begin
      P = P + (A << i);
    end
  end
end

endmodule
```

```
PS C:\Users\Marie\Documents\AI\TP2> iverilog and_gate.v test_Mul8B.v
PS C:\Users\Marie\Documents\AI\TP2> vvp a.out
VCD info: dumpfile test_Mul8B.vcd opened for output.
t=  0 x=    0, y=    0, result=         0

t= 20 x=    1, y=    0, result=         0

t= 40 x=    1, y=    2, result=         2

t= 60 x=    1, y=65535, result=     65535

t= 80 x=  255, y=65535, result=  16711425
```

Régression linéaire simple

L'idée est simplement de reprendre les modules précédents et de les réutiliser.

```
module Regression(f0, c1, c0, Cin, y);

input [15:0] f0;
input [15:0] c1;
input [31:0] c0;
input Cin;
output [31:0] y;

wire [31:0] r;
wire [31:0] rt;
wire Cout;

mult8bit mod1(f0, c1, r);
adder8bit mod2(c0, r, Cin, rt, Cout);

assign y = rt;

endmodule
```

```
PS C:\Users\Marie\Documents\AI\TP2> iverilog and_gate.v test_Regression.v
PS C:\Users\Marie\Documents\AI\TP2> vvp a.out
VCD info: dumpfile test_Regression.vcd opened for output.
t=  0 c0=         0, f0=    0, c1=    0, result=         0

t= 20 c0=         0, f0= 5000, c1=    0, result=         0

t= 40 c0=     10000, f0= 5000, c1=    0, result=     10000

t= 60 c0=     10000, f0= 5000, c1=  100, result=    510000
```

Transpiler

Pour cet exercice, j'ai simplement testé de dupliquer le nombre d'opération dans un premier temps (Module Regression_nV dans and_gate.v). Ensuite, j'ai réalisé un script python en le généralisant (Regression_nV_py.v et Regression_nV_py_test.v).

```
PS C:\Users\Marie\Documents\AI\TP2> iverilog Regression_nV_py.v Regression_nV_py_test.v
PS C:\Users\Marie\Documents\AI\TP2> vvp a.out
VCD info: dumpfile test_Transpiler_py.vcd opened for output.
c0=         0,f0=    0, c1=    0f1=    0, c2=    0f2=    0, c3=    0result=         0
c0=         2,f0=    3, c1=    1f1=    4, c2=    1f2=    5, c3=    1result=        14
```

On obtient bien les valeurs demandées pendant le code. Car :

c0 + c1*f0 + c2*f1 + c3*f2 = 14