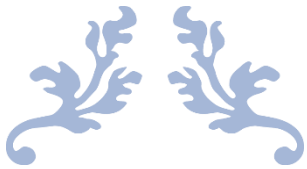


TP VERILOG

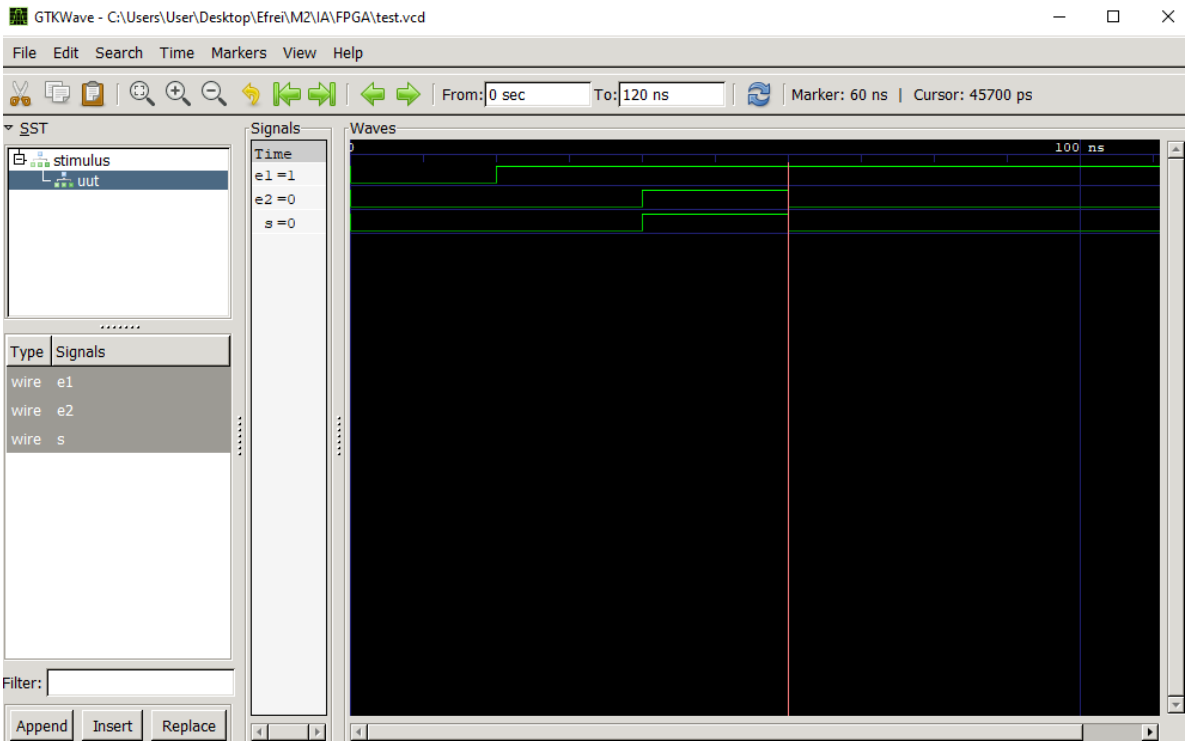
Paul Rotsztein

31 MARS 2000

EFREI PARIS

Exercice 1 – Porte ET – (and_gate.v | test_and_gate.v)

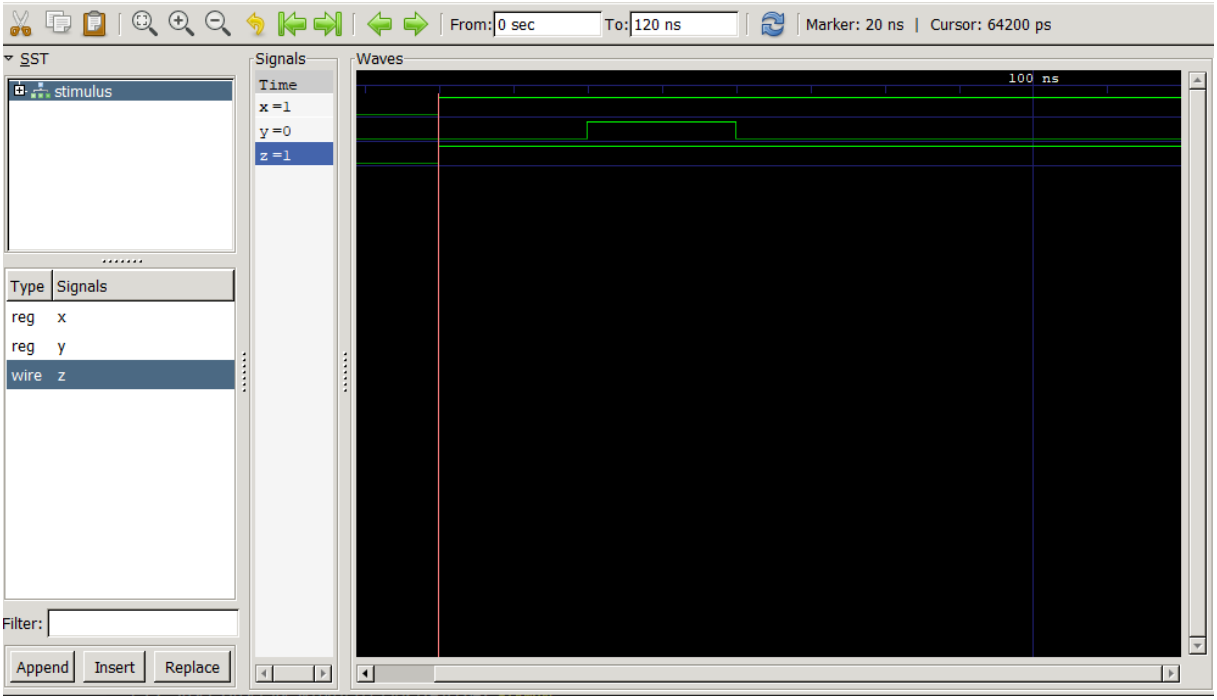
Nous avons lancé la simulation et afficher la waveform gtkform pour la porte logique ET :



On a un décalage de 20 ns entre chaque évènement comme décris dans le code. E1 = e2 = 1. Si e1 et e2 sont différents, alors s vaut 0. Sinon il vaut 1. C'est bien ce qu'on peut observer sur la waveform ci-dessus.

Exercice 2 – Porte OU - (or_gate.v | test_or_gate.v)

Pour la porte logique OU :



La sortie z est censé être à 1 lorsque x ou y vaut 1. Z doit valoir 0 lorsque x et y sont égale à 0. C'est bien ce qu'on peut observer sur la waveform ci-dessus. A 20 ns x et y valent 0, z vaut 0. Dès le premier changement de valeur après les 20 ns, on remarque que z monte à 1. Notre porte OU est validée.

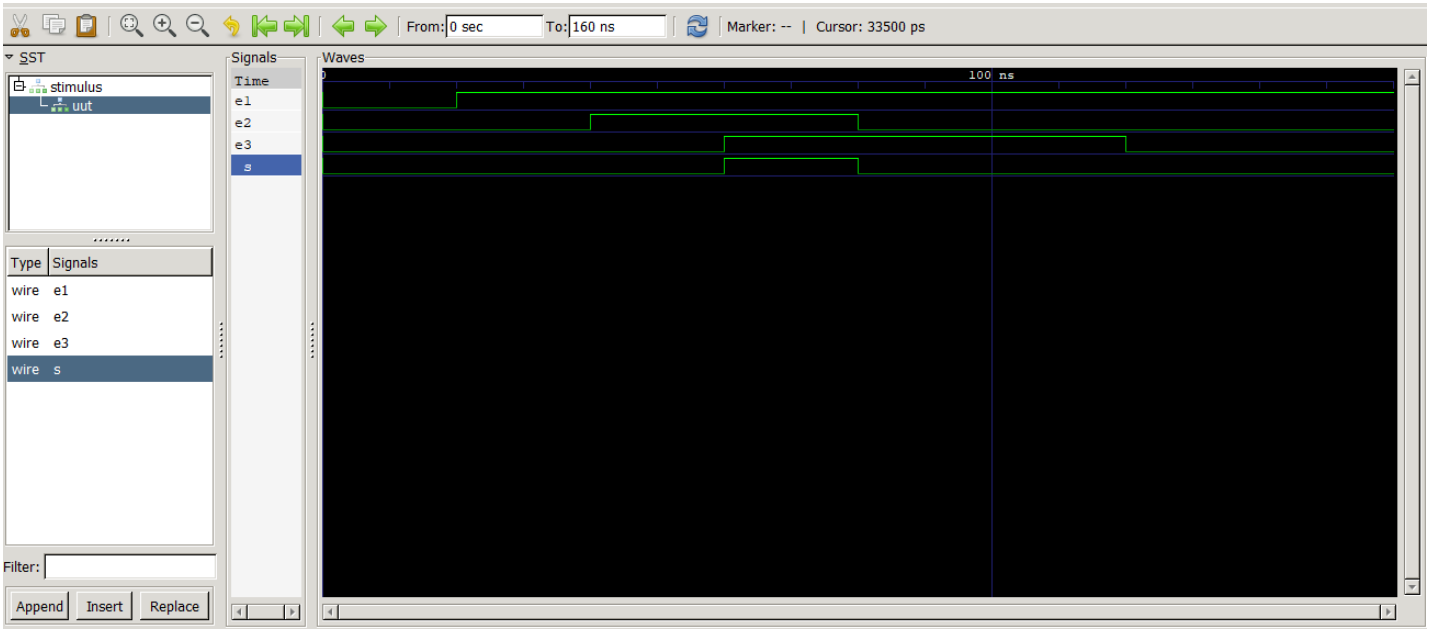
Exercice 3 – Reg vs Wire

Reg et Wire sont des types de variables utilisés dans le langage Verilog. La variable Reg est une variable registre, c'est-à-dire qu'elle conserve sa valeur entre les différents cycles de simulation. C'est donc adapté à la modélisation de stockage. A contrario, Wire est une variable fil, elle ne conserve pas sa valeur entre les cycles de simulation et est donc adapté à l'implémentation de signaux combinatoire (portes logiques par exemple…).

Exercice 4 – Assign vs Always

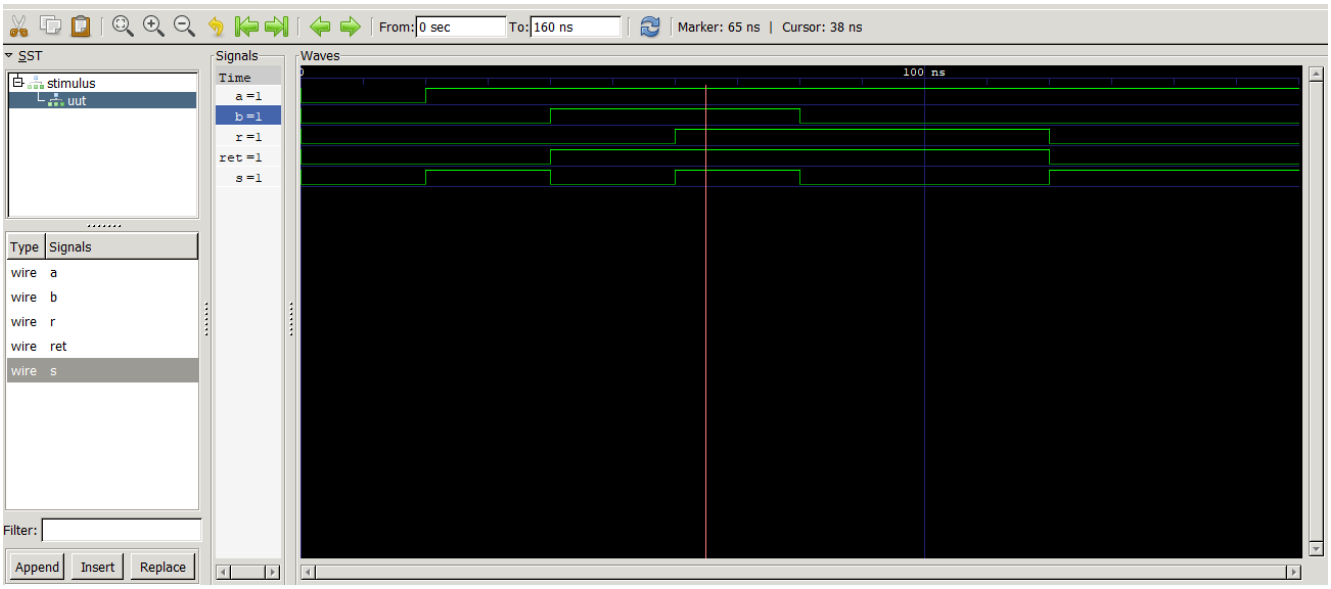
Assign est utilisé en Verilog pour assigner une valeur à une variable de type fil (wire), qui est une variable combinatoire. Always est utilisé pour assigner une valeur à une variable de type registre (reg) de manière séquentielle. La valeur assignée est calculée et stockée dans la variable à chaque cycle de simulation. C'est en fonction du type de la variable qu'on décide ce qu'on utilise entre Assign et Always.

Exercice 5 – Porte logique ET à trois entrées - (and3_gate.v | test_and3_gate.v)



La porte logique ET à trois entrées fonctionne de la même manière que celle à 2 entrées : la sortie est à 1 que lorsque toutes les entrées sont à 1. Sinon, la sortie vaut 0. C'est clairement ce qu'on peut observer sur le graphique au-dessus. La sortie S vaut 1 lorsque les trois signaux d'entrées sont à 1, sinon elle vaut 0.

Exercice 6 – Additionneur 1 bit - (add1bit.v | test_ add1bit.v)

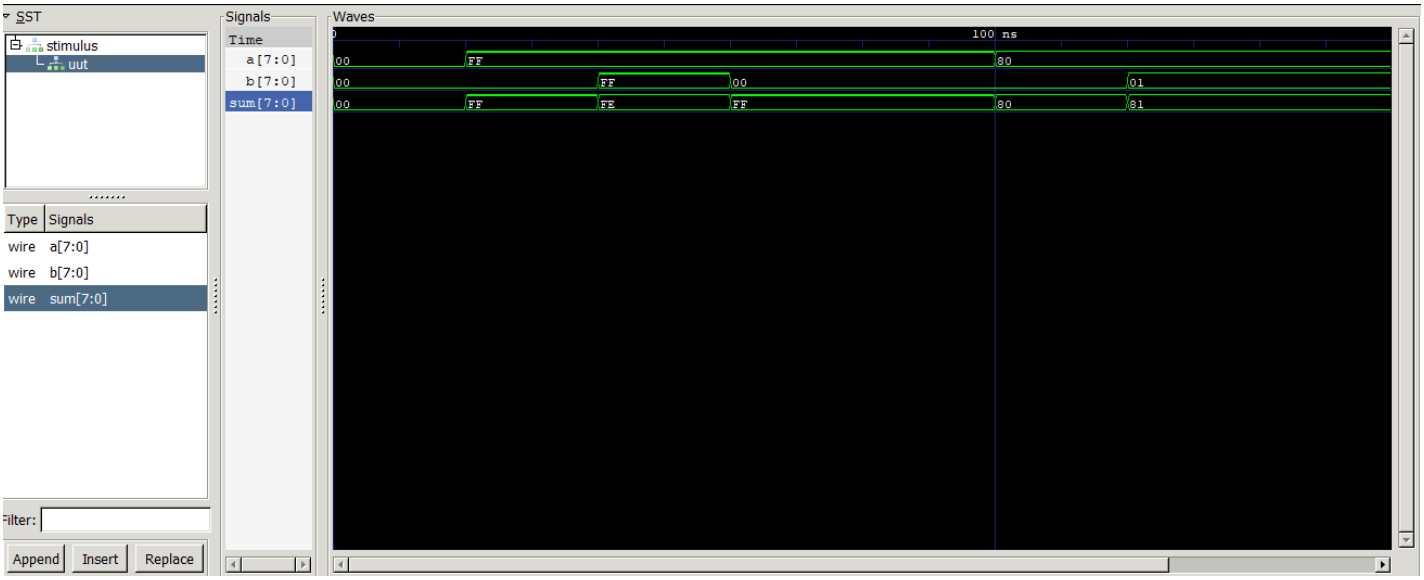


Pour l'additionneur 1 bit, on est avoir en sortie l'équivalent d'une table de vérité XOR :

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

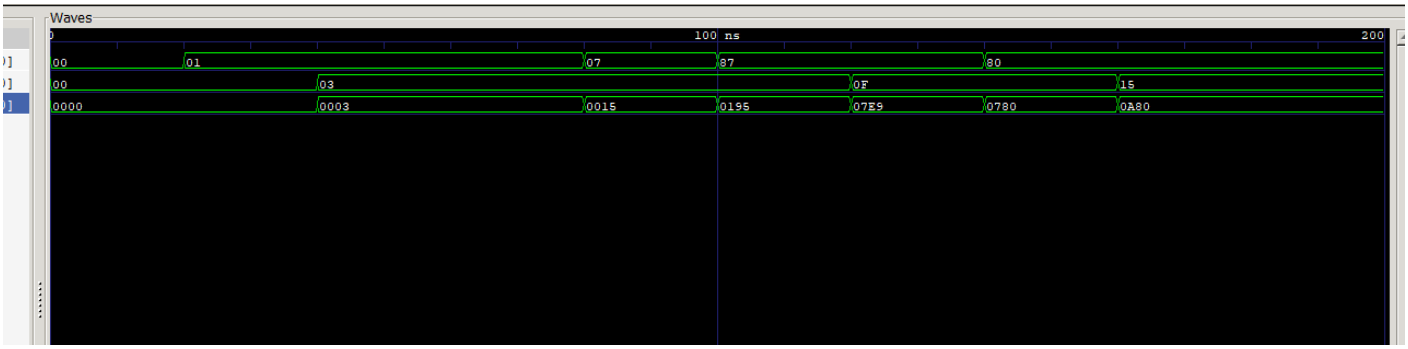
Quand a et b sont différents, la sortie est à 1, sinon elle est à 0. C'est déjà ce qu'on peut observer sur le graphe au-dessus. La retenu est incrémentée lorsque a et b valent 1 car sur 1 bit, 1 + 1 = 0(1 en retenu car il y a un dépassement d'un bit). Donc on ajoute à chaque fois au résultat de l'addition, la valeur courante de la retenu. C'est aussi ce qu'on peut observer sur la waveform.

Exercice 7 – Additionneur 8 bit - (add8bit.v | test_ add8bit.v)



Ici, on cherche à faire un additionneur sur 8 bits. On remarque bien que l'addition fonctionne correctement sur le graphe ci-dessus. La retenue n'est pas prise en compte ici, donc on remarque des dépassements de bits notamment à 40 ns avec a = b = FF ou (11111111) b : le résultat est donc FE car le résultat avec retenu serait = 1FE.

Exercice 8 – Multiplexer 8 bits - - (multi8bit.v | test_ multibit.v)



Pour la multiplication sur 8 bits, le principe est le même que pour l'addition mais on cherche à afficher le résultat sur 16 bits. Ici, encore on ne demande pas de retenu dans la consigne. On voit que les résultats sont cohérents, on peut voir qu'en hexadécimal :

- 01 * 00 = 00
- 01 * 03 =0003
- 80 * 0F = 0780
- 80 * 15 = 0A80

Notre graphe affiche donc les bons résultats !