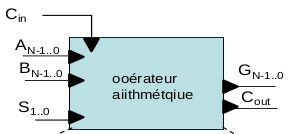
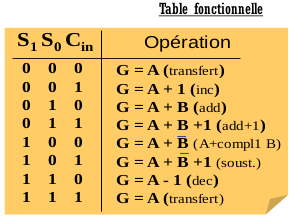
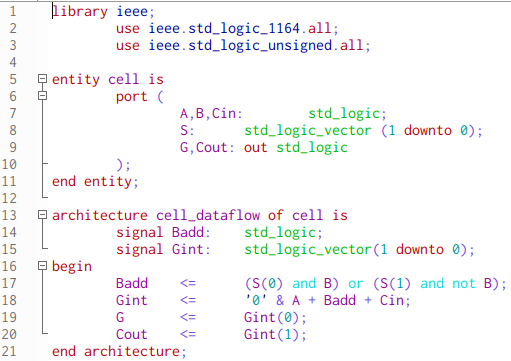
COMPTE RENDU TP4 GENERICITE

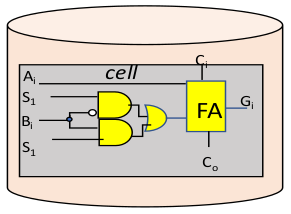
# OPERATEUR ARITHMETIQUE

Dans cette première partie, nous allons décrire un opérateur arithmétique de taille paramétrable, qui sera capable de réaliser huit opérations différentes :

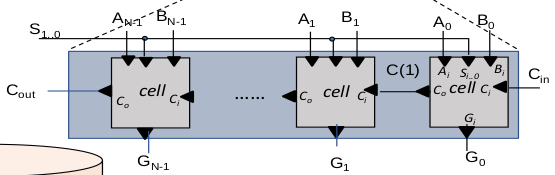


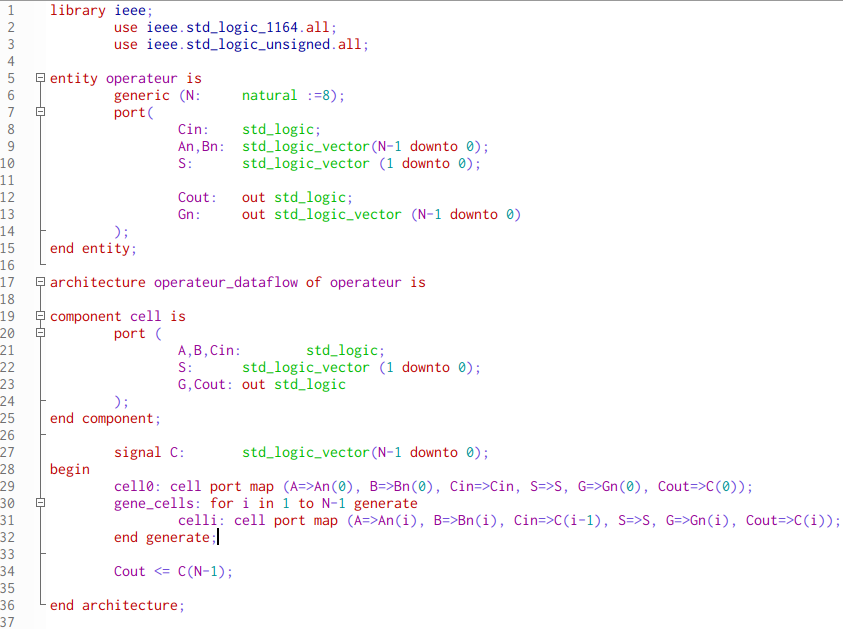
Pour se faire, nous avons dans un premier temps décrit un couple entité/architecture qui correspond à la description de la cellule élémentaire, comme on peut le voir ci-dessous.



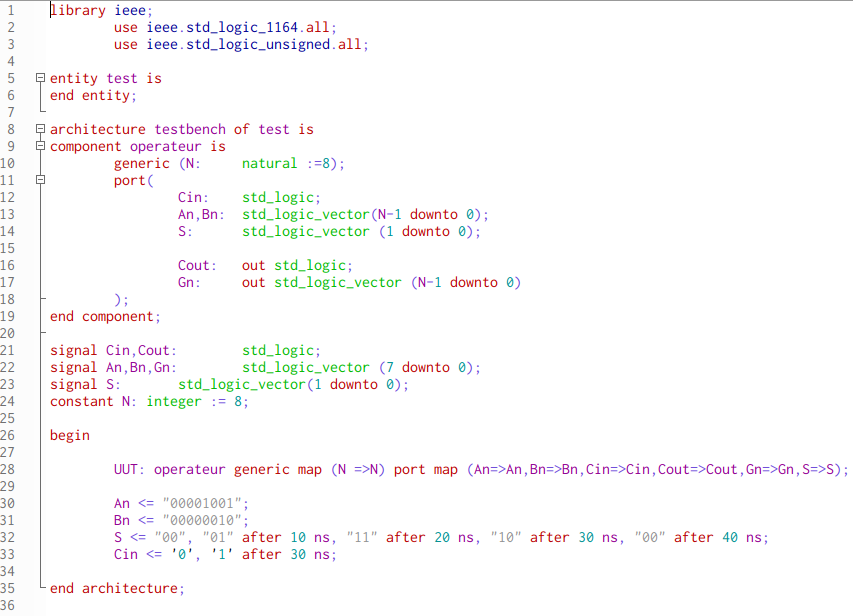


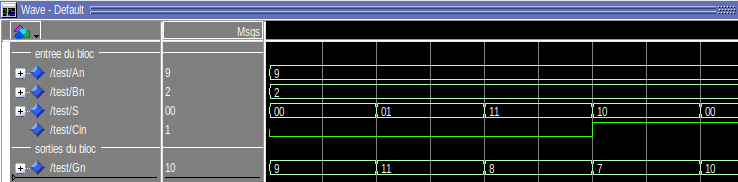
Badd, A et Cin sont les entrées du Full Adder, tandis que G et Cout correspondent aux sorties. On remarque que Gint est sur 2 bits pour prévenir d’un dépassement d’une addition de 2 nombres sur 1 bit.

Nous avons ensuite codé un second couple entité/architecture qui décrit l’opérateur complet en assemblant toutes les cellules en nombre paramétrable. Le code correspond donc à la description ci-contre :

Ainsi, afin d’assembler les N cellules, on créer la cellule 0 puis on fait une boucle for qui va de 1 jusqu’à N-1, pour créer les autres cellules. 

Nous avons ensuite écrit le test bench puis simulé pour un N = 8 cellules.



Nous pouvons voir que les opérations que nous avons testées marchent. En effet : 

000 → G = A = 9

S1S0Cin=

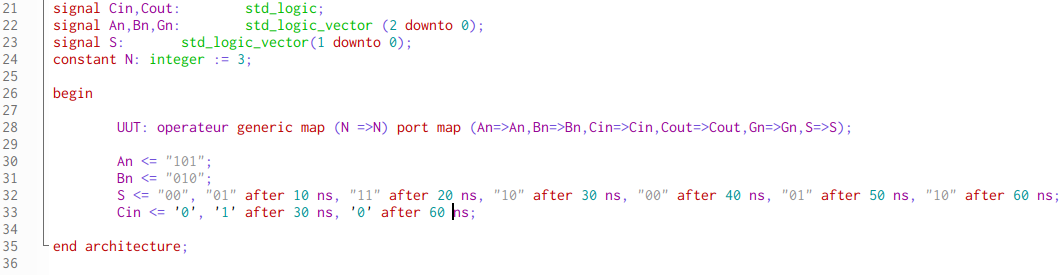
010 → G = A + B = 9+2 = 11

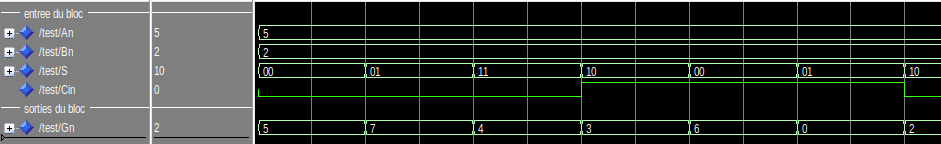
110 → G = A – 1 = 9-1 = 8

101 → G = A – B = 9-2 = 7

001 → G = A + 1 = 9+1 = 10

Nous avons ensuite refait un test avec N = 3 cellules.





Nous avons cette fois-ci fait toutes les opérations possibles. Ainsi, on peut vérifier que ça marche :

S1S0Cin=

000 → G = A = 5

010 → G = A + B = 5+2 = 7

110 → G = A – 1 = 5-1 = 4

101 → G = A – B = 5-2 = 3

001 → G = A + 1 = 5+1 = 6

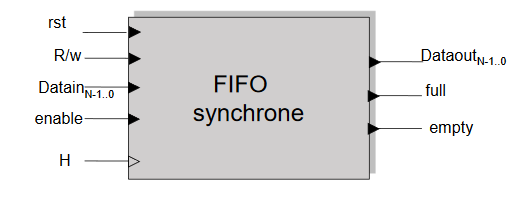
011 → G = A + B + 1 = 5+2+1 = 8

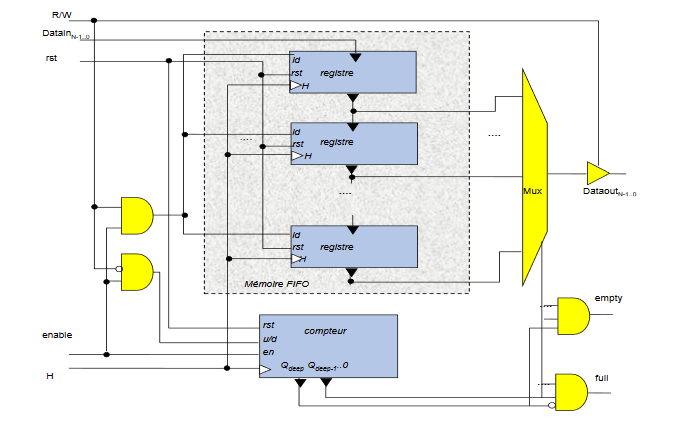
100 → G = A + not(B) = A + compl1 B = 101 + 101 = 5+5 = 10

Nous remarquons que lorsque l’on fait A+B+1 (=8), on a un dépassement car 8 est codé sur 4 bits. De même lorsque l’on fait A+not(B) (=10=8+2). Ainsi, les résultats de ces opérations sont faussés.

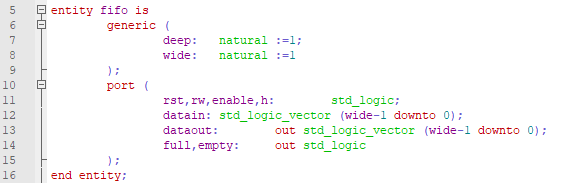
# FIFO

On souhaite décrire et tester le fonctionnement d’un circuit de stockage de données de type FIFO comme décrit ci-dessous. La taille des données ainsi que le nombre d’éléments de la file seront paramétrables respectivement par *wide* et *deep*.

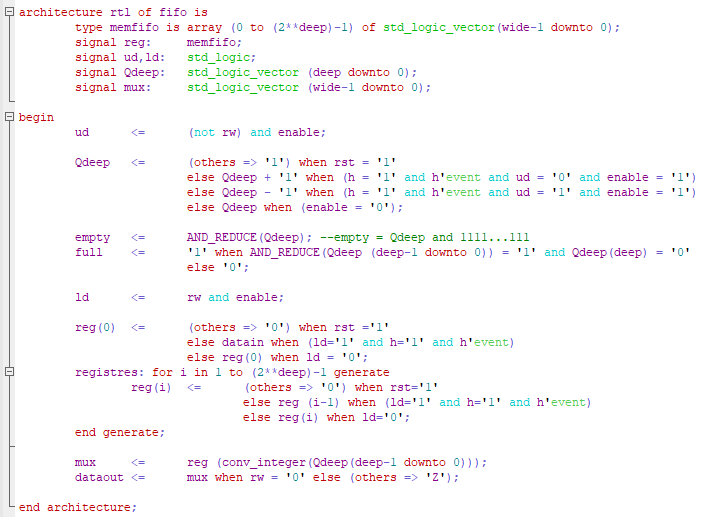




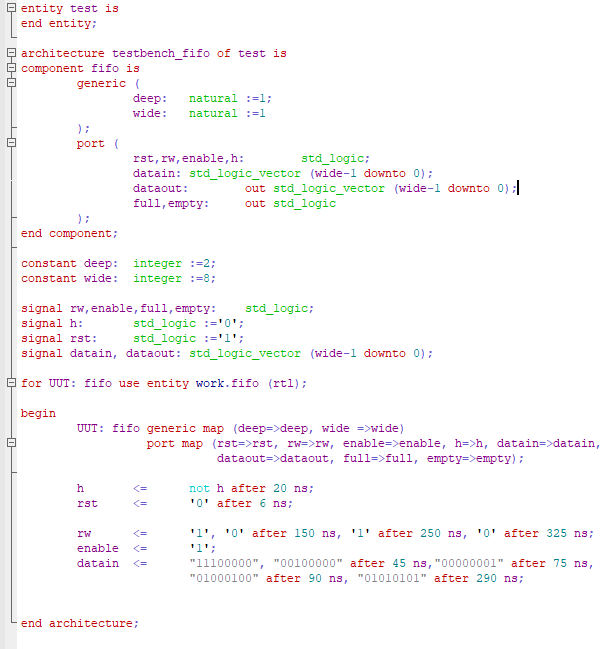
Nous commençons par définir l’entité de la FIFO :



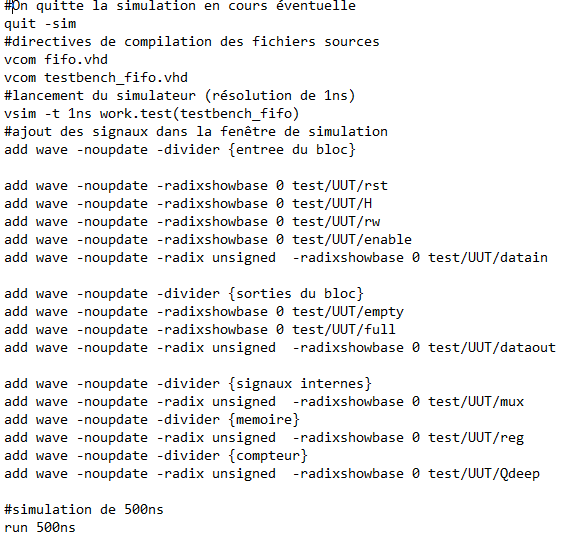
Puis nous définissons l’architecture :



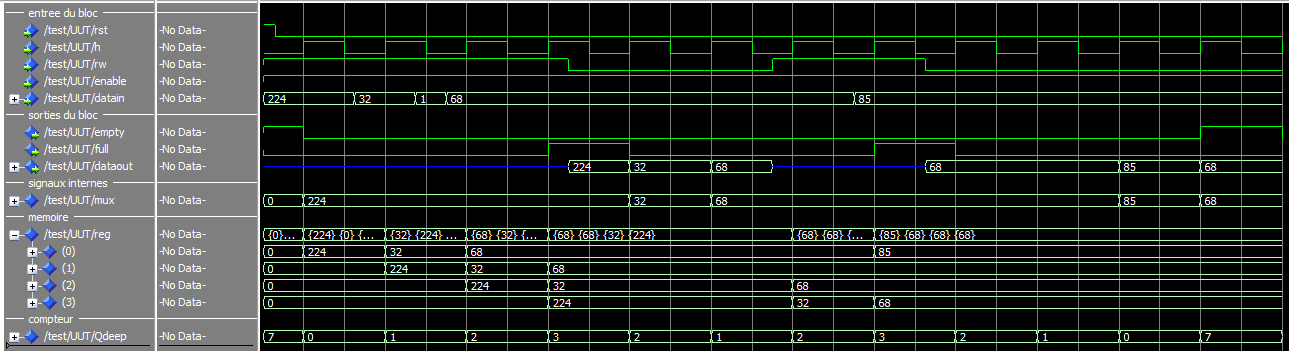
Nous implémentons ensuite un testbench à partir du modèle fournit dans l’énoncé.



Pour effectuer la simulation, nous utilisons le fichier tcl suivant.

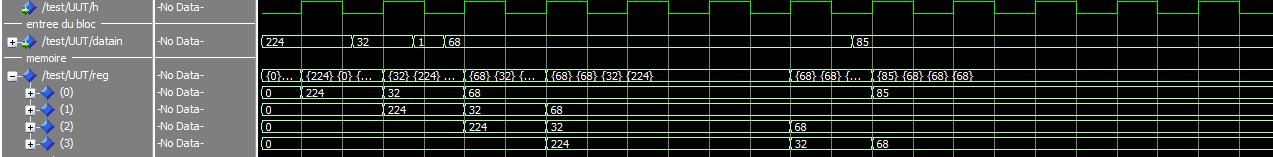


Et nous obtenons le chronogramme suivant.



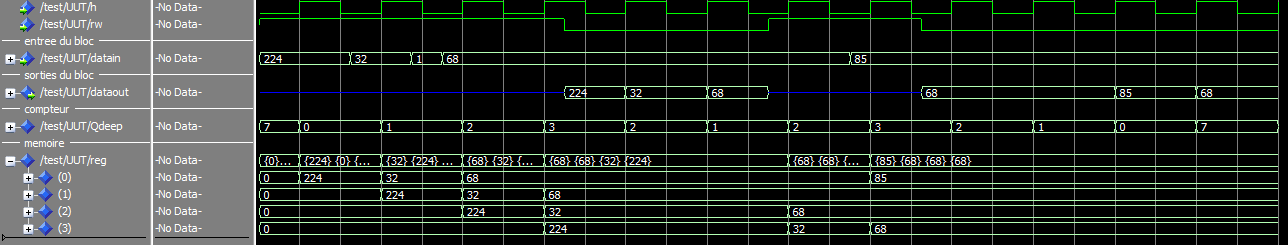
On remarque que les signaux de contrôle *empty* et *full* réagissent correctement aux entrées.

Voici la comparaison entre *datain* et le registre pour vérifier que les valeurs s’empilent bien.



Nous pouvons voir qu’à chaque coup d’horloge, la valeur de *datain* est bien empilée dans reg.

Pour vérifier que la bonne valeur est renvoyée lors du dépilage, voici la comparaison entre *datain, dataout, rw, Qdeep* et *reg*.



On remarque que lorsque *rw* passe à ‘0’, une valeur est écrite sur *dataout*. Cette valeur bien est récupéré dans *reg* par l’indice *Qdeep*.

*Qdeep* est bien à ‘-1’ lorsque la pile est vide, va s’incrémenter lorsqu’on empile une valeur et se décrémenter lorsqu’on dépile une valeur.

Le fonctionnement global de la FIFO est bien vérifié.

# Conclusion

Au travers de la réalisation d’une unité arithmétique et d’une FIFO, nous avons appris à utiliser la généricité en VHDL. Cette technique est très pratique car elle permet de rendre le code modulaire et rend les modifications beaucoup plus faciles. Par exemple, si nous avons besoin d’une FIFO dans un autre projet, nous pouvons réutiliser notre code en ne modifiant que quelques variables pour l’adapter au projet.