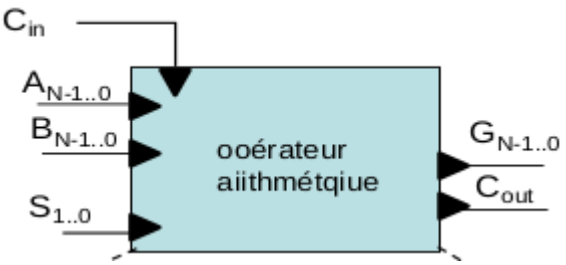
COMPTE RENDU TP4 GENERICITE

OPERATEUR ARITHMETIQUE

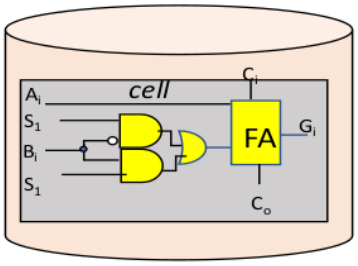
Dans cette première partie, nous allons décrire un opérateur arithmétique de taille paramétrable, qui sera capable de réaliser huit opérations différentes :

Table fonctionnelle

S_1	S_0	C_{in}	Opération
0	0	0	G = A (transfert)
0	0	1	G = A + 1 (inc)
0	1	0	G = A + B (add)
0	1	1	G = A + B +1 (add+1)
1	0	0	G = A + $\overline{B}$ (A+compl1 B)
1	0	1	G = A + $\overline{B}$ +1 (soust.)
1	1	0	G = A - 1 (dec)
1	1	1	G = A (transfert)



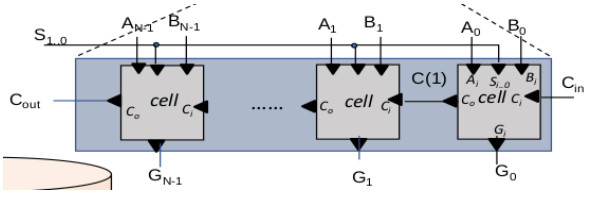
Pour se faire, nous avons dans un premier temps décrit un couple entité/architecture qui correspond à la description de la cellule élémentaire, comme on peut le voir ci-dessous.




```
1   library ieee;
2           use ieee.std_logic_1164.all;
3           use ieee.std_logic_unsigned.all;
4
5   entity cell is
6           port (
7                   A,B,Cin:        std_logic;
8                   S:      std_logic_vector (1 downto 0);
9                   G,Cout: out std_logic
10          );
11  end entity;
12
13  architecture cell_dataflow of cell is
14          signal Badd:    std_logic;
15          signal Gint:    std_logic_vector(1 downto 0);
16  begin
17          Badd    <=      (S(0) and B) or (S(1) and not B);
18          Gint    <=      '0' & A + Badd + Cin;
19          G       <=      Gint(0);
20          Cout    <=      Gint(1);
21  end architecture;
```

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 :



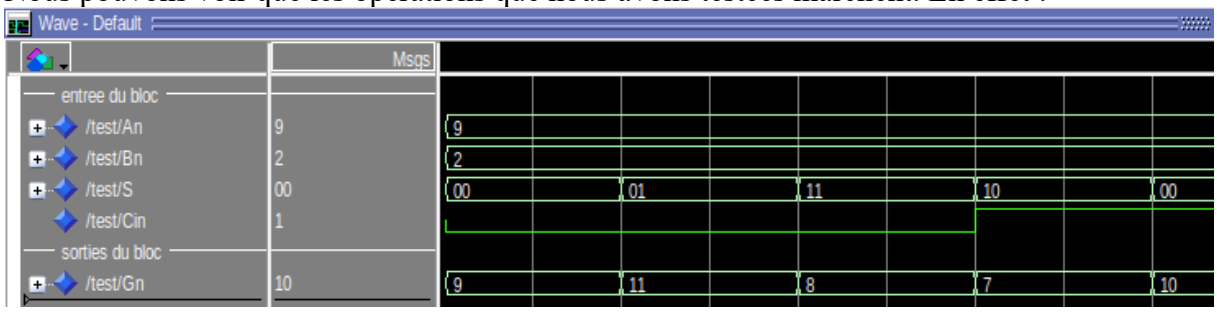
Ainsi, afin d'assembler les N cellules, on créer la cellule 0 puis on fait une boucle for qui va de 1 jusqu'à N-1, pour créer les autres cellules.

```
1   library ieee;
2           use ieee.std_logic_1164.all;
3           use ieee.std_logic_unsigned.all;
4
5   entity operateur is
6           generic (N:     natural :=8);
7           port(
8                   Cin:    std_logic;
9                   An,Bn:  std_logic_vector(N-1 downto 0);
10                  S:      std_logic_vector (1 downto 0);
11
12                  Cout:   out std_logic;
13                  Gn:     out std_logic_vector (N-1 downto 0)
14          );
15  end entity;
16
17  architecture operateur_dataflow of operateur is
18
19  component cell is
20          port (
21                  A,B,Cin:        std_logic;
22                  S:      std_logic_vector (1 downto 0);
23                  G,Cout: out std_logic
24          );
25  end component;
26
27          signal C:       std_logic_vector(N-1 downto 0);
28  begin
29          cell0: cell port map (A=>An(0), B=>Bn(0), Cin=>Cin, S=>S, G=>Gn(0), Cout=>C(0));
30          gene_cells: for i in 1 to N-1 generate
31                  celli: cell port map (A=>An(i), B=>Bn(i), Cin=>C(i-1), S=>S, G=>Gn(i), Cout=>C(i));
32          end generate;
33
34          Cout <= C(N-1);
35
36  end architecture;
37
```

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

```
1   library ieee;
2           use ieee.std_logic_1164.all;
3           use ieee.std_logic_unsigned.all;
4
5   entity test is
6   end entity;
7
8   architecture testbench of test is
9   component operateur is
10          generic (N:     natural :=8);
11          port(
12                  Cin:    std_logic;
13                  An,Bn:  std_logic_vector(N-1 downto 0);
14                  S:      std_logic_vector (1 downto 0);
15
16                  Cout:   out std_logic;
17                  Gn:     out std_logic_vector (N-1 downto 0)
18          );
19  end component;
20
21  signal Cin,Cout:        std_logic;
22  signal An,Bn,Gn:        std_logic_vector (7 downto 0);
23  signal S:       std_logic_vector(1 downto 0);
24  constant N: integer := 8;
25
26  begin
27
28          UUT: operateur generic map (N =>N) port map (An=>An,Bn=>Bn,Cin=>Cin,Cout=>Cout,Gn=>Gn,S=>S);
29
30          An <= "00001001";
31          Bn <= "00000010";
32          S <= "00", "01" after 10 ns, "11" after 20 ns, "10" after 30 ns, "00" after 40 ns;
33          Cin <= '0', '1' after 30 ns;
34
35  end architecture;
36
```

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



$S_1S_0C_{in}$= 000 → G = A = 9

010 → G = A + B = 9+2 = 11

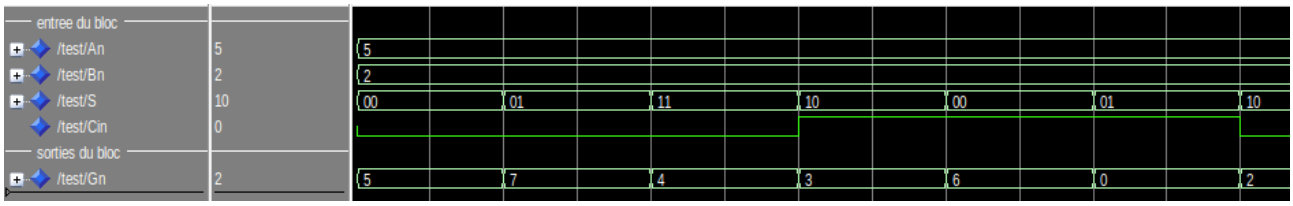
110 → G = A – 1 = 9-1 = 8

101 → G = A – B = 9-2 = 7

001 → G = A + 1 = 9+1 = 10

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

```
21  signal Cin,Cout:        std_logic;
22  signal An,Bn,Gn:        std_logic_vector (2 downto 0);
23  signal S:       std_logic_vector(1 downto 0);
24  constant N: integer := 3;
25
26  begin
27
28          UUT: operateur generic map (N =>N) port map (An=>An,Bn=>Bn,Cin=>Cin,Cout=>Cout,Gn=>Gn,S=>S);
29
30          An <= "101";
31          Bn <= "010";
32          S <= "00", "01" after 10 ns, "11" after 20 ns, "10" after 30 ns, "00" after 40 ns, "01" after 50 ns, "10" after 60 ns;
33          Cin <= '0', '1' after 30 ns, '0' after 60 ns;
34
35  end architecture;
36
```



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

$S_1S_0C_{in}$= 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

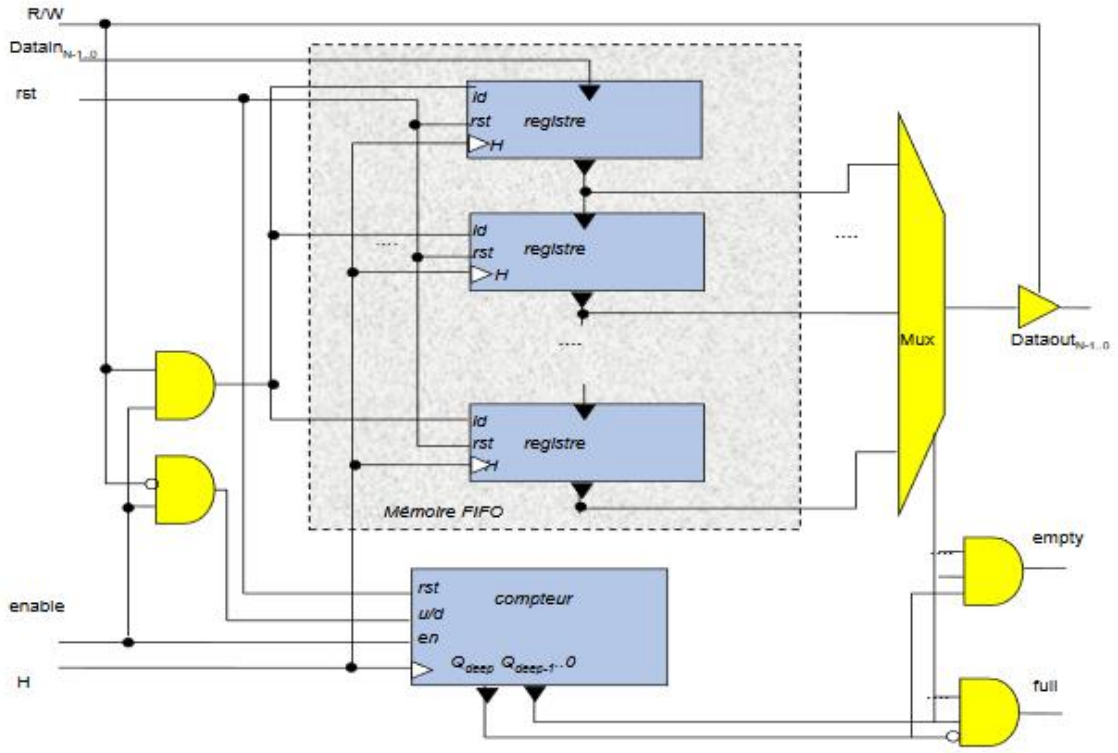
011 → G = A + B + 1 = 5+2+1 = 8

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 :

```
5   entity fifo is
6           generic (
7                   deep:   natural :=1;
8                   wide:   natural :=1
9           );
10          port (
11                  rst,rw,enable,h:        std_logic;
12                  datain: std_logic_vector (wide-1 downto 0);
13                  dataout:        out std_logic_vector (wide-1 downto 0);
14                  full,empty:     out std_logic
15          );
16  end entity;
```

Puis nous définissons l'architecture :

```
architecture rtl of fifo is
        type memfifo is array (0 to (2**deep)-1) of std_logic_vector(wide-1 downto 0);
        signal reg:     memfifo;
        signal ud,ld:   std_logic;
        signal Qdeep:   std_logic_vector (deep downto 0);
        signal mux:     std_logic_vector (wide-1 downto 0);

begin
        ud      <=      (not rw) and enable;

        Qdeep   <=      (others => '1') when rst = '1'
                        else Qdeep + '1' when (h = '1' and h'event and ud = '0' and enable = '1')
                        else Qdeep - '1' when (h = '1' and h'event and ud = '1' and enable = '1')
                        else Qdeep when (enable = '0');

        empty   <=      AND_REDUCE(Qdeep); --empty = Qdeep and 1111...111
        full    <=      '1' when AND_REDUCE(Qdeep (deep-1 downto 0)) = '1' and Qdeep(deep) = '0'
                        else '0';

        ld      <=      rw and enable;

        reg(0)  <=      (others => '0') when rst ='1'
                        else datain when (ld='1' and h='1' and h'event)
                        else reg(0) when ld = '0';
        registres: for i in 1 to (2**deep)-1 generate
                reg(i)  <=      (others => '0') when rst='1'
                                else reg (i-1) when (ld='1' and h='1' and h'event)
                                else reg(i) when ld='0';
        end generate;

        mux     <=      reg (conv_integer(Qdeep(deep-1 downto 0)));
        dataout <=      mux when rw = '0' else (others => 'Z');

end architecture;
```

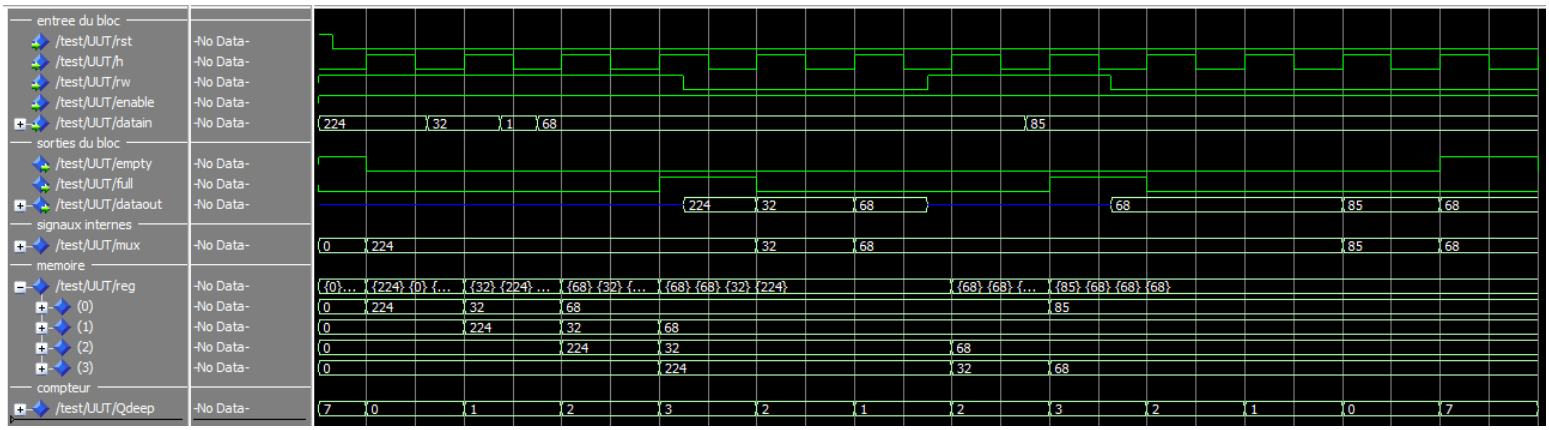
Nous implémentons ensuite un testbench à partir du modèle fournit dans l'énoncé.

```
entity test is
end entity;

architecture testbench_fifo of test is
component fifo is
        generic (
                deep:   natural :=1;
                wide:   natural :=1
        );
        port (
                rst,rw,enable,h:        std_logic;
                datain: std_logic_vector (wide-1 downto 0);
                dataout:        out std_logic_vector (wide-1 downto 0);
                full,empty:     out std_logic
        );
end component;

constant deep:  integer :=2;
constant wide:  integer :=8;

signal rw,enable,full,empty:    std_logic;
signal h:       std_logic :='0';
signal rst:     std_logic :='1';
signal datain, dataout: std_logic_vector (wide-1 downto 0);

for UUT: fifo use entity work.fifo (rtl);

begin
        UUT: fifo generic map (deep=>deep, wide =>wide)
                  port map (rst=>rst, rw=>rw, enable=>enable, h=>h, datain=>datain,
                            dataout=>dataout, full=>full, empty=>empty);

        h       <=      not h after 20 ns;
        rst     <=      '0' after 6 ns;

        rw      <=      '1', '0' after 150 ns, '1' after 250 ns, '0' after 325 ns;
        enable  <=      '1';
        datain  <=      "11100000", "00100000" after 45 ns,"00000001" after 75 ns,
                        "01000100" after 90 ns, "01010101" after 290 ns;


end architecture;
```

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

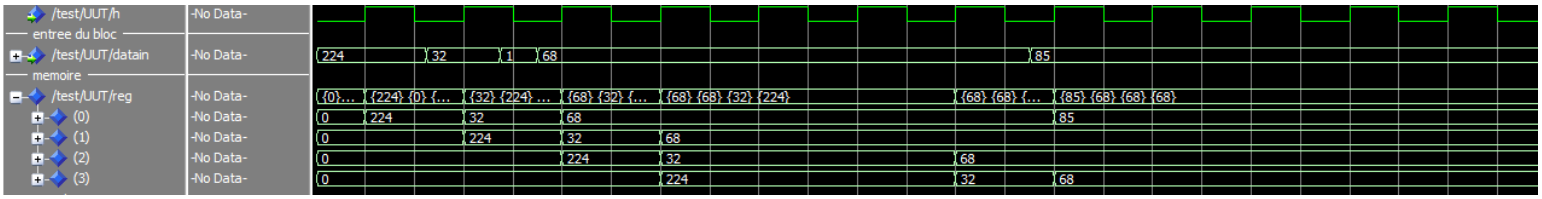
```
#On quitte la simulation en cours éventuelle
quit -sim
#directives de compilation des fichiers sources
vcom fifo.vhd
vcom testbench_fifo.vhd
#lancement du simulateur (résolution de 1ns)
vsim -t 1ns work.test(testbench_fifo)
#ajout des signaux dans la fenêtre de simulation
add wave -noupdate -divider {entree du bloc}

add wave -noupdate -radixshowbase 0 test/UUT/rst
add wave -noupdate -radixshowbase 0 test/UUT/H
add wave -noupdate -radixshowbase 0 test/UUT/rw
add wave -noupdate -radixshowbase 0 test/UUT/enable
add wave -noupdate -radix unsigned  -radixshowbase 0 test/UUT/datain

add wave -noupdate -divider {sorties du bloc}
add wave -noupdate -radixshowbase 0 test/UUT/empty
add wave -noupdate -radixshowbase 0 test/UUT/full
add wave -noupdate -radix unsigned  -radixshowbase 0 test/UUT/dataout

add wave -noupdate -divider {signaux internes}
add wave -noupdate -radix unsigned  -radixshowbase 0 test/UUT/mux
add wave -noupdate -divider {memoire}
add wave -noupdate -radix unsigned  -radixshowbase 0 test/UUT/reg
add wave -noupdate -divider {compteur}
add wave -noupdate -radix unsigned  -radixshowbase 0 test/UUT/Qdeep

#simulation de 500ns
run 500ns
```

Et nous obtenons le chronogramme suivant.



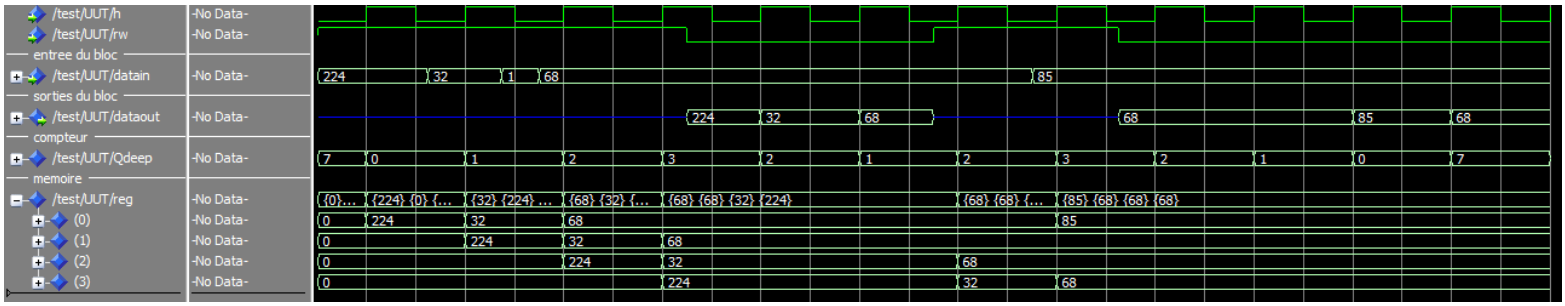
On remarque que les signaux de contrôle *empty* et *full* réagissent correctement aux entrées.

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.