FORMALISMES DE DESCRIPTION DE L'ARCHITECTURE

I. INTRODUCTION

Il faut signaler dès maintenant qu'il s'agit là de descriptions de types combinatoires c'est-à-dire qui ne font pas intervenir le temps. Ce dernier sera considéré dans une étape bien avancée. Concernant le premier type (combinatoire), trois grands formalismes peuvent coexistés pour décrire les architectures **:**

- FLOT DE DONNEES (DATAFLOW),
- STRUCTUREL : INTERCONNEXION DE COMPOSANTS
- COMPORTEMENTAL : ALGORITHMES

Il s'agit, dans ce qui suit, de décrire chacune de ces descriptions.

II. DESCRIPTION EN FLOT DE DONNEES (DATAFLOW) :

Dans cette technique, on décrit simplement les équations booléennes que l'on veut implémenter. Les instructions qu'on peut rencontrer dans ce type de description sont au nombre de trois:

- **... <= ...** -- concurrent affectation
- **...** <= ... when ... else ... -- concurrent selective affectation
- with ... select ... <= ... when ... -- concurrent selective affectation

☞ A réserver aux plus petits circuits pour des raisons de lisibilité

Certains exemples donnés ci-dessous sont déjà traités dans les paragraphes précédents, cependant pour mettre les choses en place, ces exemples sont repris dans cette partie qui leurs convient bien.

Exemple 1: Half Adder

Un demi-additionneur est un composant électronique qui prend deux bits en entrée (qu'il faut additionner) et en produit deux bits (le bit de somme somme et le bit de report carry).

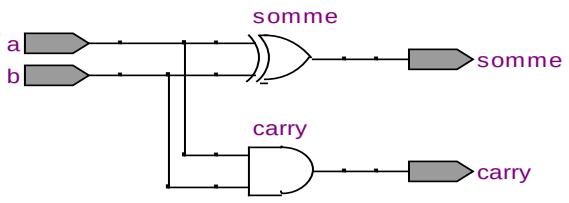


Figure IV.1: Schéma d'un demi-additionneur

La paire entity/architecture qui le dévrivent se présentent comme suit:

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity Half_add is                                  -- entity of the Half Adder.
    port(
        a, b            : in    std_logic;

        somme, carry    : out   std_logic
    );
end Half_add;

architecture Flow of Half_add is                    -- architecture of the Half Adder
begin
    somme   <=  a xor b;                            -- Description in Data Flow
    carry   <=  a and b;
end Flow;
```

Exemple 2: multiplexer 4-1 with logic Gates

Un multiplexeur est un sélectionneur d'entrées: en fonction de la valeur placée sur les entrées de sélection, il choisit l'entrée de données à répercuter à la sortie. Ici, on implémente un multiplexeur 4-1 : il dispose de quatre entrées de données (E0, E1, E2, E3) , d'une entrée de sélection (à deux bits) **SEL** et d'une sortie **S**. La table de vérité de ce multiplexeur est donnée ci-dessous :

Tableau IV.1: Sélection-Entrées/Sortie d'un Multiplexeur 4/1

SEL		S
0	0	E0
0	1	E1
1	0	E2
1	1	E3

L'équation logique qui décrit le fonctionnement de ce multiplexeur est :

$$S = \overline{SEL(0)}\,\overline{SEL(1)}\,E0 + SEL(0)\,\overline{SEL(1)}\,E1 + \overline{SEL(0)}\,SEL(1)\,E2 + SEL(0)\,SEL(1)\,E3$$

Cette équation est traduite telle quelle en code VHDL comme suit.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity MUX is
    port(
        E0, E1, E2, E3  : in    std_logic;

        Sel             : in    std_logic_vector(1 downto 0);

        S               : out   std_logic
    );
end MUX;

architecture Arch_Flow of MUX is
begin

    S <= (not Sel(0) and not Sel(1) and E0) or (Sel(0) and not Sel(1) and E1) or
```

```
    (not Sel(0) and Sel(1)and E2) or (Sel(0) and Sel(1) and E3);
end Arch_Flow;
```

Dont la synthèse produit le schéma ci-dessous.

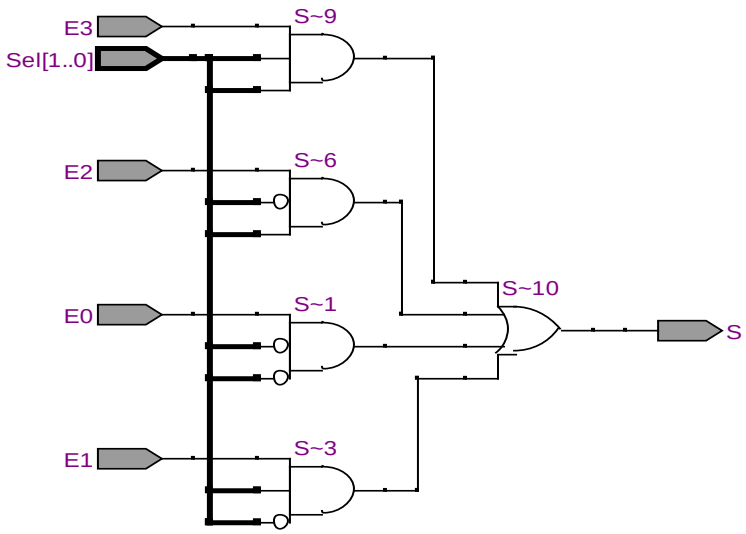


Figure IV.2: Synthèse de la description en flow de données d'une architecture

Notez bien la traduction fidèle des équations dans ce circuit.

III. DESCRIPTION COMPORTEMENTAL OU EVENEMENTIELLE (BEHAVIORAL)

Dans cette technique le process est la pièce maitresse. De manière très semblable à un langage de programmation informatique, on précise, dans un process, le fonctionnement voulu à l'aide d'une suite d'instructions de contrôles plus ou moins évoluées (conditions, boucles, etc.).

On définit des process ainsi que leurs listes de sensibilité. Dès qu'un des paramètres de la liste de sensibilité change de valeur, le code est ré exécuté. C'est utile notamment en logique séquentielle, pour lancer le code à chaque coup de l'horloge.

A l'intérieur de ces process, on peut utiliser diverses constructions bien connues: le if-then-else, le case is when, etc. Les premiers (les signaux) ne peuvent être déclarés dans un process alors que les seconds (les variables) le peuvent. On dispose également de plusieurs boucles, qui ne peuvent cependant manipuler que des variables.

En dehors d'un process, tout se produit de manière concurrente: deux process peuvent être exécutés simultanément. Par contre, à l'intérieur de celui ci, toutes les instructions sont exécutées de manière séquentielle mais les affectations des signaux ne sont faites qu'à la fin ou à la rencotre d'une instruction d'attente (wait).

Tout est bati donc autour d'un process, et il est tout naturel de s'intéresser tout particulièrement à cette outil.

III.1. LES PROCESSUS : process

Un processus (process) est une séquence d'instructions séquentielles, c'est-à-dire exécutées dans l'ordre spécifié. La déclaration de processus délimite un domaine séquentiel de l'architecture dans laquelle la déclaration apparaît.

Un process peut apparaître n'importe où dans un corps d'architecture (après le mot-clé begin). Il peut être explicite, c'est-à-dire créé avec l'instruction process, ou implicite (instructions concurrentes).

S'il est explicite alors :

- Il est cyclique
- Il dispose d'une liste de signaux auxquels il est sensible: liste de sensibilité
- Sa durée de vie est égale au temps de simulation : il ne se termine jamais
- Il peut s'endormir par utilisation de l'instruction wait qui possède plusieurs formes. Le process s'exécute au moins une fois jusqu'à la rencontre de cette instruction, puis une fois validée la condition de réveil, l'exécution continue de façon séquentielle et cyclique (après la fin, le début).

Nous l'avons déjà signalé, il y a deux types de processus :

III.1.1. PROCESSUS IMPLICITE

Toute phrase concurrente est un processus implicite avec un wait en implicite sur les signaux placés à droite de l'affectation.

Exemples

☞ L'affectation inconditionnelle d'un signal : `S<=A+B`

☞ L'affectation conditionnelle d'un signal (équivalent d'un case) :

```
S <= A when select = 1 else
     B when select = 2 else
     "0000";
```

III.1.2. PROCESSUS EXPLICITE

Un process explicite est un bloc séquentiel qui prend place obligatoirement dans le corps d'une architecture ;

La déclaration du process est contenue entre le mot clé process et le mot clé end process. Une étiquette (label) peut être attribuée à un process en vue d'une identification plus simple de

celui ci dans le code source. Elle peut être répétée à la fin de la déclaration, après les mots-clés end process.

Un process, comme l'architecture, comporte une zone déclarative et un corps. Trois syntaxes peuvent être utilisées pour la création d'un processus :

Syntaxe 1

```
Proc: process(sensitivity list)
        Zone déclarative. La portée du contenu de cette zone est limitée au process lui-même c'est-à-dire que
                          toutes les déclarations faites ici sont locales.
bigin
    Corps : description séquentiel
end process Proc ;
```

où :

- **Proc** est une étiquette : optionnelle
- Les paramètres entre parenthèse sont appelés la liste de sensibilité (sensitivity list)

Syntaxe 2

```
Proc : process
        Zone déclarative.
bigin
        Corps : description séquentiel
        wait on liste des signaux ;                    -- for example: wait on a, b
end process Proc ;
```

où :

- **Proc** : est une étiquette : optionnelle
- Par rapport à la première syntaxe, la liste de sensibilité est positionnée ici devant l'instruction wait à la fin du process et elle joue le même rôle.

➧ Cette syntaxe est couramment employée

Syntaxe 3

```
Proc: process
        Zone déclarative.
bigin
        wait on liste des signaux ;                    -- for example: wait on a, b
        Corps : description séquentiel
end process Proc ;
```

- Dans cette syntaxe, la liste de sensibilité est toujours placée devant l'instruction wait cependant l'instruction wait elle-même est placée tout à fait au début du process.

☞ La déclaration des signaux est interdite au sein d'un process. La déclaration des variables est cependant autorisée. Il s'agit bien des déclarations, l'utilisation est, cependant, autorisée pour les deux types.

Considérons le code VHDL ci-dessous, qui détermine le maximum et le minimum entre trois paramètres.

```
library ieee;
use ieee.std_logic_1164.all;

entity Main_Min_Max is
    port(
        a, b, c      : in  integer range 0 to 255;
        Max, Min  : out integer range 0 to 255
    );
end Main_Min_Max;

architecture Arch of Main_Min_Max is

    signal S_Min, S_Max  : integer range 0 to 255;

begin
    process(a, b, c)
    begin

        if (a>b) then
            S_Max <= a;
        else
            S_Max <= b;
        end if;

        if (S_Max>c) then
            Max <= S_Max;
        else
            Max <= c;
        end if;

        if (a<b) then
            S_Min <= a;
        else
            S_Min <= b;
        end if;

        if (S_Min<c) then
            Min <= S_Min;
        else
            Min <= c;
        end if;

    end process ;
end Arch ;
```

La synthèse de ce code génère le circuit ci dessous. Les signaux internes S_Min et S_Max, sont traduits, par le synthétiseur, par deux blocs S_Min et S_Max, utilisés par celui-ci pour maintenir les résultats intermédiaires des signaux exactement comme le prévoi le programme.

La simulation reflète le comportement du circuit. Elle est opérée pour une durée de 10 secondes et le circuit se comporte bien comme prévu. Pour chaque tranche du temps (slice), les sorties Min et Max recoivent bien chacune la bonne valeur parmi les trois entrées a, b et c.

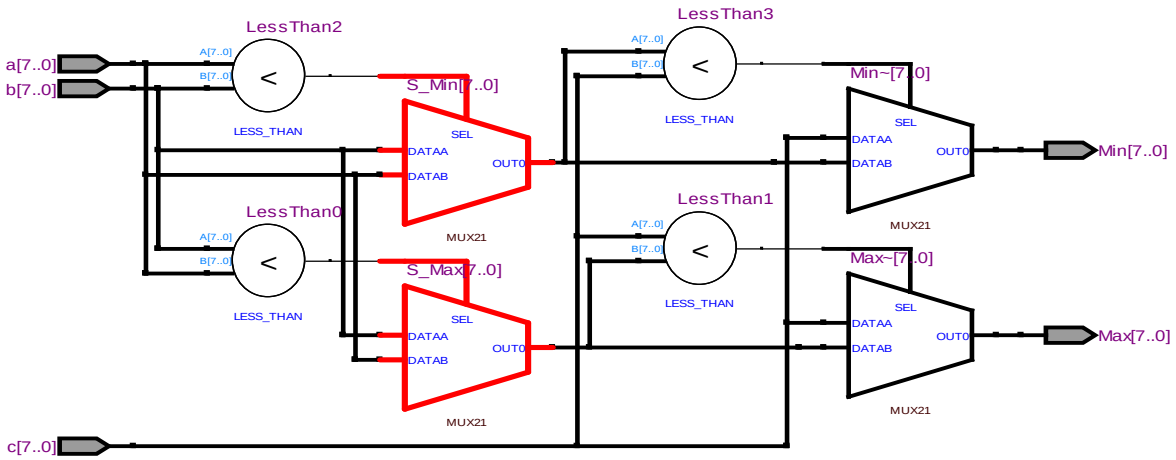


Figure IV.3: Comportement de plusieurs signaux au sein d'un process

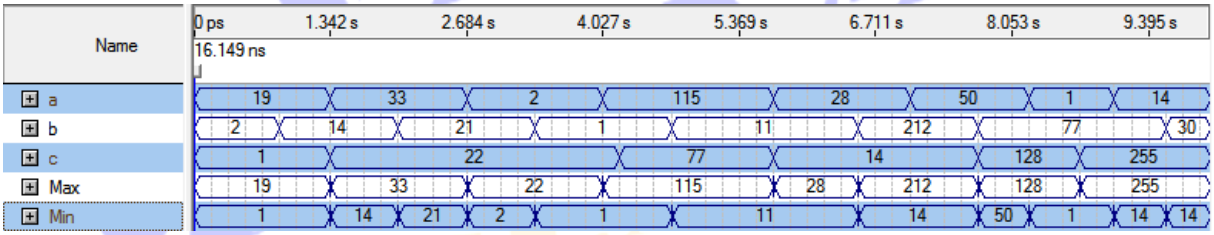


Figure IV.4: Simulation du comportement de plusieurs signaux au sein d'un process

- La synthèse aurait conduit au même circuit avec le même comportement si au lieu des signaux on avait utilisé des variables. *Bien sûr ces variables doivent être déclarées dans la partie déclarative du process*.

III.1.2.1. COMPORTEMENT D'UN PROCESS

La durée de vie d'un process est égale au temps de simulation: Un process est l'équivalent d'une boucle infinie que l'on peut interrompre (mettre en sommeil) puis redémarer (réveiller).

Le process est réveillé lorsque se produisent des évènements particuliers sur certains signaux: l'ensemble des ces signaux est appelé liste de sensibilité. Cette liste est placée, entre parenthèses, à la fin de l'entête du processus: *L'exécution d'un process n'a lieu qu'en cas de*

changement d'état de l'un (ou de plusieurs) signal (aux) compris dans cette liste de sensibilité;

- Lorsque la liste de sensibilité est manquante, le processus sera exécuté en continu. Dans ce cas, le process doit contenir une déclaration d'attente wait pour le suspendre et l'activer lorsque survient un événement ou une condition devient vraie.
- Lorsque la liste de sensibilité est présente, le processus ne peut pas contenir d'instruction d'attente wait.

III.1.2.2. CORPS D'UN PROCESSUS EXPLICITE : *instruction séquentielles*

Le corps d'un process est l'assemblage d'instructions séquentielles, c'est-à-dire qui s'exécute l'une après l'autre. Ces instructions peuvent être :

- <=, := : assignation de signal ou de variable
- if ... then ... else ... : execution conditionnelle
- case ... when ... : execution sélective
- for ... loop ... : boucle for
- while ... loop ... : boucle while

Bien que ces instructions s'exécutent de façon séquentielle l'une apès l'autre, le comportement d'assignement est bien différent entre les signaux et les variables:

- Les variables sont instatanément assignées de leurs nouvelles valeurs : Si X et Y sont deux variables alors :

```
X := expression d'assignement de X;
Y := expression d'assignement de Y;
.....
X := nouvelle expression d'assignement de X;
Y := nouvelle expression d'assignement de Y;
```

Cette écriture est bien autorisée et se comporte exactement comme on s y attendait avec un langage de programmation classique tel le C, c'est-à-dire qu'à l'instant « t » la variable X va prendre l'«expression d'assignement de X» et à l'instant suivant la varaible Y va prendre l'«expression d'assignement de Y». Bien plus tard, la variable X va être mise à jour et prendra la «nouvelle expression d'assignement de X» et à l'instant suivant la varaible Y va également être mise à jour et prendra la «nouvelle expression d'assignement de Y». A retenir donc que les variables sont assignée *immédiatement* mais qu'elles *ne sont pas visibles de l'extérieur du process;*

Voyons maintenant le comportement de ces assignements si au lieu des varaibles on a affaire à des signaux :

- Dans un ensemble d'assignements successives, à l'interieur du même process, la mise à jour des signaux n'est opéré qu'à la fin du process ou à la rencontre d'une instruction wait ;
- Si plusieurs assignement concernent le même signal, c'est uniquement le dernier assignement qui sera considéré. C'est-à-dire dans l'exemple ci-dessus, si X et Y représentaient des signaux, se sont uniquement les deux derniers assignements, X <= nouvelle expression d'assignement de X et Y <= nouvelle expression d'assignement de Y; qui seront considérés. Les deux premiers assignements sont ignorés par le synthétiseur.

Les instructions de contrôle et de boucles, mentionnées ci-dessus, ont fait l'objet de développemets bien étendus dans les chapitres précédents. Maintenant alors qu'on s'intérèsse aux process, il est tout à fait ligitime de considérer de nouveau ces instructions pour batir nos process.

Exemple 1: Multiplexeur comportemental avec l'instruction if

On peut implémenter un multiplexeur à l'aide d'une série de if, une manière très commode de procéder pour tout ce qui vient de la programmation classique.

```
library ieee;
use ieee.std_logic_1164.all;

entity Mux is
    port(
        E0, E1  : in    std_logic;
        E2, E3  : in    std_logic;
        Sel     : in    std_logic_vector(1 downto 0);
        S       : out   std_logic
    );
end Mux;

architecture Arch of Mux is
begin
    process (E0, E1, E2, E3, Sel)
    begin
        if   Sel="00"    then
            S <= E0;
        elsif Sel="01" then
            S <= E1;
        elsif Sel="10" then
            S <= E2;
        elsif SEL="11" then
            S <= E3;
        else
            S <= '-';
        end if;
    end process;
end Arch;
```

Exemple 2: behavioral Multiplexer with the instruction case

On peut y préférer un case pour des raisons de lisibilité.

```
library ieee;
use ieee.std_logic_1164.all;

    entity Mux is
        port(
            E0, E1, E2, E3  : in    std_logic;
            Sel             : in    std_logic_vector(1 downto 0);
            Output          : out   std_logic
        );
    end Mux;

    architecture Arch of Mux is
    begin
        process (E0, E1, E2, E3, Sel)
        begin
            case Sel is
                when "00" => Output <= E0;
                when "01" => Output <= E1;
                when "10" => Output <= E2;
                when "11" => Output <= E3;
            end case;
        end process;
    end Arch;
```

III.1.2.3. PROCESSES AVEC UNE LISTE DE SENSIBILITE INCOMPLETE

La majorité des synthétiseurs peuvent ne pas vérifier les listes de sensibilité des processus. Ces outils supposent que tous les signaux, du côté droit des assignations séquentielles de signaux, figurent dans la liste de sensibilité. Ce comportement est bien logique parceque la présence d'un signal dans le coté droit d'un opérateur d'assignement (<=) signifie une évaluation du coté droit et un assignement au coté gauche. Même si le signal qui figure dans la partie droite de l'opérateur <= ne fait pas partie de la liste de sensibilité du process, le bon sens dit que pour tout changement de ce signal il faut évaluer la partie droite est opérer un assignement au coté gauche, c'est-à-dire que le process doit opérer cette évaluation pour tout changement du signal en question, même si ce signal ne figure pas dans sa liste de sensibilité. Autrement dit le process devrait se comporté comme si le signal en question fait partie de sa liste de sensibilité. Ainsi, ces outils de synthèse vont interpréter les deux processus de l'exemple ci-dessous comme identiques.

```
library ieee;
use ieee.std_logic_1164.all;

entity Main_Min_Max is
    port(
      a, b, c    : in    std_logic_vector(3 downto 0);
      output     : out   std_logic_vector(3 downto 0)
```

```
library ieee;
use ieee.std_logic_1164.all;

entity Main_Min_Max is
  port(
    a, b, c    : in    std_logic_vector(3 downto 0);
    output     : out   std_logic_vector(3 downto 0)
```

```
    );
end Main_Min_Max;

architecture arch of Main_Min_Max is
begin
    Proc1: process (a, b, c)
    begin
        output <= a and b and c;
    end process Proc1;
end arch;
```

```
  );
end Main_Min_Max;

architecture arch of Main_Min_Max is
begin
  Proc1:process (a, b)
  begin
    output <= a and b and c;
  end process Proc1;
end arch;
```

Tous les outils de synthèse interpréteront le processus Proc1 comme une porte ET à 3 entrées, mais interpréteront également le processus Proc2 comme une porte ET à 3 entrées. La synthèse de l'un comme l'autre de ces deux codes donne le cirduit suivant :

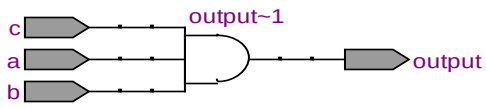


Figure IV.7: Synthèse de deux process avec deux listes de sensibilités

III.1.2.4. LE FEEDBACK

Une autre conséquence de la manière dont les affectations de signaux sont exécutées est que toute lecture d'un signal qui est déjà affecté (assigné) dans le même processs renvoie la valeur affectée au signal lors de l'exécution précédente du process.

- Lire un signal et lui attribuer une valeur dans le même process équivaut à une retro action (feedback) :
  - Dans un processus combinatoire, la valeur précédente est une sortie de la logique combinatoire et la rétroaction est donc asynchrone ;
  - Dans un processus séquentiel, la valeur précédente est la valeur stockée dans un verrou ou un registre, de sorte que la rétroaction est synchrone.

L'exemple ci dessous présente un compteur à 4 bits. La lecture/écriture du signal cout, dans le process (count <= count + "0001"), cré un feedback qui ramène l'ancienne valeur mémorisée dans un latch (bascule) pour le calcul de la nouvelle valeur. Le circuit résultant fait bien apparaître cette notion de feedback.

```
library ieee;
use ieee.std_logic_1164.all;

use ieee.std_logic_unsigned.all ;

entity Feadback is
    port (
        Clk     : in     bit;
        Reset   : in     bit;
        Count   : buffer  std_logic_vector(3 downto 0)
```

```
    );
end Feadback;

architecture Arch of Feadback is
begin
    process
    begin
        wait until Clk = '1';

        if (Reset = '1') then
            Count <= "0000";
        else
            Count <= Count + "0001";
        end if;
    end process;
end Arch;
```

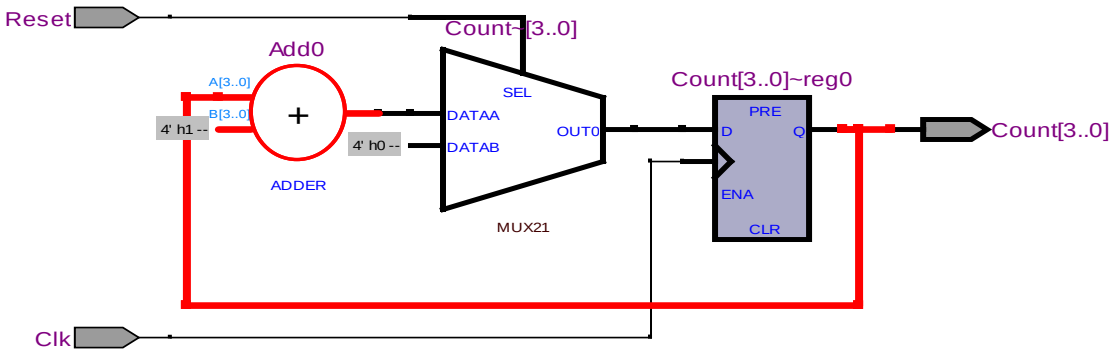


Figure IV.8: Synthèse d"un feedback

III.2. PROCESSUS SYNCHRONE ET PROCESSUS ASYNCHRONE

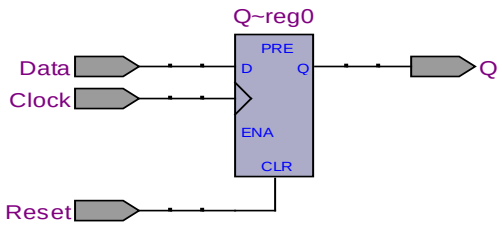
Si les instructions séquentielles d'un process sont synchronisées par un signal, on parle de process synchrone, sinon il s'agit d'un process asynchrone. Examinos les deux exemples suivants :

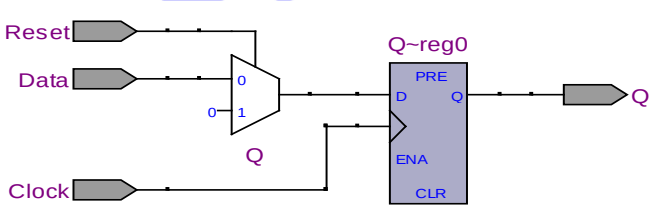
```
library ieee;
use ieee.std_logic_1164.all;

entity Asynchronous_DFF is
    port(
      Clock, Reset :  in   std_logic;

      Data         :  in   std_logic;

      Q            : out std_logic
    );
end Asynchronous_DFF;

architecture BHV of Asynchronous_DFF is
begin
    process(clock, Reset, data)
    begin
      if Reset = '1' then
        Q <= '0';
      elsif Rising_edge(Clock) then
        Q <= Data;
```

```
library ieee;
use ieee.std_logic_1164.all;

entity Synchronous_DFF is
    port(
      Clock, Reset :  in   std_logic;

      Data         :  in std_logic;

      Q            : out std_logic
    );
end Synchronous_DFF;

architecture BHV of Synchronous_DFF is
begin
      process(clock, Reset, data)
      begin
        if rising_edge(Clock) then
          if Reset = '1' then
            Q <= '0';
          else
```

```
    end if;
  end process;
end BHV;
```

```
      Q <= Data;
    end if;
  end if;
 end process;
end BHV;
```





Dans l'exemple de gauche, la bascule D est remise à zéro indépendemment de la présence ou l'absence de l'horloge Clock, on parle d'une remise à zéro asynchrone. Dans l'exemple de droite par contre, la remise à zéro de la bascule est conditionnée par la présence d'un front montant de l'horloge Clock, on parle alors de remise à zéro synchrone.

III.3. L'INSTRUCTION WAIT

Au lieu d'une liste de sensibilité, un processus peut contenir une instruction wait. L'utilisation de cette instruction a deux raisons:

1. Suspendre l'exécution d'un process;
2. Spécifier une condition qui déterminera l'activation du process suspendu.

Théoriqment, sans wait ou sans liste de sensibilité, un process se boucle indéfiniment, mais pratiquement tous les synthétiseurs refusent un process sans liste de sensibilté ou sans instruction wait et une erreur de compilation est générée en conséquence.

Lorsqu'une instruction wait est rencontrée, le processus, dans lequel apparaît cette instruction, est suspendu. Lorsque la condition spécifiée dans l'instruction wait est remplie, le processus reprend, et ses instructions sont exécutées jusqu'à ce qu'une autre instruction wait soit rencontrée.

☞ Le langage VHDL autorise plusieurs instructions wait dans un processus.

☞ Si un processus contient une instruction wait, il ne peut pas contenir de liste de sensibilité. Les deux processus ci-dessous sont exactement equivalents: Le premier, contenant une instruction d'attente explicite wait, est équivalent au deuxième, qui contient une liste de sensibilité.

```
Proc: process
begin
    x <= a and b and c;
wait on a, b, c;
```

⇔

```
Proc: process (a, b, c)
begin
  x <= a and b and c;
```

```
end process Proc;
```

```
end process Proc;
```

Il existe trois syntaxes différentes autour de wait.

III.3.1. L'INSTRUCTION wait on

Équivalent à une liste de sensibilité.

Syntaxe

```
Proc: process
begin
    -- Sequential statements
    wait on liste de signaux;
end process Proc;
```

Exemple

```
Proc: process
begin
    -- Sequential statements
wait on a,b,c;
end process Proc;
```

⇔

```
Proc: process (a,b,c)
begin
   -- Sequential statements
end process Proc;
```

III.3.2. L'INSTRUCTION wait until

Syntaxe

```
Proc: process
begin
    wait until expression;
        Instructions séquentielles;
end process Proc;
```

L'instruction wait until suspend le process jusqu'à ce que l'évaluation de expression devienne vraie, par le changement d'un ou plusieurs des signaux cités dans celle ci.

```
Proc: process
begin
   wait until clk='1';
     Q1 <= D1;
 end process Proc;
```

```
Proc: process
begin
    wait until clk'event and clk='1';
     Q1 <= D1;
end process Proc;
```

```
Proc: process
begin
    wait until not clk'stable and clk='1';
     Q1 <= D1;
 end process Proc;
```

III.3.3. L'INSTRUCTION wait for

Syntaxe

```
proc: process
begin
    Instructions séquentielles;
    wait  for durée;
        Instructions séquentielles;
```

```
end process proc;
```

Où « durée » spécifie la durée maximale pendant laquelle le process reste suspendu.

☞ L'instruction wait for n'est jamais synthétisable. Cependant, elle est toujours très utile pour les TestBench.

☞ wait for doit être suivie d'un certain temps, par exemple 10 ns.

Exemple

```
Proc: process
begin
    EN_1 <= '0';
    EN_2 <= '1';
    wait for 10 ns;
    EN_1 <= '1';
    EN_2 <= '0';
    wait for 10 ns;
    EN_1 <= '0';
    wait;
end process Proc;
```

Il est possible de combiner plusieurs conditions dans l'expression de l'instruction wait. Dans l'exemple ci dessous, le processus "Proc" sera activé lorsque l'un des signaux a ou b changera, mais uniquement lorsque la valeur du signal Clk est "1".

```
proc: process
begin
    wait on a, b until Clk = '1';
    ...
end process proc;
```

☞ L'instruction wait est interdite à l'intérieur d'une fonction (une fonction rend un résultat ***immédiat***, donc pas d'attente).

IV. DESCRIPTION STRUCTUREL: INTERCONNEXION DE COMPOSANTS

Dans cette technique, on décrit le circuit comme une série de boîtes noires, déjà décrites, pour former un circuit plus complexe : Chacune des ces boites noires est un composants;

Cette manière de procéder permet de créer des structures hiérarchiques: Au lieu de développer des programmes VHDL complexes, il est plus intéressant de découper toute l'architecture en un ensemble de petites structures (components) et combiner ces composants pour remonter vers la structure plus complexe.

Selon qu'on fasse ou non appel aux package, deux techniques sont possibles pour manipuler les composants dans un développement VHDL. Manipuler signifie que l'objet component concerné par la manipulation existe déjà, avant de parler manipulation voyons donc d'abord comment créer ces composants pour leur donner existance.

IV.1. CREATION DES COMPOSANTS

Par définition, un composant est aussi un système logique (un sous-système); à ce titre, il doit aussi être décrit par un couple entity/architecture dans lequel sont défini ses entrées/sorties et son comportement. La création d'un composant n'est donc autre que le développement de la paire Entity/Architecture qui le décrit, selon la syntaxe suivante :

Syntaxe

```
entity  entity_Identifier
   generic (generic list);
   Port(port list);
end entity_Identifier;

architecture architecture_Identifier of  entity_Identifier is
begin
    Instructions concurrentes;
end architecture_Identifier;
```

Laissons de coté, pour le moment, le terme generic et concentrons-nous sur la création des composants sans ce terme. Considérons l'exemple simple de la création d'un additionneur complet (Full Adder) selon le schéma ci-dessous.

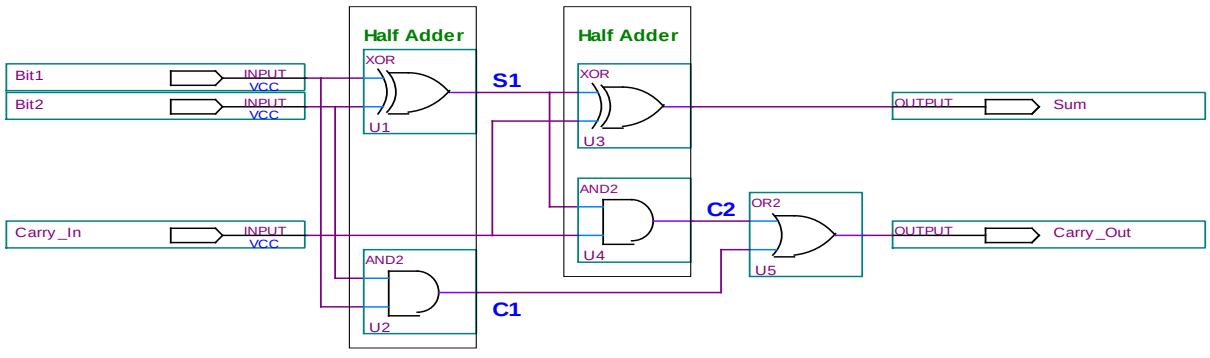


Figure IV.9: Composition d'un Full Adder à partir de Half Adder

La paire des portes (xor, and) constitue ce qu'on appelle un Demi Additionneur (Half Adder). Un Full Adder est donc composé de l'assemblage de deux Half Adder et d'une porte or. Nous nous proposons de créer un Full Adder en se servant du Half Adder comme composant de base. Développons d'abort la paire Entity/Architecture de ce dernier:

```
library ieee;
use ieee.std_logic_1164.all;

entity Half_Adder
    port (
        Bit1    : in   std_logic;
        Bit2    : in   std_logic;
        Sum     : out  std_logic
        Carry   : out  std_logic
    );
end;

architecture Arc of Half_Adder is
begin
    Sum    <= Bit1 xor  Bit2;
    Carry  <= Bit1 and Bit2;
end Arc;
```

La paire Entity/Architecture du composant Half_Adder est maintenant disponible, revenons maintenant aux technique de manipulations de ce copmposant. Il existe deux techniques pour opérer cette manipulation, selon qu'on fasse ou non appel aux package. Dans le paragraphe suivant, il sera question de la manipulation de composants sans se servir du package, leur manipulation en servant de celui ci sera considérée ultérieurement, au paragraphe dédier au package.

IV.2. MANIPULATION DES COMPOSANTS SANS UTILISATION DU PACAKGE

On vient de le signaler, le package fera l'objet d'une description bien détaillée dans un paragraphe dédié. Laissons de coté cet outil et considérons la manipulation de composants sans celui-ci.

Un programme Master, au nom de Master, se sert de composants selon la syntaxe suivante:

Syntaxe

```
entity Master is
    port(déclarations des entrées sorties de la structure Master) ;
end entity ;

architecture Arch of Master is
    -- Déclarations des composants;
    Autres declarations;
begin
    .....
    Instructions concurrentes;
    .....
    -- Instantiations des composants;
    .....
    Instructions concurrentes;
```

```
end Arch;
```

La manipulation (l'utilisation) d'un composant se fait donc en deux étapes:

IV.2.1. DECLARATIONS DES COMPOSANTS

La déclaration d'un composant est portée dans la partie déclarative de l'architecture selon la syntaxe suivante:

Syntaxe

```
component component_Identifier is
    generic(generic_list) ;
    Port(port_list) ;
end component ;
```

Dans le cas de notre Half Adder, cela donnerai :

```
component Half_Adder is
    port (
        Bit1    : in   std_logic;
        Bit2    : in   std_logic;
        Sum     : out  std_logic
        Carry   : out  std_logic
    );
end component;
```

Chaque composant qui sera utilisé nécessite d'abord une déclaration dans la partie déclarative de l'architecture qui l'utilise.

IV.2.2. INSTANCIATIONS DES COMPOSANTS

Les composants sont maintenant crées et déclarés ci dessus), il est maintenant question de les utiliser. Pour utiliser un composant il faut une copie (une instance) de celui-ci. Une fois le composant déclaré, il est possible de faire autant d'instances (copies) qu'il en faille. Cette instance s'opère selon la syntaxe suivante :

Syntaxe

```
Label: Component_Identifier  port map (liste des signaux  réels);
```

- ☞ Chaque instance d'un composant doit posséder une étiquette.
- ☞ Le mot clé port map sert à faire la correspondance (mapping) entre les signaux formels du composant (signaux de son port), et les signaux réels de l'entity qui l'utilise. Chaque signal dans la liste du port du composant (nom formel) doit être connecté à un signal de l'entity ou de l'architecture qui l'utilise (nom réel). Ces associations peuvent être implicites ou explicites:

Appliquons les directives des deux paragraphes précédents à notre Full Adder, on aura le code suivant:

```
library ieee;
use ieee.std_logic_1164.all;

entity  Full_Adder   is
    port(
        Bit1          : in   std_logic;
        Bit2          : in   std_logic;
        Carry_In      : in   std_logic;
        Sum           : out  std_logic;
        Carry_Out     : out  std_logic
    );
end fulladd;

architecture Arc of  Full_Adder  is
    component Half_Adder
        port (
            Bit1    : in   std_logic;
            Bit2    : in   std_logic;
            Sum     : out  std_logic
            Carry   : out  std_logic
        );
    end component;

    signal  S1, C1,   C2 :  std_logic;

begin
    U1: Half_Adder    port map(Bit1, Bit2, S1, C1);          -- First Instanciation of the Half Adder:
    U2: Half_Adder    port map(S1, Carry_In, Sum, C2);       -- Second Instanciation of the Half Adder:
    Carry_Out  <= C1 or C2;

end Arc;
```

Ces deux étapes, déclaration et instanciation, supposent bien sûr que le couple antity/architecture, qui décrit le comportement de chaque composant, existe quelque part. Ce quelque part désigne l'endroit où sont sauvegardés ces couples. Deux possiblités sont offertes :

IV.2.3. COMPOSANTS INTEGRES AVEC LE PROGRAMME QUI LES UTILISE

Ici toutes les paires Entity/Architecture de tous les composants font partie intégrante du programme qui les utilise (Master). Le code résultant est sauvegardé dans un fichier unique avec l'extention « **vhd** ». Pour expliciter les choses reprenons l'exemple de notre Full Adder. Si le fichier résultant est nomé Full_Adder.vhd par exemple, il aura la forme suivante:

```
library ieee;
use ieee.std_logic_1164.all;

entity  Full_Adder   is
    port(
        Bit1          : in   std_logic;                    -- First bit  to add
        Bit2          : in   std_logic;                    -- Second bit  to add
        Carry_In      : in   std_logic;                    -- Carry input from previous stage
        Sum           : out  std_logic;                    -- The sum
        Carry_Out     : out  std_logic                     -- Carry result from addition
    );
end fulladd;

architecture Arc of  Full_Adder  is
    component Half_Adder                                   --   Half Adder declaration
        port (
            Bit1    : in   std_logic;
            Bit2    : in   std_logic;
            Sum     : out  std_logic
            Carry   : out  std_logic
        );
    end component;
    signal  S1, C1,    C2 :  std_logic;
begin
    U1: Half_Adder     port map(Bit1, Bit2, S1, C1);
    U2: Half_Adder     port map(S1, Carry_In, Sum, C2);
    Carry_Out  <= C1 or C2;
end Arc;

library ieee;
use ieee.std_logic_1164.all;

entity Half_Adder
    port (
        Bit1    : in   std_logic;
        Bit2    : in   std_logic;
        Sum     : out  std_logic
        Carry   : out  std_logic
    );
end;

architecture Arc of Half_Adder is
begin
    Sum    <= Bit1 xor    Bit2;
    Carry  <= Bit1 and    Bit2;
end Arc;
```

IV.2.4. COMPOSANTS SEPARES DU PROGRAMME QUI LES UTILISE

Dans cette configuration, les paires Entity/Architecture de tous les composants sont sauvegardées dans un fichier à part entière et la paire Entity/Architecture qui les utilise dans un autre fichier. Ceci donnera naissance donc à deux fichiers avec l'extention **vhd**:

- Un fichier, au nom de My_Components.vhd, par exemple, qui rassemble toutes les paires Entity/Architecture de tous les composants: Dans le cas de notre Full Adder, ce fichier contient uniquement la paire Entity/Architecture de notre composant unique Half Adder; et
- Un fichier, au nom de Full_Adder.vhd par exemple, qui contient la paire entity/architecture Master qui utilise ces composants.

Dans le cas de notre Full Adder ces deux fichiers ont l'allure suivante :

```
--File Full_Adder.vhd

library ieee;
use ieee.std_logic_1164.all;

entity  Full_Adder  is
    port(
        Bit1, Bit2    :  in    std_logic;                    -- Two bits to add
        Carry_In      :  in    std_logic;                    -- Carry input from previous stage
        Sum           :  out   std_logic;                    -- The sum
        Carry_Out     :  out   std_logic                     -- Carry results from addition
    );
end Full_Adder;

architecture Arc  of  Full_Adder  is
    component Half_Adder
        port(
            Bit1, Bit2   : in   std_logic;
            Sum, Carry   : out std_logic
        );
    end component;
    signal  S1, C1,   C2 :  std_logic;
begin
    U1: Half_Adder    port map(Bit1, Bit2, S1, C1);
    U2: Half_Adder    port map(Bit2 => Carry_In, carry => C2, Bit1=> S1, Sum => Sum);
    Carry_Out  <= C1 or C2;
end Arc  ;


-- File My_Components.vhd

library ieee;
use ieee.std_logic_1164.all;
entity Half_Adder is
    port(
        Bit1, Bit2    : in   std_logic;
        Sum, Carry    : out std_logic
    );
end Half_Adder;
architecture arc of Half_Adder is
begin
```

```
    Sum  <=   Bit1 xor  Bit2;
    Carry <=   Bit1 and Bit2;
end arc;
```

☞ Ces deux fichiers doivent être assemblés avant la compilation du projet tout entier.

IV.3. LA GENERICITE : generic

Un generic est un paramètre dont la valeur peut être modifiée à toute instanciation. Cette modification est propagée à tous les niveaux où ce generic est utilisé. Ca permet de créer des codes flexibles et ainsi créer des structures paramétrisables et adaptables aux différents développements.

Syntaxe générale

```
entity toto is
    generic(
        generic 1;                          -- declaration of the first generic parameter
        generic 2;                          -- declaration of the second generic parameter
        ..........
        generic n                           -- declaration of the nth generic parameter
    );
    port (
    .........;
    .........;
    .........
    );
end toto;
```

Exemple:

```
generic( N : natural := 32) ;
```

IV.3.1. OU PEUT-ON UTILISER LA GENERICITE ?

La généricité est à utiliser exclusivement dans l'entity, et plus précisement avant la déclaration des ports de celle ci.

☞ Une instance d'entity generic a un generic map déclarée avant le port map

IV.3.2. UTILISATION D'UNE ENTITE GENERIQUE

Pour mieu expliciter les choses, considérons un Multiplexeur dont la largeur des données est variablee. La difficulté réside dans le fait que cette largeur est un paramètre qui entre dans la configuration des ports de l'entity, et qu'il n'est donc pas possible, à priori, de définir ce port puisque ce nombre est varaible.

La solution, pour décrire cette entity, consiste alors à créer un paramètre generic et l'utiliser pour parmétrer les ports de l'entity. Puisque ce paramètre generic est introduit avant la création des ports, il devient aussitôt valable pour la création de ceux-ci. Ainsi, tout les ports de l'entity se trouve automatiquement reconfigurer en fonction de ce paramètres.

Dans notre exemple, le Multiplexeur dispose de quatre entrées et une sortie, toutes à largeur variable et dépendant du paramètre generic Data_Width, fixé par défaut à 8. Au moment de l'appl de cettte entity, si Data_Width s'est vu attribué explicitement une valeur, c'est celle-ci qui sera considérée dans la définition des largeurs des entrées/sortie du port de cette entity, si au moment de l'appel ce paramètre n'est pas spécifié explicitement, la valeur par défaut, égale à 8, est considérée et les entrées/sortie du port de cette entity sont développés en conséquence. On a ainsi crée un Multiplexeur flexible juste par l'introduction du paramètre generic Data_Width.

Le programme VHDL qui traduit cette idée est donnée ci dessous.

```
library ieee;
use ieee.std_logic_1164.all;

entity Mux_generic is

    generic(
        Data_Width : positive := 8
    );

    port(
        Input1   : in     std_logic_vector(Data_Width downto 0);
        Input2   : in     std_logic_vector(Data_Width downto 0);
        Input3   : in     std_logic_vector(Data_Width downto 0);
        Input4   : in     std_logic_vector(Data_Width downto 0);
        Sel      : in     std_logic_vector(1 downto 0);
        Output: out    std_logic_vector(Data_Width downto 0)
    );
end Demux_generic ;

architecture Arc of Demux_generic is
begin
    process(Sel, Input1, Input2, Input3, Input4) is
    begin
        case Sel is
            when  "00"    =>  Output <= Input1;
            when  "01"    =>  Output <= Input2;
             when "10"    =>  Output <= Input3;
            when  "11"    =>  Output <= Input4;
             when others  =>  Output <= (others => 'X');
        end case;
    end process;
end Arc;
```

☞ Le parameter generic Data_Width est utilisé pour configurer le port de l'entity, c'est-à-dire pour fixer la largeur des entrées/sortie. L'appel de cette entity, en tant que component, avec des valeurs différentes pour le paramètre generic Data_Width entraine la création de multiplexeur à largeurs de données différentes.

IV.4. LA DUPLICATION DE COMPOSANTS

IV.4.1. L'INSTRUCTION generate

Le generic permet de rendre modulable le port d'une entity. Il est utilisé pour créer des entity qui s'adaptent automatiquement à différentes configuration. Qu'en est-il si on veut dupliquer cette entity un certain nombre de fois? A ce niveau, la réponse est par utilisation de la syntaxe:

```
Label: Component_Identifier
        generic map (paramètres du generic)
        port map (paramètres du port);
```

Qui permet de créer la structure (component) au nom Component_Identifier le nombre de fois (la dupliquer) dont elle est exécutée. Chaque duplication est une instance de cette structure. Malheureusement cette solution, si simple, n'est plus pratique si le nombre de duplications est important. Imaginez que cette duplication est égale à quelque centaine, même avec une opération de copie/coller, dupliquer tout ce nombre serait peu pratique et donnerai des programmes pour le moins trops longs et déficile à suivre. VHDL offre la solution à ce problème: Il s'agit de l'instruction generate. Elle est généralement utilisée pour instancier (dupliquer) des entity un nombre de fois selon les syntaxes suivantes :

Syntaxe 1

```
Label1: for Index in Start_Index to End_Index generate
    Label2: Component_Identifier
        generic map (paramètres du generic)
        port map (paramètres du port);
end generate Label1;
```

Syntaxe 2

```
Label1: if(Boolean statement) generate
    Label2: Component_Identifier
        generic map (paramètres du generic)
        port map (paramètres du port);
end generate Label1;
```

Où :

Label1: est une etiquette obligatoire pour la génération ;

Label2: est une étiquette obligatoire pour l'instanciation

index : est l'indice de la boucle

Start_Index : la valeur de départ de l'index

End_Index : la valeur de fin de l'index

Dans la première syntaxe, le composant, au nom de Component_Identifier, est dupliqué (End_Index - Start_Index +1) fois.

Dans la seconde syntaxe, la duplication du composant est opérée une seule fois, si une condition (Boolean statement) est remplie.