Cours VHDL - IV

L3-S6 - Université de Cergy-Pontoise

Laurent Rodriguez – Benoît Miramond

Plan du cours

I – Historique de conception des circuits intégrés
- **HDL**
  - Modèles de conceptions
- **VHDL**
  - Les modèles de conceptions en VHDL
  - Les 5 briques de base

II – VHDL et FPGA
- **VHDL**
  - **Syntaxe et typage**
  - **Retour sur les briques de bases**
  - **Retour sur la conception structurelle et comportementale en VHDL**
    Port map, Equations logiques, Tables de vérités (With ... Select)
- **FPGA**
  - **Qu'est ce qu'un FPGA**
  - **Flot de conception**
  - **Carte de développement et environnement de TP**

Plan du cours

III – Modélisation

- Compléments sur la description Structurelle

- Description comportementale approfondie

Circuits combinatoires

- Styles de description

Flots de données

Instructions concurrentes

Table de vérité

Fonctions

- Circuits standards

- Comparateur
- Multiplexeurs
- Encodeurs

Plan du cours

IV – Processus et gestion du temps
- **Syntaxe et sémantique**
- **Portée - signaux & variables**
- **Annotations temporelles et délais**

V – Simulation
- **TestBench**
- **GHDL & GTK-Waves**
- **Quartus & ModelSIM**

VI – Modèles génériques
- **Paramètres génériques**
- **« Generate »**
- **Boucles**
- **Exemples**

IV - Processus

PROCESSUS ET GESTION DU TEMPS

IV – Processus : Syntaxe et sémantique

Rappel : en VHDL : Toutes les instructions entre BEGIN et END d'une architecture sont **concurrentes**, c'est à dire qu'elle s'exécutent en **parallèle**.

- Un **process** se déclare entre Begin et End d'une architecture. La tache réalisée par le process est donc concurrente à toutes autres instructions de l'architecture.
- Au même titre que les autres instructions concurrentes, les process communiquent entre eux par des signaux
- Le **corps** du process contient des **instructions séquentielles.** Le résultat de synthèse sera Combinatoire (CF cours III) ou séquentiel (impliquant une mémorisation).
- La fonction de **Process** est de permettre la création d'instructions concurrente complexes.

Son comportement :

- **C'est une boucle infinie**, lorsqu'il arrive à la fin du code, il reprend automatiquement au début
- Il doit être **sensible** des **points d'arrêt** de façon à **le synchroniser**. La synchronisation est donc indiquée par un point d'arrêt qui est évènement particulier.

IV – Processus : Synchronisation

Il existe 2 types de points d'arrêts :

- Le processus est associé à une **"liste de sensibilité"** qui contient une liste de signaux qui réveillent le processus lors d'un changement d'un des signaux. Sa syntaxe est

```
PROCESS(« liste de signaux »)
BEGIN
    ...
END PROCESS ;
```

- Le processus a **des instructions d'arrêt wait** dans sa description interne. Le wait est sensible soit à un signal soit à un temps physique

```
PROCESS
BEGIN
    ...
    Wait on « liste de signaux » ;
END PROCESS ;
```

IV – Processus : Synchronisation (Liste de sensibilité)

Compléments de synchronisation par liste de sensibilité :

La liste des signaux réveillant le processus est indiquée dans la liste de sensibilité: process(liste de sensibilité).

Cette méthode est tout à fait équivalente à celle consistant à utiliser un wait on liste à la fin du corps du processus.

L' exemple suivant permet de coder la logique combinatoire :

Rappel cours III, pour que le résultat corresponde à un circuit combinatoire, tous les signaux d'entrées doivent être dans la liste.

```
process(x,y)
begin
        if x = '1'
then
            z <=
y;
        else
            z <=
'0';
        end if;
```

IV – Processus : Synchronisation (Liste de sensibilité)

Compléments de synchronisation par liste de sensibilité :

Pour fiabiliser un **processus séquentiel**, il est fortement recommandé de fonctionner en mode synchrone, c'est à dire avec l'utilisation d'une horloge qui échantillonne le calcul.

=> un processus séquentiel synchrone a une liste de sensibilité qui se réduit simplement à l'horloge et d'éventuels signaux de contrôles (en, reset, ...).

=> Les signaux codés dans ce processus seront donc des sorties de bascule D.

```
process(clk, n_reset)
begin

if n_reset = '0' then
        sortie <= (others=>'0'); -- tous les bits de sortie sont initialisés à 0 par le reset
elsif clk'event and clk='1' then
        liste d'instructions codant "sortie"
end if;

end process;
```

IV – Processus : Synchronisation (WAIT)

Compléments de synchronisation par wait :

- L'instruction WAIT permet de mettre des points d'arrêt dans le corps du processus. La syntaxe de l'instruction est la suivante :

wait [on S1,S2,...] [until CONDITION] [for DUREE] ;

Avec S1, S2, ... des signaux, CONDITION une expression générant un booléen et DUREE un temps physique d'attente.

L'instruction WAIT n'est pas synthétisable avec la condition de durée.

Elle est très utile pour les **testbench** pour générer précisément des formes de vecteurs d'entrée.

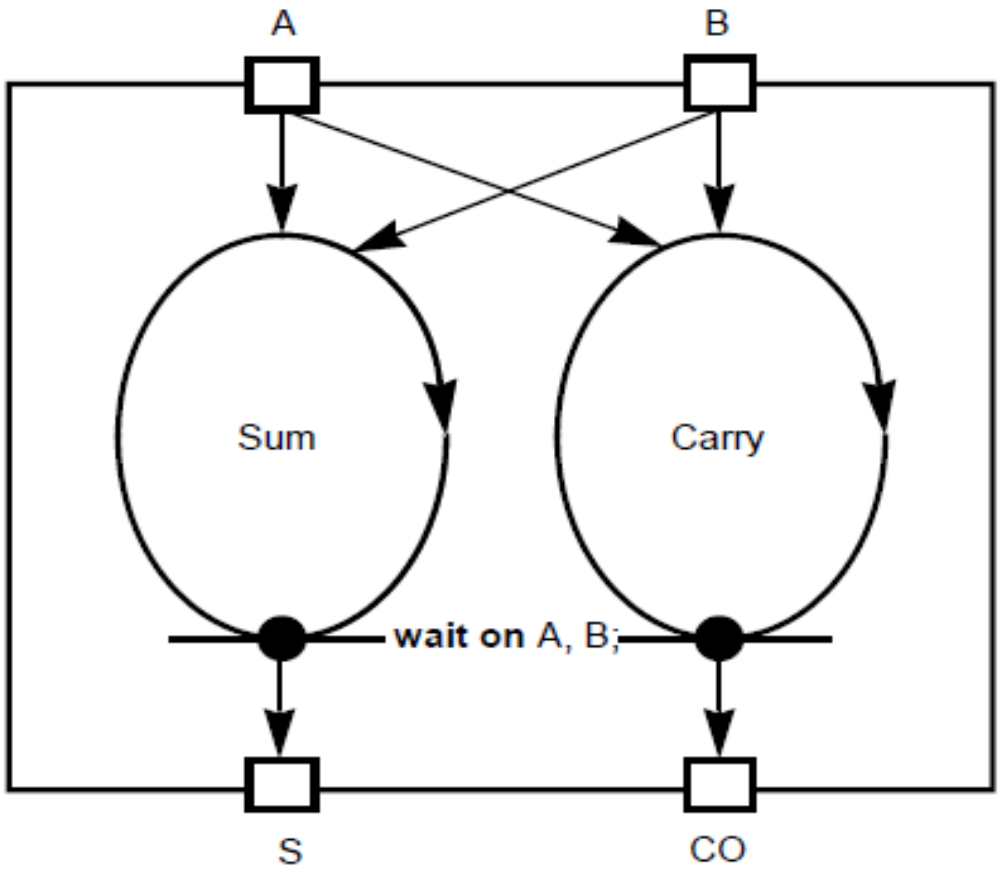
IV – Processus : Synchronisation (WAIT)

```
trame : process      -- le processus peut avoir une étiquette, ici "trame"
BEGIN                -- il n'y a pas de liste de sensibilité donc il faut des "wait"
    pulse <= '0';    -- pulse est à 0 en début de trame
    for i in 0 to 9 loop  -- on génère 10 impulsions larges de 2 périodes d'horloge
        wait until clk'event and clk='1';
        pulse <= '1';
        wait until clk'event and clk='1' ;
        pulse <= '0';
    end loop;
    wait for 100 us; -- après 100 us on reprend le process donc pulse va repasser à 0
end process;
```

IV – Processus : Synchronisation (WAIT) :

Exemple de modèle à deux processus

```
entity half_adder is
    port (A, B: in BIT; S, CO: out BIT);
end entity half_adder;

architecture bhv of half_adder is
begin

    Sum: process
    begin
        S <= A xor B;
        wait on A, B;
    end process Sum;

    Carry: process
begin
        CO <= A and B;
        wait on A, B;
    end process Carry;

end architecture bhv;
```



IV – Processus : Portée - signaux & variables

Les signaux : **équivalents à des variables globales assurant les communications entre processus.**

Ils ne sont pas mis à jour tout de suite mais avec un delta-cycle de retard par rapport au réveil du processus.

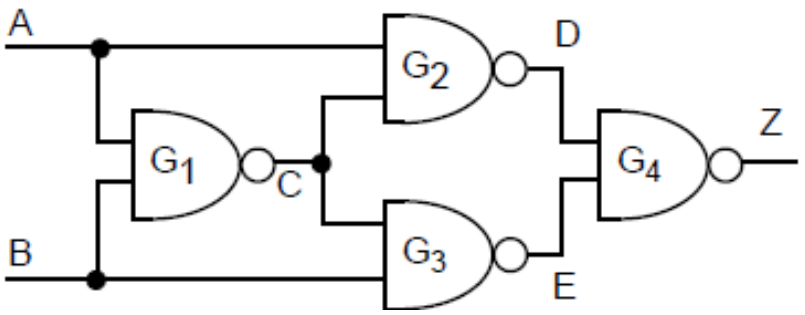
Les variables : locales à chaque processus et sont mises à jour immédiatement.

Elles sont très utiles pour effectuer un codage séquentiel classique comme avec le langage C. Les variables sont déclarées juste avant le corps du processus et sont affectées avec l'instruction d'affectation immédiate **:=** de façon à bien ne pas confondre avec l'instruction <= "reçoit" pour les signaux. Les variables gardent leur valeur quand le processus est terminé.

```
process (s)
variable x : integer;
begin
      x := s+1;
      a <= s+1;
      b <= x;
      c <= a;
end process;
```

IV – Processus : Annotations temporelles et délais

```
entity noteq is
    port (A, B: in BIT; Z: out BIT);
end entity noteq;

architecture bhv of noteq is
    signal C, D, E: BIT;
begin
    P1: C <= A nand B;
    P2: D <= A nand C;
    P3: E <= C nand B;
    P4: Z <= D nand E;
end architecture bhv;
```



t	A	B	C	D	E	Z
T0	0	0	1	1	1	0
T0+10ns	1 : G1,G2	0	1	1	1	0
Δ0 : G1	1		calcul	Calcul		
Δ1			1’	0’ : G4		Calcul
Δ2						1’ : -
T=T0+10ns						

IV – Processus : Annotations temporelles et délais

VHDL permet de spécifier des délais dans les affectations.

Il existe deux types d'affectations avec délai :

- affectation avec délai inertiel : x <= 3 after 2 ns; ou plus explicitement x <= inertial 3 after 2 ns;
- affectation avec délai de transport : x <= transport 3 after 2 ns;

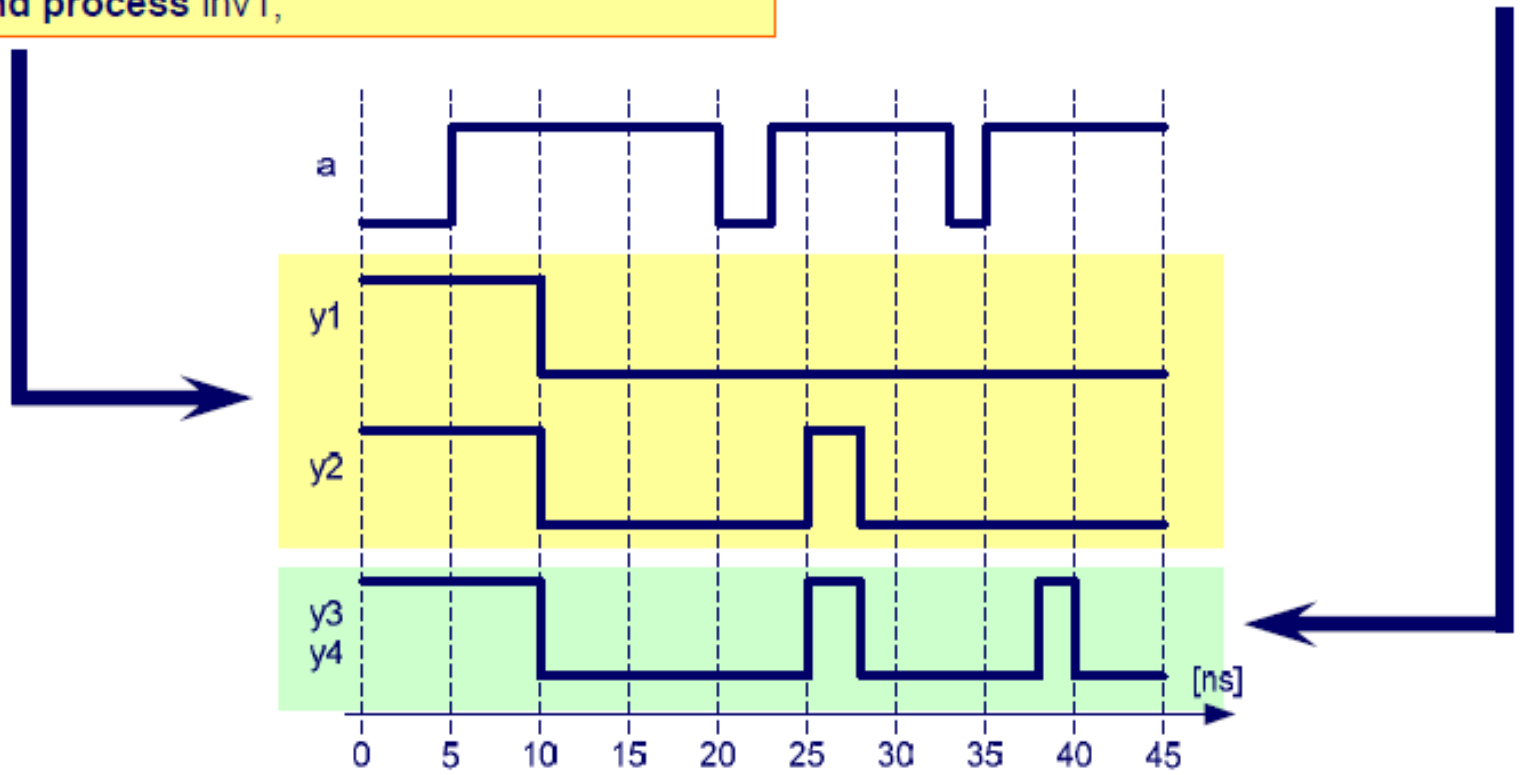
Le type inertiel permet de "filtrer" les variations de X trop courtes par rapport au délai de la transaction.

Le type transport n'opère pas de "réjection des parasites". Il respecte les temps de propagation mais est certainement moins réaliste que le mode inertiel.

IV – Processus : Exemples

```
inv1: process (A) is
begin
    Y1 <= not A after 5 ns;
--  Y1 <= inertial not A after 5 ns;
    Y2 <= reject 2 ns inertial not A after 5 ns;
end process inv1;
```

```
inv2: process (A) is
begin
    Y3 <= transport not A after 5 ns;
    Y4 <= reject 0 ns inertial not A after 5 ns;
end process inv2;
```



V - Simulation

PROCESSUS ET GESTION DU TEMPS

V – Simulation

La ***phase d'initialisation est composée des pas suivants:***
I1) Affectation des valeurs par défaut ou des valeurs initiales . tous les signaux et les variables. Les signaux et variables de type T sont initialisés à la valeur T'LEFT :
I2) Temps T := 0 ns.
I3) Exécution de tous les processus jusqu'à leur première instruction **wait.**

```
signal S1: BIT;                    -- S1 = '0' à T = 0
signal S2: INTEGER;                -- S2 = -231-1 (32 bits) à T = 0
signal S3: BIT_VECTOR(0 to 7);     -- S3 = "00000000" à T = 0
```

Le ***cycle de simulation est composé des pas suivants:***
S1) Temps T := temps de la (des) prochaine(s) transaction(s).
S2) Mise à jour des signaux si événement.
S3) Exécution de tous les processus sensibles aux signaux mis à jour jusqu'à leur prochaine instruction **wait**
S4) Si nouvelles transactions au temps courant, retour en S2 (cycle delta), sinon, si T = TIME'HIGH ou si plus de transaction, fin de la simulation, sinon, retour en S1.

V – Simulation : TestBench

Un TestBench (ou banc de test), en simulation, permet de vérifier la validité d'une architecture.

C'est un module VHDL (entity + architecture) qui instancie l'architecture à tester.

Sa déclaration d'entité de contient par d'entrées/sorties !

Le but est de la soumettre à des variations sur les entrées affin de vérifier la correction des sorties générées et de la validité de son comportement temporel.

Ex :

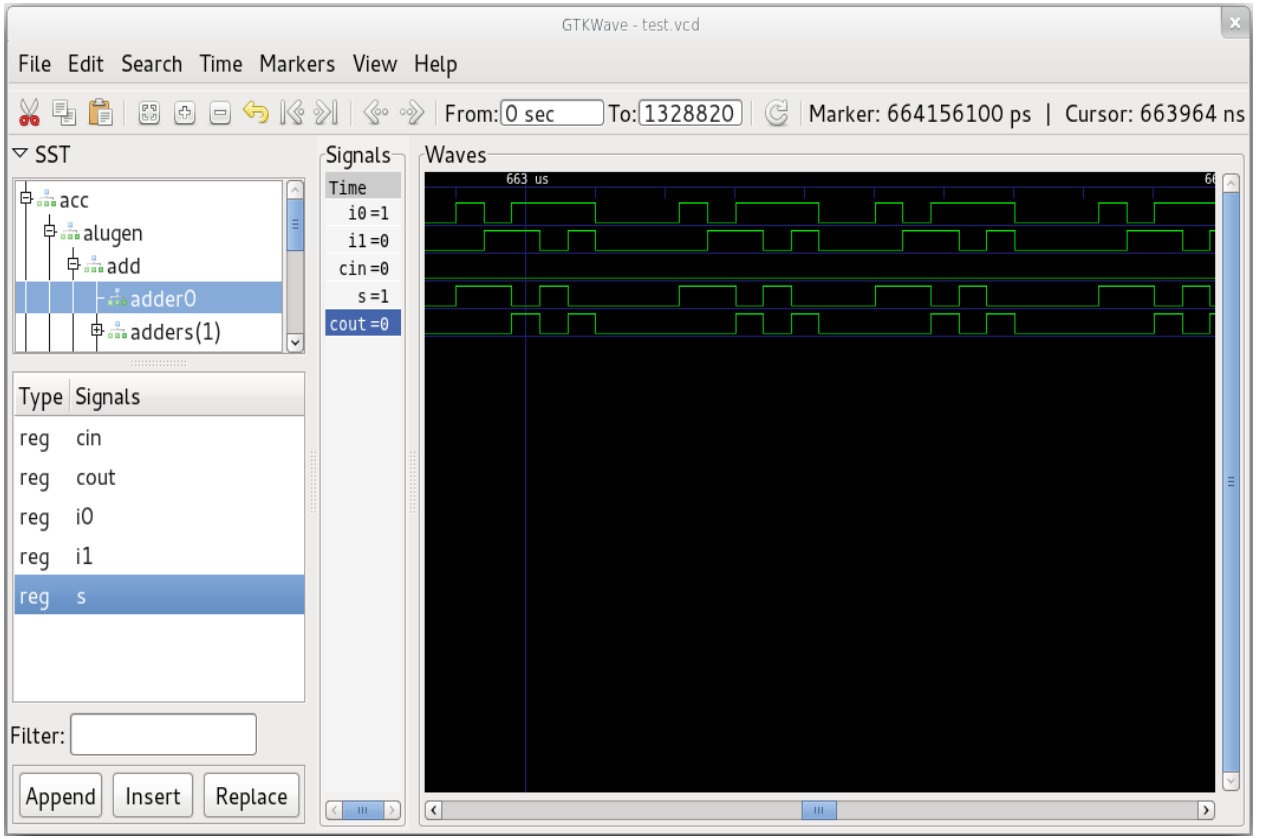
```
entity adder_tb is
end adder_tb;
```

Elle inclue la gestion du temps et la vérification de correction par assertions.

(CF : fa_tb.vhdl)

V – Simulation : avec GHDL/GTKWaves

```
SRCS= $(shell ls *.vhdl)
OBJS= $(SRCS:.vhdl=.o)

all: $(OBJS)
	ghdl -e test

run: all
	ghdl -r test --vcd=test.vcd

%.o: %.vhdl
	ghdl -a --ieee=synopsys $^

View : run
	Gtkwaves test.vcd
clean:
	rm -f *.o *~ \#*\# work*.cf
```



V – Simulation : avec Quartus/ModelSim

Il faut générer sous quartus les données nécessaires pour la simulation sous ModelSim.

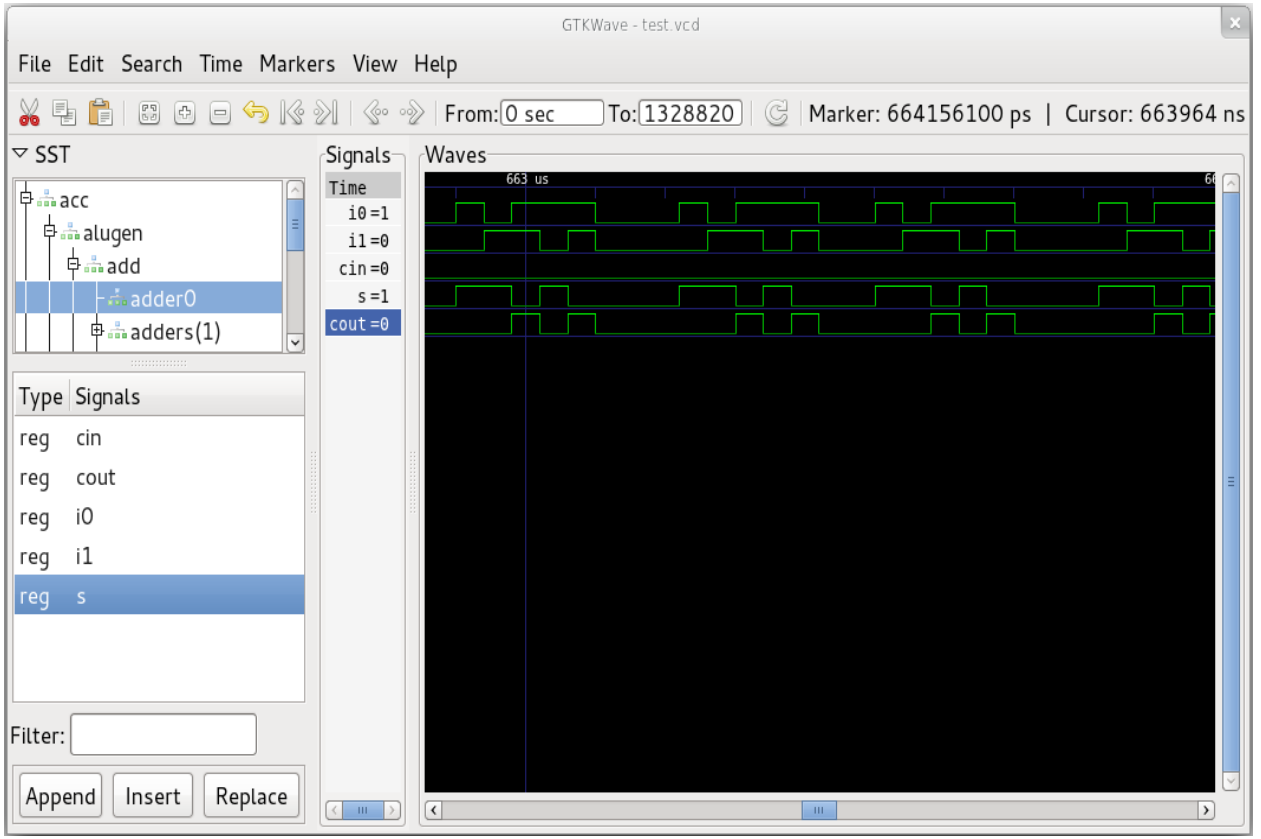
- Dans : Assignments/Settings
- sélectionner Modelsim-Altera comme outil. Décocher "Run gate-level simulation automatically after compilation".
- Compiler le projet

Dans ModelSim-Altera :

- File/Change Directory...
- sélectionner le répertoire « Simulation/ModelSim » de votre projet
- File/new/library pour créer une library « work »
- Ouvrir le fichier VHDL de votre test bench
- Compiler
- Dans la vue library double cliquer sur le fichier test bench et lancer la simulation
- La vue Waves permet de visualiser les chronogrammes à la manière de GTKWaves

V – Simulation : avec GHDL/GTKWaves

```
SRCS= $(shell ls *.vhdl)
OBJS= $(SRCS:.vhdl=.o)

all: $(OBJS)
	ghdl -e test

run: all
	ghdl -r test --vcd=test.vcd

%.o: %.vhdl
	ghdl -a --ieee=synopsys $^

View : run
	Gtkwaves test.vcd
clean:
	rm -f *.o *~ \#*\# work*.cf
```



VI – Modèles Génériques

Génération de circuits paramétrables

VI – Modèles génériques

Le langage permet de paramétrer les modèles pour les rendre plus génériques grâce à :

- Les paramètres génériques
- Les objets d'interface non contraints
- L'instruction concurrente **generate**

Syntaxe :

```
entity entity_name is
 Generic (« parameters list »);
 Port ( « ports list » );
end entity;
```

Exemple :

```
entity Nbit_adder is
 generic( SIZE : positive :=8;);
 port (
 a,b : in unsigned(SIZE-1 downto 0);
 ci : in std_logic;
 sum : out unsigned(SIZE-1 downto 0);
 co : out std_logic);
end entity;
```

Instanciation :

```
inst: Nbit_adder generic map (16) port map(A, B, C, sum, C);
```

VI – Modèles génériques: Instruction generate

Elle permet de dupliquer de manière concurrente des instructions de manière itérative ou conditionnelle !

Les étiquettes des generate sont obligatoires

- La structure de contrôle itérative :

Etiquette : For ... in ... to ... generate
    ...
End generate

Les itérateurs dans la boucle ne sont pas à déclarer

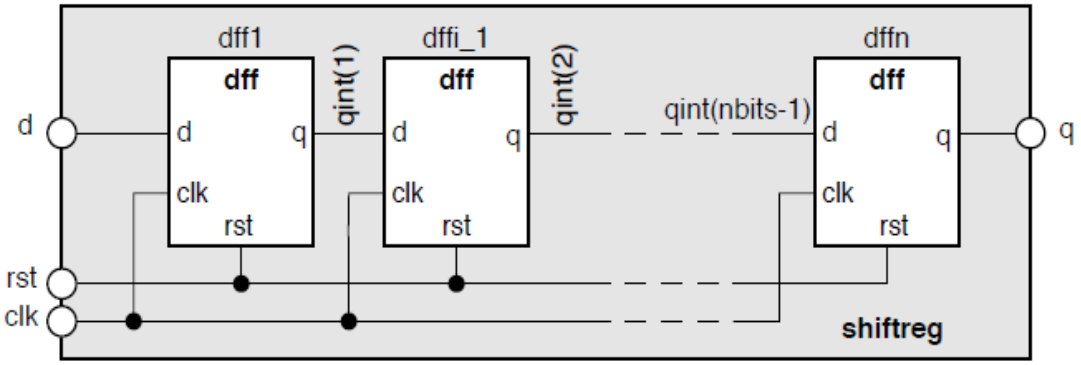
- la structure conditionnelle :

Etiquette : If ... generate

End generate ;

VI – Modèles génériques: Exemple

```
entity shiftreg is
    generic (nbits: POSITIVE := 8);
    port (clk, rst, d: in BIT; q: out BIT);
end entity shiftreg;

architecture str of shiftreg is

    component dff is
        port (clk, rst, d: in BIT; q: out BIT);
    end component dff;

    signal qint: BIT_VECTOR(1 to nbits-1);
begin
    cell_array: for i in 1 to nbits generate

        first_cell: if i = 1 generate  -- première cellule
            dff1: dff port map (clk, rst, d, qint(1));
        end generate first_cell;

        int_cell: if i > 1 and i < nbits generate  -- cellules internes
            dffi: dff port map (clk, rst, qint(i-1), qint(i));
        end generate int_cell;

        last_cell: if i = nbits generate  -- dernière cellule
            dffn: dff port map (clk, rst, qint(nbits-1), q);
        end generate last_cell;

    end generate cell_array;
end architecture str;
```



L'exemple ne duplique que des instances de composants, **mais l'instruction generate peut générer n'importe quelle instruction concurrente.**
La duplication des instructions concurrentes a lieu durant la phase d'élaboration avant de commencer la simulation proprement dite du modèle.

VI – Modèles génériques: Configuration d'un tel modèle

```
library GATES;
configuration shiftreg_str_cfg is
    for str
        for cell_array
            for first_cell
                for dff1: dff use entity GATES.dff(bhv) port map (clk, rst, d, q);
            end for; -- first_cell
            for int_cell
                for dffi: dff use entity GATES.dff(bhv) port map (clk, rst, d, q);
            end for; -- int_cell
            for last_cell
                for dffn: dff use entity GATES.dff(bhv) port map (clk, rst, d, q);
            end for; -- last_cell
        end for; -- cell_array
    end for; -- str
end configuration shiftreg_str_cfg;
```

Biblio

Documents de cours

- Ce cours en ligne à :

Http://perso-etis.ensea.fr/rodriguez/

- Un très bon support de cours

http://hdl.telecom-paristech.fr/index.html

- Le cours de Licence 2 (en particulier pour ceux qui ne l'ont pas suivi):

http://perso-etis.ensea.fr/miramond/Enseignement/L2/circuits_numeriques.html

- Le cours de Licence 3 de 2013 :

http://perso-etis.ensea.fr/miramond/Enseignement/L3/Cours_VHDL_2011.html

Documents de développement

- Quartus

http://quartushelp.altera.com/current/

- Documentation Altera sur les Cyclones II et IV (entre autre...)

http://www.altera.com/literature/lit-index.html