

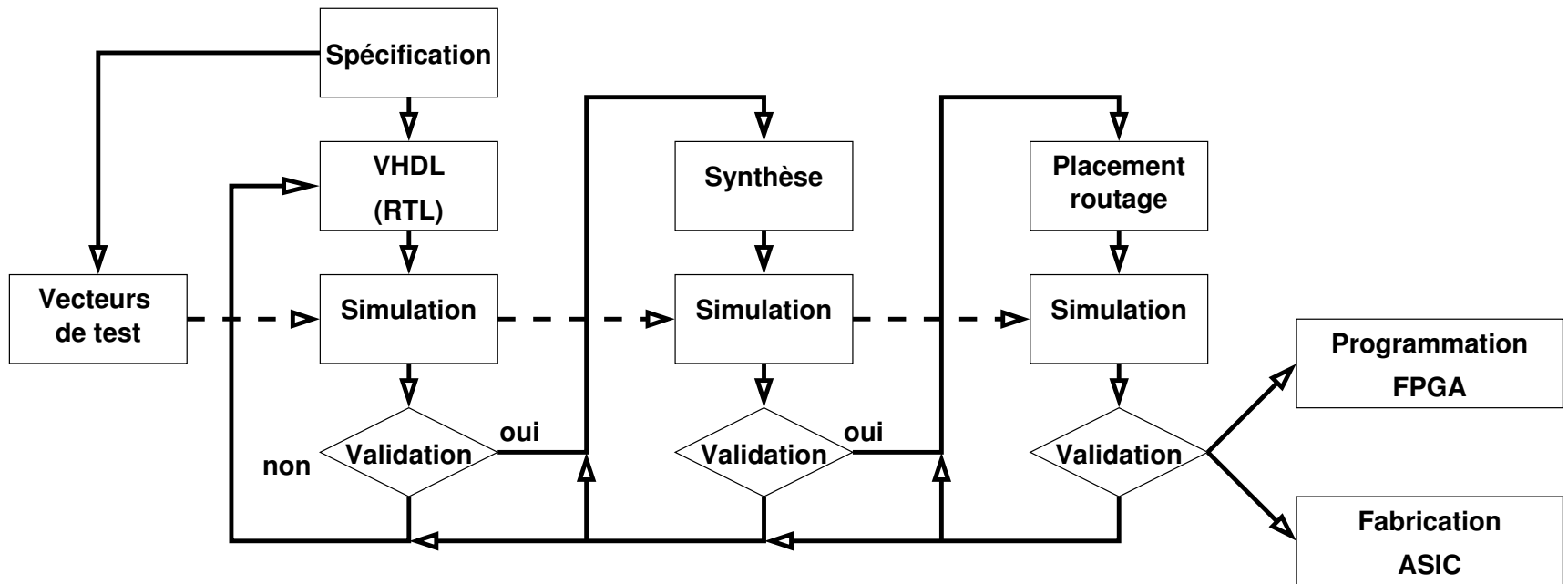
Notions de VHDL

S. Mancini

Plan

- ✖ Notions fondamentales
- ☐ Structuration des modèles
- ☐ Validation
- ☐ VHDL RTL
- ☐ Implémentation des machines à états
- ☐ Types de données pour le RTL

Flot de conception



VHDL : Very high speed integrated circuit Hardware Description Language

Langage de **description** de circuit et **de tests** de circuits.

Les éléments décrits fonctionnent **simultanément**, ce qui correspond à la simultanéité des circuits électroniques mais est très différent des langages séquentiels que vous connaissez !!!

?
? Comment décrire le parallélisme intrinsèque des circuits ?
?
?

Plusieurs unités matérielles fonctionnent en **parallèle**, en même temps.

Malheureusement les langages informatiques usuels sont **séquentiels** : les instructions s'exécutent dans l'ordre du texte du programme.

Solution proposée par VHDL

- ★ Des unités de calcul, fonctionnant en parallèle, exécutent chacune du code séquentiel (calcul).
- ★ Ces unités s'échangent des signaux.
- ★ Les calculs et échanges de signaux se font en temps nul.

Ces hypothèses, dites de **synchronisme fort**, ne sont pas réalistes mais seront vérifiées plus tard.



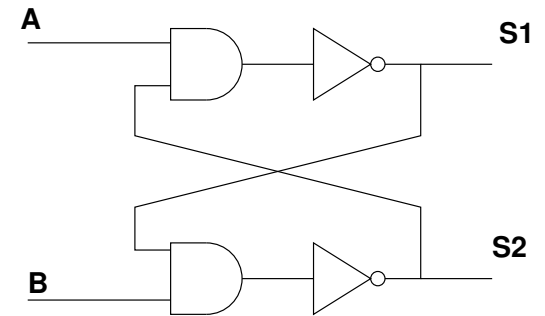
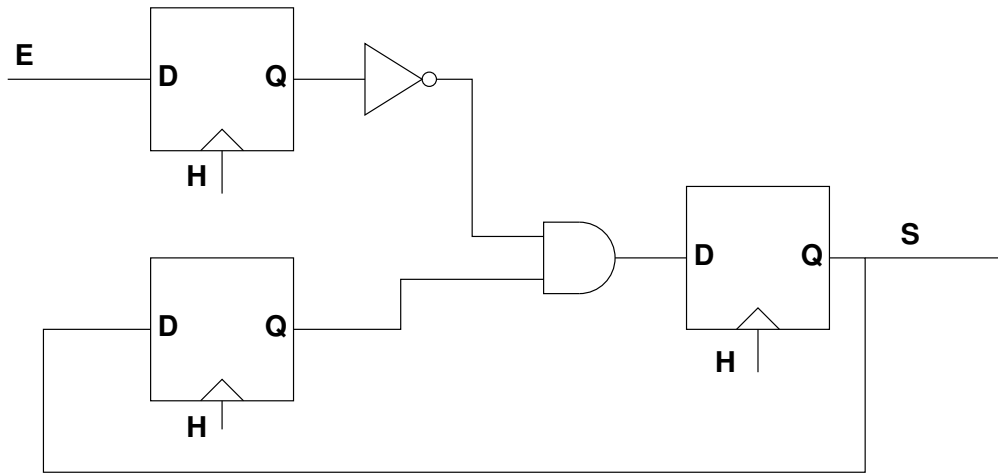
Dans les circuits numériques synchrones, cela revient à **faire abstraction des temps de propagation**. Cette hypothèse est vérifiée lorsque le chemin critique a un temps de propagation inférieur à la période d'horloge.



Ces hypothèses ne sont pas vraies sur une station de travail (le processeur est séquentiel) : on ne peut que **simuler** un modèle VHDL.

Le Delta-cycle par l'exemple

? Exercice : simuler les circuits suivants :



Notion de temps

VHDL est basé sur une notion de **temps relatif**.

Il est possible :

★ de programmer des événements :

`a <= '0' after 13 ns, '1' after 54 ns;`

★ d'attendre :

☐ un certain temps : `wait for 45 ns;`

☐ un événement : `wait until a='1';`

★ de détecter un changement de valeur : `a' event`



l'expression booléenne `a' event and a='1'` est vrai lorsqu'un front montant se produit (passage de '0' à '1').

Description fonctionnelle

VHDL est un langage informatique dérivé de ADA :

Toutes les syntaxes et grammaires de ADA sont conservées.

On a :

- ★ Types de données simple (entier, booléen, etc . . .)
- ★ Types de données complexes (tableaux, structures) et déclaration de nouveaux types
- ★ Structures de contrôle (boucles for, while, . . .)
- ★ Sous programmes et fonctions, surcharge des opérateurs



MAIS, seule une sous-partie de VHDL est utilisée pour modéliser des circuits. C'est ce qu'on appelle le

VHDL-RTL

voir les dix commandements

Notion de process

`Le process` est l'**unité de base** du VHDL. C'est l'endroit où la fonction est décrite.

`Un process` est une suite **séquentielle** d'instructions VHDL.

`Un process` **s'exécute en un temps nul** : chaque instruction s'exécute en un temps nul au sens de VHDL.



Le temps VHDL n'est pas le même que le temps mis par le processeur pour simuler l'exécution du modèle !

```
etiquette : process (liste de sensibilité)
-- zone déclarative
begin
-- code séquentiel
end process etiquette;
```




Notion de signal



Le **signal** étend la notion usuelle de variable pour tenir compte de son **évolution dans le temps**. Un **signal** permet à des **process** de communiquer.

Un signal est représenté par un **échancier** : c'est la liste des valeurs prises au cours du temps.

Un **signal** est affecté dans un **process** à une date qui peut être :

- ★ **explicite** : `a <= '0' after 13 ns, '1' after 24 ns;`
- ★ **implicite** : `a <= '0';`

   Dans le cas implicite, le signal **prend sa valeur à la fin** du **process** (i.e. fin du Δ -Cycle) ou à la prochaine instruction **wait**.

  Si un signal est affecté plusieurs fois au cours de l'exécution, seule la dernière affectation est prise en compte !

Exemple de signal

```
...  
signal a, b, c, e, f, g : std_logic;  
...  
toto : process (e, f)  
begin  
  a <= e or f;  
  g <= e xor f;  
end;  
  
gloub : process (a, b)  
begin  
  c <= a and b;  
end;  
...
```

Notion de variable

La **variable** reprend la notion classique de variable dans les programmes séquentiels. Une variable est déclarée **dans un process** .

L'affectation d'une variable se fait à la **fin de l'instruction** d'affectation. Au point virgule !

```
toto: process(s)
variable a : integer range 0 to 3;
begin
...
a := 2;
....
end toto;
```

Exemple de variable

```
...  
signal b, c, e, f, g : std_logic;  
...  
toto : process (e, f, b)  
variable a : std_logic;  
begin  
a := e or f;  
c <= a and b;  
g <= e xor f;  
end process toto;  
...
```

Plan

- ☐ Notions fondamentales
- ☒ Structuration des modèles
- ☐ Validation
- ☐ VHDL RTL
- ☐ Implémentation des machines à états
- ☐ Types de données pour le RTL

Modules et hiérarchie de conception

Les unités matérielles sont définies par un couple **entity /architecture** .

- L'entity définit l'interface du module avec l'extérieur
- L'architecture décrit le fonctionnement et la structure interne de l'entity

Une architecture contient :

★ Une zone déclarative

- ☐ les signaux internes à l'architecture
- ☐ les déclarations des sous-composants utilisés dans l'architecture (les sous-entity)

★ Un corps d'architecture

- ☐ les process qui encapsulent le code fonctionnel
- ☐ les instances des sous-composants utilisés

Conception hiérarchique

La conception **hiérarchique** permet d'utiliser une entité dans une architecture.

```
etiquette : composant_ou_entite
  port map (
    port => signal,
    ...
  );
```

Dans une `architecture` , toutes les instances d'entités et les `process` sont **parallèles**.

Dans un `process` , les instructions sont **séquentielles**.

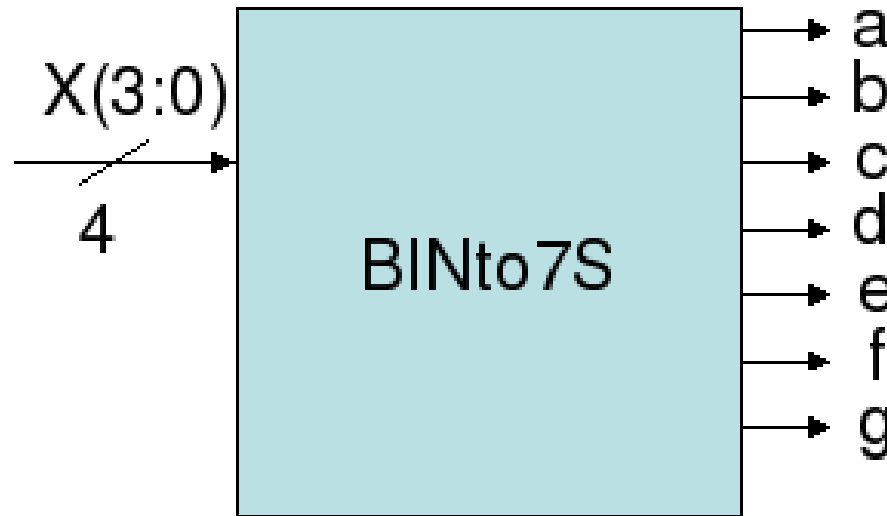


Il existe des **affectations concurrentes**, qui sont un **rac-courcis** d'écriture d'un `process` !

L'entité doit être déclarée par le mot clé `component`

Description vue de l'extérieur

```
entity binto7s is
  port (
    x : in std_logic_vector(3 downto 0) ;
    a,b,c,d,e,f,g : out std_logic
  ) ;
end binto7s;
```



Composant combinatoire simple

```
entity exemple1 is
  port (
    a, b, c : in std_logic;
    s : out std_logic
  );
end exemple1;
architecture RTL of exemple1 is
begin
  comb : process ( a, b, c)
    s <= (a or b) and not c;
  end process comb;
end RTL;
```

Bascule D

```
entity basculed is
  port (
    d : in std_logic_vector(15 downto 0);
    clk, reset : in std_logic;
    q : out std_logic_vector(15 downto 0)
  );
end basculed;
architecture RTL of basculed is
begin
  process (clk)
  begin
    if clk'event and clk='1' then
      if reset='1' then
        q <= (others => '0');
      else
        q <= d;
      end if;
    end if;
  end process;
end RTL;
```

Additionneur signé

```
entity addsub16 is
  port (
    a : in std_logic_vector(15 downto 0);
    b : in std_logic_vector(15 downto 0);
    s : out std_logic_vector(15 downto 0);
    carry : out std_logic;
    overflow : out std_logic
  );
end addsub16;

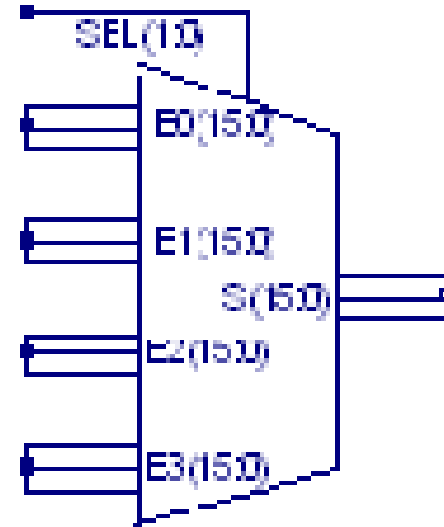
architecture RTL of addsub16 is
  signal s_interne : std_logic_vector(16 downto 0);
begin
  carry <= s_interne(16);
  s <= s_interne(15 downto 0);
  overflow <= (a(15) and b(15) and not s_interne(15)) or
    (not a(15) and not b(15) and s_interne(15));
  s_interne <= (a(15) & a) + (b(15) & b) ;
end RTL;
```



Ce sont des affectations concurrentes !
L'ordre d'exécution n'est pas celui du code.

Combinatoire et structure de contrôle

```
entity mux4_1_16bits is
  port(e0,e1,e2,e3 : in std_logic_vector(15 downto 0);
        sel : in std_logic_vector(1 downto 0);
        s : out std_logic_vector(15 downto 0)
  );
end mux4_1_16bits;
architecture RTL of mux4_1_16bits is
begin
  process (e0,e1,e2,e3,sel)
  begin
    case sel is
      when "00" =>
        s <= e0;
      when "01" =>
        s <= e1;
      when "10" =>
        s <= e2;
      when others => -- pour tous les autres cas
        s <= e3;
    end case;
  end process;
end RTL;
```



Petit exercice

?
? Que fait le code suivant ?
? A quel schéma de circuit numérique correspond-il ?
?

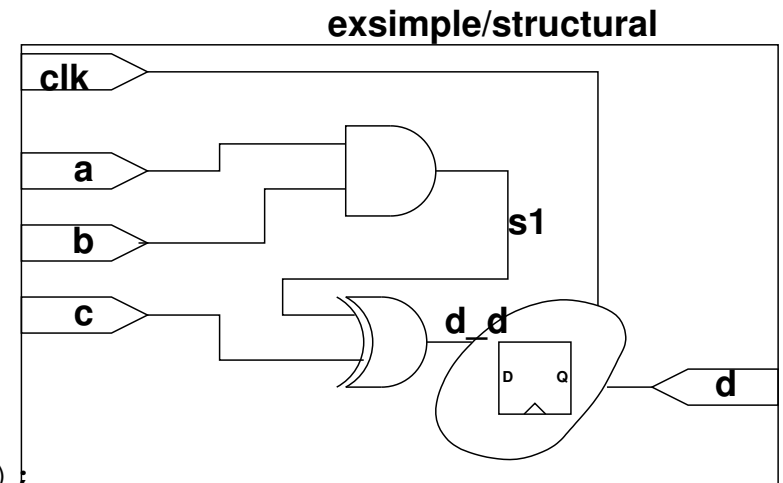
```
synchrone : process (clk)
begin
    if (clk'event and clk='1') then
        A <= A+1;
        B <= A;
    end if;
end process;

combinatoire : process (A)
begin
    C <= A+4;
end process;
```

Exemple de conception hiérarchique

```
entity exsimple is
  port (
    clk : in std_logic;
    a, b, c : in std_logic;
    d : out std_logic
  );
end exsimple;

architecture structural of exsimple is
  signal d_d, s1 : std_logic;
  component xor2
    port ( i0, i1 : in std_logic;
           o : out std_logic);
  end component;
  component and2
    port ( i0, i1 : in std_logic;
           o : out std_logic);
  end component;
begin
  bloc1 : xor2 port map (i0=>b, i1=>a, o=>s1);
  bloc2 : and2 port map (i0=>c, i1=>s1, o=>d_d);
  sync : process(clk)
  begin
    if clk'event and clk='1' then
      d <= d_d;
    end if;
  end process sync;
end structural;
```



Plan

- ☐ Notions fondamentales
- ☐ Structuration des modèles
- ☒ Validation
- ☐ VHDL RTL
- ☐ Implémentation des machines à états
- ☐ Types de données pour le RTL

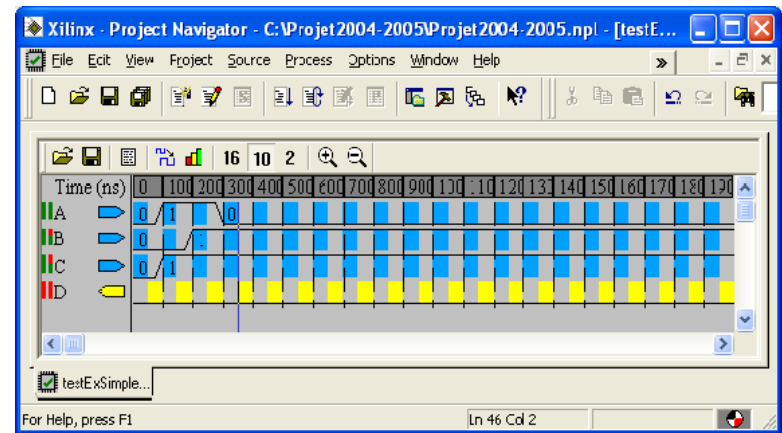
Description de tests

Utiliser VHDL pour concevoir ET tester les architectures.

```
entity testExSimple is end testExSimple;
architecture test of testExSimple is
begin
  stimulis : process
    a <= '0'; b <= '0'; c <= '0';
    wait for 100 ns;
    a <= '1'; c <= '1';
    wait for 100 ns; -- time=200 ns
    b <= '1'; -- -----
    wait for 100 ns; -- time=300 ns
    a <= '0'; -- -----
    wait for 150 ns; -- time=450 ns
  end process stimulis;

  instance_projet : projet
    port map ( a => a , ...);

end;
```



Génération d'une horloge et test synchrone

```
entity testExSimple is end testExSimple;
architecture simu of testExSimple is
    signal clk    : std_logic := '0';
begin
    clk <= not clk after 5 ns;
    stimuli : process
        a <= '0'; b <= '0'; c <= '0';
        wait until rising_edge(clk);
        a <= '1'; c <= '1';
        for i in 0 to 9 loop
            wait until rising_edge(clk);
        end loop;
        b <= '1'; -- -----
        wait until rising_edge(clk);
        a <= '0'; -- -----
        wait until rising_edge(clk);
    end process stimuli;

    instance_projet : projet
        port map ( a => a , ...);

end simu;
```

Plan

- ☐ Notions fondamentales
- ☐ Structuration des modèles
- ☐ Validation
- ☒ VHDL RTL
- ☐ Implémentation des machines à états
- ☐ Types de données pour le RTL

Le VHDL **RTL** (Register Transfert Level) est le sous-ensemble de VHDL utilisé pour décrire les circuits **numériques synchrones**.

Ces circuits sont constitués d'éléments :

★ **Combinatoires**

- ☐ Fonctions booléennes
- ☐ Arithmétique

★ **De mémorisation**

- ☐ Bascules D et dérivées (avec/sans enable, reset, ...)
- ☐ Mémoires ROM, SRAM simples/double ports

Le VHDL-RTL permet de produire des réseaux de portes équivalents à l'aide d'outils de **synthèse logique**. On parle aussi de VHDL **synthétisable**.

Modélisation de la combinatoire

Les parties combinatoires ont les caractéristiques suivantes :

- ★ Elles sont synthétisable en netlist (réseau) de **portes combinatoires** qui réalisent des fonctions **booléennes**
 - ➡ Il n'y a **pas de boucle** combinatoire
- ★ Elle réagissent en **permanence** aux entrées (de la partie combinatoire)
 - ➡ Il n'y a pas d'effet mémorisant

Modélisation des bascules D

Une bascule D est modélisée par un `process` qui réalise la copie de l'entrée sur la sortie lors d'un **front** d'horloge.

On peut ajouter une remise à zéro (ou autre valeur d'initialisation) et un peu de combinatoire.

reset synchrone

```
process (clk)
begin
  if clk'event and clk='1' then
    if (reset='1') then
      q<=(others =>'0');
    else
      q<=d;
    end if;
  end if;
end process;
```

reset asynchrone

```
process (clk, reset)
begin
  if reset='1' then
    q<=(others =>'0');
  elsif clk'event and clk='1' then
    q<=d;
  end if;
end process;
```



Pour le reset **asynchrone**, le signal de reset doit être **sans aléas**. Dans le doute, **il est déconseillé** lorsqu'il est généré en interne.

Modélisation des mémoires

Les mémoires sont des macro-blocs présents dans les FPGAs.
Il y a deux façons de les utiliser :

- ☐ Par instanciation de blocs spécifiques
- ☐ Par synthèse d'un code "type"



Nous verrons cela en détail au cours des projets !

Les 10 commandements du VHDL-RTL

- 👁 Tu décomposeras tes systèmes en **PC** et **PO**
- 👁 Tu sépareras la **combinatoire** des **registres**
- 👁 Pour faire de la **combinatoire**, tu **suivras** les règles suivantes :
 - Tu listeras tous les signaux utilisés dans le `process` dans la **liste de sensibilité**
 - Tu **affecteras un signal au moins une fois** entre le début et la fin du `process`
 - Tu ne feras **pas de boucles** combinatoire
 - Tu **n'affecteras et n'utiliseras pas** un signal dans le **même** `process`
 - Tu n'utiliseras que la **valeur** d'un signal, sans détecter de front
 - Tu n'affecteras un **signal** que dans **un seul** `process`
 - Tu n' **utiliseras pas d'instruction d'attente** entre le début et la fin du `process`
- 👁 Tu ne génèreras **pas d'horloge** ni avec des portes combinatoires ni avec des registres
- 👁 Tu seras **concis**, en regroupant les registres dans un seul `process` séquentiel

Plan

- ☐ Notions fondamentales
- ☐ Structuration des modèles
- ☐ Validation
- ☐ VHDL RTL
- ☒ Implémentation des machines à états
- ☐ Types de données pour le RTL

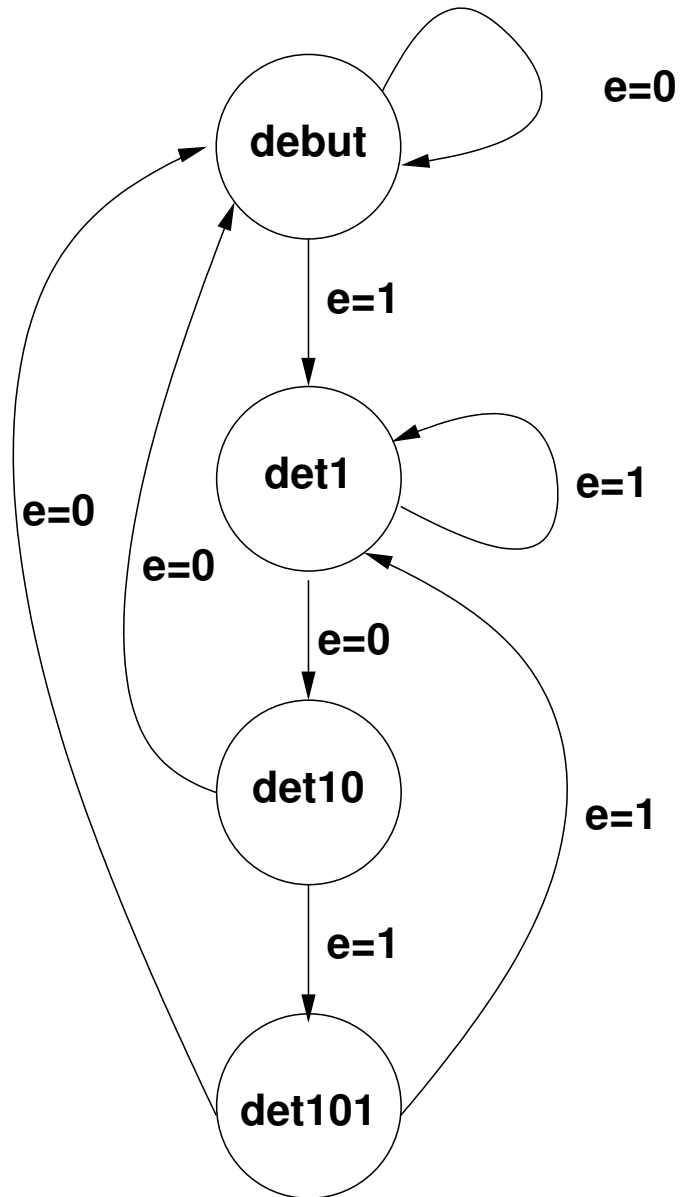
Machines à états

La conception de machines à états (FSM=Finite State Machine) est simplifiée par l'usage de VHDL.

La **méthodologie** usuelle est la suivante :

- 1 **Spécification** de la FSM
- 2 Représentation de la FSM par un **graphe d'états**
- 3 Implémentation en VHDL
 - ☆ La **mémorisation de l'état**
 - ☆ Les fonctions de **transition** et de **sortie**
- 4 Validation
- 5 Synthèse logique et placement/routage

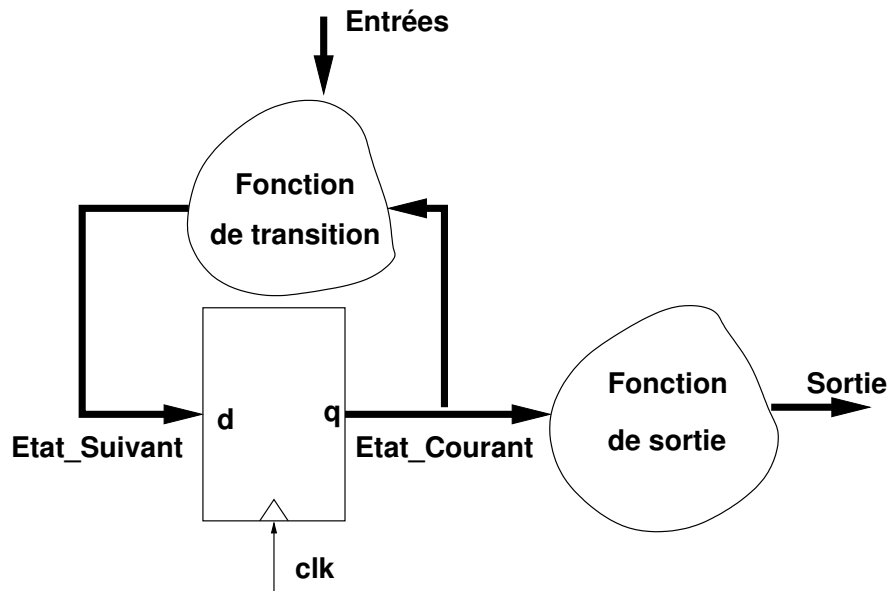
Graphe



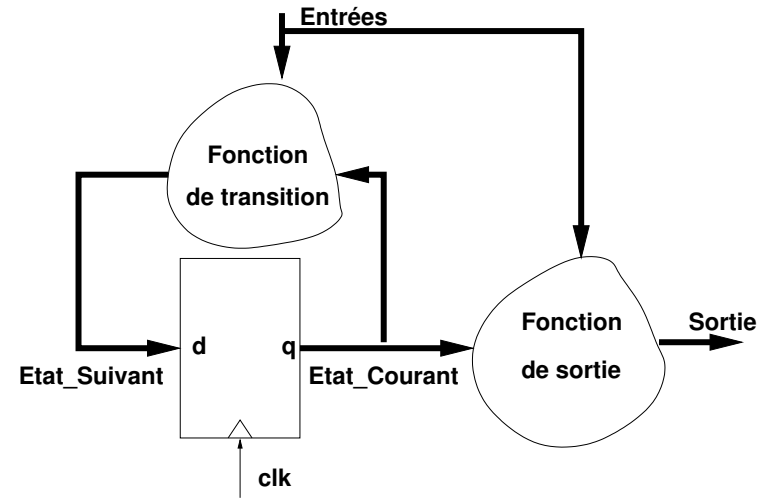
Description de l'entité

```
entity detection101 is
  port (
    reset : in std_logic;
    clk : in std_logic;
    e : in std_logic;
    s : out std_logic
  );
end detection101;
```

Architectures cibles



*Machine à états de **Moore***



*Machine à états de **Mealy***

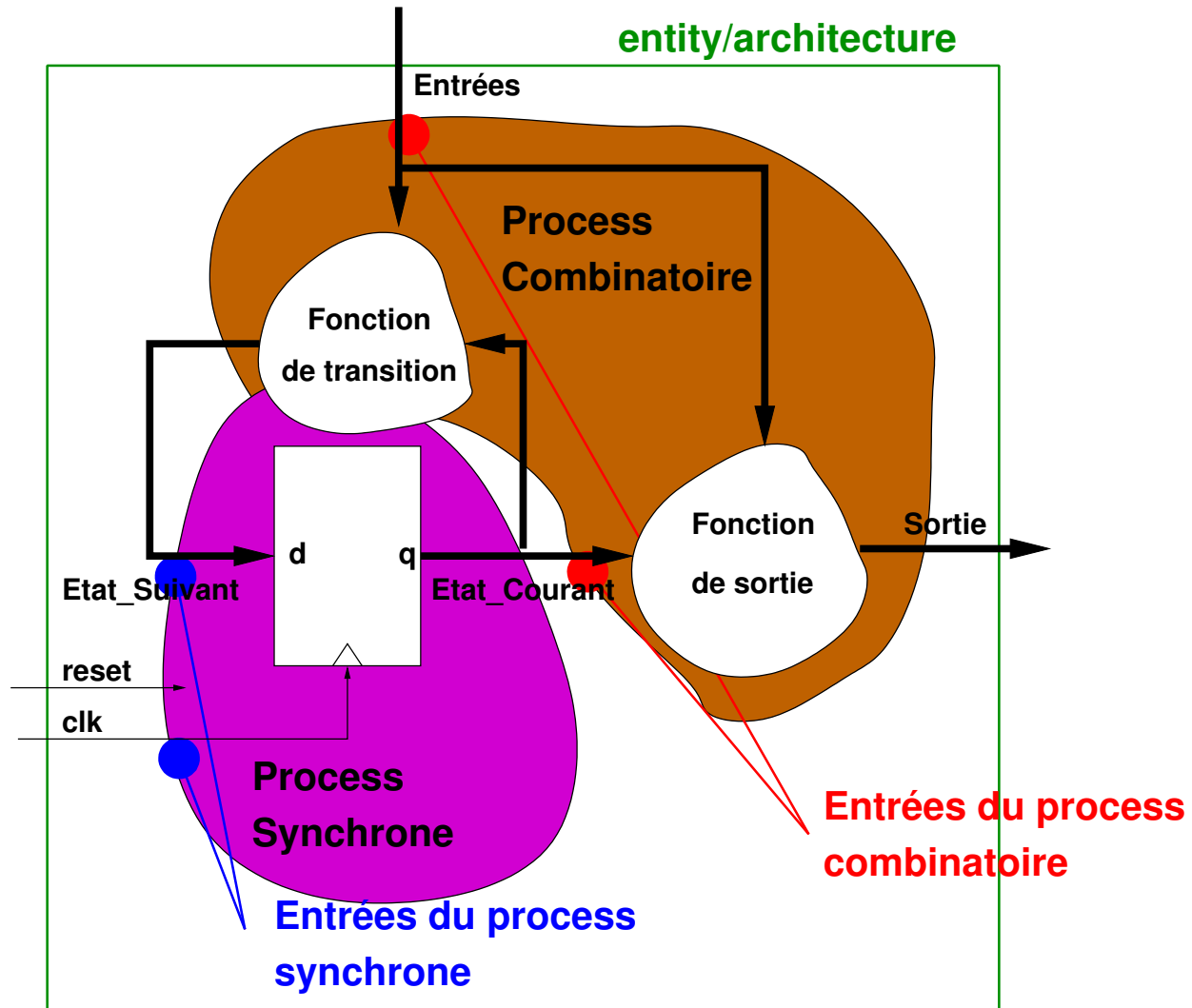
L'architecture est composée :

- ★ D'un élément de **mémorisation de l'état**
 - ➡ Un `process` **synchrone** qui met l'état courant à jour à chaque cycle d'horloge
- ★ Des **fonctions de transition et de sortie**
 - ➡ Un `process` **combinatoire** qui, pour chaque état, calcule :
 - L'état suivant en fonction de l'état et des entrées
 - La valeur de chaque sortie en fonction de l'état courant (**Moore**)



Dans le cas d'une machine de **Mealy**, une sortie peut dépendre de l'état courant et des entrées.

Représentation du modèle



Implémentation d'une FSM par deux `process`

Encodage

L'état de la FSM est stocké dans des bascules D selon un schéma d'encodage : binaire, one hot, etc

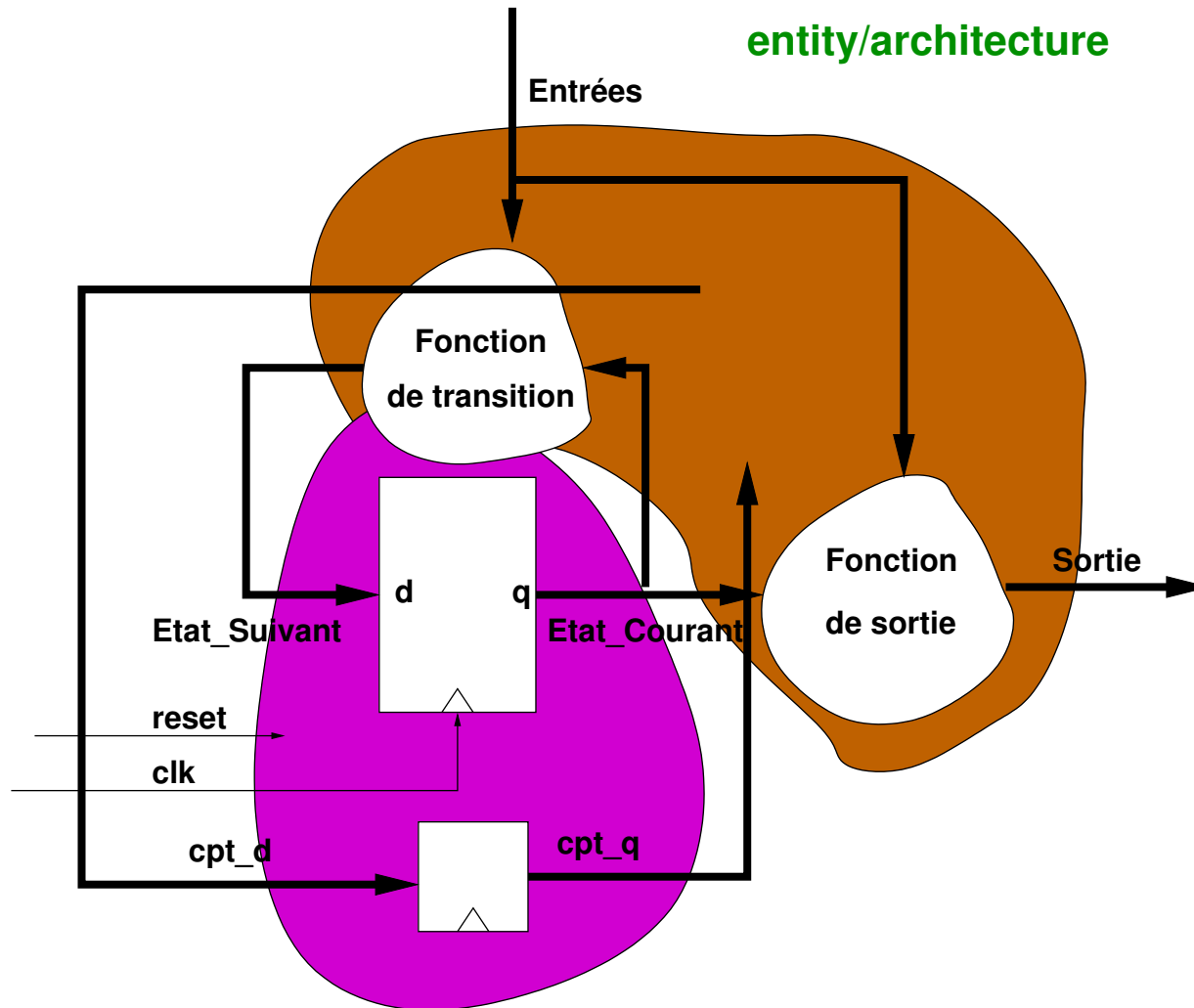
Afin de simplifier le codage de la FSM, l'**état** est représenté par un signal de **type énuméré**. La synthèse logique transforme l'énumération en mots binaires.

FSM : Le code VHDL

```
architecture RTL of detection101 is
  type state_type is (debut, det1, det10, det101);
  signal etat_suivant, etat_present : state_type;
begin
  etat_synchrone : process (clk)
  begin
    if clk'event and clk='1' then
      if reset='1' then
        etat_present<= debut;
      else
        etat_present <= etat_suivant
      end if;
    end if;
  end process etat_synchrone;
  calcul_sorties_et_etat_suivant : process (etat_present, e)
  begin
    case etat_present is
      when debut =>
        s<='0';
        if e='1' then
          etat_suivant <= det1;
        else etat_suivant <= debut;
        end if;
      when det1 =>
        s<='0';
    -- ...
    end case;
  end process calcul_sorties_et_etat_suivant;
end RTL;
```

Gestion des compteurs et registres

Des compteurs et registres peuvent être directement gérés au sein de la FSM.



FSM et compteurs : Le code VHDL

```
-- ...
signal cpt_q, cpt_d : ...
begin
  etat_synchrone : process (clk)
  begin
    if clk'event and clk='1' then
      if reset='1' then
        cpt_q <= 0;
      else
        cpt_q <= cpt_d;
      end if;
    end if;
  end process etat_synchrone;
  calcul_sorties_et_etat_suivant : process (etat_present, e, cpt_q)
  begin
    case etat_present is
      when debut =>
        s<='0';
        if cpt_q < 10 then
          cpt_d <= cpt_q+1;
          etat_suivant <= debut;
        else etat_suivant <= suite;
        end if;
      when suite =>
        cpt_d <= 14;
    end case;
  end process calcul_sorties_et_etat_suivant;
-- ...
```

Plan

- ☐ Notions fondamentales
- ☐ Structuration des modèles
- ☐ Validation
- ☐ VHDL RTL
- ☐ Implémentation des machines à états
- ☒ Types de données pour le RTL
 - ☐ Les bibliothèques “binaires”
 - ☐ Types de données classiques (ADA)
 - ☐ Boucles

Types de données

La conception de circuit numérique utilise :

❑ Les bibliothèques IEEE `std_logic` :

Avantages :

- . Théoriquement plus “proche” de la réalité électrique

Inconvénients :

- . Ambigus pour l'arithmétique
- . Difficile à vérifier
- . Très verbeux

❑ Les types de base ADA :

Avantages :

- . Code plus structuré
- . Vérification

Inconvénients :

- . Plus abstrait

Les `signal` ou `variable` peuvent être de tous les types.

Plan

- ☐ Notions fondamentales
- ☐ Structuration des modèles
- ☐ Validation
- ☐ VHDL RTL
- ☐ Implémentation des machines à états
- ☒ Types de données pour le RTL
 - ✓ Les bibliothèques “binaires”
 - ☐ Types de données classiques (ADA)
 - ☐ Boucles

Signaux “binaires”

`std_logic`

- ★ Prend les valeurs suivantes :
 - 1, 0 les valeurs logiques classiques
 - U pour “undefined” lorsque le signal n’a pas de valeur
 - X lorsqu’un signal est émis par deux `process` différents et qu’il y a un conflit
- et beaucoup d’autres que vous ne verrez pas

- ★ Exemple de déclaration :

```
signal toto, tata : std_logic;  
variable tito : std_logic;
```


Vecteur de bits

`std_logic_vector`

★ Est un tableau (vecteur) de `std_logic`

★ Exemple :

```
signal data : std_logic_vector(7 downto 0);
```

data est un vecteur de 8 bits

★ Affectation :

- . `toto <= "0010";`
- . `tata <= (2=>'1', others=>'0');`
- . `tati <= (others=>'0');`

★ Lecture :

- . `b <= toto(4);`
- . `v <= toto(6 downto 3);`

Arithmétique pour la conception RTL

La conception de circuit requiert l'utilisation de nombres codés sur un nombre de bits spécifiques. A cette fin on dispose des types :

- ❑ `signed` : sur lequel on fait des opérations en nombre **signé** (complément à deux).

Exemple : `signal a,b,c : signed(4 downto 0)`

`a,b,c` sont des nombres signés sur 5 bits, valant de -16 à +15

`a <= b + c;`

- ❑ `unsigned` : sur lequel on fait des opérations en nombre **non signé**

Exemple : `signal d,e,f : unsigned(4 downto 0)`

`d,e,f` sont des nombres signés sur 5 bits, valant de 0 à +31

`d <= e + f;`

`signed` **et** `unsigned` sont compatibles avec `std_logic_vector` .

Les bibliothèques pour le RTL

Pour utiliser les types RTL, il est nécessaire d'utiliser des bibliothèques spécifiques, déclarées en début de fichier.

```
-- pour les signaux binaires et leur arithmétique  
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_arith.all;  
use ieee.std_logic_unsigned.all;
```

Plan

- ☐ Notions fondamentales
- ☐ Structuration des modèles
- ☐ Validation
- ☐ VHDL RTL
- ☐ Implémentation des machines à états
- ☒ Types de données pour le RTL
 - ☐ Les bibliothèques “binaires”
 - ☒ Types de données classiques (ADA)
 - ☐ Boucles

Types de données classiques (ADA)

Il est possible d'utiliser les types de données de base de ADA.

Il ne sont pas prévus pour représenter des circuits MAIS la synthèse logique est capable de les synthétiser en signaux binaires équivalents.

Ils permettent d'écrire du code plus compacte, portable, générique, logique et **vérifiable** que les types "binaires".



Leur utilisation raisonnée se fait avec le souci d'être "compatible" avec la génération de circuits.

Type integer

❑ integer

Par défaut, synthétisé en `std_logic_vector` de 32 bits.

Avantage : permet toutes les opérations arithmétiques et est facilement lisible

❑ integer range a to b

Définit un entier dans l'intervalle $[a, b]$.

Synthétisé en `std_logic_vector` de taille $\log_2(b - a)$.

Avantage : le simulateur **vérifie** en permanence que la valeur est dans l'intervalle spécifié. A contrario, il n'y a pas de gestion du débordement d'opérations arithmétiques sur les types `std_logic_vector`.

Exemple :

```
signal a : std_logic;  
signal n : integer range 0 to 4;  
...  
a <= toto(n);
```

retourne le bit `n` du mot `toto`

Type boolean

❑ boolean

Type de base de ADA qui vaut `true` ou `false`

Il est synthétisé en un `std_logic` .

C'est le résultat des tests et le type attendu pour les structures de contrôle (`if` , ...)

Avantage : permet une écriture plus compacte et logique.

Exemple :

```
signal test : boolean;
```

```
...
```

```
test <= a = b;
```

test **vaut** `true` **lorsque** `a` **et** `b` **sont égaux**, `false` **sinon**.

Avec les `std_logic` on aurait écrit

```
signal test : std_logic;
```

```
...
```

```
if a=b then
```

```
test <= '1';
```

```
else
```

```
test <= '0';
```

```
end if;
```

Types énumérés

Les types énumérés sont la base de ADA et il est possible de déclarer ses propres types :

- ❑ `type toto_type is (selA, selBgg, ddd, det101);`

- ❑ Ils sont synthétisé en `std_logic_vector` de **taille optimisée**.

- ❑ **Avantage** : Est facilement lisible.

- ❑ **Utilisation** : Encodage des états de Machine à Etats, entrée de sélection des **multiplexeurs**

Tableaux

Les tableaux permettent de modéliser facilement plusieurs signaux et registres.

```
type mon_tableau is array range 0 to 15 of  
    std_logic_vector(31 downto 0);  
  
signal register_file : mon_tableau;
```

Déclare un tableau de 16 mots de 32 bits.

- ★ `register_file(3)` : le 4ème **mot** de 32 bits du tableau
- ★ `register_file(5)(12)` : le 13ème **bit** du 6ème **mot** du tableau

Il est possible de construire des tableaux de tous les types.

Types structurés

Les types structurés permettent d'englober plusieurs signaux de façon hiérarchique. Le type est constitué de champs et il est possible d'utiliser chaque champ séparément.

```
type mon_type_structure is
record
  a, b : std_logic_vector(4 downto 0);
  valid : boolean;
end record;
```

puis

```
signal toto : mon_type_structure;
...
if toto.valid then
c <= toto.a + toto.b;
else
c <= (others => '0');
...

```

Bibliothèques personnalisées

Il est possible de définir ses propres bibliothèques de types, constantes, fonctions et procédures.

Une bibliothèque est déclarée par

```
package nom_de_la_bibliotheque is
...
-- déclarations
type w32 is std_logic_vector(31 downto 0);
....
end nom_de_la_bibliotheque;
```

On peut l'utiliser dans d'autres fichiers VHDL :

```
use work.nom_de_la_bibliotheque.all;
```

Fonctions et procédures

Les fonctions et procédures peuvent être utilisés en VHDL-RTL.

Elles sont synthétisable à partir du moment ou leur code respecte les règles du VHDL-RTL

Plan

- ☐ Notions fondamentales
- ☐ Structuration des modèles
- ☐ Validation
- ☐ VHDL RTL
- ☐ Implémentation des machines à états
- ☒ Types de données pour le RTL
 - ☐ Les bibliothèques “binaires”
 - ☐ Types de données classiques (ADA)
 - ☒ Boucles

Boucles

Il y a deux types de boucles en VHDL :

- ❑ Les boucles d'**instanciation**, dans les `architecture`
Répète une instanciation en faisant varier un indice

```
for ...  
  generate  
    ...  
  end generate;
```

👉 Les instances sont concurrentes !

- ❑ Les boucles de **contrôle**, dans les `process`
Itère sur un indice ou une condition

```
<for|while> ...  
  loop  
    ...  
  end loop;
```


👉 Les itérations sont séquentielles !

Boucles de contrôle et RTL

Les boucles de contrôle ne sont synthétisables que si les bornes sont statiques.
Le circuit équivalent correspond à une mise à plat de la boucle.

Exemple : détection du premier 1 dans un vecteur

```
...  
ok := false;  
r := (others=>false);  
for i in 0 to v'high  
loop  
    if (not ok) and v(i) then  
        ok := true;  
        j := i;           -- j prend le numero du premier bit vrai  
    fi;  
    r(i) := ok;           -- valide les bit de r au-dela du premier  
end loop;
```

 Les boucles `while` ne sont pas synthétisables !

CEP

Notions de VHDL

S. Mancini

Plan Détaillé

✖ Notions fondamentales

- ☆ Flot de conception
- ☆ VHDL
- ☆ Problématique
- ☆ Solution proposée par VHDL
- ☆ Le Delta-cycle par l'exemple
- ☆ Notion de temps
- ☆ Description fonctionnelle
- ☆ Notion de process
- ☆ Notion de signal
- ☆ Exemple de signal
- ☆ Notion de variable
- ☆ Exemple de variable

✖ Structuration des modèles

- ☆ Modules et hiérarchie de conception
- ☆ Conception hiérarchique
- ☆ Description vue de l'extérieur
- ☆ Composant combinatoire simple

- ☆ Bascule D
- ☆ Additionneur signé
- ☆ Combinatoire et structure de contrôle
- ☆ Petit exercice
- ☆ Exemple de conception hiérarchique

✖ Validation

- ☆ Description de tests
- ☆ Description de tests

✖ VHDL RTL

- ☆ VHDL RTL ?
- ☆ Modélisation de la combinatoire
- ☆ Modélisation des bascules D
- ☆ Modélisation des mémoires
- ☆ Les 10 commandements du VHDL-RTL

✖ Implémentation des machines à états

- ☆ Machines à états
- ☆ Graphe

- ☆ Description de l'entité
- ☆ Architectures cibles
- ☆ Modélisation
- ☆ Représentation du modèle
- ☆ Encodage
- ☆ FSM : Le code VHDL
- ☆ Gestion des compteurs et registres
- ☆ FSM et compteurs : Le code VHDL

✖ Types de données pour le RTL

- ☆ Types de données
- Les bibliothèques "binaires"
 - ☆ Signaux "binaires"

- ☆ Vecteur de bits
- ☆ Arithmétique pour la conception RTL
- ☆ Les bibliothèques pour le RTL
- Types de données classiques (ADA)
 - ☆ Types de données classiques (ADA)
 - ☆ Type integer
 - ☆ Type boolean
 - ☆ Types énumérés
 - ☆ Tableaux
 - ☆ Types structurés
 - ☆ Bibliothèques personnalisées
 - ☆ Fonctions et procédures
- Boucles
 - ☆ Boucles
 - ☆ Boucles de contrôle et RTL