

# VHDL

VHSIC Hardware Description Language

PROCESS ET SÉQUENTIEL

© 2023-2024 Clément Foucher

Cours distribué sous licence libre 

BUT GEII Toulouse S5 ESE

# Process

LISTE DE SENSIBILITÉ

PROCESS SYNCHRONE

PROCESS ASYNCHRONE

SIMULTANÉITÉ DES ACTIONS

STRUCTURES DE CONTRÔLE

SOURCE DES SIGNAUX

EXERCICES

# Concept de process

- ▶ Rappel : un process est une suite d'instructions qui permet de décrire séquentiellement ce que l'on souhaite réaliser
  - ▶ Attention, les instructions ne sont pas *exécutées* séquentiellement (on n'est pas sur un processeur), mais permettent de décrire un certain comportement qui sera converti en un circuit logique
- ▶ On a déjà vu son utilisation dans le cadre des testbenches, mais elle n'était pas synthétisable
- ▶ Dans ce cours, nous allons étudier les process synthétisables, c'est-à-dire capables de générer un circuit logique

# Concept de process

- ▶ On parle parfois de « mode séquentiel » pour le contenu d'un process
  - ▶ Le mode séquentiel s'oppose au « mode concurrent », qui contient la partie de l'architecture qui n'est pas dans un process
- ▶ Un process est très abstrait, car contrairement au mode concurrent, on indique *ce que l'on souhaite faire* et non *le circuit à réaliser*
  - ▶ La transformation en circuit est effectuée par le synthétiseur
- ▶ Il est donc très important de suivre correctement la syntaxe, sinon on risque rapidement d'obtenir du code non synthétisable

PROCESS

# Liste de sensibilité

PROCESS SYNCHRONE

PROCESS ASYNCHRONE

SIMULTANÉITÉ DES ACTIONS

STRUCTURES DE CONTRÔLE

SOURCE DES SIGNAUX

EXERCICES

# Notion de liste de sensibilité

- ▶ La **liste de sensibilité** est indiquée entre parenthèses
- ▶ Elle indique les signaux dont un changement de valeur « déclenche l'évaluation » du process
  - ▶ Rappel abstraction : en dehors d'une simulation (sur un circuit réel), il n'y a pas réellement de déclenchement du calcul, c'est un circuit électronique, donc tout est réalisé simultanément

```
architecture ar of mon_compo is

begin

    process(<liste de sensibilité>)
    begin
        (...)
    end process;

end architecture;
```

# Notion de liste de sensibilité

- ▶ Il existe deux types de process
  - ▶ Un **process synchrone** contient *exactement* deux signaux dans la liste de sensibilité : l'horloge et le reset
  - ▶ Un **process asynchrone** indique dans sa liste de sensibilité *tous* les signaux qui sont *lus* dans le process
- ▶ Tout autre type de liste de sensibilité donne un code non synthétisable !

```
entity mon_compo is
port (clk      : in  std_logic;
      reset    : in  std_logic;
      I1       : in  std_logic;
      I2       : in  std_logic;
      O1       : out std_logic;
      O2       : out std_logic);
end entity;

architecture ar of mon_compo is
    signal S : std_logic;
begin

    process (clk, reset)
    begin
        (...)
    end process;

    process (I2, S)
    begin
        O2 <= S and I2;
    end process;

end architecture;
```

# Confusion dans la terminologie



- ▶ Notion de mode
  - ▶ mode concurrent (hors process)
  - ▶ mode séquentiel (dans les process)
- ▶ Mais il y a deux types de process
  - ▶ Process synchrone (décrit de la logique séquentielle, càd avec de la mémoire)
  - ▶ Process asynchrone (décrit de la logique combinatoire, càd sans mémorisation)
- ▶ Dans le « mode séquentiel », il est donc possible de décrire de la logique séquentielle, mais aussi de la logique combinatoire !
- ▶ Cela permet d'utiliser les opérateurs disponibles dans les process pour décrire du combinatoire

PROCESS

LISTE DE SENSIBILITÉ

# Process synchrone

PROCESS ASYNCHRONE

SIMULTANÉITÉ DES ACTIONS

STRUCTURES DE CONTRÔLE

SOURCE DES SIGNAUX

EXERCICES

# Process synchrone

10

- ▶ Rappel : dans un process synchrone, la **liste de sensibilité** contient l'horloge et le reset
- ▶ Le corps d'un process synchrone doit *obligatoirement* suivre la construction ci-contre
  - ▶ **Réinitialisation**
    - ▶ Le reset peut être actif à 0 ou à 1
    - ▶ La valeur de reset des signaux affectés n'est pas forcément 0
  - ▶ **Action sur front d'horloge**
    - ▶ Les actions qui doivent être réalisées lorsqu'un front montant arrive

```
entity mon_compo is
port(clk    : in  std_logic;
      reset : in  std_logic;
      O1    : out std_logic);
end entity;

architecture ar of mon_compo is
  signal S : std_logic;
begin

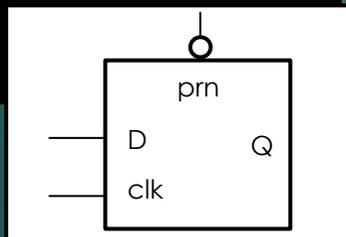
  -- Process de generation de O1
  process(clk, reset)
  begin
    if reset = '1' then
      O1 <= '0';
    elsif rising_edge(clk) then
      (... actions sur O1 ...)
    end if;
  end process;

end architecture;
```

# Exemple de synchrone

- ▶ Une bascule D (ou un registre en utilisant des vecteurs) est réalisée simplement avec le code ci-contre
- ▶ Variante avec preset actif à 0 ci-dessous

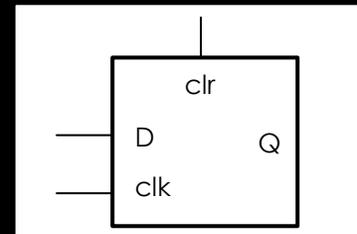
```
process(clk, prn)
begin
  if prn = '0' then
    Q <= '1';
  elsif rising_edge(clk) then
    Q <= D;
  end if;
end process;
```



```
entity D_FF is
port(clk : in  std_logic;
      clr : in  std_logic;
      D   : in  std_logic;
      Q   : out std_logic);
end entity;

architecture ar of D_FF is
begin

  process(clk, clr)
  begin
    if clr = '1' then
      Q <= '0';
    elsif rising_edge(clk) then
      Q <= D;
    end if;
  end process;
end architecture;
```



PROCESS

LISTE DE SENSIBILITÉ

PROCESS SYNCHRONE

# Process asynchrone

SIMULTANÉITÉ DES ACTIONS

STRUCTURES DE CONTRÔLE

SOURCE DES SIGNAUX

EXERCICES

# Process asynchrone : pourquoi faire ?

- ▶ Il paraît contre-intuitif d'utiliser un process pour décrire du combinatoire
- ▶ Un process asynchrone permet d'utiliser la puissance de la description séquentielle pour décrire du combinatoire
- ▶ Dans certains cas, il y a des avantages comme une meilleure lisibilité du code, ou une plus grande facilité à exprimer ce que l'on veut faire

# Process asynchrone

- ▶ Rappel : un process asynchrone indique dans sa [liste de sensibilité](#) tous les signaux qui sont lus dans le process
- ▶ Un process asynchrone ne dépend pas d'une horloge, et n'a pas de mémoire
- ▶ Basiquement, dans un process asynchrone, on peut faire la même chose qu'en mode concurrent
- ▶ Mais alors, à quoi ça sert ?

```
entity mon_compo is
port(I1 : in  std_logic;
      I2 : in  std_logic;
      O2 : out std_logic);
end entity;

architecture ar of mon_compo is
  signal S : std_logic;
begin

  S <= not I1;

  process(I2, S)
  begin
    O2 <= S and I2;
  end process;

end architecture;
```

# Process asynchrone

15

- ▶ Rappel : un process asynchrone indique dans sa [liste de sensibilité](#) tous les signaux qui sont lus dans le process
- ▶ Un process asynchrone ne dépend pas d'une horloge, et n'a pas de mémoire
- ▶ Basiquement, dans un process asynchrone, on peut faire la même chose qu'en mode concurrent
- ▶ Mais alors, à quoi ça sert ?
  - ▶ Dans un process, il est possible d'utiliser des constructions plus abstraites qu'en mode concurrent : les structures de contrôle

```
entity mon_compo is
port(I1 : in  std_logic;
      I2 : in  std_logic;
      O2 : out std_logic);
end entity;

architecture ar of mon_compo is
  signal S : std_logic;
begin

  S <= not I1;

  process(I2, S)
  begin
    if S = '1' and I2 = '1' then
      O2 <= '1';
    else
      O2 <= '0';
    end if;
  end process;

end architecture;
```

# Process asynchrone : restrictions

16

- ▶ En combinatoire, il n'y a pas de mémorisation
- ▶ Il faut donc obligatoirement donner une valeur à toutes les variables de sorties dans *tous* les cas
- ▶ Contre-exemple
  - ▶ Dans le cas ci-contre, si A vaut 1, la valeur de Y n'est pas définie
    - ▶ Quelle est la valeur de Y dans ce cas ? La valeur précédente ?
      - ▶ Ce n'est pas possible...
  - ▶ Ce process génère donc une mémoire implicite... sans horloge (latch)

```
-- Le process ci-dessous génère  
-- les signaux X et Y  
process(A, B)  
begin  
    if A = '1' then  
        X <= '1';  
    elsif B = '1' then  
        X <= '0';  
        Y <= '1';  
    end if;  
end process;
```

Pas bien !

# Process asynchrone : restrictions

17

- ▶ Le code ci-contre résout-il le problème ?
- ▶ Non !
  - ▶ Que se passe-t-il si A et B valent tout deux 0 ?
    - ▶ Pas de valeur définie pour X ni Y
  - ▶ Ce process génère donc toujours une mémoire implicite

Pas bien !

```
-- Le process ci-dessous génère
-- les signaux X et Y
process(A, B)
begin
    if A = '1' then
        X <= '1';
        Y <= '0';
    elsif B = '1' then
        X <= '0';
        Y <= '1';
    end if;
end process;
```

# Bonnes habitudes

- ▶ Une bonne habitude pour résoudre ce problème consiste à donner une *valeur par défaut*, au début des process asynchrone, à *tous* les signaux pilotés
- ▶ Si plusieurs valeurs « successives » sont assignées à un signal dans un process, la dernière valeur « gagne » (voir section suivante)

```
-- Le process ci-dessous génère
-- les signaux X et Y
process(A, B)
begin
    X <= '0';
    Y <= '0';
    if A = '1' then
        X <= '1';
        Y <= '0';
    elsif B = '1' then
        X <= '0';
        Y <= '1';
    end if;
end process;
```

# Quel circuit a-ton réalisé, ici ?

X et Y sont mis à zéro, si aucune instruction plus bas ne dit l'inverse (« Le dernier à parler gagne »)

Si A vaut 1, alors on donne une valeur à X et Y, si aucune instruction plus bas ne dit l'inverse

Si A ne vaut pas 1, mais que B vaut 1, on donne une valeur à X et Y, si aucune instruction plus bas ne dit l'inverse

```
-- Le process ci-dessous génère
-- les signaux X et Y
process(A, B)
begin
  X <= '0';
  Y <= '0';
  if A = '1' then
    X <= '1';
    Y <= '0';
  elsif B = '1' then
    X <= '0';
    Y <= '1';
  end if;
end process;
```

A	B	X	Y
1	*	1	0
0	1	0	1
0	0	0	0

PROCESS

LISTE DE SENSIBILITÉ

PROCESS SYNCHRONE

PROCESS ASYNCHRONE

# Simultanéité des actions

STRUCTURES DE CONTRÔLE

SOURCE DES SIGNAUX

EXERCICES

# Simultanéité des actions

- ▶ On ne le répètera jamais assez : le concept de séquentialité dans un process est abstrait
- ▶ En réalité, dans un circuit électronique, toutes les actions ont lieu simultanément
- ▶ Cela entraîne des effets parfois difficiles à appréhender, notamment sur la mise à jour de la valeur des signaux

# Mise à jour de la valeur des signaux

22

- ▶ Les valeurs de tous les signaux pilotés par le process sont mises à jour *simultanément* à la « fin » du process

```
-- Le process ci-dessous génère  
-- les signaux X et Y  
process (A)  
begin  
    X <= A;  
    Y <= X;  
end process;
```

Le changement de valeur de A « déclenche » l'évaluation du process

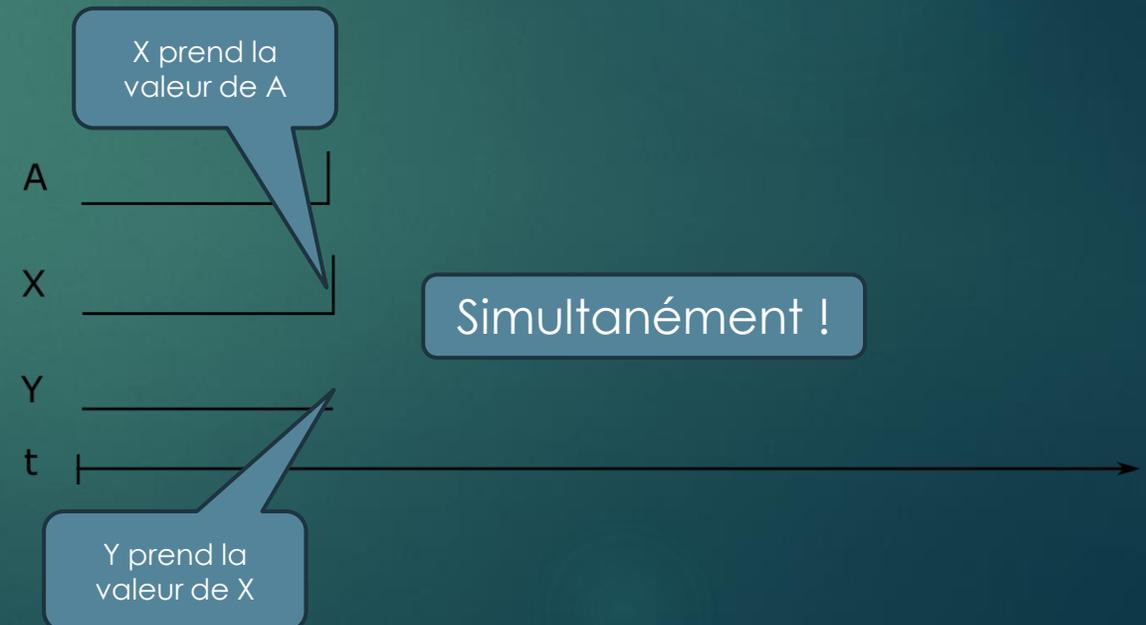


# Mise à jour de la valeur des signaux

23

- ▶ Les valeurs de tous les signaux pilotés par le process sont mises à jour *simultanément* à la « fin » du process
- ▶ Si on réutilise la valeur d'un signal modifié plus tôt dans le process, c'est donc la valeur qu'il avait au « début » du process
  - ▶ Dans le cas ci-contre, Y prend la valeur de X *telle qu'elle était au « début » du process*
  - ▶ X prend donc la valeur de A, mais Y prend la valeur de X *avant modification*

```
-- Le process ci-dessous génère  
-- les signaux X et Y  
process (A)  
begin  
    X <= A;  
    Y <= X;  
end process;
```

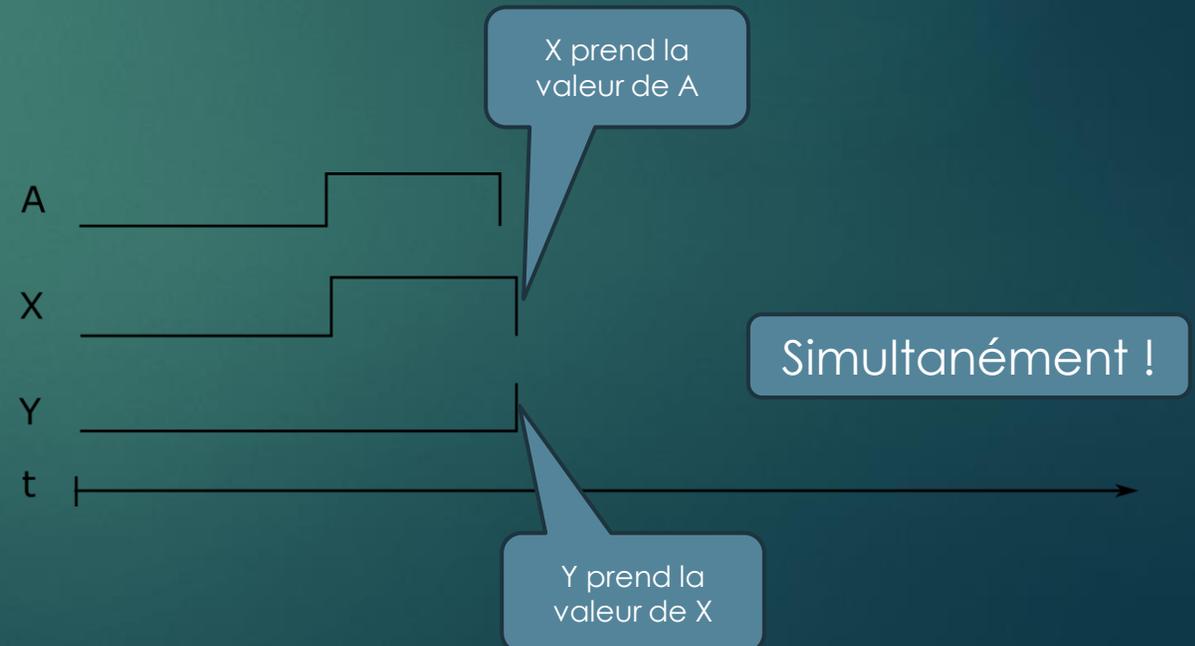


# Mise à jour de la valeur des signaux

24

- ▶ Les valeurs de tous les signaux pilotés par le process sont mises à jour *simultanément* à la « fin » du process
- ▶ Si on réutilise la valeur d'un signal modifié plus tôt dans le process, c'est donc la valeur qu'il avait au « début » du process
  - ▶ Dans le cas ci-contre, Y prend la valeur de X *telle qu'elle était au « début » du process*
  - ▶ X prend donc la valeur de A, mais Y prend la valeur de X *avant modification*

```
-- Le process ci-dessous génère  
-- les signaux X et Y  
process (A)  
begin  
    X <= A;  
    Y <= X;  
end process;
```

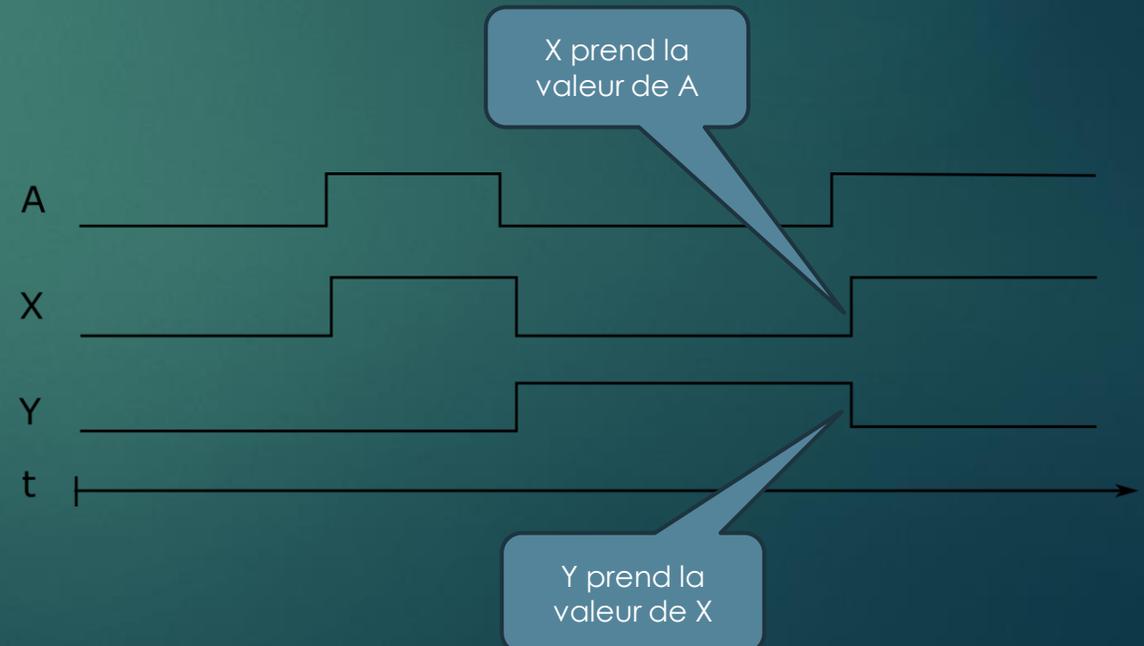


# Mise à jour de la valeur des signaux

25

- ▶ Les valeurs de tous les signaux pilotés par le process sont mises à jour *simultanément* à la « fin » du process
- ▶ Si on réutilise la valeur d'un signal modifié plus tôt dans le process, c'est donc la valeur qu'il avait au « début » du process
  - ▶ Dans le cas ci-contre, Y prend la valeur de X *telle qu'elle était au « début » du process*
  - ▶ X prend donc la valeur de A, mais Y prend la valeur de X *avant modification*
- ▶ Pour pallier à ce problème, Il existe le concept de variable (voir bonus)

```
-- Le process ci-dessous génère  
-- les signaux X et Y  
process (A)  
begin  
    X <= A;  
    Y <= X;  
end process;
```



PROCESS

LISTE DE SENSIBILITÉ

PROCESS SYNCHRONE

PROCESS ASYNCHRONE

SIMULTANÉITÉ DES ACTIONS

## Structures de contrôle

SOURCE DES SIGNAUX

EXERCICES

# Syntaxe du if

27

- ▶ Un if débute par

```
if <condition> then
```

- ▶ Et se termine par

```
end if;
```

- ▶ Notez le « ; » qui termine l'instruction if

- ▶ Il est possible de faire un ou plusieurs « sinon » à l'aide du mot-clé `elsif`

```
<condition> then
```

- ▶ Notez que le mot-clé `elsif` est en un seul mot !
- ▶ Écrire `else if` correspond en fait à ouvrir un nouveau if dans le else : il faudra alors aussi le fermer à l'aide d'un `end if`;
- ▶ Ce cas est alors équivalent à l'écriture ci-contre

```
process (clk, clr)
begin
    if clr = '1' then
        Q <= '0';
    elsif rising_edge(clk) then
        Q <= D;
    end if;
end process;
```

```
process (clk, clr)
begin
    if clr = '1' then
        Q <= '0';
    else if rising_edge(clk) then
        Q <= D; end if;
    end if;
end process;
```

```
process (clk, clr)
begin
    if clr = '1' then
        Q <= '0';
    else
        if rising_edge(clk) then
            Q <= D;
        end if;
    end if;
end process;
```

# Syntaxe du case

28

- ▶ Un case débute par  
`case <signal> is`
- ▶ Et se termine par  
`end case;`
- ▶ Notez le « ; » qui termine l'instruction case
- ▶ Chaque cas est introduit par la syntaxe `when <valeur> =>`
- ▶ On doit terminer le case par un `when others =>` pour tous les autres cas
  - ▶ Équivalent du default en C++

```
process (A)
begin
    case A is
        when "00" =>
            X <= '0';
        when "01" =>
            X <= '0';
        when "10" =>
            X <= '1';
        when others =>
            X <= '1';
    end case;
end process;
```

# Syntaxe du for ... loop

29

- ▶ Un for débute par  
`for <nom> in <range>`
- ▶ Et se termine par  
`end loop;`
- ▶ Notez le « ; » qui termine l'instruction loop

```
process (A)
begin
    for i in 0 to 15
        B(15-i) <= A(i);
    end loop;
end process;
```

Ici, `i` est  
automatiquement  
un integer

# Points d'attention sur le `for...loop`

30

- ▶ L'utilisation d'une `loop` dans un testbench ne pose aucun problème
  - ▶ Le code n'est de toutes façons déjà pas synthétisable !
- ▶ En revanche, dans un composant synthétisable, il faut faire très attention à ce que l'on fait avec un `for`
  - ▶ Dans un composant synthétisable, on utilisera le `for ... loop` *uniquement* pour itérer sur les indices d'un vecteur
  - ▶ Tous les autres usages risquent très rapidement de donner du code non synthétisable

PROCESS

LISTE DE SENSIBILITÉ

PROCESS SYNCHRONE

PROCESS ASYNCHRONE

SIMULTANÉITÉ DES ACTIONS

STRUCTURES DE CONTRÔLE

# Source des signaux

EXERCICES

# Source d'un signal

32

- ▶ On a vu dans le premier cours qu'on ne pouvait donner une valeur à un signal qu'à un seul endroit dans le code
- ▶ Un process est un ensemble indivisible d'instructions
  - ▶ On peut réaliser plusieurs affectations successives d'un même signal au sein d'un process (la dernière valeur gagne)

Plusieurs affectations de Q : pas de problème dans le même process

```
architecture ar of mon_compo is
begin

    -- Donne une valeur à Q
    process(A, B)
    begin
        Q <= '0';
        if A = '1' then
            Q <= entree1;
        elsif B = '1' then
            Q <= entree2;
        end if;
    end process;

end architecture;
```

Bien !

# Source d'un signal

33

- ▶ On a vu dans le premier cours qu'on ne pouvait donner une valeur à un signal qu'à un seul endroit dans le code
- ▶ Un process est un ensemble indivisible d'instructions
  - ▶ On peut réaliser plusieurs affectations successives d'un même signal au sein d'un process (la dernière valeur gagne)
  - ▶ Mais on NE PEUT PAS affecter le même signal dans plusieurs process, ni dans un process et une instruction concurrente

Plusieurs affectations de Q dans des process différents : interdit

```
architecture ar of mon_compo is
begin

    -- Donne une valeur à Q
    process(A, B, entree1, entree2)
    begin
        Q <= '0';
        if A = '1' then
            Q <= entree1;
        elsif B = '1' then
            Q <= entree2;
        end if;
    end process;

    -- Donne une valeur à Q
    process(force)
    begin
        if force = '1' then
            Q <= '1';
        end if;
    end process;

end architecture;
```

Pas bien !

PROCESS

LISTE DE SENSIBILITÉ

PROCESS SYNCHRONE

PROCESS ASYNCHRONE

SIMULTANÉITÉ DES ACTIONS

STRUCTURES DE CONTRÔLE

SOURCE DES SIGNAUX

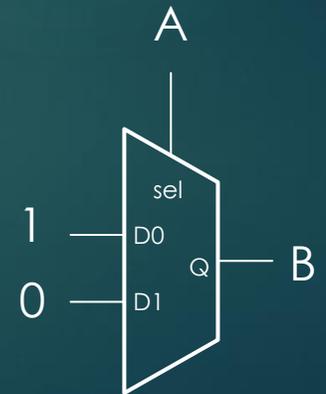
# Exercices

# Exercice

35

- ▶ Représenter le circuit électronique correspondant au process ci-contre

```
-- Tous les signaux sont des std_logic
process(A)
begin
    if A = '1' then
        B <= '0';
    else
        B <= '1';
    end if;
end process;
```

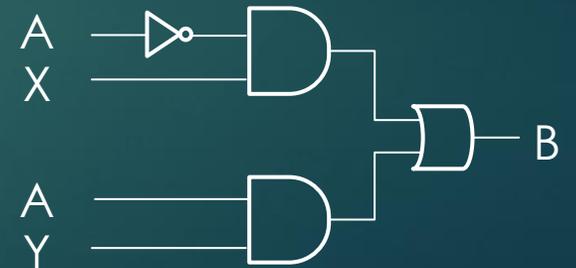
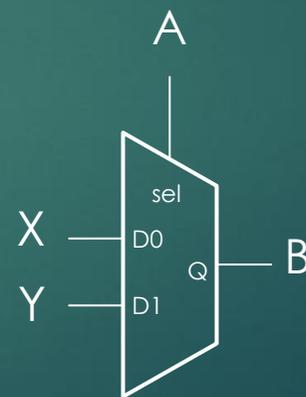


# Exercice

36

- ▶ Représenter le circuit électronique correspondant au process ci-contre

```
-- Tous les signaux sont des std_logic
process(A, X, Y)
begin
    if A = '0' then
        B <= X;
    else
        B <= Y;
    end if;
end process;
```



# Exercice

37

- ▶ Représenter le circuit électronique correspondant au process ci-contre

```
process (A)
begin
  for i in 0 to 15
    B(15-i) <= A(i);
  end loop;
end process;
```

A(15) — B(0)

A(14) — B(1)

⋮

A(1) — B(14)

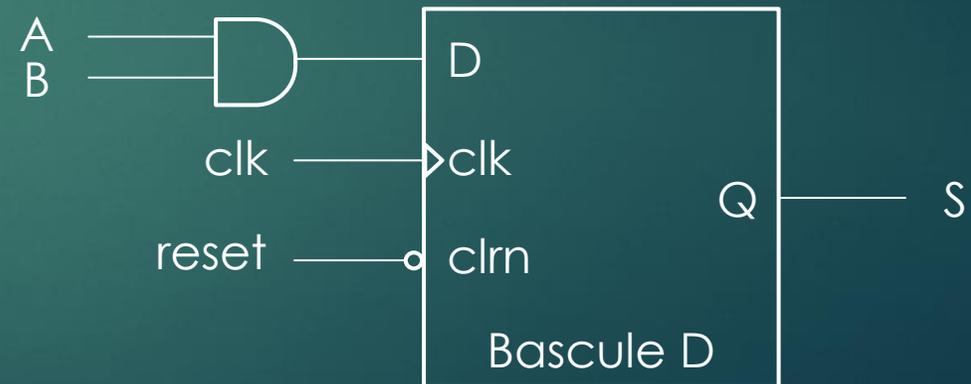
A(0) — B(15)

# Exercice

38

- ▶ Représenter le circuit électronique correspondant au process ci-contre

```
process(clk, reset)
begin
    if reset = '0' then
        S <= '0';
    elsif rising_edge(clk) then
        S <= A and B;
    end if;
end process;
```





# Bonus

# Variables

40



- ▶ Les variables sont des intermédiaires de calcul qui n'ont pas de réalité dans le circuit physique
- ▶ Elles sont utilisées pour indiquer au synthétiseur qu'il doit réaliser des calculs intermédiaires
- ▶ Cela permet de pallier au problème de la simultanéité de mise à jour des signaux
- ▶ On fait un calcul intermédiaire dans une variable, et le résultat de ce calcul est utilisé pour mettre à jour les signaux
- ▶ Notez l'opérateur d'affectation, qui est := pour les variables

```
-- Ce process n'a pas le résultat escompté
-- Note : tous les signaux sont des unsigned
process(A)
begin
    X <= A + 1;
    Y <= X + 1;
end process;
```

```
-- Ce process corrige le problème
process(A)
    variable temp : unsigned(7 downto 0);
Begin
    temp := A + 1;
    X <= temp;
    Y <= temp + 1;
end process;
```

# Label



- ▶ Il est possible de fournir un **label** (un nom) à un process
  - ▶ Cela peut notamment être utile lors de la simulation, lorsque l'on a de nombreux process dans un composant : cela permet à l'outil de vous indiquer dans quel process les évènements ont lieu

```
mon_process:process(<liste de sensibilité>)  
begin  
    (...)  
end process;
```

# Fronts d'horloge

42



- ▶ Il existe plusieurs types de fronts d'horloge
  - ▶ Montants
  - ▶ Descendants
- ▶ Dans la plupart des cas, on utilisera le front montant
- ▶ Fonctions correspondantes
  - ▶ `rising_edge (clk)`
  - ▶ `falling_edge (clk)`

# Pour aller plus loin

43

- ▶ Ressources
  - ▶ [Process](#)
  - ▶ [Différence entre variables et signaux](#)