Exercices de synthèse

Solution ALU

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity alu is
generic (N: integer:=8);
port(
   a,b       : in  std_logic_vector(N-1 downto 0);
   operation: in  std_logic_vector(2 downto 0);
   s         : out  std_logic_vector(N-1 downto 0)
   cout   : out std_logic);
end;

architecture archConc of alu is
signal res std_logic_vector(N downto 0);
begin
with operation select res <=
      std_logic_vector(unsigned('0' & a)+unsigned('0' & b)) when "000",
      std_logic_vector(unsigned('0' & a)-unsigned('0' & b)) when "001",
      '0' & a and '0' & b  when "010",
```

Exercices de synthèse

Solution ALU

```
      '0' & a or '0' & b  when "011",
      '0' & a xor '0' & b when "100",
      (others => '0') when others;
s <= res(N-1 downto 0);
cout <= res(N);
end archConc;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity alu_tb is
end;

architecture archi_alu_tb of alu_tb is
component alu
generic (N: integer:=8);
port(
   a,b       : in  std_logic_vector(N-1 downto 0);
   operation: in  std_logic_vector(2 downto 0);
```

Exercices de synthèse

Solution ALU

```
   s      : out  std_logic_vector(N-1 downto 0);
   cout : out std_logic);
end component;
signal a_t,b_t: std_logic_vector(7 downto 0);
signal s_t  : std_logic_vector(7 downto 0);
signal cout_t : std_logic;
signal op_t   : std_logic_vector(2 downto 0);
signal madd,msub,mand,mor,mxor   : boolean;
begin
alu_1: alu generic map(8) port map(a_t,b_t,op_t,s_t,cout_t);

op_t <= "000", "001" after 40 ns, "010" after 80 ns, "011" after 120
    ns, "100" after 160 ns;
a_t  <= x"01", x"02" after 10 ns, x"03" after 20 ns, x"04" after 30 ns,
        x"05" after 40 ns, x"06" after 50 ns, x"07" after 60 ns, x"08"
    after 70 ns,
        x"09" after 80 ns, x"0A" after 90 ns, x"0B" after 100 ns, x"0C"
    after 110 ns,
        x"0D" after 120 ns, x"0E" after 130 ns, x"0F" after 140 ns,
    x"10" after 150 ns;
```

Exercices de synthèse

Solution ALU

```
b_t  <= x"10", x"0F" after 10 ns, x"0E" after 20 ns, x"0D" after 30 ns,
        x"0C" after 40 ns, x"0B" after 50 ns, x"0A" after 60 ns, x"09"
    after 70 ns,
        x"08" after 80 ns, x"07" after 90 ns, x"06" after 100 ns, x"05"
    after 110 ns,
        x"04" after 120 ns, x"03" after 130 ns, x"02" after 140 ns,
    x"01" after 150 ns;
madd <= true when (op_t="000") else false;
msub <= true when (op_t="001") else false;
mand <= true when (op_t="010") else false;
mor <= true when (op_t="011") else false;
mxor <= true when (op_t="100") else false;
-- Vérification automatique du sommateur
--erreur volontaire dans la condition:
assert (now = 0 ns or (madd=true and
    (unsigned(a_t)+unsigned(b_t)=unsigned(s_t)))) report "sommateur
    incorrect!" & "au temps t=" & time'image(now) & CR & "a_t=" &
    integer'image(to_integer(unsigned(a_t))) & " b_t=" &
    integer'image(to_integer(unsigned(b_t))) & " s_t=" &
    integer'image(to_integer(unsigned(s_t))) severity ERROR;
```

Exercices de synthèse

Solution ALU

```
--version correcte:
assert (now = 0 ns or (madd=true and
     (unsigned(a_t)+unsigned(b_t)=unsigned(s_t))) or madd=false) report
     "cond=" & boolean'image(cond) & "sommateur incorrect!" & "au temps
     t=" & time'image(now) & CR & "a_t=" &
     integer'image(to_integer(unsigned(a_t))) & " b_t=" &
     integer'image(to_integer(unsigned(b_t))) & " s_t=" &
     integer'image(to_integer(unsigned(s_t))) severity NOTE;
end;
```

Sommaire

Types d'objets en VHDL

- Un objet en VHDL est une structure pouvant garder la valeur d'un type de donnée

4 types d'objets :

- signal
- variable
- constante
- et fichier (non-synthétisable)

Signal

- déclaré dans la partie déclarative d'une architecture
- `signal signal_name, signal_name, ... : data_type;`
- affectation d'un signal :

  `signal_name <= new_value;`
- tous les ports d'une entité sont considérés comme signaux
- interprétation physique : peuvent être considérés comme des fils conducteurs ou des "fils avec mémoire" (cas des sorties de registres)

Variable

- déclarée et utilisée au sein d'un `process`
- déclaration d'une variable :

```
variable variable_name, ... : data_type;
```

- affectation d'une variable :

```
variable_name := new_value;
```

- ne contient pas d'information temporelle
- utilisée comme dans un langage de programmation traditionnel
- pas d'équivalent en HW

Constante

- Valeur d'une constante ne peut pas être changée
- déclaration d'une constante :

  `constant const_name, ... : data_type:=value;`
- utilisée pour une meilleure lisibilité du code

```
constant BUS_WIDTH: integer:= 32;
constant BUS_BYTES: integer:= BUS_WIDTH/8;
```

Alias

- n'est pas un objet en sens propre du terme
- un nom alternatif pour un objet
- utilisé pour une meilleure lisibilité du code

```
signal word: std_logic(15 downto 0);
alias op: std_logic(6 downto 0) is word(15 downto 9);
alias reg1: std_logic(2 downto 0) is word(8 downto 6);
alias reg2: std_logic(2 downto 0) is word(5 downto 3);
alias reg3: std_logic(2 downto 0) is word(2 downto 0);
```

Sommaire

Process

- un ensemble d'instructions qui seront exécutées de manière séquentielle
- le `process` lui-même est concurrent
- peut être interprété comme une boite noire dont les entrées/sorties sont connues
- peut être synthétisé en circuit (mais pas forcément)
- deux types de `process` :
  - avec une liste de sensibilité
  - sans liste de sensibilité (avec un `wait`)

Process

`process` avec une liste de sensibilité :

```
process(liste_de_sensibilite)
  declarations;
begin
  instruction sequentielle;
  instruction sequentielle;
  ...;
end process;
```

- un `process` est actif lorsqu'un des signaux présents sur sa liste de sensibilité change de valeur
- si un `process` est actif, son contenu sera exécuté de manière séquentielle jusqu'à la fin

Process

- exemple :

```
signal a,b,c,y : std_logic;
process(a,b,c)
begin
 y <= a and b and c;
end process;
```

- Le comportement du modèle suivant est-il le même ?

```
signal a,b,c,y : std_logic;
process(a)
begin
 y <= a and b and c;
end process;
```

Process

- Pour un circuit combinatoire, toutes les entrées doivent être présentes dans la liste de sensibilité
- Le synthétiseur peut l'interpréter en émettant un avertissement

Process

`process` sans liste de sensibilité (avec l'instruction `wait`)

- `process` poursuit son exécution jusqu'à l'instruction `wait` où il est suspendu
- Utilisations de l'instruction `wait`
  - `wait on signals;`
  - `wait until boolean_expression;`
  - `wait for time_expression;`
- exemple :

```
signal a,b,c,y : std_logic;
process
begin
 y <= a and b and c;
 wait on a,b,c;
end process;
```

Process

- un `process` peut avoir plusieurs instructions `wait`
- un `process` avec une liste de sensibilité est préférée par les outils de synthèse
- Comment l'affectation des signaux/variables est effectuée au sein d'un `process` ?
- Pour un signal :

  ```
  signal_name <= new_value;
  ```

- Pas de différence par rapport à l'affectation en dehors d'un `process`
- Attention :

  → Au sein d'un `process`, un signal peut être affecté plusieurs fois ; seule la dernière affectation sera prise en compte

Process

- Exemple :

```
signal a,b,c,d,y : std_logic;
process(a,b,c,d)
begin                          --y_entry:=y
 y <= a or c;                  --y_exit :=a or c;
 y <= a and b;                 --y_exit :=a and b;
 y <= c and d;                 --y_exit :=c and d;
end process;                   --y <= y_exit;
```

- Cette description est identique à :

Process

```
signal a,b,c,d,y : std_logic;
process(a,b,c,d)
begin
 y <= c and d;
end process;
```

- Si les trois instructions étaient en dehors d'un `process` ?
- Affectation d'une variable

  `variable_name:=new_value;`
- L'affectation prend effet immédiatement ;
- Exemple :

Process

```
signal a,b,c: std_logic;
process(a,b,c)
  variable tmp: std_logic;
  begin
    tmp:='0';
    tmp:= tmp or a;
    tmp:= tmp or b;
    y <= tmp;
  end process;
```

- Si pour le même exemple, un signal au lieu d'une variable est utilisé ?

Process

```
signal a,b,c,tmp: std_logic;
process(a,b,c)
  begin
    tmp<='0';
    tmp<= tmp or a;
    tmp<= tmp or b;
    y <= tmp;
  end process;
```

- Cette description est identique à :

Process

```
signal a,b,c,tmp: std_logic;
process(a,b,c,tmp)
  begin
    tmp<= tmp or b;
    y <= tmp;
  end process;
```

- A l'intérieur d'un `process`, on peut utiliser l'instruction `if-then-else`

Process

```
if expr_1 then
  seq_instruction;
elsif expr2 then
  seq_instruction;
elsif expr3 then
  seq_instruction;
...
else
  seq_instruction;
end if;
```

- Tous les exemples utilisant les instructions concurrentes `when-else` vus précédemment peuvent être décrits dans un `process` en utilisant `if-then-else`

Process

- Pour obtenir le même circuit avec `when-else` et `if-then-else`, les deux structures doivent agir sur le même(s) signal(aux) de sortie
- L'instruction `if-then-else` est plus flexible puisqu'elle permet des structures imbriquées
- Exemple `when-else` :

```
sig <= value_1 when expr_1 else
       value_2 when expr_2 else
       value_3 when expr_3 else
       ...
       value_n;
```

- Exemple `if-then-else` :

Process

```
process(...)
begin
  if expr_1 then
    sig <= value_1;
  elsif expr_2 then
    sig <= value_2;
  elsif expr_3 then
    sig <= value_3;
   ...
  else
    sig <= value_n;
  end if;
end process;
```

Process

- Exemple du calcul de `max(a,b,c)`

En utilisant `if-then-else`

```
process(a,b,c)
begin
if (a>b) then
  if (a>c) then
    max <= a;
  else
    max <= c;
  end if;
else
  if (b>c) then
    max <= b;
  else
```

Process

```
    max <= c;
  end if;
end if;
end proces;
```

En utilisant `when-else`

```
signal ac_max, bc_max, max: std_logic;
...
ac_max <= a when (a>c) else c;
bc_max <= b when (b>c) else c;
max    <= ac_max when (a>b) else bc_max;
-- ou
max  <= a when ((a>b) and (a>c)) else
        c when (a>b) else
        b when (b>c) else
```

Process

```
c;
```

- Dans l'instruction `if-then-else`, seule la partie `then` est nécessaire → les instructions incomplètes sont autorisées
- Si un signal n'est pas affecté par omission, il garde sa valeur précédente → des *latchs* sont inférés

```
process(a,b)
begin
  if(a=b) then
    eq <= '1';
  end if;
end process;
```

```
process(a,b)
begin
  if(a=b) then
    eq <= '1';
  else
    eq <= eq;
  end if;
end process;
```

Process

Solution :

```
process(a,b)
begin
  if(a=b) then
    eq <= '1';
  else
    eq <= '0';
  end if;
end process;
```

Process

Un autre exemple : comparaison de deux valeurs

```
process(a,b)
begin
  if (a>b) then
    gt <= '1';
  elsif (a=b) then
    eq <= '1';
  else
    lt <= '1';
  end if;
end process;
```

Process

Solution 1 :

```
process(a,b)
begin
if (a>b) then
gt <= '1';
eq <= '0';
lt <= '0';
elsif (a=b) then
gt <= '0';
eq <= '1';
lt <= '0';
else
gt <= '0';
eq <= '0';
lt <= '1';
end if;
end process;
```

Solution 2 :

```
process(a,b)
begin
gt <= '0';
eq <= '0';
lt <= '0';
if (a>b) then
gt <= '1';
elsif (a=b) then
eq <= '1';
else
lt <= '1';
end if;
end process;
```

Process

- Dans un `process`, on utilise l'instruction `case` à la place de `with-select`
- Tout ce qui est valable pour les instructions séquentielles incomplètes de type `if-then-else` est également valable pour `case`

Instructions de boucle

```
for index in loop_range loop
 seq_instruction;
end loop;
```

- l'intervalle `loop_range` doit être statique
- Les indices doivent appartenir à l'intervalle (de gauche à droite)

Process

```
library ieee;
use ieee.std_logic_1164.all;
entity bit_xor is
   port(
      a, b: in std_logic_vector(3 downto 0);
      y: out std_logic_vector(3 downto 0)
   );
end bit_xor;

architecture demo_arch of bit_xor  is
   constant WIDTH: integer := 4;
begin
   process(a,b)
   begin
```

Process

```
      for i in (WIDTH-1) downto 0 loop
         y(i) <= a(i) xor b(i);
      end loop;
   end process;
end demo_arch;
```

Un autre exemple d'opération OU exclusif entre bits d'un vecteur :

```
library ieee;
use ieee.std_logic_1164.all;
entity reduced_xor_demo is
   port(
      a: in std_logic_vector(3 downto 0);
      y: out std_logic
   );
```

Process

```
end reduced_xor_demo;

architecture demo_arch of reduced_xor_demo is
   constant WIDTH: integer := 4;
   signal tmp: std_logic_vector(WIDTH-1 downto 0);
begin
   process(a,tmp)
   begin
      tmp(0) <= a(0);   -- boundary bit
      for i in 1 to (WIDTH-1) loop
         tmp(i) <= a(i) xor tmp(i-1);
      end loop;
   end process;
   y <= tmp(WIDTH-1);
end demo_arch;
```

Process

- "Dérouler" la boucle à réaliser
- A utiliser avec modération pour des instructions répétitives
- Exemple : opérations sur des bits d'un vecteurs

```
y(3) <= a(3) xor b(3);
y(2) <= a(2) xor b(2);
y(1) <= a(1) xor b(1);
y(0) <= a(0) xor b(0);
```

```
tmp(0) <= a(0);
tmp(1) <= a(1) xor tmp(0);
tmp(2) <= a(2) xor tmp(1);
tmp(3) <= a(3) xor tmp(2);
y      <= tmp(3);
```

Process

- Les instructions concurrentes
  - s'inspirent du matériel
  - sont claires et simple d'utilisation et peuvent être plaquées sur une structure matérielle
- Les instructions séquentielles
  - l'objectif est de décrire le comportement d'une structure matérielle
  - plus flexibles
  - peuvent être difficilement synthétisable
  - risque d'utilisation incorrecte pour la synthèse
- Penser matériel (*Think HW !*)
- Conception matérielle : différent d'une simple conversion d'un programme C vers VHDL

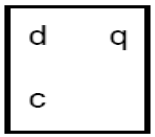
Sommaire

Circuits séquentiels

- Combinatoire vs séquentiel
  - Dans un circuit séquentiel, la ou les sorties sont une fonction des entrées et des états précédents des sorties
- Les principaux circuits séquentiels :
  - un verrou D (*D latch*)
  - une bascule D (*D flip-flop*)
  - une mémoire

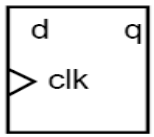
Circuits séquentiels

- Un verrou D est sensible au niveau de l'horloge
- Une bascule D est sensible au front (montant ou descendant) d'horloge



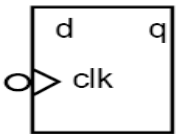
c	q*
0	q
1	d

(a) D latch



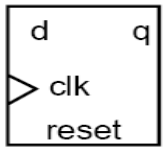
clk	q*
0	q
1	q
↓	d

(b) pos-edge triggered D FF



clk	q*
0	q
1	q
↓	d

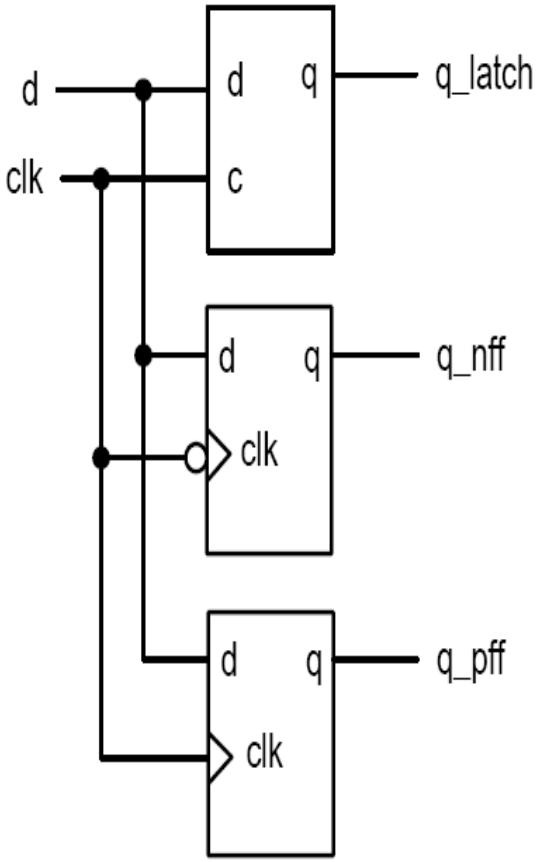
(c) neg-edge triggered D FF

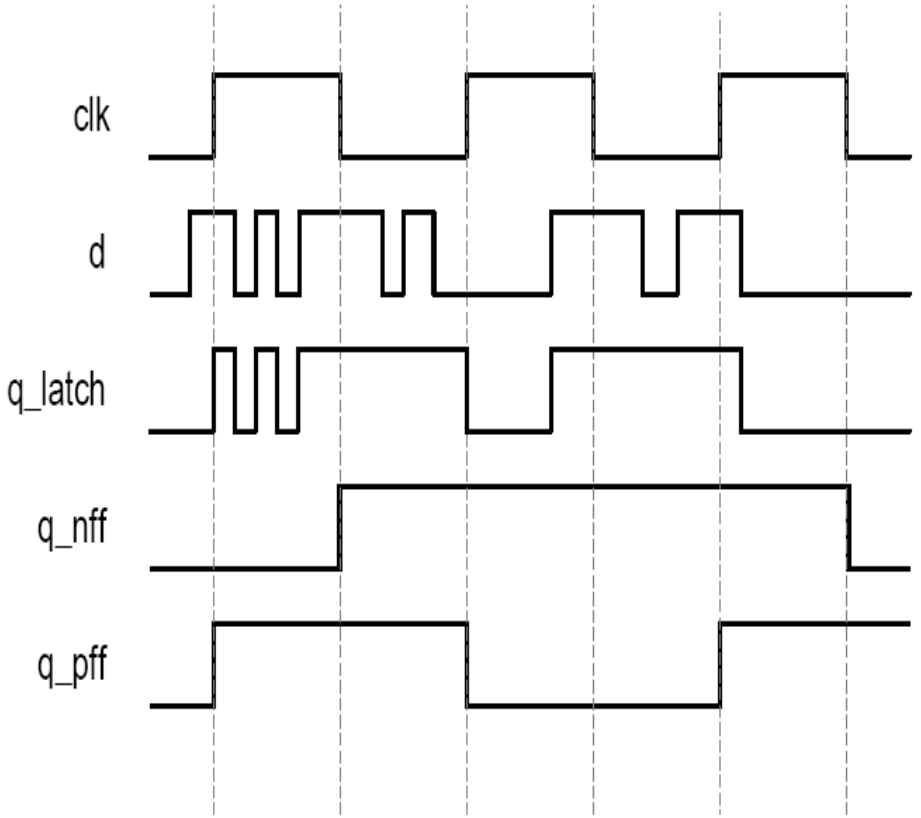


reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↓	d

(d) D FF with asynchronous reset

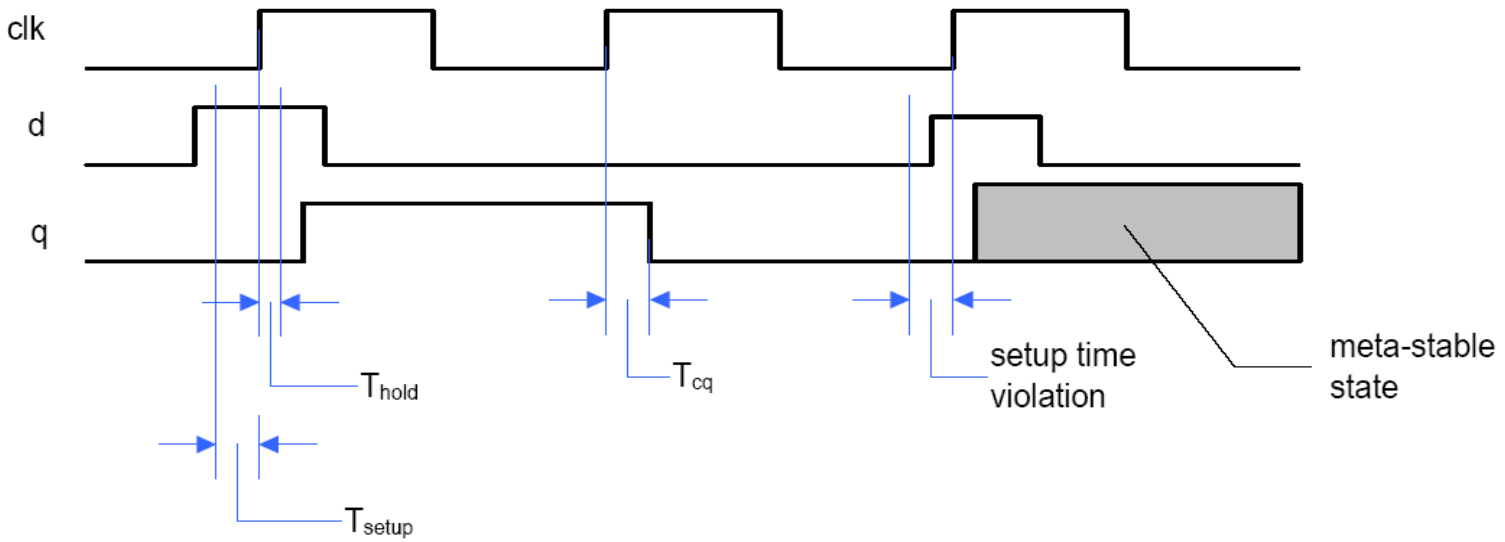
Circuits séquentiels





Circuits séquentiels

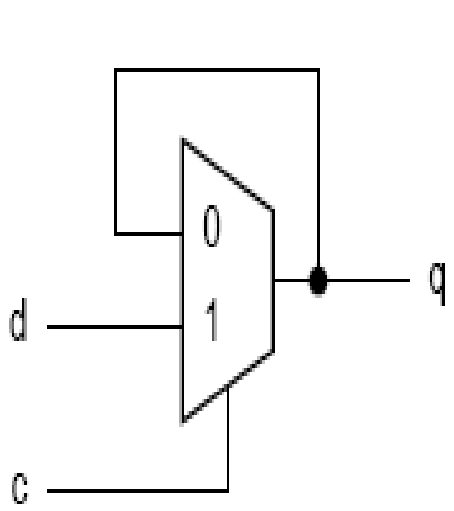
- Timing d'une bascule D
- *setup*, *hold*, *clock-to-q*

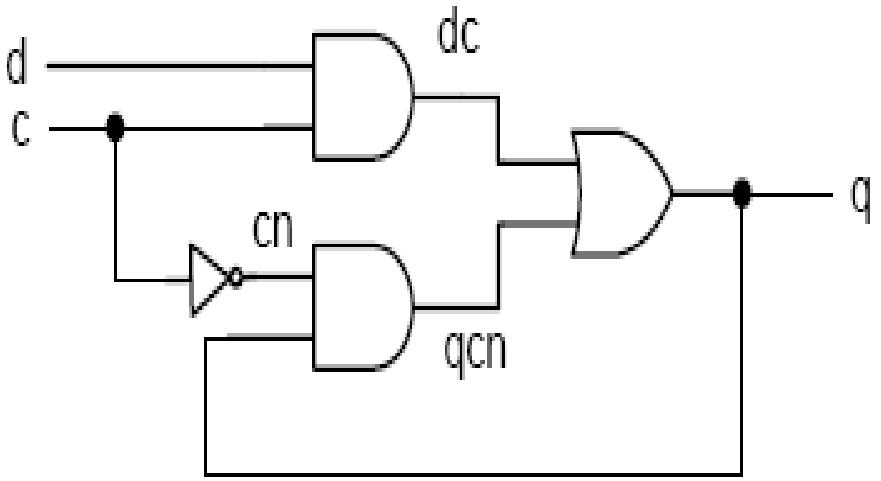


Circuits séquentiels

- Une bascule D ou un verrou D peuvent être réalisé à partir de portes logiques élémentaires
- Il s'agit de circuits combinatoires bouclés dont la conception est délicate (ils sont sensibles aux délais)
- On ne devrait pas les synthétiser à bas niveau *from scratch*
- On préférera les cellules prédéfinies ou inférées automatiquement

Circuits séquentiels





Un verrou D sensible à l'état haut

Circuits séquentiels

```
library ieee;
use ieee.std_logic_1164.all;
entity dlatch is
   port(
      c: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end dlatch;

architecture arch of dlatch is
begin
   process(c,d)
   begin
```

Circuits séquentiels

```
        if (c='1') then
            q <= d;
        end if;
    end process;
end arch;
```

Une bascule D sensible au front montant de l'horloge

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_1 is
    port(
        clk: in std_logic;
        d: in std_logic;
        q: out std_logic
```

Circuits séquentiels

```
    );
end dff_1;

architecture arch of dff_1 is
begin
    process(clk)
    begin
        if (clk'event and clk='1') then
        -- if rising_edge(clk) then
            q <= d;
        end if;
    end process;
end arch;
```

Circuits séquentiels

Exercices

1. Décrire une bascule D avec une remise à zéro (*reset*) asynchrone.
2. Rajouter à la bascule D précédente une remise à 1 (*preset*) asynchrone
3. Rajouter à la bascule D précédente un signal d'activation (*enable*) synchrone
4. Décrire en VHDL un registre 8 bits avec un *reset*
5. Décrire en VHDL un compteur générique (8 bits par défaut) avec chargement parallèle et *reset* synchrone