Exercise Book

ArchOrd (I)

Methodology :

- Define/understand desired behaviour
- Draw timing diagram
- Define entity
- Define block diagram and identify sequential components
- Write top-level architecture (often structural) and sub-level architecture (typically behavioural, RTL, ...)
- Write test bench
- Simulate and validate (correct bugs...)
- Synthesize
- Verify synthesis result versus expectations

Serial to Parallel Converter:

- **8 bits are received sequentially on a single signal (least significant bit first, bit 0)**
- **A start signal indicates bit 0**
- **As soon as the 8 bits are received, the byte is output in parallel from the module**
- **The module also outputs a valid signal to indicate that the output is ready**
- **A clock signal synchronizes the operation**

- **Interface:**
  - `std_logic` **or** `std_logic_vector`
- **Inputs:**
  - **Reset**
  - **Clk**
  - **DataIn**
  - **Start**
- **Outputs:**
  - **DataOut (8 bits)**
  - **Valid**

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

entity StoPConv is
  port (Clk, Reset, Start, DataIn : in std_logic;
        DataOut : out std_logic_vector(7 downto 0);
        Valid : out std_logic);
end StoPConv;

architecture synth of StoPConv is
  signal SIPOContenu : std_logic_vector(7 downto 0);
  signal ValidInt : std_logic;
  signal Cnt : integer range 0 to 8;

begin

  SIPO: process (Clk)
  begin
    if (Clk'event and Clk='1') then
      if (Reset='1') then
        SIPOContenu <= (others => '0');
      elsif (ValidInt='0') then
        SIPOContenu <= SIPOContenu(6 downto 0) & DataIn;
      end if;
    end if;
  end process;

  LatchOut: process (Reset, SIPOContenu, ValidInt)
  begin
    if (Reset='1') then
      DataOut <= (others => '0');
    elsif (ValidInt='1') then
      DataOut <= SIPOContenu;
    end if;
  end process;
```

```
Counter: process (Clk)
  begin
    if (Clk'event and Clk='1') then
      if (Start='1') then
        Cnt <= 0;
      elsif (Cnt /= 7) then
        Cnt <= Cnt + 1;
      end if;
    end if;
  end process;

  CounterValid: process (Cnt)
  begin
    ValidInt <= '0';
    if (Cnt = 7) then
      ValidInt <= '1';
    end if;
  end process;

  Valid <= ValidInt;

end synth;
```

Pattern Recognition:

- **A flow of bits are received sequentially on a single signal**
- **A 8-bit pattern to identify is received in parallel; a "load" signal indicates a new pattern**
- **The module outputs a "found" signal to indicate that the last 8 bits received serially are identical to the pattern**
- **A clock signal synchronizes all operations**

- **Interface:**
  - `std_logic` **or** `std_logic_vector`
- **Inputs:**
  - **Reset**
  - **Clk**
  - **Pattern (8 bits)**
  - **Load**
  - **DataIn**
- **Outputs:**
  - **Found**

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

entity Pattern is
  port (Clk, Reset, Load, DataIn : in std_logic;
        Pattern : in std_logic_vector(7 downto 0);
        Found : out std_logic );
end Pattern;

architecture synth of Pattern is
  signal SIPOContenu : std_logic_vector(7 downto 0);
  signal RegContenu : std_logic_vector(7 downto 0);

begin

  SIPO: process (clk)
  begin
    if (Clk'event and Clk='1') then
      if (Reset='1') then
        SIPOContenu <= (others => '0');
      else
        SIPOContenu <= SIPOContenu(6 downto 0) & DataIn;
      end if;
    end if;
  end process;

  Reg: process (Clk)
  begin
    if (Clk'event and Clk='1') then
      if (load='1') then
        RegContenu <= Pattern;
      end if;
    end if;
  end process;

  Comp: process (SIPOContenu, RegContenu)
  begin
    Found <= '0';
    if (SIPOContenu = RegContenu) then
      Found <= '1';
    end if;
  end process;

end synth;
```

Programmable Counter:

- Counter of <u>falling</u> edges from 0 to any number (<=15) specified as follows
- A 4-bit word signals the highest number N to be reached after which the counter restarts from 0; a "load" signal indicates a new value
- The module outputs a "zero" signal when it restarts; the "zero" signal is thus active for one clock cycle every N+1 cycles

- Interface:
  - std_logic or std_logic_vector
- Inputs:
  - Reset
  - Clk
  - MaxCount (4 bits)
  - Load
- Outputs:
  - Count (4 bits)
  - Zero

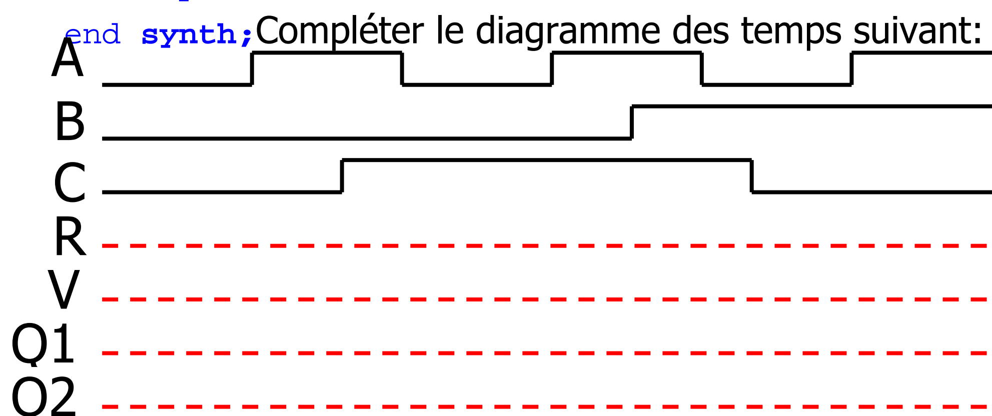
```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Counter is
  port (Clk, Reset, Load : in std_logic;
        MaxCount : in std_logic_vector(3 downto 0);
        Count : out std_logic_vector(3 downto 0);
        Zero : out std_logic );
end Counter;

architecture synth of Counter is
  signal RegContenu : std_logic_vector(3 downto 0);
  signal CountContenu : std_logic_vector(3 downto 0);
  signal CountReset, ResetInt : std_logic;

Begin
  Reg: process (Clk)
  begin
    if (Clk'event and Clk='0') then
      if (load='1') then
        RegContenu <= MaxCount;
      end if;
    end if;
  end process;

Counter: process (Clk)
  begin
    if (Clk'event and Clk='0') then
      if (CountReset='1') then
        CountContenu <= (others=>'0');
      else
        CountContenu <= CountContenu + 1;
      end if;
    end if;
  end process;

  Count <= CountContenu;

Comp: process (RegContenu, CountContenu)
  begin
    ResetInt <= '0';
    if (RegContenu = CountContenu) then
      ResetInt <= '1';
    end if;
  end process;

  CountReset <= Reset or ResetInt;
```

```
  CompZero: process (CountContenu)
  begin
    Zero <= '0';
    if (CountContenu = 0) then
      Zero <= '1';
    end if;
  end process;

end synth;
```

Dessiner le système représenté par le code VHDL suivant :

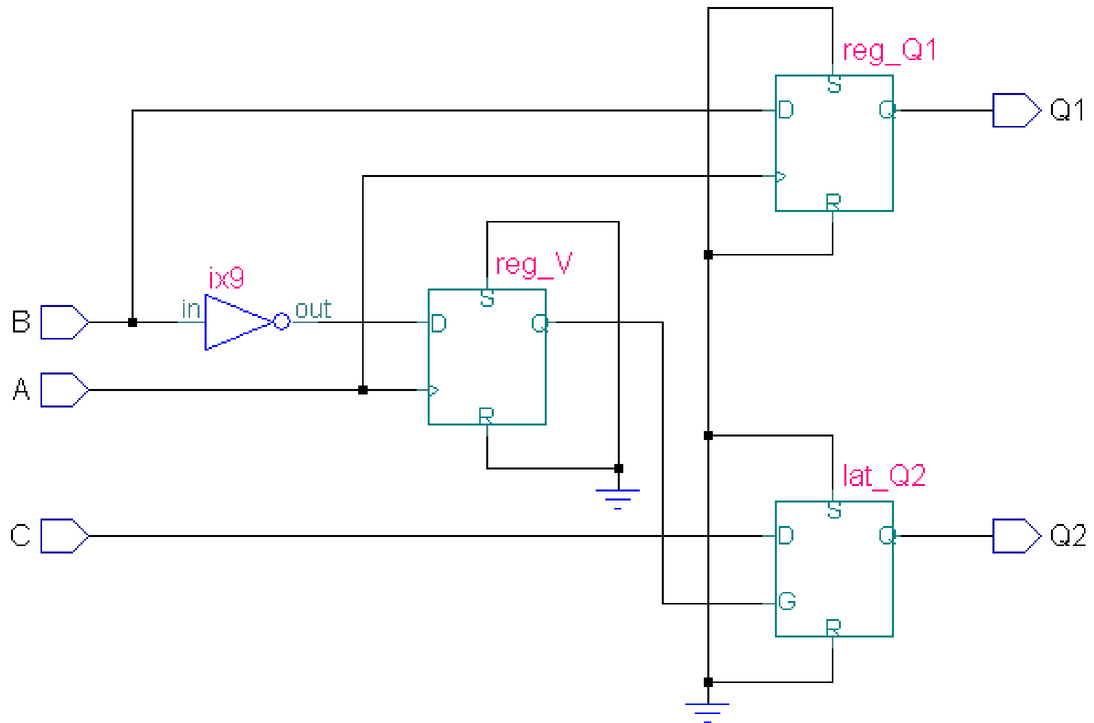
```
library IEEE;
use IEEE.std_logic_1164.all;

entity toto is
  port (A, B, C : in  std_logic;
        Q1, Q2 : out std_logic);
end toto;

architecture synth of toto is
  signal     V, R : std_logic;
begin
  process (V, C)
  begin
    if (V='1‘) then
      Q2 <= C;
    end if;
  end process;

  R <= B xor C;

  process (A)
  begin
    if (A'event and A='1') then
      Q1 <= C;
      V <= R;
    end if;
  end process;
end synth;
```

Compléter le diagramme des temps suivant:



Dessiner le système représenté par le code VHDL suivant:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity ckt is
  port (A, B, C : in std_logic;
        Q1, Q2 : out std_logic);
end ckt;

architecture toto of ckt is

begin

process (C, A, B)
  variable V : std_logic;
begin
  if V='1'
    then Q2 <= C;
  end if;
  if A'event and A='1'
    then Q1 <= B;
    V := not B;
  end if;
end process;

end toto;
```


reg_Q1
D
Q
S
R
Q1
reg_V
ix9
in
out
B
A
C
D
Q
S
R
lat_Q2
D
Q
G
S
R
Q2

Le but de cet exercice est de développer un composant effectuant les opérations de rotation. Afin de simplifier le travail, nous nous limiterons à des rotations d'un maximun de trois positions vers la droite ou vers la gauche, et nous travaillerons avec des mots de huit bits (huit bits d'entrée, $X_7,\ldots,X_0$, et huit bits de sortie, $Q_7,\ldots,Q_0$).

❶ L'opérateur est décomposé en huit tranches identiques, chacune calculant un bit Q_i du résultat. La sortie Qi dépend de sept bits d'entrée: X_{i+3}, X_{i+2}, X_{i+1}, X_i, X_{i-1}, X_{i-2} et X_{i-3}. Trois bits de contrôle sont nécessaires: RL indique le sens de la rotation; S_1 et S_0 codent le nombre de positions de la rotation. Le tableau 1 résume le fonctionnement d'une tranche.
On demande de dessiner le circuit correspondant en utilisant uniquement les multiplexeurs à deux entrées définis par le composant VHDL ci-dessous:

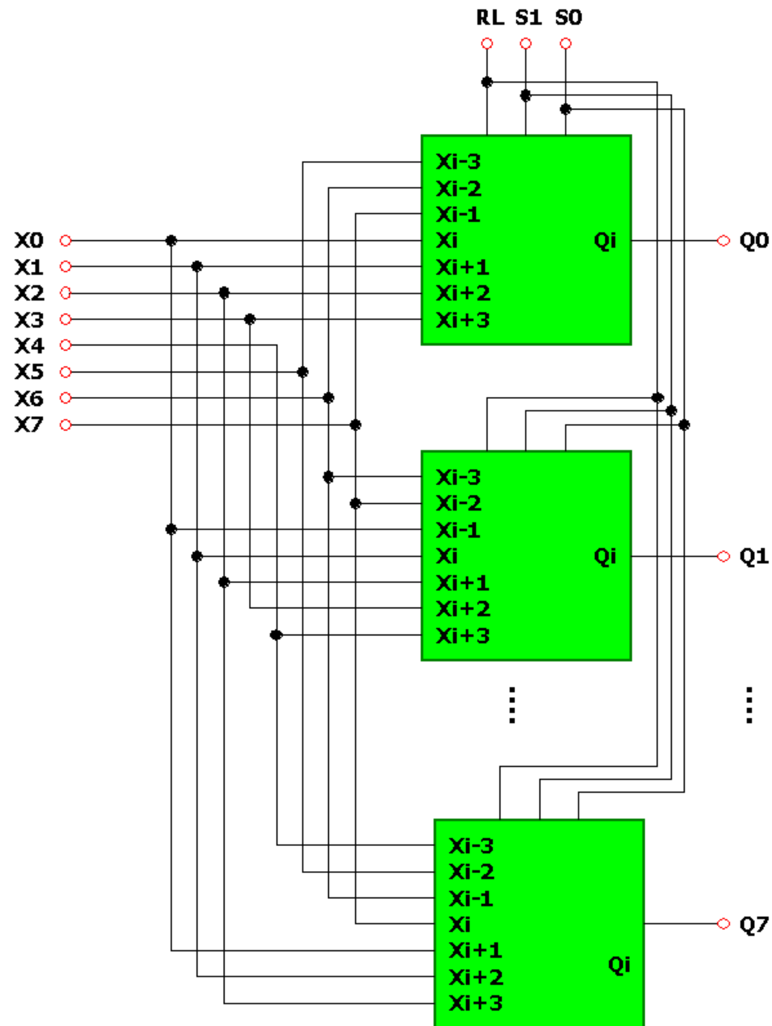
```
component mux_2x1
  port (
    D0 : in std_logic;
    D1 : in std_logic;
    S  : in std_logic;
    Q  : out std_logic); -- D0 si S=0; D1 sinon
end component;
```

RL	S_1	S_0	Q_i	Remarques
0	0	0	X_i	Aucune rotation
0	0	1	X_{i-1}	Rotation à gauche d'une position
0	1	0	X_{i-2}	Rotation à gauche de deux positions
0	1	1	X_{i-3}	Rotation à gauche de trois positions
1	0	0	X_i	Aucune rotation
1	0	1	X_{i+1}	Rotation à droite d'une position
1	1	0	X_{i+2}	Rotation à droite de deux positions
1	1	1	X_{i+3}	Rotation à droite de trois positions

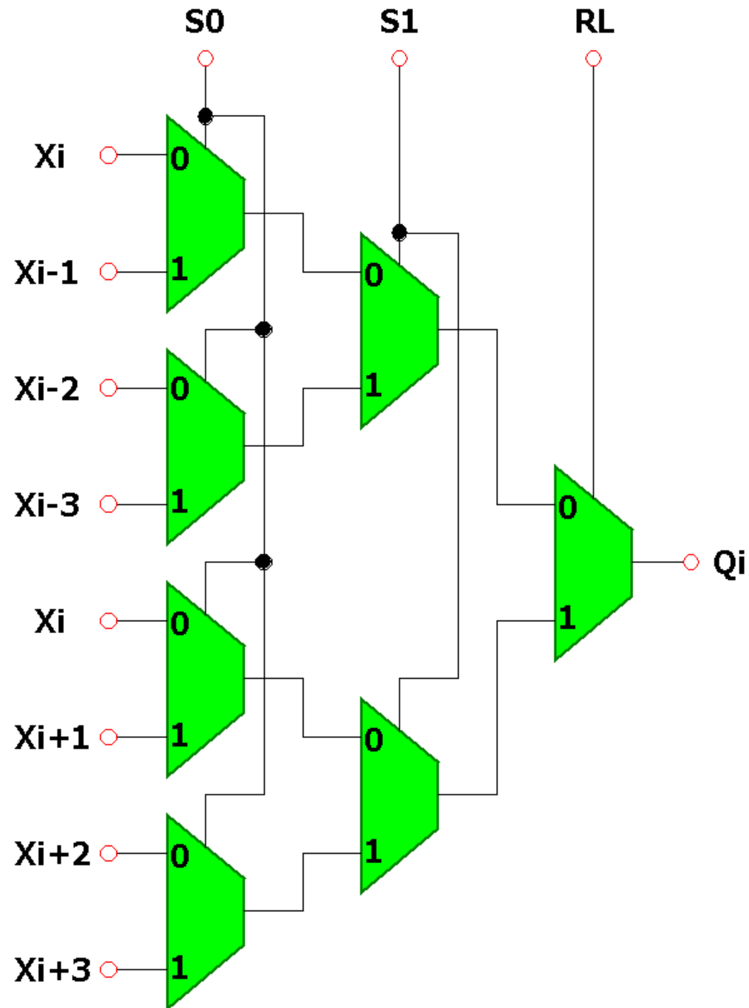
Tableau 1

❷ Après avoir donné une description VHDL du composant mux_2x1, écrire une architecture VHDL structurelle correspondant au circuit dessiné au point ❶.

❖ *Schéma logique global:*

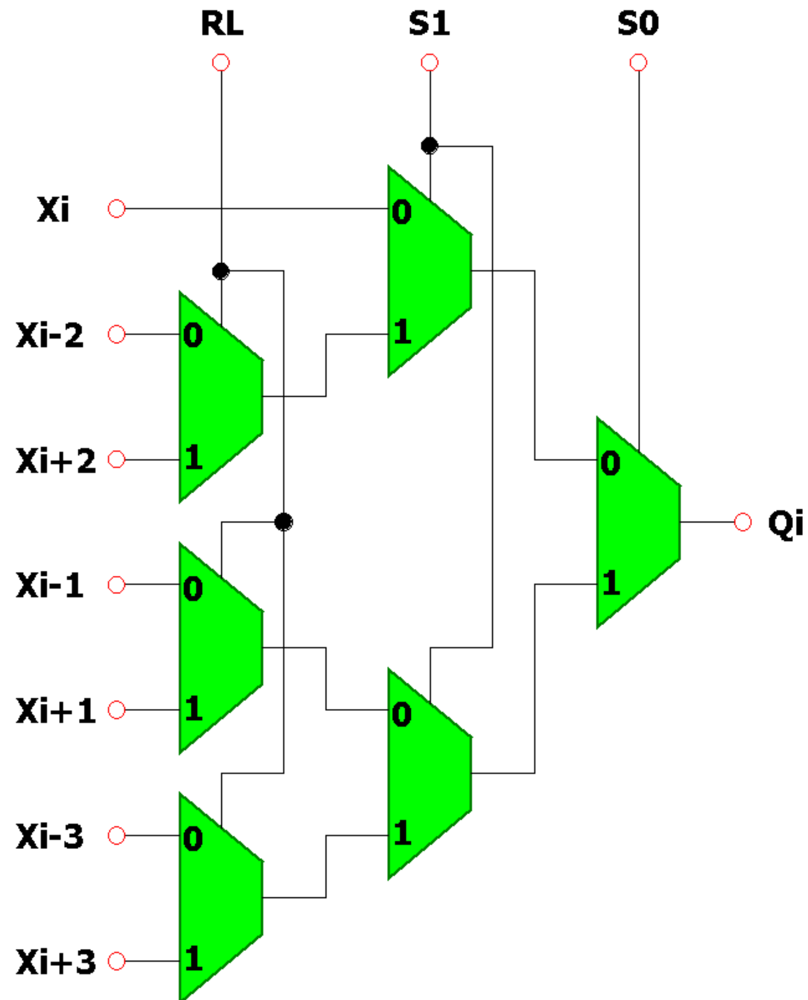


❖ *Schéma logique d'un composant:*



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY mux_2x1 IS
  port (D0, D1, S : in std_logic;
        Q : out std_logic );
END mux_2x1;

ARCHITECTURE synth OF mux_2x1 IS
BEGIN
   process (D0, D1, S)
   begin
      if S='0'
         then Q <= D0;
         else Q <= D1;
      end if;
   end process;
END synth;


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY tranche IS
   port (S0, S1, RL : in std_logic;
         X : in std_logic_vector(6 downto 0);
         Q : out std_logic );
END tranche;

ARCHITECTURE struct OF tranche IS

component mux_2x1
   port (D0, D1, S : in std_logic;
         Q : out std_logic );
end component;

signal Sa1, Sa2, Sa3, Sa4, Sb1, Sb2 : std_logic;

BEGIN
   a1: mux_2x1 port map (X(3), X(2), S0, Sa1);
   a2: mux_2x1 port map (X(1), X(0), S0, Sa2);
   a3: mux_2x1 port map (X(3), X(4), S0, Sa3);
   a4: mux_2x1 port map (X(5), X(6), S0, Sa4);

   b1: mux_2x1 port map (Sa1, Sa2, S1, Sb1);
   b2: mux_2x1 port map (Sa3, Sa4, S1, Sb2);
   c1: mux_2x1 port map (Sb1, Sb2, RL, Q);
END struct;
```

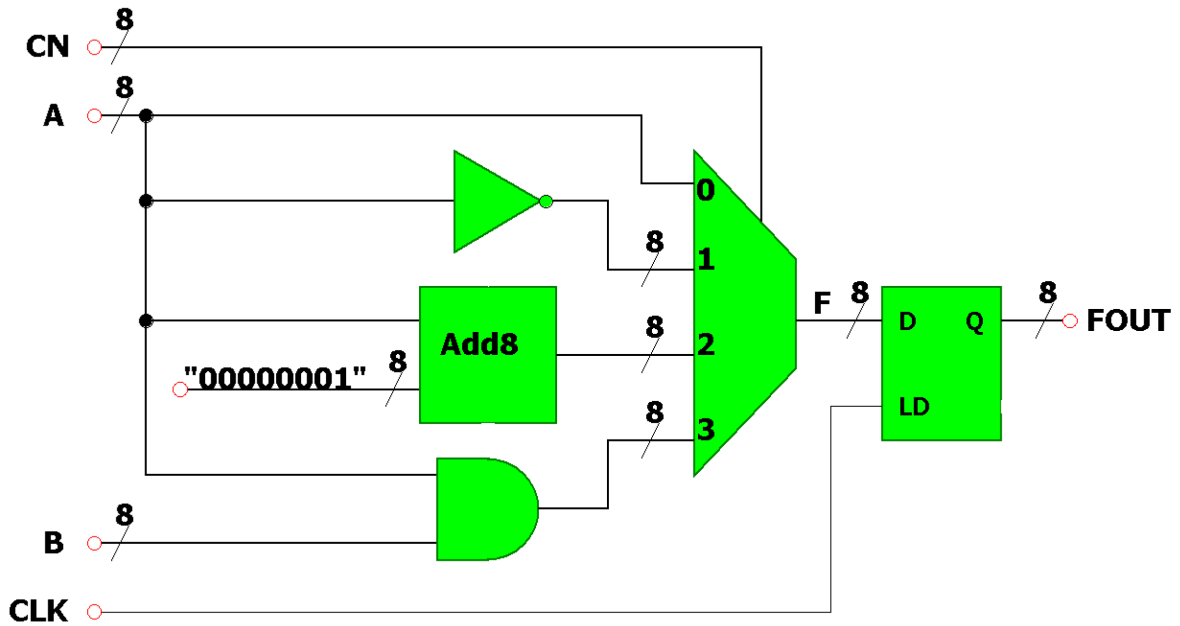
❖ *Schéma logique optimisé d'un composant:*



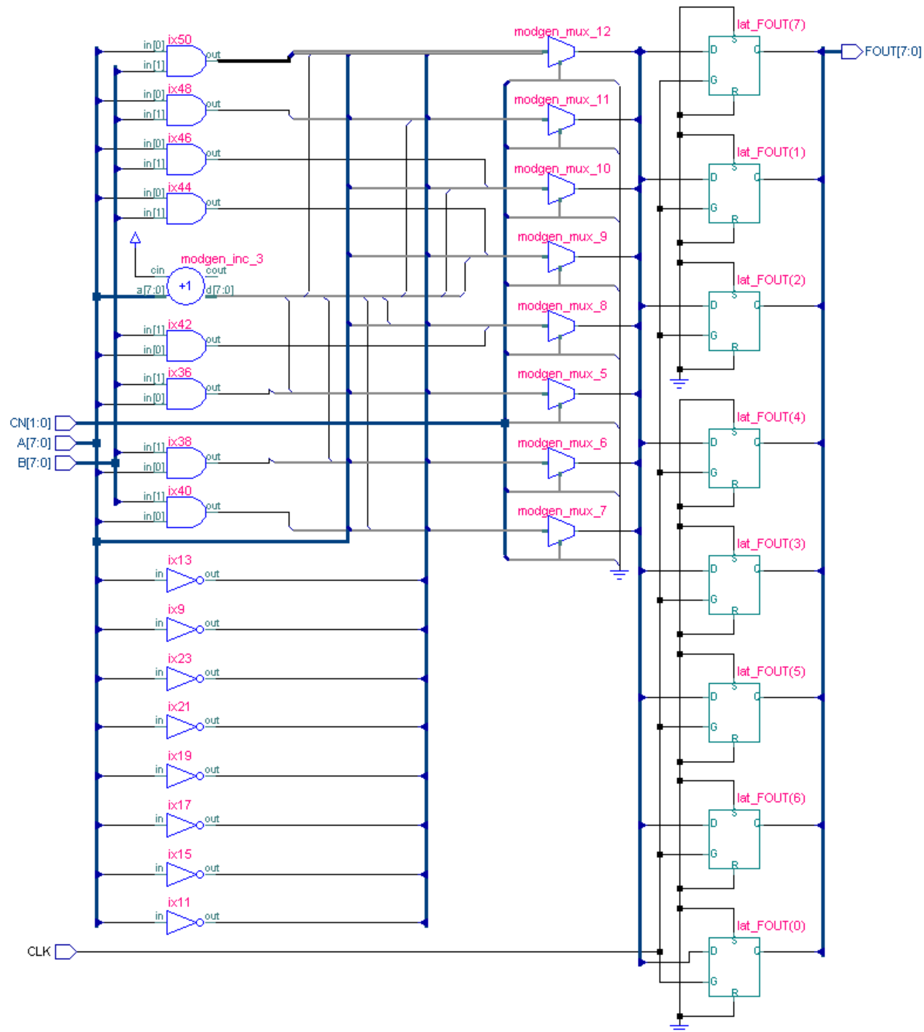
Dessiner le système représenté par le code VHDL suivant:

```
library IEEE ;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity logic is
  port (CN : in std_logic_vector (1 downto 0);
        A, B : in std_logic_vector (7 downto 0);
        CLK : in std_logic;
        FOUT : out std_logic_vector (7 downto 0));
end logic;

architecture toto of logic is
  signal F : std_logic_vector (7 downto 0)
begin

process (CN, A, B)
begin
  case CN is
    when "00" => F <= A;
    when "01" => F <= not A;
    when "10" => F <= A + "00000001";
    when "11" => F <= A and B;
    when others => null;
  end case;
end process;

process (CLK, F)
begin
  if CLK='1'
    then FOUT <= F;
  end if;
end process;

end toto;
```

❖ *Schéma logique correspondant au code VHDL :*



❖ *Schéma généré par Leonardo :*



Pour implémenter l'algorithme de cryptage IDEA, l'opération suivante est nécessaire :

$$\mathbf{S = A + B + \overline{c_{out}}(A+B)} \qquad \mathbf{(1)}$$

où les deux entrées A et B, ainsi que la sortie S, sont des valeurs non signées sur n bits.

❶ Une première solution naïve est celle illustrée par la figure 1. Montrer qu'il est impossible de réaliser ce circuit à l'aide d'un additionneur à retenue propagée (opérateur d'addition dans le code VHDL).
Indication: Montrer que le circuit comporte une boucle combinatoire conduisant dans certains cas à des oscillations. Choisissez une petite valeur de n (par exemple n=8) et donnez un exemple d'entrées A et B provoquant ce phénomène.

❷ Proposer deux circuits implémentant l'équation (1), sans créer de boucle combinatoire, pour des opérandes de seize bits, et donner les codes VHDL correspondants. Une solution doit être combinatoire et l'autre séquentielle (pour ce cas-ci, respecter le timing de la figure 2). Il est interdit d'utiliser deux additionneurs en série.

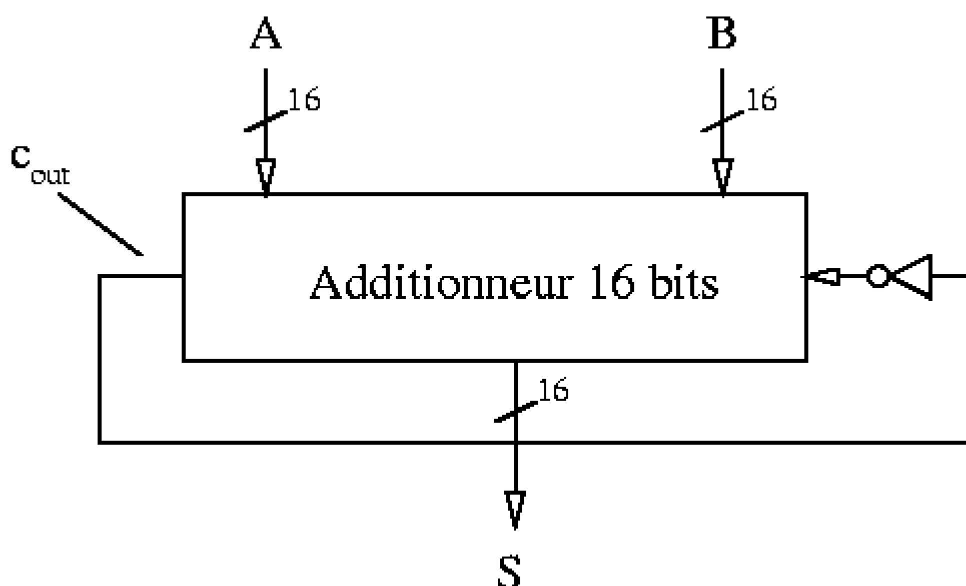


Figure 1

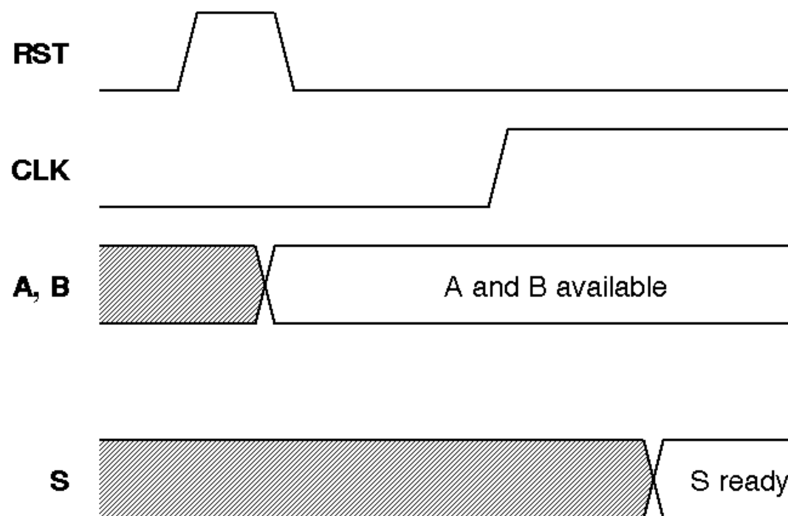


Figure 2

Les cas qui provoquent des oscillations sont ceux qui remplissent les deux conditions suivantes:

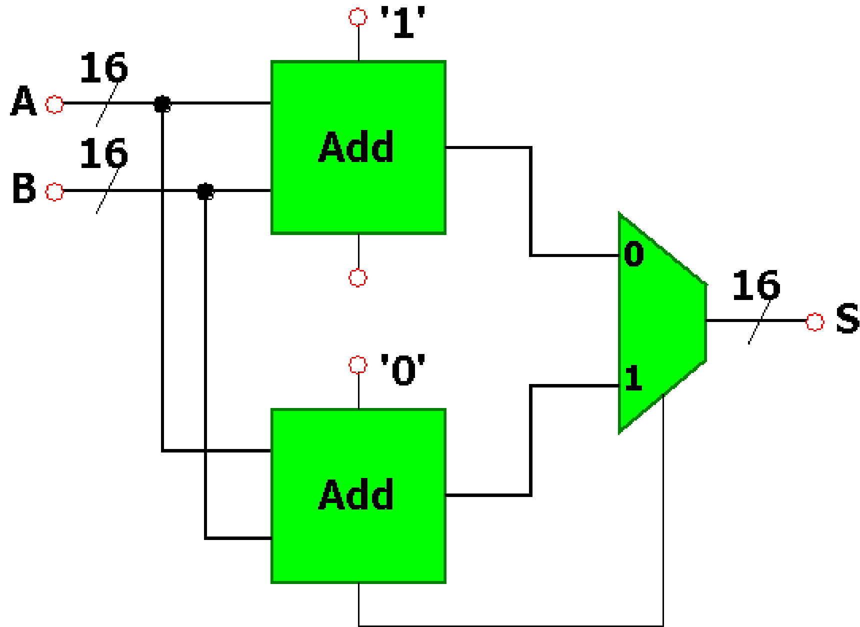
- *Pour Cin=0, Cout=0 (ce qui implique que Cin va passer à 1)*
- *Pour Cin=1, Cout=1 (ce qui implique que Cin va passer à 0)*

Ces conditions sont respectées par tous les cas où la somme des opérandes vaut 1111'1111.

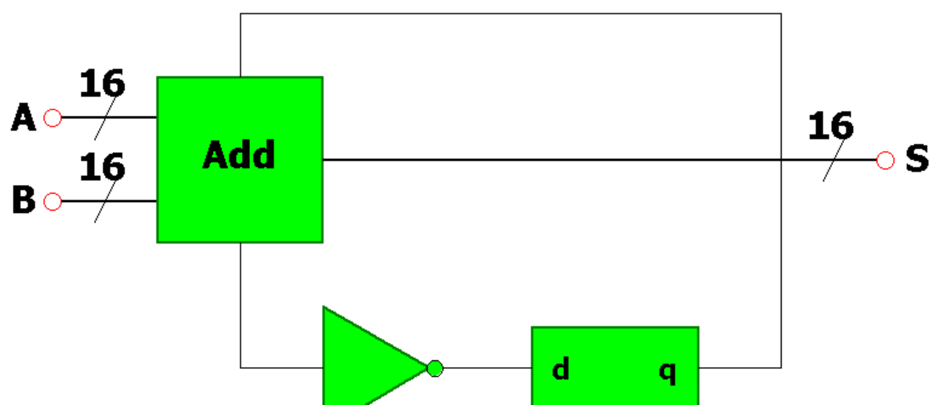
<u>Exemple</u>: A = 1111'1111h B = 0000'0000h

Séquences	A	B	S	Cout	Cout_inv
Etat initial	1111'1111	0000'0000	U	U	0
Après Tprop additionneur	1111'1111	0000'0000	1111'1111	0	0
Après Tprop inverseur	1111'1111	0000'0000	1111'1111	0	1
Après Tprop additionneur	1111'1111	0000'0000	0000'0000	1	1
Après Tprop inverseur	1111'1111	0000'0000	0000'0000	1	0
Après Tprop additionneur	1111'1111	0000'0000	1111'1111	0	0
…					

- *On constate que la sortie **S** oscille entre les deux valeurs 1111'1111h et 0000'0000h. La période d'oscillation est fonction du temps de propagation des composants.*




```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY idea IS
   port ( A, B : in std_logic_vector(15 downto 0);
              S : out std_logic_vector(15 downto 0) );
END idea;

ARCHITECTURE synth OF idea IS
   signal S1 : std_logic_vector(15 downto 0);
   signal S2 : std_logic_vector(16 downto 0);

BEGIN
   S1 <= A + B + '1';
   S2 <= ('0' & A) + ('0' & B);
   process (S1, S2)
   begin
      if (S2(16) = '1')
         then
            S <= S2(15 downto 0);
         else
            S <= S1;
      end if;
   end process;
END
```



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY idea IS
   port ( A, B : in std_logic_vector(15 downto 0);
          clk, R : in std_logic;
          S : out std_logic_vector(15 downto 0) );
END idea;

ARCHITECTURE synth OF idea_seq IS
   signal S1 : std_logic_vector(16 downto 0);
   signal cin : std_logic;
BEGIN
   S1 <= ('0' & A) + ('0' & B) + cin;
   S <= S1(15 downto 0);
   process (r, clk)
   begin
      if (r = '1') then
         cin <= '0';
      elsif (clk'event and clk='1') then
         cin <= not S1(16);
      end if;
   end process;
END synth;
```

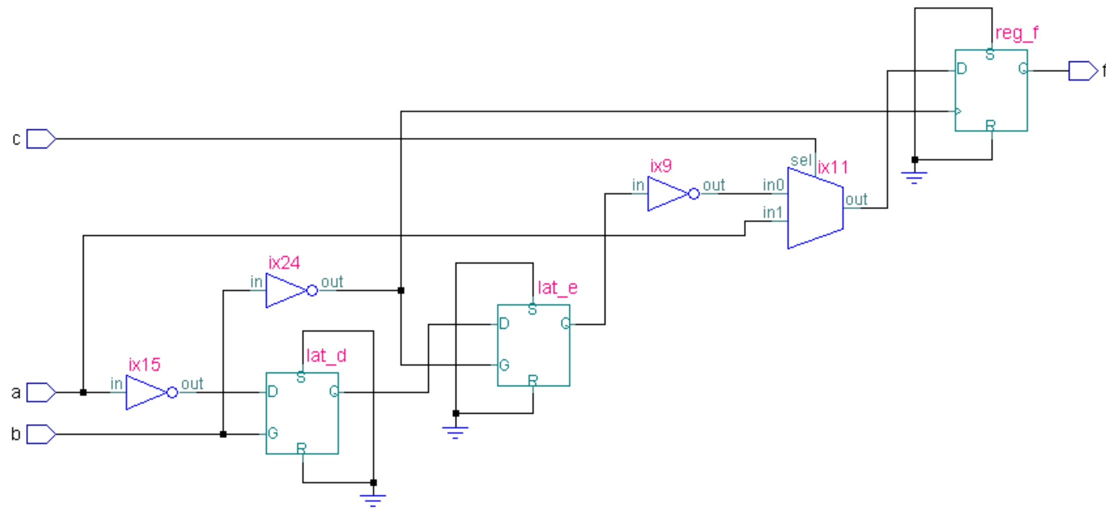
Considérer le programme VHDL de la figure 1.

❶ Dessiner le schéma logique correspondant.

❷ Indiquer si les listes de sensitivité des processus **this** et **that** contiennent des signaux superflus. Si c'est le cas, donner une liste minimale.

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

entity toto is
  port (a, b, c : in std_logic;
        f       : out std_logic);
end toto;

architecture titi of toto is
  signal d, e : std_logic;
begin

this: process (a, b, c, e)
      begin
        if (b'event) and (b = '0')
          then if c = '1'
                 then f <= a;
                 else f <= not e;
               end if;
        end if;
      end process;

that: process (a, b, d)
      begin
        if (b = '0')
          then e <= d;
          else d <= not a;
        end if;
      end process;

end titi;
```

Figure 1



- *Les listes de sensibilité minimales :*
  - *this: process (b)*
  - *that: process (a, b ,d)*

Le tableau suivant donne le code Gray pour les digits décimaux :

0	0000
1	0001
2	0011
3	0010
4	0110
5	0100
6	0101
7	0111
8	1111
9	1110

Le système représenté à la figure 2 reçoit des digits décimaux codés en Gray, sur une ligne sérielle **X** (un seul bit d'entrée, le bit de poids faible le premier). Le début de l'envoi d'un digit est indiqué par un signal **start** (c'est-à-dire, **start**=1 en même temps que le premier bit d'information). Il est possible que des erreurs de transmission fassent apparaître des codes faux (ceux n'apparaissant pas dans le tableau précédent).

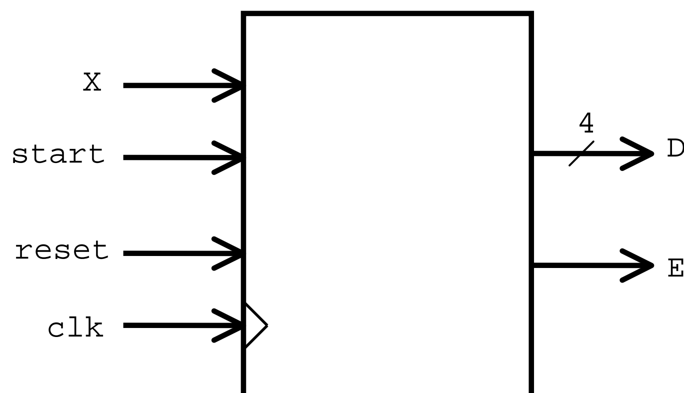


Figure 2

A la fin de l'envoi d'un digit, le système doit afficher en parallèle son equivalent décimal codé en 4 bits, avec un bit **E** associé pour indiquer une éventuelle erreur de transmission. Dans le cas d'une erreur, le digit affiché doit être 1111, avec **E**=1. (Attention : les 4 bits de D et le bit d'erreur correspondant doivent être affichés en même temps, à la fin de l'envoi).

La figure 3 donne un exemple de comportement du système (l'entrée X reçoit les digits 1001 et 0100, dans cet ordre, avec un trou d'un coup d'horloge entre les deux).

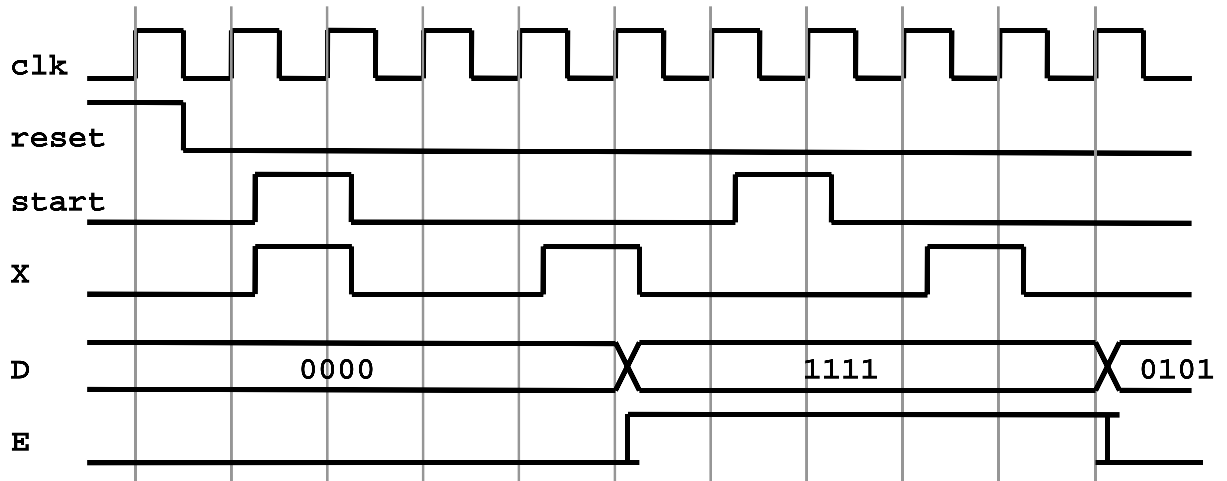


Figure 3

Ecrire le code VHDL décrivant une solution pour le système. Faire le schéma logique correspondant et expliquer son fonctionnement.

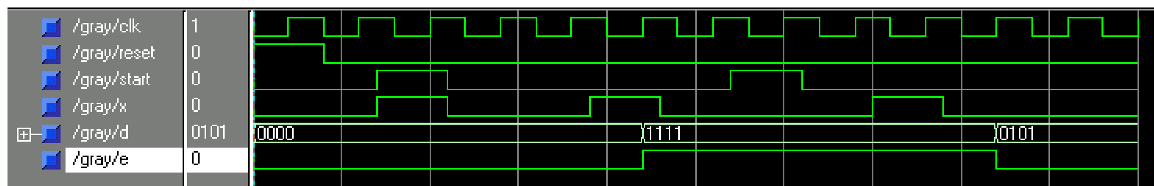
```
library ieee;
use ieee.std_logic_1164.all;

entity gray is
  port (clk       : in std_logic;
        reset     : in std_logic;
        start, x  : in std_logic;
        d         : out std_logic_vector (3 downto 0);
        e         : out std_logic);
end gray;

architecture test of gray is
  signal shift, ld : std_logic;
  signal ShiftOut  : std_logic_vector (3 downto 0);
  signal CountOut : std_logic_vector (1 downto 0);
  signal CountIn : std_logic_vector (1 downto 0);
  signal addr   : std_logic_vector (3 downto 0);
  signal MemOut : std_logic_vector (4 downto 0);

begin
compteur: process (start, CountOut)
          begin
            CountIn <= "00";
            shift <= '1';
            ld <= '0';
            case CountOut is
              when "00" => if start='1'
                             then CountIn <= "01";
                             else shift <= '0';
                           end if;
              when "01" => CountIn <= "10";
              when "10" => CountIn <= "11";
              when "11" => CountIn <= "00";
                           ld <= '1';
              when others => null;
            end case;
          end process;

          process (clk, reset)
          begin
            if reset='1'
              then CountOut <= "00";
              else if (clk'event) and (clk='1')
                     then CountOut <= CountIn;
                   end if;
            end if;
          end process;
```

```
sreg: process (clk, reset)
      begin
        if reset='1'
          then ShiftOut <= (others => '0');
          else if (clk'event) and (clk='1')
                 then if (start='1') or (shift='1')
                        then ShiftOut <= x &
ShiftOut(3 downto 1);
                      end if;
               end if;
        end if;
      end process;

addr <= x & ShiftOut (3 downto 1);

mem: process (addr)
     begin
       case addr is
         when "0000" => MemOut <= "00000";
         when "0001" => MemOut <= "00010";
         when "0010" => MemOut <= "00110";
         when "0011" => MemOut <= "00100";
         when "0100" => MemOut <= "01010";
         when "0101" => MemOut <= "01100";
         when "0110" => MemOut <= "01000";
         when "0111" => MemOut <= "01110";
         when "1110" => MemOut <= "10010";
         when "1111" => MemOut <= "10000";
         when others => MemOut <= "11111";
       end case;
     end process;

load: process (clk, reset)
      begin
        if reset='1'
          then d <= "0000";
               e <= '0';
          else if (clk'event) and (clk='1')
                 then if ld='1'
                        then d <= MemOut (4 downto 1);
                  e <= MemOut (0);
                      end if;
               end if;
        end if;
      end process;

end test;
```

Fichier de commande pour la simulation :

```
force clk 0 0, 1 40 -repeat 80
force reset 1 0, 0 80
force start 0 0, 1 140, 0 220, 1 540, 0 620
force x 0 0, 1 140, 0 220, 1 380, 0 460, 1 700, 0 780
run 1000 ns
```

Résultat de la simulation :



Supposez le programme VHDL suivant:

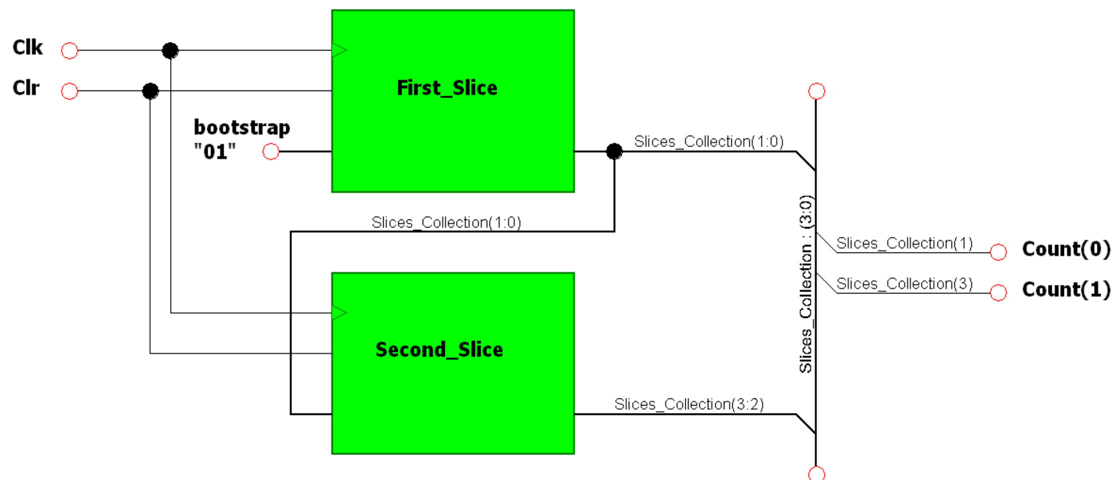
```
library ieee;
use ieee.std_logic_1164.all;

entity slice is
  port (clk            : in std_logic;
        reset          : in std_logic;
        previous_slice : in std_logic_vector(1 downto 0);
        slice          : out std_logic_vector(1 downto 0));
end slice;

architecture fsm of slice is
  signal current_slice,
         next_slice    : std_logic_vector(1 downto 0);
begin

  slice <= current_slice;

  register_layer: process (clk, reset)
                  begin
                    if reset='1'
                      then current_slice <= (others => '0');
                      elsif clk'event and clk='1'
                        then current_slice <= next_slice;
                    end if;
                  end process register_layer;

  behavior: process (current_slice, previous_slice)
    begin
      next_slice(1) <= current_slice(0);
      case previous_slice is
        when "01" =>
          next_slice(0) <= not current_slice(0);
        when others =>
          next_slice(0) <= current_slice(0);
      end case;
  end process behavior;

end fsm;
```

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

entity counter is
  port (clk   : in std_logic;
        reset : in std_logic;
        count : out std_logic_vector (1 downto 0));
end counter;

architecture sliced of counter is

  component slice
```

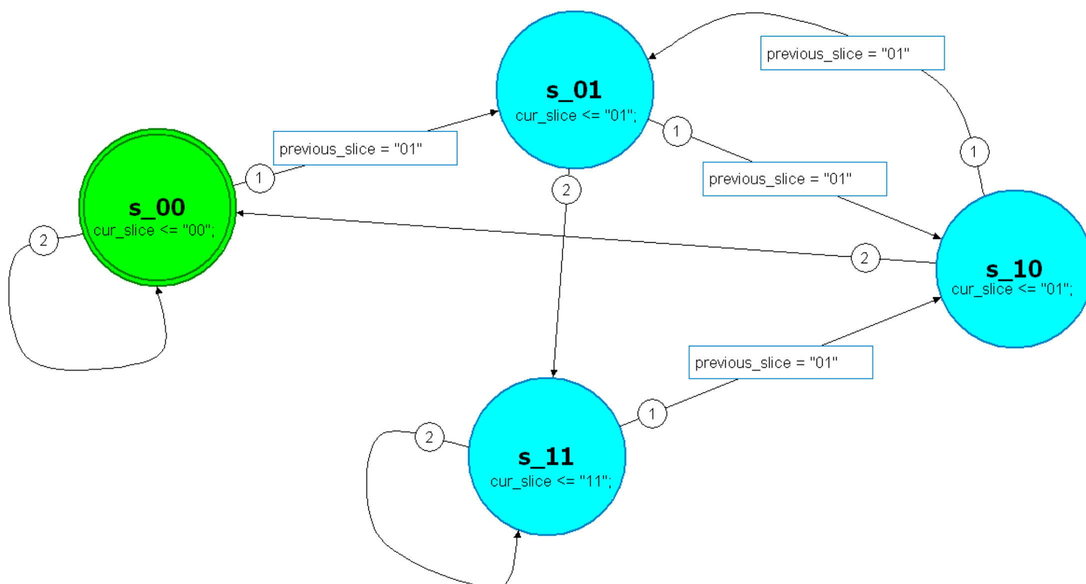
```
    port (clk            : in std_logic;
          reset          : in std_logic;
          previous_slice : in std_logic_vector(1 downto 0);
          slice          : out std_logic_vector(1 downto 0));
  end component;

  signal bootstrap         : std_logic_vector(1 downto 0);
  signal slices_collection : std_logic_vector(3 downto 0);

begin

  bootstrap <= "01";

  first_slice: slice
               port map (clk => clk,
                         reset => reset,
                         previous_slice => bootstrap,
                         slice => slices_collection(1 downto 0));

  count(0) <= slices_collection(1);

  second_slice: slice
            port map (clk => clk,
                      reset => reset,
                      previous_slice => slices_collection(1 downto
0),
                      slice => slices_collection(3 downto 2));

  count(1) <= slices_collection(3);

end sliced;
```

a. Dessinez le schéma logique de l'entité **counter**
b. Dessinez le graphe des états de la machine séquentielle de l'entité **slice**
c. Décrivez le comportement de la sortie **count**, en indiquant la séquence d'états pour chaque tranche (**slice**), à partir du **reset** initial.

a.



b. L'état de la machine correspond dans cet exemple à l'état de la sortie Current_Slice.

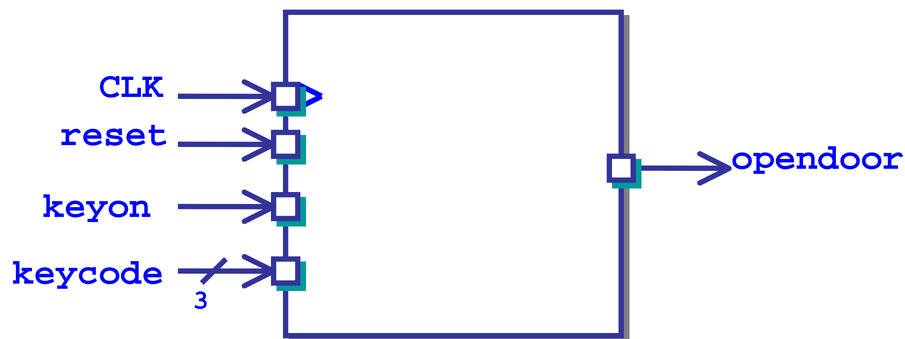


c. First_Slice: 00 -> 01 -> 10 -> 01 -> 10 ->

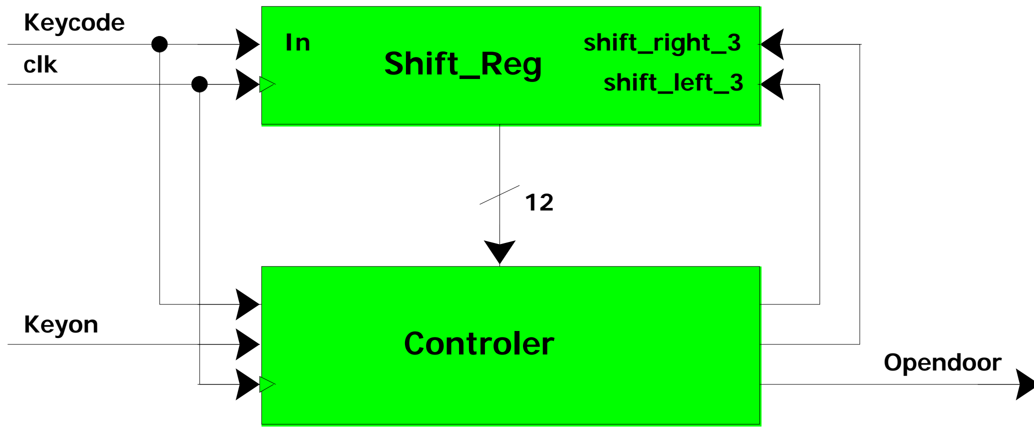
Second_Slice: 00 -> 00 -> 01 -> 11 -> 10 ->

Count: 00 -> 00 -> 01 -> 10 -> 11 ->

Supposez un système qui contrôle l'ouverture d'une porte, après introduction d'un code de 4 digits sur un clavier.
Le clavier possède 7 touches numériques, de 0 à 6, et une touche **<del>** permettant l'effacement du dernier digit introduit.
Lorsqu'une touche est pressée, un code binaire sur 3 bits est envoyé au système (signal **keycode**), ainsi qu'un signal **keyon**, actif tant que la touche est pressée (le code envoyé pour la touche **<del>** est 111).
Si la séquence 3625 est introduite, quel que soit l'état du système, un signal **opendoor** est produit pendant un seul cycle d'horloge, pour commander l'ouverture de la porte.



Dessinez le schéma logique du système et écrivez sa description en VHDL.



reset

s0

Keyon='0'
-

Keyon='1'; Keycode=7;
shift_left_3 <= '1';

Keyon='1'; Keycode!=7;
shift_right_3 <= '1';

Keyon='0'
-

s1

reg=5'2'6'3
Opendoor <= '1';

s2

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

entity porte is
  port (clk           : in std_logic;
        reset         : in std_logic;
        keyon         : in std_logic;
        keycode       : in std_logic_vector(2 downto 0);
        opendoor      : out std_logic);
end porte;

architecture synth of porte is
     type typeetat is (S0, S1, S2);
     signal state, next_state : typeetat;
     signal reg : std_logic_vector(11 downto 0);
     signal ld_shift_right_3, shift_left_3 : std_logic;

begin

  shiftreg: process(clk, reset)
  begin
    if reset='1'
      then reg <= (others=>'0');
      elsif (clk'event and clk='1')
        then
          if (ld_shift_right_3='1')
            then reg <= keycode & reg(11 downto 3);
            elsif (shift_left_3='1')
              then reg <= reg(8 downto 0) & "000";
          end if;
    end if;
  end process shiftreg;



  sync: process(clk, reset)
  begin
    if (reset='1')
      then state <= s0;
      elsif (clk'event and clk='1')
        then state <= next_state;
    end if;
  end process sync;
```

```
  ctrl: process(state, keycode, keyon, reg_contenu)
  begin

    ld_shift_right_3 <= '0';
    shift_left_3 <= '0';
    next_state <= state;
    opendoor <= '0';

    case state is
    when s0 =>
      if (keyon='1')
        then
          if(keycode="111")
            then shift_left_3 <= '1';
            else ld_shift_right_3 <= '1';
          end if;
          next_state <= s1;
      end if;

    when s1 =>
      if (reg="101010110011")      -- 5'2'6'3
        then
          opendoor <= '1';
          next_state <= s2;
        elsif (keyon='0')
          then next_state <= s0;
        end if;

    when s2 =>
      if (keyon='0')
        then next_state <= s0;
      end if;

    end case;

  end process ctrl;


end synth;
```

Analisez le programme VHDL suivant:

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

entity my_circuit is
  port (
    CLK  : in  std_logic;
    nRST : in  std_logic;
    A    : in  std_logic;
    B    : in  std_logic;
    C    : out std_logic);
end my_circuit;

architecture beh of my_circuit is

  signal A_i : std_logic_vector(1 downto 0);
  signal B_i : std_logic_vector(1 downto 0);
  signal C_i : std_logic_vector(1 downto 0);
  signal carry : std_logic;
  signal sum : std_logic_vector(1 downto 0);

begin  -- beh

  A_i <= ('0' & A);
  B_i <= ('0' & B);
  C_i <= ('0' & carry);

  sum <= A_i + B_i + C_i;

  C <= sum(0);

  process (CLK, nRST)
  begin  -- process
    if nRST = '0' then
      carry <= '0';
    elsif CLK'event and CLK = '1' then
      carry <= sum(1);
    end if;
  end process;

end beh;
```

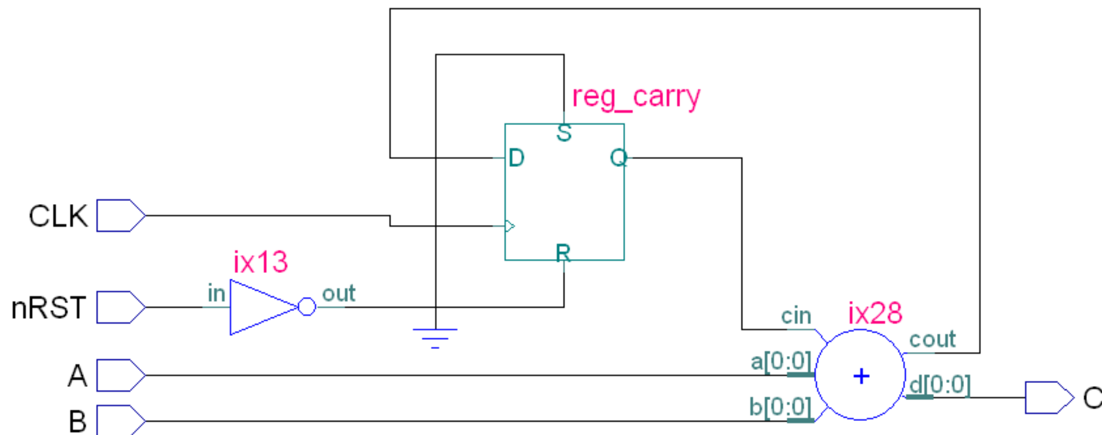
1. Dessinez le schéma logique correspondant.

2. Dessinez un diagramme de temps pour les signaux A, B, C et carry lorsque les valeurs 1011_2 et 0010_2 sont envoyées en série (en commençant par le bit de poids faible) sur les entrées A et B, respectivement. Dessinez les 6 premiers cycles d'horloge.

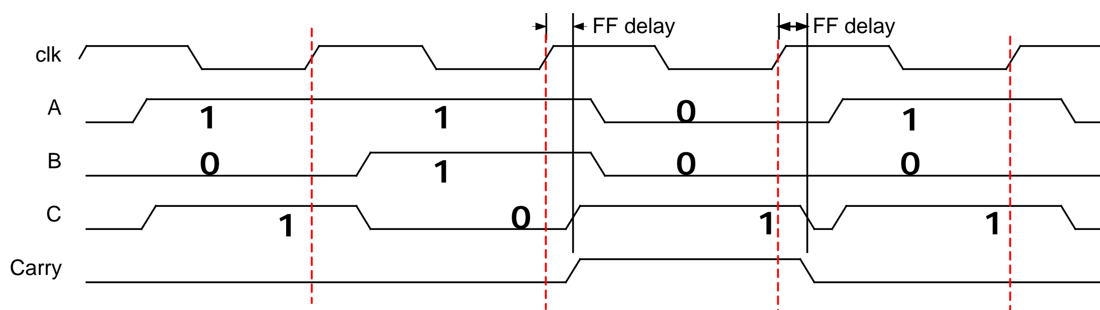
3. Décrivez en une phrase la fonction du circuit. Expliquez quel rôle a dans l'algorithme le composant réalisé par les lignes:

```
elsif CLK'event and CLK = '1'
  then carry <= sum(1);
end if;
```

1.



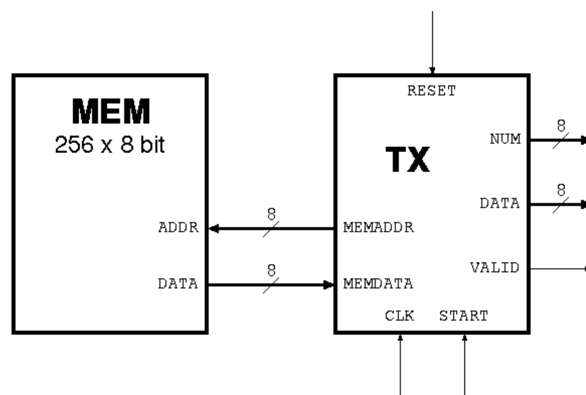
2.



3.
Ce composant est un additionneur série. (Les opérandes sont entrés en série et le produit est délivré en série, au fur et à mesure: 1 -> 0 -> 1 -> 1)

L'état du report (Carry) est mémorisé dans une bascule pour l'addition du bit suivant (de poids supérieur).

Considérez un système qui envoie 256 mots de 8 bits sur une ligne parallèle. Le transmetteur est un composant parfaitement synchrone sur le flanc montant d'un signal d'horloge CLK. Il lit les 256 mots à transmettre d'une mémoire asynchrone. Il commence la transmission le cycle suivant la réception du signal de START. Il transmet en séquence tous les mots à partir de l'adresse 0 jusqu'à l'adresse 255 ; il envoie à la fois le mot lui-même (sur le bus DATA) et son numéro identificateur (sur le bus NUM). En même temps il active le signal VALID. Après le 256e mot (dont l'identificateur est 255), il se remet en attente du signal START pour un nouvel envoi.



a. Dessinez un schéma à blocs possible pour le transmetteur TX. Utilisez des composants classiques tels que portes logiques, registres, additionneurs, etc. Si vous utilisez un contrôleur, dessinez le diagramme des états complet.

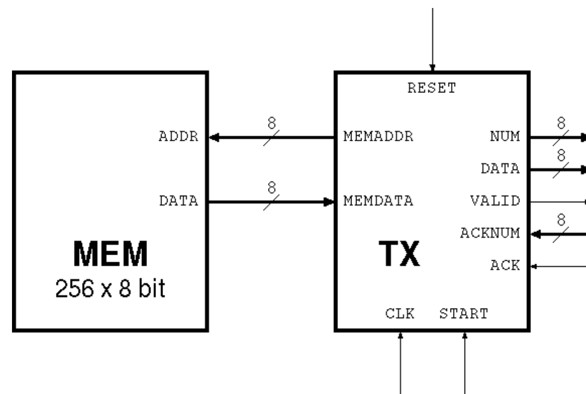
b. Ecrivez le code VHDL qui réalise le transmetteur.

c. Il se trouve que le canal sur lequel les data sont envoyées ne garantit pas une bonne transmission. On modifie le transmetteur pour qu'il implémente le protocole d'acquittement suivant :

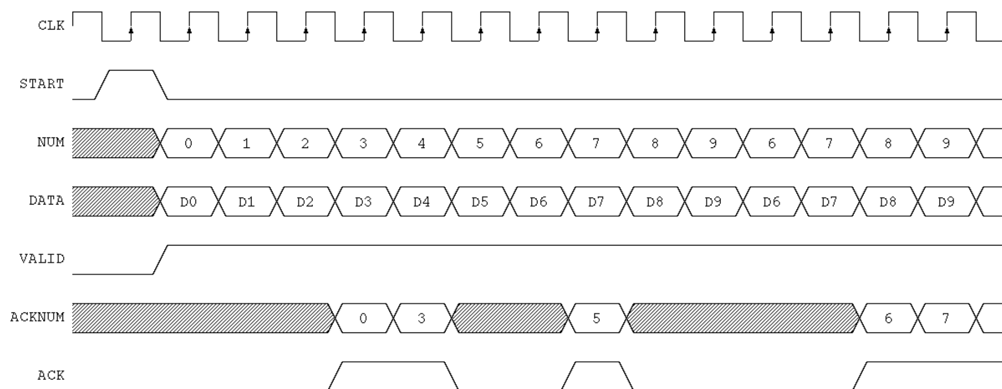
- Le transmetteur reçoit des acquittements de la part du récepteur lorsque le signal ACK est actif. Si ACK est actif, le transmetteur peut lire le numéro du mot acquitté sur le bus ACKNUM. Ces acquittements informent le transmetteur du dernier mot bien reçu à destination. Remarquez que le récepteur n'est tenu ni à acquitter

chaque mot individuellement, ni à le faire à un moment précis (avec la limitation ci-dessous).

- Le transmetteur n'envoie jamais plus que 4 mots sans avoir reçu d'acquittement. Si cela arrive, au lieu d'envoyer un cinquième mot, il revient en arrière et recommence à transmettre à partir du premier mot qui n'a pas été acquitté. Cela se répète tant qu'un nouveau mot n'est pas acquitté.



Le diagramme des temps suivant illustre le début d'une session :



Remarquez que les mots 1, 2 et 4 ne sont jamais acquittés et que cela n'a aucun impact sur la transmission. Notez aussi que, au contraire, le retard des acquittements après le mot 5 fait que – après avoir envoyé les quatre mots 6, 7, 8 et 9 – le transmetteur recommence à envoyer à partir du mot 6.

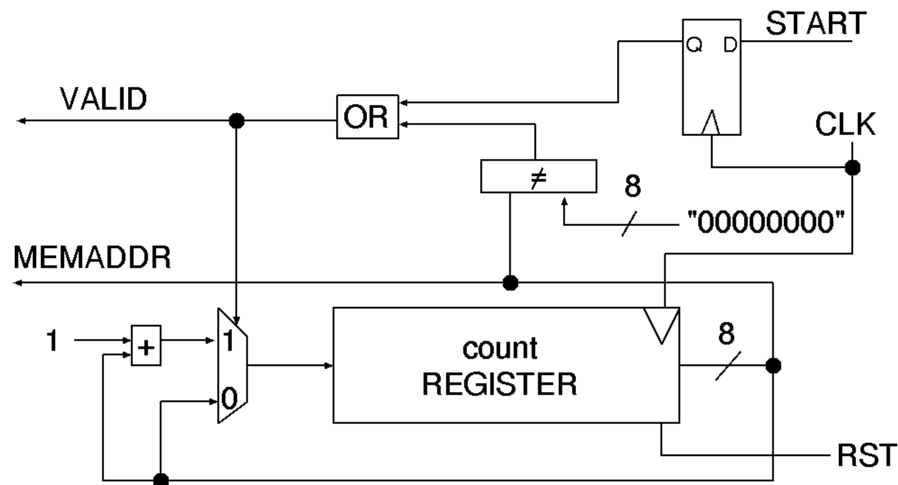
Modifiez le schéma à blocs du transmetteur TX pour réaliser ce protocole. Ignorez les détails du protocole à la fin de

l'envoi des 256 mots (par exemple, garantissez juste l'arrêt du système après le premier envoi du mot 255). Discutez en détail ou réalisez les modifications significatives au code VHDL.

a)
Le transmetteur doit effectuer la tache suivante : chaque fois que start est actif, il doit lire et transmettre le contenu de la mémoire. Pour ce faire, il doit générer successivement les adresses de 0 à 255. Une fois la mémoire lue et transmise, le transmetteur se remet en attente du signal start.
Le schéma à blocs du transmetteur est donc constitué d'un compteur qui a les propriétés suivantes :

- Il commence à compter depuis 0 quand start devient actif.
- Il s'arrete de compter lorsqu'il a atteint 255.
- Il recommence à compter depuis 0 quand start est activé à nouveau.

L'identifiant d'une donnée est choisie comme son adresse.
Le schéma bloc ci-dessous effectue cette tache.

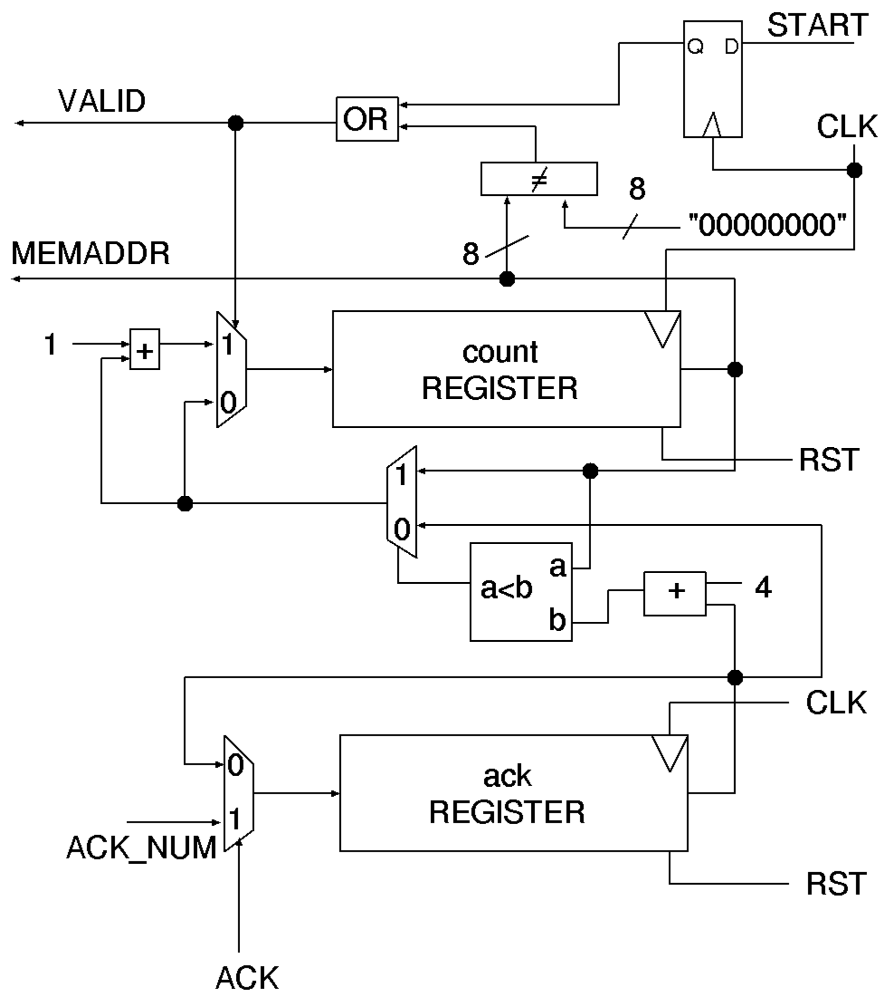


b)

```
entity transmetteur-basic (
START : IN std_logic ;
CLK : IN std_logic ;
RST : IN std_logic ;
MEMDATA : IN std_logic_vector(7 downto 0);
DATA : OUT std_logic_vector(7 downto 0);
MEMADDR : OUT std_logic_vector(7 downto 0);
NUM : OUT std_logic_vector(7 downto 0);
VALID : OUT std_logic)

architecture synth of transmetteur-basic is

signal enable, countNotZero : std_logic;
signal count: std_logic_vector(7 downto 0);
signal next_count: std_logic_vector(7 downto 0);
begin

DATA <= MEMDATA;
NUM <= count;
MEMADDR <= count;
VALID <= enable;
enable <= start_int or countIszero;
countNotzero <= '1'
    when not(count  = ´´00000000´´) else '0';

comb: process(count, enable)
begin
 next_count <= count;
 if enable = '1' then
  next_count <= count + 1;
 end if;
end comb;

reg: process(CLK, RST)
begin
 if RST = '1' then
 count <= (others = '0');
 start_int <= '0';
 elsif CLK'event and CLK='1' then
  count <= next_count;
  start_int <= start;
 end if;
end reg;

end synth;
```

c)
Le transmetteur doit maintenant garder en mémoire, en plus de ce qu'il fait en a), l'identificateur du dernier mot recu correctement par le récepteur. Au cas ou l'adresse du prochain mot à transmettre est supérieure de 4 à l'identificateur ACK_NUM recu du récepteur, l'adresse générée par le transmetteur est le dernier ACK_NUM + 1.

Le schema bloc est modifié comme suit :



Il faut rajouter au code le registre « last_acked » mémorisant la dernière valeur de ACK_NUM :

```
process (CLK,RST)
begin
if RST = '1' then
     ack_reg <= ´´00000000´´;
elsif CLK'event and CLK = 1 then
     if ACK = '1' then
      ack_reg <=  ACK;
     end if;
end if;
```

De plus, la valeur du registre ´´ack_reg´´ est prise en compte pour la nouvelle valeur du registre ´´count´´ :

```
comb : process (count, ack_reg, enable)
begin
next_count <= count;
if count = ack_reg +4 then
     next_count <= ack_reg + 1;
elsif enable = '1' then
     next_count <= count + 1;
end if;
end comb ;
```

Dessinez le circuit correspondant au code VHDL suivant (tous les signaux sont du type std_logic) :

```
architecture synthesizable of test is
  signal E : std_logic;
begin
  process (A, B, C, D, E)
  begin
    if A'event and A = '0' then
      if B = '0' then
        E <= C;
      else
        E <= D;
      end if;
      F <= E;
    end if;
  end process;
end synthesizable;
```

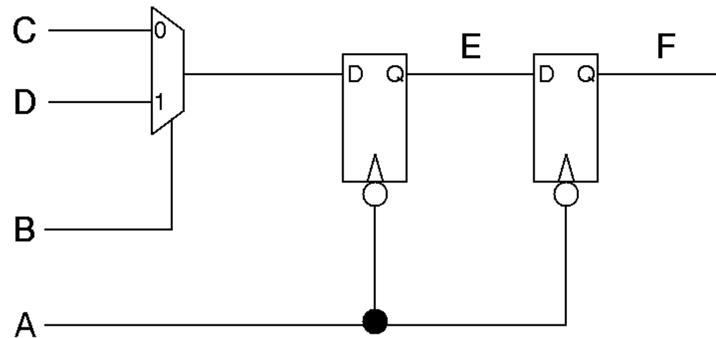
a. Dessinez le circuit correspondant si le signal E était remplacé par une variable.

b. Considérez maintenant le code VHDL suivant :

```
architecture synthesizable of test is
begin
  process (A, B, C, D)
    variable E : std_logic;
  begin
    if A'event and A = '0' then
      F <= E;
      if B = '0' then
        E := C;
      else
        E := D;
      end if;
    end if;
  end process;
end synthesizable;
```

Est-ce qu'il correspond au même circuit que l'un des deux circuits dessinés aux points a. ou b. ? Si oui, lequel ? Expliquez brièvement votre réponse.

a)
Le circuit représenté est le suivant :



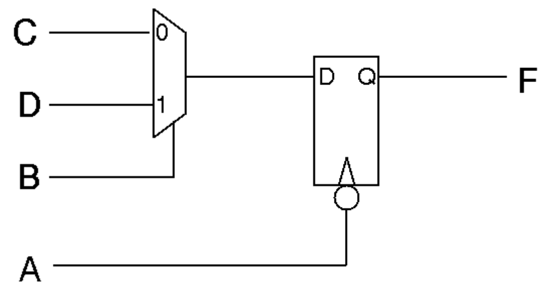
b)
Si l'on remplace E par une variable, comme suit :

```
process (A, B, C, D)
  variable E : std_logic;
  begin
    if A'event and A = '0' then
      if B = '0' then
        E := C;
      else
        E := D;
      end if;
     F <= E;
    end if;
  end process;
```

le code devient equivalent à:

```
architecture synthesizable of test is
begin
  process (A, B, C, D)
  begin
    if A'event and A = '0' then
      if B = '0' then
        F <= C;
      else
        F <= D;
      end if;
    end if;
  end process;
end synthesizable;
```

ce qui correspond au circuit :



La valeur de la variable E est assignée immédiatement et utilisée tout de suite pour F. Sa valeur entre deux exécutions du process est donc inutile.

c)
Entre deux exécutions du process, la variable E garde en mémoire sa valeur. Pour effectuer cette mémorisation, il faut une bascule. Ce cas est par conséquent identique à celui de a).

Considérez l'opération arithmétique de division entre deux entiers non signés. Elle peut se réaliser en représentation binaire exactement selon la même méthode qu'on utilise en décimal dans le calcul élémentaire : on considère itérativement des groupes de chiffres du dividende à partir de la gauche et on y soustrait le plus grand multiple possible du diviseur. Le multiple utilisé représente un chiffre du quotient, calculé de gauche a droite. Le dernier résultat de la soustraction représente le reste de la division. Etudiez à l'aide de cet exemple le procédé en binaire :

```
                     0 1 0 1 1 0 : 0 1 1 = 0 0 0 1 1 1
                     0
0 x 011 →       -0 0 0
                 0 0 0 1
0 x 011 →         - 0 0 0
                   0 0 1 0
0 x 011 →           - 0 0 0                22 : 3 = 7 (+ 1/3)
                     0 1 0 1
1 x 011 →             - 0 1 1
                       0 1 0 1
1 x 011 →               - 0 1 1
                         0 1 0 0
1 x 011 →                 - 0 1 1
                           0 0 1
```

Pour effectuer la division, on exécute les pas suivants jusqu'à que tous les chiffres du dividende aient étés considérés :

1. Ajoutez un chiffre du dividende au groupe de chiffres considérés, en prenant en compte les chiffres du dividende à partir de la gauche. À la première itération, considérez seulement le bit de poids fort.
2. Si les chiffres considérés représentent un nombre plus grand ou égal au diviseur, soustrayez le diviseur des chiffres considérés ; autrement soustrayez zéro.
3. Si vous avez soustrait le diviseur, ajoutez à droite un '1' au quotient ; autrement, ajoutez '0'.

Répétez à partir du pas 1 tant qu'il reste des chiffres au dividende.

On veut implémenter un tel opérateur avec plusieurs copies d'un circuit qui réalise exactement une itération de l'algorithme décrit ci-dessus (pas de 1 a 3).

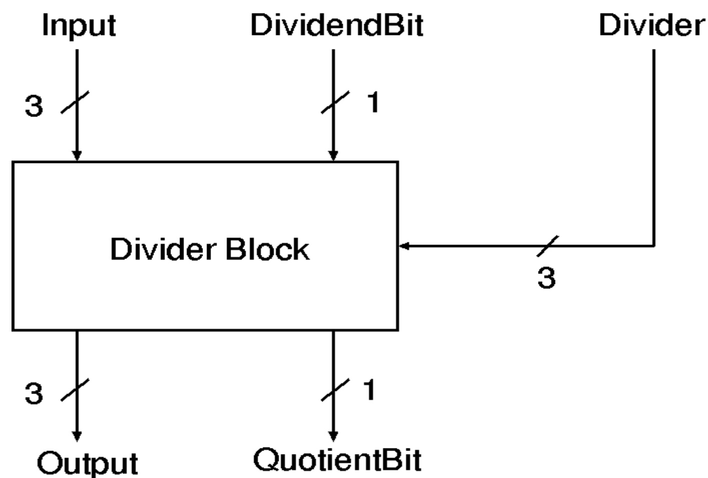
a. Décrivez les signaux d'entrée et de sortie du circuit qui réalise une itération de l'algorithme de division. Spécifiez le nombre de bits nécessaires pour encoder chacun des signaux et expliquez clairement la précision requise. (Supposez que toutes les copies de ce circuit sont identiques, même si certaines pourraient être simplifiées).

b. Combien de bits sont nécessaires pour coder le quotient ? Combien pour coder le reste ? Combien de copies du circuit ci-dessus sont nécessaires pour réaliser une division ? Montrez avec un schéma à blocs comment connecter les copies du circuit et réalisez ainsi un diviseur complet.

c. Dessinez le schéma à blocs du circuit qui réalise une itération de la division. Utilisez exclusivement des additionneurs, soustracteurs, multiplexeurs et portes logiques élémentaires, selon nécessité.

d. Ecrivez le code VHDL du circuit qui réalise une itération de la division.

a)
La figure ci-dessous représente le circuit (DividerBlock) qui effectue une itération de l'algorithme de division.



Le circuit en question a comme entrées :

- un bit du dividende (DividendBit),
- trois bits issus de l'itération précédente (Input) et,
- le diviseur (Divider), codé sur 3 bits.

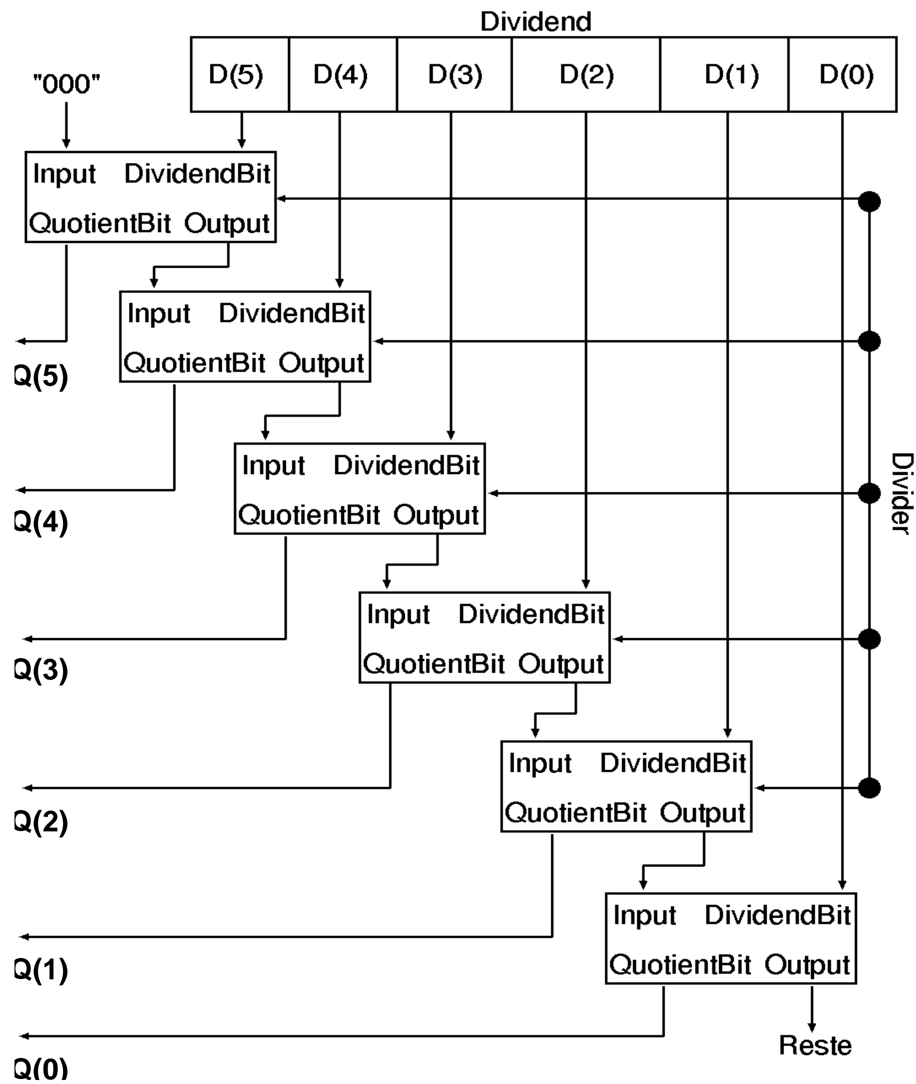
Commentaires :

1. L'entrée Input doit contenir 3 bits et non pas 2. En effet, Input représente un reste de la division partielle de l'etage précédent. Il ne peut donc représenter au maximum la valeur (diviseur — 1), qui requiert le même nombre de bits que le diviseur.

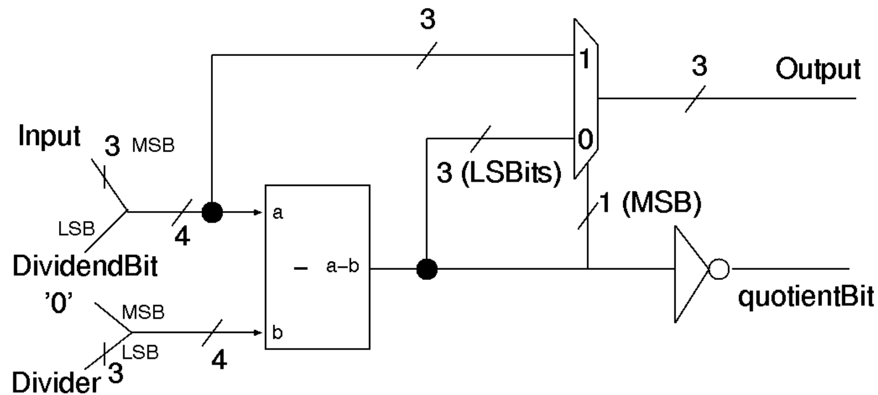
2. Le circuit a les sorties :
   - QuotientBit—le bit du quotient qui est obtenue à l'itération en question, et,
   - Output—3 bits de reste utilisés lors de l'itération suivante, ou, pour obtenir le reste final, lors de la dernière itération.

b)
Le quotient compte 6 bits (considérez la division par 1). Le reste est codé sur 3 bits (voir le commentaire sur la précision de Input). 6 copies du circuit DividerBlock sont nécéssaires, puisque que chacune fournit un bit du quotient.

La figure ci-dessous montre comment connecter les 6 répliques du circuit DividerBlock.



c)
Le schéma à blocs du circuit DividerBlock est donné ci-dessous.



Notez bien que le bit de poids fort après soustraction (équivalent à un bit de signe en complément à deux) permet d'obtenir directement de bit de quotient par inversion : si le résultat de la soustraction est négatif, le bit du quotient est nul.

d)
Le circuit est complètement combinatoire.

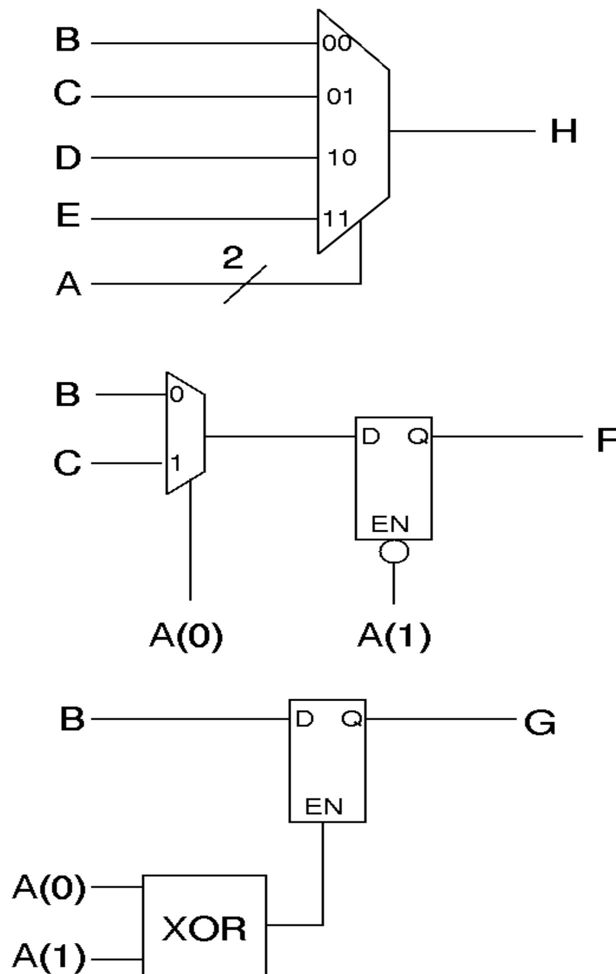
```
process(Input, DividendBit, Divider)
 signal sub : std_logic_vector(3 downto 0);
begin

 sub  <=  (input & DividendBit) - ('0' & divider);
 quotientBit  <= not sub(3);
 if sub(3) = 0 then
  Output <= sub(2 downto 0);
 else
  Output <= Input(1 downto 0) & DividendBit;
 end if;
end process;
```

a. Dessinez le circuit correspondant au code VHDL suivant :

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

entity test is
  port (
    A         : in  std_logic_vector(1 downto0);
    B, C, D, E : in  std_logic;
    F, G, H    : out std_logic);
end test;

architecture synthesizable of test is
begin
  process (A, B, C, D, E)
  begin
    case A is
      when "00" => F <= B; H <= B;
      when "01" => F <= C; G <= B; H <= C;
      when "10" => G <= B; H <= D;
      when "11" => H <= E;
      when others => null;
    end case;
  end process;
end synthesizable;
```

b. Pour chaque composant du circuit dessiné, expliquez en une dizaine de mots pourquoi le code donné lui correspond. Ecrivez une autre version de ce circuit en VHDL, en utilisant un processus séparé pour chaque composant.

a)
Le circuit représenté par ce programme contient les éléments suivants :



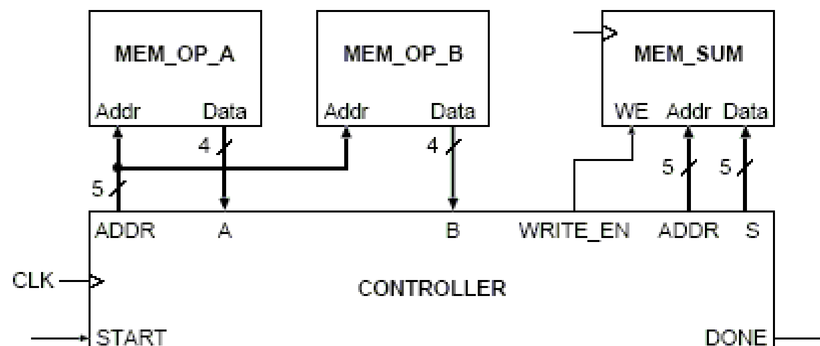
b)
Le signal H est issu d'un multiplexeur a 4 entrées.
Le signal F est mis à jour seulement lorsque A(1) vaut 0, et, prend alors une valeur dépendent de A(0). Quant au signal G, il est seulement mis a jour lorsque A(0) et A(1) sont différents.

Le code VHDL suivant décrit le même circuit :

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

entity test is
  port (
    A : in  std_logic_vector(1 downto 0);
    B, C, D, E : in  std_logic;
```

```
    F, G, H    : out std_logic);
end test;

architecture synthesizable of test is
signal Fmux,ADiff : std_logic;
begin

 ADiff <= A(0) xor A(1);

 mux0: process (A, B, C, D, E)
  begin
    case A is
      when "00" => H <= B;
      when "01" => H <= C;
      when "10" => H <= D;
      when "11" => H <= E;
      when others => null;
    end case;
  end process;

mux1: process (A, B, C)
  begin
    if A(0) = '0' then
      Fmux <= B;
    else
      Fmux <= C;
    end if;
  end process;


latch0: process (A, Fmux)
  begin
    if A(1) = '0' then
       F <= Fmux;
    end if;
  end process;

latch1 : process (ADiff, B)
  begin
    if ADiff = '1' then
     G <= B;
    end if;
  end process;

end synthesizable;
```

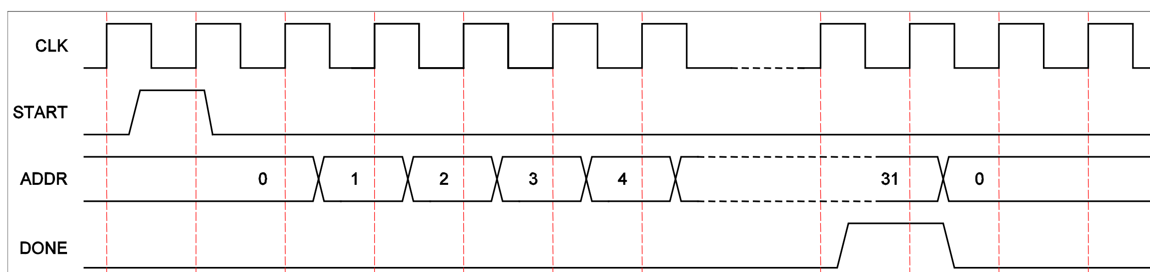
Considérez le système décrit dans la figure suivante :



A l'activation du signal **START**, on effectue à chaque cycle les tâches suivantes :

1) Lecture des opérandes **A** et **B** à une adresse des mémoires **MEM_OP_A** et **MEM_OP_B** respectivement.
2) Addition des opérandes lus **A** et **B** pour produire la somme **S**.
3) Ecriture de la somme **S** dans la mémoire **MEM_SUM** à la même adresse.
4) Incrémentation de l'adresse.

La séquence commence à l'adresse 0 et se poursuit jusqu'à l'adresse 31, c'est-à-dire lorsque les 32 opérandes contenus dans les mémoires **MEM_OP_A** et **MEM_OP_B** ont été additionnés et les 32 sommes écrites dans **MEM_SUM**. Lors de la dernière itération, le signal **DONE** est actif pendant un cycle. Les signaux de contrôle **ADDR**, **WRITE_EN** et **DONE** sont générés par un contrôleur. L'activation du signal **START** est ignorée lorsque le signal **ADDR** est différent de 0. Le chronogramme suivant illustre le fonctionnement du système :



a. Soit T_{read} le temps d'accès en lecture aux mémoires **MEM_OP_A** et **MEM_OP_B**. Soit T_{add} le temps nécessaire pour effectuer une addition. Soit T_{write} l'intervalle entre (a) le moment où la valeur à écrire doit être présente à l'entrée de la mémoire **MEM_SUM** et (b) le flanc montant de l'horloge. Soit T_{cycle} la période de l'horloge du système. Donnez une relation entre ces paramètres afin que le système fonctionne correctement. Que pourrait-on faire si cette relation n'était pas vérifiée?

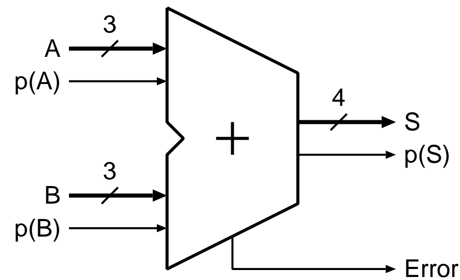
b. Décrivez le fonctionnement du contrôleur par une machine à états finis. Dessinez le diagramme des états. S'agit-il d'une machine de Mealy ou Moore ?

c. A partir de la machine à états finis, décrivez le contrôleur du système en VHDL.

d. Il se peut que l'additionneur commette parfois des erreurs. On utilise donc un additionneur capable de détecter certaines de ces erreurs. En plus des entrées et sorties ordinaires, il a un signal de sortie **ERROR** qui indique si le calcul réalisé est erroné. On veut modifier le système décrit au début pour prendre en compte ce signal d'erreur. Lorsque le signal **ERROR** est actif, le contrôleur maintient les adresses inchangées jusqu'à ce que l'additionneur ait corrigé l'erreur, ce qui désactive donc **ERROR**. La somme est alors écrite normalement dans **MEM_SUM**. Modifiez le contrôleur du système (graphe des états et VHDL) pour implémenter cette correction d'erreurs.

e. On veut maintenant construire cet additionneur qui détecte des erreurs. Pour cela on note p(X) la parité d'un mot de *n*-bits: $p(X) = X_0 \text{ xor } X_1 \text{ xor } X_2 \text{ xor } \ldots X_{n-1}$. On appelle C le mot composé par les retenues (*carries*) internes à un additionneur à retenue propagée (*ripple-carry adder*) construit à partir d'additionneurs complets (*full adders*). La détection d'erreur se base sur le fait que l'addition préserve la relation de parité suivante: p(A) xor p(B) = p(C) xor p(S).

A partir d’un additionneur *ripple carry* 3-bit, dessinez le schéma à blocs interne d'un additionneur utilisant cette relation de parité et dont l’interface externe est la suivante :



Comme indiqué dans la figure, l’additionneur reçoit les signaux de parité déjà calculés et doit produire le signal de parité pour la somme.

a)

Puisque les opérations 1), 2), 3) et 4) sont effectuées durant le même cycle, il faut que T_{read} $+T_{add}$ + T_{write} < T_{cycle} . Si cela n'est pas le cas,une solution qui ne change pas les caractéristiques des mémoires et de l'additionneur est d'introduire un registre entre les mémoires d'opérandes et l'additionneur. Cette modification implique deux conditions : T_{read} < T_{cycle} et T_{add} + T_{write} < $T_{cycle.}$. qui sont plus facilement vérifiées que la première.

b)

Pour contrôler le système, on peut utiliser un compteur de 5 bits avec « enable ».La machine à états fini correspondante est décrite dans la Figure ci-dessous.

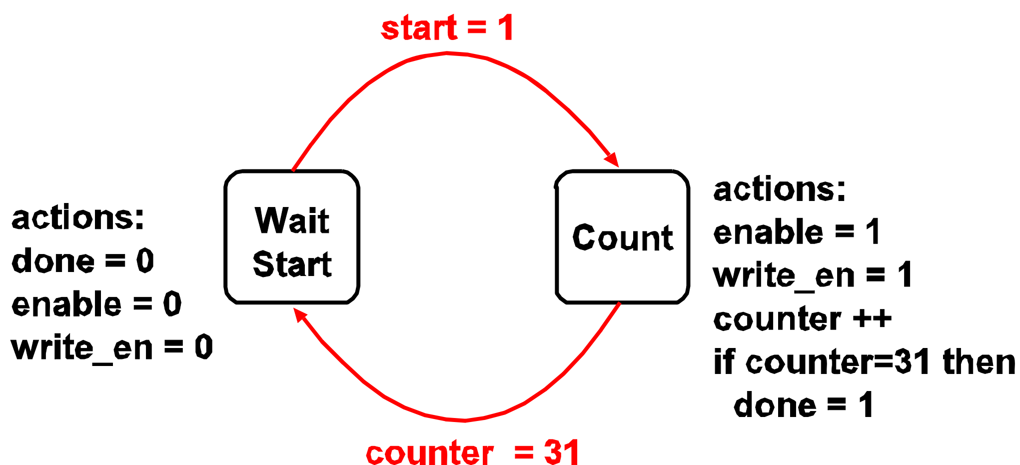
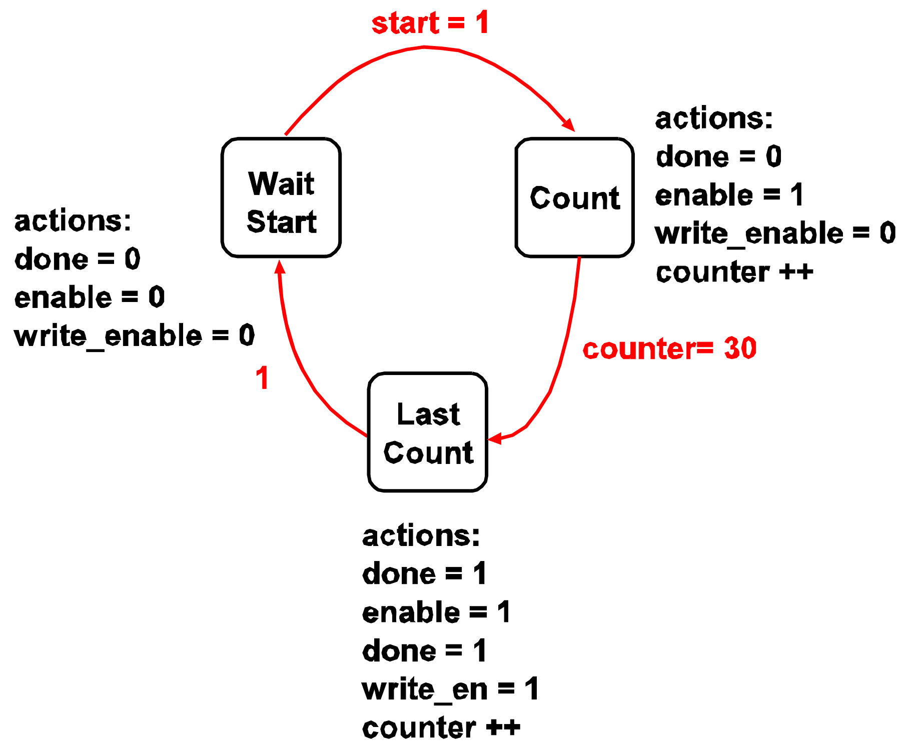


Figure 1

L'état du compteur est « counter ». Il s'agit d'une machine de Mealy parce que le signal DONE prends les valeurs 0 et 1 dans l'état « Count ».
Il est possible de décrire le contôleur par une machine de Moore, comme fait dans la figure ci-dessous :


start = 1
Wait Start
Count
actions:
done = 0
enable = 1
write_enable = 0
counter ++
actions:
done = 0
enable = 0
write_enable = 0
counter= 30
1
Last Count
actions:
done = 1
enable = 1
done = 1
write_en = 1
counter ++

c)

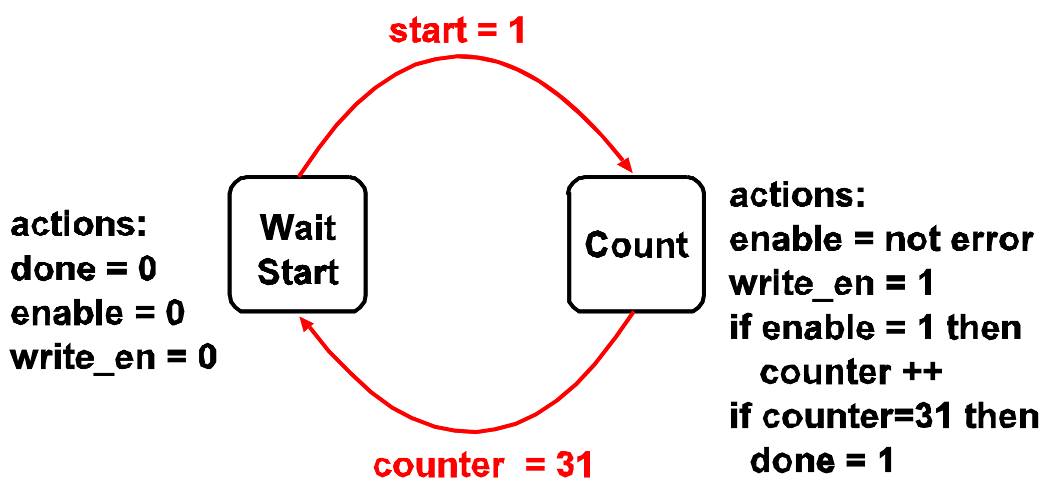
Le VHDL décrivant le système complet avec une machine d'état comme celle de la figure 1 est :

```
entity  MEM_ADDR_MEM(
clk : in std_logic;
start : in std_logic;
addr : in std_logic_vector(4 downto 0);
A : in std_logic;
B : in std_logic;
S: out std_logic;
write_en : std_logic;
done: out std_logic);

architecture synth of MEM_ADDR_MEM is

type state_t is (WaitStart, Count) ;
signal state, next_state : state_t;
signal counter, next_counter: std_logic_vector(4
downto 0);
signal enable: std_logic;



begin

S <= A + B;
addr <= counter;
write_en <= enable;

reg : process(clk)
begin
     if clk'event and clk='1' then
          state <= next_state;
          counter <= next_counter;
     end if
end process reg;

comb_count:  process(counter, enable)
begin
     next_counter <=counter;
     if enable = '1' then
          next_counter <= counter + 1;
     end if;
```

```
end process comb_count;

enable <='1' when state = Count else '0';

comb_state:  process(counter, state, start)
begin
     next_state <=state;
     if start = '1' and state = WaitStart then
          next_state <= Count;
     elsif counter = "11111" then
          next_state <= WaitStart;
     end if;
end process comb_state;
comb_out:  process(counter,)
begin
     done <='0';
     if counter = "11111" then
          done <= '1';
     end if;
end process comb_out;

end architecture synth;
```

d)

Dans ce cas, il suffit de modifier le signal « enable » du compteur :

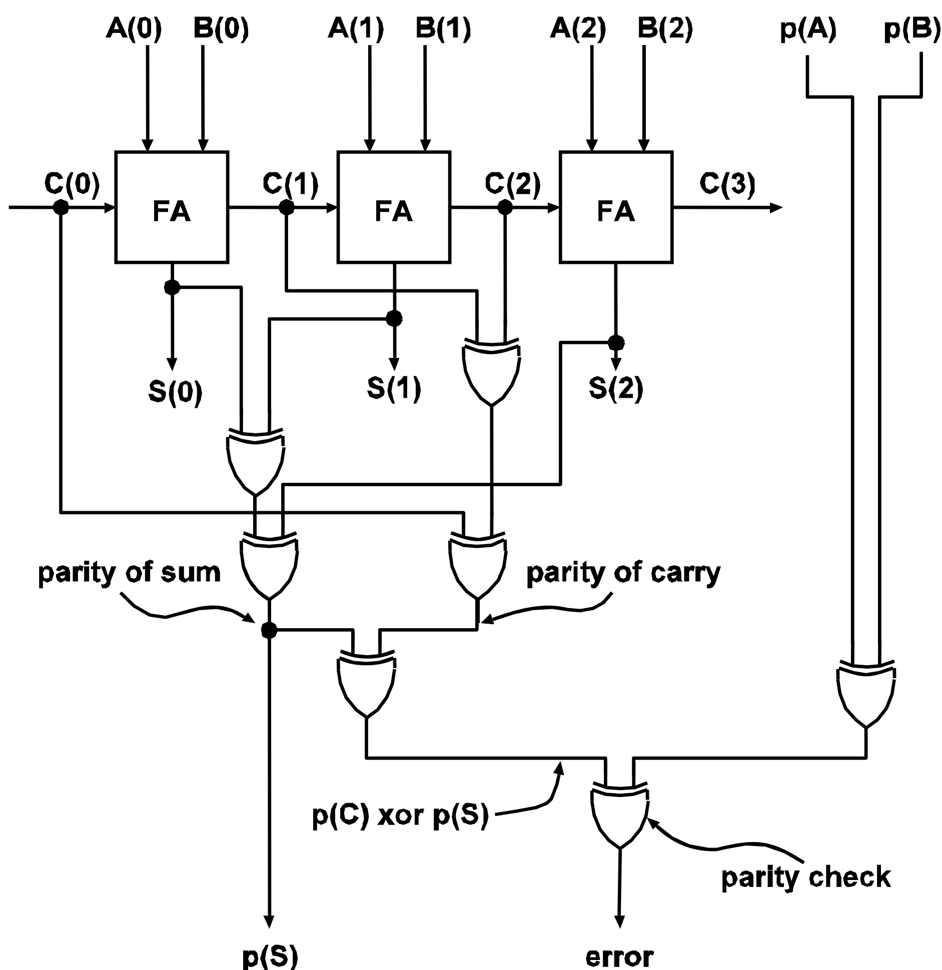


Le VDHL correspondant est identique à celui de la question c), sauf :

enable <='1' when (state = Count and error = '0') else '0';

e)

Comme indiqué dans la figure ci-dessous, la valeur binaire *p(A)xor p(B)* est comparée à la valeur binaire *p(C)xor p(S)*.



Comme la relation de l'énoncé ne précise pas comment les valeurs binaires p(C) et p(S) sont obtenues, d'autres solutions incluant plus de bits dans le calcul des parité sont considérés correctes

Considérez l'extrait de code VHDL suivant:

```
signal A, B, C, D: std_logic;
signal s1, s2, s3: std_logic;

...

process(A, B, C, s1, s2, s3)

begin

  if  s3 = '1' then
    s1 <= C;
  else
    s1 <= A;
  end if;

  if s2 = '0' then
    D <= A;
  else
     D <= s1;
  end if;

  s2 <= A xor B;
  s3 <= A xor C;

end process;
```

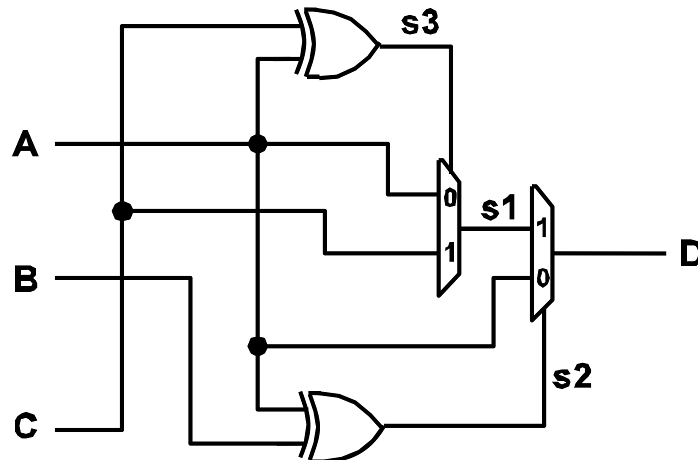
a. Dessinez le circuit décrit par ce code. Est-il séquentiel ou combinatoire? Justifiez votre réponse de façon concise.

b. Décrivez dans une table la valeur de D en fonction des 8 combinaisons possibles pour les signaux A, B, et C.

c. Ecrivez en VHDL le code d'un *process* décrivant le même circuit en utilisant des variables pour s1, s2 et s3.

a)

Le circuit correspondant est dessiné dans la figure ci-dessous :



Ce circuit est combinatoire, puisque (a) tout changement des signaux A, B, ou C provoque l'évalution des signaux s1, s2, et s3, et (b) à son tour, tout changement des signaux s1, s2, ou s3 provoque l'évaluation du signal D. Par conséquent, dépend uniquement des entrées A, B, et C, i.e., le circuit est combinatoire.

Remarque : l'omission des signaux s1, s2 et s3 dans la liste de sensitivité du process conduit à une situation où, probablement, le circuit simulé comporte des latch, et le circuit synthétisé est combinatoire car le synthétiseur ne prend pas en compte la liste de sensitivité telle que figurant dans le fichier VHDL. C'est donc une situation à éviter.

b)

Puisque le circuit est combinatoire, on peut écrire la table ci-dessous :

A	*B*	*C*	*D*
0	*0*	*0*	*0*
0	*0*	*1*	*0*
0	*1*	*0*	*0*
0	*1*	*1*	*1*
1	*0*	*0*	*0*
1	*0*	*1*	*1*
1	*1*	*0*	*1*
1	*1*	*1*	*1*

c)

```
signal A, B, C, D: std_logic;

...

process(A, B, C, s1, s2, s3)

variable s1, s2, s3: std_logic;

begin

s2 := A xor B;
s3 := A xor C;

  if  s3 = '1' then
    s1 := C;
  else
    s1 := A;
  end if;



  if s2 = '0' then
    D <= A;
  else
     D <= s1;
  end if;

end process;
```

Il est nécessaire d’assigner les variables pour s1, s2 et s3 avant leur utilisations lors des tests et/ou affectations. Autrement, la variable garde la dernière valeur qui lui a été assignée et se comporte donc comme élément de mémoire.

L'algorithme CORDIC (« *COordinate Rotation DIgital Computer* ») permet de calculer les fonctions trigonométriques en utilisant les formules itératives :

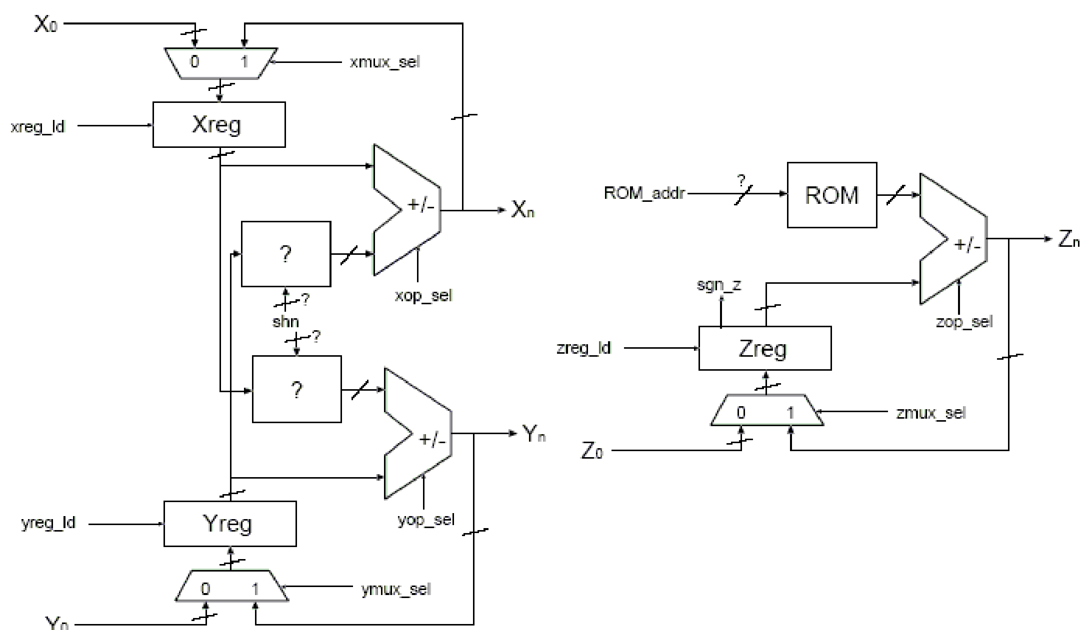
$$x_{i+1} = x_i - d_i \cdot y_i \cdot 2^{-i},$$
$$y_{i+1} = y_i + d_i \cdot x_i \cdot 2^{-i},$$
$$z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-i}),$$

où $d_i = \begin{cases} -1 & si \quad z_i < 0 \\ 1 & autrement \end{cases}$ et ($x_0 = X$, $y_0 = 0$ et $z_0 = \varphi$) sont les constantes d'initialisation.

Après n itérations, on peut écrire les valeurs de x_n et y_n ainsi :

$$x_n = A_n \cdot x_0 \cos(\varphi) \quad \text{et} \quad y_n = A_n \cdot x_0 \sin(\varphi).$$

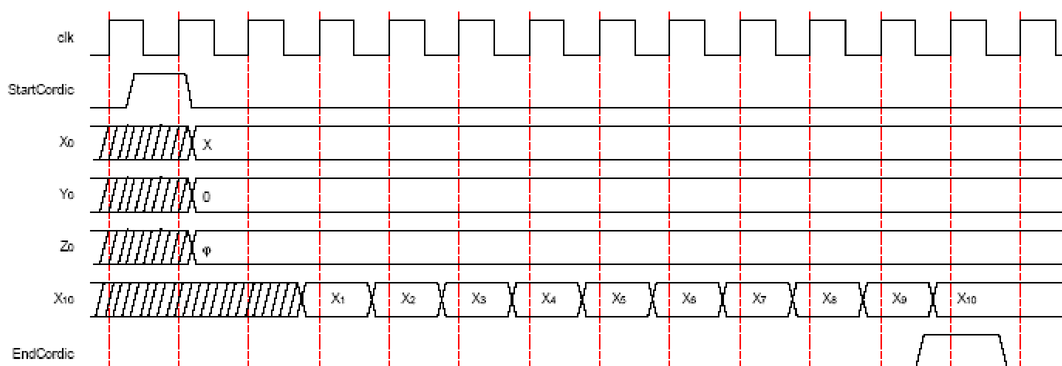
On utilise ces formules pour calculer les fonctions trigonométriques d'un angle φ. Le schéma du système qui réalise cette fonctionnalité est donné dans la figure suivante :



On effectue 10 itérations pour calculer les fonctions trigonométriques de nombres signés représentés en complément à deux sur 11 bits. Les entrées **X0**, **Y0** et **Z0** servent à charger avant le début du calcul les registres avec leur constantes d'initialisation (vous ne devez pas vous soucier du choix de ces valeurs initiales). La mémoire ROM contient les coefficients $\tan^{-1}(2^{-i})$ déjà calculés (à l'adresse 0 de la ROM on trouve le coefficient de l'itération 0, à l'adresse 1 le coefficient de l'itération 1, etc.). Les signaux de contrôle **xop_sel**, **yop_sel** et **zop_sel** déterminent l'opération de l'ALU : si le signal de contrôle est zéro, l'ALU réalise une addition, autrement une soustraction. Les multiplexeurs sont contrôlés par les signaux **xmux_sel**, **ymux_sel** et **zmux_sel** comme indiqué dans la figure. Le bit de poids fort du registre **Zreg** est connecté au signal **sgn_z**, indiquant donc le signe du nombre qui y est mémorisé.

a. Quelle est la fonction des deux blocs inconnus identifiés par un point d'interrogation sur le schéma ? Décrivez-les en VHDL. Quelle est la taille de la mémoire ROM ? Quelle est la taille en bits des signaux **ROM_addr** et **shn** ?

b. Quels sont les signaux nécessaires en entrée du contrôleur pour opérer ce circuit ? Quelles sont ses sorties ? Supposez que le signal **StartCordic** indique au contrôleur le début de l'exécution et que le signal **EndCordic** est généré par le contrôleur et indique la fin d'exécution et la validité du résultat. Ceci est résumé sur le chronogramme suivant :



c. Dessinez le diagramme d'état de la machine à états finis (FSM) en charge de contrôler l'exécution de l'algorithme.

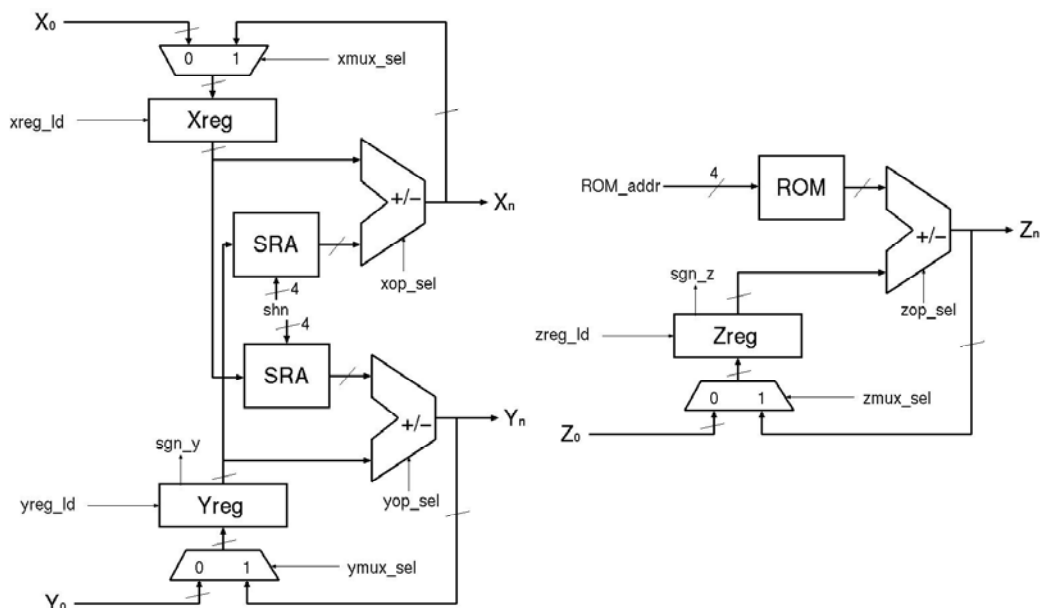
d. Ecrivez le code VHDL du contrôleur complet.

e. Est-il utile de continuer le calcul après le 10ème cycle pour augmenter la précision du résultat ? Justifiez votre réponse.

f. Le système proposé ne permet qu'un calcul à la fois : pour commencer une nouvelle série d'itérations, on doit attendre que la série précédente soit terminée. En supposant que vous avez un nombre d'ALUs illimité à disposition, suggérez le schéma d'un système « déroulé » permettant de produire des nouveaux résultats à chaque cycle d'horloge.

a)

Les deux blocs inconnus permettent de calculer $x_i \cdot 2^{-i}$ et $y_i \cdot 2^{-i}$ dans les formules itératives. Comme il s'agit de division avec des puissances de deux, il suffit d'avoir des blocs qui font le décalage arithmétique à droite. Donc, les deux blocs implémentent la fonctionnalité « Shift Right Arithmetic » (SRA montrés dans la figure ci-dessous). La valeur à l'entrée du bloc est décalée arithmétiquement à droite par shn (de 0 à 10, donc, il nous faut 4 bits pour le signal) positions. La mémoire ROM doit stocker 10 coefficients, sa taille est, donc, 16 mots dont 10 sont utilisés. Pour adresser une mémoire de cette taille, il nous faut un signal de 4 bits (ROM_addr dans la figure). Le code VHDL de SRA est donné ensuite.

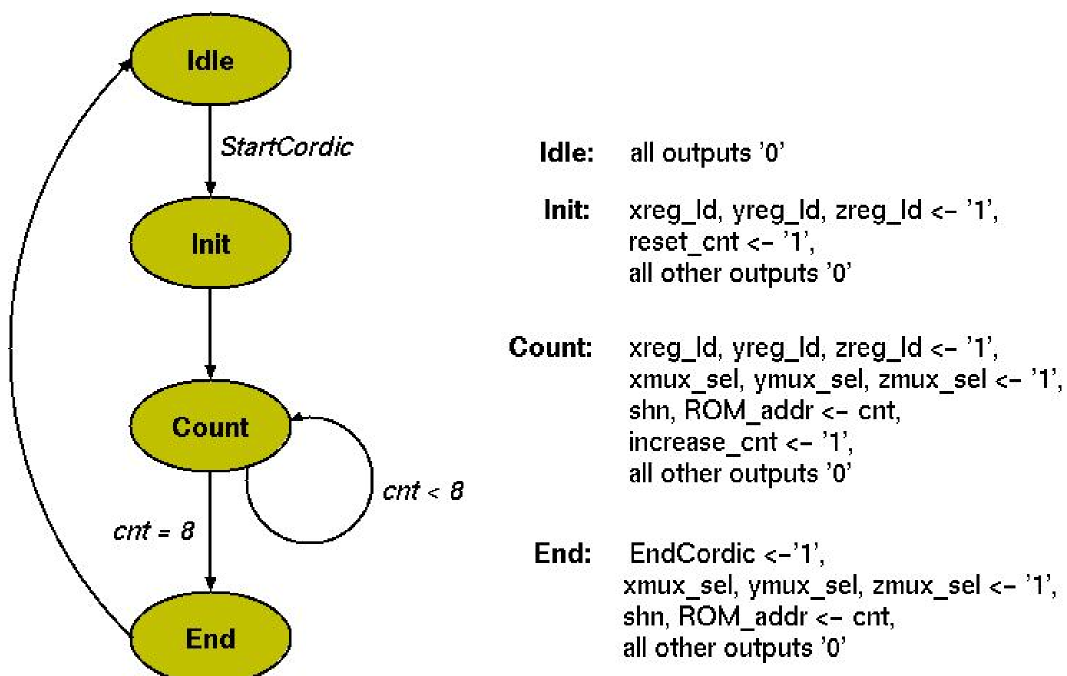



```
entity SRA is
  port(
    XY : in std_logic_vector(10 downto 0);
    shn : in std_logic_vector(3 downto 0);
    S: out std_logic_vector (10 downto 0);
  )
end entity SRA;
```

```
architecture SRA_behav of SRA is
  begin
    shift: process(shn, XY)
     variable num: integer;
    begin
      num := unsigned(shn)
      S<=(10 downto 10-num=>XY(10)) & XY(9 downto num);
    end process shift;
end architecture SRA_behav;
```

b)
Le signal en entrée du contrôleur est **StartCordic**. Les sorties sont tous les signaux nécessaires pour opérer ce circuit (**xreg_ld**, **yreg_ld**, **zreg_ld**, **xmux_sel**, **ymux_sel**, **zmux_sel**, **shn**, **ROM_addr**), plus le signal **EndCordic**. Les signaux **xop_sel**, **yop_sel**, **zop_sel** dépendent seulement du signal **sgn_z** et ne sont pas sorties du contrôleur.

c)
La figure ci-dessous montre le diagramme d'état de la machine à états finis (FSM) en charge de contrôleur. Les sorties de la machine dépendent de son état, comme la figure le montre.



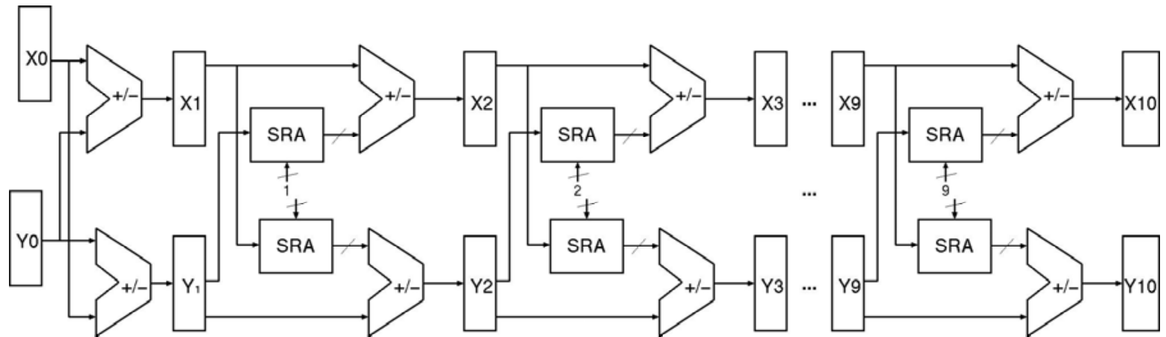
d)
Le code VHDL du contrôleur est donné ci-dessous :

```
entity CORDIC_FSM is
  port(
    clk : in std_logic;
    reset : in std_logic;
    StartCordic : in std_logic;
    EndCordic : out std_logic;
    xreg_ld, yreg_ld, zreg_ld : out std_logic;
         xmux_sel, ymux_sel, zmux_sel : out std_logic;
         ROM_addr : out std_logic_vector (3 downto 0);
         shn : out std_logic_vector (3 downto 0);
  )
end entity CORDIC_FSM;

architecture FSM of CORDIC_FSM is
   type fsm_states is (Idle, Init, Count, End);
   signal fsm_state, next_state : fsm_states;
   signal cnt : integer;
begin
  fsm_change_state: process(clk, reset)
  begin
    if (reset = '1') then
      fsm_state <= Idle;
    elseif (clk'event and clk = '1') then
      fsm_state <= next_state;
      if (reset_cnt = '1') then
         cnt <= 0;
      end if;
      if (increase_cnt = '1') then
         cnt <= cnt + 1;
      end if;
    end if;
  end process fsm_change_state;

  fsm_compute_state: process(StartCordic, fsm_state,
 cnt)
  begin
    -- default values
    ROM_addr <= (others => '0');
    shn <= (others => '0');
    xreg_ld <= '0'; yreg_ld <= '0'; zreg_ld <= '0';
    xmul_sel <= '0'; ymul_sel <= '0'; zmul_sel <= '0';
    EndCordic <= '0';
    next_state <= fsm_state;
```

```
    -- comput state
    case fsm_state is
      when Idle     =>
        if (StartCordic = '1') then
          next_state <= Init;
        end if;
      when Init   =>
        xreg_ld <= '1'; yreg_ld <= '1'; zreg_ld <= '1';
        reset_cnt <= '1';
        next_state <= Count;
      when Count   =>
        xreg_ld <= '1'; yreg_ld <= '1'; zreg_ld <= '1';
        xmul_sel <= '1'; ymul_sel <= '1'; zmul_sel<='1';
        increase_cnt <= '1';
        ROM_addr <= stdlogic(cnt,4);
        Shn <= stdlogic(cnt,4);
   if (cnt >= 8) then
           next_state <= End;
        end if;
      when End      =>
        EndCordic <= '1';
        xmul_sel <= '1'; ymul_sel <= '1'; zmul_sel<='1';
        ROM_addr <= stdlogic(cnt,4);
        Shn <= stdlogic(cnt,4);
        next_state <= Idle;
    end case;
  end process fsm_compute_state;
end architecture CORDIC_FSM;
```

e) Il n'est pas utile de continuer le calcul après le 10ème cycle car la précision de nombres utilisés est sur 11 bit. La poursuite des itérations produit seulement des oscillations au tour du point de convergence.

f) On pourrait imaginer un circuit « déroulé » qui permet de commencer un nouveau calcul à chaque cycle d'horloge. En déroulant le circuit, on multiplie les ressources utilisées. Entre chaque deux stages, il faut insérer des registres. A travers ces registres, les valeurs intermédiaires d'un calcul peuvent avancer vers la sortie, en devenant de plus à plus finales. On appelle ce type de transformation « pipelining » car la façon de traitement ressemble à un « pipe ». La figure ci-dessous donne une idée de « pipelining » du circuit CORDIC.

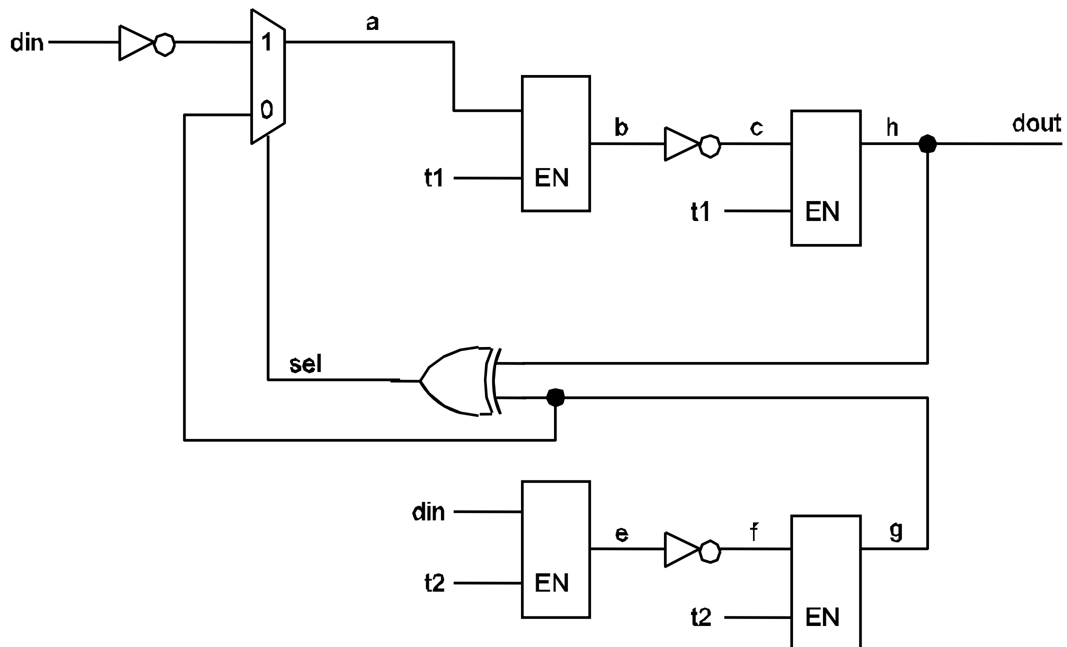

X0
Y0
+/–
X1
Y1
SRA
1
X2
Y2
2
X3
Y3
...
X9
Y9
9
X10
Y10

Considérez le circuit décrit par le VHDL ci-dessous :

```
ENTITY ent IS
     PORT (din  : IN  STD_LOGIC;
           t1   : IN  STD_LOGIC;
           t2   : IN  STD_LOGIC;
           dout : OUT STD_LOGIC);
END ent;

ARCHITECTURE arch OF ent IS
     a, b, c, d, e, f, g, h, sel : STD_LOGIC;
BEGIN
     dout <= h;
     d <= NOT din;
     c <= NOT b;
     f <= NOT e;
     sel <= g XOR h;
     a <= d WHEN sel = '1' ELSE g;

     p1 : PROCESS (t1, a, c)
     BEGIN
          IF t1 ='1' THEN
               b <= a;
          ELSE
               h <= c;
          END IF;
     END PROCESS p1;

     p2 : PROCESS (t2, din, f)
     BEGIN
          IF t2 ='1' then
                e <= din;
          ELSE
                g <= f;
          END IF;
     END PROCESS p2;
END ARCHITECTURE arch;
```

a. Dessinez le schéma logique du circuit décrit par ce code.
b. Ce circuit est-il combinatoire ou séquentiel? Justifiez votre réponse de la façon la plus précise et la plus concise possible.
c. Le circuit entre les signaux **din**, **t2** et **g** représente, à une petite exception près, un composant très fréquemment employé. Lequel ? Dessinez un circuit équivalent à cette partie du schéma en utilisant ce composant.

a)

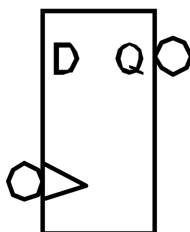


b)

*Le circuit est **séquentiel** puisqu'il contient des* latches*. Autrement dit, ces éléments de mémoire font que la sortie ne dépend pas uniquement que des entrées.*

c)

Un flip flop est constitué de 2 latche*s interconnectées par un inverseur et dont les* enable *ont des polarités opposées. On peut donc dessiner le circuit équivalent suivant.*



Il s'agit d'un flip-flop, actif au flanc descendant de l'horloge, dont la sortie est l'inverse de l'entrée.